

# 第 1 章

## .NET 体系结构

我们不能孤立地使用 C#语言，而必须和 .NET Framework 一起考虑。C#编译器专门用于 .NET，这表示用 C#编写的所有代码总是在 .NET Framework 中运行。对于 C#语言来说，可以得出两个重要的结论：

(1) C#的结构和方法论反映了 .NET 基础方法论。

(2) 在许多情况下，C#的特定语言功能取决于 .NET 的功能，或依赖于 .NET 基类。

由于这种依赖性，在开始使用 C#编程前，了解 .NET 的结构和方法论就非常重要了，这就是本章的目的。下面是本章的内容：

- 本章首先了解在 .NET 编译和运行所有的代码(包括 C#)时通常会出现什么情况。
- 对这些内容进行概述之后，就要详细阐述 Microsoft 中间语言(Microsoft Intermediate Language, MSIL 或简称为 IL)，.NET 上所有编译好的代码都要使用这种语言。本章特别要介绍 IL、通用类型系统(Common Type System, CTS)及公共语言规范(Common Language Specification, CLS)如何提供 .NET 语言之间的互操作性。最后解释各种语言如何使用 .NET，包括 Visual Basic 和 C++。
- 之后，我们将介绍 .NET 的其他特性，包括程序集、命名空间和 .NET 基类。
- 最后本章简要探讨一下 C#开发人员可以创建的应用程序类型。

### 1.1 C#与.NET 的关系

C#是一种相当新的编程语言，C#的重要性体现在以下两个方面：

- 它是专门为与 Microsoft 的 .NET Framework 一起使用而设计的。( .NET Framework 是一个功能非常丰富的平台，可开发、部署和执行分布式应用程序)。
- 它是一种基于现代面向对象设计方法的语言，在设计它时，Microsoft 还吸取了其他类似语言的经验，这些语言是近 20 年来面向对象规则得到广泛应用后才开发出来的。

有一个很重要的问题要弄明白：C#就其本身而言只是一种语言，尽管它是用于生成面向 .NET 环境的代码，但它本身不是 .NET 的一部分。 .NET 支持的一些特性，C#并不支持。而 C#语言支持的另一些特性，.NET 却不支持(例如运算符重载)！

但是，因为 C#语言是和 .NET 一起使用的，所以如果要使用 C#高效地开发应用程序，理解 Framework 就非常重要，所以本章将介绍 .NET 的内涵。



## 1.2 公共语言运行库

.NET Framework 的核心是其运行库的执行环境，称为公共语言运行库(CLR)或.NET 运行库。通常将在 CLR 的控制下运行的代码称为托管代码(managed code)。

但是，在 CLR 执行编写好的源代码之前，需要编译它们(在 C#中或其他语言中)。在.NET 中，编译分为两个阶段：

- (1) 把源代码编译为 Microsoft 中间语言(IL)。
- (2) CLR 把 IL 编译为平台专用的代码。

这个两阶段的编译过程非常重要，因为 Microsoft 中间语言(托管代码)是提供.NET 的许多优点的关键。

Microsoft 中间语言与 Java 字节代码共享一种理念：它们都是低级语言，语法很简单(使用数字代码，而不是文本代码)，可以非常快速地转换为内部机器码。对于代码来说，这种精心设计的通用语法有很重要的优点：平台无关性、提高性能和语言的互操作性。

### 1.2.1 平台无关性

首先，这意味着包含字节代码指令的同一文件可以放在任一平台中，运行时编译过程的最后阶段可以很容易完成，这样代码就可以运行在特定的平台上。换言之，编译为中间语言就可以获得.NET 平台无关性，这与编译为 Java 字节代码就会得到 Java 平台无关性是一样的。

注意.NET 的平台无关性目前只是一种可能，因为在编写本书时，.NET 只能用于 Windows 平台，但人们正在积极准备，使它可以用于其他平台(参见 Mono 项目，它用于实现.NET 的开放源代码，参见 <http://www.go-mono.com/>)。

### 1.2.2 提高性能

前面把 IL 和 Java 做了比较，实际上，IL 比 Java 字节代码的作用还要大。IL 总是即时编译的(称为 JIT 编译)，而 Java 字节代码常常是解释性的，Java 的一个缺点是，在运行应用程序时，把 Java 字节代码转换为内部可执行代码的过程会导致性能的损失(但在最近，Java 在某些平台上能进行 JIT 编译)。

JIT 编译器并不是把整个应用程序一次编译完(这样会有很长的启动时间)，而是只编译它调用的那部分代码(这是其名称由来)。代码编译过一次后，得到的内部可执行代码就存储起来，直到退出该应用程序为止，这样在下次运行这部分代码时，就不需要重新编译了。Microsoft 认为这个过程要比一开始就编译整个应用程序代码的效率高得多，因为任何应用程序的大部分代码实际上并不是在每次运行过程中都执行。使用 JIT 编译器，从来都不会编译这种代码。

这解释了为什么托管 IL 代码的执行几乎和内部机器代码的执行速度一样快，但是并没有说明为什么 Microsoft 认为这会提高性能。其原因是编译过程的最后一部分是在运行时进行的，JIT 编译器确切地知道程序运行在什么类型的处理器上，可以利用该处理器提供的任何特性或特定的机器代码指令来优化最后的可执行代码。

传统的编译器会优化代码，但它们的优化过程是独立于代码所运行的特定处理器的。这是

因为传统的编译器是在发布软件之前编译为内部机器可执行的代码。即编译器不知道代码所运行的处理器类型，例如该处理器是 x86 兼容处理器还是 Alpha 处理器，这超出了基本操作的范围。例如 Visual Studio 6 为一般的奔腾机器进行了优化，所以它生成的代码就不能利用奔腾 III 处理器的硬件特性。相反，JIT 编译器不仅可以进行 Visual Studio 6 所能完成的优化工作，还可以优化代码所运行的特定处理器。

### 1.2.3 语言的互操作性

使用 IL 不仅支持平台无关性，还支持语言的互操作性。简而言之，就是能将任何一种语言编译为中间代码，编译好的代码可以与从其他语言编译过来的代码进行交互操作。

那么除了 C# 之外，还有什么语言可以通过 .NET 进行交互操作呢？下面就简要讨论其他常见语言如何与 .NET 交互操作。

#### 1. Visual Basic 2008

Visual Basic 6 在升级到 Visual Basic .NET 2002 时，经历了一番脱胎换骨的变化，才集成到 .NET Framework 的第一版中。Visual Basic 语言对 Visual Basic 6 进行了很大的演化，也就是说，Visual Basic 6 并不适合运行 .NET 程序。例如，它与 COM 的高度集成，且只把事件处理程序作为源代码显示给开发人员，大多数后台代码不能用作源代码。另外，它不支持继承，Visual Basic 使用的标准数据类型也与 .NET 不兼容。

Visual Basic 6 在 2002 年升级为 Visual Basic .NET，对 Visual Basic 进行的改变非常大，完全可以把 Visual Basic 当作是一种新语言。现有的 Visual Basic 6 代码不能编译为 Visual Basic 2008 代码(或 Visual Basic .NET 2002、2003 和 2005 代码)，把 Visual Basic 6 程序转换为 Visual Basic 2008 时，需要对代码进行大量的改动，但大多数修改工作都可以由 Visual Studio 2008(Visual Studio 的升级版本，用于与 .NET 一起使用)自动完成。如果把 Visual Basic 6 项目读到 Visual Studio 2008 中，Visual Studio 2008 就会升级该项目，也就是说把 Visual Basic 6 源代码重写为 Visual Basic 2008 源代码。虽然这意味着其中的工作已大大减轻，但用户仍需要检查新的 Visual Basic 2008 代码，以确保项目仍可正确工作，因为这种转换并不十分完美。

这种语言升级的一个副作用是不能再把 Visual Basic 2008 编译为内部可执行代码了。Visual Basic 2008 只编译为中间语言，就像 C# 一样。如果需要使用 Visual Basic 6 编写程序，就可以这么做，但生成的可执行代码会完全忽略 .NET Framework，如果继续把 Visual Studio 作为开发环境，就需要安装 Visual Studio 6。

#### 2. Visual C++ 2008

Visual C++ 6 有许多 Microsoft 对 Windows 的特定扩展。通过 Visual C++ .NET，又加入了更多的扩展内容，来支持 .NET Framework。现有的 C++ 源代码会继续编译为内部可执行代码，不会有修改，但它会独立于 .NET 运行库运行。如果让 C++ 代码在 .NET Framework 中运行，就可以在代码的开头添加下述命令：

```
#using <mscorlib.dll>
```

还可以把标记/cil 传递给编译器, 这样编译器假定要编译托管代码, 因此会生成中间语言, 而不是内部机器码。C++的一个有趣的问题是在编译托管代码时, 编译器可以生成包含内嵌本机可执行代码的 IL。这表示在 C++代码中可以把托管类型和非托管类型合并起来, 因此托管 C++代码:

```
class MyClass
```

定义了一个普通的 C++类, 而代码:

```
ref class MyClass
```

生成了一个托管类, 就好像使用 C#或 Visual Basic 2008 编写类一样。实际上, 托管 C++比 C#更优越的一点是可以在托管 C++代码中调用非托管 C++类, 而不必采用 COM 交互功能。

如果在托管类型上试图使用.NET 不支持的特性(例如, 模板或类的多继承), 编译器就会出现一个错误。另外, 在使用托管类时, 还需要使用非标准的 C++特性。

因为 C++允许低级指针操作, C++编译器不能生成可以通过 CLR 内存类型安全测试的代码。如果 CLR 把代码标识为内存类型安全是非常重要的, 就需要用其他一些语言编写源代码, 例如 C# 或 Visual Basic 2008。

### 3. COM 和 COM+

从技术上讲, COM 和 COM+并不是面向.NET 的技术, 因为基于它们的组件不能编译为 IL(但如果原来的 COM 组件是用 C++编写的, 使用托管 C++, 在某种程度上可以这么做)。但是, COM+仍然是一个重要的工具, 因为其特性没有在.NET 中完全实现。另外, COM 组件仍可以使用——.NET 集成了 COM 的互操作性, 从而使托管代码可以调用 COM 组件, COM 组件也可以调用托管代码(见第 24 章)。在一般情况下, 把新组件编写为.NET 组件, 大多是为了方便, 因为这样可以利用.NET 基类和托管代码的其他优点。

## 1.3 中间语言

如前所述, Microsoft 中间语言显然在.NET Framework 中有非常重要的作用。C#开发人员应明白, C#代码在执行前要编译为中间语言(实际上, C#编译器仅编译为托管代码), 这是有意义的, 现在应详细讨论一下 IL 的主要特征, 因为面向.NET 的所有语言在逻辑上都需要支持 IL 的主要特征。

下面就是中间语言的主要特征:

- 面向对象和使用接口
- 值类型和引用类型之间的巨大差别
- 强数据类型
- 使用异常来处理错误
- 使用特性(attribute)

下面详细讨论这些特征。

### 1.3.1 面向对象和接口的支持

.NET 的语言无关性还有一些实际的限制。中间语言在设计时就打算实现某些特殊的编程方法，这表示面向它的语言必须与编程方法兼容，Microsoft 为 IL 选择的特定道路是传统的面向对象的编程，带有类的单一继承性。

注意：

不熟悉面向对象概念的读者应参考附录 B，获得更多的信息。

除了传统的面向对象编程外，中间语言还引入了接口的概念，它们显示了在带有 COM 的 Windows 下的第一个实现方式。.NET 接口与 COM 接口不同，它们不需要支持任何 COM 基础结构，例如，它们不是派生自 `IUnknown`，也没有对应的 GUID。但它们与 COM 接口共享下述理念：提供一个契约，实现给定接口的类必须提供该接口指定的方法和属性的实现方式。

前面介绍了使用 .NET 意味着要编译为中间语言，即需要使用传统的面向对象的方法来编程。但这并不能提供语言的互操作性。毕竟，C++ 和 Java 都使用相同的面向对象的范型，但它们仍不是可交互操作的语言。下面需要详细探讨一下语言互操作性的概念。

首先，需要确定一下语言互操作性的含义。毕竟，COM 允许以不同语言编写的组件一起工作，即可以调用彼此的方法。这就足够了吗？COM 是一个二进制标准，允许组件实例化其他组件，调用它们的方法或属性，而无须考虑编写相关组件的语言。但为了实现这个功能，每个对象都必须通过 COM 运行库来实例化，通过接口来访问。根据相关组件的线程模型，不同线程上内存空间和运行组件之间要编组数据，这还可能造成很大的性能损失。在极端情况下，组件保存为可执行文件，而不是 DLL 文件，还必须创建单独的进程来运行它们。重要的是组件要能与其他组件通信，但仅通过 COM 运行库进行通信。无论 COM 是用于允许使用不同语言的组件直接彼此通信，或者创建彼此的实例，系统都把 COM 作为中间件来处理。不仅如此，COM 结构还不允许利用继承实现，即它丧失了面向对象编程的许多优势。

一个相关的问题是，在调试时，仍必须单独调试用不同语言编写的组件。这样就不可能在调试器上调试不同语言的代码了。语言互操作性的真正含义是用一种语言编写的类应能直接与用另一种语言编写的类通信。特别是：

- 用一种语言编写的类应能继承用另一种语言编写的类。
- 一个类应能包含另一个类的实例，而不管它们是使用什么语言编写的。
- 一个对象应能直接调用用其他语言编写的另一个对象的方法。
- 对象(或对象的引用)应能在方法之间传递。
- 在不同的语言之间调用方法时，应能在调试器中调试这些方法调用，即调试不同语言编写的源代码。

这是一个雄心勃勃的目标，但令人惊讶的是，.NET 和中间语言已经实现了这个目标。在调试器上调试方法时，Visual Studio IDE 提供了这样的工具(不是 CLR 提供的)。

### 1.3.2 相异值类型和引用类型

与其他编程语言一样，中间语言提供了许多预定义的基本数据类型。它的一个特性是值类型和引用类型有明显的区别。对于值类型，变量直接保存其数据，而对于引用类型，变量仅保



存地址，对应的数据可以在该地址中找到。

在 C++ 中，引用类型类似于通过指针来访问变量，而在 Visual Basic 中，与引用类型最相似的是对象，Visual Basic 6 总是通过引用来访问对象。中间语言也有数据存储的规范：引用类型的实例总是存储在一个名为“托管堆”的内存区域中，值类型一般存储在堆栈中(但如果值类型在引用类型中声明为字段，它们就内联存储在堆中)。第 2 章“C#基础”讨论堆栈和堆，及其工作原理。

### 1.3.3 强数据类型

中间语言的一个重要方面是它基于强数据类型。所有的变量都清晰地标记为属于某个特定数据类型(在中间语言中没有 Visual Basic 和脚本语言中的 Variant 数据类型)。特别是中间语言一般不允许对模糊的数据类型执行任何操作。

例如，Visual Basic 6 开发人员习惯于传递变量，而无需考虑它们的类型，因为 Visual Basic 6 会自动进行所需的类型转换。C++ 开发人员习惯于在不同类型之间转换指针类型。执行这类操作将大大提高性能，但破坏了类型的安全性。因此，这类操作只能在某些编译为托管代码的语言中的特殊情况下进行。确实，指针(相对于引用)只能在标记了的 C# 代码块中使用，但在 Visual Basic 中不能使用(但一般在托管 C++ 中允许使用)。在代码中使用指针会立即导致 CLR 提供的内存类型安全性检查失败。

注意，一些与 .NET 兼容的语言，例如 Visual Basic 2008，在类型化方面的要求仍比较松，但这是可以的，因为编译器在后台确保在生成的 IL 上强制类型安全。

尽管强迫实现类型的安全性最初会降低性能，但在许多情况下，我们从 .NET 提供的、依赖于类型安全的服务中获得的好处更多。这些服务包括：

- 语言的互操作性
- 垃圾收集
- 安全性
- 应用程序域

下面讨论强数据类型化对这些 .NET 特性非常重要的原因。

#### 1. 语言互操作性中强数据类型的重要性

如果类派生自其他类，或包含其他类的实例，它就需要知道其他类使用的所有数据类型，这就是强数据类型非常重要的原因。实际上，过去没有任何系统指定这些信息，从而成为语言继承和交互操作的真正障碍。这类信息不只是一个标准的可执行文件或 DLL 中出现。

假定将 Visual Basic 2008 类中的一个方法定义为返回一个 Integer——Visual Basic 2008 可以使用的标准数据类型之一。但 C# 没有该名称的数据类型。显然，我们只能从该类中派生，再使用这个方法，如果编译器知道如何把 Visual Basic 2008 的 Integer 类型映射为 C# 定义的某种已知类型，就可以在 C# 代码中使用返回的类型。这个问题在 .NET 中是如何解决的？

##### (1) 通用类型系统(CTS)

这个数据类型问题在 .NET 中使用通用类型系统(CTS)得到了解决。CTS 定义了可以在中间语言中使用的预定义数据类型，所有面向 .NET Framework 的语言都可以生成最终基于这些类型的编译代码。

例如，Visual Basic 2008 的 Integer 实际上是一个 32 位有符号的整数，它实际映射为中间语言类型 Int32。因此在中间语言代码中就指定这种数据类型。C#编译器可以使用这种类型，所以就不会有问题了。在源代码中，C#用关键字 int 来表示 Int32，所以编译器就认为 Visual Basic 2008 方法返回一个 int 类型的值。

通用类型系统不仅指定了基本数据类型，还定义了一个内容丰富的类型层次结构，其中包含设计合理的位置，在这些位置上，代码允许定义它自己的类型。通用类型系统的层次结构反映了中间语言的单一继承的面向对象方法，如图 1-1 所示。

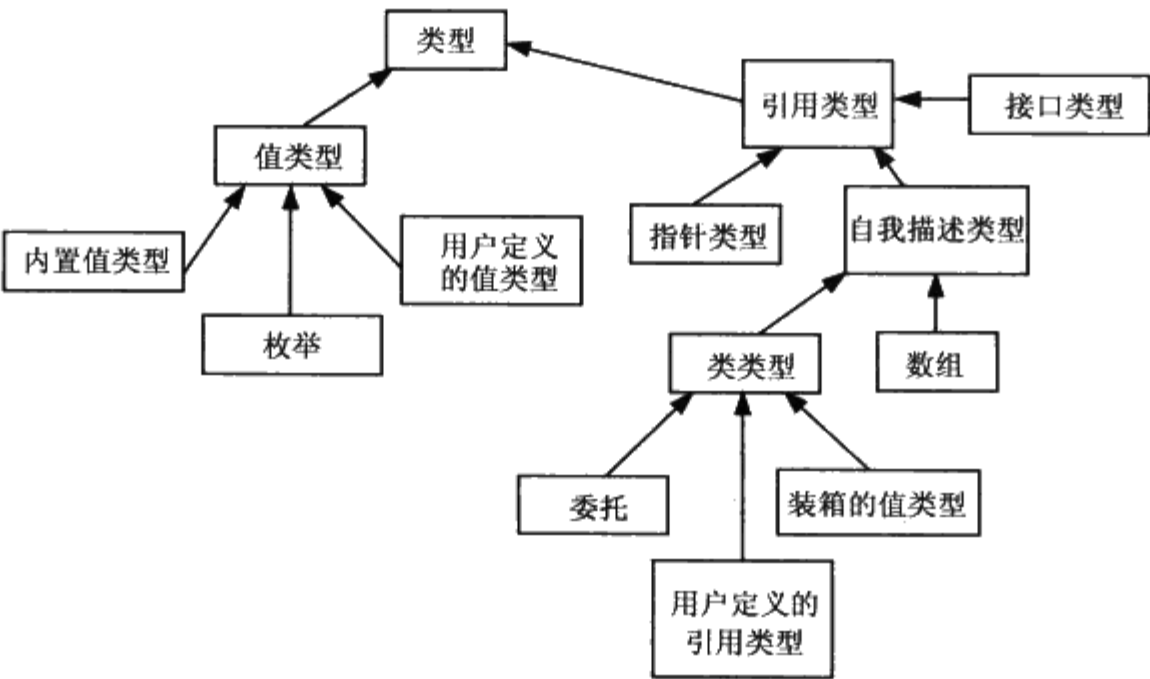


图 1-1

这个树形结构中的类型说明如表 1-1 所示。

表 1-1

类 型	含 义
Type	代表任何类型的基类
Value Type	代表任何值类型的基类
Reference Types	通过引用来访问，且存储在堆中的任何数据类型
Built-in Value Types	包含大多数标准基本类型，可以表示数字、Boolean 值或字符
Enumerations	枚举值的集合
User-defined Value Types	在源代码中定义，且保存为值类型的数据类型。在 C#中，它表示结构
Interface Types	接口
Pointer Types	指针
Self-describing Types	为垃圾回收器提供信息的数据类型(参见下一节)
Arrays	包含对象数组的类型
Class Types	可自我描述的类型，但不是数组
Delegates	用于把引用包含在方法中的类型
User-defined Reference Types	在源代码中定义，且保存为引用类型的数据类型。在 C#中，它表示类
Boxed Value Types	值类型，临时打包放在一个引用中，以便于存储在堆中



这里没有列出内置的所有值类型，因为第3章将详细介绍它们。在C#中，编译器识别的每个预定义类型都映射为一个IL内置类型。这与Visual Basic 2008是一样的。

## (2) 公共语言规范(CLS)

公共语言规范(Common Language Specification, CLS)和通用类型系统一起确保语言的互操作性。CLS是一个最低标准集，所有面向.NET的编译器都必须支持它。因为IL是一种内涵非常丰富的语言，大多数编译器的编写人员有可能把给定编译器的功能限制为只支持IL和CLS提供的一部分特性。只要编译器支持已在CLS中定义的内容，这就是很不错的。

### 提示：

编写非CLS兼容代码是完全可以接受的，只是在编写了这种代码后，就不能保证编译好的IL代码完全支持语言的互操作性。

下面的一个例子是有关区分大小写字母的。IL是区分大小写的语言。使用这些语言的开发人员常常利用区分大小写所提供的灵活性来选择变量名。但Visual Basic 2008是不区分大小写的语言。CLS就要指定CLS兼容代码不使用任何只根据大小写来区分的名称。因此，Visual Basic 2008代码可以与CLS兼容代码一起使用。

这个例子说明了CLS的两种工作方式。首先是各个编译器的功能不必强大到支持.NET的所有功能，这将鼓励人们为其他面向.NET的编程语言开发编译器。第二，它提供如下保证：如果限制类只能使用CLS兼容的特性，就要保证用其他兼容语言编写的代码可以使用这个类。

这种方法的优点是使用CLS兼容特性的限制只适用于公共和受保护的类成员和公共类。在类的私有实现方式中，可以编写非CLS代码，因为其他程序集(托管代码的单元，参见本章后面的内容)中的代码不能访问这部分代码。

这里不深入讨论CLS规范。在一般情况下，CLS对C#代码的影响不会太大，因为C#中的非CLS兼容特性非常少。

## 2. 垃圾收集

垃圾收集器用来在.NET中进行内存管理，特别是它可以恢复正在运行中的应用程序需要的内存。到目前为止，Windows平台已经使用了两种技术来释放进程向系统动态请求的内存：

- 完全以手工方式使用应用程序代码完成这些工作。
- 让对象维护引用计数。

让应用程序代码负责释放内存是低级高性能的语言使用的技术，例如C++。这种技术很有效，且可以让资源在不需要时就释放(一般情况下)，但其最大的缺点是频繁出现错误。请求内存的代码还必须显式通知系统它什么时候不再需要该内存。但这是很容易被遗漏的，从而导致内存泄漏。

尽管现代的开发环境提供了帮助检测内存泄漏的工具，但它们很难跟踪错误，因为直到内存已大量泄漏从而使Windows拒绝为进程提供资源时，它们才会发挥作用。到那个时候，由于对内存的需求很大，会使整个计算机变得相当慢。

维护引用计数是COM对象采用的一种技术，其方法是每个COM组件都保留一个计数，记录客户机目前对它的引用数。当这个计数下降到0时，组件就会删除自己，并释放相应的内存和资源。它带来的问题是仍需要客户机通知组件它们已经完成了内存的使用。只要有一个客

户机没有这么做，对象就仍驻留在内存中。在某些方面，这是比 C++ 内存泄漏更为严重的问题，因为 COM 对象可能存在于它自己的进程中，从来不会被系统删除(在 C++ 内存泄漏问题上，系统至少可以在进程中断时释放所有的内存)。

.NET 运行库采用的方法是垃圾收集器，这是一个程序，其目的是清理内存，方法是所有动态请求的内存都分配到堆上(所有的语言都是这样处理的，但在 .NET 中，CLR 维护它自己的托管堆，以供 .NET 应用程序使用)，当 .NET 检测到给定进程的托管堆已满，需要清理时，就调用垃圾收集器。垃圾收集器处理目前代码中的所有变量，检查对存储在托管堆上的对象的引用，确定哪些对象可以从代码中访问——即哪些对象有引用。没有引用的对象就不能再从代码中访问，因而被删除。Java 就使用与此类似的垃圾收集系统。

之所以在 .NET 中使用垃圾收集器，是因为中间语言已用来处理进程。其规则要求，第一，不能引用已有的对象，除非复制已有的引用。第二，中间语言是类型安全的语言。在这里，其含义是如果存在对对象的任何引用，该引用中就有足够的信息来确定对象的类型。

垃圾收集器机制不能和诸如非托管 C++ 这样的语言一起使用，因为 C++ 允许指针自由地转换数据类型。

垃圾收集器的一个重要方面是它的不确定性。换言之，不能保证什么时候会调用垃圾收集器：.NET 运行库决定需要它时，就可以调用它(除非明确调用垃圾收集器)。但可以重写这个过程，在代码中调用垃圾收集器。

### 3. 安全性

.NET 很好地补足了 Windows 提供的安全机制，因为它提供的安全机制是基于代码的安全性，而 Windows 仅提供了基于角色的安全性。

基于角色的安全性建立在运行进程的账户的身份基础上，换言之，就是谁拥有和运行进程。另一方面，基于代码的安全性建立在代码实际执行的任务和代码的可信程度上。因为中间语言提供了强大的类型安全性，所以 CLR 可以在运行代码前检查它，以确定是否有需要的安全权限。.NET 还提供了一种机制，可以在运行代码前指定代码需要什么安全权限。

基于代码的安全性非常重要，原因是它降低了运行来历不明的代码的风险(例如代码是从 Internet 上下载来的)。即使代码运行在管理员账户下，也有可能使用基于代码的安全性，来确定这段代码是否仍不能执行管理员账户一般允许执行的某些类型的操作，例如读写环境变量、读写注册表或访问 .NET 反射特性。

安全问题详见本书后面的第 20 章。

### 4. 应用程序域

应用程序域是 .NET 中的一个重要技术改进，它用于减少运行应用程序的系统开销，这些应用程序需要与其他程序分离开来，但仍需要彼此通信。典型的例子是 Web 服务器应用程序，它需要同时响应许多浏览器请求。因此，要有许多组件实例同时响应这些同时运行的请求。

在 .NET 开发出来以前，可以让这些实例共享同一个进程，但此时一个运行的实例就有可能导致整个网站的崩溃；也可以把这些实例孤立在不同的进程中，但这样做会增加相关性能的系统开销。

到现在为止，孤立代码的唯一方式是通过进程来实现的。在运行一个新的应用程序时，它

会在一个进程环境内运行。Windows 通过地址空间把进程分隔开来。这样，每个进程有 4GB 的虚拟内存来存储其数据和可执行代码(4GB 对应于 32 位系统，64 位系统要用更多的内存)。Windows 利用额外的间接方式把这些虚拟内存映射到物理内存或磁盘空间的一个特殊区域中，每个进程都会有不同的映射，虚拟地址空间块映射的物理内存之间不能有重叠，这种情况如图 1-2 所示。

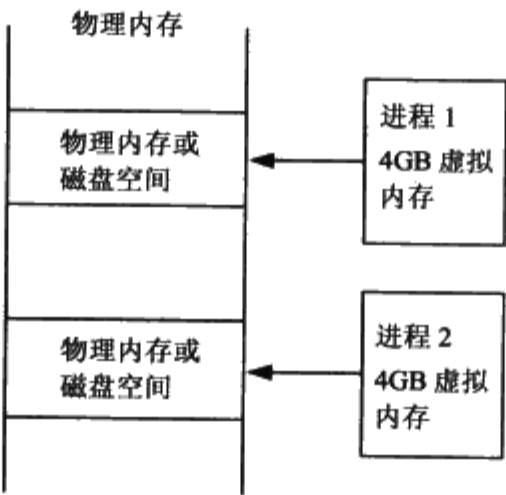


图 1-2

在一般情况下，任何进程都只能通过指定虚拟内存中的一个地址来访问内存——即进程不能直接访问物理内存，因此一个进程不可能访问分配给另一个进程的内存。这样就可以确保任何执行出错的代码不会损害其地址空间以外的数据(注意在 Windows 95/98 上，这些保护措施不像在 Windows NT/2000/XP/2003/Vista 上那样强大，所以理论上存在应用程序因写入不对应的内存而导致 Windows 崩溃的可能性)。

进程不仅是运行代码的实例相互隔离的一种方式，在 Windows NT/2000/XP/2003/Vista 系统上，它们还可以构成分配了安全权限和许可的单元。每个进程都有自己的安全标识，明确地表示 Windows 允许该进程可以执行的操作。

进程对确保安全有很大的帮助，而它们的一大缺点是性能。许多进程常常在一起工作，因此需要相互通信。一个常见的例子是进程调用一个 COM 组件，而该 COM 组件是可执行的，因此需要在它自己的进程上运行。在 COM 中使用代理时也会发生类似的情况。因为进程不能共享任何内存，所以必须使用一个复杂的编组过程在进程之间复制数据。这对性能有非常大的影响。如果需要使组件一起工作，但不希望性能有损失，唯一的方法是使用基于 DLL 的组件，让所有的组件在同一个地址空间中运行——其风险是执行出错的组件会影响其他组件。

应用程序域是分离组件的一种方式，它不会导致因在进程之间传送数据而产生的性能问题。其方法是把任何一个进程分解到多个应用程序域中，每个应用程序域大致对应一个应用程序，执行的每个线程都运行在一个具体的应用程序域中，如图 1-3 所示。

如果不同的可执行文件都运行在同一个进程空间中，显然它们就能轻松地共享数据，因为理论上它们可以直接访问彼此的数据。虽然在理论上这是可以实现的，但是 CLR 会检查每个正在运行的应用程序的代码，以确保这些代码不偏离它自己的数据区域，保证不发生直接访问其他进程的数据的情况。这初看起来是不可能的，如何告诉程序要做什么工作，而又不真正运行它？

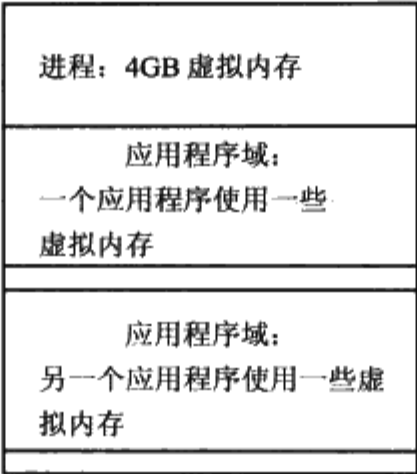


图 1-3

实际上，这么做通常是可能的，因为中间语言拥有强大的类型安全功能。在大多数情况下，除非代码明确使用不安全的特性，例如指针，否则它使用的数据类型可以确保内存不会被错误地访问。例如，.NET 数组类型执行边界检查，以禁止执行超出边界的数组操作。如果运行的应用程序的确需要与运行在不同应用程序域中的其他应用程序通信或共享数据，就必须调用.NET 的远程服务。

被验证不能访问超出其应用程序域的数据(而不是通过明确的远程机制)的代码就是内存类型安全的代码，这种代码与运行在同一个进程中但应用程序域不同的类型安全代码一起运行是安全的。

1.3.4 通过异常处理错误

.NET Framework 可以根据异常使用相同的机制处理错误情况，这与 Java 和 C++是一样的。C++开发人员应注意到，因为 IL 有非常强大的类型系统，所以在 IL 中以 C++的方式使用异常不会带来相关的性能问题。另外，.NET 和 C#也支持 finally 块，这是许多 C++开发人员长久以来的愿望。

第 14 章会详细讨论异常。简要地说，代码的某些领域被看作是异常处理程序例程，每个例程都能处理某种特殊的错误情况(例如，找不到文件，或拒绝执行某些操作的许可)。这些条件可以定义得很宽或很窄。异常结构确保在发生错误情况时，执行进程立即跳到最合适的异常处理程序例程上，处理错误情况。

异常处理的结构还提供了一种方便的方式，当对象包含错误情况的准确信息时，该对象就可以传送给错误处理例程。这个对象包括给用户提供的相应信息和在代码的什么地方检测到错误的确切信息。

大多数异常处理结构，包括异常发生时的程序流控制，都是由高级语言处理的，例如 C#、Visual Basic 2008 和 C++，任何中间语言中的命令都不支持它。例如，C#使用 try {}、catch {} 和 finally {} 代码块来处理它，详见第 14 章。

.NET 提供了一种基础结构，让面向.NET 的编译器支持异常处理。特别是它提供了一组.NET 类来表示异常，语言的互操作性则允许异常处理代码处理被抛出的异常对象，无论异常处理代码使用什么语言编写，都是这样。语言的无关性没有体现在 C++和 Java 的异常处理中，但在 COM 的错误处理机制中有一定限度的体现。COM 的错误处理机制包括从方法中返回错误代码

以及传递错误对象。在不同的语言中，异常的处理是一致的，这是多语言开发的重要一环。

### 1.3.5 特性的使用

特性(attribute)是使用 C++编写 COM 组件的开发人员很熟悉的一个功能(在 Microsoft 的 COM 接口定义语言(Interface Definition Language, IDL)中使用特性)。特性最初是为了在程序中提供与某些项相关的额外信息，以供编译器使用。

.NET 支持特性，因此现在 C++、C#和 Visual Basic 2008 也支持特性。但在.NET 中，对特性的革新是建立了一个机制，通过该机制可以在源代码中定义自己的特性。这些用户定义的特性将和对应数据类型或方法的元数据放在一起，这对于文档说明书十分有用，它们和反射技术一起使用，以根据特性执行编程任务。另外，与.NET 的语言无关性的基本原理一样，特性也可以在一种语言的源代码中定义，而被用另一种语言编写的代码读取。

本书的第 13 章详细介绍了特性。

## 1.4 程序集

程序集(assembly)是包含编译好的、面向.NET Framework 的代码的逻辑单元。本章不详细论述程序集，而在第 17 章中论述，下面概述其中的要点。

程序集是完全自我描述性的，也是一个逻辑单元而不是物理单元，它可以存储在多个文件中(动态程序集的确存储在内存中，而不是存储在文件中)。如果一个程序集存储在多个文件中，其中就会有一个包含入口点的主文件，该文件描述了程序集中的其他文件。

注意可执行代码和库代码使用相同的程序集结构。唯一的区别是可执行的程序集包含一个主程序入口点，而库程序集不包含。

程序集的一个重要特性是它们包含的元数据描述了对应代码中定义的类型和方法。程序集也包含描述程序集本身的元数据，这种程序集元数据包含在一个称为“程序集清单”的区域中，可以检查程序集的版本及其完整性。

**注意：**

ildasm 是一个基于 Windows 的实用程序，可以用于检查程序集的内容，包括程序集清单和元数据。第 17 章将介绍 ildasm。

程序集包含程序的元数据，表示调用给定程序集中的代码的应用程序或其他程序集不需要指定注册表或其他数据源，以确定如何使用该程序集。这与以前的 COM 有很大的区别，以前，组件的 GUID 和接口必须从注册表中获取，在某些情况下，方法和属性的详细信息也需要从类型库中读取。

把数据分散在 3 个以上的不同位置上，可能会出现信息不同步的情况，从而妨碍其他软件成功地使用该组件。有了程序集后，就不会发生这种情况，因为所有的元数据都与程序的可执行指令存储在一起。注意，即使程序集存储在几个文件中，数据也不会出现不同步的问题。这是因为包含程序集入口的文件也存储了其他文件的细节、散列和内容，如果一个文件被替换，或者被塞满，系统肯定会检测出来，并拒绝加载程序集。



程序集有两种类型：共享程序集和私有程序集。

### 1.4.1 私有程序集

私有程序集是最简单的一种程序集类型。私有程序集一般附带在某个软件上，且只能用于该软件。附带私有程序集的常见情况是，以可执行文件或许多库的方式提供应用程序，这些库包含的代码只能用于该应用程序。

系统可以保证私有程序集不被其他软件使用，因为应用程序只能加载位于主执行文件所在文件夹或其子文件夹中的程序集。

用户一般会希望把商用软件安装在它自己的目录下，这样软件包没有覆盖、修改或加载另一个软件包的私有程序集的风险。私有程序集只能用于自己的软件包，这样，用户对什么软件使用它们就有了更多的控制。因此，不需要采取安全措施，因为这没有其他商用软件用某个新版本的程序集覆盖原来的私有程序集的风险(但软件是专门执行怀有恶意的损害性操作的情况除外)。名称也不会有冲突。如果私有程序集中的类正巧与另一个人的私有程序集中的类同名，是不会有问题的，因为给定的应用程序只能使用私有程序集的名称。

因为私有程序集完全是自含式的，所以安装它的过程就很简单。只需把相应的文件放在文件系统的对应文件夹中即可(不需要注册表项)，这个过程称为“0 影响(xcopy)安装”。

### 1.4.2 共享程序集

共享程序集是其他应用程序可以使用的公共库。因为其他软件可以访问共享程序集，所以需要采取一定的保护措施来防止以下风险：

- 名称冲突，另一个公司的共享程序集执行的类型与自己的共享程序集中的类型同名。  
因为客户机代码理论上可以同时访问这些程序集，所以这是一个严重的问题。
- 程序集被同一个程序集的不同版本覆盖——新版本与某些已有的客户机代码不兼容。

这些问题的解决方法是把共享程序集放在文件系统的一个特定的子目录树中，称为全局程序集高速缓存(GAC)。与私有程序集不同，不能简单地把共享程序集复制到对应的文件夹中，而需要专门安装到高速缓存中，这个过程可以用许多.NET 工具来完成，其中包含对程序集的检查、在程序集高速缓存中设置一个小的文件夹层次结构，以确保程序集的完整性。

为了避免名称冲突，共享程序集应根据私钥加密法指定一个名称(私有程序集只需要指定与其主文件名相同的名称即可)。该名称称为强名(strong name)，并保证其唯一性，它必须由要引用共享程序集的应用程序来引用。

与覆盖程序集相关的问题，可以通过在程序集清单中指定版本信息来解决，也可以通过同时安装来解决。

### 1.4.3 反射

因为程序集存储了元数据，包括在程序集中定义的所有类型和这些类型的成员的细节，所以可以编程访问这些元数据。这个技术称为反射，第 13 章详细介绍了它们。该技术很有趣，因为它表示托管代码实际上可以检查其他托管代码，甚至检查它自己，以确定该代码的信息。它们



常常用于获取特性的详细信息，也可以把反射用于其他目的，例如作为实例化类或调用方法的一种间接方式，如果把方法上的类名指定为字符串，就可以选择类来实例化方法，以便在运行时调用，而不是在编译时调用，例如根据用户的输入来调用(动态绑定)。

## 1.5 .NET Framework 类

至少从开发人员的角度来看，编写托管代码的最大好处是可以使用.NET 基类库。

.NET 基类是一个内容丰富的托管代码类集合，它可以完成以前要通过 Windows API 来完成的绝大多数任务。这些类派生自与中间语言相同的对象模型，也基于单一继承性。无论.NET 基类是否合适，都可以实例化对象，也可以从它们派生自己的类。

.NET 基类的一个优点是它们非常直观和易用。例如，要启动一个线程，可以调用 Thread 类的 Start()方法。要禁用 TextBox，应把 TextBox 对象的 Enabled 属性设置为 false。Visual Basic 和 Java 开发人员非常熟悉这种方式。它们的库都很容易使用，但对于 C++开发人员来说这是极大的解脱，因为他们多年来一直在使用诸如 GetDIBits()、RegisterWndClassEx()和 IsEqualIID()这样的 API 函数，以及需要传递 Windows 句柄的函数。

另一方面，C++开发人员总是很容易访问整个 Windows API，而 Visual Basic 6 和 Java 开发人员只能访问其语言所能访问的基本操作系统功能。.NET 基类的新增内容就是把 Visual Basic 和 Java 库的易用性和 Windows API 函数的丰富功能结合起来。但 Windows 仍有许多功能不能通过基类来使用，而需要调用 API 函数。但一般情况下，这只限于比较复杂的特性。基类库足以应付日常工作的使用。如果需要调用 API 函数，.NET 提供了所谓的“平台调用”，来确保对数据类型进行正确的转换，这样无论是使用 C#、C++或 Visual Basic 2008 进行编码，该任务都不会比直接从已有的 C++代码中调用函数更困难。

**注意：**

WinCV 是一个基于 Windows 的实用程序，可以用于浏览基类库中的类、结构、接口和枚举。本书将在第 15 章介绍 WinCV。

第 3 章主要介绍基类。完成了 C#语言语法的概述后，本书的其余内容将主要说明如何使用.NET Framework 3.5 的.NET 基类库中的各种类，即各种基类是如何工作的。.NET 3.5 基类包括：

- IL 提供的核心功能，例如，通用类型系统中的基本数据类型，详见第 3 章。
- Windows GUI 支持和控件(第 31 和 34 章)
- Web 窗体(ASP.NET，第 37 和 38 章)
- 数据访问(ADO.NET，第 26~30 章)
- 目录访问(第 46 章)
- 文件系统和注册表访问(第 25 章)
- 网络和 Web 浏览(第 41 章)
- .NET 特性和反射(第 13 章)
- 访问 Windows 操作系统的各个方面(例如环境变量等，第 20 章)
- COM 互操作性(第 44 和 24 章)

附带说一下, 根据 Microsoft 源文件, 大部分 .NET 基类实际上都是用 C# 编写的!

## 1.6 命名空间

命名空间是 .NET 避免类名冲突的一种方式。例如, 命名空间可以避免下述情况: 定义一个类来表示一个顾客, 称此类为 `Customer`, 同时其他人也在做相同的事(这有一个类似的场景——顾客有相当多的业务)。

命名空间不过是数据类型的一种组合方式, 但命名空间中所有数据类型的名称都会自动加上该命名空间的名字作为其前缀。命名空间还可以相互嵌套。例如, 大多数用于一般目的的 .NET 基类位于命名空间 `System` 中, 基类 `Array` 在这个命名空间中, 所以其全名是 `System.Array`。

.NET 需要在命名空间中定义所有的类型, 例如, 可以把 `Customer` 类放在命名空间 `YourCompanyName` 中, 则这个类的全名就是 `YourCompanyName.Customer`。

注意:

如果没有显式提供命名空间, 类型就添加到一个没有名称的全局命名空间中。

Microsoft 建议在大多数情况下, 都至少要提供两个嵌套的命名空间名, 第一个是公司名, 第二个是技术名称或软件包的名称, 而类是其中的一个成员, 例如 `YourCompanyName.Sales-Services.Customer`。在大多数情况下, 这么做可以保证类的名称不会与其他组织编写的类名冲突。

第 2 章将详细介绍命名空间。

## 1.7 用 C# 创建 .NET 应用程序

C# 可以用于创建控制台应用程序: 仅使用文本、运行在 DOS 窗口中的应用程序。在进行单元测试类库、创建 Unix/Linux daemon 进程时, 就要使用控制台应用程序。但是, 我们常常使用 C# 创建利用许多与 .NET 相关的技术的应用程序, 下面简要论述可以用 C# 创建的不同类型的应用程序。

### 1.7.1 创建 ASP.NET 应用程序

ASP 是用于创建带有动态内容的 Web 页面的一种 Microsoft 技术。ASP 页面基本是一个嵌有服务器端 VBScript 或 JavaScript 代码块的 HTML 文件。当客户浏览器请求一个 ASP 页面时, Web 服务器就会发送页面的 HTML 部分, 并处理服务器端脚本。这些脚本通常会查询数据库的数据, 在 HTML 中标记数据。ASP 是客户建立基于浏览器的应用程序的一种便利方式。

但 ASP 也有缺点。首先, ASP 页面有时显示得比较慢, 因为服务器端代码是解释性的, 而不是编译的。第二, ASP 文件很难维护, 因为它不是结构化的, 服务器端的 ASP 代码和一般的 HTML 会混合在一起。第三, ASP 有时开发起来会比较困难, 因为它不支持错误处理和类型检查。特别是如果使用 VBScript, 并希望在页面中进行错误处理, 就必须使用 `On Error Resume Next` 语句, 通过 `Err.Number` 检查每个组件调用, 以确保该调用正常进行。

ASP.NET 是 ASP 的全新修订版本, 它解决了 ASP 的许多问题。但 ASP.NET 页面并没有替

代 ASP, 而是可以与原来的 ASP 应用程序在同一个服务器上并存。当然, 也可以用 C# 编写 ASP.NET。

后面的章节(第 37、38 和 39 章)会详细讨论 ASP.NET, 这里仅解释它的一些重要特性。

### 1. ASP.NET 的特性

首先, 也是最重要的是, ASP.NET 页面是结构化的。这就是说, 每个页面都是一个继承了 .NET 类 `System.Web.UI.Page` 的类, 可以重写在 `Page` 对象的生存期中调用的一系列方法, (可以把这些事件看成是页面所特有的, 对应于原 ASP 的 `global.asa` 文件中的 `OnApplication_Start` 和 `OnSession_Start` 事件)。因为可以把一个页面的功能放在有明确含义的事件处理程序中, 所以 ASP.NET 比较容易理解。

ASP.NET 页面的另一个优点是可以在 Visual Studio 2008 中创建它们, 在该环境下, 可以创建 ASP.NET 页面使用的业务逻辑和数据访问组件。Visual Studio 2008 项目(也称为解决方案)包含了与应用程序相关的所有文件。而且, 也可以在编辑器中调试传统的 ASP 页面, 在以前使用 Visual InterDev 时, 把 InterDev 和项目的 Web 服务器配置为支持调试常常是一个让人头痛的问题。

最清楚的是, ASP.NET 的后台编码功能允许进一步采用结构化的方式。ASP.NET 允许把页面的服务器端功能单独放在一个类中, 把该类编译为 DLL, 并把该 DLL 放在 HTML 部分下面的一个目录中。放在页面顶部的后台编码指令将把该文件与其 DLL 关联起来。当浏览器请求该页面时, Web 服务器就会在页面的后台 DLL 中引发类中的事件。

最后, ASP.NET 在性能的提高上非常明显。传统的 ASP 页面是和每个页面请求一起解释, 而 Web 服务器是在编译后高速缓存 ASP.NET 页面。这表示以后对 ASP.NET 页面的请求就比 ASP 页面第一次执行的速度快得多。

ASP.NET 还易于编写通过浏览器显示窗体的页面, 这在内联网环境中会使用。传统的方式是基于窗体的应用程序提供一个功能丰富的用户界面, 但较难维护, 因为它们运行在非常多的不同机器上。因此, 当用户界面是必不可少的, 并可以为用户提供扩展支持时, 人们就会依赖基于窗体的应用程序。

### 2. Web 窗体

为了简化 Web 页面的结构, Visual Studio 2008 提供了 Web 窗体。它们允许以创建 Visual Basic 6 或 C++ Builder 窗口的方式图形化地建立 ASP.NET 页面; 换言之, 就是把控件从工具箱拖放到窗体上, 再考虑窗体的代码, 为控件编写事件处理程序。在使用 C# 创建 Web 窗体时, 就是创建一个继承自 `Page` 基类的 C# 类, 以及把这个类看作是后台编码的 ASP.NET 页面。当然不必使用 C# 创建 Web 窗体, 而可以使用 Visual Basic 2008 或另一种 .NET 语言来创建。

过去, Web 开发的困难使一些开发小组不愿意使用 Web。为了成功地进行 Web 开发, 必须了解非常多的不同技术, 例如 VBScript、ASP、DHTML、JavaScript 等。把窗体概念应用于 Web 页面, Web 窗体就可以使 Web 开发容易许多。

### 3. Web 服务器控件

用于添加到 Web 窗体上的控件与 ActiveX 控件并不是同一种控件, 它们是 ASP.NET 命名

空间中的 XML 标记, 当请求一个页面时, Web 浏览器会动态地把它们转换为 HTML 和客户端脚本。Web 服务器能以不同的方式显示相同的服务器端控件, 产生一个对应于请求者特定 Web 浏览器的转换。这意味着现在很容易为 Web 页面编写相当复杂的用户界面, 而不必担心如何确保页面运行在可用的任何浏览器上, 因为 Web 窗体会完成这些任务。

可以使用 C# 或 Visual Basic 2008 扩展 Web 窗体工具箱。创建一个新服务器端控件, 仅是执行 .NET 的 `System.Web.UI.WebControls.WebControl` 类而已。

#### 4. XML Web 服务

目前, HTML 页面解决了 World Wide Web 上的大部分通信问题。有了 XML, 计算机就可以用一种独立于设备的格式, 在 Web 上彼此通信。将来, 计算机可以使用 Web 和 XML 交流信息, 而不是专用的线路和专用的格式, 例如 EDI (Electronic Data Interchange)。XML Web 服务是为面向 Web 的服务而设计的, 即远程计算机彼此提供可以分析和重新格式化的动态信息, 最后显示给用户。XML Web 服务是计算机给 Web 上的其他计算机以 XML 格式显示信息的一种便利方式。

在技术上, .NET 上的 XML Web 服务是给请求的客户返回 XML 而不是 HTML 的 ASP.NET 页面。这种页面有后台编码的 DLL, 它包含了派生自 `WebService` 类的类。Visual Studio 2008 IDE 提供的引擎简化了 Web 服务的开发。

公司选择使用 XML Web 服务主要有两个原因。第一是因为它们依赖于 HTTP, 而 XML Web 服务可以把现有的网络(HTTP)用作传输信息的媒介。第二是因为 XML Web 服务使用 XML, 该数据格式是自我描述的、非专用的、独立于平台的。

#### 1.7.2 创建 Windows 窗体

C# 和 .NET 非常适合于 Web 开发, 它们还为所谓的“胖客户端”应用程序提供了极好的支持, 这种“胖客户端”应用程序必须安装在最终用户的机器上, 来处理大多数操作, 这种支持来源于 Windows 窗体。

Windows 窗体是 Visual Basic 6 窗体的 .NET 版本, 要设计一个图形化的窗口界面, 只需把控件从工具箱拖放到 Windows 窗体上即可。要确定窗口的行为, 应为该窗体的控件编写事件处理例程。Windows Form 项目编译为 .EXE, 该 EXE 必须与 .NET 运行库一起安装在最终用户的计算机上。与其他 .NET 项目类型一样, Visual Basic 2008 和 C# 都支持 Windows Form 项目。第 31 章将详细介绍 Windows 窗体。

#### 1.7.3 使用 Windows Presentation Foundation(WPF)

有一种最新的技术叫做 Windows Presentation Foundation(WPF)。WPF 在建立应用程序时使用 XAML。XAML 表示可扩展的应用程序标记语言(Extensible Application Markup Language)。这种在 Microsoft 环境下创建应用程序的新方式在 2006 年引入, 是 .NET Framework 3.0 和 3.5 的一部分。要运行 WPF 应用程序, 需要在客户机上安装 .NET Framework 3.0 或 3.5。WPF 应用程序可用于 Windows Vista、Windows XP、Windows Server 2003 和 Windows Server 2008 (只有这些操作系统能安装 .NET Framework 3.0 或 3.5)。

XAML 是用于创建窗体的 XML 声明, 它代表 WPF 应用程序的所有可视化部分和操作。虽然可以编程利用 WPF 应用程序, 但 WPF 是迈向声明性编程的一步, 而声明性编程是编程业

的趋势。声明性编程是指，不是利用编译语言，如 C#、VB 或 Java，通过编程来创建对象，而是通过 XML 类型的编程来声明所有的元素。第 34 章详细介绍了如何使用 XAML 和 C# 建立这些新类型的应用程序。

#### 1.7.4 Windows 控件

Web 窗体和 Windows 窗体的开发方式一样，但应为它们添加不同类型的控件。Web 窗体使用 Web 服务器控件，Windows 窗体使用 Windows 控件。

Windows 控件比较类似于 ActiveX 控件。在执行 Windows 控件后，它会编译为必须安装到客户机器上的 DLL。实际上，.NET SDK 提供了一个实用程序，为 ActiveX 控件创建包装器，以便把它们放在 Windows 窗体上。与 Web 控件一样，Windows 控件的创建需要派生于特定的类 `System.Windows.Forms.Control`。

#### 1.7.5 Windows 服务

Windows 服务(最初称为 NT 服务)是一个在 Windows NT/2000/XP/2003/Vista (但没有 Windows 9x)后台运行的程序。当希望程序连续运行，响应事件，但没有用户的明确启动操作时，就应使用 Windows 服务。例如 Web 服务器上的 World Wide Web 服务，它们监听来自客户的 Web 请求。

用 C# 编写服务是非常简单的。`System.ServiceProcess` 命名空间中的 .NET Framework 基类可以处理许多与服务相关的样本任务，另外，Visual Studio 2008 允许创建 C# Windows Service 项目，为基本 Windows 服务编写 C# 源代码。第 23 章将详细介绍如何编写 C# Windows 服务。

#### 1.7.6 Windows Communication Foundation(WCF)

通过基于 Microsoft 的技术，可以采用许多方式将数据和服务从一处移动到另一处。例如，可以使用 ASP.NET Web 服务、.NET Remoting、Enterprise Services 和用于初学者的 MSMQ。应采用哪种技术？这要考虑具体要达到的目标，因为每种技术都适合于不同的场合。

因此，Microsoft 把所有这些技术集成在一起，放在 .NET Framework 3.0 和 3.5 中。现在只有一种移动数据的方式——Windows Communication Foundation(WCF)。WCF 允许建立好服务后，只要修改配置文件，就可以用多种方式提供该服务(甚至在不同的协议下)。WCF 是一种连接各种系统的强大的新方式。第 42 章将详细介绍 WCF。

### 1.8 C#在.NET 企业体系结构中的作用

C# 需要 .NET 运行库，在几年内大多数客户机——特别是大多数家用 PC——就可以安装 .NET 了。而且，安装 C# 应用程序在方式上类似于安装 .NET 可重新分布的组件。因此，企业环境中会有许多 C# 应用程序。实际上，C# 为希望建立健全的 n 层客户机/服务器应用程序的公司提供了一个绝佳的机会。

C# 与 ADO.NET 合并后，就可以快速而经常地访问数据库了，例如 SQL Server 和 Oracle



数据库。返回的数据集很容易通过 ADO.NET 对象模型或 LINQ 来处理，并自动显示为 XML，一般通过办公室内联网来传输。

一旦为新项目建立了数据库模式，C#就会为执行一层数据访问对象提供一个极好的媒介，每个对象都能提供对不同的数据库表的插入、更新和删除访问。

因为这是第一个基于组件的 C 语言，所以 C#非常适合于执行业务对象层。它为组件之间的通信封装了杂乱的信息，让开发人员把注意力集中在如何把数据访问对象组合在一起，在方法中精确地强制执行公司的业务规则。而且使用特性，C#业务对象可以配备方法级的安全检查、对象池和由 COM+服务提供的 JIT 活动。另外，.NET 附带的实用程序允许新的.NET 业务对象与原来的 COM 组件交互。

要使用 C#创建企业应用程序，可以为数据访问对象创建一个 Class Library 项目，为业务对象创建另一个 Class Library 项目。在开发时，可以使用 Console 项目测试类上的方法。喜欢编程的人可以建立能自动从批处理文件中执行的 Console 项目，测试工作代码是否中断。

注意，C#和.NET 都会影响物理封装可重用类的方式。过去，许多开发人员把许多类放在一个物理组件中，因为这样安排会使部署容易得多；如果有版本冲突问题，就知道在何处进行检查。因为部署.NET 企业组件仅是把文件复制到目录中，所以现在开发人员就可以把他们的类封装到逻辑性更高的离散组件中，而不会遇到 DLL Hell。

最后，用 C#编写的 ASP.NET 页面构成了用户界面的绝妙媒介。ASP.NET 页面是编译过的，所以执行得比较快。它们可以在 Visual Studio 2008 IDE 中调试，所以更加健壮。它们支持所有的语言特性，例如早期绑定、继承和模块化，所以用 C#编写的 ASP.NET 页面是很整洁的，很容易维护。

经验丰富的开发人员对大做广告的新技术和语言都持非常怀疑的态度，不愿意利用新平台，这仅仅是因为他们不愿意。如果读者是一位 IT 部门的企业开发人员，或者通过 World Wide Web 提供应用程序服务，即使一些比较奇异的特性如 XML Web 服务和服务器端控件不算在内，也可以确保 C#和.NET 至少提供了四个优点：

- 组件冲突将很少见，部署工作将更容易，因为同一组件的不同版本可以在同一台机器上并行运行，而不会发生冲突。
- ASP.NET 代码不再难懂。
- 可以利用.NET 基类中的许多功能。
- 对于需要 Windows 窗体用户界面的应用程序来说，利用 C#可以很容易编写这类应用程序。

在某种程度上，以前 Windows 窗体并未受到重视，因为没有 Web 窗体和基于 Internet 的应用程序。但如果用户缺乏 JavaScript、ASP 或相关技术的专业知识，Windows 窗体仍是方便而快速地创建用户界面的一种可行选择。记住管理好代码，使用户界面的逻辑与业务逻辑和数据访问代码分隔开来，这样才能在将来的某一刻把应用程序迁移到浏览器上。另外，Windows 窗体还为家用应用程序和一些小公司长期保留了重要的用户界面。Windows 窗体的新智能客户特性(很容易以在线和离线方式工作)将能开发出新的、更好的应用程序。



## 1.9 小结

本章介绍了许多基础知识，简要回顾了 .NET Framework 的重要方面以及它与 C# 的关系。首先讨论了所有面向 .NET 的语言如何编译为中间语言，之后由公共语言运行库进行编译和执行。我们还讨论了 .NET 的下述特性在编译和执行过程中的作用：

- 程序集和 .NET 基类
- COM 组件
- JIT 编译
- 应用程序域
- 垃圾收集

图 1-4 简要说明了这些特性在编译和执行过程中是如何发挥作用的。

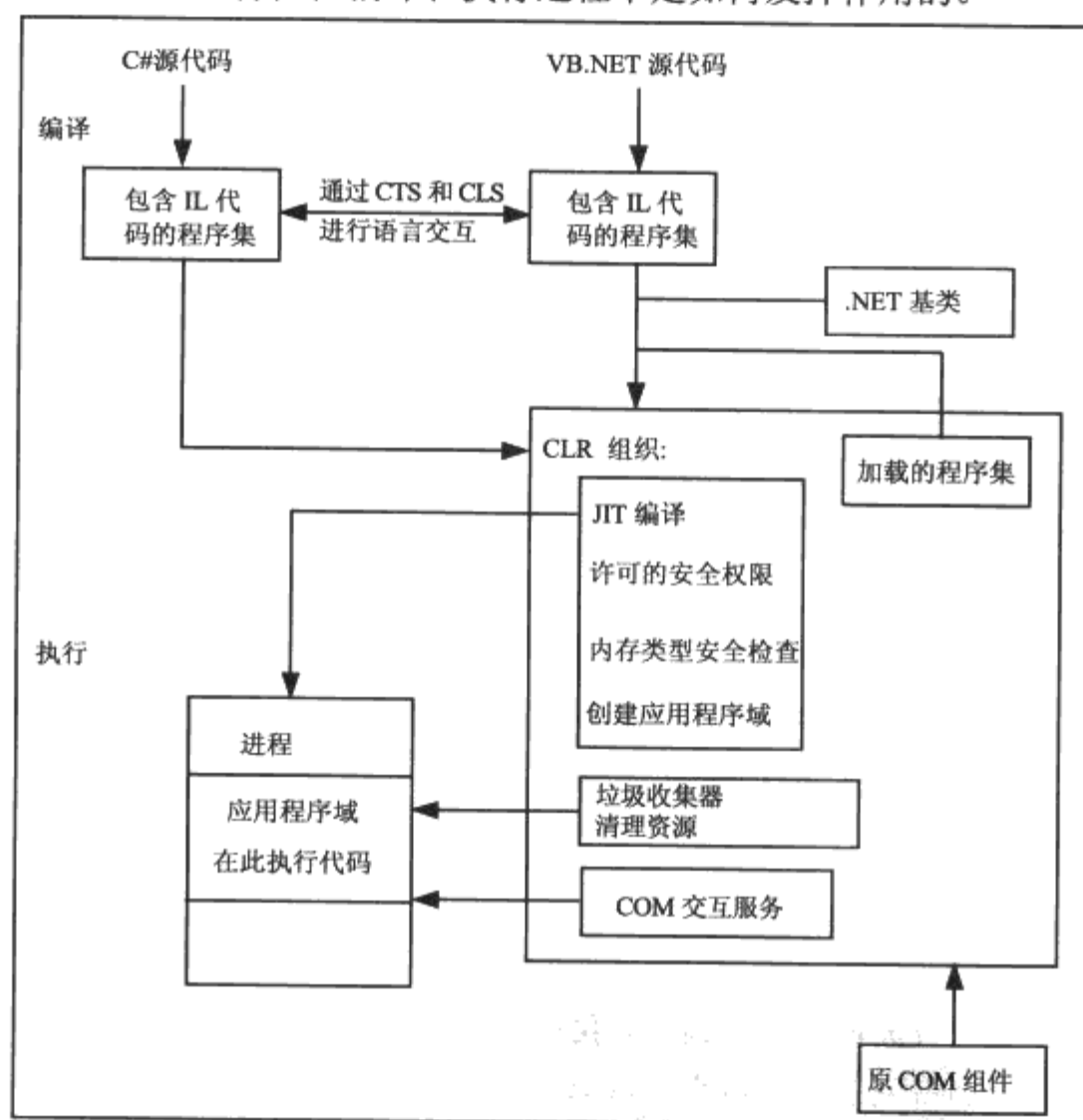


图 1-4

本章还讨论了 IL 的特性，特别是其强数据类型和面向对象的特性。探讨了这些特性如何影响面向 .NET 的语言，包括 C#，并阐述了 IL 的强类型本质如何支持语言的互操作性，以及 CLR 服务，例如垃圾收集和安全性。还讨论了 CLS 和 CTS，来帮助处理语言互操作性。

本章的最后讨论了 C# 如何用作基于几个 .NET 技术(包括 ASP.NET)的应用程序的基础。

第 2 章将介绍如何用 C# 语言编写代码。

# 第 2 章

## C# 基础

理解了 C# 的用途后，就可以学习如何使用它了。本章将介绍 C# 的基础知识，并假定读者具备 C# 编程的基本知识，这是后续章节的基础。本章的主要内容如下：

- 声明变量
- 变量的初始化和作用域
- C# 的预定义数据类型
- 在 C# 程序中使用循环和条件语句指定执行流
- 枚举
- 命名空间
- Main() 方法
- 基本的命令行 C# 编译器选项
- 使用 System.Console 执行控制台 I/O
- 使用注释和文档编制功能
- 预编译器指令
- C# 编程的推荐规则和约定

阅读完本章后，读者就有足够的 C# 知识编写简单的程序了，但还不能使用继承或其他面向对象的特征。这些内容将在本书后面的几章中讨论。

### 2.1 引言

如前所述，C# 是一种面向对象的语言。我们假定读者已经很好地掌握了面向对象(OO)编程的概念。换言之，我们希望读者懂得类、对象、接口和继承的含义。如果读者以前使用过 C++ 或 Java，就应有很好的面向对象编程(OOP)的基础。但是，如果读者不具备 OOP 的背景知识，在继续之前先熟悉 OOP 基础是很有帮助的。

如果读者对 Visual Basic 6、C++ 或 Java 中的一种语言有丰富的编程经验，就应注意在介绍 C# 基础知识时，我们对 C#、C++、Java 和 Visual Basic 6 进行了许多比较。但是，读者也许愿意阅读一本有关 C# 和自己所选语言的比较的图书，来学习 C#。如果是这样，可以从 Wrox Press 网站([www.wrox.com](http://www.wrox.com))上下载不同的文档来学习 C#。

## 2.2 第一个 C#程序

下面编译并运行最简单的 C#程序，这是一个简单的类，包含把信息写到屏幕上的控制台应用程序。

注意：

在后面的几章中，介绍了许多代码示例。编写 C#程序最常用的技巧是使用 Visual Studio 2008 生成一个基本项目，再添加自己的代码。但是，前面几章的目的是讲授 C#语言，为了简单起见，在第 14 章之前避免涉及 Visual Studio 2008。我们使代码显示为简单的文件，这样就可以使用任何文本编辑器键入它们，并在命令行上编译。

### 2.2.1 代码

在文本编辑器(例如 Notepad)中键入下面的代码，把它保存为.cs 文件(例如 First.cs)。Main()方法如下所示：

```
using System;

namespace Wrox.ProCSharp.Basics
{
    class MyFirstCSharpClass
    {
        static void Main()
        {
            Console.WriteLine("This isn't at all like Java!");
            Console.ReadLine();
            return;
        }
    }
}
```

### 2.2.2 编译并运行程序

对源文件运行 C#命令行编译器(csc.exe)，编译这个程序：

```
csc First.cs
```

如果使用 csc 命令在命令行上编译代码，就应注意.NET 命令行工具，包括 csc，只有在设置了某些环境变量后才能使用。根据安装.NET(和 Visual Studio 2008)的方式，这里显示的结果可能与您机器上的结果不同。

注意：

如果没有设置环境变量，有两种解决方法。第一种方法是在运行 csc 之前，在命令行上运行批处理文件%Microsoft Visual Studio 2008%\Common7\Tools\vcvars32.bat。其中%Microsoft Visual Studio 2008 是安装 Visual Studio 2008 的文件夹。第二种方法(更简单)是使用 Visual Studio 2008 命令行代替通常的命令提示窗口。Visual Studio 2008 命令提示在“开始”菜单 | “程序” | Microsoft Visual Studio 2008 | Microsoft Visual Studio Tools 子菜单下。它只是一个命令提示窗口，打开时会自动运行 vcvars32.bat。

编译代码，会生成一个可执行文件 `First.exe`。在命令行或 Windows Explorer 上，像运行任何可执行文件那样运行该文件，得到如下结果：

**csc First.cs**

```
Microsoft (R) Visual C# Compiler version 9.00.20404
for Microsoft (R) .NET Framework version 3.5
Copyright (C) Microsoft Corporation. All rights reserved.
```

**First.exe**

```
This isn't at all like Java!
```

这些信息也许不那么真实！这个程序与 Java 有一些非常相似的地方，但有一两个地方与 Java 或 C++ 不同(如大写的 `Main()` 函数)。下面通过这个程序详细介绍 C# 程序的基本结构。

### 2.2.3 详细介绍

首先对 C# 语法作几个解释。在 C# 中，与其他 C 风格的语言一样，大多数语句都以分号(;) 结尾，语句可以写在多个代码行上，不需要使用续行字符(例如 Visual Basic 中的下划线)。用花括号({ ... })把语句组合为块。单行注释以两个斜杠字符开头(//)，多行注释以一个斜杠和一个星号(/\*)开头，以一个星号和一个斜杠(\*/)结尾。在这些方面，C# 与 C++ 和 Java 一样，但与 Visual Basic 不同。分号和花括号使 C# 代码与 Visual Basic 代码有完全不同的外观。如果您以前使用的是 Visual Basic，就应特别注意每个语句结尾的分号。对于使用 C 风格语言的新用户，忽略分号常常是导致编译错误的一个最主要的原因。另一个方面是，C# 是区分大小写的，也就是说，变量 `myVar` 与 `MyVar` 是两个不同的变量。

在上面的代码示例中，前几行代码是处理命名空间的(如本章后面所述)，命名空间是把相关类组合在一起的方式。Java 和 C++ 开发人员应很熟悉这个概念，但对于 Visual Basic 6 开发人员来说是新概念。C# 命名空间与 C++ 命名空间或 Java 的包基本相同，但 Visual Basic 6 中没有对应的概念。`namespace` 关键字声明了应与类相关的命名空间。其后花括号中的所有代码都被认为是在这个命名空间中。编译器在 `using` 指令指定的命名空间中查找没有在当前命名空间中定义、但在代码中引用的类。这非常类似于 Java 中的 `import` 语句和 C++ 中的 `using namespace` 语句。

```
using System;
```

```
namespace Wrox.ProCSharp.Basics
{
```

在 `First.cs` 文件中使用 `using` 指令的原因是下面要使用一个库类 `System.Console`。`using System` 指令允许把这个类简写为 `Console`(类似于 `System` 命名空间中的其他类)。标准的 `System` 命名空间包含了最常用的 .NET 类型。我们用 C# 做的所有工作都依赖于 .NET 基类，认识到这一点是非常重要的；在本例中，我们使用了 `System` 命名空间中的 `Console` 类，以写入控制台窗口。C# 没有用于输入和输出的内置关键字，而是完全依赖于 .NET 类。

**注意：**

几乎所有的 C# 程序都使用 `System` 命名空间中的类，所以假定本章所有的代码文件都包含 `using System;` 语句。

接着, 声明一个类 `MyFirstClass`。但是, 因为该类位于 `Wrox.ProCSharp.Basics` 命名空间中, 所以其完整的名称是 `Wrox.ProCSharp.Basics.MyFirstCSharpClass`:

```
class MyFirstCSharpClass
{
```

与 Java 一样, 所有的 C# 代码都必须包含在一个类中, C# 中的类类似于 Java 和 C++ 中的类, 大致相当于 Visual Basic 6 中的类模块。类的声明包括 `class` 关键字, 其后是类名和一对花括号。与类相关的所有代码都应放在这对花括号中。

下面声明方法 `Main()`。每个 C# 可执行文件(例如控制台应用程序、Windows 应用程序和 Windows 服务)都必须有一个入口点——`Main` 方法(注意 M 大写):

```
static void Main()
{
```

这个方法在程序启动时调用, 类似于 C++ 和 Java 中的 `main` 函数, 或 Visual Basic 6 模块中的 `Sub Main`。该方法要么没有返回值 `void`, 要么返回一个整数(`int`)。C# 方法对应于 C++ 和 Java 中的方法(有时把 C++ 中的方法称为成员函数), 它还对应于 Visual Basic 的 `Function` 或 Visual Basic 的 `Sub`, 这取决于方法是否有返回值(与 Visual Basic 不同, C# 的函数和子例程没有概念上的区别)。

注意, C# 中的方法定义如下所示:

```
[modifiers] return_type MethodName([parameters])
{
    // Method body. NB. This code block is pseudo-code
}
```

第一个方括号中的内容表示可选关键字。修饰符(modifiers)用于指定用户所定义的方法的某些特性, 例如可以在什么地方调用该方法。在本例中, 有两个修饰符 `public` 和 `static`。修饰符 `public` 表示可以在任何地方访问该方法, 所以可以在类的外部调用它。这与 C++ 和 Java 中的 `public` 相同, 与 Visual Basic 中的 `Public` 相同。修饰符 `static` 表示方法不能在类的实例上执行, 因此不必先实例化类再调用。这是非常重要的, 因为我们创建的是一个可执行文件, 而不是类库。这与 C++ 和 Java 中的 `static` 关键字相同, 但 Visual Basic 中没有对应的关键字(在 Visual Basic 中, `Static` 关键字有不同的含义)。把返回类型设置为 `void`, 在本例中, 不包含任何参数。

最后, 看看代码语句。

```
Console.WriteLine("This isn't at all like Java!");
Console.ReadLine();
return;
```

在本例中, 我们只调用了 `System.Console` 类的 `WriteLine()` 方法, 把一行文本写到控制台窗口上。`WriteLine()` 是一个静态方法, 在调用之前不需要实例化 `Console` 对象。

`Console.ReadLine()` 读取用户的输入, 添加这行代码会让应用程序等待用户按下回车键, 之后退出应用程序。在 Visual Studio 2008 中, 控制台窗口会消失。

然后调用 `return` 退出该方法(因为这是 `Main` 方法, 所以也退出了应用程序)。在方法的首部指定 `void`, 因此没有返回值。`Return` 语句等价于 C++ 和 Java 中的 `return`, 也等价于 Visual Basic 中的 `Exit Sub` 或 `Exit Function`。

对 C#基本语法有了大致的认识后，下面就要详细讨论 C#的各个方面了。因为没有变量是不可能编写出重要的程序的，所以首先介绍 C#中的变量。

## 2.3 变量

在 C#中声明变量使用下述语法：

```
datatype identifier;
```

例如：

```
int i;
```

该语句声明 int 变量 i。编译器不会让我们使用这个变量，除非我们用一个值初始化了该变量。

声明 i 之后，就可以使用赋值运算符(=)给它分配一个值：

```
i = 10;
```

还可以在一行代码中声明变量，并初始化它的值：

```
int i = 10;
```

其语法与 C++和 Java 语法相同，但与 Visual Basic 中声明变量的语法完全不同。如果用户以前使用的是 Visual Basic 6，应记住 C#不区分对象和简单的类型，所以不需要类似 Set 的关键字，即使是要把变量指向一个对象，也不需要 Set 关键字。无论变量的数据类型是什么，声明变量的 C#语法都是相同的。

如果在一个语句中声明和初始化了多个变量，那么所有的变量都具有相同的数据类型：

```
int x = 10, y = 20; // x and y are both ints
```

要声明类型不同的变量，需要使用单独的语句。在多个变量的声明中，不能指定不同的数据类型：

```
int x = 10;  
bool y = true; // Creates a variable that stores true or false  
int x = 10, bool y = true; // This won't compile!
```

注意上面例子中的//和其后的文本，它们是注释。//字符串告诉编译器，忽略其后的文本，这些文本仅为了让人们更好地理解程序，它们并不是程序的一部分。本章后面会详细讨论代码中的注释。

### 2.3.1 变量的初始化

变量的初始化是 C#强调安全性的另一个例子。简单地说，C#编译器需要用某个初始值对变量进行初始化，之后才能在操作中引用该变量。大多数现代编译器把没有初始化标记为警告，但 C#编译器把它当作错误来看待。这就可以防止我们无意中从其他程序遗留下来的内存中获取垃圾值。

C#有两个方法可确保变量在使用前进行了初始化：



- 变量是类或结构中的字段，如果没有显式初始化，创建这些变量时，其值就默认是 0(类和结构在后面讨论)。
- 方法的局部变量必须在代码中显式初始化，之后才能在语句中使用它们的值。此时，初始化不是在声明该变量时进行的，但编译器会通过方法检查所有可能的路径，如果检测到局部变量在初始化之前就使用了它的值，就会产生错误。

C#的方法与 C++的方法相反，在 C++中，编译器让程序员确保变量在使用之前进行了初始化，在 Visual Basic 中，所有的变量都会自动把其值设置为 0。

例如，在 C#中不能使用下面的语句：

```
public static int Main()
{
    int d;
    Console.WriteLine(d); // Can't do this! Need to initialize d before use
    return 0;
}
```

注意在这段代码中，演示了如何定义 Main()，使之返回一个 int 类型的数据，而不是 void。在编译这些代码时，会得到下面的错误消息：

```
Use of unassigned local variable 'd'
```

考虑下面的语句：

```
Something objSomething;
```

在 C++中，上面的代码会在堆栈中创建 Something 类的一个实例。在 C#中，这行代码仅会为 Something 对象创建一个引用，但这个引用还没有指向任何对象。对该变量调用方法或属性会导致错误。

在 C#中实例化一个引用对象需要使用 new 关键字。如上所述，创建一个引用，使用 new 关键字把该引用指向存储在堆上的一个对象：

```
objSomething = new Something(); // This creates a Something on the heap
```

### 2.3.2 类型推断

类型推断使用 var 关键字。声明变量的语法有些变化。编译器可以根据变量的初始化值“推断”变量的类型。例如：

```
int someNumber = 0;
```

就变成：

```
var someNumber = 0;
```

即使 someNumber 从来没有声明为 int，编译器也可以确定，只要 someNumber 在其作用域内，就是一个 int。编译后，上面两个语句是等价的。

下面是另一个小例子：

```
using System;
```

```

namespace Wrox.ProCSharp.Basics
{
    class Program
    {
        static void Main(string[] args)
        {
            var name = "Bugs Bunny";
            var age = 25;
            var isRabbit = true;

            Type nameType = name.GetType();
            Type ageType = age.GetType();
            Type isRabbitType = isRabbit.GetType();

            Console.WriteLine("name is type " + nameType.ToString());
            Console.WriteLine("age is type " + ageType.ToString());
            Console.WriteLine("isRabbit is type " + isRabbitType.ToString());
        }
    }
}

```

这个程序的输出如下：

```

name is type System.String
age is type System.Int32
isRabbit is type System.Bool

```

需要遵循一些规则。变量必须初始化。否则，编译器就没有推断变量类型的依据。初始化器不能为空，且必须放在表达式中。不能把初始化器设置为一个对象，除非在初始化器中创建了一个新对象。第 3 章在讨论匿名类型时将详细探讨。

声明了变量，推断出了类型后，变量的类型就不能改变了。这与 Visual Basic 中使用的 Variant 类型不同。变量的类型建立后，就遵循其他变量类型遵循的强类型化规则。

### 2.3.3 变量的作用域

变量的作用域是可以访问该变量的代码区域。一般情况下，确定作用域有以下规则：

- 只要类在某个作用域内，其字段(也称为成员变量)也在该作用域内(在 C++、Java 和 Visual Basic 中也是这样)。
- 局部变量存在于表示声明该变量的块语句或方法结束的封闭花括号之前的作用域内。
- 在 for、while 或类似语句中声明的局部变量存在于该循环体内(C++程序员注意，这与 C++的 ANSI 标准相同。Microsoft C++编译器的早期版本不遵守该标准，在循环停止后这种变量仍存在)。

#### 1. 局部变量的作用域冲突

大型程序在不同部分为不同的变量使用相同的变量名是很常见的。只要变量的作用域是程序的不同部分，就不会有问题，也不会产生模糊性。但要注意，同名的局部变量不能在同一作

用域内声明两次，所以不能使用下面的代码：

```
int x = 20;
// some more code
int x = 30;
```

考虑下面的代码示例：

```
using System;

namespace Wrox.ProCSharp.Basics
{
    public class ScopeTest
    {
        public static int Main()
        {
            for (int i = 0; i < 10; i++)
            {
                Console.WriteLine(i);
            } // i goes out of scope here

            // We can declare a variable named i again, because
            // there's no other variable with that name in scope

            for (int i = 9; i >= 0; i--)
            {
                Console.WriteLine(i);
            } // i goes out of scope here
            return 0;
        }
    }
}
```

这段代码使用 2 个 for 循环打印出从 0~9 的数字，再打印从 9~0 的数字。重要的是在同一个方法中，代码中的变量 *i* 声明了两次。可以这么做的原因是在两次声明中，*i* 都是在循环内部声明的，所以变量 *i* 对于循环来说是局部变量。

下面是另一个例子：

```
public static int Main()
{
    int j = 20;
    for (int i = 0; i < 10; i++)
    {
        int j = 30; // Can't do this - j is still in scope
        Console.WriteLine(j + i);
    }
    return 0;
}
```

如果试图编译它，就会产生如下错误：

```
ScopeTest.cs(12,14): error CS0136: A local variable named 'j' cannot be declared in this scope because it would give a different meaning to 'j', which is already used in a 'parent or current' scope to denote something else
```

其原因是：变量 *j* 是在 for 循环开始前定义的，在执行 for 循环时应处于其作用域内，在 Main 方法结束执行后，变量 *j* 才超出作用域，第二个 *j*(不合法)则在循环的作用域内，该作用域嵌套在 Main 方法的作用域内。编译器无法区别这两个变量，所以不允许声明第二个变量。这也是与 C++ 不同的地方，在 C++ 中，允许隐藏变量。

## 2. 字段和局部变量的作用域冲突

在某些情况下，可以区分名称相同(尽管其完全限定的名称不同)、作用域相同的两个标识符。此时编译器允许声明第二个变量。原因是 C# 在变量之间有一个基本的区分，它把声明为类型级的变量看作字段，而把在方法中声明的变量看作局部变量。

考虑下面的代码：

```
using System;

namespace Wrox.ProCSharp.Basics
{
    class ScopeTest2
    {
        static int j = 20;
        Console.WriteLine(j);

        public static void Main()
        {
            int j = 30;
            Console.WriteLine(j);
            return;
        }
    }
}
```

即使在 `Main` 方法的作用域内声明了两个变量 `j`，这段代码也会编译——`j` 被定义在类级上，在该类删除前是不会超出作用域的(在本例中，当 `Main` 方法中断，程序结束时，才会删除该类)。此时，在 `Main` 方法中声明的新变量 `j` 隐藏了同名的类级变量，所以在运行这段代码时，会显示数字 30。

但是，如果要引用类级变量，该怎么办？可以使用语法 `object.fieldname`，在对象的外部引用类的字段或结构。在上面的例子中，我们访问静态方法中的一个静态字段(静态字段详见下一节)，所以不能使用类的实例，只能使用类本身的名称：

```
...
public static void Main()
{
    int j = 30;
    Console.WriteLine(j);
    Console.WriteLine(ScopeTest2.j);
}
...
```

如果要访问一个实例字段(该字段属于类的一个特定实例)，就需要使用 `this` 关键字。`this` 的作用与 C++ 和 Java 中的 `this` 相同，与 Visual Basic 中的 `Me` 相同。

### 2.3.4 常量

顾名思义，常量是其值在使用过程中不会发生变化的变量。在声明和初始化变量时，在变量的前面加上关键字 `const`，就可以把该变量指定为一个常量：

```
const int a = 100; // This value cannot be changed
```

Visual Basic 和 C++ 开发人员非常熟悉常量。但 C++ 开发人员应注意，C# 不支持 C++ 常量的所有细微的特性。在 C++ 中，变量不仅可以声明为常量，而且根据声明，还可以有常量指针、指向常量的变量指针、常量方法(不改变包含对象的内容)、方法的常量参数等。这些细微的特性在 C# 中都删除了，只能把局部变量和字段声明为常量。

常量具有如下特征：

- 常量必须在声明时初始化。指定了其值后，就不能再修改了。
- 常量的值必须能在编译时用于计算。因此，不能用从一个变量中提取的值来初始化常量。如果需要这么做，应使用只读字段(详见第 3 章)。
- 常量总是静态的。但注意，不必(实际上，是不允许)在常量声明中包含修饰符 `static`。

在程序中使用常量至少有 3 个好处：

- 常量用易于理解的清楚的名称替代了含义不明确的数字或字符串，使程序更易于阅读。
- 常量使程序更易于修改。例如，在 C# 程序中有一个 `SalesTax` 常量，该常量的值为 6%。如果以后销售税率发生变化，把新值赋给这个常量，就可以修改所有的税款计算结果，而不必查找整个程序，修改税率为 0.06 的每个项。
- 常量更容易避免程序出现错误。如果要把另一个值赋给程序中的一个常量，而该常量已经有了一个值，编译器就会报告错误。

## 2.4 预定义数据类型

前面介绍了如何声明变量和常量，下面要详细讨论 C# 中可用的数据类型。与其他语言相比，C# 对其可用的类型及其定义有更严格的描述。

### 2.4.1 值类型和引用类型

在开始介绍 C# 中的数据类型之前，理解 C# 把数据类型分为两种是非常重要的：

- 值类型
- 引用类型

下面几节将详细介绍值类型和引用类型的语法。从概念上看，其区别是值类型直接存储其值，而引用类型存储对值的引用。C# 中的值类型基本上等价于 Visual Basic 或 C++ 中的简单类型(整型、浮点型，但没有指针或引用)。引用类型与 Visual Basic 中的引用类型相同，与 C++ 中通过指针访问的类型类似。

这两种类型存储在内存的不同地方：值类型存储在堆栈中，而引用类型存储在托管堆上。注意区分某个类型是值类型还是引用类型，因为这种存储位置的不同会有不同的影响。例如，`int` 是值类型，这表示下面的语句会在内存的两个地方存储值 20：

```
// i and j are both of type int
i = 20;
j = i;
```



但考虑下面的代码。这段代码假定已经定义了一个类 `Vector`，`Vector` 是一个引用类型，它有一个 `int` 类型的成员变量 `Value`：

```
Vector x, y;  
x = new Vector ();  
x.Value = 30; // Value is a field defined in Vector class  
y = x;  
Console.WriteLine(y.Value);  
y.Value = 50;  
Console.WriteLine(x.Value);
```

要理解的重要一点是在执行这段代码后，只有一个 `Vector` 对象。`x` 和 `y` 都指向包含该对象的内存位置。因为 `x` 和 `y` 是引用类型的变量，声明这两个变量只保留了一个引用——而不会实例化给定类型的对象。这与在 C++ 中声明指针和 Visual Basic 中的对象引用是相同的——在 C++ 和 Visual Basic 中，都不会创建对象。要创建对象，就必须使用 `new` 关键字，如上所示。因为 `x` 和 `y` 引用同一个对象，所以对 `x` 的修改会影响 `y`，反之亦然。因此上面的代码会显示 30 和 50。

#### 注意：

C++ 开发人员应注意，这个语法类似于引用，而不是指针。我们使用 `.` (句点) 符号，而不是 `->` 来访问对象成员。在语法上，C# 引用看起来更类似于 C++ 引用变量。但是，抛开表面的语法，实际上它类似于 C++ 指针。

如果变量是一个引用，就可以把其值设置为 `null`，表示它不引用任何对象：

```
y = null;
```

这类似于 Java 中把引用设置为 `null`，C++ 中把指针设置为 `NULL`，或 Visual Basic 中把对象引用设置为 `Nothing`。如果将引用设置为 `null`，显然就不可能对它调用任何非静态的成员函数或字段，这么做会在运行期间抛出一个异常。

在像 C++ 这样的语言中，开发人员可以选择是直接访问某个给定的值，还是通过指针来访问。Visual Basic 的限制更多：COM 对象是引用类型，简单类型总是值类型。C# 在这方面类似于 Visual Basic：变量是值还是引用仅取决于其数据类型，所以，`int` 总是值类型。不能把 `int` 变量声明为引用(在第 6 章介绍装箱时，可以在类型为 `object` 的引用中封装值类型)。

在 C# 中，基本数据类型如 `bool` 和 `long` 都是值类型。如果声明一个 `bool` 变量，并给它赋予另一个 `bool` 变量的值，在内存中就会有俩个 `bool` 值。如果以后修改第一个 `bool` 变量的值，第二个 `bool` 变量的值也不会改变。这些类型是通过值来复制的。

相反，大多数更复杂的 C# 数据类型，包括我们自己声明的类都是引用类型。它们分配在堆中，其生存期可以跨多个函数调用，可以通过一个或几个别名来访问。CLR 执行一种精细的算法，来跟踪哪些引用变量仍是可以访问的，哪些引用变量已经不能访问了。CLR 会定期删除不能访问的对象，把它们占用的内存返回给操作系统。这是通过垃圾收集器实现的。

把基本类型(如 `int` 和 `bool`)规定为值类型，而把包含许多字段的较大类型(通常在有类的情況下)规定为引用类型，C# 设计这种方式的原因是可以得到最佳性能。如果要把自己的类型定义为值类型，就应把它声明为一个结构。

2.4.2 CTS 类型

如第 1 章所述，C#认可的基本预定义类型并没有内置于 C#语言中，而是内置于.NET Framework 中。例如，在 C#中声明一个 int 类型的数据时，声明的实际上是.NET 结构 System.Int32 的一个实例。这听起来似乎很深奥，但其意义深远：这表示在语法上，可以把所有的基本数据类型看作是支持某些方法的类。例如，要把 int i 转换为 string，可以编写下面的代码：

```
string s = i.ToString();
```

应强调的是，在这种便利语法的背后，类型实际上仍存储为基本类型。基本类型在概念上用.NET 结构表示，所以肯定没有性能损失。

下面看看 C#中定义的类型。我们将列出每个类型，以及它们的定义和对应.NET 类型(CTS 类型)的名称。C#有 15 个预定义类型，其中 13 个是值类型，2 个是引用类型(string 和 object)。

2.4.3 预定义的值类型

内置的值类型表示基本数据类型，例如整型和浮点类型、字符类型和布尔类型。

1. 整型

C#支持 8 个预定义整数类型，如表 2-1 所示。

表 2-1

名 称	CTS 类 型	说 明	范 围
sbyte	System.SByte	8 位有符号的整数	-128~127 ( $-2^7 \sim 2^7 - 1$ )
short	System.Int16	16 位有符号的整数	-32 768~32 767 ( $-2^{15} \sim 2^{15} - 1$ )
int	System.Int32	32 位有符号的整数	-2 147 483 648~2 147 483 647 ( $-2^{31} \sim 2^{31} - 1$ )
long	System.Int64	64 位有符号的整数	-9 223 372 036 854 775 808~9 223 372 036 854 775 807 ( $-2^{63} \sim 2^{63} - 1$ )
byte	System.Byte	8 位无符号的整数	0~255 ( $0 \sim 2^8 - 1$ )
ushort	System.UInt16	16 位无符号的整数	0~65535 ( $0 \sim 2^{16} - 1$ )
uint	System.UInt32	32 位无符号的整数	0~4 294 967 295 ( $0 \sim 2^{32} - 1$ )
ulong	System.UInt64	64 位无符号的整数	0~18 446 744 073 709 551 615 ( $0 \sim 2^{64} - 1$ )

Windows 的将来版本将支持 64 位处理器，可以把更大的数据块移入移出内存，获得更快的处理速度。因此，C#支持 8~64 位的有符号和无符号的整数。

当然，Visual Basic 开发人员会发现有许多类型名称是新的。C++和 Java 开发人员应注意：一些 C#类型的名称与 C++和 Java 类型一致，但其定义不同。例如，在 C#中，int 总是 32 位带符号的整数，而在 C++中，int 是带符号的整数，但其位数取决于平台(在 Windows 上是 32 位)。在 C#中，所有的数据类型都以与平台无关的方式定义，以备将来 C#和.NET 迁移到其他平台上。

byte 是 0~255(包括 255)的标准 8 位类型。注意，在强调类型的安全性时，C#认为 byte 类型和 char 类型完全不同，它们之间的编程转换必须显式写出。还要注意，与整数中的其他类型

不同，byte 类型在默认状态下是无符号的，其有符号的版本有一个特殊的名称 sbyte。

在.NET 中，short 不再很短，现在它有 16 位，Int 类型更长，有 32 位。long 类型最长，有 64 位。所有整数类型的变量都能赋予十进制或十六进制的值，后者需要 0x 前缀：

```
long x = 0x12ab;
```

如果对一个整数是 int、uint、long 或是 ulong 没有任何显式的声明，则该变量默认为 int 类型。为了把键入的值指定为其他整数类型，可以在数字后面加上如下字符：

```
uint ui = 1234U;
long l = 1234L;
ulong ul = 1234UL;
```

也可以使用小写字母 u 和 l，但后者会与整数 1 混淆。

2. 浮点类型

C#提供了许多整型数据类型，也支持浮点类型，如表 2-2 所示。C 和 C++程序员很熟悉它们。

表 2-2

名 称	CTS 类 型	说 明	位 数	范围(大致)
float	System.Single	32 位单精度浮点数	7	$\pm 1.5 \times 10^{-45} \sim \pm 3.4 \times 10^{38}$
double	System.Double	64 位双精度浮点数	15/16	$\pm 5.0 \times 10^{-324} \sim \pm 1.7 \times 10^{308}$

float 数据类型用于较小的浮点数，因为它要求的精度较低。double 数据类型比 float 数据类型大，提供的精度也大一倍(15 位)。

如果在代码中没有对某个非整数值(如 12.3)硬编码，则编译器一般假定该变量是 double。如果想指定该值为 float，可以在其后加上字符 F(或 f)：

```
float f = 12.3F;
```

3. decimal 类型

另外，decimal 类型表示精度更高的浮点数，如表 2-3 所示。

表 2-3

名 称	CTS 类 型	说 明	位 数	范围(大致)
decimal	System.Decimal	128 位高精度十进制数表示法	28	$\pm 1.0 \times 10^{-28} \sim \pm 7.9 \times 10^{28}$

CTS 和 C#一个重要的优点是提供了一种专用类型进行财务计算，这就是 decimal 类型，使用 decimal 类型提供的 28 位的方式取决于用户。换言之，可以用较大的精确度(带有美分)来表示较小的美元值，也可以在小数部分用更多的舍入来表示较大的美元值。但应注意，decimal 类型不是基本类型，所以在计算时使用该类型会有性能损失。

要把数字指定为 decimal 类型，而不是 double、float 或整型，可以在数字的后面加上字符 M(或 m)，如下所示。

```
decimal d = 12.30M;
```

4. bool 类型

C#的 bool 类型用于包含布尔值 true 或 false，如表 2-4 所示。

表 2-4

名 称	CTS 类 型	说 明	位 数	值
bool	System.Boolean	表示 true 或 false	NA	true 或 false

bool 值和整数值不能相互隐式转换。如果变量(或函数的返回类型)声明为 bool 类型，就只能使用值 true 或 false。如果试图使用 0 表示 false，非 0 值表示 true，就会出错。这与 C++相同。

5. 字符类型

为了保存单个字符的值，C#支持 char 数据类型，如表 2-5 所示。

表 2-5

名 称	CTS 类 型	值
char	System.Char	表示一个 16 位的(Unicode)字符

虽然这个数据类型在表面上类似于 C 和 C++中的 char 类型，但它们有重大区别。C++的 char 表示一个 8 位字符，而 C#的 char 包含 16 位。其部分原因是不允许在 char 类型与 8 位 byte 类型之间进行隐式转换。

尽管 8 位足够编码英语中的每个字符和数字 0~9 了，但它们不够编码更大的符号系统中的每个字符(例如中文)。为了面向全世界，计算机行业正在从 8 位字符集转向 16 位的 Unicode 模式，ASCII 编码是 Unicode 的一个子集。

char 类型的字面量是用单引号括起来的，例如'A'。如果把字符放在双引号中，编译器会把它看作是字符串，从而产生错误。

除了把 char 表示为字符字面量之外，还可以用 4 位十六进制的 Unicode 值(例如'\u0041')，带有数据类型转换的整数值(例如(char)65)，或十六进制数('\x0041')表示它们。它们还可以用转义序列表示，如表 2-6 所示。

表 2-6

转 义 序 列	字 符
\'	单引号
\"	双引号
\\	反斜杠
\0	空
\a	警告
\b	退格

(续表)

转义序列	字 符
\f	换页
\n	换行
\r	回车
\t	水平制表符
\v	垂直制表符

C++开发人员应注意，因为 C#本身有一个 string 类型，所以不需要把字符串表示为 char 类型的数组。

2.4.4 预定义的引用类型

C#支持两个预定义的引用类型，如表 2-7 所示。

表 2-7

名 称	CTS 类	说 明
object	System.Object	根类型，CTS 中的其他类型都是从它派生而来的(包括值类型)
string	System.String	Unicode 字符串

1. object 类型

许多编程语言和类结构都提供了根类型，层次结构中的其他对象都从它派生而来。C# 和.NET 也不例外。在 C#中，object 类型就是最终的父类型，所有内置类型和用户定义的类型都从它派生而来。这是 C#的一个重要特性，它把 C#与 Visual Basic 6.0 和 C++区分开来，但其行为与 Java 非常类似。所有的类型都隐含地最终派生于 System.Object 类，这样，object 类型就可以用于两个目的：

- 可以使用 object 引用绑定任何子类型的对象。例如，第 6 章将说明如何使用 object 类型把堆栈中的一个值对象装箱，再移动到堆中。object 引用也可以用于反射，此时必须有代码来处理类型未知的对象。这类似于 C++中的 void 指针或 Visual Basic 中的 Variant 数据类型。
- object 类型执行许多一般用途的基本方法，包括 Equals()、GetHashCode()、GetType() 和 ToString()。用户定义类需要使用一种面向对象技术——重写(见第 4 章)，提供其中一些方法的替代执行代码。例如，重写 ToString()时，要给类提供一个方法，给出类本身的字符串表示。如果类中没有提供这些方法的实现代码，编译器就会使用 object 类型中的实现代码，它们在类中的执行不一定正确。

后面的章节将详细讨论 object 类型。



## 2. string 类型

有 C 和 C++ 开发经验的人员可能在使用 C 风格的字符串时不太顺利。C 或 C++ 字符串不过是一个字符数组，因此客户机程序员必须做许多工作，才能把一个字符串复制到另一个字符串上，或者连接两个字符串。实际上，对于一般的 C++ 程序员来说，执行包装了这些操作细节的字符串类是一个非常头痛的耗时过程。Visual Basic 程序员的工作就比较简单，只需使用 `string` 类型即可。而 Java 程序员就更幸运了，其 `String` 类在许多方面都类似于 C# 字符串。

C# 有 `string` 关键字，在翻译为 .NET 类时，它就是 `System.String`。有了它，像字符串连接和字符串复制这样的操作就很简单了：

```
string str1 = "Hello ";
string str2 = "World";
string str3 = str1 + str2; // string concatenation
```

尽管这是一个值类型的赋值，但 `string` 是一个引用类型。`String` 对象保留在堆上，而不是堆栈上。因此，当把一个字符串变量赋给另一个字符串时，会得到对内存中同一个字符串的两个引用。但是，`string` 与引用类型在常见的操作上有一些区别。例如，修改其中一个字符串，就会创建一个全新的 `string` 对象，而另一个字符串没有改变。考虑下面的代码：

```
using System;

class StringExample
{
    public static int Main()
    {
        string s1 = "a string";
        string s2 = s1;
        Console.WriteLine("s1 is " + s1);
        Console.WriteLine("s2 is " + s2);
        s1 = "another string";
        Console.WriteLine("s1 is now " + s1);
        Console.WriteLine("s2 is now " + s2);
        return 0;
    }
}
```

其输出结果为：

```
s1 is a string
s2 is a string
s1 is now another string
s2 is now a string
```

换言之，改变 `s1` 的值对 `s2` 没有影响，这与我们期待的引用类型正好相反。当用值 "a string" 初始化 `s1` 时，就在堆上分配了一个新的 `string` 对象。在初始化 `s2` 时，引用也指向这个对象，所以 `s2` 的值也是 "a string"。但是现在要改变 `s1` 的值，而不是替换原来的值时，堆上就会为新值分配一个新对象。`s2` 变量仍指向原来的对象，所以它的值没有改变。这实际上是运算符重载的结果，运算符重载详见第 6 章。基本上，`string` 类实现为其语义遵循一般的、直观的字符串规则。

字符串字面量放在双引号中("..."); 如果试图把字符串放在单引号中, 编译器就会把它当作 `char`, 从而引发错误。C# 字符串和 `char` 一样, 可以包含 Unicode、十六进制数转义序列。因为这些转义序列以一个反斜杠开头, 所以不能在字符串中使用这个非转义的反斜杠字符, 而需要用两个反斜杠字符(`\\`)来表示它:

```
string filepath = "C:\\ProCSharp\\First.cs";
```

即使用户相信自己可以在任何情况下都记住要这么做, 但键入两个反斜杠字符会令人迷惑。幸好, C# 提供了另一种替代方式。可以在字符串字面量的前面加上字符 `@`, 在这个字符后的所有字符都看作是其原来的含义——它们不会解释为转义字符:

```
string filepath = @"C:\ProCSharp\First.cs";
```

甚至允许在字符串字面量中包含换行符:

```
string jabberwocky = @'''Twas brillig and the slithy toves  
Did gyre and gimble in the wabe.'';
```

那么 `jabberwocky` 的值就是:

```
'Twas brillig and the slithy toves  
Did gyre and gimble in the wabe.
```

## 2.5 流控制

本节将介绍 C# 语言的重要语句: 控制程序流的语句, 它们不是按代码在程序中的排列位置顺序执行的。

### 2.5.1 条件语句

条件语句可以根据条件是否满足或根据表达式的值控制代码的执行分支。C# 有两个控制代码分支的结构: `if` 语句, 测试特定条件是否满足; `switch` 语句, 它比较表达式和许多不同的值。

#### 1. `if` 语句

对于条件分支, C# 继承了 C 和 C++ 的 `if...else` 结构。对于用过程语言编程的人来说, 其语法是非常直观的:

```
if (condition)
    statement(s)
else
    statement(s)
```

如果在条件中要执行多个语句, 就需要用花括号(`{ ... }`)把这些语句组合为一个块(这也适用于其他可以把语句组合为一个块的 C# 结构, 例如 `for` 和 `while` 循环)。

```
bool isZero;
if (i == 0)
{
    isZero = true;
```

```

        Console.WriteLine("i is Zero");
    }
    else
    {
        isZero = false;
        Console.WriteLine("i is Non-zero");
    }
}

```

其语法与 C++ 和 Java 类似，但与 Visual Basic 不同。Visual Basic 开发人员注意，C# 中没有与 Visual Basic 的 EndIf 对应的语句，其规则是 if 的每个子句都只包含一个语句。如果需要多个语句，如上面的例子所示，就应把这些语句放在花括号中，这会把整组语句当作一个语句块来处理。

还可以单独使用 if 语句，不加 else 语句。也可以合并 else if 子句，测试多个条件。

```
using System;
```

```
namespace Wrox.ProCSharp.Basics
```

```
{
```

```
    class MainEntryPoint
```

```
    {
```

```
        static void Main(string[] args)
```

```
        {
```

```
            Console.WriteLine("Type in a string");
```

```
            string input;
```

```
            input = Console.ReadLine();
```

```
            if (input == "")
```

```
            {
```

```
                Console.WriteLine("You typed in an empty string");
```

```
            }
```

```
            else if (input.Length < 5)
```

```
            {
```

```
                Console.WriteLine("The string had less than 5 characters");
```

```
            }
```

```
            else if (input.Length < 10)
```

```
            {
```

```
                Console.WriteLine("The string had at least 5 but less than 10 characters");
```

```
            }
```

```
            Console.WriteLine("The string was " + input);
```

```
        }
```

```
    }
```

```
}
```

添加到 if 子句中的 else if 语句的个数没有限制。

注意在上面的例子中，我们声明了一个字符串变量 input，让用户在命令行上输入文本，把文本填充到 input 中，然后测试该字符串变量的长度。代码还说明了在 C# 中如何进行字符串处理。例如，要确定 input 的长度，可以使用 input.Length。

对于 if，要注意的一点是如果条件分支中只有一条语句，就无需使用花括号：

```

if (i == 0) Let's add some brackets here.
    Console.WriteLine("i is Zero");           // This will only execute if i == 0
Console.WriteLine("i can be anything"); // Will execute whatever the
                                         // value of i

```

但是，为了保持一致，许多程序员只要使用 if 语句，就加上花括号。

前面介绍的 if 语句还演示了用于比较数值的一些 C# 运算符。特别注意，与 C++ 和 Java 一样，C# 使用 “==” 对变量进行等于比较。此时不要使用 “=”，“=” 用于赋值。

在 C# 中，if 子句中的表达式必须等于布尔值。C++ 程序员应特别注意这一点；与 C++ 不同，C# 中的 if 语句不能直接测试整数（例如从函数中返回的值），而必须明确地把返回的整数转换为布尔值 true 或 false，例如，比较值 0 和 null：

```
if (DoSomething() != 0)
{
    // Non-zero value returned
}
else
{
    // Returned zero
}
```

这个限制用于防止 C++ 中某些常见的运行错误，特别是在 C++ 中，当应使用 “==” 时，常常误输入 “=”，导致不希望的赋值。在 C# 中，这常常会导致一个编译错误，因为除非在处理 bool 值，否则 “=” 不会返回 bool。

## 2. switch 语句

switch...case 语句适合于从一组互斥的分支中选择一个执行分支。C++ 和 Java 程序员应很熟悉它，该语句类似于 Visual Basic 中的 Select Case 语句。

其形式是 switch 参数的后面跟一组 case 子句。如果 switch 参数中表达式的值等于某个 case 子句旁边的某个值，就执行该 case 子句中的代码。此时不需要使用花括号把语句组合到块中；只需使用 break 语句标记每个 case 代码的结尾即可。也可以在 switch 语句中包含一个 default 子句，如果表达式不等于任何 case 子句的值，就执行 default 子句的代码。下面的 switch 语句测试 integerA 变量的值：

```
switch (integerA)
{
    case 1:
        Console.WriteLine("integerA =1");
        break;
    case 2:
        Console.WriteLine("integerA =2");
        break;
    case 3:
        Console.WriteLine("integerA =3");
        break;
    default:
        Console.WriteLine("integerA is not 1,2, or 3");
        break;
}
```

注意 case 的值必须是常量表达式——不允许使用变量。

C 和 C++ 程序员应很熟悉 switch...case 语句，而 C# 的 switch...case 语句更安全。特别是它禁止所有 case 中的失败条件。如果激活了块中靠前的一个 case 子句，后面的 case 子句就不会被激活，除非使用 goto 语句特别标记要激活后面的 case 子句。编译器会把没有 break 语句的 case 子句标记为错误：



Control cannot fall through from one case label ('case 2:') to another

在有限的几种情况下,这种失败是允许的,但在大多数情况下,我们不希望出现这种失败,而且这会导致出现很难察觉的逻辑错误。让代码正常工作,而不是出现异常,这样不是更好吗?

但在使用 goto 语句时,会在 switch...cases 中重复出现失败。如果确实想这么做,就应重新考虑设计方案了。下面的代码说明了如何使用 goto 模拟失败,得到的代码会非常混乱:

```
// assume country and language are of type string
switch(country)
{
    case "America":
        CallAmericanOnlyMethod();
        goto case "Britain";
    case "France":
        language = "French";
        break;
    case "Britain":
        language = "English";
        break;
}
```

但这有一种例外情况。如果一个 case 子句为空,就可以从这个 case 跳到下一个 case 上,这样就可以用相同的方式处理两个或多个 case 子句了(不需要 goto 语句)。

```
switch(country)
{
    case "au":
    case "uk":
    case "us":
        language = "English";
        break;
    case "at":
    case "de":
        language = "German";
        break;
}
```

在 C# 中, switch 语句的一个有趣的地方是 case 子句的排放顺序是无关紧要的,甚至可以把 default 子句放在最前面!因此,任何两个 case 都不能相同。这包括值相同的不同常量,所以不能这样编写:

```
// assume country is of type string
const string england = "uk";
const string britain = "uk";
switch(country)
{
    case england:
    case britain: // this will cause a compilation error
        language = "English";
        break;
}
```

上面的代码还说明了 C# 中的 switch 语句与 C++ 中的 switch 语句的另一个不同之处:在 C# 中,可以把字符串用作测试变量。



## 2.5.2 循环

C#提供了 4 种不同的循环机制(for、while、do...while 和 foreach)，在满足某个条件之前，可以重复执行代码块。for、while 和 do...while 循环与 C++中的对应循环相同。

### 1. for 循环

C#的 for 循环提供的迭代循环机制是在执行下一次迭代前，测试是否满足某个条件，其语法如下：

```
for (initializer; condition; iterator)
    statement(s)
```

其中：

- **initializer** 是指在执行第一次迭代前要计算的表达式(通常把一个局部变量初始化为循环计数器)；
- **condition** 是在每次迭代新循环前要测试的表达式(它必须等于 true，才能执行下一次迭代)；
- **iterator** 是每次迭代完要计算的表达式(通常是递增循环计数器)。当 condition 等于 false 时，迭代停止。

for 循环是所谓的预测试循环，因为循环条件是在执行循环语句前计算的，如果循环条件为假，循环语句就根本不会执行。

for 循环非常适合于一个语句或语句块重复执行预定的次数。下面的例子就是 for 循环的典型用法，这段代码输出从 0~99 的整数：

```
for (int i = 0; i < 100; i = i+1) // this is equivalent to
    // For i = 0 To 99 in VB.
{
    Console.WriteLine(i);
}
```

这里声明了一个 int 类型的变量 i，并把它初始化为 0，用作循环计数器。接着测试它是否小于 100。因为这个条件等于 true，所以执行循环中的代码，显示值 0。然后给该计数器加 1，再次执行该过程。当 i 等于 100 时，循环停止。

实际上，上述编写循环的方式并不常用。C#在给变量加 1 时有一种简化方式，即不使用 i = i+1，而简写为 i++：

```
for (int i = 0; i < 100; i++)
{
    //etc.}
```

C#的 for 循环语法比 Visual Basic 中的 For...Next 循环的功能强大得多，因为迭代器可以是任何语句。在 Visual Basic 中，只能对循环控制变量加减某个数字。在 C#中，则可以做任何事，例如，让循环控制变量乘以 2。

也可以在上面的例子中给循环变量 i 使用类型推断功能。使用类型推断功能时，循环结构变成：

```
for (var i = 0; i < 100; i++)
```

```
...
```

嵌套的 for 循环非常常见，在每次迭代外部的循环时，内部循环都要彻底执行完毕。这种模式通常用于在矩形多维数组中遍历每个元素。最外部的循环遍历每一行，内部的循环遍历某行上的每个列。下面的代码显示数字行，它还使用另一个 Console 方法 Console.Write()，该方法的作用与 Console.WriteLine() 相同，但不在输出中添加回车换行符：

```
using System;
```

```
namespace Wrox.ProCSharp.Basics
```

```
{
```

```
    class MainEntryPoint
```

```
    {
```

```
        static void Main(string[] args)
```

```
        {
```

```
            // This loop iterates through rows...
```

```
            for (int i = 0; i < 100; i+=10)
```

```
            {
```

```
                // This loop iterates through columns...
```

```
                for (int j = i; j < i + 10; j++)
```

```
                {
```

```
                    Console.Write(" " + j);
```

```
                }
```

```
                Console.WriteLine();
```

```
            }
```

```
        }
```

```
    }
```

```
}
```

尽管 j 是一个整数，但它会自动转换为字符串，以便进行连接。C++ 开发人员要注意，这比在 C++ 中处理字符串容易得多，Visual Basic 开发人员则已经习惯于此了。

C 和 C++ 程序员应注意上述例子中的一个特殊功能。在每次迭代外部的循环时，内部循环的计数器变量都要重新声明。这种语法不仅在 C# 中可行，在 C++ 中也是合法的。

上述例子的结果是：

**csc NumberTable.cs**

Microsoft (R) Visual C# .NET Compiler version 9.00.20404

for Microsoft (R) .NET Framework version 3.5

Copyright (C) Microsoft Corporation. All rights reserved.

```
0 1 2 3 4 5 6 7 8 9
```

```
10 11 12 13 14 15 16 17 18 19
```

```
20 21 22 23 24 25 26 27 28 29
```

```
30 31 32 33 34 35 36 37 38 39
```

```
40 41 42 43 44 45 46 47 48 49
```

```
50 51 52 53 54 55 56 57 58 59
```

```
60 61 62 63 64 65 66 67 68 69
```

```
70 71 72 73 74 75 76 77 78 79
```

```
80 81 82 83 84 85 86 87 88 89
90 91 92 93 94 95 96 97 98 99
```

尽管在技术上，可以在 for 循环的测试条件中计算其他变量，而不计算计数器变量，但这不太常见。也可以在 for 循环中忽略一个表达式(甚或所有表达式)。但此时，要考虑使用 while 循环。

## 2. while 循环

while 循环与 C++ 和 Java 中的 while 循环相同，与 Visual Basic 中的 While...Wend 循环相同。与 for 循环一样，while 也是一个预测试的循环。其语法是类似的，但 while 循环只有一个表达式：

```
while(condition)
    statement(s);
```

与 for 循环不同的是，while 循环最常用于下述情况：在循环开始前，不知道重复执行一个语句或语句块的次数。通常，在某次迭代中，while 循环体中的语句把布尔标记设置为 false，结束循环，如下面的例子所示。

```
bool condition = false;
while (!condition)
{
    // This loop spins until the condition is true.
    DoSomeWork();
    condition = CheckCondition(); // assume CheckCondition() returns a bool
}
```

所有的 C# 循环机制，包括 while 循环，如果只重复执行一条语句，而不是一个语句块，都可以省略花括号。许多程序员都认为最好在任何情况下都加上花括号。

## 3. do...while 循环

do...while 循环是 while 循环的后测试版本。它与 C++ 和 Java 中的 do...while 循环相同，与 Visual Basic 中的 Loop...While 循环相同，该循环的测试条件要在执行完循环体之后执行。因此 do...while 循环适合于至少执行一次循环体的情况：

```
bool condition;
do
{
    // This loop will at least execute once, even if Condition is false.
    MustBeCalledAtLeastOnce();
    condition = CheckCondition();
} while (condition);
```

## 4. foreach 循环

foreach 循环是我们讨论的最后一种 C# 循环机制。其他循环机制都是 C 和 C++ 的最早期版本，而 foreach 语句是新增的循环机制(借用于 Visual Basic)，也是非常受欢迎的一种循环。

foreach 循环可以迭代集合中的每一项。现在不必考虑集合的概念，第 10 章将介绍集合。知道集合是一种包含其他对象的对象即可。从技术上看，要使用集合对象，就必须支持 IEnumerable 接口。集合的例子有 C# 数组、System.Collection 命名空间中的集合类，以及用户

定义的集合类。从下面的代码中可以了解 `foreach` 循环的语法，其中假定 `arrayOfInts` 是一个整型数组：

```
foreach (int temp in arrayOfInts)
{
    Console.WriteLine(temp);
}
```

其中，`foreach` 循环每次迭代数组中的一个元素。它把每个元素的值放在 `int` 型的变量 `temp` 中，然后执行一次循环迭代。

这里也可以使用类型推断功能。此时，`foreach` 循环变成：

```
foreach (var temp in arrayOfInts)
```

`temp` 的类型推断为 `int`，因为这是集合项的类型。

注意，`foreach` 循环不能改变集合中各项(上面的 `temp`)的值，所以下面的代码不会编译：

```
foreach (int temp in arrayOfInts)
{
    temp++;
    Console.WriteLine(temp);
}
```

如果需要迭代集合中的各项，并改变它们的值，就应使用 `for` 循环。

### 2.5.3 跳转语句

C#提供了许多可以立即跳转到程序中另一行代码的语句，在此，先介绍 `goto` 语句。

#### 1. goto 语句

`goto` 语句可以直接跳转到程序中用标签指定的另一行(标签是一个标识符，后跟一个冒号)：

```
goto Label1;
    Console.WriteLine("This won't be executed");
Label1:
    Console.WriteLine("Continuing execution from here");
```

`goto` 语句有两个限制。不能跳转到像 `for` 循环这样的代码块中，也不能跳出类的范围，不能退出 `try...catch` 块后面的 `finally` 块(第14章将介绍如何用 `try...catch...finally` 块处理异常)。

`goto` 语句的名声不太好，在大多数情况下不允许使用它。一般情况下，使用它肯定不是面向对象编程的好方式。但是有一个地方使用它是相当方便的——在 `switch` 语句的 `case` 子句之间跳转，这是因为 C# 的 `switch` 语句在故障处理方面非常严格。前面介绍了其语法。

#### 2. break 语句

前面简要提到过 `break` 语句——在 `switch` 语句中使用它退出某个 `case` 语句。实际上，`break` 也可以用于退出 `for`、`foreach`、`while` 或 `do...while` 循环，该语句会使控制流执行循环后面的语句。

如果该语句放在嵌套的循环中，就执行最内部循环后面的语句。如果 `break` 放在 `switch` 语

句或循环外部，就会产生编译错误。

### 3. continue 语句

continue 语句类似于 break，也必须在 for、foreach、while 或 do...while 循环中使用。但它只退出循环的当前迭代，开始执行循环的下一次迭代，而不是退出循环。

### 4. return 语句

return 语句用于退出类的方法，把控制权返回方法的调用者，如果方法有返回类型，return 语句必须返回这个类型的值，如果方法没有返回类型，应使用没有表达式的 return 语句。

## 2.6 枚举

枚举是用户定义的整数类型。在声明一个枚举时，要指定该枚举可以包含的一组可接受的实例值。不仅如此，还可以给值指定易于记忆的名称。如果在代码的某个地方，要试图把一个不在可接受范围内的值赋予枚举的一个实例，编译器就会报告一个错误。这个概念对于 Visual Basic 程序员来说是新的。C++ 支持枚举，但 C# 枚举要比 C++ 枚举强大得多。

从长远来看，创建枚举可以节省大量的时间，减少许多麻烦。使用枚举比使用无格式的整数至少有如下三个优势：

- 如上所述，枚举可以使代码更易于维护，有助于确保给变量指定合法的、期望的值。
- 枚举使代码更清晰，允许用描述性的名称表示整数值，而不是用含义模糊的数来表示。
- 枚举使代码更易于键入。在给枚举类型的实例赋值时，Visual Studio IDE 会通过 IntelliSense 弹出一个包含可接受值的列表框，减少了按键次数，并能够让我们回忆起可选的值。

定义如下的枚举：

```
public enum TimeOfDay
{
    Morning = 0,
    Afternoon = 1,
    Evening = 2
}
```

本例在枚举中使用一个整数值，来表示一天的每个阶段。现在可以把这些值作为枚举的成员来访问。例如，TimeOfDay.Morning 返回数字 0。使用这个枚举一般是把合适的值传送给方法，或在 switch 语句中迭代可能的值。

```
class EnumExample
{
    public static int Main()
    {
        WriteGreeting(TimeOfDay.Morning);
        return 0;
    }

    static void WriteGreeting(TimeOfDay timeOfDay)
    {
```



```

switch(timeOfDay)
{
    case TimeOfDay.Morning:
        Console.WriteLine("Good morning!");
        break;
    case TimeOfDay.Afternoon:
        Console.WriteLine("Good afternoon!");
        break;
    case TimeOfDay.Evening:
        Console.WriteLine("Good evening!");
        break;
    default:
        Console.WriteLine("Hello!");
        break;
}
}

```

在 C# 中, 枚举的真正强大之处是它们在后台会实例化为派生于基类 `System.Enum` 的结构。这表示可以对它们调用方法, 执行有用的任务。注意因为 .NET Framework 的执行方式, 在语法上把枚举当做结构是不会有性能损失的。实际上, 一旦代码编译好, 枚举就成为基本类型, 与 `int` 和 `float` 类似。

可以获取枚举的字符串表示, 例如使用前面的 `TimeOfDay` 枚举:

```

TimeOfDay time = TimeOfDay.Afternoon;
Console.WriteLine(time.ToString());

```

会返回字符串 `Afternoon`。

另外, 还可以从字符串中获取枚举值:

```

TimeOfDay time2 = (TimeOfDay) Enum.Parse(typeof(TimeOfDay), "afternoon", true);
Console.WriteLine((int)time2);

```

这段代码说明了如何从字符串获取枚举值, 并转换为整数。要从字符串中转换, 需要使用静态的 `Enum.Parse()` 方法, 这个方法带 3 个参数, 第一个参数是要使用的枚举类型, 其语法是关键字 `typeof` 后跟放在括号中的枚举类名。`typeof` 运算符将在第 6 章详细论述。第二个参数是要转换的字符串, 第三个参数是一个 `bool`, 指定在进行转换时是否忽略大小写。最后, 注意 `Enum.Parse()` 方法实际上返回一个对象引用——我们需要把这个字符串显式转换为需要的枚举类型(这是一个拆箱操作的例子)。对于上面的代码, 将返回 1, 作为一个对象, 对应于 `TimeOfDay.Afternoon` 的枚举值。在显式转换为 `int` 时, 会再次生成 1。

`System.Enum` 上的其他方法可以返回枚举定义中的值的个数、列出值的名称等。详细信息参见 MSDN 文档。

## 2.7 数组

本章不打算详细介绍数组, 因为第 5 章将详细论述数组。但本章将介绍编写一维数组的句法。在声明 C# 中的数组时, 要在各个元素的变量类型后面, 加上一组方括号(注意数组中的所有元素必须有相同的数据类型)。

注意:

Visual Basic 用户注意, C# 中的数组使用方括号, 而不是圆括号。C++ 用户很熟悉方括号, 但应仔细查看这里给出的代码, 因为声明数组变量的 C# 语法与 C++ 语法并不相同。

例如, `int` 表示一个整数, 而 `int[]` 表示一个整型数组:

```
int[] integers;
```

要初始化特定大小的数组, 可以使用 `new` 关键字, 在类型名后面的方括号中给出数组的大小:

```
// Create a new array of 32 ints
int[] integers = new int[32];
```

所有的数组都是引用类型, 并遵循引用的语义。因此, 即使各个元素都是基本的值类型, `integers` 数组也是引用类型。如果以后编写如下代码:

```
int[] copy = integers;
```

该代码也只是把变量 `copy` 指向同一个数组, 而不是创建一个新数组。

要访问数组中的单个元素, 可以使用通常的语法, 在数组名的后面, 把元素的下标放在方括号中。所有的 C# 数组都使用基于 0 的下标方式, 所以要用下标 0 引用第一个变量:

```
integers[0] = 35;
```

同样, 用下标值 31 引用有 32 个元素的数组中的最后一个元素:

```
integers[31] = 432;
```

C# 的数组语法也非常灵活, 实际上, C# 可以在声明数组时不进行初始化, 这样以后就可以在程序中动态地指定其大小。利用这项技术, 可以创建一个空引用, 以后再使用 `new` 关键字把这个引用指向请求动态分配的内存位置:

```
int[] integers;
integers = new int[32];
```

可以使用下面的语法查看数组包含多少个元素:

```
int numElements = integers.Length; // integers is any reference to an array.
```

## 2.8 命名空间

如前所述, 命名空间提供了一种组织相关类和其他类型的方式。与文件或组件不同, 命名空间是一种逻辑组合, 而不是物理组合。在 C# 文件中定义类时, 可以把它包括在命名空间定义中。以后, 在定义另一个类, 在另一个文件中执行相关操作时, 就可以在同一个命名空间中包含它, 创建一个逻辑组合, 告诉使用类的其他开发人员: 这两个类是如何相关的以及如何使用它们:

```
namespace CustomerPhoneBookApp
{
```

```
using System;

public struct Subscriber
{
    // Code for struct here...
}
}
```

把一个类型放在命名空间中，可以有效地给这个类型指定一个较长的名称，该名称包括类型的命名空间，后面是句点(.)和类的名称。在上面的例子中，Subscriber 结构的全名是 CustomerPhoneBookApp.Subscriber。这样，有相同短名的不同的类就可以在同一个程序中使用。全名常常称为完全限定的名称。

也可以在命名空间中嵌套其他命名空间，为类型创建层次结构：

```
namespace Wrox
{
    namespace ProCSharp
    {
        namespace Basics
        {
            class NamespaceExample
            {
                // Code for the class here...
            }
        }
    }
}
```

每个命名空间名都由它所在命名空间的名称组成，这些名称用句点分隔开，首先是最外层的命名空间，最后是它自己的短名。所以 ProCSharp 命名空间的全名是 Wrox.ProCSharp，NamespaceExample 类的全名是 Wrox.ProCSharp.Basics.NamespaceExample。

使用这个语法也可以组织自己的命名空间定义中的命名空间，所以上面的代码也可以写为：

```
namespace Wrox.ProCSharp.Basics
{
    class NamespaceExample
    {
        // Code for the class here...
    }
}
```

注意不允许在另一个嵌套的命名空间中声明多部分的命名空间。

命名空间与程序集无关。同一个程序集中可以有不同的命名空间，也可以在不同的程序集中定义同一个命名空间中的类型。

### 2.8.1 using 语句

显然，命名空间相当长，键入起来很繁琐，用这种方式指定某个类也是不必要的。如本章开头所述，C#允许简写类的全名。为此，要在文件的顶部列出类的命名空间，前面加上 using 关键字。在文件的其他地方，就可以使用其类型名称来引用命名空间中的类型了：

```
using System;
using Wrox.ProCSharp;
```

如前所述,所有的 C#源代码都以语句 `using System;`开头,这仅是因为 Microsoft 提供的许多有用的类都包含在 `System` 命名空间中。

如果 `using` 指令引用的两个命名空间包含同名的类型,就必须使用完整的名称(或者至少较长的名称),确保编译器知道访问哪个类型,例如,类 `NamespaceExample` 同时存在于 `Wrox.ProCSharp.Basics` 和 `Wrox.ProCSharp.OOP` 命名空间中,如果要在命名空间 `Wrox.ProCSharp` 中创建一个类 `Test`,并在该类中实例化 `NamespaceExample` 类的一个对象,就需要指定使用哪个类:

```
using Wrox.ProCSharp;

class Test
{
    public static int Main()
    {
        Basics.NamespaceExample nSEx = new Basics.NamespaceExample();
        //do something with the nSEx variable
        return 0;
    }
}
```

#### 注意:

因为 `using` 语句在 C#文件的开头, C 和 C++也把 `#include` 语句放在这里,所以从 C++迁移到 C#的程序员常把命名空间与 C++风格的头文件相混淆。不要犯这种错误, `using` 语句在这些文件之间并没有建立物理链接。C#也没有对应于 C++头文件的部分。

公司应花一定的时间开发一种命名空间模式,这样其开发人员才能快速定位他们需要的功能,而且公司内部使用的类名也不会与外部的类库相冲突。本章后面将介绍建立命名空间模式的规则和其他命名约定。

## 2.8.2 命名空间的别名

`using` 关键字的另一个用途是给类和命名空间指定别名。如果命名空间的名称非常长,又要在代码中使用多次,但不希望该命名空间的名称包含在 `using` 指令中(例如,避免类名冲突),就可以给该命名空间指定一个别名,其语法如下:

```
using alias = NamespaceName;
```

下面的例子(前面例子的修订版本)给 `Wrox.ProCSharp.Basics` 命名空间指定 `Introduction` 别名,并使用这个别名实例化了一个 `NamespaceExample` 对象,这个对象是在该命名空间中定义的。注意命名空间别名的修饰符是 `::`。因此将先从 `Introduction` 命名空间别名开始搜索。如果在相同的作用域中引入了一个 `Introduction` 类,就会发生冲突。即使出现了冲突, `::`操作符也允许引用别名。 `NamespaceExample` 类有一个方法 `GetNamespace()`,该方法调用每个类都有的 `GetType()`方法,以访问表示类的类型的 `Type` 对象。下面使用这个对象来返回类的命名空间名:

```
using System;
using Introduction = Wrox.ProCSharp.Basics;
class Test
{
    public static int Main()
```



```

{
    Introduction::NamespaceExample NSEx =
        new Introduction::NamespaceExample();
    Console.WriteLine(NSEx.GetNamespace());
    return 0;
}
}

```

```

namespace Wrox.ProCSharp.Basics
{
    class NamespaceExample
    {
        public string GetNamespace()
        {
            return this.GetType().Namespace;
        }
    }
}

```

## 2.9 Main()方法

本章的开头提到过，C#程序是从方法 `Main()` 开始执行的。这个方法必须是类或结构的静态方法，并且其返回类型必须是 `int` 或 `void`。

虽然显式指定 `public` 修饰符是很常见的，因为按照定义，必须在程序外部调用该方法，但我们给该方法指定什么访问级别并不重要，即使把该方法标记为 `private`，它也可以运行。

### 2.9.1 多个 Main()方法

在编译C#控制台或 Windows 应用程序时，默认情况下，编译器会在类中查找与上述签名匹配的 `Main` 方法，并使这个类方法成为程序的入口。如果有多个 `Main` 方法，编译器就会返回一个错误消息，例如，考虑下面的代码 `MainExample.cs`：

```

using System;

namespace Wrox.ProCSharp.Basics
{
    class Client
    {
        public static int Main()
        {
            MathExample.Main();
            return 0;
        }
    }

    class MathExample
    {
        static int Add(int x, int y)
        {
            return x + y;
        }

        public static int Main()
        {
            int i = Add(5, 10);
            Console.WriteLine(i);
            return 0;
        }
    }
}

```



```

    }
}
}

```

上述代码中包含两个类，它们都有一个 `Main()` 方法。如果按照通常的方式编译这段代码，就会得到下述错误：

**csc MainExample.cs**

Microsoft (R) Visual C# .NET Compiler version 9.00.20404

for Microsoft (R) .NET Framework version 3.5

Copyright (C) Microsoft Corporation. All rights reserved.

MainExample.cs(7,23): error CS0017: Program 'MainExample.exe' has more than one entry point defined:

'Wrox.ProCSharp.Basics.Client.Main()'

MainExample.cs(21,23): error CS0017: Program 'MainExample.exe' has more than one entry point defined:

'Wrox.ProCSharp.Basics.MathExample.Main()'

但是，可以使用 `/main` 选项，其后跟 `Main()` 方法所属类的全名(包括命名空间)，明确告诉编译器把哪个方法作为程序的入口点：

**csc MainExample.cs /main:Wrox.ProCSharp.Basics.MathExample**

## 2.9.2 给 `Main()` 方法传送参数

前面的例子只介绍了不带参数的 `Main()` 方法。但在调用程序时，可以让 CLR 包含一个参数，将命令行参数转送给程序。这个参数是一个字符串数组，传统称为 `args`(但 C# 可以接受任何名称)。在启动程序时，可以使用这个数组，访问通过命令行传送过来的选项。

下面的例子 `ArgsExample.cs` 是在传送给 `Main` 方法的字符串数组中迭代，并把每个选项的值写入控制台窗口：

```

using System;

namespace Wrox.ProCSharp.Basics
{
    class ArgsExample
    {
        public static int Main(string[] args)
        {
            for (int i = 0; i < args.Length; i++)
            {
                Console.WriteLine(args[i]);
            }
            return 0;
        }
    }
}

```

通常使用命令行就可以编译这段代码。在运行编译好的可执行文件时，可以在程序名的后面加上参数，例如：

**ArgsExample /a /b /c**

/a  
/b  
/c

### 2.10 有关编译 C#文件的更多内容

前面介绍了如何使用 `csc.exe` 编译控制台应用程序,但其他类型的应用程序如何编译?如果要引用一个类库,该怎么办?MSDN 文档介绍了 C#编译器的所有编译选项,这里只介绍其中最重要的选项。

要回答第一个问题,应使用 `/target` 选项(常简写为 `/t`)来指定要创建的文件类型。文件类型可以是表 2-8 所示的类型中的一种。

表 2-8

选 项	输 出
/t:exe	控制台应用程序 (默认)
/t:library	带有清单的类库
/t:module	没有清单的组件
/t:winexe	Windows 应用程序 (没有控制台窗口)

如果想得到一个可由 .NET 运行库加载的非可执行文件(例如 DLL),就必须把它编译为一个库。如果把 C#文件编译为一个模块,就不会创建任何程序集。虽然模块不能由运行库加载,但可以使用 `/addmodule` 选项编译到另一个清单中。

另一个需要注意的选项是 `/out`,该选项可以指定由编译器生成的输出文件名。如果没有指定 `/out` 选项,编译器就会使用输入的 C#文件名,加上目标类型的扩展名来建立输出文件名(例如 `.exe` 表示 Windows 或控制台应用程序, `.dll` 表示类库)。注意 `/out` 和 `/t`(或 `/target`)选项必须放在要编译的文件名前面。

默认状态下,如果在未引用的程序集中引用类型,可以使用 `/reference` 或 `/r` 选项,后跟程序集的路径和文件名。下面的例子说明了如何编译类库,并在另一个程序集中引用这个库。它包含两个文件:

- 类库
- 控制台应用程序,该应用程序调用库中的一个类。

第一个文件 `MathLibrary.cs` 包含 DLL 的代码,为了简单起见,它只包含一个公共类 `MathLib` 和一个方法,该方法把两个 `int` 类型的数据加在一起:

```
namespace Wrox.ProCSharp.Basics
{
    public class MathLib
    {
        public int Add(int x, int y)
        {
            return x + y;
        }
    }
}
```

使用下述命令把这个 C# 文件编译为 .NET DLL:

```
csc /t:library MathLibrary.cs
```

控制台应用程序 MathClient.cs 将简单地实例化这个对象, 调用其 Add 方法, 在控制台窗口中显示结果:

```
using System;

namespace Wrox.ProCSharp.Basics
{
    class Client
    {
        public static void Main()
        {
            MathLib mathObj = new MathLib();
            Console.WriteLine(mathObj.Add(7, 8));
        }
    }
}
```

使用 /r 选项编译这个文件, 使之指向新编译的 DLL:

```
csc MathClient.cs /r:MathLibrary.dll
```

当然, 下面就可以像往常一样运行它了: 在命令提示符上输入 MathClient, 其结果是显示数字 15——加运算的结果。

## 2.11 控制台 I/O

现在, 读者应基本熟悉了 C# 的数据类型以及控制线程如何执行操作这些数据类型的程序。本章还要使用 Console 类的几个静态方法来读写数据, 这些方法在编写基本的 C# 程序时非常有效, 下面就详细介绍它们。

要从控制台窗口中读取一行文本, 可以使用 Console.ReadLine() 方法, 它会从控制台窗口中读取一个输入流(在用户按下回车键时停止), 并返回输入的字符串。写入控制台也有两个对应的方法, 前面已经使用过它们:

- Console.Write() 方法将指定的值写入控制台窗口。
- Console.WriteLine() 方法类似, 但在输出结果的最后添加一个换行符。

所有预定义类型(包括 object) 都有这些函数的各种形式(重载), 所以在大多数情况下, 在显示值之前不必把它们转换为字符串。

例如, 下面的代码允许用户输入一行文本, 并显示该文本:

```
string s = Console.ReadLine();

Console.WriteLine(s);
```

Console.WriteLine() 还允许用与 C 的 printf() 函数类似的方式显示格式化的结果。要以这种方式使用 WriteLine(), 应传入许多参数。第一个参数是花括号中包含标记的字符串, 在这个花括号中, 要把后续的参数插入到文本中。每个标记都包含一个基于 0 的索引, 表示列表中参数的序号。例如, {0} 表示列表中的第一个参数, 所以下面的代码:

```
int i = 10;
int j = 20;
Console.WriteLine("{0} plus {1} equals {2}", i, j, i + j);
```

会显示:

10 plus 20 equals 30

也可以为值指定宽度, 调整文本在该宽度中的位置, 正值表示右对齐, 负值表示左对齐。为此可以使用格式{n,w}, 其中 n 是参数索引, w 是宽度值。

```
int i = 940;
int j = 73;
Console.WriteLine(" {0,4}\n+{1,4}\n---\n {2,4}", i, j, i + j);
```

结果如下:

```
    940
+   73
-----
   1013
```

最后, 还可以添加一个格式字符串, 和一个可选的精度值。这里没有列出格式字符串的完整列表, 因为如第 8 章所述, 我们可以定义自己的格式字符串。但用于预定义类型的主要格式字符串如表 2-9 所示。

表 2-9

字 符 串	说 明
C	本地货币格式
D	十进制格式, 把整数转换为以 10 为基数的数, 如果给定一个精度说明符, 就加上前导 0
E	科学计数法(指数)格式。精度说明符设置小数位数(默认为 6)。格式字符串的大小写("e"或"E")确定指数符号的大小写
F	固定点格式, 精度说明符设置小数位数, 可以为 0
G	普通格式, 使用 E 或 F 格式取决于哪种格式较简单
N	数字格式, 用逗号表示千分符, 例如 32,767.44
P	百分数格式
X	十六进制格式, 精度说明符用于加上前导 0

注意格式字符串都不需要考虑大小写, 除 e/E 之外。

如果要使用格式字符串, 应把它放在给出参数个数和字段宽度的标记后面, 并用一个冒号把它们分隔开。例如, 要把 decimal 值格式化为货币格式, 且使用计算机上的地区设置, 其精度为两位小数, 则使用 C2:

```
decimal i = 940.23m;
decimal j = 73.7m;
Console.WriteLine(" {0,9:C2}\n+{1,9:C2}\n - - - - -\n {2,9:C2}", i, j, i + j);
```

在美国, 其结果是:

```

    $940.23
+   $73.70
-----
   $1,013.93

```

最后一个技巧是，可以使用占位符来代替这些格式字符串，例如：

```
double d = 0.234;
Console.WriteLine("{0:#.00}", d);
```

其结果为.23，因为如果在符号(#)的位置上没有字符，就会忽略该符号(#)，如果 0 的位置上有一个字符，就用这个字符代替 0，否则就显示 0。

## 2.12 使用注释

本节的内容表面上看起来很简单——给代码添加注释。

### 2.12.1 源文件中的内部注释

在本章开头提到过，C#使用传统的 C 风格注释方式：单行注释使用// ...，多行注释使用/\* ... \*/：

```
// This is a single-line comment
/* This comment
   spans multiple lines */
```

单行注释中的任何内容，即//后面的内容都会被编译器忽略。多行注释中/\* 和 \*/之间的所有内容也会被忽略。显然不能在多行注释中包含\*/组合，因为这会被当作注释的结尾。

实际上，可以把多行注释放在一行代码中：

```
Console.WriteLine(/*Here's a comment! */ "This will compile");
```

像这样的内联注释在使用时应小心，因为它们会使代码难以理解。但这样的注释在调试时是非常有用的，例如，在运行代码时要临时使用另一个值：

```
DoSomething(Width, /*Height*/ 100);
```

当然，字符串面值中的注释字符会按照一般的字符来处理：

```
string s = "/* This is just a normal string */";
```

### 2.12.2 XML 文档说明

如前所述，除了 C 风格的注释外，C#还有一个非常好的功能，本章将讨论这一功能。根据特定的注释自动创建 XML 格式的文档说明。这些注释都是单行注释，但都以 3 个斜杠(///)开头，而不是通常的两个斜杠。在这些注释中，可以把包含类型和类型成员的文档说明的 XML 标识符放在代码中。

编译器可以识别表 2-10 所示的标识符。



表 2-10

标识符	说明
<c>	把行中的文本标记为代码，例如<c>int i = 10;</c>
<code>	把多行标记为代码
<example>	标记为一个代码示例
<exception>	说明一个异常类(编译器要验证其语法)
<include>	包含其他文档说明文件的注释(编译器要验证其语法)
<list>	把列表插入到文档说明中
<param>	标记方法的参数(编译器要验证其语法)
<paramref>	表示一个单词是方法的参数(编译器要验证其语法)
<permission>	说明对成员的访问(编译器要验证其语法)
<remarks>	给成员添加描述
<returns>	说明方法的返回值
<see>	提供对另一个参数的交叉引用(编译器要验证其语法)
<seealso>	提供描述中的“参见”部分(编译器要验证其语法)
<summary>	提供类型或成员的简短小结
<value>	描述属性

要了解它们的工作方式，可以在上一节的 MathLibrary.cs 文件中添加一些 XML 注释，并称之为 Math.cs。我们给类及其 Add()方法添加一个<summary>元素，也给 Add()方法添加一个<returns>元素和两个<param>元素：

```
// Math.cs
namespace Wrox.ProCSharp.Basics
{
    ///<summary>
    ///  Wrox.ProCSharp.Basics.Math class.
    ///  Provides a method to add two integers.
    ///</summary>
    public class Math
    {
        ///<summary>
        ///  The Add method allows us to add two integers
        ///</summary>
        ///<returns>Result of the addition (int)</returns>
        ///<param name="x">First number to add</param>
        ///<param name="y">Second number to add</param>
        public int Add(int x, int y)
        {
            return x + y;
        }
    }
}
```

C#编译器可以把 XML 元素从特定的注释中提取出来，并使用它们生成一个 XML 文件。要让编译器为程序集生成 XML 文档说明，需在编译时指定/doc 选项，后跟要创建的文件名：

```
csc /t:library /doc:Math.xml Math.cs
```

如果 XML 注释没有生成格式正确的 XML 文档，编译器就生成一个错误。  
上面的代码会生成一个 XML 文件 Math.xml，如下所示。

```
<?xml version="1.0"?>
<doc>
  <assembly>
    <name>Math</name>
  </assembly>
  <members>
    <member name="T:Wrox.ProCSharp.Basics.Math">
      <summary>
        Wrox.ProCSharp.Basics.Math class.
        Provides a method to add two integers.
      </summary>
    </member>
    <member name=
      "M:Wrox.ProCSharp.Basics.Math.Add(System.Int32,System.Int32)">
      <summary>
        The Add method allows us to add two integers.
      </summary>
      <returns>Result of the addition (int)</returns>
      <param name="x">First number to add</param>
      <param name="y">Second number to add</param>
    </member>
  </members>
</doc>
```

注意，编译器为我们做了一些工作——它创建了一个<assembly>元素，并为该文件中的每个类型或类型成员添加一个<member>元素。每个<member>元素都有一个 name 特性，其中包含成员的全名，前面有一个字母表示其类型："T:"表示这是一个类型，"F:"表示这是一个字段，"M:"表示这是一个成员。

## 2.13 C#预处理器指令

除了前面介绍的常用关键字外，C#还有许多名为“预处理器指令”的命令。这些命令从来不会转化为可执行代码中的命令，但会影响编译过程的各个方面。例如，使用预处理器指令可以禁止编译器编译代码的某一部分。如果计划发布两个版本的代码，即基本版本和有更多功能的企业版本，就可以使用这些预处理器指令。在编译软件的基本版本时，使用预处理器指令还可以禁止编译器编译与额外功能相关的代码。另外，在编写提供调试信息的代码时，也可以使用预处理器指令。实际上，在销售软件时，一般不希望编译这部分代码。

预处理器指令的开头都有符号#。

**注意：**

C++开发人员应知道在 C 和 C++ 中，预处理器指令是非常重要的，但是，在 C# 中，并没有那么多的预处理器指令，它们的使用也不太频繁。C# 提供了其他机制来实现许多 C++ 指令的功能，例如定制特性。还要注意，C# 并没有一个像 C++ 那样的独立预处理器，所谓的预处理器指令实际上是由编译器处理的。尽管如此，C# 仍保留了一些预处理器指令，因为这些命令对预

处理器有一定的影响。

下面简要介绍预处理器指令的功能。

### 2.13.1 #define 和 #undef

#define 的用法如下所示：

```
#define DEBUG
```

它告诉编译器存在给定名称的符号，在本例中是 `DEBUG`。这有点类似于声明一个变量，但这个变量并没有真正的值，只是存在而已。这个符号不是实际代码的一部分，而只在编译器编译代码时存在。在 C# 代码中它没有任何意义。

#undef 正好相反——删除符号的定义：

```
#undef DEBUG
```

如果符号不存在，`#undef` 就没有任何作用。同样，如果符号已经存在，`#define` 也不起作用。必须把 `#define` 和 `#undef` 命令放在 C# 源代码的开头，在声明要编译的任何对象的代码之前。

`#define` 本身并没有什么用，但与其他预处理器指令(特别是 `#if`)结合使用时，它的功能就非常强大了。

**注意：**

这里应注意一般 C# 语法的一些变化。预处理器指令不用分号结束，一般一行上只有一个命令。这是因为对于预处理器指令，C# 不再要求命令用分号结束。如果它遇到一个预处理器指令，就会假定下一个命令在下一行上。

### 2.13.2 #if, #elif, #else 和 #endif

这些指令告诉编译器是否要编译某个代码块。考虑下面的方法：

```
int DoSomeWork(double x)
{
    // do something
    #if DEBUG
        Console.WriteLine("x is " + x);
    #endif
}
```

这段代码会像往常那样编译，但 `Console.WriteLine` 命令包含在 `#if` 子句内。这行代码只有在前面的 `#define` 命令定义了符号 `DEBUG` 后才执行。当编译器遇到 `#if` 语句后，将先检查相关的符号是否存在，如果符号存在，就编译 `#if` 块中的代码。否则，编译器会忽略所有的代码，直到遇到匹配的 `#endif` 指令为止。一般是在调试时定义符号 `DEBUG`，把与调试相关的代码放在 `#if` 子句中。在完成了调试后，就把 `#define` 语句注释掉，所有的调试代码会奇迹般地消失，可执行文件也会变小，最终用户不会被这些调试信息弄糊涂(显然，要做更多的测试，确保代码在没有定义 `DEBUG` 的情况下也能工作)。这项技术在 C 和 C++ 编程中非常普通，称为条件编译 (conditional compilation)。

`#elif` (=else if)和`#else` 指令可以用在`#if` 块中，其含义非常直观。也可以嵌套`#if` 块：

```
#define ENTERPRISE
#define W2K

// further on in the file

#if ENTERPRISE
    // do something
    #if W2K
        // some code that is only relevant to enterprise
        // edition running on W2K
    #endif
#elif PROFESSIONAL
    // do something else
#else
    // code for the leaner version
#endif
```

注意：

与 C++ 中的情况不同，使用`#if` 不是条件编译代码的唯一方式，C# 还通过 `Conditional` 特性提供了另一种机制，详见第 13 章。

`#if` 和 `#elif` 还支持一组逻辑运算符`!`、`==`、`!=`和`||`。如果符号存在，就被认为是 `true`，否则为 `false`，例如：

```
#if W2K && (ENTERPRISE==false) // if W2K is defined but ENTERPRISE isn't
```

### 2.13.3 #warning 和 #error

另外两个非常有用的预处理器指令是`#warning` 和`#error`，当编译器遇到它们时，会分别产生警告或错误。如果编译器遇到`#warning` 指令，会给用户显示`#warning` 指令后面的文本，之后编译继续进行。如果编译器遇到`#error` 指令，就会给用户显示后面的文本，作为一个编译错误信息，然后会立即退出编译，不会生成 IL 代码。

使用这两个指令可以检查`#define` 语句是不是做错了什么事，使用`#warning` 语句可以让自己想起做过什么事：

```
#if DEBUG && RELEASE
    #error "You've defined DEBUG and RELEASE simultaneously! "
#endif

#warning "Don't forget to remove this line before the boss tests the code! "
Console.WriteLine("*I hate this job*");
```

### 2.13.4 #region 和 #endregion

`#region` 和 `#endregion` 指令用于把一段代码标记为有给定名称的一个块，如下所示。

```
#region Member Field Declarations
    int x;
    double d;
    Currency balance;
#endregion
```

这看起来似乎没有什么用，它不影响编译过程。这些指令的优点是它们可以被某些编辑器识别，包括 Visual Studio 编辑器。这些编辑器可以使用这些指令使代码在屏幕上更好地布局。第 15 章会详细介绍它们。

### 2.13.5 #line

#line 指令可以用于改变编译器在警告和错误信息中显示的文件名和行号信息。这个指令用得并不多。如果编写代码时，在把代码发送给编译器前，要使用某些软件包改变键入的代码，就可以使用这个指令，因为这意味着编译器报告的行号或文件名与文件中的行号或编辑的文件名不匹配。#line 指令可以用于恢复这种匹配。也可以使用语法 #line default 把行号恢复为默认的行号：

```
#line 164 "Core.cs" // we happen to know this is line 164 in the file
                    // Core.cs, before the intermediate
                    // package mangles it.
```

```
// later on
```

```
#line default // restores default line numbering
```

### 2.13.6 #pragma

#pragma 指令可以抑制或恢复指定的编译警告。与命令行选项不同，#pragma 指令可以在类或方法上执行，对抑制警告的内容和抑制的时间进行更精细的控制。下面的例子禁止字段使用警告，然后在编译 MyClass 类后恢复该警告。

```
#pragma warning disable 169
public class MyClass
{
    int neverUsedField;
}
#pragma warning restore 169
```

## 2.14 C#编程规则

本节介绍编写 C# 程序时应注意的规则。

### 2.14.1 用于标识符的规则

本节将讨论变量、类、方法等的命名规则。注意本节所介绍的规则不仅是规则，也是 C# 编译器强制使用的。

标识符是给变量、用户定义的类型(例如类和结构)和这些类型的成员指定的名称。标识符区分大小写，所以 `interestRate` 和 `InterestRate` 是不同的变量。确定在 C# 中可以使用什么标识符有两个规则：

- 它们必须以一个字母或下划线开头，但可以包含数字字符；



- 不能把 C#关键字用作标识符。
- C#包含如表 2-11 所示的保留关键字。

表 2-11

abstract	do	in	Protected	true
as	double	int	Public	try
base	else	interface	Readonly	typeof
bool	enum	internal	Ref	uint
break	event	is	Return	ulong
byte	explicit	lock	Sbyte	unchecked
case	extern	long	Sealed	unsafe
catch	false	namespace	Short	ushort
char	finally	New	Sizeof	using
checked	fixed	Null	Stackalloc	virtual
class	float	Object	Static	volatile
const	for	Operator	String	void
continue	foreach	Out	struct	while
decimal	goto	Override	switch	
default	if	Params	this	
delegate	implicit	Private	throw	

如果需要把某一保留字用作标识符(例如，访问一个用另一种语言编写的类)，可以在标识符的前面加上前缀@符号,指示编译器其后的内容是一个标识符,而不是 C#关键字(所以 abstract 不是有效的标识符，而@abstract 是)。

最后，标识符也可以包含 Unicode 字符，用语法\uXXXX 来指定，其中 XXXX 是 Unicode 字符的四位十六进制编码。下面是有效标识符的一些例子：

- Name
- überfluß
- \_Identifier
- \u005fIdentifier

最后两个标识符是相同的，可以互换(005f 是下划线字符的 Unicode 代码)，所以这些标识符在同一个作用域内不要声明两次。注意虽然从语法上看，标识符中可以使用下划线字符，但在大多数情况下，最好不要这么做，因为它不符合 Microsoft 的变量命名规则，这种命名规则可以确保开发人员使用相同的命名规则，易于阅读每个人编写的代码。

2.14.2 用法约定

在任何开发环境中，通常有一些传统的编程风格。这些风格不是语言的一部分，而是约定，例如，变量如何命名，类、方法或函数如何使用等。如果使用某语言的大多数开发人员都遵循相同的约定，不同的开发人员就很容易理解彼此的代码，这一般有助于程序的维护。例如，Visual Basic 6

的一个公共(但不统一)约定是,表示字符串的变量名以小写字母s或str开头,如Dim sResult As String或Dim strMessage As String。约定主要取决于语言和环境。例如,在Windows平台上编程的C++开发人员一般使用前缀psz或lpsz表示字符串:char \*pszResult; char \*lpszMessage;,但在UNIX机器上,则不使用任何前缀:char \*Result; char \*Message;。

从本书中的示例代码中可以总结出,C#中的约定是命名变量时不使用任何前缀:string Result; string Message;。

#### 注意:

变量名用带有前缀字母来表示某个数据类型,这种约定称为Hungarian表示法。这样,其他阅读该代码的开发人员就可以立即从变量名中了解它代表什么数据类型。在有了智能编辑器和IntelliSense之后,人们普遍认为Hungarian表示法是多余的。

但是,在许多语言中,用法约定是从语言的使用过程中逐渐演变而来的,Microsoft编写的C#和整个.NET Framework都有非常多的用法约定,详见.NET/C# MSDN文档说明。这说明,从一开始,.NET程序就有非常高的互操作性,开发人员可以以此来理解代码。用法规则还得益于20年来面向对象编程的发展,因此相关的新闻组已经仔细考虑了这些用法规则,而且已经为开发团体所接受。所以我们应遵守这些约定。

但要注意,这些规则与语言规范是不同的。用户应尽可能遵循这些规则。但如果有很好的理由不遵循它们,也不会有什么問題。例如,不遵循这些用法约定,也不会出现编译错误。一般情况下,如果不遵循用法规则,就必须有一个说得过去的理由。规则应是一个正确的决策,而不是让人头痛的东西。在阅读本书的后续内容时,应注意到在本书的许多示例中,都没有遵循该约定,这通常是因为某些规则适用于大型程序,而不适合于本书中的小示例。如果编写一个完整的软件包,就应遵循这些规则,但它们并不适合于只有20行代码的独立程序。在许多情况下,遵循约定会使这些示例难以理解。

编程风格的规则非常多。这里只介绍一些比较重要的规则,以及最适合于用户的规则。如果用户要让代码完全遵循用法规则,就需要参考MSDN文档说明。

### 1. 命名约定

使程序易于理解的一个重要方面是给对象选择命名的方式,包括变量名、方法名、类名、枚举名和命名空间的名称。

显然,这些名称应反映对象的功能,且不与其他名称冲突。在.NET Framework中,一般规则也是变量名要反映变量实例的功能,而不是反映数据类型。例如,Height就是一个比较好的变量名,而IntegerValue就不太好。但是,这种规则是一种理想状态,很难达到。在处理控件时,大多数情况下使用ConfirmationDialog和ChooseEmployeeListBox等变量名比较好,这些变量名说明了变量的数据类型。

名称的约定包括以下几个方面:

#### (1) 名称的大小写

在许多情况下,名称都应使用Pascal大小写命名形式。Pascal大小写形式是指名称中单词的第一个字母大写,如EmployeeSalary、ConfirmationDialog、PlainTextEncoding。注意,命名空间、类、以及基类中的成员等的名称都应遵循该规则,最好不要使用带下划线字符的单词,即名称不应是employee\_salary。其他语言中常量的名称常常全部是大写,但在C#中最好不要这

样，因为这种名称很难阅读，而应全部使用 Pascal 大小写形式的命名约定：

```
const int MaximumLength;
```

我们还推荐使用另一种大小写模式：camel 大小写形式。这种形式类似于 Pascal 大小写形式，但名称中第一个单词的第一个字母不是大写：employeeSalary、confirmationDialog、plainTextEncoding。有三种情况可以使用 camel 大小写形式。

- 类型中所有私有成员字段的名称都应是 camel 大小写形式：

```
public int subscriberId;
```

但要注意成员字段名常常用一个下划线开头：

```
public int _subscriberId;
```

- 传递给方法的所有参数都应是 camel 大小写形式：

```
public void RecordSale(string salesmanName, int quantity);
```

- camel 大小写形式也可以用于区分同名的两个对象——比较常见的情况是属性封装一个字段：

```
private string employeeName;

public string EmployeeName
{
    get
    {
        return employeeName;
    }
}
```

如果这么做，则私有成员总是使用 camel 大小写形式，而公共的或受保护的成员总是使用 Pascal 大小写形式，这样使用这段代码的其他类就只能使用 Pascal 大小写形式的名称了(除了参数名以外)。

还要注意大小写问题。C#是区分大小写的，所以在 C#中，仅大小写不同的名称在语法上是正确的，如上面的例子。但是，程序集可能在 Visual Basic .NET 应用程序中调用，而 Visual Basic .NET 是不区分大小写的，如果使用仅大小写不同的名称，就必须使这两个名称不能在程序集的外部访问(上例是可行的，因为仅私有变量使用了 camel 大小写形式的名称)。否则，Visual Basic .NET 中的其他代码就不能正确使用这个程序集。

## (2) 名称的风格

名称的风格应保持一致。例如，如果类中的一个方法叫 ShowConfirmationDialog()，另一个方法就不能叫 ShowDialogWarning()或 WarningDialogShow()，而应是 ShowWarningDialog()。

## (3) 命名空间的名称

命名空间的名称非常重要，一定要仔细设计，以避免一个命名空间中对象的名称与其他对象同名。记住，命名空间的名称是 .NET 区分共享程序集中对象名的唯一方式。如果软件包的命名空间使用的名称与另一个软件包相同，而这两个软件包都安装在一台计算机上，就会出问题。因此，最好用自己的公司名创建顶级的命名空间，再嵌套技术范围较窄、用户所在小组或部门、或类所在软件包的命名空间。Microsoft 建议使用如下的命名空间：<CompanyName>.<TechnologyName>，例如：

```
WeaponsOfDestructionCorp.RayGunControllers
```

WeaponsOfDestructionCorp.Viruses

(4) 名称和关键字

名称不应与任何关键字冲突，这是非常重要的。实际上，如果在代码中，试图给某个对象指定与 C#关键字同名的名称，就会出现语法错误，因为编译器会假定该名称表示一个语句。但是，由于类可能由其他语言编写的代码访问，所以不能使用其他.NET 语言中的关键字作为对象的名称。一般说来，C++关键字类似于 C#关键字，不太可能与 C++混淆，Visual C++常用的关键字则用两个下划线字符开头。与 C#一样，C++关键字都是小写字母，如果要遵循公共类和成员使用 Pascal 风格的名称的约定，则在它们的名称中至少有一个字母是大写，因此不会与 C++关键字冲突。另一方面，Visual Basic 的问题会多一些，因为 Visual Basic 的关键字要比 C#的多，而且它不区分大小写，不能依赖于 Pascal 风格的名称来区分类和成员。

表 2-12 列出了 Visual Basic 中的关键字和标准函数调用，无论对 C#公共类使用什么大小写组合，这些名称都不应使用。

表 2-12

Abs	Do	Loc	RGB
Add	Double	Local	Right
AddHandler	Each	Lock	RmDir
AddressOf	Else	LOF	Rnd
Alias	Elseif	Log	RTrim
And	Empty	Long	SaveSettings
Ansi	End	Loop	Second
AppActivate	Enum	LTrim	Seek
Append	EOF	Me	Select
As	Erase	Mid	SetAttr
Asc	Err	Minute	SetException
Assembly	Error	MIRR	Shared
Atan	Event	MkDir	Shell
Auto	Exit	Module	Short
Beep	Exp	Month	Sign
Binary	Explicit	MustInherit	Sin
BitAnd	ExternalSource	MustOverride	Single
BitNot	False	MyBase	SLN
BitOr	FileAttr	MyClass	Space
BitXor	FileCopy	Namespace	Spc
Boolean	FileDateTime	New	Split
ByRef	FileLen	Next	Sqrt
Byte	Filter	Not	Static

(续表)

ByVal	Finally	Nothing	Step
Call	Fix	NotInheritable	Stop
Case	For	NotOverridable	Str
Catch	Format	Now	StrComp
CBool	FreeFile	NPer	StrConv
CByte	Friend	NPV	Strict
CDate	Function	Null	String
CDbl	FV	Object	Structure
CDec	Get	Oct	Sub
ChDir	GetAllSettings	Off	Switch
ChDrive	GetAttr	On	SYD
Choose	GetException	Open	SyncLock
Chr	GetObject	Option	Tab
CInt	GetSetting	Optional	Tan
Class	GetType	Or	Text
Clear	GoTo	Overloads	Then
CLng	Handles	Overridable	Throw
Close	Hex	Overrides	TimeOfDay
Collection	Hour	ParamArray	Timer
Command	If	Pmt	TimeSerial
Compare	Iif	PPmt	TimeValue
Const	Implements	Preserve	To
Cos	Imports	Print	Today
CreateObject	In	Private	Trim
CShort	Inherits	Property	Try
CSng	Input	Public	TypeName
CStr	InStr	Put	TypeOf
CurDir	Int	PV	UBound
Date	Integer	QBColor	UCase
DateAdd	Interface	Raise	Unicode
DateDiff	Ipmt	RaiseEvent	Unlock
DatePart	IRR	Randomize	Until
DateSerial	Is	Rate	Val
DateValue	IsArray	Read	Weekday
Day	IsDate	ReadOnly	While



(续表)

DDB	IsDBNull	ReDim	Width
Decimal	IsNumeric	Remove	With
Declare	Item	RemoveHandler	WithEvents
Default	Kill	Rename	Write
Delegate	Lcase	Replace	WriteOnly
DeleteSetting	Left	Reset	Xor
Dim	Lib	Resume	Year
Dir	Line	Return	

2. 属性和方法的使用

类中出现混乱的一个方面是一个数是用属性还是方法来表示。这没有硬性规定，但一般情况下，如果该对象的外观和操作都像一个变量，就应使用属性来表示它(属性详见第3章)，即：

- 客户机代码应能读取它的值，最好不要使用只写属性，例如，应使用 SetPassword()方法，而不是 Password 只写属性。
- 读取该值不应花太长的时间。实际上，如果它是一个属性，通常表示读取过程花的时间相对较短。
- 读取该值不应有任何不希望的负面效应。设置属性的值，不应有与该属性不直接相关的负面效应。设置对话框的宽度会改变该对话框在屏幕上的外观，这是可以的，因为它与属性是相关的。
- 应可以用任何顺序设置属性。在设置属性时，最好不要因为还没有设置另一个相关的属性而抛出一个异常。例如，如果为了使用访问数据库的类，需要设置ConnectionString、UserName 和 Password，应确保已经执行了该类，这样用户才能按照任何顺序设置它们。
- 顺序读取属性也应有相同的效果。如果属性的值可能会出现预料不到的改变，就应把它编写为一个方法。在监视汽车运动的类中，把 speed 编写为属性就不是一种好的方式，而应使用 GetSpeed()，另一方面，应把 Weight 和 EngineSize 编写为属性，因为对于给定的对象，它们是不会改变的。

如果要编码的对象满足上述所有条件，就应对它使用属性，否则就应使用方法。

3. 字段的用法

字段的用法非常简单。字段应总是私有的，但在某些情况下也可以把常量或只读字段设置为公有，原因是如果把字段设置为公有，就可以在以后扩展或修改类。

遵循上面的规则就可以编写出好的代码，而且这些规则应与面向对编程的风格一起使用。

Microsoft 在保持一致性方面相当谨慎，在编写.NET 基类时遵循了它自己的规则。在编写.NET 代码时应很好地遵循这些规则，对于基类来说，就是类、成员、命名空间的命名方式和类层次结构的工作方式等，我们自己的类与基类的风格相同，有助于提高可读性和可维护性。

## 2.15 小结

本章介绍了一些 C# 基本语法，包括编写简单的 C# 程序需要掌握的内容。我们讲述了许多基础知识，但其中有许多是熟悉 C 风格语言(甚或 JavaScript)的开发人员能立即领悟的。

C# 语法与 C++/Java 语法非常类似，但仍存在一些小区别。在许多领域，将这些语法与功能结合起来，会使编码更快速，例如高质量的字符串处理功能。C# 还有一个强大的已定义类型系统，该系统基于值类型和引用类型的区别。下面两章将进一步介绍 C# 的面向对象编程特性。

# 第 3 章

## 对象和类型

到目前为止，我们介绍了组成 C#语言的主要内容，包括变量、数据类型和程序流语句，并简要介绍了一个只包含 `Main()` 方法的完整小例子。但还没有介绍如何把这些内容组合在一起，构成一个完整的程序，其关键就在于对类的处理。这就是本章的主题。本章的主要内容如下：

- 类和结构的区别
- 类成员
- 按值和引用传送参数
- 方法重载
- 构造函数和静态构造函数
- 只读字段
- 部分类
- 静态类
- `Object` 类，其他类型都从该类派生而来

第 4 章将介绍继承以及与继承相关的特性。

提示：

本章将讨论与类相关的基本语法，但假定您已经熟悉了使用类的基本原则，例如，知道构造函数和属性的含义，因此我们只是大致论述如何把这些原则应用于 C# 代码。

本章介绍的这些概念不一定得到了大多数面向对象语言的支持。例如对象构造函数是您熟悉的、使用广泛的一个概念，但静态构造函数就是 C# 的新增内容，所以我们将解释静态构造函数的工作原理。

### 3.1 类和结构

类和结构实际上都是创建对象的模板，每个对象都包含数据，并提供了处理和访问数据的方法。类定义了每个类对象(称为实例)可以包含什么数据和功能。例如，如果一个类表示一个顾客，就可以定义字段 `CustomerID`、`FirstName`、`LastName` 和 `Address`，以包含该顾客的信息。还可以定义处理存储在这些字段中的数据的功能。接着，就可以实例化这个类的对象，以表示某个顾客，并为这个实例设置这些字段，使用其功能。

```
class PhoneCustomer
{
    public const string DayOfSendingBill = "Monday";
```

```

public int CustomerID;
public string FirstName;
public string LastName;
}

```

结构与类的区别是它们在内存中的存储方式(类是存储在堆(heap)上的引用类型,而结构是存储在堆栈(stack)上的值类型)、访问方式和一些特征(如结构不支持继承)。较小的数据类型使用结构可提高性能。但在语法上,结构与类非常相似,主要的区别是使用关键字 `struct` 代替 `class` 来声明结构。例如,如果希望所有的 `PhoneCustomer` 实例都存储在堆栈上,而不是存储在托管堆上,就可以编写下面的语句:

```

struct PhoneCustomerStruct
{
    public const string DayOfSendingBill = "Monday";
    public int CustomerID;
    public string FirstName;
    public string LastName;
}

```

对于类和结构,都使用关键字 `new` 来声明实例:这个关键字创建对象并对其进行初始化。在下面的例子中,类和结构的字段值都默认为 0:

```

PhoneCustomer myCustomer = new PhoneCustomer(); //works for a class
PhoneCustomerStruct myCustomer2 = new PhoneCustomerStruct(); // works for a struct

```

在大多数情况下,类要比结构常用得多。因此,我们先讨论类,然后指出类和结构的区别,以及选择使用结构而不使用类的特殊原因。但除非特别说明,否则就可以假定用于类的代码也适用于结构。

## 3.2 类成员

类中的数据和函数称为类的成员。Microsoft 的正式术语对数据成员和函数成员进行了区分。除了这些成员外,类还可以包含嵌套的类型(例如其他类)。类中的所有成员都可以声明为 `public`(此时可以在类的外部直接访问它们)或 `private`(此时,它们只能由类中的其他代码来访问)。与 Visual Basic、C++ 和 Java 一样,C# 在这个方面还有变化,例如 `protected`(表示成员仅能由该成员所在的类及其派生类访问),第 4 章将详细解释各种访问级别。

### 3.2.1 数据成员

数据成员包含了类的数据——字段、常量和事件。数据成员可以是静态数据(与整个类相关)或实例数据(类的每个实例都有它自己的数据副本)。通常,对于面向对象的语言,类成员总是实例成员,除非用 `static` 进行了显式的声明。

字段是与类相关的变量。在前面的例子中已经使用了 `PhoneCustomer` 类中的字段。

一旦实例化 `PhoneCustomer` 对象,就可以使用语法 `Object.FieldName` 来访问这些字段:

```

PhoneCustomer Customer1 = new PhoneCustomer();
Customer1.FirstName = "Simon";

```

常量与类的关联方式同变量与类的关联方式一样。使用 `const` 关键字来声明常量。如果它们声明为 `public`，就可以在类的外部访问。

```
class PhoneCustomer
{
    public const string DayOfSendingBill = "Monday";
    public int CustomerID;
    public string FirstName;
    public string LastName;
}
```

事件是类的成员，在发生某些行为(例如改变类的字段或属性，或者进行了某种形式的用户交互操作)时，它可以让对象通知调用程序。客户可以包含所谓“事件处理程序”的代码来响应该事件。第7章将详细介绍事件。

### 3.2.2 函数成员

函数成员提供了操作类中数据的某些功能，包括方法、属性、构造函数和终结器(`finalizer`)、运算符以及索引器。

方法是与某个类相关的函数，它们可以是实例方法，也可以是静态方法。实例方法处理类的某个实例，静态方法提供了更一般的功能，不需要实例化一个类(例如 `Console.WriteLine()` 方法)。下一节介绍方法。

属性是可以在客户机上访问的函数组，其访问方式与访问类的公共字段类似。C#为读写类上的属性提供了专用语法，所以不必使用那些名称中嵌有 `Get` 或 `Set` 的偷工减料的方法。因为属性的这种语法不同于一般函数的语法，在客户代码中，虚拟的对象被当做实际的东西。

构造函数是在实例化对象时自动调用的函数。它们必须与所属的类同名，且不能有返回类型。构造函数用于初始化字段的值。

终结器类似于构造函数，但是在 CLR 检测到不再需要某个对象时调用。它们的名称与类相同，但前面有一个~符号。C++程序员应注意，终结器在 C#中比在 C++中用得少得多，因为 CLR 会自动进行垃圾收集，另外，不可能预测什么时候调用终结器。第12章将介绍终结器。

运算符执行的最简单的操作就是+和-。在对两个整数进行相加操作时，严格地说，就是对整数使用+运算符。C#还允许指定把已有的运算符应用于自己的类(运算符重载)。第6章将详细论述运算符。

索引器允许对象以数组或集合的方式进行索引。第6章介绍索引器。

#### 1. 方法

在 Visual Basic、C 和 C++中，可以定义与类完全不相关的全局函数，但在 C#中不能这样做。在 C#中，每个函数都必须与类或结构相关。

注意，正式的 C#术语实际上区分了函数和方法。在这个术语中，“函数成员”不仅包含方法，而且也包含类或结构的一些非数据成员，例如索引器、运算符、构造函数和析构函数等，甚至还有属性。这些都不是数据成员，字段、常量和事件才是数据成员。

##### (1) 方法的声明

在 C#中，定义方法的语法与 C 风格的语言相同，与 C++和 Java 中的语法也相同。与 C++



的主要语法区别是，在 C# 中，每个方法都单独声明为 `public` 或 `private`，不能使用 `public` 块把几个方法定义组合起来。另外，所有的 C# 方法都在类定义中声明和定义。在 C# 中，不能像在 C++ 中那样把方法的实现代码分隔开来。

在 C# 中，方法的定义包括方法的修饰符(例如方法的可访问性)、返回值的类型，然后是方法名、输入参数的列表(用圆括号括起来)和方法体(用花括号括起来)。

```
[modifiers] return_type MethodName([parameters])
{
    // Method body
}
```

每个参数都包括参数的类型名及在方法体中的引用名称。但如果方法有返回值，`return` 语句就必须与返回值一起使用，以指定出口点，例如：

```
public bool IsSquare(Rectangle rect)
{
    return (rect.Height == rect.Width);
}
```

这段代码使用了一个表示矩形的 .NET 基类 `System.Drawing.Rectangle`。

如果方法没有返回值，就把返回类型指定为 `void`，因为不能省略返回类型。如果方法不带参数，仍需要在方法名的后面写上一对空的圆括号 `()`(就像本章前面的 `Main()` 方法)。此时 `return` 语句就是可选的——当到达右花括号时，方法会自动返回。注意方法可以包含任意多个 `return` 语句：

```
public bool IsPositive(int value)
{
    if (value < 0)
        return false;
    return true;
}
```

## (2) 调用方法

C# 中调用方法的语法与 C++ 和 Java 中的一样，C# 和 Visual Basic 的唯一区别是在 C# 中调用方法时，必须使用圆括号，这要比 Visual Basic 6 中有时需要括号，有时不需要括号的规则简单一些。

下面的例子 `MathTest` 说明了类的定义和实例化、方法的定义和调用的语法。除了包含 `Main()` 方法的类之外，它还定义了类 `MathTest`，该类包含两个方法和一个字段。

```
using System;

namespace Wrox.ProCSharp. MathTestSample
{
    class MainEntryPoint
    {
        static void Main()
        {
            // Try calling some static functions
            Console.WriteLine("Pi is " + MathTest.GetPi());
            int x = MathTest.GetSquareOf(5);
            Console.WriteLine("Square of 5 is " + x);
        }
    }
}
```

```

// Instantiate at MathTest object
MathTest math = new MathTest(); // this is C#'s way of
                                // instantiating a reference type

// Call non-static methods
math.value = 30;
Console.WriteLine(
    "Value field of math variable contains " + math.value);
Console.WriteLine("Square of 30 is " + math.GetSquare());
}

// Define a class named MathTest on which we will call a method
class MathTest
{
    public int value;

    public int GetSquare()
    {
        return value*value;
    }

    public static int GetSquareOf(int x)
    {
        return x*x;
    }

    public static double GetPi()
    {
        return 3.14159;
    }
}

```

运行 mathTest 示例，会得到如下结果：

**csc MathTest.cs**

```

Microsoft (R) Visual C# Compiler version 9.00.20404
for Microsoft (R) .NET Framework version 3.5
Copyright (C) Microsoft Corporation. All rights reserved.

MathTest.exe
Pi is 3.14159
Square of 5 is 25
Value field of math variable contains 30
Square of 30 is 900

```

从代码中可以看出，MathTest 类包含一个字段和一个方法，该字段包含一个数字，该方法计算数字的平方。这个类还包含两个静态方法，一个返回 pi 的值，另一个计算作为参数传入的数字的平方。

这个类有一些功能并不是 C# 程序设计的好例子。例如，GetPi() 通常作为 const 字段来执行，而好的设计应使用目前还没有介绍的概念。

C++ 和 Java 开发人员应很熟悉这个例子的大多数语法。如果您有 Visual Basic 的编程经验，只需把 MathTest 类看作一个执行字段和方法的 Visual Basic 类模块。但无论使用什么语言，都要注意两个要点。

### (3) 给方法传递参数

参数可以通过引用或值传递给方法。在变量通过引用传递给方法时，被调用的方法得到的

就是这个变量，所以在方法内部对变量进行的任何改变在方法退出后仍旧发挥作用。而如果变量是通过值传送给方法的，被调用的方法得到的是变量的一个副本，也就是说，在方法退出后，对变量进行的修改会丢失。对于复杂的数据类型，按引用传递的效率更高，因为在按值传递时，必须复制大量的数据。

在 C# 中，所有的参数都是通过值来传递的，除非特别说明。这与 C++ 是相同的，但与 Visual Basic 相反。但是，在理解引用类型的传递过程时需要注意。因为引用类型的对象只包含对象的引用，它们只给方法传递这个引用，而不是对象本身，所以对底层对象的修改会保留下来。相反，值类型的对象包含的是实际数据，所以传递给方法的是数据本身的副本。例如，int 通过值传递给方法，方法对该 int 的值所作的任何改变都没有改变原 int 对象的值。但如果数组或其他引用类型(如类)传递给方法，方法就会使用该引用改变这个数组中的值，而新值会反射到原来的数组对象上。

下面的例子 ParameterTest.cs 说明了这一点：

```
using System;

namespace Wrox.ProCSharp. ParameterTestSample
{
    class ParameterTest
    {
        static void SomeFunction(int[] ints, int i)
        {
            ints[0] = 100;
            i = 100;
        }

        public static int Main()
        {
            int i = 0;
            int[] ints = { 0, 1, 2, 4, 8 };
            // Display the original values
            Console.WriteLine("i = " + i);
            Console.WriteLine("ints[0] = " + ints[0]);
            Console.WriteLine("Calling SomeFunction...");

            // After this method returns, ints will be changed,
            // but i will not
            SomeFunction(ints, i);
            Console.WriteLine("i = " + i);
            Console.WriteLine("ints[0] = " + ints[0]);
            return 0;
        }
    }
}
```

结果如下：

**csc ParameterTest.cs**

```
Microsoft (R) Visual C# Compiler version 9.00.20404
for Microsoft (R) .NET Framework version 3.5
Copyright (C) Microsoft Corporation. All rights reserved.

ParameterTest.exe
```

```
i = 0
ints[0] = 0
Calling SomeFunction...
i = 0
ints[0] = 100
```

注意，`i` 的值保持不变，而在 `ints` 中改变的值在原来的数组中也改变了。

注意字符串是不同的，因为字符串是不能改变的(如果改变字符串的值，就会创建一个全新的字符串)，所以字符串无法采用一般引用类型的行为方式。在方法调用中，对字符串所作的任何改变都不会影响原来的字符串。这一点将在第 8 章详细讨论。

#### (4) ref 参数

通过值传送变量是默认的，也可以迫使值参数通过引用传送给方法。为此，要使用 `ref` 关键字。如果把一个参数传递给方法，且这个方法的输入参数前带有 `ref` 关键字，则该方法对变量所作的任何改变都会影响原来对象的值：

```
static void SomeFunction(int[] ints, ref int i)
{
    ints[0] = 100;
    i = 100;           //the change to i will persist after SomeFunction() exits
}
```

在调用该方法时，还需要添加 `ref` 关键字：

```
SomeFunction(ints, ref i);
```

在 C# 中添加 `ref` 关键字等同于在 C++ 中使用 `&` 语法指定按引用传递参数。但是，C# 在调用方法时要求使用 `ref` 关键字，使操作更明确(因此有助于防止错误)。

最后，C# 仍要求对传递给方法的参数进行初始化，理解这一点也是非常重要的。在传递给方法之前，无论是按值传递，还是按引用传递，变量都必须初始化。

#### (5) out 关键字

在 C 风格的语言中，函数常常能从一个例程中输出多个值，这是使用输出参数实现的，只要把输出值赋给通过引用传递给方法的变量即可。通常，变量通过引用传送的初值是不重要的，这些值会被函数重写，函数甚至从来没有使用过它们。

如果可以在 C# 中使用这种约定，就会非常方便。但 C# 要求变量在被引用前必须用一个初值进行初始化。尽管在把输入变量传递给函数前，可以用没有意义的值初始化它们，因为函数将使用真实、有意义的值初始化它们，但是这样做是没有必要的，有时甚至会引起混乱。但有一种方法能够简化 C# 编译器所坚持的输入参数的初始化。

编译器使用 `out` 关键字来初始化。在方法的输入参数前面加上 `out` 关键字时，传递给该方法的变量可以不初始化。该变量通过引用传送，所以在从被调用的方法中返回时，方法对该变量进行的任何改变都会保留下来。在调用该方法时，还需要使用 `out` 关键字，与在定义该方法时一样：

```
static void SomeFunction(out int i)
{
    i = 100;
}

public static int Main()
```



```
{
    int i; // note how i is declared but not initialized
    SomeFunction(out i);
    Console.WriteLine(i);
    return 0;
}
```

out 关键字是 C# 中的新增内容，在 Visual Basic 和 C++ 中没有对应的关键字，该关键字的引入使 C# 更安全，更不容易出错。如果在函数体中没有给 out 参数分配一个值，该方法就不能编译。

#### (6) 方法的重载

C# 支持方法的重载——方法的几个有不同签名(方法名相同、但参数的个数和类型不同)的版本，但不支持 C++ 或 Visual Basic 中的默认参数。为了重载方法，只需声明同名但参数个数或类型不同的方法即可：

```
class ResultDisplayer
{
    void DisplayResult(string result)
    {
        // implementation
    }

    void DisplayResult(int result)
    {
        // implementation
    }
}
```

因为 C# 不直接支持可选参数，所以需要使用方法重载来达到此目的：

```
class MyClass
{
    int DoSomething(int x) // want 2nd parameter with default value 10
    {
        DoSomething(x, 10);
    }

    int DoSomething(int x, int y)
    {
        // implementation
    }
}
```

在任何语言中，对于方法重载来说，如果调用了错误的重载方法，就有可能出现运行错误。第 4 章将讨论如何使代码避免这些错误。现在，知道 C# 在重载方法的参数方面有一些小区别即可：

- 两个方法不能仅在返回类型上有区别。
- 两个方法不能仅根据参数是声明为 ref 还是 out 来区分。

## 2. 属性

属性(property)不太常见，因为它们表示的概念是 C# 从 Visual Basic 中提取的，而不是从 C++/Java 中提取的。属性的概念是：它是一个方法或一对方法，在客户机代码看来，它们是一



个字段。例如 Windows 窗体的 Height 属性。假定有下面的代码：

```
// mainForm is of type System.Windows.Form
mainForm.Height = 400;
```

执行这段代码，窗口的高度设置为 400，因此窗口会在屏幕上重新设置大小。在语法上，上面的代码类似于设置一个字段，但实际上是调用了属性访问器，它包含的代码重新设置了窗体的大小。

在 C# 中定义属性，可以使用下面的语法：

```
public string SomeProperty
{
    get
    {
        return "This is the property value";
    }
    set
    {
        // do whatever needs to be done to set the property
    }
}
```

get 访问器不带参数，且必须返回属性声明的类型。也不应为 set 访问器指定任何显式参数，但编译器假定它带一个参数，其类型也与属性相同，并表示为 value。例如，下面的代码包含一个属性 ForeName，它设置了一个字段 foreName，该字段有一个长度限制。

```
private string foreName;

public string ForeName
{
    get
    {
        return foreName;
    }
    set
    {
        if (value.Length > 20)
            // code here to take error recovery action
            // (eg. throw an exception)
        else
            foreName = value;
    }
}
```

注意这里的命名模式。我们采用 C# 的区分大小写模式，使用相同的名称，但公共属性采用 Pascal 大小写命名规则，而私有属性采用 camel 大小写命名规则。一些开发人员喜欢使用前面有下划线的字段名 `_foreName`，这会为识别字段提供极大的便利。

Visual Basic 6 程序员应注意，C# 不区分 Visual Basic 6 中的 Set 和 Let，在 C# 中，写入访问器总是用关键字 set 标识。

#### (1) 只读和只写属性

在属性定义中省略 set 访问器，就可以创建只读属性。因此，把上面例子中的 ForeName 变成只读属性：

```
private string foreName;

public string ForeName
{
    get
    {
        return foreName;
    }
}
```

同样，在属性定义中省略 `get` 访问器，就可以创建只写属性。但是，这是不好的编程方式，因为这可能会使客户机代码的作者感到迷惑。一般情况下，如果要这么做，最好使用一个方法替代。

### (2) 属性的访问修饰符

C# 允许给属性的 `get` 和 `set` 访问器设置不同的访问修饰符，所以属性可以有公共的 `get` 访问器和私有或受保护的 `set` 访问器。这有助于控制属性的设置方式或时间。在下面的例子中，注意 `set` 访问器有一个私有访问修饰符，而 `get` 访问器没有任何访问修饰符。这表示 `get` 访问器具有属性的访问级别。在 `get` 和 `set` 访问器中，必须有一个具备属性的访问级别。如果 `get` 访问器的访问级别是 `protected`，就会产生一个编译错误，因为这会使两个访问器的访问级别都不是属性。

```
public string Name
{
    get
    {
        return _name;
    }
    set
    {
        _name = value;
    }
}
```

### (3) 自动实现的属性

如果属性的 `set` 和 `get` 访问器中没有任何逻辑，就可以使用自动实现的属性。这种属性会自动实现基础成员变量。上例的代码如下：

```
public string ForeName {get; set;}
```

不需要声明 `private string foreName`。编译器会自动创建它。

使用自动实现的属性，就不能在属性设置中进行属性的有效性验证。所以在上面的例子中，不能检查 `foreName` 是否少于 20 个字符。但必须有两个访问器。尝试把该属性设置为只读属性，就会出错：

```
public string ForeName {get; }
```

但是，每个访问器的访问级别可以不同。因此，下面的代码是合法的：

```
public string ForeName {get; private set;}
```

### (4) 内联

一些开发人员可能会担心，在上一节中，我们列举了标准 C# 编码方式导致了非常小的函数

的许多情形，例如通过属性访问字段，而不是直接访问字段。这些额外的函数调用是否会增加系统开销，导致性能下降？其实，不需要担心这种编程方式会在 C# 中带来性能损失。C# 代码会编译为 IL，然后在运行期间进行正常的 JIT 编译，获得内部可执行代码。JIT 编译器可生成高度优化的代码，并在适当的时候内联代码(即用内联代码来替代函数调用)。如果某个方法或属性的执行代码仅是调用另一个方法，或返回一个字段，则该方法或属性肯定是内联的。但要注意，在何处内联代码的决定完全由 CLR 做出。我们无法使用像 C++ 中 `inline` 这样的关键字来控制哪些方法是内联的。

### 3. 构造函数

在 C# 中声明基本构造函数的语法与在 Java 和 C++ 中相同。下面声明一个与包含的类同名的方法，但该方法没有返回类型：

```
public class MyClass
{
    public MyClass()
    {
    }
    // rest of class definition
}
```

与 Java 和 C++ 相同，没有必要给类提供构造函数，在我们的例子中没有提供这样的构造函数。一般情况下，如果没有提供任何构造函数，编译器会在后台创建一个默认的构造函数。这是一个非常基本的构造函数，它只能把所有的成员字段初始化为标准的默认值(例如，引用类型为 `空` 引用，数字数据类型为 0，`bool` 为 `false`)。这通常就足够了，否则就需要编写自己的构造函数。

#### 注意：

对于 C++ 程序员来说，C# 中的基本字段在默认情况下初始化为 0，而 C++ 中的基本字段不进行初始化，不需要像 C++ 那样频繁地在 C# 中编写构造函数。

构造函数的重载遵循与其他方法相同的规则。换言之，可以为构造函数提供任意多的重载，只要它们的签名有明显的区别即可：

```
public MyClass() // zero-parameter constructor
{
    // construction code
}
public MyClass(int number) // another overload
{
    // construction code
}
```

但注意，如果提供了带参数的构造函数，编译器就不会自动提供默认的构造函数，只有在没有定义任何构造函数时，编译器才会自动提供默认的构造函数。在下面的例子中，因为定义了一个带单个参数的构造函数，所以编译器会假定这是可以使用的唯一构造函数，不会隐式地提供其他构造函数：

```
public class MyNumber
{
    private int number;
```

```
public MyNumber(int number)
{
    this.number = number;
}
```

上面的代码还说明，一般使用 `this` 关键字区分成员字段和同名的参数。如果试图使用无参数的构造函数实例化 `MyNumber` 对象，就会得到一个编译错误：

```
MyNumber numb = new MyNumber(); // causes compilation error
```

注意，可以把构造函数定义为 `private` 或 `protected`，这样不相关的类就不能访问它们：

```
public class MyNumber
{
    private int number;
    private MyNumber(int number) // another overload
    {
        this.number = number;
    }
}
```

在这个例子中，我们并没有为 `MyNumber` 定义任何公共或受保护的构造函数。这就使 `MyNumber` 不能使用 `new` 运算符在外部代码中实例化(但可以在 `MyNumber` 上编写一个公共静态属性或方法，以进行实例化)。这在下面两种情况下是有用的：

- 类仅用作某些静态成员或属性的容器，因此永远不会实例化
- 希望类仅通过调用某个静态成员函数来实例化(这就是所谓对象实例化的类代理方法)

#### (1) 静态构造函数

C#的一个新特征是也可以给类编写无参数的静态构造函数。这种构造函数只执行一次，而前面的构造函数是实例构造函数，只要创建类的对象，它都会执行。静态构造函数在 C++ 和 Visual Basic 6 中没有对应的函数。

```
class MyClass
{
    static MyClass()
    {
        // initialization code
    }
    // rest of class definition
}
```

编写静态构造函数的一个原因是，类有一些静态字段或属性，需要在第一次使用类之前，从外部源中初始化这些静态字段和属性。

.NET 运行库没有确保静态构造函数什么时候执行，所以不要把要求在某个特定时刻(例如，加载程序集时)执行的代码放在静态构造函数中。也不能预计不同类的静态构造函数按照什么顺序执行。但是，可以确保静态构造函数至多运行一次，即在代码引用类之前执行。在 C# 中，静态构造函数通常在第一次调用类的成员之前执行。

注意，静态构造函数没有访问修饰符，其他 C# 代码从来不调用它，但在加载类时，总是由 .NET 运行库调用它，所以像 `public` 和 `private` 这样的访问修饰符就没有意义了。同样，静态构造函数不能带任何参数，一个类也只能有一个静态构造函数。很显然，静态构造函数只能访



问类的静态成员，不能访问实例成员。

注意，无参数的实例构造函数可以在类中与静态构造函数安全共存。尽管参数列表是相同的，但这并不矛盾，因为静态构造函数是在加载类时执行，而实例构造函数是在创建实例时执行，所以构造函数的执行不会有冲突。

如果多个类都有静态构造函数，先执行哪个静态构造函数是不确定的。此时静态构造函数中的代码不应依赖其他静态构造函数的执行情况。另一方面，如果静态字段有默认值，它们就在调用静态构造函数之前指定。

下面用一个例子来说明静态构造函数的用法，该例子基于包含用户设置的程序(用户设置假定存储在某个配置文件中)。为了简单一些，假定只有一个用户设置——BackColor，表示要在应用程序中使用的背景色。因为这里不想编写从外部数据源中读取数据的代码，所以假定该设置在工作日的背景色是红色，在周末的背景色是绿色。程序仅在控制台窗口中显示设置——但这足以说明静态构造函数是如何工作的。

```
namespace Wrox.ProCSharp.StaticConstructorSample
{
    public class UserPreferences
    {
        public static readonly Color BackColor;

        static UserPreferences()
        {
            DateTime now = DateTime.Now;
            if (now.DayOfWeek == DayOfWeek.Saturday
                || now.DayOfWeek == DayOfWeek.Sunday)
                BackColor = Color.Green;
            else
                BackColor = Color.Red;
        }

        private UserPreferences()
        {
        }
    }
}
```

这段代码说明了颜色设置如何存储在静态变量中，该静态变量在静态构造函数中进行初始化。把这个字段声明为只读类型，表示其值只能在构造函数中设置。本章后面将详细介绍只读字段。这段代码使用了 Microsoft 在 Framework 类库中支持的两个有用的结构 `System.DateTime` 和 `System.Drawing.Color`。DateTime 结构实现了静态属性 `Now` 和实例属性 `DayOfWeek`，`Now` 属性返回当前的时间，`DayOfWeek` 属性可以计算出某个日期是星期几。`Color`(详见第 33 章)用于存储颜色，它实现了各种静态属性，例如本例使用的 `Red` 和 `Green`，返回常用的颜色。为了使用 `Color` 结构，需要在编译时引用 `System.Drawing.dll` 程序集，且必须为 `System.Drawing` 命名空间添加一个 `using` 语句：

```
using System;
using System.Drawing;
```

用下面的代码测试静态构造函数：

```
class MainEntryPoint
```



```

    {
        static void Main(string[] args)
        {
            Console.WriteLine("User-preferences: BackColor is: " +
                               UserPreferences.BackColor.ToString());
        }
    }

```

编译并运行这段代码，会得到如下结果：

**StaticConstructor.exe**

```
User-preferences: BackColor is: Color [Red]
```

当然，如果在周末执行代码，颜色设置就是 Green。

## (2) 从其他构造函数中调用构造函数

有时，在一个类中有几个构造函数，以容纳某些可选参数，这些构造函数都包含一些共同的代码。例如，下面的情况：

```

class Car
{
    private string description;
    private uint nWheels;
    public Car(string model, uint nWheels)
    {
        this.description = description;
        this.nWheels = nWheels;
    }

    public Car(string description)
    {
        this.description = description;
        this.nWheels = 4;
    }
    // etc.
}

```

这两个构造函数初始化了相同的字段，显然，最好把所有的代码放在一个地方。C#有一个特殊的语法，称为构造函数初始化器，可以实现此目的：

```

class Car
{
    private string description;
    private uint nWheels;

    public Car(string description, uint nWheels)
    {
        this.description = description;
        this.nWheels = nWheels;
    }

    public Car(string description) : this(model, 4)
    {
    }
    // etc
}

```

这里，`this` 关键字仅调用参数最匹配的那个构造函数。注意，构造函数初始化器在构造函数之前执行。现在假定运行下面的代码：

```
Car myCar = new Car("Proton Persona");
```

在本例中,在带一个参数的构造函数执行之前,先执行带2个参数的构造函数(但在本例中,因为带一个参数的构造函数没有代码,所以没有区别)。

C#构造函数初始化符可以包含对同一个类的另一个构造函数的调用(使用前面介绍的语法),也可以包含对直接基类的构造函数的调用(使用相同的语法,但应使用 `base` 关键字代替 `this`)。初始化符中不能有多个调用。

在C#中,构造函数初始化符的语法类似于C++中的构造函数初始化列表,但C++开发人员要注意,除了语法类似之外,C#初始化符所包含的代码遵循完全不同的规则。可以使用C++初始化列表指定成员变量的初始值,或调用基类构造函数,而C#初始化符中的代码只能调用另一个构造函数。这就要求C#类在构造时遵循严格的顺序,但C++就没有这个要求。这个问题详见第4章,那时就会看到,C#强制遵循的顺序只不过是良好的编程习惯而已。

### 3.2.3 只读字段

常量的概念就是一个包含不能修改的值的变量,常量是C#与大多数编程语言共有的。但是,常量不必满足所有的要求。有时可能需要一些变量,其值不应改变,但在运行之前其值是未知的。C#为这种情形提供了另一个类型的变量:只读字段。

`readonly` 关键字比 `const` 灵活得多,允许把一个字段设置为常量,但可以执行一些运算,以确定它的初始值。其规则是可以在构造函数中给只读字段赋值,但不能在其他地方赋值。只读字段还可以是一个实例字段,而不是静态字段,类的每个实例可以有不同的值。与 `const` 字段不同,如果要把只读字段设置为静态,就必须显式声明。

如果有一个编辑文档的MDI程序,因为要注册,需要限制可以同时打开的文档数。现在假定要销售该软件的不同版本,而且顾客可以升级他们的版本,以便同时打开更多的文档。显然,不能在源代码中对最大文档数进行硬编码,而是需要一个字段表示这个最大文档数。这个字段必须是只读的——每次安装程序时,从注册表键或其他文件存储中读取。代码如下所示:

```
public class DocumentEditor
{
    public static readonly uint MaxDocuments;

    static DocumentEditor()
    {
        MaxDocuments = DosomethingToFindOutMaxNumber();
    }
}
```

在本例中,字段是静态的,因为每次运行程序的实例时,只需存储最大文档数一次。这就是在静态构造函数中初始化它的原因。如果只读字段是一个实例字段,就要在实例构造函数中初始化它。例如,假定编辑的每个文档都有一个创建日期,但不允许用户修改它(因为这会覆盖过去的日期)。注意,该字段也是公共的,我们不需要把只读字段设置为私有,因为按照定义,它们不能在外部修改(这个规则也适用于常量)。

如前所述,日期用基类 `System.DateTime` 表示。下面的代码使用带有3个参数(年份、月份和月份中的日)的 `System.DateTime` 构造函数,可以从MSDN文档中找到这个构造函数和其他 `DateTime` 构造函数的更多信息。

```
public class Document
```

```

{
    public readonly DateTime CreationDate;

    public Document()
    {
        // Read in creation date from file. Assume result is 1 Jan 2002
        // but in general this can be different for different instances
        // of the class
        CreationDate = new DateTime(2002, 1, 1);
    }
}

```

在上面的代码中，CreationDate 和 MaxDocuments 的处理方式与其他字段相同，但因为它们是只读的，所以不能在构造函数外部赋值：

```

void SomeMethod()
{
    MaxDocuments = 10; // compilation error here. MaxDocuments is readonly
}

```

还要注意，在构造函数中不必给只读字段赋值，如果没有赋值，它的值就是其数据类型的默认值，或者在声明时给它初始化的值。这适用于静态和实例只读字段。

### 3.3 匿名类型

第 2 章讨论了 var 关键字，用于表示隐式类型化的变量。var 与 new 关键字一起使用时，可以创建匿名类型。匿名类型只是一个继承了 Object 的、没有名称的类。该类的定义从初始化器中推断，类似于隐式类型化的变量。

如果需要一个对象包含某个人的姓氏、中间名和名字，则声明如下：

```
var captain = new {FirstName = "James", MiddleName = "T", LastName = "Kirk"};
```

这会生成一个包含 FirstName、MiddleName 和 LastName 属性的对象。如果创建另一个对象，如下所示：

```
var doctor = new {FirstName = "Leonard", MiddleName = "", LastName = "McCoy"};
```

Captain 和 doctor 的类型就是相同的。例如，可以设置 captain = doctor。

如果所设置的值来自于另一个对象，初始化器就可以简化。如果已经有一个包含 FirstName、MiddleName 和 LastName 属性的类，且有一个该类的实例 person，captain 对象就可以初始化为：

```
var captain = new {person.FirstName, person.MidleName, person.LastName};
```

person 对象的属性名应投射为新对象名 captain。所以 captain 对象应有 FirstName、MiddleName 和 LastName 属性。

这些新对象的类型名是未知的。编译器为类型“伪造”了一个名称，但只有编译器才能使用它。我们不能也不应使用新对象上的任何类型引用，因为这不会得到一致的结果。

### 3.4 结构

前面介绍了类如何封装程序中的对象，也介绍了如何将它们保存在堆中，通过这种方式可以在数据的生存期上获得很大的灵活性，但性能会有一些损失。因托管堆的优化，这种性能损失比较小。但是，有时仅需要一个小的数据结构。此时，类提供的功能多于我们需要的功能，由于性能的原因，最好使用结构。看看下面的例子：

```
class Dimensions
{
    public double Length;
    public double Width;
}
```

上面的示例代码定义了类 `Dimensions`，它只存储了一个项的长度和宽度。假定编写一个安排设备的程序，让人们试着在计算机上重新安排设备，并存储每个设备项的维数。使字段变为公共字段，就会违背编程规则，但我们实际上并不需要类的全部功能。现在只有两个数字，把它们当作一对来处理，要比单个处理方便一些。既不需要很多方法，也不需要从类中继承，也不希望.NET 运行库在堆中遇到麻烦和性能问题，只需存储两个 `double` 类型的数据即可。

为此，只需修改代码，用关键字 `struct` 代替 `class`，定义一个结构而不是类，如本章前面所述：

```
struct Dimensions
{
    public double Length;
    public double Width;
}
```

为结构定义函数与为类定义函数完全相同。下面的代码演示了结构的构造函数和属性：

```
struct Dimensions
{
    public double Length;
    public double Width;

    Dimensions(double length, double width)
    {
        Length = length;
        Width = width;
    }

    public double Diagonal
    {
        get
        {
            return Math.Sqrt(Length * Length + Width * Width);
        }
    }
}
```

在许多方面，可以把 C# 中的结构看作是缩小的类。它们基本上与类相同，但更适合于把一些数据组合起来的场合。它们与类的区别在于：

- 结构是值类型，不是引用类型。它们存储在堆栈中或存储为内联(`inline`)(如果它们是另一个保存在堆中的对象的一部分)，其生存期的限制与简单的数据类型一样。



- 结构不支持继承。
- 结构的构造函数的工作方式有一些区别。尤其是编译器总是提供一个无参数的默认构造函数，这是不允许替换的。
- 使用结构，可以指定字段如何在内存中布局(第 13 章在介绍属性时将详细论述这个问题)。

因为结构实际上是把数据项组合在一起，有时大多数甚至全部字段都声明为 `public`。严格说来，这与编写 .NET 代码的规则相背——根据 Microsoft，字段(除了 `const` 字段之外)应总是私有的，并由公共属性封装。但是，对于简单的结构，许多开发人员都认为公共字段是可接受的编程方式。

注意：

C++开发人员要注意，C#中的结构在实现方式上与类大不相同。这与 C++的情形完全不同，在 C++中，类和结构是相同的对象。

下面将详细说明类和结构之间的区别。

### 3.4.1 结构是值类型

虽然结构是值类型，但在语法上常常可以把它们当作类来处理。例如，在上面的 `Dimensions` 类的定义中，可以编写下面的代码：

```
Dimensions point = new Dimensions();  
point.Length = 3;  
point.Width = 6;
```

注意，因为结构是值类型，所以 `new` 运算符与类和其他引用类型的工作方式不同。`new` 运算符并不分配堆中的内存，而是调用相应的构造函数，根据传送给它的参数，初始化所有的字段。对于结构，可以编写下述代码：

```
Dimensions point;  
point.Length = 3;  
point.Width = 6;
```

如果 `Dimensions` 是一个类，就会产生一个编译错误，因为 `point` 包含一个未初始化的引用——不指向任何地方的一个地址，所以不能给其字段设置值。但对于结构，变量声明实际上是为整个结构分配堆栈中的空间，所以就可以赋值了。但要注意下面的代码会产生一个编译错误，编译器会抱怨用户使用了未初始化的变量：

```
Dimensions point;  
Double D = point.Length;
```

结构遵循其他数据类型都遵循的规则：在使用前所有的元素都必须进行初始化。在结构上调用 `new` 运算符，或者给所有的字段分别赋值，结构就完全初始化了。当然，如果结构定义为类的成员字段，在初始化包含对象时，该结构会自动初始化为 0。

结构是值类型，所以会影响性能，但根据使用结构的方式，这种影响可能是正面的，也可能是负面的。正面的影响是为结构分配内存时，速度非常快，因为它们将内联或者保存在堆栈中。在结构超出了作用域被删除时，速度也很快。另一方面，只要把结构作为参数来传递或者



把一个结构赋给另一个结构(例如  $A=B$ , 其中  $A$  和  $B$  是结构), 结构的所有内容就被复制, 而对于类, 则只复制引用。这样, 就会有性能损失, 根据结构的大小, 性能损失也不同。注意, 结构主要用于小的数据结构。但当把结构作为参数传递给方法时, 就应把它作为 `ref` 参数传递, 以避免性能损失——此时只传递了结构在内存中的地址, 这样传递速度就与在类中的传递速度一样快了。另一方面, 如果这样做, 就必须注意被调用的方法可以改变结构的值。

### 3.4.2 结构和继承

结构不是为继承设计的。不能从一个结构中继承, 唯一的例外是结构(和 C# 中的其他类型一样)派生于类 `System.Object`。因此, 结构也可以访问 `System.Object` 的方法。在结构中, 甚至可以重写 `System.Object` 中的方法——例如重写 `ToString()` 方法。结构的继承链是: 每个结构派生于 `System.ValueType`, `System.ValueType` 派生于 `System.Object`。ValueType 并没有给 `Object` 添加任何新成员, 但提供了一些更适合结构的执行代码。注意, 不能为结构提供其他基类: 每个结构都派生于 `ValueType`。

### 3.4.3 结构的构造函数

为结构定义构造函数的方式与为类定义构造函数的方式相同, 但不允许定义无参数的构造函数。这看起来似乎没有意义, 其原因隐藏在 .NET 运行库的执行方式中。下述情况非常少见: .NET 运行库不能调用用户提供的定制无参数构造函数, 因此 Microsoft 采用一种非常简单的方式, 禁止在 C# 的结构内使用无参数的构造函数。

前面说过, 默认构造函数把所有的字段都初始化为 0, 且总是隐式地给出, 即使提供了其他带参数的构造函数, 也是如此。也不能提供字段的初始值, 以此绕过默认构造函数。下面的代码会产生编译错误:

```
struct Dimensions
{
    public double Length = 1;    // error. Initial values not allowed
    public double Width = 2;    // error. Initial values not allowed
}
```

当然, 如果 `Dimensions` 声明为一个类, 这段代码就不会有编译错误。

另外, 可以像类那样为结构提供 `Close()` 或 `Dispose()` 方法。

## 3.5 部分类

`partial` 关键字允许把类、结构或接口放在多个文件中。一般情况下, 一个类存储在单个文件中。但有时, 多个开发人员需要访问同一个类, 或者某种类型的代码生成器生成了一个类的某部分, 所以把类放在多个文件中是有益的。

`partial` 关键字的用法是: 把 `partial` 放在 `class`、`struct` 或 `interface` 关键字的前面。在下面的例子中, `TheBigClass` 类位于两个不同的源文件 `BigClassPart1.cs` 和 `BigClassPart2.cs` 中:

```
//BigClassPart1.cs
partial class TheBigClass
```

```
{
    public void MethodOne()
    {
    }
}
```

```
//BigClassPart2.cs
partial class TheBigClass
{
    public void MethodTwo()
    {
    }
}
```

编译包含这两个源文件的项目时，会创建一个 `TheBigClass` 类，它有两个方法 `MethodOne()` 和 `MethodTwo()`。

如果声明类时使用了下面的关键字，这些关键字将应用于同一个类的所有部分：

- `public`
- `private`
- `protected`
- `internal`
- `abstract`
- `sealed`
- `new`
- 一般约束

在嵌套的类型中，只要 `partial` 关键字位于 `class` 关键字的前面，就可以嵌套部分类。在把部分类编译到类型中时，会合并属性、XML 注释、接口、泛型类型的参数属性和成员。有如下两个源文件：

```
//BigClassPart1.cs
[CustomAttribute]
partial class TheBigClass : TheBigBaseClass, IBigClass
{
    public void MethodOne()
    {
    }
}
```

```
//BigClassPart2.cs
[AnotherAttribute]
partial class TheBigClass : IOtherBigClass
{
    public void MethodTwo()
    {
    }
}
```

编译后，源文件变成：

```
[CustomAttribute]
[AnotherAttribute]
partial class TheBigClass : TheBigBaseClass, IBigClass, IOtherBigClass
```

```

{
    public void MethodOne()
    {
    }

    public void MethodTwo()
    {
    }
}

```

## 3.6 静态类

本章前面讨论了静态构造函数，它们可以初始化静态的成员变量。如果类只包含静态的方法和属性，该类就是静态的。静态类在功能上与使用私有静态构造函数创建的类相同。不能创建静态类的实例。使用 `static` 关键字，编译器可以检查以后是否给该类添加了实例成员。如果是，就生成一个编译错误。这可以确保不创建静态类的实例。静态类的语法如下所示：

```

static class StaticUtilities
{
    public static void HelperMethod()
    {
    }
}

```

调用 `HelperMethod()` 不需要 `StaticUtilities` 类型的对象。使用类型名即可进行该调用：

```
StaticUtilities.HelperMethod();
```

## 3.7 Object 类

前面提到，所有的 .NET 类都派生于 `System.Object`。实际上，如果在定义类时没有指定基类，编译器就会自动假定这个类派生于 `Object`。本章没有使用继承，所以前面介绍的每个类都派生于 `System.Object`（如前所述，对于结构，这个派生是间接的：结构总是派生于 `System.ValueType`，`System.ValueType` 派生于 `System.Object`）。

其重要性在于，除了自己定义的方法和属性外，还可以访问为 `Object` 定义的许多公共或受保护的成员方法。这些方法可以用于自己定义的所有其他类中。

### 3.7.1 System.Object 方法

下面将简要总结每个方法的作用，下一节详细论述 `ToString()` 方法。

- `ToString()` 方法：是获取对象的字符串表示的一种便捷方式。当只需要快速获取对象的内容，以用于调试时，就可以使用这个方法。在数据的格式化方面，它提供的选择非常少：例如，日期在原则上可以表示为许多不同的格式，但 `DateTime.ToString()` 没有在这方面提供任何选择。如果需要更专业的字符串表示，例如考虑用户的格式化配置或文化(区域)，就应实现 `IFormattable` 接口(详见第 8 章)。

- **GetHashCode()方法**: 如果对象放在名为映射(也称为散列表或字典)的数据结构中, 就可以使用这个方法。处理这些结构的类使用该方法确定把对象放在结构的什么地方。如果希望把类用作字典的一个键, 就需要重写 **GetHashCode()方法**。对该方法重载的执行方式有一些相当严格的限制, 这些将在第 10 章介绍字典时讨论。
- **Equals()方法(两个版本)和 ReferenceEquals()方法**: 如果把 3 个用于比较对象相等性的不同方法组合起来, 就说明 .NET Framework 在比较相等性方面有相当复杂的模式。这 3 个方法和比较运算符 `==` 在使用方式上有微妙的区别。而且, 在重写带一个参数的虚拟 **Equals()方法** 时也有一些限制, 因为 **System.Collections** 命名空间中的一些基类要调用该方法, 并希望它以特定的方式执行。第 6 章在介绍运算符时将探讨这些方法的使用。
- **Finalize()方法**: 第 12 章将介绍这个方法, 它最接近 C++ 风格的析构函数, 在引用对象被回收, 以清理资源时调用。**Finalize()方法** 的 **Object** 执行代码实际上什么也没有做, 因而被垃圾收集器忽略。如果对象拥有对未托管资源的引用, 则在该对象被删除时, 就需要删除这些引用, 此时一般要重写 **Finalize()**。垃圾收集器不能直接重写该方法, 因为它只负责托管的资源, 只能依赖用户提供的 **Finalize()**。
- **GetType()方法**: 这个方法返回从 **System.Type** 派生的类的一个实例。这个对象可以提供对象所属类的更多信息, 包括基本类型、方法、属性等。**System.Type** 还提供了 .NET 反射技术的入口。这个主题详见第 13 章。
- **MemberwiseClone()方法**: 这是 **System.Object** 中唯一没有在本书的其他地方详细论述的方法。不需要讨论这个方法, 因为它在概念上相当简单, 只是复制对象, 返回一个对副本的引用(对于值类型, 就是一个装箱的引用)。注意, 得到的副本是一个浅表复制, 即它复制了类中的所有值类型。如果类包含内嵌的引用, 就只复制引用, 而不复制引用的对象。这个方法是受保护的, 所以不能用于复制外部的对象。该方法不是虚拟的, 所以不能重写它的实现代码。

### 3.7.2 ToString()方法

第 2 章已经提到了 **ToString()** 方法, 它是快速获取对象的字符串表示的最便捷的方式。例如:

```
int i = -50;
string str = i.ToString(); // returns "-50"
```

下面是另一个例子:

```
enum Colors {Red, Orange, Yellow};
// later on in code...
Colors favoriteColor = Colors.Orange;
string str = favoriteColor.ToString(); // returns "Orange"
```

**Object.ToString()** 声明为虚类型, 在这些例子中, 该方法的实现代码都是为 C# 预定义数据类型重写过的代码, 以返回这些类型的正确字符串表示。**Colors** 枚举是一个预定义的数据类型, 它实际上实现为一个派生于 **System.Enum** 的结构, 而 **System.Enum** 有一个相当聪明的 **ToString()** 重写方法, 来处理用户定义的所有枚举。



如果不在自己定义的类中重写 `ToString()`，该类将只继承 `System.Object` 执行方式——显示类的名称。如果希望 `ToString()` 返回一个字符串，其中包含类中对象的值信息，就需要重写它。下面用一个例子 `Money` 来说明这一点。在该例子中，定义一个非常简单的类 `Money`，表示美元数。`Money` 是 `decimal` 类的包装器，提供了一个 `ToString()` 方法。注意，这个方法必须声明为 `override`，因为它将替代(重写)`Object` 提供的 `ToString()` 方法。第4章将详细讨论重写。该例子的完整代码如下所示(注意它还说明了如何使用属性封装字段):

```
using System;

namespace Wrox.ProCSharp.OOCSharp
{
    class MainEntryPoint
    {
        static void Main(string[] args)
        {
            Money cash1 = new Money();
            cash1.Amount = 40M;
            Console.WriteLine("cash1.ToString() returns: " + cash1.ToString());
            Console.ReadLine();
        }
    }
    class Money
    {
        private decimal amount;

        public decimal Amount
        {
            get
            {
                return amount;
            }
            set
            {
                amount = value;
            }
        }

        public override string ToString()
        {
            return "$" + Amount.ToString();
        }
    }
}
```

这个例子仅说明了 C# 的语法特性。C# 已经有表示货币的预定义类型 `decimal`。所以在现实生活中，不必编写这样的类来重复该功能，除非要给它添加其他方法。在许多情况下，由于格式化要求，也可以使用 `String.Format()` 方法(详见第8章)来表示货币字符串，而不是 `ToString()`。

在 `Main()` 方法中，先实例化一个 `Money` 对象，再调用 `ToString()`，执行该方法的重写版本。运行这段代码，会得到如下结果：

```
StringRepresentations
cash1.ToString() returns: $40
```



## 3.8 扩展方法

有许多方法扩展类。如果有类的源代码，继承（如第 4 章所述）就是给对象添加功能的好方法。但如果没有源代码，该怎么办？此时可以使用扩展方法，它允许改变一个类，但不需要类的源代码。

扩展方法是静态方法，是类的一部分，但实际上没有放在类的源代码中。假定上例中的 `Money` 类需要一个方法 `AddToAmount(decimal amountToAdd)`。但是，由于某种原因，程序集最初的源代码不能直接修改。此时就可以创建一个静态类，把方法 `AddToAmount` 添加为一个静态方法。代码如下：

```
namespace Chapter3.Extensions
{
    public static class MoneyExtension
    {
        public static void AddToAmount(this Money money, decimal amountToAdd)
        {
            money.Amount += amountToAdd;
        }
    }
}
```

注意 `AddToAmount` 方法的参数。对于扩展方法，第一个参数是要扩展的类型，它放在 `this` 关键字的后面。这告诉编译器，这个方法是 `Money` 类型的一部分。在这个例子中，`Money` 是要扩展的类型。在扩展方法中，可以访问所扩展类型的所有公共方法和属性。

在主程序中，`AddToAmount` 方法看起来像是另一个方法。它没有显示第一个参数，也不能对它进行任何处理。要使用新方法，需要执行如下调用，这与其他方法相同：

```
cash1.AddToAmount(10M);
```

即使扩展方法是静态的，也要使用标准的实例方法语法。注意这里使用 `cash1` 实例变量来调用 `AddToAmount`，而没有使用类型名。

如果扩展方法与类中的某个方法同名，扩展方法就从来不会被调用。类中已有的实例方法优先。

## 3.9 小结

本章介绍了 C# 中声明和处理对象的语法，论述了如何声明静态和实例字段、属性、方法和构造函数。还讨论了 C# 中新增的、其他语言的 OOP 模型中没有的新特性：静态构造函数提供了初始化静态字段的方式，利用结构可以定义高性能的类型，不需要使用托管的堆。我们还阐述了 C# 中的所有类型最终都派生于类 `System.Object`，这说明所有的类型都拥有一组基本的方法，包括 `ToString()`。

本章多次提到了继承，第 4 章将介绍 C# 中的实现(implementation)继承和接口继承。

# 第 4 章

## 继 承

第 3 章介绍了如何使用 C# 中的各个类，其重点是如何定义方法、构造函数、属性和单个类(或单个结构)中的其他成员。我们指出，所有的类最终都派生于 `System.Object` 类，但并没有说明如何创建继承类的层次结构。继承是本章的主题。我们将讨论 C# 和 .NET Framework 如何处理继承。本章的主要内容如下：

- 继承的类型
- 实现继承
- 访问修饰符
- 接口

### 4.1 继承的类型

首先介绍 C# 在继承方面支持和不支持的功能。

#### 4.1.1 实现继承和接口继承

在面向对象的编程中，有两种截然不同的继承类型：实现继承和接口继承。

- 实现继承：表示一个类型派生于一个基类型，拥有该基类型的所有成员字段和函数。在实现继承中，派生类型的每个函数采用基类型的实现代码，除非在派生类型的定义中指定重写该函数的实现代码。在需要给现有的类型添加功能，或许多相关的类型共享一组重要的公共功能时，这种类型的继承是非常有效的。例如第 31 章讨论的 `Windows Forms` 类。第 31 章也讨论了基类 `System.Windows.Forms.Control`，该类提供了常用 Windows 控件的非常复杂的实现代码，第 31 章还讨论了许多其他的类，例如 `System.Windows.Forms.TextBox` 和 `System.Windows.Forms.ListBox`，这两个类派生于 `Control`，并重写了函数，或提供了新的函数，以实现特定类型的控件。
- 接口继承：表示一个类型只继承了函数的签名，没有继承任何实现代码。在需要指定该类型具有某些可用的特性时，最好使用这种类型的继承。例如，某些类型可以指定从接口 `System.IDisposable`(详见第 12 章)中派生，从而提供一种清理资源的方法 `Dispose()`。由于某种类型清理资源的方式可能与另一种类型的完全不同，所以定义通用的实现代码是没有意义的，此时就适合使用接口继承。接口继承常常被看做提供了一种契约：让类型派生于接口，来保证为客户提供某个功能。

在传统上，像 C++ 这样的语言在实现继承方面的功能非常强大。实际上，实现继承是 C++ 编程模型的核心。另一方面，VB6 不支持类的任何实现继承，但因其底层的 COM 基础体系，所以它支持接口继承。

在 C# 中，既有实现继承，也有接口继承。它们没有强弱之分，因为这两种继承都完全内置于语言中，因此很容易为不同的情形选择最好的体系结构。

### 4.1.2 多重继承

一些语言如 C++ 支持所谓的“多重继承”，即一个类派生于多个类。使用多重继承的优点是有争议的：一方面，毫无疑问，可以使用多重继承编写非常复杂、但很紧凑的代码，如 C++ ATL 库。另一方面，使用多重实现继承的代码常常很难理解和调试(这也可以从 C++ ATL 库中看出)。如前所述，使健壮代码的编写容易一些，是开发 C# 的重要设计目标。因此，C# 不支持多重实现继承。而 C# 又允许类型派生于多个接口。这说明，C# 类可以派生于另一个类和任意多个接口。更准确地说，因为 `System.Object` 是一个公共的基类，所以每个 C# 类(除了 `Object` 类之外)都有一个基类，还可以有任意多个基接口。

### 4.1.3 结构和类

第 3 章区分了结构(值类型)和类(引用类型)。使用结构的一个限制是结构不支持继承，但每个结构都自动派生于 `System.ValueType`。实际上还应更仔细一些：不能建立结构的类型层次，但结构可以实现接口。换言之，结构并不支持实现继承，但支持接口继承。事实上，定义结构和类可以总结为：

- 结构总是派生于 `System.ValueType`，它们还可以派生于任意多个接口。
- 类总是派生于用户选择的另一个类，它们还可以派生于任意多个接口。

## 4.2 实现继承

如果要声明一个类派生于另一个类，可以使用下面的语法：

```
class MyDerivedClass : MyBaseClass
{
    // functions and data members here
}
```

注意：

这个语法非常类似于 C++ 和 Java 中的语法，但是，C++ 程序员习惯于使用公共和私有继承的概念，要注意 C# 不支持私有继承，因此基类名上没有 `public` 或 `private` 限定符。支持私有继承会大大增加语言的复杂性，实际上私有继承在 C++ 中也很少使用。

如果类(或结构)也派生于接口，则用逗号分隔开基类和接口：

```
public class MyDerivedClass : MyBaseClass, IInterface1, IInterface2
{
    // ...
}
```

```
//etc.  
}
```

对于结构，语法如下：

```
public struct MyDerivedStruct : IInterface1, IInterface2  
{  
    //etc.  
}
```

如果在类定义中没有指定基类，C#编译器就假定 `System.Object` 是基类。因此下面的两段代码生成相同的结果：

```
class MyClass : Object //derives from System.Object  
{  
    //etc.  
}
```

和

```
class MyClass //derives from System.Object  
{  
    //etc.  
}
```

第二种形式比较常用，因为它较简单。

C#支持 `object` 关键字，它用作 `System.Object` 类的假名，所以也可以编写下面的代码：

```
class MyClass : object //derives from System.Object  
{  
    //etc.  
}
```

如果要引用 `Object` 类，可以使用 `object` 关键字，智能编辑器(如 Visual Studio)会识别它，因此便于编辑代码。

### 4.2.1 虚方法

把一个基类函数声明为 `virtual`，该函数就可以在派生类中重写了：

```
class MyBaseClass  
{  
    public virtual string VirtualMethod()  
    {  
        return "This method is virtual and defined in MyBaseClass";  
    }  
}
```

也可以把属性声明为 `virtual`。对于虚属性或重写属性，语法与非虚属性是相同的，但要在定义中加上关键字 `virtual`，其语法如下所示：

```
public virtual string ForeName  
{  
    get { return foreName; }  
    set { foreName = value; }  
}  
private string foreName;
```

为了简单起见，下面的讨论将主要集中于方法，但其规则也适用于属性。

C#中虚函数的概念与标准 OOP 概念相同：可以在派生类中重写虚函数。在调用方法时，会调用对象类型的合适方法。在 C#中，函数在默认情况下不是虚拟的，但(除了构造函数以外)可以显式地声明为 `virtual`。这遵循 C++的方式，即从性能的角度来看，除非显式指定，否则函数就不是虚拟的。而在 Java 中，所有的函数都是虚拟的。但 C#的语法与 C++的语法不同，因为 C#要求在派生类的函数重写另一个函数时，要使用 `override` 关键字显式声明：

```
class MyDerivedClass : MyBaseClass
{
    public override string VirtualMethod()
    {
        return "This method is an override defined in MyDerivedClass";
    }
}
```

方法重写的语法避免了 C++中很容易发生的潜在运行错误：当派生类的方法签名无意中与基类版本略有差别时，派生类方法就不能重写基类方法了。在 C#中，这会出现一个编译错误，因为编译器会认为函数已标记为 `override`，但没有重写它的基类方法。

成员字段和静态函数都不能声明为 `virtual`，因为这个概念只对类中的实例函数成员有意义。

#### 4.2.2 隐藏方法

如果签名相同的方法在基类和派生类中都进行了声明，但该方法没有声明为 `virtual` 和 `override`，派生类方法就会隐藏基类方法。

在大多数情况下，是要重写方法，而不是隐藏方法，因为隐藏方法会存在为给定类的实例调用错误方法的危险。但是，如下面的例子所示，C#语法可以确保开发人员在编译时收到这个潜在错误的警告，使隐藏方法更加安全。这也是类库开发人员得到的版本方面的好处。

假定有人编写了类 `HisBaseClass`：

```
class HisBaseClass
{
    // various members
}
```

在将来的某一刻，要编写一个派生类，给 `HisBaseClass` 添加某个功能，特别是要添加一个目前基类中没有的方法 `MyGroovyMethod()`：

```
class MyDerivedClass: HisBaseClass
{
    public int MyGroovyMethod()
    {
        // some groovy implementation
        return 0;
    }
}
```

一年后，基类的编写者决定扩展基类的功能。为了保持一致，他也添加了一个名为 `MyGroovyMethod()` 的方法，该方法的名称和签名与前面添加的方法相同，但并不完成相同的工作。在使用基类的新方法编译代码时，程序在应该调用哪个方法上就会有潜在的冲突。这在 C#



中完全合法,但因为我们的 `MyGroovyMethod()` 与基类的 `MyGroovyMethod()` 不相关,运行这段代码的结果就可能不是我们想要的结果。C#已经为此设计了一种方式,可以很好地处理这种情况。

首先,系统会发出警告。在 C# 中,应使用 `new` 关键字声明我们要隐藏一个方法,如下所示:

```
class MyDerivedClass : HisBaseClass
{
    public new int MyGroovyMethod()
    {
        // some groovy implementation
        return 0;
    }
}
```

但是,我们的 `MyGroovyMethod()` 没有声明为 `new`,所以编译器会认为它隐藏了基类的方法,但没有显式声明,因此发出一个警告(这也适用于把 `MyGroovyMethod()` 声明为 `virtual`)。如果愿意,可以给我们的方法重命名。这么做,是最好的情形,因为这会避免许多冲突。但是,如果觉得重命名方法是不可能的(例如,已经为其他公司把软件发布为一个库,所以无法修改方法的名称),则所有的已有客户机代码仍能正确运行,选择我们的 `MyGroovyMethod()`。这是因为访问这个方法的已有代码必须通过对 `MyDerivedClass`(或进一步派生的类)的引用进行选择。

已有的代码不能通过对 `HisBaseClass` 的引用访问这个方法,因为在对 `HisBaseClass` 的早期版本进行编译时,会产生一个编译错误。这个问题只会发生在将来编写的客户机代码上。C#会发出一个警告,告诉用户在将来的代码中可能会出问题——用户应注意这个警告,不要试图在将来的代码中通过对 `HisBaseClass` 的引用调用 `MyGroovyMethod()` 方法,但所有已有的代码仍会正常工作。这是比较微妙的,但很好地说明了 C# 如何处理类的不同版本。

#### 4.2.3 调用函数的基类版本

C#有一种特殊的语法用于从派生类中调用方法的基类版本: `base.<MethodName>()`。例如,假定派生类中的一个方法要返回基类的方法返回的值的 90%,就可以使用下面的语法:

```
class CustomerAccount
{
    public virtual decimal CalculatePrice()
    {
        // implementation
        return 0.0M;
    }
}
class GoldAccount : CustomerAccount
{
    public override decimal CalculatePrice()
    {
        return base.CalculatePrice() * 0.9M;
    }
}
```

这个语法类似于 Java,但 Java 使用关键字 `super` 而不是 `base`。C++ 没有类似的关键字,但需要显式指定类名(`CustomerAccount::CalculatePrice()`)。C++ 中对应于 `base` 的内容都比较模糊,因此 C++ 允许多重继承。

注意，可以使用 `base.<MethodName>()` 语法调用基类中的任何方法，不必在同一个方法的重载中调用它。

#### 4.2.4 抽象类和抽象函数

C# 允许把类和函数声明为 `abstract`，抽象类不能实例化，而抽象函数没有执行代码，必须在非抽象的派生类中重写。显然，抽象函数也是虚拟的(但也不需要提供 `virtual` 关键字，实际上，如果提供了该关键字，就会产生一个语法错误)。如果类包含抽象函数，该类将也是抽象的，也必须声明为抽象的：

```
abstract class Building
{
    public abstract decimal CalculateHeatingCost(); // abstract method
}
```

C++ 开发人员要注意 C# 中的一些语法区别。C# 不能采用 `=0` 语法来声明抽象函数。在 C# 中，这个语法有误导作用，因为可以在类声明的成员字段上使用 `=<value>`，提供初始值：

```
abstract class Building
{
    private bool damaged = false; // field
    public abstract decimal CalculateHeatingCost(); // abstract method
}
```

注意：

C++ 开发人员还要注意术语上的细微差别：在 C++ 中，抽象函数常常描述为纯虚函数，而在 C# 中，仅使用抽象这个术语。

#### 4.2.5 密封类和密封方法

C# 允许把类和方法声明为 `sealed`。对于类来说，这表示不能继承该类；对于方法来说，这表示不能重写该方法。

```
sealed class FinalClass
{
    // etc
}
class DerivedClass : FinalClass // wrong. Will give compilation error
{
    // etc
}
```

注意：

Java 开发人员可以把 C# 中的 `sealed` 当作 Java 中的 `final`。

在把类或方法标记为 `sealed` 时，最可能的情形是：如果要对库、类或自己编写的其他类进行操作，则重写某些功能会导致错误。也可以因商业原因把类或方法标记为 `sealed`，以防第三方以违反注册协议的方式扩展该类。但一般情况下，在把类或方法标记为 `sealed` 时要小心，因为这么做会严重限制它的使用。即使不希望它能继承一个类或重写类的某个成员，仍有可能在

将来的某个时刻，有人会遇到我们没有预料到的情形。.NET 基类库大量使用了密封类，使希望从这些类中派生出自己的类的第三方开发人员无法访问这些类。例如 `string` 就是一个密封类。

把方法声明为 `sealed` 也可以实现类似的目的，但很少这么做。

```
class MyClass
{
    public sealed override void FinalMethod()
    {
        // etc.
    }
}
class DerivedClass : MyClass
{
    public override void FinalMethod()    // wrong. Will give compilation error
    {
    }
}
```

要在方法或属性上使用 `sealed` 关键字，必须先基类上把它声明为重写。如果基类上不希望有重写的方法或属性，就不要把它声明为 `virtual`。

#### 4.2.6 派生类的构造函数

第3章介绍了单个类的构造函数是如何工作的。这样，就产生了一个有趣的问题，在开始为层次结构中的类(这个类继承了其他类，也可能有定制的构造函数)定义自己的构造函数时，会发生什么情况？

假定没有为类定义任何显式的构造函数，这样编译器就会为所有的类提供默认的构造函数，在后台会进行许多操作，编译器可以很好地解决层次结构中的所有问题，每个类中的每个字段都会初始化为默认值。但在添加了一个我们自己的构造函数后，就要通过派生类的层次结构高效地控制构造过程，因此必须确保构造过程顺利进行，不要出现不能按照层次结构进行构造的问题。

为什么派生类会有某些特殊的问题？原因是在创建派生类的实例时，实际上会有多个构造函数起作用。要实例化的类的构造函数本身不能初始化类，还必须调用基类中的构造函数。这就是为什么要通过层次结构进行构造的原因。

为了说明为什么必须调用基类的构造函数，下面是手机公司 MortimerPhones 开发的一个例子。这个例子包含一个抽象类 `GenericCustomer`，它表示顾客。还有一个(非抽象)类 `Nevermore60Customer`，它表示采用特定付费方式(称为 `Nevermore60` 付费方式)的顾客。所有的顾客都有一个名字，由一个私有字段表示。在 `Nevermore60` 付费方式中，顾客前几分钟的电话费比较高，需要一个字段 `highCostMinutesUsed`，它详细说明了每个顾客该如何支付这些较高的电话费。抽象类 `GenericCustomer` 的定义如下所示：

```
abstract class GenericCustomer
{
    private string name;
    // lots of other methods etc.
}
class Nevermore60Customer : GenericCustomer
{
```

```
private uint highCostMinutesUsed;
// other methods etc.
}
```

不要担心在这些类中执行的其他方法，因为这里仅考虑构造过程。如果下载了本章的示例代码，就会发现类的定义仅包含构造函数。

下面看看使用 `new` 运算符实例化 `Nevermore60Customer` 时，会发生什么情况：

```
GenericCustomer customer = new Nevermore60Customer();
```

显然，成员字段 `name` 和 `highCostMinutesUsed` 都必须在实例化 `customer` 时进行初始化。如果没有提供自己的构造函数，而是仅依赖默认的构造函数，`name` 就会初始化为 `null` 引用，`highCostMinutesUsed` 初始化为 0。下面详细讨论其过程。

`highCostMinutesUsed` 字段没有问题：编译器提供的默认 `Nevermore60Customer` 构造函数会把它初始化为 0。

那么 `name` 呢？看看类定义，显然，`Nevermore60Customer` 构造函数不能初始化这个值。字段 `name` 声明为 `private`，这意味着派生的类不能访问它。默认的 `Nevermore60Customer` 构造函数甚至不知道存在这个字段。唯一知道这个字段的是 `GenericCustomer` 的其他成员，即如果对 `name` 进行初始化，就必须在 `GenericCustomer` 的某个构造函数中进行。无论类层次结构有多大，这种情况都会一直延续到最终的基类 `System.Object` 上。

理解了上面的问题后，就可以明白实例化派生类时会发生什么样的情况了。假定默认的构造函数在整个层次结构中使用：编译器首先找到它试图实例化的类的构造函数，在本例中是 `Nevermore60Customer`，这个默认 `Nevermore60Customer` 构造函数首先要做的是为其直接基类 `GenericCustomer` 运行默认构造函数，然后 `GenericCustomer` 构造函数为其直接基类 `System.Object` 运行默认构造函数，`System.Object` 没有任何基类，所以它的构造函数就执行，并把控制权返回给 `GenericCustomer` 构造函数。现在执行 `GenericCustomer` 构造函数，把 `name` 初始化为 `null`，再把控制权返回给 `Nevermore60Customer` 构造函数，接着执行这个构造函数，把 `highCostMinutesUsed` 初始化为 0，并退出。此时，`Nevermore60Customer` 实例就已经成功地构造和初始化了。

构造函数的调用顺序是先调用 `System.Object`，再按照层次结构由上向下进行，直到到达编译器要实例化的类为止。还要注意在这个过程中，每个构造函数都初始化它自己的类中的字段。这是它的一般工作方式，在开始添加自己的构造函数时，也应尽可能遵循这个规则。

注意构造函数的执行顺序。基类的构造函数总是最先调用。也就是说，派生类的构造函数可以在执行过程中调用它可以访问的基类方法、属性和其他成员，因为基类已经构造出来了，其字段也初始化了。如果派生类不喜欢初始化基类的方式，但要访问数据，就可以改变数据的初始值，但是，好的编程方式应尽可能避免这种情况，让基类构造函数来处理其字段。

理解了构造过程后，就可以开始添加自己的构造函数了。

### 1. 在层次结构中添加无参数的构造函数

首先讨论最简单的情况，在层次结构中用一个无参数的构造函数来替换默认的构造函数后，看看会发生什么情况。假定要把每个人的名字初始化为 `<no name>`，而不是 `null` 引用，修改 `GenericCustomer` 中的代码，如下所示：



```
public abstract class GenericCustomer
{
    private string name;
    public GenericCustomer()
        : base() // we could omit this line without affecting the compiled code
    {
        name = "<no name>";
    }
}
```

添加这段代码后，代码运行正常。Nevermore60Customer 仍有自己的默认构造函数，所以上面描述的事件顺序仍不变，但编译器会使用定制的 GenericCustomer 构造函数，而不是生成默认的构造函数，所以 name 字段按照需要总是初始化为<no name>。

注意，在定制的构造函数中，在执行 GenericCustomer 构造函数前，添加了一个对基类构造函数的调用，使用的语法与前面解释如何让构造函数的不同重载版本互相调用时使用的语法相同。唯一的区别是，这次使用的关键字是 base，而不是 this，表示这是基类的构造函数，而不是要调用的类的构造函数。在 base 关键字后面的圆括号中没有参数，这是非常重要的，因为没有给基类构造函数传送参数，所以编译器会调用无参数的构造函数。其结果是编译器会插入调用 System.Object 构造函数的代码，这正好与默认情况相同。

实际上，可以把这行代码删除，只加上为本章中大多数构造函数编写的代码：

```
public GenericCustomer()
{
    name = "<no name>";
}
```

如果编译器没有在起始花括号的前面找到对另一个构造函数的任何引用，它就会假定我们要调用基类构造函数——这符合默认构造函数的工作方式。

base 和 this 关键字是调用另一个构造函数时允许使用的唯一关键字，其他关键字都会产生编译错误。还要注意只能指定一个其他的构造函数。

到目前为止，这段代码运行正常。但是，要通过构造函数的层次结构把级数弄乱的最好方法是把构造函数声明为私有：

```
private GenericCustomer()
{
    name = "<no name>";
}
```

如果试图这样做，就会产生一个有趣的编译错误，如果不理解构造是如何按照层次结构由上而下的顺序工作的，这个错误会让人摸不着头脑。

```
'Wrox.ProCSharp.GenericCustomer()' is inaccessible due to its protection level
```

有趣的是，该错误没有发生在 GenericCustomer 类中，而是发生在 Nevermore60Customer 派生类中。编译器试图为 Nevermore60Customer 生成默认的构造函数，但又做不到，因为默认的构造函数应调用无参数的 GenericCustomer 构造函数。把该构造函数声明为 private，它就不可能访问派生类了。如果为 GenericCustomer 提供一个带有参数的构造函数，但没有提供无参数的构造函数，也会发生类似的错误。在本例中，编译器不能为 GenericCustomer 生成默认构造函数，所以当编译器试图为派生类生成默认构造函数时，会再次发现它不能做到这一点，因



为没有无参数的基类构造函数可调用。这个问题的解决方法是为派生类添加自己的构造函数——实际上不需要在这些构造函数中做任何工作，这样，编译器就不会为这些派生类生成默认构造函数了。

前面介绍了所有的理论知识，下面用一个例子来说明如何给类的层次结构添加构造函数。下一节为 MortimerPhones 样例添加带参数的构造函数。

## 2. 在层次结构中添加带参数的构造函数

首先是带一个参数的 GenericCustomer 构造函数，它仅在顾客提供其姓名时才实例化顾客：

```
abstract class GenericCustomer
{
    private string name;
    public GenericCustomer(string name)
    {
        this.name = name;
    }
}
```

到目前为止，代码运行一切正常，但刚才说过，在编译器试图为派生类创建默认构造函数时，会产生一个编译错误，因为编译器为 Nevermore60Customer 生成的默认构造函数会试图调用无参数的 GenericCustomer 构造函数，但 GenericCustomer 没有这样的构造函数。因此，需要为派生类提供一个构造函数，来避免这个错误：

```
class Nevermore60Customer : GenericCustomer
{
    private uint highCostMinutesUsed;
    public Nevermore60Customer(string name)
        : base(name)
    {
    }
}
```

现在，Nevermore60Customer 对象的实例化只能在提供了包含顾客姓名的字符串后进行，这正是我们需要的。有趣的是 Nevermore60Customer 构造函数对这个字符串所做的处理。它本身不能初始化 name 字段，因为它不能访问基类中的私有字段，但可以把顾客姓名传送给基类，以便 GenericCustomer 构造函数处理。具体方法是，把先执行的基类构造函数指定为把顾客姓名当做参数的构造函数。除此之外，它不需要执行任何操作。

下面讨论如果要处理不同的重载构造函数和一个类的层次结构，会发生什么情况。假定 Nevermore60Customers 通过朋友联系到 MortimerPhones，即 MortimerPhones 公司中有一个人是朋友，因此可以获得折扣。这表示在构造一个 Nevermore60Customer 时，还需要传递联系人的姓名。在现实生活中，构造函数必须利用该姓名去完成更复杂的工作，例如处理折扣等，但这里只是把联系人的姓名存储到另一个字段中。

此时，Nevermore60Customer 定义如下所示：

```
class Nevermore60Customer : GenericCustomer
{
    public Nevermore60Customer(string name, string referrerName)
        : base(name)
    {
        this.referrerName = referrerName;
    }
}
```

```
private string referrerName;
private uint highCostMinutesUsed;
```

该构造函数将姓名作为参数,把它传递给 GenericCustomer 构造函数进行处理。referrerName 是一个变量,我们需要声明它,这样构造函数才能在其主体中处理这个参数。

但是,并不是所有的 Nevermore60Customers 都有联系人,所以还需要有一个不需此参数的构造函数(或为它提供默认值的构造函数)。实际上,我们指定如果没有联系人,referrerName 字段就设置为<None>。下面是这个带一个参数的构造函数:

```
public Nevermore60Customer(string name)
: this(name, "<None>")
{
}
```

这样就正确建立了所有的构造函数。执行下面的代码时,检查事件链是很有益的:

```
GenericCustomer customer = new Nevermore60Customer("Arabel Jones");
```

编译器认为它需要带一个字符串参数的构造函数,所以它确认的构造函数就是刚才定义的那个构造函数,如下所示。

```
public Nevermore60Customer(string Name)
: this(Name, "<None>")
```

在实例化 customer 时,就会调用这个构造函数。之后立即把控制权传送给对应的 Nevermore60Customer 构造函数,该构造函数带 2 个参数,分别是 Arabel Jones 和<None>。在这个构造函数中,把控制权依次传送给 GenericCustomer 构造函数,该构造函数带有 1 个参数,即字符串 Arabel Jones。然后这个构造函数把控制权传送给 System.Object 默认构造函数。现在执行这些构造函数,首先执行 System.Object 构造函数,接着执行 GenericCustomer 构造函数,初始化 name 字段。然后带有两个参数的 Nevermore60Customer 构造函数得到控制权,把联系人的姓名初始化为<None>。最后,执行 Nevermore60Customer 构造函数,该构造函数带有 1 个参数——这个构造函数什么也不做。

这个过程非常简洁,设计也很合理。每个构造函数都处理变量的初始化。在这个过程中,正确地实例化了类,以备使用。如果在为类编写自己的构造函数时遵循这个规则,即便是最复杂的类,也可以顺利地初始化,不会出现任何问题。

## 4.3 修饰符

前面已经遇到许多所谓的修饰符,即应用于类型或成员的关键字。修饰符可以指定方法的可见性,例如 public 或 private,还可以指定一项的本质,例如方法是 virtual 或 abstract。C#有许多访问修饰符,下面讨论完整的修饰符列表。

### 4.3.1 可见性修饰符

表 4-1 中的修饰符确定了是否允许其他代码访问某一项。

表 4-1

修 饰 符	应 用 于	说 明
public	所有的类型或成员	任何代码均可以访问该方法
protected	类型和内嵌类型的所有成员	只有派生的类型能访问该方法
internal	类型和内嵌类型的所有成员	只能在包含它的程序集中访问该方法
private	所有的类型或成员	只能在它所属的类型中访问该方法
protected internal	类型和内嵌类型的所有成员	只能在包含它的程序集和派生类型的代码中访问该方法

注意，类型定义可以是内部或公共的，这取决于是否希望在包含类型的程序集外部访问它：

```
public class MyClass
{
    //etc.
```

不能把类型定义为 `protected`、`private` 和 `protected internal`，因为这些修饰符对于包含在命名空间中的类型来说是没有意义的。因此这些修饰符只能应用于成员。但是，可以用这些修饰符定义嵌套的类型(即包含在其他类型中的类型)，因为在这种情况下，类型也具有成员的状态。下面的代码是合法的：

```
public class OuterClass
{
    protected class InnerClass
    {
        //etc.
    }
    //etc.
}
```

如果有嵌套的类型，内部的类型总是可以访问外部类型的所有成员，所以在上面的代码中，`InnerClass` 中的代码可以访问 `OuterClass` 的所有成员，甚至可以访问 `OuterClass` 的私有成员。

4.3.2 其他修饰符

表 4-2 中的修饰符可以应用于类型的成员，而且有不同的用途。在应用于类型时，其中的几个修饰符也是有意义的。

表 4-2

修 饰 符	应 用 于	说 明
new	函数成员	成员用相同的签名隐藏继承的成员
static	所有的成员	成员不在类的具体实例上执行
virtual	仅类和函数成员	成员可以由派生类重写
abstract	仅函数成员	虚拟成员定义了成员的签名，但没有提供实现代码
override	仅函数成员	成员重写了继承的虚拟或抽象成员
sealed	类，方法和属性	密封类不能继承。对于属性和方法，成员重写了继承的虚拟成员，但继承该类的任何类都不能重写该成员。该修饰符必须与 <code>override</code> 一起使用
extern	仅静态[DllImport]方法	成员在外部用另一种语言实现

在这些修饰符中, `internal` 和 `protected internal` 是 C# 和 .NET Framework 新增的。`internal` 与 `public` 类似, 但访问仅限于同一个程序集中的其他代码, 换言之, 在同一个程序中同时编译的代码。使用 `internal` 可以确保编写的其他类都能访问某一成员, 但同时其他公司编写的其他代码不能访问它们。`protected internal` 合并了 `protected` 和 `internal`, 但这是一种 OR 合并, 而不是 AND 合并。`protected internal` 成员在同一个程序集的任何代码中都可见, 在派生类中也可见, 甚至在其他程序集中也可见。

## 4.4 接口

如前所述, 如果一个类派生于一个接口, 它就会执行某些函数。并不是所有的面向对象语言都支持接口, 所以本节将详细介绍 C# 接口的实现。

注意:

熟悉 COM 的开发人员应注意, 尽管在概念上 C# 接口类似于 COM 接口, 但它们是不同的, 底层的结构不同, 例如, C# 接口并不派生于 `IUnknown`。C# 接口根据 .NET 函数提供了一个契约。与 COM 接口不同, C# 接口不代表任何类型的二进制标准。

下面列出 Microsoft 预定义的一个接口 `System.IDisposable` 的完整定义。`IDisposable` 包含一个方法 `Dispose()`, 该方法由类执行, 用于清理代码:

```
public interface IDisposable
{
    void Dispose();
}
```

上面的代码说明, 声明接口在语法上与声明抽象类完全相同, 但不允许提供接口中任何成员的执行方式。一般情况下, 接口中只能包含方法、属性、索引器和事件的声明。

不能实例化接口, 它只能包含其成员的签名。接口不能有构造函数(如何构建不能实例化的对象?)或字段(因为这隐含了某些内部的执行方式)。接口定义也不允许包含运算符重载, 但这不是因为声明它们在原则上有什么问题, 而是因为接口通常是公共契约, 包含运算符重载会引起一些与其他 .NET 语言不兼容的问题, 例如与 VB 的不兼容, 因为 VB 不支持运算符重载。

在接口定义中还不允许声明成员上的修饰符。接口成员总是公共的, 不能声明为虚拟或静态。如果需要, 就应由执行的类来声明, 因此最好通过执行的类来声明访问修饰符, 就像上面的代码那样。

例如 `IDisposable`。如果类希望声明为公共类型, 以便执行方法 `Dispose()`, 该类就必须执行 `IDisposable`。在 C# 中, 这表示该类派生于 `IDisposable`。

```
class SomeClass : IDisposable
{
    // this class MUST contain an implementation of the
    // IDisposable.Dispose() method, otherwise
    // you get a compilation error
    public void Dispose()
    {
        // implementation of Dispose() method
    }
}
```



```

    }
    // rest of class
}

```

在这个例子中，如果 `SomeClass` 派生于 `IDisposable`，但不包含与 `IDisposable` 中签名相同的 `Dispose()` 实现代码，就会得到一个编译错误，因为该类破坏了实现 `IDisposable` 的契约。当然，编译器允许类有一个不派生于 `IDisposable` 的 `Dispose()` 方法。问题是其他代码无法识别出 `SomeClass` 支持 `IDisposable` 特性。

#### 注意：

`IDisposable` 是一个相当简单的接口，它只定义了一个方法。大多数接口都包含许多成员。

接口的另一个例子是 C# 中的 `foreach` 循环。实际上，`foreach` 循环的内部工作方式是查询对象，看看它是否实现了 `System.Collections.IEnumerable` 接口。如果是，C# 编译器就插入 IL 代码，使用这个接口上的方法迭代集合中的成员，否则，`foreach` 就会引发一个异常。第 10 章将详细介绍 `IEnumerable` 接口。但应注意，`IEnumerable` 和 `IDisposable` 在某种程度上都是有点特殊的接口，因为它们都可以由 C# 编译器识别，在 C# 编译器生成的代码中会考虑它们。显然，自己定义的接口就没有这个特权。

### 4.4.1 定义和实现接口

下面开发一个遵循接口继承规范的小例子来说明如何定义和使用接口。这个例子建立在银行账户的基础上。假定编写代码，最终允许在银行账户之间进行计算机转账业务。许多公司可以实现银行账户，但它们都是彼此赞同表示银行账户的所有类都实现接口 `IBankAccount`。该接口包含一个用于存取款的方法和一个返回余额的属性。这个接口还允许外部代码识别由不同银行账户执行的各种银行账户类。我们的目的是允许银行账户彼此通信，以便在账户之间进行转账业务，但还没有介绍这个功能。

为了使例子简单一些，我们把例子的所有代码都放在同一个源文件中，但实际上不同的银行账户类会编译到不同的程序集中，而这些程序集位于不同银行的不同机器上。但那些内容对于这里的例子来说过于复杂了。为了保留一定的真实性，我们为不同的公司定义不同的命名空间。

首先，需要定义 `IBank` 接口：

```

namespace Wrox.ProCSharp
{
    public interface IBankAccount
    {
        void PayIn(decimal amount);
        bool Withdraw(decimal amount);
        decimal Balance
        {
            get;
        }
    }
}

```

注意，接口的名称为 `IBankAccount`。接口名称传统上以字母 I 开头，以便知道这是一个接口。



注意:

如第2章所述,在大多数情况下,.NET用法规则不鼓励采用所谓的 Hungarian 表示法,在名称的前面加一个字母,表示对象的类型,接口是 Hungarian 表示法推荐采用的几种名称之一。

现在可以编写表示银行账户的类了。这些类不必彼此相关,它们可以是完全不同的类。但它们都表示银行账户,因为它们都实现了 `IBankAccount` 接口。

下面是第一个类,一个由 Royal Bank of Venus 运行的存款账户:

```
namespace Wrox.ProCSharp.VenusBank
{
    public class SaverAccount : IBankAccount
    {
        private decimal balance;
        public void PayIn(decimal amount)
        {
            balance += amount;
        }
        public bool Withdraw(decimal amount)
        {
            if (balance >= amount)
            {
                balance -= amount;
                return true;
            }
            Console.WriteLine("Withdrawal attempt failed.");
            return false;
        }
        public decimal Balance
        {
            get
            {
                return balance;
            }
        }
        public override string ToString()
        {
            return String.Format("Venus Bank Saver: Balance = {0,6:C}", balance);
        }
    }
}
```

这个类的实现代码的作用一目了然。其中包含一个私有字段 `balance`,当存款或取款时就调整这个字段。如果因为账户中的金额不足而取款失败,就会显示一个错误消息。还要注意,因为我们要使代码尽可能简单,所以不实现额外的属性,例如账户持有人的姓名。在现实生活中,这是最基本的信息,但对于本例来说,这是不必要的。

在这段代码中,唯一有趣的是类的声明:

```
public class SaverAccount : IBankAccount
```

`SaverAccount` 派生于一个接口 `IBankAccount`,我们没有明确指出任何其他基类(当然这表示 `SaverAccount` 直接派生于 `System.Object`)。另外,从接口中派生完全独立于从类中派生。

`SaverAccount` 派生于 `IBankAccount`,表示它获得了 `IBankAccount` 的所有成员,但接口并不实际实现其方法,所以 `SaverAccount` 必须提供这些方法的所有实现代码。如果没有提供实现代

码，编译器就会产生错误。接口仅表示其成员的存在性，类负责确定这些成员是虚拟还是抽象的(但只有在类本身是抽象的，这些成员才能是抽象的)。在本例中，接口方法不必是虚拟的。

为了说明不同的类如何实现相同的接口，下面假定 Planetary Bank of Jupiter 还实现一个类 Gold Account 来表示其银行账户：

```
namespace Wrox.ProCSharp.JupiterBank
{
    public class GoldAccount : IBankAccount
    {
        // etc
    }
}
```

这里没有列出 GoldAccount 类的细节，因为在本例中它基本上与 SaverAccount 的实现代码相同。GoldAccount 与 VenusAccount 没有关系，它们只是碰巧实现相同的接口而已。

有了自己的类后，就可以测试它们了。首先需要一些 using 语句：

```
using System;
using Wrox.ProCSharp;
using Wrox.ProCSharp.VenusBank;
using Wrox.ProCSharp.JupiterBank;
```

然后需要一个 Main() 方法：

```
namespace Wrox.ProCSharp
{
    class MainEntryPoint
    {
        static void Main()
        {
            IBankAccount venusAccount = new SaverAccount();
            IBankAccount jupiterAccount = new GoldAccount();
            venusAccount.PayIn(200);
            venusAccount.Withdraw(100);
            Console.WriteLine(venusAccount.ToString());
            jupiterAccount.PayIn(500);
            jupiterAccount.Withdraw(600);
            jupiterAccount.Withdraw(100);
            Console.WriteLine(jupiterAccount.ToString());
        }
    }
}
```

这段代码(如果下载本例子，它在 BankAccounts.cs 文件中)的执行结果如下：

```
C:>BankAccounts
Venus Bank Saver: Balance = £100.00
Withdrawal attempt failed.
Jupiter Bank Saver: Balance = £400.00
```

在这段代码中，一个要点是把引用变量声明为 IBankAccount 引用的方式。这表示它们可以指向实现这个接口的任何类的实例。但我们只能通过这些引用调用接口的方法——如果要调用由类执行的、不在接口中的方法，就需要把引用强制转换为合适的类型。在这段代码中，我们调用了 ToString() (不由 IBankAccount 实现)，但没有进行任何显式转换，这只是因为 ToString() 是一个 System.Object 方法，C# 编译器知道任何类都支持这个方法(换言之，从接口到

System.Object 的数据类型转换是隐式的)。第6章将介绍强制转换的语法。

接口引用完全可以看做是类引用——但接口引用的强大之处在于，它可以引用任何实现该接口的类。例如，我们可以构造接口数组，其中的每个元素都是不同的类：

```
IBankAccount[] accounts = new IBankAccount[2];
accounts[0] = new SaverAccount();
accounts[1] = new GoldAccount();
```

但注意，如果编写了如下代码，就会生成一个编译错误：

```
accounts[1] = new SomeOtherClass(); // SomeOtherClass does NOT implement
// IBankAccount: WRONG!!
```

这会导致一个如下所示的编译错误：

```
Cannot implicitly convert type 'Wrox.ProCSharp.SomeOtherClass' to
'Wrox.ProCSharp.IBankAccount'
```

#### 4.4.2 派生的接口

接口可以彼此继承，其方式与类的继承相同。下面通过定义一个新接口 `ITransferBankAccount` 来说明这个概念，该接口的功能与 `IBankAccount` 相同，只是又定义了一个方法，把资金直接转到另一个账户上。

```
namespace Wrox.ProCSharp
{
    public interface ITransferBankAccount : IBankAccount
    {
        bool TransferTo(IBankAccount destination, decimal amount);
    }
}
```

因为 `ITransferBankAccount` 派生于 `IBankAccount`，所以拥有 `IBankAccount` 的所有成员和它自己的成员。这表示执行(派生于)`ITransferBankAccount` 的任何类都必须执行 `IBankAccount` 的所有方法和在 `ITransferBankAccount` 中定义的新方法 `TransferTo()`。没有执行所有这些方法就会产生一个编译错误。

注意，`TransferTo()` 方法为目标账户使用了 `IBankAccount` 接口引用。这说明了接口的用途：在执行并调用这个方法时，不必知道转帐的对象类型，只需知道该对象执行 `IBankAccount` 即可。

下面演示 `ITransferBankAccount`：假定 Planetary Bank of Jupiter 还提供了一个当前账户。`CurrentAccount` 类的大多数执行代码与 `SaverAccount` 和 `GoldAccount` 的执行代码相同(这仅是为了使例子更简单，一般是不会这样的)，所以在下面的代码中，我们仅突出显示了不同的地方：

```
public class CurrentAccount : ITransferBankAccount
{
    private decimal balance;
    public void PayIn(decimal amount)
    {
        balance += amount;
    }
    public bool Withdraw(decimal amount)
    {
        if (balance >= amount)
```

```

        {
            balance -= amount;
            return true;
        }
        Console.WriteLine("Withdrawal attempt failed.");
        return false;
    }
    public decimal Balance
    {
        get
        {
            return balance;
        }
    }
    public bool TransferTo(IBankAccount destination, decimal amount)
    {
        bool result;
        if ((result = Withdraw(amount)) == true)
            destination.PayIn(amount);
        return result;
    }
    public override string ToString()
    {
        return String.Format("Jupiter Bank Current Account: Balance = {0,6:C}",
                               balance);
    }
}

```

可以用下面的代码验证该类：

```

static void Main()
{
    IBankAccount venusAccount = new SaverAccount();
    ITransferBankAccount jupiterAccount = new CurrentAccount();
    venusAccount.PayIn(200);
    jupiterAccount.PayIn(500);
    jupiterAccount.TransferTo(venusAccount, 100);
    Console.WriteLine(venusAccount.ToString());
    Console.WriteLine(jupiterAccount.ToString());
}

```

这段代码(CurrentAccount.cs)的结果如下所示，其中显示转账后正确的资金数：

```

C:>CurrentAccount
Venus Bank Saver: Balance = £300.00
Jupiter Bank Current Account: Balance = £400.00

```

## 4.5 小结

本章介绍了如何在 C# 中进行继承。C# 支持多接口继承和单一实现继承，还提供了许多有效的语法结构，以使代码更健壮，例如 `override` 关键字，它表示函数应在何时重写基类函数，`new` 关键字表示函数在何时隐藏基类函数，构造函数初始化器的硬性规则可以确保构造函数以健壮的方式进行交互操作。

# 第 5 章

## 数 组

如果需要使用同一类型的多个对象，就可以使用集合和数组。C#用特殊的记号声明和使用数组。Array 类在后台发挥作用，为数组中元素的排序和过滤提供了几个方法。

使用枚举器，可以迭代数组中的所有元素。

本章讨论如下内容：

- 简单数组
- 多维数组
- 锯齿数组
- Array 类
- 数组的接口
- 枚举

### 5.1 简单数组

如果需要使用同一类型的多个对象，就可以使用数组。数组是一种数据结构，可以包含同一类型的多个元素。

#### 5.1.1 数组的声明

在声明数组时，应先定义数组中元素的类型，其后是一个空方括号和一个变量名。例如，下面声明了一个包含整型元素的数组：

```
int[] myArray;
```

#### 5.1.2 数组的初始化

声明了数组后，就必须为数组分配内存，以保存数组的所有元素。数组是引用类型，所以必须给它分配堆上的内存。为此，应使用 new 运算符，指定数组中元素的类型和数量来初始化数组的变量。下面指定了数组的大小。

**提示：**

值类型和引用类型请参见第 3 章。

```
myArray = new int[4];
```



在声明和初始化后，变量 `myArray` 就引用了 4 个整型值，它们位于托管堆上，如图 5-1 所示。

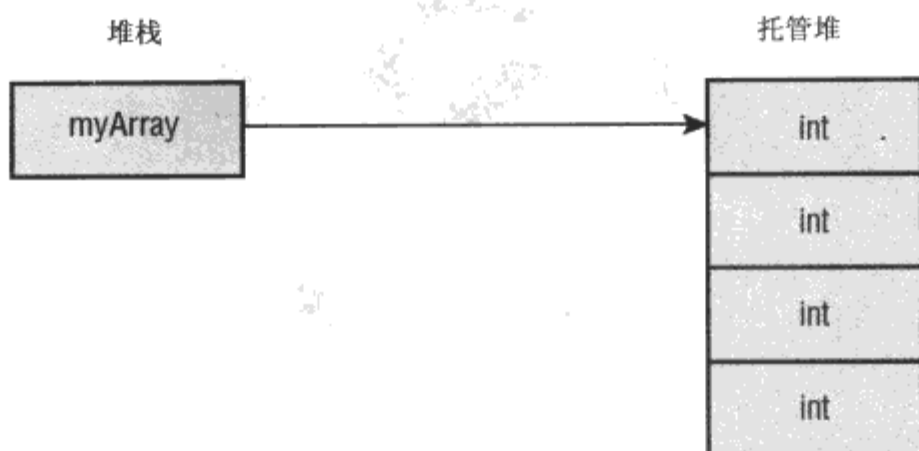


图 5-1

#### 警告：

在指定了数组的大小后，如果不复制数组中的所有元素，就不能重新设置数组的大小。如果事先不知道数组中应包含多少个元素，就可以使用集合。集合请参见第 10 章。

除了两个语句中声明和初始化数组之外，还可以在一个语句中声明和初始化数组：

```
int[] myArray = new int[4];
```

还可以使用数组初始化器为数组的每个元素赋值。数组初始化器只能在声明数组变量时使用，不能在声明数组之后使用。

```
int[] myArray = new int[4] {4, 7, 11, 2};
```

如果用花括号初始化数组，还可以不指定数组的大小，因为编译器会计算出元素的个数：

```
int[] myArray = new int[] {4, 7, 11, 2};
```

使用 C# 编译器还有一种更简化的形式。使用花括号可以同时声明和初始化数组，编译器生成的代码与前面的例子相同：

```
int[] myArray = {4, 7, 11, 2};
```

### 5.1.3 访问数组元素

数组在声明和初始化后，就可以使用索引器访问其中的元素了。数组只支持有整型参数的索引器。

#### 提示：

在定制类中，可以创建支持其他类型的索引器。创建定制索引器的内容请参见第 6 章。

通过索引器传送元素号，就可以访问数组。索引器总是以 0 开头，表示第一个元素。可以传送给索引器的最大值是元素个数减 1，因为索引从 0 开始。在下面的例子中，数组 `myArray` 用 4 个整型值声明和初始化。用索引器 0、1、2、3 就可以访问该数组中的元素。

```
int[] myArray = new int[] {4, 7, 11, 2};
int v1 = myArray[0]; // read first element
```

```
int v2 = myArray[1];    // read second element
myArray[3] = 44;        // change fourth element
```

**警告:**

如果使用错误的索引器值 (不存在对应的元素), 就会抛出 `IndexOutOfRangeException` 类型的异常。

如果不知道数组中的元素个数, 则可以在 `for` 语句中使用 `Length` 属性:

```
for (int i = 0; i < myArray.Length; i++)
{
    Console.WriteLine(myArray[i]);
}
```

除了使用 `for` 语句迭代数组中的所有元素之外, 还可以使用 `foreach` 语句:

```
for (int val in myArray)
{
    Console.WriteLine(val);
}
```

**提示:**

`foreach` 语句利用了本章后面讨论的 `IEnumerable` 和 `IEnumerator` 接口。

#### 5.1.4 使用引用类型

不但能声明预定义类型的数组, 还可以声明定制类型的数组。下面用 `Person` 类来说明, 这个类有两个构造函数、自动实现的属性 `Firstname` 和 `Lastname`、以及 `ToString()` 方法的一个重写:

```
public class Person
{
    public Person()
    {
    }

    public Person(string firstName, string lastName)
    {
        this.firstname = firstName;
        this.lastname = lastName;
    }

    public string Firstname { get; set; }
    public string Lastname { get; set; }

    public override string ToString()
    {
        return String.Format("{0} {1}", firstName, lastName);
    }
}
```

声明一个包含两个 `Person` 元素的数组, 与声明一个 `int` 数组类似:

```
Person[] myPersons = new Person[2];
```

但是必须注意，如果数组中的元素是引用类型，就必须为每个数组元素分配内存。若使用了数组中未分配内存的元素，就会抛出 `NullReferenceException` 类型的异常。

提示：  
第 14 章介绍了错误和异常的详细内容。

使用从 0 开始的索引器，可以为数组的每个元素分配内存：

```
myPersons [0] = new Person("Ayrton", "Senna");  
myPersons [1] = new Person("Michael", "Schumacher");
```

图 5-2 显示了 `Person` 数组中的对象在托管堆中的情况。`myPersons` 是一个存储在堆栈上的变量，该变量引用了存储在托管堆上的 `Person` 元素数组。这个数组有足够容纳两个引用的空间。数组中的每一项都引用了一个 `Person` 对象，而这些 `Person` 对象也存储在托管堆上。

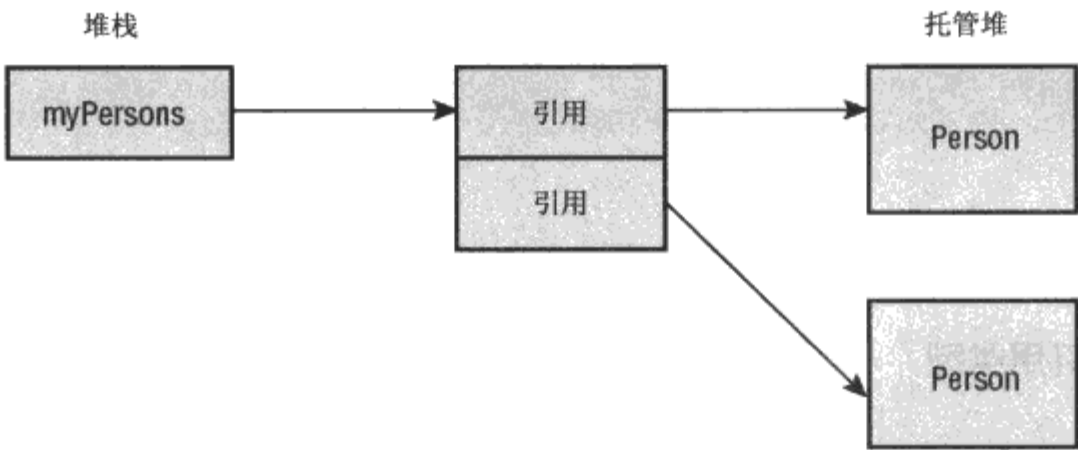


图 5-2

与 `int` 类型一样，也可以对定制类型使用数组初始化器：

```
Person[] myPersons = {new Person("Ayrton", "Senna"),  
                       new Person("Michael", "Schumacher") };
```

## 5.2 多维数组

一般数组（也称为一维数组）用一个整数来索引。多维数组用两个或多个整数来索引。

图 5-3 是二维数组的数学记号，该数组有三行三列。第一行的值是 1、2 和 3，第三行的值是 7、8 和 9。

$$A = \begin{bmatrix} 1, 2, 3 \\ 4, 5, 6 \\ 7, 8, 9 \end{bmatrix}$$

图 5-3

在 C# 中声明这个二维数组，需要在括号中加上一个逗号。数组在初始化时应指定每一维的大小（也称为阶）。接着，就可以使用两个整数作为索引器，来访问数组中的元素了：

```
int[,] twodim = new int[3, 3];  
twodim[0,0] = 1;
```

```
twodim[0,1] = 2;
twodim[0,2] = 3;
twodim[1,0] = 4;
twodim[1,1] = 5;
twodim[1,2] = 6;
twodim[2,0] = 7;
twodim[2,1] = 8;
twodim[2,2] = 9;
```

**提示：**  
数组声明之后，就不能修改其阶数了。

如果事先知道元素的值，也可以使用数组索引器来初始化二维数组。在初始化数组时，使用一个外层的花括号，每一行用包含在外层花括号中的内层花括号来初始化。

```
int[,] twodim = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9},
};
```

**提示：**  
使用数组初始化器时，必须初始化数组的每个元素，不能遗漏任何元素。

在花括号中使用两个逗号，就可以声明一个三维数组：

```
int[, ,] threedim = {
    { {1, 2}, {3, 4} },
    { {5, 6}, {7, 8} },
    { {9, 10}, {11, 12} },
};

Console.WriteLine(threedim[0,1,1]);
```

5.3 锯齿数组

二维数组的大小是矩形的，例如 3×3 个元素。而锯齿数组的大小设置是比较灵活的，在锯齿数组中，每一行都可以有不同的尺寸。

图 5-4 比较了有 3×3 个元素的二维数组和锯齿数组。图中的锯齿数组有 3 行，第一行有 2 个元素，第二行有 6 个元素，第三行有 3 个元素。

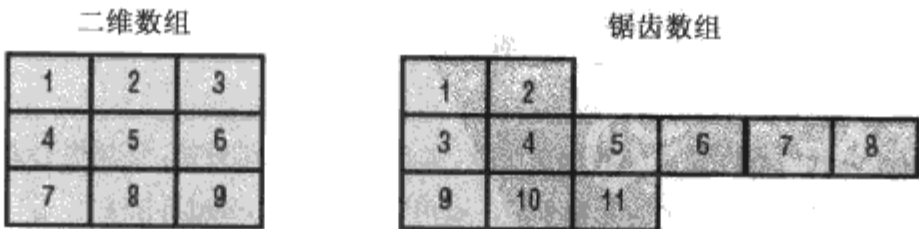


图 5-4

在声明锯齿数组时，要依次放置开闭括号。在初始化锯齿数组时，先设置该数组包含的行数。定义各行中元素个数的第二个括号设置为空，因为这类数组的每一行包含不同的元素数。之后，为每一行指定行中的元素个数：

```
int[][] jagged = new int[3][];
jagged[0] = new int[2] {1, 2};
jagged[1] = new int[6] {3, 4, 5, 6, 7, 8};
jagged[2] = new int[3] {9, 10, 11};
```

迭代锯齿数组中所有元素的代码可以放在嵌套的 for 循环中。在外层的 for 循环中，迭代每一行，内层的 for 循环迭代一行中的每个元素：

```
for ( int row = 0; row < jagged.Length; row++)
{
    for ( int element = 0; element < jagged[row].Length; element++)
    {
        Console.WriteLine("row: {0}, element: {1}, value: {2}",
            row, element, <jagged[row].[element]);
    }
}
```

该迭代显示了所有的行和每一行中的各个元素：

```
row: 0, element: 0, value: 1
row: 0, element: 1, value: 2
row: 1, element: 0, value: 3
row: 1, element: 1, value: 4
row: 1, element: 2, value: 5
row: 1, element: 3, value: 6
row: 1, element: 4, value: 7
row: 1, element: 5, value: 8
row: 2, element: 1, value: 9
row: 2, element: 2, value: 10
row: 2, element: 3, value: 11
```

## 5.4 Array 类

用括号声明数组是 C#中使用 Array 类的记号。在后台使用 C#语法，会创建一个派生于抽象基类 Array 的新类。这样，就可以使用 Array 类为每个 C#数组定义的方法和属性了。例如，前面就使用了 Length 属性，还使用 foreach 语句迭代数组。其实这是使用了 Array 类中的 GetEnumerator()方法。

### 5.4.1 属性

Array 类包含的如下属性可以用于每个数组实例。本章后面还将讨论其他更多的属性。

表 5-1

属 性	说 明
Length	Length 属性返回数组中的元素个数。如果是一个多维数组，该属性会返回所有阶的元素个数。如果需要确定一维中的元素个数，则可以使用 GetLength()方法
LongLength	Length 属性返回 int 值，而 LongLength 属性返回 long 值。如果数组包含的元素个数超出了 32 位 int 值的取值范围，就需要使用 LongLength 属性，来获得元素个数
Rank	使用 Rank 属性可以获得数组的维数



### 5.4.2 创建数组

Array 类是一个抽象类，所以不能使用构造函数来创建数组。但除了可以使用 C# 语法创建数组实例之外，还可以使用静态方法 `CreateInstance()` 创建数组。如果事先不知道元素的类型，就可以使用该静态方法，因为类型可以作为 `Type` 对象传送给 `CreateInstance()` 方法。

下面的例子说明了如何创建类型为 `int`、大小为 5 的数组。`CreateInstance()` 方法的第一个参数应是元素的类型，第二个参数定义数组的大小。可以用 `SetValue()` 方法设置值，用 `GetValue()` 方法读取值：

```
Array intArray1 = Array.CreateInstance(typeof(int), 5);
for (int i = 0; i < 5; i++)
{
    intArray1.SetValue(33, i);
}

for (int i = 0; i < 5; i++)
{
    Console.WriteLine(intArray1.GetValue(i));
}
```

还可以将已创建的数组强制转换成声明为 `int[]` 的数组：

```
int[] intArray2 = (int[])intArray1;
```

`CreateInstance()` 方法有许多重载版本，可以创建多维数组和不基于 0 的数组。下面的例子就创建了一个包含  $2 \times 3$  个元素的二维数组。第一维基于 1，第二维基于 0：

```
int[] lengths = {2, 3};
int[] lowerBounds = {1, 10};
Array racers = Array.CreateInstance(typeof(Person), lengths, lowerBounds);
```

`SetValue()` 方法设置数组的元素，其参数是每一维的索引：

```
racers.SetValue(new Person("Alain", "Prost"), 1, 10);
racers.SetValue(new Person("Emerson", "Fittipaldi"), 1, 11);
racers.SetValue(new Person("Ayrton", "Senna"), 1, 12);
racers.SetValue(new Person("Ralf", "Schumacher"), 2, 10);
racers.SetValue(new Person("Fernando", "Alonso"), 2, 11);
racers.SetValue(new Person("Jenson", "Button"), 2, 12);
```

尽管数组不是基于 0 的，但可以用一般的 C# 记号将它赋予一个变量。只需注意不要超出边界即可：

```
Person[,] racers2 = (Person[,]) racers;
Person first = racers2[1, 10];
Person last = racers2[2, 12];
```

### 5.4.3 复制数组

因为数组是引用类型，所以将一个数组变量赋予另一个数组变量，就会得到两个指向同一数组的变量。而复制数组，会使数组实现 `ICloneable` 接口。这个接口定义的 `Clone()` 方法会创建数组的浅副本。

如果数组的元素是值类型，就会复制所有的值，如图 5-5 所示：

```
int intArray1 = {1, 2};
int intArray2 = (int[])intArray1.Clone();
```

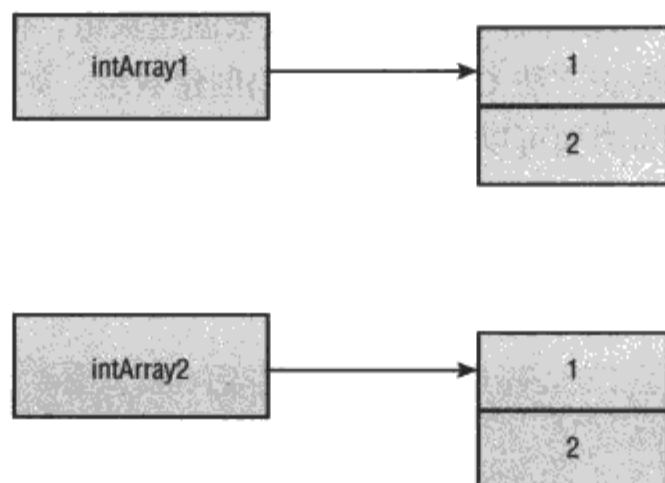


图 5-5

如果数组包含引用类型，则不复制元素，而只复制引用。图 5-6 显示了变量 `beatles` 和 `beatlesClone`，其中 `beatlesClone` 是通过在 `beatles` 上调用 `Clone()` 方法来创建的。`beatles` 和 `beatlesClone` 引用的 `Person` 对象是相同的。如果修改 `beatlesClone` 中一个元素的属性，就会改变 `beatles` 中的对应对象。

```
Person[] beatles = {
    new Person("John", "Lennon"),
    new Person("Paul", "McCartney"),
};
Person[] beatlesClone = (Person[])beatles.Clone();
```

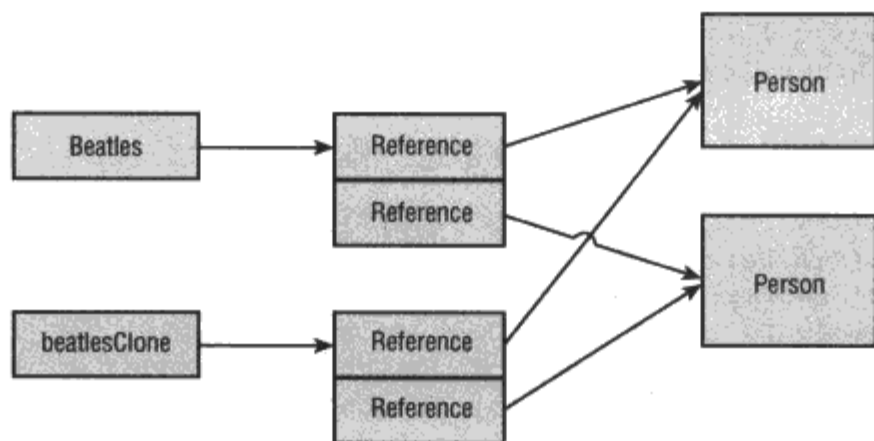


图 5-6

除了使用 `Clone()` 方法之外，还可以使用 `Array.Copy()` 方法创建浅副本。但 `Clone()` 方法和 `Copy()` 方法有一个重要区别：`Clone()` 方法会创建一个新数组，而 `Copy()` 方法只是传送了阶数相同、有足够元素空间的已有数组。

**提示：**

如果需要包含引用类型的数组的深副本，就必须迭代数组，创建新对象。

#### 5.4.4 排序

`Array` 类实现了对数组中元素的冒泡排序。`Sort()` 方法需要数组中的元素实现 `IComparable`

接口。简单类型，如 `System.String` 和 `System.Int32` 实现了 `IComparable` 接口，所以可以对包含这些类型的元素排序。

在示例程序中，数组 `name` 包含 `string` 类型的元素，这个数组是可以排序的。

```
String[] names = {
    "Christina Aguilera",
    "Shakira",
    "Beyonce",
    "Gwen Stefani"
};

Array.Sort(names);

foreach (string name in names)
{
    Console.WriteLine(name);
}
```

该应用程序的输出是排好序的数组：

```
Beyonce
Christina Aguilera
Gwen Stefani
Shakira
```

如果对数组使用定制类，就必须实现 `IComparable` 接口。这个接口只定义了一个方法 `CompareTo()`，如果要比较的对象相等，该方法就返回 0。如果实例应排在参数对象的前面，该方法就返回小于 0 的值。如果实例应排在参数对象的后面，该方法就返回大于 0 的值。

修改 `Person` 类，使之执行 `IComparable` 接口。对 `LastName` 的值进行比较。`LastName` 是 `string` 类型，而 `String` 类已经实现了 `IComparable` 接口，所以可以使用 `String` 类中 `CompareTo()` 方法的实现代码。如果 `LastName` 的值相同，就比较 `FirstName`：

```
public class Person : IComparable
{
    public int CompareTo(object obj)
    {
        Person other = obj as Person;
        int result = this.LastName.CompareTo(
            other.LastName);
        if (result == 0)
        {
            result = this.FirstName.CompareTo(
                other.FirstName);
        }
        return result;
    }
    //...
```

现在可以按照姓氏对 `Person` 对象数组排序了：

```
Person[] persons = {
    new Person("Emerson", "Fittipaldi"),
    new Person("Niki", "Lauda"),
    new Person("Ayrton", "Senna"),
    new Person("Michael", "Schumacher"),
};
```

```

Array.Sort (persons);
foreach (Person p in persons)
{
    Console.WriteLine(p);
}

```

使用 `Person` 类的排序功能，会得到按姓氏排序的姓名：

```

Emerson Fittipaldi
Niki Lauda
Michael Schumacher
Ayrton Senna

```

如果 `Person` 对象的排序方式与上述不同，或者不能修改在数组中用作元素的类，就可以执行 `IComparer` 接口。这个接口定义了方法 `Compare()`。`IComparable` 接口必须由要比较的类来执行，而 `IComparer` 接口独立于要比较的类。这就是 `Compare()` 方法定义了两个要比较的变元的原因。其返回值与 `IComparable` 接口的 `CompareTo()` 方法类似。

类 `PersonComparer` 实现了 `IComparer` 接口，可以按照 `firstName` 或 `lastName` 对 `Person` 对象排序。枚举 `PersonCompareType` 定义了与 `PersonComparer` 相当的排序选项：`FirstName` 和 `LastName`。排序的方式由类 `PersonComparer` 的构造函数定义，在该构造函数中设置了一个 `PersonCompareType` 值。`Compare()` 方法用一个 `switch` 语句指定是按 `firstName` 还是 `lastName` 排序。

```

public class PersonComparer : IComparer
{
    public enum PersonCompareType
    {
        FirstName,
        LastName
    }

    private PersonCompareType compareType;

    public PersonComparer(PersonCompareType compareType)
    {
        this.compareType = compareType;
    }

    public int Compare(object x, object y)
    {
        Person p1 = x as Person;
        Person p2 = y as Person;
        Switch (compareType)
        {
            case PersonCompareType.FirstName:
                return p1.FirstName.CompareTo(p2.FirstName);
            case PersonCompareType.LastName:
                return p1.LastName.CompareTo(p2.LastName);
            default:
                throw new ArgumentException("unexpexted compare type");
        }
    }
}

```

现在，可以将一个 `PersonComparer` 对象传送给 `Array.Sort()` 方法的第二个变元。下面是按名字对 `persons` 数组排序：

```

Array.Sort(persons,

```

```
new PersonComparer(PersonComparer. PersonCompareType.FirstName));
foreach (Person p in persons)
{
    Console.WriteLine(p);
}
```

persons 数组现在按名字排序:

Ayrton Senna  
Emerson Fittipaldi  
Michael Schumacher  
Niki Lauda

提示:  
Array 类还提供了 Sort 方法, 它需要将一个委托作为变元。第 7 章将介绍如何使用委托。

5.5 数组和集合接口

Array 类实现了 IEumerable、ICollection 和 IList 接口, 以访问和枚举数组中的元素。由于用定制数组创建的类派生于 Array 抽象类, 所以能使用通过数组变量执行的接口中的方法和属性。

5.5.1 IEumerable 接口

IEumerable 是由 foreach 语句用于迭代数组的接口。这是一个非常特殊的特性, 在下一节中讨论。

5.5.2 ICollection 接口

ICollection 接口派生于 IEumerable 接口, 并添加了如表 5-2 所示的属性和方法。这个接口主要用于确定集合中的元素个数, 或用于同步。

表 5-2

ICollection 接口的属性和方法	说 明
Count	Count 属性可确定集合中的元素个数, 它返回的值与 Length 属性相同
IsSynchronized SyncRoot	IsSynchronized 属性确定集合是否是线程安全的。对于数组, 这个属性总是返回 false。对于同步访问, SyncRoot 属性可以用于线程安全的访问。第 19 章介绍了线程和同步, 探讨了如何用集合实现线程安全性
CopyTo()	利用 CopyTo()方法可以将数组的元素复制到现有的数组中。它类似于静态方法 Array.Copy()

5.5.3 IList 接口

IList 接口派生于 ICollection 接口, 并添加了下面的属性和方法。Array 类实现 IList 接口的主要原因是, IList 接口定义了 Item 属性, 以使用索引器访问元素。IList 接口的许多其他成员是通过 Array 类抛出 NotSupportedException 异常实现的, 因为这些不应用于数组。IList 接口的所有属性和方法如表 5-3 所示。



表 5-3

ICollection 接口	说 明
Add()	Add()方法用于在集合中添加元素。对于数组，该方法会抛出 NotSupportedException 异常
Clear()	Clear()方法可清除数组中的所有元素。值类型设置为 0，引用类型设置为 null
Contains()	Contains()方法可以确定某个元素是否在数组中。其返回值是 true 或 false。这个方法会对数组中的所有元素进行线性搜索，直到找到所需元素为止
IndexOf()	IndexOf()方法与 Contains()方法类似，也是对数组中的所有元素进行线性搜索。不同的是，IndexOf()方法会返回所找到的第一个元素的索引
Insert() Remove() RemoveAt()	对于集合，Insert()方法用于插入元素，Remove()和 RemoveAt()可删除元素。对于数组，这些方法都抛出 NotSupportedException 异常
IsFixedSize	数组的大小总是固定的，所以这个属性总是返回 true
IsReadOnly	数组总是可以读/写的，所以这个属性返回 false。第 10 章将介绍如何从数组中创建只读属性
Item	Item 属性可以用整型索引访问数组

5.6 枚举

在 foreach 语句中使用枚举，可以迭代集合中的元素，且无需知道集合中的元素个数。图 5-7 显示了调用 foreach 方法的客户机和集合之间的关系。数组或集合执行带 GetEnumerator()方法的 IEnumerable 接口。GetEnumerator()方法返回一个执行 IEnumerator 接口的枚举。接着，foreach 语句就可以使用 IEnumerator 接口迭代集合了。

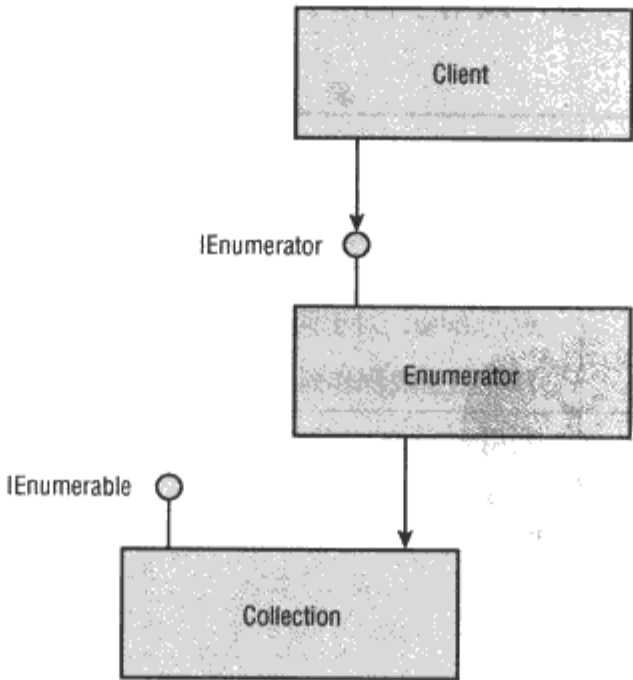


图 5-7

提示：

GetEnumerator()方法用 IEnumerable 接口定义。foreach 语句并不真的需要在集合类中执行这个接口。有一个名为 GetEnumerator()的方法，返回实现了 IEnumerator 接口的对象就足够了。

5.6.1 IEnumerator 接口

foreach 语句使用 IEnumerator 接口的方法和属性，迭代集合中的所有元素。这个接口中的属性和方法如表 5-4 所示。

表 5-4

IEnumerator 接口的方法和属性	说 明
MoveNext()	MoveNext()方法移动到集合的下一个元素上，如果有这个元素，该方法就返回 true。如果集合不再有更多的元素，该方法就返回 false
Current	属性 Current 返回光标所在的元素
Reset()	Reset() 方法将光标重新定位于集合的开头。许多枚举会抛出 NotSupportedException 异常

5.6.2 foreach 语句

C#的 foreach 语句不会解析为 IL 代码中的 foreach 语句。C#编译器会把 foreach 语句转换为 IEnumerable 接口的方法和属性。下面是一个简单的 foreach 语句，它迭代 persons 数组中的所有元素，并逐个显示它们：

```
foreach (Person p in persons)
{
    Console.WriteLine(p);
}
```

foreach 语句会解析为下面的代码段。首先，调用 GetEnumerator()方法，获得数组的一个枚举。在 while 循环中——只要 MoveNext()返回 true——用 Current 属性访问数组中的元素：

```
IEnumerator enumerator = persons.GetEnumerator();
while (enumerator.MoveNext())
{
    Person p = (Person) enumerator.Current;
    Console.WriteLine(p);
}
```

5.6.3 yield 语句

C# 1.0 使用 foreach 语句可以轻松地迭代集合。在 C# 1.0 中，创建枚举器仍需要做大量的工作。C# 2.0 添加了 yield 语句，以便于创建枚举器。

yield return 语句返回集合的一个元素，并移动到下一个元素上。yield break 可停止迭代。

下面的例子是用 yield return 语句实现一个简单集合的代码。类 HelloCollection 包含 GetEnumerator()方法。该方法的实现代码包含两个 yield return 语句，它们分别返回字符串 Hello 和 World。

```
using System;
using System.Collections;
```

```
namespace Wrox.ProCSharp.Arrays
{
    public class HelloCollection
    {
        public IEnumerator GetEnumerator()
        {
            yield return "Hello";
            yield return "World";
        }
    }
}
```

**警告:**

包含 `yield` 语句的方法或属性也称为迭代块。迭代块必须声明为返回 `IEnumerator` 或 `IEnumerable` 接口。这个块可以包含多个 `yield return` 语句或 `yield break` 语句，但不能包含 `return` 语句。

现在可以用 `foreach` 语句迭代集合了:

```
public class Program
{
    HelloCollection helloCollection = new HelloCollection();
    foreach (string s in helloCollection)
    {
        Console.WriteLine(s);
    }
}
```

使用迭代块，编译器会生成一个 `yield` 类型，其中包含一个状态机，如下面的代码所示。`yield` 类型执行 `IEnumerator` 和 `IDisposable` 接口的属性和方法。在下面的例子中，可以把 `yield` 类型看作内部类 `Enumerator`。外部类的 `GetEnumerator()` 方法实例化并返回一个新的 `yield` 类型。在 `yield` 类型中，变量 `state` 定义了迭代的当前位置，每次调用 `MoveNext()` 时，当前位置都会改变。`MoveNext()` 封装了迭代块的代码，设置了 `current` 变量的值，使 `Current` 属性根据位置返回一个对象。

```
public class HelloCollection
{
    public IEnumerator GetEnumerator()
    {
        Enumerator enumerator = new Enumerator();
        return enumerator;
    }

    public class Enumerator : IEnumerator, IDisposable
    {
        private int state;
        private object current;

        public Enumerator(int state)
        {
            this.state = state;
        }

        bool System.Collections.IEnumerator.MoveNext()
        {

```

```

switch (state)
{
    case 0:
        current = "Hello";
        state = 1;
        return true;
    case 1:
        current = "World";
        state = 2;
        return true;
    case 2:
        break;
}

return false;
}

void System.Collections.IEnumerator.Reset()
{
    throw new NotSupportedException();
}

object System.Collections.IEnumerator.Current
{
    get
    {
        return current;
    }
}

void IDisposable.Dispose()
{
}
}

```

现在使用 `yield return` 语句，很容易实现允许以不同方式迭代集合的类。类 `MusicTitles` 可以用默认方式通过 `GetEnumerator()` 方法迭代标题，用 `Reverse()` 方法逆序迭代标题，用 `Subset()` 方法搜索子集：

```

public class MusicTitles
{
    string[] names = {
        "Tubular Bells", "Hergest Ridge",
        "Ommadawn", "Platinum");

    public IEnumerator GetEnumerator()
    {
        for (int i = 0; i < 4; i++)
        {
            yield return names[i];
        }
    }

    public IEnumerable Reverse()
    {
        for (int i = 3; i >= 0; i--)
        {
            yield return names[i];
        }
    }

    public IEnumerable Subset( int index, int length)
    {
        for (int i = index; i < index + length; i++)

```



```

        {
            yield return names[i];
        }
    }
}

```

迭代字符串数组的客户代码先使用 `GetEnumerator()` 方法，该方法不必在代码中编写，因为这是默认使用的方法。然后逆序迭代标题，最后将索引和要迭代的元素个数传送给 `Subset()` 方法，来迭代子集：

```

MusicTitles titles = new MusicTitles();
foreach(string title in titles)
{
    Console.WriteLine(title);
}
Console.WriteLine();

Console.WriteLine("reverse");
foreach(string title in titles.Reverse())
{
    Console.WriteLine(title);
}
Console.WriteLine();

Console.WriteLine("subset");
foreach(string title in titles.Subset(2, 2))
{
    Console.WriteLine(title);
}

```

使用 `yield` 语句还可以完成更复杂的任务，例如从 `yield return` 中返回枚举器。

在 TicTacToe 游戏中有 9 个域，玩家轮流在这些域中放置“十”字或一个圆。这些移动操作由 `GameMoves` 类模拟。方法 `Cross()` 和 `Circle()` 是创建迭代类型的迭代块。变量 `cross` 和 `circle` 在 `GameMoves` 类的构造函数中设置为 `Cross()` 和 `Circle()` 方法。这些域不设置为调用的方法，而是设置为用迭代块定义的迭代类型。在 `Cross()` 迭代块中，将移动操作的信息写到控制台上，并递增移动次数。如果移动次数大于 9，就用 `yield break` 停止迭代；否则，就在每次迭代中返回 `yield` 类型 `cross` 的枚举对象。`Circle()` 迭代块非常类似于 `Cross()` 迭代块，只是它在每次迭代中返回 `circle` 迭代类型。

```

public class GameMoves
{
    private IEnumerator cross;
    private IEnumerator circle;

    public GameMoves()
    {
        cross = Cross();
        circle = Circle();
    }

    private int move = 0;

    public IEnumerator Cross()
    {
        while (true)
        {
            Console.WriteLine("Cross, move {0}", move);
            move++;
            if (move > 9)

```



```

        yield break;
        yield return circle;
    }
}

public IEnumerator Circle()
{
    while (true)
    {
        Console.WriteLine("Circle, move {0}", move);
        move++;
        if (move > 9)
            yield break;
        yield return cross;
    }
}
}

```

在客户程序中，可以以如下方式使用 GameMoves 类。将枚举器设置为由 game.Cross() 返回的枚举器类型，以设置第一次移动。enumerator.MoveNext 调用以迭代块定义的一次迭代，返回另一个枚举器。返回的值可以用 Current 属性访问，并设置为 enumerator 变量，用于下一次循环：

```

GameMoves game = new GameMoves();
IEnumerator enumerator = game.Cross();
while (enumerator.MoveNext())
{
    enumerator = (IEnumerator) enumerator.Current;
}

```

这个程序的输出会显示交替移动的情况，直到最后一次移动：

```

Cross, move 0
Circle, move 1
Cross, move 2
Circle, move 3
Cross, move 4
Circle, move 5
Cross, move 6
Circle, move 7
Cross, move 8

```

## 5.7 小结

本章介绍了创建和使用简单数组、多维数组和锯齿数组的 C# 记号。Array 类在 C# 数组的后台使用，这样就可以用数组变量调用这个类的属性和方法。

我们还探讨了如何使用 IComparable 和 IComparer 接口给数组中的元素排序，描述了用 Array 类执行的 IEnumerable、ICollection 和 IList 接口的特性，最后论述了 yield 语句的优点。

第 6 章介绍了运算符和强制类型转换，其中探讨了定制索引器的创建。第 7 章讨论了委托和事件。Array 类的一些方法将委托用作参数。第 10 章介绍了本章探讨的集合类。集合类有较灵活的尺寸，第 10 章还介绍了其他容器，如字典和链表。

# 第 6 章

## 运算符和类型强制转换

前几章介绍了使用 C#编写程序所需要的大部分知识。本章将首先讨论基本语言元素，接着论述 C#语言的扩展功能。本章的主要内容如下：

- C#中的可用运算符
- 处理引用类型和值类型时相等的含义
- 基本数据类型之间的数据转换
- 使用装箱技术把值类型转换为引用类型
- 通过强制转换技术在引用类型之间转换
- 重载标准的运算符，以支持对定制类型的操作
- 给定制类型添加强制转换运算符，以支持无缝的数据类型转换

### 6.1 运算符

C 和 C++开发人员应很熟悉大多数 C#运算符，这里为新程序员和 Visual Basic 开发人员介绍最重要的运算符，并介绍 C#中的一些新变化。

C#支持表 6-1 所示的运算符。

表 6-1

类 别	运 算 符
算术运算符	+ - * / %
逻辑运算符	&   ^ ~ &&    !
字符串连接运算符	+
增量和减量运算符	++ --
移位运算符	<< >>
比较运算符	== != <> <= >=
赋值运算符	= += -= *= /= %= &=  = ^= <<= >>=
成员访问运算符(用于对象和结构)	.
索引运算符(用于数组和索引器)	[]
数据类型转换运算符	()
条件运算符 (三元运算符)	?:
委托连接和删除运算符(见第 7 章)	+ -

(续表)

类 别	运 算 符
对象创建运算符	new
类型信息运算符	sizeof (只用于不安全的代码) is typeof as
溢出异常控制运算符	checked unchecked
间接寻址运算符	* -> & (只用于不安全代码) []
命名空间别名限定符(见第 2 章)	::
空接合运算符	??

有 4 个运算符(sizeof、\*、->、&)只能用于不安全的代码(这些代码绕过了 C#类型安全性的检查)，这些不安全的代码见第 12 章的讨论。还要注意，sizeof 运算符在.NET Framework 1.0 和 1.1 中使用，它需要不安全模式。自从.NET Framework 2.0 以来，就没有这个运算符了。

类 别	运 算 符
运算符关键字	sizeof(仅用于.NET Framework 1.0 和 1.1)
运算符	*, ->, &

使用 C#运算符的一个最大缺点是，与 C 风格的语言一样，赋值(=)和比较(==)运算使用不同的运算符。例如，下述语句表示“x 等于 3”：

```
x = 3;

如果要比较 x 和另一个值，就需要使用两个等号(==):

if (x == 3)
{
}
```

C#非常严格的类型安全规则防止出现常见的 C#错误，也就是在逻辑语句中使用赋值运算符代替比较运算符。在 C#中，下述语句会产生一个编译错误：

```
if (x = 3)
{
}
```

习惯使用宏字符&来连接字符串的 Visual Basic 程序员必须改变这个习惯。在 C#中，使用加号+连接字符串，而&表示两个不同整数值的按位 AND 运算。| 则在两个整数之间执行按位 OR 运算。Visual Basic 程序员可能还没有使用过%(取模)运算符，它返回除运算的余数，例如，如果 x 等于 7，则 x % 5 会返回 2。

在 C#中很少会用到指针，因此也很少用到间接寻址运算符(->)。使用它们的唯一场合是在不安全的代码块中，因为只有在此 C#才允许使用指针。指针和不安全的代码见第 12 章。

6.1.1 运算符的简化操作

表 6-2 列出了 C#中的全部简化赋值运算符。

表 6-2

运算符的简化操作	等 价 于
x++, ++x	x = x + 1
x--, --x	x = x - 1
x += y	x = x + y
x -= y	x = x - y
x *= y	x = x * y
x /= y	x = x / y
x %= y	x = x % y
x >>= y	x = x >> y
x <<= y	x = x << y
x &= y	x = x & y
x  = y	x = x   y
x ^= y	x = x ^ y

为什么用两个例子来说明++增量和--减量运算符？把运算符放在表达式的前面称为前置，把运算符放在表达式的后面称为后置。它们的执行方式有所不同。

增量或减量运算符可以作用于整个表达式，也可以作用于表达式的内部。当 x++和++x 单独占一行时，它们的作用是相同的，对应于语句 x = x + 1。但当它们用于表达式内部时，把运算符放在前面(++x)会在计算表达式之前递增 x，换言之，递增了 x 后，在表达式中使用新值进行计算。而把运算符放在后面(x++)会在计算表达式之后递增 x——使用 x 的原值计算表达式。下面的例子使用++增量运算符说明了它们的区别：

```
int x = 5;

if (++x == 6) // true - x is incremented to 6 before the evaluation
{
    Console.WriteLine("This will execute");
}
if (x++ == 7) // false - x is incremented to 7 after the evaluation
{
    Console.WriteLine("This won't");
}
```

第一个 if 条件得到 true，因为在计算表达式之前，x 从 5 递增为 6。第二个 if 语句中的条件为 false，因为在计算完整个表达式(x=6)后，x 才递增为 7。

前置运算符--x 和后置运算符 x--与此类似，但它们是递减，而不是递增。  
其他简化运算符，如+= 和 -=需要两个操作数，用于执行算术、逻辑和按位运算，改变第一个操作数的值。例如，下面两行代码是等价的：

```
x += 5;
x = x + 5;
```

下面介绍在 C#代码中频繁使用的基本运算符和类型转换运算符。

### 6.1.2 条件运算符

条件运算符(?:)也称为三元运算符,是 if...else 结构的简化形式。其名称的出处是它带有三个操作数。它可以计算一个条件,如果条件为真,就返回一个值;如果条件为假,则返回另一个值。其语法如下:

```
condition ? true_value : false_value
```

其中 condition 是要计算的 Boolean 型表达式, true\_value 是 condition 为 true 时返回的值, false\_value 是 condition 为 false 时返回的值。

恰当地使用三元运算符,可以使程序非常简洁。它特别适合于给被调用的函数提供两个参数中的一个。使用它可以把 Boolean 值转换为字符串值 true 或 false。它也很适合于显示正确的单数形式或复数形式,例如:

```
int x = 1;
string s = x + " ";
s += (x == 1 ? "man" : "men");
Console.WriteLine(s);
```

如果 x 等于 1,这段代码就显示 1 man,如果 x 等于其他数,就显示其正确的复数形式。但要注意,如果结果需要用在不同的语言中,就必须编写更复杂的例程,以考虑到不同语言的不同语法。

### 6.1.3 checked 和 unchecked 运算符

考虑下面的代码:

```
byte b = 255;
b++;
Console.WriteLine(b.ToString());
```

byte 数据类型只能包含 0~255 的数,所以递增 b 的值会导致溢出。CLR 如何处理这个溢出取决于许多方面,包括编译器选项,所以只要有未预料到的溢出风险,就需要用某种方式确保得到我们想要的结果。

为此,C#提供了 checked 和 unchecked 运算符。如果把一个代码块标记为 checked,CLR 就会执行溢出检查,如果发生溢出,就抛出异常。下面修改代码,使之包含 checked 运算符:

```
byte b = 255;
checked
{
    b++;
}
Console.WriteLine(b.ToString());
```

运行这段代码,就会得到一个错误信息:

```
Unhandled Exception: System.OverflowException: Arithmetic operation resulted in an overflow.
   at Wrox.ProCSharp.Basics.OverflowTest.Main(String[] args)
```

**注意:**

用 /checked 编译器选项进行编译,就可以检查程序中所有未标记代码中的溢出。



如果要禁止溢出检查，可以把代码标记为 `unchecked`：

```
byte b = 255;
unchecked
{
    b++;
}
Console.WriteLine(b.ToString());
```

在本例中，不会抛出异常，但会丢失数据——因为 `byte` 数据类型不能包含 256，溢出的位会被丢掉，所以 `b` 变量得到的值是 0。

注意，`unchecked` 是默认值。只有在需要把几个未检查的代码行放在一个明确标记为 `checked` 的大代码块中，才需要显式使用 `unchecked` 关键字。

#### 6.1.4 is 运算符

`is` 运算符可以检查对象是否与特定的类型兼容。“兼容”表示对象是该类型，或者派生于该类型。例如，要检查变量是否与 `object` 类型兼容，可以使用下面的代码：

```
int i = 10;
if (i is object)
{
    Console.WriteLine("i is an object");
}
```

`int` 和其他 C# 数据类型一样，也从 `object` 继承而来；表达式 `i is object` 将得到 `true`，并显示相应的信息。

#### 6.1.5 as 运算符

`as` 运算符用于执行引用类型的显式类型转换。如果要转换的类型与指定的类型兼容，转换就会成功进行；如果类型不兼容，`as` 运算符就会返回值 `null`。如下面的代码所示，如果 `object` 引用不指向 `string` 实例，把 `object` 引用转换为 `string` 就会返回 `null`：

```
object o1 = "Some String";
object o2 = 5;

string s1 = o1 as string;    //s1 = "Some String"
string s2 = o2 as string;    //s1 = null
```

`as` 运算符允许在一步中进行安全的类型转换，不需要先使用 `is` 运算符测试类型，再执行转换。

#### 6.1.6 sizeof 运算符

使用 `sizeof` 运算符可以确定堆栈中值类型需要的长度(单位是字节)：

```
unsafe
{
    Console.WriteLine(sizeof(int));
}
```

其结果是显示数字 4，因为 `int` 有 4 个字节。

注意，只能在不安全的代码中使用 `sizeof` 运算符。第 12 章将详细论述不安全的代码。

### 6.1.7 typeof 运算符

`typeof` 运算符返回一个表示特定类型的 `System.Type` 对象。例如，`typeof(string)` 返回表示 `System.String` 类型的 `Type` 对象。在使用反射技术动态查找对象的信息时，这个运算符是很有效的。第 13 章将介绍反射。

### 6.1.8 可空类型和运算符

对于布尔类型，可以给它指定 `true` 或 `false` 值。但是，要把该类型的值定义为 `undefined`，该怎么办？此时使用可空类型可以给应用程序提供一个独特的值。如果在程序中使用可空类型，就必须考虑 `null` 值在与各种运算符一起使用时的影响。通常可空类型与一元或二元运算符一起使用时，如果其中一个操作数或两个操作数都是 `null`，其结果就是 `null`。例如：

```
int? a = null;

int? b = a + 4;    // b = null
int? c = a * 5;    // c = null
```

但是在比较可空类型时，只要有一个操作数是 `null`，比较的结果就是 `false`。即不能因为一个条件是 `false`，就认为该条件的对立面是 `true`，这在使用非可空类型的程序中很常见。例如：

```
int? a = null;
int? b = -5;

if (a >= b)
    Console.WriteLine("a >= b");
else
    Console.WriteLine("a < b");
```

注意：

`null` 值的可能性表示，不能随意合并表达式中的可空类型和非可空类型，详见本章后面的内容。

### 6.1.9 空接合运算符

空接合运算符(`??`)提供了一种快捷方式，可以在处理可空类型和引用类型时表示 `Null` 值。这个运算符放在两个操作数之间，第一个操作数必须是一个可空类型或引用类型，第二个操作数必须与第一个操作数的类型相同，或者可以隐含地转换为第一个操作数的类型。空接合运算符的计算如下：如果第一个操作数不是 `null`，则整个表达式就等于第一个操作数的值。但如果第一个操作数是 `null`，则整个表达式就等于第二个操作数的值。例如：

```
int? a = null;
int b;

b = a ?? 10;    // b has the value 10
a = 3;
```

```
b = a ?? 10; // b has the value 3
```

如果第二个操作数不能隐含地转换为第一个操作数的类型，就生成一个编译错误。

6.1.10 运算符的优先级

表 6-3 显示了 C#运算符的优先级。表顶部的运算符有最高的优先级(即在包含多个运算符的表达式中，最先计算该运算符)：

表 6-3

组	运 算 符
初级运算符	() . [] x++ x-- new typeof sizeof checked unchecked
一元运算符	+ - ! ~ ++x --x 和数据类型转换
乘/除运算符	* / %
加/减运算符	+ -
移位运算符	<< >>
关系运算符	< > <= >= is as
比较运算符	== !=
按位 AND 运算符	&
按位 XOR 运算符	^
按位 OR 运算符	
布尔 AND 运算符	&&
布尔 OR 运算符	
条件运算符	?:
赋值运算符	= += -= *= /= %= &=  = ^= <<= >>= >>>=

注意：

在复杂的表达式中，应避免利用运算符优先级来生成正确的结果。使用括号指定运算符的执行顺序，可以使代码更整洁，避免出现潜在的冲突。

6.2 类型的安全性

第 1 章提到中间语言(IL)可以对其代码强制加上强类型安全性。强类型支持.NET 提供的许多服务，包括安全性和语言的交互性。因为 C#这种语言会编译为 IL，所以 C#也是强类型的。这说明数据类型并不总是可互换的。本节将介绍基本类型之间的转换。

注意：

C#还支持在不同引用类型之间的转换，允许指定自己创建的数据类型如何与其他类型进行相互转换。这些论题将在本章后面讨论。

泛型是 C#中的一个特性，它可以避免对一些常见的情形进行类型转换，泛型详见第 9 章。

6.2.1 类型转换

我们常常需要把数据从一种类型转换为另一种类型。考虑下面的代码：

```
byte value1 = 10;
byte value2 = 23;
byte total;
total = value1 + value2;
Console.WriteLine(total);
```

在编译这些代码时，会产生一个错误：

Cannot implicitly convert type 'int' to 'byte' (不能把 int 类型隐式地转换为 byte 类型)。

问题是，我们把两个 byte 型数据加在一起时，应返回 int 型结果，而不是另一个 byte。这是因为 byte 包含的数据只能为 8 位，所以把两个 byte 型数据加在一起，很容易得到不能存储在 byte 变量中的值。如果要把结果存储在一个 byte 变量中，就必须把它转换回 byte。C#有两种转换方式：隐式转换方式和显式转换方式。

1. 隐式转换方式

只要能保证值不会发生任何变化，类型转换就可以自动进行。这就是前面代码失败的原因：试图从 int 转换为 byte，而潜在地丢失了 3 个字节的数据。编译器不会告诉我们该怎么做，除非我们明确告诉它这就是我们希望的！如果在 long 型变量中存储结果，而不是 byte 型变量中，就不会有问题了：

```
byte value1 = 10;
byte value2 = 23;
long total; // this will compile fine
total = value1 + value2;
Console.WriteLine(total);
```

这是因为 long 类型变量包含的数据字节比 byte 类型多，所以数据没有丢失的危险。在这些情况下，编译器会很顺利地转换，我们也不需要显式提出要求。

表 6-4 介绍了 C#支持的隐式类型转换。

表 6-4

源 类 型	目 的 类 型
sbyte	short、int、long、float、double、decimal
byte	short、ushort、int、uint、long、ulong、float、double、decimal
short	int、long、float、double、decimal
ushort	int、uint、long、ulong、float、double、decimal
int	long、float、double、decimal
uint	long、ulong、float、double、decimal
long、ulong	float、double、decimal
float	double
char	ushort、int、uint、long、ulong、float、double、decimal

注意，只能从较小的整数类型隐式地转换为较大的整数类型，不能从较大的整数类型隐式地转换为较小的整数类型。也可以在整数和浮点数之间转换，其规则略有不同，可以在相同大小的类型之间转换，例如 `int/uint` 转换为 `float`，`long/ulong` 转换为 `double`，也可以从 `long/ulong` 转换回 `float`。这样做可能会丢失 4 个字节的数据，但这仅表示得到的 `float` 值比使用 `double` 得到的值精度低，编译器认为这是一种可以接受的错误，而其值的大小是不会受到影响的。无符号的变量可以转换为有符号的变量，只要无符号的变量值的大小在有符号的变量的范围之内即可。

在隐式转换值类型时，可空类型需要额外考虑：

- 可空类型隐式转换为其他可空类型，应遵循表 6-4 中非可空类型的转换规则。即 `int?` 隐式转换为 `long?`、`float?`、`double?` 和 `decimal?`。
- 非可空类型隐式转换为可空类型也遵循表 6-4 中的转换规则，即 `int` 隐式转换为 `long?`、`float?`、`double?` 和 `decimal?`。
- 可空类型不能隐式转换为非可空类型，此时必须进行显式转换，如下一节所述。这是因为可空类型的值可以是 `null`，但非可空类型不能表示这个值。

## 2. 显式转换方式

有许多场合不能隐式地转换类型，否则编译器会报告错误。下面是不能进行隐式转换的一些场合：

- `int` 转换为 `short`——会丢失数据
- `int` 转换为 `uint`——会丢失数据
- `uint` 转换为 `int`——会丢失数据
- `float` 转换为 `int`——会丢失小数点后面的所有数据
- 任何数字类型转换为 `char`——会丢失数据
- `decimal` 转换为任何数字类型——因为 `decimal` 类型的内部结构不同于整数和浮点数
- `int?` 转换为 `int`——可空类型的值可以是 `null`

但是，可以使用 `cast` 显式执行这些转换。在把一种类型强制转换为另一种类型时，要迫使编译器进行转换。类型转换的一般语法如下：

```
long val = 30000;
int i = (int)val; // A valid cast. The maximum int is 2147483647
```

这表示，把转换的目标类型名放在要转换的值之前的圆括号中。对于熟悉 C 的程序员来说，这是数据类型转换的典型语法。对于熟悉 C++ 数据类型转换关键字(如 `static_cast`)的程序员来说，这些关键字在 C# 中不存在，必须使用 C 风格的旧语法。

这种类型转换是一种比较危险的操作，即使在从 `long` 转换为 `int` 这样简单的转换过程中，如果原来 `long` 的值比 `int` 的最大值还大，就会出问题：

```
long val = 30000000000;
int i = (int)val; // An invalid cast. The maximum int is 2147483647
```

在本例中，不会报告错误，也得不到期望的结果。如果运行上面的代码，结果存储在 `i` 中，则其值为：



```
- 1294967296
```

最好假定显式数据转换不会给出希望的结果。如前所述，C#提供了一个 `checked` 运算符，使用它可以测试操作是否会产生算术溢出。使用这个运算符可以检查数据类型转换是否安全，如果不安全，就会让运行库抛出一个溢出异常：

```
long val = 30000000000;
int i = checked ((int)val);
```

记住，所有的显式数据类型转换都可能不安全，在应用程序中应包含这样的代码，处理可能失败的数据类型转换。第14章将使用 `try` 和 `catch` 语句引入结构化异常处理。

使用数据类型转换可以把大多数数据从一种基本类型转换为另一种基本类型。例如：给 `price` 加上 0.5，再把结果转换为 `int`：

```
double price = 25.30;
int approximatePrice = (int)(price + 0.5);
```

这么做的代价是把价格四舍五入为最接近的美元数。但在这个转换过程中，小数点后面的所有数据都会丢失。因此，如果要使用这个修改过的价格进行更多的计算，最好不要使用这种转换；如果要输出全部计算完或部分计算完的近似值，且不希望用小数点后面的数据去麻烦用户，这种转换是很好的。

下面的例子说明了把一个无符号的整数转换为 `char` 型时，会发生的情况：

```
ushort c = 43;
char symbol = (char)c;
Console.WriteLine(symbol);
```

结果是 ASCII 编码为 43 的字符，即 `+` 号。可以尝试数字类型之间的任何转换(包括 `char`)，这种转换是成功的，例如把 `decimal` 转换为 `char`，或把 `char` 转换为 `decimal`。

值类型之间的转换并不仅限于孤立的变量。还可以把类型为 `double` 的数组元素转换为类型为 `int` 的结构成员变量：

```
struct ItemDetails
{
    public string Description;
    public int ApproxPrice;
}

//...

double[] Prices = { 25.30, 26.20, 27.40, 30.00 };

ItemDetails id;
id.Description = "Whatever";
id.ApproxPrice = (int)(Prices[0] + 0.5);
```

要把一个可空类型转换为非可空类型，或转换为另一个可空类型，但可能会丢失数据，就必须使用显式转换。重要的是，在底层基本类型相同的元素之间进行转换时，就一定要使用显式转换，例如 `int?` 转换为 `int`，或 `float?` 转换为 `float`。这是因为可空类型的值可以是 `null`，非可空类型不能表示这个值。只要可以在两个非可空类型之间进行显式转换，对应可空类型之间的显式转换就可以进行。但如果从可空类型转换为非可空类型，且变量的值是 `null`，就会抛出

InvalidOperationException。例如：

```
int? a = null;
int b = (int)a; // Will throw exception
```

使用显式的数据类型转换方式，并小心使用这种技术，就可以把简单值类型的任何实例转换为另一种类型。但在进行显式的类型转换时有一些限制，例如值类型，只能在数字、char 类型和 enum 类型之间转换。不能直接把 Boolean 数据类型转换为其他类型，也不能把其他类型转换为 Boolean 数据类型。

如果需要在数字和字符串之间转换，.NET 类库提供了一些方法。Object 类有一个 ToString() 方法，该方法在所有的 .NET 预定义类型中都进行了重写，返回对象的字符串表示：

```
int i = 10;
string s = i.ToString();
```

同样，如果需要分析一个字符串，获得一个数字或 Boolean 值，就可以使用所有预定义值类型都支持的 Parse 方法：

```
string s = "100";
int i = int.Parse(s);
Console.WriteLine(i + 50); // Add 50 to prove it is really an int
```

注意，如果不能转换字符串(例如要把字符串 Hello 转换为一个整数)，Parse 方法就会注册一个错误，抛出一个异常。第 14 章将介绍异常。

## 6.2.2 装箱和拆箱

第 2 章介绍了所有类型，包括简单的预定义类型，例如 int 和 char，以及复杂类型，例如从 Object 类型中派生的类和结构。下面可以像处理对象那样处理字面值：

```
string s = 10.ToString();
```

但是，C#数据类型可以分为在堆栈上分配内存的值类型和在堆上分配内存的引用类型。如果 int 不过是堆栈上一个 4 字节的值，该如何在它上面调用方法？

C#的实现方式是通过一个有点魔术性的方式，即装箱(boxing)。装箱和拆箱(unboxing)可以把值类型转换为引用类型，或把引用类型转换为值类型。这已经在数据类型转换一节中介绍过了，即把值转换为 object 类型。装箱用于描述把一个值类型转换为引用类型。运行库会为堆上的对象创建一个临时的引用类型“box”。

该转换是隐式进行的，如上面的例子所述。还可以进行显式转换：

```
int myIntNumber = 20;
object myObject = myIntNumber;
```

拆箱用于描述相反的过程，即以前装箱的值类型转换回值类型。这里使用术语“cast”，是因为这种数据类型转换是显式进行的。其语法类似于前面的显式类型转换：

```
int myIntNumber = 20;
object myObject = myIntNumber; // Box the int
int mySecondNumber = (int)myObject; // Unbox it back into an int
```

只能把以前装箱的变量再转换为值类型。当 `o` 不是装箱后的 `int` 型时，如果执行上面的代码，就会在运行期间抛出一个异常。

这里有一个警告。在拆箱时，必须非常小心，确保得到的值变量有足够的空间存储拆箱的值中的所有字节。例如，C#的 `int` 只有 32 位，所以把 `long` 值(64 位)拆箱为 `int` 时，会产生一个 `InvalidCastException` 异常：

```
long myLongNumber = 333333423;
object myObject = (object)myLongNumber;
int myIntNumber = (int)myObject;
```

## 6.3 对象的相等比较

在讨论了运算符，并简要介绍了相等运算符后，就应考虑在处理类和结构的实例时，“相等”意味着什么。理解对象相等比较的机制对编写逻辑表达式非常重要，另外，对实现运算符重载和数据类型转换也非常重要，本章的后面将讨论运算符重载。

对象相等比较的机制对于引用类型(类的实例)的比较和值类型(基本数据类型，结构或枚举的实例)的比较来说是不同的。下面分别介绍引用类型和值类型的相等比较。

### 6.3.1 引用类型的相等比较

`System.Object` 定义了 3 个不同的方法，来比较对象的相等性：`ReferenceEquals()`和 `Equals()` 的两个版本。再加上比较运算符 `==`，实际上有 4 种进行相等比较的方式。这些方法有一些微妙的区别，下面就介绍这些方法。

#### 1. ReferenceEquals()方法

`ReferenceEquals()`是一个静态方法，测试两个引用是否指向类的同一个实例，即两个引用是否包含内存中的相同地址。作为静态方法，它不能重写，所以只能使用 `System.Object` 的实现代码。如果提供的两个引用指向同一个对象实例，`ReferenceEquals()`总是返回 `true`，否则就返回 `false`。但是它认为 `null` 等于 `null`：

```
SomeClass x, y;
x = new SomeClass();
y = new SomeClass();
bool B1 = ReferenceEquals(null, null);           //return true
bool B2 = ReferenceEquals(null, x);              //return false
bool B3 = ReferenceEquals(x, y);                 //return false because x and y
                                                //point to different objects
```

#### 2. 虚拟的 Equals()方法

`Equals()`虚拟版本的 `System.Object` 实现代码也可以比较引用。但因为这个方法是虚拟的，所以可以在自己的类中重写它，按值来比较对象。特别是如果希望类的实例用作字典中的键，就需要重写这个方法，以比较值。否则，根据重写 `Object.GetHashCode()`的方式，包含对象的字典类要么不工作，要么工作的效率非常低。在重写 `Equals()`方法时要注意，重写的代码不会抛出异常。这是因为如果抛出异常，字典类就会出问题，一些在内部调用这个方法的.NET 基

类也可能出问题。

### 3. 静态的 Equals()方法

Equals()的静态版本与其虚拟实例版本的作用相同，其区别是静态版本带有两个参数，并对它们进行相等比较。这个方法可以处理两个对象中有一个是 null 的情况，因此，如果一个对象可能是 null，这个方法就可以抛出异常，提供额外的保护。静态重载版本首先要检查它传送的引用是否为 null。如果它们都是 null，就返回 true(因为 null 与 null 相等)。如果只有一个引用是 null，就返回 false。如果两个引用都指向某个对象，它就调用 Equals()的虚拟实例版本。这表示在重写 Equals()的实例版本时，其效果相当于也重写了静态版本。

### 4. 比较运算符==

最好将比较运算符看作是严格的值比较和严格的引用比较之间的中间选项。在大多数情况下，下面的代码表示比较引用：

```
bool b = (x == y); //x, y object references
```

但是，如果把一些类看作值，其含义就会比较直观。在这些情况下，最好重写比较运算符，以执行值的比较。后面将讨论运算符的重载，但显然它的一个例子是 System.String 类，Microsoft 重写了这个运算符，比较字符串的内容，而不是它们的引用。

## 6.3.2 值类型的相等比较

在进行值类型的相等比较时，采用与引用类型相同的规则：ReferenceEquals()用于比较引用，Equals()用于比较值，比较运算符可以看作是一个中间项。但最大的区别是值类型需要装箱，才能把它们转换为引用，才能对它们执行方法。另外，Microsoft 已经在 System.ValueType 类中重载了实例方法 Equals()，以便对值类型进行合适的相等测试。如果调用 sA.Equals(sB)，其中 sA 和 sB 是某个结构的实例，则根据 sA 和 sB 是否在其所有的字段中包含相同的值，而返回 true 或 false。另一方面，在默认情况下，不能对自己的结构重载==运算符。在表达式中使用(sA==sB)会导致一个编译错误，除非在代码中为结构提供了==的重载版本。

另外，ReferenceEquals()在应用于值类型时，总是返回 false，因为为了调用这个方法，值类型需要装箱到对象中。即使使用下面的代码：

```
bool b = ReferenceEquals(v, v); //v is a variable of some value type
```

也会返回 false，因为在转换每个参数时，v 都会被单独装箱，这意味着会得到不同的引用。调用 ReferenceEquals()来比较值类型实际上没有什么意义。

尽管 System.ValueType 提供的 Equals()的默认重写代码肯定足以应付绝大多数自定义的结构，但仍可以为自己的结构重写它，以提高性能。另外，如果值类型包含作为字段的引用类型，就需要重写 Equals()，以便为这些字段提供合适的语义，因为 Equals()的默认重写版本仅比较它们的地址。



## 6.4 运算符重载

本节将介绍为类或结构定义的另一类型成员：运算符重载。

C++开发人员应很熟悉运算符重载。但是，因为这个概念对 Java 和 Visual Basic 开发人员来说是全新的，所以这里要解释一下。C++开发人员可以直接跳到主要示例上。

运算符重载的关键是在类实例上不能总是调用方法或属性，有时还需要做一些其他的工作，例如对数值进行相加、相乘或逻辑操作，如比较对象等。假定要定义一个类，表示一个数学矩阵，在数学中，矩阵可以相加和相乘，就像数字一样。所以可以编写下面的代码：

```
Matrix a, b, c;
// assume a, b and c have been initialized
Matrix d = c * (a + b);
```

通过重载运算符，就可以告诉编译器，+和\*对 Matrix 对象进行什么操作，以编写上面的代码。如果用不支持运算符重载的语言编写代码，就必须定义一个方法，以执行这些操作，结果肯定不太直观，如下所示。

```
Matrix d = c.Multiply(a.Add(b));
```

学习到现在，像+和\*这样的运算符只能用于预定义的数据类型，原因很简单：编译器认为所有常见的运算符都是用于这些数据类型的，例如，它知道如何把两个 long 加起来，或者如何从一个 double 中减去另一个 double，并生成合适的中间语言代码。但在定义自己的类或结构时，必须告诉编译器：什么方法可以调用，每个实例存储了什么字段等所有的信息。同样，如果要在自己的类上使用运算符，就必须告诉编译器相关的运算符在这个类中的含义。此时就要定义运算符重载。

要强调的另一个问题是重载不仅仅限于算术运算符。还需要考虑比较运算符 ==、<、>、!=、>=和<=。例如，语句 if(a==b)。对于类，这个语句在默认状态下会比较引用 a 和 b，检测这两个引用是否指向内存中的同一个地址，而不是检测两个实例是否包含相同的数据。对于 string 类，这种操作就会重写，比较字符串实际上就是比较每个字符串的内容。可以对自己的类进行这样的操作。对于结构，==运算符在默认状态下不做任何工作。试图比较两个结构，看看它们是否相等，就会产生一个编译错误，除非显式重载了==，告诉编译器如何进行比较。

在许多情况下，重载运算符允许生成可读性更高、更直观的代码，包括：

- 在数学领域中，几乎包括所有的数学对象：坐标、矢量、矩阵、张量和函数等。如果编写一个程序执行某些数学或物理建模，肯定会用类表示这些对象。
- 图形程序在计算屏幕上的位置时，也使用数学或相关的坐标对象。
- 表示大量金钱的类(例如，在财务程序中)。
- 字处理或文本分析程序也有表示语句、子句等的类，可以使用运算符把语句连接在一起(这是字符串连接的一种比较复杂的版本)。

另外，有许多类与运算符重载并不相关。不恰当地使用运算符重载，会使使用类型的代码很难理解。例如，把两个 DateTime 对象相乘，在概念上没有任何意义。



### 6.4.1 运算符的工作方式

为了理解运算符是如何重载的，考虑一下在编译器遇到运算符时会发生什么样的情况是很有用的——我们用相加运算符+作为例子来讲解。假定编译器遇到下面的代码：

```
int myInteger = 3;
uint myUnsignedInt = 2;
double myDouble = 4.0;
long myLong = myInteger + myUnsignedInt;
double myOtherDouble = myDouble + myInteger;
```

会发生什么情况：

```
long myLong = myInteger + myUnsignedInt;
```

编译器知道它需要把两个整数加起来，并把结果赋予 long。调用一个方法把数字加在一起时，表达式 `myInteger + myUnsignedInt` 是一种非常直观、方便的语法。该方法带有两个参数 `myInteger` 和 `myUnsignedInt`，并返回它们的和。所以编译器完成的任务与任何方法调用是一样的——它会根据参数类型查找最匹配的+运算符重载，这里是带两个整数参数的+运算符重载。与一般的重载方法一样，预定义的返回类型不会因为调用的方法版本而影响编译器的选择。在本例中调用的重载方法带两个 int 类型参数，返回一个 int，这个返回值随后会转换为 long。

下一行代码让编译器使用+运算符的另一个重载版本：

```
double myOtherDouble = myDouble + myInteger;
```

在这个例子中，参数是一个 double 类型的数据和一个 int 类型的数据，但+运算符没有带这种复合参数的重载形式，所以编译器认为，最匹配的+运算符重载是把两个 double 作为其参数的版本，并隐式地把 int 转换为 double。把两个 double 加在一起与把两个整数加在一起完全不同，浮点数存储为一个尾数和一个指数。把它们加在一起要按位移动一个 double 的尾数，让两个指数有相同的值，然后把尾数加起来，移动所得尾数的位，调整其指数，保证答案有尽可能高的精度。

现在，看看如果编译器遇到下面的代码，会发生什么：

```
Vector vect1, vect2, vect3;
// initialise vect1 and vect2
vect3 = vect1 + vect2;
vect1 = vect1*2;
```

其中，Vector 是结构，稍后再定义它。编译器知道它需要把两个 Vector 实例加起来，即 `vect1` 和 `vect2`。它会查找+运算符的重载，把两个 Vector 实例作为参数。

如果编译器找到这样的重载版本，就调用它的实现代码。如果找不到，就要看看有没有可以用作最佳匹配的其他+运算符重载，例如某个运算符重载的参数是其他数据类型，但可以隐式地转换为 Vector 实例。如果编译器找不到合适的运算符重载，就会产生一个编译错误，就像找不到其他方法调用的合适重载版本一样。

### 6.4.2 运算符重载的示例：Vector 结构

本节将开发一个结构 Vector，来演示运算符重载，这个 Vector 结构表示一个三维矢量。如果数学不是你的强项，不必担心，我们会使这个例子尽可能简单。三维矢量只是三个(double)

数字的一个集合，说明物体和原点之间的距离，表示数字的变量是  $x$ 、 $y$  和  $z$ ， $x$  表示物体与原点在  $x$  方向上的距离， $y$  表示它与原点在  $y$  方向上的距离， $z$  表示高度。把这 3 个数字组合起来，就得到总距离。例如，如果  $x=3.0$ ， $y=3.0$ ， $z=1.0$ ，一般可以写作  $(3.0, 3.0, 1.0)$ ，表示物体与原点在  $x$  方向上的距离是 3，与原点在  $y$  方向上的距离是 3，高度为 1。

矢量可以与矢量或数字相加或相乘。在这里我们使用术语“标量”(scalar)，它是数字的数学用语——在 C# 中，就是一个 `double`。相加的作用是很明显的。如果先移动  $(3.0, 3.0, 1.0)$ ，再移动  $(2.0, -4.0, -4.0)$ ，总移动量就是把这两个矢量加起来。矢量的相加是指把每个元素分别相加，因此得到  $(5.0, -1.0, -3.0)$ 。此时，数学表达式总是写成  $c=a+b$ ，其中  $a$  和  $b$  是矢量， $c$  是结果矢量。这与使用 `Vector` 结构的方式是一样的。

注意：

这个例子是作为一个结构来开发的，而不是类，但这并不重要。运算符重载用于结构和类时，其工作方式是一样的。

下面是 `Vector` 的定义——包含成员字段、构造函数和一个 `ToString()` 重写方法，以便查看 `Vector` 的内容，最后是运算符重载：

```
namespace Wrox.ProCSharp.OOCSharp
{
    struct Vector
    {
        public double x, y, z;

        public Vector(double x, double y, double z)
        {
            this.x = x;
            this.y = y;
            this.z = z;
        }

        public Vector(Vector rhs)
        {
            x = rhs.x;
            y = rhs.y;
            z = rhs.z;
        }

        public override string ToString()
        {
            return "(" + x + ", " + y + ", " + z + ")";
        }
    }
}
```

这里提供了两个构造函数，通过传递每个元素的值，或者提供另一个复制其值的 `Vector`，来指定矢量的初始值。第二个构造函数带一个 `Vector` 参数，通常称为复制构造函数，因为它们允许通过复制另一个实例来初始化一个类或结构实例。注意，为了简单起见，把字段设置为 `public`。也可以把它们设置为 `private`，编写相应的属性来访问它们，这样做不会改变这个程序的功能，只是代码会复杂一些。

下面是 `Vector` 结构的有趣部分——为 `+` 运算符提供支持的运算符重载：

```
public static Vector operator + (Vector lhs, Vector rhs)
{
    Vector result = new Vector(lhs);
    result.x += rhs.x;
```

```

        result.y += rhs.y;
        result.z += rhs.z;
        return result;
    }
}

```

运算符重载的声明方式与方法的声明方式相同，但 `operator` 关键字告诉编译器，它实际上是一个运算符重载，后面是相关运算符的符号，在本例中就是`+`。返回类型是在使用这个运算符时获得的类型。在本例中，把两个矢量加起来会得到另一个矢量，所以返回类型就是 `Vector`。对于这个`+`运算符重载，返回类型与包含类一样，但这种情况并不是必需的。两个参数就是要操作的对象。对于二元运算符(带两个参数)，如`+`和`-`运算符，第一个参数是放在运算符左边的值，第二个参数是放在运算符右边的值。

**注意：**

一般把运算符左边的参数命名为 `lhs`，运算符右边的参数命名为 `rhs`。

C#要求所有的运算符重载都声明为 `public` 和 `static`，这表示它们与它们的类或结构相关联，而不是与实例相关联，所以运算符重载的代码体不能访问非静态类成员，也不能访问 `this` 标识符；这是可以的，因为参数提供了运算符执行任务所需要知道的所有数据。

前面介绍了声明运算符`+`的语法，下面看看运算符内部的情况：

```

{
    Vector result = new Vector(lhs);
    result.x += rhs.x;
    result.y += rhs.y;
    result.z += rhs.z;
    return result;
}

```

这部分代码与声明方法的代码是完全相同的，显然，它返回一个矢量，其中包含前面定义的 `lhs` 和 `rhs` 的和，即把 `x`、`y` 和 `z` 分别相加。

下面需要编写一些简单的代码，测试 `Vector` 结构：

```

static void Main()
{
    Vector vect1, vect2, vect3;

    vect1 = new Vector(3.0, 3.0, 1.0);
    vect2 = new Vector(2.0, -4.0, -4.0);
    vect3 = vect1 + vect2;

    Console.WriteLine("vect1 = " + vect1.ToString());
    Console.WriteLine("vect2 = " + vect2.ToString());
    Console.WriteLine("vect3 = " + vect3.ToString());
}

```

把这些代码保存为 `Vectors.cs`，编译并运行它，结果如下：

**Vectors**

```

vect1 = ( 3 , 3 , 1 )
vect2 = ( 2 , -4 , -4 )
vect3 = ( 5 , -1 , -3 )

```

## 1. 添加更多的重载

矢量除了可以相加之外，还可以相乘、相减，比较它们的值。本节通过添加几个运算符重载，扩展了这个例子。这并不是一个功能全面的真实的 `Vector` 类型，但足以说明运算符重载的其他方面了。首先要重载乘法运算符，以支持标量和矢量的相乘以及矢量和矢量的相乘。

矢量乘以标量只是矢量的元素分别与标量相乘，例如， $2 * (1.0, 2.5, 2.0)$  就等于  $(2.0, 5.0, 4.0)$ 。相关的运算符重载如下所示。

```
public static Vector operator * (double lhs, Vector rhs)
{
    return new Vector(lhs * rhs.x, lhs * rhs.y, lhs * rhs.z);
}
```

但这还不够，如果 `a` 和 `b` 声明为 `Vector` 类型，就可以编写下面的代码：

```
b = 2 * a;
```

编译器会隐式地把整数 2 转换为 `double` 类型，以匹配运算符重载的签名。但不能编译下面的代码：

```
b = a * 2;
```

编译器处理运算符重载的方式和处理方法重载的方式是一样的。它会查看给定运算符的所有可用重载，找到与之最匹配的那个运算符重载。上面的语句要求第一个参数是 `Vector`，第二个参数是整数，或者可以隐式转换为整数的其他数据类型。我们没有提供这样一个重载。有一个运算符重载，其参数是一个 `double` 和一个 `Vector`，但编译器不能改变参数的顺序，所以这是不行的。还需要显式定义一个运算符重载，其参数是一个 `Vector` 和一个 `double`，有两种方式可以定义这样一个运算符重载，第一种方式和处理所有运算符的方式一样，显式执行矢量相乘操作：

```
public static Vector operator * (Vector lhs, double rhs)
{
    return new Vector(rhs * lhs.x, rhs * lhs.y, rhs * lhs.z);
}
```

假定已经编写了执行相乘操作的代码，最好重复使用该代码：

```
public static Vector operator * (Vector lhs, double rhs)
{
    return rhs * lhs;
}
```

这段代码会告诉编译器，如果有 `Vector` 和 `double` 的相乘操作，编译器就使参数的顺序反序，调用另一个运算符重载。在本章的示例代码中，我们使用第二个版本，它看起来比较简洁。利用这个版本可以编写出维护性更好的代码，因为不需要复制代码，就可在两个独立的重载中执行相乘操作。

下一个要重载的运算符是矢量相乘。在数学上，矢量相乘有两种方式，但这里我们感兴趣的是点积或内积，其结果实际上是一个标量。这就是我们介绍这个例子的原因，所以算术运算符不必返回与定义它们的类相同的类型。

在数学上，如果有两个矢量  $(x, y, z)$  和  $(X, Y, Z)$ ，其内积就是  $x*X + y*Y + z*Z$  的值。两个矢量这样相乘是很奇怪的，但这是很有效的，因为它可以用于计算各种其他的数。当然，如果要



使用 Direct3D 或 DirectDraw 编写代码来显示复杂的 3D 图形, 在计算对象放在屏幕上的什么位置时, 常常需要编写代码来计算矢量的内积, 作为中间步骤。这里我们关心的是使用 Vector 编写出  $\text{double } X = a \cdot b$ , 其中  $a$  和  $b$  是矢量, 并计算出它们的点积。相关的运算符重载如下所示:

```
public static double operator * (Vector lhs, Vector rhs)
{
    return lhs.x * rhs.x + lhs.y * rhs.y + lhs.z * rhs.z;
}
```

定义了算术运算符后, 就可以用一个简单的测试方法来看看它们是否能正常运行:

```
static void Main()
{
    // stuff to demonstrate arithmetic operations
    Vector vect1, vect2, vect3;
    vect1 = new Vector(1.0, 1.5, 2.0);
    vect2 = new Vector(0.0, 0.0, -10.0);

    vect3 = vect1 + vect2;

    Console.WriteLine("vect1 = " + vect1);
    Console.WriteLine("vect2 = " + vect2);
    Console.WriteLine("vect3 = vect1 + vect2 = " + vect3);
    Console.WriteLine("2*vect3 = " + 2*vect3);

    vect3 += vect2;

    Console.WriteLine("vect3+=vect2 gives " + vect3);

    vect3 = vect1*2;

    Console.WriteLine("Setting vect3=vect1*2 gives " + vect3);

    double dot = vect1*vect3;

    Console.WriteLine("vect1*vect3 = " + dot);
}
```

运行代码(Vectors2.cs), 得到如下所示的结果:

### Vectors2

```
vect1 = ( 1 , 1.5 , 2 )
vect2 = ( 0 , 0 , -10 )
vect3 = vect1 + vect2 = ( 1 , 1.5 , -8 )
2*vect3 = ( 2 , 3 , -16 )
vect3+=vect2 gives ( 1 , 1.5 , -18 )
Setting vect3=vect1*2 gives ( 2 , 3 , 4 )
vect1*vect3 = 14.5
```

这说明, 运算符重载会给出正确的结果, 但如果仔细看看测试代码, 就会惊奇地注意到, 实际上我们使用的是没有重载的运算符——相加赋值运算符+=:

```
vect3 += vect2;

Console.WriteLine("vect3 += vect2 gives " + vect3);
```

虽然+=一般用作单个运算符, 但实际上其操作分为两部分: 相加和赋值。与 C++ 不同, C#



不允许重载`=`运算符，但如果重载`+`运算符，编译器就会自动使用`+`运算符的重载来执行`+=`运算符的操作。`-=`、`&=`、`*=`和`/=`赋值运算符也遵循此规则。

## 2. 比较运算符的重载

C#中有6个比较运算符，它们分为3对：

- `==` 和 `!=`
- `>` 和 `<`
- `>=` 和 `<=`

C#要求成对重载比较运算符。如果重载了`==`，也必须重载`!=`，否则会产生编译错误。另外，比较运算符必须返回`bool`类型的值。这是它们与算术运算符的根本区别。两个数相加或相减的结果，理论上取决于数的类型。而两个`Vector`的相乘会得到一个标量。另一个例子是.NET基类`System.DateTime`，两个`DateTime`实例相减，得到的结果不是`DateTime`，而是一个`System.TimeSpan`实例，但比较运算得到的如果不是`bool`类型的值，就没有任何意义。

注意：

在重载`==`和`!=`时，还应重载从`System.Object`中继承的`Equals()`和`GetHashCode()`方法，否则会产生一个编译警告。原因是`Equals()`方法应执行与`==`运算符相同的相等逻辑。

除了这些区别外，重载比较运算符所遵循的规则与算术运算符相同。但比较两个数并不像想象的那么简单，例如，如果比较两个对象引用，就是比较存储对象的内存地址。比较运算符很少进行这样的比较，所以必须编写运算符，比较对象的值，返回相应的布尔结果。下面给`Vector`结构重载`==`和`!=`运算符。首先是`==`的执行代码：

```
public static bool operator == (Vector lhs, Vector rhs)
{
    if (lhs.x == rhs.x && lhs.y == rhs.y && lhs.z == rhs.z)
        return true;
    else
        return false;
}
```

这种方式仅根据矢量元素的值，来对它们进行相等比较。对于大多数结构，这就是我们希望的，但在某些情况下，可能需要仔细考虑相等的含义，例如，如果有嵌入的类，是应比较引用是否指向同一个对象(浅度比较)，还是应比较对象的值是否相等(深度比较)？

浅度比较是比较对象是否指向内存中的同一个位置，而深度比较是比较对象的值和属性是否相等。应根据具体情况进行相等检查，确定要进行什么比较。

注意：

不要通过调用从`System.Object`中继承的`Equals()`方法的实例版本，来重载比较运算符，如果这么做，在`objA`是`null`时计算(`objA==objB`)，就会产生一个异常，因为.NET运行库会试图计算`null.Equals(objB)`。采用其他方法(重写`Equals()`方法，调用比较运算符)比较安全。

还需要重载运算符`!=`，采用的方式如下：

```
public static bool operator != (Vector lhs, Vector rhs)
```

```

    {
        return ! (lhs == rhs);
    }

```

像往常一样，用一些测试代码检查重写方法的工作情况，这次定义 3 个 `Vector` 对象，并进行比较：

```

static void Main()
{
    Vector vect1, vect2, vect3;

    vect1 = new Vector(3.0, 3.0, -10.0);
    vect2 = new Vector(3.0, 3.0, -10.0);
    vect3 = new Vector(2.0, 3.0, 6.0);

    Console.WriteLine("vect1==vect2 returns " + (vect1==vect2));
    Console.WriteLine("vect1==vect3 returns " + (vect1==vect3));
    Console.WriteLine("vect2==vect3 returns " + (vect2==vect3));

    Console.WriteLine();

    Console.WriteLine("vect1!=vect2 returns " + (vect1!=vect2));
    Console.WriteLine("vect1!=vect3 returns " + (vect1!=vect3));
    Console.WriteLine("vect2!=vect3 returns " + (vect2!=vect3));
}

```

编译这些代码(下载代码中的 `Vectors3.cs`)，会得到一个编译器警告，因为我们没有为 `Vector` 重写 `Equals()`，对于本例，这是不重要的，所以忽略它。

#### csc Vectors3.cs

```

Microsoft (R) Visual C# 2008 Compiler version 3.05.20706.1
for Microsoft (R) 2005 Framework version 3.5
Copyright (C) Microsoft Corporation. All rights reserved.

```

```

Vectors3.cs(5,11): warning CS0660: 'Wrox.ProCSharp.OOCSharp.Vector' defines
operator == or operator != but does not override Object.Equals(object o)
Vectors3.cs(5,11): warning CS0661: 'Wrox.ProCSharp.OOCSharp.Vector' defines
operator == or operator != but does not override Object.GetHashCode()

```

在命令行上运行该示例，生成如下结果：

#### Vectors3

```

vect1==vect2 returns True
vect1==vect3 returns False
vect2==vect3 returns False

vect1!=vect2 returns False
vect1!=vect3 returns True
vect2!=vect3 returns True

```

### 3. 可以重载的运算符

并不是所有的运算符都可以重载。可以重载的运算符如表 6-5 所示。

表 6-5

类 别	运 算 符	限 制
算术二元运算符	+, *, /, -, %	无
算术一元运算符	+, -, ++, --	无
按位二元运算符	&,  , ^, <<, >>	无
按位一元运算符	!, ~, true, false	true 和 false 运算符必须成对重载
比较运算符	==, !=, >=, <, <=, >	必须成对重载
赋值运算符	+=, -=, *=, /=, >>=, <<=, %=, &=,  =, ^=	不能显式重载这些运算符，在重写单个运算符如 +, -, % 等时，它们会被隐式重写
索引运算符	[]	不能直接重载索引运算符。第 2 章介绍的索引器成员类型允许在类和结构上支持索引运算符
数据类型转换运算符	()	不能直接重载数据类型转换运算符。用户定义的数据类型转换(本章的后面介绍)允许定义定制的数据类型转换

6.5 用户定义的数据类型转换

本章前面介绍了如何在预定义的数据类型之间转换数值，这是通过数据类型转换过程来完成的。C#允许进行两种不同数据类型的转换：隐式转换和显式转换。

显式转换要在代码中显式标记转换，其方法是在圆括号中写出目标数据类型：

```
int I = 3;
long l = I;           // implicit
short s = (short)I;   // explicit
```

对于预定义的数据类型，当数据类型转换可能失败或丢失某些数据时，需要显式转换。例如：

- 把 int 转换为 short 时，因为 short 可能不够大，不能包含转换的数值。
- 把有符号的数据转换为无符号的数据，如果有符号的变量包含一个负值，会得到不正确的结果
- 在把浮点数转换为整数数据类型时，数字的小数部分会丢失。
- 把可空类型转换为非可空类型，null 值会导致异常。

此时应在代码中进行显式转换，告诉编译器你知道这会有丢失数据的危险，因此编写代码时要把这种可能性考虑在内。

C#允许定义自己的数据类型(结构和类)，这意味着需要某些工具支持在自己的数据类型之间进行类型转换。方法是把数据类型转换定义为相关类的一个成员运算符，数据类型转换必须标记为隐式或显式，以说明如何使用它。我们应遵循与预定义数据类型转换相同的规则，如果知道无论在源变量中存储什么值，数据类型转换总是安全的，就可以把它定义为隐式转换。另一方面，如果某些数值可能会出错，例如丢失数据或抛出异常，就应把数据类型转换定义为显式转换。

**提示:**

如果源数据值会使数据类型转换失败,或者可能会抛出异常,就应把定制数据类型转换定义为显式转换。

定义数据类型转换的语法类似于本章前面介绍的重载运算符。但它们是不一致的,数据类型转换在某种情况下可以看作是一种运算符,其作用是从源类型转换为目标类型。为了说明这个语法,下面的代码是从本节后面介绍的结构 `Currency` 示例中节选的:

```
public static implicit operator float (Currency value)
{
    // processing
}
```

运算符的返回类型定义了数据类型转换操作的目标类型,它有一个参数,即要转换的源对象。这里定义的数据类型转换可以隐式地把 `Currency` 的值转换为 `float` 型。注意,如果数据类型转换声明为隐式,编译器可以隐式或显式地使用这个转换。如果数据类型转换声明为显式,编译器就只能显式地使用它。与其他运算符重载一样,数据类型转换必须声明为 `public` 和 `static`。

**注意:**

C++开发人员应注意,这种情况与 C++是不同的,在 C++中,数据类型转换是类的实例成员。

### 6.5.1 执行用户定义的类型转换

本节将在示例 `SimpleCurrency`(和往常一样,其代码可以下载)中介绍隐式和显式使用用户定义的数据类型转换。在这个示例中,定义一个结构 `Currency`,它包含一个正的 USD(\$) 钱款。C# 为此提供了 `decimal` 类型,但如果要进行比较复杂的财务处理,仍可以编写自己的结构和类来表示钱款,在这样的类上执行特定的方法。

**注意:**

数据类型转换的语法对于结构和类是一样的。我们的示例定义了一个结构,但如果把 `Currency` 声明为类,也是可以的。

首先,结构 `Currency` 的定义如下所示。

```
struct Currency
{
    public uint Dollars;
    public ushort Cents;

    public Currency(uint dollars, ushort cents)
    {
        this.Dollars = dollars;
        this.Cents = cents;
    }

    public override string ToString()
    {
        return string.Format("${0}.{1,-2:00}", Dollars, Cents);
    }
}
```

Dollars 和 Cents 字段使用无符号的数据类型，可以确保 Currency 实例只能包含正值。这样限制，是为了在后面说明显式转换的一些要点。可以像这样使用一个类来存储公司员工的薪水信息。人们的薪水不会是负值！为了使类比较简单，我们把字段声明为 public，但通常应把它们声明为 private，并为 Dollars 和 Cents 字段定义相应的属性。

下面先假定要把 Currency 实例转换为 float 值，其中 float 值的整数部分表示美元，换言之，应编写下面的代码：

```
Currency balance = new Currency(10,50);
float f = balance; // We want f to be set to 10.5
```

为此，需要定义一个数据类型转换。给 Currency 定义添加下述代码：

```
public static implicit operator float (Currency value)
{
    return value.Dollars + (value.Cents/100.0f);
}
```

这个数据类型转换是隐式的。在本例中这是一个合理的选择，因为在 Currency 定义中，可以存储在 Currency 中的值也都可以存储在 float 中。在这个转换中，不应出现任何错误。

注意：

这里有一点欺骗性：实际上，当把 uint 转换为 float 时，会有精确度的损失，但 Microsoft 认为这种错误并不重要，因此把从 uint 到 float 的转换都当做隐式转换。

但是，如果把 float 转换为 Currency，就不能保证转换肯定成功了；float 可以存储负值，而 Currency 实例不能，float 存储的数值的量级要比 Currency 的(uint) Dollars 字段大得多。所以，如果 float 包含一个不合适的值，把它转换为 Currency 就会得到意想不到的结果。因此，从 float 转换到 Currency 就应定义为显式转换。下面是我们的第一次尝试，这次不会得到正确的结果，但对解释原因是有帮助的：

```
public static explicit operator Currency (float value)
{
    uint dollars = (uint)value;
    ushort cents = (ushort)((value-dollars)*100);
    return new Currency(dollars, cents);
}
```

下面的代码可以成功编译：

```
float amount = 45.63f;
Currency amount2 = (Currency)amount;
```

但是，下面的代码会抛出一个编译错误，因为试图隐式地使用一个显式的数据类型转换：

```
float amount = 45.63f;
Currency amount2 = amount; // wrong
```

把数据类型转换声明为显式，就是警告开发人员要小心，因为可能会丢失数据。但这不是我们希望的 Currency 结构的执行方式。下面编写一个测试程序，运行示例。其中有一个 Main() 方法，它实例化了一个 Currency 结构，试图进行几个转换。在这段代码的开头，以两种不同的方式计算 balance 的值(因为要使用它们来说明后面的内容)：



```

static void Main()
{
    try
    {
        Currency balance = new Currency(50,35);

        Console.WriteLine(balance);
        Console.WriteLine("balance is " + balance);
        Console.WriteLine("balance is (using ToString()) " + balance.ToString());

        float balance2= balance;

        Console.WriteLine("After converting to float, = " + balance2);

        balance = (Currency) balance2;

        Console.WriteLine("After converting back to Currency, = " + balance);
        Console.WriteLine("Now attempt to convert out of range value of " +
            "-$100.00 to a Currency:");
        checked
        {
            balance = (Currency) (-50.5);
            Console.WriteLine("Result is " + balance.ToString());
        }
    }
    catch(Exception e)
    {
        Console.WriteLine("Exception occurred: " + e.Message);
    }
}

```

注意，所有的代码都放在一个 `try` 块中，来捕获在数据类型转换过程中发生的任何异常。在 `checked` 块中还添加了把超出范围的值转换为 `Currency` 的测试代码，所以，负值是肯定会被捕获的。运行这段代码，得到如下所示的结果：

#### SimpleCurrency

```

50.35
Balance is $50.35
Balance is (using ToString()) $50.35
After converting to float, = 50.35
After converting back to Currency, = $50.34
Now attempt to convert out of range value of -$100.00 to a Currency:
Result is $4294967246.60486

```

这个结果表示代码并没有像我们希望的那样工作。首先，从 `float` 转换回 `Currency` 得到一个错误的结果\$50.34，而不是\$50.35。其次，在试图转换明显超出范围的值时，没有生成异常。

第一个问题是由圆整错误引起的。如果类型转换用于把 `float` 转换为 `uint`，计算机就会截去多余的数字，而不是圆整它。计算机以二进制方式存储数字，而不是十进制，小数部分 0.35 不能用二进制小数来精确表示(像  $1/3$  这样的分数不能精确表示为小数，它应等于循环小数 0.3333)。所以，计算机最后存储了一个略小于 0.35 的值，它可以用二进制格式精确表示。把该数字乘以 100，就会得到一个小于 35 的数字，截去了 34 美分。显然在本例中，这种由截去引起的错误是很严重的，避免该错误的方式是确保在数字转换过程中执行智能圆整操作。Microsoft 编写了一个类 `System.Convert` 来完成该任务。`System.Convert` 包含大量的静态方法来

执行各种数字转换，我们需要使用的是 `Convert.ToInt16()`。注意，在使用 `System.Convert` 方法时会产生额外的性能损失，所以只应在需要时才使用它们。

下面看看为什么没有抛出期望的溢出异常。此处的问题是溢出异常实际发生的位置根本不在 `Main()` 例程中——它是在转换运算符的代码中发生的，该代码在 `Main()` 方法中调用，而且没有标记为 `checked`。

其解决方法是确保类型转换本身也在 `checked` 环境下进行。进行了这两个修改后，修订后的转换代码如下所示。

```
public static explicit operator Currency (float value)
{
    checked
    {
        uint dollars = (uint)value;
        ushort cents = Convert.ToInt16((value - dollars)*100);
        return new Currency(dollars, cents);
    }
}
```

注意，使用 `Convert.ToInt16()` 计算数字的美分部分，如上所示，但没有使用它计算数字的美元部分。在计算美元值时不需要使用 `System.Convert`，因为在此我们希望截去 `float` 值。

注意：

`System.Convert` 的方法还执行它自己的溢出检查。因此对于本例的情况，不需要把对 `Convert.ToInt16()` 的调用放在 `checked` 环境下。但把 `value` 显式转换为美元值仍需要 `checked` 环境。

这里没有给出这个新 `checked` 转换的结果，因为在本节后面还要对 `SimpleCurrency` 示例进行一些修改。

注意：

如果定义了一个使用非常频繁的数据类型转换，其性能也非常好，就可以不进行任何错误检查，如果对用户定义的转换和没有检查的错误进行了清晰的说明，这也是一种合法的解决方案。

### 1. 类之间的数据类型转换

`Currency` 示例仅涉及到与 `float` (一种预定义的数据类型) 来回转换的类。实际上任何简单数据类型的转换都是可以自定义的。定义不同结构或类之间的数据类型转换是允许的，但有两个限制：

- 如果某个类直接或间接继承了另一个类，就不能定义这两个类之间的数据类型转换(这些类型的类型转换已经存在)。
- 数据类型转换必须在源或目标数据类型的内部定义。

要说明这些要求，假定有如图 6-1 所示的类层次结构。

换言之，类 `C` 和 `D` 间接派生于 `A`。在这种情况下，在 `A`、`B`、`C` 或 `D` 之间唯一合法的类型转换就是类 `C` 和 `D` 之间的转换，因为这些类并没有互相派生。这段代码如下所示(假定希望数据类型转换是显式的，这是在用户定义的数据类型之间转换的

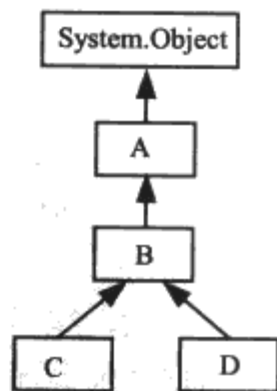


图 6-1

通常情况):

```
public static explicit operator D(C value)
{
    // and so on
}
public static explicit operator C(D value)
{
    // and so on
}
```

对于这些数据类型转换,可以选择放置定义的地方——在 C 的类定义内部,或者在 D 的类定义内部,但不能在其他地方定义。C#要求把数据类型转换的定义放在源类(或结构)或目标类(或结构)的内部。它的边界效应是不能定义两个类之间的数据类型转换,除非可以编辑它们的源代码。这是因为,这样可以防止第三方把数据类型转换引入类中。

一旦在一个类的内部定义了数据类型转换,就不能在另一个类中定义相同的数据类型转换。显然,只能有一个数据类型转换,否则编译器就不知道该选择哪个数据类型转换了。

## 2. 基类和派生类之间的数据类型转换

要了解这些数据类型转换是如何工作的,首先看看源和目标的数据类型都是引用类型的情况。考虑两个类 `MyBase` 和 `MyDerived`, 其中 `MyDerived` 直接或间接派生于 `MyBase`。

首先是从 `MyDerived` 到 `MyBase` 的转换,代码如下(假定可以使用构造函数):

```
MyDerived derivedObject = new MyDerived();
MyBase baseCopy = derivedObject;
```

在本例中,是从 `MyDerived` 隐式地转换为 `MyBase`。这是可行的,因为对类 `MyBase` 的任何引用都可以引用类 `MyBase` 的对象或派生于 `MyBase` 的对象。在 OO 编程中,派生类的实例实际上是基类的实例,但加上了一些额外的信息。在基类上定义的所有函数和字段也都在派生类上定义了。

下面看看另一种方式,编写下面的代码:

```
MyBase derivedObject = new MyDerived();
MyBase baseObject = new MyBase();
MyDerived derivedCopy1 = (MyDerived) derivedObject; // OK
MyDerived derivedCopy2 = (MyDerived) baseObject; // Throws exception
```

上面的代码都是合法的 C#代码(从句法的角度来看,是合法的),是把基类转换为派生类。但是,最后的一个语句在执行时会抛出一个异常。在进行数据类型转换时,会检查被引用的对象。因为基类引用实际上可以引用一个派生类实例,所以这个对象可能就是要转换的派生类的一个实例。如果是这样,转换就会成功,派生的引用被设置为引用这个对象。但如果该对象不是派生类(或者派生于这个类的其他类)的一个实例,转换就会失败,抛出一个异常。

注意,编译器已经提供了基类和派生类之间的转换,这种转换实际上并没有对对象进行任何数据转换。如果要进行的转换是合法的,它们也仅是把新引用设置为对对象的引用。这些转换在本质上与用户定义的转换不同。例如,在前面的 `SimpleCurrency` 示例中,我们定义了 `Currency` 结构和 `float` 之间的转换。在 `float` 到 `Currency` 的转换中,则实例化了一个新 `Currency` 结构,并用要求的值进行初始化。在基类和派生类之间的预定义转换则不是这样。如果要把

MyBase 实例转换为 MyDerived 对象，其值根据 MyBase 实例的内容来确定，就不能使用数据类型转换语法。最合适的选项通常是定义一个派生类的构造函数，它的参数是一个基类实例，让这个构造函数执行相关的初始化：

```
class DerivedClass : BaseClass
{
    public DerivedClass(BaseClass rhs)
    {
        // initialize object from the Base instance
    }
    // etc.
```

### 3. 装箱和拆箱数据类型转换

前面主要讨论了基类和派生类之间的数据类型转换，其中，基类和派生类都是引用类型。其规则也适用于转换值类型，但在转换值类型时，不是仅仅复制引用，还必须复制一些数据。

当然，不能从结构或基本值类型中派生。所以基本结构和派生结构之间的转换总是基本类型或结构与 System.Object 之间的转换(理论上可以在结构和 System.ValueType 之间进行转换，但一般很少这么做)。

从结构(或基本类型)到 object 的转换总是一种隐式转换，因为这种转换是从派生类型到基本类型的转换，即第2章中简要介绍的装箱过程。例如，Currency 结构：

```
Currency balance = new Currency(40,0);
object baseCopy = balance;
```

在执行上述隐式转换时，balance 的内容被复制到堆上，放在一个装箱的对象上，BaseCopy 对象引用设置为该对象。在后台发生的情况是：在最初定义 Currency 结构时，.NET Framework 隐式地提供另一个(隐式)类，即装箱的 Currency 类，它包含与 Currency 结构相同的所有字段，但却是一个引用类型，存储在堆上。无论这个值类型是一个结构，还是一个枚举，定义它时都存在类似的装箱引用类型，对应于所有的基本值类型，如 int、double 和 uint。不能也不必在源代码中直接编程访问这些装箱类型，但在把一个值类型转换为 object 时，它们是在后台工作的对象。在隐式地把 Currency 转换为 object 时，会实例化一个装箱的 Currency 实例，并用 Currency 结构中的所有数据进行初始化。在上面的代码中，BaseCopy 对象引用的就是这个已装箱的 Currency 实例。通过这种方式，就可以实现从派生类到基类的转换，并且，值类型的语法与引用类型的语法一样。

转换的另一种方式称为拆箱。与在基本引用类型和派生引用类型之间的转换一样，这是一种显式转换，因为如果要转换的对象不是正确的类型，会抛出一个异常：

```
object derivedObject = new Currency(40,0);
object baseObject = new object();
Currency derivedCopy1 = (Currency)derivedObject; // OK
Currency derivedCopy2 = (Currency)baseObject; // Exception thrown
```

上述代码的工作方式与前面的引用类型一样。把 derivedObject 转换为 Currency 会成功进行，因为 derivedObject 实际上引用的是装箱 Currency 实例——转换的过程是把已装箱的 Currency 对象的字段复制到一个新的 Currency 结构中。第二个转换会失败，因为 baseObject 没有引用已装箱的 Currency 对象。



在使用装箱和拆箱时，这两个过程都把数据复制到新装箱和拆箱的对象上，理解这一点是非常重要的。这样，对装箱对象的操作就不会影响原来值类型的内容。

### 6.5.2 多重数据类型转换

在定义数据类型转换时必须考虑的一个问题是，如果在进行要求的数据类型转换时，C#编译器没有可用的直接转换方式，C#编译器就会寻找一种方式，把几种转换合并起来。例如，在 Currency 结构中，假定编译器遇到下面的代码：

```
Currency balance = new Currency(10,50);
long amount = (long)balance;
double amountD = balance;
```

首先初始化一个 Currency 实例，再把它转换为一个 long。问题是不能定义这样的转换。但是，这段代码仍可以编译成功。因为编译器知道我们要定义一个从 Currency 到 float 的隐式转换，而且它知道如何显式地从 float 转换为 long。所以它会把这行代码编译为中间语言代码，首先把 balance 转换为 float，再把结果转换为 long。上述代码的最后一行也是这样，把 balance 转换为 double 型时，因为从 Currency 到 float 的转换和从 float 到 double 的转换都是隐式的，就可以在代码中把这个转换当作一种隐式转换。如果要显式地指定转换过程，可以编写如下代码：

```
Currency balance = new Currency(10,50);
long amount = (long)(float)balance;
double amountD = (double)(float)balance;
```

但是，在大多数情况下，这会使代码变得比较复杂，因此是不必要的。下面的代码会产生一个编译错误：

```
Currency balance = new Currency(10,50);
long amount = balance;
```

原因是编译器可以找到的最佳匹配的转换仍是首先转换为 float，再转换为 long，但从 float 到 long 的转换需要显式指定。

所有这些都带来太多的麻烦。转换的规则是非常直观的，主要是为了防止在开发人员不知情的情况下丢失数据。但是，在定义数据类型转换时如果不小心，编译器就有可能指定一条导致不期望的结果的路径。例如，假定编写 Currency 结构的其他小组成员要把一个 uint 转换为 Currency，而该 uint 中包含了美分的总数(美分不是美元，因为我们不希望丢掉美元的小数部分)，为此应编写如下代码：

```
public static implicit operator Currency (uint value)
{
    return new Currency(value/100u, (ushort)(value%100));
} // Don't do this!
```

注意，在这段代码中，第一个 100 后面的 u 可以确保把 value/100u 解释为 uint。如果写成 value/100，编译器就会把它解释为一个 int 型的值，而不是 uint 型的值。

在这段代码中清楚地注释了“不要这么做”。下面说明其原因。看看下面的代码段，它把包含 350 的 uint 转换为一个 Currency，再转换回 uint。那么在执行完这段代码后，bal2 中又将



包含什么?

```
uint bal = 350;
Currency balance = bal;
uint bal2 = (uint)balance;
```

答案不是 350, 而是 3! 这是符合逻辑的。我们把 350 隐式地转换为 Currency, 得到 balance.Dollars=3, balance.Cents=50。然后编译器进行通常的操作, 为转换回 uint 指定最佳路径。balance 最终会被隐式地转换为 float 型(其值为 3.5), 然后显式地转换为 uint 型, 其值为 3。

当然, 转换为另一个数据类型后, 再转换回来有时会丢失数据。例如, 把包含 5.8 的 float 转换为 int, 再转换回 float, 会丢失数字中的小数部分, 得到 5, 但丢失数字中的小数部分和一个整数被 100 整除的情况略有区别。Currency 现在成了一种相当危险的类, 它会对整数进行一些奇怪的操作。

问题是, 在转换过程中如何解释整数是有矛盾的。从 Currency 到 float 的转换会把整数 1 解释为 1 美元, 但从 uint 到 Currency 的转换会把这个整数解释为 1 美分, 这是很糟糕的。如果希望类易于使用, 就应确保所有的转换都按一种互相兼容的方式执行, 即这些转换应得到相同的结果。在本例中, 显然要重新编写从 uint 到 Currency 的转换, 把整数值 1 解释为 1 美元:

```
public static implicit operator Currency (uint value)
{
    return new Currency(value, 0);
}
```

偶尔也会觉得这种新的转换方式可能根本不需要。但实际上这种转换方式是非常有用的。没有它, 编译器在执行从 uint 到 Currency 的转换时, 就只能通过 float 来进行。此时直接转换的效率要高得多, 所以进行这种额外转换会提高性能, 但需要确保它的结果与通过 float 进行转换得到的结果相同。在其他情况下, 也可以为不同的预定义数据类型分别定义转换, 让更多的转换隐式执行, 而不是显式地执行, 但本例不是这样。

测试这种转换是否成功, 应确定无论使用什么转换路径, 它都能得到相同的结果(而不是像在从 float 到 int 的转换过程中丢失数据那样)。Currency 类就是一个很好的示例。看看下面的代码:

```
Currency balance = new Currency(50, 35);
ulong bal = (ulong) balance;
```

目前, 编译器只能采用一种方式来执行这个转换: 把 Currency 隐式地转换为 float, 再显式地转换为 ulong。从 float 到 ulong 的转换需要显式指定, 本例就显式指定了这个转换, 所以编译是成功的。

但假定要添加另一个转换, 从 Currency 隐式地转换为 uint, 就需要修改 Currency 结构, 添加从 uint 到 Currency 的转换和从 Currency 到 uint 的转换, 这段代码可以下载, 作为 SimpleCurrency2 示例:

```
public static implicit operator Currency (uint value)
{
    return new Currency(value, 0);
}
```

```
public static implicit operator uint (Currency value)
{
    return value.Dollars * 100 + value.Cents;
```

```
        return value.Dollars;
    }
}
```

现在, 编译器从 `Currency` 转换到 `ulong` 可以使用另一条路径: 先从 `Currency` 隐式地转换为 `uint`, 再隐式地转换为 `ulong`。该采用哪条路径? C# 有一些规则(本书不详细讨论这些规则, 读者可参阅 MSDN 文档说明), 告诉编译器如何确定哪条是最佳路径。但最好自己设计转换, 让所有的转换都得到相同的结果(但没有精确度的损失), 此时编译器选择哪条路径就不重要了(在本例中, 编译器会选择 `Currency`→`uint`→`ulong` 路径, 而不是 `Currency`→`float`→`ulong` 路径)。

为了测试 `SimpleCurrency2` 示例, 给 `SimpleCurrency` 的测试程序添加如下代码:

```
try
{
    Currency balance = new Currency(50,35);

    Console.WriteLine(balance);
    Console.WriteLine("balance is " + balance);
    Console.WriteLine("balance is (using ToString()) " + balance.ToString());

    uint balance3 = (uint) balance;
    Console.WriteLine("Converting to uint gives " + balance3);
}
```

运行这个示例, 得到如下所示的结果:

#### SimpleCurrency2

```
50
balance is $50.35
balance is (using ToString()) $50.35
Converting to uint gives 50
After converting to float, = 50.35
After converting back to Currency, = $50.34
Now attempt to convert out of range value of -$100.00 to a Currency:
Exception occurred: Arithmetic operation resulted in an overflow.
```

这个结果显示了到 `uint` 的转换是成功的, 但丢失了 `Currency` 的美分部分(小数部分), 把负的 `float` 转换为 `Currency` 也产生了预料中的溢出异常, 因为 `float` 到 `Currency` 的转换本身定义了一个 `checked` 环境。

但是, 这个输出结果也说明了进行转换时最后一个要注意的潜在问题: 结果的第一行没有正确显示结余, 显示了 50, 而不是 \$50.35。在下面的代码中:

```
Console.WriteLine(balance);
Console.WriteLine("balance is " + balance);
Console.WriteLine("balance is (using ToString()) " + balance.ToString());
```

只有最后两行把 `Currency` 正确显示为一个字符串。这是为什么? 问题是在把转换和方法重载合并起来时, 会出现另一个不希望的错误源。下面用倒序的方式解释这段代码。

第三行的 `Console.WriteLine()` 语句显式调用 `Currency.ToString()` 方法, 以确保 `Currency` 显示为一个字符串。第二行代码没有这么做。字符串 "balance is " 传送给 `Console.WriteLine()`, 告诉编译器这个参数应解释为字符串, 因此要隐式地调用 `Currency.ToString()` 方法。

但第一行的 `Console.WriteLine()` 方法只是把原来的 `Currency` 结构传送给 `Console.WriteLine()`。目前 `Console.WriteLine()` 有许多重载, 但它们的参数都不是 `Currency` 结构。所以编译器

会到处搜索，看看它能把 `Currency` 转换为什么，以与 `Console.WriteLine()` 的一个重载方法匹配。如上所示，`Console.WriteLine()` 的一个重载方法可以快速而高效地显示 `uint`，且其参数是一个 `uint`。因此应把 `Currency` 隐式地转换为 `uint`。

实际上，`Console.WriteLine()` 有另一个重载方法，它的参数是一个 `double`，结果是显示该 `double` 的值。如果仔细看看第一个 `SimpleCurrency` 示例的结果，就会发现该结果的第一行就是使用这个重载方法把 `Currency` 显示为一个 `double`。在这个示例中，没有直接把 `Currency` 转换为 `uint`，所以编译器选择 `Currency`→`float`→`double` 作为可用于 `Console.WriteLine()` 重载方法的首选转换方式。但在 `SimpleCurrency2` 中可以直接转换为 `uint`，所以编译器会选择后者。

如果方法调用带有多个重载方法，并要给该方法传送参数，而该参数的数据类型不匹配任何重载方法，就可以迫使编译器确定使用哪些转换方式进行数据转换，决定使用哪个重载方法(并进行相应的数据转换)。当然，编译器总是按逻辑和严格的规则来工作，但结果可能并不是我们所期望的。如果可能会出问题，最好显式指定转换路径。

## 6.6 小结

本章介绍了 C# 提供的标准运算符，描述了对等的相等机制，讨论了编译器如何把一种标准数据类型转换为另一种标准数据类型。还阐述了如何使用运算符重载在自己的数据类型上执行定制的运算符。最后，学习了运算符重载的一种特殊类型，即数据类型转换运算符，它允许用户指定如何将定制类型的实例转换为其他数据类型。

第 7 章将介绍两个密切相关的成员类型：委托和事件，在自己的类型上也可以实现这两个成员类型，以支持基于事件的对象模型。

# 第 7 章

## 委托和事件

回调(callback)函数是 Windows 编程的一个重要部分。如果您具备 C 或 C++ 编程背景,应该就曾在许多 Windows API 中使用过回调。Visual Basic 添加了 AddressOf 关键字后,开发人员就可以利用以前一度受到限制的 API 了。回调函数实际上是方法调用的指针,也称为函数指针,是一个非常强大的编程特性。.NET 以委托的形式实现了函数指针的概念。它们的特殊之处是,与 C 函数指针不同,.NET 委托是类型安全的。这说明,C 中的函数指针只不过是一个指向存储单元的指针,我们无法说出这个指针实际指向什么,像参数和返回类型等就更无从知晓了。如本章所述,.NET 把委托作为一种类型安全的操作。本章后面将学习 .NET 如何将委托用作实现事件的方式。

本章的主要内容如下:

- 委托
- 匿名方法
- $\lambda$ 表达式
- 事件

### 7.1 委托

当要把方法传送给其他方法时,需要使用委托。要了解它们的含义,可以看看下面的代码:

```
int i = int.Parse("99");
```

我们习惯于把数据作为参数传递给方法,如上面的例子所示。所以,给方法传送另一个方法听起来有点奇怪。而有时某个方法执行的操作并不是针对数据进行的,而是要对另一个方法进行操作,这就比较复杂了。在编译时我们不知道第二个方法是什么,这个信息只能在运行时得到,所以需要把第二个方法作为参数传递给第一个方法,这听起来很令人迷惑,下面用几个示例来说明:

- 启动线程——在 C# 中,可以告诉计算机并行运行某些新的执行序列。这种序列就称为线程,在基类 `System.Threading.Thread` 的一个实例上使用方法 `Start()`,就可以开始执行一个线程。如果要告诉计算机开始一个新的执行序列,就必须说明要在哪里执行该序列。必须为计算机提供开始执行的方法的细节,即 `Thread` 类的构造函数必须带有一个参数,该参数定义了要由线程调用的方法。

- 通用库类——有许多库包含执行各种标准任务的代码。这些库通常可以自我包含。这样在编写库时，就会知道任务该如何执行。但是有时在任务中还包含子任务，只有使用该库的客户机代码才知道如何执行这些子任务。例如编写一个类，它带有一个对象数组，并把它们按升序排列。但是，排序的部分过程会涉及到重复使用数组中的两个对象，比较它们，看看哪一个应放在前面。如果要编写的类必须能给任何对象数组排序，就无法提前告诉计算机应如何比较对象。处理类中对象数组的客户机代码也必须告诉类如何比较要排序的对象。换言之，客户机代码必须给类传递某个可以进行这种比较的合适方法的细节。
- 事件——一般是通知代码发生了什么事。GUI 编程主要是处理事件。在发生事件时，运行库需要知道应执行哪个方法。这就需把处理事件的方法传送为委托的一个参数。这些将在本章后面讨论。

在 C 和 C++ 中，只能提取函数的地址，并传送为一个参数。C 是没有类型安全性的。可以把任何函数传送给需要函数指针的方法。这种直接的方法会导致一些问题，例如类型的安全性，在进行面向对象编程时，方法很少是孤立存在的，在调用前，通常需与类实例相关联。而这种方法并没有考虑到这个问题。所以 .NET Framework 在语法上不允许使用这种直接的方法。如果要传递方法，就必须把方法的细节封装在一种新类型的对象中，即委托。委托只是一种特殊的对象类型，其特殊之处在于，我们以前定义的所有对象都包含数据，而委托包含的只是方法的地址。

### 7.1.1 在 C# 中声明委托

在 C# 中使用一个类时，分两个阶段。首先需要定义这个类，即告诉编译器这个类由什么字段和方法组成。然后(除非只使用静态方法)实例化类的一个对象。使用委托时，也需要经过这两个步骤。首先定义要使用的委托，对于委托，定义它就是告诉编译器这种类型的委托代表了哪种类型的方法，然后创建该委托的一个或多个实例。编译器在后台将创建表示该委托的一个类。

定义委托的语法如下：

```
delegate void IntMethodInvoker(int x);
```

在这个示例中，定义了一个委托 `IntMethodInvoker`，并指定该委托的每个实例都包含一个方法的细节，该方法带有一个 `int` 参数，并返回 `void`。理解委托的一个要点是它们的类型安全性非常高。在定义委托时，必须给出它所代表的方法签名和返回类型等全部细节。

**提示：**

理解委托的一种好方式是把委托当作给方法签名和返回类型指定名称。

假定要定义一个委托 `TwoLongsOp`，该委托代表的方法有两个 `long` 型参数，返回类型为 `double`。可以编写如下代码：

```
delegate double TwoLongsOp(long first, long second);
```

或者定义一个委托，它代表的方法不带参数，返回一个 `string` 型的值，则可以编写如下代码：

```
delegate string GetAString();
```



其语法类似于方法的定义，但没有方法体，定义的前面要加上关键字 `delegate`。因为定义委托基本上就是定义一个新类，所以可以在定义类的任何地方定义委托，既可以在另一个类的内部定义，也可以在任何类的外部定义，还可以在命名空间中把委托定义为顶层对象。根据定义的可见性，可以在委托定义上添加一般的访问修饰符：`public`、`private`、`protected` 等：

```
public delegate string GetAString();
```

注意：

实际上，“定义一个委托”是指“定义一个新类”。委托实现为派生自基类 `System.MulticastDelegate` 的类，`System.MulticastDelegate` 又派生自基类 `System.Delegate`。C#编译器知道这个类，会使用其委托语法，因此我们不需要了解这个类的具体执行情况，这是 C#与基类共同合作，使编程更易完成的另一个示例。

定义好委托后，就可以创建它的一个实例，以存储特定方法的细节。

注意：

此处，在术语方面有一个问题。类有两个不同的术语：“类”表示较广义的定义，“对象”表示类的实例。但委托只有一个术语。在创建委托的实例时，所创建的委托的实例仍称为委托。必须从上下文中确定委托的确切含义。

### 7.1.2 在 C#中使用委托

下面的代码段说明了如何使用委托。这是在 `int` 上调用 `ToString()` 方法的一种相当冗长的方式：

```
private delegate string GetAString();

static void Main()
{
    int x = 40;
    GetAString firstStringMethod = new GetAString(x.ToString());
    Console.WriteLine("String is {0}" + firstStringMethod());
    // With firstStringMethod initialized to x.ToString(),
    // the above statement is equivalent to saying
    // Console.WriteLine("String is {0}" + x.ToString()); } }
```

在这段代码中，实例化了类型为 `GetAString` 的一个委托，并对它进行初始化，使它引用整型变量 `x` 的 `ToString()` 方法。在 C#中，委托在语法上总是带有一个参数的构造函数，这个参数就是委托引用的方法。这个方法必须匹配最初定义委托时的签名。所以在这个示例中，如果用不带参数、返回一个字符串的方法来初始化 `firstStringMethod` 变量，就会产生一个编译错误。注意，`int.ToString()` 是一个实例方法（不是静态方法），所以需要指定实例 (`x`) 和方法名来正确初始化委托。

下一行代码使用这个委托来显示字符串。在任何代码中，都应提供委托实例的名称，后面的括号中应包含调用该委托中的方法时使用的参数。所以在上面的代码中，`Console.WriteLine()` 语句完全等价于注释语句中的代码行。

实际上，给委托实例提供括号与调用委托类的 `Invoke()` 方法完全相同。`firstStringMethod` 是委托类型的一个变量，所以 C#编译器会用 `firstStringMethod.Invoke()` 代替 `firstStringMethod()`。

```
firstStringMethod();
```

```
firstStringMethod. Invoke();
```

C# 2.0 使用委托推断扩展了委托的语法。为了减少输入量，只需要委托实例，就可以只传送地址的名称。这称为委托推断。只要编译器可以把委托实例解析为特定的类型，这个 C# 特性就是有效的。下面的示例用 GetString 委托的一个新实例初始化了 GetString 类型的变量 firstStringMethod:

```
GetString firstStringMethod = new GetString(x.ToString);
```

只要用变量 x 把方法名传送给变量 firstStringMethod，就可以编写出作用相同的代码:

```
GetString firstStringMethod = x.ToString;
```

C#编译器创建的代码是一样的。编译器会用 firstStringMethod 检测需要的委托类型，因此创建 GetString 委托类型的一个实例，用对象 x 把方法的地址传送给构造函数。

#### 注意:

不能调用 x.ToString()方法，把它传送给委托变量。调用 x.ToString()方法会返回一个不能赋予委托变量的字符串对象。只能把方法的地址赋予委托变量。

委托推断可以在需要委托实例的任何地方使用。委托推断也可以用于事件，因为事件基于委托(参见本章后面的内容)。

委托的一个特征是它们的类型是安全的，可以确保被调用的方法签名是正确的。但有趣的是，它们不关心在什么类型的对象上调用该方法，甚至不考虑该方法是静态方法，还是实例方法。

#### 提示:

给定委托的实例可以表示任何类型的任何对象上的实例方法或静态方法——只要方法的签名匹配于委托的签名即可。

为了说明这一点，我们扩展上面的代码，让它使用 firstStringMethod 委托在另一个对象上调用其他两个方法，其中一个是实例方法，另一个是静态方法。为此，再次使用本章前面定义的 Currency 结构。Currency 结构有自己的 ToString()重载方法和一个与 GetCurrencyUnit()的签名相同的静态方法，这样，就可以用同一个委托变量调用这些方法了:

```
struct Currency
{
    public uint Dollars;
    public ushort Cents;

    public Currency(uint dollars, ushort cents)
    {
        this.Dollars = dollars;
        this.Cents = cents;
    }
    public override string ToString()
    {
        return string.Format("${0}.{1,-2:00}", Dollars, Cents);
    }

    public static string GetCurrencyUnit()
    {

```

```

        return "Dollar";
    }

    public static explicit operator Currency (float value)
    {
        checked
        {
            uint dollars =(uint)value;
            ushort cents =(ushort)((value-dollars)*100);
            return new Currency(dollars,cents);
        }
    }

    public static implicit operator float (Currency value)
    {
        return value.Dollars + (value.Cents/100.0f);
    }

    public static implicit operator Currency (uint value)
    {
        return new Currency(value, 0);
    }

    public static implicit operator uint (Currency value)
    {
        return value.Dollars;
    }
}

```

下面就可以使用 `GetString` 实例，代码如下所示：

```

private delegate string GetString();

static void Main()
{
    int x = 40;
    GetString firstStringMethod = x.ToString;
    Console.WriteLine("String is {0}" + firstStringMethod());

    Currency balance = new Currency(34, 50);
    // firstStringMethod references an instance method
    firstStringMethod = balance.ToString;
    Console.WriteLine("String is {0}" + firstStringMethod());

    // firstStringMethod references a static method
    firstStringMethod = new GetString(Currency.GetCurrencyUnit);
    Console.WriteLine("String is {0}" + firstStringMethod());
}

```

这段代码说明了如何通过委托来调用方法，然后重新给委托指定在类的不同实例上执行的不同方法，甚至可以指定静态方法，或者在类的不同类型的实例上执行的方法，只要每个方法的签名匹配委托定义即可。

运行应用程序，会得到委托引用的不同方法的结果：

```

String is 40
String is $34.50
String is Dollar

```

但是，我们还没有说明把一个委托传递给另一个方法的具体过程，也没有给出任何有用的结果。调用 `int` 和 `Currency` 对象的 `ToString()` 的方法要比使用委托直观得多！在真正领会到委托

的用处前，需要用一個相當複雜的示例來說明委託的本質。下面就是兩個委託的示例。第一個示例僅使用委託來調用兩個不同的操作，說明了如何把委託傳遞給方法，如何使用委託數組，但這仍沒有很好地說明：沒有委託，就不能完成很多工作。第二個示例就複雜得多了，它有一個類 `BubbleSorter`，執行一個方法，按照升序排列一個對象數組，這個類沒有委託是很難編寫出來的。

### 7.1.3 簡單的委託示例

在這個示例中，定義一個類 `MathsOperations`，它有一個靜態方法，對 `double` 類型的值執行兩個操作，然後使用該委託調用這些方法。這個數學類如下所示：

```
class MathsOperations
{
    public static double MultiplyByTwo(double value)
    {
        return value*2;
    }

    public static double Square(double value)
    {
        return value*value;
    }
}
```

下面調用這些方法：

```
using System;

namespace Wrox.ProCSharp.Delegates
{
    delegate double DoubleOp(double x);

    class Program
    {
        static void Main()
        {
            DoubleOp [] operations =
            {
                MathsOperations.MultiplyByTwo,
                MathsOperations.Square
            };

            for (int i=0 ; i<operations.Length ; i++)
            {
                Console.WriteLine("Using operations[{0}]:", i);
                ProcessAndDisplayNumber(operations[i], 2.0);
                ProcessAndDisplayNumber(operations[i], 7.94);
                ProcessAndDisplayNumber(operations[i], 1.414);
                Console.WriteLine();
            }
        }

        static void ProcessAndDisplayNumber(DoubleOp action, double value)
        {
            double result = action(value);
            Console.WriteLine(
                "Value is {0}, result of operation is {1}", value, result);
        }
    }
}
```



在这段代码中，实例化了一个委托数组 `DoubleOp` (记住，一旦定义了委托类，就可以实例化它的实例，就像处理一般的类那样——所以把一些委托的实例放在数组中是可以的)。该数组的每个元素都初始化为由 `MathsOperations` 类执行的不同操作。然后循环这个数组，把每个操作应用到 3 个不同的值上。这说明了使用委托的一种方式——把方法组合到一个数组中，这样就可以在循环中调用不同的方法了。

这段代码的关键一行是把委托传递给 `ProcessAndDisplayNumber()` 方法，例如：

```
ProcessAndDisplayNumber(operations[i], 2.0);
```

其中传递了委托名，但不带任何参数，假定 `operations[i]` 是一个委托，其语法是：

- `operations[i]` 表示“这个委托”。换言之，就是委托代表的方法。
- `operations[i](2.0)` 表示“调用这个方法，参数放在括号中”。

`ProcessAndDisplayNumber()` 方法定义为把一个委托作为其第一个参数：

```
static void ProcessAndDisplayNumber(DoubleOp action, double value)
```

在这个方法中，调用：

```
double result = action(value);
```

这实际上是调用 `action` 委托实例封装的方法，其返回结果存储在 `result` 中。

运行这个示例，得到如下所示的结果：

```
SimpleDelegate
Using operations[0]:
Value is 2, result of operation is 4
Value is 7.94, result of operation is 15.88
Value is 1.414, result of operation is 2.828

Using operations[1]:
Value is 2, result of operation is 4
Value is 7.94, result of operation is 63.0436
Value is 1.414, result of operation is 1.999396
```

#### 7.1.4 BubbleSorter 示例

下面的示例将说明委托的用途。我们要编写一个类 `BubbleSorter`，它执行一个静态方法 `Sort()`，这个方法第一个参数是一个对象数组，把该数组按照升序重新排列。换言之，假定传递的是 `int` 数组：`{0, 5, 6, 2, 1}`，则返回的结果应是 `{0, 1, 2, 5, 6}`。

冒泡排序算法非常著名，是一种排序的简单方法。它适合于小组数字，因为对于大量的数字(超过 10 个)，还有更高效的算法。冒泡排序算法重复遍历数组，比较每一对数字，按照需要交换它们的位置，把最大的数字逐步移动到数组的最后。对于给 `int` 排序，进行冒泡排序的方法如下所示：

```
for (int i = 0; i < sortArray.Length; i++)
{
```



```

for (int j = i + 1; j < sortArray.Length; j++)
{
    if (sortArray[j] < sortArray[i]) // problem with this test
    {
        int temp = sortArray[i]; // swap ith and jth entries
        sortArray[i] = sortArray[j];
        sortArray[j] = temp;
    }
}
}

```

它非常适合于 `int`，但我们希望 `Sort()` 方法能给任何对象排序。换言之，如果某段客户机代码包含 `Currency` 结构数组或其他类和结构，就需要对该数组排序。这样，上面代码中的 `if(sortArray[j] < sortArray[i])` 就有问题了，因为它需要比较数组中的两个对象，看看哪一个更大。可以对 `int` 进行这样的比较，但如何对直到运行才知道或确定的新类进行比较？答案是客户机代码知道类在委托中传递的是什么方法，封装这个方法就可以进行比较。

定义如下的委托：

```
delegate bool Comparison(object x, object y);
```

给 `Sort` 方法指定下述签名：

```
static public void Sort(object [] sortArray, Comparison comparison)
```

这个方法的文档说明强调，`comparison` 必须表示一个静态方法，该方法带有两个参数，如果第二个参数的值“大于”第一个参数(换言之，它应放在数组中靠后的位置)，就返回 `true`。

设置完毕后，下面定义类 `BubbleSorter`：

```

class BubbleSorter
{
    static public void Sort(object [] sortArray, Comparison comparison)
    {
        for (int i=0 ; i<sortArray.Length ; i++)
        {
            for (int j=i+1 ; j<sortArray.Length ; j++)
            {
                if (comparison(sortArray[j], sortArray[i]))
                {
                    object temp = sortArray[i];
                    sortArray[i] = sortArray[j];
                    sortArray[j] = temp;
                }
            }
        }
    }
}

```

为了使用这个类，需要定义另一个类，建立要排序的数组。在本例中，假定 `Mortimer Phones` 移动电话公司有一个员工列表，要对照他们的薪水进行排序。每个员工分别由类 `Employee` 的一个实例表示，如下所示：

```

class Employee
{
    private string name;
    private decimal salary;
}

```

```

public Employee(string name, decimal salary)
{
    this.name = name;
    this.salary = salary;
}

public override string ToString()
{
    return string.Format("{0}, {1:C}", name, salary);
}

public static bool CompareSalary(object x, object y)
{
    Employee e1 = (Employee) x;
    Employee e2 = (Employee) y;
    return (e1.salary < e2.salary);
}
}

```

注意，为了匹配 `Comparison` 委托的签名，在这个类中必须定义 `CompareSalary`，它的参数是两个对象引用，而不是 `Employee` 引用。必须把这些参数的数据类型强制转换为 `Employee` 引用，才能进行比较。

#### 注意：

这里除了把对象用作参数之外，还可以使用强类型化的泛型。第 9 章介绍了泛型和泛型委托。

下面编写一些客户端代码，完成排序：

```

using System;

namespace Wrox.ProCSharp.Delegates
{
    delegate bool Comparison(object x, object y);

    class Program
    {
        static void Main()
        {
            Employee [] employees =
            {
                new Employee("Bugs Bunny", 20000),
                new Employee("Elmer Fudd ", 10000),
                new Employee("Daffy Duck", 25000),
                new Employee("Wiley Coyote", (decimal)1000000.38),
                new Employee("Foghorn Leghorn", 23000),
                new Employee("RoadRunner", 50000));

            BubbleSorter.Sort(employees, Employee.CompareSalary);

            foreach (var employee in employees)
            {
                Console.WriteLine(employee);
            }
        }
    }
}

```

运行这段代码，正确显示按照薪水排列的 Employee，如下所示：

```
BubbleSorter
Elmer Fudd, $10,000.00
Bugs Bunny, $20,000.00
Foghorn Leghorn, $23,000.00
Daffy Duck, $25,000.00
RoadRunner, $50,000.00
Wiley Coyote, $1,000,000.38
```

### 7.1.5 多播委托

前面使用的每个委托都只包含一个方法调用。调用委托的次数与调用方法的次数相同。如果要调用多个方法，就需要多次显式调用这个委托。委托也可以包含多个方法。这种委托称为多播委托。如果调用多播委托，就可以按顺序连续调用多个方法。为此，委托的签名就必须返回 void；否则，就只能得到委托调用的最后一个方法的结果。

下面的代码取自于 SimpleDelegate 示例。尽管其语法与以前相同，但实际上它实例化了一个多播委托 Operations：

```
delegate void DoubleOp(double value);
// delegate double DoubleOp(double value); // can't do this now

class MainEntryPoint
{
    static void Main()
    {
        DoubleOp operations = MathOperations.MultiplyByTwo;
        operations += MathOperations.Square;
```

在前面的示例中，要存储对两个方法的引用，所以实例化了一个委托数组。而这里只是在一个多播委托中添加两个操作。多播委托可以识别运算符+和+=。还可以扩展上述代码中的最后两行，它们具有相同的效果：

```
DoubleOp operation1 = MathOperations.MultiplyByTwo;
DoubleOp operation2 = MathOperations.Square;
DoubleOp operations = operation1 + operation2;
```

多播委托还识别运算符-和-=，以从委托中删除方法调用。

**注意：**

根据后面的内容，多播委托是一个派生于 System.MulticastDelegate 的类，System.MulticastDelegate 又派生于基类 System.Delegate。System.MulticastDelegate 的其他成员允许把多个方法调用链接在一起，成为一个列表。

为了说明多播委托的用法，下面把 SimpleDelegate 示例改写为一个新示例 MulticastDelegate。现在需要把委托表示为返回 void 的方法，就应重写 MathOperations 类中的方法，让它们显示其结果，而不是返回它们：

```
class MathOperations
{
    public static void MultiplyByTwo(double value)
```

```

        double result = value*2;
        Console.WriteLine(
            "Multiplying by 2: {0} gives {1}", value, result);
    }

    public static void Square(double value)
    {
        double result = value*value;
        Console.WriteLine("Squaring: {0} gives {1}", value, result);
    }
}

```

为了适应这个改变，也必须重写 `ProcessAndDisplayNumber`：

```

static void ProcessAndDisplayNumber(DoubleOp action, double valueToProcess)
{
    Console.WriteLine();
    Console.WriteLine("ProcessAndDisplayNumber called with value = {0}"
        valueToProcess);
    action(valueToProcess);
}

```

下面测试多播委托，其代码如下：

```

static void Main()
{
    DoubleOp operations = MathOperations.MultiplyByTwo;
    operations += MathOperations.Square;

    ProcessAndDisplayNumber(operations, 2.0);
    ProcessAndDisplayNumber(operations, 7.94);
    ProcessAndDisplayNumber(operations, 1.414);
    Console.WriteLine();
}

```

现在，每次调用 `ProcessAndDisplayNumber` 时，都会显示一个信息，说明它已经被调用。然后，下面的语句会按顺序调用 `action` 委托实例中的每个方法：

```
action(value);
```

运行这段代码，得到如下所示的结果：

MulticastDelegate

```

ProcessAndDisplayNumber called with value = 2
Multiplying by 2: 2 gives 4
Squaring: 2 gives 4

```

```

ProcessAndDisplayNumber called with value = 7.94
Multiplying by 2: 7.94 gives 15.88
Squaring: 7.94 gives 63.0436

```

```

ProcessAndDisplayNumber called with value = 1.414
Multiplying by 2: 1.414 gives 2.828
Squaring: 1.414 gives 1.999396

```

如果使用多播委托，就应注意对同一个委托调用方法链的顺序并未正式定义，因此应避免编写依赖于以特定顺序调用方法的代码。



通过一个委托调用多个方法还有一个大问题。多播委托包含一个逐个调用的委托集合。如果通过委托调用的一个方法抛出了异常，整个迭代就会停止。下面是 `MulticastIteration` 示例。其中定义了一个简单的委托 `DemoDelegate`，它没有参数，返回 `void`。这个委托调用方法 `One()` 和 `Two()`，这两个方法满足委托的参数和返回类型要求。注意方法 `One()` 抛出了一个异常：

```
using System;

namespace Wrox.ProCSharp.Delegates
{
    public delegate void DemoDelegate();

    class Program
    {
        static void One()
        {
            Console.WriteLine("One");
            throw new Exception("Error in one");
        }

        static void Two()
        {
            Console.WriteLine("Two");
        }
    }
}
```

在 `Main()` 方法中，创建了委托 `d1`，它引用方法 `One()`，接着把 `Two()` 方法的地址添加到同一个委托中。调用 `d1` 委托，就可以调用这两个方法。异常在 `try/catch` 块中捕获：

```
static void Main()
{
    DemoDelegate d1 = One;
    d1 += Two;

    try
    {
        d1();
    }
    catch (Exception)
    {
        Console.WriteLine("Exception caught");
    }
}
```

委托只调用了第一个方法。第一个方法抛出了异常，所以委托的迭代会停止，不再调用 `Two()` 方法。当调用方法的顺序没有指定时，结果会有所不同。

```
One
Exception Caught
```

**注意：**

错误和异常详见第 14 章。

在这种情况下，为了避免这个问题，应手动迭代方法列表。`Delegate` 类定义了方法 `GetInvocationList()`，它返回一个 `Delegate` 对象数组。现在可以使用这个委托调用与委托直接相关的方法，捕获异常，并继续下一次迭代。



```

static void Main()
{
    DemoDelegate d1 = One;
    d1 += Two;

    Delegate[] delegates = d1.GetInvocationList();
    foreach (DemoDelegate d in delegates)
    {
        try
        {
            d();
        }
        catch (Exception)
        {
            Console.WriteLine("Exception caught");
        }
    }
}

```

修改了代码后运行应用程序，会看到在捕获了异常后，将继续迭代下一个方法。

```

One
Exception caught
Two

```

### 7.1.6 匿名方法

到目前为止，要想使委托工作，方法必须已经存在(即委托是用方法的签名定义的)。但使用委托还有另外一种方式：即通过匿名方法。匿名方法是用作委托参数的一个代码块。

用匿名方法定义委托的语法与前面的定义并没有区别。但在实例化委托时，就有区别了。下面是一个非常简单的控制台应用程序，说明了如何使用匿名方法：

```

using System;

namespace Wrox.ProCSharp.Delegates
{
    class Program
    {
        delegate string DelegateTest(string val);

        static void Main()
        {
            string mid = ", middle part,";

            delegateTest anonDel = delegate(string param)
            {
                param += mid;
                param += " and this was added to the string.";
                return param;
            };

            Console.WriteLine(anonDel("Start of string"));
        }
    }
}

```

委托 `DelegateTest` 在类 `Program` 中定义，它带一个字符串参数。有区别的是 `Main` 方法。在定义 `anonDel` 时，不是传送已知的方法名，而是使用一个简单的代码块：它前面是关键字 `delegate`，后面是一个参数：

```
delegate(string param)
{
    param += mid;
    param += " and this was added to the string.";
    return param;
};
```

可以看出，该代码块使用方法级的字符串变量 `mid`，该变量是在匿名方法的外部定义的，并添加到要传送的参数中。接着代码返回该字符串值。在调用委托时，把一个字符串传送为参数，将返回的字符串输出到控制台上。

匿名方法的优点是减少了要编写的代码。不必定义仅由委托使用的方法。在为事件定义委托时，这是非常显然的。(本章后面探讨事件。)这有助于降低代码的复杂性，尤其是定义了好几个事件时，代码会显得比较简单。使用匿名方法时，代码执行得不太快。编译器仍定义了一个方法，该方法只有一个自动指定的名称，我们不需要知道这个名称。

在使用匿名方法时，必须遵循两个规则。在匿名方法中不能使用跳转语句跳到该匿名方法的外部，反之亦然：匿名方法外部的跳转语句不能跳到该匿名方法的内部。

在匿名方法内部不能访问不安全的代码。另外，也不能访问在匿名方法外部使用的 `ref` 和 `out` 参数。但可以使用在匿名方法外部定义的其他变量。

如果需要用匿名方法多次编写同一个功能，就不要使用匿名方法。而编写一个指定的方法比较好，因为该方法只需编写一次，以后可通过名称引用它。

### 7.1.7 λ表达式

C# 3.0 为匿名方法提供了一个新的语法：λ表达式。λ表达式可以用于委托类型。前面使用匿名方法的例子可以改为使用λ表达式：

```
using System;

namespace Wrox.ProCSharp.Delegates
{
    class Program
    {
        delegate string DelegateTest(string val);

        static void Main()
        {
            string mid = ", middle part,";

            DelegateTest anonDel = param =>
            {
                param += mid;
                param += " and this was added to the string.";
                return param;
            };

            Console.WriteLine(anonDel("Start of string"));
        }
    }
}
```

```
    }
}
```

$\lambda$ 运算符 $\Rightarrow$ 的左边列出了匿名方法需要的参数。这有几种编写方式。例如，如果需要在示例代码中把一个字符串参数定义为委托类型，一种方式是在括号中定义类型和变量名：

```
(string param)
```

在 $\lambda$ 表达式中，不需要给声明添加变量类型，因为编译器知道该类型：

```
(param)
```

如果只有一个参数，就可以删除括号：

```
param
```

$\lambda$ 表达式的右边列出了实现代码。在示例程序中，实现代码放在花括号中，类似于前面的匿名方法：

```
{
    param += mid;
    param += " and this was added to the string.";
    return param;
};
```

如果实现代码只有一行，也可以删除花括号和 `return` 语句，因为编译器会自动添加该语句。例如，在下面的委托中，需要一个 `int` 参数，返回一个 `bool` 值：

```
public delegate bool Predicate(int obj)
```

可以声明一个委托变量，并指定一个 $\lambda$ 表达式。在 $\lambda$ 表达式中，左边定义了变量 `x`。这个变量的类型自动设置为 `int`，因为这是通过委托定义的。实现代码返回比较 `x > 5` 的布尔结果。如果 `x` 大于 5，就返回 `true`，否则返回 `false`。

```
Predicate p1 = x => x > 5;
```

可以把这个 $\lambda$ 表达式传送给需要谓词参数的方法：

```
list.FindAll(x => x > 5);
```

这里列出了相同的 $\lambda$ 表达式，但把变量 `x` 的类型定义为 `int`，没有使用变量类型推断功能，还在实现代码中添加了 `return` 语句：

```
list.FindAll(int x) => { return x > 5; };
```

如果使用以前的语法，可以通过匿名方法完成相同的功能：

```
list.FindAll(delegate(int x) { return x > 5; });
```

通过所有这些改变，C#编译器就创建出了相同的 IL 代码。

修改前面的 `SimpleDelegate` 示例，使用 $\lambda$ 表达式，可以删除类 `MathOperations`。 `Main()`方法现在如下所示：

```
static void Main()
{
    DoubleOp multByTwo = val => val * 2;
    DoubleOp square = val => val * val;

    DoubleOp [] operations = {multByTwo, square};
```

```

for (int i=0 ; i < operations.Length ; i++)
{
    Console.WriteLine("Using operations[{0}]:", i);
    ProcessAndDisplayNumber(operations[i], 2.0);
    ProcessAndDisplayNumber(operations[i], 7.94);
    ProcessAndDisplayNumber(operations[i], 1.414);
    Console.WriteLine();
}
}

```

运行这个版本，可以得到与上例相同的结果。但优点是它删除了类。

#### 提示：

λ表达式可以用于委托是类型的任意地方。类型是 `Expression` 或 `Expression<T>` 时，也可以使用λ表达式。此时编译器会创建一个表达式树，详见第11章。

### 7.1.8 协变和抗变

委托调用的方法不需要与委托声明定义的类型相同。因此可能出现协变和抗变。

#### 1. 返回类型协变

方法的返回类型可以派生于委托定义的类型。在下面的示例中，委托 `MyDelegate` 定义为返回 `DelegateReturn` 类型。赋予委托实例 `d1` 的方法返回 `DelegateReturn2` 类型，`DelegateReturn2` 派生自 `DelegateReturn`，因此满足了委托的需求。这称为返回类型协变。

```

public class DelegateReturn
{
}

public class DelegateReturn2 : DelegateReturn
{
}

public delegate DelegateReturn MyDelegate1();

class Program
{
    static void Main()
    {
        MyDelegate1 d1 = Method1;
        d1();
    }

    static DelegateReturn2 Method1()
    {
        DelegateReturn2 d2 = new DelegateReturn2();
        return d2;
    }
}

```

#### 2. 参数类型抗变

术语“参数类型抗变”表示，委托定义参数可能不同于委托调用的方法。这里是返回类

型不同，因为方法使用的参数类型可能派生自委托定义的类型。在下面的示例代码中，委托使用的参数类型是 `DelegateParam2`，而赋予委托实例 `d2` 的方法使用的参数类型是 `DelegateParam`，`DelegateParam` 是 `DelegateParam2` 的基类。

```
public class DelegateParam
{
}

public class DelegateParam2 : DelegateParam
{
}

public delegate void MyDelegate2(DelegateParam2 p);

class Program
{
    static void Main()
    {
        MyDelegate2 d2 = Method2;
        DelegateParam2 p = new DelegateParam2();
        d2(p);
    }

    static void Method2(DelegateParam p)
    {
    }
}
```

## 7.2 事件

基于 Windows 的应用程序也是基于消息的。这说明，应用程序是通过 Windows 来通信的，Windows 又是使用预定义的消息与应用程序通信的。这些消息是包含各种信息的结构，应用程序和 Windows 使用这些信息决定下一步的操作。在 MFC 等库或 Visual Basic 等开发环境推出之前，开发人员必须处理 Windows 发送给应用程序的消息。Visual Basic 和今天的 .NET 把这些传送来的消息封装在事件中。如果需要响应某个消息，就应处理对应的事件。一个常见的例子是用户单击了窗体中的按钮后，Windows 就会给按钮消息处理程序(有时称为 Windows 过程或 `WndProc`)发送一个 `WM_MOUSECLICK` 消息。对于 .NET 开发人员来说，这就是按钮的 `Click` 事件。

在开发基于对象的应用程序时，需要使用另一种对象通信方式。在一个对象中发生了有趣的事情时，就需要通知其他对象发生了什么变化。这里又要用到事件。就像 .NET Framework 把 Windows 消息封装在事件中那样，也可以把事件用作对象之间的通信介质。

委托就用作应用程序接收到消息时封装事件的方式。在上一节介绍委托时，仅讨论了理解事件如何工作所需要的内容。但 Microsoft 设计 C# 事件的目的是让用户无需理解底层的委托，就可以使用它们。所以下面开始从客户软件的角度讨论事件，主要考虑的是需要编写什么代码来接收事件通知，而无需担心后台上究竟发生了什么，从中可以看出事件的处理十分简单。之后，编写一个生成事件的示例，介绍事件和委托之间的关系。

本节的内容对 C++ 开发人员最有用，因为 C++ 没有与事件类似的概念。另一方面，C# 事件与 Visual Basic 事件非常类似，但 C# 中的语法和底层的实现有所不同。



注意:

这里的术语“事件”有两种不同的含义。第一,表示发生了某件有趣的事情;第二,表示C#语言中已定义的一个对象,即处理通知过程的对象。在使用第二个含义时,我们常常把事件表示为C#事件,或者在其含义很容易从上下文中看出时,就表示为事件。

### 7.2.1 从接收器的角度讨论事件

事件接收器是指在发生某些事情时被通知的任何应用程序、对象或组件。当然,有事件接收器,就有事件发送器。发送器的作用是引发事件。发送器可以是应用程序中的另一个对象或程序集,在系统事件中,例如鼠标单击或键盘按键,发送器就是.NET 运行库。注意,事件的发送器并不知道接收器是谁。这就使事件非常有用。

现在,在事件接收器的某个地方有一个方法,它负责处理事件。在每次发生已注册的事件时,就执行这个事件处理程序。此时就要使用委托了。由于发送器对接收器一无所知,所以无法设置两者之间的引用类型,而是使用委托作为中介。发送器定义接收器要使用的委托,接收器将事件处理程序注册到事件中。连接事件处理程序的过程称为封装事件。封装 Click 事件的简单例子有助于说明这个过程。

首先创建一个简单的 Windows 窗体应用程序,把一个按钮控件从工具箱拖放到窗体上。在属性窗口中把按钮重命名为 `buttonOne`。在代码编辑器中把下面的代码添加到 `Form1` 构造函数中:

```
public Form1()
{
    InitializeComponent();
    buttonOne.Click += new EventHandler(Button_Click);
}
```

在 Visual Studio 中,注意在输入 `+=` 运算符之后,就只需按下 Tab 键两次,编辑器就会完成剩余的输入工作。在大多数情况下这很不错。但在这个例子中,不使用默认的处理程序名,所以应自己输入文本。

这将告诉运行库,在引发 `buttonOne` 的 Click 事件时,应执行 `Button_Click` 方法。`EventHandler` 是事件用于把处理程序(`Button_Click`)赋予事件(`Click`)的委托。注意使用 `+=` 运算符把这个新方法添加到委托列表中。这类似于本章前面介绍的多播示例。也就是说,可以为事件添加多个事件处理程序。由于这是一个多播委托,所以要遵循添加多个方法的所有规则,但是不能保证调用方法的顺序。下面在窗体上再添加一个按钮,把它重命名为 `buttonTwo`。把 `buttonTwo` 的 Click 事件也连接到同一个 `Button_Click` 方法上,如下所示:

```
buttonOne.Click += new EventHandler(Button_Click);
buttonTwo.Click += new EventHandler(Button_Click);
```

利用委托推断,可以编写下面的代码。编译器会生成与前面相同的代码。

```
buttonOne.Click += Button_Click;
buttonTwo.Click += Button_Click;
```

`EventHandler` 委托已在 .NET Framework 中定义了。它位于 `System` 命名空间,所有在 .NET Framework 中定义的事件都使用它。如前所述,委托要求添加到委托列表中的所有方法都必须

有相同的签名。显然事件委托也有这个要求。下面是 `Button_Click` 方法的定义：

```
Private void Button_Click(object sender, EventArgs e)
{
}

```

这个方法有几个重要的地方。首先，它总是返回 `void`。事件处理程序不能有返回值。其次是参数。只要使用 `EventHandler` 委托，参数就应是 `object` 和 `EventArgs`。第一个参数是引发事件的对象，在这个例子中是 `buttonOne` 或 `buttonTwo`，这取决于被单击的按钮。把一个引用发送给引发事件的对象，就可以把同一个事件处理程序赋予多个对象。例如，可以为几个按钮定义一个按钮单击处理程序，接着根据 `sender` 参数确定单击了哪个按钮。

第二个参数 `EventArgs` 是包含有关事件的其他有用信息的对象。这个参数可以是任意类型，只要它派生自 `EventArgs` 即可。`MouseDown` 事件使用 `MouseDownEventArgs`，它包含所使用按钮的属性、指针的 `X` 和 `Y` 坐标，以及与事件相关的其他信息。注意，其命名模式是在类型的后面加上 `EventArgs`。本章的后面将介绍如何创建和使用基于 `EventArgs` 的定制对象。

方法的命名也应注意。按照约定，事件处理程序应遵循“`object_event`”的命名约定。`object` 就是引发事件的对象，而 `event` 就是被引发的事件。从可读性来看，应遵循这个命名约定。

本例最后在处理程序中添加了一些代码，以完成一些工作。记住有两个按钮使用同一个处理程序。所以首先必须确定是哪个按钮引发了事件，接着调用应执行的操作。在本例中，只是在窗体的一个标签控件上输出一些文本。把一个标签控件从工具箱拖放到窗体上，并将其命名为 `labelInfo`，然后在 `Button_Click` 方法中编写如下代码：

```
if(((Button)sender).Name == "buttonOne")
    labelInfo.Text = "Button One was pressed";
else
    labelInfo.Text = "Button Two was pressed";

```

注意，由于 `sender` 参数作为对象发送，所以必须把它的数据类型转换为引发事件的对象类型，在本例中就是 `Button`。本例使用 `Name` 属性确定是哪个按钮引发了对象，也可以使用其他属性。例如 `Tag` 属性就可以处理这种情形，因为它可以包含任何内容。为了了解事件委托的多播功能，给 `buttonTwo` 的 `Click` 事件添加另一个方法。窗体的构造函数如下所示：

```
buttonOne.Click += new EventHandler(Button_Click);
buttonTwo.Click += new EventHandler(Button_Click);
buttonTwo.Click += new EventHandler(button2_Click);

```

如果让 Visual Studio 创建存根(stub)，就会在源文件的末尾得到如下方法。但是，必须添加对 `MessageBox.Show()` 函数的调用：

```
Private void button2_Click(object sender, EventArgs e)
{
    MessageBox.Show("This only happens in Button 2 click event");
}

```

如果使用  $\lambda$  表达式，就不需要 `Button_Click` 方法和 `Button2_Click` 方法了。事件的代码如下：

```
buttonOne.Click += (sender, e) => labelInfo.Text = "Button One was pressed";
buttonTwo.Click += (sender, e) => labelInfo.Text = "Button Two was pressed";
buttonTwo.Click += (sender, e) =>

```

```
{
    MessageBox.Show("This only happens in Button 2 click event");
};
```

在运行这个例子时，单击 `buttonOne` 会改变标签上的文本。单击 `buttonTwo` 不仅会改变文本，还会显示消息框。注意，不能保证标签文本在消息框显示之前改变，所以不要在处理程序中编写具有依赖性的代码。

我们已经学习了许多概念，但要在接收器中编写的代码量是很小的。记住，编写事件接收器常常比编写事件发送器要频繁得多。至少在 Windows 用户界面上，Microsoft 已经编写了所有需要的事件发送器(它们都在 .NET 基类中，在 `Windows.Forms` 命名空间中)。

### 7.2.2 生成事件

接收事件并响应它们仅是事件的一个方面。为了使事件真正发挥作用，还需要在代码中生成和引发事件。下面的例子将介绍如何创建、引发、接收和取消事件。

这个例子包含一个窗体，它会引发另一个类正在监听的事件。在引发事件后，接收对象就确定是否执行一个过程，如果该过程未能继续，就取消事件。本例的目标是确定当前时间的秒数是大于 30 还是小于 30。如果秒数小于 30，就用一个表示当前时间的字符串设置一个属性；如果秒数大于 30，就取消事件，把时间字符串设置为空。

用于生成事件的窗体包含一个按钮和一个标签。下载的示例代码把按钮命名为 `buttonRaise`，标签命名为 `labelInfo`。在创建窗体，添加两个控件后，就可以创建事件和相应的委托了。在窗体类的类声明部分，添加如下代码：

```
public delegate void ActionEventHandler(object sender, ActionCancelEventArgs ev);
public static event ActionEventHandler Action;
```

这两行代码的作用是什么？首先，我们声明了一种新的委托类型 `ActionEventHandler`。必须创建一种新委托，而不使用 .NET Framework 预定义的委托，其原因是后面要使用定制的 `EventArgs` 类。方法签名必须与委托匹配。有了一个要使用的委托后，下一行代码就定义事件。在本例中定义了 `Action` 事件，定义事件的语法要求指定与事件相关的委托。还可以使用在 .NET Framework 中定义的委托。从 `EventArgs` 类中派生出了近 100 个类，应该可以找到一个自己能使用的类。但本例使用的是定制的 `EventArgs` 类，所以必须创建一个与之匹配的新委托类型。

在一行代码中定义事件是 C# 中的一个缩写方式，它可以定义添加和删除处理程序的方法，声明委托的一个变量。除了编写一行代码之外，还可以用下面的代码达到相同的效果。声明一个事件类型的变量以及添加和删除事件处理程序的方法。在定义添加和删除事件处理程序的方法时，其语法非常类似于属性。变量值的定义也类似于添加和删除事件处理程序。

```
private static ActionEventHandler action;

public static event ActionEventHandler Action
{
    add
    {
        action += value;
    }
    remove
    {

```

```

        action -= value;
    }
}

```

**提示:**

如果不仅仅需要添加和删除事件处理程序，就可以使用定义事件的较长记号。例如，要为多个线程访问添加同步功能。WPF 控件就使用这种较长的记号给事件添加起泡和通道功能。事件的起泡和通道功能详见第 34 章。

基于 EventArgs 的新类 ActionCancelEventArgs 实际上派生自 CancelEventArgs，而 CancelEventArgs 派生自 EventArgs。CancelEventArgs 添加了 Cancel 属性，该属性是一个布尔值，它通知 sender 对象，接收器希望取消或停止事件的处理。在 ActionEventHandler 类中，还添加了 Message 属性，这是一个字符串属性，包含事件处理状态的文本信息。下面是 ActionCancelEventArgs 类的代码：

```

public class ActionCancelEventArgs : System.ComponentModel.CancelEventArgs
{
    public ActionCancelEventArgs() : this(false) {}

    public ActionCancelEventArgs(bool cancel) : this(false, String.Empty) {}
    public ActionCancelEventArgs(bool cancel, string message) : base(cancel)
    {
        this.message = message;
    }
    public string Message { get; set; }
}

```

可以看出，所有基于 EventArgs 的类都负责在发送器和接收器之间来回传送事件的信息。在大多数情况下，EventArgs 类中使用的信息都由事件处理程序中的接收器对象使用。但是，有时事件处理程序可以把信息添加到 EventArgs 类中，使之可用于发送器。这就是本例使用 EventArgs 类的方式。注意在 EventArgs 类中有两个可用的构造函数。这种额外的灵活性增加了该类的可用性。

目前声明了一个事件，定义了一个委托，并创建了 EventArgs 类。下一步需要引发事件。真正需要做的是用正确的参数调用事件，如本例所示：

```

ActionCancelEventArgs e = new ActionCancelEventArgs();
Action(this, e);

```

这非常简单。创建新的 ActionCancelEventArgs 类，并把它作为一个参数传递给事件。但是，这有一个小问题。如果事件不会在任何地方使用，该怎么办？如果还没有为事件定义处理程序，该怎么办？Action 事件实际上是空的。如果试图引发该事件，就会得到一个空引用异常。如果要派生一个新的窗体类，并使用该窗体，把 Action 事件定义为基事件，则只要引发了 Action 事件，就必须执行其他一些操作。目前，我们必须在派生的窗体中激活另一个事件处理程序，这样才能访问它。为了使这个过程容易一些，并捕获空引用错误，就必须创建一个方法 OnEventName，其中 EventName 是事件名。在这个例子中，有一个 OnAction 方法，下面是 OnAction 方法的完整代码：

```

protected void OnAction(object sender, ActionCancelEventArgs ev)

```



```

{
    if (Action != null)
    {
        Action(sender, ev);
    }
}

```

代码并不多，但完成了需要的工作。把该方法声明为 `protected`，就只有派生类可以访问它。事件在引发之前还会进行空引用测试。如果派生一个包含该方法和事件的新类，就必须重写 `OnAction` 方法，然后连接事件。为此，必须在重写代码中调用 `base.OnAction()`。否则就不会引发该事件。在整个 .NET Framework 中都用这个命名约定，并在 .NET SDK 文档中对这一命名规则进行了说明。

注意传送给 `OnAction` 方法的两个参数。它们看起来很熟悉，因为它们与需要传送给事件的参数相同。如果事件需要从另一个对象中引发，而不是从定义方法的对象中引发，就需要把访问修饰符设置为 `internal` 或 `public`，而不能设置为 `protected`。有时让类只包含事件声明，这些事件从其他类中调用是有意义的。仍可以创建 `OnEventName` 方法，但此时它们是静态方法。

目前，我们已经引发了事件，还需要一些代码来处理它。在项目中创建一个新类 `BusEntity`。本项目的目的是检查当前时间的秒数，如果它小于 30，就把一个字符串值设置为时间；如果它大于 30，就把字符串设置为 `::`，并取消事件。下面是代码：

```

using System;
using System.IO;
using System.ComponentModel;

namespace Wrox.ProCSharp.Delegates
{
    public class BusEntity
    {
        string time = String.Empty;

        public BusEntity()
        {
            Form1.Action += new Form1.ActionEventHandler(Form1_Action);
        }

        private void Form1_Action(object sender, ActionCancelEventArgs e)
        {
            e.Cancel = !DoActions();
            if (e.Cancel)
                e.Message = "Wasn't the right time.";
        }

        private bool DoActions()
        {
            bool retVal = false;
            DateTime tm = DateTime.Now;

            if (tm.Second < 30)
            {
                time = "The time is " + DateTime.Now.ToLongTimeString();
                retVal = true;
            }
            else
                time = "";
        }
    }
}

```



```

        return retVal;
    }

    public string TimeString
    {
        get {return time;}
    }
}

```

在构造函数中声明了 `Form1.Action` 事件的处理程序。注意其语法非常类似于前面 `Click` 事件的语法。由于声明事件使用的模式都是相同的，所以语法也应保持一致。还要注意如何获取 `Action` 事件的引用，而无需在 `BusEntity` 类中引用 `Form1`。在 `Form1` 类中，将 `Action` 事件声明为静态，这并不是必需的，但这样更易于创建处理程序。我们可以把事件声明为 `public`，但接着需要引用 `Form1` 的一个实例。

在构造函数中编写事件时，调用添加到委托列表中的方法 `Form1_Action`，并遵循命名标准。在处理程序中，需要确定是否取消事件。`DoActions` 方法根据前面描述的时间条件返回一个布尔值，并把 `_time` 字符串设置为正确的值。

之后，把 `DoActions` 的返回值赋给 `ActionCancelEventArgs` 的 `Cancel` 属性。`EventArgs` 类一般仅在事件发送器和接收器之间来回传递值。如果取消了事件(`ev.Cancel = true`)，`Message` 属性就设置为一个字符串值，以说明事件为什么被取消。

如果再次查看 `buttonRaise_Click` 事件处理程序的代码，就可以看出 `Cancel` 属性的使用方式：

```

private void buttonRaise_Click(object sender, EventArgs e)
{
    ActionCancelEventArgs cancelEvent = new ActionCancelEventArgs();
    OnAction(this, cancelEvent);
    if(cancelEvent.Cancel)
        labelInfo.Text = cancelEvent.Message;
    else
        labelInfo.Text = busEntity.TimeString;
}

```

注意，创建了 `ActionCancelEventArgs` 对象。接着引发了事件 `Action`，并传递了新建的 `ActionCancelEventArgs` 对象。在调用 `OnAction` 方法，引发事件时，`BusEntity` 对象中 `Action` 事件处理程序的代码就会执行。如果还有其他对象注册了事件 `Action`，它们也会执行。记住，如果其他对象也处理这个事件，它们就会看到同一个 `ActionCancelEventArgs` 对象。如果需要确定是哪个对象取消了事件，而且有多个对象取消了事件，就需要在 `ActionCancelEventArgs` 类中包含某种基于列表的数据结构。

在与事件委托一起注册的处理程序执行完毕后，就可以查询 `ActionCancelEventArgs` 对象，确定它是否被取消了。如果是，`labelInfo` 就包含 `Message` 属性值；如果事件没有被取消，`labelInfo` 就会显示当前时间。

本节这基本上说明了如何利用事件和事件中基于 `EventArgs` 的对象，在应用程序中传递信息。

## 7.3 小结

本章介绍了委托和事件的基本知识，解释了如何声明委托，如何给委托列表添加方法，并

讨论了声明事件处理程序来响应事件的过程，以及如何创建定制事件，使用引发事件的模式。

.NET 开发人员将大量使用委托和事件，特别是开发 Windows Forms 应用程序。事件是 .NET 开发人员监视应用程序执行时出现的各种 Windows 消息的方式，否则就必须监视 WndProc，捕获 WM\_MOUSEBUTTONDOWN 消息，而不是获取按钮的鼠标 Click 事件。

在设计大型应用程序时，使用委托和事件可以减少依赖性和层的关联，并能开发出具有更高复用性的组件。

匿名方法和  $\lambda$  表达式是委托的 C# 语言特性。通过它们可以减少需要编写的代码量。 $\lambda$  表达式不仅仅用于委托，详见第 11 章。

下一章介绍字符串和正则表达式。

## 字符串和正则表达式

在本书的第一部分，我们一直在使用字符串，并说明 C# 中 `string` 关键字的映射实际上指向 .NET 基类 `System.String`。`System.String` 是一个功能非常强大且用途非常广泛的基类，但它不是 .NET 中唯一与字符串相关的类。本章首先复习一下 `System.String` 的特性，再介绍如何使用其他的 .NET 类来处理字符串，特别是 `System.Text` 和 `System.Text.Regular Expressions` 命名空间中的类。本章主要介绍下述内容：

- 创建字符串：如果多次修改一个字符串，例如，在显示字符串或将其传递给其他方法或应用程序前，创建一个较长的字符串，`String` 类就会变得效率低下。对于这种情况，应使用另一个类 `System.Text.StringBuilder`，因为它是专门为这种情况设计的。
- 格式化表达式：这些表达式将用于后面几章中的 `Console.WriteLine()` 方法。格式化表达式使用两个有效的接口 `IFormatProvider` 和 `IFormattable` 来处理。在自己的类上执行这两个接口，就可以定义自己的格式化序列，这样，`Console.WriteLine()` 和类似的类就可以以指定的方式显示类的值。
- 正则表达式：.NET 还提供了一些非常复杂的类来识别字符串，或从长字符串中提取满足某些复杂条件的子字符串。例如，找出字符串中重复出现的某个字符或一组字符，或者找出以 `s` 开头、且至少包含一个 `n` 的所有单词，或者找出遵循雇员 ID 或社会安全号码约定的字符串。虽然可以使用 `String` 类，编写方法来执行这类处理，但这类方法编写起来比较繁琐，而使用 `System.Text.RegularExpressions` 命名空间中的类就比较简单，`System.Text.RegularExpressions` 专门用于执行这类处理。

### 8.1 System.String 类

在介绍其他字符串类之前，先快速复习一下 `String` 类上一些可用的方法。

`System.String` 是一个类，专门用于存储字符串，允许对字符串进行许多操作。由于这种数据类型非常重要，C# 提供了它自己的关键字和相关的语法，以便于使用这个类来处理字符串。

使用运算符重载可以连接字符串：

```
string message1 = "Hello"; //return "Hello"
message1 += ", There";    // return "Hello, There "
string message2 = message1 + "!"; // return "Hello, There!"
```

C# 还允许使用类似于索引器的语法来提取指定的字符：

```
char char4 = message[4]; // returns 'a'. Note the char is zero-indexed
```

这个类可以完成许多常见的任务，例如替换字符、删除空白和把字母变成大写形式等。可用的方法如表 8-1 所示。

表 8-1

方 法	作 用
Compare	比较字符串的内容，考虑文化背景(区域)，确定某些字符是否相等
CompareOrdinal	与 Compare 一样，但不考虑文化背景
Concat	把多个字符串实例合并为一个实例
CopyTo	把特定数量的字符从选定的下标复制到数组的一个全新实例中
Format	格式化包含各种值的字符串和如何格式化每个值的说明符
IndexOf	定位字符串中第一次出现某个给定子字符串或字符的位置
IndexOfAny	定位字符串中第一次出现某个字符或一组字符的位置
Insert	把一个字符串实例插入到另一个字符串实例的指定索引处
Join	合并字符串数组，建立一个新字符串
LastIndexOf	与 IndexOf 一样，但定位最后一次出现的位置
LastIndexOfAny	与 IndexOfAny，但定位最后一次出现的位置
PadLeft	在字符串的开头，通过添加指定的重复字符填充字符串
PadRight	在字符串的结尾，通过添加指定的重复字符填充字符串
Replace	用另一个字符或子字符串替换字符串中给定的字符或子字符串
Split	在出现给定字符的地方，把字符串拆分为一个子字符串数组
Substring	在字符串中获取给定位置的子字符串
ToLower	把字符串转换为小写形式
ToUpper	把字符串转换为大写形式
Trim	删除首尾的空白

注意：  
这个表并不完整，但可以让您明白字符串所提供的功能。

### 8.1.1 创建字符串

如上所述，string 类是一个功能非常强大的类，它执行许多很有用的方法。但是，string 类存在一个问题：重复修改给定的字符串，效率会很低，它实际上是一个不可变的数据类型，一旦对字符串对象进行了初始化，该字符串对象就不能改变了。表面上修改字符串内容的方法和运算符实际上是创建一个新的字符串，如果必要，可以把旧字符串的内容复制到新字符串中。例如，下面的代码：

```
string greetingText = "Hello from all the guys at Wrox Press. ";
```

```
greetingText += "We do hope you enjoy this book as much as we enjoyed writing it.";
```

在执行这段代码时，首先，创建一个 `System.String` 类型的对象，并初始化为文本“Hello from all the guys at Wrox Press. ”。注意句号后面有一个空格。此时.NET 运行库会为该字符串分配足够的内存来保存这个文本(39 个字符)，再设置变量 `greetingText`，表示这个字符串实例。

从语法上看，下一行代码是把更多的文本添加到字符串中。实际上并非如此，而是创建一个新字符串实例，给它分配足够的内存，以保存合并起来的文本(共 103 个字符)。最初的文本“Hello from all the people at Wrox Press.”复制到这个新字符串中，再加上额外的文本“We do hope you enjoy this book as much as we enjoyed writing it.”。然后更新存储在变量 `greetingText` 中的地址，使变量正确地指向新的字符串对象。旧的字符串对象被撤销了引用——不再有变量引用它，下一次垃圾收集器清理应用程序中所有未使用的对象时，就会删除它。

这本身还不坏，但假定要对这个字符串加密，在字母表中，用 ASCII 码中的字符替代其中的每个字母(标点符号除外)，作为非常简单的加密模式的一部分，就会把该字符串变成“Ifmmp gspn bmm uif hvst bu Xspy Qsftt. Xf ep ipqf zpv fokpz uijt cppl bt nvdi bt xf fokpzfe xsjujoh ju.”。完成这个任务有好几种方式，但最简单、最高效的一种(假定只使用 `String` 类)是使用 `String.Replace()` 方法，把字符串中指定的子字符串用另一个子字符串代替。使用 `Replace()`，加密文本的代码如下所示：

```
string greetingText = "Hello from all the guys at Wrox Press. ";
greetingText += "We do hope you enjoy this book as much as we enjoyed writing it.";
```

```
for(int i = 'z'; i>='a' ; i--)
{
    char old1 = (char)i;
    char new1 = (char)(i+1);
    greetingText = greetingText.Replace(old1, new1);
}

for(int i = 'Z'; i>='A' ; i--)
{
    char old1 = (char)i;
    char new1 = (char)(i+1);
    greetingText = greetingText.Replace(old1, new1);
}

Console.WriteLine("Encoded:\n" + greetingText);
```

注意：

为了简单起见，这段代码没有把 Z 换成 A，或把 z 换成 a。这些字符分别编码为[和{。

`Replace()`以一种智能化的方式工作，在某种程度上，它并没有创建一个新字符串，除非要对旧字符串进行某些改变。原来的字符串包含 23 个不同的小写字母，和 3 个不同的大写字母。所以 `Replace()`就分配一个新字符串，共 26 次，每个新字符串都包含 103 个字符。因此加密过程需要在堆上有一个能存储总共 2678 个字符的字符串对象，最终将等待被垃圾收集！显然，如果使用字符串进行文字处理，应用程序就会有严重的性能问题。

为了解决这个问题，Microsoft 提供了 `System.Text.StringBuilder` 类。`StringBuilder` 不像 `String` 那样支持非常多的方法。在 `StringBuilder` 上可以进行的处理仅限于替换和添加或删除字符串中的文本。但是，它的工作方式非常高效。

在使用 `String` 类构造一个字符串时，要给它分配足够的内存来保存字符串，但 `StringBuilder`



通常分配的内存会比需要的更多。开发人员可以选择显式指定 `StringBuilder` 要分配多少内存，但如果没有显式指定，存储单元量在默认情况下就根据 `StringBuilder` 初始化时的字符串长度来确定。它有两个主要的属性：

- `Length` 指定字符串的实际长度；
- `Capacity` 是字符串占据存储单元的最大长度。

对字符串的修改就在赋予 `StringBuilder` 实例的存储单元中进行，这就大大提高了添加子字符串和替换单个字符的效率。删除或插入子字符串仍然效率低下，因为这需要移动随后的字符串。只有执行扩展字符串容量的操作，才需要给字符串分配新内存，才可能移动包含的整个字符串。在添加额外的容量时，从经验来看，`StringBuilder` 如果检测到容量超出，且容量没有设置新值，就会使自己的容量翻倍。

例如，如果使用 `StringBuilder` 对象构造最初的欢迎字符串，可以编写下面的代码：

```
StringBuilder greetingBuilder =
    new StringBuilder("Hello from all the guys at Wrox Press. ", 150);
greetingBuilder.AppendFormat("We do hope you enjoy this book as much as we enjoyed
    writing it");
```

注意：

为了使用 `StringBuilder` 类，需要在代码中引用 `System.Text`。

在这段代码中，为 `StringBuilder` 设置的初始容量是 150。最好把容量设置为字符串可能的最大长度，确保 `StringBuilder` 不需要重新分配内存，因为其容量足够用了。理论上，可以设置尽可能大的数字，足够给该容量传送一个 `int`，但如果实际上给字符串分配 20 亿个字符的空间(这是 `StringBuilder` 实例允许拥有的最大理论空间)，系统就可能会没有足够的内存。

执行上面的代码，首先创建一个 `StringBuilder` 对象，如图 8-1 所示。

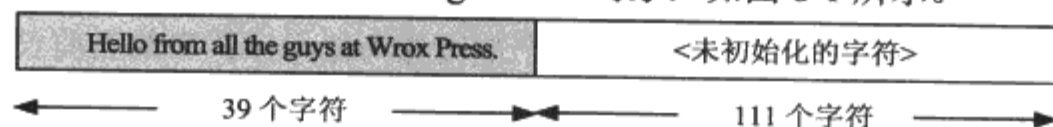


图 8-1

在调用 `Append()` 方法时，其他文本就放在空的空间中，不需要分配更多的内存。但是，多次替换文本才能获得使用 `StringBuilder` 所带来的性能提高。例如，如果要以前面的方式加密文本，就可以执行整个加密过程，无须分配更多的内存：

```
StringBuilder greetingBuilder =
    new StringBuilder("Hello from all the guys at Wrox Press. ", 150);
greetingBuilder.Append("We do hope you enjoy this book as much as we " + "enjoyed
    writing it");
```

```
Console.WriteLine("Not Encoded:\n" + greetingBuilder);
for(int i = 'z'; i>='a' ; i--)
{
    char old1 = (char)i;
    char new1 = (char)(i+1);
    greetingBuilder = greetingBuilder.Replace(old1, new1);
}
for(int i = 'Z'; i>='A' ; i--)
```

```
{
    char old1 = (char)i;
    char new1 = (char)(i+1);
    greetingBuilder = greetingBuilder.Replace(old1, new1);
}
Console.WriteLine("Encoded:\n" + greetingBuilder);
```

这段代码使用了 `StringBuilder.Replace()` 方法，它的功能与 `String.Replace()` 一样，但不需要在过程中复制字符串。在上述代码中，为存储字符串而分配的总存储单元是 150 个字符，用于 `StringBuilder` 实例以及在最后一个 `Console.WriteLine()` 语句中执行字符串操作期间分配的内存。一般，使用 `StringBuilder` 可以执行字符串的操作，`String` 可以存储字符串或显示最终结果。

8.1.2 `StringBuilder` 成员

前面介绍了 `StringBuilder` 的一个构造函数，它的参数是一个初始字符串及该字符串的容量。还有几个其他的 `StringBuilder` 构造函数，例如，可以只提供一个字符串：

```
StringBuilder sb = new StringBuilder("Hello");
```

或者用给定的容量创建一个空的 `StringBuilder`：

```
StringBuilder sb = new StringBuilder(20);
```

除了前面介绍的 `Length` 和 `Capacity` 属性外，还有一个只读属性 `MaxCapacity`，它表示对给定的 `StringBuilder` 实例的容量限制。在默认情况下，这由 `int.MaxValue` 给定(大约 20 亿，如前所述)。但在构造 `StringBuilder` 对象时，也可以把这个值设置为较低的值：

```
// This will both set initial capacity to 100, but the max will be 500.
// Hence, this StringBuilder can never grow to more than 500 characters,
// otherwise it will raise exception if you try to do that.
StringBuilder sb = new StringBuilder(100, 500);
```

还可以随时显式地设置容量，但如果把这个值设置为低于字符串的当前长度，或者超出了最大容量，就会抛出一个异常：

```
StringBuilder sb = new StringBuilder("Hello");
sb.Capacity = 100;
```

主要的 `StringBuilder` 方法如表 8-2 所示。

表 8-2

名 称	作 用
<code>Append()</code>	给当前字符串添加一个字符串
<code>AppendFormat()</code>	添加特定格式的字符串
<code>Insert()</code>	在当前字符串中插入一个子字符串
<code>Remove()</code>	从当前字符串中删除字符
<code>Replace()</code>	在当前字符串中，用某个字符替换另一个字符，或者用当前字符串中的一个子字符串替换另一字符串
<code>ToString()</code>	把当前字符串转换为 <code>System.String</code> 对象(在 <code>System.Object</code> 中被重写)

其中一些方法还有几种格式的重载方法。

注意:

AppendFormat()实际上会在调用 Console.WriteLine()时调用,它负责确定所有像{0:D}的格式化表达式应使用什么表达式替代。下一节讨论这个问题。

不能把 StringBuilder 转换为 String(隐式转换和显式转换都不行)。如果要把 StringBuilder 的内容输出为 String,唯一的方式是使用 ToString()方法。

前面介绍了 StringBuilder 类,说明了使用它提高性能的一些方式。注意,这个类并不总能提高性能。StringBuilder 类基本上应在处理多个字符串时使用。但如果只是连接两个字符串,使用 System.String 会比较好。

### 8.1.3 格式化字符串

前面的代码示例中编写了许多类和结构,对这些类和结构执行 ToString()方法,都是为了显示给定变量的内容。但是,用户常常希望以各种可能的方式显示变量的内容,在不同的文化或地区背景中有不同的格式。.NET 基类 System.DateTime 就是最明显的一个示例:可以把日期显示为 10 June 2008、10 Jun 2008、6/10/08 (美国)、10/6/08 (英国)或 10.06.2008 (德国)。

同样,第6章中编写的 Vector 结构执行 Vector.ToString()方法,是为了以(4, 56, 8)格式显示矢量。编写矢量的另一个非常常用的方式是  $4i + 56j + 8k$ 。如果要使类的用户友好性比较高,就需要使用某些工具以用户希望的方式显示它们的字符串表示。.NET 运行库定义了一种标准方式:使用接口 IFormattable,本节的主题就是说明如何把这个重要特性添加到类和结构上。

在显示一个变量时,常常需要指定它的格式,此时我们经常调用 Console.WriteLine()方法。因此,我们把这个方法作为示例,但这里的讨论适用于格式化字符串的大多数情况。例如,如果要在列表框或文本框中显示一个变量的值,一般要使用 String.Format()方法来获得该变量的合适字符串表示,但用于请求所需格式的格式说明符与传递给 Console.WriteLine()的格式相同,因此本节把 Console.WriteLine()作为一个示例来说明。首先看看在为基本类型提供格式字符串时会发生什么,再看看如何把自己的类和结构的格式说明符添加到过程中。

第2章在 Console.Write()和 Console.WriteLine()中使用了格式字符串:

```
double d = 13.45;
int i = 45;
Console.WriteLine("The double is {0,10:E} and the int contains {1}", d, i);
```

格式字符串本身大都由要显示的文本组成,但只要有要格式化的变量,它在参数列表中的下标就必须放在括号中。在括号中还可以有与该项的格式相关的其他信息,例如可以包含:

- 该项的字符串表示要占用的字符数,这个信息的前面应有一个逗号,负值表示该项应左对齐,正值表示该项应右对齐。如果该项占用的字符数比给定的多,其内容也会完整地显示出来。
- 格式说明符也可以显示出来。它的前面应有一个冒号,表示应如何格式化该项。例如,把一个数字格式化为货币,或者以科学计数法显示。

第2章简要介绍了数字类型的常见格式说明符,表8-3再次引用该表。

表 8-3

格 式 符	应 用	含 义	示 例
C	数字类型	专用场合的货币值	\$4834.50 (USA) £4834.50 (UK)
D	只用于整数类型	一般的整数	4834
E	数字类型	科学计数法	4.834E+003
F	数字类型	小数点后的位数固定	4384.50
G	数字类型	一般的数字	4384.5
N	数字类型	通常是专用场合的数字格式	4,384.50 (UK/USA) 4 384,50 (欧洲大陆)
P	数字类型	百分比计数法	432,000.00%
X	只用于整数类型	十六进制格式	1120 (如果要显示 0x1120, 需 要写上 0x)

如果要在整数上加上前导 0, 可以将格式说明符 0 重复所需的次数。例如, 格式说明符 0000 会把 3 显示为 0003, 99 显示为 0099。

这里不能给出完整的列表, 因为其他数据类型有自己的格式说明符。本节的主要目的是说明如何为自己的类定义格式说明符。

1. 字符串的格式化

为了说明如何格式化字符串, 看看执行下面的语句会得到什么结果:

```
Console.WriteLine("The double is {0,10:E} and the int contains {1}", d, i);
```

Console.WriteLine()只是把参数的完整列表传送给静态方法 String.Format(), 如果要在字符串中以其他方式格式化这些值, 例如显示在一个文本框中, 也可以调用这个方法。带有 3 个参数的 WriteLine()重载方法如下:

```
// Likely implementation of Console.WriteLine()
public void WriteLine(string format, object arg0, object arg1)
{
    Console.WriteLine(string.Format(format, arg0, arg1));
}
```

上面的代码依次调用了带有 1 个参数的重载方法 WriteLine(), 仅显示了传递过来的字符串的内容, 没有对它进行进一步的格式化。

String.Format()现在需要用对应对象的合适字符串表示来替换每个格式说明符, 构造最终的字符串。但是, 如前所述, 对于这个建立字符串的过程, 需要 StringBuilder 实例, 而不是 String 实例。在这个示例中, StringBuilder 实例是用字符串的第一部分(即文本 “The double is”)创建和初始化的。然后调用 StringBuilder.AppendFormat()方法, 传递第一个格式说明符 “{0,10:E}” 和相应的对象 double, 把这个对象的字符串表示添加到构造好的字符串中, 这个过程会继续重复调用 StringBuilder.Append()和 StringBuilder.AppendFormat()方法, 直到得到了全部格式化好的字符串为止。

下面的内容比较有趣。StringBuilder.AppendFormat()需要指出如何格式化对象，它首先检查对象，确定它是否执行 System 命名空间中的接口 IFormattable。只要试着把这个对象转换为接口，看看转换是否成功即可，或者使用 C#关键字 is，也能实现此测试。如果测试失败，AppendFormat()只会调用对象的 ToString()方法，所有的对象都从 System.Object 继承了这个方法或重写了该方法。在前面给出的编写各种类和结构的示例中，执行过程都是这样，因为我们编写的类都没有执行这个接口。这就是在前面的章节中，Object.ToString()的重写方法允许在 Console.WriteLine()语句中显示类和结构如 Vector 的原因。

但是，所有预定义的基本数字类型都执行这个接口，对于这些类型，特别是这个示例中的 double 和 int，就不会调用继承自 System.Object 的基本 ToString()方法。为了理解这个过程，需要了解 IFormattable 接口。

IFormattable 只定义了一个方法，该方法也叫作 ToString()，它带有两个参数，这与 System.Object 版本的 ToString()不同，它不带参数。下面是 IFormattable 的定义：

```
interface IFormattable
{
    string ToString(string format, IFormatProvider formatProvider);
}
```

这个 ToString()重载方法的第一个参数是一个字符串，它指定要求的格式。换言之，它是字符串的说明符部分，放在字符串的 {} 中，该参数最初传递给 Console.WriteLine()或 String.Format()。例如，在本例中，最初的语句如下：

```
Console.WriteLine("The double is {0,10:E} and the int contains {1}", d, i);
```

在计算第一个说明符 {0,10:E} 时，在 double 变量 d 上调用这个重载方法，传递给它的第一个参数是 E。StringBuilder.AppendFormat()传递的总是显示在原始字符串的合适格式说明符内冒号后面的文本。

本书不讨论 ToString()的第 2 个参数，它是执行接口 IFormatProvider 的对象引用。这个接口提供了 ToString()在格式化对象时需要考虑的更多信息——一般包括文化背景信息(.NET 文化背景类似于 Windows 时区，如果格式化货币或日期，就需要这些信息)。如果直接从源代码中调用这个 ToString()重载方法，就需要提供这样一个对象。但 StringBuilder.AppendFormat()为这个参数传递一个空值。如果 formatProvider 为空，ToString()就要使用系统设置中指定的文化背景信息。

现在回过头来看看本例。第一个要格式化的项是 double，对此要求使用指数计数法，格式说明符为 E。如前所述，StringBuilder.AppendFormat()方法会建立执行 IFormattable 接口的对象 double，因此要调用带有两个参数的 ToString()重载方法，其第一个参数是字符串“E”，第二个参数为空。现在 double 的这个方法在执行时，会考虑要求的格式和当前的文化背景，以合适的格式返回 double 的字符串表示。StringBuilder.AppendFormat()则按照需要在返回的字符串中添加前导空格，使之共有 10 个字符。

下一个要格式化的对象是 int，它不需要任何特殊的格式（格式说明符是 {1}）。由于没有格式要求，StringBuilder.AppendFormat()会给该格式字符串传递一个空引用，并适当地响应带有两个参数的 int.ToString()重载方法。由于没有特殊的格式要求，所以也可以调用不带参数的 ToString()方法。

整个字符串格式化过程如图 8-2 所示。



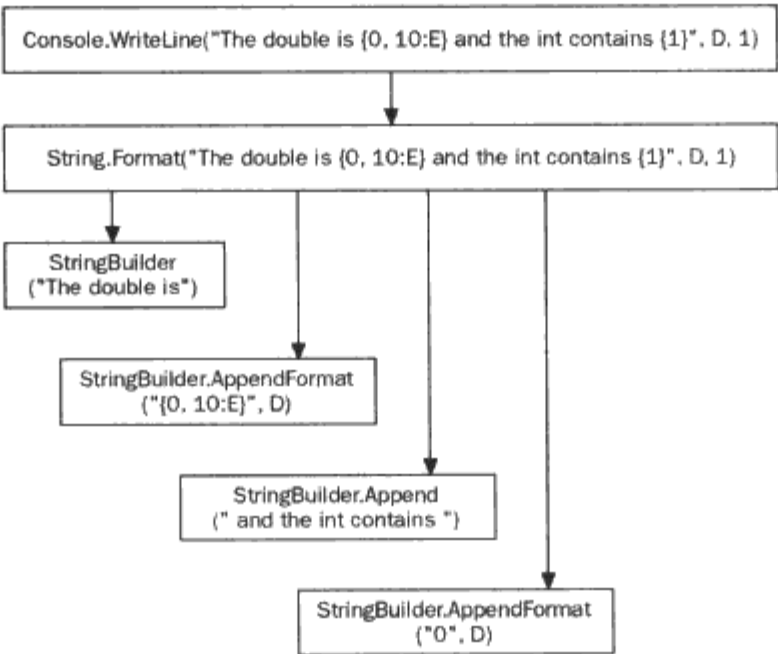


图 8-2

2. FormattableVector 示例

前面介绍了如何构造格式字符串，下面扩展本书第 6 章的 Vector 示例，以多种方式格式化矢量。这个示例的代码可以从 [www.wrox.com](http://www.wrox.com) 上下载。只要理解了所涉及的规则，实际编写代码就相当简单了。我们只需要实现 IFormattable，提供由该接口定义的 ToString()重载方法即可。

要支持的格式说明符如下：

- N 应解释为一个请求，以提供一个数字，即矢量的模，它是其成员的平方和，在数学上等于 Vector 的长度的平方，通常放在两个竖杠的中间：||34.5||。
- VE 应解释为以科学计数法显示每个成员的一个请求，例如说明符 E 应用于 double，就可以表示为(2.3E+01, 4.5E+02, 1.0E+00)。
- IJK 应解释为以格式 23i + 450j + 1k 显示矢量的一个请求。
- 其他内容应仅返回 Vector 的默认表示方法(23, 450, 1.0)。

为了简单起见，我们不以 IJK 和科学计数法的格式执行任何选项，以显示矢量，而是以不区分大小写的方式来测试说明符，允许使用 ijk 和 IJK。注意，使用什么字符串表示格式说明符完全取决于用户。

为此，首先修改 Vector 的声明，使之执行 IFormattable：

```
struct Vector : IFormattable
{
    public double x, y, z;

    // Beginning part of Vector
```

下面添加带有 2 个参数的 ToString()重载方法：

```
public string ToString(string format, IFormatProvider formatProvider)
{
    if (format == null)
    {
        return ToString();
    }
```

```

string formatUpper = format.ToUpper();
switch (formatUpper)
{
    case "N":
        return "|| " + Norm().ToString() + " ||";
    case "VE":
        return String.Format("( {0:E}, {1:E}, {2:E} )", x, y, z);
    case "IJK":
        StringBuilder sb = new StringBuilder(x.ToString(), 30);
        sb.AppendFormat(" i + ");
        sb.AppendFormat(y.ToString());
        sb.AppendFormat(" j + ");
        sb.AppendFormat(z.ToString());
        sb.AppendFormat(" k");
        return sb.ToString();
    default:
        return ToString();
}
}

```

这就是我们要编写的代码。注意在调用任何方法前，应防止使用格式字符串为空的参数。我们希望这个方法尽可能健壮，所有基本类型的格式说明符都是不区分大小写的，其他开发人员也希望能使用我们的类。对于格式说明符 VE，需要把每个成员格式化为科学计数法，所以再次使用 `String.Format()` 方法。字段 `x`、`y` 和 `z` 都是 `double` 类型。对于 IJK 格式限定符，把几个子字符串添加到字符串中，因此使用 `StringBuilder` 对象来提高性能。

为了保证完整，也可以再次使用前面开发的无参数的 `ToString()` 重载方法：

```

public override string ToString()
{
    return "( " + x + " , " + y + " , " + z + " )";
}

```

最后，需要添加一个 `Norm()` 方法，计算矢量的平方(模)，因为在开发 `Vector` 结构时，没有提供这个方法：

```

public double Norm()
{
    return x*x + y*y + z*z;
}

```

下面用一些合适的测试代码测试可格式化的矢量：

```

static void Main()
{
    Vector v1 = new Vector(1, 32, 5);
    Vector v2 = new Vector(845.4, 54.3, -7.8);
    Console.WriteLine("\nIn IJK format, \nv1 is {0,30:IJK} \nv2 is {1,30:IJK}", v1, v2);
    Console.WriteLine("\nIn default format, \nv1 is {0,30} \nv2 is {1,30}", v1, v2);
    Console.WriteLine("\nIn VE format \nv1 is {0,30:VE} \nv2 is {1,30:VE}", v1, v2);
    Console.WriteLine("\nNorms are: \nv1 is {0,20:N} \nv2 is {1,20:N}", v1, v2);
}

```

运行这个示例的结果如下所示：

```
FormattableVector
```

```

In IJK format,
v1 is          1 i + 32 j + 5 k
v2 is      845.4 i + 54.3 j + -7.8 k

In default format,
v1 is          ( 1 , 32 , 5 )
v2 is      ( 845.4 , 54.3 , -7.8 )

In VE format
v1 is ( 1.000000E+000, 3.200000E+001, 5.000000E+000 )
v2 is ( 8.454000E+002, 5.430000E+001, -7.800000E+000 )

Norms are:
v1 is          || 1050 ||
v2 is      || 717710.49 ||

```

这说明了选用的定制格式说明符是正确的。

## 8.2 正则表达式

正则表达式在各种程序中都有着难以置信的作用，但并不是所有的开发人员都知道这一点。正则表达式可以看做一种有特定功能的小型编程语言：在大的字符串表达式中定位一个子字符串。它不是一种新技术，最初它是在 UNIX 环境中开发的，与 Perl 一起使用得比较多。Microsoft 把它移植到 Windows 中，到目前为止在脚本语言中用得比较多。但 System.Text.RegularExpressions 命名空间中的许多 .NET 类都支持正则表达式。.NET Framework 的各个部分都使用正则表达式，例如，在 ASP.NET 的验证服务器控件中就使用了正则表达式。

许多人都不太熟悉正则表达式语言，所以本节将主要解释正则表达式和相关的 .NET 类。如果您很熟悉正则表达式，就可以跳过本节，学习 .NET 基类的引用。注意，.NET 正则表达式引擎是为兼容 Perl 5 的正则表达式而设计的，但有一些新特性。

### 8.2.1 正则表达式概述

正则表达式语言是一种专门用于字符串处理的语言。它包含两个功能：

- 一组用于标识字符类型的转义代码。您可能很熟悉 DOS 表达式中的 \* 字符表示任意子字符串(例如，DOS 命令 Dir Re\* 会列出所有名称以 Re 开头的文件)。正则表达式使用与 \* 类似的许多序列来表示“任意一个字符”、“一个单词”、“一个可选的字符”等。
- 一个系统。在搜索操作中，它把子字符串和中间结果的各个部分组合起来。

使用正则表达式，可以对字符串执行许多复杂而高级的操作，例如：

- 区分(可以是标记或删除)字符串中所有重复的单词，例如，把 The computer books books 转换为 The computer books。
- 把所有单词都转换为标题格式，例如把 this is a Title 转换为 This Is A Title。
- 把长于 3 个字符的所有单词都转换为标题格式，例如把 this is a Title 转换为 This is a Title。
- 确保句子有正确的大写形式。
- 区分 URI 的各个元素(例如 http://www.wrox.com，提取出协议、计算机名、文件名等)。

当然，这些都是可以在 C# 中用 `System.String` 和 `System.Text.StringBuilder` 的各种方法执行的任务。但是，在一些情况下，还需要编写相当多的 C# 代码。如果使用正则表达式，这些代码一般可以压缩为几行代码。实际上，是实例化了一个对象 `System.Text.RegularExpressions.RegEx` (甚至更简单：调用静态的 `RegEx()` 方法)，给它传送要处理的字符串和一个正则表达式(这是一个字符串，包含用正则表达式语言编写的指令)，就可以了。

正则表达式字符串初看起来像是一般的字符串，但其中包含了转义序列和有特定含义的其他字符。例如，序列 `\b` 表示一个字的开头和结尾(字的边界)，如果要表示正在查找以字符 `th` 开头的字，就可以编写正则表达式 `\bth` (即序列字边界是 `-t-h`)。如果要搜索所有以 `th` 结尾的字，就可以编写 `th\b` (序列 `t-h` 字边界)。但是，正则表达式要比这复杂得多，包括可以在搜索操作中找到存储部分文本的工具性程序。本节仅介绍正则表达式的功能。

#### 提示：

正则表达式的更多信息可参阅图书 *Beginning Regular Expressions* (ISBN: 978-0-7645-7489-4)。

假定应用程序需要把 US 电话号码转换为国际格式。在美国，电话号码的格式为 314-123-1234，常常写作 (314) 123-1234。在把这个国家格式转换为国际格式时，必须在电话号码的前面加上 +1 (美国的国家代码)，并给区号加上括号：+1 (314) 123-1234。在查找和替换时，这并不复杂，但如果要使用 `String` 类完成这个转换，就需要编写一些代码(这表示，必须使用 `System.String` 上的方法来编写代码)，而正则表达式语言可以构造一个短的字符串来表达上述含义。

所以，本节只有一个非常简单的示例，我们只考虑如何查找字符串中的某些子字符串，无须考虑如何修改它们。

### 8.2.2 RegularExpressionsPlayaround 示例

下面将开发一个小示例 `RegularExpressionsPlayaround`，执行并显示一些搜索的结果，说明正则表达式的一些特性，以及如何在 C# 中使用 .NET 正则表达式引擎。这个示例文档中使用的文本是引自另一本有关 ASP.NET 的 Wrox Press 书籍《ASP.NET 3.5 高级编程(第 5 版)》，清华大学出版社引进并出版：

```
string Text =
@"This comprehensive compendium provides a broad and thorough investigation of all
aspects of programming with ASP.NET. Entirely revised and updated for the 3.5
Release of .NET, this book will give you the information you need to master ASP.NET
and build a dynamic, successful, enterprise Web application.";
```

#### 注意：

不考虑换行，则上面的表达式是合法的 C# 代码——说明了使用字符串时应在前面加上符号 `@`。

我们把这个文本称为输入字符串。为了说明正则表达式 .NET 类，我们先进行一次纯文本的搜索，这次搜索不带任何转义序列或正则表达式命令。假定要查找所有的字符串 `ion`，把这个搜索字符串称为模式。使用正则表达式和上面声明的变量 `Text`，编写出下面的代码：

```
string Pattern = "ion";
MatchCollection Matches = Regex.Matches(Text, Pattern,
                                      RegexOptions.IgnoreCase |
                                      RegexOptions.ExplicitCapture);
foreach (Match NextMatch in Matches)
{
    Console.WriteLine(NextMatch.Index);
}
```

在这段代码中，使用了 `System.Text.RegularExpressions` 命名空间中 `Regex` 类的静态方法 `Matches()`。这个方法的参数是一些输入文本、一个模式和 `RegexOptions` 枚举中的一组可选标志。在本例中，指定所有的搜索都不应区分大小写。另一个标记 `ExplicitCapture` 改变了收集匹配的方式，对于本例，这样可以使搜索的效率更高，其原因详见后面的内容(尽管它还有这里没有介绍的其他用法)。`Matches()`返回 `MatchCollections` 对象的引用。匹配是一个技术术语，表示在表达式中查找模式实例的结果，用 `System.Text.RegularExpressions.Match` 来代表。因此，我们返回一个包含所有匹配的 `MatchCollection`，每个匹配都用一个 `Match` 对象来表示。在上面的代码中，只是在集合中迭代，使用 `Match` 类的 `Index` 属性，返回输入文本中匹配所在的索引。运行这段代码，将得到 3 个匹配。表 8-4 描述了 `RegexOptions` 枚举的一些选项。

表 8-4

成 员 名	说 明
CultureInvariant	指定忽略字符串的文化背景
ExplicitCapture	修改收集匹配的方式，确保把明确指定的匹配作为有效的搜索结果
IgnoreCase	忽略输入字符串的大小写
IgnorePatternWhitespace	在字符串中删除未转义的空白，使注释用英镑符号或短横线符号指定
Multiline	修改字符 <code>^</code> 和 <code>\$</code> ，把它们应用于每一行的开头和结尾，而不仅仅应用于整个字符串的开头和结尾
RightToLeft	从右到左地读取输入字符串，而不是从左到右地读取(适合于一些亚洲语言或其他以这种方式读取的语言)
Singleline	指定句点的含义( <code>.</code> )，它原来表示单行模式，现在改为匹配每个字符

除了一些新的.NET 基类外，其他内容都不是新的。但正则表达式的功能主要取决于模式字符串。原因是模式字符串不仅仅包含纯文本。如前所述，它还可以包含元字符和转义序列，其中元字符是给出命令的特定字符，而转义序列的工作方式与 C#的转义序列相同，它们都是以反斜杠开头的字符，具有特殊的含义。

例如，假定要找以 `n` 开头的字，就可以使用转义序列 `\b`，它表示一个字的边界(字的边界是以字母数字表中的某个字符开头，或者后面是一个空白字符或标点符号)。可以编写如下代码：

```
string Pattern = @"\bn";
MatchCollection Matches = Regex.Matches(Text, Pattern,
                                      RegexOptions.IgnoreCase |
                                      RegexOptions.ExplicitCapture);
```

注意字符串前面的符号`@`。要在运行时把`\b`传递给.NET 正则表达式引擎，反斜杠不应被



C#编译器解释为转义序列。如果要查找以序列 `ion` 结尾的字，可以使用下面的代码：

```
string Pattern = @"ion\b";
```

如果要查找以字母 `a` 开头，以序列 `ion` 结尾的所有字(在本例中仅有一个匹配 `application`)，就必须在上面的代码中添加一些内容。显然，我们需要一个以 `\ba` 开头，以 `ion\b` 结尾的模式，但中间的内容怎么办？需要告诉应用程序在 `a` 和 `ion` 中间的内容可以是任意长度的任意字符，只要这些字符不是空白即可。实际上，正确的模式如下所示。

```
string Pattern = @"ba\S*ion\b";
```

使用正则表达式要习惯的一点是，对像这样怪异的字符序列见怪不怪。但这个序列的工作是非常逻辑化的。转义序列 `\S` 表示任何不是空白的字符。`*`称为数量词，其含义是前面的字符可以重复任意次，包括 0 次。序列 `\S*`表示任意个不是空白的字符。因此，上面的模式匹配于以 `a` 开头，以 `ion` 结尾的任何单词。

表 8-5 是可以使用的一些主要的特定字符或转义序列，但这个表并不完整，完整的列表请参考 MSDN 文档。

表 8-5

符 号	含 义	示 例	匹配的示例
<code>^</code>	输入文本的开头	<code>^B</code>	<code>B</code> ，但只能是文本中的第一个字符
<code>\$</code>	输入文本的结尾	<code>X\$</code>	<code>X</code> ，但只能是文本中的最后一个字符
<code>.</code>	除了换行字符( <code>\n</code> )以外的所有单个字符	<code>i.ation</code>	<code>isation</code> 、 <code>ization</code>
<code>*</code>	可以重复 0 次或多次的前导字符	<code>ra*t</code>	<code>rt</code> 、 <code>rat</code> 、 <code>raat</code> 和 <code>raaat</code> 等
<code>+</code>	可以重复 1 次或多次的前导字符	<code>ra+t</code>	<code>rat</code> 、 <code>raat</code> 和 <code>raaat</code> 等(但不能是 <code>rt</code> )
<code>?</code>	可以重复 0 次或 1 次的前导字符	<code>ra?t</code>	只有 <code>rt</code> 和 <code>rat</code> 匹配
<code>\s</code>	任何空白字符	<code>\sa</code>	<code>[space]a</code> 、 <code>\ta</code> 、 <code>\na</code> ( <code>\t</code> 和 <code>\n</code> 与 C#的 <code>\t</code> 和 <code>\n</code> 含义相同)
<code>\S</code>	任何不是空白的字符	<code>\SF</code>	<code>aF</code> 、 <code>rF</code> 、 <code>cF</code> 、但不能是 <code>\tf</code>
<code>\b</code>	字边界	<code>ion\b</code>	以 <code>ion</code> 结尾的任何字
<code>\B</code>	不是字边界的位置	<code>\BX\B</code>	字中间的任何 <code>X</code>

如果要搜索一个元字符，也可以通过带有反斜杠的转义字符来表示。例如，`\.`(一个句点)表示除了换行字符以外的任何字符，而 `\.`表示一个点。

可以把替换的字符放在方括号中，请求匹配包含这些字符。例如，`[l|c]`表示字符可以是 `l` 或 `c`。如果要搜索 `map` 或 `man`，可以使用序列 `ma[np]`。在方括号中，也可以指定一个范围，例如 `[a-z]`表示所有的小写字母，`[A-E]`表示 `A` 到 `E` 之间的所有大写字母，`[0-9]`表示一个数字。如果要搜索一个整数(该序列只包含 0 到 9 的字符)，就可以编写 `[0-9]+`(注意，使用 `+`字符表示至少要有这样一个数字，但可以有多多个数字，所以 `9`、`83` 和 `854` 等都是匹配的)。

### 8.2.3 显示结果

本节编写一个示例 `RegularExpressionsPlayaround`，看看正则表达式的工作方式。

该示例的核心是一个方法 `WriteMatches()`，它把 `MatchCollection` 中的所有匹配以比较详细的方式显示出来。对于每个匹配，它都会显示该匹配在输入字符串中的索引、匹配的字符串和一个略长的字符串，其中包含匹配和输入文本中至多 10 个外围字符，其中至多有 5 个字符放在匹配的前面，至多 5 个字符放在匹配的后面(如果匹配的位置在输入文本的开头或结尾 5 个字符内，则结果中匹配前后的字符就会少于 5 个)。换言之，如果要匹配的单词是 `messaging`，靠近输入文本末尾的匹配应是“and messaging of d”，匹配的前后各有 5 个字符，但位于输入文本的最后一个字上的匹配就应是“g of data”——匹配的后面只有一个字符。因为在该字符的后面是字符串的结尾。这个长字符串可以更清楚地表明正则表达式是在什么地方查找到匹配的：

```
static void WriteMatches(string text, MatchCollection matches)
{
    Console.WriteLine("Original text was: \n\n" + text + "\n");
    Console.WriteLine("No. of matches: " + matches.Count);
    foreach (Match nextMatch in matches)
    {
        int Index = nextMatch.Index;
        string result = nextMatch.ToString();
        int charsBefore = (Index < 5) ? Index : 5;
        int fromEnd = text.Length - Index - result.Length;
        int charsAfter = (fromEnd < 5) ? fromEnd : 5;
        int charsToDisplay = charsBefore + charsAfter + result.Length;

        Console.WriteLine("Index: {0}, \tString: {1}, \t{2}",
            Index, result,
            text.Substring(Index - charsBefore, charsToDisplay));
    }
}
```

在这个方法中，处理过程是确定在较长的子字符串中有多少个字符可以显示，而无需超出输入文本的开头或结尾。注意在 `Match` 对象上使用了另一个属性 `Value`，它包含标识该匹配的字符串。而且，`RegularExpressionsPlayaround` 只包含名为 `Find1`、`Find2` 等的方法，这些方法根据本节中的示例执行某些搜索操作。例如，`Find2` 在字开头处查找以 `a` 开头的字符串：

```
static void Find2()
{
    string text =
        @"This comprehensive compendium provides a broad and thorough investigation of all aspects of programming with ASP.NET. Entity revised and updated for the 3.5 Release of .NET, this book will give you the information you need to master ASP.NET And build a dynamic, successful, enterprise Web application.";
    string pattern = @"\ba";
    MatchCollection matches = Regex.Matches(text, pattern,
        RegexOptions.IgnoreCase);
    WriteMatches(text, matches);
}
```

下面是一个简单的 `Main()` 方法，可以编辑并选择一个 `Find<n>()` 方法：

```
static void Main()
{
```

```
Find1();
Console.ReadLine();
}
```

这段代码还使用了命名空间 `Regex`:

```
using System;
using System.Text.RegularExpressions;
```

运行带有 `Find1()` 方法的示例, 得到如下所示的结果:

```
RegexPlayaround
Original text was:
```

```
This comprehensive compendium provides a broad and thorough investigation of all
aspects of programming with ASP.NET. Entity revised and updated for the 3.5
Release of .NET, this book will give you the information you need to master ASP.NET
And build a dynamic, successful, enterprise Web application.
```

```
No. of matches: 1
Index: 291, String: application, Web application.
```

#### 8.2.4 匹配、组合和捕获

正则表达式的一个很好的特性是可以把字符组合起来, 其方式与 C# 中的复合语句一样。在 C# 中, 可以把任意数量的语句放在花括号中, 把它们组合在一起。其结果就像一个复合语句那样。在正则表达式模式中, 也可以把任何字符组合起来(包括元字符和转义序列), 像处理一个字符那样处理它们。唯一的区别是要使用圆括号, 而不是花括号, 得到的序列称为一个组。

例如, 模式 `(an)+` 定位序列 `an` 的任意重复。量词 `+` 只应用于它前面的一个字符, 但因为我们把字符组合起来了, 所以它现在把重复的 `an` 作为一个单元来对待。`(an)+` 应用到输入文本 `bananas came to Europe late in the annals of history` 上, 会从 `bananas` 中选择出 `anan`。另一方面, 如果使用 `an+`, 则程序将从 `annals` 中选择 `ann`, 从 `bananas` 中选择出两个 `an`。表达式 `(an)+` 可以提取出 `an`、`anan`、`ananan` 等, 而表达式 `an+` 可以提取出 `an`、`ann`、`annn` 等。

**注意:**

在上面的示例中, 为什么 `(an)+` 从 `banana` 中选择的是 `anan`, 而没有把单个的 `an` 作为一个匹配? 因为匹配是不能重叠的。如果有可能重叠, 在默认情况下就选择最长的匹配。

但是, 组的功能要比这强大得多。在默认情况下, 把模式的一部分组合为一个组时, 就要求正则表达式引擎按照这个组来匹配, 或按照整个模式来匹配。换言之, 可以把组当作一个要匹配的模式来返回, 如果要把字符串分解为各个部分, 这种模式就是非常有效的。

例如, URI 的格式是 `<protocol>://<address>[:<port>]`, 其中端口是可选的。它的一个示例是 `http://www.wrox.com:4355`。假定要从一个 URI 中提取协议、地址和端口, 而且紧邻 URI 的后面可能有空白(但没有标点符号), 就可以使用下面的表达式:

```
\b(\S+)://(\S+)(?:(\S+))?\b
```

该表达式的工作方式如下: 首先, 前导和尾部的 `\b` 序列确保只需要考虑完全是字的文本部分, 在这个文本部分中, 第一组 `(\S+)://` 会选择一个或多个不是空白的字符, 其后是 `://`。在 HTTP

URI 的开头会选择出 `http://`。花括号表示把 `http` 存储为一个组。后面的序列 `(\S+)` 则在上述 URI 中选择 `www.wrox.com`，这个组在遇到词的结尾(结束 `\b`)时或标记另一个组的冒号(`:`)时结束。

下一个组选择端口(本例是:4355)。后面的 `?` 表示这个组在匹配中是可选的，如果没有:xxxx，也不会妨碍匹配的标记。这是非常重要的，因为端口号在 URI 中一般不指定，实际上，在大多数情况下，URI 是没有端口号的。但是，事情会比较复杂。我们希望指定冒号可以出现，也可以不出现，但不希望把这个冒号也存储在组中。为此，可以嵌套两个组：内部的 `(\S+)` 组选择冒号后面的内容(本例中是 4355)，外面的组包含内部的组，前面是一个冒号，该组又在序列 `?:` 的后面。这个序列表示该组不应保存(只需要保存 4355，不需要保存:4355)。不要把这两个冒号混淆了，第一个冒号是序列 `?:` 的一部分，表示不保存这个组，第二个冒号是要搜索的文本。

在下面的字符串上运行该模式，得到的匹配是 `http://www.wrox.com`。

```
Hey I've just found this amazing URI at http:// what was it -- oh yes http://www.wrox.com
```

在这个匹配中，找到了刚才提及的 3 个组，还有第四个组表示匹配本身。理论上，每个组都可以选择 0 次、1 次或多次匹配。单个的匹配就称为捕获。在第一个组 `(\S+)` 中，有一个捕获 `http`，第二个组也有一个捕获 `www.wrox.com`，但第三个组没有捕获，因为在这个 URI 中没有端口号。

注意，该字符串包含第二个 `http://`。虽然它匹配于第一个组，但不会被搜索出来，因为整个搜索表达式不匹配于这部分文本。

前面没有介绍使用组和捕获的任何 C# 示例，下面提到的 .NET 类 `RegularExpressions` 就通过 `Group` 和 `Capture` 类支持组和捕获。`GroupCollection` 和 `CaptureCollection` 分别表示组和捕获的集合，`Match` 类有一个方法 `Groups()`，它返回相应的 `GroupCollection` 对象，`Group` 类也相应地执行一个方法 `Captures()`，它返回 `CaptureCollection` 对象。这些对象之间的关系如图 8-3 所示。

把一些字符组合起来后，每次都会返回一个 `Group` 对象。如果只是希望把一些字符组合起来，作为搜索模式的一部分，实例化对象就会浪费相当大的系统开销。对于单个的组，可以用以字符序列 `?:` 开头，禁止实例化对象，就像 URI 示例那样。而对于所有的组，可以在 `Regex.Matches()` 方法上指定 `RegexOptions.ExplicitCaptures` 标志，如同前面的示例那样。

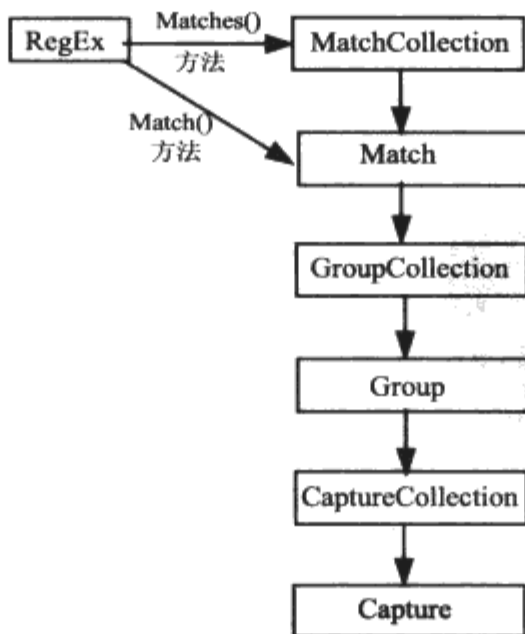


图 8-3

## 8.3 小结

在使用.NET Framework 时,可用的数据类型相当多。在应用程序(特别是关注数据提交和检索的应用程序)中,最常用的一个类型就是 `String` 数据类型。`String` 非常重要,这也是本书用一整章的篇幅介绍如何在应用程序中使用和处理 `String` 数据类型的原因。

过去在使用字符串时,常常需要通过连接来分解字符串,而在.NET Framework 中,可以使用 `StringBuilder` 类完成许多这类任务,而且性能更好。

最后,使用正则表达式进行高级的字符串处理是搜索和验证字符串的一种极佳工具。

下一章介绍 C# 的一个强大功能——泛型。



# 第 9 章

## 泛 型

CLR 2.0 的一个新特性是泛型。在 CLR 1.0 中，要创建一个灵活的类或方法，但该类或方法在编译期间不知道使用什么类，就必须以 `Object` 类为基础。而 `Object` 类在编译期间没有类型安全性，因此必须进行强制类型转换。另外，给值类型使用 `Object` 类会有性能损失。

CLR 2.0(.NET 3.5 基于 CLR 2.0)提供了泛型。有了泛型，就不再需要 `Object` 类了。泛型类使用泛型类型，并可以根据需要用特定的类型替换泛型类型。这就保证了类型安全性：如果某个类型不支持泛型类，编译器就会生成错误。

泛型是一个很强大的特性，对于集合类而言尤其如此。.NET 1.0 中的大多数集合类都基于 `Object` 类型。.NET 从 2.0 开始提供了实现为泛型的新集合类。

泛型不仅限于类，本章还将介绍用于委托、接口和方法的泛型。

本章的主要内容如下：

- 泛型概述
- 创建泛型类
- 泛型类的特性
- 泛型接口
- 泛型方法
- 泛型委托
- Framework 的其他泛型类型

### 9.1 概述

泛型并不是一个全新的结构，其他语言中有类似的概念。例如，C++模板就与泛型相当。但是，C++模板和.NET 泛型之间有一个很大的区别。对于 C++模板，在用特定的类型实例化模板时，需要模板的源代码。相反，泛型不仅是 C#语言的一种结构，而且是 CLR 定义的。所以，即使泛型类是在 C#中定义的，也可以在 Visual Basic 中用一个特定的类型实例化该泛型。

下面介绍泛型的优点和缺点，尤其是：

- 性能
- 类型安全性
- 二进制代码重用
- 代码的扩展
- 命名约定

### 9.1.1 性能

泛型的一个主要优点是性能。第 10 章介绍了 `System.Collections` 和 `System.Collections.Generic` 命名空间的泛型和非泛型集合类。对值类型使用非泛型集合类，在把值类型转换为引用类型，和把引用类型转换为值类型时，需要进行装箱和拆箱操作。

注意：

装箱和拆箱详见第 6 章，这里仅简要复习一下这些术语。

值类型存储在堆栈上，引用类型存储在堆上。C# 类是引用类型，结构是值类型。.NET 很容易把值类型转换为引用类型，所以可以在需要对象(对象是引用类型)的任意地方使用值类型。例如，`int` 可以赋予一个对象。从值类型转换为引用类型称为装箱。如果方法需要把一个对象作为参数，而且传送了一个值类型，装箱操作就会自动进行。另一方面，装箱的值类型可以使用拆箱操作转换为值类型。在拆箱时，需要使用类型转换运算符。

下面的例子显示了 `System.Collections` 命名空间中的 `ArrayList` 类。`ArrayList` 存储对象，`Add()` 方法定义为需要把一个对象作为参数，所以要装箱一个整数类型。在读取 `ArrayList` 中的值时，要进行拆箱，把对象转换为整数类型。可以使用类型转换运算符把 `ArrayList` 集合的第一个元素赋予变量 `i1`，在访问 `int` 类型的变量 `i2` 的 `foreach` 语句中，也要使用类型转换运算符：

```
ArrayList list = new ArrayList();
list.Add(44); // boxing - convert a value type to a reference type

int i1 = (int)list[0]; // unboxing - convert a reference type to a value type

foreach (int i2 in list)
{
    Console.WriteLine(i2); // unboxing
}
```

装箱和拆箱操作很容易使用，但性能损失比较大，迭代许多项时尤其如此。

`System.Collections.Generic` 命名空间中的 `List<T>` 类不使用对象，而是在使用时定义类型。在下面的例子中，`List<T>` 类的泛型类型定义为 `int`，所以 `int` 类型在 JIT 编译器动态生成的类中使用，不再进行装箱和拆箱操作：

```
List<int> list = new List<int>();
list.Add(44); // no boxing - value types are stored in the List<int>

int i1 = list[0]; // no unboxing, no cast needed

foreach (int i2 in list)
{
    Console.WriteLine(i2);
}
```

### 9.1.2 类型安全

泛型的另一个特性是类型安全。与 `ArrayList` 类一样，如果使用对象，可以在这个集合中添加任意类型。下面的例子在 `ArrayList` 类型的集合中添加一个整数、一个字符串和一个 `MyClass`

类型的对象：

```
ArrayList list = new ArrayList();
list.Add(44);
list.Add("mystring");
list.Add(new MyClass());
```

如果这个集合使用下面的 foreach 语句迭代，而该 foreach 语句使用整数元素来迭代，编译器就会编译这段代码。但并不是集合中的所有元素都可以转换为 int，所以会出现一个运行异常：

```
foreach (int i in list)
{
    Console.WriteLine(i);
}
```

错误应尽早发现。在泛型类 List<T> 中，泛型类型 T 定义了允许使用的类型。有了 List<int> 的定义，就只能把整数类型添加到集合中。编译器不会编译这段代码，因为 Add() 方法的参数无效：

```
List<int> list = new List<int>();
list.Add(44);
list.Add("mystring"); // compile time error
list.Add(new MyClass()); // compile time error
```

### 9.1.3 二进制代码的重用

泛型允许更好地重用二进制代码。泛型类可以定义一次，用许多不同的类型实例化。不需要像 C++ 模板那样访问源代码。

例如，System.Collections.Generic 命名空间中的 List<T> 类用一个 int、一个字符串和一个 MyClass 类型实例化：

```
List<int> list = new List<int>();
list.Add(44);

List<string> stringList = new List<string>();
stringList.Add("mystring");

List<MyClass> myclassList = new List<MyClass>();
myclassList.Add(new MyClass());
```

泛型类型可以在一种语言中定义，在另一种 .NET 语言中使用。

### 9.1.4 代码的扩展

在用不同的类型实例化泛型时，会创建多少代码？

因为泛型类的定义会放在程序集中，所以用某个类型实例化泛型类不会在 IL 代码中复制这些类。但是，在 JIT 编译器把泛型类编译为内部码时，会给每个值类型创建一个新类。引用类型共享同一个内部类的所有实现代码。这是因为引用类型在实例化的泛型类中只需要 4 字节的内存单元(32 位系统)，就可以引用一个引用类型。值类型包含在实例化的泛型类的内存中。而每个值类型对内存的要求都不同，所以要为每个值类型实例化一个新类。

### 9.1.5 命名约定

如果在程序中使用泛型，区分泛型类型和非泛型类型会有一定的帮助。下面是泛型类型的命名规则：

- 泛型类型的名称用字母 T 作为前缀。
- 如果没有特殊的要求，泛型类型允许用任意类替代，且只使用了一个泛型类型，就可以用字母 T 作为泛型类型的名称。

```
public class List<T> { }
public class LinkedList<T> { }
```

- 如果泛型类型有特定的要求(例如必须实现一个接口或派生于基类)，或者使用了两个或多个泛型类型，就应给泛型类型使用描述性的名称：

```
public delegate void EventHandler<TEventArgs>(object sender, TEventArgs e);
public delegate TOutput Converter<TInput, TOutput>(TInput from);
public class SortedList<TKey, TValue> { }
```

## 9.2 创建泛型类

首先介绍一个一般的、非泛型的简化链表类，它可以包含任意类型的对象，以后再把这个类转化为泛型类。

在链表中，一个元素引用其后的下一个元素。所以必须创建一个类，将对象封装在链表中，引用下一个对象。类 `LinkedListNode` 包含一个对象 `value`，它用构造函数初始化，还可以用 `Value` 属性读取。另外，`LinkedListNode` 类包含对链表中下一个元素和上一个元素的引用，这些元素都可以从属性中访问。

```
public class LinkedListNode
{
    private object value;
    public LinkedListNode(object value)
    {
        this.value = value;
    }

    public object Value
    {
        get { return value; }
    }

    private LinkedListNode next;
    public LinkedListNode Next
    {
        get { return next; }
        internal set { next = value; }
    }

    private LinkedListNode prev;
```



```

public LinkedListNode Prev
{
    get { return prev; }
    internal set { prev = value; }
}
}

```

LinkedList 类包含 LinkedListNode 类型的 first 和 last 字段，它们分别标记了链表的头尾。AddLast() 方法在链表尾添加一个新元素。首先创建一个 LinkedListNode 类型的对象。如果链表是空的，则 first 和 last 字段就设置为该新元素；否则，就把新元素添加为链表中的最后一个元素。执行 GetEnumerator() 方法时，可以用 foreach 语句迭代链表。GetEnumerator() 方法使用 yield 语句创建一个枚举器类型。

提示：

yield 语句参见第 5 章。

```

public class LinkedList : IEnumerable
{
    private LinkedListNode first;
    public LinkedListNode First
    {
        get { return first; }
    }

    private LinkedListNode last;
    public LinkedListNode Last
    {
        get { return last; }
    }

    public LinkedListNode AddLast(object node)
    {
        LinkedListNode newNode = new LinkedListNode(node);
        if (first == null)
        {
            first = newNode;
            last = first;
        }
        else
        {
            last.Next = newNode;
            last = newNode;
        }
        return newNode;
    }

    public IEnumerator GetEnumerator()
    {
        LinkedListNode current = first;
        while (current != null)
        {
            yield return current.Value;
            current = current.Next;
        }
    }
}

```

现在可以给任意类型使用 LinkedList 类了。在下面的代码中，实例化了一个新 LinkedList 对



象，添加了两个整数类型和一个字符串类型。整数类型要转换为一个对象，所以执行装箱操作，如前面所述。在 `foreach` 语句中执行拆箱操作。在 `foreach` 语句中，链表中的元素被强制转换为整数，所以对于链表中的第三个元素，会发生一个运行异常，因为它转换为 `int` 时会失败。

```
LinkedList list1 = new LinkedList();
list1.AddLast(2);
list1.AddLast(4);
list1.AddLast("6");
foreach (int i in list1)
{
    Console.WriteLine(i);
}
```

下面创建链表的泛型版本。泛型类的定义与一般类类似，只是要使用泛型类型声明。之后，泛型类型就可以在类中用作一个字段成员，或者方法的参数类型。`LinkedListNode` 类用一个泛型类型 `T` 声明。字段 `value` 的类型是 `T`，而不是 `object`。构造函数和 `Value` 属性也变为接受和返回 `T` 类型的对象。也可以返回和设置泛型类型，所以属性 `Next` 和 `Prev` 的类型是 `LinkedListNode<T>`。

```
public class LinkedListNode<T>
{
    private T value;
    public LinkedListNode(T value)
    {
        this.value = value;
    }

    public T Value
    {
        get { return value; }
    }

    private LinkedListNode<T> next;
    public LinkedListNode<T> Next
    {
        get { return next; }
        internal set { next = value; }
    }

    private LinkedListNode<T> prev;
    public LinkedListNode<T> Prev
    {
        get { return prev; }
        internal set { prev = value; }
    }
}
```

下面的代码把 `LinkedList` 类也改为泛型类。`LinkedList<T>` 包含 `LinkedListNode<T>` 元素。`LinkedList` 中的类型 `T` 定义了类型 `T` 的包含字段 `first` 和 `last`。`AddLast()` 方法现在接受类型 `T` 的参数，实例化 `LinkedListNode<T>` 类型的对象。

`IEnumerable` 接口也有一个泛型版本 `IEnumerable<T>`。`IEnumerable<T>` 派生于 `IEnumerable`，添加了返回 `IEnumerator<T>` 的 `GetEnumerator()` 方法，`LinkedList<T>` 执行泛型接口 `IEnumerable<T>`。

提示:

枚举、接口 IEnumerable 和 IEnumerator 详见第 5 章。

```
public class LinkedList<T> : IEnumerable<T>
{
    private LinkedListNode<T> first;
    public LinkedListNode<T> First
    {
        get { return first; }
    }

    private LinkedListNode<T> last;
    public LinkedListNode<T> Last
    {
        get { return last; }
    }

    public LinkedListNode<T> AddLast(T node)
    {
        LinkedListNode<T> newNode = new LinkedListNode<T>(node);
        if (first == null)
        {
            first = newNode;
            last = first;
        }
        else
        {
            last.Next = newNode;
            last = newNode;
        }
        return newNode;
    }

    public IEnumerator<T> GetEnumerator()
    {
        LinkedListNode<T> current = first;

        while (current != null)
        {
            yield return current.Value;
            current = current.Next;
        }
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}
```

使用泛型类 `LinkedList<T>`，可以用 `int` 类型实例化它，且无需装箱操作。如果不使用 `AddLast()` 方法传送 `int`，就会出现一个编译错误。使用泛型 `IEnumerable<T>`，`foreach` 语句也是类型安全的，如果 `foreach` 语句中的变量不是 `int`，也会出现一个编译错误。

```
LinkedList<int> list2 = new LinkedList<int>();
list2.AddLast(1);
list2.AddLast(3);
list2.AddLast(5);
```

```
foreach (int i in list2)
{
    Console.WriteLine(i);
}
```

同样，可以给泛型 `LinkedList<T>` 使用 `string` 类型，将字符串传送给 `AddLast()` 方法。

```
LinkedList<string> list3 = new LinkedList<string>();
list3.AddLast("2");
list3.AddLast("four");
list3.AddLast("foo");

foreach (string s in list3)
{
    Console.WriteLine(s);
}
```

提示：

每个处理对象类型的类都可以有泛型实现方式。另外，如果类使用了继承，泛型非常有助于去除类型转换操作。

### 9.3 泛型类的特性

在创建泛型类时，需要一些其他 C# 关键字。例如，不能把 `null` 赋予泛型类型。此时，可以使用 `default` 关键字。如果泛型类型不需要 `Object` 类的功能，但需要调用泛型类上的某些特定方法，就可以定义约束。

本节讨论如下主题：

- 默认值
- 约束
- 继承
- 静态成员

下面开始一个使用泛型文档管理器的示例。文档管理器用于从队列中读写文档。先创建一个新的控制台项目 `DocumentManager`，添加类 `DocumentManager<T>`。`AddDocument()` 方法将一个文档添加到队列中。如果队列不为空，`IsDocumentAvailable` 只读属性就返回 `true`。

```
using System;
using System.Collections.Generic;

namespace Wrox.ProCSharp.Generics
{
    public class DocumentManager<T>
    {
        private readonly Queue<T> documentQueue = new Queue<T>();

        public void AddDocument(T doc)
        {
            lock (this)
            {
                documentQueue.Enqueue(doc);
            }
        }
    }
}
```

```

        public bool IsDocumentAvailable
        {
            get { return documentQueue.Count > 0; }
        }
    }
}

```

### 9.3.1 默认值

现在给 `DocumentManager<T>` 类添加一个 `GetDocument()` 方法。在这个方法中，给类型 `T` 指定 `null`。但是，不能把 `null` 赋予泛型类型。原因是泛型类型也可以实例化为值类型，而 `null` 只能用于引用类型。为了解决这个问题，可以使用 `default` 关键字。通过 `default` 关键字，将 `null` 赋予引用类型，将 `0` 赋予值类型。

```

public T GetDocument()
{
    T doc = default(T);
    lock (this)
    {
        doc = documentQueue.Dequeue();
    }
    return doc;
}

```

注意：

`default` 关键字根据上下文可以有多种含义。switch 语句使用 `default` 定义默认情况。在泛型中，根据泛型类型是引用类型还是值类型，`default` 关键字用于将泛型类型初始化为 `null` 或 `0`。

### 9.3.2 约束

如果泛型类需要调用泛型类型上的方法，就必须添加约束。对于 `DocumentManager<T>`，文档的标题应在 `DisplayAllDocuments()` 方法中显示。

`Document` 类执行带有 `Title` 和 `Content` 属性的 `IDocument` 接口：

```

public interface IDocument
{
    string Title { get; set; }
    string Content { get; set; }
}

public class Document : IDocument
{
    public Document()
    {
    }

    public Document(string title, string content)
    {
        this.title = title;
        this.content = content;
    }
}

```

```

    public string Title { get; set; }
    public string Content { get; set; }
}

```

要使用 `DocumentManager<T>` 类显示文档，可以将类型 `T` 强制转换为 `IDocument` 接口，以显示标题：

```

public void DisplayAllDocuments()
{
    foreach (T doc in documentQueue)
    {
        Console.WriteLine((IDocument)doc).Title);
    }
}

```

问题是，如果类型 `T` 没有执行 `IDocument` 接口，这个类型转换就会生成一个运行异常。最好给 `DocumentManager<TDocument>` 类定义一个约束：`TDocument` 类型必须执行 `IDocument` 接口。为了在泛型类型的名称中指定该要求，将 `T` 改为 `TDocument`。`where` 子句指定了执行 `IDocument` 接口的要求。

```

public class DocumentManager<TDocument>
    where TDocument : IDocument
{

```

这样，就可以编写 `foreach` 语句，让类型 `T` 包含属性 `Title` 了。Visual Studio IntelliSense 和编译器都会提供这个支持。

```

    public void DisplayAllDocuments()
    {
        foreach (TDocument doc in documentQueue)
        {
            Console.WriteLine(doc.Title);
        }
    }
}

```

在 `Main()` 方法中，`DocumentManager<T>` 类用 `Document` 类型实例化，而 `Document` 类型执行了需要的 `IDocument` 接口。接着添加和显示新文档，检索其中一个文档：

```

static void Main()
{
    DocumentManager<Document> dm = new DocumentManager<Document>();
    dm.AddDocument(new Document("Title A", "Sample A"));
    dm.AddDocument(new Document("Title B", "Sample B"));

    dm.DisplayAllDocuments();

    if (dm.IsDocumentAvailable)
    {
        Document d = dm.GetDocument();
        Console.WriteLine(d.Content);
    }
}

```

`DocumentManager` 现在可以处理任何执行了 `IDocument` 接口的类。

在示例应用程序中，介绍了接口约束。泛型还有几种约束类型，如表 9-1 所示。



表 9-1

约 束	说 明
where T : struct	使用结构约束，类型 T 必须是值类型
where T : class	类约束指定，类型 T 必须是引用类型
where T : IFoo	指定类型 T 必须执行接口 IFoo
where T : Foo	指定类型 T 必须派生于基类 Foo
where T : new()	这是一个构造函数约束，指定类型 T 必须有一个默认构造函数
where T : U	这个约束也可以指定，类型 T1 派生于泛型类型 T2。该约束也称为裸类型约束

**注意：**  
在 CLR 2.0 中，只能为默认构造函数定义约束，不能为其他构造函数定义约束。

使用泛型类型还可以合并多个约束。where T : IFoo, new()约束和 MyClass<T>声明指定，类型 T 必须执行 IFoo 接口，且必须有一个默认构造函数。

```
public class MyClass<T>
    where T : IFoo, new()
{
    //...
```

**提示：**  
在 C#中，where 子句的一个重要限制是，不能定义必须由泛型类型执行的运算符。运算符不能在接口中定义。在 where 子句中，只能定义基类、接口和默认构造函数。

9.3.3 继承

前面创建的 LinkedList<T>类执行了 IEnumerable<T>接口：

```
public class LinkedList<T> : IEnumerable<T>
{
    //...
```

泛型类型可以执行泛型接口，也可以派生于一个类。泛型类可以派生于泛型基类：

```
public class Base<T>
{
}

public class Derived<T> : Base<T>
{
}
```

其要求是必须重复接口的泛型类型，或者必须指定基类的类型，如下所示：

```
public class Base<T>
{
}

public class Derived<T> : Base<string>
{
}
```

于是，派生类可以是泛型类或非泛型类。例如，可以定义一个抽象的泛型基类，它在派生类中用一个具体的类型实现。这允许对特定类型执行特殊的操作：

```
public abstract class Calc<T>
{
    public abstract T Add(T x, T y);
    public abstract T Sub(T x, T y);
}
public class SimpleCalc : Calc<int>
{
    public override int Add(int x, int y)
    {
        return x + y;
    }

    public override int Sub(int x, int y)
    {
        return x - y;
    }
}
```

### 9.3.4 静态成员

泛型类的静态成员需要特别关注。泛型类的静态成员只能在类的一个实例中共享。下面看一个例子。StaticDemo<T>类包含静态字段 x：

```
public class StaticDemo<T>
{
    public static int x;
}
```

由于对一个 string 类型和一个 int 类型使用了 StaticDemo<T>类，所以存在两组静态字段：

```
StaticDemo<string>.x = 4;
StaticDemo<int>.x = 5;
Console.WriteLine(StaticDemo<string>.x); // writes 4
```

## 9.4 泛型接口

使用泛型可以定义接口，接口中的方法可以带泛型参数。在链表示例中，就执行了 IEnumerable<T>接口，它定义了 GetEnumerator()方法，以返回 IEnumerator<T>。对于.NET 1.0 中的许多非泛型接口，.NET 从 2.0 开始定义了新的泛型版本，例如 IComparable<T>：

```
public interface IComparable<T>
{
    int CompareTo(T other);
}
```

第 5 章中的非泛型接口 IComparable 需要一个对象，Person 类的 CompareTo()方法才能按姓氏给人员排序：

```
public class Person : IComparable
{
    ...
}
```

```

public int CompareTo(object obj)
{
    Person other = obj as Person;
    return this.lastname.CompareTo(other.lastname);
}
//...

```

执行泛型版本时，不再需要将 `object` 的类型强制转换为 `Person`：

```

public class Person : IComparable<Person>
{
    public int CompareTo(Person other)
    {
        return this.lastname.CompareTo(other.lastname);
    }
}
//...

```

## 9.5 泛型方法

除了定义泛型类之外，还可以定义泛型方法。在泛型方法中，泛型类型用方法声明来定义。`Swap<T>`方法把 `T` 定义为泛型类型，用于两个参数和一个变量 `temp`：

```

void Swap<T>(ref T x, ref T y)
{
    T temp;
    temp = x;
    x = y;
    y = temp;
}

```

把泛型类型赋予方法调用，就可以调用泛型方法：

```

int i = 4;
int j = 5;
Swap<int>(ref i, ref j);

```

但是，因为 C#编译器会通过调用 `Swap` 方法来获取参数的类型，所以不需要把泛型类型赋予方法调用。泛型方法可以像非泛型方法那样调用：

```

int i = 4;
int j = 5;
Swap(ref i, ref j);

```

下面的例子使用泛型方法累加集合中的所有元素。为了说明泛型方法的功能，下面的 `Account` 类包含 `name` 和 `balance`：

```

public class Account
{
    private string name;
    public string Name
    {
        get
        {
            return name;
        }
    }
}

```

```

private decimal balance;
public decimal Balance
{
    get
    {
        return balance;
    }
}

public Account(string name, Decimal balance)
{
    this.name = name;
    this.balance = balance;
}
}

```

应累加结余的所有账目操作都添加到 List<Account>类型的账目列表中:

```

List<Account> accounts = new List<Account>();
accounts.Add(new Account("Christian", 1500));
accounts.Add(new Account("Sharon", 2200));
accounts.Add(new Account("Katie", 1800));

```

累加所有 Account 对象的传统方式是用 foreach 语句迭代所有的 Account 对象, 如下所示。foreach 语句使用 IEnumerable 接口迭代集合的元素, 所以 AccumulateSimple()方法的参数是 IEnumerable 类型。这样, AccumulateSimple()方法就可以用于所有实现 IEnumerable 接口的集合类。在这个方法的实现代码中, 直接访问 Account 对象的 Balance 属性:

```

public static class Algorithm
{
    public static decimal AccumulateSimple(IEnumerable e)
    {
        decimal sum = 0;
        foreach (Account a in e)
        {
            sum += a.Balance;
        }
        return sum;
    }
}

```

Accumulate()方法的调用方式如下:

```

decimal amount = Algorithm.AccumulateSimple(accounts);

```

第一个实现代码的问题是, 它只能用于 Account 对象。使用泛型方法就可以避免这个问题。

Accumulate()方法的第二个版本接受实现了 IAccount 接口的任意类型。如前面的泛型类所述, 泛型类型可以用 where 子句来限制。这个子句也可以用于泛型方法。Accumulate()方法的参数改为 IEnumerable<T>。IEnumerable<T>是 IEnumerable 接口的泛型版本, 由泛型集合类实现。

```

public static decimal Accumulate<TAccount>(IEnumerable<TAccount> coll)
    where TAccount : IAccount
{
    decimal sum = 0;

```

```
foreach (TAccount a in coll)
{
    sum += a.Balance;
}
return sum;
}
```

Account 类现在重构为执行接口 IAccount:

```
public class Account : IAccount
{
    //...
```

IAccount 接口定义了只读属性 Balance 和 Name:

```
public interface IAccount
{
    decimal Balance { get; }
    string Name { get; }
}
```

将 Account 类型定义为泛型类型参数, 就可以调用新的 Accumulate()方法:

```
decimal amount = Algorithm.Accumulate<Account>(accounts);
```

因为编译器会从方法的参数类型中自动推断出泛型类型参数, 所以以如下方式调用 Accumulate()方法是有效的:

```
decimal amount = Algorithm.Accumulate(accounts);
```

泛型类型实现 IAccount 接口的要求过于严厉。这个要求可以使用泛型委托来改变。在下一节中, Accumulate()方法将改为独立于任何接口。

## 9.6 泛型委托

如第 7 章所述, 委托是类型安全的方法引用。通过泛型委托, 委托的参数可以在以后定义。

.NET Framework 定义了一个泛型委托 EventHandler, 它的第二个参数是 EventArgs 类型, 所以不再需要为每个新参数类型定义新委托了。

```
public sealed delegate void EventHandler<EventArgs>(object sender, EventArgs e)
where EventArgs : EventArgs
```

### 9.6.1 执行委托调用的方法

把 Accumulate()方法改为有两个泛型类型。TInput 是要累加的对象类型, TSummary 是返回类型。Accumulate 的第一个参数是 IEnumerable<T>接口, 这与以前相同。第二个参数需要 Action 委托引用一个方法, 来累加所有的结余。

在实现代码中, 现在给每个元素调用 Action 委托引用的方法, 再返回计算的总和:

```
public delegate TSummary Action<TInput, TSummary>(TInput t, TSummary u);

public static TSummary Accumulate<TInput, TSummary>(IEnumerable<TInput> coll,
```



```

        Action<TInput, TSummary> action)
    {
        TSummary sum = default(TSummary);

        foreach (TInput input in coll)
        {
            sum = action(input, sum);
        }
        return sum;
    }

```

Accumulate 方法可以通过匿名方法调用，该匿名方法指定，账目的结余应累加到第二个 Action 类型的参数中：

```

decimal amount = Algorithm.Accumulate<Account, decimal>(
    accounts,
    delegate(Account a, decimal d)
    { return a.Balance + d; });

```

除了使用匿名方法之外，还可以使用λ表达式把它传送给第二个参数：

```

decimal amount = Algorithm.Accumulate < Account, decimal > (
    accounts, (a, d) => a.Balance + d);

```

提示：

匿名方法和λ表达式参见第7章。

如果 Account 结余的累加需要进行多次，就可以把该功能放在一个 AccountAdder()方法中：

```

static decimal AccountAdder(Account a, decimal d)
{
    return a.Balance + d;
}

```

联合使用 AccountAdder 方法和 Accumulate 方法：

```

decimal amount = Algorithm.Accumulate<Account, decimal>(
    accounts, AccountAdder);

```

Action 委托引用的方法可以实现任何逻辑。例如，可以进行乘法操作，而不是加法操作。

Accumulate()方法和 AccumulateIf()方法一起使用，会更灵活。在 AccumulateIf()中，使用了另一个 Predicate<T>类型的参数。Predicate<T>委托引用的方法会检查某个账目是否应累加进去。在 foreach 语句中，只有谓词 match 返回 true，才会调用 action 方法：

```

public static TSummary AccumulateIf<TInput, TSummary>(
    IEnumerable<TInput> coll,
    Action<TInput, TSummary> action,
    Predicate<TInput> match)
{
    TSummary sum = default(TSummary);

    foreach (TInput a in coll)
    {
        if (match(a))
        {
            sum = action(a, sum);
        }
    }
}

```

```
    }  
  
    return sum;  
}
```

调用 `AccumulateIf()`方法可以实现累加和执行谓词。这里按照第二个λ表达式的定义 `a => a.Balance > 2000`，只累加结余大于 2000 的账目：

```
decimal amount = Algorithm.AccumulateIf < Account, decimal > (  
    accounts, (a, d) => a.Balance + d, a => a.Balance > 2000);
```

9.6.2 对 Array 类使用泛型委托

第 5 章使用 `Comparable` 和 `Comparer` 接口，演示了 `Array` 类的几个排序技术。从 .NET 2.0 开始，`Array` 类的一些方法把泛型委托类型用作参数。表 9-2 列出了这些方法、泛型类型和功能。

表 9-2

方 法	泛型参数类型	说 明
Sort()	int Comparison<T>(T x, T y)	Sort()方法定义了几个重载版本。其中一个重载版本需要一个 <code>Comparison&lt;T&gt;</code> 类型的参数。Sort()使用委托引用的方法对集合中的所有元素排序
ForEach()	void Action<T>(T obj)	ForEach()方法对集合中的每一项调用由 <code>Action&lt;T&gt;</code> 委托引用的方法
FindAll() Find() FindLast() FindIndex() FindLastIndex()	bool Predicate<T>(T match)	FindXXX()方法将 <code>Predicate&lt;T&gt;</code> 委托作为其参数。由委托引用的方法会调用多次，并一个接一个地传送集合中的元素。Find()方法在谓词第一次返回 <code>true</code> 时停止搜索，并返回这个元素。FindIndex()返回查找到的第一个元素的索引。FindLast()和 FindLastIndex()以逆序方式对集合中的元素调用谓词，因此返回最后一项或最后一个索引。FindAll()返回一个新列表，其中包含谓词为 <code>true</code> 的所有项
ConvertAll()	TOutput Converter<TInput, TOutput>(TInput input)	ConvertAll()方法给集合中的每个元素调用 <code>Converter&lt;TInput, TOutput&gt;</code> 委托，返回一系列转换好的元素
TrueForAll()	bool Predicate<T>(T match)	TrueForAll()方法给每个元素调用谓词委托。如果谓词给每个元素都返回 <code>true</code> ，则 TrueForAll()也返回 <code>true</code> 。如果谓词为一个元素返回了 <code>false</code> ，TrueForAll()就返回 <code>false</code>

下面看看如何使用这些方法。  
Sort()方法把这个委托作为参数：

```
public delegate int Comparison<T>(T x, T y);
```

这样，就可以使用λ表达式传送两个 `Person` 对象，给数组排序。对于 `Person` 对象数组，参数 `T` 是 `Person` 类型：

```
Person[] persons = {
    new Person("Emerson", "Fittipaldi"),
    new Person("Niki", "Lauda"),
    new Person("Ayrton", "Senna"),
    new Person("Michael", "Schumacher")
};
```

```
Array.Sort(persons, (p1, p2) => p1.Firstname.CompareTo(p2.Firstname));
```

`Array.ForEach()`方法将 `Action<T>` 委托作为参数，给数组的每个元素执行操作：

```
public delegate void Action<T>(T obj);
```

于是，就可以传送 `Console.WriteLine` 方法的地址，将每个人写入控制台。`WriteLine()`方法的一个重载版本将 `Object` 类作为参数类型。由于 `Person` 派生于 `Object`，所以它适合于 `Person` 数组：

```
Array.ForEach(persons, Console.WriteLine);
```

`ForEach()`语句的结果将 `persons` 变量引用的集合中的每个人都写入控制台：

```
Emerson Fittipaldi
Niki Lauda
Ayrton Senna
Michael Schumacher
```

如果需要更多的控制，则可以传送一个 $\lambda$ 表达式，其参数应匹配委托定义的参数：

```
Array.ForEach(persons, p => Console.WriteLine("{0}", p.Lastname));
```

下面是写入控制台的姓氏：

```
Fittipaldi
Lauda
Senna
Schumacher
```

`Array.FindAll()`方法需要 `Predicate<T>` 委托：

```
public delegate bool Predicate<T>(T match);
```

`Array.FindAll()`方法为数组中的每个元素调用谓词，并返回一个谓词是 `true` 的数组。在这个例子中，对于 `Lastname` 以字符串“S”开头的所有 `Person` 对象，都返回 `true`。

```
Person[] sPersons = Array.FindAll(persons, p => p.Lastname.StartsWith("S"));
```

迭代返回的集合 `sPersons`，并写入控制台，结果如下：

```
Ayrton Senna
Michael Schumacher
```

`Array.ConvertAll()`方法使用泛型委托 `Converter` 和两个泛型类型。第一个泛型类型 `TInput` 是输入参数，第二个泛型类型 `TOutput` 是返回类型。

```
public delegate TOutput Converter<TInput, TOutput>(TInput input);
```

如果一种类型的数组应转换为另一种类型的数组，就可以使用 `ConvertAll()`方法。下面是一个与 `Person` 类无关的 `Racer` 类。`Person` 类有 `Firstname` 和 `Lastname` 属性，而 `Racer` 类为赛手的

姓名定义了一个属性 Name:

```
public class Racer
{
    public Racer(string name)
    {
        this.name = name;
    }

    public string Name {get; set;}
    public string Team {get; set;}
}
```

使用 `Array.ConvertAll()`, 很容易将 `persons` 数组转换为 `Racer` 数组。给每个 `Person` 元素调用委托。在每个 `Person` 元素的匿名方法的执行代码中, 创建了一个新的 `Racer` 对象, 将 `firstname` 和 `lastname` 连接起来传送给带一个字符串参数的构造函数。结果是一个 `Racer` 对象数组:

```
Racer[] racers =
    Array.ConvertAll<Person, Racer>(
        persons, p =>
            new Racer(String.Format("{0} {1}", p.FirstName, p.LastName));
```

## 9.7 Framework 的其他泛型类型

除了 `System.Collections.Generic` 命名空间之外, .NET Framework 还有其他泛型类型。这里讨论的结构和委托都位于 `System` 命名空间中, 用于不同的目的。

本节讨论如下内容:

- 结构 `Nullable<T>`
- 委托 `EventHandler<TEventArgs>`
- 结构 `ArraySegment<T>`

### 9.7.1 结构 `Nullable<T>`

数据库中的数字和编程语言中的数字有显著不同的特征, 因为数据库中的数字可以为空, C# 中的数字不能为空。 `Int32` 是一个结构, 而结构实现为值类型, 所以它不能为空。

只有在数据库中, 而且把 XML 数据映射为 .NET 类型, 才不存在这个问题。

这种区别常常令人很头痛, 映射数据也要多做许多工作。一种解决方案是把数据库和 XML 文件中的数字映射为引用类型, 因为引用类型可以为空值。但这也会在运行期间带来额外的系统开销。

使用 `Nullable<T>` 结构很容易解决这个问题。在下面的例子中, `Nullable<T>` 用 `Nullable<int>` 实例化。变量 `x` 现在可以像 `int` 那样使用了, 进行赋值或使用运算符执行一些计算。这是因为我们转换了 `Nullable<T>` 类型的运算符。 `x` 还可以是空。可以检查 `Nullable<T>` 的 `HasValue` 和 `Value` 属性, 如果该属性有一个值, 就可以访问该值:

```
Nullable<int> x;
x = 4;
x += 3;
```

```

if (x.HasValue)
{
    int y = x.Value;
}
x = null;

```

因为可空类型使用得非常频繁，所以 C# 有一种特殊的语法，用于定义这种类型的变量。定义这类变量时，不使用一般结构的语法，而使用 `?` 运算符。在下面的例子中，`x1` 和 `x2` 都是可空 `int` 类型的实例：

```

Nullable<int> x1;
int? x2;

```

可空类型可以与 `null` 和数字比较，如上所示。这里，`x` 的值与 `null` 比较，如果 `x` 不是 `null`，就与小于 0 的值比较：

```

int? x = GetNullableType();
if (x == null)
{
    Console.WriteLine("x is null");
}
else if (x < 0)
{
    Console.WriteLine("x is smaller than 0");
}

```

可空类型还可以使用算术运算符。变量 `x3` 是变量 `x1` 和 `x2` 的和。如果这两个可空变量中有一个的值是 `null`，它们的和就是 `null`。

```

int? x1 = GetNullableType();
int? x2 = GetNullableType();
int? x3 = x1 + x2;

```

#### 提示：

这里调用的 `GetNullableType()` 方法只是任意返回可空 `int` 的方法的占位符。为了进行测试，可以把它实现为只返回 `null` 或返回任意整数值。

非可空类型可以转换为可空类型。从非可空类型转换为可空类型时，在不需要强制类型转换的地方可以进行隐式转换。这种转换总是成功的：

```

int y1 = 4;
int? x1 = y1;

```

但从可空类型转换为非可空类型可能会失败。如果可空类型的值是 `null`，把 `null` 值赋予非可空类型，就会抛出 `InvalidOperationException` 类型的异常。这就是进行显式转换时需要类型转换运算符的原因：

```

int? x1 = GetNullableType();
int y1 = (int)x1;

```

如果不进行显式类型转换，还可以使用接合运算符(coalescing operator)从可空类型转换为非可空类型。接合运算符的语法是 `??`，为转换定义了一个默认值，以防可空类型的值是 `null`。这里，如果 `x1` 是 `null`，`y1` 的值就是 0。



```
int? x1 = GetNullableType();
int y1 = x1 ?? 0;
```

### 9.7.2 EventHandler<EventArgs>

在 Windows Forms 和 Web 应用程序中, 为许多不同的事件处理程序定义了委托。其中一些事件处理程序如下:

```
public sealed delegate void EventHandler(object sender, EventArgs e);
public sealed delegate void PaintEventHandler(object sender, PaintEventArgs e);
public sealed delegate void MouseEventHandler(object sender, MouseEventArgs e);
```

这些委托的共同点是, 第一个参数总是 `sender`, 它是事件的起源, 第二个参数是包含事件特定信息的类型。

使用新的 `EventHandler<EventArgs>`, 就不需要为每个事件处理程序定义新委托了。可以看出, 第一个参数的定义方式与以前一样, 但第二个参数是一个泛型类型 `EventArgs`。where 子句指定 `EventArgs` 的类型必须派生于基类 `EventArgs`。

```
public sealed delegate void EventHandler<EventArgs>(object sender, EventArgs e)
    where EventArgs : EventArgs
```

### 9.7.3 ArraySegment<T>

结构 `ArraySegment<T>` 表示数组的一段。如果需要数组的一部分, 就可以使用数组段。在 `ArraySegment<T>` 中, 包含了数组段的信息(偏移量和元素个数)。

在下面的例子中, 变量 `arr` 定义为有 8 个元素的 `int` 数组。`ArraySegment<int>` 类型的变量 `segment` 用于表示该整数数组的一段。该段用构造函数初始化, 在这个构造函数中, 传送了该数组、偏移量和元素个数。其中偏移量设置为 2, 所以从第三个元素开始, 元素个数设置为 3, 所以 6 是数组段的最后一个元素。

数组段可以用 `Array` 属性访问。`ArraySegment<T>` 还有 `Offset` 和 `Count` 属性, 表示定义数组段的初始化了的值。`for` 循环用于迭代数组段。`for` 循环的第一个表达式初始化为迭代开始的偏移量。第二个表达式指定数组段中的元素个数, 以确定迭代是否停止。在 `for` 循环中, 数组段包含的元素用 `Array` 属性来访问:

```
int[] arr = {1, 2, 3, 4, 5, 6, 7, 8};
ArraySegment<int> segment = new ArraySegment<int>(arr, 2, 3);

for (int i = segment.Offset; i < segment.Offset + segment.Count; i++)
{
    Console.WriteLine(segment.Array[i]);
}
```

在上面的例子中, `ArraySegment<T>` 结构有什么用处? `ArraySegment<T>` 可以作为参数传送给方法。这样, 只要一个参数就可以定义数组、偏移量和元素个数, 而不是 3 个参数。

`WorkWithSegment()` 方法把 `ArraySegment<string>` 作为参数。在这个方法的实现代码中, `Offset`、`Count` 和 `Array` 属性的用法与以前相同:

```
void WorkWithSegment(ArraySegment<string> segment)
```

```
{  
    for (int i = segment.Offset; i < segment.Offset + segment.Count; i++)  
    {  
        Console.WriteLine(segment.Array[i]);  
    }  
}
```

注意:

数组段不复制原数组的元素, 但原数组可以通过 `ArraySegment<T>` 访问。如果数组段中的元素改变了, 这些变化也会反映到原数组中。

## 9.8 小结

本章介绍了.NET 2.0 中一个非常重要的特性: 泛型。通过泛型类可以创建独立于类型的类, 泛型方法是独立于类型的方法。接口、结构和委托也可以用泛型的方式创建。泛型引入了一种新的编程方式。我们介绍了算法(尤其是操作和谓词)如何用于不同的类, 而且它们都是类型安全的。泛型委托可以去除集合中的算法。

.NET Framework 的其他类型包括 `Nullable<T>`、`EventHandler<TEventArgs>` 和 `ArraySegment<T>`。

下一章利用泛型来介绍集合类。

# 第 10 章

## 集 合

第 5 章介绍了数组和 `Array` 类执行的接口。数组的大小是固定的。如果元素个数是动态的，就应使用集合类。

`List<T>` 和 `ArrayList` 是与数组相当的集合类。还有其他类型的集合：队列、栈、链表和字典。本章介绍如何使用对象组。主要内容如下：

- 集合接口和类型
- 列表
- 队列
- 栈
- 链表
- 有序表
- 字典
- `Lookup`
- `HashSet`
- 位数组
- 性能

### 10.1 集合接口和类型

集合类可以组合为集合，存储 `Object` 类型的元素和泛型集合类。在 CLR 2.0 之前，不存在泛型。现在泛型集合类通常是集合的首选类型。泛型集合类是类型安全的，如果使用值类型，是不需要装箱操作的。如果要在集合中添加不同类型的对象，且这些对象不是相互派生的，例如在集合中添加 `int` 和 `string` 对象，就只需基于对象的集合类。另一组集合类是专用于特定类型的集合，例如 `StringCollection` 类专用于 `string` 类型。

**提示：**

泛型的内容可参见第 9 章。

对象类型的集合位于 `System.Collections` 命名空间；泛型集合类位于 `System.Collections.Generic` 命名空间；专用于特定类型的集合类位于 `System.Collections.Specialized` 命名空间。

当然，组合集合类还有其他方式。集合可以根据集合类执行的接口组合为列表、集合和字典。接口及其功能如表 10-1 所示。.NET 2.0 为集合类添加了新的泛型接口，例如 `IEnumerable<T>`

和 `ICollection<T>`。这些接口的非泛型版本将一个对象定义为方法的参数，而其泛型版本使用泛型类型 `T`。

提示：  
接口 `IEnumerable`、`ICollection` 和 `ICollection<T>` 的内容详见第 5 章。

对集合和列表非常重要的接口及其方法和属性如表 10-1 所示。

表 10-1

接 口	方法和属性	说 明
<code>IEnumerable</code> , <code>IEnumerable&lt;T&gt;</code>	<code>GetEnumerator()</code>	如果将 <code>foreach</code> 语句用于集合，就需要接口 <code>IEnumerable</code> 。这个接口定义了方法 <code>GetEnumerator()</code> ，它返回一个实现了 <code>IEnumerator</code> 的枚举。泛型接口 <code>IEnumerable&lt;T&gt;</code> 继承了非泛型接口 <code>IEnumerable</code> ，定义了一个返回 <code>IEnumerator&lt;T&gt;</code> 的 <code>GetEnumerator</code> 方法。因为这两个接口具有继承关系，所以对于每个需要 <code>IEnumerator</code> 类型参数的方法，都可以传送 <code>IEnumerator&lt;T&gt;</code> 对象
<code>ICollection</code>	<code>Count</code> , <code>IsSynchronized</code> , <code>SyncRoot</code> , <code>CopyTo()</code>	接口 <code>ICollection</code> 由集合类实现。对于实现了这个接口的集合，可以获得元素个数，把集合复制到数组中 接口 <code>ICollection&lt;T&gt;</code> 扩展了接口 <code>IEnumerable</code> 的功能
<code>ICollection&lt;T&gt;</code>	<code>Count</code> , <code>IsReadOnly</code> , <code>Add()</code> , <code>Clear()</code> , <code>Contains()</code> , <code>CopyTo()</code> <code>Remove()</code>	<code>ICollection&lt;T&gt;</code> 接口是 <code>ICollection</code> 接口的泛型版本。这个接口的泛型版本可以添加和删除元素，获得元素个数
<code>ICollection</code>	<code>IsFixedSize</code> , <code>IsReadOnly</code> , <code>Item</code> , <code>Add</code> , <code>Clear</code> , <code>Contains</code> , <code>IndexOf</code> , <code>Insert</code> , <code>Remove</code> , <code>RemoveAt</code>	接口 <code>ICollection</code> 派生于接口 <code>ICollection</code> 。 <code>ICollection</code> 允许使用索引器访问集合，还可以在集合的任意位置插入或删除元素
<code>ICollection&lt;T&gt;</code>	<code>Item</code> , <code>IndexOf</code> <code>Insert</code> , <code>Remove</code>	与接口 <code>ICollection&lt;T&gt;</code> 类似，接口 <code>ICollection&lt;T&gt;</code> 也继承了接口 <code>ICollection</code> 。 第 5 章提到， <code>Array</code> 类实现了这个接口，但添加或删除元素的方法会抛出 <code>NotSupportedException</code> 异常。在大小固定的只读集合(如 <code>Array</code> 类)中，这个接口定义的一些方法会抛出 <code>NotSupportedException</code> 异常 比较接口 <code>ICollection</code> 的非泛型版本和泛型版本，会发现新的泛型接口只为提供了索引的集合定义了重要的方法和属性。其他方法重构到 <code>ICollection&lt;T&gt;</code> 接口中

(续表)

接 口	方法和属性	说 明
IDictionary	IsFixedSize, IsReadOnly, Item, Keys, Values, Add(), Clear(), Contains(), GetEnumerator(), Remove()	接口 IDictionary 由其元素包含键和值的非泛型集合实现
IDictionary<TKey, TValue>	Item, Keys, Values, Add(), ContainsKey(), Remove(), TryGetValue()	IDictionary<TKey,TValue>由其元素包含键和值的泛型集合实现。这个接口比 IDictionary 简单
ILookup<TKey, TElement>	Count, Item, Contains()	ILookup<TKey, TElement>是.NET 3.5 中的一个新接口，一个键对应多个值的集合使用它。索引器为指定的键返回一个枚举
IComparer<T>	Compare()	接口 IComparer<T>由比较器实现，通过 Compare()方法给集合中的元素排序
IEqualityComparer<T>	Equals(), GetHashCode()	接口 IEqualityComparer<T>由一个比较器实现，该比较器可用于字典中的键。使用这个接口，可以对对象进行相等比较。

提示：

非泛型接口 ICollection 定义的一些属性用于同步不同线程对同一个集合的访问。这些属性不再用于新的泛型接口。其原因是这些属性会导致与同步相关的安全性问题，因为集合通常不仅仅必须同步。集合同步的内容可参见第 19 章。

表 10-2 列出了集合类和由这些类执行的集合接口。

表 10-2

集 合 类	集 合 接 口
ArrayList	ICollection, IEnumerable
Queue	ICollection, IEnumerable
Stack	ICollection, IEnumerable
BitArray	ICollection, IEnumerable
Hashtable	IDictionary, ICollection, IEnumerable
SortedList	IDictionary, ICollection, IEnumerable
List<T>	ICollection<T>, IEnumerable<T>, ICollection, IEnumerable
Queue<T>	ICollection<T>, IEnumerable, IEnumerable



(续表)

集 合 类	集 合 接 口
Stack<T>	IEnumerable<T>, ICollection, IEnumerable
LinkedList<T>	ICollection<T>, IEnumerable<T>, ICollection, IEnumerable
HashSet<T>	ICollection<T>, IEnumerable<T>, IEnumerable
Dictionary<TKey, TValue>	IDictionary<TKey, TValue>, ICollection<KeyValuePair<TKey, TValue>>, IEnumerable<KeyValuePair<TKey, TValue>>, IDictionary, ICollection, IEnumerable
SortedDictionary<TKey, TValue>	IDictionary<TKey, TValue>, ICollection<KeyValuePair<TKey, TValue>>, IEnumerable<KeyValuePair<TKey, TValue>>, IDictionary, ICollection, IEnumerable
SortedList<TKey, TValue>	IDictionary<TKey, TValue>, ICollection<KeyValuePair<TKey, TValue>>, IEnumerable<KeyValuePair<TKey, TValue>>, IDictionary, ICollection, IEnumerable
Lookup<TKey, TElement>	ILookup<TKey, TElement>, IEnumerable<IGrouping<TKey, TElement>>, IEnumerable

10.2 列表

.NET Framework 为动态列表提供了类 ArrayList 和 List<T>。System.Collections.Generic 命名空间中的类 List<T>的用法非常类似于 System.Collections 命名空间中的 ArrayList 类。这个类实现了 IList、ICollection 和 IEnumerable 接口。第 9 章讨论了这些接口的方法，所以本节只探讨如何使用 List<T>类。

下面的例子将 Racer 类中的成员用作要添加到集合中的成员，以表示一级方程式的一位赛手。这个类有 4 个字段：firstname、lastname、country 和获胜次数。这些字段可以用属性来访问。在该类的构造函数中，可以传送赛手的姓名和获胜次数，以设置成员。方法 ToString()重写为返回赛手的姓名。类 Racer 也实现了泛型接口 IComparer<T>，为 Racer 元素排序。

```
[Serializable]
public class Racer : IComparable<Racer>, IFormattable
{
    public Racer()
        : this(String.Empty, String.Empty, String.Empty) {}

    public Racer(string firstname, string lastname, string country)
        : this(firstname, lastname, country, 0) {}

    public Racer(string firstname, string lastname, string country, int wins)
    {
        this.firstname = firstname;
        this.lastname = lastname;
        this.country = country;
        this.wins = wins;
    }

    public string Firstname { get; set; }
```

```

public string Lastname{ get; set; }
public string Country { get; set; }
public int Wins { get; set; }

public override string ToString()
{
    return String.Format("{0} {1}",
        FirstName, LastName);
}

public string ToString(string format, IFormatProvider formatProvider)
{
    switch (format.ToUpper())
    {
        case null:
        case "N": //Name
            return ToString();
        case "F": //FirstName
            return FirstName;
        case "L": //LastName
            return LastName;
        case "W": //Wins
            return String.Format("{0}, Wins: {1}",
                ToString(), Wins);
        case "C": // Country
            return String.Format(
                "{0}, Country: {1}",
                ToString(), Country);
        case "A": // All
            return String.Format(
                "{0}, {1} Wins: {2}",
                ToString(), Country, Wins);
        default:
            throw new FormatException(String.Format(
                formatProvider,
                "Format {0} is not supported",
                format));
    }
}

public string ToString(string format)
{
    return ToString(format, null);
}

public int CompareTo(Racer other)
{
    int compare = this.LastName.CompareTo(
        other.LastName);
    if (compare == 0)
        return this.FirstName.CompareTo(
            other.FirstName);
    return compare;
}
}

```

### 10.2.1 创建列表

调用默认的构造函数，就可以创建列表对象。在泛型类 `List<T>` 中，必须在声明中为列表

的值指定类型。下面的代码说明了如何声明一个包含 `int` 的 `List<T>` 和一个包含 `Racer` 元素的列表。`ArrayList` 是一个非泛型列表，可以将任意 `Object` 类型作为其元素。

使用默认的构造函数创建一个空列表。元素添加到列表中后，列表的容量就会扩大为可接纳 4 个元素。如果添加了第 5 个元素，列表的大小就重新设置为包含 8 个元素。如果 8 个元素还不够，列表的大小就重新设置为 16。每次都会将列表的容量重新设置为原来的 2 倍。

```
ArrayList objectList = new ArrayList();

List<int> intList = new List<int>();
List<Racer> racers = new List<Racer>();
```

如果列表的容量改变了，整个集合就要重新分配到一个新的内存块中。在 `List<T>` 的实现代码中，使用了一个 `T` 类型的数组。通过重新分配内存，创建一个新数组，`Array.Copy()` 将旧数组中的元素复制到新数组中。为节省时间，如果事先知道列表中元素的个数，就可以用构造函数定义其容量。下面创建了一个容量为 10 个元素的集合。如果该容量不足以容纳要添加的元素，就把集合的大小重新设置为 20，或 40，每次都是原来的 2 倍。

```
ArrayList objectList = new ArrayList(10);
List<int> intList = new List<int>(10);
```

使用 `Capacity` 属性可以获取和设置集合的容量。

```
objectList.Capacity = 20;
intList.Capacity = 20;
```

容量与集合中元素的个数不同。集合中元素的个数可以用 `Count` 属性读取。当然，容量总是大于或等于元素个数。只要不把元素添加到列表中，元素个数就是 0。

```
Console.WriteLine(intList.Count);
```

如果已经将元素添加到列表中，且不希望添加更多的元素，就可以调用 `TrimExcess()` 方法，去除不需要的容量。但是，重新定位是需要时间的，所以如果元素个数超过了容量的 90%，`TrimExcess()` 方法将什么也不做。

```
intList.TrimExcess();
```

**提示：**

对于新的应用程序，通常可以使用泛型类 `List<T>` 替代非泛型类 `ArrayList`，而且 `ArrayList` 类的方法与 `List<T>` 非常相似，所以本节将只介绍 `List<T>`。

### 1. 集合初始化器

C# 3.0 允许使用集合初始化器给集合赋值。集合初始化器的语法类似于第 5 章介绍的数组初始化器。使用集合初始化器，可以在初始化集合时，在花括号中给集合赋值：

```
List<int> intList = new List<int>() {1, 2};
List<string> stringList =
    new List<string>() {"one", "two"};
```

**提示：**

集合初始化器是 C# 3.0 编程语言的一个特性，没有反映在已编译的程序集的 IL 代码中。

编译器会把集合初始化器变成给初始化列表中的每一项调用 Add() 方法。

## 2. 添加元素

使用 Add() 方法可以给列表添加元素，如下所示。实例化的泛型类型定义了 Add() 方法的参数类型：

```
List<int> intList = new List<int>();
intList.Add(1);
intList.Add(2);

List<string> stringList = new List<string>();
stringList.Add("one");
stringList.Add("two");
```

变量 racers 定义为类型 List<Racer>。使用 new 操作符创建相同类型的一个新对象。因为类 List<T> 用具体类 Racer 来实例化，所以现在只有 Racer 对象可以用 Add() 方法添加。在下面的例子中，创建了 5 个一级方程式赛车手，并添加到集合中。前 3 个用集合初始化器添加，后两个通过显式调用 Add() 方法来添加。

```
Racer graham = new Racer("Graham", "Hill", "UK", 14);
Racer emerson = new Racer("Emerson", "Fittipaldi", "Brazil", 14);
Racer mario = new Racer("Mario", "Andretti", "USA", 12);

List<Racer> racers = new List<Racer>(20);
    {graham, emerson, mario};

racers.Add(new Racer("Michael", "Schumacher",
    "Germany", 91));
racers.Add(new Racer("Mika", "Hakkinen",
    "Finland", 20));
```

使用 List<T> 类的 AddRange() 方法，可以一次给集合添加多个元素。AddRange() 方法的参数是 IEnumerable<T> 类型的对象，所以也可以传送一个数组，如下所示：

```
racers.AddRange(new Racer[] {
    new Racer("Niki", "Lauda", "Austria", 25)
    new Racer("Alian", "Prost", "France", 51 } );
```

### 提示：

集合初始化器只能在声明集合时使用。AddRange() 方法则可以在初始化集合后调用。

如果在实例化列表时知道集合的元素个数，也可以将执行 IEnumerable<T> 的任意对象传送给类的构造函数。这非常类似于 AddRange() 方法：

```
List<Racer> racers = new List<Racer> (new Racer[] {
    new Racer("Jochen", "Rindt", "Austria", 6)
    new Racer("Ayrton", "Senna", "Brazil", 41 } );
```

## 3. 插入元素

使用 Insert() 方法可以在指定位置插入元素：

```
racers.Insert(3, new Racer("Phil", "Hill", "USA", 3));
```

方法 InsertRange() 提供了插入大量元素的容量，类似于前面的 AddRange() 方法。



如果索引集大于集合中的元素个数，就抛出 `ArgumentOutOfRangeException` 类型的异常。

#### 4. 访问元素

执行了 `IList` 和 `IList<T>` 接口的所有类都提供了一个索引器，所以可以使用索引器，通过传送元素号来访问元素。第一个元素可以用索引值 0 来访问。指定 `racers[3]`，可以访问列表中的第 4 个元素：

```
Racer rl = racers[3];
```

可以用 `Count` 属性确定元素个数，再使用 `for` 循环迭代集合中的每个元素，使用索引器访问每一项：

```
for (int i=0; i<racers.Count; i++)
{
    Console.WriteLine(racers[i]);
}
```

**提示：**

可以通过索引访问的集合类有 `ArrayList`、`StringCollection` 和 `List<T>`。

`List<T>` 执行了接口 `IEnumerable`，所以也可以使用 `foreach` 语句迭代集合中的元素。

```
foreach (Racer r in racers)
{
    Console.WriteLine(r);
}
```

**注意：**

编译器解析 `foreach` 语句时，利用了接口 `IEnumerable` 和 `IEnumerator`，参见第 5 章。

除了使用 `foreach` 语句之外，`List<T>` 类还提供了 `ForEach()` 方法，它用 `Action<T>` 参数声明。

```
public void ForEach(Action<T> action);
```

`ForEach()` 的执行代码如下。`ForEach()` 迭代集合中的每一项，调用传送作为每一项的参数的方法。

```
public class List < T > : IList < T >
{
    private T[] items;

    //...

    public void ForEach(Action < T > action)
    {
        if (action == null) throw new ArgumentNullException("action");

        foreach (T item in items)
        {
            action(item);
        }
    }
    //...
}
```



为了给 `ForEach` 传送一个方法, `Action<T>` 声明为一个委托, 它定义了一个返回类型为 `void`、参数为 `T` 的方法。

```
public delegate void Action<T>(T obj);
```

在 `Racer` 项的列表中, `ForEach()` 方法的处理程序必须声明为以 `Racer` 对象作为参数, 返回类型是 `void`。

```
public void ActionHandler(Racer obj);
```

`Console.WriteLine()` 方法的一个重载版本将 `Object` 作为参数, 所以可以将这个方法的地址传送给 `ForEach()` 方法, 把集合中的每个赛手写入控制台:

```
racers.ForEach(Console.WriteLine);
```

也可以编写一个匿名方法, 它将 `Racer` 对象作为参数。这里, 格式 `A` 由 `IFormattable` 接口的 `ToString()` 方法用于显示赛手的所有信息:

```
racers.ForEach(
    delegate(Racer r)
    {
        Console.WriteLine("{0:A}", r);
    });
```

在 C# 3.0 中, 还可以使用  $\lambda$  表达式和以委托作为参数的方法。使用匿名方法执行的迭代也可以用  $\lambda$  表达式定义:

```
racers.ForEach(
    r => Console.WriteLine("{0:A}", r));
```

注意:

匿名方法和  $\lambda$  表达式详见第 7 章。

## 5. 删除元素

删除元素时, 可以利用索引, 或传送要删除的元素。下面的代码把 3 传送给 `RemoveAt()`, 删除第 4 个元素:

```
racers.RemoveAt(3);
```

也可以直接将 `Racer` 对象传送给 `Remove()` 方法, 删除这个元素。按索引删除比较快, 因为必须在集合中搜索要删除的元素。`Remove()` 方法先在集合中搜索, 用 `IndexOf()` 方法确定元素的索引, 再使用该索引删除元素。`IndexOf()` 方法先检查元素类型是否执行了 `IEquatable` 接口。如果是, 就调用这个接口中的 `Equals()` 方法, 确定集合中的元素是否等于传送给 `Equals()` 方法的元素。如果没有执行这个接口, 就使用 `Object` 类的 `Equals()` 方法比较元素。`Object` 类中 `Equals()` 方法的默认实现代码对值类型进行按位比较, 对引用类型只比较其引用。

注意:

第 6 章介绍了如何重写 `Equals()` 方法。

这里从集合中删除了变量 `graham` 引用的赛手。变量 `graham` 是前面在填充集合时创建的。接口 `IEquatable` 和 `Object.Equals()` 方法都没有在 `Racer` 类中重写, 所以不能用要删除的元素内容

创建一个新对象，再把它传送给 Remove()方法。

```
If(!racers.Remove(gham))
{
    Console.WriteLine("object not found in collection");
}
```

方法 RemoveRange()可以从集合中删除许多元素。它的第一个参数指定了开始删除的元素索引，第二个参数指定了要删除的元素个数。

```
int index = 3;
int count = 5;
racers.RemoveRange(index, count)
```

要从集合中删除有指定特性的所有元素，可以使用 RemoveAll()方法。这个方法在搜索元素时使用下面将讨论的 Predicate<T>参数。要删除集合中的所有元素，可以使用 ICollection<T>接口定义的 Clear()方法。

## 6. 搜索

有不同的方式在集合中搜索元素。可以获得要查找的元素的索引，或者搜索元素本身。可以使用的方法有 IndexOf()、LastIndexOf()、FindIndex()、FindLastIndex()、Find()和 FindLast()。如果只检查元素是否存在，List<T>类提供了 Exists()方法。

方法 IndexOf()需要将一个对象作为参数，如果在集合中找到该元素，这个方法就返回该元素的索引。如果没有找到该元素，就返回-1。IndexOf()方法使用 IEquatable 接口来比较元素。

```
int index1 = racers.IndexOf(mario);
```

使用方法 IndexOf()，还可以指定不需要搜索整个集合，但必须指定从哪个索引开始搜索以及要搜索的元素个数。

除了使用 IndexOf()方法搜索指定的元素之外，还可以搜索有某个特性的元素，该特性可以用 FindIndex()方法来定义。FindIndex()方法需要一个 Predicate 类型的参数：

```
public int FindIndex(Predicate<T> match);
```

Predicate<T>类型是一个委托，它返回一个布尔值，需要把类型 T 作为参数。这个委托的用法与 ForEach()方法中的 Action 委托类似。如果 Predicate 返回 true，就表示有一个匹配，找到了一个元素。如果它返回 false，就表示没有找到元素，搜索将继续。

```
public delegate bool Predicate<T>(T obj);
```

在 List<T>类中，把 Racer 对象作为类型 T，所以可以将一个方法(该方法将类型 Racer 定义为参数、且返回 bool)的地址传送给 FindIndex()方法。查找指定国家的第一个赛车手时，可以创建如下所示的 FindCountry 类。Find()方法的签名和返回类型是通过 Predicate<T>委托定义的。Find()方法使用变量 country 搜索用 FindCountry 类的构造函数定义的一个国家。

```
public class FindCountry
{
    public FindCountry(string country)
    {
        this.country = country;
    }
}
```

```

private string country;

public bool FindCountryPredicate(Racer racer)
{
    if(racer == null) throw new ArgumentNullException("racer");
    return r.Country == country;
}
}

```

使用 FindIndex()方法可以创建 FindCountry()类的一个新实例，把一个国家字符串传送给构造函数，再传送 Find 方法的地址。FindIndex()方法成功完成后，index2 就包含集合中 Country 属性设置为 Finland 的第一项的索引。

```
int index2 = racers.FindIndex(new FindCountry("Finland").FindCountryPredicate);
```

除了用处理程序方法创建一个类之外，还可以在这里创建一个λ表达式。结果与前面完全相同。现在λ表达式定义了实现代码，来搜索 Country 属性设置为 Finland 的元素。

```
int index3 = racers.FindIndex(r=> r.Country == "Finland");
```

与 IndexOf()方法类似，使用 FindIndex()方法也可以指定搜索开始的索引和要迭代的元素个数。为了从集合中最后一个元素开始向前搜索，可以使用 FindLastIndex()方法。

方法 FindIndex()返回所搜索元素的索引。除了获得索引之外，还可以直接获得集合中的元素。方法 Find()需要一个 Predicate<T>类型的参数，这与 FindIndex()方法类似。下面的方法 Find()搜索列表中 Firstname 属性设置为 Niki 的第一个赛车手。当然，也可以执行 FindLast()方法，查找与 Predicate 匹配的最后一项。

```
Racer r = racers.Find(r=> r.Firstname == "Niki");
```

要获得与 Predicate 匹配的所有项，而不是一项，可以使用 FindAll()方法。FindAll()方法使用的 Predicate<T>委托与 Find()和 FindIndex()方法相同。FindAll()方法在找到第一项后，不会停止搜索，而是迭代集合中的每一项，返回 Predicate 是 true 的所有项。

这里调用了 FindAll()方法，返回 Wins 属性设置超过 20 的所有 racer 项。所有赢得 20 多场比赛的赛车手都从 bigWinners 列表中引用。

```
List<Racer> bigWinners = racers.FindAll(r=> r.Wins > 20);
```

用 foreach 语句迭代 bigWinners 变量，结果如下：

```

foreach(Racer r in bigWinners)
{
    Console.WriteLine("{0:A}", r);
}

```

```

Michael Schumacher, Germany Wins: 91
Niki Lauda, Austria Wins: 25
Alain Prost, France Wins: 51

```

这个结果没有排序，但这是下一步要做的工作。

## 7. 排序

List<T>类可以使用 Sort()方法对元素排序。Sort()方法使用快速排序算法，比较所有的元素，直到对整个列表排好序为止。

Sort()方法定义了几个重载方法。可以传送给它的参数有泛型委托 Comparison<T>、泛型接口

`IComparer<T>`、泛型接口 `IComparable<T>` 和一个范围值。

```
public void List<T>. Sort();
public void List<T>. Sort(Comparison<T>);
public void List<T>. Sort(IComparer<T>);
public void List<T>. Sort(Int32, Int32, IComparer<T>);
```

只有集合中的元素执行了接口 `IComparable`，才能使用不带参数的 `Sort()` 方法。

类 `Racer` 实现了 `IComparable<T>` 接口，可以按姓氏对赛手排序：

```
racers. Sort();
racers. ForEach(Console.WriteLine);
```

如果需要按照元素类型不默认支持的方式排序，就应使用其他技术，例如传送执行了 `IComparer<T>` 接口的对象。

类 `RacerComparer` 给 `Racer` 类型执行了接口 `IComparer<T>`。这个类允许按名字、姓氏、国籍或获胜次数排序。排序的种类用内部枚举类型 `CompareType` 定义。`CompareType` 用类 `RacerComparer` 的构造函数设置。接口 `IComparer<Racer>` 定义了排序所需的方法 `Compare()`。在这个方法的实现代码中，使用了 `string` 和 `int` 类型的 `CompareTo()` 方法。

```
public class RacerComparer : IComparer<Racer>
{
    public enum CompareType
    {
        Firstname,
        Lastname;
        Country;
        Wins
    }

    private CompareType compareType;
    public RacerComparer(CompareType compareType)
    {
        this. compareType = compareType;
    }

    public int Compare(Racer x, Racer y)
    {
        if (x == null) throw new ArgumentNullException("x");
        if (y == null) throw new ArgumentNullException("y");

        int result;
        switch (compareType)
        {
            case compareType.Firstname:
                return x.Firstname.CompareTo(y.Firstname);
            case compareType.Lastname:
                return x.Lastname.CompareTo(y.Lastname);
            case compareType.Country:
                if ((result = x.Country.CompareTo(y. Country) == 0)
                    return x.Lastname.CompareTo(y.Lastname);
                else
                    return res;
            case compareType.Wins:
                return x.Wins.CompareTo(y.Wins);
            default:
                throw new ArgumentNullException("Invalid Compare Type");
        }
    }
}
```



```
    }
}
```

现在，可以对类 `RacerComparer` 的一个实例使用 `Sort()` 方法。传送枚举 `RacerComparer.CompareType.Country`，按属性 `Country` 对集合排序：

```
racers.Sort(new RacerComparer(RacerComparer.CompareType.Country));
racers.ForEach(Console.WriteLine);
```

排序的另一种方式是使用重载的 `Sort()` 方法，它需要一个 `Comparison<T>` 委托：

```
public void List<T> Sort (Comparison<T>);
```

`Comparison<T>` 是一个方法的委托，该方法有两个 `T` 类型的参数，返回类型为 `int`。如果参数值相等，该方法就返回 0。如果第一个参数比第二个小，就返回小于 0 的值；否则，返回大于 0 的值。

```
public delegate int Comparison<T>(T x, T y);
```

现在可以把一个  $\lambda$  表达式传送给 `Sort()` 方法，按获胜次数排序。两个参数的类型是 `Racer`，在其实现代码中，使用 `int` 方法 `CompareTo()` 比较 `Wins` 属性。在实现代码中，`r2` 和 `r1` 以逆序方式使用，所以获胜次数以降序方式排序。方法调用完后，完整的赛手列表就按赛手的获胜次数排序。

```
racers.Sort(r1, r2) => r2.Wins.CompareTo(r1.Wins));
```

也可以调用 `Reverse()` 方法，倒转整个集合的顺序。

## 8. 类型转换

使用 `List<T>` 类的 `ConvertAll()` 方法，可以把任意类型的集合转换为另一种类型。`ConvertAll()` 方法使用一个 `Converter` 委托，其定义如下：

```
public sealed delegate TOutput Converter<TInput, TOutput>(TInput from);
```

泛型类型 `TInput` 和 `TOutput` 用于转换。`TInput` 是委托方法的变元，`TOutput` 是返回类型。

在这个例子中，所有的 `Racer` 类型都应转换为 `Person` 类型。`Racer` 类型包含姓、名、国籍和获胜次数，而 `Person` 类型只包含姓名。为了进行转换，可以忽略赛手的国籍和获胜次数，但姓名必须转换：

```
[Serializable]
public class Person
{
    private string name;

    public Person(string name)
    {
        this.name = name;
    }

    public override string ToString()
    {
        return name;
    }
}
```

转换时调用了 `racers.ConvertAll<Person>()` 方法。这个方法的变元定义为一个  $\lambda$  表达式，其变元的类型是 `Racer`，返回类型是 `Person`。在  $\lambda$  表达式的实现代码中，创建并返回了一个新的



Person 对象。对于 Person 对象，把 Firstname 和 Lastname 传送给构造函数：

```
List<Person> persons = racers.ConvertAll<Person>(
    r => new Person(r.Firstname + " " + r.Lastname));
```

转换的结果是一个列表，其中包含转换过来的 Person 对象：类型为 List<Person>的 persons 列表。

### 10.2.2 只读集合

集合创建好后，就是可读写的。当然，集合必须是可读写的，否则就不能给它填充值了。但是，在填充完集合后，可以创建只读集合。List<T>集合的方法 AsReadOnly 返回 ReadOnlyCollection<T>类型的对象。ReadOnlyCollection<T>类执行的接口与 List<T>相同，但所有修改集合的方法和属性都抛出 NotSupportedException 异常。

## 10.3 队列

队列是其元素以先进先出(FIFO)的方式来处理的集合。先放在队列中的元素会先读取。队列的例子有在机场排的队、人力资源部中等待处理求职信的队列、打印队列中等待处理的打印任务、以循环方式等待 CPU 处理的线程。另外，还常常有元素根据其优先级来处理的队列。例如，在机场的队列中，商务舱乘客的处理要优先于经济舱的乘客。这里可以使用多个队列，一个队列对应一个优先级。在机场，这是很常见的，因为商务舱乘客和经济舱乘客有不同的登记队列。打印队列和线程也是这样。可以为一组队列建立一个数组，数组中的一项代表一个优先级。在每个数组项中，都有一个队列，其处理按照 FIFO 的方式进行。

**注意：**

本章的后面将使用链表的另一种实现方式，来定义优先级列表。

在.NET 的 System.Collections 命名空间中有非泛型类 Queue，在 System.Collections.Generic 命名空间中有泛型类 Queue<T>。这两个类的功能非常类似，但泛型类是强类型化的，定义了类型 T，而非泛型类基于 Object 类型。

在内部，Queue<T>类使用 T 类型的数组，这类似于 List<T>类型。另一个类似之处是它们都执行 ICollection 和 IEnumerable 接口。Queue 类执行了 ICollection、IEnumerable 和 ICloneable 接口。Queue<T>类执行了 IEnumerable 和 ICloneable 接口。Queue<T>泛型类没有执行泛型接口 ICollection<T>，因为这个接口用 Add()和 Remove()方法定义了集合中添加和删除元素的方法。

队列与列表的主要区别是队列没有执行 IList 接口。所以不能用索引器访问队列。队列只允许添加元素，该元素会放在队列的尾部(使用 Enqueue()方法)，从队列的头部获取元素(使用 Dequeue()方法)。

图 10-1 显示了队列的元素。Enqueue()方法在队列的一端添加元素，Dequeue()方法在队列的另一端读取和删除元素。用 Dequeue()方法读取元素，将同时从队列中删除该元素。再调用一次 Dequeue()方法，会删除队列中的下一项。



图 10-1

Queue 和 Queue <T>类的方法如表 10-3 所示。

表 10-3

Queue 和 Queue <T> 类的成员	说 明
Enqueue()	在队列一端添加一个元素
Dequeue()	在队列的头部读取和删除一个元素。如果在调用 Dequeue()方法时，队列中不再有元素，就抛出 InvalidOperationException 异常
Peek()	在队列的头部读取一个元素，但不删除它
Count	返回队列中的元素个数
TrimExcess()	重新设置队列的容量。Dequeue()方法从队列中删除元素，但不会重新设置队列的容量。要从队列的头部去除空元素，应使用 TrimExcess()方法
Contains()	确定某个元素是否在队列中，如果是，就返回 true
CopyTo()	把元素从队列复制到一个已有的数组中
ToArray()	ToArray()方法返回一个包含队列元素的新数组

在创建队列时，可以使用与 List<T>类型类似的构造函数。默认的构造函数会创建一个空队列，也可以使用构造函数指定容量。在把元素添加到队列中时，容量会递增，包含 4、8、16 和 32 个元素。与 List<T>类型类似，队列的容量也总是根据需要成倍增加。非泛型类 Queue 的默认构造函数与此不同，它会创建一个包含 32 项的空数组。使用构造函数的重载版本，还可以将执行了 IEnumerable<T>接口的其他集合复制到队列中。

下面的文档管理应用程序示例演示了 Queue<T>类的用法。使用一个线程将文档添加到队列中，用另一个线程从队列中读取文档，并处理它们。

存储在队列中的项是 Document 类型。Document 类定义了标题和内容：

```
public class Document
{
    private string title;
    public string Title
    {
        get
        {
            return title;
        }
    }

    private string content;
    public string Content
```

```

    {
        get
        {
            return content;
        }
    }

    public Document(string title, string content)
    {
        this.title = title;
        this.content = content;
    }
}

```

DocumentManager 类是 Queue<T>类外面的一层。DocumentManager 类定义了如何处理文档：用 AddDocument()方法将文档添加到队列中，用 GetDocument()方法从队列中获得文档。

在 AddDocument()方法中，用 Enqueue()方法把文档添加到队列的尾部。在 GetDocument()方法中，用 Dequeue()方法从队列中读取第一个文档。多个线程可以同时访问 DocumentManager，所以用 lock 语句锁定对队列的访问。

**提示：**

线程和 lock 语句参见第 19 章。

IsDocumentAvailable 是一个只读布尔属性，如果队列中还有文档，它就返回 true，否则返回 false。

```

public class DocumentManager
{
    private readonly Queue<Document> documentQueue = new Queue<Document>;

    public void AddDocument(Document doc)
    {
        lock(this)
        {
            documentQueue.Enqueue(doc);
        }
    }

    public Document GetDocument()
    {
        Document doc = null;
        lock(this)
        {
            doc = documentQueue.Dequeue();
        }
        return doc;
    }

    public bool IsDocumentAvailable
    {
        get
        {
            return documentQueue.Count > 0;
        }
    }
}

```

类 `ProcessDocuments` 在一个单独的线程中处理队列中的文档。能从外部访问的唯一方法是 `Start()`。在 `Start()` 方法中，实例化了一个新线程。创建一个 `ProcessDocuments` 对象，来启动线程，定义 `Run()` 方法作为线程的启动方法。`ThreadStart` 是一个委托，它引用了由线程启动的方法。在创建 `Thread` 对象后，就调用 `Thread.Start()` 方法来启动线程。

使用 `ProcessDocuments` 类的 `Run()` 方法定义一个无限循环。在这个循环中，使用属性 `IsDocumentAvailable` 确定队列中是否还有文档。如果还有，就从 `DocumentManager` 中提取文档并处理。这里的处理仅是把信息写入控制台。在真正的应用程序中，文档可以写入文件、数据库，或通过网络发送。

```
public class ProcessDocuments
{
    public static void Start(DocumentManager dm)
    {
        new Thread(new ProcessDocuments(dm).Run).Start();
    }

    protected ProcessDocuments(DocumentManager dm)
    {
        documentManager = dm;
    }

    private DocumentManager documentManager;

    protected void Run()
    {
        while (true)
        {
            if (documentManager.IsDocumentAvailable)
            {
                Document doc = documentManager.GetDocument();
                Console.WriteLine("Processing document {0}", doc.Title);
            }
            Thread.Sleep(new Random().Next(20));
        }
    }
}
```

在应用程序的 `Main()` 方法中，实例化一个 `DocumentManager` 对象，启动文档处理线程。接着创建 1000 个文档，并添加到 `DocumentManager` 中：

```
class Program
{
    static void Main()
    {
        DocumentManager dm = new DocumentManager();

        ProcessDocuments.Start(dm);

        // Create documents and add them to the DocumentManager
        for (int i = 0; i < 1000; i++)
        {
            Document doc = new Document("Doc " + i.ToString(), "content");
            dm.AddDocument(doc);
            Console.WriteLine("added document {0}", doc.Title);
            Thread.Sleep(new Random().Next(20));
        }
    }
}
```

在启动应用程序时，会在队列中添加和删除文档，输出如下所示：

```
Added document Doc 279
Processing document Doc 236
Added document Doc 280
Processing document Doc 237
Added document Doc 281
Processing document Doc 238
Processing document Doc 239
Processing document Doc 240
Processing document Doc 241
Added document Doc 282
Processing document Doc 242
Added document Doc 283
Processing document Doc 243
```

完成示例应用程序中描述的任务的真实程序可以处理用 Web 服务接收到的文档。

10.4 栈

栈是与队列非常类似的另一个容器，只是要使用不同的方法访问栈。最后添加到栈中的元素会最先读取。栈是一个后进先出(LIFO)容器。

图 10-2 表示一个栈，用 Push()方法在栈中添加元素，用 Pop()方法获取最近添加的元素。

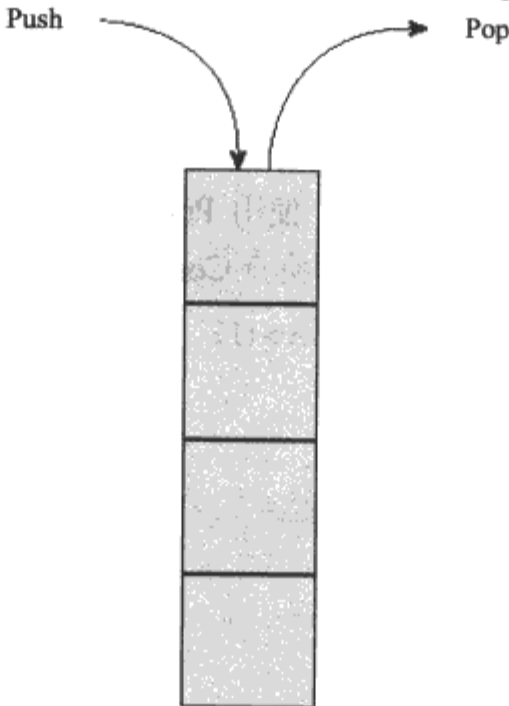


图 10-2

与 Queue 类相同，非泛型类 Statck 也执行了 ICollection、IEnumerable 和 ICloneable 接口；泛型类 Statck<T>实现了 IEnumerable<T>、ICollection 和 IEnumerable 接口。

Statck 和 Statck<T>类的成员如表 10-4 所示。

表 10-4

Statck 和 Statck<T>类的成员	说 明
Push()	在栈顶添加一个元素
Pop()	从栈顶删除一个元素，并返回该元素。如果栈是空的，就抛出 InvalidOperationException 异常
Peek()	返回栈顶元素，但不删除它



(续表)

Statck 和 Statck<T>类的成员	说 明
Count	返回栈中的元素个数
Contains()	确定某个元素是否在栈中，如果是，就返回 true
CopyTo()	把元素从栈复制到一个已有的数组中。
ToArray()	ToArray()方法返回一个包含栈中元素的新数组

在下面的例子中，使用 `Push()`方法把三个元素添加到栈中。在 `foreach` 方法中，使用 `IEnumerable` 接口迭代所有的元素。栈的枚举器不会删除元素，只会逐个返回元素。

```
Stack<char> alphabet = new Stack<char>();
alphabet.Push('A');
alphabet.Push('B');
alphabet.Push('C');

foreach (char item in alphabet)
{
    Console.Write(item);
}
Console.WriteLine();
```

因为元素的读取顺序是从最后一个添加到栈中的元素开始到第一个元素，所以得到的结果如下：

```
CBA
```

用枚举器读取元素不会改变元素的状态。使用 `Pop()`方法会从栈中读取每个元素，然后删除它们。这样，就可以使用 `while` 循环迭代集合，检查 `Count` 属性，确定栈中是否还有元素：

```
Stack<char> alphabet = new Stack<char>();
alphabet.Push('A');
alphabet.Push('B');
alphabet.Push('C');

Console.Write("First iteration: ");
foreach (char item in alphabet)
{
    Console.Write(item);
}
Console.WriteLine();

Console.Write("Second iteration: ");
while (alphabet.Count > 0)
{
    Console.Write(alphabet.Pop());
}
Console.WriteLine();
```

结果是两个 CBA。在第二次迭代后，栈变空，因为第二次迭代使用了 `Pop()`方法：

```
First iteration: CBA
Second iteration: CBA
```

10.5 链表

`LinkedList<T>`集合类没有非泛型集合的类似版本。`LinkedList<T>`是一个双向链表，其元素

指向它前面和后面的元素，如图 10-3 所示。

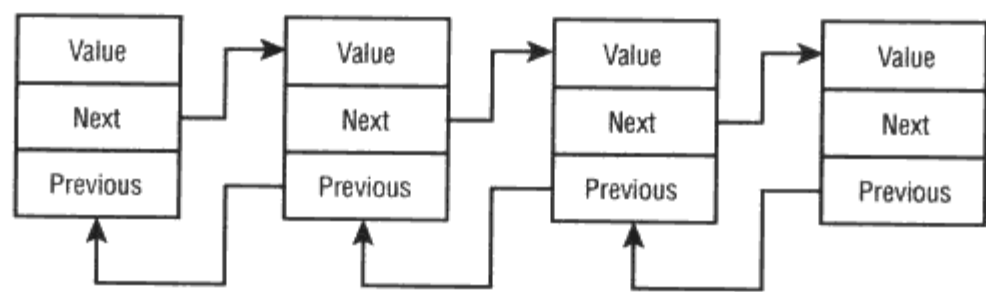


图 10-3

链表的优点是，如果将元素插入列表的中间位置，使用链表会非常快。在插入一个元素时，只需修改上一个元素的 Next 引用和下一个元素的 Previous 引用，使它们引用所插入的元素。在 List<T>和 ArrayList 类中，插入一个元素，需要移动该元素后面的所有元素。

当然，链表也有缺点。链表的元素只能一个接一个地访问，这需要较长的时间来查找位于链表中间或尾部的元素。

链表不仅能在列表中存储元素，还可以给每个元素存储下一个元素和上一个元素的信息。这就是 LinkedList<T>包含 LinkedListNode<T>类型的元素的原因。使用 LinkedListNode<T>类，可以获得列表中的下一个元素和上一个元素。表 10-5 描述了 LinkedListNode<T>的属性。

表 10-5

LinkedListNode<T>的属性	说 明
List	返回与节点相关的 LinkedList<T>
Next	返回当前节点之后的节点。其返回类型是 LinkedListNode<T>
Previous	返回当前节点之前的节点
Value	返回与节点相关的元素，其类型是 T

类 LinkedList<T>执行了 IEnumerable<T>、ICollection<T>、ICollection、IEnumerable、ISerializable 和 IDeserializationCallback 接口。这个类的成员如表 10-6 所示。

表 10-6

LinkedList<T>的成员	说 明
Count	返回链表中的元素个数
First	返回链表中的第一个节点。其返回类型是 LinkedListNode<T>。使用这个返回的节点，可以迭代集合中的其他节点
Last	返回链表中的最后一个元素。其返回类型是 LinkedListNode<T>。使用这个节点，可以逆序迭代集合中的其他节点
AddAfter() AddBefore() AddFirst() AddLast()	使用 AddXXX 方法可以在链表中添加元素。使用相应的 Add 方法，可以在链表的指定位置添加元素。AddAfter()需要一个 LinkedListNode<T>对象，在该对象中可以指定要添加的新元素后面的节点。AddBefore()把新元素放在第一个参数定义的节点前面。AddFirst()和 AddLast()把新元素添加到链表的开头和结尾 重载所有这些方法，可以添加类型 LinkedListNode<T>或类型 T 的对象。如果传送 T 对象，则创建一个新 LinkedListNode<T>对象

(续表)

LinkedList<T>的成员	说 明
Remove() RemoveFirst() RemoveLast()	Remove()、RemoveFirst()和 RemoveLast()方法从链表中删除节点。RemoveFirst()删除第一个元素，RemoveLast()删除最后一个元素。Remove()需要搜索一个对象，从链表中删除匹配该对象的第一个节点
Clear()	从链表中删除所有的节点
Contains()	在链表中搜索一个元素，如果找到该元素，就返回 true，否则返回 false
Find()	从链表的开头开始搜索传送给它的元素，并返回一个 LinkedListNode<T>
FindLast()	与 Find()方法类似，但从链表的结尾开始搜索

示例应用程序使用了一个链表 `LinkedList<T>` 和一个列表 `List<T>`。链表包含文档，这与上一个例子相同，但文档有一个额外的优先级。在链表中，文档按照优先级来排序。如果多个文档的优先级相同，则这些元素就按照文档的插入时间来排序。

图 10-4 描述了示例应用程序中的集合。`LinkedList<Document>` 是一个包含所有 `Document` 对象的链表，该图显示了文档的标题和优先级。标题指出了文档添加到链表中的时间。第一个添加的文档的标题是 `One`。第二个添加的文档的标题是 `Two`，依此类推。可以看出，文档 `One` 和 `Four` 有相同的优先级 8，因为 `One` 在 `Four` 之前添加，所以 `One` 放在链表的前面。

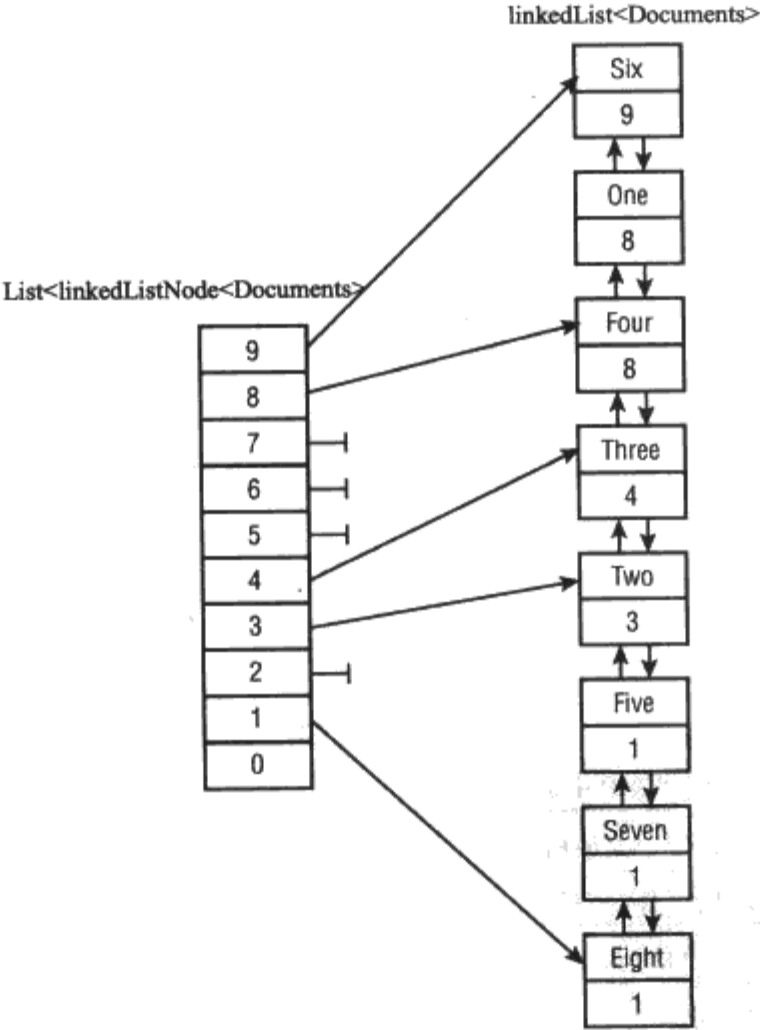


图 10-4

在链表中添加新文档时，它们应放在优先级相同的最后一个文档后面。集合 `LinkedList<Document>` 包含 `LinkedListNode<Document>` 类型的元素。类 `LinkedListNode<T>` 添加 `Next` 和 `Previous` 属性，使搜索过程能从一个节点移动到下一个节点上。要引用这类元素，应把 `List<T>` 定义为 `List<LinkedListNode<Document>>`。为了快速访问每个优先级的最后一个文档，集合

`List<LinkedListNode>` 应最多包含 10 个元素，每个元素都引用不同优先级的最后一个文档。在后面的讨论中，对不同优先级的最后一个文档的引用称为优先级节点。

在上面的例子中，`Document` 类扩展为包含优先级。优先级用类的构造函数设置：

```
public class Document
{
    private string title;
    public string Title
    {
        get
        {
            return title;
        }
    }

    private string content;
    public string Content
    {
        get
        {
            return content;
        }
    }
}
```

```
private byte priority;
public byte Priority
{
    get
    {
        return priority;
    }
}
```

```
public Document(string title, string content, byte priority)
{
    this.title = title;
    this.content = content;
    this.priority = priority;
}
```

解决方案的核心是 `PriorityDocumentManager` 类。这个类很容易使用。在这个类的公共接口中，可以把新的 `Document` 元素添加到链表中，检索第一个文档，为了便于测试，它还提供了一个方法，在元素链接到链表中时，该方法可以显示集合中的所有元素。

`PriorityDocumentManager` 类包含两个集合。`LinkedList<Document>` 类型的集合包含所有的文档。`List<LinkedListNode<Document>>` 类型的集合包含最多 10 个元素的引用，它们是添加指定优先级的新文档的入口点。这两个集合变量都用 `PriorityDocumentManager` 类的构造函数来初始化。列表集合也用 `null` 初始化：

```
public class PriorityDocumentManager
{
    private readonly LinkedList<Document> documentList;

    // priorities 0..9
    private readonly List<LinkedListNode<Document>> priorityNodes;
```

```

public PriorityDocumentManager()
{
    documentList = new LinkedList<Document>();

    priorityNodes = new List<LinkedListNode<Document>>(10);
    for (int i = 0; i < 10; i++)
    {
        priorityNodes.Add(new LinkedListNode<Document>(null));
    }
}

```

在类的公共接口中，有一个方法 `AddDocument()`。它只是调用私有方法 `AddDocumentToPriorityNode()`。把实现代码放在另一个方法中的原因是，`AddDocumentToPriorityNode()` 可以递归调用，如后面所示。

```

public void AddDocument(Document d)
{
    if (d == null) throw new ArgumentNullException("d");
    AddDocumentToPriorityNode(d, d.Priority);
}

```

在 `AddDocumentToPriorityNode()` 的实现代码中，第一个操作是检查优先级是否在允许的范围内。这里允许的范围是 0~9。如果传送了错误的值，就会抛出 `ArgumentException` 类型的异常。

接着检查是否已经有一个优先级节点与所传送的优先级相同。如果在列表集合中没有这样的优先级节点，就递归调用 `AddDocumentToPriorityNode()`，递减优先级值，检查是否有低一级的优先级节点。

如果优先级节点的优先级值与所传送的优先级值不同，也没有比该优先级值更低的优先级节点，就可以调用 `AddLast()` 方法，将文档安全地添加到链表的尾部。另外，链表节点由负责指定文档优先级的优先级节点引用。

如果存在这样的优先级节点了，就可以在链表中找到插入文档的位置。这里必须区分是存在指定优先级值的优先级节点，还是引用文档的优先级节点有较低的优先级值。对于第一种情况，可以把新文档插入由优先级节点引用的位置后面。因为优先级节点总是引用指定优先级值的最后一个文档，所以必须设置优先级节点的引用。如果引用文档的优先级节点有较低的优先级值，情况会比较复杂。这里新文档必须插入优先级值与优先级节点相同的所有文档的前面。为了找到优先级值相同的第一个文档，要通过一个 `while` 循环，使用 `Previous` 属性迭代所有的链表节点，直到找到一个优先级值不同的链表节点为止。这样，就找到了文档的插入位置，并可以设置优先级节点。

```

private void AddDocumentToPriorityNode(Document doc, int priority)
{
    if (priority > 9 || priority < 0)
        throw new ArgumentException("Priority must be between 0 and 9");

    if (priorityNodes[priority].Value == null)
    {
        priority--;
        if (priority >= 0)
        {
            // check for the next lower priority
            AddDocumentToPriorityNode(doc, priority);
        }
    }
}

```



```

else // now no priority node exists with the same priority or lower
    // add the new document to the end
    {
        documentList.AddLast(doc);
        priorityNodes[doc.Priority] = documentList.Last;
    }
    return;
}
else // a priority node exists
{
    LinkedListNode<Document> priorityNode = priorityNodes[priority];
    if (priority == doc.Priority) // priority node with the same
        // priority exists
    {
        documentList.AddAfter(priorityNode, doc);

        // set the priority node to the last document with the same priority
        priorityNodes[doc.Priority] = priorityNode.Next;
    }
    else // only priority node with a lower priority exists
    {
        // get the first node of the lower priority
        LinkedListNode<Document> firstPriorityNode = priorityNode;

        while (firstPriorityNode.Previous != null &&
            firstPriorityNode.Previous.Value.Priority ==
                priorityNode.Value.Priority)
        {
            firstPriorityNode = priorityNode.Previous;
        }

        documentList.AddBefore(firstPriorityNode, doc);

        // set the priority node to the new value
        priorityNodes[doc.Priority] = firstPriorityNode.Previous;
    }
}
}
}

```

现在还剩下一个简单的方法没有讨论。DisplayAllNodes()只是在一个 foreach 循环中，把每个文档的优先级和标题显示在控制台上。

GetDocument 方法从链表中返回第一个文档(优先级最高的文档)，并从链表中删除它：

```

public void DisplayAllNodes()
{
    foreach (Document doc in documentList)
    {
        Console.WriteLine("priority: {0}, title {1}", doc.Priority, doc.Title);
    }
}

// returns the document with the highest priority (that's first
// in the linked list)
public Document GetDocument()
{
    Document doc = documentList.First.Value;
    documentList.RemoveFirst();
    return doc;
}
}

```

在 Main() 方法中, PriorityDocumentManager 用于演示其功能。在链表中添加 8 个优先级不同的新文档, 再显示整个链表:

```
static void Main()
{
    PriorityDocumentManager pdm = new PriorityDocumentManager();
    pdm.AddDocument(new Document("one", "Sample", 8));
    pdm.AddDocument(new Document("two", "Sample", 3));
    pdm.AddDocument(new Document("three", "Sample", 4));
    pdm.AddDocument(new Document("four", "Sample", 8));
    pdm.AddDocument(new Document("five", "Sample", 1));
    pdm.AddDocument(new Document("six", "Sample", 9));
    pdm.AddDocument(new Document("seven", "Sample", 1));
    pdm.AddDocument(new Document("eight", "Sample", 1));

    pdm.DisplayAllNodes();
}
```

在处理好的结果中, 文档先按优先级排序, 再按添加文档的时间排序:

```
priority: 9, title six
priority: 8, title one
priority: 8, title four
priority: 4, title three
priority: 3, title two
priority: 1, title five
priority: 1, title seven
priority: 1, title eight
```

## 10.6 有序表

如果需要排好序的表, 可以使用 SortedList<TKey, TValue>。这个类按照键给元素排序。

下面的例子创建了一个有序表, 其中键和值都是 string 类型。默认的构造函数创建了一个空表, 再用 Add() 方法添加两本书。使用重载的构造函数, 可以定义有序表的容量, 传送执行了 IComparer<TKey> 接口的对象, 用于给有序表中的元素排序。

Add() 方法的第一个参数是键(书名), 第二个参数是值(ISBN 号)。除了使用 Add() 方法之外, 还可以使用索引器将元素添加到有序表中。索引器需要把键作为索引参数。如果键已存在, 那么 Add() 方法就抛出一个 ArgumentException 类型的异常。如果索引器使用相同的键, 就用新值替代旧值。

```
SortedList<string, string> books = new SortedList<string, string>();
books.Add(".NET 2.0 Wrox Box", "978-0-470-04840-5");
books.Add("Professional C# 2005 with .NET 3.0", "978-0-470-12472-7");

books["Beginning Visual C# 2005"] = "978-0-7645-4382-1";
books["Professional C# 2008"] = "978-0-470-19137-6";
```

可以使用 foreach 语句迭代有序表。枚举器返回的元素是 KeyValuePair<TKey, TValue> 类型, 其中包含了键和值。键可以用 Key 属性访问, 值用 Value 属性访问。

```
foreach (KeyValuePair<string, string> book in books)
{
    Console.WriteLine("{0}, {1}", book.Key, book.Value);
}
```

迭代语句会按键的顺序显示书名和 ISBN 号：

```
.NET 2.0 Wrox Box, 978-0-470-04840-5
Beginning Visual C# 2005, 978-0-7645-4382-1
Professional C# 2005 with .NET 3.0, 978-0-470-12472-7
Professional C# 2008, 978-0-470-19137-6
```

也可以使用 Values 和 Keys 属性访问值和键。Values 属性返回 IList<TValue>，Keys 属性返回 IList<TKey>，所以，可以在 foreach 中使用这些属性：

```
foreach (string isbn in books.Values)
{
    Console.WriteLine(isbn);
}

foreach (string title in books.Keys)
{
    Console.WriteLine(title);
}
```

第一个循环显示值，第二个循环显示键：

```
978-0-470-04840-5
978-0-7645-4382-1
978-0-470-12472-7
978-0-470-19137-6
.NET 2.0 Wrox Box
Beginning Visual C# 2005
Professional C# 2005 with .NET 3.0
Professional C# 2008
```

SortedList<TKey, TValue>类的属性如表 10-7 所示。

表 10-7

SortedList 的属性	说 明
Capacity	使用 Capacity 属性可以获取和设置有序表能包含的元素个数。该属性与 List<T>类似：默认构造函数会创建一个空表，添加第一个元素会使有序表的容量变成 4 个元素，之后其容量会根据需要成倍增加
Comparer	Comparer 属性返回与有序表相关的比较器。可以在构造函数中传送该比较器。默认的比较器会调用 IComparable<TKey>接口的 CompareTo()方法来比较键。键类型执行了这个接口，也可以创建定制的比较器
Count	Count 属性返回有序表中的元素个数
Item	使用索引器可以访问有序表中的元素。索引器的参数类型由键类型定义
Keys	Keys 属性返回包含所有键的 IList<TKey>
Values	Values 属性返回包含所有值的 IList<TValue>

SortedList<T>类型的方法类似于本章前面介绍的其他集合。见表 10-8。区别是 SortedList<T>需要一个键和一个值。

表 10-8

SortedList 的方法	说 明
Add()	把带有键和值的元素放在有序表中
Remove(), RemoveAt()	Remove()方法需要从有序表中删除的元素的键。使用 RemoveAt()方法可以删除指定索引的元素
Clear()	删除有序表中的所有元素
ContainsKey(), ContainsValue()	ContainsKey()和 ContainsValue()方法检查有序表是否包含指定的键或值，并返回 true 或 false
IndexOfKey(), IndexOfValue()	IndexOfKey()和 IndexOfValue()方法检查有序表是否包含指定的键或值，并返回基于整数的索引
TrimExcess()	重新设置集合的大小，将容量改为需要的元素个数
TryGetValue()	使用 TryGetValue()方法可以尝试获得指定键的值。如果键不存在，这个方法就返回 false。如果键存在，就返回 true，并把值返回为 out 参数

注意：  
除了泛型 SortedList<TKey, TValue>之外，还有一个对应的非泛型有序表 SortedList。

10.7 字典

字典表示一种非常复杂的数据结构，这种数据结构允许按照某个键来访问元素。字典也称为映射或散列表。字典的主要特性是根据键快速查找值。也可以自由添加和删除元素，这有点像 List<T>，但没有在内存中移动后续元素的性能开销。

图 10-5 是字典的一个简化表示。其中 employee-id，如 B4711，是添加到字典中的键。键会转换为一个散列。利用散列创建一个数字，它将索引和值关联起来。然后索引包含一个到值的链接。该图做了简化处理，因为一个索引项可以关联多个值，索引可以存储为一个树形结构。

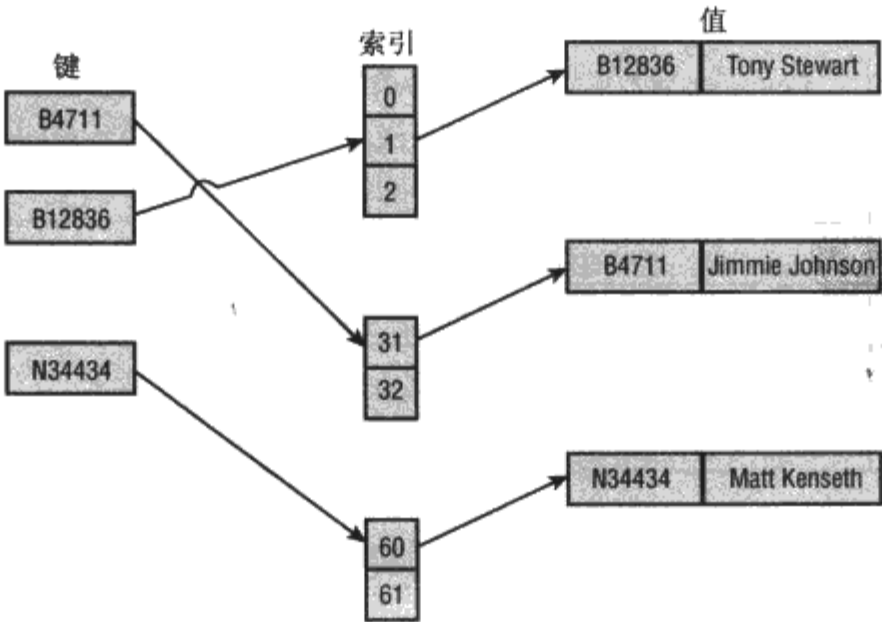


图 10-5

.NET Framework 提供了几个字典类。可以使用的最主要的类是 `Dictionary<TKey, TValue>`。这个类的属性和方法与上面的 `SortedList<TKey, TValue>` 几乎完全相同，这里不再赘述。

### 10.7.1 键的类型

用作字典中键的类型必须重写 `Object` 类的 `GetHashCode()` 方法。只要字典类需要确定元素的位置，就要调用 `GetHashCode()` 方法。`GetHashCode()` 方法返回的 `int` 由字典用于计算放置元素的索引。这里不介绍这个算法。我们只需知道，它涉及到素数，所以字典的容量是一个素数。

`GetHashCode()` 方法的实现代码必须满足如下要求：

- 相同的对象应总是返回相同的值。
- 不同的对象可以返回相同的值。
- 应执行得比较快，计算的开销不大。
- 不能抛出异常。
- 应至少使用一个实例字段。
- 散列码值应平均分布在 `int` 可以存储的整个数字区域上。
- 散列码最好在对象的生存期中不发生变化。

**提示：**

字典的性能取决于 `GetHashCode()` 方法的实现代码。

为什么要使散列码值平均分布在整数的取值区域内？如果两个键返回的散列会得到相同的索引，则字典类就必须寻找最近的可用自由单元来存储第二个数据项，这需要进行一定的搜索，以便以后检索这一项。显然这会降低性能，如果许多键都有相同的索引，这类冲突就比较多。根据 Microsoft 的部分算法的工作方式，计算出来的散列值平均分布在 `int.MinValue` 和 `int.MaxValue` 之间时，这种风险会降低到最小。

除了实现 `GetHashCode()` 方法之外，键类型还必须执行 `IEquality.Equals()` 方法，或重写 `Object` 类的 `Equals()` 方法。因为不同的键对象可能返回相同的散列码，所以字典使用 `Equals()` 方法来比较键。字典检查两个键 A 和 B 是否相等，并调用 `A.Equals(B)`。这表示必须确保下述条件总是成立：

**提示**

如果 `A.Equals(B)` 返回 `true`，则 `A.GetHashCode()` 和 `B.GetHashCode()` 必须总是返回相同的散列码。

这似乎有点奇怪，但它非常重要。如果设计出某种重写这些方法的方式，使上面的条件并不总是成立，把这个类的实例用作键的字典就不能正常工作，而是会发生可笑的事情。例如，把一个对象放在字典中后，就再也找不到它，或者试图查找某项，却返回了错误的项。

**注意：**

因此，如果为 `Equals()` 方法提供了重写版本，但没有提供 `GetHashCode()` 的重写版本，C# 编译器就会显示一个编译警告。



对于 `System.Object`，这个条件为 `true`，因为 `Equals()` 方法只是比较引用，`GetHashCode()` 总是返回一个仅基于对象地址的散列。这说明，如果散列表基于一个键，而该键没有重写这些方法，这个散列表就能正常工作。但是，这么做的问题是，只有对象完全相同，键才被认为是相等的。也就是说，把一个对象放在字典中时，必须将它与该键的引用关联起来。也不能在以后用相同的值实例化另一个键对象。如果没有重写 `Equals()` 和 `GetHashCode()`，类型在字典中使用时就不太方便。

另外，`System.String` 执行了 `IEquatable` 接口，并重载了 `GetHashCode()` 方法。`Equals()` 提供了值的比较，`GetHashCode()` 根据字符串的值返回一个散列。因此，字典中把字符串用作键是非常方便的。

数字类型，如 `Int32`，也执行了 `IEquatable` 接口，并重载了 `GetHashCode()` 方法。但是这些类型返回的散列码只是映射到值上。如果希望用作键的数字本身没有分布在可能的整数值区域内，把整数用作键就不能满足键值平均分布、获得最佳性能的规则。`Int32` 并不适合在字典中使用。

如果所使用的键类型没有执行 `IEquatable` 接口，并根据存储在字典中的键值重载 `GetHashCode()`，就可以创建一个执行了 `IEqualityComparer<T>` 接口的比较器。`IEqualityComparer<T>` 接口定义了 `GetHashCode()` 和 `Equals()` 方法，并将对象作为参数，因此可以提供与对象类型不同的实现方式。`Dictionary<TKey, TValue>` 构造函数的一个重载版本允许传送实现了 `IEqualityComparer<T>` 接口的对象。如果把这个对象赋予字典，该类就用于生成散列码并比较键。

### 10.7.2 字典示例

字典示例程序建立了一个员工字典。该字典是用 `EmployeeId` 对象来索引的，存储在字典中的每个数据项都是一个 `Employee` 对象，该对象存储员工的详细数据。

`EmployeeId` 结构定义了字典中使用的键，该结构的成员是表示员工的一个前缀字符和一个数字。这两个变量都是只读的，只能在构造函数中初始化。字典中的键不应改变，这是必须保证的。字段在构造函数中填充。`ToString()` 方法重载为获得员工 ID 的字符串表示。与键类型的要求一样，`EmployeeId` 也要执行 `IEquatable` 接口，重写 `GetHashCode()` 方法。

```
[Serializable]
public struct EmployeeId : IEquatable<EmployeeId>
{
    private readonly char prefix;
    private readonly int number;

    public EmployeeId(string id)
    {
        if (id == null) throw new ArgumentNullException("id");

        prefix = (id.ToUpper())[0];
        int numLength = id.Length - 1;
        number = int.Parse(id.Substring(1, numLength > 6 ? 6 : numLength));
    }

    public override string ToString()
    {
        return prefix.ToString() + string.Format("{0,6:000000}", number);
    }

    public override int GetHashCode()
```

```

    {
        return (number ^ number << 16) * 0x15051505;
    }
    public bool Equals(EmployeeId other)
    {
        return (prefix == other.prefix && number == other.number);
    }
}

```

由 `IEquatable<T>` 接口定义的 `Equals()` 方法比较两个 `EmployeeId` 对象的值, 如果这两个值相同, 就返回 `true`。除了执行 `IEquatable<T>` 接口中的 `Equals()` 方法之外, 还可以重写 `Object` 类中的 `Equals()` 方法。

```

public bool Equals(EmployeeId other)
{
    if (other == null) return false;
    return (prefix == other.prefix && number == other.number);
}

```

由于数字是可变的, 因此员工可以有 1~190000 的一个值。这并没有填满整数取值区域。`GetHashCode()` 使用的算法将数字向左移动 16 位, 再与原来的数字进行异或操作, 最后将结果乘以十六进制数 15051505。散列码在整数取值区域上的分布相当均匀:

```

public override int GetHashCode()
{
    return (number ^ number << 16) * 0x15051505;
}

```

注意:

在 Internet 上, 有许多更复杂的算法, 能使散列码在整数取值区域上更好地分布。也可以使用字符串的 `GetHashCode()` 方法来返回散列。

`Employee` 类是一个简单的实体类, 包含员工的姓名、薪水和 ID。构造函数初始化了所有的值, 方法 `ToString()` 返回一个实例的字符串表示。`ToString()` 方法的实现代码使用格式字符串创建字符串表示, 以提高性能。

```

[Serializable]
public class Employee
{
    private string name;
    private decimal salary;
    private readonly EmployeeId id;

    public Employee(EmployeeId id, string name, decimal salary)
    {
        this.id = id;
        this.name = name;
        this.salary = salary;
    }

    public override string ToString()
    {
        return String.Format("{0}: {1, -20} {2:C}",
            id.ToString(), name, salary);
    }
}

```

在示例程序的 `Main()` 方法中，创建一个新的 `Dictionary<TKey, TValue>` 实例，其中键是 `EmployeeId` 类型，值是 `Employee` 类型。构造函数指定了 31 个元素的容量。注意容量一般是素数。但如果指定了一个不是素数的值，也不需要担心。`Dictionary<TKey, TValue>` 类会使用传送给构造函数的整数后面的一个素数，来指定容量。用 `Add()` 方法创建员工对象和 ID，并添加到字典中。除了 `Add()` 方法外，还可以使用索引器，将键和值添加到字典中，如员工 Carl 和 Matt 所示：

```
static void Main()
{
    Dictionary<EmployeeId, Employee> employees =
        new Dictionary<EmployeeId, Employee>(31);

    EmployeeId idJeff = new EmployeeId("C7102");
    Employee jeff = new Employee(idJeff,
        "Jeff Gordon", 5164580.00m);
    employees.Add(idJeff, jeff);
    Console.WriteLine(jeff);

    EmployeeId idTony = new EmployeeId("C7105");
    Employee tony = new Employee(idTony,
        "Tony Stewart", 4814200.00m);
    employees.Add(idTony, tony);
    Console.WriteLine(tony);

    EmployeeId idDenny = new EmployeeId("C8011");
    Employee denny = new Employee(idDenny,
        "Denny Hamlin", 3718710.00m);
    employees.Add(idDenny, denny);
    Console.WriteLine(denny);

    EmployeeId idCarl = new EmployeeId("F7908");
    Employee carl = new Employee(idCarl,
        "Carl Edwards", 3285710.00m);
    employees[idCarl] = carl;
    Console.WriteLine(carl);

    EmployeeId idMatt = new EmployeeId("F7203");
    Employee matt = new Employee(idMatt,
        "Matt Kenseth", 4520330.00m);
    employees[idMatt] = matt;
    Console.WriteLine(matt);
}
```

将数据项添加到字典中后，在 `while` 循环中读取字典中的员工。让用户输入一个员工号，把该号码存储在变量 `userInput` 中。用户输入 X 即可退出程序。如果输入的键在字典中，就使用 `Dictionary<TKey, TValue>` 类的 `TryGetValue()` 方法检查它。如果找到了该键，`TryGetValue()` 方法就返回 `true`，否则返回 `false`。如果找到了与键关联的值，该值就存储在 `employee` 变量中，并写入控制台。

**注意：**

也可以使用 `Dictionary<TKey, TValue>` 类的索引器替代 `TryGetValue()` 方法，来访问存储在字典中的值。但是，如果没有找到键，索引器会抛出 `KeyNotFoundException` 类型的异常。

```
while (true)
{
    Console.Write("Enter employee id (X to exit)> ");
}
```

```
string userInput = Console.ReadLine();
userInput = userInput.ToUpper();
if (userInput == "X") break;

EmployeeId id = new EmployeeId(userInput);

Employee employee;
if (!employees.TryGetValue(id, out employee))
{
    Console.WriteLine("Employee with id " + "{0} does not exist",
        id);
}
else
{
    Console.WriteLine(employee);
}
}
```

运行应用程序，得到如下输出：

```
Enter employee ID (format:A999999, X to exit) > C7102
C007102: Jeff Gordon $5,164,580.00
Enter employee ID (format:A999999, X to exit) > F7908
F007908: Carl Edwards $3,285,710.00
Enter employee ID (format:A999999, X to exit) > X
```

10.7.3 Lookup 类

Dictionary<TKey, TValue>只为每个键支持一个值。新类 Lookup<TKey, TElement>是.NET 3.5 中新增的，它类似于 Dictionary<TKey, TValue>，但把键映射到一个值集上。这个类在程序集 System.Core 中实现，用 System.Linq 命名空间定义。

Lookup<TKey, TElement>的方法和属性如表 10-9 所示。

表 10-9

Lookup<TKey, TElement>的方法和属性	说 明
Count	属性 Count 返回集合中的元素个数
Item	使用索引器可以根据键访问特定的元素。因为同一个键可以对应多个值，所以这个属性返回所有值的枚举
Contain()	方法 Contains()根据是否用 key 参数传送元素，返回一个布尔值
ApplyResultSelector()	ApplyResultSelector()根据传送给它的转换函数，转换每一项，返回一个集合

Lookup<TKey, TElement>不能像一般的字典那样创建，而必须调用方法 ToLookup()，它返回一个 Lookup<TKey, TElement>对象。方法 ToLookup()是一个扩展方法，可以用于实现了 IEnumerable<T>的所有类。在下面的例子中，填充了一列 Racer 对象。List<T>实现了 IEnumerable<T>，所以可以在赛手列表上调用方法 ToLookup()。这个方法需要一个 Func<TSource, TKey>类型的委托，Func<TSource, TKey>类型定义了键的选择器。这里使用λ表达式 r=>r.Country，根据国家来选择赛手。Foreach 循环只使用索引器访问来自奥地利的赛手。



提示：  
扩展方法详见第 11 章，λ表达式参见第 7 章。

```
List < Racer > racers = new List < Racer > ();
racers.Add(new Racer( "Jacques", "Villeneuve",
    "Canada", 11));
racers.Add(new Racer( "Alan", "Jones",
    "Australia", 12));
racers.Add(new Racer( "Jackie", "Stewart",
    "United Kingdom", 27));
racers.Add(new Racer( "James", "Hunt",
    "United Kingdom", 10));
racers.Add(new Racer( "Jack", "Brabham",
    "Australia", 14));

Lookup < string, Racer > lookupRacers =
    (Lookup < string, Racer > )
    racers.ToLookup(r => r.Country);

foreach (Racer r in lookupRacers[ "Australia" ])
{
    Console.WriteLine(r);
}
```

结果显示了来自奥地利的赛手：

```
Alan Jones
Jack Brabham
```

10.7.4 其他字典类

Dictionary<TKey, TValue>是 Framework 中的一个主要字典类，还有其他一些类，当然也有一些非泛型的字典类。

基于 Object 类型的字典和.NET 1.0 以来可以使用的字典类如表 10-10 所示。

表 10-10

非泛型字典	说 明
Hashtable	Hashtable 是.NET 1.0 中最常用的字典类。键和值都基于 Object 类型
ListDictionary	ListDictionary 位于命名空间 System.Collections.Specialized，如果使用的元素少于 10 个，它就好比 Hashtable 快。ListDictionary 实现为链表
HybridDictionary	如果集合比较小，HybridDictionary 就使用 ListDictionary，当集合增大时，就改为使用 Hashtable。如果事先不知道元素个数，就可以使用 HybridDictionary
NameObjectCollectionBase	NameObjectCollectionBase 是一个抽象基类，将字符串类型的键关联到 Object 类型的值上。它可以用作定制字符串/Object 集合的基类。这个类在内部使用 Hashtable
NameValueCollection	NameValueCollection 派生于 NameObjectCollection。它的键和值都是字符串类型。这个类有一个特性：多个值可以使用同一个键

自从.NET 2.0 以来，泛型字典优先于基于对象的字典，如表 10-11 所示。



表 10-11

泛型字典	说 明
Dictionary<TKey, TValue>	Dictionary<TKey, TValue>是一般用途的字典，将键映射到值上
SortedDictionary<TKey, TValue>	这是一个二叉搜索树，其中的元素根据键来排序。键的类型必须执行 IComparable<TKey>接口。如果键的类型不能排序，还可以创建一个执行了 IComparer<TKey>接口的比较器，将比较器用作有序字典的构造函数中的一个参数

SortedDictionary<TKey, TValue>和 SortedList<TKey, TValue>的功能类似。但因为 SortedList<TKey, TValue>实现为一个基于数组的链表，而 SortedDictionary<TKey, TValue>实现为一个字典，所以它们有不同的特性。

- SortedList<TKey, TValue>使用的内存比 SortedDictionary<TKey, TValue>少。
- SortedDictionary<TKey, TValue>的元素插入和删除速度比较快。
- 在用已排好序的数据填充集合时，若不需要修改容量，SortedList<TKey, TValue>就比較快。

提示：

SortedList 使用的内存比 SortedDictionary 少，但 SortedDictionary 在插入和删除未排序的数据时比较快。

10.8 HashSet

.NET 3.5 在 System.Collections.Generic 命名空间中包含一个新的集合类：HashSet<T>。这个集合类包含不重复项的无序列表。这种集合称为“集(set)”。集是一个保留字，所以该类有另一个名称 HashSet<T>。这个名称很容易理解，因为这个集合基于散列值，插入元素的操作非常快，不需要像 List<T>类那样重排集合。

HashSet<T>类提供的方法可以创建合集和交集。表 10-12 列出了改变集的值的方法。

表 10-12

HashSet<T>的修改方法	说 明
Add()	如果某元素不在集合中，Add()方法就把该元素添加到集合中。在其返回值 Boolean 中，返回元素是否添加的信息
Clear()	方法 Clear()删除集合中的所有元素
Remove()	Remove()方法删除指定的元素
RemoveWhere()	RemoveWhere()方法需要一个 Predicate<T>委托作为参数。删除满足谓词条件的所有元素
CopyTo()	CopyTo()把集合中的元素复制到一个数组中
ExceptWith()	ExceptWith()方法把一个集合作为参数，从集中删除该集合中的所有元素
IntersectWith()	IntersectWith()修改了集，仅包含所传送的集合和集中都有的元素
UnionWith()	UnionWith()方法把传送为参数的集合中的所有元素添加到集中

表 10-13 列出了仅返回集的信息、不修改元素的方法。

表 10-13

HashSet<T>的验证方法	说 明
Contains()	如果所传送的元素在集合中，方法 Contains()就返回 true
IsSubsetOf()	如果参数传送的集合是集的一个子集，方法 IsSubsetOf()就返回 true
IsSupersetOf()	如果参数传送的集合是集的一个超集，方法 IsSupersetOf()就返回 true
Overlaps()	如果参数传送的集合中至少有一个元素与集中的元素相同，Overlaps()就返回 true
SetEquals()	如果参数传送的集合和集包含相同的元素，方法 SetEquals()就返回 true

在示例代码中，创建了 3 个字符串类型的新集，并用一级方程式汽车填充。HashSet<T>类实现了 ICollection<T>接口。但是在该类中，Add()方法是显式实现的，还提供了另一个 Add()方法。Add()方法的区别是返回类型，它返回一个布尔值，说明是否添加了元素。如果该元素已经在集中，就不添加它，并返回 false。

```
HashSet < string > companyTeams =
    new HashSet < string > ()
    { "Ferrari", "McLaren", "Toyota", "BMW",
      "Renault", "Honda" };
HashSet < string > traditionalTeams =
    new HashSet < string > ()
    { "Ferrari", "McLaren" };
HashSet < string > privateTeams =
    new HashSet < string > ()
    { "Red Bull", "Toro Rosso", "Spyker",
      "Super Aguri" };

if (privateTeams.Add("Williams"))
    Console.WriteLine("Williams added");
if (!companyTeams.Add("McLaren"))
    Console.WriteLine(
        "McLaren was already in this set");
```

两个 Add()方法的输出写到控制台上：

```
Williams added
McLaren was already in this set
```

方法 IsSubsetOf()和 IsSupersetOf()比较集和实现了 IEnumerable<T>接口的集合，返回一个布尔结果。这里，IsSubsetOf()验证 traditionalTeams 中的每个元素是否都包含在 companyTeams 中，IsSupersetOf()验证 traditionalTeams 是否没有与 companyTeams 比较的额外元素。

```
if (traditionalTeams.IsSubsetOf(companyTeams))
{
    Console.WriteLine("traditionalTeams is " +
        "subset of companyTeams");
}
if (companyTeams.IsSupersetOf(traditionalTeams))
{
    Console.WriteLine(
        "companyTeams is a superset of " +
```

```
    "traditionalTeams");
}
```

这个验证的结果如下：

```
traditionalTeams is a subset of companyTeams
companyTeams is a superset of traditionalTeams
```

Williams 也是一个传统队，因此这个队添加到 traditionalTeams 集合中：

```
traditionalTeams.Add("Williams");
if (privateTeams.Overlaps(traditionalTeams))
{
    Console.WriteLine("At least one team is " +
        "the same with the traditional " +
        "and private teams");
}
```

这有一个重叠，所以结果如下：

```
At least one team is the same with the traditional and private teams.
```

调用 UnionWith() 方法，给变量 allTeams 填充了 companyTeams、PrivateTeams 和 traditionalTeams 的合集：

```
HashSet < string > allTeams =
    new HashSet < string > (companyTeams);
allTeams.UnionWith(privateTeams);
allTeams.UnionWith(traditionalTeams);

Console.WriteLine();
Console.WriteLine("all teams");
foreach (var team in allTeams)
{
    Console.WriteLine(team);
}
```

这里返回所有的队，但每个队都只列出一次，因为集只包含唯一值：

```
Ferrari
McLaren
Toyota
BMW
Renault
Honda
Red Bull
Toro Rosso
Spyker
Super Aguri
Williams
```

方法 ExceptWith() 从 allTeams 集中删除所有的私人队：

```
allTeams.ExceptWith(privateTeams);
Console.WriteLine();
Console.WriteLine("no private team left");
foreach (var team in allTeams)
{
    Console.WriteLine(team);
}
```

集合中的其他元素不包含私人队：

Ferrari  
McLaren  
Toyota  
BMW  
Renault  
Honda

10.9 位数组

如果需要处理许多位，就可以使用类 BitArray 和结构 BitVector32。BitArray 位于命名空间 System.Collections，BitVector32 位于命名空间 System.Collections.Specialized。这两种类型最重要的区别是，BitArray 可以重新设置大小，如果事先不知道需要的位数，就可以使用 BitArray，它可以包含非常多的位。BitVector32 是基于栈的，因此比较快。BitVector32 仅包含 32 位，存储在一个整数中。

10.9.1 BitArray

类 BitArray 是一个引用类型，包含一个 int 数组，每 32 位使用一个新整数。这个类的成员如表 10-14 所示。

表 10-14

BitArray 类的成员	说 明
Count, Length	Count 和 Length 的 get 访问器返回数组中的位数。使用 Length 属性还可以定义新的数组大小，重新设置集合的大小
Item	可以使用索引器读写数组中的位。索引器是 bool 类型
Get(), Set()	除了使用索引器之外，还可以使用 Get 和 Set 方法访问数组中的位
SetAll()	根据传送给该方法的参数，设置所有位的值
Not()	倒转数组中所有位的值
And(), Or(), Xor()	使用 And()、Or 和 Xor()方法，可以合并两个 BitArray 对象。And()方法执行二元 AND，只有两个输入数组的位都设置为 1，结果位才是 1。Or()方法执行二元 OR，只要有一个输入数组的位设置为 1，结果位就是 1。Xor()方法是异或操作，只有一个输入数组的位设置为 1，结果位才是 1

注意：  
第 6 章介绍了处理位的 C#运算符。

帮助方法 DisplayBits()迭代 BitArray，根据位的设置情况，在控制台上显示 1 或 0：

```
static void DisplayBits(BitArray bits)
{
```

```

foreach (bool bit in bits)
{
    Console.Write(bit ? 1 : 0);
}
}

```

演示 `BitArray` 类的示例创建了一个包含 8 位的数组，其索引是 0~7。`SetAll()` 方法把这 8 位都设置为 `true`。接着 `Set()` 方法把 1 位设置为 `false`。除了 `Set()` 方法之外，还可以使用索引器，例如下面的索引 5 和 7：

```

BitArray bits1 = new BitArray(8);
bits1.SetAll(true);
bits1.Set(1, false);
bits1[5] = false;
bits1[7] = false;
Console.Write("initialized:");
DisplayBits(bits1);
Console.WriteLine();

```

这是初始化位的显示结果：

```
initialized: 10111010
```

`Not()` 方法会倒转 `BitArray` 的位：

```

Console.Write(" not ");
DisplayBits(bits1);
bits1.Not();
Console.Write(" = ");
DisplayBits(bits1);
Console.WriteLine();

```

`Not()` 方法的结果是所有的位全部倒转过来。如果某位是 `true`，执行 `Not()` 方法的结果就是 `false`，反之亦然。

```
not 10111010 = 01000101
```

这里创建了一个新的 `BitArray`。在构造函数中，使用变量 `bits1` 初始化数组，所以新数组与旧数组有相同的值。接着把位 0、1 和 4 的值设置为不同的值。在使用 `Or()` 方法之前，显示位数组 `bits1` 和 `bits2`。`Or()` 方法将改变 `bits1` 的值：

```

BitArray bits2 = new BitArray(bits1);
bits2[0] = true;
bits2[1] = false;
bits2[4] = true;
DisplayBits(bits1);
Console.Write(" or ");
DisplayBits(bits2);
Console.Write(" = ");
bits1.Or(bits2);
DisplayBits(bits1);
Console.WriteLine();

```

使用 `Or()` 方法时，从两个输入数组中提取设置位。结果是，如果某位在第一个或第二个数组中设置为 `true`，该位在执行 `Or()` 方法后就是 `true`：

```
01000101 or 10001101 = 11001101
```



下面使用 And()方法处理 bits1 和 bits2:

```
DisplayBits(bits2);
Console.Write(" and ");
DisplayBits(bits1);
Console.Write(" = ");
bits2.And(bits1);
DisplayBits(bits2);
Console.WriteLine();
```

And()方法只把在两个输入数组中都设置为 true 的位设置为 true:

10001101 and 11001101 = 10001101

最后使用 Xor()方法进行异或操作:

```
DisplayBits(bits1);
Console.Write(" xor ");
DisplayBits(bits2);
bits1.Xor(bits2);
Console.Write(" = ");
DisplayBits(bits1);
Console.WriteLine();
```

使用 Xor()方法, 只有一个(不能是两个)输入数组的位设置为 1, 结果位才是 1。

11001101 xor 10001101 = 01000000

10.9.2 BitVector32

如果事先知道需要的位数, 就可以使用 BitVector32 结构替代 BitArray。BitVector32 效率较高, 因为它是一个值类型, 在整数栈上存储位。一个整数可以存储 32 位。如果需要更多的位, 就可以使用多个 BitVector32 值或 BitArray。BitArray 可以根据需要增大, 但 BitVector32 不能。

表 10-15 列出了 BitVector32 中与 BitArray 完全不同的成员。

表 10-15

BitVector32 的成员	说 明
Data	属性 Data 把 BitVector32 中的数据返回为整数
Item	BitVector32 的值可以使用索引器设置。索引器是重载的——可以使用掩码或 BitVector32.Section 类型的片断来获取和设置值。
CreateMask()	这是一个静态方法, 用于为访问 BitVector32 中的特定位创建掩码
CreateSection()	这是一个静态方法, 用于创建 32 位中的几个片断

示例代码用默认构造函数创建了一个 BitVector32, 其中所有的 32 位都初始化为 false。接着创建掩码, 以访问位矢量中的位。对 CreateMask()的第一个调用创建了一个访问第一位的掩码。接着调用 CreateMask(), 将 bit1 设置为 1。再次调用 CreateMask(), 把第一个掩码传送为参数, 返回一个访问第二位(它是 2)的掩码。接着, 将 bit3 设置为 4, 以访问位号 3。bit4 的值是 8, 以访问位号 4。

然后, 使用掩码和索引器访问位矢量中的位, 并设置字段:

```
BitVector32 bits1 = new BitVector32();
int bit1 = BitVector32.CreateMask();
int bit2 = BitVector32.CreateMask(bit1);
int bit3 = BitVector32.CreateMask(bit2);
int bit4 = BitVector32.CreateMask(bit3);
int bit5 = BitVector32.CreateMask(bit4);

bits1[bit1] = true;
bits1[bit2] = false;
bits1[bit3] = true;
bits1[bit4] = true;
bits1[bit5] = true;
Console.WriteLine(bits1);
```

BitVector32 有一个重写的 ToString()方法，它不仅显示类名，还显示 1 或 0，来说明位是否设置了，如下所示：

```
BitVector32{0000000000000000000000000000000011101}
```

除了用 `CreateMask()` 方法创建掩码之外，还可以自己定义掩码，也可以一次设置多个位。十六进制值 `abcdef` 与二进制值 `1010 1011 1100 1101 1110 1111` 相同。用这个值定义的所有位都设置了：

```
bits1[0xabcdef] = true;
Console.WriteLine(bits1);
```

在输出中可以验证设置的位:

```
BitVector32{00000000101010111100110111101111}
```

把 32 位分别放在不同的片断中，是非常有用的。例如，IPv4 地址定义为一个 4 字节数，存储在一个整数中。可以定义 4 个片断，把这个整数拆分开。在多播 IP 消息中，使用了几个 32 位值。其中一个 32 位值放在这些片断中：16 位表示源号，8 位表示查询器的查询内部码，3 位表示查询器的健壮变量，1 位表示抑制标志，还有 4 个保留位。也可以定义自己的位含义，以节省内存。

下面的例子模拟接收到值 0x79abcdef，把这个值传送给 BitVector32 的构造函数，并设置位：

```
int received = 0x79abcdef;
BitVector32 bits2 = new BitVector32(received);
Console.WriteLine(bits2);
```

在控制台上显示了初始化的位:

```
BitVector32{01111001101010111100110111101111}
```

接着创建 6 个片断。第一个片断需要 12 位,由十六进制值 0xfff 定义(设置了 12 位)。片断 B 需要 8 位,C 片断需要 4 位,D 和 E 片断需要 3 位,F 片断需要 2 位。第一次调用 `CreateSection()`,只是接收 0xfff,为最前面的 12 位分配内存。第二次调用 `CreateSection()`时,将第一个片断传送为变元,使下一个片断从第一个片断的结尾处开始。`CreateSection()`返回一个 `BitVector32`。`Section`类型的值,它包含了该片断的偏移量和掩码。

```
// sections: FF EEE DDD CCCC BBBBBBBB AAAAAAAAAAAAAA
BitVector32.Section sectionA = BitVector32.CreateSection(0xffff);
BitVector32.Section sectionB = BitVector32.CreateSection(0xff, sectionA);
```

```
BitVector32.Section sectionC = BitVector32.CreateSection(0xf, sectionB);
BitVector32.Section sectionD = BitVector32.CreateSection(0x7, sectionC);
BitVector32.Section sectionE = BitVector32.CreateSection(0x7, sectionD);
BitVector32.Section sectionF = BitVector32.CreateSection(0x3, sectionE);
```

把一个 BitVector32.Section 传送给 BitVector32 的索引器，会返回一个 int，它映射到位矢量的片断上。这里使用一个帮助方法 IntToBinaryString()，获得 int 数的字符串表示：

```
Console.WriteLine("Section A: " + IntToBinaryString(bits2[sectionA], true));
Console.WriteLine("Section B: " + IntToBinaryString(bits2[sectionB], true));
Console.WriteLine("Section C: " + IntToBinaryString(bits2[sectionC], true));
Console.WriteLine("Section D: " + IntToBinaryString(bits2[sectionD], true));
Console.WriteLine("Section E: " + IntToBinaryString(bits2[sectionE], true));
Console.WriteLine("Section F: " + IntToBinaryString(bits2[sectionF], true));
```

方法 IntToBinaryString 接收整数中的位，返回一个包含 0 和 1 的字符串表示。在实现代码中迭代整数的 32 位。在迭代过程中，如果位设置为 1，就在 StringBuilder 的后面追加 1，否则，就追加 0。在循环中，移动一位，以检查是否设置了下一位。

```
static string IntToBinaryString(int bits, bool removeTrailingZero)
{
    StringBuilder sb = new StringBuilder(32);
    for (int i = 0; i < 32; i++)
    {
        if ((bits & 0x80000000) != 0)
        {
            sb.Append("1");
        }
        else
        {
            sb.Append("0");
        }
        bits = bits << 1;
    }
    string s = sb.ToString();
    if (removeTrailingZero)
    {
        return s.TrimStart('0');
    }
    else
    {
        return s;
    }
}
```

结果显示了片断 A~F 的位表示，现在可以用传送给位矢量的值来验证了：

```
Section A: 110111101111
Section B: 10111100
Section C: 1010
Section D: 1
Section E: 111
Section F: 1
```

## 10.10 性能

许多集合类都提供了相同的功能，例如，SortedList 与 SortedDictionary 的功能几乎完全相同。但是，其性能常常有很大区别。一个集合使用的内存少，另一个集合的元素检索速度快。在

MSDN 文档中，集合的方法常常有性能提示，给出了以大 O 记号表示的操作时间：

```
O(1)
O(log n)
O(n)
```

O(1)表示无论集合中有多少数据项，这个操作需要的时间都不变。例如，ArrayList 的 Add() 方法就是 O(1)。无论列表中有多少个元素，在列表尾部添加一个新元素的时间都是相同的。Count 属性会给出元素个数，所以很容易找到列表尾部。

O(n)表示对于集合中的每个元素，需要增加的时间量都是相同的。如果需要重新给集合分配内存，ArrayList 的 Add()方法就是 O(n)。改变容量，需要复制列表，复制的时间随元素的增加而线性增加。

O(log n)表示操作需要的时间随集合中元素的增加而增加，但每个元素需要增加的时间不是线性的，而是呈对数曲线。在集合中执行插入操作时，SortedDictionary<TKey,TValue>就是 O(log n)，而 SortedList<TKey,TValue>是 O(n)。这里 SortedDictionary<TKey,TValue>要快得多，因为它在树形结构中插入元素的效率比列表高得多。

表 10-16 列出了集合类及其执行不同操作的性能，例如添加、插入和删除元素。使用这个表可以选择性能最佳的集合类。左列是集合类，Add 列给出了在集合中添加元素所需的时间。List<T>和 HashSet<T>类把 Add 方法定义为在集合中添加元素。其他集合类用不同的方法把元素添加到集合中。例如 Stack<T>类定义了 Push()方法，Queue<T>类定义了 Enqueue()方法。这些信息也列在表中。

如果单元格中有多个大 O 值，表示若集合需要重置大小，该操作就需要一定的时间。例如，在 List<T>类中，添加元素的时间是 O(1)。如果集合的容量不够大，需要重置大小，重置大小的操作就需要 O(n)。集合越大，重置大小操作的时间就越长。最好避免重置集合的大小，而应把集合的容量设置为一个可以包含所有元素的值。

如果单元格的内容是 na，就表示这个操作不能应用于这种集合类型。

表 10-16

集 合	Add	Insert	Remove	Item	Sort	Find
List<T>	如果集合必须重置大小，就是 O(1)或 O(n)	O(n)	O(n)	O(1)	O (n log n), 最 坏 情 况 O(n ^ 2)	O(n)
Stack<T>	Push(), 如果栈必须重置大小，就是 O(1)或 O(n)	na	Pop(), O(1)	na	na	na
Queue<T>	Enqueue(), 如果队列必须重置大小，就是 O(1)或 O(n)	na	Dequeue(), O(1)	na	na	na
HashSet<T>	如果集必须重置大小，就是 O(1)或 O(n)	Add() O(1)或 O(n)	O(1)	na	na	na
LinkedList<T>	AddLast() O(1)	AddAfter() O(1)	O(1)	na	na	O(n)

(续表)

集 合	Add	Insert	Remove	Item	Sort	Find
Dictionary <TKey, TValue>	O(1) 或 O(n)	na	O(1)	O(1)	na	na
SortedDictionary <TKey, TValue>	O(log n)	na	O(log n)	O(log n)	na	na
SortedList <TKey, TValue>	无序数据为 O(n)，如果 必须重置大小，到列表 的尾部就是 O(logn)	na	O(n)	读写是 O(log n)，如果 键在列表中，就是 O(log n)；如果键不在 列表中，就是 O(n)	na	na

10.11 小结

本章介绍了如何处理不同类型的集合。数组的大小是固定的，但可以使用列表作为动态增长的集合。队列以先进先出的方式访问元素，栈以后进先出的方式访问元素。链表可以快速插入和删除元素，但搜索操作比较慢。通过键和值可以使用字典，它的搜索和插入操作比较快。集(其名称是 `hashSet<T>`)用于无序的唯一项。

本章还介绍了许多接口及其在集合访问和排序上的用法。探讨了一些特殊的集合，例如 `BitArray` 和 `BitVector32`，它们为处理位集合进行了优化。

第 11 章将详细介绍 LINQ，这是 C# 3.0 主要的新语言扩展。



# Language Integrated Query

Language Integrated Query(LINQ)是 C# 3.0 和 .NET 3.5 中最重要的新功能。LINQ 集成了 C# 编程语言中的查询语法，可以用相同的语法访问不同的数据源。LINQ 提供了不同数据源的抽象层，所以可以使用相同的语法。

本章介绍 LINQ 的核心功能和 C# 3.0 中支持新特性的语言扩展。本章的主要内容如下：

- 用 `List<T>` 在对象上执行传统查询
- 扩展方法
- $\lambda$  表达式
- LINQ 查询
- 标准查询操作符
- 表达式树
- LINQ 提供程序

提示：

本章介绍 LINQ 的核心功能。读完本章后，在数据库中使用 LINQ 的内容可查阅第 27 章，查询 XML 数据的内容可参见第 29 章。

## 11.1 LINQ 概述

在介绍 LINQ 的特性之前，本节先用一个例子来说明在 LINQ 推出之前如何查询对象。在此过程中，将说明查询如何演变为 LINQ 查询。了解了这些步骤，就知道 LINQ 查询的意义了。

本章的示例基于一级方程式世界冠军。查询将搜索一个 `Racer` 对象列表。第一个查询按照比赛的顺序得到巴西所有一级方程式世界冠军。

### 11.1.1 使用 `List<T>` 的查询

过滤和排序的第一个变体是在一个 `List<T>` 类型的列表中搜索数据。在搜索开始之前，必须确定对象类型和对象列表。

给对象定义类型 `Racer`。`Racer` 定义了几个属性和一个重载的 `ToString()` 方法，该方法以字符串格式显示赛手。这个类实现了接口 `IFormattable`，以支持格式字符串的不同变体，这个类还实现了接口 `IComparable<Racer>`，它根据 `Lastname` 为一组赛手排序。为了执行更高级的查询，

类 `Racer` 不仅包含单值属性，如 `Firstname`、`Lastname`、`Wins`、`Country` 和 `Starts`，还包含多值属性，如 `Cars` 和 `Years`。`Years` 属性列出了赛手获得冠军的年份。一些赛手曾多次获得冠军。`Cars` 属性用于列出赛手在获得冠军的年份中使用的所有车型。

```
using System;
using System.Text;

namespace Wrox.ProCSharp.LINQ
{
    [Serializable]
    public class Racer : IComparable<Racer>, IFormattable
    {
        public string FirstName {get; set;}
        public string LastName {get; set;}
        public int Wins {get; set;}
        public string Country {get; set;}
        public int Starts {get; set;}
        public string[] Cars { get; set; }
        public int[] Years { get; set; }

        public override string ToString()
        {
            return String.Format("{0} {1}", Firstname, Lastname);
        }

        public int CompareTo(Racer other)
        {
            return this.lastname.CompareTo(other.lastname);
        }

        public string ToString(string format)
        {
            return ToString(format, null);
        }

        public string ToString(string format, IFormatProvider formatProvider)
        {
            switch (format)
            {
                case null:
                case "N":
                    return ToString();
                case "F":
                    return Firstname;
                case "L":
                    return Lastname;
                case "C":
                    return Country;
                case "S":
                    return Starts.ToString();
                case "W":
                    return Wins.ToString();
                case "A":
                    return String.Format("{0} {1}, {2};" +
                        " starts: {3}, wins: {4}",
                        FirstName, LastName, Country,
                        Starts, Wins);
                default:
                    throw new FormatException(String.Format(
                        "Format {0} not supported", format));
            }
        }
    }
}
```

类 `Formula1` 在 `GetChampions()` 方法中返回一组赛手。这个列表包含了 1950 到 2007 年之间的所有一级方程式冠军。

```
using System;
using System.Collections.Generic;

namespace Wrox.ProCSharp.LINQ
{
    public static class Formula1
    {
        public static IList<Racer> GetChampions()
        {
            List<Racer> racers = new List<Racer>(40);
            racers.Add(new Racer() { FirstName = "Nino",
                LastName = "Farina", Country = "Italy",
                Starts = 33, Wins = 5,
                Years = new int[] { 1950 },
                Cars = new string[] { "Alfa Romeo" } });
            racers.Add(new Racer() {
                FirstName = "Alberto",
                LastName = "Ascari", Country = "Italy",
                Starts = 32, Wins = 10,
                Years = new int[] { 1952, 1953 },
                Cars = new string[] { "Ferrari" } });
            racers.Add(new Racer() {
                FirstName = "Juan Manuel",
                LastName = "Fangio",
                Country = "Argentina", Starts = 51,
                Wins = 24, Years = new int[]
                { 1951, 1954, 1955, 1956, 1957 },
                Cars = new string[] { "Alfa Romeo",
                    "Maserati", "Mercedes",
                    "Ferrari" } });
            racers.Add(new Racer() { FirstName = "Mike",
                LastName = "Hawthorn", Country = "UK",
                Starts = 45, Wins = 3,
                Years = new int[] { 1958 },
                Cars = new string[] { "Ferrari" } });
            racers.Add(new Racer() { FirstName = "Phil",
                LastName = "Hill", Country = "USA",
                Starts = 48, Wins = 3,
                Years = new int[] { 1961 },
                Cars = new string[] { "Ferrari" } });
            racers.Add(new Racer() { FirstName = "John",
                LastName = "Surtees", Country = "UK",
                Starts = 111, Wins = 6,
                Years = new int[] { 1964 },
                Cars = new string[] { "Ferrari" } });
            racers.Add(new Racer() { FirstName = "Jim",
                LastName = "Clark", Country = "UK",
                Starts = 72, Wins = 25,
                Years = new int[] { 1963, 1965 },
                Cars = new string[] { "Lotus" } });
            racers.Add(new Racer() { FirstName = "Jack",
                LastName = "Brabham",
```

```
Country = "Australia", Starts = 125,
Wins = 14,
Years = new int[] { 1959, 1960, 1966 },
Cars = new string[] { "Cooper",
    "Brabham" } });
racers.Add(new Racer() { FirstName = "Denny",
    LastName = "Hulme",
    Country = "New Zealand", Starts = 112,
    Wins = 8,
    Years = new int[] { 1967 },
    Cars = new string[] { "Brabham" } });
racers.Add(new Racer() { FirstName = "Graham",
    LastName = "Hill", Country = "UK",
    Starts = 176, Wins = 14,
    Years = new int[] { 1962, 1968 },
    Cars = new string[] { "BRM", "Lotus" }
    });
racers.Add(new Racer() { FirstName = "Jochen",
    LastName = "Rindt", Country = "Austria",
    Starts = 60, Wins = 6,
    Years = new int[] { 1970 },
    Cars = new string[] { "Lotus" } });
racers.Add(new Racer() { FirstName = "Jackie",
    LastName = "Stewart", Country = "UK",
    Starts = 99, Wins = 27,
    Years = new int[] { 1969, 1971, 1973 },
    Cars = new string[] { "Matra",
    "Tyrrell" } });
racers.Add(new Racer() {
    FirstName = "Emerson",
    LastName = "Fittipaldi",
    Country = "Brazil", Starts = 143,
    Wins = 14, Years = new int[] { 1972,
    1974 },
    Cars = new string[] { "Lotus",
    "McLaren" } });
racers.Add(new Racer() { FirstName = "James",
    LastName = "Hunt", Country = "UK",
    Starts = 91, Wins = 10,
    Years = new int[] { 1976 },
    Cars = new string[] { "McLaren" } });
racers.Add(new Racer() { FirstName = "Mario",
    LastName = "Andretti", Country = "USA",
    Starts = 128, Wins = 12,
    Years = new int[] { 1978 },
    Cars = new string[] { "Lotus" } });
racers.Add(new Racer() { FirstName = "Jody",
    LastName = "Scheckter",
    Country = "South Africa", Starts = 112,
    Wins = 10,
    Years = new int[] { 1979 },
    Cars = new string[] { "Ferrari" } });
racers.Add(new Racer() { FirstName = "Alan",
    LastName = "Jones",
    Country = "Australia", Starts = 115,
    Wins = 12,
    Years = new int[] { 1980 },
    Cars = new string[] { "Williams" } });
racers.Add(new Racer() { FirstName = "Keke",
    LastName = "Rosberg",
```

```

        Country = "Finland", Starts = 114,
        Wins = 5,
        Years = new int[] { 1982 },
        Cars = new string[] { "Williams" } });
racers.Add(new Racer() { FirstName = "Niki",
    LastName = "Lauda", Country = "Austria",
    Starts = 173, Wins = 25,
    Years = new int[] { 1975, 1977, 1984 },
    Cars = new string[] { "Ferrari",
        "McLaren" } });
racers.Add(new Racer() { FirstName = "Nelson",
    LastName = "Piquet", Country = "Brazil",
    Starts = 204, Wins = 23,
    Years = new int[] { 1981, 1983, 1987 },
    Cars = new string[] { "Brabham",
        "Williams" } });
racers.Add(new Racer() { FirstName = "Ayrton",
    LastName = "Senna", Country = "Brazil",
    Starts = 161, Wins = 41,
    Years = new int[] { 1988, 1990, 1991 },
    Cars = new string[] { "McLaren" } });
racers.Add(new Racer() { FirstName = "Nigel",
    LastName = "Mansell", Country = "UK",
    Starts = 187, Wins = 31,
    Years = new int[] { 1992 },
    Cars = new string[] { "Williams" } });
racers.Add(new Racer() { FirstName = "Alain",
    LastName = "Prost", Country = "France",
    Starts = 197, Wins = 51,
    Years = new int[] { 1985, 1986, 1989,
        1993 },
    Cars = new string[] { "McLaren",
        "Williams" } });
racers.Add(new Racer() { FirstName = "Damon",
    LastName = "Hill", Country = "UK",
    Starts = 114, Wins = 22,
    Years = new int[] { 1996 },
    Cars = new string[] { "Williams" } });
racers.Add(new Racer() {
    FirstName = "Jacques",
    LastName = "Villeneuve",
    Country = "Canada", Starts = 165,
    Wins = 11, Years = new int[] { 1997 },
    Cars = new string[] { "Williams" } });
racers.Add(new Racer() { FirstName = "Mika",
    LastName = "Hakkinen",
    Country = "Finland", Starts = 160,
    Wins = 20, Years = new int[] { 1998,
        1999 },
    Cars = new string[] { "McLaren" } });
racers.Add(new Racer() {
    FirstName = "Michael",
    LastName = "Schumacher",
    Country = "Germany", Starts = 250,
    Wins = 91,
    Years = new int[] { 1994, 1995, 2000,
        2001, 2002, 2003, 2004 },
    Cars = new string[] { "Benetton",
        "Ferrari" } });
racers.Add(new Racer() {

```



```

        FirstName = "Fernando",
        LastName = "Alonso", Country = "Spain",
        Starts = 105, Wins = 19,
        Years = new int[] { 2005, 2006 },
        Cars = new string[] { "Renault" } });
    racers.Add(new Racer() { FirstName = "Kimi",
        LastName = "Rikk nen",
        Country = "Finland", Starts = 122,
        Wins = 15, Years = new int[] { 2007 },
        Cars = new string[] { "Ferrari" } });
    return racers;
}
}
}

```

对于后面在多个列表中执行的查询，`GetConstructorChampions()`方法返回所有的制造商冠军。制造商冠军是从 1958 年开始设立的。

```

public static IList < Team >
    GetConstructorChampions()
{
    List < Team > teams = new List < Team > (20);
    teams.Add(new Team() { Name = "Vanwall",
        Years = new int[] { 1958 } });
    teams.Add(new Team() { Name = "Cooper",
        Years = new int[] { 1959, 1960 } });
    teams.Add(new Team() { Name = "Ferrari",
        Years = new int[] { 1961, 1964, 1975,
            1976, 1977, 1979, 1982, 1983, 1999,
            2000, 2001, 2002, 2003, 2004, 2007 } });
    teams.Add(new Team() { Name = "BRM",
        Years = new int[] { 1962 } });
    teams.Add(new Team() { Name = "Lotus",
        Years = new int[] { 1963, 1965, 1968,
            1970, 1972, 1973, 1978 } });
    teams.Add(new Team() { Name = "Brabham",
        Years = new int[] { 1966, 1967 } });
    teams.Add(new Team() { Name = "Matra",
        Years = new int[] { 1969 } });
    teams.Add(new Team() { Name = "Tyrrell",
        Years = new int[] { 1971 } });
    teams.Add(new Team() { Name = "McLaren",
        Years = new int[] { 1974, 1984, 1985,
            1988, 1989, 1990, 1991, 1998 } });
    teams.Add(new Team() { Name = "Williams",
        Years = new int[] { 1980, 1981, 1986,
            1987, 1992, 1993, 1994, 1996, 1997 } });
    teams.Add(new Team() { Name = "Benetton",
        Years = new int[] { 1995 } });
    teams.Add(new Team() { Name = "Renault",
        Years = new int[] { 2005, 2006 } });
    return teams;
}

```

现在进入对象查询的核心。首先，需要用 `GetChampions()` 静态方法获得对象列表。该列表放在泛型类 `List<T>` 中。这个类的 `FindAll()` 方法接收一个 `Predicate<T>` 委托，该委托可以实现为一个匿名方法。只返回 `Country` 属性设置为 `Brazil` 的赛手。接着，用 `Sort()` 方法给得到的列表排序。不应按照 `Lastname` 属性排序，因为这是 `Racer` 类的默认排序方式，而可以传送一个类型为 `Comparison<T>` 的委托，该委托也实现为一个匿名方法，来比较夺冠次数。使用 `r2` 对象，与 `r1` 比较，根据需要进行降序排序。`foreach` 语句最终迭代已排序的集合中的所有 `Racer` 对象。

```
private static void ObjectQuery()
{
    List<Racer> racers = new List<Racer>(Formula1.GetChampions());
    List<Racer> brazilRacers = racers.FindAll(
        delegate(Racer r)
        {
            return r.Country == "Brazil";
        });
    brazilRacers.Sort(
        delegate(Racer r1, Racer r2)
        {
            return r2.Wins.CompareTo(r1.Wins);
        });
    foreach (Racer r in brazilRacers)
    {
        Console.WriteLine("{0:A}", r);
    }
}
```

下面的列表显示了来自巴西的所有冠军，并按照夺冠次数排序：

```
Ayrton Senna, Brazil; starts: 161, wins: 41
Nelson Piquet, Brazil; starts: 204, wins: 23
Emerson Fittipaldi, Brazil; starts: 143, wins: 14
```

**提示：**

排序和过滤对象列表的内容详见第 10 章。

在前面的例子中，使用了 `List<T>` 类的 `FindAll()` 和 `Sort()` 方法。使用任何集合都可以获得这两个方法的功能，而不仅仅是 `List<T>`。这需要使用扩展方法。扩展方法是 C# 3.0 的新增特性，这也是上述例子迈向 LINQ 的第一个变化。

### 11.1.2 扩展方法

扩展方法可以将方法写入最初没有提供该方法的类中。还可以把方法添加到实现某个接口的任何类中，这样多个类就可以使用相同的实现代码。

例如，`String` 类没有 `Foo()` 方法。`String` 类是密封的，所以不能从这个类中继承。但可以执行一个扩展方法，如下所示：

```
public static class StringExtension
{
    public static void Foo(this string s)
    {
        Console.WriteLine("Foo invoked for {0}", s);
    }
}
```

```
}
}
```

扩展方法在静态类中声明，定义为一个静态方法，其中第一个参数定义了它扩展的类型。`Foo()`方法扩展了 `String` 类，因为它的第一个参数定义了 `String` 类型。为了区分扩展方法和一般的静态方法，扩展方法还需要给第一个参数使用 `this` 关键字。

现在就可以使用带 `string` 类型的 `Foo()`方法了：

```
string s = "Hello";
s.Foo();
```

结果在控制台上显示 `Foo invoked for Hello`，因为 `Hello` 是传送给 `Foo()`方法的字符串。

也许这看起来违反了面向对象的规则，因为给一个类型定义了新方法，但没有改变该类型。但实际上并非如此。扩展方法不能访问它扩展的类型的私有成员。调用扩展方法只是调用静态方法的一种新语法。对于字符串，可以用如下方式调用 `Foo()`方法，获得相同的结果：

```
string s = "Hello";
StringExtension.Foo(s);
```

要调用静态方法，应在类名的后面加上方法名。扩展方法是调用静态方法的另一种方式。不必提供定义了静态方法的类名，相反，调用静态方法是因为它带的参数类型。只需导入包含该类的命名空间，就可以将 `Foo()`扩展方法放在 `String` 类的作用域中。

定义 LINQ 扩展方法的一个类是 `System.Linq` 命名空间中的 `Enumerable`。只需导入这个命名空间，就打开了这个类的扩展方法的作用域。下面列出了 `Where()`扩展方法的实现代码。`Where()`扩展方法的第一个参数包含了 `this` 关键字，其类型是 `IEnumerable<T>`。这样，`Where`方法就可以用于实现了 `IEnumerable<T>` 的每个类型。例如数组和 `List<T>` 实现了 `IEnumerable<T>`。第二个参数是一个 `Func<T,bool>`委托，它引用了一个返回布尔值、参数类型为 `T` 的方法。这个谓词在实现代码中调用，检查 `IEnumerable<T>`源中的项是否应放在目标集合中。如果委托引用了该方法，`yield return` 语句就将源中的项返回给目标。

```
public static IEnumerable<T> Where<T>(this IEnumerable<T> source,
                                     Func<T, bool> predicate)
{
    foreach (T item in source)
        if (predicate(item))
            yield return item;
}
```

因为 `Where()`实现为一个泛型方法，所以可以用于包含在集合中的任意类型。实现了 `IEnumerable<T>`的集合都支持它。

提示：

这里的扩展方法在程序集 `System.Core` 的 `System.Linq` 命名空间中定义。

现在就可以使用 `Enumerable` 类中的扩展方法 `Where()`、`OrderByDescending()`和 `Select()`了。这些方法都返回 `IEnumerable<TSource>`，所以可以使用前面的结果依次调用这些方法。通过扩展方法的参数，使用定义了委托参数的实现代码的匿名方法。

```
private static void ExtensionMethods()
```

```

{
    List < Racer > champions =
        new List < Racer > (
            Formula1.GetChampions());
    IEnumerable < Racer > brazilChampions =
        champions.Where(
            delegate(Racer r)
            {
                return r.Country == "Brazil";
            }).OrderByDescending(
                delegate(Racer r)
                {
                    return r.Wins;
                }).Select(
                    delegate(Racer r)
                    {
                        return r;
                    });

    foreach (Racer r in brazilChampions)
    {
        Console.WriteLine("{0:A}", r);
    }
}

```

### 11.1.3 $\lambda$ 表达式

C# 3.0 给匿名方法提供了一个新的语法—— $\lambda$  表达式。除了把匿名方法传送给 `Where()`、`OrderByDescending()` 和 `Select()` 方法之外，还可以使用  $\lambda$  表达式。

这里把上面的例子改为使用  $\lambda$  表达式。现在语法比较短，也更容易理解了，因为删除了 `return` 语句、参数类型和花括号。

$\lambda$  表达式参见第 7 章。 $\lambda$  表达式在 LINQ 中非常重要，所以下面复习一下该语法。详细信息可参见第 7 章。

比较  $\lambda$  表达式和匿名委托，会发现许多类似之处。 $\lambda$  运算符 `=>` 的左边是参数，不需要添加参数类型，因为它们是由编译器解析的。 $\lambda$  运算符的右边定义了执行代码。在匿名方法中，需要花括号和 `return` 语句。在  $\lambda$  表达式中，不需要这些语法元素，因为它们是由编译器处理的。如果  $\lambda$  运算符右边有多个语句，也可以使用花括号和 `return` 语句。

```

private static void LambdaExpressions()
{
    IEnumerable < Racer > brazilChampions =
        Formula1.GetChampions().
        Where(r => r.Country == "Brazil").
        OrderByDescending(r => r.Wins).
        Select(r => r);

    foreach (Racer r in brazilChampions)
    {
        Console.WriteLine("{0:A}", r);
    }
}

```

提示:

使用无参的  $\lambda$  表达式时, `return` 语句和花括号是可选的。但在  $\lambda$  表达式中仍可以使用这些语言结构。详见第 7 章。

#### 11.1.4 LINQ 查询

最后一个需要修改的是用新的 LINQ 查询记号定义查询。语句 `from r in Formula1.GetChampions() Where r.Country == "Brazil" orderby r.Wins descending select r;` 是一个 LINQ 查询, 子句 `from`、`where`、`orderby`、`descending` 和 `select` 都是这个查询中的预定义关键字。编译器把这些子句映射到扩展方法 `Where()`、`OrderByDescending()` 和 `Select()`。  $\lambda$  表达式传送给参数。

`Where r.Country == "Brazil"` is converted to `Where(r => r.Country == "Brazil")`  
`.orderby r.Wins descending` is converted to `OrderByDescending(r => r.Wins)`.

```
private static void LinqQuery()
{
    var query = from r in Formula1.GetChampions()
                where r.Country == "Brazil"
                orderby r.Wins descending
                select r;

    foreach (Racer r in query)
    {
        Console.WriteLine("{0:A}", r);
    }
}
```

提示:

LINQ 查询是 C# 语言中的一个简化查询记号。编译器编译查询表达式, 调用扩展方法。查询表达式只是 C# 中的一个语法, 但不需要修改底层的 IL 代码。

查询表达式必须以 `from` 子句开头, 以 `select` 或 `group` 子句结束。在这两个子句之间, 可以使用 `where`、`orderby`、`join`、`let` 和其他 `from` 子句。

注意, 变量 `query` 只指定了 LINQ 查询。该查询不是通过这个赋值语句执行的, 只要使用 `foreach` 循环访问查询, 该查询就会执行。详见后面的内容。

在前面的示例中, 学习了 C# 3.0 语言特性, 以及它们与 LINQ 查询的关系。下面深入探讨 LINQ 的特性。

#### 11.1.5 推迟查询的执行

在运行期间定义查询表达式时, 查询就不会运行。查询会在迭代数据项时运行。

再看看扩展方法 `Where()`。它使用 `yield return` 语句返回谓词为 `true` 的元素。因为使用了 `yield return` 语句, 所以编译器会创建一个枚举器, 在访问枚举中的项后, 就返回它们。

```
public static IEnumerable<T> Where<T>(this IEnumerable<T> source,
                                     Func<T, bool> predicate)
{
```



```

foreach (T item in source)
    if (predicate(item))
        yield return item;
}

```

这是一个非常有趣、也非常重要的结果。在下面的例子中，创建了一个 `String` 元素集合，用名称 `arr` 填充它。接着定义一个查询，从集合中找出以字母 `J` 开头的名称。集合也应是排好序的。在定义查询时，不会执行迭代。相反，迭代在 `foreach` 语句中进行，迭代所有的项。集合中只有一个元素 `Juan` 满足 `where` 表达式的要求，即以字母 `J` 开头。迭代完成后，将 `Juan` 写入控制台。之后在集合中添加四个新名称，再次执行迭代。

```

List<string> names = new List<string>
    { "Nino", "Alberto", "Juan", "Mike", "Phil" };

var namesWithJ = from n in names
    where n.StartsWith("J")
    orderby n
    select n;

Console.WriteLine("First iteration");
foreach (string name in namesWithJ)
{
    Console.WriteLine(name);
}
Console.WriteLine();

arr.Add("John");
arr.Add("Jim");
arr.Add("Jack");
arr.Add("Denny");

Console.WriteLine("Second iteration");
foreach (string name in namesWithJ)
{
    Console.WriteLine(name);
}

```

因为迭代在查询定义时不会执行，而是在执行每个 `foreach` 语句时执行，所以可以看到其中的变化，如应用程序的结果所示：

```

First iteration
Juan

Second iteration
Jack
Jim
John
Juan

```

当然，还必须注意，每次在迭代中使用查询时，都会调用扩展方法。在大多数情况下，这是非常有效的，因为我们可以检测出源数据中的变化。但是在一些情况下，这是不可行的。调用扩展方法 `ToArray()`、`ToEnumerable()`、`ToList()` 等可以改变这个操作。在示例中，`ToList` 迭代集合，返回一个实现了 `ICollection<string>` 的集合。之后对返回的列表迭代两次，在这个过程中，数据源得到了新名称。

```

List<string> names = new List<string>
    { "Nino", "Alberto", "Juan", "Mike", "Phil" };

```

```
        IList<string> namesWithJ = (from n in arr
                                   where n.StartsWith("J")
                                   orderby n
                                   select n).ToList();

        Console.WriteLine("First iteration");
        foreach (string name in namesWithJ)
        {
            Console.WriteLine(name);
        }
        Console.WriteLine();

        arr.Add("John");
        arr.Add("Jim");
        arr.Add("Jack");
        arr.Add("Denny");

        Console.WriteLine("Second iteration");
        foreach (string name in namesWithJ)
        {
            Console.WriteLine(name);
        }
```

在结果中可以看到，两次迭代中输出保持不变，但集合中的值改变了：

```
First iteration
Juan

Second iteration
Juan
```

## 11.2 标准的查询操作符

Where、OrderByDescending 和 Select 只是 LINQ 的几个查询操作符。LINQ 查询为最常用的操作符定义了一个声明语法。还有许多标准查询操作符。

表 11-1 列出了 LINQ 定义的标准查询操作符。

表 11-1

标准查询操作符	说 明
Where OfType<TResult>	过滤操作符定义了返回元素的条件。在 Where 查询操作符中，可以使用谓词，例如 lambda 表达式定义的谓词，来返回布尔值。OfType<TResult>根据类型过滤元素，只返回 TResult 类型的元素
Select 和 SelectMany	投射操作符用于把对象转换为另一个类型的对象。Select 和 SelectMany 定义了根据选择器函数选择结果值的投射
OrderBy, ThenBy OrderByDescending ThenByDescending Reverse	排序操作符改变所返回的元素的顺序。OrderBy 按升序排序，OrderByDescending 按降序排序。如果第一次排序的结果很类似，就可以使用 ThenBy 和 ThenBy Descending 操作符进行第二次排序。Reverse 反转集合中元素的顺序

(续表)

标准查询操作符	说 明
Join, GroupJoin	连接运算符用于合并不直接相关的集合。使用 Join 操作符，可以根据键选择器函数连接两个集合，这类似于 SQL 中的 JOIN。GroupJoin 操作符连接两个集合，组合其结果
GroupBy	组合运算符把数据放在组中。GroupBy 操作符组合有公共键的元素
Any, All, Contains	如果元素序列满足指定的条件，量词操作符就返回布尔值。Any, All 和 Contains 都是量词操作符。Any 确定集合中是否有满足谓词函数的元素；All 确定集合中的所有元素是否都满足谓词函数；Contains 检查某个元素是否在集合中。这些操作符都返回一个布尔值
Take, Skip, TakeWhile, SkipWhile	分区操作符返回集合的一个子集。Take、Skip、TakeWhile 和 SkipWhile 都是分区操作符。使用它们可以得到部分结果。使用 Take 必须指定要从集合中提取的元素个数；Skip 跳过指定的元素个数，提取其他元素，TakeWhile 提取条件为真的元素
Distinct, Union, Intersect, Except	Set 操作符返回一个集合。Distinct 从集合中删除重复的元素。除了 Distinct 之外，其他 Set 操作符都需要两个集合。Union 返回出现在其中一个集合中的元素。Intersect 返回两个集合中都有的元素。Except 返回只出现在一个集合中的元素
First, FirstOrDefault, Last, LastOrDefault, ElementAt, ElementAtOrDefault, Single, SingleOrDefault	这些元素操作符仅返回一个元素。First 返回第一个满足条件的元素。FirstOrDefault 类似于 First，但如果没有找到满足条件的元素，就返回类型的默认值。Last 返回最后一个满足条件的元素。ElementAt 指定了要返回的元素的位置。Single 只返回一个满足条件的元素。如果有多个元素都满足条件，就抛出一个异常
Count, Sum, Min, Max, Average, Aggregate	合计操作符计算集合的一个值。利用这些合计操作符，可以计算所有值的总和、元素的个数、值最大和最小的元素，平均值等
ToArray, ToEnumerable, ToList, ToDictionary, toType<T>	这些转换操作符将集合转换为数组、IEnumerable、IList、IDictionary 等
Empty, Range, Repeat	这些生成操作符返回一个新集合。使用 Empty，集合是空的，Range 返回一系列数字，Repeat 返回一个始终重复一个值的集合

下面是使用这些操作符的一些例子。

### 11.2.1 过滤

下面介绍一些查询的示例。

使用 `Where` 子句，可以合并多个表达式。例如，找出赢得至少 15 场比赛的巴西和奥地利车手。传送给 `where` 子句的表达式的结果类型应是 `bool`：

```
var racers = from r in Formula1.GetChampions()
              where r.Wins > 15 & &
                  (r.Country == "Brazil" ||
                   r.Country == "Austria")
              select r;
foreach (var r in racers)
{
    Console.WriteLine("{0:A}", r);
}
```

用这个 LINQ 查询启动程序，会返回 Niki Lauda、Nelson Piquet 和 Ayrton Senna，如下：

```
Niki Lauda, Austria, Starts: 173, Wins: 25
Nelson Piquet, Brazil, Starts: 204, Wins: 23
Ayrton Senna, Brazil, Starts: 161, Wins: 41
```

并不是所有的查询都可以用 LINQ 查询完成。也不是所有的扩展方法都映射到 LINQ 查询子句上。高级查询需要使用扩展方法。为了更好地理解带扩展方法的复杂查询，最好看看简单的查询是如何映射的。使用扩展方法 `Where()` 和 `Select()`，会生成与前面 LINQ 查询非常类似的结果：

```
var racers = Formula1.GetChampions().
    Where(r => r.Wins > 15 & &
            (r.Country == "Brazil" ||
             r.Country == "Austria")).
    Select(r => r);
```

### 11.2.2 用索引来过滤

不能使用 LINQ 查询的一个例子是 `Where()` 方法的重载。在 `Where()` 方法的重载中，可以传递第二个参数——索引。索引是过滤器返回的每个结果的计数器。可以在表达式中使用这个索引，执行基于索引的计算。下面的代码由 `Where()` 扩展方法调用，它使用索引返回姓氏以 A 开头、索引为偶数的车手：

```
var racers = Formula1.GetChampions().
    Where((r, index) =>
        r.LastName.StartsWith("A") & &
        index % 2 != 0);
foreach (var r in racers)
{
    Console.WriteLine("{0:A}", r);
}
```

姓氏以 A 开头的车手有 Alberto Ascari、Mario Andretti 和 Fernando Alonso。Mario Andretti 的索引是奇数，所以他不在结果中：

```
Alberto Ascari, Italy; starts: 32, wins: 10
```

```
Fernando Alonso, Spain; starts: 105, wins: 19
```

### 11.2.3 类型过滤

为了进行基于类型的过滤，可以使用 `OfType()` 扩展方法。这里数组数据包含 `string` 和 `int` 对象。使用 `OfType()` 扩展方法，把 `string` 类传送给泛型参数，就从集合中返回字符串：

```
object[] data = { "one", 2, 3, "four", "five", 6 };
var query = data.OfType < string > ();
foreach (var s in query)
{
    Console.WriteLine(s);
}
```

运行这段代码，就会显示字符串 `one`、`four` 和 `five`。

```
one
four
five
```

### 11.2.4 复合的 from 子句

如果需要根据对象的一个成员进行过滤，而该成员本身是一个系列，就可以使用复合的 `from` 子句。`Racer` 类定义了一个属性 `Cars`，`Cars` 是一个字符串数组。要过滤驾驶 `Ferrari` 的所有冠军，可以使用如下所示的 LINQ 查询。第一个 `from` 子句访问从 `Formylal.GetChampions()` 返回的 `Racer` 对象，第二个 `from` 子句访问 `Racer` 类的属性 `Cars`，返回所有 `string` 类型的赛车。接着在 `Where` 子句中使用这些赛车过滤驾驶 `Ferrari` 的所有冠军。

```
var ferrariDrivers = from r in
    Formula1.GetChampions()
    from c in r.Cars
    where c == "Ferrari"
    orderby r.LastName
    select r.FirstName + " "
    + r.LastName;
```

这个查询的结果显示了驾驶 `Ferrari` 的所有一级方程式冠军：

```
Alberto Ascari
Juan Manuel Fangio
Mike Hawthorn
Phil Hill
Niki Lauda
Jody Scheckter
Michael Schumacher
John Surtees
```

C#编译器把复合的 `from` 子句和 LINQ 查询转换为 `SelectMany()` 扩展方法。`SelectMany()` 可用于迭代序列的序列。示例中 `SelectMany()` 方法的重载版本如下所示：

```
public static IEnumerable < TResult >
    SelectMany < TSource, TCollection, TResult > (
```



```

this IEnumerable < TSource > source,
Func < TSource,
IEnumerable < TCollection > > collectionSelector,
Func < TSource, TCollection, TResult >
resultSelector);

```

第一个参数是隐式参数，从 `GetChampions()` 方法中接收 `Racer` 对象序列。第二个参数是 `collectionSelector` 委托，它定义了内部序列。在  $\lambda$  表达式 `r=>r.Cars` 中，应返回赛车集合。第三个参数是一个委托，现在为每个赛车调用该委托，接收 `Racer` 和 `Car` 对象。 $\lambda$  表达式创建了一个匿名类型，它带 `Racer` 和 `Car` 属性。这个 `SelectMany()` 方法的结果是摊平了赛手和赛车的层次结构，为每辆赛车返回匿名类型的一个新对象集合。

这个新集合传送给 `Where()` 方法，过滤出驾驶 `Ferrari` 的赛手。最后，调用 `OrderBy()` 和 `Select()` 方法：

```

var ferrariDrivers = Formula1.GetChampions().
    SelectMany(
        r => r.Cars,
        (r, c) => new { Racer = r, Car = c }).
    Where(r => r.Car == "Ferrari").
    OrderBy(r => r.Racer.LastName).
    Select(r => r.Racer.FirstName + " " +
        r.Racer.LastName);

```

把 `SelectMany()` 泛型方法解析为这里使用的类型，所解析的类型如下所示。在这个例子中，数据源是 `Racer` 类型，所过滤的集合是一个 `string` 数组，当然所返回的匿名类型的名称是未知的，这里显示为 `TResult`：

```

public static IEnumerable < TResult >
    SelectMany < Racer, string, TResult > (
    this IEnumerable < Racer > source,
    Func < Racer, IEnumerable < string > > collectionSelector,
    Func < Racer, string, TResult > resultSelector);

```

查询仅从 LINQ 查询转换为扩展方法，所以结果与前面的相同。

### 11.2.5 排序

要对序列排序，前面使用了 `orderby` 子句。下面复习一下前面使用 `orderby descending` 子句的例子。其中赛手按照赢得比赛的次数进行降序排序，赢得比赛的次数是用关键字选择器指定的：

```

var racers = from r in Formula1.GetChampions()
    where r.Country == "Brazil"
    orderby r.Wins descending
    select r;

```

`orderby` 子句解析为 `OrderBy()` 方法，`orderby descending` 子句解析为 `OrderByDescending()` 方法：

```

var racers = Formula1.GetChampions().
    Where(r => r.Country == "Brazil").

```

```
OrderByDescending(r => r.Wins).
Select(r => r);
```

`OrderBy()`和 `OrderByDescending()`方法返回 `IOrderEnumerable<TSource>`。这个接口派生于接口 `IEnumerable<TSource>`，但包含一个额外的方法 `CreateOrderedEnumerable<TSource>()`。这个方法用于进一步给序列排序。如果根据关键字选择器来排序，两项的顺序相同，就可以使用 `ThenBy()`和 `ThenByDescending()`方法继续排序。这两个方法需要 `IOrderEnumerable<TSource>`才能工作，但也返回这个接口。所以，可以添加任意多个 `ThenBy()`和 `ThenByDescending()`方法，对集合排序。

使用 LINQ 查询时，只需把所有用于排序的不同关键字(用逗号分隔开)添加到 `orderby` 子句中。这里，所有的赛车手先按照国家排序，再按照姓氏排序，最后按照名字排序。添加到 LINQ 查询结果中的 `Take()`扩展方法用于提取前 10 个结果：

```
var racers = (from r in
    Formula1.GetChampions()
    orderby r.Country, r.LastName,
    r.FirstName
    select r).Take(10);
```

排序后的结果如下：

```
Argentina: Fangio, Juan Manuel
Australia: Brabham, Jack
Australia: Jones, Alan
Austria: Lauda, Niki
Austria: Rindt, Jochen
Brazil: Fittipaldi, Emerson
Brazil: Piquet, Nelson
Brazil: Senna, Ayrton
Canada: Villeneuve, Jacques
Finland: Hakkinen, Mika
```

使用 `OrderBy()`和 `ThenBy()`方法可以执行相同的操作：

```
var racers = Formula1.GetChampions().
    OrderBy(r => r.Country).
    ThenBy(r => r.LastName).
    ThenBy(r => r.FirstName).
    Take(10);
```

### 11.2.6 分组

要根据一个关键字值对查询结果分组，可以使用 `group` 子句。现在一级方程式冠军应按照国家分组，并列出一个国家的冠军数。子句 `group r by r.Country into g` 根据 `Country` 属性组合所有的赛车手，并定义一个新的标识符 `g`，它以后用于访问分组的结果信息。`group` 子句的结果根据应用到分组结果上的扩展方法 `Count()`来排序，如果冠军数相同，就根据关键字来排序，该关键字是国家，因为这是分组所使用的关键字。`where` 子句根据至少有两项的分组来过滤，`select` 子句创建一个带 `Country` 和 `Count` 属性的匿名类型。

```
var countries = from r in
    Formula1.GetChampions()
```

```

        group r by r.Country into g
        orderby g.Count() descending, g.Key
        where g.Count() >= 2
        select new { Country = g.Key,
            Count = g.Count() };

foreach (var item in countries)
{
    Console.WriteLine("{0, -10} {1}",
        item.Country, item.Count);
}

```

结果显示了带 Country 和 Count 属性的对象集合：

UK	9
Brazil	3
Australia	2
Austria	2
Finland	2
Italy	2
USA	2

要用扩展方法执行相同的操作，应把 `groupby` 子句解析为 `GroupBy()` 方法。在 `GroupBy()` 方法的声明中，注意它返回实现了 `IGrouping` 接口的对象枚举。`IGrouping` 接口定义了 `Key` 属性，所以在定义了对这个方法的调用后，可以访问分组的关键字：

```

public static IEnumerable < IGrouping < TKey, TSource > >
    GroupBy < TSource, TKey > (
        this IEnumerable < TSource > source,
        Func < TSource, TKey > keySelector);

```

子句 `group r by r.Country into g` 解析为 `GroupBy(r => r.Country)`，返回分组系列。分组系列首先用 `OrderByDescending()` 方法排序，再用 `ThenBy()` 方法排序。接着调用 `Where()` 和 `Select()` 方法。

```

var countries = Formula1.GetChampions().
    GroupBy(r => r.Country).
    OrderByDescending(g => g.Count()).
    ThenBy(g => g.Key).
    Where(g => g.Count() >= 2).
    Select(g => new { Country = g.Key,
        Count = g.Count() });

```

### 11.2.7 对嵌套的对象分组

如果分组的对象应包含嵌套的对象，就可以改变 `select` 子句创建的匿名类型。在下面的例子中，所创建的国家不仅应包含国家名和赛手数量这两个属性，还应包含赛手名序列。这个序列用一个赋予 `Racers` 属性的 `from/in` 内部子句指定，内部的 `from` 子句使用分组标识符 `g` 获得该分组中的所有赛手，用姓氏对它们排序，再根据姓名创建一个新字符串：

```

var countries = from r in
    Formula1.GetChampions()
    group r by r.Country into g
    orderby g.Count() descending, g.Key
    where g.Count() >= 2

```

```

select new
{
    Country = g.Key,
    Count = g.Count(),
    Racers = from r1 in g
              orderby r1.LastName
              select r1.FirstName + " "
                + r1.LastName
};
foreach (var item in countries)
{
    Console.WriteLine("{0, -10} {1}",
        item.Country, item.Count);
    foreach (var name in item.Racers)
    {
        Console.Write("{0}; ", name);
    }
    Console.WriteLine();
}

```

结果应列出某个国家的所有冠军：

```

UK 9
Jim Clark; Lewis Hamilton; Mike Hawthorn; Graham Hill; Damon Hill;
James Hunt; Nigel Mansell; Jackie Stewart; John Surtees;
Brazil 3
Emerson Fittipaldi; Nelson Piquet; Ayrton Senna;
Australia 2
Jack Brabham; Alan Jones;
Austria 2
Niki Lauda; Jochen Rindt;
Finland 2
Mika Hakkinen; Keke Rosberg;
Italy 2
Alberto Ascari; Nino Farina;
USA 2
Mario Andretti; Phil Hill;

```

### 11.2.8 连接

使用 `join` 子句可以根据特定的条件合并两个数据源，但之前要获得两个要连接的列表。在一级方程式比赛中，设有赛手冠军和制造商冠军。赛手从 `GetChampions()` 方法中返回，制造商从 `GetConstructorChampions()` 方法中返回。现在要获得一个年份列表，列出每年的赛手和制造商冠军。

为此，先定义两个查询，用于查询赛手和制造商团队：

```

var racers = from r in Formula1.GetChampions()
              from y in r.Years
              where y > 2003
              select new
              {
                  Year = y,
                  Name = r.FirstName + " " +
                      r.LastName
              }

```

```
};

var teams = from t in
    Formula1.GetConstructorChampions()
    from y in t.Years
    where y > 2003
    select new { Year = y,
        Name = t.Name };

```

有了这两个查询，再通过子句 `join t in teams on r.Year equals t.Year`，根据赛手获得冠军的年份和制造商获得冠军的年份进行连接。`select` 子句定义了一个新的匿名类型，它包含 `Year`、`Racer` 和 `Team` 属性。

```
var racersAndTeams =
    from r in racers
    join t in teams on r.Year equals t.Year
    select new
    {
        Year = r.Year,
        Racer = r.Name,
        Team = t.Name
    };
Console.WriteLine("Year Champion " +
    "Constructor Title");
foreach (var item in racersAndTeams)
{
    Console.WriteLine("{0}: {1,-20} {2}",
        item.Year, item.Racer, item.Team);
}

```

当然，也可以把它们合并为一个 LINQ 查询，但这只是一种尝试：

```
int year = 2003;
var racersAndTeams =
    from r in
        from r1 in Formula1.GetChampions()
        from yr in r1.Years
        where yr > year
        select new
        {
            Year = yr,
            Name = r1.FirstName + " " +
                r1.LastName
        }
    join t in
        from t1 in
            Formula1.GetConstructorChampions()
            from yt in t1.Years
            where yt > year
            select new { Year = yt,
                Name = t1.Name }
    on r.Year equals t.Year
    select new
    {
        Year = r.Year,
        Racer = r.Name,
        Team = t.Name
    };

```



结果显示了匿名类型中的数据:

Year	Champion	Constructor Title
2004	Michael Schumacher	Ferrari
2005	Fernando Alonso	Renault
2006	Fernando Alonso	Renault
2007	Kimi Räikkönen	Ferrari

### 11.2.9 设置操作

扩展方法 `Distinct()`、`Union()`、`Intersect()`和 `Except()`都是设置操作。下面创建一个驾驶 Ferrari 的一级方程式冠军序列和驾驶 McLaren 的一级方程式冠军序列, 然后确定是否有驾驶 Ferrari 和 McLaren 的冠军。当然, 这里可以使用 `Intersect()`扩展方法。

首先获得所有驾驶 Ferrari 的冠军。这只是一个简单的 LINQ 查询, 其中使用复合的 `from` 子句访问属性 `Cars`, 返回一个字符串对象序列。

```
var ferrariDrivers = from r in
    Formula1.GetChampions()
    from c in r.Cars
    where c == "Ferrari"
    orderby r.LastName
    select r;
```

现在建立另一个相同的查询, 但 `where` 子句的参数不同, 以获得所有驾驶 McLaren 的冠军。最好不要再次编写相同的查询。而可以创建一个方法, 给它传送参数 `car`:

```
private static IEnumerable < Racer >
    GetRacersByCar(string car)
{
    return from r in Formula1.GetChampions()
        from c in r.Cars
        where c == car
        orderby r.LastName
        select r;
}
```

但是, 因为该方法不需要在其他地方使用, 所以应定义一个委托类型的变量来保存 LINQ 查询。变量 `racerByCar` 必须是一个委托类型, 它需要一个字符串参数, 返回 `IEnumerable <Racer>`, 类似于前面实现的方法。为此, 定义了几个泛型委托 `Func<>`, 所以不需要声明自己的委托。把一个  $\lambda$  表达式赋予变量 `racerByCar`。 $\lambda$  表达式的左边定义了一个 `car` 变量, 其类型是 `Func` 委托的第一个泛型参数(字符串)。右边定义了 LINQ 查询, 它使用该参数和 `where` 子句:

```
Func < string, IEnumerable < Racer > > racersByCar =
    Car => from r in Formula1.GetChampions()
        from c in r.Cars
        where c == car
        orderby r.LastName
        select r;
```

现在可以使用 `Intersect()`扩展方法, 获得驾驶 Ferrari 和 McLaren 的所有冠军:

```
Console.WriteLine("World champion with " +
    "Ferrari and McLaren");
```

```
foreach (var racer in racersByCar("Ferrari").
    Intersect(racersByCar("McLaren")))
{
    Console.WriteLine(racer);
}
```

结果只有一个赛手 Niki Lauda:

```
World champion with Ferrari and McLaren
Niki Lauda
```

### 11.2.10 分区

扩展方法 `Take()` 和 `Skip()` 等的分区操作可用于分页，例如显示  $5 \times 5$  个赛手。

在下面的 LINQ 查询中，扩展方法 `Take()` 和 `Skip()` 添加到查询的最后。`Skip()` 方法先忽略根据页面的大小和实际的页数计算出的项数，再使用方法 `Take()` 根据页面的大小提取一定数量的项：

```
int pageSize = 5;

int numberPages = (int)Math.Ceiling(
    Formula1.GetChampions().Count() /
    (double)pageSize);

for (int page = 0; page < numberPages; page++)
{
    Console.WriteLine("Page {0}", page);

    var racers =
        (from r in Formula1.GetChampions()
         orderby r.LastName
         select r.FirstName + " " + r.LastName).
        Skip(page * pageSize).Take(pageSize);

    foreach (var name in racers)
    {
        Console.WriteLine(name);
    }
    Console.WriteLine();
}
```

下面输出了前 3 页：

```
Page 0
Fernando Alonso
Mario Andretti
Alberto Ascari
Jack Brabham
Jim Clark

Page 1
Juan Manuel Fangio
Nino Farina
Emerson Fittipaldi
```

Mika Hakkinen  
Mike Hawthorn

Page 2  
Phil Hill  
Graham Hill  
Damon Hill  
Denny Hulme  
James Hunt

分页在 Windows 或 Web 应用程序中非常有用，可以只给用户显示一部分数据。

#### 提示：

这个分页机制的一个要点是，因为查询会在每个页面上执行，所以改变底层的数据会影响结果。在继续执行分页操作时，会显示新对象。根据不同的情况，这对于应用程序而言可能是一个优点。如果这个操作是不需要的，就可以只对原来的数据源分页，然后使用映射到原数据上的高速缓存。

使用 `TakeWhile()` 和 `SkipWhile()` 方法，还可以传送一个谓词，根据谓词的结果提取或跳过某些项。

### 11.2.11 合计操作符

合计操作符如 `Count()`、`Sum()`、`Min()`、`Max()`、`Average()` 和 `Aggregate()`，不返回一个序列，而返回一个值。

`Count()` 扩展方法返回集合中的项数。下面的 `Count()` 方法应用于 `Racer` 的 `Years` 属性，过滤赛车手，只返回获得冠军次数超过 3 次的赛车手：

```
var query = from r in Formula1.GetChampions()
             where r.Years.Count() > 3
             orderby r.Years.Count() descending
             select new
             {
                 Name = r.FirstName + " " +
                     r.LastName,
                 TimesChampion = r.Years.Count()
             };

foreach (var r in query)
{
    Console.WriteLine("{0} {1}", r.Name,
        r.TimesChampion);
}
```

结果如下：

Michael Schumacher 7  
Juan Manuel Fangio 5  
Alain Prost 4

`Sum()` 方法汇总序列中的所有数字，返回这些数字的和。下面的 `Sum()` 用于计算一个国家赢得比赛的总次数。首先根据国家对手分组，再在新创建的匿名类型中，给 `Wins` 属性赋予某

个国家赢得比赛的总次数。

```
var countries =
    (from c in
      from r in Formula1.GetChampions()
      group r by r.Country into c
      select new
      {
          Country = c.Key,
          Wins = (from r1 in c
                  select r1.Wins).Sum()
      })
    orderby c.Wins descending, c.Country
    select c).Take(5);

foreach (var country in countries)
{
    Console.WriteLine("{0} {1}",
        country.Country, country.Wins);
}
```

根据获得一级方程式冠军的次数，最成功的国家是：

```
UK 138
Germany 91
Brazil 78
France 51
Finland 40
```

方法 `Min()`、`Max()`、`Average()` 和 `Aggregate()` 的使用方式与 `Count()` 和 `Sum()` 相同。`Min()` 返回集合中的最小值，`Max()` 返回集合中的最大值，`Average()` 计算集合中的平均值。对于 `Aggregate()` 方法，可以传送一个  $\lambda$  表达式，对所有的值进行汇总。

### 11.2.12 转换

本章前面提到，查询可以推迟到访问数据项时再执行。在迭代中使用查询，查询会执行。而使用转换操作符会立即执行查询，把结果放在数组、列表或字典中。

在下面的例子中，调用 `ToList()` 扩展方法，立即执行查询，把结果放在 `List<T>` 中：

```
List < Racer > racers =
    (from r in Formula1.GetChampions()
     where r.Starts > 150
     orderby r.Starts descending
     select r).ToList();
foreach (var racer in racers)
{
    Console.WriteLine("{0} {0:S}", racer);
}
```

把返回的对象放在列表中并没有这么简单。例如，对于集合中从赛车到赛手的快速访问，可以使用新类 `Lookup<TKey, TElement>`。

**提示：**

`Dictionary<TKey, TValue>` 只支持一个键对应一个值。在 `System.Linq` 命名空间的类

Lookup<TKey, TElement>中，一个键可以对应多个值。这些类详见第 10 章。

使用复合的 from 查询，可以摊平赛手和赛车序列，创建带有 Car 和 Racer 属性的匿名类型。在返回的 Lookup 对象中，键的类型应是表示汽车的 string，值的类型应是 Racer。为了进行这个选择，可以给 ToLookup()方法的一个重载版本传送一个键和一个元素选择器。键选择器表示 Car 属性，元素选择器表示 Racer 属性。

```
ILookup < string, Racer > racers =
    (from r in Formula1.GetChampions()
     from c in r.Cars
     select new
     {
         Car = c,
         Racer = r
     }).ToLookup(cr => cr.Car, cr => cr.Racer);
if (racers.Contains("Williams"))
{
    foreach (var williamsRacer in
        racers["Williams"])
    {
        Console.WriteLine(williamsRacer);
    }
}
```

用 Lookup 类的索引器访问的所有 Williams 冠军如下：

```
Alan Jones
Keke Rosberg
Nigel Mansell
Alain Prost
Damon Hill
Jacques Villeneuve
```

如果需要在未类型化的集合上使用 LINQ 查询，例如 ArrayList，就可以使用 Cast()方法。在下面的例子中，基于 Object 类型的 ArrayList 集合用 Racer 对象填充。为了定义强类型化的查询，可以使用 Cast()方法。

```
System.Collections.ArrayList list =
    new System.Collections.ArrayList(
        Formula1.GetChampions() as
        System.Collections.ICollection);

var query = from r in list.Cast < Racer > ()
            where r.Country == "USA"
            orderby r.Wins descending
            select r;
foreach (var racer in query)
{
    Console.WriteLine("{0:A}", racer);
}
```

### 11.2.13 生成操作符

生成操作符 Range()、Empty()和 Repeat()不是扩展方法，而是返回序列的正常静态方法。在 LINQ to Objects 中，这些方法可用于 Enumerable 类。



有时需要填充一个范围的数字，此时就应使用 `Range()` 方法。这个方法把第一个参数作为起始值，把第二个参数作为要填充的项数。

```
var values = Enumerable.Range(1, 20);
foreach (var item in values)
{
    Console.Write("{0} ", item);
}
Console.WriteLine();
```

当然，结果如下所示：

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

**提示：**

`Range()` 方法不返回填充了所定义值的集合，这个方法与其他方法一样，也推迟执行查询，返回一个 `RangeEnumerator`，其中只有一个 `yield return` 语句，来递增值。

可以把该结果与其他扩展方法合并起来，获得另一个结果，例如使用 `Select()` 扩展方法：

```
var values = Enumerable.Range(1, 20).
    Select(n => n * 3);
```

`Empty()` 方法返回一个不返回值的迭代器，它可以用于参数需要一个集合，且可以给参数传送空集合的情形。

`Repeat()` 方法返回一个迭代器，该迭代器把同一个值重复特定的次数。

## 11.3 表达式树

在 LINQ to Objects 中，扩展方法需要将一个委托类型作为参数，这样就可以将  $\lambda$  表达式赋予参数。 $\lambda$  表达式也可以赋予 `Expression<T>` 类型的参数。`Expression<T>` 类型指定，来自于  $\lambda$  表达式的表达式树存储在程序集中。这样，就可以在运行期间分析表达式，并进行优化，以便于查询数据源。

下面看看一个前面使用的查询表达式：

```
var brazilRacers = from r in racers
                    where r.Country == "Brazil"
                    orderby r.Wins
                    select r;
```

这个查询表达式使用了扩展方法 `Where`、`OrderBy` 和 `Select`。`Enumerable` 类定义了 `Where` 扩展方法，并将委托类型 `Func<T,bool>` 作为参数谓词。

```
public static IEnumerable<T> Where<T> (this IEnumerable<T> source,
                                       Func<T,bool> predicate);
```

这样，就把  $\lambda$  表达式赋予谓词。这里  $\lambda$  表达式类似于前面介绍的匿名方法。

```
Func<T, bool> predicate = r.Country == "Brazil";
```

Enumerable 类不是唯一一个定义了扩展方法 Where 的类。Queryable<T>类也定义了 Where 扩展方法。这个类对 Where 扩展方法的定义是不同的：

```
public static IQueryable<T> Where<T> (this IQueryable<T> source,
                                         Expression<Func<T,bool>> predicate);
```

其中，λ 表达式赋予类型 Expression<T>，它的操作是不同的：

```
Expression<Func<T, bool>> predicate = r.Country == "Brazil";
```

除了使用委托之外，编译器还会把表达式树放在程序集中。表达式树可以在运行期间读取。表达式树从派生自抽象基类 Expression 的类中建立。Expression 类与 Expression<T>不同。继承了 Expression 的表达式类有 BinaryExpression、ConstantExpression、InvocationExpression、LambdaExpression、NewExpression、NewArrayExpression、TernaryExpression、UnaryExpression 等。编译器会从 λ 表达式中创建表达式树。

例如，λ 表达式 r.Country=="Brazil" 使用了 ParameterExpression、MemberExpression、ConstantExpression 和 MethodCallExpression，来创建一个表达式树，将该树存储在程序集中。之后在运行期间使用这个树，创建一个用于底层数据源的优化查询。

方法 DisplayTree()在控制台上图形化地显示表达式树。其中传送了一个 Expression 对象，并根据表达式类型，把表达式的一些信息写到控制台上。根据表达式的类型，递归调用方法 DisplayTree()。

#### 提示：

在这个方法中，没有处理所有的表达式类型，只处理了下列示例表达式中使用的类型：

```
private static void DisplayTree(int indent,
                                string message, Expression expression)
{
    string output = String.Format("{0} {1}" +
        "! NodeType: {2}; Expr: {3} ",
        "", PadLeft(indent, ' '), message,
        expression.NodeType, expression);

    indent++;
    switch (expression.NodeType)
    {
        case ExpressionType.Lambda:
            Console.WriteLine(output);
            LambdaExpression lambdaExpr =
                (LambdaExpression)expression;
            foreach (var parameter in
                lambdaExpr.Parameters)
            {
                DisplayTree(indent, "Parameter",
                    parameter);
            }
            DisplayTree(indent, "Body",
                lambdaExpr.Body);
            break;
        case ExpressionType.Constant:
            ConstantExpression constExpr =
                (ConstantExpression)expression;
            Console.WriteLine("{0} Const Value: " +
```

```

        "{1}", output, constExpr.Value);
break;
case ExpressionType.Parameter:
    ParameterExpression paramExpr =
        (ParameterExpression)expression;
    Console.WriteLine("{0} Param Type: {1}",
        output, paramExpr.Type.Name);
    break;
case ExpressionType.Equal:
case ExpressionType.AndAlso:
case ExpressionType.GreaterThan:
    BinaryExpression binExpr =
        (BinaryExpression)expression;
    if (binExpr.Method != null)
    {
        Console.WriteLine("{0} Method: {1}",
            output, binExpr.Method.Name);
    }
    else
    {
        Console.WriteLine(output);
    }
    DisplayTree(indent, "Left",
        binExpr.Left);
    DisplayTree(indent, "Right",
        binExpr.Right);
    break;
case ExpressionType.MemberAccess:
    MemberExpression memberExpr =
        (MemberExpression)expression;
    Console.WriteLine("{0} Member Name: " +
        "{1}, Type: {2}", output,
        memberExpr.Member.Name,
        memberExpr.Type.Name);
    DisplayTree(indent, "Member Expr",
        memberExpr.Expression);
    break;
default:
    Console.WriteLine();
    Console.WriteLine("....{0} {1}",
        expression.NodeType,
        expression.Type.Name);
    break;
}
}

```

前面已经介绍了用于显示表达式树的表达式。这是一个  $\lambda$  表达式，它使用一个 **Racer** 参数，表达式体提取赢得比赛次数超过 6 次的巴西赛手：

```

Expression < Func < Racer, bool > > expression =
    r => r.Country == "Brazil" && r.Wins > 6;

DisplayTree(0, "Lambda", expression);

```

下面看看结果。 $\lambda$  表达式包含一个 **Parameter** 和一个 **AndAlso** 节点类型。**AndAlso** 节点类型的左边是一个 **Equal** 节点类型，右边是一个 **GreaterThan** 节点类型。**Equal** 节点类型的左边是 **MemberAccess** 节点类型，右边是 **Constant** 节点类型。

```

Lambda! NodeType: Lambda; Expr: r => ((r.Country = "Brazil")&&(r.Wins>6))

```

```

> Parameter! NodeType: Parameter; Expr: r Param Type: Racer
> Body! NodeType: AndAlso; Expr: ((r.Country = "Brazil") & & (r.Wins > 6))
> > Left! NodeType: Equal; Expr: (r.Country = "Brazil") Method: op_Equality
> > > Left! NodeType: MemberAccess; Expr: r.Country Member Name: Country, Type:
String
> > > > Member Expr! NodeType: Parameter; Expr: r Param Type: Racer
> > > Right! NodeType: Constant; Expr: "Brazil" Const Value: Brazil
> > Right! NodeType: GreaterThan; Expr: (r.Wins > 6)
> > > Left! NodeType: MemberAccess; Expr: r.Wins Member Name: Wins, Type: Int32
> > > > Member Expr! NodeType: Parameter; Expr: r Param Type: Racer
> > > Right! NodeType: Constant; Expr: 6 Const Value: 6

```

使用类型 `Expression<T>` 的一个例子是 LINQ to SQL。LINQ to SQL 用 `Expression<T>` 参数定义了扩展方法。这样，访问数据库的 LINQ 提供程序就可以读取表达式，创建一个运行期间优化的查询，从数据库中获取数据。

## 11.4 LINQ 提供程序

.NET 3.5 包含几个 LINQ 提供程序。LINQ 提供程序为特定的数据源实现了标准的查询操作符。LINQ 提供程序也许会实现 LINQ 定义的更多扩展方法，但至少要实现标准操作符。LINQ to XML 不仅实现了专门用于 XML 的方法，还实现了其他方法，例如 `System.Xml.Linq` 命名空间的 `Extensions` 类定义的方法 `Elements()`、`Descendants` 和 `Ancestors`。

LINQ 提供程序的实现方案是根据命名空间和第一个参数的类型来选择的。实现扩展方法的类的命名空间必须是打开的，否则扩展类就不在作用域内。在 LINQ to Objects 中定义的 `Where()` 方法参数和在 LINQ to SQL 中定义的 `Where()` 方法参数是不同的。

LINQ to Objects 中的 `Where()` 方法是用 `Enumerable` 类定义的：

```

public static IEnumerable < TSource > Where < TSource > (
    this IEnumerable < TSource > source,
    Func < TSource, bool > predicate);

```

在 `System.Linq` 命名空间中，还有另一个类实现了操作符 `Where`。这个实现代码由 LINQ to SQL 使用，这些代码在类 `Queryable` 中：

```

public static IQueryable < TSource > Where < TSource > (
    this IQueryable < TSource > source,
    Expression < Func < TSource, bool > > predicate);

```

这两个类都在 `System.Linq` 命名空间的 `System.Core` 程序集中实现。那么，它是如何定义的？使用了什么方法？无论是用 `Func<TSource, bool>` 参数传送，还是用 `Expression< Func<TSource, bool>>` 参数传送， $\lambda$  表达式都是相同的。只是编译器的操作是不同的，它根据 `source` 参数来选择。编译器根据其参数选择最匹配的方法。在 LINQ to SQL 中定义的 `DataContext` 类的 `GetTable()` 方法返回 `IQueryable<TSource>`，因此 LINQ to SQL 使用类 `Queryable` 的 `Where()` 方法。

LINQ to SQL 提供程序使用表达式树，实现了接口 `IQueryable` 和 `IQueryProvider`。

## 11.5 小结

本章介绍了 C# 3.0 版本中最重要的改进。C#在继续发展，在 C# 2.0 中，主要的新特性是泛型，它为类型安全的泛型集合类、泛型接口和委托奠定了基础。C# 3.0 的主要新特性是 LINQ。通过它可以使用与语言集成的语法查询任意数据源，只要该数据源有提供程序即可。

本章讨论了 LINQ 查询和查询所基于的语言结构，例如扩展方法和  $\lambda$  表达式，还列出了各种 LINQ 查询操作符，它们不仅用于过滤数据源，给数据源排序，还用于执行分区、分组、转换、连接等操作。

LINQ 是一个非常深奥的主题，更多的信息可查阅第 27、29 章和附录 A。还可以下载其他第三方提供程序，例如 LINQ to MySQL、LINQ to Amazon、LINQ to Flickr 和 LINQ to SharePoint。无论使用什么数据源，都可以通过 LINQ 使用相同的查询语法。

另一个重要的概念是表达式树。表达式树允许在运行期间建立对数据源的查询，因为表达式树存储在程序集中。表达式树的用法详见第 27 章。



# 第 12 章

## 内存管理和指针

本章介绍内存管理和内存访问的各个方面。尽管运行库负责为程序员处理大部分内存管理工作，但程序员仍必须理解内存管理的工作原理，了解如何处理未托管的资源。

如果很好地理解了内存管理和 C# 提供的指针功能，也就能很好地集成 C# 代码和原来的代码，并能在非常注重性能的系统高效地处理内存。

本章的主要内容如下：

- 运行库如何在堆栈和堆上分配空间
- 垃圾收集的工作原理
- 如何使用析构函数和 `System.IDisposable` 接口来确保正确释放未托管的资源
- C# 中使用指针的语法
- 如何使用指针实现基于堆栈的高性能数组

### 12.1 后台内存管理

C# 编程的一个优点是程序员不需要担心具体的内存管理，尤其是垃圾收集器会处理所有的内存清理工作。用户可以得到像 C++ 语言那样的效率，而不需要考虑像在 C++ 中那样内存管理工作的复杂性。虽然不必手工管理内存，但仍需理解后台发生的事情。本节要介绍给变量分配内存时计算机内存中发生的情况。

**注意：**

本节的许多内容是没有经过事实证明的。您应把这一节看作是一般规则的简化向导，而不是实现的确切说明。

#### 12.1.1 值数据类型

Windows 使用一个系统：虚拟寻址系统，该系统把程序可用的内存地址映射到硬件内存中的实际地址上，这些任务完全由 Windows 在后台管理，其实际结果是 32 位处理器上的每个进程都可以使用 4GB 的内存——无论计算机上有多少硬盘空间。（在 64 位处理器上，这个数字会更大）。这个 4GB 内存实际上包含了程序的所有部分，包括可执行代码、代码加载的所有 DLL，以及程序运行时使用的所有变量的内容。这个 4GB 内存称为虚拟地址空间，或虚拟内存，为了方便起见，本章将它简称为内存。

4GB 中的每个存储单元都是从 0 开始往上排序的。要访问存储在内存的某个空间中的一个值，就需要提供表示该存储单元的数字。在任何复杂的高级语言中，例如 C#、VB、C++ 和 Java，编译器负责把人们可以理解的变量名称转换为处理器可以理解的内存地址。

在进程的虚拟内存中，有一个区域称为堆栈。堆栈存储不是对象成员的值数据类型。另外，在调用一个方法时，也使用堆栈存储传递给方法的所有参数的复本。为了理解堆栈的工作原理，需要注意在 C# 中变量的作用域。如果变量 a 在变量 b 之前进入作用域，b 就会先出作用域。下面的代码：

```
{
    int a;
    // do something
    {
        int b;
        // do something else
    }
}
```

首先声明 a。在内部的代码块中声明了 b。然后内部的代码块终止，b 就出作用域，最后 a 出作用域。所以 b 的生存期会完全包含在 a 的生存期中。在释放变量时，其顺序总是与给它们分配内存的顺序相反，这就是堆栈的工作方式。

我们不知道堆栈在地址空间的什么地方，这些信息在进行 C# 开发是不需要知道的。堆栈指针(操作系统维护的一个变量)表示堆栈中下一个自由空间的地址。程序第一次运行时，堆栈指针指向为堆栈保留的内存块末尾。堆栈实际上是向下填充的，即从高内存地址向低内存地址填充。当数据入栈后，堆栈指针就会随之调整，以始终指向下一个自由空间。这种情况如图 12-1 所示。在该图中，显示了堆栈指针 800000(十六进制的 0xC3500)，下一个自由空间是地址 799999。

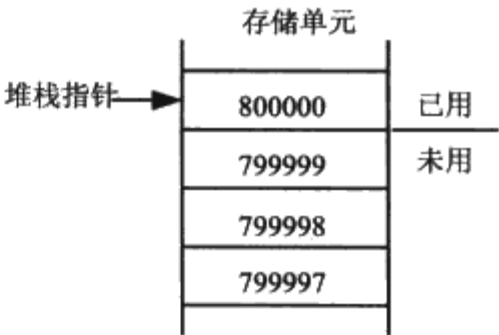


图 12-1

下面的代码会告诉编译器，需要一些存储单元以存储一个整数和一个双精度浮点数，这些存储单元会分别分配给 nRacingCars 和 engineSize，声明每个变量的代码表示开始请求访问这个变量，闭合花括号表示这两个变量出作用域的地方。

```
{
    int nRacingCars = 10;
    double engineSize = 3000.0;
    // do calculations;
}
```

假定使用如图 12-1 所示的堆栈。变量 nRacingCars 进入作用域，赋值为 10，这个值放在存储单元 799996~799999 上，这 4 个字节就在堆栈指针所指空间的下面。有 4 个字节是因为存储 int 要使用 4 个字节。为了容纳该 int，应从堆栈指针中减去 4，所以它现在指向位置 799996，即下一个自由空间 (799995)。

下一行代码声明变量 engineSize(这是一个 double)，把它初始化为 3000.0。double 要占用 8 个字节，所以值 3000.0 占据栈上的存储单元 799988~799995 上，堆栈指针减去 8，再次指向堆

栈上的下一个自由空间。

当 `engineSize` 出作用域时，计算机就知道不再需要这个变量了。因为变量的生存期总是嵌套的，当 `engineSize` 在作用域中时，无论发生什么情况，都可以保证堆栈指针总是会指向存储 `engineSize` 的空间。为了从内存中删除这个变量，应给堆栈指针递增 8，现在指向 `engineSize` 使用过的空间。此处就是放置闭合花括号的地方。当 `nRacingCars` 也出作用域时，堆栈指针就再次递增 4，此时如果内存中又放入另一个变量，从 799999 开始的存储单元就会被覆盖，这些空间以前是存储 `nRacingCars` 的。

如果编译器遇到像 `int i, j` 这样的代码，则这两个变量进入作用域的顺序就是不确定的：两个变量是同时声明的，也是同时出作用域的。此时，变量以什么顺序从内存中删除就不重要了。编译器在内部会确保先放在内存中的那个变量后删除，这样就能保证该规则不会与变量的生存期冲突。

### 12.1.2 引用数据类型

堆栈有非常高的性能，但对于所有的变量来说还是不太灵活。变量的生存期必须嵌套，在许多情况下，这种要求都过于苛刻。通常我们希望使用一个方法分配内存，来存储一些数据，并在方法退出后的很长一段时间内数据仍是可以使用的。只要是用 `new` 运算符来请求存储空间，就存在这种可能性——例如所有的引用类型。此时就要使用托管堆。

如果读者以前编写过需要管理低级内存的 C++ 代码，就会很熟悉堆(heap)。托管堆和 C++ 使用的堆不同，它在垃圾收集器的控制下工作，与传统的堆相比有很显著的性能优势。

托管堆(简称为堆)是进程的可用 4GB 中的另一个内存区域。要了解堆的工作原理和如何为引用数据类型分配内存，看看下面的代码：

```
void DoWork()
{
    Customer arabel;
    arabel = new Customer();
    Customer otherCustomer2 = new EnhancedCustomer();
}
```

在这段代码中，假定存在两个类 `Customer` 和 `EnhancedCustomer`。`EnhancedCustomer` 类扩展了 `Customer` 类。

首先，声明一个 `Customer` 引用 `arabel`，在堆栈上给这个引用分配存储空间，但这仅是一个引用，而不是实际的 `Customer` 对象。`arabel` 引用占用 4 个字节的空間，包含了存储 `Customer` 对象的地址(需要 4 个字节把 0 到 4GB 之间的内存地址表示为一个整数值)。

然后看下一行代码：

```
arabel = new Customer();
```

这行代码完成了以下操作：首先，分配堆上的内存，以存储 `Customer` 实例(一个真正的实例，不只是一个地址)。然后把变量 `arabel` 的值设置为分配给新 `Customer` 对象的内存地址(它还调用合适的 `Customer()` 构造函数初始化类实例中的字段，但我们不必担心这部分)。

`Customer` 实例没有放在堆栈中，而是放在内存的堆中。在这个例子中，现在还不知道一个 `Customer` 对象占用多少字节，但为了讨论方便，假定是 32 字节。这 32 字节包含了 `Customer`

实例字段，和.NET 用于识别和管理其类实例的一些信息。

为了在堆上找到一个存储新 Customer 对象的存储位置，.NET 运行库在堆中搜索，选取第一个未使用的、32 字节的连续块。为了讨论方便，假定其地址是 200000，arabel 引用占用堆栈中的 799996~799999 位置。这表示在实例化 arabel 对象前，内存的内容应如图 12-2 所示。

给 Customer 对象分配空间后，内存内容应如图 12-3 所示。注意，与堆栈不同，堆上的内存是向上分配的，所以自由空间在已用空间的上面。

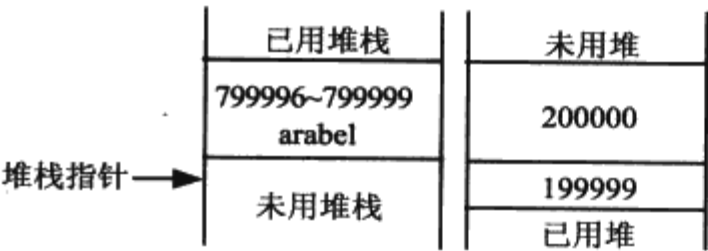


图 12-2

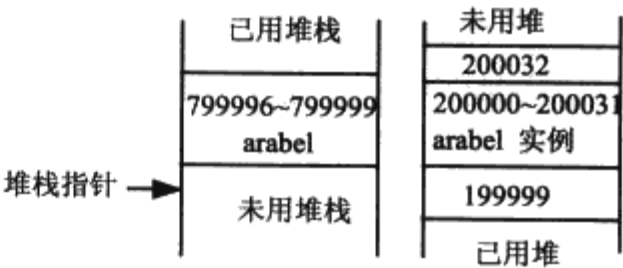


图 12-3

下一行代码声明了一个 Customer 引用，并实例化一个 Customer 对象。在这个例子中，需要在堆栈上为 otherCustomer2 引用分配空间，同时，也需要在堆上为 otherCustomer2 对象分配空间：

```
Customer otherCustomer2 = new EnhancedCustomer();
```

该行把堆栈上的 4 字节分配给 otherCustomer2 引用，它存储在 799992~799995 位置上，而 otherCustomer2 对象在堆上从 200032 开始向上分配空间。

从这个例子可以看出，建立引用变量的过程要比建立值变量的过程更复杂，且不能避免性能的降低。实际上，我们对这个过程进行了过分的简化，因为.NET 运行库需要保存堆的状态信息，在堆中添加新数据时，这些信息也需要更新。尽管有这些性能损失，但仍有一种机制，在给变量分配内存时，不会受到堆栈的限制。把一个引用变量的值赋予另一个相同类型的变量，就有两个引用内存中同一对象的变量了。当一个引用变量出作用域时，它会从堆栈中删除，如上一节所述，但引用对象的数据仍保留在堆中，一直到程序停止，或垃圾收集器删除它为止，而只有在该数据不再被任何变量引用时，才会被删除。

这就是引用数据类型的强大之处，在 C#代码中广泛使用了这个特性。这说明，我们可以对数据的生存期进行非常强大的控制，因为只要有对数据的引用，该数据就肯定存在于堆上。

12.1.3 垃圾收集

由上面的讨论和图可以看出，托管堆的工作方式非常类似于堆栈，在某种程度上，对象会在内存中一个挨一个地放置，这样就很容易使用指向下一个空闲存储单元的堆指针，来确定下一个对象的位置。在堆上添加更多的对象时，也容易调整。但这比较复杂，因为基于堆的对象的生存期与引用它们的基于堆栈的变量的作用域不匹配。

在垃圾收集器运行时，会在堆中删除不再引用的所有对象。在完成删除动作后，堆会立即把对象分散开来，与已经释放的内存混合在一起，如图 12-4 所示。

如果托管的堆也是这样，在其上给新对象分配内存就成为一个很难处理的过程，运行库必须搜索整个堆，才能找到足够大的内存块来存储每个新对象。但是，垃圾收集器不会让堆处于这种状态。只要它释放了能释放的所有对象，就会把其他对象移动回堆的端部，再次形成一个连续的块。因此，堆可以继续像堆栈那样确定在什么地方存储新对象。当然，在移动对象时，这些对象的所有引用都需要用正确的新地址来更新，但垃圾收集器也会处理更新问题。

垃圾收集器的这个压缩操作是托管的堆与未托管的旧堆的区别所在。使用托管的堆，就只需要读取堆指针的值即可，而不是搜索链接地址列表，来查找一个地方来放置新数据。因此，在.NET 下实例化对象要快得多。有趣的是，访问它们也比较快，因为对象会压缩到堆上相同的内存区域，这样需要交换的页面较少。Microsoft 相信，尽管垃圾收集器需要做一些工作，压缩堆，修改它移动的所有对象引用，致使性能降低，但这些性能会得到弥补。



图 12-4

注意：

一般情况下，垃圾收集器在.NET 运行库认为需要时运行。可以通过调用 `System.GC.Collect()`，强迫垃圾收集器在代码的某个地方运行，`System.GC` 是一个表示垃圾收集器的.NET 基类，`Collect()`方法则调用垃圾收集器。但是，这种方式适用的场合很少，例如，代码中有大量的对象刚刚停止引用，就适合调用垃圾收集器。但是，垃圾收集器的逻辑不能保证在一次垃圾收集过程中，所有未引用的对象都从堆中删除。

12.2 释放未托管的资源

垃圾收集器的出现意味着，通常不需要担心不再需要的对象，只要让这些对象的所有引用都超出作用域，并允许垃圾收集器在需要时释放资源即可。但是，垃圾收集器不知道如何释放未托管的资源(例如文件句柄、网络连接和数据库连接)。托管类在封装对未托管资源的直接或间接引用时，需要制定专门的规则，确保未托管的资源在回收类的一个实例时释放。

在定义一个类时，可以使用两种机制来自动释放未托管的资源。这些机制常常放在一起实现，因为每个机制都为问题提供了略有不同的解决方法。这两个机制是：

- 声明一个析构函数(或终结器)，作为类的一个成员
- 在类中执行 `System.IDisposable` 接口

下面依次讨论这两个机制，然后介绍如何同时实现它们，以获得最佳的效果。

12.2.1 析构函数

前面介绍了构造函数可以指定必须在创建类的实例时进行的某些操作，在垃圾收集器删除对象之前，也可以调用析构函数。由于执行这个操作，所以析构函数初看起来似乎是放置释放



未托管资源、执行一般清理操作的代码的最佳地方。但是，事情并不是如此简单。

**注意：**

在讨论 C# 中的析构函数时，在底层的 .NET 结构中，这些函数称为终结器(finalizer)。在 C# 中定义析构函数时，编译器发送给程序集的实际上是 `Finalize()` 方法。这不会影响源代码，但如果需要查看程序集的内容，就应知道这个事实。

C++ 开发人员应很熟悉析构函数的语法，它看起来类似于一个方法，与包含类同名，但前面加上了一个发音符号(~)。它没有返回类型，不带参数，没有访问修饰符。下面是一个例子：

```
class MyClass
{
    ~MyClass()
    {
        // destructor implementation
    }
}
```

C# 编译器在编译析构函数时，会隐式地把析构函数的代码编译为 `Finalize()` 方法的对应代码，确保执行父类的 `Finalize()` 方法。下面列出了编译器为 `~MyClass()` 析构函数生成的 IL 的对应 C# 代码：

```
protected override void Finalize()
{
    try
    {
        // destructor implementation
    }
    finally
    {
        base.Finalize();
    }
}
```

如上所示，在 `~MyClass()` 析构函数中执行的代码封装在 `Finalize()` 方法的一个 `try` 块中。对父类 `Finalize()` 方法的调用放在 `finally` 块中，确保该调用的执行。第 14 章会讨论 `try` 块和 `finally` 块。

有经验的 C++ 开发人员大量使用了析构函数，有时不仅用于清理资源，还提供调试信息或执行其他任务。C# 析构函数的使用要比在 C++ 中少得多，与 C++ 析构函数相比，C# 析构函数的问题是它们的不确定性。在删除 C++ 对象时，其析构函数会立即运行。但由于垃圾收集器的工作方式，无法确定 C# 对象的析构函数何时执行。所以，不能在析构函数中放置需要在某一时刻运行的代码，也不应使用能以任意顺序对不同类实例调用的析构函数。如果对象占用了宝贵而重要的资源，应尽快释放这些资源，此时就不能等待垃圾收集器来释放了。

另一个问题是 C# 析构函数的执行会延迟对象最终从内存中删除的时间。没有析构函数的对象会在垃圾收集器的一次处理中从内存中删除，但有析构函数的对象需要两次处理才能删除：第一次调用析构函数时，没有删除对象，第二次调用才真正删除对象。另外，运行库使用一个线程来执行所有对象的 `Finalize()` 方法。如果频繁使用析构函数，而且使用它们执行长时间的清理任务，对性能的影响就会非常显著。

### 12.2.2 IDisposable 接口

在 C# 中，推荐使用 `System.IDisposable` 接口替代析构函数。`IDisposable` 接口定义了一个模式(具有语言级的支持)，为释放未托管的资源提供了确定的机制，并避免产生析构函数固有的与垃圾函数器相关的问题。`IDisposable` 接口声明了一个方法 `Dispose()`，它不带参数，返回 `void`，`Myclass` 的方法 `Dispose()` 的执行代码如下：

```
class MyClass : IDisposable
{
    public void Dispose()
    {
        // implementation
    }
}
```

`Dispose()` 的执行代码显式释放由对象直接使用的的所有未托管资源，并在所有实现 `IDisposable` 接口的封装对象上调用 `Dispose()`。这样，`Dispose()` 方法在释放未托管资源的时间方面提供了精确的控制。

假定有一个类 `ResourceGobbler`，它使用某些外部资源，且执行 `IDisposable` 接口。如果要实例化这个类的实例，使用它，然后释放它，就可以使用下面的代码：

```
ResourceGobbler theInstance = new ResourceGobbler();

// do your processing

theInstance.Dispose();
```

如果在处理过程中出现异常，这段代码就没有释放 `theInstance` 使用的资源，所以应使用 `try` 块(详见第 14 章)，编写下面的代码：

```
ResourceGobbler theInstance = null;
try
{
    theInstance = new ResourceGobbler();

    // do your processing
}
finally
{
    if (theInstance != null)
    {
        theInstance.Dispose();
    }
}
```

即使在处理过程中出现了异常，这个版本也可以确保总是在 `theInstance` 上调用 `Dispose()`，总是释放由 `theInstance` 使用的资源。但是，如果总是要重复这样的结构，代码就很容易被混淆。C# 提供了一种语法，可以确保在执行 `IDisposable` 接口的对象的引用超出作用域时，在该对象上自动调用 `Dispose()`。该语法使用了 `using` 关键字来完成这一工作——该关键字在完全不同的环境下，与命名空间没有关系。下面的代码生成与 `try` 块相对应的 IL 代码：

```
using (ResourceGobbler theInstance = new ResourceGobbler())
```

```
{  
    // do your processing  
}
```

`using` 语句的后面是一对圆括号，其中是引用变量的声明和实例化，该语句使变量放在随附的语句块中。另外，在变量超出作用域时，即使出现异常，也会自动调用其 `Dispose()` 方法。如果已经使用 `try` 块来捕获其他异常，就会比较清晰，如果避免使用 `using` 语句，仅在已有的 `try` 块的 `finally` 子句中调用 `Dispose()`，还可以避免进行额外的缩进。

#### 注意：

对于某些类来说，使用 `Close()` 方法要比 `Dispose()` 更富有逻辑性，例如，在处理文件或数据库连接时就是这样。在这些情况下，常常实现 `IDisposable` 接口，再执行一个独立的 `Close()` 方法，来调用 `Dispose()`。这种方法在类的使用上比较清晰，还支持 C# 提供的 `using` 语句。

### 12.2.3 实现 `IDisposable` 接口和析构函数

前面的章节讨论了类所使用的释放未托管资源的两种方式：

- 利用运行库强制执行的析构函数，但析构函数的执行是不确定的，而且，由于垃圾收集器的工作方式，它会给运行库增加不可接受的系统开销。
- `IDisposable` 接口提供了一种机制，允许类的用户控制释放资源的时间，但需要确保执行 `Dispose()`。

一般情况下，最好的方法是执行这两种机制，获得这两种机制的优点，克服其缺点。假定大多数程序员都能正确调用 `Dispose()`，同时把执行析构函数作为一种安全的机制，以防没有调用 `Dispose()`。下面是一个双重实现的例子：

```
using System;  
  
public class ResourceHolder : IDisposable  
{  
    private bool isDisposed = false;  
  
    public void Dispose()  
    {  
        Dispose(true);  
        GC.SuppressFinalize(this);  
    }  
  
    protected virtual void Dispose(bool disposing)  
    {  
        if (!isDisposed)  
        {  
            if (disposing)  
            {  
                // Cleanup managed objects by calling their  
                // Dispose() methods.  
            }  
            // Cleanup unmanaged objects  
        }  
        isDisposed=true;  
    }  
}
```



```

~ResourceHolder()
{
    Dispose (false);
}

public void SomeMethod()
{
    // Ensure object not already disposed before execution of any method
    if (isDisposed)
    {
        throw new ObjectDisposedException("ResourceHolder");
    }

    // method implementation...
}
}

```

可以看出, `Dispose()` 有第二个 `protected` 重载方法, 它带一个 `bool` 参数, 这是真正完成清理工作的方法。 `Dispose(bool)` 由析构函数和 `IDisposable.Dispose()` 调用。这个方式的重点是确保所有的清理代码都放在一个地方。

传递给 `Dispose(bool)` 的参数表示 `Dispose(bool)` 是由析构函数调用, 还是由 `IDisposable.Dispose()` 调用——`Dispose(bool)` 不应从代码的其他地方调用, 其原因是:

- 如果客户调用 `IDisposable.Dispose()`, 该客户就指定应清理所有与该对象相关的资源, 包括托管和非托管的资源。
- 如果调用了析构函数, 原则上所有的资源仍需要清理。但是在这种情况下, 析构函数必须由垃圾收集器调用, 而且不应访问其他托管的对象, 因为我们不再能确定它们的状态了。在这种情况下, 最好清理已知的未托管资源, 希望引用的托管对象还有析构函数, 执行自己的清理过程。

`isDisposed` 成员变量表示对象是否已被删除, 并允许确保不多次删除成员变量。它还允许在执行实例方法之前测试对象是否已释放, 如 `SomeMethod()` 所示。这个简单的方法不是线程安全的, 需要调用者确保在同一时刻只有一个线程调用方法。要求客户进行同步是一个合理的假定, 在整个 .NET 类库中反复使用了这个假定(例如在集合类中)。第 19 章将讨论线程和同步。

最后, `IDisposable.Dispose()` 包含一个对 `System.GC.SuppressFinalize()` 方法的调用。GC 表示垃圾收集器, `SuppressFinalize()` 方法则告诉垃圾收集器有一个类不再需要调用其析构函数了。因为 `Dispose()` 已经完成了所有需要的清理工作, 所以析构函数不需要做任何工作。调用 `SuppressFinalize()` 就意味着垃圾收集器认为这个对象根本没有析构函数。

## 12.3 不安全的代码

如前面的章节所述, C# 非常擅长于隐藏基本内存管理, 因为它使用了垃圾收集器和引用。但是, 有时需要直接访问内存, 例如由于性能问题, 要在外部(非 .NET 环境)的 DLL 中访问一个函数, 该函数需要把一个指针当作参数来传递(许多 Windows API 函数就是这样)。本节将论述 C# 直接访问内存内容的功能。

### 12.3.1 用指针直接访问内存

下面把指针当作一个新论题来介绍，而实际上，指针并不是新东西，因为在代码中可以自由使用引用，而引用就是一个类型安全的指针。前面已经介绍了表示对象和数组的变量实际上包含存储相应数据(引用)的内存地址。指针只是一个以与引用相同的方式存储地址的变量。其区别是 C# 不允许直接访问引用变量包含的地址。有了引用后，从语法上看，变量就可以存储引用的实际内容。

C# 引用主要用于使 C# 语言易于使用，防止用户无意中执行某些破坏内存中内容的操作，另一方面，使用指针，就可以访问实际内存地址，执行新类型的操作。例如，给地址加上 4 字节，就可以查看甚至修改存储在新地址中的数据。

下面是使用指针的两个主要原因：

- 向后兼容性。尽管 .NET 运行库提供了许多工具，但仍可以调用内部的 Windows API 函数。对于某些操作来说，这可能是完成任务的唯一方式。这些 API 函数都是用 C 语言编写的，通常要求把指针作为其参数。但在许多情况下，还可以使用 `DllImport` 声明，以避免使用指针，例如使用 `System.IntPtr` 类。
- 性能。在一些情况下，速度是最重要的，而指针可以提供最优性能。假定用户知道自己在做什么，就可以确保以最高效的方式访问或处理数据。但是，注意在代码的其他区域中，不使用指针，也可以对性能做必要的改进。请使用代码配置文件，查找代码中的瓶颈，代码配置文件随 VS2008 一起安装。

但是，这种低级内存访问也是有代价的。使用指针的语法比引用类型更复杂。而且，指针使用起来比较困难，需要非常高的编程技巧和很强的能力，仔细考虑代码所完成的逻辑操作，才能成功地使用指针。如果不仔细，使用指针很容易在程序中引入微妙的难以查找的错误。例如很容易重写其他变量，导致堆栈溢出，访问某些没有存储变量的内存区域，甚至重写 .NET 运行库所需要的代码信息，因而使程序崩溃。

另外，如果使用指针，就必须为代码获取代码访问安全机制的高级别信任，否则就不能执行。在默认的代码访问安全策略中，只有代码运行在本地机器上，这才是可能的。如果代码必须运行在远程地点，例如 Internet，用户就必须给代码授予额外的许可，代码才能工作。除非用户信任您和代码，否则他们不会授予这些许可，第 20 章将讨论代码访问安全性。

尽管有这些问题，但指针在编写高效的代码时是一种非常强大和灵活的工具。

**注意：**

这里强烈建议不要使用指针，因为如果使用指针，代码不仅难以编写和调试，而且无法通过 CLR 的内存类型安全检查(详见第 1 章)。

#### 1. 用 `unsafe` 关键字编写不安全的代码

因为使用指针会带来相关的风险，所以 C# 只允许在特别标记的代码块中使用指针。标记代码所用的关键字是 `unsafe`。下面的代码把一个方法标记为 `unsafe`：

```
unsafe int GetSomeNumber()  
{
```



```
// code that can use pointers
}
```

任何方法都可以标记为 `unsafe`——无论该方法是否应用了其他修饰符(例如, 静态方法、虚拟方法等)。在这种方法中, `unsafe` 修饰符还会应用到方法的参数上, 允许把指针用作参数。还可以把整个类或结构标记为 `unsafe`, 表示所有的成员都是不安全的:

```
unsafe class MyClass
{
    // any method in this class can now use pointers
}
```

同样, 可以把成员标记为 `unsafe`:

```
class MyClass
{
    unsafe int *pX; // declaration of a pointer field in a class
}
```

也可以把方法中的一个代码块标记为 `unsafe`:

```
void MyMethod()
{
    // code that doesn't use pointers
    unsafe
    {
        // unsafe code that uses pointers here
    }
    // more 'safe' code that doesn't use pointers
}
```

但要注意, 不能把局部变量本身标记为 `unsafe`:

```
int MyMethod()
{
    unsafe int *pX; // WRONG
}
```

如果要使用不安全的局部变量, 就需要在不安全的方法或语句块中声明和使用它。在使用指针前还有一步要完成。C#编译器会拒绝不安全的代码, 除非告诉编译器代码包含不安全的代码块。标记所用的关键字是 `unsafe`。因此, 要编译包含不安全代码块的文件 `MySource.cs`(假定没有其他编译器选项), 就要使用下述命令:

```
csc /unsafe MySource.cs
```

或者

```
csc -unsafe MySource.cs
```

注意:

如果使用 Visual Studio 2005 或 2008, 就可以在项目属性窗口的 Build 选项卡中找到编译不安全代码的选项。

## 2. 指针的语法

把代码块标记为 `unsafe` 后，就可以使用下面的语法声明指针：

```
int* pWidth, pHeight;  
double* pResult;  
byte*[] pFlags;
```

这段代码声明了 4 个变量，`pWidth` 和 `pHeight` 是整数指针，`pResult` 是 `double` 型指针，`pFlags` 是 `byte` 型的指针数组。我们常常在指针变量名的前面使用前缀 `p` 来表示这些变量是指针。在变量声明中，符号 `*` 表示声明一个指针，换言之，就是存储特定类型的变量的地址。

### 提示：

C++ 开发人员应注意，这个语法与 C# 中的语法是不同的。C# 语句中的 `int* pX, pY`；对应于 C++ 语句中的 `int *pX, *pY`；在 C# 中，`*` 符号与类型相关，而不是与变量名相关。

声明了指针类型的变量后，就可以用与一般变量的方式使用它们，但首先需要学习另外两个运算符：

- `&` 表示“取地址”，并把一个值数据类型转换为指针，例如 `int` 转换为 `*int`。这个运算符称为寻址运算符。
- `*` 表示“获取地址的内容”，把一个指针转换为值数据类型(例如，`*float` 转换为 `float`)。这个运算符称为“间接寻址运算符”(有时称为“取消引用运算符”)。

从这些定义中可以看出，`&` 和 `*` 的作用是相反的。

### 注意：

符号 `&` 和 `*` 也表示按位 AND(`&`) 和乘法(`*`)运算符，那么如何以这种方式使用它们？答案是在实际使用时它们是不会混淆的：用户和编译器总是知道在什么情况下这两个符号有什么含义，因为按照新指针的定义，这些符号总是以一元运算符的形式出现——它们只作用于一个变量，并出现在代码中变量的前面。另一方面，按位 AND 和乘法运算符是二元运算符，它们需要两个操作数。

下面的代码说明了如何使用这些运算符：

```
int x = 10;  
int* pX, pY;  
pX = &x;  
pY = pX;  
*pY = 20;
```

首先声明一个整数 `x`，其值是 10。接着声明两个整数指针 `pX` 和 `pY`。然后把 `pX` 设置为指向 `x`(换言之，把 `pX` 的内容设置为 `x` 的地址)。把 `pX` 的值赋予 `pY`，所以 `pY` 也指向 `x`。最后，在语句 `*pY = 20` 中，把值 20 赋予 `pY` 指向的地址。实际上是把 `x` 的内容改为 20，因为 `pY` 指向 `x`。注意在这里，变量 `pY` 和 `x` 之间没有任何关系。只是此时 `pY` 碰巧指向存储 `x` 的存储单元而已。

要进一步理解这个过程，假定 x 存储在堆栈的存储单元 0x12F8C4 到 0x12F8C7 中(十进制就是 1243332 到 1243335，即有 4 个存储单元，因为 int 占用 4 字节)。因为堆栈向下分配内存，所以变量 pX 存储在 0x12F8C0 到 0x12F8C3 的位置上，pY 存储在 0x12F8BC 到 0x12F8BF 的位置上。注意，pX 和 pY 也分别占用 4 字节。这不是因为 int 占用 4 字节，而是因为在 32 位处理器上，需要用 4 字节存储一个地址。利用这些地址，在执行完上述代码后，堆栈应如图 12-5 所示。

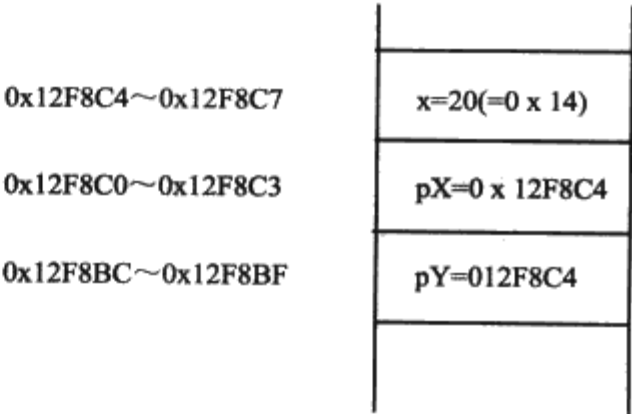


图 12-5

注意：

这个示例使用 int 来说明该过程，其中 int 存储在 32 位处理器中堆栈的连续空间上，但并不是所有的数据类型都会存储在连续的空间中。原因是 32 位处理器最擅长于在 4 字节的内存块中获取数据。这种机器上的内存会分解为 4 字节的块，在 Windows 上，每个块都时常称为 DWORD，因为这是 32 位无符号 int 在 .NET 出现之前的名字。这是从内存中获取 DWORD 的最高效的方式——跨越 DWORD 边界存储数据通常会降低硬件的性能。因此，.NET 运行库通常会给某些数据类型加上一些空间，使它们占用的内存是 4 的倍数。例如，short 数据占用 2 字节，但如果把一个 short 放在堆栈中，堆栈指针仍会减少 4，而不是 2，这样，下一个存储在堆栈中的变量就仍从 DWORD 的边界开始存储。

可以把指针声明为任意一种值类型——即任何预定义的数据类型 uint、int 和 byte 等，也可以声明为一个结构。但是不能把指针声明为一个类或数组，因为这么做会使垃圾收集器出现问题。为了正常工作，垃圾收集器需要知道在堆上创建了什么类实例，它们在什么地方。但如果代码使用指针处理类，将很容易破坏堆中 .NET 运行库为垃圾收集器维护的与类相关的信息。在这里，垃圾收集器可以访问的数据类型称为托管类型，而指针只能声明为非托管类型，因为垃圾收集器不能处理它们。

3. 将指针转换为整数类型

由于指针实际上存储了一个表示地址的整数，所以任何指针中的地址都可以转换为任何整数类型。指针到整数类型的转换必须是显式指定的，隐式的转换是不允许的。例如，编写下面的代码是合法的：

```
int x = 10;
int* pX, pY;
pX = &x;
pY = pX;
*pY = 20;
uint y = (uint)pX;
int* pD = (int*)y;
```

把指针 pX 中包含的地址转换为一个 uint，存储在变量 y 中。接着把 y 转换回 int\*，存储在新变量 pD 中。因此 pD 也指向 x 的值。

把指针的值转换为整数类型的主要目的是显示它。Console.Write()和 Console.WriteLine()方法没有带指针的重载方法，所以必须把指针转换为整数类型，这两个方法才能接受和显示它们：

```
Console.WriteLine("Address is" + pX); // wrong - will give a
                                     // compilation error
Console.WriteLine("Address is" + (uint) pX); // OK
```

可以把一个指针转换为任何整数类型，但是，因为在 32 位系统上，地址占用 4 字节，把指针转换为不是 uint、long 或 ulong 的数据类型，肯定会导致溢出错误(int 也可能导致这个问题，因为它的取值范围是 -20 亿~20 亿，而地址的取值范围是 0~40 亿)。C#是用于 64 位处理器的，地址占用 8 字节。因此在这样的系统上，把指针转换为非 ulong 的类型，就可能导致溢出错误。还要注意，checked 关键字不能用于涉及指针的转换。对于这种转换，即使在设置 checked 的情况下，发生溢出时也不会抛出异常。.NET 运行库假定，如果使用指针，就知道自己要做什么，不处理可能出现的溢出。

#### 4. 指针类型之间的转换

也可以在指向不同类型的指针之间进行显式的转换。例如：

```
byte aByte = 8;
byte* pByte = &aByte;
double* pDouble = (double*)pByte;
```

这是一段合法的代码，但如果要执行这段代码，就要小心了。在上面的示例中，如果要查找指针 pDouble 指向的 double，就会查找包含 1 字节(aByte)的内存，和一些其他内存，把它们当作包含一个 double 的内存区域来对待——这不会得到一个有意义的值。但是，可以在类型之间转换，实现类型的统一，或者把指针转换为其他类型，例如把指针转换为 sbyte，检查内存的单个字节。

#### 5. void 指针

如果要使用一个指针，但不希望指定它指向的数据类型，就可以把指针声明为 void：

```
int* pointerToInt;
void* pointerToVoid;
pointerToVoid = (void*)pointerToInt;
```

void 型指针的主要用途是调用需要 void\*型参数的 API 函数。在 C#语言中，使用 void 指针的情况并不是很多。特殊情况下，如果试图使用\*运算符间接引用 void 指针，编译器就会标记一个错误。

#### 6. 指针的算法

可以给指针加减整数。但是，编译器很智能，知道如何执行这个操作。例如，假定有一个 int 指针，要在其值上加 1。编译器会假定我们要查找 int 后面的存储单元，因此会给该值加上 4 字节，即加上 int 的字节数。如果这是一个 double 指针，加 1 就表示在指针的值上加 8 字节，即 double 的字节数。只有指针是指向 byte 或 sbyte(都是 1 字节)，才会给该指针的值加上 1。

可以对指针使用运算符+、-、+=、-=、++和--，这些运算符右边的变量必须是 long 或 ulong 类型。

注意：

不允许对 void 指针执行算术运算。

例如，假定有如下定义：

```
uint u = 3;
byte b = 8;
double d = 10.0;
uint* pUInt = &u;      // size of a uint is 4
byte* pByte = &b;       // size of a byte is 1
double* pDouble = &d;   // size of a double is 8
```

下面假定这些指针的地址是：

- pUInt: 1243332
- pByte: 1243328
- pDouble: 1243320

执行这段代码后：

```
++pUInt;          // adds (1*4)= 4 bytes to pUInt
pByte -= 3;       // subtracts (3*1)=3 bytes from pByte
double* pDouble2 = pDouble + 4; // pDouble2 = pDouble + 32 bytes (4*8 bytes)
```

指针应包含的内容是：

- pUInt: 1243336
- pByte: 1243325
- pDouble2: 1243352

提示：

给类型为 T 的指针加上 X，其中指针的值为 P，则得到的结果是  $P + X * (\text{sizeof}(T))$ 。

注意：

使用这个规则时要小心。如果给定类型的连续值存储在连续的存储单元中，指针加法就允许在存储单元中移动指针。但如果类型是 byte 或 char，其总字节数就不是 4 的倍数，连续值就不是默认地存储在连续的存储单元中。

如果两个指针都指向相同的数据类型，也可以把一个指针从另一个指针中减去。此时，结果是一个 long，其值是指针值的差被该数据类型所占用的字节数整除的结果：

```
double* pD1 = (double*)1243324; // note that it is perfectly valid to
                                // initialize a pointer like this.
double* pD2 = (double*)1243300;
long L = pD1 - pD2;              // gives the result 3 (=24/sizeof(double))
```

## 7. sizeof 运算符

这一节将介绍如何确定各种数据类型的大小。如果需要在代码中使用类型的大小，就可以



使用 `sizeof` 运算符，它的参数是数据类型的名称，返回该类型占用的字节数。例如：

```
int x = sizeof(double);
```

这将设置 `x` 的值为 8。

使用 `sizeof` 的优点是不必在代码中硬编码数据类型的大小，使代码的移植性更强。对于预定义的数据类型，`sizeof` 返回表 12-1 所示的值。

表 12-1

<code>sizeof(sbyte) = 1;</code>	<code>sizeof(byte) = 1;</code>
<code>sizeof(short) = 2;</code>	<code>sizeof(ushort) = 2;</code>
<code>sizeof(int) = 4;</code>	<code>sizeof(uint) = 4;</code>
<code>sizeof(long) = 8;</code>	<code>sizeof(ulong) = 8;</code>
<code>sizeof(char) = 2;</code>	<code>sizeof(float) = 4;</code>
<code>sizeof(double) = 8;</code>	<code>sizeof(bool) = 1;</code>

也可以对自己定义的结构使用 `sizeof`，但此时得到的结果取决于结构中的字段。不能对类使用 `sizeof`。它只能用于不安全的代码块。

8. 结构指针：指针成员访问运算符

结构指针的工作方式与预定义值类型的指针的工作方式是一样的。但是这有一个条件：结构不能包含任何引用类型，这是因为前面介绍的一个限制——指针不能指向任何引用类型。为了避免这种情况，如果创建一个指针，它指向包含引用类型的结构，编译器就会标记一个错误。

假定定义了如下结构：

```
struct MyStruct
{
    public long X;
    public float F;
}
```

就可以给它定义一个指针：

```
MyStruct* pStruct;
```

对其进行初始化：

```
MyStruct Struct = new MyStruct();
pStruct = &Struct;
```

也可以通过指针访问结构的成员值：

```
(*pStruct).X = 4;
(*pStruct).F = 3.4f;
```

但是，这个语法有点复杂。因此，C# 定义了另一个运算符，用一种比较简单的语法，通过指针访问结构的成员，该语法称为指针成员访问运算符，其符号是一个短划线，后跟一个大于

号: ->。

注意:

C++开发人员认识指针成员访问操作符。因为 C++使用这个符号完成相同的任务。

使用这个指针成员访问运算符, 上述代码可以重写为:

```
pStruct->X = 4;
pStruct->F = 3.4f;
```

也可以直接把合适类型的指针设置为指向结构中的一个字段:

```
long* pL = &(Struct.X);
float* pF = &(Struct.F);
```

或者

```
long* pL = &(pStruct->X);
float* pF = &(pStruct->F);
```

## 9. 类成员指针

前面说过, 不能创建指向类的指针, 这是因为垃圾收集器不维护指针的任何信息, 只维护引用的信息, 因此创建指向类的指针会使垃圾收集器不能正常工作。

但是, 大多数类都包含值类型的成员, 可以为这些值类型成员创建指针, 但这需要一种特殊的语法。例如, 假定把上面示例中的结构重写为类:

```
class MyClass
{
    public long X;
    public float F;
}
```

然后就可以为它的字段 X 和 F 创建指针了, 方法与前面一样。但这么做会生成一个编译错误:

```
MyClass myObject = new MyClass();
long* pL = &( myObject.X);    // wrong--compilation error
float* pF = &( myObject.F);    // wrong--compilation error
```

X 和 F 都是非托管类型, 它们嵌入在一个对象中, 存储在堆上。在垃圾收集的过程中, 垃圾收集器会把 MyObject 移动到内存的一个新单元上, 这样, pL 和 pF 就会指向错误的存储单元。由于存在这个问题, 所以编译器不允许以这种方式把托管类型的成员地址分配给指针。

解决这个问题的方法是使用 fixed 关键字, 它会告诉垃圾收集器, 类实例的某些成员有指向它们的指针, 所以这些实例不能移动。如果要声明一个指针, 使用 fixed 的语法如下所示:

```
MyClass myObject = new MyClass();
fixed (long* pObject = &( myObject.X))
{
    // do something
}
```

在关键字 fixed 后面的圆括号中, 定义和初始化指针变量。这个指针变量(在本例中是 pObject)

的作用域是花括号标记的 `fixed` 块。这样，垃圾收集器就知道，在执行 `fixed` 块中的代码时，不能移动 `MyObject` 对象。

如果要声明多个这样的指针，可以在同一个代码块前放置多个 `fixed` 语句：

```
MyClass myObject = new MyClass();
fixed (long* pX = &( myObject.X))
fixed (float* pF = &( myObject.F))
{
    // do something
}
```

如果要在不同的阶段固定几个指针，还可以嵌套整个 `fixed` 块：

```
MyClass myObject = new MyClass();
fixed (long* pX = &( myObject.X))
{
    // do something with pX
    fixed (float* pF = &( myObject.F))
    {
        // do something else with pF
    }
}
```

也可以在同一个 `fixed` 语句中初始化多个变量，但这些变量的类型必须相同：

```
MyClass myObject = new MyClass();
MyClass myObject2 = new MyClass();
fixed (long* pX = &( myObject.X), pX2 = &( myObject2.X))
{
    // etc.
}
```

在上述情况中，是否声明不同的指针，让它们指向相同或不同对象中的字段，或者指向不与类实例相关的静态字段，这一点是不重要的。

### 12.3.2 指针示例：PointerPlayaround

下面给出一个使用指针的示例：`PointerPlayaround`。它执行一些简单的指针操作，显示结果，还允许查看内存中发生的情况，并确定变量存储在什么地方：

```
using System;

namespace Wrox.ProCSharp.Memory
{
    class MainEntryPoint
    {
        static unsafe void Main()
        {
            int x=10;
            short y = -1;
            byte y2 = 4;
            double z = 1.5;
            int* pX = &x;
            short* pY = &y;
            double* pZ = &z;
```

```

Console.WriteLine(
    "Address of x is 0x{0:X}, size is {1}, value is {2}",
    (uint)&x, sizeof(int), x);
Console.WriteLine(
    "Address of y is 0x{0:X}, size is {1}, value is {2}",
    (uint)&y, sizeof(short), y);
Console.WriteLine(
    "Address of y2 is 0x{0:X}, size is {1}, value is {2}",
    (uint)&y2, sizeof(byte), y2);
Console.WriteLine(
    "Address of z is 0x{0:X}, size is {1}, value is {2}",
    (uint)&z, sizeof(double), z);
Console.WriteLine(
    "Address of pX=&x is 0x{0:X}, size is {1}, value is 0x{2:X}",
    (uint)&pX, sizeof(int*), (uint)pX);
Console.WriteLine(
    "Address of pY=&y is 0x{0:X}, size is {1}, value is 0x{2:X}",
    (uint)&pY, sizeof(short*), (uint)pY);
Console.WriteLine(
    "Address of pZ=&z is 0x{0:X}, size is {1}, value is 0x{2:X}",
    (uint)&pZ, sizeof(double*), (uint)pZ);

*pX = 20;
Console.WriteLine("After setting *pX, x = {0}", x);
Console.WriteLine("*pX = {0}", *pX);

pZ = (double*)pX;
Console.WriteLine("x treated as a double = {0}", *pZ);

Console.ReadLine();
}
}
}

```

这段代码声明了 4 个值变量：

- int x
- short y
- byte y2
- double z

还声明了指向这 3 个值的指针：pX、pY、pZ。

然后显示这 3 个变量的值，以及它们的大小和地址。注意在获取 pX、pY 和 pZ 的地址时，我们查看的是指针的指针，即值的地址的地址！还要注意，与显示地址的常见方式一致，在 Console.WriteLine() 命令中使用 {0:X} 格式说明符，确保该内存地址以十六进制格式显示。

最后，使用指针 pX 把 x 的值改为 20，执行一些指针转换，如果把 x 的内容当作 double 类型，就会得到无意义的结果。

编译运行这段代码，在得到的结果中，我们将列出用 /unsafe 标志进行编译和不用 /unsafe 标志进行编译的结果：

```

csc PointerPlayaround.cs
Microsoft (R) Visual C# 2008 Compiler version 3.05.20706.1
for Microsoft (R) .NET Framework version 3.5
Copyright (C) Microsoft Corporation. All rights reserved.

PointerPlayaround.cs(7,26): error CS0227: Unsafe code may only appear if

```



```

compiling with /unsafe

csc /unsafe PointerPlayaround.cs
Microsoft (R) Visual C# 2008 Compiler version 3.05.20706.1
for Microsoft (R) .NET Framework version 3.5
Copyright (C) Microsoft Corporation. All rights reserved.

PointerPlayaround
Address of x is 0x12F4B0, size is 4, value is 10
Address of y is 0x12F4AC, size is 2, value is -1
Address of y2 is 0x12F4A8, size is 1, value is 4
Address of z is 0x12F4A0, size is 8, value is 1.5
Address of pX=&x is 0x12F49C, size is 4, value is 0x12F4B0
Address of pY=&y is 0x12F498, size is 4, value is 0x12F4AC
Address of pZ=&z is 0x12F494, size is 4, value is 0x12F4A0
After setting *pX, x = 20
*pX = 20
x treated as a double = 2.86965129997082E-308

```

检查这些结果，可以证实本章前面的“后台内存管理”一节描述的堆栈操作，即堆栈给变量向下分配内存。注意，这还证实了堆栈中的内存块总是按照 4 字节的倍数进行分配。例如，y 是一个 short(其大小为 2 字节)，其地址是 1242284(十进制)，表示为该变量分配的内存区域是 1242284~1242287。如果 .NET 运行库严格地逐个排列变量，则 y 应只占用 2 个存储单元 1242284 和 1242285。

下一个示例 PointerPlayaround2 介绍指针的算术，以及结构指针和类成员指针。开始时，定义一个结构 CurrencyStruct，把货币值表示为美元和美分，再定义一个对应的类 CurrencyClass：

```

internal struct CurrencyStruct
{
    public long Dollars;
    public byte Cents;

    public override string ToString()
    {
        return "$" + Dollars + "." + Cents;
    }
}

class CurrencyClass
{
    public long Dollars;
    public byte Cents;

    public override string ToString()
    {
        return "$" + Dollars + "." + Cents;
    }
}

```

定义好了结构和类后，就可以对它们应用指针了。下面的代码是一个新的示例。这段代码比较长，我们对此将做详细讲解。首先显示 CurrencyStruct 结构的字节数，创建它的两个实例和一些指针，再使用 pAmount 指针初始化一个 CurrencyStruct 结构 amount1，显示变量的地址：

```

public static unsafe void Main()
{
    Console.WriteLine(
        "Size of Currency struct is " + sizeof(CurrencyStruct));
    CurrencyStruct amount1, amount2;
}

```



```

CurrencyStruct* pAmount = &amount1;
long* pDollars = &(pAmount->Dollars);
byte* pCents = &(pAmount->Cents);

Console.WriteLine("Address of amount1 is 0x{0:X}", (uint)&amount1);
Console.WriteLine("Address of amount2 is 0x{0:X}", (uint)&amount2);
Console.WriteLine("Address of pAmount is 0x{0:X}", (uint)&pAmount);
Console.WriteLine("Address of pDollars is 0x{0:X}", (uint)&pDollars);
Console.WriteLine("Address of pCents is 0x{0:X}", (uint)&pCents);
pAmount->Dollars = 20;
*pCents = 50;
Console.WriteLine("amount1 contains " + amount1);

```

现在根据堆栈的工作方式，执行一些指针操作。因为变量是按顺序声明的，所以 `amount2` 存储在 `amount1` 后面的地址上，`sizeof(CurrencyStruct)` 返回 16(见后面的屏幕输出)，所以 `CurrencyStruct` 占用的字节数是 4 的倍数。在递减了 `Currency` 指针后，它就指向 `amount2`：

```

--pAmount; // this should get it to point to amount2
Console.WriteLine("amount2 has address 0x{0:X} and contains {1}",
    (uint)pAmount, *pAmount);

```

在调用 `Console.WriteLine()` 语句时，它显示了 `amount2` 的内容，但还没有对它进行初始化。显示出来的东西就是随机的垃圾——在执行该示例前存储在内存中该单元的内容。但这有一个要点：一般情况下，C#编译器会禁止使用未初始化的值，但在开始使用指针时，就很容易绕过许多通常的编译检查。此时我们这么做，是因为编译器无法知道我们实际上要显示的是 `amount2` 的内容。因为知道了堆栈的工作方式，所以可以说出递减 `pAmount` 的结果是什么。使用指针算法，可以访问各种编译器通常禁止访问的变量和存储单元，因此指针算法是不安全的。

接下来在 `pCents` 指针上进行指针运算。`pCents` 目前指向 `amount1.Cents`，但此处的目的是使用指针算法让它指向 `amount2.Cents`，而不是直接告诉编译器我们要做什么。为此，需要从 `pCents` 指针所包含的地址中减去 `sizeof(Currency)`：

```

// do some clever casting to get pCents to point to cents
// inside amount2
CurrencyStruct* pTempCurrency = (CurrencyStruct*)pCents;
pCents = (byte*) (--pTempCurrency);
Console.WriteLine("Address of pCents is now 0x{0:X}", (uint)&pCents);

```

最后，使用 `fixed` 关键字创建一些指向类实例中字段的指针，使用这些指针设置这个实例的值。注意，这也是我们第一次查看存储在堆中(而不是堆栈)的项的地址：

```

Console.WriteLine("\nNow with classes");
// now try it out with classes
CurrencyClass amount3 = new CurrencyClass();

fixed(long* pDollars2 = &(amount3.Dollars))
fixed(byte* pCents2 = &(amount3.Cents))
{
    Console.WriteLine(
        "amount3.Dollars has address 0x{0:X}", (uint)pDollars2);
    Console.WriteLine(
        "amount3.Cents has address 0x{0:X}", (uint)pCents2);
    *pDollars2 = -100;
    Console.WriteLine("amount3 contains " + amount3);
}

```

编译并运行这段代码，得到如下所示的结果：

```
csc /unsafe PointerPlayaround2.cs
Microsoft (R) Visual C# 2008 Compiler version 3.05.20706.1
for Microsoft (R) .NET Framework version 3.5
Copyright (C) Microsoft Corporation. All rights reserved.
```

#### PointerPlayaround2

```
Size of CurrencyStruct struct is 16
Address of amount1 is 0x12F4A4
Address of amount2 is 0x12F494
Address of pAmount is 0x12F490
Address of pDollars is 0x12F48C
Address of pCents is 0x12F488
amount1 contains $20.50
amount2 has address 0x12F494 and contains $0.0
Address of pCents is now 0x12F488
```

Now with classes

```
amount3.Dollars has address 0xA64414
amount3.Cents has address 0xA6441C
amount3 contains $-100.0
```

注意在这个结果中，显示了未初始化的 amount2 值，CurrencyStruct 结构的字节数是 16，大于其字段的字节数(1 long(=8) + 1 byte(=1) = 9 字节)。

### 12.3.3 使用指针优化性能

前面用许多篇幅介绍了使用指针可以完成的各种任务，但在前面的示例中，仅是处理内存，让有兴趣的人们了解底层发生了什么事，并没有帮助人们编写出好的代码！本节将应用我们对指针的理解，用一个示例来说明使用指针可以大大提高性能。

#### 1. 创建基于堆栈的数组

本节将介绍指针的一个主要应用领域：在堆栈中创建高性能、低系统开销的数组。第 2 章介绍了 C# 如何支持数组的处理。C# 很容易使用一维数组和矩形或锯齿形多维数组，但有一个缺点：这些数组实际上都是对象，是 System.Array 的实例。因此数组只能存储在堆上，会增加系统开销。有时，我们希望创建一个使用时间比较短的高性能数组，不希望有引用对象的系统开销。而使用指针就可以做到，但指针只能用于一维数组。

为了创建一个高性能的数组，需要使用另一个关键字：stackalloc。stackalloc 命令指示 .NET 运行库分配堆栈上一定量的内存。在调用它时，需要为它提供两条信息：

- 要存储的数据类型
- 需要存储的数据项数。

例如，分配足够的内存，以存储 10 个 decimal 数据项，可以编写下面的代码：

```
decimal* pDecimals = stackalloc decimal [10];
```

注意，这个命令只是分配堆栈内存而已。它不会试图把内存初始化为任何默认值，这正好符合我们的目的。因为这是一个高性能的数组，给它不必要地初始化值会降低性能。

同样，要存储 20 个 double 数据项，可以编写下面的代码：

```
double* pDoubles = stackalloc double [20];
```

虽然这行代码指定把变量的个数存储为一个常数，但它是在运行时计算的一个数字。所以可以把上面的示例写为：

```
int size;
size = 20; // or some other value calculated at run-time
double* pDoubles = stackalloc double [size];
```

从这些代码段中可以看出，`stackalloc` 的语法有点不寻常。它的后面紧跟要存储的数据类型名(该数据类型必须是一个值类型)，之后把需要的变量个数放在方括号中。分配的字节数是变量个数乘以 `sizeof(数据类型)`。在这里，使用方括号表示这是一个数组。如果给 20 个 `double` 数据分配存储单元，就得到了一个有 20 个元素的 `double` 数组，最简单的数组类型是逐个存储元素的内存块，如图 12-6 所示。

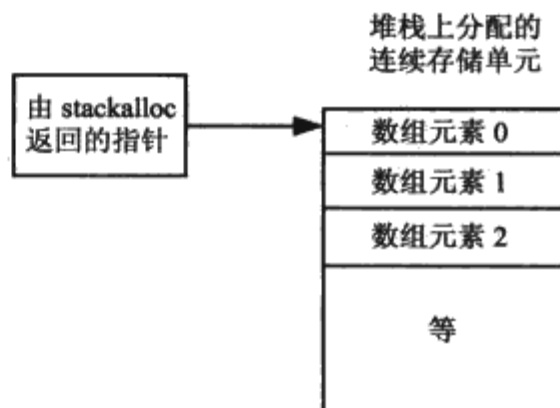


图 12-6

在图 12-6 中，显示了一个由 `stackalloc` 返回的指针，`stackalloc` 总是返回分配数据类型的指针，它指向新分配内存块的顶部。要使用这个内存块，可以取消对返回指针的引用。例如，给 20 个 `double` 数据分配内存后，把第一个元素(数组中的元素 0)设置为 3.0，可以编写下面的代码：

```
double* pDoubles = stackalloc double [20];
*pDoubles = 3.0;
```

要访问数组的下一个元素，可以使用指针算法。如前所述，如果给一个指针加 1，它的值就会增加其数据类型的字节数。在本例中，就会把指针指向下一个空闲存储单元。因此可以把数组的第二个元素(数组中元素号为 1)设置为 8.4：

```
double* pDoubles = stackalloc double [20];
*pDoubles = 3.0;
*(pDoubles+1) = 8.4;
```

同样，可以用表达式 `*(pDoubles+X)` 获得数组中下标为 X 的元素。

这样，就得到一种访问数组中元素的方式，但对于一般目的，使用这种语法过于复杂。C# 为此定义了另一种语法。对指针应用方括号时，C# 为方括号提供了一种非常明确的含义。如果变量 `p` 是任意指针类型，`X` 是一个整数，表达式 `p[X]` 就被编译器解释为 `*(p+X)`，这适用于所有的指针，不仅仅是用 `stackalloc` 初始化的指针。利用这个简捷的记号，就可以用一种非常方便的方式访问数组。实际上，访问基于堆栈的一维数组所使用的语法与访问基于堆的、由 `System.Array` 类表示的数组是一样的：

```
double *pDoubles = stackalloc double [20];
pDoubles[0] = 3.0; // pDoubles[0] is the same as *pDoubles
pDoubles[1] = 8.4; // pDoubles[1] is the same as *(pDoubles+1)
```

注意：

把数组的语法应用于指针并不是新东西。自从开发出 C 和 C++ 语言以来，它们就是这两种

语言的基础部分。实际上，C++开发人员会把这里用 `stackalloc` 获得的、基于堆栈的数组完全等同于传统的基于堆栈的 C 和 C++ 数组。这个语法和指针与数组的链接方式是 C 语言在 70 年代后期流行起来的原因之一，也是指针的使用成为 C 和 C++ 中一种大众化编程技巧的主要原因。

高性能的数组可以用与一般 C# 数组相同的方式访问，但需要注意：在 C# 中，下面的代码会抛出一个异常：

```
double [] myDoubleArray = new double [20];
myDoubleArray[50] = 3.0;
```

抛出异常的原因是：使用越界的下标来访问数组：下标是 50，而允许的最大下标是 19。但是，如果使用 `stackalloc` 声明了一个相同的数组，对数组进行边界检查时，这个数组中没有封装任何对象，因此下面的代码不会抛出异常：

```
double* pDoubles = stackalloc double [20];
pDoubles[50] = 3.0;
```

在这段代码中，我们分配了足够的内存来存储 20 个 `double` 类型的数据。接着把 `sizeof(double)` 存储单元的起始位置设置为该存储单元的起始位置加上 `50*sizeof(double)` 存储单元，来保存双精度值 3.0。但这个存储单元超出了刚才为 `double` 分配的内存区域。谁也不知道这个地址存储了什么数据。最好是只使用某个当前未使用的内存，但所重写的空间也有可能在是堆栈上用于存储其他变量，或者是某个正在执行的方法的返回地址。因此，使用指针获得高性能的同时，也会付出一些代价：需要确保自己知道在做什么，否则就会抛出非常古怪的运行时错误。

## 2. 示例 QuickArray

下面用一个 `stackalloc` 示例 `QuickArray` 来结束关于指针的讨论。在这个示例中，程序仅要求用户提供为数组分配的元素数。然后代码使用 `stackalloc` 给 `long` 型数组分配一定的存储单元。这个数组的元素是从 0 开始的整数的平方，结果显示在控制台上：

```
using System;

namespace Wrox.ProCSharp.Memory
{
    class MainEntryPoint
    {
        static unsafe void Main()
        {
            Console.WriteLine("How big an array do you want? \n> ");
            string userInput = Console.ReadLine();
            uint size = uint.Parse(userInput);

            long* pArray = stackalloc long [(int)size];
            for (int i=0 ; i<size ; i++)
                pArray[i] = i*i;

            for (int i=0 ; i<size ; i++)
                Console.WriteLine("Element {0} = {1}", i, *(pArray+i));
        }
    }
}
```



运行这个示例，得到如下所示的结果：

**QuickArray**

How big an array do you want?

> 15

Element 0 = 0  
Element 1 = 1  
Element 2 = 4  
Element 3 = 9  
Element 4 = 16  
Element 5 = 25  
Element 6 = 36  
Element 7 = 49  
Element 8 = 64  
Element 9 = 81  
Element 10 = 100  
Element 11 = 121  
Element 12 = 144  
Element 13 = 169  
Element 14 = 196

## 12.4 小结

要想成为真正优秀的 C#程序员，必须牢固掌握存储单元和垃圾收集的工作原理。本章描述了 CLR 管理以及在堆和堆栈上分配内存的方式，讨论了如何编写正确释放未托管资源的类，并介绍如何在 C#中使用指针，这些都是很难理解的高级主题，初学者常常不能正确实现。

本章为后面第 14 章的错误处理和第 19 章的线程使用打下了基础。下一章介绍 C#中的反射技术。



# 第13章

## 反 射

反射是一个普通术语，描述了在运行过程中检查和处理程序元素的功能。例如，反射允许完成以下任务：

- 枚举类型的成员
- 实例化新对象
- 执行对象的成员
- 查找类型的信息
- 查找程序集的信息
- 检查应用于类型的定制特性
- 创建和编译新程序集

这个列表列出了许多功能，包括.NET Framework 类库提供的一些最强大、最复杂的功能。但本章不可能介绍反射的所有功能，仅讨论最常用的功能。

本章的内容如下：

- 定制特性，定制特性允许把定制的元数据与程序元素关联起来。这些元数据是在编译过程中创建的，并嵌入到程序集中。
- 在运行期间使用反射的一些功能检查这些元数据。
- 支持反射的一些基类，包括 `System.Type` 和 `System.Reflection.Assembly` 类，它们可以访问反射提供的许多功能。

为了演示定制特性和反射，我们将开发一个示例，说明公司如何定期升级软件，自动解释升级的信息。在这个示例中，要定义几个定制特性，表示程序元素最后修改或创建的日期，以及发生了什么变化。然后使用反射开发一个应用程序，在程序集中查找这些特性，自动显示软件自某个给定日期以来升级的所有信息。

本章要讨论的另一个示例是一个应用程序，该程序读写数据库，并使用定制特性，把类和特性标记为对应的数据库表和列。然后在运行期间从程序集中读取这些特性，使程序可以自动从数据库的相应位置检索或写入数据，无需为每个表或列编写特定的逻辑。

### 13.1 定制特性

前面介绍了如何在程序的各个数据项上定义特性。这些特性都是 Microsoft 定义好的，作为.NET Framework 类库的一部分，许多特性都得到了 C#编译器的支持。对于这些特性，编译

器可以以特殊的方式定制编译过程,例如,可以根据 StructLayout 特性中的信息在内存中布置结构。

.NET Framework 也允许用户定义自己的特性。显然,这些特性不会影响编译过程,因为编译器不能识别它们,但这些特性在应用于程序元素时,可以在编译好的程序集中用作元数据。

这些元数据在文档说明中非常有用。但是,使定制特性非常强大的因素是使用反射,代码可以读取这些元数据,使用它们在运行期间作出决策,也就是说,定制特性可以直接影响代码运行的方式。例如,定制特性可以用于支持对定制许可类进行声明代码访问安全检查,把信息与程序元素关联起来,由测试工具使用,或者在开发可扩展的架构时,允许加载插件或模块。

### 13.1.1 编写定制特性

为了理解编写定制特性的方式,应了解一下在编译器遇到代码中某个应用了定制特性的元素时,该如何处理。以数据库为例,假定有一个 C# 属性声明,如下所示。

```
[FieldName("SocialSecurityNumber")]
public string SocialSecurityNumber
{
    get {
        // etc.
```

当 C# 编译器发现这个属性有一个特性 FieldName 时,首先会把字符串 Attribute 添加到这个名称的后面,形成一个组合名称 FieldNameAttribute,然后在其搜索路径的所有命名空间(即在 using 语句中提及的命名空间)中搜索有指定名称的类。但要注意,如果用一个特性标记数据项,而该特性的名称以字符串 Attribute 结尾,编译器就不会把该字符串加到组合名称中,而是不修改该特性名。因此,上面的代码实际上等价于:

```
[FieldNameAttribute("SocialSecurityNumber")]
public string SocialSecurityNumber
{
    get {
        // etc.
```

编译器会找到含有该名称的类,且这个类直接或间接派生自 System.Attribute。编译器还认为这个类包含控制特性用法的信息。特别是属性类需要指定:

- 特性可以应用到哪些程序元素上(类、结构、属性和方法等)
- 它是否可以多次应用到同一个程序元素上
- 特性在应用到类或接口上时,是否由派生类和接口继承
- 这个特性有哪些必选和可选参数

如果编译器找不到对应的特性类,或者找到一个这样的特性类,但使用特性的方式与特性类中的信息不匹配,编译器就会产生一个编译错误。例如,如果特性类指定该特性只能应用于字段,但我们把它应用到结构定义上,就会产生一个编译错误。

继续上面的示例,假定定义了一个 FieldName 特性:

```
[AttributeUsage(AttributeTargets.Property,
    AllowMultiple=false,
    Inherited=false)]
```

```
public class FieldNameAttribute : Attribute
{
    private string name;
    public FieldNameAttribute(string name)
    {
        this.name = name;
    }
}
```

下面几节讨论这个定义中的每个元素。

### 1. AttributeUsage 特性

要注意的第一个问题是特性(attribute)类本身用一个特性 `System.AttributeUsage` 来标记。这是 Microsoft 定义的一个特性, C#编译器为它提供了特殊的支持(`AttributeUsage` 根本不是一个特性, 它更像一个元特性, 因为它只能应用到其他特性上, 不能应用到类上)。 `AttributeUsage` 主要用于表示定制特性可以应用到哪些类型的程序元素上。这些信息由它的第一个参数给出, 该参数是必选的, 其类型是枚举类型 `AttributeTargets`。在上面的示例中, 指定 `FieldName` 特性只能应用到属性(property)上——这是因为我们在前面的代码段中把它应用到属性上。

`AttributeTargets` 枚举的成员如下:

- All
- Assembly
- Class
- Constructor
- Delegate
- Enum
- Event
- Field
- GenericParameter(仅.NET 2.0 提供)
- Interface
- Method
- Module
- Parameter
- Property
- ReturnValue
- Struct

这个列表列出了可以应用该特性的所有程序元素。注意在把特性应用到程序元素上时, 应把特性放在元素前面的方括号中。但是, 在上面的列表中, 有两个值不对应于任何程序元素: `Assembly` 和 `Module`。特性可以应用到整个程序集或模块中, 而不是应用到代码中的一个元素上, 在这种情况下, 这个特性可以放在源代码的任何地方, 但需要用关键字 `assembly` 或 `module` 来做前缀:

```
[assembly: SomeAssemblyAttribute(Parameters)]
[module: SomeAssemblyAttribute(Parameters)]
```

在指定定制特性的有效目标元素时，可以使用按位 OR 运算符把这些值组合起来。例如，如果指定 `FieldName` 特性可以应用到属性和字段上，可以编写下面的代码：

```
[AttributeUsage(AttributeTargets.Property | AttributeTargets.Field,
    AllowMultiple=false,
    Inherited=false)]
public class FieldNameAttribute : Attribute
```

也可以使用 `AttributeTargets.All` 指定特性可以应用到所有类型的程序元素上。`AttributeUsage` 特性还包含另外两个参数 `AllowMultiple` 和 `Inherited`。它们用不同的语法来指定：`<AttributeName>=<AttributeValue>`，而不是只给出这些参数的值。这些参数是可选的，如果需要，可以忽略它们。

`AllowMultiple` 参数表示一个特性是否可以多次应用到同一项上，这里把它设置为 `false`，表示如果编译器遇到下述代码，就会产生一个错误：

```
[FieldName("SocialSecurityNumber")]
[FieldName("NationalInsuranceNumber")]
public string SocialSecurityNumber
{
    // etc.
```

如果 `Inherited` 参数设置为 `true`，就表示应用到类或接口上的特性也可以自动应用到所有派生的类或接口上。如果特性应用到方法或属性上，也可以自动应用到该方法或属性的重写版本上。

## 2. 指定特性参数

下面介绍如何指定定制特性的参数。在编译器遇到下述语句时：

```
[FieldName("SocialSecurityNumber")]
public string SocialSecurityNumber
{
    // etc.
```

会检查传送给特性的参数(在本例中，是一个字符串)，并查找该特性中带这些参数的构造函数。如果找到一个这样的构造函数，编译器就会把指定的元数据传送给程序集。如果找不到，就生成一个编译错误。如后面所述，反射会从程序集中读取元数据，并实例化它们表示的特性类。因此，编译器需要确保存在这样的构造函数，才能在运行期间实例化指定的特性。

在本例中，仅为 `FieldNameAttribute` 提供了一个构造函数，而这个构造函数有一个字符串参数。因此，在把 `FieldNameAttribute` 特性应用到一个属性上时，必须为它提供一个字符串参数，如上面的代码所示。

如果可以选择特性的参数类型，当然可以提供构造函数的不同重载方法，但一般是仅提供一个构造函数，使用属性来定义其他可选参数，下面将介绍可选参数。

## 3. 指定特性的可选参数

在 `AttributeUsage` 特性中，可以使用另一个语法，把可选参数添加到特性中。这个语法指

定可选参数的名称和值, 处理特性类中的公共属性或字段。例如, 假定修改 `SocialSecurityNumber` 属性的定义, 如下所示:

```
[FieldName("SocialSecurityNumber", Comment="This is the primary key field")]
public string SocialSecurityNumber
{
    // etc.
```

在本例中, 编译器识别第二个参数的语法 `<ParameterName>=<ParameterValue>`, 所以不会把这个参数传递给 `FieldNameAttribute` 构造函数, 而是查找一个有该名称的公用属性或字段(最好不要使用公用字段, 所以一般情况下要使用属性), 编译器可以用这个属性设置第二个参数的值。如果希望上面的代码工作, 必须给 `FieldNameAttribute` 添加一些代码:

```
[AttributeUsage(AttributeTargets.Property,
    AllowMultiple=false,
    Inherited=false)]
public class FieldNameAttribute : Attribute
{
    private string comment;
    public string Comment
    {
        get
        {
            return comment;
        }
        set
        {
            comment = value;
        }
    }
    // etc.
```

### 13.1.2 定制特性示例: WhatsNewAttributes

本节开始编写前面描述过的示例 `WhatsNewAttributes`, 该示例提供了一个特性, 表示最后一次修改程序元素的时间。这个示例比前面所有的示例都复杂, 因为它包含 3 个不同的程序集:

- `WhatsNewAttributes` 程序集, 它包含特性的定义。
- `VectorClass` 程序集, 包含所应用的特性的代码。
- `LookUpWhatsNew` 程序集, 包含显示已改变的数据项信息的项目。

其中, 只有 `LookUpWhatsNew` 是前面使用的一个控制台应用程序, 其余两个程序集都是库文件, 它们都包含类的定义, 但都没有程序的入口。对于 `VectorClass` 程序集, 我们使用了 `VectorAsCollection` 示例, 但删除了入口和测试代码类, 只剩下 `Vector` 类。这些类详见本章后面的内容。

在命令行上编译, 以此管理 3 个相关的程序集要求较高的技巧, 所以我们分别给出编译这 3 个源文件的命令。也可以编辑代码示例, (可以从 Wrox Press 网站上下载), 组合为一个 Visual Studio 解决方案, 详见第 15 章。下载的文件包含所需的 Visual Studio 2008 解决方案文件。



## 1. WhatsNewAttributes 库程序集

首先从核心的 WhatsNewAttributes 程序集开始。其源代码包含在文件 WhatsNewAttributes.cs 中，该文件位于本章示例代码的 WhatsNewAttributes 解决方案的 WhatsNewAttributes 项目中。编译为库的语法非常简单：在命令行上，给编译器提供标记 target:library 即可。要编译 WhatsNewAttributes，键入：

```
csc /target:library WhatsNewAttributes.cs
```

WhatsNewAttributes.cs 文件定义了两个特性类 LastModifiedDate 和 SupportsWhatsNew-Attribute。LastModifiedDate 特性可以用于标记最后一次修改数据项的时间，它有两个必选参数(该参数传递给构造函数)：修改的日期和包含描述修改的字符串。它还有一个可选参数 Issues (表示存在一个公共属性)，它可以描述该数据项的任何重要问题。

在现实生活中，或许想把特性应用到任何对象上。为了使代码比较简单，这里仅允许将它应用于类和方法，并允许它多次应用到同一项上(AllowMultiple=true)，因为可以多次修改一项，每次修改都需要用一个不同的特性实例来标记。

SupportsWhatsNew 是一个较小的类，表示不带任何参数的特性。这个特性是一个程序集的特性，用于把程序集标记为通过 LastModifiedDate 维护的文档说明书。这样，以后查看这个程序集的程序会知道，它读取的程序集是我们使用自动文档说明过程生成的那个程序集。这部分示例的完整源代码如下所示：

```
using System;
namespace Wrox.ProCSharp.WhatsNewAttributes
{
    [AttributeUsage(
        AttributeTargets.Class | AttributeTargets.Method,
        AllowMultiple=true, Inherited=false)]
    public class LastModifiedDate : Attribute
    {
        private DateTime dateModified;
        private string changes;
        private string issues;

        public LastModifiedDate(string dateModified, string changes)
        {
            this.dateModified = DateTime.Parse(dateModified);
            this.changes = changes;
        }

        public DateTime DateModified
        {
            get
            { return dateModified; }
        }

        public string Changes
        {
            get
            { return changes; }
        }

        public string Issues
```

```

    {
        get
        { return issues; }
        set
        { issues = value; }
    }
}

[AttributeUsage(AttributeTargets.Assembly)]
public class SupportsWhatsNewAttribute : Attribute
{
}
}

```

从上面的描述可以看出，上面的代码非常简单。但要注意，不必将 set 访问器提供给 Changes 和 DateModified 属性，不需要这些访问器是因为在构造函数中，这些参数都设置为必选参数。需要 get 访问器，是因为以后可以读取这些特性的值。

## 2. VectorClass 程序集

本节就使用这些特性，我们用前面的 VectorAsCollection 示例的修订版本来说明。注意这里需要引用刚才创建的 WhatsNewAttributes 库，还需要使用 using 语句指定相应的命名空间，这样编译器才能识别出这些特性：

```

using System;
using System.Collections;
using System.Text;
using Wrox.ProCSharp.WhatsNewAttributes;

```

```
[assembly: SupportsWhatsNew]
```

在这段代码中，添加了一行用 SupportsWhatsNew 特性标记程序集本身的代码。

下面考虑 Vector 类的代码。我们并不是真的要修改这个类中的任何内容，只是添加两个 LastModified 特性，以标记出本章对 Vector 类进行的操作。把 Vector 定义为一个类，而不是结构，以简化后面显示特性所编写的代码(在 VectorAsCollection 示例中，Vector 是一个结构，但其枚举器是一个类。于是，这个示例的下一个版本在查看程序集时，必须同时考虑类和结构。这会使例子比较复杂)。

```

namespace Wrox.ProCSharp.VectorClass
{
    [LastModified("14 Feb 2008", "IEnumerable interface implemented " +
        "So Vector can now be treated as a collection")]
    [LastModified("10 Feb 2008", "IFormattable interface implemented " +
        "So Vector now responds to format specifiers N and VE")]
    class Vector : IFormattable, IEnumerable
    {
        public double x, y, z;

        public Vector(double x, double y, double z)
        {
            this.x = x;
            this.y = y;
            this.z = z;
        }
    }
}

```

```

    }

    [LastModified("10 Feb 2008",
        "Method added in order to provide formatting support")]
    public string ToString(string format, IFormatProvider formatProvider)
    {
        if (format == null)
        {
            return ToString();
        }
    }

```

再把包含的 `VectorEnumerator` 类标记为 `new`:

```

    [LastModified("14 Feb 2008",
        "Class created as part of collection support for Vector")]
    private class VectorEnumerator : IEnumerator
    {

```

为了在命令行上编译这段代码，应键入下面的命令：

```
csc /target:library /reference:WhatsNewAttributes.dll VectorClass.cs
```

上面是这个示例的代码。目前还不能运行它，因为我们只有两个库。在描述了反射的工作原理后，就介绍这个示例的最后一部分，查找和显示这些特性。

## 13.2 反射

本节先介绍 `System.Type` 类，通过这个类可以访问任何给定数据类型的信息。然后简要介绍 `System.Reflection.Assembly` 类，它可以用于访问给定程序集的信息，或者把这个程序集加载到程序中。最后把本节的代码和上一节的代码结合起来，完成 `WhatsNewAttributes` 示例。

### 13.2.1 `System.Type` 类

在本书中的许多场合中都使用了 `Type` 类，但它只存储类型的引用：

```
Type t = typeof(double)
```

我们以前把 `Type` 看作一个类，但它实际上是一个抽象的基类。只要实例化了一个 `Type` 对象，就实例化了 `Type` 的一个派生类。`Type` 有与每种数据类型对应的派生类，但一般情况下派生的类只提供各种 `Type` 方法和属性的不同重载，返回对应数据类型的正确数据。一般不增加新的方法或属性。获取指向给定类型的 `Type` 引用有 3 种常用方式：

- 使用 C# 的 `typeof` 运算符，如上所示。这个运算符的参数是类型的名称(不放在引号中)。
- 使用 `GetType()` 方法，所有的类都会从 `System.Object` 继承这个类。

```
double d = 10;
Type t = d.GetType();
```

在一个变量上调用 `GetType()`，而不是把类型的名称作为其参数。但要注意，返回的 `Type` 对象仍只与该数据类型相关：它不包含与类型实例相关的任何信息。如果有一个对象引用，但

不能确保该对象实际上是哪个类的实例，这个方法也是很有用的。

- 还可以调用 `Type` 类的静态方法 `GetType()`：

```
Type t = Type.GetType("System.Double");
```

`Type` 是许多反射技术的入口。它执行许多方法和属性，这里不可能列出所有的方法和属性，而主要介绍如何使用这个类。注意，可用的属性都是只读的：可以使用 `Type` 确定数据的类型，但不能使用它修改该类型！

1. `Type` 的属性

由 `Type` 执行的属性可以分为下述 3 类：

- 有许多属性都可以获取包含与类相关的各种名称的字符串，如表 13-1 所示。

表 13-1

属 性	返 回 值
Name	数据类型名
FullName	数据类型的完全限定名(包括命名空间名)
Namespace	定义数据类型的命名空间名

- 属性还可以进一步获取 `Type` 对象的引用，这些引用表示相关的类，如表 13-2 所示。

表 13-2

属 性	返回对应的 <code>Type</code> 引用
BaseType	这个 <code>Type</code> 的直接基本类型
UnderlyingSystemType	这个 <code>Type</code> 在 .NET 运行库中映射的类型 (某些.NET 基类实际上映射由 IL 识别的特定预定义类型)

- 许多 `Boolean` 属性表示这个类型是一个类、还是一个枚举等。这些属性包括 `IsAbstract`、`IsArray`、`IsClass`、`IsEnum`、`IsInterface`、`IsPointer`、`IsPrimitive`(一种预定义的基本数据类型)、`IsPublic`、`IsSealed` 和 `IsValueType`

例如，使用一个基本数据类型：

```
Type intType = typeof(int);
Console.WriteLine(intType.IsAbstract);      // writes false
Console.WriteLine(intType.IsClass);         // writes false
Console.WriteLine(intType.IsEnum);          // writes false
Console.WriteLine(intType.IsPrimitive);     // writes true
Console.WriteLine(intType.IsValueType);     // writes true
```

或者使用 `Vector` 类：

```
Type intType = typeof(Vector);
Console.WriteLine(intType.IsAbstract);      // writes false
Console.WriteLine(intType.IsClass);         // writes true
Console.WriteLine(intType.IsEnum);          // writes false
```



```
Console.WriteLine(intType.IsPrimitive); // writes false
Console.WriteLine(intType.IsValueType); // writes false
```

也可以获取定义类型的程序集的引用，该引用作为 `System.Reflection.Assembly` 类实例的一个引用来返回：

```
Type t = typeof (Vector);
Assembly containingAssembly = new Assembly(t);
```

2. 方法

`System.Type` 的大多数方法都用于获取对应数据类型的成员信息：构造函数、属性、方法和事件等。它有许多方法，但它们都有相同的模式。例如，有两个方法可以获取数据类型的方法信息：`GetMethod()`和 `GetMethods()`。`GetMethod()`方法返回 `System.Reflection. MethodInfo` 对象的一个引用，其中包含一个方法的信息。`GetMethods()`返回这种引用的一个数组。其区别是 `GetMethods()`返回所有方法的信息，而 `GetMethod()`返回一个方法的信息，其中该方法包含特定的参数列表。这两个方法都有重载方法，该重载方法有一个附加的参数，即 `BindingFlags` 枚举值，表示应返回哪些成员，例如，返回公有成员、实例成员和静态成员等。

例如，`GetMethods()`最简单的一个重载方法不带参数，返回数据类型所有公共方法的信息：

```
Type t = typeof(double);
MethodInfo [] methods = t.GetMethods();
foreach (MethodInfo nextMethod in methods)
{
    // etc.
}
```

`Type` 的成员方法如表 13-3 所示遵循同一个模式。

表 13-3

返回的对象类型	方法(名称为复数形式的方法返回一个数组)
ConstructorInfo	GetConstructor(), GetConstructors()
EventInfo	GetEvent(), GetEvents()
FieldInfo	GetField(), GetFields()
InterfaceInfo	GetInterface(), GetInterfaces()
MemberInfo	GetMember(), GetMembers()
MethodInfo	GetMethod(), GetMethods()
PropertyInfo	GetProperty(), GetProperties()

`GetMember()`和 `GetMembers()`方法返回数据类型的一个或所有成员的信息，这些成员可以是构造函数、属性和方法等。最后要注意，可以调用这些成员，其方式是调用 `Type` 的 `InvokeMember()`方法，或者调用 `MethodInfo`, `PropertyInfo` 和其他类的 `Invoke()`方法。



## 13.2.2 TypeView 示例

下面用一个短小的示例 TypeView 来说明 Type 类的一些功能, 这个示例可以列出数据类型的所有成员。本例中主要介绍 double 型的 TypeView 用法, 也可以修改该样列中的一行代码, 使用其他的数据类型。TypeView 提供的信息要比在控制台窗口中显示的信息多得多, 所以我们将打破常规, 在一个消息框中显示这些信息。运行 double 型的 TypeView 示例, 结果如图 13-1 所示。

该消息框显示了数据类型的名称、全名和命名空间, 以及底层类型和基类的名称。然后迭代该数据类型的所有公有实例成员, 显示所声明类型的每个成员、成员的类型(方法、字段等)以及成员的名称。声明类型是实际声明类型成员类名(换言之, 如果在 System.Double 中定义或重载, 该声明类型就是 System.Double, 如果成员继承了某个基类, 该声明类就是相关基类的名称)。

TypeView 不会显示方法的签名, 因为我们是通过 MemberInfo 对象获取所有公有实例成员的信息, 参数信息不能通过 MemberInfo 对象来获得。为了获取该信息, 需要引用 MemberInfo 和其他更特殊的对象, 即需要分别获取每一个成员类型的信息。

TypeView 会显示所有公有实例成员的信息, 但对于 double 来说, 仅定义了字段和方法。把 TypeView 编译为一个控制台应用程序, 可以在控制台应用程序中显示消息框。但是, 使用消息框就意味着需要引用基类程序集 System.Windows.Forms.dll, 它包含 System.Windows.Forms 命名空间中的类, 在这个命名空间中, 定义了我们需要的 MessageBox 类。下面列出 TypeView 的代码。开始时需要添加两条 using 语句:

```
using System;
using System.Text;
using System.Windows.Forms;
using System.Reflection;
```

需要 System.Text 的原因是我们使用 StringBuilder 对象建立在消息框中显示的文本, 以及消息框本身的 System.Windows.Forms。全部代码都放在类 MainClass 中, 这个类包含两个静态方法和一个静态字段, StringBuilder 的一个实例叫作 OutputText, 用于创建在消息框中显示的文本。Main 方法和类的声明如下所示:

```
class MainClass
{
    Static StringBuilder OutputText = new StringBuilder();

    static void Main()
    {
        // modify this line to retrieve details of any
        // other data type
        Type t = typeof(double);
```

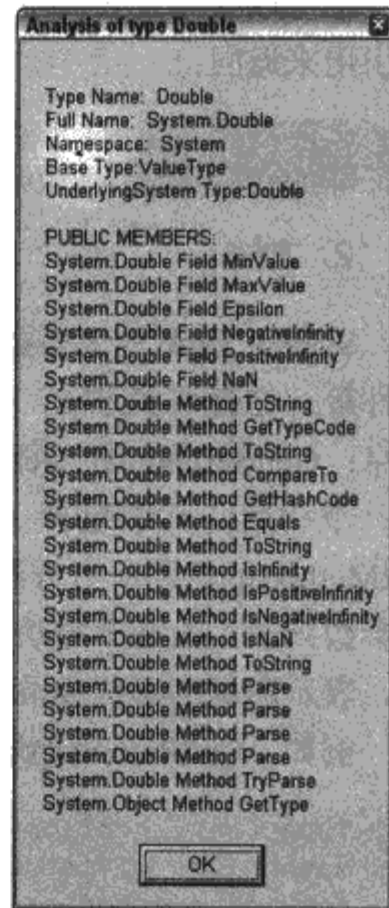


图 13-1

```

        AnalyzeType(t);
        MessageBox.Show(OutputText.ToString(), "Analysis of type "
                        + t.Name);
        Console.ReadLine();
    }

```

Main()方法首先声明一个 Type 对象,表示我们选择的数据类型,再调用方法 AnalyzeType(),从 Type 对象中提取信息,并使用该信息建立输出文本。最后在消息框中显示输出。使用 MessageBox 类是非常直观的:只需调用其静态方法 Show(),给它传递两个字符串,分别为消息框中的文本和标题。这些都由 AnalyzeType()来完成:

```

static void AnalyzeType(Type t)
{
    AddToOutput("Type Name: " + t.Name);
    AddToOutput("Full Name: " + t.FullName);
    AddToOutput("Namespace: " + t.Namespace);

    Type tBase = t.BaseType;

    if (tBase != null)
    {
        AddToOutput("Base Type:" + tBase.Name);
    }

    Type tUnderlyingSystem = t.UnderlyingSystemType;

    if (tUnderlyingSystem != null)
    {
        AddToOutput("UnderlyingSystem Type:" + tUnderlyingSystem.Name);
    }

    AddToOutput("\nPUBLIC MEMBERS:");
    MemberInfo [] Members = t.GetMembers();

    foreach (MemberInfo NextMember in Members)
    {
        AddToOutput(NextMember.DeclaringType + " " +
                    NextMember.MemberType + " " + NextMember.Name);
    }
}

```

执行这个方法,仅需调用 Type 对象的各种属性,就可以获得我们需要的类型名称的信息,再调用 GetMembers()方法,获得一个 MemberInfo 对象数组,该数组用于显示每个成员的信息。注意这里使用了一个辅助方法 AddToOutput(),该方法创建要在消息框中显示的文本:

```

static void AddToOutput(string Text)
{
    OutputText.Append("\n" + Text);
}

```

使用下面的命令编译 TypeView 程序集:

```
csc /reference:System.Windows.Forms.dll TypeView.cs
```

### 13.2.3 Assembly 类

Assembly 类是在 System.Reflection 命名空间中定义的，它允许访问给定程序集的元数据，它也包含可以加载和执行程序集(假定该程序集是可执行的)的方法。与 Type 类一样，Assembly 类包含非常多的方法和属性，这里不可能逐一论述。下面仅介绍完成示例 WhatsNewAttributes 所需要的方法和属性。

在使用 Assembly 实例做一些工作前，需要把相应的程序集加载到运行进程中。为此，可以使用静态成员 Assembly.Load()或 Assembly.LoadFrom()。这两个方法的区别是 Load()的参数是程序集的名称，运行库会在各个位置上搜索该程序集，这些位置包括本地目录和全局程序集高速缓存。而 LoadFrom()的参数是程序集的完整路径名，不会在其他位置搜索该程序集：

```
Assembly assembly1 = Assembly.Load("SomeAssembly");
Assembly assembly2 = Assembly.LoadFrom
(@"C:\My Projects\Software\SomeOtherAssembly");
```

这两个方法都有许多其他重载，它们提供了其他安全信息。加载了一个程序集后，就可以使用它的各种属性，例如查找它的全名：

```
string name = assembly1.FullName;
```

#### 1. 查找在程序集中定义的类型

Assembly 类的一个特性是可以获得在相应程序集中定义的所有类型的信息，只要调用 Assembly.GetTypes()方法，就可以返回一个包含所有类型信息的 System.Type 引用数组，然后就可以按照上一节的方式处理这些 Type 引用了：

```
Type[] types = theAssembly.GetTypes();
foreach (Type definedType in types)
{
    DoSomethingWith(definedType);
}
```

#### 2. 查找定制特性

用于查找在程序集或类型中定义了什么定制特性的方法取决于与该特性相关的对象类型。如果要确定程序集关联了什么定制特性，就需要调用 Attribute 类的一个静态方法 GetCustomAttributes()，给它传递程序集的引用：

```
Attribute [] definedAttributes =
    Attribute.GetCustomAttributes(assembly1);
// assembly1 is an Assembly object
```

#### 注意：

这是相当重要的。以前您可能想知道，在定义定制特性时，必须为它们编写类，为什么 Microsoft 没有更简单的语法。答案就在于此。定制特性与对象一样，加载了程序集后，就可以读取这些特性对象，查看它们的属性，调用它们的方法。

GetCustomAttributes()用于获取程序集的特性，它有两个重载方法：如果在调用它时，除了程序集的引用外，没有指定其他参数，该方法就会返回为这个程序集定义的所有定制特性。当然，



也可以通过指定第二个参数来调用它，第二个参数是表示特性类的一个 `Type` 对象，在这种情况下，`GetCustomAttributes()`就返回一个数组，该数组包含该特性类的所有特性。

注意，所有的特性都作为一般的 `Attribute` 引用来获取。如果要调用为定制特性定义的任何方法或属性，就需要把这些引用显式转换为相关的定制特性类。调用 `Assembly.GetCustomAttributes()` 的另一个重载方法，可以获得与给定数据类型相关的定制特性信息，这次传递的是一个 `Type` 引用，它描述了要获取的任何相关特性的类型。另一方面，如果要获得与方法、构造函数和字段等相关的特性，就需要调用 `GetCustomAttributes()` 方法，该方法是类 `MethodInfo`、`ConstructorInfo` 和 `FieldInfo` 等的一个成员。

如果只需要给定类型的一个特性，就可以调用 `GetCustomAttribute()` 方法，它返回一个 `Attribute` 对象。在 `WhatsNewAttributes` 示例中使用 `GetCustomAttribute()` 方法，是为了确定程序集中是否有特性 `SupportsWhatsNew`。为此，调用 `GetCustomAttributes()`，传递对 `WhatsNewAttributes` 程序集的一个引用和 `SupportWhatsNewAttribute` 特性的类型。如果有这个特性，就返回一个 `Attribute` 实例。如果在程序集中没有定义任何实例，就返回 `null`。如果找到两个或多个实例，`GetCustomAttribute()` 方法就抛出一个异常 `System.Reflection.AmbiguousMatchException`：

```
Attribute supportsAttribute =
    Attribute.GetCustomAttributes(assembly1,
        typeof(SupportsWhatsNewAttribute));
```

#### 13.2.4 完成 WhatsNewAttributes 示例

现在已经有足够的知识来完成 `WhatsNewAttributes` 示例了。为该示例中的最后一个程序集 `LookUpWhatsNew` 编写源代码，这部分应用程序是一个控制台应用程序，它需要引用其他两个程序集 `WhatsNewAttributes` 和 `VectorClass`。这是一个命令行应用程序，但仍可以象前面的 `TypeView` 示例那样在消息框中显示结果，因为结果是许多文本，所以不能显示在一个控制台窗口屏幕上。

这个文件的名称为 `LookUpWhatsNew.cs`，编译它的命令是：

```
csc /reference:WhatsNewAttributes.dll /reference:VectorClass.dll LookUpWhatsNew.cs
```

在这个文件的源代码中，首先指定要使用的命名空间 `System.Text`，因为需要使用一个 `StringBuilder` 对象：

```
using System;
using System.Reflection;
using System.Windows.Forms;
using System.Text;
using Wrox.ProCSharp.VectorClass;
using Wrox.ProCSharp.WhatsNewAttributes;

namespace Wrox.ProCSharp.LookUpWhatsNew
{
```

类 `WhatsNewChecker` 包含主程序入口和其他方法。我们定义的所有方法都在这个类中，它还有两个静态字段：`outputText` 和 `backDateTo`。`outputText` 包含在准备阶段创建的文本，这个文

本要写到消息框中，`backDateTo` 存储了选择的日期——自从该日期以来的所有修改都要显示出来。一般情况下，需要显示一个对话框，让用户选择这个日期，但我们不想编写这段代码，以免转移读者的注意力。因此，把 `backDateTo` 硬编码为日期 2008 年 2 月 1 日。在下载这段代码时，很容易修改这个日期：

```
class WhatsNewChecker
{
    static StringBuilder outputText = new StringBuilder(1000);
    static DateTime backDateTo = new DateTime(2008, 2, 1);

    static void Main()
    {
        Assembly theAssembly = Assembly.Load("VectorClass");
        Attribute supportsAttribute =
            Attribute.GetCustomAttribute(
                theAssembly, typeof(SupportsWhatsNewAttribute));
        string Name = theAssembly.FullName;

        AddToMessage("Assembly: " + Name);
        if (supportsAttribute == null)
        {
            AddToMessage("This assembly does not support WhatsNew attributes");
            return;
        }
        else
            AddToMessage("Defined Types:");

        Type[] types = theAssembly.GetTypes();
        foreach (Type definedType in types)
            DisplayTypeInfo(theAssembly, definedType);

        MessageBox.Show(outputText.ToString(),
            "What's New since " + backDateTo.ToLongDateString());
        Console.ReadLine();
    }
}
```

`Main()` 方法首先加载 `VectorClass` 程序集，验证它是否真的用 `SupportsWhatsNew` 特性来标记。我们知道，`VectorClass` 应用了 `SupportsWhatsNew` 特性，虽然才编译了该程序集，但这种检查还是必要的，因为用户可能希望检查这个程序集。

验证了这个程序集后，使用 `Assembly.GetTypes()` 方法获得一个数组，其中包括在该程序集中定义的所有类型，然后在这个数组中迭代。对每种类型调用一个方法 `DisplayTypeInfo()`，给 `outputText` 字段添加相关的文本，包括 `LastModifiedAttribute` 实例的信息。最后，显示带有完整文本的消息框。`DisplayTypeInfo()` 方法如下所示：

```
static void DisplayTypeInfo(Assembly theAssembly, Type type)
{
    // make sure we only pick out classes
    if (!(type.IsClass))
    {
        return;
    }
    AddToMessage("\n\n" + type.Name);

    Attribute [] attribs = Attribute.GetCustomAttributes(type);
    if (attribs.Length == 0)
```



```

    {
        AddToMessage("No changes to this class\n");
    }
    else
    {
        foreach (Attribute attrib in attribs)
        {
            WriteAttributeInfo(attrib);
        }
    }
    MethodInfo [] methods = type.GetMethods();
    AddToMessage("CHANGES TO METHODS OF THIS CLASS:");
    foreach (MethodInfo nextMethod in methods)
    {
        object [] attribs2 =
            nextMethod.GetCustomAttributes(
                typeof(LastModifiedAttribute), false);
        if (attribs2 != null)
        {
            AddToMessage(
                nextMethod.ReturnType + " " + nextMethod.Name + "()\n");
            foreach (Attribute nextAttrib in attribs2)
            {
                WriteAttributeInfo(nextAttrib);
            }
        }
    }
}

```

注意, 在这个方法中, 首先应检查参数 `Type` 引用是否表示一个类。为了简化代码, 指定 `LastModified` 特性只能应用于类或成员方法, 如果该引用不是类(它可能是一个结构、委托或枚举), 进行任何处理都是浪费时间。

接着使用 `Attribute.GetCustomAttributes()` 方法确定这个类是否有相关的 `LastModifiedAttribute` 实例。如果有, 就使用辅助方法 `WriteAttributeInfo()` 把它们的信息添加到输出文本中。

最后使用 `Type.GetMethods()` 方法迭代这个数据类型的所有成员方法, 然后对类的每个方法进行相同的处理: 检查每个方法是否有相关的 `LastModifiedAttribute` 实例, 如果有, 用 `WriteAttributeInfo()` 方法显示它们。

下面的代码显示了 `WriteAttributeInfo()` 方法, 它负责确定为给定的 `LastModifiedAttribute` 实例显示什么文本, 注意这个方法的参数是一个 `Attribute` 引用, 所以需要先把该引用转换为 `LastModifiedAttribute` 引用。之后, 就可以使用最初为这个特性定义的属性获取其参数。在把该特性添加到要显示的文本中之前, 应检查特性的日期是否是最近的:

```

static void WriteAttributeInfo(Attribute attrib)
{
    LastModifiedAttribute lastModifiedAttrib =
        attrib as LastModifiedAttribute;
    if (lastModifiedAttrib == null)
    {
        return;
    }

    // check that date is in range
    DateTime modifiedDate = lastModifiedAttrib.DateModified;

    if (modifiedDate < backDateTo)
    {

```

```

        return;
    }

    AddToMessage("  MODIFIED: " +
        modifiedDate.ToLongDateString() + ":" );
    AddToMessage("    " + lastModifiedAttrib.Changes);

    if (lastModifiedAttrib.Issues != null)
    {
        AddToMessage("    Outstanding issues:" +
            lastModifiedAttrib.Issues);
    }
}

```

最后，是辅助方法 AddToMessage():

```

static void AddToMessage(string message)
{
    outputText.Append("\n" + message);
}
}

```

运行这段代码，得到如图 13-2 所示的结果。

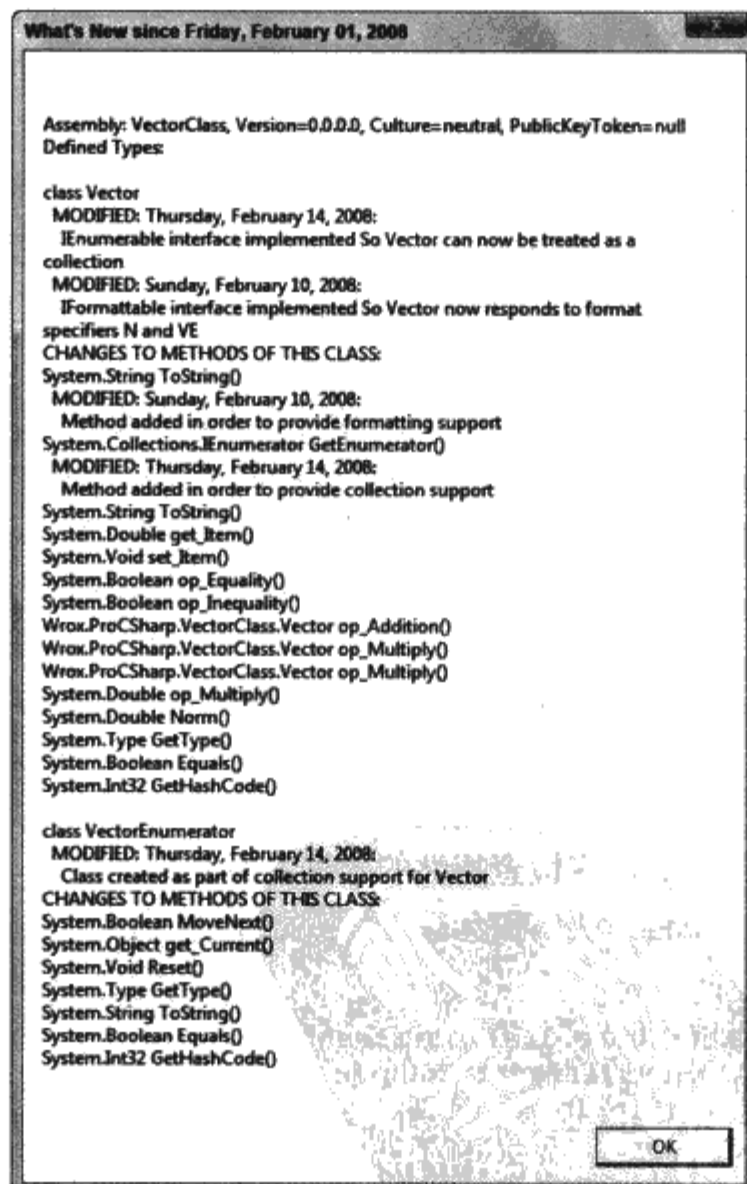


图 13-2

注意，在列出 VectorClass 程序集中定义的类型时，实际上选择了两个类：Vector 和内嵌的 VectorEnumerator 类。还要注意，这段代码把 backDateTo 日期硬编码为 2 月 1 日，实际上选择

的是日期为 2 月 14 日的特性(添加集合支持的时间), 而不是 1 月 14 日(添加 IFormattable 接口的时间)。

### 13.3 小结

本章没有介绍反射的全部内容, 反射需要一整本书来讨论。我们只介绍了 Type 和 Assembly 类, 它们是访问反射所提供的扩展功能的主要入口。

另外, 本章还探讨了反射的一个常用方面: 定制特性。介绍了如何定义和应用自己的定制特性, 以及如何在运行期间检索定制属性的信息。

第 14 章介绍异常和结构化的异常处理。

# 第14章

## 错误和异常

错误的出现并不总是编写应用程序的人的原因，有时应用程序会因为终端用户的操作或运行代码的环境而发生错误。无论如何，我们都应预测应用程序和代码中出现的错误。

.NET Framework 改进了处理错误的方式。C#处理错误的机制可以为每种错误提供定制的处理，并把识别错误的代码与处理错误的代码分离开来。

本章的主要内容如下：

- 异常类
- 使用 try-catch-finally 捕获异常
- 创建用户定义的异常

学习完本章后，您将很好地掌握 C#应用程序中的高级异常处理技术。

无论编码技术有多好，程序都必须能处理可能出现的错误。例如，在一些复杂的处理过程中，代码没有读取文件的许可，或者在发送网络请求时，网络可能会中断。在这种情况下，方法只返回相应的错误代码通常是不够的——可能方法调用嵌套了 15 级或者 20 级，此时，代码需要跳过所有的 15 或 20 级方法调用，才能完全退出任务，采取相应的措施。C#语言提供了处理这种情形的绝佳工具，称为异常处理机制。

**注意：**

在 Visual Basic 6 中，错误处理工具的功能非常有限，主要是 On Error GoTo 语句。如果您有 Visual Basic 6 的背景知识，就会发现 C#异常打开了程序中处理错误的全新世界的大门。另一方面，Java 和 C++开发人员会比较熟悉异常的规则，因为这些语言处理错误的方式与 C#相同。C++开发人员会留意异常是因为 C++可能会因此而降低性能，但在 C#中就不是这样。在 C#代码中使用异常一般不影响性能。Visual Basic 开发人员会发现，在 C#中处理异常非常类似于在 Visual Basic 中使用异常(但语法不同)。

### 14.1 异常类

在 C#中，当出现某个异常错误条件时，就会创建一个异常对象。这个对象包含有助于跟踪问题的信息。我们可以创建自己的异常类(详见后面的内容)，但.NET 提供了许多预定义的异常类。

本节将快速总结.NET 基类库中可以使用的一些异常。Microsoft 在.NET 中定义了大量的异常类，这里不可能提供详尽的列表。图 14-1 所示的类结构图显示了其中的一些类，给出了大致的模式。

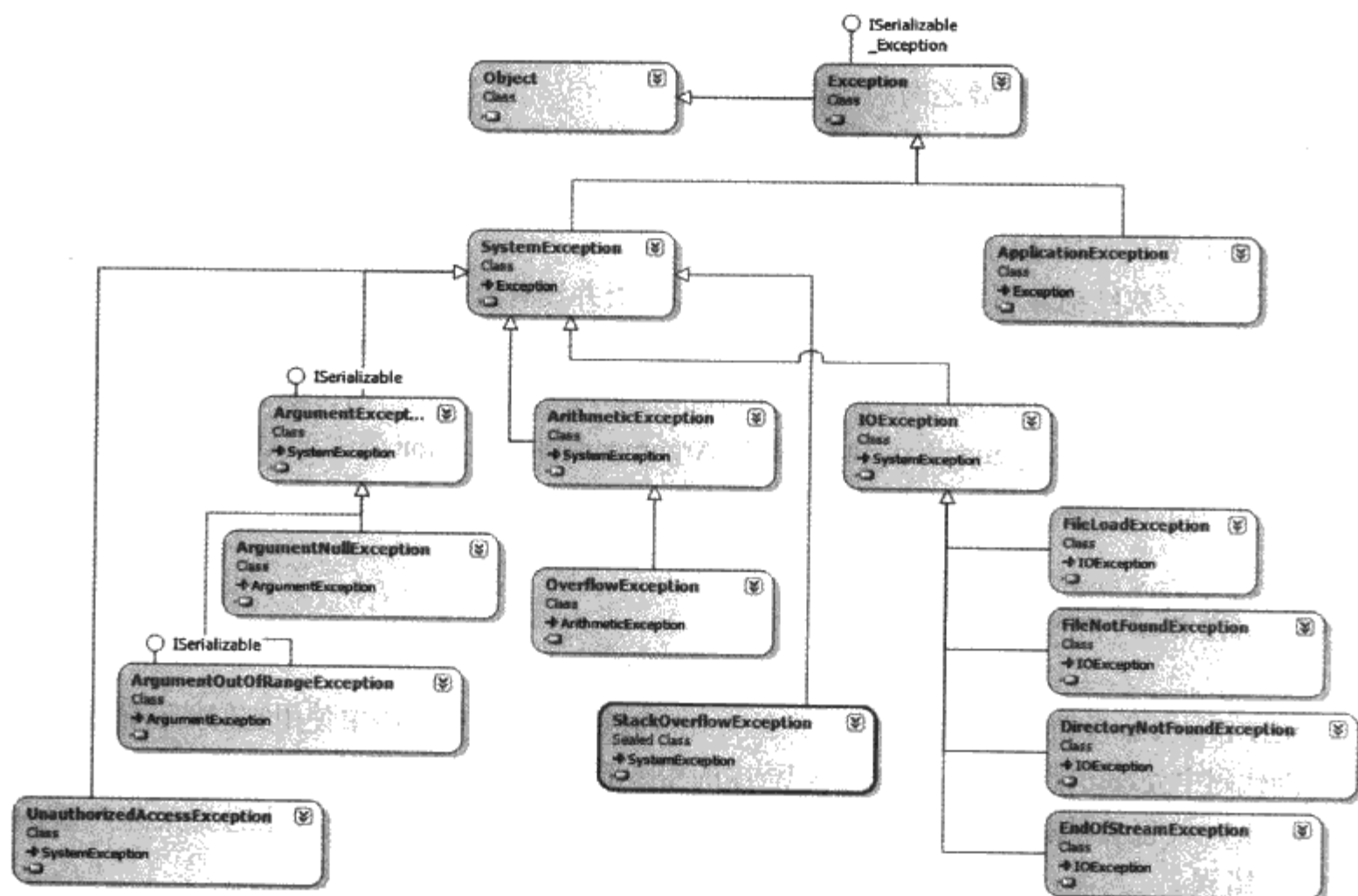


图 14-1

这个图中的所有类都在 `System` 命名空间中，但 `IOException` 和派生于 `IOException` 的类除外，它们在 `System.IO` 命名空间中，这个命名空间处理文件数据的读写。一般情况下，异常没有特定的命名空间，异常类应放在生成异常的类所在的命名空间中，因此与 IO 相关的异常就在 `System.IO` 命名空间中。许多基类命名空间中都有异常类。

对于 .NET 类来说，一般的异常类 `System.Exception` 派生于 `System.Object`，通常不在代码中抛出这个 `System.Exception` 对象，因为它无法确定错误情况的本质。

在该层次结构中有两个重要的类，它们派生于 `System.Exception`：

- `System.SystemException`——通常由 .NET 运行库生成，或者有着非常一般的本质、可以由几乎所有的应用程序生成。例如，如果 .NET 运行库检测到堆栈已满，就会抛出 `StackOverflowException`。另一方面，如果检测到调用方法时参数不正确，可以在自己的代码中选择抛出 `ArgumentException` 或其子类。`System.SystemException` 的子类包括表示致命错误和非致命错误的异常。
- `System.ApplicationException`——这个类非常重要，因为它是第三方定义的异常基类。如果自己定义的异常覆盖了应用程序独有的错误情况，就应使它们直接或间接派生于 `System.ApplicationException`。

其他可能用到的异常类包括：

- `StackOverflowException`——如果分配给堆栈的内存区域已满，就会抛出这个异常。如果一个方法连续地递归调用它自己，就可能发生堆栈溢出。这一般是一个致命错误，因为它禁止应用程序执行除了中断以外的其他任务。在这种情况下，甚至也不可能执行 `finally` 块，通常用户自己不能处理像这样的错误，而应退出应用程序。



- `EndOfStreamException`——这个异常通常是因为读到文件末尾而抛出的。第 41 章将解释流，流表示数据源之间的数据流。
- `OverflowException`——如果要在 `checked` 环境下把包含值 - 40 的 `int` 类型数据转换为 `uint` 数据，就会抛出这个异常。

我们不打算讨论图 14-1 中的所有其他异常类。

异常类层次结构并不多见，因为其中的大多数类并没有给它们的基类添加任何功能。但是在异常处理时，添加继承类的一般原因是更准确地指定错误，所以不需要重写方法或添加新方法(但常常要添加额外的属性，以包含有关错误情况的额外信息)。例如，当传递了不正确的参数值时，可给方法调用使用 `ArgumentException` 基类，`ArgumentNullException` 派生于 `ArgumentException` 类，它专门用于传送参数值是 `Null` 的情况。

## 14.2 捕获异常

.NET Framework 提供了大量的预定义基类异常对象，该如何在代码中使用它们捕获错误？为了在 C# 代码中处理可能的错误，一般要把程序的相关部分分成 3 种不同类型的代码块：

- `try` 块包含的代码组成了程序的正常操作部分，但这部分程序可能遇到某些严重的错误。
- `catch` 块包含的代码处理各种错误，这些错误是 `try` 块中的代码执行时遇到的。这个块还可以用于记录错误。
- `finally` 块包含的代码清理资源或执行要在 `try` 块或 `catch` 块末尾执行的其他操作。无论是否产生异常，都会执行 `finally` 块，理解这一点是非常重要的。因为 `finally` 块包含了应总是执行的清理代码，如果在 `finally` 块中放置了 `return` 语句，编译器就会标记一个错误。例如，可以在 `finally` 块中关闭在 `try` 块中打开的连接。`finally` 块是可选的。不需要清理代码(例如删除对象或关闭已打开的对象)，就不需要包含此块。

那么，这些块是如何组合在一起捕获错误的？下面就是其步骤：

(1) 程序流进入 `try` 块。

(2) 如果没有错误发生，就会正常执行操作。当程序流离开 `try` 块后，即使什么也没有发生，也会自动进入 `finally` 块(第(5)步)。但如果在 `try` 块中程序流检测到一个错误，程序流就会跳转到 `catch` 块(下一步)。

(3) 在 `catch` 块中处理错误。

(4) 在 `catch` 块执行完后，程序流会自动进入 `finally` 块。

(5) 执行 `finally` 块。

用于完成这些任务的 C# 语法如下所示：

```
try
{
    // code for normal execution
}
catch
{
    // error handling
}
```

```
finally
{
    // clean up
}
```

实际上，上面的代码还有几种变体：

- 可以省略 `finally` 块
- 可以提供任意多个 `catch` 块，处理不同类型的错误。但不应包含过多的 `catch` 块，以防降低应用程序的性能。
- 可以省略 `catch` 块——此时，该语法应不是标识异常，而是一种确保程序流在离开 `try` 块后执行 `finally` 块中的代码的方式，如果在 `try` 块中有几个出口，这是很有用的。

这看起来很不错，实际上是有问题的。如果执行 `try` 块中的代码，则程序流如何在错误发生时切换到 `catch` 块上？如果检测到一个错误，代码就执行一定的操作，称为“抛出一个异常”；换言之，它实例化一个异常对象，并抛出这个异常：

```
throw new OverflowException();
```

这里实例化了 `OverflowException` 类的一个异常对象。只要计算机在 `try` 块中遇到一个 `throw` 语句，就会立即查找与这个 `try` 块对应的 `catch` 块。如果有多个与 `try` 块对应的 `catch` 块，计算机就会查找与 `catch` 块对应的异常类，确定正确的 `catch` 块。例如，当抛出一个 `OverflowException` 对象时，执行流就会跳转到下面的 `catch` 块上：

```
catch (OverflowException e)
{
    // exception handling here
}
```

换言之，计算机查找的 `catch` 块应表示同一个类(或基类)中匹配的异常类实例。

有了这些额外的信息，就可以扩展刚才介绍的 `try` 块。为了讨论方便，假定可能在 `try` 块中发生两个严重错误：溢出和数组超出范围。假定代码包含两个布尔变量 `Overflow` 和 `OutOfBounds`，表示这两种错误情况是否存在。我们知道，存在表示溢出的预定义溢出类 `OverflowException`，同样，存在预定义的 `IndexOutOfRangeException` 类，用于处理数组超出范围错误。

现在，`try` 块如下所示：

```
try
{
    // code for normal execution

    if (Overflow == true)
        throw new OverflowException();

    // more processing

    if (OutOfBounds == true)
        throw new IndexOutOfRangeException();

    // otherwise continue normal execution
}
catch (OverflowException ex)
{
}
```

```

    // error handling for the overflow error condition
}
catch (IndexOutOfRangeException ex)
{
    // error handling for the index out of range error condition
}
finally
{
    // clean up
}

```

我们得到的 `try` 块看起来并不比 Visual Basic 6 的 `On Error GoTo` 语句强多少，但可以更清晰地将不同的代码段分开。实际上，C# 的错误处理有一个更强大、更灵活的机制。

这是因为 `throw` 语句可以嵌套在 `try` 块的几个方法调用中，甚至在程序流进入其他方法时，也会继续执行同一个 `try` 块。如果计算机遇到一个 `throw` 语句，就会立即退出堆栈中所有的方法调用，查找 `try` 块的结尾和合适的 `catch` 块的开头，此时，中间方法调用中的所有局部变量都会出作用域。`try...catch` 结构最适合于本节开头描述的情况：错误发生在一个方法调用中，而该方法调用可能嵌套了 15 到 20 级，这些处理操作会立即停止。

从上面的论述可以看出，`try` 块在控制程序的执行流上有着重要的作用。但是，异常类是用于处理异常情况的，这是其名称的由来。不应该用异常来控制退出 `do...while` 循环的时间。

### 14.2.1 执行多个 `catch` 块

要了解 `try...catch...finally` 块是如何工作的，最简单的方式是用两个示例来说明。第一个示例是 `SimpleException`。它多次要求用户键入一个数字，然后显示这个数字。为了便于解释这个示例，假定该数字必须在 0 和 5 之间，否则程序就不能对该数字进行正确的处理。所以，如果用户键入超出该范围的数字，程序就抛出一个异常。

程序会继续要求用户输入数字，直到用户不再输入任何内容，但按下了回车键为止。

#### 注意：

这段代码没有说明何时使用异常处理。前面已经提及，异常是用于处理异常情况的。用户总是键入一些无聊的东西，所以这种情况不会真正发生。正常情况下，程序会处理不正确的用户输入，进行即时检查，如果有问题，就要求用户重新键入。但是，在一个要求几分钟内读懂的小示例中生成异常是比较困难的，为了描述异常是如何工作的，后面将使用更真实的示例。

`SimpleExceptions` 的代码如下所示：

```

using System;

namespace Wrox.ProCSharp.AdvancedCSharp
{
    public class MainEntryPoint
    {
        public static void Main()
        {
            string userInput;
            while ( true )
            {
                try

```

```

{
    Console.WriteLine("Input a number between 0 and 5 " +
        "(or just hit return to exit)> ");
    userInput = Console.ReadLine();

    if (userInput == "")
        break;
    }
    int index = Convert.ToInt32(userInput);

    if (index < 0 || index > 5)
        throw new IndexOutOfRangeException(
            "You typed in " + userInput);
    }

    Console.WriteLine("Your number was " + index);
}
catch (IndexOutOfRangeException ex)
{
    Console.WriteLine("Exception: " +
        "Number should be between 0 and 5. " + ex.Message);
}
catch (Exception ex)
{
    Console.WriteLine(
        "An exception was thrown. Message was:{0} " + ex.Message);
}
catch
{
    Console.WriteLine("Some other exception has occurred");
}
finally
{
    Console.WriteLine("Thank you");
}
}
}
}
}

```

这段代码的核心是一个 `while` 循环，它连续使用 `Console.ReadLine()` 以请求用户输入。`ReadLine()` 返回一个字符串，所以程序首先要用 `System.Convert.ToInt32()` 方法把它转换为 `int` 型。`System.Convert` 类包含执行数据转换的各种有用的方法，并提供了 `int.Parse()` 方法的一个替代方法。一般情况下，`System.Convert` 包含执行各种类型转换的方法，C# 编译器把 `int` 解析为 `System.Int32` 基类的实例。

#### 注意：

传递给 `catch` 块的参数只能用于该 `catch` 块。这就是为什么在上面的代码中，能在后续的 `catch` 块中使用相同的参数名 `ex` 的原因。

在上面的代码中，我们也检查一个空字符串，因为该空字符串是退出 `while` 循环的条件。注意这里用 `break` 语句退出 `try` 块和 `while` 循环——这是有效的。当然，当程序流退出 `try` 块时，会执行 `finally` 块中的 `Console.WriteLine()` 语句。尽管这里仅显示一句问候，仍需要关闭文件句柄，调用各种对象的 `Dispose()` 方法，以执行清理工作。一旦计算机退出了 `finally` 块，就会继续



执行下一个语句，如果没有 `finally` 块，该语句也会执行。在本例中，我们返回 `while` 循环的开头，再次进入 `try` 块(除非执行 `while` 循环中 `break` 语句的结果是进入 `finally` 块，此时就会退出 `while` 循环)。

下面看看异常情况：

```
if (index < 0 || index > 5)
{
    throw new IndexOutOfRangeException("You typed in " + userInput);
}
```

在抛出一个异常时，需要选择要抛出的异常类型。可以使用类 `System.Exception`，但这个类是一个基类，最好不要把这个类的实例当作一个异常，因为它没有包含错误的任何信息。`.NET Framework` 定义了许多派生于 `System.Exception` 的其他异常类，每个类都对应于一种类型的异常情况，也可以定义自己的异常类。在抛出一个匹配特定错误情况的类实例时，应提供尽可能多的异常信息。在本例中，`System.IndexOutOfRangeException` 是最佳选择。`IndexOutOfRangeException` 有几个构造函数重载，我们选择的一个重载，其参数是一个描述错误的字符串。另外，也可以选择派生自己的定制异常对象，描述该应用程序环境中的错误情况。

假定用户这次键入了一个不在 0 到 5 范围内的数字，`if` 语句就会检测到一个错误，并实例化和抛出一个 `IndexOutOfRangeException` 对象。计算机会立即退出 `try` 块，查找处理 `IndexOutOfRangeException` 的 `catch` 块。它遇到的第一个 `catch` 块如下所示：

```
catch (IndexOutOfRangeException ex)
{
    Console.WriteLine(
        "Exception: Number should be between 0 and 5." + ex.Message);
}
```

由于这个 `catch` 块带一个合适类的参数，它就会传送给异常实例，并执行。在本例中，是显示错误信息和 `Exception.Message` 属性(它对应于给 `IndexOutOfRangeException` 的构造函数传递的字符串)。执行了这个 `catch` 块后，控制就切换到 `finally` 块，就好像没有发生过任何异常。

注意，本例还提供了另一个 `catch` 块：

```
catch (Exception ex)
{
    Console.WriteLine("An exception was thrown. Message was: " + ex.Message);
}
```

如果没有在前面的 `catch` 块中捕获到这类异常，这个 `catch` 块也能处理 `IndexOutOfRangeException`。——基类的一个引用也可以指向派生于它的所有类实例，所有的异常都派生于 `System.Exception`。那么为什么不执行这个 `catch` 块？答案是计算机只执行它在 `catch` 块列表中的第一个合适的 `catch` 块。但为什么还要编写第二个 `catch` 块？不仅 `try` 块包含这段代码，还有另外 3 个方法调用 `Console.ReadLine()`、`Console.Write()` 和 `Convert.ToInt32()` 也包含这段代码，它们是 `System` 命名空间中的方法。这 3 个方法都可能抛出异常。

如果键入的内容不是数字，例如 `a` 或 `hello`，`Convert.ToInt32()` 方法就会抛出一个 `System.FormatException` 类的异常，表示传递给 `ToInt32()` 的字符串不能转换为 `int`。此时，计算机会跟踪这个方法调用，查找可以处理该异常的处理程序。第一个 `catch` 块带一个



`IndexOutOfRangeException`, 不能处理这种异常。计算机接着查看第二个 `catch` 块, 显然它可以处理这类异常, 因为 `FormatException` 派生于 `Exception`, 所以把 `FormatException` 实例作为参数传送给它。

示例的这种结构是非常典型的多 `catch` 块结构。最先编写的 `catch` 块用于处理非常特殊的错误情况, 接着是比较一般的块, 它们可以处理任何错误, 我们没有为它们编写特定的错误处理程序。实际上, `catch` 块的顺序是很重要的, 如果以相反的顺序编写这两个块, 代码就不会编译, 因为第二个 `catch` 块是不会执行的(`Exception` `catch` 块会捕获所有的异常)。因此, 最上面的 `catch` 块应用于最特殊的异常情况, 最后是最一般的 `catch` 块。

但是, 在上面的例子中还有第三个 `catch` 块:

```
catch
{
    Console.WriteLine("Some other exception has occurred");
}
```

这就是最一般的 `catch` 块——它不带参数, 原因是这个 `catch` 块处理的是其他没有用 C# 编写的代码(甚或根本不是托管代码)抛出的异常。在 C# 语言中, 只有派生于 `System.Exception` 的类实例, 才能作为异常来抛出。但其他语言没有这个限制, 例如, C++ 允许把任何变量作为异常来抛出。如果代码调用了用其他语言编写的库或程序集, 抛出的异常就可能不是派生于 `System.Exception`。但在许多情况下, .NET `PInvoke` 机制会捕获这些异常, 把它们转换为 .NET `Exception` 对象。但这个 `catch` 块做的工作并不多, 因为我们不知道该异常表示什么类。

**注意:**

在上面的示例中, 也可以不添加这个通用的 `catch` 处理程序, 如果要调用其他不支持 .NET、但可能抛出异常的库, 这么做是有很有效的。这个示例包含它, 主要是为了说明这个规则。

前面分析了示例的代码, 现在可以试着运行它。下面的输出说明了不同的输入会得到不同的结果, 并说明抛出了 `IndexOutOfRangeException` 和 `FormatException`:

#### **SimpleExceptions**

```
Input a number between 0 and 5 (or just hit return to exit)> 4
Your number was 4
Thank you
Input a number between 0 and 5 (or just hit return to exit)> 0
Your number was 0
Thank you
Input a number between 0 and 5 (or just hit return to exit)> 10
Exception: Number should be between 0 and 5. You typed in 10
Thank you
Input a number between 0 and 5 (or just hit return to exit)> hello
An exception was thrown. Message was: Input string was not in a correct format.
Thank you
Input a number between 0 and 5 (or just hit return to exit)>
Thank you
```

14.2.2 在其他代码中捕获异常

在上面的示例中，说明了两个异常的处理。一个异常是 `IndexOutOfRangeException`，它由我们自己的代码抛出，另一个异常是 `FormatException`，由一个基类抛出。如果检测到错误，或者某个方法因传递的参数有误而被错误调用，库中的代码就常常会抛出一个异常。但库中的代码很少捕获这样的异常。应由客户机代码来决定如何处理这些问题。

在调试时，异常经常从基类库中抛出，调试的过程在某种程度上是确定异常抛出的原因，并去除导致错误发生的缘由。主要目标是确保代码在执行后，异常只发生在非常罕见的情况下，如果可能，应在代码中以适当的方式处理它。

14.2.3 `System.Exception` 属性

在示例中，我们只使用了异常对象的一个属性 `Message`。在 `System.Exception` 中还有许多其他属性，如表 14-1 所示。

表 14-1

属 性	说 明
<code>Data</code>	这个属性可以给异常添加键/值语句，以提供异常的额外信息
<code>HelpLink</code>	链接到一个帮助文件上，以提供该异常的更多信息
<code>InnerException</code>	如果此异常是在 <code>catch</code> 块中抛出的，它就会包含把代码发送到 <code>catch</code> 块中的异常对象
<code>Message</code>	描述错误情况的文本
<code>Source</code>	导致异常的应用程序或对象名
<code>StackTrace</code>	堆栈上方法调用的信息，它有助于跟踪抛出异常的方法
<code>TargetSite</code>	描述抛出异常的方法的 .NET 反射对象

在这些属性中，如果可以进行堆栈跟踪，`StackTrace` 和 `TargetSite` 是由 .NET 运行库自动提供的。`Source` 总是由 .NET 运行库提供为产生异常的程序集名称(但可以在代码中修改该属性，提供更专门的信息)，`Data`、`Message`、`HelpLink` 和 `InnerException` 必须由抛出异常的代码提供，其方法是在抛出异常前设置这些属性。例如，抛出异常的代码如下所示：

```
if (ErrorCondition == true)
{
    Exception myException = new ClassMyException("Help!!!!");
    myException.Source = "My Application Name";
    myException.HelpLink = "MyHelpFile.txt";
    myException.Data["ErrorDate"] = DateTime.Now;
    myException.Data.Add("AdditionalInfo", "Contact Bill from the Blue Team");
    throw myException;
}
```

其中 `ClassMyException` 是抛出的异常类名。注意所有的异常类名通常以 `Exception` 结尾。`Data` 属性可以用两种方式设置。

### 14.2.4 没有处理异常时所发生的情况

有时生成了一个异常后，代码中没有 `catch` 块能处理这类异常。前面的 `SimpleExceptions` 示例就说明了这种情况。例如，假定忽略 `FormatException` 和通用的 `catch` 块，只有处理 `IndexOutOfRangeException` 的块。此时，如果抛出一个 `FormatException` 异常，会发生什么情况呢？

答案是 .NET 运行库会捕获它。在本节的后面将介绍如何嵌套 `try` 块——实际上在本示例中，就有一个在后台处理的嵌套 `try` 块。 .NET 运行库把整个程序放在另一个更大的 `try` 块中，每个 .NET 程序都会这么做。这个 `try` 块有一个 `catch` 处理程序，它可以捕获任何类型的异常。如果代码没有处理发生的异常，程序流就会退出程序，由 .NET 运行库中的 `catch` 块捕获它。但是，结果并不是你想像的那样。代码的执行会立即中断，并给用户显示一个对话框，说明代码没有处理异常，并给出 .NET 运行库能检索到的异常信息。至少异常会被捕获，这就是第 2 章在 `Vector` 示例程序抛出一个异常时发生的情况。

一般情况下，如果编写一个可执行程序，就应捕获尽可能多的异常，并以合理的方式处理它们。如果编写一个库，最好不要捕获异常(除非某个异常表示的是代码可以处理的情况)，但要假定调用代码可以处理它们。当然，用户可能不想捕获任何 Microsoft 预定义的异常，而是抛出自己的异常对象，给客户机代码提供更特定的信息。

### 14.2.5 嵌套的 `try` 块

异常的一个特性是 `try` 块可以嵌套，如下所示：

```
try
{
    // Point A
    try
    {
        // Point B
    }
    catch
    {
        // Point C
    }
    finally
    {
        // clean up
    }
    // Point D
}
catch
{
    // error handling
}
finally
{
    // clean up
}
```

在上面的代码中，每个 `try` 块都只有一个 `catch` 块，但可以把多个 `catch` 块连接在一起。下面详细讨论嵌套的 `try` 块如何工作。

如果抛出的异常在外层的 `try` 块中, 且在内层 `try` 块的外部(标记为 A 或 D 的代码块), 这种情况就与前面介绍的情况没有任何区别: 异常由外层的 `catch` 块捕获, 并执行外层的 `finally` 块, 或者执行 `finally` 块, 由 .NET 运行库处理异常。

如果异常是在内层 `try` 块(代码块 B)中抛出的, 且有一个合适的内层 `catch` 块处理该异常, 这又是我们熟悉的情况: 在内层处理异常, 执行内层的 `finally` 块, 之后继续执行外层的 `try` 块(标记为 D 的代码块)。

现在假定异常是在代码块的内层 `try` 块中抛出的, 但内层的 `catch` 块中没有合适的处理程序。这时通常就要执行内层的 `finally` 块, 但 .NET 运行库只能退出内层的 `try` 块, 才能搜索到合适的处理程序。下一个要搜索的区域显然是外层的 `catch` 块。如果系统在这里找到一个处理程序, 就会执行该处理程序, 再执行外层的 `finally` 块, 如果没有找到合适的处理程序, 就会继续搜索。在这里, 执行的是外层的 `finally` 块, 因为没有更多的 `catch` 块, 所以控制权会返回 .NET 运行库。注意, 不会执行外层 `try` 块中标记为 D 的代码。

如果异常是在 C 代码块中抛出的, 就更有意思了。如果程序执行到代码块 C 中, 就必须处理由代码块 B 抛出的异常。在 `catch` 块中抛出另一个异常是很正常的。此时, 异常的处理就跟它是在外层 `try` 块中抛出的一样, 程序流会立即退出内层的 `catch` 块, 执行内层的 `finally` 块, 在外层的 `catch` 中搜索处理程序。同样, 如果在内层的 `finally` 块中抛出一个异常, 搜索会在外层的 `catch` 块开始, 执行最匹配的处理程序。

注意:

在 `catch` 和 `finally` 块中抛出异常是合理的。

尽管本例只有两个 `try` 块, 但无论嵌套了多少个 `try` 块, 规则都是一样的。在每个块中, .NET 运行库顺序执行 `try` 块, 查找合适的处理程序, 在每个步骤中, 当退出 `catch` 块后, 就会执行对应 `finally` 块中的清理代码, 但不执行 `finally` 块外部的代码, 直到找到合适的 `catch` 处理程序, 并执行为止。

`Try` 块的嵌套也可能发生在方法之间。例如, 方法 A 调用了 `try` 块中的方法 B, 方法 B 又包含一个 `try` 块。

前面说明了嵌套 `try` 块的工作方式。下一个问题显然是为什么要这么做? 这有两个原因:

- 修改所抛出的异常的类型。
- 在代码的不同地方处理不同类型的异常。

### 1. 修改异常的类型

当最初抛出的异常不足以说明问题时, 修改异常的类型就非常重要了。通常的情况是抛出的异常(可能由 .NET 运行库抛出)是一种相当低级的异常, 说明发生溢出(`OverflowException`)或传递给方法的参数不正确(派生于 `ArgumentException` 的一个类)。但是, 由于抛出异常的环境, 我们知道这暴露了一些底层的问题(例如, 因为刚才读取的文件包含了不正确的数据, 才发生了溢出异常)。此时处理程序对于第一个异常所能做的最佳处理就是抛出另一个异常, 以便更准确地说明这个问题, 让另一个 `catch` 块更恰当地处理它。也可以通过一个由 `System.Exception` 执行的属性 `InnerException` 来处理最初的异常。`InnerException` 只包含另一个相关异常的引用——最终的处理例程需要这个额外的信息。



当然，还应指出，异常可能在 `catch` 块中抛出。例如，可以从某个配置文件中读取数据，这个文件包含处理错误的详细指令——结果可能是这个文件不存在。

## 2. 在不同的地方处理不同的异常

嵌套 `try` 块的第二个原因是不同类型的异常可以在代码的不同地方处理。例如，在循环中，可能会发生各种异常。其中一些异常比较严重，需要退出整个循环，而另外一些则不太严重，只需退出这次迭代，进入循环的下一次迭代即可。在循环的内部有一个 `try` 块就可以处理不太严重的异常，再在循环外面用一个外层的 `try` 块来处理比较严重的错误。在下面的异常示例中，将解释具体的操作情况。

## 14.3 用户定义的异常类

下面介绍有关异常的第二个示例，这个示例叫作 `SolicitColdCall`，包含了两个嵌套的 `try` 块，说明了如何定义定制的异常类，再从 `try` 块中抛出另一个异常。

这个示例假定一家销售公司希望有更多的客户。该公司的销售部门打算给一些人打电话，希望他们成为自己的客户。用销售行业的行话来讲，就是“cold-call”一些人。为此，应有一个文本文件存储这些人的姓名，该文件应有正确的格式，第一行包含文件中的人数，后面的行包含这些人的姓名。换言之，正确的格式如下所示。

```
4
George Washington
Benedict Arnold
John Adams
Thomas Jefferson
```

这个示例的目的是在屏幕上显示这些人的姓名(由销售人员读取)，这就是为什么只把姓名放在文件中，但没有电话号码的原因。

程序要求用户输入文件的名称，然后读取文件，以显示其中的人名。这听起来是一个简单的任务，但也会出现两个错误，需要退出整个过程：

- 用户可能键入不存在的文件名。这由 `FileNotFoundException` 异常来处理。
- 文件的格式可能不正确，这里可能有两个问题。首先，文件的第一行不是整数。第二，文件中可能没有第一行指定的那么多人名。这两种情况都需要在一个定制异常中处理，我们已经专门为此编写了异常 `ColdCallFileFormatException`。

还会有其他问题，虽然不致于退出整个过程，但需要删除某个人名，继续处理文件中的下一个人名(因此这需要在内层的 `try` 块中处理)。一些人是工业间谍，为公司的竞争对手工作，显然，我们不愿意让这些知道我们要做的工作。因此应确定哪些人是工业间谍，搜索条件是他们的姓名以 `B` 开头。这些人应在第一次准备数据文件时从文件中删除，但万一被工业间谍混入，就需要检查文件中的每个姓名，如果检测到一个工业间谍，就应抛出一个 `LandLineSpyFoundException`，当然，这是另一个定制的异常对象。

最后，编写一个类 `ColdCallFileReader` 来执行这个示例，该类维护与 `cold-call` 文件的连接，并从中检索数据。我们将以非常安全的方式编写这个类，如果其方法调用不正确，就会抛出异



常。例如，如果在文件打开前，调用了读取文件的方法，就会抛出一个异常。为此，我们编写了另一个异常类 `UnexpectedException`。

### 14.3.1 捕获用户定义的异常

首先是 `SolicitColdCall` 示例的 `Main()` 方法，它捕获用户定义的异常。注意，下面要调用 `System.IO` 命名空间和 `System` 命名空间中的文件处理类。

```
using System;
using System.IO;

namespace Wrox.ProCSharp.AdvancedCSharp
{
    class MainEntryPoint
    {
        static void Main()
        {
            string fileName;
            Console.WriteLine("Please type in the name of the file " +
                "containing the names of the people to be cold-called > ");
            fileName = Console.ReadLine();
            ColdCallFileReader peopleToRing = new ColdCallFileReader();

            try
            {
                peopleToRing.Open(fileName);
                for (int i=0 ; i<peopleToRing.NPeopleToRing; i++)
                {
                    peopleToRing.ProcessNextPerson();
                }
                Console.WriteLine("All callers processed correctly");
            }
            catch(FileNotFoundException ex)
            {
                Console.WriteLine("The file {0} does not exist", fileName);
            }
            catch(ColdCallFileFormatException ex)
            {
                Console.WriteLine(
                    "The file {0} appears to have been corrupted", fileName);
                Console.WriteLine("Details of problem are: {0}", ex.Message);
                if (ex.InnerException != null)
                {
                    Console.WriteLine(
                        "Inner exception was: {0}", ex.InnerException.Message);
                }
            }
            catch(Exception ex)
            {
                Console.WriteLine("Exception occurred:\n" + ex.Message);
            }
            finally
            {
                peopleToRing.Dispose();
            }
            Console.ReadLine();
        }
    }
}
```

这段代码基本上是一个循环，处理文件中的人名。开始时，先让用户输入文件名，再实例化类 `ColdCallFileReader` 的一个对象，这个类稍后定义，用于处理文件中数据的读取。注意是在第一个 `try` 块的外部读取文件——这是因为这里实例化的变量需要在后面的 `catch` 和 `finally` 块中使用，如果在 `try` 块中声明它们，它们在 `try` 块的闭合花括号处就出了作用域。

在 `try` 块中打开文件(使用 `ColdCallFileReader.Open()` 方法)，并循环处理其中所有的人名。`ColdCallFileReader.ProcessNextPerson()` 方法会读取并显示文件中的下一个人名，而 `ColdCallFileReader.NpeopleToRing` 属性则说明文件中应有多少个人名(通过读取文件的第一行来获得)。有 3 个 `catch` 块，其中两个用于处理 `FileNotFoundException` 和 `ColdCallFileFormatException` 异常，第三个则用于处理其他 .NET 错误。

在 `FileNotFoundException` 中，我们会为它显示一个信息，注意在这个 `catch` 块中，根本不会使用异常实例，原因是这个 `catch` 块用于说明应用程序的用户友好性。异常对象一般会包含技术信息，这些技术对开发人员是很有用的，但对于最终用户来说则没有什么用，所以在本例中我们将创建一个更简单的消息。

对于 `ColdCallFileFormatException` 处理程序，我们已经完成了编码，说明了如何提供更完整的技术信息，包括内层异常的细节。

最后，如果捕获到其他一般异常，就显示一个用户友好消息，而不是让这些异常由 .NET 运行库处理。注意我们选择不处理派生于 `System.Exception` 的异常，因为不直接调用非 .NET 的代码。

`Finally` 块清理资源。在本例中，是指关闭已打开的文件。`ColdCallFileReader.Dispose()` 方法完成了这个任务。

### 14.3.2 抛出用户定义的异常

下面看看处理文件读取，以及抛出用户定义的异常的类 `ColdCallFileReader` 的定义。这个类维护一个外部文件连接，所以需要确保它根据第 4 章有关释放对象的规则，正确地释放。这个类派生于 `IDisposable`。

首先声明一些变量：

```
class ColdCallFileReader : IDisposable
{
    FileStream fs;
    StreamReader sr;
    uint nPeopleToRing;
    bool isDisposed = false;
    bool isOpen = false;
```

`FileStream` 和 `StreamReader` 都在 `System.IO` 命名空间中，都是用于读取文件的基类。`FileStream` 用于连接文件，`StreamReader` 则专门用于读取文本文件，执行方法 `StreamReader()`，该方法读取文件中的一行文本。第 25 章在深入讨论文件处理时将讨论 `StreamReader`。

`isDisposed` 字段表示是否调用了 `Dispose()` 方法，我们选择执行 `ColdCallFileReader`，这样，一旦调用了 `Dispose()` 方法，就不能重新打开文件连接，重新使用对象了。`isOpen` 也用于错误检查——在本例中，检查 `StreamReader` 是否连接到打开的文件上。

打开文件和读取第一行的过程——告诉我们文件中有多少个人名——由 `Open()` 方法来处理：

```

public void Open(string fileName)
{
    if (isDisposed)
        throw new ObjectDisposedException("peopleToRing");

    fs = new FileStream(fileName, FileMode.Open);
    sr = new StreamReader(fs);

    try
    {
        string firstLine = sr.ReadLine();
        nPeopleToRing = uint.Parse(firstLine);
        isOpen = true;
    }
    catch (FormatException e)
    {
        throw new ColdCallFileFormatException(
            "First line isn't an integer", e);
    }
}

```

与其他 `ColdCallFileReader` 方法一样，该方法首先检查在删除对象后，客户机代码是否不正确地调用了它，如果是，就抛出一个预定义的 `ObjectDisposedException` 对象。`Open()` 方法也会检查 `isDisposed` 字段，看看 `Dispose()` 是否已经被调用。因为调用 `Dispose()` 会告诉调用者现在已经处理完对象，所以，如果已经调用了 `Dispose()`，就说明有一个试图打开新文件连接的错误。

接着，这个方法包含前两个内层的 `try` 块，其目的是捕获因文件的第一行没有包含一个整数而抛出的错误。如果出现这个问题，.NET 运行库就抛出一个 `FormatException`，捕获并转换一个更有意义的异常，表示 cold-call 文件的格式有问题。注意 `System.FormatException` 表示与基本数据类型相关的格式问题，而不是与文件有关，所以在本例中它不是传递回调用例程的一个特别有用的异常。新抛出的异常会被最外层的 `try` 块捕获。注意这里不需要清理资源，所以不需要 `finally` 块。

如果一切正常，就把 `isOpen` 字段设置为 `true`，表示现在有一个有效的文件连接，可以从中间读取数据。

`ProcessNextPerson()` 方法也包含一个内层 `try` 块：

```

public void ProcessNextPerson()
{
    if (isDisposed)
        throw new ObjectDisposedException("peopleToRing");
    if (!isOpen)
        throw new UnexpectedException(
            "Attempt to access cold-call file that is not open");
    try
    {
        string name;
        name = sr.ReadLine();
        if (name == null)
            throw new ColdCallFileFormatException("Not enough names");
        if (name[0] == 'B')
        {
            throw new LandLineSpyFoundException(name);
        }
        Console.WriteLine(name);
    }
}

```



```

        catch(LandLineSpyFoundException ex)
        {
            Console.WriteLine(ex.Message);
        }

        finally
        {
        }
    }
}

```

这里可能存在两个与文件相关的错误(假定有一个打开的文件连接, `ProcessNextPerson()` 方法会先进行检查)。第一, 读取下一个人名时, 可能发现这是一个工业间谍。如果发生这种情况, 在这个方法中就使用第一个 `catch` 块捕获异常。因为这个异常已经在循环中被捕获, 所以程序流会继续在程序的 `Main` 方法中执行, 处理文件中的下一个人名。

如果读取下一个人名, 发现已经到达文件的末尾, 也会发生错误。`StreamReader` 对象的 `ReadLine()` 方法的工作方式是: 如果到达文件末尾, 就会返回一个 `null`, 而不是抛出一个异常。所以, 如果找到一个 `null` 字符串, 文件的格式就不正确, 因为文件第一行上的数字要比文件中的实际人数多, 如果发生这种错误, 就抛出一个 `ColdCallFileFormatException`, 它由外层的异常处理程序捕获(使程序中断执行)。

这里还是不需要 `finally` 块, 因为没有要清理的资源, 但这次要放置一个空的 `finally` 块, 表示在这里可以完成你希望完成的任务。

这个示例就要完成了。`ColdCallFileReader` 还有另外两个成员: `NPeopleToRing` 属性返回文件中假定的人数, `Dispose()` 方法可以关闭打开的文件。注意 `Dispose()` 方法仅返回它是否被调用——这是执行该方法的推荐方式。它还检查在关闭前是否有一个文件流要关闭。这个例子说明了防御编码技术:

```

public uint NPeopleToRing
{
    get
    {
        if (isDisposed)
        {
            throw new ObjectDisposedException("peopleToRing");
        }

        if (!isOpen)
        {
            throw new UnexpectedException(
                "Attempt to access cold-call file that is not open");
        }

        return nPeopleToRing;
    }
}

public void Dispose()
{
    if (isDisposed)
    {
        return;
    }

    isDisposed = true;
}

```

```
        isOpen = false;

        if (fs != null)
        {
            fs.Close();
            fs = null;
        }
    }
}
```

### 14.3.3 定义异常类

最后，需要定义 3 个异常类。定义自己的异常是非常简单的，因为几乎不需要添加任何额外的方法。只需执行构造函数，确保基类构造函数正确调用即可。下面是 `SalesSpyFoundException` 的完整代码：

```
class SalesSpyFoundException : ApplicationException
{
    public SalesSpyFoundException(string spyName)
        : base("Sales spy found, with name " + spyName)
    {
    }

    public SalesSpyFoundException(
        string spyName, Exception innerException)
        : base(
            "Sales spy found with name " + spyName, innerException)
    {
    }
}
```

注意，这个类派生于 `ApplicationException`，正是我们期望的定制异常。实际上，如果要更正式地创建它，可以把它放在一个中间类中，例如 `ColdCallFileException`，它派生于 `ApplicationException`，再从这个类派生出两个异常类，并确保处理代码确切地知道哪个异常处理程序处理哪个异常即可。但为了使这个示例比较简单，就不这么做了。

在 `SalesSpyFoundException` 中，处理的内容要多一些。假定传送给构造函数的信息仅是找到的间谍名，就把这个字符串转换为含义更明确的错误信息。我们还提供了两个构造函数，其中一个构造函数的参数只是一个信息，另一个构造函数的参数还有一个内层异常。在定义自己的异常类时，最好把这两个构造函数都包括进来(但以后将不能在示例中使用第二个 `SalesSpyFoundException` 构造函数)。

对于 `ColdCallFileFormatException`，规则是一样的，但不必对信息进行任何处理：

```
class ColdCallFileFormatException : ApplicationException
{
    public ColdCallFileFormatException(string message)
        : base(message)
    {
    }

    public ColdCallFileFormatException(
        string message, Exception innerException)
        : base(message, innerException)
    {
    }
}
```



最后是 `UnexpectedException`，它看起来与 `ColdCallFileFormatException` 是一样的：

```
class UnexpectedException : ApplicationException
{
    public UnexpectedException(string message)
        : base(message)
    {
    }

    public UnexpectedException(string message, Exception innerException)
        : base(message, innerException)
    {
    }
}
```

下面准备测试该程序。首先，使用 `people.txt` 文件，其内容已经在前面列出了。

```
4
George Washington
Benedict Arnold
John Adams
Thomas Jefferson
```

它有 4 个名字(与文件中第一行给出的数字匹配)，包括一个间谍。接着，使用下面的 `people2.txt` 文件，它有一个明显的格式错误：

```
49
George Washington
Benedict Arnold
John Adams
Thomas Jefferson
```

最后，使用该例子，但指定一个不存在的文件名 `people3.txt`，对这 3 个文件名运行程序 3 次，得到的结果如下：

```
SolicitColdCall
Please type in the name of the file containing the names of the people to be cold
called > people.txt
George Washington
Sales spy found, with name Benedict Arnold
John Adams
Thomas Jefferson
All callers processed correctly
```

```
SolicitColdCall
Please type in the name of the file containing the names of the people to be cold
called > people2.txt
George Washington
Sales spy found, with name Benedict Arnold
John Adams
Thomas Jefferson
The file people2.txt appears to have been corrupted
Details of the problem are: Not enough names
```

```
SolicitColdCall
Please type in the name of the file containing the names of the people to be cold
called > people3.txt
The file people3.txt does not exist
```

这个小应用程序演示了处理程序中可能存在的错误和异常的许多不同方式。

## 14.4 小结

本章介绍了 C# 通过异常处理错误情况的机制，我们不仅可以输出代码中的一般错误代码，还可以用指定的方式处理最特殊的错误情况。有时一些错误情况是通过 .NET Framework 本身提供的，有时则需要编写自己的错误情况，如本章的例子所示。在这两种情况下，都可以采用许多方式来保护应用程序的工作流，使之不出现不必要和危险的错误。

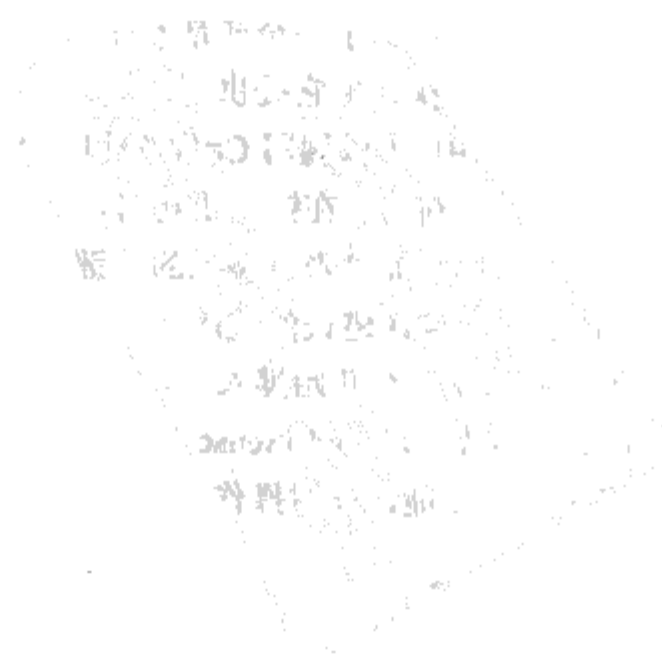
下一章将利用前面学习的许多内容，在 .NET 开发 IDE——Visual Studio 2008 中实践这些内容。

## 第 II 部分

# Visual Studio

第 15 章 Visual Studio 2008

第 16 章 部署



# 第 15 章

## Visual Studio 2008

我们已经熟悉了 C# 语言，下面开始学习本书的应用部分，这部分将介绍如何使用 C# 编写各种应用程序。在此之前，需要了解如何使用 Visual Studio 以及 .NET 环境提供的一些功能来编写程序。

本章将介绍在 .NET 环境中编程的真正含义，讨论 Visual Studio，这是主要的开发环境，在该环境中可以编写、编译、调试和优化 C# 程序。本章还提供编写良好应用程序的规则。Visual Studio 是用于编写 Web 窗体、Windows 窗体、XML Web 服务的主要 IDE。Windows 窗体及用户界面代码的编写详见第 31 章。本章的主要内容如下：

- 使用 Visual Studio 2008
- 使用 Visual Studio 2008 的重构功能
- Visual Studio 2008 的多目标功能
- 使用 WPF、WCF、WF 等新技术

本章还介绍建立面向 .NET Framework 3.0 或 3.5 的应用程序的步骤。通过 .NET Framework 3.0 类库提供的这类应用程序包括 Windows Presentation Foundation(WPF)、Windows Communication Foundation(WCF)和 Windows Workflow Foundation(WF)。使用 Visual Studio 2008 可以直接编写这些新应用程序类型。

### 15.1 使用 Visual Studio 2008

Visual Studio 2008 是一个全面集成的开发环境，用于编写、调试代码，把代码编译为程序集进行发布。实际上，Visual Studio 提供了一个非常专业的多文档界面应用程序，在该应用程序中可以进行与开发代码相关的任何操作，它提供了：

- 文本编辑器：在文本编辑器中，可以编写 C# 代码（以及 Visual Basic 2008 和 VC++ 代码）。这个文本编辑器相当复杂，例如，在键入语句时，它可以自动布局代码，如缩进代码行、匹配代码块的首尾括号、提供彩色编码的关键字等。在键入语句时，它还能执行一些语法检查，给可能产生编译错误的代码加上下划线，这也称为设计期间的调试。它还提供了 IntelliSense 功能。在开始键入时，IntelliSense 会自动显示类、字段或方法名。在开始键入方法的参数时，IntelliSense 也会显示可用重载方法的参数列表。下面的屏幕图 15-1 显示了这个功能，此时操作的是一个 .NET 基类 ListBox。

注意:

当 IntelliSense 列表框因某种原因不可见时, 请按下快捷键 Ctrl+Space, 可以在需要时打开 IntelliSense 列表框。

- 设计视图编辑器: 它可以在项目中可视化地放置用户界面和数据访问控件。此时, Visual Studio 会自动在源文件中添加必要的 C# 代码, 在项目中实例化这些控件 (在 .NET 中, 所有的控件实际上都是基类的实例)。
- 支持窗口: 它们可以查看和修改项目的各个方面, 例如, 这些窗口可以显示源代码中的类以及 Windows 窗体和 Web 窗体类中的可用属性(和它们的初始值)。也可以使用这些窗口指定编译选项, 例如代码需要引用哪些程序集。

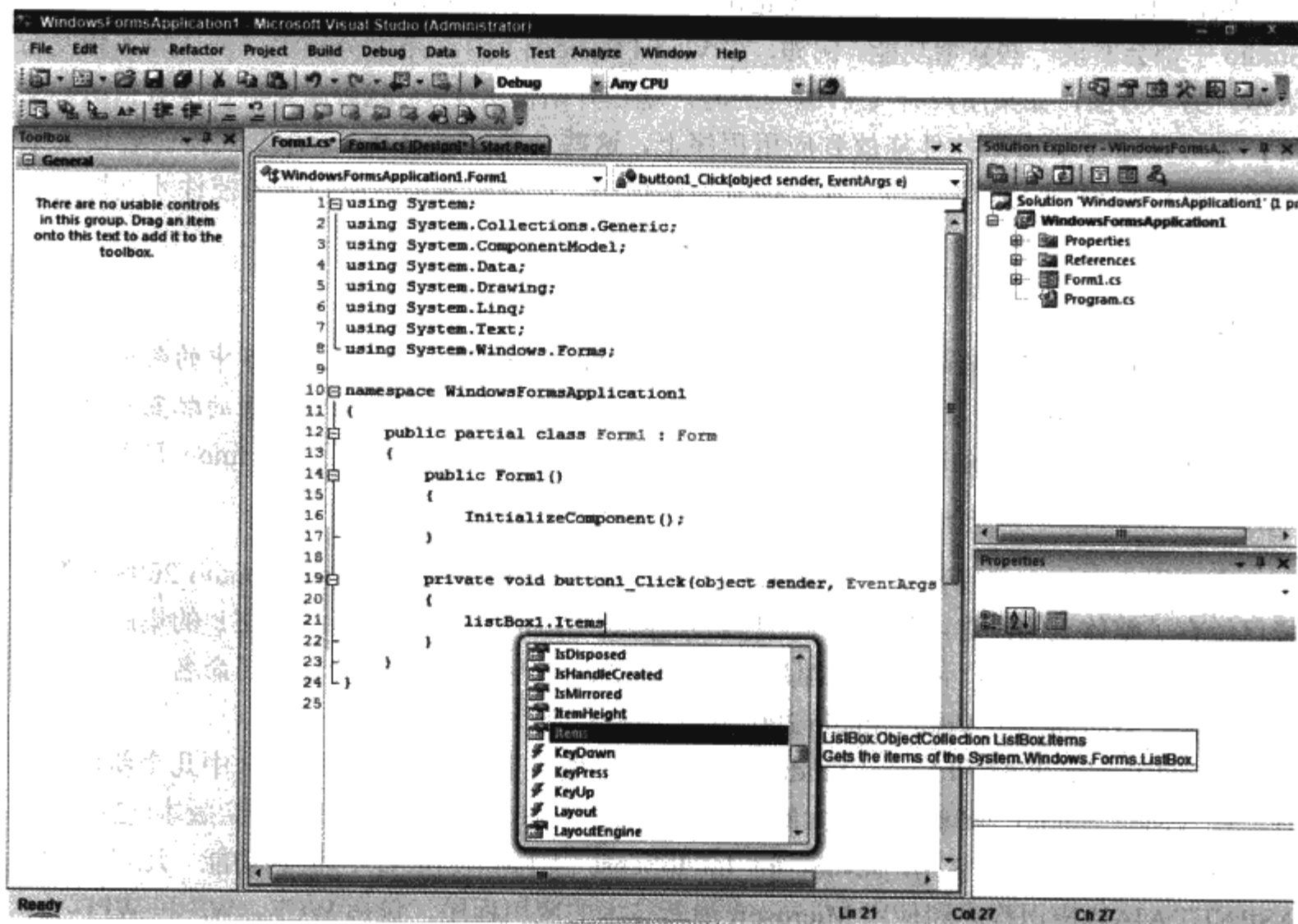


图 15-1

- 在环境中编译: 可以只选择一个菜单选项编译项目, 而不必在命令行上运行 C# 编译器。Visual Studio 会调用 C# 编译器, 把所有的相关命令行参数传递给编译器, 例如要引用的程序集和要生成什么类型的程序集(例如可执行文件或库.dll)。Visual Studio 还可以直接运行编译好的可执行文件, 用户可以查看这些文件的运行情况是否正常, 甚至可以选择不同的编译配置, 例如, 编译为发布版本或调试版本。
- 集成的调试程序: 代码在第一次运行时, 一般不会正确执行。也许在第二次、第三次才能正确运行。Visual Studio 无缝地链接到一个调试程序上, 可以在该调试环境中设置断点, 观察变量。



- 集成的 MSDN 帮助: Visual Studio 可以在 IDE 中调用 MSDN 文档说明。例如,在文本编辑器中,如果不能确定某个关键字的含义,可以选择它,按下 F1 键, Visual Studio 就打开 MSDN,显示相关的主题。同样,如果不知道某个编译错误是什么意思,可以选择错误消息,按下 F1 键,打开 MSDN,系统就会显示该错误的信息。
- 访问其他程序: Visual Studio 还能调用许多其他工具来查看和修改计算机或网络的一些内容,而无需退出开发环境。利用这些工具,可以检查运行服务和数据库连接,直接查询 SQL Server 表,甚至打开 Internet Explorer 窗口,浏览 Web。

当然,如果用户很熟悉 C++或 Visual Basic,就应很熟悉 Visual Studio 6 版本的开发环境,因此上面列出的许多特性就不是什么新内容了。Visual Studio 把以前在 Visual Studio 6 开发环境中可以使用的所有特性都组合起来,无论用户以前在 Visual Studio 6 中使用什么语言,在 Visual Studio 中都会发现一些新增功能。例如,在 Visual Basic 环境中,不能分别编译调试版本和发布版本。另一方面,如果用户有 C++编程经验,现在开始使用 C#,就可以获得许多数据访问支持,还可以通过单击把控件拖放到应用程序上,这些功能 Visual Basic 开发人员已经使用很久了,而在 C#中就是新增功能。在 C++开发环境中,只有最常用的用户界面控件才支持拖放操作。

#### 提示:

对于有 C++编程经验的人来说, Visual Studio 2008 去除了 Visual Studio 6 中的两个功能: edit-and-continue 调试和集成的配置器。 Visual Studio 2008 还不包含功能完善的配置器应用程序。但在 System.Diagnostics 命名空间中有许多 .NET 类能帮助进行配置。 perfmon 配置工具可以在命令行上使用(仅键入 perfmon),它有许多与 .NET 相关的新性能监视器。

无论用户有什么编程背景,都会发现,与 Visual Studio 6 相比, Visual Studio 2008 开发环境已经有了整体上的改进,包括增加一些新功能,一个跨语言的 IDE 和与 .NET 的集成,菜单和工具栏有一些新选项,许多选项都是 Visual Studio 6 已有的,但重新进行了命名。所以,用户需要花一些时间熟悉 Visual Studio 2008 的布局 and 命令。

Visual Studio 2005 和 Visual Studio 2008 的区别仅限于 Visual Studio 2008 中几个新增的特性。在 Visual Studio 2008 中,最大的变化是可以面向 .NET Framework 的特定版本(包括 .NET Framework 2.0、3.0 或 3.5), JavaScript IntelliSense 支持和使用 CSS 的新功能。还可以建立 ASP.NET AJAX 应用程序和使用 Microsoft 最新技术的应用程序,包括 WCF、WF 和 WPF。

在安装 Visual Studio 2008 时,会注意到一个最大的变化:这个新的 IDE 与 .NET Framework 3.5 一起工作。实际上,在安装 Visual Studio 2008 时,如果还没有安装 .NET Framework 3.0 或 3.5,就会自动安装它。与 Visual Studio 2005 一样,这个新的 IDE 不能与 .NET Framework 1.0 或 1.1 一起工作,如果仍要开发 1.0 或 1.1 版本的应用程序,就应在机器上安装 Visual Studio 2002 或 2003。安装 Visual Studio 2008 时,会安装 Visual Studio 的一个完整的新副本,但不会升级原来的 Visual Studio 2002、2003 或 2005 IDE。 Visual Studio 的这三个版本可以在机器上并行运行。

如果试图使用 Visual Studio 2008 打开 Visual Studio 2002、2003 或 2005 项目, IDE 就会发出警告:如果继续,会打开 Visual Studio Conversion Wizard(如图 15-2 所示),将该项目升级到 Visual Studio 2008。

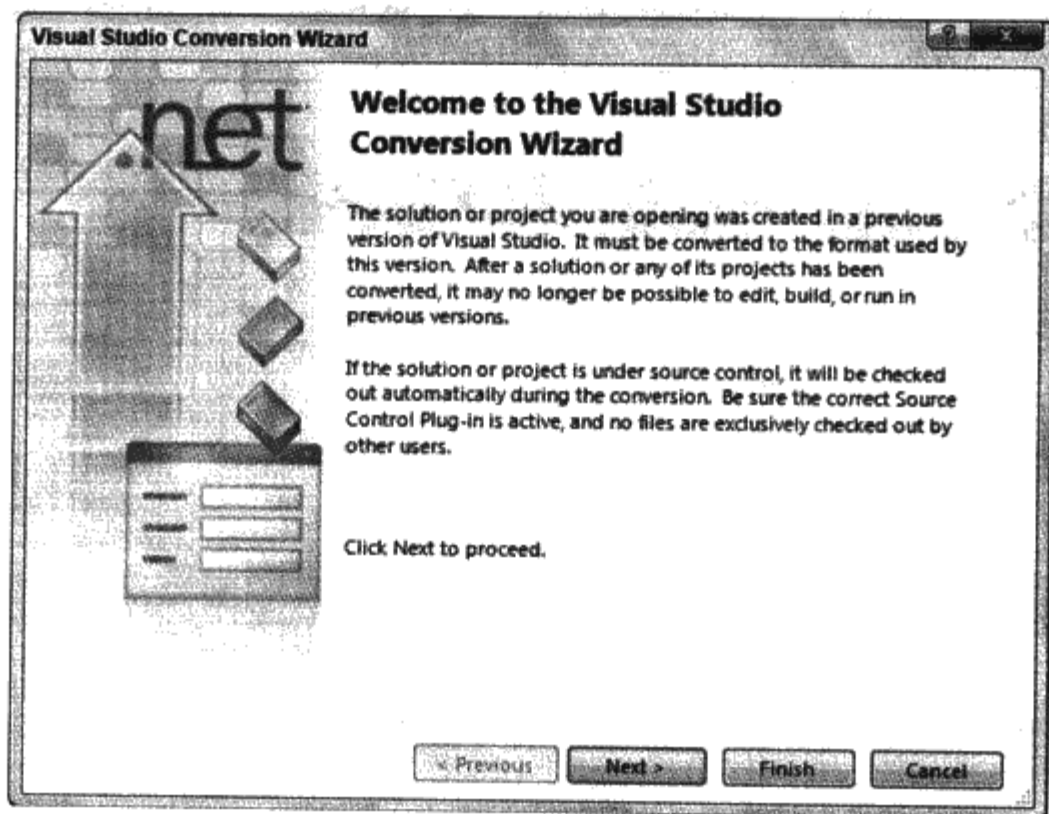


图 15-2

升级向导从 Visual Studio 2003 迁移到 Visual Studio 2008 时进行了巨大的改进。这个新向导可以对解决方案进行备份复制(如图 15-3 所示),还可以备份源控件中包含的解决方案。

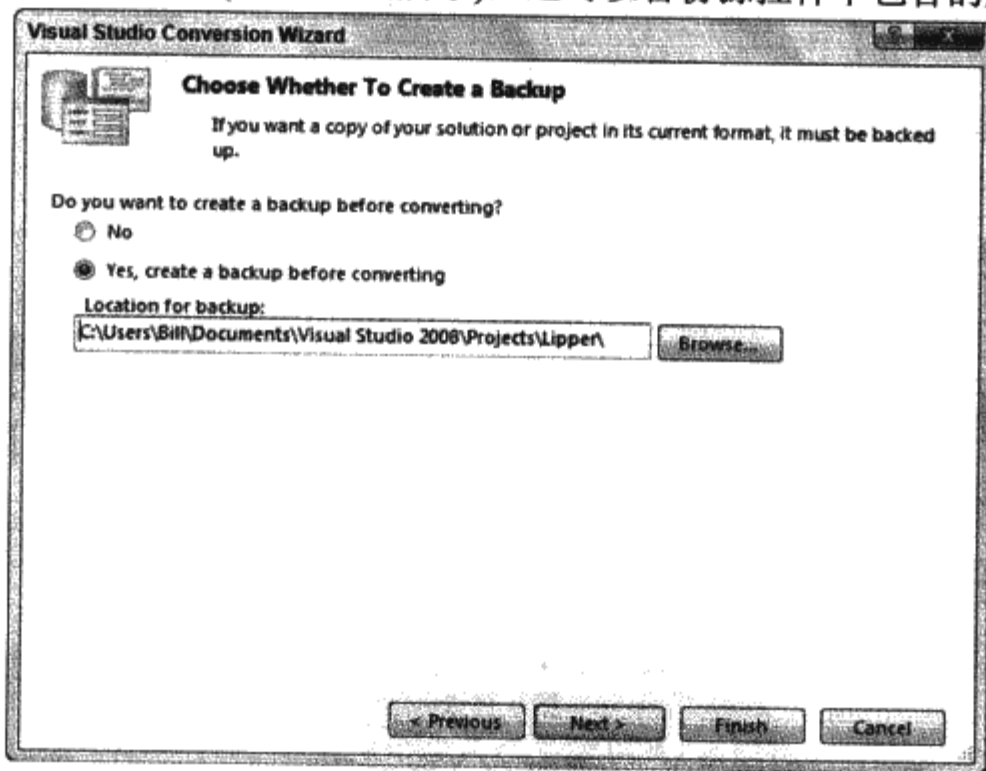


图 15-3

还可以在转换过程的最后一步生成一个转换报告。这个报告可以在 Visual Studio 的文档窗口中直接查看,如图 15-4 所示(进行了一个简单的转换)。

本书适合于专业人员使用,所以不会详细介绍 Visual Studio 2008 中的每个功能和菜单项。用户应自己熟悉该开发环境。介绍 Visual Studio 的主要目的是让用户熟悉建立和调试 C# 应用程序所涉及的所有概念,这样才能更好地使用 Visual Studio 2008。图 15-5 显示了 Visual Studio 2008 的外观(注意, Visual Studio 的外观是高度可定制的,在启动该开发环境时,看到的可能是位置不同的窗口或内容不同的窗口)。

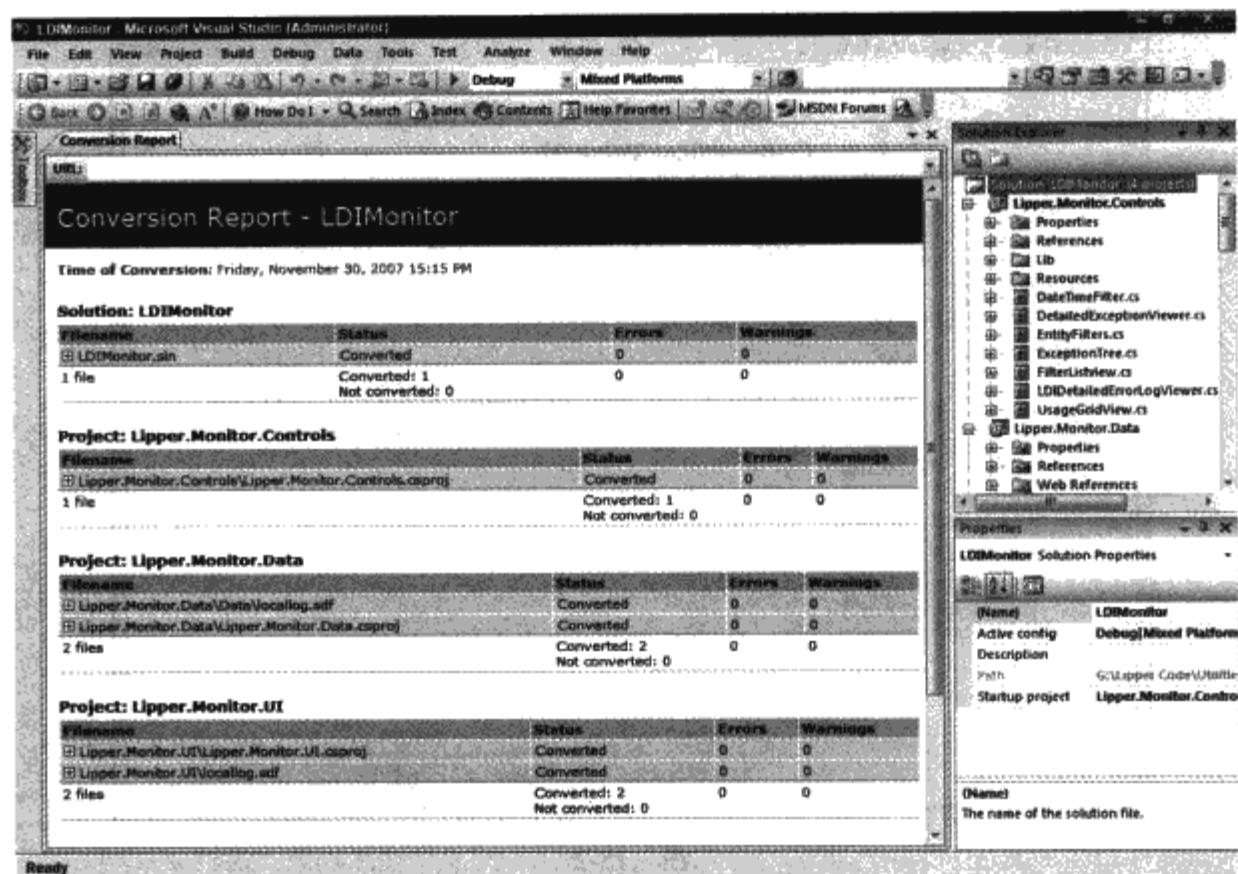


图 15-4

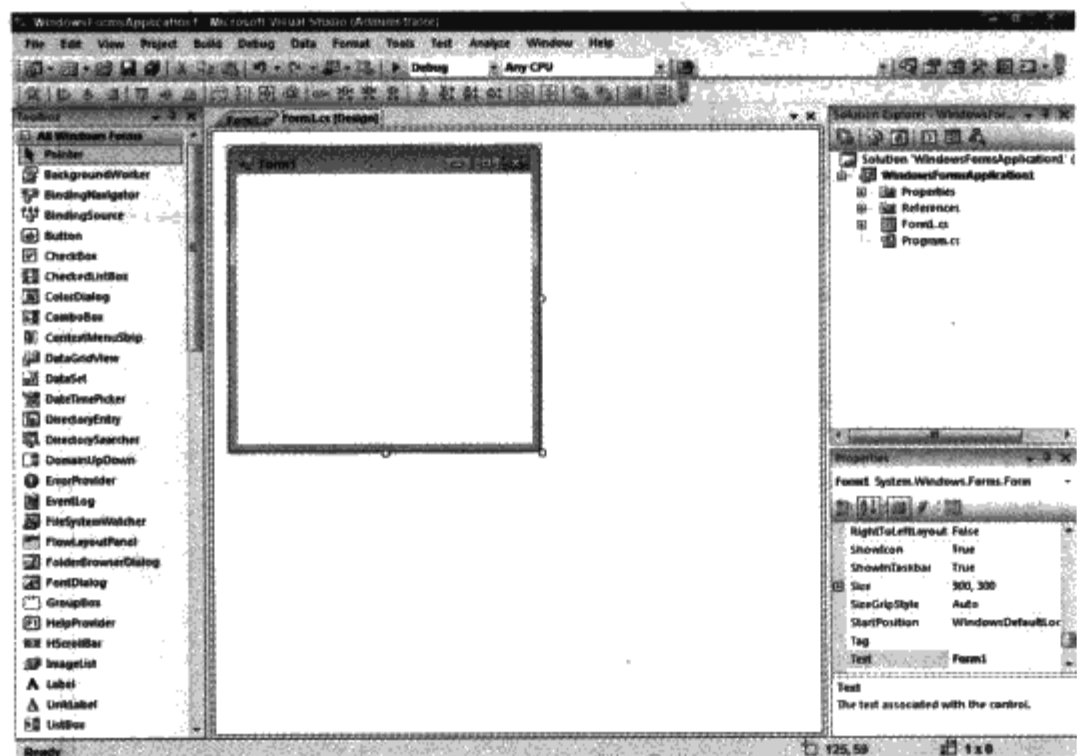


图 15-5

下面几节将创建、编写和调试项目，查看 Visual Studio 在每个阶段所能完成的操作。

15.1.1 创建项目

安装好 Visual Studio 2008 后，就可以开始编写第一个项目了。在 Visual Studio 中，很少从一个空白文件开始，从头键入 C#代码，就像本书前面的章节那样(当然，如果确实要从头开始编写代码，或者创建一个包含许多项目的解决方案，该 IDE 也提供了空应用程序项目选项)。编写项目的方式一般是先告诉 Visual Studio 要创建什么类型的项目，然后 Visual Studio 会自动生成文件和 C#代码，给出该类型项目的基本框架。接着，用户就可以在其中添加自己的代码了。

例如,如果要编写一个基于 Windows GUI 界面的应用程序(在.NET 中,这称为 Windows 窗体),Visual Studio 就会建立一个文件,其中包含的 C#源代码创建了一个基本窗体,这个窗体可以与 Windows 通信,接收事件。它还可以最大化、最小化、重新设置大小,用户只需在其中添加需要的控件和功能。如果应用程序要设计为命令行工具(控制台应用程序),Visual Studio 就会提供基本的命名空间、类和 Main()方法。

最后,在创建项目时,Visual Studio 还设置了提供给 C#编译器的编译选项——表示项目是编译为命令行应用程序、库,还是编译为 Windows 应用程序。它还告诉编译器需要引用的基类库(Windows GUI 应用程序需要引用许多与 Windows.Forms 相关的库,控制台应用程序则不需要)。当然如果必要,用户可以在编辑时,修改这些设置。

在第一次启动 Visual Studio 时,会显示空白的 IDE,出现的窗口称为 Start Page,如图 15-6 所示。这个 Start Page 是一个 HTML 页面,其中包含各种链接,通过它们可以进入有用的网站,打开现有的项目,或者启动一个新项目。

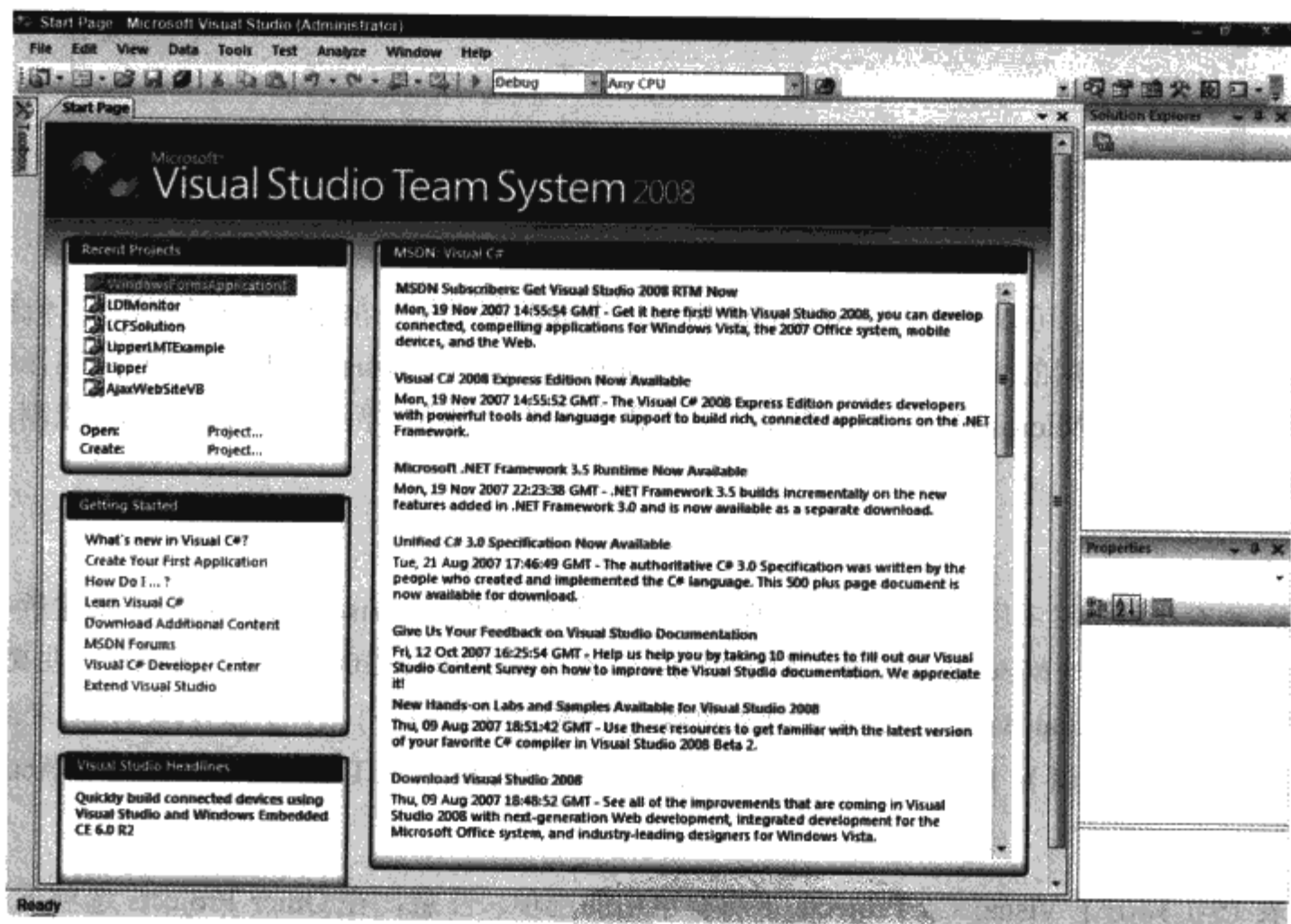


图 15-6

图 15-6 显示了使用 Visual Studio 2008 打开的 Start Page,其中有一个最近编辑的项目列表。单击其中的一个项目就可以打开它。

### 1. 选择项目类型

创建新项目时,可以在 Visual Studio 菜单上选择 File | New Project, 打开 New Project 对话框,如图 15-7 所示,其中给出了可以创建的各种项目。

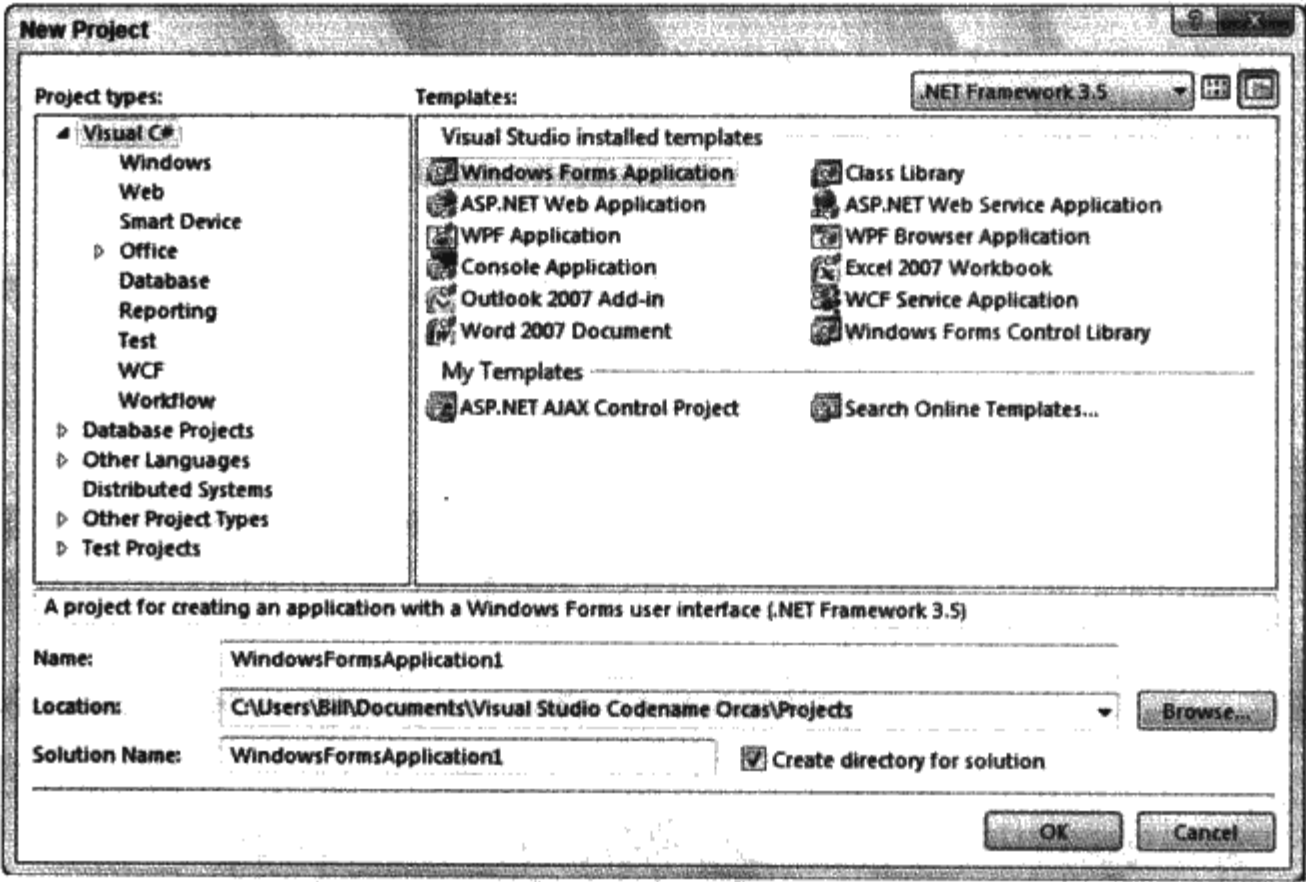


图 15-7

使用该对话框，可以选择 Visual Studio 为用户生成的某种初始框架文件和代码、编译选项，以及编译代码所使用的编译器：Visual C#、Visual Basic 2008、Visual J#或 Visual C++编译器。从这里可以看出，Microsoft 为.NET 提供了多种语言集成。对于本例，我们选择了 C#控制台应用程序。

注意：

这里不打算介绍不同类型的项目的所有选项。在 C++方面，Visual Studio 可以创建所有旧的 C++项目类型——MFC 应用程序、ALT 项目等。在 Visual Basic 2008 方面，选项有一些变化，例如，可以创建 Visual Basic 2008 命令行应用程序(控制台应用程序)、.NET 组件(类库)或者.NET 控件(Windows 控件库)，但不能创建基于 COM 的旧风格的控件(.NET 控件可以取代这种 ActiveX 控件)。

表 15-1 列出了 Visual C# Projects 下所有可用的选项。注意，在 Other Projects 选项下还有一些比较专业的 C#模板项目。

表 15-1

如果 选 择	将生成的 C#代码和编译选项
Windows Froms Application	响应事件的基本空窗体
Class Library	可以由其他代码调用的.NET 类
WPF Application	响应事件的基本空窗体。这个项目类型类似于 Windows Froms Application 项目类型(Windows Froms)，这个 Windows Application 项目类型允许建立基于 XAML 的智能客户解决方案



(续表)

如 果 选 择	将生成的 C#代码和编译选项
WPF Browser Application	类似于 Windows Application for WPF, 但这个变体类型可以建立面向浏览器的基于 XAML 的应用程序
ASP.NET Web Application	基于 ASP.NET 的网站: ASP.NET 页面和 C#类生成的、从页面发送给浏览器的 HTML 响应
ASP.NET Web Service Application	用作功能全面的 Web 服务的 C#类
ASP.NET AJAX Server Control	建立在 ASP.NET 应用程序中使用的定制服务器控件
Web Control Library	可以由 ASP.NET 页面调用的控件, 在浏览器上显示这个控件时, 可生成给出该控件外观的 HTML 代码
WPF Custom Control Library	可以在 WPF 应用程序中使用的定制控件
WPF User Control Library	用 WPF 建立的定制控件库
Windows Forms Control Library	创建在 Windows Forms 应用程序中使用的控件的项目
Console Application	在命令行提示符上或控制台窗口中运行的应用程序
WCF Service Application	用于 WCF 服务的项目类型
Windows Service	在 Windows 操作系统的后台上运行的服务
Reports Application	创建带有 Windows 用户界面和 Report 的项目
Crystal Reports Windows Application	该项目用于创建带有 Windows 用户界面和 Crystal Report 示例的 C#应用程序
SQL Server Project	该项目用于创建在 SQL Server 中使用的类
Smart Device	可以面向某种类型的移动设备的项目类型
Sequential Workflow Service Library	把顺序工作流提供为 WCF 服务的项目
State Machine Workflow Service Library	把状态机工作流提供为 WCF 服务的项目
Syndication Service Library	把 Syndication 服务提供为 WCF 服务的项目
WCF Service Library	这个项目类型可以创建 WCF 服务类库(.dll), 其端点通过 XML 配置文件来控制
Empty Workflow Project	这个项目类型提供了一个空项目, 以创建工作流
Sequential Workflow Console Application	创建顺序工作流控制台应用程序的项目
Sequential Workflow Library	创建顺序工作流库的项目
SharePoint 2007 Sequential Workflow	创建 SharePoint 顺序工作流的项目
SharePoint 2007 State Machine Workflow	创建 SharePoint 状态机工作流的项目
State Machine Workflow Console Application	创建 SharePoint 状态机工作流控制台应用程序的项目
State Machine Workflow Library	创建状态机工作流库的项目
Workflow Activity Library	这个项目类型可以创建活动库, 以后在工作流中重用于构建块
Office	这组项目类型可以建立面向 Microsoft Office(Word、Excel、PowerPoint、InfoPath、Outlook 和 SharePoint)的应用程序或插件是一系列

如前所述，这并不是.NET Framework 3.5 项目的完整列表，但这是一个很好的开始。这个项目列表增加了许多内容，主要是 WPF、WCF 和 WF。本书后面的章节将介绍这些新功能。请参阅第 34、42 和 43 章。

## 2. 新建的控制台项目

在上述对话框中选择 Console Application 选项，单击 OK 按钮，Visual Studio 就会提供几个文件，包括一个源文件 Program.cs，其中包含了最初的框架代码。图 15-8 显示了 Visual Studio 编写的代码。

可以看出，这是一个 C# 程序，但它实际上没有做任何工作，只是包含了 C# 可执行程序所必需的基本项：一个命名空间和一个包含 Main() 方法的类，其中 Main() 方法是程序的入口点。(严格说来，命名空间是不必要的，但不声明命名空间是一种不好的编程习惯)。按下 F5 键，或者选择 Debug 菜单中的 Start，这段代码就可以编译和运行。在这样做之前，在程序中加入一行代码，让应用程序完成某个工作：

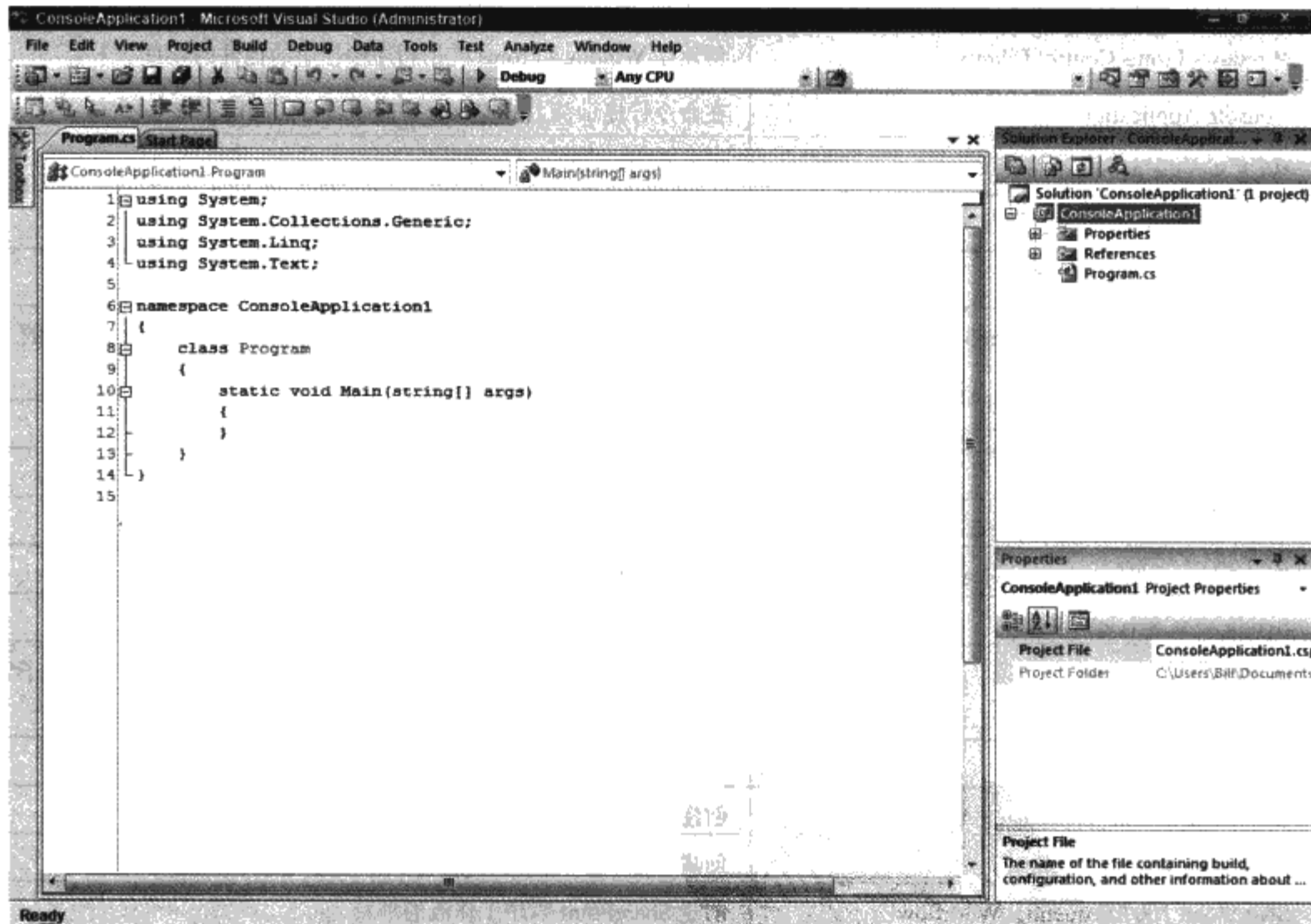


图 15-8

```
static void Main(string[] args)
{
    Console.WriteLine("Hello from all the editors at Wrox Press");
}
```

如果编译并运行了该项目，就会显示一个控制台窗口，但该窗口几乎立即就消失了，用户几乎看不到输出的信息。原因是在创建该项目时，Visual Studio 记住了用户指定的设置，所以

会把它编译并运行为控制台应用程序。然后，Windows 知道需要运行一个控制台应用程序，但没有运行该程序的控制台窗口。所以，Windows 就创建一个控制台窗口，并运行该程序。只要程序退出，Windows 就认为不再需要该控制台窗口，因此就即时删除了它。这些都是非常逻辑化的操作，但如果希望能看到项目的输出结果，这些操作对用户就没有什么帮助。

要避免这个问题，可以在 Main() 方法结束前插入下述代码：

```
static void Main(string[] args)
{
    Console.WriteLine("Hello from all the editors at Wrox Press");
    Console.ReadLine();
}
```

这样，代码运行后，会显示其输出结果，之后执行 Console.ReadLine() 语句，用户按下回车键后，程序退出。这表示在用户按下回车键之前，控制台窗口一直会挂起。

注意这仅是在 Visual Studio 中试运行控制台应用程序的问题。如果编写的是一个 Windows 应用程序，该应用程序显示的窗口会自动停留在屏幕上，直到用户显式退出程序为止。同样，如果在命令行提示符上运行一个控制台应用程序，就没有窗口消失的问题。

### 3. 其他文件的创建

Program.cs 源代码文件不是 Visual Studio 创建的唯一文件。如果查看一下 Visual Studio 创建项目的文件夹，就会看到其中不仅有 C# 文件，还有如图 15-9 所示的完整目录结构。

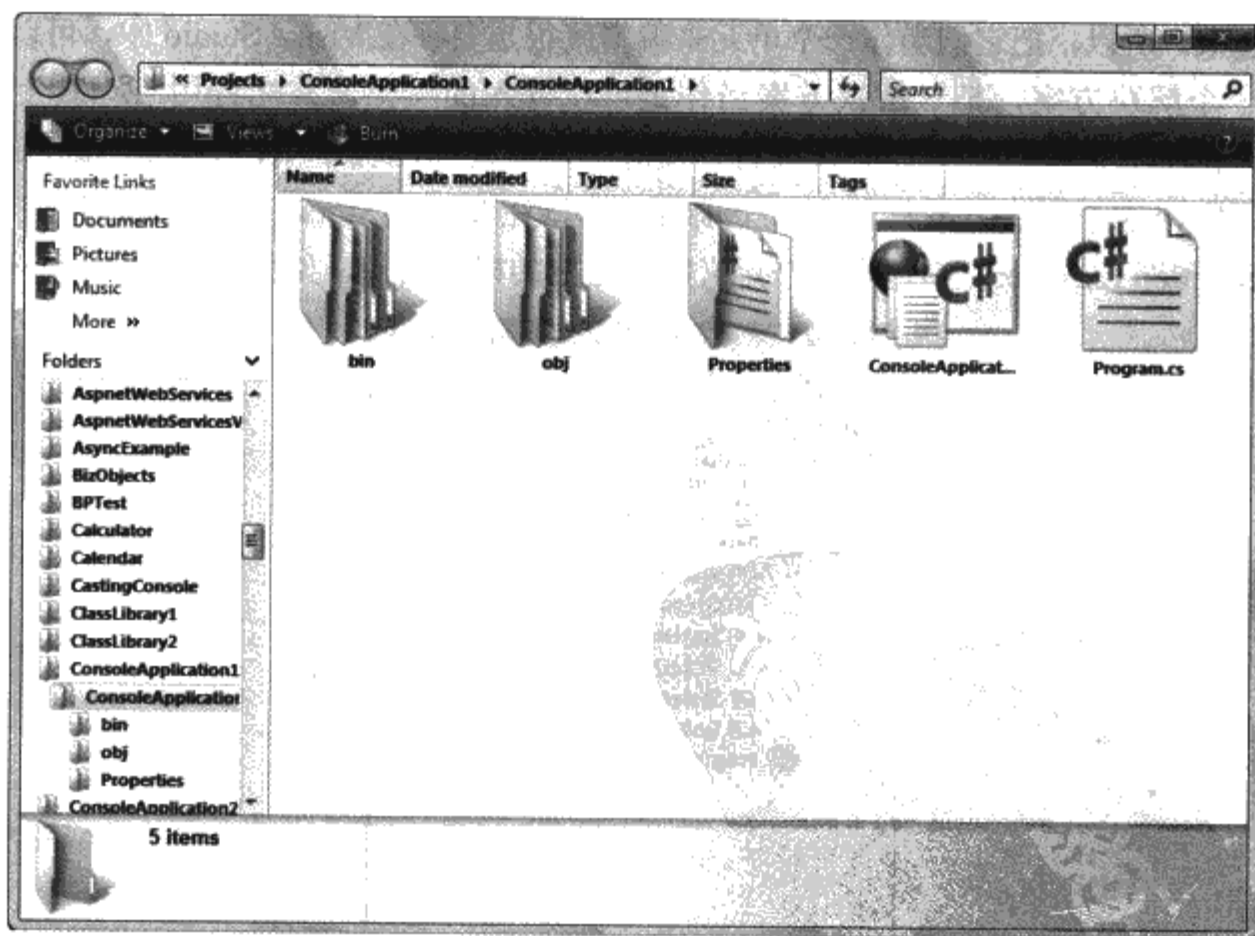


图 15-9

文件夹 bin 和 obj 存储编译好的文件和中间文件，obj 的子文件夹存储各种临时或中间文件，bin 的子文件夹存储编译好的程序集。

**注意:**

传统上, Visual Basic 开发人员只是编写它们的代码, 然后运行它们。代码在发布前, 必须编译成可执行文件, 但 Visual Basic 在调试中隐藏了这个过程。而在 C# 中, 这个过程是显式的: 要运行代码, 必须先编译它, 即在某处创建一个程序集。

还有一个包含 `AssemblyInfo.cs` 文件的 `Properties` 文件夹。在项目的主文件夹 `ConsoleApplication1` 中, 剩余的文件都是由 Visual Studio 建立的, 它们包含项目的信息(例如它包含的文件), 这样, Visual Studio 就知道如何编译项目, 在下一次打开该项目时, 知道如何读取它。

### 15.1.2 解决方案和项目

项目和解决方案的一个重要区别是:

- 项目是一组要编译到单个程序集(在某些情况下, 是单个模块)中的所有源文件和资源。例如, 项目可以是类库, 或一个 Windows GUI 应用程序。
- 解决方案是构成某个软件包(应用程序)的所有项目集。

为了说明这个区别, 考虑一下在发布一个项目(该项目包含多个程序集)时的情况。例如, 其中可能有一个用户界面, 有某些定制控件和其他组件, 它们都作为应用程序的库文件一起发布。不同的管理员甚至还有不同的用户界面。应用程序的不同部分都包含在一个独立的程序集中, 因此, 在 Visual Studio 看来, 它们都是独立的项目。但我们要同时编写这些项目, 使它们彼此连接起来。所以, 把它们当作一个单元来编辑就很重要。在 Visual Studio 中, 可以把所有的项目看作一个解决方案, 把该解决方案当作是可以读入的单元, 并允许用户在其上工作。

前面讨论了如何创建一个控制台项目。实际上, 在前面的例子中, Visual Studio 创建的是一个解决方案, 这个解决方案只包含一个项目。可以在 Visual Studio 的一个窗口中查看它, 该窗口称为 `Solution Explorer`, 它包含一个定义解决方案的树形结构, 如图 15-10 所示。

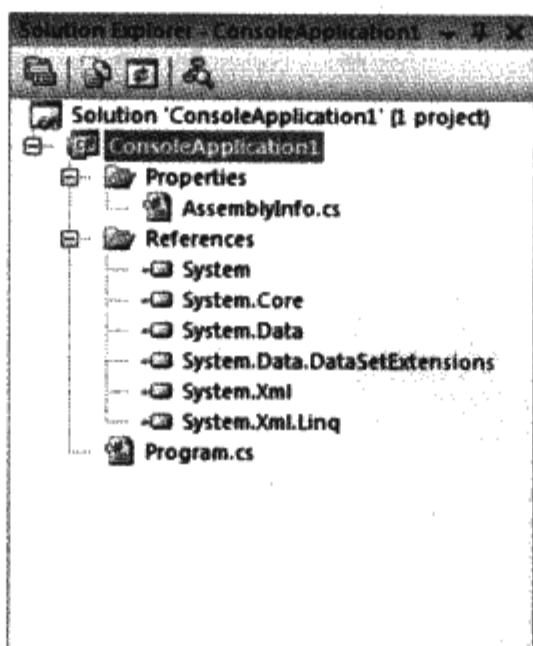


图 15-10

图 15-10 说明了项目包含源文件 `Program.cs` 和另一个 C# 源文件 `AssemblyInfo.cs` (在文件夹 `Properties` 中), `AssemblyInfo.cs` 包含程序集的描述信息和指定的版本信息(该文件详见第 17 章)。

Solution Explorer 也指定了项目通过命名空间引用的程序集。扩展 Solution Explorer 中的文件夹 Reference, 就可以看到它。

如果没有改变 Visual Studio 的任何默认设置, Solution Explorer 就在屏幕的右上角。如果看不到它, 可以选择 View 菜单中的 Solution Explorer。

解决方案用扩展名为.sln 的文件来表示, 在本例中, 就是 ConsoleApplication1.sln。该项目由主文件夹中的各个文件来表示。如果试图使用 Notepad 编辑这些文件, 就会发现它们大多数都是纯文本文件, 为了与.NET 和依赖于开放标准的.NET 工具保持一致, 它们大都是 XML 格式。

#### 注意:

C++开发人员应认识到, Visual Studio 解决方案对应于旧的 C++项目工作区(存储在.dsw 文件中), Visual Studio 项目对应于旧的 C++项目(.dsp 文件)。另一方面, Visual Basic 开发人员应注意, 解决方案对应于旧的 Visual Basic 项目组(.vbg 文件), .NET 项目对应于旧的 Visual Basic 项目(.vbp 文件)。Visual Studio 与旧 Visual Basic IDE 的区别是, Visual Studio 总是自动创建一个解决方案。在 Visual Studio 6 中, Visual Basic 开发人员最初会得到一个项目, 如果要得到项目组, 就必须在 IDE 中显式指定。

### 1. 给解决方案添加另一个项目

下面几节将说明 Visual Studio 如何处理 Windows 应用程序和控制台应用程序。为此, 本节将创建一个 Windows 项目 BasicForm, 并把它添加到当前的解决方案 ConsoleApplication1 中。

#### 注意:

最终我们将得到包含一个 Windows 应用程序和一个控制台应用程序的解决方案。这种情况并不常见——我们一般会在解决方案中包含一个应用程序和多个库, 但这个解决方案可以演示更多的代码。如果用户编写的工具需要作为 Windows 应用程序和命令行工具来运行, 就可以创建这样的解决方案。

创建新项目可以使用两种方式: 一是进入 File 菜单, 选择 New Project 选项(如前所述)。二是在 File 菜单中选择 Add | New Project。如果从菜单中选择 New Project 选项, 会打开熟悉的 New Project 对话框, 但这次 Visual Studio 要在已有的 ConsoleApplication1 项目位置中创建新项目, 如图 15-11 所示。

如果选中这个选项, 就会添加一个新项目, ConsoleApplication1 解决方案现在就应包含一个控制台应用程序和一个 Windows 应用程序。

#### 注意:

为了保持 Visual Studio 的语言无关性, 新项目不一定是 C#项目, 在同一个解决方案中, 可以有 C#项目、Visual Basic 2008 项目或 C++项目。但这里仍使用 C#, 因为这是一本介绍 C# 的书。



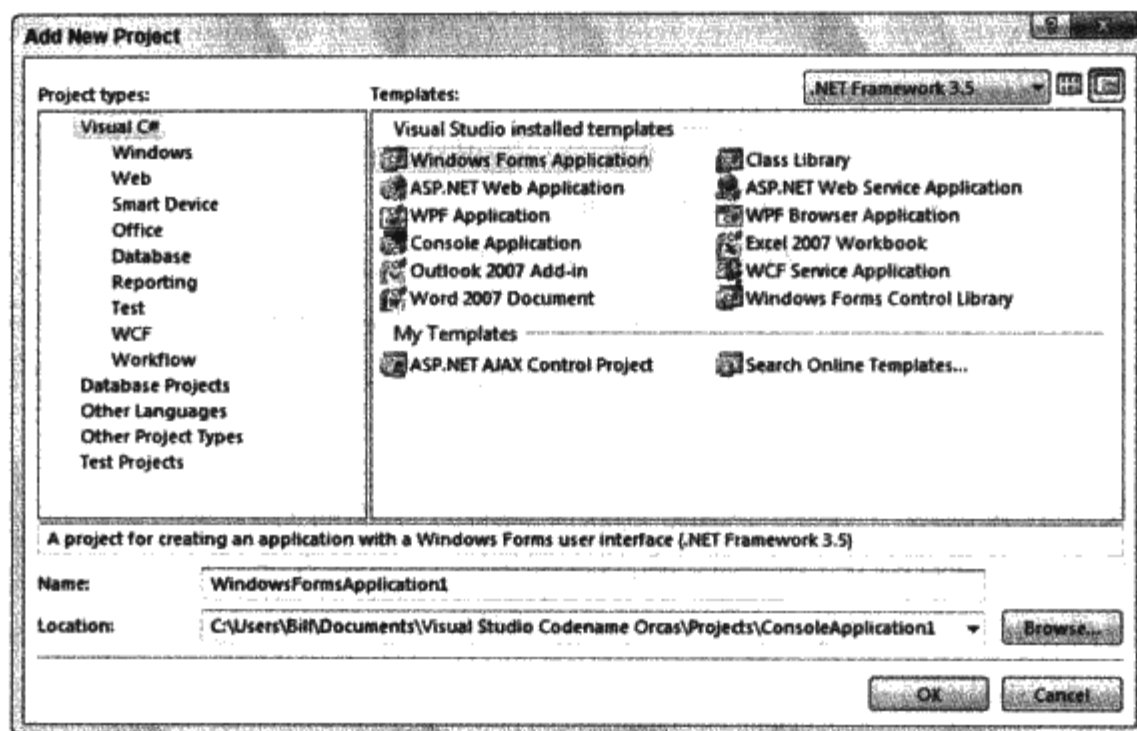


图 15-11

当然，ConsoleApplication1 对于这个解决方案来说，不再是一个合适的名称了。右击该名称，从弹出的菜单中选择 Rename，改变其名称。如果把它重命名为 DemoSolution，Solution Explorer 窗口就应如图 15-12 所示。

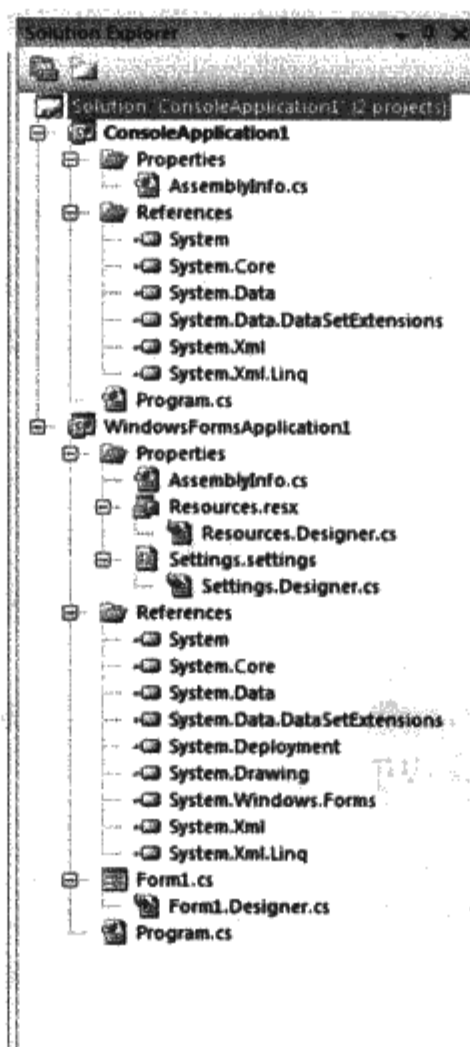


图 15-12

从这里可以看出，Visual Studio 会自动让新增的 Windows 项目引用那些对 Windows 窗体的

功能来说非常重要的某些附加基类。

如果查看一下 Windows 资源管理器，就会发现解决方案的文件名已经改为 Demo-Solution.sln。一般情况下，如果要重新命名文件，最好在 Solution Explorer 中进行，因为 Visual Studio 会自动更新其他项目文件对该文件的引用。如果使用 Windows 资源管理器重命名文件，就会中断解决方案，因为 Visual Studio 不能定位它需要读取的所有文件，用户必须手工编辑项目和解决方案文件，以更新文件引用。

## 2. 设置启动项目

如果在解决方案中有多个项目，就必须确保该解决方案在某一刻只运行一个项目。在编译解决方案时，将编译其中的所有项目。但在按下 F5 键或选择 Start 时，必须告诉 Visual Studio 先运行哪个项目。如果有一个可执行文件，它调用了几个库，显然应先运行这个可执行文件。对于本例，项目中有两个独立的可执行文件，就必须逐个调试它们。

可以告诉 Visual Studio 应运行哪个项目，方法是在 Solution Explorer 中右击该项目，在弹出的菜单中选择 Set as Startup Project。这就告诉 Visual Studio 哪个项目是当前的启动项目，因为它在 Solution Explorer 中是黑体显示，在图 15-12 上，就是 WindowsApplication1。

### 15.1.3 Windows 应用程序代码

在 Visual Studio 第一次创建应用程序时，Windows 应用程序包含的启动代码要比控制台应用程序多，因为创建一个窗口是一个比较复杂的过程。第 31 章将详细讨论 Windows 应用程序的代码，这里只给出 WindowsApplication1 项目中类 Form1 的代码，说明自动生成的代码有多少。

### 15.1.4 读取 Visual Studio 6 项目

利用 C#编写代码时，不需要读取旧的 Visual Studio 6 项目，因为 C#代码在 Visual Studio 6 中不存在。但是，语言的互操作性是 .NET Framework 的一个重要部分，C#代码可能要与 Visual Basic 代码或 C++代码一起使用。此时就需要编辑用 Visual Studio 6 创建的项目了。

Visual Studio 可以读取和升级 Visual Studio 6 项目和工作区，这不同于 Visual C++和 Visual Basic 项目。

- 在 Visual C++中，不需要修改源代码。所有的旧 C++代码使用新的 C++编译器仍可正常工作。显然它们不是托管代码，但仍可以编译为运行在 .NET 运行库外部的代码。如果代码要与 .NET Framework 集成起来，就需要编辑它。如果要用 Visual Studio 读取旧的 Visual C++项目，则只需增加一个新的解决方案文件和更新的项目文件。原来的 .dsw 和 .dsp 文件不变，这样，该项目就可以在需要时用 Visual Studio 6 编辑了。
- 对于 Visual Basic，就比较复杂了。如第 1 章所述，Visual Basic 2008 的设计与 Visual Basic 6 非常类似，也有许多相同的语法，但在许多方面，它是一种新的语言。在 Visual Basic 6 中，源代码大都由控件的事件处理程序组成，实际上，实例化主窗口和许多控件的代码并不是 Visual Basic 代码的一部分，而是隐藏在后台中，是项目配置的一部分。而

Visual Basic 2008 则与 C#的工作方式相同，把整个程序都放在源文件中，即显示主窗口和所有控件的代码都必须放在源文件中。与 C#一样，Visual Basic 2008 要求所有的代码都是面向对象的，是类的一部分，而 Visual Basic 6 则没有.NET 中类的概念。如果要使用 Visual Studio 读取 Visual Basic 6 项目，在处理前必须把全部源代码升级为 Visual Basic 2008，这会对 Visual Basic 6 代码做许多修改。在很大程度上，Visual Studio 可以自动进行修改，创建一个新的 Visual Basic 2008 解决方案，它提供的源代码与原来的 Visual Basic 6 代码大不相同，需要仔细检查生成的代码，以确保项目仍能正常工作。甚至还要找出 Visual Studio 注释掉的某些代码块，因为 Visual Studio 不能确定这些代码做什么工作，这部分代码必须手工编辑。

### 15.1.5 项目的浏览和编码

本节将介绍在给项目添加代码时，Visual Studio 所提供的一些功能。

#### 1. 可折叠的编辑器

Visual Studio 的一个重要改进是，它把可折叠的编辑器当作默认的代码编辑器，如图 15-13 所示。

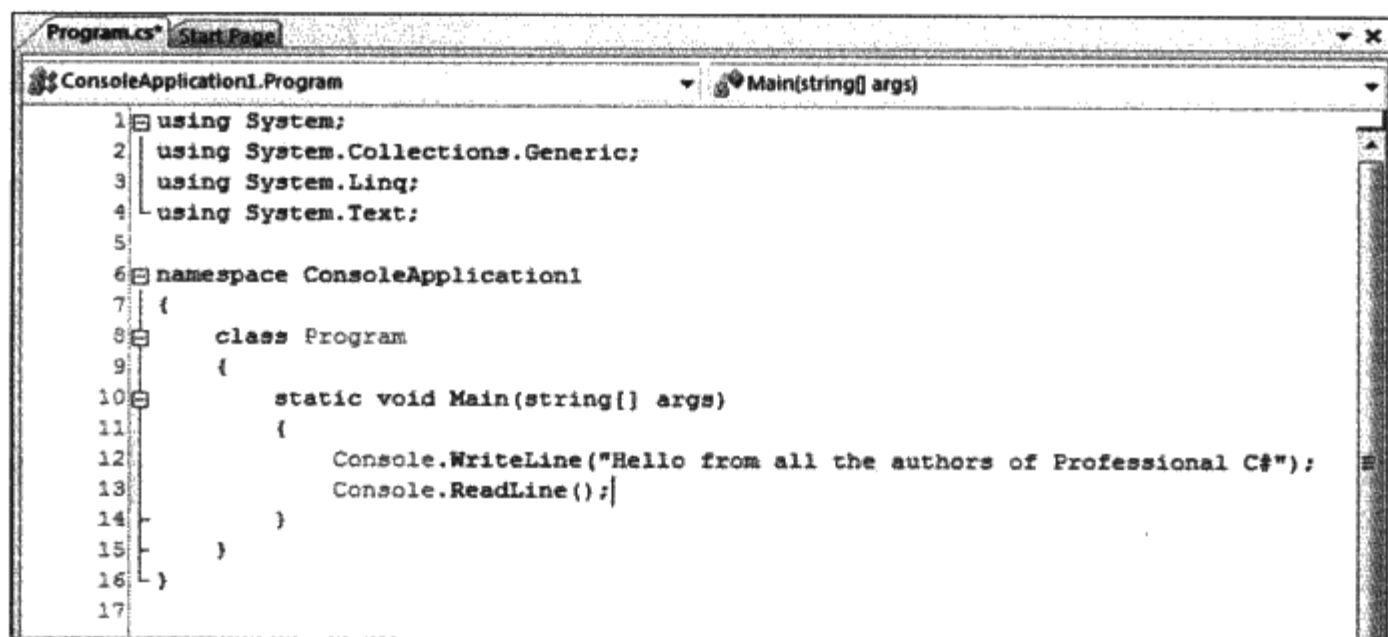


图 15-13

图 15-13 显示了前面生成的控制台应用程序的代码。但要注意窗口左边的小减号图标，它们标记出了编辑器认为是新代码块(或文档注释)的起点。单击这些图标，可以关闭对应的代码块视图，就像关闭树形控件中的节点一样，如图 15-14 所示。

在编辑时，可以只考虑要查看的那些代码块，关闭目前不想查看的代码块。而且，如果不喜欢编辑器关闭代码块的方式，可以用 C#预处理器指令`#region`和`#endregion`(详见本书前面的章节)指定另一种方式。例如，假定要折叠 `Main()`方法中的代码，可以先添加如图 15-15 所示的代码。

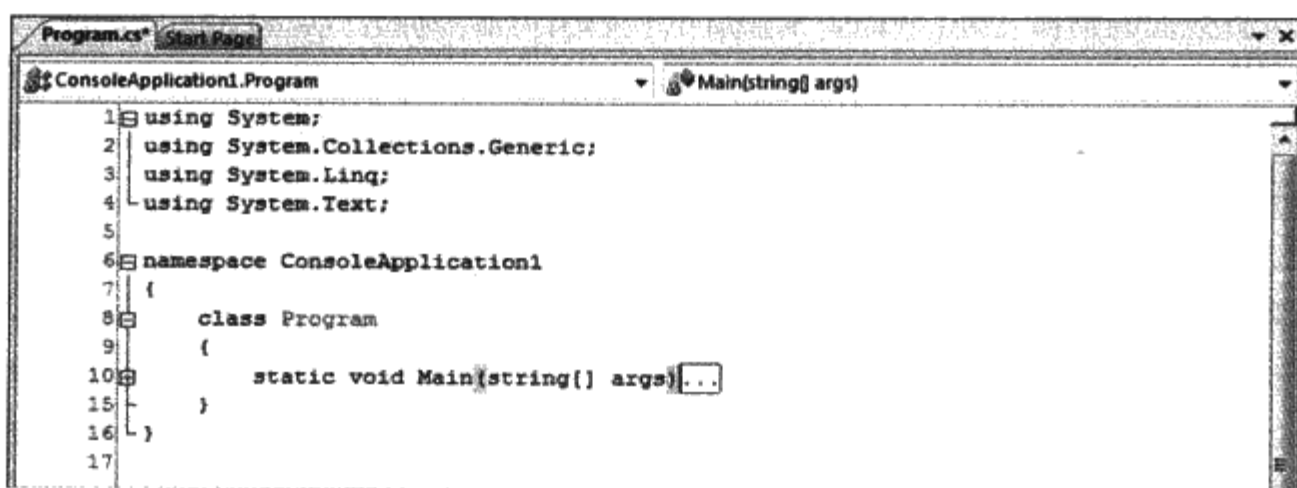


图 15-14

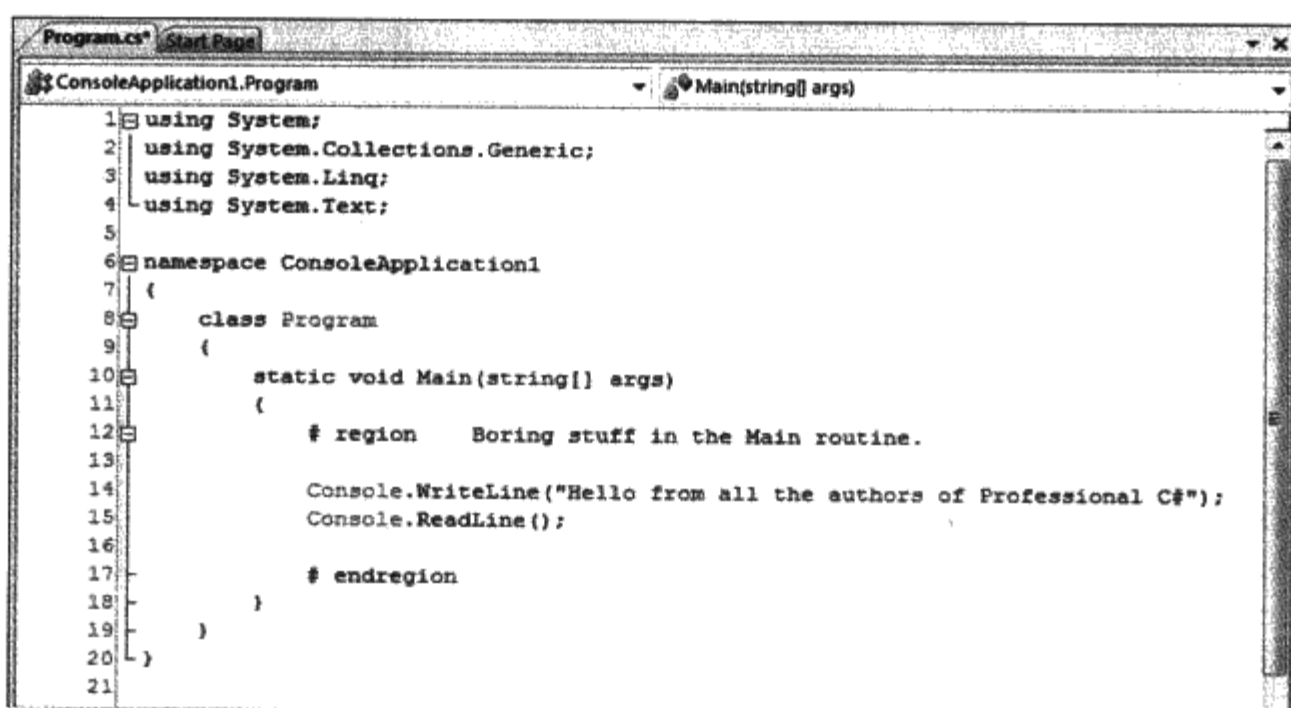


图 15-15

代码编辑器会自动检测#region 块，通过#region 指令放置一个新的减号图标，如图 15-15 所示，以便关闭该代码区域。把这个代码块放在一个区域中，就可以用在#region 指令中指定的注释标记该区域，让编辑器关闭该代码块，如图 15-16 所示。但编译器会忽略这些指令，按正常情况编译 Main()方法。

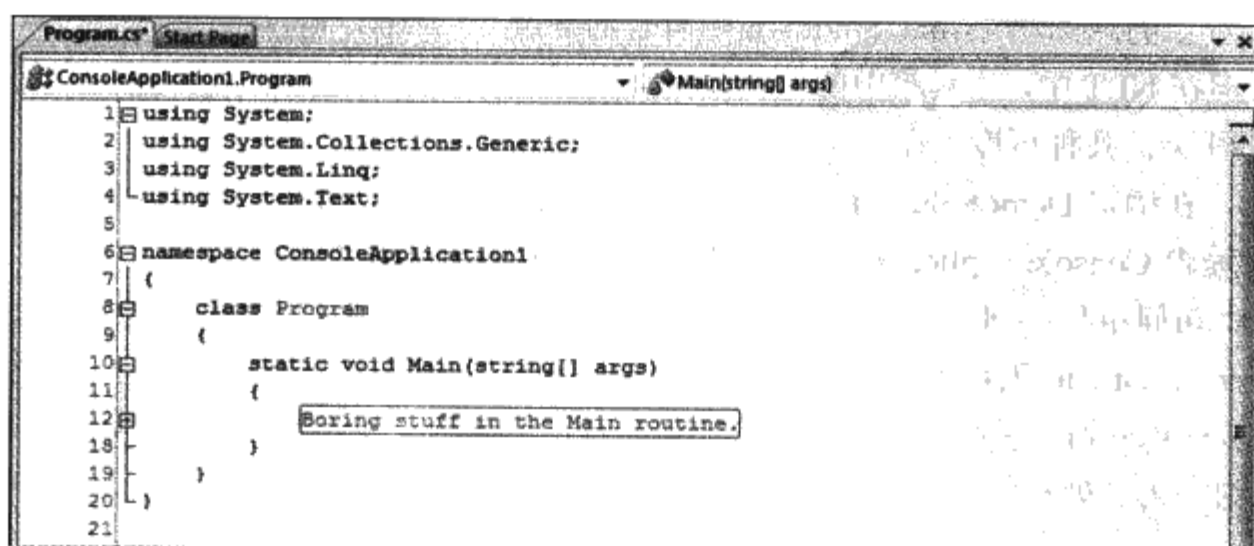


图 15-16

除了可折叠的编辑器之外，Visual Studio 的代码编辑器还拥有 Visual Studio 6 的全部功能，特别是它支持 IntelliSense 功能，这不仅减少了键入代码的工作量，还可以确保输入正确的参数。C++开发人员会注意到，Visual Studio 的 IntelliSense 功能比 Visual Studio 6 的更强大，速度也更快。IntelliSense 功能在 Visual Studio 2008 中也得到了进一步的改进。它的智能化更强，可以记住用户喜欢的选项，并从这些选项开始，而不是从该功能提供的有时很长的列表的开头开始。

代码编辑器也对代码进行一些语法检查，在编译代码前用短波浪线划出大多数语法错误。把鼠标放在有下划线的文本上面，就会弹出一个文本框，解释错误。这个功能 Visual Basic 开发人员已使用了多年，叫做设计期间的调试功能，现在 C#和 C++开发人员也可以使用它了。

2. 其他窗口

除了代码编辑器外，Visual Studio 还提供了许多其他窗口，允许用户以不同的角度查看项目。

注意：

本节的其余部分将介绍许多其他的窗口。如果某个窗口在屏幕上不可见，可以进入 View 菜单，单击合适窗口的名称。要显示设计视图和代码编辑器，可以在 Solution Explorer 中右击文件名，然后从弹出的窗口中选择 View Designer 或 View Code，也可以从 Solution Explorer 顶部的工具栏中选择。设计视图和代码编辑器共用同一个窗口。

(1) Design View 窗口

如果设计一个用户界面应用程序，例如 Windows 应用程序、Windows 控件库或者 ASP.NET 应用程序，就可以使用设计视图(Design View)窗口，它会显示窗体的整体外观。设计视图窗口一般和工具箱窗口一起使用。工具箱包含许多可以拖放到程序中的.NET 组件，如图 15-17 所示。

应用工具箱的规则与 Visual Studio 6 中所有的开发环境一样，但在.NET 中，工具箱中的组件数量大大增加了。组件的种类在某种程度上取决于用户所编辑项目的类型。例如，在编辑 DemoSolution 解决方案中的 WindwosApplication1 项目时，可以使用的组件种类就比编辑 ConsoleApplication1 项目时多。最重要的组件种类如下：

- 数据访问组件：可以连接数据源，管理它们包含的数据。这类组件可处理 Microsoft SQL Server、Oracle 和 OleDb 数据源。
- Windows Forms 控件(标记为常用控件)：表示可视化控件，例如文本框、列表框和树形视图，用于处理胖客户应用程序。

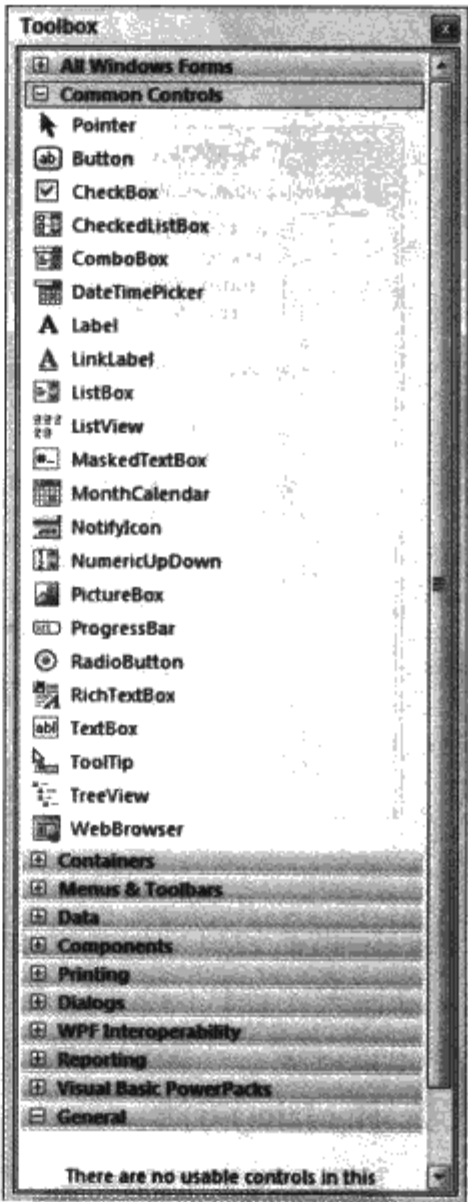


图 15-17



- Web Forms 控件(标记为标准):基本上与 Windows 控件的作用一样,但用于 Web 浏览器,把模拟控件的 HTML 输出结果发送给浏览器(只有使用 ASP.NET 应用程序时才能看到该结果)。
- 杂项组件:在机器上执行各种有用任务的.NET 类,例如连接目录服务或事件日志。

也可以把自己定制的组件类别添加到工具箱中,方法是右击任一类别,从弹出的菜单中选择 Add Tab。从该菜单中选择 Choose Items,就可以把其他工具放在工具箱中。这非常适合于添加自己喜欢的 COM 组件和 ActiveX 控件。在默认情况下,COM 组件和 ActiveX 控件不会显示在工具箱中。如果要添加一个 COM 控件,可以单击它,并拖放到项目上,就像操作.NET 控件一样。Visual Studio 会自动添加所有必需的 COM 交互操作代码,以便项目调用该控件。此时,添加到项目中的实际上是 Visual Studio 在后台创建的一个.NET 控件,它是所选 COM 控件的容器。

#### 注意:

C++开发人员可能会把工具箱当作资源编辑器的 Visual Studio 版本(有较大改进)。Visual Basic 开发人员刚开始不会认为该工具箱是新增的内容,因为在 Visual Studio 6 中就有工具箱。但是,Visual Studio 中的工具箱对源代码的作用与 Visual Basic6 IDE 中的工具箱有显著的不同。

为了说明工具箱如何工作,把一个文本框放在基本窗体项目上。首先单击工具箱中的 TextBox 控件,再单击一次,把它放在设计视图的窗体上(也可以直接把它拖放到设计界面上)。设计视图如图 15-18 所示,该图也是编译并运行 WindowsApplication1 项目的结果。

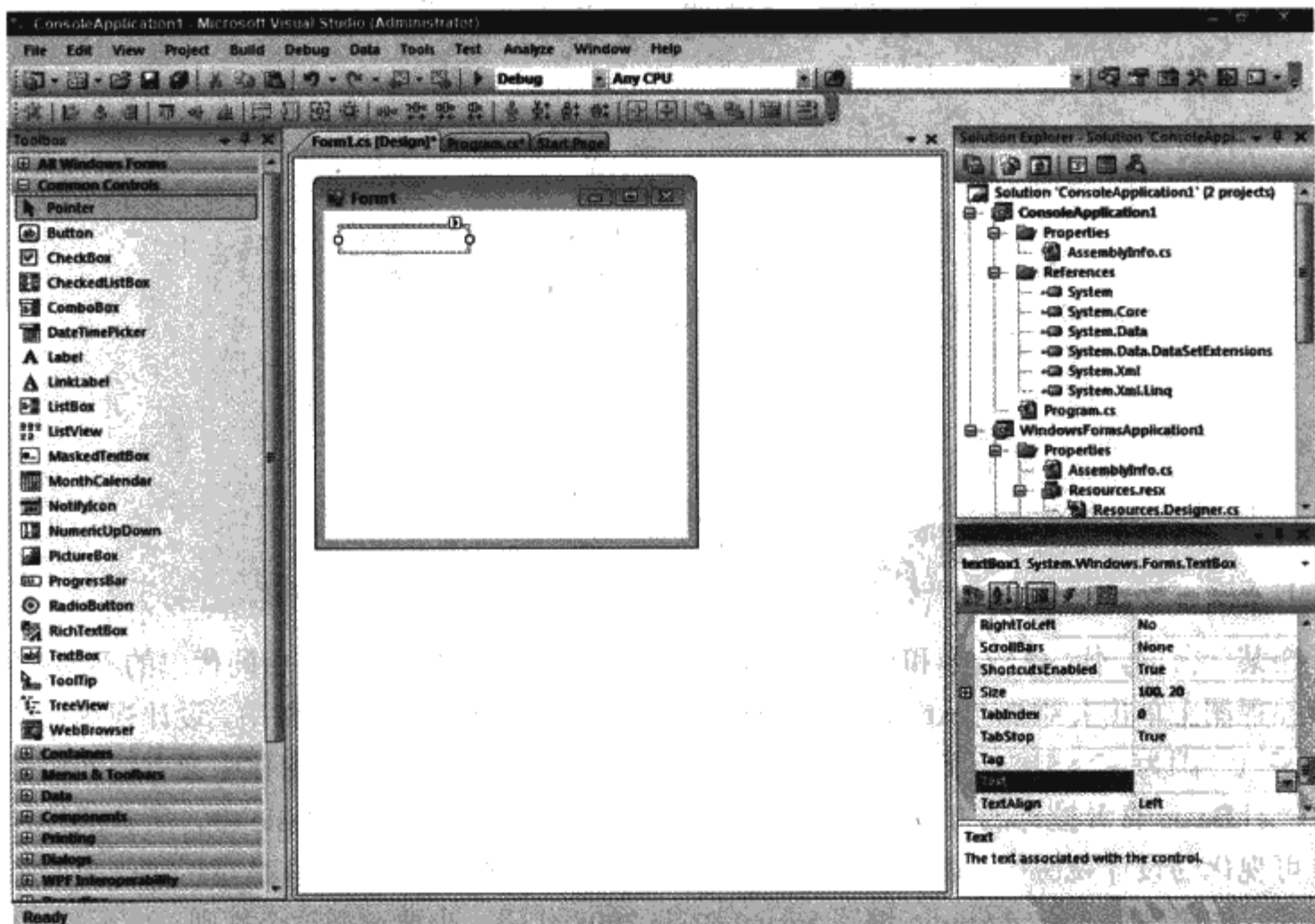


图 15-18

在窗体的代码视图上，Visual Studio 2008 并没有像 IDE 的以前版本那样，实例化放在窗体上的 TextBox 对象。单击 Visual Studio Solution Explorer 中 Form1.cs 文件旁边的加号，会看到一个文件 Form1.Designer.cs，它用于窗体及其中控件的设计。在这个类文件中，Form1 类有一个新的成员变量：

```
public class Form1
{
    private System.Windows.Forms.TextBox textBox1;
```

在方法 InitializeComponent() 中还有一些初始化代码，该方法从 Form1 构造函数中调用：

```
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InitializeComponent()
{
    this.textBox1 = new System.Windows.Forms.TextBox();
    this.SuspendLayout();
    //
    // textBox1
    //
    this.textBox1.Location = new System.Drawing.Point(0, 0);
    this.textBox1.Name = "textBox1";
    this.textBox1.Size = new System.Drawing.Size(100, 20);
    this.textBox1.TabIndex = 0;
    //
    // Form1
    //
    this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
    this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
    this.ClientSize = new System.Drawing.Size(284, 264);
    this.Controls.Add(this.textBox1);
    this.Name = "Form1";
    this.Text = "Form1";
    this.ResumeLayout(false);
    this.PerformLayout();
}
```

在某一方面，代码编辑器和设计视图是没有区别的，它们只是呈现了相同代码的不同视图。在设计视图上单击并添加 TextBox 时，编辑器会把上述代码放在 C# 源文件中。设计视图反映了这个变化，因为 Visual Studio 可以读取源代码，确定在应用程序启动时窗体中会有哪些控件。与 Visual Basic 查看控件的方式相比，这是一个重要的改变，在 Visual Basic 中，所有的控件都放在可视化的设计视图中。现在，由 C# 源代码控制应用程序，设计视图只是显示源代码的另一种方式。如果使用 Visual Studio 编写 Visual Basic 2008 代码，也要遵循这个规则。

还可以采用另外一种方式，如果把上述代码手工添加到 C# 源文件中，Visual Studio 也会自动从代码中检测到应用程序中有一个文本框控件，并会在设计视图的指定位置显示它。最好可

可视化地添加这些控件，让 Visual Studio 处理最初生成的代码，单击鼠标两次要比键入好几行代码快得多，也不容易出错！

可视化添加这些控件的另一个原因是为了确定应用程序中有这些控件，Visual Studio 需要使相关的代码遵循某些条件——手工编写的代码可能不遵循这些条件。特别是 `InitializeComponent()` 方法包含初始化文本框的代码，在它的注释中警告用户不要修改代码。这是因为 Visual Studio 要查找这个方法，以确定应用程序在启动时有哪些控件。如果在代码的其他地方创建和定义了一个控件，Visual Studio 不知道有这个控件，因此就不能在设计视图或其他窗口中编辑它。

实际上，无论有什么警告，都可以在 `InitializeComponent()` 中修改代码，但应非常小心。例如，修改一些属性的值一般不会出什么问题，如让某个控件显示不同的文本，或者给它设置另一个大小。实际上，开发人员工作室非常擅长于处理在这个方法中添加的其他代码。但要注意，如果对 `InitializeComponent()` 进行了过多的修改，Visual Studio 就有可能识别不出使用代码添加的控件。在编译代码时，这不会影响到应用程序，但可能禁用 Visual Studio 为这些控件提供的某些编辑功能。因此，如果要进行其他重要的初始化，最好在 `Form1` 构造函数或其他方法中进行。

## (2) 属性窗口

这是从旧 Visual Basic IDE 继承而来的另一个窗口。本书的第一部分说过，.NET 类可以执行属性。实际上，如第 31 章(讨论 Windows 窗体的建立)所述，表示窗体和控件的 .NET 基类有许多定义其操作和外观的属性。例如 `Width`、`Height`、`Enabled`(用户是否可以给该控件键入信息)和 `Text`(控件所显示的文本)，Visual Studio 知道其中的许多属性。对于 Visual Studio 能通过读取源代码检测到的控件来说，属性窗口可以显示和编辑大多数属性的初值，如图 15-19 所示。

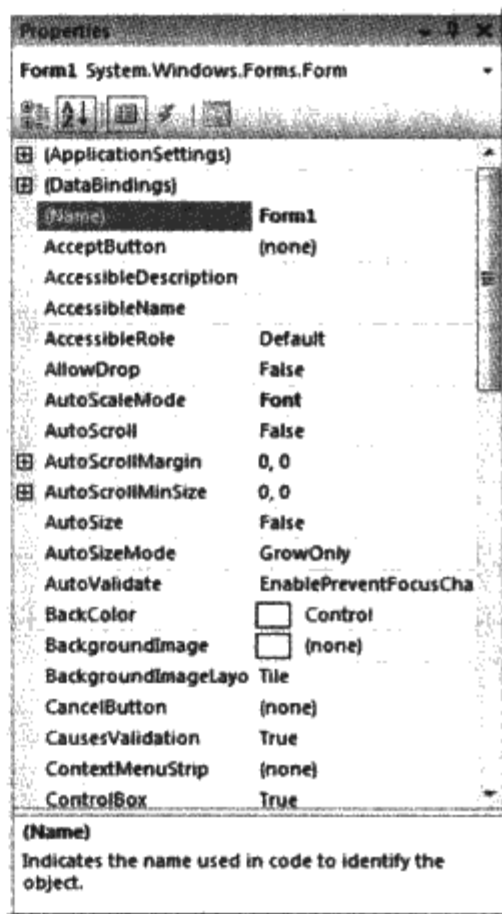


图 15-19

### 注意：

属性窗口也可以显示事件。单击窗口顶部的闪电图标，就可以查看 IDE 中当前选中的事件，或者查看在属性窗口的下拉列表中选择的事件。

在属性窗口的顶部，有一个列表框，从中可以选择要查看的控件。在本章的这个例子中，我们选择的是 `Form1`，即 `WindowsApplication1` 项目的主窗体类，将其文本编辑为“Basic Form-Hello!”。如果此时查看源代码，就会看到刚才的操作实际上是通过一个友好的用户界面编辑了源代码：

```
this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
this.AutoScaleMode = System.Windows.Forms.AutoScaleModeMode.Font;
this.ClientSize = new System.Drawing.Size(284, 264);
```

```
this.Controls.Add(this.textBox1);  
this.Name = "Form1";  
this.Text = "Basic Form - Hello";  
this.ResumeLayout(false);  
this.PerformLayout();
```

并不是属性窗口中的所有属性都会在源代码中显式指定。对于没有显式指定的属性，Visual Studio 会显示在创建窗体时给它们设置的默认值，这些默认值是在初始化窗体时设置的。显然，如果在属性窗口中修改这些属性的值，源代码中就会出现一个显式设置该属性的语句，反之亦然。有趣的是，如果属性是从其初始值改变而来，这个属性在属性窗口的列表框中就会显示为黑体。有时双击属性窗口中的属性，会返回其初始值。

属性窗口提供了一种查看控件或窗口的外观和属性的简便方式。

#### 注意：

属性窗口实现为一个 `System.Windows.Forms.PropertyGrid` 实例，该实例在内部使用第 13 章介绍的反射技术，来标识要显示的属性和属性值。

### (3) 类视图窗口

与属性窗口不同，类视图窗口最初是从 C++(或 J++)开发环境继承而来的一个窗口，如图 15-20 所示。它对于 Visual Basic 开发人员来说是新的，因为 Visual Basic 6 不支持类的概念，而使用 COM 组件。Visual Studio 实际上并没有把类视图看作是一个窗口，而是 Solution Explorer 的一个附加选项卡。默认情况下，类视图不会显示在 Visual Studio Solution Explorer 中。要打开类视图，应选择 View | Class View。类视图如图 15-20 所示，显示了代码中命名空间和类的层次结构，给出了一个树形视图，用户可以展开该视图，以查看哪个命名空间包含了什么类，类包含了什么成员。

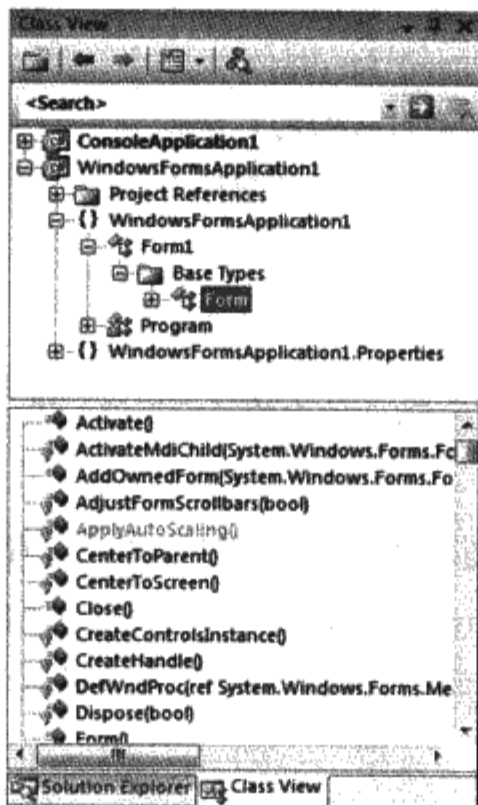


图 15-20

类视图的一个特性是，如果右击在源代码中可以访问的任一项，弹出菜单就会提供一个选



项 Go To Definition, 单击它, 就可以访问代码编辑器中该项的定义。还可以在类视图中双击该项(实际上是在源代码编辑器中右击需要的项, 从打开的弹出菜单中选择相同的选项)来完成同样的操作。弹出菜单还允许给类添加字段、方法、属性或索引器。这意味着可以在一个对话框中指定相关成员的信息, 开发环境会自动添加对应的代码。这对于字段和方法可能不太有效, 因为它们可以快速添加到代码中, 但对于属性和索引器来说就非常有用, 因为它可以减少大量的键入工作。

#### (4) 对象浏览器

在.NET 环境中编程的一个重要优点是可以确定程序集中引用的基类和其他库中有什么方法和其他代码项。这个功能是通过对象浏览器来获得的。在 Visual Studio 2008 的 View 菜单中选择 Object Brower, 就可以访问这个窗口。

对象浏览器非常类似于类视图窗口, 它也显示一个树形视图, 该树形视图给出了应用程序的类结构, 允许查看每个类的成员。用户界面则略有不同, 因为它在一个单独的窗格中显示类成员, 而不是在树形视图中显示。但真正的区别是它不仅可以查看项目中的命名空间和类, 还可以查看项目所引用的所有程序集中的命名空间和类。图 15-21 显示了利用对象浏览器查看.NET 基类中的 SystemException 类的情况。

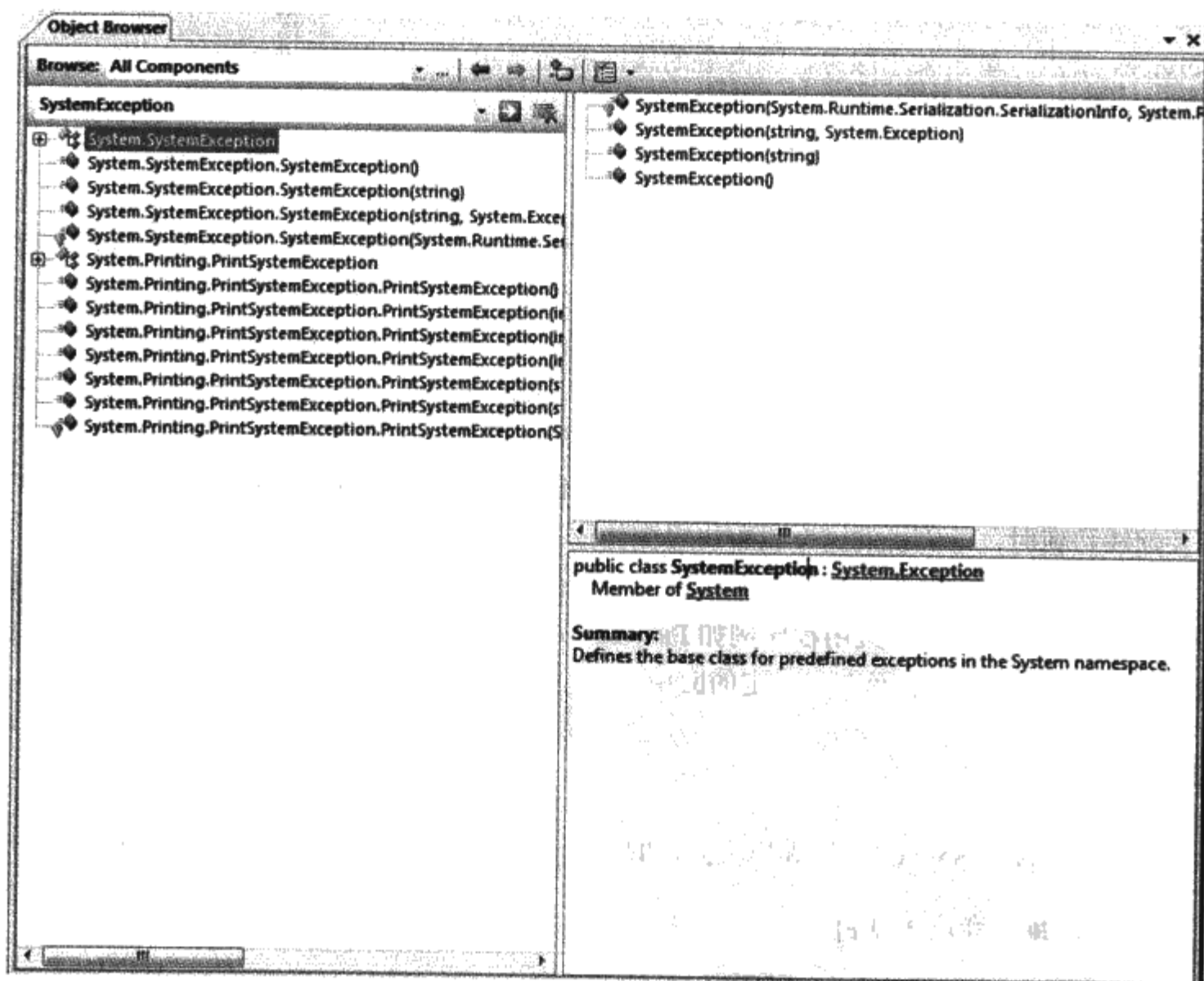


图 15-21

在对象浏览器中必须注意的一点是, 它先按照类所在的程序集对类进行分组, 再按照命名



空间对类进行分组。因为基类的命名空间常常分布在多个程序集中，所以在定位某个类时可能会遇到麻烦，除非知道该类在哪个程序集中。

对象浏览器可以查看 .NET 对象。如果由于某些原因要查看已安装的 COM 对象，就可以使用以前在 C++ IDE 中使用的 OLEView 工具，它在文件夹 C:\Program Files\Microsoft SDKs\Windows\v6.0A\Bin 中，该文件夹还有几个其他类似的工具。

注意：

Visual Basic 开发人员不应把 .NET 对象浏览器和 Visual Basic 6 IDE 的对象浏览器混为一谈，.NET 对象浏览器可以查看 .NET 类，而 Visual Basic 6 中的对象浏览器可以查看 COM 组件。如果要使用旧对象浏览器的功能，现在就应使用 OLEView 工具。

### (5) 服务器浏览器

服务器浏览器可以用于确定编码时网络中计算机各个方面的情况，如图 15-22 所示。

从上面的屏幕图中可以看出，可以通过服务器浏览器访问数据库连接、服务信息和事件日志的信息。

服务器浏览器与属性窗口链接在一起，所以，如果打开 Services 节点，单击某个服务，该服务的属性就会显示在属性窗口中。

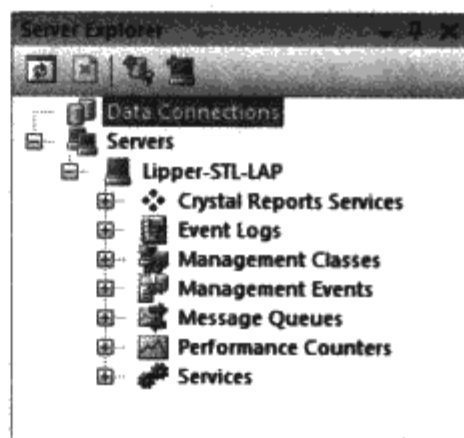


图 15-22

### 3. pin 按钮

在浏览 Visual Studio 时，前面介绍的许多窗口包含一些有趣的功能，很容易让人回想起工具栏的一些功能。除了代码编辑器外，它们都是固定的。另一个功能是在这些窗口处于浮动状态时，在每个窗口右上角的最小化按钮的旁边就会出现一个大头针图标，这个图标的工作方式就跟大头针一样，把窗口钉住。在窗口被钉住时(大头针垂直显示)，它们的操作就像一般的窗口一样。但当它们没有被钉住时，(大头针水平显示)，只要它们获得焦点，就会打开；而失去焦点时(用户单击了其他地方)，它们就会无声无息地退到 Visual Studio 应用程序的主边界上(打开和关闭该按钮时，计算机的速度也有快慢变化)。

钉住或不钉住窗口是充分利用屏幕上有限空间的另一种方式。以前在 Windows 中这个功能用得不多，但在一些第三方应用程序例如 PaintShop Pro 中就使用了类似的概念。钉住的窗口在许多基于 Unix 的系统上已经有了一定的应用。

## 15.1.6 生成项目

本节介绍 Visual Studio 为生成项目提供的选项。

### 1. 生成、编译和产生项目

在介绍各种生成选项前，首先澄清一个术语。在把源代码变成某类可执行的代码时，常常会使用 3 个不同的术语：编译、生成和产生。这三个术语的出现是因为直到最近，把源代码变成可执行代码的过程需要多个步骤(在 C++ 中仍是这样)。这在很大程度上是因为一个程序常常

包含许多源文件，例如在 C++ 中，每个源文件都需要单独编译。这样就生成了对象文件，每个对象文件都包含一些可执行代码，但每个对象文件都仅与一个源文件相关。为了生成可执行代码，这些对象文件必须链接在一起，这个过程就称为链接。把过程组合起来通常称为(至少在 Windows 平台上)生成代码。但是，在 C# 中，编译器更为专业，能够把所有的源文件当作一个块来读取和处理。因此，就没有独立的链接阶段，在 C# 中，“编译(compile)”和“生成(build)”可以互换。

术语“产生(make)”的基本含义与“生成(build)”相同，但在 C# 中一般不使用。该术语来源于旧的大型计算机系统，在该系统上，当项目由许多源文件组成时，就编写一个独立的文件，其中包含的指令可以指示编译器如何生成项目：包括哪些文件，链接什么库等，这个文件一般称为产生文件(make file)，该文件仍然是 Unix 上的标准。产生文件在 Windows 上一般不需要，但如果用户需要，也可以编写它们(或者让 Visual Studio 生成它们)。

## 2. 调试和发布项目

有 C++ 背景的开发人员都知道生成文件的概念，但 Visual Basic 开发人员就不一定知道了。在调试程序时，可执行代码进行的操作一般与实际发布该软件时所进行的操作大不相同，在发布时，除了代码正常工作外，可执行文件的尺寸应尽可能小，运行速度应尽可能快。但这些要求与调试代码时的要求并不相容。原因如下：

### (1) 优化

要获得高性能，部分依赖于编译器对代码进行的许多优化，即编译器在编译过程中一直在监视源代码，找出某些代码，以一种不影响全局效果的方式修改它们，使其效率更高。例如，如果编译器遇到下面的源代码：

```
double InchesToCm(double Ins)
{
    return Ins*2.54;
}

// later on in the code

Y = InchesToCm(X);
```

就可以用下面的代码替换它：

```
Y = X * 2.54;
```

或者把下面的代码：

```
{
    string Message = "Hi";
    Console.WriteLine(Message);
}
```

替换成：

```
Console.WriteLine("Hi");
```

因此，在过程中不必声明不必要的对象引用。

不能说 C# 编译器进行了什么优化工作，上面的两个例子在任何程序中都有可能出现，因为这类信息没有加以说明(对于托管语言如 C# 来说，上述优化很可能在 JIT 编译期间进行，而不是在 C# 编译器把源代码编译成程序集时进行)。从商业角度来看，编写编译器的公司通常不会对编译器使用的技巧进行过多的说明。这里还要强调，优化不影响源代码，它们只影响可执行代码的内容。但是，通过上面的例子，您应明白优化的内涵。

问题是，像上面例子中的优化可以使代码运行得更快，但它们对于调试来说，就没什么好处了。假定在第一个例子中，要在 `InchesToCm()` 方法内部设置一个断点，看看其中发生了什么。如果可执行代码没有 `InchesToCm()` 方法，因为编译器已删除了它，该怎么办？如果在 `Message` 变量上设置了一个监视点，但在编译好的代码中根本没有这个变量，该怎么办？

### (2) 调试程序符号

在调试时，常常需要查看变量的值，因此需要通过该变量在源代码中的名称来指定它们。问题是可执行代码一般不包含这些变量的名称——编译器用内存地址取代了它们。NET 对这种情形进行了一定的改进。程序集中的某些对象是与其名称一起存储的，但只有小部分对象是这样，例如公共类和方法。这些名称仍在程序集进行 JIT 编译时被删除。如果在调试程序检查可执行代码时，要求调试程序给出变量 `HeightInInches` 的值，也不能得到我们希望的结果，它只能给出地址，根本就没有 `HeightInInches` 引用。因此，为了调试正确，用户必须在可执行代码中添加额外的调试信息。这些信息包括变量名和代码行号，让调试程序查找与源代码指令对应的可执行机器汇编语言指令。但不能在发布版本中放置这些信息，因为这是商业机密(调试信息可以让更多的人反汇编源代码)，而且会增大可执行文件。

### (3) 额外的源代码调试命令

在调试时，我们经常会遇到这样一个相关的问题：代码中有额外的命令行显示与调试相关的重要信息。显然，在发布软件前，必须从可执行代码中完全删除这些相关命令。可以手工完成这个任务，但如果仅是以某种方式标记这些语句，让编译器在编译代码时忽略它们，再发布软件，不是更容易吗？本书的第一部分已经讨论过，在 C# 中，可以定义一个合适的处理器符号，再把它和 `Conditional` 属性结合在一起使用，这就是所谓的条件编译。

与最终销售的产品相比，即使把这些特性都加上，编译所有的商业软件和调试它们也是有细微差别的。Visual Studio 可以把这些都考虑进去，因为如前所述，它存储了编译代码时传递给编译器的所有信息，为了找出不同类型的生成文件，Visual Studio 只需要存储多套这类信息。不同的生成信息就称为配置。在创建一个项目时，Visual Studio 会自动提供两种配置，分别是 `Debug` 和 `Release`：

- 调试配置：通常会指定不进行任何优化，在可执行代码中给出额外的调试信息，编译器假定给出了调试预处理器符号 `Debug`，除非在源代码中显式指定了 `#undefined`。
- 发布配置：通常指定编译器要优化，在可执行代码中没有额外的调试信息，编译器假定没有给出任何调试预处理器符号。

也可以定义自己的配置。例如，建立专业级和企业级的生成版本，以发布两个版本的软件。过去，因为 Windows NT 支持 Unicode 字符编码，而 Windows 95 不支持，所以通常 C++ 项目有一个 Unicode 配置和一个 MBCS(多字节字符集)配置。

### 3. 选择配置

一个很明显的问题是，因为 Visual Studio 存储了多个配置的信息，在生成一个项目时，该如何确定使用哪个配置？答案是总是有一个现行配置，在要求 Visual Studio 生成项目时，就使用这个配置(注意配置是每个项目的配置，而不是解决方案的配置)。

在默认情况下，创建一个项目时，调试配置就是现行配置。单击 Build 菜单选项，选择 Configuration Manager 项，就可以改变现行配置。还可以通过 Visual Studio 工具栏中的下拉菜单来改变现行配置。

### 4. 编辑配置

除了选择现行配置外，还可以查看和编辑配置，为此，需要在 Solution Explorer 中选择相应的项目，然后在 Project 菜单中选择 Properties，打开一个非常复杂的对话框(在 Solution Explorer 中右击项目名，在弹出的菜单中选择 Properties，也可以打开这个对话框)。

该对话框包含一个树形视图，在该树形视图中可以选择许多不同的区域，以查看或编辑。这里不详细介绍所有的区域，只介绍其中两个最重要的区域。

图 15-23 显示了某个应用程序的可用属性的选项卡视图。这个屏幕图显示了本章前面创建的 ConsoleApplication1 项目的一般应用程序设置。

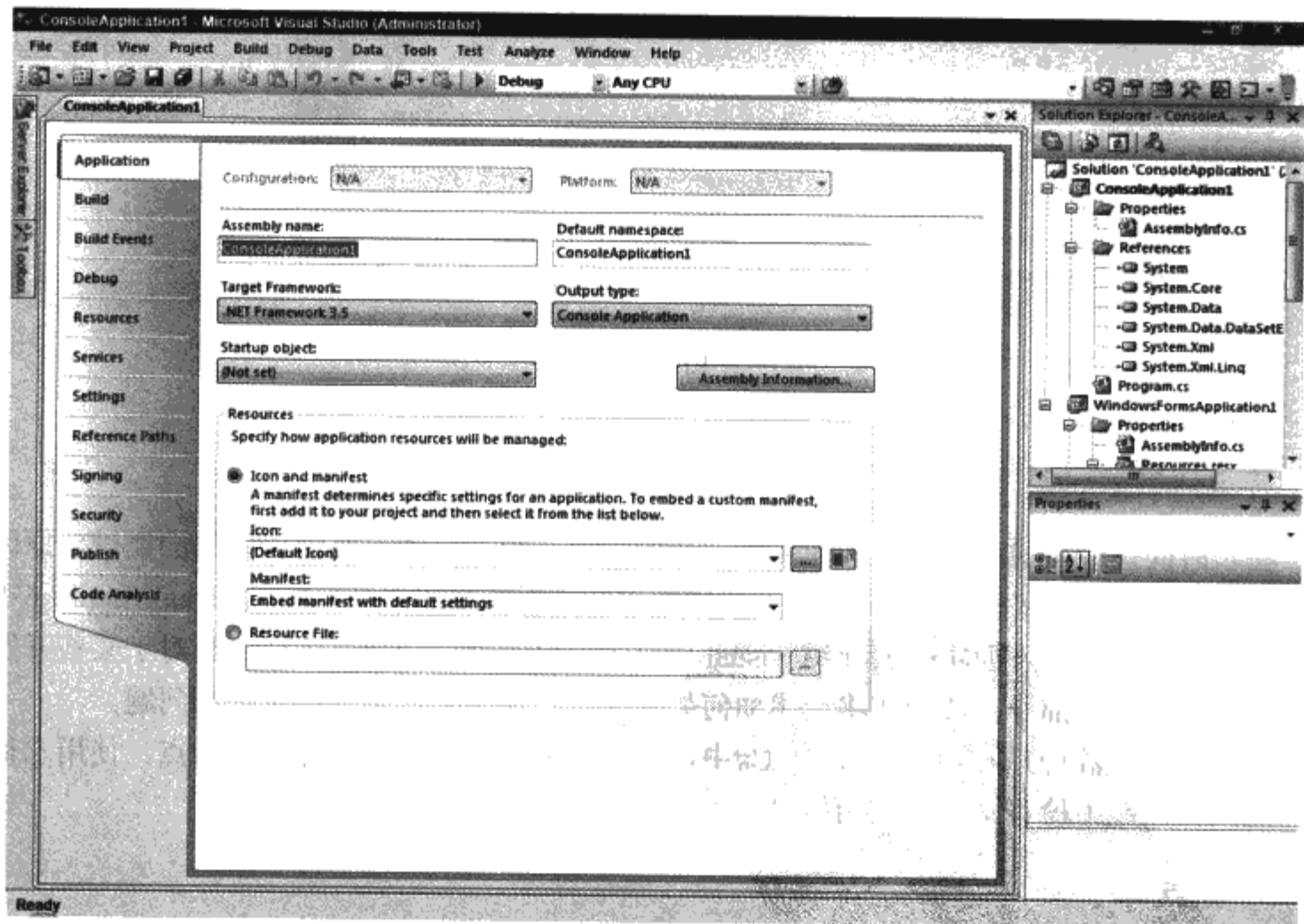


图 15-23

注意，可以选择要生成的程序集名和程序集的类型。其中的选项是控制台应用程序、Windows 应用程序和类库。当然，还可以改变程序集的类型(尽管这还有争议，但在 Visual Studio 最初生成项目时，为什么不能选择正确的项目类型呢？)。



屏幕图 15-24 显示了生成文件的配置属性。注意，对话框顶部的列表框允许指定要查看的配置。此时可以查看调试配置——假定编译器定义了 DEBUG 和 TRACE 预处理器符号。如上所述，在调试配置中，代码没有优化，并会生成额外的调试信息。

一般情况下，用户不需要调整配置设置。如果需要使用它们，了解不同配置属性的区别是非常有用的。

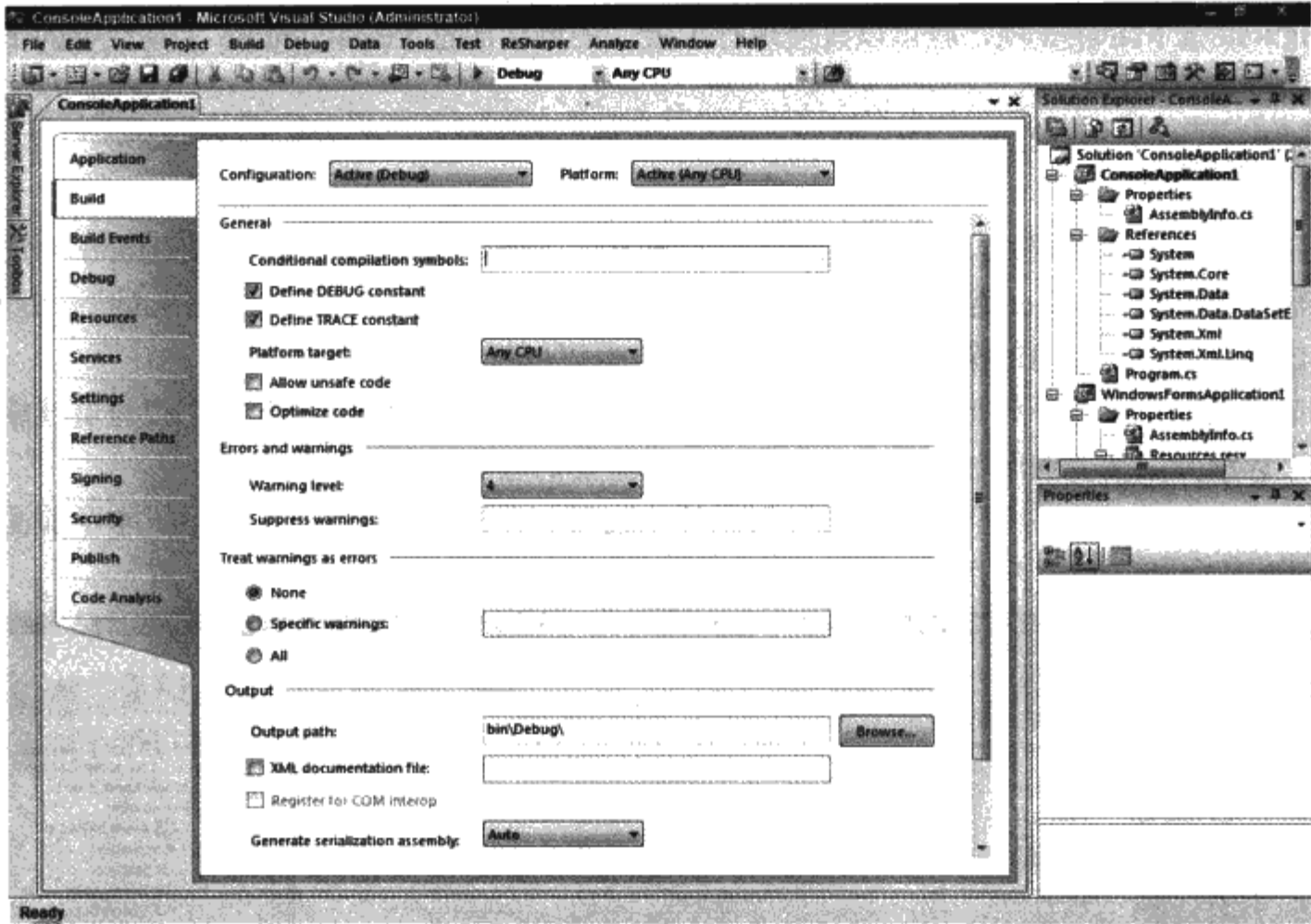


图 15-24

15.1.7 调试

在长时间讨论生成项目和生成配置后，我们还没有用大量的篇幅讨论代码的调试。原因是在 Visual Studio 中调试的规则和过程——设置断点和查看变量的值——与在 Visual Studio 6 IDE 中没有太大的区别。下面将简要介绍 Visual Studio 提供的调试功能，主要讨论对于某些开发人员来说是新内容的部分，我们还将论述如何处理异常，因为它们会给调试带来问题。

与.NET 出现以前的语言一样，在 C#中，调试所涉及的主要技术是设置断点，使用它们在代码的执行过程中检查某处发生的情况。

1. 断点

在 Visual Studio 中，可以在执行的代码中给任意一行设置断点。最简单的方式是在代码编辑器中单击该行，即在文档窗口左边的阴影区域中单击该行(或者选择该行，按下 F9 键)，这样，就在该行设置了一个断点，只要代码执行到该行，就会中断，把控制权交给调试程序。在 Visual Studio 的以前版本中，断点在代码编辑器中用该行左边的一个大圆表示。Visual Studio 则把该



该行的文本和背景用另一种颜色来突出显示。再次单击该圆，就会删除断点。

如果程序并不适合于每次执行一行就中断一次，也可以设置条件断点。为此，可以单击 **Debug | Windows | Breakpoints** 菜单项，弹出一个对话框，该对话框要求用户给出要设置的断点信息，可以使用的选项有：

- 指定只有在对设置了断点的代码执行一定次数后，才能中断程序的执行。
- 指定执行某行一定的次数后，断点才起作用，例如执行该行 20 次后，断点才起作用(可用于调试大循环)。
- 给相关变量设置断点，而不是给指令设置断点。此时，监视的是变量值，只要变量的值发生了改变，就会触发断点。但是，使用这个选项会显著减慢代码的运行速度。在指令执行完后检查变量是否改变，会大大增加处理器时间。

## 2. 监视点

遇到断点时，通常要查看变量的值。最简单的方式是在代码编辑器中，把鼠标指针放在该变量名上，此时会显示一个小方框，其中给出了该变量的值。还可以扩展该方框，显示更详细的内容，如图 15-25 所示。

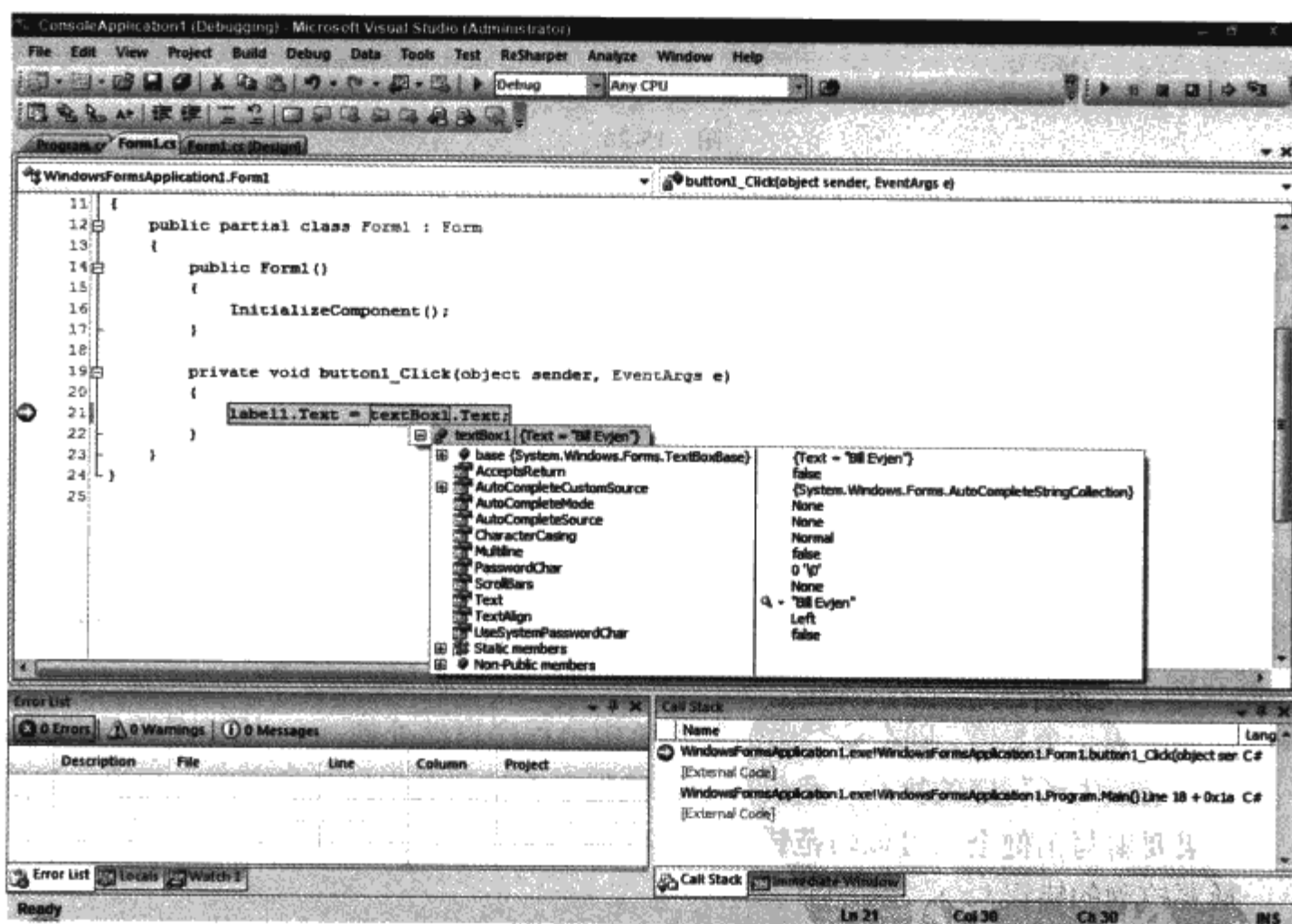


图 15-25

也可以使用监视窗口来查看变量的内容，它是一个带有标签的窗口，程序只有在调试器中运行时，该对话框才会出现，如图 15-26 所示。如果该对话框没有打开，可以选择 **Debug | Windows | Autos**。

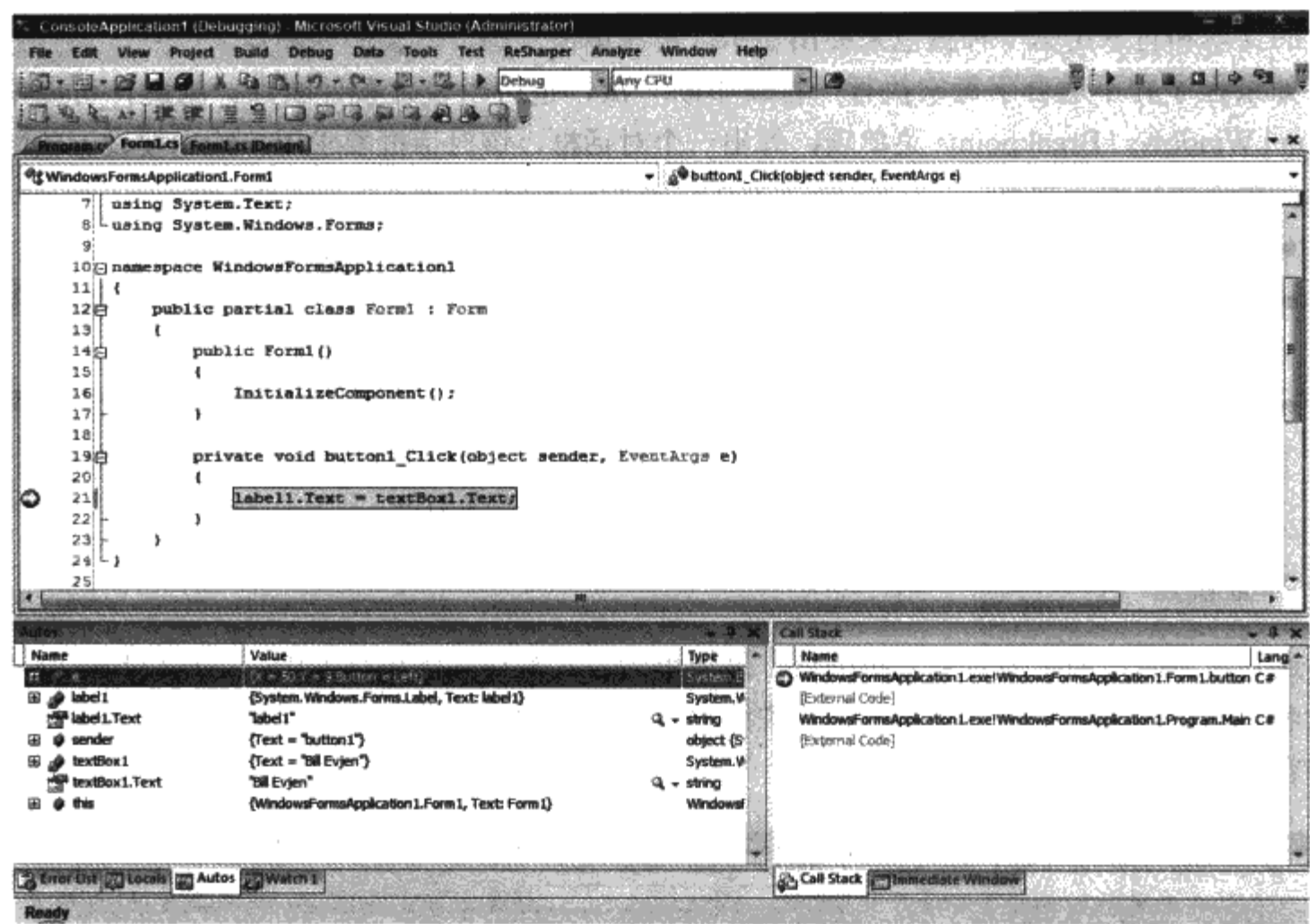


图 15-26

- 类变量或结构变量的旁边有一个+图标，单击它可以展开变量，查看其字段的值。这个窗口的 3 个选项卡主要用于监视不同的变量：
- Autos 监视程序运行时最后访问的变量。
  - Locals 监视当前执行的方法中的变量。
  - Watch 监视用户在 Watch 窗口中键入的变量。

3. 异常

应用程序中的异常可以确保在该程序中以合适的方式处理错误情况。它们可以保证应用程序正常运行，不会给用户显示许多技术性的对话框。但在调试时，异常并没有那么强大的功能，这个问题有两个方面：

- 如果产生一个异常，则在调试时常常不希望由程序自动处理——有时处理异常就意味着退出，中断程序的执行！我们要让调试程序查找到发生异常的原因。当然，问题是如果要编写出健全的可以防范错误的好代码，程序就应能处理几乎所有的异常——包括要检测的错误！
- 如果产生了一个没有编写处理程序的异常，.NET 运行库还是会查找该异常的处理程序。此时它发现没有这样的处理程序，因而中断程序。这里没有调用堆栈，不能查看变量的值，因为它们都已出了作用域。

当然，可以在 catch 块中设置断点，但这常常没有什么帮助，因为在执行 catch 块时，按照定义，程序流会退出相应的 try 块，这样，要查看的变量值就出了作用域，找不出错误发生的原因。甚至不能查看堆栈跟踪，找出 throw 语句退出程序时执行了哪个方法，因为该方法已交

出了控制权。当然，在 throw 语句上设置断点可以解决这个问题，但如果代码中有许多 throw 语句时，该如何编码？如何告诉编译器是哪个 throw 语句抛出了异常？

实际上，Visual Studio 给这些问题提供了一个非常好的解决方案。查看一下 Debug 主菜单，其中有一个 Exceptions 菜单项，选择它会弹出 Exceptions 对话框，如图 15-27 所示，指定抛出异常时要执行什么操作。可以选择继续执行或者自动停止，开始调试代码，此时程序停止执行，调试程序开始调试 throw 语句。

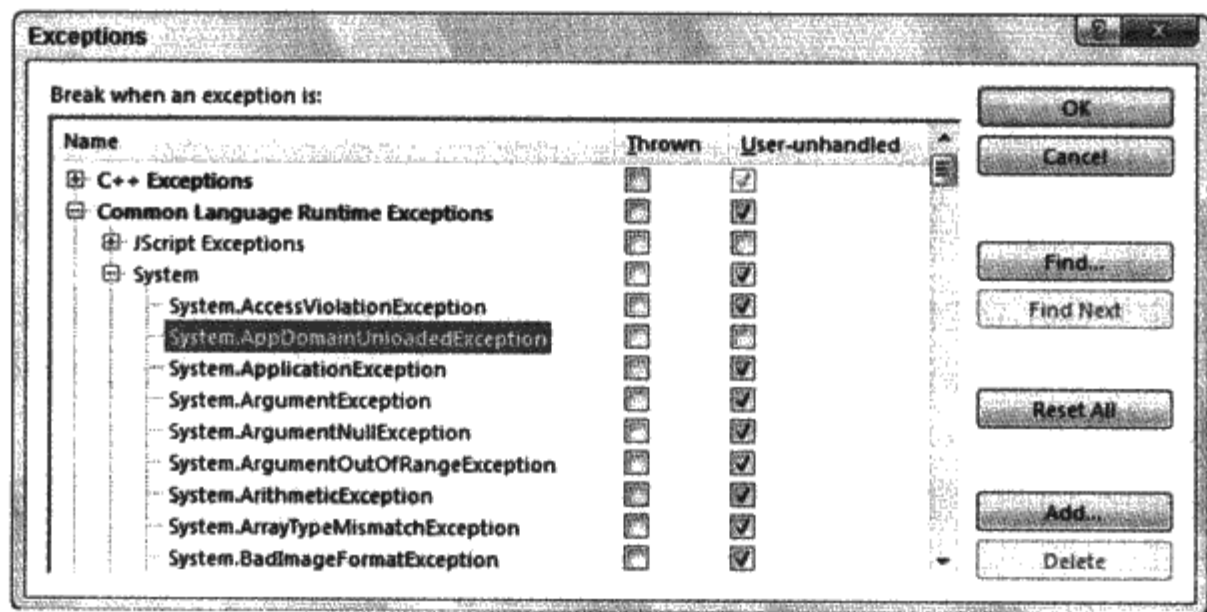


图 15-27

这个工具的强大之处是用户可以根据抛出了哪个类的异常，来定制程序的行为。例如，在图 15-27 上，告诉 Visual Studio 在遇到由任一个 .NET 基类抛出的异常时，就中断执行，开始调试程序，但如果抛出了 AppDomainUnloadedException 异常，则不中断执行。

Visual Studio 知道 .NET 基类中所有的异常类，也知道在 .NET 环境外部抛出的许多异常。Visual Studio 不会自动响应用户编写的定制异常类，但用户可以把自已的异常类手工添加到该列表中，指定哪个异常会立即停止执行程序。为此，只需单击图 15-27 中的 Add 按钮(在从树形结构中选择一个顶级节点，该按钮就成为可用的)，键入异常类的名称即可。

## 15.2 修订功能

许多开发人员在第一次为应用程序开发功能后，会改写应用程序，使之更易管理，可读性更高。这个过程称为修订。修订就是改写代码，提高可读性、性能，提供类型安全性，使应用程序更好地遵循标准 OO(面向对象)编程规则的过程。

因此，Visual Studio 2008 的 C# 环境现在包含一组修订工具，这些工具位于 Visual Studio 菜单的 Refactoring 选项下。为了说明这些工具的作用，下面在 Visual Studio 中创建一个新类 Car:

```
using System;
using System.Collections.Generic;
using System.Text;
```

```
namespace ConsoleApplication1
{
    public class Car
    {
        public string _color;
        public string _doors;

        public int Go()
        {
            int speedMph = 100;
            return speedMph;
        }
    }
}
```

现在，假定修订时希望修改代码，把\_color 和\_door 变量封装到.NET 公共属性中。Visual Studio 2008 的修订功能可以在文档窗口中右击这些属性，选择 Refactor | Encapsulate Field。打开 Encapsulate Field 对话框，如图 15-28 所示。

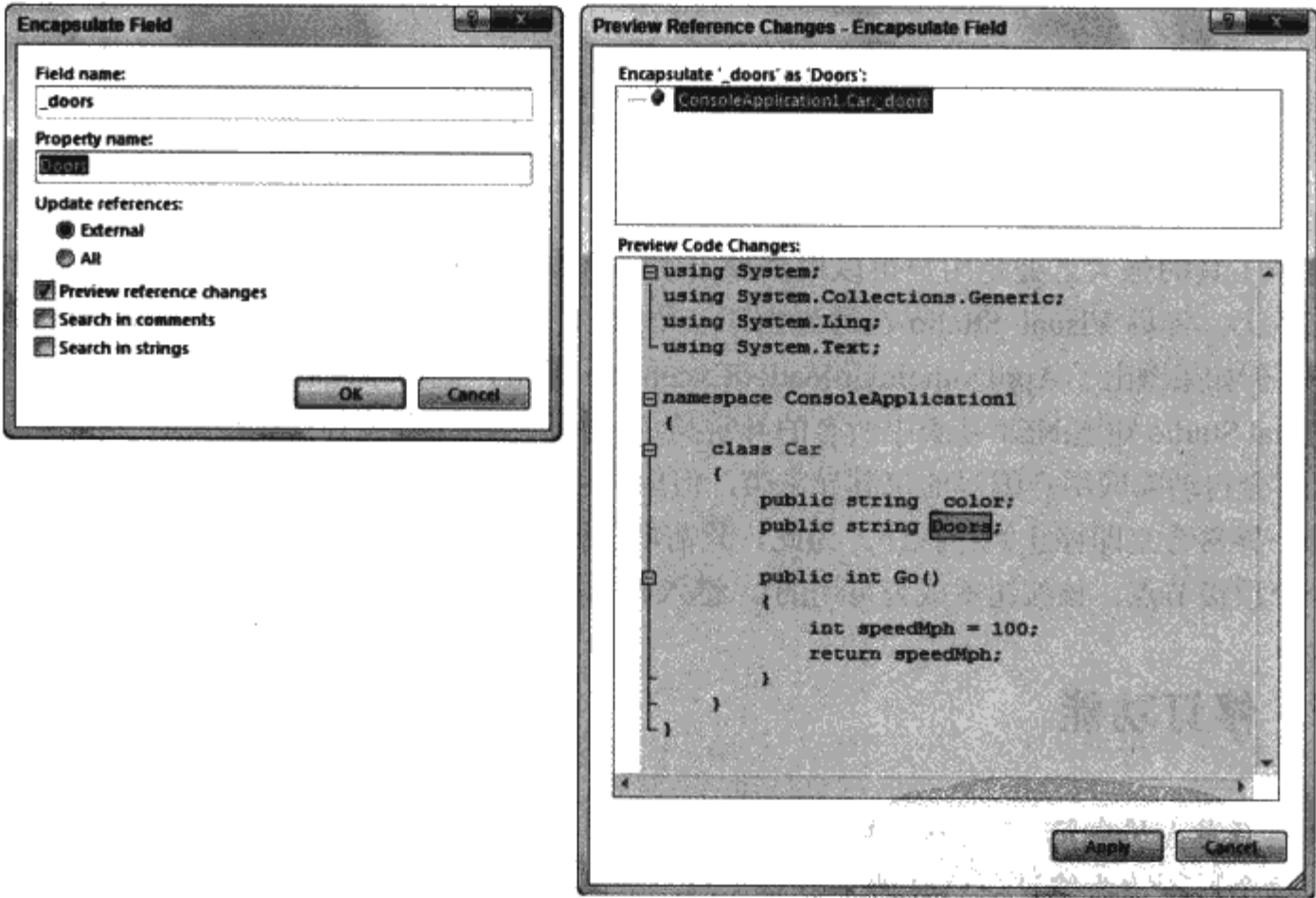


图 15-28

在这个对话框中，提供属性名，单击 OK 按钮，就把选中的公共字段转换为一个私有字段，并把它封装到一个.NET 公共属性中。单击 OK 后，代码会改为(改写了两个字段)：

```
namespace ConsoleApplication1
{
    public class Car
    {
        private string _color;
```

```
public string Color
{
    get { return _color; }
    set { _color = value; }
}
private string _doors;

public string Doors
{
    get { return _doors; }
    set { _doors = value; }
}

public int Go()
{
    int speedMph = 100;
    return speedMph;
}
}
```

可以看出，这些向导不但能修订一个页面上的代码，也能修订整个应用程序的代码。它们还可以完成如下任务：

- 给方法、局部变量、字段等重命名
- 从选中的代码中提取方法
- 根据一组已有的类型成员提取接口
- 把局部变量提升为参数
- 对参数重命名或重新排序

Visual Studio 2008 提供的新修订功能可以使代码更简洁、可读性更高，结构化更强。

### 15.3 多目标

Visual Studio 2008 是允许面向需要使用的 .NET Framework 版本的第一个 IDE 版本。打开 New Project 对话框，准备创建新项目时，注意在对话框的右上角有一个下拉列表，它允许选择要使用的 Framework 版本。这个对话框如图 15-29 所示。

在这个图中，下拉列表提供了面向 .NET Framework 2.0、3.0 或 3.5 的功能。这是因为 Framework 3.0 和 3.5 是 .NET Framework 2.0 的扩展。使用升级对话框把 Visual Studio 2005 解决方案升级到 Visual Studio 2008 时，只是把解决方案升级为使用 Visual Studio 2008，而不是把项目升级到 .NET Framework 3.5 上。项目仍在以前使用的 .NET Framework 上工作，但现在可以使用新的 Visual Studio 2008 处理项目了。



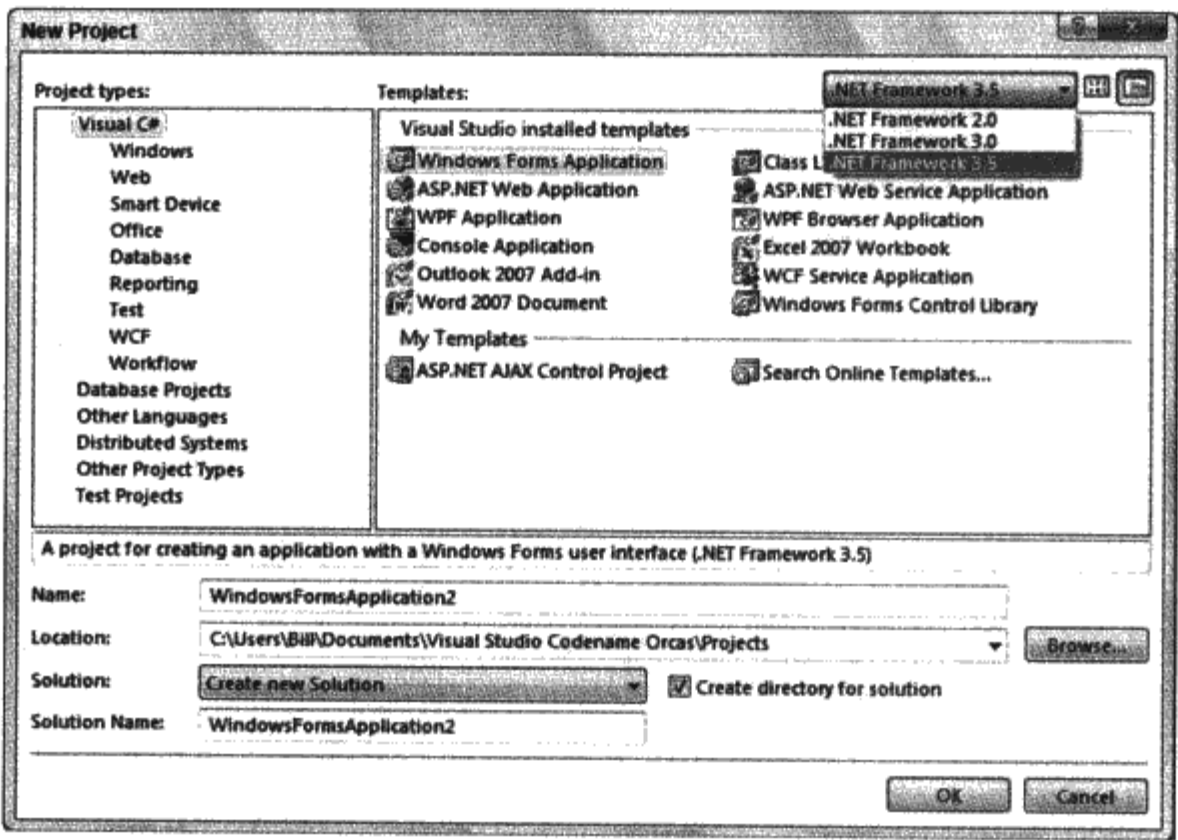


图 15-29

如果要改变解决方案使用的 Framework 版本,可以右击解决方案,选择该解决方案的属性。如果处理的是 ASP.NET 项目, 会打开如图 15-30 所示的对话框。

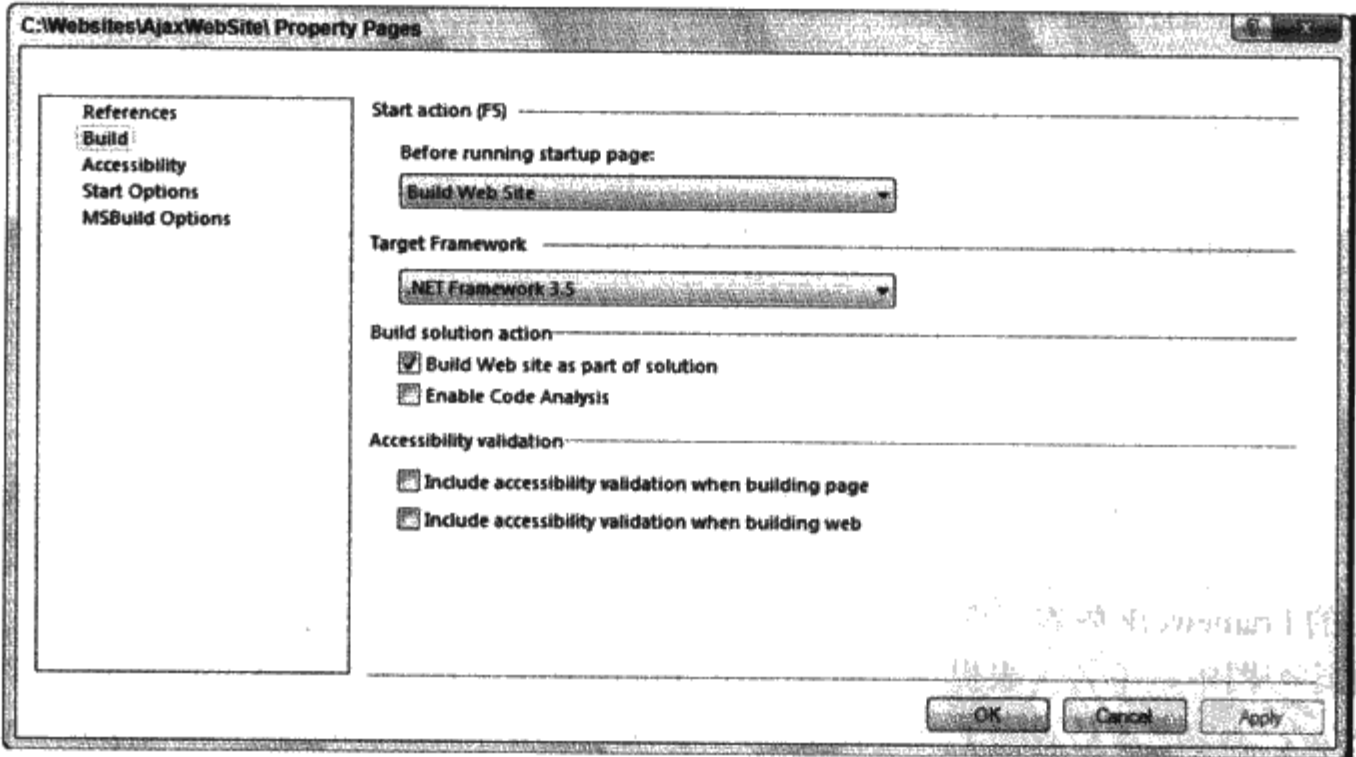


图 15-30

在这个对话框中, Build 选项卡提供了改变应用程序使用的 Framework 版本的功能。如果打开的是一个 Windows 窗体应用程序的属性页面,就可以在 Application 选项卡(第一个选项卡上)上改为使用另一个 Framework 版本, 如图 15-31 所示。

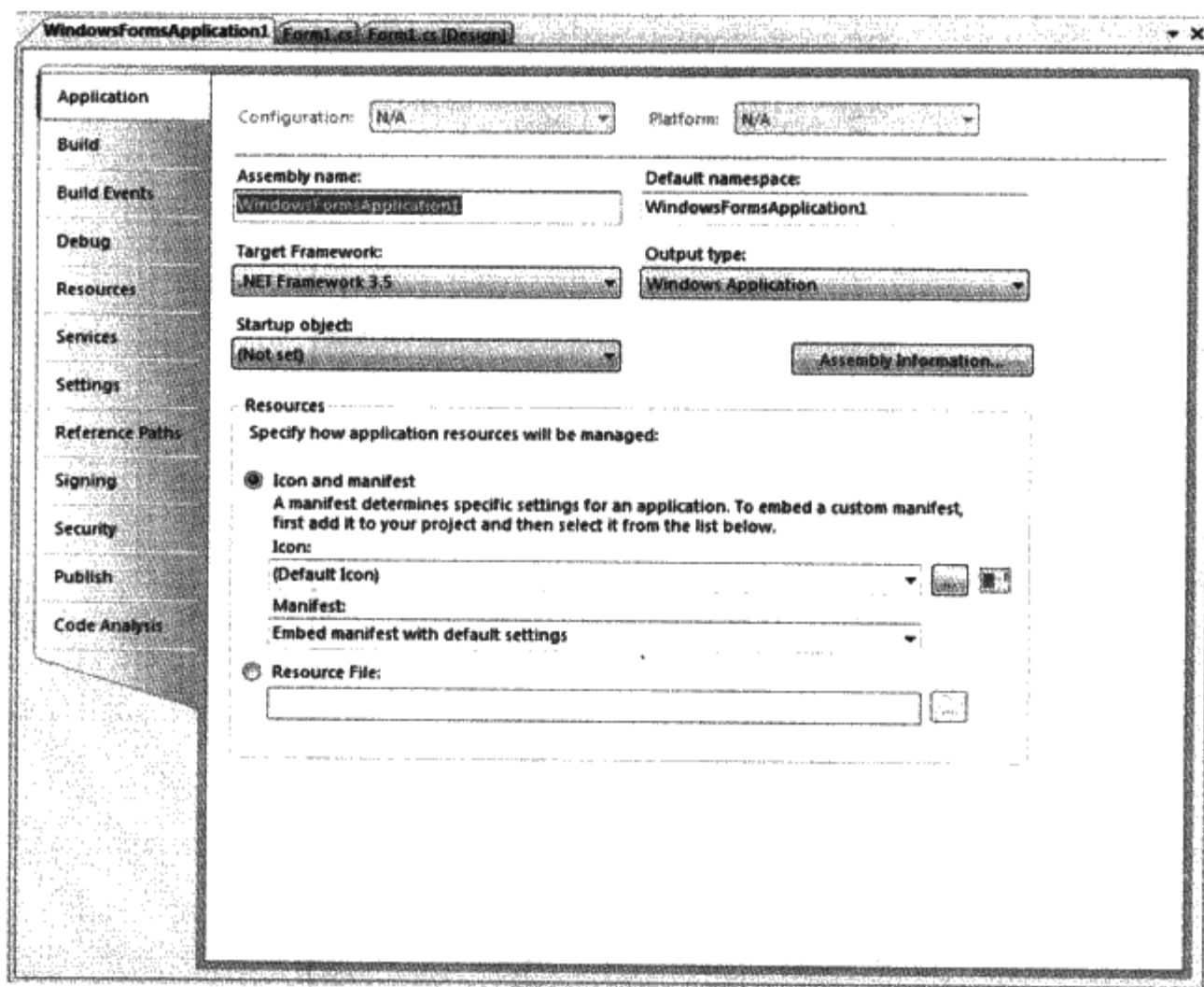


图 15-31

## 15.4 WPF、WCF、WF 等

在默认情况下, Visual Studio 2005 不允许建立面向 .NET Framework 3.0 的应用程序。Visual Studio 2005 的默认安装仅面向 .NET Framework 2.0。要开始使用面向 .NET Framework 3.0 的新技术, 需要安装另外几个软件包。

.NET Framework 3.0 允许使用类库来建立新的应用程序类型, 例如使用 Windows Presentation Foundation (WPF)、Windows Communication Foundation (WCF)、Windows Workflow Foundation (WF) 和 Windows CardSpace 的应用程序。

Visual Studio 2008 的面向 Framework 功能允许建立使用 .NET Framework 3.0 或 3.5 的这些应用程序类型。

### 15.4.1 在 Visual Studio 中建立 WPF 应用程序

.NET Framework 3.5 给 Visual Studio 带来的一个主要变化是增加了 Windows Application (WPF) 项目类型(在 Windows 类别中)。选择这个项目类型会创建一个 Windows.xaml 文件和一个 Windows.xaml.cs 文件。这个项目类型在 Solution Explorer 中默认创建的内容如图 15-32 所示(带有可搜索的新属性窗口)。

在 Visual Studio 2008 中, 最大的变化出现在文档窗口中。创建这个项目后, 文档窗口的默认视图如图 15-33 所示。

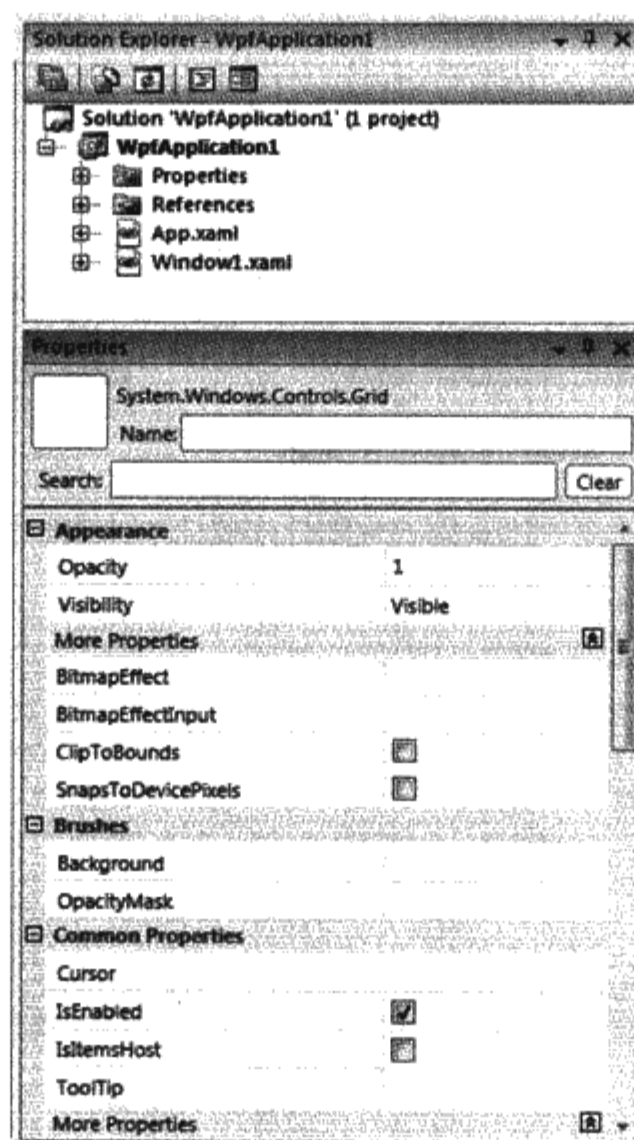


图 15-32

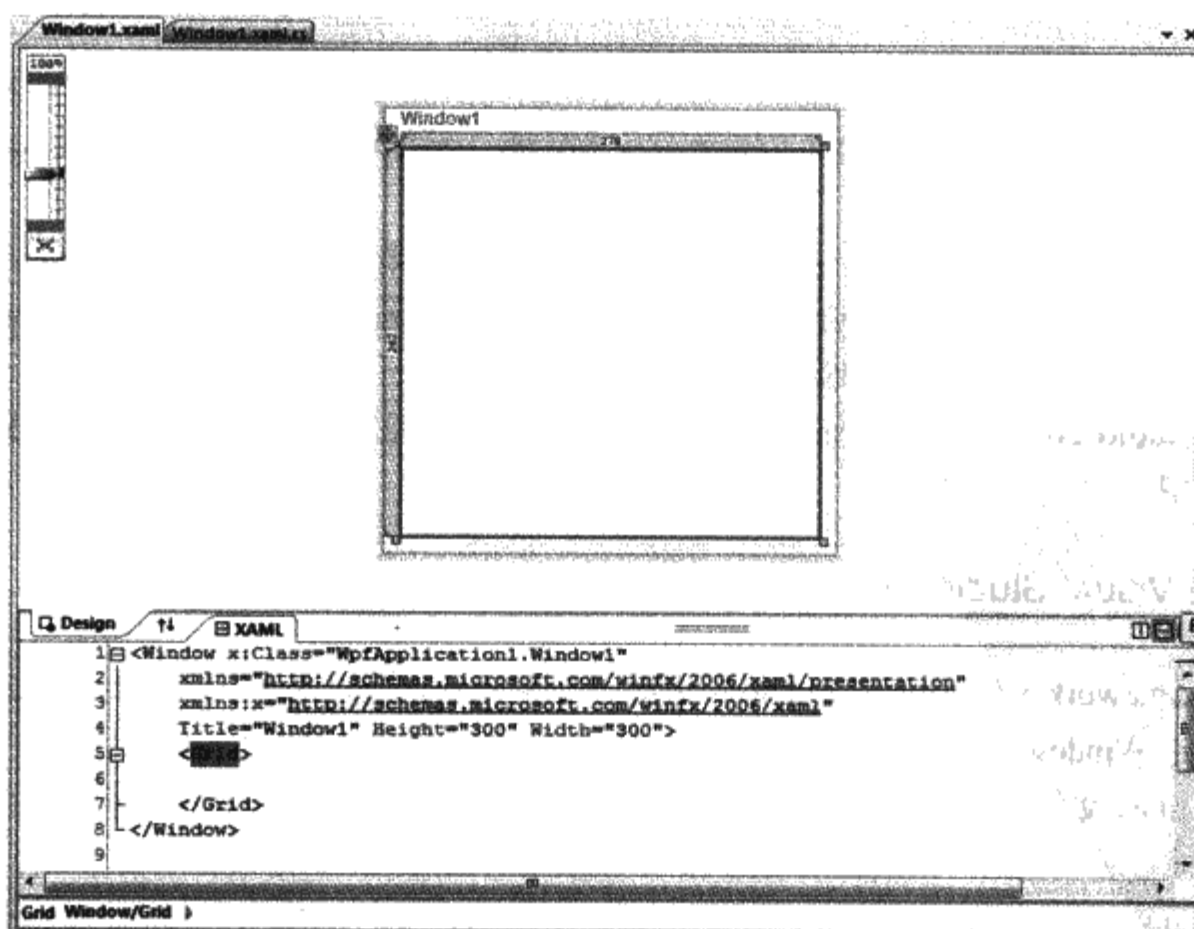


图 15-33

文档窗口有两个视图：设计视图和 XAML 视图。在设计视图中进行修改，会使 XAML 视图出现相应的变化，反之亦然。与传统的 Windows 窗体应用程序一样，WPF 应用程序也可以使用包含在 Visual Studio 工具箱中的控件。控件的这个新工具箱如图 15-34 所示。



图 15-34

#### 15.4.2 在 Visual Studio 中建立 WF 应用程序

另一个完全不同的应用程序样式(在 Visual Studio 中建立应用程序时)是 Windows Workflow 应用程序类型。例如，从 New Project 对话框的 Workflow 部分选择 Sequential Workflow Console Application 项目类型，就会创建一个控制台应用程序，它的 Solution Explorer 视图如图 15-35 所示。

在建立使用 Windows Workflow Foundation 的应用程序时，一个很大的变化是它非常依赖于设计视图。仔细查看工作流(如图 15-36 所示)，会发现它包含多个顺序步骤，甚至包含基于条件(如 if-else 语句)的操作。

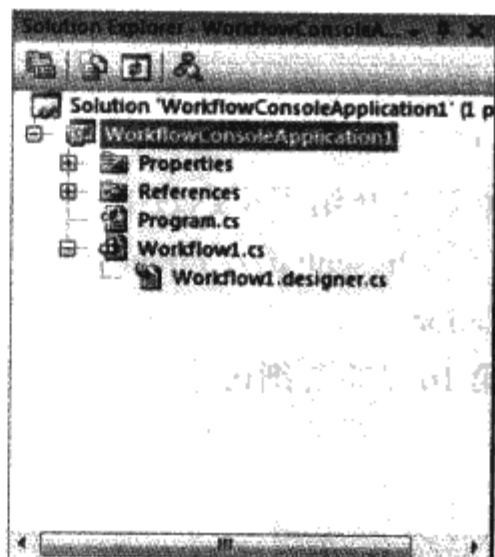


图 15-35

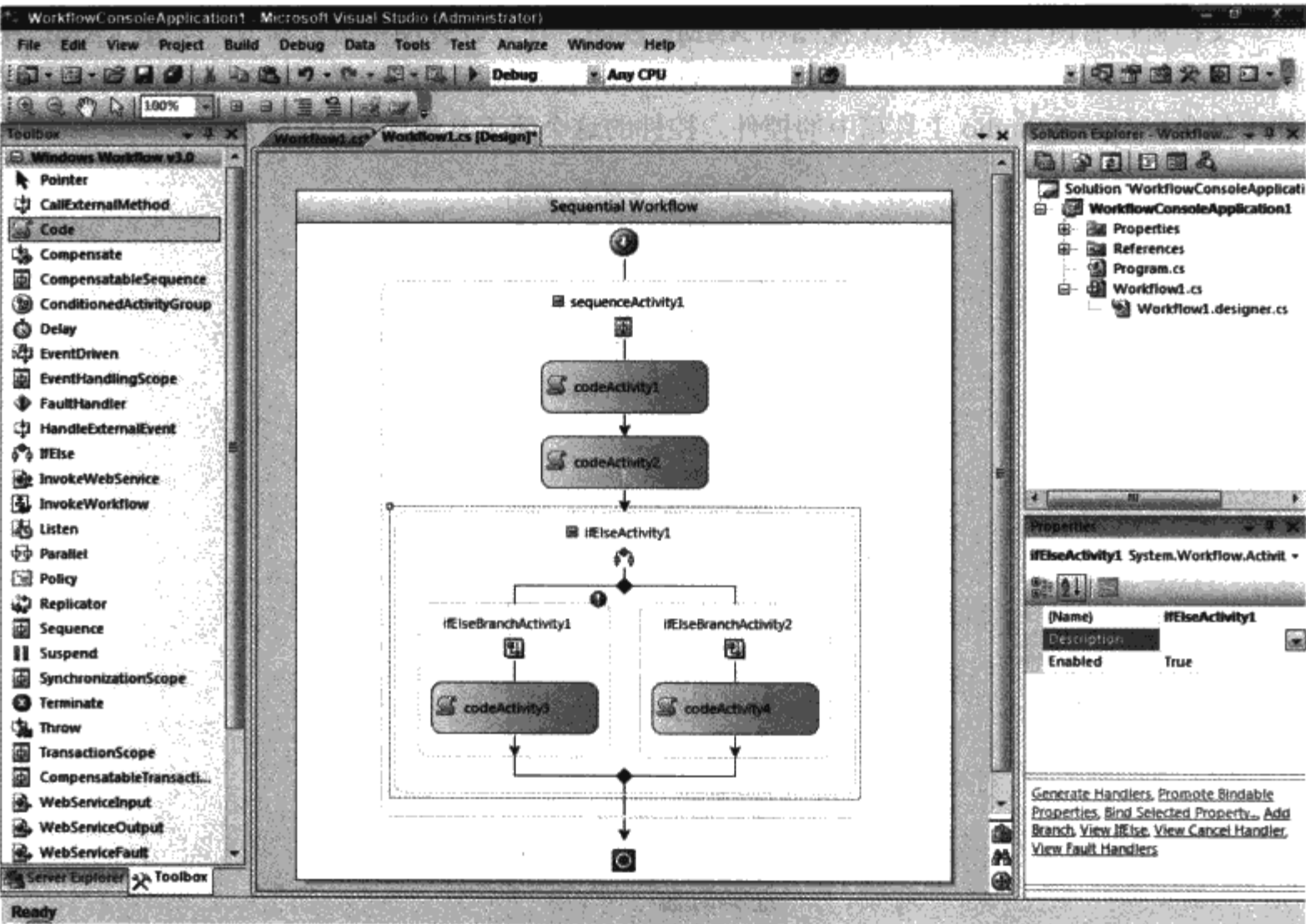


图 15-36

### 15.5 小结

本章介绍了.NET 环境中的一个最重要的编程工具 Visual Studio 2008。本章的大部分内容说明如何使用这个工具编写 C#代码(以及 C++和 Visual Basic 2008 代码)。

Visual Studio 2008 是编程界中最简单的开发环境之一。Visual Studio 很容易进行 RAD 开发，同时还可以深入了解创建应用程序的机制。本章详细介绍了如何使用 Visual Studio 完成各种任务，包括修订代码，多目标、读取 Visual Studio 6 项目、调试程序，以及许多可用于 Visual Studio 的窗口。

本章还介绍了可通过.NET Framework 3.5 创建的新项目。这些新的项目类型关注的是 Windows Presentation Foundation、Windows Communication Foundation 和 Windows Workflow Foundation。

第 16 章将详细讨论部署。



# 第 16 章

## 部 署

编译源代码并完成测试后，开发过程并没有结束。在这个阶段，需要把应用程序提供给用户。无论是 ASP.NET 应用程序、智能客户应用程序还是用 Compact Framework 构建的应用程序，软件都必须部署到目标环境中。.NET Framework 使部署工作比以前容易得多，因为不再需要注册 COM 组件，编写新的注册表项。

本章将介绍可用于应用程序部署的选项，包括 ASP.NET 应用程序和智能客户应用程序的部署选项。本章的主要内容如下：

- 部署要求
- 简单的部署情况
- 基于 Windows 安装程序的项目
- ClickOnce

### 16.1 部署的设计

部署常常是开发过程之后的工作，如果不下一定的工夫，可能会导致错误。为了避免在部署过程中出错，应在最初的设计阶段就对部署过程进行规划。任何部署问题，例如服务器的容量、桌面的安全性或从哪里加载程序集等，都应从一开始就纳入设计，这样部署过程才会比较顺利。

另一个必须在开发过程早期解决的问题是，在什么环境下测试部署。应用程序代码的单元测试和部署选项的测试可以在开发系统中进行，而部署必须在类似于目标系统的环境中测试。这是非常重要的，特别是目标计算机上不存在从属文件时。例如，第三方的库很早就安装在项目的开发计算机上，但目标计算机可能没有安装这个库。在部署软件包中很容易忘记包含这个库。在开发系统上进行的测试不可能发现这个错误，因为库已经存在了。对从属文件的说明可以帮助减少这种潜在的错误。

部署过程对于大型应用程序来说可能非常复杂。提前规划部署，在部署过程中可以节省时间和精力。

### 16.2 部署选项

本节概述 .NET 开发人员可以使用的部署选项。其中大多数选项将在本章的后面详细论述。

### 16.2.1 Xcopy 实用工具

Xcopy 实用工具允许把程序集或程序集组复制到应用程序文件夹中,从而减少了开发时间。由于程序集是自我包含的,元数据描述了包含在程序集中的内容,所以不需要在注册表中注册。每个程序集都跟踪它需要执行的其他程序集。默认情况下,程序集会在当前的应用程序文件夹中查找从属文件。把程序集移动到其他文件夹的过程将在本章后面讨论。

### 16.2.2 Copy Web 工具

如果开发的是 Web 项目,使用 Web 站点菜单中的 Copy Web 选项就会把运行应用程序所需要的组件复制到服务器上。

### 16.2.3 发布 Web 站点

在发布 Web 站点时,会编译整个站点,然后复制到指定的位置。在预编译时,所有的源代码都会从最终的输出中删除,找出和处理所有编译错误。

### 16.2.4 部署项目

Visual Studio 2008 可以为应用程序创建安装程序。基于 Microsoft Windows Installer 技术有 4 种选择:创建合并模块;为客户应用程序创建安装程序;为 Web 应用程序创建安装程序;以及为基于智能设备(Compact Framework)的应用程序创建安装程序。还可以创建 cab 文件。部署项目为安装过程提供了极大的灵活性和可定制性。这 4 种部署方式对于大型应用程序都十分有用。

### 16.2.5 ClickOnce

ClickOnce 可以建立自动升级的、基于 Windows 的应用程序。ClickOnce 允许把应用程序发布到 Web 站点、文件共享、甚或 CD 上。在对应用程序进行升级、重新生成后,开发小组可以把它们发布到相同的位置或站点上。最终用户在使用应用程序时,程序会检查是否有更新版本,如果有,就进行更新。

## 16.3 部署的要求

基于 .NET 的应用程序一般都有运行要求。在执行任何托管的应用程序之前,CLR 对目标平台都有一定的要求。

首先必须满足的要求是操作系统。目前下面的操作系统可以运行基于 .NET 的应用程序:

- Windows 98
- Windows 98, 第 2 版(SE)
- Windows Millennium Edition(ME)

- Windows NT 4.0(Service Pack 6a)
- Windows 2000
- Windows XP Home
- Windows XP Professional
- Windows XP Professional TabletPC Edition
- Windows Vista

其次，必须支持下面的服务器平台：

- Windows 2000 Server 和 Advanced Server
- Windows 2003 Server 系列

其他要求有 Windows Internet Explorer 5.01 或更高版本，MDAC 2.6 或更高版本(应用程序需要访问数据)和用于 ASP.NET 应用程序的 Internet Information Services(IIS)。

在部署 .NET 应用程序时，还必须考虑硬件要求。硬件的最低要求是：

- 客户机：奔腾 90MHz，32MB RAM
- 服务器：奔腾 133MHz，128MB RAM

要获得最佳性能，应增加 RAM，RAM 越大，.NET 应用程序运行得就越好。服务器应用程序更是如此。

如果要运行使用 Windows Presentation Foundation (WPF)、Windows Communication Foundation (WCF)或 Windows Workflow Foundation (WWF)的 .NET 3.0 应用程序，要求就更严格。.NET 3.0 至少需要 Windows XP SP2。上述列表还应添加如下内容：

- Windows XP Home (SP2)
- Windows XP Professional (SP2)
- Windows XP Professional TabletPC Edition (SP2)
- Windows Vista (不包括 IA64 平台)

支持下述服务器平台：

- Windows 2003 Server Family (SP1)
- Windows Server 2008 IA64 版本

最低的硬件要求也有变化：客户机和服务器都必须是奔腾 400 MHz，96 MB RAM。

## 16.4 部署 .NET 运行库

使用 .NET 开发应用程序时，需要依赖 .NET 运行库。这似乎很明显，但有时可以忽略这一点。如果应用程序不使用任何 .NET 3.0 功能，就只需要安装 dotnetfx.exe (64 位操作系统需要 netfx64.exe)。如果使用了 .NET 3.0 功能，还需要安装 dotnetfx3.exe。如果使用了 .NET 3.5 功能，还需要安装 netfx35\_x86.exe。

在下面对创建部署软件包的讨论中，运行库的安装是可选的。安装程序会检查是否安装了相应的运行库，如果没有，安装程序会从本地媒介中安装运行库，或者到指定的下载站点上下载并安装运行库。

## 16.5 简单的部署

如果在应用程序的初始设计阶段考虑了部署，那么部署就只是把一组文件复制到目标计算机上。对于 Web 应用程序，就只需使用 Visual Studio 2008 中的一个菜单选项。本节就讨论这种简单的部署情况。

为了了解如何设置各种部署选项，必须有一个要部署的应用程序。从 [www.wrox.com](http://www.wrox.com) 上下载的示例包含 3 个项目：SampleClientApp、SampleWebApp 和 AppSupport。SampleClientApp 是一个智能客户应用程序，SampleWebApp 是一个简单的 Web 应用程序，AppSupport 是一个类库，它包含一个简单的类，该类返回一个包含当前日期和时间的字符串。SampleClientApp 和 SampleWebApp 使用 AppSupport 的结果填充一个标签。为了使用这些示例，首先加载并构建 AppSupport。然后在其他两个应用程序中，设置对新构建的 AppSupport.dll 的引用。

下面是 AppSupport 程序集的代码：

```
using System;

namespace AppSupport
{
    ///<summary>
    ///Simple assembly to return date and time string.
    ///</summary>
    public class Support
    {
        private Support()
        {
        }

        public static string GetDateTimeInfo()
        {
            DateTime dt = DateTime.Now;
            return string.Concat(dt.ToLongDateString(), " ", dt.ToLongTimeString());
        }
    }
}
```

这个简单的程序集足以演示可用的部署选项。

### 16.5.1 Xcopy 部署

Xcopy 部署就是把一组文件复制到目标计算机上的一个文件夹中，然后在客户机上执行应用程序。这个术语来自于 DOS 命令 `xcopy.exe`。无论程序集的数目是多少，如果文件复制到同一个文件夹中，应用程序就会运行，不需要编辑配置设置或注册表。

为了理解 Xcopy 部署的工作原理，执行下面的步骤：

- (1) 打开示例下载文件中的 SampleClientApp 解决方案(Sample ClientApp.sln)。
- (2) 把目标改为 Release，进行完整的编译。
- (3) 然后，使用“我的电脑”或文件管理器导航到项目文件夹\SampleClientApp\bin\ Release，双击 SampleClientApp.exe，运行应用程序。
- (4) 现在单击按钮，打开另一个对话框。这将验证应用程序是否能正常运行。当然，这个

文件夹是 Visual Studio 放置输出的地方，所以应用程序能正常工作。

(5) 创建一个新文件夹，命名为 ClientAppTest。把这两个文件从 Release 文件夹复制到这个新文件夹中，然后删除 Release 文件夹。再次双击 SampleClientApp.exe 文件，验证它是否正常工作。

Xcopy 部署只需把程序集复制到目标机器上，就可以部署功能完善的应用程序。这里使用的示例非常简单，但这并不意味着这个过程对较复杂的应用程序无效。实际上，使用这种方法对要部署的程序集的大小和数目没有限制。不想使用 Xcopy 部署的原因是它不能把程序集放在全局程序集缓存(GAC)中，或者不能在“开始”菜单中添加图标。如果应用程序仍依赖于某种类型的 COM 库，就不能很容易地注册 COM 组件。

### 16.5.2 Xcopy 和 Web 应用程序

Xcopy 部署也可以用于 Web 应用程序，但文件夹结构有点不同。我们必须建立 Web 应用程序的虚拟目录，并配置合适的用户权限。这个过程通常需要使用 IIS 管理工具来完成。在建立虚拟目录后，Web 应用程序文件就可以复制到虚拟目录中。复制 Web 应用程序的文件有点困难，需要考虑两个配置文件和页面使用的图像。

### 16.5.3 Copy Web 工具

一种较好的方法是使用 Copy Web 工具。在 Visual Studio 2008 的 Website | Copy Web Site 菜单项中就可以访问工具。它基本上是一个 FTP 客户程序，用于给远程位置来回传送文件。远程位置可以是任意 FTP 或 Web 站点，包括本地 Web 站点、IIS Web 站点和 Remote (Frontpage) Web 站点。Copy Web 工具的另一个特性是，它会把远程服务器上的文件与源站点上的文件同步。源站点总是 Visual Studio 2008 中当前打开的站点。如果当前项目有多个开发人员，就可以使用这个工具与本地开发站点保持同步。所进行的修改可以与用于测试的公共服务器进行同步。

### 16.5.4 发布 Web 站点

Web 项目的另一个部署选项是发布 Web 站点。发布 Web 站点就是预编译整个站点，并把编译好的版本放在指定的位置。该位置可以是文件共享、FTP 位置，或可以通过 HTTP 访问的其他位置。编译过程会从程序集中去除所有的源代码，为部署创建 dll 文件。这也包括 ASPX 源文件中的标记。ASPX 文件并不包含一般的标记，而是包含程序集的一个指针。每个 ASPX 文件都与一个程序集相关。无论是模型、后台代码或单个文件，这个过程都会执行。

发布 Web 站点的优点是速度快，很安全。速度有所提高，是因为所有的程序集都已编译。否则，第一次访问页面时会会有一个延迟，因为要编译和缓存页面和从属代码。安全性有所提高，是因为不部署源代码。另外，在部署前所有的源代码都进行了预编译，找出了所有的编译错误。

使用 Website | Publish Web Site 菜单项就可以发布 Web 站点。我们需要提供要发布的位置。这也可以是文件共享、FTP 位置、Web 站点或本地磁盘路径。在完成编译后，文件就放在指定的位置。在这里可以把文件复制到阶段服务器、测试服务器或产品服务器上。



## 16.6 Installer 项目

xcopy 部署使用起来很简单,但有时它缺乏一些功能。为了克服这个缺点,Visual Studio 2008 提供了 6 个 Installer 项目类型。其中 4 个类型基于 Windows Installer 技术,表 16-1 列出了这些项目类型。

表 16-1

项目类型	说明
Setup Project	用于安装客户应用程序、中间层应用程序和运行为 Windows 服务的应用程序
Web Setup Project	用于安装基于 Web 的应用程序
Merge Module Project	创建.msm 合并模块,这些模块可以和其他基于 Windows Installer 的安装应用程序一起使用
Cab Project	创建 cab 文件,通过旧式的部署技术进行发布
Setup Wizard	帮助创建部署项目
Smart Device CAB Project	Pocket PC、Smartphone 和其他基于 CE 的应用程序的 CAB 项目

Setup 和 Web Setup Project 非常相似。主要的区别是使用 Web Setup,项目会部署到 Web 服务器上的一个虚拟目录中,而使用 Setup Project,项目会部署到文件夹结构中。这两个项目类型都基于 Windows Installer,拥有基于 Windows Installer 的安装程序的所有功能。在创建包含在多个部署项目中的组件或功能库时,一般使用 Merge Module Project。创建合并模块时,可以设置专用于组件的配置项目,而无需在主部署项目的创建过程中考虑它们。Cab Project 类型仅为应用程序创建 Cab 文件。Cab 文件由旧式的安装技术以及一些基于 Web 的安装过程使用。Setup Wizard 项目类型逐步完成创建部署项目的步骤,在此过程中向用户询问特定的问题。下面的几节讨论如何创建这些部署过程,可以改变哪些设置和属性,可以增加什么定制内容。

### 16.6.1 Windows Installer

Windows Installer 是一个服务,负责管理在大多数 Windows 操作系统上安装、更新、修复和删除应用程序。它是 Windows ME、Windows 2000、Windows XP 和 Windows Vista 的一部分,可以用于 Windows 95、Windows 98 和 Windows NT 4.0。Windows Installer 的当前版本是 3.0。

Windows Installer 在数据库中跟踪应用程序的安装。在卸载应用程序时,Windows Installer 很容易跟踪和删除已添加的注册表设置、复制到硬盘上的文件,以及已添加的桌面和“开始”菜单图标。如果有某个文件仍被另一个应用程序引用,安装程序就会把它保留在硬盘上,不会使其他的应用程序中断。数据库还可以修复应用程序。如果注册表设置或与应用程序相关的 dll 被破坏或不小心删除了,就可以修复安装。在修复过程中,安装程序会从上一次安装中读取数据库,并复制该安装。

Visual Studio 2008 中的部署项目可以创建 Windows 安装软件包。部署项目允许访问大多数需要访问的内容,以便安装给定的应用程序。但是,如果需要更多的控制,就应查看 Windows

Installer SDK，它在 Platform SDK 中，其中包含了为应用程序创建定制安装软件包的说明。下面几节将使用 Visual Studio 2008 部署项目创建这些安装软件包。

### 16.6.2 创建安装程序

为客户应用程序或 Web 应用程序创建安装软件包并不困难。第一个任务是标识应用程序需要的所有外部资源，包括配置文件、COM 组件、第三方库、控件和图像。前面说过，在项目的文档说明中应包含一个从属文件列表。这个文档说明是非常有用的。Visual Studio 2008 可以询问程序集，提取该程序集的从属文件，但我们仍需要审查这些内容，以防遗漏。

另一个问题是，在过程的什么时候创建安装软件包。如果设置了一个自动构建过程，就可以把安装软件包的构建包含在项目成功构建的过程中。在耗时而复杂的大型项目中，过程的自动进行会大大减少出错的可能性，我们可以把部署项目包含在项目解决方案中。Solution Property Pages 对话框中有一个 Configuration Properties 设置。使用这个设置可以选择要为各种构建配置包含的项目。如果选择 Release builds but not for the Debug builds 下面的 Build 复选框，安装软件包就只在创建发布版本时创建。这也是下面示例所使用的过程。图 16-1 显示了 SampleClientApp 解决方案的 Solution Property Pages 对话框。其中显示了 Debug 配置，没有给安装项目选中 Build 复选框。

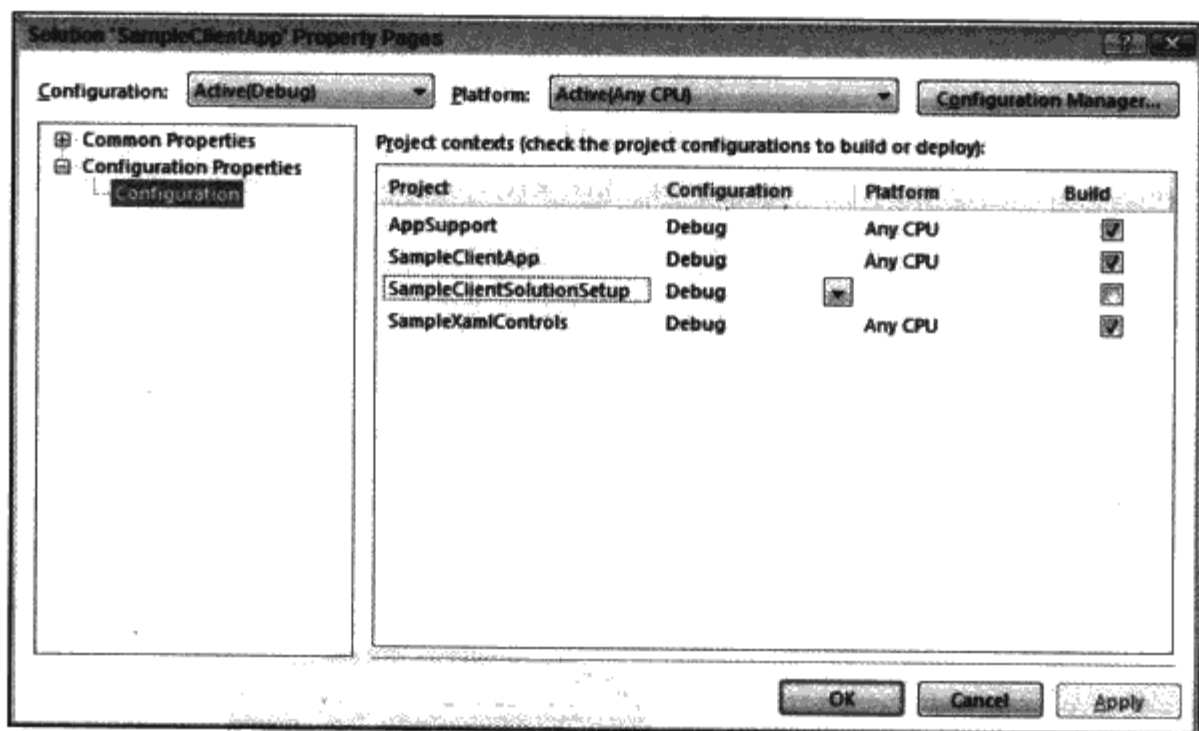


图 16-1

#### 1. 简单的客户应用程序

在下面的示例中，要为 SampleClientApp 解决方案创建一个安装程序(它包含在示例下载文件中，其中还包含已完成的安装程序项目)。

对于 SampleClientApp，创建两个部署项目。一个创建为独立的解决方案，另一个在原来的解决方案中创建，以便说明选择创建这两个部署项目的优缺点。

第一个示例将说明如何在独立的解决方案中创建部署项目。在开始创建部署项目之前，要确保部署的应用程序有一个发布版本。接着，在 Visual Studio 2008 中创建一个新项目。在 New Project

对话框中，选择左边的 Setup and Deployment Projects，再选择右边的 Setup Project，给它指定一个名称(例如 SampleClientStandaloneSetup)。

在 Solution Explorer 窗口中，单击项目，再单击 Properties 窗口，就会看到一组属性。这些属性将在应用程序的安装过程中显示。其中一些属性还会显示在“添加/删除程序”控制面板上。由于用户在安装过程中可以看到大多数属性(或者在“添加/删除程序”控制面板上查看安装时可以看到它们)，所以正确设置它们会使应用程序更专业。这个属性列表非常重要，特别是在应用程序要进行商业化部署时，更是如此。表 16-2 描述了这些属性及其值。

表 16-2

项 目 属 性	说 明
AddRemoveProgramIcon	显示在“添加/删除程序”对话框中的图标
Author	应用程序的编写者。这个属性设置通常与 Manufacturer 的相同，它显示在 msi 软件包的 Properties 对话框中的 Summary 页面上，以及“添加/删除程序”对话框的 SupportInfo 页面上的 Contact 字段中
Description	这是一个形式自由的文本字段，描述了要安装的应用程序或组件。这些信息显示在 msi 软件包的 Properties 对话框中的 Summary 页面上，以及“添加/删除程序”对话框的 SupportInfo 页面上的 Contact 字段中
DetectNewerInstalledVersion	这是一个布尔值，设置为 true 时，将检查是否已安装了应用程序的更新版本，如果是，就停止安装过程
InstallAllUsers	这是一个布尔值，设置为 true 时，将为计算机的所有用户安装应用程序。设置为 false 时，就只有当前用户能访问应用程序
Keywords	可用于在目标计算机上搜索 msi 文件。这些信息显示在 msi 软件包的 Properties 对话框中的 Summary 页面上
Localization	用于字符串资源和注册设置的地域。这会影响安装程序的用户界面
Manufacturer	生产应用程序或组件的公司名称。一般与 Author 属性中指定的信息相同。这些信息显示在 msi 软件包的 Properties 对话框中的 Summary 页面上，以及“添加/删除程序”对话框的 SupportInfo 页面上的 Publisher 字段中，用作应用程序默认安装路径的一部分
ManufacturerURL	一个 Web 站点的 URL，该站点与要安装的应用程序或组件相关
PostBuildEvent	在构建结束后执行的命令
PreBuildEvent	在构建开始前执行的命令
ProductCode	应用程序或组件的唯一的字符串 GUID。Windows Installer 使用这个属性标识应用程序的后续升级或安装
ProductName	描述应用程序的名称，在“添加/删除程序”对话框中用作应用程序的描述，还用作默认安装路径的一部分：C:\Program Files\Manufacturer\ProductName

(续表)

项 目 属 性	说 明
RemovePreviousVersions	一个布尔值，如果设置为 true，就检查计算机上是否安装了应用程序的以前版本。如果是，就调用以前版本的卸载功能，之后继续安装。这个属性使用 ProductCode 和 UpgradeCode 确定是否卸载。UpgradeCode 应相同，而 ProductCode 应不同
RunPostBuildEvent	运行 PostBuildEvent 的时间，其选项有 On successful build 或 Always
SearchPath	一个字符串，表示从属程序集、文件或合并模块的搜索路径。在开发机器上建立安装软件包时使用该属性
Subject	与应用程序相关的其他信息。这些信息显示在 msi 软件包的 Properties 对话框中的 Summary 页面上
SupportPhone	为应用程序或组件提供支持的电话号码。这些信息显示在“添加/删除程序”对话框的 SupportInfo 页面上的 Support Information 字段中
SupportURL	为应用程序或组件提供支持的 URL。这些信息显示在“添加/删除程序”对话框的 SupportInfo 页面上的 Support Information 字段中
TargetPlatform	支持 Windows 的 32 或 64 位版本
Title	安装程序的标题。它显示在 msi 软件包的 Properties 对话框中的 Summary 页面上
UpgradeCode	一个字符串 GUID，表示同一应用程序的不同版本共享的标识符。对于应用程序的不同版本或不同语言版本，不应修改 UpgradeCode，该属性由 DetectNewerInstalled Version 和 RemovePreviousVersions 使用
Version	安装程序、cab 文件或合并模块的版本号。注意它不是要安装的应用程序的版本号

设置完属性后，就可以开始添加程序集了。在这个示例中，唯一要添加的程序集是主可执行文件 SampleClientApp.exe。为此，可以在 Solution Explorer 中右击项目，或从 Project 菜单中选择 Add，此时有 4 个选项：

- Project Output: 下一个示例探讨这个选项
- File: 用于添加 readme 文本文件或不在构建过程中添加的其他文件
- Merge Module: 独立创建的合并模块
- Assembly: 使用这个选项可以选择要安装的程序集

本例选择 Assembly，打开 Component Selector 对话框，该对话框类似于给项目添加引用的对话框。浏览至应用程序的\bin\release 文件夹，选择 SampleClientApp.exe，在 Component Selector 对话框中单击 OK，现在可以看到 SampleClientApp.exe 在部署项目的 Solution Explorer 中。在 Detected Dependancies 部分，可以看到 Visual Studio 要求 SampleClientApp.exe 给出它需要的程序集。在本例中，会自动包括 AppSupport.dll。继续这个过程，直到应用程序中的所有程序集

都显示在部署项目的 Solution Explorer 中为止。

接着，需要确定把程序集部署到什么地方。在默认情况下，在 Visual Studio 2008 中会显示文件系统编辑器，这个编辑器分为两个窗格，左边的窗格显示目标机器上文件系统的层次结构，右边的窗格则显示选中文件夹的详细视图。文件夹名称可能不是我们希望看到的，但这些文件夹用于目标机器，例如，文件夹 User’s Programs Menu 映射为目标客户机的 C:\Documents and Settings\User Name\Start Menu\Programs 文件夹。

此时可以添加其他文件夹，例如特定的文件夹或定制的文件夹。要添加特定的文件夹，应确保目标机器上的文件系统在左边的窗格上突出显示，然后选择主菜单中的 Action 菜单。Add Special Folder 菜单选项提供了可以添加的文件夹列表。例如，如果要在 Application 文件夹中添加一个文件夹，就可以在编辑器的左边窗格上选择 Application 文件夹，再选择 Action 菜单。这时就会出现一个可以创建新文件夹的 Add 菜单。给新文件夹重新命名，就会在目标机器上创建它。

我们要添加的一个特定文件夹是 GAC 文件夹。如果有几个不同的应用程序使用 AppSupport.dll，就可以把它安装到 GAC 中。为了把程序集添加到 GAC 中，程序集必须有一个强名。把程序集添加到 GAC 的过程就是在 Special Folder 菜单中添加 GAC，再把要放在 GAC 中的程序集从当前文件夹拖放到 Global Assembly Cache 文件夹中。如果试图对没有强名的程序集进行这个操作，部署项目就不能编译。

如果选择 Application 文件夹，在右边的窗格上，刚才添加的程序集就会自动添加到 Application 文件夹中。还可以把程序集移动到其他文件夹中，但程序集必须能找到对方(有关探测的更多信息请参阅第 17 章)。

如果要在用户的桌面或“开始”菜单上添加应用程序的快捷方式，就应把该快捷方式拖放到适当的文件夹中。要创建桌面快捷方式，就应进入 Application 文件夹，在编辑器的右边窗格上选择该应用程序，再进入 Action 菜单，选择 Create Shortcut 菜单项，创建应用程序的快捷方式。在创建好快捷方式后，把它拖放到 User’s Desktop 文件夹中。现在安装应用程序，快捷方式就会显示在桌面上。一般情况下，应由用户决定是否需要应用程序的快捷方式。要求用户输入信息并执行相应步骤的过程将在本章后面介绍。在“开始”菜单中创建菜单项的过程与此相同。另外，如果查看刚才创建的快捷方式的属性，就可以配置快捷方式的基本属性，例如参数和要使用的图标。应用程序图标是默认图标。

在创建部署项目之前，需要检查一些项目属性。如果选择 Project 菜单，再选择 SampleClientStandaloneSetup Properties，就会打开 Project Property Pages 对话框，其中的属性是针对当前配置的。在 Configuration 下拉框中选择配置后，就可以修改表 16-3 中的属性。

表 16-3

属 性	说 明
Output file name	在编译项目时生成的 msi 或 msm 文件的名称
Package file	这个属性允许指定如何打包文件。其选项有： As loose uncompressed files: 所有的部署文件都在同一个目录下存储为 .msi 文件； In setup file: 文件打包到 .msi 文件中(默认设置)； In cabinet files: 文件打包到同一目录下的一个或多个 cab 文件中。选择这个选项时，CAB size 选项就是可用的



(续表)

属 性	说 明
Prerequisites URL	可以指定在哪里查找 .NET Framework 或 Windows Installer 2.0 等必需的程序。单击 Settings 按钮会显示一个对话框，其中包含安装过程中可以使用的技术： Windows Installer 2.0 .NET Framework Microsoft Visual J# .NET Redistributable Package 2.0 SQL Server 2005 Express Edition Microsoft Data Access Components 2.8 还有一个选项，可以从预定义的 URL 上下载必需的程序，或从安装位置加载它们
Compression	这个属性指定所包含文件的压缩样式。其选项如下： Optimized for speed: 用于较大的文件，安装时间较短(默认设置) Optimized for size: 用于较小的文件，安装时间较长 None: 不压缩
CAB size	Package file 属性设置为 In cabinet files 时，这个属性就会被激活。它不仅可以创建一个 cabinet 文件，还可以设置每个 cab 文件的最大尺寸
Authenticode Signature	在选中这个属性时，部署项目的输出就使用 Authenticode 标记，默认设置为不选中
Certificate file	用于签名的证书
Private key file	包含签名文件的数字加密键的私钥
Timestamp server URL	Timestamp 服务器的 URL，也用于 Authenticode 标记

在设置完项目属性后，就应创建部署项目，为 SampleClientApp 应用程序创建安装程序。在建立项目后，就可以在 Solution Explorer 中右击项目名，测试安装了，此时可以在弹出的菜单中访问 Install 和 UnInstall 选项。如果一切正常，就可以成功安装和卸载 SampleClientApp 应用程序。

2. 同一个解决方案项目

上一个示例成功地创建了一个部署软件包，但有几个缺点。例如，在新程序集添加到原应用程序中时会发生什么情况？部署项目不会自动识别任何改动的地方，而必须添加新程序集，再验证新的从属文件已包含进来。在较小的应用程序(如本例)中，这没有什么大不了。但在处理包含几十个甚至上百个程序集的应用程序时，这就可能很难维护了。Visual Studio 2008 为解决这个潜在的问题提供了一个简单的方法，即把部署项目包含在应用程序解决方案中，这样就可以把主项目的输出当作部署程序集了。下面以 SampleClientApp 为例来说明。

在 Visual Studio 2008 中打开 SampleClientApp 解决方案，使用 Solution Explorer 添加一个新项目，选择 Deployment and Setup Projects，再选择 Setup Project，之后按照上一节介绍的步骤进行。可以把这个项目命名为 SampleAppSolutionSetup。在前面的示例中，是在 Project 菜单中选择 Add | Assemblies，添加了程序集，这次在 Project 菜单中选择 Add | Project Output，这会

打开 Add Project Output Group 对话框, 如图 16-2 所示。

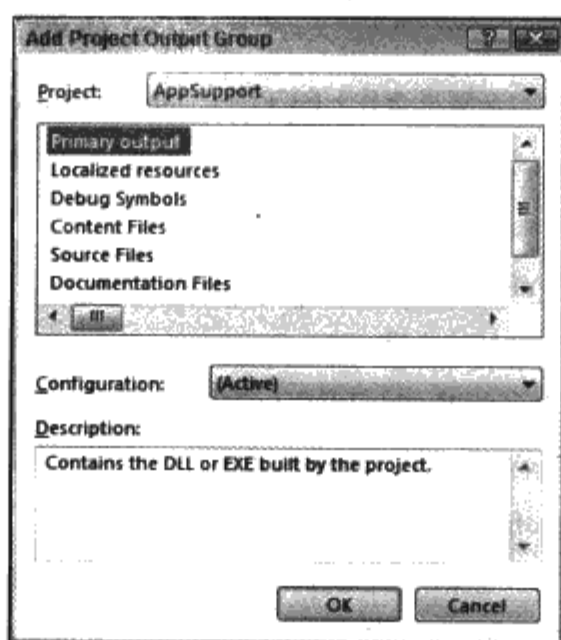


图 16-2

对话框的顶部有一个下拉列表框, 其中显示了当前解决方案中的所有项目。选择主启动项目, 然后从下面的列表中选择要包含在项目中的项, 选项有 Documentation、Primary Output、Localized Resources、Debug Symbols、Content Files 和 Source Files。首先选择 Primary Output, 这包括建立应用程序时的输出和所有的从属文件。对话框中还有一个下拉列表框, 其中列出了有效的配置: Debug、Release 以及自己添加的定制配置。这还确定了提取什么输出。对于部署, 应使用 Release 配置。

在完成这些选择后, 在 Solution Explorer 中就会给部署项目添加一个新条目。该条目的名称是 Primary output form SampleClientApp(Release .NET)。另外, 文件 AppSupport.dll 将出现在从属文件列表中。与以前一样, 不需要搜索从属程序集。

此时, 仍在应用上一节讨论的所有项目属性。可以修改 Name、Manufacturer、cab 文件大小和其他属性。在设置好属性后, 建立解决方案的 Release 版本, 并测试安装。一切应像希望的那样正常工作。

为了理解把部署软件包添加到应用程序解决方案中的优点, 下面把一个新项目添加到解决方案中。在本例中该项目名为 AppSupportII。在该项目中, 有一个简单的测试方法, 它返回字符串 Hello World。在 SampleTestApp 中设置一个对新增项目的引用, 建立解决方案的另一个 Release 版本。部署项目会自动提取新的程序集, 无需我们做任何工作。如果返回去, 打开上一个示例中的独立部署项目, 除非添加程序集, 否则它是不会被提取的。

### 3. 简单的 Web 应用程序

为 Web 应用程序创建安装软件包与创建客户安装软件包没有什么不同。下载的示例包含一个 SampleWebApp 应用程序, 它也利用了 AppSupport.dll 程序集。可以用与创建客户部署项目相同的方式创建部署项目, 即创建独立的部署项目, 或在原解决方案中创建部署项目。在这个示例中, 我们在原解决方案中创建部署项目。

启动 SampleWebApp 解决方案, 添加一个新的 Deployment and Setup 项目。这次要确保在 Templates 窗口中选择 Web Setup Project。如果查看该项目的属性视图, 就会看到 Web 应用程序

拥有与客户应用程序相同的所有属性。唯一新增的属性是 RestartWWWService。这是一个布尔值，用于在安装过程中重新启动 IIS。如果使用 ASP.NET 组件，而且没有替换 ATL 或 ISAPI dll，就不需要修改该属性。

如果查看文件系统编辑器，就会注意其中只有一个文件夹。Web Application 文件夹就是我们的虚拟目录。默认情况下目录名就是部署项目名，位于 Web 根目录下。表 16-4 解释了可以在安装程序中设置的属性。上一节讨论的属性未包括在内。

表 16-4

属 性	说 明
AllowDirectoryBrowsing	布尔值，如果设置为 true，就允许以 HTML 格式列出虚拟目录中的文件和子文件夹。该属性映射为 IIS 的 Directory Browsing 属性
AllowReadAccess	布尔值，如果设置为 true，就允许用户读取或下载文件。该属性映射为 IIS 的 Read 属性
AllowScriptSourceAccess	布尔值，如果设置为 true，就允许用户访问源代码，包括脚本。该属性映射为 IIS 的 Script source access
AllowWriteAccess	布尔值，如果设置为 true，就允许用户修改可写文件中的内容。映射为 IIS 的 Write 属性
ApplicationProtection	确定运行在服务器上的应用程序的保护级别。有效值如下： Low：应用程序与 Web 服务运行在同一个进程中 Medium：应用程序运行在同一个进程中，但不与 Web 服务运行在同一个进程中 High：应用程序运行在它自己的进程中 该属性映射为 IIS 的 Application Protection 属性。如果 IsApplication 属性为 false，则不起作用
AppMappings	列出应用程序名以及与应用程序相关的文档或数据文件。该属性映射为 IIS 的 Application Mappings 属性
Condition	Windows Installer 条件，必须满足该条件，才能安装需要的项
DefaultDocument	用户第一次浏览站点时的默认文档或启动文档
ExecutePermissions	用户执行应用程序必须拥有的许可级别。有效值如下： None：只能访问静态内容 ScriptOnly：只能访问脚本，包括 ASP ScriptAndExecutables：可以访问所有文件 该属性映射为 IIS 的 Execute Permissions
Index	布尔值，如果设置为 true，就允许给 Microsoft Indexing 服务的内容建立索引。该属性映射为 IIS 的 Index this resource 属性
IsApplication	布尔值，如果设置为 true，就让 IIS 为文件夹创建应用程序根目录
LogVisits	布尔值，如果设置为 true，就可以把对 Web 站点的访问记录到日志文件中。该属性映射为 IIS 的 Log visits 属性
Property	可以在安装期间访问的指定属性
VirtualDirectory	应用程序的虚拟目录，这相对于 Web 服务器

注意，大多数属性都是 IIS 的属性，可以在 IIS 管理工具中设置。所以有如下逻辑假设：为了在安装程序中设置这些属性，安装程序在运行时需要拥有管理员权限。这里进行的设置可能会危害到安全，所以对做出的修改要进行很好的说明。

除了这些属性之外，创建部署项目的过程非常类似于前面的客户示例中所介绍的过程。两个项目的主要区别是允许在安装过程中修改 IIS。可以看出，我们对 IIS 环境有很大的控制权。

#### 4. Web 服务器上的客户

另一个安装情况是在 Web 站点上运行安装程序，或在 Web 站点上运行应用程序。如果必须把应用程序部署给大量的用户，这就是两个很有吸引力的选项。在 Web 站点上部署，就不需要部署介质了，如 CD-ROM、DVD，甚或软盘。在 Web 站点甚至网络共享上运行应用程序，就根本不需要发布安装程序了。

在 Web 站点上运行安装程序是相当简单的，使用本章前面讨论的 Web Bootstrapper 项目编译选项即可，此时需要提供安装文件夹的 URL，在这个文件夹中，安装程序会查找需要的 msd 和其他文件。设置好这个选项，并编译部署软件包后，就可以把它复制到 Setup folder URL 属性指定的 Web 站点上。这样当用户导航到这个文件夹时，就可以运行安装程序，或者先下载再运行它。在这两种情况下，用户都必须连接到同一个站点，才能完成安装。

#### 5. 无干涉部署

还可以在 Web 站点或网络共享上运行应用程序。这个过程有点麻烦，这也是在设计应用程序时应考虑部署的一个主要原因。有时这称为无干涉部署(Not Touch Deployment, NTD)。

为了使这个过程顺利完成，应用程序代码必须以支持 NTD 的方式编写。利用 NTD 创建应用程序有两种方式：一种是把大多数应用程序代码放在 dll 程序集中，dll 放在 Web 服务器或网络的文件共享上。然后创建一个要在客户机上部署的小型应用程序可执行文件。这个存根程序(Stub Program)将使用 LoadFrom 方法调用一个 dll 程序集，启动应用程序。存根程序唯一能看到的是 dll 中的主入口。一旦加载了 dll 程序集，应用程序就从同一个 URL 或网络共享中加载其他程序集。程序集首先会在应用程序目录中查找从属程序集，这是用于启动应用程序的 URL。在用户的客户机上，存根应用程序使用的代码如下所示。这个示例调用 AppSupportII.dll 程序集，并把 TestMethod 调用的输出放在 label1 中。

```
Assembly testAssembly =
    Assembly.LoadFrom("http://localhost/AppSupport/AppSupportII.dll");
Type type = testAssembly.GetType("AppSupportII.TestClass");
object testObject = Activator.CreateInstance(type);
label1.Text = (string)type.GetMethod("TestMethod").Invoke(testObject, null);
```

这个过程使用反射技术首先从 Web 服务器上加载程序集。在这个示例中，Web 站点是本地机器(localhost)上的一个文件夹。接着，提取类的类型(这里是 TestClass)。有了类型信息后，就可以使用 Activator.CreateInstance 方法创建对象。最后一步是获取 MethodInfo 对象(GetMethod 的输出)，并调用 Invoke 方法。在比较复杂的应用程序中，这是应用程序的主入口点。从现在开始，就不再需要存根程序了。

另外，还可以把整个应用程序部署到 Web 站点上。对于这种方法，应创建一个简单的 Web 页面，其中包含一个到应用程序的安装可执行文件的链接，或者在用户桌面上有一个包含 Web

站点链接的快捷方式。单击这个链接，应用程序就会下载到用户的程序集下载缓存中，该缓存位于 Global Assembly Cache 中。应用程序从下载缓存中运行。每次请求新程序集时，都需要进入下载缓存，检查该程序集是否存在。如果不存在，就进入获取主应用程序的 URL。

以这种方式部署应用程序的优点是，在能对应用程序进行更新时，该更新版本只需部署到一个地方。我们把新程序集放在 Web 文件夹中，当用户启动应用程序时，运行库会比较 URL 中的程序集版本和下载缓存中的程序集版本，如果在 URL 中找到新版本，就下载它，替换下载缓存中的当前版本。这样，用户将总是访问应用程序的最新版本。其缺点是很难保证安全性。程序集必须有许多权限才能执行。这会使应用程序非常不安全。

要对更新过程 and 安全性进行更多的控制，ClickOnce 是一个比较好的选项。

16.7 ClickOnce

ClickOnce 是一种允许应用程序自动升级的部署技术。应用程序发布到文件共享、Web 站点或 CD 这样的媒介上。之后，ClickOnce 应用程序就可以自动升级，而无需用户的干涉。

ClickOnce 还解决了安全权限问题。一般情况下，要安装应用程序，用户需要有管理权限。而利用 ClickOnce，用户只要有运行应用程序所需的最低权限，就可以安装和运行应用程序。

16.7.1 ClickOnce 操作

ClickOnce 应用程序有两个基于 XML 的清单文件，其中一个是应用程序的清单，另一个是部署清单。这两个文件描述了部署应用程序所需的所有信息。

应用程序清单包含的应用程序信息有需要的权限、要包括的程序集和其他从属文件。部署清单包含了应用程序的部署信息。应用程序清单的位置信息包含在部署清单中。这些清单的完整模式在 .NET SDK 文档说明中。

ClickOnce 有一些限制，例如，程序集不能添加到 GAC 中。表 16-5 比较了 ClickOnce 和 Windows Installer。

表 16-5

	ClickOnce	Windows Installer
应用程序的安装位置	ClickOnce 应用程序缓存	Program Files 文件夹
给多个用户安装	否	是
安装共享文件	否	是
安装驱动程序	否	是
安装到 GAC 中	否	是
在“启动”组中添加应用程序	否	是
在 favorites 菜单中添加应用程序	否	是
注册文件类型	否	是
访问注册表	否。有访问 HKLM 的 Full Trust 权限	是
文件的二进制修补	是	否
根据需要安装程序集	是	否



在一些情况下，使用 Windows Installer 比较好，但 ClickOnce 也适用于许多应用程序。

### 16.7.2 发布应用程序

ClickOnce 需要知道的信息都包含在两个清单文件中。为 ClickOnce 部署发布应用程序的过程就是生成清单，把文件放在正确的位置。清单文件可以在 Visual Studio 2008 中生成，还可以使用一个命令行工具 `mage.exe`，它还有一个带 GUI 的版本 `mageUI.exe`。

在 Visual Studio 2008 中创建清单文件有两种方式。在 Project Properties 对话框的 Publish 选项卡底部有两个按钮，一个是 Publish Wizard，另一个是 Publish Now。Publish Wizard 要求回答几个应用程序的部署问题，然后生成清单文件，把所有需要的文件复制到部署位置。Publish Now 按钮使用在 Publish 选项卡中设置的值，创建清单文件，并把文件复制到部署位置。

为了使用命令行工具 `mage.exe`，必须传送各个 ClickOnce 属性的值。使用 `mage.exe` 可以创建和更新清单文件。在命令提示中输入 `mage.exe .help`，就会显示传送所需值的语法。

`mage.exe` 的 GUI 版本(`mageUI.exe`)类似于 Visual Studio 2008 中的 Publish 选项卡。使用 GUI 工具可以创建和更新应用程序清单和部署清单文件。

ClickOnce 应用程序会显示在“添加/删除程序”控制面板选项上，这与其他安装的应用程序一样。一个主要区别是用户可以选择卸载应用程序或回退到以前的版本。ClickOnce 在 ClickOnce 应用程序缓存中保存以前的版本。

### 16.7.3 ClickOnce 设置

两个清单文件都有几个属性。最重要的属性是应用程序应从什么地方部署。必须指定应用程序的从属文件。Publish 选项卡上有一个 Application Files 按钮，单击它会打开一个对话框，输入应用程序需要的所有程序集。单击 Prerequisite 按钮会显示一个与应用程序一起安装的通用预装程序列表。可以选择从发布应用程序的位置上安装预装程序，也可以从供应商的 Web 站点上安装预装程序。

单击 Update 按钮会显示一个对话框，其中包含了如何更新应用程序的信息。当有应用程序的新版本时，可以使用 ClickOnce 更新应用程序。其选项包括：每次启动应用程序时检查是否有更新版本，或在后台检查更新版本。如果选择了后台选项，就可以输入两次检查的间隔时间。此时可以使用允许用户拒绝或接收更新版本的选项。它可用于在后台进行更新，这样用户就不知道进行了更新。下次运行应用程序时，会使用新版本替代旧版本。还可以给更新文件使用另一个位置存储。这样，原安装软件包在一个位置，用于给新用户安装应用程序，而所有的更新版本放在另一个位置上。

安装应用程序时，可以让它以在线模式或离线模式下运行。在离线模式下，应用程序可以从“开始”菜单中运行，就好像它是用 Windows Installer 安装的。在线模式表示应用程序只能在有安装文件夹的情况下运行。

### 16.7.4 应用程序缓存

用 ClickOnce 发布的应用程序不能安装在 Program Files 文件夹中，它们会放在应用程序缓

存中，应用程序缓存位于当前用户的 Document's and Settings 文件夹的 Local Settings 子文件夹下。控制部署的这个方面，可以把应用程序的多个版本同时放在客户机上。如果应用程序设置为在线运行，就会保留用户访问过的每个版本。对于设置为本地运行的应用程序，会保留当前版本和以前的版本。

所以，把 ClickOnce 应用程序回退到以前的版本是一个非常简单的过程。如果用户进入“添加/删除程序”控制面板选项，所显示的对话框将允许删除 ClickOnce 应用程序或回退到以前的版本。管理员可以修改清单文件，使之指向以前的版本。之后，下次用户启动应用程序时，会检查是否更新版本。应用程序不是查找要部署的新程序集，而是恢复以前的版本，但不需要用户的干涉。

### 16.7.5 安全性

通过 Internet 或内联网部署的应用程序，其安全性或可信赖设置比安装到本地驱动器上的应用程序低。例如，如果应用程序从 Internet 上启动或部署，它就默认位于 Internet Security 区域。也就是说，它不能访问文件系统。如果应用程序是从文件共享中安装的，就在 Intranet 区域中运行。

如果应用程序需要的信赖度高于默认值，它就会要求用户获得运行应用程序所需的权限。这些权限在应用程序清单的 trustInfo 元素中设置。只需要授予这个设置中的权限。如果应用程序需要文件访问权限，就不需要授予 Full Trust 权限，只要文件访问权限即可。

另一个选项是使用 Trusted Application Deployment。Trusted Application Deployment 是给整个企业授予权限的方式，不需要提示用户。给每台客户机标记一个信任许可发程序，这可以通过公钥加密法完成。一般一个公司只有一个信任许可发程序。一定要把发程序的私钥放在安全的地方。

信任许可从发程序上请求。所请求的信任级别是信任许可配置的一部分。还必须给许可发程序提供用于标记应用程序的公钥。所创建的许可包含用于标记应用程序的公钥和许可发程序的公钥。这个信任许可会嵌入到部署清单中。最后一步是用自己的密钥对标记部署清单。现在应用程序就可以部署了。

在客户打开部署清单时，Trust Manager 会确定 ClickOnce 应用程序是否有较高的信任级别。首先查看发程序的许可。如果它是有效的，就比较许可中的公钥和用于标记应用程序的公钥。如果它们匹配，就给应用程序授予需要的权限。

### 16.7.6 高级选项

前面讨论的安装过程功能非常强大，可以完成许多工作。在安装过程中还可以控制许多方面。例如，可以使用 Visual Studio 2008 中的各种编辑器建立条件安装，或者添加注册键和定制对话框。SampleClientSetupSolution 示例就启用了所有这些高级选项。

#### 1. 文件系统编辑器

文件系统编辑器允许指定组成应用程序的各种文件和程序集部署到目标机器的什么地方。默认情况下会显示一组标准的部署文件夹。使用该编辑器可以添加任意多个定制和特定文件

夹。还可以给应用程序添加桌面快捷方式和“开始”菜单快捷方式。组成部署的任何文件都必须在文件系统编辑器中引用。

## 2. 注册编辑器

注册编辑器允许给注册表添加键和数据。在第一次显示该编辑器时，会显示一组标准的主键：

- HKEY\_CLASSES\_ROOT
- HKEY\_CURRENT\_USER
- HKEY\_LOCAL\_MACHINE
- HKEY\_USERS

HKEY\_CURRENT\_USER 和 HKEY\_LOCAL\_MACHINE 在 Software/[manufacturer]键中包含了额外的条目，其中 manufacturer 是在部署项目的 Manufacturer 属性中输入的信息。

要添加额外的键和值，应在编辑器的左边突出显示一个主键，从主菜单中选择 Action，再选择 New。选择要添加的键或值类型。重复这个步骤，直到得到了需要的所有注册设置为止。如果在左边的窗格中选择了 Registry on Target Machine 选项，再选择 Action 菜单，就会看到一个 Import 选项，该选项允许导入已定义好的.reg 文件。

要按键创建默认值，必须先为该键输入值。然后在右边或值窗格上选择值名称。从 File 菜单中选择 Rename，并删除该名称。按下回车键，值名称就替换为(Default)。

还可以在该编辑器中为子键和值设置一些属性。前面还没有讨论的唯一属性是 DeleteAt-Uninstall。设计良好的应用程序应在卸载过程中删除由该应用程序添加的所有键。默认设置是不删除键。

注意，维护应用程序设置的首选方法是使用基于 XML 的配置文件。与注册表项相比，这些文件提供了非常大的灵活性，更容易恢复和备份。

## 3. 文件类型编辑器

文件类型编辑器用于建立文件和应用程序之间的关系。例如，在双击.doc 文件时，就会在 Word 中打开该文件。使用这个编辑器可以为应用程序创建这种关系。

为了添加关系，执行下面的步骤：

- (1) 从 Action 菜单中选择 File Types on Target Machine。
- (2) 然后选择 Add File Type。在属性窗口中，可以设置关系的名称。
- (3) 在 Extension 属性中添加应与应用程序相关的文件扩展名。不要输入句点，可以用分号隔开多个扩展名，例如 ex1;ex2。
- (4) 在 Command 属性中选择省略号按钮。
- (5) 接着选择要与特定的文件类型相关的文件(一般是可执行文件)。注意任何一个扩展名都应只与一个应用程序相关。

默认情况下，编辑器会显示&Open as the Document Action。我们还可以添加其他选项。编辑器中显示的动作顺序就是用户右击文件类型时在弹出的菜单中显示的动作顺序。注意第一项

总是默认动作。可以为动作设置 **Argument** 属性，这是用于启动应用程序的命令行参数。

#### 4. 用户界面编辑器

有时在安装过程中要求用户提供更多的信息。用户界面编辑器可用于为一组预定义的对话框指定属性。该编辑器分为两个部分 **Install** 和 **Admin**。一个用于标准安装，另一个用于管理员安装。每个部分又分为 3 个子部分：**Start**、**Progress** 和 **End**。这些子部分表示安装过程的 3 个基本阶段，如图 16-3 所示。

表 16-6 列出了可以添加到项目中的对话框类型。

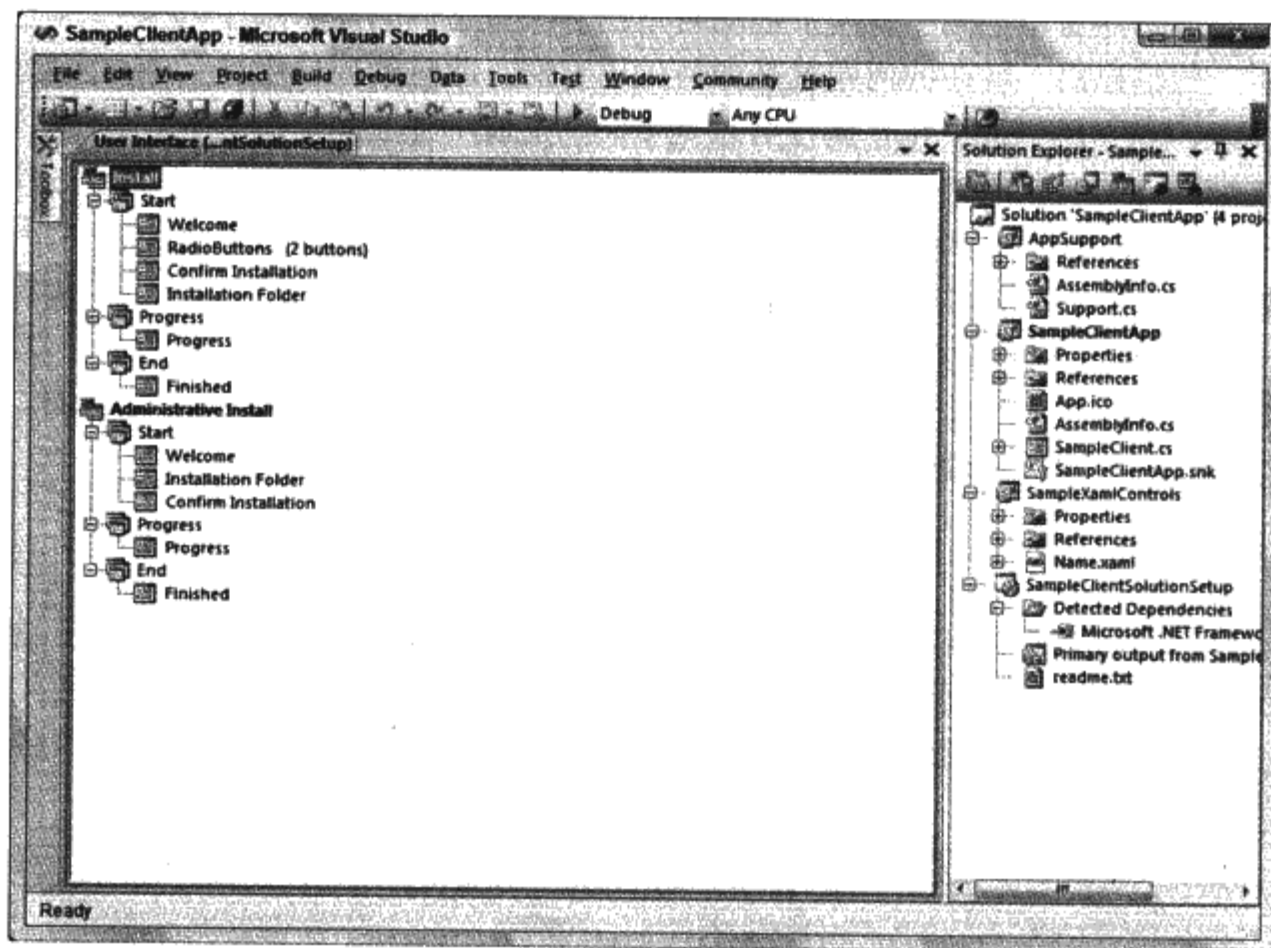


图 16-3

表 16-6

对 话 框	说 明
Checkboxes	至多包含 4 个复选框。每个复选框都包含 <b>Lable</b> 、 <b>Value</b> 和 <b>Visible</b> 属性
Confirm Installation	允许用户在安装开始之前确认各个设置
Customer Information	包含集团名称、公司名称和系列号的编辑字段。公司名称和系列号是可选的
Finished	在安装过程的最后显示
Installation Address	用于 Web 应用程序，显示一个对话框，让用户选择另一个安装 URL
Installation Folder	用于客户应用程序，显示一个对话框，让用户选择另一个安装文件夹
License Agreement	显示许可协议，该协议位于 <b>LicenseFile</b> 属性指定的文件中
Progress	在安装过程中显示一个进度指示器，说明当前的安装状态
RadioButtons	至多包含 4 个单选按钮，每个单选按钮都包含 <b>Lable</b> 和 <b>Value</b> 属性
Read Me	显示 <b>readme</b> 信息，该信息包含在 <b>ReadMe</b> 属性指定的文件中

(续表)

对 话 框	说 明
Register User	执行一个在注册过程中指导用户操作的应用程序，该应用程序必须在安装项目中提供
Splash	显示一个位图图像
TextBoxes	至多包含 4 个文本框，每个文本框都包含 Lable、Value 和 Visible 属性
Welcome	包含两个属性 WelcomeText 和 CopyrightWarning，它们都是字符串属性

这些对话框还包含一个设置横幅位图的属性，大多数对话框还包含一个设置横幅文本的属性。还可以在编辑器窗口中向上或向下拖动对话框，改变它们的显示顺序。

获得了一些信息后，现在的问题就是如何使用它们。此时就要使用项目中大多数对象都包含的 Condition 属性了。Condition 属性必须是 true，安装步骤才能继续下去。例如，假定安装程序包含 3 个可选的安装组件。在这种情况下，就可以添加一个对话框，其中包含 3 个复选框。该对话框应在 Welcome 对话框之后、Confirm Installation 对话框之前的某个地方显示。修改每个复选框的 Label 属性，描述具体的动作。第一个动作是“安装组件 A”，第二个动作是“安装组件 B”，依次类推。在文件系统编辑器中选择表示组件 A 的文件。假定对话框中复选框的名称是 CHECKBOXA1，则文件的 Condition 属性就是 CHECKBOXA1=Checked，即如果 CHECKBOXA1 被选中，就安装文件，否则就不安装。

5. 定制动作编辑器

定制动作编辑器允许定义在安装的某些阶段进行的定制步骤。定制动作应事先创建好，它可以包含 DLL、EXE、脚本或 Installer 类。动作可以包含不能在标准部署项目中定义的特定步骤。动作应在部署的 4 个特定点执行。在第一次启动编辑器时，就会看到项目中的这 4 个点，如图 16-4 所示。

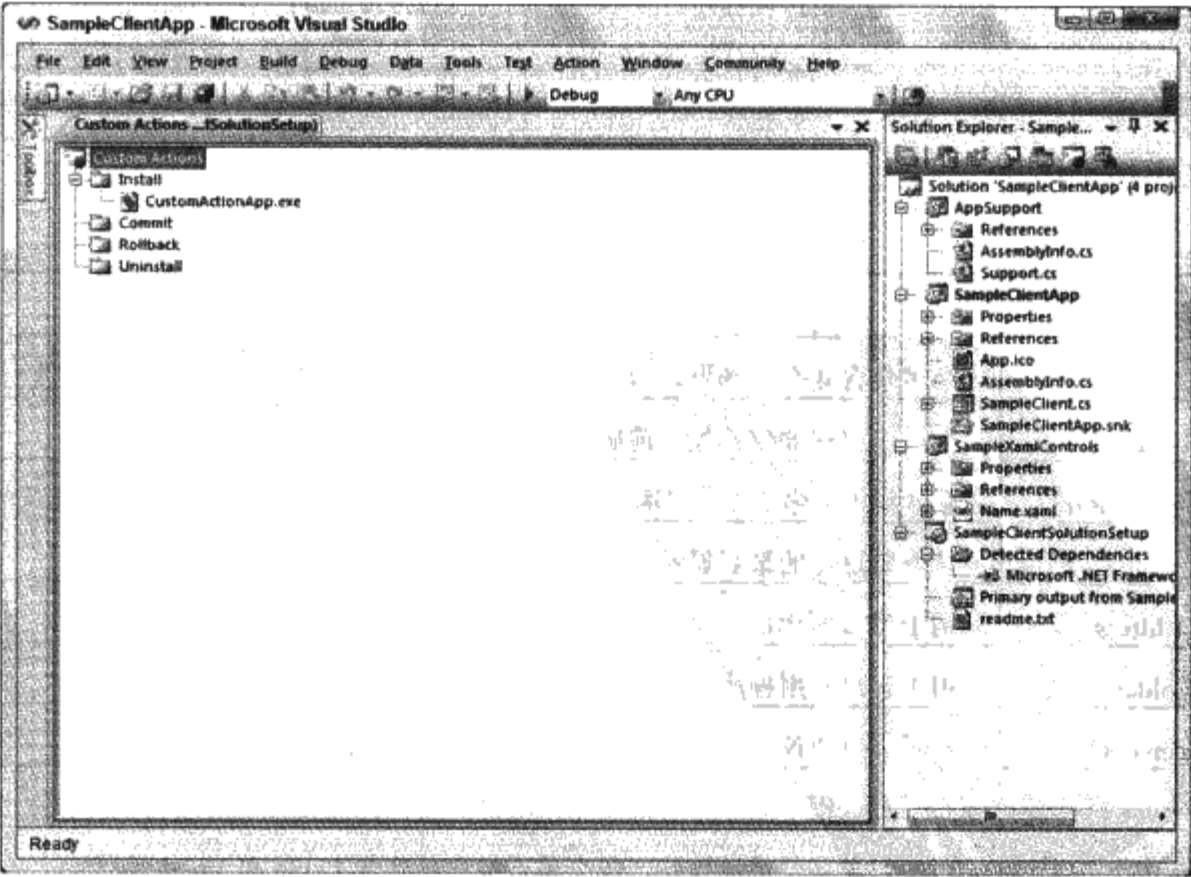


图 16-4



- Install: 动作在安装阶段的最后执行
- Commit: 动作在安装完成后执行, 不记录错误
- Rollback: 动作在回退阶段完成后执行
- Uninstall: 动作在卸载完成后执行

要添加动作, 首先选择要执行动作的安装阶段。再从 Action 菜单中选择 Add Custom Action 菜单选项, 打开文件系统对话框, 这表示包含动作的组件必须是部署项目的一部分。由于动作在要部署的目标机器上执行, 所以应列在文件系统编辑器中。

在添加完动作后, 可以从表 16-7 中选择一个或多个属性。

表 16-7

参 数	命令行参数
Condition	Windows Installer 条件, 若要执行动作, 该条件必须为 true
CustomDataAction	可用于动作的定制数据
EntryPoint	包含动作的定制 DLL 的入口。如果动作包含在可执行文件中, 这个属性就不起作用
InstallerClass	布尔值, 如果设置为 true, 就指定动作是一个 .NET 类 ProjectInstaller
Name	动作的名称, 默认为动作的文件名
SourcePath	动作在开发机器上的路径

由于动作是在部署项目外部开发的代码, 所以可以给应用程序自由添加专业化的外观。但要注意这些动作都在相关的阶段完成后发生。如果选择 Install 阶段, 动作就会在安装阶段完成后发生。如果要在该过程之前执行动作, 就应创建一个启动条件。

6. 启动条件编辑器

启动条件编辑器允许指定在安装继续之前必须满足的一些条件。启动条件可以分为不同的条件类型。基本启动条件是 File Search、Registry Search 和 Windows Installer Search。在编辑器第一次启动时, 会看到两个组 Search Target Machine 和 Launch Conditions, 如图 16-5 所示。一般需要进行搜索, 根据该搜索的成功或失败来执行条件。这是通过设置搜索的 Property 属性来实现的。可以在安装过程中访问 Property 属性, 在其他动作的 Condition 属性中也可以检查该属性。还可以在编辑器中添加启动条件。在这个条件中把 Condition 属性设置为搜索的 Property 属性值。在条件中可以指定一个 URL, 用于下载所搜索的文件、注册键或安装组件。注意在图 16-5 中, 默认添加了一个 .NET Framework 条件。

File Search 搜索文件或文件类型。可以设置许多不同的、与文件相关的属性, 来确定如何搜索文件, 这些属性包括文件名、文件夹位置、各种日期值, 版本信息和大小。还可以设置要搜索的子文件夹数目。

Registry Search 允许搜索键和值, 还允许设置搜索的根键。

Windows Installer Search 搜索指定的安装组件。这个搜索由 GUID 执行。

启动条件编辑器提供了两个预打包的启动条件, 一个是 .NET Framework 启动条件, 它允许搜索运行库的特定版本, 另一个启动条件是搜索 MDAC 的特定版本, 该搜索使用注册表搜索,

来查找相关的 MDAC 注册表项。

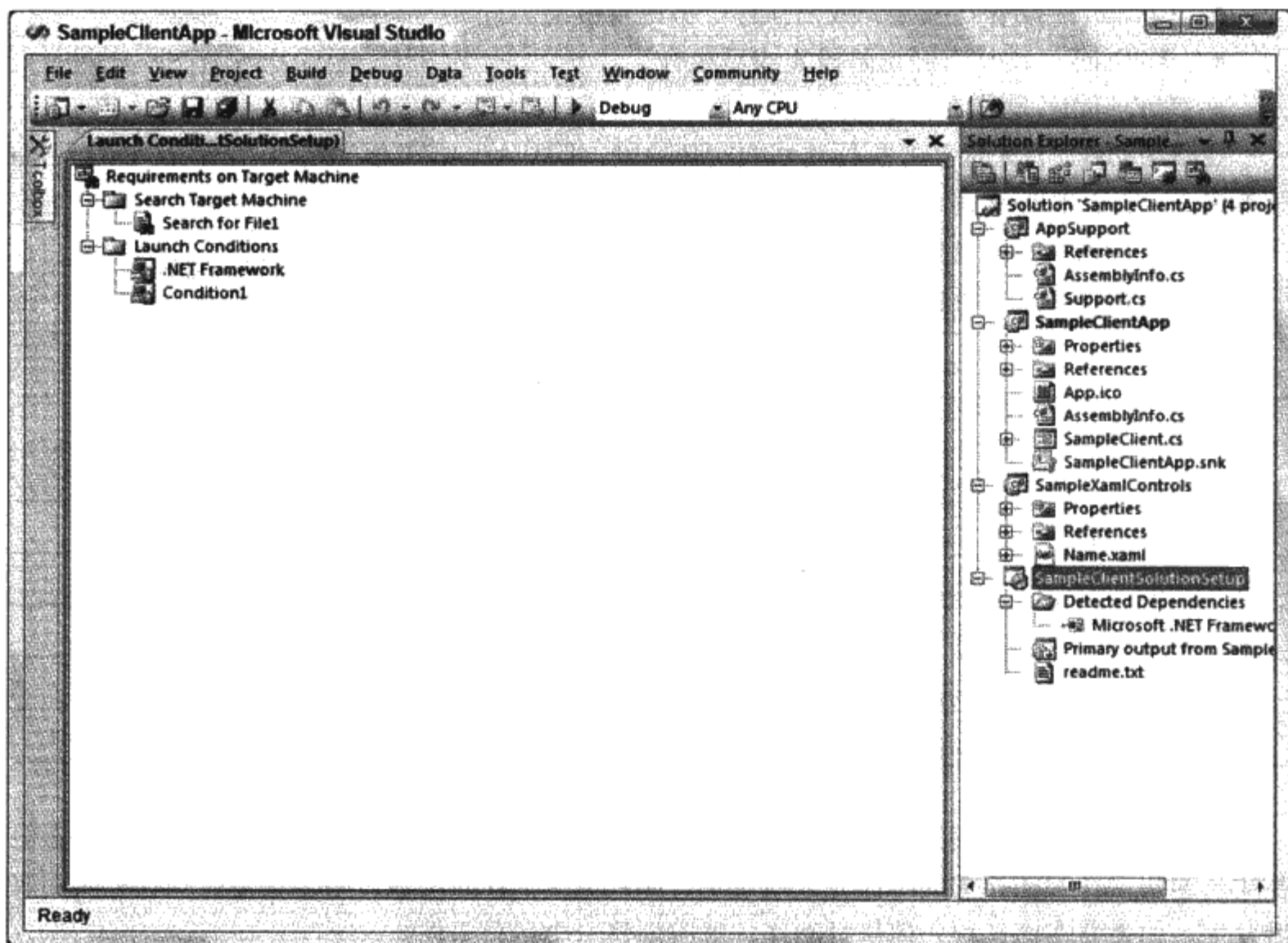


图 16-5

## 16.8 小结

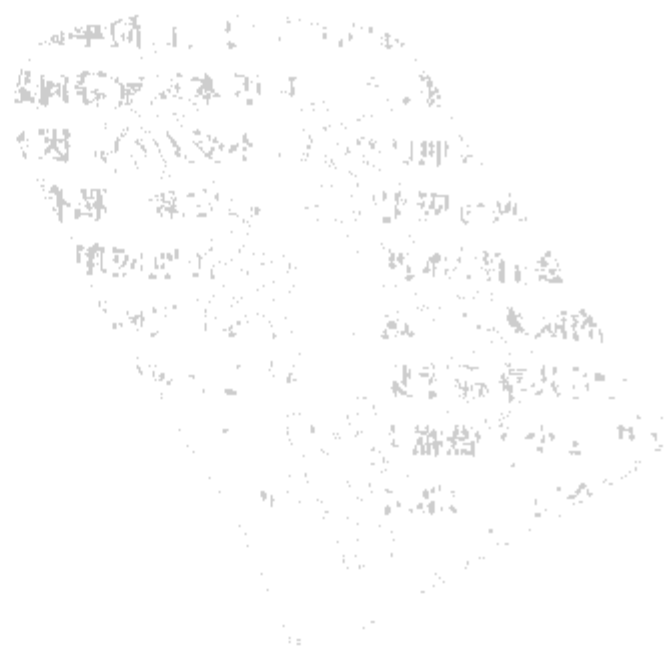
部署软件对桌面软件的开发人员来说比较困难。随着 Web 站点越来越复杂，部署基于服务器的软件也变得困难起来。本章探讨了 Visual Studio 2008 和 .NET Framework 3.5 版本的部署选项和功能；使部署更容易完成，错误也较少。

在阅读完本章后，您应能创建部署软件包，解决几乎所有的部署问题。客户应用程序可以在本地部署，或通过 Internet 或内联网来部署。本章还介绍了部署项目的扩展特性和部署项目的配置方式。我们可以使用 No Touch Deployment 和 ClickOnce 部署应用程序。ClickOnce 的安全特性为部署客户应用程序提供了一种安全、有效的方式。使用部署项目安装 Web 应用程序还可以更轻松地完成 IIS 的配置。如果预编译应用程序，发布 Web 站点还有额外的好处。

## 第Ⅲ部分

# 基 类 库

- 第 17 章 程序集
- 第 18 章 跟踪和事件
- 第 19 章 线程和同步
- 第 20 章 安全性
- 第 21 章 本地化
- 第 22 章 事务处理
- 第 23 章 Windows 服务
- 第 24 章 互操作性



# 第 17 章

## 程 序 集

程序集是.NET 用于部署和配置单元的术语。本章主要讨论什么是程序集，如何使用它们，它们的功能为什么这么强大。本章的主要内容包括：

- 概述
- 创建程序集
- 应用程序域
- 共享程序集
- 版本问题

首先概述程序集。

### 17.1 程序集的含义

程序集是.NET 应用程序的部署单元。.NET 应用程序包含一个或多个程序集。通常扩展名是 EXE 或 DLL 的.NET 可执行程序称为程序集。程序集和一般的 DLL 或 EXE 有什么区别？它们的文件扩展名虽然相同，但.NET 程序集包含元数据，这些元数据描述了程序集中定义的所有类型及其成员的信息，例如方法、属性、事件和字段。

.NET 程序集的元数据还提供了程序集中文件的信息、版本信息和所使用的程序集的信息。.NET 程序集是为以前内部 DLL 的 DLL Hell 提供的解决方案。

程序集是自我描述的安装单元，由一个或多个文件组成。一个程序集可以是一个包括元数据的 DLL 或 EXE，也可以由多个文件组成，例如资源文件、模块和 EXE。

程序集可以是私有或共享的。在简单的.NET 应用程序中，仅使用私有程序集即可工作。私有程序集没有管理、注册和版本设置等问题，只有用户自己的应用程序在使用私有程序集时才有版本问题。其他应用程序不受影响，因为它们有自己的程序集副本。在这种应用程序中使用的私有组件应与应用程序一起安装。私有程序集位于应用程序所在的目录或子目录下，所以应用程序不会有版本冲突问题。其他应用程序都不会重写私有的程序集。当然，仍可以使用私有程序集的版本号。这非常有助于代码的修改，但它不是.NET 所必需的。

在使用共享程序集时，有几个应用程序都使用这个程序集，且与它有一定的依赖关系。共享程序集减少了磁盘和内存空间的需求。使用共享程序集时，要遵循许多规则。共享程序集必须有一个特殊的版本号、唯一的名称，通常安装在全局程序集缓存(global assembly cache, GAC)中。

### 17.1.1 程序集的特性

程序集的特性可以总结如下：

- 程序集是自我描述的。不再需要考虑注册表键、从其他地方获得类库等问题，程序集包含描述程序集的元数据，元数据包括从程序集中导出的类和一个清单。下一节将介绍清单。
- 版本的相互依赖性在程序集的清单中进行了记录。任何被引用的程序集的版本存储在程序集的清单中，这样就很容易确定因错误的版本号而引起的部署失败了。以后使用的引用程序集版本可以由开发人员和系统管理员配置。在本章后面的一节中，将介绍可用的版本策略及其工作方式。
- 程序集可以并行加载。使用 Windows 2000，就可以获得并行功能，同一个 DLL 的不同版本可以在系统上同时使用。.NET 扩展了 Windows 2000 的这个功能：现在同一个程序集的不同版本也可以在一个进程中使用！那么这有什么用呢？如果程序集 A 引用共享程序集 Shared 的版本 1，程序集 B 引用共享程序集 Shared 的版本 2，而用户同时使用程序集 A 和程序集 B，则应用程序需要使用共享程序集 Shared 的这两个版本，在 .NET 中，应加载和使用两个版本。
- 应用程序使用应用程序域(Application Domain)来确保其独立性。使用应用程序域，许多应用程序就可以独立地运行在一个进程中。一个应用程序中的错误不会直接影响同一个进程中的其他应用程序。
- 安装非常简单，只需复制一个程序集中的所有文件，一个 xcopy 命令就足够了。这个特性称为 ClickOnce 部署。但是在一些情况下不能进行 ClickOnce 部署，而需要正常的 Windows 安装。第 16 章讨论应用程序的部署。

### 17.1.2 程序集的结构

程序集由描述它的元数据、描述导出类型和方法的类型元数据、MSIL 代码和资源组成。这些部分都在一个文件中，或者分布在几个文件中。

在第一个例子中，程序集的元数据、类型元数据、MSIL 代码和资源都在一个文件 Component.dll 中，这个程序集由一个文件组成，如图 17-1 所示。

第二个例子介绍的是分布在 3 个文件中的一个程序集。Component.dll 包含了程序集的元数据、类型元数据和 MSIL 代码，但不包含资源。这个程序集使用了一个图 picture.jpeg，该图没有嵌在 Component.dll 中，而是在程序集的元数据中引用。程序集的元数据还引用了一个模块 util.netmodule，该模块只包含一个类的类型元数据和 MSIL 代码，不包含程序集的元数据，所以这个模块没有版本信息；也不能单独安装。第二个例子中的这 3 个文件构成了一个程序集，这个程序集是一个安装单元，还可以在另一个文件中放置程序集清单，如图 17-2 所示。



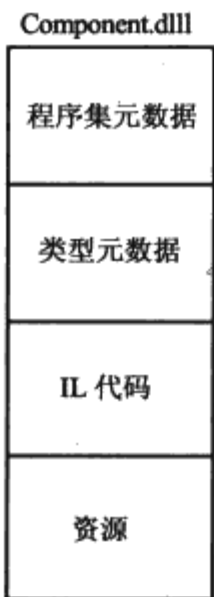


图 17-1

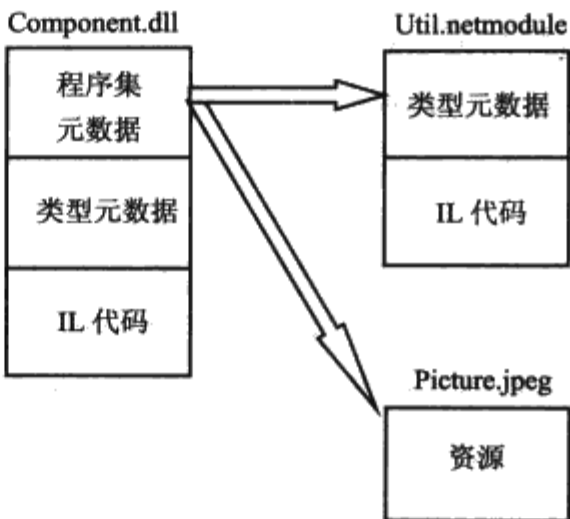


图 17-2

17.1.3 程序集的清单

程序集的一个重要部分是程序集清单，它是元数据的一部分，描述了程序集和引用它所需要的所有信息，并列出了所有的依赖关系。清单由以下部分组成：

- 标识(名称、版本、文化和公钥)。
- 属于该程序集的一个文件列表。一个程序集至少要有一个文件，也可以包含许多个文件。
- 引用程序集的列表。在程序集清单中说明了在程序集中使用的所有程序集，包括版本号和公钥。公钥用于唯一地标识程序集。后面将讨论公钥。
- 一组许可请求——运行这个程序集需要的许可。本章不介绍许可，而是在第 20 章中介绍。
- 导出的类型，假定它们在一个模块中定义，该模块在程序集中引用，程序集就包含它们，否则它们就不是程序集清单的一部分。模块是可重用的单元。类型描述与元数据一样也存储在程序集中，使用属性和方法可以从这些元数据中获得结构和类，它替代了以前用 COM 描述类的类库。使用 COM 客户机很容易在程序集清单的外部生成一个类库。反射机制使用导出类的信息，对类进行后期绑定。有关反射的内容，请参见第 13 章。

17.1.4 命名空间、程序集和组件

也许您目前会混淆命名空间、类、程序集和组件。命名空间如何与程序集的概念相匹配？命名空间完全独立于程序集。在一个程序集中可以有不同的命名空间，一个命名空间也可以分布在多个程序集中。命名空间只是类名的一种扩展，它属于类名的范畴。

例如，程序集 mscorlib 和系统都包含命名空间 System.Threading 和其他命名空间。尽管程序集包含相同的命名空间，但没有相同的类名。

### 17.1.5 私有程序集和共享程序集

程序集可以是共享的，也可以是私有的。私有程序集位于应用程序所在的目录下，或其子目录中。使用私有程序集时，不需要考虑与其他类的命名冲突或版本冲突问题。在构建过程中引用的程序集会复制到应用程序的目录下。私有程序集是构建程序集的一般方式，特别是应用程序和组件在同一个公司中建立时，就更是如此。

**提示：**

私有程序集可能仍有命名冲突（应用程序可能包含多个程序集，这些程序集是有冲突的，例如一个私有程序集中的名称与应用程序使用的一个共享程序集中的名称冲突），但命名冲突会大大减少。如果使用了多个私有程序集或使用了其他应用程序中的共享程序集，最好利用名称正确的命名空间和类型，使命名冲突降低到最少。

在使用共享程序集时，必须遵循一些规则。程序集必须是唯一的，因此，必须有一个唯一的名称（称为强名）。强名的一部分是一个强制的版本号。当组件由另一个开发商构建，而不是应用程序的开发商构建时，以及一个大应用程序分布在几个小项目中时，常常需要使用共享程序集。另外，一些技术例如 .NET Enterprise Services 需要在特定的情形下使用共享程序集。

### 17.1.6 辅助程序集

辅助程序集是只包含资源的程序集，它尤其适用于本地化。程序集有一个相关的文化，所以资源管理器会查找包含特定文化资源的辅助程序集。

**提示：**

辅助程序集的更多信息可参见第 21 章。

### 17.1.7 查看程序集

程序集可以使用命令行工具 `ildasm` 来查看，这是一个 MSIL 反汇编程序。在命令行上运行 `ildasm`，把程序集作为其参数，或者选择 `File | Open` 菜单，就可以打开程序集。

图 17-3 是 `ildasm` 打开的一个简单程序 `SharedDemo.dll`，我们后面会创建这个程序。`ildasm` 显示了程序集清单，以及 `Wrox.ProCSharp.Assemblies.Sharing` 命名空间中的类 `SharedDemo`。打开该程序集清单，就可以看到版本号、程序集的属性 and 引用的程序集及其版本。打开类的方法，就可以查看 MSIL 代码。

**注意：**

除了 `ildasm` 之外，.NET Reflector 是另一个用于分析程序集的强大工具。NET Reflector 可以搜索类型和成员，调用图，将 IL 代码反编译为 C#、C++ 或 Visual Basic。这个工具可以从 <http://www.aisto.com/roeder/dotnet> 上下载。

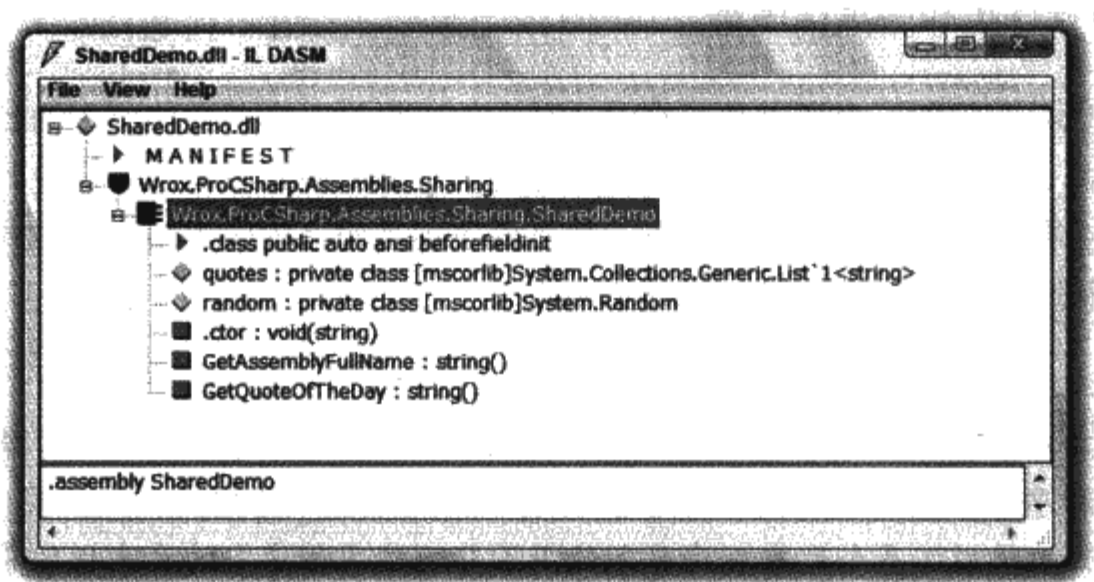


图 17-3

## 17.2 构建程序集

前面学习了程序集的含义，下面就要构建一些程序集了。当然，本书前面已经构建了一些程序集，因为.NET可执行程序也是一个程序集，但下面要介绍一下程序集的特定选项。

### 17.2.1 创建模块和程序集

在 Visual Studio 中，所有的 C# 项目类型都会创建一个程序集。无论是选择 DLL，还是 EXE 项目类型，都会创建一个程序集。使用命令行 C# 编译器 `csc`，也可以创建模块。模块是一个没有程序集特性的 DLL(所以它不是程序集，但可以在以后添加到程序集中)。命令

```
csc /target:module hello.cs
```

创建模块 `hello.netmodule`，可以使用 `ildasm` 查看这个模块。

模块也有一个清单，但在该清单中没有 `.assembly` 条目(除了引用的外部程序集之外)，因为模块没有程序集特性。不能用模块来配置版本或许可，而只能在程序集的范围内进行。在模块的清单中有程序集的引用。使用 `csc` 的 `/addmodule` 选项，可以把模块添加到现有的程序集中。

为了比较模块和程序集，下面生成一个简单的类 A，并用下面的命令编译它：

```
csc /target:module A.cs
```

编译器生成了文件 `A.netmodule`，它不包括程序集的信息(使用 `ildasm` 可以查看清单信息)，模块的清单显示了所引用的程序集 `mscorlib` 和 `.module` 条目，如图 17-4 所示。

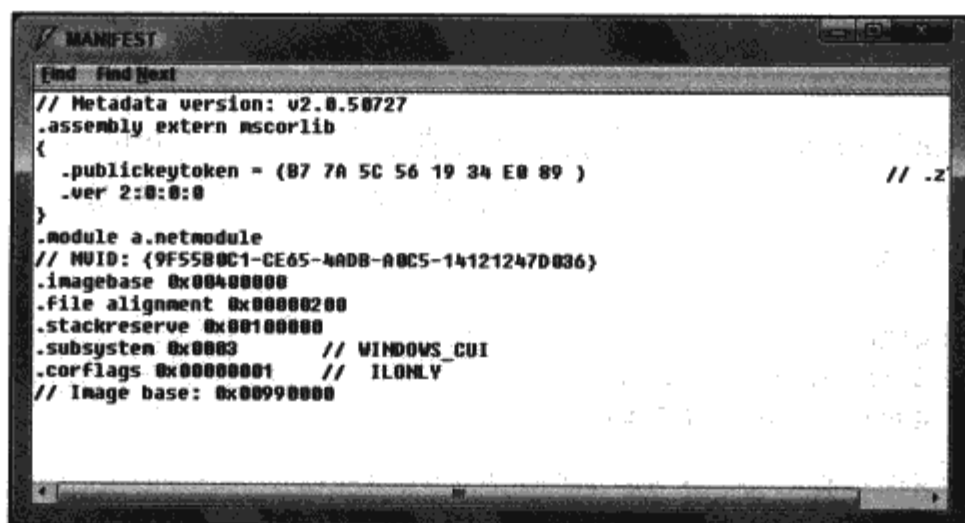


图 17-4

下面生成一个程序集 B，它包括模块 A.netmodule。不需要用一个源文件来生成这个程序集，建立该程序集的命令如下：

```
csc /target:library /addmodule:A.netmodule /out:B.dll
```

在使用 ildasm 查看程序集时，只能找到一个清单。在清单中，引用了程序集 mscorlib。接着看看带有散列算法和版本的程序集部分。算法的数量决定了用于创建程序集的散列代码的算法类型。在编程创建程序集时，可以选择该算法。清单包含属于该程序集的所有模块的一个列表。下面是属于该程序集的 module A.netmodule，从模块中导出的类是程序集清单的一部分，从程序集本身导出的类则不是，如图 17-5 所示。

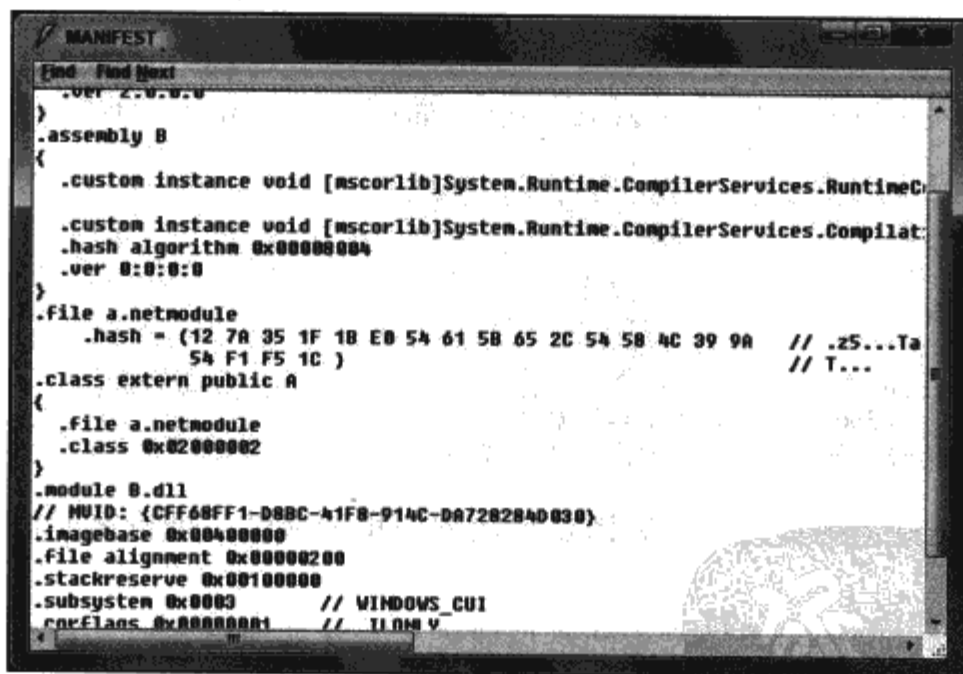


图 17-5

模块的作用是什么？模块可以更快地启动程序集，因为并不是所有的类都在一个文件中。模块只在需要时加载。使用模块的另一个原因是，要用多种编程语言来创建一个程序集：一个模块用 Visual Basic 编写，另一个模块用 C#编写，这两个模块都包括在一个程序集中。

## 17.2.2 程序集的属性

在创建一个 Visual Studio 项目时,会自动生成源文件 AssemblyInfo.cs,这个文件在 Solution Explorer 的 Properties 中。在该文件中,可以使用一般的源代码编辑器配置程序集的属性,下面是从项目模板中生成的一个文件:

```
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;
//
// General Information about an assembly is controlled through the
// following set of attributes. Change these attribute values to modify
// the information associated with an assembly.
//
[assembly: AssemblyTitle("DomainTest")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("DomainTest")]
[assembly: AssemblyCopyright("Copyright © Wrox Press 2007")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]

// Setting ComVisible to false makes the types in this assembly not visible
// to COM components. If you need to access a type in this assembly from
// COM, set the ComVisible attribute to true on that type.
[assembly: ComVisible(false)]
// The following GUID is for the ID of the typelib if this project is exposed
// to COM
[assembly: Guid("ae0acc2c-0daf-4bb0-84a3-f9f6ac48bfe9")]
//
// Version information for an assembly consists of the following four
// values:
//
// Major Version
// Minor Version
// Build Number
// Revision
//
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]
```

这个文件用于配置程序集清单。编译器读取程序集的属性,把特定的信息插入到程序集清单中。

assembly:前缀把属性标记为程序集的全局属性。与其他属性相反,程序集的全局属性与特定的语言元素无关,用于程序集属性的参数是命名空间 System.Reflection、System.Runtime.CompilerServices 和 System.Runtime.InteropServices 中的类。

注意:

第 13 章介绍了属性和如何创建和使用定制属性的内容。

表 17-1 是 System.Reflection 命名空间中定义的程序集属性列表。



表 17-1

程序集的属性	说 明
AssemblyCompany	指定公司名
AssemblyConfiguration	指定建立信息，例如零售或调试信息
AssemblyCopyright 和 AssemblyTrademark	包含版权和商标信息
AssemblyDefaultAlias	如果程序集名不容易理解(例如动态创建程序集名称时的 GUID)，就可以使用该属性。使用这个属性可以指定一个别名
AssemblyDescription	描述程序集或产品。如果查看可执行文件的属性，这个值就会显示为 Comments
AssemblyProduct	指定了程序集所属的产品名称
AssemblyTitle	给程序集提供一个友好的名称。该名称可以包含空格。使用文件属性 时，这个值就显示为 Description
AssemblyCulture	定义程序集的文化。这个属性对辅助程序集很重要
AssemblyInformationalVersion	在引用程序集时，这个属性不用于版本检查，它仅用于版本信息。 该属性非常适合于指定使用多个程序集的应用程序的版本。打开可 执行程序属性，这个值就显示为 Product Version
AssemblyVersion	这个属性给出了程序集的版本号。本章后面讨论版本问题
AssemblyFileVersion	这个属性定义了文件的版本。这个值显示在 Windows 文件属性窗口 中，但对 .NET 操作没有影响

下面是配置这些属性的一个示例：

```
[assembly: AssemblyTitle("Professional C#")]
[assembly: AssemblyDescription("Sample Application")]
[assembly: AssemblyConfiguration("Retail version")]
[assembly: AssemblyCompany("Wrox Press")]
[assembly: AssemblyProduct("Wrox Professional Series")]
[assembly: AssemblyCopyright("Copyright (C) Wrox Press 2008")]
[assembly: AssemblyTrademark("Wrox is a registered trademark of " +
"John Wiley & Sons, Inc.")]
[assembly: AssemblyCulture("")]

[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]
```

在 Visual Studio 2008 中，可以用项目属性、应用程序设置和程序集信息来配置这些属性，如图 17-6 所示。

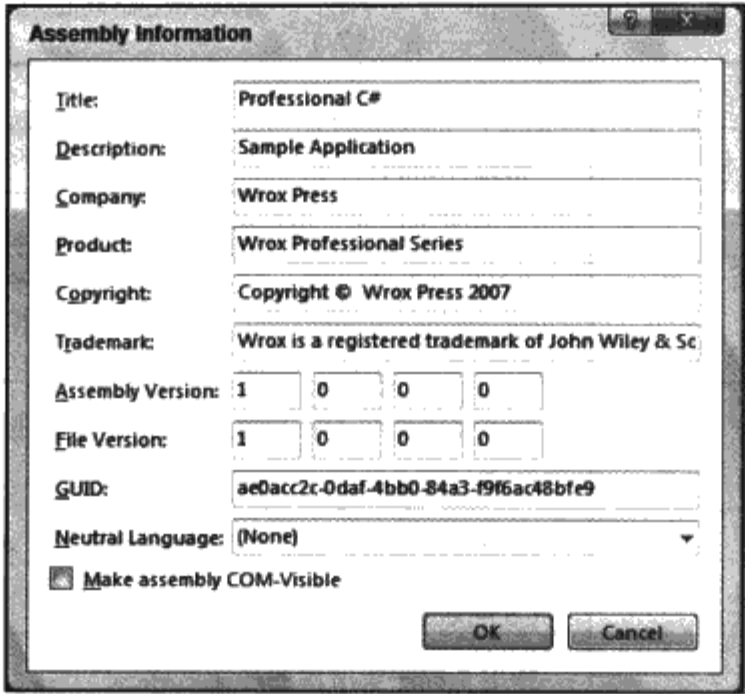


图 17-6

17.3 动态加载和创建程序集

在开发期间，添加对程序集的引用，使之包含在程序集引用中，该程序集中的类型就可用于编译器。在运行期间，只要实例化了程序集中的一个类型，或者使用了该类型的方法，就会加载所引用的程序集。除了这种自动操作之外，还可以编程加载程序集。为此，可以使用类 Assembly 的静态方法 Load()。这个方法是重载的，可以使用 AssemblyName 给它传送程序集的名称或字节数组。

还可以随时创建程序集，如下面的例子所示。这个例子演示了把 C#代码输入文本框后，启动 C#编译器，就会动态创建一个新程序集，并调用编译的代码。

要动态编译 C#代码，可以使用 Microsoft.CSharp 命名空间中的 CSharpCodeProvider 类。使用这个类可以编译代码，从 DOM 树、文件或源代码中生成程序集。

该应用程序的 UI 是使用 WPF 完成的，如图 17-7 所示。窗口由一个输入 C#代码的文本框、一个按钮和一个 TextBlock WPF 控件组成，TextBlock WPF 控件横跨最后一行的所有列，显示了结果。

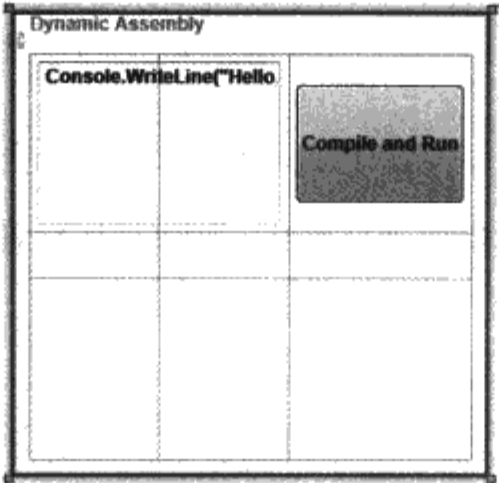


图 17-7

为了动态编译并运行 C# 代码, 类 `CodeProvider` 定义了方法 `CompileAndRun()`。这个方法编译文本框中的代码, 启动所生成的方法。

```
using System;
using System.CodeDom.Compiler;
using System.IO;
using System.Reflection;
using System.Text;
using Microsoft.CSharp;

namespace Wrox.ProCSharp.Assemblies
{
    public class CodeDriver
    {
        private string prefix =
            "using System;" +
            "public static class Driver" +
            "{ " +
            "    public static void Run()" +
            "    { ";

        private string postfix =
            "    }" +
            "}";

        public string CompileAndRun(string input, out bool hasError)
        {
            hasError = false;
            string returnData = null;

            CompilerResults results = null;
            using (CSharpCodeProvider provider = new CSharpCodeProvider())
            {
                CompilerParameters options = new CompilerParameters();
                options.GenerateInMemory = true;

                StringBuilder sb = new StringBuilder();
                sb.Append(prefix);
                sb.Append(input);
                sb.Append(postfix);

                results = provider.CompileAssemblyFromSource(
                    options, sb.ToString());
            }

            if (results.Errors.HasErrors)
            {
                hasError = true;
                StringBuilder errorMessage = new StringBuilder();
                foreach (CompilerError error in results.Errors)
                {
                    errorMessage.AppendFormat("{0} {1}", error.Line,
                        error.ErrorText);
                }
                returnData = errorMessage.ToString();
            }
            else
            {
                TextWriter temp = Console.Out;
                StringWriter writer = new StringWriter();
            }
        }
    }
}
```

```

        Console.SetOut(writer);
        Type driverType =
            results.CompiledAssembly.GetType("Driver");

        driverType.InvokeMember("Run", BindingFlags.InvokeMethod
            | BindingFlags.Static | BindingFlags.Public,
            null, null, null);
        Console.SetOut(temp);

        returnData = writer.ToString();
    }
    return returnData;
}
}
}

```

方法 `CompileAndRun()` 需要一个字符串参数 `input`，在其中可以传送一行或多行 C# 代码。所调用的每个方法都必须包含在方法和类中，所以变量 `prefix` 和 `postfix` 定义了动态创建的类 `Driver` 的结构和包含参数中代码的方法 `Run()`。使用 `StringBuilder`，把 `prefix`、`postfix` 和 `input` 变量中的代码合并起来，创建一个完整的、可编译的类。再使用这个得到的字符串，通过 `CSharpCodeProvider` 类编译代码。方法 `CompileAssemblyFromSource()` 动态创建一个程序集。因为这个程序集仅需要在内存中使用，所以设置了编译器参数选项 `GenerateInMemory`。

如果所传送的源代码包含错误，它们就会显示在 `CompilerResults` 的 `Errors` 集合中。错误和返回数据一起返回，变量 `hasError` 设置为 `true`。

如果源代码编译成功，就调用新类 `Driver` 的方法 `Run()`。这个方法的调用通过反射来实现。新编译的程序集可以使用 `CompilerResults.CompiledType` 来访问，在这个程序集中，新类 `Driver` 用变量 `driverType` 引用。接着使用 `Type` 类的 `InvokeMember()` 方法调用方法 `Run()`。这个方法定义为公共静态方法，所以必须设置 `BindingFlags`。要查看程序写到控制台上的结果，需要把控制台重定向到 `StringWriter` 上，最终使用 `returnData` 变量返回程序的全部结果。

#### 提示：

用 `InvokeMember()` 方法运行代码需要使用 .NET 反射功能，该功能详见第 13 章。

WPF 按钮的 `Click` 事件连接到 `Compile_Click()` 方法上，在该方法中，实例化了 `CodeDriver` 类，调用了方法 `CompileAndRun()`。从文本框 `textCode` 中提取输入，把结果写到 `TextBlock` 控件 `textOutput` 中。

```

private void Compile_Click(object sender, RoutedEventArgs e)
{
    CodeDriver driver = new CodeDriver ();
    bool isError;
    textOutput.Text = driver.CompileAndRun(textCode.Text, out isError);
    if (isError)
    {
        textOutput.Background = Brushes.Red;
    }
}

```

现在可以启动应用程序，在文本框中输入 C# 代码，如图 17-8 所示，编译运行代码。

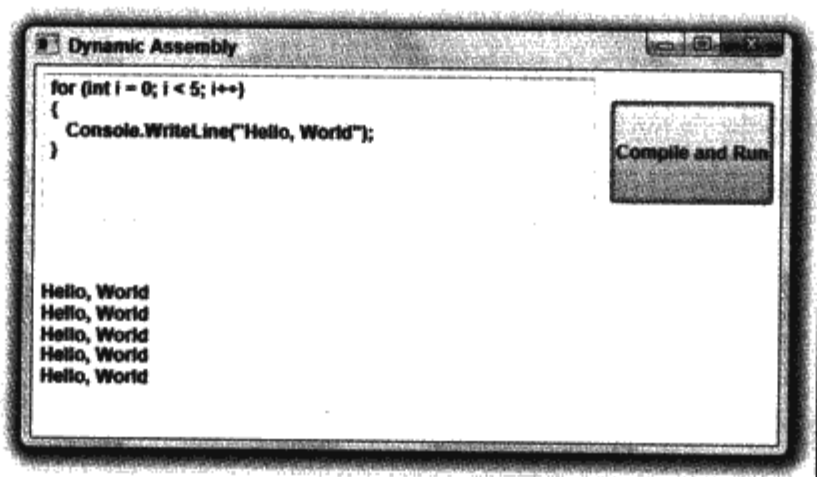


图 17-8

前面编写的程序有一个缺点：每次单击 **Compile And Run** 按钮时，都会创建并加载一个新程序集，所以程序需要越来越多的内存。不能从应用程序中卸载程序集。而要卸载程序集，需要使用应用程序域。

## 17.4 应用程序域

在.NET 之前的技术中，进程作为独立的边界来使用，每个进程都有其私有的虚拟内存，运行在一个进程中的应用程序不能写入另一个应用程序的内存，也不会以这种方式破坏其他应用程序。该进程用作应用程序之间的一个独立而安全的边界。在.NET 结构中，应用程序有一个新的边界：应用程序域。使用托管 IL 代码，运行库就不能访问同一个进程中另一个应用程序的内存。多个应用程序可以运行在一个进程的多个应用程序域中，如图 17-9 所示。

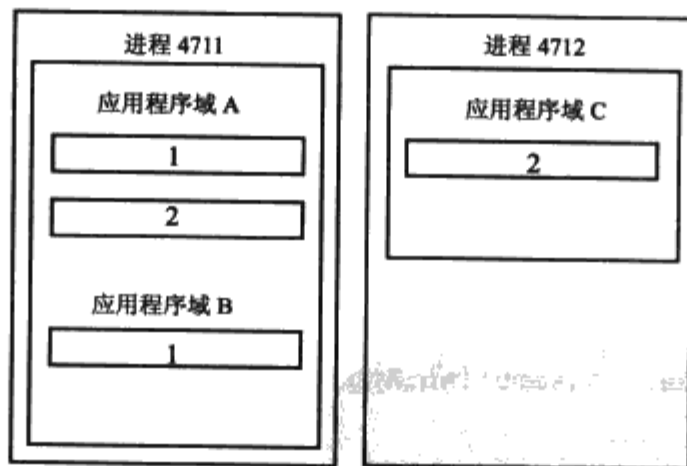


图 17-9

把程序集加载到应用程序域中。在图 17-9 中，进程 4711 有两个应用程序域。在应用程序域 A 中，实例化了对象 1 和 2，对象 1 在程序集 1 中，对象 2 在程序集 2 中。在进程 4711 中，第二个应用程序域有实例 1。要最小化内存的消耗，在应用程序域中，程序集的代码应只加载一次。实例和静态成员不能在应用程序域之间共享，也不能直接访问另一个应用程序域中的对象。此时需要一个代理(proxy)。所以在图 17-9 中，如果没有代理，应用程序域 B 中的对象 1 不能直接访问应用程序域 A 中的对象 1 或 2。



AppDomain 类用于创建和中断应用程序域，加载和卸载程序集和类型、枚举域中的程序集和线程。下面用一个小例子来说明应用程序域。

首先创建一个 C#控制台应用程序 AssemblyA，在 Main()方法中添加一个 Console.WriteLine()，这样，我们就知道这个方法什么时候调用。另外，添加一个类 Demo，其构造函数的参数是两个 int 值，用 AppDomain 类创建实例。在第二个应用程序中加载程序集 AssemblyA.exe:

```
using System;
namespace Wrox.ProCSharp.Assemblies.AppDomains
{
    public class Demo
    {
        public Demo(int val1, int val2)
        {
            Console.WriteLine("Constructor with the values {0}, {1}" +
                              " in domain {2} called", val1, val2,
                              AppDomain.CurrentDomain.FriendlyName);
        }
    }

    class Program
    {
        static void Main()
        {
            Console.WriteLine("Main in domain {0} called",
                              AppDomain.CurrentDomain.FriendlyName);
        }
    }
}
```

运行应用程序，结果如下所示。

```
Main in domain AssemblyA.exe called.
Press any key to continue ...
```

创建的第二个项目也是一个 C#控制台应用程序：DomainTest。首先，使用 AppDomain 类的 FriendlyName 属性显示当前域的名称。调用 CreateDomain()方法，创建一个新的应用程序域 New AppDomain，然后把程序集 AssemblyA 加载到新域中，通过调用 ExecuteAssembly()来调用 Main()方法：

```
using System;
namespace Wrox.ProCSharp.Assemblies.AppDomains
{
    class Program
    {
        static void Main()
        {
            AppDomain currentDomain = AppDomain.CurrentDomain;
            Console.WriteLine(currentDomain.FriendlyName);
            AppDomain secondDomain =
                AppDomain.CreateDomain("New AppDomain");
            secondDomain.ExecuteAssembly("AssemblyA.exe");
        }
    }
}
```

在启动程序 DomainTest.exe 前，要先给 DomainTest 项目引用程序集 AssemblyA.exe。通过

Visual Studio 2008 引用程序集，就是把程序集复制到项目的目录下，这样程序才能找到这个程序集。如果找不到这个程序集，就会抛出 `System.IO.FileNotFoundException` 异常。

运行 `DomainTest.exe` 程序后，会得到如下所示的控制台输出。`DomainTest.exe` 是第一个应用程序域的名称。第二行是 `New AppDomain` 中新加载的程序集的输出结果。在进程浏览器中看不到进程 `AssemblyA.exe` 的执行，因为没有创建新进程，`AssemblyA` 加载到 `DomainTest.exe` 进程中。

```
DomainTest.exe
Main in domain New AppDomain called
Press any key to continue . . .
```

在新加载的程序集中，还可以再创建一个新实例，以替代调用 `Main()` 方法。在下面的例子中，用 `CreateInstance()` 替代 `ExecuteAssembly()` 方法，它的第一个参数是程序集名 `AssemblyA`，第二个参数定义了应实例化的类 `Wrox.ProCSharp.Assemblies.AppDomains.Demo`，第三个参数 `true` 表示不区分大小写。`System.Reflection.BindingFlags.CreateInstance` 是一个绑定标志枚举值，指定应调用构造函数：

```
AppDomain secondDomain =
    AppDomain.CreateDomain("New AppDomain");
// secondDomain.ExecuteAssembly("AssemblyA.exe");
secondDomain.CreateInstance("AssemblyA",
    "Wrox.ProCSharp.Assemblies.AppDomains.Demo", true,
    System.Reflection.BindingFlags.CreateInstance,
    null, new object[] {7, 3}, null, null, null);
```

在应用程序成功运行后，会得到如下所示的控制台输出。

```
DomainTest.exe
Constructor with the values 7, 3 in domain New AppDomain called
Press any key to continue . . .
```

前面介绍了如何创建和调用应用程序域。在运行期间，主应用程序域会自动创建。`ASP.NET` 为每个运行在 Web 服务器上的 Web 应用程序创建一个应用程序域。`Internet Explorer` 创建运行托管控件的应用程序域。对于应用程序，如果要卸载一个程序集，创建应用程序域是非常有效的。卸载程序集只能通过中断应用程序域来进行。

#### 注意：

如果程序集是动态加载的，且需要在使用完后卸载程序集，应用程序域就是非常有用的。在主应用程序域中，不能删除已加载的程序集，但可以终止应用程序域，在该应用程序域中加载的所有程序集都会从内存中清除出去。

了解了应用程序域后，就可以修改前面创建的 WPF 程序了。新类 `CodeDriverInAppDomain` 使用 `AppDomain.CreateDomain` 创建了一个新应用程序域。在这个新应用程序域中，使用 `CreateInstanceAndUnwrap()` 实例化类 `CodeDriver`。使用 `CodeDriver` 实例，调用 `CompileAndRun()` 方法，之后再次卸载新应用程序域。

```
using System;
using System.Runtime.Remoting;

namespace Wrox.ProCSharp.Assemblies
{
```

```

public class CodeDriverInAppDomain
{
    public string CompileAndRun(string code, out bool hasError)
    {
        AppDomain codeDomain = AppDomain.CreateDomain("CodeDriver");

        CodeDriver codeDriver = (CodeDriver)
            codeDomain.CreateInstanceAndUnwrap("DynamicCompileWPF",
                "Wrox.ProCSharp.Assemblies.CodeDriver");

        string result = codeDriver.CompileAndRun(code, out hasError);

        AppDomain.Unload(codeDomain);

        return result;
    }
}

```

**提示:**

类 `CodeDriver` 本身现在在主应用程序域和新应用程序域中使用, 因此不能删除这个类使用的代码。如果要删除这些代码, 可以定义一个由 `CodeDriver` 实现的接口, 再在主应用程序域中使用这个接口。但是在这个例子中, 这并不是个问题, 因为只需删除用 `Driver` 类动态创建的程序集即可。

要在另一个应用程序域中访问类 `CodeDriver`, 类 `CodeDriver` 就必须派生于基类 `MarshalByRefObject`。只有派生于这个基类的类才能通过另一个应用程序域来访问。在主应用程序域中, 实例化一个代理, 通过应用程序域之间的信道调用这个类的方法。

```

using System;
using System.CodeDom.Compiler;
using System.IO;
using System.Reflection;
using System.Text;
using Microsoft.CSharp;

namespace Wrox.ProCSharp.Assemblies
{
    public class CodeDriver : MarshalByRefObject
    {

```

`Compile_Click()` 事件处理程序现在可以修改为使用新类 `CodeDriverInAppDomain`, 来替代 `CodeDriver` 类:

```

private void Compile_Click(object sender, RoutedEventArgs e)
{
    CodeDriverInAppDomain driver = new CodeDriverInAppDomain();
    bool isError;
    textOutput.Text = driver.CompileAndRun(textCode.Text, out isError);
    if (isError)
    {
        textOutput.Background = Brushes.Red;
    }
}

```

单击应用程序的 `Compile And Run` 按钮任意多次, 生成的程序集将总是会卸载。

提示:

使用 AppDomain 类的 GetAssemblies() 方法, 就可以查看应用程序域中加载的程序集。

## 17.5 共享程序集

程序集可以由一个应用程序使用, 在默认情况下不共享程序集。在使用共享程序集时, 需要考虑一些特定的要求。

本节将介绍:

- 共享程序集必须有的强名
- 全局程序集高速缓存
- 创建共享程序集
- 在全局程序集缓存中安装共享程序集
- 共享程序集的延迟签名

### 17.5.1 强名

共享程序集名的要求是它必须是全局唯一的, 必须可以保护该名称。其他人不能使用这个名称创建程序集。

COM 使用全局唯一标识符(GUID)只解决了第一个问题。但第二个问题仍没有解决, 每个人都可以盗用这个 GUID, 用相同的标识符创建不同的对象。这两个问题使用 .NET 程序集的强名都可以解决。

强名由下述项组成:

- 程序集本身的名称
- 版本号。有了版本号, 可以同时使用同一个程序集的不同版本。不同的版本可以同时存在, 并可以同时加载到同一个进程上。
- 公钥保证强名是独一无二的。它也保证引用的程序集不能被另一个源替代。
- 文化。详见第 21 章。

注意:

共享程序集必须有一个强名, 来唯一地标识该程序集。

强名是一个简单的文本名称, 附带版本号、公钥和文化。每个程序集不能有新的公钥, 但可以在公司中有一个这样的公钥, 这样该密钥就唯一地标识了公司的程序集。

但是, 这个密钥不能用作信任密钥。程序集可以利用 Authenticode 签名来建立信任关系。Authenticode 签名的密钥可以与强名中使用的密钥不同。

注意:

从开发的角度来看, 可以使用不同的公钥, 以后也可以与真正的密钥互换。该特性将在“程序集的延迟签名”中介绍。

为了唯一地标识公司中的程序集，应使用命名空间层次结构来给类命名。下面是一个组织命名空间的例子：Wrox Press 使用主命名空间 Wrox 来标识其类和命名空间。在命名空间 Wrox 下面的层次结构中，必须对命名空间进行组织，使所有的类都是唯一的。本书中的每一章都使用 Wrox.ProCSharp.<Chapter>形式的命名空间。本章使用 Wrox.ProCSharp.Assemblies 命名空间。这样，如果在某两章中都有类 Hello，也不会有名称冲突，因为它们在不同的命名空间中。可以在不同的书中使用的工具类则放在命名空间 Wrox.Utilities 中。

公司名一般用作命名空间的第一部分，但它不一定是唯一的，因此必须使用某种机制来建立强名。此时可以使用公钥。因为在强名中使用公钥/私钥规则，所以不能访问私钥的人，就不能破坏性地创建一个程序集，让客户代码无意中调用该程序集。

17.5.2 使用强名获得完整性

在创建共享组件时，必须使用公钥/私钥对。编译器把公钥写入程序集清单，创建属于该程序集的所有文件的散列表，用私钥标记这个散列表。私钥不存储在程序集中。这样就可以确保没有人可以修改该程序集。签名可以使用公钥来验证。

在开发过程中，客户程序集必须引用共享程序集。编译器把引用程序集的公钥写入客户程序集的清单中。要减少存储量，就不应把公钥写入客户程序集的清单，而应写入公钥标记。公钥标记是公钥散列表中的最后 8 位字节，且是唯一的。

在运行期间加载共享程序集时(如果客户程序集是使用本机图像生成器安装的，则应在安装期间加载)，共享程序集的散列表可以使用存储在客户程序集中的公钥来验证。除了私钥的主人外，其他人都不能修改共享程序集。销售商 A 创建了一个组件 Math，在客户机上引用该组件，黑客的组件就无法替代它。只有私钥的主人才能用新版本替换原来的共享组件。因此保证了其完整性，共享程序集会来自期望的发布者。

图 17-10 显示了一个共享组件，它的公钥由客户机程序集引用，该程序集在其清单中包含共享程序集的公钥标记。

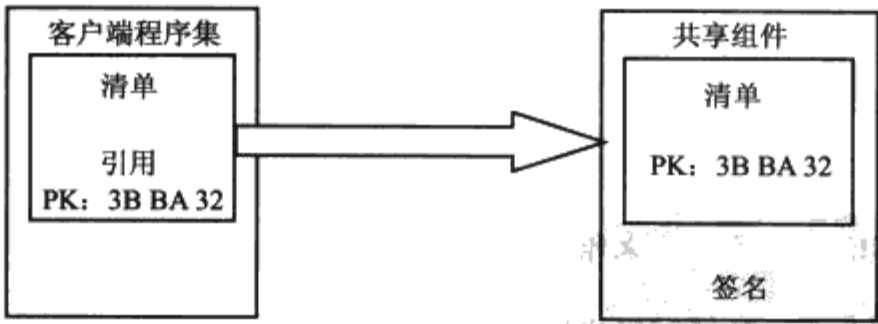


图 17-10

17.5.3 全局程序集缓存

顾名思义，全局程序集缓存(Global Assembly Cache)就是可全局使用的程序集的缓存。大多数共享程序集都安装在这个缓存中，也可以使用共享目录（也在服务器上）。

全局程序集缓存可以使用 shfusion.dll 来显示，shfusion.dll 是一个 Windows shell 扩展程序，可以查看和处理缓存的内容。Windows shell 扩展程序是一个与 Windows 资源管理器集成的



COM DLL。用户只需启动资源管理器，进入<windir>/assembly 目录即可。

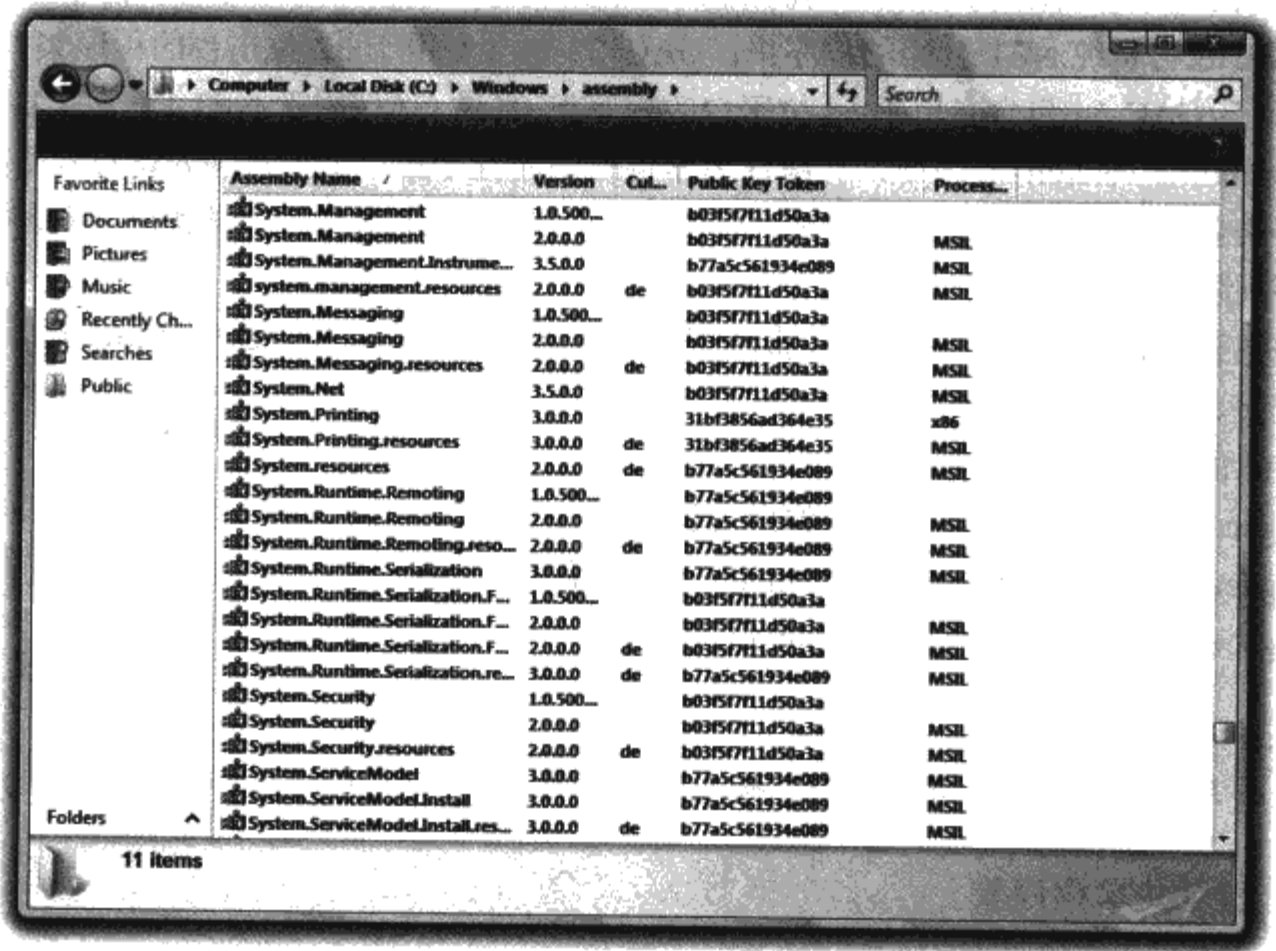


图 17-11

如图 17-11 所示，使用程序集缓存查看器，可以查看全局程序集的名称、类型、版本、文化和公钥标记。查看全局程序集的类型，可以确定程序集是否使用本机图像生成器安装的。选择一个程序集，使用其弹出菜单可以删除它，或查看其属性，如图 17-12 所示。

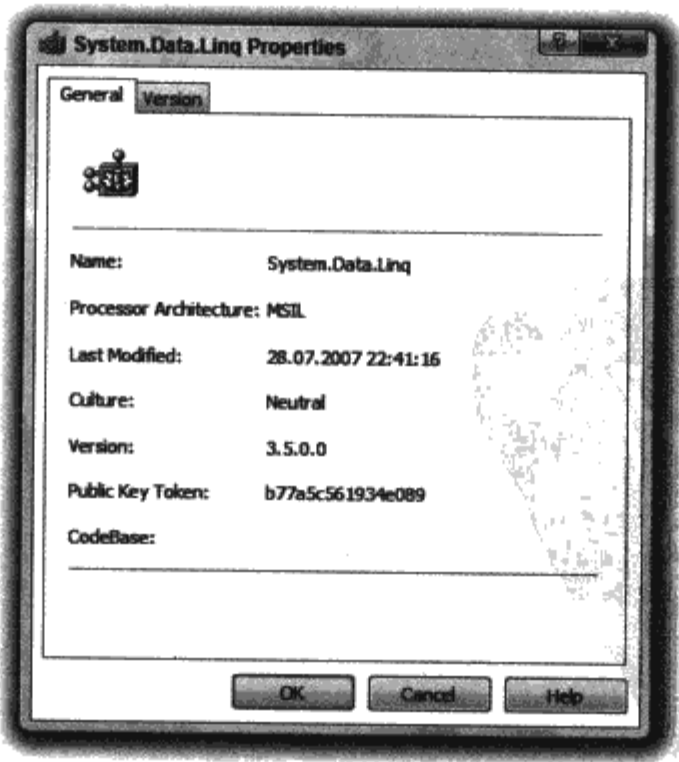


图 17-12

程序集缓存的文件和目录可以在命令行上通过浏览目录来查看。在<windir>\assembly 目录下, 有 GACxxx 和 NativeImages\_<runtime version>目录。GACxxx 是共享程序集的目录, GAC\_MSIL 包含带纯.NET 代码的程序集; GAC\_32 包含专用于 32 位平台的程序集。在 64 位系统上, 目录 GAC-64 包含专用于 64 位平台的程序集。目录 GAC 用于 .NET 1.0 和 1.1。在 Native Images\_<runtime version>目录下, 有编译为本机代码的程序集。如果再深入该目录结构, 会看到与程序集名类似的一些目录, 其下是一个版本目录和程序集本身。这样就可以安装同一个程序集的不同版本。

程序集查看器可以使用 Windows 资源管理器查看和删除程序集。gacutil.exe 工具可以使用命令行安装、卸载和显示程序集。

gacutil 的一些选项如下所示:

- gacutil /l 显示程序集缓存中的所有程序集。
- gacutil /i mydll 把共享程序集 mydll 安装到程序集缓存上。
- gacutil /u mydll 卸载程序集 mydll。

提示:

在产品系统中, 应使用安装程序, 把共享程序集安装在 GAC 中。部署参见第 16 章。

#### 17.5.4 创建共享程序集

在本例中, 创建一个共享程序集, 再创建一个使用该共享程序集的客户程序。

创建共享程序集与创建私有程序集的区别不大。首先建立一个简单的 Visual C# Class Library 项目 SharedDemo。把命名空间改为 Wrox.ProCSharp.Assemblies.Sharing, 类名改为 SharedDemo。输入下面的代码。类的构造函数把文件的所有行都将读取到一个集合中。文件名作为参数传送给构造函数。方法 GetQuoteOfTheDay()只返回这个集合的一个随机字符串。

```
using System;
using System.Collections.Generic;
using System.IO;

namespace Wrox.ProCSharp.Assemblies.Sharing
{
    public class SharedDemo
    {
        private List<string> quotes;
        private Random random;

        public SharedDemo(string filename)
        {
            quotes = new List<string> ();
            Stream stream = File.OpenRead(filename);
            StreamReader streamReader = new StreamReader(stream);
            string quote;
            while ((quote = streamReader.ReadLine()) != null)
            {
                quotes.Add(quote);
            }
            streamReader.Close();
            stream.Close();
            random = new Random();
        }
    }
}
```

```

    }
    public string GetQuoteOfTheDay()
    {
        int index = random.Next(1, quotes.Count);
        return quotes[index];
    }
}
}

```

### 17.5.5 创建强名

要共享这个程序集，需要一个强名。要创建这个名称，可以使用强名工具(sn)：

```
sn -k mykey.snk
```

强名工具生成和编写一个公钥/私钥对，并把该密钥对写到文件中，此处的文件是 mykey.snk。

在 Visual Studio 2008 中，可以选择 Signing 选项卡，用项目属性标记程序集，如图 17-22 所示。还可以使用这个工具创建密钥。但不需要为每个项目创建密钥文件。整个公司可以只使用几个密钥。最好根据安全要求创建不同的密钥，详见第 20 章。

用 Visual Studio 设置 signing 选项，会给编译器设置添加/keyfile 选项。Visual Studio 还允许创建用密码保护的密钥文件。这种文件的扩展名是.pfx，如图 17-13 所示。

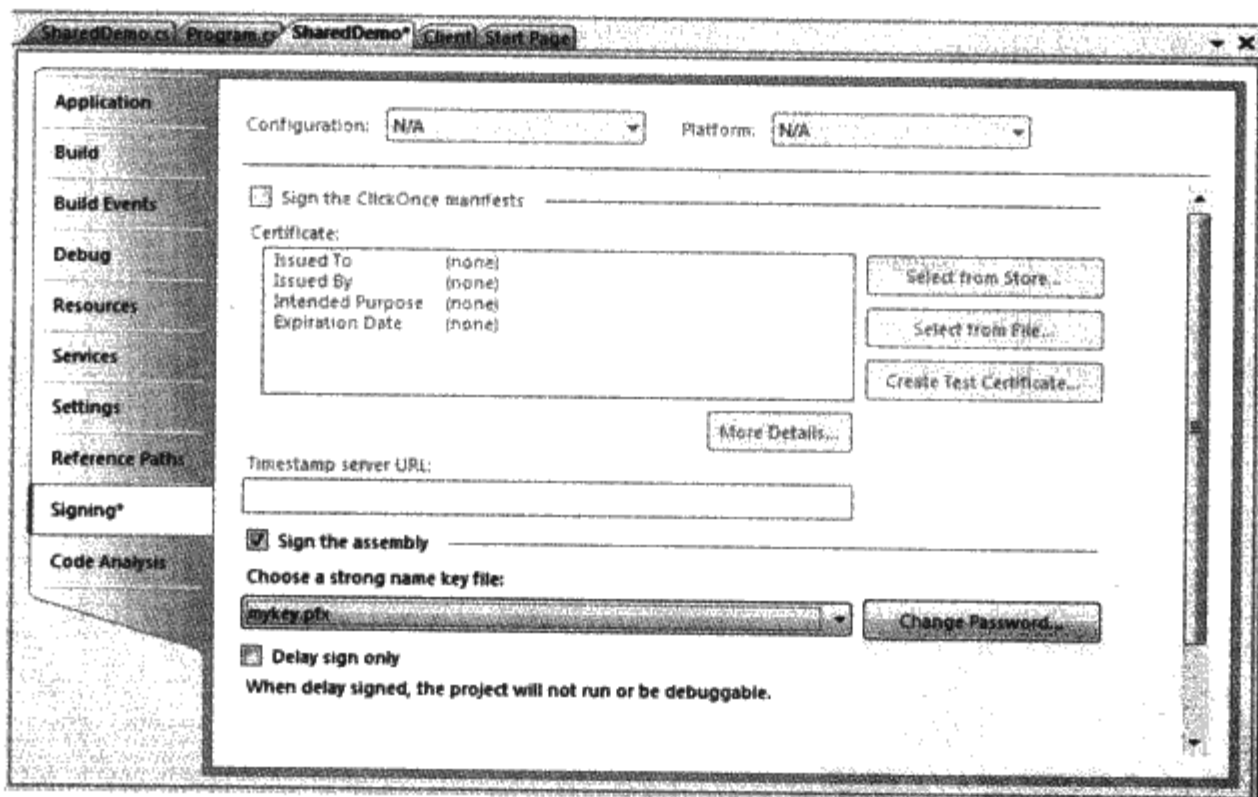


图 17-13

在重新建立该文件后，公钥就在程序集的清单中。可以使用 ildasm 验证这一点，如图 17-14 所示。

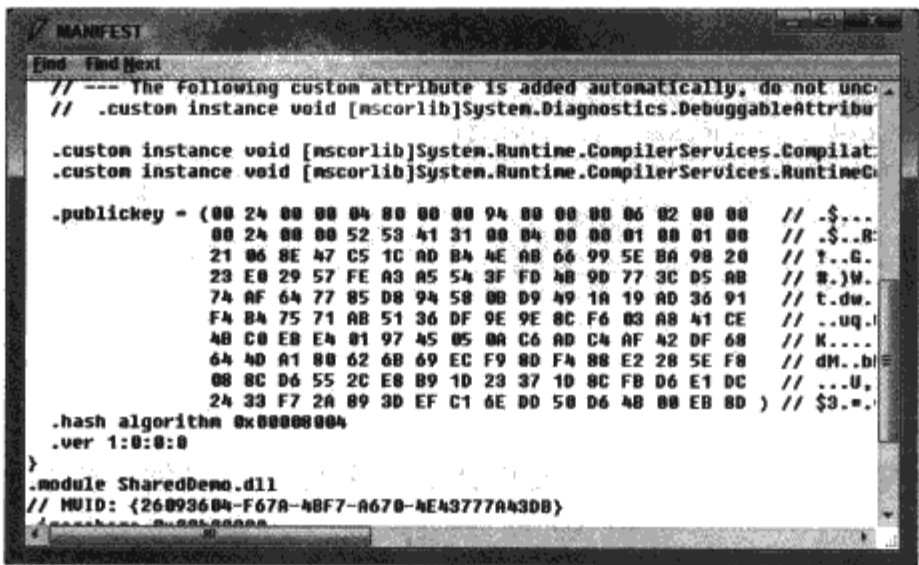


图 17-14

17.5.6 安装共享程序集

程序集中有了公钥后，就可以使用全局程序集缓存工具 gacutil 及其/i 选项把它安装到全局程序集缓存中：

```
gacutil /i SharedDemo.dll
```

用 Visual Studio 配置以后建立的事件命令行，如图 17-15 所示，就可以在全局程序集缓存中安装每个成功建立的程序集。

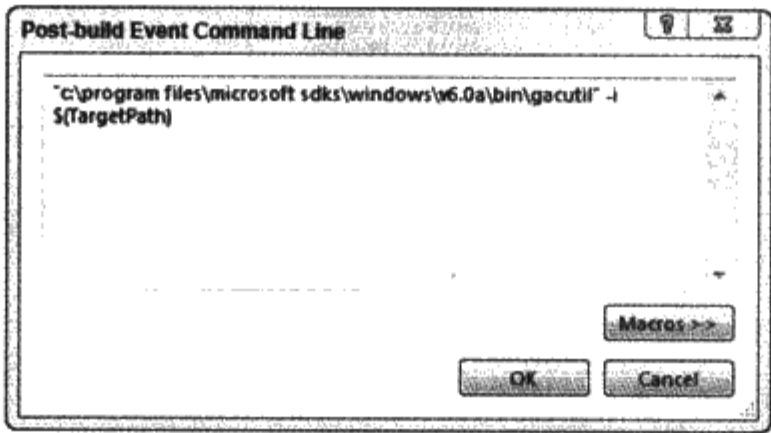


图 17-15

提示：

如果使用 Windows Vista 把程序集从 Visual Studio 安装到 GAC 中，就必须用相应的权限启动 Visual Studio。把程序集安装到 GAC 中需要管理员权限。

然后，可以使用全局程序集缓存查看器检查共享程序集的版本，看看它是否安装成功。

17.5.7 使用共享程序集

要使用共享程序集，应创建一个 C#控制台应用程序 Client。把命名空间的名称改为

Wrox.ProCSharp.Assemblies.Sharing, 以引用私有程序集方式引用共享程序集: 使用菜单 Project | Add Reference。

注意:

有了共享程序集, 引用属性 Copy Local 就可以设置为 false, 这样, 共享程序集就不会复制到输出文件的目录下, 而会从全局程序集缓存中加载。

下面是 Client 应用程序的代码:

```
using System;
namespace Wrox.ProCSharp.Assemblies.Sharing
{
    class Program
    {
        static void Main()
        {
            SharedDemo quotes = new SharedDemo(
                @"C:\ProCSharp\Assemblies\Quotes.txt");
            for (int i=0; i < 3; i++)
            {
                Console.WriteLine(quotes.GetQuoteOfTheDay());
                Console.WriteLine();
            }
        }
    }
}
```

在使用 ildasm 查看客户程序集的清单时, 可以看到对共享程序集 Shared Demo 的引用: .assembly extern SharedDemo。这个引用信息的一部分是版本号(详见后面的内容)和公钥的标记, 如图 17-16 所示。

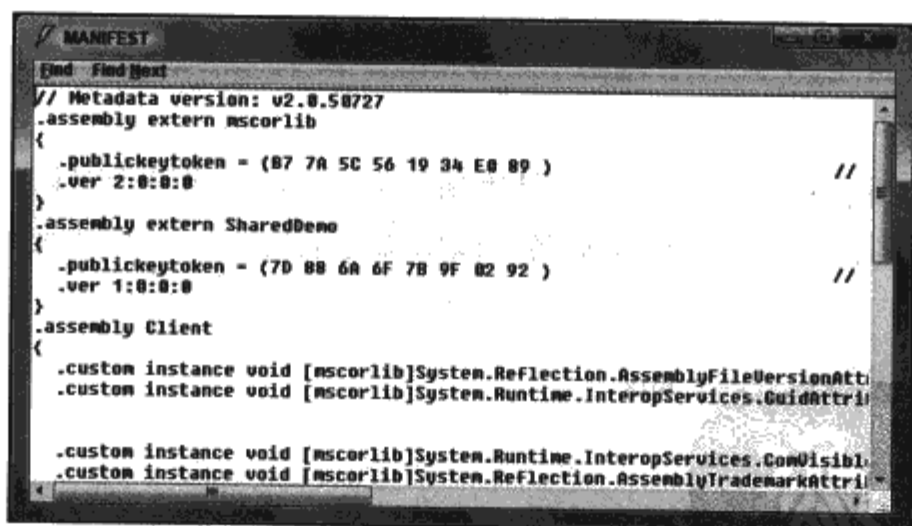


图 17-16

公钥的标记也可以使用强名工具 sn 在共享程序集中查看: sn -T 会显示程序集中的公钥标记, sn -Tp 显示标记和公钥。注意使用大写字母 T!

该程序引用了示例文件, 其结果如下所示。

"We don't like their sound. And guitar music is on the way out." - Decca Recording, Co., in rejecting the Beatles, 1962

"The ordinary 'horseless carriage' is at present a luxury for the wealthy; and



although its price will probably fall in the future, it will never come into as common use as the bicycle. " -- The Literary Digest, 1889

"Landing and moving around the moon offer so many serious problems for human beings that it may take science another 200 years to lick them", Lord Kelvin (1824-1907)

Press any key to continue ...

### 17.5.8 程序集的延迟签名

公司的私钥应安全存储。大多数公司都不允许所有的开发人员都能访问私钥。只有几个有安全权限的人才能访问它。这就是为什么程序集的签名可以以后(例如在发布之前)添加的原因。在全局程序集属性 `AssemblyDelaySign` 设置为 `true` 时, 签名就不会存储在程序集中, 但保留了足够的空间, 以便以后添加。但是, 不使用密钥, 就不能测试程序集, 在全局程序集缓存中安装它。可以使用临时密钥进行测试, 以后再用真正的公司密钥代替这个临时密钥。

程序集的延迟签名需要执行下面的步骤:

- (1) 首先必须用强名工具 `sn` 创建一个公钥/私钥对, 生成的文件 `mykey.snk` 包含公钥和私钥。

```
sn -k mykey.snk
```

- (2) 接着提取公钥, 使之可以用于开发人员。选项 `-p` 提取密钥文件的公钥。文件 `mypublickey.snk` 仅包含公钥。

```
sn -p mykey.snk mypublickey.snk
```

公司中的所有开发人员都可以使用这个密钥文件 `mykeyPub.snk`, 用 `/delaysign`+选项编译程序集。这样, 签名就没有添加到程序集中, 但可以以后添加。在 Visual Studio 2008 中, 延迟签名可以在 `Signing` 设置的复选框中设置。

- (3) 关闭签名的验证功能, 因为程序集没有包含签名。

```
sn -Vr ShareDemo.dll
```

- (4) 在发布之前, 程序集可以用 `sn` 工具重新签名。`-R` 选项用于对以前已签名或延迟签名的程序集进行重新签名。程序集的重新签名可以由部署应用程序、且拥有用于发布的私钥的人员完成。

```
sn -R MyAssembly.dll mykey.snk
```

注意:

签名的验证功能只能在开发过程中关闭。不经过验证是不能发布程序集的, 因为这个程序集可能被怀有恶意的程序集替代。

提示:

程序集的重新签名可以通过在 MSBuild 文件中定义任务来自动完成, 参见第 15 章。

### 17.5.9 引用

属性对话框还列出了引用总数。这个引用总数表示: 如果应用程序仍需要引用程序集, 被

缓存的程序集就不能删除。例如,如果 Microsoft 安装程序包(MSI 文件)安装了一个共享程序集,就只能在卸载应用程序时删除它,而不能从全局程序集缓存中删除它。从全局程序集缓存中删除程序集会得到一个错误消息:“程序集<name>不能卸载,因为其他应用程序还需要它。”

使用 `gacutil` 工具的选项 `/r` 可以设置程序集的引用。该选项需要一个引用类型、一个引用 ID 和描述。引用的类型可以是下面 3 个选项中的一个: `UNINSTALL_KEY`、`FILEPATH` 和 `OPAQUE`。`UNINSTALL_KEY` 由 MSI 使用,定义一个卸载时也需要注册键。`FILEPATH` 可以指定一个目录,应用程序的根目录是有效的。`OPAQUE` 引用类型允许设置任意类型的引用。

命令行:

```
gacutil /i shareddemo.dll /r FILEPATH c:\ProCSharp\Assemblies\Client "Shared Demo"
```

通过对客户应用程序目录的引用,在全局程序集缓存中安装程序集 `SharedDemo`。这个程序集的另一个版本可以使用另一个目录安装,或使用 `OPAQUE` ID 安装,如下面的命令行:

```
gacutil /i shareddemo.dll OPAQUE 4711 "Opaque installation"
```

现在全局程序集缓存中只有一个程序集,但有两个引用。为了从全局程序集缓存中删除程序集,必须删除这两个引用:

```
gacutil /u shareddemo OPAQUE 4711 "Opaque installation"
gacutil /u shareddemo FILEPATH c:\ProCSharp\Assemblies\Client "Shared Demo"
```

提示:

要删除共享程序集,选项 `/u` 需要不带文件扩展名 `DLL` 的程序集。而安装程序集的选项 `/i` 需要包含文件扩展名的文件程序集名。

注意:

第 15 章介绍了程序集的部署,在 MSI 包中要处理引用总数。

### 17.5.10 本机图像生成器

使用内部图像生成器(native image generator)`Ngen.exe`,可以在安装期间把 IL 代码编译为本机代码。这样程序启动就比较快,因为不再需要在运行时进行编译。比较预编译的程序集和需要运行 JIT 编译器的程序集,其性能在编译 IL 代码后没有太大区别。使用内部图像生成器获得的唯一改进是应用程序启动比较快,因为不需要运行 JIT 了。减少应用程序的启动时间是使用内部图像生成器的主要原因。如果从可执行文件中创建本机图像,也应从可执行文件加载的所有 DLL 中创建本机图像,否则就仍需要运行 JIT 编译器。

`Ngen` 工具在本机图像缓存中安装本机图像,本机图像缓存的物理目录是 `<windows>\assembly\NativeImages <Runtime- Version>`。

使用 `ngen install myassembly` 可以把 MSIL 代码编译为本机代码,并把它安装到本机图像缓存上。如果要把程序集安装到本机图像缓存中,就应在安装程序中完成。

使用 `ngen` 的选项 `display` 还可以显示本机图像缓存中的所有程序集。如果把程序集的名称添加到选项 `display` 上,就可以得到这个程序集所有已安装版本的信息,以及依赖于本机程序集的程序集,如下所示。

```
C:\> ngen display System.Windows.Forms
Microsoft (R) CLR Native Image Generator - Version 2.0.50727.3178
Copyright (C) Microsoft Corporation. All rights reserved.
```

NGEN Roots:

```
System.Windows.Forms, Version=2.0.0.0, Culture=Neutral,
PublicKeyToken=b77a5c561934e089, processorArchitecture=msil
```

NGEN Roots that depend on "System.Windows.Forms":

```
ComSvcConfig, Version=3.0.0.0, Culture=Neutral,
  PublicKeyToken=b03f5f7f11d50a3a, processorArchitecture=msil
ehpg, Version=6.0.6000.0, Culture=Neutral,
  PublicKeyToken=31bf3856ad364e35, processorArchitecture=msil
ehpgdat, Version=6.0.6000.0, Culture=Neutral,
  PublicKeyToken=31bf3856ad364e35, processorArchitecture=msil
ehExtCOM, Version=6.0.6000.0, Culture=Neutral,
  PublicKeyToken=31bf3856ad364e35, processorArchitecture=msil
ehexthost, Version=6.0.6000.0, Culture=Neutral,
  PublicKeyToken=31bf3856ad364e35, processorArchitecture=msil
ehRecObj, Version=6.0.6000.0, Culture=Neutral,
  PublicKeyToken=31bf3856ad364e35, processorArchitecture=msil
ehshell, Version=6.0.6000.0, Culture=Neutral,
  PublicKeyToken=31bf3856ad364e35, processorArchitecture=msil
EventViewer, Version=6.0.0.0, Culture=Neutral,
  PublicKeyToken=31bf3856ad364e35, processorArchitecture=msil
```

如果系统的安全设置有变化,就不能保证本机图像达到运行应用程序的安全要求。这就是本机图像在系统配置变化时就无效的原因。使用命令 `ngen update`, 会重建所有的本机图像, 以包含新配置。

安装.NET 2.0 运行库时, 会安装本机图像服务 (或 Window Service CLR Optimization Service) Microsoft .NET Framework NGEN v2.0.50727\_X86。这个服务可以用于延迟本机图像的编译, 重新生成已失效的本机图像。

安装程序可以使用 `ngen install myassembly /queue` 命令, 延迟通过本机图像服务将 `myassembly` 编译为本机图像的过程。`ngen update /queue` 重新生成已失效的本机图像。使用 `ngen-queue` 选项 `pause`、`continue` 和 `status`, 可以控制服务, 获取状态信息。

#### 提示:

为什么不能在开发系统中创建本机图像, 而只能在产品系统中描述本机图像? 因为本机图像生成器会处理与目标系统一起安装的 CPU, 编译为 CPU 类型优化的代码。而在安装应用程序的过程中, CPU 才是已知的。

## 17.6 配置.NET 应用程序

COM 组件使用注册表配置组件。.NET 应用程序的配置是使用配置文件完成的。使用注册表配置, 就不可能使用 `xcopy` 部署了。配置文件只是进行简单的复制。配置文件使用 XML 语法来指定应用程序的启动和运行库配置。

本节将介绍:

- 可以使用 XML 基本配置文件进行哪些配置

- 用强名引用的程序集如何重新定向到另一个版本上
- 如何指定程序集的目录，以便在子目录中查找私有程序集，在公共目录或服务器上查找共享程序集

### 17.6.1 配置类别

可以把配置分为如下几类：

- 启动设置，用于指定需要的运行库版本。同一个系统上可能安装了不同版本的运行库，使用<startup>元素来指定运行库版本。
- 运行库设置，用于指定运行库如何进行垃圾收集，如何进行程序集绑定。也可以使用这些设置指定版本策略和编码基(code base)。本章的后面将详细介绍运行库配置。
- WCF 设置用于利用 WCF 配置应用程序。参见第 42 章。
- 安全设置，详见第 20 章。第 20 章介绍了加密配置和许可。

这些设置可以在三种配置文件中给出：

- 应用程序配置文件包含应用程序的特定设置，例如程序集的绑定信息，远程对象的配置等。这个配置文件放在可执行文件所在的目录下，它与可执行文件同名，但最后添加了.config 扩展名。ASP.NET 配置文件名为 web.config。
- 机器配置文件可以用于全系统配置。也可以在这里指定程序集绑定和远程配置。在绑定过程中，应在考虑应用程序配置文件之前考虑机器配置文件。应用程序配置可以重写机器配置中的设置。应用程序配置文件应优先用于应用程序特定的设置，这样机器配置文件会比较小，也容易管理。机器配置文件位于%runtime\_install\_path%\config\Machine.config 中。
- 发布方的策略文件(Publisher policy files)由组件的创建者用于指定共享程序集可以与旧版本兼容。如果新程序集版本仅修改了共享程序集的一个错误，就不必把应用程序配置文件放在每个使用该组件的应用程序目录中，发布者可以添加一个发布方的策略文件，把它标记为“可兼容的”。当组件不能用于所有的应用程序时，就可以在应用程序配置文件中重写发布方的策略设置。与其他配置文件不同，发布方的策略文件存储在全局程序集缓存中。

该如何使用这些配置文件？客户程序如何根据程序集是共享还是私有的来查找程序集(也称为绑定)？私有程序集必须位于应用程序所在的目录或子目录下。进程 probing 可用于查找这样的程序集。如果程序集没有强名，probing 就不使用版本号。

共享程序集可以安装在全局程序集缓存中，或放在一个目录、网络共享或 Web 站点上。我们用 codeBase 配置来指定这样一个目录(详见后面的内容)。在绑定公共程序集时，公钥、版本和文化都很重要。所需程序集的引用记录在客户程序集的清单中，包括名称、版本和公钥标记。所有的配置文件都要检查，以应用正确的版本策略。全局程序集缓存和配置文件中指定的 codeBase 也要检查，然后检查应用程序的目录，之后应用 probing 规则。

17.6.2 为搜索程序集配置目录

前面已经介绍了如何把共享程序集安装到全局程序集缓存中。如果不把共享程序集安装到全局程序集缓存中，还可以使用配置文件配置特定的共享目录。如果要在服务器上使用共享组件，就可以使用这个功能。如果要在应用程序之间共享一个程序集，但不希望它在全局程序集缓存中共享，就可以把该程序集放在一个共享目录中。

查找程序集的正确目录有两种方式：使用 XML 配置文件中的 codeBase 元素，或者使用 probing。codeBase 配置只用于共享程序集，而 probing 可用于私有和共享程序集。

1. <codeBase>

也可以使用.NET 配置工具来配置<codeBase>。在 Applications 树的 Configured Assemblies 中，选择已配置应用程序 SharedDemo 的属性来配置 codeBase。与 Binding Policy 一样，可以使用 Codebases 选项卡来配置版本列表。在图 17-17 中，显示了从 Web 服务器 <http://www.christiannagel.com/WroxUtils> 上加载的版本 1.1。

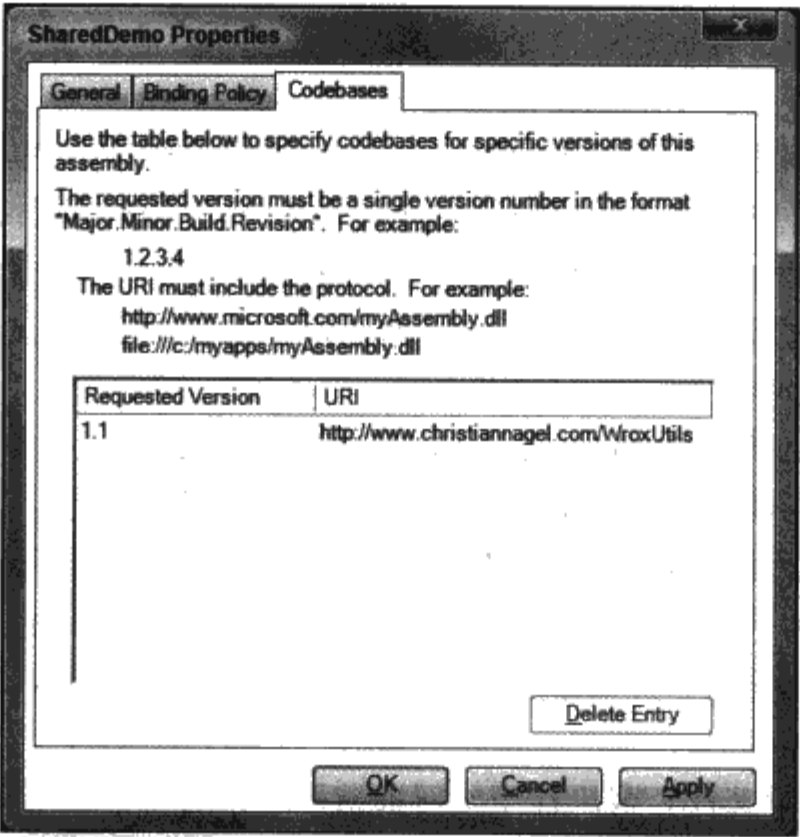


图 17-17

使用.NET 配置工具创建这个应用程序配置文件：

```
<?xml version="1.0"?>
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly xmlns="">
        <assemblyIdentity name="SimpleShared"
          publicKeyToken="7d886a6f7b9f0292" />
        <codeBase version="1.1" href="http://www.christiannagel.com/WroxUtils" />
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```



```
</runtime>
</configuration>
```

`<codeBase>`元素有属性 `version` 和 `href`，使用 `version` 属性必须指定程序集的原引用版本，使用 `href` 属性可以定义加载程序集的目录。在本例中，使用 HTTP 协议所在的路径。使用 `href="file:C:/WroxUtils"` 可以指定本地系统上的一个目录或一个共享。

注意：

在使用从网络上加载的程序集时，会抛出 `System.Security.Permissions` 异常。必须为从网上加载的程序集配置所需的许可。第 20 章介绍了如何为程序集配置安全许可。

## 2. <probing>

如果没有配置 `<codeBase>`，程序集也没有存储在全局程序集缓存中，运行库就会利用 `probing` 来查找程序集。.NET 运行库会在应用程序目录中查找文件扩展名为 `.dll` 或 `.exe` 的程序集，或在其子目录中搜索同名的程序集。如果没有找到程序集，会继续搜索。可以在应用程序配置文件的 `<runtime>` 部分中，用 `<probing>` 元素配置搜索目录。使用 .NET Framework 配置工具选择应用程序的属性，也可以很容易地完成这个 XML 配置。使用 .NET Framework 配置工具中的搜索路径可以配置 `probing` 所在的目录，如图 17-18 所示。

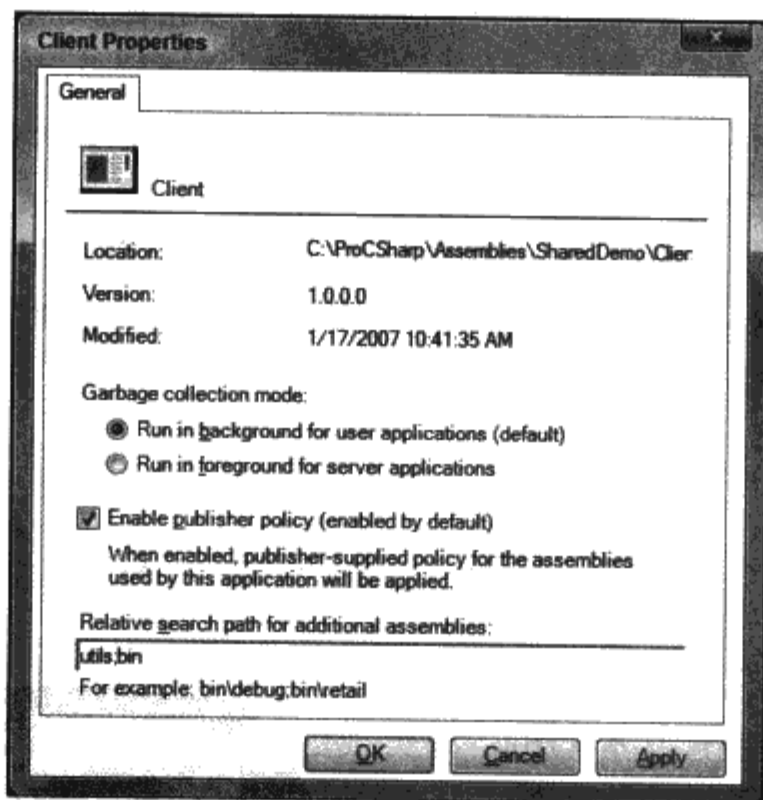


图 17-18

得到的 XML 文件如下所示。

```
<?xml version="1.0"?>
<configuration>
  <runtime>
    <gcConcurrent enabled="true" />
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <probing privatePath="bin;utils;" xmlns="" />
    </assemblyBinding>
  </runtime>
```

```
</configuration>
```

<probing>元素只有一个必需的属性 `privatePath`。这个应用程序配置文件告诉运行库，应在应用程序的根目录下搜索程序集，再在 `bin` 和 `util` 目录中搜索。这两个目录是应用程序根目录的子目录。不可能在应用程序根目录或其子目录的外部引用私有程序集。在应用程序根目录外部的程序集必须有一个共享名，并可以使用元素 `<codeBase>` 来引用。

## 17.7 版本问题

对于私有程序集来说，版本问题并不重要，因为引用的程序集会与客户程序一起复制。客户程序使用其私有目录下的程序集。

但是，共享程序集就不是这样。下面先看看共享时发生的一些传统问题。使用共享组件，则表示多个客户应用程序可以使用同一个组件。在用新版本更新共享组件时，新版本会中断现有客户程序的执行。我们不可能不购买新版本，因为现有组件的新版本中提供了新功能。但我们可以小心操作，让程序保持向后兼容。但事情并不总是很顺利。

这个问题的解决方法是使用一种结构，允许安装共享组件的不同版本，客户程序使用它们在建立过程中使用的版本。这解决了许多问题，但并没有解决全部问题。如果在客户程序引用的组件中检测到了一个错误，该怎么办？我们可以更新这个组件，并确保客户程序使用新版本，而不是在建立过程中引用的那个版本。

因此，根据新版本中更改的类型，有时要使用更新的版本，有时则要使用旧一点的版本。所有这些都可以通过 .NET 结构来实现。

在 .NET 中，默认情况下使用原来引用的程序集。使用配置文件可以把引用重新定向于一个不同的版本。版本问题在绑定结构中有非常关键的作用——客户程序获得正确的程序集，在该程序集中，存储了其组件。

### 17.7.1 版本号

程序集的版本号由 4 个部分组成，例如 1.0.400.3300，各部分分别是：

```
<Major>.<Minor>.<Build>.<Revision>
```

这些号码根据应用程序配置来使用。

**注意：**

如果进行的改动与以前的版本不兼容，最好改变主号码或次号码，否则，就只修改建立或修正版本号。这样就可以假定把程序集重新定向到新版本，其中，只有修改了的建立和修正版本号是安全的。

在 Visual Studio 2008 中，使用项目设置中的程序集信息可以指定程序集的版本号。项目设置将程序集属性 `[AssemblyVersion]` 写入 `AssemblyInfo.cs` 文件：

```
[assembly: AssemblyVersion("1.0.0.0")]
```

除了定义全部四个版本号之外，还可以在第三或四个位置放置\*号：

```
[assembly: AssemblyVersion("1.0.*")]
```

在这个设置中，前两个数字指定主版本号和次版本号，\*表示建立版本号和修正版本号是自动生成的。建立版本号是自从 2000 年 1 月 1 日以来的天数，修正版本号表示自从当地时间的午夜开始的秒数。自动设置的版本号在开发期间是很有帮助的，但在发布之前，最好定义特定的版本号。

这个版本存储在程序集清单的 `assembly` 部分。

在客户应用程序中引用程序集，会在客户应用程序的程序集清单中存储引用的程序集版本。

### 17.7.2 编程获取版本

要查看在客户应用程序中使用的程序集版本，可以在 `SharedDemo` 类中添加 `GetAssemblyFullName()` 方法，返回程序集的强名。要使用 `Assembly` 类，应导入 `System.Reflection` 命名空间。

```
public string GetAssemblyFullName()
{
    return Assembly.GetExecutingAssembly().FullName;
}
```

`FullName` 是 `Assembly` 类的一个属性，这个属性包含了类名、版本、位置和公钥标记，在客户应用程序中调用 `GetAssemblyFullName()` 时，可以在其结果中看到这些内容。

在客户应用程序中，创建共享组件后，在 `Main()` 方法中添加一个对 `GetAssemblyFullName()` 的调用：

```
static void Main()
{
    SharedDemo quotes = new
        SharedDemo (@":\ProCSharp\Assemblies\Quotes.txt");
    Console.WriteLine(quotes.GetAssemblyFullName());
}
```

使用 `gacutil` 在全局程序集缓存中再次注册共享程序集 `SharedDemo` 的新版本。如果没有找到引用的版本，就会抛出一个异常 `System.IO.FileLoadException`，因为对正确程序集的绑定失败了。

成功运行后，可以看到引用的程序集的完整名称，如下所示。

```
SharedDemo, Version=1.0.0.0, Culture=neutral, PublicKeyToken=7d886a6f7b9f0292
Press any key to continue ...
```

使用这个客户程序，可以试试执行这个共享组件的不同配置。

### 17.7.3 应用程序配置文件

使用配置文件可以指定应绑定共享程序集的另一个版本，下面创建共享程序集 `SharedDemo` 的一个新版本，其主版本和次版本号是 1.1。我们不想重新建立客户程序，只想在现有的客户

程序中使用程序集的新版本。这适用于下述场合：共享程序集有一个错误需要修改，或者因为新版本是兼容的，所以要删除旧版本。

图 17-19 显示了全局程序集缓存查看器，其中 SharedDemo 程序集安装了 1.0.0.0 和 1.0.3300.0 版本。

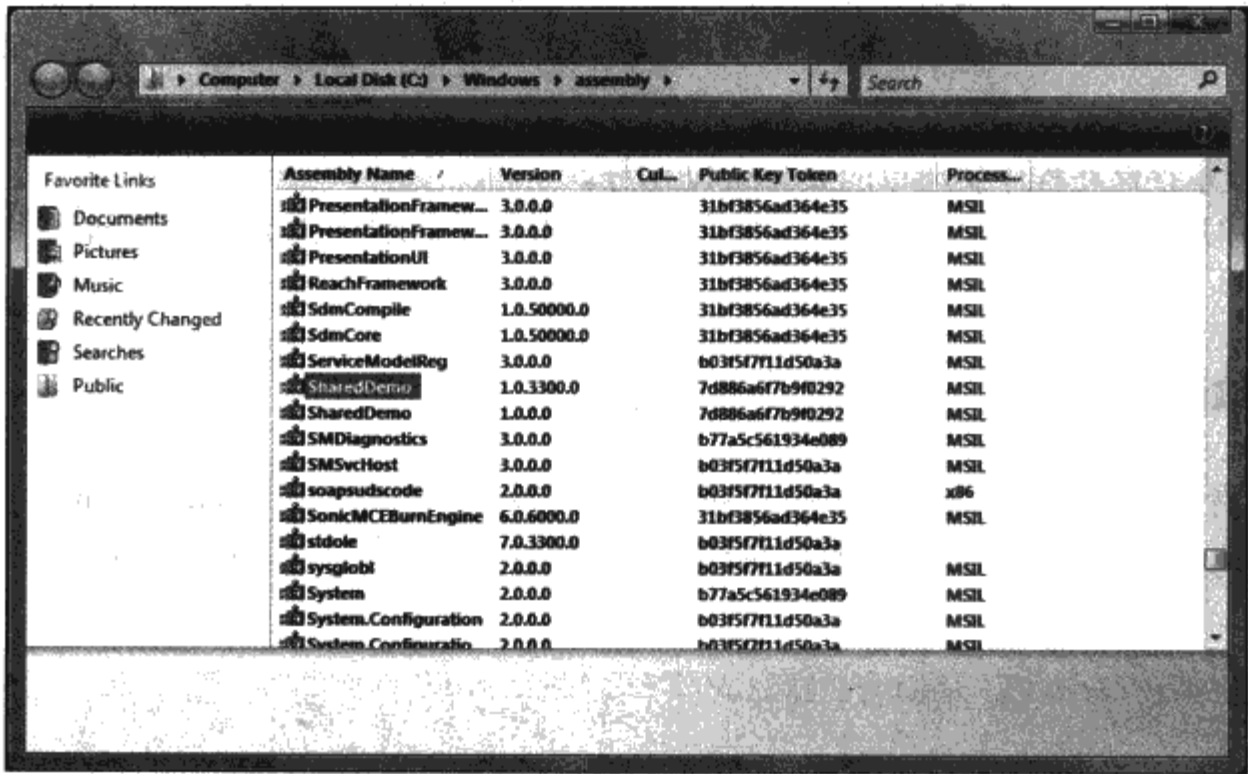


图 17-19

图 17-20 显示了客户应用程序的清单，其中客户程序引用了程序集 SharedDemo 的 1.0.0.0 版本。

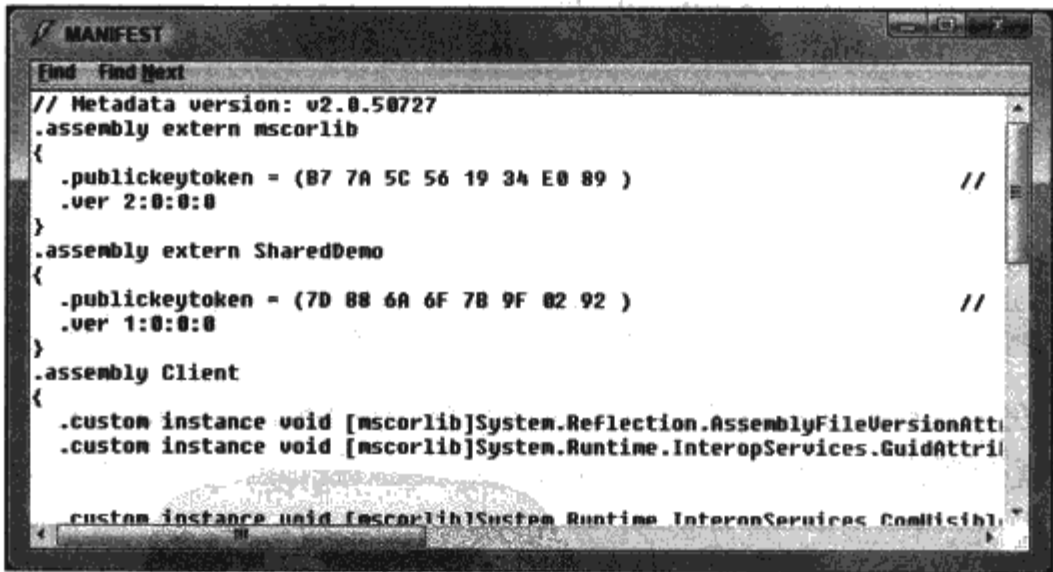


图 17-20

现在需要使用应用程序配置文件。不需要直接使用 XML 来处理，.NET Framework 配置工具(如图 17-21 所示)可以创建应用程序和机器配置文件。该工具是一个 MMC 插件，可以在控制面板的“管理工具”中启动。

提示：

这个工具附带在 Framework SDK 上，但没有附带在 .NET 运行库上，所以系统管理员不能使用这个工具。

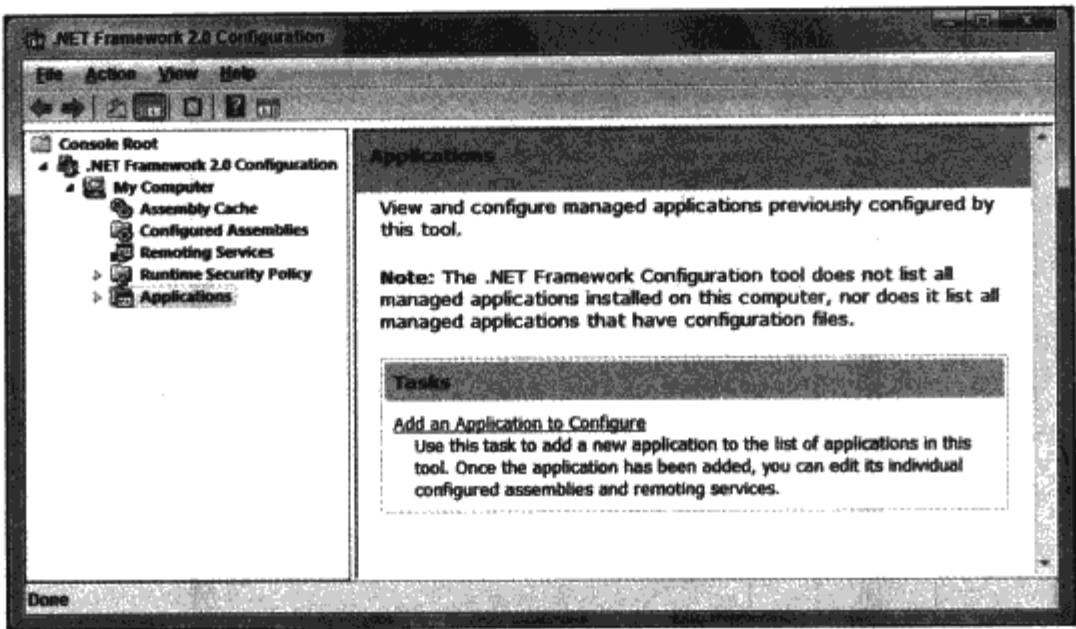


图 17-21

在左边选择 Applications，然后选择菜单 Action | Add，就可以选择要配置的.NET 应用程序。如果 Client.exe 应用程序没有显示在列表中，可以单击 Other...按钮，查找可执行文件。我们可以选择 Client.exe 应用程序，为该应用程序创建一个应用程序配置文件。在给.NET 配置工具添加了客户应用程序后，就可以查看程序集的从属文件，如图 17-22 所示。

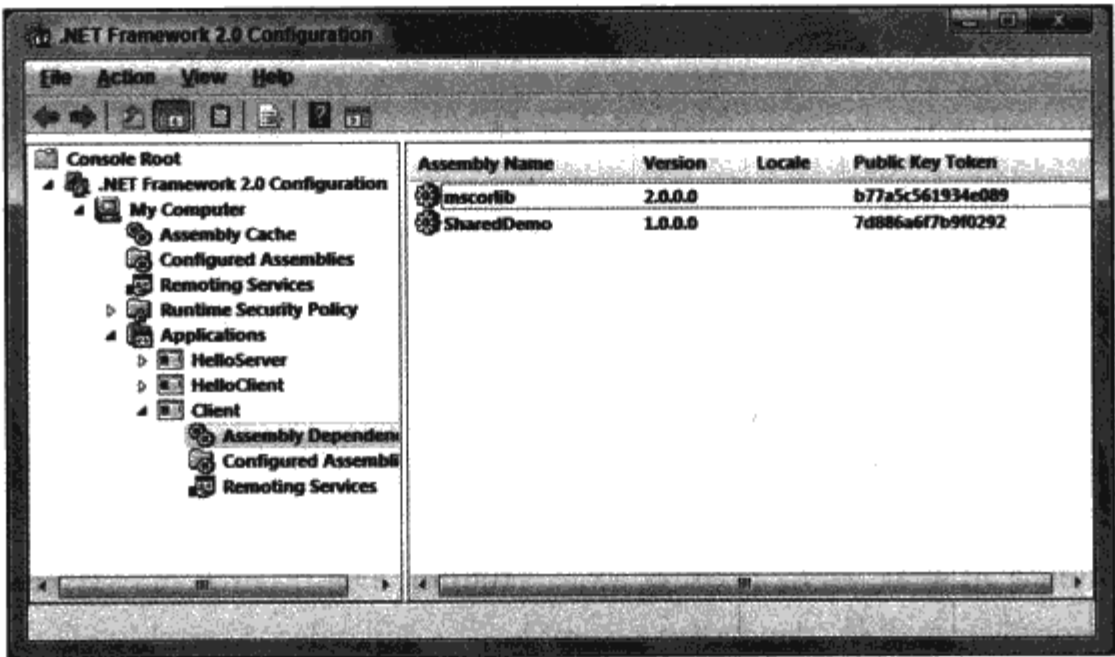


图 17-22

在树型视图中选择 Configured Assemblies 和菜单 Action | Add...，就可以在从属文件列表中配置程序集 SharedDemo 的从属文件。选择 Binding policy 选项卡，定义应使用的版本，如图 17-23 所示。

使用 Requested Version 可以指定客户程序集的清单中引用的版本。New Version 指定共享程序集的新版本。在图 17-23 中，我们指定应使用版本 1.0.3300.0，而不是使用 1.0.0.0 到 1.0.3300.0 之间的某个版本。



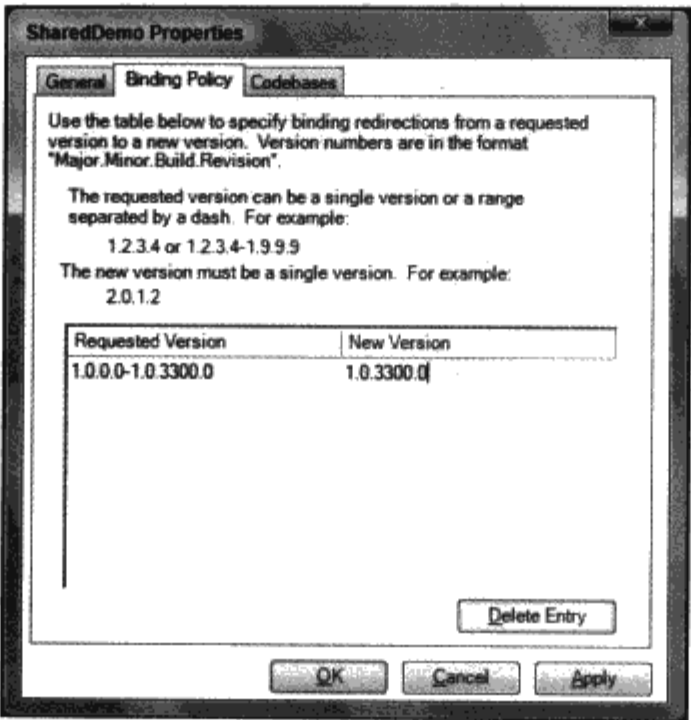


图 17-23

现在，在 Client.exe 应用程序所在的目录下有一个应用程序配置文件 Client.exe.config，其中包含 XML 代码：

```
<?xml version="1.0"?>
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="SharedDemo"
          publicKeyToken="7d886a6f7b9f0292" />
        <publisherPolicy apply="yes" />
        <bindingRedirect oldVersion="1.0.0.0-1.0.3300.0"
          newVersion="1.0.3300.0" />
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

使用 <runtime> 元素配置运行库设置，<runtime> 的子元素是 <assemblyBinding>，<assemblyBinding> 的子元素是 <dependentAssembly>。<dependentAssembly> 有一个必需的子元素 <assemblyIdentity>，使用该元素可以指定引用的程序集名称。Name 是 <assemblyIdentity> 唯一的一个必选属性。其可选属性是 publicKeyToken 和 culture。<dependentAssembly> 用于版本重定向的另一个子元素是 <bindingRedirect>，使用该元素可以指定从属程序集的新旧版本。

用这个配置文件启动客户程序，会得到引用的共享程序集的新版本。

17.7.4 发布方的策略文件

使用全局程序集缓存器中的共享程序集，可以使用发布方的策略避免版本冲突问题。假定有一个共享程序集由一些应用程序使用。如果在共享程序集中有一个关键的错误，会出现什么情况？可以看出，不需要重新建立所有使用该共享程序集的应用程序，因为可以使用配置文件重新定向到这个共享程序集的新版本。我们也许不了解所有使用该共享程序集的应用程序，但

要为所有这些应用程序修改错误。此时可以创建发布方的策略文件，把所有这些应用程序重新定向到该共享程序集的新版本。

注意：

发布方的策略文件只能应用于安装在全局程序集缓存中的共享程序集。

要建立发布者的策略，必须：

- 创建发布者的策略文件
- 创建发布者的策略程序集
- 把发布者的策略程序集添加到全局程序集缓存器中

### 1. 创建发布方的策略文件

发布方的策略文件是一个把现有版本或某个版本范围重新定向到新版本的 XML 文件。其使用的语法与应用程序配置文件相同，所以可以使用前面创建的同一个文件，把旧版本 1.0.0.0~1.0.3300.00 重新定向到新版本 1.0.3300.00 上。

把前面创建的文件重新命名为 `mypolicy.config`，把它用作发布方的策略文件，并删除元素 `<publisherPolicy>`。

```
<?xml version="1.0"?>
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="SharedDemo"
                          publicKeyToken="7d886a6f7b9f0292" />
        <bindingRedirect oldVersion="1.0.0.0-1.0.3300.0"
                          newVersion="1.0.3300.0" />
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

### 2. 创建发布方的策略程序集

要把发布方的策略文件与共享程序集关联起来，必须创建一个发布方策略程序集，把它放到全局程序集缓存器中。可以创建这种文件的工具是程序集连接程序 `al`。选项 `/linkresource` 把发布方的策略文件添加到生成的程序集中。生成的程序集名称必须以 `policy` 开头，其后是应重新定向的程序集的主次版本号，以及共享程序集的文件名。在本例中，发布方策略程序集必须命名为 `policy.1.0.SharedDemo.dll`，才能重新定向主版本号为 1、次版本号为 0 的程序集 `Shared Demo`。必须用 `/keyfile` 选项把一个密钥添加到这个发布方密钥上，新加的这个密钥与用于标识共享程序集 `SharedDemo` 的密钥相同，这样才能保证该版本从同一个发布方处重新定向。

```
al /linkresource:mypolicy.config /out:policy.1.0. SharedDemo.dll
/keyfile:...\mykey.snk
```

### 3. 将发布方的策略程序集添加到全局程序集缓存中

现在，可以使用实用程序 `gacutil` 把发布方的策略程序集添加到全局程序集缓存器中：

```
gacutil -i policy.1.0. SharedDemo.dll
```

现在，可以删除位于客户应用程序目录上的应用程序配置文件，并启动该客户应用程序。尽管客户程序集引用的是 1.0.0.0，但由于有了发布方策略，所以我们使用共享程序集的新版本 1.0.3300.0。

#### 4. 重写发布方策略

有了发布方策略，共享程序集的发布方就可以保证程序集的新版本与旧版本兼容。从传统 DLL 的变化来看，这种保证并不总是可靠。也许只有一个应用程序在使用新的共享程序集。为了修改使用新版本的应用程序中的错误，可以使用应用程序配置文件，重写发布方策略。

使用 .NET Framework 配置工具，可以取消对 Enable Publisher Policy 复选框的选择，重写发布方策略，如图 17-24 所示。

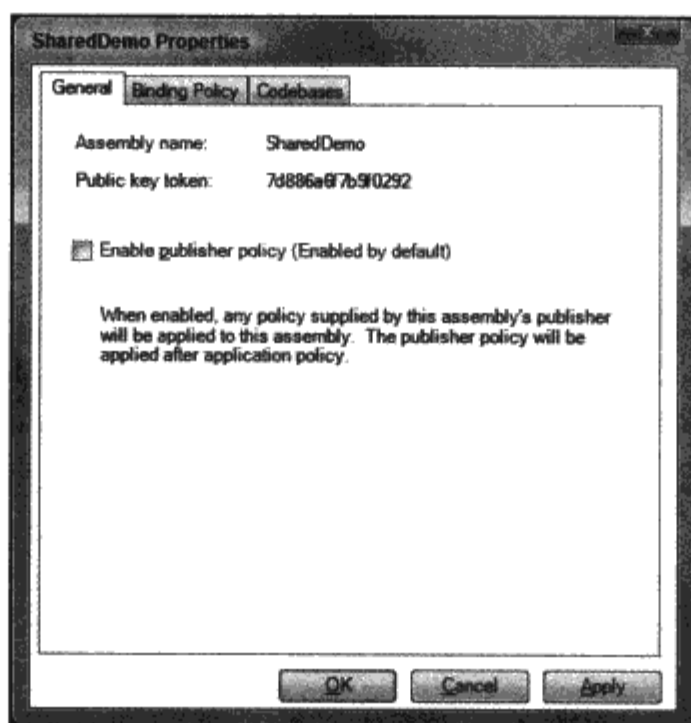


图 17-24

用 .NET Framework Configuration 禁用发布方策略，得到的配置文件将包含 XML 元素 `<publisherPolicy>` 和属性 `apply="no"`。

```
<?xml version="1.0"?>
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="SharedDemo" publicKeyToken="7d886a6f7b9f0292" />
        <publisherPolicy apply="no" />
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

禁用发布方策略，可以在应用程序配置文件中配置不同版本的重定向。

### 17.7.5 运行库的版本

不仅可以安装和使用程序集多个版本，还可以安装和使用 .NET 运行库 (CLR) 的多个版本。CLR 的 1.0、1.1 和 2.0(及未来)版本可以同时安装在一个操作系统上。Visual Studio 2008 默认在 CLR 2.0 和 .NET 2.0、3.0 或 3.5 上运行应用程序。在 CLR 2.0 中，改变了程序集的文件格式，所以不能在 CLR 1.1 上运行 CLR 2.0 应用程序。

如果应用程序是用 CLR 1.1 建立的，可以在只安装了 CLR 1.0 运行库的系统上运行。同样，如果未来的版本只改变了次版本号，就仍可以在 CLR 2.0 运行库版本上运行。

使用 CLR 1.0 建立的应用程序可以不加修改地在 CLR 1.1 上运行。如果操作系统安装了运行库的两个版本，应用程序就运行在建立它时使用的版本上。但是，如果操作系统仅安装了 1.1 版本，而应用程序是使用 1.0 版本建立的，就会使用 1.1 新版本运行该应用程序。注册键 HKEY\_LOCAL\_MACHINE\Software\Microsoft\NETFramework\policy 列出了某个运行库的版本范围。

如果应用程序是使用 .NET 1.1 版本建立的，只要其中不包含只能在 .NET 1.1 上使用的类和方法，它也可以不加修改地在 .NET 1.0 上运行。为此，需要使用应用程序配置文件。

在应用程序配置文件中，不仅可以重新定向引用程序集的版本，还可以定义运行库所需的版本。在一台机器上可以安装不同的 .NET 运行库版本。在应用程序配置文件中可以指定应用程序需要的版本。元素 `<supportedVersion>` 标记了应用程序支持的运行库版本：

```
<?xml version="1.0"?>
<configuration>
  <startup>
    < supportedVersion version="v1.1.4322" />
    <supportedVersion version="v1.0.3512" />
  </startup>
</configuration>
```

为了使 .NET 1.0 应用程序仍能运行在 .NET 1.1 运行库上，有一点要注意。`<supportedVersion>` 元素是 .NET 1.1 中的新元素，.NET 1.0 用 `<requiredRuntime>` 元素指定需要的运行库版本。所以对于 .NET 1.0 应用程序，必须进行两个配置：

```
<?xml version="1.0"?>
<configuration>
  <startup>
    <supportedRuntime version="v1.1.4322" />
    <supportedRuntime version="v1.0.3705" />
    <requiredRuntime version="v1.0.3512" /> safeMode="true" />
  </startup>
</configuration>
```

**注意：**

`<requiredRuntime>` 并没有重写 `<supportedRuntime>` 的配置，因为 `<requiredRuntime>` 仅用于 .NET 1.0，而 `<supportedRuntime>` 可用于 .NET 1.1 和后续版本。

**提示：**

不能为库配置支持的运行库。库总是使用应用程序进程选择的运行库。

## 17.8 小结

程序集是.NET 平台的新安装单元。Microsoft 从以前的体系结构中吸取教训,进行了全新的设计,以避免出现旧问题。本章讨论了程序集的特性:程序集是自我描述的,不需要类型库和注册信息。版本的依赖性会准确地记录下来,这样,使用程序集时,旧 DLL 的 DLL Hell 问题就不复存在了。由于具备这些特性,开发、部署和管理就容易多了。

本章区分了私有和共享程序集,并介绍了如何创建共享程序集。有了私有程序集,就不必关心唯一性和版本冲突问题了,因为这些程序集仅在一个应用程序中复制和使用。共享程序集需要使用一个密钥来保持其唯一性、确定其版本。我们介绍了全局程序集缓存,它可以用作共享程序集的智能存储。

使用本机图像生成器可以更快地启动应用程序。有了本机图像生成器,就不需要运行 JIT 编译器,因为本机代码是在安装期间创建的。

本章还阐述了避免版本冲突问题,以使用与在开发过程中使用的版本不同的程序集。这是通过发布方策略和应用程序配置文件实现的。最后还讨论了 probing 如何使用私有程序集。

我们还讨论了如何动态加载程序集,在运行期间创建程序集。如果要了解更多的信息,可以参阅第 36 章介绍的.NET3.5 中的插件模型。



# 第 18 章

## 跟踪和事件

第 14 章介绍了错误和异常处理。除了处理异常代码之外，还可以在运行的应用程序中获得一些实时信息，找出应用程序在生产过程中某些问题的原因，或者监控需要的资源，以尽早适应较大的用户负载。这就是命名空间 `System.Diagnostics` 的作用。

应用程序不抛出异常，但有时不像期望的那样运行。应用程序可能在大多数系统上都运行良好，只在几个系统上出问题。在实时系统上，可以通过改变配置值，来改变记录方式，获得应用程序运行的详细的实时信息。这可以用跟踪功能来实现。

如果应用程序出了问题，就需要通知系统管理员。使用事件查看器，系统管理员可以交互地监控应用程序的问题，通过添加订阅功能来了解发生的特殊事件。事件日志机制允许写入应用程序的信息。

为了分析应用程序需要的资源，在指定的时间间隔内监控应用程序，计划另一个应用程序的分布或扩展系统资源，系统管理员可使用性能监控器。使用性能计数器可以写入应用程序的实时数据。

本章介绍这三个功能，演示如何在应用程序中使用它们：

- 跟踪
- 事件日志
- 性能监控

### 18.1 跟踪

利用跟踪功能可以查看正在运行的应用程序的消息。为了获得这些信息，可以在调试器中启动应用程序。在调试过程中，可以单步执行应用程序，在特定的代码行上设置断点，或在满足某些条件时设置断点。调试的问题是已发布的程序可能以不同的方式运行。例如，程序在断点处停止运行时，应用程序的其他线程也会挂起。另外，在发布版本中，编译器生成的输出进行了优化，因此会产生不同的效果。此时就需要从发布版本中获得信息。跟踪消息要写入调试和发布代码中。

下面的场景描述了跟踪功能的作用。在部署应用程序后，它运行在一个系统中时没有问题，而在另一个系统上出现了问题。在出问题的系统上打开跟踪功能，就会获得应用程序中所出现问题的详细信息。在运行时没有问题的系统上，将跟踪功能配置为把错误消息重定向到 Windows 事件日志系统中。系统管理员会查看重要的错误，跟踪功能的开销非常小，因为仅在需要时配

置跟踪级别。

跟踪体系架构有 4 个主要部分：

- 源是跟踪信息的源头，使用源可以发送跟踪消息。
- 开关定义了要记录的信息级别。例如，可以只请求错误信息或详细的信息。
- 跟踪监听器定义了写入跟踪消息的位置。
- 监听器可以关联过滤器。过滤器定义了监听器应写入哪些跟踪消息。这样，就可以给同一个源头使用不同的监听器，写入不同级别的信息。

图 18-1 显示了主要的跟踪类和它们在 Visual Studio 类图中的连接方式。TraceSource 类使用一个开关来定义要记录的信息。它有一个相关的 TraceListenerCollection，来指定跟踪消息的写入位置。集合由 TraceListener 对象组成，每个监听器都连接了一个 TraceFilter。

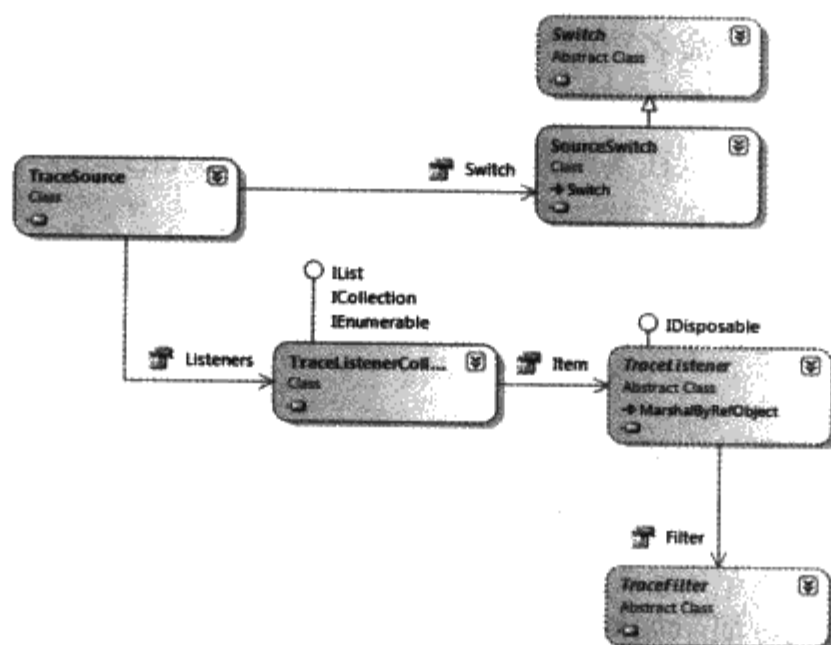


图 18-1

### 18.1.1 跟踪源

用 TraceSource 类可以写入跟踪消息。跟踪需要编译器设置的 Trace 标志。在 Visual Studio 项目中，Trace 标志在调试和发布版本中已进行了默认设置，但可以通过项目的 Build 属性修改它。

**提示：**

与写入跟踪信息的 Trace 相比，TraceSource 类较难使用，但提供的选项较多。

要写入跟踪消息，需要创建一个新的 TraceSource 实例。在构造函数中，定义了跟踪源的名称。方法 TraceInformation 在跟踪输出中写入一个消息。TraceEvent() 方法不只写入消息，还需要一个 TraceEventType 类型的枚举值，来定义跟踪消息的类型。TraceEventType.Error 把消息指定为一个错误消息。可以用跟踪开关将它定义为只查看错误消息。TraceEvent() 方法的第二个参数是一个标识符。ID 可以用于应用程序内部。例如，可以使用 id 1 进入一个方法，用 id 2 退出方法。TraceEvent() 方法是重载的，只需要 TraceEventType 和 ID 这两个参数。使用重载方法的第三个参数，可以传送写入跟踪的消息。TraceEvent() 方法还可以传送格式字符串和任意数

量的参数，其方式与 `Console.WriteLine()` 相同。`TraceInformation()` 只是调用标识符为 0 的 `TraceEvent()` 方法。`TraceInformation()` 方法是 `TraceEvent()` 的一个简化版本。在 `TraceData()` 方法中，可以传送任意对象，例如异常实例，来替代消息。要确保数据由监听器写入，且不存储在内存中，就需要执行 `Flush()`。如果不再需要跟踪源，就可以调用 `Close()` 方法，关闭与跟踪源相关的所有监听器。`Close()` 方法也会执行 `Flush()`。

```
TraceSource source1 = new TraceSource("Wrox.ProCSharp.Tracing");
source1.TraceInformation("Info message");
source1.TraceEvent(TraceEventType.Error, 3, "Error message");
source1.TraceData(TraceEventType.Information, 2, new int[] { 1, 2, 3 });
source1.Flush();
source1.Close();
```

#### 警告：

可以在应用程序中使用不同的跟踪源。为不同的库定义不同的跟踪源，这样就可以为应用程序的不同部分打开不同的跟踪级别。要使用跟踪源，必须知道它的名称。跟踪源的常用名称与命名空间的名称相同。

`TraceEventType` 枚举作为一个变元传送给 `TraceEvent()` 方法，该枚举定义了下面的级别，来指定问题的严重级别：`Verbose`、`Information`、`Warning`、`Error`、`Critical`。`Critical` 定义了致命错误或使应用程序崩溃的错误；`Error` 表示可恢复的错误。`Verbose` 级别的跟踪消息可给出详细的调试信息。`TraceEventType` 还定义了操作级别：`Start`、`Stop`、`Suspend` 和 `Resume`。这些级别在逻辑操作中定义了时间事件。

代码在编写时没有显示任何跟踪消息，因为与跟踪源相关的开关是关闭的。

### 18.1.2 跟踪开关

要启用或禁用跟踪消息，可以配置一个跟踪开关。跟踪开关是派生自抽象基类 `Switch` 的类。派生类是 `BooleanSwitch`、`TraceSwitch` 和 `SourceSwitch`。类 `BooleanSwitch` 可以打开和关闭，其他两个类提供了由 `TraceLevel` 枚举定义的范围级别。要配置跟踪开关，必须知道与 `TraceLevel` 枚举相关的值。`TraceLevel` 定义的值有 `Off`、`Error`、`Warning`、`Info` 和 `Verbose`。

设置 `TraceSource` 的 `Switch` 属性，可以用编程方式关联跟踪开关。这里关联的开关是 `SourceSwitch` 类型，其名称是 `MySwitch`，级别为 `Verbose`。

```
TraceSource source1 = new TraceSource("Wrox.ProCSharp.Tracing ");
source1.Switch = new SourceSwitch("MySwitch", "Verbose");
```

把级别设置为 `Verbose`，表示应写入所有的跟踪消息。如果把值设置为 `Error`，就应只显示错误消息。把值设置为 `Information`，表示显示错误、警告和详细消息。在 `Output` 窗口中运行调试器，再次写入跟踪消息，就可以看到这些消息。

通常，要改变开关级别，不希望重新编译应用程序，而最好只改变配置。跟踪源可以在应用程序配置文件中配置。跟踪的配置在 `<system.diagnostics>` 元素中完成。跟踪源在 `<source>` 元素中定义为 `<sources>` 的一个子元素。配置文件中的跟踪源名称必须准确匹配程序代码中的跟踪源名称。这里跟踪源的开关类型是 `System.Diagnostics.SourceSwitch`，名称是 `MySourceSwitch`。该开关在 `<switches>` 段中定义，开关的级别设置为 `Verbose`。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.diagnostics>
    <sources>
      <source name="Wrox.ProCSharp.Tracing" switchName="MySourceSwitch"
        switchType="System.Diagnostics.SourceSwitch" />
    </sources>
    <switches>
      <add name="MySourceSwitch" value="Verbose"/>
    </switches>
  </system.diagnostics>
</configuration>
```

现在，要修改跟踪级别，只需修改配置文件，而无需重新编译代码。修改配置文件后，必须重新启动应用程序。

在调试会话中运行时，当前的跟踪消息只写入 Visual Studio 的 Output 窗口。添加跟踪监听器可以改变这种情况。

18.1.3 跟踪监听器

默认情况下，跟踪信息写入 Visual Studio 调试器的 Output 窗口。只要改变应用程序的配置，就可以将跟踪输出重定向到不同的位置。

跟踪信息应写入什么位置由跟踪监听器确定。跟踪监听器派生自抽象基类 `TraceListener`。`.NET Framework` 定义的跟踪监听器如表 18-1 所示。

表 18-1

跟踪监听器	说 明
DefaultTraceListener	默认的跟踪监听器会自动添加到 <code>Trace</code> 类的监听器集合中。默认输出会写入关联的调试器中。在 Visual Studio 中，它们会在调试会话过程中显示在 Output 窗口中
EventLogTraceListener	<code>EventLogTraceListener</code> 将跟踪信息写入事件日志。在 <code>EventLogTraceListener</code> 的构造函数中，可以指定事件日志源或 <code>EventLog</code> 类型的对象。事件日志详见本章后面的内容
TextWriterTraceListener	利用 <code>TextWriterTraceListener</code> 跟踪，可以将输出写入文件、 <code>TextWriter</code> 或 <code>Stream</code> 。文件处理的内容可参见第 25 章 <code>TextWriterTraceListener</code> 是 <code>ConsoleTraceListener</code> 、 <code>DelimitedListTraceListener</code> 和 <code>XmlWriterTraceListener</code> 的基类
ConsoleTraceListener	<code>ConsoleTraceListener</code> 将跟踪消息写入控制台
DelimitedListTraceListener	<code>DelimitedListTraceListener</code> 将跟踪消息写入有分隔符的文件。使用跟踪输出选项，可以定义许多不同的跟踪信息，例如进程 ID、时间等，用有分隔符的文件更容易读取信息
XmlWriterTraceListener	除了使用有分隔符的文件之外，还可以使用 <code>XmlWriterTraceListener</code> 把跟踪信息重定向到 XML 文件中

(续表)

跟踪监听器	说 明
IisTraceListener	IisTraceListener 是 .NET 3.0 的新类
WebPageTraceListener	ASP.NET 有另一个跟踪选项，在动态创建的输出文件 trace.axd 中获得网页的 ASP.NET 跟踪信息。如果配置 WebPageTraceListener，System.Diagnostics 跟踪信息也写入文件 trace.axd 中

.NET Framework 发布了许多可写入跟踪信息的监听器。如果监听器不满足用户的需要，就可以从基类 `TraceListener` 中派生一个类，创建定制的监听器。使用定制的监听器，可以将跟踪信息写入 Web 服务，将消息写入手机...其实这没有那么有趣。使用 `Verbose` 跟踪级别，这会变得相当昂贵。

创建一个监听器对象，将它赋予 `TraceSource` 类的 `Listeners` 属性，就可以用编程方式配置跟踪监听器。但是，只改变配置，就定义另一个监听器会比较有趣。

可以把监听器配置为 `<source>` 元素的子元素。在监听器中，可以定义监听器类的类型，使用 `initializeData` 指定监听器的输出写入什么位置。这里的配置将 `XmlWriterTraceListener` 定义为写入 `demotrace.xml` 文件，将 `DelimitedListTraceListener` 定义为写入 `demotrace.txt` 文件。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.diagnostics>
    <sources>
      <source name="Wrox.ProCSharp.Tracing" switchName="MySourceSwitch"
        switchType="System.Diagnostics.SourceSwitch">
        <listeners>
          <add name="xmlListener"
            type="System.Diagnostics.XmlWriterTraceListener"
            traceOutputOptions="None" initializeData="c:/logs/demotrace.xml" />
          <add name="delimitedListener" delimiter=":"
            type="System.Diagnostics.DelimitedListTraceListener"
            traceOutputOptions="DateTime, ProcessId"
            initializeData="c:/logs/demotrace.txt" />
        </listeners>
      </source>
    </sources>
    <switches>
      <add name="MySourceSwitch" value="Verbose"/>
    </switches>
  </system.diagnostics>
</configuration>
```

**提示：**  
此时可能从 XML 模式中得到一个与分隔符属性声明相关的警告，可以忽略它。

在监听器中，还可以指定把哪些额外信息写入跟踪日志。这些信息和 XML 特性 `traceOutputOptions` 一起定义，由 `TraceOptions` 枚举定义。该枚举定义了 `Callstack`、`DateTime`、`LogicalOperationStack`、`ProcessId`、`ThreadId` 和 `None`。需要的信息可以用逗号分隔符添加到 XML 特性 `traceOutputOptions` 中，如带分隔符的跟踪监听器所示。



`DelimitedListTraceListener` 中带分隔符的文件输出, 包括进程 ID 和日期/时间, 如下所示:

```
"Wrox.ProCSharp.Tracing":Information:0:"Info message"::4188:""::
"2007-01-23T12:38:31.3750000Z"::
"Wrox.ProCSharp.Tracing":Error:3:"Error message"::4188:""::
"2007-01-23T12:38:31.3810000Z"::
```

`XmlWriterTraceListener` 中的 XML 输出总是包含计算机名、进程 ID、线程 ID、消息、创建的时间、源和活动 ID。其他字段, 如调用堆栈、逻辑操作堆栈、时间戳等, 都依赖于跟踪输出选项。

#### 提示:

可以使用 `XmlDocument` 和 `XPathNavigator` 类分析 XML 文件中的内容。这些类详见第 28 章。

如果监听器应由多个跟踪源使用, 就可以将监听器配置添加到 `<sharedListeners>` 元素中, 它独立于跟踪源。配置为共享监听器的监听器名称必须在跟踪源的监听器中引用:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.diagnostics>
    <sources>
      <source name="Wrox.ProCSharp.Tracing" switchName="MySourceSwitch"
        switchType="System.Diagnostics.SourceSwitch">
        <listeners>
          <add name="xmlListener"
            type="System.Diagnostics.XmlWriterTraceListener"
            traceOutputOptions="None" initializeData="c:/logs/demotrace.xml" />
          <add name="delimitedListener" />
        </listeners>
      </source>
    </sources>
    <sharedListeners>
      <add name="delimitedListener" delimiter=":"
        type="System.Diagnostics.DelimitedListTraceListener"
        traceOutputOptions="DateTime, ProcessId"
        initializeData="c:/logs/demotrace.txt" />
    </sharedListeners>
    <switches>
      <add name="MySourceSwitch" value="Verbose"/>
    </switches>
  </system.diagnostics>
</configuration>
```

### 18.1.4 过滤器

每个监听器都有一个 `Filter` 属性, 它定义了监听器是否应写入跟踪消息。例如, 多个监听器可以与同一个跟踪源一起使用。其中一个监听器将详细信息写入日志文件, 另一个监听器将错误信息写入事件日志。在监听器写入跟踪信息之外, 调用相关过滤器对象的 `ShouldTrace()` 方法, 确定是否应写入跟踪信息。

过滤器是派生自抽象基类 `TraceFilter` 的类。`.NET 3.0` 提供了两个过滤器: `SourceFilter` 和 `EventTypeFilter`。使用 `SourceFilter`, 可以指定只从指定的源中写入跟踪信息。`EventTypeFilter` 是

对开关功能的扩展。使用开关,可以根据跟踪级别,确定事件源是否应将跟踪信息写入监听器。如果写入跟踪信息,监听器就可以使用过滤器确定是否应写入信息。

改变的配置现在确定,只有严重级别为警告或更高,带分隔符的监听器才应写入跟踪信息,因为定义了 `EventTypeFilter`。XML 监听器指定了 `SourceFilter`,只接收源 `Wrox.ProCSharp.Tracing` 中的跟踪信息。如果定义了大量的跟踪源,以便将跟踪信息写入同一个监听器,就可以修改该监听器的配置,只处理指定源中的跟踪信息。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.diagnostics>
    <sources>
      <source name="Wrox.ProCSharp.Tracing" switchName="MySourceSwitch"
        switchType="System.Diagnostics.SourceSwitch">
        <listeners>
          <add name="xmlListener" />
          <add name="delimitedListener" />
        </listeners>
      </source>
    </sources>
    <sharedListeners>
      <add name="delimitedListener" delimiter=":"
        type="System.Diagnostics.DelimitedListTraceListener"
        traceOutputOptions="DateTime, ProcessId"
        initializeData="c:/logs/demotrace.txt">
        <filter type="System.Diagnostics.EventTypeFilter"
          initializeData="Warning" />
        </add>
      <add name="xmlListener"
        type="System.Diagnostics.XmlWriterTraceListener"
        traceOutputOptions="None" initializeData="c:/logs/demotrace.xml">
        <filter type="System.Diagnostics.SourceFilter"
          initializeData="Wrox.ProCSharp.Tracing" />
        </add>
      </sharedListeners>
    <switches>
      <add name="MySourceSwitch" value="Verbose"/>
    </switches>
  </system.diagnostics>
</configuration>
```

跟踪架构可以扩展。可以编写一个派生自 `TraceListener` 基类的定制监听器,同样,也可以创建一个派生自 `TraceFilter` 的定制过滤器。因此,可以创建一个过滤器,根据时间、以后发生的异常或天气,指定写入跟踪消息。

### 18.1.5 断言

跟踪的另一个功能是断言。断言是程序路径中的重要问题。使用断言,可以显示信息和错误,中止或继续运行应用程序。如果写入一个由另一个开发人员使用的库,断言会提供许多帮助。

在 `Foo()` 方法中, `Trace.Assert()` 检查参数 `o`, 确定它是否不为空。如果条件是 `false`, 就会显示如图 18-2 所示的错误信息。如果条件是 `true`, 程序会继续运行。`Bar()` 方法包含一个 `Trace.Assert()` 例子, 它验证参数是否大于 10, 且小于 20。如果条件是 `false`, 就再次显示错误信息。

```

static void Foo(object o)
{
    Trace.Assert(o != null, "Expecting an object");
    Console.WriteLine(o);
}

static void Bar(int x)
{
    Trace.Assert(x > 10 && x < 20, "x should be between 10 and 20");
    Console.WriteLine(x);
}

static void Main()
{
    Foo(null);
    Bar(3);
}

```

可以用<assert 元素创建一个应用程序配置文件，禁用断言信息：

```

<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.diagnostics>
    <assert assertuientabled="false"/>
  </system.diagnostics>
</configuration>

```

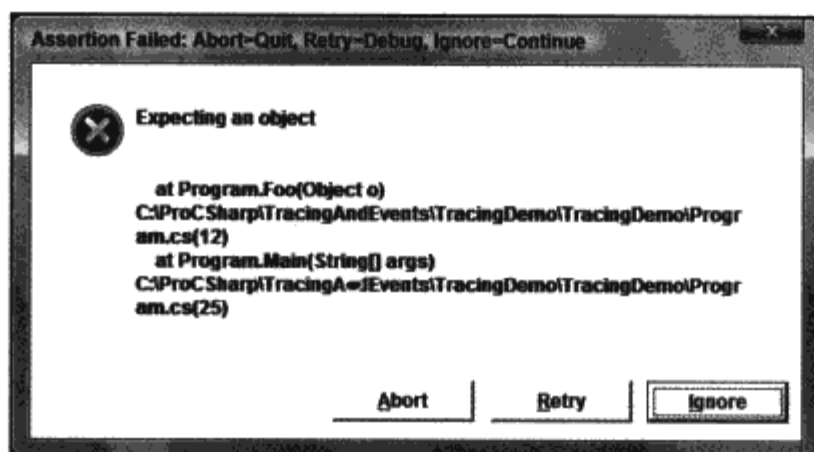


图 18-2

## 18.2 事件日志

系统管理员使用事件查看器获得系统和应用程序的重要信息和警告信息。应将应用程序的错误信息写入事件日志，再用事件查看器读取这些信息。

如果配置了 `EventLogTraceListener` 类，跟踪信息就可以写入事件日志。`EventLogTraceListener` 类有一个与之相关的 `EventLog` 对象，可以写入事件日志项。也可以直接使用 `EventLog` 类读写事件日志。

本节介绍如下内容：

- 事件日志体系架构
- `System.Diagnostics` 命名空间中用于事件日志的类
- 在服务和其他应用程序类型中添加事件日志功能

- 用 EventLog 类的 EnableRaisingEvents 属性创建事件日志监听器
- 图 18-3 显示了调制解调器的一个日志项。

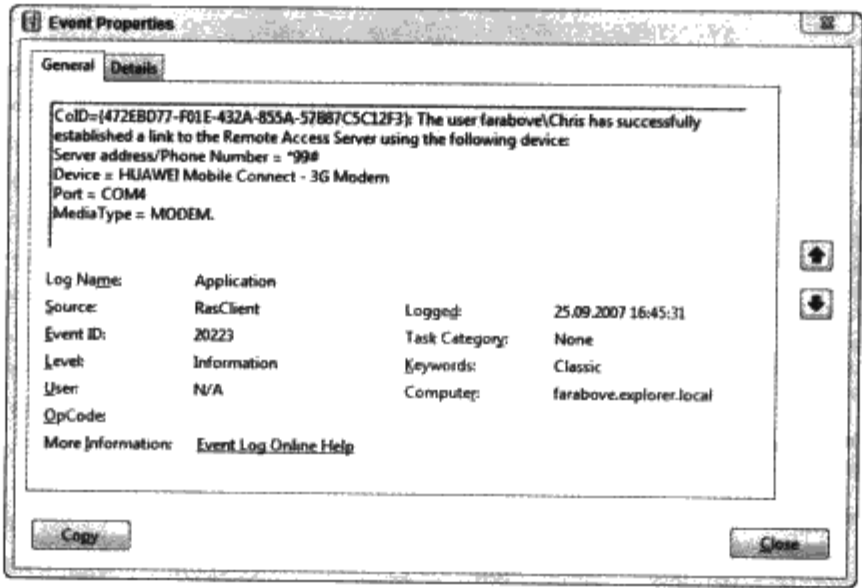


图 18-3

对于定制的事件日志，可以使用 System.Diagnostics 命名空间中的类。

18.2.1 事件日志体系架构

事件日志信息存储在几个日志文件中。最重要的日志文件是应用程序、安全性和系统日志文件。查看事件日志服务的注册表配置，会注意到 HKEY\_LOCAL\_MACHINE\System\CurrentControlSet\Services\Eventlog 下的几项带有指向特定文件的配置。系统日志文件用于系统和设备驱动程序。应用程序和服务则将事件写入应用程序日志。安全性日志是应用程序的只读日志。操作系统的审计功能使用安全性日志。每个应用程序还可以创建定制类别和日志文件，在其中写入事件日志项。例如，Windows OneCars 和 Media Center 就可以这么做。

使用管理工具“事件查看器”就可以读取这些事件。要启动事件查看器，可以在 Visual Studio 的 Server Explorer 中右击 Event Logs 项，从弹出的菜单中选择 Launch Event Viewer。事件查看器如图 18-4 所示。

在事件日志中，可以看到如下信息：

- 类型：该类型可以是 Information、Warning 或 Error。Information 是不常成功的操作，Warning 表示不太重要的问题，Error 表示重要问题。其他类型有 FailureAudit 和 SuccessAudit，但这些类型仅用于安全性日志。
- 日期：Date and Time 显示事件发生的时间。
- 源：Source 是记录事件的软件名。应用程序日志的源在如下元素中配置：

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Eventlog\Application[ApplicationName]
```

在这个键下，EventMessageFile 值配置为指向一个源 DLL，它保存了错误信息。

- 类别：Category 可以定义为允许在使用事件查看器时过滤事件日志。类别也可以通过事件源来定义。
- 事件标识符：Event identifier 指定某个事件信息。

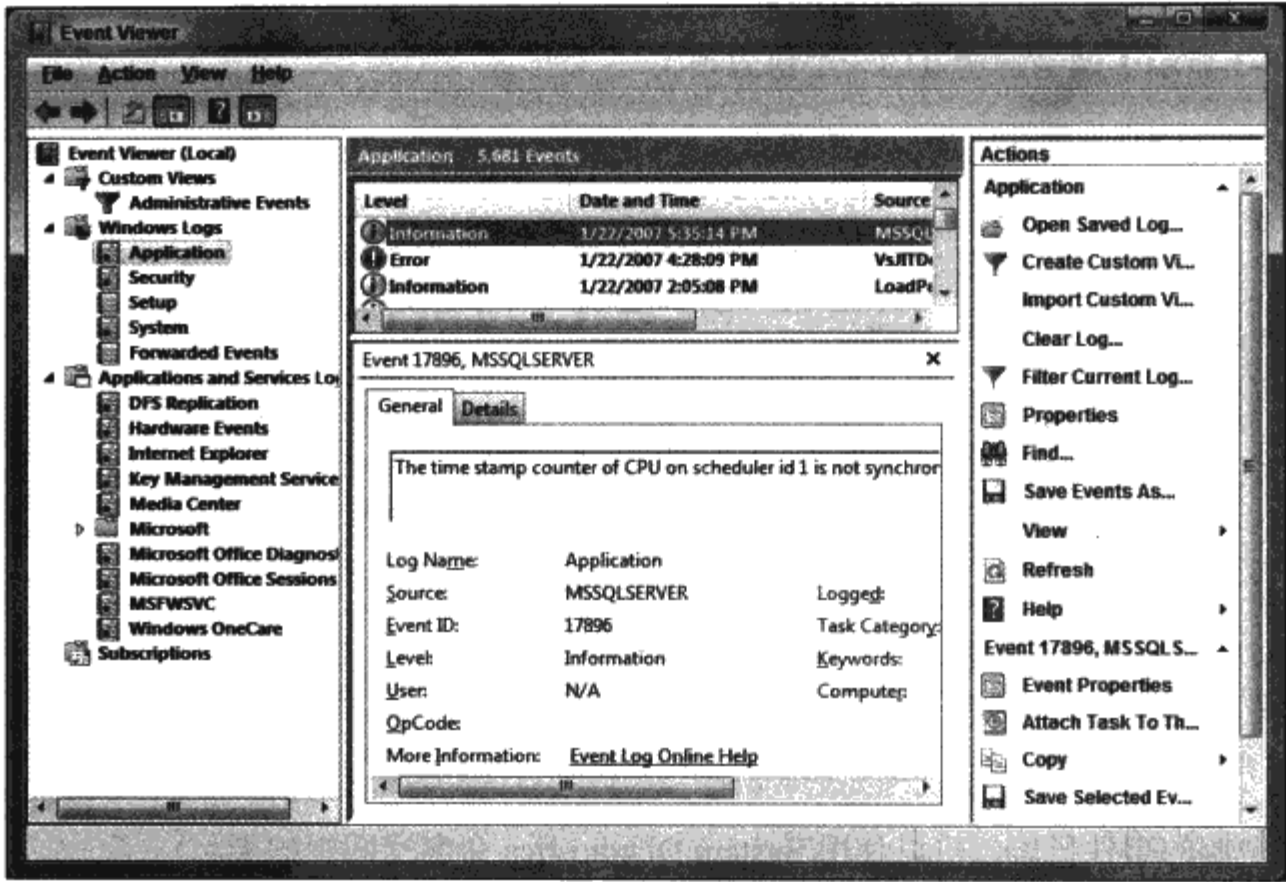


图 18-4

18.2.2 事件日志类

System.Diagnostics 命名空间中的一些类用于事件日志，如表 18-2 所示。

表 18-2

类	说 明
EventLog	使用 EventLog 类可以读写事件日志中的项，把应用程序建立为事件源
EventLogEntry	EventLogEntry 类表示事件日志中的一项。使用 EventLogEntryCollection 可以迭代 EventLogEntry 项
EventLogInstaller	EventLogInstaller 类是 EventLog 组件的安装程序。EventLogInstaller 调用 EventLog.CreateEventSource()创建事件源
EventLogTraceListener	使用 EventLogTraceListener 可以把跟踪信息写入事件日志。这个类实现了抽象类 TraceListener

事件日志的核心是 EventLog 类。这个类的成员如表 18-3 所示。

表 18-3

EventLog 类的成员	说 明
Entries	使用 Entries 属性可以读取事件日志。Entries 返回一个 EventLogEntryCollection，它包含的 EventLogEntry 对象存储了事件信息。不需要调用 Read()方法。只要访问这个属性，就会填充该集合



(续表)

EventLog 类的成员	说 明
Log	使用 Log 属性可指定用于读写事件日志的记录
LogDisplayName	LogDisplayName 是一个只读属性，返回日志的显示名称
MachineName	使用 MachineName 可以指定在哪个系统上读写日志项
Source	Source 属性指定要写入日志项的源
CreateEventSource()	CreateEventSource()创建一个新的事件源和一个新的日志文件(如果在该方法中指定了新的日志文件)
DeleteEventSource()	要删除一个事件源，可以调用 DeleteEventSource()
SourceExists()	在创建事件源之前，可以使用这个元素验证该源是否已存在
WriteEntry() WriteEvent()	用 WriteEntry()或 WriteEvent()方法可以写入事件日志项。WriteEntry()比较简单，只需传送一个字符串即可。WriteEvent()方法比较灵活，这里可以使用独立于应用程序的信息文件，还可以支持本地化
Clear()	Clear()方法删除事件日志中的所有项
Delete()	Delete()删除一个完整的事件日志

18.2.3 创建事件源

在写入事件之前，必须创建一个事件源。为此，可以使用 EventLog 类或 EventLogInstaller 类的 CreateEventSource()方法。在创建事件源时需要管理员权限，所以最好用一个安装程序定义新事件源。

提示：  
第 16 章介绍了如何创建安装程序。

下面的例子验证了事件日志源 EventLogDemoApp 已存在。如果它不存在，就实例化一个 EventSourceCreationData 类型的对象，定义源名 EventLogDemoApp 和日志名 ProCSharpLog。这里，该源的所有事件都写入 ProCSharpLog 事件日志。默认为应用程序日志。

```
if (!EventLog.SourceExists("EventLogDemoApp"))
{
    EventSourceCreationData eventSourceData =
        new EventSourceCreationData("EventlogDemoApp", "ProCSharpLog");
    EventLog.CreateEventSource(eventSourceData);
}
```

事件源的名称是写入事件的应用程序的标识符。系统管理员在读取日志时，该信息有助于识别事件日志项，将它们映射到应用程序类别上。事件日志源的名称可以随意，如用于性能监听器的 LoadPerf，用于 Microsoft SQL Server 的 MSSQLSERVER，用于 Windows 安装程序的 MsiInstaller、Winlogon、Tcpip、TimeService 等。

为事件日志设置名称 `Application`，会把事件日志项写入应用程序日志。也可以创建自己的日志，方法是指定不同的应用程序日志名。日志文件位于 `<windows>\System32\WinEvt\Logs` 目录下。

使用 `EventSourceCreationData`，还可以为事件日志指定更多的特性，如表 18-4 所示。

表 18-4

EventSourceCreationData	说 明
Source	属性 Source 可获取或设置事件源的名称
LogName	LogName 定义了事件日志项写到什么日志中。默认为应用程序日志
MachineName	使用 MachineName 可以定义读写日志项的系统
CategoryResourceFile	使用 CategoryResourceFile 属性可以定义类别的资源文件。类别可以用于过滤单一源中的事件日志项
CategoryCount	CategoryCount 属性定义了类别资源文件中的类别个数
MessageResourceFile	除了指定程序应写入事件日志中的信息之外，信息还可以在一个资源文件中定义，该文件由 MessageResourceFile 属性指定。该资源文件中的信息是可以本地化的
ParameterResourceFile	资源文件中的信息可以带参数。参数可以用一个资源文件中定义的字符串替代，该资源文件由 ParameterResourceFile 属性指定

18.2.4 写入事件日志

要写入事件日志项，可以使用 `EventLog` 类的 `WriteEntry()`或 `WriteEvent()`方法。

`EventLog` 类有一个静态方法 `WriteEntry()` 和一个实例方法 `WriteEntry()`。静态方法 `WriteEntry()`的参数是事件源。该源也可以用 `EventLog` 类的构造函数设置。在下面的构造函数中，定义了日志名、本地机器和事件源名。接着将信息作为 `WriteEntry()`的第一个参数，写入三个事件日志项。`WriteEntry()`是重载版本。第二个参数是 `EventLogEntryType` 类型的枚举。使用 `EventLogEntryType`，可以指定事件日志项的严重级别。其值可以是 `Information`、`Warning`、`Error`、用于审计的 `FailureAudit` 和 `SuccessAudit`。根据该类型，事件查看器会显示不同的图标。第三个参数指定与应用程序相关的事件 ID，它可以由应用程序使用。另外，还可以传送与应用程序相关的二进制数据和类别。

```
using (EventLog log = new EventLog("ProCSharpLog", ".", "EventLogDemoApp"))
{
    log.WriteEntry("Message 1");
    log.WriteEntry("Message 2", EventLogEntryType.Warning);
    log.WriteEntry("Message 3", EventLogEntryType.Information, 33);
}
```

18.2.5 资源文件

除了在 C#代码中为事件日志定义信息，把它传送给 `WriteEntry()`方法之外，还可以创建信息资源文件，在该资源文件中定义信息，将信息的标识符传送给 `WriteEvent()`方法。资源文件

还支持本地化。

**提示：**

信息资源文件是内置的资源文件，它与 .NET 资源文件没有共同之处。 .NET 资源文件详见第 21 章。

资源文件是一个文本文件，扩展名是 .mc。这个文件用于定义信息的语法是非常严格的。示例文件 EventLogMessages.mc 包含四个类别，之后是事件信息。每个信息都有一个 ID，它可以由写入日志项的应用程序使用。可以从应用程序中传送出来的参数在信息文本中用 % 语法定义。

**提示：**

信息文件的语法可参见 MSDN 文档中的“信息文本文件”。

```
; // EventLogDemoMessages.mc
; // *****
; // - Event categories -
; // Categories must be numbered consecutively starting at 1.
; // *****

MessageId=0x1
Severity=Success
SymbolicName=INSTALL_CATEGORY
Language=English
Installation
.

MessageId=0x2
Severity=Success
SymbolicName=DATA_CATEGORY
Language=English
Database Query
.

MessageId=0x3
Severity=Success
SymbolicName=UPDATE_CATEGORY
Language=English
Data Update
.

MessageId=0x4
Severity=Success
SymbolicName=NETWORK_CATEGORY
Language=English
Network Communication
.

; // - Event messages -
; // *****
MessageId = 1000
Severity = Success
Facility = Application
SymbolicName = MSG_CONNECT_1000
Language=English
Connection successful.
.

MessageId = 1001
```

```

Severity = Error
Facility = Application
SymbolicName = MSG_CONNECT_FAILED_1001
Language=English
Could not connect to server %1.
.

MessageId = 1002
Severity = Error
Facility = Application
SymbolicName = MSG_DB_UPDATE_1002
Language=English
Database update failed.
.

MessageId = 1003
Severity = Success
Facility = Application
SymbolicName = APP_UPDATE
Language=English
Application %%5002 updated.
.

; // - Event log display name -
; // *****

MessageId = 5001
Severity = Success
Facility = Application
SymbolicName = EVENT_LOG_DISPLAY_NAME_MSGID
Language=English
Professional C# Sample Event Log
.

; // - Event message parameters -
; // Language independent insertion strings
; // *****

MessageId = 5002
Severity = Success
Facility = Application
SymbolicName = EVENT_LOG_SERVICE_NAME_MSGID
Language=English
EventLogDemo.EXE
.

```

使用信息编译器 `mc.exe`，创建一个二进制的信息文件。`mc -s EventLogDemo- Messages.mc` 会把包含信息的源文件编译为一个扩展名为 `.bin` 的信息文件和文件 `Messages.rc`，它包含对二进制信息文件的引用。

```
mc -s EventLogMessages.mc
```

接着，需要使用资源编译器 `rc.exe`。`rc EventLogDemoMessages.rc` 会创建资源文件 `Messages.res`：

```
rc EventLogMessages.rc
```

使用链接器，可以把二进制信息文件 `EventLogDemoMessages.Res` 绑定到一个内置的 DLL 上。

```
link /DLL /SUBSYSTEM:WINDOWS /NOENTRY /MACHINE:x86 EventLogDemoMessages.Res
```

现在，就可以用下面的代码注册一个事件源，来定义资源文件了。首先检查事件源 `EventLogDemoApp` 是否存在。如果它不存在，就必须创建事件日志。接着检查资源文件是否可



用。MSDN 文档中的一些示例演示了如何把信息文件写入 <windows>\system32 目录, 但这里不应这么做。把信息 DLL 复制到与程序相关的目录中, 该目录可以使用 `SpecialFolder` 枚举的值 `ProgramFiles` 指定。如果需要在多个应用程序中共享信息文件, 就可以把它放在 `Environment.SpecialFolder.CommonProgramFiles` 中。如果该文件存在, 就实例化类型 `EventSourceCreationData` 的一个新对象。在构造函数中, 定义了源的名称和日志的名称。使用属性 `CategoryResourceFile`、`MessageResourceFile` 和 `ParameterResourceFile` 定义资源文件的引用。在创建了事件源后, 就可以使用事件源在注册表中找到资源文件的信息。方法 `CreateEventSource` 注册新的事件源和日志文件。最后, `EventLog` 类的方法 `RegisterDisplayName()` 指定日志显示在事件查看器中的名称。从信息文件中提取 ID 5001。

#### 提示:

如果要删除以前创建的事件源, 可以使用 `EventLog.DeleteEventSource(sourceName)`; 要删除日志, 可以调用 `EventLog.Delete(logName)`。

```
string logName = "ProCSharpLog";
string sourceName = "EventLogDemoApp";
string resourceFile = Environment.GetFolderPath(
    Environment.SpecialFolder.ProgramFiles) +
    @"\procsharp\EventLogDemoMessages.dll";

if (!EventLog.SourceExists(sourceName))
{
    if (!File.Exists(resourceFile))
    {
        Console.WriteLine("Message resource file does not exist");
        return;
    }

    EventSourceCreationData eventSource =
        new EventSourceCreationData(sourceName, logName);
    eventSource.CategoryResourceFile = resourceFile;
    eventSource.CategoryCount = 4;
    eventSource.MessageResourceFile = resourceFile;
    eventSource.ParameterResourceFile = resourceFile;

    EventLog.CreateEventSource(eventSource);
}
else
{
    logName = EventLog.LogNameFromSourceName(sourceName, ".");
}

EventLog evLog = new EventLog(logName, ".", sourceName);
evLog.RegisterDisplayName(resourceFile, 5001);
```

现在, 可以使用 `WriteEvent()` 方法替代 `WriteEntry()` 写入事件日志项。 `WriteEvent()` 需要把一个 `EventInstance` 类型的对象作为参数。使用 `EventInstance` 可以指定信息 ID、类别和 `EventLogEntryType` 类型的严重级别。除 `EventInstance` 之外, `WriteEvent()` 的参数还允许将二进制数据指定为字节数组。

```
EventLog log = new EventLog(logName, ".", sourceName);
EventInstance info1 = new EventInstance(1000, 4,
    EventLogEntryType.Information);
```



```

log.WriteEvent(info1);
EventInstance info2 = new EventInstance(1001, 4, EventLogEntryType.Error);
log.WriteEvent(info2, "avalon");

EventInstance info3 = new EventInstance(1002, 3, EventLogEntryType.Error);
byte[] additionalInfo = { 1, 2, 3 };
log.WriteEvent(info3, additionalInfo);

log.Dispose();

```

**提示:**

信息标识符可以定义带常量值的类，在应用程序中为标识符提供更有意义的名称。

使用事件查看器可以读取事件日志项。

**事件日志监听器**

除了使用事件查看器读取事件日志项之外，还可以创建定制的事件日志读取器，根据需要来监听指定类型的事件。可以创建一个读取器，将重要的信息显示在屏幕上，或把 SMS 发送给系统管理员。

接着，编写一个应用程序，在服务程序遇到问题时接收一个事件。创建一个简单的 Windows 应用程序，监听 Quote 服务的事件。这个 Windows 应用程序只包含一个列表框和一个 Exit 按钮，如图 18-5 所示。



图 18-5

把一个 EventLog 组件从工具箱拖放到设计视图上，将 Log 属性设置为 Application。可以把 Source 属性设置为特定的源，只接收这个源中的事件日志项，例如源 EventLogDemoApp 只从前面创建的应用程序中接收事件日志。如果使 Source 属性为空，就可以接收所有源的事件。还需要修改属性 EnableRaisingEvents，其默认值是 false，把它设置为 true，表示每次发生这个事件时，都接收它。还可以为 EventLog 类的 EntryWritten 事件添加一个事件处理程序。为这个事件添加处理程序 OnEntryWritten()。

处理程序 OnEntryWritten() 将对象 EntryWrittenEventArgs 接收为变元，在该变元中可以获得事件的完整信息。使用 Entry 属性返回一个 EventLogEntry 对象和时间、事件源、类型、类别等信息：

```

protected void OnEntryWritten (object sender,
    System.Diagnostics.EntryWrittenEventArgs e)
{
    StringBuilder sb = new StringBuilder();
    sb.AppendFormat("{0} {1} {2}",
        e.Entry.TimeGenerated.ToShortTimeString(),
        e.Entry.Source,
        e.Entry.Message);
    listBoxEvents.Items.Add(sb.ToString());
}

```

该应用程序显示了如图 18-6 所示的事件日志信息：

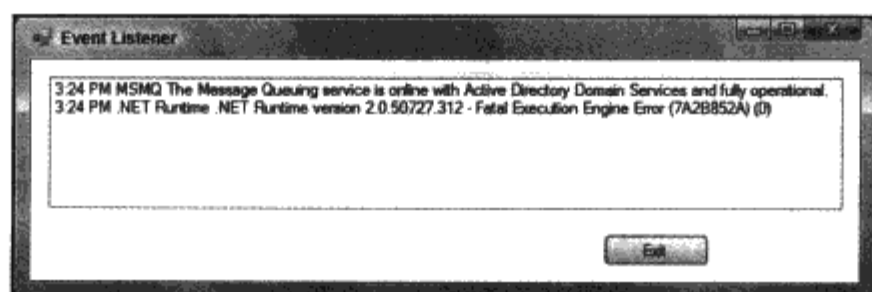


图 18-6

## 18.3 性能监控

性能监控可以用于获取应用程序的一般信息。性能监控是一个强大的工具，有助于理解系统的工作负载，观察变化和趋势，尤其是运行在服务器上的应用程序。

Microsoft Windows 有许多性能对象，例如 System、Memory、Objects、Process、Processor、Thread、Cache 等。这些对象有许多要监控的计数。例如，使用 Process 对象可以监控所有进程或特定进程实例的用户时间、句柄数、页面错误、线程数等。一些应用程序，如 SQL Server，还添加了与应用程序相关的对象。

对于 quote 服务程序，获取客户请求数、在线上传送的数据大小等是比较有趣的。

### 18.3.1 性能监控类

System.Diagnostics 命名空间为性能监控提供了如下类：

- PerformanceCounter 可以用于监控计数和写入计数。还可以使用这个类创建新的性能类别。
- PerformanceCounterCategory 可以查看所有已有的类别，创建新类别。可以以编程方式获得一个类别中的所有计数器。
- PerformanceCounterInstaller 用于安装性能计数器，它的用法类似于前面讨论的 EventLogInstaller。

### 18.3.2 性能计数器的构建

示例程序是一个简单的 Windows 应用程序，它只有一个按钮，用于说明如何编写性能计数。同样，可以在 Windows 服务(参见第 23 章)、网络应用程序(参见第 41 章)和其他要接收实时计数的应用程序中添加性能计数器。

使用 Visual Studio，可以创建新的性能计数器类别，具体方法是在 Server Explorer 中选择性能计数器，在弹出的菜单中选择菜单项 Create New Category，这会启动 Performance Counter Builder，如图 18-7 所示。

将性能计数器类别的名称设置为 Wrox Performance Counters。表 18-5 列出了 quote 服务的所有性能计数器。

Performance Counter Builder 将配置写入性能数据库。这也可以用 System.Diagnostics 命名空间的 PerformanceCounterCategory 类中的 Create()方法动态完成。使用 Visual Studio 很容易在以后添加其他系统的安装程序。

表 18-5

名 称	说 明	类 型
按钮单击#	按钮单击的总次数#	NumberOfItems32
按钮单击/秒#	一秒内的按钮单击次数#	RateOfCountsPerSecond32
鼠标移动事件#	鼠标移动事件的总数#	NumberOfItems32
鼠标移动事件/秒#	一秒内鼠标移动事件的总数#	RateOfCountsPerSecond32

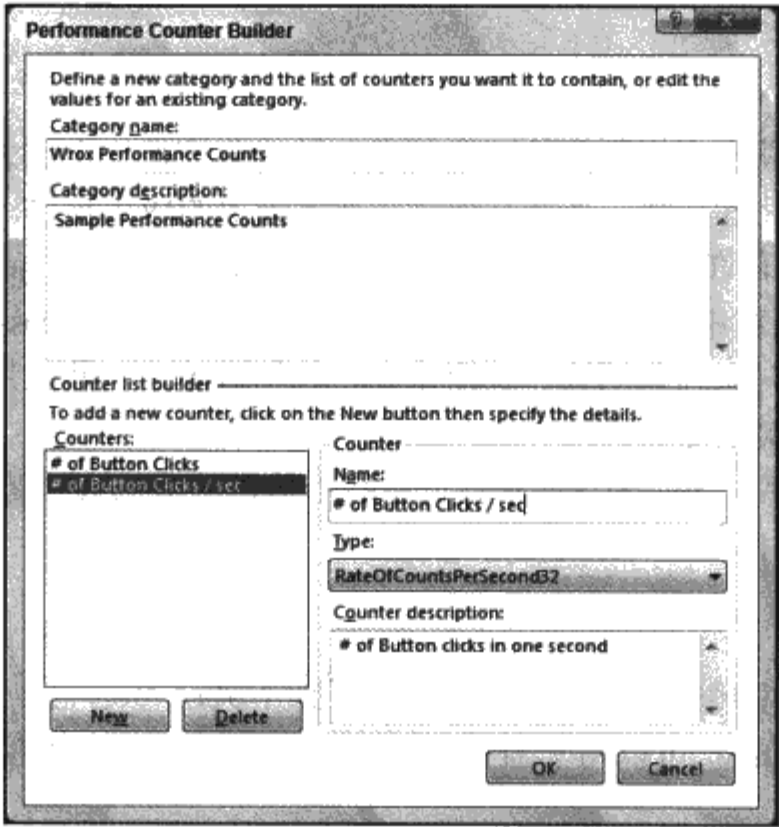


图 18-7

18.3.3 添加 PerformanceCounter 组件

现在可以从工具箱中添加 PerformanceCounter 组件了。除了使用工具箱 Components 类别中的组件之外，还可以直接把前面创建的性能计数器从 Server Explorer 拖放到设计视图上。这样，实例是自动配置的，所有对象的 CategoryName 属性都设置为 Wrox Performance Counts，CounterName 属性设置为所选类别中的一个值。在这个应用程序中，不是读取性能计数，而是写入它，所以必须把 ReadOnly 属性设置为 false。还要把 MachineName 属性设置为.，使应用程序在本地写入性能计数。

把 PerformanceCounter 组件添加到设计器中，按照上述方式设置属性后，会在 InitializeComponent()中生成一些代码，如下所示：

```
private void InitializeComponent()
{
    //...
    //
    // performanceCounterButtonClicks
    //
    this.performanceCounterButtonClicks.CategoryName = "Wrox Performance Counts";
```

```

        this.performanceCounterButtonClicks.CounterName = "# of Button Clicks";
        this.performanceCounterButtonClicks.ReadOnly = false;
        //
        // performanceCounterButtonClicksPerSec
        //
        this.performanceCounterButtonClicksPerSec.CategoryName =
            "Wrox Performance Counts";
        this.performanceCounterButtonClicksPerSec.CounterName =
            "# of Button Clicks / sec";
        this.performanceCounterButtonClicksPerSec.ReadOnly = false;
        //
        // performanceCounterMouseMoveEvents
        //
        this.performanceCounterMouseMoveEvents.CategoryName =
            "Wrox Performance Counts";
        this.performanceCounterMouseMoveEvents.CounterName =
            "# of Mouse Move Events";
        this.performanceCounterMouseMoveEvents.ReadOnly = false;
        //
        // performanceCounterMouseMoveEventsPerSec
        //
        this.performanceCounterMouseMoveEventsPerSec.CategoryName =
            "Wrox Performance Counts";
        this.performanceCounterMouseMoveEventsPerSec.CounterName =
            "# of Mouse Move Events / sec";
        this.performanceCounterMouseMoveEventsPerSec.ReadOnly = false;
        //...
    }

```

为了计算性能值，需要在类 `Form1` 中添加字段 `clickCountPerSec` 和 `mouseMoveCountPerSec`：

```

public partial class Form1 : Form
{
    // Performance monitoring counter values
    private int clickCountPerSec = 0;
    private int mouseMoveCountPerSec = 0;

```

给按钮的 `Click` 事件添加事件处理程序，给 `MouseMove` 事件添加事件处理程序，在处理程序中添加如下代码：

```

private void button1_Click(object sender, EventArgs e)
{
    performanceCounterButtonClicks.Increment();
    clickCountPerSec++;
}

private void OnMouseMove(object sender, MouseEventArgs e)
{
    performanceCounterMouseMoveEvents.Increment();
    mouseMoveCountPerSec++;
}

```

`PerformanceCounter` 对象的 `Increment()` 方法给计数器递增 1。如果需要给计数器添加其他信息，例如添加一个字节的收发计数，就可以使用 `IncrementBy()` 方法。对于显示以秒为单位的值的性能计数，只需递增两个变量 `clickCountPerSec` 和 `mouseMovePerSec`。

为了显示每秒更新的值，可添加一个 `Timer` 组件。把 `OnTimer()` 方法设置为这个组件的 `Elapsed` 事件。如果把 `Interval` 属性设置为 1000，则每秒调用一次 `OnTimer()` 方法。在这个方法

的实现代码中，使用 PerformanceCounter 类的 RawValue 属性设置性能计数。

```
protected void OnTimer (object sender, System.Timers.ElapsedEventArgs e)
{
    performanceCounterButtonClicksPerSec.RawValue = clickCountPerSec;
    clickCountPerSec = 0;

    performanceCounterMouseMoveEventsPerSec.RawValue = mouseMoveCountPerSec;
    mouseMoveCountPerSec = 0;
}
```

必须启动计时器：

```
public Form1()
{
    InitializeComponent();

    this.timer1.Start();
}
```

### 18.3.4 perfmon.exe

现在就可以监控应用程序了。在 Windows XP 中选择“管理工具”|“性能”，在 Windows Vista 中选择“管理工具”|“可靠性和性能监控器”，就可以启动“性能”工具。选择“性能监控器”，单击工具栏上的+按钮，可以添加性能计数。Quote 服务显示为一个性能对象。所有已配置的计数器都显示在计数器列表中，如图 18-8 所示。

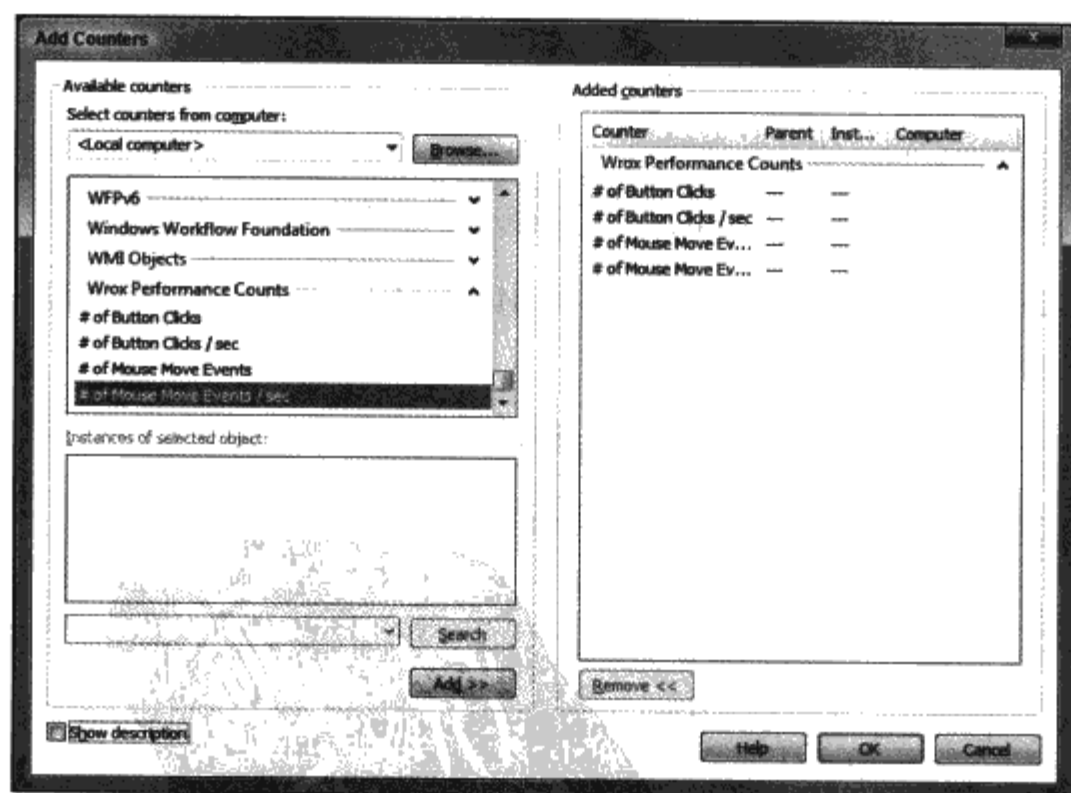


图 18-8

在性能监控器中添加了计数器后，就可以查看服务随时间变化的值，如图 18-9 所示。使用这个性能工具，还可以创建日志文件，在以后分析性能。



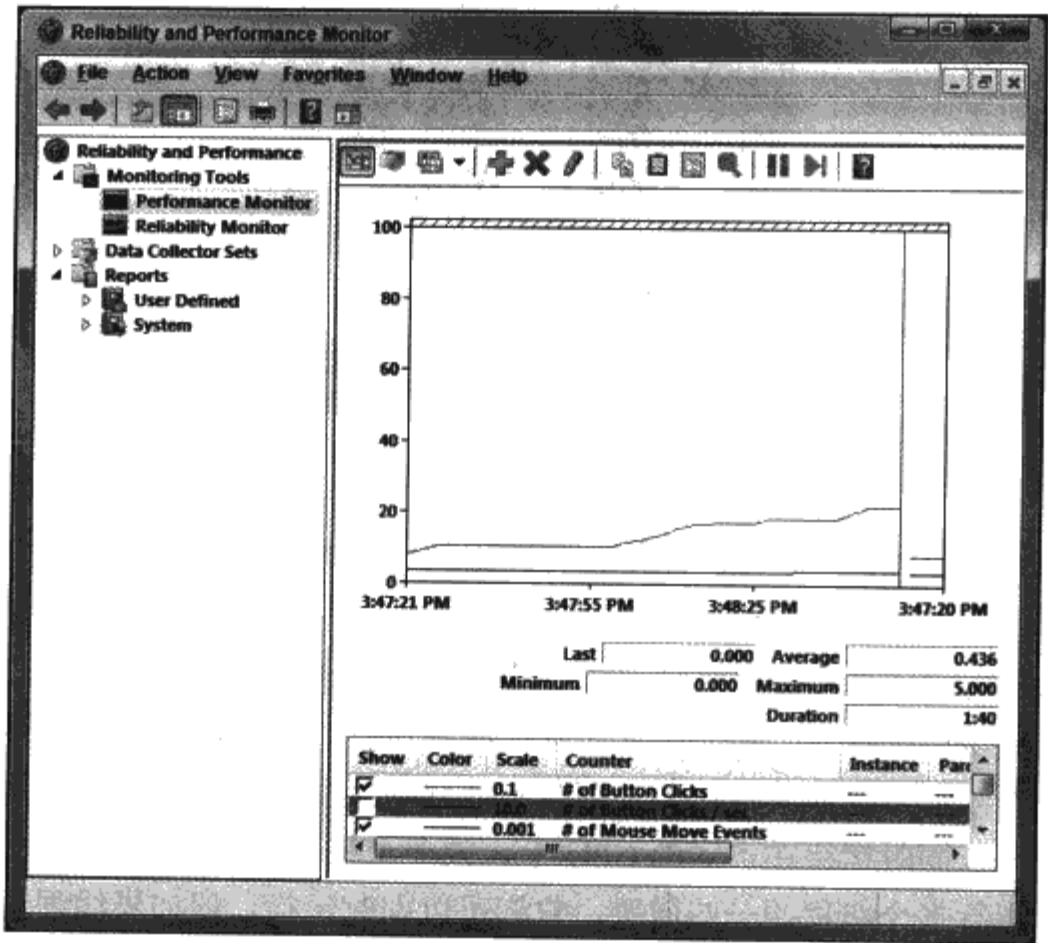


图 18-9

18.4 小结

本章介绍了跟踪和日志功能，它们有助于找出应用程序中的问题。应尽早规划，把这些功能内置于应用程序中。这可以避免以后的许多麻烦。

使用跟踪功能，可以把调试信息写入应用程序，也可以用于最终发布的产品。如果出了问题，可以修改配置值，打开跟踪功能，找出问题。

事件日志为系统管理员提供信息，帮助找出应用程序的某些严重问题。性能监控有助于分析应用程序的负载，提前规划将来可能需要的资源。

下一章学习如何编写多线程的应用程序。

# 第19章

## 线程和同步

使用线程有几个原因。在应用程序中进行网络调用需要一定的时间。用户不希望在安装用户界面时只是等待，直到服务器返回一个响应为止。用户可以在这个过程中执行其他一些操作，甚至取消发送给服务器的请求。这些都可以使用线程来实现。

对于所有需要等待的操作，例如文件、数据库或网络访问的启动都需要一定的时间，此时就可以启动一个新线程，完成其他任务。即使是处理密集型的任务，线程也是有帮助的。一个进程的多个线程可以同时运行在不同的 CPU 上，或多个核心 CPU 的不同核心上。

还必须注意运行多个线程的一些问题。它们可以同时运行，但如果线程访问相同的数据，就很容易出问题。必须实现同步机制。

本章介绍用多个线程编写应用程序所需了解的知识，包括：

- 线程概述
- 使用委托的轻型线程
- 线程类
- 线程池
- 线程问题
- 同步技术
- 计时器
- COM 空间
- 基于事件的异步模式

### 19.1 概述

线程是程序中独立的指令流。使用 C# 编写任何程序时，都有一个入口：Main() 方法。程序从 Main() 方法的第一条语句开始执行，直到这个方法返回为止。

这个程序结构非常适合于有一个可识别的任务序列的程序，但程序常常需要同时完成多个任务。线程对客户端和服务端应用程序都非常重要。在 Visual Studio 编辑器中输入 C# 代码时，Dynamic Help 窗口会立即显示与所输入代码相关的主题。后台线程会搜索帮助。Microsoft Word 的拼写检查器也会做相同的事。一个线程等待用户输入，另一个线程进行后台搜索。第三个线程将写入的数据存储在临时文件中，第四个线程从 Internet 上下载其他数据。

运行在服务器上的应用程序中，一个线程等待客户的请求，称为监听器线程。只要接收到

请求，就把它传送给另一个工作线程，之后继续与客户通信。监听器线程会立即返回，接收下一个客户发送的下一个请求。

使用 Windows 任务管理器，可以从菜单 View | Select Columns 中打开 Threads 列，查看进程和每个进程的线程号。在图 19-1 中，只有 cmd.exe 运行在一个线程中，其他应用程序都使用多个线程。Internet Explorer 运行了 51 个线程。

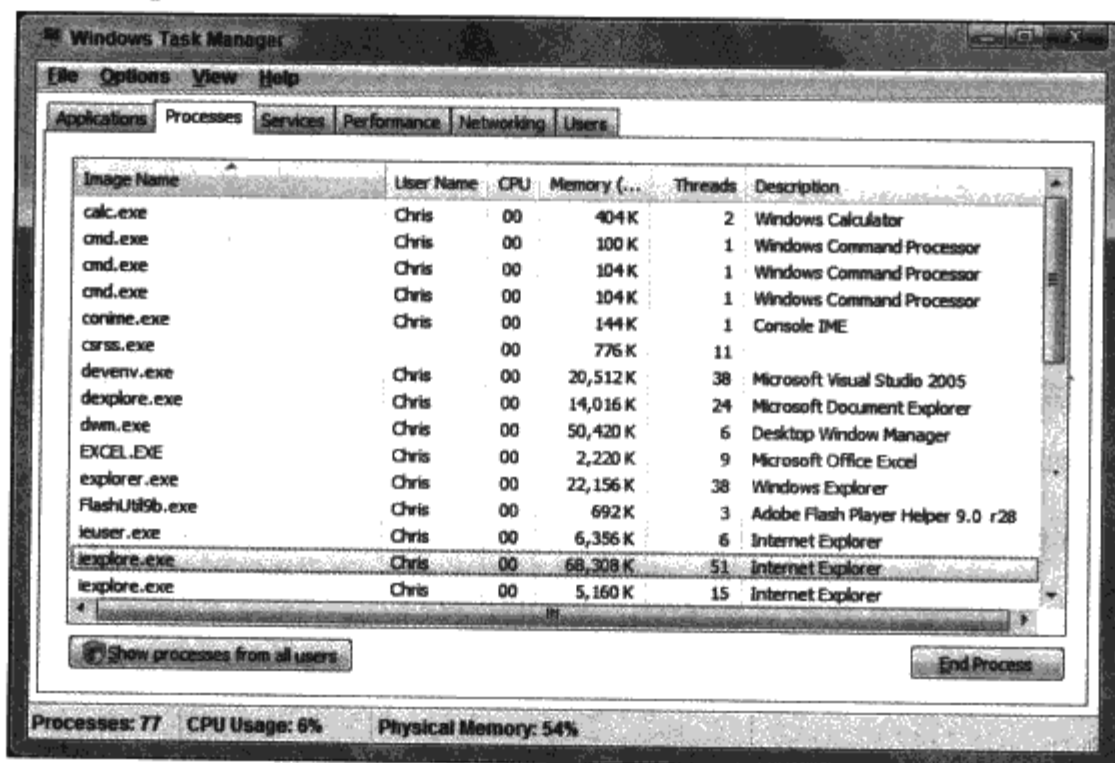


图 19-1

进程包含资源，例如 Window 句柄、文件系统句柄或其他核心对象。每个进程都分配了虚拟内存。一个进程至少包含一个线程。操作系统会调度线程。线程有一个优先级、正在处理的程序的位置计数器、一个存储其本地变量的堆栈。每个线程都有自己的堆栈，但程序代码的内存和堆由一个进程的所有线程共享。这使一个进程中的所有线程之间的通信非常快——该进程的所有线程都寻址相同的虚拟内存。但是，这也使处理比较困难，因为多个线程可以修改同一个内存位置。

进程管理的资源包括虚拟内存和 Window 句柄，其中至少包含一个线程。线程是运行程序所必需的。

在 .NET 中，托管的线程由 Thread 类定义。托管的线程不一定映射为一个操作系统线程。尽管这种情况可能出现，但应由 .NET 运行库负责将托管的线程映射到操作系统的物理线程上。在这方面，SQL Server 2005 的运行主机与 Windows 应用程序的运行主机完全不同。使用 ProcessThread 类可以获得内部线程的信息，但在托管的应用程序中，通常最好使用托管的线程。

## 19.2 异步委托

创建线程的一种简单方式是定义一个委托，异步调用它。第 7 章提到，委托是方法的类型安全的引用。Delegate 类还支持异步调用方法。在后台，Delegate 类会创建一个执行任务的线程。

提示:

委托使用线程池来完成异步任务。线程池详见本章后面的内容。

为了演示委托的异步特性,启动一个方法,它需要一定的时间才能执行完毕。方法 `TakesAWhile` 至少需要作为变元传送过来的毫秒数才能执行完,因为它调用了 `Thread.Sleep()` 方法:

```
static int TakesAWhile(int data, int ms)
{
    Console.WriteLine("TakesAWhile started");
    Thread.Sleep(ms);
    Console.WriteLine("TakesAWhile completed");
    return ++data;
}
```

要在委托中调用这个方法,必须定义一个有相同参数和返回类型的委托,如下面的 `TakesAWhileDelegate` 所示:

```
public delegate int TakesAWhileDelegate(int data, int ms);
```

现在可以使用不同的技术异步调用委托,返回结果。

### 19.2.1 投票

一种技术是投票,检查委托是否完成了任务。所创建的 `Delegate` 类提供了方法 `BeginInvoke()`,在该方法中,可以传送用委托类型定义的输入参数。`BeginInvoke()`方法总是有两个 `AsyncCallback` 和 `Object` 类型的额外参数(稍后讨论)。现在重要的是 `BeginInvoke()`方法的返回类型: `IAsyncResult`。在 `IAsyncResult` 中,可以获得委托的信息,并验证委托是否完成了任务,这是 `IsCompleted` 属性的功劳。只要委托没有完成其任务,程序的主线程就继续执行 `while` 循环。

```
static void Main()
{
    // synchronous method call
    // TakesAWhile(1, 3000);

    // asynchronous by using a delegate
    TakesAWhileDelegate d1 = TakesAWhile;

    IAsyncResult ar = d1.BeginInvoke(1, 3000, null, null);
    while (!ar.IsCompleted)
    {
        // doing something else in the main thread
        Console.WriteLine(".");
        Thread.Sleep(50);
    }
    int result = d1.EndInvoke(ar);
    Console.WriteLine("result: {0}", result);
}
```

运行应用程序,可以看到主线程和委托线程同时运行,在委托线程执行完毕后,主线程就停止循环。

```
.TakesAWhile started
.....TakesAWhile completed
result: 2
```

除了检查委托是否完成之外，还可以在完成了由主线程执行的工作后，调用委托类型的 `EndInvoke()` 方法。`EndInvoke()` 方法会一直等待，直到委托完成其任务为止。

#### 警告：

如果不等待委托完成其任务就结束主线程，委托线程就会停止。

### 19.2.2 等待句柄

等待异步委托的结果的另一种方式是使用与 `IAsyncResult` 相关的等待句柄。使用 `AsyncWaitHandle` 属性可以访问等待句柄。这个属性返回一个 `WaitHandle` 类型的对象，它可以等待委托线程完成其任务。方法 `WaitOne()` 将一个超时时间作为可选的第一个参数，在其中可以定义要等待的最大时间。这里设置为 50 毫秒。如果发生超时，`WaitOne()` 就返回 `false`，`while` 循环会继续执行。如果等待操作成功，就用一个中断退出 `while` 循环，用委托的 `EndInvoke()` 方法接收结果。

```
static void Main()
{
    TakesAWhileDelegate dl = TakesAWhile;
    IAsyncResult ar = dl.BeginInvoke(1, 3000, null, null);
    while (true)
    {
        Console.WriteLine(".");
        if (ar.AsyncWaitHandle.WaitOne(50, false))
        {
            Console.WriteLine("Can get the result now");
            break;
        }
    }
    int result = dl.EndInvoke(ar);
    Console.WriteLine("result: {0}", result);
}
```

#### 提示：

等待句柄的内容详见本章后面的“同步”一节。

### 19.2.3 异步回调

等待委托的结果的第三种方式是使用异步回调。在 `BeginInvoke()` 方法的第三个参数中，可以传送一个满足 `AsyncCallback` 委托的需求的方法。`AsyncCallback` 委托定义了一个 `IAsyncResult` 类型的参数，其返回类型是 `void`。这里，把方法 `TakesAWhileCompleted` 的地址赋予第三个参数，以满足 `AsyncCallback` 委托的需求。对于最后一个参数，可以传送任意对象，以便从回调方法中访问它。传送委托实例是可行的，这样回调方法就可以使用它获得异步方法的结果。



现在, 只要委托 `TakesAWhileDelegate` 完成了其任务, 就调用 `TakesAWhileCompleted()` 方法。不需要在主线程中等待结果。但是在委托线程的任务未完成之前, 不能停止主线程, 除非刚刚停止的委托线程没有出问题。

```
static void Main()
{
    TakesAWhileDelegate d1 = TakesAWhile;

    d1.BeginInvoke(1, 3000, TakesAWhileCompleted, d1);
    for (int i = 0; i < 100; i++)
    {
        Console.Write(".");
        Thread.Sleep(50);
    }
}
```

方法 `TakesAWhileCompleted()` 用 `AsyncCallback` 委托指定的参数和返回类型来定义。用 `BeginInvoke()` 方法传送的最后一个参数可以使用 `ar.AsyncState` 读取。在 `TakesAWhileDelegate` 委托中, 可以调用 `EndInvoke()` 方法获得结果。

```
static void TakesAWhileCompleted(IAsyncResult ar)
{
    if (ar == null) throw new ArgumentNullException("ar");

    TakesAWhileDelegate d1 = ar.AsyncState as TakesAWhileDelegate;
    Trace.Assert(d1 != null, "Invalid object type");

    int result = d1.EndInvoke(ar);
    Console.WriteLine("result: {0}", result);
}
```

#### 警告:

使用回调方法, 必须注意这个方法在委托线程中调用, 而不是在主线程中调用。

除了定义一个单独的方法, 给它传送 `BeginInvoke()` 方法之外,  $\lambda$  表达式也非常适合这种情况。`ar` 参数是 `IAsyncResult` 类型。在执行代码中, 不需要把一个值赋予 `BeginInvoke()` 方法的最后一个参数, 因为  $\lambda$  表达式可以直接访问该方法外部的变量 `d1`。但是,  $\lambda$  表达式的执行代码仍是在委托线程中调用, 以这种方式定义方法时, 这不是很明显。

```
static void Main()
{
    TakesAWhileDelegate d1 = TakesAWhile;

    d1.BeginInvoke(1, 3000,
        ar=>
        {
            int result = d1.EndInvoke(ar);
            Console.WriteLine("result: {0}", result);
        },
        null);
    for (int i = 0; i < 100; i++)
    {
        Console.Write(".");
        Thread.Sleep(50);
    }
}
```

提示:

只有代码不太多, 且实现代码不需要用于不同的地方时, 才应使用 $\lambda$ 表达式。在这种情况下, 定义一个单独的方法比较好。 $\lambda$ 表达式详见第 7 章。

编程模型和所有这些利用异步委托的选项——投票、等待句柄和异步调用——不仅能用于委托, 编程模型(即异步模式)在 .NET Framework 的各个地方都能见到。例如, 可以用 `HttpRequest` 类的 `BeginGetResponse()` 方法异步发送 HTTP Web 请求, 使用 `SqlCommand` 类的 `BeginExecute-Reader()` 方法给数据库发送异步请求。其参数类似于委托的 `BeginInvoke()` 方法, 也可以使用相同的方式获得结果。

提示:

`HttpRequest` 参见第 41 章, `SqlCommand` 参见第 26 章。

除了使用委托创建线程之外, 还可以使用 `Thread` 类创建线程, 如下一节所述。

## 19.3 Thread 类

使用 `Thread` 类可以创建和控制线程。下面的代码是创建和启动一个新线程的简单例子。`Thread` 类的构造函数接受 `ThreadStart` 和 `ParameterizedThreadStart` 类型的委托参数。`ThreadStart` 委托定义了一个返回类型为 `void` 的无参数方法。在创建了 `Thread` 对象后, 就可以用 `Start()` 方法启动线程了:

```
using System;
using System.Threading;

namespace Wrox.ProCSharp.Threading
{
    class Program
    {
        static void Main()
        {
            Thread t1 = new Thread(ThreadMain);
            t1.Start();
            Console.WriteLine("This is the main thread.");
        }

        static void ThreadMain()
        {
            Console.WriteLine("Running in a thread.");
        }
    }
}
```

运行这个程序, 得到两个线程的输出:

```
This is the main thread.
Running in a thread.
```

不能保证哪个结果先输出。线程由操作系统调度, 每次哪个线程在前面都是不同的。前面探讨了 $\lambda$ 表达式如何与异步委托一起使用。异步委托还可以与 `Thread` 类一起使用, 将

线程方法的实现代码传送给 Thread 构造函数的变元:

```
using System;
using System.Threading;

namespace Wrox.ProCSharp.Threading
{
    class Program
    {
        static void Main()
        {
            Thread t1 = new Thread(()=>
                Console.WriteLine("running in a thread"));
            t1.Start();
            Console.WriteLine("This is the main thread.");
        }
    }
}
```

在创建好线程后, 如果不需要用引用线程的变量来控制线程, 还可以用更简洁的方式编写代码。用构造函数创建一个新的 Thread 对象, 将λ表达式传送给构造函数, 用返回的 Thread 对象直接调用 Start()方法:

```
using System.Threading;

namespace Wrox.ProCSharp.Threading
{
    class Program
    {
        static void Main()
        {
            new Thread(()=>
                Console.WriteLine("running in a thread");
            ).Start();
            Console.WriteLine("This is the main thread.");
        }
    }
}
```

但是, 采用一个引用 Thread 对象的变量是有原因的。例如, 为了更好地控制线程, 可以在启动线程前, 设置 Name 属性, 给线程指定名称。为了获得当前线程的名称, 可以使用静态属性 Thread.CurrentThread, 获取当前线程的 Thread 实例, 访问 Name 属性, 进行读取访问。线程也有一个托管的线程 ID, 可以用 ManagedThreadId 属性读取它。

```
static void Main()
{
    Thread t1 = new Thread(ThreadMain);
    t1.Name = "MyNewThread1";
    t1.Start();
    Console.WriteLine("This is the main thread.");
}

static void ThreadMain()
{
    Console.WriteLine("Running in the thread {0}, id: {1}.",
        Thread.CurrentThread.Name, Thread.CurrentThread.ManagedThreadId);
}
```

在应用程序的输出中, 现在还可以看到线程名和 ID:

```
This is the main thread.
Running in the thread MyNewThread1, id: 3.
```

**警告:**

给线程指定名称，非常有助于调试线程。在 Visual Studio 的调试会话中，可以打开 Debug Location 工具栏，查看线程的名称。

**19.3.1 给线程传送数据**

如果需要给线程传送一些数据，可以采用两种方式。一种方式是使用带 `ParameterizedThreadStart` 委托参数的 `Thread` 构造函数，另一种方式是创建一个定制类，把线程的方法定义为实例方法，这样就可以初始化实例的数据，之后启动线程。

要给线程传送数据，需要某个存储数据的类或结构。这里定义了包含字符串的结构 `Data`，也可以传送任意对象。

```
public struct Data
{
    public string Message;
}
```

如果使用了 `ParameterizedThreadStart` 委托，线程的入口点必须有一个 `object` 类型的参数，返回类型为 `void`。对象可以转换为数据，这里是把信息写入控制台。

```
static void ThreadMainWithParameters(object o)
{
    Data d = (Data)o;
    Console.WriteLine("Running in a thread, received {0}", d.Message);
}
```

在 `Thread` 类的构造函数中，可以将新的入口点赋予 `ThreadMainWithParameters`，传送变量 `d`，调用 `Start()` 方法。

```
static void Main()
{
    Data d = new Data();
    d.Message = "Info";
    Thread t2 = new Thread(ThreadMainWithParameters);
    t2.Start(d);
}
```

给新线程传送数据的另一种方式是定义一个类（参见类 `MyThread`），在其中定义需要的字段，将线程的主方法定义为类的一个实例方法：

```
public class MyThread
{
    private string data;

    public MyThread(string data)
    {
        this.data = data;
    }

    public void ThreadMain()
    {

```



```

        Console.WriteLine("Running in a thread, data: {0}", data);
    }
}

```

这样,就可以创建 `MyThread` 的一个对象,给 `Thread` 类的构造函数传送对象和 `Thread Main()` 方法。线程可以访问数据。

```

MyThread obj = new MyThread("info");
Thread t3 = new Thread(obj.ThreadMain);
t3.Start();

```

### 19.3.2 后台线程

只要有一个前台线程在运行,应用程序的进程就在运行。如果多个前台线程在运行,而 `Main` 方法结束了,应用程序的进程就是激活的,直到所有前台线程完成其任务为止。

在默认情况下,用 `Thread` 类创建的线程是前台线程。线程池中的线程总是后台线程。

在用 `Thread` 类创建线程时,可以设置属性 `IsBackground`,以确定该线程是前台线程还是后台线程。`Main()`方法将线程 `t1` 的 `IsBackground` 属性设置为 `false`(默认值)。在启动新线程后,主线程就把结束信息写入控制台。新线程会写入启动和结束信息,在这个过程中它要睡眠 3 秒。在这 3 秒中,新线程会完成其工作,主线程才结束。

```

class Program
{
    static void Main()
    {
        Thread t1 = new Thread(ThreadMain);
        t1.Name = "MyNewThread1";
        t1.IsBackground = false;
        t1.Start();
        Console.WriteLine("Main thread ending now...");
    }

    static void ThreadMain()
    {
        Console.WriteLine("Thread {0} started", Thread.CurrentThread.Name);
        Thread.Sleep(3000);
        Console.WriteLine("Thread {0} completed", Thread.CurrentThread.Name);
    }
}

```

在启动应用程序时,会看到写入控制台的完成信息,尽管主线程会早一步完成其工作。原因是新线程也是一个前台线程。

```

Main thread ending now...
Thread MyNewThread1 started
Thread MyNewThread1 completed

```

如果将启动新线程的 `IsBackground` 属性改为 `true`,显示在控制台上的结果就会不同。在一个系统上,可以看到新线程的启动信息,但没有结束信息。如果线程没有正常结束,还有可能看不到启动信息。

```

Main thread ending now...
Thread MyNewThread1 started

```



后台线程非常适合于完成后台任务。例如，如果关闭 Word 应用程序，拼写检查器继续运行其进程就没有意义了。在应用程序结束时，拼写检查器线程就可以关闭了。但是，组织 Outlook 信息库的线程应一直是激活的，直到 Outlook 结束，它才结束。

### 19.3.3 线程的优先级

前面提到，操作系统是在调度线程。给线程指定优先级，就可以影响这个调度。

在改变优先级之前，必须理解线程调度器。操作系统根据优先级来调度线程。优先级最高的线程在 CPU 上运行。线程如果在等待资源，就会停止运行，释放 CPU。线程必须等待有几个原因，例如响应睡眠指令、等待磁盘 I/O 的完成，等待网络包的到达等。如果线程不是主动释放 CPU，线程调度器就会抢先安排该线程。如果线程有一个时间量，就可以继续使用 CPU。如果优先级相同的多个线程等待使用 CPU，线程调度器就会使用一个循环调度规则，将 CPU 逐个交给线程使用。如果线程是被其他线程抢先了，它就会排在队列的最后。

只有优先级相同的多个线程在运行，才用得上时间量和循环规则。优先级是动态的。如果线程是 CPU 密集型的（一直需要 CPU，且不等待资源），其优先级就低于用该线程定义的基本优先级。如果线程在等待资源，就会推动优先级向上移动，它的优先级就会增加。由于有这个推动，线程才有可能在下次等待结束时获得 CPU。

在 Thread 类中，可以设置 Priority 属性，以影响线程的基本优先级。Priority 属性需要一个 ThreadPriority 枚举定义的值。该值定义的级别有 Highest、AboveNormal、BelowNormal 和 Lowest。

**提示：**

在给线程指定较高的优先级时要小心，因为这可能降低其他线程的运行几率。如果需要，可以改变优先级一段较短的时间。

### 19.3.4 控制线程

调用 Thread 对象的 Start() 方法，可以创建线程。但是，在调用 Start() 方法后，新线程仍不在 Running 状态，而是在 Unstarted 状态。操作系统的线程调度器选择了要运行的线程后，线程就会改为 Running 状态。读取 Thread.ThreadState 属性，就可以获得线程的当前状态。

使用 Thread.Sleep() 方法，会使线程处于 WaitSleepJoin 状态，在用 Sleep() 方法定义的时间过后，线程就会再次被调用。

要停止另一个线程，可以调用 Thread.Abort() 方法。调用这个方法时，会在接到中止命令的线程中抛出 ThreadAbortException 类型的异常。用一个处理程序捕获这个异常，线程可以在结束前完成一些清理工作。线程还可以在接收到调用 Thread.ResetAbort() 方法的结果 ThreadAbortException 后继续运行。如果线程没有重置中止，接收到中止请求的线程状态将从 AbortRequested 改为 Aborted。

如果需要等待线程的结束，就可以调用 Thread.Join() 方法。Thread.Join() 方法会停止当前线程，把它设置为 WaitSleepJoin 状态，直到加入的线程完成为止。

.NET 1.0 也支持 Thread.Suspend() 和 Thread.Resume() 方法，它们分别用于暂停和继续一个线程。但是，线程在得到 Suspend 请求时，我们并不知道线程在做什么，它可能处于有锁的同

步段中。这很容易导致死锁。这就是这些方法现在被废弃的原因。另外，还可以使用同步对象给线程发信号，这样线程就可以挂起了。于是，线程就知道进入等待状态的最佳时机。

## 19.4 线程池

创建线程是需要时间的。如果有不同的小任务要完成，就可以事先创建许多线程，在应完成这些任务时发出请求。这个线程数应在需要更多的线程时增加，在需要释放资源时减少。

不需要自己创建这样一个列表。该列表由 `ThreadPool` 类管理。这个类会在需要时增减池中线程的个数，直到最大的线程数。池中的最大线程数是可以配置的。在双核 CPU 中，默认设置为 50 个工作线程和 1000 个 I/O 线程。也可以指定在创建线程池时应立即启动的最小线程数，以及线程池中可用的最大线程数。如果有更多的工作要处理，线程池中线程的使用也到了极限，最新的工作就要排队，必须等待线程完成其任务。

下面的示例程序首先要读取工作线程和 I/O 线程的最大线程数，把这个信息写入控制台。接着在 for 循环中，调用 `ThreadPool.QueueUserWorkItem()` 方法，传送一个 `WaitCallback` 类型的委托，把方法 `JobForAThread()` 赋予线程池中的线程。线程池收到这个请求后，就会从池中选择一个线程，来调用该方法。如果线程池还没有运行，就会创建一个线程池，启动第一个线程。如果线程池已经在运行，且有一个自由线程，就把工作传送给这个线程。

```
using System;
using System.Threading;

namespace Wrox.ProCSharp.Threading
{
    class Program
    {
        static void Main()
        {
            int nWorkerThreads;
            int nCompletionPortThreads;
            ThreadPool.GetMaxThreads(out nWorkerThreads, out nCompletionPortThreads);
            Console.WriteLine("Max worker threads: {0}, I/O completion threads: {1}", nWorkerThreads, nCompletionPortThreads);

            for (int i = 0; i < 5; i++)
            {
                ThreadPool.QueueUserWorkItem(JobForAThread);
            }
            Thread.Sleep(3000);
        }

        static void JobForAThread(object state)
        {
            for (int i = 0; i < 3; i++)
            {
                Console.WriteLine("loop {0}, running inside pooled thread {1}", i, Thread.CurrentThread.ManagedThreadId);
                Thread.Sleep(50);
            }
        }
    }
}
```

运行应用程序，可以看到 50 个工作线程的当前设置。5 个任务只由两个线程池中的线程处理，读者运行该程序的结果可能与此不同，也可以改变任务的睡眠时间和要处理的任务数，得到完全不同的结果。

```
Max worker threads: 50, I/O completion threads: 1000
loop 0, running inside pooled thread 4
loop 0, running inside pooled thread 3
loop 1, running inside pooled thread 4
loop 1, running inside pooled thread 3
loop 2, running inside pooled thread 4
loop 2, running inside pooled thread 3
loop 0, running inside pooled thread 4
loop 0, running inside pooled thread 3
loop 1, running inside pooled thread 4
loop 1, running inside pooled thread 3
loop 2, running inside pooled thread 4
loop 2, running inside pooled thread 3
loop 0, running inside pooled thread 4
loop 1, running inside pooled thread 4
loop 2, running inside pooled thread 4
```

线程池使用起来很简单，但它有一些限制：

- 线程池中的所有线程都是后台线程。如果进程中的所有前台线程都结束了，所有的后台线程就会停止。不能把线程池中的线程改为前台线程。
- 不能给线程池中的线程设置优先级或名称。
- 对于 COM 对象，线程池中的所有线程都是多线程单元(multithreaded apartment, MTA)线程。许多 COM 对象都需要单线程单元(single-threaded apartment, MTA)线程。
- 线程池中的线程只能用于时间较短的任务。如果线程要一直运行(如 Word 的拼写检查器线程)，就应使用 Thread 类创建一个线程。

## 19.5 线程问题

用多个线程编程并不容易。在启动访问相同数据的多个线程时，会遇到难以发现的问题。为了避免这些问题，必须特别注意同步问题和多个线程可能发生的其他问题。下面探讨与线程相关的问题：竞态条件和死锁。

### 19.5.1 竞态条件

如果两个或多个线程访问相同的对象，或者访问不同步的共享状态，就会出现竞态条件。

为了演示竞态条件，定义一个 StateObject 类，它包含一个 int 字段和一个方法 Change State。在 ChangeState 方法的实现代码中，验证 state 变量是否包含 5。如果是，就递增其值。下一个语句是 Trace.Assert，它验证 state 现在是否包含 6。在给包含 5 的变量递增了 1 后，该变量的值就应是 6。但事实不一定是这样。例如，如果一个线程刚刚执行完 if(state == 5) 语句，它就被其他线程抢先，调度器去运行另一个线程了。第二个线程现在进入 if 体，由于 state 的值仍是 5，

所以将它递增为 6。第一个线程现在再次被安排执行，在下一个语句中，state 被递增为 7。这时就发生了竞态条件，显示断言信息。

```
public class StateObject
{
    private int state = 5;
    public void ChangeState(int loop)
    {
        if (state == 5)
        {
            state++;
            Trace.Assert(state == 6, "Race condition occurred after " +
                loop + " loops");
        }
        state = 5;
    }
}
```

下面定义一个线程方法来验证这一点。SampleThread 类的方法 RaceCondition() 将一个 StateObject 对象作为其参数。在一个无限 while 循环中，调用方法 ChangeState()。变量 i 仅用于显示断言信息中的循环数。

```
public class SampleThread
{
    public void RaceCondition(object o)
    {
        Trace.Assert(o is StateObject, "o must be of type StateObject");
        StateObject state = o as StateObject;

        int i = 0;
        while (true)
        {
            state.ChangeState(i++);
        }
    }
}
```

在程序的 Main 方法中，创建了一个新的 StateObject 对象，它由所有的线程共享。在 Thread 类的构造函数中，给 RaceCondition 的地址传送一个 SampleThread 类型的对象，以创建 Thread 对象。接着传送 state 对象，使用 Start() 方法启动这个线程。

```
static void Main()
{
    StateObject state = new StateObject();
    for (int i = 0; i < 20; i++)
    {
        new Thread(new SampleThread().RaceCondition).Start(state);
    }
}
```

启动程序，就会出现竞态条件。在竞态条件第一次出现后，还需要多长时间才能第二次出现竞态条件，取决于系统以及将程序建立为发布版本还是调试版本。如果建立为发布版本，该问题的出现次数会比较多，因为代码被优化了。如果系统中有多个 CPU 或使用双核 CPU，其中多个线程可以同时运行，该问题也会比单核 CPU 的出现次数多。在单核 CPU 中，若线程调度是抢先式的，也会出现该问题，只是没有那么频繁。



图 19-2 显示在 3816 个循环后，发生竞态条件的程序断言。多启动应用程序几次，总是会得到不同的结果。

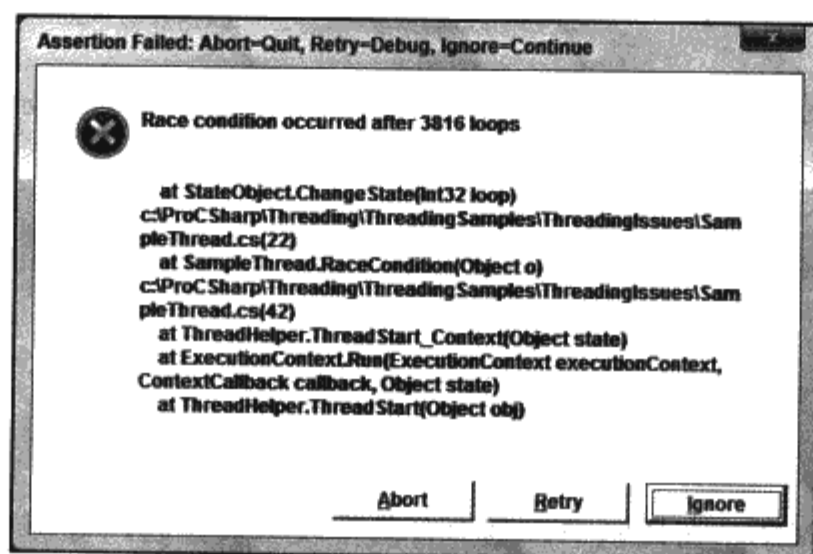


图 19-2

要避免该问题，可以锁定共享的对象。这可以在线程中完成：用下面的 `lock` 语句锁定在线程中共享的变量 `state`。只有一个线程能在锁定块中处理共享的 `state` 对象。由于这个对象由所有的线程共享，因此如果一个线程锁定了 `state`，另一个线程就必须等待该锁定的解除。一旦进行了锁定，线程就拥有该锁定，直到该锁定块的末尾才解除锁定。如果每个改变 `state` 变量引用的对象的线程都使用一个锁定，竞态条件就不会出现。

```
public class SampleThread
{
    public void RaceCondition(object o)
    {
        Trace.Assert(o is StateObject, "o must be of type StateObject");
        StateObject state = o as StateObject;

        int i = 0;
        while (true)
        {
            lock (state) // no race condition with this lock
            {
                state.ChangeState(i++);
            }
        }
    }
}
```

在使用共享对象时，除了进行锁定之外，还可以将共享对象设置为线程安全的对象。其中 `ChangeState()` 方法包含一个 `lock` 语句。由于不能锁定 `state` 变量本身(只有引用类型才能用于锁定)，因此定义一个 `object` 类型的变量 `sync`，将它用于 `lock` 语句。如果每次 `state` 值修改时，都使用同一个同步对象来锁定，竞态条件就不会出现。

```
public class StateObject
{
    private int state = 5;
    private object sync = new object();

    public void ChangeState(int loop)
```



```

    {
        lock (sync)
        {
            if (state == 5)
            {
                state++;
                Trace.Assert(state == 6, "Race condition occurred after " +
                    loop + " loops");
            }
            state = 5;
        }
    }
}

```

### 19.5.2 死锁

过多的锁定也会有麻烦。在死锁中，至少有两个线程被挂起，等待对方解除锁定。由于两个线程都在等待对方，就出现了死锁，线程将无限等待下去。

为了演示死锁，下面实例化两个 `StateObject` 类型的对象，并传送给 `SampleThread` 类的构造函数。创建两个线程，其中一个线程运行方法 `Deadlock1()`，另一个线程运行方法 `Deadlock2()`：

```

StateObject state1 = new StateObject();
StateObject state2 = new StateObject();
new Thread(new SampleThread(state1, state2).Deadlock1).Start();
new Thread(new SampleThread(state1, state2).Deadlock2).Start();

```

方法 `Deadlock1()` 和 `Deadlock2()` 现在改变两个对象 `s1` 和 `s2` 的状态。这就进行了两个锁定。方法 `Deadlock1()` 先锁定 `s1`，接着锁定 `s2`。方法 `Deadlock2()` 先锁定 `s2`，再锁定 `s1`。现在，有可能方法 `Deadlock1()` 中 `s1` 的锁定会被解除。接着出现一次线程切换，`Deadlock2()` 开始运行，并锁定 `s2`。第二个线程现在等待 `s1` 锁定的解除。因为它需要等待，所以线程调度器再次调度第一个线程，但第一个线程在等待 `s2` 锁定的解除。这两个线程现在都在等待，只要锁定块没有结束，就不会解除锁定。这是一个典型的死锁。

```

public class SampleThread
{
    public SampleThread(StateObject s1, StateObject s2)
    {
        this.s1 = s1;
        this.s2 = s2;
    }

    private StateObject s1;
    private StateObject s2;
    public void Deadlock1()
    {
        int i = 0;
        while (true)
        {
            lock (s1)
            {
                lock (s2)
                {
                    s1.ChangeState(i);
                    s2.ChangeState(i++);
                    Console.WriteLine("still running, {0}", i);
                }
            }
        }
    }
}

```

```

    }
}

public void Deadlock2()
{
    int i = 0;
    while (true)
    {
        lock (s2)
        {
            lock (s1)
            {
                s1.ChangeState(i);
                s2.ChangeState(i++);
                Console.WriteLine("still running, {0}", i);
            }
        }
    }
}
}

```

结果是，程序运行了许多循环，不久就没有响应了。“仍在运行”的信息仅在控制台上写入几次。死锁问题的发生频率也取决于系统配置，每次运行的结果都不同。

死锁问题并不总是很明显。一个线程锁定了 s1，接着锁定 s2，另一个线程锁定了 s2，接着锁定 s1。只需改变锁定顺序，这两个线程就会以相同的顺序进行锁定。但是，锁定可能隐藏在方法的深处。为了避免这个问题，可以在应用程序的体系架构中，从一开始就设计好锁定顺序，也可以为锁定定义超时时间。如何定义超时时间详见下一节的内容。

## 19.6 同步

要避免同步问题，最好不要在线程之间共享数据。当然，这并不总是可行的。如果需要共享数据，就必须使用同步技术，确保一次只有一个线程访问和改变共享状态。注意，同步问题与竞态条件和死锁有关。如果不注意这些问题，就很难在应用程序中找到问题的原因，因为线程问题是不定期发生的。

本节讨论可以用于多个线程的同步技术：

- lock 语句
- Interlocked 类
- Monitor 类
- 等待句柄
- Mutex 类
- Semaphore 类
- Event 类
- ReaderWriterLockSlim

lock 语句、Interlocked 类和 Monitor 类可用于进程内部的同步。Mutex 类、Semaphore 类、Event 类和 ReaderWriterLockSlim 类提供了多个进程中的线程同步。



## 19.6.1 lock 语句和线程安全

C#为多个线程的同步提供了自己的关键字：lock 语句。lock 语句是设置锁定和解除锁定的一种简单方式。

在添加 lock 语句之前，先进入另一个竞态条件。类 SharedState 演示了如何使用线程共享的状态，并保存一个整数值。

```
public class SharedState
{
    public int State{ get; set;}
}
```

类 Task 包含方法 DoTheTask(), 该方法是新线程的入口点。在其实现代码中，将 SharedState 的 State 递增 50000 次。变量 sharedState 在这个类的构造函数中初始化：

```
public class Task
{
    SharedState sharedState;
    public Task(SharedState sharedState)
    {
        this.sharedState = sharedState;
    }
    public void DoTheTask()
    {
        for (int i = 0; i < 50000; i++)
        {
            sharedState.State += 1;
        }
    }
}
```

在 Main()方法中，创建一个 SharedState 对象，并传送给 20 个 Thread 对象的构造函数。在启动所有的线程后，Main()方法进入另一个循环，使 20 个线程处于等待状态，直到所有的线程都执行完毕为止。线程执行完毕后，把共享状态的合计值写入控制台。因为执行了 50000 个循环，有 20 个线程，所以写入控制台的值应是 1000000。但是，事实常常并非如此。

```
class Program
{
    static void Main()
    {
        int numThreads = 20;
        SharedState state = new SharedState();
        Thread[] threads = new Thread[numThreads];

        for (int i = 0; i < numThreads; i++)
        {
            threads[i] = new Thread(new Task(state).DoTheTask);
            threads[i].Start();
        }

        for (int i = 0; i < numThreads; i++)
        {
            threads[i].Join();
        }
        Console.WriteLine("summarized {0}", state.State);
    }
}
```

```

    }
}

```

多次运行应用程序的结果如下所示：

```

summarized 939270
summarized 993799
summarized 998304
summarized 937630

```

每次运行的结果都不同，但没有一个结果是正确的。调试版本和发布版本的区别很大。所使用的 CPU 类型不同，结果也不一样。如果将循环次数改为比较小的值，就会多次得到正确的值，但不是每次。这个应用程序非常小，很容易看出问题，但该问题的原因在大型应用程序中就很难确定。

必须在这个程序中添加同步功能，这可以用 lock 关键字实现。

用 lock 语句定义的对象表示，要等待指定对象的锁定解除。只能传送引用类型。锁定值类型只是锁定了一个副本，这是没有什么意义的。编译器会提供一个锁定值类型的错误。进行了锁定后——只有一个线程得到了锁定块，就可以运行 lock 语句块。在 lock 语句块的最后，对象的锁定被解除，另一个等待锁定的线程就可以获得该锁定块了。

```

lock (obj)
{
    // synchronized region
}

```

要锁定静态成员，可以把锁定放在 object 类型上：

```

lock (typeof(StaticClass))
{
}

```

使用 lock 关键字可以将类的实例成员设置为线程安全。这样，一次只有一个线程能访问该实例的 DoThis() 和 DoThat() 方法。

```

public class Demo
{
    public void DoThis()
    {
        lock (this)
        {
            // only one thread a time can access the DoThis and DoThat methods
        }
    }
    public void DoThat()
    {
        lock (this)
        {
        }
    }
}

```

但是，因为实例的对象也可以用于外部的同步访问，我们不能在类中控制这种访问，所以应采用 SyncRoot 模式。在 SyncRoot 模式中，创建了一个私有对象 syncRoot，将这个对象用于 lock 语句。



```

public class Demo
{
    private object syncRoot = new object();

    public void DoThis()
    {
        lock (syncRoot)
        {
            // only one thread a time can access the DoThis and DoThat methods
        }
    }
    public void DoThat()
    {
        lock (syncRoot)
        {
        }
    }
}

```

使用锁定是需要时间的，且并不总是必需的。可以创建类的两个版本，一个同步版本，一个异步版本。这里用修改类 `Demo` 来演示。类 `Demo` 本身并不是同步的，这可以在 `DoThis()` 和 `DoThat()` 方法中看出。该类还定义了 `IsSynchronized` 属性，客户可以从该属性中获得类的同步选项信息。为了获得该类的同步版本，可以使用静态方法 `Synchronized()` 传送一个非同步对象，这个方法会返回 `SynchronizedDemo` 类型的对象。`SynchronizedDemo` 实现为派生自基类 `Demo` 的一个内部类，并重写了基类中的虚成员。重写的成员使用了 `SyncRoot` 模式。

```

public class Demo
{
    private class SynchronizedDemo : Demo
    {
        private object syncRoot = new object();
        private Demo d;

        public SynchronizedDemo(Demo d)
        {
            this.d = d;
        }

        public override bool IsSynchronized
        {
            get { return true; }
        }

        public override void DoThis()
        {
            lock (syncRoot)
            {
                d.DoThis();
            }
        }

        public override void DoThat()
        {
            lock (syncRoot)
            {
                d.DoThat();
            }
        }
    }
}

```



```

public virtual bool IsSynchronized
{
    get { return false; }
}

public static Demo Synchronized(Demo d)
{
    if (!d.IsSynchronized)
    {
        return new SynchronizedDemo(d);
    }
    return d;
}

public virtual void DoThis()
{
}

public virtual void DoThat()
{
}
}

```

必须注意, 在使用 `SynchronizedDemo` 类时, 只有方法是同步的, 对这个类的两个成员的调用并没有同步。

#### 警告:

`SyncRoot` 模式可能使线程安全产生负面影响。.NET 1.0 集合类实现了 `SyncRoot` 模式; .NET 2.0 的泛型集合类不再实现这个模式。

下面研究一下前面的例子。如果试图用 `SyncRoot` 模式锁定对属性的访问, 使 `SharedState` 类变成线程安全的, 仍会出现前面描述的竞态条件。

```

public class SharedState
{
    private int state = 0;
    private object syncRoot = new object();

    public int State // there's still a race condition, don't do this!
    {
        get { lock (syncRoot) { return state; } }
        set { lock (syncRoot) { state = value; } }
    }
}

```

调用方法 `DoTheTask()` 的线程访问 `SharedState` 类的 `get` 存取器, 以获得 `state` 的当前值, 接着 `get` 存取器给 `state` 设置新值。在调用对象的 `get` 和 `set` 存取器期间, 对象没有锁定, 另一个线程可以获得临时值。

```

public void DoTheTask()
{
    for (int i = 0; i < 50000; i++)
    {
        sharedState.State += 1;
    }
}

```

所以，最好不改变 `SharedState` 类，让它没有线程安全性。

```
public class SharedState
{
    private int state = 0;

    public int State
    {
        get { return state; }
        set { state = value; }
    }
}
```

然后在方法 `DoTheTask()` 中，将 `lock` 语句添加到合适的地方：

```
public void DoTheTask()
{
    for (int i = 0; i < 50000; i++)
    {
        lock (sharedState)
        {
            sharedState.State += 1;
        }
    }
}
```

这样，应用程序的结果就总是正确的：

```
summarized 1000000
```

#### 警告：

在一个地方使用 `lock` 语句并不意味着，访问对象的其他线程都在等待。必须对每个访问共享状态的线程显式使用同步功能。

当然，还必须修改 `SharedState` 类的设计，将递增提供为一个原子操作。这是一个设计问题——什么是类的原子功能？

```
public class SharedState
{
    private int state = 0;
    private object syncRoot = new object();
```

```
    public int State
    {
        get { return state; }
    }
    public int IncrementState()
    {
        lock (syncRoot)
        {
            return ++state;
        }
    }
}
```

锁定状态递增还有一种更快的方式，如下所示。

19.6.2 Interlocked

Interlocked 类用于使变量的简单语句原子化。i++不是线程安全的，它的操作包括从内存中获取一个值，给该值递增 1，再将它存储回内存。这些操作都可能会被线程调度器打断。Interlocked 类提供了以线程安全的方式递增、递减和交换值的方法。

Interlocked 类提供的方法如表 19-1 所示。

表 19-1

Interlocked 类的成员	说 明
Increment()	Increment()方法递增一个变量，把结果存储到一个原子操作中
Decrement()	Decrement()递减一个变量，并存储结果
Exchange()	Exchange()将一个变量设置为指定的值，并返回变量的初始值
CompareExchange()	CompareExchange()对两个变量进行相等比较，如果它们相等，就设置指定的值，返回初始值
Add()	Add()对两个值执行相加操作，用结果替代第一个变量
Read()	Read()方法用于在一个原子操作中从内存中读取 64 位值。在 32 位系统中，读取 64 位不是原子化的，而需要从两个内存地址中读取 在 64 位系统中，不需要 Read()方法，因为访问 64 位是一个原子操作

与其他同步技术相比，使用 Interlocked 类会快得多。但是，它只能用于简单的同步问题。

例如，这里不使用 lock 语句锁定对 someState 变量的访问，把它设置为一个新值，以防它是空的，而可以使用 Interlocked 类，它比较快：

```
lock (this)
{
    if (someState == null)
    {
        someState = newState;
    }
}
```

这个功能相同、但比较快的版本使用了 Interlocked.CompareExchange 方法：

```
Interlocked.CompareExchange<SomeState>(ref someState, newState, null);
```

不在 lock 语句中执行递增操作：

```
public int State
{
    get
    {
        lock (this)
        {
            return ++state;
        }
    }
}
```

而使用较快的 Interlocked.Increment()：

```

public int State
{
    get
    {
        return Interlocked.Increment(ref state);
    }
}

```

### 19.6.3 Monitor 类

C#的 lock 语句由编译器解析为使用 Monitor 类。下面的 lock 语句:

```

lock (obj)
{
    // synchronized region for obj
}

```

解析为调用 Enter()方法,该方法会一直等待,直到线程获得对象的锁定为止。一次只有一个线程能成为对象锁定的拥有者。只要解除了锁定,线程就可以进入同步段。Monitor 类的 Exit()方法解除了锁定。Exit()方法放在 try 块的 finally 处理程序中,所以如果抛出了异常,也会解除该锁定。

提示:

try/finally 详见第 14 章。

```

Monitor.Enter(obj);
try
{
    // synchronized region for obj
}
finally
{
    Monitor.Exit(obj);
}

```

与 C#的 lock 语句相比, Monitor 类的主要优点是:可以添加一个等待获得锁定的超时值。这样就不会无限期地等待获得锁定,而可以使用 TryEnter 方法,给它传送一个超时值,确定等待获得锁定的最长时间。如果得到了 obj 的锁定, TryEnter 方法就返回 true,访问由对象 obj 锁定的状态。如果另一个线程锁定 obj 的时间超过了 500 毫秒, TryEnter 方法就返回 false,线程不再等待,而是执行其他操作。也许在以后,该线程会尝试再次获得该锁定。

```

if (Monitor.TryEnter(obj, 500))
{
    try
    {
        // acquired the lock
        // synchronized region for obj
    }
    finally
    {
        Monitor.Exit(obj);
    }
}
else

```

```
{
    // didn't get the lock, do something else
}
```

19.6.4 等待句柄

WaitHandle 是一个抽象基类，用于等待一个信号的设置。可以等待不同的信号，因为 WaitHandle 是一个基类，可以从中派生一些类。

在本章前面使用异步委托时，已经使用了 WaitHandle。异步委托的方法 BeginInvoke() 返回一个实现了 IAsyncResult 接口的对象。使用 IAsyncResult 接口，可以用属性 AsyncWaitHandle 访问 WaitHandle。在调用 WaitOne() 方法时，线程会等待接收一个与等待句柄相关的信号。

```
static void Main()
{
    TakesAWhileDelegate dl = TakesAWhile;

    IAsyncResult ar = dl.BeginInvoke(1, 3000, null, null);
    while (true)
    {
        Console.WriteLine(".");
        if (ar.AsyncWaitHandle.WaitOne(50, false))
        {
            Console.WriteLine("Can get the result now");
            break;
        }
    }
    int result = dl.EndInvoke(ar);
    Console.WriteLine("result: {0}", result);
}
```

WaitHandle 类定义的、执行等待的方法如表 19-2 所示。

表 19-2

WaitHandle 类的成员	说 明
WaitOne()	WaitOne() 是一个实例方法，利用它可以等待一个信号的发生。也可以为最大等待时间指定一个超时值
WaitAll()	WaitAll() 是一个静态方法，用于传送 WaitHandle 对象的数组，并等待所有的句柄发出信号
WaitAny()	WaitAny() 是一个静态方法，用于传送 WaitHandle 对象的数组，并等待其中一个句柄发出信号。这个方法返回发出信号的等待句柄对象的索引，以便确定可以在程序中继续执行什么功能。如果在句柄发出信号之前超时，WaitAny() 就返回 WaitTimeout

使用 SafeWaitHandle 属性，还可以将一个内置句柄赋予一个操作系统资源，并等待该句柄。例如，可以指定一个 SafeWaitHandle 等待文件 I/O 操作的完成，或者指定定制的 SafeTransactionHandle，参见第 22 章。

类 Mutex、Event 和 Semaphore 派生自基类 WaitHandle，所以可以在等待时使用它们。



## 19.6.5 Mutex 类

Mutex(mutual exclusion, 互斥)是.NET Framework 中提供同步访问多个进程的一个类。它非常类似于 Monitor 类, 因为它们都只有一个线程能拥有锁定。只有一个线程能获得互斥锁定, 访问受互斥锁定保护的同步代码区域。

在 Mutex 类的构造函数中, 可以指定互斥锁定是否最初应由调用线程拥有, 定义互斥锁定的名称, 获得互斥锁定是否已存在的信息。在下面的示例代码中, 第三个参数定义为输出参数, 接收一个表示互斥锁定是否为新创建的布尔值。如果返回的值是 false, 就表示互斥锁定已经定义。互斥锁定可以在另一个进程中定义, 因为操作系统知道有名称的互斥锁定, 它由不同的进程共享。如果没有给互斥锁定指定名称, 互斥锁定就是未命名的, 不在不同的进程之间共享。

```
bool createdNew;
Mutex mutex = new Mutex(false, "ProCSharpMutex", out createdNew);
```

要打开已有的互斥锁定, 还可以使用方法 Mutex.OpenExisting(), 它不需要用构造函数创建互斥锁定时需要的.NET 权限。

Mutex 类派生自基类 WaitHandle, 因此可以利用 WaitOne()方法获得互斥锁定, 在该过程中成为该互斥锁定的拥有者。调用 ReleaseMutex()方法, 即可释放互斥锁定。

```
if (mutex.WaitOne())
{
    try
    {
        // synchronized region
    }
    finally
    {
        mutex.ReleaseMutex();
    }
}
else
{
    // some problem happened while waiting
}
```

由于系统知道有名称的互斥锁定, 因此可以使用它禁止应用程序启动两次。在下面的 Windows 窗体应用程序中, 调用了 Mutex 对象的构造函数。接着验证名称为 SingletonWinAppMutex 的互斥锁定是否存在。如果存在, 应用程序就退出。

```
static class Program
{
    [STAThread]
    static void Main()
    {
        bool createdNew;
        Mutex mutex = new Mutex(false, "SingletonWinAppMutex", out createdNew);
        if (!createdNew)
        {
            MessageBox.Show("You can only start one instance of the application");
            Application.Exit();
            return;
        }
    }
}
```

```

        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);
        Application.Run(new Form1());
    }
}

```

### 19.6.6 Semaphore 类

旗语(Semaphore)锁定非常类似于互斥锁定,其区别是,旗语锁定可以同时由多个线程使用。旗语锁定是一种计数的互斥锁定。使用旗语锁定,可以定义允许同时访问受旗语锁定保护的资源的线程个数。如果有许多资源,且只允许一定数量的线程访问该资源,就可以使用旗语锁定。例如,要访问系统上的物理 I/O 端口,且有三个端口可用,就允许三个线程同时访问 I/O 端口,但第四个线程需要等待前三个线程中的一个释放资源。

在下面的示例程序中,Main()方法创建了 6 个线程和一个计数为 4 的旗语锁定。在 Semaphore 类的构造函数中,定义了锁定数的计数,它可以用旗语锁定(第二个参数)来获得,还定义了最初自由的锁定数(第一个参数)。如果第一个参数的值小于第二个参数,它们的差就是已经赋予线程的旗语锁定数。与互斥锁定一样,也可以给旗语锁定指定名称,使之在不同的进程之间共享。这里定义旗语锁定时没有指定名称,所以它只能在这个进程中使用。在创建了 Semaphore 对象之后,启动六个线程,它们都获得了相同的旗语锁定。

```

using System;
using System.Threading;
using System.Diagnostics;

namespace Wrox.ProCSharp.Threading
{
    class Program
    {
        static void Main()
        {
            int threadCount = 6;
            int semaphoreCount = 4;
            Semaphore semaphore = new Semaphore(semaphoreCount, semaphoreCount);
            Thread[] threads = new Thread[threadCount];

            for (int i = 0; i < threadCount; i++)
            {
                threads[i] = new Thread(ThreadMain);
                threads[i].Start(semaphore);
            }

            for (int i = 0; i < threadCount; i++)
            {
                threads[i].Join();
            }
            Console.WriteLine("All threads finished");
        }
    }
}

```

在线程的主方法 ThreadMain()中,线程利用 WaitOne()锁定了旗语。旗语锁定的计数是 4,所以有四个线程可以获得锁定。线程 5 必须等待,这里还定义了最大等待时间为 500 毫秒。如果在该等待时间过后未能获得锁定,线程就把一个信息写入控制台,在循环中继续等待。只要获得了锁定,线程就把一个信息写入控制台,睡眠一段时间,然后解除锁定。在解除锁定时,

一定要解除资源的锁定。这就是在 finally 处理程序中调用 Semaphore 类的 Release() 方法的原因。

```
static void ThreadMain(object o)
{
    Semaphore semaphore = o as Semaphore;
    Trace.Assert(semaphore != null, "o must be a Semaphore type");
    bool isCompleted = false;
    while (!isCompleted)
    {
        if (semaphore.WaitOne(600, false))
        {
            try
            {
                Console.WriteLine("Thread {0} locks the semaphore",
                    Thread.CurrentThread.ManagedThreadId);
                Thread.Sleep(2000);
            }
            finally
            {
                semaphore.Release();
                Console.WriteLine("Thread {0} releases the semaphore",
                    Thread.CurrentThread.ManagedThreadId);
                isCompleted = true;
            }
        }
        else
        {
            Console.WriteLine("Timeout for thread {0}; wait again",
                Thread.CurrentThread.ManagedThreadId);
        }
    }
}
```

运行应用程序，可以看到有四个线程获得了锁定。ID 为 7 和 8 的线程需要等待。该等待会重复进行，直到四个获得锁定的线程之一解除了旗语锁定。

```
Thread 3 locks the semaphore
Thread 4 locks the semaphore
Thread 5 locks the semaphore
Thread 6 locks the semaphore
Timeout for thread 8; wait again
Timeout for thread 7; wait again
Timeout for thread 8; wait again
Timeout for thread 7; wait again
Timeout for thread 7; wait again
Timeout for thread 8; wait again
Thread 3 releases the semaphore
Thread 8 locks the semaphore
Thread 4 releases the semaphore
Thread 7 locks the semaphore
Thread 5 releases the semaphore
Thread 6 releases the semaphore
Thread 8 releases the semaphore
Thread 7 releases the semaphore
All threads finished
```



### 19.6.7 Events 类

事件是另一个系统级的资源同步方法。为了在托管代码中使用系统事件，.NET Framework 在 System.Threading 命名空间中提供了 ManualResetEvent 和 AutoResetEvent 类。

**提示：**

第 7 章介绍了 C# 中的 event 关键字，它与 System.Threading 命名空间中的 Event 类没有关系。event 关键字基于委托，而上述两个 Event 类是 .NET 封装器，用于系统级的内置事件资源的同步。

可以使用事件通知其他线程：这里有一些数据，完成了一些操作等。事件可以发信号，也可以不发信号。使用前面介绍的 WaitHandle 类，线程可以等待处于发信号状态的事件。

调用 Set() 方法，即可使 ManualResetEvent 发信号。调用 Reset() 方法，可以使之返回不发信号的状态。如果多个线程等待一个事件发出信号，并调用了 Set() 方法，就释放所有等待的线程。另外，如果一个线程刚刚调用了 WaitOne() 方法，但事件已经发出了信号，等待的线程就可以继续等待。

AutoResetEvent 也是通过 Set() 方法发信号。也可以使用 Reset() 方法使之返回不发信号的状态。但是，如果一个线程在等待自动重置的事件发信号，当第一个线程的等待结束时，该事件会自动变为不发信号的状态。这样，如果多个线程在等待事件发信号，就只有一个线程结束其等待状态，它不是等待时间最长的线程，而是优先级最高的线程。

为了演示 AutoResetEvent 类的事件，下面的 ThreadTask 类定义了 Calculation() 方法，这是线程的入口点。在这个方法中，线程接收用于计算的输入数据（由结构 InputData 定义），将结果写入变量 result，它可以通过 Result 属性来访问。只要完成了计算（在随机的一段时间过后），就调用 AutoResetEvent 的 Set() 方法，向事件发信号。

```
public struct InputData
{
    public int X;
    public int Y;
    public InputData(int x, int y)
    {
        this.X = x;
        this.Y = y;
    }
}

public class ThreadTask
{
    private AutoResetEvent autoEvent;
    private int result;

    public int Result
    {
        get { return result; }
    }

    public ThreadTask(AutoResetEvent ev)
    {
        this.autoEvent = ev;
    }
}
```

```

    }

    public void Calculation(object obj)
    {
        InputData data = (InputData)obj;
        Console.WriteLine("Thread {0} starts calculation",
            Thread.CurrentThread.ManagedThreadId);
        Thread.Sleep(new Random().Next(3000));
        result = data.X + data.Y;

        // signal the event - completed!
        Console.WriteLine("Thread {0} is ready",
            Thread.CurrentThread.ManagedThreadId);
        autoEvent.Set();
    }
}

```

程序的 Main() 方法定义了包含四个 AutoResetEvent 对象的数组和包含四个 ThreadTask 对象的数组。每个 ThreadTask 在构造函数中用一个 AutoResetEvent 对象初始化，这样每个线程在完成时都有自己的事件对象来发信号。现在使用 ThreadPool 类调用 QueueUserWorkItem() 方法，让后台线程执行计算任务。

```

class Program
{
    static void Main()
    {
        int taskCount = 4;

        AutoResetEvent[] autoEvents = new AutoResetEvent[taskCount];
        ThreadTask[] tasks = new ThreadTask[taskCount];

        for (int i = 0; i < taskCount; i++)
        {
            autoEvents[i] = new AutoResetEvent(false);
            tasks[i] = new ThreadTask(autoEvents[i]);

            ThreadPool.QueueUserWorkItem(tasks[i].Calculation,
                new InputData(i + 1, i + 3));
        }
        //...
    }
}

```

WaitHandle 类现在用于等待数组中的任意一个事件。WaitAny() 等待向任意一个事件发信号。从 WaitAny() 返回的 index 匹配数组中传送给 WaitAny() 的事件，以提供向哪个事件发信号的信息，并从这个事件中读取结果。

```

        for (int i = 0; i < taskCount; i++)
        {
            int index = WaitHandle.WaitAny(autoEvents);
            if (index == WaitHandle.WaitTimeout)
            {
                Console.WriteLine("Timeout!!");
            }
            else
            {
                Console.WriteLine("finished task for {0}, result: {1}", index,
                    tasks[index].Result);
            }
        }
    }
}

```



启动应用程序，可以看到线程在进行计算，设置事件，通知主线程，它可以读取结果了。由于随机时间、是调试版本还是发布版本、以及硬件的不同，会看到不同的顺序，线程池中有不同数量的线程在执行任务。这里重用了线程池中的线程 4，完成了两个任务，因为它比较快，能第一个完成计算。

```
Thread 3 starts calculation
Thread 4 starts calculation
Thread 5 starts calculation
Thread 4 is ready
finished task for 1, result: 6
Thread 4 starts calculation
Thread 3 is ready
finished task for 0, result: 4
Thread 4 is ready
finished task for 3, result: 10
Thread 5 is ready
finished task for 2, result: 8
```

19.6.8 ReaderWriterLockSlim

锁定机制要允许锁定多个阅读器(而不是一个写入器)访问资源，可以使用类 ReaderWriterLockSlim。这个类提供了一个锁定功能，如果没有写入器锁定资源，就允许多个阅读器访问资源，但只能有一个写入器锁定该资源。

提示：

ReaderWriterLockSlim 是 .NET 3.0 中新增的。 .NET 1.0 中有类似功能的类是 ReaderWriterLock。 ReaderWriterLockSlim 重新设计为防止死锁，提供更好的性能。

ReaderWriterLockSlim 的方法和属性如表 19-3 所示。

表 19-3

ReaderWriterLockSlim 的方法	说 明
TryEnterReadLock() EnterReadLock() ExitReadLock()	使用 TryEnterReadLock()和 EnterReadLock(), 会添加对访问资源的读取锁定。只要没有写入锁定，读取锁定就是成功的。允许同时进行多个读取操作 使用 TryEnterReadLock()可以为等待获得锁定的最大时间指定一个超时值。 ExitReadLock()释放锁定
TryEnterUpgradableReadLock() EnterUpgradableReadLock() ExitUpgradableReadLock()	如果在执行了对资源的读取访问后，读取锁定需要改为写入锁定，就可以使用 TryEnterUpgradableReadLock()和 EnterUpgradableReadLock()。有读取锁定的线程可以获得写入锁定，而无需释放读取锁定
TryEnterWriteLock() EnterWriteLock() ExitWriteLock()	TryEnterWriteLock()和 EnterWriteLock()用于获得对资源的写入锁定。只有一个请求该锁定的线程能获得这个锁定。另外，任何线程都不能有读取锁定。在等待写入锁定时，还必须释放所有的读取锁定。 如果一个包含写入锁定的线程试图再次获得写入锁定，还创建了 ReaderWriterLockSlim，且把 RecursionPolicy 设置为 LockRecursionPolicy.SupportsRecursion，就可以获得该锁定

ReaderWriterLockSlim 类的属性给出了当前锁定的一些状态信息。

表 19-4

ReaderWriterLockSlim 类的属性	说 明
CurrentReadCount	返回获得读取锁定的线程数
IsReadLockHeld	这些属性返回对应锁定类型的布尔值
IsUpgradableReadLockHeld	
IsWriteLockHeld	
WaitingReadCount	这些属性返回等待对应锁定类型的线程数
WaitingUpgradableReadCount	
WaitingWriteCount	
RecursionPolicy	在递归条件下，一个线程可以再次获得锁定。Recursion Policy 是一个只读属性，返回 LockRecursionPolicy。递归策略可以用 Reader- WriterLockSlim 构造函数配置为 NoRecursion 或 SupportsRecursion
RecursiveReadCount	
RecursiveUpgradableReadCount	
RecursiveWriteCount	

下面的示例程序创建了一个包含 6 项的集合和一个 ReaderWriterLockSlim 对象。方法 ReaderMethod() 获得一个读取锁定，读取列表中的所有项，把它们写到控制台上。方法 WriterMethod() 试图获得一个写入锁定，改变集合的所有值。在 Main() 方法中，启动 6 个线程，调用方法 ReaderMethod() 或 WriterMethod()。

```
using System;
using System.Collections.Generic;
using System.Threading;

namespace Wrox.ProCSharp.Threading
{
    class Program
    {
        private static List<int> items = new List<int> ()
            { 0, 1, 2, 3, 4, 5 };
        private static ReaderWriterLockSlim rwl = new
            ReaderWriterLockSlim(LockRecursionPolicy.SupportsRecursion);

        static void ReaderMethod(object reader)
        {
            try
            {
                rwl.EnterReadLock();
                for (int i = 0; i < items.Count; i++)
                {
                    Console.WriteLine("reader {0}, loop: {1}, item: {2}",
                        reader, i, items[i]);
                    Thread.Sleep(40);
                }
            }
            finally
            {
                rwl.ExitReadLock();
            }
        }
    }
}
```

```

    }
}

static void WriterMethod(object writer)
{
    try
    {
        while (!rw1.TryEnterWriteLock(50))
        {
            Console.WriteLine("Writer {0} waiting for the write lock",
                writer);
            Console.WriteLine("current reader count: {0}",
                rw1.CurrentReadCount);
        }
        Console.WriteLine("Writer {0} acquired the lock", writer);
        for (int i = 0; i < items.Count; i++)
        {
            items[i]++;
            Thread.Sleep(50);
        }
        Console.WriteLine("Writer {0} finished", writer);
    }
    finally
    {
        rw1.ExitWriteLock();
    }
}

static void Main()
{
    new Thread(WriterMethod).Start(1);
    new Thread(ReaderMethod).Start(1);
    new Thread(ReaderMethod).Start(2);
    new Thread(WriterMethod).Start(2);
    new Thread(ReaderMethod).Start(3);
    new Thread(ReaderMethod).Start(4);
}
}

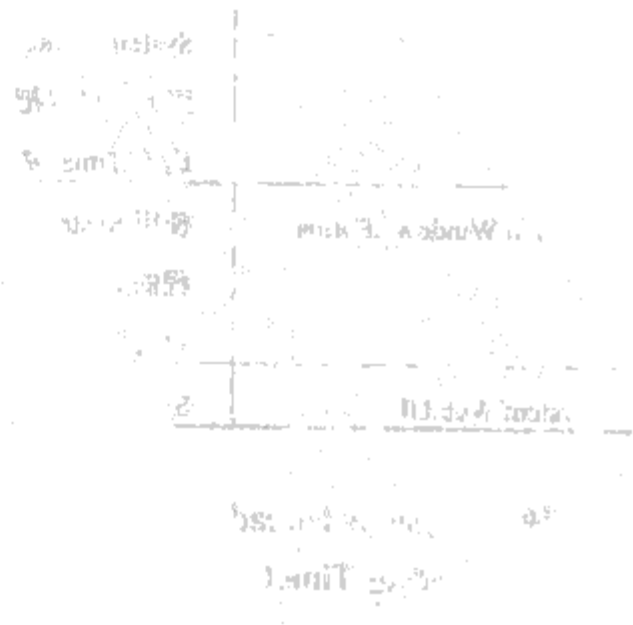
```

运行这个应用程序，第一个写入器先获得锁定。第二个写入器和所有的读取器需要等待。接着读取器可以同时工作，而第二个写入器仍在等待资源。

```

Writer 1 acquired the lock
Writer 2 waiting for the write lock
current reader count: 0
Writer 2 waiting for the write lock
current reader count: 0
Writer 2 waiting for the write lock
current reader count: 0
Writer 2 waiting for the write lock
current reader count: 0
Writer 1 finished
reader 4, loop: 0, item: 1
reader 1, loop: 0, item: 1
Writer 2 waiting for the write lock
current reader count: 4
reader 2, loop: 0, item: 1
reader 3, loop: 0, item: 1
reader 4, loop: 1, item: 2
reader 1, loop: 1, item: 2

```



```
reader 3, loop: 1, item: 2
reader 2, loop: 1, item: 2
Writer 2 waiting for the write lock
current reader count: 4
reader 4, loop: 2, item: 3
reader 1, loop: 2, item: 3
reader 2, loop: 2, item: 3
reader 3, loop: 2, item: 3
Writer 2 waiting for the write lock
current reader count: 4
reader 4, loop: 3, item: 4
reader 1, loop: 3, item: 4
reader 2, loop: 3, item: 4
reader 3, loop: 3, item: 4
reader 4, loop: 4, item: 5
reader 1, loop: 4, item: 5
Writer 2 waiting for the write lock
current reader count: 4
reader 2, loop: 4, item: 5
reader 3, loop: 4, item: 5
reader 4, loop: 5, item: 6
reader 1, loop: 5, item: 6
reader 2, loop: 5, item: 6
reader 3, loop: 5, item: 6
Writer 2 waiting for the write lock
current reader count: 4
Writer 2 acquired the lock
Writer 2 finished
```

19.7 Timer 类

.NET Framework 提供了几个 Timer 类，用于在某个时间间隔后调用方法。表 19-5 列出了 Timer 类、其命名空间和功能。

表 19-5

命 名 空 间	说 明
System.Threading	System.Threading 命名空间中的 Timer 类提供了核心功能。在构造函数中，可以传送一个委托，该委托应在指定的时间间隔后调用
System.Timers	System.Timers 命名空间中的 Timer 类是一个组件，因为它派生于 Component 基类。因此，可以把它从工具箱拖放到服务器应用程序（如 Windows 服务）的设计界面上。这个 Timer 类使用 System.Threading.Timer，但提供了基于事件的机制，而不是委托
System.Windows.Forms	使用 System.Threading 和 System.Timers 命名空间中的 Timer 类，可以从不是调用线程的另一个线程中调用回调函数或事件方法。Windows 窗体控件绑定到创建线程上。对这个线程的回调是通过 System.Windows.Forms 命名空间中的 Timer 类完成的
System.Web.UI	System.Web.UI 命名空间中的 Timer 类是一个 AJAX 扩展，可以用于 Web 页面

使用 System.Threading.Timer 类，可以把要调用的方法传送为构造函数的第一个参数。这个方法必须满足 TimeCallback 委托的要求，定义 void 返回类型和 object 参数。在第二个参数中，

可以传送任意对象，用回调方法中的 `object` 参数接收。例如，可以传送 `Event` 对象，给调用者发送信号。第三个参数指定第一次调用回调方法时的时间段。最后一个参数指定了回调的重复时间间隔。如果计时器应只启动一次，就把第 4 个参数设置为 -1。

如果创建 `Timer` 对象后应改变时间间隔，就可以用 `Change()` 方法传送新值：

```
private static void ThreadingTimer()
{
    System.Threading.Timer t1 = new System.Threading.Timer(
        TimeAction, null, TimeSpan.FromSeconds(2),
        TimeSpan.FromSeconds(3));

    Thread.Sleep(15000);

    t1.Dispose();
}

static void TimeAction(object o)
{
    Console.WriteLine("System.Threading.Timer {0:T}", DateTime.Now);
}
```

`System.Timer` 命名空间中的 `Timer` 类的构造函数只需要一个时间间隔。该时间间隔后应调用的方法用 `Elapsed` 事件指定。这个事件需要一个 `ElapsedEventHandler` 类型的委托，这个委托需要的对象和 `ElapsedEventArgs` 参数与 `TimeAction` 方法相同。`AutoReset` 属性指定计时器是否重复启动。把这个属性设置为 `false`，事件就只启动一次。调用 `Start` 方法允许计时器启动事件。除了调用 `Start` 方法之外，还可以把 `Enabled` 属性设置为 `true`。在后台，`Start()` 什么也不做。`Stop()` 方法把 `Enabled` 属性设置为 `false`，停止计时器。

```
private static void TimersTimer()
{
    System.Timers.Timer t1 = new System.Timers.Timer(1000);
    t1.AutoReset = true;
    t1.Elapsed += TimeAction;
    t1.Start();
    Thread.Sleep(10000);
    t1.Stop();

    t1.Dispose();
}

static void TimeAction(object sender, System.Timers.ElapsedEventArgs e)
{
    Console.WriteLine("System.Timers.Timer {0:T}", e.SignalTime);
}
```

## 19.8 COM 单元

线程总是一个与 COM 对象相关的重要主题。COM 为同步定义了单元模型。在单线程单元 (STA) 中，COM 运行库会执行同步。多线程单元 (MTA) 的性能比较好，但没有 COM 运行库的同步功能。

COM 组件在注册表中设置了一个配置值，从而定义了它需要的单元模型。以线程安全的方式开发的 COM 组件支持 MTA。多个线程可以同时访问这个组件，该组件必须自己实现同步。



不能处理多个线程的 COM 组件需要使用 STA。在 STA 中, 只有一个线程(总是这样)访问组件。另一个线程只有使用代理, 给连接到 COM 对象上的线程发送一个 Windows 信息, 才能访问组件。STA 使用该 Windows 信息进行同步。

VB6 组件仅支持 STA 模型。用 both 选项配置的 COM 组件支持 STA 和 MTA。

COM 组件定义了对单元的要求, 而实例化 COM 对象的线程定义了运行它的单元。这个单元应该就是 COM 需要的单元。

.NET 线程默认运行在 MTA 上。在 Windows 应用程序的 Main() 方法中, 有时可以看到特性 [STAThread]。这个特性指定, 主线程加入 STA。Windows 窗体应用程序需要一个 STA 线程。

```
[STAThread]
static void Main()
{
    //...
```

在创建新线程时, 可以将 [STAThread] 或 [MTAThread] 特性应用于线程的入口点方法, 或调用 Thread 类的 SetApartmentThread() 方法, 来定义单元模型, 之后启动线程:

```
Thread t1 = new Thread(DoSomeWork);
t1.SetApartmentState(ApartmentState.STA);
t1.Start();
```

使用 GetApartmentThread() 方法可以获得线程的单元。

**提示:**

第 24 章介绍了 .NET 与 COM 组件的交互操作和 COM 单元模型的详细内容。

## 19.9 基于事件的异步模式

本章前面介绍了基于 IAsyncResult 接口的异步模式。在异步回调中, 回调线程不同于调用线程。使用 Windows 窗体或 WPF 时, 这是一个问题, 因为 Windows 窗体和 WPF 控件绑定到一个线程上。对于每个控件, 都只能从创建该控件的线程中调用方法。也就是说, 如果有一个后台线程, 就不能直接在这个线程中访问 UI 控件。

在 Windows 窗体控件中, 唯一可以从非创建线程中调用的是方法 Invoke()、BeginInvoke()、EndInvoke() 和属性 InvokeRequired。BeginInvoke() 和 EndInvoke() 是 Invoke() 的异步版本。这些方法会切换到创建控件的线程上, 调用赋予一个委托参数的方法, 该委托参数可以传送给这些方法。这些方法的使用并不简单, 这就是 .NET 2.0 新组件和新异步模式一起开发的原因, 即基于事件的异步模式。

在基于事件的异步模式中, 异步组件提供了后缀为 Async 的方法。例如同步方法 DoATask() 的异步版本是 DoATaskAsync()。为了得到结果信息, 组件还需要定义一个后缀为 Completed 的事件, 例如 DoATaskCompleted。DoATaskAsync() 方法中的操作在后台线程上运行时, 会在调用线程中触发 DoATaskCompleted 事件。

在基于事件的异步模式中, 异步组件可以支持取消操作, 提供进度信息。对于取消操作, 方法的名称应是 CancelAsync()。对于进度信息, 应提供后缀为 ProgressChanged 的事件, 例如

DoATaskProgressChanged。

提示：

如果没有编写过 Windows 应用程序，就可以跳过这一节，继续阅读后面的内容。注意，在 Windows 应用程序中使用线程会增加复杂性，应在阅读了 Windows 窗体的章节(第 31~33 章)或 WPF(第 34 和 35 章)后，再阅读本节。无论如何，从 Windows 窗体的角度来看，这里演示的 Windows 窗体应用程序都是非常简单的。

19.9.1 BackgroundWorker 类

BackgroundWorker 类是异步事件模式的一种实现方案。这个类实现的方法、属性和事件如表 19-6 所示。

表 19-6

BackgroundWorker 类的成员	说 明
IsBusy	在激活异步任务时，属性 IsBusy 返回 true
CancellationPending	在调用 CancelAsync()方法后，属性 CancellationPending 返回 true。如果这个属性设置为 true，异步任务就应停止其工作
RunWorkerAsync() DoWork	方法 RunWorkerAsync()引发 DoWork 事件，在一个单独的线程中启动异步任务
CancelAsync() WorkerSupportCancellation	如果启用了取消功能(将 WorkerSupportCancellation 属性设置为 true)，就可以用 CancelAsync()方法取消异步任务
ReportProgress() ProgressChanged WorkerReportsProgress	如果 WorkerReportsProgress 属性设置为 true，BackgroundWorker 就可以给出异步任务进度的临时反馈信息。调用 ReportProgress()方法，异步任务可以提供已完成的工作百分数反馈信息，之后这个方法会引发 ProgressChanged 事件
RunWorkerCompleted	无论取消与否，只要完成了异步任务，就引发 RunWorkerCompleted 事件

提示：

另一个实现异步事件模式的类是 System.Net 命名空间中的组件 WebClient。这个类使用 WebRequest 和 WebResponse 类，但提供了一个易于使用的接口。WebRequest 和 WebResponse 类也提供了异步编程方式，它基于异步模式和 IAsyncResult 接口。

下面的示例程序演示了 BackgroundWorker 控件在 Windows 窗体应用程序中的用法，它执行一个需要一定时间的任务。创建一个新的 Windows 窗体应用程序，在窗体上添加三个标签控件、三个文本框控件、两个按钮控件、一个进度条控件和一个 BackgroundWorker 控件，如图 19-3 所示。

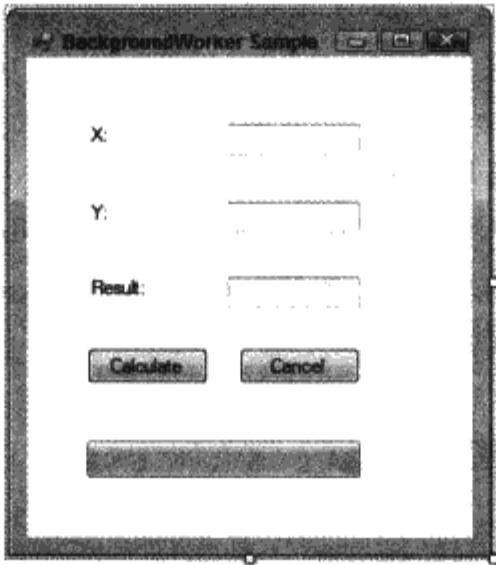


图 19-3

按表 19-7 配置控件的属性。

表 19-7

控 件	属性和事件	值
标签	Text	X:
文本框	Name	textbox
标签	Text	Y:
文本框	Name	textBoxY
标签	Text	Result:
文本框	Name	textBoxResult
按钮	Name	buttonCalculate
Text	Calculate	
Click	OnCalculate	
按钮	Name	buttonCancel
Text	Cancel	
Enabled	False	
Click	OnCancel	
进度条	Name	ProgressBar
BackgroundWorker	Name	backgroundWorker
DoWork	OnDoWork	
WorkCompleted	OnWorkCompleted	

在项目中添加结构 CalcInput。这个结构用于包含文本框控件中的输入数据。

```
public struct CalcInput
{
    public CalcInput(int x, int y)
    {
```

```

        this.x = x;
        this.y = y;
    }
    public int x;
    public int y;
}

```

方法 `OnCalculate()` 是按钮控件 `buttonCalculate` 的 `Click` 事件处理程序。在执行过程中, 按钮 `buttonCalculate` 被禁用, 所以在计算完成之前, 用户不能再次单击该按钮。要启动 `BackgroundWorker`, 可调用方法 `RunWorkerAsync()`。`BackgroundWorker` 使用线程池中的一个线程来计算。`RunWorkerAsync()` 需要将输入参数传送给 `DoWork` 事件的处理程序。

```

private void OnCalculate(object sender, EventArgs e)
{
    this.buttonCalculate.Enabled = false;
    this.textBoxResult.Text = String.Empty;
    this.buttonCancel.Enabled = true;
    this.progressBar.Value = 0;

    backgroundWorker.RunWorkerAsync(new CalcInput(
        int.Parse(this.textBoxX.Text), int.Parse(this.textBoxY.Text)));
}

```

方法 `OnDoWork()` 连接到 `BackgroundWorker` 控件的 `DoWork` 事件上。在 `DoWorkEventArgs` 中, 通过属性 `Argument` 接收输入参数。其执行代码模拟的功能需要一定的执行时间和 5 秒的睡眠时间。在睡眠时间过后, 将计算的结果写入 `DoEventArgs` 的 `Result` 属性。如果将计算和睡眠操作添加到 `OnCalculate()` 方法中, 在用户输入时, Windows 应用程序就不能获得用户输入。但是, 这里使用一个单独的线程, 用户界面仍是激活的。

```

private void OnDoWork(object sender, DoWorkEventArgs e)
{
    CalcInput input = (CalcInput)e.Argument;

    Thread.Sleep(5000);
    e.Result = input.x + input.y;
}

```

方法 `OnDoWork()` 完成后, `BackgroundWorker` 控件就引发 `RunWorkerCompleted` 事件。方法 `OnWorkCompleted()` 与这个事件相关。这里从 `RunWorkerCompletedEventArgs` 参数的 `Result` 属性中接收结果, 该结果写入文本框控件 `result` 中。在引发该事件时, `BackgroundWorker` 控件会把控制权交给创建它的线程, 所以不需要使用 Windows 窗体控件的 `Invoke` 方法, 而可以直接调用 Windows 窗体控件的属性和方法。

```

private void OnWorkCompleted(object sender, RunWorkerCompletedEventArgs e)
{
    this.textBoxResult.Text = e.Result.ToString();

    this.buttonCalculate.Enabled = true;
    this.buttonCancel.Enabled = false;
    this.progressBar.Value = 100;
}

```

现在可以测试应用程序, 看看计算过程是否独立于 UI 线程, UI 仍是激活的, 窗体可以四处移动。但是, 取消和进度条功能仍需要实现。

## 19.9.2 激活取消功能

要激活取消功能，以停止线程的运行，必须把 `BackgroundWorker` 控件的属性 `WorkerSupportsCancellation` 设置为 `true`。接着，实现与按钮 `buttonCancel` 的 `Click` 事件相关的 `OnCancel` 处理程序。`BackgroundWorker` 控件的 `CancelAsync()` 方法可以取消正在进行的异步任务。

```
private void OnCancel(object sender, EventArgs e)
{
    backgroundWorker.CancelAsync();
}
```

异步任务不会自动取消。在执行异步任务的 `OnDoWork()` 处理程序中，必须修改其实现代码，检查 `BackgroundWorker` 控件的属性 `CancellationPending`。这个属性在调用 `CancelAsync()` 方法时设置。如果要执行取消操作，就把 `DoWorkEventArgs` 的 `Cancel` 属性设置为 `true`，退出处理程序。

```
private void OnDoWork(object sender, DoWorkEventArgs e)
{
    CalcInput input = (CalcInput)e.Argument;

    for (int i = 0; i < 10; i++)
    {
        Thread.Sleep(500);

        if (backgroundWorker.CancellationPending)
        {
            e.Cancel = true;
            return;
        }
    }

    e.Result = input.x + input.y;
}
```

如果异步方法成功完成或被取消，就调用完成处理程序 `OnWorkCompleted()`。如果取消了该方法，就不能访问 `Result` 属性，因为这会抛出一个 `InvalidOperationException` 异常，并显示操作被取消的信息。所以必须检查 `RunWorkerCompletedEventArgs` 的 `Cancelled` 属性，并根据不同的情况执行不同的操作：

```
private void OnWorkCompleted(object sender, RunWorkerCompletedEventArgs e)
{
    if (e.Cancelled)
    {
        this.textBoxResult.Text = "Cancelled";
    }
    else
    {
        this.textBoxResult.Text = e.Result.ToString();
    }
    this.buttonCalculate.Enabled = true;
    this.buttonCancel.Enabled = false;
}
```

再次运行应用程序，就可以在用户界面上取消异步进程了。



### 19.9.3 激活进度功能

为了获得用户界面的进度信息，必须将 `BackgroundWorker` 控件的 `WorkerReportsProgress` 属性设置为 `true`。

在 `OnDoWork` 方法中，可以用 `ReportProgress()` 方法报告 `BackgroundWorker` 控件的进度。

```
private void OnDoWork(object sender, DoWorkEventArgs e)
{
    CalcInput input = (CalcInput)e.Argument;

    for (int i = 0; i < 10; i++)
    {
        Thread.Sleep(500);
        backgroundWorker.ReportProgress(i * 10);
        if (backgroundWorker.CancellationPending)
        {
            e.Cancel = true;
            return;
        }
    }

    e.Result = input.x + input.y;
}
```

方法 `ReportProgress()` 引发 `BackgroundWorker` 控件的 `ProgressChanged` 事件，这个事件会将控件改为 UI 线程。

在 `ProgressChanged` 事件中添加方法 `OnProgressChanged()`，在其实现代码中，给进度条控件设置一个从 `ProgressChangedEventArgs` 的 `ProgressPercentage` 属性中接收的新值。

```
private void OnProgressChanged(object sender, ProgressChangedEventArgs e)
{
    this.progressBar.Value = e.ProgressPercentage;
}
```

在 `OnWorkCompleted()` 事件处理程序中，进度条最终设置为 100%。

```
private void OnWorkCompleted(object sender, RunWorkerCompletedEventArgs e)
{
    if (e.Cancelled)
    {
        this.textBoxResult.Text = "Cancelled";
    }
    else
    {
        this.textBoxResult.Text = e.Result.ToString();
    }
    this.buttonCalculate.Enabled = true;
    this.buttonCancel.Enabled = false;
    this.progressBar.Value = 100;
}
```

图 19-4 是正在计算的应用程序。

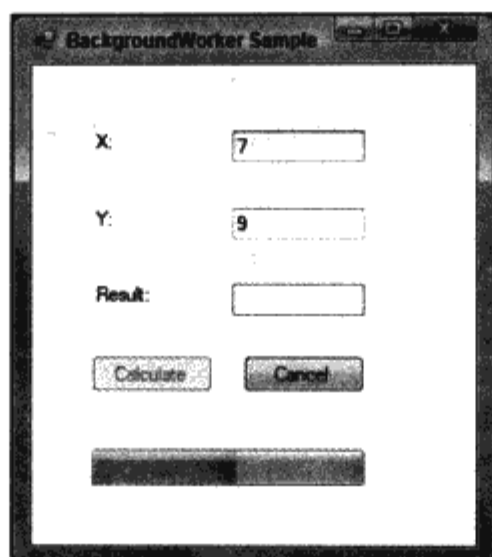


图 19-4

#### 19.9.4 创建基于事件的异步组件

要创建一个支持基于事件的异步模式的定制组件，需要做更多的工作。为了用一个简单的情形演示这个过程，类 `AsyncComponent` 只在某个时间段后返回一个转换过来的输入字符串，与前面的同步方法 `LongTask()` 相同。为了提供异步支持，公共接口提供了异步方法 `LongTaskAsync()` 和事件 `LongTaskCompleted`。这个事件的类型是 `LongTaskCompletedEventHandler`，定义了参数 `object sender` 和 `LongTaskCompletedEventArgs e`。 `LongTaskCompletedEventArgs` 是一个新类型，调用者可以从其中读取异步操作的结果。

另外，还需要一些帮助方法，如 `DoLongTask` 和 `CompletionMethod`，如下所述。

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Threading;

namespace Wrox.ProCSharp.Threading
{
    public delegate void LongTaskCompletedEventHandler(object sender,
        LongTaskCompletedEventArgs e);

    public partial class AsyncComponent : Component
    {
        private Dictionary< object, AsyncOperation > userStateDictionary =
            new Dictionary< object, AsyncOperation > ();
        private SendOrPostCallback onCompletedDelegate;

        public AsyncComponent()
        {
            InitializeComponent();
            InitializeDelegates();
        }

        public AsyncComponent(IContainer container)
        {
            container.Add(this);

            InitializeComponent();
            InitializeDelegates();
        }
    }
}
```

```

    }
    private void InitializeDelegates()
    {
        onCompletedDelegate = LongTaskCompletion;
    }

    public string LongTask(string input)
    {
        Console.WriteLine("LongTask started");
        Thread.Sleep(5000);
        Console.WriteLine("LongTask finished");
        return input.ToUpper();
    }

    public void LongTaskAsync(string input, object taskId)
    {
        //...
    }

    public event LongTaskCompletedEventHandler LongTaskCompleted;

    private void LongTaskCompletion(object operationState)
    {
        //...
    }

    protected void OnLongTaskCompleted(LongTaskCompletedEventArgs e)
    {
        //...
    }

    private delegate void LongTaskWorkHandler(string input,
        AsyncOperation asyncOp);
    // running in a background thread
    private void DoLongTask(string input, AsyncOperation asyncOp)
    {
        //...
    }

    private void CompletionMethod(string output, Exception ex,
        bool cancelled, AsyncOperation asyncOp)
    {
        //...
    }
}

public class LongTaskCompletedEventArgs : AsyncCompletedEventArgs
{
    //...
}

```

方法 `LongTaskAsync()` 需要异步启动同步的操作。如果组件允许同时启动几次异步任务，客户就需要能把不同的结果映射到启动的任务上。因此方法 `LongTaskAsync()` 的第二个参数必须是一个 `taskId`，客户机可以使用它映射结果。当然，在组件内部，必须记录任务 ID，才能映射结果。`.NET` 提供了类 `AsyncOperationManager`，来创建 `AsyncOperationObjects`，以跟踪操作的状态。类 `AsyncOperationManager` 有一个方法 `CreateOperation`，可以给它传送一个任务标识

符,该方法就返回一个 `AsyncOperation` 对象。这个操作保存为前面创建的字典 `userStateDictionary` 中的一项。

接着,创建一个 `LongTaskWorkHandler` 类型的委托,把方法 `DoLongTask()` 赋予委托实例。`BeginInvoke()` 是委托的一个方法,可以使用线程池中的一个线程异步启动方法 `DoLongTask()`。

```
public void LongTaskAsync(string input, object taskId)
{
    AsyncOperation asyncOp =
        AsyncOperationManager.CreateOperation(taskId);

    lock (userStateDictionary)
    {
        if (userStateDictionary.ContainsKey(taskId))
            throw new ArgumentException("taskId must be unique", "taskId");

        userStateDictionary[taskId] = asyncOp;
    }
    LongTaskWorkHandler longTaskDelegate = DoLongTask;
    longTaskDelegate.BeginInvoke(input, asyncOp, null, null);
}
```

委托类型 `LongTaskWorkHandler` 在 `AsyncComponent` 类中用私有访问修饰符定义,因为它不需要在外部使用。这个委托需要的参数是调用者的所有输入参数和 `AsyncOperation` 参数,用于获得状态,映射操作的结果。

```
private delegate void LongTaskWorkHandler(string input,
    AsyncOperation asyncOp);
```

方法 `DoLongTask()` 现在使用委托来异步调用。可以调用同步方法 `LongTask()`, 获得输出值。

在同步方法中可能出现的异常不应向上传递到后台线程,所以应捕获任何异常,并用 `Exception` 类型的变量 `e` 保存它。最后,调用 `CompletionMethod()` 方法,向调用者通报结果。

```
// running in a background thread
private void DoLongTask(string input, AsyncOperation asyncOp)
{
    Exception e = null;
    string output = null;
    try
    {
        output = LongTask(input);
    }
    catch (Exception ex)
    {
        e = ex;
    }
    this.CompletionMethod(output, e, false, asyncOp);
}
```

在 `CompletionMethod()` 方法的实现代码中,删除操作时清除了 `userStateDictionary`。`AsyncOperation` 对象的 `PostOperationCompleted()` 方法结束了异步操作的生存期,并使用 `onCompletedDelegate` 方法通知调用者。这个方法确保在线程上根据应用程序类型的需要调用委托。要给调用者通报信息,应创建一个 `LongTaskCompletedEventArgs` 类型的对象,传送给方法 `PostOperationCompleted()`。

```

private void CompletionMethod(string output, Exception ex,
    bool cancelled, AsyncOperation asyncOp)
{
    lock (userStateDictionary)
    {
        userStateDictionary.Remove(asyncOp.UserSuppliedState);
    }

    // results of the operation
    LongTaskCompletedEventArgs e = new LongTaskCompletedEventArgs(
        output, ex, cancelled, asyncOp.UserSuppliedState);

    asyncOp.PostOperationCompleted(onCompletedDelegate, e);
}
}

```

为了给调用者传送信息，类 `LongTaskCompletedEventArgs` 应派生于基类 `AsyncCompletedEventArgs`，并添加一个包含输出信息的属性。在构造函数中，调用基构造函数，来传送异常、取消和用户状态信息。

```

public class LongTaskCompletedEventArgs : AsyncCompletedEventArgs
{
    public LongTaskCompletedEventArgs(string output, Exception e,
        bool cancelled, object state)
        : base(e, cancelled, state)
    {
        this.output = output;
    }

    private string output;

    public string Output
    {
        get
        {
            RaiseExceptionIfNecessary();
            return output;
        }
    }
}

```

方法 `asyncOp.PostOperationCompleted()` 使用了 `onCompletedDelegate`。初始化这个委托以便引用方法 `LongTaskCompletion`。`LongTaskCompletion` 必须满足 `SendOrPostCallbackDelegate` 的参数要求。实现代码把参数转换为 `LongTaskCompletedEventArgs`，这是传送给 `PostOperationCompleted()` 方法的对象类型，再调用 `onLongTaskCompleted` 方法。

```

private void LongTaskCompletion(object operationState)
{
    LongTaskCompletedEventArgs e =
        operationState as LongTaskCompletedEventArgs;

    OnLongTaskCompleted(e);
}

```

接着，`onLongTaskCompleted` 方法触发事件 `LongTaskCompleted`，给调用者返回 `LongTaskCompletedEventArgs`。



```
protected void OnLongTaskCompleted(LongTaskCompletedEventArgs e)
{
    if (LongTaskCompleted != null)
    {
        LongTaskCompleted(this, e);
    }
}
```

创建了组件后，使用它是很容易的。事件 `LongTaskCompleted` 赋予方法 `Comp_LongTaskCompleted`，调用方法 `LongTaskAsync()`。在一个简单的控制台应用程序中，事件处理程序 `Comp_LongTaskCompleted` 从不是主线程的另一个线程中调用。(这不同于 Windows 窗体应用程序，如下所述)。

```
static void Main()
{
    Console.WriteLine("Main thread: {0}",
        Thread.CurrentThread.ManagedThreadId);

    AsyncComponent comp = new AsyncComponent();
    comp.LongTaskCompleted += Comp_LongTaskCompleted;

    comp.LongTaskAsync("input", 33);

    Console.ReadLine();
}

static void Comp_LongTaskCompleted(object sender,
    LongTaskCompletedEventArgs e)
{
    Console.WriteLine("completed, result: {0}, thread: {1}", e.Output,
        Thread.CurrentThread.ManagedThreadId);
}
```

在 Windows 窗体应用程序中，`SynchronizationContext` 设置为 `WindowsFormsSynchronizationContext`，因此在同一个线程中调用事件处理程序：

```
WindowsFormsSynchronizationContext syncContext =
    new WindowsFormsSynchronizationContext();
SynchronizationContext.SetSynchronizationContext(syncContext);
```

## 19.10 小结

本章介绍了如何通过 `System.Threading` 命名空间编写多线程应用程序。在应用程序中使用多线程要仔细规划。太多的线程会导致资源问题，线程不足又会使应用程序执行缓慢，执行效果也不好。

我们探讨了创建多线程的各种方法，例如使用委托、计时器、`ThreadPool` 和 `Thread` 类。还研究了各种同步技术，例如简单的 `lock` 语句、`Monitor`、`Semaphore` 和 `Event` 类。学习了如何用 `IAAsyncResult` 接口编写异步模式，以及基于事件异步模式。

.NET Framework 中的 `System.Threading` 命名空间提供了处理线程的多种方式，但 .NET Framework 并没有完成多线程中所有困难的任务。我们必须考虑线程的优先级和同步问题。本章讨论了这些问题，介绍了如何在 C# 应用程序中为它们编码。还论述了与死锁和竞态条件相关的问题。

如果要在 C# 应用程序中使用多线程功能，就必须仔细规划。

# 第20章

## 安 全 性

安全性有几个重要方面需要考虑。一是应用程序的用户，访问应用程序的是一个真正的用户，还是伪装成用户的某个人？如何确定这个用户是可以信任的？如本章所述，用户首先需要进行身份验证，再进行授权，以验证该用户是否可以使用他请求的资源。

在网络上存储或发送什么数据？有人可以通过网络嗅探器访问这些数据吗？这里，数据的加密是很重要的。

另一方面是应用程序本身。如何确定这个应用程序是可以信任的？应用程序来自于何处或者有什么凭证？在 Web 主机环境下，这是非常重要的。Web 主机提供程序不允许其用户访问系统上的所有资源。根据程序集的凭证，为应用程序使用不同的许可。

本章将讨论 .NET 中有助于管理安全的一些特性，其中包括 .NET 怎样避开有害的代码、怎样管理安全性策略，以及怎样编程访问安全子系统等。本章的主要内容如下：

- 身份验证和授权
- 加密
- 资源的访问控制
- 代码访问安全性
- 管理安全策略

### 20.1 身份验证和授权

身份验证是标识用户的过程，授权在验证了所标识用户可以访问特定资源之后进行。

#### 20.1.1 标识和 Principal

使用标识可以验证运行应用程序的用户。WindowsIdentity 类表示一个 Windows 用户。如果没有用 Windows 账户验证用户，也可以使用实现了 IIdentity 接口的其他类。通过这个接口可以访问用户名、该用户是否获得验证和验证类型等信息。

Principal 是一个包含用户标识和用户所属角色的对象。接口 IPrincipal 定义了返回 IIdentity 对象的属性 Identity 和方法 IsInRole，在该方法中，可以验证用户是否是指定角色的一个成员。角色是有相同安全权限的用户集合，是用户的管理单元。角色可以是 Windows 组或自己定义的一个字符串集合。

.NET 中的 Principal 类有 WindowsPrincipal 和 GenericPrincipal。还可以创建实现了 IPrincipal

接口的定制 `Principal` 类。

下面创建一个控制台应用程序，它可以访问某个应用程序中的 `principal`，以便访问底层的 Windows 账户。它需要引用 `System.Security.Principal` 和 `System.Threading` 命名空间。首先，必须指定 .NET 使用底层的 Windows 账户自动捕捉 `Principal`。因为从安全的角度考虑，.NET 不会自动填充线程的 `CurrentPrincipal` 属性。完成这项工作的代码如下所示：

```
using System;
using System.Security.Principal;
using System.Threading;

namespace Wrox.ProCSharp.Security
{
    class Program
    {
        static void Main()
        {
            AppDomain.CurrentDomain.SetPrincipalPolicy(
                PrincipalPolicy.WindowsPrincipal);
        }
    }
}
```

使用 `WindowsIdentity.GetCurrent()` 可以访问 Windows 账户的详细信息，但是，这个方法最好在只访问一次 `principal` 时使用。如果要多次访问 `Principal`，比较有效的方法是设置策略，让当前的线程能够访问 `principal`。当使用 `SetPrincipalPolicy` 方法时，指定当前线程中的 `principal` 保存一个 `WindowsIdentity` 对象。所有的标识类(例如 `WindowsIdentity`)都执行 `IIdentity` 接口，该接口包含三个属性(`AuthenticationType`、`IsAuthenticated` 和 `Name`)，可由所有的派生标识类执行。

下面添加一些代码，从 `Thread` 对象中访问 `principal` 的属性：

```
WindowsPrincipal principal =
    (WindowsPrincipal)Thread.CurrentPrincipal;
WindowsIdentity identity = (WindowsIdentity)principal.Identity;
Console.WriteLine("IdentityType: " + identity.ToString());
Console.WriteLine("Name: {0}", identity.Name);
Console.WriteLine("'Users'? : {0} ",
    principal.IsInRole("BUILTIN\\Users"));
Console.WriteLine("'Administrators'? {0}",
    principal.IsInRole(WindowsBuiltInRole.Administrator));
Console.WriteLine("Authenticated: {0}", identity.IsAuthenticated);
Console.WriteLine("AuthType: {0}", identity.AuthenticationType);
Console.WriteLine("Anonymous? {0}", identity.IsAnonymous);
Console.WriteLine("Token: {0}", identity.Token);
}
```

下面是从控制台应用程序中输出的结果。当然，由于机器的配置和与账户相关的角色有所不同，实际输出的结果也不尽相同：

```
IdentityType: System.Security.Principal.WindowsIdentity
Name: cnagel\christian
'Users'? : True
'Administrators'? : True
Authenticated: True
AuthType: NTLM
Anonymous?: False
Token: 368
```

很明显，如果能很容易访问当前用户及其角色的详细信息，然后使用那些信息决定允许或

拒绝用户执行某些动作,是非常有好处的。利用角色和 Windows 用户组,管理员可以完成使用标准用户管理工具所能完成的工作,这样,在用户的角色改变时,通常可以避免更改代码。下面将详细讨论角色。

### 20.1.2 角色

基于角色的安全性可以很好地解决资源的访问问题。例如,在金融行业中,员工的角色决定了他们能够访问的信息和能够执行的操作。

此外,基于角色的安全性也可以与 Windows 账户或定制的用户目录一起使用,以便对基于 Web 资源的访问进行管理。例如,Web 站点可以限制用户对其内容的访问,直到用户在那个站点注册为止,而且只有用户成为那个 Web 站点的付费订阅者之后,才能访问站点上的特殊内容。在众多的方法中,只有 ASP.NET 能更容易实现基于角色的安全性,因为许多代码都是以服务器为基础的。

例如,如果要执行一个需要验证的 Web 服务,可以使用 Windows 的账户子系统,编写一个 Web 方法。但是,要确保用户在访问该方法的功能之前,成为某一 Windows 用户组的成员。

设想有一个依赖于 Windows 账户的内联网应用程序。系统有一个 Manager 组和一个 Assistant 组,根据用户在公司中的角色,把用户分配到其中的一个组中。假设应用程序包含一个显示员工信息的特性,且只允许 Managers 组中的成员访问这个特性。很容易使用代码检查当前的用户是否是 Manager 组的成员,以此来决定是否允许用户访问该特性。

但是,如果以后决定重新安排账户组,并引进一个新组 Personnel,这个组中的成员也可以访问员工信息,就会出问题。此时就必须仔细检查,更新所有的代码,以包括这个新组的规则。

更好的解决方法是创建权限(例如 ReadEmployeeDetails),把所创建的权限赋予需要权限的组。如果代码使用了 ReadEmployeeDetails 权限,则更新应用程序以允许 Personnel 组中的成员访问员工信息就变得非常简单了,只需创建一个组,并把用户放到那个组中,然后把权限 ReadEmployeeDetails 赋予那个组即可。

### 20.1.3 声明基于角色的安全性

如同代码访问的安全性一样,可以使用命令式的请求,调用 IPincipal 类的 IsInRole()方法或使用属性执行基于角色的安全性请求(用户必须是 Administrator 组中的成员)。在类或方法的级别上,可以使用 [PrincipalPermission] 属性声明性地规定权限的需求,其代码如下所示:

```
using System;
using System.Security;
using System.Security.Principal;
using System.Security.Permissions;

namespace Wrox.ProCSharp.Security
{
    class Program
    {
        static void Main()
        {
            AppDomain.CurrentDomain.SetPrincipalPolicy(
                PrincipalPolicy.WindowsPrincipal);
        }
    }
}
```



```

    try
    {
        ShowMessage();
    }
    catch (SecurityException exception)
    {
        Console.WriteLine("Security exception caught (" +
            exception.Message + ")");
        Console.WriteLine("The current principal must be in the local"
            + "Users group");
    }
    Console.ReadLine();
}

[PrincipalPermission(SecurityAction.Demand,
    Role = "BUILTIN\\Users")]
static void ShowMessage()
{
    Console.WriteLine("The current principal is logged in locally ");
    Console.WriteLine("(they are a member of the local Users group)");
}
}
}

```

除非在 Windows 本地 Users 组的用户环境中执行应用程序，否则 ShowMessage()方法将产生一个异常。对于 Web 应用程序而言，运行 ASP.NET 代码的账户必须处于组中，但在实际应用中一定不会把这个账户添加到管理员组中。

如果使用本地 User 组中的账户运行上面的代码，则输出结果如下所示：

```

The current principal is logged in locally
(member of the local Users group)

```

#### 20.1.4 客户应用程序服务

Visual Studio 2008 很容易给 ASP.NET Web 应用程序使用以前建立的身份验证服务。使用这个服务可以给 Windows 和 Web 应用程序使用相同的身份验证机制。这是一个提供程序模型，它主要基于 System.Web.Security 命名空间中的 Membership 和 Roles 类。使用 Membership 类可以验证、创建、删除和查找用户，改变密码以及与用户相关的许多其他操作。使用 Roles 类可以添加和删除角色，给用户获取角色，改变用户的角色。角色和用户的存储位置取决于提供程序。ActiveDirectoryMembershipProvider 访问 Active Directory 中的用户和角色，SqlMembershipProvider 使用 SQL Server 数据库。对于客户应用程序服务，.NET 3.5 有两个新的提供程序：ClientFormsAuthenticationMembershipProvider 和 ClientWindows AuthenticationMembershipProvider。

下面使用客户应用程序服务和 Forms 验证模式。为此，首先需要启动一个应用程序服务器，才能在 Windows 窗体或 WPF 中使用这个服务。

##### 1. 应用程序服务

要使用客户应用程序服务，可以创建一个 ASP.NET Web 服务项目，它提供了应用程序服务。

该项目需要一个成员提供程序。这里的示例代码定义了类 SampleMembershipProvider，它派生于基类 MembershipProvider。必须重写基类中的所有抽象方法。对于登录，只需实现



`ValidateUser` 方法。其他方法都可以用属性 `ApplicationName` 抛出一个 `NotSupportedException` 异常。这里的示例代码使用包含用户名和密码的 `Dictionary<string, string>`。当然，也可以改为使用自己的实现代码，如从数据库中读取用户名和密码。

```
using System;
using System.Collections.Generic;
using System.Collections.Specialized;
using System.Web.Security;

namespace Wrox.ProCSharp.Security
{
    public class SampleMembershipProvider : MembershipProvider
    {
        private Dictionary < string, string > users = null;
        internal static string ManagerUserName = "Manager".ToLowerInvariant();
        internal static string EmployeeUserName =
            "Employee".ToLowerInvariant();
        public override void Initialize(string name, NameValueCollection config)
        {
            users = new Dictionary < string, string > ();
            users.Add(ManagerUserName, "secret@Pa$$w0rd");
            users.Add(EmployeeUserName, "s0me@Secret");

            base.Initialize(name, config);
        }
        public override string ApplicationName
        {
            get
            {
                throw new NotImplementedException();
            }
            set
            {
                throw new NotImplementedException();
            }
        }

        // override abstract Membership members
        // ...

        public override bool ValidateUser(string username, string password)
        {
            if (users.ContainsKey(username.ToLowerInvariant()))
            {
                return password.Equals(users[username.ToLowerInvariant()]);
            }
            return false;
        }
    }
}
```

为了使用角色，还需要实现一个角色提供程序。类 `SampleRoleProvider` 派生于基类 `RoleProvider`，实现了方法 `GetRolesForUser()` 和 `IsUserInRole()`：

```
using System;
using System.Collections.Specialized;
using System.Web.Security;
namespace Wrox.ProCSharp.Security
```

```

{
    public class SampleRoleProvider : RoleProvider
    {
        internal static string ManagerRoleName =
            "Manager".ToLowerInvariant();
        internal static string EmployeeRoleName =
            "Employee".ToLowerInvariant();

        public override void Initialize(string name,
            NameValueCollection config)
        {
            base.Initialize(name, config);
        }

        public override void AddUsersToRoles(string[] usernames,
            string[] roleNames)
        {
            throw new NotImplementedException();
        }

        //... override abstract RoleProvider members
        public override string[] GetRolesForUser(string username)
        {
            if (string.Compare(username,
                SampleMembershipProvider.ManagerUserName, true) == 0)
            {
                return new string[] { ManagerRoleName };
            }
            else if (string.Compare(username,
                SampleMembershipProvider.EmployeeUserName, true) == 0)
            {
                return new string[] { EmployeeRoleName };
            }
            else
            {
                return new string[0];
            }
        }

        public override bool IsUserInRole(string username, string roleName)
        {
            string[] roles = GetRolesForUser(username);
            foreach (string role in roles)
            {
                if (string.Compare(role, roleName, true) == 0)
                {
                    return true;
                }
            }
            return false;
        }
    }
}

```

身份验证服务必须在 Web.Config 文件中配置。在产品系统上,从安全的角度来看,最好用包含应用程序服务的服务器配置 SSL:

```

< system.web.extensions >
  < scripting >
    < webServices >

```

```

    < authenticationService enabled="true" requireSSL="false"/ >
    < roleService enabled="true"/ >
  < /webServices >
< /scripting >
< /system.web.extensions >

```

在<system.web>段中, membership 和 rolemanager 元素必须配置为引用实现了成员和角色提供程序的类:

```

< system.web >
  < membership defaultProvider="SampleMembershipProvider" >
    < providers >
      < add name="SampleMembershipProvider"
        type="Wrox.ProCSharp.Security.SampleMembershipProvider"/ >
    < /providers >
  < /membership >
  < roleManager enabled="true" defaultProvider="SampleRoleProvider" >
    < providers >
      < add name="SampleRoleProvider"
        type="Wrox.ProCSharp.Security.SampleRoleProvider"/ >
    < /providers >
  < /roleManager >

```

在调试时,可以用项目属性的 Web 选项卡指定端口号和虚拟路径。示例应用程序使用端口 55555 和虚拟路径/AppServices。如果使用其他值,就需要修改客户应用程序的配置。

现在应用程序服务可以在客户应用程序中使用了。

## 2. 客户应用程序

在客户应用程序中使用 WPF。Windows 窗体的使用方式相同。Visual Studio 2008 有一个新的项目设置 Services, 它允许使用客户应用程序服务。这里可以设置 Forms 验证模式, 把身份验证和角色服务的位置设置为前面定义的地址 <http://localhost:55555/AppServices>。在项目配置中, 只需引用程序集 System.Web 和 System.Web.Extensions, 修改应用程序配置文件, 把成员和角色提供程序配置为使用 ClientFormsAuthenticationMembershipProvider 和 ClientRoleProvider 以及这些提供程序使用的 Web 服务地址。

```

< ?xml version="1.0" encoding="utf-8"? >
< configuration >
  < system.web >
    < membership defaultProvider=
      "ClientAuthenticationMembershipProvider" >
      < providers >
        < add name="ClientAuthenticationMembershipProvider"
          type="System.Web.ClientServices.Providers.
            ClientFormsAuthenticationMembershipProvider,
            System.Web.Extensions, Version=3.5.0.0, Culture=neutral,
            PublicKeyToken=31bf3856ad364e35" serviceUri=
              "http://localhost:55555/AppServices
                /Authentication_JSON_AppService.axd" / >
      < /providers >
    < /membership >
    < roleManager defaultProvider="ClientRoleProvider" enabled="true" >
      < providers >
        < add name="ClientRoleProvider"
          type="System.Web.ClientServices.Providers.ClientRoleProvider,

```

```

        System.Web.Extensions, Version=3.5.0.0, Culture=neutral,
        PublicKeyToken=31bf3856ad364e35" serviceUri=
        "http://localhost:55555/AppServices/Role_JSON_AppService.axd"
        cacheTimeout="86400" / >
    < /providers >
< /roleManager >
< /system.web >
< /configuration >

```

Windows 应用程序仅使用标签、文本框、密码框和按钮控件，如图 20-1 所示。只有登录成功，才会显示标签的内容 User Validated。



图 20-1

Button.Click 事件的处理程序调用 Membership 类的 ValidateUser() 方法。由于配置了提供程序 ClientAuthenticationMembershipProvider，所以该提供程序调用 Web 服务，接着调用 SampleMembershipProvider 类的 ValidateUser() 方法，验证登录是否成功。如果成功，标签 labelValidatedInfo 就可见，否则弹出一个消息框：

```

private void buttonLogin_Click(object sender, RoutedEventArgs e)
{
    try
    {
        if (Membership.ValidateUser(textUsername.Text,
            textPassword.Password))
        {
            // user validated!
            labelValidatedInfo.Visibility = Visibility.Visible;
        }
        else
        {
            MessageBox.Show("Username or password not valid",
                "Client Authentication Services", MessageBoxButton.OK,
                MessageBoxImage.Warning);
        }
    }
    catch (WebException ex)
    {
        MessageBox.Show(ex.Message, "Client Application Services",
            MessageBoxButton.OK, MessageBoxImage.Error);
    }
}

```

## 20.2 加密

敏感数据应得到保护，使未授权的用户不能读取它们。这对于在网络中传送的数据和存储在某处的数据都有效。可以用对称或不对称密钥来加密这些数据。

在对称加密中，可以使用同一个密钥进行加密和解密。但在不对称的加密中，加密和解密使用不同的密钥：公钥/私钥。如果使用一个公钥进行加密，就应使用对应的私钥进行解密，而不是使用公钥解密。同样，如果使用一个私钥加密，就应使用对应的公钥解密，而不是使用私钥解密。

公钥/私钥总是成对创建。公钥可以由任何人使用，甚至可以放在 Web 站点上，但私钥必须安全地加锁。下面看看使用公钥和私钥的例子。

如果 Alice 给 Bob 发了一封电子邮件，如图 20-2 所示，Alice 希望能保证除了 Bob 外，其他人都不能阅读该邮件，所以使用 Bob 的公钥。信息是使用 Bob 的公钥加密的。Bob 打开该邮件，并使用他秘密存储的私钥解密。这种方式可以保证除了 Bob 外，其他人都不能阅读 Alice 的邮件。

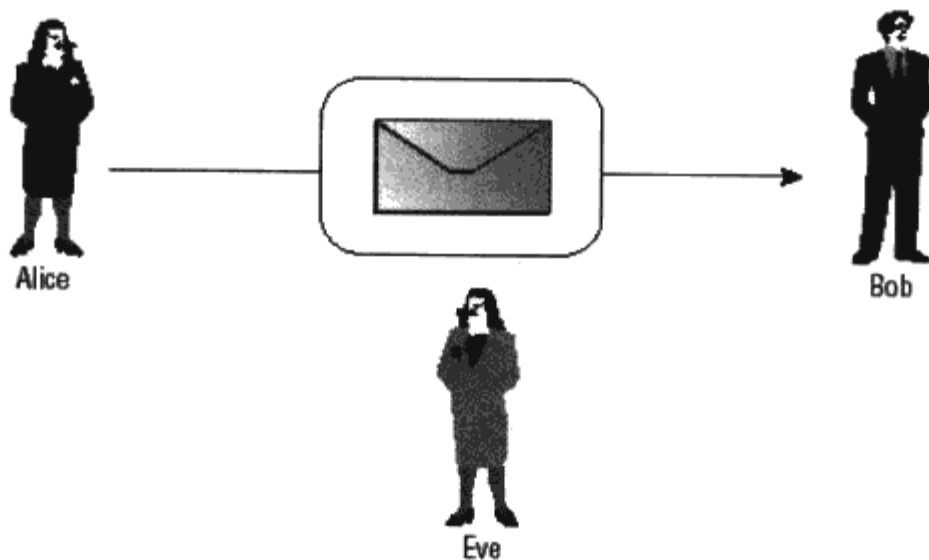


图 20-2

但这还有一个问题：Bob 不能确保邮件是 Alice 发送来的。Eve 可以使用 Bob 的公钥加密发送给 Bob 的邮件。我们把这个规则扩展一下。下面再次从 Alice 给 Bob 发电子邮件开始。在 Alice 使用 Bob 的公钥加密邮件之前，她添加了自己的签名，再使用自己的私钥加密该签名。然后使用 Bob 的公钥加密邮件。这样就保证了除 Bob 外，其他人都不能阅读该邮件。在 Bob 解密邮件时，检测到一个加密的签名。这个签名可以使用 Alice 的公钥来解密。而 Bob 可以访问 Alice 的公钥，因为这个密钥是公钥。在解密了签名后，Bob 就可以确定是 Alice 发送了电子邮件。

使用对称密钥的加密和解密算法比使用非对称密钥快得多。对称密钥的问题是密钥必须以安全的方式互换。在网络通信中，一种方式是先使用非对称的密钥进行互换，再使用对称密钥加密在线上发送的数据。

在 .NET Framework 中，可以使用 `System.Security.Cryptography` 命名空间中的类来加密。它实现了几个对称和非对称算法。有几个不同的算法类用于不同的目的。.NET 3.5 中的一些新类以 Cng 作为前缀或后缀。Cng 表示 Cryptography Next Generation，用于 Windows Vista 和 Windows



Server 2008。这个 API 可以使用基于提供程序的模型，编写独立于算法的程序。如果使用的是 Windows Server 2003，就需要注意使用了什么加密类。

表 20-1 列出了 System.Security.Cryptography 命名空间中的加密类及其功能。没有 Cng、Managed 或 CryptoServiceProvider 后缀的类是抽象基类，例如 MD5。Managed 后缀表示这个算法用托管代码实现，其他类可能封装了内部的 Windows API 调用。CryptoService- Provider 后缀用于实现了抽象基类的类， Cng 后缀用于利用新 Cryptography CNG API 的类，它只能用于 Windows Vista 和 Windows Server 2008。

表 20-1

类别	类	说 明
散列	MD5, MD5Cng SHA1, SHA1Managed, SHA1Cng SHA256, SHA256Managed, SHA256Cng SHA384, SHA384Managed, SHA384Cng SHA512, SHA512Managed, SHA512Cng	散列算法的目标是从任意长度的二进制字符串中创建一个长度固定的散列值。这些算法和数字签名一起用于保证数据的完整性。如果再次散列相同的二进制字符串，会返回相同的散列结果。MD5(Message Digest Algorithm 5)是由 RSA 实验室开发的，比 SHA1 快。SHA1 在抵御暴力攻击方面比较强大。SHA 算法是由美国国家安全局(NSA)设计的。MD5 使用 128 位的散列值，SHA1 使用 160 位。其他 SHA 算法在其名称中包含了散列长度。SHA512 是这些算法中最强大的，其散列长度为 512 位，也是最慢的
对称	DES, DESCryptoServiceProvider TripleDES, TripleDESCryptoServiceProvider Aes, AesCryptoServiceProvider, AesManaged RC2, RC2CryptoServiceProvider Rijandel, RijandelManaged	对称密钥算法使用相同的密钥进行数据的加密和解密。DES(Data Encryption Standard)现在认为是不安全的，因为它只使用 56 位的密钥，可以在不超过 24 小时的时间内破解。Triple-DES 是 DES 的继任者，其密钥的长度是 168 位，但它提供的有效安全性只有 112 位。AES(Advanced Encryption Standard)的密钥长度是 128、192 或 256 位。Rijandel 非常类似于 AES，只是在密钥长度方面的选项较多。AES 是美国政府采用的加密标准
非对称	DSA, DSACryptoServiceProvider ECDsa, ECDsaCng ECDiffieHellman, ECDiffieHellmanCng RSA, RSACryptoServiceProvider	非对称算法使用不同的密钥进行加密和解密。RSA(Rivest, Shamir, Adleman)是第一个用于签名和加密的算法。这个算法广泛用于电子商务协议。 DSA(Digital Signature Algorithm)是数字签名的一个美国联邦政府标准。 ECDSA(Elliptic Curve DSA)和 ECDiffieHellman 使用基于椭圆曲线组的算法。这些算法比较安全，且使用较短的密钥长度。例如，DSA 的密钥长度为 1024 位，其安全性类似于 160 位的 ECDSA。因此，ECDSA 比较快。 ECDiffieHellman 算法用于以安全的方式在公共信道中互换私钥

下面用例子说明这些算法如何编程使用。

### 20.2.1 签名

第一个例子演示如何使用 ECDSA 算法进行签名。Alice 创建了一个签名，它用 Alice 的私钥加密，可以使用 Alice 的公钥访问。因此保证该签名来自于 Alice。

首先，看看 Main()方法中的主要步骤：创建 Alice 的密钥，给字符串 Alice 签名，最后使用公钥验证该签名是否真的来自于 Alice。要签名的消息使用 Encoding 类转换为一个字节数组。要把加密的签名写入控制台，包含该签名的字节数组应使用 Convert.ToBase64-String()方法转换为一个字符串。

#### 提示：

千万不要使用 Encoding 类把加密的数据转换为字符串。Encoding 类验证和转换 Unicode 不允许使用的无效值，因此把字符串转换回字节数组会得到另一个结果。

```
using System;
using System.Security.Cryptography;
using System.Text;

namespace Wrox.ProCSharp.Security
{
    class Program
    {
        internal static CngKey aliceKeySignature;
        internal static byte[] alicePubKeyBlob;

        static void Main()
        {
            CreateKeys();

            byte[] aliceData = Encoding.UTF8.GetBytes("Alice");
            byte[] aliceSignature = CreateSignature(aliceData,
                aliceKeySignature);
            Console.WriteLine("Alice created signature: {0}",
                Convert.ToBase64String(aliceSignature));
            if (VerifySignature(aliceData, aliceSignature, alicePubKeyBlob))
            {
                Console.WriteLine("Alice signature verified successfully");
            }
        }
    }
}
```

CreateKeys()方法为 Alice 创建新的密钥对。这个密钥对存储在一个静态字段中，所以可以在其他方法中访问。CngKey 类的 Create()方法把算法作为一个参数，为算法定义密钥对。在 Export()方法中，导出密钥对中的公钥。这个公钥可以提供给 Bob，来验证签名。Alice 保留其私钥。除了使用 CngKey 类创建密钥对之外，还可以打开存储在密钥存储区中的已有密钥。通常 Alice 在其私有存储区中有一个证书，其中包含了一个密钥对，该存储区可以用 CngKey.Open()访问。

```
static void CreateKeys()
{
    aliceKeySignature = CngKey.Create(CngAlgorithm.ECDsaP256);
    alicePubKeyBlob = aliceKeySignature.Export(
```

```
CngKeyBlobFormat.GenericPublicBlob);
}
```

有了密钥对，Alice 就可以使用 `ECDsaCng` 类创建签名了。这个类的构造函数从 Alice 那里接收包含公钥和私钥的 `CngKey`。再使用私钥，通过 `SignData()` 方法给数据签名。

```
static byte[] CreateSignature(byte[] data, CngKey key)
{
    ECDsaCng signingAlg = new ECDsaCng(key);
    byte[] signature = signingAlg.SignData(data);
    signingAlg.Clear();

    return signature;
}
```

要验证签名是否真的来自于 Alice，Bob 使用 Alice 的公钥检查签名。包含公钥 blob 的字节数组可以用静态方法 `Import()` 导入 `CngKey` 对象。然后使用 `ECDsaCng` 类，调用 `VerifyData()` 验证签名。

```
static bool VerifySignature(byte[] data, byte[] signature,
    byte[] pubKey)
{
    bool retValue = false;
    using (CngKey key = CngKey.Import(pubKey,
        CngKeyBlobFormat.GenericPublicBlob))
    {
        ECDsaCng signingAlg = new ECDsaCng(key);
        retValue = signingAlg.VerifyData(data, signature);
        signingAlg.Clear();
    }
    return retValue;
}
```

### 20.2.2 密钥的互换和安全传送

下面是一个比较复杂的例子，它使用 **Diffie Hellman** 算法互换一个对称密钥，进行安全的传送。在 `Main()` 方法中，可以看到其主要功能。Alice 创建了一个加密的消息，把它发送给 Bob。在此之前，要先为 Alice 和 Bob 创建密钥对。Bob 只能访问 Alice 的公钥，Alice 也只能访问 Bob 的公钥。

```
using System;
using System.IO;
using System.Security.Cryptography;
using System.Text;

namespace Wrox.ProCSharp.Security
{
    class Program
    {
        static CngKey aliceKey;
        static CngKey bobKey;
        static byte[] alicePubKeyBlob;
        static byte[] bobPubKeyBlob;
```

```

static void Main()
{
    CreateKeys();
    byte[] encryptedData = AliceSendsData("secret message");
    BobReceivesData(encryptedData);
}

```

在 `CreateKeys()` 方法的实现代码中，创建了密钥，它们用于 EC Diffie Hellman 256 算法。

```

private static void CreateKeys()
{
    aliceKey = CngKey.Create(CngAlgorithm.ECDiffieHellmanP256);
    bobKey = CngKey.Create(CngAlgorithm.ECDiffieHellmanP256);
    alicePubKeyBlob = aliceKey.Export(CngKeyBlobFormat.EccPublicBlob);
    bobPubKeyBlob = bobKey.Export(CngKeyBlobFormat.EccPublicBlob);
}

```

在 `AliceSendsData()` 方法中，包含文本字符的字符串使用 `Encoding` 类转换为一个字节数组。创建一个 `ECDiffieHellmanCng` 对象，用 Alice 的密钥对初始化它。Alice 调用 `DeriveKeyMaterial()` 方法，使用其密钥对和 Bob 的公钥创建一个对称密钥。返回的对称密钥使用对称算法 AES 加密数据。`AesCryptsServiceProvider` 需要密钥和一个初始化矢量(IV)。IV 是从 `GenerateIV()` 方法中动态生成的，对称密钥用 EC Diffie Hellman 算法互换，还必须互换 IV。从安全的角度来看，在网络上传送未加密的 IV 是可行的——只是密钥互换必须是安全的。IV 存储为内存流中的第一个内容，其后是加密的数据，其中，`CryptoStream` 类使用 `AesCryptsServiceProvider` 类创建的 `encryptor`。在访问内存流中的加密数据之前，必须关闭加密流。否则，加密数据就会丢失最后的位。

```

private static byte[] AliceSendsData(string message)
{
    Console.WriteLine("Alice sends message: {0}", message);
    byte[] rawData = Encoding.UTF8.GetBytes(message);
    byte[] encryptedData = null;

    ECDiffieHellmanCng aliceAlgorithm = new ECDiffieHellmanCng(aliceKey);
    using (CngKey bobPubKey = CngKey.Import(bobPubKeyBlob,
        CngKeyBlobFormat.EccPublicBlob))
    {
        byte[] symmKey = aliceAlgorithm.DeriveKeyMaterial(bobPubKey);
        Console.WriteLine("Alice creates this symmetric key with " +
            "Bobs public key information: {0}",
            Convert.ToBase64String(symmKey));

        AesCryptoServiceProvider aes = new AesCryptoServiceProvider();
        aes.Key = symmKey;
        aes.GenerateIV();
        using (ICryptoTransform encryptor = aes.CreateEncryptor())
        using (MemoryStream ms = new MemoryStream())
        {
            // create CryptoStream and encrypt data to send
            CryptoStream cs = new CryptoStream(ms, encryptor,
                CryptoStreamMode.Write);

            // write initialization vector not encrypted
            ms.Write(aes.IV, 0, aes.IV.Length);
            cs.Write(rawData, 0, rawData.Length);
            cs.Close();
        }
    }
}

```

```

        encryptedData = ms.ToArray();
    }
    aes.Clear();
}
Console.WriteLine("Alice: message is encrypted: {0}",
    Convert.ToBase64String(encryptedData)); ;
Console.WriteLine();
return encryptedData;
}

```

Bob 从 `BobReceivesData()` 方法的参数中接收加密数据。首先，必须读取未加密的初始化矢量。AesCryptoServiceProvider 类的 `BlockSize` 属性返回块的位数。给位数除以 8，就可以计算出字节数。最快的方式是把数据偏移 3 位。偏移 1 位就是除以 2，偏移 2 位就是除以 4，偏移 3 位就是除以 8。在 for 循环中，包含未加密 IV 的原字节数组的前几个字节写入数组 `iv`。接着用 Bob 的密钥对实例化一个 `ECDiffieHellmanCng` 对象。使用 Alice 的公钥，从 `DeriveKeyMaterial()` 方法中返回对称密钥。比较 Alice 和 Bob 创建的对称密钥，可以看出所创建的密钥值是相同的。使用这个对称密钥和初始化矢量，来自 Alice 的消息就可以用 AesCryptoServiceProvider 类解密。

```

private static void BobReceivesData(byte[] encryptedData)
{
    Console.WriteLine("Bob receives encrypted data");
    byte[] rawData = null;

    AesCryptoServiceProvider aes = new AesCryptoServiceProvider();

    int nBytes = aes.BlockSize >> 3;
    byte[] iv = new byte[nBytes];
    for (int i = 0; i < iv.Length; i++)
        iv[i] = encryptedData[i];

    ECDiffieHellmanCng bobAlgorithm = new ECDiffieHellmanCng(bobKey);

    using (CngKey alicePubKey = CngKey.Import(alicePubKeyBlob,
        CngKeyBlobFormat.EccPublicBlob))
    {
        byte[] symmKey = bobAlgorithm.DeriveKeyMaterial(alicePubKey);
        Console.WriteLine("Bob creates this symmetric key with " +
            "Alices public key information: {0}",
            Convert.ToBase64String(symmKey));

        aes.Key = symmKey;
        aes.IV = iv;

        using (ICryptoTransform decryptor = aes.CreateDecryptor())
        using (MemoryStream ms = new MemoryStream())
        {
            CryptoStream cs = new CryptoStream(ms, decryptor,
                CryptoStreamMode.Write);
            cs.Write(encryptedData, nBytes, encryptedData.Length - nBytes);
            cs.Close();

            rawData = ms.ToArray();

            Console.WriteLine("Bob decrypts message to: {0}",
                Encoding.UTF8.GetString(rawData));
        }
        aes.Clear();
    }
}

```



```
}  
}  
  
运行应用程序，会在控制台上看到如下输出。来自 Alice 的消息被加密，Bob 用安全互换  
的对称密钥解密它。
```

```
Alice sends message: secret message  
Alice creates this symmetric key with Bobs public key information:  
5NWat8AemzFCYo1IIae9S3Vn4AXyai4aL8ATFo41vbw=  
Alice: message is encrypted: 3C5U9CpYxnoFTk3Ew2V0T5Po0Jgryc5R7Te8ztau5N0=  
  
Bob receives encrypted message  
Bob creates this symmetric key with Alices public key information:  
5NWat8AemzFCYo1IIae9S3Vn4AXyai4aL8ATFo41vbw=  
Bob decrypts message to: secret message
```

20.3 资源的访问控制

在操作系统中，资源，如文件和注册表键，以及命名通道的句柄，都是使用访问控制表来保护的。图 20-3 显示了这个结构。资源有一个关联的安全性描述器。安全性描述器包含了资源的拥有者信息，引用了两个访问控制表：自定访问控制表(DACL)和系统访问控制表(SACL)。DACL 确定了谁有访问权，谁没有访问权；SACL 确定了安全事件日志的审核规则。ACL 包含一个访问控制项(ACE)的列表。ACE 包含类型、安全标识符和权限。在 DACL 中，ACE 的类型可以是允许访问或拒绝访问。可以用文件设置和获得的权限是创建、读取、写入、删除、修改、改变许可和获得拥有权。

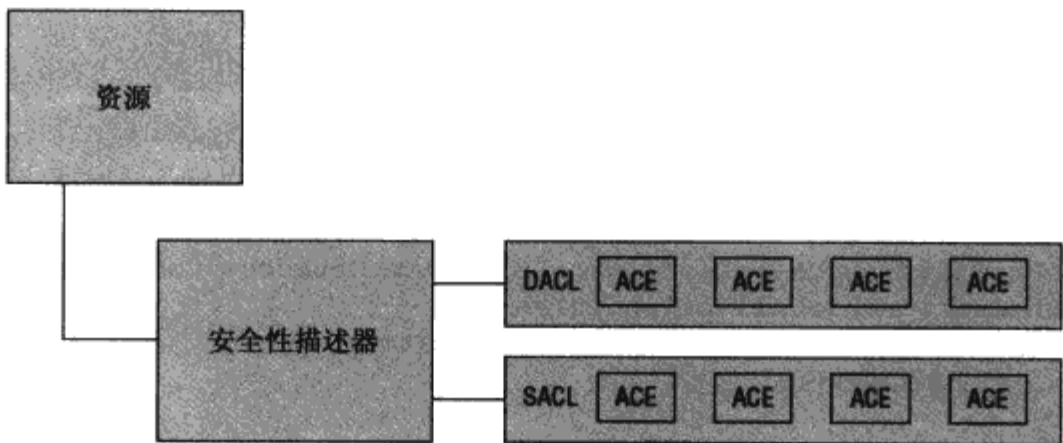


图 20-3

读取和修改访问控制的类在命名空间 System.Security.AccessControl 中。  
下面的程序演示了如何从文件中读取访问控制表。

FileStream 类定义了 GetAccessControl()方法，它返回一个 FileSecurity 对象。FileSecurity 是一个 .NET 类，表示文件的安全描述符。FileSecurity 派生于基类 ObjectSecurity、CommonObjectSecurity、NativeObjectSecurity 和 FileSystemSecurity。其他表示安全描述符的类有 CryptoKeySecurity、EventWaitHandleSecurity、MutexSecurity、RegistrySecurity、SemaphoreSecurity、PipeSecurity 和 ActiveDirectorySecurity。所有这些对象都可以使用访问控制表保护。一般情况下，对应的 .NET 类定义了 GetAccessControl()方法，返回相应的安全类；例

如, `Mutex.GetAccessControl()`方法返回一个 `MutexSecurity`, `PipeStream.GetAccessControl()`方法返回一个 `PipeSecurity`。

`FileSecurity` 类定义了读取和修改 DACL 和 SACL 的方法。`GetAccessRules()`方法以 `AuthorizationRuleCollection` 类的形式返回 DACL。要访问 SACL,可以使用方法 `GetAuditRules()`。

在 `GetAccess()`方法中,可以确定是否应使用继承的访问规则(不仅仅是用对象直接定义的访问规则)。最后一个参数定义了应返回的安全标识符的类型。这个类型必须派生于基类 `IdentityReference`。可能的类型有 `NTAccount` 和 `SecurityIdentifier`。这两个类都表示用户或组。`NTAccount` 类按名称查找安全性对象, `SecurityIdentifier` 类按唯一的安全标识符查找安全性对象。

返回的 `AuthorizationRuleCollection` 包含 `AuthorizationRule` 对象。`AuthorizationRule` 是 ACE 的 .NET 表示。在这里的例子中,访问一个文件,所以 `AuthorizationRule` 可以强制转换为 `FileSystemAccessRule`。在其他资源的 ACE 中,存在不同的 .NET 表示,例如 `MutexAccessRule` 和 `PipeAccessRule`。在 `FileSystemAccessRule` 类中, `AccessControlType`、`FileSystemRights` 和 `IdentityReference` 返回 ACE 的信息。

```
using System;
using System.IO;
using System.Security.AccessControl;
using System.Security.Principal;

namespace Wrox.ProCSharp.Security
{
    class Program
    {
        static void Main(string[] args)
        {
            string filename = null;
            if (args.Length == 0)
                return;

            filename = args[0];

            FileStream stream = File.Open(filename, FileMode.Open);
            FileSecurity securityDescriptor = stream.GetAccessControl();
            AuthorizationRuleCollection rules =
                securityDescriptor.GetAccessRules(true, true,
                    typeof(NTAccount));

            foreach (AuthorizationRule rule in rules)
            {
                FileSystemAccessRule fileRule = rule as FileSystemAccessRule;
                Console.WriteLine("Access type: {0}", fileRule.AccessControlType);
                Console.WriteLine("Rights: {0}", fileRule.FileSystemRights);
                Console.WriteLine("Identity: {0}",
                    fileRule.IdentityReference.Value);
                Console.WriteLine();
            }
        }
    }
}
```

运行应用程序,传送一个文件名,就可以看到文件的访问控制表。这里的输出列出了管理员和系统的全部权限、验证用户的修改权限和属于组 `Users` 的所有用户的读取和执行权限。

```
Access type: Allow
Rights: FullControl
Identity: BUILTIN\Administrators
```

```
Access type: Allow
Rights: FullControl
Identity: NT AUTHORITY\SYSTEM
```

```
Access type: Allow
Rights: Modify, Synchronize
Identity: NT AUTHORITY\Authenticated Users
```

```
Access type: Allow
Rights: ReadAndExecute, Synchronize
Identity: BUILTIN\Users
```

设置访问权限非常类似于访问权限的读取。要设置访问权限，几个可以得到保护的资源类提供了 `SetAccessControl()` 和 `ModifyAccessControl()` 方法。这里的示例代码调用 `File` 类的 `SetAccessControl()` 方法，修改了文件的访问控制表。给这个方法传送一个 `FileSecurity` 对象。`FileSecurity` 对象填充了 `FileSystemAccessRule` 对象。这里列出的访问规则拒绝 `Sales` 组的写入访问权限，给 `Everyone` 组提供了读取访问权限，给 `Developers` 组提供了全部访问权限。

#### 提示：

只有定义了 `Windows` 组 `Sales` 和 `Developers`，这个程序才能在系统上运行。可以修改程序，使用自己环境下的可用组。

```
private static void WriteAcl(string filename)
{
    NTAccount salesIdentity = new NTAccount("Sales");
    NTAccount developersIdentity = new NTAccount("Developers");
    NTAccount everyoneIdentity = new NTAccount("Everyone");

    FileSystemAccessRule salesAce = new FileSystemAccessRule(
        salesIdentity, FileSystemRights.Write, AccessControlType.Deny);
    FileSystemAccessRule everyoneAce = new FileSystemAccessRule(
        everyoneIdentity, FileSystemRights.Read,
        AccessControlType.Allow);
    FileSystemAccessRule developersAce = new FileSystemAccessRule(
        developersIdentity, FileSystemRights.FullControl,
        AccessControlType.Allow);

    FileSecurity securityDescriptor = new FileSecurity();
    securityDescriptor.SetAccessRule(everyoneAce);
    securityDescriptor.SetAccessRule(developersAce);
    securityDescriptor.SetAccessRule(salesAce);

    File.SetAccessControl(filename, securityDescriptor);
}
```

#### 提示：

打开 `Properties`，在 `Windows` 资源管理器中选择一个文件，选择 `Security` 选项卡，列出访问控制表，就可以验证访问权限。

## 20.4 代码访问的安全性

代码访问安全性的重要性是什么？在基于角色的安全性中，可以定义用户允许做什么。代码访问安全性指定了代码能做什么，它取决于代码的凭证——代码来自于何处？根据代码的来源，应用不同的权限。当然，仍应用基于代码的安全性。代码不能执行不允许用户执行的操作。

代码访问的安全性是运行库的一个特性，它根据代码的信任级别来管理代码。如果 CLR 非常信赖代码，允许它们运行，就会开始执行代码。但是，根据提供给程序集的权限，代码也许要在有限制的环境中运行。如果代码没有得到足够的信赖去运行，或者虽然代码运行了，但试图执行没有相关权限的操作，则会产生一个安全异常(其类型是 `SecurityException` 或它的子类)。代码访问的安全性系统意味着可以停止有害代码的运行，也可以允许代码运行在受保护的环境中，在受保护的环境中，我们相信代码不会进行破坏。

代码访问安全性在不同的环境下也很重要。使用 ClickOnce 部署时，在许多情况下使用完全信任级别是可以的，这表示代码允许执行用户可以执行的所有操作。如果 ClickOnce 部署的应用程序安装在公司中，就可以信任自己的应用程序。应用程序附带的证书也可以提供我们信任的供应商的足够信息，以便在拥有对自己系统的完全权限的情况下，使用 ClickOnce 部署该供应商的应用程序。当然，ClickOnce 也可能只有有限的权限。代码访问安全性比 ClickOnce 更重要的环境有主机环境和带插件的主机。如果创建了一个插件主机，就不希望把所有的权限都赋予从应用程序中加载的插件。而可以限制被调用的程序集的权限。Web 站点发布公司不希望来自不同用户的 Web 应用程序运行在服务器上，且对系统有全部权限。它们可能会使运行着成百上千个 Web 应用程序的服务器崩溃。限制 Web 应用程序的权限是一个好办法。

### 提示：

第 16 章详细介绍了 ClickOnce，插件的创建详见第 36 章。

代码访问的安全性以如下概念为基础：权限(Permission)、权限集(Permission sets)、代码组(Code Group)和策略(policies)。下面讨论这几个概念，因为它们构成了后面章节的基础：

- 权限：权限是允许每一个代码组执行的动作。例如，权限包括“读取文件系统上的文件”、“写入 Active Directory”和“使用套接字打开网络连接”等。有几个预定义的权限，也可以创建自己的权限。
- 权限集是权限的集合。在权限集中，不需要把每个权限都应用于代码；权限组合为权限集。权限集的例子有 FullTrust、LocalIntranet 和 Internet。可以创建包含所需权限的权限集。有 FullTrust 权限的程序集拥有对所有资源的全部访问权限。有 LocalIntranet 的程序集是受限的，除了使用隔离的存储器之外，不能写入文件系统。
- 代码组：代码组用于把具有相似特征的代码集合到一组。它定义了代码的来源。已有的代码组包括 Internet 和 Intranet。Internet 组定义了来自 Internet 的代码，Intranet 组定义了来自 LAN 的代码。把程序集放到代码组中所使用的信息称为“证据”。CLR 收集的其他证据包括代码的发布者、代码的强名以及下载代码的 URI 等。代码组的排列是层次状的，程序集总是与几个代码组相匹配。层次根部的代码组称为“All Code”，

包含其他所有的代码组。层次用于确定程序集属于哪一个代码组，如果程序集提供的证据与树中的代码组不匹配，则程序集不属于树的任何一个代码组。

- 策略：允许系统管理员为整个公司、机器和用户定义不同级别的权限。代码组在所有这些策略中定义，并合并权限。

### 20.4.1 权限

.NET 权限独立于操作系统权限。.NET 权限仅由 CLR 验证。程序集请求特定操作的权限(例如 File 类请求 FileIOPermission)，CLR 验证该程序集是否被授予了该权限，以继续执行。

可以应用于程序集或代码中请求的权限有非常精细的权限列表。下面列出了 CLR 提供的代码访问权限。从中可以看出，使用这些权限，可以很好地控制代码允许做什么和不允许做什么：

- DirectoryServicesPermission: 通过 System.DirectoryServices 类访问 Active Directory 的能力
- DnsPermission: 使用 TCP/IP 域名系统(DNS)的能力
- EnvironmentPermission: 读写环境变量的能力
- EventLogPermission: 读写事件日志的能力
- FileDialogPermission: 访问用户在 Open 对话框中选择的文件的能力。通常用于没有赋予 FileIOPermission 权限，不能对文件进行有限的访问时
- FileIOPermission: 处理文件的能力(其中包括读文件、写文件、添加文件的内容，创建、更改和访问文件夹)
- IsolatedStorageFilePermission: 访问私有虚拟文件系统的能力
- IsolatedStoragePermission: 访问孤立存储器的能力，存储器与各个用户相关，并具有代码身份的一些特征，孤立存储器详见第 25 章
- MessageQueuePermission: 通过 Microsoft Message Queue 使用消息队列的能力
- PerformanceCounterPermission: 利用性能计数器的能力
- PrintingPermission: 打印的能力
- ReflectionPermission: 使用 System.Reflection 在运行期间查找类型信息的能力
- RegistryPermission: 读、写、创建和删除注册表项和值的能力
- SecurityPermission: 执行、断言权限、调用非托管的代码、忽略验证和其他权力的能力
- ServiceControllerPermission: 控制 Windows 服务的能力
- SocketPermission: 在网络传输地址上创建或接受 TCP/IP 连接的能力
- SQLClientPermission: 使用 SQL Server 的 .NET 数据提供程序访问 SQLServer 数据库的能力
- UIPermission: 访问用户界面的能力
- WebPermission: 连接 Web 或接受 Web 连接的能力

对于上面的每一个权限类，通常可以指定更深级别的粒度。例如，DirectoryServicesPermission 可以区分读和写访问权限，也可以确定允许访问或拒绝访问目录服务中的哪些项。

在实践中，如果要利用与上面列出的权限相关的资源，最好在应用程序中加入 try...catch



错误处理块，以便应用程序运行在受限制的权限下时，能够很好地进行处理。应用程序的设计应该指定应用程序在这些情况下怎样运行，而不应该假定应用程序运行在开发它时的同一安全性策略下。例如，如果应用程序不能访问本地磁盘，它是应该退出呢，还是以另一种方式工作呢？

以代码的身份为基础，CLR 可以赋予代码组另一个权限集合。这些权限与 CLR 收集的关于程序集的证据直接相关，它们称为身份权限(Identity Permissions)。下面是身份权限类的名称：

- PublisherIdentityPermission: 软件发布者的数字签名
- SiteIdentityPermission: Web 站点的名称，代码来自这个 Web 站点
- StrongNameIdentityPermission: 程序集的强名
- URLIdentityPermission: URL，代码来自这个 URL(其中包括协议，例如 http://)
- ZoneIdentityPermission: 程序集来自的区域

把权限赋予代码组，就不需要处理每个程序集了。通常在程序块中应用权限，这就是.NET 提供权限集合的原因。代码访问的权限组合在指定的集合中，下面是已命名的权限集合：

- FullTrust: 没有权限的限制
- SkipVerification: 不进行验证
- Execution: 运行的能力，但是不能访问受保护的资源
- Nothing: 没有权限，代码不能执行
- LocalIntranet: 本地内部网的默认策略，它是权限全集的子集。例如，文件 IO 只能在程序集生成的共享上进行读取访问
- Internet: 未知来源的代码的默认策略，这是限制最严格的策略。例如，在这个权限集合下执行的代码没有文件 IO 能力，不能读写事件日志，也不能读写环境变量
- Everything: 这个集合中的所有权限，其中不包括忽略代码验证的权限。管理员可以改变这个权限集合中的权限。默认策略要更强大时，可以使用这个权限集合

**注意：**

只能修改 Everything 权限集合的定义，而其他权限是固定的，不能改变。当然，还可以创建自己的权限集合。

因为只有 CLR 能够把身份权限赋予代码，所以身份权限不能包括在权限集合中。例如，如果一段代码是来自某个发布者，管理员把与另一个发布者相关的身份权限赋予这段代码是毫无意义的。CLR 在需要时赋予身份权限，这样就可以随时利用那些身份权限。

### 1. 要求权限

程序集可以用声明或编程的方式要求权限。为了说明要求权限的工作情况，下面创建一个 Windows Forms 应用程序，它包含一个按钮，当单击按钮时，就访问本地文件系统上的一个文件。如果应用程序没有访问本地磁盘的相关权限(FileIOPermission)，就把按钮标记为不可用(即把按钮变为灰色)。

如果导入命名空间 System.Security.Permissions，就把 Form1 类的构造函数改为检查权限：创建一个 FileIOPermission 对象，并调用这个对象的 Demand()方法，然后对结果进行操作：

```

public Form1()
{
    InitializeComponent();

    try
    {
        FileIOPermission fileioperm = new
            FileIOPermission(FileIOPermissionAccess.AllAccess, @"c:\");
        fileIOPermission.Demand();
    }
    catch
    {
        button1.Enabled = false;
    }
}

```

FileIOPermission 包含在 System.Security.Permissions 命名空间中。这个命名空间包含了权限的全集，除此之外，它还为声明性的权限属性提供了类，为用于创建权限对象的参数提供了枚举值(例如，在创建 FileIOPermission 时，指定是需要完全访问还是只读访问)。

如果从本地磁盘中运行应用程序(在默认状态下安全性策略允许访问本地的存储器)，则对话框中的按钮就是可用的。但是，如果把可执行文件复制到网络的共享区域，再次运行它，则应用程序就处于 LocalIntranet 权限集中，这意味着应用程序不能访问本地的存储器，按钮将变为灰色。

在单击事件处理程序的实现代码中，不需要检查必要的安全性，因为 .NET Framework 中相关的类将请求文件权限，CLR 将确保堆栈中的每一个调用者在继续工作之前都能得到那些权限。如果要从内部网上运行应用程序，试图打开本地磁盘上的文件，除非安全性策略已经更改为把本地磁盘的访问权限赋予应用程序，否则将产生异常。

当代码试图违反被赋予的权限时，CLR 就会产生异常，捕获到的异常是 SecurityException 类型，通过它可以访问许多有用的信息，其中包括可读的堆栈踪迹(SecurityException.StackTrace)和对产生异常的方法的引用(SecurityException.TargetSite)。SecurityException 甚至还能提供 SecurityException.PermissionType 属性，这个属性返回导致发生安全异常的 Permission 对象的类型。如果在诊断安全异常时还有问题，则问题应该发生在所调用的第一个部分。从上面的示例代码中删除 try 和 catch 块，就可以看到安全异常。

## 2. 声明的权限

通过编程调用权限类，可以拒绝、请求和断言权限，除此之外，还可以使用属性，声明指定权限需求。

使用声明安全性的最大好处是通过反射可以访问设置信息。对于系统管理员而言，这也是非常有益处的，因为他们需要经常查看应用程序的安全需求。

例如，可以指定一个方法，它必须具有从 C:\目录下读取数据的权限才能执行，其代码如下所示：

```

using System;
using System.Security.Permissions;

namespace Wrox.ProCSharp.Security
{

```

```

class Program
{
    static void Main()
    {
        MyClass.Method();
    }
}

[FileIOPermission(SecurityAction.Assert, Read="C:/")]
class MyClass
{
    public static void Method()
    {
        // implementation goes here
    }
}

```

注意，如果使用属性断言或请求权限，就不能捕捉到操作失败时引发的异常，因为在 try-catch-finally 子句中没有放置命令式代码。

### 3. 请求权限

从上面的内容可以看出，要求权限(通过代码或采用声明的方式)指的是在运行期间明确声明需要权限。但是，可以把程序集配置为在程序集开始执行时请求权限，也就是说，在程序集执行之前声明它需要的权限。

可以通过以下 3 种方式请求权限：

- 最小权限：代码运行时必需的权限
- 可选权限：代码可以使用这些权限，但是没有这些权限，代码仍然可以有效地运行
- 拒绝权限：确保没有赋予代码的权限

在程序集开始执行时请求权限有以下几个原因：

- 如果程序集需要某些权限才能运行，则只有在程序集开始执行时而不是在执行期间声明需要权限才有意义，这样可以确保用户不会在程序开始工作后遇到障碍。
- 只赋予所需的权限，不赋予其他任何权限。没有明确地请求权限，程序集就会得到比执行所需更多的权限。这会增加程序集被其他代码用于恶意目的的可能性。
- 如果只请求权限的最小集合，则可以增加程序集运行的可能性，因为不能预测最终用户实际使用的安全策略。

如果进行比较复杂的部署，应用程序很有可能安装在没有必需权限的机器上，在这种情况下，请求权限有可能是最有用的。通常，最好让应用程序在开始执行时，就知道它没有权限，而不是在执行到一半时才知道。

在 Visual Studio 中，可以选择属性的 Security 选项卡，检查应用程序的必需权限，如图 20-4 所示。单击 Calculate Permissions 按钮会检查程序集的代码，列出所有的必需权限。

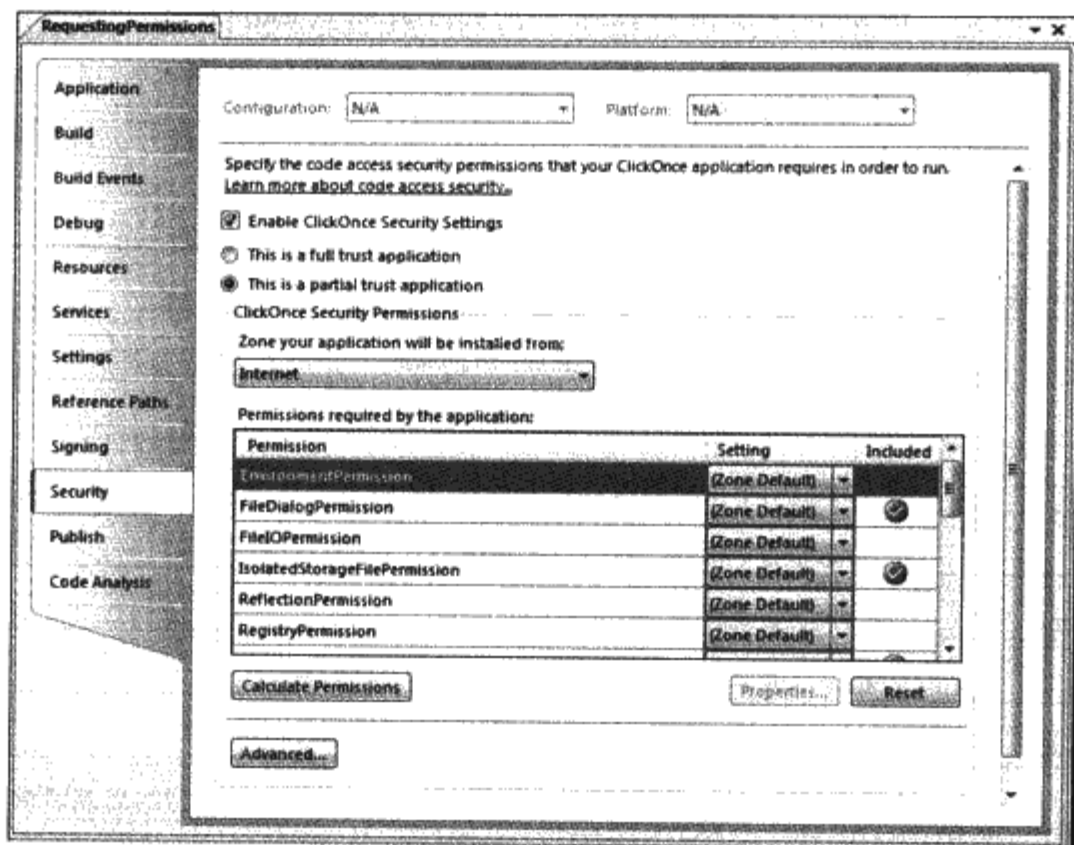


图 20-4

除了使用 Visual Studio 之外，还可以使用命令行工具 `permcals.exe`，计算程序集的必需权限。

#### 命令行

```
permcals.exe -show -stacks -cleancache DemandingPermissions.exe
```

创建了一个包含所有必需权限的 XML 文件。选项 `-show` 表示立即打开 XML 文件。选项 `-stacks` 把堆栈信息添加到 XML 文件中，以查看权限请求来自何处。

必需权限可以作为属性添加到程序集中。下面的 3 个示例演示了怎样使用属性请求权限。如果使用下载的代码，这些示例可以在 `RequestingPermissions` 项目中找到。第一个属性请求把 `UIPermission` 权限赋予程序集，这个权限允许应用程序访问用户界面。这里请求的是最小的权限集合，因此如果程序集没有被赋予所请求的权限，就不能启动：

```
using System.Security.Permissions;
[assembly:UIPermissionAttribute(SecurityAction.RequestMinimum, Unrestricted=true)]
```

接下来，有一个请求不让程序集访问 `C:\驱动器`。这个属性的设置意味着整个程序集不能访问 `C:\驱动器`：

```
[assembly:FileIOPermission(SecurityAction.RequestRefuse, Read="C:/")]
```

最后，使用属性请求赋予程序集访问非托管的代码的权限，这个权限是可选的：

```
[assembly:SecurityPermissionAttribute(SecurityAction.RequestOptional,
Flags = SecurityPermissionFlag.UnmanagedCode)]
```

如果应用程序至少要在一个地方访问非托管的代码，就应把上面这个属性添加到应用程序中。在这种情况下，把权限指定为可选的，因此，应用程序即使没有访问非托管代码的权限，

也可以正常运行。但是，如果程序集没有被赋予访问非托管代码的权限，而程序集试图访问非托管的代码，则将产生 `SecurityException` 异常，应用程序应该预料到这一点，并进行相应的处理。`SecurityAction` 枚举值的完整列表如表 20-2 所示，其中一些值将在后面详细描述。

表 20-2

SecurityAction 枚举	说 明
Assert	允许代码访问调用者不能访问的资源
Demand	要求调用堆栈中的所有调用者有特定的权限
DemandChoice	要求调用堆栈中的所有调用者有一个特定的权限
Deny	迫使对某个权限的后续请求都失败，拒绝该权限
InheritanceDemand	要求派生类拥有特定的权限
LinkDemand	要求当前调用者有特定的权限
LinkDemandChoice	要求当前调用者有一个特定的权限
PermitOnly	类似于 <code>Deny</code> ，没有由 <code>PermitOnly</code> 明确列出的、对资源的后续请求都被拒绝
RequestMinimum	应用于程序集，它包含程序集正确执行所需要的权限
RequestOptional	应用于程序集，它请求程序集可以使用的权限(如果有)，以提供额外的特性和功能
RequestRefuse	应用于程序集，包含不希望程序集拥有的权限

当考虑应用程序的权限需求时，通常必须选择下面两种情况中的一种：

- 在开始执行时请求需要的所有权限，如果没有赋予那些权限，则退出执行。
- 在开始执行时不请求权限，但是执行过程中一直准备着处理安全异常。

一旦使用权限属性以上述方式配置程序集之后，就可以对包含程序集清单的程序集文件使用 `permview.exe` 实用程序和 `-assembly` 选项查看权限：

```
>permcalc.exe -show -assembly RequestingPermissions.exe
```

对应用程序使用前面讨论的 3 个属性，执行这个命令，可以得到如下结果：

```
Microsoft (R) .NET Framework Permissions Calculator.
Copyright (C) Microsoft Corporation 2005. All rights reserved.

Analyzing...
|-----|
.....

RequestingPermissions.exe
Minimal permission set:
<PermissionSet class="System.Security.PermissionSet"
version="1">
<IPermission class="System.Security.Permissions.UIPermission, mscorlib,
Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" version="1"
Unrestricted="true"/>
</PermissionSet>

Optional permission set:
<PermissionSet class="System.Security.PermissionSet"
version="1">
```



```
<IPermission class="System.Security.Permissions.SecurityPermission, mscorlib,
Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" version="1"
Flags="SecurityPermissionFlag.UnmanagedCode" />
</PermissionSet>
```

Refused permission set:

```
<PermissionSet class="System.Security.PermissionSet"
version="1">
<IPermission class="System.Security.Permissions.FileIOPermission, mscorlib,
Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" version="1"
Read="C:" />
</PermissionSet>
```

Generating output...

Writing file: permcalc.exe.PermCalc.xml...

除了可以请求权限之外，还可以请求完整的权限集，请求权限集的优点是不必处理每个权限。但只能请求不能修改的权限集。**Everything** 权限集能够在程序集运行时通过安全性策略来改变，因此，不能请求这个权限集。

下面的示例阐明了如何请求内置的权限集：

```
[assembly:PermissionSetAttribute(SecurityAction.RequestMinimum,
Name = "FullTrust")]
```

在这个示例中，程序集至少要请求内置权限集 **FullTrust**。如果程序集没有被赋予这个权限集，则运行期间会抛出安全异常。

#### 4. 隐式的权限

在赋予权限时，通常有一些隐式的语句可以赋予其他权限。例如，如果赋予了访问 **C:\** 的权限 **FileIOPermission**，就隐含着也可以访问 **C:\** 的子目录。

如果要检查赋予权限时是否隐式地赋予了其他的权限作为子集，可以使用下面的代码：

```
class Program
{
    static void Main()
    {
        CodeAccessPermission permissionA =
            new FileIOPermission(FileIOPermissionAccess.AllAccess, @"C:\");
        CodeAccessPermission permissionB =
            new FileIOPermission(FileIOPermissionAccess.Read, @"C:\temp");
        if (permissionB.IsSubsetOf(permissionA))
        {
            Console.WriteLine("PermissionB is a subset of PermissionA");
        }
    }
}
```

代码的执行结果如下：

```
PermissionB is a subset of PermissionA
```

#### 5. 拒绝权限

有些情况下，需要执行一项操作，并绝对确保所调用的方法在一个受保护的环境中执行。

程序集不允许执行未预料到的操作。例如，在调用一个插件组件时，要确保它不会访问本地磁盘。为此，可以创建一个权限的实例，并确保没有给它授予 Deny()方法，然后，在调用权限类之前，首先调用权限类的 Deny()方法：

```
using System;
using System.IO;
using System.Security;
using System.Security.Permissions;

namespace Wrox.ProCSharp.Security
{
    class Program
    {
        static void Main()
        {
            CodeAccessPermission permission =
                new FileIOPermission(FileIOPermissionAccess.AllAccess, @"C:\");
            permission.Deny();
            UntrustworthyClass.Method();
            CodeAccessPermission.RevertDeny();
        }
    }
    class UntrustworthyClass
    {
        public static void Method()
        {
            try
            {
                StreamReader din = File.OpenText(@"C:\textfile.txt");
            }
            catch
            {
                Console.WriteLine("Failed to open file");
            }
        }
    }
}
```

如果执行这段代码，结果将显示出“Failed to open file”这样的信息，原因是不可信的类没有访问本地磁盘的权限。

注意，Deny()是在 permission 对象的实例上调用的，而 RevertDeny()是静态调用的。其原因是 RevertDeny()回复当前堆栈帧中所有的拒绝请求，如果多次调用了 Deny()，则只需调用一次 RevertDeny()。

## 6. 断言权限

假定有一个完全可信的程序集安装在用户的系统上。在程序集中，有一个方法用于把审计信息保存到本地磁盘上的文本文件中。如果以后安装一个要利用审计特性的应用程序，则那个应用程序必须拥有相关的 FileIOPermission 权限，才能把数据保存到磁盘上。

这似乎很过分，但我们的目的是对本地磁盘的操作进行严格限制。如果具有有限权限的程序集调用更加可信的程序集，则可以暂时增大堆栈上权限的范围，这样，更加可信的程序集就可以代表本身没有权限的调用程序执行一些操作。

断言权限非常重要的另一个例子是，创建程序集，以使用平台调用方式来调用本机代码。

调用本机代码的程序集需要完全信任权限，但这个程序集的所有调用者都需要完全信任权限吗？看看.NET Framework 程序集的情况。File 类调用内部的 Windows API CreateFile(0, 因此需要完全信任权限。File 类本身断言它自己需要的权限，所以调用者不需要有这个权限，但要求有 FileIOPermission 权限。(平台调用方式参见第 24 章。)

为此，更加可信的程序集可以断言它们需要的权限。如果程序集具有它需要的权限以断言额外的权限，则调用程序在堆栈中就不需要拥有更大范围的权限。

下面的代码包含了一个 AuditClass 类，这个类执行 Save() 方法，Save() 方法接收一个字符串，并且把审计数据保存到 C:\audit.txt 文件中。AuditClass 方法断言它需要的权限，以便把审计行添加到文件中。为了进行测试，应用程序的 Main() 方法显式地拒绝了 Audit 方法需要的文件访问权限：

```
using System;
using System.IO;
using System.Security;
using System.Security.Permissions;

namespace Wrox.ProCSharp.Security
{
    class Program
    {
        static void Main()
        {
            CodeAccessPermission permission =
                new FileIOPermission(FileIOPermissionAccess.Append,
                                    @"C:\audit.txt");
            permission.Deny();
            AuditClass.Save("some data to audit");
            CodeAccessPermission.RevertDeny();
        }
    }
    class AuditClass
    {
        public static void Save(string value)
        {
            try
            {
                FileIOPermission permission =
                    new FileIOPermission(FileIOPermissionAccess.Append,
                                        @"C:\audit.txt");
                permission.Assert();
                FileStream stream = new FileStream(@"C:\audit.txt",
                                                    FileMode.Append, FileAccess.Write);

                // code to write to audit file here...
                CodeAccessPermission.RevertAssert();
                Console.WriteLine("Data written to audit file");
            }
            catch
            {
                Console.WriteLine("Failed to write data to audit file");
            }
        }
    }
}
```

在执行上面的代码时，对 `AuditClass` 方法的调用并不会抛出安全异常，即使在调用 `AuditClass` 方法时，它没有访问本地磁盘的权限，情况还是如此。

与 `RevertDeny()` 一样，`RevertAssert()` 也是静态的方法。在目前的架构中，`RevertAssert()` 方法也要回复所有的断言。

使用断言时必须非常小心。我们显式地把权限赋予一个方法，这个方法被其他没有那些权限的代码所调用，可能会产生一个安全漏洞。例如，在审计示例中，即使安全性策略指出已安装的应用程序不能把数据写到本地磁盘上，如果审计程序集为写入数据断言了 `FileIOPermissions`，程序集仍能把数据写到本地磁盘上。

为了进行断言，审计程序集在安装时必须带有 `FileIOAccess` 和 `SecurityPermission`。`SecurityPermission` 允许程序集进行断言，程序集需要 `SecurityPermission` 和要断言的权限才能成功完成。

## 20.4.2 代码组

本节介绍程序集及其权限的管理。这里不是管理每个程序集，而是定义代码组。代码组有一个称之为成员条件的入口需求(entry requirement)。如果要把程序集划归某个代码组，该程序集就必须符合那个代码组的成员条件。例如，成员条件可以是“程序集来自 <http://www.microsoft.com> 站点”或“这个软件的发布者是 Microsoft 公司”等。

每一个代码组都有一个并且只有一个成员条件。程序集可以在多个代码组中。在 .NET 中，代码组可以使用的成员条件如下：

- **Zone**: 代码来自的区域
- **Site**: 代码来自的 Web 站点
- **Strong name**: 代码唯一的、可验证的名称。强名详见第 17 章。
- **Publisher**: 代码发布者
- **URL**: 代码来自的具体位置
- **Hash value**: 程序集的散列值
- **Skip verification**: 请求 `Skip verification` 的代码避开代码验证检查。代码的验证可以确保代码以定义合理和可接受的方式访问类型。运行库不能确保类型不安全的代码是安全的。
- **Application directory**: 程序集在应用程序中的位置
- **All code**: 符合条件的所有代码
- **Custom**: 与用户相关的条件

上述成员条件的第一个类型就是 `Zone`，这个类型是最常用的。`Zone` 是一段代码区域的开端，表示下面的内容：`MyComputer`、`Internet`、`Intranet`、`Trusted` 或 `Untrusted`。使用 Windows Security Center 中的 `Internet` 选项可以对这些区域进行管理。

代码组的安排是层次状的，根部是 `All Code` 成员条件，如图 20-5 所示。从这个图中可以看出，每一个代码组都有一个成员条件，并且指定赋予代码组的权限。注意，如果程序集不符合代码组中的成员条件，则 CLR 不会给它匹配那个代码组下面的代码组。



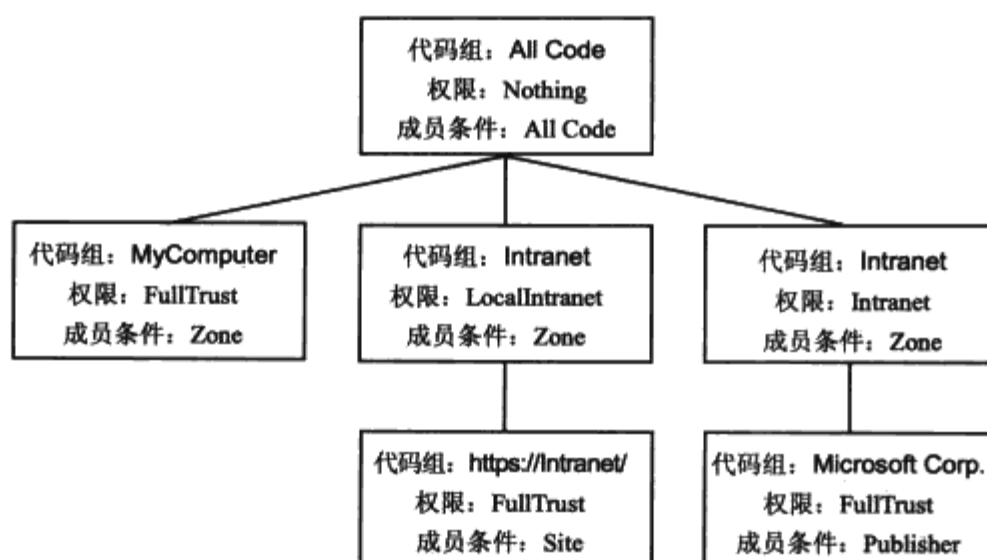


图 20-5

### 1. Caspol.exe——代码访问安全性策略工具

本节将花大量的篇幅介绍命令行工具：代码访问安全性策略工具。要得到它的选项列表，只需输入下面的命令：

```
caspol.exe -?
```

使用下面的命令把输出发送给一个文本文件：

```
caspol.exe > output.txt
```

使用 **caspol.exe**，可以查看机器上的代码组。**caspol.exe** 的执行结果是列出机器上代码组的层次结构，并描述每一个代码组。键入如下命令：

```
caspol.exe -listdescription
```

另外，**-listdescription** 选项还有一个缩写方式：**-ld**。下面是该命令的部分结果：

```
Microsoft (R) .NET Framework CasPol 2.0.50727.1426
Copyright (C) Microsoft Corporation. All rights reserved.
```

```
Security is ON
Execution checking is ON
Policy change prompt is ON
```

```
Level = Machine
```

```
Full Trust Assemblies:
```

```
1. All_Code: Code group grants no permissions and forms the root of the code
group tree.
```

```
1.1. My_Computer_Zone: Code group grants full trust to all code originating
on the local computer
```

```
1.1.1. Microsoft_Strong_Name: Code group grants full trust to code signed
with the Microsoft strong name.
```

```
1.1.2. ECMA_Strong_Name: Code group grants full trust to code signed with
the ECMA strong name.
```

```
1.2. LocalIntranet_Zone: Code group grants the intranet permission set to
```



code from the intranet zone. This permission set grants intranet code the right to use isolated storage, full UI access, some capability to do reflection, and limited access to environment variables.

1.2.1. Intranet\_Same\_Site\_Access: All intranet code gets the right to connect back to the site of its origin.

1.2.2. Intranet\_Same\_Directory\_Access: All intranet code gets the right to read from its install directory.

1.3. Internet\_Zone: Code group grants code from the Internet zone the Internet permission set. This permission set grants Internet code the right to use isolated storage and limited UI access.

.NET 安全子系统确保每一个代码组中的代码只能做某些事情。例如，Internet 区域中的代码在默认状态下比本地驱动器中的代码有更严格的限制；本地驱动器中的代码通常有访问本地磁盘上数据的权限，但是 Internet 中的程序集在默认状态下就没有这个权限。

使用 `caspol.exe` 和它在 Microsoft Management Console 中的等价选项，可以为每一个代码访问组指定信任级别，还可以按照更小的粒度方式管理代码组和权限。

再看看代码访问组，但是，这次的信息比上次要少一些。确保以本地管理员身份登录后，打开命令提示窗口，输入下面的命令：

```
caspol.exe -listgroups
```

得到如下信息：

```
Microsoft (R) .NET Framework CasPol 2.0.50727.1426
Copyright (C) Microsoft Corporation. All rights reserved.
```

```
Security is ON
Execution checking is ON
Policy change prompt is ON
```

```
Level = Machine
```

```
Code Groups:
```

```
1. All code: Nothing
```

```
1.1. Zone - MyComputer: FullTrust
```

```
1.1.1. StrongName -
```

```
002400000480000094000000060200000024000052534131000400000100010007D1FA57C4AED9F0A32E8
4AA0FAEFD0DE9E8FD6AEC8F87FB03766C834C99921EB23BE79AD9D5DCC1DD9AD236132102900B723CF980
957FC4E177108FC607774F29E8320E92EA05ECE4E821C0A5EFE8F1645C4C0C93C1AB99285D622CAA652C1
DFAD63D745D6F2DE5F17E5EAF0FC4963D261C8A12436518206DC093344D5AD293: FullTrust
```

```
1.1.2. StrongName - 00000000000000000400000000000000: FullTrust
```

```
1.2. Zone - Intranet: LocalIntranet
```

```
1.2.1. All code: Same site Web.
```

```
1.2.2. All code: Same directory FileIO - 'Read, PathDiscovery'
```

```
1.3. Zone - Internet: Internet
```

```
1.3.1. All code: Same site Web.
```

```
1.4. Zone - Untrusted: Nothing
```

```
1.5. Zone - Trusted: Internet
```

```
1.5.1. All code: Same site Web.
```

```
Success
```

在输出结果的开头，是 Security is ON。在本章后面的内容中，我们将会看到 Security 可以

先关闭, 然后再打开。

在默认状态下, Execution Checking 设置的值是 on, 这意味着所有的程序集在运行之前, 必须给它们赋予执行权限。如果使用 caspol (caspol.exe -execution on|off) 关闭执行检查, 则程序集没有执行权限也可以运行。在这种情况下, 如果程序集在运行过程中有违背安全性策略的行为, 就会产生安全异常。

Policy change prompt 选项指定在更改安全性策略时, 是否可以看到 "Are you sure" 警告信息。

把代码划分为这些组之后, 就可以更精细地管理安全性, 还可以实现对更少代码的完全信任。注意, 每一个组都有一个标记(例如 "1.2"), 这些标记是 .NET 自动生成的, 在不同的机器上是不同的。通常不是按照每一个程序集来管理安全性, 而是使用代码组来管理。

如果一台机器上同时安装了几个 .NET 版本, 则 caspol.exe 只更改与它相关的 .NET 安装版本的安全性策略。

#### (1) 查看程序集的代码组

如果程序集符合代码组的成员条件, 它们就属于代码组。回到代码组的示例中, 从 Web 站点 <http://intranet/> 载入程序集, 则它匹配的代码组如图 20-6 所示。这个程序集也是根代码组 (All Code) 的成员。如果程序集来自本地的网络, 则它还是 Intranet 代码组的成员; 但是, 当从某一指定站点(如 <http://intranet/>) 载入程序集时, 它也被赋予 FullTrust 权限, 这意味着程序集的运行没有限制条件。

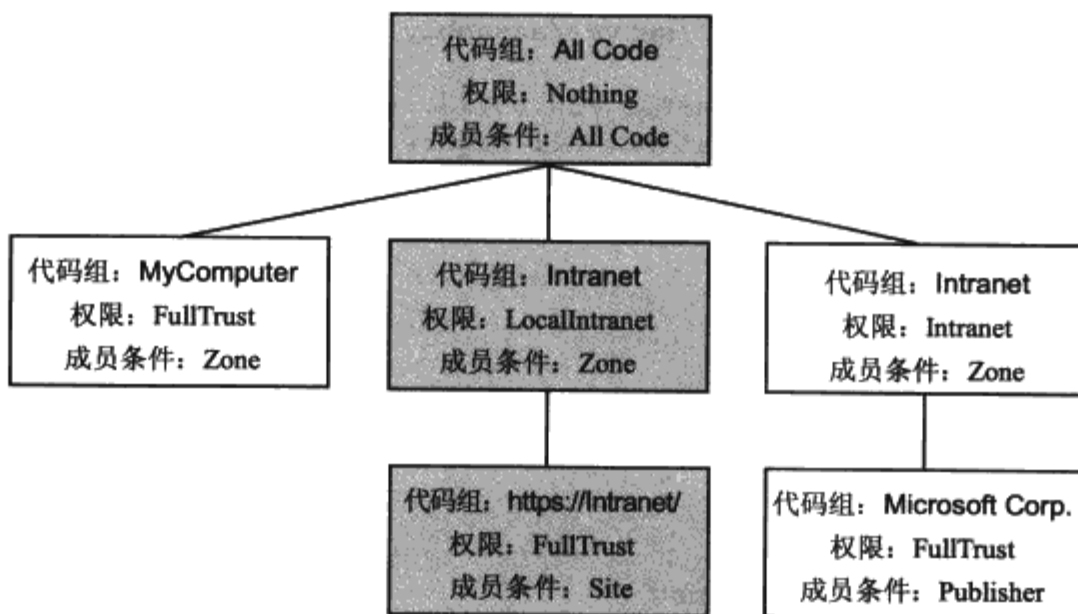


图 20-6

使用如下命令, 很容易查看程序集所属的代码组:

```
caspol.exe -resolvegroup assembly.dll
```

对本地磁盘上的程序集运行这个命令, 可以得到如下结果:

```
Microsoft (R) .NET Framework CasPol 2.0.50727.1426
Copyright (C) Microsoft Corporation. All rights reserved.
```

```
Level = Enterprise
```

```
Code Groups:
```

```
1. All code: FullTrust
```

```

Level = Machine

Code Groups:

1. All code: Nothing
   1.1. Zone - MyComputer: FullTrust

Level = User

Code Groups:

1. All code: FullTrust

Success

```

注意，代码组是在 3 个级别上列出的，这 3 个级别是 Enterprise、Machine 和 User。现在只关注 Machine 级别。如果要了解这 3 个级别之间的关系，赋予程序集的有效权限是这 3 个级别上权限的交集。例如，如果在 Enterprise 级别的策略中删除了 Internet 区域的 FullTrust 权限，则 Internet 区域中代码的所有权限都被取消，其他两个级别的设置就变得不相关了。

现在，对这个程序集执行 `caspol.exe` 命令，看看它所属的代码组。但是这次程序集是通过 HTTP 协议在 Web 服务器上访问的，这样，程序集成为不同代码组的成员，它的权限受到更严格的限制：

```
caspol.exe -resolvegroup http://server/assembly.dll
```

```

Microsoft (R) .NET Framework CasPol 2.0.50727.1426
Copyright (C) Microsoft Corporation. All rights reserved.

```

```

Level = Enterprise

Code Groups:

1. All code: FullTrust
Level = Machine

Code Groups:

1. All code: Nothing
   1.1. Zone - Internet: Internet
       1.1.1. All code: Same site Web.

Level = User

Code Groups:

1. All code: FullTrust

Success

```

程序集授予了 Internet 权限和 Same Site Web 权限。权限的交集允许代码对 UI 进行有限的访问，并可以连接到最初生成它的站点。

## 2. 代码访问权限和权限集

想像自己正在一个大公司中管理办公机器网络的安全性策略。在这种情况下，CLR 在执行

代码之前收集代码的证据是非常有用的；同样，一旦 CLR 知道了代码来自何处，管理员就必然有机会控制代码在他管理的数百台机器上的行为。这个问题需要使用权限来解决。

一旦程序集与代码组相匹配，CLR 就会根据安全性策略赋予程序集一些权限。在 Windows 中管理权限时，通常不是把权限赋予用户，而是把权限赋予用户组。程序集也是如此，也就是说，把权限应用于代码组，而不是各个程序集，这就大大简化了 .NET 中安全性策略的管理。

再看看程序集的权限。假定用户在使用一个 Microsoft 应用程序，并试图使用一个以前从没用过的特性。应用程序没有把代码的副本保存在本地，因此必须在 Internet 上请求代码，然后下载到下载程序集缓存中。如果代码是由特定的公司(这个公司已经使用证书签署了程序集)通过 Internet 发布的，则程序集所属代码组中的成员关系如图 20-7 所示。

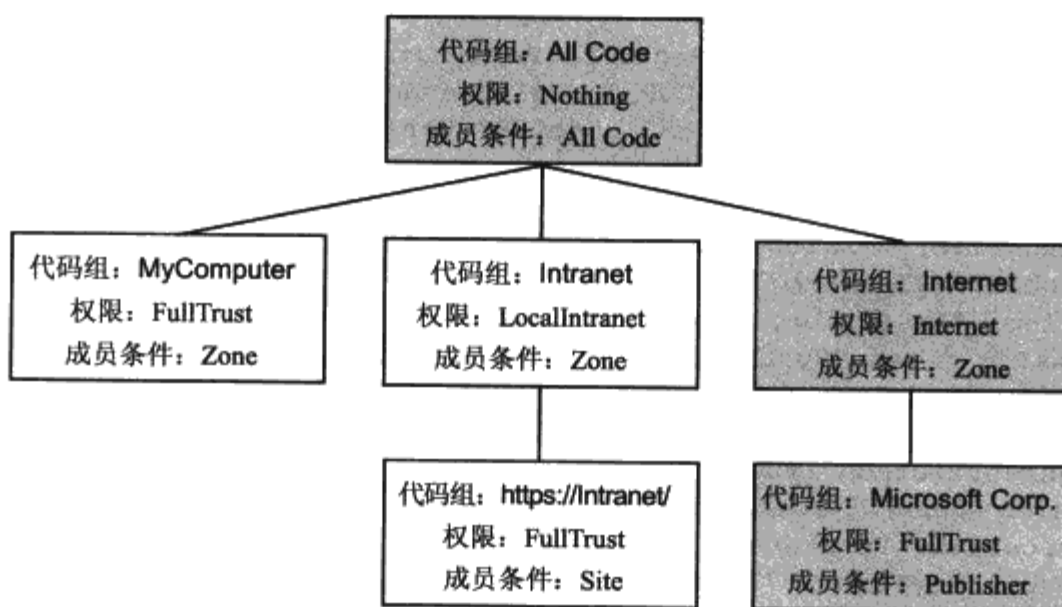


图 20-7

依照这个示例中的策略，代码组 All Code 和 Internet 的权限有限，但图中右下角的代码组却赋予程序集 FullTrust 权限。程序集的有效权限是它所属所有代码组中权限的并集。当权限以这种方式合并时，有效的权限就是被授予的最高权限，也就是说，程序集所属的每一个代码组都会向程序集的有效权限集中添加权限。

正像可以查看程序集所属的代码组一样，也可以查看赋予程序集所属代码组的权限。在查看权限时，不但能够看到代码的访问权限(即允许代码做什么)，也可以看到代码的身份权限(身份权限能访问代码在运行期间表现出来的证据)。使用如下的命令，可以查看程序集代码组的权限：

```
caspol.exe -resolveperm assembly.dll
```

对一个程序集使用这个命令，并且查看在通过本地的内部网访问程序集时，赋予程序集的代码访问权限和身份权限。如果输入下面的命令，就可以看到代码访问权限和 3 个身份权限：

```
caspol.exe -resolveperm http://somehost/assembly.dll
```

```
Microsoft (R) .NET Framework CasPol 2.0.50727.1426
Copyright (C) Microsoft Corporation. All rights reserved.
```

```
Resolving permissions for level = Enterprise
Resolving permissions for level = Machine
Resolving permissions for level = User
```

```
Grant =
```

```

<PermissionSet class="System.Security.PermissionSet"
    version="1">
    <IPermission class="System.Security.Permissions.EnvironmentPermission,
        mscorlib, Version=2.0.0.0, Culture=neutral,
        PublicKeyToken=b77a5c561934e089" Version="1" Read="Username"/>
    <IPermission class="System.Security.Permissions.FileDialogPermission,
        mscorlib, Version=2.0.0.0, Culture=neutral,
        PublicKeyToken=b77a5c561934e089"
        version="1" Unrestricted="true"/>
    <IPermission class="System.Security.Permissions.IsolatedStorageFilePermission,
        mscorlib, Version=2.0.0.0, Culture=neutral,
        PublicKeyToken=b77a5c561934e089" version="1"
        Allowed="AssemblyIsolationByUser"
        UserQuota="9223372036854775807" Expiry="9223372036854775807"
        Permanent="True"/>
    <IPermission class="System.Security.Permissions.ReflectionPermission,
        mscorlib, Version="2.0.0.0, Culture=neutral,
        PublicKeyToken= b77a5c561934e089" Version="1"
        Flags="ReflectionEmit" />
    <IPermission class="System.Security.Permissions.SecurityPermission,
        mscorlib, Version=2.0.0.0, Culture=neutral,
        PublicKeyToken=b77a5c561934e089" version="1"
        Flags="Assertion, Execution, BindingRedirects"/>
    <IPermission class="System.Security.Permissions.UIPermission,
        mscorlib, Version=2.0.0.0, Culture=neutral,
        PublicKeyToken=b77a5c561934e089" version="1"
        Unrestricted="true" />
    <IPermission class="System.Security.Permissions.SiteIdentityPermission,
        mscorlib, Version=2.0.0.0, Culture=neutral,
        PublicKeyToken=b77a5c561934e089" version="1"
        Site="somehost" />
    <IPermission class="System.Security.Permissions.UrlIdentityPermission,
        mscorlib, Version=2.0.0.0, Culture=neutral,
        PublicKeyToken=b77a5c561934e089" version="1"
        Url="http://somehost/assembly.dll" />
    <IPermission class="System.Security.Permissions.ZoneIdentityPermission,
        mscorlib, Version=2.0.0.0, Culture=neutral,
        PublicKeyToken=b77a5c561934e089" version="1"
        Zone="Intranet" />
    <IPermission class="System.Net.DnsPermission,
        System, Version=2.0.0.0, Culture=neutral,
        PublicKeyToken=b77a5c561934e089" version="1"
        Unrestricted="true" />
    <IPermission class="System.Drawing.Printing.PrintingPermission,
        System.Drawing, Version=2.0.0.0, Culture=neutral,
        PublicKeyToken=b03f5f7f11d50a3a" version="1"
        Level="DefaultPrinting" />
    <IPermission class="System.Net.WebPermission,
        System, Version=2.0.0.0, Culture=neutral,
        PublicKeyToken=b77a5c561934e089" version="1">
        <ConnectAccess>
            <URI uri=" (https|http)://somehost/*[/?]" />
        </ConnectAccess>
    </IPermission>
</PermissionSet>

Success

```

上面的权限都是以 XML 方式显示出来的，其中包括定义权限的类、包含类的程序集、权限的版本以及加密标记。从中可以看出，我们能够创建自己的权限，此外，每一个身份权限都



包括更详细的信息，例如 `UrlIdentityPermission` 类的详细信息，通过这个类可以访问生成代码的 URL 等。

在输出结果的开头，要格外注意 `caspol.exe` 解析 Enterprise、Machine 和 User 级别上权限的方式和列出有效权限的方式。下面讨论策略的这 3 个级别。

20.4.3 策略的级别：Machine、User 和 Enterprise

前面讨论的都是单机环境中的安全性。但通常需要对具体的用户或整个公司指定安全性策略，这就是 .NET 不提供一个代码组级别，而是提供 3 个代码组级别的原因：

- Machine
- Enterprise
- User

3 个代码组级别是单独管理、并行存在的，如图 20-8 所示。

如果有 3 个安全性策略，该应用哪个策略？有效的权限就是 3 个策略级别上权限的交集。也就是说，每一个策略级别都可以否决另一个级别的权限，对于管理员来说，这是一件好事，因为他们的设置可以覆盖用户的设置。

为了使用 `caspol.exe` 处理 User 级别或 Enterprise 级别上的代码组和权限，必须添加 `-enterprise` 参数或 `-user` 参数，以改变命令的模式。默认状态下，`caspol.exe` 工作在 Machine 级别上，到目前为止，我们一直在 Machine 级别上使用 `caspol.exe`。下面的命令列出了 User 级别上的代码组：

```
caspol.exe -user -listgroups
```

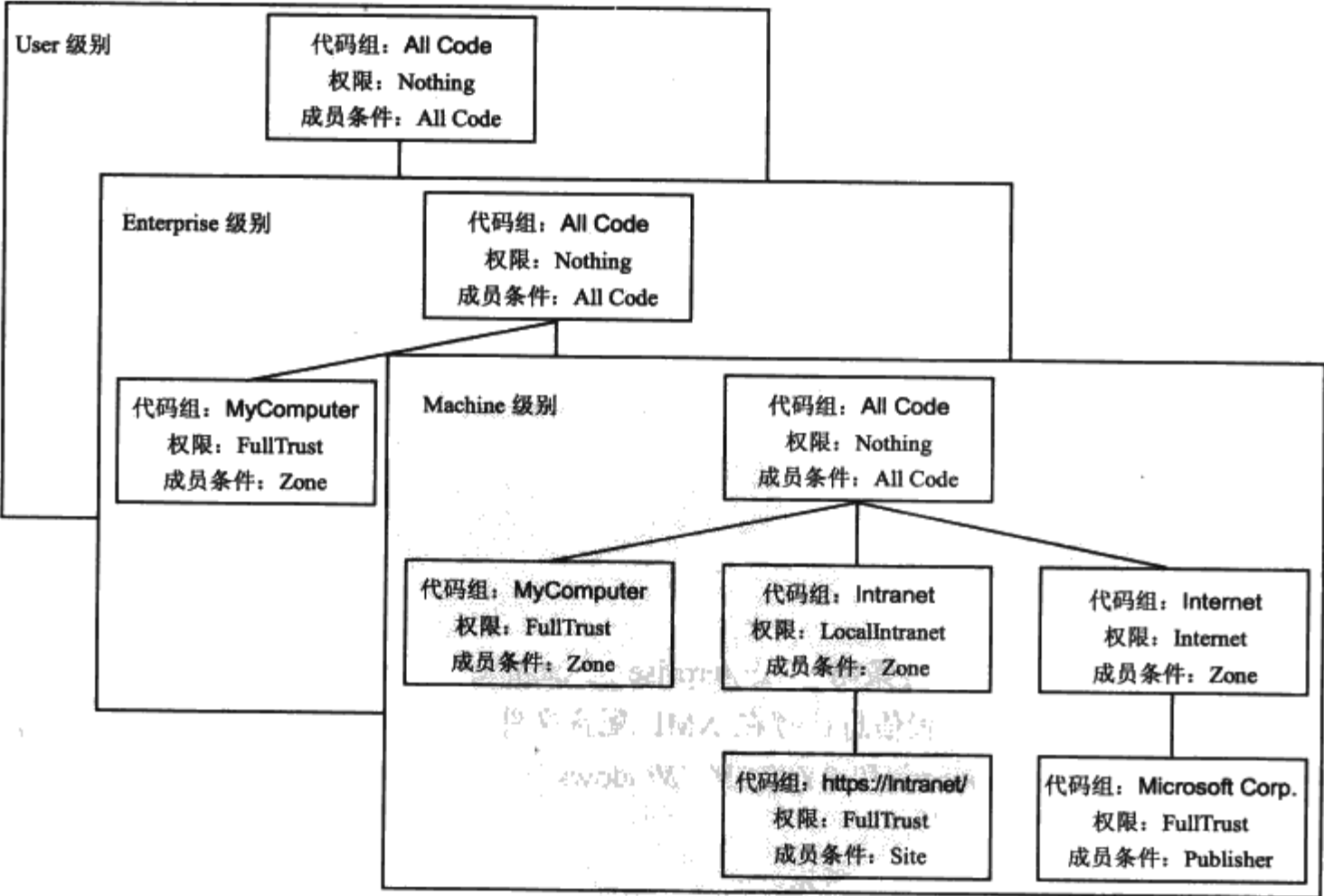


图 20-8

在默认的安装上，这个命令的执行结果如下所示：

```
Security is ON
Execution checking is ON
Policy change prompt is ON

Level = User

Code Groups:

1. All code: FullTrust
Success
```

运行同样的命令，但是这次看一下 Enterprise 级别上的代码组：

**caspol.exe -enterprise -listgroups**

这个命令的执行结果如下所示：

```
Security is ON
Execution checking is ON
Policy change prompt is ON

Level = Enterprise

Code Groups:

1. All code: FullTrust
Success
```

从中可以看出，在默认状态下，User 级别和 Enterprise 级别都配置为允许给 All Code 代码组赋予 FullTrust 权限。其结果是，.NET 安全性的默认设置对 Enterprise 级别和 User 级别没有任何限制，实际执行的策略完全由 Machine 级别的安全性所决定。例如，如果要把比 FullTrust 限制更严格的权限或权限集赋予 Enterprise 级别或 User 级别，则那些限制条件将对所有的权限都起作用，并且有可能覆盖 Machine 级别上的权限。由于有效的权限是所有权限的交集，因此，如果要把 FullTrust 应用到代码组上，就必须在 3 个策略级别上把这个权限赋予该代码组。

当以管理员的身份运行 caspol.exe 时，caspol.exe 把 Machine 级别作为默认级别。但是，如果退出并以非 Administrator 用户组中的用户身份重新登录，caspol.exe 将使用 User 级别代替默认的 Machine 级别。此外，如果用户更改后的安全性使 caspol.exe 不能运行，caspol.exe 就不允许用户对安全性策略进行改动。

## 20.5 安全策略的管理

如前所述，3 个级别的安全策略(即 Enterprise、Machine 和 User)把代码组、权限和权限集连接起来。.NET 中的安全配置信息保存在 XML 配置文件中，这个文件受 Windows 安全策略保护。例如，只有 Administrator 和 SYSTEM Windows 组中的用户能够修改 Machine 级别的安全策略。

保存安全策略的文件位置如下所示：

- Enterprise 策略配置

```
<windir> \Microsoft.NET\Framework\<version> \Config\enterprise.config
```

- Machine 策略配置

```
<windir> \Microsoft.NET\Framework\<version> \Config\security.config
```

- User 策略配置

```
%USERPROFILE%\application data\Microsoft\CLR security config\<version>
\security.config
```

子目录<version>随着安装在机器上的 CLR 版本的不同而有所变化。.NET 2.0、3.0 和 3.5 基于同一个运行库版本，所以在这些 Framework 上有一个配置即可。必要时，可以手动编辑这些配置文件。例如，当管理员需要为没有登录到他们账户的用户配置安全策略时，就需要手动编辑配置文件。但是，一般而言，最好使用 `caspol.exe` 或其他管理工具来管理安全策略。

有了所有的准备知识后，下面创建一个简单的应用程序，这个应用程序将访问本地驱动器，对本地驱动器的访问操作是需要认真管理的。这个应用程序是 C# Windows Forms 应用程序，它包含一个列表框和一个按钮，如图 20-9 所示。如果单击这个按钮，则本地用户文件夹中的 `animals.txt` 文件的内容将填写到列表框中。在启动应用程序之前，必须把这个文件复制到该文件夹中。在 Windows Vista 中，文件是 `c:\users\<username>\Documents\animals.txt`。

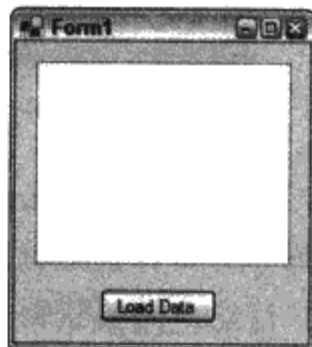


图 20-9

这个应用程序是使用 Visual Studio 创建的，唯一的改动是把列表框和 Load Data 按钮添加到窗体上，并给按钮添加了一个事件，其代码如下所示：

```
// Example from SimpleExample
private void OnLoadData(object sender, System.EventArgs e)
{
    string filename = Environment.GetFolderPath(
        Environment.SpecialFolder.MyDocuments), "animals.txt";
    Using (StreamReader stream = File.OpenText(filename))
    {
        string str;
        while ((str=stream.ReadLine()) != null)
        {
            listAnimals.Items.Add(str);
        }
    }
}
```

这个按钮从用户文件夹下打开一个简单的文本文件 `animals.txt`，这个文本文件包含几行内容，每一行都是一个动物名称，把每一行的动物名称都载入到字符串中，然后使用字符串创建列表框中的每一项。

如果从本地机器上运行这个应用程序，并单击窗体上的按钮，数据就会加载并显示在列表框中，如图 20-10 所示。运行库在后台赋予程序集访问用户界面和从本地磁盘读取数据这两个权限。

前面曾经说过，Intranet 区域代码组上的权限比本地机器上的权限有更多的限制，特别是

Intranet 区域代码组中的代码不允许访问本地磁盘。如果再次运行应用程序，但是这次是从网络共享上运行，则它与前面的运行没有什么区别，因为它被授予执行和访问用户界面的权限，但是，如果单击窗体上的 Load Data 按钮，就会产生安全异常，如图 20-11 所示。从显示的异常信息文本中可以看出，System.Security.Permissions.FileIOPermission 对象没有赋予应用程序，这个对象就是 Framework 中的类所要求的权限，有了这个权限，才能从本地磁盘上载入数据。



图 20-10

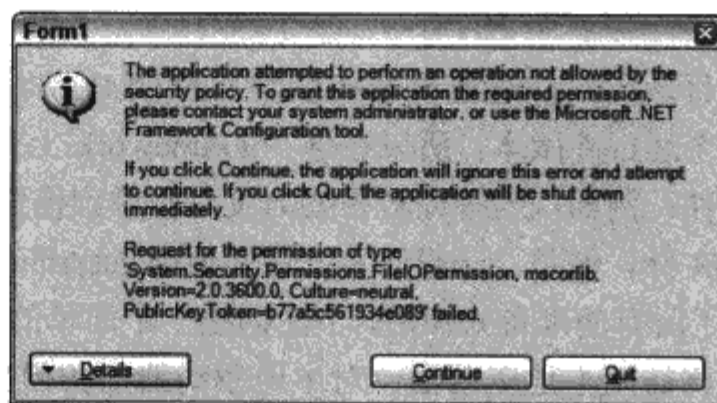


图 20-11

默认状态下，Intranet 代码组被赋予 LocalIntranet 权限集，把权限集更改为 FullTrust，Intranet 区域中的所有代码就可以完全无限制地运行。

首先，需要获得 LocalIntranet 代码组的数字标签，为此，可以使用下面的命令：

```
caspol.exe -listgroups
```

这会得到如下结果：

Code Groups:

1. All code: Nothing

1.1. Zone - MyComputer: FullTrust

1.1.1. StrongName -

```
002400000480000094000000060200000024000052534131000400000100010007D1FA57C4AED9F0A32E8
4AA0FAEFD0DE9E8FD6AEC8F87FB03766C834C99921EB23BE79AD9D5DCC1DD9AD236132102900B723CF980
957FC4E177108FC607774F29E8320E92EA05ECE4E821C0A5EFE8F1645C4C0C93C1AB99285D622CAA652C1
DFAD63D745D6F2DE5F17E5EAF0FC4963D261C8A12436518206DC093344D5AD293: FullTrust
```

1.1.2. StrongName - 00000000000000000400000000000000: FullTrust

1.2. Zone - Intranet: LocalIntranet

1.2.1. All code: Same site Web.

1.2.2. All code: Same directory FileIO - 'Read, PathDiscovery'

1.3. Zone - Internet: Internet

1.3.1. All code: Same site Web.

1.4. Zone - Untrusted: Nothing

1.5. Zone - Trusted: Internet

1.5.1. All code: Same site Web.

注意 LocalIntranet 组被列为 1.2。现在使用下面的命令应用完全信任级别：

```
caspol.exe -chggroup 1.2 FullTrust
```

现在，如果再次从网络共享上运行这个应用程序，并单击按钮，C:\驱动器根目录中文件的内容就被填写到列表框中，并且没有异常情况出现。

因此，如果利用的资源是由权限管理的，最好扩展代码，捕获安全异常，让应用程序能正

常退出。例如，在上面的应用程序中，可以为文件的访问代码添加一个 try-catch 块，如果抛出了 SecurityException 异常，就在列表框中显示：“Permission denied accessing file”。

```
// Code from SimpleExample
```

```
private void OnLoadData(object sender, System.EventArgs e)
{
    try
    {
        string filename = Environment.GetFolderPath(
            Environment.SpecialFolder.MyDocuments) + @"\animals.txt";
        StreamReader din = File.OpenText(filename);
        string str;
        while ((str=din.ReadLine()) != null)
        {
            listAnimals.Items.Add(str);
        }
    }
    catch (SecurityException ex)
    {
        MessageBox.Show(ex.Message);
    }
}
```

在实际应用中，如果要通过网络共享运行某一应用程序，最好的解决方法就是使客户机不共享内部网上的代码。而是使用代码组和成员条件去严格控制应用程序的需求，例如使用应用程序在内部网上的位置、强名或证明发布者身份的证书等。

### 20.5.1 代码组和权限的管理

在管理 .NET 上的安全时，如果由于安全异常而导致程序集执行失败，通常有以下 3 种选择：

- 放松策略权限：可以修改 Machine 策略的权限，允许给特定的代码组赋予更多的权限。但最好不要给内联网或 Internet 上的程序集赋予更多的权限，因为这会使 Trojan 木马程序获得访问系统的权限，而可以添加有特定权限的新代码组。
- 移动程序集：不能像信任本地系统上的程序集那样信任网络共享上的程序集。而应创建一个新的代码组，把程序集移动到本地系统上，使之有更多的权限。
- 把强名应用到程序集上：最好给程序集应用强名，再创建一个信任强名的代码组。

在作出这 3 种决定时，必须考虑程序集的信任级别。

### 20.5.2 安全性的启用和禁用

默认状态下，.NET 的安全性是启用的。如果由于某种原因需要禁用安全性，可以使用如下命令：

```
caspol.exe -security off
```

作为一个新的安全性功能，在命令行上运行这个命令会临时禁用安全性。只要按下回车键，会再次打开安全性。只要需要，就可以一直打开这个命令提示，继续运行另一个命令提示。完成了与安全性相关的任务后，按下回车键，再次打开安全性。也可以显式打开安全性：



```
caspol.exe -security on
```

如果需要把安全性配置还原到原始状态，可以输入下面的命令：

```
>caspol.exe -reset
```

这个命令可以把安全策略重新设置为安装时的默认状态。

### 20.5.3 代码组的创建

可以创建自己的代码组，然后把一些权限赋予它们。例如，可以指定信任来自 `www.wrox.com` 站点的所有代码，使那些代码具有系统的完全访问权(不信任来自其他 Web 站点的代码)。

前面运行 `caspol` 时，列出了可用的组和数字赋值。接着，Zone:Internet 标记为 1.3，所以输入如下命令：

```
>caspol.exe -addgroup 1.3 -site www.wrox.com FullTrust
```

注意，当试图显式地更改机器上的安全性策略时，这个命令将要求进行确认。现在，如果再次运行 `caspol.exe -listgroups` 命令，则将添加新的代码组，并赋予它 Full Trust 权限：

```
...
1.2. Zone - Intranet: LocalIntranet
    1.2.1. All code: Same site Web.
    1.2.2. All code: Same directory FileIO - Read, PathDiscovery
1.3. Zone - Internet: Internet
    1.3.1. All code: Same site Web.
    1.3.2. Site - www.wrox.com: FullTrust
1.4. Zone - Untrusted: Nothing
1.5. Zone - Trusted: Internet
    1.5.1. All code: Same site Web.
```

下面看看另一个示例。假设想在 Intranet 代码组(1.2)下面创建一个代码组，并且把 FullTrust 权限赋予所有从某一网络共享上运行的应用程序，其命令如下所示：

```
>caspol.exe -addgroup 1.2 -url file:///\\intranetserver/sharename/* FullTrust
```

### 20.5.4 代码组的删除

使用如下命令，可以删除已经创建的代码组：

```
>caspol.exe -remgroup 1.3.2
```

如果想更改安全策略，该命令会要求进行确认。如果给出肯定的答复，则它将说明代码组已被删除。

注意：

不能删除 All Code 代码组，但可以删除它下面的代码组，其中包括 Internet、MyComputer 和 LocalIntranet。

## 20.5.5 代码组权限的更改

使用 `caspol.exe`, 可以放松或加强对赋予代码组的权限的限制。如果要把 `FullTrust` 权限赋予 `Intranet` 区域, 首先需要得到代表 `Intranet` 代码组的标签, 其命令如下:

```
>caspol.exe -listgroups
```

这个命令的执行结果将显示出 `Intranet` 代码组:

Code Groups:

```
1. All code: Nothing
  1.1. Zone - MyComputer: FullTrust
    1.1.1. StrongName -
002400000480000094000000060200000024000052534131000400000100010007D1FA57C4AED9F0A3
2E84AA0FAEFD0DE9E8FD6AEC8F87FB03766C834C99921EB23BE79AD9D5DCC1DD9AD236132102900B72
3CF980957FC4E177108FC607774F29E8320E92EA05ECE4E821C0A5EFE8F1645C4C0C93C1AB99285D
622CAA652C1DFAD63D745D6F2DE5F17E5EAF0FC4963D261C8A12436518206DC093344D5AD293:
FullTrust
    1.1.2. StrongName - 00000000000000000400000000000000: FullTrust
  1.2. Zone - Intranet: LocalIntranet
    1.2.1. All code: Same site Web.
    1.2.2. All code: Same directory FileIO - Read, PathDiscovery
  1.3. Zone - Internet: Internet
    1.3.1. All code: Same site Web.
  1.4. Zone - Untrusted: Nothing
  1.5. Zone - Trusted: Internet
    1.5.1. All code: Same site Web.
```

在知道 `Intranet` 代码组的标签是 1.2 之后, 可以输入下面的命令, 更改代码组的权限:

```
caspol.exe -chggroup 1.2 FullTrust
```

这个命令会要求对安全性策略的更改进行确认。现在, 如果再次运行 `caspol.exe -listgroups` 命令, `Intranet` 行末尾的权限就更改为 `FullTrust`, 如以下代码所示。

Code Groups:

```
1. All code: Nothing
  1.1. Zone - MyComputer: FullTrust
    1.1.1. StrongName -
002400000480000094000000060200000024000052534131000400000100010007D1FA57C4AED9F0
A32E84AA0FAEFD0DE9E8FD6AEC8F87FB03766C834C99921EB23BE79AD9D5DCC1DD9AD23613210290
0B723CF980957FC4E177108FC607774F29E8320E92EA05ECE4E821C0A5EFE8F1645C4C0C93C1AB99
285D622CAA652C1DFAD63D745D6F2DE5F17E5EAF0FC4963D261C8A12436518206DC093344D5AD293:
FullTrust
    1.1.2. StrongName - 00000000000000000400000000000000: FullTrust
  1.2. Zone - Intranet: FullTrust
    1.2.1. All code: Same site Web.
    1.2.2. All code: Same directory FileIO - Read, PathDiscovery
  1.3. Zone - Internet: Internet
    1.3.1. All code: Same site Web.
  1.4. Zone - Untrusted: Nothing
  1.5. Zone - Trusted: Internet
    1.5.1. All code: Same site Web.
```

## 20.5.6 权限集的创建和应用

可以使用如下命令创建新的权限集：

```
>caspol.exe -addpset MyCustomPermissionSet permissionset.xml
```

这个命令指定创建一个名叫 MyCustomPermissionSet 的新权限集，这个新权限集用指定的 XML 文件内容配置，而且这个 XML 文件必须包含标准的格式以指定 PermissionSet。下面是 Everything 权限集的权限集文件，可以裁减这一权限集文件，创建自己想要的权限集。

```
<PermissionSet class="System.Security.NamedPermissionSet" version="1"
  Name="Everything"
  Description="Allows unrestricted access to all resources covered by
    built-in permissions">
  <IPermission class="System.Security.Permissions.EnvironmentPermission,
    mscorlib, Version=2.0.0.0, Culture=neutral,
    PublicKeyToken=b77a5c561934e089" version="1" Unrestricted="true" />
  <IPermission class="System.Security.Permissions.FileDialogPermission,
    mscorlib, Version=2.0.0.0, Culture=neutral,
    PublicKeyToken=b77a5c561934e089" version="1" Unrestricted="true" />
  <IPermission class="System.Security.Permissions.FileIOPermission,
    mscorlib, Version=2.0.0.0, Culture=neutral,
    PublicKeyToken=b77a5c561934e089" version="1" Unrestricted="true" />
  <IPermission class="System.Security.Permissions.IsolatedStorageFilePermission,
    mscorlib, Version=2.0.0.0, Culture=neutral,
    PublicKeyToken=b77a5c561934e089" version="1" Unrestricted="true" />
  <IPermission class="System.Security.Permissions.ReflectionPermission,
    mscorlib, Version=2.0.0.0, Culture=neutral,
    PublicKeyToken=b77a5c561934e089" version="1" Unrestricted="true" />
  <IPermission class="System.Security.Permissions.RegistryPermission,
    mscorlib, Version=2.0.0.0, Culture=neutral,
    PublicKeyToken=b77a5c561934e089" version="1" Unrestricted="true" />
  <IPermission class="System.Security.Permissions.SecurityPermission,
    mscorlib, Version=2.0.0.0, Culture=neutral,
    PublicKeyToken=b77a5c561934e089" version="1"
    Flags="Assertion, UnmanagedCode, Execution, ControlThread,
      ControlEvidence, ControlPolicy, SerializationFormatter,
      ControlDomainPolicy, ControlPrincipal, ControlAppDomain,
      RemotingConfiguration, Infrastructure, BindingRedirects" />
  <IPermission class="System.Security.Permissions.UIPermission,
    mscorlib, Version=2.0.0.0, Culture=neutral,
    PublicKeyToken=b77a5c561934e089" version="1" Unrestricted="true" />
  <IPermission class="System.Security.Permissions.KeyContainerPermission,
    mscorlib, Version=2.0.0.0, Culture=neutral,
    PublicKeyToken=b77a5c561934e089" version="1" Unrestricted="true" />
  <IPermission class="System.Net.DnsPermission, System, Version=2.0.0.0,
    Culture=neutral, PublicKeyToken=b77a5c561934e089" version="1"
    Unrestricted="true" />
  <IPermission class="System.Windows.Forms.WebBrowserPermission, System,
    Version=2.0.0.0, Culture=neutral,
    PublicKeyToken=b77a5c561934e089" version="1" Unrestricted="True" />
  <IPermission class="System.Drawing.Printing.PrintingPermission,
    System.Drawing, Version=2.0.0.0, Culture=neutral,
    PublicKeyToken=b03f5f7f11d50a3a" version="1" Unrestricted="true" />
  <IPermission class="System.Net.SocketPermission, System,
    Version=2.0.0.0, Culture=neutral,
    PublicKeyToken=b77a5c561934e089" version="1" Unrestricted="true" />
  <IPermission class="System.Net.WebPermission, System,
```

```

    Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
    version="1" Unrestricted="true" />
  <IPermission class="System.Diagnostics.EventLogPermission, System,
    Version=2.0.0.0, Culture=neutral,
    PublicKeyToken=b77a5c561934e089" version="1" Unrestricted="true" />
  <IPermission class="System.Diagnostics.PerformanceCounterPermission,
    System, Version=2.0.0.0, Culture=neutral,
    PublicKeyToken=b77a5c561934e089" version="1" Unrestricted="true" />
  <IPermission class="System.Data.OleDb.OleDbPermission,
    System.Data, Version=2.0.0.0, Culture=neutral,
    PublicKeyToken=b77a5c561934e089" version="1" Unrestricted="true" />
  <IPermission class="System.Data.SqlClient.SqlClientPermission,
    System.Data, Version=2.0.0.0, Culture=neutral,
    PublicKeyToken=b77a5c561934e089" version="1" Unrestricted="true" />
  <IPermission class="System.Security.Permissions.StorePermission,
    System.Security, Version=2.0.0.0, Culture=neutral,
    PublicKeyToken=b03f5f7f11d50a3a" version="1" Unrestricted="true" />
</PermissionSet>

```

为了查看所有 XML 格式的权限集，可以使用下面的命令：

```
>caspol.exe -listpset
```

如果要把 XML PermissionSet 配置文件应用到现有的权限集上，给已有的权限集指定新定义，可以使用下面的命令：

```
>caspol.exe -chgpset permissionset.xml MyCustomPermissionSet
```

### 20.5.7 使用强名发布代码

当使用强名确认程序集的身份及其完整性时，使用 .NET 可以把程序集与代码组进行匹配。在通过网络部署程序集时(例如，通过 Internet 发布软件时)，使用强名的情况非常普遍。

对于软件公司而言，如果想利用 Internet 为客户提供代码，就可以创建一个程序集，为它起一个强名。强名可以确保唯一地识别程序集，保护程序集免受改动。软件公司的客户可以把强名合并到他们的代码访问安全策略中，然后，明确地把权限赋予与强名相匹配的程序集。如第 17 章所述，强名包括程序集中所有文件的散列表的校验和。这样，就可以得到自从发布者创建强名之后程序集没有被更改的强证据。

注意，如果应用程序使用了安装程序，则安装程序将安装已经拥有强名的程序集。强名是在程序集发送给客户之前为程序集的发布而产生的，安装程序不运行这些命令。原因是强名要保证程序集离开软件公司之后没有改动，为此，软件公司通常不但把应用程序代码发送给客户，还把程序集的强名副本单独发送给客户。把强名发送给客户时，最好使用比较安全的方式(例如通过传真或加密的电子邮件)，以确保程序集在传送过程中没有更改。

下面看一个示例，在这个示例中，要创建一个具有强名的程序集，这个程序集是通过强名发布的，程序集的接收者可以使用强名把 FullTrust 权限赋予程序集。

首先需要创建一对密钥。强名的创建在第 17 章介绍过了，这里不再重复。用密钥重新建立程序集，可以确保重新计算散列，使强名免受恶意的更改。另外，程序集可以用强名进行唯一标识，这个标识可以在代码组的成员条件中使用。代码组的成员条件将要求程序集的强名与代码组相匹配。



下面的命令说明, 使用指定程序集清单文件中的强名创建一个新的代码组。代码组独立于程序集的版本号, 我们只是要把 FullTrust 权限赋予代码组:

```
>caspol.exe -addgroup 1 -strong -file SimpleExample.exe
      -noname -noversion FullTrust
```

因为强名提供了强有力的证据, 以说明程序集是可信赖的, 所以这个样例中的应用程序可以在任何区域中运行, 包括 Internet 区域。如果使用 `caspol.exe -listgroups` 查看代码组, 就可以看到新的代码组(1.6)和它的相关公钥(十六进制):

Code Groups:

```
1. All code: Nothing
  1.1. Zone - MyComputer: FullTrust
    1.1.1. StrongName -
002400000480000094000000060200000024000052534131000400000100010007D1FA57C4AED9F0
A32E84AA0FAEFD0DE9E8FD6AEC8F87FB03766C834C99921EB23BE79AD9D5DCC1DD9AD23613210290
0B723CF980957FC4E177108FC607774F29E8320E92EA05ECE4E821C0A5EFE8F1645C4C0C93C1AB99
285D622CAA652C1DFAD63D745D6F2DE5F17E5EAF0FC4963D261C8A12436518206DC093344D5AD293:
FullTrust
    1.1.2. StrongName - 00000000000000000400000000000000: FullTrust
  1.2. Zone - Intranet: LocalIntranet
    1.2.1. All code: Same site Web
    1.2.2. All code: Same directory FileIO - 'Read, PathDiscovery'
  1.3. Zone - Internet: Internet
    1.3.1. All code: Same site Web
  1.4. Zone - Untrusted: Nothing
  1.5. Zone - Trusted: Internet
    1.5.1. All code: Same site Web
  1.6. StrongName -
0024000004800000940000000602000000240000525341310004000001000100AD093685E490DC91
2EFF4A3471E4024904E4EAA2367DB539E43D3ED287D954C531BFA51DDBE2D773C4CA8E3210776B0F
4D1A8FC3D2DC978F8CAEA951F29662F9879DC5D1D5755C990AA29DD73E4F4D4FDD223D26F3FBD8B1
7DEF878FFDD2E42A0C9110B42D1E0C452ED8ABE62228A73F2CD9509E9D1631D88DF0238466785CCC:
FullTrust
Success
```

如果要访问程序集中的强名, 可以对程序集清单文件执行 `secutil.exe` 命令。添加 `-hex` 选项, 可以使公钥以十六进制的形式显示出来(与 `caspol.exe` 一样); `-strongname` 参数指定显示强名。执行下面的命令, 可以看到一个列表, 列表中包括强名公钥、程序集名称以及程序集版本:

```
>secutil.exe -hex -strongname SimpleExample.exe
```

```
Microsoft (R) .NET Framework SecUtil 3.5.21004.1
Copyright (C) Microsoft Corporation. All rights reserved.
```

Public Key =

```
0x0024000004800000940000000602000000240000525341310004000001000100D5133
5D1B5B64BE976AD8B08030F8E36A0DBBC3EEB5F8A18D0E30E8951DA059B440281997D76
0FFF61A6252A284061C1D714EFEE5B329F410983A01DB324FA85BCE6C4E6384A2F3BC1F
FA01E2586816B23888CFADD38D5AA5DF041ACE2F81D9E8B591556852E83C473017A1785
203B12F56B6D9DC23A8C9F691A0BC525D7B7EA
```

Name =

SimpleExample



```
Version =
1.0.0.0
Success
```

已安装的两个强名代码组在默认状态下引用的是什么呢？其中一个 Microsoft 代码的强名公钥，另一个是已经提交给 ECMA 以进行标准化的那部分 .NET 代码的强名密钥，Microsoft 对此的控制权较小。

### 20.5.8 使用证书发布代码

上一节讨论了怎样把强名应用到程序集上，以便系统管理员使用代码访问组，明确地把权限赋予与强名相匹配的程序集。这种安全策略的管理方法非常有效，但是有时它需要在比较高的级别上工作。这样，安全策略的管理员将以软件的发布者(而不是各个软件组件)为基础把权限赋予程序集。当从 Internet 上下载具有 Authenticode 签名的可执行程序时，使用了与以前相似的方法。

为了提供软件发布者的信息，可以利用数字证书来标记程序集，让软件的消费者验证软件发布者的身份。在商业化的环境中，可以从 Verisign 或 Thawte 之类的公司中获取证书。

从软件厂商购买证书(而不是创建自己的证书)的优点是，那些证书可以证明软件有很高的可信度，软件厂商是可信的第三方。但是，为了测试，.NET 提供了一个命令行实用程序，使用它可以创建测试证书。创建证书和使用证书发布软件的过程是相当复杂的，但是，为了说明这个过程，下面给出一个示例，简略地阐述所涉及的问题。

设想有一个名叫 ABC 的公司，下面要为公司的软件产品“ABC Suite”创建证书。首先，创建一个测试证书，方法是执行下面的命令：

```
>makecert -sv abckey.pvk -r -n "CN=ABC Corporation" abccorptest.cer
```

这个命令为 ABC 公司创建了一个测试证书，并把所创建的证书保存到 abccorptest.cer 文件中。-sv abckey.pvk 参数创建一个密钥文件，来保存私钥。在创建密钥文件时，需要输入一个必须记住的密码。

创建证书后，就可以用软件发布证书测试工具(Cert2spc.exe)创建一个软件发布测试证书：

```
>cert2spc abccorptest.cer abccorptest.spc
```

为了使用证书标记程序集，在包含程序集清单的程序集文件中使用 signcode.exe 实用程序。通常，标记程序集的最简单方法就是使用 signtool.exe 的向导模式。键入带 signwizard 参数的 signtool.exe 命令，就可以启动向导。

单击 Next 按钮，指定要标记的文件位置。对于程序集而言，选择包含清单的文件，如 SimpleExample.exe。单击 Next 按钮。在 Signing Options 页面上，选择 Custom 选项，定义以前创建的证书文件。

在下一个对话框中，指定要用于标记程序集的证书。单击 Select from File 按钮，并浏览到 abccorptest.SPC 文件，将会看到如图 20-12 所示的确认对话框。

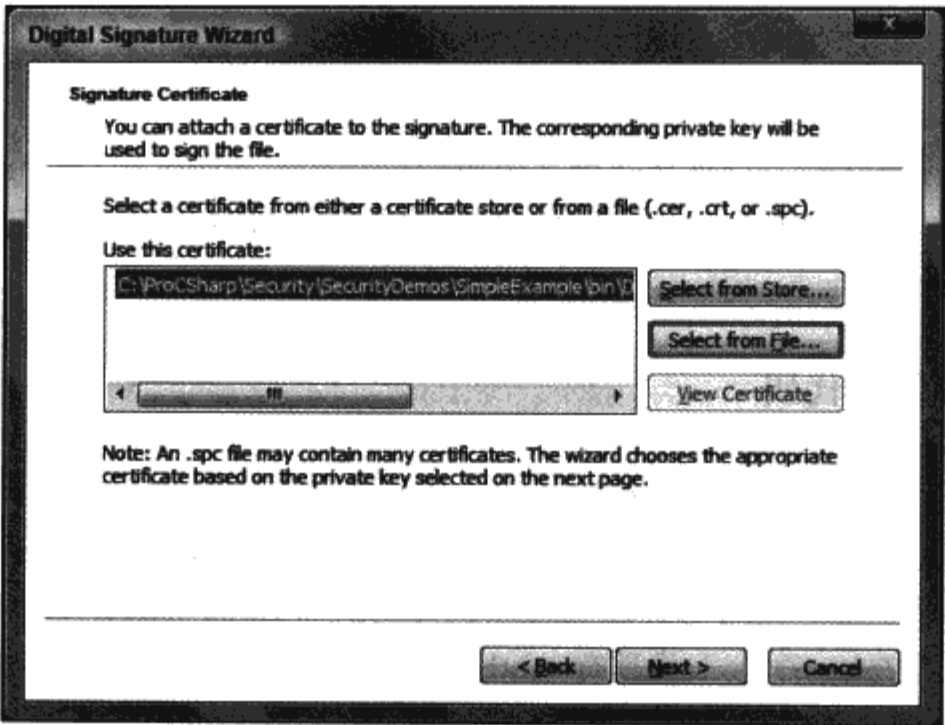


图 20-12

下一个对话框要求指定私钥文件。这个密钥文件 abcney.pvk 是由 makecert 实用程序创建的，因此，可以在图 20-13 中选择一些选项。加密法服务提供程序是一个执行加密标准的应用程序。

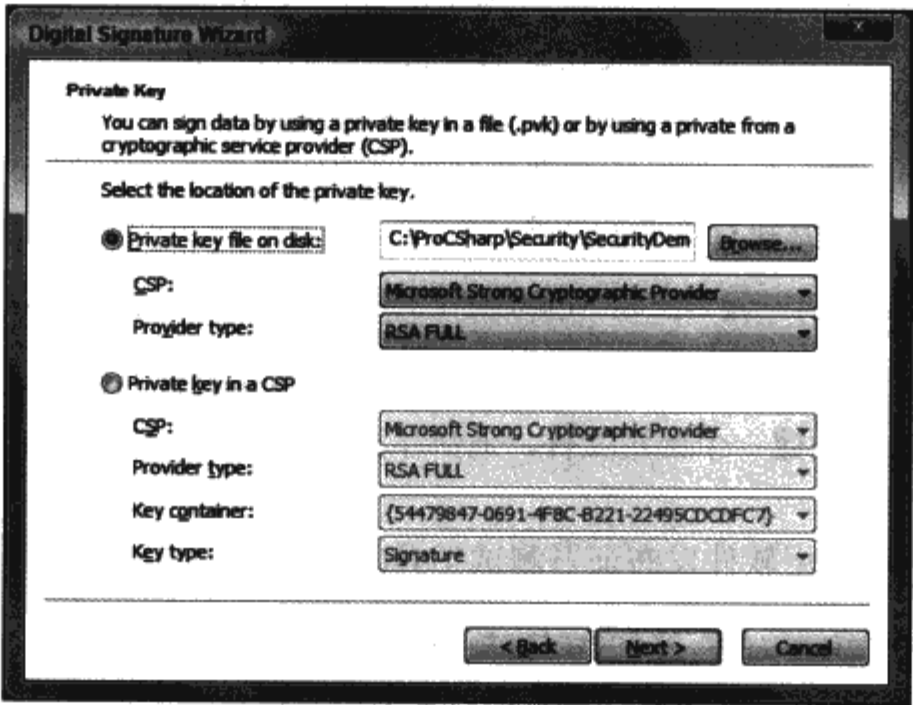


图 20-13

然后，回答一系列用于标记程序集的加密算法问题(md5 或 sha1)，指定应用程序的名称及 URL，最后是一个确认对话框。

由于可执行程序已经用证书标记过，因此，程序集的接收者可以访问软件发布者的强证据。因为可信任的第三方证明了发布者的身份，所以运行库可以检查证书，根据代码的标识对程序集的发布者与信赖度很高的代码组进行匹配。

测试证书必须与可信任的证书一起安装。启动证书管理器 certmgr:

```
>certmgr
```

选择 Trusted Root Certification Authorities 选项卡，再选择树型视图下面的 Certification。单

击 Action | All Task | Import...按钮，导入证书文件。在 Certificate Import 向导中，选择证书文件 abccorptest.cer。

单击 Next 按钮，选择证书库 Trusted Root Certification Authorities，如图 20-14 所示。

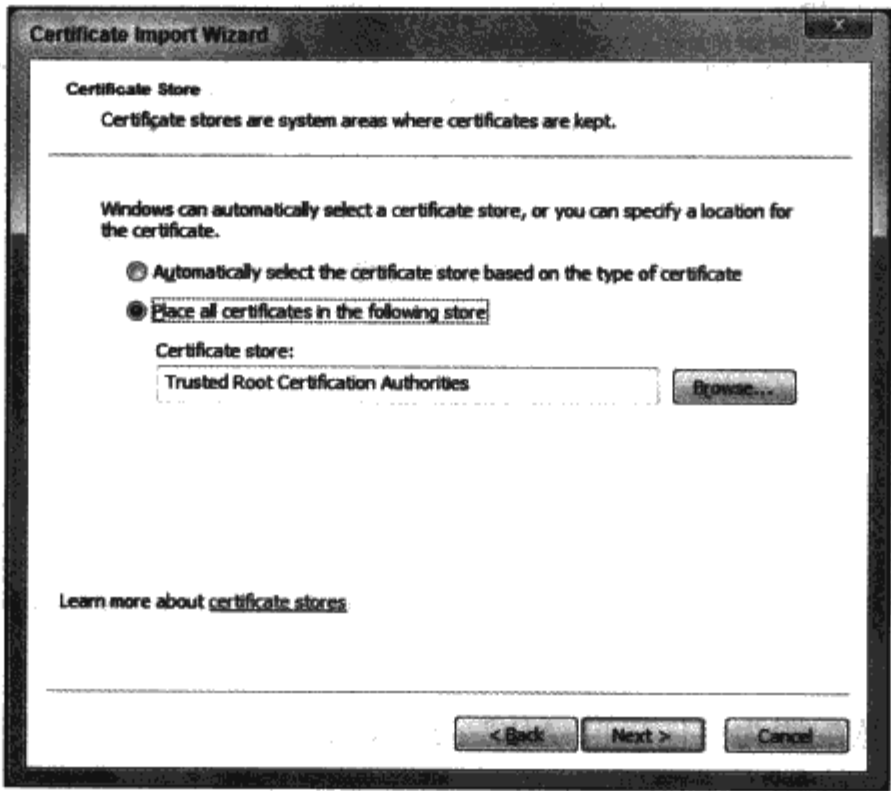


图 20-14

在导入操作完成之前，会显示一个警告对话框，如图 20-15 所示，因为测试证书不能验证。单击 Yes，安装证书。

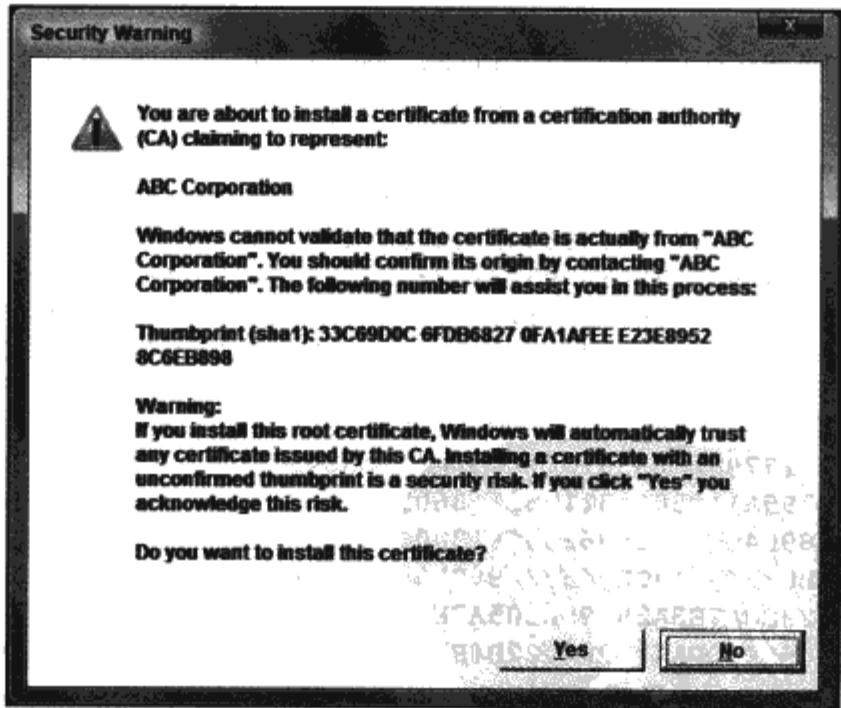


图 20-15

将证书安装为一个可信任的根权限之后，就可以在证书列表中看到它，如图 20-16 所示。

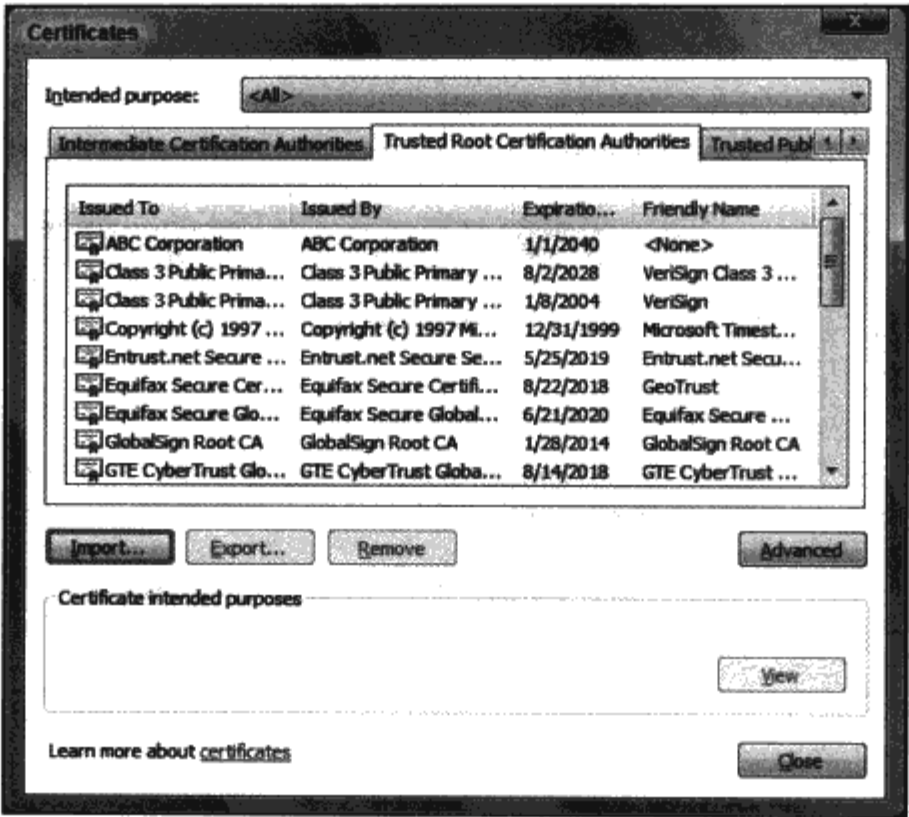


图 20-16

现在，考虑把机器配置为信任来自 ABC 公司的软件。为此，可以创建一个新的代码访问组，让这个组与来自 ABC 公司的软件相匹配。只需使用 secutil.exe 工具，从程序集中获取代表证书的十六进制信息：

```
>secutil.exe -hex -x SimpleExample.exe
```

这个命令将输出如下结果：

```
Microsoft (R) .NET Framework SecUtil 2.0.50727.42
Copyright (c) Microsoft Corp. All rights reserved.

X.509 Certificate =
0x3082020830820171A0030201020210229DECFA1C01E89D46E23F35B6284691300D06092A864886
F70D0101040500301A311830160603550403130F41424320436F72706F726174696F6E301E170D30
37303132323130323334395A170D3339313233313233353935395A301A311830160603550403130F
41424320436F72706F726174696F6E30819F300D06092A864886F70D0101050003818D00308189
02818100B7AE9EC301F76CC661EBF7F9C23E2B4A92F6B4BE318F50B7CB0DA36D4BFECC69E390384A
C33717779A0EAD683536A18B98FC8CA67D10CA05B9FF5AEAA42BCA01D85F95E794427915B9AAA8CC
5C55E9855F5F5D7A5FEEBDF788E2B574E9CBB11B30BC424260415B28A73509048ADDC9BEF28C07E9
C8CE166CB92074D07D17798F0203010001A34F304D304B0603551D010444304280101BA15BEAA3E3
B66F2497401512C79799A11C301A311830160603550403130F41424320436F72706F726174696F6E
8210229DECFA1C01E89D46E23F35B6284691300D06092A864886F70D010104050003818100746FFF
169DE478C34684FAABDBF326A8CEB4588B96C0948BA14D5C73ACF174E5608CBAE8C7BB77B2A38622
E7662BA75F9D0E2A328C8A7E3A28790DC05A7E32557150F8F549E2B3F36F8A609248AF0943877840
48A7A4B0FFA505A7105A4DDDAAF12DC622B4E7956247BEF3D95F187DAEF1A92A34DE83880174ADCF
F93A97BBA8
Success
```

现在，使用下面的命令创建新的代码组，并把 FullTrust 权限应用到 ABC 公司发布的程序集上：

```
>caspol -addgroup 1 -pub -hex
"0x3082020830820171A0030201020210229DECFA1C01E89D46E23F35B6284691300D06092A86488
6F70D0101040500301A311830160603550403130F41424320436F72706F726174696F6E301E170D3
```

```

037303132323130323334395A170D3339313233313233353935395A301A311830160603550403130
F41424320436F72706F726174696F6E30819F300D06092A864886F70D010101050003818D0030818
902818100B7AE9EC301F76CC661EBF7F9C23E2B4A92F6B4BE318F50B7CB0DA36D4BFEC69E390384
AC33717779A0EAD683536A18B98FC8CA67D10CA05B9FF5AEAA42BCA01D85F95E794427915B9AAA8C
C5C55E9855F5F5D7A5FEEBDF788E2B574E9CBB11B30BC424260415B28A73509048ADDC9BEF28C07E
9C8CE166CB92074D07D17798F0203010001A34F304D304B0603551D010444304280101BA15BEAA3E
3B66F2497401512C79799A11C301A311830160603550403130F41424320436F72706F726174696F6
E8210229DECFA1C01E89D46E23F35B6284691300D06092A864886F70D010104050003818100746FF
F169DE478C34684FAABDBF326A8CEB4588B96C0948BA14D5C73ACF174E5608CBAE8C7BB77B2A3862
2E7662BA75F9D0E2A328C8A7E3A28790DC05A7E32557150F8F549E2B3F36F8A609248AF094387784
048A7A4B0FFA505A7105A4DDDAF12DC622B4E7956247BEF3D95F187DAEF1A92A34DE83880174ADC
FF93A97BBA8"
FullTrust

```

上述命令中的参数指定代码组应该添加到最高的层次(1)，还指定代码组的成员条件是 Publisher 类型。最后一个参数指定赋予的权限集(即 FullTrust)。这个命令要求进行如下确认：

```

Microsoft (R) .NET Framework CasPol 2.0.50727.1426
Copyright (C) Microsoft Corporation. All rights reserved.

The operation you are performing will alter security policy.
Are you sure you want to perform this operation? (yes/no)
y
Added union code group with "-pub" membership condition to the Machine level.
Success

```

现在，机器已经配置为完全信任使用 ABC 公司的证书标记过的所有程序集。为了对此进行确认，可以运行 `caspol.exe -lg` 命令，列出新的代码访问组。

另一个检查是使用 `caspol.exe` 命令，列出与程序集相匹配的代码组。

```
>caspol.exe -resolvegroup securityappl1.exe
```

```
Level = Enterprise
```

```
Code Groups:
```

```
1. All code: FullTrust
```

```
Level = Machine
```

```
Code Groups:
```

```
1. All code: Nothing
```

```
1.1. Zone - MyComputer: FullTrust
```

```
1.7. Publisher -
```

```
30818902818100E537F563C2304ECDA2DBEC892DED389C3C17E36500F381BD96E1C76185420F4EEA
46051AD6972139AC7F0BCE3A473F7B9E1DA0DB5F19CCB0A1774C7065DF9E56E4EC6E1F301FEEA899
BD7D37A66F8150A987CD105059B402DE641FB635A7E122F70A1F766D4A2B5030B32BA5189E1C918B
0EF9E87151DACA49EB0160D05181590203010001: FullTrust

```

```
Level = User
```

```
Code Groups:
```

```
1. All code: FullTrust
```

```
Success
```

在结果的中部，程序集已经成功地与新的代码组相匹配，赋予程序集的权限集是 FullTrust。



## 20.6 小结

本章讨论了与.NET 应用程序相关的几个安全性方面。代码访问安全性给应用程序添加了一个安全层,因为它根据应用程序的证书给应用程序授予不同的权限。应用程序的可信程度有多高?这取决于它应用的.NET 权限。权限在权限集中组合,用 User、Enterprise 和 Machine 级别的策略来管理。

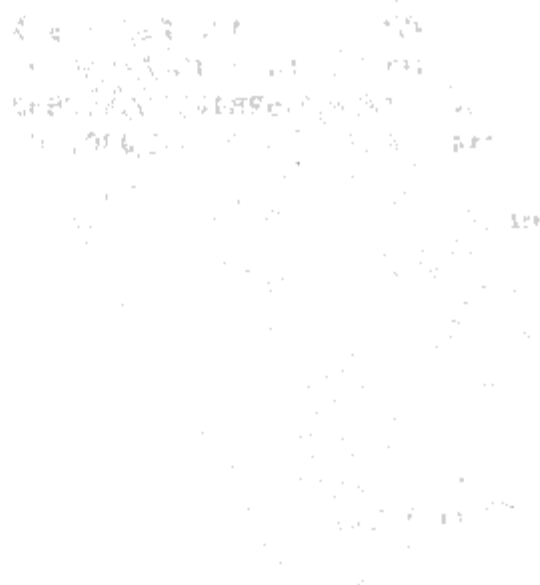
用基于角色的安全性进行身份验证和授权,可以确定哪些用户可以访问应用程序中的特性。用户用标识和 principal 表示,这些类实现了 IIdentity 和 IPrincipal 接口。角色的验证可以在代码中完成,也可以使用属性用简单的方式完成。

本章介绍了加密方法,演示了数据的签名和加密,以安全的方式交换密钥。.NET 提供几个加密算法,包括对称加密和非对称加密算法。.NET 3.5 还支持 Windows Vista 和 Windows Server 2008 中的 Cryptography Next Generation。

使用访问控制表还可以确定如何读取和修改对操作系统资源(如文件)的访问权限。ACL 的编程方式与安全管道、注册表键、Active Directory 项以及许多其他操作系统资源的编程方式相同。

我们还学习了如何使用工具如 caspol 管理安全策略,如何发布带证书的代码。

如果应用程序在不同的区域和不同的语言中使用,可以参阅下一章介绍的.NET 的全球化和本地化特性。



# 第 21 章

## 本地化

价值 1.25 亿美元的 NASA 火星气象卫星在 1999 年 9 月 23 日失踪了, 其原因是一个工程组为一个关键的太空操作使用了米制单位, 而另一个工程组使用了英制单位。当编写的应用程序要在世界各国发布时, 必须考虑不同的文化和区域。

不同的文化在日历、数字和日期格式上各不相同。用字母 A-Z 排序也会导致不同的结果。为了使应用程序可应用于全球市场, 就必须对应用程序进行全球化和本地化。

全球化(globalization)用于国际化的应用程序: 使应用程序可以在全球范围内销售。采用全球化策略, 应用程序应根据文化、不同的日历等支持不同的数字和日期格式。本地化(localization)用于为特定的文化翻译应用程序。而字符串的翻译可以使用资源。

.NET 支持 Windows 和 Web 应用程序的全球化和本地化。要使应用程序全球化, 可以使用 System.Globalization 命名空间中的类; 要使应用程序本地化, 可以使用 System.Resources 命名空间支持的资源。

本章将介绍 .NET 应用程序的全球化和本地化。主要内容如下:

- 使用表示文化和区域的类
- 应用程序的国际化
- 应用程序的本地化

### 21.1 System.Globalization 命名空间

System.Globalization 命名空间包含了所有的文化和区域类, 以支持不同的日期格式、不同的数字格式, 甚至不同的日历, 例如 GregorianCalendar 类、HebrewCalendar 类和 JapaneseCalendar 类等。使用这些类可以根据不同的地区显示不同的表示法。

本节讨论使用命名空间 System.Globalization 时要考虑的如下问题:

- Unicode 问题
- 文化和区域
- 显示所有文化及其特性的例子
- 排序

#### 21.1.1 Unicode 问题

一个 Unicode 字符有 16 位, 所以共有 65 536 个 Unicode 字符。这对于当前在信息技术中

使用的所有语言来说够用吗？例如，汉语就需要 80 000 多个字符。但是，Unicode 可以解决这个问题。使用 Unicode 必须区分基本字符和组合字符。可以给一个基本字符添加若干个组合字符，组成一个可显示的字符或一个文本元素。

例如，冰岛的字符 Ogonek，就可以使用基本字符 0x006F(拉丁小字母 o)、字符 0x0328(组合 Ogonek)和 0x0304(组合 Macron)组合而成，如图 21-1 所示。组合字符在 0x0300 到 0x0345 之间定义，对于美国和欧洲市场，预定义的字符可以处理特定的字符。字符 Ogonek 也可以用预定义的字符 0x01ED 来定义。

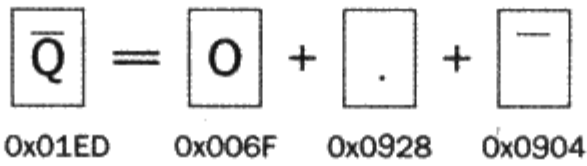


图 21-1

对于亚洲市场，只有汉语需要 80 000 多个字符，但没有这么多的预定义字符。在亚洲语言中，总是要处理组合字符。其问题在于获取显示字符或文本元素的正确数字，得到基本字符，而不是组合字符。命名空间 System.Globalization 提供的类 StringInfo 可以用于处理这个问题。

表 21-1 列出了类 StringInfo 的静态方法，这些方法有助于处理组合字符。

表 21-1

方 法	说 明
GetNextTextElement	返回指定字符串的第一个文本元素(基本字符和所有的组合字符)
GetTextElementEnumerator	返回一个 TextElementEnumerator 对象，允许迭代字符串中的所有文本元素
ParseCombiningCharacters	返回一个整型数组，以引用字符串中的所有基本字符

提示：

一个显示字符可以包含多个 Unicode 字符。要解决这个问题，如果编写的应用程序要在世界各地销售，就不应使用数据类型 char，而应使用 string。string 可以包含由基本字符和组合字符组成的文本元素，但 char 不能。

21.1.2 文化和区域

世界分为若干个文化和区域，应用程序必须知道这些文化和区域的区别。文化是基于用户的语言和文化习惯的一组特性。RFC 1766([www.ietf.org/rfc/rfc1766.txt](http://www.ietf.org/rfc/rfc1766.txt))定义了文化的名称，这些名称根据语言和国家或区域的不同在世界各地使用。例如 en-AU、en-CA、en-GB 和 en-US 分别用于表示澳大利亚、加拿大、英国和美国的英语。

在命名空间 System.Globalization 中，最重要的类是 CultureInfo。这个类表示文化，定义了日历、数字和日期的格式、以及和文化一起使用的排序字符串。

类 RegionInfo 表示区域设置(例如货币)，说明该区域是否使用米制系统。在同一个区域中，可以使用多种语言。例如西班牙区域就有 Basque(eu-ES)、Catalan(ca-ES)、Spanish(es-ES)和 Galician(gl-ES)文化。一个区域可以有多种语言，一种语言也可以在多个区域使用，例如，墨西哥、西班牙、危地马拉、阿根廷和秘鲁等都使用西班牙语。

本章的后面将介绍一个示例应用程序，说明文化和区域的这些特性。

### 1. 特定、中立和不变的文化

在 .NET Framework 中使用文化，必须区分 3 种类型：特定、中立和不变的文化(invariant culture)。

特定的文化与真正存在的文化相关，这种文化用上一节介绍的 RFC 1766 定义。特定的文化可以映射为中立的文化。例如，de 是特定文化 de-AT、de-DE、de-CH 等的中立文化，de 是德语(German)的简写，AT、DE 和 CH 分别是澳大利亚(Austria)、德国(Germany)和瑞士(Switzerland)等国家的简写。

在翻译应用程序时，通常不需要为每个区域翻译，因为澳大利亚和瑞士等国使用的德语没有太大的区别。所以可以使用中立的文化来本地化应用程序，而不需要使用特定的文化。

不变的文化独立于真正的文化。在文件中存储格式化的数字或日期，或通过网络把它们传送到服务器上时，最好使用独立于任何用户设置的文化。

图 21-2 显示了文化类型的相互关系。

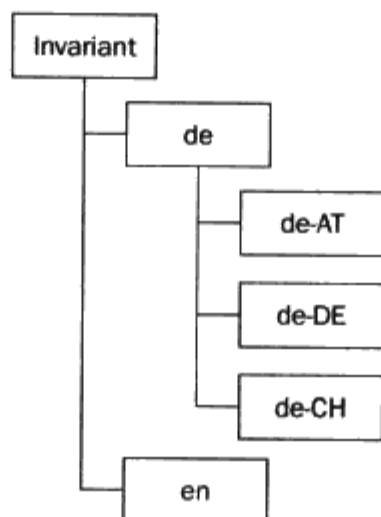


图 21-2

### 2. CurrentCulture 和 CurrentUICulture

设置文化时，必须区分用户界面的文化和数字及日期格式的文化。文化与线程相关，有了这两种文化类型，就可以把两个文化设置应用于线程。Thread 类提供了属性 CurrentCulture 和 CurrentUICulture。属性 CurrentCulture 用于设置与格式化和排序选项一起使用的文化，而属性 CurrentUICulture 用于设置用户界面的语言。

使用 Windows 控制面板中的“区域和语言”选项，就可以改变 CurrentCulture 的默认设置，如图 21-3 所示。使用这个配置，还可以改变文化的默认数字、时间和日期格式。

CurrentUICulture 不依赖于这个配置，而依赖于操作系统的语言。这有一个例外：如果 Windows Vista 或 Windows XP 安装了多语言用户界面(Multi-language User Interface, MUI)，就可以用区域配置改变用户界面的语言，这会影响 CurrentUICulture 属性。

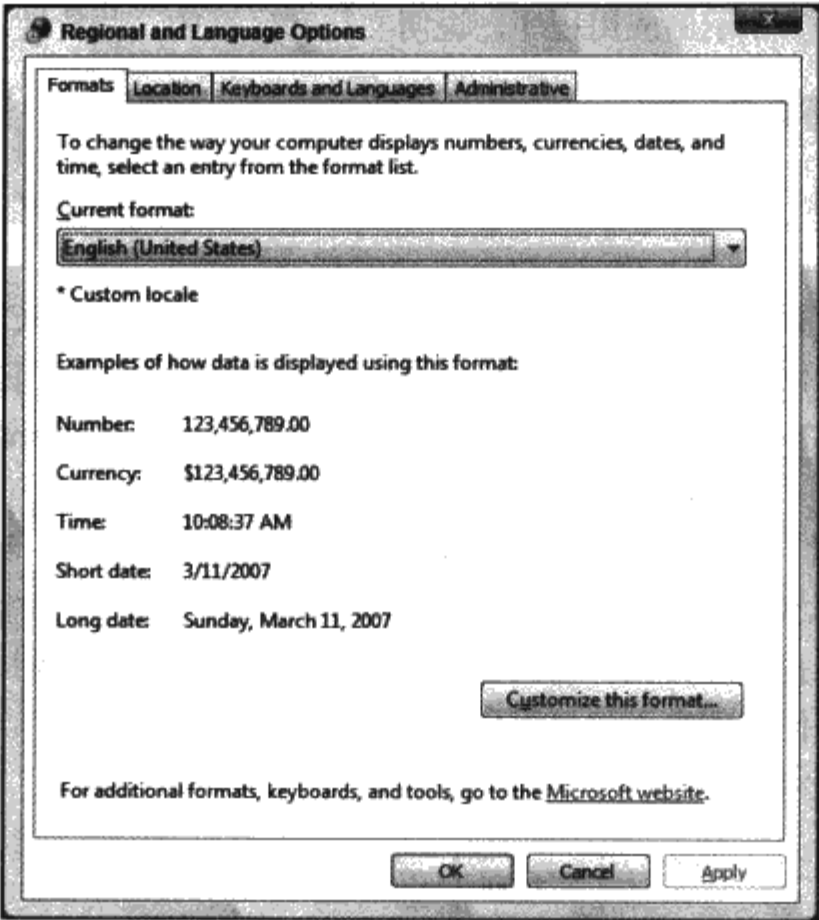


图 21-3

这些设置都使用默认值，在许多情况下，不需要改变默认值。如果需要改变文化，只需把线程的两个文化改为 Spanish 文化，如下面的代码所示：

```
System.Globalization.CultureInfo ci= new
System.Globalization.CultureInfo("es-ES");
System.Threading.Thread.CurrentThread.CurrentCulture = ci;
System.Threading.Thread.CurrentThread.CurrentUICulture = ci;
```

前面已学习了文化的设置，下面讨论 CurrentCulture 设置对数字和日期格式的影响。

3. 数字格式

System 命名空间中的数字结构 Int16、Int32 和 Int64 等都有一个重载的 ToString()方法。这个方法可以根据地域创建不同的数字表示法。对于 Int32 结构，ToString()有下述 4 个重载版本：

```
public string ToString();
public string ToString(IFormatProvider);
public string ToString(string);
public string ToString(string, IFormatProvider);
```

不带参数的 ToString()返回一个没有格式化选项的字符串。也可以给 ToString()传递一个字符串和一个实现 IFormatProvider 接口的类。

该字符串指定表示法的格式。而这个格式可以是标准数字格式化字符串或者图形数字格式化字符串。对于标准数字格式化，字符串是预定义的，C 表示货币，D 表示小数，E 表示科学计数法，F 表示定点输出，G 表示一般输出，N 表示数字，X 表示十六进制。对于图形数字格式化字符串，可以指定位数、节和组分隔符、百分号等。图形数字格式字符串####，###表示两



个3位数块被一个组分隔符分开。

IFormatProvider 接口由 NumberFormatInfo、DateTimeFormatInfo 和 CultureInfo 类实现。这个接口定义了方法 GetFormat(), 返回一个格式对象。

NumberFormatInfo 可以为数字定义定制的格式。使用 NumberFormatInfo 的默认构造函数, 可以创建独立于文化的对象或不变的对象。使用这个类的属性, 可以改变所有的格式化选项, 例如正号、百分号、数字组分隔符和货币符号等。静态属性 InvariantInfo 返回一个与文化无关的只读 NumberFormatInfo 对象。NumberFormatInfo 对象的格式化值取决于当前线程的 CultureInfo, 它由静态属性 CurrentInfo 返回。

下一个示例使用了一个简单的控制台项目。在这段代码中, 第一个示例显示了在当前线程的文化格式中所显示的数字(这里是 English-US, 是操作系统的设置)。第二个示例使用了带有 IFormatProvider 参数的 ToString()方法。CultureInfo 实现 IFormatProvider, 所以创建了一个使用法国文化的 CultureInfo 对象。第三个示例改变了当前线程的文化。使用 Thread 实例的属性 CurrentCulture, 把文化改为德国文化:

```
using System;
using System.Globalization;
using System.Threading;

namespace Wrox.ProCSharp.Localization
{
    class Program
    {
        static void Main()
        {
            int val = 1234567890;

            // culture of the current thread
            Console.WriteLine(val.ToString("N"));

            // use IFormatProvider
            Console.WriteLine(val.ToString("N", new CultureInfo("fr-FR")));

            // change the culture of the thread
            Thread.CurrentThread.CurrentCulture = new CultureInfo("de-DE");
            Console.WriteLine(val.ToString("N"));
        }
    }
}
```

结果如下所示。可以把这个结果与前面列举的美国、英国、法国和德国文化的结果进行比较。

```
1,234,567,890.00
1 234 567 890,00
1.234.567.890,00
```

#### 4. 日期格式

对于日期, 也提供了与数字类似的支持。DateTime 结构有一些把日期转换为字符串的方法。公共实例方法 ToLongDateString()、ToLongTimeString()、ToShortDateString() 和 ToShortTimeString() 都使用当前文化来创建字符串表示法。使用 ToString() 法, 可以指定另一种文化:

```

public string ToString();
public string ToString(IFormatProvider);
public string ToString(string);
public string ToString(string, IFormatProvider);

```

使用 ToString()方法的字符串参数,可以指定预定义格式化字符或定制的格式化字符串,把日期类型转换为字符串类型。类 DateTimeFormatInfo 指定了可能的值。DateTimeFormatInfo 指定的格式字符串有不同的含义,例如 D 表示长日期格式, d 表示短日期格式, ddd 表示星期的缩写, dddd 表示星期的全称, yyyy 表示年份, T 表示长时间格式, t 表示短时间格式。使用 IFormatProvider 参数可以指定文化。使用不带 IFormatProvider 参数的重载方法,表示所使用的是当前线程的文化:

```

DateTime d = new DateTime(2007, 02, 14);

// current culture
Console.WriteLine(d.ToLongDateString());

// use IFormatProvider
Console.WriteLine(d.ToString("D", new CultureInfo("fr-FR")));

// use culture of thread
CultureInfo ci = Thread.CurrentThread.CurrentCulture;
Console.WriteLine(ci.ToString() + ": " + d.ToString("D"));

ci = new CultureInfo("es-ES");
Thread.CurrentThread.CurrentCulture = ci;
Console.WriteLine(ci.ToString() + ": " + d.ToString("D"));

```

这个示例程序的结果说明了使用线程当前文化的 ToLongDateString(), 给 ToString()方法传递一个 CultureInfo 实例,则显示其法国版本,把线程的 CurrentCulture 属性改为 es-ES,则显示其西班牙版本,如下所示。

```

Thursday, February 14, 2008
jeudi 14 février 2008
en-US: Thursday, February 14, 2008
es-ES: jueves, 14 de febrero de 2008

```

### 21.1.3 使用文化

为了全面介绍文化,下面使用一个 WPF 应用程序示例,列出所有的文化,描述文化属性的不同特性。图 21-4 显示了该应用程序在 Visual Studio 2008 WPF 设计器中的用户界面。

在应用程序的初始化阶段,所有可用的文化都添加到应用程序左边的树形视图控件中。这个初始化在 AddCulturesToTree()方法中进行,该方法在窗体类 CultureDemoForm 的构造函数中调用:

```

public CultureDemoForm()
{
    InitializeComponent();

    AddCulturesToTree();
}

```

图 21-4

在方法 `AddCulturesToTree()` 中，将通过静态方法 `CultureInfo.GetCultures()` 获取所有的文化。给这个方法传送 `CultureTypes.AllCultures`，就会返回所有可用文化的数组。该数组用一个  $\lambda$  表达式排序，这个  $\lambda$  表达式要传送给 `Array.Sort()` 方法的第二个参数的 `Comparison` 委托。在 `foreach` 循环中，把每个文化添加到树形视图中。为每个文化创建一个 `TreeNode` 对象，因为 WPF `TreeView` 类使用 `TreeViewItem` 对象来显示。将 `TreeViewItem` 对象的 `Tag` 属性设置为 `CultureInfo` 对象，以便以后访问这个树型视图中的 `CultureInfo` 对象。

`TreeViewItem` 对象添加到树中的什么地方取决于文化类型。如果文化有一个父文化，它就会添加到树的根节点上。要查找父文化，必须把所有的文化保存到一个字典中。第 10 章介绍了字典，第 7 章介绍了  $\lambda$  表达式。

```
// add all cultures to the tree view
public void AddCulturesToTree()
{
    Dictionary< string, TreeViewItem > culturesByName =
        new Dictionary< string, TreeViewItem > ();

    // get all cultures
    var cultures =
        CultureInfo.GetCultures(CultureTypes.AllCultures);
    Array.Sort(cultures, (c1, c2) => c1.Name.CompareTo(c2.Name));

    TreeViewItem[] nodes = new TreeViewItem[cultures.Length];

    int i = 0;
    foreach (var ci in cultures)
    {
        nodes[i] = new TreeNode();
        nodes[i].Header = ci.DisplayName;
        nodes[i].Tag = ci;
        culturesByName.Add(ci.Name, nodes[i]);

        TreeViewItem parent;
```

```

        if (String.IsNullOrEmpty(ci.Parent.Name) & &
            culturesByName.TryGetValue(ci.Parent.Name, out parent))
        {
            parent.Items.Add(nodes[i]);
        }
        else
        {
            treeCultures.Items.Add(nodes[i]);
        }
        i++;
    }
}

```

在用户选择树中的一个节点时，就会调用 TreeView 的 AfterSelect 事件处理程序。在这里，这个处理程序在 TreeCultures\_SelectedItemChanged ()方法中执行。在这个方法中，先调用方法 ClearTextFields()清除所有的字段，再选择 TreeViewItem 对象的 Tag 属性，从树中获取 CultureInfo 对象。接着使用 CultureInfo 对象的属性 Name、NativeName 和 EnglishName 设置一些文本字段。如果 CultureInfo 是一个可以使用 IsNeutralCulture 属性进行查询的中立文化，就设置相应的复选框。

```

private void TreeCultures_SelectedItemChanged(object sender,
    RoutedPropertyChangedEventArgs < object > e)
{
    ClearTextFields();

    // get CultureInfo object from tree
    CultureInfo ci = (CultureInfo)((TreeViewItem)e.NewValue).Tag;

    textCultureName.Text = ci.Name;
    textNativeName.Text = ci.NativeName;
    textEnglishName.Text = ci.EnglishName;

    checkIsNeutral.IsChecked = ci.IsNeutralCulture;
}

```

然后获取文化的日历信息。CultureInfo 类的 Calendar 属性返回特定文化的默认 Calendar 对象。因为 Calendar 类没有名称属性，所以需要使用基类的 ToString()方法获取类的名称，并删除要在文本字段 textCalendar 中显示的这个字符串的命名空间。

因为一种文化可能支持多种日历，所以 OptionalCalendars 属性返回额外支持的 Calendar 对象数组。这些可选的日历显示在列表框 listCalendars 中。派生自 Calendar 的 GregorianCalendar 类还有一个 CalendarType 属性，它列出了 Gregorian 日历的类型。这个类型可以是 GregorianCalendarTypes 枚举中的一个值：Arabic、MiddleEastFrench、TransliteratedFrench、USEnglish 或 Localized，这取决于文化。使用 Gregorian 日历，类型还可以显示在列表框中。

```

// default calendar
textCalendar.Text = ci.Calendar.ToString().
    Remove(0, 21).Replace("Calendar", "");

// fill optional calendars
listCalendars.Items.Clear();
foreach (Calendar optCal in ci.OptionalCalendars)
{
    StringBuilder calName = new StringBuilder(50);
    calName.Append(optCal.ToString());
    calName.Remove(0, 21);
    calName.Replace("Calendar", "");
}

```



```
// for GregorianCalendar add type information
GregorianCalendar = optCal as GregorianCalendar;
if (gregCal != null)
{
    calName.AppendFormat(" {0}", gregCal.CalendarType.ToString());
}
listCalendars.Items.Add(calName.ToString());
}
```

接着在 if 语句中使用 `!ci.IsNeutralCulture`，检查文化是否为特定文化(不是中立文化)。使用方法 `ShowSamples()` 显示数字和日期示例。这个方法将在下一步执行。使用方法 `ShowRegionInformation()` 显示区域的一些信息。对于不变的文化，只能显示数字和日期示例，不能显示区域信息。不变的文件与实际的语言无关，所以与区域也无关。

```
//display number and date samples
if(!ci.IsNeutralCulture)
{
    groupSamples.Enabled = true;
    ShowSamples(ci);

    //invariant culture doesn't have a region
    if (ci.ThreeLetterISOLanguageName == "IVL")
    {
        groupRegion.Enabled = false;
    }
    else
    {
        groupRegion.Enabled = true;
        ShowRegionInformation(ci.Name);
    }
}
else // neutral culture: no region, no number/date formatting
{
    groupSamples.Enabled = false;
    groupRegionInformation.Enabled = false;
}
}
```

为了显示一些本地的数字和日期，把 `CultureInfo` 类型的选中对象传送给 `ToString()` 方法的 `IFormatProvider` 参数。

```
private void ShowSamples(CultureInfo ci)
{
    double number = 9876543.21;
    textSampleNumber.Text = number.ToString("N", ci);

    DateTime today = DateTime.Today;
    textSampleDate.Text = today.ToString("D", ci);

    DateTime now = DateTime.Now;
    textSampleTime.Text = now.ToString("T", ci);
}
```

为了显示与 `RegionInfo` 对象相关的信息，通过在方法 `ShowRegionInformation()` 中传送选中的文化标识符，构造一个 `RegionInfo` 对象，然后访问属性 `DisplayName`、`Currency Symbol`、`ISOCurrencySymbol` 和 `IsMetric`，显示这些信息。



```
private void ShowRegionInformation(String culture)
{
    RegionInfo ri = new RegionInfo(culture);
    textRegionName.Text = ri.DisplayName;
    textCurrency.Text = ri.CurrencySymbol;
    textCurrencyName.Text = ri.ISOCurrencySymbol;
    checkIsMetric.Checked = ri.IsMetric;
}
```

启动应用程序，在树形视图中就会看到所有的文化，选择一个文化后，就会列出该文化的特性，如图 21-5 所示。

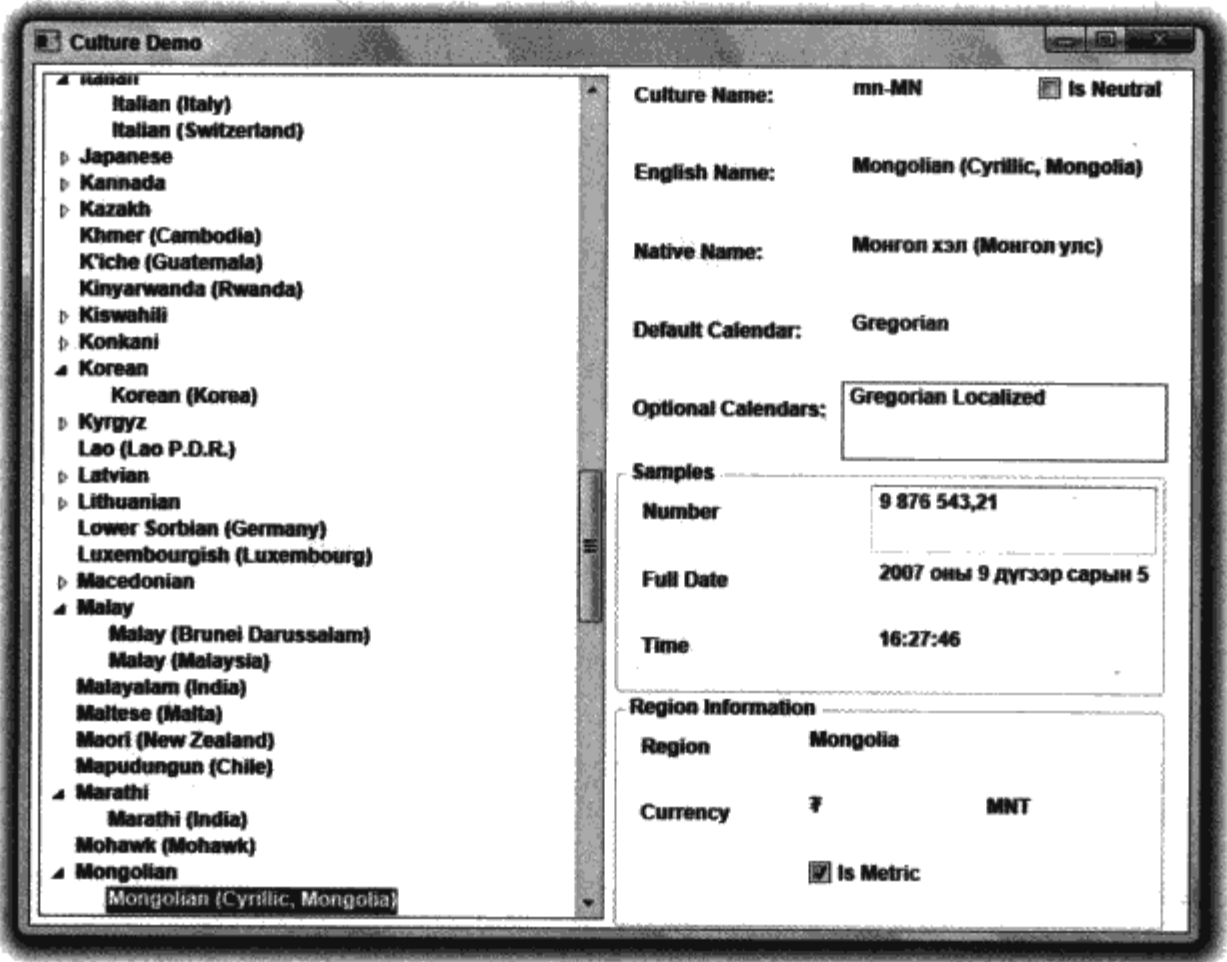


图 21-5

21.1.4 排序

排序字符串取决于文化。一些文化有不同的排列顺序。例如在芬兰，字符 V 和 W 就是相同的。在默认情况下，为排序而比较字符串的算法要使用与文化相关的排序方式，即排序依赖于文化。

为了演示芬兰的排序方式，下面创建一个控制台应用程序小示例，对数组中尚未排序的美国州名进行排序。我们将使用 System.Collections、System.Threading 和 System.Globalization 命名空间中的类，所以必须声明这些命名空间。下面的方法 DisplayName()用于在控制台上显示数组或集合中的所有元素：

```
static void DisplayNames(string title, IEnumerable<string> e)
{
    Console.WriteLine(title);
    foreach (string s in e)
```

```

        Console.Write(s + " - ");
        Console.WriteLine();
        Console.WriteLine();
    }

```

在 Main()方法中, 在创建了包含一些美国州名的数组后, 就把线程属性 CurrentCulture 设置为 Finnish 文化, 这样, 下面的 Array.Sort()方法就使用芬兰的排列顺序。调用方法 DisplayName() 在控制台上显示所有的州名:

```

static void Main()
{
    string[] names = {"Alabama", "Texas", "Washington",
                     "Virginia", "Wisconsin", "Wyoming",
                     "Kentucky", "Missouri", "Utah", "Hawaii",
                     "Kansas", "Louisiana", "Alaska", "Arizona"};
    Thread.CurrentThread.CurrentCulture =
        new CultureInfo("fi-FI");

    Array.Sort(names);
    DisplayNames("Sorted using the Finnish culture", names);
}

```

在以芬兰排列顺序第一次显示美国州名后, 数组将再次排序。如果希望排序独立于用户的文化, 就可以使用不变的文化。在已排序的数组要发送到服务器上, 或存储到某个地方时, 就可以采用这种方式。

为此, 给 Array.Sort()方法传送第二个参数。Sort()方法希望第二个参数是实现 IComparer 的一个对象。System.Collections 命名空间中的 Comparer 类实现 IComparer 接口。Comparer.DefaultInvariant 返回一个 Comparer 对象, 该对象使用不变的文化比较数组值, 以进行独立于文化的排序。

```

// sort using the invariant culture
Array.Sort(names, System.Collections.Comparer.DefaultInvariant);
DisplayNames("Sorted using the invariant culture", names);
}

```

这个程序的输出显示了用 Finnish 文化进行排序的结果和独立于文化的排序结果。在使用独立于文件的排序方式时, Virginia 排在 Washington 的前面。用 Finnish 文化进行排序时, Virginia 排在 Washington 的后面。

```

Sorted using the Finnish culture
Alabama - Alaska - Arizona - Hawaii - Kansas - Kentucky - Louisiana - Missouri
- Texas - Utah - Washington - Virginia - Wisconsin - Wyoming -

Sorted using the invariant culture
Alabama - Alaska - Arizona - Hawaii - Kansas - Kentucky - Louisiana - Missouri
- Texas - Utah - Virginia - Washington - Wisconsin - Wyoming -

```

#### 提示:

如果对集合进行的排序应独立于文化, 该集合就必须用不变的文化进行排序。在把排序结果发送给服务器或存储在文件中时, 这种方式尤其有效。

除了依赖地域的格式化和测量系统之外, 文本和图片也可能因文化的不同而不同。此时就需要使用资源了。

## 21.2 资源

像图片或字符串表这样的资源可以放在资源文件或辅助程序集中。在本地化应用程序时，这种资源非常有用，.NET 对本地化资源的搜索提供了内置支持。

在说明如何使用资源本地化应用程序之前，先讨论如何创建和读取资源，而无需考虑方言。

### 21.2.1 创建资源文件

资源文件包含图形、字符串表等。要创建资源文件，可以使用一般的文本文件，或者是利用 XML 的.resX 文件。下面从一个简单的文本文件开始。

内嵌字符串表的资源可以使用一般的文本文件来创建。该文本文件只是把字符串赋予键。该键的名称可以在程序中用于获取数值。键和数值中都可以包含空格。

这个例子显示了文件 strings.txt 中的一个简单字符串表：

```
Title = Professional C#  
Chapter = Localization  
Author = Christian Nagel  
Publisher = Wrox Press
```

**提示：**

在保存带 Unicode 字符的文本文件时，必须将文件和相应的编码一起保存。为此，可以在 Save 对话框中选择 Unicode 编码。

### 21.2.2 资源文件生成器

可以使用资源文件生成器工具 resgen.exe 在 strings.txt 的外部创建一个资源文件，键入如下代码：

```
resgen strings.txt
```

会创建文件 strings.resources。这个资源文件可以作为一个外部文件添加到程序集中，或者内嵌到 DLL 或 EXE 中。resgen 还可以创建基于 XML 的.resX 资源文件。创建 XML 文件的一种简单方法是使用 ResGen 本身：

```
resgen strings.txt strings.resX
```

这个命令创建了 XML 资源文件 strings.resX。本章后面在介绍本地化时，将讨论如何使用 XML 资源文件。

自.NET 2.0 以来，resgen 支持强类型化的资源。强类型化的资源用一个访问资源的类表示。这个类可以用 resgen 工具的/str 选项创建：

```
resgen /str:C#,DemoNamespace,DemoResource,DemoResource.cs strings.resX
```

在/str 选项中，按照语言、命名空间、类名和源代码文件名的顺序定义资源。

resgen 工具不支持添加图形。在.NET Framework SDK 示例中，有一个 ResXGen 示例。使

用 ResXGen 可以把图形添加到.resx 文件中。还可以使用 ResourceWriter 类或 ResXResourceWriter 类以编程方式把图形添加到资源中。

### 21.2.3 ResourceWriter

除了使用 resgen 工具创建资源文件外,还可以编写一个简单的程序。ResourceWriter 是 System.Resources 命名空间的一个类,它可以编写二进制资源文件;ResXResourceWriter 编写基于 XML 的资源文件。这两个类也支持图形和其他可串行化的对象。在使用 ResXResourceWriter 类时,必须引用 System.Windows.Forms 程序集。

下面的代码使用构造函数和文件名 Demo.resx 创建一个 ResXResourceWriter 对象 rw。在创建了一个实例后,使用 ResXResourceWriter 类的 AddResource()方法可以添加至多 2GB 的资源。AddResource()的第一个参数指定资源名,第二个参数指定数值。可以使用 Image 类的一个实例来添加图形资源。要使用 Image 类,必须引用 System.Drawing 程序集,还要添加 using 指令,以打开命名空间 System.Drawing。

下面打开文件 logo.gif,创建一个 Image 对象。必须把图形复制到可执行文件的目录下,或者在 Image.FromFile()方法的参数中指定图形的完整路径。using 语句指定应在 using 块的尾部自动释放图形资源。把其他简单的字符串资源添加到 ResXResourceWriter 对象中。ResXResourceWriter 类的 Close()方法会自动调用 ResourceWriter.Generate(),最后把资源写入文件 Demo.resx 中:

```
using System;
using System.Resources;
using System.Drawing;

class Program
{
    static void Main()
    {
        ResXResourceWriter rw = new ResXResourceWriter("Demo.resx");
        using (Image image = Image.FromFile("logo.gif"))
        {
            rw.AddResource("WroxLogo", image);
            rw.AddResource("Title", "Professional C#");
            rw.AddResource("Chapter", "Localization");
            rw.AddResource("Author", "Christian Nagel");
            rw.AddResource("Publisher", "Wrox Press");
            rw.Close();
        }
    }
}
```

启动这个小程序,创建嵌入了图像 logo.gif 的资源文件 Demo.resx,这个文件将用于下面的一个 Windows 应用程序。

### 21.2.4 使用资源文件

使用 C#命令行编译器 csc.exe 和/resource 选项,或直接使用 Visual Studio 2008,可以把资源文件添加到程序集中。为了说明如何在 Visual Studio 2008 中使用资源文件,下面创建一个



C# Windows 应用程序 ResourceDemo。

在 Solution Explorer 的弹出菜单(Add | Add Existing Item)中, 把前面创建的资源文件 Demo.resources 添加到这个项目中。默认情况下, 这个资源的 Build Action 设置为 Embedded Resource, 这样, 这个资源就嵌入到输出的程序集中, 如图 21-6 所示。

把应用程序的 Neutral Language 设置为主要语言, 如 English(United States), 如图 21-7 所示。



图 21-6

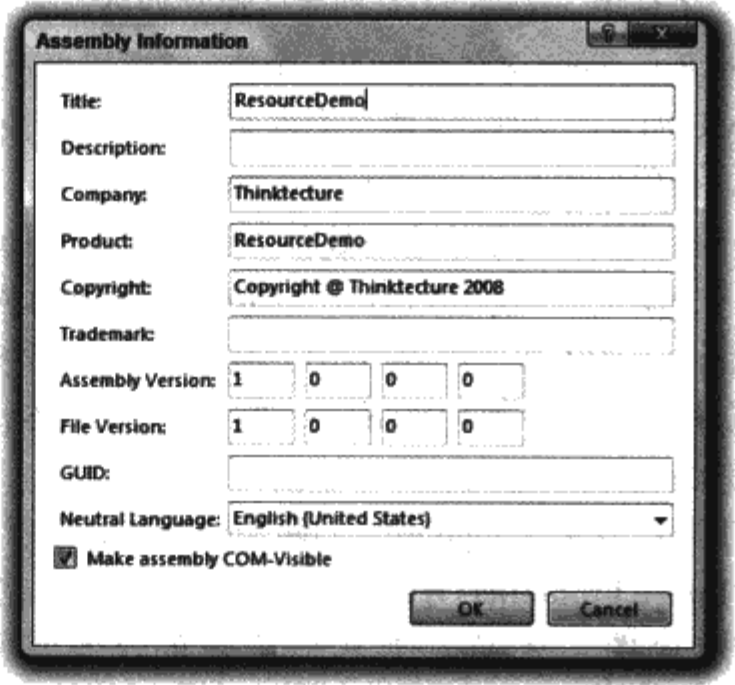


图 21-7

改变这个设置, 会在 assemblyinfo.cs 文件中添加属性[NeutralResourceLanguageAttribute]:

```
[assembly: NeutralResourceLanguageAttribute("en-US")]
```

设置这个选项会提高 ResourceManager 的性能, 因为它会更快找到 en-US 的资源, 该资源还会用作默认的反馈。使用这个属性也可以通过构造函数的第二个参数指定默认资源的位置。使用 UltimateResourceFallbackLocation 枚举可以指定要在主程序集或辅助程序集(值 MainAssembly 和 Satellite)中存储的默认资源。

建立项目后, 使用 ildasm 查看生成的程序集时, 会在程序集清单中看到.mresource, 如图 21-8 所示。它声明了程序集中资源的名称。如果.mresource 声明为 public(与本例一样), 该资源就会从程序集中导出, 且可以用于其他程序集。如果.mresource 声明为 private, 则表示该资源不能导出, 只能用于该程序集内部。

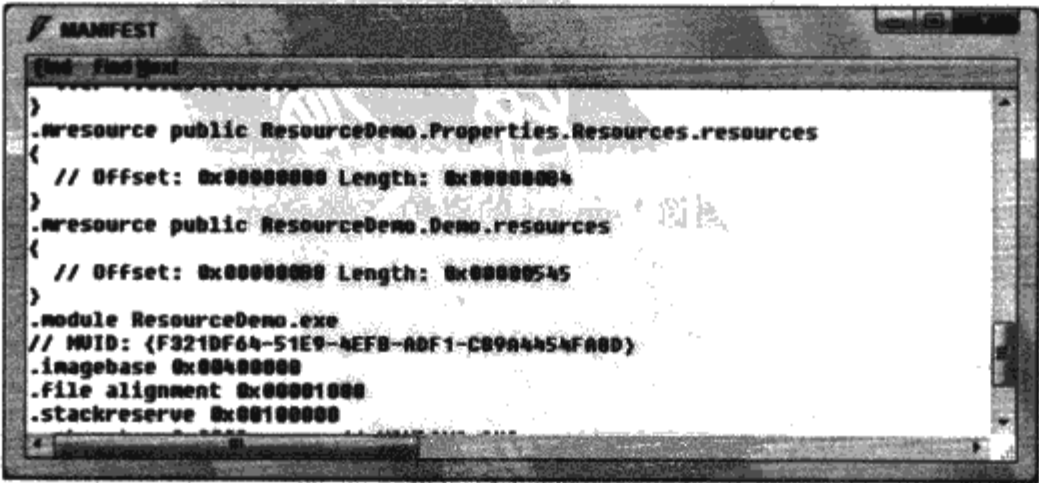


图 21-8



在使用 Visual Studio 2008 把资源添加到程序集中时, 资源总是公共的, 如图 21-8 所示。如果使用程序集生成工具来创建程序集, 就可以使用命令行选项来区分是添加了公共资源, 还是添加了私有资源。选项 `/embed:demo.resources,Y` 把资源作为公共资源添加, 而 `/embed:demo.resources,N` 把资源作为私有资源添加。

#### 提示:

如果程序集是使用 Visual Studio 2008 创建的, 以后就可以修改资源的可见性。使用 `ilasm` 打开程序集, 用 `File | Dump` 生成一个 MSIL 源文件。接着就可以使用文本编辑器来修改 MSIL 代码了。使用文本编辑器可以把 `.mresource public` 改为 `.mresource private`。利用工具 `ilasm`, 可以用这段 MSIL 源代码重新生成程序集: `ilasm /exe ResourceDemo.il`。

在该 Windows 应用程序中, 把 Windows Forms 元素从工具箱拖放到设计器中, 添加一些文本框和一个图形, 以显示资源中的值。修改文本框和标签的 `Text` 和 `Name` 属性, 如下面的代码所示。将 `PictureBox` 的 `Name` 属性改为 `logo`。图 21-9 显示了窗体设计器中的最终窗体, `PictureBox` 控件显示为一个放在左上角的没有网格的矩形。

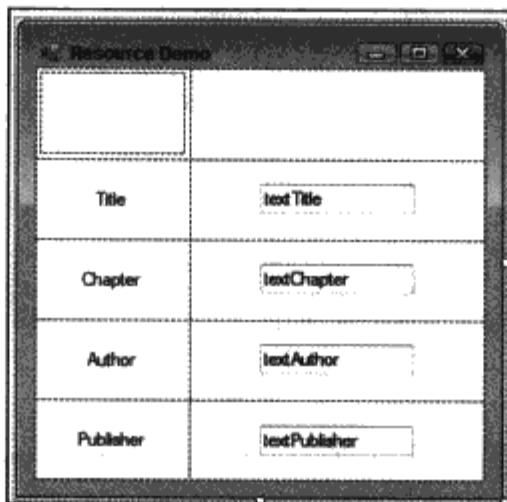


图 21-9

要访问嵌入的资源, 可以使用 `System.Resources` 命名空间中的 `ResourceManager` 类。把嵌入了资源的程序集传递给 `ResourceManager` 类的构造函数。在本例中, 是把资源嵌入到正在执行的程序集中, 所以应把 `Assembly.GetExecutingAssembly()` 的结果作为构造函数的第二个参数。第一个参数是资源的根名。根名由命名空间和资源文件名(不带资源扩展名)组成。如前所述, 使用 `ildasm` 来显示该名称。为此, 只需删除资源的扩展名 `resources` 即可。还可以使用 `System.Reflection.Assembly` 类的 `GetManifestResourceNames()` 方法获取该名称。

```
using System.Reflection;
using System.Resources;

//...

partial class ResourceDemoForm : Form
{
    private System.Resources.ResourceManager rm;

    public ResourceDemoForm()
    {
        InitializeComponent();
    }
}
```

```
Assembly assembly = Assembly.GetExecutingAssembly();

rm = new ResourceManager("ResourceDemo.Demo", assembly);
```

使用 `ResourceManager` 实例 `rm`，通过指定方法 `GetObject()` 和 `GetString()` 的键，就可以获得所有的资源：

```
logo.Image = (Image)rm.GetObject("WroxLogo");
textTitle.Text = rm.GetString("Title");
textChapter.Text = rm.GetString("Chapter");
textAuthor.Text = rm.GetString("Author");
textPublisher.Text = rm.GetString("Publisher");
}
```

运行这段代码，会得到如图 21-10 所示的字符串和图片资源。

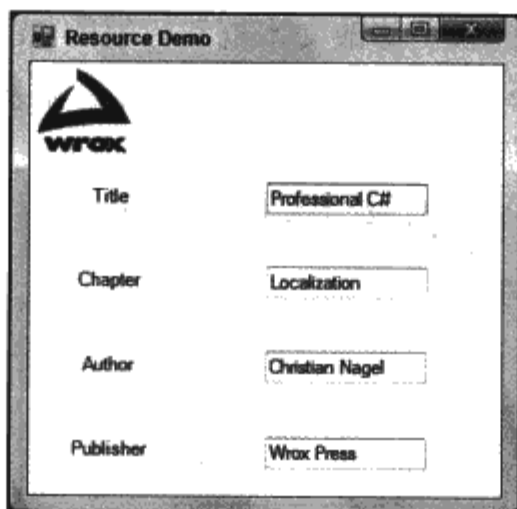


图 21-10

在强类型化的资源中，可以简化前面在 `ResourceDemoForm` 类的构造函数中编写的代码；不需要实例化 `ResourceManager`，使用索引符访问资源，而只需使用属性访问资源名：

```
public ResourceDemoForm()
{
    InitializeComponent();
```

```
pictureLogo.Image = Demo.WroxLogo;
textTitle.Text = Demo.Title;
textChapter.Text = Demo.Chapter;
textAuthor.Text = Demo.Author;
textPublisher.Text = Demo.Publisher;
}
```

要创建强类型化的资源，必须把基于 XML 的资源文件中的 `Custom Tool` 属性设置为 `ResXFileCodeGenerator`。设置这个选项后，就会创建 `Demo` 类(它与资源同名)。这个类的静态属性为所有的资源提供了强类型化的资源名。执行静态属性，就可以使用 `ResourceManager` 对象，该对象在第一次访问时实例化，并高速缓存：

```
/// <summary>
///   A strongly-typed resource class, for looking up localized strings, etc.
/// </summary>
// This class was auto-generated by the StronglyTypedResourceBuilder
// class via a tool like ResGen or Visual Studio.NET.
```

```

// To add or remove a member, edit your .ResX file then rerun ResGen
// with the /str option, or rebuild your Visual Studio project.
[global::System.CodeDom.Compiler.GeneratedCodeAttribute(
    "System.Resources.Tools.StronglyTypedResourceBuilder",
    "2.0.0.0")]
[global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
[global::System.Runtime.CompilerServices.CompilerGeneratedAttribute()]
class Demo {

    private static global::System.Resources.ResourceManager resourceMan;

    private static global::System.Globalization.CultureInfo resourceCulture;

    [global::System.Diagnostics.CodeAnalysis.SuppressMessageAttribute(
        "Microsoft.Performance", "CA1811:AvoidUncalledPrivateCode")]
    internal Demo() {
    }

    /// <summary>
    /// Returns the cached ResourceManager instance used by this class.
    /// </summary>
    [global::System.ComponentModel.EditorBrowsableAttribute(
        global::System.ComponentModel.EditorBrowsableState.Advanced)]
    internal static global::System.Resources.ResourceManager
        ResourceManager {
        get {
            if ((resourceMan == null)) {
                global::System.Resources.ResourceManager temp =
                    new global::System.Resources.ResourceManager(
                        "ResourceDemo.Demo", typeof(Demo).Assembly);
                resourceMan = temp;
            }
            return resourceMan;
        }
    }

    /// <summary>
    /// Overrides the current thread's CurrentUICulture property for all
    /// resource lookups using this strongly typed resource class.
    /// </summary>
    [global::System.ComponentModel.EditorBrowsableAttribute(
        global::System.ComponentModel.EditorBrowsableState.Advanced)]
    internal static System.Globalization.CultureInfo Culture {
        get {
            return resourceCulture;
        }
        set {
            resourceCulture = value;
        }
    }

    /// <summary>
    /// Looks up a localized string similar to "Christian Nagel".
    /// </summary>
    internal static string Author {
        get {

```

```

        return ResourceManager.GetString("Author", resourceCulture);
    }
}

/// <summary>
/// Looks up a localized string similar to "Localization".
/// </summary>
internal static string Chapter {
    get {
        return ResourceManager.GetString("Chapter", resourceCulture);
    }
}

/// <summary>
/// Looks up a localized string similar to "Wrox Press".
/// </summary>
internal static string Publisher {
    get {
        return ResourceManager.GetString("Publisher", resourceCulture);
    }
}

/// <summary>
/// Looks up a localized string similar to "Professional C#".
/// </summary>
internal static string Title {
    get {
        return ResourceManager.GetString("Title", resourceCulture);
    }
}

internal static System.Drawing.Bitmap WroxLogo {
    get {
        return ((System.Drawing.Bitmap) (ResourceManager.GetObject(
            "WroxLogo", resourceCulture)));
    }
}
}
}

```

### 21.2.5 System.Resources 命名空间

在举例之前，我们先复习一下 System.Resources 命名空间中处理资源的类：

- ResourceManager 类可以用于从程序集或资源文件中获取当前文化的资源。使用 ResourceManager 还可以获取特定文化的 ResourceSet。
- ResourceSet 表示特定文化的资源。在创建 ResourceSet 实例时，它会枚举一个实现接口 IResourceReader 的类，并在散列表中存储所有的资源。
- 接口 IResourceReader 用于在 ResourceSet 中枚举资源。类 ResourceReader 实现这个接口。
- ResourceWriter 类用于创建资源文件。ResourceWriter 实现接口 IResourceWriter。
- 类 ResXResourceSet、ResXResourceReader 和 ResXResourceWriter 分别类似于 ResourceSet、ResourceReader 和 ResourceWriter，但创建的是基于 XML 的资源文件 .resX，而不是二进制文件。ResXFileRef 可以用于链接资源，而不是把资源嵌入到 XML 文件中。

## 21.3 使用 Visual Studio 的 Windows Forms 本地化示例

下面创建一个简单的 Windows 应用程序，来说明如何使用 Visual Studio 2008 进行本地化。这个应用程序没有使用复杂的 Windows 窗体，也没有任何实际的内部功能，因为这里主要是说明本地化功能。在自动生成的源代码中，把命名空间改为 `Wrox.ProCSharp.Localization`，类名改为 `BookOfTheDayForm`，命名空间不仅在源文件 `BookOfTheDayForm.cs` 中作了修改，还在项目设置中进行了修改，所以所有生成的资源文件都可以获得这个命名空间。为此，需要在 `Project | Properties` 中选择 `Common Properties`，为所有新建的条目修改该命名空间。

**注意：**

第 31~33 章将详细介绍了 Windows 窗体应用程序。

为了说明本地化的一些问题，这个程序有一个图形、一些文本，一个日期和一个数字。图片显示的是一个已进行了本地化的标记。图 21-11 在 Windows 窗体设计器中显示了该应用程序的这个窗体。

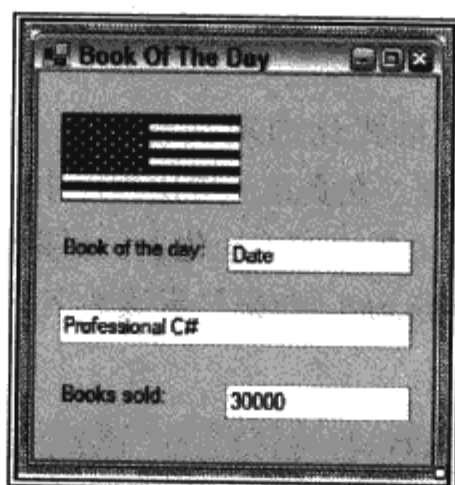


图 21-11

Windows 窗体元素的 Name 和 Text 属性值如表 21-2 所示。

表 21-2

名 称	文 本
labelBookOfTheDay	Book of the day
labelItemsSold	Books sold
textDate	Date
textTitle	Professional C#
textItemsSold	30000
pictureFlag	

除了这个窗体外，还要显示一个消息框，根据当前时间显示不同的问候信息。这说明了动态创建的对话框在进行本地化时必须采用不同的方式。在方法 `WelcomeMessage()` 中，使用



MessageBox.Show()显示一个信息框,在窗体类 BookOfTheDayForm 的构造函数中调用方法 WelcomeMessage(),之后调用 InitializeComponent()。

下面是 WelcomeMessage()方法的代码:

```
public void WelcomeMessage()
{
    DateTime now = DateTime.Now;
    string message;
    if (now.Hour <= 12)
    {
        message = "Good Morning";
    }
    else if (now.Hour <= 19)
    {
        message = "Good Afternoon";
    }
    else
    {
        message = "Good Evening";
    }
    MessageBox.Show(message + " \nThis is a localization sample.");
}
```

窗体中的数字和日期应使用格式化选项来设置。我们添加一个新方法 SetDateAndNumber(),用格式选项来设置这些值。在真正的应用程序中,这些值应从一个 Web 服务或数据库中得到,但在本例中,要把注意力集中在本地化上。日期使用 D 选项来格式化(显示长日期名)。使用图形数字格式化字符串###,###,###来显示该数字,其中"#"表示一个数字,","是一个组分隔符:

```
public void SetDateAndNumber()
{
    DateTime today = DateTime.Today;
    textDate.Text = today.ToString("D");
    int itemsSold = 327444;
    textItemsSold.Text = itemsSold.ToString("###,###,###");
}
```

在 BookOfTheDayForm 类的构造函数中,调用了 WelcomeMessage()和 SetDateAndNumber()方法:

```
public BookOfTheDayForm()
{
    WelcomeMessage();
    InitializeComponent();
    SetDateAndNumber();
}
```

Windows 窗体设计器的一个功能是可以把窗体的 Localizable 属性从 false 改为 true。这个设置的结果是为对话框创建一个基于 XML 的资源文件,存储所有的资源字符串、属性(包括 Windows 窗体元素的位置和大小),嵌入的图形等。另外,对方法 InitializeComponent()的实现

代码也进行了修改：创建类 `System.Resources.ResourceManager` 的一个实例，为了获取文本字段以及图形的值和位置，应使用 `GetObject()` 方法，而不是直接在代码中写入值。`GetObject()` 使用当前线程的 `CurrentUICulture` 属性来查找合适的本地化资源。

下面是在 `BookOfTheDayForm.Designer.cs` 文件中把 `Localizable` 属性设置为 `true` 之前的部分 `InitializeComponent()`，其中设置了 `textTitle` 的所有属性。

```
private void InitializeComponent()
{
    //...
    this.textTitle = new System.Windows.Forms.TextBox();
    //...
    //
    // textTitle
    //
    this.textTitle.Location = new System.Drawing.Point(24, 152);
    this.textTitle.Name = "textTitle";
    this.textTitle.Size = new System.Drawing.Size(256, 20);
    this.textTitle.TabIndex = 2;
    this.textTitle.Text = "Professional C#";
}
```

把 `Localizable` 属性设置为 `true` 后，`InitializeComponent()` 方法的代码会自动修改，如下所示：

```
private void InitializeComponent()
{
    System.ComponentModel.ComponentResourceManager resources = new
        System.ComponentModel.ComponentResourceManager(
            typeof(BookOfTheDayForm));
    //...
    this.textTitle = new System.Windows.Forms.TextBox();
    //...
    resources.ApplyResources(this.textTitle, "textTitle");
}
```

资源管理器是从哪里获取的数据？在把 `Localizable` 属性设置为 `true` 时，就生成了一个资源文件 `BookOfTheDay.resX`。在这个文件中，首先找到 XML 资源的模式，接着找到窗体中的所有元素：`Type`、`Text`、`Location`、`TabIndex` 等。

类 `ComponentResourceManager` 派生自 `ResourceManager`，提供了 `ApplyResources()` 方法。在这个方法中，资源使用第二个参数定义，并应用于第一个参数中的对象。

下面的 XML 片段说明了 `textBoxTitle` 的几个属性：`Location` 属性的值是 13 133，`TabIndex` 属性的值是 2，`Text` 属性设置为 `Professional C#` 等。对于每个值，还存储了值的类型。例如，`Location` 属性的类型是 `System.Drawing.Point`，这个类在程序集 `System.Drawing` 中。

为什么位置和大小也存储在这个 XML 文件中？在转换时，许多字符串都会有完全不同的大小，不再适合于原来的位置。把位置和大小都存储在资源文件中后，需要进行本地化的位置和大小也都存储在这些文件中，从而与 C# 代码分开：

```
<data name="textTitle.Anchor" type="System.Windows.Forms.AnchorStyles,
    System.Windows.Forms">
    <value>Bottom, Left, Right</value>
</data>
<data name="textTitle.Location" type="System.Drawing.Point, System.Drawing">
    <value>13, 133</value>
```

```
</data>
<data name="textTitle.Size" type="System.Drawing.Size, System.Drawing">
  <value>196, 20</value>
</data>
<data name="textTitle.TabIndex" type="System.Int32, mscorlib">
  <value>2</value>
</data>
<data name="textTitle.Text">
  <value xml:space="preserve">Professional C#</value>
</data>
```

在修改其中一些资源值时，不需要直接使用 XML 代码来修改。我们可以在 Visual Studio 2008 设计器中直接修改这些资源。只要修改窗体的 Language 属性和一些窗体元素的属性，就会为指定的语言生成一个新资源。把 Language 属性设置为 German，就会创建德国版本的窗体。把 Language 属性设置为 French，就会创建法国版本的窗体。对于每种语言，都会生成一个属性已被修改的资源文件：BookOfTheDayForm.de.resX 和 BookOfTheDayForm.fr.resX。

表 21-3 是德国版本的窗体需要进行的修改。

表 21-3

德国—名称	值
\$this.Text (窗体的标题)	Buch des Tages
labelItemsSold.Text	Bücher verkauft:
labelBookOfTheDay.Text	Buch des Tages

表 21-4 是法国版本的窗体应进行的修改。

表 21-4

法国—名称	值
\$this.Text (窗体的标题)	Le livre du jour
labelItemsSold.Text	Des livres vendus
labelBookOfTheDay.Text	Le livre du jour

图像不再默认移动到辅助程序集中。但在示例应用程序中，标记应根据国家的不同而不同。为此，必须在 Resources.resx 文件中添加美国标记的图像。这个文件在 Visual Studio Solution Explorer 的 Properties 部分。使用资源编辑器选择 Images 类别，如图 21-12 所示，添加 americanflag.bmp 文件。为了用图像进行本地化，图像必须在所有的语言中有相同的名称。在这里，Resources.resx 文件中的图像名是 Flag。可以在属性编辑器中给图像重命名。在属性编辑器中，还可以修改图像是链接还是嵌入。为了提高资源的性能，图像默认为链接。对于链接的图像，图像文件必须与应用程序一起发布。如果要在程序集中嵌入图像，就可以把 Persistence 属性改为 Embedded。

把 Resource.resx 文件复制到 Resource.de.resx 和 Resource.fr.resx 中，用 GermanFlag.bmp 和 FranceFlag.bmp 替代标记，就可以添加标记的本地化版本。因为只有中立资源需要强类型化的资源类，所以可以给所有特定语言的资源文件清除 CustomTool 属性。

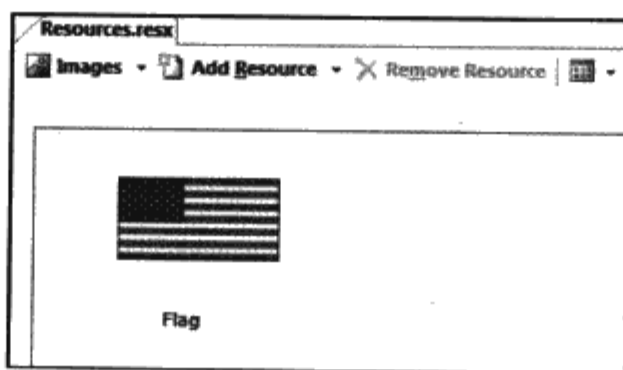


图 21-12

现在编译该项目，为每种语言创建一个辅助程序集。在 debug 目录里(根据目前的配置，可能是 release)，创建语言子目录如 de 和 fr。在这个子目录下，保存了文件 BookOfTheDay.resources.dll。这些文件就是只包含本地化资源的辅助程序集。使用 ildasm 打开这个程序集，可以看到其已嵌入资源的程序集清单，以及一个定义好的地区。这个程序集在程序集属性中有一个地区 de，所以它在 de 子目录下。其中还有一个带有 .mresource 的资源名：它的前缀是命名空间名 Wrox.ProCSharp.Localization，其后是类名 BookOfTheDayForm 和语言代码 de。

### 21.3.1 编程修改文化

在翻译资源，建立辅助程序集后，就可以根据为用户配置的文化获得正确的译文。此时不翻译欢迎信息，它们是以另一种方式翻译的，稍后讨论。

除了系统配置外，还可以把语言代码当作命令行参数传递给应用程序，以便进行测试。修改 BookOfTheDayForm 构造函数，以传递文化字符串，并根据这个字符串设置文化。创建一个 CultureInfo 实例，把它传递给当前线程的 CurrentCulture 和 CurrentUICulture 属性。注意 CurrentCulture 用于格式化，而 CurrentUICulture 用于加载资源。

```
public BookOfTheDayForm(string culture)
{
    if (!String.IsNullOrEmpty(culture))
    {
        CultureInfo ci = new CultureInfo(culture);
        // set culture for formatting
        Thread.CurrentThread.CurrentCulture = ci;
        // set culture for resources
        Thread.CurrentThread.CurrentUICulture = ci;
    }

    WelcomeMessage();

    InitializeComponent();
    SetDateAndNumber();
}
```

在 Program.cs 文件的 Main()方法中实例化 BookOfTheDayForm。在这个方法中，把文化字符串传送给 BookOfTheDayForm 构造函数：

```
[STAThread]
static void Main(string[] args)
{
```



```
string culture = "";
if (args.Length == 1)
{
    culture = args[0];
}

Application.EnableVisualStyles();
Application.SetCompatibleTextRenderingDefault(false);
Application.Run(new BookOfTheDayForm(culture));
}
```

现在可以使用命令行选项启动应用程序，查看格式化选项和在 Windows 窗体设计器中生成的资源。图 21-13 和图 21-14 分别显示了使用 fr-FR 和 de-DE 命令行选项启动的应用程序。

欢迎信息框还有一个问题，这些字符串在程序中都是硬编码的。因为这些字符串都不是窗体中元素的属性，所以窗体设计器就不能像处理 Windows 控件的属性那样，通过改变窗体的 Localizable 属性提取 XML 资源。而必须自己修改代码。



图 21-13



图 21-14

21.3.2 使用定制资源文件

对于欢迎信息，必须翻译硬编码的字符串。英语文本翻译成德语和法语的译文如表 21-5 所示。可以在 Resources.resx 文件中直接编写定制的资源信息和与特定语言相关的派生文本，当然，也可以创建新的资源文件。

表 21-5

名 称	英 语	德 语	法 语
Good_Morning	Good Morning	Guten Morgen	Bonjour
Good_Afternoon	Good Afternoon	Guten Tag	Bonjour
Good_Evening	Good Evening	Guten Abend	Bonsoir
Message1	This is a localization sample.	Das ist ein Beispiel mit Lokalisierung.	C'est un exemple avec la localization.

WelcomeMessage()方法的源代码也必须改为使用资源。在强类型化的资源中，不需要实例化 ResourceMessenger 类，而可以使用强类型化资源的属性：



```

public void WelcomeMessage()
{
    DateTime now = DateTime.Now;
    string message;
    if (now.Hour <= 12)
    {
        message = Properties.Resources.GoodMorning;
    }
    else if (now.Hour <= 19)
    {
        message = Properties.Resources.GoodAfternoon;
    }
    else
    {
        message = Properties.Resources.GoodEvening;
    }
    MessageBox.Show(message + "\n" +
        Properties.Resources.Message1);
}

```

使用英语、德语或法语启动程序时，会得到如图 21-15、21-16 和 21-17 所示的消息框。

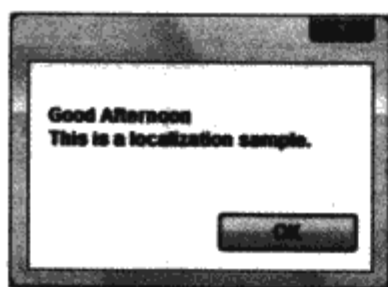


图 21-15

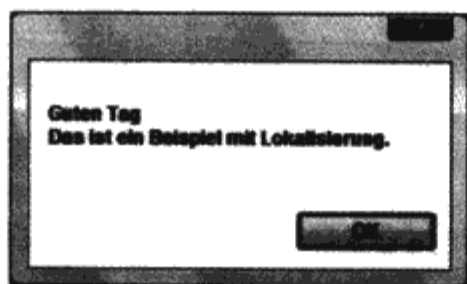


图 21-16

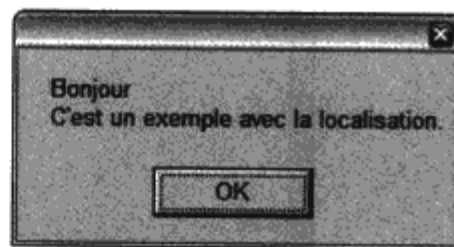


图 21-17

### 21.3.3 资源的自动回退

对于法国和德国版本，我们已经在辅助程序集中包含了所有的资源。如果不是所有标签和文本框的值都要修改，这就没有问题。只需要把要修改的值放在辅助程序集中，其他值放在父程序集中即可。例如，对于 de-at (奥地利)，可以把 GoodAfternoon 资源的值改为 Grüß Gott，但其他值不应修改。在运行时，如果在 de-at 辅助程序集中找不到 Good Morning 资源的值，就应搜索父程序集。de-at 辅助程序集的父亲程序集是 de。如果 de 程序集中也没有这个资源，就应在 de 的父亲程序集(即中枢程序集)中搜索该值。中枢程序集不包含文化代码。

**注意：**

在主程序集的文化代码中，不应定义任何文化！

### 21.3.4 外包翻译

使用资源文件很容易完成外包翻译(outsource translation)的任务。在翻译资源文件时，不需要安装 Visual Studio，一个简单的 XML 编辑器就可以满足许多需要。使用 XML 编辑器的缺点是没有机会重新安排 Windows 窗体元素，如果翻译过来的文本不适合标签或按钮的原边框，其大小是不能改变的。使用 Windows 窗体设计器进行翻译是很自然的选择。

Microsoft .NET Framework SDK 附带的一个工具可以满足所有这些需要：Windows 资源本

地化编辑器 winres.exe(参见图 21-18)。使用该工具的用户无需访问 C#源文件,只需要翻译二进制文件或基于 XML 的资源文件。在完成这些翻译工作后,就可以把资源文件导入到 Visual Studio 项目中,建立辅助程序集了。

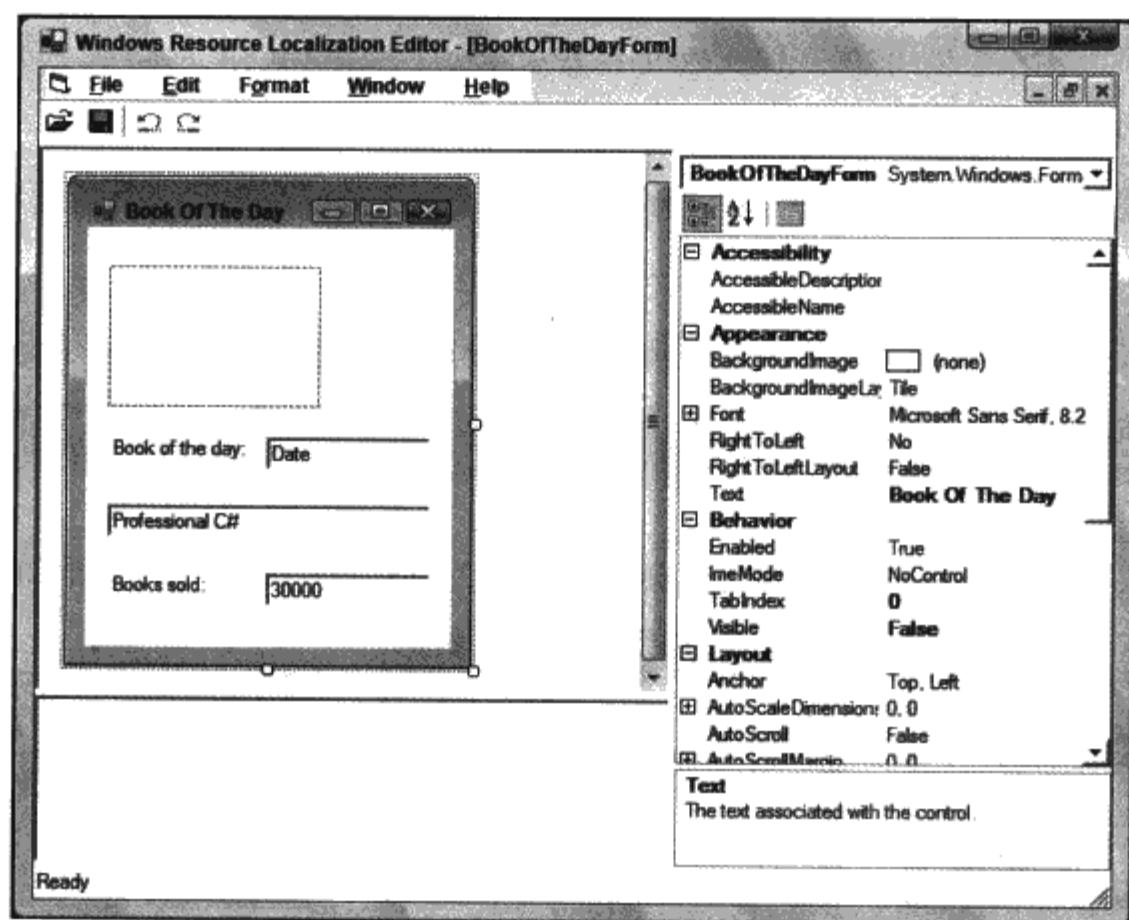


图 21-18

如果不希望翻译工作室改变标签和按钮的大小和位置,且不能处理 XML 文件,就可以发送一个基于文本的简单文件。使用命令行工具 resgen.exe,可以从 XML 文件中创建一个文本文件:

```
resgen myresource.resX myresource.txt
```

在收到翻译工作室完成的翻译工作后,就可以从返回的文本文件中创建一个 XML 文件:

```
resgen myresource.es.txt myresource.es.resX
```

## 21.4 用 ASP.NET 本地化

ASP.NET 应用程序的本地化与 Windows 应用程序的本地化类似。第 37 章将讨论 ASP.NET 应用程序的功能,本节只讨论 ASP.NET 应用程序的本地化问题。ASP.NET 2.0 和 Visual Studio 2008 有许多支持本地化的新特性。本地化和国际化的基本概念与前面讨论的一样,但 ASP.NET 的本地化有一些特殊的问题。

如前所述,在 ASP.NET 中,必须区分用户界面文化和用于格式化的文化。这两种文化都可以在 Web 和页面级上定义,也可以编程定义。

要独立于 Web 服务器的操作系统,文化和用户界面文化可以用 web.config 配置文件中的

<globalization>元素定义:

```
<configuration>
  <system.web>
    <globalization culture = "en-US" uiCulture= "en-US" />
  </system.web>
</configuration>
```

如果特定 Web 页面的配置不同, 则可以使用 Page 指令指定文化:

```
<%Page Language="C#" Culture = "en-US" UICulture= "en-US" %>
```

用户可以用浏览器配置语言。在 Internet Explorer 中, 这个设置用 Language Preference 选项定义, 如图 21-19 所示。

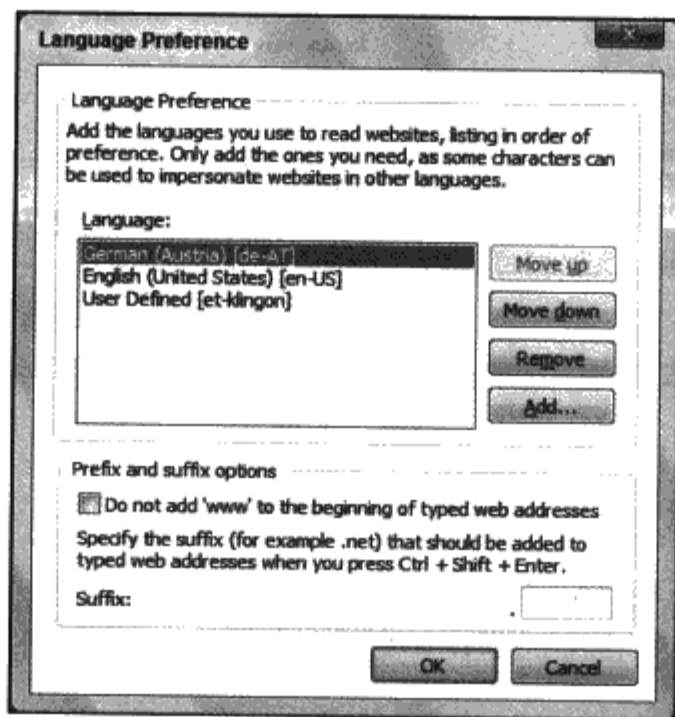


图 21-19

如果页面的语言应根据客户程序的语言设置而不同, 就可以把线程的文化编程设置为客户程序接收的语言设置。ASP.NET 2.0 的一个自动设置可以完成这个任务。把文化设置为 Auto, 就可以根据文化设置指定线程的文化。

```
<%Page Language="C#" Culture = "Auto" UICulture= "Auto" %>
```

在使用资源时, ASP.NET 会区分用于整个 Web 站点的资源和只用于一个页面的资源。

如果资源在一个页面中使用, 就可以在设计视图中选择 Visual Studio 2008 菜单 Tools | Generate Local Resource, 为页面创建资源。这会创建一个子目录 App\_LocalResources, 存储每个页面的资源文件。这些资源可以用与 Windows 应用程序相同的方式进行本地化。Web 控件和本地资源文件之间的关系用 meta:resourcekey 属性指定, 如下面的 ASP.NET 标签控件。LabelResource1 是资源名, 该名称可以在本地资源文件中修改。

```
<asp:Label ID="Label1" Runat="server" Text="Label"
  meta:resourcekey="LabelResource1"></asp:Label>
```

对于要在多个页面上共享的资源，必须创建一个子目录 `Application_Resources`，在这个子目录中，可以添加资源文件(如 `Messages.resx`)及其资源。为了把 Web 控件与这些资源关联起来，可以使用属性编辑器中的 Expressions。单击 Expressions 按钮，打开 Expressions 对话框，如图 21-20 所示。在该对话框中，可以选择表达式类型 `Resources`，设置 `ClassKey` 名(这是资源文件的名称，这里会生成一个强类型化的资源文件)、`ResourceKey` 的名称(资源的名称)。

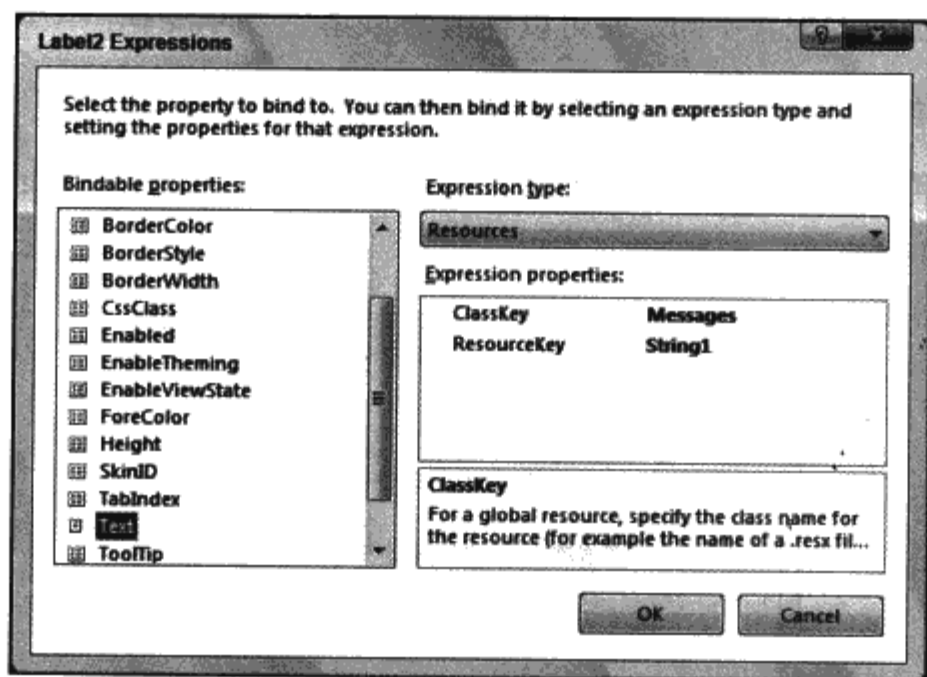


图 21-20

在 ASPX 文件中，用绑定表达式语法`<%$`关联资源：

```
<asp:Label ID="Label1" Runat="server"
    Text="<%$ Resources:Messages, String1 %>">
</asp:Label>
```

## 21.5 用 WPF 本地化

Visual Studio 2008 没有给 WPF(Windows Presentation Foundation)应用程序的本地化提供强大的支持，但不必等到推出下一个版本，就能本地化 WPF 应用程序。WPF 从一开始就内置了本地化支持，本地化应用程序有几个选项。可以使用与 Windows Forms 和 ASP.NET 应用程序类似的 .NET 资源，也可以使用 XAML(XML for Application Markup Language)资源字典。

下面讨论这些选项。WPF 和 XAML 详见第 34 和 35 章。

### 21.5.1 WPF 应用程序

为了演示对 WPF 应用程序使用资源，创建一个简单的 WPF 应用程序，它只包含一个按钮，如图 21-21 所示。

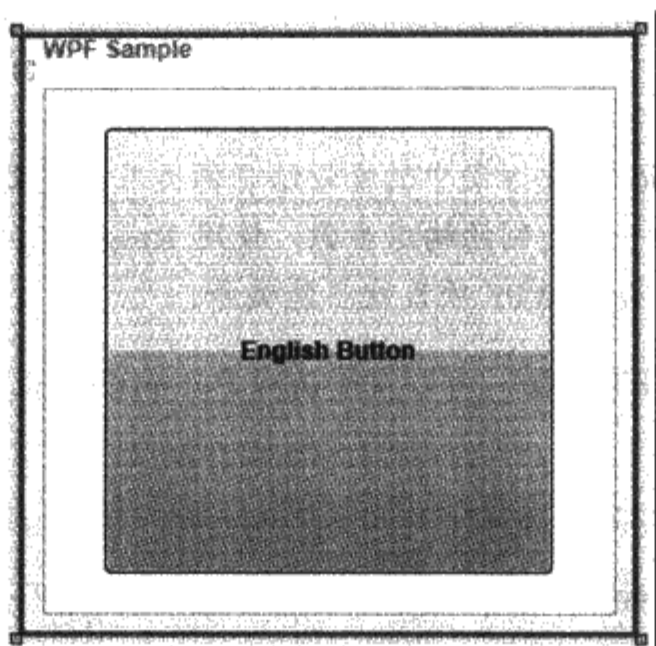


图 21-21

这个应用程序的 XAML 代码如下：

```
< Window x:Class="Wrox.ProCSharp.Localization.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="WPF Sample" Height="300" Width="300" >
  < Grid >
    < Button Name="button1" Margin="30,20,30,20" Click="Button_Click"
      Content="English Button" / >
  < /Grid >
< /Window >
```

在按钮的单击事件处理程序中，弹出一个包含示例消息的消息框：

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show("English Message");
}
```

### 21.5.2 .NET 资源

可以用处理其他应用程序的方式把 .NET 资源添加到 WPF 应用程序中。在文件 Resources.resx 中定义资源 Button1Text 和 Button1Message。

要使用生成的资源类，需要修改 XAML 代码。添加一个 XML 命名空间别名，以引用 .NET 命名空间 Wrox.ProCSharp.Localization.Properties，如下所示。这里，别名设置为 props。在 XAML 元素中，这个类的属性可以用于 x:Static 标记扩展。Button 的 Content 属性设置为 Resources 类的 Button1Text 属性。

```
< Window x:Class="Wrox.ProCSharp.Localization.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:props="clr-namespace:Wrox.ProCSharp.Localization.Properties"
  Title="WPF Sample" Height="300" Width="300" >
  < Grid >
    < Button Name="button1" Margin="30,20,30,20" Click="Button_Click"
      Content="{x:Static props:Resources.Button1Text}" / >
  < /Grid >
```



```
< /Grid >
< /Window >
```

**提示:**

因为添加到 Visual Studio 解决方案中的资源给资源类及其成员使用了 internal 访问修饰符, 且不能修改它, 所以应添加一个定制的构建步骤, 使用 Resgen 实用程序和/publicClass 选项, 把生成的类添加到项目中。否则 WPF 项目就不能编译。

要在后台代码中使用 .NET 资源, 可以直接访问 Button1Message 属性, 与 Windows Forms 应用程序使用的方式相同:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show(Properties.Resources.Button1Message);
}
```

### 21.5.3 用 XAML 本地化

除了使用 .NET 资源本地化 WPF 应用程序之外, 还可以直接使用 XAML 创建本地化内容。本地化过程的步骤如下:

- 从主要内容中创建一个辅助程序集
- 对可以本地化的内容使用资源字典
- 给应本地化的元素添加 x:Uid 属性
- 从程序集中提取本地化内容
- 翻译内容
- 为每种语言创建辅助程序集

在编译 WPF 应用程序时, XAML 代码编译为二进制格式 BAML, 存储在一个程序集中。要把 BAML 代码从主程序集移动到一个独立的辅助程序集中, 可以修改 .csproj 构建文件, 添加如下的 <UICulture> 元素, 作为 <propertyGroup> 元素的一个子元素。这里的文化是 en-US, 它定义了项目的默认文化。用这个构建设置构建项目, 会创建一个子目录 en-US, 并创建一个辅助程序集, 其中包含了默认语言的 BAML 代码。

```
< UICulture > en-US < /UICulture >
```

把 BAML 分离到一个辅助程序集中, 还应应用 NeutralResourcesLanguage 属性, 给辅助程序集提供资源回退位置。如果决定把 BAML 保存在主程序集中(不给 .csproj 文件定义 <UICulture>), UltimateResourceFallbackLocation 就应设置为 MainAssembly。

```
[assembly: NeutralResourcesLanguage("en-US",
    UltimateResourceFallbackLocation.Satellite)]
```

对于需要本地化的后台编码内容, 可以添加一个资源字典。使用 XAML 可以在 <ResourceDictionary> 元素中定义资源, 如下所示。在 Visual Studio 中, 可以添加一个新的资源字典项, 定义文件名, 以创建新的资源字典。在这里的例子中, 资源字典包含一个字符串项。要访问 System 命名空间中的 String 类型, 需要定义一个 XML 命名空间别名。这里把别名 system 设置为程序集 mscorlib 中的命名空间 System。所定义的字符串可以用键 message1 访问。这个

资源字典在 LocalizedStrings.xaml 文件中定义。

```
< ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:system="clr-namespace:System;assembly=mscorlib"
>
    < system:String x:Key="message1" > English Message < /system:String >
< /ResourceDictionary >
```

为了使资源字典可用于应用程序，必须把它添加到资源中。如果资源字典只需在一个窗口或特定的 WPF 元素中使用，就可以把它添加到该窗口或该 WPF 元素的资源集合中。这里，资源字典添加到<Application>元素的文件 App.xaml 中，因此可用于整个应用程序。

```
< Application x:Class="Wrox.ProCSharp.Localization.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="Window1.xaml" >
    < Application.Resources >
        < ResourceDictionary >
            < ResourceDictionary.MergedDictionaries >
                < ResourceDictionary Source="LocalizationStrings.xaml" / >
            < /ResourceDictionary.MergedDictionaries >
        < /ResourceDictionary >
    < /Application.Resources >
< /Application >
```

要在后台代码中使用 XML 资源字典，可以使用 FindResource()方法。因为资源是用<Application>元素定义的，所以使用 Application 类的对象查找该资源。还可以在 WPF 元素中使用 FindResource()方法，因为资源以层次结构的方式来搜索。在这个简单的应用程序中，如果使用 Button 的 FindResource()方法，且没有在 Button 资源中找到它，就可以在 Grid 中搜索资源。如果还没有找到，就查找 Window 资源，最后查找 Application 资源。

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    string message1 =
        (string)Application.Current.FindResource("message1");
    MessageBox.Show(message1);
}
```

在 WPF 元素中，x:Uid 属性用作需要本地化的元素的唯一标识符。不必把这个属性手工应用于 XAML 内容，而可以使用 msbuild 命令和如下选项：

```
msbuild /t:updateuid
```

在项目文件所在的目录下调用这个命令时，项目的 XAML 文件会修改，给每个元素添加 x:Uid 属性和一个唯一标识符。前面的 XAML 现在就应用了新属性：

```
< Window x:Uid="Window_1" x:Class="Wrox.ProCSharp.Localization.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="WPF Sample" Height="300" Width="300" >
    < Grid x:Uid="Grid_1" >
        < Button x:Uid="button1" Name="button1" Margin="30,20,30,20"
            Click="Button_Click" Content="English Button" / >
```

```
< /Grid >
< /Window >
```

如果在添加了 x:Uid 属性后修改 XAML 文件, 就可以用选项 /t:checkuid 验证 x:Uid 属性的正确性。

编译项目, 会创建一个包含 BAML 代码的辅助程序集。在这个辅助程序集中, 可以用 System.Windows.Markup.Localizer 命名空间中的类提取需要本地化的内容。在 Windows SDK 中包含示例程序 LocBaml。这个程序可以用于从 BAML 中提取本地化内容。我们需要把这个可执行的、包含默认内容的辅助程序集和 LocBaml.exe 复制到一个目录下, 启动示例程序, 用本地化内容生成一个.csv 文件。

```
LocBaml /parse WPFandXAMLresources.resources.dll /out: trans.csv
```

可以使用 Microsoft Excel 打开.csv 文件, 翻译其内容。从.csv 文件中提取的内容包含了按钮的内容和资源字典的消息, 如下所示:

```
WPFandXAMLResources.g.en-US.resources:localizationstrings.baml,
system:String_1:System.String.$Content,None,True,True,,English Message
WPFandXAMLResources.g.en-US.resources:window1.baml,
button1:System.Windows.Controls.ContentControl.Content,Button,True,
True,,
English Button
```

这个文件包含如下字段:

- BAML 的名称
- 资源的标识符
- 提供内容类型的资源类别
- 资源是否可翻译(可读)的布尔值
- 资源是否可翻译(可修改)的布尔值
- 本地化注释
- 资源的值

本地化资源后, 就可以为新语言创建一个新目录(例如用于德语的 de 目录)。目录结构与本章前面的辅助程序集相同。使用 LocBaml 工具, 可以用翻译过来的内容创建辅助程序集:

```
LocBaml /generate WPFandXAMLresources.resources.dll
/trans:trans_de.csv /out: ../de/cul:de-DE
```

现在, 设置线程文化和查找辅助程序集的规则与 Windows Forms 应用程序相同。

## 21.6 定制的资源读取器

资源读取器是 .NET Framework 3.5 的一部分, 利用资源读取器, 可以从资源文件和辅助程序集中读取资源。如果要把资源放在另一个存储单元(例如数据库)中, 就可以创建一个定制的资源读取器来读取这些资源。

要使用定制的资源读取器, 还必须创建定制的资源集和定制的资源管理器。但是, 这些都不难, 因为可以从已有的类中派生定制类。

在示例应用程序中，需要创建一个简单的数据库，其中只有一个表，用于存储信息，该表只有一列，存储每种支持的语言。表 21-6 列出了这些列和相应的值。

表 21-6

键	默 认	de	es	fr	it
Welcome	Welcome	Willkommen	Recepcion	Bienvenue	Benvenuto
Good Morning	Good Morning	Guten Morgen	Buonas diaz	Bonjour	Buona Mattina
Good Evening	Good Evening	Guten Abend	Buonas noches	Bonsoir	Buona sera
Thank you	Thank you	Danke	Gracias	Merci	Grazie
Goodbye	Goodbye	Auf Wiedersehen	Adios	Au revoir	Arrivederci

对于定制的资源读取器，用 3 个类创建一个组件库，这些类分别是 DatabaseResourceReader、DatabaseResourceSet 和 DatabaseResourceManager。

### 21.6.1 创建 DatabaseResourceReader 类

类 DatabaseResourceReader 定义了两个字段，即访问数据库所需要的数据源名 connectionString 和读取器返回的语言。这些字段在这个类的构造函数中填充。字段 language 设置为文化名，这个文化名将 与 CultureInfo 对象一起传送给构造函数。

```
public class DatabaseResourceReader : IResourceReader
{
    private string connectionString;
    private string language;

    public DatabaseResourceReader(string connectionString, CultureInfo culture)
    {
        this.connectionString = connectionString;
        this.language = culture.Name;
    }
}
```

资源读取器必须实现接口 IResourceReader，这个接口定义了方法 Close() 和 GetEnumerator()，GetEnumerator() 返回一个 IDictionaryEnumerator，它为资源返回键和值。在 GetEnumerator() 的实现代码中，将创建一个散列表，其中存储了特定语言所有的键和值。接着，使用 System.Data.SqlClient 命名空间中的类 SqlConnection 在 SQL Server 中访问数据库。Connection.CreateCommand() 创建一个 SqlCommand 对象，用于指定 SQL SELECT 语句，以访问数据库中的数据。如果语言设置为 de，Select 语句就是 SELECT [key], [de] FROM Messages。接着，使用 SqlDataReader 对象从数据库中读取所有的值，并把它们放在散列表中。最后，返回散列表的枚举。

**注意：**

有关 ADO.NET 数据访问的更多信息请参阅第 26 章。

```
public System.Collections.IDictionaryEnumerator GetEnumerator()
```

```

{
    Dictionary < string, string > dict = new Dictionary < string, string > ();

    SqlConnection connection = new SqlConnection(connectionString);
    SqlCommand command = connection.CreateCommand();
    if (String.IsNullOrEmpty(language))
        language = "Default";

    command.CommandText = "SELECT [key], [" + language + "] " + "FROM Messages";

    try
    {
        connection.Open();

        SqlDataReader reader = command.ExecuteReader();
        while (reader.Read())
        {
            if (reader.GetValue(1) != System.DBNull.Value)
                dict.Add(reader.GetString(0), reader.GetString(1));
        }
        reader.Close();
    }
    catch //ignore missing column in the database
    {
        if (ex.Number != 207) // ignore missing columns in the database
            throw; // rethrow all other exceptions
    }
    finally
    {
        connection.Close();
    }
    return dict.GetEnumerator();
}

public void Close()
{
}

```

接口 `IResourceReader` 派生自 `IEnumerable` 和 `IDisposable`, 所以也必须实现返回 `IEnumerable` 接口的方法 `GetEnumerator()` 和 `Dispose()`。

```

IEnumerator IEnumerable.GetEnumerator()

```

```

{
    return this.GetEnumerator();
}

```

```

void IDisposable.Dispose()

```

```

{
}

```

## 21.6.2 创建 DatabaseResourceSet 类

类 `DatabaseResourceSet` 可以使用基类 `ResourceSet` 的几乎所有执行代码。我们只需另一个构造函数, 用我们的资源读取器 `DatabaseResourceReader` 初始化基类即可。`ResourceSet` 的构造函数允许传送一个实现 `IResourceReader` 接口的对象, 这个要求 `DatabaseResourceReader` 可以满足。



```

public class DatabaseResourceSet : ResourceSet
{
    internal DatabaseResourceSet(string connectionString, CultureInfo culture)
        : base(new DatabaseResourceReader(connectionString, culture))
    {
    }

    public override Type GetDefaultReader()
    {
        return typeof(DatabaseResourceReader);
    }
}

```

### 21.6.3 创建 DatabaseResourceManager 类

第三个要创建的类是定制的资源管理器 **DatabaseResourceManager**，它派生于类 **ResourceManager**，我们还需要执行一个新的构造函数，并重写方法 **InternalGetResourceSet()**。

在构造函数中，创建一个新的散列表，来存储所有查询到的资源集，并把它们放在由基类定义的字段 **ResourceSets** 中。

```

public class DatabaseResourceManager : ResourceManager
{
    private string connectionString;

    public DatabaseResourceManager(string connectionString)
    {
        this.connectionString = connectionString;
        ResourceSets = new Hashtable();
    }
}

```

可以使用 **ResourceManager** 类中的方法(例如 **GetString()**和 **GetObject()**)来访问资源，并调用方法 **InternalGetResourceSet()**来访问资源集，返回适当的值。

在方法 **InternalGetResourceSet()**的执行代码中，首先检查查询到的文化的资源集是否已在散列表中，如果是，就把它们返回给调用程序。如果资源集不在散列表中，就用查询到的文化创建一个新对象 **DatabaseResourceSet**，把它添加到散列表中，再将它返回给调用程序。

```

protected override ResourceSet InternalGetResourceSet(
    CultureInfo culture, bool createIfNotExists, bool tryParents)
{
    DatabaseResourceSet rs = null;

    If (ResourceSets.Contains(culture.Name))
    {
        rs = ResourceSets[culture.Name] as DatabaseResourceSet;
    }
    else
    {
        rs = new DatabaseResourceSet(connectionString, culture);
        ResourceSets.Add(culture.Name, rs);
    }
    return rs;
}

```

### 21.6.4 DatabaseResourceReader 的客户应用程序

在客户应用程序中使用类 `ResourceManager` 的方式与该类在前面的使用方式没有太大的区别。唯一的区别是要使用定制类 `DatabaseResourceManager` 代替类 `ResourceManager`。下面的代码说明了如何使用自己的资源管理器。

通过把数据库连接字符串传送给构造函数，创建一个新的对象 `DatabaseResource-Manager`。接着像以前一样，调用在基类中实现的 `GetString()` 方法，给它传送键和一个 `CultureInfo` 类型的可选对象，以指定文化。然后，从数据库中获取一个资源值，因为这个资源管理器使用了类 `DatabaseResourceSet` 和 `DatabaseResourceReader`。

```
DatabaseResourceManager rm = new DatabaseResourceManager(
    "server=localhost; database=LocalizationDemo; trusted_connection=true");

string spanishWelcome = rm.GetString("Welcome",
    new CultureInfo("es-ES"));
string italianThankyou = rm.GetString("Thank you",
    new CultureInfo("it"));
string threadDefaultGoodMorning = rm.GetString("Good Morning");
```

## 21.7 创建定制文化

随着时间的推移，.NET Framework 支持的语言越来越多。但并不是所有的语言都得到了.NET 的支持。可以创建定制文化。例如，给一个区域的少数民族创建定制文件，给不同的方言创建子文化。

定制文化和区域可以用 `System.Globalization` 命名空间中的 `CultureAndRegionInfoBuilder` 类创建。这个类在 `sysglobl.dll` 文件的 `sysglobl` 程序集中。

在 `CultureAndRegionInfoBuilder` 类的构造函数中，可以传送文化名。该跟踪函数的第二个参数是 `CultureAndRegionModifiers` 类型的一个枚举。这个枚举有三个值：`Neutral` 表示中立文化，如果应替换已有的 Framework 文化，就使用 `Replacement`，第三个值是 `None`。

在实例化 `CultureAndRegionInfoBuilder` 对象后，就可以设置属性，来配置文化。使用这个类的一些属性，可以定义所有的文化和区域信息，如名称、日历、数字格式、米制信息等。如果文化基于已有的文化和区域，就可以使用 `LoadDataFromCultureInfo()` 和 `LoadDataFromRegionInfo()` 方法设置实例的属性，之后通过设置属性来修改有区别的文化。

调用 `Register()` 方法，给操作系统注册新文化。描述文化的文件位于 `<windows>\Globalization` 目录，其扩展名是 `.nlp`。

```
// Create a Styria culture
CultureAndRegionInfoBuilder styria = new CultureAndRegionInfoBuilder(
    "de-AT-ST", CultureAndRegionModifiers.None);
CultureInfo parent = new CultureInfo("de-AT");
styria.LoadDataFromCultureInfo(parent);
styria.LoadDataFromRegionInfo(new RegionInfo("AT"));
styria.Parent = parent;
styria.RegionNativeName = "Steiermark";
styria.RegionEnglishName = "Styria";
styria.CultureEnglishName = "Styria (Austria)";
```

```
styria.CultureNativeName = "Steirisch";
styria.Register();
```

新建的文化就可以像其他文化那样使用了：

```
CultureInfo ci = new CultureInfo("de-AT-ST");
Thread.CurrentThread.CurrentCulture = ci;
Thread.CurrentThread.CurrentUICulture = ci;
```

文化可以用于格式化和资源。如果再次启动本章前面编写的 Cultures In Action 应用程序，就可以看到定制文化。

## 21.8 小结

本章讨论了.NET 应用程序的国际化和本地化。

在应用程序的国际化方面，我们讨论了命名空间 `System.Globalization`，用于格式化依赖文化的数字和日期。而且说明了在默认情况下，字符串的排序取决于文化。我们使用不变的文化进行独立于文化的排序。使用 `CultureAndRegionInfoBuilder` 类可以创建定制文件。

应用程序的本地化需要使用资源。资源可以放在文件、辅助程序集或定制的存储单元(如数据库)中。本地化所使用的类在 `System.Resources` 命名空间中。要从其他地方读取资源，例如读辅助程序集或资源文件，可以创建一个定制的资源读取器。

我们还学习了如何本地化 Windows Forms、WPF 和 ASP.NET 应用程序。

下一章介绍另一个完全不同的主题——事务处理。事务处理不仅仅可用于数据库，除了介绍数据库事务处理之外，下一章还会讨论基于内存的事务处理资源和事务处理的文件系统。

# 第22章

## 事务处理

事务处理的主要特性是，任务要么全部完成，要么都不完成。在写入一些记录时，要么写入所有记录，要么什么都不写入。如果在写入一个记录时出现了一个失败，在事务处理中已写入的其他数据就会回滚。

事务处理常用于数据库，但利用 `System.Transactions` 命名空间中的类，还可以对不稳定的、基于内存的对象执行事务处理，如一系列对象。对于支持事务处理的一系列对象，如果添加或删除了一个对象时事务处理失败，这个列表的操作会自动撤销。写入一个基于内存的列表与写入数据库一样，也可以在事务处理中完成。

在 Windows Vista 中，文件系统和注册表也支持事务处理。在注册表中写入一个文件，并做出一些修改的操作可以通过事务处理来完成。

本章讨论如下与事务处理相关的内容：

- 事务处理阶段和 ACID 属性概述
- 传统的事务处理
- 可提交的事务处理
- 事务处理的改进
- 依赖的事务处理
- 环境事务处理
- 事务处理的孤立级别
- 定制资源管理器
- Windows Vista 和 Windows Server 2008 的事务处理

### 22.1 概述

什么是事务处理？考虑一下在 Web 站点上订购图书。图书订购进程会把客户要购买的图书从仓库中取出，放在客户的订购框中，再从客户的信用卡收取购买图书的费用。这两个动作要么都成功完成，要么都不完成。如果从仓库中取出图书时出现错误，就不应从信用卡中收取费用。这个工作可以用事务处理来完成。

事务处理最常见的用途是写入或更新数据库中的数据。在消息队列中写入消息，或将数据写入文件或注册表时，也可以使用事务处理。一个事务处理可以包含多个操作。

提示：

`System.Messaging` 命名空间参见第 45 章。

图 22-1 显示了事务处理中的主要元素。事务处理由事务处理管理器来管理和协调。每个影响事务处理结果的资源都由一个资源管理器来管理。事务处理管理器与资源管理器通信，以定义事务处理的结果。

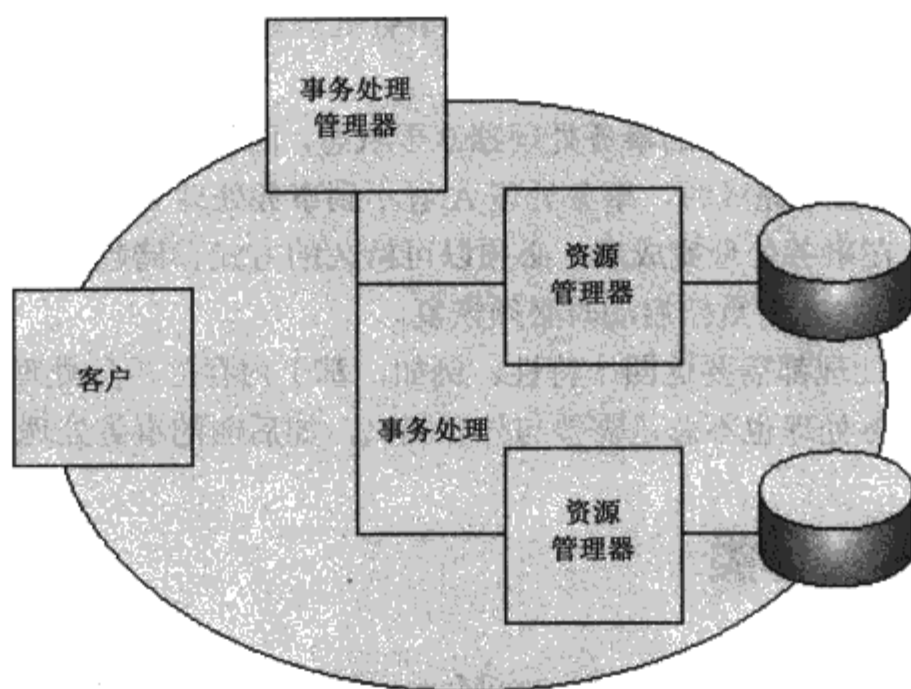


图 22-1

### 22.1.1 事务处理阶段

事务处理分为激活、准备和提交三个阶段。

- 激活阶段：在这个阶段创建事务处理。为资源管理事务处理的资源管理器可以用事务处理登记。
- 准备阶段：在这个阶段，每个资源管理器都可以定义事务处理的结果。事务处理的创建者发出结束事务处理的指令时，就启动这个阶段。事务处理管理器给所有的资源管理器发出一个“准备”消息。如果资源管理器可以成功生成事务处理的结果，就向事务处理管理器发出一个“已准备好”消息。如果资源管理器未能准备好，就可以中止事务处理，发出一个“回滚”消息，强制事务处理管理器执行回滚操作。在发出了“已准备好”消息后，资源管理器必须保证在提交阶段能成功完成工作。为此，稳定的资源管理器必须将准备状态信息写入一个日志，这样，在准备和提交过程中出现停电等故障时，就可以从该状态继续执行。
- 提交阶段：当所有的资源管理器都成功准备好了，就开始这个阶段。即所有资源管理器都发出了“已准备好”消息。接着，事务处理管理器就可以给所有的参与者发送一个“提交”消息，完成工作了。资源管理器现在可以完成事务处理中的工作，返回“已提交”消息。

### 22.1.2 ACID 属性

事务处理有一些特殊的要求，例如事务处理的结果必须处于有效的状态。如果服务器断电了，也需要有有效状态。事务处理的特性可以用术语 ACID 来定义，ACID 是 Atomicity、



Consistency、Isolation 和 Durability 的首字母缩写。

- Atomicity: 表示一个工作单元。在事务处理中, 要么整个工作单元都成功完成, 要么都不完成。
- Consistency: 事务处理开始前的状态和事务处理完成后的状态必须有效。在执行事务处理的过程中, 状态可以有临时值。
- Isolation: 表示并发进行的事务处理独立于状态, 而状态在事务处理过程中可能发生变化。在事务处理未完成时, 事务处理 A 看不到事务处理 B 中的临时状态。
- Durability: 在事务处理完成后, 必须以可持久的方式存储起来。如果关闭电源或服务器崩溃, 这些状态在重新启动时必须恢复。

并不是每个事务处理都需要这四个特性。例如, 基于内存的事务处理(如将一项写入列表)就不需要持久性。事务处理也不总是需要与外界隔离, 如后面的事务处理隔离级别所述。

## 22.2 数据库和实体类

本章的事务处理使用示例数据库 CourseManagement, 它由图 22-2 所示的结构定义。表 Courses 包含课程信息: 课程号和名称; 如课程号为 2124 的课程, 其名称是 C#编程。表 CourseDates 包含指定课程的日期, 它链接到表 Courses 上。表 Students 包含选修某门课程的人。表 CourseAttendees 是 Students 和 CourseDates 之间的链接, 它定义了哪些学生选修了什么课程。

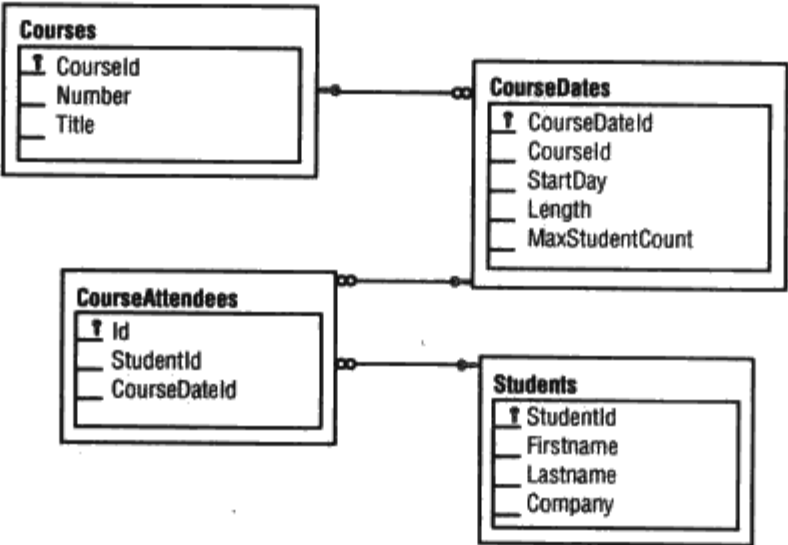


图 22-2

提示:

可以从 Wrox 网站上下载本章的数据库和源代码。

在本章的示例程序中, 使用了一个包含实体和数据访问类的库。这个类 Student 包含的属性定义了一个学生, 如 Firstname、Lastname 和 Company:

```
using System;

namespace Wrox.ProCSharp.Transactions
{
    [Serializable]
    public class Student
    {
    }
```

```

public Student() { }

public Student(string firstname, string lastname)
{
    this.firstname = firstname;
    this.lastname = lastname;
}

public string FirstName { get; set; }
public string LastName { get; set; }
public string Company { get; set; }
public int Id { get; set; }

public override string ToString()
{
    return firstname + " " + lastname;
}
}

```

将学生信息添加到数据库中是在类 `StudentData` 的 `AddStudent()` 方法中完成的。这里创建一个 ADO.NET 连接，来连接 SQL Server 数据库，用 `SqlCommand` 对象定义 SQL 语句，调用 `ExecuteNonQuery()` 来执行该命令：

```

using System;
using System.Collections.Generic;
using System.Data.SqlClient;
using System.Data;
using System.Transactions;

namespace Wrox.ProCSharp.Transactions
{
    public class StudentData
    {
        public void AddStudent(Student s)
        {
            SqlConnection connection = new SqlConnection(
                Properties.Settings.Default.CourseManagementConnectionString);
            connection.Open();
            try
            {
                SqlCommand command = connection.CreateCommand();

                command.CommandText = "INSERT INTO Students " +
                    "(Firstname, Lastname, Company) VALUES " +
                    "(@Firstname, @Lastname, @Company)";
                command.Parameters.AddWithValue("@Firstname", s.Firstname);
                command.Parameters.AddWithValue("@Lastname", s.Lastname);
                command.Parameters.AddWithValue("@Company", s.Company);

                command.ExecuteNonQuery();
            }
            finally
            {
                connection.Close();
            }
        }
    }
}

```

提示:

ADO.NET 详见第 26 章。

## 22.3 传统的事务处理

在发布 System.Transaction 命名空间之前,可以直接用 ADO.NET 创建事务处理,也可以通过组件、属性和 COM+运行库(位于 System.EnterpriseServices 命名空间)进行事务处理。所以,本章将比较新的事务处理模型与传统的事务处理方式,简要论述 ADO.NET 事务处理和用 Enterprise Services 进行事务处理的方式。

### 22.3.1 ADO.NET 事务处理

首先看看传统的 ADO.NET 事务处理。如果没有手工创建事务处理,每个 SQL 语句就都有一个事务处理。如果多个语句应参与到一个事务处理中,就必须手工创建一个事务处理。

下面的代码段演示了如何使用 ADO.NET 事务处理。SqlConnection 类定义了方法 BeginTransaction(),它返回一个 SqlTransaction 类型的对象。这个事务处理对象必须与要参与事务处理的每个命令关联起来。要把命令关联到事务处理上,可将 SqlCommand 类的 Transaction 属性设置为 SqlTransaction 实例。为了使事务处理成功完成,必须调用 SqlTransaction 对象的 Commit()方法。如果有错误,就必须调用 Rollback()方法,撤销每个修改。使用 try/catch 有助于检查错误,在 catch 块中执行回滚。

```
using System;
using System.Data.SqlClient;
using System.Diagnostics;

namespace Wrox.ProCSharp.Transactions
{
    public class CourseData
    {
        public void AddCourse(Course c)
        {
            SqlConnection connection = new SqlConnection(
                Properties.Settings.Default.CourseManagementConnectionString);
            SqlCommand courseCommand = connection.CreateCommand();
            courseCommand.CommandText =
                "INSERT INTO Courses (Number, Title) VALUES (@Number, @Title)";
            connection.Open();
            SqlTransaction tx = connection.BeginTransaction();

            try
            {
                courseCommand.Transaction = tx;

                courseCommand.Parameters.AddWithValue("@Number", c.Number);
                courseCommand.Parameters.AddWithValue("@Title", c.Title);
                courseCommand.ExecuteNonQuery();

                tx.Commit();
            }
            catch (Exception ex)
```

```

    {
        Trace.WriteLine("Error: " + ex.Message);
        tx.Rollback();
    }
    finally
    {
        connection.Close();
    }
}
}
}

```

如果有多个命令要运行在一个事务处理中，每个命令都必须与该事务处理关联起来。事务处理还与一个连接关联起来，所以这些命令也必须关联到同一个连接实例上。ADO.NET 事务处理不支持跨多个连接的事务处理：它总是关联到一个连接上的本地事务处理。

如果使用多个对象创建了一个对象持久模型，例如类 `Course` 和 `CourseData` 应在一个事务处理中持续使用，就很难使用 ADO.NET 事务处理来实现。这需把事务处理传送给参与其中的所有对象。

#### 警告：

ADO.NET 事务处理不是分布式事务处理。在 ADO.NET 事务处理中，很难让多个对象在同一个事务处理中工作。

### 22.3.2 System.EnterpriseServices

利用 Enterprise Services，可以免费获得许多服务，其中之一是自动事务处理。使用 `System.EnterpriseServices` 中的事务处理的优点是，不需要明确进行事务处理，运行库会自动创建事务处理，只需给有事务处理要求的类添加 `[Transaction]` 特性即可。`[AutoComplete]` 特性把方法标记为自动设置事务处理的状态位：如果该方法成功，就设置成功位，因此提交事务处理。如果发生异常，就中止事务处理。

```

using System;
using System.Data.SqlClient;
using System.EnterpriseServices;
using System.Diagnostics;

namespace Wrox.ProCSharp.Transactions
{
    [Transaction(TransactionOption.Required)]
    public class CourseData : ServicedComponent
    {
        [AutoComplete]
        public void AddCourse(Course c)
        {
            SqlConnection connection = new SqlConnection(
                Properties.Settings.Default.CourseManagementConnectionString);
            SqlCommand courseCommand = connection.CreateCommand();
            courseCommand.CommandText =
                "INSERT INTO Courses (Number, Title) VALUES (@Number, @Title)";
            connection.Open();
            try
            {
                courseCommand.Parameters.AddWithValue("@Number", c.Number);
            }
        }
    }
}

```



```
        courseCommand.Parameters.AddWithValue("@Title", c.Title);
        courseCommand.ExecuteNonQuery();
    }
    finally
    {
        connection.Close();
    }
}
}
```

用 System.EnterpriseServices 创建事务处理的一大优点是，多个对象能轻松地运行在同一个事务处理中，事务处理还可以自动登记。缺点是它需要 COM+主机模型，使用这个技术的类必须派生自基类 ServicedComponent。

提示：  
Enterprise Services 和使用 COM+事务处理服务的内容详见第 44 章。

## 22.4 System.Transactions

自.NET 2.0 以来增加了 System.Transactions 命名空间，为.NET 应用程序带来了一个新的事务处理编程模型。图 22-3 是一个 Visual Studio 类图，其中包含 System.Transaction 命名空间中的事务处理类以及它们的关系：Transaction、CommittableTransaction、DependentTransaction 和 SubordinateTransaction。

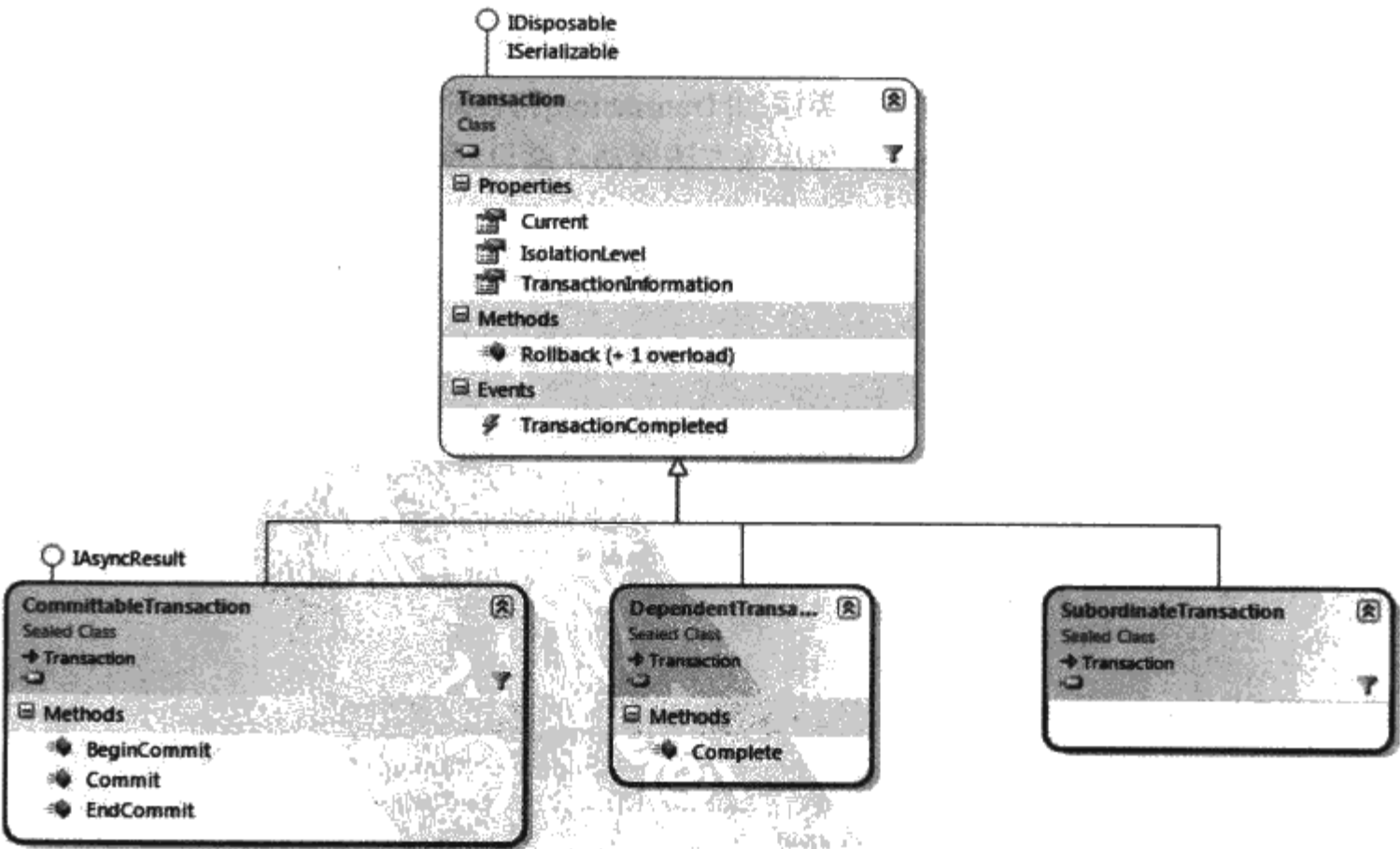


图 22-3

Transaction 是所有事务处理类的基类，定义了所有事务处理类都可以使用的属性、方法和



事件。**CommittableTransaction** 是唯一一个支持提交的事务处理类。这个类有一个 **Commit()**方法，其他事务处理类都只能执行回滚。**DependentTransaction** 类用于依赖于其他事务处理的事务处理。依赖的事务处理可以依赖从可提交的事务处理中创建的事务处理。如果事务处理成功，就把依赖的事务处理添加到可提交的事务处理的结果中。类 **SubordinateTransaction** 和分布式事务处理协调器(DTC)一起使用。这个类表示非根事务处理，但可以由 DTC 管理。

表 22-1 列出了 **Transaction** 类的属性和方法。

表 22-1

Transaction 类的成员	说 明
Current	Current 是一个静态属性，不需要有实例。Transaction.Current 返回一个环境事务处理(如果存在)。环境事务处理在后面讨论
IsolationLevel	IsolationLevel 属性返回 IsolationLevel 类型的对象。IsolationLevel 是一个枚举，它定义了其他事务处理必须有什么访问权限才能访问事务处理的临时结果。它会影响 ACID 中的 I；并不是所有的事务处理都是隔离的
TransactionInformation	TransactionInformation 属性返回一个 TransactionInformation 对象。该对象提供了事务处理的当前状态信息、事务处理的创建时间和事务处理标识符
EnlistVolatile() EnlistDurable() EnlistPromotableSinglePhase()	使用 登记方法 EnlistVolatile()、EnlistDurable() 和 EnlistPromotableSinglePhase()，可以登记参与事务处理的定制资源管理器
Rollback()	使用 Rollback()方法，可以中止一个事务处理，撤销所有的改变，把所有的结果设置为事务处理之前的状态
DependentClone()	使用 DependentClone()方法，可以创建一个依赖于当前事务处理的事务处理
TransactionCompleted	TransactionCompleted 是一个事件，在事务处理完成时引发——事务处理可能成功，也可能失败。在 TransactionCompletedEvent Handler 类型的事件处理程序中，可以访问 Transaction 对象，读取其状态

为了演示 **System.Transaction** 命名空间的特性，下面在一个独立程序集中的 **Utilities** 类提供了一些静态方法。**AbortTx()** 方法根据用户的输入返回 **true** 或 **false**。方法 **DisplayTransactionInformation()**将一个 **TransactionInformation** 对象作为参数，显示事务处理中的所有信息：创建时间、状态、本地和分布式标识符。

```
public static class Utilities
{
    public static bool AbortTx()
    {
        Console.WriteLine("Abort the Transaction (y/n)?");
        return Console.ReadLine() == "y";
    }

    public static void DisplayTransactionInformation(
        TransactionInformation ti)
    {
        if (ti != null)
```

```

    {
        Console.WriteLine("Creation Time: {0:T}", ti.CreationTime);
        Console.WriteLine("Status: {0}", ti.Status);
        Console.WriteLine("Local ID: {0}", ti.LocalIdentifier);
        Console.WriteLine("Distributed ID: {0}", ti.DistributedIdentifier);
        Console.WriteLine();
    }
}

```

### 22.4.1 可提交的事务处理

Transaction 类不能以编程方式提交，它没有提交事务处理的方法。基类 Transaction 只支持事务处理的中止。唯一支持提交的事务处理类是 CommittableTransaction。

在 ADO.NET 中，事务处理可以用连接来登记。为此，在 StudentData 类中添加 AddStudent() 方法，它将一个 System.Transactions.Transaction 对象作为第二个参数。调用 SqlConnection 类的 EnlistTransaction 方法，用连接来登记对象 tx。这样，ADO.NET 连接就关联到事务处理上。

```

public void AddStudent(Student s, Transaction tx)
{
    SqlConnection connection = new SqlConnection(
        Properties.Settings.Default.CourseManagementConnectionString);
    connection.Open();
    try
    {
        if (tx != null)
            connection.EnlistTransaction(tx);
        SqlCommand command = connection.CreateCommand();

        command.CommandText = "INSERT INTO Students (Firstname, Lastname, Company)
            VALUES (@Firstname, @Lastname, @Company)";
        command.Parameters.AddWithValue("@Firstname", s.Firstname);
        command.Parameters.AddWithValue("@Lastname", s.Lastname);
        command.Parameters.AddWithValue("@Company", s.Company);

        command.ExecuteNonQuery();
    }
    finally
    {
        connection.Close();
    }
}

```

在控制台应用程序 CommittableTransaction 的 Main()方法中，先创建一个 CommittableTransaction 类型的事务处理。之后，将信息显示在控制台上。接着，创建一个 Student 对象，在 AddStudent()方法中把这个对象写入数据库。如果在事务处理的外部验证数据库中的记录，就看不到刚才添加的学生，除非事务处理已完成。如果事务处理失败，就执行回滚，不把学生写入数据库。

在调用 AddStudent()方法之后，就调用帮助方法 Utilities.AbortTx()，确定事务处理是否要中止。如果用户要中止，就抛出一个 ApplicationException 类型的异常，在 catch 块中，调用 Transaction 类的 Rollback()方法，回滚事务处理。不把记录写入数据库。如果用户不中止，Commit()方法就提交事务处理，提交事务处理的完成状态。

```

static void Main()
{
    CommittableTransaction tx = new CommittableTransaction();
    Utilities.DisplayTransactionInformation("TX created",
        tx.TransactionInformation);

    try
    {
        Student s1 = new Student();
        s1.Firstname = "Neno";
        s1.Lastname = "Loye";
        s1.Company = "thinkecture";
        StudentData db = new StudentData();
        db.AddStudent(s1, tx);

        if (Utilities.AbortTx())
        {
            throw new ApplicationException("transaction abort");
        }

        tx.Commit();
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
        Console.WriteLine();
        tx.Rollback();
    }

    Utilities.DisplayTransactionInformation(tx.TransactionInformation);
}

```

在这里的应用程序输出中，事务处理是激活的，有一个本地标识符。在第一次运行应用程序时，选择中止事务处理。在完成事务处理后，会看到中止状态。

```

TX Created
Creation Time: 7:30:49 PM
Status: Active
Local ID: bdcflcdc-a67e-4ccc-9a5c-cbdfe0fe9177:1
Distributed ID: 00000000-0000-0000-0000-000000000000
Abort the Transaction (y/n)? y
Transaction abort

TX finished
Creation Time: 7:30:49 PM
Status: Aborted
Local ID: bdcflcdc-a67e-4ccc-9a5c-cbdfe0fe9177:1
Distributed ID: 00000000-0000-0000-0000-000000000000
Press any key to continue ...

```

应用程序的第二次运行没有中止事务处理。事务处理的状态是已提交，数据写入数据库。

```

TX Created
Creation Time: 7:33:04 PM
Status: Active
Local ID: 708bda71-fa24-46a9-86b4-18b83120f6af:1
Distributed ID: 00000000-0000-0000-0000-000000000000

Abort the Transaction (y/n)? n

TX finished

```

```

Creation Time: 7:33:04 PM
Status: Committed
Local ID: 708bda71-fa24-46a9-86b4-18b83120f6af:1
Distributed ID: 00000000-0000-0000-0000-000000000000

```

Press any key to continue ...

## 22.4.2 事务处理的升级

`System.Transactions` 支持可升级的事务处理。根据参与事务处理的资源，会创建本地的事务处理或分布式事务处理。SQL Server 2005 和 2008 支持可升级的事务处理，但目前我们只看到本地事务处理。在前面的例子中，分布式 ID 总是设置为 0，只赋予了本地 ID。对于不支持可升级的事务处理的资源，会创建分布式事务处理。如果多个资源添加到事务处理中，事务处理就可以先设置为本地事务处理，再根据需要升级为分布式事务处理。当多个 SQL Server 数据库连接添加到事务处理中时，就会进行这种升级。事务处理开始时是一个本地事务处理，之后升级为分布式事务处理。

现在修改控制台应用程序，使用同一个事务处理对象 `tx` 添加第二个学生。每个 `AddStudent()` 方法打开一个新连接时，就会在添加第二个学生后，把两个连接关联到事务处理上。

```

static void Main()
{
    CommittableTransaction tx = new CommittableTransaction();
    Utilities.DisplayTransactionInformation("TX created",
        tx.TransactionInformation);
    try
    {
        Student s1 = new Student();
        s1.Firstname = "Neno";
        s1.Lastname = "Loye";
        s1.Company = "thinktecture";
        StudentData db = new StudentData();
        db.AddStudent(s1, tx);

        Student s2 = new Student();
        s2.Firstname = "Dominick";
        s2.Lastname = "Baier";
        s2.Company = "thinktecture";
        db.AddStudent(s2, tx);

        Utilities.DisplayTransactionInformation("2nd connection enlisted",
            tx.TransactionInformation);

        if (Utilities.AbortTx())
        {
            throw new ApplicationException("transaction abort");
        }

        tx.Commit();
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
        Console.WriteLine();
        tx.Rollback();
    }
}

```



```

Utilities.DisplayTransactionInformation("TX finished",
    tx.TransactionInformation);
}

```

现在运行应用程序，就会看到，对于添加的第一个学生，其分布式标识符是 0，但对于添加的第二个学生，升级了事务处理，所以分布式标识符与该事务处理关联起来。

```

TX created
Creation Time: 7:56:24 PM
Status: Active
Local ID: 0d2f5ada-32aa-40eb-b9d7-cc6aa9a2a554:1
Distributed ID: 00000000-0000-0000-0000-000000000000

2nd connection enlisted
Creation Time: 7:56:24 PM
Status: Active
Local ID: 0d2f5ada-32aa-40eb-b9d7-cc6aa9a2a554:1
Distributed ID: 70762617-2ee8-4d23-aa87-6ac8c1418bdfd

Abort the Transaction (y/n)?

```

事务处理的升级需要启动分布式事务处理协调器(DTC)。如果在系统中升级事务处理时失败，应验证一下 DTC 服务是否启动。启动 Component Services MMC 插件，就可以看到运行在系统上的所有 DTC 事务处理的状态。在树型视图中选择 Transaction List，就可以看到所有激活的事务处理。在图 22-4 中，有一个激活的事务处理，其分布式标识符与前面控制台的输出相同。如果验证系统上的输出，应确保该事务处理设置了超时时间，在超过指定时间后，就会中止。在超过该时间后，在事务处理列表中就看不到该事务处理了。还可以用这个工具验证事务处理的统计数据。Transaction Statistics 显示了提交和中止的事务处理数。

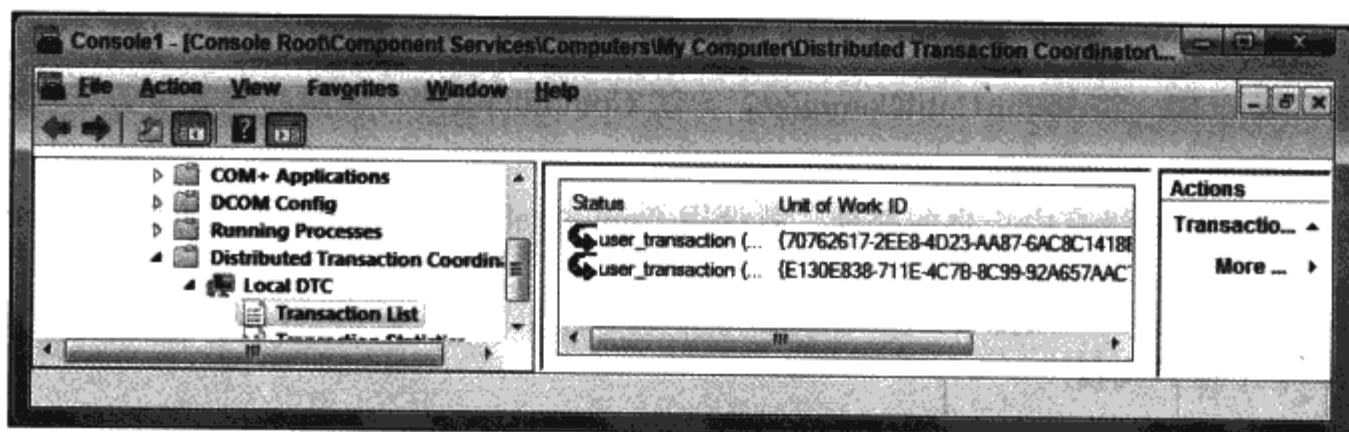


图 22-4

#### 提示:

要启动 Component Services MMC 插件，可以启动 Microsoft Management Console(mmc.exe)应用程序，选择菜单 File | Add/Remove Snap-In，再从插件列表中选择 Component Services。

### 22.4.3 依赖的事务处理

使用依赖的事务处理，可以影响多个线程中的某个事务处理。依赖事务处理会依靠另一个事务处理，影响事务处理的结果。

示例应用程序 DependentTransactions 为一个新线程创建了一个依赖的事务处理。TxThread()



是新线程的一个方法，它将一个 `DependentTransaction` 对象传送为参数。依赖事务处理的信息用帮助方法 `DisplayTransactionInformation()` 来显示。在线程退出之前，调用依赖事务处理的 `Complete()` 方法来定义事务处理的结果。依赖事务处理可以调用 `Complete()` 或 `Rollback()` 方法来定义事务处理的结果。`Complete()` 方法设置成功位。如果根事务处理完成了，且所有依赖事务处理都把成功位设置为 `true`，就提交事务处理。如果某个依赖事务处理调用 `Rollback()` 方法来设置中止位，事务处理就会中止。

```
static void TxThread(object obj)
{
    DependentTransaction tx = obj as DependentTransaction;
    Utilities.DisplayTransactionInformation("Dependent Transaction",
        tx.TransactionInformation);

    Thread.Sleep(3000);

    tx.Complete();

    Utilities.DisplayTransactionInformation("Dependent TX Complete",
        tx.TransactionInformation);
}
```

在 `Main()` 方法中，先实例化类 `CommittableTransaction`，创建一个根事务处理，显示事务处理的信息。接着 `tx.DependentClone()` 创建一个依赖事务处理。这个依赖事务处理传送给方法 `TxThread()`，它定义为新线程的入口点。

方法 `DependentClone()` 需要一个 `DependentCloneOption` 类型的变元。`DependentCloneOption` 是一个枚举，其值是 `BlockCommitUntilComplete` 和 `RollbackIfNotComplete`。如果根事务处理在依赖事务处理之前完成，这个选项就很重要。把该选项设置为 `RollbackIfNotComplete`，如果依赖事务处理没有在根事务处理的 `Commit()` 方法之前调用 `Complete()` 方法，事务处理就中止。把该选项设置为 `BlockCommitUntilComplete`，方法 `Commit()` 就等待由所有依赖事务处理定义的结果。

接着，如果用户没有中止事务处理，就调用 `CommittableTransaction` 类的方法 `Commit()`。

提示：

第 19 章介绍了线程。

```
static void Main()
{
    CommittableTransaction tx = new CommittableTransaction();
    Utilities.DisplayTransactionInformation("Root TX created",
        tx.TransactionInformation);
    try
    {
        new Thread(TxThread).Start(
            tx.DependentClone(DependentCloneOption.BlockCommitUntilComplete));

        if (Utilities.AbortTx())
        {
            throw new ApplicationException("transaction abort");
        }

        tx.Commit();
    }
}
```

```

    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
        tx.Rollback();
    }

    Utilities.DisplayTransactionInformation("TX finished",
        tx.TransactionInformation);
}

```

在应用程序的输出中，可以看到根事务处理及其标识符。由于使用了选项 `DependentCloneOption. BlockCommitUntilComplete`，根事务处理在方法 `Commit()` 中等待，直到定义了依赖事务处理的结果。完成了依赖事务处理后，就提交事务处理。

```

Root TX created
Creation Time: 8:35:25 PM
Status: Active
Local ID: 50126e07-cd28-4e0f-a21f-a81a8e14a1a8:1
Distributed ID: 00000000-0000-0000-0000-000000000000

Abort the Transaction (y/n)? n

Dependent Transaction
Creation Time: 8:35:25 PM
Status: Active
Local ID: 50126e07-cd28-4e0f-a21f-a81a8e14a1a8:1
Distributed ID: 00000000-0000-0000-0000-000000000000

Dependent TX Complete
Root TX finished
Creation Time: 8:35:25 PM
Status: Committed
Local ID: 50126e07-cd28-4e0f-a21f-a81a8e14a1a8:1
Distributed ID: 00000000-0000-0000-0000-000000000000

Creation Time: 8:35:25 PM
Status: Committed
Local ID: 50126e07-cd28-4e0f-a21f-a81a8e14a1a8:1
Distributed ID: 00000000-0000-0000-0000-000000000000

Press any key to continue ...

```

#### 22.4.4 环境事务处理

`System.Transactions` 的一大优点是环境事务处理特性。使用环境事务处理，就不需要手工登记与事务处理的连接；在支持环境事务处理的资源中，这是自动实现的。

环境事务处理与当前的线程关联起来。可以使用静态属性 `Transaction.Current` 获取和设置环境事务处理。支持环境事务处理的 API 会检查这个属性，以获得环境事务处理，用该事务处理登记。ADO.NET 连接支持环境事务处理。

可以创建一个 `CommittableTransaction` 对象，把它赋予属性 `Transaction.Current`，以初始化环境事务处理。创建环境事务处理的另一种方式是使用 `TransactionScope` 类。`TransactionScope` 的构造函数会创建一个环境事务处理。由于 `TransactionScope` 实现了接口 `IDisposable`，因此可以方便地将 `using` 语句用于事务处理作用域。

`TransactionScope` 的成员如表 22-2 所示。

表 22-2

TransactionScope 的成员	说 明
构造函数	使用 TransactionScope 的构造函数，可以定义事务处理的要求。也可以传送已有的事务处理，定义事务处理的超时值
Complete()	调用 Complete()方法，可以设置事务处理作用域的成功位
Dispose()	Dispose()方法完成该作用域，如果该作用域与根事务处理相关，就提交或中止事务处理。如果所有的依赖事务处理都设置了成功位，就提交 Dispose()方法，否则就回滚事务处理

TransactionScope 类实现了接口 IDisposable,所以可以用 using 语句定义事务处理的作用域。默认构造函数创建了一个新的事务处理,之后创建 TransactionScope 实例,用 Transaction.Current 属性的 get 存取器访问事务处理,在控制台上显示事务处理的信息。

要获得事务处理何时完成的信息，就应为环境事务处理的 TransactionCompleted 事件设置 OnTransactionCompleted()方法。

然后，调用 StudentData.AddStudent()方法，创建一个新的 Student 对象，并写入数据库。对于环境事务处理，不再需要给这个方法传送 Transaction 对象，因为 SqlConnection 类支持环境事务处理，会自动把它登记到连接上。接着 TransactionScope 类的 Complete()方法设置成功位，在 using 语句的最后删除 TransactionScope 对象，完成提交。如果没有调用 Complete()方法，Dispose()就中止事务处理。

提示：

如果 ADO.Net 连接不应使用环境事务处理来登记，就可以用连接字符串设置值 Enlist=false。

```
static void Main()
{
    using (TransactionScope scope = new TransactionScope())
    {
        Transaction.Current.TransactionCompleted += OnTransactionCompleted;

        Utilities.DisplayTransactionInformation("Ambient TX created",
            Transaction.Current.TransactionInformation);

        Student s1 = new Student();
        s1.Firstname = "Ingo";
        s1.Lastname = "Rammer";
        s1.Company = "thinktexture";
        StudentData db = new StudentData();
        db.AddStudent(s1);

        if (!Utilities.AbortTx())
            scope.Complete();
        else
            Console.WriteLine("transaction will be aborted");
    }
} // Dispose
static void OnTransactionCompleted(object sender, TransactionEventArgs e)
{
    Utilities.DisplayTransactionInformation("TX completed",
```

```
        e.Transaction.TransactionInformation);
    }
```

运行应用程序，可以看到在创建 TransactionScope 类的一个实例后，有一个激活的环境事务处理。应用程序的最终结果是 TransactionCompleted 事件处理程序的输出，即显示事务处理结束状态。

```
Ambient TX created
Creation Time: 9:55:40 PM
Status: Active
Local ID: a06df6fb-7266-435e-b90e-f024f1d6966e:1
Distributed ID: 00000000-0000-0000-0000-000000000000

Abort the Transaction (y/n)? n

TX completed
Creation Time: 9:55:40 PM
Status: Committed
Local ID: a06df6fb-7266-435e-b90e-f024f1d6966e:1
Distributed ID: 00000000-0000-0000-0000-000000000000

Press any key to continue ...
```

1. 嵌套的作用域和环境事务处理

使用 TransactionScope 类，还可以嵌套作用域。嵌套的作用域可以位于原来的作用域中，或位于从作用域中调用的方法里。嵌套的作用域可以使用与外层作用域相同的事务处理，抑制事务处理，或创建独立于外层作用域的新事务处理。作用域的要求通过 TransactionScopeOption 枚举来定义，该枚举传送给 TransactionScope 类的构造函数。

TransactionScopeOption 枚举的值及其作用如表 22-3 所示。

表 22-3

TransactionScope- Option 的成员	说 明
Required	Required 指定，作用域需要一个事务处理。如果外层的作用域已经包含了一个环境事务处理，内层的作用域就使用已有的事务处理。如果环境事务处理不存在，就创建一个新的事务处理  如果两个作用域共享同一个事务处理，这两个作用域都会影响事务处理的结果。只有所有作用域都设置了成功位，事务处理才能提交。如果一个作用域没有在根作用域被删除之前调用了 Complete()方法，事务处理就中止
RequiresNew	RequiresNew 总是创建一个新的事务处理。如果外层的作用域已定义了一个事务处理，则内层作用域中的事务处理就是完全独立的。两个事务处理都可以独立地提交或中止
Suppress	使用 Suppress，无论外层的作用域是否包含一个事务处理，作用域都不会包含环境事务处理

下面的例子定义了两个作用域，内层作用域使用 TransactionScopeOption.RequiresNew 选项，配置为需要一个新的事务处理：

```
using (TransactionScope scope = new TransactionScope())
{
```



```

Transaction.Current.TransactionCompleted += OnTransactionCompleted;

Utilities.DisplayTransactionInformation("Ambient TX created",
    Transaction.Current.TransactionInformation);
using (TransactionScope scope2 =
    new TransactionScope(TransactionScopeOption.RequiresNew))
{
    Transaction.Current.TransactionCompleted += OnTransactionCompleted;

    Utilities.DisplayTransactionInformation(
        "Inner Transaction Scope",
        Transaction.Current.TransactionInformation);

    scope2.Complete();
}
scope.Complete();
}

```

运行应用程序，会看到两个作用域有不同的事务处理标识符，但使用相同的线程。让一个线程因为作用域不同而有不同的环境事务处理，事务处理标识符在 GUID 后面的最后一个数字就是不同的。

```

Ambient TX created
Creation Time: 11:01:09 PM
Status: Active
Local ID: 54ac1276-5c2d-4159-84ab-36b0217c9c84:1
Distributed ID: 00000000-0000-0000-0000-000000000000

Inner Transaction Scope
Creation Time: 11:01:09 PM
Status: Active
Local ID: 54ac1276-5c2d-4159-84ab-36b0217c9c84:2
Distributed ID: 00000000-0000-0000-0000-000000000000

TX completed
Creation Time: 11:01:09 PM
Status: Committed
Local ID: 54ac1276-5c2d-4159-84ab-36b0217c9c84:2
Distributed ID: 00000000-0000-0000-0000-000000000000

TX completed
Creation Time: 11:01:09 PM
Status: Committed
Local ID: 54ac1276-5c2d-4159-84ab-36b0217c9c84:1
Distributed ID: 00000000-0000-0000-0000-000000000000

```

如果把内层作用域的设置改为 `TransactionScopeOption.Required`，两个作用域就会使用同一个事务处理，且都影响事务处理的结果。

## 2. 多线程和环境事务处理

如果多个线程使用同一个环境事务处理，就需要多做一些工作。一个环境事务处理绑定到一个线程上，所以如果创建了一个新线程，它就不会有起始线程中的环境事务处理。

这个行为在下一个例子中演示。在 `Main()` 方法中，创建了一个 `TransactionScope`。在这个事务处理的作用域中，启动一个新线程。新线程的主要方法 `ThreadMethod()` 创建了一个新的事务处理作用域。在创建该作用域的过程中，没有传送任何参数，因此使用默认选项



**TransactionScopeOption.Required**。如果存在一个环境事务处理，就使用已有的事务处理。如果没有环境事务处理，就创建一个新的事务处理。

```
using System;
using System.Threading;
using System.Transactions;

namespace Wrox.ProCSharp.Transactions
{
    class Program
    {
        static void Main()
        {
            try
            {
                using (TransactionScope scope = new TransactionScope())
                {
                    Transaction.Current.TransactionCompleted +=
                        TransactionCompleted;

                    Utilities.DisplayTransactionInformation("Main thread TX",
                        Transaction.Current.TransactionInformation);

                    new Thread(ThreadMethod).Start(null);

                    scope.Complete();
                }
            }
            catch (TransactionAbortedException ex)
            {
                Console.WriteLine("Main - Transaction was aborted, {0}",
                    ex.Message);
            }
        }

        static void TransactionCompleted(object sender,
            TransactionEventArgs e)
        {
            Utilities.DisplayTransactionInformation("TX completed",
                e.Transaction.TransactionInformation);
        }

        static void ThreadMethod(object dependentTx)
        {
            try
            {
                using (TransactionScope scope = new TransactionScope())
                {
                    Transaction.Current.TransactionCompleted +=
                        Current_TransactionCompleted;

                    Utilities.DisplayTransactionInformation("Thread TX",
                        Transaction.Current.TransactionInformation);
                    scope.Complete();
                }
            }
            catch (TransactionAbortedException ex)
            {
                Console.WriteLine("ThreadMethod - Transaction was aborted, {0}",
                    ex.Message);
            }
        }
    }
}
```

```
    }  
  }  
}
```

启动应用程序，就会看到两个线程中的事务处理是完全独立的。新线程中的事务处理有一个不同的事务处理 ID。事务处理 ID 的区别是 GUID 后面的最后一个数字不同，这与嵌套的作用域相同，嵌套的作用域也需要一个新的事务处理。

```
Main thread TX  
Creation Time: 21:41:25  
Status: Active  
Local ID: fle736ae-84ab-4540-b71e-3de272ffc476:1  
Distributed ID: 00000000-0000-0000-0000-000000000000  
  
TX completed  
Creation Time: 21:41:25  
Status: Committed  
Local ID: fle736ae-84ab-4540-b71e-3de272ffc476:1  
Distributed ID: 00000000-0000-0000-0000-000000000000  
  
Thread TX  
Creation Time: 21:41:25  
Status: Active  
Local ID: fle736ae-84ab-4540-b71e-3de272ffc476:2  
Distributed ID: 00000000-0000-0000-0000-000000000000  
  
TX completed  
Creation Time: 21:41:25  
Status: Committed  
Local ID: fle736ae-84ab-4540-b71e-3de272ffc476:2  
Distributed ID: 00000000-0000-0000-0000-000000000000
```

要在另一个线程中使用同一个环境事务处理，需要使用依赖事务处理。现在修改示例，给新线程传送一个依赖事务处理。依赖事务处理是调用环境事务处理上的 `DependentClone()` 方法，从环境事务处理中创建的。在这个方法中，设置了 `DependentCloneOption.BlockCommitUntilComplete`，所以调用线程会等到新线程完成后，才提交事务处理。

```
class Program  
{  
    static void Main()  
    {  
        try  
        {  
            using (TransactionScope scope = new TransactionScope())  
            {  
                Transaction.Current.TransactionCompleted +=  
                    TransactionCompleted;  
                Utilities.DisplayTransactionInformation("Main thread TX",  
                    Transaction.Current.TransactionInformation);  
                new Thread(ThreadMethod).Start(  
                    Transaction.Current.DependentClone(  
                        DependentCloneOption.BlockCommitUntilComplete));  
                scope.Complete();  
            }  
        }  
        catch (TransactionAbortedException ex)
```

```

    {
        Console.WriteLine("Main - Transaction was aborted, {0}",
            ex.Message);
    }
}

```

在线程的方法中，使用 `Transaction.Current` 属性的 `set` 访问器把所传送的依赖事务处理赋予环境事务处理。现在事务处理作用域通过依赖事务处理来使用同一个事务处理。使用完依赖事务处理后，需要调用 `DependentTransaction` 对象的 `Complete()` 方法。

```

static void ThreadMethod(object dependentTx)
{
    DependentTransaction dTx = dependentTx as DependentTransaction;

    try
    {
        Transaction.Current = dTx;
        using (TransactionScope scope = new TransactionScope())
        {
            Transaction.Current.TransactionCompleted +=
                Current_TransactionCompleted;
            Utilities.DisplayTransactionInformation("Thread TX",
                Transaction.Current.TransactionInformation);
            scope.Complete();
        }
    }
    catch (TransactionAbortedException ex)
    {
        Console.WriteLine("ThreadMethod - Transaction was aborted, {0}",
            ex.Message);
    }
    finally
    {
        if (dTx != null)
        {
            dTx.Complete();
        }
    }
}

```

现在运行应用程序，主线程和新建的线程都使用同一个事务处理，并影响该事务处理。线程列出的事务处理有相同的标识符。如果一个线程没有调用 `Complete()` 方法设置成功位，就终止整个事务处理。

```

Main thread TX
Creation Time: 23:00:57
Status: Active
Local ID: 2fb1b54d-61f5-4d4e-a55e-f4a9e04778be:1
Distributed ID: 00000000-0000-0000-0000-000000000000

```

```

Thread TX
Creation Time: 23:00:57
Status: Active
Local ID: 2fb1b54d-61f5-4d4e-a55e-f4a9e04778be:1
Distributed ID: 00000000-0000-0000-0000-000000000000

```

```

TX completed
Creation Time: 23:00:57

```

```
Status: Committed
Local ID: 2fb1b54d-61f5-4d4e-a55e-f4a9e04778be:1
Distributed ID: 00000000-0000-0000-0000-000000000000

TX completed
Creation Time: 23:00:57
Status: Committed
Local ID: 2fb1b54d-61f5-4d4e-a55e-f4a9e04778be:1
Distributed ID: 00000000-0000-0000-0000-000000000000
```

22.5 隔离级别

本章的开头介绍了用于描述事务处理的 ACID 特性。ACID 中的字母 I(Isolation, 隔离)并不总是需要。出于性能原因,可以降低隔离要求,但必须了解改变隔离级别带来的问题。

如果不完全隔离事务处理外部的作用域,就可能出问题,这些问题有三类:

- 脏读取: 在脏读取操作中,一个事务处理可以读取在另一个事务处理中改变的记录。因为在另一个事务处理中改变的记录可能回滚为最初的状态,所以从另一个事务处理中读取这个临时状态就称为“脏读取”——数据并没有提交。通过锁定要改变的记录,就可以避免这个问题。
- 不能重复的读取操作: 当数据在事务处理中读取,而该事务处理运行的同时,另一个事务处理修改了相同的记录,此时,就会出现不能重复的读取操作。如果记录在事务处理中读取多次,结果就会不同——不能重复。锁定读取的记录,即可避免这个问题。
- 幻影读取: 当读取一个范围内的记录,例如使用 WHERE 子句读取时,就会出现幻影读取问题。在一个事务处理中读取这些记录时,另一个事务处理可以添加一个属于该范围的新记录。用相同的 WHERE 子句再次读取这些记录,会返回数量不同的记录。在更新一个范围的记录时,幻影读取是一个特殊的问题。例如,UPDATE Addresses SET Zip=4711 WHERE (Zip=2315)会把所有记录的邮政编码 2315 更新为 4711。如果在更新过程中,另一个用户添加了一个邮政编码为 2315 的新记录,完成更新后,数据库将仍包含邮政编码为 2315 的记录。这个问题可以通过范围锁定来避免。

在定义隔离要求时,可以设置隔离级别。隔离级别用 IsolationLevel 枚举定义,在创建事务处理时,会配置该枚举(使用 CommittableTransaction 类的构造函数或 TransactionScope 类的构造函数)。IsolationLevel 枚举定义了锁定操作。表 22-4 列出了 IsolationLevel 枚举的值。

表 22-4

隔离级别	说 明
ReadUncommitted	使用 ReadUncommitted, 事务处理不会相互隔离。使用这个级别,不等待其他事务处理释放锁定的记录。这样,就可以从其他事务处理中读取未提交的数据——脏读取。这个级别通常仅用于读取临时修改不大重要的记录,例如报表
ReadCommitted	ReadCommitted 等待其他事务处理释放对记录的写入锁定。这样,就不会出现脏读取操作。这个级别为读取当前的记录设置读取锁定,为要写入的记录设置写入锁定,直到事务处理完成为止。对于要读取的记录,在移动到下一个记录上时,每个记录都是未锁定的,所以可能出现无法重复的读取操作

(续表)

隔离级别	说 明
RepeatableRead	RepeatableRead 为读取的记录设置锁定，直到事务处理完成为止。这样，就避免了不可重复的读取问题。但幻影读取仍可能发生
Serializable	Serializable 设置范围锁定。在运行事务处理时，不可能添加与所读取的记录位于同一个范围的新记录
Snapshot	Snapshot 只能用于 SQL Server 2005 及其以后的版本。在复制修改的记录时，这个级别会减少锁定。这样，其他事务处理就可以读取旧数据，而无需等待解锁
Unspecified	Unspecified 表示，提供程序使用另一个隔离级别值，它不同于 IsolationLevel 枚举定义的值
Chaos	Chaos 类似于 ReadUncommitted，但除了执行 ReadUncommitted 值的操作之外，它不能锁定更新的记录

表 22-5 总结了设置最常用的事务处理隔离级别时可能出现的问题。

表 22-5

隔离级别	脏 读 取	不可重复的读取	幻 影 读 取
ReadUncommitted	Y	Y	Y
ReadCommitted	N	Y	Y
RepeatableRead	N	N	Y
Serializable	N	N	N

下面的代码段说明了如何使用 TransactionScope 类设置隔离级别。在 TransactionScope 类的构造函数中，可以设置前面讨论的 TransactionScopeOption 和 TransactionOptions。TransactionOptions 类允许定义 IsolationLevel 和 Timeout。

```
TransactionOptions options = new TransactionOptions();
options.IsolationLevel = IsolationLevel.ReadUncommitted;
options.Timeout = TimeSpan.FromSeconds(90);
using (TransactionScope scope =
    new TransactionScope(TransactionScopeOption.Required,
        options))
{
    // Read data without waiting for locks from other transactions,
    // dirty reads are possible.
}
```

22.6 定制资源管理器

新事务处理模型的一个最大的优点是，很容易创建参与事务处理的定制资源管理器。资源管理器不仅管理稳定的资源，还管理不稳定或内存中的资源，例如简单的 int 和泛型列表。

图 22-5 显示了资源管理器和事务处理类之间的关系。资源管理器实现了接口



IEnlistmentNotification，该接口定义了方法 Prepare()、InDoubt()、Commit()和 Rollback()。资源管理器实现这个接口，是为了管理资源的事务处理。作为事务处理的一部分，资源管理器必须用 Transaction 类登记。不稳定的资源管理器调用方法 EnlistVolatile()，稳定的资源管理器调用方法 EnlistDurable()。根据事务处理的结果，事务处理管理器通过资源管理器从 IEnlistmentNotification 接口中调用不同的方法。

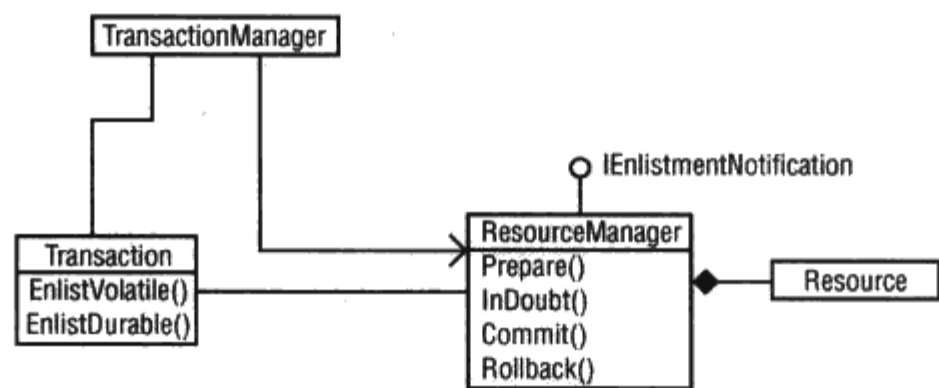


图 22-5

表 22-6 介绍了 IEnlistmentNotification 接口中必须通过资源管理器实现的方法。本章前面解释了激活、准备和提交阶段。

表 22-6

IEnlistment-Notification 成员	说 明
Prepare()	事务处理管理器调用 Prepare()方法准备事务处理。资源管理器调用 Preparing Enlistment 参数(它传送给 Prepare()方法)的 Prepared()方法，来完成准备阶段。如果工作没有成功完成，资源管理器就调用 ForceRollback()方法，通知事务处理管理器 稳定的资源管理器必须编写一个日志，才能在准备阶段之后成功完成事务处理
Commit()	所有的资源管理器都成功准备好了事务处理，事务处理管理器就调用 Commit()方法。资源管理器现在可以完成工作，使之可在事务处理的外部可见，并调用 Enlistment 参数的 Done()方法
Rollback()	如果一个资源管理器没有成功准备事务处理，那么事务处理管理器就调用所有资源管理器的 Rollback()方法。在状态返回为事务处理之前的状态后，资源管理器就调用 Enlistment 参数的 Done()方法
InDoubt()	如果事务处理管理器调用 Commit()方法后出现了一个问题(资源管理器没有用 Done()方法返回完成信息)，事务处理管理器就调用 InDoubt()方法

事务处理的资源

事务处理的资源必须保存实时值和临时值。实时值从事务处理的外部读取，定义了事务处理回滚时的有效状态。临时值定义了事务处理提交时的有效状态。

为了使非事务处理类型变成事务处理类型，泛型类 Transactional<T>封装了一个非泛型类

型，它的用法如下：

```
Transactional<int> txInt = new Transactional<int>();
Transactional<string> txString = new Transactional<string>();
```

下面看看类 `Transactional<T>` 的实现代码。托管资源的实时值包含在变量 `liveValue` 中，与事务处理相关的临时值存储在 `ResourceManager<T>` 中。变量 `enlistedTransaction` 与环境事务处理(假定存在一个)关联起来。

```
using System.Diagnostics;
using System.Transactions;

namespace Wrox.ProCSharp.Transactions
{
    public partial class Transactional<T>
    {
        private T liveValue;
        private ResourceManager<T> enlistment;
        private Transaction enlistedTransaction;
```

在 `Transactional` 构造函数中，实时值设置为变量 `liveValue`。如果在环境事务处理中调用该构造函数，就调用帮助方法 `GetEnlistment()`。`GetEnlistment()` 先检查是否有一个环境事务处理。如果没有登记事务处理，就实例化 `ResourceManager<T>` 帮助类，调用 `EnlistVolatile()` 方法，用该事务处理登记资源管理器。另外，变量 `enlistedTransaction` 设置为该环境事务处理。

如果环境事务处理不同于已登记的事务处理，就抛出一个异常。该实现代码不支持在两个不同的事务处理中修改相同的值。如果有这个要求，就可以创建一个锁定，等待一个事务处理释放该锁定，之后在另一个事务处理中修改它。

```
public Transactional(T value)
{
    if (Transaction.Current == null)
    {
        this.liveValue = value;
    }
    else
    {
        this.liveValue = default(T);
        GetEnlistment().Value = value;
    }
}

public Transactional()
    : this(default(T)) {}

private ResourceManager<T> GetEnlistment()
{
    Transaction tx = Transaction.Current;
    Trace.Assert(tx != null, "Must be invoked with ambient transaction");

    if (enlistedTransaction == null)
    {
        enlistment = new ResourceManager<T>(this, tx);
        tx.EnlistVolatile(enlistment, EnlistmentOptions.None);
        enlistedTransaction = tx;
        return enlistment;
    }
}
```

```

    }
    else if (enlistedTransaction == Transaction.Current)
    {
        return enlistment;
    }
    else
    {
        throw new TransactionException(
            "This class only supports enlisting with one transaction");
    }
}

```

属性 Value 返回所包含的类的值，并设置该值。但是，在事务处理中，不能只设置和返回变量 liveValue。只要对象在事务处理的外部，才能设置和返回它。为了使代码更容易理解，属性 Value 在其实现代码中使用了 GetValue()和 SetValue()方法。

```

public T Value
{
    get { return GetValue(); }
    set { SetValue(value); }
}

```

方法 GetValue()检查是否存在环境事务处理。如果不存在，就返回变量 liveValue。如果有环境事务处理，前面的 GetEnlistment()就返回资源管理器，使用 Value 属性，返回事务处理中所包含对象的临时值。

SetValue()方法非常类似于 GetValue()，其区别是它修改实时值或临时值。

```

protected virtual T GetValue()
{
    if (Transaction.Current == null)
    {
        return liveValue;
    }
    else
    {
        return GetEnlistment().Value;
    }
}

protected virtual void SetValue(T value)
{
    if (Transaction.Current == null)
    {
        liveValue = value;
    }
    else
    {
        GetEnlistment().Value = value;
    }
}

```

Transactional<T>类中实现的 Commit()和 Rollback()方法在资源管理器中调用。Commit()方法从第一个变元的临时值中设置实时值，使变量 enlistedTransaction 置空，因为事务处理已完成了。在 Rollback()方法中，事务处理也完成了，但这里忽略了临时值，只使用实时值。

```

internal void Commit(T value, Transaction tx)

```

```

    {
        liveValue = value;
        enlistedTransaction = null;
    }

    internal void Rollback(Transaction tx)
    {
        enlistedTransaction = null;
    }
}

```

`Transactional<T>`类使用的资源管理器仅用于 `Transactional<T>`类，所以它实现为一个内部类。在构造函数中，父变量设置为与事务处理封装类关联起来。在事务处理中使用的临时值从实时值中复制。注意事务处理需要隔离。

```

using System;
using System.Transactions;
using System.Diagnostics;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

namespace Wrox.ProCSharp.Transactions
{
    public partial class Transactional<T>
    {
        internal class ResourceManager<T1> : IEnlistmentNotification
        {
            private Transactional<T1> parent;
            private Transaction currentTransaction;

            internal ResourceManager(Transactional<T1> parent, Transaction tx)
            {
                this.parent = parent;
                tempValue = DeepCopy(parent.liveValue);
                currentTransaction = tx;
            }

            public T1 Value { get; set; }
        }
    }
}

```

因为临时值可能在事务处理中变化，所以封装类的实时值可能不会在事务处理中发生变化。在创建一些类的副本时，可以调用在 `ICloneable` 接口中定义的 `Clone()`方法。但是，在定义 `Clone()`方法时，允许实现代码创建浅度或深度副本。如果类型 `T` 包含引用类型，并实现了浅度副本，改变临时值也会改变初始值。这会与事务处理的隔离和一致特性冲突。这里需要一个深度副本。

为了进行深度复制，方法 `DeepCopy()`可在流中串行化和并行化对象。在 C# 3.0 中，不能定义对类型 `T` 的限制，即指定需要串行化，所以 `Transactional<T>`类的静态构造函数会检查 `Type`对象的 `IsSerializable` 属性，以确定类型是否可以串行化。

```

static ResourceManager()
{
    Type t = typeof(T1);
    Trace.Assert(t.IsSerializable, "Type " + t.Name + " is not serializable");
}

private T1 DeepCopy(T1 value)
{
}

```



```

using (MemoryStream stream = new MemoryStream())
{
    BinaryFormatter formatter = new BinaryFormatter();
    formatter.Serialize(stream, value);
    stream.Flush();
    stream.Seek(0, SeekOrigin.Begin);

    return (T1)formatter.Deserialize(stream);
}
}

```

接口 `IEnlistmentNotification` 由类 `ResourceManager<T>` 实现。这是用事务处理进行登记的要求。

只有用 `preparingEnlistment` 调用 `Prepared()` 方法, `Prepare()` 的实现代码才会响应。在将临时值赋予实时值时, 不应有问题, 所以 `Prepare()` 方法会成功。在 `Commit()` 方法的实现代码中, 调用父对象的 `Commit()` 方法。其中, 把变量 `liveValue` 设置为事务处理中使用的 `ResourceManager` 的值。 `Rollback()` 方法仅完成工作, 不改变实时值。对于不稳定的资源, 在 `InDoubt()` 方法中不执行太多的操作。写入一个日志项可能会有用。

```

public void Prepare(PreparingEnlistment preparingEnlistment)
{
    preparingEnlistment.Prepared();
}

public void Commit(Enlistment enlistment)
{
    parent.Commit(tempValue, currentTransaction);
    enlistment.Done();
}

public void Rollback(Enlistment enlistment)
{
    parent.Rollback(currentTransaction);
    enlistment.Done();
}

public void InDoubt(Enlistment enlistment)
{
    enlistment.Done();
}
}
}

```

现在, 只要类型是可以串行化的, `Transactional<T>` 类就可以用于使非事务处理类变成事务处理类, 例如, `int`、`string`, 甚至更复杂的类, 如 `Student`。

```

using System;
using System.Transactions;

namespace Wrox.ProCSharp.Transactions
{
    class Program
    {
        static void Main()
        {
            Transactional<int> intVal = new Transactional<int>(1);

```



```

Transactional<Student> student1 = new Transactional<Student>(new Student());
student1.Value.Firstname = "Andrew";
student1.Value.Lastname = "Wilson";

Console.WriteLine("before the transaction, value: {0}", intVal.Value);
Console.WriteLine("before the transaction, student: {0}", student1.Value);

using (TransactionScope scope = new TransactionScope())
{
    intVal.Value = 2;
    Console.WriteLine("inside transaction, value: {0}", intVal.Value);

    student1.Value.Firstname = "Ten";
    student1.Value.Lastname = "Sixty-Nine";

    if (!Utilities.AbortTx())
        scope.Complete();
}
Console.WriteLine("outside of transaction, value: {0}", intVal.Value);
Console.WriteLine("outside of transaction, student: {0}", student1.Value);
}
}

```

下面的控制台输出显示了应用程序的一次运行及提交了的事务处理。

```

before the transaction, value: 1
before the transaction, student: Andrew Wilson
inside transaction, value: 2

Abort the Transaction (y/n)? n

outside of transaction, value: 2
outside of transaction, student: Ten Sixty-Nine

Press any key to continue . . .

```

## 22.7 Windows Vista 和 Windows Server 2008

### 的事务处理

可以编写一个定制的稳定资源管理器，来处理 File 和 Registry 类。基于文件的稳定资源管理器可以复制原来的文件，将对临时文件的修改写入一个临时目录，使这些改变永久保存起来。在提交事务处理时，原始文件会用临时文件替代。在 Windows Vista 和 Windows Server 2008 中，不再需要为文件和注册表编写定制的稳定资源管理器。这两个操作系统支持用文件系统和注册表进行内部的事务处理。为此，Windows Vista 和 Windows Server 2008 增加了新的 API 调用，如 CreateFileTransacted、CreateHardLinkTransacted、CreateSymbolic LinkTransacted、CopyFileTransacted 等。这些 API 调用的共同之处是，它们都需要把事务处理的一个句柄传送为变元，它们都不支持环境事务处理。不能在 .NET 3.5 中进行事务处理 API 调用，但可以使用 Platform Invoke 创建一个定制的封装器。

**提示：**

Platform Invoke 详见第 24 章。

示例应用程序封装了本地方法 `CreateFileTransacted()`，在.NET 应用程序中创建事务处理的文件流。

在调用本地方法时，本地方法的参数必须映射为.NET 数据类型。出于安全考虑，.NET 2.0 引入了 `SafeHandle` 类来映射内置的 `HANDLE` 类型。`SafeHandle` 是一个抽象类型，它封装了操作系统句柄，支持句柄资源的关键清理操作。根据句柄的允许值，可以使用派生类 `SafeHandleMinusOneIsInvalid` 和 `SafeHandleZeroOrMinusOneIsInvalid` 封装本地句柄。`SafeFileHandle` 类派生自 `SafeHandleZeroOrMinusOneIsInvalid`。为了把句柄映射到事务处理上，定义了 `SafeTransactionHandle` 类。

```
using System;
using Microsoft.Win32.SafeHandles;
using System.Runtime.Versioning;
using System.Security.Permissions;

namespace Wrox.ProCSharp.Transactions
{
    [SecurityPermission(SecurityAction.LinkDemand, UnmanagedCode = true)]
    public sealed class SafeTransactionHandle : SafeHandleZeroOrMinusOneIsInvalid
    {
        private SafeTransactionHandle()
            : base(true) { }
        public SafeTransactionHandle(IntPtr preexistingHandle, bool ownsHandle)
            : base(ownsHandle)
        {
            SetHandle(preexistingHandle);
        }

        [ResourceExposure(ResourceScope.Machine)]
        [ResourceConsumption(ResourceScope.Machine)]
        protected override bool ReleaseHandle()
        {
            return CloseHandle(handle);
        }
    }
}
```

.NET 中的所有本地方法都是用下面的 `NativeMethods` 类定义的。在示例中，所需的本地 API 是 `CreateFileTransacted()` 和 `CloseHandle()`，它们定义为类的静态成员。方法声明为 `extern`，因为它没有 C# 实现代码。其实现代码在本地 DLL 中由 `DllImport` 属性定义。这两个方法都可以在本地 DLL `Kernel32.dll` 中找到。在方法声明中，用 Windows API 调用定义的参数映射为.NET 数据类型。`txHandle` 参数表示事务处理的一个句柄，其类型是以前定义的 `SafeTransactionHandle`。

```
using System;
using System.Runtime.ConstrainedExecution;
using System.Runtime.InteropServices;
using System.Runtime.Versioning;
using Microsoft.Win32.SafeHandles;

namespace Wrox.ProCSharp.Transactions
{
    internal static class NativeMethods
    {
        [DllImport("Kernel32.dll",
            CallingConvention = CallingConvention.StdCall,
```

```

        CharSet = CharSet.Unicode)]
    internal static extern SafeFileHandle CreateFileTransacted(
        String lpFileName,
        uint dwDesiredAccess,
        uint dwShareMode,
        IntPtr lpSecurityAttributes,
        uint dwCreationDisposition,
        int dwFlagsAndAttributes,
        IntPtr hTemplateFile,
        SafeTransactionHandle txHandle,
        IntPtr miniVersion,
        IntPtr extendedParameter);

    [DllImport("Kernel32.dll", SetLastError = true)]
    [ResourceExposure(ResourceScope.Machine)]
    [ReliabilityContract(Consistency.WillNotCorruptState, Cer.Success)]
    [return: MarshalAs(UnmanagedType.Bool)]
    internal static extern bool CloseHandle(IntPtr handle);
}
}

```

接口 `IKernelTransaction` 用于获得事务处理句柄, 把它传送给事务处理的 Windows API 调用。这是一个 COM 接口, 必须使用 COM Interop 属性封装到 .NET 中, 如下面所示。属性 GUID 必须有与接口定义相同的标识符, 因为这是用于 COM 接口定义的标识符。

```

using System;
using System.Runtime.InteropServices;

namespace Wrox.ProCSharp.Transactions
{
    [ComImport]
    [Guid("79427A2B-F895-40e0-BE79-B57DC82ED231")]
    [InterfaceType(ComInterfaceType.InterfaceIsIUnknown)]
    public interface IKernelTransaction
    {
        void GetHandle(out SafeTransactionHandle ktmHandle);
    }
}

```

为了在 .NET 应用程序中更便于使用 Windows API 调用, 类 `TransactionFile` 定义了方法 `GetTransactedFileStream()`。这个方法的参数是一个文件名, 返回一个 `System.IO.FileStream`。返回的流是一个一般的 .NET 流, 它只引用一个事务处理文件。

在实现代码中, `TransactionInterop.GetDtcTransaction()` 创建了 `IKernelTransaction` 的一个接口指针, 它指向环境事务处理, 传送为 `GetDtcTransaction()` 的一个变元。使用接口 `IKernelTransaction`, 创建 `SafeTransactionHandle` 类型的句柄。然后把这个句柄传送给封装的 API 调用 `NativeMethods.CreateFileTransacted()`。在返回的文件句柄中, 创建一个新的 `FileStream` 实例, 返回给调用者。

```

using System.IO;
using System.Transactions;
using Microsoft.Win32.SafeHandles;
using System.Runtime.InteropServices;

namespace Wrox.ProCSharp.Transactions
{

```

```

public static class TransactedFile
{
    internal const short FILE_ATTRIBUTE_NORMAL = 0x80;
    internal const short INVALID_HANDLE_VALUE = -1;
    internal const uint GENERIC_READ = 0x80000000;
    internal const uint GENERIC_WRITE = 0x40000000;
    internal const uint CREATE_NEW = 1;
    internal const uint CREATE_ALWAYS = 2;
    internal const uint OPEN_EXISTING = 3;

    [FileIOPermission(SecurityAction.Demand, Unrestricted=true)]
    public static FileStream GetTransactedFileStream(string fileName)
    {
        IKernelTransaction ktx = (IKernelTransaction)
            TransactionInterop.GetDtcTransaction(Transaction.Current);

        SafeTransactionHandle txHandle;
        ktx.GetHandle(out txHandle);

        SafeFileHandle fileHandle = NativeMethods.CreateFileTransacted(
            fileName, GENERIC_WRITE, 0,
            IntPtr.Zero, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL,
            null,
            txHandle, IntPtr.Zero, IntPtr.Zero);
        return new FileStream(fileHandle, FileAccess.Write);
    }
}

```

现在，很容易在.NET 代码中使用事务处理 API 了。可以用 `TransactionScope` 类创建一个环境事务处理，在环境事务处理的作用域内使用 `TransactedFile` 类。如果中止了事务处理，就不会把数据写入文件。如果提交了事务处理，就可以在临时目录下找到文件。

```

using System;
using System.Transactions;
using System.IO;

namespace Wrox.ProCSharp.Transactions
{
    class Program
    {
        static void Main()
        {
            using (TransactionScope scope = new TransactionScope())
            {
                FileStream stream =
                    TransactedFile.GetTransactedFileStream("c:/temp/sample.txt");
                StreamWriter writer = new StreamWriter(stream);
                writer.WriteLine("Write a transactional file");
                writer.Close();

                if (!Utilities.AbortTx())
                    scope.Complete();
            }
        }
    }
}

```

现在就可以在同一个事务处理中使用数据库、不稳定的资源和文件了。

## 22.8 小结

本章学习了事务处理的特性，以及如何使用 `System.Transactions` 命名空间中的类创建和管理事务处理。

事务处理用 ACID 特性来描述：原子性、一致性、隔离性和持久性。并不是所有的这些特性都是必需的，例如不稳定的资源就不需要支持持久性和隔离选项。

进行事务处理的最简单的方式是创建环境事务处理，使用 `TransactionScope` 类。环境事务处理非常适合于处理不明确打开和关闭数据库连接的 ADO.NET 数据适配器和 LINQ to SQL，ADO.NET 详见第 26 章，LINQ to SQL ADO.NET 详见第 27 章。

在多个线程中使用同一个事务处理，可以使用 `DependentTransaction` 类创建对另一个事务处理的依赖。登记一个实现了 `IEnlistmentNotification` 接口的资源管理器，可以创建参与事务处理的定制资源。

最后，探讨了如何通过 .NET Framework 和 C# 使用 Windows Vista 和 Windows Server 2008 事务处理。

在 .NET Enterprise Services 中，可以创建使用 `System.Transactions` 的自动事务处理。这个技术详见第 44 章。

下一章介绍如何创建在操作系统启动时自动启动 Windows 服务。事务处理在服务中也很有用。



# 第23章

## Windows 服务

Windows 服务是可以在系统启动时自动打开(不需要任何人登录机器)的程序。

本章的主要内容如下:

- Windows 服务的体系结构; 服务程序、服务控制程序和服务配置程序的功能。
- 如何使用 `System.ServiceProcess` 命名空间中的类实现 Windows 服务。
- 在注册表中配置 Windows 服务的安装程序。
- 使用 `ServiceController` 类编写控制 Windows 服务的程序。
- 解决 Windows 服务的问题
- 响应操作系统的电源事件

首先讨论 Windows 服务的体系结构。本章的代码可以从 Wrox 网站 [www.wrox.com](http://www.wrox.com) 上下载。

### 23.1 Windows 服务

Windows 服务指的是操作系统启动时可以自动打开的应用程序。Windows 服务可以在没有交互式用户登录系统的情况下运行, 在后台进行某些处理。例如, 在 Windows Server 上, 系统联网服务应可以在客户机上服务, 无需以后登录到服务器上。在客户系统上, 服务也很有用。例如, 从 Internet 上获取新软件版本, 或在本地磁盘上进行文件清理工作。可以把 Windows 服务配置为从已进行特殊配置的用户账户或系统用户账户上运行, 用户账户的权力比系统管理员的权力更大。

**注意:**

除非特别说明, 否则把 Windows 服务简称为服务。

下面是一些服务的示例:

- Simple TCP/IP Services 是驻留一些小 TCP/IP 服务器的服务程序: 如 echo、daytime 和 quote 等。
- World Wide Publishing Service 是 Internet Information Server 的服务。
- Event Log 服务用于把消息记录到事件日志系统中。
- Windows Search 服务用于在磁盘上创建数据的索引。

可以使用 Services 管理工具查看系统上的所有服务。在 Windows 2003 Server 上, 可以通过 Start | Programs | Administrative Tools | Services 访问此工具; 而在 Windows Vista 和 Windows XP

中，可以通过 Settings | Control Panel | Administrative Tools | Services 启动此管理工具，其窗口如图 23-1 所示。

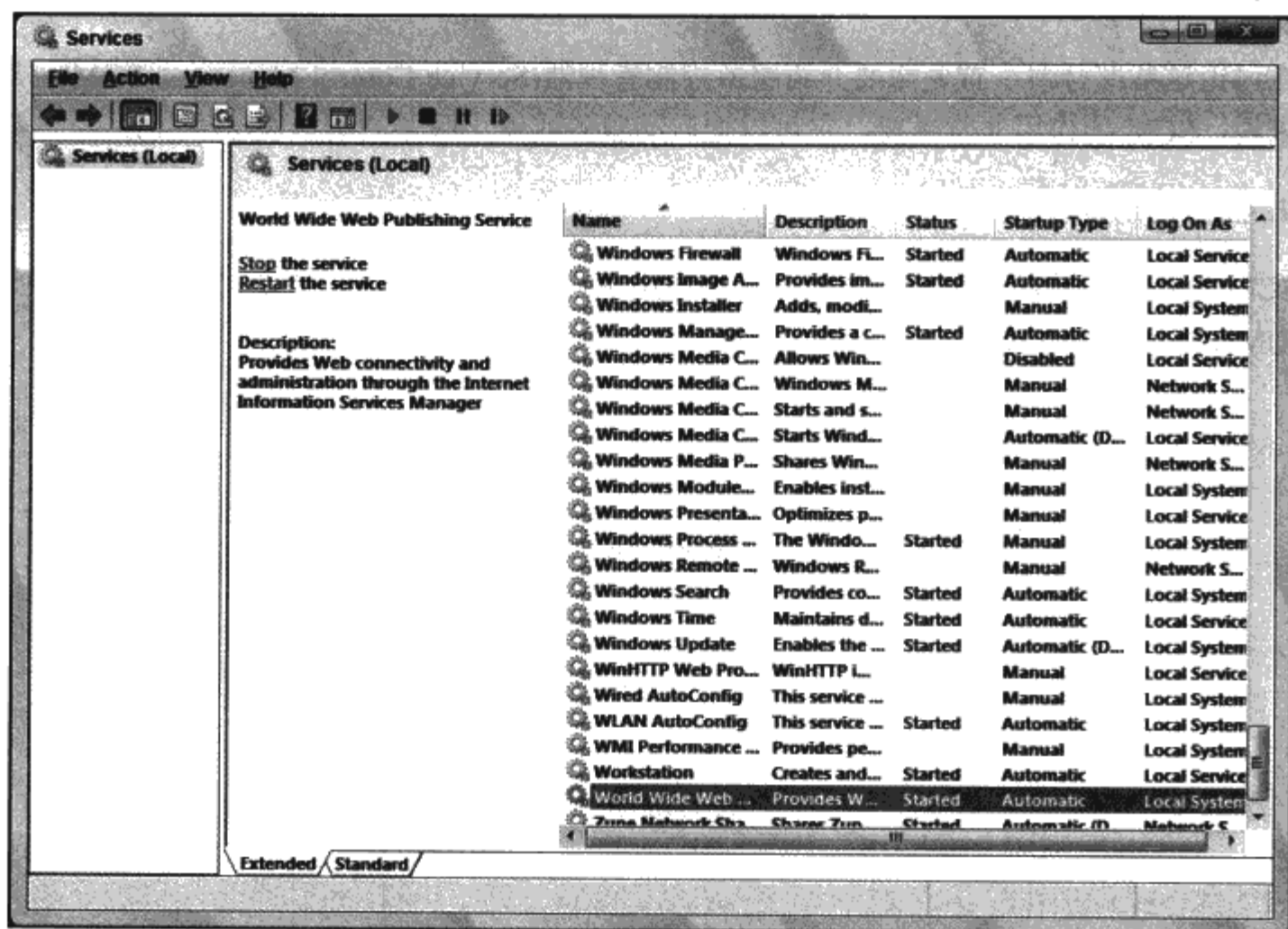


图 23-1

## 23.2 Windows 服务的体系结构

操作 Windows 服务需要 3 种程序：

- 服务程序
- 服务控制程序
- 服务配置程序

服务程序本身用于提供需要的功能。服务控制程序可以把控制请求发送给服务，例如开始、停止、暂停和继续。使用服务配置程序可以安装服务，这意味着服务不但要复制到文件系统中，还要写到注册表中，并配置为一个服务。.NET 组件不需要把信息写入注册表，所以可以使用 xcopy 命令安装它们；但是，服务的安装需要注册表配置。此外，服务配置程序可以在以后改变服务的配置。

下面介绍 Windows 服务的 3 个组成部分。

### 23.2.1 服务程序

在讨论服务的 .NET 实现方式之前，首先讨论服务的 Windows 体系和服务的内部功能。服务程序实现服务的功能。服务程序需要 3 个部分：

- 主函数
- service-main 函数
- 处理程序

在讨论这些部分前，首先需要介绍服务控制管理器(Service Control Manager, SCM)。对于服务来说，SCM 的作用非常重要，它可以把启动服务或停止服务的请求发送给服务。

1. 服务控制管理器

SCM 是操作系统的一个组成部分，它的作用是与服务进行通信。图 23-2 阐明了这种通信处理 UML 序列图的方式。

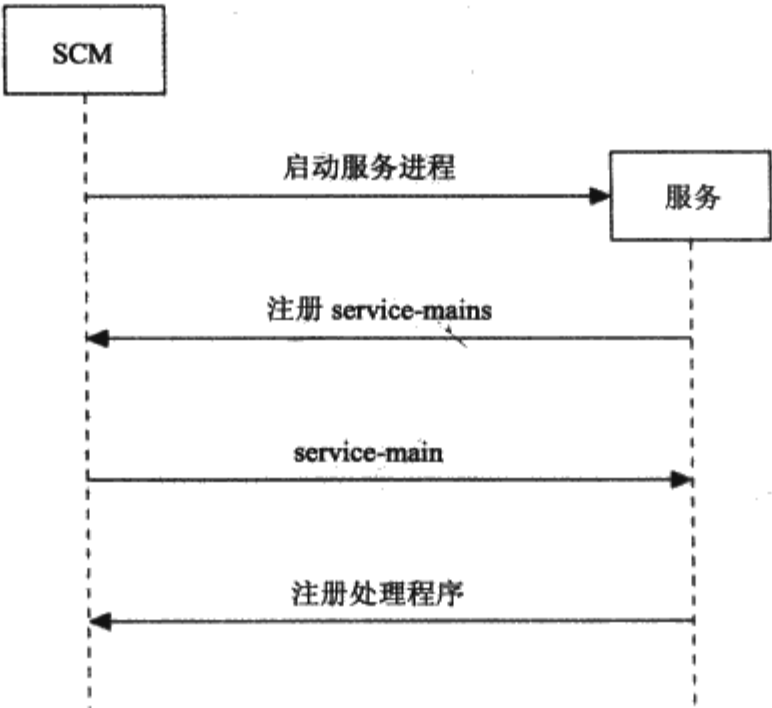


图 23-2

如果将服务设置为自动启动，则在系统启动时，将启动该服务的进程，进而调用该进程的主函数。服务负责为它的每一个服务都注册一个 service-main 函数。主函数是服务程序的入口，在这里，service-main 函数的入口必须用 SCM 注册。

2. 主函数、service-main 和处理程序

服务的主函数是程序的一般入口点，即 Main()方法，它可以注册多个 service-main 函数，service-main 函数包含服务的功能。服务必须为所提供的每个服务注册一个 service-main 函数。服务程序可以在一个程序中提供许多服务，例如<windows>\system32\services.exe 这个服务程序就包括 Alerter、Application Management、Computer Browser 和 DHCP Client 等服务。

SCM 现在为每一个应该启动的服务调用 service-main 函数。service-main 函数的一个重要任务是用 SCM 注册一个处理程序。

处理程序函数是服务程序的第三部分，处理程序必须对来自 SCM 的事件作出响应。服务可以停止、暂停或重新开始，处理程序必须对这些事件做出响应。

一旦使用 SCM 注册了处理程序，服务控制程序可以把停止、暂停和继续服务的请求发送给 SCM。服务控制程序独立于 SCM 和服务本身。在操作系统中有许多服务控制程序，以前介

绍的 MMC Services 管理单元就是其中的一个。也可以编写自己的服务控制程序，一个比较好的服务控制程序是 SQL Server Service Manager，如图 23-3 所示。

### 23.2.2 服务控制程序

顾名思义，使用服务控制程序可以控制服务。为了停止、暂停和重新启动服务，可以把控制代码发送给服务，处理程序应该响应这些事件。此外，还可以询问服务的实际状态，执行响应定制控制代码的定制处理程序。

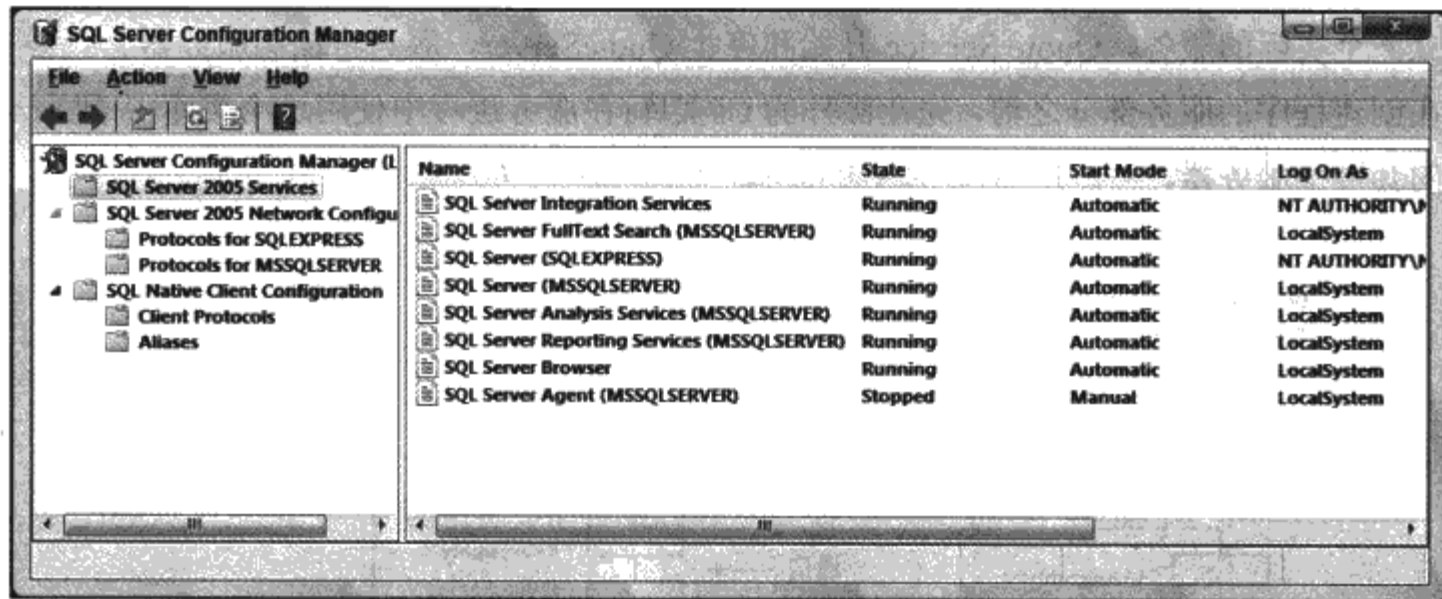


图 23-3

### 23.2.3 服务配置程序

不能使用 xcopy 安装服务，服务必须在注册表中配置，注册表包含了服务的启动类型，该启动类型可以设置为自动、手动或禁用。必须配置服务程序的用户、服务的依存关系(例如，一个服务必须在另一个服务开始之前启动)。所有的配置工作都在服务配置程序中进行，安装程序可以使用服务配置程序配置服务，服务配置程序也可以在以后改变服务配置参数。

## 23.3 System.ServiceProcess 命名空间

在 .NET Framework 中，可以在 System.ServiceProcess 命名空间中找到实现服务的 3 个部分的服务类：

- 从 ServiceBase 类继承的类可以实现服务。ServiceBase 类用于注册服务、响应开始和停止请求。
- ServiceController 类用于实现服务控制程序。使用这个类，可以把请求发送给服务。
- 顾名思义，ServiceProcessInstaller 类和 ServiceInstaller 类用于安装和配置服务程序。

下面介绍怎样创建新的服务。

## 23.4 创建 Windows 服务

我们创建的服务将驻留在引用服务器内。对于客户发出的每一个请求，引用服务器都返回引用文件的一个随机引用。解决方案的第一部分由 3 个程序集完成，一个用于客户机，两个用于服务器，图 23-4 显示了这个解决方案。程序集 QuoteServer 包含实际的功能。服务可以在内存中读取引用，然后在套接字服务器的帮助下响应引用的请求。QuoteClient 是 Windows Forms 胖客户应用程序。这个应用程序创建客户套接字，以便与 Quote Server 进行通信。第三个程序集将建立一个实际的服务，Quote Service 开始和停止 QuoteServer，服务将控制服务器。

在创建程序的服务部分之前，先在额外的 C#类库(在服务进程中使用这个类库)中建立一个简单的套接字服务器。

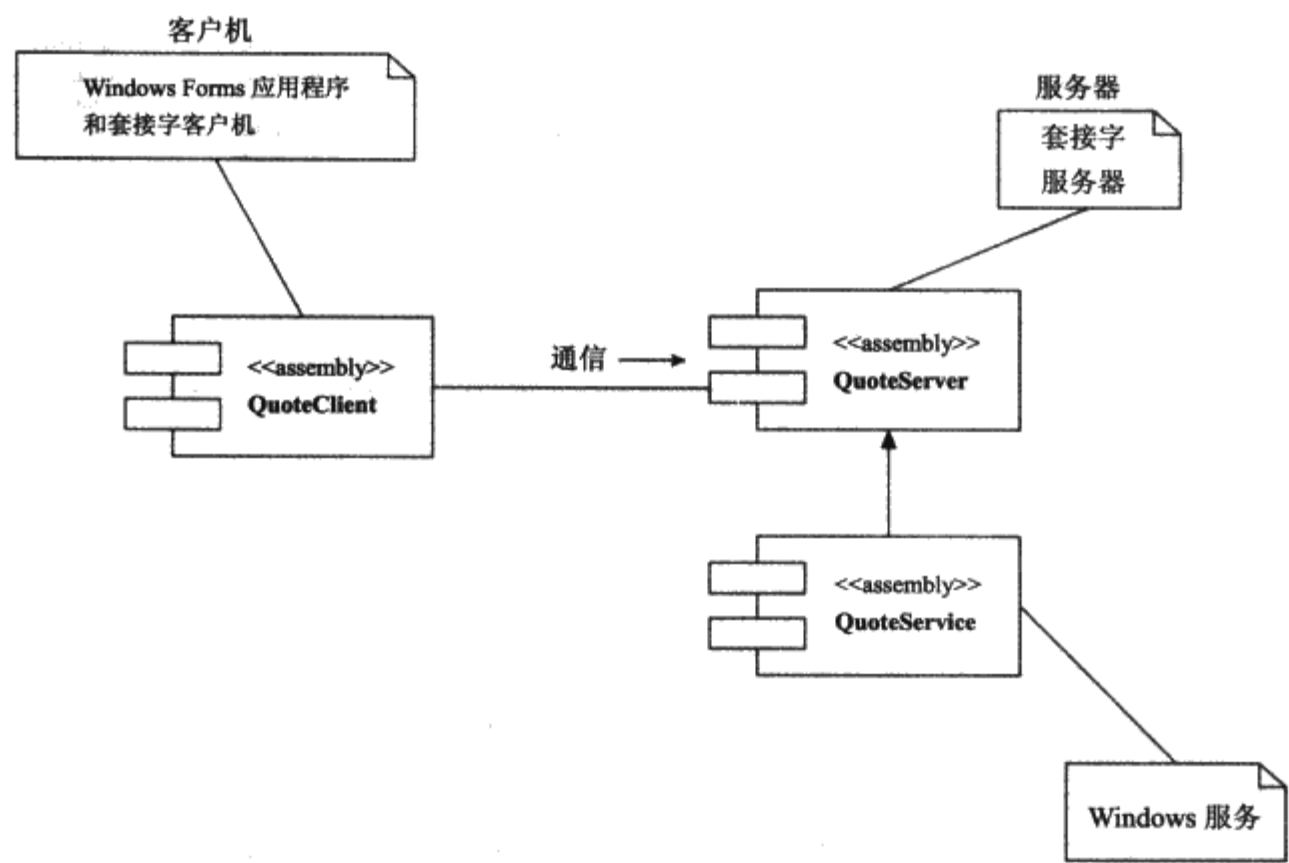


图 23-4

### 23.4.1 使用套接字的类库

可以在服务中建立任何功能，例如扫描文件、进行备份或病毒检查，或者启动 WCF 服务器。但所有的服务程序都有一些类似的地方。这种程序必须能启动(并返回给调用者)，能停止和暂停。下面讨论用套接字服务器实现的程序。

对于 Windows Vista 系统，Simple TCP/IP Services 可以安装为 Windows 组件的一个组成部分。Simple TCP/IP Services 的一个部分是“quote of the day”TCP/IP 服务器，它的缩写是“qotd”。这个简单的服务在端口 17 处监听，并使用文件<windir>\system32\drivers\etc \quotes 中的随机消息响应每一个请求。我们将在这里创建一个相似的服务器，它返回一个 Unicode 字符串，而不是象 qotd 服务器那样返回 ASCII 代码。



首先创建一个类库 QuoteServer，执行服务器的代码。下面详细解释 QuoteServer.cs 文件中 QuoteServer 类的源代码：

```
using System;
using System.IO;
using System.Threading;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Collections.Generic;
namespace Wrox.ProCSharp.WinServices
{
    public class QuoteServer
    {
        private TcpListener listener;
        private int port;
        private string filename;
        private List<string> quotes;
        private Random random;
        private Thread listenerThread;
```

重载 QuoteServer() 构造函数，把文件名和端口传递给调用程序。只接收文件名的构造函数使用服务器的 7890 默认端口，默认的构造函数把引用的默认文件名定义为 quotes.txt：

```
public QuoteServer() : this ("quotes.txt")
{
}
public QuoteServer(string filename) : this(filename, 7890)
{
}
public QuoteServer(string filename, int port)
{
    this.filename = filename;
    this.port = port;
}
```

ReadQuotes() 是一个帮助方法，它从构造函数指定的文件中读取所有的引用，所有的引用都添加给 StringCollection 引用。此外，创建 Random 类的一个实例，用于返回随机的引用：

```
protected void ReadQuotes()
{
    quotes = new List<string>();
    Stream stream = File.OpenRead(filename);
    StreamReader streamReader = new StreamReader(stream);
    string quote;
    while ((quote = streamReader.ReadLine()) != null)
    {
        quotes.Add(quote);
    }
    streamReader.Close();
    stream.Close();
    random = new Random();
}
```

另一个帮助方法是 GetRandomQuoteOfTheDay()，它返回 StringCollection 引用的一个随机引用：

```
protected string GetRandomQuoteOfTheDay()
```

```

    {
        int index = random.Next(0, quotes.Count);
        return quotes[index];
    }

```

在 `Start()` 方法中, 使用帮助函数 `ReadQuotes()`, 在 `StringCollection` 中读取包含引用的完整文件。在新的线程打开之后, 它立即调用 `Listener()` 方法。这类似于第 41 章的 `TcpReceive` 示例。

这里使用了线程, 因为 `Start()` 方法不能停下来等待客户, 它必须立即返回给调用者(即 SCM)。如果方法没有及时返回给调用者(30 秒), SCM 就假定启动失败。监听线程设置为后台线程, 这样应用程序就可以在不该线程的情况下退出。设置线程的 `Name` 属性, 是因为这有助于调试, 因为其名称会显示在调试器中:

```

public void Start()
{
    ReadQuotes();
    listenerThread = new Thread(ListenerThread);
    listenerThread.IsBackground = true;
    listenerThread.Name = "Listener";
    listenerThread.Start();
}

```

线程函数 `ListenerThread()` 创建一个 `TcpListener` 实例。在 `AcceptSocket()` 方法中, 我们等待客户进行连接。客户一连接, `AcceptSocket()` 就返回一个与客户相关联的套接字。之后使用 `socket.Send()`, 调用 `GetRandomQuoteOfTheDay()` 把返回的随机引用发送给客户:

```

protected void ListenerThread ()
{
    try
    {
        IPAddress ipAddress = IPAddress.Parse("127.0.0.1");
        listener = new TcpListener(ipAddress, port);
        listener.Start();
        while (true)
        {
            Socket clientSocket = listener.AcceptSocket();
            string message = GetRandomQuoteOfTheDay();
            UnicodeEncoding encoder = new UnicodeEncoding();
            byte[] buffer = encoder.GetBytes(message);
            clientSocket.Send(buffer, buffer.Length, 0);
            clientSocket.Close();
        }
    }
    catch (SocketException ex)
    {
        Console.WriteLine(ex.Message);
    }
}

```

除了 `Start()` 方法之外, 还需要有其他的方法来控制服务: `Stop()`、`Suspend()` 和 `Resume()`:

```

public void Stop()
{
    listener.Stop();
}
public void Suspend()

```

```

    {
        listener.Stop();
    }
    public void Resume()
    {
        Start();
    }
}

```

另一个公共方法是 `RefreshQuotes()`。如果包含引用的文件发生了变化，就要使用这个方法重新读取文件：

```

    public void RefreshQuotes()
    {
        ReadQuotes();
    }
}

```

在服务器上建立服务之前，首先应该建立一个测试程序，这个测试程序要创建 `QuoteServer` 的一个实例，并调用 `Start()`。这样，不需要处理与具体服务相关的问题，就能够测试服务的功能。测试服务器必须手动启动，使用调试程序，可以很容易调试代码。

测试程序是一个 C# 控制台应用程序 `TestQuoteServer`，我们必须引用 `QuoteServer` 类的程序集。包含引用的文件必须复制到 `c:\ProCSharp\services` 目录中(或者必须在构造函数中改动参数，以指定在什么地方复制文件)。在调用构造函数之后，就调用 `QuoteServer` 实例的 `Start()` 方法。`Start()` 在创建线程之后立即返回，因此，在按下回车键之前，控制台应用程序一直处于运行状态。

```

static void Main
{
    QuoteServer qs = new QuoteServer
    (@c:\ProCSharp\WindowsServices\quotes.txt",4567);
    qs.Start();
    Console.WriteLine("Hit return to exit");
    Console.ReadLine();
    qs.Stop();
}

```

注意，`QuoteServer` 将运行在使用这个程序的本地主机 4567 端口上——后面的内容将需要在客户机中使用这些设置。

### 23.4.2 TcpClient 示例

客户端是一个简单的 WPF Windows 应用程序，可以在此请求服务器的引用。客户端应用程序使用 `TcpClient` 类连接正在运行的服务器，然后接收返回的消息，并显示在文本框中。如图 23-5 所示。

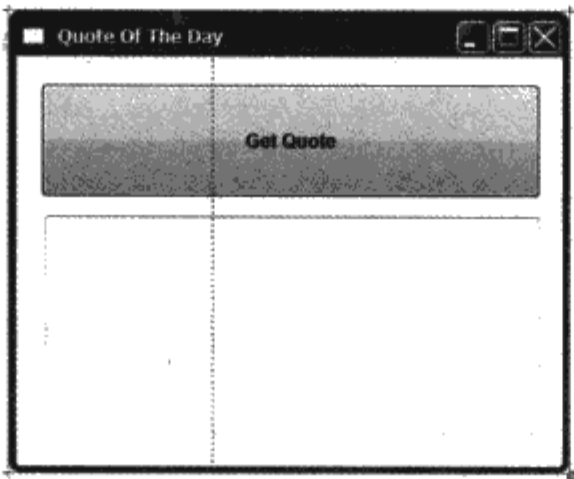


图 23-5

连接服务器的服务器名称和端口信息用应用程序的设置来配置。在项目的属性中，可以用 Settings 选项卡来添加设置，如图 23-6 所示。这里定义了 ServerName 和 PortName 设置，以及一些默认值。把 Scope 设置为 User，设置就会保存到用户特定的配置文件中，应用程序的每个用户都可以有不同的设置。Visual Studio 的 Settings 特性也会创建一个 Settings 类，以使用一个强类型化的类来读写设置。

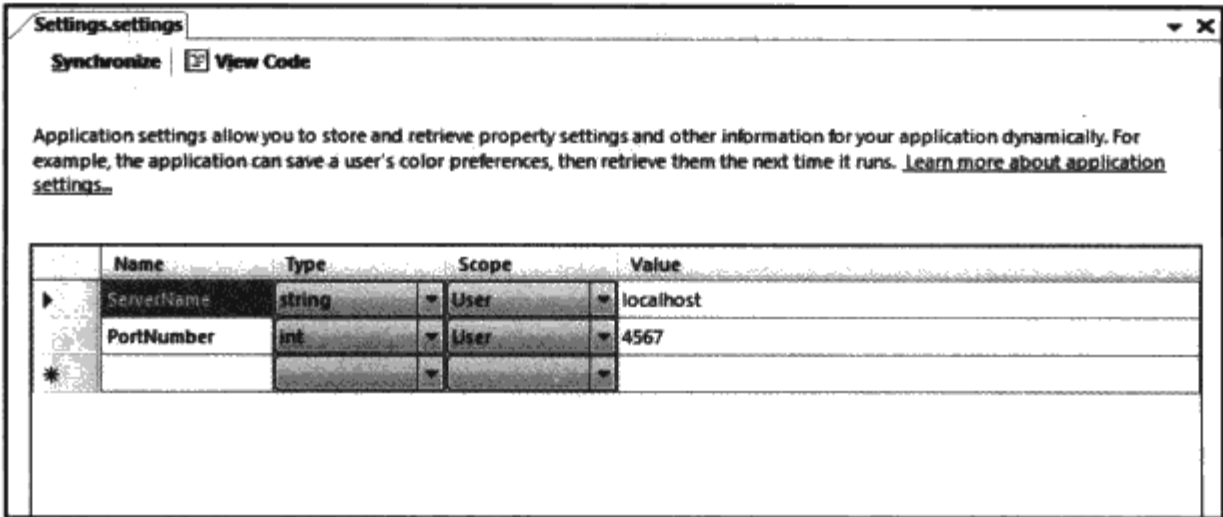


图 23-6

必须在代码中使用下面的 using 语句：

```
using System;
using System.Text;
using System.Windows;
using System.Windows.Input;
using System.Net.Sockets;
```

在类 QuoteOfTheDayWindow 的构造函数中，可以给 buttonGetQuote 按钮的 Click 事件定义一个处理程序方法：

```
public QuoteOfTheDayWindow()
{
    InitializeComponent();
    this.buttonGetQuote.Click += new RoutedEventHandler(OnGetQuote);
}
```

客户端的主要功能体现在 Get Quote 按钮的单击事件的处理程序中。



```

protected void OnGetQuote(object sender, RoutedEventArgs e)
{
    Cursor curerntCursor = this.Cursor;
    this.Cursor = Cursors.Wait;

    string serverName = Properties.Settings.Default.ServerName;
    int port = Properties.Settings.Default.PortNumber;

    TcpClient client = new TcpClient();
    NetworkStream stream = null;
    try
    {
        client.Connect(serverName, port);
        stream = client.GetStream();
        byte[] buffer = new Byte[1024];
        int received = stream.Read(buffer, 0, 1024);
        if (received <= 0)
        {
            return;
        }
        textQuote.Content = Encoding.Unicode.GetString(buffer).Trim('\0');
    }
    catch (SocketException ex)
    {
        MessageBox.Show(ex.Message, "Error Quote of the day",
            MessageBoxButton.OK, MessageBoxImage.Error);
    }
    finally
    {
        if (stream != null)
        {
            stream.Close();
        }

        if (client.Connected)
        {
            client.Close();
        }
    }
    this.Cursor = oldCursor;
}

```

在打开测试服务器和这个 Windows 应用程序客户机之后，就可以对功能进行测试。如果运行成功，就可以得到如图 23-7 所示的结果。

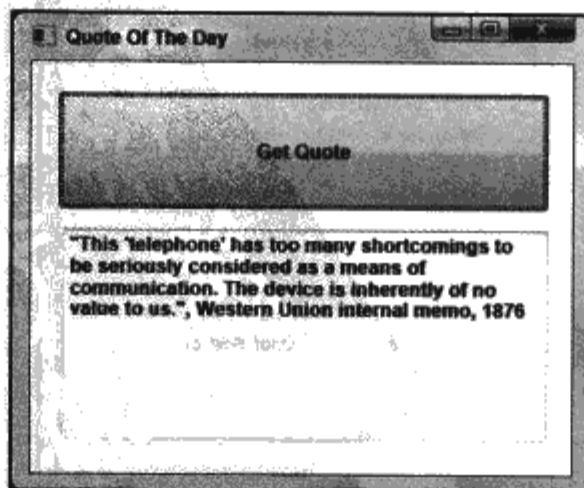


图 23-7



现在继续向服务器中添加服务功能。程序已经在运行，还需要添加什么呢？通常，在系统启动时，不需要任何人登录系统，服务器程序就应该自动地打开。我们希望使用服务控制程序对此进行控制。

23.4.3 Windows 服务项目

使用 C# Windows 服务的新项目向导可以创建 Windows 服务，该项目命名为 QuoteService，其窗口如图 23-8 所示。

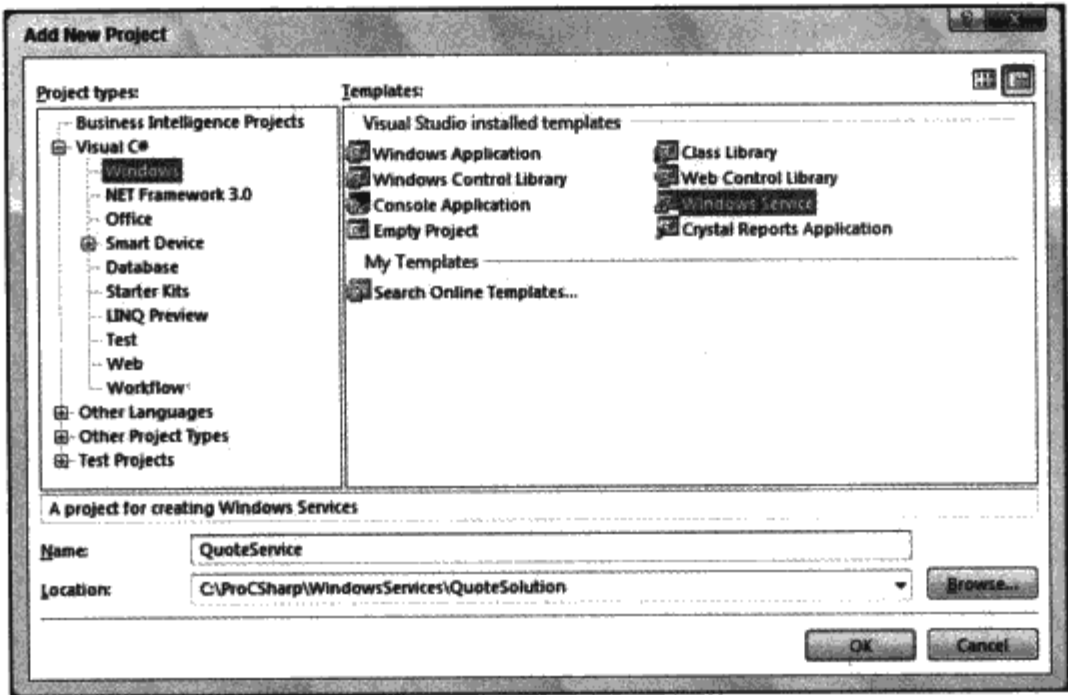


图 23-8

在单击 OK 按钮开始创建 Windows 服务应用程序之后，就会出现一个外观与 Windows Forms 应用程序相似的设计器，但是不能在其中插入 Windows Forms 组件，因为应用程序不能直接在屏幕上显示任何信息，本章的后面将使用设计器添加性能计数器和事件日志等其他组件。

选择这个服务的属性，可以打开如图 23-9 所示的属性编辑窗口。

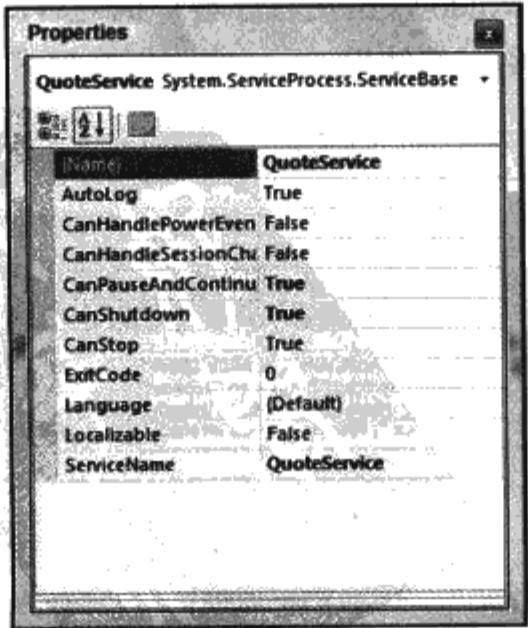


图 23-9

使用服务属性可以配置如下值：

- `AutoLog` 指定启动和停止服务的事件自动写到日志文件中。
- `CanPauseAndContinue`、`CanShutdown` 和 `CanStop` 指定服务可以处理暂停、继续、关闭和停止服务的请求。
- `ServiceName` 是写到注册表中的服务名称，使用这个名称可以控制服务。
- `CanHandleSessionChangeEvent` 确定服务是否能处理终端服务会话中的改变事件。
- `CanHandlePowerEvent` 选项对运行在膝上计算机或移动设备上的服务有效。如果启用这个选项，服务就可以响应低电源事件，改变服务的操作方式。

提示：

不管项目的名称是什么，默认的服务名称都是 `WinService1`。可以只安装一个 `WinService1` 服务。如果在测试过程中出现了安装错误，有可能已经安装了 `WinService1` 服务。因此，在服务开发的初始阶段，一定要用属性编辑器把服务的名称改为比较适当的名称。

使用属性编辑器改变上述属性，在 `InitializeComponent()` 方法中设置 `ServiceBase` 派生类的值。`Windows Forms` 应用程序中也使用 `InitializeComponent()` 方法，对于服务而言，这个方法的使用方式与 `Windows Forms` 应用程序相似。

向导将生成代码，但是我们将把文件名改为 `QuoteService.cs`，把命名空间的名称改为 `Wrox.ProCSharp.WinServices`，并把类名改为 `QuoteService`。后面将详细讨论这些代码。

## 1. ServiceBase 类

`ServiceBase` 类是所有用 .NET Framework 开发的 Windows 服务的基类。`QuoteService` 类就是从 `ServiceBase` 类派生出来的；`QuoteService` 类使用一个未标注的帮助类 `System.ServiceProcess.NativeMethods` 与 SCM 进行通信，`System.ServiceProcess.NativeMethods` 是 Win32 API 调用的包装类。`ServiceBase` 类是私有的，因此，不能在这里的代码中使用它。

图 23-10 显示了 SCM、`QuoteService` 类和 `System.ServiceProcess` 命名空间中的类是怎样相互作用的。在这个图中，垂直方向为对象的生命线，水平方向为通信情况，通信是按照时间的先后顺序而进行的。

SCM 启动应该启动的服务进程。首先调用 `Main()` 方法。在示例服务的 `Main()` 方法中，调用 `ServiceBase` 基类的 `Run()` 方法。`Run()` 使用 SCM 中的 `NativeMethods.StartServiceCtrlDispatcher()` 注册 `ServiceMainCallback()` 方法，并把记录写到事件日志中。

接下来，SCM 在服务程序中调用已注册的 `ServiceMainCallback()` 方法。`ServiceMainCallback()` 本身使用 `NativeMethods.RegisterServiceCtrlHandler[Ex]()` 在 SCM 中注册处理程序，并在 SCM 中设置服务的状态。之后调用 `OnStart()` 方法。在 `OnStart()` 中，必须执行启动代码。如果 `OnStart()` 执行成功，就把字符串 `Service Started Successful` 写到事件日志中。

处理程序是在 `ServiceCommandCallback()` 方法中执行的。当改变了对服务的请求时，SCM 就调用 `ServiceCommandCallback()` 方法。`ServiceCommandCallback()` 方法再把请求发送给 `OnPause()`、`OnContinue()`、`OnStop()`、`OnCustomCommand()` 和 `OnPowerEvent()`。

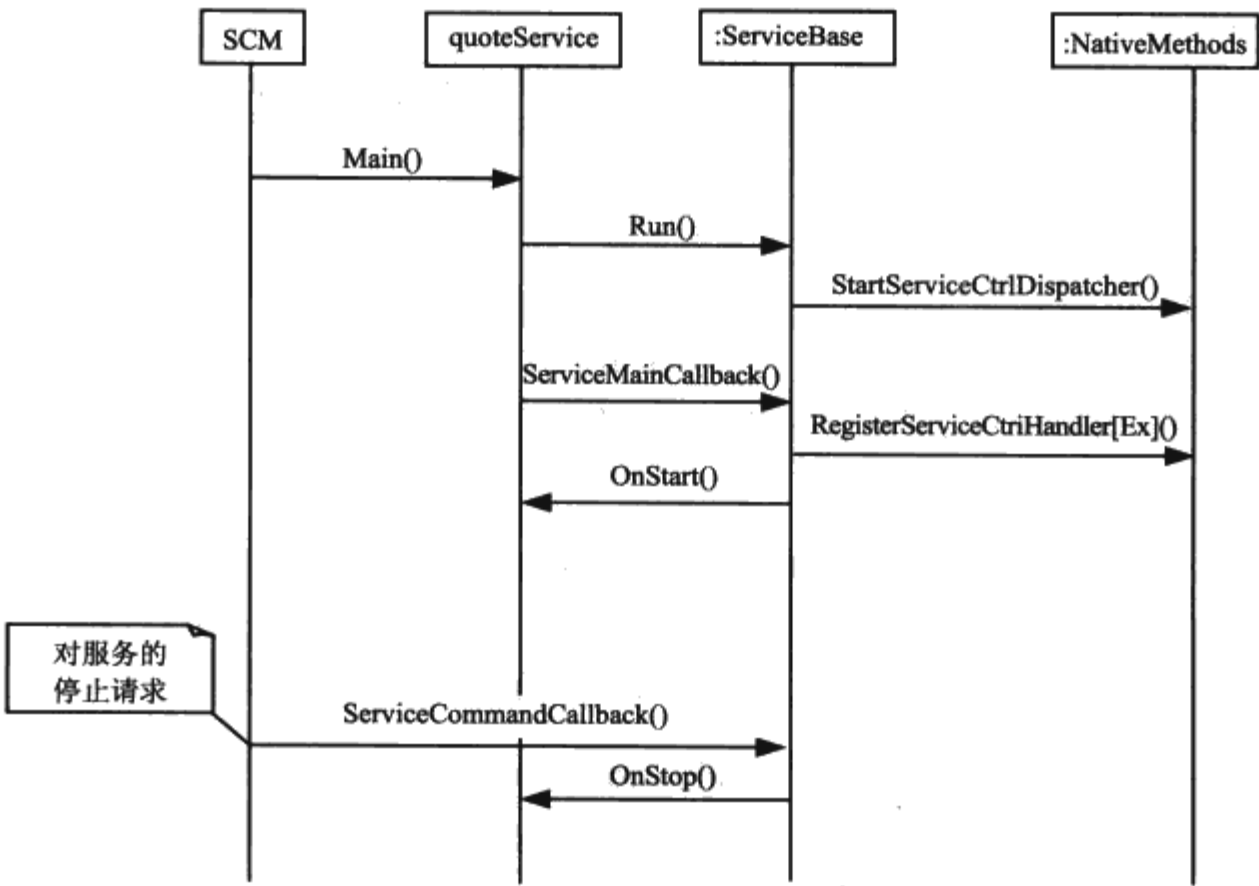


图 23-10

2. 主函数

现在讨论服务进程中由应用程序向导生成的主函数。在主函数中，声明了一个元素为 `ServiceBase` 类的数组 `ServicesToRun`。创建 `QuoteService` 类的一个实例，并作为 `ServicesToRun` 数组的第一个元素。如果在这个服务进程中要运行多个服务，就需要把具体服务类的多个实例添加到数组中。然后把 `ServicesToRun` 数组传递给 `ServiceBase` 类的静态方法 `Run()`。使用 `ServiceBase` 的 `Run()` 方法，可以把 SCM 引用提供给服务的入口点。服务进程的主线程现在处于停滞状态，等待服务的结束。

下面是自动生成的代码：

```
/// < summary >
/// The main entry point for the process
/// < /summary >
static void Main()
{
    ServiceBase[] ServicesToRun;
    ServicesToRun = new ServiceBase[]
    {
        new QuoteService()
    };
    ServiceBase.Run(ServicesToRun);
}
```

如果进程中只有一个服务，就可以删除数组。`Run()` 方法接收从 `ServiceBase` 派生出来的单个对象，因此 `Main()` 方法可以简化为：

```
System.ServiceProcess.ServiceBase.Run(new QuoteService());
```

服务程序 `Services.exe` 包含多个服务。如果有类似的服务，其中有多服务运行在一个进程中，且需要初始化多个服务的某些共享状态，则共享的初始化必须在 `Run()` 方法运行之前完成。在运行 `Run()` 方法时，主线程处于停滞状态，直到服务进程停止为止，以后的指令在服务结束之前不能执行。

初始化花费的时间不应该超过 30 秒。如果执行初始化代码所花费的时间过多，则服务控制管理器就认为服务启动失败了。初始化时间不应该超过 30 秒，必须是针对速度最慢的机器而言。如果初始化的时间过长，就应该在另一线程中进行初始化，以便主线程及时地调用 `Run()`。然后，事件对象可以用信号通知线程已经完成了它的工作。

### 3. 服务的启动

在服务启动时，调用 `OnStart()` 方法。这时，可以启动前面创建的套接字服务器。为了使用 `QuoteServer`，必须引用 `QuoteServer.dll` 程序集。调用 `OnStart()` 的线程不能停滞下来，`OnStart()` 方法必须返回给调用者（即 `ServiceBase` 类的 `ServiceMainCallback()` 方法）。`ServiceBase` 类注册处理程序，并在调用 `OnStart()` 之后把服务成功启动的消息通知给 SCM：

```
protected override void OnStart(string[] args)
{
    quoteServer = new QuoteServer(@"c:\ProCSharp\WindowsServices\quotes.txt",
                                   5678);
    quoteServer.Start();
}
```

`quoteServer` 变量声明为类中的私有成员：

```
namespace Wrox.ProCSharp.WinServices
{
    public class QuoteService : ServiceBase
    {
        private QuoteServer quoteServer;
```

### 4. 处理程序方法

当停止服务时，就调用 `OnStop` 方法。应该在 `OnStop` 方法中停止服务的功能：

```
protected override void OnStop()
{
    quoteServer.Stop();
}
```

除了 `OnStart()` 和 `OnStop()` 之外，还可以重写服务类中的下列处理程序：

- `OnPause()`：在暂停服务时，调用这个方法。
- `OnContinue()`：当服务从暂停状态返回到正常操作时，调用这个方法。为了调用已重写的 `OnPause()` 方法和 `OnContinue()` 方法，`CanPauseAndContinue` 属性必须设置为 `true`。
- `OnShutdown()`：当 Windows 操作系统关闭时，调用这个方法。通常情况下，`OnShutdown()` 方法的行为应该与 `OnStop()` 方法相似。如果需要更多的时间关闭服务，则可以请求更多的时间。与 `OnPause()` 和 `OnContinue()` 相似，必须设置一个属性使该操作有效，即 `CanShutdown` 属性必须设置为 `true`。

- **OnPowerEvent()**: 在系统的电源状态发生变化时, 调用这个方法。电源状态发生变化的信息在 **PowerBroadcastStatus** 类型的参数中, **PowerBroadcastStatus** 是一个枚举, 其值是 **BatteryLow** 和 **PowerStatusChange**。在这个方法中, 还可以获得系统是否要挂起的信息(**QuerySuspend**), 此时可以同意或拒绝挂起。电源事件详见本章后面的内容。
- **OnCustomCommand()**: 这个处理程序可以为服务控制程序发送过来的定制命令提供服务。**OnCustomCommand()**方法有一个用于获取定制命令号码的 **int** 参数, 号码的取值范围是 128 至 256, 小于 128 的值是为系统预留的。在我们的服务中, 使用定制命令号码为 128 的命令重新读取引用文件:

```
protected override void OnPause()
{
    quoteServer.Suspend();
}
protected override void OnContinue()
{
    quoteServer.Resume();
}
protected override void OnShutdown()
{
    OnStop();
}

public const int commandRefresh = 128;
protected override void OnCustomCommand(int command)
{
    switch (command)
    {
        case commandRefresh:
            quoteServer.RefreshQuotes();
            break;

        default:
            break;
    }
}
```

#### 23.4.4 线程和服务

如前所述, 如果服务的初始化花费的时间过多, 则 SCM 就假定服务启动失败。为了解决这个问题, 必须创建线程。

服务类中的 **OnStart()**方法必须及时返回。如果从 **TCPLListener** 类中调用一个 **AcceptSocket()**之类的停滞方法, 就必须启动一个线程去完成调用工作。使用能处理多个客户机的联网服务器时, 线程池也是非常有用的。**AcceptSocket()**方法应接收调用, 并在线程池的另一个线程中进行处理, 这样就不需等待代码的执行, 系统看起来似乎是立即响应的。

#### 23.4.5 服务的安装

服务必须在注册表中配置, 所有的服务都可以在 **HKEY\_LOCAL\_MACHINE\System\CurrentControlSet\Services** 中找到。使用 **regedit** 命令, 可以查看注册表的项目。在注册表中, 可以看到服务的类型、显示名称、可执行文件的路径、启动配置以及其他信息, 图 23-11 显示



了 W3SVC 服务的注册表配置。

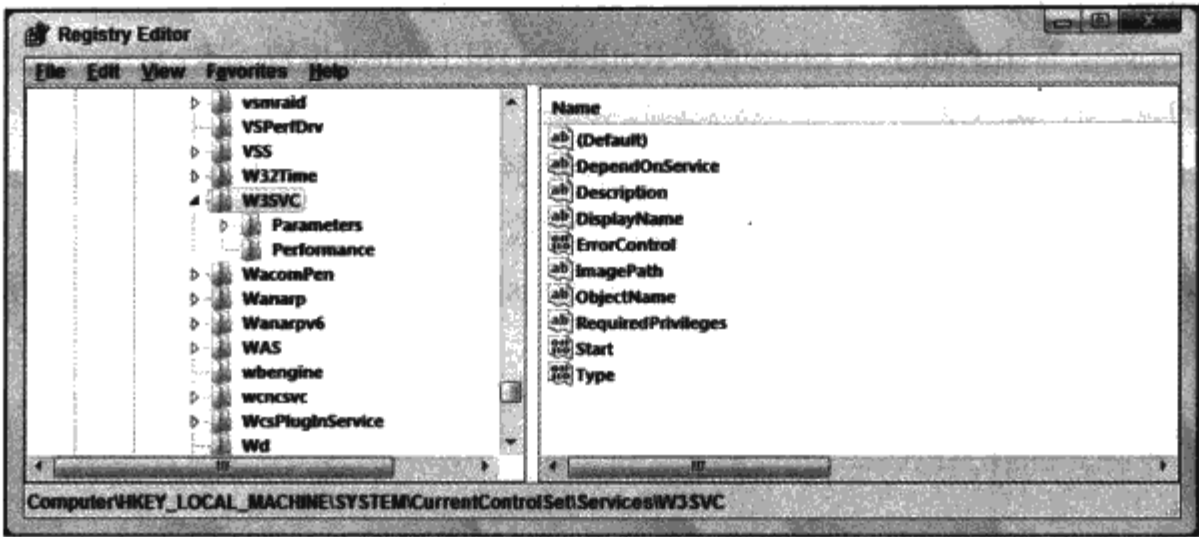


图 23-11

使用 `System.ServiceProcess` 命名空间中的安装类，可以完成服务在注册表中的配置。下面讨论这些内容。

23.4.6 安装程序

切换到 Visual Studio 的设计视图，从弹出菜单中选择 `Add Installer` 选项，就可以给服务添加安装程序。使用 `Add Installer` 选项时，将创建一个新的 `ProjectInstaller` 类、一个 `ServiceInstaller` 实例和一个 `ServiceProcessInstaller` 实例。

图 23-12 显示了服务的安装程序类。

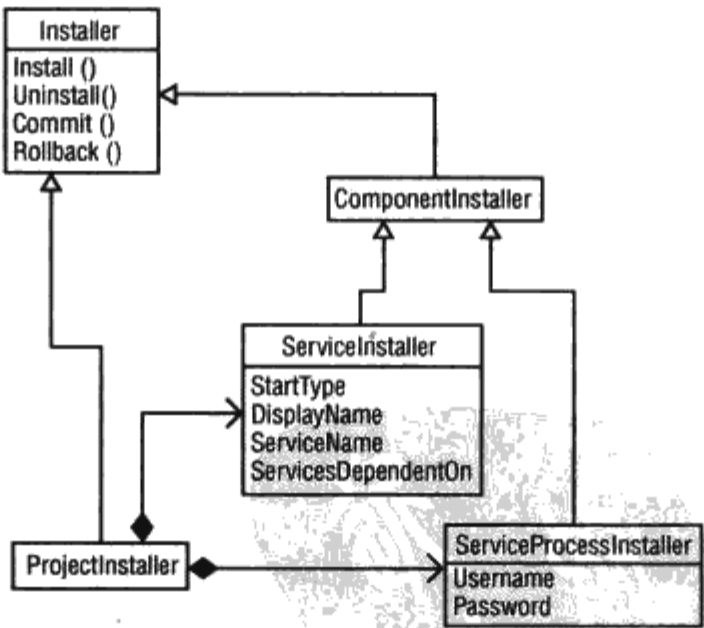


图 23-12

根据这张图表，下面详细讨论由 `Add Installer` 选项创建的 `ProjectInstaller.cs` 文件中的源代码。

1. 安装程序类

`ProjectInstaller` 类是从 `System.Configuration.Install.Installer` 派生出来的，它是所有定制安装

程序类的基类。使用 `Installer` 类，可以创建基于事务的安装。使用基于事务的安装时，如果安装失败了，系统还可以回滚到以前的状态，安装程序所做的所有修改都会被取消。如图 23-12 所示，`Installer` 类中有 `Install()`、`Commit()`、`Rollback()` 和 `Uninstall()` 方法，这些方法都是从安装程序中调用的。

如果 `RunInstaller` 属性的值为 `true`，则在安装程序集时就会调用 `ProjectInstaller` 类。定制的安装程序和 `installutil.exe` (这个程序以后将用到) 都能检查该属性。

与 `Windows` 窗体应用程序类似，在 `ProjectInstaller` 类的构造函数内部调用 `InitializeComponent()`：

```
using System.ComponentModel;
using System.Configuration.Install;

namespace Wrox.ProCSharp.WinServices
{
    [RunInstaller(true)]
    public partial class ProjectInstaller : Installer
    {
        public ProjectInstaller()
        {
            InitializeComponent();
        }
    }
}
```

## 2. `ServiceProcessInstaller` 类和 `ServiceInstaller` 类

在 `InitializeComponent()` 的执行代码中，创建了 `ServiceProcessInstaller` 类和 `ServiceInstaller` 类的实例。这两个类都是从 `ComponentInstaller` 类中派生出来的，`ComponentInstaller` 类本身派生于 `Installer`。

从 `ComponentInstaller` 中派生出来的类可以用作安装进程的一个部分。注意，一个服务进程可以包括多个服务。`ServiceProcessInstaller` 类用于配置进程，为这个进程中所有服务定义值，而 `ServiceInstaller` 类用于服务的配置，因此，每个服务都需要 `ServiceInstaller` 类的一个实例。如果进程中有 3 个服务，则必须添加额外的 `ServiceInstaller` 对象，本例需要 3 个 `ServiceInstaller` 实例：

```
partial class ProjectInstaller
{
    /// <summary>
    ///     Required designer variable.
    /// </summary>
    private System.ComponentModel.Container components = null;

    /// <summary>
    ///     Required method for Designer support - do not modify
    ///     the contents of this method with the code editor.
    /// </summary>
    private void InitializeComponent()
    {
        this.serviceProcessInstaller1 =
            new System.ServiceProcess.ServiceProcessInstaller();
        this.serviceInstaller1 =
            new System.ServiceProcess.ServiceInstaller();
    }
}
```

```
//
// serviceProcessInstaller1
//
this.serviceProcessInstaller1.Password = null;
this.serviceProcessInstaller1.Username = null;
//
// serviceInstaller1
//
this.serviceInstaller1.ServiceName = "QuoteService";
//
// ProjectInstaller
//
this.Installers.AddRange(
    new System.Configuration.Install.Installer[]
    {this.serviceProcessInstaller1,
      this.serviceInstaller1});
}

private System.ServiceProcess.ServiceProcessInstaller
    serviceProcessInstaller1;
private System.ServiceProcess.ServiceInstaller serviceInstaller1;
}
```

ServiceProcessInstaller 安装一个执行 ServiceBase 类的可执行文件。ServiceProcess Installer 类包含用于整个进程的属性。在进程中所有服务共享的属性如表 23-1 所示。

表 23-1

属 性	描 述
Username, Password	如果 Account 属性设置为 ServiceAccount.User, 则 Username 属性和 Password 属性指出服务是在哪一个账户下运行
Account	使用这个属性, 可以指定服务的账户类型
HelpText	HelpText 是只读属性, 它返回的文本用于帮助设置用户名和密码

用于运行服务的进程可以用 ServiceProcessInstaller 类的 Account 属性指定, 其值可以是 ServiceAccount 枚举的任一值, 如表 23-2 所示。

表 23-2

值	意 义
LocalSystem	设置这个值可以指定服务在本地系统上使用有高度权限的用户账户, 但这个账户允许匿名用户进入网络, 因此它没有网络上的权限
LocalService	这个账户类型给任意远程服务器提供计算机的证书
NetworkService	类似于 LocalService, 这个值指定把计算机的证书传送给远程服务器, 但与 LocalService 不同, 这种服务可以以非授权用户的身份登录本地系统。顾名思义, 这个账户只能用于需从网络上获得资源的服务
User	把 Account 属性设置为 ServiceAccount.User, 表示可以指定应在服务中使用的账户

ServiceInstaller 是每一个服务都需要的类，这个类的属性可以用于进程中的每一个服务，其属性有 StartType、DisplayName、ServiceName 和 ServicesDependedOn，如表 23-3 所示。

表 23-3

属 性	说 明
StartType	StartType 指出服务是手动启动还是自动启动。它的值可以是：ServiceStartMode Automatic、ServiceStartMode Manual、ServiceStartMode Disabled。如果使用 ServiceStartMode Disabled，服务就不能启动。这个选项可用于不应在系统中启动的服务。例如，如果没有得到需要的硬件控制器，就可以把该选项设置为 Disabled
DisplayName	DisplayName 是服务显示给用户的友好名称。此外，这个名称也用于控制和监视服务的管理工具
ServiceName	ServiceName 是服务的名称。这个值必须与服务程序中 ServiceBase 类的 ServiceName 属性一致，这个名称与把 ServiceInstaller 配置为需要的服务程序相关
ServicesDependentOn	指定必须在服务启动之前启动的一个服务组。当服务启动时，所有相依存的服务都自动启动，并且我们的服务也将启动

提示：

如果在 ServiceBase 的派生类中改变了服务的名称，还必须修改 ServiceInstaller 对象中 Service Name 属性的值。

注意：

在测试阶段，最好把 StartType 的值设置为 Manual。如果服务因程序中的错误不能停止，仍可以重新启动系统。如果把 StartType 的值设置为 Automatic，服务就会在重新启动系统时自动启动！当确信没有问题时，可以在以后改变这个配置。

3. ServiceInstallerDialog 类

System.ServiceProcess.Design 命名空间中的另一个安装程序类是 ServiceInstaller Dialog。在安装过程中，如果希望系统管理员输入用户名和密码，就可以使用这个类。

如果 ServiceProcessInstaller 类的 Account 属性设置为 ServiceAccount.User，Username 和 Password 属性设置为 null，则在安装时，图 23-13 所示的 Set Service Login 对话框将自动显示出来。此时，也可以取消安装。

4. installutil

在把安装类添加到项目中之后，就可以使用 installutil.exe 实用程序安装和卸载服务了。这个实用程序可以用于安装包含 Installer 类的所有程序集。installutil.exe 实用程序调用派生于 Installer 类的方法 Installer()进行安装，调用 UnInstaller()方法进行卸载。

安装和卸载服务的命令分别是：

```
installutil quoteservice.exe
installutil /u quoteservice.exe
```

#### 注意:

如果安装失败了,一定要检查安装日志文件 `InstallUtil.InstallLog` 和 `<servicename>.InstallLog`。通常,在安装日志文件中可以发现一些非常有用的信息,例如:“指定的服务已存在”。

### 5. 客户程序

在成功地安装服务后,就可以从 Services MMC 中手动启动服务(详细内容请参阅 23.5 节),并启动客户应用程序,图 23-14 显示了访问服务的客户。

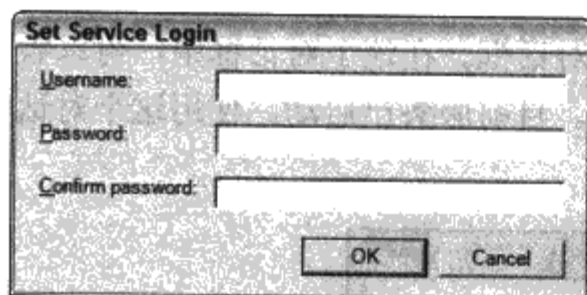


图 23-13

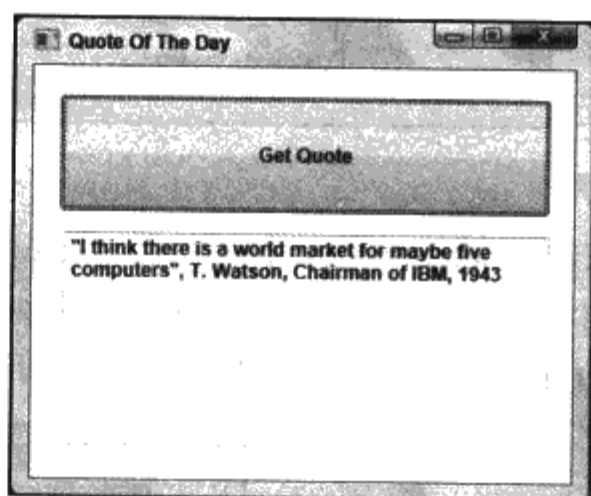


图 23-14

## 23.5 服务的监视和控制

可以使用 Services MMC 管理单元对服务进行监视和控制。Services MMC 管理单元是 Computer Management 管理工具的一部分。在每个 Windows 操作系统上,还有一个命令行实用程序 `net.exe`,使用这个程序可以控制服务。`sc.exe` 是另一个命令行实用程序,它的功能比 `net.exe` 更强大,是 Platform SDK 的一部分。本节将创建一个小的 Windows 应用程序,利用 `System.ServiceProcess.ServiceController` 类监视和控制服务。

### 23.5.1 MMC 计算机管理

使用 Microsoft Management Console (MMC) 的 Services 管理单元,可以查看所有服务的状态,也可以把停止、启用和禁用服务的控制请求发送给服务,并改变它们的配置。Services 管理单元既是服务控制程序,又是服务配置程序,如图 23-15 所示。



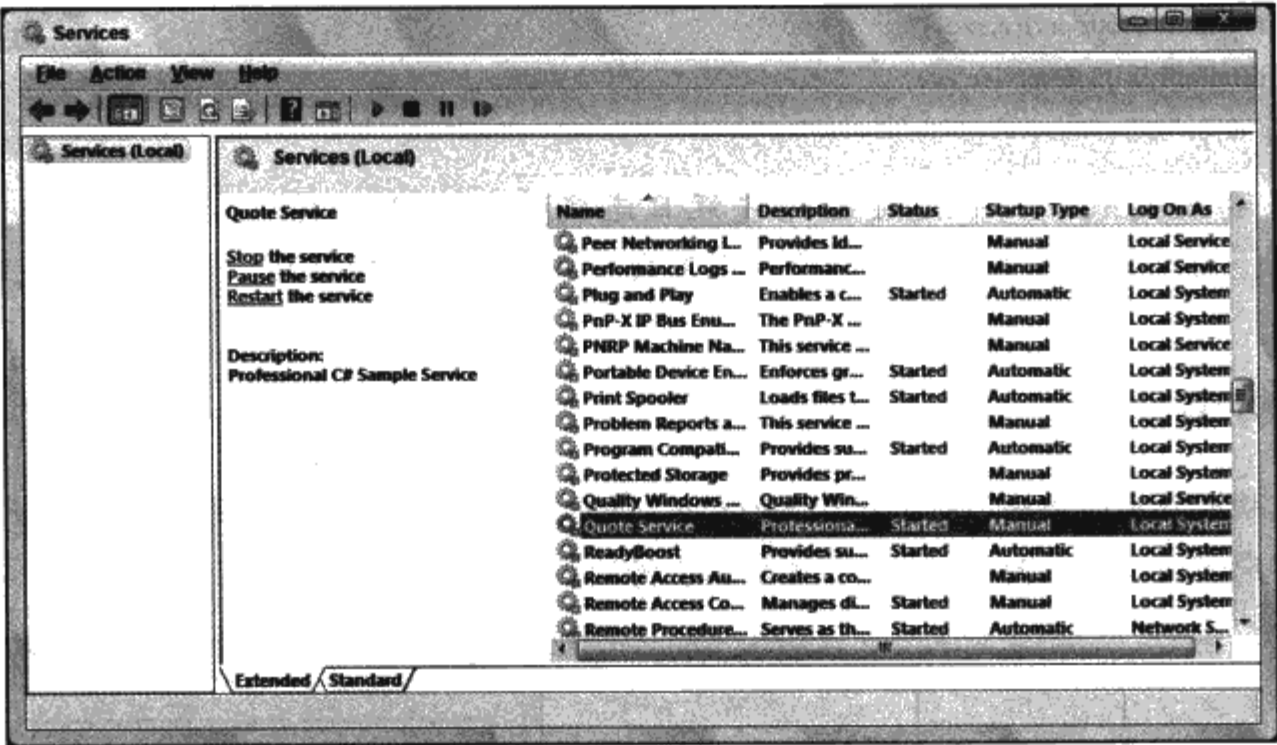


图 23-15

双击 QuoteService，打开如图 23-16 所示的属性对话框。在这个对话框中，可以看到服务的名称、描述、可执行文件的路径、启动类型和状态。目前服务已启动。使用这个对话框中的 Log On 选项卡，可以改变服务进程的账户。

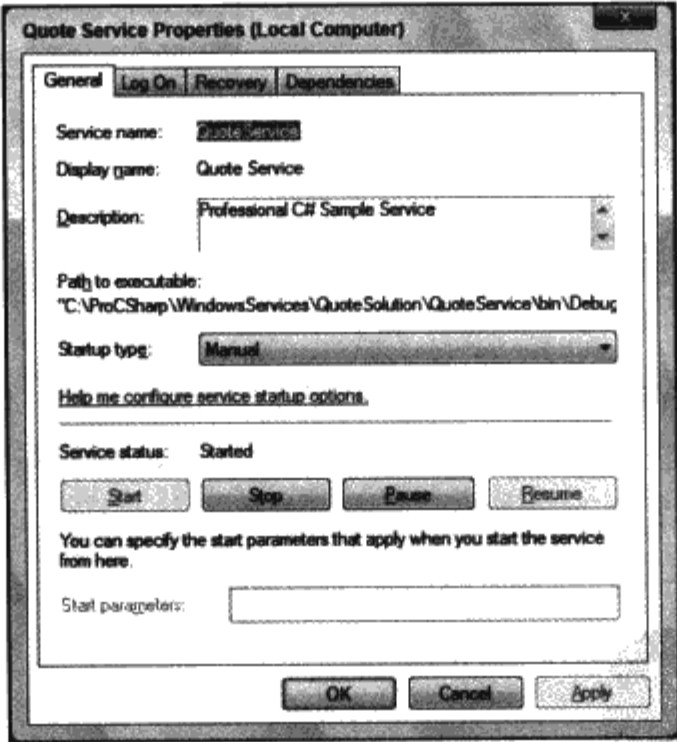


图 23-16

23.5.2 net.exe

Services 管理单元使用起来很简单，但是系统管理员不能使其自动化，原因是它不能用在管理脚本中。要控制服务，可以用命令行实用程序 net.exe 来完成。net start 显示所有正在运行的服务，net start <servicename> 启动服务，net stop <servicename> 向服务发送停止请求。此外使用 net pause 和 net continue 可以暂停和继续服务(当然，它们只有在服务允许的情况下才能使用)。

图 23-17 所示的控制台窗口显示了 net start 的结果。

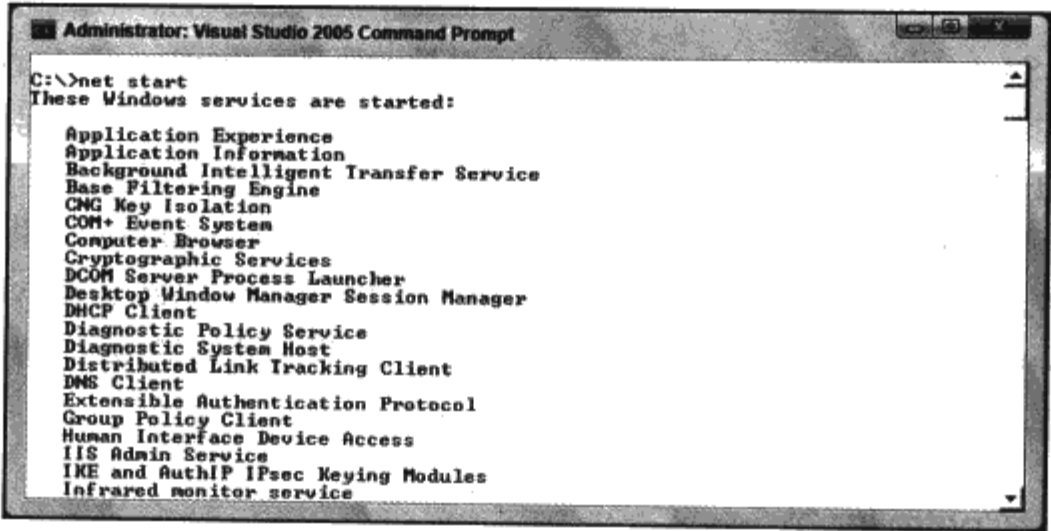


图 23-17

23.5.3 sc.exe

sc.exe 是不太出名的一个实用程序，它是操作系统的一部分。

sc.exe 是管理服务的一个很有用的工具。与 net.exe 相比，sc.exe 的功能更加强大。使用 sc.exe，可以检查服务的实际状态，配置、删除以及添加服务。当服务的卸载程序不能正常工作时，可以使用 sc.exe 卸载服务，如图 23-18 所示。

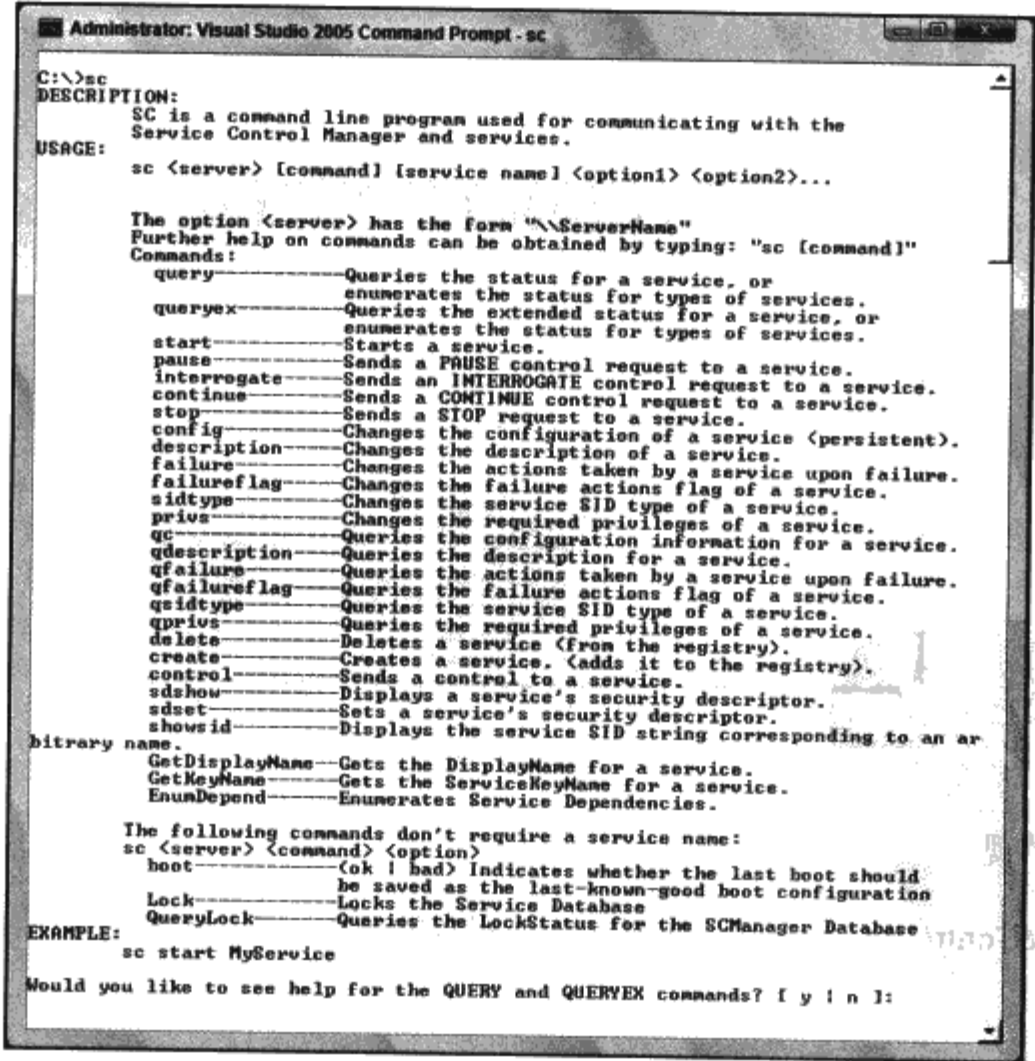


图 23-18

23.5.4 Visual Studio Server Explorer

使用 Visual Studio 中的 Server Explorer，也可以控制服务；Services 在 Servers 和计算机的名称的下面。选择一个服务，打开弹出菜单，就可以启动和停止服务。这个弹出菜单也可以用于把 ServiceController 类添加到项目中。如果要控制应用程序中的具体服务，则可以把服务从 Server Explorer 拖放到设计器上：即把 ServiceController 实例添加到应用程序中，ServiceController 对象的属性自动设置为访问选中的服务，引用程序集 System.ServiceProcess.dll。使用 ServiceController 实例控制服务的方式与使用下一节中开发的应用程序来控制服务的方式相同。

23.5.5 ServiceController 类

下面创建一个小的 Windows 应用程序，该应用程序使用 ServiceController 类监视和控制 Windows 服务。

创建一个 WPF 应用程序，这个应用程序的用户界面包含一个显示所有服务的列表框、4 个文本框(分别用于显示服务的显示名称、状态、类型和名称)和 6 个按钮，其中 4 个按钮用于发送控制事件，一个按钮用于刷新列表，一个按钮用于退出应用程序，如图 23-19 所示。

提示：  
WPF 详见第 34 章。

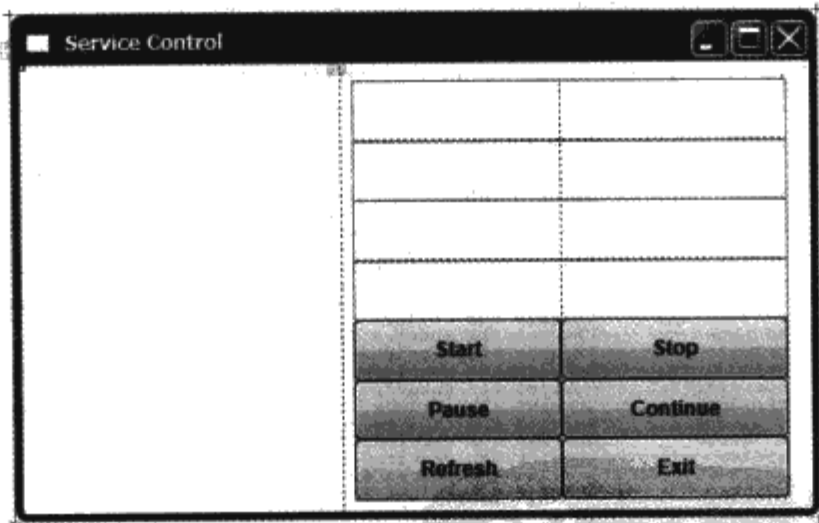


图 23-19

1. 服务的监视

使用 ServiceController 类，可以获取每一个服务的信息。表 23-4 列出了 ServiceController 类的属性。

表 23-4

属 性	描 述
CanPauseAndContinue	如果暂停和继续服务的请求可以发送给服务，则这个属性返回 true
CanShutdown	如果服务有系统关闭的处理程序，则它的值为 true
CanStop	如果服务是可以停止的，则它的值为 true
DependentServices	它返回一个依存服务的集合。如果服务停止，则所有依存的服务都预先停止
ServicesDependentOn	这个属性返回这个服务所依存的服务集合
DisplayName	这个属性返回服务应该显示的名称
MachineName	这个属性返回运行服务的机器名
ServiceName	指定服务的名称
ServiceType	指定服务的类型。服务可以运行在共享的进程中。在共享的进程中，多个服务使用同一进程(Win32ShareProcess)，此外，服务也可以运行在只包含一个服务的进程(Win32OwnProcess)中。如果服务可以与桌面交互，则类型就是 InteractiveProcess
Status	这个属性返回服务的状态。状态可以是正在运行、停止、暂停或某些中间模式(如启动待决、停止待决)等。状态值在 ServiceControllerStatus 枚举中定义

在示例应用程序中，使用 DisplayName、ServiceName、ServiceType 和 Status 属性显示服务信息。此外，CanPauseAndContinue 和 CanStop 用于启用和禁用 Pause、Continue 和 Stop 按钮。

为了得到用户界面的所有必要信息，创建一个 ServiceControllerInfo 类。这个类可以用于数据绑定，提供状态信息、服务名、服务类型，以及哪些控制服务的按钮应启用或禁用的信息。

提示：

因为使用了 System.ServiceProcess.ServiceController 类，所以必须引用 System.ServiceProcess 程序集。

ServiceControllerInfo 包含一个嵌入的 ServiceController，它用 ServiceControllerInfo 类的构造函数设置。还有一个只读的属性 Controller，来访问嵌入的 ServiceController。

```
public class ServiceControllerInfo
{
    private ServiceController controller;

    public ServiceControllerInfo(ServiceController controller)
    {
        this.controller = controller;
    }

    public ServiceController Controller
    {
        get { return controller; }
    }
}
```

为了显示服务的当前信息，可以使用 ServiceControllerInfo 类的只读属性 DisplayName、ServiceName、ServiceTypeName 和 ServiceStatusName。DisplayName 和 ServiceName 属性的执行代码只访问底层类 ServiceController 的 DisplayName 和 ServiceName 属性。在 ServiceTypeName

和 `ServiceStatusName` 属性的执行代码中，完成更多的工作：服务的状态和类型不太容易返回，因为要显示一个字符串，而不是只显示 `ServiceController` 类返回的数字。属性 `ServiceTypeName` 返回一个表示服务类型的字符串。从 `ServiceController.ServiceType` 属性中得到的 `ServiceType` 代表一组标记，使用按位 OR 运算符，可以把这组标记组合在一起。`InteractiveProcess` 位可以与 `Win32OwnProcess` 和 `Win32ShareProcess` 一起设置。首先，在检查其他的值之前，一定要先检查 `InteractiveProcess` 位以前是否设置过。使用该服务，返回的字符串是“Win 32 Service Process”或“Win 32 Shared Process”。

```
public string ServiceTypeName
{
    get
    {
        ServiceType type = controller.ServiceType;
        string serviceTypeName = "";
        if ((type & ServiceType.InteractiveProcess) != 0)
        {
            serviceTypeName = "Interactive ";
            type -= ServiceType.InteractiveProcess;
        }
        switch (type)
        {
            case ServiceType.Adapter:
                serviceTypeName += "Adapter";
                break;

            case ServiceType.FileSystemDriver:
            case ServiceType.KernelDriver:
            case ServiceType.RecognizerDriver:
                serviceTypeName += "Driver";
                break;

            case ServiceType.Win32OwnProcess:
                serviceTypeName += "Win32 Service Process";
                break;

            case ServiceType.Win32ShareProcess:
                serviceTypeName += "Win32 Shared Process";
                break;
            default:
                serviceTypeName += "unknown type " + type.ToString();
                break;
        }
        return serviceTypeName;
    }
}

public string ServiceStatusName
{
    get
    {
        switch (controller.Status)
        {
            case ServiceControllerStatus.ContinuePending:
                return "Continue Pending";
            case ServiceControllerStatus.Paused:
                return "Paused";
            case ServiceControllerStatus.PausePending:
                return "Pause Pending";
        }
    }
}
```



```

        case ServiceControllerStatus.StartPending:
            return "Start Pending";
        case ServiceControllerStatus.Running:
            return "Running";
        case ServiceControllerStatus.Stopped:
            return "Stopped";
        case ServiceControllerStatus.StopPending:
            return "Stop Pending";
        default:
            return "Unknown status";
    }
}

public string DisplayName
{
    get { return controller.DisplayName; }
}

public string ServiceName
{
    get { return controller.ServiceName; }
}

```

`ServiceControllerInfo` 类还有一些属性可以启用 `Start`、`Stop`、`Pause` 和 `Continue` 按钮：`EnableStart`、`EnableStop`、`EnablePause` 和 `EnableContinue`，这些属性根据服务的当前状态返回一个布尔值。

```

public bool EnableStart
{
    get
    {
        return controller.Status == ServiceControllerStatus.Stopped;
    }
}

public bool EnableStop
{
    get
    {
        return controller.Status == ServiceControllerStatus.Running;
    }
}

public bool EnablePause
{
    get
    {
        return controller.Status == ServiceControllerStatus.Running &&
            controller.CanPauseAndContinue;
    }
}

public bool EnableContinue
{
    get
    {
        return controller.Status == ServiceControllerStatus.Paused;
    }
}
}

```

在 `ServiceControlWindow` 类中, `RefreshServiceList()`方法使用 `ServiceController.Get Services()` 获取在列表框中显示的所有服务。`GetServices()`方法返回一个 `ServiceController` 实例数组, 它们表示在操作系统上安装的所有 Windows 服务。`ServiceController` 类还有一个静态方法 `GetDevices()`, 它返回一个表示所有设备驱动程序的 `ServiceController` 数组。返回的数组利用泛型方法 `Array.Sort()` 按照 `DisplayName` 来排序, 这是传送给 `Sort()`方法的匿名方法定义的。使用 `Array.ConvertAll()`, 将 `ServiceController` 实例转换为 `ServiceControllerInfo` 类型。这里传送了一个匿名方法, 它调用每个 `ServiceController` 对象的 `ServiceControllerInfo` 构造函数。最后, 将 `ServiceControllerInfo` 数组赋予窗口的 `DataContext` 属性, 进行数据绑定。

```
protected void RefreshServiceList()
{
    ServiceController[] services = ServiceController.GetServices();

    Array.Sort<ServiceController>(services,
        delegate(ServiceController s1, ServiceController s2)
        {
            return s1.DisplayName.CompareTo(s2.DisplayName);
        });
    ServiceControllerInfo[] serviceInfo =
        Array.ConvertAll<ServiceController, ServiceControllerInfo>(
            services,
            delegate(ServiceController controller)
            {
                return new ServiceControllerInfo(controller);
            });
    this.DataContext = serviceInfo;
}
```

在列表框中获得所有服务的 `RefreshServiceList()`方法在 `ServiceControlWindow` 类的构造函数中调用。这个构造函数还为按钮的 `Click` 事件定义了事件处理程序:

```
public ServiceControlWindow()
{
    InitializeComponent();

    buttonStart.Click += OnServiceCommand;
    buttonStop.Click += OnServiceCommand;
    buttonPause.Click += OnServiceCommand;
    buttonContinue.Click += OnServiceCommand;
    buttonRefresh.Click += OnRefresh;
    buttonExit.Click += OnExit;

    RefreshServiceList();
}
```

现在, 就可以定义 XAML 代码, 把信息绑定到控件上了。

首先, 为显示在列表框中的信息定义一个 `DataTemplate`。列表框包含一个标签, 其 `Content` 属性绑定到数据源的 `DisplayName` 属性上。在绑定 `ServiceControllerInfo` 对象数组时, 用 `ServiceControllerInfo` 类定义 `DisplayName` 属性:

```
<Window.Resources>
  <DataTemplate x:Key="listTemplate">
    <Label Content="{Binding Path=DisplayName}"/>
  </DataTemplate>
</Window.Resources>
```

放在窗口左边的列表框将 `ItemsSource` 属性设置为 `{Binding}`。这样，显示在列表中的数据就从 `RefreshServiceList()`方法设置的 `DataContext` 属性中获得。`ItemTemplate` 属性引用了前面用 `DataTemplate` 定义的资源 `listTemplate`。`IsSynchronizedWithCurrentItem` 属性设置为 `True`，所以位于同一个窗口中的文本框和按钮控件就绑定到列表框中当前选择的项上。

```
<ListBox Grid.Row="0" Grid.Column="0" HorizontalAlignment="Left"
  Name="listBoxServices" VerticalAlignment="Top"
  ItemsSource="{Binding}"
  ItemTemplate="{StaticResource listTemplate}"
  IsSynchronizedWithCurrentItem="True">
</ListBox>
```

对于文本框控件，`Text` 属性绑定到 `ServiceControllerInfo` 实例的对应属性上。按钮控件是启用还是禁用也在数据绑定中定义，即把 `IsEnabled` 属性绑定到 `ServiceControllerInfo` 实例的对应属性上，该属性返回一个布尔值：

```
<TextBox Grid.Row="0" Grid.ColumnSpan="2" Name="textDisplayName"
  Text="{Binding Path=DisplayName, Mode=OneTime}" />
<TextBox Grid.Row="1" Grid.ColumnSpan="2" Name="textStatus"
  Text="{Binding Path=ServiceStatusName, Mode=OneTime}" />
<TextBox Grid.Row="2" Grid.ColumnSpan="2" Name="textType"
  Text="{Binding Path=ServiceTypeName, Mode=OneTime}" />
<TextBox Grid.Row="3" Grid.ColumnSpan="2" Name="textName"
  Text="{Binding Path=ServiceName, Mode=OneTime}" />
<Button Grid.Row="4" Grid.Column="0" Name="buttonStart" Content="Start"
  IsEnabled="{Binding Path=EnableStart, Mode=OneTime}" />
<Button Grid.Row="4" Grid.Column="1" Name="buttonStop" Content="Stop"
  IsEnabled="{Binding Path=EnableStop, Mode=OneTime}" />
<Button Grid.Row="5" Grid.Column="0" Name="buttonPause" Content="Pause"
  IsEnabled="{Binding Path=EnablePause, Mode=OneTime}" />
<Button Grid.Row="5" Grid.Column="1" Name="buttonContinue" Content="Continue"
  IsEnabled="{Binding Path=EnableContinue, Mode=OneTime}" />
```

2. 服务的控制

使用 `ServiceController` 类，也可以把控制请求发送给服务，该类的方法如表 23-5 所示。

表 23-5

方 法	说 明
Start()	Start() 告诉 SCM 应启动服务。在我们的服务程序中，调用了 OnStart()
Stop()	如果 CanStop 属性在服务类中的值是 true，则在 SCM 的帮助下，Stop()调用服务程序中的 OnStop()
Pause()	如果 CanPauseAndContinue 属性的值是 true，则 Pause() 调用 OnPause()
Continue()	如果 CanPauseAndContinue 属性的值是 true，则 Continue 调用 OnContinue()
ExecuteCommand()	使用 ExecuteCommand()可以把定制的命令发送给服务



下面就是控制服务的代码。因为启动、停止、挂起和暂停服务的代码是相似的，所以仅为这4个按钮使用一个处理程序：

```
protected void OnServiceCommand(object sender, RoutedEventArgs e)
{
    Cursor oldCursor = Cursor.Current;
    Cursor.Current = Cursors.Wait;
    ServiceControllerInfo si =
        (ServiceControllerInfo)listBoxServices.SelectedItem;
    if (sender == this.buttonStart)
    {
        si.Controller.Start();
        si.Controller.WaitForStatus(ServiceControllerStatus.Running);
    }
    else if (sender == this.buttonStop)
    {
        si.Controller.Stop();
        si.Controller.WaitForStatus(ServiceControllerStatus.Stopped);
    }
    else if (sender == this.buttonPause)
    {
        si.Controller.Pause();
        si.Controller.WaitForStatus(ServiceControllerStatus.Paused);
    }
    else if (sender == this.buttonContinue)
    {
        si.Controller.Continue();
        si.Controller.WaitForStatus(ServiceControllerStatus.Running);
    }

    int index = listBoxServices.SelectedIndex;
    RefreshServiceList();
    listBoxServices.SelectedIndex = index;
    Cursor.Current = oldCursor;
}

protected void OnExit(object sender, RoutedEventArgs e)
{
    Application.Current.Shutdown();
}

protected void OnRefresh_Click(object sender, RoutedEventArgs e)
{
    RefreshServiceList();
}
```

由于控制服务要花费一定的时间，因此，光标在第一个语句中切换为等待光标。然后，根据被按的按钮调用 `ServiceController` 方法。使用 `WaitForStatus()` 方法，等待服务把状态改为被请求的值，但是，我们最多等待 10 秒。在 10 秒之后，就会刷新列表框中的信息，其目的是当用户选择与以前相同的服务时，服务的新状态能够显示出来。最后运行的应用程序结果如图 23-20 所示。



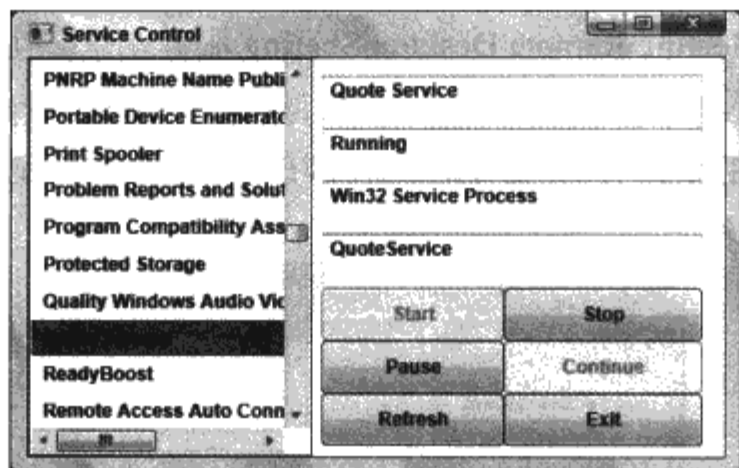


图 23-20

## 23.6 故障排除

服务方面的故障排除与标准应用程序的故障排除并不相同。本节将讨论一些服务问题、交互式服务的问题和事件日志。

创建服务的最好方式就是在创建服务之前，先创建一个具有所需功能的程序集和一个测试客户程序，以便进行正常的调试和错误处理。应用程序一运行，就可以使用那个程序集创建服务。当然，对于服务来说，仍然存在下列问题：

- 在服务中，错误信息不显示在消息框中(除了运行在客户系统上的交互式服务之外)，而是使用事件日志服务把错误写入事件日志。当然，在使用服务的客户应用程序中，可以显示一个消息框，通知用户出现了错误。
- 服务不能从调试器中启动，但是调试器可以与正在运行的服务联系起来。打开带有服务源代码的解决方案，并且设置断点。从 Visual Studio 的 Debug 菜单中选择 Processes 命令，捕获正在运行的服务进程。
- 性能监视器可以用于监视服务的行为。可以把自己的性能对象添加给服务，这样可以添加一些有用的信息，以便进行调试。例如在 Quote 服务中，可以建立一个对象，让它给出返回的引用总数和初始化花费的时间等。

### 23.6.1 交互式服务

如果交互式服务运行在已登录的用户账户下，把消息框显示给用户是非常有用的。如果服务运行在锁在计算机机房中的服务器上，就不用显示消息框。在打开消息框，等待用户的输入时，由于计算机房间中没有人理会服务器，因此用户的输入许多天都不会发生；更糟糕的是，如果服务没有配置为交互式的服务，则消息框就会在另一个隐藏的窗口中打开，这样，没有人可以响应那个隐藏的消息框，因此，服务将一直处于停滞状态。

**注意：**

不要为运行在服务器系统上的服务打开对话框，因为没有人响应这个对话框。

如果要与用户交互，可以配置一个交互式的服务。例如，用于向用户显示纸张输出信息的



Print Spooler 服务和 NetMeeting Remote Desktop Sharing 服务就是交互式的服务。

在配置交互式服务时，必须设置 Services 配置工具中的 Allow service to interact with desktop 选项，如图 23-21 所示。把 SERVICE\_INTERACTIVE\_PROCESS 标记添加给类型，可以改变服务的类型。

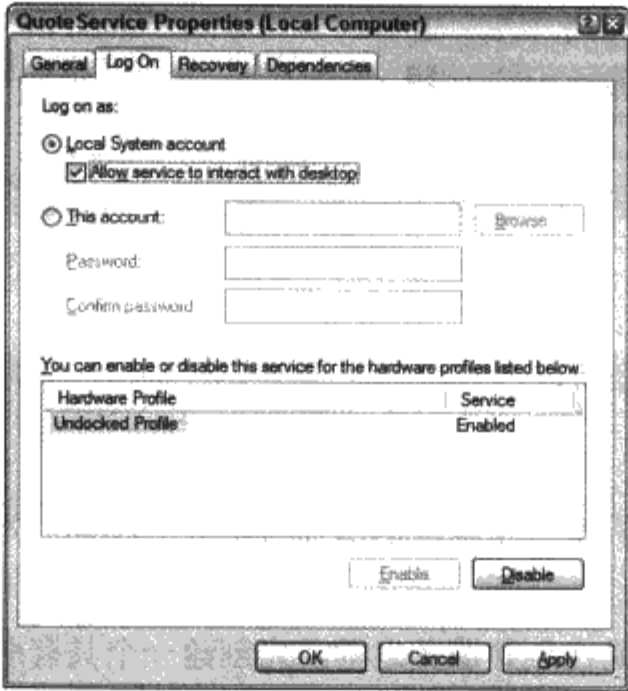


图 23-21

23.6.2 事件日志

把事件添加到事件日志中，服务就可以报告错误和其他信息。当 AutoLog 属性设置为 true 时，从 ServiceBase 中派生出来的服务类可以自动把事件写入到日志中。ServiceBase 类检查 AutoLog 属性，并且在启动、停止、暂停和继续请求时编写日志条目。

图 23-22 是服务中的日志条目实例。

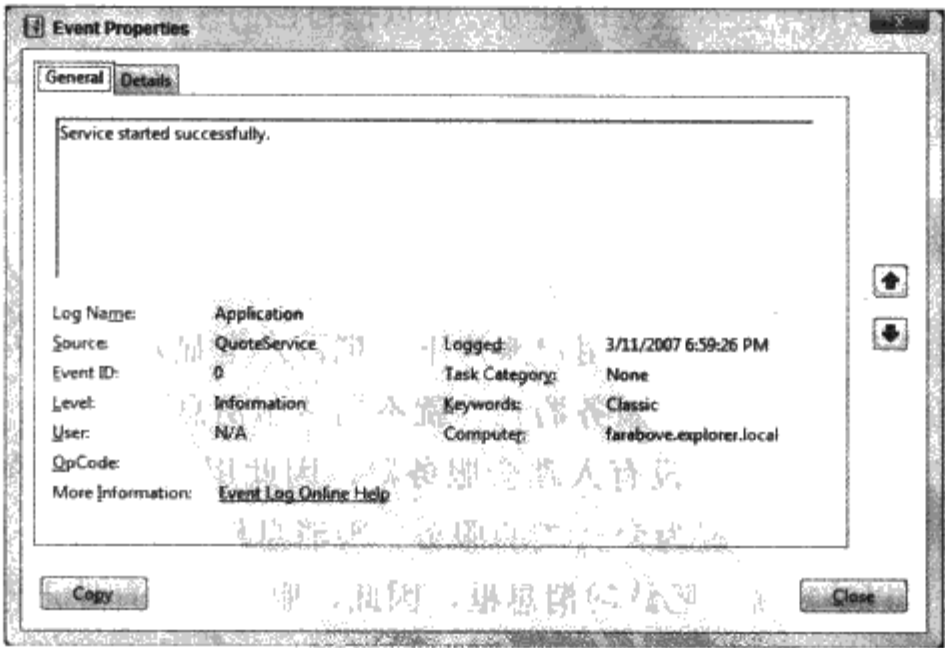


图 23-22

提示：  
事件日志和编写定制事件的内容详见第 18 章。

## 23.7 电源事件

Windows 服务可以响应电源状态的变化了。例如，系统可以休眠，在系统休眠时，内存中的内容保存到硬盘上，这样，系统的启动速度就会比较快。此外，为了减少电源的消耗，Windows 服务还支持系统等待，但是，在需要时系统可以自动唤醒。

对于所有的电源事件而言，服务可以接收带有附加参数的控制代码 `SERVICE_CONTROL_POWEREVENT`。在参数中可以找到事件的原因，事件的原因可以是电池没电了、系统正处于等待状态或电源状态改变了等。根据事件的原因，服务将作出不同的反应，例如服务减慢、等待背景线程、关闭网络连接和关闭文件等。

`System.ServiceProcess` 命名空间中的类也支持电源事件。使用 `CanPauseAndContinue` 属性，可以把服务配置为响应暂停事件和继续事件。也可以为电源管理设置 `CanHandlePowerEvent` 属性。使用 Win32 API 方法 `RegisterServiceCtrlHandlerEx()`，可以在 SCM 中注册处理电源事件的 Windows 服务。

如果 `CanHandlePowerEvent` 属性的值为 `true`，就会调用 `ServiceBase` 类的 `OnPowerEvent()` 方法。可以重写这个方法，接收电源事件，根据服务的执行情况进行响应。电源事件的原因在 `PowerBroadcastStatus` 类型的参数中传送。这个枚举的值如表 23-6 所示。

表 23-6

PowerStatus 的值	描 述
BatteryLow	电池没有电了，应该把服务的功能降低到最小的程度
PowerStatusChange	电池电源的开关 A/C 进行了切换，或者电池电源低于一定的阈值等
QuerySuspend	系统请求许可进入挂起模式。可以拒绝这样的许可，也可以为进入挂起模式做一些准备工作，例如关闭文件和断开网络连接等
QuerySuspendFailed	把系统变为挂起模式的请求被拒绝，可以继续以前的工作
Suspend	没有人拒绝请求进入挂起模式。系统不久将处于挂起模式下

## 23.8 小结

本章讨论了 Windows 服务的体系结构和如何使用 .NET Framework 创建 Windows 服务。应用程序可以与 Windows 服务一起在系统启动时自动启动，也可以把具有特权的 `System` 账户用作服务的用户。Windows 服务从主函数、`service-main` 函数和处理程序中创建，本章还介绍了与 Windows 服务相关的其他程序，例如服务控制程序和服务安装程序。

.NET Framework 对 Windows 服务提供了很好的支持。创建、控制和安装服务所需的代码都封装在 `System.ServiceProcess` 命名空间的 .NET Framework 类中。从 `ServiceBase` 类中派生一个类，就可以重写服务暂停、恢复或停止时调用的方法。对于服务的安装，类 `ServiceProcessInstaller` 和 `ServiceInstaller` 可以处理服务所需的所有注册表配置。还可以使用 `ServiceController` 控制和监视服务。

下一章介绍与本机代码的交互操作。许多 .NET 类在后台使用本机代码。例如 `ServiceBase` 类封装了 Windows API `CreateService()`。下一章还会探讨如何在自己的类中使用本机方法和 COM 对象。

# 第24章

## 互操作性

如果您在学习.NET之前编写过 Windows 程序,通常没有时间和资源用.NET再重新编写以前的程序。有时重写代码有助于做一些修订,重新思考应用程序的体系架构,从长远来看,还有助于提高效率,更便于用新技术添加新特性。但是,我们不会为使用一种新技术而重写已有的代码。我们本来有数千行可运行的代码,重写它们需要的精力太多,还不如把它们迁移到托管的环境中。

这也同样适用于 Microsoft。在命名空间 `System.DirectoryService` 中,Microsoft 并没有重新编写 COM 对象来访问有层次的数据存储,这个命名空间中的类实际上是访问 ADSI COM 对象的包装器。`System.Data.OleDb` 命名空间也是这样,由这个命名空间中的类所使用的 OLE DB 提供程序包含相当复杂的 COM 接口。

我们自己的解决方案也会面临相同的问题。如果在.NET应用程序中要使用已有的 COM 对象,或者要编写在旧 COM 客户程序中使用的.NET组件,就应使用本章介绍的 COM 互操作性。

如果没有要与应用程序集成的 COM 组件,或旧 COM 客户程序要使用一些.NET组件,就应跳过本章。

本章主要内容如下:

- COM 和.NET 技术
- 在.NET应用程序中使用 COM 对象
- 在 COM 客户程序中使用.NET组件
- 调用本地方法的 Platform Invoke(平台调用)

与其他章节一样,本章的示例代码也可以从 Wrox 网站 [www.wrox.com](http://www.wrox.com) 上下载。

### 24.1 .NET 和 COM

COM 是.NET以前的技术。COM 定义了一个组件模型,在该模型中,组件可以用不同的编程语言编写。用 C++编写的组件可以在 VB 客户程序中使用。组件还可以在本地的进程中使用,跨进程使用或在网络上使用。看起来是不是很熟悉?当然,.NET 的目标也是这样。但这些目标的实现方式是不同的。COM 概念使用起来越来越复杂,已经不能扩展了。.NET 达到了与 COM 类似的目标,但引入了新概念,实现起来更容易。

即使到了今天,使用 COM 和.NET交互操作的主要问题是要理解 COM。是 COM 客户程序使用.NET组件,还是.NET应用程序使用 COM 组件并不重要,而是必须理解 COM。所以这里首先比较 COM 和.NET。

如果您已经熟练掌握了 COM 技术, 本节将是 COM 知识的复习。否则, 您将学习到 COM 的概念——现在是使用 .NET——我们不再需要在日常事务中处理它了。但是, 在把 COM 技术集成到 .NET 应用程序中时, COM 的问题仍旧存在。

COM 和 .NET 有许多类似的概念和不同的解决方案。下面将讨论:

- 元数据
- 释放内存
- 接口
- 方法绑定
- 数据类型
- 注册
- 线程
- 错误处理
- 事件处理

### 24.1.1 元数据

在 COM 中, 组件的所有信息都存储在类型库中。类型库包含的信息有接口、方法和参数的名称和 ID 等。而在 .NET 中, 所有这些信息都可以存储在程序集中, 如第 13 章和第 17 章所述。COM 存在的问题是, 类型库是不能扩展的。在 C++ 中, IDL(接口定义语言)文件用于描述接口和方法。其中一些 IDL 修饰符不在类型库中, 因为 Visual Basic(和负责开发类型库的 Visual Basic 小组)不能使用这些 IDL 修饰符。而在 .NET 中, 不存在这个问题, 因为 .NET 元数据可以使用定制特性来扩展。

因此, 一些 COM 组件有类型库, 而其他 COM 组件没有。如果没有类型库可用, 就可以使用 C++ 头文件来描述接口和方法。在 .NET 中, 使用带有类型库的 COM 组件是比较容易的, 也可使用不带类型库的 COM 组件。在这种情况下, 必须使用 C# 代码重新定义 COM 接口。

### 24.1.2 释放内存

在 .NET 中, 内存的释放是由垃圾收集器完成的。这完全不同于 COM, COM 依赖的是引用数。

接口 `IUnknown` 是每个 COM 对象必须实现的一个接口, 它提供了 3 个方法。其中两个方法与引用数有关。如果需要另一个接口指针, 客户程序就必须调用方法 `AddRef()`, 这个方法会递增引用数。方法 `Release()` 会递减引用数, 如果所得的引用数是 0, 就销毁对象, 释放内存。

### 24.1.3 接口

接口是 COM 的核心, 它区分了在客户程序和对象之间使用的契约和实现方式。接口(契约)定义了由组件提供的方法, 可以由客户程序使用。而在 .NET 中, 接口也有非常重要的作用。

COM 有 3 种接口类型: 定制接口、分派接口(dispatch interface)和双重接口。

1. 定制接口

定制接口派生自接口 IUnknown。定制接口定义了虚拟表(vtable)中的方法顺序，所以客户程序可以直接访问接口的方法。这也表示在开发阶段客户程序需要知道虚拟表，因为方法的绑定是使用内存地址进行的。因此，定制接口不能由脚本客户程序使用。图 24-1 显示了定制接口 IMath 的虚拟表，除了接口 IUnknown 的方法之外，该接口还提供了方法 Add()和 Sub()。

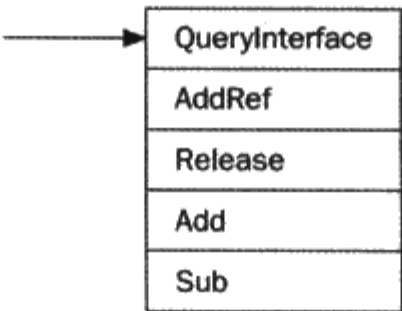


图 24-1

2. 分派接口

因为脚本客户程序(和早期的 Visual Basic 客户程序)不支持定制接口，所以需要另外一种接口类型，而在分派接口中，可用于客户程序的接口总是 IDispatch 接口。IDispatch 接口派生自 IUnknown 接口，除了接口 IUnknown 的方法之外，它还提供了 4 个方法，其中两个比较重要的方法是 GetIDsOfNames()和 Invoke()。如图 24-2 所示，在分派接口中需要两个表。第一个表把方法或特性名映射到分派 ID(dispatch id)，第二个表把分派 ID 映射到方法或特性的实现代码。

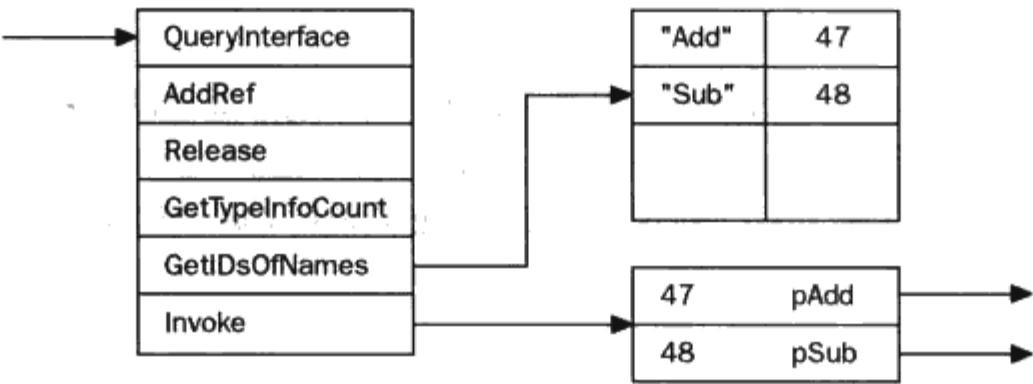


图 24-2

在客户程序调用组件中的方法时，要先调用方法 GetIDsOfNames()，并给它传送要调用的方法名。方法 GetIDsOfNames()会查找名称-ID 表，返回分派 ID，客户程序再使用这个 ID 调用方法 Invoke()。

注意：

通常，IDispatch 接口的两个表存储在类型库中，但这不是必需的，一些组件把这两个表存储在其他地方。

3. 双重接口

可以想像，分派接口比定制接口慢得多。另一方面，脚本客户程序不能使用定制接口。双



重接口可以解决这个问题。如图 24-3 所示，双重接口派生自 IDispatch 接口，但提供了可以在虚拟表中直接使用的接口方法。脚本客户程序可以使用 IDispatch 接口调用方法，而了解虚拟表的客户程序可以直接调用方法。

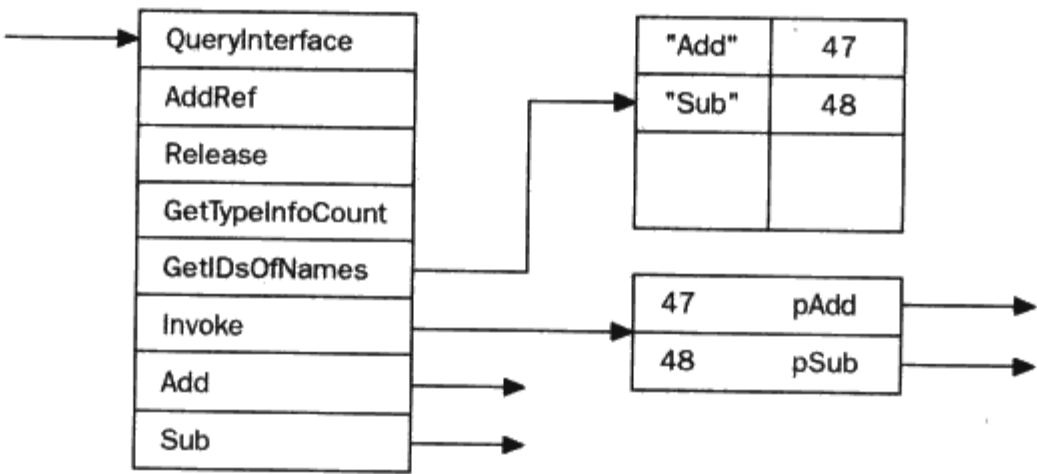


图 24-3

4. 强制类型转换和 QueryInterface

如果.NET 类实现多个接口，就可以进行强制类型转换，得到一个接口或另一个接口。而在 COM 中，接口 IUnknown 通过方法 `QueryInterface()` 提供了类似的机制。如上一节所述，接口 IUnknown 是其他接口的基础接口，所以可以以任何方式使用方法 `QueryInterface()`。

24.1.4 方法绑定

客户程序映射方法的方式是用早期绑定和后期绑定来定义的。后期绑定表示要调用的方法是在运行期间确定的。.NET 使用 `System.Reflection` 命名空间来实现后期绑定(参阅第 13 章)。

COM 使用上面讨论的 IDispatch 接口进行后期绑定。后期绑定可以使用分派接口和双重接口来实现。

在 COM 中，早期绑定有两个不同的选项。早期绑定的一种方式也称为虚拟表绑定，它直接使用虚拟表，可以通过定制接口和双重接口来实现。早期绑定的第二种方式也称为 id 绑定。其中分派 id 存储在客户程序代码中，在运行期间只需要调用一次 `Invoke()`。`GetIDsOfNames()` 方法在设计期间调用。对于这种客户程序，记住不必改变分派 id 是非常重要的。

24.1.5 数据类型

对于双重接口和分派接口，COM 能使用的数据类型被局限于一个自动兼容的数据类型列表。接口 IDispatch 的 `Invoke()` 方法接收 VARIANT 数据类型的数组。VARIANT 是许多不同数据类型的组合，例如 `BYTE`、`SHORT`、`LONG`、`FLOAT`、`DOUBLE`、`BSTR`、`IUnknown*`、`IDispatch*` 等。VARIANT 在 Visual Basic 中很容易使用，但在 C++ 中使用时就比较复杂了。在.NET 中，使用 `Object` 类代替了 VARIANT。

在定制接口中，C++ 能使用的所有数据类型也可用于 COM。但是，使用这个组件的客户程序只能采用某些语言来编程。

### 24.1.6 注册

.NET 区分了私有程序集和共享程序集，详见第 17 章。而在 COM 中，所有的组件都进行了注册配置，是全局可用的。

所有的 COM 对象都有一个唯一的标识符，该标识符由一个 128 位的数字组成，也称为类 ID(CLSID)。创建 COM 对象时，COM API 调用 `CoCreateInstance()` 会在注册表中查找 CLSID 和 DLL 或 EXE 的路径，然后加载 DLL 或启动 EXE，并实例化组件。

这个 128 位数字不容易记忆，所以许多 COM 对象还有一个 prog id，该 id 很容易记忆，例如 `Excel.Application` 就映射到一个 CLSID。

除了 CLSID 之外，COM 对象还为每个接口和类型库指定了一个唯一的标识符(IID 和 typelib id)。

本章的后面将详细讨论注册表中的信息。

### 24.1.7 线程

COM 使用了单元模型，这样程序员就不必考虑线程问题了。但是，这也增加了复杂性。不同的操作系统版本添加了不同的单元类型。本节讨论单线程单元和多线程单元。

注意：

.NET 中的线程详见第 19 章。

#### 1. 单线程单元

单线程单元(STA)是在 Windows NT 3.51 中引入的。在 STA 中，只允许一个线程(创建实例的线程)访问组件。但是，在一个进程中允许有多个单线程单元，如图 24-4 所示。

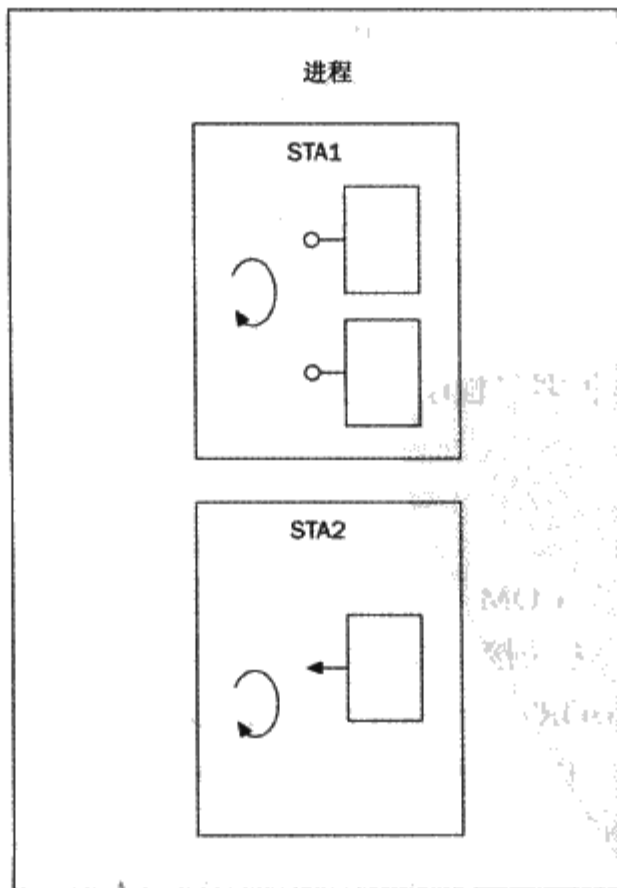


图 24-4

在图 24-4 中，里面带棒棒糖的矩形表示 COM 组件。组件和线程(弯曲箭头)包含在单元中。外部的矩形表示进程。

在 STA 中，不需要考虑多个线程访问实例变量的问题，因为这种保护是由 COM 特性实现的，只有一个线程可以访问组件。

COM 对象在编程时不是线程安全的，因此 STA 需要在注册表中把注册键 ThreadingModel 设置为 Apartment。

2. 多线程单元

Windows NT 4.0 引入了多线程单元(MTA)的概念。在 MTA 中，多个线程可以同时访问组件。图 24-5 显示了带一个 MTA 和两个 STA 的进程。

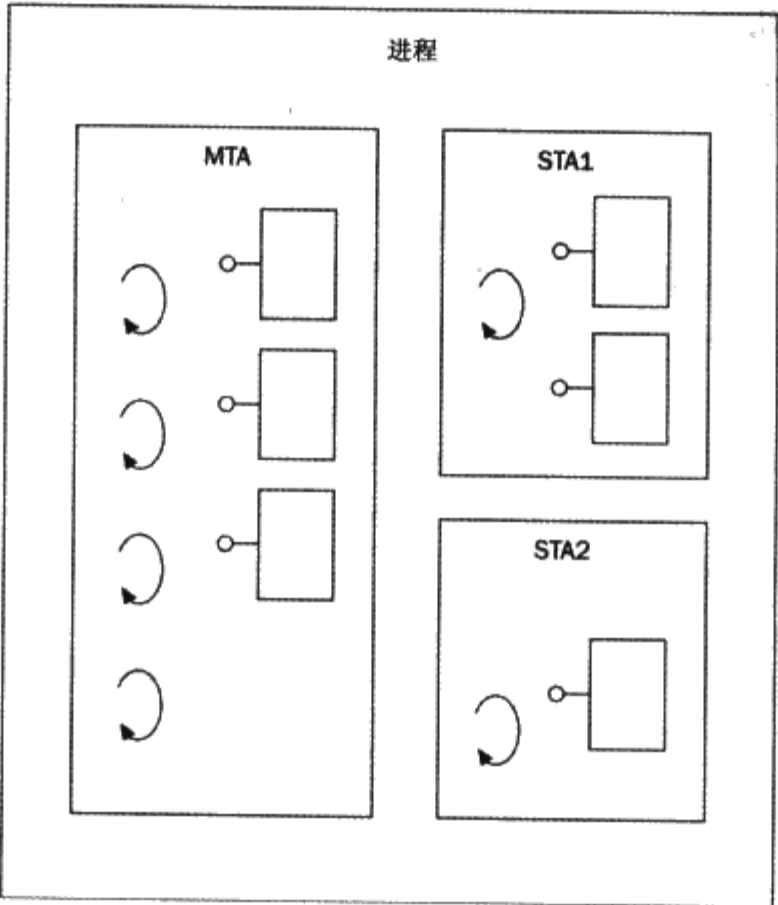


图 24-5

COM 对象在编程时是线程安全的，因此 MTA 需要在注册表中把键 ThreadingModel 设置为 Free。值 Both 用于不考虑单元类型的线程安全的 COM 对象。

注意：

Vasual Basic 6.0 不支持多线程单元。如果使用以 VB6 开发的 COM 对象，就必须了解这一点。

24.1.8 错误处理

在.NET 中，错误是通过抛出异常来生成的。在旧 COM 技术中，错误是通过方法返回 HRESULT 值来定义的。HRESULT 的值是 S\_OK，表示方法成功。

如果 COM 组件提供了详细的错误消息，COM 组件就实现接口 `ISupportErrorInfo`，该接口不但提供了错误消息，还提供了帮助文件的链接、错误源，在方法返回时还会返回一个错误信息对象。在.NET 中，实现接口 `ISupportErrorInfo` 的对象会自动映射到详细的错误信息和一个.NET 异常。

提示：  
跟踪和记录错误的内容详见第 18 章。

24.1.9 事件处理

.NET 用 C#关键字 `event` 和 `delegate` 提供了事件处理机制(参阅第 7 章)。

图 24-6 显示了 COM 的事件处理结构。在 COM 事件中，组件必须实现接口 `IConnectionPointContainer` 和另一个实现接口 `IConnectionPoint` 的连接点对象(CPO)。在图 24-6 中，组件还定义了一个由 CPO 调用的输出接口 `ICompletedEvents`。客户程序必须在 sink 对象中实现这个输出接口，而 sink 对象本身是一个 COM 对象。在执行过程中，客户程序在服务器中查询接口 `IConnectionPointContainer`。通过这个接口，客户程序让 CPO 执行方法 `FindConnectionPoint()`，获得指向所返回的 `IConnectionPoint` 的指针。客户程序再使用这个接口指针调用 `Advise()`方法，并把指向 sink 对象的指针传送给服务器。接着，组件就可以在客户程序的 sink 对象中调用方法了。

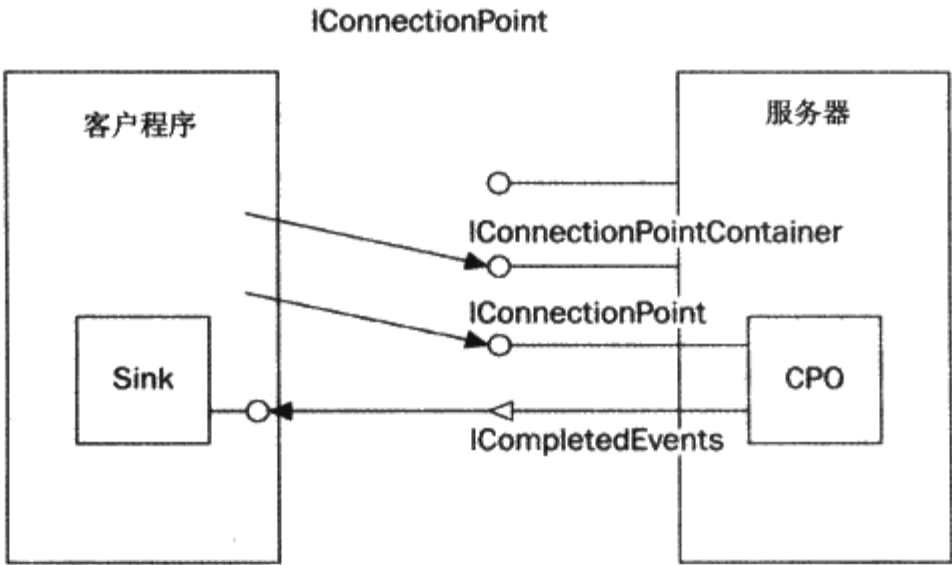


图 24-6

本章的后面将讨论.NET 事件和 COM 事件如何映射，让.NET 客户程序处理 COM 事件，COM 对象处理.NET 事件。

24.2 编组

从.NET 传送给 COM 组件和从 COM 组件传送给.NET 的数据必须转换为相应的表示法，这个机制也称为编组(marshaling)。具体的转换过程取决于所传递数据的类型。这里必须区分 blittable 和 non-blittable 数据类型。

blittable 数据类型在.NET 和 COM 中有相同的表示法,不需要转换。简单的数据类型如 byte、short、int 和 long, 仅包含这些简单数据类型的类和数组都属于 blittable 数据类型。blittable 类型的数组必须是一维的。

non-blittable 数据类型需要进行转换。表 24-1 列出了一些 non-blittable 的 COM 数据类型及其对应的.NET 数据类型。non-blittable 数据类型需要转换,所以需要更多的开销。

表 24-1

COM 数据类型	.NET 数据类型
SAFEARRAY	Array
VARIANT	Object
BSTR	String
IUnknown*,IDispatch*	Object

24.3 在.NET 客户程序中使用 COM 组件

要理解.NET 应用程序如何使用 COM 组件,首先必须创建 COM 组件。创建 COM 组件不能使用 C#或 Visual Basic 2005, 而应使用 Visual Basic 6 或 C++(或其他支持 COM 的语言)。本章使用 Active Template Library(ATL)和 C++。

注意:

使用 C#或 Visual Basic 9.0 可以创建.NET 组件,通过一个封装器就可以把该.NET 组件用作 COM 对象,而封装器是真正的 COM 组件。封装在 COM 组件中的.NET 组件由.NET 客户程序通过 COM 交互操作功能来使用是没有意义的。

因为本书讲述的不是 COM, 所以不讨论代码的各个方面, 只讨论建立示例所需要的代码。

24.3.1 创建 COM 组件

要用 ATL 和 C++创建 COM 组件,先创建一个新的 ATL 项目。选择 File | New | Project 后, 就会在 Visual C++ Projects 组中看到 ATL Project 向导。把名称设置为 COMServer。在 Application Settings 中, 选择 Dynamic Link Library, 再按下 Finish。

ATL Project 向导刚才已为服务器创建了基础代码。还需要一个 COM 对象。在 Solution Explorer 中添加一个类, 选择 ATL Simple Object。在打开的对话框中, 为 Short name 字段输入 COMDemo。其他字段都是自动填充的, 但把接口名改为 IWelcome, 如图 24-7 所示。单击 Finish 为类和接口创建基本代码。



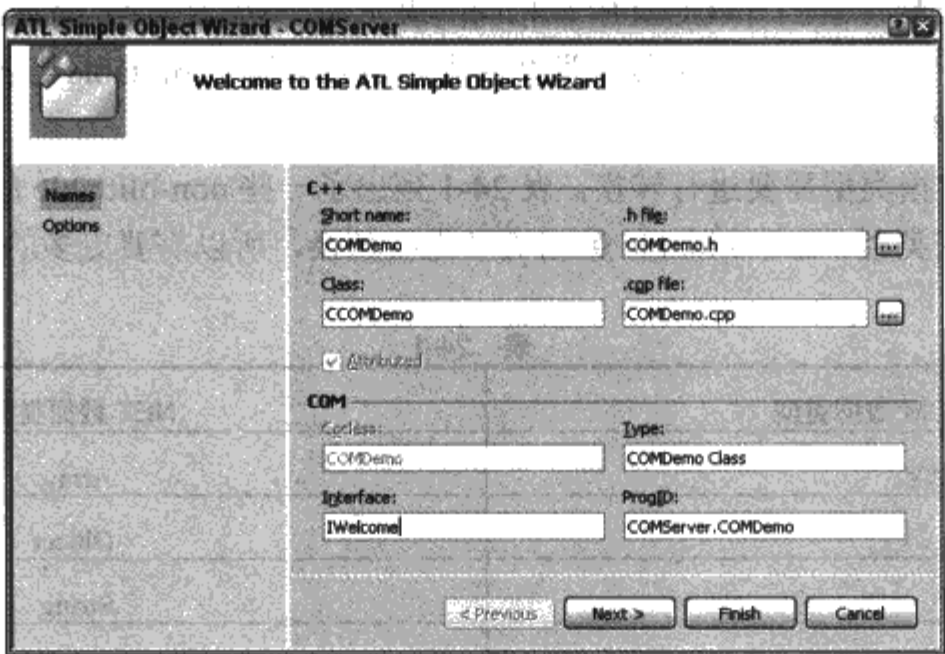


图 24-7

COM 组件提供了两个接口，所以可以看到 QueryInterface()方法是如何在.NET 中映射的。COM 组件还提供了 3 个简单的方法，所以我们可以看到交互操作是如何进行的。在 Class 视图中，选择接口 IWelcome，添加方法 Greeting()，如图 24-8 所示，该方法有 3 个参数：

```
HRESULT Greeting([in] BSTR name, [out, retval] BSTR* message);
```

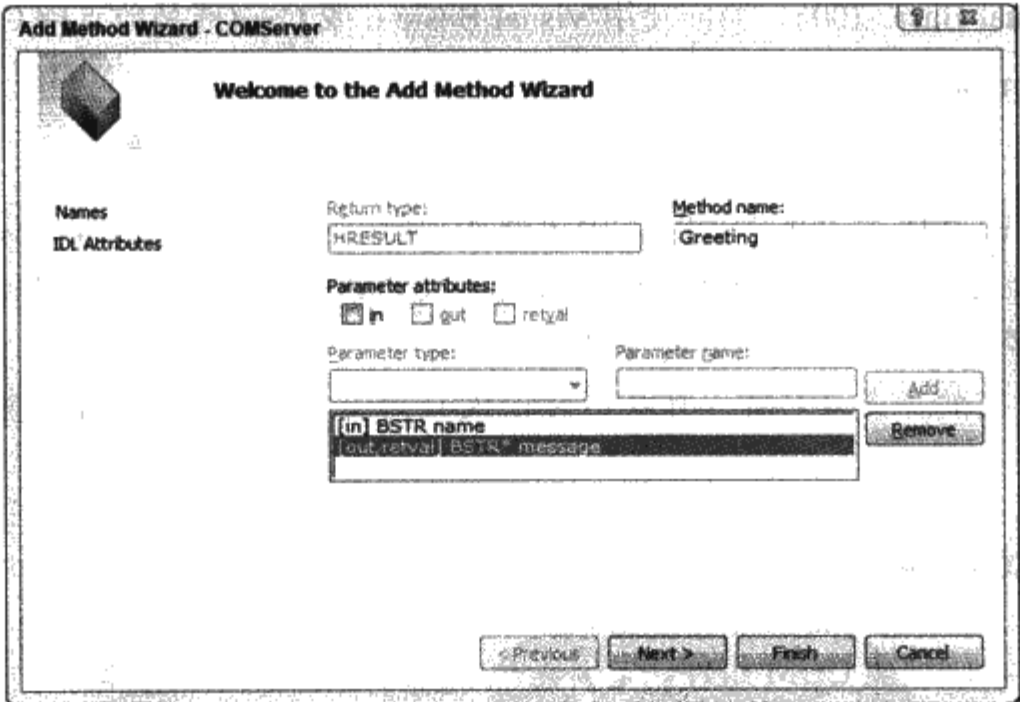


图 24-8

IDL 文件 COMDemo.idl 给 COM 定义了接口。向导在文件 COMDemo.idl 中生成的代码如下所示。唯一标识符 uuid 会有所不同。IWelcome 接口定义了方法 Greeting()。关键字 \_interface 之前的方括号定义了接口的一些特性。uuid 定义了接口的 ID，dual 标识了接口的类型。

```
[  
    object,  
    uuid(615B801E-3A5C-44EA-913B-8C8F53BBFB3F),  
    dual,  
    nonextensible,  
    ...  
]  
interface IWelcome  
{  
    HRESULT Greeting([in] BSTR name, [out, retval] BSTR* message);  
};
```

```

    helpstring("IWelcome Interface"),
    pointer_default(unique)
]
interface IWelcome : IDispatch{
    [id(1), helpstring("method Greeting")] HRESULT Greeting(
        [in] BSTR name, [out,retval] BSTR* message);
};

```

IDL 文件还定义了类型库的内容, 它是执行 IWelcome 接口的 COM 对象(coclass):

```

[
    uuid(1CE0DFFF-ADA8-47DD-BA06-DDD89C584242),
    version(1.0),
    helpstring("COMServer 1.0 Type Library")
]
library COMServerLib
{
    importlib("stdole2.tlb");
    [
        uuid(AB13E0B8-F8E1-497E-985F-FA30C5F449AA),
        helpstring("COMDemo Class")
    ]
    coclass COMDemo
    {
        [default] interface IWelcome;
    };
};

```

#### 提示:

在定制特性中, 可以改变由 .NET 包装器类生成的类和接口的名称。只需给特性 custom 添加标识符 0F21F359-AB84-41e8-9A78-36D110E6D2F9, 并给 .NET 中显示的内容指定名称即可。

在 IWelcome 接口的头文件部分, 添加带有相同标识符和名称 Wrox.ProCSharp 的定制特性 COMInterop.Server.IWelcome。给类 CCOMDemo 添加带有相应名称的同一特性。

```

[
    object,
    uuid(615B801E-3A5C-44EA-913B-8C8F53BBFB3F),
    dual,
    nonextensible,
    helpstring("IWelcome Interface"),
    pointer default(unique),
    custom(0F21F359-AB84-41e8-9A78-36D110E6D2F9,
        "Wrox.ProCSharp.COMInterop.Server.IWelcome")
]
interface IWelcome : IDispatch{
    [id(1), helpstring("method Greeting")] HRESULT Greeting([in] BSTR name,
        [out,retval] BSTR* message);
};

library COMServerLib
{
    importlib("stdole2.tlb");
    [
        uuid(AB13E0B8-F8E1-497E-985F-FA30C5F449AA),
        helpstring("COMDemo Class"),
        custom(0F21F359-AB84-41e8-9A78-36D110E6D2F9,
            "Wrox.ProCSharp.COMInterop.Server.COMDemo"),
    ]
    coclass COMDemo
    {
        [default] interface IWelcome;
    };
};

```

```

    ]
    coclass COMDemo
    {
        [default] interface IWelcome;
    };

```

现在给文件 COMDemo.idl 添加第二个接口。可以把 IWelcome 接口的头文件部分复制到新接口 IMath 的头文件部分，但要确保修改用 uuid 关键字定义的唯一标识符。可以用实用工具 guidgen 生成这样一个 ID。接口 IMath 提供了两个方法 Add() 和 Sub()。

```

// IMath
[
    object,
    uuid("2158751B-896E-461d-9012-EF1680BE0628"),
    dual,
    nonextensible,
    helpstring("IMath Interface"),
    pointer_default(unique),
    custom(0F21F359-AB84-41e8-9A78-36D110E6D2F9,
        "Wrox.ProCSharp.COMInterop.Server.IMath")
]
interface IMath : IDispatch
{
    [id(1)] HRESULT Add([in] LONG val1, [in] LONG val2, [out, retval] LONG* result);
    [id(2)] HRESULT Sub([in] LONG val1, [in] LONG val2, [out, retval] LONG* result);
};

```

类 CCOMDemo 必须修改，使之实现接口 IWelcome 和 IMath，接口 IWelcome 是默认接口。

```

[
    uuid(AB13E0B8-F8E1-497E-985F-FA30C5F449AA),
    helpstring("COMDemo Class"),
    custom(0F21F359-AB84-41e8-9A78-36D110E6D2F9,
        "Wrox.ProCSharp.COMInterop.Server.COMDemo")
]
coclass COMDemo
{
    [default] interface IWelcome;
    interface IMath;
};

```

现在可以把注意力从 IDL 文件转向 C++ 代码了。在 COMDemo.h 文件中，包含了 COM 对象的类定义。CCOMDemo 类使用多重继承机制，继承了模板类 CComObjectRootEx、CComCoClass 和 IDispatchImpl。CComObjectRootEx 提供了 IUnknown 接口功能的执行代码，如 AddRef 和 Release。CComCoClass 创建了一个实例工厂，来实例化模板变元的对象，这里是 CComDemo。IDispatchImpl 提供了 IDispatch 接口中方法的执行代码。

利用 BEGIN\_COM\_MAP 和 END\_COM\_MAP 中的宏，创建一个映射，定义 COM 类实现的所有 COM 接口。这个映射由 QueryInterface 方法的执行代码使用。

```

class ATL_NO_VTABLE CCOMDemo :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CCOMDemo, &CLSID COMDemo>,
    public IDispatchImpl<IWelcome, &IID_IWelcome, &LIBID_COMServerLib,
        /*wMajor =*/ 1, /*wMinor =*/ 0>
{
public:

```

```

CCOMDemo()
{
}

DECLARE_REGISTRY_RESOURCEID(IDR_COMDEMO)

BEGIN_COM_MAP(CCOMDemo)
    COM_INTERFACE_ENTRY(IWelcome)
    COM_INTERFACE_ENTRY(IDispatch)
END_COM_MAP()

DECLARE_PROTECT_FINAL_CONSTRUCT()

HRESULT FinalConstruct()
{
    return S_OK;
}

void FinalRelease()
{
}

public:
    STDMETHODCALLTYPE(Greeting)(BSTR name, BSTR* message);
};

OBJECT_ENTRY_AUTO(__uuidof(CCOMDemo), CCOMDemo)

```

有了这个类定义，还必须添加第二个接口 `IMath`，以及用 `IMath` 接口定义的方法：

```

class ATL_NO_VTABLE CCOMDemo :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CCOMDemo, &CLSID_COMDemo>,
    public IDispatchImpl<IWelcome, &IID_IWelcome, &LIBID_COMServerLib,
        /*wMajor =*/ 1, /*wMinor =*/ 0>,
    public IDispatchImpl<IMath, &IID_IMath, &LIBID_COMServerLib, 1, 0>
{
public:
    CCOMDemo()
    {
    }

    DECLARE_REGISTRY_RESOURCEID(IDR_COMDEMO)

    BEGIN_COM_MAP(CCOMDemo)
        COM_INTERFACE_ENTRY(IWelcome)
        COM_INTERFACE_ENTRY(IMath)
        COM_INTERFACE_ENTRY2(IDispatch, IWelcome)
    END_COM_MAP()

    DECLARE_PROTECT_FINAL_CONSTRUCT()

    HRESULT FinalConstruct()
    {
        return S_OK;
    }

    void FinalRelease()
    {
    }
}

```



```

    }

public:
    STDMETHOD(Greeting)(BSTR name, BSTR* message);
    STDMETHOD(Add)(long val1, long val2, long* result);
    STDMETHOD(Sub)(long val1, long val2, long* result);
};

OBJECT_ENTRY_AUTO(__uuidof(COMDemo), CCOMDemo)

```

现在可以在文件 `COMDemo.cpp` 中用下面的代码执行 3 个方法了。`CComBSTR` 是一个很容易处理 `BSTR` 的 ATL 类。在 `Greeting()` 方法中, 只返回一个欢迎信息, 并把第一个参数传入的名称添加到返回的信息中。`Add()` 方法把两个值加在一起, 而 `Sub()` 方法进行减法操作, 返回相减的结果。

```

STDMETHODIMP CCOMDemo::Greeting(BSTR name, BSTR* message)
{
    CComBSTR tmp("Welcome, ");
    tmp.Append(name);
    *message = tmp;
    return S_OK;
}

STDMETHODIMP CCOMDemo::Add(LONG val1, LONG val2, LONG* result)
{
    *result = val1 + val2;
    return S_OK;
}

STDMETHODIMP CCOMDemo::Sub(LONG val1, LONG val2, LONG* result)
{
    *result = val1 - val2;
    return S_OK;
}

```

现在就可以建立组件了。建立过程也是指在注册表中配置组件。

### 24.3.2 创建 Runtime Callable Wrapper

现在可以在 .NET 中使用 COM 组件了。为此, 必须创建一个 `Runtime Callable Wrapper (RCW)`。使用 `RCW`, .NET 客户程序就可以使用 .NET 对象而不是 COM 组件, 所以不需要处理 COM 特性, 这是由包装器来处理的。`RCW` 隐藏了 `IUnknown` 和 `IDispatch` 接口(如图 24-9 所示), 并处理 COM 对象的引用数。

`RCW` 可以使用命令行实用工具 `tlbimp` 或 `Visual Studio` 来创建。启动命令:

```
tlbimp COMServer.dll /out: Interop.COMServer.dll
```

这个命令会创建文件 `Interop.COMServer.dll`, 其中包含带有包装器类的 .NET 程序集。在这个生成的程序集中, 可以找到命名空间 `COMWrapper`、类 `CCOMDemoCalss`、接口 `CCOMDecmo`、`IMath` 和 `IWelcome`。命名空间的名称可以使用实用工具 `tlbimp` 的选项来修改。选项 `/namespace` 允许指定不同的命名空间, `/asmversion` 可以定义程序集的版本号。



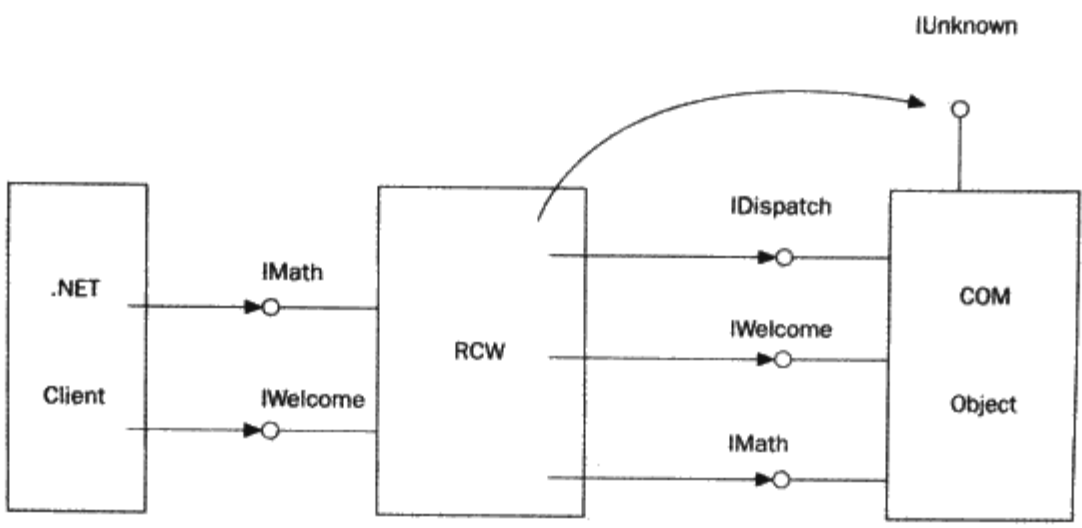


图 24-9

注意:

这个命令行实用工具的另一个重要选项是/keyfile，它可以把一个强名赋给所生成的程序集。关于强名，详见第 17 章。

RCW 还可以使用 Visual Studio 来创建。要创建一个简单的示例应用程序，应创建一个 C# 控制台项目。在 Solution Explorer 中，选择 Add Reference 对话框中的 COM 选项卡，向下滚动到 COMServer 1.0 Type Library 选项上，可以添加对 COM 服务器的引用，如图 24-10 所示。这里列出了在注册表中配置的所有 COM 对象。从列表选择一个 COM 组件，创建一个带 RCW 类的程序集。

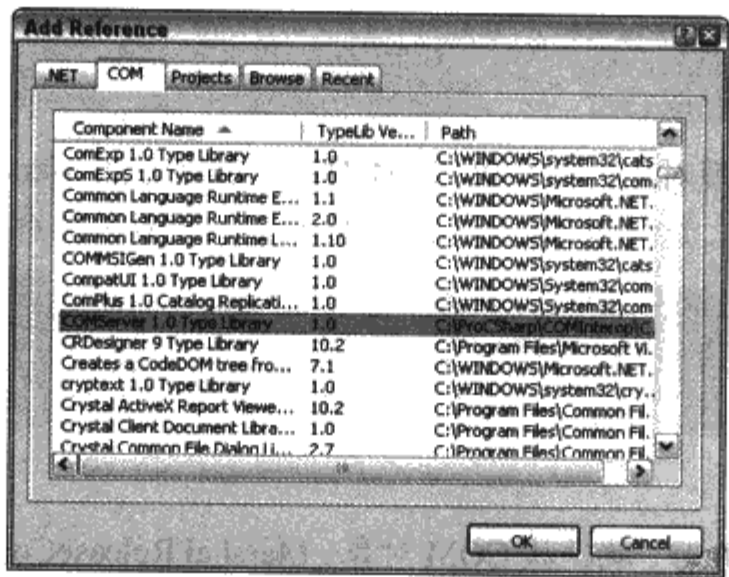


图 24-10

24.3.3 使用 RCW

在创建好包装器类后，就可以为应用程序编写代码来实例化和访问组件了。由于在 C++ 文件中有定制特性，所以 RCW 类生成的命名空间是 Wrox.ProCSharp.COMInterop.Server。在上面的声明中添加这个命名空间和 System.Runtime.InteropServices 命名空间。在 System.Runtime.InteropServices 命名空间中，使用 Marshal 类可以释放 COM 对象。

```
using System;
using System.Runtime.InteropServices;
```

```
using Wrox.ProCSharp.COMInterop.Server;

namespace Wrox.ProCSharp.COMInterop.Client
{
    class Program
    {
        [STAThread]
        static void Main(string[] args)
        {
```

现在可以像使用.NET类那样使用COM组件了。obj是COMDemo类型的变量，COMDemo是一个.NET接口，它提供了IWelcome和IMath接口的方法。还可以对特定的接口进行强制数据类型转换，例如IWelcome。有了声明为IWelcome类型的变量后，就可以调用方法Greeting()。

#### 注意：

尽管COMDemo是一个接口，但可以实例化COMDemo类型的新对象。与一般的接口不同，可以对封装的COM接口进行这类操作。

```
COMDemo obj = new COMDemo();
IWelcome welcome = obj;
Console.WriteLine(welcome.Greeting("Christian"));
```

如果对象(如本例所示)提供了多个接口，就可以声明其他接口的变量，通过使用简单的赋值语句和强制转换运算符，包装器类就可以通过QueryInterface()方法和COM对象返回第二个接口指针。使用math变量可以调用IMath接口的方法。

```
IMath math;
math = (IMath)obj;
int x = math.Add(4,5);
Console.WriteLine(x);
```

如果在垃圾收集器清理对象之前释放COM对象，静态方法Marshal.ReleaseComObject()就调用组件的Release()方法，这样组件就可以销毁它自己，并释放内存了。

```
Marshal.ReleaseComObject(math);
}
}
}
```

#### 提示：

前面提到，引用数为0时，就释放COM对象。Marshal.ReleaseComObject()会调用Release()方法给引用数减1。RCW仅调用一次AddRef()方法，来递增引用数，所以无论有多少对RCW的引用，调用一次Marshal.ReleaseComObject()，就足以释放对象了。

在使用Marshal.ReleaseComObject()释放COM对象后，就不能使用引用该对象的变量了。在本例中，COM对象是使用math变量释放的。welcome变量也引用了COM对象，它不能在释放对象后使用。否则，就会生成一个InvalidComObjectException类型的异常。

#### 注意：

COM对象在不再需要时就释放，这是很重要的。COM对象使用内置的内存堆，而.NET对象使用托管的内存堆。垃圾收集器只处理托管的内存。

可以看出,有了 RCW,就可以像使用.NET 对象那样来使用 COM 组件。  
RCW 的特殊情况是可交互操作的主程序集。

#### 24.3.4 可交互操作的主程序集

可交互操作的主程序集是 COM 组件的供应商已经准备好的程序集。它可以更容易地使用 COM 对象。可交互操作的主程序集不同于自动生成的 RCW。

可交互操作的主程序集在目录<program file>\Microsoft .NET\Primary Interop Assemblies 中。它已经存在,用于.NET 中的 ADO。如果添加对 COM 库 Microsoft ActiveX Data Objects 2.7 Library 的引用,就不创建包装器类,因为可交互操作的主程序集已经存在;否则就引用可交互操作的主程序集。

#### 24.3.5 线程问题

如本章前面所述,COM 组件根据执行的线程安全与否,标记它所在的单元(STA 或 MTA)。但是,线程必须加入到单元中。添加线程的单元可以用[STAThread]和[MTAThread]特性定义,这两个特性可以应用于应用程序的方法 Main()。[STAThread]特性表示线程加入 STA。而特性[MTAThread]表示线程加入 MTA。如果没有应用特性,默认情况下就加入 MTA。

还可以使用 Thread 类的 ApartmentState 特性编程设置单元状态。ApartmentState 特性允许设置 ApartmentState 枚举中的一个值。ApartmentState 枚举的值有 STA 和 MTA(如果没有设置,就使用 Unknown)。注意线程的单元状态只能设置一次。如果第二次设置它,就会忽略第二次的设置。

**提示:**

如果线程选择了组件所不支持的单元,该怎么办? COM 运行库会自动创建 COM 组件的正确单元。但是,如果在调用组件的方法时跨越了单元边界,性能就会降低。

#### 24.3.6 添加连接点

为了理解 COM 事件在.NET 应用程序中的处理方式,首先必须扩展 COM 组件。使用特性在 ATL 类中执行 COM 事件类似于执行.NET 中的事件,但功能不同。

首先必须在接口定义文件 COMDemo.idl 中添加另一个接口。接口 \_ICompletesEvents 由客户程序(.NET 应用程序)实现,由组件调用。在本例中,在完成了计算后,方法 Completes()由组件调用。这个接口也称为输出接口。输出接口必须是分派接口或定制接口。所有的客户程序都支持分派接口。id 为 0F21F359-AB84-41e8-9A78-36D110E6D2F9 的定制属性定义了 RCW 中创建的接口名。输出接口还必须在 coclass 段中写入组件支持的接口,标记为 source 接口。

```
library COMServerLib
{
    importlib("stdole2.tlb");
    [
        uuid(5CFF102B-0961-4EC6-8BB4-759A3AB6EF48),
        helpstring("_ICompletedEvents Interface"),
```

```

        custom(0F21F359-AB84-41e8-9A78-36D110E6D2F9,
            "Wrox.ProCSharp.COMInterop.Server.ICompletedEvents"),
    ]
    dispinterface _ICompletedEvents
    {
        properties:
        methods:
            [id(1)] void Completed(void);
    };

[
    uuid(AB13E0B8-F8E1-497E-985F-FA30C5F449AA),
    helpstring("COMDemo Class")
    custom(0F21F359-AB84-41e8-9A78-36D110E6D2F9,
        "Wrox.ProCSharp.COMInterop.Server.COMDemo"),
]
coclass COMDemo
{
    [default] interface IWelcome;
    interface IMath;
    [default, source] dispinterface _ICompletedEvents;
};

```

使用向导可以创建执行代码，将事件发送给客户程序。打开类视图，选择 CComDemo 类，打开弹出菜单，启动 Implement Connection Point Wizard，如图 24-11 所示。给带连接点的执行代码选择源接口 ICompletedEvents。

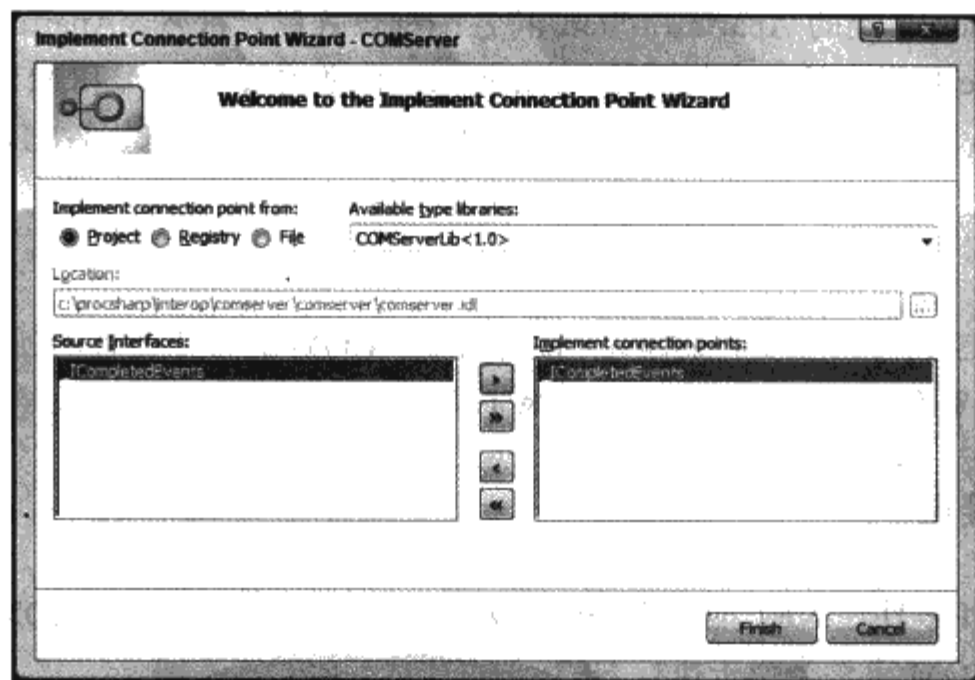


图 24-11

向导创建了代理类 CProxy\_ICompletedEvents，将事件发送给客户程序。另外，还修改了 CCOMDemo 类。这个类现在继承了 IConnectionPointContainerImpl 和代理类。IConnectionPointContainer 接口添加到接口映射中，连接点映射添加到源接口 ICompletedEvents 中。

```

class ATL_NO_VTABLE CCOMDemo :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CCOMDemo, &CLSID_COMDemo>,
    public IDispatchImpl<IWelcome, &IID_IWelcome, &LIBID_COMServerLib,
        /*wMajor =*/ 1, /*wMinor =*/ 0>,
    public IDispatchImpl<IMath, &IID_IMath, &LIBID_COMServerLib, 1, 0>,
    public IConnectionPointContainerImpl<CCOMDemo>,

```

```

    public CProxy_ICompletedEvents<CCOMDemo>
    {
    public:

    //...

    BEGIN_COM_MAP(CCOMDemo)
        COM_INTERFACE_ENTRY(IWelcome)
        COM_INTERFACE_ENTRY(IMath)
        COM_INTERFACE_ENTRY2(IDispatch, IWelcome)
        COM_INTERFACE_ENTRY(IConnectionPointContainer)
    END_COM_MAP()

    //...

    public:
        BEGIN_CONNECTION_POINT_MAP(CCOMDemo)
            CONNECTION_POINT_ENTRY(__uuidof(_ICompletedEvents))
        END_CONNECTION_POINT_MAP()
    };

```

最后，在文件 COMDemo.cpp 的 Add() 和 Sub() 方法中调用代理类的方法 Fire\_Completed()。

```

STDMETHODIMP CCOMDemo::Add(LONG val1, LONG val2, LONG* result)
{
    *result = val1 + val2;
    Fire_Completed();
    result S_OK;
}

STDMETHODIMP CCOMDemo::Sub(LONG val1, LONG val2, LONG* result)
{
    *result = val1 - val2;
    Fire_Completed();
    return S_OK;
}

```

在重新建立 COM DLL 之后，就可以把 .NET 客户程序改为使用这些 COM 事件，这与使用一般的 .NET 事件一样。

```

static void Main()
{
    COMDemo obj = new COMDemo();

    IWelcome welcome = obj;
    Console.WriteLine(welcome.Greeting("Christian"));

    obj.Completed +=
        new ICompletedEvents_CompletedEventHandler(
            delegate
            {
                Console.WriteLine("Calculation completed");
            });

    IMath math = (IMath)welcome;
    int result = math.Add(3, 5);
    Console.WriteLine(result);

    Marshal.ReleaseComObject(math);
}

```

可以看出，RCW 提供了从 COM 事件到 .NET 事件的自动映射。可以在 .NET 客户程序中像



使用.NET 事件那样使用 COM 事件。

### 24.3.7 在 Windows 窗体中使用 ActiveX 控件

ActiveX 控件是带有一个用户界面和许多可选 COM 接口的 COM 对象, 这些 COM 接口可以处理用户界面, 与容器进行交互操作。ActiveX 控件可以被许多不同的容器使用, 例如 Internet Explorer、Word、Excel, 用 VB6、MFC(Microsoft Foundation Classes)或 ATL(Active Template Library)编写的应用程序。Windows 窗体应用程序是另一个可以管理 ActiveX 控件的容器。ActiveX 控件可以像前面讨论的 Windows 窗体控件那样使用。

#### 1. ActiveX 控件导入器

与 RCW 一样, 也可以为 ActiveX 控件创建包装器。ActiveX 控件的包装器可以使用命令行实用工具 Windows Forms ActiveX Control Importer(aximp.exe)来创建。这个实用工具会创建一个派生于基类 System.Windows.Forms.AxHost 的类, 它用作包装器, 以使用 ActiveX 控件。

在 Web 窗体控件中输入下面的命令, 就可以创建一个包装器类:

```
aximp c:\windows\system32\shdocvw.dll
```

ActiveX 控件也可以直接使用 Visual Studio 导入。如果 ActiveX 控件在工具箱中配置好了, 就可以拖放到创建包装器的 Windows 窗体控件中。

#### 2. 创建 Windows 窗体应用程序

要查看 ActiveX 控件在 Windows 窗体应用程序中的运行情况, 需要创建一个简单的 Windows 窗体应用程序项目。在这个应用程序中, 建立一个简单的 Internet 浏览器, 它使用来自操作系统的 Web Browser 控件。

创建如图 24-12 所示的窗体, 该窗体应包含一个工具栏, 其中包含一个文本框和三个按钮。文本框 toolStripTextUrl 用于输入 URL, 3 个浏览 Web 页面的按钮, 其名称分别是 toolStripButtonNavigate、toolStripButtonBack 和 toolStripButtonForward, 最后还有一个名为 statusStrip 的状态栏。状态栏也需要一个标签来显示状态信息。

使用 Visual Studio, 可以把 ActiveX 控件添加到工具栏中, 以与 Windows 窗体控件相同的方式使用它们。在 Customize Toolbox 快捷菜单中, 选择菜单项 Add | Remove Items, 再在 COM Components 类别中选择 Microsoft Web Browser 控件, 如图 24-13 所示。

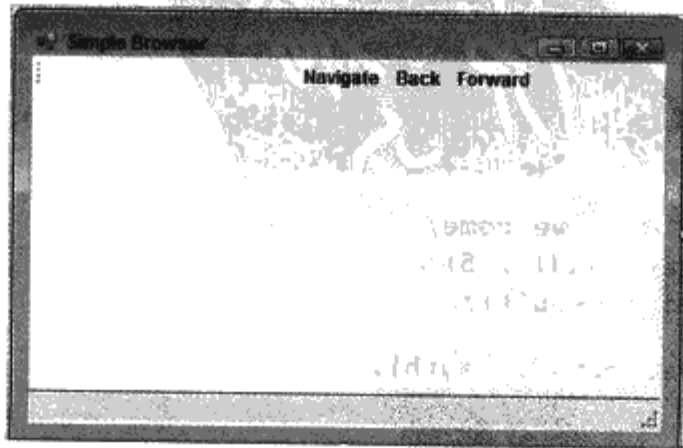


图 24-12

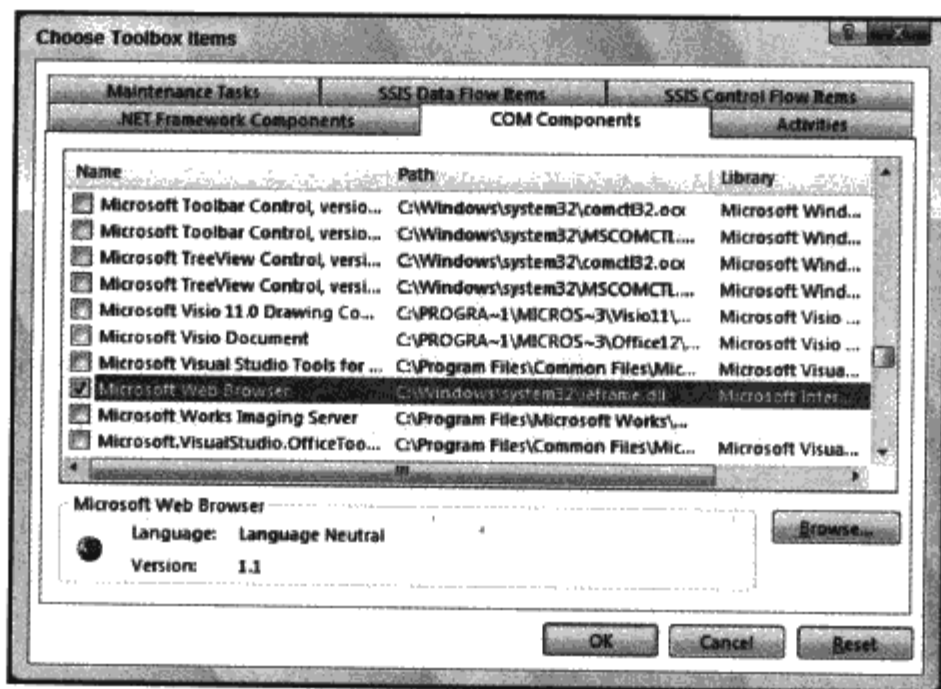


图 24-13

这样，图标就会显示在工具栏上。与其他 Windows 控件类似，可以把这个图标拖放到 Windows 窗体设计器中，以创建(使用 aximp 实用工具)包含 ActiveX 控件的包装器程序集。在项目中使用引用可以看到包装器程序集：AxSHDocVw 和 SHDocVw。现在可以使用生成的变量 axWebBrowser1 调用控件的方法，如下面的代码所示。给按钮 toolStripButton Navigate 添加一个 click 事件处理程序，让浏览器导航到 Web 页面上。为此使用的方法 Navigate()需要在它的第一个参数中传递 URL 字符串，该字符串可以通过访问文本框控件 toolStripTextUrl 的 Text 特性来获得。

```
private void OnNavigate(object sender, System.EventArgs e)
{
    try
    {
        axWebBrowser1.Navigate(toolStripTextUrl.Text);
    }
    catch (COMException ex)
    {
        statusStrip.Items[0].Text = ex.Message;
    }
}
```

在 Back 和 Forward 按钮的 Click 事件处理程序中，调用浏览器控件的 GoBack()和 GoForward()方法：

```
private void OnBack(object sender, System.EventArgs e)
{
    try
    {
        axWebBrowser1.GoBack();
    }
    catch (COMException ex)
    {
        statusStrip.Items[0].Text = ex.Message;
    }
}

private void OnForward(object sender, System.EventArgs e)
{
}
```

```

try
{
    axWebBrowser1.GoForward();
}
catch (COMException ex)
{
    statusStrip.Items[0].Text = ex.Message;
}
}

```

Web 控件也提供了一些可用作 .NET 事件的事件。给事件 `StatusTextChanged` 添加事件处理程序 `OnStatusChange()`，把由控件返回的状态设置为 Windows 窗体应用程序中的状态栏。

```

private void OnStatusChange(object sender,
    AxSHDocVw.DWebBrowserEvents2 StatusTextChangedEvent e)
{
    statusStrip.Items[0].Text = e.text;
}

```

现在就有了一个简单的浏览器，可用于导航到 Web 页面上，如图 24-14 所示。

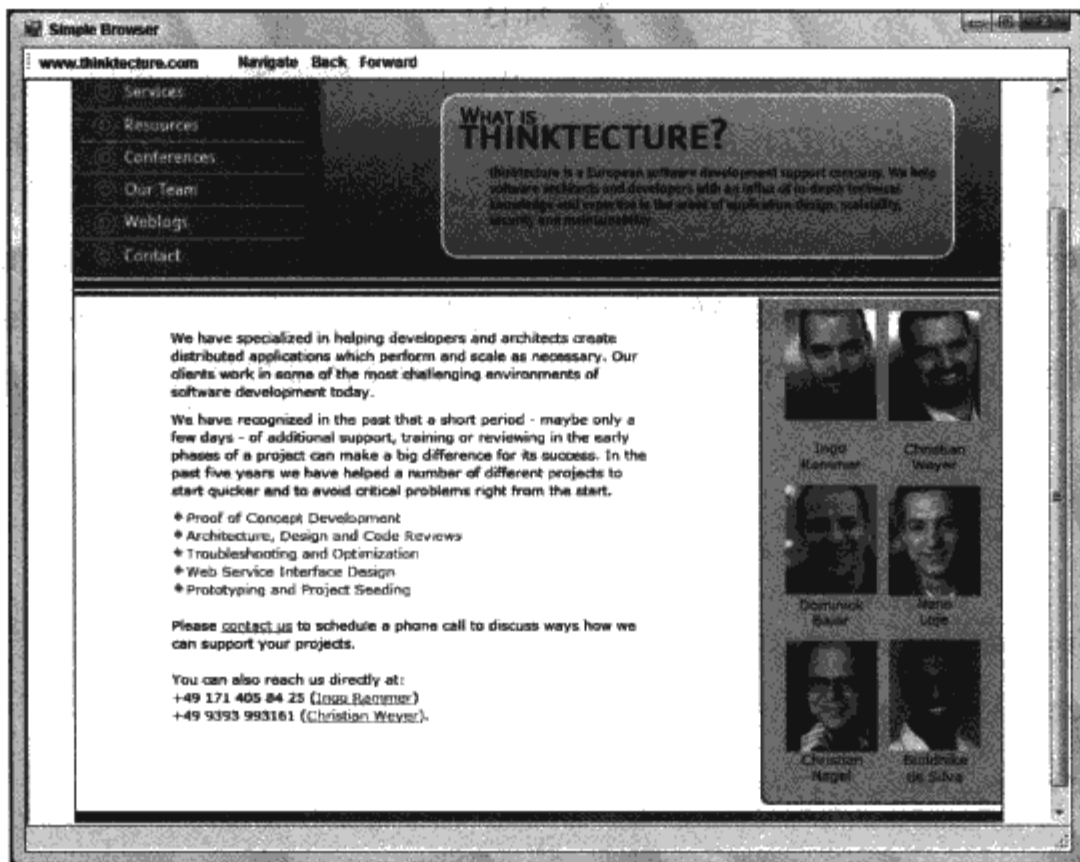


图 24-14

### 24.3.8 在 ASP.NET 中使用 COM 对象

可以用以前介绍的方式在 ASP.NET 中使用 COM 对象。但是，有一个重要的区别。ASP.NET 运行库在默认情况下运行于 MTA 中。如果用线程模型值 `Apartment` 配置 COM 对象(因为所有的 COM 对象都是用 VB6 编写的)，就会生成一个异常。考虑到性能和可伸缩性，最好避免在 ASP.NET 中使用 STA。如果的确希望在 ASP.NET 中使用 STA 对象，可以用 `Page` 指令设置 `AspNetCompat` 特性，如下面的代码所示。注意在使用这个选项时，Web 站点的性能可能会有损失。

```
<%@ Page AspNetCompat = "true" Language="C#" %>
```

提示:

使用 STA COM 对象和 ASP.NET, 会导致可伸缩问题, 最好避免使用 STA COM 对象和 ASP.NET.

## 24.4 在 COM 客户程序中使用 .NET 组件

前面讨论了如何在 .NET 客户程序中访问 COM 组件。在使用 VB6、MFC 或 ATL 编写的旧 COM 客户程序中访问 .NET 组件也同样十分有意义。

### 24.4.1 COM Callable Wrapper

如果要使用 .NET 客户程序访问 COM 组件, 就必须使用 RCW。要在 COM 客户程序中访问 .NET 组件, 就必须使用 COM Callable Wrapper (CCW)。图 24-15 显示了封装 .NET 类的 CCW, 提供了 COM 客户程序希望使用的 COM 接口。CCW 提供了 IUnknown、IDispatch、ISupportError Info 及其他接口, 它还为事件提供了接口 IConnectionPointContainer 和 IConnectionPoint。COM 客户程序可以从 COM 对象中得到它需要的信息, 但 .NET 组件是在后台上。包装器处理接口 IUnknown 中的 AddRef()、Release()、QueryInterface() 方法。而在 .NET 对象中, 可以依赖垃圾收集器, 不需要处理引用数。

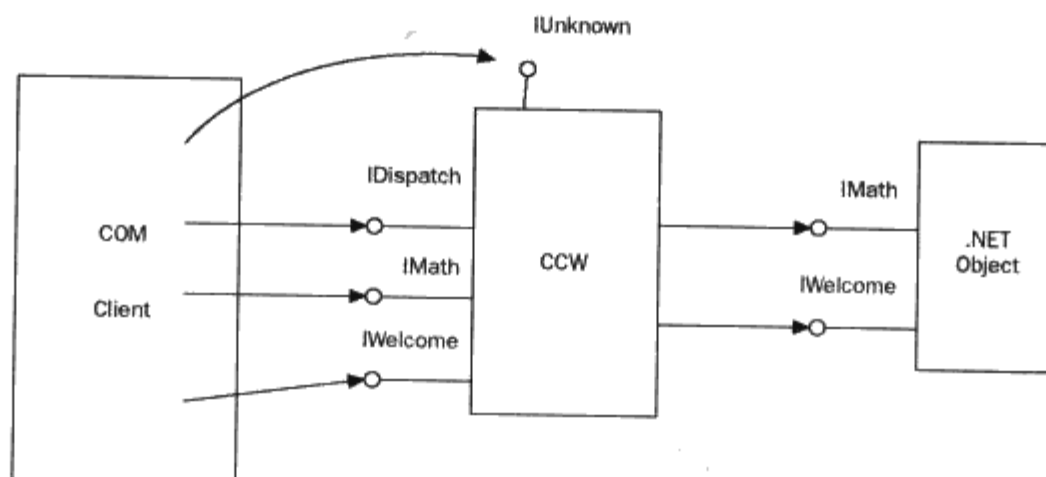


图 24-15

### 24.4.2 创建 .NET 组件

在下面的示例中, 要在 .NET 类中建立与 COM 组件相同的功能。首先创建一个 C# 类库, 命名为 DotNetComponent。接着添加接口 IWelcome 和 IMath, 以及实现这些接口的类 NetComponent。属性 [ComVisible(true)] 使类和接口可用于 COM:

```
using System;
using System.Runtime.InteropServices;

namespace Wrox.ProCSharp.COMInterop.Server
{
    [ComVisible(true)]
```

```

public interface IWelcome
{
    string Greeting(string name);
}

[ComVisible(true)]
public interface IMath
{
    int Add(int val1, int val2);
    int Sub(int val1, int val2);
}

[ComVisible(true)]
public class DotnetComponent : IWelcome, IMath
{
    public DotnetComponent()
    {
    }

    public string Greeting(string name)
    {
        return "Hello " + name;
    }

    public int Add(int val1, int val2)
    {
        return val1 + val2;
    }

    public int Sub(int val1, int val2)
    {
        return val1 - val2;
    }
}

```

建立了项目后，就可以创建类型库了。

### 24.4.3 创建类型库

类型库可以用命令行实用工具 `tlbexp` 创建。命令：

```
tlbexp DotNetComponent.dll
```

会创建类型库 `DotNetComponent.tlb`。使用实用工具 `OLE/COM Object Viewer` (`oleview32.exe`) 可以查看类型库，它在 `Microsoft SDK` 中。可以在 `Visual Studio 2008 Command Prompt` 中访问这个实用工具，选择 `File | View TypeLib`，打开类型库。在下面的代码中，可以查看接口定义。唯一的 ID 可能不同。

类型库的名称是根据程序集的名称创建的。类型库的头文件还在定制特性中定义了程序集的全名，所有的接口都在定义之前声明：

```

//Generated .IDL file (by the OLE/COM object Viewer)
//
//typelib filename: DotNetComponent.dll

[
    uuid(0AA0953A-B2A0-32CB-A5AC-5DA0DF698EB8),

```



```

    version(1.0),
    custom(90883F05-3D24-11D2-8F17-00A0C9A6186D, DotNetComponent,
        Version=1.0.0.0, Culture=neutral, PublicKeyToken=null)
]
library DotNetComponent
{
    //TLib : Common Language Runtime Library :
    //{BED7F4EA-1A96-11D2-8F08-00A0C9A6186D}
    importlib("mscorlib.tlb");
    //TLib : OLE Automation : {00020430-0000-0000-C000-00000000000046}
    importlib("stdole2.tlb");

    //Forward declare all types defined in this typelib
    interface IWelcome;
    interface IMath;
    interface _DotNetComponent;

```

在下面生成的代码中，可以看到接口 `IWelcome` 和 `IMath` 定义为 COM 双重接口。在 C# 代码中声明的所有方法都列在类型库定义中。参数有所改变：.NET 类型映射为 COM 类型(如 `String` 类映射为 `BSTR` 类型)，签名也改变了，返回一个 `HRESULT`。由于接口是双重接口，所以还生成了分派 ID。

```

[
    odl,
    uuid(F39A4143-F88D-321E-9A24-8208E256A2DF),
    version(1.0),
    dual,
    oleautomation,
    custom(0F21F359-Ab84-41EB-9A78-36D110E6D2F9,
        Wrox.ProCSharp.COMInterop.Server.IWelcome)
]
interface IWelcome : IDispatch{
    [id(0x6000200000)]
    HRESULT Greeting([in] BSTR name, [out, retval] BSTR* pRetVal);
};

[
    odl,
    uuid(EF596F3F-B69B-3657-9D48-C906CBF12565),
    version(1.0),
    dual,
    oleautomation,
    custom(0F21F359-Ab84-41EB-9A78-36D110E6D2F9,
        Wrox.ProCSharp.COMInterop.Server.IMath)
]
interface IMath : IDispatch
{
    [id(0x6000200000)] HRESULT Add([in] long val1, [in] long val2,
        [out, retval] long* pRetVal);
    [id(0x6000200001)] HRESULT Sub([in] long val1, [in] long val2,
        [out, retval] long* pRetVal);
};

```

`coclass` 部分标记了 COM 对象本身。头文件中的 `uuid` 是用于实例化对象的 `CLSID`。类 `DotNetComponent` 支持接口 `_DotNetComponent`、`_Object`、`IWelcome` 和 `IMath`。`_Object` 在文件 `mscorlib.tlb` 中定义，该文件包含在前面的代码段中，提供了基类 `Object` 的方法。组件的默认接口是 `_DotNetComponent`，它在 `Coclass` 部分的后面定义为一个分派接口。在接口声明中，它标

记为双重，但因为不包含方法，所以是一个分派接口。在这个接口中，可以使用后期绑定访问组件的所有方法。

```
[
    uuid(5BCD9C26-D68D-38C2-92E3-DA0C1741A8CD),
    version(1.0),
    custom(0F21F359-Ab84-41EB-9A78-36D110E6D2F9,
        Wrox.ProCSharp.COMInterop.Server.DotnetComponent)
]
coclass DotnetComponent
[default interface _DotnetComponent;
 interface _Object;
 interface IWelcome;
 interface IMath;
];

[
    odl,
    uuid(884C59C6-B3C2-3455-BB74-52753C409097),
    hidden,
    dual,
    oleautomation,
    custom(0F21F359-Ab84-41EB-9A78-36D110E6D2F9,
        Wrox.ProCSharp.COMInterop.Server.DotnetComponent)
]
interface _DotnetComponent : IDispatch
{
};
};
```

生成类型库有许多默认选项。其优点是可以修改一些从.NET 到 COM 的默认映射。这可以使用命名空间 System.Runtime.InteropServices 中的几个特性来实现。

24.4.4 COM 互操作特性

把命名空间 System.Runtime.InteropServices 中的特性应用于类、接口或方法，可以修改 CCW 的实现方式。表 24-2 列出了这些特性和描述。

表 24-2

特 性	说 明
Guid	<p>这个特性可以赋予程序集、接口和类。把 Guid 用作程序集特性，会定义类型库 id；把它应用于接口，会定义接口 id(IID)；把它设置为一个类，会定义类 id(CLSID)。</p> <p>使用这个特性需要定义的唯一 ID，可以用实用工具 guidgen 来创建。</p> <p>在每次建立程序时，都会自动修改 CLSID 和类型库 id。如果不希望在每次建立程序时都改变它们，可以使用这个特性来固定它们。IID 只有在接口的签名改变时才被修改。例如，添加或删除了方法，或者改变了某些参数。因为在 COM 中，IID 应在每个接口的新版本中改变，所以这是很好的默认方式，通常不需要对 IID 应用 Guid 特性。对接口应用固定 IID 的唯一原因是.NET 接口是已有 COM 接口的准确表示，而且 COM 客户程序希望使用这个标识符</p>

(续表)

特 性	说 明
ProgId	这个特性可以应用于类，用于指定在注册表中配置对象时使用什么名称
ComVisible	这个特性设置为 false 时，可以在 COM 中隐藏类、接口、委托。它禁止创建 COM 表示
InterfaceType	这个特性如果设置为 COMInterfaceType 枚举值，就允许修改为.NET 接口创建的默认双重接口类型。COMInterfaceType 枚举值有 InterfaceIsDual、InterfaceIsIDispatch 和 InterfaceIsIUnknown。如果要把定制接口类型应用于.NET 接口，就设置这个特性： [InterfaceType(ComInterfaceType.InterfaceIsIUnknown)]
ClassInterface	这个特性允许修改为类创建的默认分派接口。ClassInterface 的参数是 ClassInterfaceType 枚举的值。该枚举值有 AutoDispatch、AutoDual 和 None。在前面的示例中默认值是 AutoDispatch，因为创建了一个分派接口。如果类只能由定义的接口访问，就应给这个类应用属性[ClassInterface (ClassInterfaceType.None)]
DispId	这个特性可以与双重和分派接口一起使用，以定义方法和特性的 dispid
In Out	如果参数应发送给组件[In]，或者参数从组件发送给客户程序[out]，或者参数是双向的[In,out]，COM 就允许把特性指定为参数类型
Optional	COM 方法的参数可以是可选的。可选参数可以用 Optional 特性标记

现在可以修改 C#代码，为 IWelcome 接口指定双重接口类型，为 IMath 接口指定定制接口类型，用参数 ClassInterfaceType.None 为类 DotNetComponent 指定 ClassInterface 特性，不生成单独的 COM 接口。属性 progid 和 guid 指定了 prog ID 和 GUID：

```
[interfaceType(ComInterfaceType.InterfaceIsDual)]
[ComVisible(true)]
public interface IWelcome
{
    [DispId(60040)]    string Greeting(string name);
}

[interfaceType(ComInterfaceType.InterfaceIsIUnknown)]
[ComVisible(true)]
public interface IMath
{
    int Add(int val1, int val2);
    int Sub(int val1, int val2);
}

[ClassInterface(ClassInterfaceType.None)]
[ProgId("Wrox.DotnetComponent")]
[Guid("77839717-40DD-8297-35B98A8402C7")]
[ComVisible(true)]
public class DotnetComponent : IWelcome, IMath
{
    public DotnetComponent()
    {
    }
}
```

重新建立类库和类型库，修改接口定义。使用 OleView.exe 可以验证这一点。如下面的 IDL 代码所示，接口 IWelcome 仍是双重接口，而 IMath 现在是一个派生自 IUnknown 的定制接口，



不是派生自 IDispatch。在 coclass 部分，删除了接口 \_DotNetComponent，目前 IWelcome 是新的默认接口，因为它是类 DotNetComponent 继承列表中的第一个接口。

```
// Generated .IDL file (by the OLE/COM Object Viewer)
//
// typelib filename: <could not determine filename>

[
    uuid(11E86506-EA54-3611-A55C-6830C48A554B),
    version(1.0),
    custom(90883F05-3D28-11D2-8F17-00A0C9A6186D, DotNetComponent,
        Version=1.0.1321.28677, Culture=neutral, PublicKeyToken=null)
]
library DotnetComponent
{
    // TLib : Common Language Runtime Library :
    // {BED7F4EA-1A96-11D2-8F08-00A0C9A6186D}
    importlib("mscorlib.tlb");
    // TLib : OLE Automation : {00020430-0000-0000-C000-000000000046}
    importlib("stdole2.tlb");

    // Forward declare all types defined in this typelib
    interface IWelcome;
    interface IMath;

    [
        odl,
        uuid(F39A4143-F88D-321E-9A33-8208E256A2DF),
        version(1.0),
        dual,
        oleautomation,
        custom(0F21F359-AB84-41E8-9A78-36D110E6D2F9,
            Wrox.ProCSharp.COMInterop.Server.IWelcome)
    ]
    interface IWelcome : IDispatch {
        [id(0x0000ea88)]
        HRESULT Greeting([in] BSTR name, [out, retval] BSTR* pRetVal);
    };

    [
        odl,
        uuid(EF596F3F-B69B-3657-9D48-C906CBF12565),
        version(1.0),
        oleautomation,
        custom(0F21F359-AB84-41E8-9A78-36D110E6D2F9,
            Wrox.ProCSharp.COMInterop.Server.IMath)
    ]
    interface IMath : IUnknown {
        HRESULT _stdcall Add([in] long val1, [in] long val2,
            [out, retval] long* pRetVal);
        HRESULT _stdcall Sub([in] long val1, [in] long val2,
            [out, retval] long* pRetVal);
    };

    [
        uuid(77839717-40DD-4876-8297-35B98A8402C7),
        version(1.0),
        custom(0F21F359-AB84-41E8-9A78-36D110E6D2F9,
            Wrox.ProCSharp.COMInterop.Server.DotnetComponent)
    ]
    coclass DotnetComponent {
```

```

interface _Object;
[default] interface IWelcome;
interface IMath;
};
};

```

### 24.4.5 COM 注册

在.NET 组件用作 COM 对象之前,需要在注册表中配置它。另外,如果不希望把程序集复制到客户程序所在的目录下,还需要把程序集安装在全局程序集缓存中。全局程序集缓存详见第 17 章。

要把程序集安装在全局程序集缓存中,必须用强名标识它(使用 VS2008,可以在解决方案的属性中定义强名),然后在全局程序集缓存中注册程序集:

```
gacutil -i dotnetcomponent.dll
```

现在可以使用 regasm 实用工具在注册表中配置组件了。选项/tlb 可以提取类型库,在注册表中配置类型库:

```
regasm dotnetcomponent.dll /tlb
```

下面将列出写入注册表中的.NET 组件信息。所有的 COM 信息都在 HKEY\_CLASSES\_ROOT(HKCR)选项中。进入这个选项,写入 progid 的键(在本例中,是 Wrox.DotNetComponent)和 CLSID。

键 HKCR\CLSID\{CLSID}\InProcServer32 有如下选项:

- mscoree.dll: 它表示 CCW。这是一个真正的 COM 对象,负责存储.NET 组件。这个 COM 对象访问.NET 组件,为客户程序提供 COM 操作。通过一般的 COM 实例化机制从客户程序上加载和实例化文件 mscoree.dll。
- ThreadingModel=Both: 这是 mscoree.dll COM 对象的一个特性。这个组件以支持 STA 和 MTA 的方式编写。
- Assembly= DotNetComponent, Version=1.0.0.0, Culture=neutral, PublicKeyToken = 5cd57c93b4d9c41a: Assembly 的值存储了程序集的全名,其中包括版本号和公钥令牌,所以程序集可以唯一地识别出来。这里注册的程序集将通过 mscoree.dll 加载。
- Class=Wrox.ProCSharp.COMInterop.Server.DotNetComponent: 类名也由 mscoree.dll 使用。这是要被实例化的类。
- RuntimeVersion=v1.1.4322: 注册项 RuntimeVersion 指定要用于存储.NET 程序集的.NET 运行库的版本。

除了这里列出的配置之外,所有的接口和类型库也用它们的标识符配置。

### 24.4.6 创建 COM 客户程序

现在该创建 COM 客户程序了。首先创建一个简单的 C++ Win32 控制台应用程序项目,命名为 COMClient。可以在项目向导中选择默认的选项,按下 Finish。

在文件 COMClient.cpp 的开头,添加一个预处理器命令,以包含<iostream>头文件,导入



为.NET 组件创建的类型库。导入语句创建了一个“智能指针”类，这样更容易处理 COM 对象。在建立过程中，导入语句会创建.tlb 和.tli 文件，它们位于项目的 debug 目录下，包含了智能指针类。接着添加 using namespace 指令，打开命名空间 std 和 DotNetComponent，std 用于把输出消息写到控制台上，DotNetComponent 是在智能指针类中创建的。

```
//COMClient.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include <iostream>
#import "../DotNetComponent/bin/debug/DotnetComponent.tlb"

using namespace std;
using namespace DotnetComponent;
```

在\_tmain()方法中，在进行任何 COM 调用之前，要先用 API 调用 CoInitialize()实例化 COM。CoInitialize()会为线程创建 STA 并进入 STA。变量 spWelcome 是 IWelcome 类型的智能指针，智能指针方法 CreateInstance()的参数是 progid，它使用 COM API CoCreateInstance()创建 COM 对象。运算符->用智能指针重写了，这样就可以调用 COM 对象的方法，如 Greeting()。

```
int _tmain(int argc, _TCHAR* argv[])
{
    HRESULT hr;
    hr = CoInitialize(NULL);

    try
    {
        IWelcomePtr spWelcome;
        hr = spWelcome.CreateInstance("Wrox.DotnetComponent"); // CoCreateInstance()

        cout<< spWelcome->Greeting("Bill")<<endl;
    }
```

.NET 组件支持的第二个接口是 IMath，它也有一个封装 COM 接口的智能指针 IMathPtr。可以直接把一个智能指针赋予另一个智能指针，例如 spMath=spWelcome；在智能指针的实现代码(重写=运算符)中，执行了 QueryInterface()方法。使用 IMath 接口的引用可以调用 Add()方法。

```
IMathPtr spMath;
spMath = spWelcome; // QueryInterface()

long result = spMath->Add(4,5);
cout<<" result: "<< result <<endl;
}
```

如果 COM 对象返回一个 HRESULT 错误值(如果.NET 组件生成异常，返回 HRESULT 的 CCW 就会返回一个错误值)，智能指针就封装 HRESULT 错误，并生成\_com\_error 异常。错误在 catch 块中处理。在程序的最后，使用 CoUninitialize()关闭和卸载 COM DLL。

```
catch (_com_error& e)
{
    cout<<e.ErrorMessage()<<endl;
}

CoUninitialize();
```

```
return 0;
}
```

现在可以运行应用程序了，在控制台上得到 Greeting() 和 Add() 方法的输出。还可以试着调试智能指针类，在这里可以看到直接调用了 COM API。

#### 提示：

如果得到一个组件找不到的异常，就应检查是否在全局程序集缓存中安装了在注册表中配置的程序集的同一版本。

### 24.4.7 添加连接点

在 .NET 组件中添加对 COM 事件的支持，需要对 .NET 类的实现代码作一些修改。提供 COM 事件并不是简单地使用 event 和 delegate 关键字，还需要添加更多的 COM 互操作特性。

首先需要给 .NET 项目添加另一个接口 IMathEvents。这个接口是组件的源或输出接口，由客户程序中的 sink 对象执行。源接口必须是分派接口或定制接口。脚本客户程序只支持分派接口。分派接口通常优先于源接口。

```
[InterfaceType(ComInterfaceType.InterfaceIsIDispatch)]
[ComVisible(true)]
public interface IMathEvents
{
    [DispId(46200)] void CalculationCompleted();
}
```

接着，添加一个委托。委托必须与输出接口中的方法有相同的签名和返回类型。如果源接口中有多个方法，每个方法的参数都有所不同，就必须为每个方法都指定一个委托。因为 COM 客户程序不需要直接访问这个委托，所以委托可以用特性 [ComVisible(false)] 标记：

```
[ComVisible(false)]
public delegate void CalculationCompletedDelegate();
```

在类 DotNetComponent 中，必须指定源接口。这可以使用特性 [ComSourceInterfaces] 来实现。添加特性 [ComSourceInterface]，像前面那样指定输出接口。可以用特性类的不同构造函数添加多个接口，但是支持多个源接口的唯一客户程序语言是 C++。VB6 客户程序仅支持一个源接口。

```
[ClassInterface(ClassInterfaceType.None)]
[ProgId("Wrox.DotnetComponent")]
[Guid("77839717-40DD-8297-35B98A8402C7")]
[ComSourceInterfaces(typeof(IMathevents))]
[ComVisible(true)]
public class DotnetComponent : IWelcome, IMath
{
    public DotnetComponent()
    {
    }
}
```

在类 DotNetComponent 中，必须为源接口的每个方法声明一个事件。方法的类型必须是委托名，事件名必须是源接口中的方法名。可以给 Add() 方法和 Sub() 方法添加事件调用。这一步

是调用事件的正常.NET 方式, 详见第 7 章。

```
public event CalculationCompletedDelegate CalculationCompleted ;
```

```
public int Add(int val1, int val2)
{
```

```
    int result = val1 + val2;
    if (CalculationCompleted != null)
        CalculationCompleted();
    return result;
}
```

```
public int Sub(int val1, int val2)
{
```

```
    int result = val1 - val2;
    if (CalculationCompleted != null)
        CalculationCompleted();
    return result;
}
```

**提示:**

事件名必须是源接口的方法名, 否则就不能为 COM 客户程序映射事件。

#### 24.4.8 用 sink 对象创建客户程序

在建立和注册.NET 程序集, 把它安装到全局程序集缓存中后, 就可以使用事件源建立客户程序了。这次我们使用 VB6 编写一个实现 IDispatch 接口的回调或 sink 对象。为此只需添加 WithEvents 关键字, 与目前 VB 处理.NET 事件的方式相同。使用 C++所需的工作较多, 但这里可以使用 Active Template Library。

打开前面创建的 C++控制台应用程序, 在 stdafx.h 文件中添加如下 include 语句:

```
#include <atlbase.h>
extern CComModule _Module;
#include <atlcom.h>
```

文件 stdafx.cpp 需要包含 ATL 执行文件 atlimpl.cpp:

```
#include <atlimpl.cpp>
```

给 COMClient.cpp 文件添加新类 CEventHandler, 这个类包含组件调用的 IDispatch 接口的执行代码。IDispatch 接口是由基类 IDispEventImpl 实现的。这个类读取类型库, 把方法和参数的分派 ID 匹配到类的方法上。IDispEventImpl 类的模板参数有 sink 对象的 ID (这里使用 ID 4)、执行回调方法的类 (CEventHandler)、回调接口的接口 ID (DIID\_IMathEvents)、类型库的 ID (LIBID\_DotnetComponent) 和类型库的版本号。指定的 ID (DIID\_IMathEvents 和 LIBID\_DotnetComponent) 在#import 语句创建的文件 dotnet component.tlh 中。

BEGIN\_SINK\_MAP 和 END\_SINK\_MAP 封装的 sink 映射定义了由 sink 对象执行的方法。SINK\_ENTRY\_EX 把方法 OnCalcCompleted 映射到分派 ID 46200 上。这个分派 ID 用.NET 组件中 IMathEvents 接口的 CalculationCompleted 方法定义。

```
class CEventHandler : public IDispEventImpl < 4, CEventHandler,
```

```

& DIID_IMathEvents, & LIBID_DotnetComponent, 1, 0 >
{
    public:
    BEGIN_SINK_MAP(CEventHandler)
        SINK_ENTRY_EX(4, DIID_IMathEvents, 46200, OnCalcCompleted)
    END_SINK_MAP()
    HRESULT __stdcall OnCalcCompleted()
    {
        cout << "calculation completed" << endl;
        return S_OK;
    }
};

```

现在, 主方法需要修改, 把事件 sink 对象推荐给组件, 这样组件才能回调到 sink 中。为此需要使用 CEventHandler 类的 DispEventAdvise() 方法, 并传送一个 IUnknown 接口指针。方法 DispEventUnadvise() 再注销 sink 对象。

```

int _tmain(int argc, _TCHAR* argv[])
{
    HRESULT hr;
    hr = CoInitialize(NULL);

    try
    {
        IWelcomePtr spWelcome;
        hr = spWelcome.CreateInstance("Wrox.DotnetComponent");

        IUnknownPtr spUnknown = spWelcome;

        cout << spWelcome->Greeting("Isabella") << endl;

        CEventHandler* eventHandler = new CEventHandler();
        hr = eventHandler->DispEventAdvise(spUnknown);

        IMathPtr spMath;
        spMath = spWelcome; // QueryInterface()

        long result = spMath->Add(4, 5);
        cout << "result:" << result << endl;

        eventHandler->DispEventUnadvise(spWelcome.GetInterfacePtr());
        delete eventHandler;
    }
    catch (_com_error & e)
    {
        cout << e.ErrorMessage() << endl;
    }

    CoUninitialize();
    return 0;
}

```

#### 24.4.9 在 Internet Explorer 中运行 Windows 窗体控件

Windows 窗体控件在 Internet Explorer 中可以用作 ActiveX 控件。因为有许多不同的 ActiveX 控件容器, 所有这些容器对 ActiveX 控件有不同的要求, 所以 Microsoft 不支持在任何容器中包含 Windows 窗体控件。支持的容器只有 Internet Explorer 和 MFC 容器(MFC 容器最早在 VS.NET 2003

中获得支持)。但在 MFC 容器中, 必须手工修改代码, 以包含 MFC 应用程序中的 ActiveX 控件。

要在 Internet Explorer 中包含 Windows 窗体控件, 必须把程序集文件复制到 Web 服务器上, 在 HTML 页面上添加一些该控件的信息。为了支持 Windows 窗体控件, 应扩展<object>标记语法。使用特性 classid, 可以添加程序集文件和类名, 它们用符号#分隔开:

```
classid="<assembly file>#class name".
```

对于程序集文件 ControlDemo.dll 和命名空间 Wrxo.ProCSharp.COMInterop 中的类 UserControl1, 语法如下所示:

```
<object id="myControl"
  classid = "ControlDemo.dll#Wrox.ProCSharp.COMInterop.UserControl1"
  height="400" width="400">
</object>
```

只要用户打开了 HTML 页面, 程序集就会下载到客户程序系统中。程序集存储在下载的程序集缓存中, 每次用户访问页面时, 都会重新检查版本号。如果版本号没有改变, 就使用本地缓存中的程序集。

**提示:**

要在 Web 页面中使用 Windows 窗体控件, 客户机必须安装了 .NET 运行库, 必须使用 Internet Explorer 5.5 或更高版本, 安全设置必须允许下载程序集。

## 24.5 平台调用

并不是 Windows API 调用的所有特性都可用于 .NET Framework。不仅旧 Win32 API 调用是这样, Windows Vista 和 Windows Server 2008 中的新特性也是如此。也许读者编写过一些导出非托管方法的 DLL, 希望在 C# 中使用它们。

**提示:**

附录 C 介绍了一些 Windows Vista 和 Windows Server 2008 的专用特性。

要重用一個非托管的库, 它不包含 COM 对象, 只包含导出的函数, 就可以使用平台调用服务。通过这个服务, CLR 会加载包含所需函数的 DLL, 并编组参数。

要使用非托管的函数, 首先必须找到要导出的函数名。为此, 可以使用 dumpbin 工具及其 /exports 选项。

例如, 下面的命令:

```
dumpbin /exports c:\windows\system32\kernel32.dll | more
```

列出了 DLL 文件 kernel32.dll 中的所有导出函数。在这个例子中, 使用 Win32 API 函数 CreateHardLink() 创建与一个已有文件的硬链接。在这个 API 调用中, 可以有几个文件名引用同一个文件, 只要这些文件名在同一个硬盘上即可。这个 API 调用不能用于 .NET Framework 3.5, 所以必须使用平台调用服务。

要调用内置的函数, 必须定义一个 C# 外部方法, 它的参数个数必须与内置函数相同, 用非



托管方法定义参数类型必须对应于用托管代码映射的类型。

Windows API 调用 CreateHardLink()在 C++中的定义如下所示：

```
BOOL CreateHardLink(  
    LPCTSTR lpFileName,  
    LPCTSTR lpExistingFileName,  
    LPSECURITY_ATTRIBUTES lpSecurityAttributes);
```

现在，这个定义必须映射为.NET 数据类型。非托管代码的返回类型是 BOOL，它会映射为 bool 数据类型。LPCTSTR 定义一个指向 const 字符串的 long 指针。Windows API 给数据类型使用 Hungarian 命名约定。LP 是一个 long 指针，C 是常量，STR 是非空字符串。T 把类型标记为泛型类型，根据编译器设置，类型解析为 LPCSTR (ANSI 字符串)或 LPWSTR (宽 Unicode 字符串)。C 字符串映射为.NET 类型 String。LPSECURITY\_ATTRIBUTES 是一个指向 SECURITY\_ATTRIBUTES 类型的结构的 long 指针。我们把 NULL 传送给这个参数，所以可以把这个类型映射为 IntPtr。这个方法 C#声明必须用 extern 修饰符来标记，因为它在 C#代码中没有执行代码。其执行代码在 DLL 文件 kernel32.dll 中，这是用[DllImport]属性引用的。.NET 声明 CreateHardLink()的返回类型是 bool，内置方法 CreateHardLink()返回 BOOL，所以需要额外说明一下。C++有不同的 Boolean 数据类型，例如内置的 bool 和 Windows 定义的 BOOL，它们有不同的值，所以属性[MarshalAs]指定.NET 类型 bool 应映射到什么内置类型上。

```
[DllImport("kernel32.dll", SetLastError="true",  
    EntryPoint="CreateHardLink", CharSet=CharSet.Auto)]  
[return: MarshalAs(UnmanagedType.Bool)]  
public static extern bool CreateHardLink(string fileName,  
    string existingFilename, IntPtr securityAttributes);
```

表 24-3 列出了可以用[DllImport]属性指定的设置。

表 24-3	
DllImport 属性或字段	说 明
EntryPoint	可以给函数的 C#声明指定一个不同于非托管库中的名称。在非托管库中，方法的名称在 EntryPoint 字段中定义
CallingConvention	根据编译器和用于编译非托管函数的编译器设置，可以使用不同的调用约定。调用约定定义了参数的处理方式和它们在堆栈中的位置。可以通过设置一个可枚举的值，来定义调用约定。  Windows API 通常在 Windows 操作系统上使用 stdCall 调用约定，在 Windows CE 上使用 Cdecl 调用约定。把值设置为 CallingConvention.Winapi，可以使 Windows API 在 Windows 和 Windows CE 上工作
CharSet	字符串参数可以是 ANSI 或 Unicode。使用 CharSet 设置，可以确定如何管理字符串。用 CharSet 枚举定义的值有 Ansi, Unicode 和 Auto。CharSet.Auto 在 Windows NT 平台上使用 Unicode，在 Windows 98 和 Windows ME 上使用 ANSI
SetLastError	如果非托管的函数使用 Windows API SetLastError 设置一个错误，就可以把 SetLastError 字段设置为 true。这样，就可以在以后使用 Marshal.GetLastWin32Error()读取错误号

为了在.NET 环境中更方便地使用 CreateHardLink()方法, 应遵循如下规则:

- 创建一个内部类 NativeMethods, 它封装了平台调用服务的方法调用
- 创建一个公共类, 为.NET 应用程序提供内置方法功能
- 使用安全属性标记必要的安全性

在示例代码中, 类 FileUtility 中的公共方法 CreateHardLink()就是.NET 应用程序使用的方法。与内置 Windows API 方法 CreateHardLink()相比, 公共方法 CreateHardLink()将文件名参数的顺序倒转了。第一个参数是已有文件的名称, 第二个参数是新文件的名称。这类似于 Framework 中的其他类, 例如 File.Copy()。第三个参数给新文件名传送安全特性, 但在这里的执行代码中没有使用它, 所以公共方法只有两个参数。返回类型也修改了, 不是通过返回值 false 来返回一个错误, 而是抛出一个异常。在出现错误时, 非托管方法 CreateHardLink()会使用非托管的 API SetLastError()设置错误号。要从.NET 中读取这个值, [DllImport]字段 SetLastError 应设置为 true。在托管方法 CreateHardLink()中, 通过调用 Marshal.GetLastWin32Error()来读取错误号。要从这个号码中创建错误消息, 应使用 System.ComponentModel 命名空间中的 Win32Exception 类。这个类通过构造函数来接收错误号, 返回一个本地化的错误消息。在出现错误时, 抛出一个 IOException 类型的异常, 它有一个 Win32Exception 类型的内部异常。公共方法 CreateHardLink()使用了属性 FileIOPermission, 来确定调用者是否有必要的权限。.NET 安全性的内容详见第 20 章。

```
using System;
using System.Runtime.InteropServices;
using System.ComponentModel;
using System.IO;

namespace Wrox.ProCSharp.Interop
{
    internal static class NativeMethods
    {
        [DllImport("kernel32.dll", SetLastError=true,
            EntryPoint="CreateHardLink", CharSet=CharSet.Unicode)]
        [return: MarshalAs(UnmanagedType.Bool)]
        private static extern bool CreateHardLink(
            string newFileName, string existingFileName,
            IntPtr securityAttributes);

        internal static void CreateHardLink(string oldFileName,
            string newFileName)
        {
            if (!CreateHardLink(newFileName, oldFileName, IntPtr.Zero))
            {
                Win32Exception ex = new Win32Exception(
                    Marshal.GetLastWin32Error());
                throw new IOException(ex.Message, ex);
            }
        }
    }

    public static class FileUtility
    {
        [FileIOPermission(SecurityAction.LinkDemand, Unrestricted=true)]
        public static void CreateHardLink(string oldFileName,
            string newFileName)
        {
            NativeMethods.CreateHardLink(oldFileName, newFileName);
        }
    }
}
```

```

        {
            NativeMethods.CreateHardLink(oldFileName, newFileName);
        }
    }
}

```

这个类现在可以用于创建硬链接了。如果 file1.txt 文件不存在，就会得到一个异常和信息“系统找不到指定的文件”。如果该文件存在，就会得到一个新的文件名，它引用原来的文件。很容易验证这一点：修改一个文件中的文本，该文本也会出现在另一个文件中。

```

static void Main()
{
    try
    {
        FileUtility.CreateHardLink("file1.txt", "file2.txt");
    }
    catch (IOException ex)
    {
        Console.WriteLine(ex.Message);
    }
}

```

在内置方法调用中，常常需要使用 Windows 句柄。Windows 句柄是一个 32 位值，它取决于句柄类型，一些值不允许使用。在 .NET 1.0 中，句柄通常使用 IntPtr 结构，因为可以用这个结构设置任意可能的 32 位值。但是，在一些句柄类型中，这会导致安全问题，以及线程竞态条件，在最后的清理阶段泄露句柄。因此 .NET 2.0 引入了 SafeHandle 类。这是每个 Windows 句柄的抽象基类。Microsoft.Win32.SafeHandles 命名空间中的派生类有 SafeHandleZeroOrMinusOneIsInvalid 和 SafeHandleMinusOneIsInvalid。顾名思义，这些类不接受无效的 0 或 -1 值。更进一步的派生句柄类型有 SafeFileHandle、SafeWaitHandle、SafeNCryptHandle 和 SafePipeHandle，它们可以由特定的 Windows API 调用使用。

例如，要映射 Windows API CreateFile()，可以使用下面的声明返回一个 SafeFileHandle。当然，也可以使用 .NET 类 File 和 FileInfo。

```

[DllImport("Kernel32.dll", SetLastError = true,
    CharSet = CharSet.Unicode)]
internal static extern SafeFileHandle CreateFile(
    string fileName,
    [MarshalAs(UnmanagedType.U4)] FileAccess fileAccess,
    [MarshalAs(UnmanagedType.U4)] FileShare fileShare,
    IntPtr securityAttributes,
    [MarshalAs(UnmanagedType.U4)] FileMode creationDisposition,
    int flags,
    SafeFileHandle template);

```

**提示：**

第 22 章介绍了如何创建定制的 SafeHandle 类，来处理 Windows Vista 中的事务文件 API。

## 24.6 小结

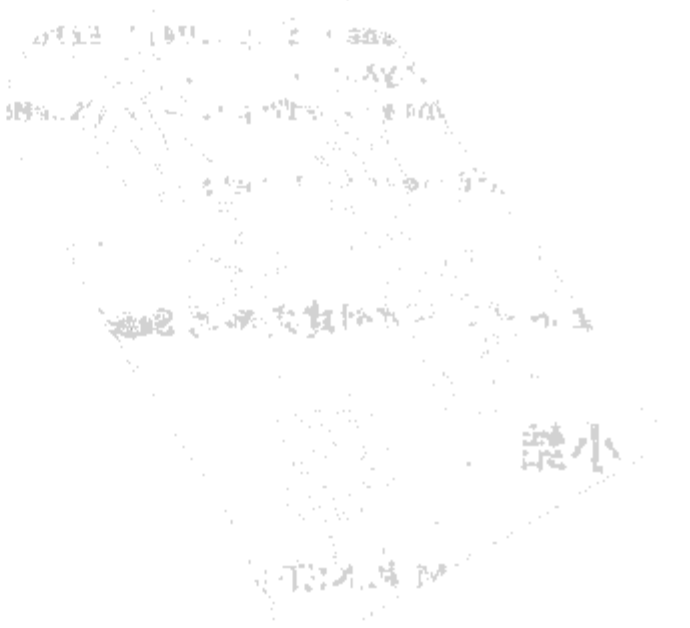
本章介绍了 COM 和 .NET 应用程序交互的不同实现方式。无需重新编写应用程序和组件，

就可以在.NET 应用程序中像使用.NET 类那样使用 COM 组件。实现这个功能的工具是 `tlbimp`，它创建了 RCW，在.NET 的后面隐藏了 COM 对象。

同样，`tlbexp` 在.NET 组件中创建了一个由 CCW 使用的类型库，CCW 将.NET 组件隐藏在 COM 的后面。要把.NET 类用作 COM 组件，就必须使用 `System.Runtime.InteropServices` 命名空间中的一些特性，以定义 COM 客户程序需要的具体 COM 特性。

利用平台调用服务，可以使用 C#调用内部方法。平台调用服务需要用 C#和.NET 数据类型重新定义内部方法。定义了映射后，就可以像 C#方法那样调用内部方法了。进行交互操作的另一个方法是使用 It Just Works (IJW) with C++/CLI 技术。C++/CLI 详见附录 B。

本书的下一部分介绍数据。下一章讨论如何访问文件系统，后面的章节论述如何读写数据库，处理 XML。



## 第Ⅳ部分

# 数 据

- 第 25 章 文件和注册表操作
- 第 26 章 .NET 数据访问
- 第 27 章 LINQ to SQL
- 第 28 章 处理 XML
- 第 29 章 LINQ to XML
- 第 30 章 .NET 编程和 SQL Server



# 第25章

## 文件和注册表操作

本章将介绍如何在 C# 中执行读写文件和系统注册表的任务。主要内容如下：

- 介绍目录结构，确定其中有哪些文件和文件夹，并介绍它们的属性。
- 移动、复制和删除文件和文件夹
- 读写文本文件
- 读写注册表键
- 读写独立存储器

Microsoft 提供了非常直观的对象模型，这些模型包括了所有这些领域。本章还将介绍如何使用 .NET 基类执行上面的任务。对于文件系统操作，相关的类都在 `System.IO` 命名空间中，而注册表操作由 `System.Win32` 命名空间中的类来执行。

注意：

.NET 基类也包含 `System.Runtime.Serialization` 命名空间中的许多类和接口，它们都与串行化有关。串行化是把一些数据(例如，文档的内容)转换为字节流并存储在某个地方的过程。本章不讨论这些类，而主要讨论可直接访问文件的类。

注意，在修改文件或注册表项目时，安全性显得更为重要。第 20 章介绍了安全性的各个方面。但在本章中，仅假定用户有足够的访问权限运行修改文件或注册表项目的所有示例，如果在拥有管理权限的账户下运行，就是这种情况。

### 25.1 管理文件系统

图 25-1 中的类可以用于浏览文件系统和执行操作，例如移动、复制和删除文件。这些类的作用是：

- `System.MarshalByRefObject`——.NET 类中用于远程操作的基对象类，允许在应用程序域之间编组数据。
- `FileSystemInfo`——表示任何文件系统对象的基类。
- `FileInfo` 和 `File`——表示文件系统上的文件。
- `DirectoryInfo` 和 `Directory`——表示文件系统上的文件夹。
- `Path`——这个类包含的静态成员可以用于处理路径名。
- `DriveInfo`——它的属性和方法提供了指定驱动器的信息。

注意:

在 Windows 上, 包含文件并用于组织文件系统的对象称为文件夹。例如, 在路径 C:\My Documents\ReadMe.txt 中, ReadMe.txt 是一个文件, My Documents 是一个文件夹。文件夹是一个 Windows 专用的术语: 在其他操作系统上, 用术语“目录”代替文件夹, Microsoft 为了使 .NET 具有平台无关性, 对应的 .NET 基类都称为 Directory 和 DirectoryInfo。因为它有可能与 LDAP 目录(详见第 46 章)混淆, 而且本书与 Windows 有关, 所以本章仍使用文件夹。

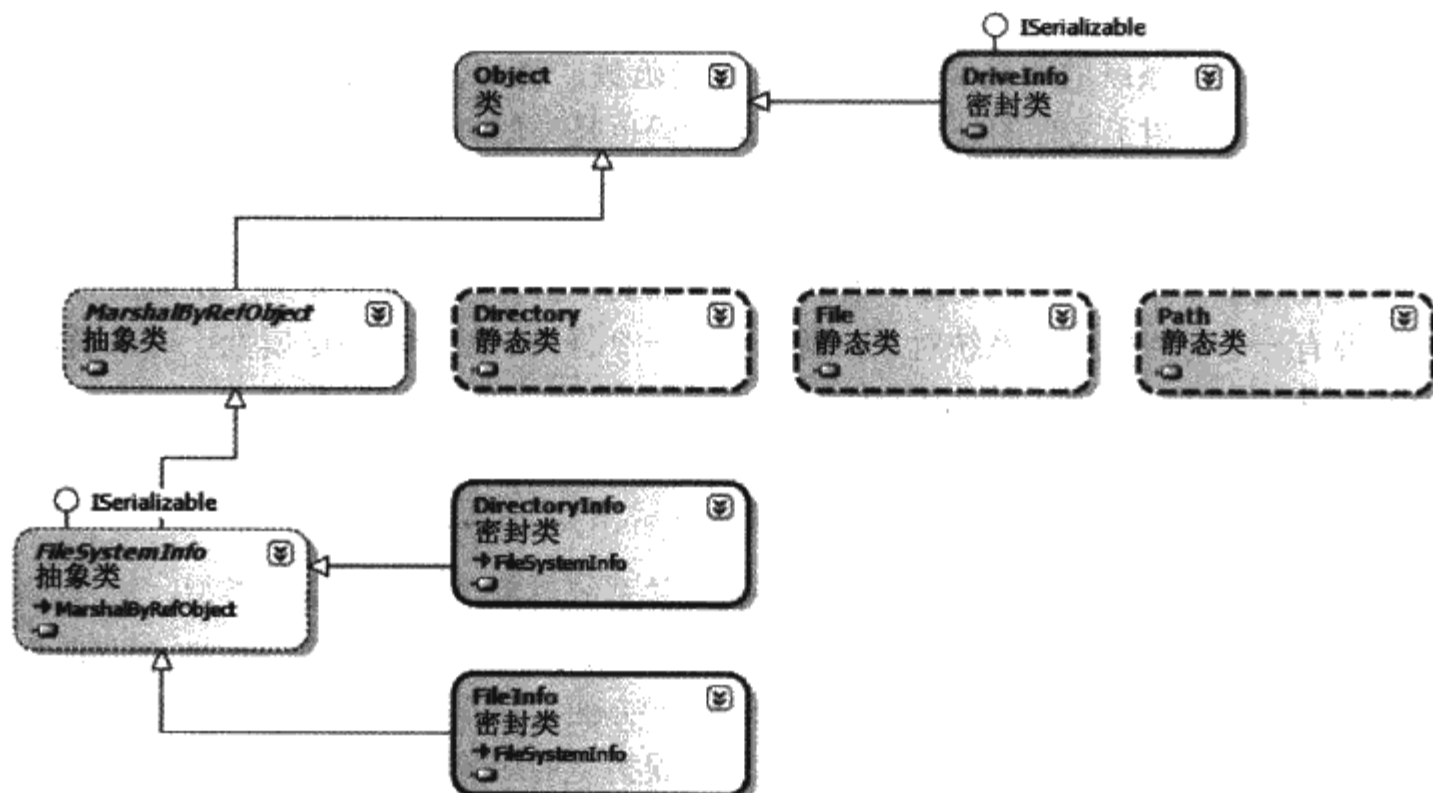


图 25-1

### 25.1.1 表示文件和文件夹的 .NET 类

注意, 上面的列表有两个表示文件夹的类, 和两个表示文件的类。使用哪个类主要依赖于访问该文件夹或文件的次数:

- Directory 和 File 只包含静态方法, 不能被实例化。只要调用一个成员方法, 提供合适文件系统对象的路径, 就可以使用这些类。如果只对文件夹或文件执行一个操作, 使用这些类就很有效, 因为这样可以省去实例化 .NET 类的系统开销。
- DirectoryInfo 和 FileInfo 执行与 Directory 和 File 大致相同的公共方法, 并拥有一些公共属性和构造函数, 但它们都是有状态的, 并且这些类的成员都不是静态的。需要实例化这些类, 并把每个实例与特定的文件夹或文件关联起来。如果使用同一个对象执行多个操作, 使用这些类就比较有效, 因为在构造时它们将读取合适文件系统对象的身份验证和其他信息, 无论对每个对象(类实例)调用了多少方法, 都不需要再次读取这些信息。比较而言, 在调用每个方法时, 相应的无状态类需要再次检查文件或文件夹的内容。

本节主要使用 FileInfo 和 DirectoryInfo 类, 但我们调用的许多方法(不是全部)也可以由 File 和 Directory 执行(但这些方法需要一个额外的参数——文件系统对象的路径名, 这两个方法的

名称略有不同)。例如：

```
FileInfo myFile = new FileInfo(@"C:\Program Files\My Program\ReadMe.txt");
myFile.CopyTo(@"D:\Copies\ReadMe.txt");
```

与下面的代码有相同的效果：

```
File.Copy(@"C:\Program Files\My Program\ReadMe.txt", @"D:\Copies\ReadMe.txt");
```

第一个代码段执行的时间略长，因为需要实例化一个 FileInfo 对象 MyFile，但 MyFile 可以对同一个文件执行进一步的操作。第二个示例不需要实例化对象来复制文件。

把包含对应文件系统的路径字符串传递给构造函数，就可以实例化 FileInfo 或 DirectoryInfo 类。刚才已经介绍了文件的处理，文件夹的代码也是类似的：

```
DirectoryInfo myFolder = new DirectoryInfo(@"C:\Program Files");
```

如果路径表示一个不存在的对象，在构造时不会抛出异常。但如果是第一次调用方法，而该方法需要有一个对应的文件系统对象，就会抛出一个异常。检查 Exists 属性，可以确定对象是否存在，其类型是否合适，FileInfo 和 DirectoryInfo 类都会执行该属性：

```
FileInfo test = new FileInfo(@"C:\Windows");
Console.WriteLine(test.Exists.ToString());
```

注意，对于这个属性，要返回 true，对应的文件系统对象必须是合适的类型。换言之，如果实例化了一个 FileInfo 对象，该对象提供了文件夹的路径，或者实例化了 DirectoryInfo 对象，并给它提供了文件路径，Exists 的值就是 false。另一方面，如果可能，这些对象的大多数属性和方法都会返回一个值——它们不会因为调用了错误类型的对象而抛出异常，除非要求它们完成一些不可能的任务。例如，上面的代码段会先显示 false(因为 C:\Windows 是一个文件夹)，但接着就会显示创建文件夹的时间——因为文件夹仍拥有该信息。另一方面，如果使用 FileInfo.Open()方法，以打开文件的方式打开文件夹，就会产生一个异常。

在确定了是否存在对应的文件系统对象后，就可以(如果使用 FileInfo 或 DirectoryInfo 类)使用许多属性来确定该对象的信息，这些属性如表 25-1 所示。

表 25-1

名 称	作 用
CreationTime	创建文件或文件夹的时间
DirectoryName (仅用于 FileInfo)	包含文件夹的完整路径名
Parent (仅用于 DirectoryInfo)	指定子目录的父目录
Exists	文件或文件夹是否存在
Extension	文件的扩展名，对于文件夹则返回空白
FullName	文件或文件夹的完整路径名
LastAccessTime	最后一次访问文件或文件夹的时间
LastWriteTime	最后一次修改文件或文件夹的时间
Name	文件或文件夹的名称

(续表)

名 称	作 用
Root(仅用于 DirectoryInfo)	路径的根部分
Length(仅用于 FileInfo)	返回文件的大小(字节)

也可以使用表 25-2 所示的方法对文件系统对象执行操作。

表 25-2

名 称	作 用
Create()	创建给定名称的文件夹或空文件。对于 FileInfo，该方法会返回一个流对象，以便写入文件。本章后面讨论流
Delete()	删除文件或文件夹。对于文件夹，有一个可以递归的 Delete 选项
MoveTo()	移动和/或重命名文件或文件夹
CopyTo()	(只适用于 FileInfo)复制文件，注意文件夹没有复制方法，如果复制完整的目录树，需要单独复制每个文件，创建对应于旧文件夹的新文件夹
GetDirectories()	(只适用于 DirectoryInfo) 返回 DirectoryInfo 对象数组，该数组表示文件夹中包含的所有文件夹
GetFiles()	(只适用于 DirectoryInfo) 返回 FileInfo 对象数组，该数组表示文件夹中包含的所有文件
GetFileSystemObjects()	(只适用于 DirectoryInfo) 返回 FileInfo 和 DirectoryInfo 对象，它把文件夹中包含的所有对象表示为一个 FileSystemInfo 引用数组

注意，表 25-2 给出了主要的属性和方法，但没有列出所有的属性和方法。

注意：

在表 25-2 中，没有列出读写文件数据的大多数属性和方法。读写文件数据实际上是使用流对象完成的，本章后面会介绍流对象。FileInfo 也可以执行 Open()、OpenRead()、OpenText()、OpenWrite()、Create()和 CreateText()等方法，它们都返回流对象。

有趣的是，创建时间、最后一次访问时间和最后一次写入时间都是可写入的。

```
// displays the creation time of a file, then changes it and displays it
// again
FileInfo test = new FileInfo(@"C:\MyFile.txt");
Console.WriteLine(test.Exists.ToString());
Console.WriteLine(test.CreationTime.ToString());
test.CreationTime = new DateTime(2008, 1, 1, 7, 30, 0);
Console.WriteLine(test.CreationTime.ToString());
```

运行这个应用程序，结果如下所示：

```
True
2/5/2007 2:59:32 PM
```

1/1/2008 7:30:00 AM

能手工修改这些属性看起来很奇怪，但相当有效。例如，如果有一个程序可以通过读取、删除文件来有效地修改文件，用新内容创建新文件，则可以修改创建日期，匹配旧文件的最初创建日期。

25.1.2 Path 类

我们不能实例化 Path 类。它有一些静态方法，可以更容易地对路径名执行操作。例如，假定要显示文件夹 C:\My Documents 中 ReadMe.txt 文件的完整路径，可以用下述代码查找文件的路径：

```
Console.WriteLine(Path.Combine(@"C:\My Documents", "ReadMe.txt"));
```

使用 Path 类要比手工处理各个符号容易得多。因为 Path 类在处理不同操作系统上的路径名时，要使用不同的格式。在编写本书时，Windows 是 .NET 唯一支持的操作系统，但如果 .NET 以后要移植到 Unix 上，Path 就要处理 Unix 路径，Unix 把/(并不是\ )用作路径名中的分隔符。Path.Combine()是这个类常常使用的一个方法，Path 也执行其他方法，提供路径的信息，或者以要求的格式显示信息。

表 25-3 列出了 Path 类的一些属性。

表 25-3

属 性	说 明
AltDirectorySeparatorChar	提供一种与平台无关的方式，来指定分隔目录级别的另一个字符。在 Windows 上使用/符号，在 UNIX 上使用\符号
DirectorySeparatorChar	提供一种与平台无关的方式，来指定分隔目录级别的另一个字符。在 Windows 上使用/符号，在 UNIX 上使用\符号
PathSeparator	提供一种与平台无关的方式，来指定划分环境变量的路径字符串，默认为分号
VolumeSeparatorChar	提供一种与平台无关的方式，来指定容量分隔符，默认为冒号

下一节用一个示例来说明如何浏览目录，查看文件的属性。

25.1.3 示例：文件浏览器

本节要创建一个 C#示例应用程序 FileProperties。该程序显示了一个简单的用户界面，可以浏览文件系统，查看文件的创建时间、最后一次访问时间，最后一次写入时间和文件的大小。这个应用程序的代码可以从 Wrox 网站上 [www.wrox.com](http://www.wrox.com) 下载。

FileProperties 应用程序如图 25-2 所示。在窗口顶部的主文本框中输入文件夹或文件的名称，再单击 Display 按钮。如果键入了文件夹的路径，其内容就显示在列表框中。如果键入文件的路径，其信息就显示在窗体底部的文本框中，父文件夹的内容则显示在列表框中。屏幕图 25-2 显示了使用 FileProperties 查看一个文件夹的情况。



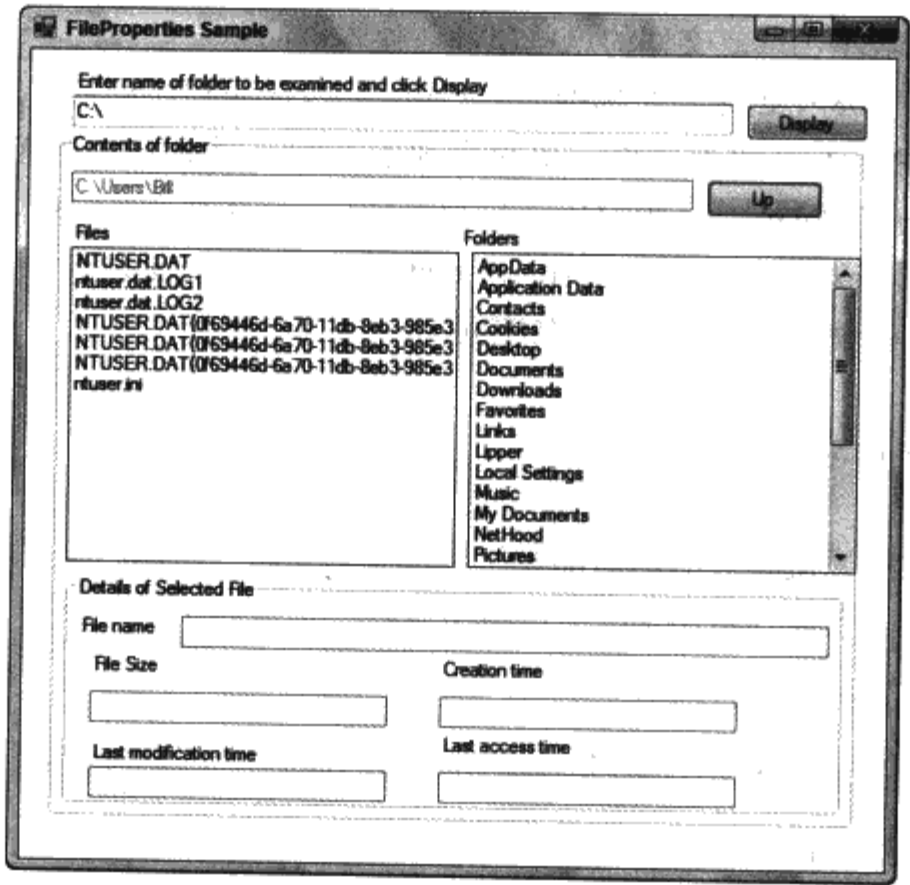


图 25-2

在浏览文件系统时，单击右边列表框中的任何一个文件夹，就可以查看它下面的文件夹，或者单击 Up 按钮，查看其父文件夹。图 25-2 显示了 My Documents 文件夹的内容。用户还可以单击列表框中的一个文件名，选择该文件。此时其属性就会显示在应用程序底部的文本框中，如图 25-3 所示。

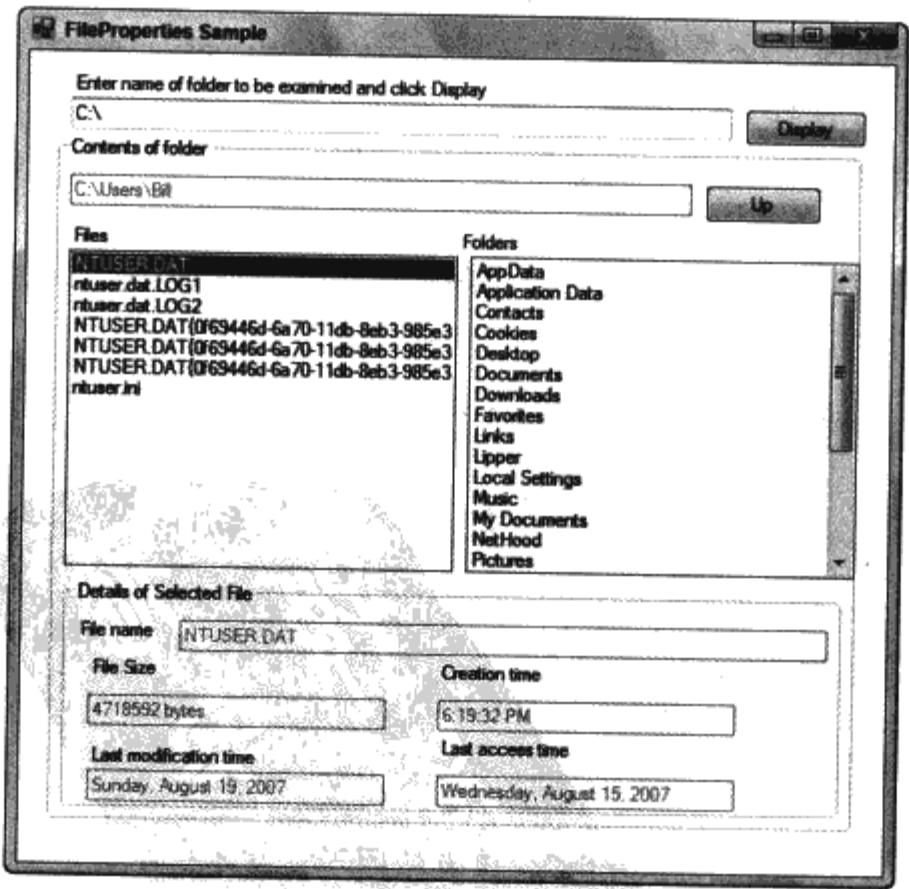


图 25-3

注意，还可以使用 DirectoryInfo 属性显示文件夹的创建时间、最后一次访问时间和最后一

次修改时间。我们只显示已选定文件的这些属性，使本例简单一些。

在 Visual Studio 2008 中创建一个标准的 C# Windows 应用程序项目，从工具箱的 Windows Forms 区域添加各种文本框和列表框。用更直观的名称来重命名这些控件：textBoxInput、textBoxFolder、buttonDisplay、buttonUp、listBoxFiles、listBoxFolders、textBoxFileName、textBoxCreationTime、textBoxLastAccessTime、textBoxLastWriteTime 和 textBoxFileSize。

然后需要指出，将使用 System.IO 命名空间：

```
using System;
using System.Windows.Forms;
using System.IO;
```

本章中所有与文件系统相关的示例都要使用该命名空间，但我们没有显式说明其余示例中的这部分代码。然后在主窗体中添加一个成员字段：

```
partial class Form1 : Form
{
    private string currentFolderPath;
```

currentFolderPath 存储了文件夹的路径，其内容显示在列表框中。

现在需要为用户生成的事件添加事件处理程序。用户可能输入的是：

- 用户单击 Display 按钮：此时，需要确定用户在主文本框中键入的内容是文件的路径还是文件夹的路径。如果是文件夹，列表框中就会列出该文件夹中的文件和子文件夹。如果是文件，仍要对包含该文件的文件夹进行上述操作，还要在下面的文本框中显示文件的属性。
- 用户单击 Files 列表框中的一个文件名：此时，在下面的文本框中显示文件的属性。
- 用户单击 Folders 列表框中的一个文件夹名：此时，将清理所有的控件，并且在列表框中显示这个子文件夹的内容。
- 用户单击 Up 按钮：此时，将清理所有的控件，并且在列表框中显示文件夹的父文件夹中的内容。

在列出事件处理程序的代码前，先列出实际完成所有任务的方法的代码。首先，需要清除所有控件的内容，这个方法很容易理解：

```
protected void ClearAllFields()
{
    listBoxFolders.Items.Clear();
    listBoxFiles.Items.Clear();
    textBoxFolder.Text = "";
    textBoxFileName.Text = "";
    textBoxCreationTime.Text = "";
    textBoxLastAccessTime.Text = "";
    textBoxLastWriteTime.Text = "";
    textBoxFileSize.Text = "";
}
```

其次，定义一个方法 DisplayFileInfo()，该方法用于在文本框中显示给定文件的信息。它带有一个字符串参数，即文件的完整路径名，它根据该路径创建一个 FileInfo 对象：

```
protected void DisplayFileInfo(string fileFullName)
{
```

```

        FileInfo theFile = new FileInfo(fileFullName);
        if (!theFile.Exists)
            throw new FileNotFoundException("File not found: " + fileFullName);
        textBoxFileName.Text = theFile.Name;
        textBoxCreationTime.Text = theFile.CreationTime.ToLongTimeString();
        textBoxLastAccessTime.Text = theFile.LastAccessTime.ToLongDateString();
        textBoxLastWriteTime.Text = theFile.LastWriteTime.ToLongDateString();
        textBoxFileSize.Text = theFile.Length.ToString() + " bytes";
    }

```

注意，如果在指定位置定位文件时有任何问题，我们将采取措施，处理抛出的异常。异常在调用例程(一个事件处理程序)中处理。最后，定义一个方法 `DisplayFolderList()`，在两个列表框中显示给定文件夹的内容。该文件夹的完整路径名作为参数传递给该方法：

```

protected void DisplayFolderList(string folderFullName)
{
    DirectoryInfo theFolder = new DirectoryInfo(folderFullName);
    if (!theFolder.Exists)
        throw new FileNotFoundException("Folder not found: " + folderFullName);
    ClearAllFields();
    textBoxFolder.Text = theFolder.FullName;
    currentFolderPath = theFolder.FullName;

    // list all subfolders in folder
    foreach(DirectoryInfo nextFolder in theFolder.GetDirectories())
        listBoxFolders.Items.Add(nextFolder.Name);

    // list all files in folder
    foreach(FileInfo nextFile in theFolder.GetFiles())
        listBoxFiles.Items.Add(nextFile.Name);
}

```

现在看看事件处理程序。用户单击 `Display` 按钮的事件处理程序是最复杂的，因为它需要处理用户输入的 3 种不同的文本。用户可能输入文件夹的路径名、文件的路径名或什么也不输入：

```

protected void OnDisplayButtonClick(object sender, EventArgs e)
{
    try
    {
        string folderPath = textBoxInput.Text;
        DirectoryInfo theFolder = new DirectoryInfo(folderPath);
        if (theFolder.Exists)
        {
            DisplayFolderList(theFolder.FullName);
            return;
        }
        FileInfo theFile = new FileInfo(folderPath);
        if (theFile.Exists)
        {
            DisplayFolderList(theFile.Directory.FullName);
            int index = listBoxFiles.Items.IndexOf(theFile.Name);
            listBoxFiles.SetSelected(index, true);
            return;
        }
        throw new FileNotFoundException("There is no file or folder with "
            + "this name: " + textBoxInput.Text);
    }
    catch(Exception ex)
    {
    }
}

```



```

        MessageBox.Show(ex.Message);
    }
}

```

在上面的代码中，如果用户提供的文本表示一个文件夹或文件，就应实例化 `DirectoryInfo` 和 `FileInfo`，并检查每个对象的 `Exists` 属性。如果它们都不存在，就抛出一个异常。如果输入了一个文件夹，就调用 `DisplayFolderList`，给列表框填充数据。如果输入了一个文件，就需要给列表框填充数据，给显示文件属性的文本框填充数据。具体处理过程是，首先给列表框填充数据，然后在文件列表框中编程选择合适的文件名，这与用户选择该项目的效果相同——引发选中项目的事件。然后退出当前事件处理程序，调用选中项目的事件处理程序，显示文件属性。

下面的代码是一个事件处理程序，当用户选中或编程选中文件列表框中的一个项目时，就可以由用户或上面编写的代码调用该事件处理程序。它仅构造所选文件的完整路径名，并把该路径传递给前面给出的 `DisplayFileInfo()` 方法：

```

protected void OnListBoxFilesSelected(object sender, EventArgs e)
{
    try
    {
        string selectedString = listBoxFiles.SelectedItem.ToString();
        string fullFileName = Path.Combine(currentFolderPath, selectedString);
        DisplayFileInfo(fullFileName);
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}

```

处理 `Folders` 列表框中的文件夹选择操作的事件处理程序以非常类似的方式实现。但此时调用 `DisplayFolderList()` 来更新列表框的内容：

```

protected void OnListBoxFoldersSelected(object sender, EventArgs e)
{
    try
    {
        string selectedString = listBoxFolders.SelectedItem.ToString();
        string fullPathName = Path.Combine(currentFolderPath, selectedString);
        DisplayFolderList(fullPathName);
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}

```

最后，在单击 `Up` 按钮时，必须调用 `DisplayFolderList()`，但这次需要获得当前显示的文件夹的父文件夹。这可以通过 `FileInfo.DirectoryName` 属性来得到，该属性返回父文件夹的路径：

```

protected void OnUpButtonClick(object sender, EventArgs e)
{
    try
    {
        string folderPath = new FileInfo(currentFolderPath).DirectoryName;
        DisplayFolderList(folderPath);
    }
}

```

```

    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}

```

## 25.2 移动、复制和删除文件

前面已经提到，移动和删除文件或文件夹可以使用 `FileInfo` 和 `DirectoryInfo` 类的 `MoveTo()` 和 `Delete()` 方法来完成。`File` 和 `Directory` 类的这两个对应方法是 `Move()` 和 `Delete()`。`FileInfo` 和 `File` 类也分别执行 `CopyTo()` 和 `Copy()` 方法。没有复制完整文件夹的方法，而应复制文件夹中的每个文件。

这些方法的使用非常直观——SDK 文档提供了详细的解释。本节介绍在特定情况下，调用 `File` 类的静态方法 `Move()`、`Copy()` 和 `Delete()` 的作用。为此，把前面的 `FileProperties` 示例扩展为一个新示例 `FilePropertiesAndMovement`。这个示例有一个额外的功能：无论什么时候显示文件的属性，该应用程序都会给出删除该文件、把该文件移动和复制到其他地方的选项。

### 25.2.1 FilePropertiesAndMovement 示例

图 25-4 所示为新示例应用程序的用户界面。

从这个屏幕图上可以看出，它的外观非常类似于 `FileProperties` 示例，但在窗口的底部添加了一个组，其中包含三个按钮和一个文本框。这些控件仅在示例显示了文件的属性时才能使用，在其他情况下，它们都是禁用的。我们还压缩了现有的控件，防止主窗体过大。在显示文件的属性时，该示例会自动把文件的完整路径名放在底部的文本框中，供用户编辑。用户可以单击底部的任何一个按钮，执行相应的操作。此时，会显示一个相应的信息框，确认该操作，如图 25-5 所示。

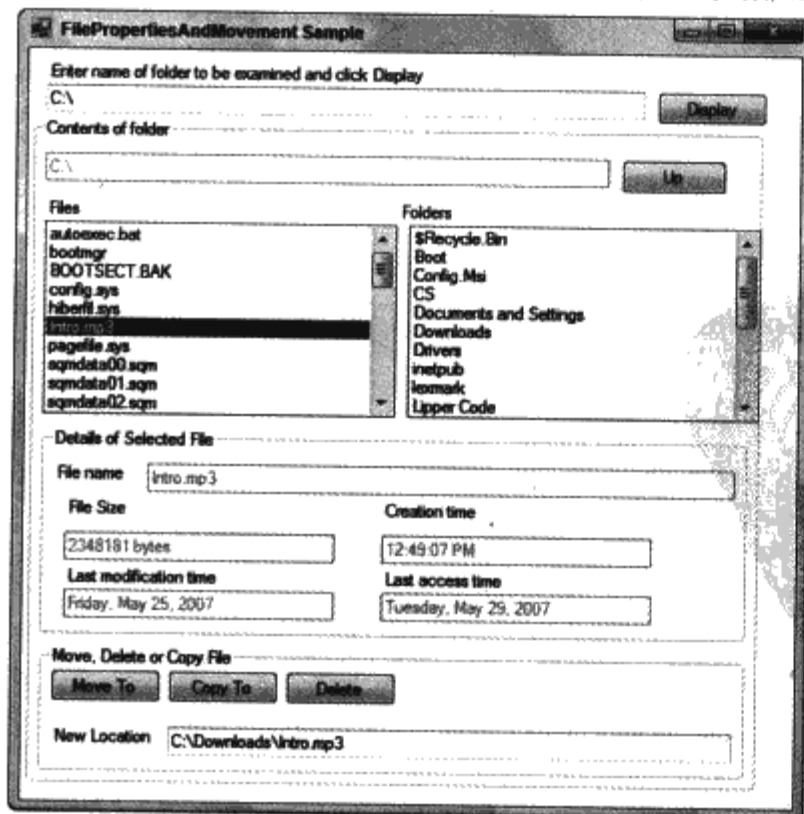


图 25-4

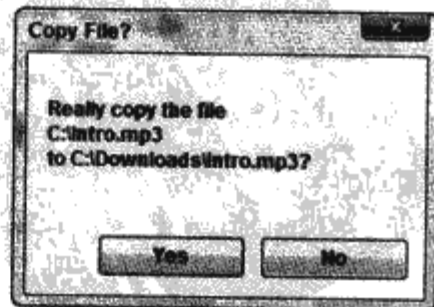


图 25-5



当用户单击了 Yes 按钮后, 就可以开始执行某些动作。用户在窗体上执行的某些动作会使显示不正确。例如, 在移动和删除文件时, 显然不能在同一个地方显示该文件的内容。而且, 如果改变同一个文件夹上的文件名, 显示的信息也会不正确。此时, FilePropertiesAndMovement 示例会重新设置其控件, 在文件的操作结束后, 只显示包含文件的文件夹。

### 25.2.2 示例 FilePropertiesAndMovement 的代码

为此, 需要在 FileProperties 示例中添加相关的控件, 及其事件处理程序代码。我们添加的控件是 buttonDelete、buttonCopyTo、buttonMoveTo 和 textBoxNewPath。

首先看看用户单击 Delete 按钮时调用的事件处理程序:

```
protected void OnDeleteButtonClick(object sender, EventArgs e)
{
    try
    {
        string filePath = Path.Combine(currentFolderPath,
                                         textBoxFileName.Text);
        string query = "Really delete the file\n" + filePath + "?";
        if (MessageBox.Show(query,
                            "Delete File?", MessageBoxButtons.YesNo) == DialogResult.Yes)
        {
            File.Delete(filePath);
            DisplayFolderList(currentFolderPath);
        }
    }
    catch (Exception ex)
    {
        MessageBox.Show("Unable to delete file. The following exception"
                        + " occurred:\n" + ex.Message, "Failed");
    }
}
```

这个方法的代码包含在一个 try 块中, 这是因为很显然会抛出一个异常, 例如在用户单击了 delete 按钮后, 如果不允许删除该文件, 或者当时有另一个进程移动了该文件, 就会抛出一个异常。在 CurrentParentPath 字段中构造要删除文件的路径, 其中包含父文件夹的路径, textBoxFileName 文本框中的文本(其中包含文件名)。

移动和复制文件的方法以类似的方式构造:

```
protected void OnMoveButtonClick(object sender, EventArgs e)
{
    try
    {
        string filePath = Path.Combine(currentFolderPath,
                                         textBoxFileName.Text);
        string query = "Really move the file\n" + filePath + "\nto "
                        + textBoxNewPath.Text + "?";
        if (MessageBox.Show(query,
                            "Move File?", MessageBoxButtons.YesNo) == DialogResult.Yes)
        {
            File.Move(filePath, textBoxNewPath.Text);
            DisplayFolderList(currentFolderPath);
        }
    }
}
```

```

        catch(Exception ex)
        {
            MessageBox.Show("Unable to move file. The following exception"
                + " occurred:\n" + ex.Message, "Failed");
        }
    }

    protected void OnCopyButtonClick(object sender, EventArgs e)
    {
        try
        {
            string filePath = Path.Combine(currentFolderPath,
                textBoxFileName.Text);
            string query = "Really copy the file\n" + filePath + "\nto "
                + textBoxNewPath.Text + "?";
            if (MessageBox.Show(query,
                "Copy File?", MessageBoxButtons.YesNo) == DialogResult.Yes)
            {
                File.Copy(filePath, textBoxNewPath.Text);
                DisplayFolderList(currentFolderPath);
            }
        }
        catch(Exception ex)
        {
            MessageBox.Show("Unable to copy file. The following exception"
                + " occurred:\n" + ex.Message, "Failed");
        }
    }
}

```

还需要确保新按钮和文本框在合适的时间是可用的或禁用的。要使它们在显示文件的内容时可用，需要把下述代码添加到 `DisplayFileInfo()` 中：

```

protected void DisplayFileInfo(string fileFullName)
{
    FileInfo theFile = new FileInfo(fileFullName);
    if (!theFile.Exists)
        throw new FileNotFoundException("File not found: " + fileFullName);

    textBoxFileName.Text = theFile.Name;
    textBoxCreationTime.Text = theFile.CreationTime.ToLongTimeString();
    textBoxLastAccessTime.Text = theFile.LastAccessTime.ToLongDateString();
    textBoxLastWriteTime.Text = theFile.LastWriteTime.ToLongDateString();
    textBoxFileSize.Text = theFile.Length.ToString() + " bytes";

    // enable move, copy, delete buttons
    textBoxNewPath.Text = theFile.FullName;
    textBoxNewPath.Enabled = true;
    buttonCopyTo.Enabled = true;
    buttonDelete.Enabled = true;
    buttonMoveTo.Enabled = true;
}

```

还需要修改 `DisplayFolderList`：

```

protected void DisplayFolderList(string folderFullName)
{
    DirectoryInfo theFolder = new DirectoryInfo(folderFullName);
    if (!theFolder.Exists)
        throw new DirectoryNotFoundException("Folder not found: " + folderFullName);
}

```

```

ClearAllFields();
DisableMoveFeatures();
textBoxFolder.Text = theFolder.FullName;
currentFolderPath = theFolder.FullName;

// list all subfolders in folder
foreach(DirectoryInfo nextFolder in theFolder.GetDirectories())
    listBoxFolders.Items.Add(nextFolder.Name);

// list all files in folder
foreach(FileInfo nextFile in theFolder.GetFiles())
    listBoxFiles.Items.Add(nextFile.Name);
}

```

DisableMoveFeatures 是禁用新控件的一个小工具函数：

```

void DisableMoveFeatures()
{
    textBoxNewPath.Text = "";
    textBoxNewPath.Enabled = false;
    buttonCopyTo.Enabled = false;
    buttonDelete.Enabled = false;
    buttonMoveTo.Enabled = false;
}

```

还需要给 ClearAllFields() 添加额外的代码，以清除额外的文本框：

```

protected void ClearAllFields()
{
    listBoxFolders.Items.Clear();
    listBoxFiles.Items.Clear();
    textBoxFolder.Text = "";
    textBoxFileName.Text = "";
    textBoxCreationTime.Text = "";
    textBoxLastAccessTime.Text = "";
    textBoxLastWriteTime.Text = "";
    textBoxFileSize.Text = "";
    textBoxNewPath.Text = "";
}

```

这样，代码就完整了。

## 25.3 读写文件

读写文件在原则上是非常简单的，但不是通过 DirectoryInfo 或 FileInfo 对象完成的。在 .NET Framework 3.5 中，可以通过 File 对象读写文件。本章的后面将学习如何使用许多其他的类来读写文件，这些类表示一个通用的概念：流。

在 .NET Framework 2.0 推出之前，读写文件比较费劲，可以使用 Framework 中的类来读写文件，但不是很简单。 .NET Framework 2.0 扩展了 File 类，只需编写一行代码，就可以读写文件。 .NET Framework 3.5 也有这个功能。

### 25.3.1 读取文件

在下面读取文件的示例中，创建一个 Windows 窗体应用程序，它包含一个常规的文本框、一个按钮和一个多行文本框。最后，窗体如图 25-6 所示。

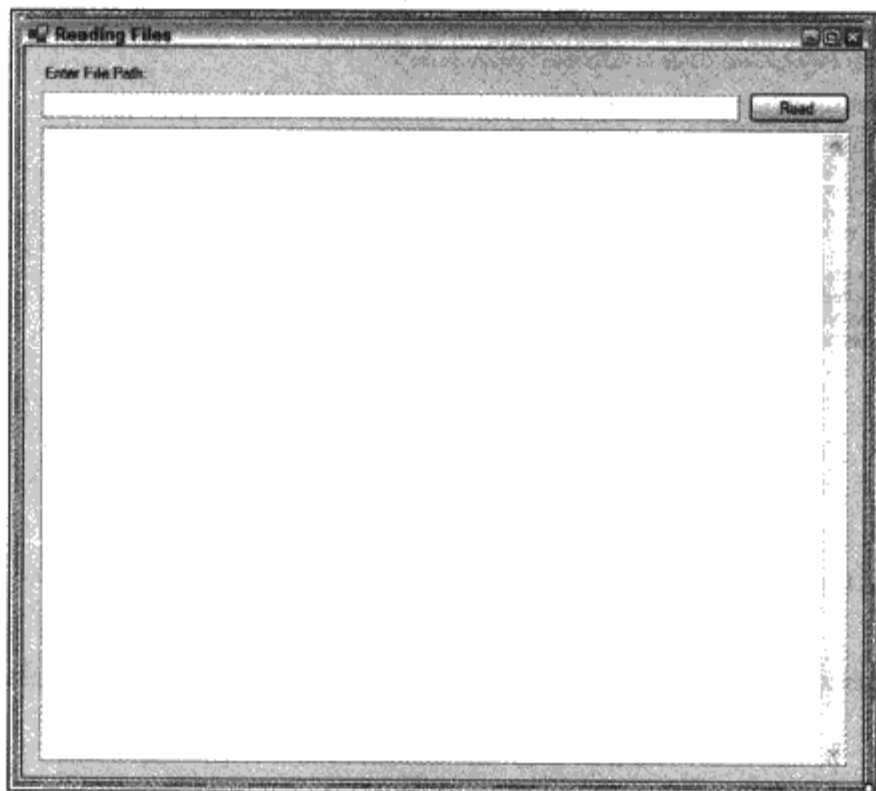


图 25-6

这个窗体的作用是，终端用户在第一个文本框中输入某个文件的路径，单击 Read 按钮。此时应用程序就应读取指定的文件，在多行文本框中显示文件的内容。其代码如下：

```
using System;
using System.Windows.Forms;
using System.IO;

namespace ReadingFiles
{
    partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            textBox2.Text = File.ReadAllText(textBox1.Text);
        }
    }
}
```

在建立这个示例时，第一步是添加 using 语句，引入 System.IO 命名空间。之后，使用窗体上 Send 按钮的 button1\_Click 事件，用文件中的内容填充文本框。现在，就可以使用 File.ReadAllText 方法获得文件的内容。在 .NET Framework 2.0 中，可以使用一个语句读取文件。ReadAll 方法会打开指定的文件，读取内容，然后关闭文件。ReadAll 方法的返回值是包含文件全部内容的



字符串。最终结果如图 25-7 所示。

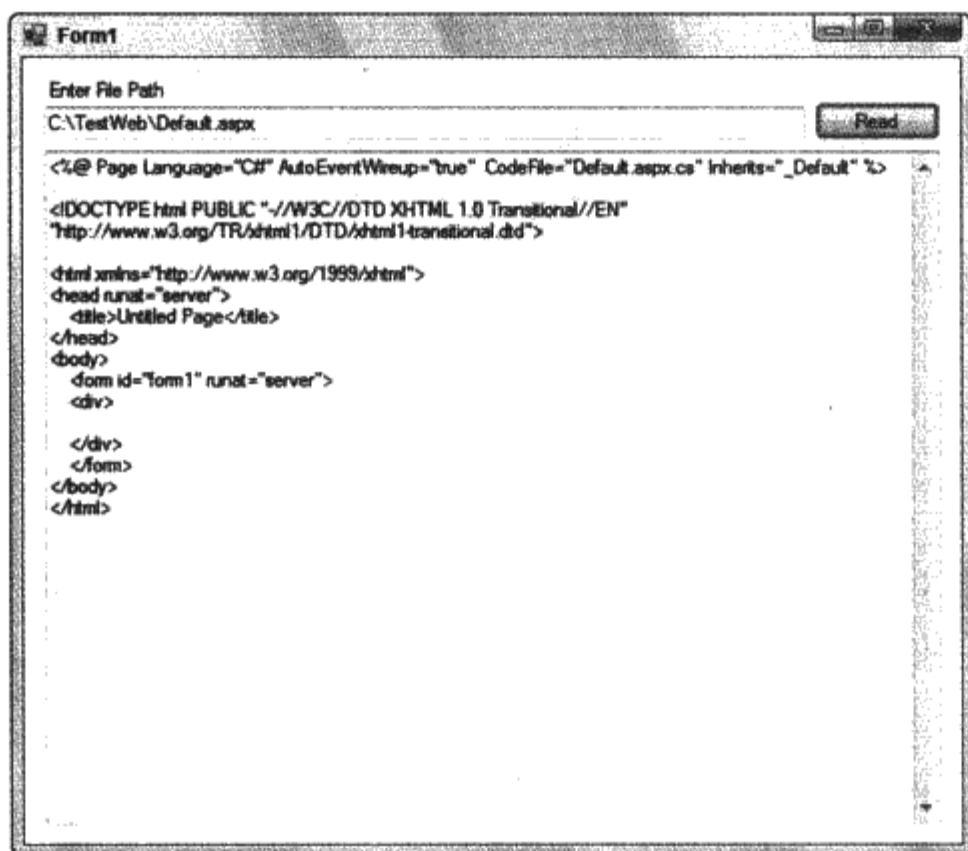


图 25-7

上面示例中的 `File.ReadAllText()` 签名有如下结构：

```
File.ReadAllText (FilePath);
```

另一个选项指定了要读取的文件的编码格式：

```
File.ReadAllText (FilePath, Encoding);
```

使用这个签名可以指定在打开和读取文件内容时使用的编码格式。因此，可以编写如下代码：

```
File.ReadAllText (textBox1.Text, Encoding.ASCII);
```

打开并处理文件的其他选项有 `ReadAllBytes()` 和 `ReadAllLines()` 方法。`ReadAllBytes()` 方法可以打开二进制文件，将其内容读入一个字节数组。前面列出的 `ReadAllText()` 方法会在一个字符串数组实例中提供指定文件的全部内容。我们对此不感兴趣，而关注以逐行方式处理文件中内容的方式。此时，应使用 `ReadAllLines()` 方法，因为它拥有这个功能。

### 25.3.2 写入文件

在 .NET Framework 中，读取文件是一个非常简单的过程，写入文件也一样简单。基类库除了提供 `ReadAll`、`ReadAllLines` 和 `ReadAllBytes` 方法，以几种不同的方式读取文件之外，还提供了写入文件的方法 `WriteAll`、`WriteAllBytes` 和 `WriteAllLines`。

为了说明如何写入文件，使用同一个 Windows 窗体应用程序，但把窗体上的多行文本框用于将数据输入文件。`button1_Click` 事件处理程序的代码如下所示：

```
private void button1_Click(object sender, EventArgs e)
```



```
{
    File.WriteAll(textBox1.Text, textBox2.Text);
}
```

建立窗体，并启动，在第一个文本框中输入 C:\Testing.txt，在第二个文本框中输入一些内容，然后单击按钮。什么也没有发生，但如果查看 C 盘，就会看到包含指定内容的 Testing.txt 文件。

如果在保存和关闭文件之前，给文件指定了一些内容，WriteAll 方法会进入指定的位置，创建一个新的文本文件。只需一行代码即可完成文件的写入操作。

如果再次运行应用程序，指定同一个文件(Testing.txt)，但输入一些新内容，再次单击按钮，应用程序会执行相同的任务，但注意，这次新内容不是添加到原有内容的后面，而是完全覆盖以前的内容。事实上，WriteAll、WriteAllBytes 和 WriteAllLines 方法都会覆盖以前的文件，所以在使用这些方法时要小心。

上面示例中的 WriteAll 方法使用如下签名：

```
File.WriteAll(filePath, Content)
```

还可以指定新文件的编码格式：

```
File.WriteAll(filePath, Content, Encoding)
```

WriteAllBytes 方法可以使用字节数组把内容写入文件，WriteAllLines 方法可以把字符串数组写入文件，如下面的事件处理程序所示：

```
private void button1_Click(object sender, EventArgs e)
{
    string[] movies =
        {"Grease",
         "Close Encounters of the Third Kind",
         "The Day After Tomorrow"};

    File.WriteAllLines("C:\Testing.txt", movies);
}
```

现在单击按钮，应用程序就会提供 Testing.txt 文件，其内容如下：

```
Grease
Close Encounters of the Third Kind
The Day After Tomorrow
```

WriteAllLines 方法把字符串数组中的每个元素单独放在一行上。

数据不仅可以写入磁盘，还可以放在其他地方(例如指定的管道或内存)，所以必须理解如何在 .NET 中使用流处理文件输入输出，作为一种移动文件内容的方式。

### 25.3.3 流

流的概念已经存在很长时间了。流是一个用于传输数据的对象，数据的传输有两个方向：

- 如果数据从外部源传输到程序中，这就是读取流。
- 如果数据从程序传输到外部源，这就是写入流。

外部源常常是一个文件，但也不完全都是文件。它还可能是：

- 使用一些网络协议读写网络上的数据，其目的是选择数据，或从另一个计算机上发送数据。
- 读写到指定的管道上。
- 把数据读写到一个内存区域上。

在这些示例中，Microsoft 提供了一个.NET 基类 `System.IO.MemoryStream` 来读写内存，而 `System.Net.Sockets.NetworkStream` 处理网络数据。读写管道没有基本的流类，但有一个一般的流类 `System.IO.Stream`，如果要编写一个这样的类，可以从这个基类继承。流对外部数据源不做任何假定。

外部源甚至可以是代码中的一个变量。这听起来很荒谬，但使用流在变量之间传输数据的技术是一个非常有用的技巧，可以在数据类型之间转换数据。C 语言使用它在整型和字符串之间转换数据类型，或者使用函数 `sprintf()` 格式化字符串。

使用一个独立的对象来传输数据，比使用 `FileInfo` 或 `DirectoryInfo` 类更好，因为把传输数据的概念与特定数据源分离开来，可以更容易切换数据源。流对象本身包含许多代码，可以在外部数据源和代码中的变量之间移动数据，把这些代码与特定数据源的概念分离开来，就更容易实现不同环境下代码的重用(通过继承)。例如，前面提到的 `StringReader` 和 `StringWriter` 类，与本章后面用于读写文本文件的两个类 `StreamReader` 和 `StreamWriter` 一样，都是同一继承树上的一部分，这些类在后台共享许多代码。

在 `System.IO` 命名空间中，与流相关的类层次结构如图 25-8 所示。

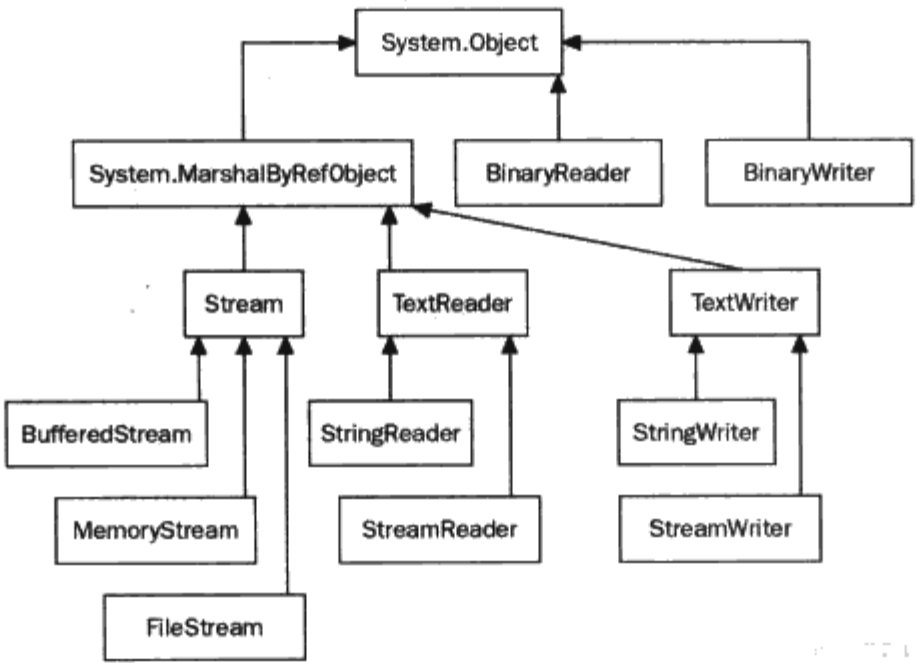


图 25-8

对于文件的读写，最常用的类如下：

- `FileStream`(文件流)：这个类主要用于在二进制文件中读写二进制数据——也可以使用它读写任何文件。
- `StreamReader`(流读取器)和 `StreamWriter`(流写入器)：这两个类是专门用于读写文本文件的。

虽然我们没有在示例中使用另外两个类 `BinaryReader` 和 `BinaryWriter`，但它们也是很有用的。`BinaryReader` 和 `BinaryWriter` 这两个类本身并不执行流，而是提供其他流对象的包装器。`BinaryReader` 和 `BinaryWriter` 还可以对二进制数据进行额外的格式化，直接从相关的流中读写

C#变量的内容。最简单的方式是把 `BinaryReader` 和 `BinaryWriter` 放在流和代码之间，进行额外的格式化，如图 25-9 所示。

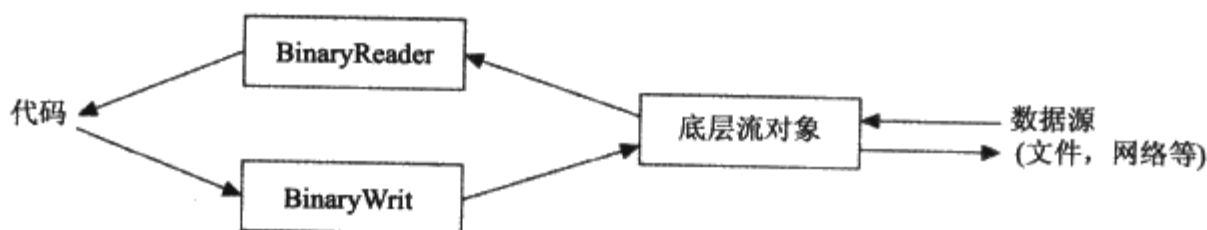


图 25-9

使用这些类和直接使用底层的流对象之间的区别是基本流是按照字节来工作的。例如，在保存某个文档时，需要把类型为 `long` 的变量内容写入到一个二进制文件中，每个 `long` 都占用 8 个字节，如果使用一般的二进制流，就必须显式地写入内存的 8 个字节。在 C# 代码中，必须显式执行一些按位操作，从 `long` 值中提取这 8 个字节。使用 `BinaryWriter` 实例，可以把整个操作封装在 `BinaryWriter.Write()` 方法的一个重载方法中，该方法的参数是 `long` 型，它把 8 个字节写入流中(如果流指向一个文件，就写入该文件)。对应的 `BinaryReader.Read()` 方法则从流中提取 8 个字节，恢复 `long` 的值。`BinaryReader` 和 `BinaryWriter` 类的更多信息详见 SDK 文档。

#### 25.3.4 缓存的流

从性能上看，在读写文件时，输出结果会被缓存。如果程序要求读取文件流中下面的 2 个字节，该流会把请求传送给 Windows，Windows 不会连接文件系统，再定位文件，并从磁盘中读取文件，仅读取 2 个字节，而是在一次读取过程中，获取文件中的一个大块，把该块保存在一个内存区域即缓冲区上。以后对流中数据的请求就会从该缓冲区中读取，直到读取完该缓冲区为止。此时，Windows 会从文件中再获取另一个数据块。写入文件的方式与此相同。对于文件，操作系统会自动完成读写操作，但需要编写一个流类，从其他没有保存到缓冲区的设备中读取数据。如果是这样，就应从 `BufferedStream` 派生一个类，以执行缓冲操作(但 `BufferedStream` 不执行缓冲操作，它是专门为程序频繁切换读数据和写数据而设计的)。

#### 25.3.5 使用 `FileStream` 类读写二进制文件

读写二进制文件通常要使用 `FileStream` 类。如果使用 .NET Framework 1.x，就总是使用这个类。

##### 1. `FileStream` 类

`FileStream` 实例用于读写文件中的数据。要构造 `FileStream` 实例，需要以下 4 条信息：

- 要访问的文件。
- 表示如何打开文件的模式。例如，创建一个新文件或打开一个现有的文件。如果打开一个现有的文件，写入操作是覆盖文件原来的内容，还是添加到文件的末尾？
- 表示访问文件的方式——是只读、只写，还是读写？
- 共享访问——表示是否独占访问文件。如果允许其他流同时访问文件，则这些流是只读、只写，还是读写文件？

第一条信息通常用一个包含文件完整路径名的字符串来表示，本章只考虑需要该字符串的构造函数。除了这些构造函数外，其他的构造函数用老式的 Windows-API 文件句柄来处理文件。其余 3 条信息分别由 3 个.NET 枚举 FileMode、FileAccess 和 FileShare 来表示，这些枚举的值很容易理解，如表 25-4 所示。

表 25-4

枚 举	值
FileMode	Append、Create、CreateNew、Open、OpenOrCreate 和 Truncate
FileAccess	Read、ReadWrite 和 Write
FileShare	Inheritable、None、Read、ReadWrite 和 Write

注意，对于 FileMode，如果要求的模式与文件的现有状态不一致，就会抛出一个异常。如果文件不存在，Append、Open 和 Truncate 会抛出一个异常，如果文件存在，CreateNew 会抛出一个异常。Create 和 OpenOrCreate 可以处理这两种情况，但 Create 会删除现有的文件，创建一个新的空文件。FileAccess 和 FileShare 枚举是按位标志，所以这些值可以与 C#的按位 OR 运算符|合并使用。

FileStream 有许多构造函数，其中 3 个最简单的构造函数如下所示。

```
// creates file with read-write access and allows other streams read access
FileStream fs = new FileStream(@"C:\C# Projects\Project.doc",
    FileMode.Create);
// as above, but we only get write access to the file
FileStream fs2 = new FileStream(@"C:\C# Projects\Project2.doc",
    FileMode.Create, FileAccess.Write);
// as above but other streams don't get any access to the file while
// fs3 is open
FileStream fs3 = new FileStream(@"C:\C# Projects\Project3.doc",
    FileMode.Create, FileAccess.Write, FileShare.None);
```

从这段代码中可以看出，构造函数的这些重载方法会把 FileAccess.ReadWrite 和 FileShare.Read 的默认值作为第 3、4 个参数，也可以以多种方式从 FileInfo 实例中创建一个文件流：

```
FileInfo myFile4 = new FileInfo(@"C:\C# Projects\Project4.doc");
FileStream fs4 = myFile4.OpenRead();
FileInfo myFile5= new FileInfo(@"C:\C# Projects\Project5doc");
FileStream fs5 = myFile5.OpenWrite();
FileInfo myFile6= new FileInfo(@"C:\C# Projects\Project6doc");
FileStream fs6 = myFile6.Open(FileMode.Append, FileAccess.Write,
    FileShare.None);
FileInfo myFile7 = new FileInfo(@"C:\C# Projects\Project7.doc");
FileStream fs7 = myFile7.Create();
```

FileInfo.OpenRead()提供的流只能读取现有的文件，而 FileInfo ().OpenWrite()可以进行读写访问。FileInfo.Open()允许显式指定模式、访问方式和文件共享参数。

使用完一个流后，就应关闭它：

```
fs.Close();
```

关闭流会释放与它相关的资源，允许其他应用程序为同一个文件设置流。这个操作也会刷新缓存。在打开和关闭流之间，可以读写其中的数据。`FileStream` 有许多方法可以进行这样的读写。

`ReadByte()` 是读取数据的最简单的方式，它从流中读取一个字节，把结果转换为一个 0~255 之间的一个整数。如果到达该流的末尾，就返回 -1：

```
int NextByte = fs.ReadByte();
```

如果要一次读取多个字节，可以调用 `Read()` 方法，它可以把特定数量的字节读入到一个数组中。`Read()` 方法返回实际读取的字节数——如果这个值是 0，就表示到达了流的尾端。下面是读入 `Byte` 数组 `ByteArray` 的一个示例：

```
int nBytesRead = fs.Read(ByteArray, 0, nBytes);
```

`Read()` 的第二个参数是一个偏移值，使用它可以要求 `Read` 读取的数据从数组的某个元素开始填充，而不是从第一个元素开始。第三个参数是要读入数组的字节数。

如果要给文件写入数据，可以使用两个并行方法 `WriteByte()` 和 `Write()`。`WriteByte()` 方法把一个字节写入流：

```
byte NextByte = 100;
fs.WriteByte(NextByte);
```

另外，`Write()` 则写入一个字节数组。例如，如果用一些值初始化前面的 `ByteArray`，就可以使用下面的代码写入数组的前 `nBytes` 个字节：

```
fs.Write(ByteArray, 0, nBytes);
```

与 `Read()` 一样，第二个参数可以从数组的某个元素开始写入，而不是从第一个元素开始。`WriteByte()` 和 `Write()` 都没有返回值。

除了这些方法以外，`FileStream` 还有其他方法和属性可以处理簿记任务，例如确定流中有多少字节，锁定流或刷新缓冲区等。其他方法通常不是基本读写数据所必需的，如果需要使用它们，可以参阅 SDK 文档说明书。

## 2. BinaryFileReader 示例

下面编写一个示例 `BinaryFileReader` 说明 `FileStream` 类的用法。这个示例可以读取和显示任何文件。在 Visual Studio 2008 中创建一个 Windows 应用程序，添加一个菜单项，它可以打开一个标准的 `OpenFileDialog`，要求用户指定要读取的文件，然后把该文件显示为二进制码。在读取二进制文件时，需要显示非打印(non-printable)字符。此时可以在多行文本框中逐个显示文件中的每个字节，每行显示 16 个字节。如果字节表示一个可显示的 ASCII 字符，就显示该字符，否则就以十六进制的格式显示该字节的值。在这两种情况下，显示的文本之间都用空格隔开，这样每个显示的字节都占用 4 列，显得它们的排列非常整齐。

图 25-10 是 `BinaryFileReader` 应用程序查看文本文件时所显示的情况(因为 `BinaryFileReader` 可以查看任何文件，所以可以在文本文件和二进制文件中使用它)。本示例读取一个基本的 ASP.NET 页面(.aspx)。



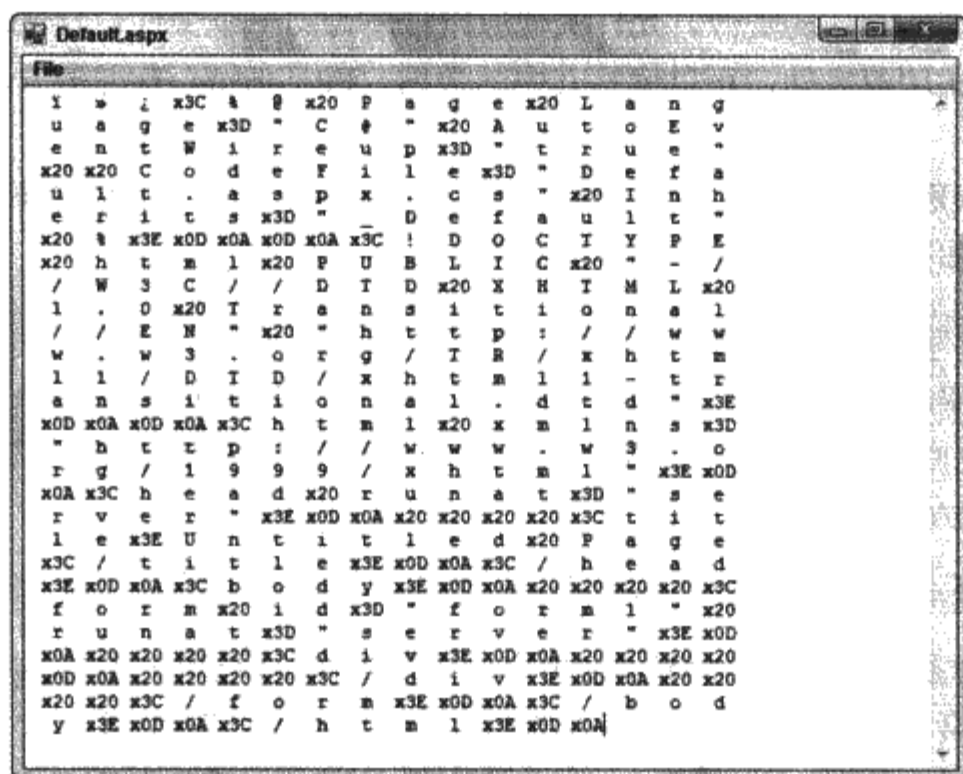


图 25-10

显然，这个格式更适合于查看单个字节的值，而不适合显示文本。本章的后面会开发一个专门用于读取文本文件的示例——然后就可以查看文件中的内容了。这个示例的优点是可以查看任何文件的内容。

这个示例没有说明如何写入文件。这是因为我们不希望把文本框中的内容转换为二进制流，增加程序的复杂性。在后面开发可读写文本文件的示例中，将介绍如何写入文件。

下面看看用于获得这些结果的代码。首先，需要一个额外的 `using` 语句，引入 `System.IO` 命名空间：

```
using System.IO;
```

接着，给主窗体类添加两个字段，一个字段表示文件对话框，另一个是表示当前文件的路径的字符串：

```
Partial class Form1:Form
{
```

```
private OpenFileDialog chooseOpenFileDialog = new OpenFileDialog();
private string chosenFile;
```

还需要添加一些标准的 Windows 窗体代码，来处理菜单和文件对话框：

```
public Form1()
{
    InitializeComponent();
    menuFileOpen.Click += OnFileOpen;
    chooseOpenFileDialog.FileOk += OnOpenFileDialogOK;
}

void OnFileOpen(object Sender, EventArgs e)
{
    chooseOpenFileDialog.ShowDialog();
}

void OnOpenFileDialogOK(object Sender, CancelEventArgs e)
{
}
```

```

        chosenFile = chooseOpenFileDialog.FileName;
        this.Text = Path.GetFileName(chosenFile);
        DisplayFile();
    }

```

可以看出，用户单击 OK，在文件对话框中选择一个文件后，就会调用方法 `DisplayFile()`，读取所选中的文件：

```

void DisplayFile()
{
    int nCols = 16;
    FileStream inStream = new FileStream(chosenFile, FileMode.Open,
                                         FileAccess.Read);
    long nBytesToRead = inStream.Length;
    if (nBytesToRead > 65536/4)
        nBytesToRead = 65536/4;

    int nLines = (int)(nBytesToRead/nCols) + 1;
    string [] lines = new string[nLines];
    int nBytesRead = 0;

    for (int i=0 ; i<nLines ; i++)
    {
        StringBuilder nextLine = new StringBuilder();
        nextLine.Capacity = 4*nCols;

        for (int j = 0 ; j<nCols ; j++)
        {
            int nextByte = inStream.ReadByte();
            nBytesRead++;
            if (nextByte < 0 || nBytesRead > 65536)
                break;
            char nextChar = (char)nextByte;
            if (nextChar < 16)
                nextLine.Append(" x0" + string.Format("{0,1:X}",
                                                         (int)nextChar));
            else if
                (char.IsLetterOrDigit(nextChar) ||
                 char.IsPunctuation(nextChar))
                nextLine.Append(" " + nextChar + " ");
            else
                nextLine.Append(" x" + string.Format("{0,2:X}",
                                                         (int)nextChar));
        }
        lines[i] = nextLine.ToString();
    }
    inStream.Close();
    this.textBoxContents.Lines = lines;
}

```

在这个方法中进行了许多工作，首先为所选文件实例化一个 `FileStream`，指定要打开一个现有文件进行读取。然后确定要读取多少个字节，和显示多少行。这个字节数一般是文件中的字节数。文本框最多只能显示 65 536 个字符，但以我们选择的格式，文件中的每个字节可以显示 4 个字符，因此如果文件的字节数大于  $65\,536/4 = 16\,384$ ，就需要覆盖一些已显示的字节。

#### 注意：

如果要在这种环境下显示较长的文件，就需要使用 `System.Windows.Forms` 命名空间中的

RichTextBox 类。RichTextBox 类似于文本框，但它有更高级的格式化功能，在显示的文本大小方面没有限制，此处使用 TextBox，是为了让示例比较简单，用户可以只考虑读取文件的过程。

在该方法中，有相当多的部分是两个嵌套的 for 循环，它们构造出每个要显示的文本行。我们使用 StringBuilder 类来构造每一文本行，其性能方面的原因是：可以把每个字节的合适文本追加到表示每行文本的字符串上。如果一行上有一个新字符串，并复制构造该行时的一半字符，则不仅要花很多时间分配字符串，还会浪费堆上的许多内存。注意，可显示的字符是指字母、数字或标点符号，这与相关的静态方法 System.Char 指定的一样。值小于 16 的字符都不是可显示的字符，因此回车符(13)和换行符(10)都是二进制字符(如果这些字符单独占一行，多行文本框不能正确显示它们)。

另外，使用属性窗口，把文本框的字体改为固定宽度的字体——我们选择 Courier New 9pt 常规字体，并把文本框设置为有水平和垂直滚动条。

最后，关闭流，把文本框的内容设置为已建立的字符串数组。

### 25.3.6 读写文本文件

理论上，可以使用 FileStream 类读取和显示文本文件。前面已经介绍了这个类。上面显示 NewFile.txt 文件的格式不太容易理解，但这并不是 FileStream 类的问题——而在于我们在文本框中显示结果所使用的方式。

如果知道某个文件包含文本，通常就可以使用 StreamReader 和 StreamWriter 类更方便地读写它们，而不是 FileStream 类。这是因为这些类工作的级别比较高，特别适合于读写文本。它们执行的方法可以根据流的内容，自动检测出停止读取文本较方便的位置，特别是：

- 这些类执行的方法(StreamReader.ReadLine()和 StreamWriter.WriteLine())可以一次读写一行文本。在读取文件时，流会自动确定下一个回车符的位置，并在该处停止读取。在写入文件时，流会自动把回车符和换行符添加到文本的末尾。
- 使用 StreamReader 和 StreamWriter 类，就不需要担心文件中使用的编码方式(文本格式)了。可能的编码方式是 ASCII(一个字节表示一个字符)或者基于 Unicode 的格式，UNICODE、UTF7、UTF8 和 UTF32。Windows 9x 系统上的文本文件总是 ASCII 格式，因为 Windows 9x 系统不支持 Unicode，但 Windows NT、2000、XP、2003、Vista 和 Windows Server 2008 都支持 Unicode，所以文本文件除了包含 ASCII 数据之外，理论上可以包含 Unicode、UTF7、UTF8 或 UTF32 数据。其约定是：如果文件是 ASCII 格式，就只包含文本。如果是 Unicode 格式，就用文件的前两个或三个字节来表示，这几个字节可以设置为表示文件中格式的值的特定组合。

这些字节称为字节码标记。在使用标准 Windows 应用程序打开一个文件时，例如 Notepad 或 WordPad，不需要考虑这个问题，因为这些应用程序都支持不同的编码方法，会自动正确地读取文件。StreamReader 类也是这样，它可以正确读取任何格式的文件，而 StreamWriter 类可以使用任何一种编码技术格式化它要写入的文本。另一方面，如果要使用 FileStream 类读取和显示文本文件，就必须自己处理这个过程。

## 1. StreamReader 类

StreamReader 用于读取文本文件。用某些方式构造 StreamReader 要比构造 FileStream 实例更简单, 因为使用 StreamReader 时不需要 FileStream 的一些选项。特别是不需要模式和访问类型, 因为 StreamReader 只能执行读取操作。除此以外, 没有指定共享许可的选项, 但 StreamReader 有两个新选项:

- 需要指定不同的编码方法所执行的不同操作。可以构造一个 StreamReader 检查文件开头的字节码标记, 确定编码方法, 或者告诉 StreamReader 该文件使用某个编码方法。
- 不提供要读取的文件名, 而为另一个流提供引用。

最后一个选项需要解释一下, 因为它涉及到把读写数据的模型建立在流概念上的另一个优点。StreamReader 工作在相对较高的级别上, 如果有另一个流在读取其他源的数据, 就要使用由 StreamReader 提供的工具来处理这个流, 就好像这个流包含文本, 此时 StreamReader 就非常有用。可以把这个流的输出传送到 StreamReader 上, 这样, StreamReader 就可以读取和处理任何数据源(不仅仅是文件)中的数据了。前面在讨论 BinaryReader 类时也讨论了这种情况。但在本书中, 只使用 StreamReader 来直接连接文件。

因此, StreamReader 有非常多的构造函数。而且, 还有两个返回 StreamReader 引用的 FileInfo 方法: OpenText() 和 CreateText()。下面仅说明其中一些构造函数。

最简单的构造函数只带一个文件名参数。StreamReader 会检查字节码标记, 确定编码方法:

```
StreamReader sr = new StreamReader(@"C:\My Documents\ReadMe.txt");
```

另外, 如果指定 UTF8 编码方法:

```
StreamReader sr = new StreamReader(@"C:\My Documents\ReadMe.txt",  
                                     Encoding.UTF8);
```

使用类 System.Text.Encoding 上的几个属性之一, 就可以指定编码方法。这个类是一个抽象基类, 可以根据这个类定义许多类, 其方法可执行实际的文本编码。每个属性都返回相应类的一个实例, 可以使用的属性包括:

- ASCII
- Unicode
- UTF7
- UTF8
- UTF32
- BigEndianUnicode

下面的示例解释了如何把 StreamReader 关联到 FileStream 上。其优点是可以显式指定是否创建文件和共享许可, 如果直接把 StreamReader 关联到文件上, 就不能这么做:

```
FileStream fs = new FileStream(@"C:\My Documents\ReadMe.txt",  
                               FileMode.Open, FileAccess.Read, FileShare.None);  
StreamReader sr = new StreamReader(fs);
```

对于本例, 指定 StreamReader 查找字节码标记, 以确定使用了什么编码方法, 以后的示例



也是这样，从一个 `FileInfo` 实例中获得 `StreamReader`：

```
FileInfo myFile = new FileInfo(@"C:\My Documents\ReadMe.txt");
StreamReader sr = myFile.OpenText();
```

与 `FileStream` 一样，应在使用后关闭 `StreamReader`。如果没有这样做，就会致使文件一直锁定，因此不能执行其他进程(除非使用 `FileStream` 构造 `StreamReader`，指定 `FileShare.ShareReadWrite`)：

```
sr.Close();
```

介绍完实例化 `StreamReader` 后，就可以用该实例作一些工作了。与 `FileStream` 一样，我们仅指出可以用于读取数据的许多方式，在 SDK 文档说明书中可以到查阅其他不太常用的 `StreamReader` 方法。

最简单的方法是 `ReadLine()`，该方法一次读取一行，但返回的字符串中不包括标记该行结束的回车换行符：

```
string nextLine = sr.ReadLine();
```

另外，还可以在一个字符串中提取文件的所有剩余内容(严格地说，是流的全部剩余内容)：

```
string restOfStream = sr.ReadToEnd();
```

可以只读取一个字符：

```
int nextChar = sr.Read();
```

`Read()`的重载方法可以把返回的字符转换为一个整数，如果到达流的尾端，就返回-1。最后，可以用一个偏移值，把给定个数的字符读到数组中：

```
// to read 100 characters in.
int nChars = 100;
char [] charArray = new char[nChars];
int nCharsRead = sr.Read(charArray, 0, nChars);
```

如果要求读取的字符数多于文件中剩余的字符数，`nCharsRead` 应小于 `nChars`。

## 2. StreamWriter 类

`StreamWriter` 类的工作方式与 `StreamReader` 类似，但 `StreamWriter` 只能用于写入文件(或另一个流)。构造 `StreamWriter` 的方法包括：

```
StreamWriter sw = new StreamWriter(@"C:\My Documents\ReadMe.txt");
```

上面的代码使用了 UTF8 编码方法，.NET 把这种编码方法设置为默认的编码方法。还可以指定其他的编码方法：

```
StreamWriter sw = new StreamWriter(@"C:\My Documents\ReadMe.txt", true,
    Encoding.ASCII);
```

在这个构造函数中，第二个参数是 `Boolean` 型，表示文件是否应以追加方式打开。构造函数的参数不能仅是一个文件名和一个编码类。



当然，可以把 `StreamWriter` 关联到一个文件流上，以获得打开文件的更多控制选项：

```
FileStream fs = new FileStream(@"C:\My Documents\ReadMe.txt",
    FileMode.CreateNew, FileAccess.Write, FileShare.Read);
StreamWriter sw = new StreamWriter(fs);
```

`FileInfo` 不执行返回 `StreamWriter` 类的任何方法。

另外，如果要创建一个新文件，并开始给它写入数据，可以使用下面的代码：

```
FileInfo myFile = new FileInfo(@"C:\My Documents\NewFile.txt");
StreamWriter sw = myFile.CreateText();
```

与其他流类一样，在使用完后，要关闭 `StreamWriter`：

```
sw.Close();
```

写入流可以使用 `StreamWriter.Write()` 的 4 个重载方法来完成。最简单的方式是写入一个流，后面加上一个回车换行符：

```
string nextLine = "Groovy Line";
sw.WriteLine();
```

也可以写入一个字符：

```
char nextChar = 'a';
sw.Write(nextChar);
```

也可以写入一个字符数组：

```
char [] charArray = new char[100];
// initialize these characters
sw.Write(charArray);
```

甚至可以写入字符数组的一部分：

```
int nCharsToWrite = 50;
int startAtLocation = 25;
char [] charArray = new char[100];
// initialize these characters
sw.Write(charArray, startAtLocation, nCharsToWrite);
```

### 3. ReadWriteText 示例

`ReadWriteText` 示例说明了 `StreamReader` 和 `StreamWriter` 类的用法。它非常类似于前面的 `ReadBinaryFile` 示例，但假定要读取的文件是一个文本文件，并显示其内容。它还可以保存文件(包括在文本框中对文本进行的修改)。它将以 Unicode 格式保存文件。

图 25-11 所示的 `ReadWriteText` 用于显示前面的 `NewFile.aspx` 文件。但这次读取内容会更容易一些。

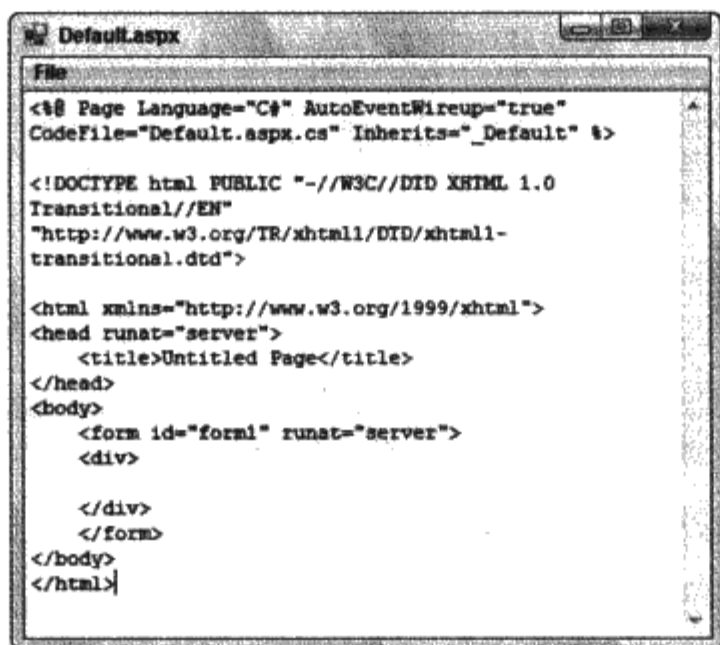


图 25-11

这里不打算介绍给打开文件对话框添加事件处理程序的详细内容，因为它们基本上与前面的 `BinaryFileReader` 示例相同。与这个示例相同，打开一个新文件，将调用 `DisplayFile()` 方法。其唯一的区别是 `DisplayFile` 的执行方式，本例有一个保存文件的选项。这由另一个菜单项 `Save` 来表示，这个选项的处理程序调用我们添加到代码中的另一个方法 `SaveFile()` (注意，这个新文件总是覆盖原来的文件——这个示例没有写入另一个文件的选项)。

首先看看 `SaveFile()`，因为它是最简单的一个函数。首先利用 `StreamReader.WriteLine()` 方法把文本框中的每行文本依次写入 `StreamWriter` 流，并在每行文本的最后加上回车换行符：

```
void SaveFile()
{
    StreamWriter sw = new StreamWriter(chosenFile, false, Encoding.Unicode);

    foreach (string line in textBoxContents.Lines)
        sw.WriteLine(line);
    sw.Close();
}
```

`chosenFile` 是主窗体的一个字符串字段，它包含已经读取的文件名称(与前面的示例一样)。注意在打开流时指定 `Unicode` 编码方式。如果要以其他格式写入文件，则只需要改变该参数的值。如果要把文本追加到文件中，这个构造函数的第二个参数就设置为 `true`，但本例不是这样。在构造时必须为 `StreamWriter` 设置编码方式，可以使用只读属性 `Encoding`。

下面介绍文件的读取方式。读取过程比较复杂，因为我们不知道要读取的文件中包含多少行文本(换言之，文件中包含多少个 `(char)13 - (char)10` 序列，因为 `char(13) - char(10)` 是行末的回车换行符)。解决这个问题方式是，先把文件读入一个 `StringCollection` 类的实例，该类在 `System.Collections.Specialized` 命名空间中，主要用于保存可动态扩展的一组字符串。它的两个方法是我们感兴趣的：把字符串添加到集合中的 `Add()` 和把字符串集合复制到一个数组(一个 `System.Array` 实例)中的 `CopyTo()`。 `StringCollection` 对象的每个元素包含文件中的一行文本。

DisplayFile()方法调用另一个方法 ReadFileIntoStringCollection(), 来读取文件。之后, 就知道文件中有多少行文本了。把 StringCollection 复制到大小固定的数组中, 并把数组中的内容填充到文本框中。在进行复制时, 只复制了字符串的引用, 没有复制字符串本身, 所以该过程的执行效率很高:

```
void DisplayFile()
{
    StringCollection linesCollection = ReadFileIntoStringCollection();
    string [] linesArray = new string[linesCollection.Count];
    linesCollection.CopyTo(linesArray, 0);
    this.textBoxContents.Lines = linesArray;
}
```

StringCollection.CopyTo()的第二个参数表示目标数组中的下标, 我们从该下标指定的位置开始复制集合。

下面看看 ReadFileIntoStringCollection()方法。使用 StreamReader 读取每一行文本。编译时需要计算读取的字符数, 以确保不超出文本框的范围:

```
StringCollection ReadFileIntoStringCollection()
{
    const int MaxBytes = 65536;
    StreamReader sr = new StreamReader(chosenFile);
    StringCollection result = new StringCollection();
    int nBytesRead = 0;
    string nextLine;
    while ( (nextLine = sr.ReadLine()) != null)
    {
        nBytesRead += nextLine.Length;
        if (nBytesRead > MaxBytes)
            break;
        result.Add(nextLine);
    }
    sr.Close();
    return result;
}
```

这就是该示例的完整代码。

如果运行 ReadWriteText, 读取 NewFile.aspx 文件, 然后保存它, 该文件的格式就是 Unicode。任何常用的 Windows 应用程序(Notepad, Wordpad)都没有提供这种格式, 甚至 ReadWriteText 示例也只能在 Windows NT/2000/XP/2003/Vista/2008 下正确读取和显示文件。因为 Windows 9x 不支持 Unicode, 像 Notepad 这样的应用程序不能识别其他平台上的 Unicode 文件(如果从 Wrox Press 网站上下载了这个示例, 就可以试试)。但是, 如果使用前面的 ReadBinaryFile 示例显示文件, 就会立即看出它们的区别, 如图 25-12 所示。最前面的两个字节表示文件的格式是 Unicode, 之后, 每个字符都用两个字节来表示。这是非常明显的, 因为在这个文件中, 每个字符的高位字节都是 0, 所以每隔一个字节就显示 x00。

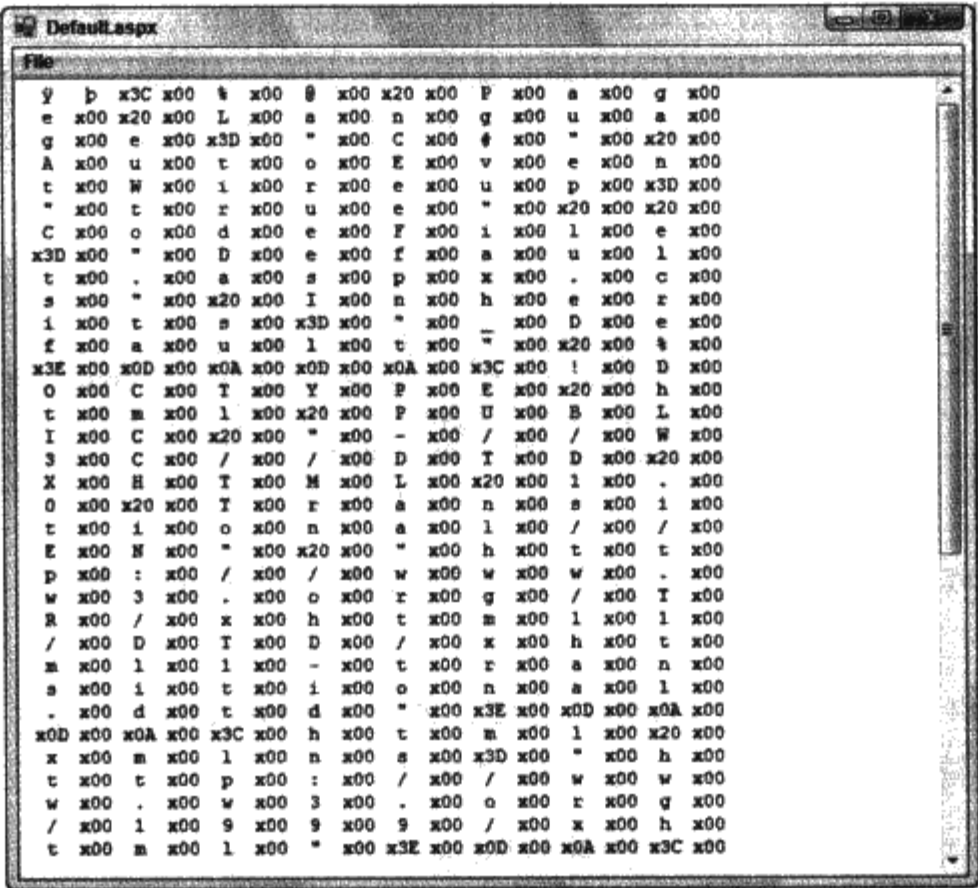


图 25-12

25.4 读取驱动器信息

除了处理文件和目录之外，.NET Framework 2.0 还可以从指定的驱动器中读取信息。这是使用 DriveInfo 类实现的。DriveInfo 类可以扫描系统，提供可用驱动器的列表，还可以进一步提供驱动器的大量细节。

为了演示 DriveInfo 类的用法，创建一个简单的 Windows 窗体，列出计算机上的所有驱动器，再提供用户选择的驱动器的信息。Windows 窗体包含一个简单的 ListBox，如图 25-13 所示。

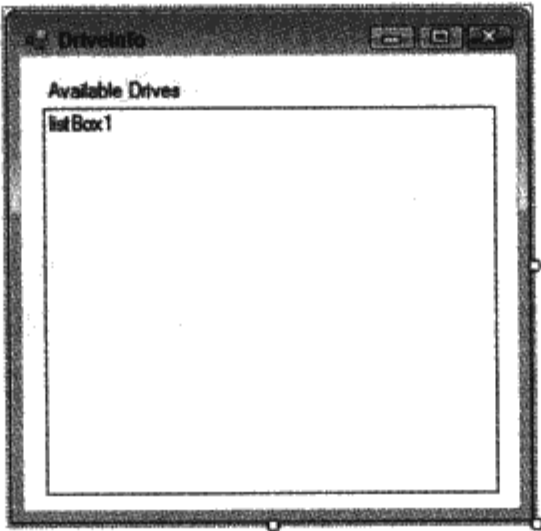


图 25-13



建立好窗体后，其代码包含两个事件，一个在窗体加载时引发，另一个在终端用户从列表框中选择驱动器时引发。这个窗体的代码如下所示：

```
using System;
using System.Windows.Forms;
using System.IO;

namespace DriveInfo
{
    partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            DriveInfo[] di = DriveInfo.GetDrives();

            foreach (DriveInfo itemDrive in di)
            {
                listBox1.Items.Add(itemDrive.Name);
            }
        }

        private void listBox1_SelectedIndexChanged(object sender, EventArgs e)
        {
            DriveInfo di = new DriveInfo(listBox1.SelectedItem.ToString());

            MessageBox.Show("Available Free Space: "
                + di.AvailableFreeSpace + "\n" +
                "Drive Format: " + di.DriveFormat + "\n" +
                "Drive Type: " + di.DriveType + "\n" +
                "Is Ready: " + di.IsReady.ToString() + "\n" +
                "Name: " + di.Name + "\n" +
                "Root Directory: " + di.RootDirectory + "\n" +
                "ToString() Value: " + di.ToString() + "\n" +
                "Total Free Space: " + di.TotalFreeSpace + "\n" +
                "Total Size: " + di.TotalSize + "\n" +
                "Volume Label: " + di.VolumeLabel.ToString(), di.Name +
                " DRIVE INFO");
        }
    }
}
```

首先，用 `using` 关键字导入 `System.IO` 命名空间。在 `Form1_Load` 事件中使用 `DriveInfo` 类获取系统上的所有驱动器。这需要使用 `DriveInfo` 对象数组，再用 `DriveInfo.GetDrives()` 方法填充这个数组。接着使用 `foreach` 循环迭代找到的每个驱动器，用循环的结果填充列表框。这会生成如图 25-14 所示的结果。





图 25-14

这个窗体允许最终用户在列表框中选择一个驱动器。选择好驱动器后，就显示一个消息框，其中包含所选驱动器的信息。如图 25-14 所示，当前的计算机上有 6 个驱动器。选择每个驱动器会生成如图 25-15 所示的消息框。



图 25-15

在这里可以看出，这些消息框列出了 3 个完全不同的驱动器。第一个驱动器 C:\是硬盘，因为消息框显示它的驱动器类型是 Fixed。第二个驱动器 D:\是 CD/DVD 驱动器，第三个驱动器 F:\是 USB 笔，其驱动器类型是 Removable。

## 25.5 文件的安全性

在第一次引入 .NET Framework 1.0/1.1 时，不能方便地访问和使用文件、目录和注册表键的访问控制表(ACL)。那时，要访问 ACL，通常要进行一些 COM 交互操作，所以需要使用 ACL 的高级编程技巧。

自 .NET Framework 2.0 以来，这些都有了很大的变化，因为这个版本通过 System.Security.AccessControl 命名空间，使 ACL 的使用变得非常容易。利用这个新的命名空间，可以为文件、注册表键、网络共享、Active Directory 对象等处理安全设置。

### 25.5.1 从文件中读取 ACL

本节在一个使用 `System.Security.AccessControl` 的示例中，将为文件和目录使用 ACL。首先了解如何查看指定文件的 ACL。这个示例在一个控制台应用程序中完成，如下所示：

```
using System;
using System.IO;
using System.Security.AccessControl;
using System.Security.Principal;

namespace ConsoleApplication1
{
    internal class Program
    {
        private static string myFilePath;

        private static void Main()
        {
            Console.WriteLine("Provide full file path: ");
            myFilePath = Console.ReadLine();

            try
            {
                using (FileStream myFile = new FileStream(myFilePath,
                    FileMode.Open, FileAccess.Read))
                {
                    FileSecurity fileSec = myFile.GetAccessControl();

                    foreach (FileSystemAccessRule fileRule in
                        fileSec.GetAccessRules(true, true,
                            typeof(NTAccount)))
                    {
                        Console.WriteLine("{0} {1} {2} access for {3}", myFilePath,
                            fileRule.AccessControlType == AccessControlType.Allow ?
                                "provides" : "denies",
                            fileRule.FileSystemRights,
                            fileRule.IdentityReference.ToString());
                    }
                }
            }
            catch
            {
                Console.WriteLine("Incorrect file path given!");
            }

            Console.ReadLine();
        }
    }
}
```

为了使这个示例正常工作，首先引用 `System.Security.AccessControl` 命名空间，这样就可以在程序的后面访问 `FileSecurity` 和 `FileSystemAccessRule` 类了。

在检索到指定的文件，并放在 `FileStream` 对象中后，就使用 `File` 对象上的 `GetAccessControl` 方法提取该文件的 ACL。`GetAccessControl` 方法中的这些信息放在类 `FileSecurity` 中。这个类有对引用项的访问权限。每个访问权限都用一个 `FileSystemAccessRule` 对象表示。所以要使用 `foreach` 循环迭代 `FileSecurity` 对象中的所有访问权限。

对根目录的一个简单文本文件运行这个示例，结果如下所示：

```
Provide full file path: C:\Sample.txt
C:\Sample.txt provides FullControl access for BUILTIN\Administrators
C:\Sample.txt provides FullControl access for NT AUTHORITY\SYSTEM
C:\Sample.txt provides FullControl access for PUSHKIN\Bill
C:\Sample.txt provides ReadAndExecute, Synchronize access for BUILTIN\Users
```

下一节介绍读取目录的 ACL，而不是文件的 ACL。

## 25.5.2 从目录中读取 ACL

读取目录(而不是文件)的 ACL 信息，与前面的示例没有什么区别，其代码如下所示：

```
using System;
using System.IO;
using System.Security.AccessControl;
using System.Security.Principal;

namespace ConsoleApplication1
{
    internal class Program
    {
        private static string mentionedDir;

        private static void Main()
        {
            Console.WriteLine("Provide full directory path: ");
            mentionedDir = Console.ReadLine();

            try
            {
                DirectoryInfo myDir = new DirectoryInfo(mentionedDir);

                if (myDir.Exists)
                {
                    DirectorySecurity myDirSec = myDir.GetAccessControl();

                    foreach (FileSystemAccessRule fileRule in
                        myDirSec.GetAccessRules(true, true,
                            typeof(NTAccount)))
                    {
                        Console.WriteLine("{0} {1} {2} access for {3}",
                            mentionedDir, fileRule.AccessControlType ==
                                AccessControlType.Allow ? "provides" : "denies",
                            fileRule.FileSystemRights,
                            fileRule.IdentityReference.ToString());
                    }
                }
            }
            catch
            {
                Console.WriteLine("Incorrect directory provided!");
            }

            Console.ReadLine();
        }
    }
}
```

这个示例的不同之处是，它使用 `DirectoryInfo` 类，该类还包含一个 `GetAccessControl` 方法，来提取目录的 ACL 信息。运行这个例子的结果如下所示：

```
Provide full directory path: C:\Test
C:\Test provides FullControl access for BUILTIN\Administrators
C:\Test provides FullControl access for NT AUTHORITY\SYSTEM
C:\Test provides FullControl access for PUSHKIN\Bill
C:\Test provides 268435456 access for CREATOR OWNER
C:\Test provides ReadAndExecute, Synchronize access for BUILTIN\Users
C:\Test provides AppendData access for BUILTIN\Users
C:\Test provides CreateFiles access for BUILTIN\Users
```

ACL 的最后一个用法是使用 `System.Security.AccessControl` 命名空间添加和删除文件的 ACL 中的项。

### 25.5.3 添加和删除文件中的 ACL 项

还可以使用与前面示例相同的对象处理资源的 ACL。下面的示例修改了前面读取文件 ACL 信息的代码，在这个示例中，读取指定文件的 ACL，修改后再次读取它：

```
try
{
    using (FileStream myFile = new FileStream(myFilePath,
        FileMode.Open, FileAccess.ReadWrite))
    {
        FileSecurity fileSec = myFile.GetAccessControl();

        Console.WriteLine("ACL list before modification:");

        foreach (FileSystemAccessRule fileRule in
            fileSec.GetAccessRules(true, true,
                typeof(System.Security.Principal.NTAccount)))
        {
            Console.WriteLine("{0} {1} {2} access for {3}", myFilePath,
                fileRule.AccessControlType == AccessControlType.Allow ?
                "provides" : "denies",
                fileRule.FileSystemRights,
                fileRule.IdentityReference.ToString());
        }

        Console.WriteLine();
        Console.WriteLine("ACL list after modification:");

        FileSystemAccessRule newRule = new FileSystemAccessRule(
            new System.Security.Principal.NTAccount(@"PUSHKIN\Tuija"),
            FileSystemRights.FullControl,
            AccessControlType.Allow);

        fileSec.AddAccessRule(newRule);
        File.SetAccessControl(myFilePath, fileSec);

        foreach (FileSystemAccessRule fileRule in
            fileSec.GetAccessRules(true, true,
                typeof(System.Security.Principal.NTAccount)))
        {
            Console.WriteLine("{0} {1} {2} access for {3}", myFilePath,
                fileRule.AccessControlType == AccessControlType.Allow ?
```



```
"provides" : "denies",
fileRule.FileSystemRights,
fileRule.IdentityReference.ToString());
```

在这个示例中，给文件的 ACL 添加了一个新的访问规则。这是使用 `FileSystemAccessRule` 对象完成的。`FileSystemAccessRule` 类是一个抽象的访问控制项（ACE）实例。ACE 定义了要使用的用户账户、这个用户账户可以处理的访问类型以及是允许还是拒绝这个访问。在创建这个对象的新实例时，创建了一个新的 `NTAccount` 对象，并给文件赋予 Full Control 权限。即使创建了新的 `NTAccount` 对象，仍必须引用已有的用户。接着使用 `FileSecurity` 类的 `AddAccessRule` 方法赋予新规则。之后，使用 `FileSecurity` 对象引用，通过 `File` 类的 `SetAccessControl` 方法给当前文件设置访问控制。

接着，再次列出文件的 ACL。下面是上述代码的执行结果：

```
Provide full file path: C:\Sample.txt
ACL list before modification:
C:\Sample.txt provides FullControl access for BUILTIN\Administrators
C:\Sample.txt provides FullControl access for NT AUTHORITY\SYSTEM
C:\Sample.txt provides FullControl access for PUSHKIN\Bill
C:\Sample.txt provides ReadAndExecute, Synchronize access for BUILTIN\Users

ACL list after modification:
C:\Sample.txt provides FullControl access for PUSHKIN\Tuija
C:\Sample.txt provides FullControl access for BUILTIN\Administrators
C:\Sample.txt provides FullControl access for NT AUTHORITY\SYSTEM
C:\Sample.txt provides FullControl access for PUSHKIN\Bill
C:\Sample.txt provides ReadAndExecute, Synchronize access for BUILTIN\Users
```

要从 ACL 列表中删除规则，并不需要对代码进行很多修改。在上面的代码示例中，只需把下面的一行

```
fileSec.AddAccessRule(newRule);
```

改为：

```
fileSec.RemoveAccessRule(newRule);
```

就删除了刚才添加的规则。

## 25.6 读写注册表

自 Windows 95 以来的所有 Windows 版本中，注册表是包含 Windows 安装、用户配置、以及已安装软件和设备的所有配置信息的核心存储库。目前，几乎所有的商用软件都使用注册表来存储这些信息，COM 组件必须把它的信息存储在注册表中，才能由客户程序调用。.NET Framework 及其 0 压缩安装概念略微减弱了注册表的重要性，因为程序集是完全自包含的，所以程序集的信息不需要放在注册表中，即使是共享程序集也是这样。另外，.NET Framework 引入了独立存储器的概念，通过它应用程序可以在文件中存储专用于每个用户的信息，.NET



Framework 将确保为每个在机器上注册的用户单独存储数据。

应用程序现在使用 Windows Installer 来安装，开发人员不再需要直接操作注册表来安装应用程序了。但是，如果发布完整的应用程序，应用程序也可能要使用注册表来保存配置信息。例如，如果应用程序要显示在控制面板的 Add/Remove Programs 对话框中，仍需要使用相应的注册表项。还需要使用注册表处理与原有代码的向后兼容性。

注册表的库和 .NET 库一样复杂，它包括访问注册表的类。其中有两个类涉及到注册表，即 Registry 和 RegistryKey，这两个类都在 Microsoft.Win32 命名空间中。在介绍这两个类前，先简要介绍一下注册表本身的结构。

### 25.6.1 注册表

注册表的层次结构非常类似于文件系统。查看和修改注册表内容的一般方式是使用 regedit 或 regedt32。其中，regedit 在所有的 Windows 版本中都有，自从 Windows 95 开始就是标准版本了，而 regedt32 则在 Windows NT 和 Windows 2000 中才有，其用户友好性不如 regedit，但可以访问 regedit 不能访问的安全性信息。Windows Server 2003 把 regedit 和 regedt32 合并为一个新的编辑器 regedit。这里只讨论 Windows XP Professional 的 regedit，在 Run...对话框或命令行提示符中键入 regedit，即可运行它。

运行 regedit 后，得到屏幕图 25-16。

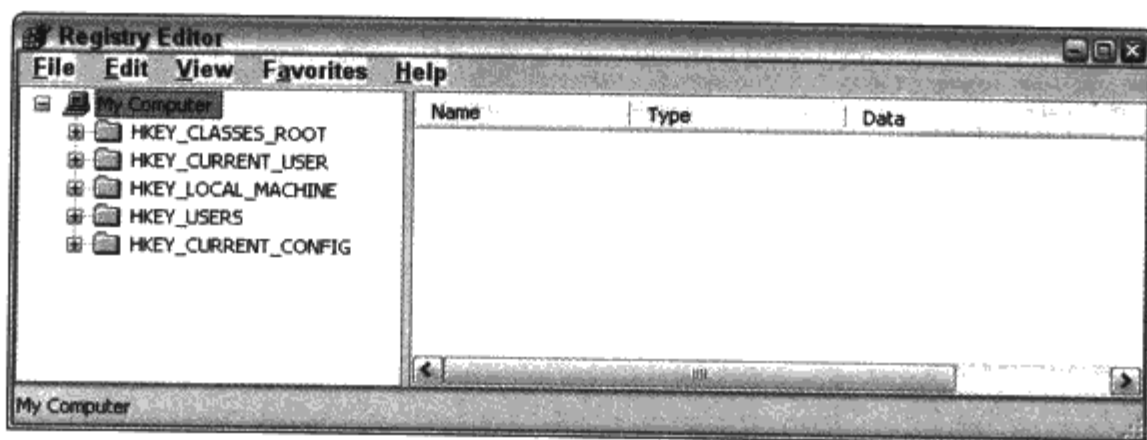


图 25-16

regedit 的树形视图/列表视图风格的用户界面非常类似于 Windows Explorer，与注册表本身的层次结构相匹配。但它们是有一些区别的。

在文件系统中，最上面的节点是磁盘的分区 C:\、D:\等。在注册表中，最上面的节点是注册表巢(registry hive)，它们的位置是不能改变的——它们是固定的，有 7 个注册表巢(但使用 regedit 只能看到其中的 5 个)：

- HKEY\_CLASSES\_ROOT(HKCR)包含系统上文件类型的细节(.txt、.doc 等)，以及应用程序可以打开的文件类型。它还包含所有 COM 组件的注册信息(后者通常是注册表中最大的一个区域，因为目前的 Windows 带有非常多的 COM 组件)。
- HKEY\_CURRENT\_USER(HKCU)包含用户目前登录的机器的用户配置。这些设置包括桌面设置、环境变量、网络和打印机连接和其他定义用户操作环境的设置。

- HKEY\_LOCAL\_MACHINE(HKLM)是一个很大的巢，其中包含所有安装到机器上的软件和硬件信息，这些设置不是用户特有的，而是可用于所有登录到机器上的用户。它还包含 HKCR 巢：HKCR 实际上并不是一个独立的巢，而只是一个对注册表项 HKLM/SOFTWARE/ Classes 的方便映射。
- HKEY\_USERS(HKUSR)包含所有用户的用户配置。它还包含 HKCU 巢，HKCU 巢是对 HKEY\_USERS 中一个键的映射。
- HKEY\_CURRENT\_CONFIG(HKCF)包含机器上硬件的信息。

其余的两个键包含临时信息，这些信息常常会更改：

- HKEY\_DYN\_DATA 是一个一般容器，包含需要存储在注册表中的违规数据。
- HKEY\_PERFORMANCE\_DATA 包含与运行应用程序的性能相关的信息。

在这些巢中，有一个注册表键的树形结构。每个键在许多方面都类似于文件系统中的文件夹或文件。但是，它们有一个重要区别：文件系统可以区分文件(文件中包含数据)和文件夹(其中主要包含其他文件夹或文件)，但注册表中只有键。键可以包含数据和其他键。

如果键中包含数据，这个键就表示为一组值。每个值都有一个相关的名称、数据类型和数据，另外，键还可以有默认值，这个值是没有名称的。

使用 regedit 可以查看这个结构，了解其中的注册表项。屏幕图 25-17 显示了键 HKCU/Control Panel/Appearance 中的内容，其中包含当前登录用户所选的颜色模式的信息。Regedit 在树形视图中用一个打开的文件夹图标来显示要查看的键。

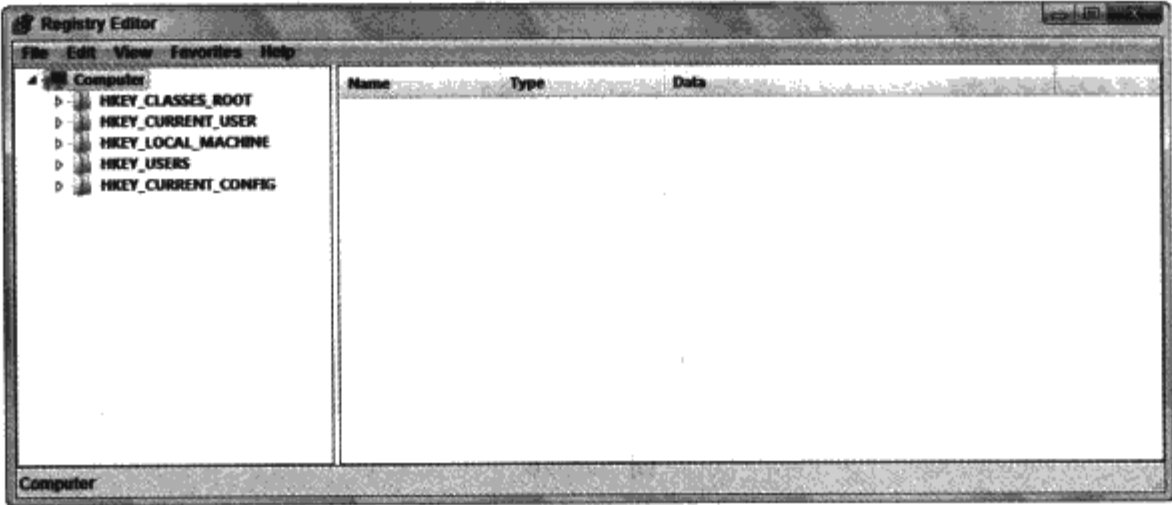


图 25-17

HKCU/Control Panel/Appearance 键有 3 个命名的值集，但其默认值不包含任何数据。在图 25-17 中，标记为 Type 的列显示了每个值的数据类型。注册表项可以格式化为这 3 个数据类型中的一个。这些类型分别是：

- REG\_SZ(大致相当于.NET 字符串实例——这种匹配并不精确，因为注册表项的数据类型不是.NET 数据类型)
- REG\_DWORD(大致相当于 uint)
- REG\_BINARY(大致相当于 byte)。

要在注册表中存储数据的应用程序会创建许多注册键，通常把它们存储在键 HKLM\Software\<CompanyName>中。注意，这些键不一定包含数据。有时键的存在就可以给应用程序提供足够的信息。

### 25.6.2 .NET 注册表类

要访问注册表，可以使用 Microsoft.Win32 命名空间中的两个类 `Registry` 和 `RegistryKey`。`RegistryKey` 实例表示一个注册表项，这个类的方法可以浏览子键、创建新键、读取或修改键中的值。换言之，该类可以完成对注册表项进行的所有操作，包括设置键的安全级别。`RegistryKey` 是处理注册表用得最多的类。`Registry` 类只能对注册表键进行单一的访问，执行简单的操作。`Registry` 的另一个作用是提供表示顶级键的 `RegistryKey` 实例(不同的巢)，以便开始在注册表中浏览。`Registry` 是通过静态属性来提供这些实例的，这些属性共有 7 个，分别是 `ClassesRoot`、`CurrentConfig`、`CurrentUser`、`DynData`、`LocalMachine`、`Performance Data` 和 `Users`。用户可以很快猜出它们分别与哪个巢相对应。

例如，要获得一个表示 HKLM 键的 `RegistryKey` 实例，可以编写下面的代码：

```
RegistryKey hklm = Registry.LocalMachine;
```

获得 `RegistryKey` 对象引用的过程，视为打开一个键。

用户可能会认为，因为注册表的层次结构类似于文件系统，所以 `RegistryKey` 的方法类似于 `DirectoryInfo` 的方法，但实际上并非如此。访问注册表的方式通常不同于使用文件和文件夹的方式，`RegistryKey` 执行的方法可以反映这种不同。

最明显的区别是如何在注册表的给定位置上打开一个注册表项。`Registry` 类没有用户可以使用公共构造函数，也没有直接通过键的名称来访问键的方法。但可以在相关的巢中从上至下浏览该键。如果要实例化一个 `RegistryKey` 对象，唯一的方式是从 `Registry` 的静态属性开始，向下浏览。例如，要读取 `HKLM/Software/Microsoft` 键中的一些数据，可以使用下面的代码获得它的一个引用：

```
RegistryKey hklm = Registry.LocalMachine;
RegistryKey hkSoftware = hklm.OpenSubKey("Software");
RegistryKey hkMicrosoft = hkSoftware.OpenSubKey("Microsoft");
```

以这种方式访问注册表项是只读访问。如果要写入该键(包括写入其值，或创建和删除其子键)，就需要使用 `OpenSubKey` 的另一个重写方法，该方法的第二个参数是 `bool` 类型，表示是否要对该键进行读写访问。例如，如果要修改 `Microsoft` 键(并假定用户是一个系统管理员，有修改该键的许可)，就应编写如下代码：

```
RegistryKey hklm = Registry.LocalMachine;
RegistryKey hkSoftware = hklm.OpenSubKey("Software");
RegistryKey hkMicrosoft = hkSoftware.OpenSubKey("Microsoft", true);
```

因为这个键包含 `Microsoft` 应用程序使用的信息，在大多数情况下，就不应修改这个特定键。

如果这个键已经存在，就应调用 `OpenSubKey()` 方法。如果这个键不存在，就返回一个空引用。如果要创建一个键，就应使用 `CreateSubKey()` 方法(该方法会通过返回的引用，自动提供该键的读写访问)：

```
RegistryKey hklm = Registry.LocalMachine;
RegistryKey hkSoftware = hklm.OpenSubKey("Software");
RegistryKey hkMine = hkSoftware.CreateSubKey("MyOwnSoftware");
```

CreateSubKey()工作的方式非常有趣：如果键不存在，它就创建这个键。但如果键已经存在，它就会返回一个表示该键的 RegistryKey 实例。这个方法采用这样的工作方式，其原因是用户总是可以使用这个键。注册表包含长期数据，例如 Windows 和各种应用程序的配置信息。因此用户并不需要经常显式地创建键。

更常见的是，应用程序需要确保某些数据在注册表中是存在的。换言之，如果这些数据不存在，就要创建相关的键，但如果它们存在，就不需要做任何事。CreateSubKey()就可以完成这项任务。与 FileInfo.Open()的情况不同，CreateSubKey()不会删除任何数据。如果要删除注册表项，就需要显式调用 RegistryKey.Delete()方法，因为注册表对于 Windows 是非常重要的。如果删除了一些重要的键，就会中断 Windows 的执行，此时就需要调试 C#注册表调用了。

定位了要读取或修改的注册表项后，就可以使用 SetValue() 或 GetValue()方法设置或获取该键中的值。这两个方法的参数都是一个字符串，其中字符串给出了值的名称，SetValue()还需要一个包含值的信息的对象引用。这个参数定义为对象引用，实际上可以是任何一个类的引用。SetValue()根据所提供的类的类型，确定把值设置为 REG\_SZ、REG\_DWORD，还是 REG\_BINARY。例如：

```
RegistryKey hkMine = HkSoftware.CreateSubKey("MyOwnSoftware");
hkMine.SetValue("MyStringValue", "Hello World");
hkMine.SetValue("MyIntValue", 20);
```

这段代码设置键包含两个值：MyStringValue 的类型是 REG\_SZ，而 MyIntValue 的类型是 REG\_DWORD，这里只考虑这两种类型，在后面的示例中会使用它们。

RegistryKey.GetValue()的工作方式也是这样。它返回一个对象引用，如果该方法检测到值的类型为 REG\_SZ，就返回一个字符串引用，如果值的类型为 REG\_DWORD，就返回一个 int 型值。

```
string stringValue = (string)hkMine.GetValue("MyStringValue");
int intValue = (int)hkMine.GetValue("MyIntValue");
```

最后，完成了读取或修改数据后，应关闭该键：

```
hkMine.Close();
```

RegistryKey 有许多方法和属性。表 25-5 和 25-6 列出了其中最有用的方法和属性。

表 25-5

属 性 名	作 用
Name	键的名称(只读)
SubKeyCount	键的子键个数
ValueCount	键包含的值的个数

表 25-6

方 法 名	作 用
Close()	关闭键
CreateSubKey()	创建给定名称的子键(如果该子键已经存在, 就打开它)
DeleteSubKey()	删除指定的子键
DeleteSubKeyTree()	递归删除子键及其所有的子键
DeleteValue()	从键中删除一个指定的值
GetAccessControl()	返回指定注册表键的访问控制表(ACL), 这是.NET Framework 2.0 的新增方法
GetSubKeyNames()	返回包含子键名称的字符串数组
GetValue()	返回指定的值
GetValueKind()	返回指定的值, 检索其注册表数据类型
GetValueNames()	返回一个包含所有键值名称的字符串数组
OpenSubKey()	返回表示给定子键的 RegistryKey 实例引用
SetAccessControl()	把访问控制表(ACL)应用于指定的注册表键。
SetValue()	设置指定的值

25.6.3 SelfPlacingWindow 示例

下面用一个示例程序来说明注册表类的用法, 该示例叫做 SelfPlacingWindow, 它是一个简单的 C# Windows 应用程序, 几乎没有任何功能。唯一的功能是单击一个按钮, 打开一个标准 Windows 颜色对话框(由 System.Windows.Forms.ColorDialog 类表示), 让用户选择一种颜色, 使之成为窗体的背景色。

尽管缺乏功能, 但这个自我调节的窗口可以以一种重要而非常友好的方式放在本书开发的其他应用程序上。如果在退出应用程序前, 在屏幕上拖动该窗口, 改变它的大小, 最大化或最小化它, 该窗口都会记住新的位置、背景色, 这样下一次加载它时, 该窗口就会自动恢复用户上一次选择的显示方式。它可以记住这些信息, 因为它在关闭时, 把这些信息写入注册表。这样, 我们不仅介绍了.NET 注册表类, 还介绍了它们的典型用法, 任何商用的 Windows 窗体应用程序都可以以这种方式使用这个类。

SelfPlacingWindow 在注册表的 HKLM\Software\WroxPress\SelfPlacingWindow 键中存储信息, HKLM 通常用于存储应用程序的配置信息, 但要注意, 它不是用户指定的。如果要更熟练掌握某个应用程序, 还应复制 HK\_Users 集中的信息, 这样每个用户才能得到他们自己的配置。

注意:

如果在真正的.NET 应用程序执行这个功能, 就要考虑使用独立的存储器, 而不使用注册表来存储这个信息。另一方面, 因为独立的存储器只能用于.NET 中, 所以如果需要与非.NET 应用程序交互操作, 就需要使用注册表。

在最初运行示例时, 它会查找这个键, 但肯定找不到它, 因此它会使用在开发环境中设置



的默认大小、颜色和位置。该示例还有一个列表框，其中显示了从注册表中读取的信息。在第一次运行该示例时，得到屏幕图 25-18。



图 25-18

如果修改背景色，在屏幕上移动 SelfPlacingWindow 或重新设置它的大小，在应用程序退出前，会创建 HKLM\Software\WroxPress\SelfPlacingWindow 键，并在其中写入新的配置信息。使用 regedit 可以查看这些信息，如图 25-19 所示。

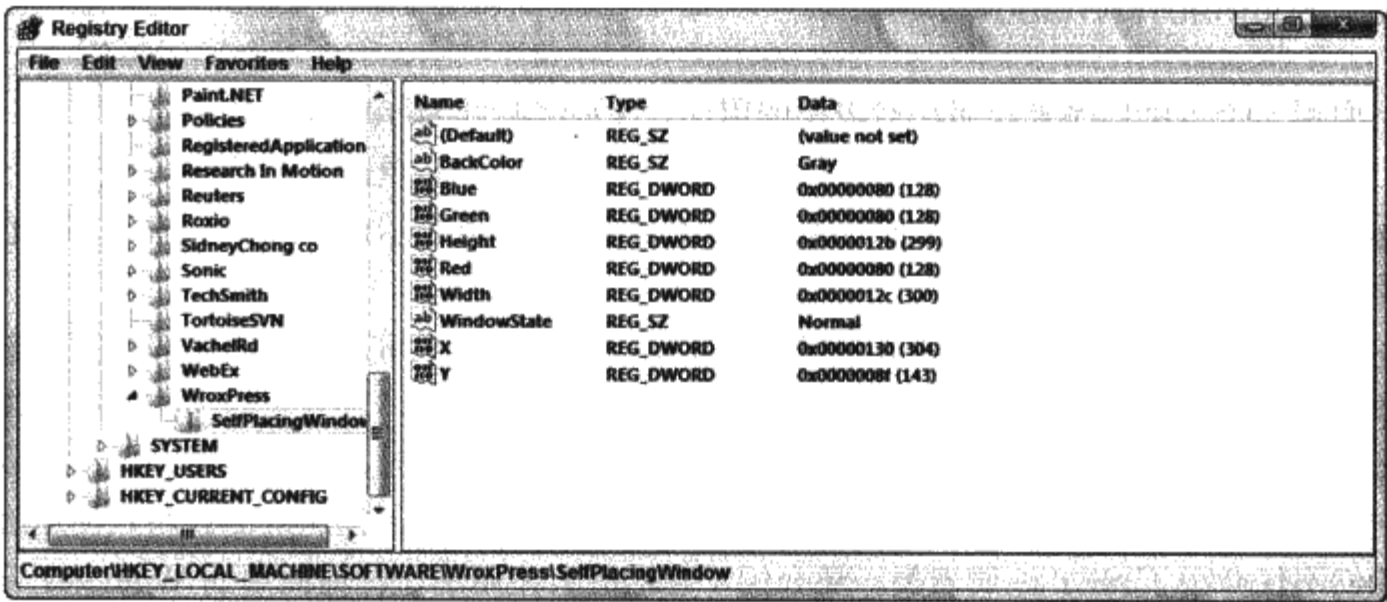


图 25-19

从这个屏幕图上可以看出，SelfPlacingWindow 在注册表项中放置了许多值。  
值 Red、Green 和 Blue 给出了组成所选背景色的颜色成分(参见第 33 章)。现在，系统上显示的颜色完全可以用这 3 种颜色成分描述，颜色成分用 0 到 255 之间的数字来表示(或者十六进制的 0x00 到 0xff)。这里给出的值组成了一种亮绿色。另外，还有 4 个 REG\_DWORD 值，它们表示窗口的位置和大小：X 和 Y 是窗口左上角在桌面上的坐标——即从屏幕的左上角向右的像素数，和向下的像素数。Width 和 Height 指定窗口的大小。WindowState 是这里唯一使用的字符串数据类型(REG\_SZ)，它可以包含 normal、maximised 或 minimised 字符串中的一个，这取决于退出应用程序时它的最终状态。

如果现在再次运行 SelfPlacingWindow, 它就会读取这个注册表项, 并自动定位, 如图 25-20 所示。

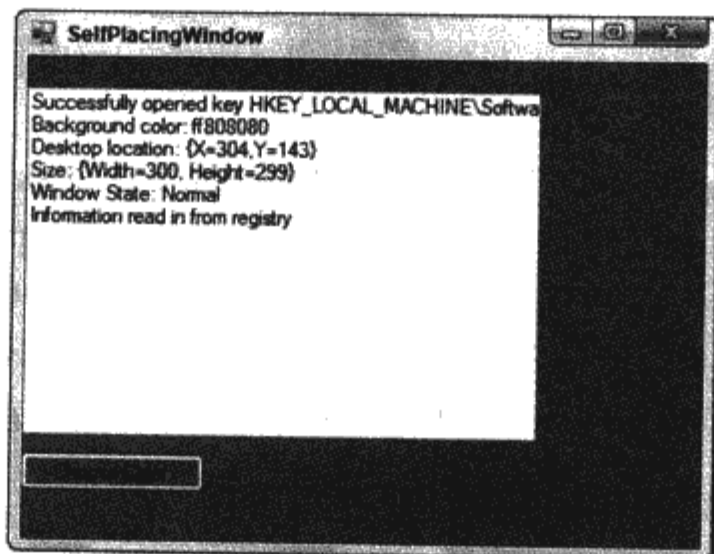


图 25-20

这次退出 SelfPlacingWindow 时, 它会用本次退出该示例时设置的新值重写原来的注册表设置。要开发该示例, 在 Visual Studio 中创建一个 Windows Forms 项目, 使用开发环境的工具箱添加列表框和按钮。把这些控件的名称分别改为 listBoxMessages 和 buttonChooseColor。还需要确保使用 Microsoft.Win32 命名空间:

```
using System;
using System.Drawing;
using System.Windows.Forms;
using Microsoft.Win32;
```

在主 Form1 类中添加一个字段(chooseColorDialog), 表示颜色对话框:

```
public class Form1 : System.Windows.Forms.Form
{
    private readonly ColorDialog chooseColorDialog = new ColorDialog();
```

在 Form1 构造函数中执行如下操作:

```
public Form1()
{
    InitializeComponent();
    buttonChooseColor.Click += OnClickChooseColor;
    try
    {
        if (ReadSettings() == false)
            listBoxMessages.Items.Add("No information in registry");
        else
            listBoxMessages.Items.Add("Information read in from registry");
        StartPosition = FormStartPosition.Manual;
    }
    catch (Exception e)
    {
        listBoxMessages.Items.Add("A problem occurred reading in data from registry:");
        listBoxMessages.Items.Add(e.Message);
    }
}
```

在这个构造函数中, 首先为用户单击按钮建立事件处理程序。该处理程序是一个方法 `OnClickChooseColor`(详见后面的内容)。使用另一个方法 `ReadSettings()` 读取配置信息, 如果该方法在注册表中找到信息, 就返回 `true`。否则就返回 `false`(应返回 `false`, 因为这是第一次运行该程序)。把这部分构造函数放在 `try` 块中, 以防读取注册表值时产生任何异常(如果用户使用 `regedit` 查看该注册表, 就有可能产生异常)。

`StartPosition = FormStartPosition.Manual;` 语句说明窗体位于 `DeskTopLocation` 属性的原始起始位置, 而不是使用 Windows 默认的位置(默认的操作)。其值是 `FormStartPosition` 枚举中的值。

`SelfPlacingWindow` 也是本书中为 `Dispose()` 方法添加代码的几个程序之一。在应用程序正常中断时, 就会调用 `Dispose()` 方法, 所以 `Dispose()` 方法是把配置信息保存到注册表中的理想位置。`Dispose()` 方法放在 `Form1.Designer.cs` 文件中。在这个方法中, 编写了另一个方法 `SaveSettings()`:

```
protected override void Dispose( bool disposing )
{
    if( disposing && (components != null))
    {
        components.Dispose();
    }
    SaveSettings();
    base.Dispose( disposing );
}
```

`SaveSettings()` 和 `ReadSettings()` 方法包含了我们感兴趣的注册表代码, 但在介绍这些代码前, 要先说明如何处理用户单击按钮的事件。该事件需要显示颜色对话框, 把背景色设置为用户选择的颜色:

```
void OnClickChooseColor(object Sender, EventArgs e)
{
    if(chooseColorDialog.ShowDialog() == DialogResult.OK)
        BackColor = chooseColorDialog.Color;
}
```

下面看看如何保存这些设置:

```
void SaveSettings()
{
    RegistryKey softwareKey =
        Registry.LocalMachine.OpenSubKey("Software", true);
    RegistryKey wroxKey = softwareKey.CreateSubKey("WroxPress");
    RegistryKey selfPlacingWindowKey =
        wroxKey.CreateSubKey("SelfPlacingWindow");
    selfPlacingWindowKey.SetValue("BackColor",
        (object)BackColor.ToKnownColor());
    selfPlacingWindowKey.SetValue("Red", (object)(int)BackColor.R);
    selfPlacingWindowKey.SetValue("Green", (object)(int)BackColor.G);
    selfPlacingWindowKey.SetValue("Blue", (object)(int)BackColor.B);
    selfPlacingWindowKey.SetValue("Width", (object)Width);
    selfPlacingWindowKey.SetValue("Height", (object)Height);
    selfPlacingWindowKey.SetValue("X", (object)DesktopLocation.X);
    selfPlacingWindowKey.SetValue("Y", (object)DesktopLocation.Y);
    selfPlacingWindowKey.SetValue("WindowState",
        (object)WindowState.ToString());
}
```

其中完成了许多任务。首先浏览注册表, 使用前面介绍的技巧, 从表示 `HKLM` 巢的静态

属性 `Registry.LocalMachine` 开始，找到 `HKLM\Software\WroxPress\SelfPlacingWindow` 键。

接着使用 `RegistryKey.OpenSubKey()` 方法，而不是 `RegistryKey.CreateSubKey()` 来获取 `HKLM\Software` 键，这是因为这个键已经存在，如果它不存在，计算机就会出现严重错误，这个键包含了许多系统软件的设置。还需要对这个键进行写入访问，这是因为，如果 `WroxPress` 键不存在，就需要创建它——这涉及到写入父键。

下一个要浏览的键是 `HKLM\Software\WroxPress`，我们不能肯定这个键是否存在，如果它不存在，使用 `CreateSubKey()` 方法自动创建它。注意 `CreateSubKey()` 自动提供对键的写入访问。得到了 `HKLM\Software\WroxPress\SelfPlacingWindow` 键后，就应多次调用 `RegistryKey.SetValue` 方法，创建或设置合适的值。注意还有两个比较复杂的问题。

首先，应注意这里使用了以前未见过的两个类。`Form` 类的 `DesktopPosition` 属性表示屏幕左上角的位置，其类型是 `Point`。第 33 章介绍了 `Point` 结构。`Point` 包含两个 `int` 值，`X` 和 `Y` 表示在屏幕上的水平和垂直位置。我们还使用了 `Form.BackColor` 属性的 3 种成员属性，它们是 `Color` 类的 3 个实例：`R`、`G` 和 `B`；`Color` 表示颜色，3 种属性给出了组成该颜色的红、绿、蓝 3 种颜色成分，它们的类型都是 `byte`。我们还使用了 `Form.WindowState` 属性，它包含一个枚举，该枚举给出了窗口的当前状态——`minimized`、`maximized` 和 `restored`。

另一个比较复杂的问题是需要仔细考虑数据的转换：`SetValue()` 带两个参数：一个给出键名的字符串和一个包含键值的 `System.Object` 实例。`SetValue` 对存储的值可以选择格式——键值可以存储为 `REG_SZ`、`REG_BINARY` 或 `REG_DWORD`。选择什么格式存储，主要取决于所给的数据类型。因此对于 `WindowState`，给它传送一个字符串，`SetValue()` 就会认为这个值应转换为 `REG_SZ`。同样，对于所提供的各种位置和大小，应把 `int` 转换为 `REG_DWORD`。但是，颜色成分要比把它们存储为 `REG_DWORD` 复杂得多，因为它们是数字类型。如果 `SetValue()` 认为这个数据的类型是 `byte`，就会把它存储为字符串，即注册表中的 `REG_SZ`。为了避免这个问题，应把颜色成分转换为 `int` 型。

我们还把所有的值的数据类型都显式转换为 `Object` 类型，实际上并不需要这么做，把任何数据类型转换为 `Object` 都可以隐式进行，但为了使过程更清晰，理解 `SetValue` 的第二个参数是一个对象引用，我们进行了显式转换。

因为要读取每个值，所以 `ReadSettings()` 方法比较长，在此还需解释它，在列表框中显示它，并对主窗体的相应属性进行适当的调整。`ReadSettings()` 如下所示。

```
bool ReadSettings()
{
    RegistryKey softwareKey =
        Registry.LocalMachine.OpenSubKey("Software");
    RegistryKey wroxKey = softwareKey.OpenSubKey("WroxPress");
    if (wroxKey == null)
        return false;
    RegistryKey selfPlacingWindowKey =
        wroxKey.OpenSubKey("SelfPlacingWindow");
    if (selfPlacingWindowKey == null)
        return false;
    else
        listBoxMessages.Items.Add("Successfully opened key " +
            selfPlacingWindowKey.ToString());
    int redComponent = (int)selfPlacingWindowKey.GetValue("Red");
    int greenComponent = (int)selfPlacingWindowKey.GetValue("Green");
}
```



```

int blueComponent = (int)selfPlacingWindowKey.GetValue("Blue");
BackColor = Color.FromArgb(redComponent, greenComponent,
    blueComponent);
listBoxMessages.Items.Add("Background color: " + BackColor.Name);
int X = (int)selfPlacingWindowKey.GetValue("X");
int Y = (int)selfPlacingWindowKey.GetValue("Y");
DesktopLocation = new Point(X, Y);
listBoxMessages.Items.Add("Desktop location: " +
    DesktopLocation);
this.Height = (int)selfPlacingWindowKey.GetValue("Height");
this.Width = (int)selfPlacingWindowKey.GetValue("Width");
listBoxMessages.Items.Add("Size: " + new
    Size(Width, Height));
string initialWindowState =
    (string)selfPlacingWindowKey.GetValue("WindowState");
listBoxMessages.Items.Add("Window State: " + initialWindowState);
this.WindowState = (FormWindowState)FormWindowState.Parse
    (WindowState.GetType(), initialWindowState);
return true;
}

```

在 `ReadSettings()` 中，首先浏览 `HKLM/Software/WroxPress/SelfPlacingWindow` 注册表项，但在本例中，要找到这个键，才能读取其中的内容。如果它不存在，在第一次运行示例时才会创建它。本例将停止读取键，而且不打算创建任何键。现在使用 `RegistryKey.OpenSubkey()` 方法。如果 `OpenSubkey()` 返回空引用，就表示该注册表项不存在，因此只能给调用代码返回 `false` 值。

在读取该键时，使用了 `RegistryKey.GetValue()` 方法，该方法返回一个对象引用，这表示这个方法可以返回任何类的实例。与 `SetValue()` 一样，它返回对应于键中数据类型的对象类，这样就可以假定 `REG_SZ` 键会提供一个字符串，其他键会提供一个整数。另外，还要转换 `SetValue()` 返回的引用。如果引发了异常，就说明有人打开了注册表，且改变了值的类型，因此这种转换就会引发异常，`Form1` 构造函数中的处理程序会捕获该异常。

这段代码的其余部分还使用了一个数据类型 `Size` 结构，`Size` 结构类似于 `Point` 结构，但用于表示大小，而不是坐标。它有两个成员属性 `Width` 和 `Height`，使用 `Size` 结构便于确定窗体的大小，以便显示列表框。

## 25.7 读写独立存储器

除了读写注册表之外，还可以读写独立存储器中的值。如果在写入注册表或磁盘时有问题，就可以使用独立存储器。它可以用于存储应用程序状态或用户设置。

独立存储器可以看作一个虚拟磁盘，在其中可以保存只能由创建它们的应用程序或与其他应用程序实例共享的数据项。独立存储器的访问类型有两种。第一种是由用户和程序集访问。

在用户和程序集访问独立存储器时，机器上有一个存储器位置，它可以由多个应用程序实例访问。这种访问是通过用户身份和应用程序（或程序集）身份来保证的，如图 25-21 所示。

这说明，同一个应用程序的多个实例可以在同一个存储器上工作。独立存储器的第二种访问类型是由用户、程序集和域来访问。在这种访问类型中，每个应用程序实例都在它自己的独立存储器中工作，如图 25-22 所示。



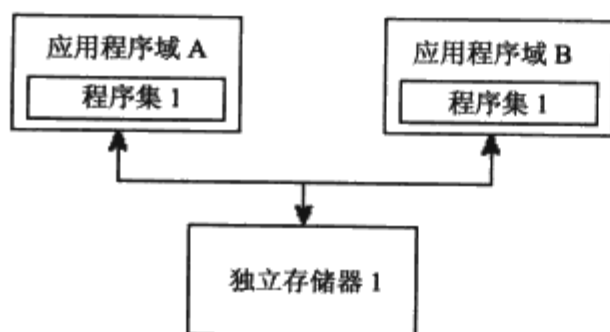


图 25-21

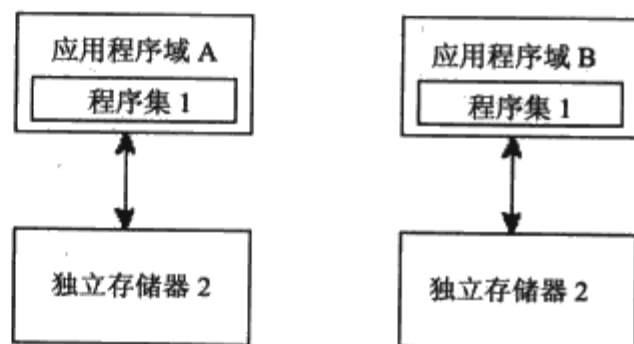


图 25-22

在这种情况下，每个应用程序实例都在它自己的存储器中工作，每个应用程序实例记录的设置都只与它自己相关。这是独立存储器的一种更精细的方法。

为了举例说明如何在 Windows 窗体应用程序中使用独立存储器（也可以在 ASP.NET 应用程序中使用它），下面修改本章前面使用的 SelfPlacingWindow 示例，演示如何将信息记录到注册表中。在新的 ReadSettings() 和 SaveSettings() 方法中，从独立存储器中读写值，而不是在注册表中读写值。

#### 提示：

这里只列出了 ReadSettings() 和 SaveSettings() 方法中的代码。该应用程序还有更多的代码，参见上一节中的其他代码。

首先，需要重写 SaveSettings() 方法。为了使代码正常工作，需要添加项目的 using 指令：

```
using System.IO;
using System.IO.IsolatedStorage;
using System.Text;
```

SaveSettings() 方法详见下面的代码示例：

```
void SaveSettings()
{
    IsolatedStorageFile storFile = IsolatedStorageFile.GetUserStoreForDomain();
    IsolatedStorageFileStream storStream = new
        IsolatedStorageFileStream("SelfPlacingWindow.xml",
        FileMode.Create, FileAccess.Write);

    System.Xml.XmlTextWriter writer = new
        System.Xml.XmlTextWriter(storStream, Encoding.UTF8);
    writer.Formatting = System.Xml.Formatting.Indented;

    writer.WriteStartDocument();
    writer.WriteStartElement("Settings");

    writer.WriteStartElement("BackColor");
    writer.WriteValue(BackColor.ToKnownColor().ToString());
    writer.WriteEndElement();

    writer.WriteStartElement("Red");
    writer.WriteValue(BackColor.R);
    writer.WriteEndElement();

    writer.WriteStartElement("Green");
    writer.WriteValue(BackColor.G);
    writer.WriteEndElement();
}
```

```

writer.WriteStartElement("Blue");
writer.WriteValue(BackColor.B);
writer.WriteEndElement();

writer.WriteStartElement("Width");
writer.WriteValue(Width);
writer.WriteEndElement();

writer.WriteStartElement("Height");
writer.WriteValue(Height);
writer.WriteEndElement();

writer.WriteStartElement("X");
writer.WriteValue(DesktopLocation.X);
writer.WriteEndElement();

writer.WriteStartElement("Y");
writer.WriteValue(DesktopLocation.Y);
writer.WriteEndElement();

writer.WriteStartElement("WindowState");
writer.WriteValue(WindowState.ToString());
writer.WriteEndElement();

writer.WriteEndElement();
writer.Flush();
writer.Close();

storStream.Close();
storFile.Close();
}

```

这些代码比注册表示例略多一些，这主要是因为需要建立放在独立存储器中的 XML 文档。在这段代码中，第一个重要的地方是：

```

IsolatedStorageFile storFile = IsolatedStorageFile.GetUserStoreForDomain();
IsolatedStorageFileStream storStream = new
    IsolatedStorageFileStream("SelfPlacingWindow.xml",
        FileMode.Create, FileAccess.Write);

```

这段代码使用访问的用户、程序集和域类型创建了 `IsolatedStorageFile` 的一个实例，使用 `IsolatedStorageFileStream` 对象创建了一个流，它将创建虚拟文件 `SelfPlacingWindow.xml`。

之后创建一个 `XmlTextWriter` 对象，建立 XML 文档，将 XML 内容写入 `IsolatedStorageFileStream` 对象实例。

```

System.Xml.XmlTextWriter writer = new
    System.Xml.XmlTextWriter(storStream, Encoding.UTF8);

```

创建 `XmlTextWriter` 对象之后，将所有的值逐个节点地写入 XML 文档。接着，关闭所有的对象。现在所有的数据就都存储在独立存储器中了。

从存储器中读取数据是通过 `ReadSettings()` 方法实现的。这个方法如下所示：

```

bool ReadSettings()
{
    IsolatedStorageFile storFile = IsolatedStorageFile.GetUserStoreForDomain();
    string[] userFiles = storFile.GetFileNames("SelfPlacingWindow.xml");

    foreach (string userFile in userFiles)

```

```

{
    if(userFile == "SelfPlacingWindow.xml")
    {
        listBoxMessages.Items.Add("Successfully opened file " +
            userFile.ToString());

        StreamReader storStream =
            new StreamReader(new IsolatedStorageFileStream("SelfPlacingWindow.xml",
                FileMode.Open, storFile));
        System.Xml.XmlTextReader reader = new
            System.Xml.XmlTextReader(storStream);

        int redComponent = 0;
        int greenComponent = 0;
        int blueComponent = 0;

        int X = 0;
        int Y = 0;

        while (reader.Read())
        {
            switch (reader.Name)
            {
                case "Red":
                    redComponent = int.Parse(reader.ReadString());
                    break;
                case "Green":
                    greenComponent = int.Parse(reader.ReadString());
                    break;
                case "Blue":
                    blueComponent = int.Parse(reader.ReadString());
                    break;
                case "X":
                    X = int.Parse(reader.ReadString());
                    break;
                case "Y":
                    Y = int.Parse(reader.ReadString());
                    break;
                case "Width":
                    this.Width = int.Parse(reader.ReadString());
                    break;
                case "Height":
                    this.Height = int.Parse(reader.ReadString());
                    break;
                case "WindowState":
                    this.WindowState = (FormWindowState)FormWindowState.Parse
                        (WindowState.GetType(), reader.ReadString());
                    break;
                default:
                    break;
            }
        }

        this.BackColor =
            Color.FromArgb(redComponent, greenComponent, blueComponent);
        this.DesktopLocation = new Point(X, Y);

        listBoxMessages.Items.Add("Background color: " + BackColor.Name);
        listBoxMessages.Items.Add("Desktop location: " +
            DesktopLocation.ToString());
        listBoxMessages.Items.Add("Size: " + new Size(Width, Height).ToString());
        listBoxMessages.Items.Add("Window State: " + WindowState.ToString());
    }
}

```



```

        storStream.Close();
        storFile.Close();

        return true;
    }
}

```

使用 `GetFileNames()` 方法, `SelfPlacingWindow.xml` 文档会从独立存储器中弹出, 放在一个流中, 再使用 `XmlTextReader` 对象分析它。

```

IsolatedStorageFile storFile = IsolatedStorageFile.GetUserStoreForDomain();
string[] userFiles = storFile.GetFileNames("SelfPlacingWindow.xml");

foreach (string userFile in userFiles)
{
    if(userFile == "SelfPlacingWindow.xml")
    {
        listBoxMessages.Items.Add("Successfully opened file " +
                                   userFile.ToString());

        StreamReader storStream =
            new StreamReader(new IsolatedStorageFileStream("SelfPlacingWindow.xml",
                                                            FileMode.Open, storFile));
    }
}

```

XML 文档包含在 `IsolatedStorageFileStream` 对象中后, 就使用 `XmlTextReader` 对象分析它:

```

System.Xml.XmlTextReader reader = new
    System.Xml.XmlTextReader(storStream);

```

之后, 通过 `XmlTextReader` 对象从流中读取它。元素值会放在应用程序中。`SelfPlacingWindow` 示例使用注册表记录和检索应用程序状态值, 而使用独立存储器也是有效的。应用程序会与以前一样记录颜色、大小和位置。

## 25.8 小结

本章介绍了如何在 C# 代码中使用 .NET 基类来访问注册表和文件系统, 在这两种情况下, 基类的对象模型比较简单, 但功能强大, 很容易执行这些领域中几乎所有的操作。对于文件系统, 可以复制文件、移动、创建、删除文件和文件夹, 读写二进制文件和文本文件, 而对于注册表, 可以创建、修改和读取键。

本章还介绍了独立存储器以及如何在应用程序中使用它们存储应用程序状态。

本章假定用户在有足够访问权限的账户上运行代码。显然, 安全性对于文件访问是非常重要的, 详见第 20 章。

下一章介绍数据访问、ADO.NET、XML 和 XML 模式。

# 第26章

## .NET 数据访问

本章讨论如何使用 ADO.NET 获取 C# 程序中的数据，主要介绍如下内容：

- 连接数据库：如何使用 `SqlConnection` 和 `OleDbConnection` 类连接数据库，以及断开与数据库的连接。
- 执行命令：ADO.NET 有命令对象的概念，该对象可以直接执行 SQL 命令，也可以执行带返回值的存储过程。这里将深入讨论命令对象上的各种选项，并说明如何为 `Sql` 和 `OleDb` 类的每个选项使用命令。
- 存储过程：如何使用命令对象来调用存储过程，这些存储过程的结果如何集成到高速缓存在客户机上的数据中。
- ADO.NET 对象模型：这与 ADO 中可用的对象完全不同，本节将讨论 `DataSet`、`DataTable`、`DataRow` 和 `DataColumn` 类。`DataSet` 也可以包含表之间、约束之间的关系。类层次结构在 .NET Framework 2.0 版本中有许多变化，本章也将介绍这些变化。
- 使用 XML 和 XML 模式：我们将讨论 XML 框架，它是构建 ADO.NET 的基础。

Microsoft 还在 C# 3.0 中添加了对 LINQ 的支持。这个主题不在本章介绍，但为了保证完整性，还是做一个简要讨论。.NET 中的新数据访问功能可参见第 28、29 和 31 章。

与其他章节一样，可以从 Wrox 网站 [www.wrox.com](http://www.wrox.com) 上下载本章示例的代码。下面首先简要介绍 ADO.NET。

### 26.1 ADO.NET 概述

ADO.NET 比现有 API 在技术上高出很多。它与 ADO 仅仅是名称类似，类和访问数据的方法则完全不同。

ADO (ActiveX Data Objects) 是一个 COM 组件库，在过去的几年中，这些组件有许多版本。在目前的版本 2.8 中，ADO 主要包含 `Connection`、`Command`、`Recordset` 和 `Field` 对象。使用 ADO 时，要打开与数据库的连接，把一些数据选出来，放在记录集中，这些数据由字段组成，接着处理这些数据，并在服务器上进行更新，最后关闭连接。ADO 还引入了概念：断开连接的记录集，当不适合使连接打开相当长的时间时，就可以使用该概念。

ADO 还有几个没有完全解决的问题，其中最著名的就是笨拙的断开连接的记录集。对这个支持的需求要比以往“以 Web 为中心”的计算更高，所以需要一种新方式。ADO.NET 和 ADO (不仅仅是名称) 有许多相似之处，所以从 ADO 中升级不会很困难。而且，如果使用的是 SQL Server，它有一组托管的新类，可以很好地发挥出数据库的最佳性能。单是这一个理由，就足以迁移到



ADO.NET 了。

ADO.NET 附带了 4 个数据库客户命名空间，一个用于 SQL Server，另一个用于 Oracle，第 3 个用于 ODBC 数据源，第四个用于通过 OLEBC 实现的数据库。如果数据库不是 SQL Server 或 Oracle，就应使用 OLE DB，除非还能使用 ODBC。

26.1.1 命名空间

本章所有的示例都以某种方式访问数据。表 26-1 所示的命名空间提供了在.NET 数据访问中使用的类和接口。

表 26-1

命 名 空 间	说 明
System.Data	所有的一般数据访问类
System.Data.Common	各个数据提供程序共享(或重写)的类
System.Data.Odbc	ODBC 提供程序的类
System.Data.OleDb	OLE DB 提供程序的类
System.Data.ProviderBase	新的基类和连接类
System.Data.Oracle	Oracle 提供程序的类
System.Data.Sql	用于 SQL Server 数据访问的通用新接口和类
System.Data.SqlClient	Sql Server 提供程序的类
System.Data.SqlTypes	Sql Server 数据类型

下面两节列出 ADO.NET 中主要的类。

26.1.2 共享类

ADO.NET 包含许多类，无论是使用 SQL Server 类，还是使用 OLE DB 类，都可以使用它们。表 26-2 所示的类包含在 System.Data 命名空间中。

表 26-2

类	说 明
DataSet	这个对象主要用于断开连接，它包含一组 DataTable，以及这些表之间的关系
DataTable	数据的一个容器， DataTable 由一个或多个 DataColumn 组成，每个 DataColumn 由一个或多个包含数据的 DataRow 生成
DataRow	许多数值，类似于数据库表的一行，或电子数据表中的一行。
DataColumn	包含列的定义，例如名称和数据类型
DataRelation	DataSet 中两个 DataTable 之间的链接，用于外键码和主/从关系
Constraint	为 DataColumn (或一组数据列)定义规则，例如唯一值

下面两个类包含在 System.Data.Common 命名空间中。见表 26-3。

表 26-3

类	说 明
DataColumnMapping	用 DataTable 中的列名映射数据库中的列名
DataTableMapping	将数据库中的表名映射到 DataSet 中的 DataTable

26.1.3 数据库特定的类

除了共享类外，ADO.NET 还包含许多数据库特定的类。这些类执行一组在 System.Data 命名空间中定义的标准接口，允许类按照一般形式来使用。例如，SqlConnection 和 OleDbConnection 类派生于执行 IDbConnection 接口的 DbConnection 类。见表 26-4。

表 26-4

类	说 明
SqlCommand、OleDbCommand、OracleCommand 和 ODBCCommand	SQL 语句或存储过程调用的包装器，SqlCommand 示例详见本章后面的内容
SqlCommandBuilder、OleDbCommandBuilder、OracleCommandBuilder 和 ODBCCommandBuilder	用于从一个 SELECT 语句中生成 SQL 命令(例如 INSERT、UPDATE 和 DELETE 语句)
SqlConnection、OleDbConnection、OracleConnection 和 ODBCConnection	数据库连接。类似于 ADO Connection，示例详见本章后面的内容
SqlDataAdapter、OleDbDataAdapter、OracleDataAdapter 和 ODBCDataAdapter	用于存储选择、插入、更新和删除语句的类，因此可以用于生成 DataSet 和更新数据库，SqlDataAdapter 示例详见本章后面的内容
SqlDataReader、OleDbDataReader、OracleDataReader 和 ODBCDataReader	只向前的连接数据读取器，SqlDataReader 示例详见本章后面的内容
SqlParameter、OleDbParameter、OracleParameter 和 ODBCParameter	为存储过程定义一个参数，SqlParameter 示例详见本章后面的内容
SqlTransaction、OleDbTransaction、OracleTransaction 和 ODBCTransaction	数据库事务处理，包装在一个对象中

从上面的列表可以看出，每种对象都有 4 个类，分别用于 .NET 1.1 中的每个提供程序。在本章的后面部分，除非特别说明，否则前缀<provider>将用于表示所使用的类依赖于数据库提供程序。在 .NET 2.0 中，设计人员更新了这些类的层次结构。在 1.1 版本中，各个连接类的共同点是，它们都实现了 IConnection 接口，这在 .NET 2.0 中有了变化，因为现在它们都共享一个公共基类。其他类也是这样，如 Command、DataAdapter、DataReader 等也共享公共基类。

ADO.NET 类最重要的新特性是：它们是以断开连接的方式工作，这在目前以 Web 为中心的环境中是非常重要的。我们常常把服务(例如在线书店)构建为连接一个服务器，检索一些数

据,再在客户机上处理这些数据,之后重新连接服务器,把数据传送回去,进行处理。ADO.NET 的断开连接的本质就可以实现这种操作。

ADO 2.1 引入了断开连接的记录集,允许从数据库中检索数据,把它们传送给客户机,进行处理,再重新连接服务器。但它们使用起来常常很繁琐,因为断开连接的工作方式不是一开始就设计好的。ADO.NET 类则不同,除了一种情况(<provider> DataReader)外,它们都用于脱机处理数据库。

注意:

本章将介绍在 .NET Framework 中用于数据访问的类和接口,主要论述连接数据库时使用的 SQL 类,因为 Framework SDK 示例安装了一个 MSDE 数据库(SQL Server)。在大多数情况下, OleDb、Oracle 和 ODBC 类类似于 SQL 代码。

## 26.2 使用数据库连接

为了访问数据库,需要提供某种类型的连接参数,例如运行数据库的机器和登录证书。使用 ADO 的用户会很快熟悉 .NET 连接类 OleDbConnection 和 SqlConnection,图 26-1 显示了两个连接类及类的层次结构。

在 .NET 1.0 和 1.1 版本中有很大的变化,但使用连接类(和 ADO.NET 中的其他类)是向后兼容的。

在本章的示例中,使用 Northwind 数据库,它是和 .NET Framework SDK 示例一起安装的。下面的代码段说明了如何创建、打开和关闭 Northwind 数据库的连接。

```
using System.Data.SqlClient;

string source = "server=(local);" +
    "integrated security=SSPI;" +
    "database=Northwind";
SqlConnection conn = new SqlConnection(source);
conn.Open();

// Do something useful

conn.Close();
```

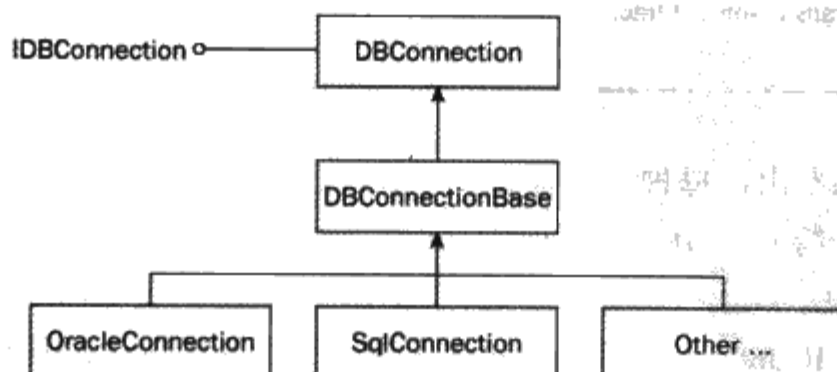


图 26-1

如果以前使用过 ADO 或 OLE DB,就会很熟悉连接字符串。如果使用的是 OleDb 提供程

序, 就应能剪切和粘贴旧代码。在该示例的连接字符串中, 使用的参数如下所示。连接字符串中的参数用分号分隔开。

- **server=(local)**: 表示要连接的数据库服务器。SQL Server 允许在同一台机器上运行多个不同的数据库服务器实例, 所以这里在本地机器上连接默认的 SQL Server 实例。如果使用 SQL Express, 就把服务器部分改为 **server=/sqlexpress**。
- **integrated security=SSPI**: 这个参数使用 Windows Authentication 连接数据库, 最好在源代码中使用这个参数, 而不是用户名和密码。
- **database=Northwind**: 这描述了要连接的数据库实例。每个 SQL Server 进程都可以有几个数据库实例。

#### 提示:

如果忘记数据库连接字符串的格式, 可以使用下面的 URL: [www.connectionstrings.com](http://www.connectionstrings.com)

这个示例使用定义的连接字符串打开数据库连接, 再关闭该连接。连接打开后, 就可以对数据源执行命令, 完成后, 就可以关闭连接。

SQL Server 有另一个模式的身份验证。它可以使用 Windows 集成的安全性, 这样在登录时提供的证书就会传送给 SQL Server。为此, 应删除连接字符串的 **uid** 和 **pwd** 部分, 并添加 **Integrated Security=SSPI**。

在本章的下载代码中, 有一个文件 **Login.cs** 简化了本章的示例。它链接了所有的示例代码, 包括用于这些示例的数据库连接信息; 可以修改该文件, 使用自己的服务器名称、用户名和密码。在默认情况下, 该文件使用 Windows 集成的安全性, 但是可以根据需要修改用户名和密码。

### 26.2.1 管理连接字符串

在以前的 .NET 版本中, 由开发人员管理数据库的连接字符串, 其方法常常是把连接字符串存储在应用程序配置文件中, 更常见的是, 在应用程序的某个地方硬编码连接字符串。

在 .NET 2.0 中, 有一种预定义的方式来存储连接字符串, 甚至是以类型未知的方式使用数据库连接。例如, 现在可以编写应用程序, 插入各个数据库提供程序, 而这些都无需修改主应用程序。

要定义数据库连接字符串, 应使用配置文件中新的 **<connectionStrings>** 段。在这里可以指定连接的名称、数据库连接字符串的参数, 还需要指定这个连接类型的提供程序。下面是一个例子:

```
<configuration>
...
<connectionStrings>
  <add name="Northwind"
        providerName="System.Data.SqlClient"
        connectionString="server=(local);integrated security=SSPI;
                           database=Northwind" />
</connectionStrings>
</configuration>
```

本章将在其他例子中使用这个连接字符串。

在配置文件中定义了数据库连接信息后，就需要在应用程序中利用该信息。我们常常要创建一个方法，根据连接的名称检索数据库连接：

```
private DbConnection GetDatabaseConnection ( string name )
{
   ConnectionStringSettings settings =
        ConfigurationManager.ConnectionStrings[name];

    DbProviderFactory factory = DbProviderFactories.GetFactory
        ( settings.ProviderName ) ;

    DbConnection conn = factory.CreateConnection ( ) ;
    conn.ConnectionString = settings.ConnectionString ;

    return conn ;
}
```

这段代码读取指定的连接字符串段(使用新的 `ConnectionStringSettings` 类)，再从一般的 `DbProviderFactories` 类中请求一个提供程序库，这要使用 `ProviderName` 属性，它在应用程序配置文件中设置为 `System.Data.SqlClient`。它会映射为实际的 `factory` 类，用于为 SQL Server 生成数据库连接，在本例中应使用 `System.Data.SqlClient` 中的 `SqlClientFactory` 类。必须添加对 `System.Configuration` 程序集的引用，才能解析上述代码使用的 `Configuration Manager` 类。

在 .NET 2.0 的 `machine.config` 文件中，有一个 `DbProviderFactories` 段，它把别名(如 `System.Data.SqlClient`)映射为该类数据库的 `factory` 对象。下面是一个该文件中删节的信息副本：

```
<system.data>
  <DbProviderFactories>
    ...
    <add name="SqlClient Data Provider"
        invariant="System.Data.SqlClient" support="FF"
        description=".Net Framework Data Provider for SqlServer"
        type="System.Data.SqlClient.SqlClientFactory, System.Data,
            Version=2.0.3600.0, Culture=neutral,
            PublicKeyToken=b77a5c561934e089" />
    ...
  </DbProviderFactories>
</system.data>
```

这段代码只显示了 `SqlClient` 提供程序的项，还有 `Odbc`、`OleDb`、`Oracle` 和 `SqlCE` 的项。

在这个例子中，`DbProviderFactory` 类只从机器配置设置中查找 `factory` 类，并使用具体的 `factory` 类实例化连接对象。对于 `SqlClientFactory` 类，就是要构建 `SqlConnection` 的一个实例，并把它返回给调用程序。

这对于获得数据库连接来说，似乎是不必要的工作，如果应用程序从来不运行在其他数据库上，这些工作的确没有必要。但如果使用前面的 `factory` 方法和一般的 `Db*` 类(例如 `DbConnection`、`DbCommand` 和 `DbDataReader`)，就会发现，以后将该应用程序迁移到另一个数据库系统上是非常简单的。



### 26.2.2 高效地使用连接

一般情况下，当在 .NET 中使用“稀缺”的资源时，例如数据库连接、窗口或图形对象，最好确保每个资源在使用完后立即关闭。尽管 .NET 的设计人员实现了自动的垃圾收集，垃圾最终都会被回收，但仍需要尽可能早地释放资源，以避免出现资源匮乏的情况。

当编写访问数据库的代码时，这是非常明显的，因为使连接打开的时间略长于需要的时间，就可能影响其他会话。在极端的情况下，不关闭连接会使其他用户无法进入一整组数据表，极大地降低了应用程序的性能。关闭数据库连接应是强制的，所以本节讨论如何构建代码，把一直打开资源的风险降到最低。

主要有两种方式可以确保数据库连接等类似的“稀缺”资源在使用完后立即释放。

#### 1. 第一种方式——利用 try...catch...finally 语句块

确保释放资源的第一种方式是利用 try...catch...finally 块，确保在 finally 块中关闭任何已打开的连接。下面是一个小示例：

```
try
{
    // Open the connection
    conn.Open();
    // Do something useful
}
catch ( Exception ex )
{
    // Log the exception
}
finally
{
    // Ensure that the connection is freed
    conn.Close ( ) ;
}
```

在 finally 块中，可以释放已经使用的任何资源。这种方式的唯一麻烦是必须确保关闭连接。很容易忘记在 finally 块中添加关闭连接的命令，所以应在编码风格上添加一些不容易出现反常情况的内容。

另外，在给定的方法中可能会打开许多资源(例如两个数据库连接和一个文件)，这样 try...catch...finally 块的层次有时可能不容易看懂。但还有另一个方式可以确保资源的关闭——使用 using 语句。

#### 2. 第二种方式——使用 using 语句块

在开发 C# 的过程中，.NET 在对象不再引用之后清理它们的方法是使用非决定性的析构方式，这已经引起了非常热烈的讨论。

在 C++ 中，对象只要使用完毕，就会自动调用其析构函数。这对于设计基于资源的类的人员来说，是一个非常好的消息，因为如果用户忘记关闭资源，使用析构函数是非常理想的。只要对象使用完毕，就会调用 C++ 析构函数。所以，如果出现了异常，但没有捕获，有析构函数的对象就会调用它们的析构函数。

在 C# 和其他托管语言中, 没有自动、决定性的析构方式, 而是有一个垃圾收集器, 它会在未来的某个时刻释放资源。它是非决定性的, 因为我们不能确定这个过程在什么时候发生。忘记关闭数据库连接可能会导致 .NET 可执行程序的各种问题。幸运的是, 我们还有解决的方法。下面的代码说明了如何使用 `using` 子句确保实现 `IDisposable` 接口(详见第 12 章)的对象在退出块时立即被释放。

```
string source = "server=(local);" +  
               " integrated security=SSPI;" +  
               "database=Northwind";
```

```
using ( SqlConnection conn = new SqlConnection ( source ) )  
{  
    // Open the connection  
    conn.Open ( ) ;  
  
    // Do something useful  
}
```

在这个实例中, 无论块是如何退出的, `using` 子句都会确保关闭数据库连接。

查看一下连接类的 `Dispose()` 方法的 IL 代码, 它们都检查连接对象的当前状态, 如果其状态为打开, 就调用 `Close()` 方法。浏览 .NET 程序集的一个强大工具是 `Reflector`(可以从 [www.aisto.com/roeder/dotnet/](http://www.aisto.com/roeder/dotnet/) 上获得)。这个工具允许查看任何 .NET 方法的 IL 代码, 还可以把 IL 代码反编译为 C# 源代码, 让我们轻松地确定给定方法的作用。

在编程时, 应至少使用这两个方法中的一个, 或者两个方法都使用。无论在哪里获得资源, 最好都使用 `using()` 语句, 因为尽管我们都会编写 `Close()` 语句, 但有时会忘记, 此时 `using` 子句就会发挥作用。这两种方式都没有好的异常处理方式来替代, 所以在大多数情况下, 最好组合使用这两种方法, 如下面的示例所示。

```
try  
{  
    using ( SqlConnection conn = new SqlConnection ( source ) )  
    {  
        // Open the connection  
        conn.Open ( ) ;  
  
        // Do something useful  
  
        // Close it myself  
        conn.Close ( ) ;  
    }  
} catch ( SqlException e )  
{  
    // Log the exception  
}
```

这里显式调用了 `Close()`, 但这是不必要的, 因为 `using` 子句将确保在任何情况下都执行关闭操作。但是, 应确保像这样的资源尽可能早地释放。因为在块的其余部分可能有更多的代码, 而在这些地方没有必要锁定资源。

另外, 如果在 `using` 块中出现了异常, `using` 子句就会确保在资源上调用 `IDisposable.Dispose` 方法, 在本例中将确保总是关闭数据库连接。这样, 与必须确保在异常子句中关闭连接相比,

代码的可读性更高。还要注意，异常定义为 `SqlException`，而不是捕获所有异常的 `Exception` 类型——应总是捕获特定的异常，把不显式处理的其他异常放在异常堆栈中。

最后，如果编写一个封装资源的类，无论该资源是什么，都应实现 `IDisposable` 接口，关闭资源。这样，任何使用该类的代码都可以利用 `using()` 语句，以确保资源被释放。

26.2.3 事务处理

通常，对数据库要进行多次更新，这些更新必须在事务处理的范围内进行。我们常常要在代码中查找一个事务处理读写，它传送给许多方法，以更新数据库，但在 .NET 2.0 及其更高版本中，给 `System.Transaction` 命名空间中添加了 `TransactionScope` 类，它极大地简化了事务处理代码的编写，因为可以把几个事务处理方法合并到一个事务处理范围中，事务处理流会根据需要执行每个方法。

下面的代码是在 `Sql Server` 连接上开始事务处理的：

```
string source = "server=(local);" +
               " integrated security=SSPI;" +
               "database=Northwind";

using (TransactionScope scope = new TransactionScope
    (TransactionScopeOption.Required))
{
    using (SqlConnection conn = new SqlConnection(source))
    {
        // Do something in SQL
        ...
        // Then mark complete
        scope.Complete();
    }
}
```

这段代码使用 `scope.Complete()` 方法把事务处理显式标记为完成。如果不调用这个方法，事务处理就会回滚，不对数据库进行任何修改。

在使用事务处理作用域时，可以选择在该事务处理中执行的命令的独立级别。该级别确定了如何在一个数据库会话中查看在另一个数据库会话中所进行的修改，并不是所有的数据库引擎都支持表 26-5 所示的 4 个级别。

表 26-5 事务处理的独立级别

独立级别	说 明
ReadCommitted	SQL Server 默认级别。这个级别可以确保只有第一个事务处理结束后，在第二个事务处理中才能访问第一个事务处理写入的数据。
ReadUncommitted	即使一个事务处理还没有处理完数据，也允许另一个事务处理从数据库中读取数据。例如，如果两个用户在访问同一个数据库，第一个用户插入一些数据，但没有完成事务处理（通过 <code>Commit</code> 或 <code>Rollback</code> 方法），第二个用户把它们的独立级别设置为 <code>ReadUncommitted</code> ，因此可以访问数据。

(续表)

独 立 级 别	说 明
RepeatableRead	<p>这个级别扩展了 ReadCommitted 级别，如果在事务处理中使用了相同的语句，无论是否有其他潜在的数据库更新，总是可以返回相同的数据。这个级别要求对数据进行额外的锁定，这会降低性能。</p> <p>这个级别可以保证，对于初始查询的每一行，都不会修改数据，但允许显示“假想(phantom)”行——这些行是在进行事务处理时，由另一个事务处理插入的全新数据行</p>
Serializable	<p>这是最“高级”的事务处理级别，对数据库中的数据进行串行化访问。利用这种独立级别，不会显示假想行，所以在可串行化的事务处理中使用的 SQL 语句总是检索相同的数据。</p> <p>可串行化的事务处理对性能的负面影响不应低估，如果不肯定是否需要这个独立级别，最好不要使用它</p>

SQL Server 的默认独立级别 ReadCommitted 是数据一致性和数据可用性的一种很好的折衷，因为它比 RepeatableRead 或 Serializable 模式中需要的数据锁定都少。但是，有时应提高独立级别，这样在.NET 中，才能用一种非默认的级别开始事务处理。使用哪个级别没有硬性规则，全凭经验。

注意：

如果当前使用的是不支持事务处理的数据库，应转而使用支持它的数据库。一旦我们成为可以完全信任的雇员，且拥有错误数据库的全部访问权限，就可能试键入 id=99999 以删除对应的错误，但实际上输入的是<而不是=，此时会删除整个错误数据库，这可不是我们希望的。幸好 IS 小组每天晚上都会备份该数据库，可以恢复它，但使用回滚命令会更简单。

26.3 命令

在 26.2 节中简要介绍了针对数据库执行的命令。简言之，命令就是一个要在数据库上执行的 SQL 文本字符串。命令也可以是一个存储过程，或者返回表中所有列和所有行的表名(例如 SELECT \*样式的子句)。

把 SQL 子句作为一个参数传递给 Command 类的构造函数，就可以构造一个命令，如下所示：

```
string source = "server=(local);" +
               " integrated security=SSPI;" +
               "database=Northwind";
string select = "SELECT ContactName,CompanyName FROM Customers";
SqlConnection conn = new SqlConnection(source);
conn.Open();

SqlCommand cmd = new SqlCommand(select, conn);
```

<provider>Command 类的属性 CommandType 可以定义某个命令是 SQL 子句、存储过程的调用、或完整的表语句(仅从给定的表中选择所有的列和行)。表 26-6 总结了 CommandType 枚举。

表 26-6

命令类型	样 例
Text(默认)	<pre>String select = "SELECT ContactName FROM Customers"; SqlCommand cmd = new SqlCommand(select, conn);</pre>
StoredProcedure	<pre>SqlCommand cmd=new SqlCommand("CustOrderHist", conn); cmd.CommandType = CommandType.StoredProcedure; cmd.Parameters.Add("@CustomerID", "QUICK");</pre>
TableDirect	<pre>OleDbCommand cmd = new OleDbCommand("Categories", conn); cmd.CommandType = CommandType.TableDirect;</pre>

在执行存储过程时，需要把参数传送给过程。上面的示例直接设置了参数@CustomerID，但设置参数的值还可以使用其他方式，详见本章的后面。注意在.NET 2.0 中，给命令参数集合添加了 AddWithValue 方法，而废弃了 Add(name, value)成员。如果习惯于使用这个构建参数的方法来调用存储过程，在重新编译代码时，会得到一个编译警告。最好现在就修改代码，因为 Microsoft 在.NET 的后续版本中将删除旧方法。

提示：

TableDirect 命令类型只对 OleDb 提供程序有效——如果试图把这个命令类型用于其他提供程序，就会产生异常。

26.3.1 执行命令

定义好命令后，就需要执行它们。执行语句有许多方式，这取决于要从命令中返回什么数据。<provider>Command 类提供了下述可执行的命令：

- ExecuteNonQuery()——执行命令，但不返回任何结果。
- ExecuteReader ()——执行命令，返回一个类型化的 IDataReader。
- ExecuteScalar ()——执行命令，返回一个值。

除了上述命令外，SqlCommand 类也提供了下面的方法：

- ExecuteXmlReader()——执行命令，返回一个 XmlReader 对象，它可以传送从数据库中返回的 XML 代码段。

与其他章节一样，可以从 Wrox 网站 [www.wrox.com](http://www.wrox.com) 上下载本节的示例代码。

1. ExecuteNonQuery()方法

这个方法一般用于 UPDATE、INSERT 或 DELETE 语句，唯一的返回值是受影响的记录



个数。但如果调用带输出参数的存储过程，该方法就有返回值：

```
using System;
using System.Data.SqlClient;
public class ExecuteNonQueryExample
{
    public static void Main(string[] args)
    {
        string source = "server=(local);" +
            " integrated security=SSPI;" +
            "database=Northwind";
        string select = "UPDATE Customers " +
            "SET ContactName = 'Bob' " +
            "WHERE ContactName = 'Bill'";
        SqlConnection conn = new SqlConnection(source);
        conn.Open();
        SqlCommand cmd = new SqlCommand(select, conn);
        int rowsReturned = cmd.ExecuteNonQuery();
        Console.WriteLine("{0} rows returned.", rowsReturned);
        conn.Close();
    }
}
```

ExecuteNonQuery()返回命令所操作的行数，它为一个整数。

## 2. ExecuteReader()方法

这个方法执行命令，根据使用的提供程序返回一个类型化的 DataReader 对象，返回的对象可以用于迭代返回的记录，如下面的代码所示。

```
using System;
using System.Data.SqlClient;
public class ExecuteReaderExample
{
    public static void Main(string[] args)
    {
        string source = "server=(local);" +
            " integrated security=SSPI;" +
            "database=Northwind";
        string select = "SELECT ContactName,CompanyName FROM Customers";
        SqlConnection conn = new SqlConnection(source);
        conn.Open();
        SqlCommand cmd = new SqlCommand(select, conn);
        SqlDataReader reader = cmd.ExecuteReader();
        while(reader.Read())
        {
            Console.WriteLine("Contact : {0,-20} Company : {1}" ,
                reader[0] , reader[1]);
        }
    }
}
```

图 26-2 显示了这段代码的结果。

本章的后面将讨论<provider>DataReader 对象。

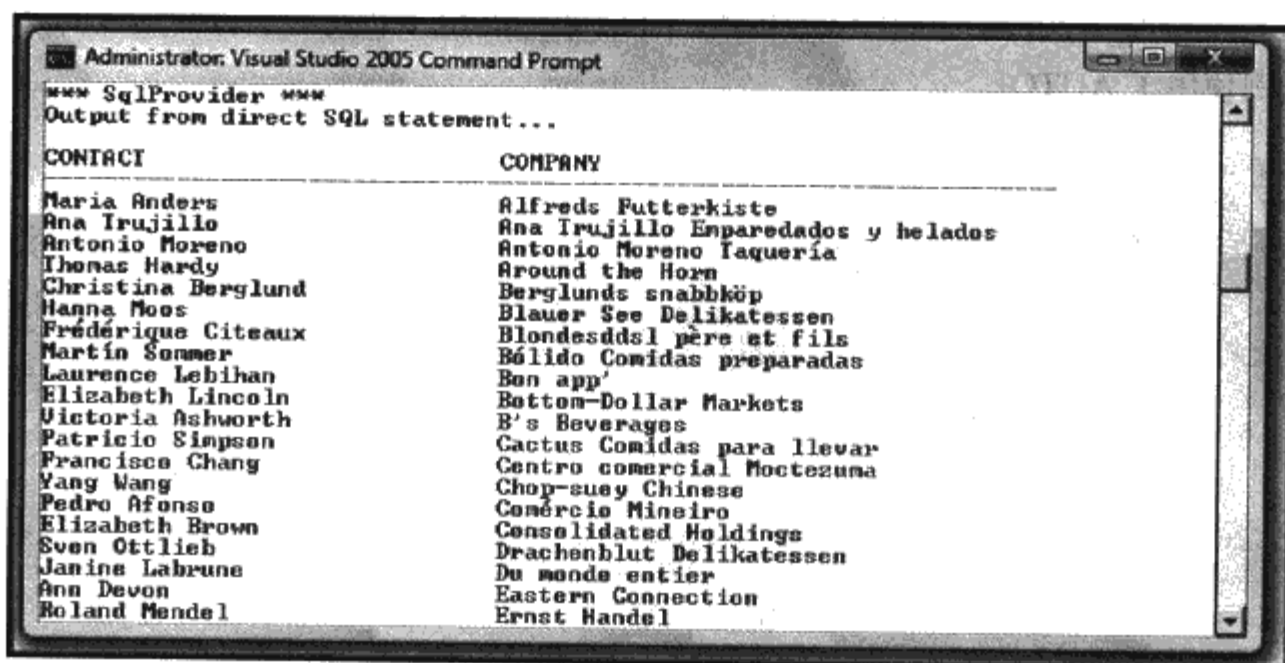


图 26-2

### 3. ExecuteScalar()方法

在许多情况下, 需要从 SQL 语句返回一个结果, 例如给定表中的记录个数, 或者服务器的当前日期/时间。ExecuteScalar 方法就可以用于这些场合:

```
using System;
using System.Data.SqlClient;
public class ExecuteScalarExample
{
    public static void Main(string[] args)
    {
        string source = "server=(local);" +
            " integrated security=SSPI;" +
            "database=Northwind";
        string select = "SELECT COUNT(*) FROM Customers";
        SqlConnection conn = new SqlConnection(source);
        conn.Open();
        SqlCommand cmd = new SqlCommand(select, conn);
        object o = cmd.ExecuteScalar();
        Console.WriteLine ( o );
    }
}
```

该方法返回一个对象, 如果需要, 可以把该对象的数据类型转换为合适的类型。如果所调用的 SQL 只返回一行, 最好使用 ExecuteScalar 方法来检索这一行。这也适合于只返回一个值的存储过程。

### 4. ExecuteXmlReader()方法(只用于 SqlConnection 提供程序)

顾名思义, 这个方法执行命令, 给调用程序返回一个 XmlReader 对象。SQL Server 允许使用 FOR XML 子句来扩展 SQL 子句, 这个子句可以带有下列 3 个选项中的一个:

- FOR XML AUTO: 根据 FROM 子句中的表建立一个树
- FOR XML RAW: 结果集中的行映射为元素, 其中的列映射为属性
- FOR XML EXPLICIT: 必须指定要返回的 XML 树的形状

*Professional SQL Server 2000 XML*(ISBN 1-861005-46-6)一书列出了这些选项的完整描述。下面的示例使用了 AUTO:

```
using System;
using System.Data.SqlClient;
using System.Xml;
public class ExecuteXmlReaderExample
{
    public static void Main(string[] args)
    {
        string source = "server=(local);" +
            " integrated security=SSPI;" +
            "database=Northwind";
        string select = "SELECT ContactName,CompanyName " +
            "FROM Customers FOR XML AUTO";
        SqlConnection conn = new SqlConnection(source);
        conn.Open();
        SqlCommand cmd = new SqlCommand(select, conn);
        XmlReader xr = cmd.ExecuteXmlReader();
        xr.Read();
        string data;
        do
        {
            data = xr.ReadOuterXml();
            if (!string.IsNullOrEmpty(data))
                Console.WriteLine(data);
        } while (!string.IsNullOrEmpty(data));
        conn.Close();
    }
}
```

注意,必须导入 System.Xml 命名空间,才能输出返回的 XML。这个命名空间和 .NET Framework 其他的 XML 功能将在第 28 章中详细论述。本例在 SQL 语句中包含了 FOR XML AUTO 子句,然后调用 ExecuteXmlReader()方法。代码的结果如图 26-3 所示。

在 SQL 子句中,我们指定了 FROM Customers,这样 Customers 类型的元素就显示在输出中。为它添加元素,每个元素对应于从数据库中选择出来的列。这就为每个从数据库中选择出来的行建立了 XML 标志。

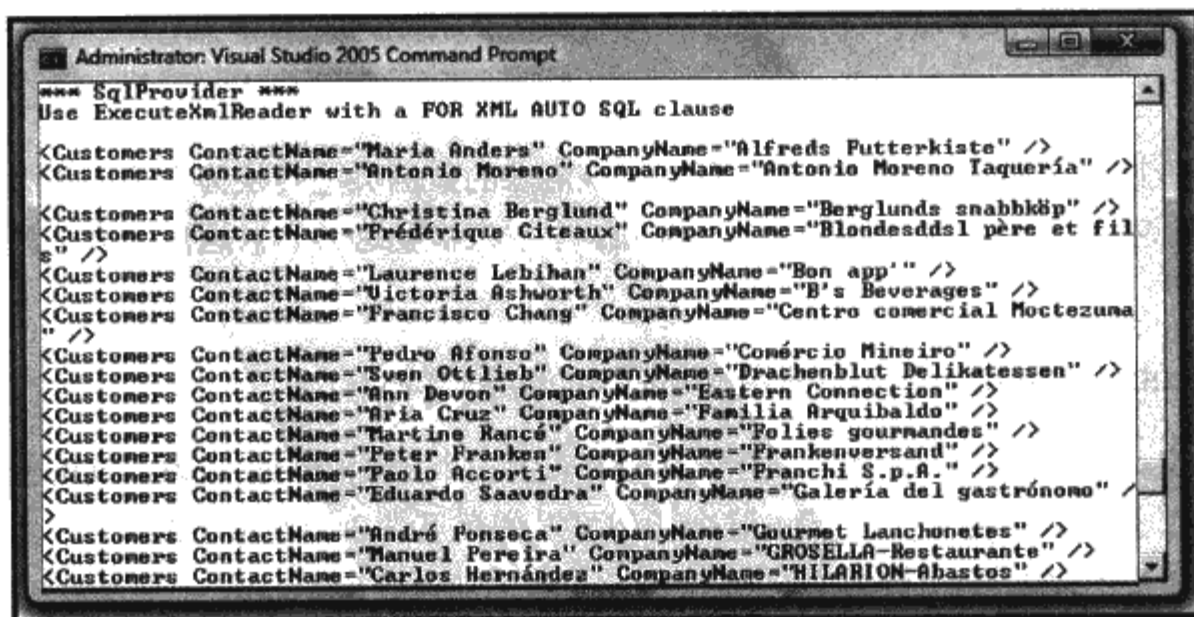


图 26-3

### 26.3.2 调用存储过程

用一个命令对象调用存储过程，就是定义存储过程的名称，给过程的每个参数添加参数定义，然后用上一节中给出的方法执行命令。

为了使本节的示例更有说服力，下面定义一组可以用于在 Northwind 示例数据库的 Region 表中插入、更新和删除记录的存储过程，这个表尽管很小，但可以用于给每种常见的存储过程编写示例。

#### 1. 调用没有返回值的存储过程

调用存储过程的最简单示例是不给调用者返回任何值。下面定义了两个这样的存储过程，一个用于更新现有的 Region 记录，另一个用于删除指定的 Region 记录。

##### (1) 记录的更新

更新 Region 记录是很简单的，因为(假定主键码不能更新)只有一个列可以更新。直接在 SQL Server 查询分析器中键入这些示例，或者运行本章下载代码中的 StoredProcs.sql 文件，在该文件中包含本节的所有存储过程：

```
CREATE PROCEDURE RegionUpdate (@RegionID INTEGER,
                               @RegionDescription NCHAR(50)) AS
SET NOCOUNT OFF
UPDATE Region
SET RegionDescription = @RegionDescription
WHERE RegionID = @RegionID
GO
```

给真实世界中的表执行更新命令，需要重复选择和返回已更新的记录。这个存储过程有两个输入参数(@RegionID 和 @RegionDescription)，对数据库执行 UPDATE 语句。

要在 .NET 代码中运行这个存储过程，需要定义一个 SQL 命令，并执行它：

```
SqlCommand aCommand = new SqlCommand("RegionUpdate", conn);

cmd.CommandType = CommandType.StoredProcedure;
cmd.Parameters.AddWithValue ( "@RegionID", 23 );
cmd.Parameters.AddWithValue ( "@RegionDescription", "Something" );
```

这段代码创建了一个新 SqlCommand 对象 cmd，并把它定义为一个存储过程。然后，使用 AddWithValue() 方法依次添加每个参数，这会构建一个参数，并设置其值，也可以手工构建 SqlParameter 实例，把它们添加到 Parameters 集合中。

该存储过程有两个参数：要更新的 Region 记录的唯一主键码；给这个记录的新描述。创建好命令后，就可以用下面的命令执行它：

```
cmd.ExecuteNonQuery();
```

由于该过程没有返回值，因此使用 ExecuteNonQuery 就足够了。命令参数可以直接使用 AddWithValue() 方法设置，也可以通过构建 SqlParameter 实例来设置。注意命令集合可以按位置或参数名来索引。

##### (2) 记录的删除

下一个存储过程可用于从数据库中删除一个 Region 记录：



```
CREATE PROCEDURE RegionDelete (@RegionID INTEGER) AS
SET NOCOUNT OFF
DELETE FROM Region
WHERE RegionID = @RegionID
GO
```

这个过程只需要该记录的主键码值。下面的代码使用 SqlCommand 对象调用这个存储过程：

```
SqlCommand aCommand = new SqlCommand("RegionDelete", conn);
aCommand.CommandType = CommandType.StoredProcedure;
aCommand.Parameters.Add(new SqlParameter("@RegionID", SqlDbType.Int, 0,
"RegionID"));
aCommand.UpdatedRowSource = UpdateRowSource.None;
```

这个命令只接收一个参数，如下面的代码所示，它执行 RegionDelete 存储过程，这是一个按照名称设置参数的示例。如果有许多对同一个存储过程的调用，就应构建 SqlParameter 实例，设置其值，如下面的示例所示，其性能要比为每个调用重新构建整个 SqlCommand 更好：

```
aCommand.Parameters["@RegionID"].Value = 999;
aCommand.ExecuteNonQuery();
```

## 2. 调用返回输出参数的存储过程

前面两个执行存储过程的示例都没有返回值。如果存储过程包含输出参数，则它们就需要在 .NET 客户程序中定义，以便在过程返回时填充其输出参数。下面的示例说明了如何在数据库中插入记录，把该记录的主键码返回给调用者。

### 记录的插入

Region 表仅由一个主键码(RegionID)和描述字段(RegionDescription)组成。要插入一个记录，需要生成该数字主键码，再把新行插入到数据库中。在这个示例中，通过在存储过程中创建一个主键码，简化了主键码的生成。使用的方法未经过任何加工，这就是本章的后面用一节的篇幅介绍键的生成的原因。现在使用这个示例就足够了：

```
CREATE PROCEDURE RegionInsert(@RegionDescription NCHAR(50),
@RegionID INTEGER OUTPUT) AS
SET NOCOUNT OFF
SELECT @RegionID = MAX(RegionID) + 1
FROM Region
INSERT INTO Region(RegionID, RegionDescription)
VALUES(@RegionID, @RegionDescription)
GO
```

插入过程创建一个新 Region 记录，在数据库本身生成主键码值时，这个值作为输出参数从过程中返回(@RegionID)。这对于这个简单示例来说就足够了，但对于比较复杂的表(特别是有默认值的表)，通常不使用输出参数，而选择整个插入的行，把该行返回给调用者。.NET 类可以处理这两种情况。

```
SqlCommand aCommand = new SqlCommand("RegionInsert", conn);
aCommand.CommandType = CommandType.StoredProcedure;
aCommand.Parameters.Add(new SqlParameter("@RegionDescription",
SqlDbType.NChar,
50,
"RegionDescription"));
```



```

aCommand.Parameters.Add(new SqlParameter("@RegionID" ,
                                         SqlDbType.Int,
                                         0 ,
                                         ParameterDirection.Output ,
                                         false ,
                                         0 ,
                                         0 ,
                                         "RegionID" ,
                                         DataRowVersion.Default ,
                                         null));
aCommand.UpdatedRowSource = UpdateRowSource.OutputParameters;

```

其中参数的定义比较复杂。第二个参数@RegionID 定义为包含其参数定向, 在这个示例中是 Output。除这个标志之外, 该示例还在最后一行使用 UpdateRowSource 枚举表示数据通过输出参数从这个存储过程中返回。当从一个 DataTable(详见本章后面的内容)中执行存储过程调用时, 主要使用这个标志。

调用这个存储过程类似于前面的示例, 但在这个实例中, 需要在执行完过程后读取输出参数:

```

aCommand.Parameters["@RegionDescription"].Value = "South West";
aCommand.ExecuteNonQuery();
int newRegionID = (int) aCommand.Parameters["@RegionID"].Value;

```

在执行完命令后, 读取@RegionID 参数的值, 并把它的数据类型转换为整型。上述命令的一个缩写版本是 ExecuteScalar()方法, 它返回存储过程返回的第一个值(返回为对象)。

如果调用的存储过程返回输出参数和一组记录行, 该怎么办? 在该实例中, 应定义合适的参数, 不是调用 ExecuteNonQuery(), 而应调用另一个方法(例如 ExecuteReader()), 遍历所有的返回记录。

## 26.4 快速数据访问: 数据读取器

数据读取器(data reader)是从数据源中选择某些数据的最简单快捷的方法, 但这也是功能最弱的一个方法。不能直接实例化数据阅读器, 即实例是调用 ExecuteReader()方法后从相应数据库的命令对象(如 SqlCommand)中返回的。

下面的代码说明了如何从 Northwind 数据库的 Customer 表中选择数据。这个示例连接了数据库, 选择许多记录, 循环所选的记录, 并把它们输出到控制台上。

这个示例使用 OLE DB 提供程序作为一个来自 SQL 提供程序的简化的数据暂存器。在大多数情况下, OleDbClient 类与 SqlClient 类是一对一的关系, 例如 OleDbConnection 对象就类似于在前面的示例中所使用的 SqlConnection 对象。

要对 OLE DB 数据源执行命令, 应使用 OleDbCommand 类。下面的代码执行一个简单的 SQL 语句, 读取记录, 返回一个 OleDbDataReader 对象。

注意下面的第二个 using 语句使 OleDb 类可用。

```

using System;
using System.Data.OleDb;

```

目前所利用的所有数据提供程序都在同一个程序集中, 所以只需要引用 System.Data.dll 程

序集, 就可以导入本节使用的所有类。唯一的例外是 Oracle 类, 它位于 System.Data. Oracle.dll 程序集中:

```
public class DataReaderExample
{
    public static void Main(string[] args)
    {
        string source = "Provider=SQLOLEDB;" +
            "server=(local);" +
            "integrated security=SSPI;" +
            "database=northwind";
        string select = "SELECT ContactName,CompanyName FROM Customers";
        OleDbConnection conn = new OleDbConnection(source);
        conn.Open();
        OleDbCommand cmd = new OleDbCommand(select, conn);
        OleDbDataReader aReader = cmd.ExecuteReader();
        while(aReader.Read())
            Console.WriteLine("{0}' from {1}",
                aReader.GetString(0), aReader.GetString(1));
        aReader.Close();
        conn.Close();
    }
}
```

前面的代码包含其他章节介绍的许多熟悉的 C# 功能。要编译该示例, 使用下面的命令:

```
csc /t:exe /debug+ DataReaderExample.cs /r:System.Data.dll
```

在前面的示例中, 下面的代码根据源连接字符串, 创建一个新 OLE DB.NET 数据库连接:

```
OleDbConnection conn = new OleDbConnection(source);
conn.Open();
OleDbCommand cmd = new OleDbCommand(select, conn);
```

第三行根据特定的 Select 语句创建一个新 OleDbCommand 对象, 以及执行命令时所使用的数据库连接。当有一个有效的命令时, 就需要执行它, 返回一个初始化了的 OleDbDataReader:

```
OleDbDataReader aReader = cmd.ExecuteReader();
```

OleDbDataReader 是一个只向前的连接游标, 即只能沿着一个方向遍历记录, 而使用的数据库连接一直打开, 直到关闭 DataReader 为止。

#### 提示:

OleDbDataReader 会使数据库连接一直处于打开状态, 直到显式关闭为止。

OleDbDataReader 类不能直接实例化, 它总是通过调用 OleDbCommand 类的 ExecuteReader 方法来返回。打开了一个数据读取器后, 就可以用各种方式访问包含在该读取器中的数据。

关闭 OleDbDataReader 对象(显式调用 Close() 或通过垃圾收集器收集对象)时, 底层的连接也会关闭。这取决于调用了哪个 ExecuteReader() 方法。如果调用了 ExecuteReader() 方法, 并传递了 CommandBehavior.CloseConnection, 就会在关闭读取器时关闭连接。

OleDbDataReader 类有一个索引器, 可以使用常见的数组语法访问任何字段(但不是类型安全的访问):

```
object o = aReader[0];
```

或者

```
object o = aReader["CategoryID"];
```

假定 CategoryID 字段是 SELECT 语句中用于填充阅读器的第一个字段,那么这两行语句的功能就是相同的,但后者比前者慢一些。为了验证这一点,编写一个简单的测试程序,从打开的数据读取器上对同一列进行一百万次的迭代访问,获取一些足够大的数字,虽然在一个循环中可能并不会对同一列访问一百万次,但按每秒来计算,就可能编写出最佳的代码。

另外,数字索引器平均每 0.09 秒就进行一百万次的访问,而文本索引器需要 0.63 秒。原因是文本方法是从模式的内部查找列号,再使用列号的顺序进行访问。如果知道这个区别,就可以更好地访问数据。

是否应使用数字索引器?也许,但还有一种更好的方式。

除了上面给出的索引器外, OleDbDataReader 还有一组类型安全的方法可以用于读取列,这些方法很容易理解,且都以 Get 开头。有一些方法可以读取大多数类型的数据,例如 GetInt32、GetFloat 和 GetGuid 等。

前面使用 GetInt32 的一百万次迭代用了 0.06 秒,数字索引器中的系统开销是获取数据类型,调用与 GetInt32 相同的代码,然后装箱(本实例是拆箱)为一个整数。如果以前知道这种模式,希望使用加密数字而不是列名,且允许对每个列访问使用类型安全的函数,这样运行速度就会比使用文本格式的列名快 10 倍(选择同一列的上百万个副本)。

毫无疑问,在可维护性和速度之间有一个折衷的问题。如果必须使用数字索引器,就应在类的范围内为每一个要访问的列定义常量。上面的代码可以用于从任何 OLE DB 数据库中选择数据,但有许多 SQL Server 的特定类可以使用,只是其便利性有明显的损失。

下面的示例与上一示例基本相同,但在这个实例中用 SQL 提供程序和 SQL 类的引用替换了 OLE DB 提供程序和对 OLE DB 类的所有引用。该示例在 04\_DataReaderSql 目录下:

```
using System;
using System.Data.SqlClient;
public class DataReaderSql
{
    public static int Main(string[] args)
    {
        string source = "server=(local);" +
            "integrated security=SSPI;" +
            "database=northwind";
        string select = "SELECT ContactName,CompanyName FROM Customers";
        SqlConnection conn = new SqlConnection(source);
        conn.Open();
        SqlCommand cmd = new SqlCommand(select, conn);
        SqlDataReader aReader = cmd.ExecuteReader();
        while(aReader.Read())
            Console.WriteLine("{0}' from {1}", aReader.GetString(0),
                aReader.GetString(1));
        aReader.Close();
        conn.Close();
        return 0;
    }
}
```

注意一下区别是什么？如果键入这些代码，用 sql 替换所有的 OleDb，改变数据源字符串，重新编译。这是很容易的。

对 SQL 提供程序的索引器进行相同的性能测试，这次数字索引器也使用 0.13 秒就完成了百万次的访问，基于索引器的字符串运行了 0.65 秒。

26.5 管理数据和关系：DataSet 类

DataSet 类是数据的脱机容器。它不包含数据库连接的概念，实际上存储在 DataSet 中的数据不一定来源于数据库，它可以是 CSV 文件中的记录，或是从测量设备中读取的点。

DataSet 类由一组数据表组成，每个表都有一些数据列和数据行，如图 26-4 所示。除了定义数据外，还可以在 DataSet 中定义表之间的链接。例如，我们常常要定义父/子关系(通常也称为主/从关系)。表中的一个记录(即 Order)链接到另一个表的许多记录上(即 Order\_Details)，这种关系可以在 DataSet 中定义和导航。

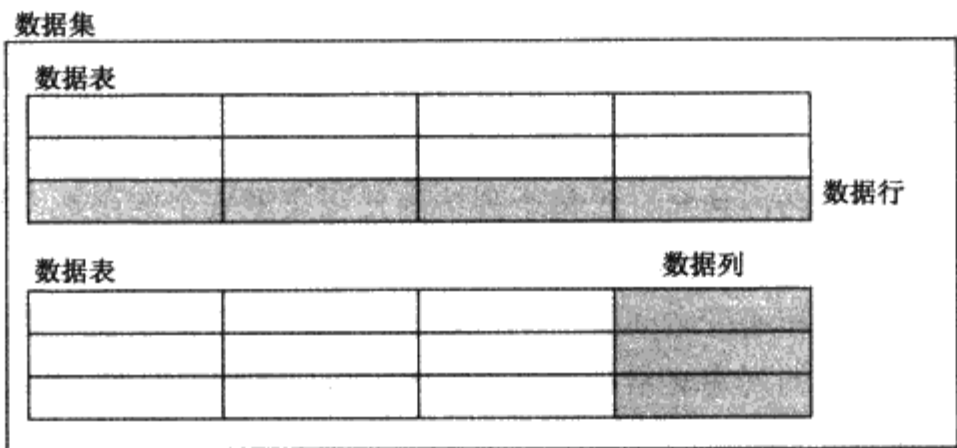


图 26-4

下一节描述和 DataSet 一起使用的类。

26.5.1 数据表

数据表非常类似于物理数据库表，它由一些带有特定属性的列组成，可能包含 0 行或多行数据。数据表也可以定义主键码(可以是一个列或多个列)，列上也可以包含约束。这些信息在本章的其他部分称为“模式”。

为数据表定义模式有几种方式(把 DataSet 类当作一个整体)，这些在介绍了数据列和数据行后讨论。图 26-5 显示了一些可通过数据表访问的对象。

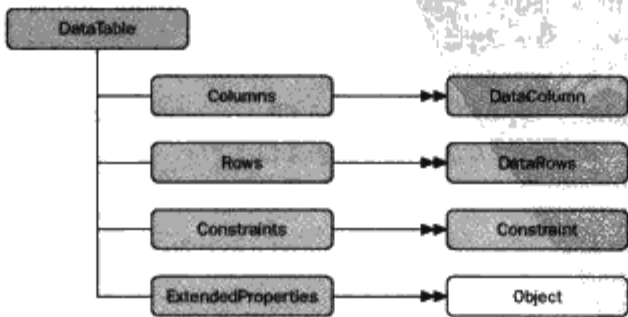


图 26-5



**DataTable** 对象(和  **DataColumn**)可以附带任意多个扩展属性。这个集合可以用附属于对象的用户自定义信息来填充。例如，某个列有一个输入掩码，用于验证列的内容是否有效，比较常见的示例是 US 社会安全号。当数据在中间层中构造，要返回给客户机，进行某些处理时，最适合使用扩展的属性。例如，可以在扩展的属性(如 **min** 和 **max**)中存储数字列的有效性标准，在验证用户输入时在 UI 层使用它们。

填充数据表时，可以从数据库中选择数据，从文件中读取数据，或在代码中手工填充，**Rows** 集合会包含这些检索出来的数据。

**Columns** 集合包含已经添加到表中的  **DataColumn** 实例，它们定义了数据的模式，例如数据类型、是否可为空和默认值等。**Constraints** 集合可以用唯一或主键码约束来填充。

数据表使用模式信息的一个示例是在 **DataGrid**(详见第 32 章)中显示数据。**DataGrid** 控件使用属性(例如列的数据类型)来确定该列应使用什么控件。数据库中的 **bit** 列在 **DataGrid** 中显示为一个复选框。如果列在数据库模式中定义为 **NOT NULL**，该信息就存储在  **DataColumn** 中，以便在用户试图移出数据行时测试该列。

26.5.2 数据列

**DataColumn** 对象定义了数据表中某列的属性，例如该列的数据类型，该列是否为只读，以及其他属性。列可以在代码中创建，或者由运行库自动生成。

在创建一个列时，给它指定名称是很有用的，否则运行库就会为该列生成一个名称，其格式是 **Columnn**，其中 **n** 是一个递增的数字。

列的数据类型可以在构造函数中提供，也可以通过设置 **DataType** 属性来指定。把数据加载到数据表中后，就不能改变列的数据类型了，否则会抛出 **ArgumentException** 异常。

创建的数据列可以包含表 26-7 所示的 .NET Framework 数据类型。

表 26-7

Boolean	Decimal	Int64	TimeSpan
Byte	Double	Sbyte	UInt16
Char	Int16	Single	UInt32
DateTime	Int32	String	UInt64

一旦创建好，就要给  **DataColumn** 对象设置其他属性，例如该列是否可为空或者设置默认值。下面的代码段显示了给  **DataColumn** 设置的一些常见选项：

```
 DataColumn customerID = new DataColumn("CustomerID", typeof(int));
 customerID.AllowDBNull = false;
 customerID.ReadOnly = false;
 customerID.AutoIncrement = true;
 customerID.AutoIncrementSeed = 1000;
 DataColumn name = new DataColumn("Name", typeof(string));
 name.AllowDBNull = false;
 name.Unique = true;
```



可以给 DataColumn 对象设置如表 26-8 所示的属性。

表 26-8

属 性	说 明
AllowDBNull	如果为 true，该列就可以设置为 DBNull
AutoIncrement	指定该列的值自动生成为一个递增的数字
AutoIncrementSeed	定义 AutoIncrement 列最初的种子值
AutoIncrementStep	用默认的步骤定义自动生成列值的递增量
Caption	可以用于在屏幕上显示列名
ColumnMapping	指定当 DataSet 通过调用 DataSet.WriteXml 来保存时，列如何映射到 XML 上
ColumnName	列名。如果没有在构造函数中设置，就由运行库自动生成
DataType	列的 System.Type 值
DefaultValue	可以定义列的默认值
Expression	定义在所计算的列中使用的表达式

1. 数据行

这个类构成了 DataTable 类的另一部分。数据表中的列根据 DataTable 类来定义，表中的实际数据用 DataRow 对象来访问。下面的示例说明了如何访问数据表中的行。首先是连接：

```
string source = "server=(local);" +
    "integrated security=SSPI;" +
    "database=northwind";
string select = "SELECT ContactName,CompanyName FROM Customers";
SqlConnection conn = new SqlConnection(source);
```

下面的代码显示了 SqlDataAdapter 类，它用于选择 DataSet 中的数据。SqlDataAdapter 使用 SQL 子句，在 DataSet 中用下面查询的结果填写表 Customers。SqlDataAdapter 类将在 26.7 节中进一步讨论。

```
SqlDataAdapter da = new SqlDataAdapter(select, conn);
DataSet ds = new DataSet();
da.Fill(ds, "Customers");
```

在下面的代码中，注意使用 DataRow 的索引器访问数据行上的值。给定列的值可以用几个重载的索引器来检索，这样就可以通过已知的列号、列名或 DataColumn 来检索数据的值：

```
foreach(DataRow row in ds.Tables["Customers"].Rows)
    Console.WriteLine("' {0}' from {1}", row[0], row[1]);
```

DataRow 最吸引人的一个方面就是它的版本功能。DataRow 可以接收某一行上指定列的各

个值，其版本见表 26-9。

表 26-9

DataRow 的 Version 值	说 明
Current	列中目前存在的值，如果没有进行编辑，该值与初值相同。如果进行了编辑，该值就是最后输入的一个有效值
Default	默认值(列的任何默认设置)
Original	最初从数据库中选择出来的列值。如果调用了 DataRow 的 AcceptChanges 方法，该值就更新为 Current 值
Proposed	对列进行修改时，可以检索到这个已改变的值。如果在行上调用了方法 BeginEdit()，并进行了修改，每一列都会有一个推荐值，直到调用了 EndEdit() 或 CancelEdit()为止

可以以许多方式使用给定列的版本。例如，在数据库中更新数据行时，常常使用如下 SQL 语句：

```
UPDATE Products
SET     Name = Column.Current
WHERE  ProductID = xxx
AND     Name = Column.Original;
```

显然，这段代码永远不会编译，但它说明了列的初值和当前值的一个用法。  
要从 DataRow 索引器中检索某个版本的值，应使用索引器方法，把 DataRowVersion 值作为一个参数。下面的代码段说明了如何获得 DataTable 中每一列的所有值：

```
foreach (DataRow row in ds.Tables["Customers"].Rows )
{
    foreach ( DataColumn dc in ds.Tables["Customers"].Columns )
    {
        Console.WriteLine ("{0} Current = {1}" , dc.ColumnName ,
                               row[dc,DataRowVersion.Current]);
        Console.WriteLine ("  Default = {0}" , row[dc,DataRowVersion.Default]);
        Console.WriteLine ("   Original = {0}" ,
                               row[dc,DataRowVersion.Original]);
    }
}
```

整个数据行有一个状态标志 RowState，可以用于确定在返回数据库时需要对该行进行什么操作。RowState 标志跟踪对 DataTable 所作的所有改变，例如添加新行、删除现有的行，改变表中的列。当数据与数据库同步时，行的状态标志用于确定应执行什么 SQL 操作。这些标志由 DataRowState 枚举定义，如表 26-10 所示。

表 26-10

DataRowState 值	说 明
Added	把新数据行添加到 DataTable 的 Rows 集合中。在客户机中创建的所有行都设置为这个值，最终在与数据库同步时，会使用 SQL INSERT 语句
Deleted	通过 DataRow.Delete() 方法把 DataTable 中的数据行标记为删除。该行仍存在 DataTable 中，但在屏幕上看不到它(除非显式设置 DataView)。DataView 在下一章讨论。在 DataTable 中标记为已删除的数据行将在与数据库同步时从数据库中删除
Detached	数据行在创建后立即显示为这个状态，调用 DataRow.Remove() 也可以返回这个状态。分立的行不是任何 DataTable 的一部分，因此处于这种状态的行不能使用任何 SQL 语句
Modified	如果列中的值发生了改变，数据行就处于这个状态
Unchanged	自从最后一次调用 AcceptChanges 以来，数据行都没有发生改变

行的状态也取决于在其上调用的方法。一般在成功更新数据源(即把改变返回数据库后)之后调用 AcceptChanges 方法。

修改 DataRow 中数据最常见的方式是使用索引器，但如果对数据进行了许多修改，就需要考虑使用 BeginEdit() 和 EndEdit() 方法。

在对 DataRow 中的列进行了修改后，就会在该行的 DataTable 上引发 ColumnChanging 事件。该事件可以重写 DataColumnChangeEventArgs 类的 ProposedValue 属性，按照需要修改它。这是在列值上进行某些数据有效性验证的一种方式。如果在进行修改前调用 BeginEdit() 方法，就不会引发 ColumnChanging 事件，于是可以进行多次修改，再调用 EndEdit() 方法，保存这些修改。如果要回退到初值，应调用 CancelEdit()。

DataRow 可以以某种方式链接到其他数据行上，在数据行之间能够建立可导航的链接，这在主/从数据表中非常常见。DataRow 包含一个 GetChildRows() 方法，该方法可以从同一个 DataSet 的另一个表中把一组相关行返回为当前行。这些将在本章后面的“数据关系”一节中介绍。

2. 模式的生成

为 DataTable 创建模式有 3 种方式：

- 让运行库来完成
- 编写代码来创建表
- 使用 XML 模式生成器

(1) 运行库生成的模式

前面的 DataRow 示例用下面的代码从数据库中选择数据，并生成一个 DataSet:

```
SqlDataAdapter da = new SqlDataAdapter(select , conn);
DataSet ds = new DataSet();
da.Fill(ds , "Customers");
```

这是很容易使用的，但也有几个缺点。例如，必须利用默认的列名来处理——这是可以的，

但在某些情况下，还要把物理数据库的列(如 PKID)重新命名为一个用户友好性更高的名称。

自然，可以在 SQL 子句中给列指定别名，例如 `SELECT PID AS PersonID FROM Person Table`。最好不要在 SQL 中重新给列命名，因为列实际上只需要在屏幕上显示一个“比较好”的名称即可。

自动生成 `DataTable/DataColumn` 的另一个潜在问题是不能控制运行库为列选择的数据类型。运行库可以确定正确的数据类型，但有时需要对此有更多的控制。例如，为给定的列定义枚举类型，以简化类的用户代码。如果接受运行库生成的默认列类型，该列就可能是一个 32 位的整数，而不是有预定选项的枚举。

最后，也是最有可能出的问题是，在使用自动生成的表时，不能对 `DataTable` 中的数据进行类型安全的访问——索引器就会返回 `object` 的实例，而不是派生的数据类型。如果要用代码对表达式进行类型转换，就可以跳过下面的章节。

## (2) 手工编码的模式

用生成的代码来创建 `DataTable`，再用相关的 `DataColumns` 来填充是相当简单的。本节的示例将访问 Northwind 数据库中的 Product 表，如图 26-6 所示。

Column Name	Data Type	Allow Nulls
ProductID	int	<input type="checkbox"/>
ProductName	nvarchar(40)	<input type="checkbox"/>
SupplierID	int	<input checked="" type="checkbox"/>
CategoryID	int	<input checked="" type="checkbox"/>
QuantityPerUnit	nvarchar(20)	<input checked="" type="checkbox"/>
UnitPrice	money	<input checked="" type="checkbox"/>
UnitsInStock	smallint	<input checked="" type="checkbox"/>
UnitsOnOrder	smallint	<input checked="" type="checkbox"/>
ReorderLevel	smallint	<input checked="" type="checkbox"/>
Discontinued	bit	<input type="checkbox"/>

图 26-6

下面的代码生成了一个 `DataTable`，对应于上面的模式(但没有包含可为空的列)：

```
public static void ManufactureProductDataTable(DataSet ds)
{
    DataTable products = new DataTable("Products");
    products.Columns.Add(new DataColumn("ProductID", typeof(int)));
    products.Columns.Add(new DataColumn("ProductName", typeof(string)));
    products.Columns.Add(new DataColumn("SupplierID", typeof(int)));
    products.Columns.Add(new DataColumn("CategoryID", typeof(int)));
    products.Columns.Add(new DataColumn("QuantityPerUnit", typeof(string)));
    products.Columns.Add(new DataColumn("UnitPrice", typeof(decimal)));
    products.Columns.Add(new DataColumn("UnitsInStock", typeof(short)));
    products.Columns.Add(new DataColumn("UnitsOnOrder", typeof(short)));
    products.Columns.Add(new DataColumn("ReorderLevel", typeof(short)));
    products.Columns.Add(new DataColumn("Discontinued", typeof(bool)));
    ds.Tables.Add(products);
}
```

可以改变 `DataRow` 示例中的代码，使用新生成的表定义：

```
string source = "server=(local);" +
    "integrated security=sspi;" +
    "database=Northwind";
```

```

string select = "SELECT * FROM Products";
SqlConnection conn = new SqlConnection(source);
SqlDataAdapter cmd = new SqlDataAdapter(select, conn);
DataSet ds = new DataSet();
ManufactureProductDataTable(ds);
cmd.Fill(ds, "Products");
foreach(DataRow row in ds.Tables["Products"].Rows)
    Console.WriteLine("'{0}' from {1}", row[0], row[1]);

```

**ManufactureProductDataTable()**方法创建一个新 **DataTable**，依次添加每个列，最后把这个表添加到 **DataSet** 中表的清单上。**DataSet** 有一个索引器，它的参数是表名，给调用者返回该 **DataTable**。

上面的示例仍不是类型安全的，因为在列上使用了索引器来检索数据。最好是有一个类(或一组类)派生自 **DataSet**、**DataTable** 和 **DataRow**，为表、行和列定义类型安全的存取器。可以自己生成这段代码，这并不是特别乏味，最终将得到可以进行类型安全访问的类。

如果不愿意自己生成这些类型安全的类，可以使用帮助。**.NET Framework** 允许使用 XML 模式来定义 **DataSet**、**DataTables** 和本节介绍的其他类。本章后面的 XML 模式一节详细介绍了这个方法。

### 26.5.3 数据关系

在编写应用程序时，常常需要获取和缓存各种信息表。**DataSet** 类是这些信息的容器，使用一般的 OLE DB，需要提供一种奇怪的 SQL 语法来执行分层的数据关系，提供程序本身不能没有它自己的“怪癖”。

另一方面，**DataSet** 类从一开始就是为建立数据表之间的关系而设计的。本节的代码说明了如何手工生成并填充两个数据表。如果手中有 SQL Server 或 NorthWind 数据库，就可以运行这个示例。

```

DataSet ds = new DataSet("Relationships");
ds.Tables.Add(CreateBuildingTable());
ds.Tables.Add(CreateRoomTable());
ds.Relations.Add("Rooms",
    ds.Tables["Building"].Columns["BuildingID"],
    ds.Tables["Room"].Columns["BuildingID"]);

```

这两个表仅包含一个主键和名称字段，Room 表有一个 BuildingID 外键，如图 26-7 所示。



图 26-7

这些表都非常简单，下面的代码迭代 **Buildings** 表中的行，并遍历数据关系，列出 **Rooms** 表中所有的子行。

```

foreach(DataRow theBuilding in ds.Tables["Building"].Rows)
{

```



```
DataRow[] children = theBuilding.GetChildRows("Rooms");
int roomCount = children.Length;
Console.WriteLine("Building {0} contains {1} room{2}",
    theBuilding["Name"],
    roomCount,
    roomCount > 1 ? "s" : "");
// Loop through the rooms
foreach(DataRow theRoom in children)
    Console.WriteLine("Room: {0}", theRoom["Name"]);
}
```

DataSet 类和其他分层的旧 recordset 对象之间的主要区别是关系显示的方式。在分层的 recordset 对象中，关系显示为行中的一个伪列，这个列本身是一个可以迭代的 recordset 对象。但在 ADO.NET 中，通过调用 GetChildRows()方法就可以遍历关系。

```
DataRow[] children = theBuilding.GetChildRows("Rooms");
```

该方法有许多形式，但上面的示例只使用关系的名称在父子行之间来回遍历。它返回一组数据行，使用前面示例中的索引器就可以更新这些行。

更有趣的数据关系是可以用两种方式遍历这些数据。在 DataTable 类上使用 ParentRelations 属性，可以从父数据行中找到子数据行，也可以在子记录中找到父数据行。这个属性返回一个 DataRelationCollection，该集合可以使用[]数组语法来索引(例如，DataRelations["Rooms"]), 另外，GetChildRows()方法也可以如下所示进行调用；

```
foreach(DataRow theRoom in ds.Tables["Room"].Rows)
{
    DataRow[] parents = theRoom.GetParentRows("Rooms");
    foreach(DataRow theBuilding in parents)
        Console.WriteLine("Room {0} is contained in building {1}",
            theRoom["Name"],
            theBuilding["Name"]);
}
```

方法 GetParentRows(返回 0 行或多行数据)或 GetParentRow(根据给定的某种关系检索一个父行)都有许多重写版本，可以检索出父行。

26.5.4 数据约束

DataTable 擅长于改变在客户机上创建的列的数据类型。ADO.NET 允许在列上创建一些约束，对数据应用一些规则。

运行库目前支持表 26-11 所示的约束类型，它们包含在 System.Data 命名空间的类中。

表 26-11

约 束	说 明
ForeignKeyConstraint	在 DataSet 的两个 DataTable 之间建立链接
UniqueConstraint	确保给定的列是唯一的

## 1. 设置主键码

在关系数据库的表中，可以提供一个主键码，该主键码可以基于 `DataTable` 中的一个或多个列。

下面的代码为 `Product` 表创建了一个主键码，其模式是前面手工构建的。

表上的主键码只是约束的一种形式。当主键码添加到 `DataTable` 中时，运行库也会对键码列生成一个唯一的约束。这是因为并没有 `PrimaryKey` 约束类型，主键码是一个或多个列上的唯一约束。

```
public static void ManufacturePrimaryKey(DataTable dt)
{
    DataColumn[] pk = new DataColumn[1];
    pk[0] = dt.Columns["ProductID"];
    dt.PrimaryKey = pk;
}
```

因为主键码可以包含几个列，所以它可以作为一组 `DataColumn` 键入。通过给表的一组列指定属性，就可以给这些列设置主键码。

要检查表中的约束，可以迭代 `ConstraintsCollection`。上述代码自动生成的约束是 `Constraint1`，这个名称没有什么用，应避免这种情况，最好先在代码中创建约束，然后定义组成主键码的列。

创建主键码之前，下面的代码给约束命名：

```
DataColumn[] pk = new DataColumn[1];
pk[0] = dt.Columns["ProductID"];
dt.Constraints.Add(new UniqueConstraint("PK_Products", pk[0]));
dt.PrimaryKey = pk;
```

唯一约束可以应用到任意多个列上。

## 2. 设置外键码

除了唯一约束外，`DataTable` 类还可以包含外键码约束，它们主要用于建立主/从关系，如果正确建立了约束，`Foreign Key` 约束还可以在表之间复制列。在主/从关系的表中，常常有一个父记录(`Order`)和许多子记录(`Order Line`)，它们是通过父记录的主键码链接起来的。

外键码约束只能用于同一个 `DataSet` 中的表，所以下面的示例使用 `Northwind` 数据库中的 `Categories` 表，给该表和 `Products` 表之间指定约束，如图 26-8 所示。

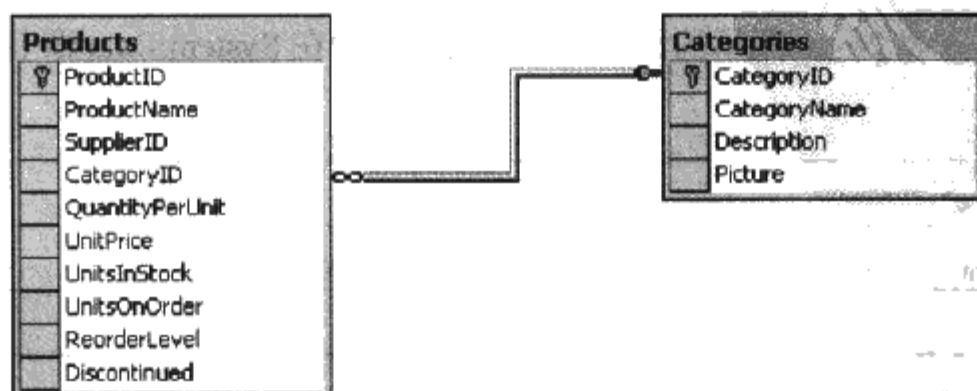


图 26-8

第一步是为 Categories 表生成一个新的数据表:

```
DataTable categories = new DataTable("Categories");
categories.Columns.Add(new DataColumn("CategoryID", typeof(int)));
categories.Columns.Add(new DataColumn("CategoryName", typeof(string)));
categories.Columns.Add(new DataColumn("Description", typeof(string)));
categories.Constraints.Add(new UniqueConstraint("PK_Categories",
    categories.Columns["CategoryID"]));
categories.PrimaryKey = new DataColumn[1]
    {categories.Columns["CategoryID"]};
```

上述代码的最后一行为 Categories 表创建了主键码。在本例中, 主键码是一个单列, 但可以使用数组语法在多个列上生成一个键码。

然后, 需要在两个表之间创建约束:

```
DataColumn parent = ds.Tables["Categories"].Columns["CategoryID"];
DataColumn child = ds.Tables["Products"].Columns["CategoryID"];
ForeignKeyConstraint fk =
    new ForeignKeyConstraint("FK_Product_CategoryID", parent, child);
fk.UpdateRule = Rule.Cascade;
fk.DeleteRule = Rule.SetNull;
ds.Tables["Products"].Constraints.Add(fk);
```

这个约束应用到 Categories.CategoryID 和 Products.CategoryID 之间的链接上。有 4 个不同的构造函数用于 ForeignKeyConstraint 类, 但应使用可以给约束命名的构造函数。

### 3. 设置更新和删除约束

除了在父表和子表之间定义约束之外, 还可以在更新约束中的一个列时定义应执行的操作。

上面的示例设置了更新规则和删除规则, 在对父表中的列(或行)执行某种操作时, 使用这些规则, 以确定应对子表中的行进行什么操作。通过 Rule 枚举可以应用 4 种不同的规则:

- Cascade——如果更新了父键, 就应把新的键值复制到所有的子记录上。如果删除了父记录, 也将删除子记录, 这是默认选项。
- None——不执行任何操作, 这个选项会留下子数据表中的孤立行。
- SetDefault——如果定义了一个子记录, 那么每个受影响的子记录都把外键码列设置为其默认值。
- SetNull——所有的子行都把主列设置为 DBNull。(按照 Microsoft 使用的命名约定, 主列应是 SetDBNull。)

提示:

如果 EnforceConstraints 属性为 true, 约束就只能在 DataSet 中使用。

前面介绍了组成 DataSet 中约束部分的主类, 解释了如何在代码中手工生成这些类。还可以使用 .NET 附带的 XML 模式文件和 XSD 工具定义 DataTable、DataRow、DataColumn、DataRelation 和 Constraint。下一节将说明如何建立一个简单的模式, 以及如何生成类型安全的类, 以访问数据。

## 26.6 XML 模式：用 XSD 生成代码

XML 已经在 ADO.NET 确立了牢固的地位——实际上，现在在对象之间远程传递数据的格式是 XML。有了 .NET 运行库，就可以在 XML 模式定义文件(XSD)中描述 DataTable 了。而且，可以定义整个 DataSet，其中有许多 DataTable，这些表之间有一定的关系，并包括描述数据的其他信息。

在定义 XSD 文件时，运行库中有一个新工具，该工具可以把这个模式转换为对应的数据访问类，例如类型安全的 DataTable 类，如上所述。本节先介绍一个简单的 XSD 文件(Products.xsd)，该文件描述与前面 Products 示例相同的信息，再扩展它，使其包括一些额外的功能。

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema id="Products" targetNamespace="http://tempuri.org/XMLSchema1.xsd"
  xmlns:mstns="http://tempuri.org/XMLSchema1.xsd"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
  <xs:element name="Product">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="ProductID" msdata:ReadOnly="true"
          msdata:AutoIncrement="true" type="xs:int" />
        <xs:element name="ProductName" type="xs:string" />
        <xs:element name="SupplierID" type="xs:int" minOccurs="0" />
        <xs:element name="CategoryID" type="xs:int" minOccurs="0" />
        <xs:element name="QuantityPerUnit" type="xs:string" minOccurs="0" />
        <xs:element name="UnitPrice" type="xs:decimal" minOccurs="0" />
        <xs:element name="UnitsInStock" type="xs:short" minOccurs="0" />
        <xs:element name="UnitsOnOrder" type="xs:short" minOccurs="0" />
        <xs:element name="ReorderLevel" type="xs:short" minOccurs="0" />
        <xs:element name="Discontinued" type="xs:boolean" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

第 26 章将详细论述其中的一些选项。现在应知道，这个文件定义一个模式，其中把 id 属性设置为 Products。还定义了一个比较复杂的类 Product，其中包含了许多元素，每个元素对应于 Products 表中的一个字段。

这些元素映射到数据类上。Products 模式映射到派生于 DataSet 的一个类上，类 Product 映射到派生于 DataTable 的一个类上。每个子元素映射到派生于 DataColumn 的一个类上。所有列的集合都映射到派生于 DataRow 的一个类上。

.NET Framework 中有一个工具，只要输入 XSD 文件，该工具就可以生成这些类的代码。因为该工具唯一的工作是执行 XSD 文件上的各种功能，所以它称为 XSD.EXE。

### 用 XSD 生成代码

假定把上面的文件保存为 Product.xsd，在命令提示符上输入下述命令，把该文件转换为代码：



```
xsd Product.xsd /d
```

这会创建文件 Product.cs。

该命令有许多选项可以和 XSD 一起使用，以改变生成的输出结果，其中一些比较常用的选项如表 26-12 所示。

表 26-12

选 项	说 明
/dataset (/d)	生成派生自 DataSet、DataTable 和 DataRow 的类
/language:<language>	允许选择编写输出文件的语言。C#是默认语言,也可以选择用 VB 编写 Visual Basic.NET 文件
/namespace:<namespace>	定义存储生成代码的命名空间,默认为没有命名空间

下面节选了 Products 模式在 XSD 中的输出结果，并做了略微的修改，重新进行了格式化，以便可以放在本书中。要查看完整的输出结果，可以在 Products 模式(或者自己建立的某个模式)上运行 XSD.EXE，查看生成的.cs 文件。该示例包含了所有的源代码和 Products.xsd 文件，这个输出结果是 [www.wroc.com](http://www.wroc.com) 上下载代码文件的一部分。

```
//-----
// <autogenerated>
//   This code was generated by a tool.
//   Runtime Version:2.0.50727.312
//
//   Changes to this file may cause incorrect behavior and will be lost if
//   the code is regenerated.
// </autogenerated>
//-----

using System;

//
// This source code was auto-generated by xsd, Version=2.0.40426.16.
//

[Serializable()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
[System.Diagnostics.DebuggerStepThrough()]
[System.ComponentModel.ToolboxItem(true)]
[System.Xml.Serialization.XmlSchemaProviderAttribute("GetTypedDataSetSchema")]
[System.Xml.Serialization.XmlRootAttribute("Products")]
public partial class Products : System.Data.DataSet {
    {
        private ProductDataTable tableProduct;
        public Products()
        public ProductDataTable Product
        public override DataSet Clone()
        public delegate void ProductRowChangeEventHandler ( object sender,
                                                            ProductRowChangeEvent e);

        [System.Diagnostics.DebuggerStepThrough()]
        public partial class ProductDataTable : DataTable, IEnumerable
```



```
[System.Diagnostics.DebuggerStepThrough()]
public class ProductRow : DataRow
{
}
```

为了集中论述公共接口，这段代码删除了所有受保护和私有的成员。**ProductDataTable** 和 **ProductRow** 定义是两个嵌套的类，后面会介绍它们。下面简单地解释一下 **DataSet** 派生类，之后讨论这些类的代码。

**Products()**构造函数调用一个私有方法 **InitClass()**，构造派生自 **DataTable** 类的 **ProductDataTable** 的一个实例，把该表添加到 **DataSet** 的 **Tables** 集合上。**Products** 数据表可以通过下面的代码来访问：

```
DataSet ds = new Products();
DataTable products = ds.Tables["Products"];
```

更简单的方式是使用 **Product** 属性来访问，该属性在派生的 **DataSet** 对象上：

```
DataTable products = ds.Product;
```

因为 **Product** 属性是强类型化的，所以可以使用 **ProductDataTable**，而不是上面的 **DataTable** 引用。

**ProductDataTable** 类包含更多的代码：

```
[System.Serializable()]
[System.Diagnostics.DebuggerStepThrough()]
[System.Xml.Serialization.XmlSchemaProviderAttribute("GetTypedTableSchema")]
public partial class ProductDataTable : DataTable, System.Collections.IEnumerable
{
    private DataColumn columnProductID;
    private DataColumn columnProductName;
    private DataColumn columnSupplierID;
    private DataColumn columnCategoryID;
    private DataColumn columnQuantityPerUnit;
    private DataColumn columnUnitPrice;
    private DataColumn columnUnitsInStock;
    private DataColumn columnUnitsOnOrder;
    private DataColumn columnReorderLevel;
    private DataColumn columnDiscontinued;

    public ProductDataTable() {
        this.TableName = "Product";
        this.BeginInit();
        this.InitClass();
        this.EndInit();
    }
}
```

**ProductDataTable** 类派生于 **DataTable**，它实现 **IEnumerable** 接口，为表中的每一列定义了一个私有的 **DataColumn** 实例，通过调用私有的 **InitClass** 成员，在构造函数中初始化这些实例。每一列都有一个内部的访问器，**DataRow** 类将使用这个访问器(后面介绍)。

```
[System.ComponentModel.Browsable(false)]
public int Count
{
    get { return this.Rows.Count; }
}
internal DataColumn ProductIDColumn
{
}
```

```

    get { return this.columnProductID; }
}
// Other row accessors removed for clarity - there is one for each of the columns

```

给表添加数据行是由 `AddProductRow()` 的两个重载方法实现的(但它们的内容完全不同, 但名称相同)。第一个重载方法的参数是一个已经构造出来的 `DataRow`, 且没有返回值。另一个重载方法则带一组参数值, 每个参数对应于 `DataTable` 中的一列, 该重载方法构造一个新行, 设置这个新行中的值, 把该行添加到 `DataTable`, 并给调用者返回该行。这些不同的函数不应有相同的名称。

```

public void AddProductRow(ProductRow row)
{
    this.Rows.Add(row);
}

public ProductRow AddProductRow ( string ProductName , int SupplierID ,
                                   int CategoryID , string QuantityPerUnit ,
                                   System.Decimal UnitPrice , short UnitsInStock ,
                                   short UnitsOnOrder , short ReorderLevel ,
                                   bool Discontinued )
{
    ProductRow rowProductRow = ((ProductRow)(this.NewRow()));
    rowProductRow.ItemArray = new object[]
    {
        null,
        ProductName,
        SupplierID,
        CategoryID,
        QuantityPerUnit,
        UnitPrice,
        UnitsInStock,
        UnitsOnOrder,
        ReorderLevel,
        Discontinued
    };
    this.Rows.Add(rowProductRow);
    return rowProductRow;
}

```

`DataSet` 派生类中的 `InitClass` 成员把表添加到 `DataSet`, 与此相同, `ProductDataTable` 上的 `InitClass` 成员把列添加到 `DataTable` 中, 每一列的属性都按需要设置, 再把该列添加到列集合的尾部。

```

private void InitClass()
{
    this.columnProductID = new DataColumn ( "ProductID",
                                             typeof(int),
                                             null,
                                             System.Data.MappingType.Element);
    this.columnProductID.ExtendedProperties.Add
        ("Generator_ChangedEventName", "ProductIDChanged");
    this.columnProductID.ExtendedProperties.Add
        ("Generator_ChangingEventName", "ProductIDChanging");
    this.columnProductID.ExtendedProperties.Add
        ("Generator_ColumnPropNameInRow", "ProductID");
    this.columnProductID.ExtendedProperties.Add
        ("Generator_ColumnPropNameInTable", "ProductIDColumn");
}

```

```

        this.columnProductID.ExtendedProperties.Add
            ("Generator_ColumnVarNameInTable", "columnProductID");
        this.columnProductID.ExtendedProperties.Add
            ("Generator_DelegateName", "ProductIDChangeEventHandler");
        this.columnProductID.ExtendedProperties.Add
            ("Generator_EventArgName", "ProductIDChangeEventArgs");
        this.Columns.Add(this.columnProductID);
        // Other columns removed for clarity

        this.columnProductID.AutoIncrement = true;
        this.columnProductID.AllowDBNull = false;
        this.columnProductID.ReadOnly = true;
        this.columnProductName.AllowDBNull = false;
        this.columnDiscontinued.AllowDBNull = false;
    }

    public ProductRow NewProductRow()
    {
        return ((ProductRow)(this.NewRow()));
    }

```

`NewRowFromBuilder` 在 `DataTable` 的 `NewRow()` 方法内部调用，创建了一个强类型化的新行。`DataRowBuilder` 实例由 `DataTable` 创建，其成员仅能在 `System.Data` 程序集中访问。

```

protected override DataRow NewRowFromBuilder(DataRowBuilder builder)
{
    return new ProductRow(builder);
}

```

最后一个要讨论的类是 `ProductRow`，它派生自 `DataRow`。这个类用于提供对数据表中所有字段的类型安全的访问。它封装了特定行的存储器，并提供成员来读取(和写入)表中的每个字段。

另外，对于每个可为空的字段，还有函数可以把该字段设置为 `null`，并检查该字段是否为 `null`。下面的示例列出了 `SupplierID` 列的函数：

```

[System.Diagnostics.DebuggerStepThrough()]
public class ProductRow : DataRow
{
    private ProductDataTable tableProduct;

    internal ProductRow(DataRowBuilder rb) : base(rb)
    {
        this.tableProduct = ((ProductDataTable)(this.Table));
    }

    public int ProductID
    {
        get { return ((int)(this[this.tableProduct.ProductIDColumn])); }
        set { this[this.tableProduct.ProductIDColumn] = value; }
    }
    // Other column accessors/mutators removed for clarity

    public bool IsSupplierIDNull()
    {
        return this.IsNull(this.tableProduct.SupplierIDColumn);
    }
}

```



```

public void SetSupplierIDNull()
{
    this[this.tableProduct.SupplierIDColumn] = System.Convert.DBNull;
}
}

```

下面的代码利用 XSD 工具中的类的输出从 Product 表中检索数据，并在控制台上显示这些数据：

```

using System;
using System.Data;
using System.Data.SqlClient;

public class XSD_DataSet
{
    public static void Main()
    {
        string source = "server=(local);" +
            " integrated security=SSPI;" +
            "database=northwind";
        string select = "SELECT * FROM Products";
        SqlConnection conn = new SqlConnection(source);
        SqlDataAdapter da = new SqlDataAdapter(select, conn);
        Products ds = new Products();
        da.Fill(ds, "Product");
        foreach(Products.ProductRow row in ds.Product )
            Console.WriteLine("'{'0}' from {1}" ,
                row.ProductID ,
                row.ProductName);
    }
}

```

重要的代码已突出显示出来了。XSD 文件的输出结果包含一个派生自 DataSet 的类 Products，它使用数据适配器来创建和填充。foreach 语句使用了强类型化的 ProductRow 和 Product 属性，返回 Product 数据表。

要编译这个示例，执行下面的命令：

```
xsd product.xsd /d
```

和

```
csc /recurse:*.cs
```

第一个命令从 Products.XSD 模式中生成 Products.cs 文件，然后，csc 命令使用/recurse:\*.cs 参数查找扩展名为.cs 的文件，并把它们添加到所生成的程序集中。

## 26.7 填充数据集

定义了数据集的模式，并准备好 DataTables、DataColumns、Constrain 类和一些必需内容后，就需要用这些信息填充 DataSet 了。从外部源中读取数据，并插入到 DataSet 中，有两种方式：

- 使用数据适配器
- 把 XML 读入数据集

## 26.7.1 用数据适配器来填充 DataSet

前面讨论数据行的一节简要介绍了 SqlDataAdapter 类，使用该类的代码如下所示：

```
string select = "SELECT ContactName,CompanyName FROM Customers";
SqlConnection conn = new SqlConnection(source);
SqlDataAdapter da = new SqlDataAdapter(select, conn);
DataSet ds = new DataSet();
da.Fill(ds, "Customers");
```

突出显示的两行代码显示了 SqlDataAdapter 类——实际上，其他数据适配器在功能上与 SqlDataAdapter 是完全相同的。

为了把数据插入到 DataSet 中，需要执行某种形式的命令以选择该数据。该命令可以是 SQL SELECT 语句，一个存储过程的调用，或者是 TableDirect 命令(用于 OleDb 提供程序)。上面的示例使用了 SqlDataAdapter 的一个构造函数，把传送过来的 SQL SELECT 子句转换为一个 SqlCommand，在适配器上调用 Fill()方法时执行这个命令。

在本章前面的存储过程示例中，定义了 INSERT、UPDATE 和 DELETE 命令，但没有给出 SELECT 过程，本节介绍该过程，并说明如何从 SqlDataAdapter 上调用存储过程，把数据填充到 DataSet 中。

## 在数据适配器上使用存储过程

首先需要定义一个存储过程，SELECT 存储过程如下所示：

```
CREATE PROCEDURE RegionSelect AS
SET NOCOUNT OFF
SELECT * FROM Region
GO
```

这个存储过程可以直接输入到 SQL Server 查询分析器中，或者可以运行这个示例所使用的 StoredProc.sql 文件。

接着，需要定义一个执行该存储过程的 SqlCommand，这段代码非常简单，并且大部分已经在前面“执行命令”的一节中介绍过了：

```
private static SqlCommand GenerateSelectCommand(SqlConnection conn)
{
    SqlCommand aCommand = new SqlCommand("RegionSelect", conn);
    aCommand.CommandType = CommandType.StoredProcedure;
    aCommand.UpdatedRowSource = UpdateRowSource.None;
    return aCommand;
}
```

这个方法生成了一个 SqlCommand，该 SqlCommand 在执行时会调用 RegionSelect 过程。最后是把这个命令和 SqlDataAdapter 类关联起来，调用 Fill()方法：

```
DataSet ds = new DataSet();
// Create a data adapter to fill the DataSet
SqlDataAdapter da = new SqlDataAdapter();
// Set the data adapter's select command
da.SelectCommand = GenerateSelectCommand(conn);
da.Fill(ds, "Region");
```



其中创建了一个新 `SqlDataAdapter`，把生成的 `SqlCommand` 赋给数据适配器的 `SelectCommand` 属性，然后调用执行存储过程的 `Fill()` 方法，把返回的所有行插入到 `Region` 数据表中(在本例中，它是由运行库生成的)。

数据适配器不仅仅能通过执行命令来选择数据，“保存对数据集的修改”一节会介绍数据适配器的其他功能。

### 26.7.2 从 XML 中给数据集填充数据

除了为给定的 `DataSet` 和相关表生成模式外，`DataSet` 还可以读写本机 XML 中的数据，例如磁盘上的文件，数据流或文本读取器。

要把 XML 加载到 `DataSet` 中，只需调用一个 `ReadXML()` 方法，例如下面的代码将从磁盘文件中读取数据：

```
DataSet ds = new DataSet();
ds.ReadXml(".\\MyData.xml");
```

`ReadXml` 方法试图从输入的 XML 中加载任何内联模式信息，如果加载了某个模式，就使用这个模式验证从该文件中加载的数据的有效性。如果没有使用内联模式，`DataSet` 就会在加载数据时扩展其内部的结构，这类似于前面示例中的 `Fill` 方法的作用，该方法检索数据，并根据选择出来的数据构造 `DataTable`。

## 26.8 保存对数据集的修改

在 `DataSet` 中编辑完数据后，就需要保留这些改变。最常见的示例是从数据库中选择数据，并把它显示给用户，把这些更新返回给数据库。

在没有“连接数据库的”应用程序中，这些改变可以保存在 XML 文件中，并传送到中间层的应用程序服务器上，最后进行处理，更新几个数据源。

`DataSet` 可以用于这两个示例，更重要的是这很容易完成。

### 26.8.1 通过数据适配器进行更新

除了 `SqlDataAdapter` 最有可能包含的 `SelectCommand` 之外，还可以定义 `InsertCommand`、`UpdateCommand` 和 `DeleteCommand`。顾名思义，这些对象都是适用于相应提供程序的命令对象(例如 `SqlCommand` 或 `OleDbCommand`)实例。

有了这种灵活性后，就可以自由调整应用程序，对频繁使用的命令(例如 `select` 和 `insert`)采用合适的存储过程来执行，对不常使用的命令(例如 `delete`)采用 SQL 命令的方式执行。一般应为所有的数据库交互操作提供存储过程，因为这会更快速，更容易调整。

本节的示例使用“调用存储过程”一节中的存储过程，插入、更新和删除 `Region` 记录，再把这些与上面编写的 `RegionSelect` 过程结合起来，生成一个新示例，这个示例使用这些命令来检索和更新 `DataSet` 中的代码。代码的主体如下所示。

## 1. 插入一个新行

把新行添加到 `DataTable` 中有两种方式。第一种方式是调用 `NewRow()` 函数, 返回一个空行, 然后向其填充数据, 最后把它添加到 `Rows` 集合中:

```
DataRow r = ds.Tables["Region"].NewRow();
r["RegionID"] = 999;
r["RegionDescription"] = "North West";
ds.Tables["Region"].Rows.Add(r);
```

第二种方式是把一组数据传送给 `Rows.Add()` 方法:

```
DataRow r = ds.Tables["Region"].Rows.Add
(new object [] { 999, "North West" });
```

`DataTable` 中的每个新行都把自己的 `RowState` 设置为 `Added`, 在对数据库进行修改前, 这个示例先把记录清空, 然后把下面的行添加到 `DataTable` 中(以任何一种方式)。注意右边一列显示行的状态:

New row pending inserting into database	
1 Eastern	Unchanged
2 Western	Unchanged
3 Northern	Unchanged
4 Southern	Unchanged
999 North West	Added

要用 `DataAdapter` 更新数据库, 调用 `Update` 方法:

```
da.Update(ds, "Region");
```

对于 `DataTable` 中的每一新行, 这将执行存储过程(在本例中是 `RegionInsert`), 然后再次把 `DataTable` 中的记录清空, 查看对数据库进行的修改。

New row updated and new RegionID assigned by database	
1 Eastern	Unchanged
2 Western	Unchanged
3 Northern	Unchanged
4 Southern	Unchanged
5 North West	Unchanged

看看 `DataTable` 中的最后一行。把代码中的 `RegionID` 设置为 999, 但在执行 `RegionInsert` 存储过程后, 该值改为 5, 这是本例的目的——数据库会生成主键码, 并且更新 `DataTable` 中的数据。`DataTable` 中的数据更新是因为源代码中的 `SqlCommand` 定义会把 `UpdatedRowSource` 属性设置为 `UpdateRowSource.OutputParameters`:

```
SqlCommand aCommand = new SqlCommand("RegionInsert", conn);

aCommand.CommandType = CommandType.StoredProcedure;
aCommand.Parameters.Add(new SqlParameter("@RegionDescription",
    SqlDbType.NChar,
    50,
    "RegionDescription"));
aCommand.Parameters.Add(new SqlParameter("@RegionID",
    SqlDbType.Int,
```

```
0 ,
ParameterDirection.Output ,
false ,
0 ,
0 ,
"RegionID" , // Defines the SOURCE column
DataRowVersion.Default ,
null));
aCommand.UpdatedRowSource = UpdateRowSource.OutputParameters;
```

无论何时数据适配器执行这个命令，输出参数都应映射回数据行的源，在本例中是 DataTable 中的一行，该标志说明了应更新什么数据——存储过程有一个输出参数映射为 DataRow，它应用的列是 RegionID，这是在命令中定义的。

UpdateRowSource 的值如表 26-13 所示。

表 26-13

UpdateRowSource 值	说 明
Both	存储过程可以返回输出参数和一个完整的数据库记录。这些数据源都用于更新源数据行
FirstReturnedRecord	该命令返回一个记录，该记录的内容应合并到最初的源 DataRow 中，当给定的表有许多默认(或计算)列时，使用这个值很有用，因为在执行插入语句之后，这些行需要与客户机上的 DataRow 同步。例如 INSERT (列) INTO (表) WITH (主键码)，SELECT (列) FROM (表) WHERE (主键码)。返回的记录应合并到源数据行上
None	删除从该命令返回的所有数据
OutputParameters	命令的任何输出参数都映射到 DataRow 的对应列上

2. 更新现有的行

更新 DataTable 中一个已有的行只需使用带有一个列名或列号的 DataRow 索引器即可，如下面的代码所示：

```
r["RegionDescription"]="North West England";
r[1] = "North East England";
```

这两个语句是等价的(在本例中)：

Changed RegionID 5 description	
1 Eastern	Unchanged
2 Western	Unchanged
3 Northern	Unchanged
4 Southern	Unchanged
5 North West England	Modified

在更新数据库前，被更新的行应把其状态设置为 Modified，其值如上所示。

### 3. 删除一行

删除一行需要调用 Delete()方法:

```
r.Delete();
```

被删除的行把其行状态设置为 Deleted, 但不能从被删除的 DataRow 中读取列, 因为它们已经不再有效, 当调用适配器的 Update()方法时, 所有被删除的行都会使用 DeleteCommand, 在本例中是执行 RegionDelete 存储过程。

## 26.8.2 写入 XML 输出结果

如上所述, DataSet 支持在 XML 中定义其模式。可以从 XML 文档中读取数据, 也可以把数据写入 XML 文档。

DataSet.WriteXml 方法可以输出存储在 DataSet 中的各种数据, 可以选择只输出数据, 也可以输出数据和模式。下面是为上述 Region 示例编写的代码:

```
ds.WriteXml("..\WithoutSchema.xml");
ds.WriteXml("..\WithSchema.xml", XmlWriteMode.WriteSchema);
```

第一个文件 WithoutSchema.xml 如下所示:

```
<?xml version="1.0" standalone="yes"?>
<NewDataSet>
  <Region>
    <RegionID>1</RegionID>
    <RegionDescription>Eastern</RegionDescription>
  </Region>
  <Region>
    <RegionID>2</RegionID>
    <RegionDescription>Western</RegionDescription>
  </Region>
  <Region>
    <RegionID>3</RegionID>
    <RegionDescription>Northern</RegionDescription>
  </Region>
  <Region>
    <RegionID>4</RegionID>
    <RegionDescription>Southern</RegionDescription>
  </Region>
</NewDataSet>
```

RegionDescription 上的闭合标记在页面的右边, 因为数据库列定义为 NCHAR(50), 这是一个包含 50 个字符的字符串, 其中用空格填充。

WithSchema.xml 文件生成的结果包含 DataSet 的 XML 模式和数据:

```
<?xml version="1.0" standalone="yes"?>
<NewDataSet>
  <xs:schema id="NewDataSet" xmlns=""
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
    <xs:element name="NewDataSet" msdata:IsDataSet="true">
      <xs:complexType>
        <xs:choice maxOccurs="unbounded">
```

```

    <xs:element name="Region">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="RegionID"
            msdata:AutoIncrement="true"
            msdata:AutoIncrementSeed="1"
            type="xs:int" />
          <xs:element name="RegionDescription"
            type="xs:string" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:choice>
</xs:complexType>
</xs:element>
</xs:schema>
<Region>
  <RegionID>1</RegionID>
  <RegionDescription>Eastern </RegionDescription>
</Region>
<Region>
  <RegionID>2</RegionID>
  <RegionDescription>Western </RegionDescription>
</Region>
<Region>
  <RegionID>3</RegionID>
  <RegionDescription>Northern </RegionDescription>
</Region>
<Region>
  <RegionID>4</RegionID>
  <RegionDescription>Southern </RegionDescription>
</Region>
</NewDataSet>

```

注意，使用 msdata 模式中的文件，定义 DataSet 中列的附加属性，例如 AutoIncrement 和 AutoIncrementSeed，这些属性直接对应于 DataColumn 类上的可定义属性。

## 26.9 使用 ADO.NET

本节介绍使用 ADO.NET 开发数据访问应用程序时的一些常见问题。

### 26.9.1 分层开发

开发与数据交互操作的应用程序时，常常要把应用程序分层，常见的模型是一个应用层(前端)、一个数据服务层和数据库本身。

但使用这个模型的难题是，确定在层之间传输什么数据，以及应采用什么格式来传输数据。有了 ADO.NET，就不必担心这个问题了，这种结构一开始就提供了这种支持。

在 ADO.NET 中，对复制完整的记录集有比 OLEDB 更好的支持。在 .NET 中，只需复制 DataSet 即可：

```

DataSet source = {some dataset};
DataSet dest = source.Copy();

```



这将创建源 DataSet 的一个完全相同的副本——所有的 DataTable、DataColumn、DataRow 和 Relation 都会复制下来，所有的数据所处的状态都与它在源 DataSet 中所处的状态完全相同。如果只需要复制 DataSet 的模式，可以试试下面的代码：

```
DataSet source = {some dataset};
DataSet dest = source.Clone();
```

这也会复制所有的表、关系等，但每个复制的 DataTable 都是空的。这个过程非常简单。在编写一个分层的系统(无论该系统是基于 Win32 还是基于 Web)时，常见的要求是在层之间附带尽可能少的数据。这减少了资源的消耗。

要达到这个要求，DataSet 类提供了 GetChanges() 方法。这个方法执行许多任务，返回一个 DataSet，其中只包含源数据集中修改过的行。这是在层之间传输数据时最理想的情况，因为只有很少量的数据在网络上传输。

下面的示例说明了如何生成对 DataSet 的一个修改：

```
DataSet source = {some dataset};
DataSet dest = source.GetChanges();
```

这是很乏味的，下面介绍的内容会比较有趣。GetChanges 方法有两个重载形式，一个重载方法的参数是 DataRowState 枚举的一个值，返回对应于该状态的行。GetChanges() 只调用 GetChanges(Deleted | Modified | Added)，该方法首先检查，以确保调用 HasChanges 进行了一些修改。如果没有修改，就立即给调用者返回空值。

下一个操作是复制当前的 DataSet。完成后，会建立新的 DataSet，以忽略违反约束的情况 (EnforceConstraints = false)，然后，把每个表中所有修改的行都复制到新的 DataSet 中。

在得到一个只包含修改内容的 DataSet 后，就可以把它们移动到数据服务层上进行处理。数据在数据库中更新后，修改了的数据集就返回给调用者(例如，存储过程的一些输出参数在某些列上有更新的值)，然后使用 Merge() 方法，把这些修改的内容合并到源 DataSet 中。操作的顺序如图 26-9 所示。

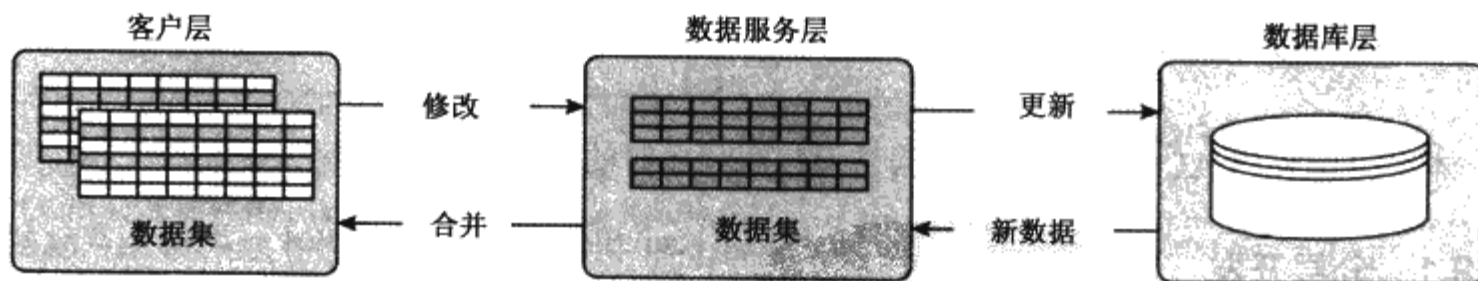


图 26-9

### 26.9.2 生成 SQL Server 的键

本章前面给出的 RegionInsert 存储过程在给数据库插入数据时生成了一个主键码值。该例中生成该键的方法相当原始，没有很好地考虑，真正的应用程序应利用生成键的其他策略。

首先要定义一个 Identity 列，从存储过程中返回 @@IDENTITY 值。下面的存储过程显示了如何为 Northwind 示例数据库中的 Categories 表定义该值。在 SQL 查询分析器中输入这个存储过程，或者运行下载代码中的 StoredProcs.sql 文件：

```

CREATE PROCEDURE CategoryInsert(@CategoryName NVARCHAR(15),
                                @Description NTEXT,
                                @CategoryID INTEGER OUTPUT) AS
SET NOCOUNT OFF
INSERT INTO Categories (CategoryName, Description)
VALUES(@CategoryName, @Description)
SELECT @CategoryID = @@IDENTITY
GO

```

这将把一个新行插入到 Category 表中，给调用者返回生成的主键码(CategoryID 列的值)。在查询分析器中输入下述 SQL 命令，可以测试该过程：

```

DECLARE @CatID int;
EXECUTE CategoryInsert 'Pasties' , 'Heaven Sent Food' , @CatID OUTPUT;
PRINT @CatID;

```

当把该过程作为批处理命令来执行时，会把一个新行插入到 Category 表中，返回新记录的标识(identity)值，然后把该列显示给用户。

假定过了几个月后，有人要添加一个简单的审计跟踪，记录所有插入的记录，以及对 Category 名的修改。此时，定义如图 26-10 所示的一个表，记录 Category 的新名和旧名。

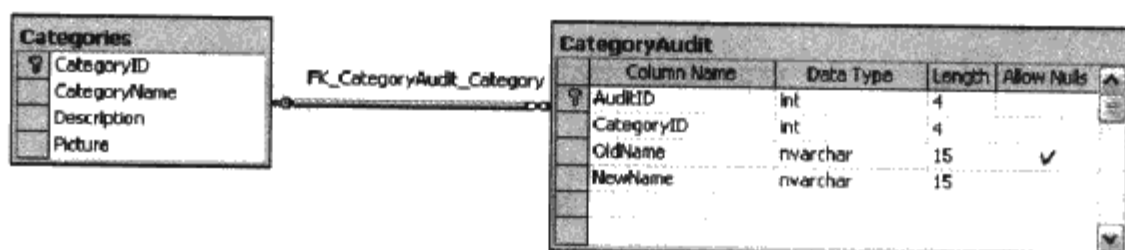


图 26-10

这个表的脚本在 StoredProcs.sql 文件中。AuditID 列定义为一个 IDENTITY 列，然后构造两个数据库触发器，记录对 CategoryName 字段的修改：

```

CREATE TRIGGER CategoryInsertTrigger
ON Categories
AFTER UPDATE
AS
INSERT INTO CategoryAudit(CategoryID , OldName , NewName )
SELECT old.CategoryID, old.CategoryName, new.CategoryName
FROM Deleted AS old,
Categories AS new
WHERE old.CategoryID = new.CategoryID;
GO

```

对于习惯使用 Oracle 存储过程的用户来说，SQL Server 其实没有 OLD 和 NEW 行的概念，而是使用一个插入触发器，在内存中有一个 Inserted 表可用于插入记录，在 Deleted 表中删除和更新旧记录。

该触发器获取要处理的记录的 CategoryID，把它与 CategoryName 列的新旧值一起存储起来。

现在，调用原来的存储过程插入一个新 CategoryID 时，会得到一个 identity 值，但它不再是插入到 Categories 表中的行的 identity 值，而是 CategoryAudit 表中为该行生成的新值。

要查看这些值，可打开 SQL Server Enterprise 管理器的一个副本，查看 Categories 表的内容，

如图 26-11 所示。

CategoryID	CategoryName	Description	Picture
1	Beverages	Soft drinks, coffees, teas, beers, and ales	<Binary>
2	Condiments	Sweet and savory sauces, relishes, spreads, and seasonings	<Binary>
3	Confections	Desserts, candies, and sweet breads	<Binary>
4	Dairy Products	Cheeses	<Binary>
5	Grains/Cereals	Breads, crackers, pasta, and cereal	<Binary>
6	Meat/Poultry	Prepared meats	<Binary>
7	Produce	Dried fruit and bean curd	<Binary>
8	Seafood	Seaweed and fish	<Binary>

图 26-11

图 26-11 列出了 Northwind 数据库中所有的类别。

Categories 表中的下一个 identity 值是 9，所以执行上面的代码插入一个新行，看看返回了什么 ID，如下所示。

```
DECLARE @CatID int;
EXECUTE CategoryInsert 'Pasties' , 'Heaven Sent Food' , @CatID OUTPUT;
PRINT @CatID;
```

其输出值是 1。如果查看 CategoryAudit 表，则这是新插入的审核记录的 identity 值，不是新建 category 记录的值，如图 26-12 所示。

AuditID	CategoryID	OldName	NewName
1	9	<NULL>	Pasties

图 26-12

问题是@@IDENTITY 仍在起作用，它返回由会话创建的最后一个 identity 值，所以不是 100%的可靠。

除了@@IDENTITY 外，还可以使用另外两个 identity 函数，它们都不会出任何问题。第一个函数是 SCOPE\_IDENTITY()，它返回在当前“范围”内最后一个创建的 identity 值。SQL Server 把该范围定义为一个存储过程、触发器或函数。在大多数情况下，这个函数可以正常工作。但如果因某种原因，有人在存储过程中添加了另一个 INSERT 语句，就会得到这个值，而不是期望的值。

另一个函数是 IDENT\_CURRENT()，它返回任何范围中为给定表生成的最后一个 identity 值。例如，如果两个用户同时访问 SQL Server，其中一个用户就有可能得到另一个用户生成的 identity 值。

可以想像，找出这个问题的原因是不太容易的。在 SQL Server 中使用 IDENTITY 列时要多加小心。

26.9.3 命名约定

下面的提示和约定与.NET 并不直接相关，但应共享和遵循，特别是在给约束命名时是非常有效的，如果您对此主题已经有了自己的命名方式，就可以跳过本节。

### 1. 数据库表的约定

- 总是使用单数名称——Product 而不是 Products，这是一个普遍适用的约定，因为我们必须给客户解释某种数据库模式，从语法上看，“product 表包含产品”要比“products 表包含产品”好得多。但 Northwind 数据库并没有遵循这一约定。
- 给表中的字段采用某种形式的命名约定，我们采用的是表的主键码为<Table>\_ID(假定主键码是一个列)，字段采用 Name，这是考虑到记录的用户友好性，记录本身的文本信息采用 Description。采用好的命名约定，则只要查看一下数据库中的表，就知道其中的字段主要用于什么目的。

### 2. 数据库列的约定

- 使用单数名称，而不是复数名称。
- 链接到另一个表中的列名应与该表的主键码名相同，所以，链接到 Product 表的列名为 Product\_ID。链接到 Sample 表的列名为 Sample\_ID，这并不总是可行的，如果一个表有另一个表的多个引用，这个命名约定就无效了。此时应使用其他方式命名。
- 日期字段应有一个\_On 后缀，例如 Modified\_On、Created\_On。按照这种命名约定，如果读取一些 SQL 输出，很容易从列的名称中知道该列的含义。
- 记录用户的字段名应有一个\_By 后缀，例如 Modified\_By 和 Created\_By，这将有助于阅读。

### 3. 约束的约定

- 如果可能，在约束名中包含表名和列名，例如 CK\_<Table>\_<Field>。对于 person 表中的 Sex(性别)列，其检查约束可以是 CK\_PERSON\_SEX，而 product 和 supplier 表之间的外键码名可以是 FK\_Product\_Supplier\_ID。
- 约束类型的前面加一个前缀，例如 CK 表示检查约束，FK 表示外键码约束，也可以指定更为特殊的名称，例如 Age 列上的 CK\_PERSON\_AGE\_GT0 表示该年龄应大于 0。
- 如果必须限制约束名的长度，可以在其中包含表名，而不包含列名。在发现有违反约束的情况时，通常很容易知道哪个表出现错误，但有时不容易检查出是哪个列出了问题。Oracle 允许名称的长度为 30 个字符，而这种限制很容易被打破。

### 4. 存储过程

在过去几年中，把 C 放在每个声明的类前面令人困惑。同样，许多 SQL Server 开发人员也困惑于在每个存储过程的前面加上 sp\_ 或类似的东西，这不是一个好方法。

SQL Server 在所有的系统存储过程前面使用 sp\_ 前缀。所以一方面，用户会对 sp\_widget 是否为 SQL Server 标准存储过程产生疑问，另外，当查找存储过程时，SQL Server 会把带有 sp\_ 前缀的过程与没有该前缀的过程区别对待。

如果使用这个前缀，但没有用该存储过程的数据库/拥有者来限定，SQL Server 就会在当前范围内查找，然后跳到主数据库中查找存储过程。没有 sp\_ 前缀，用户就会早一些得到错误。更糟糕的是所创建的本地存储过程(在自己的数据库中创建的)与系统的存储过程有相同的名称

和参数。应尽可能避免这种情况，如果有疑问，就不要使用前缀。

在调用存储过程时，应在名称的前面加上过程拥有者前缀，如 `dbo.selectWidgets`。这将比不使用该前缀略快一些，因为 SQL Server 查找该存储过程所做的工作较少。这不会对应用程序的执行速度有很大的影响，但可以自由使用。

在数据库或代码中命名实体时，应采用一致的命名约定。

## 26.10 小结

数据访问是一个很大的主题，特别是在 .NET 中，有非常丰富的内容要论及。本章概述了 ADO.NET 命名空间中的主要类，解释了在处理数据源中的数据时如何使用这些类。

首先通过 `SqlConnection` (SQL Server 专用)和 `OleDbConnection` (任何 OLE DB 数据源)的使用，讨论了 `Connection` 对象的用法。这两个类的编程模型非常类似，一般其中一个类可以替代另一个类，代码仍能继续运行。在 .NET 1.1 中，还可以使用 Oracle 提供程序和 ODBC 提供程序。

接着阐述了如何正确地进行连接，这样稀缺的资源就可以尽可能早地关闭。所有的连接类都执行 `IDisposable` 接口，在对象放在 `using` 子句中时调用。如果本章只有一件值得注意的事，那就是尽早关闭数据库连接的重要性。

然后通过执行没有返回数据的示例、利用输入和输出参数调用存储过程，讨论了数据库命令。本章描述了各种执行方法，包括只能在 SQL Server 提供程序上使用的 `ExecuteXmlReader` 方法。这大大简化了基于 XML 的数据的选择和处理。

这里详细介绍了 `System.Data` 命名空间中的常用类，包括 `DataSet`、`DataTable`、`DataColumn`、`DataRow`，以及关系和约束。`DataSet` 类是数据的绝佳容器，各种方法使之成为跨层数据流的理想容器。`DataSet` 中的数据可以用 XML 来表示，以利于传输，另外，可用的方法能在层次之间传输最少量的数据。把许多数据表放在一个 `DataSet` 中可以大大提高其使用性。下一章将扩展论述能自动维护主/从数据行之间关系的功能。

除了把模式存储在 `DataSet` 中外，.NET 还包括数据适配器，它与各种 `Command` 对象组合使用，可以把数据选择出来，放在 `DataSet` 中，以后还可以更新数据库中的数据。数据适配器的一个优点是可以为 4 个操作(SELECT、INSERT、UPDATE 和 DELETE)中的每一个定义不同的命令。系统可以根据数据库模式信息和一个 SELECT 语句创建一组默认的命令，但为了得到最佳性能，可以使用一组存储过程，并定义一组合适的 `DataAdapter` 命令，仅把需要的信息传送给这些存储过程。

我们还介绍了 XSD 工具(`xsd.exe`)，使用一个示例来说明如何在 .NET 中基于 XML 模式使用类。产生的类可以用于应用程序，它们的自动生成减少了大量的键入工作。

最后论述了用于数据库开发的一些好的方式和命名约定。

访问 SQL Server 数据库的更多知识，详见第 27 章。



# 第 27 章

## LINQ to SQL

在 .NET Framework 3.5 中, 最令人激动的新增功能是添加到 C# 2008 中的 .NET Language Integrated Query Framework (LINQ)。LINQ 主要在数据集成的基础上提供了一种轻型方式。这是一个很好的功能, 因为数据是最重要的。

每个应用程序都以某种方式处理数据, 无论这些数据来自内存(内存中的数据)、数据库、XML 文件、文本文件还是其他地方。许多开发人员发现很难把 C# 中强类型化的、面向对象的数据移动到对象是辅助类的数据库层中。在最好的情况下, 从一个环境转换到另一个环境充满了易于出错的操作。

在 C# 中, 用对象编程意味着利用一个强类型化的强大功能来处理代码。很容易在命名空间中导航, 使用 Visual Studio IDE 中的调试器等。但是, 在访问数据时, 情况就大不相同了。

那是一个没有强类型化的环境, 调试是很痛苦的, 甚至是不可能的, 大部分时间都花在把字符串作为命令传送给数据库。开发人员还必须注意底层的数据, 了解它的构造方式以及所有数据点的相互关系。

微软公司把 LINQ 提供为一种轻型方式, 为底层的数据存储提供了一个强类型化的界面。LINQ 为开发人员提供了在编码环境下工作的方式, 可以把底层的数据作为对象来访问, 使用 IDE、IntelliSense 甚至调试功能。

有了 LINQ, 我们创建的查询现在就变成 .NET Framework 和其他环境中的一流成员。在对要操作的数据存储执行查询时, 会很快发现它们现在的操作方式类似于系统中的类型。这说明, 现在可以使用任意兼容 .NET 的语言来查询底层的数据存储, 这在以前是不可能的。

**提示:**

第 11 章概述了 LINQ。

图 27-1 显示了 LINQ 在查询数据中的作用。

在图 27-1 中, 根据要在应用程序中处理的底层数据的不同, 有不同类型的 LINQ 功能。下面列出了各种 LINQ 技术:

- LINQ to Objects
- LINQ to DataSets
- LINQ to SQL
- LINQ to Entities
- LINQ to XML

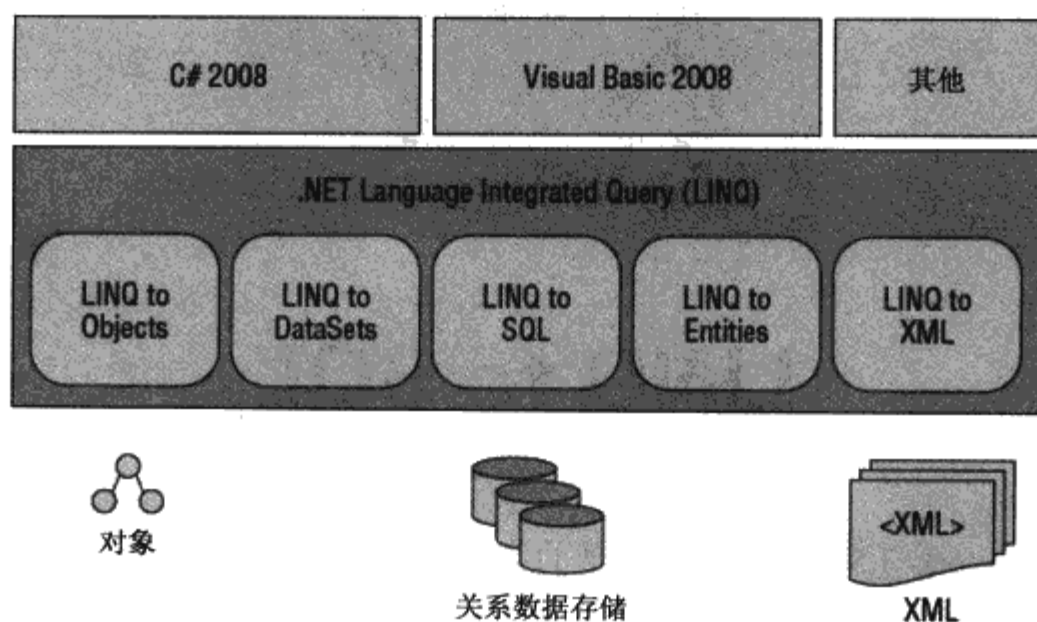


图 27-1

类库中提供了可以使用 LINQ 查询的对象，像查询其他数据存储那样。其实，对象只不过是存储在内存中的数据。实际上，对象本身可能就在查询数据。此时应使用 LINQ to Objects。

LINQ to SQL(本章的重点)、LINQ to Entities 和 LINQ to DataSets 都提供了查询关系数据的方式。使用 LINQ 可以直接查询数据库，甚至查询数据库中的存储过程。图中的最后一项是使用 LINQ to XML 查询 XML(详见第 29 章)。LINQ 非常强大的原因是它对所查询的内容没有任何限制，因为查询非常类似。

本章内容如下：

- 使用 LINQ to SQL 和 Visual Studio 2008
- LINQ to SQL 对象如何映射到数据库实体上
- 建立 LINQ to SQL 操作，但不使用 O/R 设计器
- 使用 O/R 设计器和定制对象
- 用 LINQ 查询 SQL Server 数据库
- 存储过程和 LINQ to SQL

## 27.1 LINQ to SQL 和 Visual Studio 2008

LINQ to SQL 是在 SQL Server 数据库上设置一个强类型化界面的方式。LINQ to SQL 提供的方法是查询 SQL Server 的最简单的方法。不仅在数据库中查询单个表非常简单，而且，如果调用 Northwind 数据库的 Customers 表，提取该数据库的 Orders 表中某位顾客的特定订单，LINQ 就使用两个表之间的关系来建立查询。LINQ 会查询数据库，提取要在代码中使用的数据(也是强类型化的)。

注意，LINQ to SQL 不仅可以查询数据，还可以执行需要的插入/更新/删除语句。

也可以与整个过程交互操作，定制所执行的操作，给 CRUD(Create/Read/Update/Delete)操作添加自己的业务逻辑。

Visual Studio 2008 非常重视 LINQ to SQL，因为这是一个可扩展的用户界面，允许设计要使用的 LINQ to SQL 类。

本章的下一节介绍如何建立第一个 LINQ to SQL 实例，从 Northwind 数据库的 Products 表中提取数据项。

### 27.1.1 使用 LINQ to SQL 调用 Products 表——创建控制台应用程序

作为使用 LINQ to SQL 的一个例子，本章首先调用 Northwind 数据库中的单个表，用这个表把一些结果显示在屏幕上。

首先创建一个控制台应用程序(使用 .NET Framework 3.5)，把 Northwind 数据库文件添加到这个项目中(Northwind.MDF)。

**提示：**

下面的例子使用 SQL Server Express Database 文件 Northwind.mdf。要获得这个数据库，请搜索“Northwind and pubs Sample Database for SQL Server 2000”。这个链接在 <http://www.microsoft.com/downloads/details.aspx?familyid=06616212-0356-46a0-8da2-eebc53-a68034&displaylang=en>。安装后，Northwind.mdf 文件位于 C:\SQL Server 2000 Sample Database 目录下。要把这个数据库添加到应用程序中，可以右击正在操作的解决方案，选择 Add Existing Item。在打开的对话框中，找到刚才安装的 Northwind.mdf 文件。如果在获得使用该数据库的权限方面有问题，可以在 Visual Studio Server Explorer 中建立与该文件的数据连接，使自己成为该数据库的相应用户。Visual Studio 就会进行相应的改变，允许使用该数据库。

现在，在 Visual Studio 2008 中创建 .NET Framework 3.5 提供的许多应用程序类型时，默认已经有了使用 LINQ 的正确引用。在创建控制台应用程序时，代码中会包含如下 using 语句：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Sockets;
using System.Runtime.Remoting.Messaging;
using System.Text;
```

可以看出，代码已经包含了需要的 LINQ 引用。下一步是添加一个 LINQ to SQL 类。

### 27.1.2 添加 LINQ to SQL 类

使用 LINQ to SQL 时，一个主要优点是 Visual Studio 2008 会使该任务尽可能简单。Visual Studio 2008 提供了一个与对象相关的映射设计器，称为 O/R 设计器，它允许可视化地设计数据库要映射的对象。

首先，右击解决方案，从打开的菜单中选择 Add New Item。在 Add New Item 对话框中，包含了 LINQ to SQL Classes 选项。如图 27-2 所示。

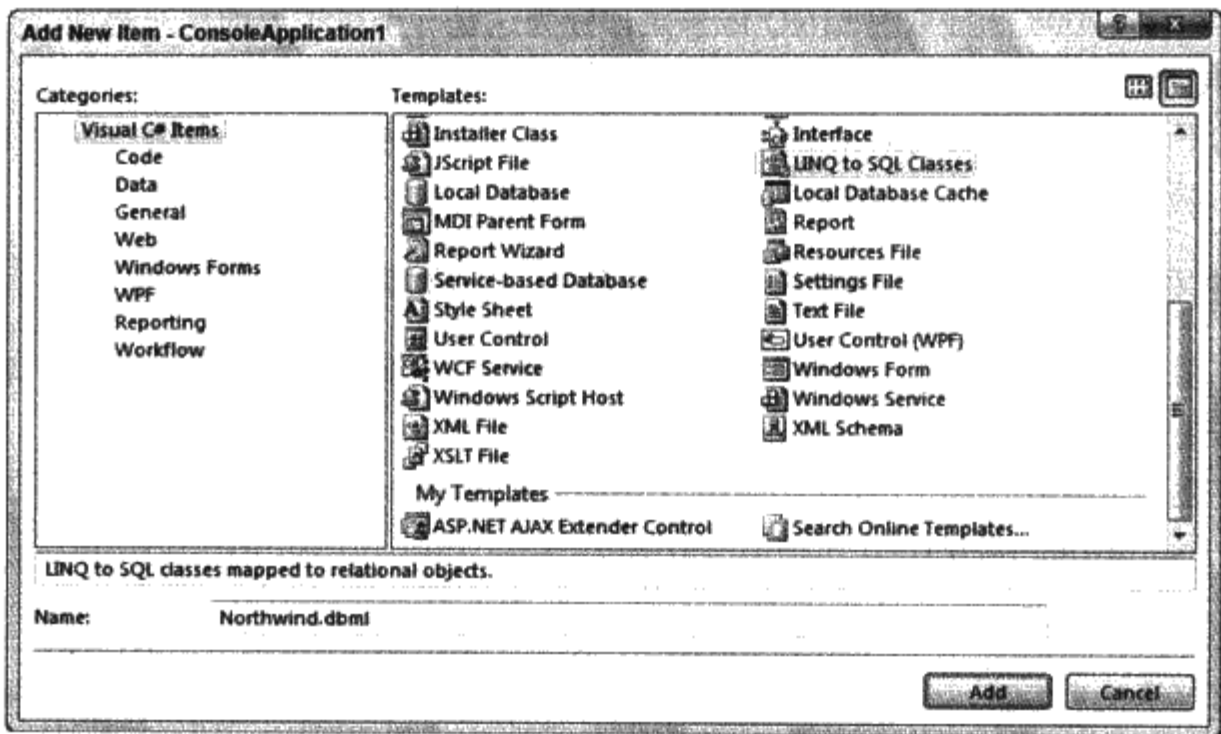


图 27-2

这个例子使用 Northwind 数据库，所以把文件命名为 Northwind.dbml。单击 Add 按钮，就会创建几个文件，图 27-3 显示了添加 Northwind.dbml 文件后的 Solution Explorer。

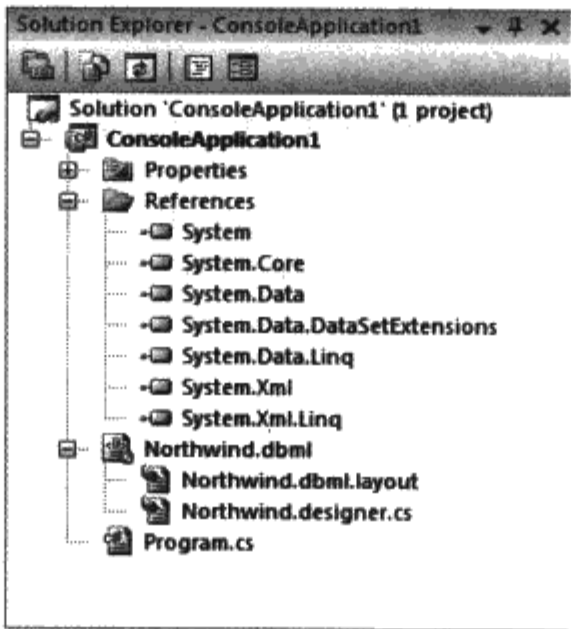


图 27-3

这个操作在项目中添加了许多内容。添加了 Northwind.dbml 文件，它包含两个组件。前面添加的 LINQ to SQL 类要使用 LINQ，所以也添加了下面的引用：System.Core、System.Data.DataSetExtensions、System.Data.Linq 和 System.XML.Linq。

27.1.3 O/R 设计器概述

在项目中添加 LINQ to SQL 类(Northwind.dbml 文件)时，还在 IDE 上添加了一个新功能：.dbml 文件的可视化表示。新的 O/R 设计器在 IDE 的文档窗口中显示为一个选项卡。图 27-4 显示了 O/R 设计器第一次启动时的视图。

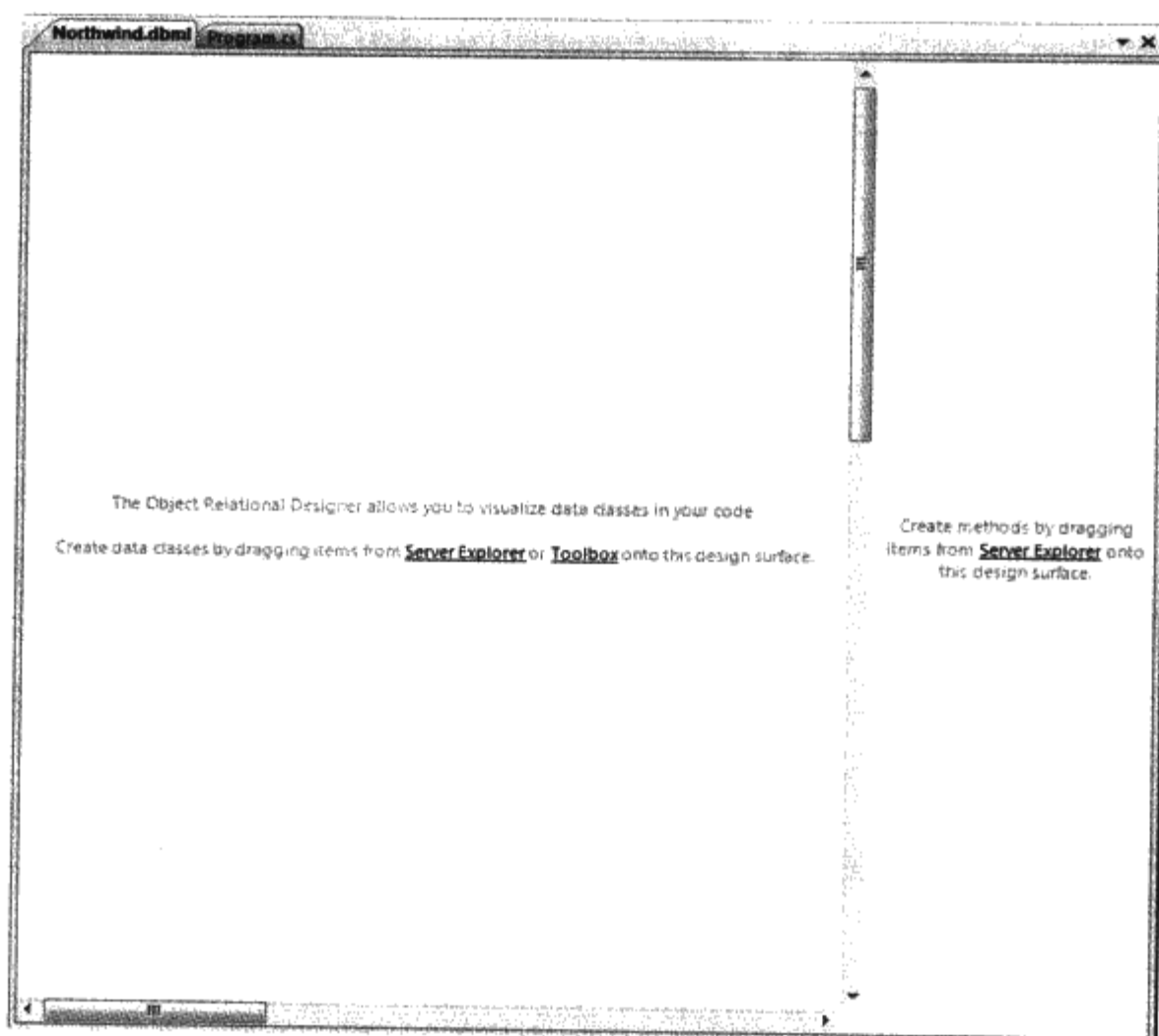


图 27-4

O/R 设计器由两部分组成。第一部分用于显示数据类，它可以是表、类、关联和继承。在这个设计界面上拖动这些项，可以显示所操作的对象的可视化表示。第二部分(右边)用于显示方法，这些方法映射到数据库中的存储过程上。

在 O/R 设计器中查看.dbml 文件时，Visual Studio 工具箱中还有一组 Object Relational Designer 控件。该工具箱如图 27-5 所示。

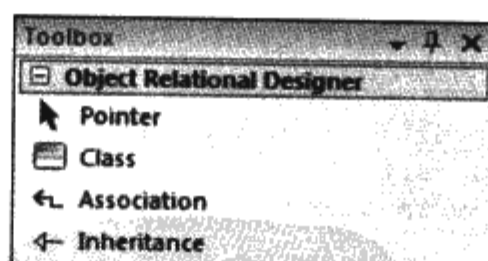


图 27-5

#### 27.1.4 创建 Product 对象

这个例子要使用 Northwind 数据库中的 Products 表，这表示要创建一个 Products 表，使用 LINQ to SQL 映射这个表。这只需在 Visual Studio 的 Server Explorer 中打开包含在该数据库中的表视图，把 Products 表拖放到 O/R 设计器的设计界面上。这个操作的结果如图 27-6 所示。



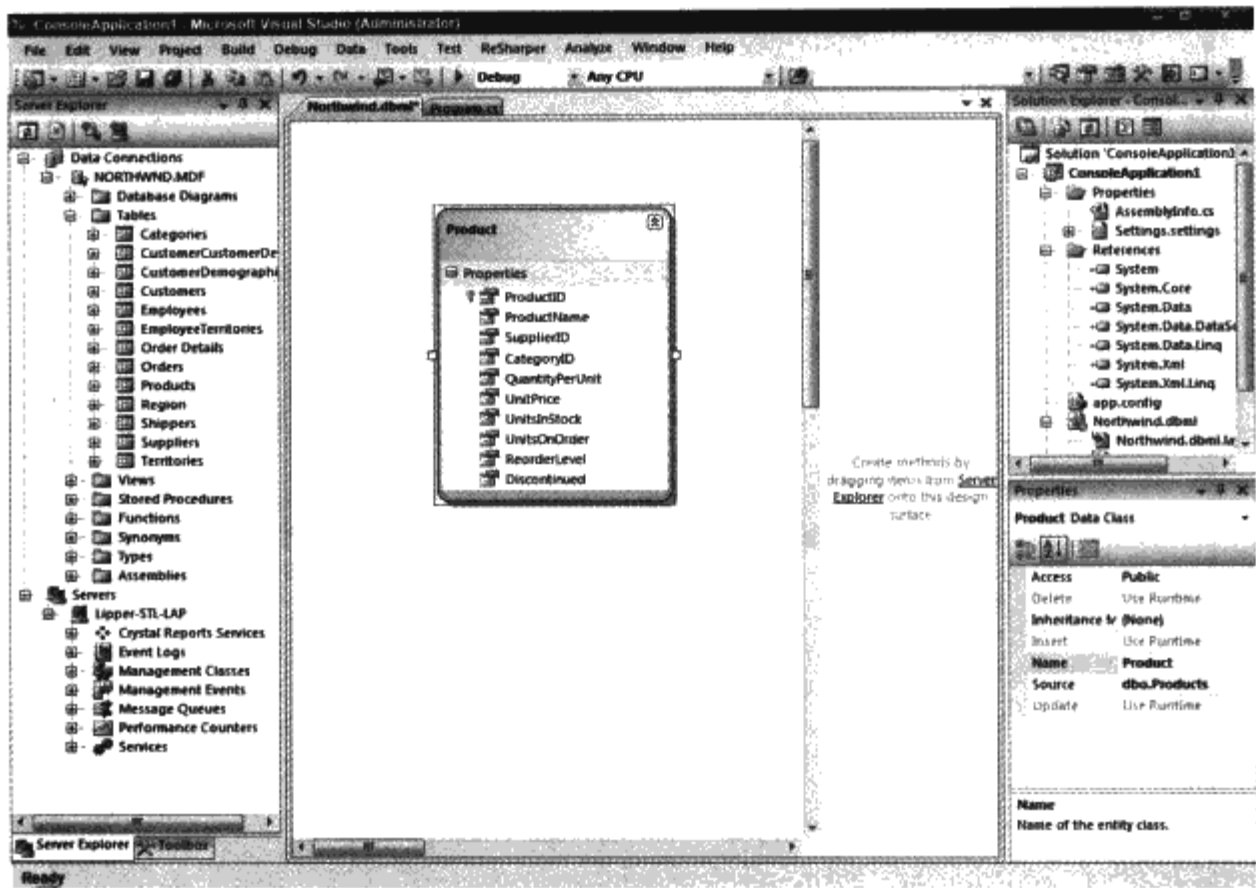


图 27-6

执行这个操作，会把一些代码添加到.dbml 文件的设计器文件中。这些类可以对 Products 表进行强类型化访问。为了演示这个访问，把注意力转向控制台应用程序的 Program.cs 文件上。下面列出了这个例子需要的代码：

```
using System;
using System.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main(string[] args)
        {
            NorthwindDataContext dc = new NorthwindDataContext();
            var query = dc.Products;
            foreach (Product item in query)
            {
                Console.WriteLine("{0} | {1} | {2}",
                    item.ProductID, item.ProductName, item.UnitsInStock);
            }
            Console.ReadLine();
        }
    }
}
```

这些代码并不多，但查询了 Northwind 数据库中的 Products 表，提取出了要显示的数据。下面单步执行这段代码，从 Mian()方法的第一行开始：

```
NorthwindDataContext dc = new NorthwindDataContext();
```

NorthwindDataContext 对象是 DataContext 类型的一个对象。基本上可以把这个对象看作映

射到 `Connection` 类型对象上的一个对象。这个对象可以使用连接字符串，为需要的操作连接数据库。

下一行比较有趣：

```
var query = dc.Products;
```

这里使用了新的 `var` 关键字，它是一个隐式的类型化变量。如果不能确定输出类型，就可以使用 `var` 来替代类型定义，再在编译期间设置类型。事实上，代码 `dc.Products` 返回一个 `System.Data.Linq.Table<ConsoleApplication1.Product>` 对象，这就是编译应用程序时设置的 `var`。因此，这表示也可以编写如下语句：

```
Table < Product > query = dc.Products;
```

这种方法比较好，因为程序员以后查看应用程序的代码时，很容易理解其含义。使用 `var` 关键字还有一个隐含的优点：程序员可能会发现它有问题。要使用 `Table<Product>`（这基本上是 `Product` 对象的一个泛型列表），就应引用 `System.Data.Linq` 命名空间。

赋予 `query` 对象的值是 `Products` 属性的值，它是 `Table<Product>` 类型。之后，下面的代码迭代 `Table<Product>` 中的 `Product` 对象集合：

```
foreach (Product item in query)
{
    Console.WriteLine("{0} | {1} | {2}",
        item.ProductID, item.ProductName, item.UnitsInStock);
}
```

这个迭代从 `Product` 对象中提取出了 `ProductID`、`ProductName` 和 `UnitsInStock` 属性，并把它们写到程序中。我们仅使用了表的几项，所以还可以使用 O/R 设计器中的选项删除从数据库中提取出来的、我们不感兴趣的列。程序的结果如下：

```
1 | Chai | 39
2 | Chang | 17
3 | Aniseed Syrup | 13
4 | Chef Anton's Cajun Seasoning | 53
5 | Chef Anton's Gumbo Mix | 0

** Results removed for space reasons **

73 | R d Kaviar | 101
74 | Longlife Tofu | 4
75 | Rh nbr u Klosterbier | 125
76 | Lakkalik ri | 57
77 | Original Frankfurter gr ü ne So e | 32
```

从这个例子可以看出，很容易使用 LINQ to SQL 查询 SQL Server 数据库。

## 27.2 对象映射到 LINQ 对象上

LINQ 的优点是提供了在代码(和 IntelliSense)中使用的强类型化对象，这些对象还映射到已有的数据库对象上。另外，LINQ 不过是这些已有数据库对象上的一个瘦层面。表 27-1 列出了

数据库对象和 LINQ 对象之间的映射。

表 27-1

数据库对象	LINQ 对象
数据库	DataContext
表	类和集合
视图	类和集合
列	属性
关系	嵌套的集合
存储过程	方法

左列是数据库，数据库是一个完整的实体——表、视图、触发器、存储过程构成了数据库。右边的 LINQ 对象中，有一个 DataContext 对象，它绑定到数据库上。为了与数据库进行必要的交互操作，该对象包含一个连接字符串，并管理所发生的所有事务处理，还负责记录操作，管理数据的输出。DataContext 对象全面管理与数据库的事务处理。

如前面的例子所示，表转换为类。这表示如果有一个 Products 表，就有一个 Product 类。注意 LINQ 的名称是比较友好的，因为它把复数形式的表名改为单数形式，给要在代码中使用的类指定了合适的名称。除了用作类的数据库表之外，数据库视图也处理为类。而列处理为属性，因此可以直接管理列的属性(名称和类型定义)。

关系是在各个对象之间映射的嵌套集合。因此可以定义映射到多个项上的关系。

还必须理解存储过程的映射。在代码中，存储过程实际上映射到 DataContext 实例的方法上。下一节将详细介绍 DataContext 和 LINQ 中的表对象。

在处理 LINQ to SQL 的体系结构时，注意其中有 3 层：应用程序、LINQ to SQL 层和 SQL Server 数据库。在前面的例子中，可以在应用程序的代码中创建强类型化的查询：

```
dc.Products;
```

这个查询会被 LINQ to SQL 层转换为 SQL 查询，之后提供给数据库：

```
SELECT [t0].[ProductID], [t0].[ProductName], [t0].[SupplierID],
[t0].[CategoryID], [t0].[QuantityPerUnit], [t0].[UnitPrice],
[t0].[UnitsInStock], [t0].[UnitsOnOrder], [t0].[ReorderLevel],
[t0].[Discontinued]
FROM [dbo].[Products] AS [t0]
```

结果，LINQ to SQL 层通过这个查询从数据库中提取行，把返回的数据变成一个强类型化的对象集合，以便于使用。

27.2.1 DataContext 对象

在使用 LINQ to SQL 时，DataContext 对象管理在所操作的数据库中发生的事务处理。使用 DataContext 对象可以执行许多操作。

在实例化该对象时，需要几个可选参数，如下：

- 一个字符串，表示 SQL Server Express 数据库文件的位置或所使用的 SQL Server 的名称
- 连接字符串
- 另一个 DataContext 对象

前两个可选字符串参数也可以包含自己的数据库映射文件。实例化这个对象后，可以在许多不同的操作中编程使用它。

### 1. 使用 ExecuteQuery

使用 DataContext 对象完成的一个简单任务是用 ExecuteQuery<T>() 方法快速运行自己编写的命令。例如，如果要使用 ExecuteQuery<T>() 方法提取 Products 表中的所有产品，代码应如下所示：

```
using System;
using System.Collections.Generic;
using System.Data.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main(string[] args)
        {
            DataContext dc = new DataContext(@"Data Source=.\SQLEXPRESS;
            AttachDbFilename=|DataDirectory|\NORTHWND.MDF;
            Integrated Security=True;User Instance=True");

            IEnumerable < Product > myProducts =
                dc.ExecuteQuery < Product > ("SELECT * FROM PRODUCTS", "");
            foreach (Product item in myProducts)
            {
                Console.WriteLine(item.ProductID + " | " + item.ProductName);
            }
            Console.ReadLine();
        }
    }
}
```

在这个例子中，调用 ExecuteQuery<T>() 方法时传送了一个查询字符串，返回一个 Product 对象集合。在该方法调用中使用的查询是一个简单的 Select 语句，它不需要传送任何参数。由于查询中没有传送任何参数，因此需要使用双引号作为方法调用的第二个必选参数。如果要在查询中替换任何值，可以按如下方式构建 ExecuteQuery<T>() 调用：

```
IEnumerable < Product > myProducts =
    dc.ExecuteQuery <Product> ("SELECT * FROM PRODUCTS WHERE UnitsInStock >
    {0}", 50);
```

在这个例子中，{0} 是一个占位符，用于替换要传入的参数值，ExecuteQuery<T>() 方法的第二个参数是在替换过程中使用的参数。

## 2. 使用 Connection 属性

Connection 属性返回 System.Data.SqlClient.SqlConnection 的一个实例, 由 DataContext 对象使用。如果需要与应用程序中使用的其他 ADO.NET 代码共享这个连接, 或者需要获得该连接的 SqlConnection 属性或方法, 就可以使用这个属性。例如, 获得连接字符串就很简单:

```
NorthwindDataContext dc = new NorthwindDataContext();
Console.WriteLine(dc.Connection.ConnectionString);
```

## 3. 使用 Transaction 属性

如果有一个可以使用的 ADO.NET 事务处理, 就可以使用 Transaction 属性把这个事务处理赋予 DataContext 对象实例。还可以通过 .NET 2.0 Framework 中的 TransactionScope 对象使用事务处理:

```
using System;
using System.Collections.Generic;
using System.Data.Linq;
using System.Transactions;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main(string[] args)
        {
            NorthwindDataContext dc = new NorthwindDataContext();

            using (TransactionScope myScope = new TransactionScope())
            {
                Product p1 = new Product() { ProductName = "Bill's Product" };
                dc.Products.InsertOnSubmit(p1);

                Product p2 = new Product() { ProductName = "Another Product" };
                dc.Products.InsertOnSubmit(p2);
                try
                {
                    dc.SubmitChanges();
                    Console.WriteLine(p1.ProductID);
                    Console.WriteLine(p2.ProductID);
                }
                catch (Exception ex)
                {
                    Console.WriteLine(ex.ToString());
                }
                myScope.Complete();
            }
            Console.ReadLine();
        }
    }
}
```

在这个例子中, 使用了 TransactionScope 对象, 如果数据库中的某个操作失败, 就回退所有的操作, 返回最初的状态。



4. DataContext 对象的其他方法和属性

除了前面介绍的成员之外，DataContext 对象还有许多其他方法和属性。表 27-2 列出了 DataContext 对象的一些方法。

表 27-2

方 法	说 明
CreateDatabase	可以在服务器上创建数据库
DatabaseExists	可以确定数据库是否存在，是否可以打开
DeleteDatabase	删除相关的数据库
ExecuteCommand	可以给数据库传送要执行的命令
ExecuteQuery	可以给数据库传送查询
GetChangeSet	DataContext 对象跟踪数据库中发生的变化，这个命令可以访问这些变化
GetCommand	可以访问要执行的命令
GetTable	可以访问数据库中的表集合
Refresh	可以利用数据库中存储的数据刷新对象
SubmitChanges	在数据库上执行代码建立的 CRUD 命令
Translate	把 IDataReader 转换为对象

除了这些方法之外，DataContext 对象还包含一些属性，如表 27-3 所示。

表 27-3

属 性	说 明
ChangeConflicts	调用 SubmitChanges()方法时，提供一个导致并发冲突的对象集合
CommandTimeout	可以设置一个超时时间，命令允许在这个时间段内在数据库上执行。如果查询需要执行较长时间，就应把这个属性设置为较高的值
Connection	可以处理客户使用的 System.Data.SqlClient.SqlConnection 对象
DeferredLoadingEnabled	可以指定是否延迟加载一对多或一对一关系
LoadOptions	可以指定或检索 DataLoadOptions 对象的值
Log	可以指定在查询中使用的命令的输出位置
Mapping	提供映射所基于的 MetaModel
ObjectTrackingEnabled	指定是否跟踪数据库中对象的变化，以进行事务处理。如果处理只读数据库，就应把这个属性设置为 false
Transaction	可以指定数据库中使用的本地事务处理

27.2.2 Table<TEntity>对象

Table<TEntity>对象表示在数据库中操作的表。例如，前面使用了 Product 类，它就是一个

Table<Product>实例。如本章所述，有许多方法都可以用于 Table<TEntity>对象。其中一些方法在表 27-4 中定义。

表 27-4

方 法	说 明
Attach	可以把一个实体关联到 DataContext 实例上
AttachAll	可以把一个实体集合关联到 DataContext 实例上
DeleteAllOnSubmit<TSubEntity>	可以把所有未决的操作置于准备删除的状态。从 DataContext 对象上调用 SubmitChanges()方法时，就执行所有的操作
DeleteOnSubmit	可以把一个未决的操作置于准备删除的状态。从 DataContext 对象上调用 SubmitChanges()方法时，就执行所有的操作
GetModifiedMembers	提供一组已修改的对象，可以访问它们的当前值和改变后的值
GetNewBindingList	提供一个新列表，以绑定到数据存储上
GetOriginalEntityState	提供一个对象的实例，且显示其初始状态
InsertAllOnSubmit<TSubEntity>	可以把所有未决的操作置于准备插入的状态。从 DataContext 对象上调用 SubmitChanges()方法时，就执行所有的操作
InsertOnSubmit	可以把一个未决的操作置于准备插入的状态。从 DataContext 对象上调用 SubmitChanges()方法时，就执行所有的操作

27.3 不使用 O/R 设计器工作

Visual Studio 2008 中的新 O/R 设计器很容易创建 LINQ to SQL 需要的所有对象，但底层的框架允许从头开始完成所有的任务。这将最大限度地控制实际的情形和发生的事件。

27.3.1 创建自己的定制对象

为了完成与前面 Customer 表相同的任务，需要通过一个类来展示 Customer 表。首先在项目中创建一个新类 Customer.cs，这个类的代码如下所示：

```
using System.Data.Linq.Mapping;

namespace ConsoleApplication1
{
    [Table(Name = "Customers")]
    public class Customer
    {
        [Column(IsPrimaryKey = true)]
        public string CustomerID { get; set; }
        [Column]
        public string CompanyName { get; set; }
        [Column]
        public string ContactName { get; set; }
        [Column]
        public string ContactTitle { get; set; }
    }
}
```



```

        [Column]
        public string Address { get; set; }
        [Column]
        public string City { get; set; }
        [Column]
        public string Region { get; set; }
        [Column]
        public string PostalCode { get; set; }
        [Column]
        public string Country { get; set; }
        [Column]
        public string Phone { get; set; }
        [Column]
        public string Fax { get; set; }
    }
}

```

`Customer.cs` 文件定义了要用于 LINQ to SQL 的 `Customer` 对象。这个类指定了 `Table` 属性，以表示表类。`Table` 类属性包含一个属性 `Name`，它定义了数据库中使用的表名，这个表名要在连接字符串中引用。使用 `Table` 属性还表示，需要在代码中建立对 `System.Data.Linq.Mapping` 命名空间的引用。

除了 `Table` 属性之外，类中定义的每个属性都使用了 `Column` 属性。如前所述，SQL Server 数据库中的列映射到代码的属性上。

### 27.3.2 用定制的对象和 LINQ 进行查询

有了 `Customer` 类后，就可以查询 Northwind 数据库中的 `Customers` 表了。完成这一任务的代码如下所示：

```

using System;
using System.Data.Linq;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
            DataContext dc = new DataContext(@"Data Source=.\SQLEXPRESS;
            AttachDbFilename=|DataDirectory|\NORTHWND.MDF;
            Integrated Security=True;User Instance=True");

            dc.Log = Console.Out; // Used for outputting the SQL used

            Table < Customer > myCustomers = dc.GetTable < Customer > ();
            foreach (Customer item in myCustomers)
            {
                Console.WriteLine("{0} | {1}",
                    item.CompanyName, item.Country);
            }
            Console.ReadLine();
        }
    }
}

```

在这个例子中，使用了默认的 `DataContext` 对象，把包含 SQL Server Express 数据库

Northwind 的连接字符串传送为一个参数。再用 `GetTable<TEntity>()` 方法填充 `Customer` 类型的 `Table` 类。这个例子中的 `GetTable<TEntity>()` 操作使用了定制的 `Customer` 类:

```
dc.GetTable < Customer >();
```

LINQ to SQL 使用 `DataContext` 对象在 SQL Server 数据库上执行查询, 把返回的行作为强类型化的 `Customer` 对象。这就允许迭代 `Table` 对象集合中的每个 `Customer` 对象, 获得需要的信息, 如下面的 `Console.WriteLine()` 语句所示:

```
foreach (Customer item in myCustomers)
{
    Console.WriteLine("{0} | {1}",
        item.CompanyName, item.Country);
}
```

运行这段代码, 会在控制台应用程序中生成如下结果:

```
SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName],
[t0].[ContactTitle], [t0].[Address], [t0].[City], [t0].[Region],
[t0].[PostalCode], [t0].[Country], [t0].[Phone], [t0].[Fax]
FROM [Customers] AS [t0]
-- Context: SqlProvider(Sql2005) Model:
AttributedMetaModel Build: 3.5.21022.8
Alfreds Futterkiste | Germany
Ana Trujillo Emparedados y helados | Mexico
Antonio Moreno Taquería | Mexico
Around the Horn | UK
Berglunds snabbköp | Sweden
// Output removed for clarity
Wartian Herkku | Finland
Wellington Importadora | Brazil
White Clover Markets | USA
Wilman Kala | Finland
Wolski Zajazd | Poland
```

### 27.3.3 用查询限制所调用的列

注意, 查询检索出了 `Customer` 类文件中指定的每个列。如果删除不需要的列, 就可以得到一个新的 `Customer` 类文件, 如下所示:

```
using System.Data.Linq.Mapping;

namespace ConsoleApplication1
{
    [Table(Name = "Customers")]
    public class Customer
    {
        [Column(IsPrimaryKey = true)]
        public string CustomerID { get; set; }
        [Column]
        public string CompanyName { get; set; }
        [Column]
        public string Country { get; set; }
    }
}
```

这里删除了应用程序未使用的所有列。现在，如果运行控制台应用程序，查看生成的 SQL 查询，结果如下：

```
SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[Country]
FROM [Customers] AS [t0]
```

可以看出，在对 Customers 表的查询中，仅使用了 Customer 类中定义的 3 列。

CustomerID 属性比较有趣，因为可以在 Column 属性中使用 IsPrimaryKey 设置，把这个列标记为表的主键。这个设置带一个布尔值，这里它设置为 true。

### 27.3.4 使用列名

列的另一个要点是在 Customer 类中定义的属性名必须与数据库中使用的名称相同。例如，如果把 CustomerID 属性名改为 MyCustomerID，运行控制台应用程序时，就会得到如下异常：

```
System.Data.SqlClient.SqlException was unhandled
  Message="Invalid column name 'MyCustomerID'."
  Source=".Net SqlClient Data Provider"
  ErrorCode=-2146232060
  Class=16
  LineNumber=1
  Number=207
  Procedure=" "
  Server="\\\\.\\pipe\\F5E22E37-1AF9-44\\tsql\\query"
```

为了改正这个错误，需要在前面创建的 Customer 定制类中定义列的名称。为此，可以使用 Column 属性，如下所示：

```
[Column(IsPrimaryKey = true, Name = "CustomerID")]
public string MyCustomerID { get; set; }
```

与 Table 属性一样，Column 属性也包含一个 Name 属性，以指定列显示在 Customers 表中的名称。

这会生成如下查询：

```
SELECT [t0].[CustomerID] AS [MyCustomerID], [t0].[CompanyName],
[t0].[Country] FROM [Customers] AS [t0]
```

这也意味着，需要使用新的名称 MyCustomerID 引用列，例如 item.MyCustomerID。

### 27.3.5 创建自己的 DataContext 对象

使用普通的 DataContext 对象有时并不是最好的方法，而创建自己的 DataContext 类可以进行更多的控制。为此，创建一个新类 MyNorthwindDataContext.cs，使这个类继承 DataContext。该类最简单的形式如下：

```
using System.Data.Linq;

namespace ConsoleApplication1
{
    public class MyNorthwindDataContext : DataContext
    {
        public Table < Customer > Customers;
```



```

public MyNorthwindDataContext()
    : base(@"Data Source=.\SQLEXPRESS;
        AttachDbFilename=|DataDirectory|\NORTHWND.MDF;
        Integrated Security=True;User Instance=True")
    {
    }
}

```

这个类 `MyNorthwindDataContext` 继承了 `DataContext`，从前面创建的 `Customer` 类中提供了 `Table<Customer>` 对象的一个实例。构造函数是这个类的另一个要求。这个构造函数使用一个基本实例来初始化对象的新实例，以引用文件(这里是对 SQL 数据库文件的连接)。

现在使用自己的 `DataContext` 对象，可以改变应用程序中的代码，如下所示：

```

using System;
using System.Data.Linq;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
            MyNorthwindDataContext dc = new MyNorthwindDataContext();
            Table < Customer > myCustomers = dc.Customers;

            foreach (Customer item in myCustomers)
            {
                Console.WriteLine("{0} | {1}",
                    item.CompanyName, item.Country);
            }
            Console.ReadLine();
        }
    }
}

```

创建了 `MyNorthwindDataContext` 对象的实例后，就可以让该类管理对数据库的连接。现在还可以通过 `dc.Customers` 语句直接访问 `Customer` 类。

#### 提示：

本章的例子都比较纯粹，因为它们都没有包含建立应用程序时通常包含的错误处理和记录功能，而只是演示本章讨论的要点。

## 27.4 定制对象和 O/R 设计器

除了在自己的.cs 文件中建立定制对象，再把该类关联到自己建立的 `DataContext` 对象上之外，还可以在 Visual Studio 2008 中使用 O/R 设计器建立自己的类文件。以这种方式使用 Visual Studio 时，Visual Studio 会创建相应的.cs 文件，O/R 设计器还提供了类文件和自己建立的所有关系的可视化表示。

在查看.dbml 文件的设计器视图时，注意工具箱中有 3 项：Class、Association 和 Inheritance。

对于这个例子，从工具箱中选择 Class 对象，把它拖放到设计界面上。这会显示一般类的图像，如图 27-7 所示。

现在可以单击 Class1 名称，把这个类重命名为 Customer。右击该名称的旁边，从打开的菜单中选择 Add | Property，可以给类文件添加属性。对于这个例子，给 Customer 类添加 3 个属性 CustomerID、CompanyName 和 Country。如果突出显示 CustomerID 属性，可以在 Visual Studio 的属性窗口中配置该属性，把 Primary Key 设置从 False 改为 True。也可以突出显示整个类，进入属性对话框，把 Source 属性改为 Customers，因为这是 Customer 对象需要使用的表名。之后，类的可视化表示就如图 27-8 所示。



图 27-7

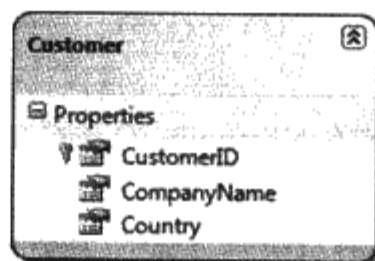


图 27-8

从图 27-8 可以看出，CustomerID 属性的名称旁边显示了主键图标。此时单击 Northwind.dbml 文件旁边的加号，就会看到两个文件 Northwind.dbml.layout 和 Northwind.designer.cs。Northwind.dbml.layout 文件是有助于 Visual Studio 在 O/R 设计器中进行可视化表示的 XML 文件。但最重要的文件是 Northwind.designer.cs，这是我们创建的 Customer 类文件。打开这个文件，会看到 Visual Studio 创建的代码。

首先，Customer 类文件在页面的代码中：

```
[Table(Name="Customers")]
public partial class Customer : INotifyPropertyChanging,
    INotifyPropertyChanged
{
    // Code removed for clarity
}
```

Customer 类是根据我们在设计器中提供的名称而指定的类名。这个类有一个 Table 属性，其值是 Customers，因为这是该对象在连接到 Northwind 数据库上时需要使用的数据库名。

在 Customer 类中，包含了前面定义的 3 个属性，这里只列出一个属性 CustomerID：

```
[Column(Storage="_CustomerID", CanBeNull=false, IsPrimaryKey=true)]
public string CustomerID
{
    get
    {
        return this._CustomerID;
    }
    set
    {
        if ((this._CustomerID != value))
        {
            this.OnCustomerIDChanging(value);
            this.SendPropertyChanging();
            this._CustomerID = value;
            this.SendPropertyChanged("CustomerID");
        }
    }
}
```

```

        this.OnCustomerIDChanged();
    }
}

```

与前面示例在建立类时一样，所定义的属性使用 `Column` 特性和这个特性提供的一些属性。例如，主键是用 `IsPrimaryKey` 项设置的。

除了 `Customer` 类之外，在创建的文件中还有一个继承自 `DataContext` 对象的类：

```

[System.Data.Linq.Mapping.DatabaseAttribute(Name="NORTHWND")]
public partial class NorthwindDataContext : System.Data.Linq.DataContext
{
    // Code removed for clarity
}

```

`DataContext` 对象 `NorthwindDataContext` 可以连接到 `Northwind` 数据库上，与前面的例子一样。

使用 O/R 设计器可以使数据库对象和类文件的创建更简单、直接。但如果要完全控制，就可以自己编写所有的代码，得到期望的结果。

## 27.5 查询数据库

如前所述，在应用程序的代码中有许多方式查询数据库。该查询最简单的形式如下：

```
Table < Product > query = dc.Products;
```

这个命令把整个 `Products` 表放在 `query` 对象实例中。

### 27.5.1 使用查询表达式

除了使用 `dc.Products` 把表直接提取出数据库之外，还可以在代码中直接使用强类型化的查询表达式。下面的代码就是一个例子：

```

using System;
using System.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main(string[] args)
        {
            NorthwindDataContext dc = new NorthwindDataContext();
            var query = from p in dc.Products
                        select p;
            foreach (Product item in query)
            {
                Console.WriteLine(item.ProductID + " | " + item.ProductName);
            }
            Console.ReadLine();
        }
    }
}

```

在这个例子中,用 `from p in dc.Products select p;`的查询值填充 `query` 对象(即 `Table<Product>` 对象)。为了便于理解,这个命令放在两行上,也可以放在一行上。

27.5.2 查询表达式

在代码中可以使用许多查询表达式,上面的例子仅是一个简单的 `select` 语句,它返回整个表。下表 27-5 列出了其他一些查询表达式。

表 27-5	
选 项	语 法
Project	<code>select &lt;expression&gt;</code>
Filter	<code>where &lt; expression &gt;, distinct</code>
Test	<code>any(&lt;expression &gt;), all(&lt;expression &gt;)</code>
Join	<code>&lt;expression&gt; join &lt;expression&gt; on &lt;expression&gt; equals &lt;expression&gt;</code>
Group	<code>group by &lt;expression&gt;, into &lt;expression&gt;, &lt;expression&gt; group join&lt;decision&gt; on &lt;expression&gt; equals &lt;expression&gt; into &lt;expression&gt;</code>
Aggregate	<code>Count([&lt;expression&gt;]), sum(&lt;expression&gt;), min(&lt;expression&gt;), max(&lt;expression&gt;), avg(&lt;expression&gt;)</code>
Patition	<code>Skip [while] &lt;expression&gt;, take [while] &lt;expression&gt;</code>
Set	<code>Union, intersect, except</code>
Order	<code>Order by &lt;expression&gt;, &lt;expression&gt; [ascending   descending]</code>

27.5.3 使用表达式过滤

除了直接查询整个表之外,还可以使用 `where` 和 `distinct` 选项过滤数据项。下面的例子就查询 `Products` 表中指定类型的记录:

```
var query = from p in dc.Products
            where p.ProductName.StartsWith("L")
            select p;
```

这个查询从 `Products` 表中选择所有以字母 `L` 开头的记录,这是通过 `where p.ProductName.StartsWith("L")`表达式实现的。`ProductName` 属性有许多选择方法,可以细调需要的过滤条件。这个操作生成如下结果:

```
65 | Louisiana Fiery Hot Pepper Sauce
66 | Louisiana Hot Spiced Okra
67 | Laughing Lumberjack Lager
74 | Longlife Tofu
76 | Lakkalik      ri
```

还可以给列表添加任意多个表达式。例如,下面的例子给查询添加了两个 `where` 语句:

```
var query = from p in dc.Products
            where p.ProductName.StartsWith("L")
```

```
where p.ProductName.EndsWith("i")
select p;
```

其中一个过滤表达式查找产品名称以字母 L 开头的数据项，第二个表达式确保还应用了第二个条件，即数据项必须以字母 i 结尾。这会生成如下结果：

76 | Lakkalikööri

27.5.4 连接

除了操作一个表之外，还可以操作多个表，用查询进行连接。如果把 Customers 和 Orders 表都拖放到 Northwind.dbml 设计界面上，就会得到如图 27-9 所示的结果：

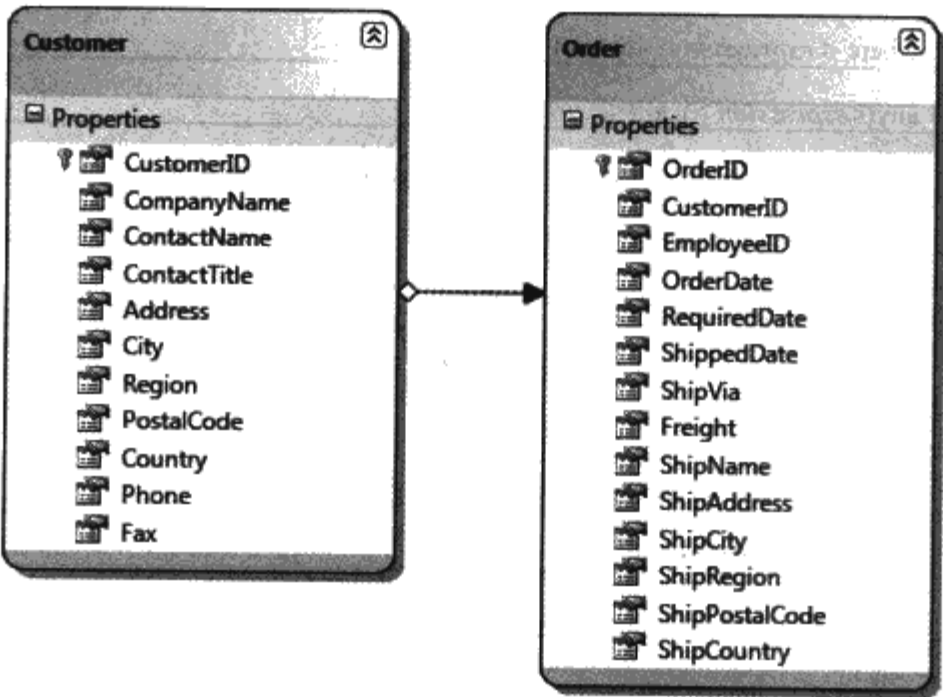


图 27-9

从图 27-9 可以看出，把这些元素拖放到设计界面上后，Visual Studio 就知道这些元素直接存在一个关系，并在代码中创建这个关系，用黑色箭头来表示。

现在，就可以在查询中使用 join 语句操作这两个表了，如下面的例子所示：

```
using System;
using System.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main(string[] args)
        {
            NorthwindDataContext dc = new NorthwindDataContext();
            dc.Log = Console.Out;

            var query = from c in dc.Customers
                        join o in dc.Orders on c.CustomerID equals o.CustomerID
                        orderby c.CustomerID
                        select new { c.CustomerID, c.CompanyName,
                                    c.Country, o.OrderID, o.OrderDate };
        }
    }
}
```



```

foreach (var item in query)
{
    Console.WriteLine(item.CustomerID + " | " + item.CompanyName
        + " | " + item.Country + " | " + item.OrderID
        + " | " + item.OrderDate);
}
Console.ReadLine();
}
}

```

这个例子从 Customers 表中提取数据，并连接 Orders 表中与 CustomerID 匹配的记录。这是通过 join 语句实现的：

```
join o in dc.Orders on c.CustomerID equals o.CustomerID
```

接着用 select new 语句创建一个新对象，这个新对象包含 Customers 表中的 CustomerID、CompanyName、Country 列，以及 Orders 表中的 OrderID 和 OrderDate 列。

在迭代这个新对象的集合时，foreach 语句还使用了 var 关键字，因为类型在此刻是未知的：

```

foreach (var item in query)
{
    Console.WriteLine(item.CustomerID + " | " + item.CompanyName
        + " | " + item.Country + " | " + item.OrderID
        + " | " + item.OrderDate);
}

```

无论如何，item 对象可以访问我们指定的所有属性。运行这个类型，会得到如下结果：

```

WILMK | Wilman Kala | Finland | 10695 | 10/7/1997 12:00:00 AM
WILMK | Wilman Kala | Finland | 10615 | 7/30/1997 12:00:00 AM
WILMK | Wilman Kala | Finland | 10673 | 9/18/1997 12:00:00 AM
WILMK | Wilman Kala | Finland | 11005 | 4/7/1998 12:00:00 AM
WILMK | Wilman Kala | Finland | 10879 | 2/10/1998 12:00:00 AM
WILMK | Wilman Kala | Finland | 10873 | 2/6/1998 12:00:00 AM
WILMK | Wilman Kala | Finland | 10910 | 2/26/1998 12:00:00 AM

```

### 27.5.5 组合数据项

也可以通过查询组合数据项。在 Northwind.dbml 示例中，把 Categories 表拖放到设计界面上，在这个表和 Products 表之间存在一个关系。下面的例子说明了如何按类别组合产品：

```

using System;
using System.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main(string[] args)
        {
            NorthwindDataContext dc = new NorthwindDataContext();
            var query = from p in dc.Products
                orderby p.Category.CategoryName ascending
                group p by p.Category.CategoryName into g
                select new { Category = g.Key, Products = g };
            foreach (var item in query)
            {

```

```

        Console.WriteLine(item.Category);
        foreach (var innerItem in item.Products)
        {
            Console.WriteLine(" " + innerItem.ProductName);
        }
        Console.WriteLine();
    }
    Console.ReadLine();
}
}

```

这个例子创建了一个新对象，它是一组类别，再把整个 Product 表打包到这个新表 g 中。在此之前，类别使用 orderby 语句按名称排序，因为所提供的订单是按升序排列的(另一个选项是降序)。其输出是 Category(通过 Key 属性传送)和 Product 实例。foreach 语句给类别迭代一次，给在类别中找到的每个产品迭代一次。

这个程序的部分结果如下：

```

Beverages
  Chai
  Chang
  Guaraná Fantástica
  Sasquatch Ale
  Steeleye Stout
  Côte de Blaye
  Chartreuse verte
  Ipoh Coffee
  Laughing Lumberjack Lager
  Outback Lager
  Rhönbräu Klosterbier
  Lakkalikööri

Condiments
  Aniseed Syrup
  Chef Anton's Cajun Seasoning
  Chef Anton's Gumbo Mix
  Grandma's Boysenberry Spread
  Northwoods Cranberry Sauce
  Genen Shouyu
  Gula Malacca
  Sirop d'érable
  Vegie-spread
  Louisiana Fiery Hot Pepper Sauce
  Louisiana Hot Spiced Okra
  Original Frankfurter grüne Soße

```

除了本章介绍的命令和表达式之外，还有许多其他的命令和表达式。

## 27.6 存储过程

前面都是直接查询表，让 LINQ 为操作创建相应的 SQL 语句。在使用包含许多存储过程的已有数据库和遵循使用存储过程的最佳实践规范的数据库时，LINQ 也是一个可行的选项。

LINQ to SQL 把存储过程看作方法调用。如图 27-4 所示，O/R 设计器可以把表拖放到其设

计界面上，之后就可以编程处理表了。在 O/R 设计器的右侧有一个区域，可以拖放存储过程。

拖放到 O/R 设计器这个部分的所有存储过程都变成可以在 DataContext 对象中使用的方法。例如，把 TenMostExpensiveProducts 存储过程拖放到 O/R 设计器的这个部分上。

下面的例子说明了如何在 Northwind 数据库中调用这个存储过程：

```
using System;
using System.Collections.Generic;
using System.Data.Linq;
using System.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main(string[] args)
        {
            NorthwindDataContext dc = new NorthwindDataContext();
            ISingleResult< Ten_Most_Expensive_ProductsResult > result =
                dc.Ten_Most_Expensive_Products();
            foreach (Ten_Most_Expensive_ProductsResult item in result)
            {
                Console.WriteLine(item.TenMostExpensiveProducts + " | " +
                    item.UnitPrice);
            }
            Console.ReadLine();
        }
    }
}
```

在这个例子中，存储过程输出的数据行放在 ISingleResult<Ten\_Most\_Expensive\_ProductsResult>对象中。之后，迭代这个对象，这与前面的例子一样简单。

从这个例子可以看出，调用存储过程是很简单的。

## 27.7 小结

.NET Framework 3.5 的一个杰出的特性是该平台提供的 LINQ 功能。本章介绍了 LINQ to SQL 的使用和查询 SQL Server 数据库时可以使用的一些选项。

使用 LINQ to SQL 可以通过一组强类型化的操作对数据库执行 CRUD 操作。也可以使用已有的访问功能与 ADO.NET 交互操作，或使用存储过程。

下一章介绍 XML 的处理，为第 29 章做准备。

# 第28章

## 处 理 XML

XML 在 .NET Framework 中有着重要的作用。 .NET Framework 不仅允许在应用程序中使用 XML, .NET Framework 本身也在配置文件和源代码文档中使用 XML。另外, SOAP、Web 服务和 ADO.NET 也使用 XML。

为了扩展使用 XML, .NET Framework 包含了 System.Xml 命名空间。这个命名空间带有许多处理 XML 的类。本章将讨论这些类。

本章介绍如何使用 XmlDocument(这是 DOM 的实现方式)类, 以及 .NET 为 SAX 提供的一种替代品(XmlReader 和 XmlWriter 类)。还要讨论 XPath 和 XSLT 的类实现方式。接着介绍 XML 和 ADO.NET 如何一起工作, 如何把其中一种格式转换为另一种格式。还介绍了如何把对象串行化为 XML, 使用 System.Xml.Serialization 命名空间中的类从 XML 文档中创建一个对象(并行化)。更重要的是, 要介绍如何把 XML 合并到 C# 应用程序中。

注意 XML 命名空间可以用许多不同的方式得到类似的结果。我们不可能把这些方式都放在一章中介绍, 所以这里仅介绍其中一种方式, 并提及完成同一任务的其他方式。

因为篇幅有限, 不能从头开始介绍 XML, 所以本章假定读者已经熟悉了 XML 技术。因此, 您应知道元素、属性和节点, 还应知道格式规范的文档的含义, 您也应熟悉 SAX 和 DOM。如果要更多地了解 XML, 可以参阅 *Beginning XML*(Wiley 出版社, ISBN 0-7645-7077-3)。

本章主要内容如下:

- XML 标准
- XmlReader 和 XmlWriter
- XmlDocument
- XPathDocument
- XmlNavigator

首先介绍目前使用的 XML 标准。

### 28.1 .NET 支持的 XML 标准

World Wide Web Consortium (W3C) 开发了一组标准, 给 XML 提供了强大的功能和潜力。没有这些标准, XML 不会对开发过程有影响。W3C 网站(<http://www.w3.org>)包含了 XML 的所有信息。

.NET Framework 支持下述 W3C 标准:

- XML 1.0([www.w3.org/TR/1998/REC-xml-19980210](http://www.w3.org/TR/1998/REC-xml-19980210)): 包括 DTD 支持
- XML 命名空间([www.w3.org/TR/REC-xml-names](http://www.w3.org/TR/REC-xml-names)): 包括流级和 DOM
- XML 模式([www.w3.org/2001/XMLSchema](http://www.w3.org/2001/XMLSchema))
- XPath 表达式(<http://www.w3.org/TR/xpath>)
- XSLT 转换(<http://www.w3.org/TR/xslt>)
- DOM Level 1 核心(<http://www.w3.org/TR/REC-DOM-Level-1/>)
- DOM Level 2 核心(<http://www.w3.org/TR/DOM-Level-2-Core/>)
- Soap 1.1(<http://www.w3.org/TR/SOAP>)

随着 Framework 走向成熟、W3C 更新所推荐的标准，标准支持的级别也会改变，因此，必须确保标准和 Microsoft 提供的支持级别都是最新的。

## 28.2 System.Xml 命名空间

对 XML 处理的支持是由.NET 的 System.Xml 命名空间中的类提供的。下面看看(没有特定的顺序)System.Xml 命名空间中的一些比较重要的类。表 28-1 列出了主要的 XML 读取器和写入器类。

表 28-1

类 名	说 明
XmlReader	抽象的读取器类，提供快速、没有缓存的 XML 数据。XmlReader 是只向前的，类似于 SAX 分析器
XmlWriter	抽象的写入器类，以流或文件的格式提供快速、没有缓存的 XML 数据
XmlTextReader	扩展 XmlReader，提供访问 XML 数据的快速只向前流
XmlTextWriter	扩展 XmlWriter，快速生成只向前的 XML 流

表 28-2 列出了用于处理 XML 的其他一些重要的类。

表 28-2

类 名	说 明
XmlNode	抽象类，表示 XML 文档中的一个节点。是 XML 命名空间中几个类的基类
XmlDocument	扩展 XmlNode，这是 W3C DOM 的实现方式，给出 XML 文档在内存中的树形表示，可以浏览和编辑它们
XmlDataDocument	扩展 XmlDocument，即从 XML 数据中加载的文档，或从 ADO.NET DataSet 的关系数据中加载的文档，允许把 XML 和关系数据混合在同一个视图中
XmlResolver	抽象类，分析基于 XML 的外部资源，例如 DTD 和模式引用，也可以用于处理 <xsl:include>和 <xsl:import>元素
XmlNodeList	可以迭代的一组 XmlNode
XmlUrlResolver	扩展 XmlResolver，用 URI(Uniform Resource Identifier)解析外部资源

System.Xml 命名空间中的许多类都提供了管理 XML 文档和流的方式，而其他类(例如



XmlDataDocument 类)则提供了 XML 数据库和存储在 DataSet 中的关系数据之间的桥梁。

注意:

XML 命名空间可用于 .NET 的任何语言,这表示,本章中所有的示例也可以用 VB.NET、managed C++ 等来编写。

## 28.3 使用 System.Xml 类

下面几个示例将使用 books.xml 作为数据源。books.xml 可以从 Wrox 网站([www.wrox.com](http://www.wrox.com))上下载,它也包含在 .NET SDK 的几个示例中。books.xml 文件是假想书店的书目清单,包含类型、作者姓名、价格和 ISBN 号码等信息。与其他章节一样,本章中的所有代码示例也可以在 Wrox 网站上得到。

下面是 books.xml 文件:

```
<?xml version='1.0'?>
<!-- This file represents a fragment of a book store inventory database -->
<bookstore>
  <book genre="autobiography" publicationdate="1981" ISBN="1-861003-28-0">
    <title>The Autobiography of Benjamin Franklin</title>
    <author>
      <first-name>Benjamin</first-name>
      <last-name>Franklin</last-name>
    </author>
    <price>8.99</price>
  </book>
  <book genre="novel" publicationdate="1967" ISBN="0-201-63361-2">
    <title>The Confidence Man</title>
    <author>
      <first-name>Herman</first-name>
      <last-name>Melville</last-name>
    </author>
    <price>11.99</price>
  </book>
  <book genre="philosophy" publicationdate="1991" ISBN="1-861001-57-6">
    <title>The Gorgias</title>
    <author>
      <name>Plato</name>
    </author>
    <price>9.99</price>
  </book>
</bookstore>
```

## 28.4 读写流格式的 XML

如果您曾经使用过 SAX,就应很熟悉 XmlReader 和 XmlWriter 类。基于 XmlReader 的类提供了一种非常迅速、只向前的只读光标来处理 XML 数据。它是一个流模型,所以内存要求不是很高。但是,它没有提供基于 DOM 模型的浏览功能和读写功能。基于 XmlWriter 的类可以生成遵循 W3C 的 XML 1.0 Namespace Recommendations 的 XML 文档。

XmlReader 和 XmlWriter 都是抽象类。下面的类派生于 XmlReader:

- XmlNodeReader
- XmlTextReader
- XmlValidatingReader

下面的类派生于 XmlWriter 的类:

- XmlTextWriter
- XmlQueryWriter

XmlTextReader 和 XmlTextWriter 与 System.IO 命名空间中一个基于流的对象或与 TextReader/TextWriter 对象一起使用。XmlNodeReader 把 XmlNode 作为其源,而不是一个流。XmlValidatingReader 添加了 DTD 和模式验证,因此提供了数据的有效性验证。本章的后面会详细介绍这些类。

### 28.4.1 使用 XmlReader 类

XmlReader 非常类似于 MSXML SDK 中的 SAX。它们最大的一个区别是 SAX 是一种推模型(push model),它把数据推入应用程序中,开发人员必须接受它,而 XmlReader 是一种拉模型,把应用程序请求的数据拉入该应用程序。这样编程就有一种更简单、更直观的模式。另一个优点是拉模型(pull model)可以选择把什么数据传送到应用程序中。如果不需要所有的数据,就不需要处理它们。而在推模型中,所有的 XML 数据都必须由应用程序处理,无论是否需要这些数据。

下面介绍一个非常简单的示例,读取 XML 数据,后面将详细介绍 XmlReader 类,这些代码在 XmlReaderSample 文件夹中。下面的代码将读取 book.xml 文档中的数据。在读取每个节点时,都要检查 NodeType 属性。如果节点是一个文本节点,就把其值追加到文本框中:

```
using System.Xml;

private void button3_Click(object sender, EventArgs e)
{
    richTextBox1.Clear();
    XmlReader rdr = XmlReader.Create("books.xml");
    while (rdr.Read())
    {
        if (rdr.NodeType == XmlNodeType.Text)
            richTextBox1.AppendText(rdr.Value + "\r\n");
    }
}
```

如前所述,XmlReader 是一个抽象类,所以,要直接使用 XmlReader 类,必须添加静态方法 Create,该方法返回一个 XmlReader 对象。Create 方法有 9 个重载版本。在上面的例子中,该方法有一个字符串参数,表示 XmlDocument 的文件名。还可以给该方法传送基于流和基于 TextReader 的对象。

另一个可以使用的对象是 XmlReaderSettings,它指定了读取器的特性。例如,可以使用模式来验证数据流。把 Schemas 属性设置为一个有效的 XMLSchemaSet 对象,来缓存 XSD 模式。接着把 XmlReaderSettings 对象的 XsdValidate 属性设置为 true。

有几个 Ignore 属性可用于控制读取器处理某些节点和值的方式。这些属性包括 IgnoreComments、IgnoreIdentityConstraints、IgnoreInlineSchema、IgnoreProcessingInstructions、

IgnoreSchemaLocation 和 IgnoreWhitespace, 它们可以从文档中提取某些项。

### 1. Read 方法

遍历文档有几种方式, 如前面的示例所示, Read()可以进入下一个节点。然后查看该节点是否有一个值(HasValue())、该节点是否有属性(HasAttributes())。也可以使用 ReadStartElement()方法, 查看当前节点是否是起始元素, 如果是起始元素, 就可以定位到下一个节点上。如果不是起始元素, 就引发一个 XmlException。调用这个方法与调用 Read()后再调用 IsStartElement 是一样的。

ReadElementString()类似于 ReadString(), 但可以把元素名作为参数。如果下一个 Content 节点不是起始标记, 或者 Name 属性不匹配当前的节点 Name, 就会引发异常。

下面的示例说明了如何使用 ReadElementString()。注意这个示例使用 FileStream, 所以需要利用 using 语句来包括 System.IO 命名空间:

```
private void button6_Click(object sender, EventArgs e)
{
    richTextBox1.Clear();
    XmlReader rdr = XmlReader.Create("books.xml");
    while (!rdr.EOF)
    {
        //if we hit an element type, try and load it in the listbox
        if (rdr.MoveToContent() == XmlNodeType.Element && rdr.Name == "title")
        {
            richTextBox1.AppendText(rdr.ReadElementString() + "\r\n");
        }
        else
        {
            //otherwise move on
            rdr.Read();
        }
    }
}
```

在 while 循环中, 使用 MoveToContent 查找类型为 XmlNodeType.Element、名称为 title 的节点。我们使用 XmlTextReader 的 EOF 属性作为循环条件。如果节点的类型不是 Element, 或者名称不是 title, else 子句就会调用 Read()方法进入下一个节点。查找到一个满足条件的节点后, 就把 ReadElementString()的结果添加到列表框中。这样就在 listbox 中添加一个书名。注意, 在成功执行了 ReadElementString()后, 不需要调用 Read()方法, 这是因为 ReadElementString()已经查看了整个 Element, 然后定位到下一个节点上。

如果删除了 if 子句中的 && rdr.Name=="title", 在抛出 XmlException 异常时, 就必须捕获它。如果查看一下数据文件, 就会发现 MoveToContent()查找到的第一个元素是<bookstore>, 因为它是一个元素, 所以通过了 if 语句中的检查。但是, 它不包含简单的文本类型, 因此会让 ReadElementString()引发一个 XmlException 异常。解决这个问题的一种方式是把 ReadElementString()调用放在它自己的函数中。现在, 如果在这个函数中 ReadElementString()调用失败, 就可以处理错误, 返回给调用函数。

下面就调用这个新方法 LoadTextBox(), 把 XmlTextReader 作为参数。进行了这些修改后, LoadTextBox()方法如下所示:

```
private void LoadTextBox(XmlReader reader)
```



```

{
    try
    {
        richTextBox1.AppendText (reader.ReadElementString() + "\r\n");
    }
    // if an XmlException is raised, ignore it.
    catch(XmlException er){}
}

```

上面示例中的下述代码

```

if (tr.MoveToContent() == XmlNodeType.Element && tr.Name == "title")
{
    richTextBox1.AppendText (tr.ReadElementString() + "\r\n");
}
else
{
    //otherwise move on
    tr.Read();
}

```

就会变成:

```

if (tr.MoveToContent() == XmlNodeType.Element)
{
    LoadTextBox(tr);
}
else
{
    //otherwise move on
    tr.Read();
}

```

运行这段代码, 结果应与前面示例的结果是一样的。因此, 完成这个任务有多种不同的方式。这体现了 System.Xml 命名空间中类的灵活性。

XmlReader 类还可以读取强类型化的数据, 它有几个 ReadElementContent 方法, 例如 ReadElementContentDouble、ReadElementContentBoolean 等。下面的示例说明了如何把值读取为小数, 并对该值进行数学处理。在本例中, 要给价格元素中的值增加 25%:

```

private void button5_Click(object sender, EventArgs e)
{
    richTextBox1.Clear();
    XmlReader rdr = XmlReader.Create("books.xml");
    while (rdr.Read())
    {
        if (rdr.NodeType == XmlNodeType.Element)
        {
            if (rdr.Name == "price")
            {
                decimal price = rdr.ReadElementContentDecimal();
                richTextBox1.AppendText("Current Price = " + price + "\r\n");
                price += price * (decimal).25;
                richTextBox1.AppendText("New Price = " + price + "\r\n\r\n");
            }
            else if(rdr.Name=="title")
                richTextBox1.AppendText(rdr.ReadElementContentString() + "\r\n");
        }
    }
}

```

如果不能把值转换为小数，就生成一个 `FormatException` 异常。与把值读取为一个字符串，再把它转换为合适的数据类型相比，这个方法的效率较高。

## 2. 检索属性数据

在运行示例代码时，注意在读取节点时，没有看到属性。这是因为属性不是文档结构的一部分。针对元素节点，可以检查属性是否存在，并可检索属性值。

例如，如果有属性，`HasAttributes` 就返回 `true`，否则就返回 `false`。`AttributeCount` 属性确定属性的个数。`GetAttribute()`方法按照名称或索引来获取属性。如果要一次迭代一个属性，就可以使用 `MoveToFirstAttribute()`和 `MoveToNextAttribute()`方法。

下面的示例迭代 `book.xml` 文档中的属性：

```
private void button7_Click(object sender, EventArgs e)
{
    richTextBox1.Clear();
    XmlReader tr = XmlReader.Create("books.xml");
    //Read in node at a time
    while (tr.Read())
    {
        //check to see if it's a NodeType element
        if (tr.NodeType == XmlNodeType.Element)
        {
            //if it's an element, then let's look at the attributes.
            for (int i = 0; i < tr.AttributeCount; i++)
            {
                richTextBox1.AppendText(tr.GetAttribute(i) + "\r\n");
            }
        }
    }
}
```

这次查找元素节点。找到一个节点后，就迭代其所有的属性，使用 `GetAttribute()`方法把属性值加载到列表框中。在本例中，这些属性是 `genre`、`publicationdate` 和 `ISBN`。

### 28.4.2 使用 `XmlReader` 类进行验证

有时不但要知道文档的格式是规范的，还要确定文档是有效的。`XmlReader` 可以使用 `XmlReaderSettings` 类，根据 XSD 模式验证 XML。XSD 模式添加到 `XMLSchemaSet` 中，通过 `Schemas` 属性可以访问 `XMLSchemaSet.XsdValidate` 属性还必须设置为 `true`，这个属性默认为 `false`。

下面的示例演示了 `XmlReaderSettings` 类的用法。这个 XSD 模式用于验证 `book.xml` 文档：

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema attributeFormDefault="unqualified"
    elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="bookstore">
        <xs:complexType>
            <xs:sequence>
                <xs:element maxOccurs="unbounded" name="book">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="title" type="xs:string" />
                            <xs:element name="author">
```



```

        <xs:complexType>
            <xs:sequence>
                <xs:element minOccurs="0" name="name" type="xs:string" />
                <xs:element minOccurs="0" name="first-name" type="xs:string" />
                <xs:element minOccurs="0" name="last-name" type="xs:string" />
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="price" type="xs:decimal" />
</xs:sequence>
<xs:attribute name="genre" type="xs:string" use="required" />
<!-- <xs:attribute name="publicationdate"
        type="xs:unsignedShort" use="required" /> -->
<xs:attribute name="ISBN" type="xs:string" use="required" />
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>

```

这个模式在 Visual Studio 中从 book.xml 中生成。注意 publicationdate 属性被注释掉了，这样在验证到该属性时会失败。

下面的代码使用该模式验证 books.xml 文档：

```

private void button8_Click(object sender, EventArgs e)
{
    richTextBox1.Clear();

    XmlReaderSettings settings = new XmlReaderSettings();
    settings.Schemas.Add(null, "books.xsd");
    settings.XsdValidate = true;
    settings.ValidationEventHandler +=
        new System.Xml.Schema.ValidationEventHandler(settings_ValidationEventHandler);
    XmlReader rdr = XmlReader.Create("books.xml", settings);
    while (rdr.Read())
    {
        if (rdr.NodeType == XmlNodeType.Text)
            richTextBox1.AppendText(rdr.Value + "\r\n");
    }
}

```

创建好 XmlReaderSettings 对象后，就把模式 books.xsd 添加到 XmlSchemaSet 对象中。XmlSchemaSet 的 Add 方法有 4 个重载版本，第一个重载版本把 XmlSchema 对象作为参数，XmlSchema 对象可以用于创建模式，而无需在磁盘上创建模式文件。另一个重载版本把另一个 XmlSchemaSet 对象作为参数。第三个重载版本带两个字符串参数。第一个字符串是目标命名空间，第二个字符串是 XSD 文档的 URL。如果目标命名空间参数为空，就使用模式的 targetNamespace。最后一个重载版本也把 targetNamespace 作为第一个参数，但使用基于 XmlReader 对象读取模式。XmlSchemaSet 在处理要验证的文档之前预处理模式。

引用了模式后，XsdValidate 属性就设置为 ValidationType 枚举的一个值，该枚举的有效值是 DTD、Schema 和 None。如果选中的值设置为 None，就不进行验证。

我们使用的是 XmlReader 对象，所以，如果文档中有验证问题，在读取器读取属性或元素之前，该问题是检查不出来的。验证失败时，会生成一个 XmlSchemaValidationException。这个异常可以在 catch 块中处理，但处理异常会很难控制数据流。为了解决这个问题，可以使用

XmlReaderSettings 类中的 ValidationEvent。这样，就可以处理验证失败，且无需使用异常处理程序。该事件还可以由验证警告引发，验证警告不会生成异常。ValidationEvent 传送一个 ValidationEventArgs 对象，该对象包含 Severity 属性。这个属性确定事件是由错误还是警告引发。如果事件是由错误引发，还会传送引发事件的异常。还有一个消息属性。在本例中，消息显示在 MessageBox 中。

### 28.4.3 使用 XmlWriter 类

XmlWriter 类可以把 XML 写入一个流、文件、StringBuilder、TextWriter 或另一个 XmlWriter 对象中。与 XmlReader 一样，XmlWriter 类以只向前、未缓存的方式进行写入。XmlWriter 的可配置性很高，可以指定是否缩进文本、缩进量、在属性值中使用什么引号以及是否支持命名空间等信息。与 XmlReader 一样，这个配置使用 XmlWriterSettings 对象进行。

下面是一个简单的示例，说明了如何使用 XmlWriter 类：

```
private void button9_Click(object sender, EventArgs e)
{
    XmlWriterSettings settings = new XmlWriterSettings();
    settings.Indent = true;
    settings.NewLineOnAttributes = true;
    XmlWriter writer = XmlWriter.Create("booknew.xml", settings);
    writer.WriteStartDocument();
    //Start creating elements and attributes
    writer.WriteStartElement("book");
    writer.WriteAttributeString("genre", "Mystery");
    writer.WriteAttributeString("publicationdate", "2001");
    writer.WriteAttributeString("ISBN", "123456789");
    writer.WriteElementString("title", "Case of the Missing Cookie");
    writer.WriteStartElement("author");
    writer.WriteElementString("name", "Cookie Monster");
    writer.WriteEndElement();
    writer.WriteElementString("price", "9.99");
    writer.WriteEndElement();
    writer.WriteEndDocument();
    //clean up
    writer.Flush();
    writer.Close();
}
```

这里编写一个新的 XML 文件 booknew.xml，并给一本新书添加数据。注意 XmlWriter 会用新文件覆盖旧文件。本章的后面会把一个新元素或节点插入到现有的文档中，使用 Create 静态方法实例化 XmlWriter 对象。在本例中，把一个表示文件名的字符串和 XmlWriterSettings 类的一个实例传送为参数。

XmlWriterSettings 类的属性控制生成 XML 的方式。CheckedCharacters 属性是一个布尔值，如果 XML 中的字符不遵循 W3C XML 1.0 规范，该属性就会引发一个异常。Encoding 类设置生成 XML 所使用的编码，默认为 Encoding.UTF8。Indent 属性是一个布尔值，确定元素是否应缩进。IndentChars 属性设置为用于缩进的字符串，默认为两个空格。NewLine 属性可以确定换行符。在上面的示例中，NewLineOnAttribute 设置为 true，所以把每个属性单独放在一行上，更便于读取生成的 XML。

WriteStartDocument() 添加文档声明。现在开始写入数据。首先是 book 元素，添加 genre、

publicationdate 和 ISBN 属性。然后写入 title、author 和 price 元素。注意 author 元素有一个子元素 name。

单击按钮，生成 booknew.xml 文件：

```
<?xml version="1.0" encoding="utf-8"?>
<book
  genre="Mystery"
  publicationdate="2001"
  ISBN="123456789">
  <title>Case of the Missing Cookie</title>
  <author>
    <name>Cookie Monster</name>
  </author>
  <price>9.99</price>
</book>
```

在开始和结束写入元素和属性时，要注意控制元素的嵌套。在给 authors 元素添加 name 子元素时，就可以看到这种嵌套。注意 WriteStartElement()和 WriteEndElement()方法调用是如何安排的，以及它们是如何在输出文件中生成嵌套的元素的。

除了 WriteElementString()和 WriteAttributeString()方法外，还有其他几个专用的写入方法。WriteCData()可以输出一个 CData 部分(<![CDATA[...]]>)，把要写入的文本作为一个参数。WriteComment()以正确的 XML 格式写入注释。WriteChars()写入字符缓冲区的内容，其工作方式类似于前面的 ReadChars()，它们都使用相同类型的参数。WriteChar()需要一个缓冲区(一个字符数组)、写入的起始位置(一个整数)和要写入的字符个数(一个整数)。

使用基于 XmlReader 和 XmlWriter 的类读写 XML 是非常灵活的，使用起来也很简单。下面介绍如何使用 System.Xml 命名空间中的 XmlDocument 和 XmlNode 类执行 DOM。

### 28.5 在.NET 中使用 DOM

.NET 中的文档对象模型(Document Object Model, DOM)支持 W3C DOM Level 1 和 Core DOM Level 2 规范。DOM 是通过 XmlNode 类来实现的。XmlNode 是一个抽象类，它表示 XML 文档的一个节点。

还有一个 XmlNodeList 类，它是节点的一个有序列表。这是一个活动的节点列表，对节点的任何修改都会立即反映到列表中。XmlNodeList 支持索引访问或迭代访问。

XmlNode 和 XmlNodeList 类组成了.NET Framework 中 DOM 的核心，表 28-3 是基于 XmlNode 的一些类。

表 28-3

类 名	说 明
XmlLinkedNode	返回当前节点之前或之后的节点。给 XmlNode 添加 NextSibling 和 PreviousSibling 属性
XmlDocument	表示整个文档，执行 DOM Level 1 和 Level 2 规范
XmlDocumentFragment	表示文档树的一个片段
XmlAttribute	表示 XmlElement 对象的一个属性对象

(续表)

类 名	说 明
XmlEntity	表示一个已分析或未分析的实体节点
XmlNotation	包含在 DTD 或模式中声明的记号

表 28-4 中的类扩展了 XmlCharacterData。

表 28-4

类 名	说 明
XmlCDATASection	表示文档中的一个 CDATA 部分
XmlComment	表示一个 XML 注释对象
XmlSignificantWhitespace	表示带有空白的节点。只有 PreserveWhiteSpace 标志为 true 时，才能创建节点
XmlWhitespace	表示元素内容中的空白格，只有 PreserveWhiteSpace 标志为 true 时，才能创建节点
XmlText	表示元素或属性的文本内容。

最后，表 28-5 的类扩展了 XmlLinkedNode。

表 28-5

类 名	说 明
XmlDeclaration	表示声明节点 (<?xml version='1.0'...>)
XmlDocumentType	表示与文档类型声明相关的数据
XmlElement	表示一个 XML 元素对象
XmlEntityReferenceNode	表示一个实体引用节点
XmlProcessingInstruction	包含 XML 处理指令

可以看出，.NET 使其类适合于任何 XML 类型。因此，该工具集是非常灵活和强大的。我们打算详细介绍每个类，而是用几个示例来说明可以完成什么任务。

使用 XmlDocument 对象

XmlDocument 及其派生类 XmlDataDocument(详见本章后面的内容)是用于在.NET 中表示 DOM 的类。与 XmlReader 和 XmlWriter 不同，XmlDocument 具有读写功能，并可以随机访问 DOM 树。XmlDocument 类似于 MSXML 中的 DOM 执行方式。如果您用 MSXML 编过程序，就会觉得使用 XmlDocument 很合适。

下面介绍的示例创建一个 XmlDocument 对象，加载磁盘上的一个文档，再从标题元素中加载带有数据的列表框，这类似于 XmlReader 一节的示例，区别是本例选择要使用的节点，而不是像 XmlReader 示例那样浏览整个文档。

下面是该示例的代码，与 XmlReader 示例相比，这个示例是比较简单的：

```
private void button1_Click(object sender, System.EventArgs e)
```



```

{
    // doc is declared at the module level
    // change path to match your path structure
    doc.Load("books.xml");
    // get only the nodes that we want
    XmlNodeList nodeList = _doc.GetElementsByTagName("title");
    // iterate through the XmlNodeList
    textBox1.Text = "";
    foreach(XmlNode node in nodeList)
        textBox1.Text += node.OuterXml + "\r\n";
}

```

注意，我们在本节的示例中添加了模块级的声明：

```
private XmlDocument doc=new XmlDocument();
```

如果这就是我们需要完成的工作，使用 `XmlReader` 加载列表框就是加载文本框的一种非常高效的方式，原因是我们只浏览一次文档，就完成了处理。这就是 `XmlReader` 的工作方式。但如果要重新查看某个节点，最好使用 `XmlDocument`。

下面的示例使用 `XPath` 语法从文档中检索一组节点：

```

private void button2_Click(object sender, EventArgs e)
{
    //doc is declared at the module level
    //change path to math your path structure
    doc.Load("books.xml");
    //get only the nodes that we want.
    XmlNodeList nodeList = _doc.SelectNodes("/bookstore/book/title");
    textBox1.Text = "";
    //iterate through the XmlNodeList
    foreach (XmlNode node in nodeList)
    {
        textBox1.Text += node.OuterXml + "\r\n";
    }
}

```

`SelectNodes()` 返回 `NodeList` 或一个 `XmlNodes` 集合。这个列表只包含匹配传送为 `SelectNodes` 参数的 `XPath` 语句。在这个示例中，只需查看 `title` 节点。如果调用了 `SelectSingleNode`，就会接收到一个节点对象，它包含 `XmlDocument` 中满足 `XPath` 条件的第一个节点。

下面简要介绍一下 `SelectSingleNode()` 方法，它是 `XmlDocument` 类的 `Xpath` 实现方式，`SelectSingleNode()` 和 `SelectNodes()` 都是在 `XmlNode` 中定义的，而 `XmlDocument` 是基于 `XmlNode` 的。`SelectSingleNode()` 返回一个 `XmlNode`，`SelectNodes()` 返回一个 `XmlNodeList`。`System.Xml.XPath` 命名空间包含许多 `Xpath` 实现方式。后面的一节会介绍它们。

### 插入节点

前面的示例使用 `XmlTextWriter` 创建一个新文档。其局限性是不能把节点插入到当前文档中。而使用 `XmlDocument` 类可以做到这一点。把上一个示例中的 `button1_Click()` 事件处理程序作如下改动(在下载代码的 `DOMSample3` 中)：

```

private void button4_Click(object sender, System.EventArgs e)
{
    //change path to match your structure

```



```

doc.Load("books.xml");
//create a new 'book' element
XmlElement newBook=doc.CreateElement("book");
//set some attributes
newBook.SetAttribute("genre","Mystery");
newBook.SetAttribute("publicationdate","2001");
newBook.SetAttribute("ISBN","123456789");
//create a new 'title' element
XmlElement newTitle=doc.CreateElement("title");
newTitle.InnerText="The Case of the Missing Cookie";
newBook.AppendChild(newTitle);
//create new author element
XmlElement newAuthor=doc.CreateElement("author");
newBook.AppendChild(newAuthor);
//create new name element
XmlElement newName=doc.CreateElement("name");
newName.InnerText="C. Monster";
newAuthor.AppendChild(newName);
//create new price element
XmlElement newPrice=doc.CreateElement("price");
newPrice.InnerText="9.95";
newBook.AppendChild(newPrice);
//add to the current document
doc.DocumentElement.AppendChild(newBook);
//write out the doc to disk
XmlTextWriter tr=new XmlTextWriter("booksEdit.xml",null);
tr.Formatting=Formatting.Indented;
doc.WriteContentTo(tr);
tr.Close();
//load listBox1 with all of the titles, including new one
XmlNodeList nodeList=doc.GetElementsByTagName("title");
foreach(XmlNode node in nodeList)
{
    textBox1.Text += node.OuterXml + "\r\n";
}
}

```

在执行这段代码后，会得到与上一个示例相同的结果，但本例在列表框中添加了一本书：*The Case of the Missing Cookie*。仔细查看代码，这是一个相当简单的过程。首先，创建一个新的 book 元素：

```
XmlElement newBook = doc.CreateElement("book");
```

CreateElement()有 3 个重载方法，可以指定：

- 元素名
- 名称和命名空间 URI
- 前缀、本地名和命名空间

创建了该元素后，就要添加属性了：

```

newBook.SetAttribute("genre","Mystery");
newBook.SetAttribute("publicationdate","2001");
newBook.SetAttribute("ISBN","123456789");

```

创建了属性后，就要添加书籍的其他元素了：

```

XmlElement newTitle = doc.CreateElement("title");
newTitle.InnerText = "The Case of the Missing Cookie";

```

```
newBook.AppendChild(newTitle);
```

再次创建一个基于 `XmlElement` 的新对象(`newTitle`), 把 `InnerText` 属性设置为新书名, 把该元素添加为 `book` 元素的一个子元素。对 `book` 元素中的其他元素重复这一操作。注意把 `name` 元素添加为 `author` 元素的一个子元素。这样就可以在其他 `book` 元素中得到合适的嵌套关系。

最后把 `newBook` 元素添加到 `doc.DocumentElement` 节点上, 它与其他 `book` 元素同级。现在用新元素更新现有的文档。

最后, 把新 XML 文档写到磁盘上。在这个示例中, 创建一个新 `XmlTextWriter`, 把它传送给 `WriteContentTo` 方法。`WriteContentTo` 和 `WriteTo` 方法都带一个 `XmlTextWriter` 参数。`WriteContentTo` 把当前节点及其所有的子节点都保存到 `XmlTextWriter`, 而 `WriteTo` 只保存当前节点。因为 `doc` 是一个基于 `XmlDocument` 的对象, 它表示整个文档, 所以应保存它。我们还使用了 `Save` 方法, 它总是保存整个文档, `Save` 有 4 个重载方法, 其参数分别是一个包含文件名和路径的字符串、基于 `Stream` 的对象、基于 `TextWriter` 的对象和基于 `XmlWriter` 的对象。

我们还在 `XmlTextWriter` 上调用了 `Close()` 方法, 刷新内部缓存, 并关闭文件。

在运行这个示例时, 会得到如图 28-1 所示的屏幕图。注意列表框底部的新项。

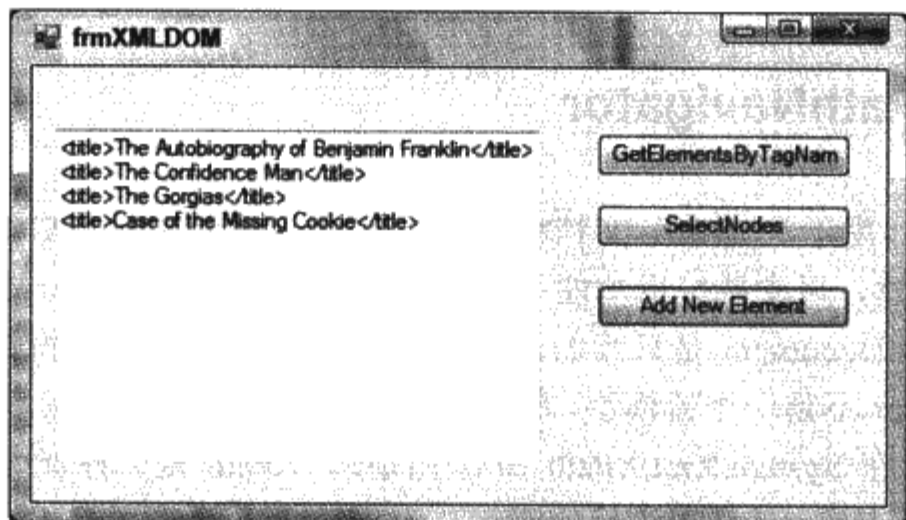


图 28-1

本章的前面说明了如何使用 `XmlTextWriter` 类创建一个文档, 还可以使用 `XmlDocument`。使用哪个比较好? 如果要写入 XML 流的数据已经准备好, 最好选择 `XmlTextWriter` 类。但是, 如果需要一次建立 XML 文档的一小部分, 在不同的地方插入节点, 用 `XmlDocument` 创建文档就比较好。为此, 可以把下面的代码:

```
doc.Load("books.xml");
```

改为:

```
//create the declaration section
XmlDeclaration newDec = doc.CreateXmlDeclaration("1.0", null, null);
doc.AppendChild(newDec);
//create the new root element
XmlElement newRoot = doc.CreateElement("newBookstore");
doc.AppendChild(newRoot);
```

首先创建一个新 `XmlDeclaration`, 其参数是版本(目前是"1.0")、编码和 `standalone` 标志。如果没有使用 `null`, 编码参数应设置为一个字符串, 该字符串应是 `System.Text.Encoding` 类的一部分。`null` 默认为 UTF-8。`standalone` 标志可以是 `yes`、`no` 或 `null`, 但如果是 `null`, 就不使用该属

性，也不包含在文档中。

要创建的下一个元素是 `DocumentElement`。在本例中，它称为 `newBookstore`，这样区别就比较明显。代码的其余部分与前面的示例相同，执行的方式也相同。下面是从代码中生成的 `booksEdit.xml`：

```
<?xml version="1.0"?>
<newBookstore>
  <book genre="Mystery" publicationdate="2001" ISBN="123456789">
    <title>The Case of the Missing Cookie</title>
    <author>
      <name>C. Monster</name>
    </author>
    <price>9.95</price>
  </book>
</newBookstore>
```

在希望随机访问文档时，可以使用 `XmlDocument` 类。在希望有一个流类型的模型时，可以使用基于 `XmlReader` 的类。基于 `XmlDocument` 的 `XmlNode` 的灵活性要求的内存比较多，读取文档的性能也没有使用 `XmlReader` 好，遍历 XML 文档还有另一种方式：`XPathNavigator`。

## 28.6 使用 XPathNavigator

`XPathNavigator` 用于从 XML 文档中选择、迭代、编辑数据。`XPathNavigator` 可以从 `XmlDocument` 中创建，以支持编辑功能；它也可以从 `XPathDocument` 中创建，此时只能用于读取。因为 `XPathDocument` 是只读的，所以 `XPathNavigator` 执行得很好。与 `XmlReader` 不同，`XPathNavigator` 不是一个流模型，所以同一个文档不需要重新读取和分析，也能使用。

`XPathNavigator` 在 `System.Xml.XPath` 命名空间中，`XPath` 是一种查询语言，可以从 XML 文档中选择特定的节点或元素，以进行处理。

### 28.6.1 System.Xml.XPath 命名空间

`System.Xml.XPath` 命名空间是建立在速度的基础上的，它提供了 XML 文档的一种只读视图，但没有编辑功能。这个命名空间中的类可以采用光标的方式在 XML 文档上进行快速迭代和选择操作。

表 28-6 列出了 `System.Xml.XPath` 命名空间中的重要类，并对每个类的功能进行了简单的说明。

表 28-6

类 名	说 明
<code>XPathDocument</code>	提供整个 XML 文档的视图，只读
<code>XPathNavigator</code>	提供 <code>XPathDocument</code> 的浏览功能
<code>XPathNodeIterator</code>	提供节点集的迭代功能
<code>XPathExpression</code>	编译好的 XPath 表达式，由 <code>SelectNodes</code> 、 <code>SelectSingleNode</code> 、 <code>Evaluate</code> 和 <code>Matches</code> 使用
<code>XPathException</code>	XPath 异常类

1. XPathDocument 类

XPathDocument 没有提供 XmlDocument 类的任何功能，它唯一的功能是创建 XPathNavigator。因此，这是 XPathDocument 类上唯一可用的方法(其他由 Object 提供)。

XPathDocument 可以用许多方式创建。可以给构造函数传送 XmlReader、XML 文档的文件名或基于流的对象，其灵活性非常大。例如，可以使用 XmlValidatingReader 验证 XML，并使用该对象创建 XPathDocument。

2. XPathNavigator 类

XPathNavigator 包含移动和选择所需元素的所有方法，其中的一些移动方法如表 28-7 所示。

表 28-7

方 法 名	说 明
MoveTo()	把 XPathNavigator 作为参数，移动当前位置到 XPathNavigator 指定的地方
MoveToAttribute()	移动到指定的属性，其参数是属性名和命名空间
MoveToFirstAttribute()	移动到当前元素中的第一个属性上，如果成功，就返回 true
MoveToNextAttribute()	移动到当前元素中的下一个属性上，如果成功，就返回 true
MoveToFirst()	移动到当前节点中的第一个同级节点上，如果成功，就返回 true。否则返回 false
MoveToLast()	移动到当前节点中的最后一个同级节点上，如果成功，就返回 true
MoveToNext()	移动到当前节点中的下一个同级节点上，如果成功，就返回 true
MoveToPrevious()	移动到当前节点中的上一个同级节点上，如果成功，就返回 true
MoveToFirstChild()	移动到当前元素中的第一个子元素上，如果成功，就返回 true
MoveToId()	移动到 ID 参数提供的元素上，文档中需要有一个模式，元素的数据类型必须是 ID 类型
MoveToParent()	移动到当前节点的父节点上，如果成功，就返回 true
MoveToRoot()	移动到文档的根节点上

要选择文档的一个子集，可以使用表 28-8 所示的 Select()方法。

表 28-8

方 法 名	说 明
Select()	使用 XPath 表达式选择一个节点集
SelectAncestors`()	根据 XPath 表达式选择当前节点的所有祖先节点
SelectChildren()	根据 XPath 表达式选择当前节点的所有子节点
SelectDescendants()	根据 XPath 表达式选择当前节点的所有子孙节点
SelectSingleNode()	使用 XPath 表达式选择一个节点

如果 XPathNavigator 是从 XPathDocument 中创建的，它就是只读的。如果 XPathNavigator 是从 XmlDocument 中创建的，它就可以用于编辑文档。查看 CanEdit 属性就可以验证这一点。



如果该属性是 `true`，就可以使用某个 `Insert` 方法。`InsertBefore` 和 `InsertAfter` 会在当前节点的前面和后面创建一个新节点。新节点的源可以是 `XmlReader` 或字符串。还可以返回一个 `XmlWriter`，用于写入新节点信息。

使用 `ValueAs` 属性可以从节点中读取强类型化的值。注意这与 `XmReader` 不同，`XmReader` 使用 `ReadValue` 方法。

### 3. XPathNodeIterator 类

`XPathNodeIterator` 可以看作是 `XPath` 中的 `NodeList` 或 `NodeSet`，这个对象有 3 个属性和两个方法：

- `Clone`——创建它本身的一个新副本。
- `Count`——`XPathNodeIterator` 对象中的节点数。
- `Current`——返回指向当前节点的 `XPathNavigator`。
- `CurrentPosition()`——返回表示当前位置的一个整数。
- `MoveNext()`——移动到匹配 `XPath` 表达式的下一个节点上，创建 `XPathNodeIterator`。

`XPathNodeIterator` 由 `XPathNavigators` 的 `Select` 方法返回，使用它可以迭代 `XPathNavigator` 的 `Select` 方法返回的节点集。使用 `XPathNodeIterator` 的 `MoveNext` 方法不会改变创建它的 `XPathNavigator` 的位置。

### 4. 使用 XPath 命名空间中的类

要理解这些类的用法，最好是查看一下迭代 `books.xml` 文档的代码，确定导航是如何工作的。为了使用这些示例，首先需要添加对 `System.Xml.Xsl` 和 `System.Xml.XPath` 命名空间的引用，如下所示：

```
using System.Xml.XPath;
using System.Xml.Xsl;
```

这个示例使用了 `booksxpath.xml` 文件，它类似于前面使用的 `books.xml`，但 `booksxpath.xml` 添加了两本书。下面是窗体代码，这段代码在 `XmlSample` 项目中：

```
private void button1_Click(object sender, EventArgs e)
{
    //modify to match your path structure
    XPathDocument doc = new XPathDocument("books.xml");
    //create the XPath navigator
    XPathNavigator nav = ((IXPathNavigable)doc).CreateNavigator();
    //create the XPathNodeIterator of book nodes
    // that have genre attribute value of novel
    XPathNodeIterator iter = nav.Select("/bookstore/book[@genre='novel']");
    textBox1.Text = "";
    while (iter.MoveNext())
    {
        XPathNodeIterator newIter =
        iter.Current.SelectDescendants(XPathNodeType.Element, false);
        while (newIter.MoveNext())
            textBox1.Text += newIter.Current.Name + ": "
            + newIter.Current.Value + "\r\n";
    }
}
```



在 `button1_Click()` 方法中, 首先创建 `XPathDocument` (叫做 `doc`), 其参数是要打开的文档的文件和路径字符串。下面一行代码创建 `XPathNavigator`:

```
XPathNavigator nav = doc.CreateNavigator();
```

本例用 `Select` 方法获取 `genre` 属性值为 `novel` 的所有节点, 然后使用 `MoveNext()` 方法迭代书籍列表中的所有小说。

要把数据加载到列表框中, 使用 `XPathNodeIterator.Current` 属性。根据 `XPathNodeIterator` 指向的节点, 创建一个新的 `XPathNavigator` 对象。在本例中, 为文档中的一个 `book` 节点创建 `XPathNavigator`。

之后的循环提取这个 `XPathNavigator`, 调用 `Select` 方法的另一个重载方法 `SelectDescendants` 创建另一个 `XpathNodeIterator`。这样, `XPathNodeIterator` 就包含了 `book` 节点的所有子节点。

然后, 在这个 `XPathNodeIterator` 上执行另一个 `MoveNext()` 循环, 给文本框加载元素名称和元素值。在执行代码后, 显示 28-2 所示的屏幕图, 注意只列出了小说。

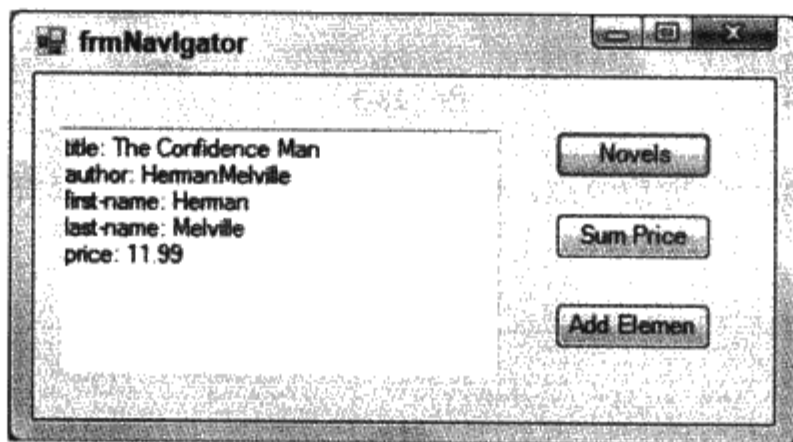


图 28-2

如果要把这些书的成本相加, 该怎么办? `XPathNavigator` 为此包含了 `Evaluate` 方法。`Evaluate` 有 3 个重载方法, 第一个包含一个字符串, 该字符串就是 `XPath` 函数调用。第二个 `Evaluate` 重载方法的参数是 `XPathExpression` 对象, 第三个 `Evaluate` 重载方法的参数是 `XPathExpression` 和 `XPathNodeIterator`。下面的代码类似于前面的示例, 但这次迭代文档中的所有节点。最后的 `Evaluate` 方法汇总了所有书籍的成本:

```
private void button2_Click(object sender, EventArgs e)
{
    //modify to match your path structure
    XPathDocument doc = new XPathDocument("books.xml");
    //create the XPath navigator
    XPathNavigator nav = ((IXPathNavigable)doc).CreateNavigator();
    //create the XPathNodeIterator of book nodes
    // that have genre attribute value of novel
    XPathNodeIterator iter = nav.Select("/bookstore/book");
    textBox1.Text = "";
    while (iter.MoveNext())
    {
        XPathNodeIterator newIter =
            iter.Current.SelectDescendants(XPathNodeType.Element, false);
        while (newIter.MoveNext())
            textBox1.Text += newIter.Current.Name + ": " +
                newIter.Current.Value + "\r\n";
    }
}
```

```

}
textBox1.Text+= "===== ";
textBox1.Text+="Total Cost = " + nav.Evaluate("sum(/bookstore/book/price)");
}

```

这次，可以看到列表框中书籍的总成本，如图 28-3 所示。

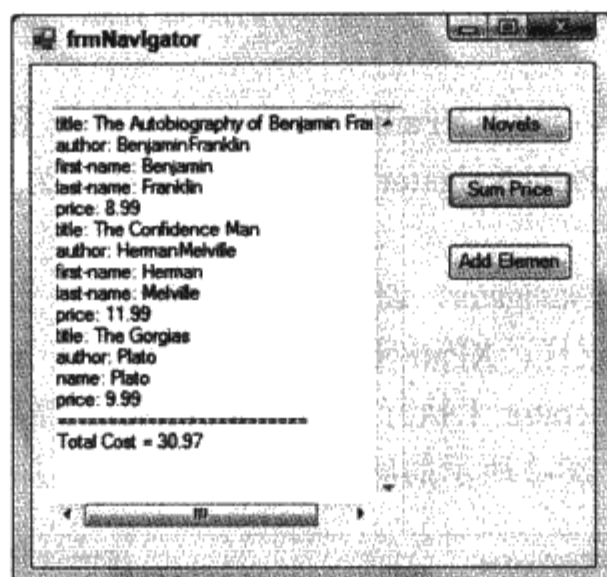


图 28-3

现在，假定需要添加一个折扣节点。使用 `InsertAfter` 方法可以很容易插入该节点。下面是代码：

```

private void button3_Click(object sender, EventArgs e)
{
    XmlDocument doc = new XmlDocument();
    doc.Load("books.xml");
    XPathNavigator nav = doc.CreateNavigator();
    if (nav.CanEdit)
    {
        XPathNodeIterator iter = nav.Select("/bookstore/book/price");
        while (iter.MoveNext())
        {
            iter.Current.InsertAfter("<disc>5</disc>");
        }
    }
    doc.Save("newbooks.xml");
}

```

这段代码在价格元素的后面添加了 `<disc>5</disc>` 元素。首先选择前面所有的价格节点，使用 `XPathNodeIterator` 迭代节点，插入新节点。修改后的文档保存为一个新名称 `newbooks.xml`。下面是新文档的内容：

```

<?xml version="1.0"?>
<!-- This file represents a fragment of a book store inventory database -->
<bookstore>
  <book genre="autobiography" publicationdate="1991" ISBN="1-861003-11-0">
    <title>The Autobiography of Benjamin Franklin</title>
    <author>
      <first-name>Benjamin</first-name>
      <last-name>Franklin</last-name>
    </author>
    <price>8.99</price>
    <disc>5</disc>
  </book>

```

```

</book>
<book genre="novel" publicationdate="1967" ISBN="0-201-63361-2">
  <title>The Confidence Man</title>
  <author>
    <first-name>Herman</first-name>
    <last-name>Melville</last-name>
  </author>
  <price>11.99</price>
  <disc>5</disc>
</book>
<book genre="philosophy" publicationdate="1991" ISBN="1-861001-57-6">
  <title>The Gorgias</title>
  <author>
    <name>Plato</name>
  </author>
  <price>9.99</price>
  <disc>5</disc>
</book>
</bookstore>

```

节点可以插入到选中的节点之前或之后。还可以修改节点，删除它们。如果对许多节点做了改动，最好使用从 XmlDocument 中创建的 XPathNavigator。

## 28.6.2 System.Xml.Xsl 命名空间

System.Xml.Xsl 命名空间包含 .NET Framework 用于支持 XSL Transform 的类。这个命名空间中的类可以和任何实现 XPathNavigator 接口的存储器一起使用。在目前的 .NET Framework 中，包含 XmlDocument、XmlDataDocument 和 XPathDocument。与 XPath 一样，应使用最有效的存储器。如果计划创建一个定制存储器，例如文件系统，并进行一定的转换，就应在类中实现 XPathNavigator 接口。

XSL 建立在一个流拉模式(streaming pull mode)上。因此，可以把几个转换链接在一起。如果需要，甚至可以在转换之间应用一个定制读取器，这样在设计时就会有很大的灵活性。

### 1. 转换 XML

第一个示例获取 books.xml 文档，并使用 XSLT 文件 book.xsl 把它转换为一个简单的 HTML 文档，以进行显示(这段代码在 XPathXSLSample3 文件夹中)，需要添加如下 using 语句：

```

using System.IO;
using System.Xml.Xsl;
using System.Xml.XPath;

```

下面是执行转换的代码：

```

private void button1_Click(object sender, EventArgs e)
{
    XslCompiledTransform trans = new XslCompiledTransform();
    trans.Load("books.xsl");
    trans.Transform("books.xml", "out.html");
    webBrowser1.Navigate(AppDomain.CurrentDomain.BaseDirectory +
        "out.html");
}

```

这就是一个简单的转换。首先创建一个新的 XmlCompiledTransform 对象。它加载 books.xsl



转换文件，然后执行转换。在本例中，把带文件名的字符串用作输入。输出是 out.html。之后把这个文件加载到窗体使用的 Web 浏览器控件上。这里不把文件名 books.xml 用作输入文档，而是使用一个基于 XPathNavigator 的对象。该对象可以创建 XPathNavigator。

创建 XmlCompiledTransform 对象，加载样式表后，就执行转换。Transform 方法的参数可以是 XPathNavigator 对象、流、TextWriters、XmlWriters 和 URI 的任意组合，因此转换流上的灵活性很大。可以把转换的结果作为输入传送给下一个转换操作。

XsltArgumentLists 和 XmlResolver 对象也包含在参数选项中。下一节介绍 XsltArgumentList 对象。基于 XmlResolver 的对象用于解析当前文档外部的数据项，例如模式、证书和样式表。

books.xsl 文档是一个很简单的样式表，如下所示：

```
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <html>
    <head>
      <title>Price List</title>
    </head>
    <body>
      <table>
        <xsl:apply-templates/>
      </table>
    </body>
  </html>
</xsl:template>
<xsl:template match="bookstore">
  <xsl:apply-templates select="book"/>
</xsl:template>
<xsl:template match="book">
  <tr><td>
    <xsl:value-of select="title"/>
  </td><td>
    <xsl:value-of select="price"/>
  </td></tr>
</xsl:template>
</xsl:stylesheet>
```

## 2. 使用 XsltArgumentList

XsltArgumentList 是把对象和方法绑定到命名空间上的一种方式。绑定好后，就可以在转换过程中调用该方法。下面是一个示例：

```
private void button3_Click(object sender, EventArgs e)
{
  //new XPathDocument
  XPathDocument doc = new XPathDocument("books.xml");
  //new XslTransform
  XslCompiledTransform trans = new XslCompiledTransform();
  trans.Load("booksarg.xsl");
  //new XmlTextWriter since we are creating a new xml document
  XmlWriter xw = new XmlTextWriter("argSample.xml", null);
  //create the XsltArgumentList and new BookUtils object
  XsltArgumentList argBook = new XsltArgumentList();
  BookUtils bu = new BookUtils();
  //this tells the argumentlist about BookUtils
  argBook.AddExtensionObject("urn:XslSample", bu);
}
```

```
//new XPathNavigator
XPathNavigator nav = doc.CreateNavigator();
//do the transform
trans.Transform(nav, argBook, xw);
xw.Close();
webBrowser1.Navigate(AppDomain.CurrentDomain.BaseDirectory + "//argSample.xml");
}
```

下面是 BooksUtil 类的代码，这个类将在转换过程中调用：

```
class BookUtils
{
    public BookUtils() { }

    public string ShowText()
    {
        return "This came from the ShowText method!";
    }
}
```

下面是转换的结果。其结果已进行了格式化，以便于查看(argSample.xml)：

```
<books>
  <discbook>
    <booktitle>The Autobiography of Benjamin Franklin</booktitle>
    <showtext>This came from the ShowText method!</showtext>
  </discbook>
  <discbook>
    <booktitle>The Confidence Man</booktitle>
    <showtext>This came from the ShowText method!</showtext>
  </discbook>
  <discbook>
    <booktitle>The Gorgias</booktitle>
    <showtext>This came from the ShowText method!</showtext>
  </discbook>
  <discbook>
    <booktitle>The Great Cookie Caper</booktitle>
    <showtext>This came from the ShowText method!</showtext>
  </discbook>
  <discbook>
    <booktitle>A Really Great Book</booktitle>
    <showtext>This came from the ShowText method!</showtext>
  </discbook>
</books>
```

在本例中，我们定义一个新类 BookUtils。在这个类中有一个不起作用的方法，它返回字符串 This came from the ShowText method!。在 button3\_Click 事件中，创建了 XPathDocument 和 XsltTransform 对象，前面的示例把 XML 文档和转换文档直接加载到 XslCompiledTransform 对象中，这次使用了 XPathNavigator 来加载文档。

接着执行下面的代码：

```
XsltArgumentList argBook=new XsltArgumentList();
BookUtils bu=new BookUtils();
argBook.AddExtensionObject("urn:ProCSharp",bu);
```

这段代码创建了 XsltArgumentList 对象。接着创建 BookUtils 对象的一个实例，在调用 AddExtensionObject 方法时，其参数是扩展的命名空间和要调用方法的对象。在调用 Transform



时, 其参数是 `XsltArgumentList (argBook)`、前面创建的 `XPathNavigator` 和 `XmlWriter` 对象。

下面是 `booksarg.xsl` 文档(基于 `books.xsl`):

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                  xmlns:bookUtil="urn:XslSample">
  <xsl:output method="xml" indent="yes"/>

  <xsl:template match="/">
    <xsl:element name="books">
      <xsl:apply-templates/>
    </xsl:element>
  </xsl:template>
  <xsl:template match="bookstore">
    <xsl:apply-templates select="book"/>
  </xsl:template>
  <xsl:template match="book">
    <xsl:element name="discbook">
      <xsl:element name="booktitle">
        <xsl:value-of select="title"/>
      </xsl:element>
      <xsl:element name="showtext">
        <xsl:value-of select="bookUtil:ShowText()" />
      </xsl:element>
    </xsl:element>
  </xsl:template>
</xsl:stylesheet>
```

两行重要的代码已突出显示。首先, 在给 `XsltArgumentList` 添加对象时, 添加了前面创建的命名空间。然后, 在调用方法时, 使用标准的 XSLT 命名空间前缀语法。

另一种方式是使用 XSLT 脚本。可以在该样式表中包含 C#、VB 和 JavaScript 代码。最重要的是在 `Transform.Load` 调用中编译该脚本, 这与目前的非.NET 实现方式不同。这样就可以执行已经编译好的脚本。

下面对前面的 XSLT 文件也进行这样的操作。首先给样式表添加脚本, 在 `booksscript.xsl` 中进行的修改如下所示:

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                  xmlns:msxsl="urn:schemas-microsoft-com:xslt"
                  xmlns:user="http://wrox.com">

  <msxsl:script language="C#" implements-prefix="user">

    string ShowText()
    {
      return "This came from the ShowText method!";
    }
  </msxsl:script>

  <xsl:output method="xml" indent="yes"/>
  <xsl:template match="/">
    <xsl:element name="books">
      <xsl:apply-templates/>
    </xsl:element>
  </xsl:template>
  <xsl:template match="bookstore">
    <xsl:apply-templates select="book"/>
  </xsl:template>
  <xsl:template match="book">
    <xsl:element name="discbook">
```

```

<xsl:element name="booktitle">
  <xsl:value-of select="title"/>
</xsl:element>
<xsl:element name="showtext">
  <xsl:value-of select="user:ShowText()" />
</xsl:element>
</xsl:template>
</xsl:stylesheet>

```

其中的改变已突出显示。设置脚本的命名空间，添加代码(可以从 VS.NET IDE 中复制和粘贴)，在样式表上执行调用。输出结果与前面示例的相同。

### 28.6.3 调试 XSLT

Visual Studio 2008 可以调试转换操作。我们可以单步执行转换代码，查看变量，访问调用堆栈，设置断点，这些都与调试 C#源代码相同。调试转换代码有两种方式：仅使用样式表和 XML 输入文件，或者运行转换代码所在的应用程序。

#### 1. 在不运行应用程序的情况下调试

第一次创建转换操作时，有时并不希望运行整个应用程序，只希望使样式表工作。Visual Studio 2008 允许使用 XSLT 编辑器进行这个操作。

把 books.xml 样式表加载到 Visual Studio 2008 XSLT 编辑器中。在下面的代码上设置一个断点：

```
< xsl:value-of select="title"/ >
```

现在选择 XML 菜单，再选择 Debug XSLT。此时需要指定 XML 输入文档，这就是我们要转换的 XML。在默认配置下，结果如图 28-4 所示。

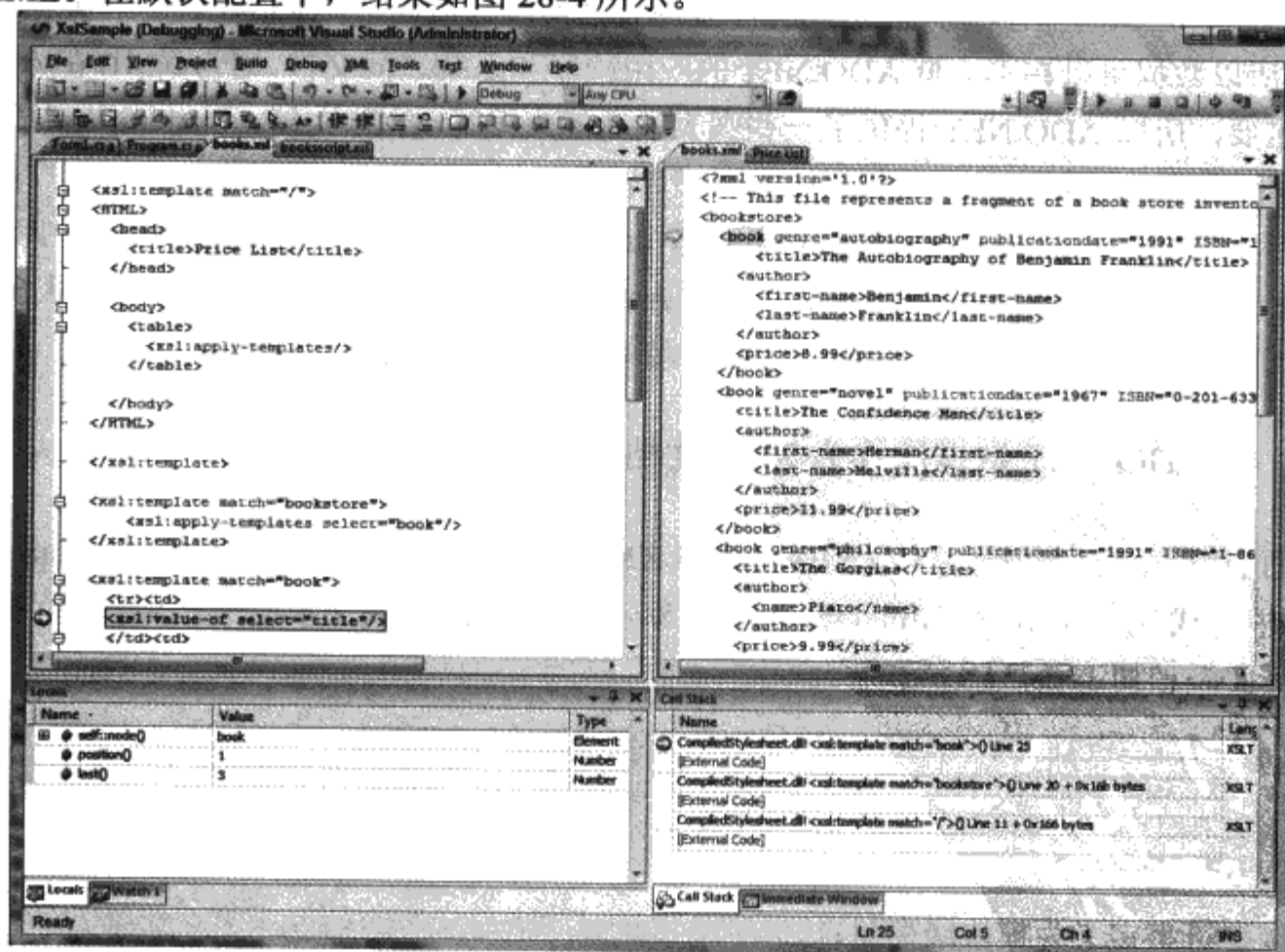


图 28-4

暂停转换，就可以查看与调试源代码时相同的几乎所有的调试信息。注意调试器显示了 XSLT、输入文档、当前调试的元素和转换的结果。现在就可以单步执行转换代码了。如果 XSLT 包含脚本代码，还可以在脚本中设置断点，使用相同的调试功能。

## 2. 在运行应用程序的情况下调试

如果要同时调试转换代码和应用程序，就必须在创建 `XslCompiledTransform` 对象时做一个小小的改动。构造函数的一个重载版本把一个布尔值作为参数，这个参数是 `enableDebug`，默认为 `false`，表示即使在转换代码中有一个断点，则运行了调用转换代码的应用程序，也不会中断。如果把该参数设置为 `true`，就会生成 CSLT 的调试信息，并在断点处暂停。所以在前面的示例中，创建 `XslCompiledTransform` 的代码应改为：

```
XslCompiledTransform trans = new XslCompiledTransform(true);
```

现在，应用程序运行在调试模式下，所以 XSLT 会生成调试信息，我们可以在样式表中获得完整的 Visual Studio 调试功能。

总之，在进行转换时，一定要记住使用正确的 XML 数据存储。如果不需要编辑功能，就使用 `XPathDocument`；如果要从 ADO.NET 中获得数据，就使用 `XmlDataDocument`；如果需要编辑数据，就使用 `XmlDocument`。其他过程都相同。

## 28.7 XML 和 ADO.NET

XML 是把 ADO.NET 绑定到其他语言中的纽带。ADO.NET 从一开始就设计为在 XML 环境中工作。XML 用于在数据库和应用程序或网页之间传输数据。ADO.NET 使用 XML 进行远程传输，所以数据可以在不支持 ADO.NET 的应用程序和系统之间交换。因为 XML 在 ADO.NET 中非常重要，所以 ADO.NET 提供了一些强大的功能来读写 XML 文档。`System.Xml` 命名空间也包含可以使用 ADO.NET 关系数据的类。

用于示例的数据库来自于 `AdventureWorksLT` 示例应用程序。示例数据库可以从 [codeplex.com/SqlServerSamples](http://codeplex.com/SqlServerSamples) 上下载。注意 `AdventureWorks` 数据库有几个版本，大多数版本都可以工作，但 LT 版本是简化版本，足以满足本章的目的。

### 28.7.1 将 ADO.NET 数据转换为 XML 文档

下面的第一个示例使用 ADO.NET、流和 XML 把数据库中的一些数据推入 `DataSet`，并从 `DataSet` 中加载带有 XML 的 `XmlDocument` 对象，把 XML 加载到文本框中。为了运行下面几个示例，需要添加如下 `using` 语句：

```
using System.Data;
using System.Xml;
using System.Data.SqlClient;
using System.IO;
```

连接字符串定义为模块级变量：

```
string _connectString = "Server=.\SQLExpress;
    Database=adventureworkslt;Trusted_Connection=Yes";
```

对于 ADO.NET 示例，在窗体上添加一个 DataGrid，这样就可以在 ADO.NET DataSet 中查看数据了，因为数据被绑定到网格上。还可以查看生成的 XML 文档中的数据，这些数据加载到了文本框中。下面是第一个示例的代码。本示例的第一步是创建标准的 ADO.NET 对象，以创建数据集。之后把它绑定到网格上。

```
private void button1_Click(object sender, EventArgs e)
{
    XmlDocument doc = new XmlDocument();
    DataSet ds = new DataSet("XMLProducts");
    SqlConnection conn = new SqlConnection(_connectString);
    SqlDataAdapter da = new SqlDataAdapter
        ("SELECT Name, StandardCost FROM SalesLT.Product", conn);
    //fill the dataset
    da.Fill(ds, "Products");
    //load data into grid
    dataGridView1.DataSource = ds.Tables["Products"];
}
```

在创建了 ADO.NET 对象，并绑定到网格上后，再实例化 MemoryStream、StreamReader 和 StreamWriter 对象。StreamReader 和 StreamWriter 对象使用 MemoryStream 来浏览 XML 文档：

```
MemoryStream memStrm=new MemoryStream();
StreamReader strmRead=new StreamReader(memStrm);
StreamWriter strmWrite=new StreamWriter(memStrm);
```

使用 MemoryStream 的原因是不必把数据写入磁盘。还可以使用基于 Stream 类的其他对象，例如 FileStream。

下一步是生成 XML。调用 DataSet 类的 WriteXml() 方法，它生成一个 XML 文档。WriteXml() 有两个重载方法，一个重载方法的参数是带有文件路径和名称的字符串，另一个重载方法还有一个参数，即模式。这个模式是一个 XmlWriteMode 枚举，其值可以是

- IgnoreSchema
- WriteSchema
- DiffGram

如果不希望 WriteXml() 方法把内联模式写入 XML 文件的开头，就使用 IgnoreSchema 参数，如果要写入内联模式，就使用 WriteSchema 参数。DiffGram 显示在 DataSet 中编辑前后的数据。本节的后面介绍 DiffGram。

```
//write the xml from the dataset to the memory stream
ds.WriteXml(strmWrite, XmlWriteMode.IgnoreSchema);
memStrm.Seek(0, SeekOrigin.Begin);
//read from the memory stream to a XmlDocument object
doc.Load(strmRead);
//get all of the products elements
XmlNodeList nodeList = doc.SelectNodes("//XMLProducts/Products");
textBox1.Text = "";

foreach (XmlNode node in nodeList)
{
    textBox1.Text += node.InnerXml + "\r\n";
}
```

在 28-5 所示的屏幕图中，可以查看列表框中的数据和绑定的数据网格。

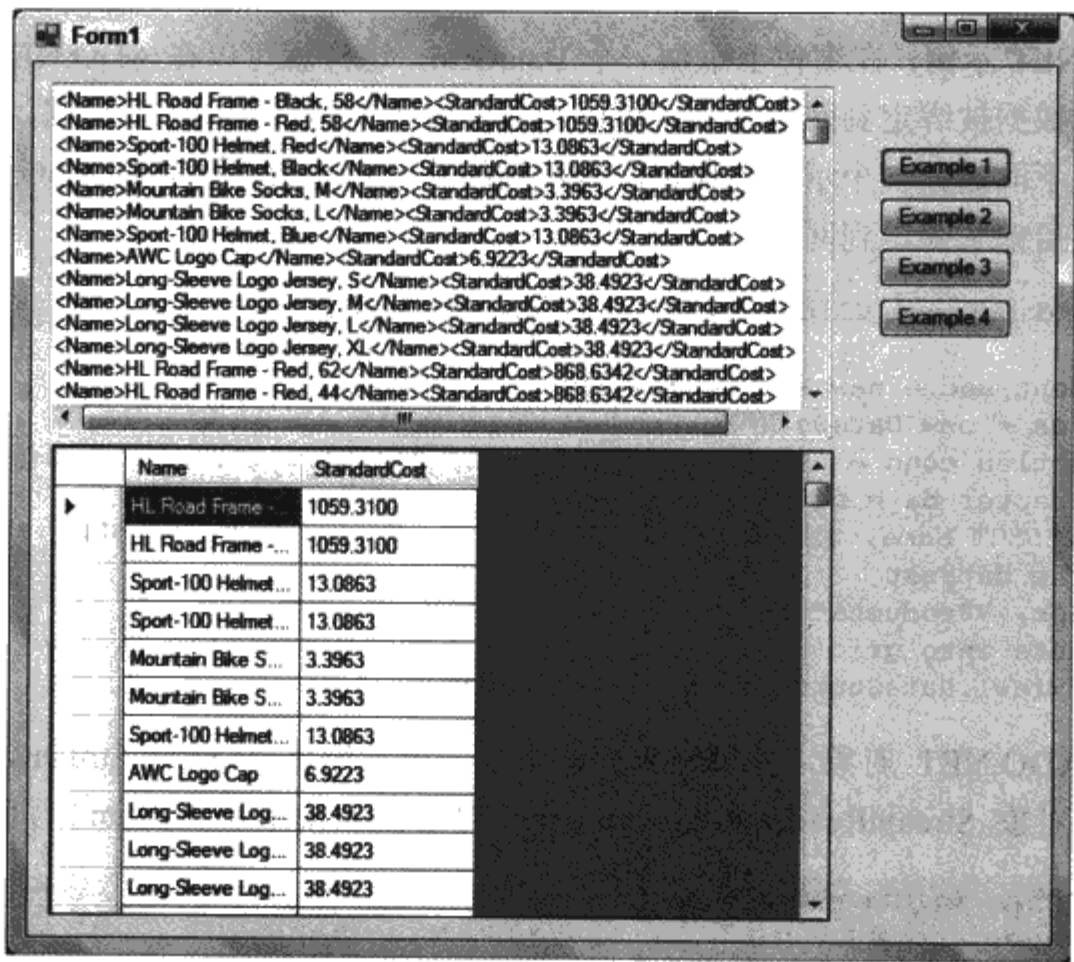


图 28-5

如果只需要该模式，可调用 WriteXmlSchema()而不是 WriteXml()。这个方法有 4 个重载方法，第一个重载方法的参数是一个字符串，其中包含了 XML 文档的路径和文件名，第二个重载方法使用基于 XmlWriter 类的一个对象。第三个重载方法使用基于 TextWriter 类的对象。第四个重载方法使用派生于 Stream 类的对象。

如果要把 XML 文档存入磁盘，就应执行下述操作：

```
string file = "c:\\test\\product.xml";
ds.WriteXml(file);
```

在磁盘上会生成一个格式正确的 XML 文档，并可以由另一个流、DataSet 来读取，或者由另一个应用程序或网站使用。因为没有指定 XmlMode 参数，所以这个 XmlDocument 包含了模式。在本例中，把流作为 XmlDocument.Load()方法的参数。

现在数据有两个视图，但更重要的是，可以使用两个不同的模型来处理数据。可以使用 System.Data 命名空间来操纵数据，也可以使用 System.XML 命名空间来处理数据。这样应用程序就可以有某些非常灵活的设计，因为现在不必把一个对象模型绑定到程序上。这是 ADO.NET 和 System.Xml 组合的强大之处。相同数据可以有多个视图，访问数据也有多种方式。

下一个示例去掉了 3 个流，使用内置于 System.Xml 命名空间中的一些 ADO 功能简化了过程。但需要修改模块级的代码行：

```
private XmlDocument doc = new XmlDocument();
```

为：



```
private XmlDataDocument doc;
```

这么做的原因是现在使用的是 XmlDataDocument。下面的代码在 ADOSample2 文件夹中：

```
private void button3_Click(object sender, EventArgs e)
{
    XmlDataDocument doc;
    //create a dataset
    DataSet ds = new DataSet("XMLProducts");
    //connect to the northwind database and
    //select all of the rows from products table
    SqlConnection conn = new SqlConnection(_connectString);
    SqlDataAdapter da = new SqlDataAdapter
        ("SELECT Name, StandardCost FROM SalesLT.Product", conn);
    //fill the dataset
    da.Fill(ds, "Products");
    ds.WriteXml("sample.xml", XmlWriteMode.WriteSchema);
    //load data into grid
    dataGridView1.DataSource = ds.Tables[0];
    doc = new XmlDataDocument(ds);
    //get all of the products elements
    XmlNodeList nodeList = doc.GetElementsByTagName("Products");
    textBox1.Text = "";
    foreach (XmlNode node in nodeList)
    {
        textBox1.Text += node.InnerXml + "\r\n";
    }
}
```

可以看出，把 DataSet 对象加载到 XML 文档中的代码已经进行了简化。它没有使用 XmlDocument 类，而使用了 XmlDataDocument 类，这个类是为了使用 DataSet 对象的数据而专门设计的。

XmlDataDocument 基于 XmlDocument 类，所以它拥有 XmlDocument 类的所有功能。一个主要区别是 XmlDataDocument 有重载的构造函数。注意下面的代码实例化了 XmlDataDocument 对象 doc：

```
doc = new XmlDataDocument(ds);
```

它的参数是我们创建的 DataSet 对象 ds，从 DataSet 对象中创建 XML 文档，而且不必使用 Load() 方法。实际上，如果实例化一个新的 XmlDataDocument 对象时，不把 DataSet 作为参数，该 XmlDataDocument 就会包含一个名为 NewDataSet 的 DataSet，其 tables 集合中没有任何 DataTable，在创建了基于 XmlDataDocument 的对象后，还可以设置一个 DataSet 属性。

如果下面的代码添加到 DataSet.Fill 调用的后面：

```
ds.WriteXml("c:\\test\\sample.xml", XmlWriteMode.WriteSchema);
```

就会在文件夹 c:\test 中生成下面的 XML 文件 sample.xml：

```
< ?xml version="1.0" standalone="yes"? >
< XMLProducts >
  < xs:schema id="XMLProducts" xmlns=""
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:msdata="urn:schemas-microsoft-com:xml-msdata" >
  < xs:element name="XMLProducts" msdata:IsDataSet="true" msdata:
    UseCurrentLocale="true" >
```

```

< xs:complexType >
  < xs:choice minOccurs="0" maxOccurs="unbounded" >
    < xs:element name="Products" >
      < xs:complexType >
        < xs:sequence >
          < xs:element name="Name" type="xs:string" minOccurs="0" / >
          < xs:element name="StandardCost" type="xs:decimal" minOccurs="0" / >
        < /xs:sequence >
      < /xs:complexType >
    < /xs:element >
  < /xs:choice >
< /xs:complexType >
< /xs:element >
< /xs:schema >
< Products >
  < Name > HL Road Frame - Black, 58 < /Name >
  < StandardCost > 1059.3100 < /StandardCost >
< /Products >
< Products >
  < Name > HL Road Frame - Red, 58 < /Name >
  < StandardCost > 1059.3100 < /StandardCost >
< /Products >
< Products >
  < Name > Sport-100 Helmet, Red < /Name >
  < StandardCost > 13.0863 < /StandardCost >
< /Products >
< /XMLProducts >

```

这里只显示了第一个 products 元素。实际的 XML 文件应包含 Northwind 数据库中 products 表的所有 products 元素。

### 转换关系数据

这看起来非常简单，因为只有一个表。但对于关系数据，例如 DataSet 中有多个 DataTables 和 Relations，该怎么办？其工作方式仍旧是这样。下面的示例使用了两个关系表：

```

private void button5_Click(object sender, EventArgs e)
{
  XmlDocument doc = new XmlDocument();
  DataSet ds = new DataSet("XMLProducts");
  SqlConnection conn = new SqlConnection(_connectString);
  SqlDataAdapter daProduct = new SqlDataAdapter
    ("SELECT Name, StandardCost,
     ProductCategoryID FROM SalesLT.Product", conn);
  SqlDataAdapter daCategory = new SqlDataAdapter
    ("SELECT ProductCategoryID, Name from SalesLT.ProductCategory", conn);
  //Fill DataSet from both SqlAdapters
  daProduct.Fill(ds, "Products");
  daCategory.Fill(ds, "Categories");
  //Add the relation
  ds.Relations.Add(ds.Tables["Categories"].Columns["ProductCategoryID"],
    ds.Tables["Products"].Columns["ProductCategoryID"]);
  //Write the Xml to a file so we can look at it later
  ds.WriteXml("Products.xml", XmlWriteMode.WriteSchema);
  //load data into grid
  dataGridView1.DataSource = ds.Tables[0];
  //create the XmlDataDocument
  doc = new XmlDataDocument(ds);
  //Select the productname elements and load them in the grid
}

```

```

XmlNodeList nodeList = doc.SelectNodes("//XMLProducts/Products");
textBox1.Text = "";
foreach (XmlNode node in nodeList)
{
    textBox1.Text += node.InnerXml + "\r\n";
}
}

```

在这个示例中，在 XMLAuthors 数据集中创建了两个 DataTable: Products 和 Categories。在两个表的 ProductCategoryID 列上创建一个新的关系。

执行与上一个示例相同的 WriteXml()方法调用，得到如下 XML 文件(SuppProd.xml):

```

< ?xml version="1.0" standalone="yes"? >
< XMLProducts >
  < xs:schema id="XMLProducts" xmlns=""
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:msdata="urn:schemas-microsoft-com:xml-msdata" >
    < xs:element name="XMLProducts" msdata:IsDataSet="true"
      msdata:UseCurrentLocale="true" >
      < xs:complexType >
        < xs:choice minOccurs="0" maxOccurs="unbounded" >
          < xs:element name="Products" >
            < xs:complexType >
              < xs:sequence >
                < xs:element name="Name" type="xs:string"
                  minOccurs="0" / >
                < xs:element name="StandardCost"
                  type="xs:decimal" minOccurs="0" / >
                < xs:element name="ProductCategoryID"
                  type="xs:int" minOccurs="0" / >
              < /xs:sequence >
            < /xs:complexType >
          < /xs:element >
          < xs:element name="Categories" >
            < xs:complexType >
              < xs:sequence >
                < xs:element name="ProductCategoryID"
                  type="xs:int" minOccurs="0" / >
                < xs:element name="Name" type="xs:string"
                  minOccurs="0" / >
              < /xs:sequence >
            < /xs:complexType >
          < /xs:element >
        < /xs:choice >
      < /xs:complexType >
    < xs:unique name="Constraint1" >
      < xs:selector xpath="."//Categories" / >
      < xs:field xpath="ProductCategoryID" / >
    < /xs:unique >
    < xs:keyref name="Relation1" refer="Constraint1" >
      < xs:selector xpath="."//Products" / >
      < xs:field xpath="ProductCategoryID" / >
    < /xs:keyref >
  < /xs:schema >
  < Products >
    < Name > HL Road Frame - Black, 58 < /Name >
    < StandardCost > 1059.3100 < /StandardCost >
    < ProductCategoryID > 18 < /ProductCategoryID >

```

```
< /Products >
< Products >
  < Name > HL Road Frame - Red, 58 < /Name >
  < StandardCost > 1059.3100 < /StandardCost >
  < > 18 < /ProductCategoryID >
< /Products >
< /XMLProducts >
```

该模式包含 DataSet 中的两个 DataTable。数据包含两个表中的所有数据。为简洁起见，这里只显示了第一个 Products 和 ProductCategory 记录。与以前一样，使用正确的 XmlWriteMode 参数可以只保存模式或数据。

28.7.2 把 XML 文档转换为 ADO.NET 数据

现在有一个 XML 文档，准备把它转换为 ADO.NET 的 DataSet。这样就可以把 XML 加载到数据库中，或者把数据绑定到 .NET 数据控件上，例如 DataGrid。此时可以把 XML 文档用作数据库，完全消除数据库的系统开销。如果数据比较少，这是完全有可能的。看看下面这些代码(位于 ADOSample5 中)：

```
private void button7_Click(object sender, EventArgs e)
{
    //create the DataSet
    DataSet ds = new DataSet("XMLProducts");
    //read in the xml document
    ds.ReadXml("Products.xml");
    //load data into grid
    dataGridView1.DataSource = ds.Tables[0];
    textBox1.Text = "";
    foreach (DataTable dt in ds.Tables)
    {
        textBox1.Text += dt.TableName + "\r\n";
        foreach (DataColumn col in dt.Columns)
        {
            textBox1.Text += "\t" + col.ColumnName + " - "
                + col.DataType.FullName + "\r\n";
        }
    }
}
```

这很简单。实例化一个新 DataSet 对象，调用 ReadXml()方法，把 XML 放在 DataSet 的一个 DataTable 中。与 WriteXml()方法一样，ReadXml()的参数是 XmlReadMode。ReadXml()还可以使用 XmlReadMode 中的更多选项(如表 28-9 所示)。

表 28-9

选 项	说 明
Auto	把 XmlReadMode 设置为最合适的值。如果数据是 DiffGram 格式，就选择 DiffGram；如果已经读取了模式，或者检测到某个内联模式，就选择 ReadSchema。如果没有为 DataSet 指定模式，也没有检测到内联模式，就选择 IgnoreSchema
DiffGram	读取 DiffGram，把变化应用到 DataSet 上
Fragment	读取包含 XDR 模式片断的文档，例如 SQL Server 创建的类型

(续表)

选 项	说 明
IgnoreSchema	忽略任何找到的内联模式，把数据读入当前的 DataSet 模式中，如果数据与 DataSet 模式不匹配，就删除它
InferSchema	忽略任何内联模式，根据 XML 文档中的数据创建模式。如果 DataSet 中已经有一个模式，就使用该模式，如果需要，可以用其他列或表格来扩展它。如果存在一个列，但其数据类型不符，就会抛出一个异常
ReadSchema	读取内联模式，加载数据。不重写 DataSet 中的模式，但如果内联模式中的表已经存在于 DataSet 中，就抛出一个异常

这里也有 ReadXmlSchema()方法，它读取独立的模式，创建相应的表、列和关系。如果模式没有和数据建立关联，就可以使用 ReadXmlSchema()方法。该方法也有 4 个重载方法：其参数分别是带有文件和路径名的字符串、基于 Stream 的对象、基于 TextReader 的对象和基于 XmlReader 的对象。

要说明如何正确地创建数据表，迭代表和列，在文本框中显示名称。可以把它与最初的 Northwind 数据库比较一下，看看这些内容是否有改变。最后一个 foreach 循环执行这个任务。

图 28-6 显示了结果。

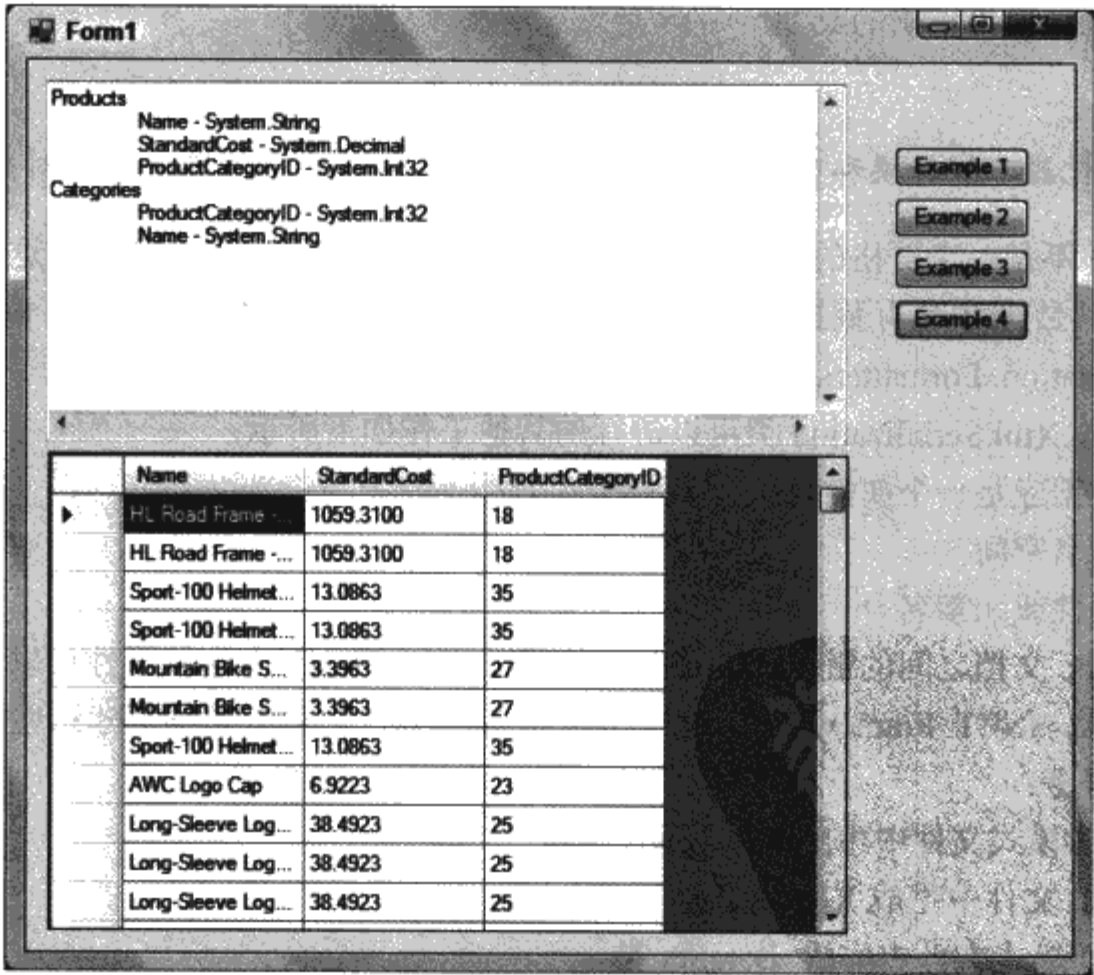


图 28-6

从列表框中可以看出，DataTables 中包含的列都有正确的名称和数据类型。

还要注意，最后的两个示例都没有在数据库传入或传出任何数据，所以没有定义 SqlDataAdapter 或 SqlConnection。这说明了 System.Xml 命名空间和 ADO.NET 的灵活性：



可以用多种格式查看相同的数据。如果需要进行转换,以 HTML 格式显示数据,或者需要把数据绑定到网格上,就应获取这些数据,用一个方法调用,把它们以需要的格式显示出来。

## 28.8 在 XML 中串行化对象

串行化是把一个对象存入磁盘的过程。应用程序的另一部分,甚至另一个应用程序都可以并行化对象,使它的状态与串行化之前相同。.NET Framework 为此提供了两种方式。

本节将介绍 `System.Xml.Serialization` 命名空间。它包含的类可用于把对象串行化为 XML 文档或流。这表示对象的公共属性和公共字段将转换为 XML 元素和/或属性。

`System.Xml.Serialization` 命名空间中最重要的是 `XmlSerializer`。要串行化对象,首先需要实例化一个 `XmlSerializer` 对象,指定要串行化的对象类型,然后实例化一个流/写入器对象,把文件写入流/文档。最后一步是在 `XmlSerializer` 上调用 `Serialize()` 方法,给它传送流/写入器对象和要串行化的对象。

被串行化的数据可以为基本类型的数据、字段、数组、`XmlElement`s 和 `XmlAttribute` 对象格式的内嵌 XML。

为了从 XML 文档中并行化对象,应执行上述过程的逆过程。即创建一个流/读取器对象和一个 `XmlSerializer` 对象,然后给 `Deserialize()` 方法传送该流/读取器对象。这个方法返回并行化的对象,但需要转换为正确的类型。

注意:

XML 串行化器不能转换私有数据,只能转换公共数据,它也不能串行化对象图表。

但是,这并不是一个严格的限制。对类进行仔细设计,就很容易避免突破这个限制。如果需要串行化公共数据和私有数据,以及包含许多嵌套对象的对象图形,就可以使用 `System.Runtime.Serialization.Formatters.Binary` 命名空间。

使用 `System.Xml.Serialization` 类可以进行的其他工作如下所示:

- 确定数据应是一个属性还是元素
- 指定命名空间
- 改变属性或元素名

对象和 XML 文档之间的链接是给类加上注释的定制 C# 属性,这些属性可以告诉串行化程序如何写入数据。.NET Framework 中有一个工具 `xsd.exe`,它可以帮助创建这些属性。`xsd.exe` 可以完成如下任务:

- 从 XDR 模式文件中生成一个 XML 模式
- 从 XML 文件中生成 XML 模式
- 从 XSD 模式文件中生成 `DataSet` 类
- 生成运行库类,运行库类包含 `XmlSerialization` 的定制属性
- 从已经开发出来的类中生成 XSD
- 限制在代码中创建的元素
- 确定生成代码的编程语言(C#、VB 或 JScript)

- 在编译好的程序集中创建类中的模式

参见 Framework 文档说明书, 了解 xsd.exe 命令行选项的详细内容。

尽管 xsd.exe 具备这些功能, 但不一定用它为串行化创建类。这个过程是很简单的。下面介绍一个简单的应用程序, 它串行化一个类。示例代码的起始部分比较简单, 创建一个新的 Product 对象 pd, 并给它填充一些数据:

```
private void button1_Click(object sender, System.EventArgs e)
{
    //new products object
    Products pd=new Products();
    //set some properties
    pd.ProductID=200;
    pd.CategoryID=100;
    pd.Discontinued=false;
    pd.ProductName="Serialize Objects";
    pd.QuantityPerUnit="6";
    pd.ReorderLevel=1;
    pd.SupplierID=1;
    pd.UnitPrice=1000;
    pd.UnitsInStock=10;
    pd.UnitsOnOrder=0;
}
```

XmlSerializer 类的 Serialize 方法实际执行串行化, 它有 9 个重载方法。一个必需的参数是要写入数据的流, 可以是 Stream、TextWriter 或 XmlWriter。在本例中, 创建了一个基于 TextWriter 的对象 tr。接着创建了基于 XmlSerializer 的对象 sr。XmlSerializer 需要知道要串行化的对象的类型信息, 所以对要串行化的类型使用 typeof 关键字。在创建 sr 对象后, 调用 Serialize() 方法, 其参数是 tr(基于 Stream 的对象)和要串行化的对象, 在本例中是 pd。确保完成后关闭该数据流。

```
//new TextWriter and XmlSerializer
TextWriter tr = new StreamWriter("serialprod.xml");
XmlSerializer sr = new XmlSerializer(typeof(Product));
//serialize object
sr.Serialize(tr, pd);
tr.Close();
webBrowser1.Navigate(AppDomain.CurrentDomain.BaseDirectory + "//serialprod.xml");
```

下面介绍 Products 类, 即要串行化的类。这个类与以前编写的其他类的唯一区别是给它增加了 C# 属性。这些属性中的 XmlRootAttribute 和 XmlElementAttribute 类从 System.Attribute 类继承而来。不要把这些属性与 XML 文档中的属性相混淆。C# 属性仅是一些声明信息, 在运行期间可以由 CLR 获取(详见第 7 章)。在本例中, 添加一些描述如何串行化对象的属性:

```
//class that will be serialized.
//attributes determine how object is serialized
[System.Xml.Serialization.XmlRootAttribute()]
public class Product {
    private int prodId;
    private string prodName;
    private int suppId;
    private int catId;
    private string qtyPerUnit;
    private Decimal unitPrice;
    private short unitsInStock;
    private short unitsOnOrder;
```

```
private short reorderLvl;
private bool discount;
private int disc;

//added the Discount attribute
[XmlAttributeAttribute(AttributeName="Discount")]
public int Discount {
    get {return disc;}
    set {disc=value;}
}

[XmlElementAttribute()]
public int ProductID {
    get {return prodId;}
    set {prodId=value;}
}

[XmlElementAttribute()]
public string ProductName {
    get {return prodName;}
    set {prodName=value;}
}

[XmlElementAttribute()]
public int SupplierID {
    get {return suppId;}
    set {suppId=value;}
}

[XmlElementAttribute()]
public int CategoryID {
    get {return catId;}
    set {catId=value;}
}

[XmlElementAttribute()]
public string QuantityPerUnit {
    get {return qtyPerUnit;}
    set {qtyPerUnit=value;}
}

[XmlElementAttribute()]
public Decimal UnitPrice {
    get {return unitPrice;}
    set {unitPrice=value;}
}

[XmlElementAttribute()]
public short UnitsInStock {
    get {return unitsInStock;}
    set {unitsInStock=value;}
}

[XmlElementAttribute()]
public short UnitsOnOrder {
    get {return unitsOnOrder;}
    set {unitsOnOrder=value;}
}

[XmlElementAttribute()]
public short ReorderLevel {
    get {return reorderLvl;}
}
```



```

        set {reorderLvl=value;}
    }

    [XmlElementAttribute()]
    public bool Discontinued {
        get {return discount;}
        set {discount=value;}
    }

    public override string ToString()
    {
        StringBuilder outText = new StringBuilder();
        outText.Append(prodId);
        outText.Append("");
        outText.Append(prodName);
        outText.Append("");
        outText.Append(unitPrice);
        return outText.ToString();
    }
}

```

在 `Products` 类定义前面的属性中调用的 `XmlRootAttribute()` 把这个类标识为根元素(在 XML 文件中, 由串行化生成的)。包含 `XmlElementAttribute()` 的属性把该属性下面的成员看做表示一个 XML 元素。

注意重写了 `ToString()` 方法, 它提供了运行并行化示例时显示在消息框中的字符串。

查看一下刚才在串行化过程中创建的 XML 文档, 就会发现它与前面创建的其他 XML 文档非常类似。这就是本练习的目的。下面就是这个文档:

```

<?xml version="1.0" encoding="utf-8"?>
<Products xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xmlns:xsd="http://www.w3.org/2001/XMLSchema" Discount="0">
  <ProductID>200</ProductID>
  <ProductName>Serialize Objects</ProductName>
  <SupplierID>1</SupplierID>
  <CategoryID>100</CategoryID>
  <QuantityPerUnit>6</QuantityPerUnit>
  <UnitPrice>1000</UnitPrice>
  <UnitsInStock>10</UnitsInStock>
  <UnitsOnOrder>0</UnitsOnOrder>
  <ReorderLevel>1</ReorderLevel>
  <Discontinued>false</Discontinued>
</Products>

```

这里没有任何不寻常的地方。可以以 XML 文档的任何方式来使用这个文档。可以对它进行转换, 以 HTML 格式显示它, 使用 ADO.NET 将其加载到 `DataSet` 中, 用它加载 `XmlDocument`, 或者像在该示例中那样, 对它进行并行化, 创建一个对象, 该对象的状态与串行化前 `pd` 的状态一样(这就是第二个按钮的作用)。

接着添加另一个按钮事件处理程序, 并行化一个基于 `Products` 的新对象 `newPd`。这次使用 `FileStream` 对象读取 XML:

```

private void button2_Click(object sender, System.EventArgs e)
{
    //create a reference to products type
    Products newPd;
    //new filestream to open serialized object
    FileStream f=new FileStream("serialprod.xml", FileMode.Open);
}

```

再传入 `Product` 的类型信息，创建一个新 `XmlSerializer`。然后就可以调用 `Deserialize()` 方法。注意在创建 `newPd` 对象时，仍需要进行显式的类型转换。此时 `newPd` 与 `pd` 的状态完全一样：

```
//new serializer
XmlSerializer newSr=new XmlSerializer(typeof(Products));
//deserialize the object
newPd=(Products)newSr.Deserialize(f);
f.Close();
MessageBox.Show(newPd.ToString());
}
```

消息框应显示产品 ID、产品名称和刚才并行化的对象的单价。这是因为使用了在 `Product` 类中实现的 `ToString()` 方法。

如果有派生的类和可能返回一个数组的属性，也可以使用 `XmlSerializer`。下面介绍一个解决这些问题的复杂示例。

首先定义 3 个新类 `Product`、`BookProduct` (派生于 `Product`) 和 `Inventory` (它包含其他两个类)。注意又重写了 `ToString()` 方法，这次要列出 `Inventory` 类中的项：

```
public class BookProduct : Product
{
    private string isbnNum;
    public BookProduct() {}
    public string ISBN
    {
        get {return isbnNum;}
        set {isbnNum=value;}
    }
}
public class Inventory
{
    private Product[] stuff;
    public Inventory() {}
    //need to have an attribute entry for each data type
    [XmlAttribute("Prod",typeof(Product)),
    XmlArrayItem("Book",typeof(BookProduct))]
    public Product[] InventoryItems
    {
        get {return stuff;}
        set {stuff=value;}
    }
    public override string ToString()
    {
        StringBuilder outText = new StringBuilder();
        foreach (Product prod in stuff)
        {
            outText.Append(prod.ProductName);
            outText.Append("\r\n");
        }
        return outText.ToString();
    }
}
```

在此我们只对 `Inventory` 类感兴趣。如果串行化这个类，就需要插入一个属性，该属性为每个要添加到数组中的类型包含一个 `XmlAttribute` 构造函数。注意，`XmlAttribute` 是由 `XmlAttributeAttribute` 类表示的 .NET 属性名。



这些构造函数的第一个参数是在串行化过程中创建的 XML 文档中的元素名。如果不使用 `ElementName` 参数,元素的名称就会与对象类型名相同(在本例中,就是 `Product` 和 `BookProduct`)。必须指定的第二个参数是对象的类型。

如果属性返回一个对象数组或基本类型的数组,还要使用 `XmlArrayAttribute` 类。因为要在数组中返回不同的类型,所以使用 `XmlArrayItemAttribute`,它允许进行更高级别的控制。

在 `button4_Click` 事件处理程序中,创建一个新的 `Product` 对象和一个新的 `BookProduct` 对象(`newProd` 和 `newBook`)。给每个对象的各种属性添加数据,再把这些对象添加到一个 `Product` 数组中。把这个数组作为参数,创建一个新的 `Inventory` 对象,然后串行化 `Inventory` 对象,以便在以后重新创建它:

```
private void button4_Click(object sender, EventArgs e)
{
    //create the XmlAttributes boject
    XmlAttributes attrs = new XmlAttributes();
    //add the types of the objects that will be serialized
    attrs.XmlElements.Add(new XmlElementAttribute("Book", typeof(BookProduct)));
    attrs.XmlElements.Add(new XmlElementAttribute("Product", typeof(Product)));
    XmlAttributeOverrides attrOver = new XmlAttributeOverrides();
    //add to the attributes collection
    attrOver.Add(typeof(Inventory), "InventoryItems", attrs);
    //create the Product and Book objects
    Product newProd = new Product();
    BookProduct newBook = new BookProduct();
    newProd.ProductID = 100;
    newProd.ProductName = "Product Thing";
    newProd.SupplierID = 10;
    newBook.ProductID = 101;
    newBook.ProductName = "How to Use Your New Product Thing";
    newBook.SupplierID = 10;
    newBook.ISBN = "123456789";
    Product[] addProd = { newProd, newBook };
    Inventory inv = new Inventory();
    inv.InventoryItems = addProd;
    TextWriter tr = new StreamWriter("inventory.xml");
    XmlSerializer sr = new XmlSerializer(typeof(Inventory), attrOver);
    sr.Serialize(tr, inv);
    tr.Close();
    webBrowser1.Navigate(AppDomain.CurrentDomain.BaseDirectory + "//inventory.xml");
}
```

XML 文档如下所示:

```
<?xml version="1.0" encoding="utf-8"?>
<Inventory xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Product Discount="0">
    <ProductID>100</ProductID>
    <ProductName>Product Thing</ProductName>
    <SupplierID>10</SupplierID>
    <CategoryID>0</CategoryID>
    <UnitPrice>0</UnitPrice>
    <UnitsInStock>0</UnitsInStock>
    <UnitsOnOrder>0</UnitsOnOrder>
    <ReorderLevel>0</ReorderLevel>
    <Discontinued>false</Discontinued>
```

```

</Product>
<Book Discount="0">
  <ProductID>101</ProductID>
  <ProductName>How to Use Your New Product Thing</ProductName>
  <SupplierID>10</SupplierID>
  <CategoryID>0</CategoryID>
  <UnitPrice>0</UnitPrice>
  <UnitsInStock>0</UnitsInStock>
  <UnitsOnOrder>0</UnitsOnOrder>
  <ReorderLevel>0</ReorderLevel>
  <Discontinued>>false</Discontinued>
  <ISBN>123456789</ISBN>
</Book>
</Inventory>

```

button2\_Click 事件处理程序执行 Inventory 对象的并行化。注意在新建的 newInv 对象中，我们迭代了数组，以说明其数据保持不变：

```

private void button2_Click(object sender, System.EventArgs e)
{
    Inventory newInv;
    FileStream f = new FileStream("order.xml", FileMode.Open);
    XmlSerializer newSr = new XmlSerializer(typeof(Inventory));
    newInv = (Inventory)newSr.Deserialize(f);
    foreach (Product prod in newInv.InventoryItems)
        listBox1.Items.Add(prod.ProductName);
    f.Close();
}

```

### 不能访问源代码的串行化

这些代码都很好地发挥了作用，但如果不能访问已经串行化的类型的源代码，该怎么办？如果没有源代码，就不能添加属性。此时可以采用另一种方式。可以使用 XmlAttributes 类和 XmlAttributeOverrides 类，这两个类可以完成刚才的任务，但不需要添加属性。下面的代码说明了这两个类的工作方式。

对于这个示例，假定 Inventory、Product 和派生的 BookProduct 类在一个单独的 DLL 中，而且没有源代码。Product 和 BookProduct 类与前面的示例相同，但应注意 Inventory 类中没有添加属性：

```

public class Inventory
{
    private Product[] stuff;
    public Inventory() {}
    public Product[] InventoryItems
    {
        get { return stuff; }
        set { stuff = value; }
    }
}

```

下面处理 button1\_Click() 事件处理程序中的串行化：

```

private void button1_Click(object sender, System.EventArgs e)
{

```

串行化过程的第一步是创建一个 `XmlAttributes` 对象，为每个要重写的数据类型创建一个 `XmlElementAttribute` 对象：

```
XmlAttributes attrs = new XmlAttributes();
attrs.XmlElements.Add(new XmlElementAttribute("Book",typeof(BookProduct)));
attrs.XmlElements.Add(new XmlElementAttribute("Product",typeof(Product)));
```

从中可以看出，我们给 `XmlAttributes` 类的 `XmlElements` 集合添加了新的 `XmlElementAttribute`。`XmlAttributes` 类的属性对应于可以应用的属性，前面示例中的 `XmlArray` 和 `XmlArrayItems` 仅是其中的几个属性而已。现在有一个 `XmlAttributes` 对象，并在 `XmlElements` 集合中添加了两个基于 `XmlElementAttribute` 的对象。

接着创建 `XmlAttributeOverrides` 对象：

```
XmlAttributeOverrides attrOver = new XmlAttributeOverrides();
attrOver.Add(typeof(Inventory),"InventoryItems",attrs);
```

这个类的 `Add()` 方法有两个重载方法。第一个重载方法的参数是要重写的对象的类型信息和前面创建的 `XmlAttributes` 对象，本例中使用了第二个重载方法，其参数也是一个字符串值，该字符串是重写对象的成员。在本例中，要重写 `Inventory` 类中的 `InventoryItems` 成员。

下面把添加的 `XmlAttributeOverrides` 对象作为参数，创建 `XmlSerializer` 对象。现在 `XmlSerializer` 知道我们要重写的类和需要为这些类型返回的内容。

```
//create the Product and Book objects
Product newProd = new Product();
BookProduct newBook = new BookProduct();
newProd.ProductID = 100;
newProd.ProductName = "Product Thing";
newProd.SupplierID = 10;
newBook.ProductID = 101;
newBook.ProductName = "How to Use Your New Product Thing";
newBook.SupplierID = 10;
newBook.ISBN = "123456789";
Product[] addProd = {newProd,newBook};

Inventory inv = new Inventory();
inv.InventoryItems = addProd;
TextWriter tr = new StreamWriter("inventory.xml");
XmlSerializer sr = new XmlSerializer(typeof(Inventory),attrOver);
sr.Serialize(tr,inv);
tr.Close();
}
```

如果执行 `Serialize` 方法，将得到如下 XML 输出：

```
<?xml version="1.0" encoding="utf-8"?>
<Inventory xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Product Discount="0">
    <ProductID>100</ProductID>
    <ProductName>Product Thing</ProductName>
    <SupplierID>10</SupplierID>
    <CategoryID>0</CategoryID>
    <UnitPrice>0</UnitPrice>
    <UnitsInStock>0</UnitsInStock>
    <UnitsOnOrder>0</UnitsOnOrder>
```



```

    <ReorderLevel>0</ReorderLevel>
    <Discontinued>>false</Discontinued>
  </Product>
  <Book Discount="0">
    <ProductID>101</ProductID>
    <ProductName>How to Use Your New Product Thing</ProductName>
    <SupplierID>10</SupplierID>
    <CategoryID>0</CategoryID>
    <UnitPrice>0</UnitPrice>
    <UnitsInStock>0</UnitsInStock>
    <UnitsOnOrder>0</UnitsOnOrder>
    <ReorderLevel>0</ReorderLevel>
    <Discontinued>>false</Discontinued>
    <ISBN>123456789</ISBN>
  </Book>
</Inventory>

```

可以看出,得到的 XML 与前面的示例完全相同。为了并行化对象,重新创建基于 Inventory 的对象,需要创建在串行化对象时创建的 XmlAttributes、XmlElementAttribute 和 XmlAttributeOverrides 对象。之后就可以读取 XML,像以前那样重新创建 Inventory 对象了。下面的代码并行化 Inventory 对象:

```

private void button2_Click(object sender, System.EventArgs e)
{
    //create the new XmlAttributes collection
    XmlAttributes attrs=new XmlAttributes();
    //add the type information to the elements collection
    attrs.XmlElements.Add(new XmlElementAttribute("Book",typeof(BookProduct)));
    attrs.XmlElements.Add(new XmlElementAttribute("Product",typeof(Product)));

    XmlAttributeOverrides attrOver=new XmlAttributeOverrides();
    //add to the Attributes collection
    attrOver.Add(typeof(Inventory),"InventoryItems",attrs);

    //need a new Inventory object to deserialize to
    Inventory newInv;

    //deserialize and load data into the listbox from deserialized object
    FileStream f=new FileStream("../..\\..\\..\\inventory.xml", FileMode.Open);
    XmlSerializer newSr=new XmlSerializer(typeof(Inventory),attrOver);

    newInv=(Inventory)newSr.Deserialize(f);
    if(newInv!=null)
    {
        foreach(Product prod in newInv.InventoryItems)
            listBox1.Items.Add(prod.ProductName);
    }
    f.Close();
}

```

注意,前几行代码与串行化对象所用的代码相同。

System.Xml.XmlSerialize 命名空间提供了一个功能非常强大的工具集,可以把对象串行化到 XML 中。把对象串行化和并行化到 XML 中替代了把对象保存为二进制格式,因此可以通过 XML 对对象进行其他处理。这将大大增强设计的灵活性。

## 28.9 小结

本章介绍了 .NET Framework 的 System.Xml 命名空间中的许多内容，其中包括如何使用基于 XMLReader 和 XmlWriter 的类快速读写 XML 文档，如何在 .NET 中执行 DOM，如何使用 DOM 的强大功能。XML 和 ADO.NET 实际上有非常密切的关系。DataSet 和 XML 文档仅是相同底层结构的两个不同视图而已。当然，我们还介绍了 XPath 和 XSL Transform，以及添加到 VS 中的调试功能。

最后，可以把对象串行化到 XML 中，还可以通过两个方法调用对其进行反串行化。

XML 是以后的几年中应用程序开发的一个重要部分。.NET Framework 提供了操作 XML 的丰富而强大的工具集。第 29 章介绍了如何使用 LINQ to XML。



# 第 29 章

## LINQ to XML

如第 27 章所述,在.NET Framework 3.5 中,最大、最令人激动的新增功能是添加到 C# 2008 中的.NET Language Integrated Query 架构(LINQ)。LINQ 根据在查询数据时所使用的最终数据存储方式有许多形式。第 27 章介绍了使用 LINQ to SQL 查询 SQL Server 数据库,本章介绍使用 LINQ 查询 XML 数据源。

本章内容如下:

- LINQ to XML 给表带来的变化
- System.Xml.Linq 命名空间中的新对象
- 如何使用 LINQ 查询 XML 文档
- 使用 LINQ 移动 XML 文档
- 使用 LINQ to SQL 和 LINQ to XML

Extensible Markup Language(XML)现在在世界范围内使用。Internet 或个人计算机上的许多应用程序都使用某种形式的 XML 运行或管理应用程序的进程。早期关于 XML 的图书把 XML 看作“另一个伟大的成就”。现在,它是“一个伟大的成就”。事实上,再也没有比 XML 更伟大的成就了。

微软公司通过多年的努力,使 XML 的使用在.NET 领域尽可能简单。在.NET Framework 的每个新版本中都增加了 XML 使用上的其他功能和改进。实际上,2005 年在洛杉矶召开的微软企业应用程序大会上,比尔·盖茨在其演讲中特别强调了微软公司在 XML 方面的诺言。他承诺,每年,XML 都将更进一步地深入 Windows 核心。查看一下.NET Framework,就会认同这个观点。

因此,本章介绍如何使用 LINQ to XML 查询 XML 文档。图 29-1 显示了 LINQ 在查询 XML 数据中的作用。

使用 LINQ to XML 时,第 27 章介绍的许多有关使用 LINQ to SQL 的内容都可以应用于本章。

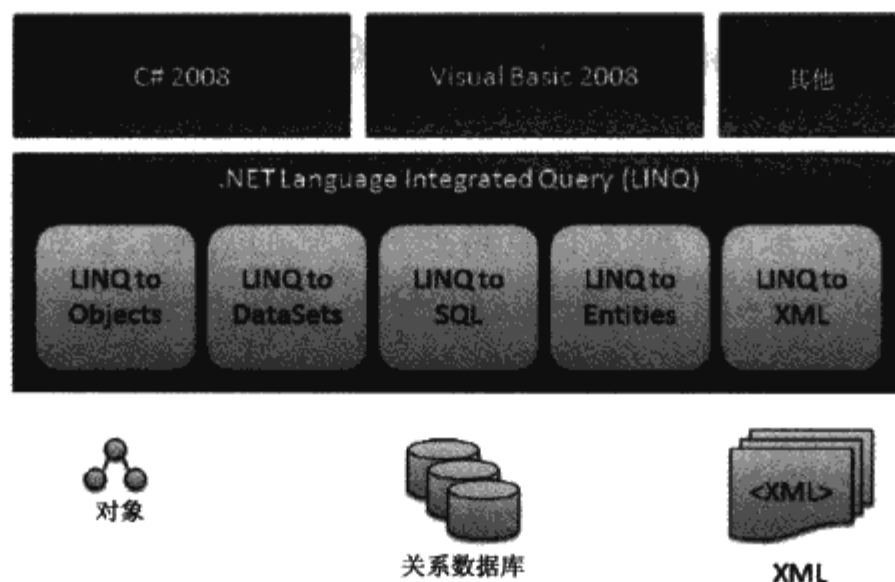


图 29-1

## 29.1 LINQ to XML 和 .NET 3.5

把 LINQ 引入 .NET Framework 3.5 中时，其重点是便于访问要在应用程序中处理的数据。在应用程序中，一个主要的数据存储形式是 XML，因此创建了 LINQ to XML。

在 LINQ to XML 推出之前，通过 `System.Xml` 使用 XML 并不是很容易实现。引入了 `System.Xml.Linq` 命名空间后，就可以利用一系列功能在代码中方便地处理 XML 了。

### 29.1.1 创建 XML 文档的新对象

在应用程序代码中创建 XML 时，许多开发人员都使用 `XmlDocument` 对象。这个对象可以创建 XML 文档，以层次结构的方式追加元素、属性和其他项。在 LINQ to XML 和 `System.Xml.Linq` 命名空间中，有一些新对象使 XML 文档的创建非常简单。

### 29.1.2 Visual Basic 2008 开辟了另一条道路

LINQ to XML 功能集的一个有趣的地方是，微软公司的 Visual Basic 2008 团队使 LINQ to XML 的功能在某些领域更进了一步。例如，在 C# 2008 中不能完成的某些任务可以在 Visual Basic 2008 中完成，包括把 XML 作为 Visual Basic 2008 的核心部分。XML 字面量现在是 Visual Basic 语言的一个部分，可以直接在代码中分析 XML 片段，所包含的 XML 不再作为字符串来处理。

### 29.1.3 命名空间和前缀

.NET Framework 2.0 忽略的一个问题是框架中的项如何在文档中包含 XML 命名空间和前缀。LINQ to XML 把这个问题当作 XML 的一个重要部分，而处理这类对象的功能很简单。

## 29.2 .NET Framework 3.5 中的新 XML 对象

即使 LINQ 查询功能不能在 .NET Framework 3.5 中使用, 这个版本提供的新 XML 对象也非常好, 可以处理 XML, 取代 DOM 的直接处理, 它们甚至可以独立于 LINQ。在新的 System.Xml.Linq 命名空间中, 有一系列新的 LINQ to XML 帮助对象, 使处理内存中的 XML 文档变成非常简单。

下面几节介绍这个新命名空间中的新对象。

**提示:**

本章的许多示例都使用了 Hamlet.xml 文件。这个文件在 <http://metalab.unc.edu/bosak/xml/eg/shaks200.zip> 上, 包含了莎士比亚的所有戏剧。

### 29.2.1 XDocument 对象

XDocument 替代了 .NET 3.5 之前的 XmlDocument 对象, 它更容易处理 XML 文档。XDocument 对象还和这个命名空间中的其他新对象一起使用, 例如 XNamespace、XComment、XElement 和 XAttribute 对象。

XDocument 对象的一个重要成员是 Load() 方法:

```
XDocument xdoc = XDocument.Load(@"C:\Hamlet.xml");
```

这个操作会把 Hamlet.xml 文件的内容加载为一个内存中的 XDocument 对象。还可以给 Load() 方法传送 TextReader 或 XmlReader 对象。现在就可以编程处理 XML 了:

```
XDocument xdoc = XDocument.Load(@"C:\Hamlet.xml");
Console.WriteLine(xdoc.Root.Name.ToString());
Console.WriteLine(xdoc.Root.HasAttributes.ToString());
```

输出的结果如下:

```
PLAY
False
```

另一个重要的成员是 Save() 方法, 它类似于 Load() 方法, 可以保存到一个物理磁盘位置, 或 TextWriter 或 XmlWriter 对象中:

```
XDocument xdoc = XDocument.Load(@"C:\Hamlet.xml");
xdoc.Save(@"C:\CopyOfHamlet.xml");
```

### 29.2.2 XElement 对象

一个常用的对象是 XElement。使用这个对象可以轻松地创建包含单个元素的对象, 该对象可以是 XML 文档本身, 甚至可以只是 XML 片段。例如, 下面的例子写入了一个 XML 元素及其相应的值:

```
XElement xe = new XElement("Company", "Lipper");
Console.WriteLine(xe.ToString());
```

在创建新的 XElement 对象时，可以定义该元素的名称和元素中使用的值。在这个例子中，元素的名称是<Company>，<Company>元素的值是 Lipper。在引用了 System.Xml.Linq 命名空间的控制台应用程序中运行它，得到的结果如下：

```
< Company > Lipper < /Company >
```

还可以使用多个 XElement 对象创建比较完整的 XML 文档，如下所示：

```
using System;
using System.Linq;
using System.Xml.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main()
        {
            XElement xe = new XElement("Company",
                new XElement("CompanyName", "Lipper"),
                new XElement("CompanyAddress",
                    new XElement("Address", "123 Main Street"),
                    new XElement("City", "St. Louis"),
                    new XElement("State", "MO"),
                    new XElement("Country", "USA")));
            Console.WriteLine(xe.ToString());
            Console.ReadLine();
        }
    }
}
```

运行这个应用程序，得到的结果如图 29-2 所示。

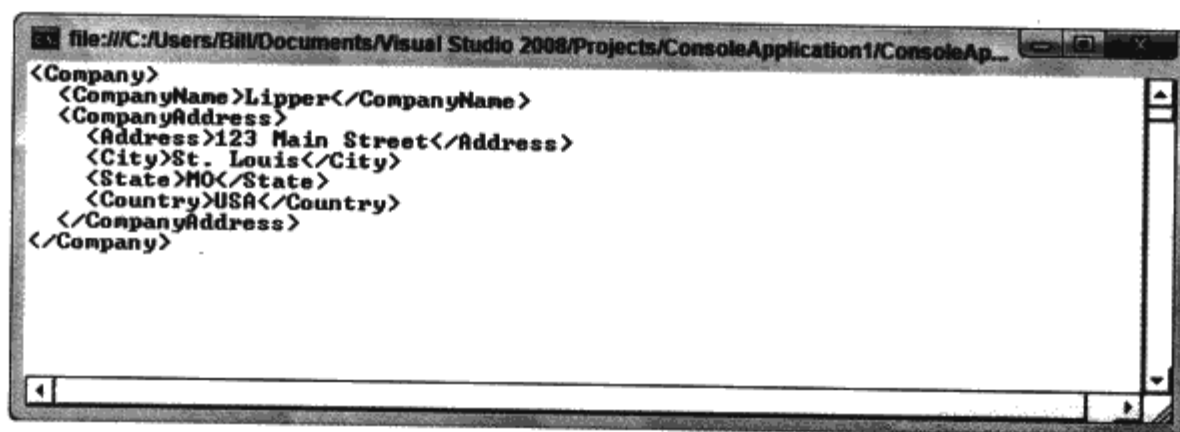


图 29-2

### 29.2.3 XNamespace 对象

XNamespace 对象表示 XML 命名空间，很容易应用于文档中的元素。例如，在前面的例子中，很容易给根元素应用一个命名空间：

```
using System;
using System.Linq;
using System.Xml.Linq;

namespace ConsoleApplication1
{
    class Class1
```

```

{
    static void Main()
    {
        XNamespace ns = "http://www.lippperweb.com/ns/1";

        XElement xe = new XElement(ns + "Company",
            new XElement("CompanyName", "Lipper"),
            new XElement("CompanyAddress",
                new XElement("Address", "123 Main Street"),
                new XElement("City", "St. Louis"),
                new XElement("State", "MO"),
                new XElement("Country", "USA")));

        Console.WriteLine(xe.ToString());

        Console.ReadLine();
    }
}

```

在这个例子中，创建了一个 XNamespace 对象，给它赋予 `http://www.lippperweb.com/ns/1`。之后，就可以在根元素 `<company>` 和 XElement 对象的实例中使用它了。

```
XElement xe = new XElement(ns + "Company", // ...
```

这会生成如图 29-3 所示的结果。

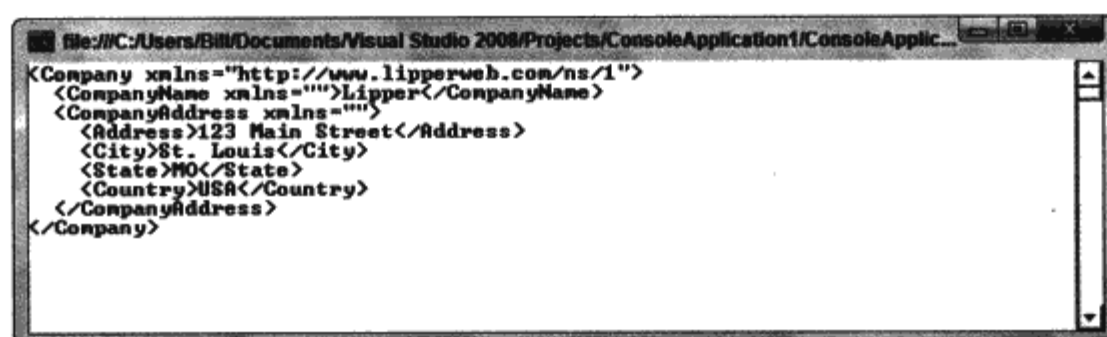


图 29-3

除了处理根元素之外，还可以把命名空间应用于所有的元素，如下所示：

```

using System;
using System.Linq;
using System.Xml.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main()
        {
            XNamespace ns1 = "http://www.lippperweb.com/ns/root";
            XNamespace ns2 = "http://www.lippperweb.com/ns/sub";

            XElement xe = new XElement(ns1 + "Company",
                new XElement(ns2 + "CompanyName", "Lipper"),
                new XElement(ns2 + "CompanyAddress",
                    new XElement(ns2 + "Address", "123 Main Street"),
                    new XElement(ns2 + "City", "St. Louis"),
                    new XElement(ns2 + "State", "MO"),

```



```

        new XElement(ns2 + "Country", "USA")));

    Console.WriteLine(xe.ToString());

    Console.ReadLine();
}
}
}

```

这会生成如图 29-4 所示的结果。

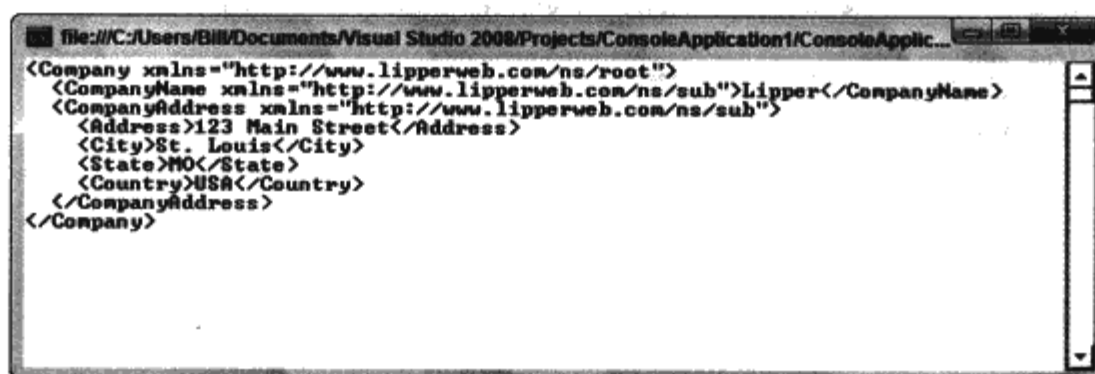


图 29-4

在这个例子中，子命名空间应用于指定的所有对象，但<Address>、<City>、<State>和<Country>元素除外，因为它们继承自其父对象<CompanyAddress>，而<CompanyAddress>有命名空间声明。

#### 29.2.4 XComment 对象

XComment 对象可以把 XML 注释添加到 XML 文档中。下面的例子说明了如何把一个注释添加到文档的开头：

```

using System;
using System.Linq;
using System.Xml.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main(string[] args)
        {
            XDocument xdoc = new XDocument();

            XComment xc = new XComment("Here is a comment.");
            xdoc.Add(xc);

            XElement xe = new XElement("Company",
                new XElement("CompanyName", "Lipper"),
                new XElement("CompanyAddress",
                    new XComment("Here is another comment."),
                    new XElement("Address", "123 Main Street"),
                    new XElement("City", "St. Louis"),
                    new XElement("State", "MO"),
                    new XElement("Country", "USA")));
            xdoc.Add(xe);
        }
    }
}

```

```

        Console.WriteLine(xdoc.ToString());

        Console.ReadLine();
    }
}

```

这个例子把包含两个 XML 注释的 XDocument 对象写到控制台上, 其中一个注释写到文档的开头, 另一个注释写到 <CompanyAddress> 元素内部, 其结果如图 29-5 所示。

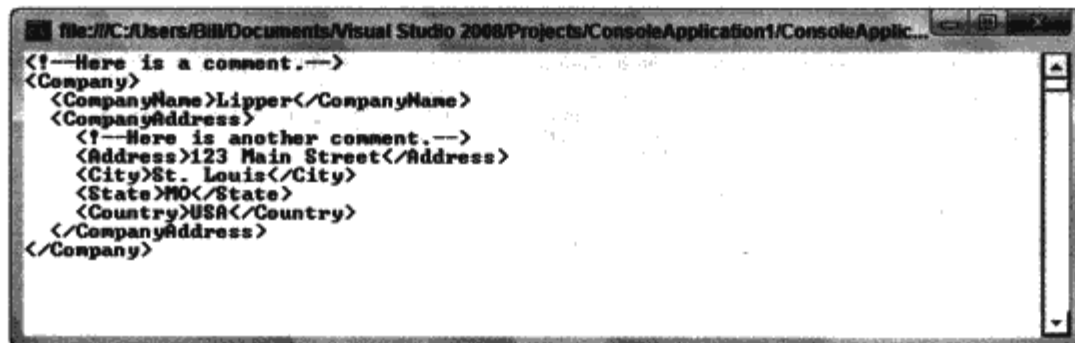


图 29-5

### 29.2.5 XAttribute 对象

除了元素之外, XML 的另一个要素是属性。添加和使用属性是通过 XAttribute 对象完成的。下面的例子演示了给根节点 <Customers> 添加一个属性:

```

using System;
using System.Linq;
using System.Xml.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main()
        {
            XElement xe = new XElement("Company",
                new XAttribute("MyAttribute", "MyAttributeValue"),
                new XElement("CompanyName", "Lipper"),
                new XElement("CompanyAddress",
                    new XElement("Address", "123 Main Street"),
                    new XElement("City", "St. Louis"),
                    new XElement("State", "MO"),
                    new XElement("Country", "USA")));

            Console.WriteLine(xe.ToString());

            Console.ReadLine();
        }
    }
}

```

这里把属性 MyAttribute 及其值 MyAttributeValue 添加到 XML 文档的根元素上, 结果如图 29-6 所示。

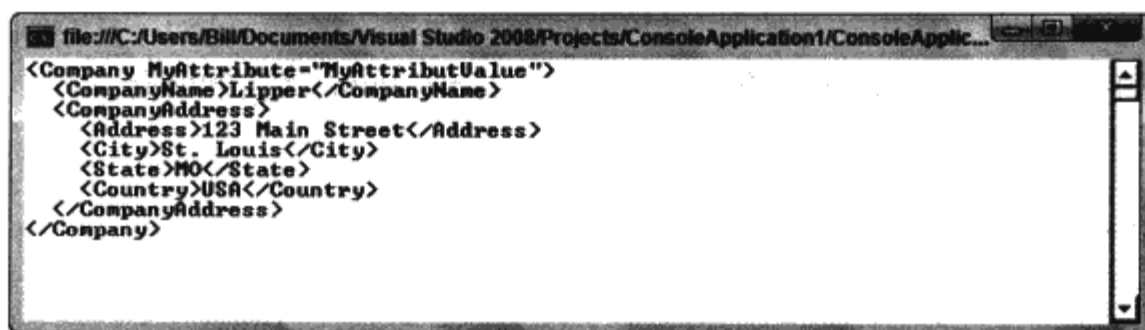


图 29-6

## 29.3 使用 LINQ 查询 XML 文档

现在可以把 XML 文档放在 XDocument 对象中，操作这个文档的各个部分。还可以使用 LINQ to XML 查询 XML 文档，操作其结果。

### 29.3.1 查询静态的 XML 文档

使用 LINQ to XML 查询静态的 XML 文档几乎不需要做任何工作。下面的例子就使用 hamlet.xml 文件和查询获得戏剧中的所有演员。每位演员都在 XML 文档中用 <PERSONA> 元素定义：

```

using System;
using System.Linq;
using System.Xml.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main(string[] args)
        {
            XDocument xdoc = XDocument.Load(@"C:\hamlet.xml");

            var query = from people in xdoc.Descendants("PERSONA")
                        select people.Value;
            Console.WriteLine("{0} Players Found", query.Count());
            Console.WriteLine();

            foreach (var item in query)
            {
                Console.WriteLine(item);
            }
            Console.ReadLine();
        }
    }
}

```

在这个例子中，XDocument 对象加载了一个物理 XML 文件 hamlet.xml，对文档的内容执行一个 LINQ 查询：

```

var query = from people in xdoc.Descendants("PERSONA")
            select people.Value;

```

`people` 对象表示在文档中找到的所有<PERSONA>元素。接着 `select` 语句获取这些元素的值。之后,使用 `Console.WriteLine()`方法写出通过 `query.Count()`找到的所有演员。再在 `foreach` 循环中把每一项写到控制台上。结果如下所示:

26 Players Found

```
CLAUDIUS, king of Denmark.
HAMLET, son to the late king, and nephew to the present king.
POLONIUS, lord chamberlain.
HORATIO, friend to Hamlet.
LAERTES, son to Polonius.
LUCIANUS, nephew to the king.
VOLTIMAND
CORNELIUS
ROSENCRANTZ
GUILDENSTERN
OSRIC
A Gentleman
A Priest.
MARCELLUS
BERNARDO
FRANCISCO, a soldier.
REYNALDO, servant to Polonius.
Players.
Two Clowns, grave-diggers.
FORTINBRAS, prince of Norway.
A Captain.
English Ambassadors.
GERTRUDE, queen of Denmark, and mother to Hamlet.
OPHELIA, daughter to Polonius.
Lords, Ladies, Officers, Soldiers, Sailors, Messengers, and other Attendants.
Ghost of Hamlet' s Father.
```

### 29.3.2 查询动态的 XML 文档

目前, Internet 上有许多动态的 XML 文档。给指定的 URL 端点发送一个请求, 就会找到博客种子、podcast 种子等许多提供 XML 文档的内容。这些种子可以在浏览器上查看, 或者通过 RSS 集合器查看, 或用作纯粹的 XML。

```
using System;
using System.Linq;
using System.Xml.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main()
        {
            XDocument xdoc =
                XDocument.Load(@"http://geekswithblogs.net/evjen/Rss.aspx");
            var query = from rssFeed in xdoc.Descendants("channel")
                        select new
                        {
                            Title = rssFeed.Element("title").Value,
                            Description = rssFeed.Element("description").Value,
                            Link = rssFeed.Element("link").Value,
```

```

    };

    foreach (var item in query)
    {
        Console.WriteLine("TITLE: " + item.Title);
        Console.WriteLine("DESCRIPTION: " + item.Description);
        Console.WriteLine("LINK: " + item.Link);
    }
    Console.WriteLine();

    var queryPosts = from myPosts in xdoc.Descendants("item")
        select new
        {
            Title = myPosts.Element("title").Value,
            Published =
                DateTime.Parse(
                    myPosts.Element("pubDate").Value),
            Description =
                myPosts.Element("description").Value,
            Url = myPosts.Element("link").Value,
            Comments = myPosts.Element("comments").Value
        };
    foreach (var item in queryPosts)
    {
        Console.WriteLine(item.Title);
    }
    Console.ReadLine();
}
}
}

```

在这段代码中，XDocument 对象的 Load() 方法指向一个 URL，从该 URL 中检索 XML。第一个查询提取种子中 <channel> 元素的所有主要子元素，创建新对象 Title、Description 和 Link，以获取这些子元素的值。

之后，运行一个 foreach 语句，迭代该查询找到的所有项，结果如下：

```

TITLE: Bill Evjen's Blog
DESCRIPTION: Code, Life and Community
LINK: http://geekswithblogs.net/evjen/Default.aspx

```

第二个查询遍历所有的 <item> 元素及其各个子元素(这些都是在博客中找到的博客项)。尽管找到的许多项都放在属性中，但在 foreach 循环中只使用了 Title 属性。这个查询的部分结果如下所示：

```

AJAX Control Toolkit Controls Grayed Out - HOW TO FIX
Welcome .NET 3.5!
Visual Studio 2008 Released
IIS 7.0 Rocks the House!
Word Issue - Couldn't Select Text
Microsoft Releases XML Schema Designer CTP1
Silverlight Book
Microsoft Tafiti as a beta
ReSharper on Visual Studio 2008
Windows Vista Updates for Performance and Reliability Issues
New Version of ODP.NET for .NET 2.0 Released as Beta Today
First Review of Profess
Go to MIX07 for free!

```



```

Microsoft Surface and the Future of Home Computing?
Alas my friends - I'm *not* TechEd bound
New Book - Professional VB 2005 with .NET 3.0!
An article showing Oracle and .NET working together
My Latest Book - Professional XML
CISCO VPN Client Software on Windows Vista
Server-Side Excel Generation
Scott Guthrie Gives Short Review of Professional ASP.NET 2.0 SE
Windows Forms Additions in the Next Version of .NET
Tag, I'm It

```

## 29.4 处理 XML 文档

如果处理了 XML 文档 hamlet.xml, 就会注意到该文件相当大。在本章中, 查询 XML 文档有两种方式, 但下面几节介绍 XML 文档的读写操作。

### 29.4.1 读取 XML 文档

前面介绍了如何使用 LINQ 查询语句查询 XML 文档, 如下所示:

```

var query = from people in xdoc.Descendants("PERSONA")
            select people.Value;

```

这个查询返回文档中的所有演员。使用 XDocument 对象的 Element() 方法, 还可以获取 XML 文档中的特定值。例如, 继续使用 hamlet.xml 文档, 下面的 XML 片段说明了如何在 XML 文档中表示标题:

```

< ?xml version="1.0"? >

< PLAY >
  < TITLE > The Tragedy of Hamlet, Prince of Denmark < /TITLE >

  <!-- XML removed for clarity -->

< /PLAY >

```

可以看出, <TITLE> 元素是 <PLAY> 元素的一个嵌套元素。在控制台应用程序中使用下面的代码可以获得标题:

```

XDocument xdoc = XDocument.Load(@"C:\hamlet.xml");
Console.WriteLine(xdoc.Element("PLAY").Element("TITLE").Value);

```

这段代码把标题 The Tragedy of Hamlet, Prince of Denmark 写到控制台屏幕上。在代码中, 可以使用两个 Element() 方法调用进入 XML 文档的层次结构, 第一个 Element() 方法调用 <PLAY> 元素, 第二个 Element() 方法调用嵌套在 <PLAY> 元素中的 <TITLE> 元素。

仔细查看 hamlet.xml 文档, 会发现它使用 <PERSONA> 元素定义了一个很大的演员列表:

```

< ?xml version="1.0"? >

< PLAY >
  < TITLE > The Tragedy of Hamlet, Prince of Denmark < /TITLE >

  <!-- XML removed for clarity -->

```

```

< PERSONAE >
  < TITLE > Dramatis Personae < /TITLE >

  < PERSONA > CLAUDIUS, king of Denmark. < /PERSONA >
  < PERSONA > HAMLET, son to the late king,
  and nephew to the present king. < /PERSONA >
  < PERSONA > POLONIUS, lord chamberlain. < /PERSONA >
  < PERSONA > HORATIO, friend to Hamlet. < /PERSONA >
  < PERSONA > LAERTES, son to Polonius. < /PERSONA >
  < PERSONA > LUCIANUS, nephew to the king. < /PERSONA >

  <!-- XML removed for clarity -->

< /PERSONAE >
< /PLAY >

```

使用这个 XML 文档，复习下面这段使用该 XML 的 C# 代码：

```

XDocument xdoc = XDocument.Load(@"C:\hamlet.xml");
Console.WriteLine(
    xdoc.Element("PLAY").Element("PERSONAE").Element("PERSONA").Value);

```

这段代码首先访问<PLAY>元素，再访问<PERSONA>元素，最后使用<PERSONA>元素。但是，其结果如下：

```
CLAUDIUS, king of Denmark.
```

原因是尽管有一个<PERSONA>元素集合，但我们只处理使用 Element().Value 调用遇到的第一个<PERSONA>元素。

## 29.4.2 写入 XML 文档

除了读取 XML 文档之外，还可以写入该文档。例如，如果要改变 Hamlet 戏剧文件的第一个演员名，就可以使用下面的代码：

```

using System;
using System.Linq;
using System.Xml.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main()
        {
            XDocument xdoc = XDocument.Load(@"C:\hamlet.xml");

            xdoc.Element("PLAY").Element("PERSONAE").
                Element("PERSONA").SetValue("Bill Evjen, king of Denmark");

            Console.WriteLine(xdoc.Element("PLAY").
                Element("PERSONAE").Element("PERSONA").Value);

            Console.ReadLine();
        }
    }
}

```

在这个例子中,使用 `Element()`对象的 `SetValue()`方法把<PERSONA>元素的第一个实例重写为 Bill Evjen, king of Denmark。调用了 `SetValue()`方法,将值应用于 XML 文档后,就使用与前面的相同的方法检索该值。运行这段代码,会发现第一个<PERSONA>元素的值改变了。

修改文档的另一种方式(在这个例子中是给文档添加项)是把需要的元素创建为 `XElement` 对象,再把它们添加到文档中:

```
using System;
using System.Linq;
using System.Xml.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main()
        {
            XDocument xdoc = XDocument.Load(@"C:\hamlet.xml");

            XElement xe = new XElement("PERSONA",
                "Bill Evjen, king of Denmark");

            xdoc.Element("PLAY").Element("PERSONAE").Add(xe);

            var query = from people in xdoc.Descendants("PERSONA")
                        select people.Value;

            Console.WriteLine("{0} Players Found", query.Count());
            Console.WriteLine();

            foreach (var item in query)
            {
                Console.WriteLine(item);
            }
            Console.ReadLine();
        }
    }
}
```

在这个例子中,创建了一个 `XElement` 文档 `xe`。`xe` 的构造会提供如下 XML 结果:

```
< PERSONA > Bill Evjen, king of Denmark < /PERSONA >
```

接着使用 `XDocument` 对象的 `Element().Add()`方法,添加所创建的元素:

```
xdoc.Element("PLAY").Element("PERSONAE").Add(xe);
```

查询所有的演员时,会得到 27 个演员,而不是 26 个,新加的一个演员在列表的底部。除了 `Add()`之外,还可以使用 `AddFirst()`方法,它会把元素添加到列表的开头,而不是默认的末尾。

## 29.5 使用 LINQ to SQL 和 LINQ to XML

使用 LINQ to SQL 或 LINQ to XML 时,只能处理指定的数据源。事实上,在使用 LINQ 时,可以把多个数据源混合在一起。例如,本节就使用 LINQ to SQL 查询 Northwind 数据库中的顾

客，再把提取出的结果转变为一个 XML 文档。

**提示：**

获得 Northwind 示例数据库文件的方法和使用 LINQ to SQL 的内容详见第 27 章。

### 29.5.1 建立 LINQ to SQL 组件

首先把 SQL Server Express Edition 数据库文件 Northwind 添加到项目中。之后，右击项目，给它添加一个新的 LINQ to SQL 类文件，命名为 Northwind.dbml。

这个操作会打开一个可使用的设计界面。在 Server Explorer 中，把表从数据库拖放到这个设计界面上。要拖放的表有 Customers 和 Orders。此时注意在这两个表之间建立了一个关系。IDE 中的视图应如图 29-7 所示。

有了 Northwind.dbml 后，就准备查询这个数据库结构，把结果输出为 XML 文件。

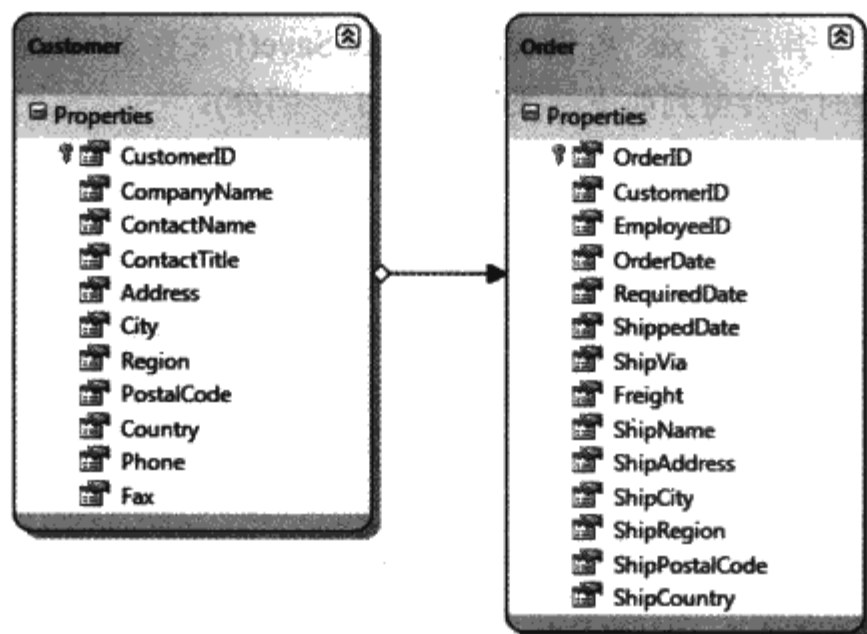


图 29-7

### 29.5.2 查询数据库，输出 XML

在控制台应用程序中，下一步是在 Program.cs 文件中添加如下代码：

```
using System;
using System.Linq;
using System.Xml.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main()
        {
            NorthwindDataContext dc = new NorthwindDataContext();

            XElement xe = new XElement("Customer",
                from c in dc.Customers
                select new XElement("Customer",
                    new XElement("CustomerId", c.CustomerID),
```

```

        new XElement("CompanyName", c.CompanyName),
        new XElement("Country", c.Country),
        new XElement("OrderNum", c.Orders.Count)));

    xe.Save(@"C:\myCustomers.xml");
    Console.WriteLine("File created");
    Console.ReadLine();
}
}
}

```

这个例子创建了 `NorthwindDataContext` 对象的一个新实例,它是用前面创建的 LINQ to SQL 类自动创建的。接着,不是执行正常的查询:

```
var query = [ query ]
```

而是在 `XElement` 对象 `xe` 中填充要执行的查询。在查询的 `select` 语句中,还创建 `Customers` 对象的迭代,它包含嵌套的元素 `<Customer>`、`<CustomerId>`、`<CompanyName>`、`<Country>` 和 `<OrderNum>`。查询完毕后, `xe` 实例就使用 `xe.Save()` 保存到磁盘上。进入磁盘查看 `myCustomers.xml` 文件时,会得到如下结果(仅显示了一部分):

```

< ?xml version="1.0" encoding="utf-8"? >
< Customer >
  < Customer >
    < CustomerId > ALFKI < /CustomerId >
    < CompanyName > Alfreds Futterkiste < /CompanyName >
    < Country > Germany < /Country >
    < OrderNum > 6 < /OrderNum >
  < /Customer >
  < Customer >
    < CustomerId > ANATR < /CustomerId >
    < CompanyName > Ana Trujillo Emparedados y helados < /CompanyName >
    < Country > Mexico < /Country >
    < OrderNum > 4 < /OrderNum >
  < /Customer >

  <!-- XML removed for clarity -->

  < Customer >
    < CustomerId > WILMK < /CustomerId >
    < CompanyName > Wilman Kala < /CompanyName >
    < Country > Finland < /Country >
    < OrderNum > 7 < /OrderNum >
  < /Customer >
  < Customer >
    < CustomerId > WOLZA < /CustomerId >
    < CompanyName > Wolski Zajazd < /CompanyName >
    < Country > Poland < /Country >
    < OrderNum > 7 < /OrderNum >
  < /Customer >
< /Customer >

```

因此,可以看出,使用 LINQ 很容易混合两个数据源。使用 LINQ to SQL 从数据库中提取出顾客信息,再使用 LINQ to XML 创建一个 XML 文件,并输出到磁盘上。



## 29.6 小结

本章介绍了如何使用 LINQ to XML 和读写 XML 文件及 XML 源(可以是静态或动态的)的一些选项。

使用 LINQ to XML 可以通过一系列强类型化的操作对 XML 文件及 XML 源执行 CRUD 操作。也可以联合使用 `XmlReader`、`XmlWriter` 和新的 LINQ to XML 功能编写代码。

本章还介绍了新的 LINQ to XML 帮助对象 `XDocument`、`XElement`、`XNamespace`、`XAttribute` 和 `XComment`。这些都是使 XML 操作比以前更方便的杰出新对象。

下一章介绍 Microsoft SQL Server 的编程。

# 第30章

## .NET 编程和 SQL Server

SQL Server 2005 是这个数据库产品存储 .NET 运行库的第一个版本。实际上，它是 Microsoft 的 SQL Server 产品在近 6 年中的第一个新版本，可以在 SQL Server 进程中运行 .NET 程序集。而且，SQL Server 2005 可以用 .NET 编程语言，如 C# 和 Visual Basic，创建存储过程、函数和数据类型。

本章介绍如下主要内容：

- 用 SQL Server 执行 .NET 运行库
- System.Data.SqlServer 命名空间中的类
- 创建用户定义的类型
- 创建用户定义的合计函数
- 存储过程
- 用户定义的函数
- 触发器
- XML 数据类型

注意：

本章需要 SQL Server 2005 或更高版本。

SQL Server 有许多新特性不能直接与 CLR 关联起来，例如许多对 T-SQL 的改进，但本章没有介绍它们。要了解这些特性的更多信息，可参阅 *SQL Server 2005 Express Edition Starter Kit* (Wiley 出版社，ISBN 0-7645-8923-7)。

本章的示例使用 ProCSharp 数据库和 AdventureWorks 数据库，ProCSharp 数据库可以与代码示例一起下载。AdventureWorks 数据库是 Microsoft 的一个示例数据库，可以在安装 SQL Server 时选择安装它。

### 30.1 .NET 运行库的主机

SQL Server 是 .NET 运行库的一个主机。在 CLR 2.0 以前的版本中，.NET 应用程序已经有多个主机，例如 Windows 窗体的主机和 ASP.NET 的主机。Internet Explorer 是另一个能运行 Windows 窗体控件的运行库主机。

SQL Server 允许在 SQL Server 进程上运行 .NET 程序集，在该进程中，可以用 CLR 代码创

建存储过程、函数、数据类型和触发器。

每个使用 CLR 代码的数据库都创建了它自己的应用程序域。这将确保一个数据库的 CLR 代码不影响其他数据库。

**注意：**

应用程序域的内容详见第 17 章。

.NET 1.0 设计了一个考虑周详的安全环境，其安全性已得到了证明。这个安全环境并不足以应付非常重要的数据库——.NET 需要进行一些扩展。SQL Server 作为 .NET 运行库的主机，定义了另一些许可级：safe、external 和 unsafe。

**注意：**

基于证书的安全性详见第 20 章。

- **Safe:** 在 safe 安全级上，只能使用计算的 CLR 类。程序集只能进行本地数据访问。这些类的功能类似于 T-SQL 存储过程。代码访问安全机制指定，只有具备 .NET 许可，才能执行 CLR 代码。
- **External:** 在 external 安全级上，可以使用客户端的 ADO.NET 访问网络、文件系统、注册表或其他数据库。
- **Unsafe:** 在 unsafe 安全级上，可以执行任何操作，因为这个安全级允许调用本机代码。有 unsafe 许可级的程序集只能由数据库管理员安装。

为了在 SQL Server 上运行定制的 .NET 代码，CLR 必须用 sp\_configure 存储过程激活：

```
sp_configure [clr enabled], 1
reconfigure
```

在 .NET 2.0 中，在 System.Security.Permissions 命名空间中引入了 HostProtectionAttribute 属性类，以更好地保护主机环境。利用这个属性，可以指定方法是否使用共享状态、进行同步，或者控制主机环境。因为这种操作在 SQL Server 代码中通常是不需要的(会影响 SQL Server 的性能)，所以应用了这些设置的程序集不允许加载到安全级是 safe 和 external 的 SQL Server 上。

为了在 SQL Server 上使用程序集，程序集可以使用 CREATE ASSEMBLY 命令安装。通过这个命令，可以在 SQL Server 中使用程序集的名称、程序集的路径和安全级别：

```
CREATE ASSEMBLY mylibrary FROM c:/ProCSharp/SqlServer/Demo.dll
WITH PERMISSION SET = SAFE
```

在 Visual Studio 2008 中，所生成程序集的许可级别可以用项目的 Database 属性定义，如图 30-1 所示。

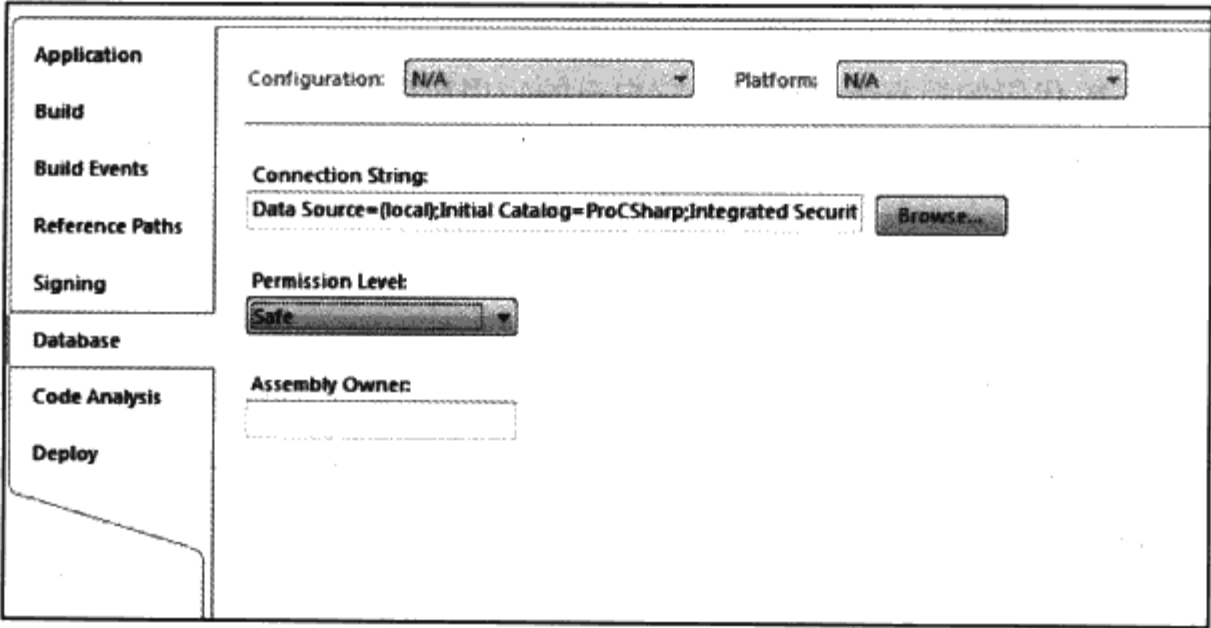


图 30-1

### 30.2 Microsoft.SqlServer.Server

第 26 章讨论了 `System.Data.SqlClient` 命名空间中的类。这里讨论另一个命名空间 `Microsoft.SqlServer.Server`。它包含专用于 .NET Framework 的类、接口和枚举。但是，`System.Data.SqlClient` 中的许多类也需要服务器端的代码。

表 30-1 列出了 `Microsoft.SqlServer.Server` 命名空间中的主要类及其功能。

表 30-1

类	说 明
SqlContext	与 HTTP 环境一样，SQL 环境也与客户机的请求相关。使用 <code>SqlContext</code> 类的静态成员，可以访问 <code>SqlPipe</code> 、 <code>SqlTriggerContext</code> 和 <code>WindowsIdentity</code>
SqlPipe	使用 <code>SqlPipe</code> 类，可以把结果或信息发送给客户机。这个类提供了 <code>ExecuteAndSend()</code> 、 <code>Send()</code> 和 <code>SendResultsRow()</code> 方法。 <code>Send()</code> 方法有不同的重载版本，分别发送 <code>SqlDataReader</code> 、 <code>SqlDataRecord</code> 和 <code>string</code>
SqlDataRecord	<code>SqlDataRecord</code> 表示一行数据。这个类和 <code>SqlPipe</code> 一起使用，收发来自客户机的信息
SqlTriggerContext	<code>SqlTriggerContext</code> 类在触发器中使用。这个类提供了已引发的触发器信息

这个命名空间还包含几个属性类 `SqlProcedureAttribute`、`SqlFunctionAttribute`、`SqlUserDefinedAttribute` 和 `SqlTriggerAttribute`。这些类用于在 SQL Server 中部署存储过程、函数、用户定义的类型和触发器。在 Visual Studio 中部署时，需要应用这些属性。在使用 SQL 语句部署数据库对象时，不需要这些属性，但这些属性的一些特性会影响数据库对象的特性。

本章后面编写存储过程和用户定义的函数时，会介绍这些类。但下面先看看如何使用 C# 创建用户定义的类型。

### 30.3 用户定义的类型

用户定义的类型(UDT)的用法与一般的 SQL Server 数据类型一样,都是定义表中一列的类型。在 SQL Server 的旧版本中,就可以定义 UDT。当然,这些 UDT 只能以 SQL 类型为基础,例如下面代码中的 ZIP 类型。`sp_addtype` 存储过程允许创建用户定义的类型。这里,用户定义的类型 ZIP 以 CHAR 数据类型为基础,其长度为 5。`NOT NULL` 指定 ZIP 数据类型不允许使用 NULL。把 ZIP 用作一个数据类型,就不再需要记住,它有 5 个字符,且不能为空:

```
EXEC sp_addtype ZIP 'CHAR(5)', 'NOT NULL'
```

在 SQL Server 2005 和以后的版本中,UDT 可以用 CLR 类定义。但这个特性不能在数据库中添加面向对象特性,例如,把 `Person` 类创建为包含 `Person` 数据类型。SQL Server 是一个关系数据库,这对 UDT 也是正确的。不能创建 UDT 的类层次结构,也不能用 `SELECT` 语句引用 UDT 类型的字段或属性。如果必须对 `Person` 类的属性(例如 `Firstname` 或 `Lastname`)进行访问或排序,最好在 `Person` 表中为姓或名分别定义列,或使用 XML 数据类型。

UDT 是非常简单的数据类型。在推出 .NET 之前,还可以创建定制的数据类型,例如 ZIP 数据类型。不能利用 UDT 创建类层次结构,也不能把复杂的数据类型放在数据库中。UDT 的一个要求是,它必须能转换为字符串,因为字符串表示用于显示值。

可以定义数据在 SQL Server 中的存储方式:或者使用自动执行的机制,以内部格式存储数据;或者把数据转换为字节流,定义数据的存储方式。

#### 30.3.1 创建 UDT

下面探讨如何创建用户定义的类型。我们创建一个 `SqlCoordinate` 类型,表示用于显示经度和纬度的世界坐标系,以便于定义地点、城市等的位置。使用 Visual Studio 创建 CLR 对象,可以使用 Visual Studio 2008 SQL Server Project (在 Visual C# | Database 类别中),选择 Solution Explorer,使用 User-Defined Type 模板添加一个 UDT,指定类型的名称 `SqlCoordinate`。在模板中,已经定义好了定制类型的基本功能:

```
using System;
using System.Data;
using System.Data.Sql;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;

[Serializable]
[Microsoft.SqlServer.Server.SqlUserDefinedType(Format.Native)]
public struct SqlSqlCoordinate : INullable
{
    public override string ToString()
    {
        // Replace the following code with your code
        return "";
    }

    public bool IsNull
    {
```



```

        get
        {
            // Put your code here
            return m_Null;
        }
    }

    public static SqlSqlCoordinate Null
    {
        get
        {
            SqlSqlCoordinate h = new SqlSqlCoordinate();
            h.m_Null = true;
            return h;
        }
    }

    public static SqlSqlCoordinate Parse(SqlString s)
    {
        if (s.IsNull)
            return Null;
        SqlSqlCoordinate u = new SqlSqlCoordinate();
        // Put your code here
        return u;
    }

    // This is a place-holder method
    public string Method1()
    {
        //Insert method code here
        return "Hello";
    }

    // This is a place-holder static method
    public static SqlString Method2()
    {
        // Insert method code here
        return new SqlString("Hello");
    }

    // This is a placeholder field member
    public int var1;
    // Private member
    private bool m_Null;
}

```

因为这个类型也可以直接在客户机代码中使用，所以最好添加一个命名空间，这不是自动实现的。

SqlCoordinate 实现了 INullable 接口。INullable 接口是 UDT 必须实现的，因为数据库类型也可以为空。[SqlUserDefinedType] 属性由 Visual Studio 用于对 UDT 自动部署。参数 Format.Native 指定要使用的串行化格式。可以使用两种串行化格式：Format.Native 和 Format.UserDefined。Format.Native 是简单的串行化格式，引擎会对实例进行串行化和并行化。这种串行化只对字面量数据类型有效(字面量数据类型在托管代码和本地代码中有相同的内存表示)。在 Coordinate 类中，要串行化的数据类型是 int 和 bool，它们都是字面量数据类型。string 不是字面量数据类型。使用 Format.UserDefined 需要实现 IBinarySerialize 接口。该接口提供了用户定义的类型定制执行方式。在这个接口中，必须执行方法 Read() 和 write()，才能把数据

串行化到 BinaryReader 和 BinaryWriter 中。

注意：

字面量数据类型在托管和非托管的内存中有相同的内存表示。字面量数据类型不需要类型转换，这些数据类型有 byte、sbyte、short、ushort、int、uint、long、ulong，以及只包含这些数据类型的组合，如数组和结构。

```
namespace Wrox.ProCSharp.SqlServer
{
    [Serializable]
    [SqlUserDefinedType(Format.Native)]
    public struct SqlCoordinate : INullable
    {
        private int longitude;
        private int latitude;
        private bool isNull;
    }
}
```

[SqlUserDefinedType]属性允许设置几个特性，如表 30-2 所示。

表 30-2

SqlUserDefinedType-Attribute 属性	说 明
Format	特性 Format 指定数据类型在 SQL Server 中的存储方式，目前支持的格式有 Format.Native 和 Format.UserDefined
IsByteOrdered	如果 IsByteOrdered 特性设置为 true，就可以创建数据类型的索引，由 SQL 语句 GROUP BY 和 ORDER BY 使用。磁盘的表示可以用于二元比较。每个实例都只能有一个串行化表示，所以二元比较是可以成功的。其默认值是 false
IsFixedLength	如果所有实例的磁盘表示都有相同的大小，IsFixedLength 就可以设置为 true
MaxByteSize	存储数据所需的最大字节数用 MaxByteSize 设置。这个属性只能在用户定义的串行化中指定
Name	利用 Name 属性，可以设置类型的另一个名称。默认情况下使用类名
ValidationMethodName	利用 ValidationMethodName 属性，可以把方法名定义为在进行并行化时验证实例

为了表示坐标的方向，定义了 Orientation 枚举：

```
public enum Orientation
{
    NorthEast,
    NorthWest,
    SouthEast,
    SouthWest
}
```

这个枚举只能在 SqlCoordinate 结构的方法中使用，不能用作成员字段，因为枚举不是字面量数据类型。以后的版本可能支持在 SQL Server 中使用带内置格式的枚举。

SqlCoordinate 结构指定一些构造函数来初始化 longitude、latitude 和 isNull 变量。如果没有

给 longitude 和 latitude 赋值, 就把 isNull 变量设置为 true, 这是默认构造函数的执行情况。UDT 需要默认构造函数。

在世界坐标系统中, 经度和纬度都定义为度、分、秒。奥地利的维也纳位于经度 48° 14', 纬度 16° 20'。符号°、'、"分别表示度、分、秒。

在变量 longitude 和 latitude 中, 经度和纬度值用秒来存储。有 7 个整型参数的构造函数把度、分、秒转换为秒, 如果坐标基于南和西, 就把经度和纬度设置为负值:

```
public SqlCoordinate(int longitude, int latitude)
{
    isNull = false;
    this.longitude = longitude;
    this.latitude = latitude;
}

public SqlCoordinate(int longitudeDegrees, int longitudeMinutes,
    int longitudeSeconds, int latitudeDegrees, int latitudeMinutes,
    int latitudeSeconds, Orientation orientation)
{
    isNull = false;
    this.longitude = longitudeSeconds + 60 * longitudeMinutes + 3600 *
        longitudeDegrees;
    this.latitude = latitudeSeconds + 60 * latitudeMinutes + 3600 *
        latitudeDegrees;
    switch (orientation)
    {
        case Orientation.SouthWest:
            longitude = -longitude;
            latitude = -latitude;
            break;
        case Orientation.SouthEast:
            longitude = -longitude;
            break;
        case Orientation.NorthWest:
            latitude = -latitude;
            break;
    }
}
```

INullable 接口定义了 IsNull 属性, 必须实现该接口, 才能支持可空性。静态属性 Null 用于创建一个表示空值的对象。在 get 访问器中, 创建了一个 SqlCoordinate 对象, isNull 字段设置为 true:

```
public bool IsNull
{
    get
    {
        return isNull;
    }
}

public static SqlCoordinate Null
{
    get
    {
        SqlCoordinate c = new SqlCoordinate();
        c.isNull = true;
    }
}
```



```

        return c;
    }
}

```

UDT 必须转换为字符串, 也应能从字符串中转换回来。要转换为字符串, 必须重写 `Object` 类的 `ToString()` 方法。这里, 变量 `longitude` 和 `latitude` 转换为一个字符串表示, 以显示度、分和秒:

```

public override string ToString()
{
    if (this.IsNull)
        return null;
    char northSouth = longitude > 0 ? 'N' : 'S';
    char eastWest = latitude > 0 ? 'E' : 'W';

    int longitudeDegrees = Math.Abs(longitude) / 3600;
    int remainingSeconds = Math.Abs(longitude) % 3600;
    int longitudeMinutes = remainingSeconds / 60;
    int longitudeSeconds = remainingSeconds % 60;

    int latitudeDegrees = Math.Abs(latitude) / 3600;
    remainingSeconds = Math.Abs(latitude) % 3600;
    int latitudeMinutes = remainingSeconds / 60;
    int latitudeSeconds = remainingSeconds % 60;

    return String.Format("{0}° {1}' {2}\" {3}, {4}° {5}' {6}\" {7}",
        longitudeDegrees, longitudeMinutes, longitudeSeconds, northSouth,
        latitudeDegrees, latitudeMinutes, latitudeSeconds, eastWest);
}

```

用户输入的字符串在静态方法 `Parse()` 的 `SqlString` 参数中表示。首先, `Parse()` 方法检查该字符串是否表示空值, 如果是, 就调用 `Null` 属性, 返回一个空的 `SqlCoordinate` 对象。如果 `SqlString s` 不表示空值, 就转换字符串的文本, 把经度和纬度值传送给 `SqlCoordinate` 构造函数:

```

public static SqlCoordinate Parse(SqlString s)
{
    if (s.IsNull)
        return SqlCoordinate.Null;

    try
    {
        string[] coordinates = s.Value.Split(',');
        char[] separators = { '°', '\'', '\"' };
        string[] longitudeVals = coordinates[0].Split(separators);
        string[] latitudeVals = coordinates[1].Split(separators);

        Orientation orientation;
        if (longitudeVals[3] == "N" && latitudeVals[3] == "E")
            orientation = Orientation.NorthEast;
        else if (longitudeVals[3] == "S" && latitudeVals[3] == "W")
            orientation = Orientation.SouthWest;
        else if (longitudeVals[3] == "S" && latitudeVals[3] == "E")
            orientation = Orientation.SouthEast;
        else
            orientation = Orientation.NorthWest;

        return new SqlCoordinate(
            int.Parse(longitudeVals[0]), int.Parse(longitudeVals[1]),
            int.Parse(longitudeVals[2]),
            int.Parse(latitudeVals[0]), int.Parse(latitudeVals[1]),
            int.Parse(latitudeVals[2]), orientation);
    }
}

```

```
    }
    catch (Exception ex)
    {
        throw new ArgumentException(
            "Argument has a wrong syntax. " +
            "This syntax is required: 37° 47'0\"N,122° 26'0\"W",
            ex.Message);
    }
}
```

30.3.2 使用 UDT

在建立好程序集后，就可以用 SQL Server 部署它。在 SQL Server 中，UDT 可以在 Visual Studio 2008 中用 Build | Deploy Project 菜单配置，或用下面的 SQL 命令配置：

```
CREATE ASSEMBLY SampleTypes FROM
'c:\ProCSharp\SQL2005\PropCSharp.SqlTypes.dll'
CREATE TYPE Coordinate EXTERNAL NAME
[ProCSharp.SqlTypes].[ProCSharp.SqlTypes.SqlCoordinate]
```

使用 EXTERNAL NAME 时，必须设置程序集名和类名，包括命名空间。  
现在可以创建一个表 Cities，其中包含数据类型 SqlCoordinate，如图 30-2 和 30-3 所示。

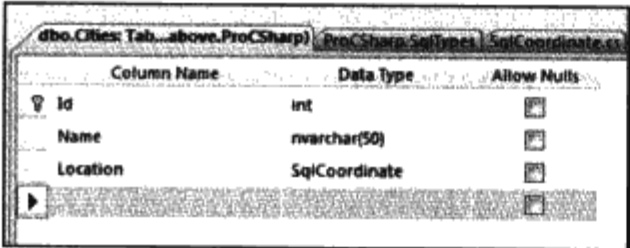
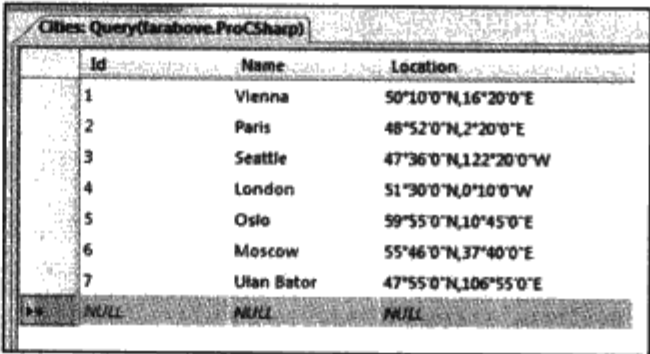


图 30-2



Id	Name	Location
1	Vienna	50°10'0"N,16°20'0"E
2	Paris	48°52'0"N,2°20'0"E
3	Seattle	47°36'0"N,122°20'0"W
4	London	51°30'0"N,0°10'0"W
5	Oslo	59°55'0"N,10°45'0"E
6	Moscow	55°46'0"N,37°40'0"E
7	Ulan Bator	47°55'0"N,106°55'0"E
8	NULL	NULL

图 30-3

30.3.3 在客户端代码中使用 UDT

必须引用 UDT 的程序集，才能在客户端代码中使用 UDT。它的用法与客户端上的其他类型一样。

**提示：**  
包含 UDT 的程序集用于客户机和 SQL 服务器，所以最好把 UDT 放在一个独立的程序集中，而不是放在其他 SQL Server 扩展中，例如存储过程和函数。

在示例代码中，SqlCommand 对象的 SELECT 语句引用 Cities 表的列，该表包含类型为 SqlCoordinate 的 Location 列。调用 ToString()方法，会执行 SqlCoordinate 类的 ToString()方法，以字符串格式显示坐标值：

```
// UDTCClient
using System;
using System.Data;
using System.Data.SqlClient;
using Wrox.ProCSharp.SqlServer;
```



```

class Program
{
    static void Main()
    {
        string connectionString =
            @"server=(local);database=ProCSharp;trusted_connection=true";
        SqlConnection connection = new SqlConnection(connectionString);
        SqlCommand command = connection.CreateCommand();
        command.CommandText = "SELECT Id, Name, Location FROM Cities";
        connection.Open();

        SqlDataReader reader =
            command.ExecuteReader(CommandBehavior.CloseConnection);
        while (reader.Read())
        {
            Console.WriteLine("{0,-10}: {1}", reader[1].ToString(),
                reader[2].ToString());
        }
        reader.Close();
    }
}

```

当然，还可以把返回的对象从 `SqlDataReader` 转换为 `SqlCoordinate` 类型，以使用 `SqlCoordinate` 类型的其他方法。

```
SqlCoordinate coordinate = (SqlCoordinate)reader[2];
```

运行应用程序，结果如下：

Vienna	50°10'0"N,16°20'0"E
Paris	48°52'0"N,2°20'0"E
Seattle	47°36'0"N,122°20'0"W
London	51°30'0"N,0°10'0"W
Oslo	59°55'0"N,10°45'0"E
Moscow	55°46'0"N,37°40'0"E
Ulan Bator	47°55'0"N,106°55'0"E

提示：

有了 UDT 的所有强大功能之后，还必须注意一个重要的限制。在部署 UDT 的一个新版本之前，必须删除已有的版本。如果删除所有使用 UDT 类型的列，就必须这么做。不要给频繁修改的类型使用 UDT。

## 30.4 用户定义的合计函数

合计函数是根据多行返回一个值的函数。内置的合计函数有 `COUNT`、`AVG` 和 `SUM`。`COUNT` 返回所有选中记录的记录个数，`AVG` 返回选中行的一列的平均值，`SUM` 返回一列的所有值的总和。所有的内置合计函数都只能用于内置值类型。

内置合计函数 `AVG` 的一种简单用法如下，它把 AdventureWorks 示例数据库中的 `ListPrice` 列传送给 `SELECT` 语句中的 `AVG` 合计函数，返回所有产品的平均单价：

```

SELECT AVG(ListPrice) AS 'average list price'
FROM Production.Product

```

SELECT 语句返回所有产品的平均单价:

```
average list price
438,6662
```

SELECT 语句返回 ListPrice 列的平均值。合计函数还可以用于组。在下面的例子中, AVG 合计函数与 GROUP BY 子句一起使用, 返回每个产品系列的平均单价:

```
SELECT ProductLine, AVG(ListPrice) AS 'average list price'
FROM Production.Product
GROUP BY ProductLine
```

平均单价按照产品系列进行组合:

ProductLine	average list price
NULL	16,8429
M	827,0639
R	965,3488
S	50,3988
T	840,7621

对于定制的值类型, 如果要对选中的一些行执行某个计算操作, 就可以创建用户定义的合计函数。

30.4.1 创建用户定义的合计函数

要用 CLR 代码编写用户定义的合计函数, 必须实现一个简单的类, 它的方法有 Init()、Accumulate()、Merge()和 Terminate()。这些方法的功能如表 30-3 所示。

表 30-3

UDT 方法	说 明
Init()	为要处理的每组行调用 Init()方法。在这个方法中, 可以为每组行执行初始化
Accumulate()	为所有组中的每个值调用方法 Accumulate()。这个方法的参数必须是正确的累加类型, 还可以是用户定义的类型
Merge()	合计的结果必须与另一个合计结果合并起来时, 调用方法 Merge()
Terminate()	在处理完每个组的最后一行后, 调用方法 Terminate()。这里, 合计的结果必须用正确的数据类型返回

下面的代码示例说明了如何执行一个简单的用户定义合计函数, 计算每个组中所有行的总和。为了用 Visual Studio 进行部署, 把[SqlUserDefinedAggregate]属性应用于 SampleSum 类。与用户定义的类型一样, 用户定义的合计函数为存储合计结果使用的格式也必须用 Format 枚举中的值定义。Format.Native 用于对字面量数据类型进行自动串行化。

在示例代码中, 变量 sum 用于累加组中的所有值。在 Init()方法中, 给每个要累加的新组初始化变量 sum。为每个值调用的 Accumulate()方法把参数的值加到 sum 变量中。在 Merge()方法中, 将一个合计的组添加到当前组中。最后, Terminate()方法返回组的结果。

```
[Serializable]
```

```

[SqlUserDefinedAggregate(Format.Native)]
public struct SampleSum
{
    private int sum;

    public void Init()
    {
        sum = 0;
    }

    public void Accumulate(SqlInt32 Value)
    {
        sum += Value.Value;
    }

    public void Merge(SampleSum Group)
    {
        sum += Group.sum;
    }

    public SqlInt32 Terminate()
    {
        return new SqlInt32(sum);
    }
}

```

**提示:**

可以使用 Visual Studio 中的 Aggregate 模板, 为建立用户定义的合计函数创建核心代码。Visual Studio 中的模板创建了一个结构, 它使用 SqlString 类型作为参数, 用 Accumulate 和 Terminate 方法返回类型。可以把这个类型改为表示合计要求的类型。在本例中, 使用了 SqlInt32 类型。

### 30.4.2 使用用户定义的合计函数

用户定义的合计函数可以用 Visual Studio 或 CREATE AGGREGATE 语句部署。在下面的语句中, CREATE AGGREGATE 是合计函数名, 参数是(@value int), 返回类型 EXTERNAL NAME 需要程序集名和包含命名空间的.NET 类型。

```

CREATE AGGREGATE [SampleSum] (@value int) RETURNS [int] EXTERNAL NAME
[Demo].[SampleSum]

```

在安装用户定义的合计函数后, 就可以在下面的 SELECT 语句中使用, 在这个语句中, 把 Products 和 PurchaseOrderDetails 表连接起来, 返回已订购的产品数。对于用户定义的合计函数, PurchaseOrderDetails 表的 OrderQty 列定义为一个参数:

```

SELECT Purchasing.PurchaseOrderDetail.ProductID AS Id,
       Production.Product.Name AS Product,
       dbo.SampleSum(Purchasing.PurchaseOrderDetail.OrderQty) AS Sum
FROM Production.Product INNER JOIN
     Purchasing.PurchaseOrderDetail ON
     Purchasing.PurchaseOrderDetail.ProductID = Production.Product.ProductID
GROUP BY Purchasing.PurchaseOrderDetail.ProductID, Production.Product.Name
ORDER BY Id

```

返回的结果使用合计函数 SampleSum 显示已订购的产品数:

Id	Product	Sum
1	Adjustable Race	154
2	Bearing Ball	150
4	Headset Ball Bearings	153
317	LL Crankarm	44000
318	ML Crankarm	44000
319	HL Crankarm	71500
320	Chainring Bolts	375
321	Chainring Nut	375
322	Chainring	7440

## 30.5 存储过程

SQL Server 可以用 C# 创建存储过程, 存储过程是一个子例程, 物理存储在数据库中。但它们不能替代 T-SQL。在过程主要是数据驱动时, T-SQL 还是有优势的。

下面是 T-SQL 存储过程 GetCustomerOrders, 它返回 AdventureWorks 数据库中的客户订单信息。这个存储过程返回用 CustomerID 参数指定的顾客的订单:

```
CREATE PROCEDURE GetCustomerOrders
(
    @CustomerID int
)
AS
SELECT SalesOrderID, OrderDate, DueDate, ShipDate FROM Sales.SalesOrderHeader
WHERE (CustomerID = @CustomerID)
ORDER BY SalesOrderID
```

### 30.5.1 创建存储过程

在下面的代码清单中, 用 C# 实现相同的存储过程比较复杂。[SqlProcedure] 属性用于把存储过程标记为部署。在这个执行代码中, 创建了一个 SqlCommand 对象。在 SqlConnection 对象的构造函数中, 传送了字符串 Context Connection=true, 以使用调用该存储过程的客户端打开的连接。与第 26 章的代码类似, 设置了 SQL SELECT 语句, 添加了一个参数。ExecuteReader() 方法返回一个 SqlDataReader 对象。通过调用 SqlPipe 的 Send() 方法, 把这个读取器对象返回给客户端:

```
using System.Data;
using Microsoft.SqlServer.Server;
using System.Data.SqlClient;

public partial class StoredProcedures
{
    [SqlProcedure]
    public static void GetCustomerOrdersCLR(int customerId)
    {
        SqlConnection connection = new SqlConnection("Context Connection=true");
        connection.Open();
        SqlCommand command = new SqlCommand();
```



```

        command.Connection = connection;
        command.CommandText = "SELECT SalesOrderID, OrderDate, DueDate, ShipDate " +
            "FROM Sales.SalesOrderHeader " +
            "WHERE (CustomerID = @CustomerID)" +
            "ORDER BY SalesOrderID";

        command.Parameters.Add("@CustomerID", SqlDbType.Int);
        command.Parameters["@CustomerID"].Value = customerId;

        SqlDataReader reader = command.ExecuteReader();
        SqlPipe pipe = SqlContext.Pipe;
        pipe.Send(reader);
        connection.Close();
    }
};

```

CLR 存储过程在 SQL Server 中使用 Visual Studio 或 CREATE PROCEDURE 语句部署。在这个 SQL 语句中，定义了存储过程的参数，以及程序集、类和方法的名称：

```

CREATE PROCEDURE GetCustomersOrders
(
    @CustomerID nchar(5)
)
AS EXTERNAL NAME Demo.StoredProcedures.GetCustomersOrdersCLR

```

### 30.5.2 使用存储过程

CLR 存储过程可以像一般的 T-SQL 存储过程那样，也是使用 System.Data.SqlClient 命名空间中的类调用。首先，创建一个 SqlConnection 对象，CreateCommand() 方法返回一个 SqlCommand 对象。通过这个命令对象，把存储过程的名称 GetCustomersOrdersCLR 指定给 CommandText 属性。与所有的存储过程一样，CommandType 属性必须设置为 CommandType.StoredProcedure。ExecuteReader() 方法返回一个 SqlDataReader 对象，用于逐个记录地读取：

```

using System;
using System.Data;
using System.Data.SqlClient;
//...

string connectionString =
    @"server=(local);database=AdventureWorks;trusted connection=true";
SqlConnection connection = new SqlConnection(connectionString);
SqlCommand command = connection.CreateCommand();
command.CommandText = "GetCustomerOrdersCLR";
command.CommandType = CommandType.StoredProcedure;
SqlParameter param = new SqlParameter("@customerId", 3);
command.Parameters.Add(param);
connection.Open();
SqlDataReader reader =
    command.ExecuteReader(CommandBehavior.CloseConnection);
while (reader.Read())
{
    Console.WriteLine("{0} {1:d}", reader["SalesOrderID"],
        reader["OrderDate"]);
}
reader.Close();

```



注意:

System.Data.SqlClient 命名空间中的类详见第 26 章。

调用用 T-SQL 和 C#编写的存储过程没有任何区别。调用存储过程的代码完全相同;在调用程序的代码中,不知道存储过程是用 T-SQL 还是 CLR 执行的。上述代码的结果是 ID 为 3 的客户的订单日期:

```
44124 9/1/2001
44791 12/1/2001
45568 3/1/2002
46377 6/1/2002
47439 9/1/2002
48378 12/1/2002
```

如前所述,数据驱动的存储过程用 T-SQL 编写比较好,代码比较短。用 CLR 编写存储过程的优点是,适合于需要一些特殊的数据处理,例如使用.NET 加密类处理。

## 30.6 用户定义的函数

用户定义的函数有点类似于存储过程。其主要区别是,用户定义的函数可以在 SQL 语句中调用。

### 30.6.1 创建用户定义的函数

CLR 用户定义的函数可以用属性[SqlFunction]来指定。示例函数 CalcHash()把传送进来的字符串转换为散列的字符串。用于散列字符串的 MD5 算法通过 System.Security.Cryptography 命名空间中的 MD5CryptoServiceProvider 类实现。ComputeHash()方法从输入的字节数组中计算散列。返回一个计算出来的散列字节数组。该数组会使用 StringBuilder 类转换回 string。

```
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;
using System.Text;
using System.Security.Cryptography;

public partial class UserDefinedFunctions
{
    [SqlFunction]
    public static SqlString CalcHash(SqlString value)
    {
        byte[] source;
        byte[] hash;

        source = ASCIIEncoding.ASCII.GetBytes(value.ToString());
        hash = new MD5CryptoServiceProvider().ComputeHash(source);

        StringBuilder output = new StringBuilder(hash.Length);

        for (int i = 0; i < hash.Length - 1; i++)
        {
            output.Append(hash[i].ToString("X2"));
        }
    }
}
```

```

        return new SqlString(output.ToString());
    }
}

```

### 30.6.2 使用用户定义的函数

用户定义的函数可以在 SQL Server 中部署, 与其他 .NET 扩展程序一样, 也是使用 Visual Studio 2008 或 CREATE FUNCTION 语句:

```

CREATE FUNCTION CalcHash
(
    @value nvarchar
)
RETURNS nvarchar
AS EXTERNAL NAME Demo.UserDefinedFunctions.CalcHash

```

CalcHash()函数的用法如下面的 SELECT 语句所示, 其中信用卡号从 AdventureWorks 数据库的 CreditCard 表中访问, 只从信用卡号中返回散列代码:

```

SELECT Sales.CreditCard.CardType AS [Card Type],
       dbo.CalcHash(Sales.CreditCard.CardNumber) AS [Hashed Card]
FROM Sales.CreditCard INNER JOIN Sales.ContactCreditCard ON
     Sales.CreditCard.CreditCardID = Sales.ContactCreditCard.CreditCardID
WHERE Sales.ContactCreditCard.ContactID = 11

```

返回的结果显示 ID 为 11 的联系人的散列信用卡号:

Card Type	Hashed Card
ColonialVoice	7482F7B4E613F71144A9B336A3B9F6

## 30.7 触发器

触发器是一种特殊的存储过程, 在修改表(例如插入、更新或删除一行)时调用。触发器与表和激活它们的动作(例如行的插入/更新/删除)关联在一起。

有了触发器, 行的修改就可以通过相关的表来分层处理, 或执行更复杂的数据完整性操作。

在触发器中, 可以访问行的当前数据和原始数据, 所以可以把表重置回以前的状态。触发器会自动与启动触发器的命令所在的事务处理关联起来, 所以会得到正确的事务处理结果。

下面的触发器 uCreditCard 可以用于 AdventureWorks 数据库。这个触发器在更新 CreditCard 表中的一行时启动。使用这个触发器, 将 CreditCard 表中的 ModifiedDate 列更新为当前日期。要访问已修改的数据, 可以使用插入的临时表。

```

CREATE TRIGGER [Sales].[uCreditCard] ON [Sales].[CreditCard]
AFTER UPDATE NOT FOR REPLICATION AS
BEGIN
    SET NOCOUNT ON;

    UPDATE [Sales].[CreditCard]
    SET [Sales].[CreditCard].[ModifiedDate] = GETDATE()
    FROM inserted
    WHERE inserted.[CreditCardID] = [Sales].[CreditCard].[CreditCardID];
END;

```

## 30.7.1 创建触发器

本节的例子演示了在 Users 表中插入新记录时触发器实现了数据完整性。要使用 CLR 创建触发器，必须定义一个简单的类，它包含应用了 [SqlTrigger] 属性的静态方法。[SqlTrigger] 属性指定了与触发器关联的表和启动触发器的事件。在下面的例子中，关联的表是 Target 属性指定的 Users，Event 属性定义了触发器启动的时间。这里事件字符串设置为 FOR INSERT，表示触发器在把一个新行插入 Users 表中时启动。

SqlContext.TriggerContext 属性在 SqlTriggerContext 类型的对象中返回触发器环境，SqlTriggerContext 类提供了 3 个属性：ColumnsUpdated 返回一个布尔数组，标记每个已修改的列，EventData 包含 XML 格式的更新数据和原始数据，TriggerAction 返回 TriggerAction 类型的枚举，标记触发器启动的原因。在这个例子中，要确定触发器环境的 TriggerAction 是否设置为 TriggerAction.Insert，之后再继续。

触发器可以访问临时表，例如，这里访问的是 INSERTED 表。通过 SQL 语句 INSERT、UPDATE 和 DELETE，可以创建临时表。INSERT 语句创建 INSERTED 表，DELETE 语句创建 DELETED 表。通过 UPDATE 语句，可以使用 INSERTED 和 DELETED 表。临时表的列与触发器关联的表相同。SQL 语句 SELECT Username, Email FROM INSERTED 用于访问用户名和电子邮件，检查电子邮件地址的语法是否正确。SqlCommand.ExecuteNonQuery() 返回一个在 SqlDataRecord 中表示的数据行。用户名和电子邮件从数据记录中读取。使用正则表达式类 RegEx 和 IsMatch() 方法，检查电子邮件地址是否遵循有效的电子邮件句法。如果不遵循，就抛出一个异常，不插入记录，因为事务处理会回退：

```
using System;
using Microsoft.SqlServer.Server;
using System.Data.SqlClient;
using System.Text.RegularExpressions;

public partial class Triggers
{
    [SqlTrigger(Name="InsertContact", Target="Person.Contact", Event="FOR INSERT")]
    public static void InsertContact()
    {
        SqlTriggerContext triggerContext = SqlContext.TriggerContext;

        if (triggerContext.TriggerAction == TriggerAction.Insert)
        {
            SqlConnection connection = new SqlConnection("Context Connection=true");
            SqlCommand command = new SqlCommand();
            command.Connection = connection;
            command.CommandText = "SELECT EmailAddress FROM INSERTED";
            connection.Open();
            string email = (string)command.ExecuteScalar();
            connection.Close();

            if (!Regex.IsMatch(email,
                @"([\w-]+\.)?[\w-]+@([\w-]+\.)?[\w-]+\.[\w-]+$"))
            {
                throw new FormatException("Invalid email");
            }
        }
    }
}
```

### 30.7.2 使用触发器

使用 Visual Studio 2008 的部署功能，可以把触发器部署到数据库上。可以使用 CREATE TRIGGER 命令手工创建触发器：

```
CREATE TRIGGER InsertUser ON Users
FOR INSERT
AS EXTERNAL NAME Demo.UserRegistrationTriggers.InsertUser
```

用不正确的电子邮件把行插入 Users 表，会抛出一个异常，且插入操作不会执行。

## 30.8 XML 数据类型

SQL Server 一个主要的新特性是 XML 数据类型。在 SQL Server 的旧版本中，XML 数据存储在字符串或 blob 中。现在 XML 是一个受到支持的数据类型，允许把 SQL 查询和 XQuery 表达式合并起来，在 XML 数据中搜索。XML 数据类型可以用作变量、参数、列或 UDF 的返回值。

在 Office 2007 中，可以把 Word 和 Excel 文档存储为 XML。Word 和 Excel 也允许使用定制的 XML 模式，只用 XML 存储内容，不存储表达方式。Office 应用程序的输出可以直接存储在 SQL Server 中，还可以在这些数据中搜索。当然，定制的 XML 数据也可以在 SQL Server 中存储。

注意：

不要给关系数据使用 XML 类型。如果搜索一些元素，且为这些数据明确定义了模式，则以关系方式存储这些元素，就可以更快找到数据。如果数据是带有层次结构的，一些元素是可选的，且随时间而变化，存储为 XML 数据就有许多优点。

### 30.8.1 包含 XML 数据的表

创建包含 XML 数据的表非常简单，只需给列选择 Xml 数据类型即可。下面的 SQL 命令 CREATE TABLE 创建了 Exams 表，其中的 ID 列是主键码，Number 和 Info 列的类型是 xml：

```
CREATE TABLE [dbo].[Exams] (
    [Id] [int] IDENTITY(1,1) NOT NULL,
    [Number] [nvarchar] (10) NOT NULL,
    [Info] [xml] NOT NULL,
    CONSTRAINT [PK_Exams] PRIMARY KEY CLUSTERED
    (
        [Id] ASC
    ) ON [PRIMARY]
) ON [PRIMARY]
```

现在进行一个简单的测试，用如下数据填充表：

```
INSERT INTO Exams values('70-536',
    '<Exam Number="70-536">
    <Title>TS: Microsoft .NET Framework 2.0 - Application Development Foundation
```



```

</Title>
<Certification Name="MCTS Windows Applications" Status="Core" />
<Certification Name="MCTS Web Applications" Status="Core" />
<Certification Name="MCTS Distributed Applications" Status="Core" />
<Topic>Developing applications that use system types and collections</Topic>
<Topic>Implementing service processes, threading, and application domains
</Topic>
<Topic>Embedding configuration, diagnostics, management, and installation features
</Topic>
<Topic>Implementing serialization and input/output functionality</Topic>
<Topic>Improving the security</Topic>
<Topic>Implementing interoperability, reflection, and mailing functionality
</Topic>
<Topic>Implementing globalization, drawing, and text manipulation functionality
</Topic>
</Exam>')

INSERT INTO Exams values('70-528',
'<Exam Number="70-528">
<Title>TS: Microsoft .NET Framework - Web-Based Client Development</Title>
<Certification Name="MCTS Web Applications" Status="Core" />
<Course>2541</Course>
<Course>2542</Course>
<Course>2543</Course>
<Course>2544</Course>
<Topic>Creating and Programming a Web Application</Topic>
<Topic>Integrating Data in a Web Application by using ADO.NET, XML, and
Data-Bound Controls</Topic>
<Topic>Creating Custom Web Controls</Topic>
<Topic>Tracing, Configuring, and Deploying Applications</Topic>
<Topic>Customizing and Personalizing a Web Application</Topic>
<Topic>Implementing Authentication and Authorization</Topic>
<Topic>Creating ASP.NET Mobile Web Applications</Topic>
</Exam>')

INSERT INTO Exams values('70-526',
'<Exam Number="70-526">
<Title>TS: Microsoft .NET Framework 2.0 - Windows-Based Client Development
</Title>
<Certification Name="MCTS Windows Applications" Status="Core" />
<Course>2541</Course>
<Course>2542</Course>
<Course>2546</Course>
<Course>2547</Course>
<Topic>Creating a UI for a Windows Forms Application by Using Standard Controls
</Topic>
<Topic>Integrating Data in a Windows Forms Application</Topic>
<Topic>Implementing Printing and Reporting Functionality</Topic>
<Topic>Enhancing Usability</Topic>
<Topic>Implementing Asynchronous Programming Techniques to Improve the User
Experience</Topic>
<Topic>Developing Windows Forms Controls</Topic>
<Topic>Configuring and Deploying Applications</Topic>
</Exam>')
```

### 30.8.2 读取 XML 值

在 ADO.NET 中, 可以用 `SqlDataReader` 对象读取 XML 数据。`SqlDataReader` 的 `GetSqlXml()` 方法返回一个 `SqlXml` 对象。`SqlXml` 类有一个 `Value` 属性, 它返回完整的 XML 表示, `SqlXml`



类的 `CreateReader()` 方法返回一个 `XmlReader` 对象。

`XmlReader` 的 `Read()` 方法在 `while` 循环中重复执行，逐个节点地读取数据。在输出中，我们仅看看 `Number` 属性的值、元素 `Title` 和 `Course` 的值。读取器定位的节点与相应的 XML 元素名比较，把对应的值写入控制台。

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using System.Xml;

class Program
{
    static void Main()
    {
        string connectionString =
            @"server=(local);database=ProCSharp;trusted_connection=true";
        SqlConnection connection = new SqlConnection(connectionString);
        SqlCommand command = connection.CreateCommand();
        command.CommandText = "SELECT Id, Number, Info FROM Exams";
        connection.Open();
        SqlDataReader reader = command.ExecuteReader(
            CommandBehavior.CloseConnection);
        while (reader.Read())
        {
            SqlXml xml = reader.GetSqlXml(2);

            XmlReader xmlReader = xml.CreateReader();

            StringBuilder courses = new StringBuilder("Course(s): ", 40);

            while (xmlReader.Read())
            {
                if (xmlReader.Name == "Exam" && xmlReader.IsStartElement)
                {
                    Console.WriteLine("Exam: {0}",
                        xmlReader.GetAttribute("Number"));
                }
                else if (xmlReader.Name == "Title" && xmlReader.IsStartElement)
                {
                    Console.WriteLine("Title: {0}", xmlReader.ReadString());
                }
                else if (xmlReader.Name == "Course" &&
                    xmlReader.IsStartElement)
                {
                    courses.AppendFormat("{0} ", xmlReader.ReadString());
                }
            }
            xmlReader.Close();
            Console.WriteLine(courses.ToString());
            Console.WriteLine();
        }
        reader.Close();
    }
}
```

运行应用程序，结果如下：

Exam: 70-536

Title: TS: Microsoft .NET Framework 2.0 - Application Development Foundation  
Course(s): 2956 2957

Exam: 70-528

Title: TS: Microsoft .NET Framework 2.0 - Web-Based Client Development  
Course(s): 2541 2542 2543 2544

Exam: 70-526

Title: TS: Microsoft .NET Framework 2.0 - Windows-Based Client Development  
Course(s): 2541 2542 2546 2547

除了使用 `XmlReader` 类之外,还可以使用 DOM 模型把完整的 XML 内容读入 `XmlDocument` 类,分析元素。`SelectSingleNode()` 方法的参数是一个 XPath 表达式,返回一个 `XmlNode` 对象。XPath 表达式 `//Exam` 在完整的 XML 树中查找 XML 元素 `Exam`。返回的 `XmlNode` 对象可用于读取所表示元素的子元素。访问 `Number` 属性的值,把测验数写入控制台,再访问 `Title` 元素,把 `Title` 元素的内容写入控制台,所有 `Course` 元素的内容也都写入控制台。

```
string connectionString =
    @"server=(local);database=ProCSharp;trusted_connection=true";
SqlConnection connection = new SqlConnection(connectionString);
SqlCommand command = connection.CreateCommand();
command.CommandText = "SELECT Id, Number, Info FROM Exams";
connection.Open();
SqlDataReader reader = command.ExecuteReader(
    CommandBehavior.CloseConnection);
while (reader.Read())
{
    SqlXml xml = reader.GetSqlXml(2);
    XmlDocument doc = new XmlDocument();
    doc.LoadXml(xml.Value);
    XmlNode examNode = doc.SelectSingleNode("//Exam");
    Console.WriteLine("Exam: {0}",
        examNode.Attributes["Number"].Value);
    XmlNode titleNode = examNode.SelectSingleNode("./Title");
    Console.WriteLine("Title: {0}", titleNode.InnerText);
    Console.Write("Course(s): ");
    foreach (XmlNode courseNode in examNode.SelectNodes("./Course"))
    {
        Console.Write("{0} ", courseNode.InnerText);
    }
    Console.WriteLine();
}
reader.Close();
```

#### 提示:

`XmlReader` 和 `XmlDocument` 类详见第 28 章。

在 .NET 3.5 中,还有另一个选项可以访问数据库中的 XML 列。可以合并 LINQ to SQL 和 LINQ to XML,使编程代码更短。

从 Data templates 类别中选择 LINQ to SQL Classes 模板,就可以使用 LINQ to SQL 设计器。把文件命名为 `ProCSharp.dbml`,给 `ProCSharp` 数据库创建一个映射。把 `Exams` 表从 `Solution Explorer` 拖放到设计界面上,创建该映射,如图 30-4 所示。

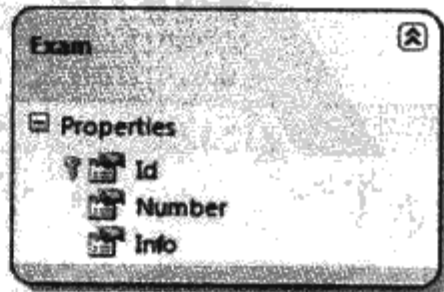


图 30-4

设计器创建的映射类是 ProCSharpDataContext, 它定义了一个 Exams 属性, 用于返回所有的 exam 行。这里使用一个 foreach 语句迭代所有的记录。当然, 如果不需要所有的记录, 就可以定义一个包含 where 表达式的 LINQ 查询。Exam 类根据数据库表中的列定义了 Id、Number 和 Info 属性。Info 属性的类型是 XDocument, 所以可以使用 System.Xml.Linq 命名空间中的 LINQ to XML 新类来访问。调用 Element() 方法, 传送 XML 元素名 Exam, 返回一个 XElement 对象, 该对象可以用更简单的方式访问 Number 属性、元素 Title 和 Course 的值, 与前面的 XmlDocument 类相同。

```
using System;
using System.Xml.Linq;

namespace Wrox.ProCSharp.SqlServer
{
    class Program
    {
        static void Main()
        {
            ProCSharpDataContext db = new ProCSharpDataContext();

            foreach (Exam item in db.Exams)
            {
                XElement exam = item.Info.Element("Exam");
                Console.WriteLine("Exam: {0}", exam.Attribute("Number").Value);
                Console.WriteLine("Title: {0}", exam.Element("Title").Value);
                Console.Write("Course(s): ");
                foreach (var course in exam.Elements("Course"))
                {
                    Console.Write("{0} ", course.Value);
                }
                Console.WriteLine();
            }
        }
    }
}
```

提示:

LINQ to SQL 和 LINQ to XML 详见第 27 和 29 章。

### 30.8.3 数据的查询

前面还没有涉及到 XML 数据类型的主要特性。SQL 语句 SELECT 可以与 XML XQuery 合并起来。

SELECT 语句与 XQuery 表达式合并起来, 读取 XML 值, 如下所示:

```
SELECT [Id], [Number], [Info].query('/Exam/Course') AS Course FROM [Exams]
```

XQuery 表达式/Exam/Course 访问 Exam 元素的 Course 子元素, 这个查询的结果返回 id、测验数和课程。

```
1 70-536 < Course > 2956 < /Course > < Course > 2957 < /Course >
2 70-528 <Course>2541</Course><Course>2542</Course><Course>2543</Course>
      <Course>2544</Course>
3 70-526 <Course>2541</Course><Course>2542</Course><Course>2546</Course>
```

```
<Course>2547</Course>
```

在 XQuery 表达式中, 可以创建复杂的语句, 查询单元格中 XML 内容的数据。下面的例子把测验信息转换为 XML, 列出课程的信息:

```
SELECT Info.query('
  for $course in /Exam/Course
  return
    <Course>
      <Exam>{ data(/Exam[1]/@Number) }<Exam>
      <Number>{ data($course) }</Number>
    </Course>')
AS Course
FROM Exams
WHERE Id=2
```

这里用 SELECT [Info]... FROM Exams WHERE Id = 2 选择了一行。在这个 SQL 查询的结果中, 使用了 XQuery 表达式的 for 和 return 语句。for \$course in /Exam/Course 迭代所有的 Course 元素。\$course 声明一个变量, 它用每个迭代来设置(类似于 C# 的 foreach 语句)。在 return 语句的后面指定了每一行的查询结果。每个课程元素的结果都放在 <Course> 元素中。<Course> 元素中嵌入了 <Exam> 和 <Number>。<Exam> 元素中的文本用 data(/Exam[1] /@Number) 定义。data() 是一个 XQuery 函数, 返回用参数指定的节点值。/Exam[1] 节点用于访问第一个 <Exam> 元素, @Number 指定 XML 属性 Number。<Number> 元素中的文本在 \$course 变量中定义。

注意:

在 C# 中, 集合中的第一个元素用索引 0 访问, 但在 XPath 中, 集合中的第一个元素用索引 1 访问。

这个查询的结果如下:

```
<Course>
  <Exam>70-528</Exam>
  <Number>2541</Number>
</Course>
<Course>
  <Exam>70-528</Exam>
  <Number>2542</Number>
</Course>
<Course>
  <Exam>70-528</Exam>
  <Number>2543</Number>
</Course>
<Course>
  <Exam>70-528</Exam>
  <Number>2544</Number>
</Course>
```

可以修改 XQuery 语句, 使之包含一个 where 子句, 来过滤 XML 元素。下面的例子仅从 XML 列中返回数值课程号大于 2542 的课程:

```
SELECT [Info].query('
  for $course in /Exam/Course
  where ($course > 2542)
  return
```



```
<Course>
  <Exam>{ data(/Exam[1]/@Number) }</Exam>
  <Number>{ data($course) }</Number>
</Course>')
AS Course
FROM [Exams]
WHERE Id=2
```

结果变成只有两个课程号:

```
<Course
  <Exam>70-528</Exam>
  <Number>2543</Number>
</Course>
<Course>
  <Exam>70-528</Exam>
  <Number>2544</Number>
</Course>
```

SQL Server 中的 XQuery 可以使用几个其他的 XQuery 函数, 获得最小值、最大值、总和、处理字符串、数字, 检查集合中的位置等。

下面的例子使用 count() 函数获得 /Exam/Course 元素的个数:

```
SELECT [Id], [Number], [Info].query('
  count(/Exam/Course)')
  AS CourseCount
FROM [Exams]
```

返回的数据显示测验的课程数:

Id	Number	CourseCount
1	70-536	0
2	70-528	4
3	70-526	4

### 30.8.4 XML 数据修改语言(XML DML)

W3C (<http://www.w3c.org>) 在定义 XQuery 时, 只允许查询数据。由于 XQuery 有这个限制, Microsoft 定义了 XQuery 的扩展, 称为 XML 数据修改语言(XML DML)。有了 XML DML, 就可以修改 XML 数据。insert、delete 和 replace value of 关键字扩展了 XQuery。

本节介绍在单元格中插入、删除和修改 XML 内容的一些例子。

可以使用 insert 关键字在 XML 列中插入一些 XML 内容, 而无需替换整个 XML 单元格。这里 <Course>2555</Course> 插入为第一个 Exam 元素的最后一个子元素:

```
UPDATE Exams
SET Info.modify('
  insert <Course>2555</Course> as last into Exam[1]')
WHERE id=3
```

XML 内容可以用 delete 关键字删除。在第一个 Exam 元素中, 删除最后一个 Course 元素。用 last() 函数选择最后一个元素:

```
UPDATE Exams
SET Info.modify('
```



```
delete /Exam[1]/Course[last()])
FROM Exams WHERE id=3
```

还可以修改 XML 内容, 这需要使用 `replace value of` 关键字。 `/Exam/Course[text() = 2543]` 表达式只访问文本内容包含字符串 2543 的子元素 `Course`。在这些元素中, `text()` 函数只访问要替换的文本内容。如果查询只返回一个元素, 还需要指定只替换一个元素。这就是返回的第一个文本元素用 `[1]` 指定的原因。2599 指定新的课程号是 2599:

```
UPDATE [Exams]
SET [Info].modify('
  replace value of (/Exam/Course[text() = 2543]/text())[1] with 2599')
FROM [Exams]
```

### 30.8.5 XML 索引

如果经常在 XML 数据中搜索某些特定的元素, 就可以在 XML 数据类型中指定索引。在 XML 索引中, 其类型必须区分为主 XML 索引或次 XML 索引。创建主 XML 索引是为了完整、一致地表示 XML 值。

下面的 SQL 命令 `CREATE PRIMARY XML INDEX` 在 `Info` 列上创建了索引 `idx_exams`:

```
CREATE PRIMARY XML INDEX idx_exams on Exams (Info)
```

如果查询包含 XPath 表达式, 以直接访问 XML 类型的 XML 元素, 主索引就没有什么帮助。对于 XPath 和 XQuery 表达式, 可以使用 XML 次索引。要创建 XML 次索引, 主索引必须存在。有了次索引, 就必须区分索引类型:

- PATH 索引
- VALUE 索引
- PROPERTY 索引

如果使用了 `exists()` 或 `query()` 函数, 且通过 XPath 表达式访问 XML 元素, 就使用 PATH 索引。使用 XPath 表达式 `/Exam/Course` 时, 建立 PATH 索引是有帮助的:

```
CREATE XML INDEX idx_examNumbers on Exams (Info)
USING XML INDEX idx_exams FOR PATH
```

如果属性是使用 `value()` 函数从元素中提取出来的, 就使用 PROPERTY 索引。包含创建索引的 `FOR PROPERTY` 语句定义了一个 PROPERTY 索引:

```
CREATE XML INDEX idx_examsNumbers on Exams (Info)
USING XML INDEX idx_exams FOR PROPERTY
```

如果元素是使用 XPath 子轴或自轴表达式通过树型结构搜索出来的, 使用 VALUE 索引会得到最佳性能。XPath 表达式 `//Certification` 会搜索带有子轴或自轴的所有 `Certification` 元素。`[@Name="MCTS Web Applications"]` 表达式只返回属性 `Name` 的值是 `MCTS Web Applications` 的元素。

```
SELECT [Info].query('/Exam/Title/text()') FROM [Exams]
WHERE [Info].exist('//Certification[@Name="MCTS Web Applications"]') = 1
```

返回的结果列出了包含所请求证书的测验名:

TS: Microsoft .NET Framework 2.0 - Application Development Foundation  
 TS: Microsoft .NET Framework - Web-Based Client Development

VALUE 索引用 FOR VALUE 语句创建:

```
CREATE XML INDEX idx_examNumbers on Exams (Info)
  USING XML INDEX idx_exams FOR VALUE
```

### 30.8.6 强类型化的 XML

SQL Server 中的 XML 数据类型也可以用 XML 模式强类型化。有了强类型化的 XML 列，就可以在插入 XML 数据时验证该数据是否遵循模式。

XML 模式可以用语句 CREATE XML SCHEMA COLLECTION 创建。下面的语句创建了一个 XML 模式 CourseSchema，它定义了 CourseElt 类型，其中包含一系列 Number 和 Title，它们的类型都是 string，CourseElt 类型还包含一个元素 Any，它可以是任意类型。Number 和 Title 只能出现一次。因为 Any 的 minOccurs 属性设置为 0，maxOccurs 属性设置为 unbounded，所以这个元素是可选的。于是可以在未来的版本中给 CourseElt 类型添加额外的信息，而模式仍是有效的。最后，Course 元素名是 CourseElt 类型。

```
CREATE XML SCHEMA COLLECTION CourseSchema AS
'<?xml version="1.0" encoding="UTF-8"?>
<xs:schema id="Courses" targetNamespace="http://thinktecture.com/Courses.xsd"
  elementFormDefault="qualified" xmlns="http://thinktecture.com/Courses.xsd"
  xmlns:mstns="http://thinktecture.com/Courses.xsd"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="CourseElt">
    <xs:sequence>
      <xs:element name="Number" type="xs:string" maxOccurs="1" minOccurs="1" />
      <xs:element name="Title" type="xs:string" maxOccurs="1" minOccurs="1" />
      <xs:element name="Any" type="xs:anyType"
        maxOccurs="unbounded" minOccurs="0" />
    </xs:sequence>
  </xs:complexType>
  <xs:element name="Course" type="CourseElt">
  </xs:element>
</xs:schema>'
```

使用这个模式，有效的 XML 如下所示：

```
<Course xmlns="http://thinktecture.com/Courses.xsd">
<Number>2549</Number>
<Title>Advanced Distributed Application Development with Visual Studio 2005</Title>
</Course>
```

在 Visual Studio Database 项目类型中，不支持给数据库添加模式。这个特性不能在 Visual Studio 2008 的 GUI 中使用，但可以手工实现。要使用 Visual Studio 2008 创建 XML 模式，可以使用空的 Visual Studio 项目类型，给项目添加一个新的 XML 模式，再把模式的 XML 语法复制到 CREATE XML SCHEMA 语句中。

除了使用 Visual Studio 之外，还可以把 XML 语法复制到 SQL Server Management Studio 中，创建并查看 XML 模式，如图 30-5 所示。对象浏览器在 Types 项的下面列出了 XML 模式。

用 XML 数据类型进行设置，就可以把 XML 模式赋予一列：

```
CREATE TABLE Courses
(
    [Id] [int] IDENTITY(1,1) NOT NULL,
    [Course] [xml]([dbo].[CourseSchema]) NOT NULL
)
```

用 Visual Studio 2008 或 SQL Server Management Studio 创建表，设置属性 XML schema namespace，就可以把 XML 模式赋予一列。

现在再给 XML 列添加数据时，会验证其模式。如果 XML 不符合模式定义，就会抛出一个 SqlException 异常，显示一个 XML Validation 错误。

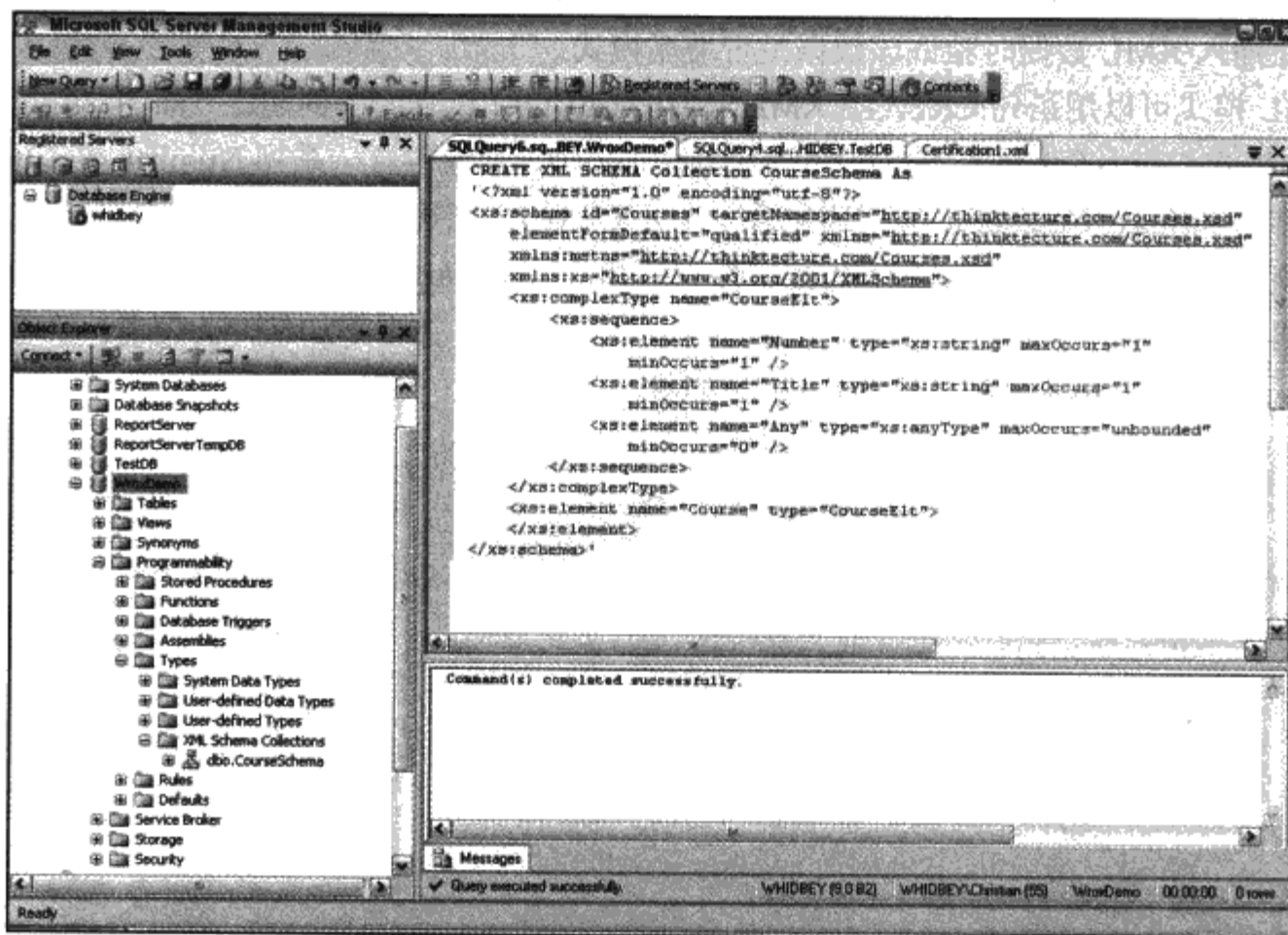


图 30-5

## 30.9 小结

本章讨论了 SQL Server 中与 CLR 功能相关的新特性。CLR 由 SQL Server 执行，所以可以用 C# 创建用户定义的类型、合计函数、存储过程、函数和触发器。

用户定义的类型对 .NET 类有一些严格的要求，以转换为字符串，或从字符串转换回来。数据在 SQL Server 中的内部存储方式取决于在类型中定义的格式。用户定义的合计函数可以使用 .NET 类进行定制的计算。通过存储过程和函数，可以将 CLR 类用于服务器端代码。

使用 CLR 和 SQL Server，并不表示 T-SQL 已过时。如果进行数据密集型的查询，T-SQL 的优点是代码较少。如果使用加密等 .NET 特性，CLR 类可以改进数据的处理。

本章还介绍了 SQL Server 的 XML 数据类型，把 XQuery 表达式和 T-SQL 语句合并起来。

这是第 IV 部分的最后一章。第 V 部分详细介绍了如何定义应用程序的用户界面。在用户界面上可以使用 Windows 窗体、WPF 和 ASP.NET。

# 第 V 部分

## 显 示

- 第 31 章 Windows 窗体
- 第 32 章 数据绑定
- 第 33 章 使用 GDI+绘图
- 第 34 章 Windows Presentation Foundation
- 第 35 章 高级 WPF
- 第 36 章 插件
- 第 37 章 ASP.NET 页面
- 第 38 章 ASP.NET 开发
- 第 39 章 ASP.NET AJAX
- 第 40 章 Visual Studio Tools for Office



# 第 31 章

## Windows 窗体

基于 Web 的应用程序在过去几年非常流行。从管理员的角度来看,把所有的应用程序逻辑放在一个中央服务器上是非常吸引人的。但部署基于客户的软件会非常困难,特别是部署基于 COM 的客户软件。基于 Web 的应用程序的缺点是它们不能提供丰富的用户体验。.NET Framework 允许开发人员创建丰富、智能的客户应用程序,而且不再有部署问题和以前的 DLL Hell。无论选择 Windows 窗体还是 WPF(参见第 34 章),客户应用程序都不再难以开发或部署。

Windows 窗体已经对 Windows 开发产生了影响。当应用程序处于初始设计阶段时,是建立基于 Web 的应用程序还是建立客户应用程序已经很难抉择了。Windows 客户应用程序开发起来非常快速和高效,它们可以为用户提供丰富的体验。

Visual Basic 开发人员对 Windows 窗体应比较熟悉。创建新窗体(也称为窗口或对话框)也采用把控件从工具箱拖放到窗体设计器上的方式。但是,如果您在创建消息泵和监视消息时使用的是 C 样式的传统 Windows 编程,或者您是一位 MFC 程序员,就会发现现在可以获得需要的低级内部功能了。现在可以重写 `wndproc`,捕获这些消息,但常常并不是真需要它们。

本章将主要介绍 Windows 窗体的如下方面:

- Form 类
- Windows 窗体的类层次结构
- `System.Windows.Forms` 命名空间中的控件和组件
- 菜单和工具栏
- 创建控件
- 创建用户控件

### 31.1 创建 Windows 窗体应用程序

首先需要创建一个 Windows 窗体应用程序。下面的示例创建了一个空白窗体,并把它显示在屏幕上。这个示例没有使用 Visual Studio 2008,而是在文本编辑器中输入代码,使用命令行编译器进行编译。下面是代码清单:

```
using System;
using System.Windows.Forms;
namespace NotepadForms
{
    public class MyForm : System.Windows.Forms.Form
```



```
{
    public MyForm()
    {
    }

    [STAThread]
    static void Main()
    {
        Application.Run(new MyForm());
    }
}
```

在编译和运行这个示例时，会得到一个没有标题的小空白窗体。该窗体没有什么实际功能，但它却是一个 Windows 窗体。

代码中有两个地方需要注意。第一个是使用继承功能来创建 MyForm 类。下面的代码声明 MyForm 派生于 System.Windows.Forms。

```
public class MyForm : System.Windows.Forms.Form
```

Form 类是 System.Windows.Forms 命名空间的一个主要类。代码的其他部分如下：

```
[STAThread]
static void Main()
{
    Application.Run(new MyForm());
}
```

Main 是 C# 客户应用程序的默认入口。一般在大型应用程序中，Main() 方法不位于窗体中，而是位于类中，它负责完成需要的启动处理。在本例中，我们在项目属性对话框中设置启动的类名。注意属性 [STAThread]，它把 COM 线程模型设置为单线程单元 (Single-Threaded Apartment, STA)。COM 交互操作需要 STA 线程模型，默认为添加到 Windows 窗体项目中。

Application.Run() 方法负责启动标准的应用程序消息循环。它有 3 个重载版本：第一个重载版本不带参数，第二个重载版本把 ApplicationContext 对象作为其参数，本例中的第三个重载版本把窗体对象作为其参数。在这个示例中，MyForm 对象是应用程序的主窗体，这表示在关闭这个窗体时，应用程序就结束了。使用 ApplicationContext 类，可以对主消息循环何时结束和应用程序何时退出有更多的控制权。

Application 类包含一些非常有用的功能。它提供了一些静态方法和属性，用于控制应用程序的启动和停止过程，访问由应用程序处理的 Windows 消息。表 31-1 列出了其中一些比较有用的方法和属性。

表 31-1

方法/属性	说 明
CommonAppDataPath	对应用程序的所有用户都通用的数据路径。一般是 BasePath\Company Name\Product Name\Version，其中 BasePath 是 C:\Documents and Settings\username\ApplicationData。如果该路径不存在，就创建一个
ExecutablePath	这是启动应用程序的可执行文件的路径和文件名

(续表)

方法/属性	说 明
LocalUserAppDataPath	类似于 CommonAppDataPath，但这个属性支持漫游
MessageLoop	如果在当前线程上存在消息循环，就返回 True，否则返回 false
StartupPath	类似于 ExecutablePath，但不返回文件名
AddMessageFilter	用于预处理消息。在基于 IMessageFilter 的对象上执行，消息可以从消息循环中过滤出来，或者在消息发送到循环中之前进行特殊的处理
DoEvents	类似于 Visual Basic 的 DoEvents 语句，允许处理队列中的消息
EnableVisualStyles	允许对应用程序的各种可视化元素使用 XP 可视化样式。它有两个重载版本，接收清单信息。一个重载版本的参数是清单流，另一个重载版本的参数是清单所在的完整名称和路径
Exit 和 ExitThread	Exit 结束所有当前运行的消息循环，并退出应用程序。ExitThread 只结束消息循环，关闭当前线程上的所有窗口

在 Visual Studio 2008 中生成这个示例时，它会是什么样子？首先要注意，创建了两个文件，其原因是 Visual Studio 2008 利用了 Framework 的部分类特性，把设计器生成的代码放在一个独立的文件中。使用默认名称 Form1，这两个文件就是 Form1.cs 和 Form1.Designer.cs。除非选择了 Project 菜单中的 Show All Files 选项，否则在 Solution Explorer 中看不到 Form1.Designer.cs。下面是 Visual Studio 2008 为两个文件生成的代码。第一个文件是 Form1.cs：

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace VisualStudioForm
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
    }
}
```

这很简单，其中包含一些 using 语句和一个简单的构造函数。接着是 Form1.Designer.cs 的代码：

```
namespace VisualStudioForm
{
    partial class Form1
    {
        ...
    }
}
```

```

    /// < summary >
    /// Required designer variable.
    /// < /summary >
    private System.ComponentModel.IContainer components = null;
    /// < summary >
    /// Clean up any resources being used.
    /// < /summary >
    /// < param name="disposing" > true if managed resources should be disposed;
    /// otherwise, false. < /param >
    protected override void Dispose(bool disposing)
    {
        if (disposing && (components != null))
        {
            components.Dispose();
        }
        base.Dispose(disposing);
    }
    #region Windows Form Designer generated code
    /// < summary >
    /// Required method for Designer support - do not modify
    /// the contents of this method with the code editor.
    /// < /summary >
    private void InitializeComponent()
    {
        this.components = new System.ComponentModel.Container();
        this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
        this.Text = "Form1";
    }
    #endregion
}

```

窗体的设计器文件一般不应直接编辑。唯一的例外是要在 `Dispose` 方法中进行特殊的处理。`InitializeComponent` 方法详见本章后面的内容。

首先，这个示例应用程序的代码比简单的命令行示例长。在类的开头有好几个 `using` 语句，在本例中大多数是不必要的。但保留它们并无大碍。类 `Form1` 派生于 `System.Windows.Forms`，与前面的 `Notepad` 示例一样，但代码从一开始就不同。`Form1.Designer` 文件的第一行代码如下：

```
private System.ComponentModel.Container components = null;
```

在这个示例中，这行代码并没有做什么工作。当给窗体添加组件时，也就把该组件添加给了组件对象，该组件对象是一个容器。添加到这个容器中的原因与窗体的释放有关。窗体类支持 `IDisposable` 接口，因为它是在 `Component` 类中执行的。在组件添加到组件容器中时，容器将确保组件被正确地跟踪，并在释放窗体时释放它。如果查看代码中的 `Dispose` 方法，就可以看到它：

```

protected override void Dispose(bool disposing)
{
    if (disposing && (components != null))
    {
        components.Dispose();
    }
    base.Dispose(disposing);
}

```

在此可以看到，在调用 `Dispose` 方法时，也会调用组件对象的 `Dispose` 方法，因为组件对象包含其他组件，所以它们也会被释放。

`Form1` 类的构造函数在 `Form1.cs` 中，如下所示：

```
public Form1()
{
    InitializeComponent();
}
```

注意对 `InitializeComponent()` 的调用。`InitializeComponent()` 在 `Form1.Designer.cs` 中，顾名思义，`InitializeComponent()` 初始化了添加到窗体上的所有控件，还初始化了窗体的属性。在本示例中，`InitializeComponent()` 如下所示：

```
private void InitializeComponent()
{
    this.components = new System.ComponentModel.Container();
    this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
    this.Text = "Form1";
}
```

这是很基本的初始化代码。该方法与 Visual Studio 的设计器相关联。使用设计器修改窗体时，这些改动会在 `InitializeComponent()` 中反映出来。如果在 `InitializeComponent()` 中修改了任意类型的代码，下次在设计器中进行修改时，这些改动就会丢失。每次在设计器中进行修改后，`InitializeComponent()` 都会重新生成。如果需要为窗体或窗体上的控件和组件添加其他初始化代码，就应在调用 `InitializeComponent()` 后添加。`InitializeComponent()` 还负责实例化控件，这样在 `InitializeComponent()` 之前所有引用控件的调用都会失败，并生成一个空引用异常。

要在窗体上添加控件或组件，可以按下 `Ctrl+Alt+X` 或者在 Visual Studio 2008 的 View 菜单中选择 Toolbox。此时 `Form1` 应处于设计模式。在 Solution Explorer 中右击 `Form1.cs`，从弹出的菜单中选择 View Designer。选择 Button 控件，把它拖放到设计器的窗体上。也可以双击该控件，把它添加到窗体上。对 TextBox 控件进行相同的操作。

在窗体上添加了 TextBox 控件和 Button 控件后，`InitializeComponent()` 就会扩展，包含如下代码：

```
private void InitializeComponent()
{
    this.button1 = new System.Windows.Forms.Button();
    this.textBox1 = new System.Windows.Forms.TextBox();
    this.SuspendLayout();
    //
    // button1
    //
    this.button1.Location = new System.Drawing.Point(77, 137);
    this.button1.Name = "button1";
    this.button1.Size = new System.Drawing.Size(75, 23);
    this.button1.TabIndex = 0;
    this.button1.Text = "button1";
    this.button1.UseVisualStyleBackColor = true;
    //
    // textBox1
    //
    this.textBox1.Location = new System.Drawing.Point(67, 75);
```

```

this.textBox1.Name = "textBox1";
this.textBox1.Size = new System.Drawing.Size(100, 20);
this.textBox1.TabIndex = 1;
//
// Form1
//
this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
this.ClientSize = new System.Drawing.Size(284, 264);
this.Controls.Add(this.textBox1);
this.Controls.Add(this.button1);
this.Name = "Form1";
this.Text = "Form1";
this.ResumeLayout(false);
this.PerformLayout();
}

```

如果查看该方法中的前 3 行代码, 就会看到 `TextBox` 控件和 `Button` 控件被实例化了。注意给控件指定的名称 `textBox1` 和 `button1`。默认情况下, 设计器会使用控件的名称, 并在该名称的最后添加一个整数值。在添加另一个按钮时, 设计器会使用名称 `button2`, 依次类推。下一行代码是 `SuspendLayout` 和 `ResumeLayout` 对的部分。`SuspendLayout()` 临时挂起控件第一次初始化时发生的布局事件。在该方法的最后, 将调用 `ResumeLayout()` 方法, 把事件重置为正常状态。在包含许多控件的复杂窗体上, `InitializeComponent()` 方法会非常长。

要修改控件的属性值, 可以按下 F4, 或从 `View` 菜单中选择 `Properties Window`。该窗口允许修改控件或组件的大多数属性。在 `Properties` 窗口中进行了修改后, `InitializeComponent()` 方法就会重新编写, 以反映新属性值。例如, 如果在 `Properties` 窗口中把 `Text` 属性改为 `My Button`, `InitializeComponent()` 就会包含如下代码:

```

//
// button1
//
this.button1.Location = new System.Drawing.Point(77, 137);
this.button1.Name = "button1";
this.button1.Size = new System.Drawing.Size(75, 23);
this.button1.TabIndex = 0;
this.button1.Text = "My Button";
this.button1.UseVisualStyleBackColor = true;

```

如果使用的不是 `Visual Studio` 的编辑器, 就需要在设计中包含 `InitializeComponent()` 类型函数。把所有这些初始化代码放在一个地方, 有助于使构造函数更简洁, 如果有多个构造函数, 还需要确保能从每个构造函数中调用初始化代码。

## 类层次结构

在设计和构建定制控件时, 理解层次结构是非常重要的。如果定制控件派生于已有的控件, 例如对于带有额外属性和方法的文本框, 就应使定制控件派生于文本框控件, 再重写、添加属性和方法, 以满足要求。但是, 如果创建的控件与 .NET Framework 中的已有控件不匹配, 就必须从 3 个基类中派生: 如果需要自动滚动功能, 就从 `Control` 或 `Scrollable Control` 中派生, 如果控件应是其他控件的容器, 就应从 `ContainerControl` 类中派生。



本章的剩余内容将介绍其中的许多类，它们如何协同工作，以及如何使用它们建立具有专业化外观的客户应用程序。

## 31.2 Control 类

`System.Windows.Forms` 命名空间中有一个特殊的类，它是每个控件和窗体的基类，这个类就是 `System.Windows.Forms.Control`。`Control` 类执行核心功能，创建用户所见的界面。`Control` 类派生于 `System.ComponentModel.Component` 类。`Component` 类为 `Control` 类提供了必要的基础结构，在把控件拖放到设计界面上以及包含在另一个对象时需要它。`Control` 类为派生于它的类提供了一个很长的功能列表。这个列表太长，不能在这里全部列出，所以我们仅介绍 `Control` 类提供的比较重要的功能。本章的后面在介绍基于 `Control` 类的特定控件时，将在一些示例代码中论述属性和方法。下面几小节将按照功能组合方法和属性，把相关的功能放在一起进行讨论。

### 31.2.1 大小和位置

控件的大小和位置由属性 `Height`、`Width`、`Top`、`Bottom`、`Left`、`Right` 以及辅助属性 `Size` 和 `Location` 确定。区别是 `Height`、`Width`、`Top`、`Bottom`、`Left`、`Right` 属性值都是一个整数，而 `Size` 的值使用一个 `Size` 结构来表示，`Location` 的值使用一个 `Point` 结构来表示。`Size` 结构和 `Point` 结构都包含 `XY` 坐标。`Point` 结构一般相对于一个位置，而 `Size` 结构是对象的高和宽。`Size` 和 `Point` 都位于 `System.Drawing` 命名空间。它们非常类似，因为它们都提供了 `XY` 坐标对，还拥有用于简单的比较和转换的重写运算符。例如，可以对两个 `Size` 结构执行相加操作。对于 `Point` 结构，加法运算符已进行了重写，可以把 `Size` 结构加到 `Point` 结构上，得到一个新的 `Point`。其结果是给某个位置加上某个距离值，得到一个新位置。如果动态创建窗体或控件，这是非常方便的。

`Bounds` 属性返回一个 `Rectangle` 对象，它表示一个控件区域。这个区域包含滚动条和标题栏。`Rectangle` 也位于 `System.Drawing` 命名空间。`ClientSize` 属性是一个 `Size` 结构，表示控件的客户区域，不包含滚动条和标题栏。

`PointToClient` 和 `PointToScreen` 方法是方便的转换方法，它们的参数是 `Point` 结构，返回一个 `Point` 结构。`PointToClient` 的 `Point` 参数表示屏幕坐标，该方法把屏幕坐标转换为基于当前客户对象的坐标。这非常便于进行拖放操作。`PointToScreen` 正好与之相反，它提取客户对象的坐标，把它们转换为屏幕坐标。还有 `RectangleToScreen` 和 `ScreenToRectangle` 方法，它们具有相同的功能，只是用 `Rectangle` 结构代替 `Point` 结构。

`Dock` 属性确定子控件停放在父控件的哪条边上。`DockStyle` 枚举值用作其属性值。这个值可以是 `Top`、`Bottom`、`Right`、`Left`、`Fill` 和 `None`。`Fill` 会使控件的大小正好匹配父控件的客户区域。

`Anchor` 属性把子控件的一条边与父控件的一条边对齐，这与停靠不同，因为它不设置父控件的一条边，而是把到该边界的当前距离设置为常量。例如，如果把子控件的右边界与父控件的右边界对齐，并重新设置父控件的大小，子控件右边界到父控件右边界的距离将保持不变。

Anchor 属性采用 AnchorStyles 枚举的值, 其值是 Top、Bottom、Right、Left 和 None。通过设置该属性值, 可以在重新设置父控件的大小时, 动态地设置子控件的大小。这样, 当用户重新设置窗体的大小时, 按钮和文本框就不会被剪切或隐藏。

Dock 和 Anchor 属性与 Flow 和 Table 布局控件(详见本章后面的内容)一起使用时, 可以创建非常复杂的用户窗口。对于包含许多控件的复杂窗体来说, 窗口大小的重新设置比较困难。这些工具有助于完成这个任务。

### 31.2.2 外观

与控件外观相关的属性有 BackColor 和 ForeColor, 它们把 System.Drawing.Color 对象作为其值。BackgroundImage 属性把基于 Image 的对象作为其值。System.Drawing.Image 是一个抽象类, 用作 Bitmap 和 Metafile 类的基类。BackColorImageLayout 属性使用 ImageLayout 枚举设置图像在控件上的显示方式, 其有效值是 Center、Tile、Stretch、Zoom 和 None。

Font 和 Text 属性处理文字的显示。要修改 Font 属性, 需要创建一个 Font 对象。在创建 Font 对象时, 要指定字体名称、字号和样式。

### 31.2.3 用户交互操作

用户交互操作最好描述为控件创建和响应的各种事件。一些比较常见的事件有 Click、DoubleClick、KeyDown、KeyPress、Validating 和 Paint。

鼠标事件 Click、DoubleClick、MouseDown、MouseUp、MouseEnter、MouseLeave 和 MouseHover 处理鼠标和控件的交互操作。如果处理 Click 和 DoubleClick 事件, 每次捕获一个 DoubleClick 事件时, 也会引发 Click 事件。如果处理不正确, 就会出现我们不希望的结果。Click 和 DoubleClick 事件都把 EventArgs 作为其参数, 而 MouseDown 和 MouseUp 事件把 MouseEventArgs 作为其参数。MouseEventArgs 包含几个有用的信息, 例如单击的按钮、按钮被单击的次数、鼠标轮制动器(鼠标轮上的凹槽)的数目和鼠标的当前 XY 坐标。如果可以访问这些信息, 就必须处理 MouseDown 或 MouseUp 事件, 而不是 Click 或 DoubleClick 事件。

键盘事件的工作方式与此类似: 需要一些信息来确定处理什么事件。对于简单的情况, KeyPress 事件接收一个 KeyPressEventArgs, 它包含表示被按键的字符值 KeyChar。Handled 属性用于确定事件是否已处理。把 Handled 属性设置为 true, 事件就不会由操作系统进行默认处理。如果需要被按的键的更多信息, 则处理 KeyDown 或 KeyUp 事件会比较合适。它们都接收 KeyEventArgs。KeyEventArgs 中的属性包括 Ctrl、Alt 或 Shift 键是否被按下。KeyCode 属性返回一个 Keys 枚举值, 表示被按下的键。与 KeyPressEventArgs.KeyChar 不同, KeyCode 属性指定键盘上的每个键, 而不仅仅是字母数字键。KeyData 属性返回一个 Key 值, 还设置修饰符。修饰符与值进行 OR 运算, 指定是否同时按下了 Shift 或 Ctrl 键。KeyValue 属性是 Keys 枚举的整数值。Modifiers 属性包含一个 Keys 值, 它表示被按下的修饰符键。如果选择了多个修饰符键, 这些值就进行 OR 运算。键盘事件以下述顺序来引发:

(1) KeyDown

(2) KeyPress

### (3) KeyUp

Validating、Validated、Enter、Leave、GotFocus 和 LostFocus 事件都处理获得焦点(或被激活)和失去焦点的控件。在用户用 tab 键选择一个控件或用鼠标选择该控件时,该控件就获得了焦点。Enter、Leave、GotFocus 和 LostFocus 事件的功能似乎非常类似。GotFocus 和 LostFocus 事件是低级事件,与 Windows 消息 WM\_SETFOCUS 和 WM\_KILLFOCUS 相关。一般应尽可能使用 Enter 和 Leave 事件。Validating 和 Validated 事件在验证控件时发生。这些事件接收一个 CancelEventArgs,利用该参数,把 Cancel 属性设置为 true,就可以取消以后的事件。如果定制了验证代码,而且验证失败,就可以把 Cancel 属性设置为 true,且控件也不会失去焦点。Validating 事件在验证过程中发生,Validated 事件在验证过程后发生。这些事件的引发顺序如下:

- (1) Enter
- (2) GotFocus
- (3) Leave
- (4) Validating
- (5) Validated
- (6) LostFocus

理解这些事件的引发顺序是很重要的,可以避免不小心创建递归事件。例如,在控件的 LostFocus 事件中设置控件的焦点,就会创建一个消息死锁,且应用程序会停止响应。

#### 31.2.4 Windows 功能

System.Windows.Forms 命名空间是依赖 Windows 功能的少数几个命名空间之一。Control 类就是一个很好的示例。如果对 System.Windows.Forms.dll 进行反编译,就会看到 UnsafeNativeMethods 类的引用列表。.NET Framework 使用这个类封装所有的标准 Win32 API 调用。使用与 Win32 API 的交互操作,标准 Windows 应用程序的外观和操作系统就可以通过 System.Windows.Forms 命名空间获得。

支持与 Windows 交互操作的功能包括 Handle 和 IsHandleCreated 属性。Handle 属性返回一个包含控件 HWND(Windows 句柄)的 IntPtr。窗口句柄是唯一标识窗口的 HWND。可以将控件看作是一个窗口,所以它有相应的 HWND。可以使用 Handle 属性进行任意数量的 Win32 API 调用。

为了访问 Windows 消息,可以重写 WndProc 方法。该方法把一个 Message 对象作为其参数。Message 对象是 Windows 消息的一个简单封装器。它包含 HWnd、LParam、WParam、Msg 和 Result 属性。如果希望由系统处理消息,就必须确保把消息传送给 base.WndProc(msg)方法。如果希望自己处理消息,就不需要传送消息。

#### 31.2.5 杂项功能

一些条目较难分类,例如数据绑定功能。BindingContext 属性返回一个 BindingManagerBase 对象。DataBindings 集合包含一个 ControlBindingsCollection,它是控件的绑定对象集合,数据绑定详见第 32 章。

CompanyName、ProductName 和 Product 版本提供了控件的初始数据及其当前版本。

Invalidate 方法允许使控件的一个区域失效，以进行重新绘制。可以使整个控件失效，或指定要失效的区域或矩形。这会把一个绘制消息传送给控件的 WindProc。还可以同时使子控件失效。

组成 Control 类的还有几十个属性、方法和事件。这个列表列出了比较常用的成员，希望对您可用的功能有一个大致的了解。

### 31.3 标准控件和组件

前一节介绍了控件常用的一些方法和属性。本节将讨论.NET Framework 提供的各种控件，解释每个控件提供的附加功能。下载的示例(www.wrox.com)包含一个示例应用程序 FormExample，这个应用程序是一个 MDI 应用程序(稍后讨论)，包含一个窗体 frmControls，其中包含许多具备基本功能的控件。图 31-1 显示了这个示例的外观。

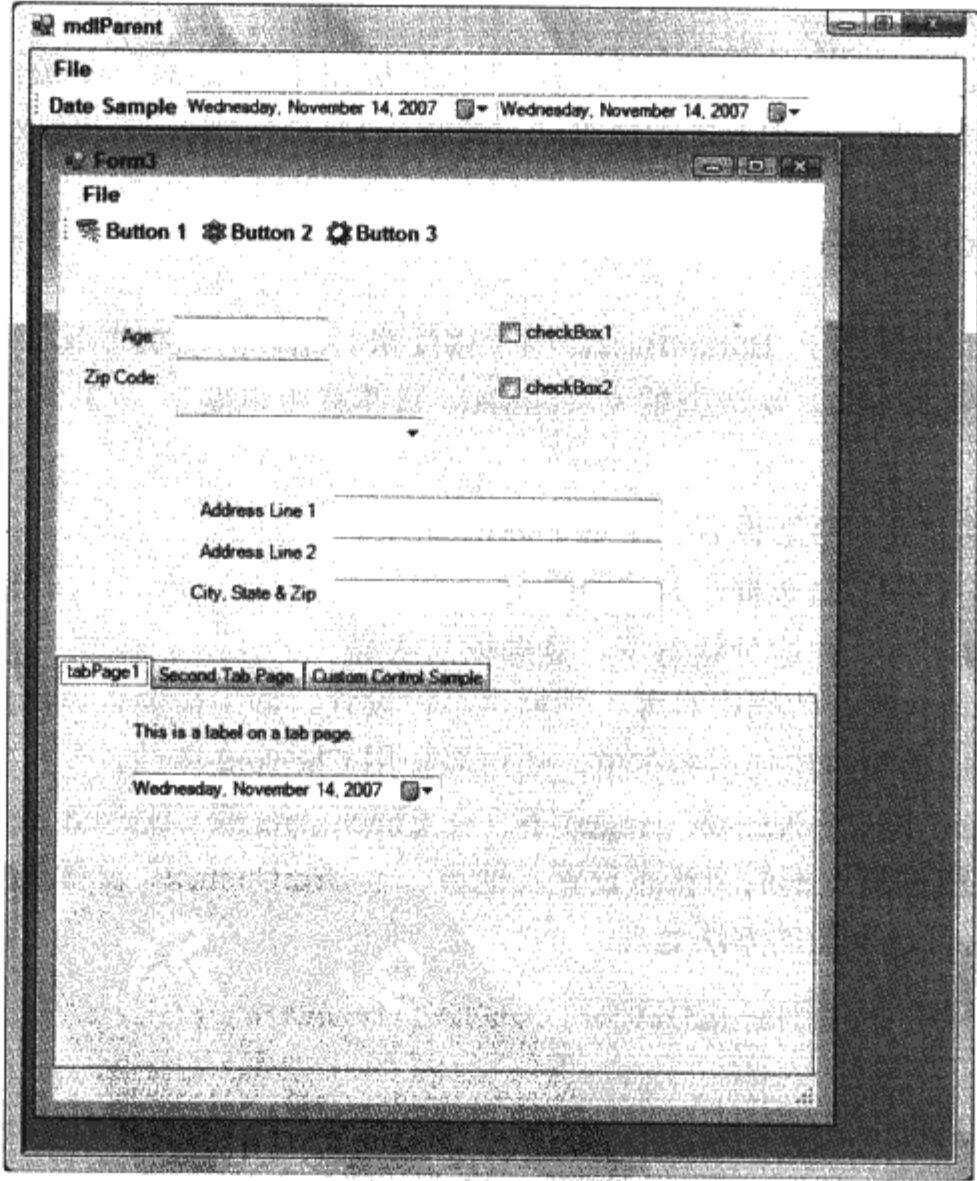


图 31-1

#### 31.3.1 Button 控件

Button 类表示简单的命令按钮，派生自 ButtonBase 类。该类最常见的用法是编写处理按钮

Click 事件的代码。下面的代码执行 Click 事件的处理程序。在单击按钮时，会弹出一个显示按钮名称的消息框。

```
private void btnTest_Click(object sender, System.EventArgs e)
{
    MessageBox.Show(((Button)sender).Name + " was clicked.");
}
```

在 PerformClick 方法中，可以模仿按钮上的 Click 事件，而无需用户单击按钮。NotifyDefault 方法把一个布尔值作为其参数，告诉按钮把它自己绘制为默认按钮。一般情况下，窗体上的默认按钮有略粗的边框。要把按钮标识为默认，可以把窗体上的 AcceptButton 属性设置为按钮。接着，在用户按下回车键时，就会引发默认按钮的单击事件。图 31-2 显示了标题为 Default 的默认按钮(注意黑色的边框)。

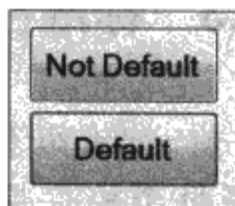


图 31-2

按钮可以包含图像和文本。图像通过 ImageList 对象或 Image 属性提供。ImageList 对象就是由放在窗体上的组件管理的一个图像列表。它们在本章的后面解释。

Text 和 Image 都包含 Align 属性，用以对齐按钮上的文本和图像。Align 属性使用 ContentAlignment 枚举的值。文本或图像可以与按钮的左、右、上、下边界对齐。

### 31.3.2 CheckBox 控件

CheckBox 控件也派生于 ButtonBase，用于接受来自用户的二状态或三状态响应。如果把 ThreeState 属性设置为 true，复选框的 CheckState 属性就可以是以下 3 个 CheckState 枚举值之一：

- Checked: 复选框有一个选中标记
- Unchecked: 复选框没有选中标记
- Indeterminate: 在这种状态下，复选框为灰显

Indeterminate 值只能在代码中设置，不能由用户设置。如果需要告诉用户选项还未设置，就可以使用这个值。如果希望使用布尔值，还可以使用 Checked 属性。

CheckedChanged 和 CheckStateChanged 事件在 CheckState 或 Checked 属性改变时发生。捕获的这些事件可以根据复选框的新状态设置其他值。在 frmControls 窗体类中，几个复选框的 CheckedChanged 事件由下面的方法处理：

```
private void checkBoxChanged(object sender, EventArgs e)
{
    CheckBox checkBox = (CheckBox)sender;
    MessageBox.Show(checkBox.Name + " new value is " + checkBox.Checked.ToString());
}
```

在每个复选框的 Checked 属性改变时，都会显示一个消息框，其中包含了改变的复选框名称和新值。



### 31.3.3 RadioButton 控件

最后一个派生自 `ButtonBase` 的控件是 `RadioButton`(单选按钮)。单选按钮一般用作一个组，有时称为选项按钮。单选按钮允许用户从几个选项中选择一个。当同一个容器中有多个 `RadioButton` 控件时，一次只能选择一个按钮。所以如果有 3 个选项，例如 `Red`、`Green` 和 `Blue`，如果 `Red` 选项被选中，而用户单击 `Blue`，则 `Red` 会自动取消选中。

`Appearance` 属性使用 `Appearance` 枚举值，即 `Button` 或 `Normal`。当选择 `Normal` 时，单选按钮看起来像一个小圆圈，在它的旁边有一个标签。选择按钮会填充圆圈，选择另一个按钮会取消对当前选中按钮的选择，使圆圈为空。当选中 `Button` 时，`RadioButton` 控件看起来像一个标准按钮，但工作方式类似于开关，选中是指焦点在位置中，取消选中是指正常状态或焦点在位置外。

`CheckedAlign` 属性确定圆圈与标签文本的相对位置，它可以在标签的顶部、左右两边或下方。

只要 `Checked` 属性的值改变，就会引发 `CheckedChanged` 事件。这样就可以根据控件的新值执行其他动作。

### 31.3.4 ComboBox 控件、ListBox 控件和 CheckedListBox 控件

`ComboBox`、`ListBox` 和 `CheckedListBox` 都派生于 `ListControl` 类。这个类提供了一些基本的列表管理功能。使用列表控件最重要的事是，给列表添加数据和选择数据。使用哪个列表一般取决于列表的用法和列表中数据的类型。如果需要选择多个选项，或用户需要在任意时刻查看列表中的几个项，最好使用 `ListBox` 和 `CheckedListBox`。如果一次只选择一个选项，就可以使用 `ComboBox`。

在使用列表框之前，必须先添加数据。为此，应给 `ListBox.ObjectCollection` 添加对象。这个集合可以使用列表的 `Items` 属性访问。由于该集合存储了对象，因此可以把任意有效的 .NET 类型添加到列表中。要标识对象，需要设置两个重要的属性。第一个是 `DisplayMember` 属性，这个设置告诉列表控件，在列表中显示对象的哪个属性。另一个是 `ValueMember` 属性，它是要返回值的对象属性。如果在列表中添加了字符串，这两个属性就默认使用字符串值。示例应用程序中的 `frmLists` 窗体显示了如何把对象和字符串(也是对象)加载到列表框中。该例子使用 `Vendor` 对象作为列表数据。`Vendor` 对象只包含两个属性 `Name` 和 `PhoneNo`。`DisplayMember` 属性设置为 `Name`，这告诉列表控件，把列表中 `Name` 属性的值显示给用户。

访问列表控件中的数据有两种方式，如下面的代码示例所示。列表中加载了 `Vendor` 对象，设置了 `DisplayMember` 和 `ValueMember` 属性。这段代码在示例应用程序的 `frmLists` 窗体中。

首先是 `LoadList` 方法，它给列表加载了 `Vendor` 对象或一个包含供应商姓名的简单字符串。选中一个选项按钮，看看在列表中加载了哪些值：

```
private void LoadList(Control ctrlToLoad)
{
    ListBox tmpCtrl = null;

    if (ctrlToLoad is ListBox)
        tmpCtrl = (ListBox)ctrlToLoad;
```

```

tmpCtrl.Items.Clear();
tmpCtrl.DataSource = null;

if (radioButton1.Checked)
{
    //load objects
    tmpCtrl.Items.Add(new Vendor("XYZ Company", "555-555-1234"));
    tmpCtrl.Items.Add(new Vendor("ABC Company", "555-555-2345"));
    tmpCtrl.Items.Add(new Vendor("Other Company", "555-555-3456"));
    tmpCtrl.Items.Add(new Vendor("Another Company", "555-555-4567"));
    tmpCtrl.Items.Add(new Vendor("More Company", "555-555-6789"));
    tmpCtrl.Items.Add(new Vendor("Last Company", "555-555-7890"));
    tmpCtrl.DisplayMember = "Name";
}
else
{
    tmpCtrl.Items.Clear();
    tmpCtrl.Items.Add("XYZ Company");
    tmpCtrl.Items.Add("ABC Company");
    tmpCtrl.Items.Add("Other Company");
    tmpCtrl.Items.Add("Another Company");
    tmpCtrl.Items.Add("More Company");
    tmpCtrl.Items.Add("Last Company");
}
}

```

在列表中加载了数据后，就可以使用 `SelectedItem` 和 `SelectedIndex` 属性获取数据。`SelectedItem` 返回当前选中的对象。如果列表设置为允许选择多个选项，就不能保证返回选中的选项。此时，应使用 `SelectObject` 集合。它包含列表中当前选中的所有选项。

如果需要特定索引的选项，可以使用 `Items` 属性访问 `ListBox.ObjectCollection`。这是一个标准的 .NET 集合类，所以该集合中的项可以用与其他集合类相同的方式访问。

如果使用 `DataBinding` 填充列表，`SelectedValue` 属性就会返回选中对象内设置为 `ValueMember` 属性的属性值。如果 `Phone` 设置为 `ValueMember`，`SelectedValue` 就从选中的项中返回 `Phone` 值。要使用 `ValueMember` 和 `SelectedValue`，列表必须通过 `DataSource` 属性来加载。必须先使用对象加载 `ArrayList` 或其他基于 `ICollection` 的集合，再给列表赋予 `DataSource` 属性。下面的小例子演示了这个方法：

```

listBox1.DataSource = null;
System.Collections.ArrayList lst = new System.Collections.ArrayList();
lst.Add(new Vendor("XYZ Company", "555-555-1234"));
lst.Add(new Vendor("ABC Company", "555-555-2345"));
lst.Add(new Vendor("Other Company", "555-555-3456"));
lst.Add(new Vendor("Another Company", "555-555-4567"));
lst.Add(new Vendor("More Company", "555-555-6789"));
lst.Add(new Vendor("Last Company", "555-555-7890"));
listBox1.Items.Clear();
listBox1.DataSource = lst;
listBox1.DisplayMember = "Name";
listBox1.ValueMember = "Phone";

```

使用 `SelectedValue`，但未使用 `DataBinding`，会导致 `NullException` 错误。

下面的代码显示了访问列表中数据的语法：

```

//obj is set to the selected Vendor object

```

```
obj = listBox1.SelectedItem;

//obj is set to the Vendor object with index of 3 (4th object)
obj = listBox.Items[3];

//obj is set to the values of the Phone property of the selected vendor object
//This example assumes that databinding was used to populate the list
listBox1.ValuesMember = "Phone";
obj = listBox1.SelectedValue;
```

注意，这些方法的返回类型都是 object。要使用 obj 的值，必须把它转换为正确的数据类型。

ComboBox 的 Items 属性返回 ComboBox.ObjectCollection。ComboBox 组合了编辑控件和列表框。把一个 DropDownStyle 枚举值传送给 DropDownStyle 属性，就可以设置 ComboBox 的样式。表 31-2 列出了 DropDownStyle 的各个值。

表 31-2

值	说 明
DropDown	组合框的文本部分是可以编辑的，用户可以输入值。用户必须单击箭头按钮，才能显示列表
DropDownList	文本部分不能编辑。用户必须从列表中选择
Simple	类似于 DropDown，但列表总是可见的

如果列表中的值比较宽，就可以使用 DropDownWidth 属性改变控件下拉部分的宽度。MaxDropDownItems 属性设置在显示列表的下拉部分时的最大项数。

FindString 和 FindStringExact 方法是列表控件的另外两个有用的方法。FindString 在列表中查找以传入字符串开头的第一个字符串。FindStringExact 查找与传入字符串匹配的第一个字符串。它们都返回找到的值的索引，如果没有找到，就返回 -1。它们还可以将要搜索的起始索引整数作为参数。

31.3.5 DateTimePicker 控件

DateTimePicker 允许用户在许多不同的格式中选择一个日期或时间值(或两者)。可以以任何标准时间日期格式显示基于 DateTime 的值。Format 属性使用 DateTimePickerFormat 枚举，它可以把格式设置为 Long、Short、Time 或 Custom。如果 Format 属性设置为 DateTimePicker.Format.Custom，就可以把 CustomFormat 属性设置为表示格式的字符串。

DateTimePicker 还包含 Text 属性和 Value 属性。Text 属性返回 DateTime 值的文本表示，Value 属性返回 DateTime 对象。还可以用 MinDate 和 Maxdate 属性设置日期所允许的最大值和最小值。

在用户单击向下箭头时，会显示一个日历，允许用户选择日历中的一个日期。DateTimePicker 还包含一些属性，这些属性可以设置标题、月份背景色和前景色，改变日期的外观。

ShowUpDown 属性确定控件上是否显示 UpDown 箭头。单击向上或向下箭头就可以改变当前突出显示的值。



## 31.3.6 ErrorProvider 组件

**ErrorProvider** 实际上并不是一个控件，而是一个组件。当把该组件拖放到设计器上时，它会显示在设计器下方的组件栏中。当存在一个错误条件或验证失败时，**ErrorProvider** 可以在控件的旁边显示一个图标。假定有一个 **TextBox** 控件用于输入年龄。业务规则是年龄值不能大于 65。如果用户试图输入大于 65 的年龄，就必须通知用户该年龄大于所允许的值，需要改变输入的值。有效值的检查在文本框的 **Validated** 事件中进行。如果验证失败，就调用 **SetError** 方法，传送引起错误的控件和一个字符串，将该错误告知用户。然后，一个图标开始闪烁，表示出现了一个错误，用户把鼠标放在该图标上时，会显示错误文本。图 31-3 显示了文本框中输入无效值时出现的图标。

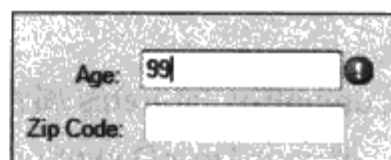


图 31-3

可以为窗体上产生错误的每个控件创建一个 **ErrorProvider**，但如果窗体上有很多控件，这就会很麻烦。另一种选择是使用一个 **ErrorProvider**，在验证事件中调用被验证控件的 **IconLocation** 方法和一个 **ErrorIconAlignment** 枚举值。这个值设置图标与哪个控件对齐。接着调用 **SetError** 方法。如果没有错误，就调用 **SetError** 方法，并把空字符串作为它的错误字符串，清除 **ErrorProvider**。下面的示例说明了其工作原理：

```
private void txtAge_Validating(object sender,
                             System.ComponentModel.CancelEventArgs e)
{
    if (txtAge.Text.Length > 0 && Convert.ToInt32(txtAge.Text) > 65)
    {
        errMain.SetIconAlignment((Control)sender, ErrorIconAlignment.MiddleRight);
        errMain.SetError((Control)sender, "Value must be less than 65.");
        e.Cancel = true;
    }
    else
    {
        errMain.SetError((Control)sender, "");
    }
}

private void txtZipCode_Validating(object sender,
                                   System.ComponentModel.CancelEventArgs e)
{
    if (txtZipCode.Text.Length != 5)
    {
        errMain.SetIconAlignment((Control)sender, ErrorIconAlignment.MiddleRight);
        errMain.SetError((Control)sender, "Must be 5 characters.");
        e.Cancel = true;
    }
    else
    {
        errMain.SetError((Control)sender, "");
    }
}
```

如果验证失败(例如 **txtAge** 中的年龄超过 65)，就调用 **ErrorProvider** 组件 **errMain** 的 **SetIcon** 方法。该方法把图标放在验证失败的控件旁边。接着设置错误，这样，在用户把鼠标放在该图标上时，就会显示一个消息，告知用户哪个控件应为验证失败负责。

### 31.3.7 HelpProvider 组件

HelpProvider 类似于 ErrorProvider，也是一个组件，而不是控件。HelpProvider 允许挂起控件，显示帮助主题。要把控件与 HelpProvider 关联起来，应调用 SetShowHelp 方法，给该方法传送该控件和一个确定是否显示帮助的布尔值。HelpNamespace 属性允许设置帮助文件。在设置 HelpNamespace 属性时，只要按下 F1，就会显示帮助文件，用 HelpProvider 注册的控件还会获得焦点。可以用 SetHelpKeyword 方法为帮助文件设置一个关键字。SetHelpNavigator 带一个 HelpNavigator 枚举值，用于确定显示帮助文件中的哪个元素。可以把它设置为特定的主题、索引、目录表或搜索页面。SetHelpString 把与帮助相关的文本字符串值关联到控件上。如果没有设置 HelpNamespace 属性，按下 F1 就会在弹出窗口中显示这个文本。下面在上一个示例中添加一个 HelpProvider：

```
helpProvider1.SetHelpString(txtAge, "Enter an age that is less than 65");  
helpProvider1.SetHelpString(txtZipCode, "Enter a 5 digit zip code");
```

### 31.3.8 ImageList 组件

ImageList 组件就是一个图像列表。一般情况下，这个属性用于存储一个图像集合，这些图像用作工具栏图标或 TreeView 控件上的图标。许多控件都包含 ImageList 属性。这个属性一般和 ImageIndex 属性一起使用。ImageList 属性设置为 ImageList 组件的一个实例，ImageIndex 属性设置为 ImageList 中应在控件中显示的图像的索引。使用 ImageIndex.Images 属性的 Add 方法可以把图像添加到 ImageList 组件中。Images 属性返回一个 ImageCollection。

两个最常用的属性是 ImageSize 和 ColorDepth。ImageSize 使用 Size 结构作为其值。其默认值是 16×16，但可以取 1~256 之间的任意值。ColorDepth 使用 ColorDepth 枚举作为其值。颜色深度值可以从 4 位~32 位。在 .NET Framework 1.1 中，默认是 ColorDepth.Depth8Bit。

### 31.3.9 Label 控件

Label 控件一般用于给用户提供描述文本。文本可以与其他控件或当前系统状态相关。通常标签和文本框一起使用。标签为用户提供了在文本框中输入的数据类型的描述。标签控件总是只读的，用户不能修改 Text 属性的字符串值。但是，可以在代码中修改 Text 属性。UseMnemonic 属性允许用户启用访问键功能。在 Text 属性中，给一个字符前面加上宏符号&时，标签控件中的该字母就会加上下划线。按下 Alt 键和带有下划线的字母就会把焦点移动到 tab 顺序的下一个控件上。如果 Text 属性的文本包含一个宏符号，就应添加第二个宏符号，其后的字母将不带下划线。例如，如果标签文本是 Nuts & Bolts，就应把属性设置为 Nuts && Bolts。由于标签控件是只读的，所以不能获得焦点。这就是焦点会移动到下一个控件上的原因。因此要记住，如果启用 Mnemonic，就必须正确设置窗体上的 tab 顺序。

AutoSize 属性是一个布尔值，指定标签是否根据标签的内容自动设置其大小。在多语言应用程序中，Text 属性的长度会根据当前语言的不同而变化，此时就可以使用这个属性。



## 31.3.10 ListView 控件

ListView 控件允许以 4 种不同的方式显示条目。可以显示文本和可选的大图标、显示文本和可选的小图标、在垂直列表中显示文本和小图标、以及在详细视图中显示条目文本，并在列中显示子条目。这听起来应很熟悉，因为文件管理器的右边就用这种方式显示文件夹的内容。ListView 包含一个 ListViewItem 集合。ListView 允许设置一个用于显示的 Text 属性，它的另一个属性 SubItems 包含在详细视图中显示的文本。

下面的示例演示了如何使用 ListView。这个示例包含国家的短列表。每个 CountryList 对象都包含国家名、国家简称和货币属性。下面是 CountryList 类的代码：

```
using System;

namespace FormSample
{
    public class CountryItem : System.Windows.Forms.ListViewItem
    {
        string _cntryName = "";
        string _cntryAbbrev = "";

        public CountryItem(string countryName,
                           string countryAbbreviation, string currency)
        {
            _cntryName = countryName;
            _cntryAbbrev = countryAbbreviation;
            base.Text = _cntryName;
            base.SubItems.Add(currency);
        }

        public string CountryName
        {
            get { return _cntryName; }
        }

        public string CountryAbbreviation
        {
            get { return _cntryAbbrev; }
        }
    }
}
```

注意 CountryList 类派生自 ListViewItem。这是因为我们只能给 ListView 控件添加基于 ListViewItem 的对象。在构造函数中，给 base.Text 属性传送国家名，给 base.SubItems 属性添加货币值。这会在列表中显示国家名，在 Details 视图的另一个单独的列中显示货币。

接着，需要在窗体的代码中给 ListView 控件添加几个 CountryItem 对象：

```
lvCountries.Items.Add(new CountryItem("United States", "US", "Dollar"));
lvCountries.Items[0].ImageIndex = 0;
lvCountries.Items.Add(new CountryItem("Great Britain", "GB", "Pound"));
lvCountries.Items[1].ImageIndex = 1;
lvCountries.Items.Add(new CountryItem("Canada", "CA", "Dollar"));
lvCountries.Items[2].ImageIndex = 2;
lvCountries.Items.Add(new CountryItem("Japan", "JP", "Yen"));
lvCountries.Items[3].ImageIndex = 3;
lvCountries.Items.Add(new CountryItem("Germany", "GM", "Deutch Mark"));
lvCountries.Items[4].ImageIndex = 4;
```

这里给 ListView 控件 lvCountries 的 Items 集合添加了一个新的 CountryItem。注意在把 CountryItem 添加到控件中后，才设置 CountryItem 的 ImageIndex 属性。有两个 ImageIndex 属性，一个用于大图标，一个用于小图标(SmallImageList 和 LargeImageList 属性)。要给两个 ImageList 指定不同的图像大小，应确保以相同的顺序给 ImageList 添加条目。这样，每个 ImageList 的索引就表示仅尺寸不同的相同图像。在本例中，ImageList 包含我们添加的每个国家的标记图标。

窗体的顶部是一个组合框 cbView，它列出了 4 个不同 View 枚举值。把这些条目添加到 cbView 中，如下所示：

```
cbView.Items.Add(View.LargeIcon);
cbView.Items.Add(View.SmallIcon);
cbView.Items.Add(View.List);
cbView.Items.Add(View.Details);
cbView.SelectedIndex = 0;
```

在 cbView 的 SelectedIndexChanged 事件中，添加下面这行代码：

```
lvCountries.View = (View) cbView.SelectedItem;
```

这行代码把 lvCountries 的 View 属性设置为在 ComboBox 控件中选中的新值。注意需要把它转换 View 类型，因为从 cbView 的 SelectedItem 属性中返回的是 object 类型。

最后，必须给 Column 集合添加列。这些列在 Details 视图中显示。在本例中添加了两列 Country Name 和 Currency。列的顺序是 ListViewItem 的 Text，然后是 ListViewItem.SubItems 集合中的每一项，按照它们在集合中的顺序显示。添加列时，需要创建一个 ColumnHeader 对象，设置 Text 属性，还可以设置 Width 和 Alignment 属性。在创建 ColumnHeader 对象后，就可以把它添加到 Column 属性中。添加列的另一种方法是使用 Columns.Add 方法的重写版本，它允许传送 Text、Width 和 Alignment 值。下面是一个示例：

```
lvCountries.Column.Add("Country", 100, HorizontalAlignment.Left);
lvCountries.Column.Add("Currency", 100, HorizontalAlignment.Left);
```

如果把 AllowColumnReorder 属性设置为 true，用户就可以拖动列标题，重新安排列的顺序。

ListView 上的 CheckBoxes 属性在 ListView 的条目旁边显示复选框，允许用户在 ListView 控件中选择多个条目。使用 CheckedItems 集合可以检查哪些项目被选中。

Alignment 属性设置 Large 和 Small 图标视图中图标的对齐方式。该值可以是 ListViewAlignment 枚举中的任意值，即 Default、Left、Top、SnapToGrid。Default 值允许用户把图标放在任意位置。在选择 Left 或 Top 时，条目应与 ListView 控件的左边或顶边对齐。在选择 SnapToGrid 时，条目会捕捉到 ListView 控件上不可见的栅格。AutoArrange 属性可以设置为布尔值，它会根据 Alignment 属性自动对齐图标。

### 31.3.11 PictureBox 控件

PictureBox 控件用于显示图像。图像可以是 BMP、JPEG、GIF、PNG、元文件或图标。SizeMode 属性使用 PictureBoxSizeMode 枚举确定图像在控件中的大小和位置。SizeMode 属性可以是 AutoSize、CenterImage、Normal 和 StretchImage。

设置 `ClientSize` 属性, 可以改变 `PictureBox` 的显示区域大小。要加载 `PictureBox`, 首先创建一个基于 `Image` 的对象。例如, 要把 JPEG 文件加载到 `PictureBox` 中, 需要编写如下代码:

```
Bitmap myJpeg = new Bitmap("mypic.jpg");  
pictureBox1.Image = (Image) myJpeg;
```

注意需要转换回 `Image` 类型, 因为这是 `Image` 属性所要求的。

### 31.3.12 ProgressBar 控件

`ProgressBar` 控件是较长操作的状态的可视化表示。它指示用户正在进行某个操作, 用户应等待。`ProgressBar` 控件工作时要设置 `Minimum` 和 `Maximum` 属性。这些属性对应于进度指示器的最左端(`Minimum`)和最右端(`Maximum`)。设置 `Step` 属性, 以确定每次调用 `PerformStep` 方法时数值的增量。还可以使用 `Increment` 方法, 递增在方法调用中传入的值。`Value` 属性返回 `ProgressBar` 的当前值。

可以使用 `Text` 属性通知用户已完成了操作的百分数或还未处理的条目数。还有一个 `BackgroundImage` 属性可以定制进度条的外观。

### 31.3.13 TextBox 控件、RichTextBox 控件与 MaskedTextBox 控件

`TextBox` 控件是工具箱中最常用的控件之一。`TextBox`、`RichTextBox` 和 `MaskedTextBox` 控件都派生于 `TextBoxBase`。`TextBoxBase` 提供了 `MultiLine` 和 `Lines` 属性, `MultiLine` 属性是一个布尔值, 允许 `TextBox` 控件在多行中显示文本。文本框中的每一行都是字符串数组的一部分。这个数组通过 `Lines` 属性来访问。`Text` 属性把整个文本框内容返回为一个字符串。`TextLength` 是返回的文本字符串的总长。`MaxLength` 属性把文本的长度限制为指定的数字。

`SelectedText`、`SelectionLength` 和 `SelectionStart` 都处理文本框中当前选中的文本。选中的文本是控件获得焦点时突出显示的文本。

`TextBox` 控件增加了几个有趣的属性。`AcceptsReturn` 属性是一个布尔值, 允许 `TextBox` 把回车键接受为一个换行符, 或者激活窗体上的默认按钮。这个属性设置为 `true` 时, 按下回车键会在文本框中创建一个新行。`CharactorCasing` 确定文本框中文本的大小写。`CharactorCasing` 枚举包含 3 个值 `Lower`、`Normal` 和 `Upper`。`Lower` 会使所有的文本小写, `Upper` 则把所有的文本转变为大写, `Normal` 把文本显示为输入时的形式。`PasswordChar` 属性用一个字符表示用户在文本框中输入文本时要显示给用户的内容, 这通常用于输入密码和 PIN。`text` 属性返回输入的文本, 只有显示的内容会受这个属性的影响。

`RichTextBox` 是一个文本编辑控件, 它可以处理特殊格式的文本。顾名思义, `RichTextBox` 控件使用 `Rich Text Format(RTF)` 处理特殊的格式。使用 `Selection` 属性 `SelectionFont`、`SelectionColor`、`SelectionBullet` 可以修改格式, 使用 `SelectionIndent`、`SelectionRightIndent`、`SelectionHangingIndent` 可以修改段落的格式。所有 `Selection` 属性的工作方式都相同。如果有一个突出显示的文本段, 对 `Selection` 属性的修改就会影响选中的文本。如果没有选中文本, 这些修改就对当前插入点后面的文本起作用。

控件的文本可以使用 `Text` 属性或 `Rtf` 属性提取。`Text` 属性只返回控件的文本, 而 `Rtf` 属性

返回带格式的文本。

LoadFile 方法可以用两种方式从文件中加载文本。它可以使用一个表示文件名和路径的字符串，也可以使用一个流对象。还可以指定 RichTextBoxStreamType。表 31-3 列出了 RichTextBoxStreamType 的值。

表 31-3

值	说 明
PlainText	没有格式信息，包含 OLE 对象，允许使用空格
RichNoOleObjs	Rich 文本格式，但不包含 OLE 对象已经包含的空格
RichText	格式化的 RTF，且包含 OLE 对象
TextTextOleObjs	无格式文本，用文本替换 OLE 对象
UnicodePlainText	与 PlainText 相同，但编码为 Unicode

SaveFile 方法使用相同的参数，把控件中的数据存储在指定的文件中。如果文件已经存在，就覆盖它。

MaskedTextBox 可以限制用户在控件中输入的内容，它还可以自动格式化输入的数据。使用几个属性可以验证或格式化用户的输入。Mask 属性包含覆盖字符串，覆盖字符串类似于格式字符串，使用 Mask 字符串可以设置允许的字符数、允许字符的数据类型和数据的格式。基于 MaskedTextProvider 的类也提供了需要的格式化和验证信息。MaskedTextProvider 只能在它的构造函数中设置。

有 3 个不同的属性返回 MaskedTextControl 的文本。Text 属性返回控件的当前文本，它可以根据控件是否获得焦点而不同，而控件是否获得焦点取决于 HidePromptOnLeave 属性的值。该属性是一个字符串，告诉用户应输入什么内容。InputText 属性总是只返回用户输入的文本。OutputText 属性返回根据 IncludeLiterals 和 IncludePrompt 属性格式化的文本。例如，如果对电话号码进行覆盖，Mask 字符串就应包含括号和几个短横线。这些都是字面量字符，如果 IncludeLiteral 属性设置为 true，括号和短横线就应包含在 OutputText 属性中。

MaskedTextBox 控件还有几个额外的事件。OutputTextChanged 和 InputTextChanged 在 InputText 或 OutputText 改变时触发。

31.3.14 Panel 控件

Panel 控件就是包含其他控件的控件。把控件组合在一起，放在一个面板上，将更容易管理这些控件。例如，可以禁用面板，从而禁用该面板上的所有控件。Panel 控件派生于 ScrollableControl，所以还可以使用 AutoScroll 属性。如果可用区域上有过多的控件要显示，就可以把它们放在一个面板上，并把 AutoScroll 属性设置为 true，这样就可以滚动所有的控件了。

面板在默认情况下不显示边框，但把 BorderStyle 属性设置为不是 none 的其他值，就可以使用面板通过边框可视化地组合相关的控件。这会使用户界面更友好。

Panel 是 FlowLayoutPanel、TableLayoutPanel、TabPage 和 SplitterPanel 的基类。使用这些控件，可以创建非常复杂或专业化的窗体或窗口。FlowLayoutPanel 和 TableLayoutPanel 对创建



正确设置大小的窗体尤其有帮助。

### 31.3.15 FlowLayoutPanel 和 TableLayoutPanel 控件

FlowLayoutPanel 和 TableLayoutPanel 是 .NET Framework 的新增控件。顾名思义, 面板可以采用 Web 窗体的方式给 Windows 窗体布局。FlowLayoutPanel 是一个容器, 允许以垂直或水平的方式放置包含的控件。除了放置控件之外, 还可以剪辑控件。放置的方向使用 FlowDirection 属性和 FlowDirection 枚举来设置。WrapContents 属性确定在重新设置窗体的大小时, 控件是放在下一行、下一列, 还是剪辑控件。

TableLayoutPanel 使用栅格结构控制控件的布局。所有的 Windows 窗体控件都是 TableLayoutPanel 的子控件, 包括另一个 TableLayoutPanel。所以窗口的布局可以非常灵活, 并可以动态设置。把一个控件添加到 TableLayoutPanel 上时, 会给属性页面的 Layout 类别添加 4 个属性 Column、ColumnSpan、Row 和 RowSpan。与 Web 页面上的 html 表一样, 可以给每个控件设置列和行间距。控件默认放置在表的单元格中心, 但这可以使用 Anchor 和 Dock 属性改变。

行和列的默认样式可以使用 RowStyles 和 ColumnsStyles 集合改变。这两个集合分别包含 RowStyle 和 ColumnsStyle 对象。Style 对象有一个公共属性 SizeType。SizeType 使用 SizeType 枚举来确定列宽或行高。该枚举值包含 AutoSize、Absolute 和 Percent。AutoSize 与其他同等控件共享该空间。Absolute 允许使用一组像素值来设置大小。Percent 要求控件把列或宽度设置为父控件的一个百分数。

行、列和子控件都可以在运行期间添加和删除。GrowStyle 属性使用 TableLayoutPanelGrowStyle 枚举值来设置在已填满的表中添加一个新控件时, 是给表添加列、行, 还是使表保持固定的大小。如果其值是 FixedSized, 则在试图添加另一个控件时, 就抛出一个 ArgumentException 异常。如果表中的单元格为空, 控件就放在空单元格中。这个属性仅在表格已满, 但要添加控件时起作用。

示例应用程序中的 frmPanel 窗体有 FlowLayoutPanels 和 TableLayoutPanels, 并在其中放置了各种控件。试验这些控件, 尤其要试用布局面板中控件的 Dock 和 Anchor 属性, 这是理解其工作方式的最佳途径。

### 31.3.16 SplitContainer 控件

SplitContainer 控件把 3 个控件组合在一起, 其中有两个面板控件, 在它们之间有一个分隔栏。用户可以移动分隔栏, 重新设置面板的大小。在重新设置面板的大小时, 面板上的控件也可以重新设置大小。SplitContainer 的最佳示例是文件管理器。左面板包含文件的树型视图, 右面板包含文件夹内容的列表视图。用户在分隔栏上移动鼠标时, 光标就会改变, 此时可以移动分隔栏。SplitContainer 可以包含任意控件, 包括布局面板和其他 SplitContainer。因此, 可以创建非常复杂、专业化很高的窗体。

分隔栏的移动和定位可以用 SplitterDistance 和 SplitterIncrement 属性控制。SplitterDistance



属性确定分隔栏与控件左边界或顶边的距离, `SplitterIncrement` 确定在拖动时分隔栏移动的像素值。面板可以使用 `Panel1MinSize` 和 `Panel2MinSize` 属性设置其最小尺寸, 这些属性的单位也是像素。

`Splitter` 控件会引发与移动相关的两个事件 `SplitterMoving` 和 `SplitterMoved`。`SplitterMoving` 事件在移动过程中引发, `SplitterMoved` 在移动结束后引发。它们都接收一个 `SplitterEventArgs`。`SplitterEventArgs` 的 `SplitX` 和 `SplitY` 属性表示 `Splitter` 左上角的 X 和 Y 坐标, X 和 Y 属性表示鼠标指针的 X 和 Y 坐标的。

### 31.3.17 TabControl 控件和 TabPages 控件

`TabControl` 允许把相关的组件组合到一系列选项卡页面上。`TabControl` 管理 `TabPage` 集合。有几个属性可以控制 `TabControl` 的外观。`Appearance` 属性使用 `TabAppearance` 枚举确定选项卡的外观。其值是 `FlatButtons`、`Buttons` 或 `Normal`。`Multiline` 属性的值是一个布尔值, 确定是否显示多行选项卡。如果 `Multiline` 属性设置为 `false`, 而有多多个选项卡不能一次显示出来, 就提供一组箭头, 允许用户滚动查看剩余的选项卡。

`TabPage` 的 `text` 属性是在选项卡上显示的内容。`Text` 属性也在重写的构造函数中用作参数。

一旦创建了 `TabPage` 控件, 它基本上就是一个容器控件, 用于放置其他控件。`Visual Studio 2008` 中的设计器使用集合编辑器, 很容易给 `TabControl` 控件添加 `TabPage` 控件。在添加每个页面时都可以设置各种属性。接着把其他子控件拖放到每个 `TabPage` 控件上。

通过查看 `SelectedTab` 属性可以确定当前的选项卡。每次选择新选项卡时, 都会引发 `SelectedIndex` 事件。通过监听 `SelectedIndex` 属性, 再用 `SelectedTab` 属性确认当前选项卡, 就可以对每个选项卡进行特定的处理。

### 31.3.18 ToolStrip 控件

`ToolStrip` 控件是一个用于创建工具栏、菜单结构和状态栏的容器控件。`ToolStrip` 直接用于工具栏, 还可以用作 `MenuStrip` 和 `StatusStrip` 控件的基类。

`ToolStrip` 控件在用于工具栏时, 使用一组基于抽象类 `ToolStripItem` 的控件。`ToolStripItem` 可以添加公共显示和布局功能, 并管理控件使用的大多数事件。`ToolStripItem` 派生于 `System.ComponentModel.Component` 类, 而不是 `Control` 类。基于 `ToolStripItem` 的类必须包含在基于 `ToolStrip` 的容器中。

`Image` 和 `Text` 是要设置的最常见属性。`Image` 可以用 `Image` 属性设置, 也可以使用 `ImageList` 控件, 把它设置为 `ToolStrip` 控件的 `ImageList` 属性。然后就可以设置各个控件的 `ImageIndex` 属性。

`ToolStripItem` 上文本的格式化用 `Font`、`TextAlign` 和 `TextDirection` 属性来处理。`TextAlign` 设置文本与控件的对齐方式, 它可以是 `ControlAlignment` 枚举中的任一值, 默认为 `MiddleRight`。`TextDirection` 属性设置文本的方向, 其值可以是 `ToolStripTextDirection` 枚举中的任一值, 包括 `Horizontal`、`Inherit`、`Vertical270` 和 `Vertical90`。`Vertical270` 把文本旋转 270°, `Vertical90` 把文本旋转 90°。

`DisplayStyle` 属性控制在控件上是显示文本、图像、文本和图像, 还是什么都不显示。在

AutoSize 设置为 true 时, ToolStripItem 会重新设置其大小, 确保只使用最少量的空间。  
直接派生于 ToolStripItem 的控件如表 31-4 所示。

表 31-4

Tool Strip Items	说 明
ToolStripButton	表示用户可以选择的按钮
ToolStripLabel	在 ToolStrip 上显示不能选择的文本或图像。ToolStripLabel 还可以显示一个或多个超链接
ToolStripSeparator	用于分解和组合其他 ToolStripItems。选项根据功能来组合
ToolStripDropDownItem	显示下拉选项。是 ToolStripDropDownButton、ToolStripMenuItem 和 ToolStripSplitButton 的基类
ToolStripControlHost	在 ToolStrip 上存放其他非 ToolStripItem 的派生控件。是 ToolStripComboBox、ToolStripProgressBar 和 ToolStripTextBox 的基类

表 31-4 中的前两项 ToolStripDropDownItem 和 ToolStripControlHost 需要详细探讨。  
ToolStripDropDownItem 是 ToolStripMenuItems 的基类, 用于建立菜单结构。ToolStripMenuItems 添加到 MenuStrip 控件上。如前所述, MenuStrips 派生于 ToolStrip 控件。在处理和扩展菜单项时, 这是很重要的。因为工具栏和菜单派生于同一个类, 所以很容易创建管理并执行命令的框架。

ToolStripControlHost 可以包含其他不派生自 ToolStripItem 的控件。可以直接放在 ToolStrip 中的控件是派生自 ToolStripItem 的控件。下面的示例演示了如何在 ToolStrip 中放置 DateTimePicker 控件:

```
public mdiParent()
{
    InitializeComponent();

    ToolStripControlHost _dateTimeCtl;
    _dateTimeCtl = new ToolStripControlHost(new DateTimePicker());
    ((DateTimePicker)_dateTimeCtl.Control).ValueChanged +=
        delegate {
            toolStripLabel1.Text =
            ((DateTimePicker)_dateTimeCtl.Control).Value.Subtract(DateTime.Now).ToString();
        };

    _dateTimeCtl.Width = 200;
    _dateTimeCtl.DisplayStyle = ToolStripItemDisplayStyle.Text;
    toolStrip1.Items.Add(_dateTimeCtl);
}
```

这是示例代码中 frmMain 窗体的构造函数。首先, 声明并实例化一个 ToolStripControl Host。注意在实例化该控件时, 应把窗体要包含的控件传送给构造函数。下一行代码启动 DateTimePicker 控件的 ValueChanged 事件。这个控件可以通过 ToolStripHostControl 的 Control 属性访问, 并返回一个 Control 对象, 因此需要将其类型转换为正确的类型。之后, 就可以使用窗体包含的控件的属性和方法了。

提供更好封装性能的另一种方法是创建一个派生自 ToolStripControlHost 的新类。下面的代

码是 DateTimePicker 工具栏控件的另一个版本 ToolStripDateTimePicker:

```
namespace FormsSample.SampleControls
{
    public class ToolStripDateTimePicker: System.Windows.Forms.ToolStripControlHost
    {
        //need to declare the event that will be exposed
        public event EventHandler ValueChanged;

        public ToolStripDateTimePicker () : base(new DateTimePicker())
        {
        }
        //create strong typed Control property.
        public new DateTimePicker Control
        {
            get{return (DateTimePicker)base.Control;}
        }
        //create a strong typed Value property
        public DateTime Value
        {
            get { return Control.Value; }
        }

        protected override void OnSubscribeControlEvents(Control control)
        {
            base.OnSubscribeControlEvents(control);
            ((DateTimePicker)control).ValueChanged +=
                new EventHandler(HandleValueChanged);
        }

        protected override void OnUnsubscribeControlEvents(Control control)
        {
            base.OnSubscribeControlEvents(control);
            ((DateTimePicker)control).ValueChanged -=
                new EventHandler(HandleValueChanged);
        }

        private void HandleValueChanged (object sender, EventArgs e)
        {
            if (ValueChanged != null)
                ValueChanged(this, e);
        }
    }
}
```

这个类的主要工作是提供 DateTimePicker 选中的属性、方法和事件。这样，主机应用程序就不必维护底层控件的引用了。提供事件的过程有点复杂。OnSubscribeControlEvents 方法用于为基于 ToolStripControlHost 的类(本例是 ToolStripDateTimePicker)同步被包含控件(在这里是 DateTimePicker)的事件。在这个例子中，ValueChanged 事件传送给 ToolStripDateTimePicker。其作用是允许控件的用户在主机应用程序中建立事件，就好像 ToolStripDateTimePicker 派生于 DateTimePicker，而不是 ToolStripControlHost 一样。下面的示例代码演示了这一点，它使用了 ToolStripDateTimePicker:

```
public mdiParent()
{
    ToolStripDateTimePicker otherDateTimePicker = new ToolStripDateTimePicker ();
    otherDateTimePicker.Width = 200;
```

```
otherDateTimePicker.ValueChanged +=  
    new EventHandler(otherDateTimePicker_ValueChanged);  
toolStrip1.Items.Add(otherDateTimePicker);  
}
```

注意，在建立 ValueChanged 事件处理程序时，要引用 ToolStripDateTimePicker 类，而不是像前面的示例那样引用 DateTimePicker 控件。这个示例的代码与第一个例子相比简洁了许多，不仅如此，由于 DateTimePicker 封装在另一个类中，因此封装性能大大提高了，ToolStripDateTimePicker 在应用程序的其他部分或其他项目中的使用也简单了许多。

### 31.3.19 ToolStrip 控件

MenuStrip 控件是应用程序菜单结构的容器。如前所述，MenuStrip 派生于 ToolStrip 类。在建立菜单系统时，要给 MenuStrip 添加 ToolStripMenu 对象。这可以在代码中完成，也可以在 Visual Studio 的设计器中进行。把一个 MenuStrip 控件拖放到设计器的一个窗体中，MenuStrip 就允许直接在菜单项上输入菜单文本。

MenuStrip 控件只有两个额外的属性。GripStyle 使用 ToolStripGripStyle 枚举把栅格设置为可见或隐藏。MdiWindowListItem 属性提取或返回 ToolStripMenuItem。这个 ToolStripMenuItem 是在 MDI 应用程序中显示所有已打开窗口的菜单。

### 31.3.20 ContextMenuStrip 控件

要显示弹出菜单，或在用户右击鼠标时显示一个菜单，就应使用 ContextMenuStrip 类。与 MenuStrip 一样，ContextMenuStrip 也是 ToolStripMenuItems 对象的容器，但它派生于 ToolStripDropDownMenu。ContextMenu 的创建与 MenuStrip 相同，也是添加 ToolStripMenuItems，定义每一项的 Click 事件，执行某个任务。弹出菜单应赋予特定的控件，为此，要设置控件的 ContextMenuStrip 属性。在用户右击该控件时，就显示该菜单。

### 31.3.21 ToolStripMenuItem 控件

ToolStripMenuItem 是建立菜单结构的类。每个 ToolStripMenuItem 对象都表示菜单系统上的一个菜单选项。每个 ToolStripMenuItem 都有一个包含子菜单的 ToolStripItem Collection。这个功能继承自 ToolStripDropDownItem。

由于 ToolStripMenuItem 派生于 ToolStripItem，因此可以使用所有的格式化属性。图像在菜单文本的右边显示为小图标。菜单项的旁边可以有复选框标记，用 Checked 和 CheckState 属性设置该标记。

还可以给每个菜单项指定快捷键。快捷键一般包含两个按键，如 Ctrl+C (Copy 的快捷键)。在指定快捷键时，把 ShowShortcutKey 属性设置为 true，还可以在菜单上显示该快捷键。

在用户单击菜单项或使用定义好的快捷键时，应执行某个任务。为此，最常见的方式是处理 Click 事件。如果使用了 Checked 属性，还可以使用 CheckStateChanged 和 CheckedChanged 事件确定选中状态的变化。



### 31.3.22 ToolStripManager 类

菜单和工具栏结构可以很大，这样就难以管理。ToolStripManager 类可以创建较小、易于管理的菜单或工具栏结构，并在需要时合并它们。例如，如果窗体上有几个不同的控件，每个控件都必须显示一个弹出菜单。所有的控件都要使用几个菜单项，但每个控件还有几个独特的菜单项。可以在一个 ContextMenuStrip 上定义共有菜单选项，独特的菜单项可以预先定义，或在运行期间创建。对于需要弹出菜单的每个控件，应复制共有菜单选项，再使用 ToolStripManager.Merge 方法把独特的菜单选项与公有菜单选项合并起来。最后得到的菜单赋予控件的 ContextMenuStrip 属性。

### 31.3.23 ToolStripContainer 控件

ToolStripContainer 控件用于停放基于 ToolStrip 的控件。添加一个 ToolStripContainer，把 Docked 属性设置为 Fill，就在窗体的两侧边添加了一个 ToolStripPanel，在窗体的中间添加了一个 ToolStripContainerPanel。在 ToolStripPanels 中可以添加任意 ToolStrip (ToolStrip、MenuStrip 或 StatusStrip)。用户可以选择 ToolStrips，把它拖放到窗体的两边或底部。把 ToolStripPanels 的 Visible 属性设置为 false，就不能把 ToolStrip 放在面板上了。窗体中心的 ToolStripContainerPanel 可以用于放置窗体需要的其他控件。

## 31.4 窗体

本章的前面讨论了如何创建简单的 Windows 应用程序。该示例包含一个派生于 System.Windows.Forms.Form 的类。根据 .NET Framework 说明文档，“窗体是应用程序中窗口的表示方式。”如果您具有 Visual Basic 背景，就会很熟悉术语“窗体”。如果您是使用 MFC 的 C++ 程序员，就可能习惯把窗体称为窗口、对话框或框架。无论怎样，窗体都是与用户交互的基本方式。我们已经介绍了 Control 类中一些较常见的属性、方法和事件，而且 Form 类派生于 Control 类，所以在 Form 类中存在这些属性、方法和事件。Form 类还在 Control 类的基础上添加了大量的功能，本节就介绍这些功能。

### 31.4.1 Form 类

Windows 客户应用程序可以包含一个窗体或上百个窗体。它们可以是基于 SDI(单文档界面，Single Document Interface)或 MDI(多文档界面，Multiple Document Interface)的应用程序。但无论怎样，System.Windows.Forms.Form 类都是 Windows 客户应用程序的核心。Form 类派生于 ContainerControl，ContainerControl 又派生于 ScrollableControl，ScrollableControl 则派生于 Control 类。因此可以假定，窗体可以是其他控件的容器，当所包含的控件在客户区域中显示不下时可以滚动显示，窗体可以拥有与其他控件相同的属性、方法和事件。所以，Form 类相当复杂。本节就介绍这些功能。



### 1. 窗体的实例化和释放

理解创建窗体的过程是很重要的。我们要完成的工作取决于编写初始化代码的位置。对于实例，事件以如下顺序发生：

- 构造函数
- Load
- Activated
- Closing
- Closed
- Deactivate

前3个事件在初始化过程中发生。根据初始化的类型，可以确定要关联哪个事件。这个类的构造函数在对象的实例化过程中执行。Load事件在对象实例化后，窗体可见之前发生。它与构造函数的区别是窗体的可见性。在引发Load事件时，窗体已存在，但不可见。在构造函数的执行过程中，窗体还不存在，处在实例化过程中。Activated事件在窗体处于可见状态并处于当前状态时发生。

有一种情形会略微改动一下这个事件执行顺序。如果在窗体构造函数执行的过程中，Visible属性设置为true，或调用了Show方法(它把Visible属性设置为true)，就会立即引发Load事件。这也会使窗体可见，并处于当前状态，所以还会引发Activate事件。如果在设置Visible属性后还有代码，就执行这些代码。所以启动事件的执行顺序如下所示：

- 构造函数，执行到Visible = true为止
- Load
- Activate
- 构造函数，执行Visible = true之后的代码

这会产生一些未预料到的结果。最好在构造函数中进行尽可能多的初始化。

关闭窗口时会发生什么情况？Closing事件可以取消处理，它把CancelEventArgs作为一个参数，如果把Cancel属性设置为true，就会取消该事件，窗体仍处于打开状态。Closing事件在窗体关闭时发生，而Closed事件在窗体关闭后发生。这两个事件都允许执行必要的清理工作。注意，Deactivate事件在窗体关闭后发生，这是另一个可能产生难以查找的错误的来源。确保不在Deactivate事件中执行防止窗体被正常垃圾收集的操作。例如，设置对另一个对象的引用会使窗体仍未被释放。

如果调用Application.Exit()方法，且当前有一个或多个窗体处于打开状态，就不会引发Closing和Closed事件。如果打开了正要清理的文件或数据库连接，这就是一个需要考虑的问题，此时应调用Dispose方法，所以另一种更好的方法是把大多数清理代码放在Dispose方法中。

与窗体启动相关的一些属性有StartPosition、ShowInTaskbar和TopMost。StartPosition可以是FormStartPosition枚举中的一个值：

- CenterParent: 窗体位于父窗体的客户区域中心。
- CenterScreen: 窗体位于当前屏幕的中心。
- Manual: 窗体的位置根据Location属性的值来确定。
- WindowsDefaultBounds: 窗体位于默认的Windows位置，使用默认的大小。

- **WindowsDefaultLocation**: 窗体位于默认的 Windows 位置, 但其大小根据 **Size** 属性来定。

**ShowInTaskbar** 属性确定窗体是否应在任务栏上可见。如果窗体是一个子窗体, 且只希望父窗体显示在任务栏上时, 才使用这个属性。**TopMost** 属性指定窗体在应用程序启动时位于最上面, 即使窗体没有立即获得焦点, 也位于最上面。

为了让用户与应用程序交互, 用户必须能看到窗体。利用 **Show** 和 **ShowDialog** 方法就可以实现这一点。**Show** 方法仅使窗体对用户可见。下面的代码演示了如何创建一个窗体, 并把它显示给用户。假定要显示的窗体叫做 **MyFormClass**:

```
MyFormClass myForm = new MyFormClass();
myForm.Show();
```

这是非常简单的。但它的一个缺点是没有给调用代码发送任何通知, 说明 **MyForm** 已处理完, 并退出。有时这并不重要, **Show** 方法工作得很好。如果需要提供某种通知, 使用 **ShowDialog** 方法是一种比较好的选择。

在调用 **Show** 方法后, **Show** 方法后面的代码会立即执行。在调用 **ShowDialog** 方法后, 调用代码被暂停执行, 等到调用 **ShowDialog** 方法的窗体关闭后再继续执行。不仅调用代码被暂停执行, 而且窗体也可以返回一个 **DialogResult** 值。**DialogResult** 枚举是一组标识符, 它们描述了对话框关闭的原因, 包括 **OK**、**Cancel**、**Yes**、**No** 和其他几个标识符。为了让窗体返回一个 **DialogResult** 值, 必须设置窗体的 **DialogResult** 属性, 或者在窗体的一个按钮上设置 **DialogResult** 属性。

例如, 假定应用程序的一部分要求提供客户的电话号码。窗体包含一个输入电话号码的文本框, 和两个按钮 **OK** 和 **Cancel**。如果把 **OK** 按钮的 **DialogResult** 属性设置为 **DialogResult.OK**, 把 **Cancel** 按钮的 **DialogResult** 属性设置为 **DialogResult.Cancel**, 则在选择其中一个按钮时, 窗体就会不可见, 并给调用它的窗体返回相应的 **DialogResult** 值。现在注意窗体没有释放, 只是把 **Visible** 属性设置为 **false**。这是因为仍必须从窗体中获取值。在这个示例中, 我们需要电话号码。在窗体上为电话号码创建一个属性, 这样父窗体就可以获取值, 并调用窗体上的 **Close** 方法了。下面就是子窗体的代码:

```
namespace FormsSample.DialogSample
{
    partial class Phone : Form
    {
        public Phone()
        {
            InitializeComponent();

            btnOK.DialogResult = DialogResult.OK;
            btnCancel.DialogResult = DialogResult.Cancel;
        }

        public string PhoneNumber
        {
            get { return textBox1.Text; }
            set { textBox1.Text = value; }
        }
    }
}
```

首先要注意，不包含处理按钮单击事件的代码。因为设置了每个按钮的 DialogResult 属性，所以在单击 OK 或 Cancel 按钮后，窗体就消失了。添加的唯一属性是 PhoneNumber。下面的代码显示了父窗体中调用 Phone 对话框的方法：

```
Phone frm = new Phone();

frm.ShowDialog();
if (frm.DialogResult == DialogResult.OK)
{
    label1.Text = "Phone number is " + frm.PhoneNumber;
}
else if (frm.DialogResult == DialogResult.Cancel)
{
    label1.Text = "Form was canceled.";
}

frm.Close();
```

这看起来非常简单。创建新的 Phone 对象 frm，在调用 frm.ShowDialog() 方法时，这个方法中的代码会停止执行，等待 Phone 窗体返回。接着检查 Phone 窗体的 DialogResult 属性。由于窗体还未释放，是不可见的，所以仍可以访问公共属性，其中一个公共属性就是 PhoneNumber。一旦获取了需要的数据，就可以调用窗体的 Close 方法。

一切正常，但如果返回的电话号码格式不正确，该怎么办？如果把 ShowDialog 放在循环中，就可以再次调用它，让用户重新输入值。这样就可以得到正确的值，注意，如果用户单击了 Cancel 按钮，还必须处理 DialogResult.Cancel：

```
Phone frm = new Phone();

while (true)
{
    frm.ShowDialog();
    if (frm.DialogResult == DialogResult.OK)
    {
        label1.Text = "Phone number is " + frm.PhoneNumber;
        if (frm.PhoneNumber.Length == 8 | frm.PhoneNumber.Length == 12)
        {
            break;
        }
        else
        {
            MessageBox.Show("Phone number was not formatted correctly.  
Please correct entry.");
        }
    }
    else if (frm.DialogResult == DialogResult.Cancel)
    {
        label1.Text = "Form was canceled.";
        break;
    }
}

frm.Close();
```

如果电话号码的长度没有通过简单的测试，Phone 窗体就会显示出来，让用户更正错误。ShowDialog 框没有创建窗体的新实例，在窗体上输入的文本仍在该窗体上，所以如果必须重新设置窗体，就需要程序员自己完成。

## 2. 外观

用户首先看到的是应用程序的窗体。这应是应用程序中第一个也是最重要的功能。如果应用程序不解决业务问题，则其外观就无关紧要了。这并不是说，窗体和应用程序的整体 GUI 设计不应美观。像颜色组合、字体大小和窗口大小等的设计都可以使应用程序更容易让用户操作和接受。

有时不希望用户访问系统菜单。在单击窗口左上角的图标时，这个菜单就会显示出来。一般情况下，它包含恢复、最小化、最大化和关闭等选项。ControlBox 属性允许设置系统菜单的可见性。还可以用 MaximizeBox 和 MinimizeBox 属性设置最大化和最小化按钮的可见性。如果删除了所有的按钮，再把 Text 属性设置为空字符串(“”), 标题栏就会完全消失。

如果设置了窗体的 Icon 属性，但没有把 ControlBox 属性设置为 false，图标就会显示在窗体的左上角。通常应把 Icon 属性设置为 app.ico，这会使每个窗体的图标都与应用程序的图标相同。

FormBorderStyle 属性用于设置显示在窗体周围的边框类型。它使用 FormBorderStyle 枚举，其值是：

- Fixed3D
- FixedDialog
- FixedSingle
- FixedToolWindow
- None
- Sizable
- SizableToolWindow

大多数值的意义都一目了然，只有两个工具窗口边框除外。无论怎样设置 ShowInTaskBar 属性，工具窗口都不显示在任务栏上。当用户按下 Alt+Tab 时，工具窗口也不会显示在窗口列表中。默认设置是 Sizable。

除非明确要求，否则大多数 GUI 元素的颜色都应设置为系统颜色，而不是特定的颜色。这样，如果一些用户喜欢把所有的按钮设置为紫字绿底，应用程序就会采用这种颜色设置。为了把控件设置为使用特定的系统颜色，必须调用 System.Drawing.Color 类的 FromKnownColor 方法。FromKnownColor 方法将一个 KnownColor 枚举值作为其参数。在该枚举中定义了许多颜色和各種 GUI 元素颜色，例如 Control、ActiveBorder 和 Desktop。例如，如果窗体的背景色应总是匹配 Desktop 颜色，就应使用下面的代码：

```
myForm.BackColor = Color.FromKnownColor(KnownColor.Desktop);
```

如果用户改变桌面的颜色，窗体的背景色也会随之改变。这将给应用程序增加友好性。用户可以为桌面选择某种奇怪的颜色组合，但这由他们的偏好决定。

Windows XP 引入了一个叫做可视化样式的特性。当鼠标指针停在按钮、文本框、菜单和其他控件上或单击它们时，这些控件的外观会改变，并做出响应。调用 Application.EnableVisualStyles 方法，可以激活应用程序的可视化样式。这个方法必须在实例化任何类型的 GUI 之前调用，所以，它一般在 Main 方法中调用，如下面的示例所示：



```
[STAThread]
static void Main()
{
    Application.EnableVisualStyles();
    Application.Run(new Form1());
}
```

这段代码允许支持可视化样式的各种控件采用可视化样式。由于 `EnableVisualStyles` 方法存在一个问题，所以必须在调用 `EnableVisualStyles` 之后立即添加 `Application.DoEvents()` 方法。这应能解决工具栏上的图标在运行期间消失的问题。`EnableVisualStyles` 方法也可以在 .NET Framework 1.1 中使用。

对于控件，还有一个必须完成的任务。大多数控件都包含 `FaltStyle` 属性，它把 `FaltStyle` 枚举作为其值。这个属性可以使用如下 4 个值：

- **Flat**: 当鼠标指针停在控件上时，控件显示为平面模式。
- **Flat3D**: 类似于 `Flat`，但当鼠标指针停在控件上时，控件显示为 3D 模式。
- **Standard**: 控件显示为 3D 模式。
- **System**: 控件的外观由操作系统控制。

为了在控件上使用可视化样式，`FaltStyle` 属性应设置为 `FaltStyle.System`。应用程序现在采用 XP 的外观和操作方式。

### 31.4.2 多文档界面

当应用程序可以显示同一类型窗体的多个实例，或以某种方式包含不同的窗体时，就应使用 MDI 类型的应用程序。例如可以同时显示多个编辑窗口的文本编辑器和 Microsoft Access，可以在 Access 中同时打开查询窗口、设计窗口和表窗口。这些窗口都不会超出主 Access 应用程序的边界。

包含本章示例的项目就是一个 MDI 应用程序，项目中的窗体 `mdiParent` 就是 MDI 父窗体。把 `IsMdiContainer` 设置为 `true`，会把该窗体设置为 MDI 父窗体。如果在设计器中创建窗体，注意其背景会变成暗灰色，说明这是一个 MDI 父窗体。仍可以给该窗体添加控件，但最好不要这么做。

为了使子窗体成为 MDI 子窗体，子窗体需要知道其父窗体是哪个窗体。为此，应把子窗体的 `MdiParent` 属性设置为父窗体。在这个例子中，所有的子窗体都是用 `ShowMdiChild` 方法创建的，它的参数是要显示的子窗体的引用。把 `MdiParent` 属性设置为 `this` (表示引用 `mdiParent` 窗体) 后，就显示该窗体。下面是 `ShowMdiParent` 方法的代码：

```
private void ShowMdiChild(Form childForm)
{
    childForm.MdiParent = this;
    childForm.Show();
}
```

MDI 应用程序的一个问题是，在任意时刻可以打开几个子窗体。对当前活动的子窗体的引用可以使用父窗体的 `ActiveMdiChild` 属性来提取。这就是 Window 菜单中 **Current Active** 菜单项的作用。单击该菜单项，会显示一个包含窗体名称和文本值的消息框。



子窗体可以调用 `LayoutMdi` 方法来安排。`LayoutMdi` 方法把 `MdiLayout` 枚举值作为参数。其值可以是 `Cascade`、`TileHorizontal` 和 `TileVertical`。

### 31.4.3 定制控件

使用控件和组件是使窗体软件包(如 Windows 窗体)的开发非常高效的一个主要因素。创建自己的控件、组件和用户控件会使开发更加高效。创建控件可以把功能封装在能多次使用的软件包中。

创建控件有许多方式。可以从头开始创建,从 `Control`、`ScrollableControl` 或 `ContainerControl` 中派生自己的类。除了给控件添加需要的功能之外,还必须重写 `Paint` 事件,完成绘制工作。如果控件是当前控件的一个改进版本,就必须从该控件中派生出要改进的控件。例如,如果需要一个 `TextBox` 控件,但要改变其背景色,设置其 `ReadOnly` 属性,则创建一个全新的 `TextBox` 控件就会浪费时间。从 `TextBox` 控件中派生,再重写 `ReadOnly` 属性即可。由于 `TextBox` 控件的 `ReadOnly` 属性没有标记为重写,所以必须使用 `new` 语句。下面的代码展示了新的 `ReadOnly` 属性:

```
public new bool ReadOnly
{
    get { return base.ReadOnly; }
    set {
        if(value)
            this.BackgroundColor = Color.Red;
        else
            this.BackgroundColor = Color.FromKnownColor(KnownColor.Window);

        base.ReadOnly = value;
    }
}
```

对于属性 `get`, 返回为基本对象设置的内容。把文本框设置为只读的属性在这里并不重要,应把这个功能传递给基本对象。在属性集中,检查传递的值是 `true` 还是 `false`。如果是 `true`,就要将颜色改为只读颜色(在本例中是 `Red`); 如果是 `false`,就把 `BackColor` 设置为默认值。最后,把值传送给基本对象,让文本框成为只读文本框。可以看出,重写一个简单的属性,就可以给控件添加一个新功能。

#### 1. 控件属性

可以给定制控件添加属性,改进设计期间控件的功能。表 31-5 描述了一些比较有用的属性。

表 31-5

属性名称	描述
<code>BindableAttribute</code>	在设计期间用于确定属性是否支持双向数据绑定
<code>BrowsableAttribute</code>	确定属性是否显示在可视化设计器中
<code>CategoryAttribute</code>	确定在属性窗口中,属性显示在哪个类别中。使用预定义的类别,或创建新的类别。默认值为 <code>Misc</code>
<code>DefaultEventAttribute</code>	指定类的默认事件

(续表)

属 性 名 称	描 述
DefaultPropertyAttribute	指定类的默认属性
DefaultValueAttribute	指定属性的默认值。一般是初始值
DecriptionAttribute	在选中属性时，这是显示在设计器窗口底部的文本
DesignOnlyAttribute	把属性标记为只能在设计模式下编辑

还有其他属性(与在设计期间使用的编辑器相关)以及一些在设计期间使用的高级功能。此外还应添加 Category 和 Decription 属性，它们能帮助使用该控件的其他开发人员更好地理解属性的作用。为了添加 Intellisence 支持，应给每个属性、方法和事件都添加 XML 注释。在使用/doc 选项编译控件时，所生成的注释 XML 文件将为控件提供 Intellisence 支持。

2. 基于 TreeView 的定制控件

本节将介绍如何根据 TreeView 控件开发定制控件。这个控件显示驱动器中的文件结构。我们还将给它添加属性，设置基本文件夹或根文件夹，确定显示哪些文件和文件夹，并使用上一节讨论的各种属性。

与任何新项目一样，也必须为控件定义需求。下面是必须完成的基本需求列表：

- 读取文件夹和文件，并显示给用户
- 在树形层次结构视图中显示文件夹结构
- 可以隐藏文件
- 定义哪个文件夹是基本文件夹或根文件夹
- 返回当前选中的文件夹
- 提供延迟加载文件结构的功能

这应是一个好的起点。把 TreeView 控件作为新控件的基本控件，就可以满足上述的一个需求。

TreeView 控件以层次结构的形式显示数据。它显示的数据描述了列表中的对象，并可以带有图标。单击对象或使用箭头键，就可以展开和折叠这个列表。

在 Visual Studio 2008 中创建一个新的 Windows 控件库项目，命名为 FolderTree，删除类 UserControl1，添加一个新类，命名为 FolderTree。因为 FolderTree 派生于 TreeView，所以类声明由：

```
public class FolderTree
```

改为：

```
public class FolderTree : System.Windows.Forms.TreeView
```

此时就有了一个功能全面、且能工作的 FolderTree 控件了。该控件可以完成 TreeView 能完成的所有任务。

TreeView 控件包含一个 TreeNode 对象集合。我们不能直接把文件和文件夹加载到控件中，但有两种方式可以把加载的 TreeNode 映射到 TreeView 的 Node 集合和它表示的文件与文件

夹中。

例如，在处理每个文件夹时，都会创建一个新的 `TreeNode` 对象，`Text` 属性设置为文件或文件夹的名称。如果在某一刻需要文件或文件夹的其他信息，就必须再次进入磁盘收集该信息，或把与文件或文件夹相关的其他数据存储在 `Tag` 属性中。

另一个方法是创建一个派生自 `TreeNode` 的新类。可以添加新属性和方法，且仍能使用 `TreeNode` 的基本功能。本例就使用这个方法，其设计更为灵活。如果需要新属性，就可以添加它们，无需中断已有的代码。

有两类对象必须加载到控件中：文件和文件夹。每类对象都有自己的特性。例如，文件夹有一个 `DirectoryInfo` 对象，它包含了额外的信息，而文件有一个 `FileInfo` 对象。由于有这些区别，所以我们使用两个类来加载 `TreeView` 控件：`FileNode` 和 `FolderNode`。在项目中添加这两个类，每个类都派生于 `TreeNode`。下面是 `FileNode` 的代码：

```
namespace FormsSample.SampleControls
{
    public class FileNode : System.Windows.Forms.TreeNode
    {
        string _fileName = "";
        FileInfo _info;

        public FileNode(string fileName)
        {
            _fileName = fileName;
            _info = new FileInfo(_fileName);
            base.Text = _info.Name;
            if (_info.Extension.ToLower() == ".exe")
                this.ForeColor = System.Drawing.Color.Red;
        }

        public string FileName
        {
            get { return _fileName; }
            set { _fileName = value; }
        }

        public FileInfo FileNodeInfo
        {
            get { return _info; }
        }
    }
}
```

正在处理的文件名被传送给 `FileNode` 的构造函数。在构造函数中，为文件创建 `FileInfo` 对象，并把它设置为成员变量 `_info`。`base.Text` 属性设置为文件名。因为 `FileNode` 派生于 `TreeNode`，所以设置 `TreeNode` 的 `Text` 属性，这是显示在 `TreeView` 控件中的文本。

再添加两个属性来获取数据。`FileName` 返回文件名，`FileNodeInfo` 返回文件的 `FileInfo` 对象。

下面是 `FolderNode` 类的代码。它的结构非常类似于 `FileNode`，区别是用 `DirectoryInfo` 属性代替了 `FileInfo` 属性，用 `FolderPath` 代替了 `FileName`。

```

namespace FormsSample.SampleControls
{
    public class FolderNode : System.Windows.Forms.TreeNode
    {
        string _folderPath = "";
        DirectoryInfo _info;

        public FolderNode(string folderPath)
        {
            _folderPath = folderPath;
            _info = new DirectoryInfo(folderPath);
            this.Text = _info.Name;
        }

        public string FolderPath
        {
            get { return _folderPath; }
            set { _folderPath = value; }
        }

        public DirectoryInfo FolderNodeInfo
        {
            get { return _info; }
        }
    }
}

```

现在构建 FolderTree 控件。根据要求，我们需要一个属性来读取和设置 RootFolder，还需要 ShowFiles 属性，来确定文件是否显示在树中。SelectedFolder 属性返回树中当前突出显示的文件夹。下面是 FolderTree 控件的代码：

```

using System;
using System.Windows.Forms;
using System.IO;
using System.ComponentModel;

namespace FolderTree
{
    ///<summary>
    /// Summary description for FolderTreeCtrl.
    ///</summary>
    public class FolderTree : System.Windows.Forms.TreeView
    {
        string _rootFolder = "";
        bool _showFiles = true;
        bool _inInit = false;

        public FolderTree()
        {
        }

        [Category("Behavior"),
        Description("Gets or sets the base or root folder of the tree"),
        DefaultValue("C:\\")]
        public string RootFolder
        {

```



```

        get {return _rootFolder;}
        set
        {
            _rootFolder = value;
            if(!_inInit)
                InitializeTree();
        }
    }

    [Category("Behavior"),
        Description("Indicates whether files will seen in the list."),
        DefaultValue(true)]
    public bool ShowFiles
    {
        get {return _showFiles;}
        set {_showFiles = value;}
    }

    [Browsable(false)]
    public string SelectedFolder
    {
        get
        {
            if(this.SelectedNode is FolderNode)
                return (FolderNode)this.SelectedNode.FolderPath;

            return "";
        }
    }
}

```

我们添加了 3 个属性：ShowFiles、SelectedFolder 和 RootFolder。注意已经添加的属性。为 ShowFiles 和 RootFolder 设置 Category、Description 和 DefaultValues。这两个属性显示在设计模式下的属性浏览器中。SelectedFolder 在设计期间没有什么意义，所以选择 Browsable=false 属性。SelectedFolder 不在属性浏览器中显示，但由于它是一个公共属性，所以会在 Intellisense 中出现，并可以通过代码访问它。

接着，初始化文件系统的加载操作。初始化控件有点困难。在设计期间和运行期间进行初始化都必须仔细考虑。控件位于设计器中时，实际上是在运行。如果在构造函数中调用了一个数据库，这个调用就是在设计器中拖放控件时进行的。在 FolderTree 控件中，这就存在一个问题。

下面看看加载文件的方法：

```

private void LoadTree(FolderNode folder)
{
    string[] dirs = Directory.GetDirecoties(folder.FolderPath);
    foreach(string dir in dirs)
    {
        folderNode tmpfolder = new FolderNode(dir);
        folder.Nodes.Add(tmpfolder);
        LoadTree(tmpfolder);
    }
    if (_showFiles)
    {

```



```

        string[] files = Directory.GetFiles(folder.FolderPath);
        foreach(string file in files)
        {
            FileNode fnode = new FileNode(file);
            folder.Nodes.Add(fnode);
        }
    }
}

```

showFiles 是一个布尔成员变量，在 ShowFiles 属性中设置。如果设置为 true，就在树中显示文件。现在唯一的问题是何时调用 LoadTree。我们有几种选择。可以在设置 RootFolder 属性时调用，这在一些情况下是很理想的，但在设计期间并不合适。控件在设计器中是激活的，所以在设置 RootNode 属性时，控件就会试图加载文件系统。

要解决这个问题，应查看 DesignMode 属性，如果控件在设计器中，它就返回 true。现在可以编写代码，初始化控件了：

```

private void InitializeTree()
{
    if(!this.DesignMode && _rootFolder != "")
    {
        FolderNode rootNode = new FolderNode(_rootFolder);
        LoadTree(rootNode);
        this.Nodes.Clear();
        this.Nodes.Add(rootNode);
    }
}

```

如果控件不处于设计模式，且 \_rootFolder 也不是空字符串，就开始加载树。先创建 Root 节点，再把它传送给 LoadTree 方法。

另一个选择是执行公共方法 Init。在 Init 方法中，调用 LoadTree 方法。这个选择存在的问题是使用控件的开发人员需要调用 Init。根据情况的不同，这或许是一个可以接受的解决方法。

对于添加的灵活性，可以实现 ISupportInitialize 接口。ISupportInitialize 有两个方法 BeginInit 和 EndInit。在控件实现 ISupportInitialize 接口时，就会在 InitializeComponent 生成的代码中自动调用 BeginInit 和 EndInit 方法。这样就可以延缓初始化过程，直到设置完所有的属性为止。ISupportInitialize 还允许父窗体中的代码延缓初始化过程。如果在代码中设置 RootNode，先调用 BeginInit 就可以在控件加载文件系统之前设置 RootNode 属性和其他属性，或执行动作。在调用 EndInit 时，初始化控件。BeginInit 和 EndInit 方法如下所示：

```

#region ISupportInitialize Members

void ISupportInitialize.BeginInit()
{
    _inInit = true;
}

void ISupportInitialize.EndInit ()
{
    if (_rootFolder != "")
    {
        InitializeTree();
    }
}

```

```

        _inInit = false;
    }

    #endregion

```

在 `BeginInit` 方法中，只是把成员变量 `_inInit` 设置为 `true`。这个标志用于确定控件是否处于初始化过程，并在 `RootFolder` 属性中使用。如果在 `InitializeComponent` 类的外部设置 `RootFolder` 属性，树就需要重新初始化。在 `RootFolder` 属性中，检查 `_inInit` 是 `true` 还是 `false`。如果是 `true`，就不需要查看初始化过程。如果 `_inInit` 是 `false`，就调用 `InitializeTree`。还可以用一个公共方法 `Init` 完成这个任务。

在 `EndInit` 方法中，检查控件是否处于设计模式，`_rootFolder` 是否被赋予了有效的路径。然后仅调用 `InitializeTree`。

为了添加专业化的外观，需要添加一个位图图像。当把控件添加到项目中时，这个图标将显示在工具箱中。位图图像应是  $16 \times 16$  像素，16 色。可以用任意图形编辑器创建这个图像文件，只要正确设置了大小和颜色深度即可。甚至可以在 Visual Studio 2008 中创建这个文件：右击项目，选择 `Add New Item`。在列表中选择 `Bitmap File`，打开图像编辑器。在创建好位图文件之后，把它添加到项目中，确保它位于控件的命名空间中，且与控件同名。最后，把位图的 `Build Action` 设置为 `Embedded Resource`：在 `Solution Explorer` 中右击位图文件，选择 `Properties`。然后从 `Build Action` 属性中选择 `Embedded Resource`。

要测试控件，在同一个解决方案中创建 `TestHarness` 项目。`TestHarness` 是一个简单的 Windows 窗体应用程序，只包含一个窗体。在引用部分添加对 `FolderTreeCtl` 项目的引用。在 `Toolbox` 窗口中添加对 `FolderTreeCtl.DLL` 的引用。`FolderTreeCtl` 现在应显示在工具箱中，且位图已添加为图标。单击该图标，把它拖放到 `TestHarness` 窗体上。把 `RootFolder` 设置为一个可用的文件夹，并运行解决方案。

这还不是一个完整的控件。还必须做一些工作，改进控件，使之成为功能全面、可用于产品的控件。例如，可以添加：

- **异常：**如果控件试图加载用户不能访问的文件夹，就会引发一个异常。
- **后台加载：**加载大的文件夹树是很费时间的。应改进初始化过程，以利用后台线程来加载大文件夹树。
- **颜色代码：**可以让某些类型的文本显示为不同的颜色。
- **图标：**可以添加一个 `ImageList` 控件，在加载时为每个文件或文件夹添加一个图标。

### 3. 用户控件

用户控件是 Windows 窗体中功能比较强大的一个特性。它们允许把用户界面设计封装到可重用的软件包中，以便以后把软件包插入其他项目。公司有若干个常用用户控件库是很常见的。不仅用户界面功能可以包含在用户控件中，而且常见的数据验证也可以包含在用户控件中，例如格式化电话号码或 id 号。在用户控件中可以包含一个预定义的项目列表，以便快速加载列表框或组合框。状态代码或国家代码也可以划在这个类别中。在用户控件中尽可能组合许多与当前应用程序不相关的功能，会使控件在公司中的用途更大。

本节将创建一个简单的地址用户控件，并添加两个事件，使控件可用于数据绑定。地址控件有几个文本字段用于输入两个地址行：城市、州和邮政编码。

要在当前项目中创建用户控件，只需在 Solution Explorer 中右击项目，选择 Add，再选择 Add New User Control。还可以创建一个新的 Control Library 项目，并添加用户控件。在启动新的用户控件后，就会在设计器上看到一个没有边框的窗体。在这里可以拖放组成用户控件的控件。用户控件实际上是把一个或多个控件添加到一个容器控件中，这有点类似于创建窗体。对于地址控件，需要 5 个文本框控件和 3 个标签控件。这些控件的布局可以任意，只要看着舒服即可，如图 31-4 所示。

这个示例中的 TextBox 控件命名如下：

- txtAddress1
- txtAddress2
- txtCity
- txtState
- txtZip

在安排好了文本框控件，并指定了有效的名称后，就可以添加公共属性。我们希望把 TextBox 控件的可访问性设置为 public，而不是 private，但这并不好，因为这违背了封装要添加到属性中的功能的初衷。下面是必须添加的属性代码：

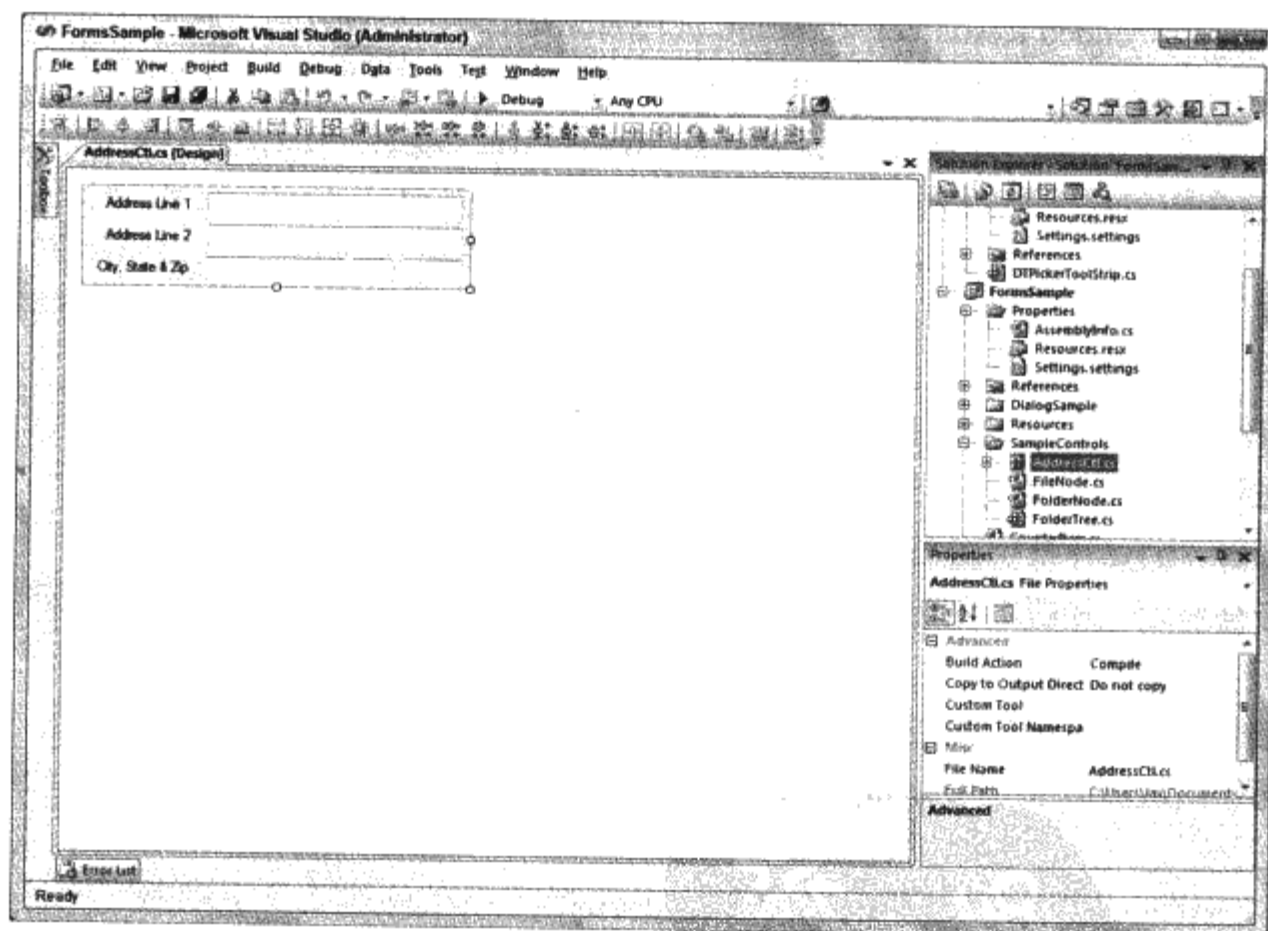


图 31-5

```
public string AddressLine1
{
    get{return txtAddress1.Text;}
    set{
        if(txtAddress1.Text != value)
        {
            txtAddress1.Text = value;
            if(AddressLine1Changed != null)
                AddressLine1Changed(this, PropertyChangedEventArgs.Empty);
        }
    }
}
```

```

    }
}

public string AddressLine2
{
    get{return txtAddress2.Text;}
    set{
        if(txtAddress2.Text != value)
        {
            txtAddress2.Text = value;
            if(AddressLine2Changed != null)
                AddressLine2Changed(this, PropertyChangedEventArgs.Empty);
        }
    }
}

public string City
{
    get{return txtCity.Text;}
    set{
        if(txtCity.Text != value)
        {
            txtCity.Text = value;
            if(CityChanged != null)
                CityChanged(this, PropertyChangedEventArgs.Empty);
        }
    }
}

public string State
{
    get{return txtState.Text;}
    set{
        if(txtState.Text != value)
        {
            txtState.Text = value;
            if(StateChanged != null)
                StateChanged(this, PropertyChangedEventArgs.Empty);
        }
    }
}

public string Zip
{
    get{return txtZip.Text;}
    set{
        if(txtZip.Text != value)
        {
            txtZip.Text = value;
            if(ZipChanged != null)
                ZipChanged(this, PropertyChangedEventArgs.Empty);
        }
    }
}

```

属性的 `get` 部分非常简单，它返回相应 `TextBox` 控件的 `Text` 属性值。属性的 `set` 部分做的工作比较多。所有的 `set` 部分都以相同的方式工作。先检查属性值是否改变。如果新值与当前值相同，就快速退出。如果不相同，就把文本框的 `Text` 属性设置为新值，并测试事件是否实例



化。要查找的事件是属性的修改事件。它采用特殊的命名格式 `propertynameChanged`，其中 `propertyname` 是属性的名称。在 `AddressLine1` 属性中，这个事件称为 `AddressLine1Changed`。该属性的声明如下：

```
public event EventHandler AddressLine1Changed;
public event EventHandler AddressLine2Changed;
public event EventHandler CityChanged;
public event EventHandler StateChanged;
public event EventHandler ZipChanged;
```

事件的作用是通知绑定操作，属性已改变。一旦进行验证，绑定操作就会确保把新值返回给控件所绑定的对象。要支持绑定，还需要做另一个工作。用户对文本框的修改将不再直接设置属性。所以在文本框改变时也必须引发 `propertynameChanged` 事件。最简单的方式是监视 `TextBox` 控件的 `TextChanged` 事件。这个示例只包含一个 `TextChanged` 事件处理程序，且所有的文本框都使用它。检查控件名，确定是哪个控件引发了事件，并引发相应的 `propertynameChanged` 事件。下面是事件处理程序的代码：

```
private void TextBoxControls_TextChanged(
    object sender, System.EventArgs e)
{
    switch(((textBox)sender).Name)
    {
        case "txtAddress1":
            if(AddressLine1Changed != null)
                AddressLine1Changed(this, EventArgs.Empty);

            break;

        case "txtAddress2":
            if(AddressLine2Changed != null)
                AddressLine2Changed(this, EventArgs.Empty);

            break;

        case "txtCity":
            if(CityChanged != null)
                CityChanged(this, EventArgs.Empty);

            break;

        case "txtState":
            if(StateChanged != null)
                StateChanged(this, EventArgs.Empty);

            break;

        case "txtZip":
            if(ZipChanged != null)
                ZipChanged(this, EventArgs.Empty);

            break;
    }
}
```

这个示例使用简单的 `switch` 语句来确定是哪个文本框引发了 `TextChanged` 事件。接着进行



检查，验证事件是有效的，且不等于空。然后引发 `TextChanged` 事件。注意发送了一个空 `EventArgs (EventArgs.Empty)`。这些事件都已添加到属性中，以支持数据绑定，但这并不意味着使用该控件必须进行数据绑定。这些属性可以在代码中设置和读取，不必使用数据绑定。它们都被添加进来，所以用户控件可以在绑定可用时使用绑定。这只是使用户控件尽可能灵活的一种方式，以用于尽可能多的场合。

用户控件实际上是带有某些额外特性的控件，前一节讨论的所有在设计期间可能存在的问题在用户控件中也会出现。初始化用户控件可能会引发 `FolderTree` 示例中探讨的问题。所以在设计用户控件时必须小心，应避免出现使用该控件的其他开发人员不能访问数据存储等问题。

与控件的创建类似，属性也可以应用于用户控件。当把用户控件放在设计器中时，它的公共属性和方法会显示在属性窗口中。在地址用户控件的示例中，最好给地址属性添加 `Category`、`Description` 和 `DefaultValue` 特性。可以创建一个新的 `AddressData` 类别，并把默认值都设置为“”。下面是这些特性应用于 `AddressLine1` 属性的示例：

```
[Category("AddressData");
    Description("Gets or sets the AddressLine1 value"),
    DefaultValue(" ")]
public string AddressLine1
{
    get{return txtAddress1.Text;}
    set{
        if(txtAddress1.Text != value)
        {
            txtAddress1.Text = value;
            if(AddressLine1Changed != null)
                AddressLine1Changed(this, EventArgs.Empty);
        }
    }
}
```

可以看出，要添加新的类别，只需在 `Category` 特性中设置文本，新类别就会自动添加。

仍有许多可以改进的地方。例如，可以在控件中包含一个州名和缩写列表。除了 `state` 属性之外，用户控件还可以显示州名和州的缩写。我们还应添加异常处理，并可以添加地址行的验证功能。确保大小写正确，弄清楚 `AddressLine1` 是否可选，在 `AddressLine2` 中是否应输入单元和房间号，而不是在 `AddressLine1` 中输入。

## 31.5 小结

本章介绍了建立基于 Windows 的客户应用程序的基础知识。解释了每个基本控件，讨论了 `Windows.Forms` 命名空间的层次结构，论述了控件的各种属性和方法。

我们还阐述了如何创建基本定制控件和基本用户控件。创建自己的控件是非常灵活的，无论如何强调都不过分。通过创建自己的定制控件工具箱，基于 Windows 的客户应用程序将更容易开发和测试，因为可以重用测试过的同一组件。

下一章介绍如何把数据源链接到窗体的控件上，这样可以使创建出来的窗体自动更新数据，使之与窗体上的数据同步。

# 第32章

## 数 据 绑 定

第 26 章介绍了选择和修改数据的各种方式,本章接着第 25 章的内容,继续介绍如何把绑定到各种 Windows 控件上的数据显示给用户。本章主要内容如下:

- 使用 DataGridView 控件显示数据
- .NET 数据绑定功能及其工作方式
- 如何使用 Server Explorer 创建连接,生成 DataSet 类(不需要编写代码)
- 如何对 DataGrid 中的数据行进行测试和反射

本章的示例代码可以从 Wrox 网站 [www.wrox.com](http://www.wrox.com) 上下载。

### 32.1 DataGridView 控件

.NET 的最初版本中的 DataGrid 控件有强大的功能,但在许多方面,它都不适用于商业应用程序,例如不能显示图像、下拉控件或锁定列等。该控件给人感觉只完成了一半,所以许多控件经销商都提供了定制的栅格控件,以克服这些缺陷,并提供更多的功能。

在.NET 2.0 中有了另一个栅格控件 DataGridView。它解决了 DataGrid 控件最初的许多问题,还增加了许多只能在插件产品中使用的功能。

新控件具有与 DataGrid 类似的绑定功能,可以绑定到 Array、DataTable、DataView 或 DataSet 类,或者绑定到实现 IListSource 或 IList 接口的组件上。DataGridView 控件可以显示数据的许多视图。在最简单的情况下,设置 DataSource 和 DataMember 属性,就可以显示数据(与 DataSet 类一样)。注意,这个新控件不是 DataGrid 的插件替代品,所以其编程接口完全不同于 DataGrid。这个控件还提供了更复杂的功能,本章将讨论这些功能。

#### 32.1.1 显示列表数据

第 19 章介绍了选择数据和把数据放在一个数据表中的各种方式,但仅使用了 Console.WriteLine()方法以非常基本的形式显示数据。

下面的示例将说明如何获取一些数据,并在 DataGridView 控件中显示,为此,建立一个新的应用程序 DisplayTabularData,如图 32-1 所示。

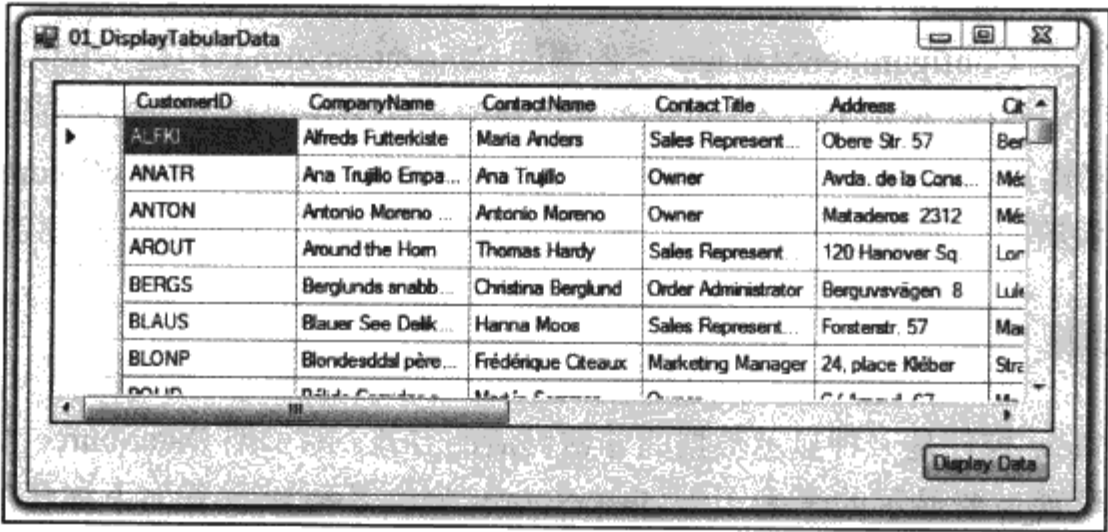


图 32-1

这个简单的应用程序从 Northwind 数据库的 customer 表中选择每个记录，在 DataGridView 控件中把它们显示给用户。其代码如下所示(不包含窗体和控件定义代码):

```
using System;
using System.Configuration;
using System.Data;
using System.Data.Common;
using System.Data.SqlClient;
using System.Windows.Forms;

namespace DisplayTabularData
{
    partial class Form1: Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void getData_Click(object sender, EventArgs e)
        {
            string customers = "SELECT * FROM Customers";

            using (SqlConnection con =
                new SqlConnection (ConfigurationSettings.
                    ConnectionStrings["northwind"].ConnectionString))
            {
                DataSet ds = new DataSet();

                SqlDataAdapter da = new SqlDataAdapter(customers, con);

                da.Fill(ds, "Customers");

                dataGridView.AutoGenerateColumns = true;
                dataGridView.DataSource = ds;
                dataGridView.DataMember = "Customers";
            }
        }
    }
}
```

窗体包含 `getData` 按钮，单击它会调用示例代码中的 `getData_Click` 方法。

这段代码使用 `ConfigurationManager` 类的属性 `ConnectionStrings` 构建了 `SqlConnection` 对象。之后构建一个 `DataSet`，使用 `DataAdapter` 对象填充数据库表中的数据。然后设置 `DataSource` 和 `DataMember` 属性，在 `DataGridView` 控件中显示数据。注意 `AutoGenerateColumns` 属性也设置为 `true`，以确保给用户显示一些数据。如果这个标记没有指定，就需要自己创建所有的列。

### 32.1.2 数据源

`DataGridView` 控件是一种显示数据的非常灵活的方式。除了把 `DataSource` 设置为 `DataSet`，`DataMember` 设置为要显示的表名之外，`DataSource` 属性还可以设置为下述任何一个数据源：

- 数组(网格可以绑定到任何一个一维数组上)
- `DataTable`
- `DataView`
- `DataSet` 或 `DataViewManager`
- 实现 `ICollection` 接口的组件
- 实现 `IStructuralList` 接口的组件
- 泛型集合类或派生于泛型集合类的对象

下面几节将给出这些数据源的示例。

#### 1. 显示数组中的数据

这初看起来非常简单，创建一个数组，填充一些数据，再在 `DataGridView` 控件上设置 `DataSource` 属性。下面是一些示例代码：

```
string[] stuff = new string[] { "One", "Two", "Three" };
dataGridView1.DataSource = stuff;
```

如果数据源包含多个表(例如使用 `DataSet` 或 `DataViewManager`)，就需要设置 `DataMember` 属性。

可以用这个数组代码替换上面示例中的 `getData_Click` 事件处理程序，这段代码的结果如图 32-2 所示。

可以看出，网格显示出了数组中定义的字符串的长度，而不是这些字符串。原因是在把数组用作 `DataGridView` 控件的数据源时，网格会查找数组中对象的第一个公共属性，并显示这个值，而不会显示字符串值。字符串的第一个(也是唯一的)公共属性是其长度，所以就显示这个长度值。使用 `TypeDescriptor` 类的 `GetProperties` 方法可以获得任意类的属性列表，该方法返回的是一个 `PropertyDescriptor` 对象集合，接着，就可以在显示数据时使用它。`.NET` 的 `PropertyGrid` 控件在显示任意对象时，就使用这个方法。

在 `DataGridView` 中显示字符串的一种解决方法是创建一个包装器类，如下所示：

```
protected class Item
{
    public Item(string text)
    {
        _text = text;
```

```

    }
    public string Text
    {
        get{return _text;}
    }
    private string _text;
}

```

在数据源数组代码中添加这个 Item 类(从进行的各种处理来讲,它也可以是一个结构)的数组后,结果如图 32-3 所示。



图 32-2



图 32-3

## 2. DataTable

有两种方式在 DataGridView 控件中显示 DataTable:

- 如果 DataTable 是独立的,就把控件的 DataSource 属性设置为这个表。
- 如果在 DataSet 中包含 DataTable,就把控件的 DataSource 属性设置为 DataSet, DataMember 属性设置为 DataSet 中的 DataTable 名。

图 32-4 所示为运行 DataSourceDataTable 示例代码的结果。

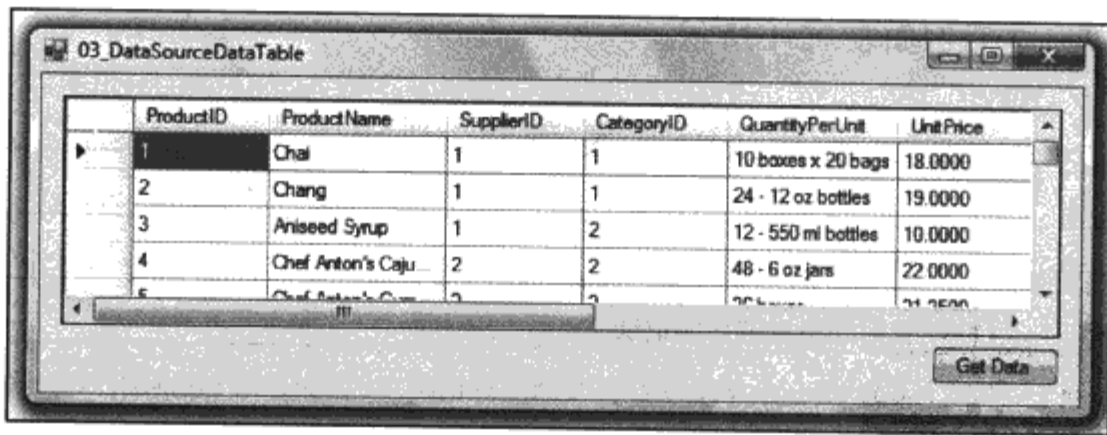


图 32-4

注意,最后一列显示了一个复选框,而不是更常见的编辑控件。DataGridView 控件没有显示其他信息,而是从数据源中读取模式(在本例中是 Products 表),根据列的类型推断应显示什么控件。与以前的 DataGrid 控件不同,DataGridView 控件还可以显示图像列、按钮和组合框。

在修改了 DataGrid 中的字段时,数据库中的数据不会改变,因为此时数据仅存储在本地计算机上——没有与数据库的活动连接。后面将讨论更新数据源中的数据。



3. 显示 DataView 中的数据

DataView 提供了一种过滤和排序 DataTable 中数据的一种方式。在从数据库中选择数据时，用户一般可以单击列标题，对数据排序。此外，还可以只过滤要显示在某些行中的数据，例如用户修改过的所有数据。DataView 允许过滤要显示给用户的数据行，但不允许过滤 DataTable 中的数据列。

提示：  
DataView 不允许过滤要显示的数据列，只允许过滤要显示的数据行。

根据现有的 DataTable 创建 DataView 的代码如下所示：

```
ViewSource code...  
DataView dv = new DataView(dataTable);
```

创建好后，就可以改变 DataView 上的设置，当该视图显示在 DataGrid 中时，这些设置会影响要显示的数据，以及对这些数据进行的操作。例如：

- 设置 AllowEdit = false 表示在数据行上禁用所有列的编辑功能。
- 设置 AllowNew = false 表示禁用新行功能。
- 设置 AllowDelete = false 表示禁用删除行的功能。
- 设置 RowStateFilter 只显示指定状态的行。
- 设置 RowFilter 可过滤数据行。

下一节将介绍如何使用 RowStateFilter 设置，其他选项都很容易理解。

(1) 通过数据过滤数据行

创建好 DataView 后，就可以通过设置 RowFilter 属性，来改变视图显示的数据。这个属性是一个字符串，可用作按照给定条件过滤数据的一种方式——该字符串的值就是过滤条件。其语法类似于一般 SQL 中的 WHERE 子句，但主要是对已经从数据库中选择出来的数据进行操作。

过滤子句的一些示例如表 32-1 所示。

表 32-1	
子 句	说 明
UnitsInStock > 50	只显示 UnitsInStock 列大于 50 的行
Client = 'Smith'	只返回给定客户的记录
County LIKE 'C*'	返回 County 字段以 C 开头的所有记录——例如返回 Cornwall、Cumbria、Cheshire 和 Cambridgeshire 所在的行。可以使用 % 表示匹配一个字符的通配符，而 * 表示匹配 0 个或多个字符的通配符

运行库尽可能在过滤表达式中使用与源列相匹配的数据类型。例如，在前面的示例中，使用 UnitsInStock > '50' 表达式就是合法的，尽管该列是一个整数列。但如果提供了一个无效的过滤字符串，就会产生 EvaluateException 异常。

(2) 根据状态过滤数据行

DataView 中的每一行都有一个定义好的行状态，它们的值如表 32-2 所示，这些状态也可

以用于过滤用户查看的行。

表 32-2

DataViewRowState	说 明
Added	列出新创建的所有行
CurrentRows	列出除了被删除的行以外的其他行
Deleted	列出最初被选中，且已经删除的行——不显示已经删除的新建行
ModifiedCurrent	列出所有已被修改的行，并显示这些行的当前值
ModifiedOriginal	列出所有已被修改的行，但显示这些行的初值，而不是当前值
OriginalRows	列出最初从数据源中选中的所有行，不包括新行。显示列的初值(即如果进行了修改，不显示当前值)
Unchanged	列出没有修改的行

图 32-5 显示了两个网格，一个网格显示已添加、删除或修改的数据行，另一个网格显示状态为表 32-2 中一种的行。

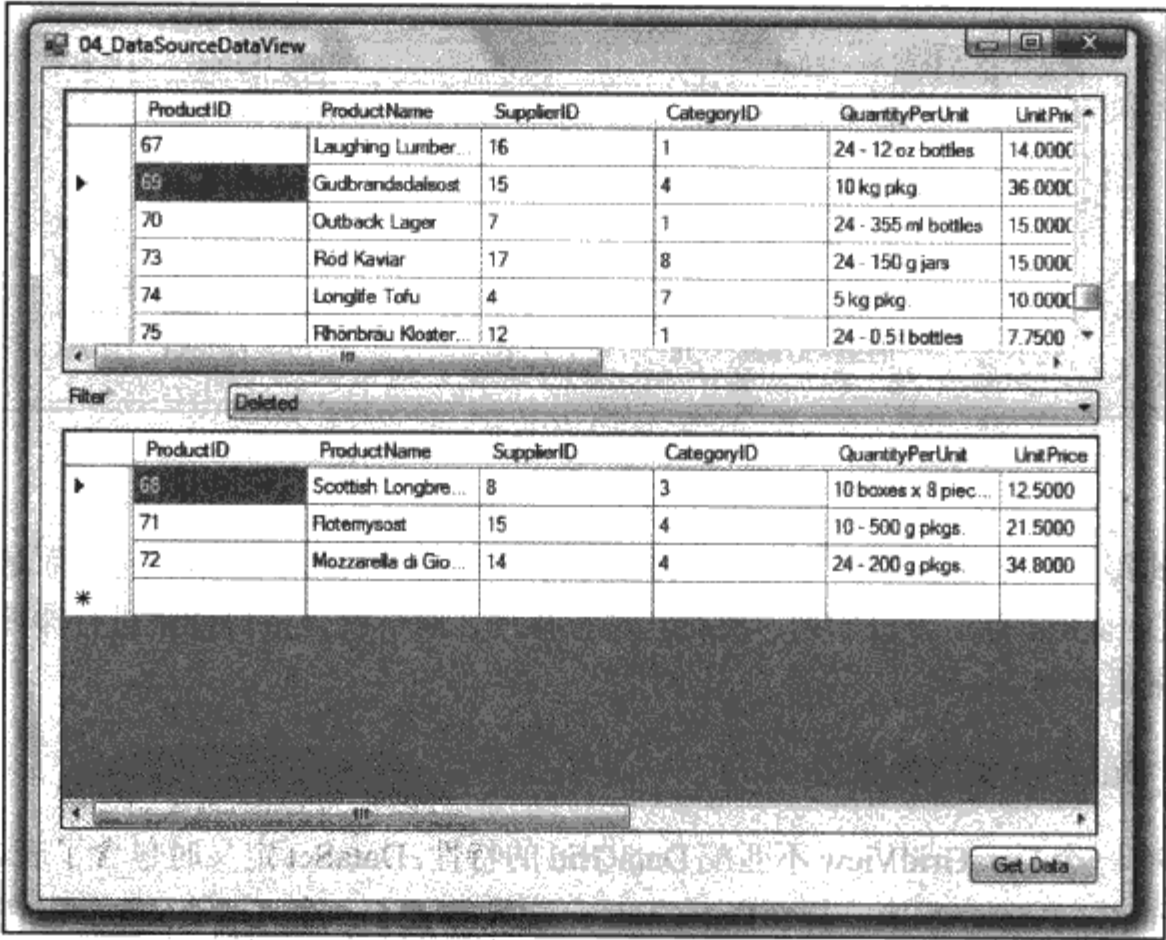


图 32-5

过滤器不仅可以用于可见的行，还可以用于这些行中列的状态。在进行 Modified- Original 或 ModifiedCurrent 选择时，这是很明显的。这两个状态都在第 20 章介绍过了，它们都是基于 DataRowVersion 枚举的。例如，如果用户更新了数据行中的一列，该行就会在选择 ModifiedOriginal 或 ModifiedCurrent 时显示出来，但其实际值可以是数据库中选择出来的初值(如果选择了 ModifiedOriginal)，或者 DataColumn 中的当前值(如果选择了 ModifiedCurrent)。

### (3) 对数据行进行排序

除了过滤数据外,有时还需要对 DataView 中的数据进行排序。可以在 DataGridView 控件中单击列标题,这会按照升序或降序的顺序对该列进行排序,如图 32-6 所示。唯一的问题是控件只能对一列进行排序,而底层的 DataView 控件可以对多个列进行排序。

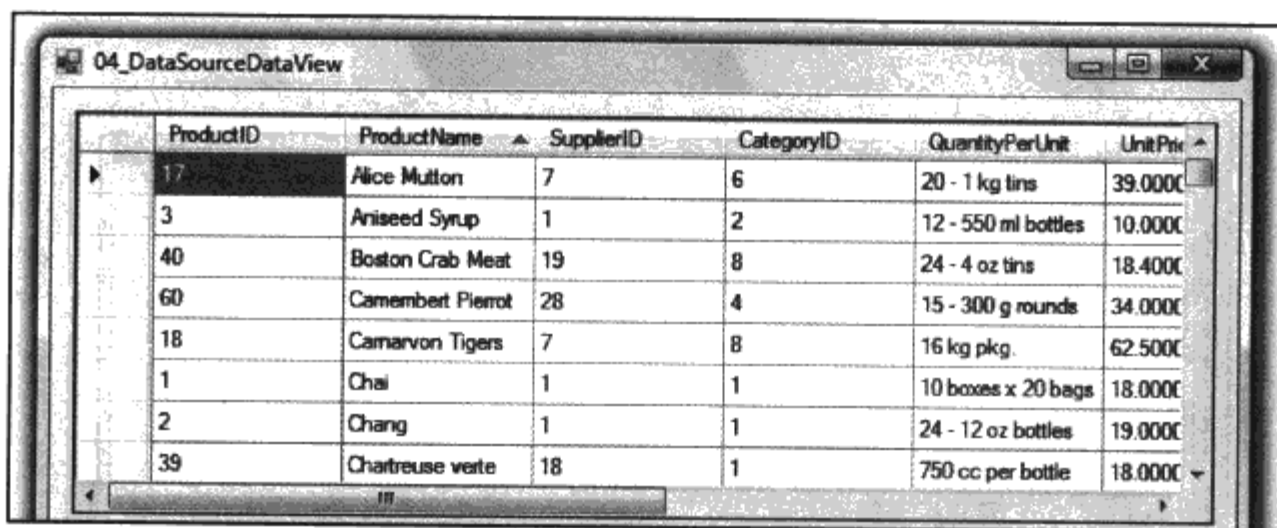
在对数据列进行排序时,可以单击列的标题(例如上面的 ProductName 列),也可以通过代码排序,DataGridView 会显示一个箭头位图,表示对哪一列进行排序。

要编程设置列的排列顺序,可以使用 DataView 的 Sort 属性:

```
dataView.Sort = "ProductName";
dataView.Sort = "ProductName ASC, ProductID DESC";
```

上面的第一行按照 ProductName 列对数据排序,如图 32-6 所示。第二行按照 ProductName 列对数据进行升序排序,再以 ProductID 的降序来排序。

DataView 支持对列进行升序或降序排序——默认为升序。如果选择对 DataView 中的多个列进行排序,DataGridView 就不会显示任何排序箭头。



ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitPrice
17	Alice Mutton	7	6	20 - 1 kg tins	39.0000
3	Aniseed Syrup	1	2	12 - 550 ml bottles	10.0000
40	Boston Crab Meat	19	8	24 - 4 oz tins	18.4000
60	Camembert Pierrot	28	4	15 - 300 g rounds	34.0000
18	Camaron Tigers	7	8	16 kg pkg.	62.5000
1	Chai	1	1	10 boxes x 20 bags	18.0000
2	Chang	1	1	24 - 12 oz bottles	19.0000
39	Chartreuse verte	18	1	750 cc per bottle	18.0000

图 32-6

网格中的每一列都是强类型化的,其排序顺序不是基于列的字符串表示,而是基于该列的数据。如果 DataGridView 有一个日期列,要对它进行排序,网格就会按日期来进行排序,而不是按日期字符串来进行排序。

### 4. 显示 DataSet 类中的数据

DataSet 有一个 DataGridView 不匹配 DataGridView 的特性:DataSet 定义时包含了表之间的关系。和以前的 DataGridView 示例一样,DataGridView 一次只能显示一个 DataTable,但在下面的示例 DataSourceDataSet 中,可以在屏幕上浏览 DataSet 中的关系。下面的代码可以根据 Northwind 数据库中的 Customers 和 Orders 表生成这样一个 DataSet。这个示例从两个 DataTable 中加载数据,然后在这些表之间创建了一个关系 CustomerOrders:

```
string orders = "SELECT * FROM Orders";
string customers = "SELECT * FROM Customers";
SqlConnection conn = new SqlConnection(source);
SqlDataAdapter da = new SqlDataAdapter(orders, conn);
```

```

DataSet ds = new DataSet();
da.Fill(ds, "Orders");
da = new SqlDataAdapter(customers, conn);
da.Fill(ds, "Customers");
ds.Relations.Add("CustomerOrders",
    ds.Tables["Customers"].Columns["CustomerID"],
    ds.Tables["Orders"].Columns["CustomerID"]);

```

创建好后，通过调用 `SetDataBinding`，就可以把 `DataSet` 绑定到 `DataGrid` 上：

```
dataGrid1.SetDataBinding(ds, "Customers");
```

这会得到如图 32-7 所示的屏幕图。

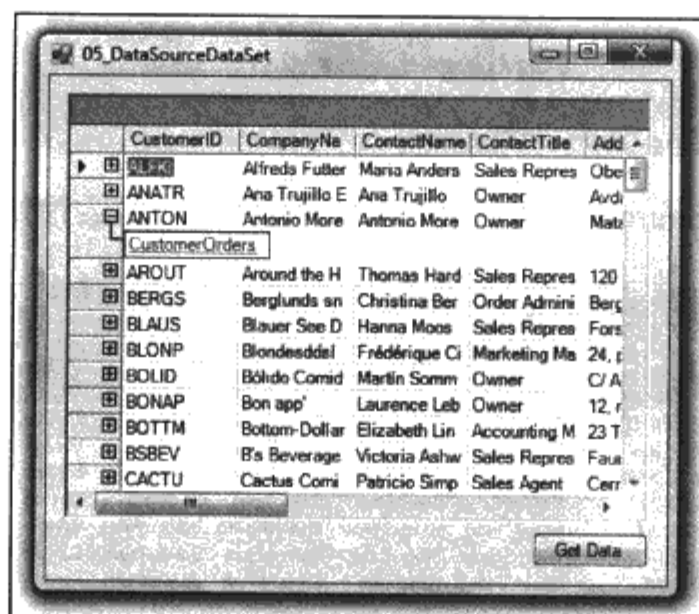


图 32-7

与本章前面的 `DataGridView` 示例不同，每个记录的左边都有一个+号，这表示 `DataSet` 在 `customers` 和 `orders` 表之间有一个可导航的关系。在代码中可以定义许多这类关系。

单击+号，就会显示关系列表(如果关系已经显示出来，单击+号就会隐藏该关系)。单击关系名，就可以定位到链接的记录上，如图 32-8 所示，在本例中是列出选中客户的所有订单。

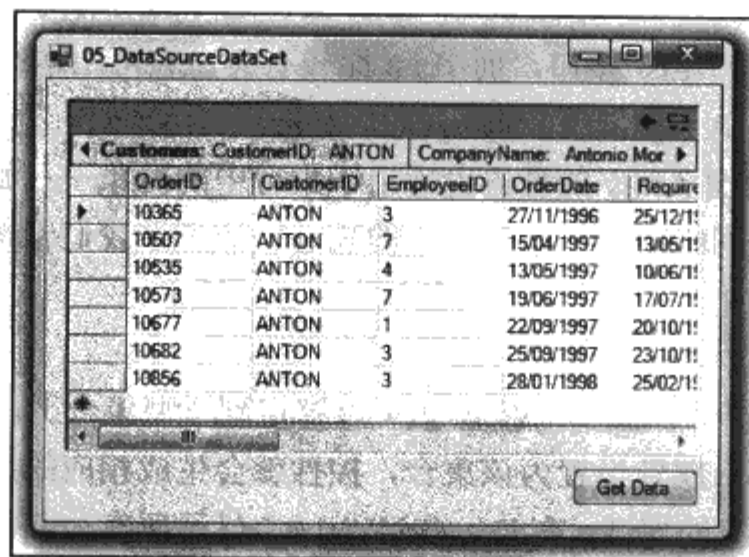


图 32-8

DataGrid 控件的右上角还包含两个新图标。箭头允许用户导航回父行，显示上一页的内容。标题行显示父记录的细节，单击另一个按钮会隐藏或显示该箭头。

在 DataViewManager 中显示数据

DataViewManager 中显示的数据与 DataSet 中显示的数据相同，但在为 DataSet 创建 DataViewManager 时，会为每个 DataTable 创建一个单独的 DataView，再执行代码，根据过滤条件或者行的状态改变显示出来的行，如上面的 DataView 示例所示。即使代码不需要过滤数据，也可以把 DataSet 包装到 DataViewManager 中以进行显示，因为这样在修改源代码时可以使用更多的选项。

下面的示例根据上一例中的 DataSet 创建一个 DataViewManager，然后改变 Customer 表中的 DataView，使之只显示来自英国的客户：

```
DataViewManager dvm = new DataViewManager(ds);
dvm.DataViewSettings["Customers"].RowFilter = "Country='UK'";
dataGrid.SetDataBinding(dvm, "Customers");
```

如图 32-9 所示为 DataSourceDataViewManager 示例代码的运行结果。

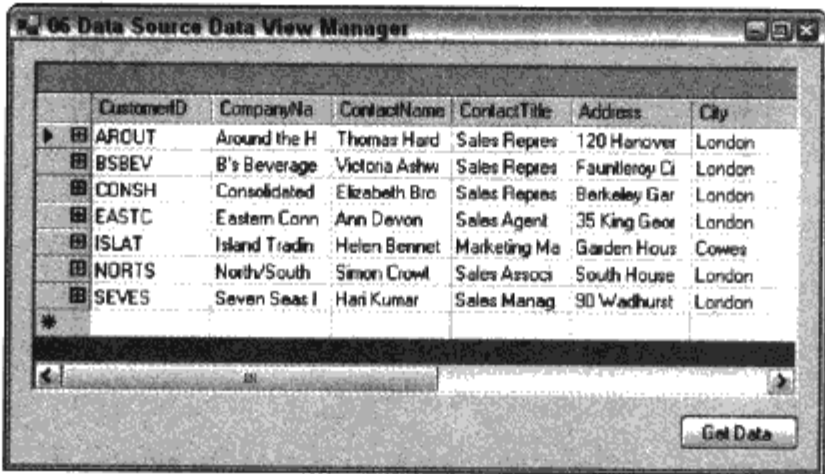


图 32-9

5. IListSource 和 IList 接口

DataGridView 还支持执行 IListSource 和 IList 接口之一的所有对象。IListSource 只有一个方法 GetList(), 它返回一个 IList 接口，而 IList 比较有趣，它可由运行库中的许多类执行，执行这个接口的类有 Array、ArrayList 和 StringCollection。

在使用 IList 时，对前面 Array 的警告也适用于集合中的对象——如果使用 StringCollection 作为 DataGrid 的数据源，网格就会显示字符串的长度，而不是我们希望显示的元素文本。

6. 显示泛型集合

除了已介绍的类型之外，DataGridView 还可以绑定到泛型集合上。其语法与本章前面的示例相同，也是把 DataSource 属性设置为该集合，控件就会生成相应的显示结果。

所显示的列也基于对象的属性：在 DataGridView 中显示所有公共的可读字段。下面的示例显示了一个定义好的 list 类：

```
class PersonList : List < Person >
```



```

{
}

class Person
{
    public Person( string name, Sex sex, DateTime dob )
    {
        _name = name;
        _sex = sex;
        _dateOfBirth = dob;
    }

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }

    public Sex Sex
    {
        get { return _sex; }
        set { _sex = value; }
    }

    public DateTime DateOfBirth
    {
        get { return _dateOfBirth; }
        set { _dateOfBirth = value; }
    }

    private string _name;
    private Sex _sex;
    private DateTime _dateOfBirth;
}

enum Sex
{
    Male,
    Female
}

```

这段代码显示了 Person 类的几个实例，它们是在 PersonList 类中构建的，如图 32-10 所示。

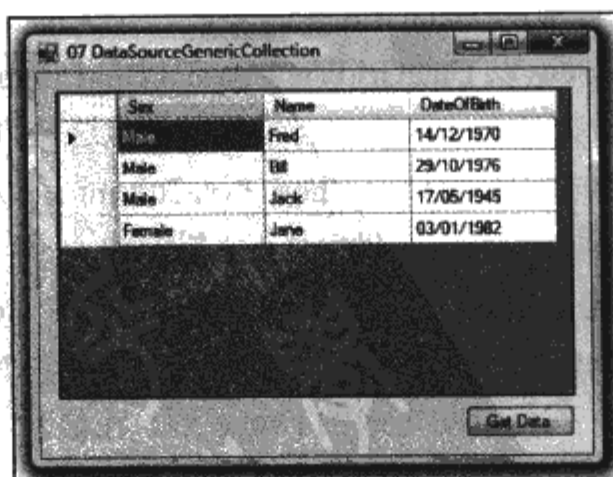


图 32-10

在一些情况下，需要在网格中隐藏某些属性，此时可以使用 Browsible 特性，如上面的代

码所示。标记为 non-browsable 的属性不会显示在属性网格中。

```
[Browsable(false)]
public bool IsEmployed
{
    ...
}
```

DataGridView 使用 Browsable 特性确定是显示还是隐藏一个属性。如果没有设置 Browsable 特性，就默认为显示属性。如果属性是只读的，网格控件就显示对象的值，但在网格中它是只读的。

在网格视图中的所有修改都会反映到底层对象上。例如，如果在上面的代码中，修改了用户界面上的人名，就会调用该属性的设置器方法。

32.2 DataGridView 类的层次结构

DataGridView 主要部分的类层次结构如图 32-11 所示。

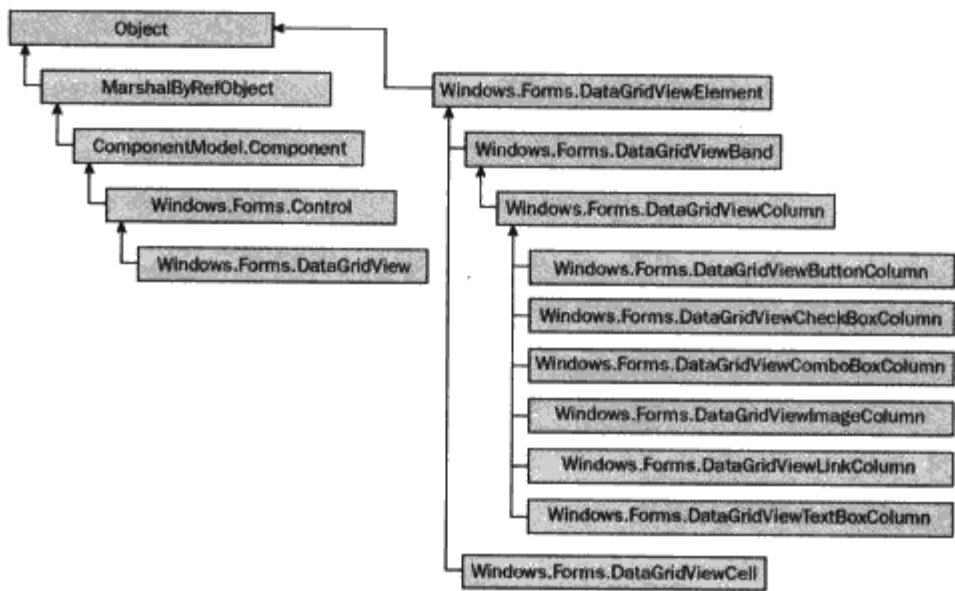


图 32-11

DataGridView 控件在显示数据时利用了派生自 DataGridViewColumn 的对象，如图 32-11 所示。现在，显示数据的选项比以前的 DataGrid 控件多。一个大的变化是在 DataGrid 中显示下拉列的这个功能，现在 DataGridView 以 DataGridViewComboBoxColumn 的形式提供。

在为 DataGridView 指定数据源时，默认要自动构建列。这些列是根据数据源中的数据类型创建的，所以布尔字段映射为 DataGridViewCheckBoxColumn。如果要自己完成列的创建，就可以把 AutoGenerateColumns 属性设置为 false，自己构建列。

下面的例子演示了如何构建列，并包含一个图像和一列 ComboBox。代码利用了一个 DataSet，把数据提取到两个 DataTable 中。第一个 DataTable 包含 Northwind 数据库中的雇员信息，第二个表包含 EmployeeID 列和自动生成的 Name 列，在显示 ComboBox 时要使用这两列：

```
using (SqlConnection con =
    new SqlConnection (
        ConfigurationSettings.ConnectionStrings["northwind"].ConnectionString ) )
{
```

```

string select = "SELECT EmployeeID, FirstName, LastName, Photo,
                IsNull(ReportsTo,0) as ReportsTo FROM Employees";

SqlDataAdapter da = new SqlDataAdapter(select, con);

DataSet ds = new DataSet();

da.Fill(ds, "Employees");

select = "SELECT EmployeeID, FirstName + ' ' + LastName as Name
        FROM Employees UNION SELECT 0, '(None)'";

da = new SqlDataAdapter(select, con);
da.Fill(ds, "Managers");

// Construct the columns in the grid view
SetupColumns(ds);

// Set the default height for a row
dataGridView.RowTemplate.Height = 100 ;

// Then setup the datasource
dataGridView.AutoGenerateColumns = false;
dataGridView.DataSource = ds.Tables["Employees"];
}

```

这里要注意两个地方。第一个 select 语句用 0 替代 ReportsTo 列中的 null 值，数据库中有一行在这个字段中包含了 NULL 值，表示这个人没有上级。但是，在绑定数据时，ComboBox 需要这个列中有值，否则在显示网格时会抛出一个异常。在这个例子中，选择显示 0，因为表中不存在 0——这通常称为 sentinel 值，因为它对应用程序有特殊的含义。

第二个 SQL 子句给 ComboBox 选择数据，包括创建值 Zero 和 (None) 时生成的行。在图 32-12 中，第二行显示了 (None)。

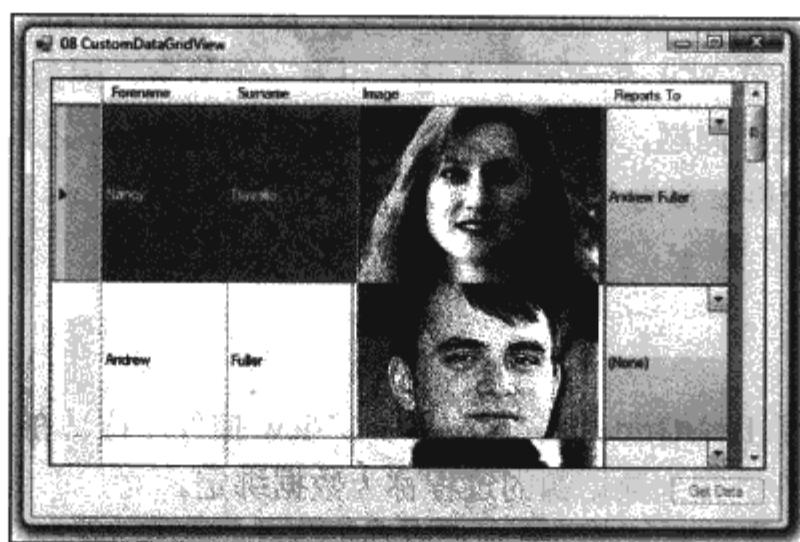


图 32-12

定制的列用下面的函数创建：

```

private void SetupColumns(DataSet ds)
{
    DataGridViewTextBoxColumn forenameColumn = new DataGridViewTextBoxColumn();
    forenameColumn.DataPropertyName = "FirstName";
}

```

```

forenameColumn.HeaderText = "Forename";
forenameColumn.ValueType = typeof(string);
forenameColumn.Frozen = true;
dataGridView.Columns.Add(forenameColumn);

DataGridViewTextBoxColumn surnameColumn = new DataGridViewTextBoxColumn();
surnameColumn.DataPropertyName = "LastName";
surnameColumn.HeaderText = "Surname";
surnameColumn.Frozen = true;
surnameColumn.ValueType = typeof(string);
dataGridView.Columns.Add(surnameColumn);

DataGridViewImageColumn photoColumn = new DataGridViewImageColumn();
photoColumn.DataPropertyName = "Photo";
photoColumn.Width = 100;
photoColumn.HeaderText = "Image";
photoColumn.ReadOnly = true;
photoColumn.ImageLayout = DataGridViewImageCellLayout.Normal;
dataGridView.Columns.Add(photoColumn);

DataGridViewComboBoxColumn reportsToColumn = new DataGridViewComboBoxColumn();
reportsToColumn.HeaderText = "Reports To";
reportsToColumn.DataSource = ds.Tables["Managers"];
reportsToColumn.DisplayMember = "Name";
reportsToColumn.ValueMember = "EmployeeID";
reportsToColumn.DataPropertyName = "ReportsTo";
dataGridView.Columns.Add(reportsToColumn);
}

```

ComboBox 在本例的最后创建, 并使用传送过来的 DataSet 中的 Managers 表用作其数据源。该表包含 Name 和 EmployeeID 列, 它们分别赋予 DisplayMember 和 ValueMember 属性。这两个属性定义了 ComboBox 的数据来自何方。

DataPropertyName 设置为组合框链接的主表中的列, 它提供了列的初始值, 如果用户从组合框中选择另一个值, 就更新这个值。

这个例子还需要在更新数据库时正确处理空值。目前, 如果在屏幕上选择了(None), 该例子仅把值 0 写入数据行。在 SQL Server 中, 这会引发一个异常, 因为这违反了 ReportsTo 列的外键码约束。为了解决这个问题, 需要在把数据发送回 SQL Server 之前, 预先处理它, 即把行中值为 0 的 ReportsTo 列再设置为 NULL。

### 32.3 数据绑定

前面的示例都使用了 DataGrid 控件和 DataGridView 控件, 这是在 .NET 运行库中可以显示数据的两个控件。把控件链接到数据源的过程称为数据绑定。

在 MFC 库中, 把数据从类变量链接到一组控件上的过程称为对话框数据交换(Dialog Data Exchange, DDX)。在 .NET 中, 可用于把数据绑定到控件上的功能更容易使用, 也更强大。例如, 在 .NET 中, 可以把数据绑定到控件的大多数属性上, 而不仅仅是文本属性。还可以用类似的方式把数据绑定到 ASP.NET 控件上(参阅第 37 章)。



32.3.1 简单的绑定

支持单一绑定的控件一般一次只显示一个值，例如文本框或单选按钮，下面的示例说明了如何把 DataTable 中的一列绑定到 TextBox 上：

```
DataSet ds = CreateDataSet();
textBox1.DataBindings.Add("Text", ds, "Products.ProductName");
```

用上面的方法 CreateDataSet()从 Products 表中检索一些数据，并保存到返回的 DataSet 中，之后，第二行代码把控件(textBox1)的 Text 属性绑定到 Products.ProductName 列上。图 32-13 显示了这种数据绑定的结果。

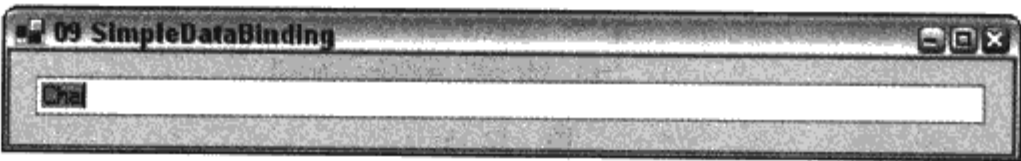


图 32-13

文本框显示了数据库中的一个字符串，要检查这个数据是否放在正确的列上并有正确的值，可以使用 SQL Server Management Studio 工具，验证 Products 表中的内容，如图 32-14 所示。

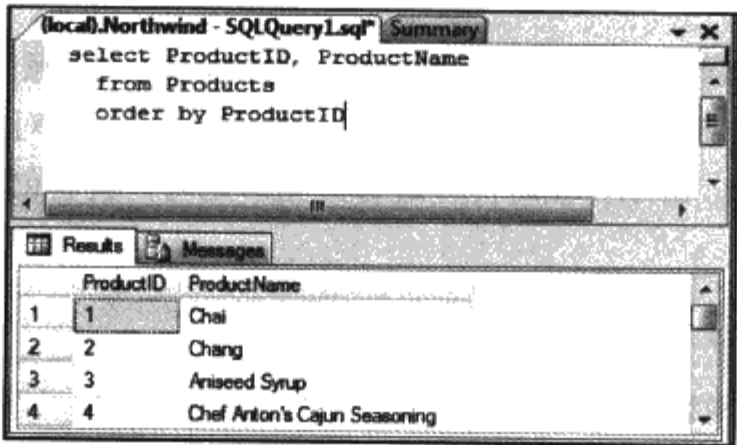


图 32-14

屏幕上有一个文本框，既不能滚动到下一个或上一个记录上，也不能更新数据库，因此下一节介绍一个更真实的示例，说明如何使用其他对象绑定数据。

32.3.2 数据绑定对象

图 32-15 显示了数据绑定中使用的对象的类层次结构。本节将讨论 System.Windows.Forms 命名空间中的类 BindingContext、CurrencyManager 和 PropertyManager，说明在把数据绑定到窗体上的一个或多个控件上时，它们是如何交互的。带阴影的对象就是在绑定中使用的对象。

在前面的示例中，我们使用 TextBox 控件的 DataBindings 属性把 DataSet 的一列绑定到控件的 Text 属性上，DataBindings 属性是图 32-15 所示的 ControlsBindingsCollection 的一个实例。



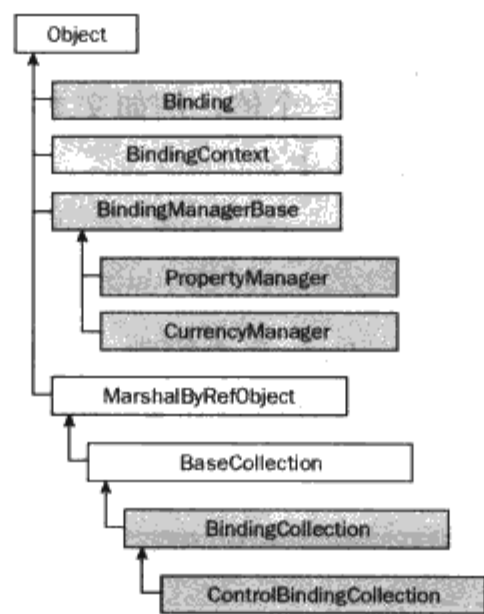


图 32-15

```
textBox1.DataBindings.Add("Text", ds, "Products.ProductName");
```

这行代码给 ControlBindingsCollection 添加一个 Binding 对象。

1. BindingContext

每个 Windows 窗体都有 BindingContext 属性，实际上，Form 派生自 Control，该属性是在 Control 中定义的，所以大多数控件都有这个属性。BindingContext 对象有一个 BindingManagerBase 实例集合，如图 32-16 所示。在对控件进行数据绑定时，就会创建这些实例，并把它们添加到绑定管理器对象中。

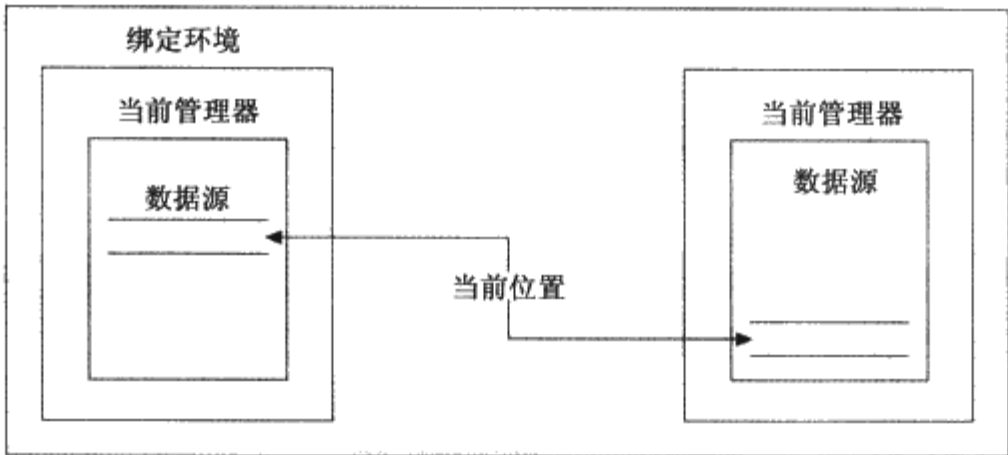


图 32-16

BindingContext 可以包含几个数据源，包装在 CurrencyManager 或 PropertyManager 中。使用哪个类取决于数据源本身。

如果数据源包含一个项目列表，例如 DataTable、DataView 或实现 IList 接口的对象，就使用 CurrencyManager，因为它可以在该数据源中保存当前位置。如果数据源只返回一个值，就把 PropertyManager 存储在 BindingContext 中。

只为给定的数据源创建一次 CurrencyManager 或 PropertyManager。如果把两个文本框绑定到 DataTable 的一个行上，则在 BindingContext 中只创建一个 CurrencyManager。

添加到窗体中的每个控件都链接到窗体的绑定管理器上，因此所有的控件都共享相同的实

例。在最初创建一个控件时，其 `BindingContext` 属性为空。在把控件添加到窗体的 `Controls` 集合中时，就把 `BindingContext` 设置为该窗体的 `Controls` 集合。

要把控件绑定到一个列上，需要给其 `DataBindings` 属性添加一项，这是 `ControlBindingsCollection` 的一个实例。下面的代码可以创建一个新绑定：

```
textBox1.DataBindings.Add("Text", ds, "Products.ProductName");
```

`ControlBindingsCollection` 的 `Add()`方法会从传递给它的参数中创建 `Binding` 对象的一个实例，并把它添加到绑定集合中，如图 32-17 所示。

图 32-17 显示了把 `Binding` 实例添加到控件中的情况。绑定把控件链接到数据源上，存储在 `Form`(或控件本身)的 `BindingContext` 中。数据源内部的改变会反映到控件上，控件中的改变也会反映到数据源上。

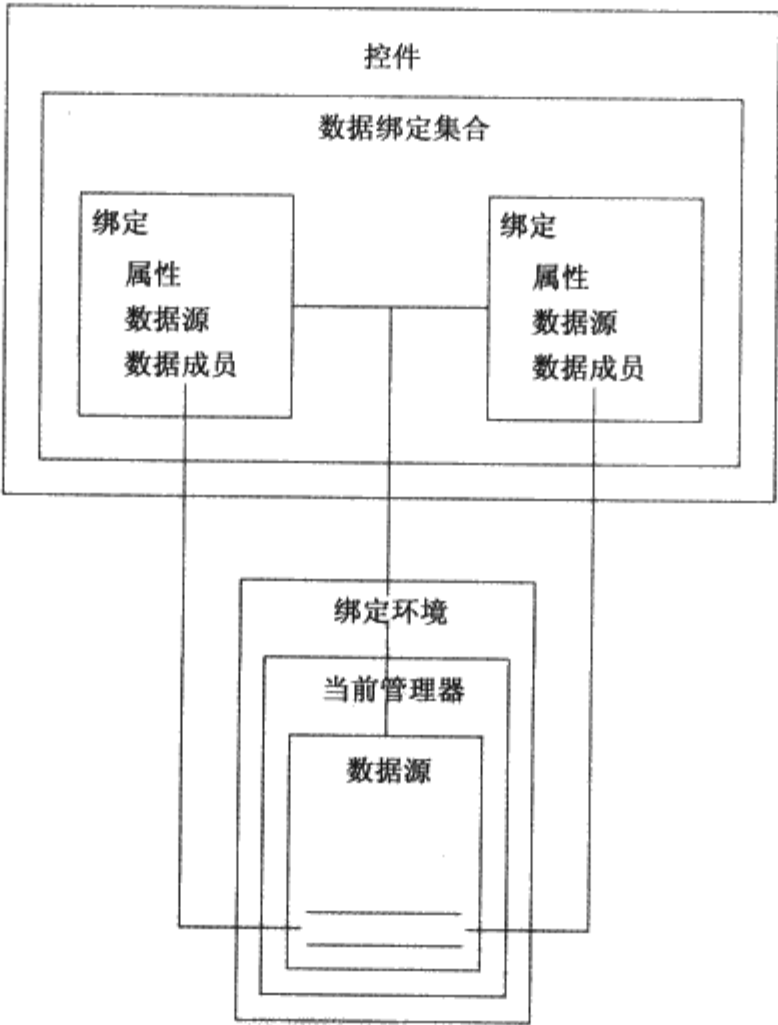


图 32-17

2. Binding 类

这个类把控件的一个属性链接到数据源的一个成员上。在改变该成员时，控件的属性会更新，以反映这个改变。反之亦然，如果文本框中的文本被更新，这个改变也会反映到数据源上。

可以把任何列绑定到控件的任何属性上，例如，可以把列绑定到一个文本框的 `Text` 属性中，也可以把另一个列绑定到文本框的 `Color` 属性上。可以把控件的属性绑定到完全不同的数据源上，例如，单元格的颜色在一个颜色表中定义，而实际的数据在另一个表中定义。

### 3. CurrencyManager 和 PropertyManager

在创建 Binding 对象时，如果这是第一次绑定数据源中的数据，就会创建对应的 CurrencyManager 或 PropertyManager 对象。这个类的作用是定义当前记录在数据源中的位置，在改变当前的记录时，需要调整所有的 ListBindings。图 32-18 显示了 Products 表中的两个字段，包含一种通过跟踪栏控件在记录之间移动的方式。

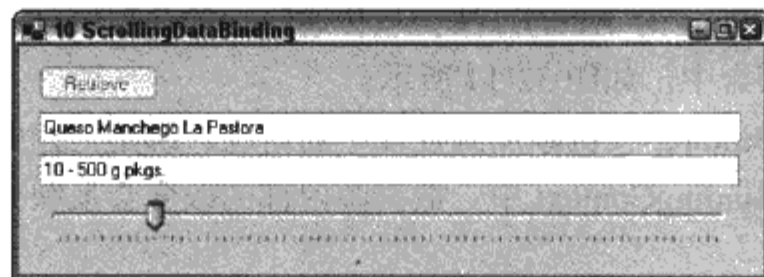


图 32-18

ScrollingDataBinding 示例的代码如下所示：

```
namespace ScrollingDataBinding
{
    partial class Form1: Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private DataSet CreateDataSet()
        {
            string customers = "SELECT * FROM Products";
            DataSet ds = new DataSet();

            using (SqlConnection con = new SqlConnection (
                ConfigurationSettings.
                ConnectionStrings["northwind"].ConnectionString))
            {
                SqlDataAdapter da = new SqlDataAdapter(customers, con);

                da.Fill(ds, "Products");
            }

            return ds;
        }

        private void trackBar_Scroll(object sender, EventArgs e)
        {
            this.BindingContext[ds, "Products"].Position = trackBar.Value;
        }

        private void retrieveButton_Click(object sender, EventArgs e)
        {
            retrieveButton.Enabled = false;

            ds = CreateDataSet();

            textName.DataBindings.Add("Text", ds, "Products.ProductName");
            textQuan.DataBindings.Add("Text", ds, "Products.QuantityPerUnit");
        }
    }
}
```

```

        trackBar.Minimum = 0;
        trackBar.Maximum = this.BindingContext[ds, "Products"].Count - 1;

        textName.Enabled = true;
        textQuan.Enabled = true;
        trackBar.Enabled = true;
    }

    private DataSet ds;
}

```

滚动机制在 `trackBar_Scroll` 事件处理程序中提供，该处理程序把 `BindingContext` 的位置设置为跟踪条的当前位置，改变 `BindingContext` 会更新显示在屏幕中的数据。

在 `retrieveButton_Click` 事件中添加一个数据绑定表达式，就把数据绑定到两个文本框上，这里控件的 `Text` 属性设置为数据源中的字段。可以把控件的任意简单数据绑定到数据源的一项上，例如，可以绑定 `text color`、`enabled` 等属性。

在开始检索数据时，跟踪栏的最大位置就设置为记录的个数。接着在上面的滚动方法中，把 `Products` 数据表中 `BindingContext` 的位置设置为滚动条的位置，这样就可以有效地改变 `DataTable` 中的当前记录，绑定到当前行上的所有控件(在本例中是两个文本框)就会被更新。

本节介绍了如何绑定到各种数据源(例如数组、数据表、数据视图和各种其他数据容器)上，如何排序和过滤数据。下一节将讨论如何扩展 Visual Studio，以允许进行数据访问，得到与应用程序的更好集成。

## 32.4 Visual Studio 和数据访问

本节讨论 Visual Studio 把数据集成到 GUI 中的一些新方式，具体地说，就是讨论如何创建连接，选择一些数据，生成一个 `DataSet`，再使用生成的所有对象创建一个简单的应用程序。Visual Studio 提供的工具可以使用 `OleDbConnection` 或 `SqlConnection` 类创建一个数据库连接，使用哪个类取决于要连接的数据库类型。定义了一个连接后，就可以创建一个 `DataSet`，在 Visual Studio 中给它填充数据，这会为 `DataSet` 生成一个 XSD 文件和.cs 代码，最终会创建一个类型安全的 `DataSet`。

### 32.4.1 创建一个连接

首先，创建一个新的 Windows 应用程序，之后创建一个数据库连接。使用 Server Explorer，可以管理数据访问的各个方面，如图 32-19 所示。

在本例中，需要创建 Northwind 数据库的一个连接。从 `Data Connections` 项的弹出菜单中选择 `Add Connection` 选项，会打开一个向导，该向导允许选择数据库提供程序。这里选择 .NET Framework Provider for SQL Server。Add Connection 对话框如图 32-20 所示。

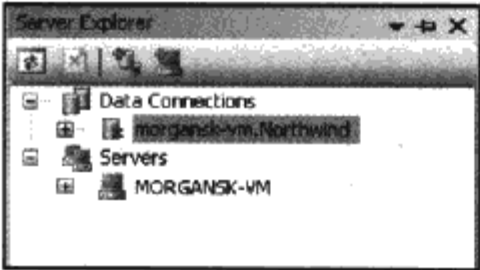


图 32-19

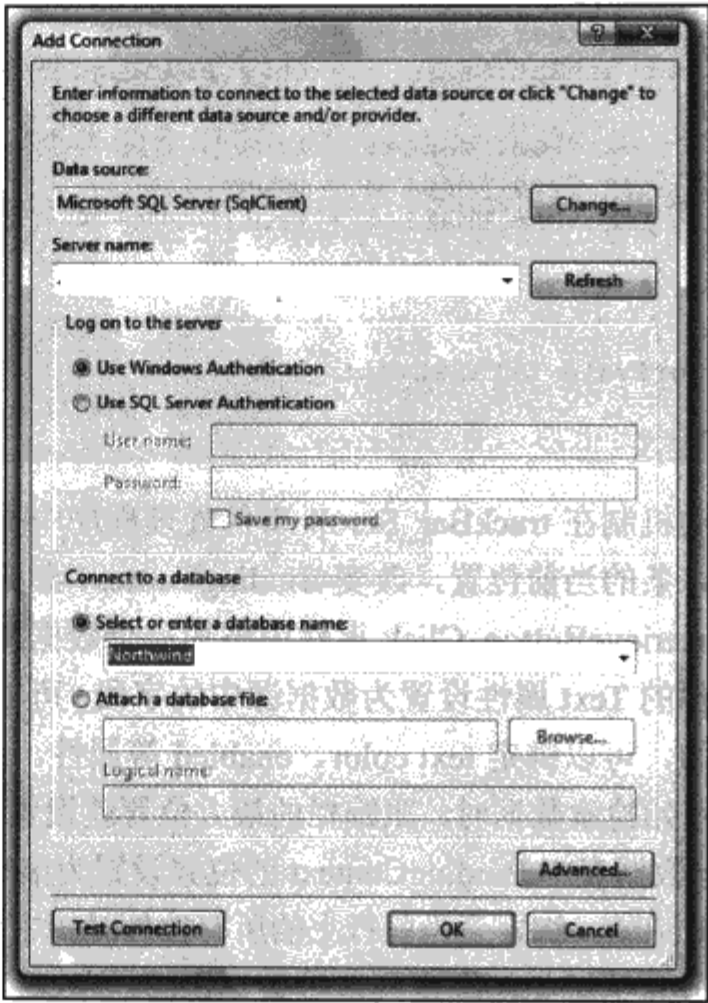


图 32-20

根据.NET Framework 安装，示例数据库可能位于 SQL Server、MSDE(Microsoft SQL Server Data Engine)，或者在这两个地方都有。

要连接本地 MSDE 数据库(如果有)，键入(Local)\NETSDK 作为服务器的名称。要连接一般的 SQL Server 实例，应键入(local)或 ‘.’，选择当前机器上的一个数据库，或者选择网络上需要的服务器名称。还需要输入用户名和密码来访问数据库。

从数据库的下拉列表中选择 Northwind 数据库，为了确保正确安装了所有的文件，单击 Test Connection 按钮，如果一切安装正确，就应显示一个包含确认信息的信息框。

Visual Studio 2005 在数据访问方面有许多变化，这些在用户界面的几个地方可以看出。我喜欢使用新的 Data 菜单，它可以查看已添加到项目中的数据源，添加新数据源，预览底层数据库(或其他数据源)中的数据。

下面的例子使用 Northwind 数据库连接生成一个用户界面，用于从 Employees 表中选择数据。第一步是从 Data 菜单中选择 Add New Data Source，打开一个向导，利用该向导完成后面的步骤。图 32-21 显示了 Data Sources Configuration 向导的一部分，从中可以为数据源选择合适的表。



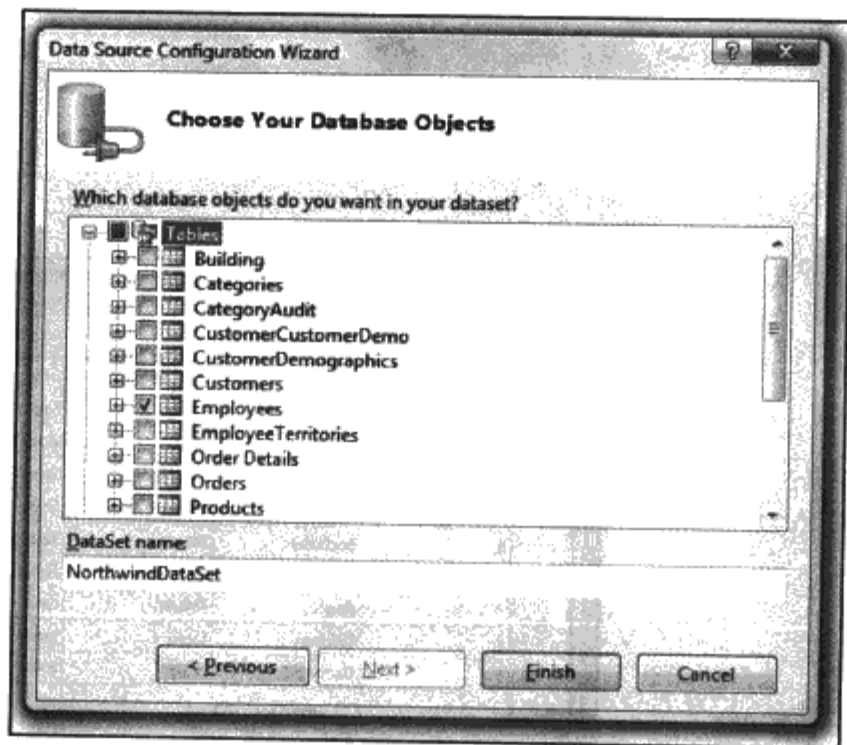


图 32-21

在向导中，可以选择数据源，它可以是数据库、本地数据库文件(例如.mdb 文件)、Web 服务或一个对象。接着向导要求用户根据所选择的数据源类型提供更详细的信息。对于数据库连接，要提供的信息有连接名称(它随后存储在应用程序配置文件中，如下面的代码所示)，接着选择提供数据的表、视图或存储过程。最后，这会在应用程序中生成一个强类型化的 DataSet。

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <connectionStrings>
    <add name="SimpleApp.Properties.Settings.NorthwindConnection"
      connectionString="Data Source=.;Integrated Security=True;
        Initial Catalog=Northwind"
      providerName="System.Data.SqlClient" />
  </connectionStrings>
</configuration>
```

其中包括连接的名称、连接字符串和提供程序的名称(在生成连接对象时使用)。可以根据需要手工编辑这些信息。要给雇员数据显示一个用户界面，可以把选中的数据从 Data Sources 窗口拖放到窗体上。这会生成两种 User Interface 样式中的一种：利用前面介绍的 DataGridView 控件的网格样式 UI，或者一次只显示一个记录的 Details 视图。图 32-22 显示了 Details 视图。

把数据源拖放到窗体上会生成许多对象，它们有的可见，有的不可见。不可见的对象在窗体的组件区域中创建，包含 DataConnector、强类型化的 DataSet 和 TableAdapter(它包含用于选择或更新数据的 SQL)。可见的对象根据是选择 DataGridView 还是 Details 视图来创建。无论选择 DataGridView 还是 Details 视图，都会创建一个 DataNavigator 控件，用于给数据分页显示。图 32-23 显示了使用 DataGridView 控件生成的用户界面——Visual Studio 2008 的一个目标是简化数据访问，即不编写任何代码，就能生成功能全面的窗体。

在创建数据源时，要在解决方案中添加许多文件，要查看它们，需要单击 Solution Explorer 中的 Show All Files 按钮，然后就可以展开数据集节点，查看添加的文件。其中比较有趣的是.Designer.cs 文件，它包含用于填充数据集的 C#源代码。

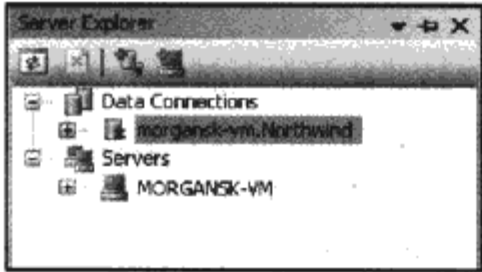


图 32-19

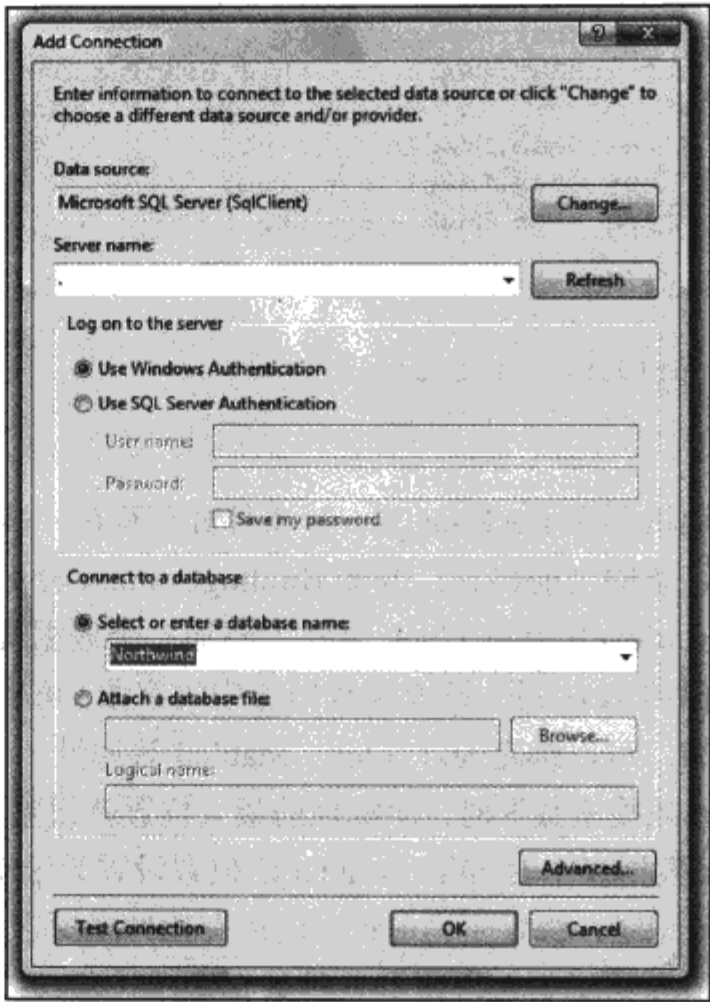


图 32-20

根据.NET Framework 安装，示例数据库可能位于 SQL Server、MSDE(Microsoft SQL Server Data Engine)，或者在这两个地方都有。

要连接本地 MSDE 数据库(如果有)，键入(Local)\NETSDK 作为服务器的名称。要连接一般的 SQL Server 实例，应键入(local)或 ‘.’，选择当前机器上的一个数据库，或者选择网络上需要的服务器名称。还需要输入用户名和密码来访问数据库。

从数据库的下拉列表中选择 Northwind 数据库，为了确保正确安装了所有的文件，单击 Test Connection 按钮，如果一切安装正确，就应显示一个包含确认信息的信息框。

Visual Studio 2005 在数据访问方面有许多变化，这些在用户界面的几个地方可以看出。我喜欢使用新的 Data 菜单，它可以查看已添加到项目中的数据源，添加新数据源，预览底层数据库(或其他数据源)中的数据。

下面的例子使用 Northwind 数据库连接生成一个用户界面，用于从 Employees 表中选择数据。第一步是从 Data 菜单中选择 Add New Data Source，打开一个向导，利用该向导完成后面的步骤。图 32-21 显示了 Data Sources Configuration 向导的一部分，从中可以为数据源选择合适的表。

上的 Save 按钮，编写一个事件处理程序，更新数据库。

在 IDE 上，从数据导航控件中选择 Save 按钮，把 Enabled 属性改为 true。然后双击按钮，生成一个事件处理程序。在这个事件处理程序中，把屏幕上数据的改变返回至数据库：

```
private void dataNavigatorSaveItem_Click(object sender, EventArgs e)
{
    employeesTableAdapter.Update(employeesDataset.Employees);
}
```

Visual Studio 完成了大部分工作，我们只需使用表适配器类的 Update 方法。表适配器有 6 个 Update 方法，这个例子使用把 DataTable 作为参数的重写版本。

#### 32.4.4 其他常见的要求

在显示数据时，一个常见的要求是为给定的行提供一个弹出式菜单。这有很多解决方式，本例介绍的一种方式可以简化需要的代码，特别适合于下述情况：显示环境是 DataGrid，在该环境中显示了带有一些关系的 DataSet。问题是弹出的菜单与所选的行有关，该行可以来自于该 DataSet 中的任意一个源数据表。

因为弹出菜单的功能似乎很一般，所以这里利用一个支持建立菜单的基类(ContextDataRow)，每个数据行类都支持从这个基类派生来的一个弹出菜单。

用户右击 DataGrid 中一行的任何一部分，都可以查看该行，确定它是否派生于 ContextDataRow，如果是，就可以调用 PopupMenu()函数。这可以使用一个接口来实现，但在本例中，使用基类会比较简单。

本例说明了如何生成 DataRow 和 DataTable 类，以便对数据进行类型安全的访问，这与前面 XSD 示例采用的方式相同。但这次代码是手工编写的，来说明在这种情况下如何使用定制属性和反射。

图 32-24 显示了这个示例的类层次结构。

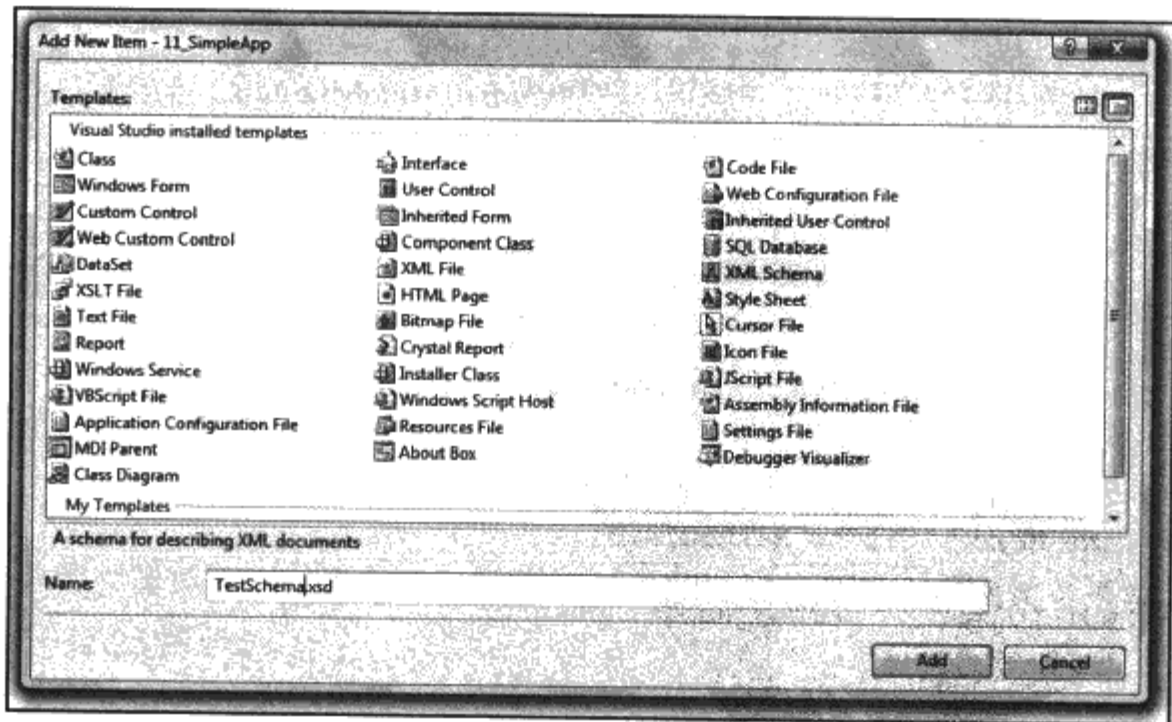


图 32-24

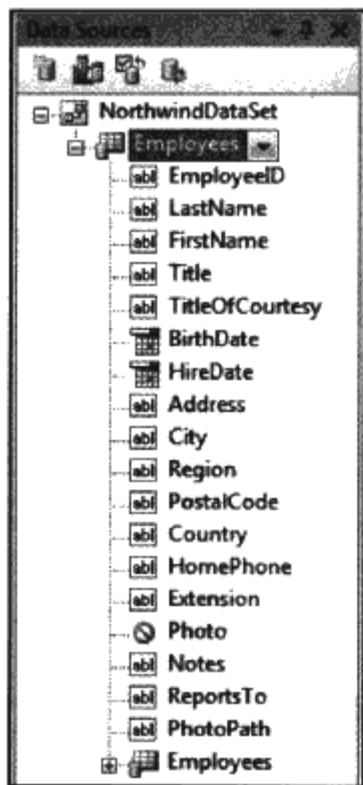


图 32-22

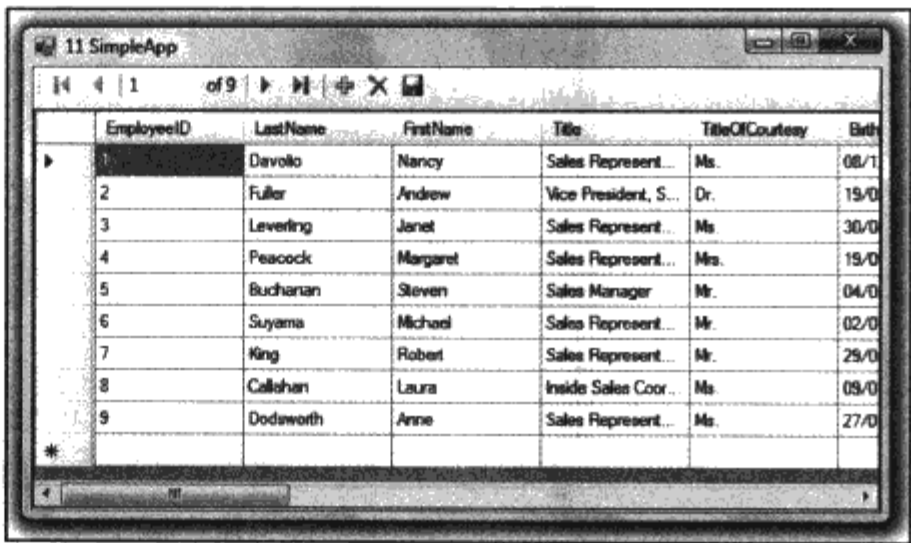


图 32-23

在.Designer.cs 文件中定义了几个类，它们表示强类型化的数据集，其操作方式类似于标准的 DataAdapter 类。这个类在内部使用 DataAdapter 填充 DataSet。

32.4.2 选择数据

生成的表数据适配器包含 SELECT、INSERT、UPDATE 和 DELETE 命令。显然，也可以调用存储过程，而不直接使用 SQL 命令。向导生成的代码也可以完成这个工作。Visual Studio 给 .Designer 文件添加如下代码：

```
private System.Data.SqlClient.SqlCommand m_DeleteCommand;
private System.Data.SqlClient.SqlCommand m_InsertCommand;
private System.Data.SqlClient.SqlCommand m_UpdateCommand;
private System.Data.SqlClient.SqlDataAdapter m_adapter;
```

除了 Select 命令之外，为每个 SQL 命令定义一个对象和一个 SqlDataAdapter。在文件后面的 InitializeComponent()方法中，向导生成了创建这些命令和数据适配器的代码。

在 Visual Studio 的以前版本中，为 Insert 和 Update 生成的命令也包含一个 select 子句，以便使数据与服务器上的数据再次同步。其实只是计算数据库中的所有字段(例如标识列和计算字段)。

向导生成的代码可以工作，但有时它并不是最佳的。对于产品系统，所有生成的 SQL 语句都应用存储过程的调用来替换。如果 INSERT 或 UPDATE 子句不需要重新同步数据，删除多余的 SQL 子句可以加速应用程序的运行。

32.4.3 更新数据源

应用程序已经从数据库中选择了数据，本节将说明把数据的改变返回至数据库。如果您按照上一节的步骤进行，就应得到一个包含所有必要对象的应用程序。剩下的就只是激活工具栏

```

        if (parms.Length == 0)
        {
            // Lastly check if there is a ContextMenuAttribute on the
            // method...

            object[] atts = meth.GetCustomAttributes
                (typeof(ContextMenuAttribute), true);
            bInclude = (atts.Length == 1);
        }
    }
    return bInclude;
}
}

```

`ContextDataRow` 类派生自 `DataRow`，且该类只包含两个成员函数。第一个函数 `PopupMenu` 使用反射来查找对应于特定签名的方法，它还给用户显示这些选项的一个弹出菜单。第二个函数 `Filter()` 在枚举方法时由 `PopupMenu` 用作一个委托。如果成员函数符合相应的调用约定，它就只返回 `true`。

```

MemberInfo[] members = this.GetType().FindMembers(MemberTypes.Method,
    BindingFlags.Public | BindingFlags.Instance,
    new System.Reflection.MemberFilter(Filter),
    null);

```

这个语句用于过滤当前对象上的所有方法，仅返回与下述条件相匹配的方法：

- 成员必须是一个方法
- 成员必须是一个公共实例方法
- 成员必须没有返回值
- 成员必须不带参数
- 成员必须包含 `ContextMenuAttribute`

最后一个条件是一个定制属性，是专门为这个示例编写的。在完成 `PopupMenu` 方法的分析后讨论这个定制属性：

```

ContextMenu menu = new ContextMenu();
foreach (MethodInfo meth in members)
{
    // ... Add the menu item
}
System.Drawing.Point pt = new System.Drawing.Point(x,y);
menu.Show(parent, pt);

```

创建一个关联菜单实例，为每个满足上述条件的方法添加一个弹出菜单选项。接着，显示该菜单，如图 32-25 所示。

这个示例中的主要困难是下面这段代码，在弹出的菜单中，每个要显示的成员函数都要重复一次该段代码：

```

System.Type ctxtype = typeof(ContextMenuAttribute);
ContextMenuAttribute[] ctx = (ContextMenuAttribute[])
    meth.GetCustomAttributes(ctxtype, true);
MenuCommand callback = new MenuCommand(this, meth);
MenuItem item = new MenuItem(ctx[0].Caption,
    new EventHandler(callback.Execute));

```



这个示例的代码如下：

```
using System;
using System.Windows.Forms;
using System.Data;
using System.Data.SqlClient;
using System.Reflection;

public class ContextDataRow : DataRow
{
    public ContextDataRow(DataRowBuilder builder) : base(builder)
    {
    }
    public void PopupMenu(System.Windows.Forms.Control parent, int x, int y)
    {
        // Use reflection to get the list of popup menu commands
        MemberInfo[] members = this.GetType().FindMembers (MemberTypes.Method,
            BindingFlags.Public | BindingFlags.Instance ,
            new System.Reflection.MemberFilter(Filter),
            null);
        if (members.Length > 0)
        {
            // Create a context menu
            ContextMenu menu = new ContextMenu();

            // Now loop through those members and generate the popup menu
            // Note the cast to MethodInfo in the foreach
            foreach (MethodInfo meth in members)
            {
                // Get the caption for the operation from the
                // ContextMenuAttribute
                ContextMenuAttribute[] ctx = (ContextMenuAttribute[])
                    meth.GetCustomAttributes(typeof(ContextMenuAttribute), true);
                MenuCommand callback = new MenuCommand(this, meth);
                MenuItem item = new MenuItem(ctx[0].Caption, new
                    EventHandler(callback.Execute));
                item.DefaultItem = ctx[0].Default;
                menu.MenuItems.Add(item);
            }
            System.Drawing.Point pt = new System.Drawing.Point(x,y);
            menu.Show(parent, pt);
        }
    }

    private bool Filter(MemberInfo member, object criteria)
    {
        bool bInclude = false;

        // Cast MemberInfo to MethodInfo
        MethodInfo meth = member as MethodInfo;
        if (meth != null)
            if (meth.ReturnType == typeof(void))
            {
                ParameterInfo[] parms = meth.GetParameters();
            }
        }
    }
}
```

然后执行最终显示在关联菜单中的方法：

```
public class CustomerRow : ContextDataRow
{
    public CustomerRow(DataRowBuilder builder) : base(builder)
    {
    }
    public string CustomerID
    {
        get { return (string)this["CustomerID"]; }
        set { this["CustomerID"] = value; }
    }

    // Other properties omitted for clarity

    [ContextMenu("Blacklist Customer")]
    public void Blacklist()
    {
        // Do something
    }
    [ContextMenu("Get Contact", Default=true)]
    public void GetContact()
    {
        // Do something else
    }
}
```

该类派生于 `ContextDataRow`，包括相应的 `getter/setter` 方法，其属性名与每个字段是相同的，然后添加一些方法，在反射到类上时会使用这些方法：

```
[ContextMenu("Blacklist Customer")]
public void Blacklist()
{
    // Do something
}
```

每个要显示在关联菜单中的方法都有相同的签名，且包括定制的 `ContextMenu` 属性。

## 2. 使用属性

编写 `ContextMenu` 属性的理念是要能为给定的菜单选项提供一个自由文本名称，下面的示例还使用了一个 `Default` 标志，它用于指定默认的菜单选项。下面给出完整的属性类：

```
[AttributeUsage(AttributeTargets.Method, AllowMultiple=false, Inherited=true)]
public class ContextMenuAttribute : System.Attribute
{
    public ContextMenuAttribute(string caption)
    {
        Caption = caption;
        Default = false;
    }
    public readonly string Caption;
}
```

类的 `AttributeUsage` 属性把 `ContextMenuAttribute` 标记为唯一可以在方法上使用的属性，它还定义了在任何给定的方法上只有该对象的一个实例。`Inherited=true` 子句定义了属性是否可以放在一个超类方法上，且仍由子类反映出来。

```
item.DefaultItem = ctx[0].Default;
menu.MenuItems.Add(item);
```

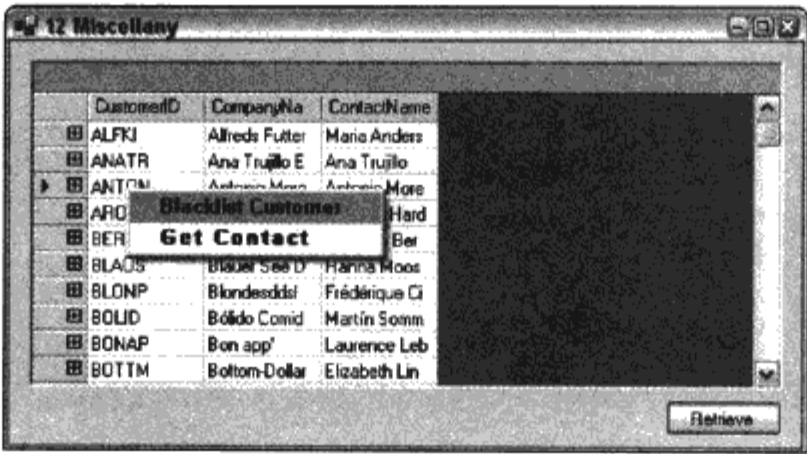


图 32-25

在弹出菜单中显示的每个方法都有一个 ContextMenuAttribute 属性。它为菜单选项定义了一个用户友好的名称，因为 C#方法名不能有空格，所以在弹出菜单中使用英语要比使用某些内部代码更好。该属性从方法中获取，创建一个新菜单项，并把它添加到弹出式菜单的菜单项集合中。

这个示例代码还说明了简化的 Command 类的用法(一种常见的设计模式)。这个示例中使用的 MenuCommand 是在用户选择关联菜单中的一项时触发的，它会把调用传送给方法的接收器，在本例中是对象及所属的方法。这也有助于使接收器对象中的代码更容易与用户界面代码隔离开来。下面几节将解释这些代码。

1. 生成的表和行

本章前面的 XSD 示例显示的代码是在使用 Visual Studio 编辑器生成一组数据访问类时生成的。下面的类显示了 DataTable 需要的方法，这是为这个类生成的最小一组代码(它们都是手工生成的):

```
public class CustomerTable : DataTable
{
    public CustomerTable() : base("Customers")
    {
        this.Columns.Add("CustomerID", typeof(string));
        this.Columns.Add("CompanyName", typeof(string));
        this.Columns.Add("ContactName", typeof(string));
    }
    protected override System.Type GetRowType()
    {
        return typeof(CustomerRow);
    }
    protected override DataRow NewRowFromBuilder(DataRowBuilder builder)
    {
        return (DataRow) new CustomerRow(builder);
    }
}
```

首先，DataTable 必须重写 GetRowType()方法，这是在生成表的新行时由.NET 在内部使用的。这个方法应返回用于表示每一行的类型。

其次，需要执行 NewRowFromBuilder()，它也是在为表创建新行时由运行库调用的。这对于最小执行方式是足够的。对应的 CustomerRow 类是相当简单的，它为行中的每一列执行属性，

#### 4. 获得选中的行

这个示例的最后一个问题是如何处理 DataSet 中用户选择的行。首先考虑“它必须是 DataGrid 的一个属性”，但在 DataGrid 上不能使用这个属性。查看一下从 MouseUp() 事件处理程序中获得的测试信息，但只有在显示一个 DataTable 中的数据时，这才有一定的帮助。

下面介绍如何填充网格：

```
dataGrid.SetDataBinding(ds, "Customers");
```

这个方法给 BindingContext 添加一个新的 CurrencyManager，它表示当前数据表和 DataSet。现在，DataGrid 有两个属性 DataSource 和 DataMember，在调用 SetDataBinding() 时设置它们。本例中的 DataSource 是一个 DataSet，DataMember 是 Customers。

给定数据源和一个数据成员，以及窗体的 BindingContext，就可以使用下面的代码查找当前行：

```
protected void dataGrid_MouseUp(object sender, MouseEventArgs e)
{
    // Perform a hit test
    if(e.Button == MouseButton.Right)
    {
        // Find which row the user clicked on, if any
        DataGrid.HitTestInfo hti = dataGrid.HitTest(e.X, e.Y);

        // Check if the user hit a cell
        if(hti.Type == DataGrid.HitTestType.Cell)
        {
            // Find the DataRow that corresponds to the cell
            //the user has clicked upon

```

在调用 dataGrid.HitTest() 确定用户在什么地方单击后，就要为 DataGrid 检索 BindingManagerBase 实例了：

```
BindingManagerBase bmb = this.BindingContext[ dataGrid.DataSource,
                                                dataGrid.DataMember];
```

它使用了 DataGrid 的 DataSource 和 DataMember，给要返回的对象命名。现在只需查找用户单击的行，显示关联菜单。右击了一个行后，当前行的指示符一般不会移动，但这还不够，还要移动行的指示符，再显示弹出菜单。HitTestInfo 对象中有行号，所以只需移动 BindingManagerBase 对象的当前位置：

```
bmb.Position = hti.Row;
```

这会改变单元格的指示符，同时意味着在调用类来获取 Row 时，将返回当前行，而不是上次选择的行：

```
DataRowView drv = bmb.Current as DataRowView;
if(drv != null)
{
    ContextDataRow ctx = drv.Row as ContextDataRow;
    if(ctx != null) ctx.PopupMenu(dataGrid, e.X, e.Y);
}
}
```



可以把许多其他成员添加到这个属性上，例如：

- 菜单选项的热键
- 要显示的图像
- 要显示在工具栏中的文本，当鼠标放在菜单选项上时，就会显示该文本
- 帮助上下文 ID

3. 分派方法

当菜单显示在.NET 中时，每个菜单选项都通过委托的方式链接到该选项的处理代码上。把菜单选项链接到代码上的机制中，有两个选项：

- 执行与 System.EventHandler 有相同签名的方法，其定义如下：

```
public delegate void EventHandler(object sender, EventArgs e);
```

- 定义一个代理类，它执行上述委托，把调用传送给接收的类。这称为 Command 模型，也是本例选择的模型。

Command 模型通过一个简单的中间类把调用的发送者和接收者分离开来。对于本例，该模型就矫枉过正了，但可以使每个 DataRow 上的方法更简单一些(因为它们不需要给委托传送参数)，而且可扩展性更高：

```
public class MenuCommand
{
    public MenuCommand(object receiver, MethodInfo method)
    {
        Receiver = receiver;
        Method = method;
    }
    public void Execute(object sender, EventArgs e)
    {
        Method.Invoke(Receiver, new object[] { } );
    }
    public readonly object Receiver;
    public readonly MethodInfo Method;
}
```

该类只提供了一个 EventHandler 委托(Execute 方法)，它在接收器对象上调用所希望的方法。本例显示了两两种不同类型的行—— Customers 表中的行和 Orders 表中的行。自然，这些类型的数据的处理选项都是不同的。图 32-25 显示了可以应用到 Customer 行上的操作。图 32-26 显示了可以应用到 Order 行上的操作。

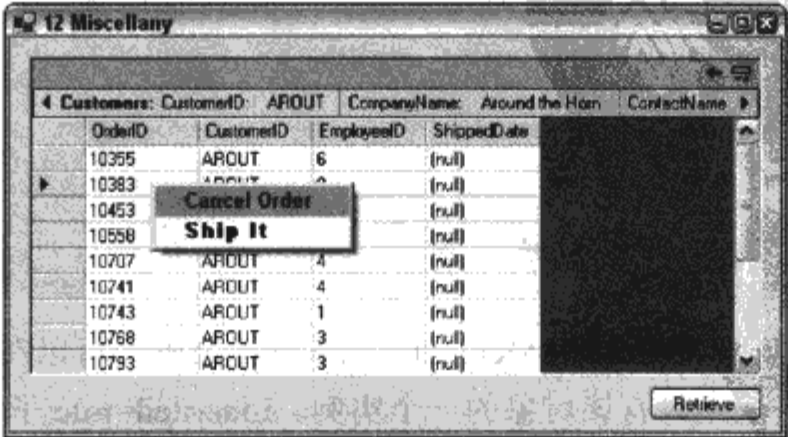


图 32-26



# 第33章

## 使用 GDI+ 绘图

在本书中，有 8 章内容介绍用户交互和 .NET Framework，第 31 章主要介绍了如何显示对话框或 SDI、MDI 窗口，以及如何把各种控件放在这些窗口上，如按钮、文本框和列表框。第 32 章介绍在 Windows 窗体中使用许多 Windows 窗体控件处理各种数据源中的数据。

这些标准控件的功能非常强大，使用它们就可以获得许多应用程序的完整用户界面。但是，有时还需要在用户界面上有更大的灵活性。例如，要在窗口的确定位置以给定的字体绘制文本，或者显示图像，但不使用图像框控件，只使用形状和图形。这些都不能使用第 31 章介绍的控件来完成。要显示这种类型的输出，应用程序必须直接告诉操作系统需要在其窗口的什么地方显示什么内容。

本章主要介绍如何绘制以下内容：

- 绘图规则
- 直线、简单图形
- .BMP 图像和其他图像文件
- 文本
- 处理打印

在这个过程中，还需要使用各种帮助对象，包括钢笔(用于定义直线的特性)、画笔(用于定义区域的填充方式)和字体(用于定义文本字符的图形)。我们还将介绍设备如何解释和显示不同的颜色。

下面首先讨论 GDI+ 技术。GDI+ 由 .NET 基类集组成，这些基类可用于在屏幕上完成定制绘图，能把合适的指令发送到图形设备的驱动程序上，确保在监视器屏幕上显示正确的输出(或打印到硬拷贝中)。

### 33.1 理解绘图规则

本节讨论一些基本规则，只有理解了它们，才能开始在屏幕上绘图。首先概述 GDI，GDI+ 技术就建立在 GDI 上，然后说明它与 GDI+ 的关系。接着介绍几个简单的例子。

#### 33.1.1 GDI 和 GDI+

一般来说，Windows 的一个优点(实际上是现代操作系统的优点)是它可以让开发人员不考虑特定设备的细节。例如，不需要理解硬盘设备驱动程序，只需在相关的 .NET 类中调用合适

的方法(在.NET 推出之前,使用等价的 Windows API 函数),就可以编程读写磁盘上的文件。这个规则也适用于绘图。计算机在屏幕上绘图时,把指令发送给视频卡。问题是市面上有几百种不同的视频卡,大多数有不同的指令集和功能。如果把这个考虑在内,在应用程序中为每个视频卡驱动程序编写在屏幕上绘图的特定代码,这样的应用程序就根本不可能编写出来。这就是为什么在 Windows 最早期的版本中就有 Windows Graphical Device Interface (GDI)的原因。

GDI+提供了一个抽象层,隐藏了不同视频卡之间的区别,这样就可以调用 Windows API 函数完成指定的任务了,GDI 会在内部指出在运行特定的代码时,如何让客户机的视频卡完成要绘制的图形。GDI 还可以完成其他任务。大多数计算机都有多个显示设备——例如,监视器和打印机。GDI 成功地使应用程序所使用的打印机看起来与屏幕一样。如果要打印某些东西,而不是显示它们,只需告诉系统输出的设备是打印机,再用相同的方式调用相同的 Windows API 函数即可。

可以看出,DC(设备环境)是一个功能非常强大的对象,在 GDI 下,所有的绘图工作都必须通过设备环境来完成。DC 甚至可用于不涉及在屏幕或其他硬件设备上绘图的其他操作,例如在内存中修改图像。

GDI 给开发人员提供了一个相当高级的 API,但它仍是一个基于旧的 Windows API 并且有 C 语言风格函数的 API,所以使用起来不是很方便。GDI+在很大程度上是 GDI 和应用程序之间的一层,提供了更直观、基于继承性的对象模型。尽管 GDI+基本上是 GDI 的一个包装器,但 Microsoft 已经能通过 GDI+提供新功能了,它还有一些性能方面的改进。

.NET 基类库的 GDI+部分非常大,本章不解释其特性。这是因为只要解释其中的几个类、方法和属性,就会把本章变成一个仅列出 GDI+类和方法的参考指南。而理解绘图的基本规则更重要;这样您应可以自己研究这些类。当然,关于 GDI+中类和方法的完整列表,可以参阅 SDK 文档说明。

注意:

有 VB6 背景的开发人员会发现,自己并不熟悉绘图过程涉及的概念,因为 VB6 的重点是处理绘图的控件。有 C++/MFC 背景的开发人员则比较熟悉这个领域,因为 MFC 要求开发人员使用 GDI 更多地控制绘图过程。但是,即使您具备很好的 GDI 背景知识,也会发现本章有许多新东西。

1. GDI +命名空间

表 33-1 列出了 GDI+基类的主要命名空间。  
本章使用的几乎所有的类、结构等都包含在 System.Drawing 命名空间中。

表 33-1

命 名 空 间	说 明
System.Drawing	包含与基本绘图功能有关的大多数类、结构、枚举和委托
System.Drawing.Drawing2D	为大多数高级 2D 和矢量绘图操作提供了支持,包括消除锯齿、几何转换和图形路径

(续表)

命名空间	说明
System.Drawing.Imaging	帮助处理图像(位图、GIF 文件等)的各种类
System.Drawing.Printing	把打印机或打印预览窗口作为输出设备时使用的类
System.Drawing.Design	一些预定义的对话框、属性表和其他用户界面元素，与在设计期间扩展用户界面相关
System.Drawing.Text	对字体和字体系列执行更高级操作的类

2. 设备环境和 Graphics 对象

在 GDI 中，识别输出设备的方式是使用设备环境(DC)对象。该对象存储特定设备的信息，并能把 GDI API 函数调用转换为要发送给该设备的指令。还可以查询设备环境对象，确定对应的设备有什么功能(例如，打印机是彩色的，还是黑白的)，这样才能据此调整输出结果。如果要求设备完成它不能完成的任务，设备环境对象就会检测到，并采取相应的措施(这取决于具体的情形，例如抛出一个异常，或修改请求，获得与该设备的能力最相近的匹配)。

设备环境对象不仅可以处理硬件设备，还可以用作 Windows 的一个桥梁，因此能考虑到 Windows 绘图的要求或限制。例如，如果 Windows 知道只有一小部分应用程序窗口需要重新绘制，设备环境就可以捕获和撤销在该区域外绘图的工作。因为设备环境与 Windows 的关系非常密切，通过设备环境来工作就可以在其他方面简化代码。

例如，硬件设备需要知道在什么地方绘制对象，通常它们需要相对于屏幕(或输出设备)左上角的坐标。但应用程序常常要使用自己的坐标系统在自己窗口的客户区域(用于绘图的窗口)上绘图。而因为窗口可以放在屏幕上的任何位置，用户可以随时移动它，所以在两个坐标之间转换就是一个比较困难的任务。设备环境总是知道窗口在什么地方，并能自动进行这种转换。

在 GDI+中，设备环境包装在.NET 基类 System.Drawing.Graphics 中。大多数绘图工作都是调用 Graphics 实例的方法完成的。实际上，因为 Graphics 类负责处理大多数绘图操作，所以 GDI+中很少有操作不涉及到 Graphics 实例。理解如何处理这个对象是理解如何使用 GDI+在显示设备上绘图的关键。

33.1.2 绘制图形

下面用一个小示例 DisplayAtStartup 来说明如何在应用程序的主窗口中绘图。本章的示例都在 Visual Studio 2008 中创建为 C# Windows 应用程序。对于这种类型的项目，代码向导会提供一个类 Form1，它派生自 System.Windows.Form，表示应用程序的主窗口。还会生成一个类 Program(在 Program.cs 文件中)，表示应用程序的主起点。除非特别声明，否则在所有的示例中，新代码或修改过的代码都添加到向导生成的代码中(可以从 Wrox 网站 [www.wrox.com](http://www.wrox.com) 上下载示例代码)。

注意：

在.NET 的用法中，当说到显示各种控件的应用程序时，“窗口”大都用术语“窗体”来代

替，表示一个矩形对象，它占据了屏幕上的一块区域，代表应用程序。在本章中，我们使用术语“窗口”，因为在手工绘图时，它更有意义。当谈到用于实例化窗体/窗口的.NET类时，使用术语“窗体”。最后，“绘图”或“绘制”可以互换使用，描述在屏幕或其他显示设备上显示一些项的过程。

第一个示例创建一个窗体，并在启动窗体时在构造函数中绘制它。这并不是在屏幕上绘图的最佳方式，这个示例并不能在启动后按照需要重新绘制窗体。但利用这个示例，我们不必做太多的工作，就可以说明绘图的许多问题。

对于这个示例，启动 Visual Studio 2008，创建一个 Windows 应用程序，首先把窗体的背景色设置为白色。把这行代码放在 `InitializeComponent()` 方法的后面，这样 Visual Studio 2008 就会识别该命令，并改变窗体的设计视图的外观。在 Visual Studio Solution Explorer 中单击 Show All Files 按钮，再展开 `Form1.cs` 文件旁边的加号，就可以看到 `Form1.Designer.cs` 文件，在这个文件中，包含了 `InitializeComponent()` 方法。也可以使用设计视图设置背景色，这会添加相同的代码：

```
private void InitializeComponent()
{
    this.components = new System.ComponentModel.Container();
    this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
    this.Text = "Form1";
    this.BackColor = System.Drawing.Color.White;
}
```

接着，给 `Form1` 构造函数添加代码。使用窗体的 `CreateGraphics()` 方法创建一个 `Graphics` 对象，这个对象包含绘图时需要使用的 Windows 设备环境。创建的设备环境与显示设备相关，也与这个窗口相关。

```
public Form1()
{
    InitializeComponent();

    Graphics dc = this.CreateGraphics();
    Show();
    Pen bluePen = new Pen(Color.Blue, 3);
    dc.DrawRectangle(bluePen, 0, 0, 50, 50);
    Pen redPen = new Pen(Color.Red, 2);
    dc.DrawEllipse(redPen, 0, 50, 80, 60);
}
```

然后调用 `Show()` 方法显示窗口。之所以让窗口立即显示，是因为在窗口显示出来之前，我们不能做任何工作——没有可供绘图的地方。

最后，显示一个矩形，其坐标是(0,0)，宽度和高度是 50，再绘制一个椭圆，其坐标是(0, 50)，宽度是 80，高度是 50。注意坐标(x,y)表示从窗口的客户区域左上角开始向右的 x 个像素，向下的 y 个像素——这些是显示出来的图形的左上角坐标。

我们使用的 `DrawRectangle()` 和 `DrawEllipse()` 重载方法分别带 5 个参数。第一个参数是类 `System.Drawing.Pen` 的实例。`Pen` 是许多帮助绘图的支持对象中的一个，它包含如何绘制线条的信息。第一个 `Pen` 表示线条应是蓝色的，其宽度为 3 个像素；第二个 `Pen` 表示线条应是红色

的，其宽度为 2 个像素。后面的 4 个参数是坐标和大小。对于矩形，它们分别表示矩形的左上角坐标(x,y)、其宽度和高度。对于椭圆，这些数值的含义是相同的，但它们是指椭圆假想的外接矩形，而不是椭圆本身。运行代码，会得到如图 33-1 所示的图形。当然，本书不是彩页书，所以看不到颜色。

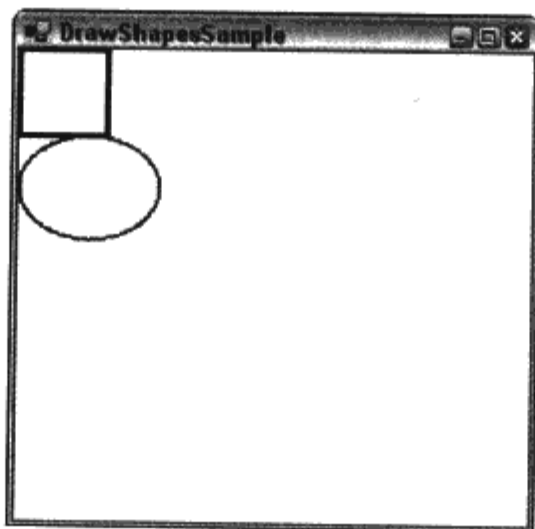


图 33-1

这个屏幕图说明了两个问题。首先，用户可以很清楚地看到窗口客户区域的位置。这是一个白色的区域——该区域受到 `BackColor` 属性设置的影响。还要注意，矩形放在该区域的一角，因为我们指定了坐标(0,0)。其次，注意椭圆的顶部与矩形有轻度的重叠，这与代码中给出的坐标有点不同，这是因为 Windows 在重叠的区域上放置了矩形和椭圆的边框。在默认情况下，Windows 会试图把图形边框所在的线条放在中心位置——但这并不是总能做到的，因为线条是以像素为单位来绘制的，但每个图形的边框理论上位于两个像素之间。结果，1 个像素宽的线条就会正好位于图形顶边和左边的里面，而在右边和底边的外面。这样，从严格意义上讲，相邻图形的边框就会有一个像素的重叠。我们指定的线条宽度比较大，因此重叠区域也会比较大。设置 `Pen.Alignment` 属性(详见 SDK 文档说明)，就可以改变默认的操作方式，但这里使用默认的操作方式就足够了。

但如果运行这个示例，就会注意到窗体的执行过程有点奇怪。如果把它放在那里，或者用鼠标在屏幕移动该窗体，它就工作正常。但如果最小化该窗体，再恢复它，绘制好的图形就不见了。如果在示例中绘制另一个窗体，使之遮挡一部分图形，情况也是这样。如果在该窗体上拖动另一个窗口，使之只遮挡一部分图形，再把该窗口拖离这个窗体，临时被挡住的部分就消失了，只剩下一半椭圆或矩形了！

这是怎么回事？其原因是，如果窗口的一部分被隐藏了，Windows 通常会立即删除与其中显示的内容相关的所有信息。这是必需的，否则存储屏幕数据的内存量就会是个天文数字。一般的计算机在运行时，视频卡设置为显示 1024×768 像素、24 位彩色模式，这表示屏幕上的每个像素占据 3 个字节，则显示整个屏幕就需要 2.25MB(本章后面会说明 24 位颜色的含义)。但是，用户常常让任务栏上有 10 个或 20 个最小化窗口。下面考虑一种最糟糕的情况：20 个窗口，每个窗口如果没有最小化，就占用整个屏幕，如果 Windows 存储了这些窗口包含的可视化信息，当用户恢复它们时，它们就会有 45MB。目前，比较好的图形卡有 64MB 的内存，可以应付这种情况，但在几年前图形卡有 4MB 的内存就不错了。剩余的部分需要存储在计算机的主内存



中。许多人仍在使用旧机器，一些人甚至还在使用 4MB 的图形卡。很显然，Windows 不可能这样管理用户界面。

在窗口的某一部分消失时，那些像素也就丢失了。因为 Windows 释放了保存这些像素的内存。但要注意，窗口的一部分被隐藏了，当它检测到窗口不再被隐藏时，就请求拥有该窗口的应用程序重新绘制其内容。这个规则有一些例外——窗口的一小部分被挡住的时间比较短(例如，从主菜单中选择一个项目，该菜单项向下拉出，临时挡住了下面的窗口)。但一般情况下，如果窗口的一部分被挡住，应用程序就需要在以后重新绘制它。

这就是示例应用程序的一个问题。我们把绘图代码放在 Form1 的构造函数中，当应用程序启动时，就调用该函数一次，不能在以后需要时再次调用该构造函数，重新绘制图形。

在使用 Windows 窗体的服务器控件时，不需要知道这些，这是因为标准控件非常专业，能在 Windows 需要时重新绘制它们自己。这是编写控件时不需要担心实际绘图过程的原因之一。如果要应用程序在屏幕上绘图，还需要在 Windows 要求重新绘制窗口的全部或部分时，确保应用程序会正确响应。下一节将修改这个示例，完成应用程序的响应。

### 33.1.3 使用 OnPaint()绘制图形

上面的解释让您觉得绘制自己的用户界面是比较复杂的，实际上并非如此。让应用程序在需要时绘制自身是非常简单的。

Windows 会利用 Paint 事件通知应用程序完成一些重新绘制的要求。有趣的是，Form 类已经执行了这个事件的处理程序，因此不需要再添加处理程序了。Paint 事件的 Form1 处理程序处理虚方法 OnPaint()的调用，并给它传送一个参数 PaintEventArgs，这表示，我们只需重写 OnPaint()执行画图操作。

我们选择重写 OnPaint()，也可以为 Paint 事件添加自己的事件处理程序(例如 Form1\_Paint 方法)来得到相同的结果，其方式与为其他 Windows Form 事件添加处理程序一样。后一个方法更方便一些，因为可以通过 Visual Studio 2008 属性窗口添加新的事件处理程序，不必键入某些代码。但是我们采用重写 OnPaint()的方式要略为灵活一些，因为这样可以控制何时调用基类窗口进行处理，在文档说明中推荐采用这种方式。

下面创建一个 Windows 应用程序 DrawShapes 来完成这个操作。与以前一样，使用属性窗口把背景色设置为白色，再把窗体的文本改为 DrawShapes Sample，接着在 Form1 类中添加如下代码：

```
protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);
    Graphics dc = e.Graphics;
    Pen bluePen = new Pen(Color.Blue, 3);
    dc.DrawRectangle(bluePen, 0, 0, 50, 50);
    Pen redPen = new Pen(Color.Red, 2);
    dc.DrawEllipse(redPen, 0, 50, 80, 60);
}
```

注意，OnPaint()声明为 protected。OnPaint()一般在类的内部使用，所以类外部的其他代码不知道存在 OnPaint()。

`PaintEventArgs` 是一个派生自 `EventArgs` 的类，一般用于传送有关事件的信息。`PaintEventArgs` 有另外两个属性，其中比较重要的是 `Graphics` 实例，它们主要用于优化绘制窗口中需要绘制的部分。这样就不必调用 `CreateGraphics()`，在 `OnPaint()` 方法中获取设备环境了——用户总是可以得到设备环境。后面将介绍其他属性，它包含哪些窗口部分需要重新绘制的详细信息。

在 `OnPaint()` 的执行代码中，首先从 `PaintEventArgs` 中引用 `Graphics` 对象，再像以前那样绘制图形。最后调用基类的 `OnPaint()` 方法，这一步是非常重要的。我们重写了 `OnPaint()` 方法，完成了绘图工作，但 Windows 在绘图过程中可能会执行一些它自己的工作。这些工作都在 .NET 基类的 `OnPaint()` 方法中完成。

#### 注意：

对于这个示例，删除 `base.OnPaint()` 的调用似乎并没有任何影响，但不要试图删除这个调用。这样有可能阻止 Windows 正确执行任务，结果是无法预料的。

在应用程序第一次启动，窗口第一次显示出来时，也调用了 `OnPaint()`，所以不需要在构造函数中复制绘图代码。

运行这段代码，得到的结果将与前面的示例的结果相同，但现在，当最小化窗口或隐藏它的一部分时，应用程序会正确执行。

### 33.1.4 使用剪切区域

上一节的 `DrawShapes` 示例说明了在窗口中绘图的主要规则，但它并不是很高效。原因是它试图绘制窗口中的所有内容，而没有考虑需要绘制多少内容。如图 33-2 所示，运行 `DrawShapes` 示例，当该示例在屏幕上绘图时，打开另一个窗口，把它移动到 `DrawShapes` 窗体上，使之隐藏一部分窗体。

但移动上面的窗口时，`DrawShapes` 窗口会再次全部显示出来，Windows 通常会给窗体发送一个 `Paint` 事件，要求它重新绘制本身。矩形和椭圆都位于客户区域的左上角，所以在任何时候都是可见的，在本例中不需要重新绘制这部分，而只需要重新绘制白色背景区域。但是，Windows 并不知道这一点，它认为应引发 `Paint` 事件，调用 `OnPaint()` 方法的执行代码。`OnPaint()` 不必重新绘制矩形和椭圆。

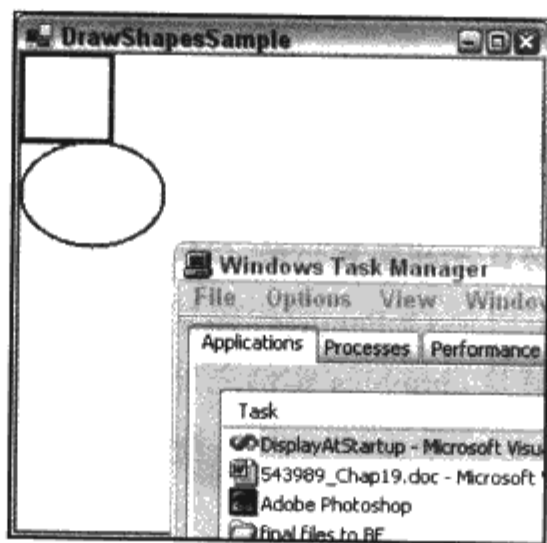


图 33-2

在本例中，没有重新绘制图形。原因是我们使用了设备环境。Windows 将利用重新绘制某些区域所需要的信息预先初始化设备环境。在 GDI 中，被标记出来的重绘区域称为无效区域，但在 GDI+ 中，该术语改为剪切区域，设备环境知道这个区域的内容，它截取在这个区域外部的绘图操作，且不把相关的绘图命令传送给图形卡。这听起来不错，但仍有一个潜在的性能损失。在确定是在无效区域外部绘图前，我们不知道必须进行多少设备环境处理。在某些情况下，要处理的任务比较多，因为计算哪些像素需要改变为什么颜色，将会占用许多处理器时间(好的图形卡会提供硬件加速，对此有一定的帮助)。

其底线是让 Graphics 实例完成在无效区域外部的绘图工作，肯定会浪费处理器时间，减慢应用程序的运行。在设计优良的应用程序中，代码将执行一些检查，以查看需要进行哪些绘图工作，然后调用相关的 Graphics 实例方法。本节将编写一个新示例 DrawShapesWithClipping，修改 DisplayShapes 示例，只完成需要的重新绘制工作。在 OnPaint() 代码中，进行一个简单的测试，看看无效区域是否与需要绘制的区域重叠，如果是，就调用绘图方法。

首先，需要获得剪切区域的信息。这需要使用 PaintEventArgs 的另一个属性。这个属性叫做 ClipRectangle，包含要重绘区域的坐标，并包装在一个结构实例 System.Drawing.Rectangle 中。Rectangle 是一个相当简单的结构，包含 4 个属性：Top、Bottom、Left 和 Right。它们分别包含矩形的上下的垂直坐标、左右的水平坐标。

接着，需要确定进行什么测试，以决定是否进行绘制。这里进行一个简单的测试。注意，在绘图过程中，矩形和椭圆完全包含在(0,0)到(80,130)的矩形客户区域中，实际上，点(82,132)就已经在安全区域中了，因为线条大约偏离这个区域一个像素。所以我们要看看剪切区域的左上角是否在这个矩形区域内。如果是，就重新绘制，如果不是，就不必麻烦了。

下面是代码：

```
protected override void OnPaint( PaintEventArgs e )
{
    base.OnPaint(e);
    Graphics dc = e.Graphics;

    if (e.ClipRectangle.Top < 132 && e.ClipRectangle.Left < 82)
    {
        Pen bluePen = new Pen(Color.Blue, 3);
        dc.DrawRectangle(bluePen, 0, 0, 50, 50);
        Pen redPen = new Pen(Color.Red, 2);
        dc.DrawEllipse(redPen, 0, 50, 80, 60);
    }
}
```

注意，显示的结果与以前显示的结果完全相同——但这次进行了早期测试，确定了哪些区域不需要绘制，提高了性能。还要注意这个是否进行绘图的测试是非常粗略的。还可以进行更精细的测试，确定矩形或者椭圆是否要重新绘制。这里有一个平衡。可以在 OnPaint() 中进行更复杂的测试，以提高性能，也可以使 OnPaint() 代码复杂一些。进行一些测试总是值得的，因为编写一些代码，可以更多地了解除 Graphics 实例之外的绘制内容，Graphics 实例只是盲目地执行绘图命令。

### 33.2 测量坐标和区域

在上一个示例中，我们遇到了基本结构 `Rectangle`，它用于表示矩形的坐标。GDI+使用几个类似的结构来表示坐标或区域。下面介绍几个结构，它们都是在 `System.Drawing` 命名空间中定义的，如表 33-2 所示。

表 33-2

结 构	主要的公共属性
Point 和 PointF	X,Y
Size 和 SizeF	Width, Height
Rectangle 和 RectangleF	Left, Right , Top, Bottom, Width, Height, X, Y, Location, Size

注意，其中的许多对象都有许多其他属性、方法或运算符重载，这里没有列出来。本节只讨论最重要的成员。

#### 33.2.1 Point 和 PointF 结构

从概念上讲，`Point` 在这些结构中最简单的，在数学上，它等价于一个二维矢量，包含两个公共整型属性，表示它与某个特定位置的水平和垂直距离(在屏幕上)，如图 33-3 所示。

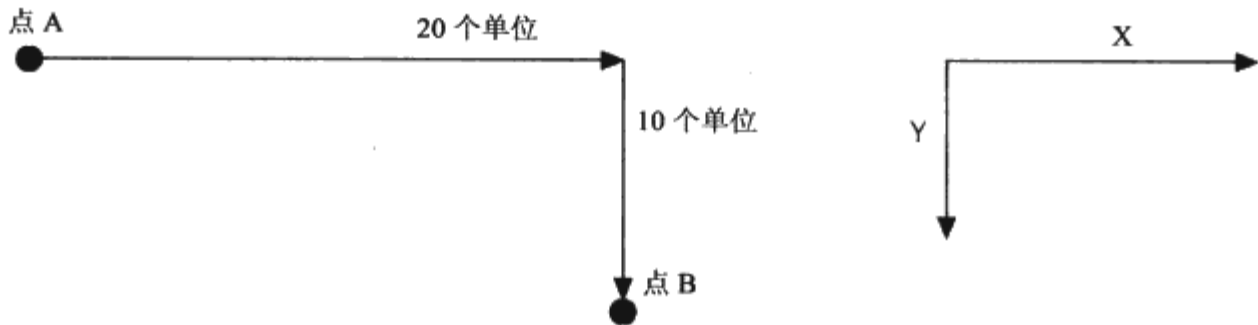


图 33-3

为了从点 A 到点 B，需要水平移动 20 个单位，并向下垂直移动 10 个单位，在图中标为 `x` 和 `y`，这就是它们的一般含义。我们可以创建一个 `Point` 结构，表示它们：

```
Point ab = new Point(20, 10);
Console.WriteLine("Moved {0} across, {1} down", ab.X, ab.Y);
```

`X` 和 `Y` 都是读写属性，因此可以在 `Point` 中设置这些值：

```
Point ab = new Point();
ab.X = 20;
ab.Y = 10;
Console.WriteLine("Moved {0} across, {1} down", ab.X, ab.Y);
```

注意，按照惯例，水平和垂直坐标表示为 `x` 和 `y`(小写)，但相应的 `Point` 属性是 `X` 和 `Y`(大写)，因为在 C# 中，公共属性的一般约定是名称以大写字母开头。

`PointF` 与 `Point` 完全相同，但 `X` 和 `Y` 属性的类型是 `float`，而不是 `int`。`PointF` 用于坐标不是

整数值的情况。已经为这些结构定义了数据类型转换，这样就可以把 `Point` 隐式转换为 `PointF`，(注意，因为 `Point` 和 `PointF` 是结构，这种转换实际上涉及到数据的复制)。但没有相应的逆过程，要把 `PointF` 转换为 `Point`，必须显式地复制值，或使用下面的 3 个转换方法 `Round()`、`Truncate()` 和 `Ceiling()`：

```
PointF abFloat = new PointF(33.5F, 10.9F);

// converting to Point
Point ab = new Point();
ab.X = (int)abFloat.X;
ab.Y = (int)abFloat.Y;
Point ab1 = Point.Round(abFloat);
Point ab2 = Point.Truncate(abFloat);
Point ab3 = Point.Ceiling(abFloat);

// but conversion back to PointF is implicit
PointF abFloat2 = ab;
```

下面看看测量单位。在默认情况下，GDI+把单位看作是屏幕(或打印机，无论图形设备是什么，都可以这样认为)上的像素，这就是 `Graphics` 对象方法把它们接收到的坐标看作其参数的方式。例如，点 `new Point(20,10)`表示在屏幕上水平移动 20 个像素，向下垂直移动 10 个像素。通常这些像素从窗口客户区域的左上角开始测量，如上面的示例所示。但是，情况并不总是如此。例如，在某些情况下，需要以窗口的左上角(包括其边框)为原点来绘图，甚至以屏幕的左上角为原点。但在大多数情况下，除非文档说明书说明，否则都可以假定像素值是相对于客户区域的左上角。

在分析了滚动后，本章将在讨论 3 个坐标系统(世界、页面和设备坐标)时介绍这个主题。

### 33.2.2 Size 和 SizeF 结构

与 `Point` 和 `PointF` 一样，`Size` 也有两个变体。`Size` 结构用于 `int` 类型，`SizeF` 用于 `float` 类型，除此之外，`Size` 和 `SizeF` 是完全相同的。下面主要讨论 `Size` 结构。

在许多情况下，`Size` 结构与 `Point` 结构是相同的，它也有两个整型属性，表示水平和垂直距离——主要区别是这两个属性的名称不是 `X` 和 `Y`，而是 `Width` 和 `Height`。前面的图 33-3 可以表示为：

```
Size ab = new Size(20,10);
Console.WriteLine("Moved {0} across, {1} down", ab.Width, ab.Height);
```

严格地讲，`Size` 在数学上与 `Point` 表示的含义相同；但在概念上它使用的方式略有不同。`Point` 用于说明实体在什么地方，而 `Size` 用于说明实体有多大。但是，`Size` 和 `Point` 是紧密相关的，目前甚至支持它们之间的显式转换：

```
Point point = new Point(20, 10);
Size size = (Size) point;
Point anotherPoint = (Point) size;
```

例如，考虑前面绘制的矩形，其左上角的坐标是(0,0)，大小是(50,50)。这个矩形的大小是(50,50)，可以用一个 `Size` 实例来表示。其右下角的坐标也是(50,50)，但它由一个 `Point` 实例来



表示。要理解这个区别，假定在另一个位置绘制该矩形，其左上角的坐标是(10,10)：

```
dc.DrawRectangle(bluePen, 10, 10, 50, 50);
```

现在其右下角的坐标是(60,60)，但大小不变，仍是(50,50)。

Point 和 Size 结构的相加运算符都已经重载了，所以可以把一个 Size 加到 Point 结构上，得到另一个 Point 结构：

```
static void Main(string[] args)
{
    Point topLeft = new Point(10, 10);
    Size rectangleSize = new Size(50, 50);
    Point bottomRight = topLeft + rectangleSize;
    Console.WriteLine("topLeft = " + topLeft);
    Console.WriteLine("bottomRight = " + bottomRight);
    Console.WriteLine("Size = " + rectangleSize);
}
```

把这段代码作为控制台应用程序 PointAndSizes 来运行，会得到如图 33-4 所示的结果。

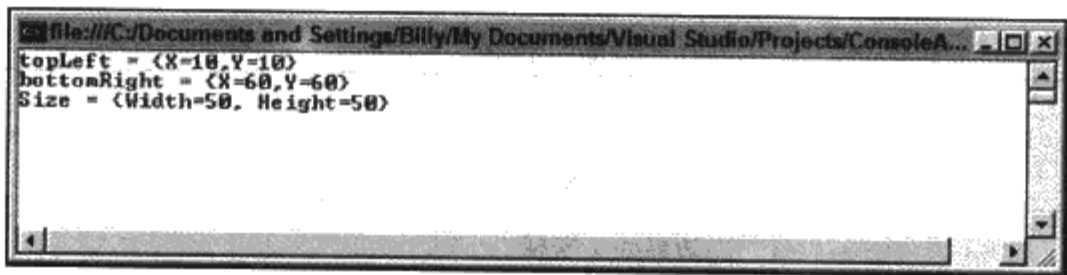


图 33-4

注意，这个结果也说明了 Point 和 Size 的 ToString() 方法已被重写，并以 {X,Y} 格式显示值。

也可以从一个 Point 减去某个 Size，得到另一个 Point，还可以把两个 Size 加在一起，得到另一个 Size。但不能把一个 Point 加到另一个 Point 上。Microsoft 认为 Point 相加在概念上没有意义，所以不支持加(+)运算符的任何重载版本进行这样的操作。

还可以在 Point 和 Size 之间进行显式的数据类型转换：

```
Point topLeft = new Point(10, 10);
Size s1 = (Size)topLeft;
Point p1 = (Point)s1;
```

在进行这样的数据类型转换时，s1.Width 被赋予 topLeft.X 的值，s1.Height 被赋予 topLeft.Y 的值，因此 s1 包含(10,10)。p1 最终的值与 topLeft 的值相同。

### 33.2.3 Rectangle 和 RectangleF 结构

这两个结构表示一个矩形区域(通常在屏幕上)。与 Point 和 Size 一样，这里只介绍 Rectangle 结构，RectangleF 与 Rectangle 基本相同，但它的属性类型是 float，而 Rectangle 的属性类型是 int。

Rectangle 可以看作由一个 Point 和一个 Size 组成，其中 Point 表示矩形的左上角，Size 表示其大小。它的一个构造函数把 Point 和 Size 作为其参数。下面重新编写前面 DrawShapes 示

例的代码，绘制一个矩形：

```
Graphics dc = e.Graphics;
Pen bluePen = new Pen(Color.Blue, 3);
Point topLeft = new Point(0,0);
Size howBig = new Size(50,50);
Rectangle rectangleArea = new Rectangle(topLeft, howBig);
dc.DrawRectangle(bluePen, rectangleArea);
```

这段代码也使用 Graphics.DrawRectangle()的另一个重载方法,它的参数是 Pen 和 Rectangle 结构。

通过按顺序提供矩形的左上角水平和垂直坐标，宽度和高度(它们都是数字)，可以构造一个 Rectangle：

```
Rectangle rectangleArea = new Rectangle(0, 0, 50, 50)
```

Rectangle 包含几个读写属性，如表 33-3 所示，可以用不同的属性组合来设置或提取它的维数。

表 33-3

属 性	说 明
int Left	左边界的 x 坐标
int Right	右边界的 x 坐标
int Top	顶边的 y 坐标
int Bottom	底边的 y 坐标
int X	与 Left 相同
int Y	与 Top 相同
int Width	矩形的宽度
int Height	矩形的高度
Point Location	左上角
Size Size	矩形的大小

注意，这些属性都不是独立的，例如，设置 Width 会影响 Right 的值。

33.2.4 Region

Region 表示屏幕上一个有复杂图形的区域。例如，图 33-5 中的阴影区域就可以用 Region 表示。

可以想象，初始化 Region 实例的过程相当复杂。从广义上看，可以指定哪些简单的图形组成这个区域，或指定绘制这个区域边界的路径。如果需要处理这样的区域，就应掌握 SDK 文档中的 Region 类。

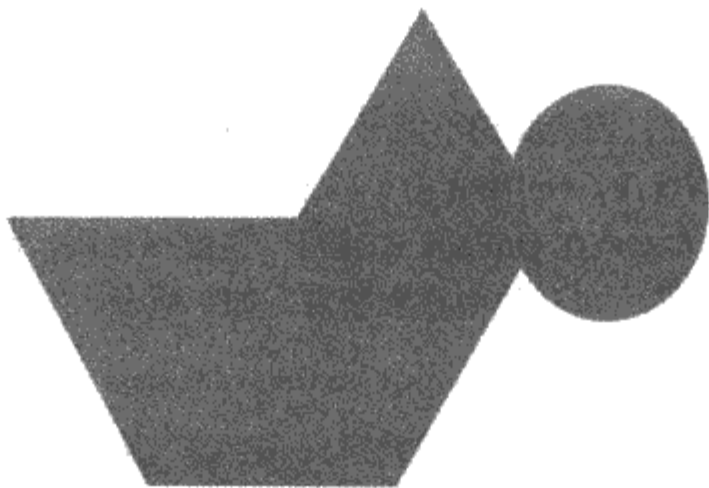


图 33-5

### 33.3 调试须知

下面准备进行一些更高级的绘图工作。但首先介绍几个调试问题。如果在本章的示例中设置了断点，就会注意到调试图形例程不像调试其他程序那样简单。这是因为进入和退出调试程序常常会把 Paint 信息传送给应用程序。结果是在 OnPaint 重载方法上设置的断点会让应用程序一遍又一遍地绘制它本身，这样应用程序就不能完成任何工作。

这是很典型的一种情形。要明白为什么应用程序没有正确显示，可以在 OnPaint 上设置断点。应用程序会像期望的那样，遇到断点后，就会进入调试程序，此时在前景上会显示开发环境 MDI 窗口。如果把开发环境设置为满屏显示，以便更易于观察所有的调试信息，就会完全隐藏目前正在调试的应用程序。

接着，检查某些变量的值，希望找出某些有用的信息。然后按下 F5，告诉应用程序继续执行，完成某些处理后，看看应用程序在显示其他内容时会发生什么。但首先发生的是应用程序显示在前景中，Windows 检测到窗体再次可见，并提示给它发送了一个 Paint 事件。当然这表示程序遇到了断点。如果这就是我们想要的结果，那就很好。但更常见的是，我们希望以后在应用程序绘制了某些有趣的内容后再遇到断点，例如在选择某些菜单项，读取一个文件或者以其他方式改变显示的内容之后再遇到断点。这听起来就是我们需要的结果。我们根本没有在 OnPaint 中设置断点，应用程序也不会显示它在最初的启动窗口中显示的内容之外的其他内容。

有一种方式可以解决这个问题。如果有足够大的屏幕，最简单的方式就是平铺开发环境窗口，而不是把它设置为最大化，使之远离应用程序窗口，这样应用程序就不会被挡住了。但在大多数情况下，这并不是一个有效的解决方案，因为这样会使开发环境窗口过小（也可以使用第二个监视器）。另一个解决方案使用相同的规则，即把应用程序声明为在调试时放在最上层。方法是在 Form 类中设置属性 TopMost，这很容易在 InitializeComponent 方法中完成：

```
private void InitializeComponent()  
{  
    this.TopMost = true;  
}
```

也可以在 Visual Studio 2008 的属性窗口中设置这个属性。

窗口设置为 TopMost 表示应用程序不会被其他窗口挡住(除了其他放在最上层的窗口)。它

总是放在其他窗口的上面，甚至在另一个应用程序得到焦点时，也是这样。这是任务管理器的执行方式。

利用这个技巧时必须小心，因为我们不能确定 Windows 何时会决定应为某种原因引发 Paint 事件。如果在某些特殊的情况下，OnPaint 出了问题(例如，应用程序在选择某个菜单项后绘图，但此时出了问题)。最好的方式是在 OnPaint 中编写一些虚拟代码，测试某些条件，这些条件只在特殊的情况下才为 true。然后在 if 块中设置断点，如下所示。

```
protected override void OnPaint( PaintEventArgs e )
{
    // Condition() evaluates to true when we want to break
    if ( Condition() == true)
    {
        int ii = 0;    // <-- SET BREAKPOINT HERE!!!
    }
}
```

上面的这段代码是设置条件断点的一种简捷方式。

### 33.4 绘制可滚动的窗口

前面的 DrawShapes 示例运行良好，因为需要绘制的内容正好适合最初的窗口大小。本节介绍如果绘制的内容不适合窗口的大小，需要做哪些工作。

下面扩展 DrawShapes 示例，以解释滚动的概念。为了使该示例更符合实际，首先创建一个 BigShapes 示例，该示例将矩形和椭圆画大一些。此时将使用 Point、Size 和 Rectangle 结构定义绘图区域，说明如何使用它们。进行了这样的修改后，Form1 类的相关部分如下所示：

```
// member fields
private Point rectangleTopLeft = new Point(0, 0);
private Size rectangleSize = new Size(200,200);
private Point ellipseTopLeft = new Point(50, 200);
private Size ellipseSize = new Size(200, 150);
private Pen bluePen = new Pen(Color.Blue, 3);
private Pen redPen = new Pen(Color.Red, 2);

protected override void OnPaint( PaintEventArgs e )
{
    base.OnPaint(e);
    Graphics dc = e.Graphics;

    if (e.ClipRectangle.Top < 350 || e.ClipRectangle.Left < 250)
    {
        Rectangle rectangleArea =
            new Rectangle (rectangleTopLeft, rectangleSize);
        Rectangle ellipseArea =
            new Rectangle (ellipseTopLeft, ellipseSize);
        dc.DrawRectangle(bluePen, rectangleArea);
        dc.DrawEllipse(redPen, ellipseArea);
    }
}
```

注意，这里还把 Pen、Size 和 Point 对象变成成员字段——这比每次需要绘图时都创建一个新 Pen 的效率高。

运行这个示例，得到如图 33-6 所示的结果。

这里有一个问题，图形在 300×300 像素的绘图区域中放不下。

一般情况下，如果文档太大，不能完全显示，应用程序就会添加滚动条，以便用户滚动窗口，查看其中选中的部分。这是另一个区域，在该区域中如果使用标准控件建立 Windows 窗体，就让 .NET 运行库和基类来处理。如果窗体上有各种控件，Form 实例一般知道这些控件在哪里，如果其窗口比较小，Form 实例就知道需要添加滚动条。Form 实例还会自动添加滚动条，不仅如此，它还可以正确绘制用户滚动到的部分屏幕。此时，用户不需要在代码中做什么工作。但在本章中，我们要在屏幕上绘制图形，所以要帮助 Form 实例确定何时能滚动。

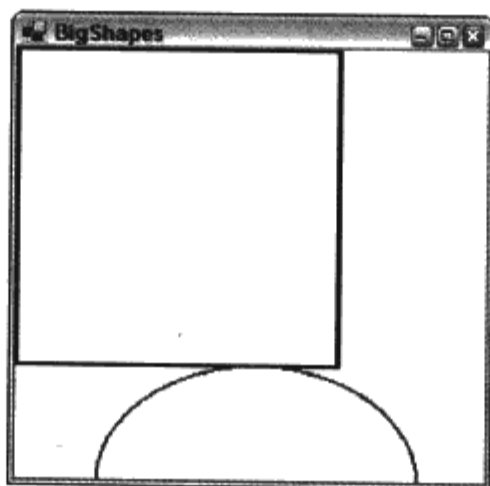


图 33-6

添加滚动条是很简单的。Form 仍会处理所有的操作——因为它不知道绘图区域有多大。在上面的 BigShapes 示例中没有滚动条的原因是，Windows 不知道它们需要滚动条。我们需要确定的是，矩形的大小从文档的左上角(或者是在进行任何滚动前的客户区域左上角)开始向下延伸，其大小应足以包含整个文档。本章把这个区域称为文档区域，在图 33-7 中可以看出，本例的文档区域应是(250, 350)像素。

使用相关的属性 `Form.AutoScrollMinSize` 即可确定文档的大小。因此给 `InitializeComponent()` 方法或 `Form1` 构造函数添加下述代码：

```
private void InitializeComponent()
{
    this.components = new System.ComponentModel.Container();
    this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
    this.Text = "Form1";
    this.BackColor = System.Drawing.Color.White;
    this.AutoScrollMinSize = new Size(250, 350);
}
```

另外，`AutoScrollMinSize` 属性还可以用 Visual Studio 2008 属性窗口设置。注意要访问 `Size` 类，需要添加下面的 `using` 语句：

```
using System.Drawing;
```

在应用程序启动时设置最小尺寸，并保持不变，在这个应用程序中是必要的，因为我们知道屏幕区域一般有多大。在运行该应用程序时，这个“文档”是不会改变大小的。但要记住，如果应用程序显示文件内容的操作，或者执行某些改变屏幕区域的操作，就需要在其他时间设置这个属性(此时，必须手工调整代码，Visual Studio 2008 属性窗口只能在构建窗体时设置属性



的初始值)。

设置 `MinScrollSize` 只是一个开始，仅有它是不够的。如图 33-8 所示为示例应用程序目前的外观——开始时，让屏幕正确显示图形。

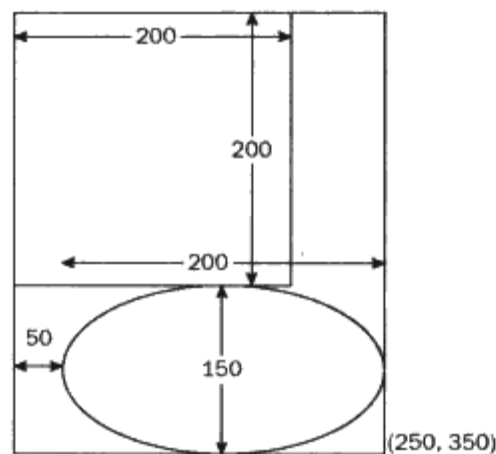


图 33-7

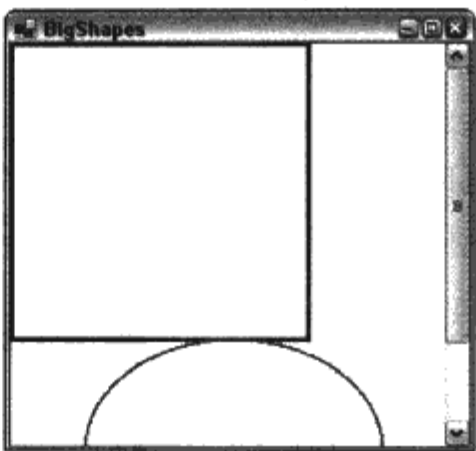


图 33-8

注意，不仅窗体正确地设置了滚动条，而且它们的大小也正确设置了，以指定文档正确显示的比例。可以试着在运行示例时重新设置窗口的大小，这样就会发现滚动条会正确响应，甚至如果使窗口变得足够大，不再需要滚动条时，滚动条就会消失。

但是，如果使用一个滚动条，并向下滚动它，会发生什么情况呢？如图 33-9 所示。显然，出现了错误！

出错的原因是我们没有在 `OnPaint()` 重写方法的代码中考虑滚动条的位置。如果最小化窗口，再恢复它，重新绘制一遍窗口，就可以很清楚地看出这一点。结果如图 33-10 所示。

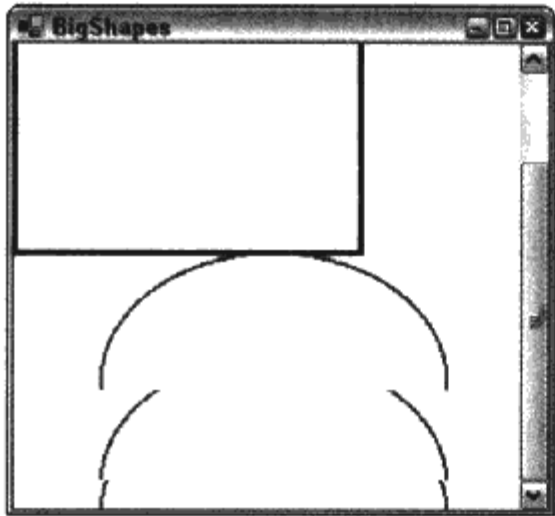


图 33-9

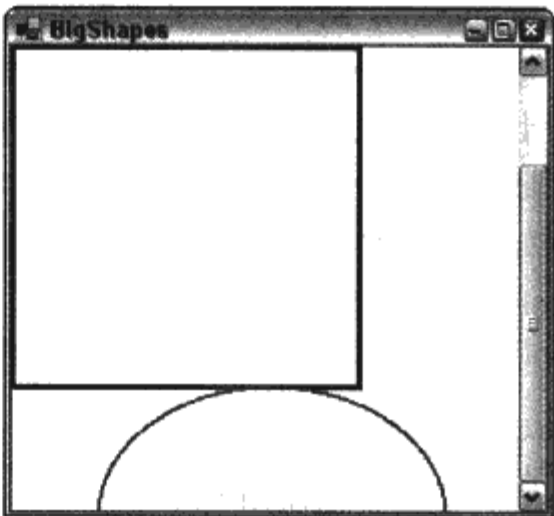


图 33-10

图形像以前一样进行了绘制，矩形的左上角嵌套在客户区域的左上角，就好像根本没有移动过滚动条一样。

在更正这个问题前，先介绍一下在这些屏幕图上发生了什么。

首先从 `BigShapes` 示例开始，如图 33-8 所示。在这个例子中，整个窗口刚刚重新进行了绘制。看看前面的代码，该代码的作用是使 `graphics` 实例用左上角坐标(0,0)(相对于窗口客户区域的左上角)绘制一个矩形——它是已经绘制过的。问题是，`graphics` 实例在默认情况下把坐标解释为是相对于客户窗口的，它不知道滚动条的存在。代码还没有尝试为滚动条的位置调整坐标。椭圆也是这样。

下面处理图 33-9 的问题。在滚动后，注意窗口上半部分显示正确，这是因为它们是在应用程序第一次启动时绘制的。在滚动窗口时，Windows 没有要求应用程序重新绘制已经显示在屏幕中的内容。Windows 只指出屏幕上目前显示的内容可以平滑地移动，以匹配滚动条的位置。这是一个非常高效的过程，因为它也能使用某些硬件加速来完成。在这个屏幕图中，有错误的是窗口下部的 1/3 部分。在应用程序第一次显示时，没有绘制这部分窗口，因为在滚动窗口前，这部分在客户区域的外部。这表示 Windows 要求 BigShapes 应用程序绘制这个区域。它引发 Paint 事件，把这个区域作为剪切的矩形。这也是 OnPaint() 重载方法完成的任务。

问题的另一种表达方式是我们将坐标表示为相对于文档开头的左上角——需要转换它们，使之相对于客户区域的左上角。图 33-11 说明了这一点。

为了使该图更清晰，我们向下向右扩展了该文档，超出了屏幕的边界，但这不会改变我们的推论，我们还假定其上还有一个水平滚动条和一个垂直滚动条。

在该图中，细矩形标记了屏幕区域的边框和整个文档的边框。粗线条标记了要绘制的矩形和椭圆。P 标记要绘制的某个随意点，这个点在后面会作为一个示例。在调用绘图方法时，提供 graphics 实例和从 B 点到 P 点的矢量，这个矢量表示为一个 Point 实例。我们实际上需要给出从点 A 到点 P 的矢量。

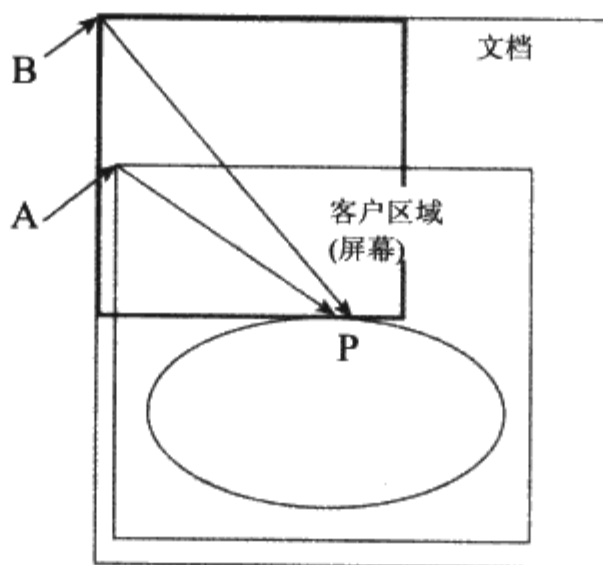


图 33-11

问题是，我们不知道从 A 点到 P 点的矢量。而知道从 B 点到 P 点的矢量，这是 P 相对于文档左上角的坐标——我们要在文档的 P 点绘图。还知道从 B 点到 A 点的矢量，这是滚动的距离，它存储在 Form 类的一个属性 AutoScrollPosition 中。但是不知道从 A 点到 P 点的矢量。

要解决这个问题，只需进行矢量相减即可。例如，要从 B 点到 P 点，可以水平移动 150 个像素，垂直向下移动 200 个像素。而要从 B 点到 A 点，就需要水平移动 10 个像素，垂直向下移动 57 个像素。这表示，要从 A 点到 P 点，需要水平移动 140 个像素(=150-10)，垂直向下移动 143 个像素(=200-57)。为了使之更简单，Graphics 类执行了一个方法来进行这些计算，这个方法是 TranslateTransform，我们给它传送水平和垂直坐标，表示客户区域的左上角相对于文档的左上角(AutoScrollPosition 属性，它是图中从 B 到 A 的矢量)，然后 Graphics 设备考虑客户区域相对于文档区域的位置，计算出这些坐标。

解释完之后，所需做的是把下面这行代码添加到绘图代码中：

```
dc.TranslateTransform(this.AutoScrollPosition.X, this.AutoScrollPosition.Y);
```

但在本示例中，它有点复杂，因为我们还要测试剪切区域，看看是否需要进行绘制工作。这个测试需要调整，把滚动的位置也考虑在内。完成后，该示例的整个绘图代码如下所示：

```
protected override void OnPaint( PaintEventArgs e )
{
    base.OnPaint(e);
    Graphics dc = e.Graphics;
    Size scrollOffset = new Size(this.AutoScrollPosition);

    if (e.ClipRectangle.Top+scrollOffset.Width < 350 ||
        e.ClipRectangle.Left+scrollOffset.Height < 250)
    {
        Rectangle rectangleArea = new Rectangle
            (rectangleTopLeft+scrollOffset, rectangleSize);
        Rectangle ellipseArea = new Rectangle
            (ellipseTopLeft+scrollOffset, ellipseSize);
        dc.DrawRectangle(bluePen, rectangleArea);
        dc.DrawEllipse(redPen, ellipseArea);
    }
}
```

现在，滚动代码工作正常，最后得到正确的滚动屏幕图，如图 33-12 所示。

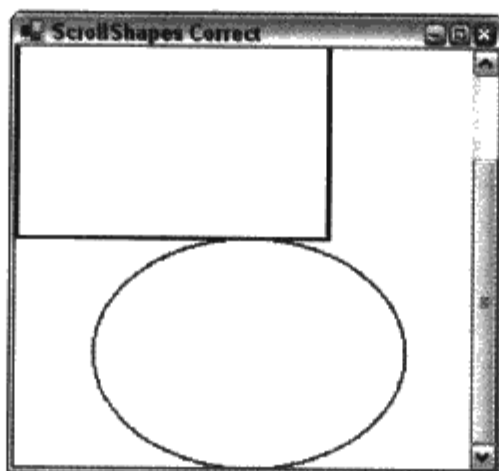


图 33-12

## 33.5 世界、页面和设备坐标

测量相对于文档区域左上角的位置和测量相对于屏幕(桌面)左上角的位置之间的区别非常重要，GDI+为它们指定了不同的名称：

- 世界坐标(World Coordinate): 要测量的点距离文档区域左上角的位置(以像素为单位)。
- 页面坐标(Page Coordinate): 要测量的点距离客户区域左上角的位置(以像素为单位)。

**注意：**

熟悉 GDI 的开发人员要注意，世界坐标对应于 GDI 中的逻辑坐标。页面坐标对应于设备坐标。还要注意，编写逻辑和设备坐标之间的转换代码在 GDI+中有了变化。在 GDI 中，转换是使用 Windows API 函数 LPtoDP()和 DPtoLP()通过设备环境进行的，而在 GDI+中，由 Control 类来维护转换过程中所需要的信息，Form 和各种 Windows 窗体控件设备派生于 Control 类。

GDI+ 还有第 3 种坐标，即设备坐标(Device Coordinate)。设备坐标类似于页面坐标，但其测量单位不是像素；而是用户通过调用 `Graphics.PageUnit` 属性指定的单位。它可以使用的单位除了默认的像素外，还包括英寸和毫米。本章虽然没有使用 `PageUnit` 属性，但它可用作获取设备的不同像素密度的方式。例如，在大多数监视器上，100 像素大约是 1 英寸。但是，激光打印机可以达到 1 200 dpi(点/英寸)——这表示一个 100 像素宽的图形在该激光打印机上打印时会比较小。把单位设置为英寸，指定图形为 1 英寸宽，就可以确保图形在不同的设备上有相同的大小。

```
Graphics dc = this.CreateGraphics();
dc.PageUnit = GraphicsUnit.Inch;
```

`GraphicsUnit` 枚举中的可用值如下：

- `Display`: 指定显示的测量单位
- `Document`: 把文档单位(1/300 英寸)定义为测量单位
- `Inch`: 把英寸定义为测量单位
- `Millimeter`: 把毫米定义为测量单位
- `Pixel`: 把像素定义为测量单位
- `Point`: 把打印机的点数(1/72 英寸)定义为测量单位
- `World`: 把世界坐标系定义为测量单位

## 33.6 颜色

本节介绍如何在绘制图形时指定使用的颜色。

在 GDI+ 中，颜色用 `System.Drawing.Color` 结构的实例来表示。一般情况下，初始化这个结构后，就不能使用对应的 `Color` 实例对该结构进行操作了——只能把它传送给其他需要 `Color` 的调用方法。前面我们遇到过这种结构，在前面的每个示例中都设置了窗口客户区域的背景色，还设置了要显示的各种图形的颜色。`Form.BackColor` 属性返回一个 `Color` 实例。本节将详细介绍这个结构，特别是要介绍构建 `Color` 的几种不同方式。

### 33.6.1 红绿蓝(RGB)值

监视器可以显示的颜色总数非常大——超过 160 万。其确切的数字是 2 的 24 次方，即 16 777 216。显然，我们需要对这些颜色进行索引，才能指定在给定的某个像素上要显示什么颜色。

给颜色进行索引的最常见方式是把它们分为红绿蓝成分，这种方式基于下述原则：人眼可以分辨的任何颜色都是由一定量的红色光、绿色光和蓝色光组成的。这些光称为成分(component)。实际上，如果每种成份的光分为 256 种不同的强度，它们提供了足够平滑的过渡，可以把人眼能分辨出来的图像显示为具有照片质量。因此，指定颜色时，可以给出这些成分的量，其值在 0~255 之间，其中 0 表示没有这种成分，255 表示这种成分的光达到最大的强度。

这给出了向 GDI+ 说明颜色的第一种方式。可以调用静态函数 `Color.FromArgb()` 指定该颜色的红绿蓝值。Microsoft 没有为此提供构造函数，原因是除了一般的 RGB 成分外，还有其他方

式表示颜色。因此，Microsoft 认为给定义的构造函数传递参数会引起误解：

```
Color redColor = Color.FromArgb(255,0,0);
Color funnyOrangyBrownColor = Color.FromArgb(255,155,100);
Color blackColor = Color.FromArgb(0,0,0);
Color whiteColor = Color.FromArgb(255,255,255);
```

3 个参数分别是红绿蓝值。这个函数有许多重载方法，其中一些也允许指定 Alpha 混合值(这是方法名 FromArgb() 中的 A)。Alpha 混合超出了本章的范围，但把它与屏幕上已有的颜色混合起来，可以描绘出半透明的颜色。这可以得到一些很漂亮的效果，常常用于游戏。

### 33.6.2 命名的颜色

使用 FromArgb() 构造颜色是一种非常灵活的技巧，因为它表示我们可以指定人眼能辨识出的任何颜色。但是，如果要得到一种简单、标准、众所周知的纯色，例如红色或蓝色，命名想要的颜色是比较简单的。因此 Microsoft 还在 Color 中提供了许多静态属性，每个属性都返回一种命名的颜色。在下面的示例中，把窗口的背景色设置为白色时，就使用了其中一种属性：

```
this.BackColor = Color.White;

// has the same effect as:
// this.BackColor = Color.FromArgb(255, 255, 255);
```

有几百种这样的颜色。完整的列表参见 SDK 文档说明。它们包括所有的纯色：红、白、蓝、绿和黑等，还包括 MediumAquaMarine、LightCoral 和 DarkOrchid 等颜色。还有一个 KnownColor 枚举，它列出了命名的颜色。

#### 注意：

每种命名的颜色都表示一组 RGB 值，它们最初是多年前选择出来用在 Internet 上的。这种方式提供了色谱上一组有用的颜色，Web 浏览器可以辨识出这些颜色的名称——因此不必在 HTML 代码中显式写出它们的 RGB 值。几年前，这些颜色仍很重要，因为早期的浏览器不必准确地显示非常多的颜色，命名的颜色可以在大多数浏览器中准确地显示出来。目前它们已经不那么重要了，因为现代的 Web 浏览器能准确地显示任何 RGB 值。还有一些 Web 安全的调色板可以为开发人员提供可用于大多数浏览器的、非常丰富的颜色。

### 33.6.3 图形显示模式和安全的调色板

原则上监视器可以显示超过 160 万种 RGB 颜色，实际上这取决于如何在计算机上设置显示属性。在 Windows 中，传统上有 3 个主要的颜色选项(但有些机器还提供其他选项，这取决于硬件)：真彩色(24 位)、增强色(16 位)和 256 色。(在目前的一些图形卡上，真彩色是 32 位的，因为硬件进行了优化，但此时 32 位中只有 24 位用于该颜色)。

只有真彩色模式允许同时显示所有的 RGB 颜色。这听起来是最佳选择，但它是有代价的：完整的 RGB 值需要用 3 个字节来保存，这表示要显示的每个像素都需要用图形卡内存中的 3 个字节来保存。如果图形卡内存需要额外的费用(这种限制现在已经不像以前那样普遍了)，就可以选择其他模式。增强色模式用两个字节表示一个像素。每个 RGB 成分用 5 位就足够了。



所以红色只有 32 种不同的强度，而不是 256 种。蓝色和绿色也是这样，总共有 65536 种颜色。这对于需要偶尔查看照片质量级的图像来说是足够了，但比较微妙的阴影区域会被破坏。

256 色模式给出的颜色更少。但是在这种模式下，可以选择任何颜色，系统会建立一个调色板，这是一个从 160 万 RGB 颜色中选择出来的 256 种颜色列表。在调色板中指定了颜色后，图形设备就只显示所指定的这些颜色。调色板在任何时候都可以改变——但图形设备每次只能在屏幕上显示 256 种不同的颜色。当获得高性能和视频内存需要额外的费用时，才使用 256 色模式。大多数计算机游戏都使用这种模式——它们仍能得到相当好的图形，因为调色板经过了非常仔细的选择。

一般情况下，如果显示设备使用增强色或 256 色模式，并要显示某种 RGB 颜色，它就会从能显示的颜色池中选择一种在数学上最接近的匹配颜色。因此知道颜色模式是非常重要的。如果要绘制某些涉及微妙阴影区域或照片质量级的图像，而用户没有选择 24 位颜色模式，就看不到期望的效果。如果要使用 GDI+ 进行绘制，就应该用不同的颜色模式测试应用程序(应用程序也可以编程设置给定的颜色模式，但本章不讨论这个问题)。

#### 33.6.4 安全调色板

下面简要介绍一下安全调色板。这是一种非常常见的默认调色板。它工作的方式是为每种颜色成分设置 6 个间隔相等的值，这些值分别是 0, 51, 102, 153, 204, 255。换言之，红色成分可以是这些值中的任一个。绿色成分和蓝色成分也一样。所以安全调色板中的颜色就包括(0,0,0) (黑色)、(153,0,0) (暗红色)、(0, 255, 102) (蓝绿色)等，这样就得到了  $6^3=216$  种颜色。这是一种让调色板包含色谱中间隔相等的颜色和所有亮度的简单方式，但实际上这是不可行的，因为数学上等间隔的颜色成分并不表示这些颜色的区别在人眼看来也是相等的。

如果把 Windows 设置为 256 色模式，默认的调色板就是安全调色板，其中添加了 20 种标准的 Windows 颜色和 20 种备用颜色。

### 33.7 画笔和钢笔

本节介绍两个辅助类，在绘制图形时需要使用它们。前面已经见过 Pen 类了，它用于告诉 graphics 实例如何绘制线条。相关的类是 System.Drawing.Brush，它告诉 graphics 实例如何填充区域。例如，Pen 用于绘制前面示例中矩形和椭圆的边框。如果需要把这些图形绘制为实心的，就要使用画笔指定如何填充它们。这两个类有一个共同点：很难对它们调用任何方法。用需要的颜色和其他属性构造一个 Pen 或 Brush 实例，再把它传送给需要 Pen 或 Brush 的绘图方法即可。

注意：

如果读者以前使用 GDI 编程，可能会注意到在前两个示例中，在 GDI+ 中使用 Pen 的方式是不同的。在 GDI 中，一般是调用一个 Windows API 函数 SelectObject()，它把钢笔关联到设备环境上。这个钢笔用于所有需要钢笔的绘图操作中，直到再次调用 SelectObject() 通知设备环境停止使用它时为止。这个规则也适用于画笔和其他对象，例如字体和位图。而使用 GDI+，Microsoft 使用一种无状态的模式，其中没有默认的钢笔或其他帮助对象。只需给每个方法调用指定合适的帮助对象即可。

### 33.7.1 画笔

GDI+有几种不同类型的画笔,本章不准备详细介绍它们,这里仅解释几个比较简单的画笔,读者掌握其要领即可。每种画笔都由一个派生自抽象类 `System.Drawing.Brush` 的类实例来表示。最简单的画笔 `System.Drawing.SolidBrush` 仅指定了区域用纯色来填充:

```
Brush solidBeigeBrush = new SolidBrush(Color.Beige);
Brush solidFunnyOrangyBrownBrush =
    new SolidBrush(Color.FromArgb(255,155,100));
```

另外,如果画笔是一种 Web 安全颜色,就可以用另一个类 `System.Drawing.Brushes` 构造出画笔。`Brushes` 是永远不能实例化的一个类(它有一个私有构造函数,禁止实例化)。它有许多静态属性,每个属性都返回指定颜色的画笔。可以像下面这样使用画笔:

```
Brush solidAzureBrush = Brushes.Azure;
Brush solidChocolateBrush = Brushes.Chocolate;
```

比较复杂的一种画笔是影线画笔(hatch brush),它通过绘制一种模式来填充区域,这种类型的画笔比较高级,所以在 `Drawing2D` 命名空间中,用 `System.Drawing.Drawing2D.HatchBrush` 类表示。`Brushes` 类不能帮助我们使用影线画笔,而需通过提供一个影线型式和两种颜色(前景色和背景色,背景色可以忽略,此时将使用默认的黑色),来显式构造一个影线画笔。影线型式可以取自于枚举 `System.Drawing.Drawing2D.HatchStyle`,其中有许多 `HatchStyle` 值,其完整列表参阅 SDK 文档说明。为了让用户掌握这个概念,一般的型式包括 `ForwardDiagonal`、`Cross`、`Diagonal Cross`、`SmallConfetti` 和 `ZigZag`。构造影线画笔的示例如下所示。

```
Brush crossBrush = new HatchBrush(HatchStyle.Cross, Color.Azure);

// background color of CrossBrush is black

Brush brickBrush = new HatchBrush(HatchStyle.DiagonalBrick,
    Color.DarkGoldenrod, Color.Cyan);
```

GDI 只能使用实线和影线画笔, GDI+ 添加了两种新画笔:

- `System.Drawing.Drawing2D.LinearGradientBrush` 用一种在屏幕上可变的颜色填充区域。
- `System.Drawing.Drawing2D.PathGradientBrush` 与此类似,但其颜色沿着要填充的区域的路径而变化。

如果细心使用,这些画笔会得到一些惊人的效果。

### 33.7.2 钢笔

与画笔不同,钢笔只用一个类 `System.Drawing.Pen` 来表示。但钢笔比画笔复杂一些,因为它需要指定线条应有多宽(像素),对于一条比较宽的线段,还要确定如何填充该线条中的区域。钢笔还可以指定其他许多属性,本章不讨论它们,其中包括前面提到的 `Alignment` 属性,该属性表示相对于图形的边框,线条该如何绘制,以及在线条的末尾绘制什么图形(是否使图形平滑过渡)。

粗线条中的区域可以用纯色填充,或者使用画笔来填充。因此 `Pen` 实例可以包含 `Brush` 实例的引用。这是非常强大的,因为这表示我们可以绘制有影线填充或线性阴影的线条。构造 `Pen`

实例有四种不同的方式：可以传送一种颜色，或者传送一个画笔。这两个构造函数都会生成一个像素宽的钢笔。另外，还可以传送一种颜色或画笔，以及一个表示钢笔宽度的 float 类型的值。（该宽度必须是一个 float 类型的值，以允许执行绘图操作的 Graphics 对象使用非默认的单位，例如毫米或英寸，例如可以指定宽度是英寸的某个分数）。例如，可以构造如下的钢笔：

```
Brush brickBrush = new HatchBrush(HatchStyle.DiagonalBrick,
                                   Color.DarkGoldenrod, Color.Cyan);

Pen solidBluePen = new Pen(Color.FromArgb(0,0,255));
Pen solidWideBluePen = new Pen(Color.Blue, 4);
Pen brickPen = new Pen(brickBrush);
Pen brickWidePen = new Pen(brickBrush, 10);
```

另外，为了快速构造钢笔，还可以使用类 System.Drawing.Pens，它与 Brushes 类一样，包含许多存储好的钢笔。这些钢笔的宽度都是一个像素，使用通常的 Web 安全颜色，这样就可以用下述方式构建一个钢笔：

```
Pen solidYellowPen = Pens.Yellow;
```

33.8 绘制图形和线条

前面介绍了在屏幕上绘制规定的图形所需要的所有基类和对象。下面复习一些 Graphics 类可以使用的绘图方法，用一个小示例来介绍几个画笔和钢笔。

System.Drawing.Graphics 有很多方法，利用这些方法可以绘制各种线条、空心图形和实心图形。表 33-4 所示的列表并不完整，但给出了主要的方法，您应能据此掌握绘制各种图形的方法。

表 33-4

方 法	常 见 参 数	绘制的图形
DrawLine	钢笔、起点和终点	一段直线
DrawRectangle	钢笔、位置和大小	空心矩形
DrawEllipse	钢笔、位置和大小	空心椭圆
FillRectangle	画笔、位置和大小	实心矩形
FillEllipse	画笔、位置和大小	实心椭圆
DrawLines	钢笔、点数组	一组线，把数组中的每个点按顺序连接起来
DrawBezier	钢笔、4 个点	通过两个端点的一条光滑曲线，剩余的两个点用于控制曲线的形状
DrawCurve	钢笔、点数组	通过点的一条光滑曲线
DrawArc	钢笔、矩形、两个角	由角度定义的矩形中圆的一部分
DrawClosedCurve	钢笔、点数组	与 DrawCurve 一样，但还要绘制一条用以闭合曲线的直线
DrawPie	钢笔、矩形、两个角	矩形中的空心楔形
FillPie	画笔、矩形、两个角	矩形中的实心楔形
DrawPolygon	钢笔、点数组	与 DrawLines 一样，但还要连接第一点和最后一点，以闭合绘制的图形

在结束绘制简单对象的主题前，用一个简单示例来说明使用画笔可以得到的各种可视效果。该示例是 `ScrollMoreShapes`，它是 `ScrollShapes` 的修正版本。除了矩形和椭圆外，我们还添加了一条粗线，用各种定制的画笔填充图形。前面解释了绘图规则，所以这里只给出代码，而不作过多的注释。首先，因为添加了新画笔，所以需指定使用命名空间 `System.Drawing.Drawing2D`：

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Text;
using System.Windows.Forms;
```

接着是 `Form1` 类中的一些额外字段，其中包含了要绘制图形的位置信息，以及要使用的各种钢笔和画笔：

```
private Rectangle rectangleBounds = new Rectangle(new Point(0,0),
    new Size(200,200));
private Rectangle ellipseBounds = new Rectangle(new Point(50,200),
    new Size(200,150));
private Pen bluePen = new Pen(Color.Blue, 3);
private Pen redPen = new Pen(Color.Red, 2);
private Brush solidAzureBrush = Brushes.Azure;
private Brush solidYellowBrush = new SolidBrush(Color.Yellow);
static private Brush brickBrush = new HatchBrush(HatchStyle.DiagonalBrick,
    Color.DarkGoldenrod, Color.Cyan);
private Pen brickWidePen = new Pen(brickBrush, 10);
```

把 `BrickBrush` 字段声明为静态，就可以使用该字段的值初始化 `BrickWidePen` 字段了。C# 不允许使用一个实例字段初始化另一个实例字段，因为还没有定义要先初始化哪个实例字段，如果把字段声明为静态字段就可以解决这个问题，因为只实例化了 `Form1` 类的实例，字段是静态字段还是实例字段就不重要了。

下面是 `OnPaint()` 重写方法：

```
protected override void OnPaint( PaintEventArgs e )
{
    base.OnPaint(e);
    Graphics dc = e.Graphics;
    Point scrollOffset = this.AutoScrollPosition;
    dc.TranslateTransform(scrollOffset.X, scrollOffset.Y);

    if (e.ClipRectangle.Top+scrollOffset.X < 350 ||
        e.ClipRectangle.Left+scrollOffset.Y < 250)
    {
        dc.DrawRectangle(bluePen, rectangleBounds);
        dc.FillRectangle(solidYellowBrush, rectangleBounds);
        dc.DrawEllipse(redPen, ellipseBounds);
        dc.FillEllipse(solidAzureBrush, ellipseBounds);
        dc.DrawLine(brickWidePen, rectangleBounds.Location,
            ellipseBounds.Location+ellipseBounds.Size);
    }
}
```

与以前一样，也将 `AutoScrollMinSize` 设置为(250,350)，结果如图 33-13 所示。



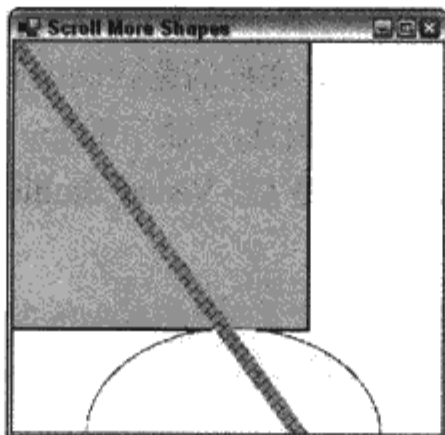


图 33-13

注意，粗对角线已在矩形和椭圆上绘制出来，它需要最后一个绘制。

### 33.9 显示图像

使用 GDI+ 最常想做的是显示文件中已有的图像。这确实要比绘制自己的用户界面简单多了，因为图像已经绘制好了。我们只需要加载文件，让 GDI+ 显示它即可。图像可以只包含一个线条或一个图标，也可以比较复杂，例如一张照片。对图像也可以执行某些操作，例如拉伸或旋转图像，也可以选择只显示图像的一部分。

本节将先给出一个示例，再讨论显示图像时需要注意的一些问题。可以这么做的原因是显示图像的代码是非常简单的。

我们需要 .NET 的一个基类 `System.Drawing.Image`。Image 实例表示一个图像。读取图像仅需用使用一行代码：

```
Image myImage = Image.FromFile("FileName");
```

`FromFile()` 是 `Image` 的一个静态成员，是实例化图像的常用方式。文件可以是任何支持的图像文件格式，包括 `.bmp`、`.jpg`、`.gif` 和 `.png`。

显示图像是很简单的，假定有一个合适的 `Graphics` 实例，则调用 `Graphics.DrawImage Unscaled()` 或 `Graphics.DrawImage()` 就足够了。这些方法都有许多重载方法，可以根据图像的位置和要绘制的大小非常灵活地处理用户提供的信息。下面要使用 `DrawImage()`：

```
dc.DrawImage(myImage, points);
```

在这行代码中，假定 `dc` 是一个 `Graphics` 实例，`MyImage` 是要显示的图像，`points` 是一个 `Point` 结构数组，其中 `points[0]`、`points[1]` 和 `points[2]` 是图像左上角、右上角和左下角的坐标。

**注意：**

熟悉 GDI 的开发人员可以从图像中看出 GDI 与 GDI+ 的最大区别。在 GDI 中，显示图像涉及到几个重要的步骤。如果图像是一个位图，加载它是很简单的，但如果它是其他类型的文件，加载它会涉及一系列 OLE 对象的调用。把加载的图像显示到屏幕上要获得它的一个句柄，把它放在一个内存设备环境中，然后在设备环境之间执行一个块传输。设备环境和句柄仍在后台上，但如果要开始在代码中对图像进行复杂的编辑，就需要它们。简单的任务封装在 GDI+ 对象模型上。



下面用一个示例 `DisplayImage` 来说明显示图像的过程。这个示例在应用程序的主窗口中显示.jpg 文件。要使操作过程简单一些，.jpg 文件的路径硬编码到应用程序中(如果运行该示例，就需要改变该路径，以反映系统中文件的位置)。显示的.jpg 文件是圣彼得堡的日落图片。

与其他例子一样，`DisplayImage` 项目是 C# Visual Studio 2008 生成的一个标准的 Windows 应用程序。在 `Form1` 类中添加下述字段：

```
Image piccy;
private Point [] piccyBounds;
```

然后在 `Form1()` 构造函数中加载文件：

```
public Form1()
{
    InitializeComponent();

    piccy =
        Image.FromFile(@"C:\ProCSharp\GdiPlus\Images\London.bmp");
    this.AutoScrollMinSize = piccy.Size;
    piccyBounds = new Point[3];
    piccyBounds[0] = new Point(0,0); // top left
    piccyBounds[1] = new Point(piccy.Width,0); // top right
    piccyBounds[2] = new Point(0,piccy.Height); // bottom left
}
```

注意，图像的大小(像素)是通过其 `Size` 属性来获得的，我们使用该属性设置文档区域。再建立一个 `piccyBounds` 数组，用于指定图像在屏幕上的位置。选择 3 个角的坐标，按实际大小来绘制图像，但如果要重新设置图像的大小、拉伸图像，或把图像变形为非矩形的平行四边形，可以改变 `piccyBounds` 数组中 `Point` 的值。

通过 `OnPaint()` 重写方法显示图像：

```
protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);
    Graphics dc = e.Graphics;
    dc.ScaleTransform(1.0f, 1.0f);
    dc.TranslateTransform(this.AutoScrollPosition.X, this.AutoScrollPosition.Y);
    dc.DrawImage(piccy, piccyBounds);
}
```

最后，特别注意对向导生成的 `Form1.Dispose()` 方法代码进行修改：

```
protected override void Dispose(bool disposing)
{
    piccy.Dispose();
    if (disposing)
    {
        if (components != null)
        {
            components.Dispose();
        }
    }
    base.Dispose(disposing);
}
```

只要不再需要该图像了，就应立即删除它，因为图像在使用时一般会占用许多内存。在调

用 `Image.Dispose()` 后, `Image` 实例就不再引用任何图像, 所以不再显示图像(除非加载了一个新图像)。

运行代码, 会得到如图 33-14 所示的结果。



图 33-14

### 33.10 处理图像时的问题

显示图像是很简单的, 但需要理解一些底层技术。

理解图像最重要的一点是, 图像总是矩形的。这不只是方便了人们, 其原因是其底层的技术。所有的现代图形卡都内置了硬件, 可以非常高效地从内存的一个地方把像素块复制到另一个地方。假定像素块表示一个矩形区域, 这个硬件加速操作可以虚拟为一个操作, 而且执行速度非常快。实际上, 这是现代高性能图像的关键。这个操作称为位图块传输(或者 `BitBlt`)。`Graphics.DrawImageUnscaled()` 在内部使用 `BitBlt`, 这就是为什么能够看见一个大图像的原因, 该图像也许包含上百万个像素, 但几乎是立即就显示出来。如果计算机必须把图像一个像素一个像素地复制到屏幕上, 该图像只能在几秒钟内逐渐画出来。

`BitBlt` 的效率非常高, 所以图像的所有绘制和处理操作都是使用 `BitBlt` 完成的。甚至图像的某些编辑也是使用表示内存区域的设备环境之间图像的 `BitBlt` 完成的。在 GDI 中, Windows 32 API 函数 `BitBlt()` 是最重要、使用最广泛的图像处理函数, 而在 GDI+ 中, `BitBlt` 操作一般隐藏在 GDI+ 对象模型中。

图像的 `BitBlt` 区域不可能是矩形的, 但很容易模拟类似的效果。一种方式是为了 `BitBlt`, 把某种颜色标记为透明的。这样源图像中该颜色的区域不会覆盖目的设备中对应区域的现有颜色。在 `BitBlt` 过程中还可以指定, 结果图像的每个像素会在进行 `BitBlt` 前, 对源图像上和目的

设备上该像素的颜色进行某些逻辑操作来形成(例如按位 AND)。这样的操作是由硬件加速来支持的,用于产生各种微妙的效果。注意 Graphics 对象执行另一个方法 DrawImage(),该方法类似于 DrawImageUnscaled(),但它有许多重载方法,可以指定 BitBlt 更复杂的形式,以便在绘图过程中使用。DrawImage()还可以只绘制图像的某个特定部分,或者对它执行其他操作,例如在绘图时缩放(缩放其大小)它。

### 33.11 绘制文本

到目前为止,本章还有一个非常重要的问题要讨论——显示文本。因为在屏幕上绘制文本通常比绘制简单图形更复杂。在不考虑外观的情况下,只显示一两行文本是非常简单的——它只需调用 Graphics 实例的一个方法 Graphics.DrawString()。但如果要显示一个文档,其中有许多文本,则事情就变得复杂多了,这有两个原因:

- 如果只考虑外观,则需要理解字体。图形的绘制需要使用画笔和钢笔作为帮助对象,绘制文本的过程则需要把字体作为帮助对象。理解字体并不容易。
- 在窗口中需要仔细布局文本。用户通常期望文字一个跟一个地排列——排成一行,其间有一定的间隔。这是一个比想象中更困难的任务。开始时,一般事先不知道屏幕上文字之间的间隔有多大。这需要计算(使用方法 Graphics.MeasureString())。另外,屏幕上文字占用的间隔会影响文档中后续的文字在屏幕上的显示位置。如果应用程序自动换行,就需要在确定应在何处断开前仔细斟酌文字的大小。下次运行 Windows 的 Word 时,仔细看看 Word 是如何重新定位用户键入的文本的。这里有许多复杂的处理操作。任何 GDI+应用程序都不像 Word 那样复杂,但如果需要显示文本,仍需要考虑同样的问题。

总之,好的文本处理需要一定的技巧。假定知道字体和显示的位置,把一行文本显示在屏幕上的过程就非常简单。因此,下面介绍一个小示例,说明如何显示一些文本。在此之后,探讨字体和字体系列的一些规则,然后介绍一个非常真实的文本处理示例 CapsEditor。

### 33.12 简单的文本示例

这个示例 DisplayText 是常见的 Windows Forms。这次重写了 OnPaint(), 添加了成员字段,如下所示:

```
private Brush blackBrush = Brushes.Black;
private Brush blueBrush = Brushes.Blue;
private Font haettenschweilerFont = new Font("Haettenschweiler", 12);
private Font boldTimesFont = new Font("Times New Roman", 10, FontStyle.Bold);
private Font italicCourierFont = new Font("Courier", 11, FontStyle.Italic |
    FontStyle.Underline);

protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);
    Graphics dc = e.Graphics;
```

```

dc.DrawString("This is a groovy string", haettenschweilerFont, blackBrush,
    10, 10);
dc.DrawString("This is a groovy string " +
    "with some very long text that will never fit in the box",
    boldTimesFont, blueBrush,
    new Rectangle(new Point(10, 40), new Size(100, 40)));
dc.DrawString("This is a groovy string", italicCourierFont, blackBrush,
    new Point(10, 100));
}

```

运行这个示例，会得到如图 33-15 所示的结果。

这个示例说明了如何使用 `Graphics.DrawString()` 方法绘制文本，`DrawString()` 有许多重载方法，这里介绍其中的 3 个。但这些重载方法都需要用参数指定要显示的文本、字符串所使用的字体，以及用于构造各种直线和曲线以组成每个文本字符的画笔。其余的参数有另外两种指定方式。但一般情况下，可以指定一个 `Point` (或两个数字) 或一个 `Rectangle`。

如果指定 `Point`，文本就从该 `Point` 的左上角开始，并向右延伸。如果指定 `Rectangle`，则 `Graphics` 实例就把字符串放在矩形的内部。如果文本在矩形内部容纳不下，就会被剪切，如图 33-15 中的第四行文本所示。把矩形传送给 `DrawString()`，表示绘图过程将持续较长时间，因为 `DrawString()` 需要指定在什么地方放置换行符，但其结果应看起来好一些(如果字符串可以放在矩形中)。



图 33-15

这个示例还说明了构造字体的两种方法，一般需要字体的名称及其大小(高度)。也可以选择传送不同的样式以修改文本的绘制方式(黑体、下划线等)。

### 33.13 字体和字体系列

字体描述的是每个字母的显示方式。在一个文档中选择合适的字体和提供适当数量的不同字体，对于提高该文档的可读性是非常重要的。

大多数人在给字体命名时，会指定 `Arial`、`Times New Roman` (Windows 用户) 或 `Times`、`Helvetica` (Mac OS 用户) 等名称。实际上，这些都不是字体，它们是字体系列(font families)，字体系列以通俗的话来说，是文本的可视化风格。字体系列是应用程序整体外观中的一个关键因



素,大多数人都习惯于识别常用字体系列的风格,甚至我们已经意识不到这一点了。

而字体应是像 Arial 9-point italic 这样的东西。换言之,字体添加了更多的信息,例如指定文本的大小,以及是否对该文本进行某种修改等,如文本是否为黑体、斜体、下划线,或者显示为小型大写字母或下标,这种修改在技术上称为风格,但在某些情况下该术语有误导作用,因为可视化外观主要取决于字体系列。

通过指定文本的高度,就可以测量其大小。高度以点为单位来测量,这是一个传统单位,表示 1/72 英寸(0.351mm)。例如,10 点的字母字体高度大约为 1/7 英寸或 3.5mm。但字体大小为 10 点的 7 行文本,在屏幕或纸上的高度不等于 1 英寸,因为还需要考虑行与行之间的间隔。

#### 注意:

严格来讲,测量高度并不像这样简单,因为还需要考虑几个不同的高度。例如,较高字母,如 A 或 F 的高度(这是指我们讨论高度时字母的真实高度),重音字母如 Å 或 Ñ 中的额外高度(内部前导),以及字母的尾部超出底线的部分高度如 y 和 g(下行高度)。但是本章不讨论这些高度,一旦指定了字体系列和主要高度,这些高度就会自动确定。

在处理字体时,还会遇到其他常用于描述某种字体系列的术语:

- Serif 字体系列在组成字符的许多线条尾部有一个小标记(这些标记称为 serif)。Times New Roman 就是这种字体的一个典型例子。
- 相反, Sans Serif 字体系列没有这些小标记。例如 Arial 和 Verdana。没有小标记常会使文本看起来比较生硬,所以 Sans Serif 字体常用于重要的文本。
- True Type 字体系列以一种精确的数学方式定义组成字符的曲线形状。这表示可以使用相同的定义来计算如何在系列内部绘制任何大小的字体。目前,我们实际上使用的所有字体都是 True Type 字体。Windows 3.1 时代的一些旧字体系列是通过指定每种字体大小的每个字符位图来定义的,但现在最好不要使用这些字体。

Microsoft 提供了两个类来处理何时选择或处理字体:

- System.Drawing.Font
- System.Drawing.FontFamily

前面已经介绍了 Font 类的主要用法。在绘制文本时,实例化 Font 的一个实例,并把它传送给 DrawString 方法,以确定应该如何绘制文本。FontFamily 实例用于表示一个字体系列。

FontFamily 类的一个用法是:如果知道需要某种类型的字体(Serif、SansSerif 或 Monospace),但不介意使用哪种字体,就可以使用该类。静态属性 properties GenericSerif, GenericSansSerif 和 GenericMonospace 返回满足这些条件的默认字体:

```
FontFamily sansSerifFont = FontFamily.GenericSansSerif;
```

但一般来说,如果编写一个专业应用程序,就应以更专业的方式选择字体。可以执行绘图代码,检查哪些字体可用。然后选择合适的字体,例如从预定义的字体列表中选择第一个可用的字体。如果能让应用程序的用户友好性更高,列表中的第一个选项可能是用户上次运行软件时选择的字体。通常情况下,最好使用最常用的字体系列,例如 Arial 和 Times New Roman。但如果要使用某种不存在的字体显示文本,结果通常是不可预料的,Windows 仅是替代了标准系统字体,系统很容易绘制出文本,但看起来并不是非常友好,如果文档出现这种情况,会使





```
using System.Drawing.Text;
```

然后在 Form1 类中添加如下常量:

```
private const int margin = 10;
```

margin 是文本与文档边缘之间的左边距和上边距的大小——它防止文本显示在客户区域边缘的右边。

因此, 该示例以快捷方式显示字体系列, 代码比较粗糙, 在许多情况下并不是以真正应用程序中的方式执行任务。例如, 我们为文档的大小硬编码了一个估计值(200,1500), 使用 Visual Studio 2008 属性窗口把 AutoScrollMinSize 属性设置为这个值。一般情况下必须查看要显示的文本, 计算出文档的大小。下一节介绍这个过程。

下面是 OnPaint()方法:

```
protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);
    int verticalCoordinate = margin;
    InstalledFontCollection insFont = new InstalledFontCollection();
    FontFamily [] families = insFont.Families;
    e.Graphics.Transform(AutoScrollPosition.X,
                        AutoScrollPosition.Y);
    foreach (FontFamily family in families)
    {
        if (family.IsStyleAvailable(FontStyle.Regular))
        {
            Font f = new Font(family.Name, 12);
            Point topLeftCorner = new Point(margin, verticalCoordinate);
            verticalCoordinate += f.Height;
            e.Graphics.DrawString (family.Name, f,
                                Brushes.Black, topLeftCorner);
            f.Dispose();
        }
    }
}
```

在这段代码中, 首先使用 InstalledFontCollection 对象获得一个数组, 其中包含所有可用的字体系列。对于每个系列, 我们实例化大小为 12 点的一个 Font。为 Font 使用了一个简单的构造函数——有许多构造函数可以指定更多的选项。我们使用的构造函数带有两个参数: 系列名称和字体的大小:

```
Font f = new Font(family.Name, 12);
```

这个构造函数构造了一个一般风格的字体。但是为了安全起见, 在使用该字体显示文本前, 先检查一下这种风格是否可用于每个字体系列, 这是利用方法 FontFamily.IsStyleAvailable() 实现的, 这种检查是非常重要的, 因为并不是所有的字体都可以使用所有的风格:

```
if (family.IsStyleAvailable(FontStyle.Regular))
```

FontFamily.IsStyleAvailable()带一个参数, 即 FontStyle 枚举。该枚举包含许多标记, 它们可以用按位 OR 运算符来组合。可能的标记有 Bold、Italic、Regular、Strikeout 和 Underline。

最后, 使用 Font 类的一个属性 Height, 该属性返回显示该字体文本需要的高度, 以便算出

行间距：

```
Font f = new Font(family.Name, 12);
Point topLeftCorner = new Point(margin, verticalCoordinate);
verticalCoordinate += f.Height;
```

为了使事情简单化，OnPaint()的这个版本暴露出一些不太好的编程问题。首先，没有检查哪些文档区域需要绘制——而是显示所有的内容。另外，如前所述，实例化 Font 是一个计算密集型的过程，所以应保存字体，而不是每次调用 OnPaint()时都实例化新副本。这样设计出来的代码将需要花费相当长的时间来绘图。为了节省内存，帮助垃圾收集器，在完成操作后对每个 Font 实例调用 Dispose()。如果不这样，在 10 或 20 次绘图操作后，就会存在许多存储不再需要的字体的内存。

### 33.15 编辑文本文档：CapsEditor 示例

下面是本章中一个比较大的示例。CapsEditor 示例用于说明如何把前面介绍的绘图规则应用到真正的示例中。这个示例除了响应用户通过鼠标输入的信息外，不需要任何新资料，但它将说明如何管理文本的绘制，让应用程序具有高性能，并确保主窗口客户区域的内容总是最新的。

CapsEditor 程序允许用户读取文本文件，该文本逐行显示在客户区域中。如果用户双击任何一行文本，该行就会全部变为大写。这就是该示例的功能。即使只有这个有限的功能，也涉及到许多复杂的工作：确保把所有的信息都显示在正确的位置上，并考虑性能问题。特别是这里有一个新元素，文档的内容可以修改——用户选择菜单项，读取一个新文件，或用户双击一行，使之变为大写字母时，都要修改文件的内容。在第一种情况下，需要更新文档的大小，使滚动条仍能正确工作，并重新显示所有的内容。在第二种情况下，需要仔细检查文档的大小是否发生了变化，哪些文本需要重新显示。

首先看看 CapsEditor 的外观。第一次运行该应用程序时，没有加载文档，显示结果如图 33-17 所示。

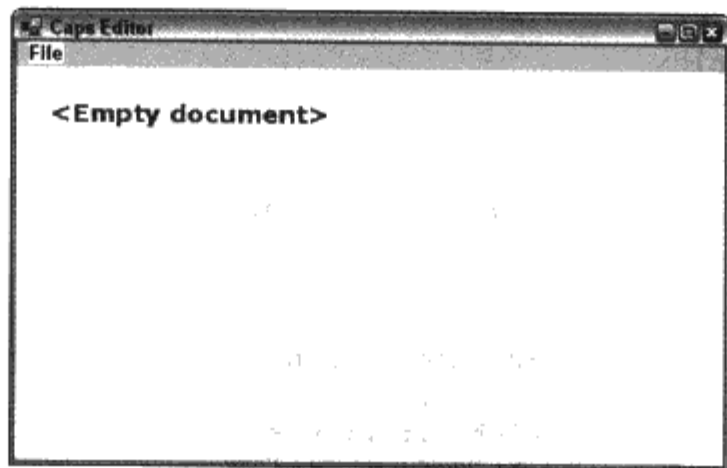


图 33-17

File 菜单有两个菜单项：Open 和 Exit。Exit 退出应用程序，Open 调用 OpenFileDialog，读取用户选择的任何文件。图 33-18 是 CapsEditor 显示其源文件 Form1.cs 的情形(我们已经双击两行，把它们转换为大写)。

水平和垂直滚动条的大小也是正确的。滚动客户区域，使之正好显示整个文档。CapsEditor 不会给文本换行——即使没有这个功能，该示例也比较复杂了。它只显示文件被读取的各行文本。对文件的大小没有限制，但这里假定这是一个文本文件，不包含任何非打印字符。

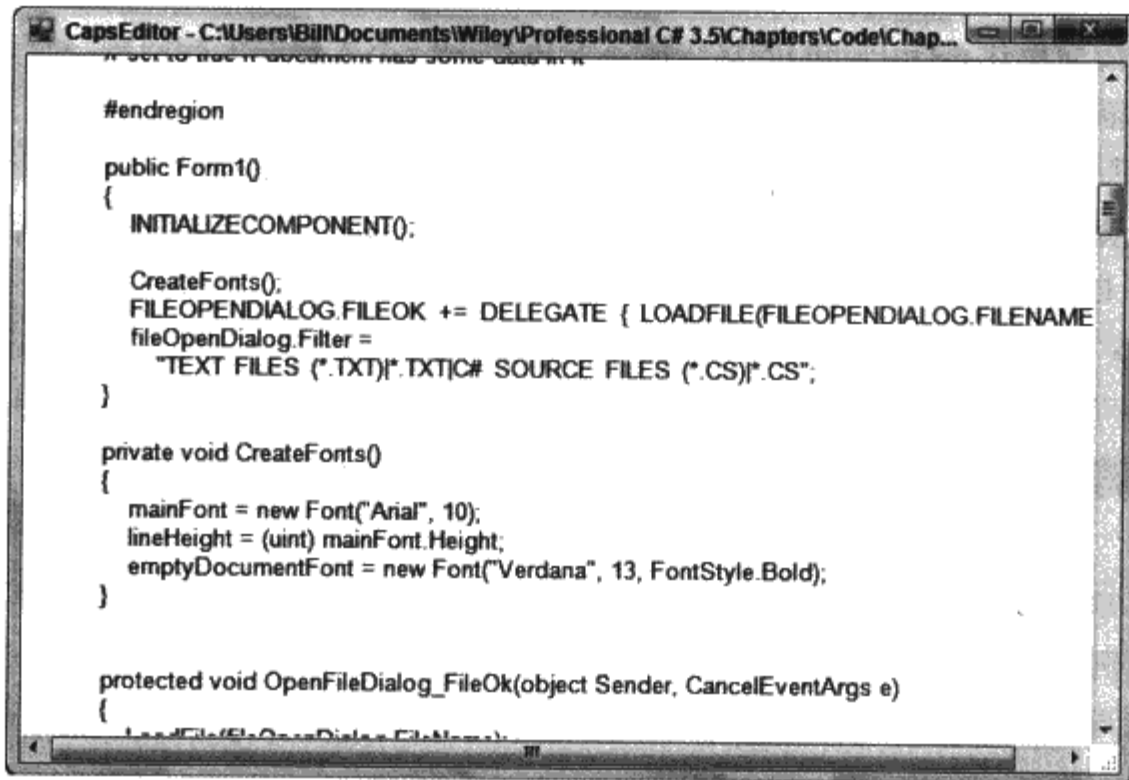


图 33-18

首先添加一些 using 命令：

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Drawing;
using System.Windows.Forms;
using System.IO;
```

这是因为我们使用了 System.IO 命名空间中的 StreamReader 类。接着在 Form1 类中添加一些字段：

```
#region Constant fields
private const string standardTitle = "CapsEditor";
// default text in titlebar
private const uint margin = 10;
// horizontal and vertical margin in client area
#endregion
#region Member fields
// The 'document'
private readonly List < TextLineInformation > documentLines =
new List < TextLineInformation > ();
private uint lineHeight; // height in pixels of one line
private Size documentSize; // how big a client area is needed to
// display document
private uint nLines; // number of lines in document
private Font mainFont; // font used to display all lines
private Font emptyDocumentFont; // font used to display empty message
private readonly Brush mainBrush = Brushes.Blue;
// brush used to display document text
```

```
private readonly Brush emptyDocumentBrush = Brushes.Red;
    // brush used to display empty document message
private Point mouseDoubleClickPosition;
    // location mouse is pointing to when double-clicked
private readonly OpenFileDialog fileOpenDialog = new OpenFileDialog();
    // standard open file dialog
private bool documentHasData = false;
    // set to true if document has some data in it
#endregion
```

这些字段中的大多数都很容易理解。documentLines 字段是一个 List < TextLineInformation >, 包含文件中已经读入的实际文本。实际上, 这个字段包含了文档中的数据。documentLines 的每个元素都包含一行已读入的文本的信息。它是一个 List < TextLineInformation >, 而不是常见的 C# 数组, 所以在读取文件时, 可以给它动态地添加元素。

如前所述, 每个 documentLines 元素都包含一行文本信息。这些信息实际上是另一个类 TextLineInformation 的实例:

```
class TextLineInformation
{
    public string Text;
    public uint Width;
}
```

TextLineInformation 看起来像在使用结构, 而不是类, 因为它只是把几个字段组合在一起。但它的实例总是作为 List < TextLineInformation > 的元素来访问的, 这样其元素就会存储为引用类型。

每个 TextLineInformation 实例都存储了一行文本——它可以看作是显示为一个项的最小单元, 一般情况下, 在 GDI+ 应用程序中, 对于每个这样的项, 都要存储该项的文本, 以及显示它的世界坐标和其大小。(只要用户进行了滚动操作, 页面坐标就会改变, 而世界坐标通常只有在文档的其他部分以某种方式进行了修改时才会改变)。本例只需存储项的 Width 即可。其原因是此时的高度是选中字体的高度, 它对于所有的文本行都一样, 所以不必为每行文本存储高度。它只需要在字段 Form1.lineHeight 中存储一次即可, 位置也是这样。在这里, x 坐标等于页边距, y 坐标很容易计算出来:

```
margin + lineHeight * (本行上面显示的行数)
```

如果要显示和操作单个单词, 而不是整行文本, 则每个单词的 x 坐标就必须使用该文本行上在该单词前面的所有单词的宽度来计算, 但为了使这个计算简单一些, 应把每行文本当作单一的项来看待。

下面处理主菜单。这部分应用程序是真正的 Windows 窗体(参见第 31 章), 而不是 GDI+。在 Visual Studio 2008 中使用设计视图添加菜单项, 把它们重新命名为 menuFile、menuFileOpen 和 menuFileExit。接着使用 Visual Studio 2008 属性窗口添加 File Open 和 File Exit 菜单项的事件处理程序。这些事件处理程序的名称是 Visual Studio 2008 生成的, 即 menuFileOpen\_Click() 和 menuFileExit\_Click()。

还需要在 Form1() 构造函数中添加一些初始化代码:

```
public Form1()
{
```



```
InitializeComponent();
```

```
CreateFonts();
fileOpenDialog.FileOk += delegate { LoadFile(fileOpenDialog.FileName); };
fileOpenDialog.Filter =
    "Text files (*.txt)|*.txt|C# source files (*.cs)|*.cs";
}
```

这里添加的事件处理程序是在用户单击 FileOpen 对话框中的 OK 按钮时执行的。我们还给“打开文件”对话框设置了过滤器，只能加载文本文件。由于本示例选择了.txt 文件和 C#源文件，所以可以使用应用程序查看示例的源代码。

CreateFonts()是一个帮助方法，它挑选出要使用的字体：

```
private void CreateFonts()
{
    mainFont = new Font("Arial", 10);
    lineHeight = (uint)mainFont.Height;
    emptyDocumentFont = new Font("Verdana", 13, FontStyle.Bold);
}
```

处理程序的实际定义是标准的：

```
protected void OpenFileDialog_FileOk(object Sender, CancelEventArgs e)
{
    this.LoadFile(fileOpenDialog.FileName);
}

protected void menuFileOpen_Click(object sender, EventArgs e)
{
    fileOpenDialog.ShowDialog();
}

protected void menuFileExit_Click(object sender, EventArgs e)
{
    this.Close();
}
```

下面介绍 LoadFile()方法。这个方法处理文件的打开和读取操作，并确保引发 Paint 事件，使用新文件重新绘图：

```
private void LoadFile(string FileName)
{
    StreamReader sr = new StreamReader(FileName);
    string nextLine;
    documentLines.Clear();
    nLines = 0;
    TextLineInformation nextLineInfo;
    while ( (nextLine = sr.ReadLine()) != null)
    {
        nextLineInfo = new TextLineInformation();
        nextLineInfo.Text = nextLine;
        documentLines.Add(nextLineInfo);
        ++nLines;
    }
    sr.Close();
    documentHasData = (nLines>0) ? true : false;
```

```

        CalculateLineWidths();
        CalculateDocumentSize();

        this.Text = standardTitle + " - " + FileName;
        this.Invalidate();
    }

```

这个功能的大部分都是标准的文件读取过程(参见第 25 章)。注意在读取文件时,给 documentLines ArrayList 添加文本行,使这个数组最终按顺序包含每行文本的信息。在读取文件后,设置 documentHasDat 标记,指定是否要显示信息。下一个任务是确定要显示内容的位置,之后确定显示文件的客户区域有多大——即用于设置滚动条的文档大小。最后,设置标题栏文本,并调用 Invalidate()。Invalidate()是 Microsoft 提供的一个非常重要的方法,所以下一节介绍这个方法的应用,之后介绍 CalculateLineWidths()和 CalculateDocumentSize()方法的代码。

### 33.15.1 Invalidate()方法

Invalidate()是 System.Windows.Forms.Form 的一个成员,它把客户窗口区域标记为无效,因此在需要重新绘制时,它可以确保引发 Paint 事件。Invalidate()有两个重载方法:可以给它传送一个矩形,指定(使用页面坐标)需要重新绘制哪个窗口区域,如果不提供任何参数,它就把整个客户区域标记为无效。

如果知道需要绘制某些内容,为什么不调用 OnPaint()或直接完成绘制任务的其他方法?一般情况下,最好不要直接调用绘图例程,如果代码要完成某些绘图任务,一般应调用 Invalidate()。其原因如下所示:

- 绘图总是 GDI+应用程序执行的一种处理器密集型的任务。在其他工作的中间进行绘图会妨碍其他工作的进行。在前面的示例中,如果在 LoadFile()方法中直接调用一个方法来完成绘图,LoadFile()方法就将在绘图工作完成后才能返回。在这段时间里,应用程序不会响应其他事件。另一方面,通过调用 Invalidate(),在从 LoadFile 返回之前,就可以让 Windows 引发一个 Paint 事件。接着 Windows 就可以检查等待处理的事件了。其内部的工作方式是事件被当作消息队列中一个消息。Windows 会定期检查该队列,如果其中有事件,Windows 就选择它,并调用相应的事件处理程序。现在 Paint 事件是队列中的唯一事件,所以 OnPaint()会被立即调用。但是,在一个比较复杂的应用程序中,可能会有其他事件,其中一些的优先权比 OnPaint()高。特别是如果用户已决定退出应用程序,该事件就会用消息 WM\_QUIT 来标记。
- 如果有一个比较复杂的多线程应用程序,就会希望用一个线程处理所有的绘图操作。使用 Invalidate()可以把所有的绘图操作传递到消息队列中,这有助于确保无论其他线程请求什么绘图操作,都由同一个线程完成所有的绘图操作(无论什么线程负责消息队列,都是由线程 Application.Run()处理绘图操作)。
- 还有一个与性能有关的原因。假定在某一时刻有几个不同的屏幕绘制请求,也许代码仅能修改文档,以确保显示更新的文档,而同时用户刚刚移开另一个覆盖部分客户区域的窗口。调用 Invalidate(),可以让 Windows 注意到发生的事件。Windows 就会在需要时合并 Paint 事件,合并无效的区域,这样绘图操作就只执行一次。

- 最后，执行绘图的代码可能是应用程序中最复杂的代码部分，特别是当有一个比较专业化的用户界面时，就更是如此。需要长时间维护该代码的人员希望我们把所有的绘图代码都放在一个地方，且尽可能简单——如果程序的其他部分没有过多的路径进入该代码部分，维护就更容易。

其底线是最好把所有的绘图代码都放在 `OnPaint()` 例程中，或者在该方法中调用的其他方法。但是要维持一个平衡。如果要在屏幕上替换一个字符，最好不要影响到已经绘制好的其他内容，此时可能不需要使用 `Invalidate()`，而只需编写一个独立的绘图例程。

注意：

在非常复杂的应用程序中，甚至可以编写一个完整的类，专门负责在屏幕上绘图。几年前 MFC 仍是 GDI 密集型应用程序的标准技术，MFC 就遵循这个模式，使用一个 C++ 类 `C<ApplicationName>View` 完成绘图操作。但即使是这样，这个类也有一个成员函数 `OnDraw()`，用作大多数绘图请求的入口点。

### 33.15.2 计算项和文档的大小

下面返回 CapsEditor 示例，介绍在 `LoadFile()` 中调用的 `CalculateLineWidths()` 和 `CalculateDocumentSize()` 方法：

```
private void CalculateLineWidths()
{
    Graphics dc = this.CreateGraphics();
    foreach (TextLineInformation nextLine in documentLines)
    {
        nextLine.Width = (uint)dc.MeasureString(nextLine.Text, mainFont).Width;
    }
}
```

这个方法仅遍历已经读取的每行文本，使用 `Graphics.MeasureString()` 方法处理和存储字符串需要的水平间距。存储这个值是因为 `MeasureString()` 要进行大量的计算。如果没有把 CapsEditor 示例编写得非常简单，就不能很容易地计算出每一行的高度和位置，而这个方法也肯定需要按计算所有项的方式来实现。

知道屏幕上每个项的大小后，就可以计算每个项的位置，并计算出文档的大小。高度基本上是每行文本的高度乘以行数。宽度则需要计算，遍历每行文本，确定哪一行最长。对于高度和宽度，也可以在显示的文档周围设置一个较小的页边距，使应用程序看起来更吸引人。

下面是计算文档大小的方法：

```
private void CalculateDocumentSize()
{
    if (!documentHasData)
    {
        documentSize = new Size(100, 200);
    }
    else
    {
        documentSize.Height = (int)(nLines*lineHeight) + 2*(int)margin;
        uint maxLineLength = 0;
        foreach (TextLineInformation nextWord in documentLines)
```



```

    {
        uint tempLineLength = nextWord.Width + 2*margin;
        if (tempLineLength > maxLineLength)
            maxLineLength = tempLineLength;
    }
    maxLineLength += 2*margin;
    documentSize.Width = (int)maxLineLength;
}
this.AutoScrollMinSize = documentSize;
}

```

这个方法首先检查是否有数据要显示。如果没有，就使用硬编码的文档大小，该大小足以显示很大的红色<Empty Document>警告。如果要正确显示数据，就应使用 `MeasureString()` 确定警告信息到底有多大。

计算出文档大小后，就设置 `Form.AutoScrollMinSize` 属性，告诉 `Form` 实例文档有多大。完成后，后台就会发生一些有趣的事。在设置这个属性的过程中，客户区域会标记为无效，引发 `Paint` 事件。改变文档的大小就意味着需要添加或修改滚动条，需要重新绘制整个客户区域。为什么说这很有趣？如果回过头来看看 `LoadFile()` 的代码，就会发现在该方法中调用 `Invalidate()` 是多余的。在设置文档大小时，肯定要使客户区域无效。在 `LoadFile()` 方法中显式调用 `Invalidate()`，说明了在一般情况下应如何完成任务。实际上在这个示例中，再次调用 `Invalidate()` 将重复请求 `Paint` 事件，这是不必要的。但是，这论证了前面论述的 `Invalidate()` 给 `Windows` 一个优化性能的机会。第二个 `Paint` 事件实际上并不会被引发：`Windows` 发现队列中已经有一个 `Paint` 事件了，就会比较请求的无效区域，看看是否需要合并它们。在本例中，两个 `Paint` 事件都指定了整个客户区域，所以不需要合并，`Windows` 会撤消第二个 `Paint` 请求。当然，这个过程会占用处理器一定的时间，但与某些绘图操作所占用的时间相比，它可以忽略不计。

### 33.15.3 OnPaint()

前面介绍了 `CapsEditor` 如何加载文件，下面看看如何绘图：

```

protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);
    Graphics dc = e.Graphics;
    int scrollPositionX = this.AutoScrollPosition.X;
    int scrollPositionY = this.AutoScrollPosition.Y;
    dc.TranslateTransform(scrollPositionX, scrollPositionY);

    if (!documentHasData)
    {
        dc.DrawString("<Empty document>", emptyDocumentFont,
            emptyDocumentBrush, new Point(20,20));
        base.OnPaint(e);
        return;
    }

    // work out which lines are in clipping rectangle
    int minLineInClipRegion =
        WorldYCoordinateToLineIndex(e.ClipRectangle.Top -
            scrollPositionY);
    if (minLineInClipRegion == -1)

```

```

        minLineInClipRegion = 0;
        int maxLineInClipRegion =
            WorldYCoordinateToLineIndex(e.ClipRectangle.Bottom -
                                         scrollPositionY);
        if (maxLineInClipRegion >= this.documentLines.Count ||
            maxLineInClipRegion == -1)
            maxLineInClipRegion = this.documentLines.Count-1;

        TextLineInformation nextLine;
        for (int i=minLineInClipRegion; i<=maxLineInClipRegion ; i++)
        {
            nextLine = (TextLineInformation)documentLines[i];
            dc.DrawString(nextLine.Text, mainFont, mainBrush,
                           this.LineIndexToWorldCoordinates(i));
        }
    }
}

```

这个 OnPaint()重载方法的核心是一个循环，迭代文档的每行文本，它调用 Graphics.DrawString()绘制每行文本。其余的代码主要是优化绘图操作——通常是确定哪些部分需要绘制，而不是告诉 graphics 实例绘制所有的内容。

首先检查一下文档中是否有数据。如果没有，就显示一个消息，调用基类的 OnPaint()方法并退出。如果有数据，就查看剪切的矩形，即调用另一个方法 WorldYCoordinateToLineIndex，后面再介绍这个方法，该方法带一个相对于文档顶部的 y 坐标，计算此时应显示文档中的哪些文本。

第一次调用方法 WorldYCoordinateToLineIndex()时，给它传送坐标值 e.ClipRectangle.Top - scrollPositionY，这是剪切区域的顶部，把它转换为世界坐标。如果返回值为-1，就假定需要从文档的开头开始(如果剪切区域位于顶部边距，就会出现这种情况)。

完成后，对剪切矩形的底部重复相同的过程，找出文档在剪切区域中的最后一行文本。第一行和最后一行的索引分别存储在 minLineInClipRegion 和 maxLineInClipRegion 中。这样就可以在这些值之间运行 for 循环，执行绘图操作了。在绘图循环中，需要通过 WorldYCoordinateToLineIndex()进行逆转换：给出一行文本的索引，要确定该行文本应绘制在什么位置。这个计算实际上是相当简单的，我们把它封装到另一个方法 LineIndexToWorldCoordinates()中，它返回该行文本左上角的坐标。返回的坐标是世界坐标，这很好，因为我们已在 Graphics 对象上调用了 TranslateTransform()，所以在要求显示文本时，需要给它传送世界坐标，而不是页面坐标。

#### 33.15.4 坐标转换

本节介绍在 CapsEditor 示例中编写的几个辅助方法，以帮助进行坐标转换。这些方法是上一节谈到的 WorldYCoordinateToLineIndex()和 LineIndexToWorldCoordinates()方法，还有几个其他的方法。

首先，LineIndexToWorldCoordinates()的参数是一个给定的行索引，它使用已知的页边距和行的高度，计算出该行左上角的世界坐标：

```

private Point LineIndexToWorldCoordinates(int index)
{

```



```

        Point TopLeftCorner = new Point(
            (int)margin, (int)(lineHeight*index + margin));
        return TopLeftCorner;
    }

```

我们还使用一个方法在 `OnPaint()` 中进行逆转换。`WorldYCoordinateToLineIndex()` 可计算出行索引，但它只考虑了一个垂直的世界坐标。这是因为它是相对于剪切区域的顶部和底部来计算行索引的。

```

private int WorldYCoordinateToLineIndex(int y)
{
    if (y < margin)
        return -1;
    return (int)((y-margin)/lineHeight);
}

```

还有 3 个方法，它们在处理程序例程中调用，响应用户的双击鼠标操作。首先，用一个方法计算出要在给定世界坐标处显示的文本行的索引。与 `WorldYCoordinateToLineIndex()` 不同，这个方法考虑了 `x` 和 `y` 坐标，如果该坐标上没有文本行，它就返回 -1：

```

private int WorldCoordinatesToLineIndex(Point position)
{
    if (!documentHasData)
        return -1;
    if (position.Y < margin || position.X < margin)
        return -1;
    int index = (int)(position.Y-margin)/(int)this.lineHeight;
    // check position isn't below document
    if (index >= documentLines.Count)
        return -1;
    // now check that horizontal position is within this line
    TextLineInformation theLine =
        (TextLineInformation)documentLines[index];
    if (position.X > margin + theLine.Width)
        return -1;

    // all is OK. We can return answer
    return index;
}

```

最后，需要转换行索引和页面坐标(而不是世界坐标)，下面的方法完成这个任务：

```

private Point LineIndexToPageCoordinates(int index)
{
    return LineIndexToWorldCoordinates(index) +
        new Size(AutoScrollPosition);
}

private int PageCoordinatesToLineIndex(Point position)
{
    return WorldCoordinatesToLineIndex(position - new
        Size(AutoScrollPosition));
}

```

注意在转换到页面坐标时，添加了 `AutoScrollPosition`，这是一个负值。

虽然这些方法看起来并不是很有趣，但它们说明了需要经常使用的一个技巧。在 GDI+ 中，

系统常常会给出一些坐标(例如用户单击鼠标的坐标),而我们需要确定在该坐标处要显示什么内容。或者相反——给出要显示的内容,确定它在什么地方显示。因此,如果编写一个 GDI+ 应用程序,就会发现编写完成坐标转换的方法是很有效的。

33.15.5 响应用户的输入

到目前为止,除了 CapsEditor 示例中的 File 菜单外,本章介绍的所有内容都是单向的:应用程序告诉用户一些信息,在屏幕上显示它们。当然,几乎所有的软件都是双向的:用户也可以与应用程序通信。下面就把这个功能添加到 CapsEditor 示例中。

让 GDI+应用程序响应用户输入,比编写代码在屏幕上绘图简单多了。实际上第 31 章已经介绍了如何处理用户输入。即重写 Form 类的方法,在相关的事件处理程序中调用。在引发 Paint 事件时,就是以这种方式调用 OnPaint()的。

当用户单击或移动鼠标时,可以重写的方法如表 33-5 所示。

表 33-5

方 法	何 时 调 用
OnClick(EventArgs e)	单击鼠标
OnDoubleClick(EventArgs e)	双击鼠标
OnMouseDown(MouseEventArgs e)	按下鼠标左键
OnMouseHover(MouseEventArgs e)	鼠标在移动后仍停留在某处
OnMouseMove(MouseEventArgs e)	移动鼠标
OnMouseUp (MouseEventArgs e)	释放鼠标左键

如果要检测用户什么时候键入文本,可以重写表 33-6 中的方法。

表 33-6

方 法	何 时 调 用
OnKeyDown(KeyEventArgs e)	按下一个键
OnKeyPress(KeyPressEventArgs e)	按下并释放一个键
OnKeyUp(KeyEventArgs e)	释放被按下的键

注意,其中的一些事件是重叠的。例如,如果用户按下鼠标按钮,就会引发 MouseDown 事件。如果按钮被立即释放,就会引发 MouseUp 事件和 Click 事件。另外,一些方法带有一个派生于 EventArgs 的参数,而不是 EventArgs 本身的实例。这些派生类的实例可以给出特定事件的更多信息。MouseEventArgs 有两个属性 X 和 Y,它们给出鼠标按下时的设备坐标信息。KeyEventArgs 和 KeyPressEventArgs 的属性则指定与事件相关的键。

这就是用户响应方法。用户应考虑自己要完成的工作的逻辑。唯一要注意的是,编写 GDI+ 应用程序可能要比 Windows.Forms 应用程序做更多的逻辑工作,这是因为在 Windows.Forms 应用程序中,一般响应的是比较高级的事件(例如,文本框的 TextChanged 事件)。GDI+则相反,其中

的事件都比较基本，用户单击鼠标，或按下键 H。应用程序的操作取决于一系列事件，而不是一个事件。例如，在 Word 中，为了选择一些文本，用户通常要单击鼠标左键，再移动鼠标，最后释放鼠标左键。应用程序接收到 `MouseDown` 事件，但仅有这个事件是不能完成什么任务的，只是记录下该鼠标单击时光标的位置。那么，当接收到 `MouseDown` 事件时，就可以检查刚才的记录，确定鼠标左键是否被按下，如果是，突出显示的文本就是用户选择的文本。当用户释放鼠标左键时，对应操作(在 `OnMouseUp()` 方法中)就需要检查一下是否有拖动操作，即鼠标按钮被按住并移动。只有这样，这个序列才算完成。

另外，因为某些事件有重叠，常常要选择让代码响应哪个事件。

一般规则是仔细考虑鼠标移动或单击和键盘事件的每个组合的逻辑，确保应用程序以直观的方式响应，让应用程序在各种情况下都按照期望的方式执行。我们的工作大多数是思考，而不是编码，但通过编码，可以更精细地完成工作，因为我们需要考虑用户输入的许多组合。例如，如果用户开始键入文本，而鼠标按钮处于按下状态，应用程序该如何响应？这听起来是不可能的，但迟早用户会尝试这么做的！

在 `CapsEditor` 示例中，为了使工作尽可能简单，并没有真正考虑任何组合的用户输入。只响应了用户的双击，此时把光标所在的那行文本变为大写字母。

这应是一个相当简单的任务，但有一个障碍，需要捕获 `DoubleClick` 事件。但上表说明，这个事件带一个 `EventArgs` 参数，不是 `MouseEventArgs` 参数，问题是如果要把一行文本正确标识为大写，我们需要知道用户双击时光标在什么地方，并需要一个 `MouseEventArgs` 参数。有两个解决办法，一个使用一个静态方法，由 `Form1` 对象 `Control.MousePosition` 执行，确定光标的位置：

```
protected override void OnDoubleClick(EventArgs e)
{
    Point MouseLocation = Control.MousePosition;
    // handle double click }
```

在大多数情况下，这个方法可以正常工作。但如果应用程序(甚至是其他一些有较高优先级的应用程序)在用户双击时进行某些计算密集型的工作，该方法就会有问题。此时要在等待大约半秒钟后，才调用 `OnDoubleClick()` 事件处理程序。我们不期望有这样的延迟，因为这会让用户感到很厌烦，但有时因为应用程序不能控制的原因(例如计算机较慢)，会偶尔发生这种情况。半秒的时间足够光标在屏幕上移动到其他位置了——此时会对完全错误的位置执行 `Control.MousePosition`！

比较好的方法是依赖于鼠标事件之间的重叠。双击鼠标的第一部分涉及到按下鼠标左键。这表示如果调用了 `OnDoubleClick()`，我们就知道 `OnMouseDown()` 刚被调用，此时鼠标的位置不变。可以使用 `OnMouseDown()` 重写方法记录光标的位置，为 `OnDoubleClick()` 作准备。这就是在 `CapsEditor` 中采用的方法：

```
protected override void OnMouseDown(MouseEventArgs e)
{
    base.OnMouseDown(e);
    this.mouseDoubleClickPosition = new Point(e.X, e.Y);
}
```

下面看看 `OnDoubleClick()` 重写方法，该方法完成了许多工作：

```
protected override void OnDoubleClick(EventArgs e)
{
    int i = PageCoordinatesToLineIndex(this.mouseDoubleClickPosition);
```

```

        if (i >= 0)
        {
            TextLineInformation lineToBeChanged =
                (TextLineInformation)documentLines[i];
            lineToBeChanged.Text = lineToBeChanged.Text.ToUpper();
            Graphics dc = this.CreateGraphics();
            uint newWidth = (uint)dc.MeasureString(lineToBeChanged.Text,
                mainFont).Width;
            if (newWidth > lineToBeChanged.Width)
                lineToBeChanged.Width = newWidth;
            if (newWidth+2*margin > this.documentSize.Width)
            {
                this.documentSize.Width = (int)newWidth;
                this.AutoScrollMinSize = this.documentSize;
            }
            Rectangle changedRectangle = new Rectangle(
                LineIndexToPageCoordinates(i),
                new Size((int)newWidth,
                    (int)this.lineHeight));
            this.Invalidate(changedRectangle);
        }
        base.OnDoubleClick(e);
    }
}

```

首先调用 `PageCoordinatesToLineIndex()` 计算在用户双击时光标在哪行文本上。如果它返回 -1, 则光标没有在任何文本行上, 所以什么也不做。当然, 调用 `OnDoubleClick()` 的基类版本, 可以让 Windows 执行一些默认操作。

假定已经标识出了一行文本, 就可以使用 `string.ToUpper()` 方法把它转换为大写。这是很容易的。比较难的部分是确定要在什么地方重新显示什么文本。幸运的是, 因为这个示例非常简单, 没有太多的组合。可以假定一个开头, 把文本转换为大写通常要么让这行文本的宽度保持不变, 要么增大其宽度。大写字母比小写字母大, 所以, 宽度是不会减小的。因为本例没有换行功能, 所以文本行不会移动到下一行上, 使其他文本向下移动。把文本行转换为大写的操作不会改变已显示的其他文本行的位置, 这是一个非常大的简化!

接着代码使用 `Graphics.MeasureString()` 计算出文本的新宽度。这有两种可能性:

- 新宽度会使该文本更长, 增大了整个文档的宽度。如果是这样, 就需要把 `AutoScrollMinSize` 设置为新值, 以便正确放置滚动条。
- 文档的大小没有变化。

在这两种情况下, 都需要调用 `Invalidate()` 重新绘制屏幕。只有一行文本改变了, 所以不希望重新绘制整个文档。而需要计算出仅包含被修改的文本行的矩形边界, 并把这个矩形传送给 `Invalidate()`, 确保只重新绘制该行文本。这就是以前代码的作用。`Invalidate()` 会在鼠标事件处理程序返回时调用 `OnPaint()`。在本章前面, 曾提到在 `OnPaint()` 中设置断点的困难, 如果运行示例, 在 `OnPaint()` 中设置断点, 捕获绘图操作, 就会发现传送给 `OnPaint()` 的 `PaintEventArgs` 参数包含一个与指定矩形匹配的剪切区域。因为重载了 `OnPaint()` 方法, 考虑了剪切区域, 所以只重新绘制要求的文本行。

## 33.16 打印

前面主要介绍的是如何在屏幕上绘图。但有时还需要应用程序生成数据的硬拷贝。这就是



本节的主题。我们将扩展 CapsEditor 示例，使之能进行打印预览，并打印出正在编辑的文档。

遗憾的是，本书没有详细讲述这个过程，所以这里执行的打印功能是非常简单的。通常，如果要想让应用程序打印数据，要在主 File 菜单中添加 3 个菜单项：

- Page Setup: 允许用户选择各种选项，例如要打印哪个页面，要使用什么打印机等。
- Print Preview 打开一个新窗体，显示打印机拷贝的模仿品。
- Print: 实际打印文档。

在本例中，为了使例子简单一些，不执行 Page Setup 菜单项。打印也只使用默认的设置。但要注意，如果要执行 Page Setup，Microsoft 已经编写了一个页面设置对话框类，以供使用。这个类是 System.Windows.Forms.PrintDialog。一般应编写一个事件处理程序，显示这个窗体，保存用户选择的设置。

在许多情况下，打印与在屏幕上显示是一样的：提供一个设备环境(Graphics 实例)，对该实例调用所有常见的显示命令。Microsoft 编写了许多类以帮助完成这个工作，我们需要的两个主要的类是 System.Drawing.Printing.PrintDocument 和 System.Drawing.Printing.PrintPreviewDialog。这两个类可以确保传送给设备环境的绘图命令能得到正确的处理，我们不必担心什么内容应打印到何处的问题。

打印/打印预览和显示在屏幕上有一些重要的区别。打印机不能滚动，但它们使用打印纸。所以需要确保找到一种合适的方式给文档分页，按要求绘制每一页。其他工作还有计算文档有多少内容可以放在一个页面上，因此计算出需要多少页，文档的每个部分应写到哪个页面上。

尽管上面描述得相当复杂，但打印过程是相当简单的。从编程的角度来看，需要采取的步骤大致如下：

- 打印：实例化一个 PrintDocument 对象，调用其 Print() 方法。这个方法会在内部引发事件 PrintPage，发出打印第一个页面的信号。PrintPage 带一个参数 PrintPageEventArgs，它提供了页面大小和设置的信息，以及用于绘图命令的 Graphics 对象。因此应为这个事件编写一个事件处理程序，并执行该处理程序，以打印页面。这个事件处理程序还应把 PrintPageEventArgs 的布尔属性 HasMorePages 设置为 true 或 false，表示是否还有要打印的页面。PrintDocument.Print() 方法将重复引发 PrintPage 事件，直到 HasMorePages 设置为 false 为止。
- 打印预览：此时，实例化 PrintDocument 对象和 PrintPreviewDialog 对象。使用属性 PrintPreviewDialog.Document 把 PrintDocument 关联到 PrintPreviewDialog 上，然后调用对话框的 ShowDialog() 方法。这个方法会模式化地显示对话框，使之呈现为 Windows 标准打印预览窗体，显示文档的页面。在内部，则重复引发 PrintPage 事件，多次显示页面，直到 HasMorePages 属性为 false 为止。不需为此编写一个事件处理程序；而可以使用打印每个页面时使用的同一个事件处理程序，因为绘图代码在这两种情况下应是一样的(毕竟，打印预览的内容应与打印出来的版本看起来完全相同！)。

## 实现 Print 和 Print Preview

前面讨论了要执行的一般步骤，下面看看如何在代码中完成这些步骤。可以从 [www.wrox.com](http://www.wrox.com) 上下载 PrintingCapsEdit 项目，它包含 CapsEditor 项目，但进行了下述修改。

首先使用 Visual Studio 2008 设计视图在 File 菜单中添加两个新菜单项：Print 和 Print



Preview。再使用属性窗口把这两个项命名为 `menuFilePrint` 和 `menuFilePrintPreview`，把它们设置为应用程序启动时是禁用的(只有打开了文档，才能进行打印!)。在主窗体的 `LoadFile()` 方法中添加如下代码，激活这两个菜单项，`LoadFile()` 方法负责把文件加载到 `CapsEditor` 应用程序中：

```
private void LoadFile(string FileName)
{
    StreamReader sr = new StreamReader(FileName);
    string nextLine;
    documentLines.Clear();
    nLines = 0;
    TextLineInformation nextLineInfo;
    while ( (nextLine = sr.ReadLine()) != null)
    {
        nextLineInfo = new TextLineInformation();
        nextLineInfo.Text = nextLine;
        documentLines.Add(nextLineInfo);
        ++nLines;
    }
    sr.Close();
    if (nLines > 0)
    {
        documentHasData = true;
        menuFilePrint.Enabled = true;
        menuFilePrintPreview.Enabled = true;
    }
    else
    {
        documentHasData = false;
        menuFilePrint.Enabled = false;
        menuFilePrintPreview.Enabled = false;
    }

    CalculateLineWidths();
    CalculateDocumentSize();

    Text = standardTitle + " - " + FileName;
    Invalidate();
}
```

上面突出显示的代码是添加到这个方法中的新代码。接着给 `Form1` 类添加一个成员字段：

```
public class Form1 : System.Windows.Forms.Form
{
    private int pagesPrinted = 0;
```

这个字段用于指定目前正在打印的页面。使它成为一个成员字段，因为需要在 `PrintPage` 事件处理程序的调用之间记忆这个信息。

接着是用户选择 `Print` 或 `Print Preview` 菜单项时执行的事件处理程序：

```
private void menuFilePrintPreview_Click(object sender, System.EventArgs e)
{
    this.pagesPrinted = 0;
    PrintPreviewDialog ppd = new PrintPreviewDialog();
    PrintDocument pd = new PrintDocument();
    pd.PrintPage += this.pd_PrintPage;
    ppd.Document = pd;
    ppd.ShowDialog();
}
```

```
private void menuFilePrint_Click(object sender, System.EventArgs e)
{
    this.pagesPrinted = 0;
    PrintDocument pd = new PrintDocument();
    pd.PrintPage += new PrintPageEventHandler
        (this.pd_PrintPage);
    pd.Print();
}
```

前面解释了涉及打印的主要过程，可以看出，这些事件处理程序仅执行了这个过程。在打印和打印预览中，都实例化了一个 `PrintDocument` 对象，并把一个事件处理程序关联到该对象的 `PrintPage` 事件上。对于打印，应调用 `PrintDocument.Print()`，而对于打印预览，则把 `PrintDocument` 对象关联到 `PrintPreviewDialog` 上，并调用打印预览对话框对象的 `ShowDialog()` 方法。主要工作将在 `PrintPage` 事件的处理程序中完成。下面是该处理程序的代码：

```
private void pd_PrintPage(object sender, PrintPageEventArgs e)
{
    float yPos = 0;
    float leftMargin = e.MarginBounds.Left;
    float topMargin = e.MarginBounds.Top;
    string line = null;

    // Calculate the number of lines per page.
    int linesPerPage = (int)(e.MarginBounds.Height /
        mainFont.GetHeight(e.Graphics));
    int lineNo = this.pagesPrinted * linesPerPage;

    // Print each line of the file.
    int count = 0;
    while(count < linesPerPage && lineNo < this.nLines)
    {
        line = ((TextLineInformation)this.documentLines[lineNo]).Text;
        yPos = topMargin + (count * mainFont.GetHeight(e.Graphics));
        e.Graphics.DrawString(line, mainFont, Brushes.Blue,
            leftMargin, yPos, new StringFormat());
        lineNo++;
        count++;
    }

    // If more lines exist, print another page.
    if(this.nLines > lineNo)
        e.HasMorePages = true;
    else
        e.HasMorePages = false;
    pagesPrinted++;
}
```

在声明了两个局部变量后，应先计算出一个页面上可以显示多少行文本——即页面的高度除以一行文本的高度，并取整。页面的高度可以从 `PrintPageEventArgs.MarginBounds` 属性获得，这个属性是一个 `RectangleF` 结构，该结构已初始化为页面的边界。一行文本的高度可以从 `Form1.mainFont` 字段中获得，该字段是显示文本所使用的字体。这里必须使用相同的字体进行打印。注意，对于 `PrintingCapsEditor` 示例，每页上的行数总是相同的，所以可以在第一次计算出该行数后，把它高速缓存起来。但是，该计算并不困难，在比较复杂的应用程序中，这个值可能会有变化，所以每次打印页面时重新计算它也是可以的。

我们还初始化了一个变量 `lineNo`，这是一个文档中文本行的基于 0 的索引，索引为 0 的文本

行是文档中的第一行，这个信息是非常重要的，因为在原则上，可以调用 `pd_PrintPage()` 方法打印任何页面，而不仅仅是打印第一页。`lineNo` 的值是每个页面中的行数和已打印的页面数的乘积。

接着运行一个循环，打印每一行文本。当打印完文档中的所有文本行，或者打印完本页面上的所有文本行时，这个循环就终止(只要满足这两个条件之一即可)。最后，检查是否还有要打印的文档，并据此设置 `PrintPageEventArgs` 的 `HasMorePages` 属性，递增 `pagesPrinted` 字段，这样下一次调用 `PrintPage` 事件处理程序时，就会打印出正确的页面了。

这个事件处理程序要注意的一点是不必担心绘图命令的发送目的地。我们使用所提供的 `Graphics` 对象和 `PrintPageEventArgs`。Microsoft 编写的 `PrintDocument` 类将在内部确保，如果进行打印，`Graphics` 对象就与打印机关联起来，如果进行打印预览，`Graphics` 对象就与屏幕上的打印预览窗体关联起来。

最后，需要确保为类型定义导入 `System.Drawing.Printing` 命名空间：

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Drawing.Printing;
using System.Text;
using System.Windows.Forms;
using System.IO;
```

剩下的就是编译项目，并检查代码的工作情况。如果运行 `CapsEdit`，加载一个文本文档(与以前一样，为该项目选择 C# 源文件)，并选择 `Print Preview`，就可以显示出该文档的打印版本，如图 33-19 所示。

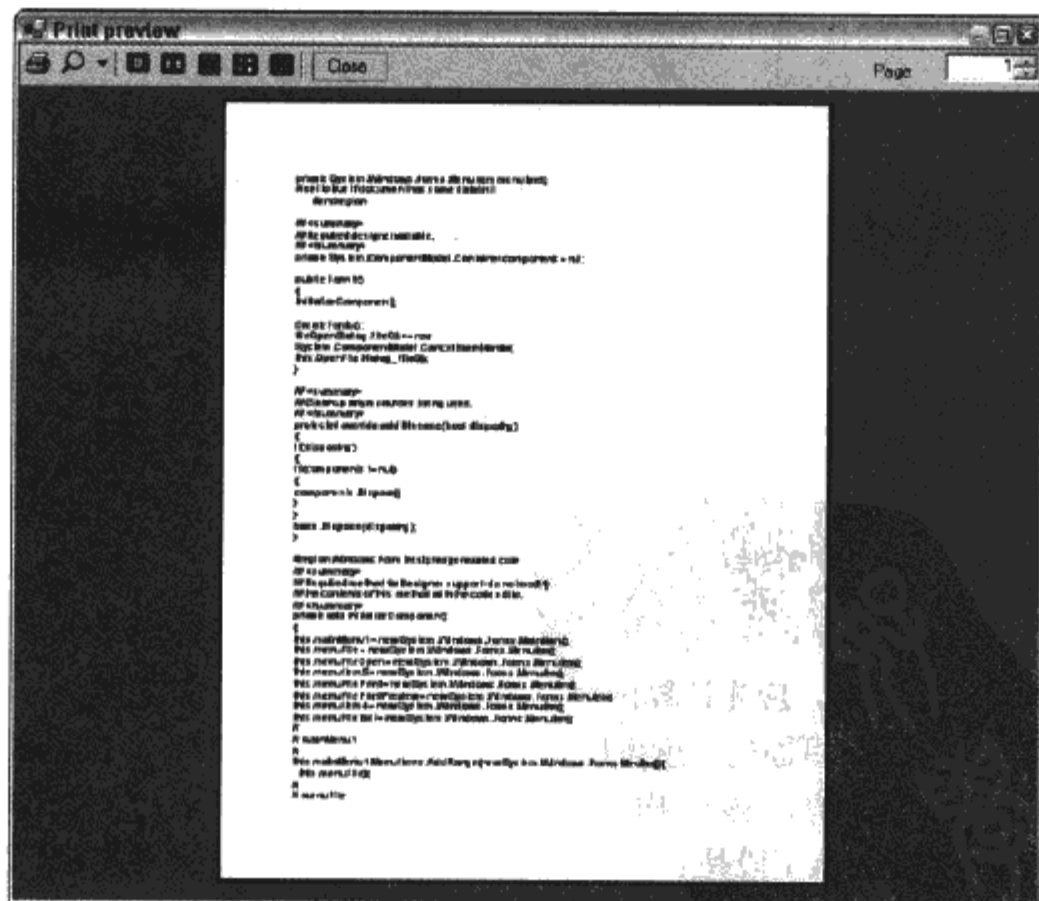


图 33-19

在图 33-19 上，滚动到文档的第 5 页上，把预览设置为显示正常的尺寸。`PrintPreviewDialog` 提供了许多功能，这可以从窗体顶部的工具栏中看出来。可用的选项包括打印文档、缩小和放

大文档、一次显示 2、3、4 或 6 页。这些选项的功能都是很完整的，不需要我们做任何工作。例如，如果把缩放改为自动，单击显示 4 页的选项(从右数的第 3 个工具栏)，就会得到如图 33-20 所示的结果。

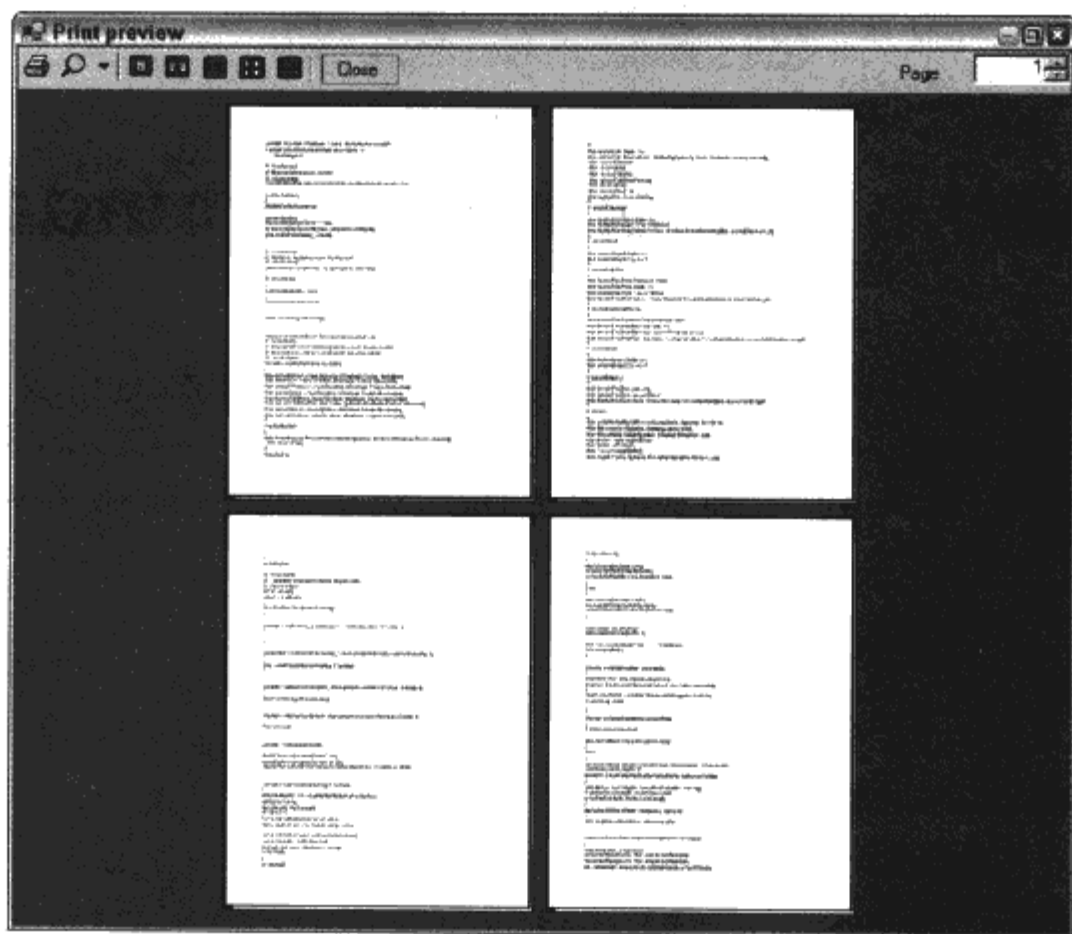


图 33-20

### 33.17 小结

本章介绍了如何在显示设备上绘图操作，其中绘图操作是通过代码实现的，而不是由一些预定义的控件或对话框来实现的，这就是 GDI+ 的实质。GDI+ 是一个功能强大的工具，有许多 .NET 基类都可用于在设备上绘图。绘图过程实际上是非常简单的，在大多数情况下，只使用几个 C# 语句，就可以绘制文本和专业化的图形或图像。但是，管理绘图操作——后台的工作涉及到计算出要绘制的内容、确定在什么地方绘制，以及在任何给定的情况下，哪些内容需要重新绘制，哪些内容不需要重新绘制。这些工作都非常复杂，需要经过仔细的算法设计。因此，很好地理解 GDI+ 的工作方式，以及 Windows 采取什么操作来完成工作也是非常重要的。特别是，由于 Windows 的体系结构，在绘图过程中，应使窗口的区域失效，并依赖 Windows 通过引发 Paint 事件来响应。

本章并没有介绍绘图操作所涉及到的所有 .NET 类，但如果您理解了绘图操作的规则，就可以通过查看 SDK 文档说明中这些类的方法列表，实例化它们的实例，看看它们能做什么，以掌握它们。最后，绘图与其他编程一样，需要逻辑、仔细的思考和清晰的算法。应用它们，就能够编写出不依赖于标准控件的专业化用户界面。许多应用程序完全依赖于控件来设计其用户界面。这是很有效的，但这种应用程序看起来非常类似。通过添加一些 GDI+ 代码完成定制的绘图操作，可以使软件比较独特，比较新颖——这绝不仅仅有助于软件的销售！

下一章介绍最新的胖客户显示技术 WPF。

# 第34章

## Windows Presentation Foundation

Windows Presentation Foundation(WPF)是.NET Framework 3.0 中的三个主要扩展之一。WPF 是为智能客户应用程序创建 UI 的一个新库。Windows 窗体控件基于 Windows 内置控件，利用了基于屏幕像素的 Windows 句柄。而 WPF 基于 DirectX。应用程序不再使用 Windows 句柄，更便于重新设置 UI 的大小，并内置了音频和视频的支持。

本章的主要论题如下：

- WPF 概述
- 用作基本绘图元素的图形
- 利用转换功能实现缩放、旋转和倾斜
- 填充元素的不同笔刷
- WPF 控件及其特性
- 如何用 WPF 面板定义布局
- WPF 事件处理机制
- 样式、模板和资源

### 34.1 概述

WPF 的一个主要特性是设计人员和开发人员的工作很容易分开。设计人员的工作成果可以直接供开发人员使用。为此，必须理解 XAML。本章的第一个主题是概述 WPF，理解 XAML 的规则，讨论设计人员和开发人员如何合作。WPF 由几个包含了上千个类的程序集组成。因此用户可以在这些类中浏览，查找需要的类，大致了解 WPF 中的类层次结构和命名空间。

#### 34.1.1 XAML

XML for Applications Markup Language(XAML)是一种 XML 语法，用于定义用户界面的层次结构。在下面的代码行中，声明了一个内容为 Click Me!、名为 button1 的按钮。<Button>元素指定使用 Button 类：

```
<Button Name="button1">Click Me!</Button>
```

**提示：**

XAML 元素总是有一个 .NET 类。在特性和子元素中，可以设置属性的值，定义事件的处理程序方法。



为了测试简单的 XAML 代码，可以启动实用工具 XAMLPad.exe（参见图 34-1），在编辑字段中输入 XAML 代码。在 XAMLPad 中，可以在已准备好的<Page>和<Grid>元素中编写<Button>元素。利用 XAMLPad 可以立即查看 XAML 结果。

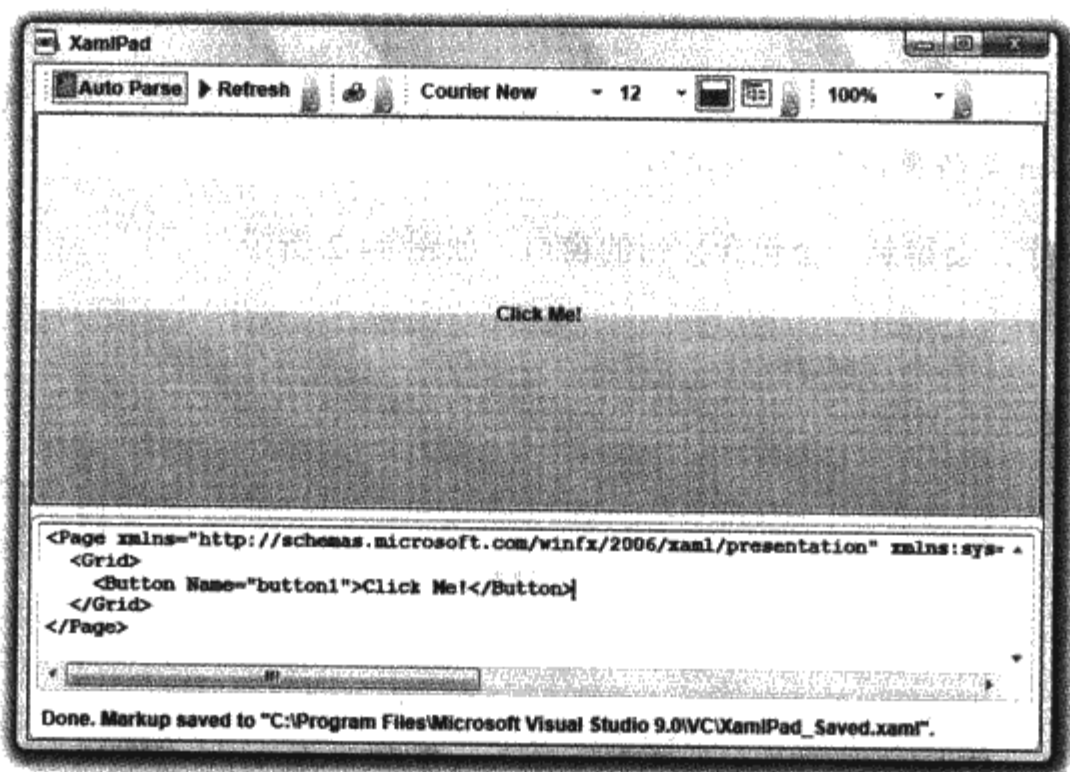


图 34-1

XAML 代码可以由 WPF 运行库解释，也可以编译为 BAML（Binary Applications Markup Language），在默认情况下，这是由 Visual Studio WPF 项目完成的。BAML 添加为可执行文件的一个资源。

除了编写 XAML 之外，也可以用 C# 代码创建按钮。我们可以创建一个正常的 C# 控制台应用程序，添加对程序集 WindowsBase、PresentationCore 和 PresentationFramework 的引用，再编写下面的代码。在 Main() 方法中，从 System.Windows 命名空间中创建一个 Window 对象，设置 Title 属性。接着从 System.Windows.Controls 命名空间中创建一个 Button 对象，设置 Content 属性，将窗口的 Content 属性设置为该按钮。Application 类的 Run() 方法负责处理 Windows 消息。

```
using System;
using System.Windows;
using System.Windows.Controls;

namespace Wrox.ProCSharp.WPF
{
    class Program
    {
        [STAThread]
        static void Main()
        {
            Window mainWindow = new Window();
            mainWindow.Title = "WPF Application";
            Button button1 = new Button();
            button1.Content = "Click Me!";
            mainWindow.Content = button1;
            button1.Click +=
                (sender, e) => MessageBox.Show("Button clicked");
        }
    }
}
```

```
Application app = new Application();  
app.Run(mainWindow);  
}  
}
```

提示:

Application 类也可以用 XAML 定义。在 Visual Studio WPF 项目中, 打开 App.xaml 文件, 它包含 Application 类的属性和 StartupUri。

运行应用程序, 会得到一个包含按钮的窗口, 如图 34-2 所示。

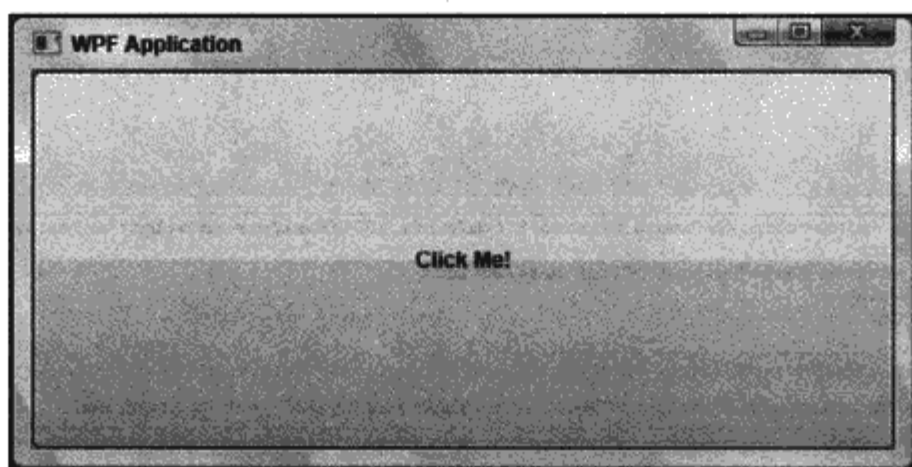


图 34-2

可以看出, WPF 的编程非常类似于 Window 窗体的编程, 其区别是按钮有 Content 属性, 而不是 Text 属性。但是, 与通过代码创建 UI 窗体相比, XAML 有一些非常好的优点。利用 XAML, 设计人员和开发人员可以更好地合作。设计人员可以用 XAML 代码设计一个漂亮的 UI, 开发人员在使用 C# 在后台代码中添加功能。使用 XAML 更便于将 UI 与功能分开。

使用后台代码和 XAML, 可以在 C# 代码中直接与用 XAML 定义的元素交互操作。只需为该元素定义名称, 将该名称用作变量, 来修改属性, 调用方法。

按钮有一个 Content 属性, 而不是 Text 属性, 因为按钮可以显示任意信息。可以给按钮添加文本、图形、列表框、视频——等任何元素。

### 1. 将属性用作特性

在使用 XAML 之前, 需要了解 XAML 语法的重要特性。使用 XML 特性可以指定类的属性。下面的例子说明了如何设置 Button 类的 Content 和 Background 属性。

```
<Button Content="Click Me!" Background="LightGreen" />
```

### 2. 将属性用作元素

除了使用 XML 特性之外, 属性也可以指定为子元素。指定 Button 元素的子元素, 就可以直接设置 Content 的值。对于 Button 的其他属性, 子元素名用外部元素的名称定义, 之后是属性名:

```
<Button>  
  <Button.Background>
```

```

    LightGreen
</Button.Background>
Click Me!
</Button>

```

在上面的例子中，不一定要使用子元素；使用 XML 特性，也可以得到相同的结果。但是，如果特性值比字符串还复杂，就不能再使用特性了。例如，背景不仅可以设置为一种简单的颜色，还可以设置为笔刷，如设置为线性渐变笔刷：

```

<Button>
  <Button.Background>
    <LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
      <GradientStop Color="Yellow" Offset="0.0" />
      <GradientStop Color="Orange" Offset="0.25" />
      <GradientStop Color="Red" Offset="0.75" />
      <GradientStop Color="Violet" Offset="1.0" />
    </LinearGradientBrush>
  </Button.Background>
  Click Me!
</Button>

```

### 3. 依赖属性

在用 WPF 编程时，常常会遇到“依赖属性”这个术语。WPF 元素是带有方法、属性和事件的类。WPF 元素的几乎每个属性都是依赖属性，这是什么意思？依赖属性可以依赖其他输入，例如主题和用户喜好。依赖属性与数据绑定、动画、资源和样式一起使用。

从编程的角度来看，要读写依赖属性，可以调用强类型化的属性，也可以给方法传送依赖属性对象。

只有派生自 `DependencyObject` 基类的类才能包含依赖属性。下面的类 `MyDependencyObject` 定义了依赖属性 `SomeState`。`SomeStateProperty` 是 `DependencyProperty` 类型的一个静态字段，它支持依赖属性。依赖属性使用 `Register()` 方法通过 WPF 依赖属性系统来注册。`Register()` 方法可以获取依赖属性的名称、依赖属性的类型和拥有者的类型。使用 `DependencyObject` 基类的 `SetValue()` 方法，可以设置依赖属性的值。使用 `GetValue()` 方法可以获取其值。依赖属性常常还有强类型化的访问权限。除了使用 `DependencyObject` 基类的方法之外，类 `MyDependencyObject` 还包含属性 `SomeState`，它从 `set` 和 `get` 存取器的实现代码中调用基类的方法。不应在 `SetValue()` 和 `GetValue()` 方法的执行代码中执行其他操作，因为不可能调用这些属性存取器。

```

public class MyDependencyObject : DependencyObject
{
    public static readonly DependencyProperty SomeStateProperty =
        DependencyProperty.Register("SomeState", typeof(String),
            typeof(MyDependencyObject));

    public string SomeState
    {
        get { return (string)this.GetValue(SomeStateProperty); }
        set { this.SetValue(SomeStateProperty, value); }
    }
}

```

**提示:**

在 WPF 中, 类 `DependencyObject` 位于层次结构的最高层。每个 WPF 元素都派生自这个基类。

**4. 附带属性**

WPF 元素也可以从父元素中获得特性。例如, 如果 `Button` 元素位于 `Canvas` 元素中, 按钮的 `Top` 和 `Left` 属性把父元素的名称作为前缀。这种属性称为附带属性:

```
<Canvas>
  <Button Canvas.Top="30" Canvas.Left="40">
    Click Me!
  </Button>
</Canvas>
```

在后台代码中编写相同的功能有点不同, 因为 `Button` 类没有 `Canvas.Top` 和 `Canvas.Left` 属性, 但它包含在 `Canvas` 类中。

设置所有类都有的附带属性有一个命名模式。支持附带属性的类有静态方法 `Set<Property>` 和 `Get<Property>`, 其中第一个参数是应用属性值的对象。`Canvas` 类定义了静态方法 `SetLeft()` 和 `SetTop()`, 它们会获得与前面 XAML 代码相同的结果:

```
[STAThread]
static void Main()
{
    Window mainWindow = new Window();
    Canvas canvas = new Canvas();
    mainWindow.Content = canvas;

    Button button1 = new Button();
    canvas.Children.Add(button1);

    button1.Content = "Click Me!";
    Canvas.SetLeft(button1, 40);
    Canvas.SetTop(button1, 30);

    Application app = new Application();
    app.Run(mainWindow);
}
```

**提示:**

附带属性可以实现为依赖对象。方法 `DependencyProperty.RegisterAttached()` 会注册附带属性。

**5. 标记扩展**

在为元素设置值时, 可以直接设置值, 但有时标记扩展非常有帮助。标记扩展包含花括号, 其后是定义了标记扩展类型的字符串标志。下面是一个 `StaticResource` 标记扩展:

```
<Button Name="button1" Style="{StaticResource key}" Content="Click Me" />
```

除了使用标记扩展之外, 还可以使用子元素编写相同功能的代码:

```
<Button Name="button1">
  <Button.Style>
    <StaticResource ResourceKey="key" />
  </Button.Style>
</Button>
```



```

</Button.Style>
Click Me!
</Button>

```

标记扩展主要用于访问资源和数据绑定，本章后面将讨论这两个主题。

### 34.1.2 设计人员和开发人员的合作

开发人员常常不仅要实现 Windows 应用程序，还负责应用程序的设计。尤其是应用程序仅用于内部使用，就更是如此。如果雇用懂得 UI 设计技巧的人来设计 UI，开发人员通常会得到设计人员的一个 JPG 文件，该文件描述了 UI 的外观。接着开发人员就要试图实现设计人员的规划。设计人员的一个简单修改，如给列表框和按钮指定不同的外观，就需要开发人员投入大量的精力设计定制的控件。因此，开发人员设计的 UI 与最初的设计大相径庭。

WPF 改变了这种情况。设计人员和开发人员可以使用相同的 XAML 代码。设计人员可以使用 Expression Blend 工具，开发人员则使用 Visual Studio 2008，但他们可以用同一个项目文件工作。在这个合作过程中，设计人员使用与 Visual Studio 中相同的项目文件在 Expression Blend 中启动一个项目，接着，开发人员编写后台代码，同时设计人员改进 UI。在开发人员改进功能的同时，设计人员还可以利用开发人员提供的功能，添加新的 UI 特性。

当然，还可以用 Visual Studio 启动应用程序，以后再用 Expression Blend 改进 UI。只是要小心，不要像处理 Windows 窗体那样设计 UI，因为这没有充分利用 WPF。

图 34-3 显示了用 WPF 创建的 Expression Blend。

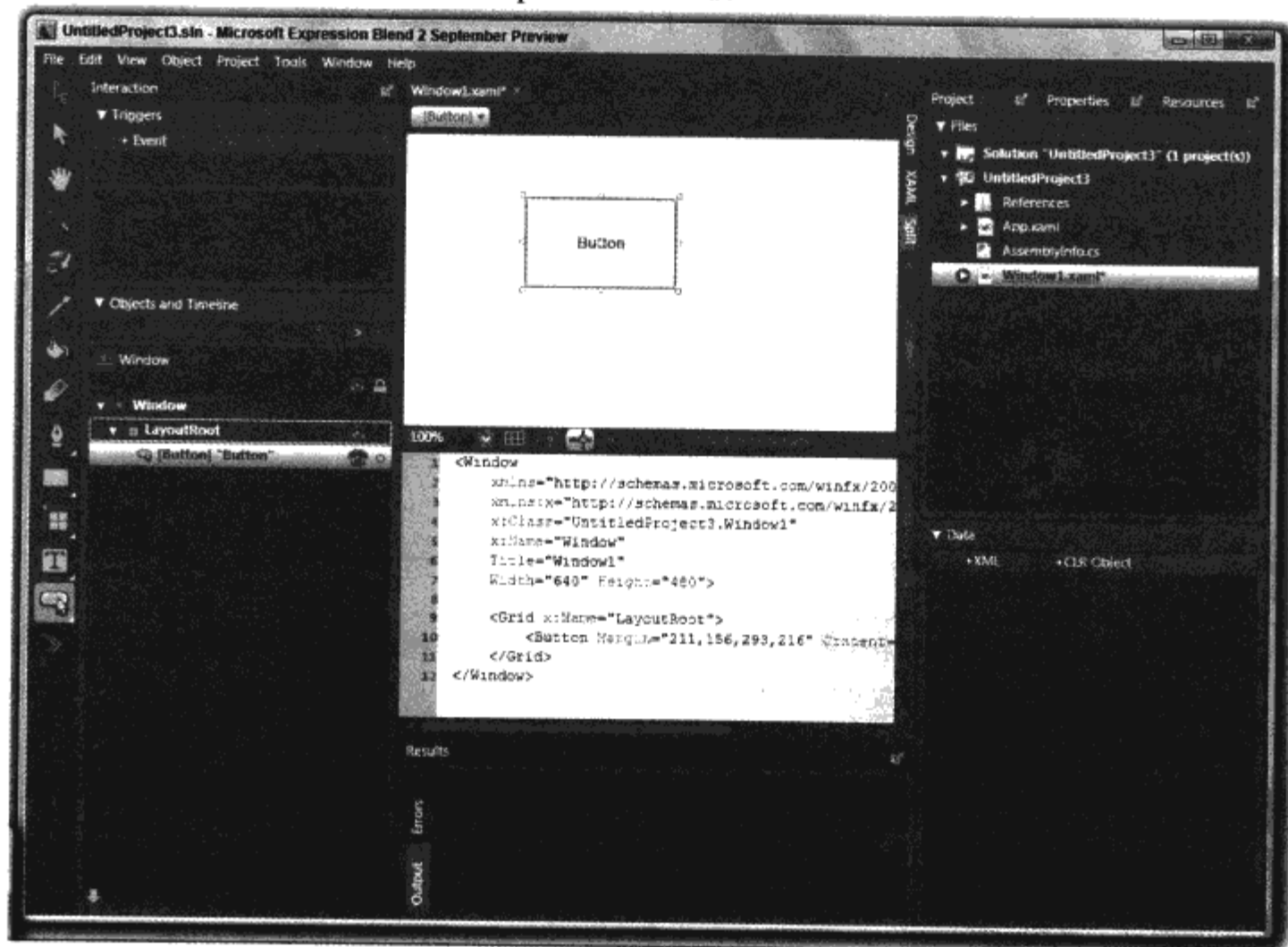


图 34-3



提示:

比较 Expression Blend 和 Visual Studio 扩展, Expression Blend 的优秀特性有定义样式、创建动画、使用图形等。为了合作, Expression Blend 可以使用开发人员编写的后台编码类,设计人员可以在 WPF 元素中指定与 .NET 类的数据绑定。设计人员还可以在 Expression Blend 中启动完整的应用程序,来测试它。因为 Expression Blend 使用与 Visual Studio 相同的 MS-Build 文件,所以能编译后台编码的 C# 代码,运行应用程序。

34.1.3 类层次结构

WPF 包含上千个类,有很深的层次结构。为了帮助理解类之间的关系,图 34-4 在一个类图中列出了一些 WPF 类。表 34-1 描述了一些类及其功能。

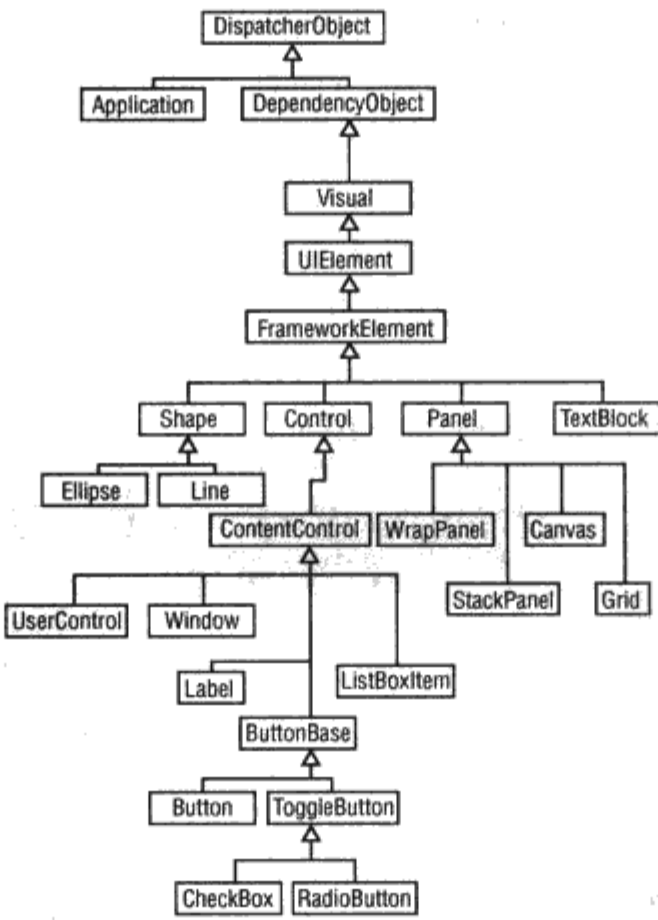


图 34-4

表 34-1

类	说 明
DispatcherObject	DispatcherObject 是一个抽象基类,用于绑定到一个线程上的类。与 Windows 窗体类似,WPF 也要求仅从创建线程中调用方法和属性。派生自 DispatcherObject 的类有一个关联的 Dispatcher 对象,它可以用于切换线程
Application	在 WPF 应用程序中,会创建 Application 类的一个实例。这个类实现了 Singleton 模式,用于访问应用程序的窗口、资源和属性
DependencyObject	DependencyObject 是所有支持依赖属性的类的基类。依赖属性如前所述
Visual	所有可见元素的基类是 Visual。这个类包含点击测试和转换等特性

(续表)

类	说 明
UIElement	所有需要基本显示功能的 WPF 元素的抽象基类是 UIElement。这个类提供了鼠标移动、拖放、按键的通道和起泡事件；提供了可以由派生类重写的虚显示方法；以及布局方法。WPF 不再使用 Window 句柄，这个类就可以用作 Window 句柄
FrameworkElement	FrameworkElement 派生自基类 UIElement，实现了由基类定义的方法的默认代码
Shape	Shape 是所有图形元素的基类，例如 Line、Ellipse、Polygon、Rectangle
Control	Control 派生自 FrameworkElement，是所有用户交互元素的基类
Panel	Panel 派生自 FrameworkElement，是所有面板的抽象基类，这个类的 Children 属性用于面板中的所有 UI 元素，定义了安排子控件的方法。派生自 Panel 的类为子控件的布置方式定义了不同的类，例如 WrapPanel、StackPanel、Canvas、Grid
ContentControl	ContentControl 是所有有单个内容的控件的基类，如 Label、Button。内容控件的默认样式是受限制的，但可以使用模板改变其外观

可以看出，WPF 类有非常深的层次结构。本章和下一章将介绍提供核心功能的类，但不可能用两章的篇幅涵盖 WPF 的所有特性。

31.1.4 命名空间

Windows 窗体类和 WPF 类很容易混淆。Windows 窗体类位于 System.Windows.Forms 命名空间，而 WPF 类位于 System.Windows 命名空间及其子命名空间中，但不位于 System.Windows.Forms 命名空间。Windows 窗体的 Button 类的全称是 System.Windows.Forms.Button，而用于 WPF 的 Button 类的全称是 System.Windows.Controls.Button。Windows 窗体参见第 31 和 32 章。

WPF 的命名空间及其功能如表 34-2 所述。

表 34-2

命 名 空 间	说 明
System.Windows	这是 WPF 的核心命名空间，其中包含 WPF 的核心类，如 Application 类、用于依赖对象的类、DependencyObject 和 DependencyProperty，所有 WPF 元素的基类 FrameworkElement
System.Windows.Annotations	这个命名空间中的类用于用户在应用程序数据上创建的标识和记录，它们与文档分开存储。命名空间 System.Windows.Annotations.Storage 包含了存储标识的类
System.Windows.Automation	System.Windows.Automation 命名空间用于自动完成 WPF 应用程序。它有几个子命名空间。System.Windows.Automation.Peers 命名空间包含用于自动化的 WPF 元素，如 ButtonAutomationPeer 和 CheckBoxAutomationPeer。 如果创建定制的自动化提供程序，就需要 System.Windows.Automation.Provider 命名空间

(续表)

命 名 空 间	说 明
System.Windows.Controls	这个命名空间包含了所有 WPF 控件, 如 Button、Border、Canvas、ComboBox、Expander、Slider、ToolTip、TreeView 等。在命名空间 System.Windows.Controls.Primitives 中, 包含了在复杂控件中使用的类, 如 Popup、ScrollBar、StatusBar、TabPanel 等
System.Windows.Converters	这个命名空间包含了用于数据转换的类。但它没有包含所有的转换类。核心转换类在 System.Windows 命名空间中定义
System.Windows.Data	这个命名空间由 WPF 数据绑定使用。其中的一个重要类是 Binding, 它用于定义 WPF 目标元素和 CLR 源之间的绑定
System.Windows.Documents	在处理文档时, 可以使用这个命名空间中的许多类。内容元素 FixedDocument 和 FlowDocument 可以包含这个命名空间中的其他元素。System.Windows.Documents.Serialization 命名空间中的类可以将文档写入磁盘
System.Windows.Ink	Windows Tablet PC 和 Ultra Mobile PC 使用得越来越多。在这些 PC 上, ink 可以用于用户输入。System.Windows.Ink 命名空间包含处理 ink 输入的类
System.Windows.Input	这个命名空间包含的几个类用于命令处理、键盘输入、使用触针等
System.Windows.Interop	这个命名空间中的类用于集成 Win32 和 WPF
System.Windows.Markup	用于 XAML 标记代码的类位于这个命名空间
System.Windows.Media	要使用图像、音频和视频内容, 可以使用这个命名空间中的类
System.Windows.Navigation	这个命名空间包含在窗口之间导航的类
System.Windows.Resources	这个命名空间包含资源的支持类
System.Windows.Shapes	UI 的核心类位于这个命名空间, 如 Line、Ellipse、Rectangle 等
System.Windows.Threading	WPF 元素类似于绑定到单个线程上的 Windows 窗体控件。这个命名空间中的类可以处理多个线程, 例如 Dispatcher 类就属于这个命名空间
System.Windows.Xps	XML Paper Specification(XPS)是一个新的文档规范, Microsoft Word 也支持该规范。在 System.Windows.Xps、System.Windows.Xps.Packaging 和 System.Windows.Xps.Serialization 命名空间中, 包含了创建和传送 XPS 文档的类

34.2 形状

形状是 WPF 的核心元素。利用形状, 可以绘制矩形、线条、椭圆、路径、多边形和多义线等二维图形, 这些图形用派生自抽象类 Shape 的类表示。图形在 System.Windows.Shapes 命名空间中定义。

下面的 XAML 示例绘制了一个带腿的黄色笑脸, 它用一个椭圆表示笑脸, 两个椭圆表示眼睛, 一个路径表示嘴, 四个线条表示腿:

```
<Window x:Class="SimpleControls.Window1"
```

```

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="WPF Samples" Height="260" Width="230">

<Canvas>
  <Ellipse Canvas.Left="50" Canvas.Top="50" Width="100" Height="100"
    Stroke="Blue" StrokeThickness="4" Fill="Yellow" />
  <Ellipse Canvas.Left="60" Canvas.Top="65" Width="25" Height="25"
    Stroke="Blue" StrokeThickness="3" Fill="White" />
  <Ellipse Canvas.Left="70" Canvas.Top="75" Width="5" Height="5" Fill="Black" />
  <Path Stroke="Blue" StrokeThickness="4" Data="M 62,125 Q 95,122 102,108" />
  <Line X1="124" X2="132" Y1="144" Y2="166" Stroke="Blue" StrokeThickness="4" />
  <Line X1="114" X2="133" Y1="169" Y2="166" Stroke="Blue" StrokeThickness="4" />
  <Line X1="92" X2="82" Y1="146" Y2="168" Stroke="Blue" StrokeThickness="4" />
  <Line X1="68" X2="83" Y1="160" Y2="168" Stroke="Blue" StrokeThickness="4" />
</Canvas>

</Window>

```

图 34-5 显示了这些 XAML 代码的结果。

无论是按钮还是线条、矩形等图形，所有这些 WPF 元素都可以通过编程来访问。把 Path 元素的 Name 属性设置为 mouth，就可以用变量名 mouth 以编程方式访问这个元素：

```

<Path Name="mouth" Stroke="Blue" StrokeThickness="4"
  Data="M 62,125 Q 95,122 102,108" />

```

在 Path 元素的后台编码属性 Data 中，mouth 设置为一个新的图形。为了设置路径，Path 类支持 PathGeometry 和路径标记语法。字母 M 定义了路径的起点，字母 Q 指定了二次贝塞尔曲线的一个控制点和终点。运行应用程序，会看到如图 34-6 所示的窗口。

```

public Window1()
{
    InitializeComponent();
    mouth.Data = Geometry.Parse("M 62,125 Q 95,122 102,128");
}

```

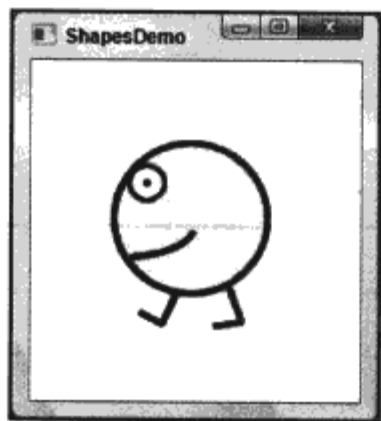


图 34-5

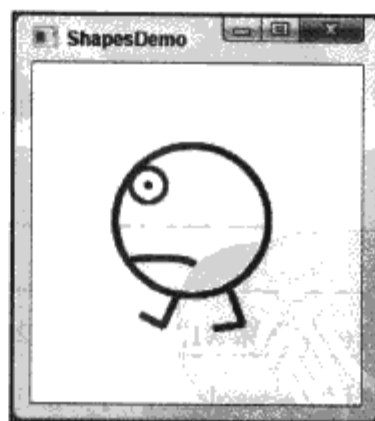


图 34-6

在本章的前面提到，按钮可以包含任何内容。对 XAML 代码做一点儿修改，将 Button 元素作为内容添加到窗口中，会使图形显示在按钮中，如图 34-7 所示。

```

<Window x:Class="ShapesDemo.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="ShapesDemo" Height="260" Width="230">

```

```
<Button Margin="5">
  <Canvas Height="250" Width="220">
    <Ellipse Canvas.Left="50" Canvas.Top="50" Width="100"
      Height="100" Stroke="Blue" StrokeThickness="4"
      Fill="Yellow" />
    <Ellipse Canvas.Left="60" Canvas.Top="65" Width="25"
      Height="25" Stroke="Blue" StrokeThickness="3"
      Fill="White" />
    <Ellipse Canvas.Left="70" Canvas.Top="75" Width="5"
      Height="5" Fill="Black" />
    <Path Name="mouth" Stroke="Blue" StrokeThickness="4"
      Data="M 62,125 Q 95,122 102,108" />
    <Line X1="124" X2="132" Y1="144" Y2="166" Stroke="Blue"
      StrokeThickness="4" />
    <Line X1="114" X2="133" Y1="169" Y2="166" Stroke="Blue"
      StrokeThickness="4" />
    <Line X1="92" X2="82" Y1="146" Y2="168" Stroke="Blue"
      StrokeThickness="4" />
    <Line X1="68" X2="83" Y1="160" Y2="168" Stroke="Blue"
      StrokeThickness="4" />
  </Canvas>
</Button>
</Window>
```

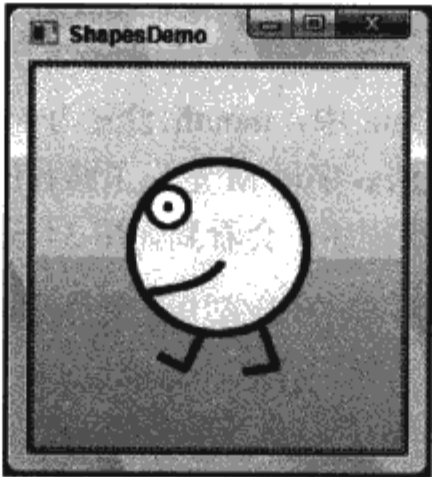


图 34-7

表 34-3 描述了 System.Windows.Shapes 命名空间中的图形。

表 34-3

Shape 类	说 明
Line	可以在坐标 X1,Y1 到 X2,Y2 之间绘制一条线
Rectangle	使用 Rectangle 类, 可以指定 Width 和 Height, 绘制一个矩形
Ellipse	使用 Ellipse 类, 可以绘制一个椭圆
Path	使用 Path 类可以绘制一系列直线和曲线。Data 属性是 Geometry 类型。还可以使用派生自基类 Geometry 的类绘制图形, 或使用路径标记语法来定义图形
Polygon	使用 Polygon 类可以绘制由线段连接而成的封闭图形。多边形由一系列赋予 Points 属性的 Point 对象定义
PolyLine	类似于 Polygon 类, 使用 PolyLine 也可以绘制连接起来的线段, 与多边形的区别是多义线不一定是封闭图形



### 34.3 变换

因为 WPF 基于 DirectX, DirectX 是基于矢量的, 所以可以重置每个元素的大小。基于矢量的图形现在可以缩放、旋转和倾斜。即使没有手工计算位置, 也可以进行单击测试 (例如移动鼠标和单击鼠标)。

给 Canvas 元素的 LayoutTransform 属性添加 ScaleTransform 元素, 如下所示, 把整个笑脸的内容在 X 和 Y 方向上放大 2 倍。

```
<Canvas.LayoutTransform>
  <ScaleTransform ScaleX="2" ScaleY="2" />
</Canvas.LayoutTransform>
```

旋转与缩放的执行方式相同。使用 RotateTransform 元素, 可以定义旋转的角度:

```
<Canvas.LayoutTransform>
  <RotateTransform Angle="40" />
</Canvas.LayoutTransform>
```

对于倾斜, 可以使用 SkewTransform 元素。此时可以指定 X 和 Y 方向的倾斜角度:

```
<Canvas.LayoutTransform>
  <SkewTransform AngleX="20" AngleY="25" />
</Canvas.LayoutTransform>
```

图 34-8 显示了这些变换的结果。这些图放在一个 StackPanel 中。从左到右, 第一个图重置了大小, 第二个图旋转了, 第三个图倾斜了。为了更容易看出它们的区别, 可以把 Canvas 元素的 Background 属性设置为不同的颜色。

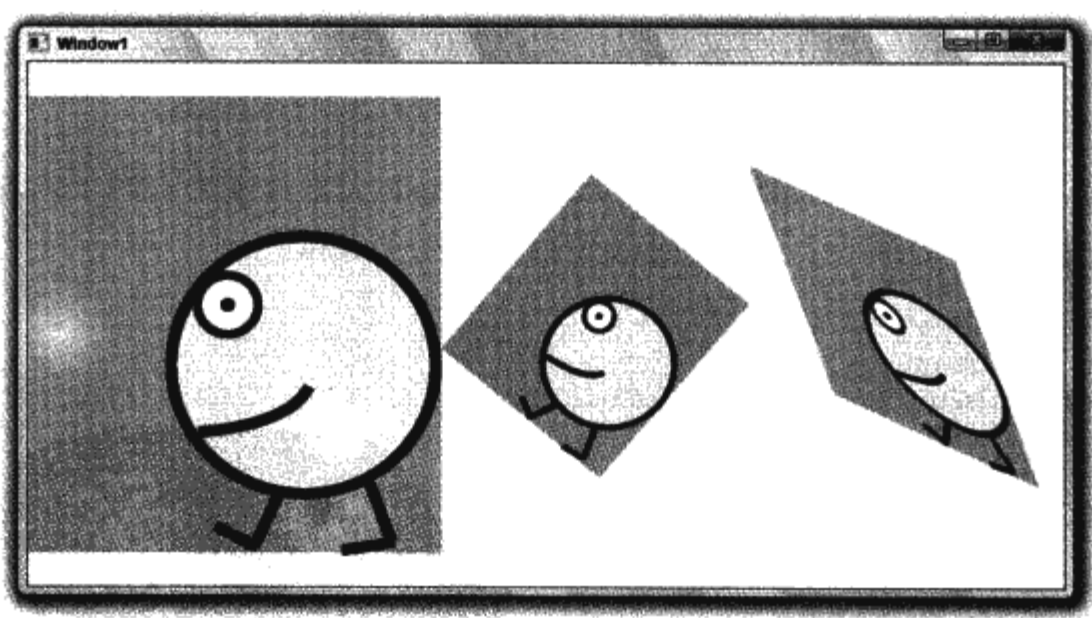


图 34-8

### 34.4 笔刷

本节介绍如何使用 WPF 提供的笔刷绘制背景和前景。本节将参考图 34-9, 它显示了在 Button 元素的 Background 属性上使用各种笔刷的效果。

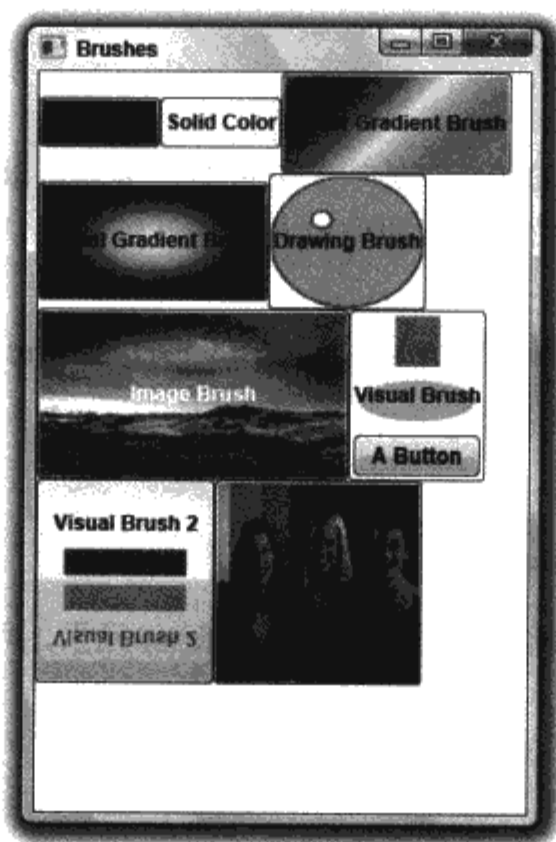


图 34-9

### 34.4.1 SolidColorBrush

图 34-9 中的第一个按钮使用了 SolidColorBrush，顾名思义，这个笔刷使用纯色。完成的区域用同一种颜色绘制。

把 Background 属性设置为定义纯色的字符串，就可以定义纯色。该字符串会转换为一个 SolidColorBrush 元素。

```
<Button Height="30" Background="Purple">Solid Color</Button>
```

当然，设置 Background 子元素，把 SolidColorBrush 元素添加为它的子内容，也可以得到这个效果。应用程序中的第二个按钮给背景使用了纯色 Yellow：

```
<Button Height="30" >
  <Button.Background>
    <SolidColorBrush>Yellow</SolidColorBrush>
  </Button.Background>
  Solid Color
</Button>
```

### 34.4.2 LinearGradientBrush

对于平滑的颜色变化，可以使用 LinearGradientBrush，如第三个按钮所示。这个笔刷定义了 StartPoint 和 EndPoint 属性。使用这些属性可以为线性渐变指定两个坐标。默认的渐变方向是从 0,0 到 1,1 的对角线。定义其他值可以给渐变指定不同的方向。例如，StartPoint 指定为 0,0，EndPoint 指定为 0,1，就得到了一个垂直渐变。StartPoint 不变，EndPoint 指定为 1,0，就得到了一个水平渐变。

在这个笔刷中，可以用 `GradientStop` 元素定义指定偏移位置的颜色值。在各个偏移位置之间，颜色是平滑过渡的。

```
<Button Height="60">
  <Button.Background>
    <LinearGradientBrush StartPoint="0,0"
      EndPoint="0.5,1.2">
      <GradientStop Color="Red" Offset="0"></GradientStop>
      <GradientStop Color="Blue" Offset="0.2">
      </GradientStop>
      <GradientStop Color="BlanchedAlmond" Offset="0.7">
      </GradientStop>
      <GradientStop Color="DarkOrange" Offset="1">
      </GradientStop>
    </LinearGradientBrush>
  </Button.Background>
  Linear Gradient Brush
</Button>
```

### 34.4.3 RadialGradientBrush

使用 `RadialGradientBrush` 可以以放射方式产生平滑的颜色改变。在图 34-9 中，第四个按钮使用了 `RadialGradientBrush`。这个笔刷定义了从 `GradientOrigin` 点开始的颜色。

```
<Button Height="70" >
  <Button.Background>
    <RadialGradientBrush Center="0.5,0.5"
      GradientOrigin="0.5,0.5"
      RadiusX="0.5" RadiusY="0.5" SpreadMethod="Pad">
      <GradientStop Color="White" Offset="0" />
      <GradientStop Color="LightBlue" Offset="0.4" />
      <GradientStop Color="DarkBlue" Offset="1" />
    </RadialGradientBrush>
  </Button.Background>
  Radial Gradient Brush
</Button>
```

### 34.4.4 DrawingBrush

`DrawingBrush` 可以定义用笔刷绘制的图形。用笔刷绘制的图形在 `GeometryDrawing` 元素中定义。`Geometry` 属性中的 `GeometryGroup` 元素包含 `Geometry` 元素，例如 `EllipseGeometry`、`LineGeometry`、`RectangleGeometry` 和 `CombineGeometry`。

```
<Button Height="80">
  <Button.Background>
    <DrawingBrush>
      <DrawingBrush.Drawing>
        <GeometryDrawing Brush="LightBlue">
          <GeometryDrawing.Geometry>
            <GeometryGroup>
              <EllipseGeometry RadiusX="30" RadiusY="30"
                Center="20,20" />
              <EllipseGeometry RadiusX="4" RadiusY="4"
                Center="10,10" />
            </GeometryGroup>
          </GeometryDrawing.Geometry>
        </GeometryDrawing>
      </DrawingBrush.Drawing>
    </DrawingBrush>
  </Button.Background>
  Drawing Brush
</Button>
```

```

        </GeometryDrawing.Geometry>
        <GeometryDrawing.Pen>
            <Pen>
                <Pen.Brush>Red
            </Pen.Brush>
        </Pen>
    </GeometryDrawing.Pen>
</GeometryDrawing>
</DrawingBrush.Drawing>
</DrawingBrush>
</Button.Background>
Drawing Brush
</Button>

```

### 34.4.5 ImageBrush

要把图像加载到笔刷中，可以使用 `ImageBrush` 元素。在这个元素中，显示了 `ImageSource` 属性定义的图像。

```

<Button Height="100">
    <Button.Background>
        <ImageBrush
            ImageSource=" C:\Windows\Web\Wallpaper\img21.bmp"
        />
    </Button.Background>
    <Button.Foreground>White</Button.Foreground>
    Image Brush
</Button>

```

### 34.4.6 VisualBrush

`VisualBrush` 可以在笔刷中使用其他 WPF 元素。下面给 `Visual` 属性添加一个 WPF 元素。图 34-9 中的第 7 个按钮包含一个矩形、一个椭圆和一个按钮。

```

<Button Height="100">
    <Button.Background>
        <VisualBrush >
            <VisualBrush.Visual>
                <StackPanel Background="White">
                    <Rectangle Width="25" Height="25"
                        Fill="LightCoral" Margin="2" />
                    <Ellipse Width="65" Height="20"
                        Fill="Aqua" Margin="5" />
                    <Button Margin="2">A Button</Button>
                </StackPanel>
            </VisualBrush.Visual>
        </VisualBrush>
    </Button.Background>
    Visual Brush
</Button>

```

在 `VisualBrush` 中，还可以创建反射等效果。这里显示的按钮包含一个 `StackPanel`，它包含一个边框和一个矩形。边框包含一个 `StackPanel`，该 `StackPanel` 又包含一个标签和一个矩形。但这不是重点。第二个矩形是用 `VisualBrush` 填充的。这个笔刷定义了一个不透明值和一个变换。`Visual` 属性绑定到 `Border` 元素上。变换是通过设置 `VisualBrush` 的 `RelativeTransform` 属性

来完成的。这个变换使用了相对坐标。把 `ScaleY` 设置为 -1, 完成了 Y 向的反射。`TranslateTransform` 在 Y 向上移动变换, 使反射效果位于原对象的下面。图 34-9 中的第 8 个按钮(`VisualBrush2`)显示了其效果。

#### 提示:

这里使用的数据绑定和 `Binding` 元素详见下一章。

```
<Button Height="120">
  <StackPanel>
    <Border x:Name="reflected">
      <Border.Background>Yellow</Border.Background>
      <StackPanel>
        <Label>Visual Brush 2</Label>
        <Rectangle Width="70" Height="15" Margin="2"
          Fill="BlueViolet" />
      </StackPanel>
    </Border>
    <Rectangle Height="30">
      <Rectangle.Fill>
        <VisualBrush Opacity="0.35" Stretch="None"
          Visual="{Binding ElementName=reflected}">
          <VisualBrush.RelativeTransform>
            <TransformGroup>
              <ScaleTransform ScaleX="1" ScaleY="-1" />
              <TranslateTransform Y="1" />
            </TransformGroup>
          </VisualBrush.RelativeTransform>
        </VisualBrush>
      </Rectangle.Fill>
    </Rectangle>
  </StackPanel>
</Button>
```

只要把 `Visual` 属性设置为 `MediaElement`, 就可以使用 `VisualBrush` 显示视频。对于 `MediaControl`, `Source` 属性应设置为 WMV 文件。在图 34-9 中, 第 9 个按钮显示了 3 个女人, 它就是显示视频的一个例子。但是在纸质媒介中, 很难显示视频。读者可以自己试一试——如果使用 Windows Vista 的 Ultimate 版本, 就可以在硬盘上找到这个视频。否则, 可以选择另一个视频文件。

```
<Button Height="120">
  <Button.Background>
    <VisualBrush>
      <VisualBrush.Visual>
        <MediaElement x:Name="video"
          Source="C:\Windows\ehome\ColorTint.wmv" />
      </VisualBrush.Visual>
    </VisualBrush>
  </Button.Background>
</Button>
```



### 34.5 控件

可以给 WPF 使用上百个控件。为了更好地理解它们，我们把控件分为如下类别：

- 简单控件
- 内容控件
- 有标题的内容控件
- 项控件
- 有标题的项控件

#### 34.5.1 简单控件

简单控件是没有 Content 属性的控件。例如，Button 类可以包含任意图形、任意元素，这对于简单控件而言没有问题。表 34-4 列出了简单控件及其功能。

表 34-4

简单控件	说 明
PasswordBox	PasswordBox 控件用于输入密码。这个控件有用于输入密码的特殊属性，例如 PasswordChar 定义了当用户输入密码时显示的字符，Password 可以访问输入的密码。PasswordChanged 事件在修改密码时调用
ScrollBar	ScrollBar 控件包含一个 Thumb，用户可以在 Thumb 中选择一个值。如果文档在屏幕中放不下，就可以使用滚动条。一些控件包含滚动条，如果内容过多，就显示滚动条
ProgressBar	使用 ProgressBar 控件，可以指示某个时间较长的操作的进度
Slider	使用 Slider 控件，用户可以移动 Thumb，选择一个范围的值。ScrollBar、ProgressBar 和 Slider 派生自同一个基类 RangeBase
Textbox	Textbox 控件用于显示简单的无格式文本
RichTextbox	RichTextbox 控件通过 FlowDocument 类支持带格式的文本。RichTextBox 和 TextBox 派生自同一个基类 TextBoxBase

提示：

尽管简单控件没有 Content 属性，但通过定义模板，完全可以定制这些控件的外观。模板详见本章后面的内容。

#### 34.5.2 内容控件

ContentControl 有 Content 属性，利用 Content 属性，可以给控件添加任意内容。Button 类派生自基类 ContentControl，所以可以在这个控件中添加任意内容。在上面的例子中，Button 中有一个 Canvas 控件。表 34-5 列出了内容控件。

表 34-5

ContentControl 控件	说 明
Button RepeatButton ToggleButton CheckBox RadioButton	类 Button、RepeatButton、ToggleButton 和 GridViewColumnHeader 派生自同一个基类 ButtonBase。所有这些按钮都响应 Click 事件。RepeatButton 会重复响应 Click 事件，直到释放按钮为止  ToggleButton 是 CheckBox 和 RadioButton 的基类。这些按钮有开关状态。CheckBox 可以由用户选择和取消选择，RadioButton 可以由用户选择。清除 RadioButton 的选择必须通过编程来实现
Label	Label 类表示控件的文本标签。这个类也支持访问键，例如菜单命令
Frame	Frame 控件支持导航。使用 Navigate()方法可以导航到一个页面内容上。如果该内容是一个网页，就使用浏览器控件来显示
ListBoxItem	ListBoxItem 是 ListBox 控件中的一项
StatusBarItem	StatusBarItem 是 StatusBar 控件中的一项
ScrollViewer	ScrollViewer 是一个包含滚动条的内容控件，可以把任意内容放入这个控件，滚动条会在需要时显示
ToolTip	ToolTip 创建一个弹出窗口，显示控件的附加信息
UserControl	将 UserControl 类用作基类，可以为创建定制控件提供一种简单方式。但是，基类 UserControl 不支持模板
Window	Window 类可以创建窗口和对话框。使用这个类，会获得一个带有最小化/最大化/关闭按钮和系统菜单的框架。在显示对话框时，可以使用方法 ShowDialog()，方法 Show()会打开一个窗口
NavigationWindow	类 NavigationWindow 派生自 Window 类，支持内容导航

只有 Frame 控件包含在下面 XAML 代码的 Window 中。Source 属性设置为 <http://www.wrox.com>，所以 Frame 控件导航到这个网站上，如图 34-10 所示。



图 34-10

```
<Window x:Class="FrameSample.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="FrameSample" Height="400" Width="400">
  <Frame Source="http://www.wrox.com" />
</Window>
```

34.5.3 有标题的内容控件

带标题的内容控件派生自 HeaderContentControl 基类。HeaderContentControl 类又派生自基类 ContentControl。HeaderContentControl 类的 Header 属性定义了标题的内容，HeaderTemplate 属性可以对标题进行完全的定制。派生自基类 HeaderContentControl 的控件如表 34-6 所示。

表 34-6

HeaderContentControl	说 明
Expander	使用 Expander 控件，可以创建一个带对话框的"高级"模式，它在默认情况下不显示所有的信息，只有用户展开它，才会显示更多的信息。在未展开模式下，只显示标题信息，在展开模式下显示内容
GroupBox	GroupBox 控件提供了边框和标题来组合控件
TabItem	TabItem 控件是 TabControl 类中的项。TabItem 的 Header 属性定义了标题的内容，这些内容用 TabControl 的标签显示

Expander 控件的简单用法如下面的例子所示。Expander 控件的属性 Header 设置为 Click for more。这个文本用于显示扩展。这个控件的内容只有在控件展开时才显示。图 34-11 中的示例程序包含折叠的 Expander 控件，图 34-12 中的示例程序展开了 Expander 控件。

```
<Window x:Class="ExpanderSample.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Expander Sample" Height="300" Width="300">
  <StackPanel>
    <TextBlock>Short information</TextBlock>
    <Expander Header="Click for more">
      <Border Height="200" Width="200" Background="Yellow">
        <TextBlock HorizontalAlignment="Center"
          VerticalAlignment="Center">
          More information here!
        </TextBlock>
      </Border>
    </Expander>
  </StackPanel>
</Window>
```



图 34-11

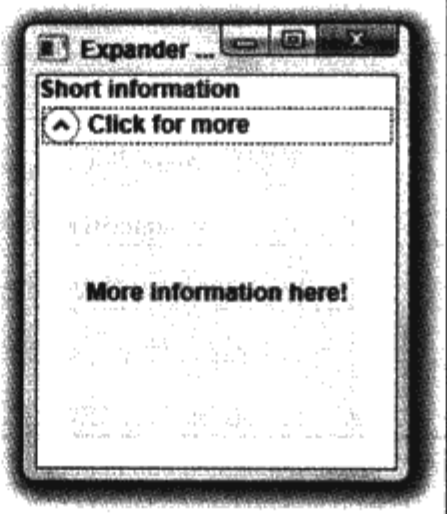


图 34-12

提示：

如果在展开 Expander 控件时，要修改该控件的标题文本，可以创建一个触发器，触发器详见本章后面的内容。

34.5.4 项控件

类 ItemsControl 包含一系列可以用 Items 属性访问的数据项。派生自 ItemsControl 的类如表 34-7 所示。

表 34-7

ItemsControl	说 明
Menu ContextMenu	类 Menu 和 ContextMenu 派生自抽象基类 MenuBase。把 MenuItem 元素放在数据项列表和相关的命令中，就可以给用户提供了菜单
StatusBar	StatusBar 控件通常显示在应用程序的底部，为用户提供状态信息。可以把 StatusBarItem 元素放在 StatusBar 列表中
TreeView	要分层显示数据项，可以使用 TreeView 控件
ListBox ComboBox TabControl	ListBox、ComboBox 和 TabControl 都有相同的抽象基类 Selector。这个基类可以从列表中选择数据项。ListBox 显示列表中的数据项，ComboBox 有一个附带的 Button 控件，只有点击按钮，才会显示数据项。在 TabControl 中，内容可以排列为表格

34.5.5 带标题的项控件

HeaderItemsControl 是包含数据项和标题的控件的基类。类 HeaderItemsControl 派生自 ItemsControl。

派生自 HeaderItemsControl 的类如表 34-8 所示。

表 34-8

HeaderedItemsControl	说 明
MenuItem	菜单类 Menu 和 ContextMenu 包含 MenuItem 类型的数据项。菜单项可以连接到命令上，因为 MenuItem 类实现了接口 ICommandSource
TreeViewItem	TreeViewItem 类可以包含 TreeViewItem 类型的数据项
ToolBar	ToolBar 控件是一组控件(通常是 Button 和 Separator 元素)的容器。可以将 ToolBar 放在 ToolBarTray 中，它会重新安排 ToolBar 控件的位置

34.6 布局

为了定义应用程序的布局，可以使用派生自 Panel 基类中的类。这里讨论几个布局容器。布局容器要完成两个主要任务：测量和安排。在测量时，容器要求其子控件有合适的大小。由于控件的大小不一定合适，所以容器需要确定和安排其子控件的大小和位置。

34.6.1 StackPanel

Window 可以只包含一个元素，作为其内容。如果要包含多个元素，可以将 StackPanel 用作 Window 的一个子元素，在 StackPanel 中添加元素。StackPanel 是一个简单的容器控件，只能一个一个地显示元素。StackPanel 的显示方向可以是水平或垂直。类 ToolBarPanel 派生自 StackPanel。

```
<Window x:Class="LayoutSamples.Window1"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Layout Samples" Height="300" Width="283">

    <StackPanel Orientation="Vertical">
        <Label>Label</Label>
        <TextBox>TextBox</TextBox>
        <CheckBox>Checkbox</CheckBox>
        <CheckBox>Checkbox</CheckBox>
        <ListBox>
            <ListBoxItem>ListBoxItem One</ListBoxItem>
            <ListBoxItem>ListBoxItem Two</ListBoxItem>
        </ListBox>
        <Button>Button</Button>
    </StackPanel>

</Window>
```

在图 34-13 中，可以看到 StackPanel 垂直显示的子控件。

提示：

对于与 StackPanel 的数据绑定项，如果没有足够的空间显示所有的项，就可以使用 VirtualizingStackPanel。这个面板只生成显示的项。



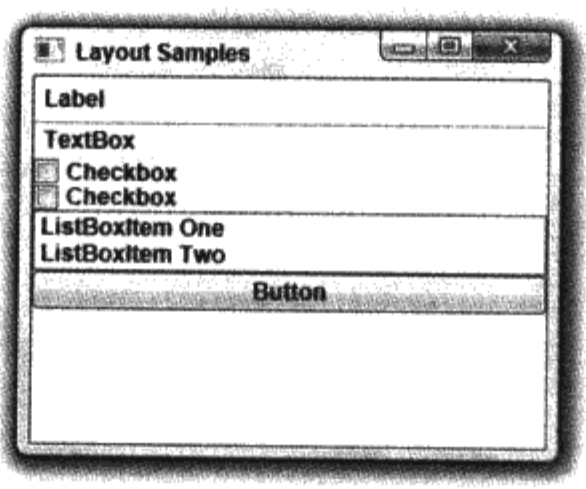


图 34-13

### 34.6.2 WrapPanel

WrapPanel 将子元素自左向右地排列，若一个水平行中放不下，就排在下一行。面板的排列方向可以是水平或垂直。

```
<Window x:Class="LayoutSamples.WrapPanelDemo"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Layout Samples" Height="160" Width="250">
```

```
<WrapPanel>
  <Button Width="100">Button</Button>
  <Button Width="100">Button</Button>
  <Button Width="100">Button</Button>
  <Button Width="100">Button</Button>
  <Button Width="100">Button</Button>
  <Button Width="100">Button</Button>
  <Button Width="100">Button</Button>
  <Button Width="100">Button</Button>
</WrapPanel>
```

```
</Window>
```

图 34-14 显示了面板的排列结果。如果重新设置了应用程序的大小，按钮会重新排列，填满一行。

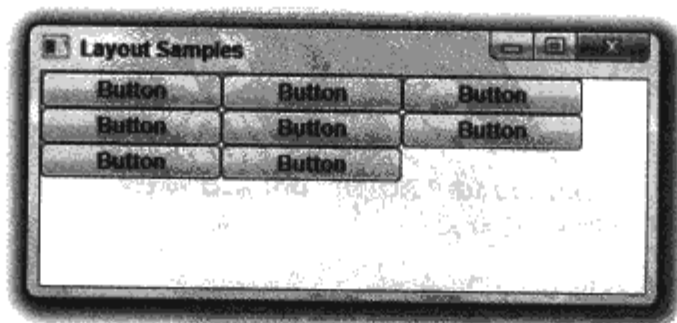


图 34-14

### 34.6.3 Canvas

Canvas 是一个允许显式指定控件位置的面板。它定义了相关的属性 Left、Right、Top 和

Bottom, 这些属性可以由子元素在面板中定位时使用。

```
<Window x:Class="LayoutSamples.CanvasDemo"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Layout Samples" Height="300" Width="300">

  <Canvas Background="LightBlue">
    <Label Canvas.Top="30" Canvas.Left="20">Enter here:</Label>
    <TextBox Canvas.Top="30" Canvas.Left="130" Width="100"></TextBox>
    <Button Canvas.Top="70" Canvas.Left="130">Click Me!</Button>
  </Canvas>

</Window>
```

图 34-15 显示了 Canvas 面板的结果, 其中定位了子元素 Label、TextBox 和 Button。

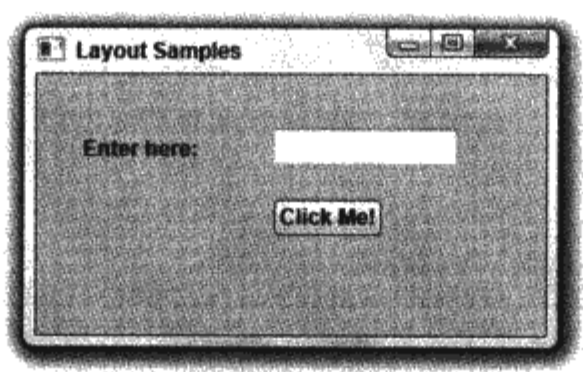


图 34-15

#### 34.6.4 DockPanel

DockPanel 非常类似于 Windows 窗体的停靠功能。DockPanel 可以指定安排子控件的区域。DockPanel 定义了附带属性 Dock, 可以在控件的子控件中将它设置为 Left、Right、Top 和 Bottom。图 34-16 显示了安排在 DockPanel 中的带边框的文本块。为了便于区别, 为不同的区域指定了不同的颜色:

```
<Window x:Class="LayoutSamples.DockPanelDemo"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Layout Samples" Height="300" Width="300">
```

```
<DockPanel Background="LightBlue">
  <Border Height="25" Background="AliceBlue" DockPanel.Dock="Top">
    <TextBlock>Menu</TextBlock>
  </Border>
  <Border Height="25" Background="Aqua" DockPanel.Dock="Top">
    <TextBlock>Toolbar</TextBlock>
  </Border>
  <Border Height="30" Background="LightSteelBlue" DockPanel.Dock="Bottom">
    <TextBlock>Status</TextBlock>
  </Border>
  <Border Width="80" Background="Azure" DockPanel.Dock="Left">
    <TextBlock>Left Side</TextBlock>
  </Border>
  <Border Background="HotPink">
    <TextBlock>Remaining Part</TextBlock>
  </Border>
</DockPanel>
```

```

    </Border>
  </DockPanel>

</Window>

```

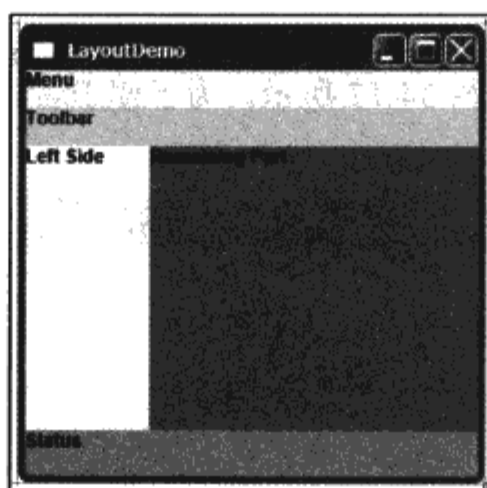


图 34-16

### 34.6.5 Grid

使用 Grid，可以在行和列中放置控件。对于每一列，可以指定 ColumnDefinition，对于每一行，可以指定 RowDefinition。下面的示例代码显示两列和三行。在每一列、每一行中，都可以指定宽度或高度。ColumnDefinition 有一个 Width 依赖属性，RowDefinition 有一个 Height 依赖属性。可以以像素、厘米、英寸或点为单位定义高度和宽度，或者把它们设置为 Auto，根据内容来确定其大小。Grid 还允许根据具体情况指定大小，即根据可用的空间以及与其他行和列的相对位置，计算行和列的空间。在为列提供可用空间时，可以将 Width 属性设置为\*，要使某一列的空间是另一列的 2 倍，应指定 2\*。下面的示例代码定义了两列和三行，但没有定义列和行的其他设置，默认使用根据具体情况指定大小的设置。

这个 Grid 包含几个 Label 和 TextBox 控件。这些控件的父控件是 Grid，所以可以设置相关的属性 Column、ColumnSpan、Row 和 RowSpan。

```

<Window x:Class="LayoutSamples.GridDemo"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Layout Samples" Height="300" Width="283">

  <Grid ShowGridLines="True">
    <Grid.ColumnDefinitions>
      <ColumnDefinition />
      <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition />
      <RowDefinition />
    </Grid.RowDefinitions>
    <Label Grid.Column="0" Grid.ColumnSpan="2" Grid.Row="0"
      VerticalAlignment="Center" HorizontalAlignment="Center">Title</Label>
    <Label Grid.Column="0" Grid.Row="1" VerticalAlignment="Center">
      Firstname:</Label>
    <TextBox Grid.Column="1" Grid.Row="1" Width="100" Height="30"></TextBox>

```

```
<Label Grid.Column="0" Grid.Row ="2" VerticalAlignment="Center">
    Lastname:</Label>
<TextBox Grid.Column="1" Grid.Row="2" Width="100" Height="30"></TextBox>
</Grid>

</Window>
```

在 Grid 中安排控件的结果如图 34-17 所示。为了便于看到列和行，ShowGridLines 属性设置为 true。

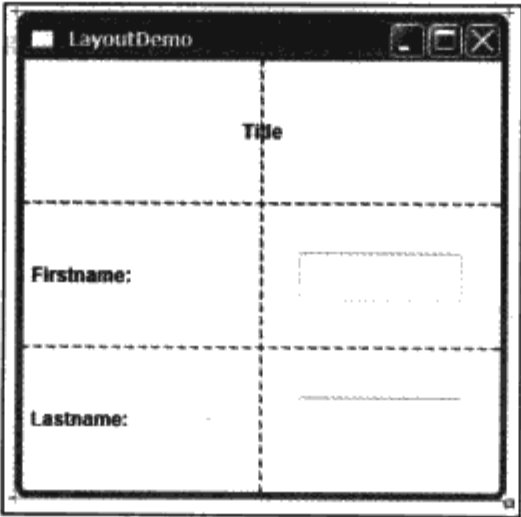


图 34-17

提示：  
要使 Grid 的每个单元格有相同的尺寸，可以使用 UniformGrid 类。

34.7 事件处理

WPF 类定义了能添加处理程序的事件，例如 MouseEnter、MouseLeave、MouseMove、Click 等。它们基于 .NET 中的事件和委托机制。 .NET 中的事件和委托机制详见第 7 章。

在 WPF 中，可以用 XAML 或在后台代码中指定事件处理程序。在 button1 中，XML 特性 Click 用于将方法 button\_Click 赋予单击事件。 Button2 没有在 XAML 中指定事件处理程序：

```
<Button Name="button1" Click="button_Click">Button 1</Button>
<Button Name="button2" >Button 2</Button>
```

在后台代码中，创建了 RoutedEventHandler 委托的一个实例，将 button\_Click 方法传送给该委托，从而指定了 Button2 的 Click 事件。在两个按钮中调用的方法 button\_Click ()，定义了 RoutedEventHandler 委托描述的变元：

```
public Window1()
{
    InitializeComponent();
    button2.Click += button_Click;
}

void OnButtonClick(object sender, RoutedEventArgs e)
{
    MessageBox.Show("Click Event");
}
```

WPF 的事件处理机制基于 .NET 事件，但扩展了起泡和通道特性。如前所述，Button 可以包含图形、列表框、另一个按钮等。如果 CheckBox 包含在 Button 中，单击了 CheckBox，会发生什么情况？事件应到达什么地方？答案是事件会起泡。首先，Click 事件会到达 CheckBox，然后向上起泡，到达 Button。这样，就可以用 Button 的 Click 事件处理包含在该 Button 中的所有元素的 Click 事件。

一些事件有通道事件，其他事件有起泡事件。通道事件先到达外部的元素，再沿着通道到达内部的元素。起泡事件先从内部元素开始，再到达外部的元素。通道和起泡事件通常是成对的。通道事件的前缀是 Preview，例如 PreviewMouseMove。这个事件从外部控件到达内部控件。在 PreviewMouseMove 事件后，会发生 MouseMove 事件。这个事件是一个起泡事件，它从内部控件到达外部控件。

将事件变元的 Handled 属性设置为 true，就可以关闭事件的通道和起泡功能。Handled 属性是 RoutedEventArgs 类的一个成员，所有参与通道和起泡特性的事件处理程序都有一个 RoutedEventArgs 类型或派生自 RoutedEventArgs 类型的事件变元。

**提示：**

如果将事件变元的 Handled 属性设置为 true，关闭事件的通道功能，则通道事件之后的起泡事件也不再会发生。

## 34.8 样式、模板和资源

设置 Button 元素的 FontSize 和 Background 属性，就可以定义 WPF 元素的外观和操作方式，如下所示：

```
<StackPanel>
  <Button Name="button1" Width="150" FontSize="12" Background="AliceBlue">
    Click Me!
  </Button>
</StackPanel>
```

除了定义每个元素的外观和操作方式之外，还可以定义用资源存储的样式。为了完全定制控件的外观，可以使用模板，再把它们存储到资源中。

### 34.8.1 样式

要定义样式，可以使用包含 Setter 元素的 Style 元素。使用 Setter，可以指定样式的 Property 和 Value，例如属性 Button.Background 和值 AliceBlue。

为了把样式赋予指定的元素，可以将样式赋予某一类型的所有元素，或者为该样式使用一个键。要把样式赋予某一类型的所有元素，可使用 Style 的 TargetType 属性，指定 x>Type 标记扩展(x>Type Button)，将样式赋予一个按钮。

```
<Window.Resources>
  <Style TargetType="{x>Type Button}">
    <Setter Property="Button.Background" Value="LemonChiffon" />
  </Style>
</Window.Resources>
```



```
<Setter Property="Button.FontSize" Value="18" />
</Style>
<Style x:Key="ButtonStyle">
  <Setter Property="Button.Background" Value="AliceBlue" />
  <Setter Property="Button.FontSize" Value="18" />
</Style>
</Window.Resources>
```

在下面的 XAML 代码中，button2 没有用元素属性定义样式，而是使用为 Button 类型定义的样式。button3 的 Style 属性用 StaticResource 标记扩展设置为 {StaticResource ButtonStyle}，其中 ButtonStyle 指定了前面定义的样式资源的键值，所以 button3 的背景为 AliceBlue。

```
<Button Name="button2" Width="150">Click Me!</Button>
<Button Name="button3" Width="150" Style="{StaticResource ButtonStyle}">
  Click Me, Too!
</Button>
```

除了把按钮的 Background 设置为单个值之外，还可以将 Background 属性设置为定义了渐变色的 LinearGradientBrush，如下所示：

```
<Style x:Key="FancyButtonStyle">
  <Setter Property="Button.FontSize" Value="22" />
  <Setter Property="Button.Foreground" Value="White" />
  <Setter Property="Button.Background">
    <Setter.Value>
      <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
        <GradientStop Offset="0.0" Color="LightCyan" />
        <GradientStop Offset="0.14" Color="Cyan" />
        <GradientStop Offset="0.7" Color="DarkCyan" />
      </LinearGradientBrush>
    </Setter.Value>
  </Setter>
</Style>
```

button4 的样式采用青色的线性渐变效果：

```
<Button Name="button4" Width="200" Style="{StaticResource FancyButtonStyle}">
  Fancy!
</Button>
```

图 34-18 显示了这些按钮的效果。



图 34-18

34.8.2 资源

从样式示例可以看出，样式通常存储在资源中。可以在资源中定义任意元素，例如，前面为按钮的背景样式创建了笔刷，它本身就可以定义为一个资源，这样就可以在需要笔刷的地方使用它了。

下面的示例在 StackPanel 资源中定义了 LinearGradientBrush，它的键名是 MyGradientBrush。Button1 使用 StaticResource 标记扩展将 Background 属性赋予 MyGradientBrush 资源。图 34-19 显示了这段 XAML 代码的结果。

```
<Window x:Class="ResourcesSample.Window1"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Resources Sample" Height="100" Width="300"
        >
    <Window.Resources>
    </Window.Resources>
    <StackPanel>
        <StackPanel.Resources>
            <LinearGradientBrush x:Key="MyGradientBrush" StartPoint="0.5,0" EndPoint="0.5,1">
                <GradientStop Offset="0.0" Color="LightCyan" />
                <GradientStop Offset="0.14" Color="Cyan" />
                <GradientStop Offset="0.7" Color="DarkCyan" />
            </LinearGradientBrush>
        </StackPanel.Resources>
        <Button Name="button1" Width="200" Height="50" Foreground="White"
            Background="{StaticResource MyGradientBrush}">
            Click Me!
        </Button>
    </StackPanel>
</Window>
```

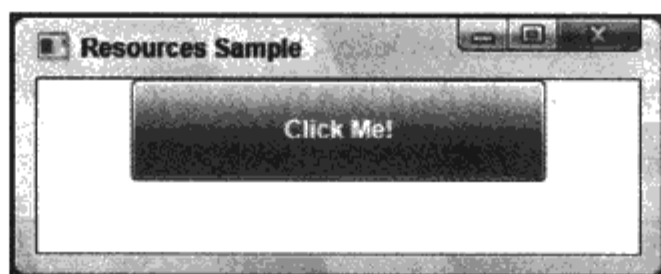


图 34-19

这里，资源用 StackPanel 定义。在上面的例子中，资源用 Window 元素定义。基类 FrameworkElement 定义了 ResourceDictionary 类型的 Resources 属性。这就是资源可以用派生自 FrameworkElement 的所有类(任意 WPF 元素)来定义的原因。

资源是按层次结构来搜索的。如果用 Window 元素定义资源，它就会应用于 Window 的所有子元素。如果 Window 包含一个 Grid，该 Grid 又包含一个 StackPanel，资源是用 StackPanel 定义的，该资源就会应用于 StackPanel 中的所有控件。如果 StackPanel 包含一个按钮，但只为该按钮定义了资源，这个样式就只对该按钮有效。

#### 警告：

对于层次结构，需要注意是否为样式使用了没有 Key 的 TargetType。如果用 Canvas 元素定义了一个资源，并把样式的 TargetType 设置为应用于 TextBox 元素，该样式就会应用于 Canvas 中的所有 TextBox 元素。如果 Canvas 中有一个 ListBox，该样式甚至会应用于 ListBox 包含的 TextBox 元素。

如果需要将一个样式应用于多个窗口，就可以为应用程序定义样式。在一个 Visual Studio WPF 项目中，创建 App.xaml 文件，以定义应用程序的全局资源。应用程序样式对其中的每个

窗口都有效；每个元素都可以访问为应用程序定义的资源。如果在父窗口中找不到资源，就可以在整个应用程序范围内搜索它。

```
<Application x:Class="StylingDemo.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  StartupUri="Window1.xaml">
  <Application.Resources>
  </Application.Resources>
</Application>
```

## 1. 系统资源

还有许多系统级的颜色和字体资源，可用于所有应用程序。这些资源用 `SystemColors`、`SystemFonts` 和 `SystemParameters` 类定义。

- 使用 `SystemColors` 可以获得边框、控件、桌面和窗口的颜色设置，例如，`ActiveBorder Color`、`ControlBrush`、`DesktopColor`、`WindowColor`、`WindowBrush` 等。
- 类 `SystemFonts` 返回菜单、状态栏、消息框等的字体设置，例如 `CaptionFont`、`DialogFont`、`MenuFont`、`MessageBoxFont`、`StatusFont` 等。
- 类 `SystemParameters` 设置了菜单按钮、光标、图标、边框标题、时间信息、键盘设置的大小。例如 `BorderWidth`、`CaptionHeight`、`CaptionWidth`、`MenuButtonWidth`、`MenuPopupAnimation`、`MenuShowDelay`、`SmallIconHeight`、`SmallIconWidth` 等。

在图 34-20 显示的对话框中，用户可以配置这些设置。在控制面板上，可以找到带 `Personalization` 设置的 `Appearance` 对话框。

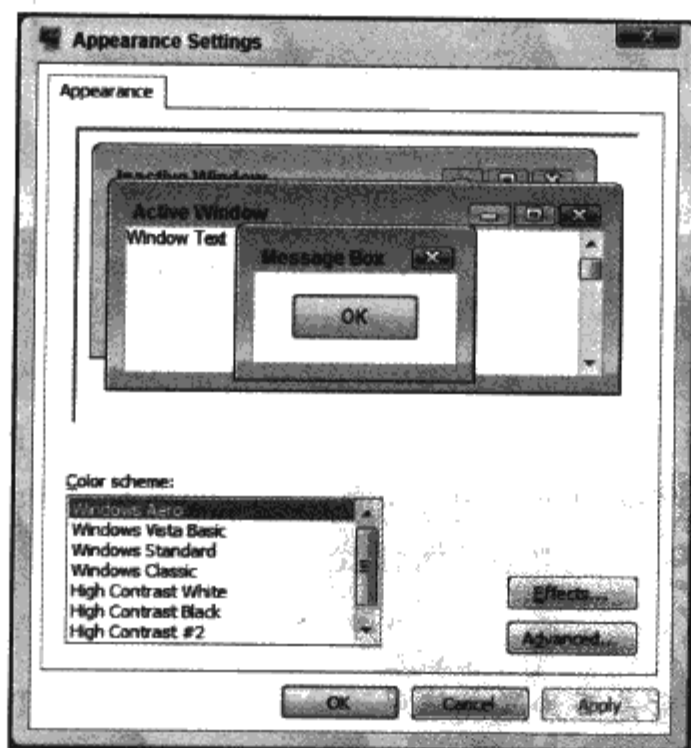


图 34-20

## 2. 在代码中访问资源

要在后台代码中访问资源，基类 `FrameworkElement` 实现了方法 `FindResource()`，所以可以给每个 WPF 对象调用方法 `FindResource()`。

为此, button2 没有指定背景, 但将 Click 事件赋予了方法 button1\_Click()。

```
<StackPanel Name="MyContainer">
  <StackPanel.Resources>
    <LinearGradientBrush x:Key="MyGradientBrush" StartPoint="0.5,0" EndPoint="0.5,1">
      <GradientStop Offset="0.0" Color="LightCyan" />
      <GradientStop Offset="0.14" Color="Cyan" />
      <GradientStop Offset="0.7" Color="DarkCyan" />
    </LinearGradientBrush>
  </StackPanel.Resources>
  <Button Name="button2" Width="200" Height="50" Click="button1_Click">

    Apply Resource Programmatically
  </Button>
```

在方法 button1\_Click() 的实现代码中, 给单击的按钮调用方法 FindResource(), 按照层次结构搜索资源 MyGradientBrush, 将该笔刷应用于控件的 Background 属性。

```
public void OnApplyResource(object sender, RoutedEventArgs e)
{
    Control ctrl = sender as Control;
    ctrl.Background = ctrl.FindResource("MyGradientBrush") as Brush;
}
```

#### 警告:

如果方法 FindResource() 没有找到资源键, 会抛出一个异常。如果不知道资源是否可用, 就可以使用 TryFindResource() 方法来替代。如果找不到资源, TryFindResource() 方法就返回 null。

### 3. 动态资源

在 StaticResource 标记扩展中, 资源是在加载期间搜索。如果在运行程序的过程中改变了资源, 就应使用 DynamicResource 标记扩展。

下面的例子使用与前面相同的资源。button1 把资源用作 StaticResource, button3 使用 DynamicResource 标记扩展把资源用作 DynamicResource。button2 用于以编程方式改变资源。它的 Click 事件处理程序方法赋予了 button2\_Click。

```
<Button Name="button1" Width="200" Height="50"
  Background="{StaticResource MyGradientBrush}">
  Static Resource
</Button>
<Button Name="button2" Width="200" Height="50"
  Click="button2_Click">
  Change Resource
</Button>
<Button Name="button3" Width="200" Height="50"
  Background="{DynamicResource MyGradientBrush}">
  Dynamic Resource
</Button>
```

button2\_Click() 方法的实现代码清除了 StackPanel 的资源, 用 MyGradientBrush 名称添加了一个新资源。这个新资源非常类似于在 XAML 代码中定义的资源, 它只定义了不同的颜色。

```
public void button2_Click(object sender, RoutedEventArgs e)
{
    MyContainer.Resources.Clear();
```



```

LinearGradientBrush brush = new LinearGradientBrush();
brush.StartPoint = new Point(0.5, 0);
brush.EndPoint = new Point(0.5, 1);
GradientStopCollection stops = new GradientStopCollection();
stops.Add(new GradientStop(Colors.White, 0.0));
stops.Add(new GradientStop(Colors.Yellow, 0.14));
stops.Add(new GradientStop(Colors.YellowGreen, 0.7));
brush.GradientStops = stops;
MyContainer.Resources.Add("MyGradientBrush", brush);
}

```

如果运行应用程序，单击第三个按钮，动态修改资源，button4 就会立即获得新资源。用 StaticResource 定义的 button1 仍旧加载旧资源。

#### 警告：

DynamicResource 需要的性能资源比 StaticResource 更多，因为其资源总是在需要时加载。只有需要在运行期间进行修改，才给资源使用 DynamicResource。

#### 4. 触发器

使用触发器，可以动态修改控件的外观和操作方式，因为一些事件或属性值改变了。例如，用户在按钮上移动鼠标，按钮就会改变其外观。通常，这必须在 C#代码中实现，但使用 WPF，也可以用 XAML 实现，而这只会影响 UI。

Style 类有一个 Triggers 属性，通过它可以指定属性触发器。下面的示例将两个 TextBox 元素放在 Canvas 面板中。利用 Window 资源定义一个样式 TextBoxStyle，TextBox 元素使用 Style 属性引用该样式。TextBoxStyle 指定，将 Background 设置为 LightBlue，FontSize 设置为 17。这是应用程序启动时 TextBox 元素的样式。使用触发器可以改变控件的样式。触发器在 Style.Triggers 元素中用 Trigger 元素定义。将一个触发器赋予属性 IsMouseOver，另一个触发器赋予属性 IsKeyboardFocused。这两个属性通过应用了样式的 TextBox 类定义。如果属性 IsMouseOver 的值是 true，就会引发触发器，将 Background 属性设置为 Red，FontSize 设置为 22。如果 TextBox 得到了焦点，属性 IsKeyboardFocused 就是 True，引发第二个触发器，将 TextBox 的 Background 属性设置为 Yellow。

```

<Window x:Class="TriggerSample.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="TriggerSample" Height="200" Width="400"
    >
    <Window.Resources>
        <Style x:Key="TextBoxStyle" TargetType="{x:Type TextBox}">
            <Setter Property="Background" Value="LightBlue" />
            <Setter Property="FontSize" Value="17" />
            <Style.Triggers>
                <Trigger Property="IsMouseOver" Value="True">
                    <Setter Property="Background" Value="Red" />
                    <Setter Property="FontSize" Value="22" />
                </Trigger>
                <Trigger Property="IsKeyboardFocused" Value="True">
                    <Setter Property="Background" Value="Yellow" />
                    <Setter Property="FontSize" Value="22" />
                </Trigger>
            </Style.Triggers>
        </Style>
    </Window.Resources>

```



```
        </Style.Triggers>
    </Style>
</Window.Resources>
<Canvas>
    <TextBox Canvas.Top="80" Canvas.Left="30" Width="300"
        Style="{StaticResource TextBoxStyle}" />
    <TextBox Canvas.Top="120" Canvas.Left="30" Width="300"
        Style="{StaticResource TextBoxStyle}" />
</Canvas>
</Window>
```

当引发触发器的原因不再有效时，就不必将属性值重置为初始值。例如不必定义 `IsMouseOver = true` 和 `IsMouseOver=false` 的触发器。只要引发触发器的原因不再有效，触发器进行的修改就会自动重置为初始值。

图 34-21 显示了触发器示例程序，其中，第一个文本框获得了焦点，第二个文本框使用样式的默认值指定背景和字号。

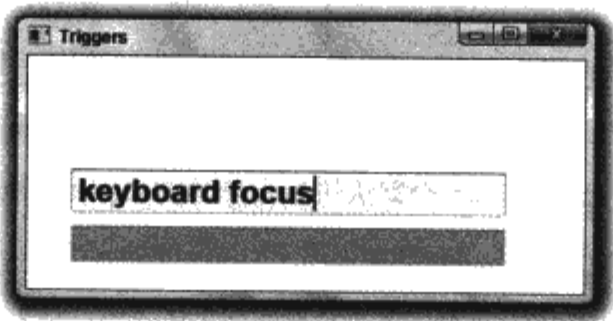


图 34-21

**警告：**

使用属性触发器，很容易改变控件的外观、字体、颜色、不透明度等。在鼠标滑过控件时，键盘设置焦点时，……都不需要编写任何代码。

Trigger 类定义了表 34-9 中的属性，以指定触发器的操作。

表 34-9

Trigger 属性	说 明
Property Value	使用属性触发器，Property 和 Value 属性用于指定触发器的引发时间，例如，Property = "IsMouseOver", Value = "True"
Setters	一旦引发触发器，就可以使用 Setters 定义一个 Setter 元素集合，来改变属性值。Setter 类定义了属性 Property、TargetName 和 Value，以修改对象属性
EnterActions ExitActions	除了定义 Setters 之外，还可以定义 EnterActions 和 ExitActions。使用这两个属性，可以定义一个 TriggerAction 元素集合。EnterActions 在启动触发器时引发(此时属性触发器应用了 Property/Value 组合)。ExitActions 在触发器结束之前引发(此时还没有应用 Property/Value 组合)。用这些操作指定的触发器操作派生自基类 TriggerAction，例如 SoundPlayerAction 和 BeginStoryboard。使用 SoundPlayerAction 可以开始播放声音，BeginStoryboard 用于动画，详见本章后面的内容。

提示:

属性触发器只是 WPF 中的一种触发器。另一种触发器是事件触发器。事件触发器在后面与动画一起论述。

5. 模板

本章前面介绍过，Button 控件可以包含任何内容，例如简单的文本，还可以给按钮添加一个 Canvas 元素，Canvas 元素可以包含图形。也可以给按钮添加 Grid、视频。按钮还可以完成更多的操作。

控件的外观、操作方式及其功能在 WPF 中是完全分离的。按钮有默认的外观，但可以用模板完全定制其外观。

表 34-10

模 板 类 型	说 明
ControlTemplate	使用 ControlTemplate 可以指定控件的可视化结构，重新设计其外观
ItemsPanelTemplate	对于 ItemsControl，可以赋予一个 ItemsPanelTemplate，以指定其项的布局。每个 ItemsControl 都有一个默认的 ItemsPanelTemplate。MenuItem 使用 WrapPanel，StatusBar 使用 DockPanel，ListBox 使用 VirtualizingStackPanel
DataTemplate	DataTemplates 非常适用于对象的图形化表示。给列表框指定样式，所以列表框中的项根据 ToString()方法的输出来显示。应用 DataTemplate，可以重写其操作，定义项的定制显示
HierarchicalData Template	HierarchicalDataTemplate 用于安排对象的树形结构。这个控件支持 HeaderedItemsControls，例如 TreeViewItem 和 MenuItem

下面的示例有几个按钮，然后逐步定制列表框，这样就可以立即看到改变的结果。首先，用两个非常简单的按钮作为开始，第一个按钮根本没有应用样式，第二个按钮引用了样式 ButtonStyle1，来改变 Background 和 FoneSize。图 34-22 显示了第一个结果。

```
<Window x:Class="TemplateSample.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Template Sample" Height="300" Width="300">
  <Window.Resources>
    <Style x:Key="ButtonStyle1" TargetType="{x:Type Button}">
      <Setter Property="Background" Value="Yellow" />
      <Setter Property="FontSize" Value="18" />
    </Style>
  </Window.Resources>
  <StackPanel>
    <Button Name="button1" Height="50" Width="150">Default Button</Button>
    <Button Name="button2" Height="50" Width="150"
      Style="{StaticResource ButtonStyle1}">Styled Button
    </Button>
  </StackPanel>
</Window>
```

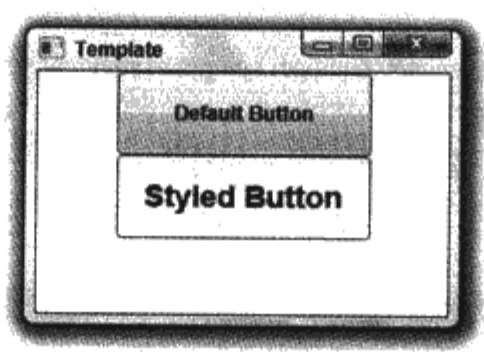


图 34-22

现在，给资源添加新样式 `ButtonStyle2`。这个样式再次将 `TargetType` 设置为 `Button` 类型。`Setter` 现在指定了 `Template` 属性。而指定 `Template` 属性，可以完全改变按钮的外观。`Template` 属性的值通过 `ControlTemplate` 元素来定义。`ControlTemplate` 定义了控件的内容，允许访问控件的内容，如后面所述。这里，`ControlTemplate` 定义了一个包含两行的 `Grid`。该行根据具体情况指定大小，其中第一行的高度是第二行的两倍。接着定义了两个 `Rectangle` 元素。第一个矩形横跨两行，并将 `Stroke` 属性设置为 `Green`，获得绿色的边框，再设置 `RadiusX` 和 `RadiusY` 值，得到圆角矩形。第二个矩形位于第一个矩形内部，其 `Fill` 属性设置为线性渐变笔刷。`button3` 的内容设置为 `Template Button`，并引用样式 `ButtonStyle2`。图 34-23 显示了采用新样式的 `button3`，但内容没有显示。

```
<Window x:Class="TemplateSample.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Template Sample" Height="300" Width="300"
>
<Window.Resources>
<!-- other styles -->
<Style x:Key="ButtonStyle2" TargetType="{x:Type Button}">
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate>
        <Grid>
          <Grid.RowDefinitions>
            <RowDefinition Height="2*" />
            <RowDefinition Height="*" />
          </Grid.RowDefinitions>
          <Rectangle Grid.RowSpan="2" RadiusX="4" RadiusY="8" Stroke="Green" />
          <Rectangle RadiusX="4" RadiusY="8" Margin="2">
            <Rectangle.Fill>
              <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                <GradientStop Offset="0" Color="LightBlue" />
                <GradientStop Offset="0.5" Color="#ffff" />
                <GradientStop Offset="1" Color="#6faa" />
              </LinearGradientBrush>
            </Rectangle.Fill>
          </Rectangle>
        </Grid>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
</Style>
</Window.Resources>
```

```

<StackPanel>
<!-- other buttons -->
  <Button Name="button3" Background="Yellow" Height="100" Width="220" FontSize="24"
    Style="{StaticResource ButtonStyle2}">Template Button
  </Button>
</StackPanel>
</Window>

```

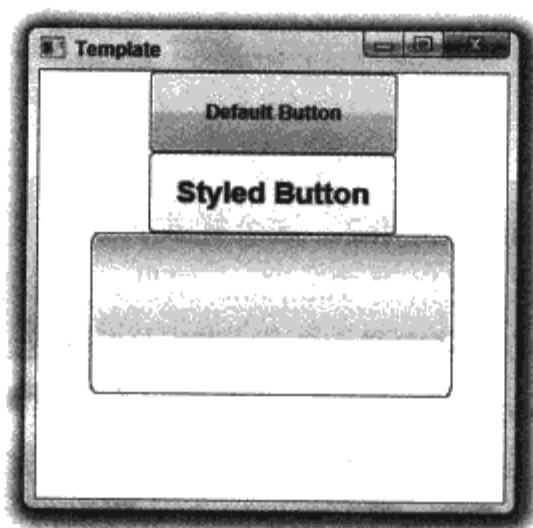


图 34-23

按钮现在的外观完全不同，但按钮的内容未在图 34-23 中显示出来。必须扩展前面创建的模板。模板中的第一个矩形现在把其 Fill 属性设置为 {TemplateBinding Background}。标记扩展 TemplateBinding 允许控件模板使用模板控件中的内容。这里，矩形用按钮定义的背景来填充。button3 定义了黄色背景，所以这个背景与控件模板中第二个矩形的背景合并起来。定义了第二个矩形后，使用 ContentPresenter 元素。该元素从模板控件中提取内容，根据定义放置这些内容，这里是放在两个行上，因为 Grid.RowSpan 设置为 2。如果定义了一个 ContentPresenter 元素，就必须用 ControlTemplate 设置 TargetType。使用 TemplateBinding 标记扩展，将 HorizontalAlignment、VerticalAlignment 和 Margin 属性设置为按钮定义的值，以指定内容在按钮中的位置。使用 ControlTemplate 还可以在资源中定义前面的触发器。图 34-24 显示了按钮的新结果，其中包括内容和合并了模板的背景。

```

<Style x:Key="ButtonStyle2" TargetType="{x:Type Button}">
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="{x:Type Button}">
        <Grid>
          <Grid.RowDefinitions>
            <RowDefinition Height="2*" />
            <RowDefinition Height="*" />
          </Grid.RowDefinitions>
          <Rectangle Grid.RowSpan="2" RadiusX="4" RadiusY="8" Stroke="Green"
            Fill="{TemplateBinding Background}" />
          <Rectangle RadiusX="4" RadiusY="8" Margin="2">
            <Rectangle.Fill>
              <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                <GradientStop Offset="0" Color="LightBlue" />
                <GradientStop Offset="0.5" Color="#afff" />
                <GradientStop Offset="1" Color="#6faa" />
              </LinearGradientBrush>
            </Rectangle.Fill>
          </Rectangle>
        </Grid>
      </ControlTemplate>
    </Setter.Value>
  </Setter>

```

```

</Rectangle>
<ContentPresenter Grid.RowSpan="2"
    HorizontalAlignment="{TemplateBinding HorizontalContentAlignment}"
    VerticalAlignment="{TemplateBinding VerticalContentAlignment}"
    Margin="{TemplateBinding Padding}" />
</Grid>
<ControlTemplate.Triggers>
    <Trigger Property="IsMouseOver" Value="True">
        <Setter Property="Foreground" Value="Aqua" />
    </Trigger>
    <Trigger Property="IsPressed" Value="True">
        <Setter Property="Foreground" Value="Black" />
    </Trigger>
</ControlTemplate.Triggers>
</ControlTemplate>
</Setter.Value>
</Setter>
</Style>
</Window.Resources>

```



图 34-24

下面使用透明特性创建一个更有趣的按钮。样式 GelButton 设置了属性 Background、Height、Foreground、Margin 和 Template。模板是这个样式中最有趣的部分。该模板将 Grid 指定为一行一列。

在这个单元格中，有一个名为 GelBackground 的矩形。这个矩形是圆角矩形，因为指定了 RadiusX 和 RadiusY 值，该矩形的边框非常细，且采用线性渐变色，因为 StrokeThickness 设置为 0.35，还为这类笔触设置了线性渐变笔刷。

第二个矩形 GelShine 是一个高度为 15 像素的小矩形，因为设置了 Margin，所以在第一个矩形中可见。其笔触是透明的，所以该矩形没有边框。这个矩形只使用一个线性渐变填充笔刷，其颜色从部分透明的浅色变为完全透明。这会使该矩形具有微微发亮的效果。

在这两个矩形之后，再创建一个 ContentPresenter 元素，它为内容指定对齐方式，并从按钮中提取要显示的内容。

这样一个样式化的按钮在屏幕上看起来很漂亮。但是，如果用鼠标单击它，或使鼠标滑过该按钮，它不会有任何动作。对于模板样式的按钮，必须给它指定触发器，使它在响应鼠标单击时显示不同的外观。属性触发器 IsMouseOver 为 Rectangle.Fill 属性定义一个新值，用辐射渐变笔刷给矩形填充另一种颜色。有新填充效果的矩形通过 TargetName 来引用。属性触发器



IsPressed 与前面的触发器非常类似，但用另一种颜色的辐射渐变笔刷填充矩形。图 34-25 显示了一个引用样式 GelButton 的按钮。图 34-26 显示了鼠标滑过时该按钮的外观，从中可以看出辐射渐变笔刷的效果。

```
<Style x:Key="GelButton" TargetType="{x:Type Button}">
  <Setter Property="Background" Value="Black" />
  <Setter Property="Height" Value="40" />
  <Setter Property="Foreground" Value="White" />
  <Setter Property="Margin" Value="3" />
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="{x:Type Button}">
        <Grid>

          <Rectangle Name="GelBackground" RadiusX="9" RadiusY="9"
            Fill="{TemplateBinding Background}" StrokeThickness="0.35">
            <Rectangle.Stroke>
              <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                <GradientStop Offset="0" Color="White" />
                <GradientStop Offset="1" Color="#666666" />
              </LinearGradientBrush>
            </Rectangle.Stroke>
          </Rectangle>

          <Rectangle Name="GelShine" Margin="2,2,2,0" VerticalAlignment="Top"
            RadiusX="6" RadiusY="6" Stroke="Transparent" Height="15px">
            <Rectangle.Fill>
              <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                <GradientStop Offset="0" Color="#ccffffff" />
                <GradientStop Offset="1" Color="Transparent" />
              </LinearGradientBrush>
            </Rectangle.Fill>
          </Rectangle>

          <ContentPresenter Name="GelButtonContent" VerticalAlignment="Center"
            HorizontalAlignment="Center"
            Content="{TemplateBinding Content}" />
        </Grid>

        <ControlTemplate.Triggers>
          <Trigger Property="IsMouseOver" Value="True">
            <Setter Property="Rectangle.Fill" TargetName="GelBackground">
              <Setter.Value>
                <RadialGradientBrush>
                  <GradientStop Offset="0" Color="Lime" />
                  <GradientStop Offset="1" Color="DarkGreen" />
                </RadialGradientBrush>
              </Setter.Value>
            </Setter>
            <Setter Property="Foreground" Value="Black" />
          </Trigger>
          <Trigger Property="IsPressed" Value="True">
            <Setter Property="Rectangle.Fill" TargetName="GelBackground">
              <Setter.Value>
                <RadialGradientBrush>
                  <GradientStop Offset="0" Color="#ffcc00" />
                  <GradientStop Offset="1" Color="#cc9900" />
                </RadialGradientBrush>
              </Setter.Value>
            </Setter>
          </Trigger>
        </ControlTemplate.Triggers>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
</Style>
```

```

        </ControlTemplate.Triggers>
    </ControlTemplate>
</Setter.Value>
</Setter>
</Style>

```

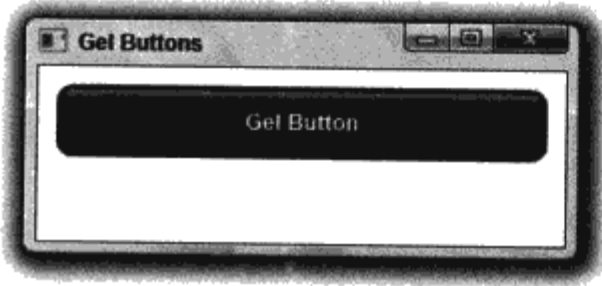


图 34-25

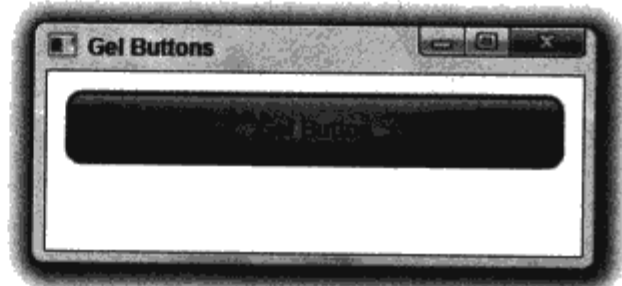


图 34-26

除了使用矩形按钮之外，还可以创建椭圆按钮。下面的例子将探讨如何使一种样式基于另一种样式。

用 `Style` 元素设置 `BasedOn` 属性，就可以使样式 `RoundedGelButton` 基于样式 `GelButton`。如果一种样式基于另一种样式，新样式就会获得基样式的所有设置，除非重新定义了设置。例如，`RoundedGelButton` 样式从 `GelButton` 样式中获得了 `Foreground` 和 `Margin` 设置，因为这些设置没有重新定义。如果改变了基样式中的一个设置，基于该样式的所有样式都会自动获得这个新设置值。

`Height` 和 `Template` 属性在新样式中重新定义了。其中，模板定义了两个 `Ellipse` 元素来替代矩形。外部椭圆的 `GelBackground` 定义了一个边框有渐变效果的黑色椭圆。第二个椭圆较小，其顶边距第一个椭圆 5 像素，底边距第一个椭圆 50 像素。这个椭圆也使用线性渐变效果，其颜色从一种浅色变为透明色，并指定了“发光”效果。还要为 `IsMouseOver` 和 `IsPressed` 指定触发器，改变第一个椭圆的 `Fill` 属性值。

在图 34-27 中，显示了基于 `RoundedGelButton` 样式的新按钮，它仍旧是一个按钮。

```

<Style x:Key="RoundedGelButton" BasedOn="{StaticResource GelButton}"
    TargetType="Button">
    <Setter Property="Width" Value="100" />
    <Setter Property="Height" Value="100" />
    <Setter Property="Grid.Row" Value="2" />
    <Setter Property="Template">
        <Setter.Value>
            <ControlTemplate TargetType="{x:Type Button}">
                <Grid>

                    <Ellipse Name="GelBackground" StrokeThickness="0.5" Fill="Black">
                        <Ellipse.Stroke>
                            <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                                <GradientStop Offset="0" Color="#ff7e7e7e" />
                                <GradientStop Offset="1" Color="Black" />
                            </LinearGradientBrush>
                        </Ellipse.Stroke>
                    </Ellipse>

                    <Ellipse Margin="15,5,15,50">
                        <Ellipse.Fill>
                            <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                                <GradientStop Offset="0" Color="#aaffffff" />

```

```

        <GradientStop Offset="1" Color="Transparent" />
    </LinearGradientBrush>
</Ellipse.Fill>
</Ellipse>

    <ContentPresenter Name="GelButtonContent" VerticalAlignment="Center"
        HorizontalAlignment="Center" Content="{TemplateBinding Content}"
    />
</Grid>

<ControlTemplate.Triggers>
    <Trigger Property="IsMouseOver" Value="True">
        <Setter Property="Rectangle.Fill" TargetName="GelBackground">
            <Setter.Value>
                <RadialGradientBrush>
                    <GradientStop Offset="0" Color="Lime" />
                    <GradientStop Offset="1" Color="DarkGreen" />
                </RadialGradientBrush>
            </Setter.Value>
        </Setter>
        <Setter Property="Foreground" Value="Black" />
    </Trigger>

    <Trigger Property="IsPressed" Value="True">
        <Setter Property="Rectangle.Fill" TargetName="GelBackground">
            <Setter.Value>
                <RadialGradientBrush>
                    <GradientStop Offset="0" Color="#ffcc00" />
                    <GradientStop Offset="1" Color="#cc9900" />
                </RadialGradientBrush>
            </Setter.Value>
        </Setter>
        <Setter Property="Foreground" Value="Black" />
    </Trigger>
</ControlTemplate.Triggers>
</ControlTemplate>

</Setter.Value>
</Setter>
</Style>

```



图 34-27

### 34.8.3 创建列表框的样式

创建按钮或标签的样式是一个简单的任务。如何改变包含一组元素的父元素的样式，例如列表框？列表框也有操作方式和外观。列表框可以显示一组元素，用户可以从列表中选



择一个或多个元素。至于操作，ListBox 类定义了方法、属性和事件。列表框的外观与其操作是分开的。列表框元素有一个默认的外观，但可以通过创建模板，改变这个外观。

为了显示列表中的某些数据项，下面创建 Country 类，来表示名称和图像路径标志。Country 类定义了 Name 和 ImagePath 属性，用重写的 ToString() 方法表示默认的字符串：

```
public class Country
{
    public Country(string name)
        : this(name, null)
    {
    }

    public Country(string name, string imagePath)
    {
        this.name = name;
        this.imagePath = imagePath;
    }

    public string Name { get; set; }
    public string ImagePath { get; set; }
    public override string ToString()
    {
        return name;
    }
}
```

静态类 Countries 返回几个要显示的国家：

```
public static class Countries
{
    public static Country[] GetCountries()
    {
        List<Country> countries = new List<Country>();
        countries.Add(new Country("Austria", "Images/Austria.bmp"));
        countries.Add(new Country("Germany", "Images/Germany.bmp"));
        countries.Add(new Country("Norway", "Images/Norway.bmp"));
        countries.Add(new Country("USA", "Images/USA.bmp"));

        return countries.ToArray();
    }
}
```

在后台编码文件中，Window1 类的构造函数将 Window1 实例的 DataContext 属性设置为要从 Countries.GetCountries() 方法中返回的国家列表。DataContext 属性是数据绑定的一个特性，详见下一章。

```
public partial class Window1 : System.Windows.Window
{
    public Window1()
    {
        InitializeComponent();
        this.DataContext = Countries.GetCountries();
    }
}
```

在 XAML 代码中，定义了 countryList1 列表框。countryList1 没有使用样式，只使用了列表框元素的默认外观。属性 ItemsSource 设置为 Binding 标记扩展，它由数据绑定使用，在后台代

码中，数据绑定用于一个 `Country` 对象数组。图 34-28 显示了列表框的默认外观。在默认情况下，只在一个简单的列表中显示 `ToString()` 方法返回的国家名称。

```
<Window x:Class="ListboxStyling.Window1"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Listbox Styling" Height="300" Width="300">
    <StackPanel>
        <ListBox Name="countryList1" ItemsSource="{Binding}" />
    </StackPanel>
</Window>
```

`Country` 对象有名称和标志。当然，还可以在列表框中显示这两个值。为此，必须定义一个模板。

列表框元素包含 `ListBoxItem` 元素。使用 `ItemTemplate` 可以定义列表项的内容。样式 `listBoxStyle1` 定义了一个 `ItemTemplate`，其值为 `DataTemplate`。`DataTemplate` 用于将数据绑定到元素上。`Binding` 标记扩展可以用于 `DataTemplate` 元素。

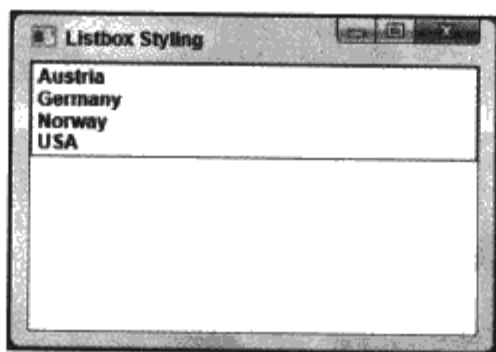


图 34-28

`DataTemplate` 包含一个带三列的栅格。第一列包含字符串“Country:”，第二列包含国家的名称，第三列包含该国家的标志。因为国家名称的长度是不同的，但看起来每个国家名称应有相同的大小，所以给第二列定义 `SharedSizeGroup` 属性。只有这一列使用共享大小的信息，因为设置了属性 `Grid.IsSharedSizeScope`。

行和列定义之后，就可以看到两个 `TextBlock` 元素。第一个 `TextBlock` 元素包含文本“Country:”，第二个 `TextBlock` 元素绑定到 `Country` 类定义的 `Name` 属性上。

第三列的内容是一个包含栅格的 `Border` 元素。栅格包含一个带线性边框的矩形和一个绑定到 `Country` 类中 `ImagePath` 属性上的 `Image` 元素。图 34-29 显示了列表框中的国家，其效果与前面完全不同。

```
<Window.Resources>
    <Style x:Key="listBoxStyle1" TargetType="{x:Type ListBox}" >
        <Setter Property="ItemTemplate">
            <Setter.Value>
                <DataTemplate>
                    <Grid>
                        <Grid.ColumnDefinitions>
                            <ColumnDefinition Width="Auto" />
                            <ColumnDefinition Width="*" SharedSizeGroup="MiddleColumn" />
                            <ColumnDefinition Width="Auto" />
                        </Grid.ColumnDefinitions>
```



```

        <Grid.RowDefinitions>
            <RowDefinition Height="60" />
        </Grid.RowDefinitions>

        <TextBlock FontSize="16" VerticalAlignment="Center" Margin="5"
            FontStyle="Italic" Grid.Column="0">Country:</TextBlock>

        <TextBlock FontSize="16" VerticalAlignment="Center" Margin="5"
            Text="{Binding Name}" FontWeight="Bold" Grid.Column="1" />

        <Border Margin="4,0" Grid.Column="2" BorderThickness="2" CornerRadius="4">
            <Border.BorderBrush>
                <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                    <GradientStop Offset="0" Color="#aaa" />
                    <GradientStop Offset="1" Color="#222" />
                </LinearGradientBrush>
            </Border.BorderBrush>

            <Grid>
                <Rectangle>
                    <Rectangle.Fill>
                        <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                            <GradientStop Offset="0" Color="#444" />
                            <GradientStop Offset="1" Color="#fff" />
                        </LinearGradientBrush>
                    </Rectangle.Fill>

                    </Rectangle>

                    <Image Width="48" Margin="2,2,2,1" Source="{Binding ImagePath}" />
                </Grid>
            </Border>

        </Grid>
    </DataTemplate>
</Setter.Value>
</Setter>
<Setter Property="Grid.IsSharedSizeScope" Value="True" />

</Style>
</Window.Resources>

```

列表框中的数据项不一定是垂直排列，还可以给这个功能提供另一种视图。下一个样式 `listBoxStyle2` 定义了一个模板，使列表项水平显示，且带一个滚动条。

在前面的示例中，只创建了一个 `ItemTemplate`，来定义列表项在默认列表框中的外观。现在创建一个模板，定义另一个列表框。该模板包含一个 `ControlTemplate` 元素，它定义了列表框的元素。`ControlTemplate` 元素现在是一个包含 `StackPanel` 的 `ScrollViewer`——带滚动条的视图。列表项现在应水平排列，所以 `StackPanel` 的 `Orientation` 设置为 `Horizontal`。`StackPanel` 还包含用 `ItemTemplate` 定义的数据项，所以 `StackPanel` 元素的 `IsItemsHost` 设置为 `true`。`IsItemsHost` 属性可以用于包含一系列数据项的所有 `Panel` 元素。

`ItemTemplate` 定义了 `StackPanel` 中数据项的外观。在其中创建一个包含两行的栅格，第一行包含绑定到 `ImagePath` 属性上的 `Image` 元素，第二行包含绑定到 `Name` 属性上的 `TextBlock`。

图 34-30 显示了使用 `listBoxStyle2` 样式的列表框，当视图太小，不能显示列表中的所有项



图 34-29

时,滚动条会自动显示出来。

```
<Style x:Key="listBoxStyle2" TargetType="{x:Type ListBox}">
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="{x:Type ListBox}">
        <ScrollViewer HorizontalScrollBarVisibility="Auto">
          <StackPanel Name="StackPanel1" IsItemsHost="True"
            Orientation="Horizontal" />
        </ScrollViewer>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
  <Setter Property="VerticalAlignment" Value="Center" />
  <Setter Property="ItemTemplate">
    <Setter.Value>
      <DataTemplate>
        <Grid>
          <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
          </Grid.ColumnDefinitions>
          <Grid.RowDefinitions>
            <RowDefinition Height="60" />
            <RowDefinition Height="30" />
          </Grid.RowDefinitions>
          <Image Grid.Row="0" Width="48" Margin="2,2,2,1"
            Source="{Binding ImagePath}" />
          <TextBlock Grid.Row="1" FontSize="14" HorizontalAlignment="Center"
            Margin="5" Text="{Binding Name}" FontWeight="Bold" />
        </Grid>
      </DataTemplate>
    </Setter.Value>
  </Setter>
</Style>
```

显然,将控件的外观与其操作分开是很有好处的。也许用户有许多方式,能很好地显示列表中的数据项,使之满足应用程序的要求。也许用户只想在窗口中显示一定数量的数据项,先水平排列,一行放不下时,就继续排在下一行。此时就可以使用 `WrapPanel`。当然,可以给列表框的模板定义一个 `WrapPanel`,如 `listBoxStyle3` 所示。图 34-31 显示了使用 `WrapPanel` 的结果:



图 34-30

```
<Style x:Key="listBoxStyle3" TargetType="{x:Type ListBox}">
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="{x:Type ListBox}">
        <ScrollViewer VerticalScrollBarVisibility="Auto"
          HorizontalScrollBarVisibility="Disabled">
          <WrapPanel IsItemsHost="True" />
        </ScrollViewer>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
</Style>
```

```

</Setter>
<Setter Property="ItemTemplate">
  <Setter.Value>
    <DataTemplate>
      <Grid>
        <Grid.ColumnDefinitions>
          <ColumnDefinition Width="140" />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
          <RowDefinition Height="60" />
          <RowDefinition Height="30" />
        </Grid.RowDefinitions>

        <Image Grid.Row="0" Width="48" Margin="2,2,2,1"
          Source="{Binding ImagePath}" />

        <TextBlock Grid.Row="1" FontSize="14" HorizontalAlignment="Center"
          Margin="5" Text="{Binding Name}" />

      </Grid>
    </DataTemplate>
  </Setter.Value>
</Setter>
</Style>

```

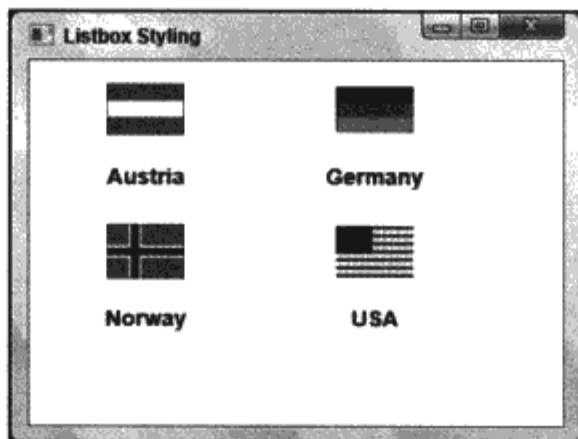


图 34-31

提示:

下一章介绍了 DataTemplate 和数据绑定功能。

## 34.9 小结

本章介绍了 WPF 的许多特性。WPF 便于分开开发人员和设计人员的工作。Microsoft Expression Blend 和 Visual Studio 都可以使用 XAML 代码。XAML 代码能很好地分开 UI 及其功能。所有的 UI 特性都可以使用 XAML 创建，其功能用后台代码创建。

我们还探讨了基于矢量图形的许多控件和容器。WPF 元素是矢量图形，所以可以缩放、倾斜和旋转。内容控件的内容非常灵活，所以事件处理机制基于起泡和通道事件。

可以使用不同类型的笔刷绘制背景和前景元素，例如可以使用纯色笔刷、线性渐变和辐射渐变笔刷、可视化笔刷完成反射功能或显示视频。

样式和模板可以定制控件的外观。触发器可以动态改变 WPF 控件的属性。连续改变 WPF 控件的属性值，就可以制作出动画。

下一章继续介绍 WPF，主要探讨动画、3D、数据绑定和其他几个特性。

# 第 35 章

## 高级 WPF

上一章介绍了 WPF 的一些核心功能，本章继续用 WPF 编程，介绍创建完整应用程序的一些重要方面，例如数据绑定和命令处理，还要探讨动画和 3D 编程。

本章的主要内容如下：

- 数据绑定
- 命令
- 动画
- 3D
- Windows Forms 集成

### 35.1 数据绑定

上一章在给列表框设置样式时，介绍了数据绑定的几个特性。当然，数据绑定还有许多特性。WPF 数据绑定与 Windows Forms 数据绑定相比，进了一大步。本节介绍 WPF 中的数据绑定，讨论如下主题：

- 概述
- 用 XAML 绑定
- 简单的对象绑定
- 对象数据提供程序
- 列表绑定
- 绑定到 XML 上

#### 35.1.1 概述

在 WPF 数据绑定中，目标可以是 WPF 元素的任意依赖属性，CLR 对象的每个属性都可以是绑定源。WPF 元素实现为 .NET 类，所以每个 WPF 元素都可以用作绑定源。图 35-1 显示了绑定源和绑定目标之间的连接。Binding 对象定义了该连接。

Binding 对象支持源与目标之间的几种绑定模式。绑定可以是单向的，即源信息传送给目标，但如果用户在用户界面上修改了该信息，源不会更新。要更新源，需要双向绑定。

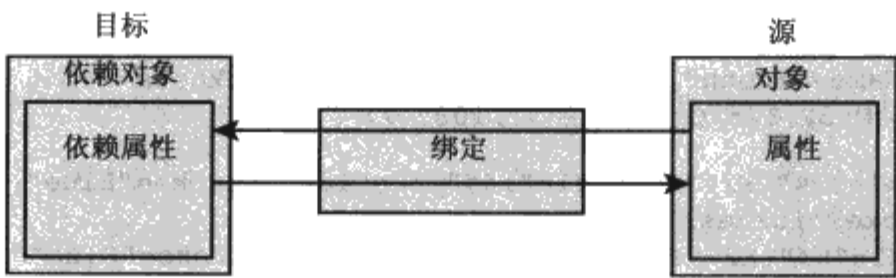


图 35-1

表 35-1 列出了绑定模式及其要求。

表 35-1

绑定模式	说明
一次性	绑定从源指向目标，且仅在应用程序启动时，或数据内容改变时绑定一次。通过这种模式可以获得数据的快照
单向	绑定从源指向目标。可以用于只读数据，因为它不能在用户界面上修改数据。要更新用户界面，源必须实现接口 <code>INotifyPropertyChanged</code>
双向	在双向绑定中，用户可以在 UI 上修改数据。绑定是双向的——从源指向目标，从目标指向源。源对象需要实现读写属性，才能把改动的内容从 UI 更新到源对象上
指向源的单向	采用这种绑定模式，如果目标属性改变了，源对象也会更新

35.1.2 用 XAML 绑定

WPF 元素不仅是数据绑定的目标，还是绑定的源。可以把一个 WPF 元素的源属性绑定到另一个 WPF 元素的目标属性上。

下面的例子使用前面创建的笑脸，它是用 WPF 图形创建的，然后将其绑定到一个滑块上，以便在窗口中移动它。滑块是源元素，其名称是 `slider`。属性 `Value` 给出了滑块的位置值。数据绑定的目标是内部的 `Canvas` 元素。这个 `Canvas` 元素的名称是 `FunnyFace`，包含了绘制笑脸所需的所有图形。该 `Canvas` 元素包含在外部的 `Canvas` 元素中，所以可以设置关联的属性，在外部的 `Canvas` 元素中定位内部的 `Canvas` 元素。关联属性 `Canvas.Left` 设置为 `Binding` 标记扩展。在 `Binding` 标记扩展中，`ElementName` 设置为 `slider`，以引用 WPF 滑块元素，`Path` 设置为 `Value`，从 `Value` 属性中获取值。

```
<Window x:Class="DataBindingSample.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="DataBindingSample" Height="345" Width="310">
  <StackPanel>
    <Canvas Height="210" Width="280">
      <Canvas Canvas.Top="0"
        Canvas.Left="{Binding Path=Value, ElementName=slider}"
        Name="FunnyFace" Height="210" Width="230">
        <Ellipse Canvas.Left="20" Canvas.Top="50" Width="100" Height="100"
          Stroke="Blue" StrokeThickness="4" Fill="Yellow" />
        <Ellipse Canvas.Left="40" Canvas.Top="65" Width="25" Height="25"
          Stroke="Blue" StrokeThickness="3" Fill="White" />
      </Canvas>
    </Canvas>
  </StackPanel>
</Window>
```



```

<Ellipse Canvas.Left="50" Canvas.Top="75" Width="5" Height="5"
    Fill="Black" />
<Path Name="mouth" Stroke="Blue" StrokeThickness="4"
    Data="M 32,125 Q 65,122 72,108" />

<Line X1="94" X2="102" Y1="144" Y2="166" Stroke="Blue"
    StrokeThickness="4" />
<Line X1="84" X2="103" Y1="169" Y2="166" Stroke="Blue"
    StrokeThickness="4" />

<Line X1="62" X2="52" Y1="146" Y2="168" Stroke="Blue"
    StrokeThickness="4" />
<Line X1="38" X2="53" Y1="160" Y2="168" Stroke="Blue"
    StrokeThickness="4" />
</Canvas>
</Canvas>

<Slider Name="slider" Orientation="Horizontal" Value="10" Maximum="100" />

</StackPanel>
</Window>

```

运行应用程序时，可以移动滑块，笑脸就会移动，如图 35-2 和 35-3 所示。



图 35-2



图 35-3

除了用 XAML 代码定义绑定信息之外，如这个例子使用 Binding 元数据扩展来定义，还可以使用后台代码。绑定的 XAML 版本如下：

```

<Canvas Canvas.Top="0"
    Canvas.Left="{Binding Path=Value, ElementName=slider}"
    Name="FunnyFace" Height="210" Width="230">

```

在后台代码中，必须创建一个新的 Binding 对象，设置 Path 和 Source 属性。Source 属性必须设置为源对象，这里是 WPF 对象 slider。Path 属性设置为一个 PropertyPath 实例，它用源对象的 Value 属性名初始化。对于目标，可以调用 SetBinding() 方法来定义绑定。这里，目标是名为 FunnyFace 的 Canvas 对象。SetBinding() 方法需要两个参数，第一个参数是一个依赖属性，第二个参数是绑定对象。Canvas.Left 属性应是绑定的，这样 DependencyProperty 类型的依赖属性才能用 Canvas.LeftProperty 字段访问：

```

Binding binding = new Binding();
binding.Path = new PropertyPath("Value");
binding.Source = slider;

FunnyFace.SetBinding(Canvas.LeftProperty, binding);

```

使用 Binding 类，可以配置许多绑定选项，如表 35-2 所示。

表 35-2

Binding 类的成员	说 明
Source	使用 Source 属性，可以定义数据绑定的源对象
RelativeSource	使用 RelativeSource 属性，可以指定与目标对象相关的源对象。当错误来源于同一个控件时，它可用于显示错误消息
ElementName	如果源对象是一个 WPF 元素，就可以用 ElementName 属性指定源对象
Path	使用 Path 属性，可以指定到源对象的路径。它可以是源对象的属性，也支持子元素的索引符和属性
XPath	使用 XML 数据源时，可以定义一个 XPath 查询表达式，来获得要绑定的数据
Mode	模式定义了绑定的方向。Mode 属性是 BindingMode 类型。BindingMode 是一个枚举，其值如下：Default、OneTime、OneWay、TwoWay、OneWayToSource。默认模式依赖于目标：对于文本框，双向绑定是默认值；对于只读的标签，默认为单向；OneTime 表示数据仅从源中加载一次。OneWay 还可以将对源对象的修改更新到目标对象中。TwoWay 绑定表示，对 WPF 元素的修改可以写回源对象。OneWayToSource 表示，从不读取数据，但总是从目标对象写入源对象
Converter	使用 Converter 属性，可以指定一个转换器类，来回转换 UI 的数据。转换器类必须实现接口 IValueConverter，它定义了方法 Convert() 和 ConvertBack()。使用 ConverterParameter 属性可以给转换方法传送参数。转换器对文化很敏感，文化可以用 ConverterCultrue 属性设置
FallbackValue	使用 FallbackValue 属性，可以定义一个在绑定没有返回值时使用的默认值
ValidationRules	使用 ValidationRules 属性，可以定义一个 ValidationRule 对象集合，在用 WPF 目标元素更新源对象之前检查该集合。ExceptionValidationRule 类派生自 ValidationRule 类，负责检查异常

35.1.3 简单对象的绑定

要绑定 CLR 对象，只需使用.NET 类定义属性，如下面的例子就使用类 Book 定义了属性 Title、Publisher、Isbn 和 Authors:

```
public class Book
{
    public Book(string title, string publisher, string isbn,
        params string[] authors)
    {
        this.title = title;
        this.publisher = publisher;
        this.isbn = isbn;
        foreach (string author in authors)
        {
            this.authors.Add(author);
        }
    }

    public Book()
```

```

        : this("unknown", "unknown", "unknown")
    {
    }

    public string Title{ get; set; }
    public string Publisher{ get; set; }
    public string Isbn{ get; set; }

    public override string ToString()
    {
        return title;
    }

    private readonly List<string> authors = new List<string>();
    public string[] Authors
    {
        get { return authors.ToArray(); }
    }
}

```

在用户界面上，定义了几个标签和文本框控件，以显示图书信息。使用 Binding 标记扩展，将文本框控件绑定到 Book 类的属性上。在 Binding 标记扩展中，仅定义了 Path 属性，将它绑定到 Book 类的属性上。不需要定义源对象，因为源对象是通过指定 DataContext 来定义的，如下面的后台代码所示。对于 TextBox 元素，模式定义为默认值，即双向绑定：

```

<Window x:Class="ObjectBindingSample.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Object Binding Sample" Height="300" Width="340">

    <Grid Name="bookGrid" Margin="5,5,5,5" >
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="30*" />
            <ColumnDefinition Width="70*" />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="50" />
            <RowDefinition Height="50" />
            <RowDefinition Height="50" />
            <RowDefinition Height="50" />
            <RowDefinition Height="50" />
        </Grid.RowDefinitions>
        <Label Grid.Column="0" Grid.Row="0">Title:</Label>
        <TextBox Margin="5" Height="30" Grid.Column="1" Grid.Row="0"
            Text="{Binding Title}" />

        <Label Grid.Column="0" Grid.Row="1">Publisher:</Label>
        <TextBox Margin="5" Height="30" Grid.Column="1" Grid.Row="1"
            Text="{Binding Publisher}" />

        <Label Grid.Column="0" Grid.Row="2">ISBN:</Label>
        <TextBox Margin="5" Height="30" Grid.Column="1" Grid.Row="2"
            Text="{Binding Isbn}" />

        <Button Margin="5" Grid.Column="1" Grid.Row="4" Click="bookButton_Click"
            Name="bookButton">Open Dialog</Button>

    </Grid>
</Window>

```

在后台代码中定义了一个新的 `Book` 对象，并将其赋予 `Grid` 控件的 `DataContext` 属性。`DataContext` 是一个依赖属性，用基类 `FrameworkElement` 定义。指定 `Grid` 控件的 `DataContext` 属性表示，`Grid` 控件中的每个元素都默认绑定到同一个数据环境上。

```
public partial class Window1 : System.Windows.Window
{
    private Book book1 = new Book();

    public Window1()
    {
        InitializeComponent();
        book1.Title = "Professional C# 2005 with .NET C# 3.0";
        book1.Publisher = "Wrox Press";
        book1.Isbn = "978-0764575341";
        bookGrid.DataContext = book1;
    }
}
```

启动应用程序，就会看到图 35-4 所示的绑定数据。

为了实现双向绑定(对 WPF 元素的修改反映到 CLR 对象上)，定义了方法 `OnOpenBookDialog()`。这个方法指定为 `bookButton` 的 `Click` 事件，如下面的 XAML 代码所示。在执行时，会弹出一个消息框，显示 `book1` 对象的当前标题和 ISBN 号。图 35-5 显示了在运行过程中修改了一个输入后消息框的结果。

```
void OnOpenBookDialog(object sender, RoutedEventArgs e)
{
    string message = book1.Title;
    string caption = book1.Isbn;
    MessageBox.Show(message, caption);
}
```

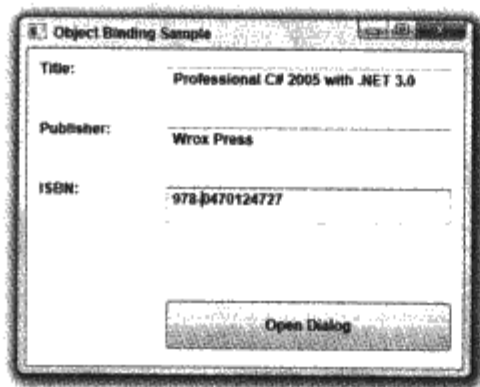


图 35-4

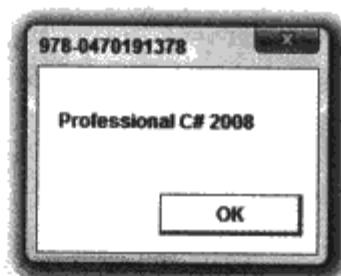


图 35-5

#### 35.1.4 对象数据提供程序

除了在后台代码中定义对象之外，还可以用 XAML 定义对象实例。为此，必须引用在 XML 根元素的命名空间中声明的命名空间。XML 属性 `xmlns:src="clr-namespace:Wrox.ProCsharp.WPF"` 将 .NET 命名空间 `Wrox.ProCSharp.WPF` 赋予 XML 命名空间别名 `src`。

现在，在 `Window` 资源中用 `Book` 元素定义 `Book` 类的一个对象。给 XML 属性 `Title` 和 `Publisher` 赋值，就设置了类 `Book` 的属性值。`x:Key="theBook"` 定义了资源的标识符，以便引用 `book` 对象。在 `TextBox` 元素中，`Source` 是用 `Binding` 标记扩展定义的，以引用 `theBook` 资源。

**提示:**

可以合并标记扩展。在下面的例子中, 引用图书资源的 StaticResource 标记扩展包含在 Binding 标记扩展中。

```
<Window x:Class="ObjectBindingSample.Window1"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:src="clr-namespace:Wrox.ProCSharp.WPF"
        Title="Object Binding Sample" Height="300" Width="340">
  <Window.Resources>
    <src:Book x:Key="theBook" Title="Professional C# 2008"
              Publisher="Wrox Press" />
  </Window.Resources>

  <!-- ... -->
  <TextBox Margin="5" Height="30" Grid.Column="1" Grid.Row="0"
            Text="{Binding Source={StaticResource theBook}, Path=Title}" />
  <!-- ... -->
```

**提示:**

如果要引用的 .NET 命名空间在另一个程序集中, 就必须把该程序集添加到 XML 声明中。

```
xmlns:system="clr-namespace:System;assembly=mscorlib".
```

除了直接在 XAML 代码中定义对象实例外, 还可以定义一个对象数据提供程序, 来引用类, 调用方法。为了使用 ObjectDataProvider, 最好创建一个返回要显示的对象的工厂类, 如下面的 BookFactory 类所示:

```
public class BookFactory
{
    private List<Book> books = new List<Book>();

    public BookFactory()
    {
        books.Add(new Book("Professional C# 2008",
                           "Wrox Press", "978-0470191378"));
    }

    public Book GetTheBook()
    {
        return books[0];
    }
}
```

ObjectDataProvider 元素可以在资源段中定义。XML 属性 ObjectType 定义了类的名称, MethodName 指定了要调用以获得 book 对象的方法名:

```
<Window.Resources>
  <ObjectDataProvider ObjectType="src:BookFactory" MethodName="GetTheBook"
    x:Key="theBook">
  </ObjectDataProvider>
</Window.Resources>
```

可以用 ObjectDataProvider 类指定的属性如表 35-3 所示。



表 35-3

ObjectDataProvider	说 明
ObjectType	ObjectType 属性定义了要创建的实例类型
ConstrutorParameters	使用 ConstructorParameters 集合可以在类中添加创建实例的参数
MethodName	MethodName 属性定义了由对象数据提供程序调用的方法名
MethodParameters	使用 MethodParameters 属性可以给通过 MethodName 属性定义的方法指定参数
ObjectInstance	使用 ObjectInstance 属性, 可以获取和设置由 ObjectDataProvider 类使用的对象。例如, 可以用编程方式指定已有的对象, 以便使用 ObjectDataProvider 实例化一个对象, 而不是定义 ObjectType
Data	使用 Data 属性, 可以访问用于数据绑定的底层对象。如果定义了 MethodName, 则使用 Data 属性, 可以访问从指定的方法返回的对象

35.1.5 列表绑定

绑定到列表上比绑定到简单对象上更灵活, 但这两种绑定非常类似。可以在后台代码中将完整的列表赋予 DataContext, 也可以使用 ObjectDataProvider 访问一个对象工厂, 以返回一个列表。对支持绑定到列表上的元素(如列表框), 会绑定整个列表。对于只支持绑定一个对象的元素(如文本框), 就绑定当前的数据项。

使用 BookFactory 类, 会返回一个 Book 对象列表:

```
public class BookFactory
{
    private List<Book> books = new List<Book>();

    public BookFactory()
    {
        books.Add(new Book("Professional C# 2008", "Wrox Press",
            "978-0470191378", "Christian Nagel", "Bill Evjen",
            "Jay Glynn", "Karli Watson", "Morgan Skinner"));
        books.Add(new Book("Professional C# 2005 with .NET 3.0",
            "Wrox Press", "978-0-470-12472-7", "Christian Nagel",
            "Bill Evjen", "Jay Glynn", "Karli Watson", "Morgan Skinner"));
        books.Add(new Book("Professional C# 2005",
            "Wrox Press", "978-0-7645-7534-1", "Christian Nagel",
            "Bill Evjen", "Jay Glynn", "Karli Watson", "Morgan Skinner",
            "Allen Jones"));
        books.Add(new Book("Beginning Visual C#",
            "Wrox Press", "978-0-7645-4382-1", "Karli Watson",
            "David Espinosa", "Zach Greenvoss", "Jacob Hammer Pedersen",
            "Christian Nagel", "John D. Reid", "Matthew Reynolds",
            "Morgan Skinner", "Eric White"));
        books.Add(new Book("ASP.NET Professional Secrets",
            "Wiley", "978-0-7645-2628-2", "Bill Evjen",
            "Thiru Thangarathinam", "Bill Hatfield", "Doug Seven",
            "S. Srinivasa Sivakumar", "Dave Wanta", "Jason T. Roff"));
        books.Add(new Book("Design and Analysis of Distributed Algorithms",
            "Wiley", "978-0-471-71997-7", "Nicolo Santoro"));
    }

    public List<Book> GetBooks()
```

```

    {
        return books;
    }
}

```

在 WPF 后台代码中，类 Window1 的构造函数实例化了一个 BookFactory 类，并调用 GetBooks()方法，将 Book 数组赋予 Window1 实例的 DataContext 属性：

```

public partial class Window1 : System.Windows.Window
{
    private BookFactory factory = new BookFactory();

    public Window1()
    {
        InitializeComponent();

        this.DataContext = factory.GetBooks();
    }
}

```

在 XAML 中，只需一个支持列表的控件，如列表框，绑定 ItemSource 属性，如下所示：

```

<Window x:Class="ListBindingSample.Window1"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="List Binding Sample" Height="300" Width="518"
        >
    <DockPanel>
        <Grid>
            <Grid.ColumnDefinitions>
                <ColumnDefinition />
            </Grid.ColumnDefinitions>
            <Grid.RowDefinitions>
                <RowDefinition />
                <RowDefinition />
                <RowDefinition />
                <RowDefinition />
            </Grid.RowDefinitions>
            <ListBox HorizontalAlignment="Left" Margin="5" Grid.RowSpan="4"
                    Grid.Row="0" Grid.Column="0" Name="booksList"
                    ItemsSource="{Binding}" />
        </Grid>
    </DockPanel>
</Window>

```

因为 Window 将 Book 数组赋予 DataContext，列表框放在 Window 中，所以列表框会用默认模板显示所有的图书，如图 35-6 所示。

为了使列表框有更灵活的布局，必须定义一个模板，就像前一章为列表框定义样式那样。样式 listBoxStyle 包含的模板 ItemTemplate 定义了一个带标签元素的 DataTemplate。标签的内容绑定到 Title 上。数据项模板重复应用于列表中的每一项。

列表框元素指定了 Style 属性。ItemsSource 也设置为默认绑定。图 35-7 显示了使用新的列表框样式后应用程序的输出。

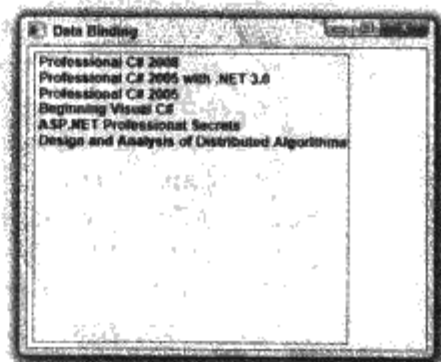


图 35-6

```

<Window.Resources>
  <Style x:Key="listBoxStyle" TargetType="{x:Type ListBox}" >
    <Setter Property="ItemTemplate">
      <Setter.Value>
        <DataTemplate>
          <Label Content="{Binding Title}" />
        </DataTemplate>
      </Setter.Value>
    </Setter>
  </Style>
</Window.Resources>

<!-- ... -->
<ListBox HorizontalAlignment="Left" Margin="5"
  Style="{StaticResource listBoxStyle}" Grid.RowSpan="4"
  ItemsSource="{Binding}" />

```

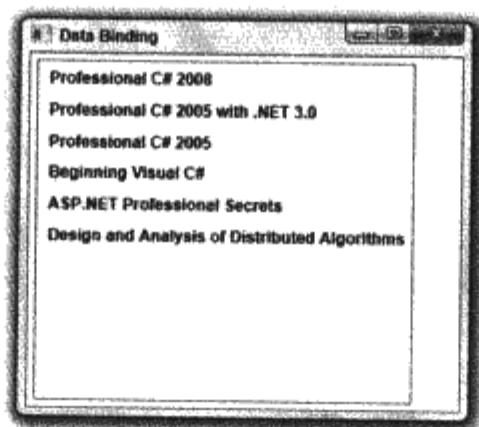


图 35-7

### 1. 主从绑定

除了显示列表中的所有元素之外，还应能显示选中项的详细信息。这不需要做太多的工作。只需将元素定义为显示当前选项即可。在示例程序中，用 **Binding** 标记扩展将三个标签元素分别定义为 **Book** 的属性 **Title**、**Publisher** 和 **Isbn**。这里必须对列表框进行一个重要的修改。在默认情况下，标签绑定到列表的第一项上。设置列表框的属性 **IsSynchronizedWithCurrentItem = "True"**，就会把列表框的选项设置为当前项。在图 35-8 中显示了结果：选中的项显示在详细信息标签中。

```

<Window x:Class="Wrox.ProCSharp.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="List Binding Sample" Height="300" Width="518"
  >
  <Window.Resources>
    <Style x:Key="listBoxStyle" TargetType="{x:Type ListBox}" >
      <Setter Property="ItemTemplate">
        <Setter.Value>
          <DataTemplate>
            <Label Content="{Binding Title}" />
          </DataTemplate>
        </Setter.Value>
      </Setter>
    </Style>
    <Style x:Key="labelStyle" TargetType="{x:Type Label}">
      <Setter Property="Width" Value="190" />
    </Style>
  </Window.Resources>

```

```

        <Setter Property="Height" Value="40" />
        <Setter Property="Margin" Value="5,5,5,5" />
    </Style>
</Window.Resources>
<DockPanel>
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition />
            <RowDefinition />
            <RowDefinition />
        </Grid.RowDefinitions>

        <ListBox IsSynchronizedWithCurrentItem="True" HorizontalAlignment="Left"
            Margin="5" Style="{StaticResource listBoxStyle}"
            Grid.RowSpan="4" ItemsSource="{Binding}" />
        <Label Style="{StaticResource labelStyle}" Content="{Binding Title}"
            Grid.Row="0" Grid.Column="1" />
        <Label Style="{StaticResource labelStyle}" Content="{Binding Publisher}"
            Grid.Row="1" Grid.Column="1" />
        <Label Style="{StaticResource labelStyle}" Content="{Binding Isbn}"
            Grid.Row="2" Grid.Column="1" />
    </Grid>
</DockPanel>
</Window>

```

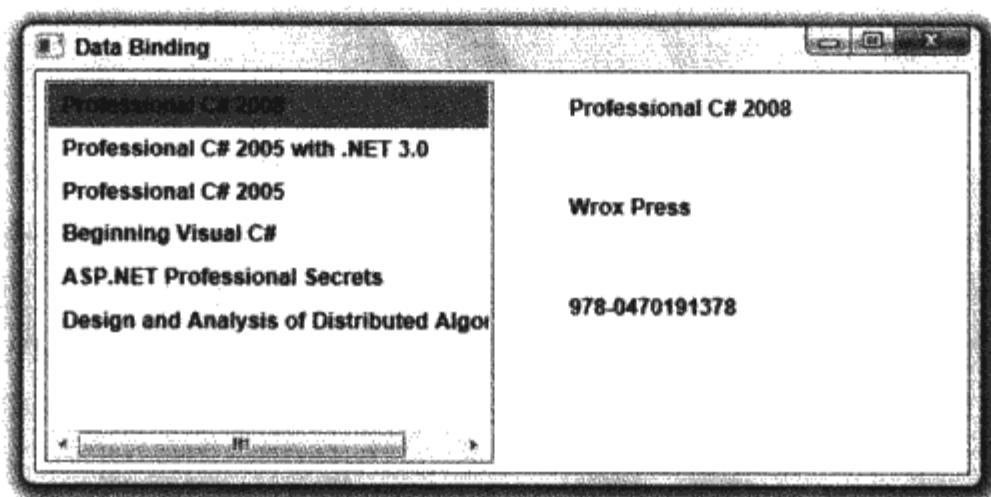


图 35-8

## 2. 值的转换

图书的作者还没有显示在结果中。如果将 `Authors` 属性绑定到标签元素上, 就要调用 `Array` 类的 `ToString()` 方法, 它只返回类型的名称。一种解决方法是将 `Authors` 属性绑定到一个列表框上。对于列表框, 可以定义一个模板, 显示特定的视图。另一种解决方法是将 `Authors` 属性返回的字符串数组转换为一个字符串, 再将该字符串用于绑定。

类 `StringArrayConverter` 可以将字符串数组转换为字符串。WPF 转换器类必须实现命名空间 `System.Windows.Data` 中的接口 `IValueConverter`。这个接口定义了方法 `Convert()` 和 `ConvertBack()`。在 `StringArrayConverter` 中, `Convert()` 方法会通过 `String.Join()` 方法把 `value` 变量中的字符串数组转换为字符串。Join() 方法的分隔符参数值从 `Convert()` 方法返回的 `parameter` 变

量中提取。

提示：

String 类的方法的更多信息参见第 8 章。

```
public class StringArrayConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter,
        System.Globalization.CultureInfo culture)
    {
        string[] stringCollection = (string[])value;
        string separator = (string)parameter;

        return String.Join(separator, stringCollection);
    }

    public object ConvertBack(object value, Type targetType, object parameter,
        System.Globalization.CultureInfo culture)
    {
        string s = (string)value;
        char separator = (char)parameter;

        return s.Split(separator);
    }
}
```

在 XAML 代码中，StringArrayConverter 类可以声明为一个资源，以便在 Binding 标记扩展中引用它：

```
<Window x:Class="Wrox.ProCSharp.WPF.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:src="clr-namespace:Wrox.ProCSharp.WPF"
    Title="List Binding Sample" Height="300" Width="518"
>
<Window.Resources>
    <src:StringArrayConverter x:Key="stringArrayConverter" />
    <!-- ... -->
```

为了输出多个结果，声明一个 TextBlock 元素，其 TextWrapping 属性设置为 Wrap，以便显示多个作者。在 Binding 标记扩展中，Path 设置为 Authors，它定义为一个返回字符串数组的属性。Converter 属性指定字符串数组从资源 stringArrayConverter 中转换。转换器的 Convert 方法将 ConverterParameter 参数，' ' 作为输入来分隔多个作者。

```
<TextBlock Width="190" Height="50" Margin="5" TextWrapping="Wrap"
    Text="{Binding Path=Authors,
        Converter={StaticResource stringArrayConverter},
        ConverterParameter=', '}"
    Grid.Row="3" Grid.Column="1" />
```

图 35-9 显示了图书的细节，包括作者。



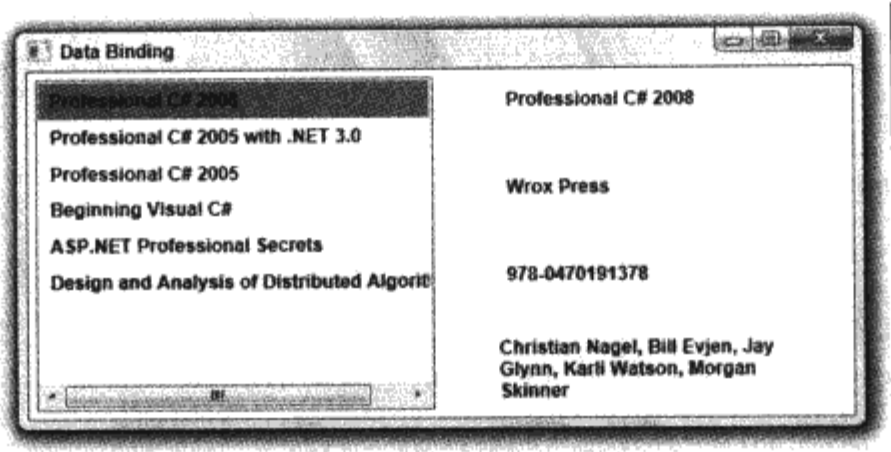


图 35-9

3. 动态添加列表项

如果列表项要动态添加，该怎么办？必须通知 WPF 元素：要在列表中添加元素。

在 WPF 应用程序的 XAML 代码中，要给 StackPanel 添加一个按钮元素。给 Click 事件指定方法 OnAddBook():

```
<!-- ... -->
<DockPanel>
  <StackPanel Orientation="Horizontal" DockPanel.Dock="Bottom" Height="60">
    <Button Click="addBookButton_Click" Name="addBookButton"
      Margin="5" Width="80" Height="40">Add Book</Button>
  </StackPanel>
  <Grid >
<!-- ... -->
```

在方法 OnAddBook()中，包含了 addBookButton()方法的事件处理程序代码，这个方法将一个新的 Book 对象添加到列表中。如果用 BookFactory 测试应用程序(因为它已实现)，不会通知 WPF 元素：已在列表中添加了一个新对象。

```
void OnAddBook(object sender, RoutedEventArgs e)
{
    factory.AddBook(new Book(".NET 2.0 Wrox Box", "Wrox Press",
        "978-0-470-04840-5"));
}
```

赋予 DataContext 的对象必须实现接口 INotifyCollectionChanged。这个接口定义了由 WPF 应用程序使用的 CollectionChanged 事件。除了用定制的集合类实现这个接口之外，还可以使用泛型集合类 ObservableCollection<T>，该类在 WindowsBase 程序集的 System.Collections.ObjectModel 命名空间中定义。现在，把一个新数据项添加到集合中，这个新数据项会立即显示在列表框中。

```
public class BookFactory
{
    private ObservableCollection<Book> books = new ObservableCollection<Book>();

    // ...

    public void AddBook(Book b)
    {
        books.Add(b);
    }
}
```

```

public ObservableCollection<Book> GetBooks()
{
    return books;
}

```

#### 4. 数据模板

上一章介绍了如何用模板来定制控件。还可以为数据类型定义模板，例如 `Book` 类。无论在何处使用 `Book` 类，都使用该模板定义默认外观。

在下面的例子中，在 `Window` 资源内部定义 `DataTemplate`。`DataType` 属性引用命名空间 `Wrox.ProCSharp.WPF` 中的 `Book` 类。该模板定义，在 `StackPanel` 中包含一个边框和两个标签元素。列表框元素并没有引用该模板。列表框定义的唯一一个属性是 `ItemsSource`，其值是默认的 `Binding` 标记扩展。因为 `DataTemplate` 没有定义键，所以由包含 `Book` 对象的所有列表使用。图 35-10 显示了使用数据模板的应用程序的结果。

```

<Window x:Class="Wrox.ProCSharp.WPF.DataTemplateDemo"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:src="clr-namespace:Wrox.ProCSharp.WPF"
        Title="Data Template Sample" Height="300" Width="300"
        >
    <Window.Resources>
        <DataTemplate DataType="{x:Type src:Book}">
            <Border BorderBrush="Blue" BorderThickness="2" Background="LightBlue"
                Margin="10" Padding="15">
                <StackPanel>
                    <Label Content="{Binding Path=Title}" />
                    <Label Content="{Binding Path=Publisher}" />
                </StackPanel>
            </Border>
        </DataTemplate>
    </Window.Resources>
    <Grid>
        <ListBox ItemsSource="{Binding}" />
    </Grid>
</Window>

```

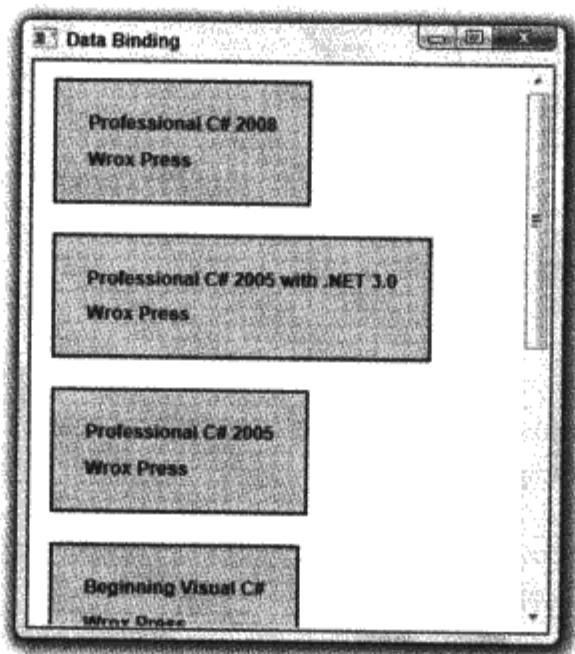


图 35-10

如果要对相同的数据类型使用另一个数据模板，可以创建一个数据模板选择器。数据模板选择器在派生于基类 `DataTemplateSelector` 的类中实现。

下面的数据模板选择器根据发布者选择另一个模板。在 Window 资源中，定义了这些模板。一个模板可以通过键名 `WroxBookTemplate` 来访问，另一个模板的键名是 `WileyBookTemplate`：

```
< DataTemplate x:Key="WroxBookTemplate" DataType="{x:Type src:Book}" >
  < Border BorderBrush="Blue" BorderThickness="2" Background="LightBlue"
    Margin="10" Padding="15" >
    < StackPanel >
      < Label Content="{Binding Path=Title}" / >
      < Label Content="{Binding Path=Publisher}" / >
    < /StackPanel >
  < /Border >
< /DataTemplate >

< DataTemplate x:Key="WileyBookTemplate" DataType="{x:Type src:Book}" >
  < Border BorderBrush="Yellow" BorderThickness="2"
    Background="LightGreen" Margin="10" Padding="15" >
    < StackPanel >
      < Label Content="{Binding Path=Title}" / >
      < Label Content="{Binding Path=Publisher}" / >
    < /StackPanel >
  < /Border >
< /DataTemplate >
```

要选择模板，类 `BookDataTemplateSelector` 必须重写基类 `DataTemplateSelector` 中的 `SelectTemplate` 方法。其实现代码根据 `Book` 类中的 `Publisher` 属性选择模板：

```
using System.Windows;
using System.Windows.Controls;

namespace Wrox.ProCSharp.WPF
{
    public class BookDataTemplateSelector : DataTemplateSelector
    {
        public override DataTemplate SelectTemplate(object item,
            DependencyObject container)
        {
            if (item != null && item is Book)
            {
                Window window = Application.Current.MainWindow;

                Book book = item as Book;
                switch (book.Publisher)
                {
                    case "Wrox Press":
                        return window.FindResource("WroxBookTemplate")
                            as DataTemplate;
                    case "Wiley":
                        return window.FindResource("WileyBookTemplate")
                            as DataTemplate;
                    default:
                        return window.FindResource("BookTemplate") as DataTemplate;
                }
            }
        }
    }
}
```

```

    }
    }
    return null;
}
}
}

```

要在 XAML 代码中访问类 `BookDataTemplateSelector`，这个类必须在 Window 资源中定义：

```
< src:BookDataTemplateSelector x:Key="bookTemplateSelector" / >
```

现在选择器类可以赋予 `ListBox` 的 `ItemTemplateSelector` 属性：

```
< ListBox ItemsSource="{Binding}"
    ItemTemplateSelector="{StaticResource bookTemplateSelector}" / >
```

运行这个应用程序，可以看到不同发布者的不同数据模板，如图 35-11 所示。

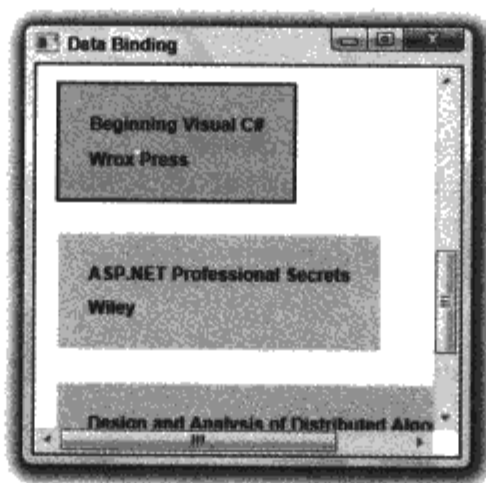


图 35-11

### 35.1.6 绑定到 XML 上

WPF 数据绑定还支持绑定到 XML 数据上。可以将 `XmlDataProvider` 用作数据源，使用 XPath 表达式绑定元素。为了分层次显示，可以使用 `TreeView` 控件，通过 `HierarchicalData Template` 为数据项创建视图。

下面包含 `Book` 元素的 XML 文件将用作后面例子的数据源：

```
<?xml version="1.0" encoding="utf-8" ?>
<Books>
  <Book isbn="978-0-470-12472-7">
    <Title>Professional C# 2008</Title>
    <Publisher>Wrox Press</Publisher>
    <Author>Christian Nagel</Author>
    <Author>Bill Evjen</Author>
    <Author>Jay Glynn</Author>
    <Author>Karli Watson</Author>
    <Author>Morgan Skinner</Author>
  </Book>
  <Book isbn="978-0-7645-4382-1">
    <Title>Beginning Visual C# 2008</Title>
    <Publisher>Wrox Press</Publisher>
    <Author>Karli Watson</Author>
    <Author>David Espinosa</Author>
    <Author>Zach Greenvoss</Author>
  </Book>
</Books>
```

```

    <Author>Jacob Hammer Pedersen</Author>
    <Author>Christian Nagel</Author>
    <Author>John D. Reid</Author>
    <Author>Matthew Reynolds</Author>
    <Author>Morgan Skinner</Author>
    <Author>Eric White</Author>
  </Book>
</Books>

```

与定义对象数据提供程序类似，也可以定义 `XmlDataProvider`。`ObjectDataProvider` 和 `XmlDataProvider` 都派生自同一个基类 `DataSourceProvider`。在示例的 `XmlDataProvider` 中，`Source` 属性设置为引用 XML 文件 `books.xml`。`XPath` 属性定义了一个 `XPath` 表达式，以引用 XML 根元素 `Books`。`Grid` 元素通过 `DataContext` 属性引用 XML 数据源。在栅格的 `DataContext` 属性中，所有的 `Book` 元素都需要列表绑定，所以 `XPath` 表达式设置为 `Book`。在栅格中，把列表框元素绑定到默认的数据环境中，并使用 `DataTemplate` 将标题包含在 `TextBlock` 元素中，作为列表框的数据项。在栅格中，还有三个标签元素，它们的数据绑定设置为 `XPath` 表达式，以显示标题、出版社和 ISBN 号。

```

<Window x:Class="XmlBindingSample.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="XmlBindingSample" Height="348" Width="498"
  >
  <Window.Resources>
    <XmlDataProvider x:Key="books" Source="Books.xml" XPath="Books" />

    <DataTemplate x:Key="listTemplate">
      <TextBlock Text="{Binding XPath=Title}" />
    </DataTemplate>

    <Style x:Key="labelStyle" TargetType="{x:Type Label}">
      <Setter Property="Width" Value="190" />
      <Setter Property="Height" Value="40" />
      <Setter Property="Margin" Value="5,5,5,5" />
    </Style>

  </Window.Resources>
  <Grid DataContext="{Binding Source={StaticResource books}, XPath=Book}">
    <Grid.ColumnDefinitions>
      <ColumnDefinition />
      <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition />
      <RowDefinition />
      <RowDefinition />
    </Grid.RowDefinitions>

    <ListBox IsSynchronizedWithCurrentItem="True" Margin="5,5,5,5"
      Grid.Column="0" Grid.RowSpan="4" ItemsSource="{Binding}"
      ItemTemplate="{StaticResource listTemplate}" />
    <Label Style="{StaticResource labelStyle}" Content="{Binding XPath=Title}"
      Grid.Row="0" Grid.Column="1" />
    <Label Style="{StaticResource labelStyle}" Content="{Binding XPath=Publisher}"
      Grid.Row="1" Grid.Column="1" />
    <Label Style="{StaticResource labelStyle}" Content="{Binding XPath=@isbn}"
      Grid.Row="2" Grid.Column="1" />
  </Grid>

```



```
</Grid>
</Window>
```

图 35-12 显示了 XML 绑定的结果。

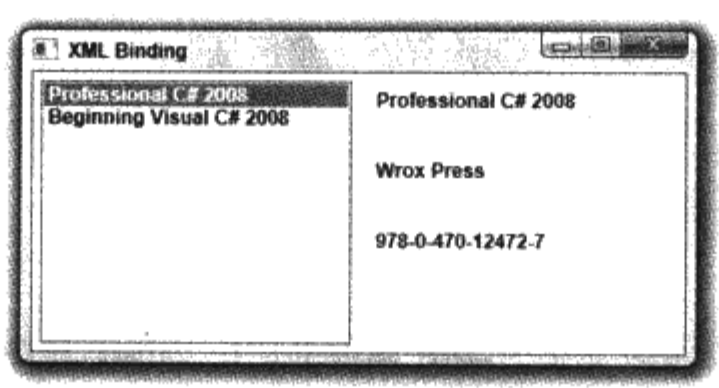


图 35-12

如果 XML 数据应以层次结构的方式显示，就可以使用 TreeView 控件。

### 35.1.7 绑定的验证

在把数据用于 .NET 对象之前，有几个选项可验证用户的数据，这些选项如下：

- 处理异常
- 数据错误信息
- 定制验证规则

#### 1. 处理异常

这里演示的一个选项是如果在类 `SomeData` 中设置了无效值，这个 .NET 类就抛出一个异常。属性 `Value1` 只接受大于等于 5 且小于 12 的值：

```
public class SomeData
{
    private int value1;
    public int Value1 {
        get
        {
            return value1;
        }
        set
        {
            if (value < 5 || value > 12)
                throw new ArgumentException(
                    "value must not be less than 5 or greater than 12");
            value1 = value;
        }
    }
}
```

在 `Window1` 类的构造函数中，初始化 `SomeData` 类的一个新对象，并传送给 `DataContext`，用于数据绑定：

```
public partial class Window1 : Window
{
    SomeData pl = new SomeData() { Value1 = 11 };
}
```

```
public Window1()
{
    InitializeComponent();
    this.DataContext = pl;
}
```

事件处理程序方法 `buttonSubmit_Click` 显示一个消息框，列出 `SomeData` 实例的实际值：

```
private void buttonSubmit_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show(pl.Value1.ToString());
}
```

在简单的数据绑定中，把文本框的 `Text` 属性绑定到 `Value1` 属性上。如果现在运行应用程序，试图把值改为某个无效值，单击 `Submit` 按钮，就可以验证该值永远不会改变。WPF 会捕获并忽略属性 `Value1` 的 `set` 访问器抛出的异常。

```
< Label Margin="5" Grid.Row="0" Grid.Column="0" > Value1: < /Label >
< TextBox Margin="5" Grid.Row="0" Grid.Column="1"
    Text="{Binding Path=Value1}" / >
```

要在输入字段的内容发生变化时显示错误，可以把 `Binding` 标记扩展的 `ValidatesOnException` 属性设置为 `True`。输入一个无效值(设置该值时，会抛出一个异常)，文本框就会以红色线条框出，如图 35-13 所示。

```
< Label Margin="5" Grid.Row="0" Grid.Column="0" > Value1: < /Label >
< TextBox Margin="5" Grid.Row="0" Grid.Column="1"
    Text="{Binding Path=Value1, ValidatesOnExceptions=True}" / >
```

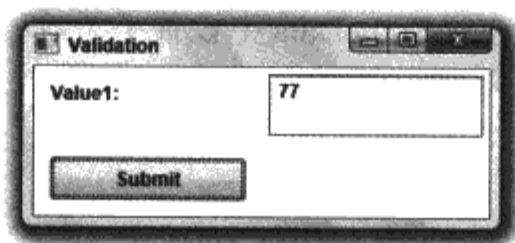


图 35-13

要以另一种方式给用户返回错误信息，可以把 `Validation` 类定义的关联属性 `ErrorTemplate` 赋予一个为错误定义 UI 的模板。标记错误的新模板用键 `validationTemplate` 表示。`ControlTemplate` 在已有的控件内容前面添加了一个红色的感叹号。

```
< ControlTemplate x:Key="validationTemplate" >
    < DockPanel >
        < TextBlock Foreground="Red" FontSize="20" > ! < /TextBlock >
        < AdornedElementPlaceholder / >
    < /DockPanel >
< /ControlTemplate >
```

用 `Validation.ErrorTemplate` 关联属性设置 `validationTemplate` 会激活带文本框的模板：

```
< Label Margin="5" Grid.Row="0" Grid.Column="0" > Value1: < /Label >
< TextBox Margin="5" Grid.Row="0" Grid.Column="1"
    Text="{Binding Path=Value1, ValidatesOnExceptions=True}"
```

```
Validation.ErrorTemplate="{StaticResource validationTemplate}" />
```

应用程序的新界面如图 35-14 所示。

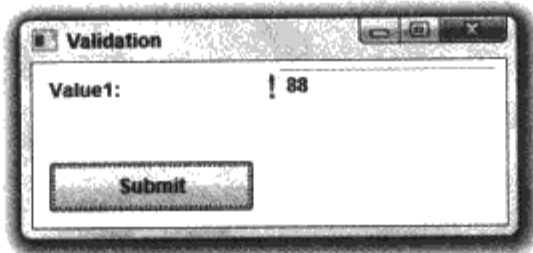


图 35-14

#### 提示:

定制错误消息的另一个选项是注册 `Validation` 类的 `Error` 事件。这里必须把属性 `NotifyOnValidationError` 设置为 `true`。

错误信息可以在 `Validation` 类的 `Errors` 集合中访问。要在文本框的工具提示中显示错误信息，可以创建一个属性触发器，如下所示。只要 `Validation` 类的 `HasError` 属性设置为 `True`，就激活触发器。触发器设置文本框的 `ToolTip` 属性：

```
< Style TargetType="{x:Type TextBox}" >
  < Style.Triggers >
    < Trigger Property="Validation.HasError" Value="True" >
      < Setter Property="ToolTip"
        Value="{Binding RelativeSource=
          {x:Static RelativeSource.Self},
          Path=(Validation.Errors)[0].ErrorContent}" />
    < /Trigger >
  < /Style.Triggers >
< /Style >
```

## 2. 数据错误信息

处理错误的另一种方式是确定 .NET 对象是否执行了接口 `IDataErrorInfo`。

类 `SomeData` 现在改为执行接口 `IDataErrorInfo`。这个接口定义了属性 `Error` 和带字符串参数的索引器。在数据绑定的过程中验证 WPF 时，会调用索引器，把要验证的属性名传送为 `columnName` 参数。在执行代码中，会验证其值是否有效，如果无效就传送一个错误字符串。

下面验证属性 `Value2`，它是使用 C# 3.0 简单属性标记实现的。

```
public class SomeData : IDataErrorInfo
{
    private int value1;
    public int Value1 {
        get
        {
            return value1;
        }
        set
        {
            if (value < 5 || value > 12)
                throw new ArgumentException(
                    "value must not be less than 5 or greater than 12");
            value1 = value;
        }
    }
}
```

```

    }
    public int Value2 { get; set; }

    string IDataErrorInfo.Error
    {
        get
        {
            return null;
        }
    }
    string IDataErrorInfo.this[string columnName]
    {
        get
        {
            if (columnName == "Value2")
            {
                if (this.Value2 < 0 || this.Value2 > 80)
                    return "age must not be less than 0 or greater than 80";
            }
            return null;
        }
    }
}

```

**提示：**

在.NET 实体类中，索引器返回什么内容并不清楚，例如调用索引器，会从 **Person** 类型的对象中返回什么？因此最好在接口 **IDataErrorInfo** 中包含显式的执行代码。这样，这个索引器只能使用接口来访问，.NET 类可以有另一个执行方式，以实现其他目的。

如果把 **Binding** 类的 **ValidationOnDataErrors** 属性设置为 **true**，就在数据绑定的过程中使用接口 **IDataErrorInfo**。这里，改变文本框时，绑定机制会调用接口的索引器，把 **Value2** 传送给 **columnName** 变量：

```

< Label Margin="5" Grid.Row="1" Grid.Column="0" > Value2: < /Label >
< TextBox Margin="5" Grid.Row="1" Grid.Column="1"
    Text="{Binding Path=Value2, ValidatesOnDataErrors=True}" / >

```

**3. 定制验证规则**

为了更多地控制验证，可以实现定制验证规则。实现了定制验证规则的类必须派生于基类 **ValidationRule**。在前面的两个例子中，也使用了验证规则。派生于 **ValidationRule** 抽象基类的两个类是 **DataErrorValidationRule** 和 **ExceptionValidationRule**。设置属性 **Validates- OnDataErrors**，使用接口 **IDataErrorInfo**，就激活了 **DataErrorValidationRule**。**Exception- ValidationRule** 处理异常，设置 **ValidationOnException** 会激活 **ExceptionValidationRule**。

下面实现一个验证规则，来验证正则表达式。类 **RegularExpressionValidationRule** 派生于基类 **ValidationRule**，重写了基类定义的抽象方法 **Validate()**。在其执行代码中，使用命名空间 **System.Text.RegularExpressions** 中的 **Regex** 类验证 **Expression** 属性定义的表达式。

```

public class RegularExpressionValidationRule : ValidationRule
{
    public string Expression { get; set; }
    public string ErrorMessage { get; set; }
}

```

```

public override ValidationResult Validate(object value,
    CultureInfo cultureInfo)
{
    ValidationResult result = null;
    if (value != null)
    {
        Regex regEx = new Regex(Expression);
        bool isMatch = regEx.IsMatch(value.ToString());
        result = new ValidationResult(isMatch, isMatch ?
            null : ErrorMessage);
    }
    return result;
}
}

```

这里没有使用 **Binding** 标记扩展，而是把绑定作为 **TextBox.Text** 元素的一个子元素。绑定的对象现在定义了 **Email** 属性，它用简单的属性语法来实现。**UpdateSourceTrigger** 属性定义了源代码何时更新。更新源代码的选项如下：

- 属性值变化时更新，可以是用户输入的每个字符时更新
- 焦点失去时更新
- 显式指定更新时间

**ValidationRules** 是 **Binding** 类的一个属性，它包含 **ValidationRule** 元素。这里使用的验证规则是定制类 **RegularExpressionValidationRule**，其中 **Expression** 属性设置为一个正则表达式，用于验证输入是否是有效的电子邮件，**ErrorMessage** 属性给出 **TextBox** 的输入数据无效时显示的错误消息：

```

< Label Margin="5" Grid.Row="2" Grid.Column="0" > Email: < /Label >
< TextBox Margin="5" Grid.Row="2" Grid.Column="1" >
    < TextBox.Text >
        < Binding Path="Email" UpdateSourceTrigger="LostFocus" >
            < Binding.ValidationRules >
                < src:RegularExpressionValidationRule
                    Expression="([\w-\.]*)@((\[[0-9]{1,3}\. [0-9]{1,3}\. [0-9]{1,3}\.])
                    |(([\w-]+\.)+)[a-zA-Z]{2,4}|[0-9]{1,3}))(\?)?$"
                    ErrorMessage="Email is not valid" / >
            < /Binding.ValidationRules >
        < /Binding >
    < /TextBox.Text >
< /TextBox >

```

## 35.2 命令绑定

WPF 的 **Menu** 和 **ToolBar** 控件与 Windows 窗体的对应控件有相同的功能：启动命令。使用这些控件，可以添加事件处理程序，执行命令。启动命令还有其他方式：选择菜单，单击工具栏按钮，按下键盘上的某个键。为了处理这些不同的输入方式，WPF 提供了另一个功能：命令。

一些 WPF 控件还提供了预定义命令的执行代码，更便于获得某些功能。

WPF 通过命令类提供了一些预定义的命令：**ApplicationCommands**、**EditingCommands**、**ComponentCommands** 和 **NavigationCommands**。这些命令类都是静态类，其静态属性返回 **RoutedUICommand** 对象。例如，**ApplicationCommands** 的属性包括 **New**、**Open**、**Save**、**SaveAs**、



Print 和 Close。许多应用程序都有这些命令。

为了说明这些命令的功能，创建一个简单的 WPF 项目，添加一个 Menu 控件，其菜单项包含 Undo、Redo、Cut、Copy 和 Paste。文本框 textContent 占用了窗口中的剩余空间，可以输入多行文本。在窗口中，创建了一个 DockPanel，来定义布局。使带有 MenuItem 元素的 Menu 控件停靠在顶边，标题设置为定义菜单的文本。\_定义了无需使用鼠标、可以直接用键盘访问菜单项的字母。按下 Alt 键时，标题文本中的字母下面就会出现下划线。Command 属性定义了与菜单项关联的命令。

```
<Window x:Class="Wrox.ProCSharp.WPF.WPFEditorWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="WPF Editor" Height="300" Width="300"
>
    < DockPanel >
        < Menu DockPanel.Dock="Top" >
            < MenuItem Header="_Edit" >
                < MenuItem Name="editUndoMenu" Header="_Undo"
                    Command="ApplicationCommands.Undo" / >
                < MenuItem Name="editRedoMenu" Header="_Redo"
                    Command="ApplicationCommands.Redo" / >
                < Separator / >
                < MenuItem Name="editCutMenu" Header="Cu_t"
                    Command="ApplicationCommands.Cut" / >
                < MenuItem Name="editCopyMenu" Header="_Copy"
                    Command="ApplicationCommands.Copy" / >
                < MenuItem Name="editPasteMenu" Header="_Paste"
                    Command="ApplicationCommands.Paste" / >
            < /MenuItem >
        < /Menu >
        < TextBox Name="textContent" TextWrapping="Wrap" AcceptsReturn="True"
            AcceptsTab="True" / >
    < /DockPanel >
</Window>
```

这就是要为剪切板功能做的工作。TextBox 类已经包含了这些预定义命令绑定的功能。启动应用程序，在文本框中输入文本时，就可以看到菜单项。在文本框中选择文本，可以使用 Cut 和 Copy 菜单项。图 35-15 显示了运行着的应用程序。

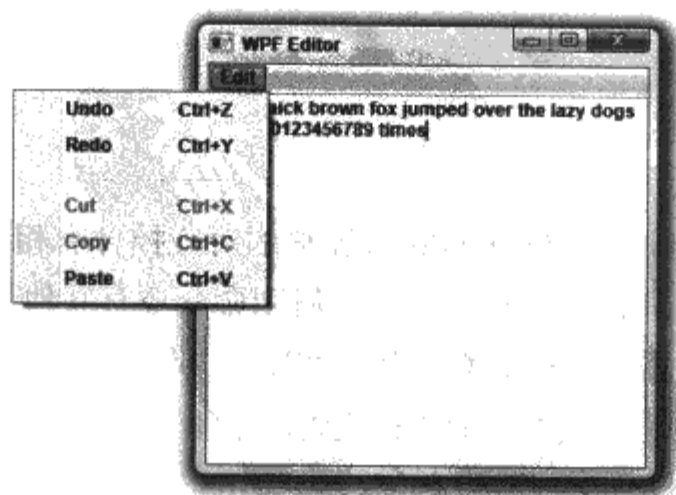


图 35-15

现在修改应用程序，添加前面没有用控件间定义的命令绑定。把打开和保存文件的命令添加到编辑器中。

为了使命令可以访问，应在 **Menu** 元素中添加更多的 **MenuItem** 元素，如下所示：

```
< MenuItem Header="_File" >
  < MenuItem Name="fileNewMenu" Header="_New"
    Command="ApplicationCommands.New" / >
  < MenuItem Name="fileOpenMenu" Header="_Open"
    Command="ApplicationCommands.Open" / >
  < Separator / >
  < MenuItem Name="fileSave" Header="_Save"
    Command="ApplicationCommands.Save" / >
  < MenuItem Name="fileSaveAs" Header="Save _As"
    Command="ApplicationCommands.SaveAs" / >
< /MenuItem >
```

命令也可以从工具栏中访问。对于 **ToolBar** 元素，定义了可以从工具栏上访问的命令。为了安排工具栏，将 **ToolBar** 元素放在 **ToolBarTray** 中：

```
<ToolBarTray DockPanel.Dock="Top">
  <ToolBar>
    <Button Command="ApplicationCommands.New">
      <Image Source="toolbargraphics/New.bmp" />
    </Button>
    <Button Command="ApplicationCommands.Open">
      <Image Source="toolbargraphics/Open.bmp" />
    </Button>
    <Button Command="ApplicationCommands.Save">
      <Image Source="toolbargraphics/Save.bmp" />
    </Button>
  </ToolBar>
</ToolBarTray>
```

现在需要定义命令绑定，把命令关联到事件处理程序上。命令绑定可以赋予派生自层次结构中最上面的 **UIElement** 基类的任意 WPF 类。可以定义 **CommandBinding** 元素，将命令绑定赋予 **CommandBindings** 属性。**CommandBinding** 类的 **Command** 属性可以指定一个实现了  **ICommand** 接口的对象、为事件 **CanExecute** 和 **Executed** 指定事件处理程序。示例程序把命令绑定赋予 **Window** 类。**Executed** 事件设置为实现命令功能的事件处理程序方法。如果命令不应在所有情况下都是可用的，就可以将事件 **CanExecute** 设置为一个处理程序，指定命令是否可用。

```
< Window.CommandBindings >
  < CommandBinding Command="ApplicationCommands.New"
    Executed="NewFileExecuted" / >
  < CommandBinding Command="ApplicationCommands.Open"
    Executed="OpenFileExecuted" / >
  < CommandBinding Command="ApplicationCommands.Save"
    Executed="SaveFileExecuted"
    CanExecute="SaveFileCanExecute" / >
  < CommandBinding Command="ApplicationCommands.SaveAs"
    Executed="SaveAsFileExecuted" CanExecute="SaveFileCanExecute" / >
< /Window.CommandBindings >
```

在处理程序方法的后台代码中，**NewFileExecuted()**会清空文本框，将文件名 **untitled.txt** 写入 **Window** 类的 **Title** 属性。在 **OpenFileExecuted()**中，创建了 **Microsoft.Win32.OpenFileDialog**，并显示为一个对话框。成功退出对话框后，会打开所选的文件，将其内容写入 **TextBox** 控件。

**提示:**

打开文件的对话框在 WPF 中没有预定义。可以为选择文件和文件夹创建一个定制窗口，也可以使用 Microsoft.Win32 命名空间中的 OpenFileDialog 类，它封装在 Windows 对话框中。

```
public partial class Window1 : System.Windows.Window
{
    private string fileName;
    private readonly string defaultFileName;
    private const string appName = "WPF Editor";
    private bool isChanged = false;

    public Window1()
    {
        defaultFileName = System.IO.Path.Combine(
            Environment.GetFolderPath(
                Environment.SpecialFolder.MyDocuments),
            @"untitled.txt");
        InitializeComponent();
        NewFile();
    }

    private void NewFileExecuted(object sender, ExecutedRoutedEventArgs e)
    {
        NewFile();
    }

    private void NewFile()
    {
        textContent.Clear();
        filename = defaultFilename;
        SetTitle();
        isChanged = false;
    }

    private void SetTitle()
    {
        Title = String.Format("{0} {1}",
            System.IO.Path.GetFileName(filename), appName);
    }

    private void OpenFileExecuted
        (object sender, ExecutedRoutedEventArgs e)
    {
        try
        {
            OpenFileDialog dlg = new OpenFileDialog();
            bool? dialogResult = dlg.ShowDialog();
            if (dialogResult == true)
            {
                filename = dlg.FileName;
                SetTitle();
                textContent.Text = File.ReadAllText(filename);
            }
        }
        catch (IOException ex)
        {
            MessageBox.Show(ex.Message, "Error WPF Editor",
                MessageBoxButton.OK, MessageBoxImage.Error);
        }
    }
}
```

```
}
```

处理程序 `SaveFileCanExecute()` 根据内容是否有变化，确定保存文件的命令是否可用：

```
private void SaveFileCanExecute(object sender,
    CanExecuteRoutedEventArgs e)
{
    if (isChanged)
    {
        e.CanExecute = true;
    }
    else
    {
        e.CanExecute = false;
    }
}
```

应用程序打开了文件 `sample.txt`，如图 35-16 所示。

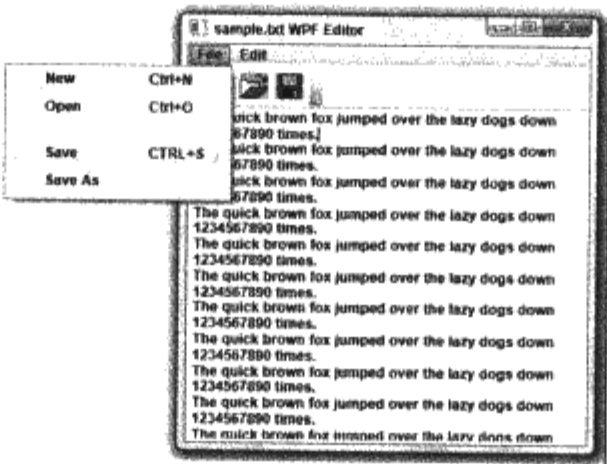


图 35-16

### 35.3 动画

在动画中，可以使用移动的元素、通过颜色的改变、变换等制作顺畅的切换效果。WPF 使动画的制作非常简单。还可以连续改变任意依赖属性的值。不同的动画类可以根据其类型，连续改变不同属性的值。

动画的主要元素如下：

- 时间线：定义了值随时间的变化方式。有不同类型的时间线，可用于改变不同类型的值。所有时间线的基类都是 `Timeline`。为了连续改变 `double`，可以使用 `DoubleAnimation` 类。`Int32Animation` 类是 `int` 值的动画类。
- 情节板：用于合并动画。`Storyboard` 类派生自基类 `TimelineGroup`，`TimelineGroup` 又派生自基类 `Timeline`。使用 `DoubleAnimation`，可以连续改变 `double`，使用 `Storyboard` 类可以合并所有的动画。
- 触发器：通过触发器可以启动和停止动画。前面介绍了属性触发器。当属性值变化时，属性触发器就会启动。还可以创建事件触发器，当事件发生时，事件触发器就会启动。

提示：

动画类的命名空间是 `System.Windows.Media.Animation`。

### 35.3.1 时间线

时间线定义了值随时间变化的方式。第一个示例连续改变椭圆的大小。其中 `DoubleAnimation` 就是时间线，它改变了所使用的 `double`。Ellipse 类的 `Triggers` 属性设置为 `EventTrigger`。椭圆加载时，就触发用 `EventTrigger` 的 `RoutedEvent` 属性定义的事件触发器。`BeginStoryboard` 是启动故事板的启动操作。在故事板中，`DoubleAnimation` 元素用于连续改变 Ellipse 类的 `Width` 属性。动画在 3 秒内把椭圆的宽度从 100 改为 300，在之后的 3 秒内再改回来。

```
< Window x:Class="EllipseAnimation.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Ellipse Animation" Height="300" Width="300" >
  < Grid >
    < Ellipse Height="50" Width="100" Fill="SteelBlue" >
      < Ellipse.Triggers >
        < EventTrigger RoutedEvent="Ellipse.Loaded" >
          < EventTrigger.Actions >
            < BeginStoryboard >
              < Storyboard Duration="00:00:06" RepeatBehavior="Forever" >
                < DoubleAnimation
                  Storyboard.TargetProperty="(Ellipse.Width)"
                  Duration="0:0:3" AutoReverse="True"
                  FillBehavior="Stop" RepeatBehavior="Forever"
                  AccelerationRatio="0.9" DecelerationRatio="0.1"
                  From="100" To="300" / >
              < /Storyboard >
            < /BeginStoryboard >
          < /EventTrigger.Actions >
        < /EventTrigger >
      < /Ellipse.Triggers >
    < /Ellipse >
  < /Grid >
< /Window >
```

图 35-17 和 35-18 显示了连续改变的椭圆的两个状态。



图 35-17



图 35-18

动画并不仅仅是连续显示在屏幕上的一般窗口动画。还可以给业务应用程序添加动画，使用户界面的响应更好。

下面的例子演示了一个相当好的动画，还说明了如何在样式中定义动画。在 `Window` 资源中，有一个用于按钮的样式 `AnimatedButtonStyle`。在模板中定义了一个矩形 `outline`。这个模



板使用很细的笔触，其宽度设置为 0.4。

该模板为 IsMouseOver 属性定义了一个属性触发器。当鼠标滑过按钮时，就应用这个触发器的 EnterActions 属性。启动操作是 BeginStoryboard，它是一个触发器动作，可以包含并启动 Storyboard 元素。Storyboard 元素定义了一个 DoubleAnimation，可以连续改变 double 值。在这个动画中改变的属性值是矩形元素 outline 的 StrokeThickness 属性。该值平滑地改为 1.2，因为 By 属性指定，该属性变化的时间长度是 Duration 属性设置的 0.3 秒。在动画结束时，笔触的宽度重新设置为其初始值，因为 AutoReverse="True"。总之，只要鼠标滑过按钮，outline 的边框就在 0.3 秒内增加 1.2。图 35-19 显示了没有改变的按钮，图 35-20 显示了鼠标滑过按钮 0.3 秒后的按钮。在纸质媒介上，不可能显示平滑的动画和按钮外观的中间状态。

```
<Window x:Class="AnimationSample.ButtonAnimation"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Animation Sample" Height="300" Width="300">

    <Window.Resources>
        <Style x:Key="AnimatedButtonStyle" TargetType="{x:Type Button}">
            <Setter Property="Template">
                <Setter.Value>
                    <ControlTemplate TargetType="{x:Type Button}">
                        <Grid>

                            <Rectangle Name="outline" RadiusX="9" RadiusY="9" Stroke="Black"
                                Fill="{TemplateBinding Background}" StrokeThickness="0.4">
                            </Rectangle>

                            <ContentPresenter VerticalAlignment="Center"
                                HorizontalAlignment="Center"
                                />
                        </Grid>
                        <ControlTemplate.Triggers>
                            <Trigger Property="IsMouseOver" Value="True">
                                <Trigger.EnterActions>
                                    <BeginStoryboard>
                                        <Storyboard>
                                            <DoubleAnimation Duration="0:0:0.3" AutoReverse="True"
                                                Storyboard.TargetProperty="(Rectangle.StrokeThickness)"
                                                Storyboard.TargetName="outline" By="1.2" />
                                        </Storyboard>
                                    </BeginStoryboard>
                                </Trigger.EnterActions>
                            </Trigger>
                        </ControlTemplate.Triggers>
                    </ControlTemplate>
                </Setter.Value>
            </Setter>
        </Style>
    </Window.Resources>
    <Grid>

        <Button Style="{StaticResource MyButtonStyle}" Width="200" Height="100">
            Click Me!
        </Button>
    </Grid>
</Window>
```



图 35-19



图 35-20

Timeline 可以完成的任务如表 35-4 所示。

表 35-4

Timeline 属性	说 明
AutoReverse	使用 AutoReverse 属性，可以指定连续改变的值在动画结束后是否返回初始值
SpeedRatio	使用 SpeedRatio，可以改变动画的执行速度。在这个属性中，可以定义父子元素的相对关系。默认值为 1；将速率设置为较小的值，会使动画执行较慢；将速率设置为高于 1 的值，会使动画执行较快
BeginTime	使用 BeginTime，可以指定从触发器事件开始到动画开始之间的时间长度。其单位可以是天、小时、分钟、秒和几分之秒。根据 SpeedRatio，这可以不是真实的时间。例如，如果 SpeedRatio 设置为 2，开始时间设置为 6 秒，动画就在 3 秒后开始
AccelerationRatio DecelerationRatio	在动画中，值不一定是线性变化。可以指定 AccelerationRatio 或 Deceleration Ratio，定义加速度和减速度。这两个值的总和不能超过 1
Duration	使用 Duration 属性，可以指定动画执行一次的时间长度
RepeatBehavior	给 RepeatBehavior 属性指定一个 RepeatBehavior 结构，可以定义动画的重复次数或重复时间
FillBehavior	如果父元素的时间线有不同的持续时间，则 FillBehavior 属性就很重要。例如，如果父元素的时间线比实际动画的时间短，则将 FillBehavior 设置为 Stop 就表示动画停止。如果父元素的时间线比实际动画的时间长，HoldEnd 就会一直执行动画，直到连续改变的值重新设置为其初始值为止(假定 AutoReverse 设置为 true)

根据 Timeline 类的类型，还可以使用其他一些属性。例如，使用 DoubleAnimation，可以指定如表 35-5 所示的属性。

表 35-5

DoubleAnimation 属性	说 明
From, To	设置 From 和 To 属性，可以指定开始和结束动画的值
By	除了为动画定义开始值之外，还可以设置 By 属性，用绑定属性的当前值启动动画，该属性值会递增由 By 属性指定的值，直到动画结束为止

35.3.2 触发器

除了使用属性触发器之外，还可以定义一个事件触发器，来启动动画。下一个例子将为前面示例中的笑脸创建一个动画，一旦引发了按钮的 Click 事件，眼睛就会移动。这个例子还说明，可以在 XAML 和后台代码中启动动画。

图 35-21 显示了这个笑脸动画的运行结果。

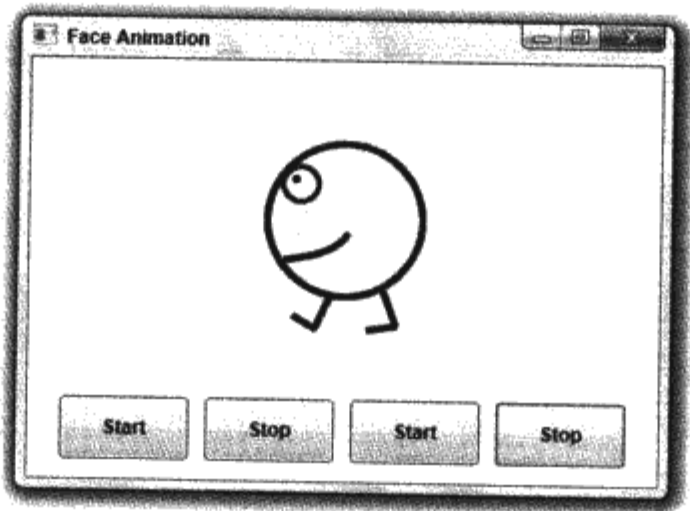


图 35-21

在 Window 元素中，定义了一个 DockPanel 元素，来安排笑脸和按钮的位置。包含 Canvas 元素的栅格停靠在顶部。停靠在底部的是包含四个按钮的 StackPanel 元素。前两个按钮用于在后台代码中连续改变眼睛的位置，后两个按钮用于在 XAML 中连续改变眼睛的位置。

动画在<DockPanel.Triggers>段中定义。这里没有使用属性触发器，而使用了事件触发器。RoutedEvent 和 SourceName 属性定义了按钮 startButtonXAML，该按钮的 Click 事件发生时，就启动第一个事件触发器。触发器的动作由 BeginStoryboard 元素定义，它启动所包含的 Storyboard。BeginStoryboard 定义了一个名称，它用于控制情节板的暂停、继续和停止动作。Storyboard 元素包含两个动画。第一个动画改变眼睛的 Canvas.Left 位置值，第二个动画改变 Canvas.Top 值。这两个动画有不同的时间值，使用指定的重复执行方式使眼睛的运动更有趣。

在按钮 stopButtonXAML 的 Click 事件发生时，就启动第二个事件触发器。在这里，情节板用 StopStoryboard 元素停止，该元素引用了起始情节板 beginMoveEye。

```
<Window x:Class="AnimatedFace.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Animated Face" Height="300" Width="406">
  <DockPanel>

    <Grid DockPanel.Dock="Top">
```

```

<!-- Funny Face -->
<Canvas Width="200" Height="200">
  <Ellipse Canvas.Left="50" Canvas.Top="50" Width="100" Height="100"
    Stroke="Blue" StrokeThickness="4" Fill="Yellow" />
  <Ellipse Canvas.Left="60" Canvas.Top="65" Width="25" Height="25"
    Stroke="Blue" StrokeThickness="3" Fill="White" />
  <Ellipse Name="eye" Canvas.Left="67" Canvas.Top="72" Width="5"
    Height="5" Fill="Black" />
  <Path Name="mouth" Stroke="Blue" StrokeThickness="4"
    Data="M 62,125 Q 95,122 102,108" />
  <Line Name="LeftLeg" X1="92" X2="82" Y1="146" Y2="168" Stroke="Blue"
    StrokeThickness="4" />
  <Line Name="LeftFoot" X1="68" X2="83" Y1="160" Y2="169" Stroke="Blue"
    StrokeThickness="4" />
  <Line Name="RightLeg" X1="124" X2="132" Y1="144" Y2="166" Stroke="Blue"
    StrokeThickness="4" />
  <Line Name="RightFoot" X1="114" X2="133" Y1="169" Y2="166" Stroke="Blue"
    StrokeThickness="4" />
</Canvas>
</Grid>

```

```

<StackPanel DockPanel.Dock="Bottom" Orientation="Horizontal">
  <Button Width="80" Height="40" Margin="20,5,5,5"
    Name="startAnimationButton">Start</Button>
  <Button Width="80" Height="40" Margin="5,5,5,5"
    Name="stopAnimationButton">Stop</Button>
  <Button Width="80" Height="40" Margin="5,5,5,5"
    Name="startButtonXAML">Start</Button>
  <Button Width="80" Height="40" Margin="5,5,5,5"
    Name="stopButtonXAML">Stop</Button>
</StackPanel>

```

```

<DockPanel.Triggers>

```

```

  <EventTrigger RoutedEvent="Button.Click" SourceName="startButtonXAML">
    <BeginStoryboard Name="beginMoveEye">
      <Storyboard Name="moveEye">
        <DoubleAnimation RepeatBehavior="Forever" DecelerationRatio=".8"
          AutoReverse="True" By="8" Duration="0:0:1"
          Storyboard.TargetName="eye"
          Storyboard.TargetProperty="(Canvas.Left)" />
        <DoubleAnimation RepeatBehavior="Forever" AutoReverse="True" By="8"
          Duration="0:0:5" Storyboard.TargetName="eye"
          Storyboard.TargetProperty="(Canvas.Top)" />
      </Storyboard>
    </BeginStoryboard>
  </EventTrigger>

  <EventTrigger RoutedEvent="Button.Click" SourceName="stopButtonXAML">
    <StopStoryboard BeginStoryboardName="beginMoveEye" />
  </EventTrigger>

```

```

</DockPanel.Triggers>
</DockPanel>

```

```

</Window>

```

除了在 XAML 的事件触发器中直接启动和停止动画之外，还可以在后台代码中控制动画。按钮 `startAnimationButton` 和 `stopAnimationButton` 分别关联了事件处理程序 `OnStartAnimation` 和

OnStopAnimation。在这些事件处理程序中，动画用 Begin()方法启动，用 Stop()方法停止。在 Begin()方法中，第二个参数设置为 true，允许用停止请求来控制动画。

```
public partial class Window1 : System.Windows.Window
{
    public Window1()
    {
        InitializeComponent();
        startAnimationButton.Click += OnStartAnimation;
        stopAnimationButton.Click += OnStopAnimation;
    }

    void OnStartAnimation(object sender, RoutedEventArgs e)
    {
        moveEye.Begin(eye, true);
    }

    void OnStopAnimation(object sender, RoutedEventArgs e)
    {
        moveEye.Stop(eye);
    }
}
```

现在就可以启动应用程序，单击一个 Start 按钮，观看眼睛的移动了。

35.3.3 故事板

Storyboard 类继承自基类 Timeline，但可以包含几个时间线。Storyboard 类可以用于控制时间线。表 35-6 描述了 Storyboard 类的方法。

表 35-6

Storyboard 类的方法	说 明
Begin()	Begin()方法启动与情节板关联的动画
BeginAnimation()	BeginAnimation()方法可以为一个依赖属性启动单个动画
CreateClock()	CreateClock()方法返回一个 Clock 对象，用于控制动画
Pause(), Resume()	使用 Pause()和 Resume()可以暂停、恢复动画的播放
Seek()	使用 Seek()方法，可以使动画移动到指定的时间处
Stop()	Stop()方法挂起时钟，停止动画

EventTrigger 类可以定义事件发生时的操作。表 35-7 描述了这个类的属性。

表 35-7

EventTrigger 类的属性	说 明
RoutedEvent	使用 RoutedEvent 属性，可以定义在触发器开始时的事件，例如按钮的 Click 事件
SourceName	SourceName 属性确定事件应连接到哪个 WPF 元素上



可以放在 `EventTrigger` 中的触发器动作如表 35-8 所示。在前面的例子中，使用了 `BeginStoryboard` 和 `StopStoryboard` 动作，表 35-8 列出了其他动作。

表 35-8

TriggerAction 类	说 明
SoundPlayerAction	使用 SoundPlayerAction，可以播放.wav 文件
BeginStoryboard	BeginStoryboard 启动由 Storyboard 定义的动画
PauseStoryboard	PauseStoryboard 暂停动画
ResumeStoryboard	ResumeStoryboard 重新启动暂停的动画
StopStoryboard	StopStoryboard 停止运行的动画
SeekStoryboard	SeekStoryboard 可以改变动画的当前时间
SkipStoryboardToFill	SkipStoryboardToFill 使动画向前移动到结束时间
SetStoryboardSpeedRatio	SetStoryboardSpeedRatio 可以改变动画的播放速度

35.4 在 WPF 中添加 3D 特性

本节介绍 WPF 中的 3D 特性，其中包含了开始使用该特性的信息。

提示：

WPF 中的 3D 特性在 `System.Windows.Media.Media3D` 命名空间中。

为了理解 WPF 中的 3D 特性，一定要知道坐标系统之间的区别。图 35-22 显示了 WPF 3D 中的坐标系统。原点位于中心。X 轴的正值在右边，负值在左边。Y 轴是垂直的，正值在上边，负值在下边。Z 轴在指向观察者的方向上定义了正值。

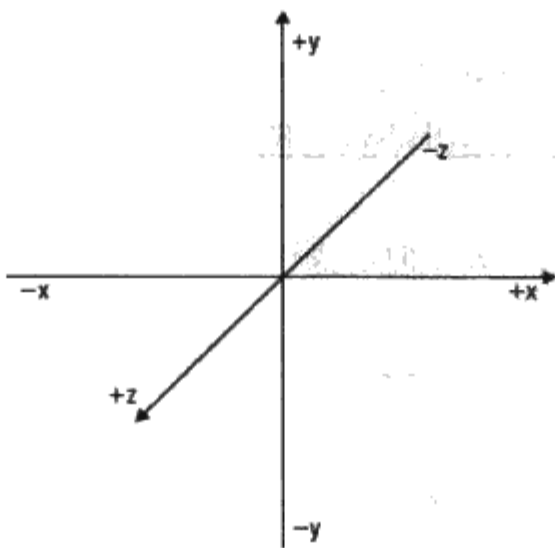


图 35-22

表 35-9 描述了最重要的类及其功能。

表 35-9	
类	说 明
ViewPort3D	ViewPort3D 定义了 3D 对象的渲染表面。这个元素包含 3D 绘图的所有可见元素
ModelVisual3D	ModelVisual3D 包含在 ViewPort3D 中，它包含了所有可见元素。可以给完整的模型指定变换
GeometryModel3D	GeometryModel3D 包含在 ModelVisual3D 中，它包含网格和材质
Geometry3D	Geometry3D 是一个抽象基类，定义了几何形状。派生于 Geometry3D 的类是 MeshGeometry3D。使用 MeshGeometry3D 可以定义三角形的位置，建立 3D 模型
Material	Material 是一个抽象基类，定义了 MeshGeometry3D 指定的三角形的前边和后边。Material 包含在 GeometryModel3D 中。.NET 3.5 定义了几个材质类，例如 DiffuseMaterial、EmissiveMaterial 和 SpecularMaterial。根据材质的类型，以不同的方式计算灯光。EmissiveMaterial 利用灯光的计算，使材质发出等于笔刷颜色的光。DiffuseMaterial 使用漫射光，SpecularMaterial 定义了镜面发光模型。使用 MaterialGroup 类可以创建由其他材质合并而成的材质
Light	Light 是灯光的抽象基类。其派生类有 AmbientLight、DirectionalLight、PointLight 和 SpotLight。AmbientLight 是不自然的光，会近似照亮整个场景。使用这种光看不到边界。DirectionalLight 定义了定向光。太阳光就是一种定向光，光线来自一边，此时可以看到边界和阴影。PointLight 是一种位于指定位置的光，会照亮所有的方向。SpotLight 照亮指定的方向。这个光定义了一个圆锥，会得到一个发出光亮的区域
Camera	Camera 是摄像机的抽象基类，用于把 3D 场景映射为 2D 显示。其派生类是 PerspectiveCamera、OrthographicCamera 和 MatrixCamera。在 PerspectiveCamera 中，3D 对象离得越远就越小，这不同于 OrthographicCamera，在 Orthographic Camera 中，摄像机的距离对对象的大小没有影响。在 MatrixCamera 中，可以在矩阵中定义视图和变换
Transform3D	Transform3D 是 3D 变换的抽象基类。其派生类是 RotateTransform3D、ScaleTransform3D、TranslateTransform3D、MatrixTransform3D 和 Transform3D Group。TranslateTransform3D 允许在 x、y 和 z 向上变换对象，ScaleTransform3D 可以重置对象的大小。RotateTransform3D 可以在 x、y 和 z 向上把对象旋转指定的角度。Transform3DGroup 可以合并其他变换效果

三角形

本节从一个简单的 3D 示例开始。3D 模型由三角形组成，所以最简单的模型是一个三角形。三角形用 MeshGeometry3D 的 Positions 属性定义。3 个顶点都使用相同的 z 坐标-4，x、y 坐标分别为-1-1、1-1 和 01。属性 TriangleIndices 指定了逆时针的位置顺序。使用这个属性可以确定三角形的哪一边是可见的。三角形的一边显示了用 GeometryModel3D 类的 Meterial 属性定义的颜色，其他边显示了 BackMeterial 属性定义的颜色。

用于显示场景的摄像机位于坐标 0,0,0，其方向指向 0,0,-8。把摄像机的位置改变到左边，矩形就移动到右边，反之亦然。改变摄像机的 y 位置，矩形就会变大或变小。

这个场景中使用的光线是 `AmbientLight`，它用白色光照亮了整个场景。图 35-23 显示了三角形的效果。

```
< Window x:Class="Triangle3D.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="3D" Height="300" Width="300" >
  < Grid >
    < Viewport3D >
      < Viewport3D.Camera >
        < PerspectiveCamera Position="0 0 0" LookDirection="0 0 -8" / >
      < /Viewport3D.Camera >

      < ModelVisual3D >
        < ModelVisual3D.Content >
          < AmbientLight Color="White" / >
        < /ModelVisual3D.Content >
      < /ModelVisual3D >

      < ModelVisual3D >
        < ModelVisual3D.Content >
          < GeometryModel3D >
            < GeometryModel3D.Geometry >
              < MeshGeometry3D
                Positions="-1 -1 -4, 1 -1 -4, 0 1 -4"
                TriangleIndices="0, 1, 2" / >
            < /GeometryModel3D.Geometry >
          < GeometryModel3D.Material >
            < MaterialGroup >
              < DiffuseMaterial >
                < DiffuseMaterial.Brush >
                  < SolidColorBrush Color="Red" / >
                < /DiffuseMaterial.Brush >
              < /DiffuseMaterial >
            < /MaterialGroup >
          < /GeometryModel3D.Material >
        < /GeometryModel3D >
      < /ModelVisual3D.Content >
    < /ModelVisual3D >
  < /Viewport3D >
< /Grid >
< /Window >
```



图 35-23

## 1. 改变光线

图 35-23 仅显示了一个简单的三角形，它与 2D 的效果相同。但是，下面将继续添加 3D 特性。例如，用 `SpotLight` 元素把环境光改为聚光灯，就可以看到三角形的另一个外观，使用聚光灯可以定义光源的位置和光线的照射方向。给光源的位置指定 `-1 1 2`，光就位于三角形的左边顶点处，其 `y` 坐标是三角形的高度。之后，光线向下向左照射。图 35-24 显示了三角形的新外观。

```
< ModelVisual3D >
  < ModelVisual3D.Content >
    < SpotLight Position="-1 1 2" Color="White"
      Direction="-1.5, -1, -5" / >
  < /ModelVisual3D.Content >
< /ModelVisual3D >
```

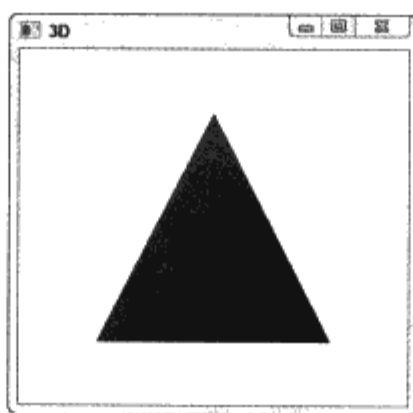


图 35-24

## 2. 添加纹理

除了给三角形的材质使用纯色笔刷之外，还可以使用其他笔刷，例如 `LinearGradient-Brush`，如下面的 XAML 代码所示。用 `DiffuseMaterial` 定义的 `LinearGradientBrush` 元素指定了黄色、橙色、红色、蓝色和紫罗兰色的渐变点。要把使用这种笔刷的对象的 2D 表面映射到 3D 几何体上，必须设置 `TextCoordinates` 属性。`TextCoordinates` 定义了 2D 点的集合，它可以映射到 3D 位置上。图 35-25 显示了示例应用程序中笔刷的 2D 坐标。三角形中的第一个位置 `-1 -1` 映射到笔刷坐标 `0 1` 上，右下角的位置 `1 -1` 映射到笔刷的 `1 1` 上，即紫罗兰色；`0 1` 映射到 `0.5 0` 上。图 35-26 显示了材质为渐变笔刷的三角形，这里也使用了环境光。

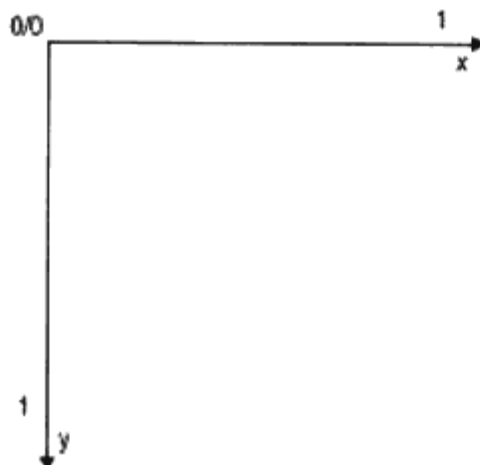


图 35-25

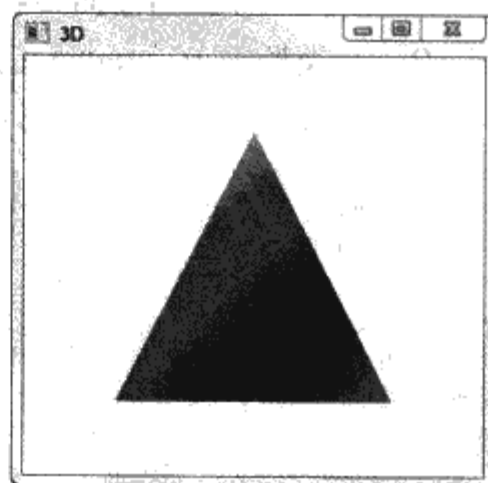


图 35-26

```

< ModelVisual3D >
  < ModelVisual3D.Content >
    < GeometryModel3D >
      < GeometryModel3D.Geometry >
        < MeshGeometry3D
          Positions="-1 -1 -4, 1 -1 -4, 0 1 -4"
          TriangleIndices="0, 1, 2"
          TextureCoordinates="0 1, 1 1, 0.5 0" / >
        < /GeometryModel3D.Geometry >

      < GeometryModel3D.Material >
        < MaterialGroup >
          < DiffuseMaterial >
            < DiffuseMaterial.Brush >
              < LinearGradientBrush StartPoint="0,0"
                EndPoint="1,1" >
                < GradientStop Color="Yellow" Offset="0" / >
                < GradientStop Color="Orange" Offset="0.25" / >
                < GradientStop Color="Red" Offset="0.50" / >
                < GradientStop Color="Blue" Offset="0.75" / >
                < GradientStop Color="Violet" Offset="1" / >
              < /LinearGradientBrush >
            < /DiffuseMaterial.Brush >
          < /DiffuseMaterial >
        < /MaterialGroup >
      < /GeometryModel3D.Material >
    < /GeometryModel3D >
  < /ModelVisual3D.Content >
< /ModelVisual3D >

```

**提示:**

可以用类似的方式添加文本和其他控件。为此，只需用要绘制的元素创建 `VisualBrush`。  
`VisualBrush` 详见第 34 章。

**3. 3D 对象**

下面研究真正的 3D 对象：立方体。立方体由 5 个矩形组成：后面、前面、左面、右面和底面。每个矩形都由两个三角形组成，因为这是网格的核心。在 WPF 和 3D 术语中，网格用于描述建立 3D 形状的基本三角形。

下面是立方体中前面矩形的代码，该矩形由两个三角形组成。三角形的位置按 `TriangleIndices` 定义的逆时针设置。立方体的前面用红色的笔刷绘制，后面用灰色笔刷绘制。这两个笔刷都是 `SolidColorBrush` 类型，用 Window 的资源定义。

```

<!-- Front -->
< GeometryModel3D >
  < GeometryModel3D.Geometry >
    < MeshGeometry3D
      Positions="-1 -1 1, 1 -1 1, 1 1 1, 1 1 1,
        -1 1 1, -1 -1 1"
      TriangleIndices="0 1 2, 3 4 5" / >
    < /GeometryModel3D.Geometry >
  < GeometryModel3D.Material >
    < DiffuseMaterial Brush="{StaticResource redBrush}" / >
  < /GeometryModel3D.Material >
  < GeometryModel3D.BackMaterial >
    < DiffuseMaterial Brush="{StaticResource grayBrush}" / >
  < /GeometryModel3D.BackMaterial >
< /GeometryModel3D >

```



```

    < /GeometryModel3D.BackMaterial >
  < /GeometryModel3D >

```

其他矩形非常类似，只是在不同的位置上。下面是立方体左面的 XAML 代码：

```

<!-- Left side -->
< GeometryModel3D >
  < GeometryModel3D.Geometry >
    < MeshGeometry3D
      Positions="-1 -1 1, -1 1 1, -1 -1 -1, -1 -1 -1, -1 1 1,
        -1 1 -1"
      TriangleIndices="0 1 2, 3 4 5" / >
    < /GeometryModel3D.Geometry >
    < GeometryModel3D.Material >
      < DiffuseMaterial Brush="{StaticResource redBrush}" / >
    < /GeometryModel3D.Material >
    < GeometryModel3D.BackMaterial >
      < DiffuseMaterial Brush="{StaticResource grayBrush}" / >
    < /GeometryModel3D.BackMaterial >
  < /GeometryModel3D >

```

#### 提示：

示例代码为立方体的每个面定义了一个 `GeometryModel3D`，仅是为了更好地理解代码。只要每个面都使用相同的材质，就可以定义一个网格，它包含立方体所有面的全部 10 个三角形。

所有的矩形都在 `Model3DGroup` 中组合，所以可以对立方体的所有面进行变换：

```

<!-- the model -->
< ModelVisual3D >
  < ModelVisual3D.Content >
    < Model3DGroup >

      <!-- GeometryModel3D elements for every side of the box -->

    < /Model3DGroup >

```

使用 `Model3DGroup` 的 `Transform` 属性，就可以变换这个组中的所有几何体。下面使用 `RotateTransform3D` 定义一个 `AxisAngleRotation3D`。要在运行期间旋转立方体，`Angle` 属性要绑定到 `Slider` 控件的值上。

```

<!-- Transformation of the complete model -->
< Model3DGroup.Transform >
  < RotateTransform3D CenterX="0" CenterY="0" CenterZ="0" >
    < RotateTransform3D.Rotation >
      < AxisAngleRotation3D x:Name="axisRotation"
        Axis="0, 0, 0"
        Angle="{Binding Path=Value,
          ElementName=axisAngle}" / >
    < /RotateTransform3D.Rotation >
  < /RotateTransform3D >
< /Model3DGroup.Transform >

```

为了查看立方体，需要一个摄像机。这里使用 `PerspectiveCamera`，因此立方体离摄像机越远，就越小。摄像机的位置的方向在运行期间设置。

```

<!-- Camera -->
< Viewport3D.Camera >
  < PerspectiveCamera x:Name="camera"

```

```

        Position="{Binding Path=Text,
            ElementName=textCameraPosition}"
        LookDirection="{Binding Path=Text,
            ElementName=textCameraDirection}" / >
< /Viewport3D.Camera >

```

应用程序使用两个不同的光源，其中一个光源是 **DirectionalLight**：

```

<!-- directional light -->
< ModelVisual3D >
    < ModelVisual3D.Content >
        < DirectionalLight Color="White" x:Name="directionalLight" >
            < DirectionalLight.Direction >
                < Vector3D X="1" Y="2" Z="3" / >
            < /DirectionalLight.Direction >
        < /DirectionalLight >
    < /ModelVisual3D.Content >
< /ModelVisual3D >

```

另一个光源是 **SpotLight**。使用这个光源可以突出显示立方体的一个特定区域。**SpotLight** 定义了属性 **InnerConeAngle** 和 **OuterConeAngle**，以指定完全照亮的区域：

```

<!-- spot light -->
< ModelVisual3D >
    < ModelVisual3D.Content >
        < SpotLight x:Name="spotLight"
            InnerConeAngle="{Binding Path=Value,
                ElementName=spotInnerCone}"
            OuterConeAngle="{Binding Path=Value,
                ElementName=spotOuterCone}"
            Color="#FFFFFF"
            Direction="{Binding Path=Text, ElementName=spotDirection}"
            Position="{Binding Path=Text, ElementName=spotPosition}"
            Range="{Binding Path=Value, ElementName=spotRange}" / >
    < /ModelVisual3D.Content >
< /ModelVisual3D >

```

运行应用程序，就可以改变立方体的旋转角度、摄像机和灯光，如图 35-27 所示。

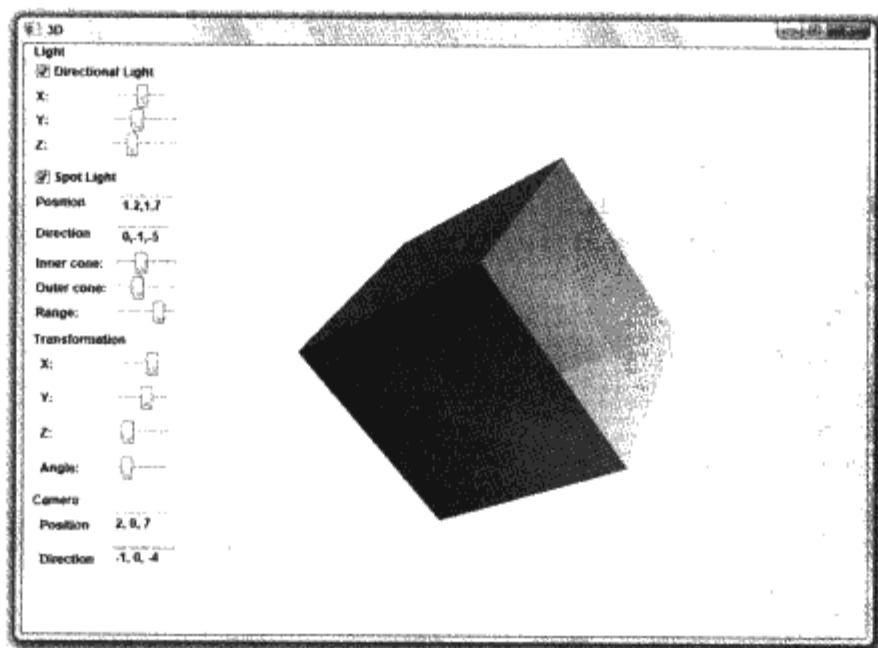


图 35-27

**提示:**

创建仅包含矩形或三角形的 3D 模型是很简单的。不应手工创建更复杂的模型, 而应使用对应的工具。WPF 的 3D 工具在 [www.codeplex.com/3DTools](http://www.codeplex.com/3DTools) 上。

## 35.5 Windows 窗体集成

除了给 WPF 从头开始编写用户界面之外, 还可以在 WPF 应用程序中使用已有的 Windows 窗体控件, 创建要在 Windows 窗体应用程序中使用的新 WPF 控件。集成 Windows 窗体和 WPF 的最佳方式是创建控件, 将它们集成到另一个应用程序中。

**警告:**

Windows 窗体和 WPF 的集成有一个很大的缺陷。如果集成了 Windows 窗体和 WPF, Windows 窗体控件仍显示以前的外观。Windows 窗体控件和应用程序并没有获得 WPF 的新外观。从用户界面的角度来看, 最好完全重写用户界面。

**提示:**

要集成 Windows 窗体和 WPF, 需要使用 WindowsFormsIntegration 程序集的 System.Windows.Forms.Integration 命名空间中的类。

### 35.5.1 Windows 窗体中的 WPF 控件

可以在 Windows 窗体应用程序中使用 WPF 控件。WPF 元素是一个一般的 .NET 类。但是, 不能在 Windows 窗体代码中直接使用它; WPF 控件不是 Windows 窗体控件。集成可以利用 System.Windows.Forms.Integration 命名空间中的 ElementHost 封装类来进行。ElementHost 是一个 Windows 窗体控件, 因为它派生自 System.Windows.Forms.Control, 在 Windows 窗体应用程序中, 可以像其他 Windows 窗体控件那样使用。ElementHost 包含和管理 WPF 控件。

下面是一个简单的 WPF 控件。在 Visual Studio 2008 中, 可以创建一个定制的 WPF 用户控件库。示例控件派生自基类 UserControl, 包含一个栅格和一个带定制内容的按钮。

```
<UserControl x:Class="WPFControl.UserControl1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" >
    <Grid>
        <Button>
            <Canvas Height="230" Width="230">
                <Ellipse Canvas.Left="50" Canvas.Top="50" Width="100" Height="100"
                    Stroke="Blue" StrokeThickness="4" Fill="Yellow" />
                <Ellipse Canvas.Left="60" Canvas.Top="65" Width="25" Height="25"
                    Stroke="Blue" StrokeThickness="3" Fill="White" />
                <Ellipse Canvas.Left="70" Canvas.Top="75" Width="5" Height="5" Fill="Black" />
                <Path Name="mouth" Stroke="Blue" StrokeThickness="4"
                    Data="M 62,125 Q 95,122 102,108" />
                <Line X1="124" X2="132" Y1="144" Y2="166" Stroke="Blue" StrokeThickness="4" />
                <Line X1="114" X2="133" Y1="169" Y2="166" Stroke="Blue" StrokeThickness="4" />
                <Line X1="92" X2="82" Y1="146" Y2="168" Stroke="Blue" StrokeThickness="4" />
            </Canvas>
        </Button>
    </Grid>
</UserControl>
```

```

        <Line X1="68" X2="83" Y1="160" Y2="168" Stroke="Blue" StrokeThickness="4" />
    </Canvas>
</Button>
</Grid>
</UserControl>

```

可以选择 Windows 窗体应用程序模板，创建一个 Windows 窗体应用程序。因为 WPF 用户控件项目在 Windows 窗体应用程序所在的解决方案中，所以可以把 WPF 用户控件从工具箱拖放到 Windows 窗体应用程序的设计界面上。这会添加对程序集 `PresentationCore`、`PresentationFramework`、`WindowsBase`、`WindowsFormsIntegration` 和包含 WPF 控件的程序集的引用。

在设计器生成的代码中，有一个引用 WPF 用户控件的变量和一个封装控件的 `ElementHost` 类型的对象。

```

private System.Windows.Forms.Integration.ElementHost elementHost1;
private WPFControl.UserControl1 userControl11;

```

在 `InitializeComponent` 方法中，初始化了对象，把 WPF 控件实例赋予 `ElementHost` 类的 `Child` 属性：

```

private void InitializeComponent()
{
    this.elementHost1 = new
        System.Windows.Forms.Integration.ElementHost();
    this.userControl11 = new WPFControl.UserControl1();
    this.SuspendLayout();
    //
    // elementHost1
    //
    this.elementHost1.Location = new System.Drawing.Point(39, 44);
    this.elementHost1.Name = "elementHost1";
    this.elementHost1.Size = new System.Drawing.Size(259, 229);
    this.elementHost1.TabIndex = 0;
    this.elementHost1.Text = "elementHost1";
    this.elementHost1.Child = this.userControl11;
    //...
}

```

启动 Windows 窗体应用程序，WPF 控件和 Windows 窗体控件会显示在一个窗体中，如图 35-28 所示。

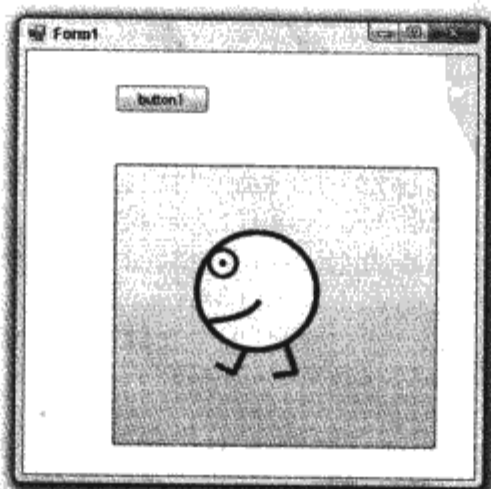


图 35-28

当然，还可以给 WPF 控件添加方法、属性和事件，再以与其他控件相同的方式使用它们。

### 35.5.2 WPF 应用程序中的 Windows 窗体控件

还可以用另一种方式集成 Windows 窗体和 WPF：把 Windows 窗体控件放在 WPF 应用程序中。与用于在 Windows 窗体中包含 WPF 控件的 `ElementHost` 类一样，现在需要一个封装器，它是一个包含 Windows 窗体控件的 WPF 控件。这个类的名称是 `WindowsFormsHost`，它也位于 `WindowsFormsIntegration` 程序集中。`WindowsFormsHost` 类派生自 `HwndHost` 和 `FrameworkElement`，因此可以用作 WPF 元素。

对于这种集成，应先创建一个 Windows 控件库。使用设计器给窗体添加一个文本框和一个按钮控件。修改按钮的 `Text` 属性，在后台代码中添加属性 `ButtonText`：

```
public partial class UserControl1 : UserControl
{
    public UserControl1()
    {
        InitializeComponent();
    }

    public string ButtonText
    {
        get { return button1.Text; }
        set { button1.Text = value; }
    }
}
```

在 WPF 应用程序中，可以把 `WindowsFormsHost` 对象从工具箱添加到设计器上。这需要引用程序集 `WindowsFormsIntegration`、`System.Windows.Forms` 和包含 Windows 窗体控件的程序集。要在 XAML 中使用 Windows 窗体控件，必须添加一个 XML 命名空间别名，以引用 .NET 命名空间。因为 Windows 窗体控件与 WPF 应用程序位于不同的程序集中，所以还必须将该程序集的名称添加到命名空间别名中。Windows 窗体控件现在可以包含在 `WindowsFormsHost` 元素中，如下所示。可以在 XAML 中，直接给属性 `ButtonText` 赋值，其方式与 .NET Framework 元素类似：

```
<Window x:Class="WPFApplication.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:wf="clr-namespace:Wrox.ProCSharp.WPF;assembly=WindowsFormsControl"
    Title="WPF Interop Application" Height="300" Width="300"
>
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
    </Grid.RowDefinitions>
    <WindowsFormsHost Grid.Row="0" Height="180">
        <wf:UserControl1 x:Name="myControl" ButtonText="Click Me!" />
    </WindowsFormsHost>
    <StackPanel Grid.Row="1">
        <TextBox Margin="5,5,5,5" Width="140" Height="30"></TextBox>
        <Button Margin="5,5,5,5" Width="80" Height="40">WPF Button</Button>
    </StackPanel>
</Grid>
</Window>
```



图 35-29 显示了 WPF 应用程序的一个视图。当然，Windows 窗体控件仍像一个 Windows 窗体控件，没有获得 WPF 提供的重新设置大小和样式特性。



图 35-29

## 35.6 WPF 浏览器应用程序

VS2008 有另一个 WPF 项目模板：WPF 浏览器应用程序。这个应用程序可以运行在 IE 中，但我们使用的 .NET Framework 版本必须安装在客户机系统中。在该模板中可以获得用于浏览器的丰富客户机功能。但是，在 WPF 浏览器应用程序中，需要把 .NET Framework 安装在客户机系统中，且仅支持 IE。

创建这样的项目类型，需要 XBAP(XAML Browser Application)文件。XBAP 是一个 XML 文件，定义了应用程序和它包含的程序集，用于 ClickOnce 部署。

XBAP 应用程序是一个部分信任的应用程序。只能在 Internet 权限中使用 .NET 代码。

**提示：**

ClickOnce 参见第 16 章。

**提示：**

WPF 浏览器应用程序不同于 Silverlight。Silverlight 定义了一个 WPF 子集，它不需要把 .NET Framework 安装在客户机系统中，但需要把一个插件安装在浏览器上，且支持不同的浏览器和不同的操作系统。Silverlight 1.0 不能用 .NET 编程，只能使用 JavaScript 编程访问 XAML 元素。Silverlight 1.1 支持 .NET Microframework。

## 35.7 小结

本章介绍了 WPF 的许多特性。WPF 的数据绑定特性比 Windows 窗体前进了一大步。可以把 .NET 类的任意属性绑定到 WPF 元素的属性上。绑定模式定义了绑定的方向。可以绑定 .NET 对象和列表，定义数据模板，为 .NET 类创建默认的外观。

命令绑定可以把处理程序的代码映射到菜单和工具栏上。还可以用 WPF 进行复制和粘贴，因为这个技术的命令处理程序已经包含在 `TextBox` 控件中。

动画允许用户动态改变 WPF 元素的每个属性。动画可以非常有趣，使 UI 响应更快，更吸引人。

WPF 还可以把 3D 映射到屏幕的 2D 表面上，我们学习了如何创建 3D 模型，用不同的光源和摄像机观察它。

**提示：**

本章和上一章概述了 WPF，所提供的信息足以使用这个技术。WPF 的更多信息可参阅有关 WPF 的图书，例如 Chris Andrade 等编著的 *Profession WPF Programming: .NET Development with the Windows Presentation Foundation*(Wiley 出版社于 2007 年出版)。

# 第 36 章

## 插 件

插件可以在以后给应用程序添加功能。我们可以创建一个主机应用程序，随时间的推移给它添加越来越多的功能——这些功能可以是开发团队编写的，其他供应商也可以创建插件，扩展该应用程序。

目前，插件在许多不同的应用程序上使用，例如 IE 和 Visual Studio。IE 是一个主机应用程序，它提供了一个插件框架，许多公司都使用这个框架提供查看 Web 页面时的扩展程序。Shockwave Flash Object 可以查看带 Flash 内容的 Web 页面。Google 工具栏提供了特殊的 Google 功能，可以在 IE 中快速访问。Visual Studio 也有一个插件模型，可以用不同层次的扩展程序扩展 Visual Studio。

定制应用程序总是可以创建插件模型，以动态加载和使用程序集中的功能。利用插件模型时，需要考虑许多问题。如何检测新的程序集？如何解决版本问题？插件可以改变主机应用程序的稳定性吗？

.NET Framework 3.5 提供了一个框架，用程序集 System.AddIn 来保存和创建插件。这个框架也称为 Managed AddIn Framework (MAF)。

**提示：**

插件还有其他称呼，如 add-on 或 plug-in。

本章内容如下：

- System.AddIn 体系结构
- 创建简单的插件

### 36.1 System.AddIn 体系结构

创建允许在运行期间添加插件的应用程序时，需要处理一些问题。例如，如何找到插件，如何解决版本问题，使主机应用程序和插件可以独立地升级。要解决这些问题，有几种方式。本节讨论插件的问题和 MAF 解决它们的体系结构：

- 插件的问题
- 管道体系结构
- 发现
- 激活
- 隔离

- 生存期
- 版本问题

36.1.1 插件的问题

要创建一个主机应用程序，动态加载以后添加的程序集，必须解决几个问题，如表 36-1 所示。

表 36-1

插件问题	说 明
发现	如何为主机应用程序查找新插件？这有几个不同的选项。一个选项是在配置文件中添加插件的信息。其缺点是安装新插件时，需要修改已有的配置文件。另一个选项是把包含插件的程序集复制到预定义的目录中，通过反射读取程序集的信息。  反射的更多内容可参见第 13 章
激活	程序集动态加载后，还不能使用 new 运算符创建它的实例。但可以用 Activator 类创建这类程序集。另外，如果插件加载到另一个应用程序域中新进程中，还需要使用不同的激活选项。程序集和应用程序域的更多内容可参见第 17 章
隔离	插件可能会使主机应用程序崩溃，读者可能见过 IE 因各种插件而崩溃的情况。根据主机应用程序的类型和插件的集成方式，插件可以加载到另一个应用程序域或另一个进程中
生存期	清理对象是垃圾回收器的工作。但是，垃圾回收器在这里没有任何帮助，因为插件可能在另一个应用程序域中或另一个进程中激活。把对象保存在内存中的其他方式有引用计数、租借和承办机制
版本	版本问题是插件的一个大问题。通常主机的一个新版本仍可以加载旧插件，而旧主机应有加载新插件的选项

下面探讨 MAF 的体系结构，说明这个框架如何解决这些问题。MAF 的设计目标如下：

- 应易于开发插件
- 在运行期间查找插件应很高效
- 开发主机程序应是一个很简单的过程，但不像开发插件那么容易
- 插件和主机应用程序应独立地升级

36.1.2 管道体系结构

MAF 体系结构基于一个包含 7 个程序集的管道。这个管道解决了插件的版本问题。因为管道中的程序集之间的依赖性很低，所以合同、主机程序和插件升级到新版本可以完全互不干扰。

图 36-1 显示了 MAF 体系结构的管道。其中心是合同程序集。这个程序集包含一个合同接口，其中列出了插件必须实现、可以由主机程序调用的方法和属性。合同的左边是主机端，右边是插件端。图中还显示了程序集之间的依赖性。最左端的主机程序集与合同程序集没有依赖性，插件程序集与合同程序集也没有依赖性，这两个程序集都没有实现合同定义的接口，只是有一个对视图程序集的引用。主机应用程序引用主机视图；插件引用插件视图。视图包含抽象

的视图类，该类定义的方法和属性与合同相同。

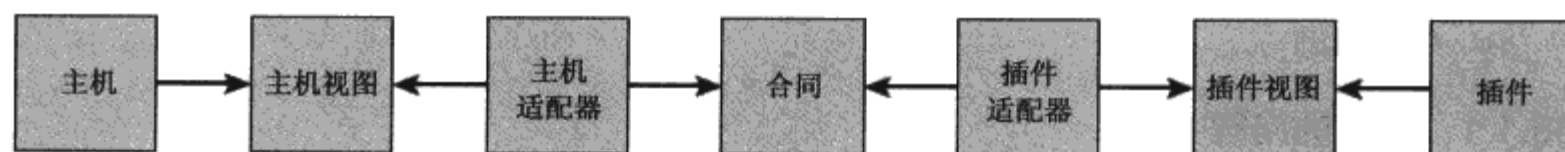


图 36-1

图 36-2 显示了管道中类的关系。主机类与抽象的主机视图类有一个关联，并调用其方法。抽象的主机视图类由主机适配器实现。适配器在视图和合同之间建立连接。插件适配器实现了合同的方法和属性。这个适配器包含对插件视图的引用，把来自主机端的调用传送给插件视图。主机适配器类定义了一个具体的类，它派生自主机视图的抽象基类，实现了方法和属性。这个适配器包含对合同的引用，把来自视图的调用传送给合同。

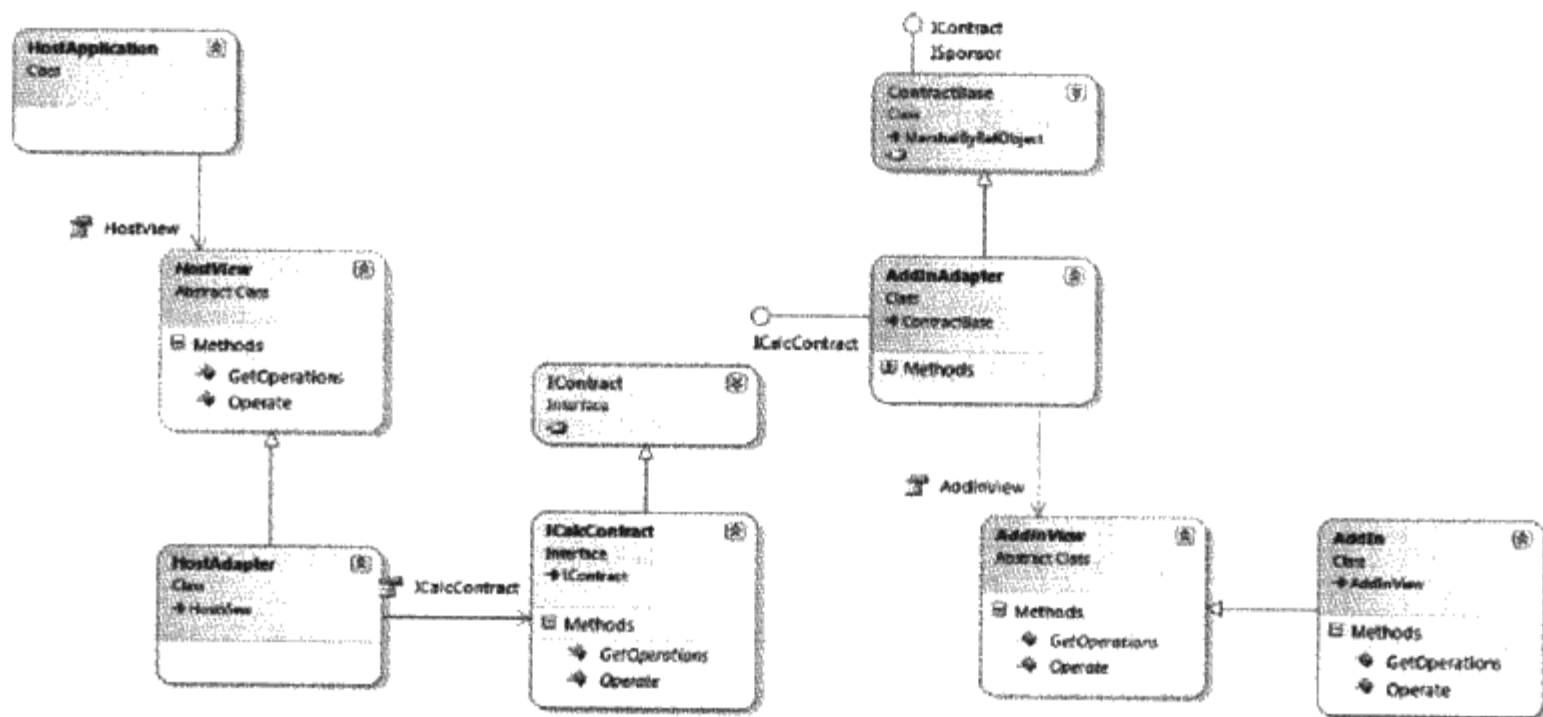


图 36-2

有了这个模型，插件端和主机端可以完全独立地升级了，只是需要使用映射层。例如，如果主机的一个新版本使用全新的方法和属性，合同就仍可以保持不变，只有适配器需要修改。也可以定义新的合同。适配器可以修改，也可以同时使用几个合同。

### 36.1.3 发现

如何为主机应用程序查找新插件？MAF 体系结构使用一个预定义的目录结构来查找插件和管道的其他程序集。管道的组成部分保存在这些子目录中：

- HostSideAdapters
- Contracts
- AddInSideAdapters
- AddInViews
- AddIns

除了 AddIns 目录之外，其他目录都直接包含管道特定部分的程序集。AddIns 目录为每个插件程序集包含一个子目录。插件也可以保存在完全独立于其他管道组件的目录中。



管道的程序集需要使用反射来动态加载，才能获得插件的所有信息。而且，对于许多插件而言，这还会增加主机应用程序的启动时间。因此，MAF 使用一个高速缓存，来保存管道组件的信息。该高速缓存是由安装插件的程序创建的，如果主机应用程序有管道目录的写入权限，该高速缓存就由主机应用程序创建。

给管道组件高速缓存的信息是调用 AddInStore 类的方法来创建的。Update()方法查找还没有列在保存文件中的新插件。Rebuild()方法用插件的信息重建完全二进制的保存文件。

表 36-2 列出了 AddInStore 类的成员。

表 36-2	
AddInStore 成员	说 明
Rebuild() RebuildAddIns()	Rebuild()方法为管道的所有组件重建高速缓存。如果插件存储在另一个目录下，就可以使用 RebuildAddIns()重建插件的高速缓存
Update() UpdateAddIns()	Rebuild()方法重建管道的完整高速缓存，Update()方法只用新管道组件更新高速缓存。 UpdateAddIns()方法只更新插件的高速缓存
FindAddIn() FindAddIns()	这些方法都使用高速缓存查找插件。FindAddIns()方法返回匹配主机视图的所有插件集合。 FindAddIn()方法返回一个特定的插件

36.1.4 激活和隔离

AddInStore 类的 FindAddIns()方法返回表示插件的 AddInToken 对象集合。使用 AddInToken 类可以访问插件的信息，例如名称、描述、发布者和版本。使用 Activate()方法可以激活插件。表 36-3 列出了 AddInToken 类的属性和方法。

表 36-3	
AddInToken 成员	说 明
Name、Publisher、 Version、Description	AddInToken 类的 Name、Publisher、Version 和 Description 属性返回用特性 AddInAttribute 赋予插件的信息
AssemblyName	AssemblyName 返回包含插件的程序集名称
EnableDirectConnect	使用 EnableDirectConnect 属性可以设置一个值，主机程序应使用该值直接连接到插件上，而不使用管道的组件。只有插件和主机程序运行在同一个应用程序域，插件视图和主机视图的类型相同时，才能使用这个属性。该属性仍要求管道的所有组件都存在
QualificationData	插件可以用特性 QualificationDataAttribute 标记应用程序域和安全需求。插件可以列出安全需求和隔离需求。例如，[QualificationData ("Isolation", "NewAppDomain")]表示插件必须保存在新进程中。可以从 AddInToken 中读取这些信息，激活有特定需求的插件。除了应用程序域和安全需求之外，还可以使用这个特性通过管道传送定制信息
Activate()	插件用 Activate()方法激活，利用这个方法的参数，可以定义插件是否加载到新应用程序域或新进程中。还可以定义插件获得的权限

一个插件可能使整个应用程序崩溃，例如 IE 可能因一个失败的插件而崩溃。根据应用程

序类型和插件的类型，可以让插件运行在另一个应用程序域或另一个进程中，来避免这个问题。MAF 给出了几个选项。可以在新应用程序域或新进程中激活插件。新应用程序域还可以有有限的权限。

AddInToken 类的 Activate()方法有几个重载版本，在这些版本中，可以传送加载插件的环境参数。表 36-4 列出了不同的选项。

表 36-4

AddInToken.Activate()的参数	说 明
AppDomain	可以传送一个加载插件的新应用程序域，这样可以使插件独立于主机应用程序，还可以从应用程序域中卸载插件
AddInSecurityLevel	如果插件应使用不同的安全级别来运行，就传送枚举 AddInSecurityLevel 的一个值，其值可以是 Internet、Intranet、FullTrust 和 Host
PermissionSet	如果预定义的安全级别不够安全，还可以给插件的应用程序域赋予 PermissionSet
AddInProcess	插件还可以运行在与主机应用程序不同的进程中。可以给 Activate()方法传送一个新的 AddInProcess。如果所有的插件都卸载了，新进程就可以退出，否则新进程就继续运行。这个选项可以用 KeepAlive 属性设置
AddInEnvironment	传送 AddInEnvironment 对象是定义加载插件的应用程序域的另一个选项。在 AddInEnvironment 的构造函数中，可以传送一个 AppDomain 对象。还可以用 AddInController 类的 AddInEnvironment 属性获得插件的已有 AddInEnvironment

提示：  
应用程序域详见第 17 章。

应用程序的类型也会限制可以使用的选项。WPF 插件目前不支持跨进程。Windows Forms 不能在不同的应用程序域之间连接 Windows 控件。

下面列出调用 AddInToken 的 Activate()方法时管道的执行步骤：

- (1) 用指定的权限创建应用程序域。
- (2) 用 Assembly.LoadFrom()方法把插件的程序集加载到新的应用程序域中。
- (3) 用反射调用插件的默认构造函数。因为插件派生于在插件视图中定义的基类，所以也加载了视图的程序集。
- (4) 接着构造插件端适配器的一个实例。插件的实例传送给适配器的构造函数，使适配器能连接合同和插件。插件适配器派生于基类 MarshalByRefObject，所以可以在应用程序域之间调用。
- (5) 激活代码给主机应用程序的应用程序域返回插件端适配器的一个代理。插件适配器实现了合同接口，所以该代理包含合同接口的方法和实现。
- (6) 主机端适配器的实例在主机应用程序的应用程序域中构造。插件端适配器的代理传送

给该构造函数。激活代码会从插件令牌中查找主机端适配器的类型。  
主机端适配器返回给主机应用程序。

36.1.5 合同

合同定义了主机端和插件端之间的界限。合同用一个接口来定义，该定义必须派生于基接口 `IServiceContract`。合同必须仔细考虑，因为它根据需要提供灵活的插件场景。

合同没有版本支持，不能改变，所以插件以前的实现代码仍可以在新的主机程序中运行。新版本应通过定义新合同来创建。

合同的类型有一些限制，其原因是版本问题，而且应用程序域要从主机应用程序跨越到插件上。类型必须是安全的，且支持版本，能在边界(应用程序域或跨进程)之间传送，也能在主机程序和插件之间传送。

可以用合同传送的类型可以是：

- 基本类型
- 其他合同
- 可串行化的系统类型
- 简单的可串行化定制类型，包括基本类型、合同，以及没有实现代码的类型

接口 `IServiceContract` 的成员如表 36-5 所示。

表 36-5

IServiceContract 的成员	说 明
<code>QueryContract()</code>	使用 <code>QueryContract()</code> 可以查询合同，验证是否也实现了另一个合同。插件可以支持几个合同
<code>RemoteToString()</code>	<code>QueryContract()</code> 的参数需要合同的字符串表示。 <code>RemoteToString()</code> 返回当前合同的字符串表示
<code>AcquireLifetimeToken()</code> <code>RevokeLifetimeToken()</code>	客户机调用 <code>AcquireLifetimeToken()</code> 来保存对合同的引用。 <code>AcquireLifetimeToken()</code> 会递增引用计数。 <code>RevokeLifetimeToken()</code> 递减引用计数
<code>RemoteEquals()</code>	<code>RemoteEquals()</code> 可用于比较两个合同引用

合同接口在 `System.AddIn.Contract`、`System.AddIn.Contract.Collections` 和 `System.AddIn.Contract.Automation` 命名空间中定义。表 36-6 列出了可以用于合同的合同接口。

表 36-6

合 同	说 明
<code>IListContract&lt;T&gt;</code>	<code>IListContract&lt;T&gt;</code> 可用于返回一个合同列表
<code>IEnumeratorContract&lt;T&gt;</code>	<code>IEnumeratorContract&lt;T&gt;</code> 用于枚举 <code>IListContract&lt;T&gt;</code> 的元素
<code>IServiceProviderContract</code>	一个插件可以为其他插件提供服务。提供服务的插件称为服务提供程序，它实现了接口 <code>IServiceProviderContract</code> 。通过 <code>QueryService()</code> 方法，可以查询实现该接口的插件提供了什么服务

(续表)

合 同	说 明
IProfferServiceContract	IProfferServiceContract 是服务提供程序和 IServiceProviderContract 提供的接口。 IProfferServiceContract 定义了方法 ProfferService() 和 Revoke Service()。 ProfferService() 给所提供的服务添加了一个 IServiceProvider Contract，而 RevokeService()删除它
INativeHandleContract	这个接口允许使用 GetHandle()方法访问内部的 Windows 句柄。这个合同由 WPF 主机程序用于使用 WPF 插件

36.1.6 生存期

插件需要加载多长时间？使用多少时间？何时可以卸载应用程序域？这有几个选项。一个选项是使用引用计数。每次使用插件都会递增引用计数。如果引用计数递减到 0，就可以卸载插件。另一个选项是使用垃圾回收器。如果垃圾回收器在运行，且没有对对象的引用，该对象就是垃圾回收器的目标。.NET Remoting 使用了租约机制，是使对象保持激活状态的承办者。只要租期到了，就询问承办者该对象是否应继续保持激活状态。

卸载插件还有一个特殊的问题，因为插件运行在不同的应用程序域、不同的进程中。但垃圾回收器不能跨进程工作。MAF 使用一个混合的模型来管理生存期。在单个应用程序域中，使用垃圾回收机制。在管道内部使用一个隐式的承办机制，但引用计数可用于从外部控制承办者。

下面考虑一种情况：插件加载到另一个应用程序域中。在主机应用程序中，当不再需要引用时，垃圾回收器清理了主机视图和主机端适配器。而在插件端，合同定义了方法 AcquireLifetimeToken()和 RevokeLifetimeToken()，来递增和递减承办者的引用计数。这两个版本不仅递增和递减一个值，还可以在某个团体频繁调用 RevokeLifetimeToken()方法时，提早释放对象。而 AcquireLifetimeToken()返回一个表示生存期令牌的标识符，这个标识符必须用于调用 RevokeLifetimeToken()方法。所以这两个方法总是成对调用。

通常不必处理 AcquireLifetimeToken()和 RevokeLifetimeToken()方法的调用，而可以使用 ContractHandle 类，在构造函数中调用 AcquireLifetimeToken()，在终结器中调用 RevokeLifetimeToken()。

提示：  
终结器详见第 12 章。

在插件加载到新应用程序域的情形中，当不再需要插件时，可以删除加载的代码。MAF 使用一个简单的模型把一个插件定义为应用程序域的拥有者，如果不再需要这个插件，就卸载应用程序域。如果在激活插件时创建了应用程序域，这个插件就是应用程序域的拥有者。如果应用程序域是以前创建的，就不会自动卸载。

ContractHandle 类在主机端适配器中用来增加插件的引用计数。这个类的成员如表 36-7 所示。

表 36-7

ContractHandle 成员	说 明
Contract	在 ContractHandle 类的构造过程中，可以指定一个实现了 IContract 的对象，来保存对它的引用。Contract 属性返回这个对象
Dispose()	Dispose()方法可以调用，而不是等待垃圾回收器执行清理操作，撤回生存期令牌
AppDomainOwner()	AppDomainOwner()是 ContractHandle 类的一个静态方法，如果插件拥有该方法传送的应用程序域，该方法就返回插件适配器
ContractOwnsAppDomain()	使用静态方法 ContractOwnsAppDomain()，可以验证指定的合同是否是应用程序域的拥有者。如果是，在删除合同时，会卸载应用程序域

36.1.7 版本问题

版本问题是插件的一个大问题。主机应用程序可以利用插件进一步开发。插件的一个要求是主机应用程序的新版本仍可以加载插件的旧版本。旧主机程序仍可以运行插件的新版本。那么，合同该如何修改呢？

System.AddIn 完全独立于主机应用程序和插件的实现，这是通过包含 7 部分的管道概念实现的。

36.2 插件示例

下面是一个主机应用程序的简单示例，它可以加载计算器插件。插件支持不同的计算操作。我们需要创建一个解决方案，它包含 6 个库项目和一个控制台应用程序。示例应用程序的项目如表 36-8 所示。这个表列出了需要引用的程序集。在解决方案中引用了其他项目后，还需要把 Copy Local 属性设置为 False，这样程序集就不会复制。但 HostApp 控制台项目例外，它需要对 HostView 项目的引用。这个程序集必须复制，才能在主机应用程序中找到。另外，还需要修改所生成的程序集的输出路径，使程序集复制到管道的正确目录下。

表 36-8

项 目	引 用	输 出 路 径	说 明
CalcContract	System.AddIn.Contract	..\Pipeline\ Contracts\	这个程序集包含与插件通信的合同。合同用接口定义
CalcView	System.AddIn	..\Pipeline\ AddInViews\	CalcView 程序集包含一个由插件引用的抽象类，这是合同的插件端
CalcAddIn	System.AddIn.CalcView	..\Pipeline\AddIns\ CalcAddIn\	CalcAddIn 是引用插件视图程序集的插件项目。这个程序集包含插件的实现代码



(续表)

项 目	引 用	输 出 路 径	说 明
CalcAddIn Adapter	System.AddIn System.AddIn.Contract CalcView CalcContract	..\Pipeline\ AddInSideAdapters\	CalcAddInAdapter 连接插件视图和 合同程序集，把合同映射到插件视 图上
HostView			包含主机视图的抽象类的程序集不 需要引用任何插件程序集，也没有解 决方案中的其他项目引用
HostAdapter	System.AddIn System.AddIn.Contract HostView CalcContract	..\Pipeline\ HostSideAdapters\	主机适配器把主机视图映射到合同 上。因此需要引用这些项目
HostApp	System.AddIn HostView		主机应用程序激活插件

36.2.1 计算器合同

下面实现合同程序集。合同程序集包含一个合同接口，该接口定义了在主程序程序和插件之间通信的协议。

下面的代码是为计算器示例应用程序定义的合同。应用程序给合同定义了方法 `GetOperations()` 和 `Operate()`。`GetOperations()` 方法返回计算器插件支持的一组数学操作。数学操作是由 `IOperationContract` 接口定义的，`IOperationContract` 接口本身就是一个合同，定义了只读属性 `Name` 和 `NumberOperands`。

`Operate()` 方法调用插件中的操作，它需要 `IOperation` 接口定义的一个操作和通过 `double` 数组提供的操作数。

有了这个接口，插件就支持需要任意多个 `double` 操作数的操作，且返回一个 `double`。  
属性 `AddInContract` 由 `AddInStore` 用于建立高速缓存。这个属性把类标记为插件合同接口。

```
using System.AddIn.Contract;
using System.AddIn.Pipeline;
namespace Wrox.ProCSharp.AddIns
{
    [AddInContract]
    public interface ICalculatorContract : IContract
    {
        IListContract < IOperationContract > GetOperations();
        double Operate(IOperationContract operation, double[] operands);
    }
    public interface IOperationContract : IContract
    {
        string Name { get; }
        int NumberOperands { get; }
    }
}
```

```

    }
}

```

### 36.2.2 计算器插件视图

插件视图重新定义了插件眼中的合同。该合同定义了接口 `ICalculatorContract` 和 `IOperationContract`。为此，插件视图定义了抽象类 `Calculator` 和具体的类 `Operation`。

在 `Operation` 中，没有每个插件都需要的特定实现代码，因为该类已经用插件视图程序集实现了。这个类用属性 `Name` 和 `NumberOperands` 描述了数学计算的一个操作。

抽象类 `Calculator` 定义了需要由插件实现的方法。虽然合同定义了需要在应用程序域和进程之间传送的参数和返回类型，但插件视图不需要。这里可以使用类型，以便于插件开发人员编写插件。`GetOperations()`方法返回 `IList<Operation>`，而不是 `IlistOperation<IOperationContract>`，这与合同程序集不同。

`AddInBase` 属性把类标识为插件视图，用于存储。

```

using System.AddIn.Pipeline;
using System.Collections.Generic;
namespace Wrox.ProCSharp.AddIns
{
    [AddInBase]
    public abstract class Calculator
    {
        public abstract IList < Operation > GetOperations();
        public abstract double Operate(Operation operation, double[] operand);
    }
    public class Operation
    {
        public string Name { get; set; }
        public int NumberOperands { get; set; }
    }
}

```

### 36.2.3 计算器插件适配器

插件适配器把合同映射到插件视图上。这个程序集引用了合同和插件视图程序集。适配器的实现代码需要把合同中的方法 `IListContract<IOperationContract> GetOperations()`映射到视图方法 `IList<Operation> GetOperations()`上。

该程序集包含类 `OperationViewToContractAddInAdapter` 和 `CalculatorViewToContractAddInAdapter`。这两个类实现了接口 `IOperationContract` 和 `ICalculatorContract`。基接口 `ICContract` 的方法可以通过派生于基类 `ContractBase` 来实现。这个基类提供了默认的实现代码。`OperationViewToContractAddInAdapter` 实现了 `IOperationContract` 接口的其他成员，并把调用传送给在构造函数中指定的 `Operation View`。

类 `OperationViewToContractAddInAdapter` 还包含静态帮助方法 `ViewToContractAdapter()`和 `ContractToViewAdapter()`，前者把 `Operation` 映射到 `IOperationContract` 上，后者把 `IOperationContract` 映射到 `Operation` 上。

```

using System.AddIn.Pipeline;
namespace Wrox.ProCSharp.AddIns
{

```

```

internal class OperationViewToContractAddInAdapter : ContractBase,
    IOperationContract
{
    private Operation view;
    public OperationViewToContractAddInAdapter(Operation view)
    {
        this.view = view;
    }
    public string Name
    {
        get { return view.Name; }
    }
    public int NumberOperands
    {
        get { return view.NumberOperands; }
    }
    public static IOperationContract ViewToContractAdapter(Operation view)
    {
        return new OperationViewToContractAddInAdapter(view);
    }
    public static Operation ContractToViewAdapter(
        IOperationContract contract)
    {
        return (contract as OperationViewToContractAddInAdapter).view;
    }
}

```

类 `CalculatorViewToContractAddInAdapter` 非常类似于 `OperationViewToContractAddInAdapter`：它派生自 `ContractBase`，继承了 `OperationContract` 接口的默认实现代码，还实现了一个合同接口。但 `ICalculatorContract` 接口是用 `GetOperations()` 和 `Operate()` 方法实现的。

适配器的 `Operate()` 方法调用视图类 `Calculator` 的 `Operate()` 方法，其中 `IOperationContract` 需要转换为 `Operation`。这是使用类 `OperationViewToContractAddInAdapter` 的静态帮助方法 `ContractViewToAdapter()` 完成的。

`GetOperations()` 方法的实现需要把集合 `ICollectionContract<IOperationContract>` 转换为 `ICollection<Operation>`。对于这个集合转换，类 `CollectionAdapters` 定义了转换方法 `ToICollection()` 和 `ToICollectionContract()`。其中 `ToICollectionContract()` 方法用于这个转换。

属性 `AddInAdapter` 把类标识为插件端适配器，用于插件的存储。

```

using System.AddIn.Contract;
using System.AddIn.Pipeline;
namespace Wrox.ProCSharp.AddIns
{
    [AddInAdapter]
    internal class CalculatorViewToContractAddInAdapter : ContractBase,
        ICalculatorContract
    {
        private Calculator view;
        public CalculatorViewToContractAddInAdapter(Calculator view)
        {
            this.view = view;
        }
        public ICollectionContract < IOperationContract > GetOperations()
        {
            return CollectionAdapters.ToICollectionContract < Operation,

```

```

        IOperationContract > (view.GetOperations(),
        OperationViewToContractAddInAdapter.ViewToContractAdapter,
        OperationViewToContractAddInAdapter.ContractToViewAdapter);
    }
    public double Operate(IOperationContract operation, double[] operands)
    {
        return view.Operate(
            OperationViewToContractAddInAdapter.ContractToViewAdapter(
                operation), operands);
    }
}
}

```

提示:

因为适配器类是由 .NET 反射功能调用的, 所以这些类可以使用内部的访问修饰符。这些类是要具体实现的, 所以最好使用 `internal` 访问修饰符。

### 36.2.4 计算器插件

插件现在包含具体的实现代码。它是用类 `CalculatorV1` 实现的。插件程序集依赖于插件视图程序集, 因为它需要实现抽象类 `Calculator`。

属性 `AddIn` 把类标记为插件, 用于插件的存储, 并添加了发布者、版本和描述信息。在主机端, 这些信息可以从 `AddInToken` 中访问。

`CalculatorV1` 在方法 `GetOperations()` 中返回一组支持的操作。`Operate()` 方法根据操作计算操作数。

```

using System;
using System.AddIn;
using System.Collections.Generic;
namespace Wrox.ProCSharp.AddIns
{
    [AddIn("CalculatorAddIn", Publisher="Wrox Press", Version="1.0.0.0",
        Description="Sample AddIn")]
    public class CalculatorV1 : Calculator
    {
        private List < Operation > operations;
        public CalculatorV1()
        {
            operations = new List < Operation > ();
            operations.Add(new Operation() { Name = "+", NumberOperands = 2 });
            operations.Add(new Operation() { Name = "-", NumberOperands = 2 });
            operations.Add(new Operation() { Name = "/", NumberOperands = 2 });
            operations.Add(new Operation() { Name = "*", NumberOperands = 2 });
        }
        public override IList < Operation > GetOperations()
        {
            return operations;
        }
        public override double Operate(Operation operation, double[] operand)
        {
            switch (operation.Name)
            {
                case "+":
                    return operand[0] + operand[1];
                case "-":
                    return operand[0] - operand[1];
            }
        }
    }
}

```

```

        case "/":
            return operand[0] / operand[1];
        case "*":
            return operand[0] * operand[1];
        default:
            throw new InvalidOperationException(
                String.Format("invalid operation {0}", operation.Name));
    }
}
}
}

```

### 36.2.5 计算器主机视图

下面看看主机端的主机视图。与插件视图类似，主机视图也定义了一个抽象类，其方法类似于合同。但是，这里定义的方法是由主机应用程序调用的。

类 **Calculator** 和 **Operation** 都是抽象的，因为其成员由主机适配器实现。它们只需要定义由主机应用程序使用的接口：

```

using System.Collections.Generic;
namespace Wrox.ProCSharp.AddIns
{
    public abstract class Calculator
    {
        public abstract IList < Operation > GetOperations();
        public abstract double Operate(Operation operation,
            params double[] operand);
    }
    public abstract class Operation
    {
        public abstract string Name { get; }
        public abstract int NumberOperands { get; }
    }
}

```

### 36.2.6 计算机主机适配器

主机适配器程序集引用了主机视图和合同，把视图映射到合同上。类 **OperationContractToViewHostAdapter** 实现了抽象类 **Operation** 的成员。**CalculatorContractToViewHostAdapter** 实现了抽象类 **Calculator** 的成员。

在 **OperationContractToViewHostAdapter** 中，在构造函数中指定了对合同的引用。适配器类还包含一个 **ContractHandle** 实例，它添加了对合同的生存期引用，所以只要主机应用程序需要，插件就保持加载状态。

```

using System.AddIn.Pipeline;
namespace Wrox.ProCSharp.AddIns
{
    internal class OperationContractToViewHostAdapter : Operation
    {
        private ContractHandle handle;
        public IOperationContract Contract { get; private set; }
        public OperationContractToViewHostAdapter(IOperationContract contract)
        {
            this.Contract = contract;
            handle = new ContractHandle(contract);
        }
    }
}

```



```

    }
    public override string Name
    {
        get
        {
            return Contract.Name;
        }
    }
    public override int NumberOperands
    {
        get
        {
            return Contract.NumberOperands;
        }
    }
    }
    internal static class OperationHostAdapters
    {
        internal static IOperationContract ViewToContractAdapter
            (Operation view)
        {
            return ((OperationContractToViewHostAdapter)view).Contract;
        }
        internal static Operation ContractToViewAdapter(
            IOperationContract contract)
        {
            return new OperationContractToViewHostAdapter(contract);
        }
    }
}

```

类 `CalculatorContractToViewHostAdapter` 实现了抽象主机视图类 `Calculator` 的成员，并把调用传送给合同。该类还有一个 `ContractHandle` 实例，它包含对合同的引用，这类似于插件端类型转换的适配器。但这次仅需要从插件适配器向主机端的类型转换。

属性 `HostAdapter` 把类标记为一个需要在 `HostSideAdapters` 目录下安装的适配器。

```

using System.Collections.Generic;
using System.AddIn.Pipeline;
namespace Wrox.ProCSharp.AddIns
{
    [HostAdapter]
    internal class CalculatorContractToViewHostAdapter : Calculator
    {
        private ICalculatorContract contract;
        private ContractHandle handle;
        public CalculatorContractToViewHostAdapter
            (ICalculatorContract contract)
        {
            this.contract = contract;
            handle = new ContractHandle(contract);
        }
        public override IList < Operation > GetOperations()
        {
            return CollectionAdapters.ToIList < IOperationContract, Operation > (
                contract.GetOperations(),
                OperationHostAdapters.ContractToViewAdapter,
                OperationHostAdapters.ViewToContractAdapter);
        }
        public override double Operate(Operation operation, double[] operands)

```

```
{
    return contract.Operate
        (OperationHostAdapters.ViewToContractAdapter(
            operation), operands);
}
```

36.2.7 计算器主机

示例主机应用程序使用了 WPF 技术。这个应用程序的用户界面如图 36-3 所示。其顶部是可用的插件列表。左边是活动插件的操作。选择要调用的操作时，就会显示操作数。输入了操作数的值后，就可以调用插件的操作。

底部的按钮用于重建和更新插件存储器，以及退出应用程序。

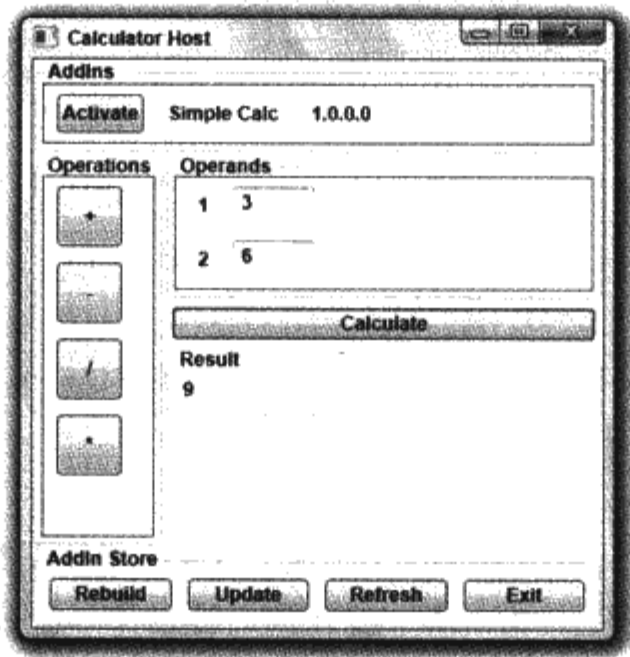


图 36-3

下面的 XAML 代码显示了用户界面的树形结构。在 ListBox 元素中，给项模板使用不同的样式，为插件列表、操作列表和操作数列表指定特定的表示方式。

```
< DockPanel >
  < GroupBox Header="AddIn Store" DockPanel.Dock="Bottom" >
    < UniformGrid Columns="4" >
      < Button x:Name="rebuildStore" Click="RebuildStore"
        Margin="5" > Rebuild < /Button >
      < Button x:Name="updateStore" Click="UpdateStore"
        Margin="5" > Update < /Button >
      < Button x:Name="refresh" Click="RefreshAddIns"
        Margin="5" > Refresh < /Button >
      < Button x:Name="exit" Click="App_Exit" Margin="5" > Exit < /Button >
    < /UniformGrid >
  < /GroupBox >
  < GroupBox Header="AddIns" DockPanel.Dock="Top" >
    < ListBox x:Name="listAddIns" ItemsSource="{Binding}"
      Style="{StaticResource listAddInsStyle}" / >
  < /GroupBox >
  < GroupBox DockPanel.Dock="Left" Header="Operations" >
    < ListBox x:Name="listOperations" ItemsSource="{Binding}"
      Style="{StaticResource listOperationsStyle}" / >
  < /GroupBox >
< /DockPanel >
```

```

< /GroupBox >
< StackPanel DockPanel.Dock="Right" Orientation="Vertical" >
    < GroupBox Header="Operands" >
        < ListBox x:Name="listOperands" ItemsSource="{Binding}"
            Style="{StaticResource listOperandsStyle}" >
        < /ListBox >
    < /GroupBox >
    < Button x:Name="buttonCalculate" Click="Calculate" IsEnabled="False"
        Margin="5" > Calculate < /Button >
    < GroupBox DockPanel.Dock="Bottom" Header="Result" >
        < Label x:Name="labelResult" / >
    < /GroupBox >
< /StackPanel >
< /DockPanel >

```

**提示:**

项模板的内容可参见第 35 章。

在后台代码中，FindAddIns()方法在 Window 的构造函数中调用。FindAddIns()方法使用 AddInStore 类获得 AddInToken 对象的集合，把它们传送给列表框 listAddIns 的 DataContext 属性，以显示它们。AddInStore.FindAddIns()方法的第一个参数传送主机视图定义的抽象类 Calculator，从存储器中查找应用于合同的所有插件。第二个参数传送从应用程序配置文件中读取的管道目录。运行 Wrox 下载网站上的示例应用程序时，需要修改应用程序配置文件中的目录，以匹配自己的目录结构。

```

using System;
using System.AddIn.Hosting;
using System.AddIn.Pipeline;
using System.IO;
using System.Linq;
using System.Windows;
using System.Windows.Controls;
using Wrox.ProCSharp.AddIns.Properties;
namespace Wrox.ProCSharp.AddIns
{
    public partial class CalculatorHostWindow : Window
    {
        private Calculator activeAddIn = null;
        private Operation currentOperation = null;
        public CalculatorHostWindow()
        {
            InitializeComponent();
            FindAddIns();
        }
        void FindAddIns()
        {
            try
            {
                this.listAddIns.DataContext =
                    AddInStore.FindAddIns(typeof(Calculator),
                        Settings.Default.PipelinePath);
            }
            catch (DirectoryNotFoundException ex)
            {
                MessageBox.Show("Verify the pipeline directory in the " +
                    "config file");
                Application.Current.Shutdown();
            }
        }
    }
}

```

```

    }
}
//...

```

要更新 Add-In 存储器的高速缓存，UpdateStore()和 RebuildStore()方法应映射到 Update 和 Rebuild 按钮的 Click 事件上。在这些方法的实现代码中，使用了 AddInStore 类的 Update()和 Rebuild()方法。如果程序集存储在错误的目录下，这些方法就返回一个警告字符串数组。由于管道结构比较复杂，第一次把程序集复制到正确的目录下时，其项目配置不太可能完全正确。阅读这些方法返回的信息，可以清楚地了解错在何处。例如，信息“在程序集 \Pipeline\AddInSideAdapters\CalcView.dll 中没有找到可用的 AddInAdapter 部分”表示，程序集 CalcView 存储在错误的目录下。

```

private void UpdateStore(object sender, RoutedEventArgs e)
{
    string[] messages = AddInStore.Update(Settings.Default.PipelinePath);
    if (messages.Length != 0)
    {
        MessageBox.Show(string.Join("\n", messages),
            "AddInStore Warnings", MessageBoxButton.OK,
            MessageBoxImage.Warning);
    }
}
private void RebuildStore(object sender, RoutedEventArgs e)
{
    string[] messages =
        AddInStore.Rebuild(Settings.Default.PipelinePath);
    if (messages.Length != 0)
    {
        MessageBox.Show(string.Join("\n", messages),
            "AddInStore Warnings", MessageBoxButton.OK,
            MessageBoxImage.Warning);
    }
}

```

在图 36-2 中，可用插件的旁边有一个 Activate 按钮。单击这个按钮会调用处理方法 ActivateAddIn()。在这个方法的代码中，使用 AddInToken 类的 Activate()方法激活插件。其中插件加载到用 AddInProcess 类创建的一个新进程中。AddInProcess 类启动了进程 AddInProcess32.exe。把该进程的 KeepAlive 属性设置为 false，则只要最后一个插件引用被垃圾回收了，该进程就停止。参数 AddInSecurityLevel.Internet 使插件在有限的权限下运行。ActivateAddIn()的最后一个语句调用 ListOperations()方法，ListOperations()方法又调用插件的 GetOperations()方法。GetOperations()方法把返回的列表赋予列表框 listOperations 的 DataContext 属性，显示所有的操作。

```

private void ActivateAddIn(object sender, RoutedEventArgs e)
{
    FrameworkElement el = sender as FrameworkElement;
    Trace.Assert(el != null, "ActivateAddIn invoked from the wrong " +
        "control type");

    AddInToken addIn = el.Tag as AddInToken;
    Trace.Assert(el.Tag != null, String.Format(
        "An AddInToken must be assigned to the Tag property " +
        "of the control {0}", el.Name));
}

```

```

AddInProcess process = new AddInProcess();
process.KeepAlive = false;

activeAddIn = addIn.Activate < Calculator > (process,
    AddInSecurityLevel.Internet);
ListOperations();
}
void ListOperations()
{
    this.listOperations.DataContext = activeAddIn.GetOperations();
}

```

激活插件，在 UI 上显示操作列表后，用户就可以选择操作了。**Operations** 类别中按钮的 Click 事件赋予处理程序方法 **OperationSelected()**。在这个方法的代码中，检索赋予了按钮的 Tag 属性的 **Operation** 对象，获得操作所需要的操作数个数。为了让用户给操作数添加值，把一个 **OperandUI** 对象数组绑定到列表框 **listOperands** 上。

```

private void OperationSelected(object sender, RoutedEventArgs e)
{
    FrameworkElement el = sender as FrameworkElement;
    Trace.Assert(el != null, "OperationSelected invoked from " +
        "the wrong control type");
    Operation op = el.Tag as Operation;
    Trace.Assert(el.Tag != null, String.Format(
        "An AddInToken must be assigned to the Tag property " +
        "of the control {0}", el.Name));
    currentOperation = op;
    ListOperands(new double[op.NumberOperands]);
}
private class OperandUI
{
    public int Index { get; set; }
    public double Value { get; set; }
}
void ListOperands(double[] operands)
{
    this.listOperands.DataContext =
        operands.Select((operand, index) =>
            new OperandUI()
            { Index = index + 1, Value = operand }).ToArray();
}

```

在 **Calculate** 按钮的 Click 事件中调用了 **Calculate()** 方法。这里操作数从 UI 中提取，把操作和操作数传送给插件的 **Operate()** 方法，结果显示为标签的内容：

```

private void Calculate(object sender, RoutedEventArgs e)
{
    OperandUI[] operandsUI = (OperandUI[])this.listOperands.DataContext;
    double[] operands = operandsUI.Select(opui => opui.Value).ToArray();
    labelResult.Content = activeAddIn.Operate(currentOperation,
        operands);
}

```

### 36.2.8 其他插件

艰苦的工作已经完成了。管道组件和主机应用程序都创建好了。管道现在可以工作了，还可以在主机应用程序中添加其他插件，例如下面的 **Advanced Calculator** 插件：



```
[AddIn("Advanced Calc", Publisher = "Wrox Press", Version = "1.1.0.0",
    Description = "Another AddIn Sample")]
public class AdvancedCalculatorV1 : Calculator
```

36.3 小结

本章学习了.NET 3.5 技术的一个新概念：Managed AddIn Framework。MAF 使用管道概念使主机程序集和插件程序集的创建完全独立。清楚定义的合同隔离开了主机视图和插件视图。适配器可以使主机端和插件端独立地修改。

下一章是介绍用 ASP.NET 开发 UI 的 3 章中的第一章。

# 第 37 章

## ASP.NET 页面

如果您是 C# 和 .NET 的新手，肯定不理解为什么本书要包含介绍 ASP.NET 的内容。这是一种全新的语言，对吗？但实际上并非如此。使用 C# 可以创建 ASP.NET 页面。

ASP.NET 是 .NET Framework 的一部分。在通过 HTTP 请求建立文档时，它可以在 Web 服务器上动态创建文档。该文档主要是 HTML 和 XHTML 文档，但也可以创建 XML 文档、CSS 文件、图像、PDF 文档，或者支持 MIME 类型的文档。

在某些方面，ASP.NET 类似于许多其他技术，例如 PHP、ASP、ColdFusion 等，但它们有一个重要的区别。顾名思义，ASP.NET 可以与 .NET Framework 完全集成，它包含了对 C# 的支持。

您可能使用过动态生成内容的 ASP 技术。这种技术使用脚本语言，例如 VBScript 或 JScript 来编程，结果却不是很好。但对于那些习惯于“正确的”已编译编程语言的人来说，这种技术很笨拙，肯定会导致性能的损失。

与更高级的编程语言相比，一个主要区别是 ASP.NET 提供了完整的服务器端对象模型，可以在运行期间使用。ASP.NET 可以在其环境中把页面上的所有控件作为对象来访问。在服务器端，还可以访问其他 .NET 类，与许多有用的服务集成起来。在页面上使用的控件有许多功能，实际上可以完成 Windows Forms 类的几乎所有的功能，有非常大的灵活性。因此，生成 HTML 内容的 ASP.NET 通常称为 Web 窗体。

本章将详细介绍 ASP.NET，包括 ASP.NET 如何工作，ASP.NET 可以完成什么任务，以及什么地方适合使用 C#。下面是本章的主要内容：

- ASP.NET 简介
- 如何使用服务器控件创建 ASP.NET Web 窗体
- 如何使用 ADO.NET 把数据绑定到 ASP.NET 控件上
- 应用程序配置

### 37.1 ASP.NET 概述

ASP.NET 使用 Internet Information Server(IIS)来传送内容，以响应 HTTP 请求。ASP.NET 页面在 .aspx 文件中，其基本结构如图 37-1 所示。

在 ASP.NET 处理过程中，可以访问所有的 .NET 类、C# 或其他语言创建的定制组件、数

数据库等。实际上，这与运行 C#应用程序一样；在 ASP.NET 中使用 C#就是在运行 C#程序。

ASP.NET 文件可以包含下述内容：

- 服务器的处理指令
- C#、VB.NET、JScript.NET 代码或.NET Framework 支持的其他语言的代码
- 对应已生成资源的窗体内容，例如 HTML
- 客户端的脚本代码
- 内嵌的 ASP.NET 服务器控件

实际上，ASP.NET 文件也可以很简单，如下所示。

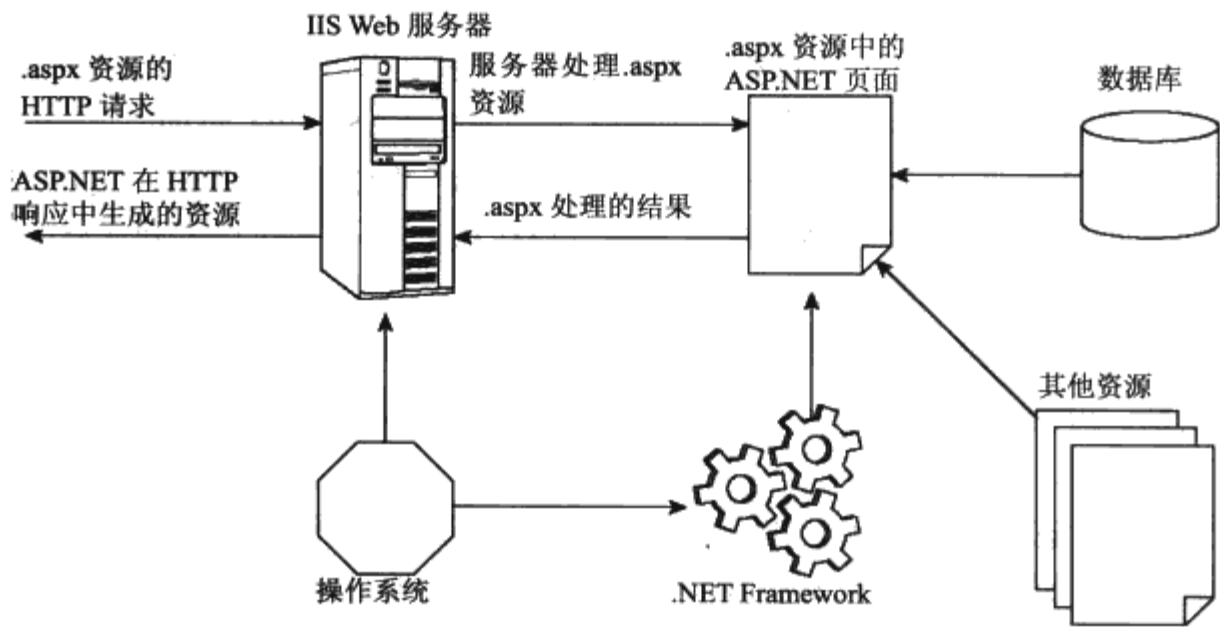


图 37-1

Hello!

结果很简单，返回一个只包含这个文本的 HTML 页面(因为 ASP.NET 页面的默认输出是 HTML)。

本章后面会提到，也可以把代码的某些部分分解为其他文件，这可以提供更合理的结构。

### ASP.NET 中的状态管理

ASP.NET 页面的一个重要属性是它们是无状态的。在默认情况下，在用户的请求之间，并没有信息存储在服务器上(但有一些方法可以完成存储信息的任务，详见下面的内容)。这初看起来有点奇怪，因为状态管理对于用户友好的交互会话是非常重要的。但是，ASP.NET 提供了一种相当好的方式来解决这个问题，使会话管理几乎是完全透明的。

简言之，Web 窗体上控件的状态信息(包括文本框中输入的数据、下拉列表中的选项等)存储在隐藏的 viewstate 字段中，这个字段是服务器生成的页面的一部分，并传送给用户。后续的操作称为回送(postback)，例如触发需要服务器端处理的事件，提交窗体数据，把这些信息传送回服务器。在服务器上，这些信息用于重新填充页面对象模型，以便操作它，就像在本地进行修改一样。

稍后详细介绍这个主题。

## 37.2 ASP.NET Web 窗体

如前所述, ASP.NET 中的许多功能是使用 Web 窗体实现的。稍后我们将创建一个简单的 Web 窗体, 深入介绍这种技术。但这里先简要介绍 Web 窗体的设计。注意许多 ASP.NET 开发人员仅使用文本编辑器(例如 Notepad)创建文件。这里不推荐这么做, 因为 Visual Studio 或 Web Developer Express 等 IDE 提供的优点是很重要的, 只是使用 Notepad 等文本编辑器是创建文件的一种方法, 所以这里提及它。如果使用文本编辑器, 在把 Web 应用程序的哪些部分放在什么地方等方面有非常大的灵活性, 例如, 可以把所有代码都组合到一个文件中。把代码放在<script>标记中, 在起始<script>标记中使用两个属性, 如下所示:

```
<script language="c#" runat="server">
    // Server-side code goes here.
</script>
```

这里的 runat="server"属性是很重要的, 因为它指示 ASP.NET 引擎在服务器上执行这段代码, 而不是把它传送给客户, 因此可以访问前面讨论的环境。我们可以在服务器端脚本块中放置函数、事件处理程序等。

如果省略 runat="server"属性, 就是在提供客户端代码, 如果使用本章后面要介绍的服务器端编码方式, 就会失败。但是, 可以使用<script>元素提供 JavaScript 等语言编写的客户端脚本。例如:

```
<script language="JavaScript" type="text/JavaScript">
    // Client-side code goes here; we can also use "vbscript".
</script>
```

**注意:**

type 属性是可选的, 但如果需要兼容 XHTML, 它就是必需的。

在页面中添加 JavaScript 代码的功能也包含在 ASP.NET 中, 这好像有点奇怪。但是, JavaScript 允许给 Web 页面添加动态的客户端操作, 这是非常有用的。Ajax 编程就允许添加 JavaScript 代码, 详见第 39 章。

可以在 Visual Studio 中创建 ASP.NET 文件。这是非常重要的, 因为我们已经熟悉了在这个环境中进行 C#编程。在这个环境中, Web 应用程序的默认项目设置提供了一种比单个.aspx 文件略微复杂的结构, 使之更富于逻辑(更接近编程, 而不像 Web 开发)。据此, 本章将使用 Visual Studio 进行 ASP.NET 编程(而不是 Notepad)。

.aspx 文件也可以包含括在<%和%>标记中的代码块。但是, 函数定义和变量声明不能放在这里。可以插入代码, 当执行到块时就执行这些代码。当输出简单的 HTML 内容时, 这是很有效的。这种方式类似于旧风格的 ASP 页面, 但有一个重要的区别: 代码是已经编译好的, 不是解释性的。这样, 性能会好得多。

下面举一个示例。要创建一个新的 Web 应用程序, 应在 Visual Studio 中, 使用 File | New | Web Site 菜单项, 打开一个对话框。在对话框中选择 Visual C#语言类型和 ASP.NET Web Site 模板, 现在要进行选择: Visual Studio 可以在几个不同的位置创建 Web 站点:

- 本地 IIS Web 服务器上

- 本地磁盘上，它配置为使用内置的 Visual Web Developer Web 服务器
- 可通过 FTP 访问的任意位置
- 支持 Front Page Server Extensions 的远程 Web 服务器上

不必考虑后两个选项，它们使用远程服务器，所以现在应选择前两项。一般情况下，IIS 是安装 ASP.NET Web 站点的最佳位置，因为它最接近部署 Web 站点时需要的配置。另一个选项使用内置的 Web 服务器，适合于测试，但有一些限制：

- 只有本地计算机能访问 Web 站点
- 访问 SMTP 等服务受到限制
- 安全模型与 IIS 不同：应用程序运行在当前用户的账户下，而不是运行在 ASP.NET 的特定账户下

最后一点需要澄清，因为在访问数据库或其他需要验证身份的数据时，安全性是非常重要的。在默认情况下，运行在 IIS 上的 Web 应用程序会在 Windows XP、2000 和 Vista Web 服务器的 ASPNET 账户下运行，或在 Windows Server 2003 的 NETWORK SERVICES 账户下运行。如果使用 IIS，这是可以配置的，但如果使用内置的 Web 服务器，就不能配置它。

为了便于演示，或者计算机上可能没有安装 IIS，则可以使用内置的 Web 服务器。在这个阶段不必担心安全性，只需选择它即可。

在 C:\ProCSharp\Chapter37 目录下使用 File System 选项创建一个新的 ASP.NET Web Site，称为 PCSWebApp1。如图 37-2 所示。

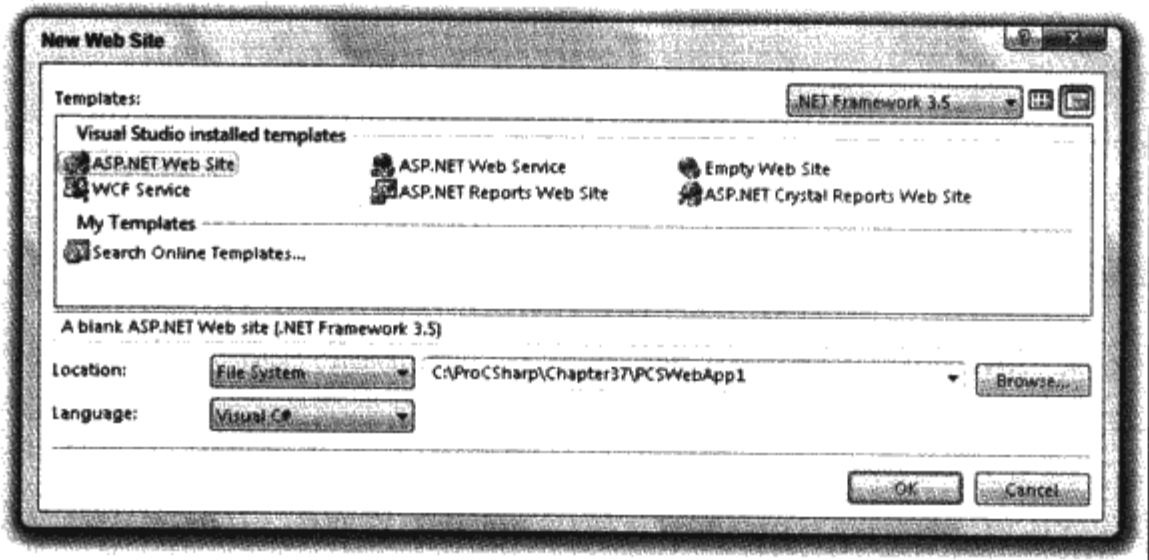


图 37-2

稍后，Visual Studio 应建立如下内容：

- 新的解决方案 PCSWebApp1，包含 C# Web 应用程序 PCSWebApp1
- 保留文件夹 App\_Data，包含数据文件，例如 XML 文件或数据库文件
- Default.aspx，Web 应用程序中的第一个 ASP.NET 页面
- Default.aspx.cs，Default.aspx 的后台代码类文件
- Web.config，Web 应用程序的配置文件

这些都可以在 Solution Explorer 中看到，如图 37-3 所示。



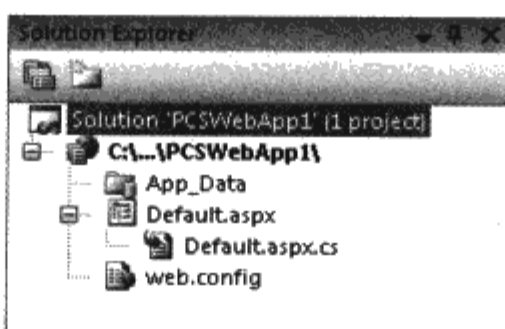


图 37-3

可以在设计视图或源代码(HTML)视图中查看.aspx 文件。这与 Windows 窗体(参见第 31 章)完全相同。Visual Studio 中的起始视图是 Default.aspx 的设计或源代码视图(使用左下角的按钮可以切换视图)。设计视图如图 37-4 所示。

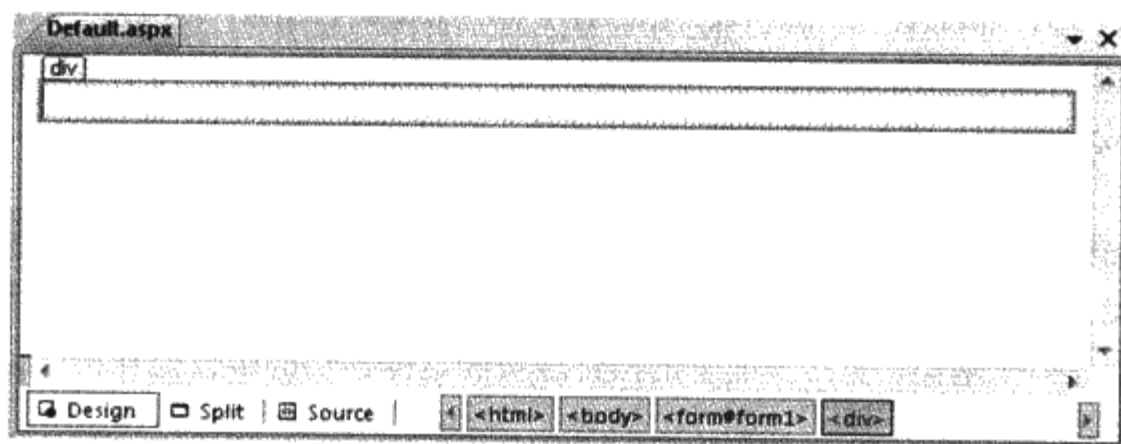


图 37-4

在窗体(当前为空)的下面,可以看到在窗体的 HTML 中光标当前的位置。这里光标在<form>元素的<div>元素中,<form>元素在页面的<body>元素的,显示为<form#form1>,用它的 id 属性表示。<div>元素也显示在设计视图中。

页面的源代码视图显示了在.aspx 文件中生成的代码:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs"
    Inherits="Default" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Untitled Page</title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
        </div>
    </form>
</body>
</html>
```

如果读者熟悉 HTML 语法,就会觉得这些代码很眼熟。这里列出了 HTML 页面中遵循 XHTML 模式的基本代码,并包含几行额外的代码。最重要的元素是<form>,它的 id 属性是 form1,包含了 ASP.NET 代码。这里最重要的属性是 runat。与本节前面的服务器端代码块一样,这个属性设置为 server,表示窗体的处理将在服务器上进行。如果没有包含这个属性,就不会

在服务器端上完成任何处理，窗体也不会执行任何操作。在 ASP.NET 页面中，只有一个服务器端<form>元素。

这段代码中另一个比较重要的东西是顶部的<@% Page %>标记，它定义了对于 C# Web 应用程序开发人员来说非常重要的页面特性。首先，language 属性指定在页面中使用 C#语言，与前面的<script>块一样(Web 应用程序默认的语言是 VB，使用 Web.config 配置文件可以修改这个属性)。下面的三个属性 AutoEventWireup、CodeFile 和 Inherits 用于把 Web 窗体关联到后台代码文件中的一个类上，这里是 Default.aspx.cs 文件中的部分类\_Default。这就需要讨论 ASP.NET 代码模型了。

### 37.2.1 ASP.NET 代码模型

在 ASP.NET 中，布局(HTML)代码、ASP.NET 控件和 C#代码用于生成用户看到的 HTML。布局和 ASP.NET 代码存储在.aspx 文件中，也就是上一节的.aspx 文件。用于定制窗体操作的 C#代码包含在.aspx 文件中，也可以像前面的例子那样，放在单独的.aspx.cs 文件中，通常称为后台编码文件。

在处理 ASP.NET Web 窗体时，一般在用户请求页面时，预编译站点，此时会发生几个事件：

- ASP.NET 处理器执行页面，确定必须创建什么对象，以实例化页面对象模型。
- 动态创建一个基类，包括页面上的控件成员和这些控件的事件处理程序(例如按钮单击事件)。
- 包含在.aspx 页面中的其他代码，与这个基类合并，构成完整的对象模型
- 编译所有的代码，并高速缓存起来，以备处理以后的请求
- 生成 HTML，返回给用户

在 Web 站点 PCSWebApp1 中，为 Default.aspx 生成的后台代码文件的内容最初非常少。首先看看需要在 Web 页面上使用的默认命名空间引用的集合：

```
using System;
using System.Data;
using System.Configuration;
using System.Linq;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;

using System.Xml.Linq;
```

在这些引用的下面，Default.aspx 部分类的定义几乎是空的：

```
public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
    }
}
```

这里可以使用 `Page_Load()` 事件处理程序添加加载页面时需要的代码。在添加事件处理程序时，这个类文件会包含越来越多的代码。注意没有把这个事件处理程序关联到页面上的代码，这是由 ASP.NET 运行库处理的。这要归功于 `AutoEventWireup` 属性，把它设置为 `false`，表示必须自己在代码中把事件处理程序与事件关联起来。

这个类是一个部分类定义，因为前面介绍的过程需要它。在预编译页面时，会从页面的 ASP.NET 代码中创建一个单独的部分类定义，这包括添加到页面上的所有控件。在设计期间，编译器会推断这个部分类定义，以便在后台代码中使用 `IntelliSense`，来引用页面上的控件。

### 37.2.2 ASP.NET 服务器控件

前面生成的代码并不能完成许多工作，所以下面就应添加一些内容。在 Visual Studio 中使用 Web 窗体设计器，它支持拖放操作，其方式与 Windows 窗体设计器相同。

可以添加到 ASP.NET 页面上的控件有 3 种类型：

- HTML 服务器控件——这些控件模拟 HTML 元素，HTML 开发人员会很熟悉它们。
- Web 服务器控件——这是一组新的控件，其中一些控件的功能与 HTML 控件相同，但它们的属性和其他元素有一个公共的命名模式，便于进行开发，而且可以与相似的 Windows 窗体控件保持一致。还有一些全新的、非常强大的控件，如本章后面所述。Web 服务器控件有几种类型，包括标准控件，如按钮、验证用户输入的验证控件、简化用户管理的登录控件，和处理数据源的一些较复杂的控件。
- 定制控件和用户控件——由开发人员定义的控件，我们可以用第 38 章介绍的许多方式来定义它们。

#### 提示：

下一节列出了常用 Web 服务器控件及其使用说明的完整列表。下一章将介绍其他控件。本章没有介绍 HTML 控件。这些控件提供的功能，Web 服务器也能提供，而且 Web 服务器控件为熟悉编程的开发人员提供了一个功能比 HTML 更丰富的环境。学会如何使用 Web 服务器控件后，使用 HTML 服务器控件就不难了。也可以参阅清华大学出版社出版的《ASP.NET 2.0 高级编程》。

下面在上一节创建的 Web 站点 `PCSWebApp1` 中，添加两个 Web 服务器控件。所有的 Web 服务器控件都以下述 XML 元素的方式使用：

```
<asp:controlName runat="server" attribute="value">Contents</asp:controlName>
```

其中 `controlName` 是 ASP.NET 服务器控件的名称，`attribute="value"` 是一个或多个属性规范，`Contents` 指定控件的内容。一些控件可以使用属性和控件元素的内容来设置属性，例如 `Label` (用于显示简单文本)，其文本可以用两种方式指定。其他控件可以使用元素包含模式来定义它们的层次结构，例如 `Table` (定义一个表) 可以包含 `TableRow` 元素，指定表中的行。

注意，控件的语法是基于 XML 的(它们也可以内嵌在非 XML 代码中，例如 HTML)。省略闭合标记、表示空元素的 `/>`，或者重叠控件，都会产生错误。

最后，再看看 Web 服务器控件上的 `runat="server"` 属性。把它放在这里和放在其他地方是一

样的，遗漏这个属性也会产生错误，结果将是一个不能运行的 Web 窗体。

第一个示例应简单一些。修改 Default.aspx 的 HTML 设计视图，代码如下。

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs"
Inherits="_Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Untitled Page</title>
</head>
<body>
  <form id="form1" runat="server">
    <div>
      <asp:Label runat="server" ID="resultLabel" /><br />
      <asp:Button runat="server" ID="triggerButton" Text="Click Me" />
    </div>
  </form>
</body>
</html>
```

这里添加了两个 Web 窗体控件：标签和按钮。

#### 注意：

在添加控件时，Visual Studio 的 IntelliSense 会提示代码输入项，这与 C# 代码编辑器一样。如果在隔开的视图中编辑代码，再同步视图，在源代码面板上编辑的元素会在设计面板上突出显示。

回过头来看看设计屏幕，其中已经添加了控件，并用它们的 ID 属性命名(ID 属性常常称为控件的标识符)。与 Windows 窗体一样，可以通过 Properties 窗口访问所有的属性、事件等，如果进行了修改，代码或设计会通过 Properties 窗口立即反馈回来。

#### 注意：

也可以使用 CSS 属性窗口和其他样式窗口，给控件指定样式。但除非很熟悉 CSS，否则现在不要使用这个技术，而是应关注控件的功能。

我们添加的所有服务器控件都会自动成为对象模型的一部分，该对象模型是在这段后置代码中为窗体构建的。Windows 窗体开发人员可以即时得到这个对象模型，并开始认识到它与 Windows 窗体的类似性。

要让这个应用程序完成一些工作，应添加单击按钮的事件处理程序。可以在 Properties 窗口中为按钮输入一个方法名，也可以双击该按钮，得到默认的事件处理程序。如果双击按钮，就可以自动添加一个事件处理方法：

```
protected void triggerButton_Click(object sender, EventArgs e)
{
}
}
```

把一些代码添加到 Default.aspx 中，就可以把事件处理程序链接到按钮上：

```

<div>
  <asp:Label Runat="server" ID="resultLabel" /><br />
  <asp:Button Runat="server" ID="triggerButton" Text="Click Me"
    OnClick="triggerButton_Click" />
</div>

```

其中 OnClick 属性告诉 ASP.NET 运行库，在生成窗体的代码模型时，把按钮的单击事件包装到 triggerButton\_Click 方法中。

修改 triggerButton\_Click() 中的代码（注意标签控件类型是从 ASP.NET 代码中推断出来的，所以可以直接在后台代码中使用）：

```

void triggerButton_Click(object sender, EventArgs e)
{
    resultLabel.Text = "Button clicked!";
}

```

下面准备运行它。不需要建立项目，只需保存所有的内容，把 Web 浏览器指向 Web 站点的地址。如果使用 IIS，这就很简单，因为我们知道指向的 URL。但本例使用内置的 Web 服务器，所以需要启动运行。最快捷的方式是按下 Ctrl+F5，启动服务器，打开一个浏览器，并指向指定的 URL。

在运行内置的 Web 服务器时，系统栏中会显示一个图标。双击这个图标，会看到 Web 服务器执行的过程，并可以在需要时停止它，如图 37-5 所示。

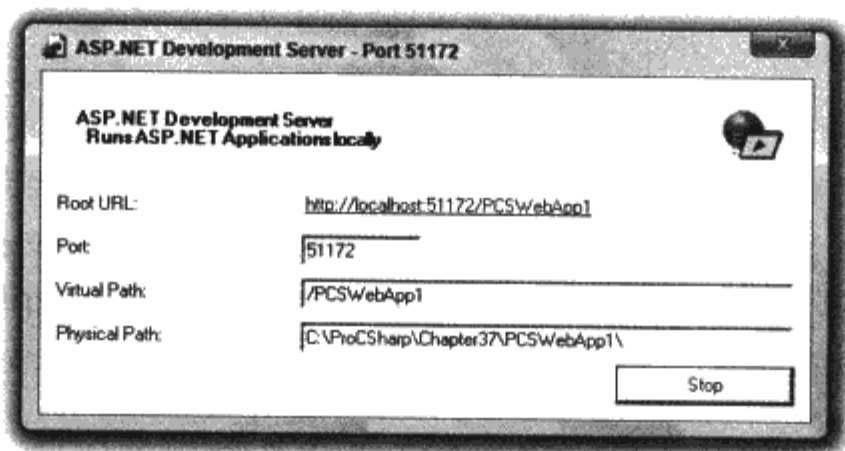


图 37-5

在图 37-5 中，可以看到 Web 服务器运行的端口和创建 Web 站点的 URL。

打开的浏览器应显示 Web 页面上的 Click Me 按钮。在单击这个按钮前，使用 Page | View Source (在 IE7 中)快速查看一下浏览器接收到的代码。<form>部分应如下所示：

```

<form method="post" action="Default.aspx" id="form1">
  <div>
    <input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
      value="/wEPDwUKLTE2MjY5MTY1NWRkQw+7xydPDuBqgjPjjMHnYk872ZE=" />
    </div>
    <div>
      <span id="resultLabel"></span><br />
      <input type="submit" name="triggerButton" value="Click Me"
        id="triggerButton" />
    </div>
    <div>
      <input type="hidden" name="__EVENTVALIDATION" id="__EVENTVALIDATION"
        value="/wEWAqK39qTFBwLHpP+yC4rCCl22/GGMAFwD017nokvyFZ8Q" />
    </div>
  </form>

```



```
< /div >
</form>
```

Web 服务器控件生成了 HTML，`<span>`和`<input>`分别代表`<asp:Label>`和`<asp:Button>`。还有一个名为 VIEWSTATE 的`<input type="hidden">`字段，把前面提到的窗体状态封装起来。在窗体传送回服务器以重新创建 UI，以及跟踪改变时使用这些信息。注意`<form>`元素已经进行了配置，通过 HTTP POST 操作(在 `method` 中指定)把数据传送回 `Default.aspx`(在 `action` 中指定)，它还被赋予了一个名称 `form1`。

在单击按钮，查看文本后，可再次浏览源 HTML(下面添加了必要的空格，使代码比较清晰)：

```
<form method="post" action="Default.aspx" id="form1">
  <div>
    <input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
      value="/wEPDwUKLTE2MjY5MTY1NQ9kFgICA9kFgICAQ8PFgIeBFRleHQFD0JldHRvbiB
        jBGlja2VkIWRkZnN3zQXZqDnF2ddEBw4Kj7MEqj9pJ" />
    </div>
    <div>
      <span id="resultLabel">Button clicked!</span><br />
      <input type="submit" name="triggerButton" value="Click Me"
        id="triggerButton" />
    </div>
    <div>
      <input type="hidden" name="__EVENTVALIDATION" id="__EVENTVALIDATION"
        value="/wEWAgl13MPHAWLHpP+yCwFffz4kBL6+KP1xjB0lgrAageely" />
    </div>
  </form>
```

这次 viewstate 的值包含比较多的信息，因为 HTML 的结果不仅仅取决于 ASP.NET 页面的默认输出。在复杂的窗体上，这可能是一个非常长的字符串，但这是由系统在后台完成的，我们几乎可以不考虑状态管理，只要回送过程之间保存字段值即可。在 Viewstate 字符串过长时，可以禁用不需要保留状态信息的控件的 Viewstate。也可以禁用整个页面的 Viewstate。如果页面不需要在回送过程中保留状态，以提高性能，就可以禁用整个页面的 Viewstate。

**注意：**

Viewstate 详见第 38 章。

为了说明不必手工进行任何编译，把 `Default.aspx.cs` 中的文本“Button clicked!”改为其他内容，保存文件，再次单击按钮。Web 页面上的文本会做相应的改变。

## 1. 控件面板

本节介绍可用控件，之后把它们组合到一个更丰富、更有趣的应用程序中。本节的内容对应于编辑 ASP.NET 页面时工具箱中的类别，如图 37-6 所示。

注意，在控件的描述中使用了“属性”——ASP.NET 代码中使用的属性与它同名。这里的引用并不完整，许多控件和属性都没有介绍，只介绍了最常用的属性。本章介绍的控件在 `Standard`、`Data` 和 `Validation` 类别中。`Navigation and Login` 和 `WebParts` 类别在第 38 章介绍，`AJAX` 扩展控件在第 39 章介绍，`Reporting` 控件可以在 Web 页面上报告信息，包括 `Crystal Reports`，本书不讨论。

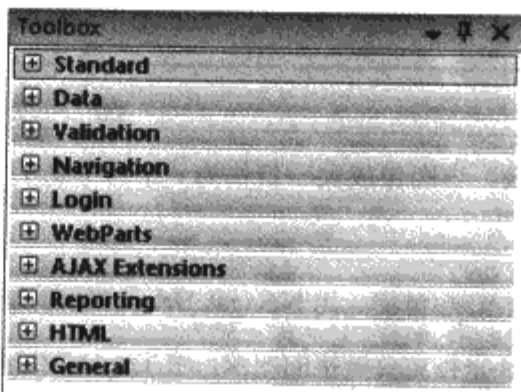


图 37-6

(1) 标准 Web 服务器控件

几乎所有的 Web 服务器控件(这个类别和其他类别)都继承了 System.Web.UI. WebControls. WebControl, 而 System.Web.UI.WebControls.WebControl 又继承了 System.Web. UI.Control。没有使用这个继承特性的 Web 服务器控件则直接派生于 Control 或更专门的基类, 而该基类又最终派生于 Control。因此, Web 服务器控件有许多共同的属性和事件, 如果需要, 就可以使用这些属性和事件。这里不可能介绍所有的元素, 只介绍 Web 服务器控件自身的属性和事件。

许多常用的继承属性主要用于处理显示格式, 这是很容易控制的, 例如属性 ForeColor、BackColor、Font 等, 也可以使用 CSS(Cascading Style Sheet)类来控制。此时, 应在一个独立的文件中, 把字符串属性 CssClass 设置为 CSS 类的名称。还可以使用 CSS 属性窗口和样式管理窗口给 CSS 控件设置样式。其他属性包括: Width 和 Height, 用于设置控件的大小; AccessKey 和 TabIndex, 便于用户的交互操作; Enabled, 设置控件的功能是否可以在 Web 窗体上使用。

一些控件还包含其他控件, 在页面上建立控件层次结构。使用 Controls 属性就可以访问给定控件包含的控件, 使用 Parent 可以访问控件的容器。

对于事件, 最常用的是继承来的 Load 事件, 它执行控件的初始化, PreRender 在控件输出 HTML 前进行最后一次修改。

可以使用的事件和属性很多, 下一章将详细介绍它们, 尤其是下一章将介绍更高级的样式设置技术。表 37-1 详细描述了标准 Web 服务器控件。

表 37-1

控 件	说 明
Label	显示简单文本, 使用 Text 属性设置和编程修改显示的文本
TextBox	提供一个用户可以编辑的文本框。使用 Text 属性访问输入的数据, Text Changed 事件可处理回送的选择变化。如果要求进行自动回送(而不是使用按钮), 就应把 AutoPostBack 属性设置为 true
Button	用户单击的标准按钮。Text 属性用于设置按钮上的文本, Click 事件用于响应单击(服务器回送是自动的)。也可以使用 Command 事件响应单击, 该事件可以访问接收的附加属性 CommandName 和 CommandArgument
LinkButton	与 Button 相同, 但把按钮显示为超链接

(续表)

控 件	说 明
ImageButton	显示一个图像，该图像放大一倍作为一个可单击的按钮，其属性和事件继承了 Button 和 Image
HyperLink	添加一个 HTML 超链接。用 NavigateUrl 设置目的地，用 Text 设置要显示的文本。也可以使用 ImageUrl 来指定要链接的图像，用 Target 指定要使用的浏览器窗口。这个控件没有非标准的事件，如果在链接后要执行其他处理，就应使用 LinkButton
DropDownList	允许用户选择一个列表项，可以直接从列表中选择，也可以键入前面的一或两个字母来选择。使用属性 Items 设置项目列表(这是一个包含 ListItem 对象的 ListItemCollection 类)，SelectedItem 和 SelectedIndex 属性可确定选择的内容。SelectedIndexChanged 事件可用于确定选项是否改变，这个控件也有 AutoPostBack 属性，所以选项的改变会触发一个回送操作
ListBox	允许用户从列表选择一个或多个列表。把 SelectionMode 设置为 Multiple 或 Single，可以确定一次选择多少个选项，Rows 确定要显示的选项个数。其他属性和事件与 DropDownList 控件相同
CheckBox	显示一个复选框。选择的状态存储在布尔属性 Checked 中，与复选框相关的文本存储在 Text 属性中。AutoPostBack 属性可以用于启动自动回送，CheckedChanged 事件则执行改变操作
CheckBoxList	创建一组复选框。属性和事件与其他列表控件相同，例如 DropDownList
RadioButton	显示一个单选按钮。一般情况下，它们都组合在一个组中，其中只有一个 RadioButton 控件是激活的。使用 GroupName 属性可以把 RadioButton 控件链接到一个组中。其他属性和事件与 CheckBox 相同
RadioButtonList	创建一组单选按钮，在这个组中，一次只能选择一个按钮。其属性和事件与其他列表控件相同
Image	显示一个图像。使用 ImageUrl 进行图像引用，如果图像加载失败，由 AlternateText 提供对应的文本
ImageMap	类似于 Image，但在用户单击图像中的一个或多个热区时，可以指定要触发的动作。要执行的动作可以是回送给服务器或重定向到另一个 URL 上。热区由派生于 HotSpot 的嵌入控件提供，例如 RectangleHotSpot 和 CircleHotSpot
Table	指定一个表。在设计期间可以使用它、TableRow 和 TableCell，或者使用 TableRowCollection 类的 Rows 属性编程指定数据行。也可以在运行期间进行修改时使用这个属性。与 TableRow 和 TableCell 一样，这个控件有几个只能用于表格的格式属性
BulletedList	把一个选项列表格式化为一个项目符号列表。与其他列表控件不同，这个控件有一个 Click 事件，用于确定用户在回送期间单击了哪个选项。其他属性和事件与 DropDownList 相同

(续表)

控 件	说 明
HiddenField	用于提供隐藏的字段，以存储不显示的值。这个控件可存储需要另一种存储机制才能发挥作用的设置。使用 Value 属性访问存储的值
Literal	执行与 Label 相同的功能，但没有样式属性，只有一个 Text 属性(因为它派生于 Control，而不是 WebControl)
Calendar	允许用户从图像日历中选择一个日期。这个控件有许多与格式相关的属性，但其基本功能是使用 SelectedDate 和 VisibleDate 属性(其类型是 System.Date Time)来访问由用户选择的日期和月份，并显示出来(总是包含 VisibleDate)。其关联的关键事件是 SelectionChanged。这个控件的回送是自动的
AdRotator	顺序显示几个图像。在每个服务器循环后，显示另一个图像。使用 Advertisement- File 属性指定描述图像的 XML 文件，AdCreated 事件在每个图像发回之前执行处理操作。也可以使用 Target 属性在单击一个图像时命名一个要打开的窗口
FileUpload	这个控件给用户显示一个文本框和一个 Browse 按钮，以选择要上传的文件。用户选择了文件之后，就可以使用 HasFile 属性确定是否选择了文件，然后使用后台代码中的 SaveAs() 方法执行文件的上传
Wizard	这个高级控件用于简化用户在几个页面中输入数据的常见任务。可以给向导添加多个步骤，按顺序或不按顺序显示给用户，并依赖此控件来维护状态
Xml	这是一个更复杂的文本显示控件，用于显示用 XSLT 样式表传输的 XML 内容，这些 XML 内容是使用 Document、DocumentContent 或 DocumentSource 属性中的一个设置(取决于原始 XML 的格式)的，XSLT 样式表(可选)是使用 Transform 或 TransformSource 来设置的
MultiView	这个控件包含一个或多个 View 控件，每次只显示一个 View 控件。当前显示的视图用 ActiveViewIndex 指定，如果视图改变了(可能因为单击了当前视图上的 Next 链接)，就可以使用 ActiveViewChanged 事件检测出来
Panel	添加其他控件的容器。可以使用 HorizontalAlign 和 Wrap 指定内容如何安排
PlaceHolder	这个控件不显示任何输出，但可以方便地把其他控件组合在一起，或者用编程的方式把控件添加到给定的位置。被包含的控件可以使用 Controls 属性来访问
View	控件的容器，类似于 PlaceHolder，但主要用作 MultiView 的子控件。使用 Visible 属性可以指定是否显示给定的 View，使用 Activate 和 Deactivate 事件检测激活状态的变化
Substitution	指定一组不与其他输出一一起高速缓存的 Web 页面，这是一个与 ASP.NET 高速缓存相关的高级主题，本书不涉及
Localize	与 Literal 相同，但允许使用项目资源指定要在不同区域显示的文本，使文本本地化

(2) 数据 Web 服务器控件  
数据 Web 服务器控件分为两类：

- 数据源控件(SqlDataSource、AccessDataSource、ObjectDataSource、XmlDataSource、和 SiteMapDataSource)
- 数据显示控件(GridView、DataList、DetailsView、FormView、Repeater 和 ReportViewer)

一般情况下，应把一个数据源控件(不可见)放在页面上，以链接数据源；然后添加一个绑定到数据源控件的数据显示控件，来显示该数据。一些更高级的数据显示控件，如 GridView，还可以编辑数据。

所有的数据源控件都派生于 System.Web.UI.DataSource 或 System.Web.UI.HierarchicalDataSource。这些类的方法，如 GetView()(或 GetHierarchicalView())，可以访问内部数据视图，还可以设置样式。

表 37-2 描述了各种数据源控件。注意本节没有探讨属性，这主要是因为这些控件最好通过图形化的向导来配置。本章的后面将使用这些控件，使读者更好地理解它们的工作方式。

表 37-2

控 件	说 明
SqlDataSource	用作 SQL Server 数据库中存储的数据的管道。把这个控件放在页面上，就可以使用数据显示控件操作 SQL Server 数据。本章后面将使用这个控件
AccessDataSource	与 SqlDataSource 相同，但处理存储在 Microsoft Access 数据库中的数据
LinqDataSource	这个控件可以处理支持 LINQ 数据模型的对象
ObjectDataSource	这个控件可以处理存储在自己创建的对象中的数据，这些对象可能组合在一个集合类中。这是把定制的对象模型显示在 ASP.NET 页面上的非常快捷的方式
XmlDataSource	可以绑定到 XML 数据上。它可以绑定导航控件，例如 TreeView。利用这个控件，还可以使用 XSL 样式表传输 XML 数据
SiteMapDataSource	可以绑定到层次站点地图数据上。详见第 38 章的导航 Web 服务器控件

接着是数据显示控件，如表 37-3 所示。其中几个控件可以满足各种需求。一些控件的功能比另外一些控件强，但我们常常使用最简单的控件(例如不需要编辑数据项)。

表 37-3

控 件	说 明
GridView	以数据行的格式显示多个数据项(例如数据库中的行)，其中每一行包含表示数据字段的列。利用这个控件的属性，可以选择、排序和编辑数据项
DataList	显示多个数据项，可以为每一项提供模板，以任意指定的方式显示数据字段。与 GridView 一样，可以选择、排序和编辑数据项
DetailsView	以表格形式显示一个数据项，表中的每一行都与一个数据字段相关。这个控件可以添加、编辑和删除数据项
FormView	使用模板显示一个数据项。与 DetailsView 一样，这个控件也可以添加、编辑和删除数据项
Repeater	与 DataList 相同，但不能选择和编辑数据
RepeaterViewer	显示报表服务数据的高级控件，本书不涉及



(3) 验证 Web 服务器控件

验证控件可以在不编写任何代码的前提(在大多数情况下)下验证用户的输入。只要有回送，每个验证控件就会检查控件是否有效，并相应地改变 IsValid 属性的值。如果这个属性是 false，被验证控件的用户输入就没有通过验证。包含所有控件的页面也有一个 IsValid 属性——如果页面中任一个有效性验证控件的 IsValid 属性设置为 false，该页面的 IsValid 属性就是 false。可以在服务器端的代码上检查这个属性，并对它进行操作。

验证控件还有第二个功能。它们不仅可以在运行期间验证控件的有效性，还可以自动给用户输出有帮助的提示，把 ErrorMessage 属性设置为希望的文本，在用户试图回送无效的数据时，就会看到这些文本。

存储在 ErrorMessage 中的文本可以在验证控件所在的位置输出，也可以和页面上其他验证控件的信息一起输出在一个独立的位置。第二种方式可以使用 ValidationSummary 控件来获得，并把所有的错误信息和附加文本按照需要显示出来。

在支持这些控件的浏览器中，验证控件甚至可以生成客户端的 JavaScript 函数，来简化验证任务的执行。在某些情况下，是不会有回送的，因为验证控件在某些环境下禁止回送，输出错误信息，而不涉及服务器的执行。

所有的验证控件都继承于 BaseValidator，所以它们共享几个重要的属性。最重要的是上面讨论的 ErrorMessage 属性；ControlToValidate 属性也是比较重要的，它指定要验证的控件的编程 ID。另一个重要的属性是 Display，它确定是把文本放在验证汇总的位置上(该属性设置为 none)，还是放在验证控件的位置上。也可以给错误信息留一些空间，即不显示这些错误信息(把 Display 设置为 Static)，或者按照需要给这些信息动态分配空间，这会使页面的内容有轻微的改变(把 Display 设置为 Dynamic)。表 37-4 描述了各个验证控件。

表 37-4

控 件	说 明
RequiredFieldValidator	如果用户在 TextBox 等控件中输入数据，就检查这些数据
CompareValidator	用于检查输入的数据是否满足简单的要求。利用一个运算符集合，通过 Operator 和 ValueToCompare 属性进行验证。Operator 的值可以是 Equal、GreaterThan、GreaterThanEqual、LessThan、LessThanEqual、NotEqual 或 DataTypeCheck。DataTypeCheck 可以比较 ValueToCompare 的数据类型和控件中要验证的数据。ValueToCompare 是一个字符串属性，但根据其内容可以把它解释为另一种数据类型。要进一步比较控件，可以把 type 属性设置为 Currency、Date、Double、Integer 或 String
RangeValidator	验证控件中的数据，看看其值是否在 MaximumValue 和 MinimumValue 属性值之间，其 Type 属性对应于每个 CompareValidator
RegularExpressionValidator	根据存储在 ValidationExpression 中的正则表达式验证字段的内容，可以用于验证邮政编码、电话号码、IP 号码等

(续表)

控 件	说 明
CustomValidator	使用定制函数验证控件中的数据。ClientValidationFunction 指定用于验证一个控件的客户端函数(这表示我们不能使用 C#)。这个函数应返回一个 Boolean 类型的值,表示验证是否成功。另外,还可以使用 ServerValidate 事件指定用于验证数据的服务器端函数。这个函数是一个 bool 类型的事件处理程序,其参数是一个包含要验证数据的字符串,而不是 EventArgs 参数。如果验证成功,就返回 true,否则返回 false
ValidationSummary	为所有设置了 ErrorMessage 的验证控件显示验证错误。通过设置 DisplayMode (BulletList、List 或 SingleParagraph) 和 HeaderText 属性,其显示的内容可以格式化;把 ShowSummary 设置为 false,就会禁止显示;把 ShowMessageBox 设置为 true,内容就会显示在弹出的消息框中

2. 服务器控件的示例

在这个示例中,要为一个 Web 应用程序(会议室登记工具)创建构架。与本书的其他示例一样,可以从 Wrox 网站 [www.wrox.com](http://www.wrox.com) 上下载示例应用程序的代码。现在仅介绍前端和简单的事件处理,后面将使用 ADO.NET 和数据绑定扩展这个示例,使之包含服务器端的事务逻辑。

要创建的 Web 窗体包含的字段有:用户名、事件名、会议室和参加者,以及可从中选择日期的一个日历(假定本例的作用是处理日常事件)。除了日历外,所有的字段都使用验证控件,只有日历在服务器端验证,并提供一个默认日期,以防没有输入日期。

为了测试用户界面,窗体上也提供了一个 Label 控件,使用它可以显示提交的结果。

首先,在 Visual Studio 中,在 C:\ProCSharp\Chapter37\目录下创建一个新的 Web 站点,命名为 PCSWebApp2。然后修改 Default.aspx 中的代码:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs"
    Inherits="_Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Meeting Room Booker</title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <h1 style="text-align: center;">
                Enter details and set a day to initiate an event.
            </h1>
        </div>
```

页面的标题是用 HTML 标记 <h1>括起来的,以得到大型的标题格式的文本,之后,窗体的主体放在 HTML 标记<table>中。可以使用一个 Web 服务器控件表格,但这会导致不

必要的复杂性, 因为使用表格的目的仅仅是为了格式化显示而已, 不是用于动态的 UI 元素 (在设计 Web 窗体时, 不要添加不必要的服务器控件)。这个表格有 3 列, 第 1 列包含简单的文本标签, 第 2 列包含对应于文本标签的 UI 字段 (以及这些字段的验证控件), 第 3 列包含一个日期控件, 可以从中选择日期, 这个控件占用了 4 行。第 5 行包含一个跨越所有列的提交按钮, 第 6 行包含一个 `ValidationSummary` 控件, 在需要时可以显示错误信息 (所有其他验证控件都设置了 `Display="None"`, 因为它们都使用这个汇总来显示错误)。在表的下面是一个简单的标签, 使用它可以显示结果, 以后我们还将添加数据库访问。

```
<div style="text-align: center;">
  <table style="text-align: left; border-color: #000000; border-width: 2px;
    background-color: #fff99e;" cellpadding="8" cellspacing="0" rules="none"
    width="540">
    <tr>
      <td valign="top">
        Your Name:</td>
      <td valign="top">
        <asp:TextBox ID="nameBox" Runat="server" Width="160px" />
        <asp:RequiredFieldValidator ID="validateName" Runat="server"
          ErrorMessage="You must enter a name."
          ControlToValidate="nameBox" Display="None" />
      </td>
      <td valign="middle" rowspan="4">
        <asp:Calendar ID="calendar" Runat="server" BackColor="White" />
      </td>
    </tr>
    <tr>
      <td valign="top">
        Event Name:</td>
      <td valign="top">
        <asp:TextBox ID="eventBox" Runat="server" Width="160px" />
        <asp:RequiredFieldValidator ID="validateEvent" Runat="server"
          ErrorMessage="You must enter an event name."
          ControlToValidate="eventBox" Display="None" />
      </td>
    </tr>
  </table>
```

这个文件中的大多数 ASP.NET 代码都非常简单, 许多代码只要浏览一遍就可以理解。特别要注意的是, 在代码中用于选择会议室和多个会议参加者的列表项目是如何附加到控件的:

```
<tr>
  <td valign="top">
    Meeting Room:</td>
  <td valign="top">
    <asp:DropDownList ID="roomList" Runat="server" Width="160px">
      <asp:ListItem Value="1">The Happy Room</asp:ListItem>
      <asp:ListItem Value="2">The Angry Room</asp:ListItem>
      <asp:ListItem Value="3">The Depressing
        Room</asp:ListItem>
      <asp:ListItem Value="4">The Funked Out
        Room</asp:ListItem>
    </asp:DropDownList>
    <asp:RequiredFieldValidator ID="validateRoom" Runat="server"
      ErrorMessage="You must select a room."
      ControlToValidate="roomList" Display="None" />
  </td>
</tr>
```

```
|  |  |
| --- | --- |
| Attendees: | Bill Gates</asp:ListItem> Monica Lewinsky</asp:ListItem> Vincent Price</asp:ListItem> Vlad the Impaler</asp:ListItem> Iggy Pop</asp:ListItem> William Shakespeare</asp:ListItem> </asp:ListBox> |

```

其中把 ListItem 对象与两个 Web 服务器控件关联起来。这些对象本身并不是 Web 服务器控件(它们仅继承了 System.Object), 因此不需要使用 Runat="server"。在处理页面时, 使用 <asp:ListItem> 项创建 ListItem 对象, 再把它们添加到父列表控件的 Items 集合中, 这便于初始化列表, 而无需编写代码(必须创建一个 ListItemCollection 对象, 添加 ListItem 对象, 再把集合传送给列表控件)。当然, 也可以编程完成这些工作:

```

<asp:RequiredFieldValidator ID="validateAttendees" Runat="server"
ErrorMessage="You must have at least one attendee."
ControlToValidate="attendeeList" Display="None" />
</td>
</tr>
|  |  |  |
| --- | --- | --- |
| <asp:Button ID="submitButton" Runat="server" Width="100%" Text="Submit meeting room request" /> | | |
| <asp:ValidationSummary ID="validationSummary" Runat="server" HeaderText="Before submitting your request:" /> | | |

</tr>
</table>
</div>
<div>
<p>
Results:
<asp:Label Runat="server" ID="resultLabel" Text="None." />
</p>
</div>
</form>
</body>
</html>

```

在设计视图上, 创建的窗体如图 37-7 所示。这是一个功能全面的 UI, 它可以在服务器请求之间维护它自己的状态, 并验证用户输入。上述代码非常简洁, 实际上, 我们几乎不需要做什么工作, 至少对于这个示例来说是这样, 而只需把按钮单击事件与提交按钮关联起来。



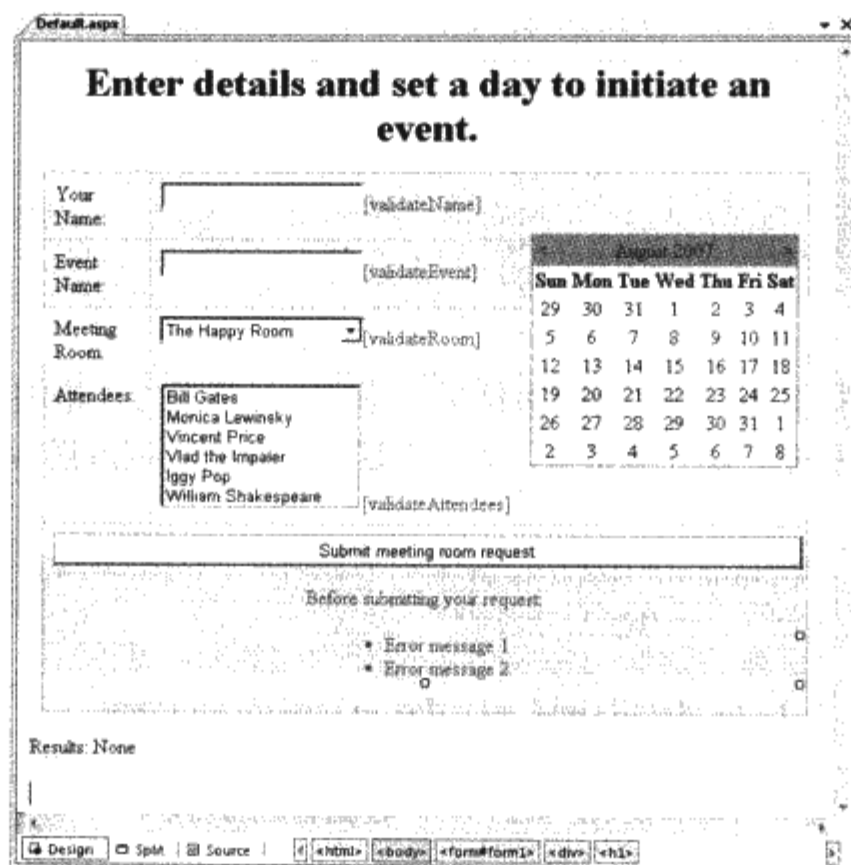


图 37-7

实际并非如此。因为我们没有验证日历控件。这很简单，只需给它设置一个初始值。在页面的 `Page_Load()` 事件处理程序中，可以设置该值：

```
private void Page_Load(object sender, System.EventArgs e)
{
    if (!this.IsPostBack)
    {
        calendar.SelectedDate = System.DateTime.Now;
    }
}
```

我们把今天的日期作为初始值。注意首先检查页面的 `IsPostBack` 属性，看看是否会把调用 `Page_Load()` 作为回送操作结果。如果正在进行回送，这个属性就应是 `true`，不必改变选中的日期(毕竟，我们不希望丢失用户的选择)。

要添加按钮单击处理程序，只需双击该按钮，并添加如下代码：

```
private void submitButton_Click(object sender, System.EventArgs e)
{
    if (this.IsValid)
    {
        resultLabel.Text = roomList.SelectedItem.Text +
            " has been booked on " +
            calendar.SelectedDate.ToLongDateString() +
            " by " + nameBox.Text + " for " +
            eventBox.Text + " event. ";
        foreach (ListItem attendee in attendeeList.Items)
        {

```



```
        if (attendee.Selected)
        {
            resultLabel.Text += attendee.Text + ", ";
        }
    }
    resultLabel.Text += " and " + nameBox.Text +
        " will be attending.";
}
```

把 resultLabel 控件的 Text 属性设置为结果字符串，显示在主表格的下方。在 IE 中，这个提交结果应如图 37-8 所示，除非有错误，否则在这种情况下就应激活 ValidationSummary，如图 37-9 所示。

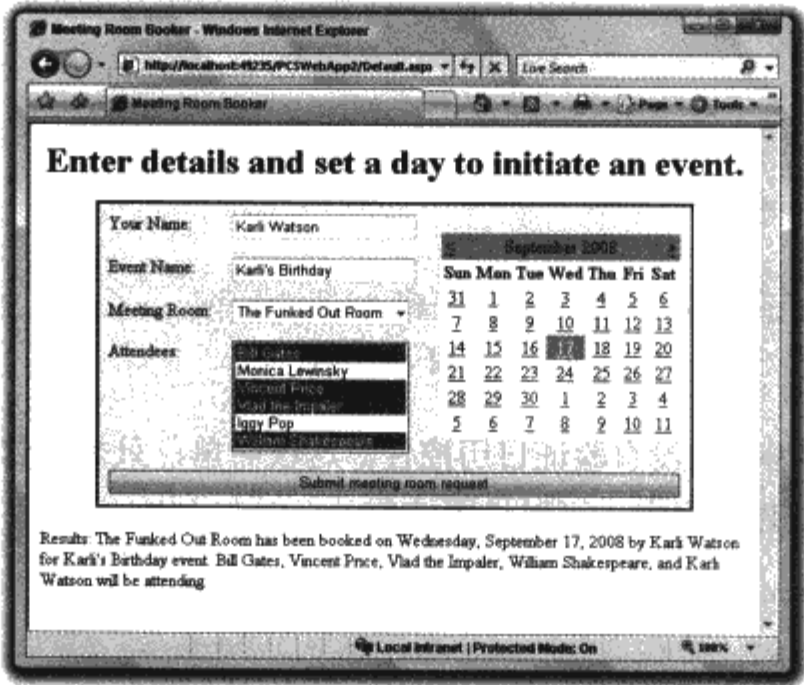


图 37-8

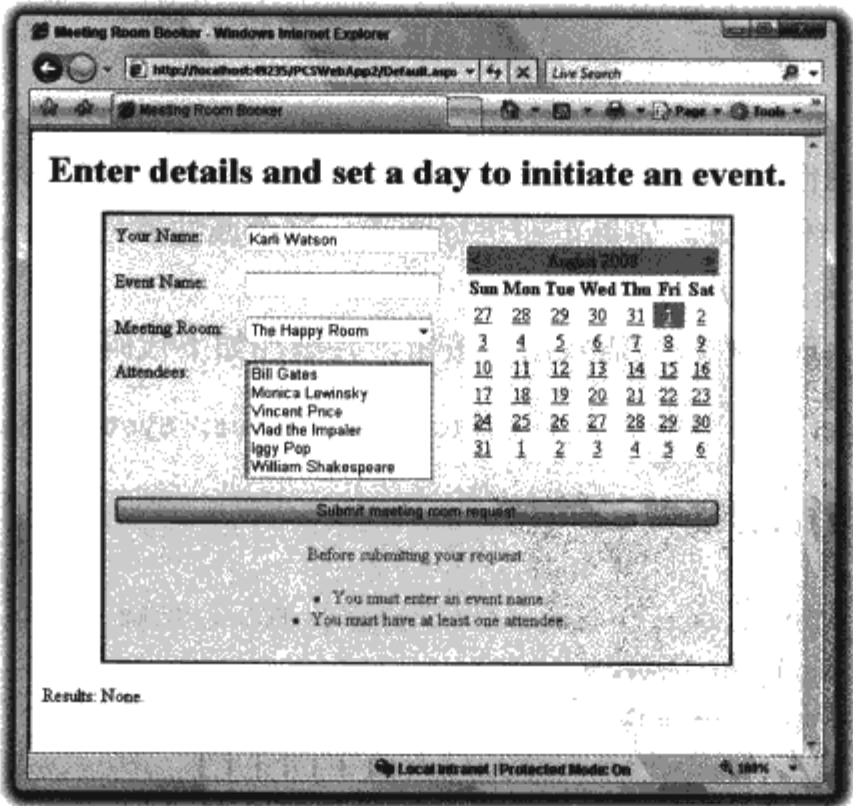


图 37-9

## 37.3 ADO.NET 和数据绑定

上一节创建的 Web 窗体应用程序有很好的功能，但只包含静态数据。另外，会议登记过程不包含永久的事件数据。为了解决这两个问题，可以使用 ADO.NET 访问存储在数据库中的数据，这样就可以存储和检索会议数据，以及会议室和参加者的列表。

数据绑定可以使检索数据的过程变得非常简单。像列表框(和一些更专业的控件)这样的控件可以使用这种技巧。它们可以绑定到执行 IEnumerable、ICollection、或 IListSource 接口的任何对象上，包括数据源 Web 服务器控件。

本节首先更新会议登记应用程序，使之支持数据；再使用一些其他支持数据的 Web 控件，介绍其他一些可以通过数据绑定完成的任务。

### 37.3.1 更新会议登记应用程序

为了区别于上一个示例，在 C:\ProCSharp\Chapter37\目录下创建一个新的 Web 站点 PCSWebApp3，复制前面创建的 PCSWebApp2 应用程序中的代码。在开始编写新代码前，先看看要访问的数据库。

#### 1. 数据库

在本例中，使用一个 Microsoft SQL Server Express 数据库 MeetingRoomBooker.mdf，该数据库可以从本书的代码中下载。企业级的应用程序应使用 SQL Server 数据库，但涉及到的技术是相同的，使用 SQL Server Express 会使测试更简单一些，代码也相同。

**提示：**

如果添加这个数据库的另一个版本，就需要在 Solution Explorer 的 App\_Data 文件夹中添加一个新数据库。为此，右击 App\_Data 文件夹，选择 Add New Item，再选择数据库，命名为 MeetingRoomBooker，然后单击 Add。这也会在 Server Explorer 窗口中配置一个数据连接，以供使用。之后就可以按照下一节的要求添加表，提供自己的数据。另外，要通过编写代码来使用下载的数据库，只需把它复制到 Web 站点的 App\_Data 文件夹下。

该数据库包含 3 个表：

- Attendees 包含事件参加者的一个列表。
- Rooms 包含会议室的一个列表。
- Events 包含登记事件的一个列表。

#### (1) 参加者

Attendees 表包含表 37-5 所示的列。

该数据库包含 20 个参加者的信息，他们都有电子邮件地址。在一个比较高级的应用程序中，电子邮件会自动发送给已登记的参加者，我们把这个任务留给您来完成，其中使用的技巧可以在本书的其他地方找到。

表 37-5

列	类 型	说 明
ID	Identity, 主键	参加者的身份标识号码
Name	varchar, 必选, 50 个字符	参加者的姓名
Email	varchar, 可选, 50 个字符	参加者的电子邮件地址

(2) 会议室

Rooms 表包含表 37-6 所示的列。

表 37-6

列	类 型	说 明
ID	Identity, 主键	房间标识号码
Room	varchar, 必选, 50 个字符	房间名

数据库中提供了 20 条记录。

(3) 事件

Events 表包含表 37-7 所示的列。

表 37-7

列	类 型	说 明
ID	Identity, 主键	会议标识号码
Name	varchar, 必选, 255 个字符	会议名称
Room	Int, 必选	会议室 ID.
AttendeeList	Text, 必选	参加者姓名列表
EventDate	DateTime, 必选	会议日期

下载的数据库中提供了几个会议。

2. 数据库的绑定

要绑定数据的两个控件是 attendeeList 和 roomList。在此之前，需要添加 Web 服务器控件 SqlDataSource，以映射要在 MeetingRoomBooker.mdf 数据库中访问的表。最快捷的方式是把它从工具箱拖放到 Web 窗体 Default.aspx 上，通过配置向导配置它们。图 37-10 显示了如何为 SqlDataSource 控件 MRBAttendeeData 访问这个向导。

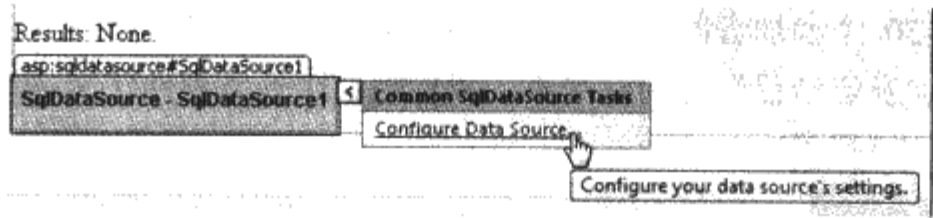


图 37-10

在数据源配置向导的第一个页面上，需要选择前面创建的数据库连接。接着，选择把连接字符串保存为 MRBConnectionString，然后从数据库的 Attendees 表中选择\*(所有字段)。

之后，把 SqlDataSource 控件的 ID 改为 MRBAttendeeData。还需要添加并配置另外两个 SqlDataSource 控件，以使用 MRBRoomData 和 MRBEventData 的 ID 值从 Rooms 和 Events 表中获得数据。这两个控件可以使用前面保存的 MRBConnectionString 进行连接。

添加了这些数据源之后，它们在窗体代码中的语法是非常简单的：

```
<asp:SqlDataSource ID="MRBAttendeeData" runat="server"
    ConnectionString="<%= ConnectionStrings.MRBConnectionString %>"
    SelectCommand="SELECT * FROM [Attendees]"></asp:SqlDataSource>
<asp:SqlDataSource ID="MRBRoomData" runat="server"
    ConnectionString="<%= ConnectionStrings.MRBConnectionString %>"
    SelectCommand="SELECT * FROM [Rooms]"></asp:SqlDataSource>
<asp:SqlDataSource ID="MRBEventData" runat="server"
    ConnectionString="<%= ConnectionStrings.MRBConnectionString %>"
    SelectCommand="SELECT * FROM [Events]"></asp:SqlDataSource>
```

连接字符串的定义在 web.config 文件中，本章后面将详细探讨这个文件。

接着，设置 roomList 和 attendeeList 控件的数据绑定属性。对于 roomList，需要的设置如下：

- DataSourceID—MRBRoomData
- DataTextField—Room
- DataValueField—ID

同样，对于 attendeeList，需要的设置如下：

- DataSourceID—MRBAttendeeData
- DataTextField—Name
- DataValueField—ID

也可以从代码中删除这些控件已有的硬编码列表项。

现在运行这个应用程序，从数据绑定控件中得到所有可用的参加者和会议室数据。稍后使用 MRBEventData 控件。

### 3. 定制日历控件

在把会议添加到数据库中之前，先修改一下日历的显示。最好用另一种颜色显示登记之前的日期，以防该日期被选中。这要求修改在日历中设置日期的方式，以及日期单元格的显示方式。

首先是日期选择。有 3 个地方需要查看会议登记的日期，并修改相应选择：一是在 Page\_Load() 中设置初始日期时；二是在用户试图从日历中选择日期时；三是登记一个会议，并设置一个新的日期，以防用户在选择新日期前，在同一天连续登记两个会议。这些都是很常见的情况，也可以创建一个私有方法来执行这个计算。这个方法应接受一个试用日期作为参数，并返回要使用的日期，该日期可以与试用日期相同，也可以是试用日期之后的某个日期。

在添加这个方法之前，需要让代码访问 Events 表中的数据。为此可以使用 MRBEventData 控件，因为这个控件可以填充 DataView。所以，添加下面的私有成员和属性：

```
private DataView eventData;
```

```

private DataView EventData
{
    get
    {
        if (eventData == null)
        {
            eventData =
                MRBEventData.Select(new DataSourceSelectArguments()) as DataView;
        }
        return eventData;
    }
    set
    {
        eventData = value;
    }
}

```

EventData 属性用需要的数据填充 eventData 成员，其结果缓存起来，供以后使用。这里使用 SqlDataSource.Select() 方法获得 DataView。

把这个 GetFreeDate() 方法添加到后台代码文件中：

```

private System.DateTime GetFreeDate(System.DateTime trialDate)
{
    if (EventData.Count > 0)
    {
        System.DateTime testDate;
        bool trialDateOK = false;
        while (!trialDateOK)
        {
            trialDateOK = true;
            foreach (DataRowView testRow in EventData)
            {
                testDate = (System.DateTime)testRow["EventDate"];
                if (testDate.Date == trialDate.Date)
                {
                    trialDateOK = false;
                    trialDate = trialDate.AddDays(1);
                }
            }
        }
    }
    return trialDate;
}

```

这段简单的代码使用 EventData DataView 提取会议数据。首先看看一般情况：没有登记任何会议，此时返回该试用日期，以确认该日期，接着对 Event 表中的日期进行迭代，把该日期与试用日期比较。如果找到一个匹配，就给试用日期加一天，执行另一次搜索。

从 DataTable 中提取数据是相当简单的：

```
testDate = (System.DateTime)testRow["EventDate"];
```

把列数据转换为 Sytem.DateTime，这样会更精确。

使用 getFreeDate() 的第一个地方是在 Page\_Load() 后面。这表示只需对设置 SelectedDate 属性的代码稍加修改：

```
if (!this.IsPostBack)
```



```
{
    System.DateTime trialDate = System.DateTime.Now;
    calendar.SelectedDate = getFreeDate(trialDate);
}
```

接着需要响应日历上的日期选择。为此，需要先为日历的 SelectionChanged 事件添加一个事件处理程序，强制检查现有会议的日期。双击设计器中的日历，添加如下代码：

```
private void calendar_SelectionChanged(object sender, System.EventArgs e)
{
    System.DateTime trialDate = calendar.SelectedDate;
    calendar.SelectedDate = getFreeDate(trialDate);
}
```

这段代码与 Page\_Load()相同。

执行这种检查的第三个地方是响应登记按钮的单击。后面会解释它，因为后面进行了许多改变。

接着把日历的日期单元格变为另一种颜色，以表示现存的会议。为此，需要给日期对象的 DayRender 事件添加一个事件处理程序。每次显示一个日期时，都会触发这个事件，并允许通过在处理程序中接收到的 DayRenderEventArgs 参数的 Cell 和 Date 属性，访问要显示的单元格对象和这个单元格的日期。我们需要比较要显示的单元格中的日期和 eventTable 对象中的日期，如果匹配，就可以使用 Cell.BackColor 属性为单元格着色：

```
void calendar_DayRender(object sender, DayRenderEventArgs e)
{
    if (EventData.Count > 0)
    {
        System.DateTime testDate;
        foreach (DataRowView testRow in EventData)
        {
            testDate = (System.DateTime)testRow["EventDate"];
            if (testDate.Date == e.Day.Date)
            {
                e.Cell.BackColor = System.Drawing.Color.Red;
            }
        }
    }
}
```

这里使用红色，得到屏幕图 37-11。6 月的 12、15、22 日都有会议，所以用户选择了 24 日。

June 2008						
Sun	Mon	Tue	Wed	Thu	Fri	Sat
25	26	27	28	29	30	31
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	1	2	3	4	5

图 37-11

添加了日期选择逻辑后，就不可能选择显示为红色的一天，如果要选择这样的日期，就会

选择该日期后面的某一天。例如，在图 37-6 的日历中单击 6 月 15 日，就会选择 16 日。

#### 4. 给数据库添加会议数据

submitButton\_Click()事件处理程序目前从会议特性中组合了一个字符串，并在 resultLabel 控件中显示它。要给数据库添加一个会议，需要把创建出来的字符串重新格式化到一个 SQL INSERT 查询中，并执行它。

##### 注意：

在开发环境中，不必过多地考虑安全性。通过 Web 站点解决方案添加一个 SQL Server 2005 Express 数据库，把 SqlDataSource 控件配置为使用该数据库，会自动建立一个连接字符串，它可用于写入数据库。在比较高级的场合下，可以使用其他账户访问资源，例如域账户用于访问网络其他地方的 SQL Server 实例。ASP.NET 中有这个功能(通过模拟、COM+服务或其他方式获得)，但超出了本书的范围。在大多数情况下，只要正确配置连接字符串，能正常完成任务即可。

下面的许多代码都是很熟悉的：

```
void submitButton_Click(object sender, EventArgs e)
{
    if (this.IsValid)
    {
        System.Text.StringBuilder sb = new System.Text.StringBuilder();
        foreach (ListItem attendee in attendeeList.Items)
        {
            if (attendee.Selected)
            {
                sb.AppendFormat("{0} ({1}), ", attendee.Text, attendee.Value);
            }
        }
        sb.AppendFormat(" and {0}", nameBox.Text);
        string attendees = sb.ToString();
        try
        {
            System.Data.SqlClient.SqlConnection conn =
                new System.Data.SqlClient.SqlConnection(
                    ConfigurationManager.ConnectionStrings["MRBConnectionString"]
                        .ConnectionString);
            System.Data.SqlClient.SqlCommand insertCommand =
                new System.Data.SqlClient.SqlCommand("INSERT INTO [Events] "
                    + "(Name, Room, AttendeeList, EventDate) VALUES (@Name, "
                    + "@Room, @AttendeeList, @EventDate)", conn);
            insertCommand.Parameters.Add(
                "Name", SqlDbType.VarChar, 255).Value = eventBox.Text;
            insertCommand.Parameters.Add(
                "Room", SqlDbType.Int, 4).Value = roomList.SelectedValue;
            insertCommand.Parameters.Add(
                "AttendeeList", SqlDbType.Text, 16).Value = attendees;
            insertCommand.Parameters.Add(
                "EventDate", SqlDbType.DateTime, 8).Value = calendar.SelectedDate;
```

这里最有趣的是如何使用下面的语法访问前面创建的连接字符串：

```
ConfigurationManager.ConnectionStrings["MRBConnectionString"].ConnectionString
```

ConfigurationManager 类可以访问所有配置信息,它们都存储在 Web 应用程序的 Web.Config 配置文件中。该文件详见本章后面的内容。

创建了 SQL 命令后,就可以使用它插入新事件:

```
conn.Open();
int queryResult = insertCommand.ExecuteNonQuery();
conn.Close();
```

ExecuteNonQuery()返回一个整数,表示查询会影响表中的多少行。如果它等于 1,插入就是成功的。此时把一个成功的信息放在 resultLabel 中,清除 eventData,因为它现在已过期了。把日历选择改为一个新的、没有会议的日期。因为 GetFreeDate()使用 eventData,而 eventData 属性在没有数据时会自动刷新它自己,所以会刷新存储的会议数据:

```
if (queryResult == 1)
{
    resultLabel.Text = "Event Added.";
    eventData = null;
    calendar.SelectedDate =
        GetFreeDate(calendar.SelectedDate.AddDays(1));
}
```

如果 ExecuteNonQuery()返回的数字不是 1,就会有问题。在本例中不必担心,只需抛出一个异常,该异常会在一般的 catch 块中捕获,该 catch 块位于数据库访问代码中。它会在 resultLabel 中显示一个故障通知:

```
else
{
    throw new System.Data.DataException("Unknown data error.");
}
catch
{
    resultLabel.Text = "Event not added due to DB access "
        + "problem.";
}
}
```

支持数据的会议登记应用程序就完成了。

### 37.3.2 数据绑定的更多内容

如前所述,Web 服务器控件有几个处理数据显示的控件:GridView、DataList、DetailsView、FormView 和 Repeater。在把数据输出到网页上时,这些都是非常有用的,因为它们会自动执行许多任务,否则将需要编写许多代码。

首先,介绍如何使用这些控件,在 PCSWebApp3 的底部显示一个会议列表。

把一个 GridView 控件从工具箱拖放到 Default.aspx 的底部,选择前面添加的 MRBEventData 数据源,如图 37-12 所示。

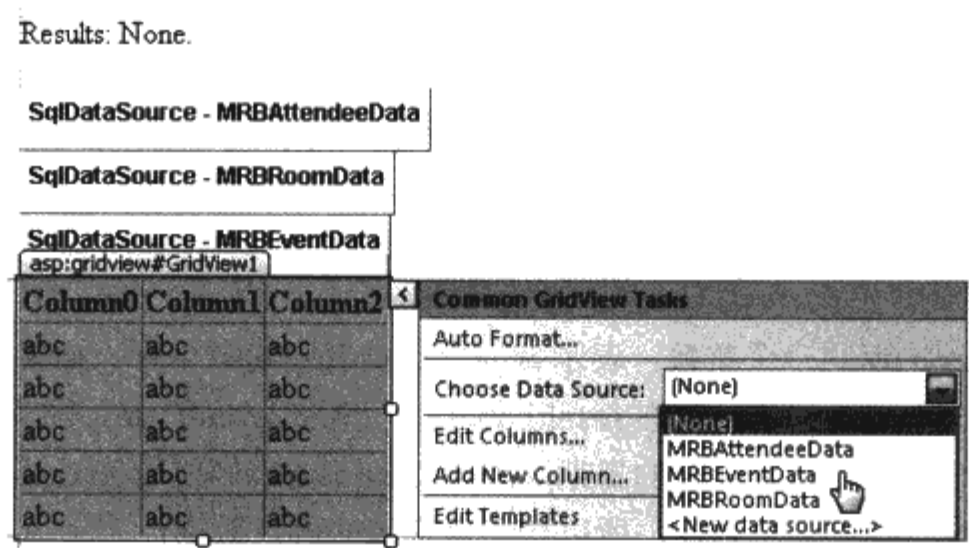


图 37-12

接着单击 Refresh Schema，这就是在窗体底部显示会议列表所需做的工作，现在查看 Web 站点，就会看到会议，如图 37-13 所示。

ID	Name	Room	AttendeeList	EventDate
1	My Birthday	4	Iggy Pop (5), Sean Connery (7), Albert Einstein (10), George Clooney (14), Jules Verne (18), Robin Hood (20), and Karl Watson	9/17/2008 12:00:00 AM
2	Dinner	1	Bill Gates (1), Monika Lewinsky (2), and Bruce Lee	8/5/2008 12:00:00 AM
3	Discussion of darkness	6	Vlad the Impaler (4), Myra Hindley (13), and Beelzebub	10/29/2008 12:00:00 AM
4	Christmas with Pals	9	Dr Frank N Furter (11), Bobby Davro (15), John F Kennedy (16), Stephen King (19), and Karl Watson	12/25/2008 12:00:00 AM
5	Escape	17	Monika Lewinsky (2), Stephen King (19), and Spartacus	5/10/2008 12:00:00 AM
6	Planetary Conquest	14	Bill Gates (1), Albert Einstein (10), Dr Frank N Furter (11), Bobby Davro (15), and Darth Vader	6/15/2008 12:00:00 AM
7	Homecoming Celebration	7	William Shakespeare (6), Christopher Columbus (12), Robin Hood (20), and Ulysses	6/22/2008 12:00:00 AM
8	Dalek Reunion Ball	12	Roger Moore (8), George Clooney (14), Bobby Davro (15), and Davros	6/12/2008 12:00:00 AM
9	Romantic meal for two	13	George Clooney (14), and Donna Watson	3/29/2008 12:00:00 AM

图 37-13

还可以对 submitButton\_Click()做进一步的修改，确保在添加新记录时更新数据：

```
if (queryResult == 1)
{
    resultLabel.Text = "Event Added.";
    EventDate = null;
    calendar.SelectedDate =
        GetFreeDate(calendar.SelectedDate.AddDays(1));
    GridView1.DataBind();
}
```

所有的数据绑定控件都支持这个方法，如果调用顶层的(this) DataBind()方法，窗体就会调用该方法。

注意，在图 37-13 中，EventDate 字段的日期/时间显示有点麻烦。由于我们只查看日期，因此时间总是 12:00:00，其实这个信息不需要显示。下一节将学习如何以更友好的方式在

DataList 控件中显示这个日期信息。DataGrid 控件包含许多属性，它们可以用于格式化显示的数据，但这部分内容由您自学。

1. 使用模板显示数据

许多数据显示控件可以使用模板来格式化要显示的数据。模板在 ASP.NET 中是 HTML 的参数化部分，用作某些控件的输出元素。它们可以定制如何将数据输出到浏览器上，不需要做太多的工作就可以得到专业级的显示结果。

有几个模板可用于定制列表的各个方面。对于 Repeater 和 DataList 来说，一个重要的模板是<ItemTemplate>，它可以用于显示 Repeater、DataList 和 ListView 控件的每个数据项。在控件声明中声明这个模板(和其他模板)，例如：

```
<asp:DataList Runat="server" ... >
  <ItemTemplate>
    ...
  </ItemTemplate>
</asp:DataList>
```

在模板声明中，一般是输出 HTML 的部分内容，参数是绑定到控件的数据。在输出这些参数时，应使用一种特殊的语法：

```
<%# expression %>
```

expression 占位符是把参数绑定到页面或控件属性上的表达式，但它常常是由 Eval()或 Bind()表达式组成。通过指定表中的列，这个函数可以从绑定到控件的表中输出数据，Eval()使用下面的语法：

```
<%# Eval("ColumnName") %>
```

还有第二个可选参数，可以格式化返回的数据，它的语法与其他地方使用的字符串格式化表达式相同。该参数可以把日期字符串格式化为可读性更高的格式，这正是前面示例所缺乏的。

Bind()表达式与 Eval()相同，但可以把数据插入服务器控件的属性，例如：

```
<asp:Label RunAt="server" ID="ColumnDisplay" Text='<%# Bind("ColumnName") %>' />
```

注意，双引号可在 Bind()参数中使用，所以应使用单引号把属性值括起来。  
表 37-8 列出了可用的模板以及它们的用法。

表 37-8		
模 板	应 用 于	说 明
<ItemTemplate>	DataList, Repeater	列表项使用的模板
<HeaderTemplate>	DataList, DetailsView, FormView, Repeater	列表项前输出内容使用的模板
<FooterTemplate>	DataList, DetailsView, FormView, Repeater	列表项后输出内容使用的模板
<LayoutTemplate>	ListView	用于指定输出周围的项



(续表)

模 板	应 用 于	说 明
<SeparatorTemplate>	DataList, Repeater	列表中项之间使用的模板
<ItemSeparatorTemplate>	ListView	列表中项之间使用的模板
<AlternatingItemTemplate>	DataList, ListView	其他项使用的模板，有助于查看
<SelectedItemTemplate>	DataList, ListView	列表中所选项使用的模板
<EditItemTemplate>	DataList, FormView, ListView	用于列表中正在编辑的项的模板
<InsertItemTemplate>	FormView, ListView	用于列表中正在插入的项的模板
<EmptyDataTemplate>	GridView, DetailsView, FormView	用于显示空项，例如 GridView 中没有记录时使用它
<PagerTemplate>	GridView, DetailsView, FormView	用于格式化分页
<GroupTemplate>	ListView	用于指定输出周围的项的组合
<GroupSeparatorTemplate>	ListView	列表中项之间使用的模板
<EmptyItemTemplate>	ListView	使用分组的项时，该模板用于为组中的空项提供输出。这个模板在组中没有足够的项时使用

了解模板最简单的方式是举一个例子。

2. 使用模板

在 PCSWebApp3 的 Default.aspx 页面顶部扩展表格，使之包含一个 ListView，显示存储在数据库中的每个会议。使这些会议成为可选择的，这样单击每个会议的名称，就在 FormView 控件中显示它们的信息。

首先需要为数据绑定控件创建新的数据源。每个数据绑定控件最好有自己的数据源。

ListView 控件需要 SqlDataSource 控件 MRBEventData2, MRBEventData2 与 MRBEventData 相同，但它只需返回 Name 和 ID 数据。需要的代码如下：

```
<asp:SqlDataSource ID="MRBEventData2" Runat="server"
  SelectCommand="SELECT [ID], [Name] FROM [Events]"
  ConnectionString="<%$ ConnectionStrings:MRBConnectionString %>"
/>
```

FormView 控件的数据源 MRBEventDetailData 比较复杂，但可以通过数据源配置向导方便地建立它。这个数据源使用 ListView 控件的选中项 EventList，获取选中的项的数据。这可以使用 SQL 查询中的一个参数实现，如下所示：

```
<asp:SqlDataSource ID="MRBEventDetailData" Runat="server"
  SelectCommand="SELECT dbo.Events.Name, dbo.Rooms.Room, dbo.Events.AttendeeList,
    dbo.Events.EventDate FROM dbo.Events INNER JOIN dbo.Rooms
    ON dbo.Events.ID = dbo.Rooms.ID WHERE dbo.Events.ID = @ID"
```

```

ConnectionString="<%%$ ConnectionStrings:MRBConnectionString %>">
<SelectParameters>
  <asp:ControlParameter Name="ID" DefaultValue="-1" ControlID="EventList"
    PropertyName="SelectedValue" />
</SelectParameters>
</asp:SqlDataSource>

```

其中 ID 参数会得到在 Select 查询的 @ID 处插入的值。ControlParameter 项从 EventList 的 SelectedValue 属性中提取这个值，如果没有选中的项，就使用 -1。初看起来，这个语法有点古怪，但它非常灵活。一旦使用向导生成了这些对象，就不需要自己建立它们了。

下面需要添加 DataList 和 FormView 控件。修改 PCSWebApp3 项目的 Default.aspx 中的代码：

```

<tr>
  <td align="center" colspan=3>
    <asp:ValidationSummary ID=validationSummary Runat="server"
      HeaderText="Before submitting your request:" />
  </td>
</tr>
<tr>
  <td align="left" colspan=3 style="width: 40%;">
    <table cellpadding="4" style="width: 100%;">
      <tr>
        <td colspan=2 style="text-align: center;">
          <h2>Event details</h2>
        </td>
      </tr>
      <tr>
        <td style="width: 40%; background-color: #ccffcc; valign="top">
          <asp:ListView ID="EventList" runat="server"
            DataSourceID="MRBEventData2" DataKeyNames="ID"
            OnSelectedIndexChanged="
              EventList_SelectedIndexChanged" >
            <LayoutTemplate >
              <ul >
                <asp:Placeholder ID="itemPlaceholder"
                  runat="server" / >
              </ul >
            </LayoutTemplate >
            <ItemTemplate >
              <li >
                <asp:LinkButton Text=' < %% Bind("Name") % > '
                  runat="server" ID="NameLink" CommandName="Select"
                  CommandArgument=' < %% Bind("ID") % > '
                  CausesValidation="false" / >
              </li >
            </ItemTemplate >
            <SelectedItemTemplate >
              <li >
                <b > < %% Eval("Name") % > </b >
              </li >
            </SelectedItemTemplate >
          </asp:ListView >
        </td>

```

```
 <asp:FormView ID="FormView1" Runat="server"         DataSourceID="MRBEventDetailData">         <ItemTemplate>           <h3><%# Eval("Name") %></h3>           <b>Date:</b>           <%# Eval("EventDate", "{0:D}") %>           <br />           <b>Room:</b>           <%# Eval("Room") %>           <br />           <b>Attendees:</b>           <%# Eval("AttendeeList") %>         </ItemTemplate>       </asp:FormView>     </td>   </tr> </table> </td> </tr> </table> |
```

我们添加了一个新的表行，其中包含一个表，该表中的一列是一个 ListView 控件，另一列是一个 FormView 控件。

ListView 使用 <LayoutTemplate> 输出一个项目列表，使用 <ItemTemplate> 和 <SelectedItemTemplate> 显示会议信息。在 <LayoutTemplate> 中，用 ID 属性为 “itemPlaceholder” 的 Placeholder 控件给数据项指定一个容器元素。为了提供选择，对会议名称链接执行 Select 命令，该会议名称链接显示在 <ItemTemplate> 中，这样就可以自动修改选择。我们还使用了 OnSelectedIndexChanged 事件，当 Select 命令修改选择时触发这个事件，以更新列表，用不同的风格显示选中的项。事件处理程序如下所示：

```

void EventList_SelectedIndexChanged(object sender, EventArgs e)
{
    EventList.DataBind();
}

```

还需要确保新会议添加到列表中：

```

if (queryResult == 1)
{
    resultLabel.Text = "Event Added.";
    EventData = null;
    calendar.SelectedDate =
        GetFreeDate(calendar.SelectedDate.AddDays(1));
    GridView1.DataBind();
    EventList.DataBind();
}

```

现在会议的详细信息就显示在表中，如图 37-14 所示。

使用模板和数据绑定控件可以完成许多任务，需要用一本书的篇幅来介绍。但是，这里介绍的内容已经足够您开始试用它们了。

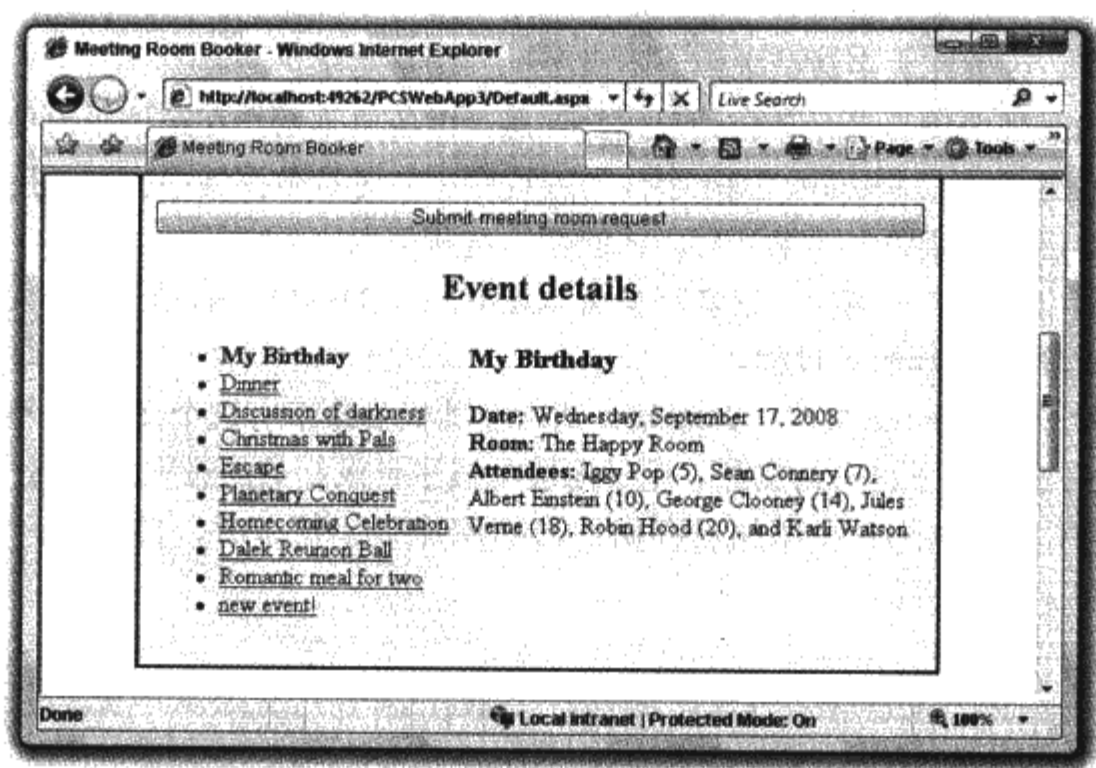


图 37-14

## 37.4 应用程序配置

本章一直暗示一个事情，那就是应用程序都包含网页和配置设置。这是一个重要的概念，必须掌握，特别是为多个同步用户配置网站时，这个概念就更加重要了。

首先介绍一些术语和应用程序的生命期。应用程序定义为项目中的所有文件，由 web.config 文件配置。在第一次创建应用程序时，将创建一个 Application 对象，即收到第一个 HTTP 请求时创建该对象。此时还将触发 Application\_Start 事件，创建一个 HttpApplication 实例池。每个输入的请求都会接收到这样一个实例，执行请求的处理过程。注意，HttpApplication 对象不需要处理同步访问，与全局 Application 对象不同。所有的 HttpApplication 实例完成任务后，就触发 Application\_End 事件，应用程序终止执行，消除 Application 对象。

上面涉及到的事件处理程序(以及本章讨论的所有其他事件处理程序)可以在 Global.asax 文件中定义，该文件可以添加到任意 Web 站点项目中。生成的文件包含空格，用户可以在这些空格中填写信息，例如：

```
protected void Application_Start(Object sender, EventArgs e)
{
}
```

在单个用户使用 Web 应用程序时，会启动一个会话。与应用程序类似，会话将创建一个用户特定的 Session 对象，并触发 Session\_Start 事件。在一个会话中，每个请求都将触发 Application\_BeginRequest 和 Application\_EndRequest 事件。在一个会话中可以多次触发这两个事件，因为这些事件会访问应用程序中的不同资源。会话可以手动终止，如果没有接收到更多的请求，会话也会因超时而停止。会话终止将触发 Session\_End 事件，消除 Session 对象。

了解这个过程，可以执行几个操作，以简化应用程序。例如，如果应用程序的所有实例都

使用一个资源密集型对象，就可以考虑在应用程序一级上实例化它。这将提高性能，减少多个用户使用的内存，因为在大多数请求中，不需要进行这个实例化。

可以使用的另一个技巧是存储会话级别的信息，以备单个用户在跨请求时使用。这些信息包括用户第一次连接（在 Session\_Start 事件处理程序中）时从数据库中提取的用户特定信息，在会话终止（通过超时或用户请求）后，才能使用这些信息。

这些技巧超出了本书的范围，读者可参阅《ASP.NET 2.0 高级编程》(清华大学出版社引进并已出版)，它们有助于理解该过程。

最后，看看 Web.Config 文件。Web 站点通常在其根目录下有这个文件(但不会在默认情况下创建它)，在其子目录下也有该文件，用于配置与该子目录相关的设置(如安全性)。本章开发的 Web 站点 PCSWebApp3 在添加已存储的数据库连接字符串时，会接收一个自动生成的 Web.Config 文件，如下所示：

```
<connectionStrings>
  <add name="MRBConnectionString" connectionString="Data Source=.\SQLEXPRESS;
    AttachDbFilename=|DataDirectory|\MeetingRoomBooker.mdf;
    Integrated Security=True;User Instance=True"
    providerName="System.Data.SqlClient" />
</connectionStrings>
```

如果在调试模式下运行项目，就会在 Web.Config 文件中看到一些额外的设置。

可以手工编辑 Web.Config 文件，还可以使用一个工具配置 Web 站点(及其底层的配置文件)。这个工具在 Visual Studio Website 菜单的 ASP.NET Configuration 上。该工具如图 37-15 所示。

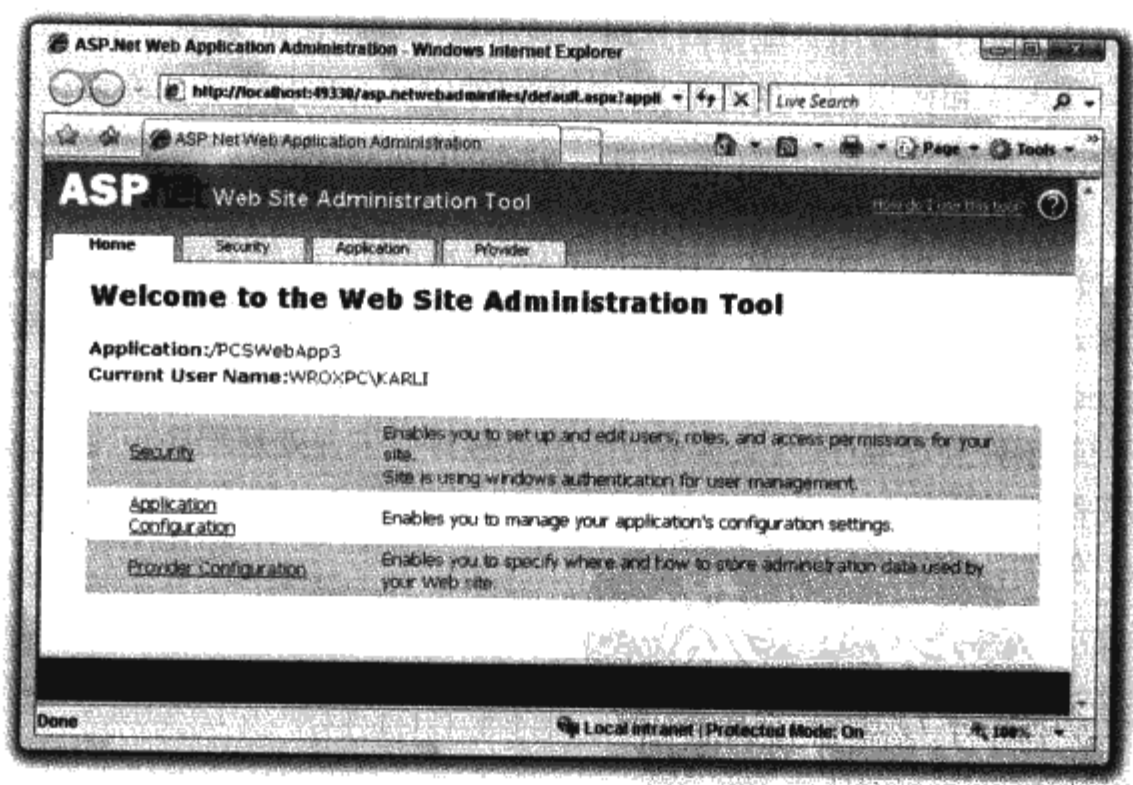


图 37-15

从文本中可以看出，这个工具可以配置许多设置，包括安全性。下一章将详细介绍这个工具。



## 37.5 小结

本章概述如何利用 ASP.NET 创建 Web 应用程序，介绍如何使用 C# 和 Web 服务器控件提供一个优秀的开发环境。我们还开发了一个会议登记应用程序，阐述了许多技术，例如各种服务器控件、ADO.NET 的数据绑定。

主要内容如下：

- ASP.NET 概述，以及它与 .NET 开发环境的配合使用
- ASP.NET 的基本语法，状态的管理，把 C# 代码集成到 ASP.NET 页面中
- 使用 Visual Studio 创建 ASP.NET Web 应用程序，存储和测试 Web 站点的选项
- ASP.NET 开发人员可以使用的 Web 控件，它们如何传送动态或数据驱动的内容
- 使用事件处理程序检测并执行用户与控件的交互操作，通过页面和显示事件定制控件
- 把数据绑定到 Web 控件上，使用模板和数据绑定表达式格式化要显示的数据
- 综合运用这些技巧，建立会议室登记应用程序

掌握了这些知识，就可以建立功能强大的 Web 应用程序了。但这只涉及 ASP.NET 的皮毛。在开始 Web 开发之前，应先了解更多的信息。第 38 章将扩展 ASP.NET 知识，学习更重要的 Web 主题，包括 master 页面、skinning 和个性化。

# 第38章

## ASP.NET 开 发

在进行 Web 开发时通常会出现这样的情况：即可用的工具的功能虽然强大，但不符合具体项目的需求，可能是给定控件的工作方式并不像所期望的那样，也可能是一部分代码本来的目的是能够在多个页面上重用，但是许多开发人员实现起来却相当复杂。在这些情况下，定制控件的建立就尤为迫切。简言之，定制控件可以把多个现有的控件包装在一起，这些现有控件还可能有指定布局的额外属性；定制控件也可以与现有的控件完全不同。使用定制控件与使用 ASP.NET 中的控件一样简单，能使 Web 站点的编码非常容易。

本章的第一部分将介绍控件开发人员可用的选项，并编写一个简单的用户控件，我们还将介绍构建高级控件的基础知识，但不详细探讨它们，这些主题需要一本书的篇幅来讨论。

接着介绍 Master 页面，这是 ASP.NET 2.0 的一个新技术，可以为 Web 站点提供模板。使用 Master 页面可以在 Web 站点上通过大量的重用代码，实现复杂的 Web 页面布局。本章还将说明如何使用 Web 导航服务器控件和 Master 页面，提供 Web 站点上一致的导航布局。

站点导航可以把用户分为不同的组，只允许某些用户(注册到站点上的用户，或站点管理员)访问某些部分。本章还将介绍 Web 站点的安全性和登录，通过 Web 登录服务器控件很容易实现该功能。

之后介绍一些高级样式设置技巧，即提供和选择 Web 站点的主题，主题把 Web 页面的显示与其功能分隔开。我们可以为站点提供 CSS 样式表，给 Web 服务器控件提供不同的样式。

最后使用 Web Part 定位和定制页面上的控件，让用户动态地个性化 Web 页面。

本章的主要内容如下：

- 用户控件和定制控件
- Master 页面
- 站点导航
- 安全性
- 主题
- Web Part

本章将开发一个大型示例应用程序，它包含本章和上一章介绍的所有技术。这个应用程序 PCSDemoSite 在本章的下载代码中。要包含所有的代码，会使本章非常长，但在学习其中的技术之前不需要运行它。相关的代码段会在需要时列出，其他代码(大多数是前面介绍的内容或简单代码)请读者自学。

## 38.1 用户控件和定制控件

在过去,实现定制控件是非常复杂的,尤其在大型系统中,由于使用定制控件需要复杂的注册过程,因此定制控件的实现就更为复杂。即使在简单的系统上,创建定制控件所需进行的编码也是一个相当复杂的过程。老版本 Web 语言的脚本编码功能也不能对手工编写的对象模型提供较好的访问,因此各个方面的性能都比较差。

.NET Framework 使用简单的编程技术,为定制控件的创建提供了一个理想的设置。ASP.NET 服务器控件的每个方面都可以随意定制,包括模板制作、客户端脚本编码等功能。但是,也不必为所有这些功能编写代码;控件越简单,创建就越容易。

另外,.NET 系统中固有的程序集动态查询功能使 Web 应用程序在新 Web 服务器上的安装如同复制包含代码的目录结构一样简单。要使用自己创建的控件,只需复制包含这些控件的程序集和其他代码即可。甚至可以把频繁使用的控件放在 Web 服务器上一个位于全局程序集缓存器(GAC)的程序集中,这样服务器上所有的 Web 应用程序就可以访问它们了。

本章将介绍两类不同的控件:

- 用户控件——即把现有的 ASP.NET 页转化为控件
- 定制控件——即组合几个控件的功能、扩展现有的控件以及从头创建新的控件

我们将创建一个简单的控件,显示一副扑克牌(黑桃、方块、红桃和梅花),以便轻松地把它嵌入到其他 ASP.NET 页面中,以此来阐明用户控件的用法。对于定制控件,不打算详细介绍,只探讨基本规则和查找定制控件的地址。

### 38.1.1 用户控件

用户控件是用 ASP.NET 代码创建的控件,就像在标准的 ASP.NET Web 页面中创建控件一样,不同之处在于一旦创建了用户控件,就可以轻松地在多个 ASP.NET 页面中重用它们。

例如,假定已经创建了一个显示数据库中信息的页面,信息也许是关于订单的,就不必创建一个固定的页面去显示信息,而可以把相关的代码放到用户控件中,然后把该控件插入到任意多个不同的 Web 页面中。

此外,可以给用户控件定义属性和方法,例如,可以指定 Web 页面上显示数据库表时的背景色属性,或者指定一个方法,重新进行数据库查询,以检查数据库中的变化。

下面创建一个简单的用户控件,与其他章节一样,本章的示例项目也可以从 Wrox 网站 [www.wrox.com](http://www.wrox.com) 上下载。

#### 一个简单的用户控件

在 Visual Studio 中,在 C:\ProCSharp\Chapter38 目录下创建一个新 Web 站点 PCSUserC-WebApp1。一旦生成标准文件,就可以选择 Website | Add New Item...菜单选项,添加名称为 PCSUserC1.ascx 的 Web 用户控件,如图 38-1 所示。

给项目添加的文件的扩展名为 .ascx 和 .ascx.cs,它们的工作方式与前面的 .aspx 文件非常相似。.ascx 文件包含 ASP.NET 代码,看起来与普通的 .aspx 文件非常相似。.ascx.cs 文件是后台代

码文件，它为用户控件定义了定制代码，定义的方式与在.aspx.cs 文件中定义窗体的方式一样。

与.aspx 文件相似，也可以在设计视图或源代码视图中查看.ascx 文件。在源代码视图中查看文件，可以发现一个重要的区别：.ascx 文件没有显示 HTML 代码，特别是没有<form>元素，原因在于：用户控件要插入到其他文件的 ASP.NET 窗体中，因此不需要自己的<form>标记。生成的代码如下所示。

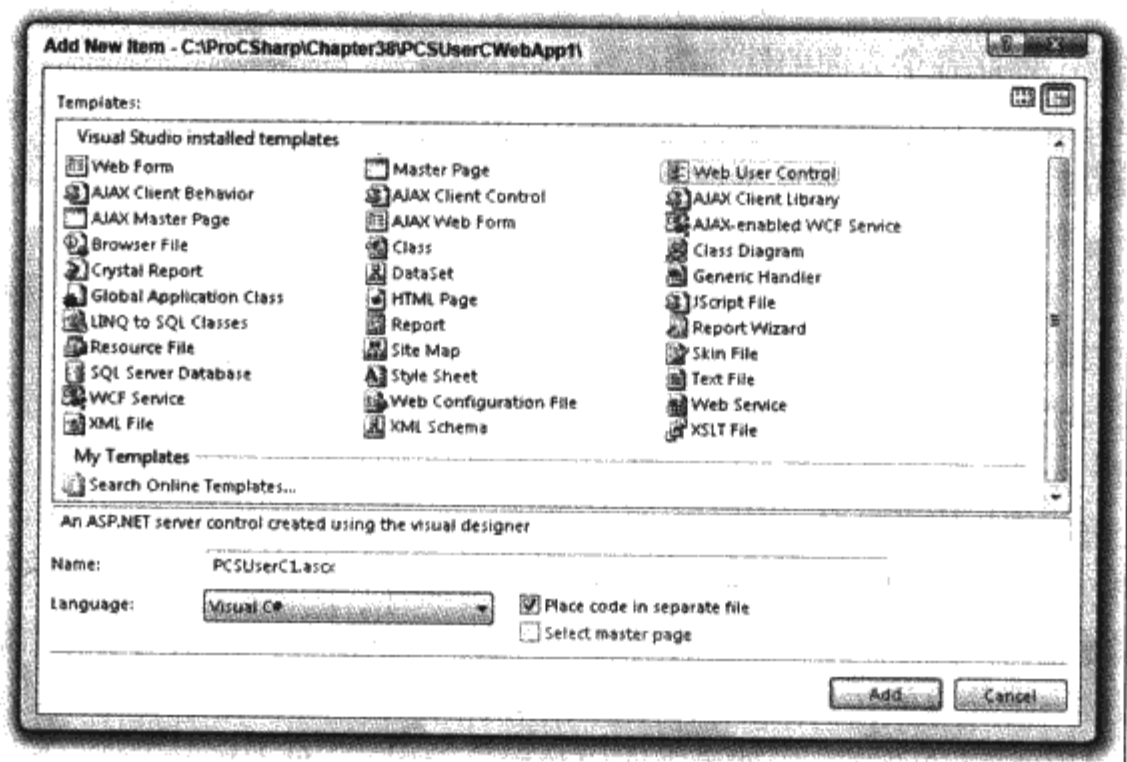


图 38-1

```
<%@ Control Language="c#" AutoEventWireup="true" CodeFile="PCSUserC1.ascx.cs"
    Inherits="PCSUserC1" %>
```

这非常类似于在.aspx 文件中生成的<%@ Page %>指令，但指定了 Control，而不是 Page，CodeFile 属性指定了后台代码文件，Inherits 指定了在后台代码文件中页面继承的类名。.ascx.cs 文件中生成的代码与自动生成的.aspx.cs 文件一样，其中包含一个空的类定义和 Page\_Load() 事件处理程序。

本例的简单控件是一个显示图形的控件，显示的图形对应于扑克牌中的花色(即梅花、方块、红桃和黑桃)。这里所需的图形是 Visual Studio 附带的图形；它们在本章的下载代码中，位于 CardSuitImages 目录，其文件名分别是 CLUB.BMP、DIAMOND.BMP、HEART.BMP 和 SPADE.BMP。把这些图形文件复制到项目目录的新子目录 Images 中，以便在后面使用它们。如果不能访问这个目录，可以使用其他任意图形，因为它们对于代码的功能而言并不重要。

#### 注意：

与 Visual Studio 的以前版本不同，在 Visual Studio 外部对 Web 站点结构进行的修改会自动反映到 IDE 上。只需单击 Solution Explorer 窗口中的 Refresh 按钮，就会看到新的 Images 目录和位图文件。

给新控件添加一些代码。在 PCSUserC1.ascx 的 HTML 视图添加下列代码：

```
<%@ Control Language="c#" AutoEventWireup="true" CodeFile="PCSUserC1.ascx.cs"
    Inherits="PCSUserC1" %>
<table cellpadding="4">
```

```

<tr valign="middle">
  <td>
    <asp:Image Runat="server" ID="suitPic" ImageURL="~/Images/club.bmp"/>
  </td>
  <td>
    <asp:Label Runat="server" ID="suitLabel">Club</asp:Label>
  </td>
</tr>
</table>

```

这段代码定义了控件的默认状态,即一个梅花图形和一个标签。图像路径前面的~表示“从 Web 站点的根目录开始”。在给控件添加功能之前,先把这个控件添加到项目的 Web 页面 WebForm1.aspx 上,测试这个默认状态。

为了在.aspx 文件中使用定制的控件,首先需要指定如何引用该控件,也就是说,如何在 HTML 中引用代表控件的标记名称。为此,在 Default.aspx 中代码的顶部使用<%Register%>指令,如下所示。

```
<% Register TagPrefix="PCS" TagName="UserC1" Src="PCSUserC1.ascx" %>
```

属性 TagPrefix 和 TagName 指定要使用的标记名称(指定的格式为<TagPrefix:TagName>),使用属性 Src 指向包含用户控件的文件。现在,添加下面的元素,就可以使用控件了:

```

<form id="Form1" method="post" runat="server">
  <div>
    <PCS:UserC1 Runat="server" ID="myUserControl"/>
  </div>
</form>

```

这就是测试用户控件需要做的所有工作,运行项目的结果如图 38-2 所示。

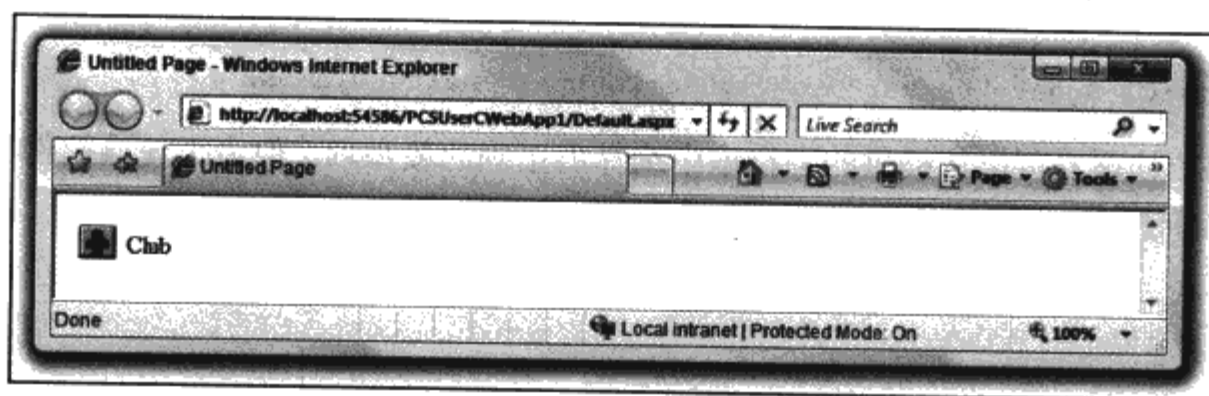


图 38-2

可以看出,这个控件在表格布局中组合了两个现有的控件,即图形控件和标签控件,因此它属于合成控件一类。

为了控制显示的花色图形,可以在元素<PCS:UserC1>上使用属性。用户控件元素上的属性会自动映射到用户控件的特性上,因此,只需给控件的后台代码 PCSUserC1.ascx.cs 添加一个特性。这个特性称为 Suit,让它接收合适的花色值。为了便于表示控件的状态,可以定义一个枚举,来保存 4 个花色名称。最佳方式是在 Web 站点上添加一个目录 App\_Code(App\_Code 是另一个“特殊”的目录,与 App\_Data 一样,它的功能取决于编程人员,这里是为 Web 应用程序保存其他代码文件。要添加这个目录,可以右击 Solution Explorer 中的 Web site,单击 Add ASP.NET Folder App\_Code),然后在这个目录中添加一个.cs 文件 suit.cs,其代码如下:



```
using System;
```

```
public enum suit
{
    club, diamond, heart, spade
}
```

类 PCSUserC1 需要一个成员变量，以保存花色类型 currentSuit:

```
public partial class PCSUserC1 : System.Web.UI.UserControl
{
    protected suit currentSuit;
```

再添加一个访问这个成员变量的属性 Suit:

```
public suit Suit
{
    get
    {
        return currentSuit;
    }
    set
    {
        currentSuit = value;
        suitPic.ImageUrl = "~/Images/" + currentSuit.ToString() + ".bmp";
        suitLabel.Text = currentSuit.ToString();
    }
}
```

这里的 set 存取器把图形的 URL 设置为前面复制的一个文件，并把要显示的文本设置为花色名称。

下面需要给 Default.aspx 添加代码以访问这个新的属性。使用刚才添加的属性选择花色:

```
<PCS:UserC1 Runat="server" id="myUserControl" Suit="diamond"/>
```

ASP.NET 处理器可以从提供的字符串中选择出正确的枚举项。但为了使该控件更有趣、更吸引人，下面使用一个单选按钮列表来选择花色:

```
<form id="Form1" runat="server">
    <div>
        <PCS:UserC1 Runat="server" ID="myUserControl"/>
        <asp:RadioButtonList Runat="server" ID="suitList" AutoPostBack="True">
            <asp:ListItem Value="club" Selected="True">Club</asp:ListItem>
            <asp:ListItem Value="diamond">Diamond</asp:ListItem>
            <asp:ListItem Value="heart">Heart</asp:ListItem>
            <asp:ListItem Value="spade">Spade</asp:ListItem>
        </asp:RadioButtonList>
    </div>
</form>
```

还需要给列表的 SelectedIndexChanged 事件添加事件处理程序。双击设计视图中的单选按钮列表，就可以添加处理程序。

**注意:**

把列表的 Autopostback 属性设置为 true，是因为除非进行回送操作，否则将不在服务器上执行 suitList\_SelectedIndexChanged 事件处理程序，在默认状态下，这个控件也不会触发回送操作。

在 Default.aspx.cs 中, 方法 `suitList_SelectedIndexChanged()` 需要以下代码:

```
public partial class Default
{
    protected void suitList_SelectedIndexChanged(object sender, EventArgs e)
    {
        myUserControl.Suit = (suit)Enum.Parse(typeof(suit),
            suitList.SelectedItem.Value);
    }
}
```

我们知道, 元素 `<ListItem>` 上的 `value` 属性代表前面定义的枚举 `suit` 的有效值, 因此简单地把这些值解析为枚举类型, 并把它们用作用户控件的 `Suit` 属性值。使用简单的数据类型转换语法, 就可以把返回的对象类型转换为 `suit`, 因为这个类型不能通过隐式转换而得到。

在运行 Web 应用程序时, 可以改变花色, 如图 38-3 所示。

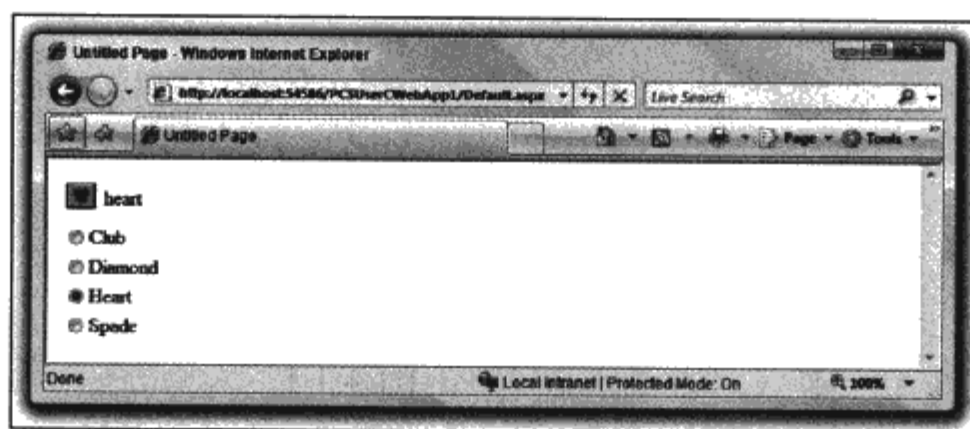


图 38-3

接下来, 给控件添加一些方法。这是非常简单的, 只需给 `PCSUserC1` 类添加方法就可以了:

```
public void Club()
{
    Suit = suit.club;
}

public void Diamond()
{
    Suit = suit.diamond;
}

public void Heart()
{
    Suit = suit.heart;
}

public void Spade()
{
    Suit = suit.spade;
}
```

4 个方法 `Club()`、`Diamond()`、`Heart()` 和 `Spade()` 分别用于改变显示在屏幕上的扑克牌的花色。在 .aspx 页面上的 4 个 `ImageButton` 控件上调用这些函数:

```
</asp:RadioButtonList>
<asp:ImageButton Runat="server" ID="clubButton"
    ImageUrl="~/Images/CLUB.BMP" OnClick="clubButton_Click" />
```

```
<asp:ImageButton Runat="server" ID="diamondButton"
    ImageUrl="~/Images/DIAMOND.BMP" OnClick="diamondButton_Click" />
<asp:ImageButton Runat="server" ID="heartButton"
    ImageUrl="~/Images/HEART.BMP" OnClick="heartButton_Click" />
<asp:ImageButton Runat="server" ID="spadeButton"
    ImageUrl="~/Images/SPADE.BMP" OnClick="spadeButton_Click" />
</div>
</form>
```

事件处理程序如下：

```
protected void clubButton_Click(object sender, ImageClickEventArgs e)
{
    myUserControl.Club();
    suitList.SelectedIndex = 0;
}

protected void diamondButton_Click(object sender, ImageClickEventArgs e)
{
    myUserControl.Diamond();
    suitList.SelectedIndex = 1;
}

protected void heartButton_Click(object sender, ImageClickEventArgs e)
{
    myUserControl.Heart();
    suitList.SelectedIndex = 2;
}

protected void spadeButton_Click(object sender, ImageClickEventArgs e)
{
    myUserControl.Spade();
    suitList.SelectedIndex = 3;
}
```

注意：

这 4 个按钮可以使用一个事件处理程序，因为它们有相同的方法签名，都要通过传送给 sender 的值检测按下了哪个按钮，动态确定应调用 myUserControl 的哪个方法，动态设置哪个索引。但与需要的代码量相比，其性能差异不是很大，所以为了简单起见，还是把它们分开，放在 4 个事件处理程序中。

既然有了 4 个新按钮，就可以改变扑克牌的花色，如图 38-4 所示。

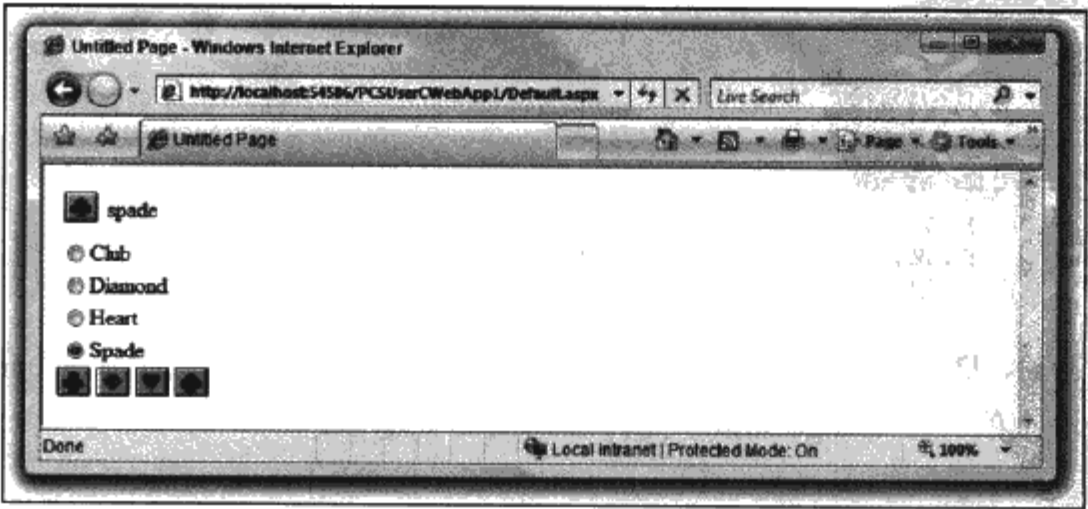


图 38-4

完成了用户控件的创建后，使用<%@Register%>指令和为控件创建的两个源代码文件 (PCSUserC1.ascx 和 PCSUserC1.ascx.cs)，就可以在其他 Web 页面中使用这个用户控件了。

38.1.2 PCSDemoSite 中的用户控件

在 PCSDemoSite 中，要把上一章的 Meeting Room Booker 应用程序转换为用户控件，以便于重用。为了查看这个控件，必须以 User1 的身份，用密码 User1!!登录站点(本章后面将介绍系统的登录)，然后导航到 Meeting Room Booker 页面，如图 38-5 所示。

除了样式上有显著的变化之外，本章后面的主题还对该页面进行了如下大的改动：

- 用户名自动从用户信息中获取
- 在页面底部不显示额外的数据，后台代码文件中也相应地删除了 DataBind()调用。
- 控件的下面没有显示得到的标签，用户查看在日历中添加的会议和会议列表，就可以获得足够多的反馈，无需报告会议的添加是否成功。
- 包含用户控件的页面使用了 Master 页面。

要进行这些修改，代码的改动其实很简单，但这里不介绍它们。本章后面在进行登录时还会介绍这个控件。

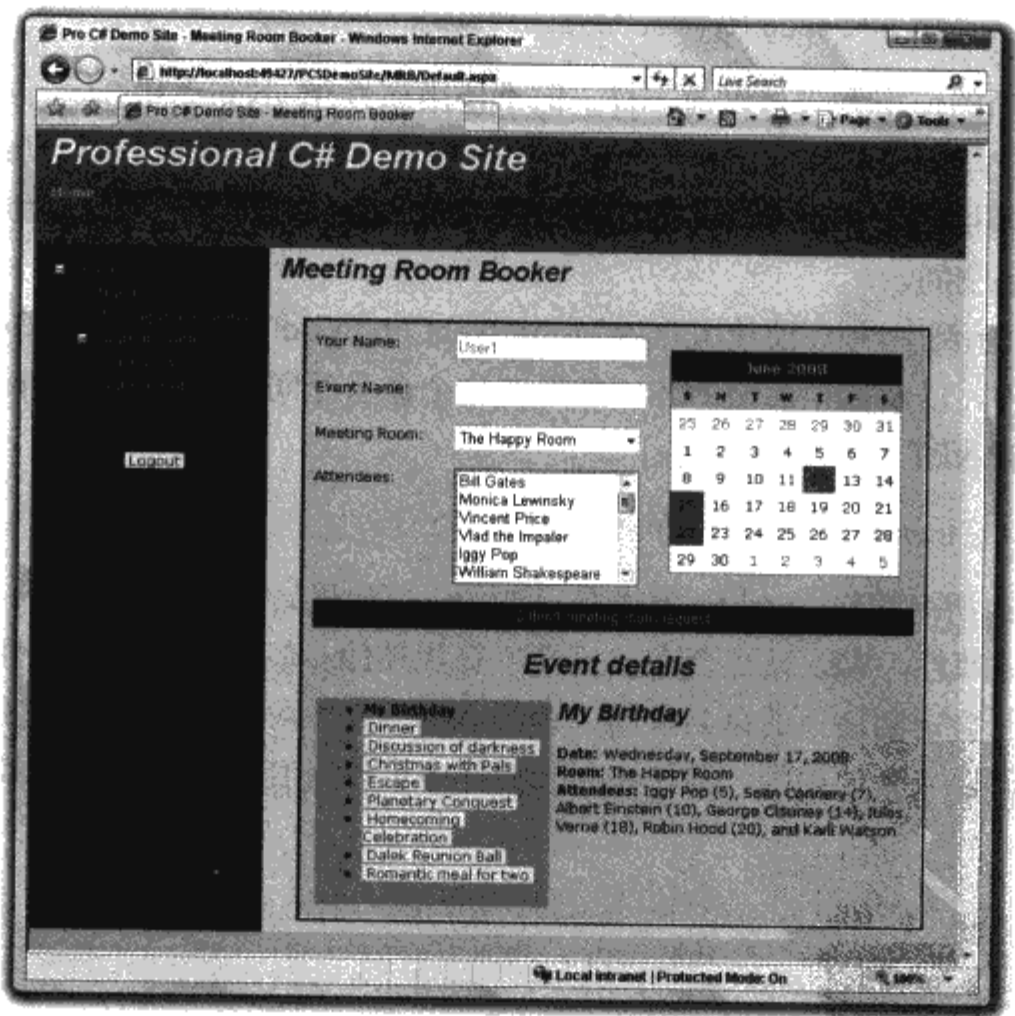


图 38-5

38.1.3 定制控件

与用户控件相比，定制控件又进了一步，原因是定制控件完全包含在 C#程序集中，不需要



单独的 ASP.NET 代码。这意味着不需要在 .ascx 文件中组装 UI。相反，完全可以控制输出的内容，即控件所生成的 HTML。

一般情况下，开发定制控件要比开发用户控件花费的时间更长，原因是定制控件的语法更复杂一些，通常需要编写更多的代码才能得到结果。在开发用户控件时，只需简单地把几个现有的控件组合在一起，而开发定制控件就不那么简单了。

为了使定制控件拥有最可定制化的功能，可以从 `System.Web.UI.WebControls.WebControl` 派生一个类，这样就创建了一个完全定制控件。此外，可以扩展现有控件的功能，创建一个派生的定制控件。最后，可以像上一节那样把几个现有的控件组合在一起，但要使用更有逻辑的结构，创建一个合成的定制控件。

以上述 3 种方法创建的定制控件都可以按同一种方式在 ASP.NET 页面中使用。我们只需把生成的程序集放在 Web 应用程序可以找到的地方，并使用 `<%Register%>` 指令注册要使用的元素名称。这里“Web 应用程序可以找到的地方”有两种情况：可以把程序集放在 Web 应用程序的 bin 目录下，如果希望服务器上的所有 Web 应用程序都可以访问它，就可以把它放在 GAC 中。如果只在一个 Web 站点上使用用户控件，就可以把控件的 .cs 文件放在站点的 App\_Code 目录下。

`<%Register%>` 指令的语法在定制控件中有一些变化：

```
<%@ Register TagPrefix="PCS" Namespace="PCSCustomWebControls"
    Assembly="PCSCustomWebControls"%>
```

TagPrefix 选项的使用方式与以前一样，但不使用属性 TagName 和 Src，原因是所使用的定制控件程序集包含几个定制控件，每一个定制控件都以其类名来命名，所以 TagName 就变得多余了。此外可以使用 .NET Framework 的动态查找功能去查找程序集，方法是给出程序集的名称和包含控件的命名空间。

在上面的代码示例中，程序要使用 PCSCustomWebControls.dll 程序集和 PCSCustomWebControls 命名空间中的控件，以及标记前缀 PCS。如果在这个命名空间中有名为 Control1 的控件，则可以通过下面的 ASP.NET 代码去使用它：

```
<PCS:Control1 Runat="server" ID="MyControl1"/>
```

`<%Register%>` 指令的 Assembly 属性是可选的——如果站点的 App\_Code 目录下有定制控件，就可以忽略该属性，Web 站点会在该目录下查找控件。但 Namespace 属性不是可选的，必须在代码文件中包含定制控件的命名空间，否则 ASP.NET 运行库就找不到它们。

定制控件是可以嵌套使用的，例如在列表控件中可以嵌套 `<asp:ListItem>` 控件，以填充该列表控件：

```
<asp:DropDownList ID="roomList" Runat="server" Width="160px">
    <asp:ListItem Value="1">The Happy Room</asp:ListItem>
    <asp:ListItem Value="2">The Angry Room</asp:ListItem>
    <asp:ListItem Value="3">The Depressing Room</asp:ListItem>
    <asp:ListItem Value="4">The Funked Out Room</asp:ListItem>
</asp:DropDownList>
```

可以以类似的方式创建一些控件，把它们解释为其他控件的子控件。这是比较高级的技术，本章不探讨。



## 定制控件示例

下面将理论用于实践。我们将使用 C:\ProCSharp\Chapter38\目录下的一个 Web 站点 PCSCustomCWebApp1, 在其 App\_Code 目录下有一个定制控件, 这个控件是已有 Label 控件的彩色版本, 可以给文本中的每个字母设置不同的颜色。

RainbowLabel 控件的代码在 App\_Code\Rainbow.cs 文件中, 首先是下述 using 语句:

```
using System;
using System.Data;
using System.Configuration;
using System.Linq;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
using System.Xml.Linq;
using System.Drawing;
```

除了 System.Drawing 之外, 这些都是把类文件添加到 Web 站点时的默认命名空间。System.Drawing 命名空间用于 Color 枚举。该文件的类在私有数组 Colors 中包含一个私有颜色数组, 这些颜色用于文本中的字母:

```
namespace PCSCustomWebControls
{
    public class RainbowLabel : Label
    {
        private Color[] colors = new Color[] {Color.Red, Color.Orange,
                                                Color.Yellow,
                                                Color.GreenYellow,
                                                Color.Blue, Color.Indigo,
                                                Color.Violet};
```

注意 PCSCustomWebControls 命名空间用于包含控件。如前所述, 这是一个必须的命名空间, 有了它, Web 页面才能正确引用控件。

为了迭代颜色, 还要在私有属性 offset 中存储一个偏移整数值:

```
private int offset
{
    get
    {
        object rawOffset = ViewState["_offset"];
        if (rawOffset != null)
        {
            return (int)rawOffset;
        }
        else
        {
            ViewState["_offset"] = 0;
            return 0;
        }
    }
    set
    {
        ViewState["_offset"] = value;
```

```
    }
}
```

注意这个属性只是在一个成员字段中存储了一个值，这是 ASP.NET 维护状态的方式，如上一章所述。控件在每个回送操作中都会实例化，所以要存储值，必须使用视图状态。为了易于访问，只需使用 `ViewState` 集合，它可以存储能串行化的任意对象。如果没有这么做，在每次回送时，偏移值都会恢复其初始值。

要修改偏移值，可以使用 `Cycle()` 方法：

```
public void Cycle()
{
    offset = ++offset;
}
```

这个方法只是递增在视图状态中为 `offset` 存储的值。

最后，重写定制控件最重要的方法 `Render()`。在这个方法中，要输出 HTML，而且这是一个实现起来非常复杂的方法。如果考虑显示控件的所有浏览器，以及可能影响显示的所有变量，这个方法就会非常大。幸好，对于本例来说，这个方法相当简单：

```
protected override void Render(HtmlTextWriter output)
{
    string text = Text;
    for (int pos = 0; pos < text.Length; pos++)
    {
        int rgb = colors[(pos + offset) % colors.Length].ToArgb()
            & 0xFFFFFFFF;
        output.Write(string.Format(
            " < font color=\"#{0:X6}\" > {1} < /font > ", rgb, text[pos]));
    }
}
```

这个方法允许访问输出流，以显示定制控件的内容。只有两种情况不需要实现这个方法：

- 设计一个没有可视化表示的控件(通常称为组件)
- 从已有的控件中派生，且不需要改变其显示特性。

定制控件还可以有定制方法、引发定制事件、响应子控件等。对于 `RainbowLabel`，不需要考虑这些。

下面修改 `Default.aspx`，以显示控件，访问 `Cycle()`，如下所示：

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs"
    Inherits="_Default" %>

<%@ Register TagPrefix="pcs" Namespace="PCSCustomWebControls" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Untitled Page</title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
```

```

        <pcs:RainbowLabel runat="server" ID="rainbowLabel1"
            Text="Multicolored label!" />
        <asp:Button Runat="server" ID="cycleButton" Text="Cycle colors"
            OnClick="cycleButton_Click" />
    </div>
</form>
</body>
</html>

```

Default.aspx.cs 中需要的代码非常简单:

```

public partial class _Default : System.Web.UI.Page
{
    protected void cycleButton_Click(object sender, EventArgs e)
    {
        rainbowLabel1.Cycle();
    }
}

```

现在可以查看示例文本, 给文本中的字母循环使用不同的颜色, 如图 38-6 所示。

可以对定制控件做许多工作, 实际上, 其可能性是无穷的, 但必须进行实践, 才能发现更多的特性。

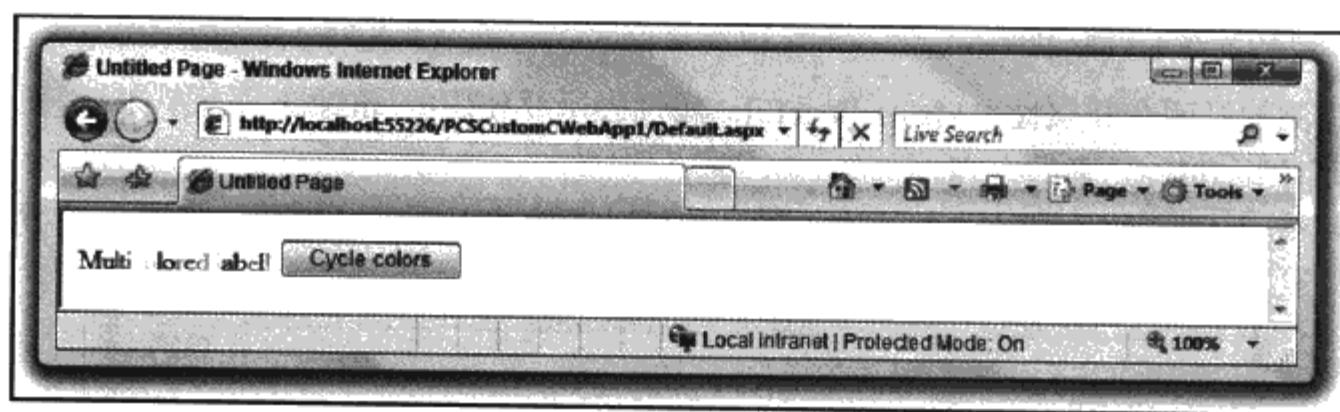


图 38-6

## 38.2 Master 页面

Master 页面可以使 Web 站点更容易设计。把所有(至少是大多数)的页面布局都放在一个文件中, 就可以为站点的各个 Web 页面考虑更重要的事情了。

Master 页面在扩展名为.master 的文件中创建, 并可以像其他站点内容那样, 通过 Website | Add New Item... 菜单项添加。初看起来, 为 Master 页面生成的代码类似于标准.aspx 页面:

```

<%@ Master Language="C#" AutoEventWireup="true" CodeFile="MyMasterPage.master.cs"
    Inherits="MyMasterPage" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Untitled Page</title>
    <asp:ContentPlaceHolder id="head" runat="server" >
    </asp:ContentPlaceHolder >
</head>

```

```
<body>
  <form id="form1" runat="server">
    <div>
      <asp:ContentPlaceHolder ID="ContentPlaceHolder1" Runat="server">
      </asp:ContentPlaceHolder>
    </div>
  </form>
</body>
</html>
```

其区别如下：

- 使用`<%@ Master %>`指令，而不是`<%@ Page %>`指令，但属性是相同的。
- 在页面标题中放置一个 ID 为 head 的 ContentPlaceHolder 控件。
- 在页面中放置一个 ID 为 ContentPlaceHolder1 的 ContentPlaceHolder 控件。

这个 ContentPlaceHolder 控件使 master 页面非常有用。在一个页面中可以有任意多个 ContentPlaceHolder 控件，它们都由使用 master 页面的.aspx 页面用于“插入”内容。可以把默认内容插入 ContentPlaceHolder 控件，但.aspx 页面会重写这个内容。

.aspx 页面要使用 master 页面，需要修改`<%@ Page %>`指令，如下所示：

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs"
Inherits="_Default" MasterPageFile="~/MyMasterPage.master"
Title="Page Title" %>
```

这里添加了两个新属性：MasterPageFile 属性表示要使用的 master 页面，Title 属性设置 master 页面中<title>元素的内容。

把一个.aspx 页面添加到 Web 站点上时，就可以选择使用 master 页面，如图 38-7 所示。

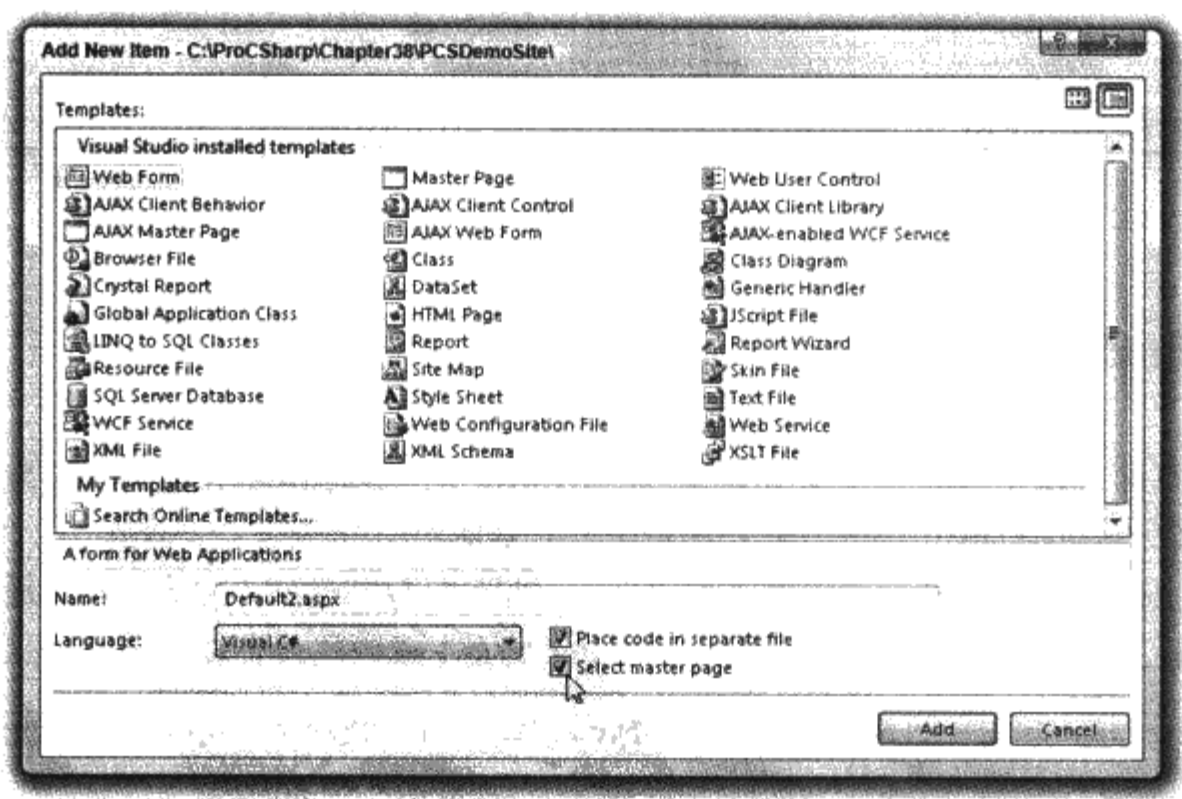


图 38-7

如果选择使用 master 页面，就可以进入站点结构，查找要使用的 master 页面，如图 38-8 所示。

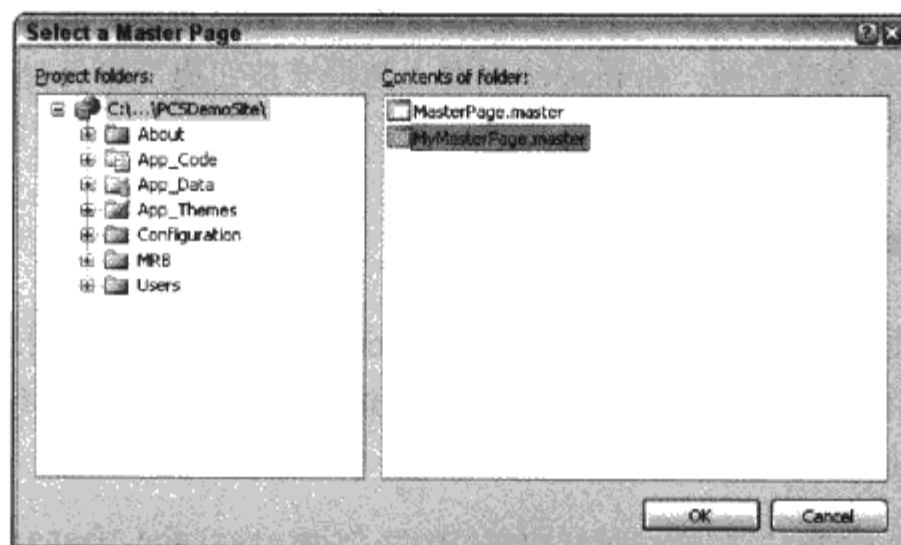


图 38-8

如果要使用默认的 master 页面内容, .aspx 页面就不必包含其他代码。实际上, 包含 Form 控件是错误的, 因为页面只能有一个 Form 控件, 而 master 页面已经有了一个 Form 控件。

使用 master 页面的 .aspx 页面可以包含不是指令的非根级内容、脚本元素和 Content 控件。可以有任意多个 Content 控件, 每个 Content 控件都会把内容插入 master 页面的一个 ContentPlaceHolder 控件中。唯一要注意的是, 确保 Content 控件的 ContentPlaceHolderID 属性匹配要插入内容的 ContentPlaceHolder 控件的 ID。所以, 要把内容添加到前面的 master 页面上, 只需编写下面的代码:

```
<%@ Page Language="C#" MasterPageFile="~/MyMasterPage.master" AutoEventWireup="true"
    CodeFile="Default2.aspx.cs" Inherits="Default2" Title="Untitled Page" %>
<asp:Content ID="Content1" ContentPlaceHolderID="head" Runat="Server" >
< /asp:Content >
<asp:Content ID="Content1" ContentPlaceHolderID="ContentPlaceHolder1"
    runat="Server">
    Custom content!
</asp:Content>
```

master 页面的真正强大之处在于, 把 ContentPlaceHolder 控件封装在其他页面内容中。例如导航控件、站点徽标和其他 HTML。可以为主要内容、边栏内容、脚标文本等提供多个 ContentPlaceHolder 控件。

如果不希望为特定的 ContentPlaceHolder 提供内容, 就可以忽略页面上的 Content 控件。例如, 可以从上面的代码中删除 Content1 控件, 这不会影响最终的显示结果。

### 38.2.1 在 Web 页面中访问 Master 页面

给 Web 页面添加 master 页面时, 有时需要在 Web 页面的代码中访问 master 页面。为此, 可以使用 Page.Master 属性, 它以 MasterPage 对象的形式返回对 master 页面的一个引用。可以把这个对象的类型强制转换为 master 页面的类型, 该类型由 master 页面文件定义(在上一节中, 这个类称为 MyMasterPage)。有了这个引用后, 就可以访问 master 页面类的任意公共成员了。

另外, 还可以使用 MasterPage.FindControl()方法通过标识符定位 master 页面上的控件, 以便处理 master 页面上内容占位符外部的内容。



一个典型用法是，如果定义了一个用于标准窗体的 **master** 页面，其中包含一个提交按钮，就可以在子页面上定位提交按钮，在 **master** 页面上为提交按钮提交事件处理程序。这样，就可以提供定制的验证逻辑，来响应窗体的提交了。

### 38.2.2 嵌套的 Master 页面

**Master** 页面选项 **Select** 也可以在创建新 **Master** 页面时使用。使用这个选项可以根据父 **Master** 页面创建嵌套的 **Master** 页面。例如，可以创建一个 **Master** 页面 **MyNestedMasterPage**，它使用 **MyMasterPage**：

```
< %@ Master Language="C#" MasterPageFile="~/MyMasterPage.master"
  AutoEventWireup="false" CodeFile="MyNestedMasterPage.master.cs"
  Inherits="MyNestedMasterPage" % >
< asp:Content ID="Content1" ContentPlaceHolderID="head" Runat="Server" >
  <!-- Disabled for child controls. -- >
< /asp:Content >
< asp:Content ID="Content2" ContentPlaceHolderID="ContentPlaceHolder1"
  Runat="Server" >
  First nested place holder:
  <asp:ContentPlaceHolder ID="NestedContentPlaceHolder1" runat="server" >
  </asp:ContentPlaceHolder >
  < br / >
  < br / >
  Second nested place holder:
  <asp:ContentPlaceHolder ID="NestedContentPlaceHolder2" runat="server" >
  < /asp:ContentPlaceHolder >
< /asp:Content >
```

使用这个 **Master** 页面的页面为 **NestedContentPlaceHolder1** 和 **NestedContentPlaceHolder2** 提供了内容，但不能直接访问 **MyMasterPage** 指定的 **ContentPlaceHolder** 控件。在这个例子中，**MyNestedMasterPage** 修改了 **head** 控件的内容，为 **ContentPlaceHolder1** 控件提供了一个模板。

创建一系列嵌套的 **Master** 页面，可以为页面提供其他布局，且不改变基本 **Master** 页面的某些方面。例如，根 **Master** 页面包含导航和基本布局，嵌套的 **Master** 页面可以为不同数量的列提供布局。接着在站点的页面上使用嵌套的 **Master** 页面，使不同的页面在这些布局之间快速切换。

### 38.2.3 PCSDemoSite 中的 Master 页面

在 **PCSDemoSite** 中，使用了一个 **master** 页面 **MasterPage.master**(这是 **master** 页面的默认名称)，其代码如下所示：

```
<%@ Master Language="C#" AutoEventWireup="true" CodeFile="MasterPage.master.cs"
  Inherits="MasterPage" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
  "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <link rel="stylesheet" href="StyleSheet.css" type="text/css" />
  <title></title>
</head>
<body>
```

```

<form id="form1" runat="server">
  <div id="header">
    <h1><asp:literal ID="Literal1" runat="server"
      text="<%= AppSettings.SiteTitle %>" /></h1>
    <asp:SiteMapPath ID="SiteMapPath1" Runat="server" CssClass="breadcrumb" />
  </div>
  <div id="nav">
    <div class="navTree">
      <asp:TreeView ID="TreeView1" runat="server"
        DataSourceID="SiteMapDataSource1" ShowLines="True" />
    </div>
    <br />
    <br />
    <asp:LoginView ID="LoginView1" Runat="server">
      <LoggedInTemplate>
        You are currently logged in as
        <b><asp:LoginName ID="LoginName1" Runat="server" /></b>.
        <asp:LoginStatus ID="LoginStatus1" Runat="server" />
      </LoggedInTemplate>
    </asp:LoginView>
  </div>
  <div id="body">
    <asp:ContentPlaceHolder ID="ContentPlaceHolder1" Runat="server" />
  </div>
</form>
<asp:SiteMapDataSource ID="SiteMapDataSource1" Runat="server" />
</body>
</html>

```

这里的许多控件前面都没有见过，稍后将介绍它们。这里要注意的是，<div>元素包含各个内容部分(标题、导航栏和页面体)，使用<%= AppSettings.SiteTitle %>从 Web.config 文件中获得站点标题：

```

<appSettings>
  <add key="SiteTitle" value="Professional C# Demo Site"/>
</appSettings>

```

还有 StyleSheet.css 的一个样式表链接：

```
<link rel="stylesheet" href="StyleSheet.css" type="text/css" />
```

这个 CSS 样式表包含此页面上<div>元素的基本布局信息，以及会议室登记工具控件的一部分。

```

div#header
{
  position: absolute;
  top: 0px;
  left: 0px;
  height: 80px;
  width: 780px;
  padding: 10px;
}

div#nav
{
  position: absolute;
  left: 0px;
  top: 100px;
}

```

```
width: 180px;
height: 580px;
padding: 10px;
}

div#body
{
    position: absolute;
    left: 200px;
    top: 100px;
    width: 580px;
    height: 580px;
    padding: 10px;
}

.mrbEventList
{
    width: 40%;
}
```

注意，这个样式信息不包含颜色、字体等。这是主题中的样式表要获得的信息，主题详见本章后面的内容。这里只有布局信息，例如<div>的大小。

注意：

本章的站点尽可能遵循最佳实践方式。布局使用 CSS 而不是表格，很快就会成为 Web 站点布局的业界标准，所以要认真掌握。在上面的代码中，符号#用于格式化有指定 id 属性的<div>元素，.mrbEventList 格式化有指定 class 属性的 HTML 元素。

38.3 站点导航

有 3 个 Web 导航服务器控件 SiteMapPath、Menu 和 TreeView，可以用于为 Web 站点提供 XML 站点地图。如果使用另一个站点地图提供程序，站点地图就以另一种格式提供。一旦创建了这样一个数据源，这些 Web 导航服务器控件就可以自动为用户生成位置和导航信息。稍后介绍一个 XML 站点地图示例。

也可以使用 TreeView 控件显示其他结构化数据，但使用站点地图，可以提供导航信息的另一种视图。

Web 导航服务器控件如表 38-1 所示。

表 38-1

控 件	说 明
SiteMapPath	显示路径样式的信息，允许用户查看他们在站点结构中的位置，并导航到父区域中。可以提供各种模板，例如 NodeStyle 和 CurrentNodeStyle，来定制路径信息的外观
Menu	通过 SiteMapDataSource 控件链接到站点地图信息上，可以查看完整的站点结构，其外观由模板定制
TreeView	可以在树型结构中显示层次化的数据，例如内容表。树中的节点存储在 Nodes 属性中，选中的节点存储在 SelectedNode 中。有几个事件可以在服务器端处理用户交互操作，包括 SelectedNodeChanged 和 TreeNode Collapsed。这个控件一般是数据绑定的

要为站点提供站点地图 XML 文件，可以使用 Web site | Add New Item 菜单项添加一个站点地图文件(.sitemap)。通过提供程序链接到站点地图上。默认的 XML 提供程序会在站点的根目录下查找 Web.sitemap 文件，所以，除非要使用另一个提供程序，否则就应接受所提供的默认文件名。

站点地图 XML 文件包含一个<siteMap>根元素，这个根元素包含一个<siteMapNode>元素，<siteMapNode>元素可以包含任意多个嵌套的<siteMapNode>元素。

每个<siteMapNode>元素都使用表 38-2 中的属性。

表 38-2

属 性	说 明
title	页面标题，用作站点地图中的链接文本
url	页面位置，用作站点地图中的超链接位置
roles	用户角色，允许查看菜单中的这个站点地图项
description	可选文本，用于站点地图的弹出式工具提示

站点有了 Web.sitemap 文件后，添加链接信息就只需在页面上放置如下代码：

```
<asp:SiteMapPath ID="SiteMapPath1" Runat="server" />
```

这将使用默认的提供程序和当前的 URL 位置，格式化父页面的链接列表。  
添加一个菜单或树型视图菜单需要 SiteMapDataSource 控件，这也是非常简单的：

```
<asp:SiteMapDataSource ID="SiteMapDataSource1" Runat="server" />
```

如果使用定制的提供程序，唯一的区别是，可以通过 SiteMapProvider 属性指定该提供程序 ID。还可以使用 StartingNodeOffset 删除菜单数据的上一层(例如根级的 Home 项)，使用 ShowStartingNode="False"将只删除顶级链接，使用 StartFromCurrentNode="True"表示从当前位置开始，使用 StrtingNodeUrl 会重写根节点。

只要把 DataSourceID 设置为 SiteMapDataSource 的 ID，Menu 和 TreeView 控件就可以使用这个数据源中的数据。这两个控件都包含许多样式属性，并可以设置主题，详见本章后面的内容。

PCSDemoSite 中的导航

PCSDemoSite 的站点地图如下所示：

```
<?xml version="1.0" encoding="utf-8" ?>
<siteMap>
  <siteMapNode url="~/Default.aspx" title="Home">
    <siteMapNode url="~/About/Default.aspx" title="About" />
    <siteMapNode url="~/MRB/Default.aspx" title="Meeting Room Booker"
      roles="RegisteredUser,SiteAdministrator" />
    <siteMapNode url="~/Configuration/Default.aspx" title="Configuration"
      roles="RegisteredUser,SiteAdministrator">
      <siteMapNode url="~/Configuration/Themes/Default.aspx" title="Themes"
        roles="RegisteredUser,SiteAdministrator"/>
    
```

```

</siteMapNode>
<siteMapNode url="~/Users/Default.aspx" title="User Area"
  roles="SiteAdministrator" />
<siteMapNode url="~/Login.aspx" title="Login Details" />
</siteMapNode>
</siteMap>

```

PCSDemoSite 站点使用定制的提供程序从 Web.sitemap 中获得信息，这是必需的，因为默认的提供程序会忽略 roles 属性。这个定制提供程序在 Web 站点的 Web.config 文件中定义，如下所示：

```

<configuration xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
  ...
  <system.Web>
    ...
    <siteMap defaultProvider="CustomProvider">
      <providers>
        <add name="CustomProvider"
          description="SiteMap provider which reads in .sitemap XML files."
          type="System.Web.XmlSiteMapProvider, System.Web, Version=2.0.3600.0,
            Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"
          siteMapFile="Web.sitemap" securityTrimmingEnabled="true" />
      </providers>
    </siteMap>
    ...
  </system.Web>
</configuration>

```

这个定制提供程序和默认提供程序的唯一区别是添加了 securityTrimmingEnabled="true"，它告诉提供程序，只为当前用户允许查看的节点提供数据。这种可见性是由用户的角色成员确定的，如下一节所述。

PCSDemoSite 中的 MasterPage.master 页面包含 SiteMapPath、TreeView 导航显示和一个数据源，如下所示：

```

<div id="header">
  <h1><asp:literal ID="Literal1" runat="server"
    text="<%$ AppSettings:SiteTitle %>" /></h1>
  <asp:SiteMapPath ID="SiteMapPath1" Runat="server" CssClass="breadcrumb" />
</div>
<div id="nav">
  <div class="navTree">
    <asp:TreeView ID="TreeView1" runat="server"
      DataSourceID="SiteMapDataSource1" ShowLines="True" />
  </div>
  <br />
  <br />
  <asp:LoginView ID="LoginView1" Runat="server">
    <LoggedInTemplate>
      You are currently logged in as
      <b><asp:LoginName ID="LoginName1" Runat="server" /></b>.
      <asp:LoginStatus ID="LoginStatus1" Runat="server" />
    </LoggedInTemplate>
  </asp:LoginView>
</div>
<div id="body">
  <asp:ContentPlaceHolder ID="ContentPlaceHolder1" Runat="server" />
</div>
</form>
<asp:SiteMapDataSource ID="SiteMapDataSource1" Runat="server" />

```



唯一要注意的是，给 SiteMapPath 和 TreeView 提供了 CSS 类，以便于使用主题特性。

## 38.4 安全性

在 Web 站点中，安全性和用户管理的实现常常是一个相当复杂的事情，这是有原因的，我们必须考虑许多因素，包括：

- 要实现哪种用户管理系统？用户要映射到 Windows 用户账户上吗？还是实现某种独立的管理系统？
- 如何实现登录系统？
- 是让用户在站点上注册吗？如何注册？
- 如何让一些用户只查看某些内容，只执行某些操作，而给另一些用户提供额外的权限？
- 忘记了密码，该怎么办？

在 ASP.NET 2.0 中，有一整套工具来处理这些问题，实际上，使用该工具只需几分钟就可以在站点上实现一个用户系统。我们有 3 种身份验证系统：

- Windows 身份验证：用户有 Windows 账户，一般在内联网站点或 WAN 入口处使用
- Forms 身份验证：Web 站点维护它自己的用户列表，完成自己的身份验证
- Passport 身份验证：Microsoft 提供的一个集中式身份验证服务

完整讨论 ASP.NET 中的安全性至少需要一章的篇幅，但可以快速浏览一下实现安全性所涉及的工作。这里将只讨论 Forms 身份验证，因为这是最通用的系统，能很快建立和运转起来。

实现 Forms 身份验证的最快捷方式是使用 Website | ASP.NET Configuration 工具。上一章介绍过这个工具，它有一个 Security 选项卡，其中是一个安全向导。这个向导允许选择身份验证类型、添加角色、添加用户、保护站点的各个区域。

### 38.4.1 使用安全向导添加 Forms 身份验证功能

为了便于说明，在 C:\ProCSharp\Chapter38\ 目录下创建一个新 Web 站点 PCSAuthenticationDemo。之后，打开 Website | ASP.NET Configuration 工具。进入 Security 选项卡。单击 Use the security setup wizard to configure security step by step 链接，阅读其中的信息后，单击第一步中的 Next，在第二步中，选择 From the internet，如图 38-9 所示。

单击 Next，在确认使用默认的 Advanced provider settings 提供程序存储安全信息后，单击 Next。这个提供程序的信息可以通过 Provider 选项卡配置，在 Provider 选项卡中，可以选择把信息存储到其他地方，例如 SQL Server 数据库，但选择 access 数据库将便于演示。

选择 Enable roles for this Web site，如图 38-10 所示，单击 Next。

接着，添加一些角色，如图 38-11 所示。

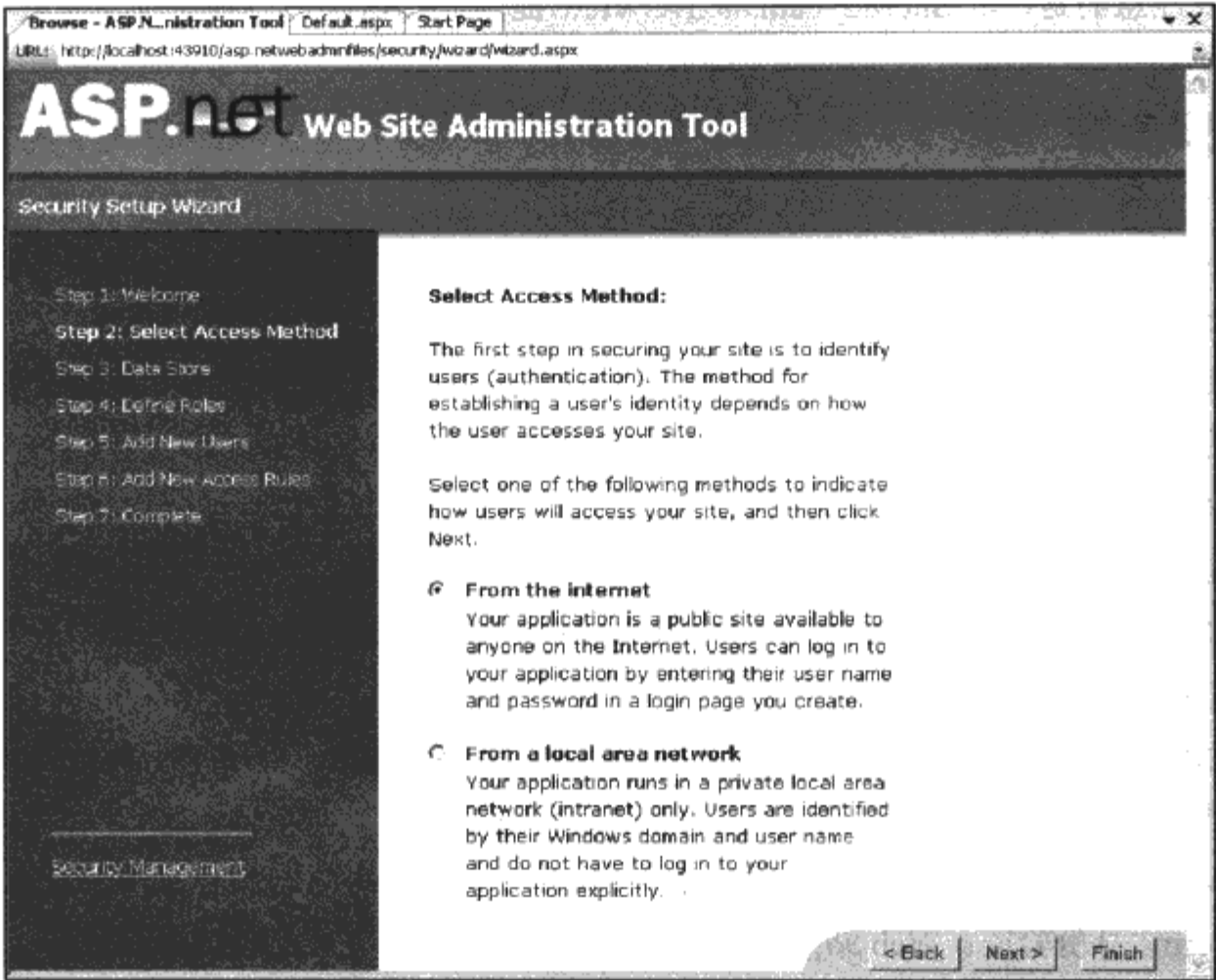


图 38-9

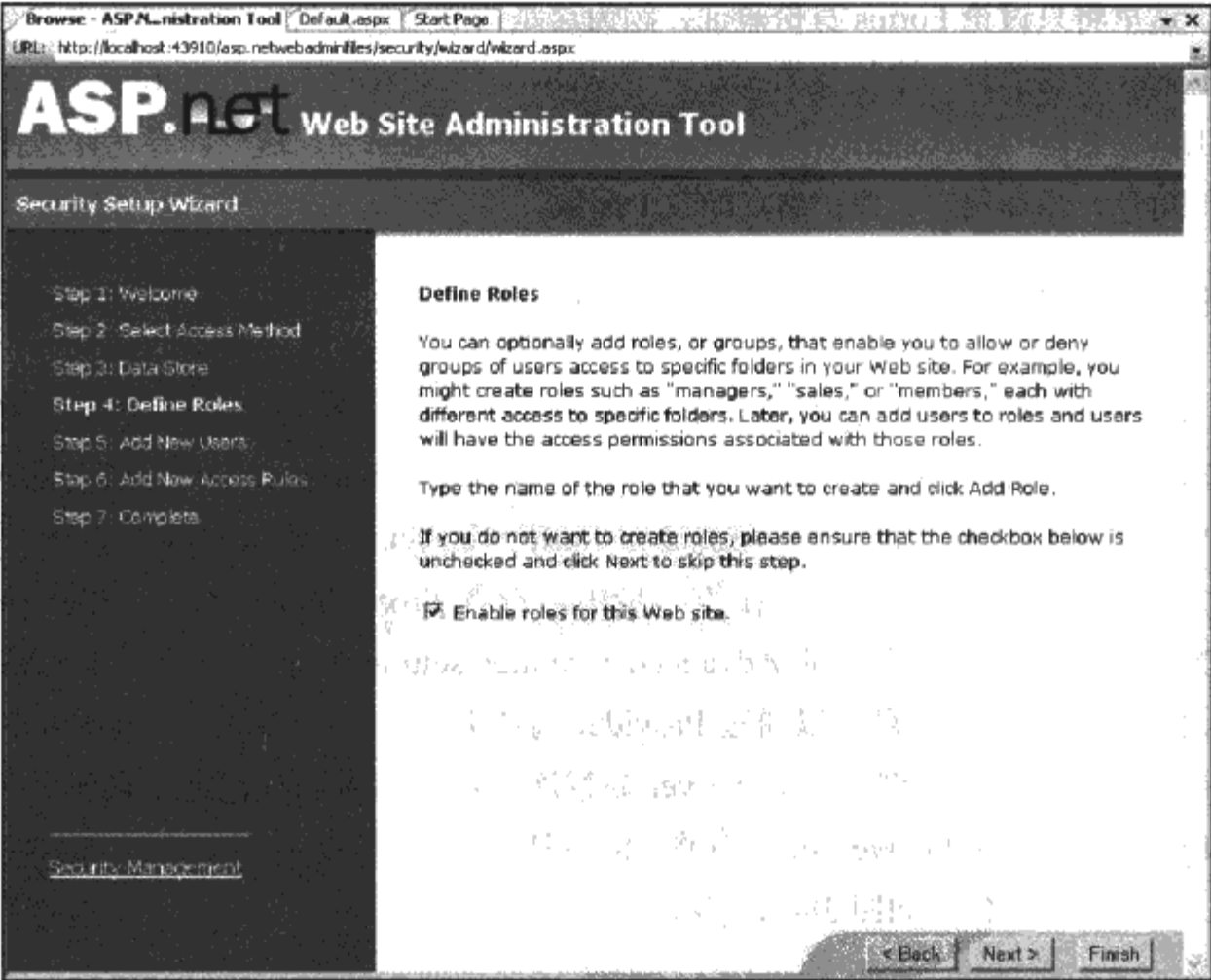


图 38-10

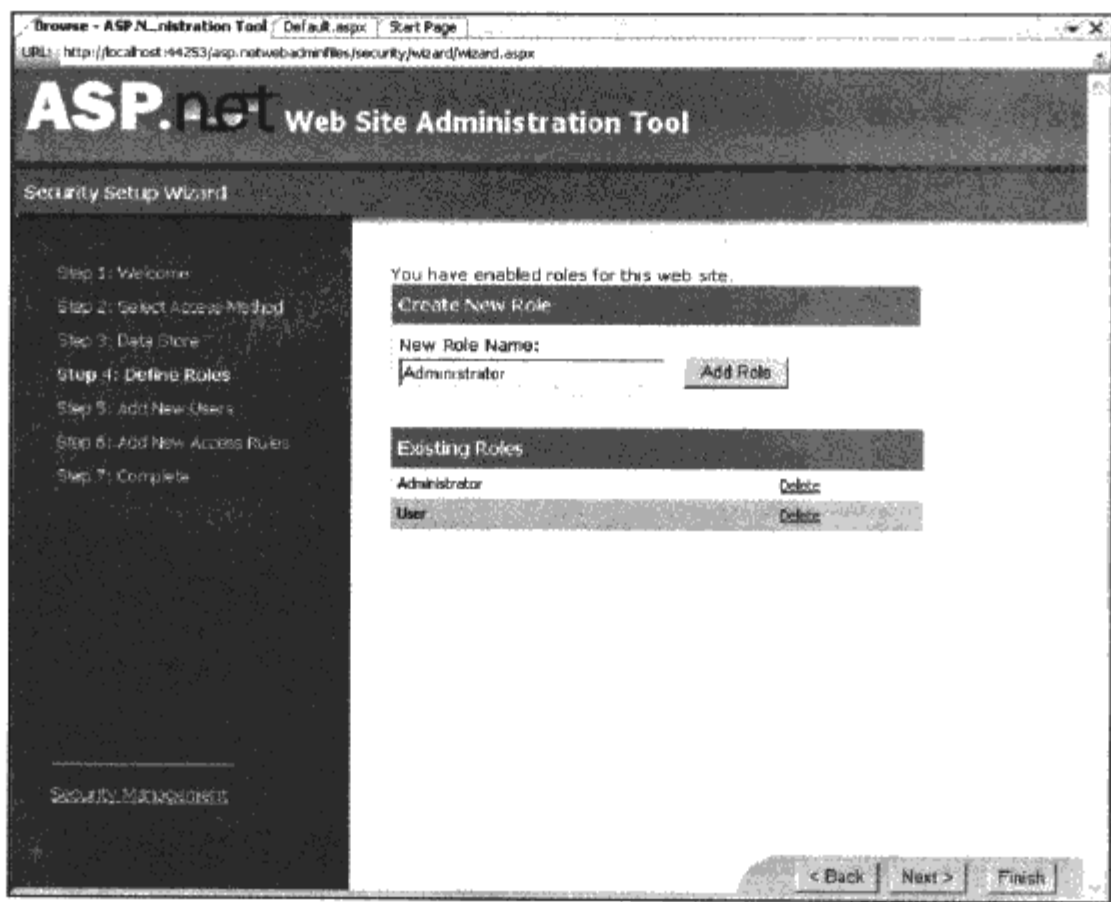


图 38-11

然后，添加一些用户，如图 38-12 所示。注意密码(在 machine.config 中定义)的默认安全角色是相当强的，密码至少有 7 个字符，至少包括 1 个符号和混合了大小写的字母。

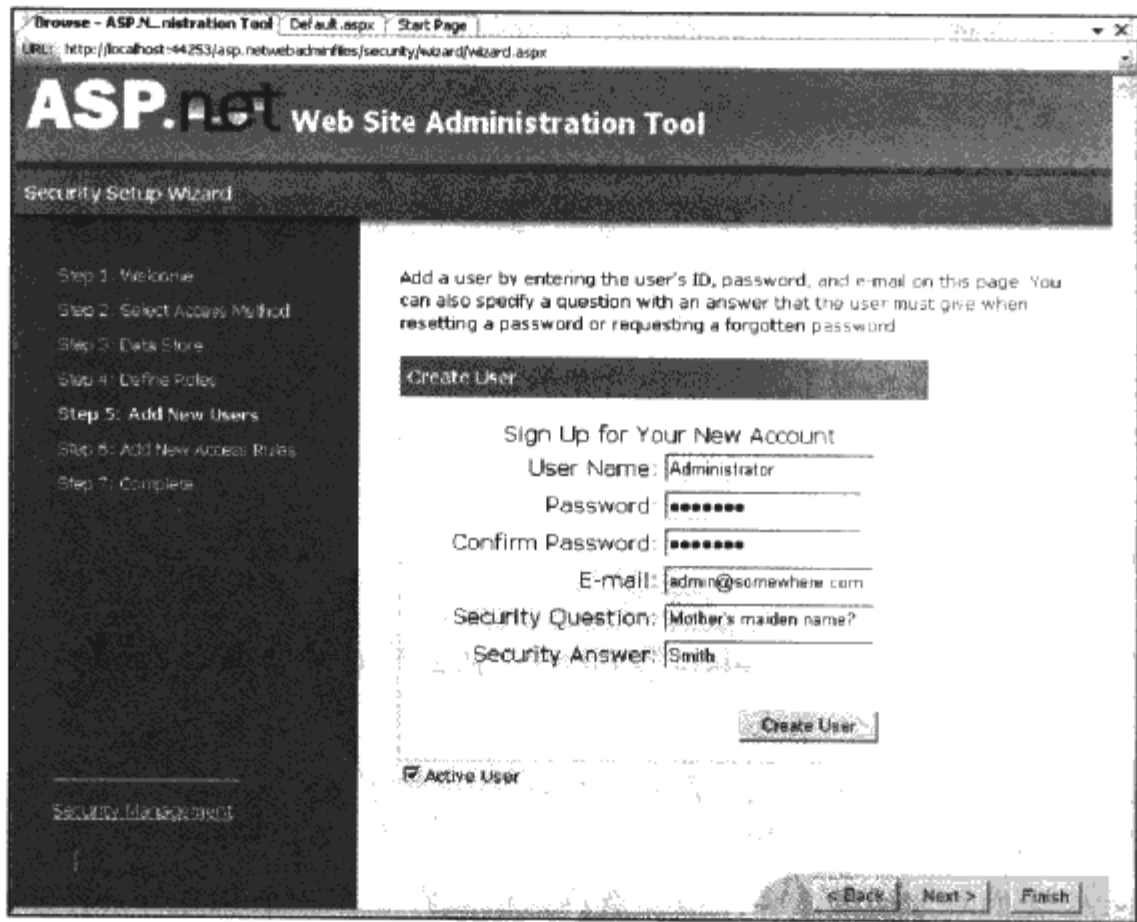


图 38-12

**提示：**  
在本章的下载代码中，为这个例子添加了两个用户。用户名分别是 User 和 Administrator，

其密码均为 Pa\$\$w0rd。

单击 Next 按钮，可以定义站点的访问规则。在默认情况下，所有的用户和角色都可以访问站点的所有区域。在这个对话框中，可以限制角色、用户或匿名用户的访问区域。可以为站点的每个目录限制访问，因为这可以通过目录中的 Web.config 文件实现，如后面所示。现在跳过这一步，完成身份验证的设置。

最后一步是把用户赋予角色，这是通过 Security 选项卡上的 Manage users 链接完成的，在该选项卡上，可以编辑用户角色，如图 38-13 所示。

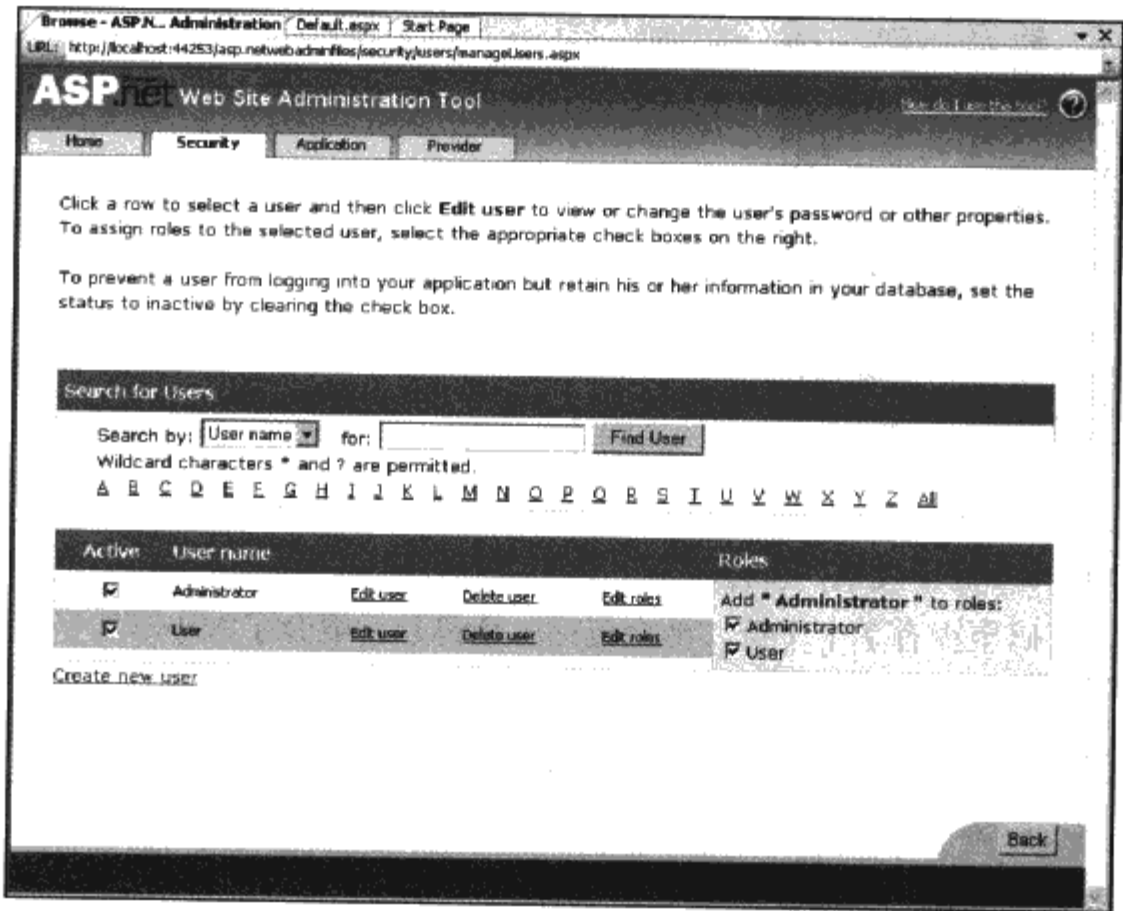


图 38-13

完成后，就有一个用户系统了，其中包含角色和用户。  
现在就可以给 Web 站点添加几个控件，进行身份验证了。

38.4.2 实现登录系统

如果在运行安全向导后刷新 Solution Explorer，就会看到一个 Web.config 文件添加到项目中，其内容如下：

```
<roleManager enabled="true" />
<authentication mode="Forms" />
```

这似乎不太像我们刚才做的工作，但注意许多信息存储在一个 SQL Express 数据库中，该数据库在 App\_Data 目录下，叫做 ASPNETDB.MDF。使用任意标准数据库管理工具都可以查看存储在这个文件中的数据，包括 Visual Studio。甚至可以直接在这个数据库中添加用户和角色。  
在默认情况下，登录系统是通过 Web 站点根目录下的一个 Login.aspx 页面实现的。如果用

户试图导航到无权访问的位置，他们就会自动重定向到这个页面，在成功登录后，就会进入希望的位置。

在 PCSAuthenticationDemo 站点中添加一个 Web 窗体 Login.aspx，把一个 Login 控件从工具箱拖放到这个窗体上。

这就是允许用户登录到 Web 站点上所需的全部工作。在浏览器上打开站点，导航到 Login.aspx，再输入在向导中添加的一个用户的信息，如图 38-14 所示。

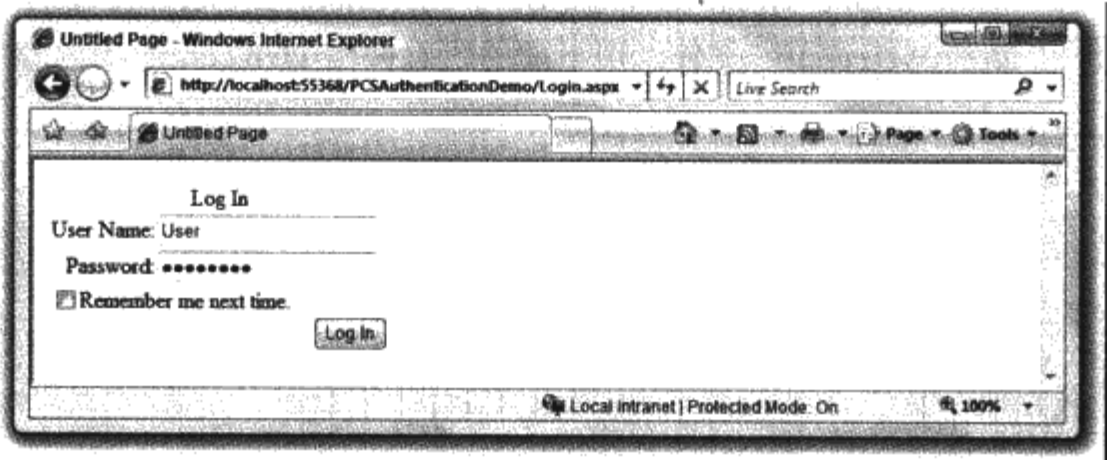


图 38-14

登录后，就会返回 Default.aspx，目前这是一个空页面。

38.4.3 Web 登录服务器控件

工具箱的 Login 部分包含几个控件，如表 38-3 所示。

表 38-3

控 件	说 明
Login	如前所示，这个控件允许用户登录 Web 站点。这个控件的大多数属性都用于给所提供的模板指定样式。还可以使用 DestinationPageUrl 强迫登录时重定向到指定的位置，使用 VisibleWhenLoggedIn 可以确定登录的用户是否能看到该控件，各种文本属性，如 CreateUserText，可以输出对用户有帮助的信息
LoginView	这个控件可以显示各种内容，这取决于用户是否登录，或用户的角色是什么。可以把内容放在<AnonymousTemplate>和<LoggedInTemplate >中，使用<RoleGroups>可以控制这个控件的输出
PasswordRecovery	这个控件允许用户把密码邮寄给他自己，可以使用为用户定义的密码恢复问题。它的大多数属性也是用于格式化显示的，但 MailDefinition- Subject 属性用于配置发送给用户地址的电子邮件，SuccessPageUrl 属性在请求用户输入密码后重定向用户
LoginStatus	显示 Login 或 Logout 链接，这取决于用户是否登录，其文本和图像都可以定制
LoginName	给当前登录的用户输出用户名
CreateUserWizard	显示一个窗体，用户可以使用该窗体注册到站点上，并添加到用户列表中。与其他登录控件一样，它也有许多与布局格式相关的属性，但默认的格式已足够好了
ChangePassword	允许用户修改密码，它有 3 个字段，一个字段用于表示旧密码，其余两个字段表示新密码和确认密码。它也有许多样式属性



这些控件将在稍后的 PCSDemoSite 中使用。

#### 38.4.4 保护目录

最后要讨论的是如何限制对目录的访问。这可以通过前面介绍的 Site Configuration 工具实现，但手工完成也很简单。

给 PCSAuthenticationDemo 添加一个目录 SecureDirectory，在这个目录中添加 Web 页面 Default.aspx 和一个新的 Web.config 文件。用下面的代码替代 Web.config 的内容：

```
<?xml version="1.0" ?>
<configuration xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
  <system.web>
    <authorization>
      <deny users="?" />
      <allow roles="Administrator"/>
      <deny roles="User" />
    </authorization>
  </system.web>
</configuration>
```

<authorization>元素可以包含一个或多个表示权限规则的<deny>或<allow>元素，每个元素都有一个 users 或 roles 属性，表示该规则应用于什么成员。规则是从上到下应用的，所以如果规则的成员关系有重叠，那么较特殊的规则一般应放在靠上的位置。在这个例子中，“？”表示匿名用户，他们和 User 角色中的用户会被拒绝访问这个目录。注意如果这里的<allow>规则放在 User 角色的<deny>规则之前，则表示只有 User 和 Administrator 角色中的用户允许访问这个目录。所有的 users 角色都考虑在内，但规则的顺序仍是有用的。

现在，在登录到 Web 站点，导航到 SecureDirectory/Default.aspx 时，只有位于 Admin 角色中，才能访问该页面。其他用户或未通过身份验证的用户都会被重定向到登录页面。

#### 38.4.5 PCSDemoSite 中的安全性

PCSDemoSite 站点使用了前面介绍的 Login 控件、LoginView 控件、LoginStatus 控件、LoginName 控件、PasswordRecovery 控件和 ChangePassword 控件。

一个区别是包含了 Guest 角色，其结果是 guest 用户不能修改其密码，这使用 LoginView 比较合适，如 Login.aspx 所示：

```
<asp:Content ID="Content1" ContentPlaceHolderID="ContentPlaceHolder1"
  Runat="server">
  <h2>Login Page</h2>
  <asp:LoginView ID="LoginView1" Runat="server">
    <RoleGroups>
      <asp:RoleGroup Roles="Guest">
        <ContentTemplate>
          You are currently logged in as <b>
            <asp:LoginName ID="LoginName1" Runat="server" /></b>
          <br />
          <br />
          <asp:LoginStatus ID="LoginStatus1" Runat="server" />
        </ContentTemplate>
      </asp:RoleGroup>
    </RoleGroups>
  </asp:LoginView>
</asp:Content>
```

```

</asp:RoleGroup>
<asp:RoleGroup Roles="RegisteredUser,SiteAdministrator">
  <ContentTemplate>
    You are currently logged in as <b>
      <asp:LoginName ID="LoginName2" Runat="server" /></b>.
    <br />
    <br />
    <asp:ChangePassword ID="ChangePassword1" Runat="server">
    </asp:ChangePassword>
    <br />
    <br />
    <asp:LoginStatus ID="LoginStatus2" Runat="server" />
  </ContentTemplate>
</asp:RoleGroup>
</RoleGroups>
<AnonymousTemplate>
  <asp:Login ID="Login1" Runat="server">
  </asp:Login>
  <asp:PasswordRecovery ID="PasswordRecovery1" Runat="Server" />
</AnonymousTemplate>
</asp:LoginView>
</asp:Content>

```

这里的视图会显示下述几个页面中的一个：

- 给匿名用户显示 Login 和 PasswordRecovery 控件。
- 给 Guest 用户显示 LoginName 和 LoginStatus 控件，如果需要，还会显示登录的用户名，并允许注销。
- 给 RegisteredUser 和 SiteAdministrator 用户显示 LoginName、LoginStatus 和 Change Password 控件。

该站点在各个目录中还包含各自的 Web.config 文件，以限制访问，也可以根据角色限制定向到其他地方。

#### 注意：

为站点配置的用户显示在“关于”页面上，也可以添加自己的已配置用户。基本站点的用户(及其密码)是 User1(User1!!)、Admin(Admin!!)和 Guest(Guest!!)。

这里要注意，站点的根目录拒绝匿名用户，但 Themes 目录(详见下一节)重写了这个设置，允许匿名用户访问。这是必需的，因为如果没有重写该设置，匿名用户就会看到没有主题设置的站点，而主题文件是不能访问的。另外，根 Web.config 文件中的完整安全规范如下：

```
<configuration xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
```

```
...
```

```
<location path="StyleSheet.css">
```

```
<system.web>
```

```
<authorization>
```

```
<allow users="?" />
```

```
</authorization>
```

```
</system.web>
```

```
</location>
```

```
<system.web>
```

```
<authorization>
```

```
<deny users="?" />
```

```
</authorization>
```

```
...
</system.web>
</configuration>
```

其中,使用<location>元素重写了用 path 属性指定的文件的默认设置,这里是 StyleSheet.css 文件。<location>元素可以把任意<system.web>设置应用于指定的文件或目录,并可以把所有与目录相关的设置集中在一个地方(替代多个 Web.config 文件)。在上面的代码中,给匿名用户授予了访问 Web 站点中根样式表的权限,这是必需的,因为这个文件在 master 页面中定义了<div>元素的布局。不这么做,为匿名用户显示登录页面的 HTML 就很难读懂。

另一个要注意的地方是,在会议室登记工具控件的代码后置文件中,有如下 Page\_Load() 事件处理程序:

```
void Page_Load(object sender, EventArgs e)
{
    if (!this.IsPostBack)
    {
        nameBox.Text = Context.User.Identity.Name;
        System.DateTime trialDate = System.DateTime.Now;
        calendar.SelectedDate = GetFreeDate(trialDate);
    }
}
```

其中,用户名是从当前的环境中提取出来的。在后台代码文件中,还要频繁使用 Context.User.IsInRole()来检查访问。

## 38.5 主题

在 ASP.NET 页面中合并使用 master 页面和 CSS 样式表后,还要把窗体和函数分开,即把页面的外观和操作方式分开定义。利用主题,可以在一步中完成这个任务,并可以把所提供的几个主题之一动态应用于页面的外观。

主题包含如下内容:

- 主题的名称
- 可选的 CSS 样式表
- 可以给各个控件类型设置样式的 Skin 文件(.skin)

这些内容以两种不同的方式应用于页面: Theme 和 StyleSheetTheme:

- Theme: 所有的 skin 属性都应用于控件,重写页面上控件已有的所有属性
- StyleSheetTheme: 已有的控件属性优先于 skin 文件中定义的属性

CSS 样式表的工作方式与方法的使用方式相同,因为它们都以标准的 CSS 方式应用。

### 38.5.1 把主题应用于页面

可以用几种声明或编程的方式把主题应用于页面,应用主题最简单的声明方式是在<%@ Page %>指令中使用 Theme 或 StyleSheetTheme 属性:

```
<%@ Page Theme="myTheme" ... %>
```

或者:

```
<%@ Page StyleSheetTheme="myTheme" ... %>
```

其中, **myTheme** 是给主题定义的名称。

另外,还可以在 Web 站点的 **Web.config** 文件中使用一项,给该站点上的所有页面指定要使用的主题:

```
<configuration xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
  <system.web>
    <pages Theme="myTheme" />
  </system.web>
</configuration>
```

这里也可以使用 **Theme** 或 **StyleSheetTheme** 属性。还可以使用 **<location>** 元素重写某个页面或目录的这个设置,其方式与上一节中给安全信息使用该元素的方式相同。

如果采用编程方式,可以在页面的后台代码文件中应用主题,但只能在 **Page\_PreInit()** 事件处理程序中应用主题,该事件在页面生存期的早期触发。在这个事件中,只需把 **Page.Theme** 或 **Page.StyleSheetTheme** 属性设置为要应用的主题名即可,例如:

```
protected override void OnPreInit(EventArgs e)
{
    Page.Theme = "myTheme";
}
```

通过代码应用主题时,可以动态应用一组主题中的一个主题文件。这个技巧将在 **PCSDemoSite** 中使用,如后面所示。

### 38.5.2 定义主题

主题在 ASP.NET 中的另一个特殊目录下定义,在这里是 **App\_Themes**。**App\_Themes** 目录可以包含任意多个子目录,每个子目录下有一个主题,子目录名就是主题名。

定义主题时,需要把主题的所有必要文件放在主题子目录下。对于 CSS 样式表,不需要考虑文件名,主题系统会自动查找扩展名为 **.css** 的文件。同样, **.skin** 文件也可以有任意文件名,但最好使用多个 **.skin** 文件,每个 **.skin** 文件用于要设置样式的一个控件类型,每个 **.skin** 文件的名称都用该控件名指定。

**Skin** 文件包含服务器控件的定义,其格式与标准 ASP.NET 页面中使用的格式相同。其区别是 **skin** 文件中的控件不会添加到页面上,它们只用于提取属性。按钮样式的定义一般放在 **Button.skin** 文件中,其内容如下所示:

```
<asp:Button Runat="server" BackColor="#444499" BorderColor="#000000" ForeColor="#ccccff" />
```

这个文件实际上是从 **PCSDemoSite** 的 **DefaultTheme** 主题中提取的,负责设置本章前面 **Meeting Room Booker** 页面中的按钮外观。

以这种方式为控件类型创建 **skin** 文件时,不需要使用 **ID** 属性。

38.5.3 PCSDemoSite 中的主题

Web 站点 PCSDemoSite 包含 3 个主题，可以在/Configuration/Themes/Default.aspx 页面上选择这 3 个主题——只要登录为 RegisteredUser 或 SiteAdministrator 角色的一个成员即可。这个页面如图 38-15 所示。

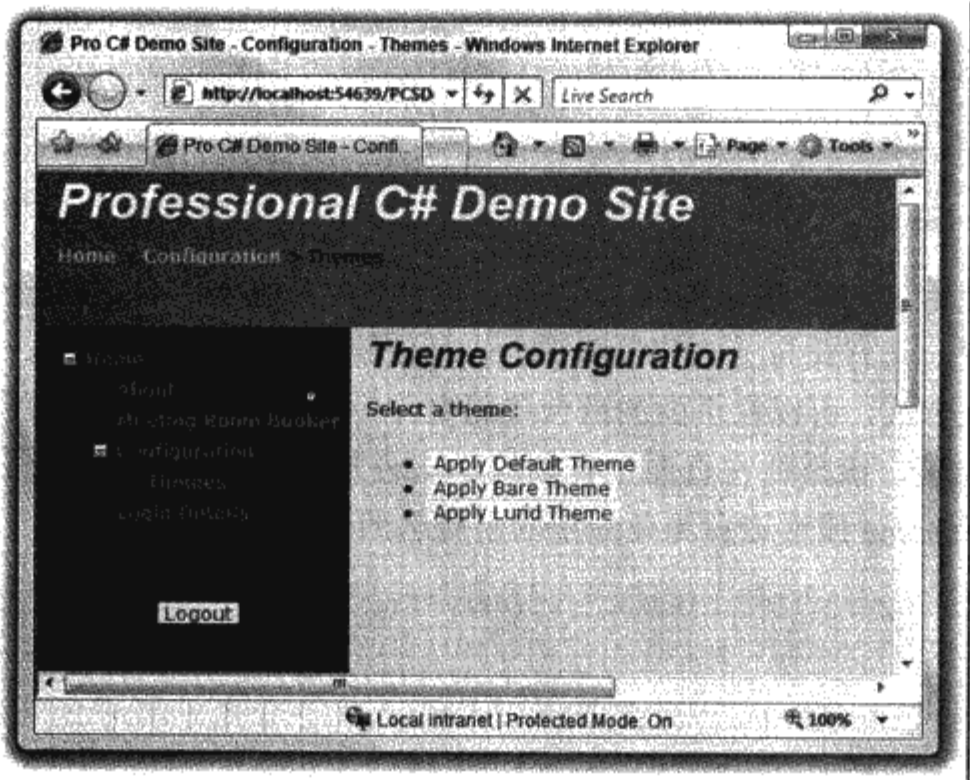


图 38-15

这里使用的主题是 DefaultTheme，也可以在这个页面上选择其他主题，图 38-16 显示了 BareTheme 主题。

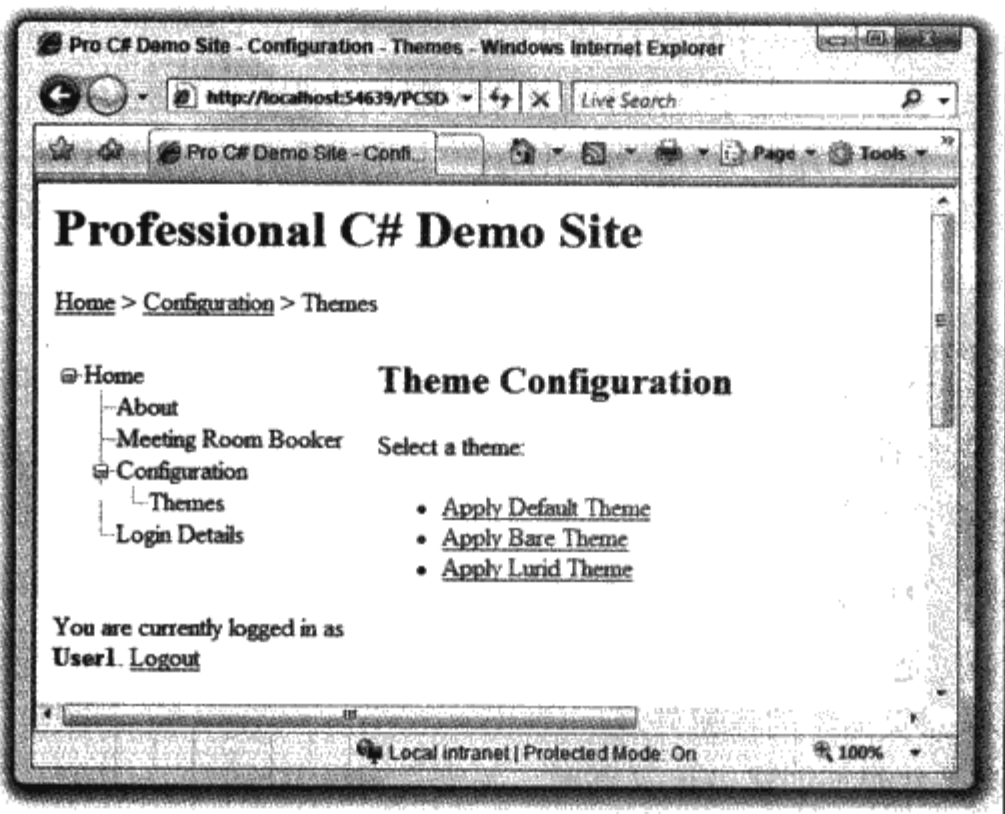


图 38-16

这个主题适合于 Web 页面的可打印版本。BareTheme 目录并不包含文件，这里使用的文件



是根样式表 StyleSheet.css。

图 38-17 显示了 LuridTheme 主题。

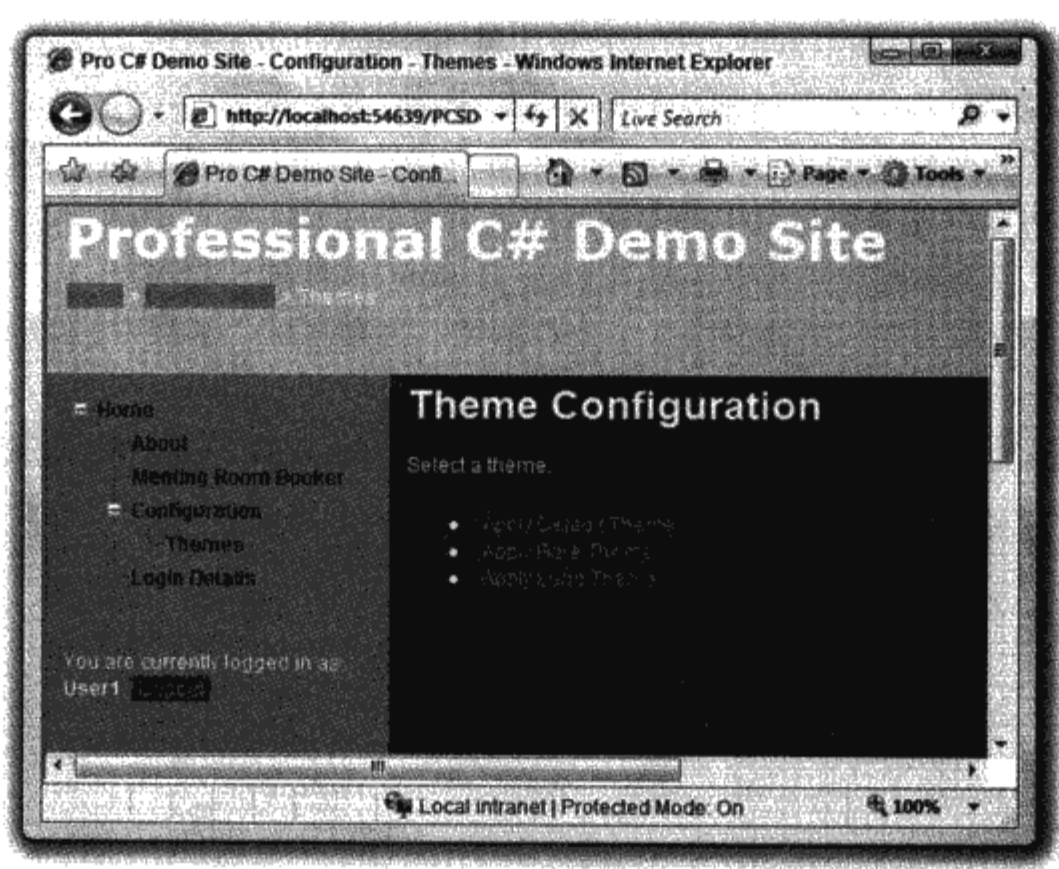


图 38-17

这个主题的颜色非常鲜亮，但很难阅读，它说明了站点的外观可以使用主题动态改变。更重要的是，像这样的主题可以提供 Web 站点的高对比度或大文本版本，以便于访问。

在 PCSDemoSite 上，当前选择的主题存储在会话状态中，所以在站点上浏览时，主题会保持不变。/Configuration/Themes/Default.aspx 的后台代码文件如下所示：

```
public partial class _Default : MyPageBase
{
    private void ApplyTheme(string themeName)
    {
        if (Session["SessionTheme"] != null)
        {
            Session.Remove("SessionTheme");
        }
        Session.Add("SessionTheme", themeName);
        Response.Redirect("~/Configuration/Themes", true);
    }

    void applyDefaultTheme_Click(object sender, EventArgs e)
    {
        ApplyTheme("DefaultTheme");
    }

    void applyBareTheme_Click(object sender, EventArgs e)
    {
        ApplyTheme("BareTheme");
    }

    void applyLuridTheme_Click(object sender, EventArgs e)
    {

```

```

        ApplyTheme("LuridTheme");
    }
}

```

这里的关键功能在 `ApplyTheme()` 中，它使用键 `SessionTheme` 把所选主题的名称放在会话状态中，并确定会话状态中是否已有一个主题，如果是，就删除它。

如前所述，主题必须在 `Page_PreInit()` 事件处理程序中应用，不能在所有页面使用的 `master` 页面中访问它，所以，如果要把选中的主题应用于所有的页面，有两个选项：

- 在所有要应用主题的页面中，重写 `Page_PreInit()` 事件处理程序
- 为所有要应用主题的页面提供一个公共基类，再重写这个基类中的 `Page_PreInit()` 事件处理程序

`PCSDemoSite` 使用第二个选项，在 `Code/MyPageBase.cs` 中提供了一个公共页面基类：

```

public class MyPageBase : Page
{
    protected override void OnPreInit(EventArgs e)
    {
        // theming
        if (Session["SessionTheme"] != null)
        {
            Page.Theme = Session["SessionTheme"] as string;
        }
        else
        {
            Page.Theme = "DefaultTheme";
        }

        // base call
        base.OnPreInit(e);
    }
}

```

这个事件处理程序检查 `SessionTheme` 中条目的会话状态，如果会话状态中有一项，就应用选中的主题，否则就使用 `DefaultTheme`。

注意这个类继承了通用的页面基类 `Page`，这是必需的，否则页面就不能执行为一个 ASP.NET Web 页面。

为了使程序正常工作，还要为所有的 Web 页面指定这个基类。这有几种方式，最简单的方式是在页面的 `<@ Page %>` 指令或后台代码文件中指定。前者适合于简单的页面，但页面不能使用定制的后台代码文件，因为页面在其定制的后台代码文件中不再使用 `<@ Page %>` 指令。另一种方式是修改页面在后台代码文件中继承的类。在默认情况下，新页面继承了 `Page`，但可以改变这个继承。在前面主题选择页面的后台代码文件中，注意有如下代码：

```

public partial class _Default : MyPageBase
{
    ...
}

```

这里把 `MyPageBase` 指定为 `_Default` 类的基类，所以使用在 `MyPageBase.cs` 中重写的方法。

## 38.6 Web Parts

ASP.NET 包含一组名为 Web Parts 的服务器控件，它们允许用户使 Web 页面个性化。在基于 SharePoint 的网站和 MSN 主页 <http://www.msn.com/> 上都可以看到这些控件。在使用 Web Parts 时，可以得到的功能如下：

- 给用户显示默认的页面布局。这个布局包含许多 Web Parts 组件，每个 Web Parts 都有标题和内容。
- 用户可以修改 Web Parts 在页面上的位置。
- 用户可以定制页面上 Web Parts 的外观，或者从页面中删除它们。
- 为用户提供一个 Web Parts 类别，允许用户将 Web Partst 拖放到页面上。
- 用户可以从页面中导出 Web Parts，再把它们导入另一个页面或站点上。
- Web Parts 之间可以建立连接。例如，显示在一个 Web Parts 上的内容可以是显示在另一个 Web Parts 上的内容的图形表示。
- 用户进行的任何修改都可以在浏览站点的过程中保存下来。

ASP.NET 为使用 Web Parts 功能提供了一个完整的架构，包括管理和编辑控件。

Web Parts 的使用是一个复杂的课题，本节不描述 Web Parts 的所有功能，列出 Web Parts 组件提供的所有属性和方法，只概述 Web Parts，让读者理解它的基本功能。

### 38.6.1 Web Parts 应用程序组件

工具箱的 Web Parts 部分包含 13 个控件，如图 38-18 所示(注意指针不是控件)。

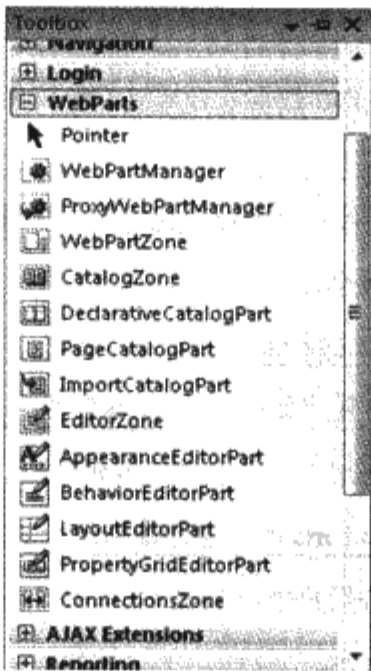


图 38-18

这些控件如表 38-4 所示。该表还介绍了 Web Parts 页面的一些重要概念。

表 38-4

控 件	说 明
WebPartManager	每个使用 Web Parts 的页面必须有一个(只能有一个)WebPartManager 控件的实例。可以把它放在 master 页面上。这个控件负责执行大部分 Web Parts 功能，不需要用户太多的干涉。只要根据自己需要的功能，将该控件放在 Web 页面上即可。对于更高级的功能，可以使用这个控件提供的许多属性和事件
ProxyWebPartManager	如果把 WebPartManager 控件放在 master 页面上，就很难在各个页面上配置它——实际上不能这么做。这与 Web Parts 之间静态连接的定义相关。ProxyWebPart Manager 控件可以在 Web 页面上声明性地定义静态连接，避免出现不能在同一个页面上放置两个 WebPartManager 控件的问题
WebPartZone	WebPartZone 控件用于定义可以包含 Web Parts 的页面区域。一般在页面上使用多个 WebPartZone 控件。例如，可以在页面的三列布局中使用三个 WebPartZone 控件。用户可以在 WebPartZone 区域之间移动 Web Parts，或者在一个 WebPartZone 中重新定位它们
CatalogZone	CatalogZone 控件允许用户把 Web Parts 添加到页面上。这个控件包含的控件派生于 CatalogZone，CatalogZone 提供了三个控件，本表的后面将介绍这三个控件。CatalogZone 及其包含的控件是否可见，取决于 WebPartManager 设置的当前显示模式
DeclarativeCatalogPart	DeclarativeCatalogPart 控件允许在线定义 Web Parts 控件。之后，这些控件就可以通过 CatalogZone 控件用于用户
PageCatalogPart	用户可以删除(关闭)显示在页面上的 Web Parts。为了检索它们，PageCatalogPart 控件提供了一组可以在页面上替换的、关闭的 Web Parts
ImportCatalogPart	ImportCatalogPart 控件允许通过 CatalogPart 接口把从页面中导出的 Web Part 再导入另一个页面
EditorZone	EditorZone 控件包含的控件允许用户根据 Web Parts 包含的控件，编辑 Web Parts 显示和行为的各个方面。它可以包含派生于 EditorPart 的控件，包括本表中后面的四个控件。与 CatalogZone 一样，这个控件的显示取决于当前显示模式
AppearanceEditorPart	这个控件允许用户修改 Web Parts 控件的外观和大小，并能隐藏它们
BehaviorEditorPart	这个控件允许用户使用它的许多属性配置 Web Parts 的行为，例如 Web Parts 是否可以关闭，Web Parts 的标题链接到什么 URL 上
LayoutEditorPart	这个控件允许用户修改 Web Parts 的布局属性，例如它包含在什么区域，在最小化状态下它是否显示
PropertyGridEditorPart	这是最一般的 Web Parts 编辑器控件，允许定义可以为定制 Web Parts 控件编辑的属性。之后，用户就可以编辑这些属性了
ConnectionsZone	这个控件允许用户在支持连接功能的 Web Parts 之间创建连接。与 CatalogZone 和 EditorZone 不同，这个控件中不放置任何其他控件。这个控件生成的用户界面取决于页面上可以进行连接的控件。这个控件的可见性取决于显示模式

注意，表 38-4 中的控件没有包含任何特定的 Web Part 控件。这是因为这些控件是我们自己创建的。任何放在 WebPartZone 区域中的控件会自动变成 Web Part，包括(最重要的)用户控件。使用用户控件，可以把其他控件组合在一起，提供 Web Part 控件的用户界面和功能。

### 38.6.2 Web Parts 示例

为了演示 Web Parts 的功能，可以查看本章下载代码中的示例 PCSWebParts。这个示例将使用 PCSAuthenticationDemo 例子的安全数据库。它有两个用户，其用户名是 User 和 Administrator，密码是 Pa\$\$w0rd。还可以登录为用户，处理页面上的 Web Parts，注销，之后再登录为另一个用户，以完全不同的方式处理 Web Parts。这两个用户的个性化会在站点访问的过程中保存下来。

登录到站点上后，显示的结果(用 User 登录)如图 38-19 所示。

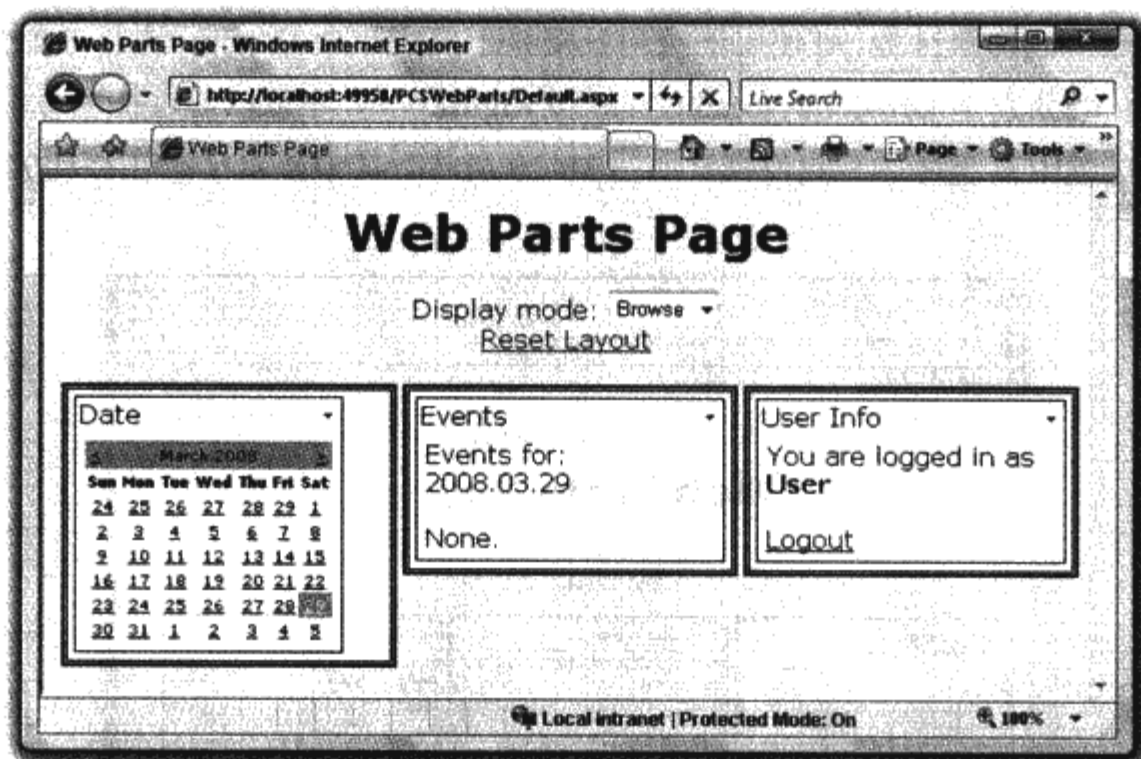


图 38-19

这个页面包含如下控件：

- 1 个 WebPartManager 控件(没有可见的组件)
- 3 个 WebPartZone 控件
- 3 个 Web Parts(Date、Events 和 User Info)，分别放在 3 个 WebPartZone 控件中，其中两个 Web Parts 通过一个静态连接关联起来，如果修改 Date 中的日期，Events 中显示的日期就会更新。
- 改变显示模式的下拉列表。这个列表没有包含所有可能的显示模式，只包含了可用的显示模式。可用的模式是从 WebPartManager 控件中获得的，如后面所示。列出的模式有：



- **Browse:** 这个模式是默认的，允许查看和使用 Web Parts。在这个模式下，每个 Web Part 都可以使用下拉菜单最小化和关闭，下拉菜单可以从每个 Web Part 的右上角访问。
- **Design:** 在这个模式下，可以重新定位 Web Parts。
- **Edit:** 在这个模式下，可以编辑 Web Part 属性。每个 Web Part 的下拉菜单中，有一个额外的菜单项 Edit。
- **Catalog:** 在这个模式下，可以给页面添加新 Web Parts。
- 一个链接。将 Web Part 布局重置为默认情况(仅用于当前用户)
- 一个 EditorZone 控件(只在 Edit 模式下可见)
- 一个 CatalogZone 控件(只在 Catalog 模式下可见)
- 类别中可以添加到页面上的一个额外 Web Part

每个 Web Part 都在用户控件中定义。

为了演示布局的修改过程，可使用下拉列表将显示模式改为 Design。注意每个 WebPartZone 都带有一个 ID 值(分别是 LeftZone、CenterZone 和 RightZone)。还可以通过拖动标题来移动 Web Parts，在拖动过程中甚至可以看到反馈。如图 38-20 所示，其中显示了正在移动的名为 Date 的 Web Parts。

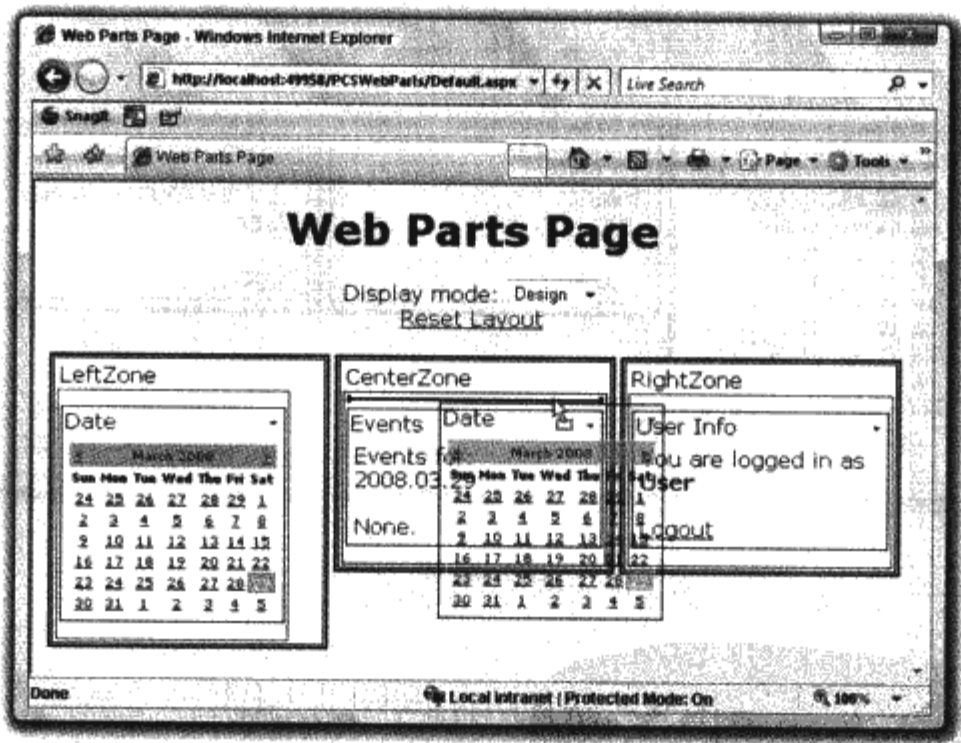


图 38-20

接着，在类别中添加一个新的 Web Part。将显示模式改为 Catalog，注意 CatalogZone 在页面的底部可见。单击 Declarative Catalog 链接，就可以在页面上添加一个 Links 控件，如图 38-21 所示。

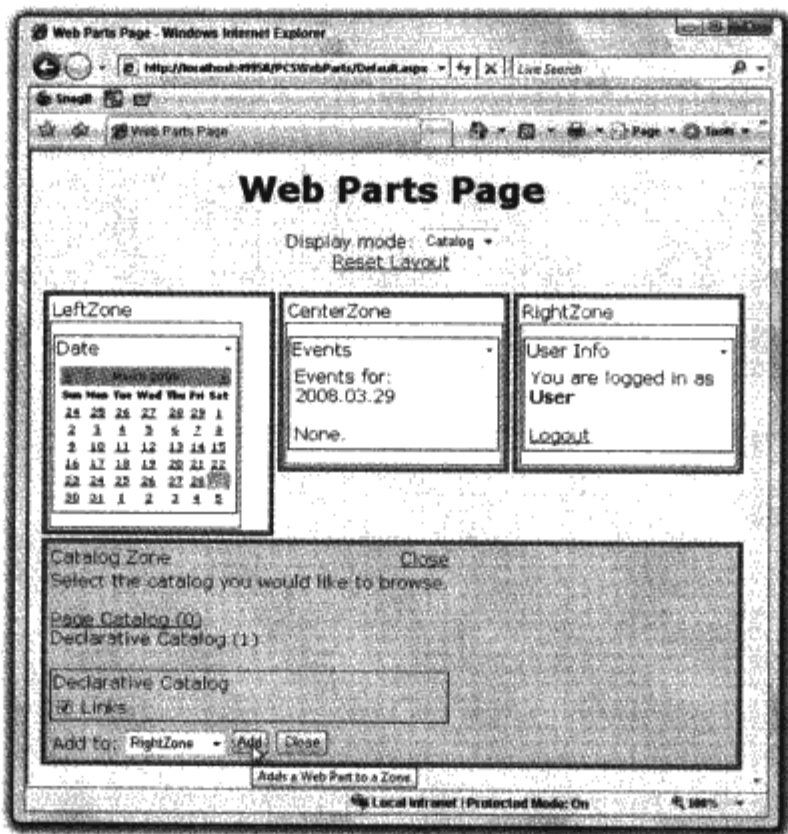


图 38-21

注意这里还有一个 Page Catalog 链接。如果使用下拉菜单选择 Web Part，就会看到 Page Catalog 链接，它没有删除，只是隐藏了。

之后，把显示模式改为 Edit，再从 Web Part 的下拉列表中选择 Edit，如图 38-22 所示。

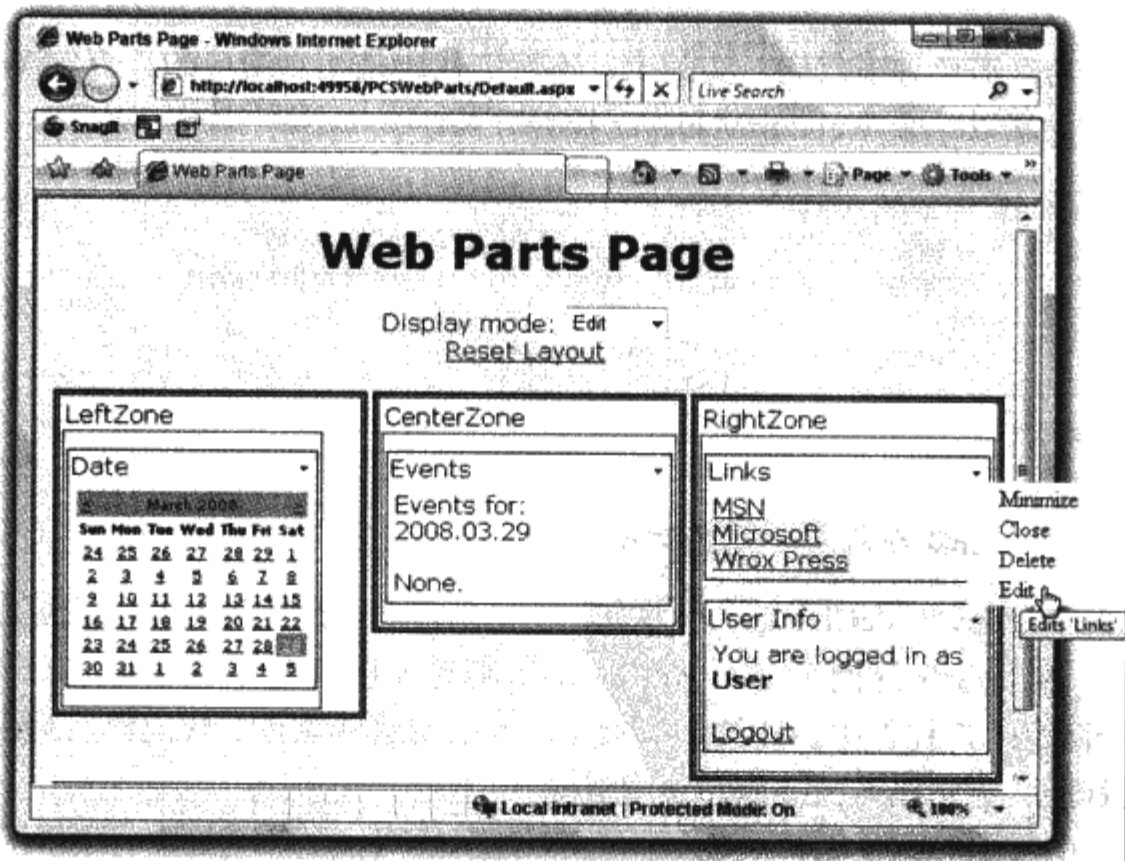


图 38-22

选择这个菜单项时，会打开 EditorZone 控件。在本例中，这个控件包含一个 AppearanceEditorPart 控件，如图 38-23 所示。

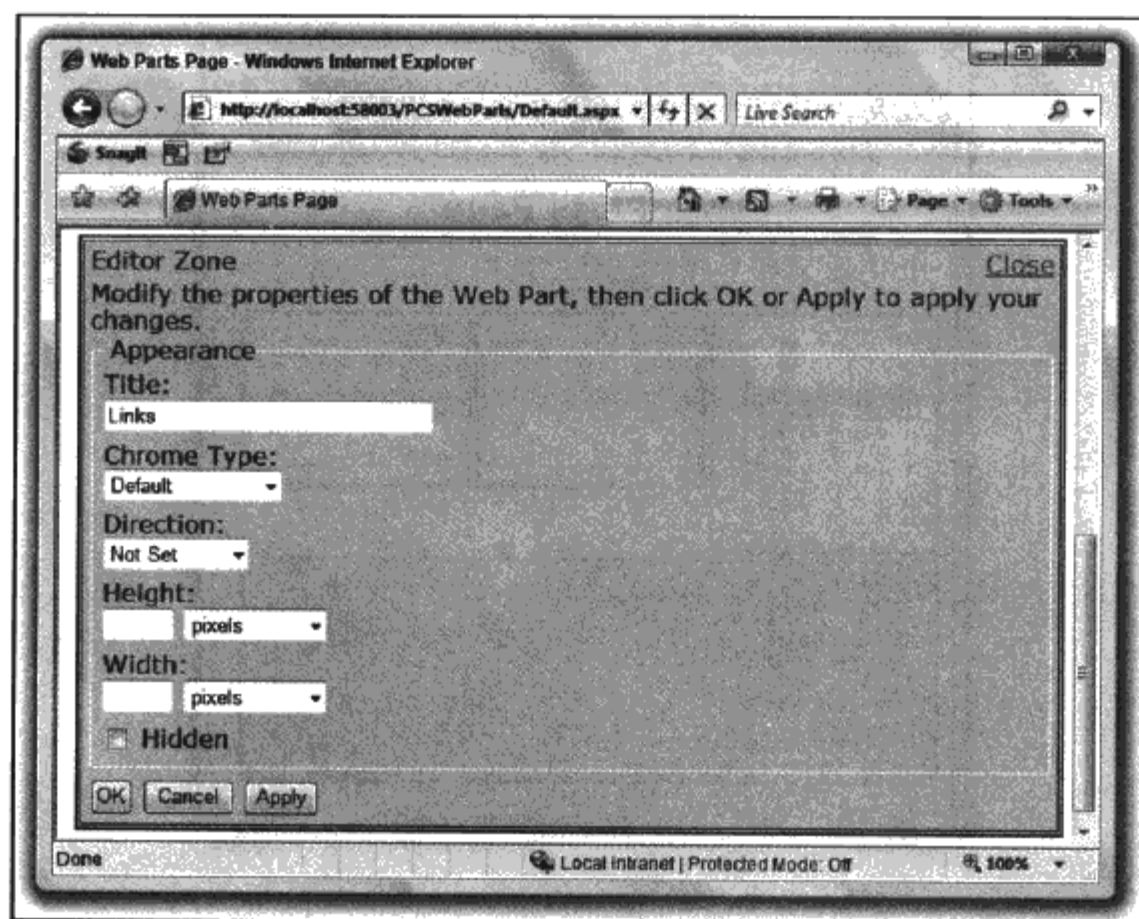


图 38-23

使用这个界面可以编辑和应用 Web Parts 的属性值。

完成了修改后，确认存储了它们，以便用户注销，再登录为另一个用户，之后切换回第一个用户。

现在，这个功能需要许多代码。实际上，本例的代码是相当简单的。查看 Web Parts 页面的代码。<form>元素以一个 WebPartManager 控件开头：

```
<form id="form1" runat="server">
  <asp:WebPartManager ID="WebPartManager1" runat="server"
    OnDisplayModeChanged="WebPartManager1_DisplayModeChanged">
    <StaticConnections>
      <asp:WebPartConnection ID="dateConnection"
        ConsumerConnectionPointID="DateConsumer"
        ConsumerID="EventListControl1" ProviderConnectionPointID="DateProvider"
        ProviderID="DateSelectorControl1" />
    </StaticConnections>
  </asp:WebPartManager>
```

这个控件的 DisplayModeChanged 事件有一个处理程序，用于显示或隐藏页面底部的<div>编辑器。在 Date 和 Events 这两个 Web Parts 之间的静态连接也有一个规范，为此，要为用于这些 Web Parts 的两个用户控件中的连接定义端点，并引用这些端点。代码稍后列出。

接着定义标题、显示模式改变器和重置链接：

```
<div class="mainDiv">
  <h1>Web Parts Page</h1>
  Display mode:
  <asp:DropDownList ID="displayMode" runat="server" AutoPostBack="True"
    OnSelectedIndexChanged="displayMode_SelectedIndexChanged" />
  <br />
  <asp:LinkButton runat="server" ID="resetButton" Text="Reset Layout"
```

```

OnClick="resetButton_Click" />
<br />
<br />

```

使用 `WebPartManager1.SupportedDisplayModes` 属性, 在 `Page_Load()` 事件处理程序中填充显示模式下拉列表。重置按钮使用 `WebPartManager1.Personalization.ResetPersonalizationState()` 方法给当前用户重置个性化状态。

接着是三个 `WebPartZone` 控件, 每个控件都包含一个加载为 `Web Part` 的用户控件:

```

<div class="innerDiv">
  <div class="zoneDiv">
    <asp:WebPartZone ID="LeftZone" runat="server">
      <ZoneTemplate>
        <uc1:DateSelectorControl ID="DateSelectorControl1" runat="server"
          title="Date" />
      </ZoneTemplate>
    </asp:WebPartZone>
  </div>
  <div class="zoneDiv">
    <asp:WebPartZone ID="CenterZone" runat="server">
      <ZoneTemplate>
        <uc2:EventListControl ID="EventListControl1" runat="server"
          title="Events" />
      </ZoneTemplate>
    </asp:WebPartZone>
  </div>
  <div class="zoneDiv">
    <asp:WebPartZone ID="RightZone" runat="server">
      <ZoneTemplate>
        <uc4:UserInfo ID="UserInfo1" runat="server" title="User Info" />
      </ZoneTemplate>
    </asp:WebPartZone>
  </div>

```

最后是 `EditorZone` 和 `CatalogZone` 控件, 它们分别包含一个 `AppearanceEditor` 控件、`PageCatalogPart` 和 `DeclarativeCatalogPart` 控件:

```

<asp:Placeholder runat="server" ID="editorPH" Visible="false">
  <div class="footerDiv">
    <asp:EditorZone ID="EditorZone1" runat="server">
      <ZoneTemplate>
        <asp:AppearanceEditorPart ID="AppearanceEditorPart1"
          runat="server" />
      </ZoneTemplate>
    </asp:EditorZone>
    <asp:CatalogZone ID="CatalogZone1" runat="server">
      <ZoneTemplate>
        <asp:PageCatalogPart ID="PageCatalogPart1" runat="server" />
        <asp:DeclarativeCatalogPart ID="DeclarativeCatalogPart1"
          runat="server">
          <WebPartsTemplate>
            <uc3:LinksControl ID="LinksControl1" runat="server"
              title="Links" />
          </WebPartsTemplate>
        </asp:DeclarativeCatalogPart>
      </ZoneTemplate>
    </asp:CatalogZone>
  </div>

```



```

        </asp:Placeholder>
    </div>
</div>
</form>

```

DeclarativeCatalogPart 控件包含第四个用户控件，这是一个 Links 控件，用户可以把它添加到页面上。

Web Parts 的代码是相当简单的。例如，Links 控件只包含下述代码：

```

<%@ Control Language="C#" AutoEventWireup="true" CodeFile="LinksControl.ascx.cs"
    Inherits="LinksControl" %>
<a href="http://www.msn.com/">MSN</a>
<br />
<a href="http://www.microsoft.com/">Microsoft</a>
<br />
<a href="http://www.wrox.com/">Wrox Press</a>

```

不需要额外的标记，就可以使这个用户控件作为一个 Web Part 工作。这里唯一要注意的是，用户控件的<uc3:LinksControl>元素有一个 title 属性，但用户控件没有 Title 属性。这个属性由 DeclarativeCatalogPart 控件给 Web Part 推断要显示的标题（可以在运行期间用 AppearanceEditorPart 编辑）。

将一个接口引用从 DateSelectorControl 传送给 EventListControl（这些 Web Parts 使用的两个用户控件类），就建立了 Date 和 Events 控件之间的连接。

```

public interface IDateProvider
{
    SelectedDatesCollection SelectedDates
    {
        get;
    }
}

```

DateSelectorControl 支持这个接口，所以可以使用 this 传送 IDateProvider 的一个实例。引用通过 DateSelectorControl 中的端点方法来传送，该引用是用 ConnectionProvider 属性修饰的：

```

[ConnectionProvider("Date Provider", "DateProvider")]
public IDateProvider ProvideDate()
{
    return this;
}

```

这就把 Web Part 标记为一个提供程序控件了。之后，就可以通过端点 ID 引用提供程序了，本例是 DateProvider。

要使用提供程序，可以使用 ConnectionConsumer 属性在 EventListControl 中指定一个使用方法：

```

[ConnectionConsumer("Date Consumer", "DateConsumer")]
public void GetDate(IDateProvider provider)
{
    this.provider = provider;
    IsConnected = true;
    SetDateLabel();
}

```



这个方法存储了一个传送过来的 `IDateProvider` 接口引用，设置一个标记，修改控件中的标签文本。

这个例子没有更多需要解释的地方。代码中有几个小装饰段，还有 `Page_Load()` 中事件处理程序的信息，但这里不需要讨论它们。查看本章的下载代码，可以进一步研究它们。

但是 **Web Parts** 可以完成更多的工作。**Web Parts** 架构非常强大，功能非常丰富，需要一整本书的篇幅来探讨。但本节概述了 **Web Parts**，揭密了它们的一些功能。

## 38.7 小结

本章介绍了创建 **ASP.NET** 页面和 **Web** 站点的几个高级技术，并在示例 **Web** 站点 **PCSDemoSite** 中演示了这些技术。

首先探讨了如何使用 **C#** 创建可重用的 **ASP.NET** 服务器控件。其中讨论了如何从现有的 **ASP.NET** 页面中创建简单的用户控件，也讨论了如何从头创建定制控件。还研究了如何把上一章的会议室登记工具示例修改为一个用户控件。

接着介绍了 **master** 页面，如何为 **Web** 站点的页面提供模板，这是重用代码和简化开发的另一种方式。在 **PCSDemoSite** 中，有一个 **master** 页面包含 **Web** 导航服务器控件，允许用户在站点上浏览，并建立了主题的框架。本章后面介绍的主题非常适合于把功能与设计分开，是一个强大的可访问性技术。

我们还简要介绍了安全性，探讨了如何在 **Web** 站点上实现基于窗体的身份验证。

最后研究了 **Web Parts**，介绍了如何使用 **Web Parts** 服务器控件把基本应用程序和这个技术提供的一些功能集成在一起。

本章仅涉及 **ASP.NET 2.0** 功能的皮毛。例如，使用定制控件可以完成许多任务，模板和后期绑定的讨论是非常有趣的，并可以据此创建控件。但掌握了本章的内容后，就可以开始建立自己的定制控件了，还可以试验本章讨论的其他技术。

下一章介绍使用 **Ajax** 技术使 **ASP.NET** 应用程序更动态化的方式。

# 第39章

## ASP.NET AJAX

Web 应用程序编程是一个不断变化和改进的主题。前面两章介绍了如何使用 ASP.NET 创建功能全面的 Web 应用程序，读者可能以为，前面已经探讨了创建自己的 Web 应用程序所需的所有工具。但是，如果花点时间查看当前的网站，就会注意到，最近的网站在使用方面比老网站好得多。许多目前最好的网站都提供了丰富的用户界面，其响应能力与 Windows 应用程序差不多。它们是使用客户端处理技术实现的，主要是 JavaScript 代码和一种新技术 Ajax。

出现这个变化，是因为客户用于浏览网站的浏览器和客户用于运行浏览器的计算机更强大。Web 浏览器的当前版本，例如 Internet Explorer 7 和 Firefox，也支持各种标准。这些标准，包括 JavaScript，使 Web 应用程序提供的功能比使用普通的 HTML 提供的功能强大得多。前面的章节介绍了一些这方面的功能，例如使用层叠样式表(CSS)设置 Web 应用程序的样式。

本章介绍的 Ajax 并不是一个新技术，它只是一个标准的合并，以识别当前 Web 浏览器的丰富的潜在功能。

在支持 Ajax 的 Web 应用程序中，最重要的特性是 Web 浏览器能在操作的外部与 Web 服务器通信。这称为异步回送或部分页面的回送。实际上，这意味着用户可以与服务器端的功能和数据交互，而无需更新整个页面。例如，单击一个链接，移动到表的第二页数据上时，Ajax 可以只刷新表的内容，而不刷新整个 Web 页面。也就是说，需要的 Internet 通信量较少，从而使 Web 应用程序的响应比较快。本章的后面将介绍这个例子，还会举许多例子来说明 Ajax 在 Web 应用程序中的巨大作用。

本章将在代码中使用 Ajax 的 Microsoft 实现方式，它称为 ASP.NET AJAX。这个实现方式采用了 Ajax 模型，将它应用于 ASP.NET 架构。ASP.NET AJAX 提供了许多服务器控件和客户端技术，它们专用于 ASP.NET 开发人员，可以毫不费力地在 Web 应用程序中添加 Ajax 功能。

本章的内容如下：

- 首先学习 Ajax 和实现 Ajax 的技术。
- 学习 ASP.NET AJAX 及其组成部分，以及 ASP.NET AJAX 提供的功能。
- 介绍如何通过服务器端和客户端代码在 Web 应用程序中使用 ASP.NET AJAX。这是本章最大的一部分。

### 39.1 Ajax 的概念

Ajax 允许通过异步回送和动态的客户端 Web 页面处理，改进 Web 应用程序的用户界面。

术语“Ajax”由 Jesse James Garrett 提出，是 Asynchronous JavaScript and XML 的缩写。

提示：

Ajax 不是一个缩写词，因此不能写作 AJAX。但是，在产品名称 ASP.NET AJAX 中它是大写，这是 Ajax 的 Microsoft 实现方式，如下一节所述。

根据定义，Ajax 显然涉及到 JavaScript 和 XML。但是，Ajax 编程需要使用其他技术，如表 39-1 所述。

表 39-1

技 术	说 明
HTML/XHTML	HTML(超文本标记语言, Hypertext Markup Language)是显示和布局语言，由 Web 浏览器用于在图形化用户界面上显示信息。在前面两章中，学习了 HTML 如何实现这个功能，ASP.NET 如何生成 HTML 代码。可扩展的 HTML(XHTML)是使用 XML 结构的一个较严谨的 HTML 版本
CSS	CSS(层叠样式表)是 HTML 元素根据一个样式表中定义的规则设置样式的方式。它允许将样式同时应用于多个 HTML 元素，能在不修改 HTML 的情况下改变 Web 页面的外观。CSS 包含布局和样式信息，所以也可以使用 CSS 在页面上定位 HTML 元素。前面的章节还在例子中介绍了具体的操作
DOM	DOM(文档对象模型)是在层次结构中表示和处理(X)HTML 代码的一种方式。它允许访问页面上的“表 x 中第三行的第二列”，且无需使用比较基本的文本处理方式定位这个元素
JavaScript	JavaScript 是一个客户端脚本编辑技术，允许在 Web 浏览器上执行代码。JavaScript 的语法类似于其他基于 C 的语言，包括 C#，提供了变量、函数、分支代码、循环语句、事件处理程序和其他编程元素。但是与 C#不同，JavaScript 不是强类型化的，JavaScript 代码的调试比较困难。对于 Ajax 编程，JavaScript 是一种关键技术，因为它允许利用 DOM 处理功能，动态修改 Web 页面
XML	XML 是标记数据的一种独立于平台的方式，对 Ajax 非常关键，它既是处理数据的方式，也是客户机和服务器之间的通信语言
XmlHttpRequest	自 Internet Explorer 5 以来，浏览器就把 XmlHttpRequest API 作为一种在客户机和服务器之间进行异步通信的方式。Microsoft 最初把它引入为一种技术，以访问通过 Internet 存储在 Exchange 服务器中的电子邮件，它使用的产品叫做 Outlook Web Access。后来它变成在 Web 应用程序中进行异步通信的标准方式，是支持 Ajax 的 Web 应用程序的一个核心技术。这个 API 的 Microsoft 实现方式称为 XMLHttpRequest，它利用所谓的 XMLHttpRequest 协议来通信

Ajax 还需要用服务器端代码处理部分页面的回送和完整页面的回送，这包括服务器控件的事件处理程序和 Web 服务(Web 服务详见第 37 章)。图 39-1 显示了这些技术如何在 Ajax Web 浏览器模型中联合使用，并与传统的 Web 浏览器模型进行比较。

在 AJAX 推出之前，表 39-1 中的前四个技术(HTML、CSS、DOM 和 JavaScript)用于创建所谓的动态 HTML(DHTML) Web 应用程序。这些应用程序比较著名有两个原因：它们提供的用户界面要好得多；它们一般只能用于一种类型的 Web 浏览器。

自 DHTML 推出以来，标准已有了改进，Web 浏览器的相关标准级别也提高了。但是，它们仍有区别，Ajax 解决方案必须考虑这些区别。也就是说，大多数开发人员实现 Ajax 解决方案还相当慢。只有开发出更抽象的 Ajax 架构(例如 ASP.NET AJAX)，创建支持 Ajax 的网站才是企业级开发的一个可行选项。

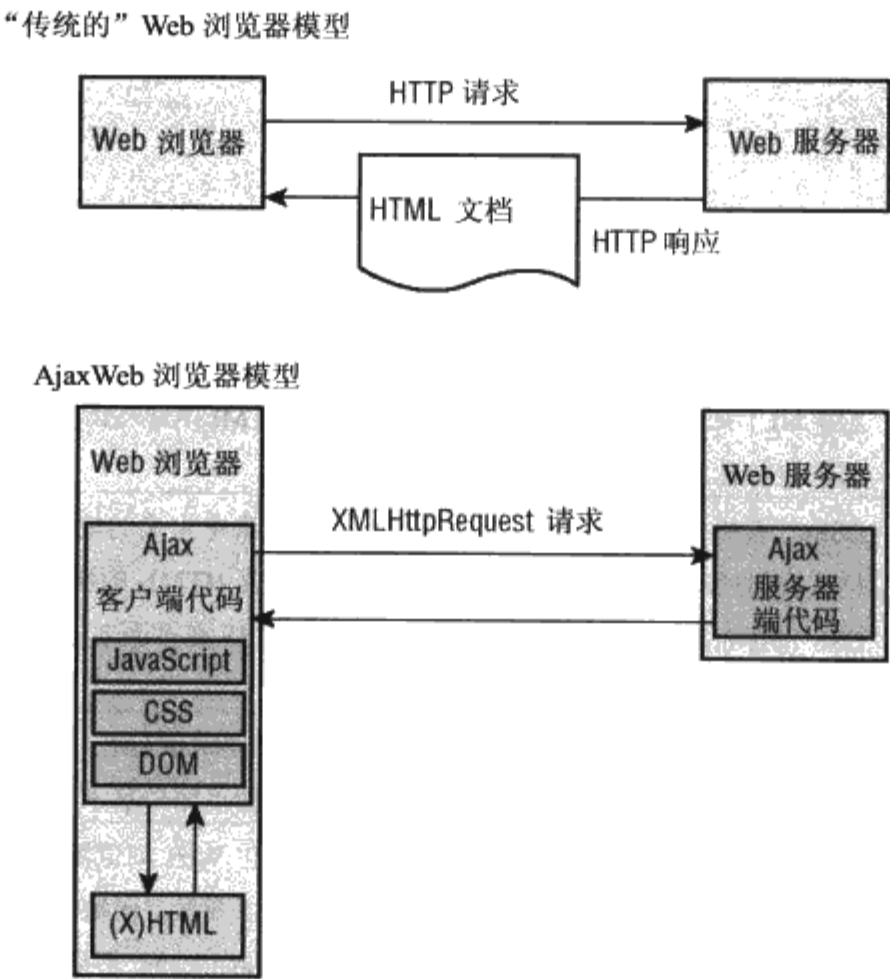


图 39-1

### 39.2 ASP.NET AJAX

ASP.NET AJAX 是 Ajax 架构的 Microsoft 实现方式，专用于 ASP.NET 开发人员。在 ASP.NET 最新版本中，ASP.NET AJAX 是 ASP.NET 核心功能的一部分。网站 <http://ajax.asp.net> 上可以用于以前的 ASP.NET 版本，包含相关的文档说明、论坛和示例代码，可以用于我们使用的 ASP.NET 版本。

ASP.NET AJAX 提供了如下功能：

- 服务器端架构允许 ASP.NET Web 页面响应部分页面的回送操作。
- ASP.NET 服务器控件便于实现 Ajax 功能。
- HTTP 处理程序允许 ASP.NET Web 服务在部分页面的回送操作中，使用 JavaScript Object Notation(JSON)串行化功能与客户端代码通信。
- Web 服务支持客户端代码访问 ASP.NET 应用程序服务，包括身份验证和个性化服务。
- 网站模板可用于创建支持 ASP.NET AJAX 的 Web 应用程序。

- 客户端的 JavaScript 库对 JavaScript 语法进行了许多改进，还提供了许多代码，来简化 Ajax 功能的实现。

这些服务器控件和服务端端的架构统称为 ASP.NET Extensions。ASP.NET AJAX 的客户端部分称为 AJAX 库。

还可以从网站 <http://ajax.asp.net> 上下载另外两个软件包：

- ASP.NET AJAX Control Toolkit: 这个下载软件包包含了由开发团体创建的其他服务器控件。这些控件是共享的，可以查看和修改它们。
- Microsoft AJAX Library 3.5: 这个下载软件包包含 JavaScript 客户端架构，它们由 ASP.NET AJAX 用于执行 Ajax 功能。如果开发的是 ASP.NET AJAX 应用程序，就不需要这个软件包。这个下载软件包适用于其他语言，例如 PHP，使用与 ASP.NET AJAX 相同的代码基执行 Ajax 功能。它超出了本章的范围。

**提示：**

还有一个下载软件包称为 Futures，过去用于给 ASP.NET AJAX 应用程序添加额外的、预先发布的功能或原始功能。但在编写本书时，不清楚这个下载软件包是否在 VS 2008 中得到支持，所以本章不介绍它。

这两个下载软件包提供了功能丰富的架构，可以用于在自己的 ASP.NET Web 应用程序中添加 Ajax 功能。下面几节将介绍 ASP.NET AJAX 的各个组成部分。

### 39.2.1 核心功能

ASP.NET AJAX 的核心功能分为两个部分：AJAX 扩展和 AJAX 库。

#### 1. AJAX 扩展

在安装 ASP.NET AJAX 时，会在 GAC 中安装两个程序集：

- System.Web.Extensions.dll: 这个程序集包含 ASP.NET AJAX 功能，包括 AJAX 扩展和 AJAX 库 JavaScript 文件，它们可以通过 ScriptManager 组件(稍后介绍)来获得。
- System.Web.Extensions.Design.dll: 这个程序集包含用于 AJAX 扩展服务器控件的 ASP.NET Designer 组件，这些服务器控件由 ASP.NET Designer 在 Visual Studio 或 Visual Web 开发程序中使用。

ASP.NET AJAX 中的许多 AJAX 扩展组件都涉及支持部分页面的回送和用于 Web 服务的 JSON 串行化。这包括各种 HTTP 处理程序组件和对已有 ASP.NET 架构的扩展。这些功能都可以通过网站的 Web.config 文件来配置。还有用于其他配置的和属性。但大多数配置都是透明的，用户很少需要改变支持 ASP.NET AJAX 的网站模板提供的默认设置。

与 AJAX 扩展的主要交互操作是使用服务器控件将 Ajax 功能添加到 Web 应用程序中。有几个服务器控件可以用各种方式增强 Web 应用程序。表 39-2 列出了一些服务器端组件。本章的后面将介绍它们。



表 39-2

控 件	说 明
ScriptManager	<p>这个控件是 ASP.NET AJAX 功能的核心，使用部分页面回送功能的每个页面都需要它。它的主要作用是管理对 AJAX 库 JavaScript 文件的客户端引用，AJAX 库 JavaScript 文件位于 ASP.NET AJAX 程序集中。AJAX 库主要由 AJAX 扩展服务器控件使用，这些控件会生成自己的客户端代码。</p> <p>这个控件还负责配置要在客户端代码中访问的 Web 服务。给 ScriptManager 控件提供 Web 服务的信息，就可以生成客户端类和服务器端类，来透明地管理与 Web 服务的异步通信。</p> <p>还可以使用 ScriptManager 控件维护对自己的 JavaScript 文件的引用，和 Futures CTP 中包含的其他 JavaScript 文件的引用</p>
UpdatePanel	<p>UpdatePanel 控件非常有用，也许是最常用的 ASP.NET 控件。这个控件与标准的 ASP.NET 占位符类似，可以包含其他控件。更重要的是，在部分页面的回送过程中，它还把页面的一个部分标记为可以独立于其他页面部分来更新的区域。</p> <p>UpdatePanel 控件包含的、产生回送操作的任意控件(如按钮控件)，都不会产生整个页面的回送操作，它们只执行部分页面的回送，只更新 UpdatePanel 的内容。</p> <p>在许多情况下，只需要这个控件实现 AJAX 功能。例如，可以把一个 GridView 控件放在 UpdatePanel 控件中，该控件的分页、排序和其他回送功能都在部分页面的回送过程中发挥作用</p>
UpdateProgress	<p>在部分页面的回送过程中，这个控件可以为用户提供反馈。在更新 UpdatePanel 时，可以为要显示的 UpdateProgress 控件提供一个模板。例如，可以使用浮点数的&lt;div&gt;控件显示消息“Updating...”，告诉用户应用程序正在忙。注意部分页面的回送不会干扰 Web 页面的其他区域，其他区域仍可以响应</p>
Timer	<p>ASP.NET AJAX 的 Timer 控件是使 UpdatePanel 定期更新的一种有效方式。可以把这个控件配置为定期触发回送操作。如果这个控件包含在 UpdatePanel 控件中，则每次触发 Timer 控件时，都会更新该 UpdatePanel 控件。Timer 控件也有关联的事件，所以可以执行定期的服务器端处理</p>
AsyncPostBackTrigger	<p>这个控件可以在未包含在 UpdatePanel 中的控件里触发 UpdatePanel 的更新操作。例如，可以在 Web 页面的其他地方放置一个下拉列表，来更新包含 GridView 控件的 UpdatePanel</p>

AJAX 扩展还包含 ExtenderControl 抽象基类，来扩展已有的 ASP.NET 服务器控件。它由 ASP.NET 2.0 AJAX Futures CTP 中的各种类使用，如后面所述。

2. AJAX 库

在支持 ASP.NET AJAX 的 Web 应用程序中，AJAX 库包含的 JavaScript 文件由客户端代码使用。在这些 JavaScript 文件中包含许多功能，其中一些是改进 JavaScript 语言的通用代码，一些则专用于 Ajax 功能。AJAX 库包含的功能彼此互为基础，如表 39-3 所示。

表 39-3

功 能 层	说 明
浏览器兼容性	AJAX 库的最底层代码根据客户机的 Web 浏览器来映射各种 JavaScript 功能，这是必需的，因为 JavaScript 在不同浏览器中的实现方式是有区别的。提供这个功能层，其他层上的 JavaScript 代码就不必考虑浏览器的兼容性了，我们也可以编写独立于浏览器、在所有客户机环境中工作的代码
核心服务	这一层包含对 JavaScript 语言的增强，尤其是 OOP 功能。使用这一层的代码，可以使用 JavaScript 脚本文件定义命名空间、类、派生类和接口。C#开发人员对此特别感兴趣，因为它使 JavaScript 代码的编写非常类似于用 C#编写.NET 代码，且鼓励代码的重用
基类库	客户基类库(BCL)包含许多 JavaScript 类，它们为 AJAX 库层次结构中下层的类提供了底层功能。这些类中的大多数都不能直接使用
联网	联网层上的类允许客户端代码异步调用服务器端代码。这一层包含的基本架构可以调用 URL，响应回调函数的结果。在大多数情况下，这些功能都不能直接使用，而应使用封装了该功能的类。这一层还包含用于 JSON 串行化和并行化的类，大多数联网类都在客户端的 System.Net 命名空间中
用户界面	这一层包含的类抽象了用户界面元素，如 HTML 元素和 DOM 事件。可以使用这一层的方法和属性编写独立于语言的 JavaScript 代码，来处理客户机上的 Web 页面。用户界面类包含在 System.UI 命名空间中
控件	AJAX 库的最后一层包含最高级代码，它们提供了 Ajax 操作和服务端控件功能。这包括动态生成代码，用于在客户端的 JavaScript 代码中调用 Web 服务

AJAX 库可以用于扩展和定制支持 ASP.NET AJAX 的 Web 应用程序的操作，但注意，不一定要这么做。要想在应用程序中不使用任何附加的 JavaScript，还有很长的路要走，只有需要更高级的功能，才需要这么做。如果要编写附带的客户端代码，使用 AJAX 库提供的功能会比较容易完成任务。

39.2.2 ASP.NET AJAX Control Toolkit

AJAX Control Toolkit 是附加服务器控件的一个集合，包括由 ASP.NET AJAX 团体编写的扩展控件。扩展控件可以在已有的 ASP.NET 服务器控件中添加功能，一般是给它附带一个客户端操作。例如，AJAX Control Toolkit 中的一个扩展器能在文本框中放置“watermark”文本，以扩展文本框控件，当用户还没有在文本框中添加任何内容时，就会显示该文本。这个扩展控件在服务器控件 TextBoxWatermark 中实现。

使用 AJAX Control Toolkit 可以给站点添加许多功能，它们超出了核心下载包的范围。这些控件可以使浏览操作更有趣，也许能为增强 Web 应用程序提供许多新思路。但是，AJAX Control Toolkit 独立于核心下载包，所以这些控件并没有获得与核心下载包中的控件相同的支持。

## 39.3 使用 ASP.NET AJAX

前面介绍了 ASP.NET AJAX 的组件部分，下面就开始探讨如何使用它们增强网站。本节将讨论支持 ASP.NET AJAX 的 Web 应用程序如何工作，如何使用该软件包中的各种功能。首先仔细研究一个简单的应用程序，然后在后续的章节中添加其他功能。

### 39.3.1 ASP.NET AJAX 网站示例

ASP.NET AJAX 模板包含了 ASP.NET AJAX 的所有核心功能。也可以使用 AJAX Control Toolkit Web Site 模板，以包含 AJAX Control Toolkit 中的控件。本示例要在 C:\ProCSharp\Chapter39 目录中创建一个使用默认 ASP.NET Web Site 模板的新网站 PCSAjaxWebApp1。

修改 Default.aspx 中的代码：

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs"
    Inherits="_Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Pro C# ASP.NET AJAX Sample</title>
</head>
<body>
    <form id="form1" runat="server">
        <asp:ScriptManager ID="ScriptManager1" runat="server" />
        <div>
            <h1>Pro C# ASP.NET AJAX Sample</h1>
            This sample obtains a list of primes up to a maximum value.
            <br />
            Maximum:
            <asp:TextBox runat="server" id="MaxValue" Text="2500" />
            <br />
            Result:
            <asp:UpdatePanel runat="server" ID="ResultPanel">
                <ContentTemplate>
                    <asp:Button runat="server" ID="GoButton" Text="Calculate " />
                    <br />
                    <asp:Label runat="server" ID="ResultLabel" />
                    <br />
                    <small>Panel render time: <% =DateTime.Now.ToLongTimeString() %></small>
                </ContentTemplate>
            </asp:UpdatePanel>
            <asp:UpdateProgress runat="server" ID="UpdateProgress1">
                <ProgressTemplate>
                    <div style="position: absolute; left: 100px; top: 200px;
                        padding: 40px 60px 40px 60px; background-color: lightyellow;
                        border: black 1px solid; font-weight: bold; font-size: larger;
                        filter: alpha(opacity=80);">Updating...</div>
                </ProgressTemplate>
            </asp:UpdateProgress>
            <small>Page render time: <% =DateTime.Now.ToLongTimeString() %></small>
        </div>
```

```

        </form>
    </body>
</html>

```

切换到设计视图(注意 ASP.NET AJAX 控件, 如 UpdatePanel 和 UpdateProgress, 有可视化的设计组件), 双击 Calculate 按钮, 添加一个事件处理程序。修改代码, 如下所示:

```

protected void GoButton_Click(object sender, EventArgs e)
{
    int maxValue = 0;
    System.Text.StringBuilder resultText =
        new System.Text.StringBuilder();
    if (int.TryParse(MaxValue.Text, out maxValue))
    {
        for (int trial = 2; trial <= maxValue; trial++)
        {
            bool isPrime = true;
            for (int divisor = 2; divisor <= Math.Sqrt(trial); divisor++)
            {
                if (trial % divisor == 0)
                {
                    isPrime = false;
                    break;
                }
            }
            if (isPrime)
            {
                resultText.AppendFormat("{0} ", trial);
            }
        }
    }
    else
    {
        resultText.Append("Unable to parse maximum value.");
    }
    ResultLabel.Text = resultText.ToString();
}

```

保存修改的内容, 按下 F5, 运行项目。如果有提示, 就在 Web.config 中启动调试功能。在显示如图 39-2 所示的 Web 页面时, 注意两个显示时间是相同的。

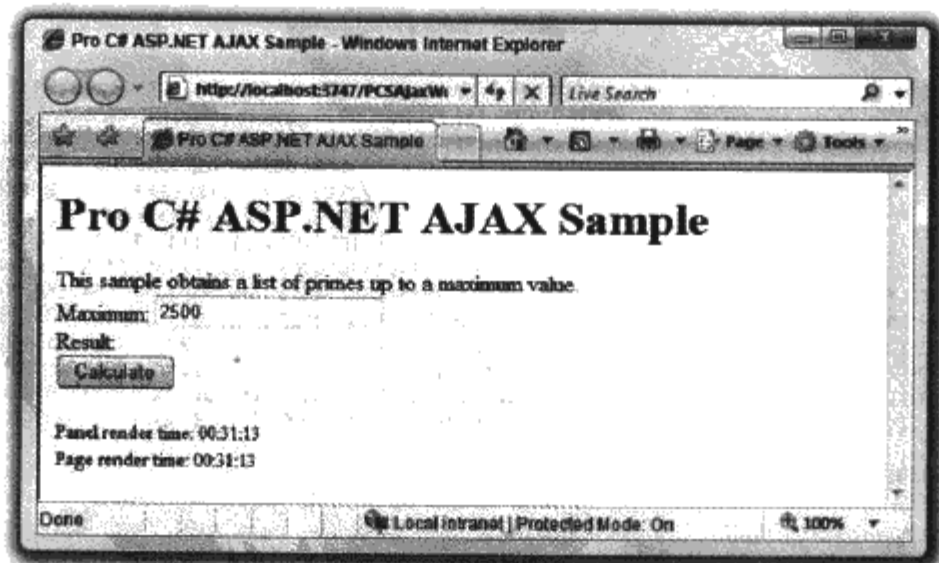


图 39-2

单击 Calculate 按钮, 显示小于等于 2500 的素数。除非在较慢的机器上运行, 否则应立即

得到结果。注意显示时间现在已经不同了，只有 UpdatePanel 中的显示时间改变了，如图 39-3 所示。

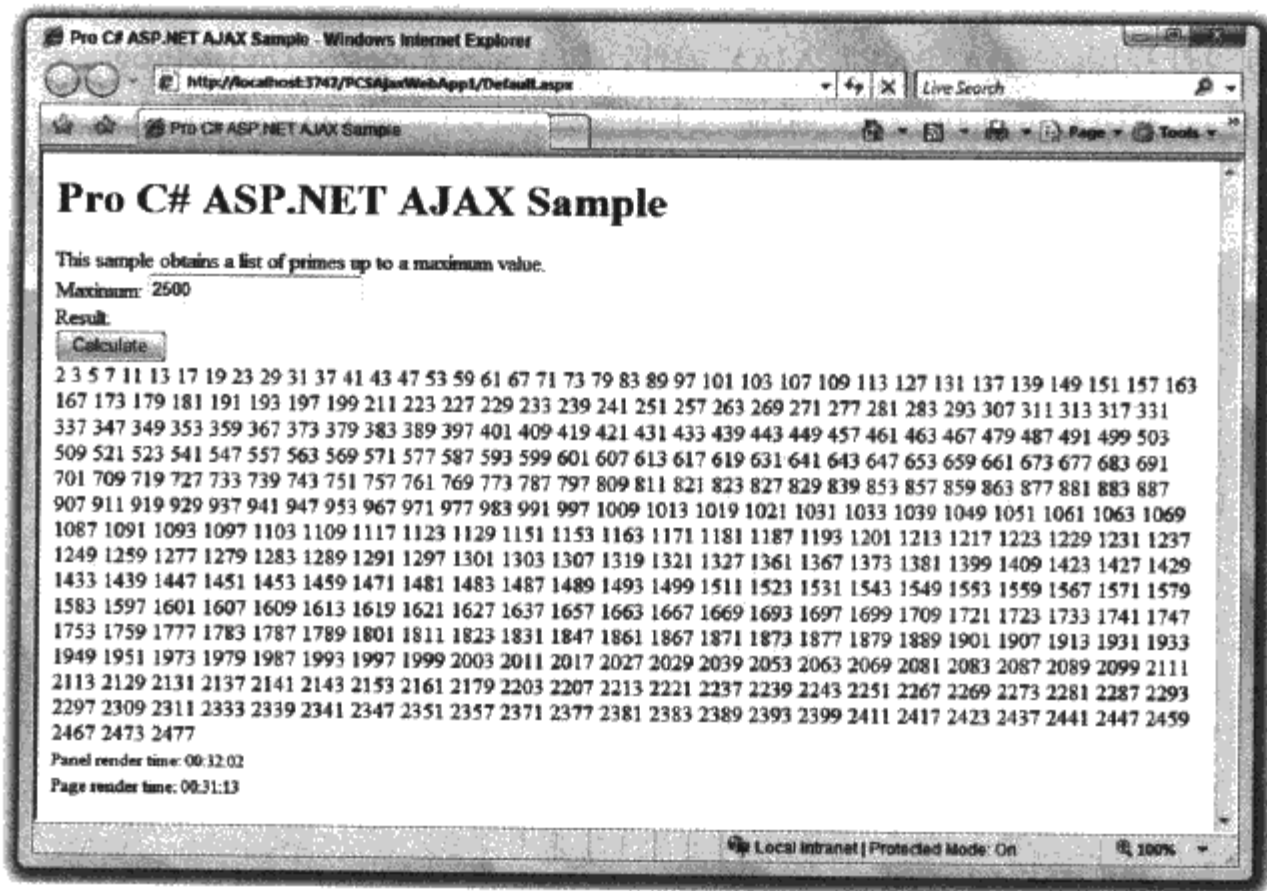


图 39-3

最后，在最大值中添加一些 0，引入一个处理延迟(在较快的 PC 上添加三个 0 就足够了)，再次单击 Calculate 按钮。这次在显示结果之前，注意 UpdateProgress 控件显示一个部分透明的反馈消息，如图 39-4 所示。

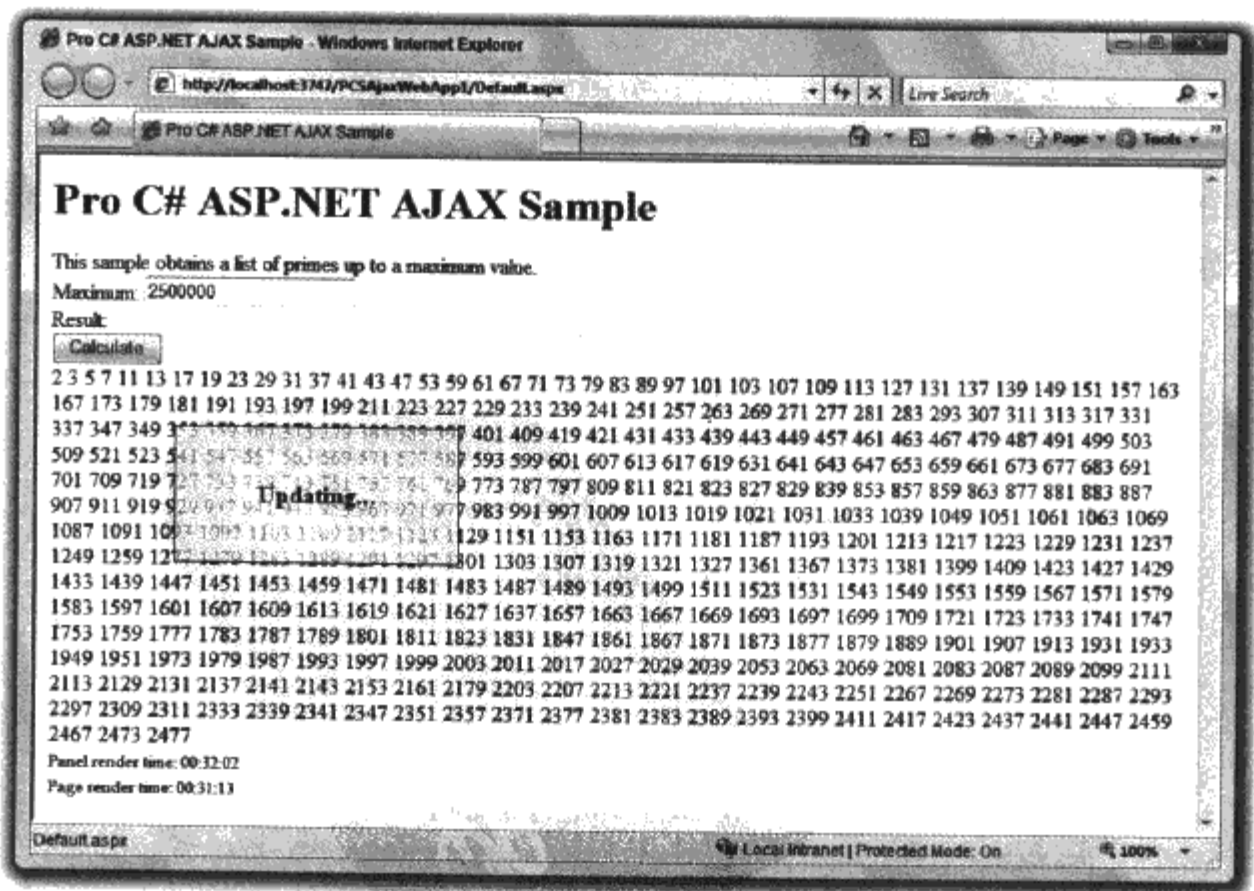


图 39-4



更新应用程序时，页面仍是可以响应的。例如，可以滚动页面。

**提示：**

更新完成时，浏览器的滚动位置设置为单击 Calculate 按钮之前的地方。在大多数情况下，部分页面的更新会很快执行完，这非常有利于可用性。

关闭浏览器，返回 Visual Studio。

### 39.3.2 支持 ASP.NET AJAX 的网站配置

学习了一个简单的支持 ASP.NET AJAX 的 Web 应用程序之后，就可以研究它的工作原理了。首先看看应用程序的 Web.config 文件，尤其是<configuration>的<system.web>配置段中的如下两个代码块：

```
< ?xml version="1.0"? >
< configuration >
  ...
  < system.web >
    < compilation debug="true" >
      < assemblies >
        < add assembly="System.Core, Version=3.5.0.0, Culture=neutral,
          PublicKeyToken=B77A5C561934E089"/ >
        < add assembly="System.Web.Extensions, Version=3.5.0.0,
          Culture=neutral, PublicKeyToken=31BF3856AD364E35"/ >
        < add assembly="System.Data.DataSetExtensions, Version=3.5.0.0,
          Culture=neutral, PublicKeyToken=B77A5C561934E089"/ >
        < add assembly="System.Xml.Linq, Version=3.5.0.0, Culture=neutral,
          PublicKeyToken=B77A5C561934E089"/ >
      < /assemblies >
    < /compilation >
    ...
    < compilation debug="true" >
      < pages >
        < controls >
          < add tagPrefix="asp" namespace="System.Web.UI"
            assembly="System.Web.Extensions, Version=3.5.0.0,
            Culture=neutral,PublicKeyToken=31BF3856AD364E35"/ >
          < add tagPrefix="asp" namespace="System.Web.UI.WebControls"
            assembly="System.Web.Extensions, Version=3.5.0.0,
            Culture=neutral,PublicKeyToken=31BF3856AD364E35"/ >
        < /controls >
      < /pages >
    < /compilation >
    ...
  < /system.web >
  ...
< /configuration >
```

<compilation>中<assemblies>配置段的代码确保，ASP.NET AJAX 程序集 System.Web.Extensions.dll 从 GAC 中加载。<pages>中<controls>配置元素的代码引用这个程序集，将它包含的控件（在 System.Web.UI 和 System.Web.UI.WebControls 命名空间中）关联到标记前缀 asp 上。这两个配置段对于所有支持 ASP.NET AJAX 的 Web 应用程序都是必需的。

下面的两段<httpHandlers>和<httpModules>也是 ASP.NET AJAX 功能所需要的。

<httpHandlers>段定义了三项内容：第一，Web 服务.asmx 的处理程序用 System.Web.Extensions 命名空间中的一个新类替代。这个新类可以通过 AJAX 库处理来自客户端调用的请求，包括 JSON 串行化和并行化。第二，添加一个处理程序，以使用 ASP.NET 应用程序服务。第三，给 ScriptResource.axd 资源添加一个新的处理程序。这个资源用于 ASP.NET AJAX 程序集中的 AJAX 库 JavaScript 文件，这样这些文件就不需要直接包含在应用程序中了。

```
<system.web>
```

```
...
<httpHandlers>
  <remove verb="*" path="*.asmx"/>
  <add verb="*" path="*.asmx" validate="false"
    type="System.Web.Script.Services.ScriptHandlerFactory,
      System.Web.Extensions, Version=1.0.61025.0, Culture=neutral,
      PublicKeyToken=31bf3856ad364e35"/>
  <add verb="*" path="*_AppService.axd" validate="false"
    type="System.Web.Script.Services.ScriptHandlerFactory,
      System.Web.Extensions, Version=1.0.61025.0, Culture=neutral,
      PublicKeyToken=31bf3856ad364e35"/>
  <add verb="GET,HEAD" path="ScriptResource.axd"
    type="System.Web.Handlers.ScriptResourceHandler, System.Web.Extensions,
      Version=1.0.61025.0, Culture=neutral,
      PublicKeyToken=31bf3856ad364e35" validate="false"/>
</httpHandlers>
```

```
</system.web>
```

<httpModules>段添加了一个新的 HTTP 模块，它在 Web 应用程序中添加了 HTTP 请求的其他处理代码。这将支持部分页面的回送。

```
<system.web>
```

```
...
<httpModules>
  <add name="ScriptModule"
    type="System.Web.Handlers.ScriptModule, System.Web.Extensions,
      Version=1.0.61025.0, Culture=neutral,
      PublicKeyToken=31bf3856ad364e35"/>
</httpModules>
```

```
</system.web>
```

其他的配置设置是通过<configSections>设置来确定的，<configSections>设置是<configuration>的第一个子元素。这一段这里没有列出，它必须包含进来，以便使用<system.web.extensions>和<system.webServer>段。

**提示：**

<system.web.extensions>段没有包含在默认的 ASP.NET Web Site 配置文件中，详见下一节。

下一个配置段<system.webServer>包含的设置与 IIS 7 Web 服务器相关，如果使用 IIS 的早期版本，就不需要这一段。这里没有列出这一段。

最后是一个<runtime>段：

```
< runtime >
  < assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1" >
    < dependentAssembly >
```

```

    < assemblyIdentity name="System.Web.Extensions"
      publicKeyToken="31bf3856ad364e35"/ >
    < bindingRedirect oldVersion="1.0.0.0-1.1.0.0" newVersion="3.5.0.0" / >
  < /dependentAssembly >
  < dependentAssembly >
    < assemblyIdentity name="System.Web.Extensions.Design"
      publicKeyToken="31bf3856ad364e35"/ >
    < bindingRedirect oldVersion="1.0.0.0-1.1.0.0" newVersion="3.5.0.0"/ >
  < /dependentAssembly >
< /assemblyBinding >
< /runtime >

```

包含这段是为了确保与 ASP.NET AJAX 的旧版本兼容, 除非安装了 ASP.NET AJAX 1.0 版本, 否则它不会有影响。如果安装了 1.0 版本, 这段就会启动第三方控件, 绑定到 ASP.NET AJAX 的最新版本上。

### 1. 其他配置选项

<system.web.extensions>段包含的设置为 ASP.NET AJAX 提供了其他配置, 这些配置都是可选的。在默认的 ASP.NET Web 应用程序模板中不包含它们, 可以用这个配置段添加的大多数配置都与 Web 服务相关, 包含在<webServices>元素中, 该元素放在<scripting>元素中。首先, 可以添加一段, 通过 Web 服务访问 ASP.NET 身份验证访问(也可以选择强制 SSL):

```

< system.web.extensions >
  < scripting >
    < webServices >
      < authenticationService enabled="true" requireSSL = "true|false"/ >

```

接下来, 通过 profile Web 服务, 启用和配置对 ASP.NET 个性化功能的访问。

```

    < profileService enabled="true"
      readAccessProperties="propertyname1,propertyname2"
      writeAccessProperties="propertyname1,propertyname2" / >

```

最后一个与 Web 服务相关的设置是通过角色 Web 服务启用和配置对 ASP.NET 角色功能的访问。

```

    < roleService enabled="true"/ >
  < /webServices >

```

最后, <system.web.extensions>段包含一个元素, 允许配置异步通信的压缩和缓存:

```

    <scriptResourceHandler enableCompression="true" enableCaching="true" />
  </scripting>
</system.web.extensions>

```

### 2. AJAX Control Toolkit 的其他配置

要使用 AJAX Control Toolkit 中的控件, 可以在 web.config 中添加如下配置:

```

< controls >
  ...
  < add namespace="AjaxControlToolkit" assembly="AjaxControlToolkit"
    tagPrefix="ajaxToolkit"/ >
< /controls >

```

这将工具集中的控件映射到 `ajaxToolkit` 标记前缀上。这些控件包含在 `AjaxControlToolkit.dll` 程序集中，该程序集在 Web 应用程序的 `/bin` 目录下。

还可以使用 `<%@ Register %>` 指令在 Web 页面上注册控件。

```
< %@ Register Assembly="AjaxControlToolkit" Namespace="AjaxControlToolkit"
    TagPrefix="ajaxToolkit" % >
```

### 39.3.3 添加 ASP.NET AJAX 功能

一旦通过网站模板将网站配置为使用 ASP.NET AJAX，或手工配置新 ASP.NET 网站或已有的 ASP.NET 网站，

在网站上添加 AJAX 功能的第一步是在 Web 页面上添加一个 `ScriptManager` 控件，之后，添加 `UpdatePanel` 等服务器控件，以启用部分页面的显示功能，再添加 `Futures CTP` 和 `AJAX Control Toolkit` 中的动态控件，给应用程序增加可用性和功能。还可以添加客户端代码，使用 `AJAX` 库进一步定制和增强应用程序的功能。

本节介绍可以使用服务器控件添加的功能。本章的后面将讨论客户端技术。

#### 1. `ScriptManager` 控件

如本章前面所述，`ScriptManager` 控件必须包含在使用部分页面回送和其他几个 ASP.NET AJAX 功能的所有页面上。

**提示：**

为了确保在 Web 应用程序的所有页面上都包含 `ScriptManager` 控件，必须将这个控件添加到应用程序使用的 master 页面上。

除了启用 ASP.NET AJAX 功能之外，还可以使用属性配置这个控件。在这些属性中，最简单的是 `EnablePartialRendering`，其默认值是 `true`。如果把这个属性设置为 `false`，就禁用了所有异步回送处理功能，例如 `UpdatePanel` 控件提供的页面回送功能。如果要给经理做一个演示，比较支持 AJAX 的网站和传统的网站，就可以这么做。

使用 `ScriptManager` 控件有几个原因，例如下面的情形：

- 确定是否把调用服务器端代码作为部分页面回送的结果
- 添加对其他客户端 JavaScript 文件的引用
- 引用 Web 服务
- 给客户返回错误消息

下面几节介绍这些配置选项。

#### (1) 检测部分页面的回送

`ScriptManager` 控件包含一个布尔属性 `IsInAsyncPostBack`。可以在服务器端代码中使用这个属性，检测部分页面是否正在回送。注意 `ScriptManager` 控件可能在 master 页面上。除了通过 master 页面访问这个控件之外，还可以使用静态方法 `GetCurrent()`，获得对当前 `ScriptManager` 实例的引用。例如：

```
ScriptManager scriptManager = ScriptManager.GetCurrent(this);
if (scriptManager != null && scriptManager.IsInAsyncPostBack)
{
    // Code to execute for partial-page postbacks.
}
```

必须将对 Page 控件的引用传送给 GetCurrent()方法。例如,如果在 ASP.NET Web 页面的 Page\_Load()事件处理程序中使用这个方法,就可以将 this 用作 Page 引用。另外,注意检查 null 引用,以避免异常。

## (2) 客户端 JavaScript 引用

除了在 HTML 页面的标题或页面的 <script> 元素中添加代码之外,还可以使用 ScriptManager 类的 Scripts 属性。这可以使脚本引用集中在一起,更便于维护它们。为此,可以给 <UpdatePanel> 控件元素添加一个 <Scripts> 子元素,再给 <Scripts> 添加 <asp:ScriptReference> 子控件元素。使用 ScriptReference 控件的 Path 属性引用定制脚本。

下面的例子说明了如何在 Web 应用程序的根文件夹下,添加对一个定制脚本文件 MyScript.js 的引用:

```
<asp:ScriptManager runat="server" ID="ScriptManager1">
    <Scripts>
        <asp:ScriptReference Path="~/MyScript.js" />
    </Scripts>
</asp:ScriptManager>
```

## (3) Web 服务引用

为了从客户端 JavaScript 代码中访问 Web 服务,ASP.NET AJAX 必须生成一个代理类。要控制这个操作,可以使用 ScriptManager 类的 Services 属性。与 Scripts 属性一样,也可以以声明方式指定这个属性,但这次要使用 <Services> 元素。给这个元素添加 <asp:ServiceReference> 控件。对于 Services 属性中的每个 ScriptReference 对象,都需要使用 Path 属性指定 Web 服务的路径。

ServiceReference 类也有一个 InlineScript 属性,它默认为 false。这个属性是 false 时,客户端代码向服务器发出请求,会得到一个代理类,来调用 Web 服务。为了改进性能(尤其是在一个页面上使用大量 Web 服务的情况),可以将 InlineScript 设置为 true,这会在页面的客户端脚本中定义代理类。

ASP.NET Web 服务的文件扩展名是 .asmx。如果不想详细阅读本章,但希望在 Web 应用程序的根文件夹下添加对 Web 服务 MyService.asmx 的引用,应使用下面的代码:

```
<asp:ScriptManager runat="server" ID="ScriptManager1">
    <Services>
        <asp:ServiceReference Path="~/MyService.asmx" />
    </Services>
</asp:ScriptManager>
```

采用这种方式只能添加对本地 Web 服务的引用(即 Web 服务和调用代码在同一个 Web 应用程序中)。可以通过本地 Web 方法间接调用远程 Web 服务。

本章的后面将讨论如何从客户端 JavaScript 代码中异步调用 Web 方法,这些方法是以这种方式使用代理类生成的。



#### (4) 客户端错误消息

如果在部分页面的回送过程中抛出了异常，默认操作是将异常包含的错误消息放在客户端 JavaScript 警报消息框中。处理 ScriptManager 实例的 AsyncPostBackError 事件，可以定制要显示的消息。在这个事件的处理程序中，可以使用 AsyncPostBackErrorEventArgs.Exception 属性访问抛出的异常，使用 ScriptManager.AsyncPostBackErrorMessage 属性设置显示给客户端的消息。这么做可以给用户隐藏异常细节。

如果要重写默认操作，以另一种方式显示消息，就必须使用 JavaScript 处理客户端对象 PageRequestManager 的 endRequest 事件，详见本章后面的内容。

## 2. 使用 UpdatePanel 控件

UpdatePanel 控件是编写支持 ASP.NET AJAX 的 Web 应用程序时最常用的控件。如本章前面的简单例子所述，这个控件可以封装 Web 页面的一部分，使之参与部分页面的回送操作。为此，要在页面上添加一个 UpdatePanel 控件，用需要的控件填充其子元素 <ContentTemplate>。

```
<asp:UpdatePanel runat="Server" ID="UpdatePanel1">
  <ContentTemplate>
    ...
  </ContentTemplate>
</asp:UpdatePanel>
```

根据 UpdatePanel 控件的 RenderMode 属性值，<ContentTemplate>模板的内容显示在<div>或<span>元素中。这个属性的默认值是 Block，即显示在<div>元素中。要使用<span>元素，应将 RenderMode 属性设置为 Inline。

#### (1) 一个 Web 页面上的多个 UpdatePanel 控件

可以在一个页面上包含任意多个 UpdatePanel 控件。如果回送操作是由包含在页面上的任一个 UpdatePanel 控件的<ContentTemplate>模板中的控件引发，就进行部分页面的回送，而不是整个页面的回送。这会使所有的 UpdatePanel 控件根据其 UpdateMode 属性值进行更新。这个属性的默认值是 Always，表示 UpdatePanel 为页面上的部分页面回送操作而更新，即使这个操作是由另一个 UpdatePanel 控件引发的，也是如此。如果把这个属性设置为 Conditional，UpdatePanel 就仅在它包含的控件引发部分页面回送操作时更新，或者在启动了已定义的触发器时更新。触发器稍后介绍。

如果把 UpdateMode 属性设置为 Conditional，还可以将 ChildrenAsTriggers 属性设置为 false，禁止 UpdatePanel 包含的控件触发 UpdatePanel 的更新操作。但要注意，在这种情况下，这些控件仍会触发一个部分页面回送操作，它会使页面上的其他 UpdatePanel 进行更新。例如，这会使 UpdateMode 属性值为 Always 的 UpdatePanel 进行更新，如下面的代码所示：

```
<asp:UpdatePanel runat="Server" ID="UpdatePanel1" UpdateMode="Conditional"
  ChildrenAsTriggers="false">
  <ContentTemplate>
    <asp:Button runat="Server" ID="Button1" Text="Click Me" />
    <small>Panel 1 render time: <% =DateTime.Now.ToLongTimeString() %></small>
  </ContentTemplate>
</asp:UpdatePanel>
<asp:UpdatePanel runat="Server" ID="UpdatePanel2">
```

```

<ContentTemplate>
  <small>Panel 2 render time: <% =DateTime.Now.ToLongTimeString() %></small>
</ContentTemplate>
</asp:UpdatePanel>
<small>Page render time: <% =DateTime.Now.ToLongTimeString() %></small>

```

在这段代码中，UpdatePanel2 控件的 UpdateMode 属性值设置为默认的 Always。单击按钮时，会引发一个部分页面回送操作，但只更新 UpdatePanel2。注意只更新了“Panel2 render time”标签。

## (2) 服务器端的 UpdatePanel 更新

有时页面上有多个 UpdatePanel 控件，可以不更新其中的一个，除非满足某些条件。在这种情况下，应将 UpdatePanel 的 UpdateMode 属性设置为 Conditional，如上一节所述，再把 ChildrenAsTriggers 属性设置为 false。接着，对于页面上引发部分页面回送操作的控件，在服务器端的事件处理程序中，(有条件地)调用 UpdatePanel 的 Update()方法，例如：

```

protected void Button1_Click(object sender, EventArgs e)
{
    if (TestSomeCondition())
    {
        UpdatePanel1.Update();
    }
}

```

## (3) UpdatePanel 的触发器

给 Web 页面上其他地方的控件的 Triggers 属性添加触发器，就可以通过该控件更新 UpdatePanel 控件。触发器是 Web 页面上其他地方的控件的事件与 UpdatePanel 控件之间的关联。所有的控件都有默认事件(如按钮控件的默认事件是 Click)，所以可以不指定事件名。有两种触发器可以添加，它们用两个类表示：

- AsyncPostBackTrigger: 这个类会在指定控件的指定事件发生时，更新 UpdatePanel 控件。
- PostBackTrigger: 这个类会在指定控件的指定事件发生时，更新整个页面。

一般使用 AsyncPostBackTrigger，但如果希望 UpdatePanel 中的一个控件引发整个页面的回送操作，就可以使用 PostBackTrigger。

这两个触发器类有两个属性 ControlID 和 EventName，ControlID 指定了通过其标识符启动触发器的控件，EventName 指定了控件中链接到触发器上的事件名。

为了扩展前面的例子，考虑下面的代码：

```

<asp:UpdatePanel runat="Server" ID="UpdatePanel1" UpdateMode="Conditional"
  ChildrenAsTriggers="false">
  <Triggers>
    <asp:AsyncPostBackTrigger ControlID="Button2" />
  </Triggers>
  <ContentTemplate>
    <asp:Button runat="Server" ID="Button1" Text="Click Me" />
    <small>Panel 1 render time: <% =DateTime.Now.ToLongTimeString() %></small>
  </ContentTemplate>
</asp:UpdatePanel>
<asp:UpdatePanel runat="Server" ID="UpdatePanel2">
  <ContentTemplate>
    <asp:Button runat="Server" ID="Button2" Text="Click Me" />
  </ContentTemplate>
</asp:UpdatePanel>

```

```

    <small>Panel 2 render time: <% =DateTime.Now.ToLongTimeString() %></small>
  </ContentTemplate>
</asp:UpdatePanel>
<small>Page render time: <% =DateTime.Now.ToLongTimeString() %></small>

```

新的按钮控件 Button2 指定为 UpdatePanel1 中的一个触发器。单击这个按钮时，会更新 UpdatePanel1 和 UpdatePanel2。更新 UpdatePanel1 是因为启动了触发器，更新 UpdatePanel2 是因为它使用了 UpdateMode 的默认值 Always。

### 3. 使用 UpdateProgress

如前面的例子所示，UpdateProgress 控件允许在部分页面的回送过程中给用户显示进度消息。使用 ProgressTemplate 属性可提供显示进度的模板，为此，一般使用控件的 <ProgressTemplate> 元素。

使用 AssociatedUpdatePanelID 属性将 UpdateProgress 控件与指定的 UpdatePanel 关联起来，就可以在页面上放置多个 UpdateProgress 控件。如果没有设置该属性(默认)，无论哪个 UpdatePanel 引发了部分页面回送操作，都显示 UpdateProgress 模板。

在执行部分页面回送操作时，显示 UpdateProgress 模板之前有一个延迟。这个延迟可以通过 DisplayAfter 属性来配置，DisplayAfter 是一个 int 属性，指定了延迟时间(单位是毫秒)，默认为 500 毫秒。

最后，可以使用布尔属性 DynamicLayout 指定在显示模板之前，是否为模板分配空间。这个属性的默认值是 true，此时页面上的空间是动态分配的，所以为了在线显示进度模板，需要删除其他控件。如果把这个属性设置为 false，就在显示模板之前，为模板分配空间，这样页面上其他控件的布局就不会改变。可以根据显示进度时要达到的效果设置这个属性。对于使用绝对坐标定位的进度模板，如前面的例子所示，应将这个属性设置为默认值。

### 4. 使用扩展器控件

ASP.NET AJAX 的核心软件包包含一个类 ExtenderControl，它的作用是允许扩展其他 ASP.NET 服务器控件(即增加功能)。它广泛应用于 AJAX Control Toolkit，效果不错。可以使用 AJAX Control Toolkit 中的模板创建自己的扩展控件。ExtenderControl 控件的工作方式都是类似的：把它们放在页面上，与目标控件关联起来，添加进一步的配置。接着扩展器就会执行客户端代码，以添加功能。

为了了解扩展控件，在一个简单的例子中创建一个新的网站 PCSExtenderDemo，放在 C:\ProCSharp\Chapter39 目录下，把 AJAX Control Toolkit 程序集添加到网站的 bin 目录下，给 Default.aspx 添加如下代码：

```

<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs"
    Inherits="_Default" %>
< % Register Assembly="AjaxControlToolkit"
    Namespace="AjaxControlToolkit" TagPrefix="ajaxToolkit" % >

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Color Selector</title>

```

```

</head>
<body>
  <form id="form1" runat="server">
    <asp:ScriptManager ID="ScriptManager1" runat="server" />
    <div>
      <asp:UpdatePanel runat="server" ID="updatePanel1">
        <ContentTemplate>
          <span style="display: inline-block; padding: 2px;">
            My favorite color is:
          </span>
          <asp:Label runat="server" ID="favoriteColorLabel" Text="green"
            style="color: #00dd00; display: inline-block; padding: 2px;
            width: 70px; font-weight: bold;" />
          <ajaxToolkit:DropDownExtender runat="server" ID="dropDownExtender1"
            TargetControlID="favoriteColorLabel" DropDownControlID="colDropDown" />
          <asp:Panel ID="colDropDown" runat="server"
            Style="display: none; visibility: hidden; width: 60px; padding: 8px;
            border: double 4px black; background-color: #ffffdd;
            font-weight: bold;">
            <asp:LinkButton runat="server" ID="OptionRed" Text="red"
              OnClick="OnSelect" style="color: #ff0000;" /><br />
            <asp:LinkButton runat="server" ID="OptionOrange" Text="orange"
              OnClick="OnSelect" style="color: #dd7700;" /><br />
            <asp:LinkButton runat="server" ID="OptionYellow" Text="yellow"
              OnClick="OnSelect" style="color: #dddd00;" /><br />
            <asp:LinkButton runat="server" ID="OptionGreen" Text="green"
              OnClick="OnSelect" style="color: #00dd00;" /><br />
            <asp:LinkButton runat="server" ID="OptionBlue" Text="blue"
              OnClick="OnSelect" style="color: #0000dd;" /><br />
            <asp:LinkButton runat="server" ID="OptionPurple" Text="purple"
              OnClick="OnSelect" style="color: #dd00ff;" />
          </asp:Panel>
        </ContentTemplate>
      </asp:UpdatePanel>
    </div>
  </form>
</body>
</html>

```

还需要在这个文件的后台代码中添加如下事件处理程序:

```

protected void OnSelect(object sender, EventArgs e)
{
    favoriteColorLabel.Text = ((LinkButton)sender).Text;
    favoriteColorLabel.Style["color"] = ((LinkButton)sender).Style["color"];
}

```

在浏览器中, 刚开始并没有显示很多内容, 扩展器似乎没有什么作用, 如图 39-5 所示。

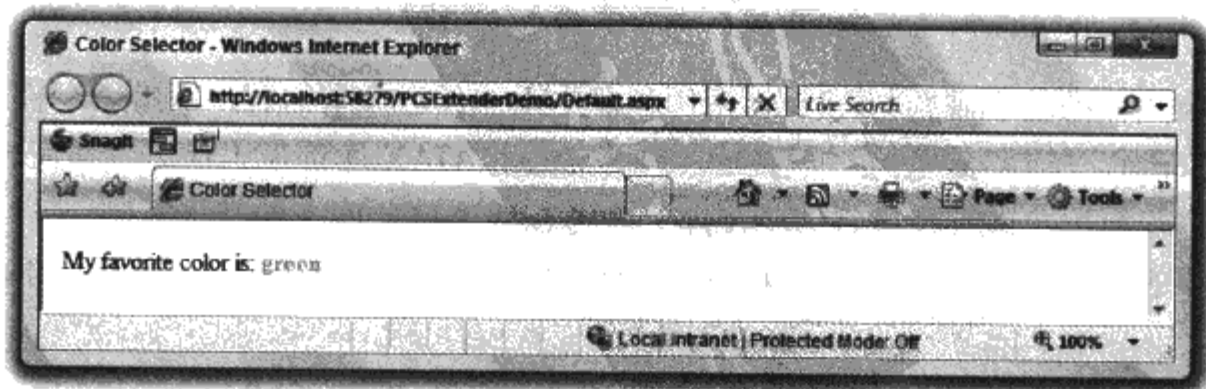


图 39-5



但是，把鼠标停在文本“green”上时，会动态显示一个下拉框。如果单击这个下拉框，就会显示一个列表，如图 39-6 所示。

单击下拉列表中的一个链接时，文本会改变(一个部分页面回送操作之后)。

对于这个简单的例子，要注意两点。第一，非常容易将扩展器与目标控件关联起来。第二，下拉列表用定制代码设置了样式，这表示可以在列表中放置任意内容。这个简单的扩展器是给 Web 应用程序添加功能的有效方式，使用起来也很简单。

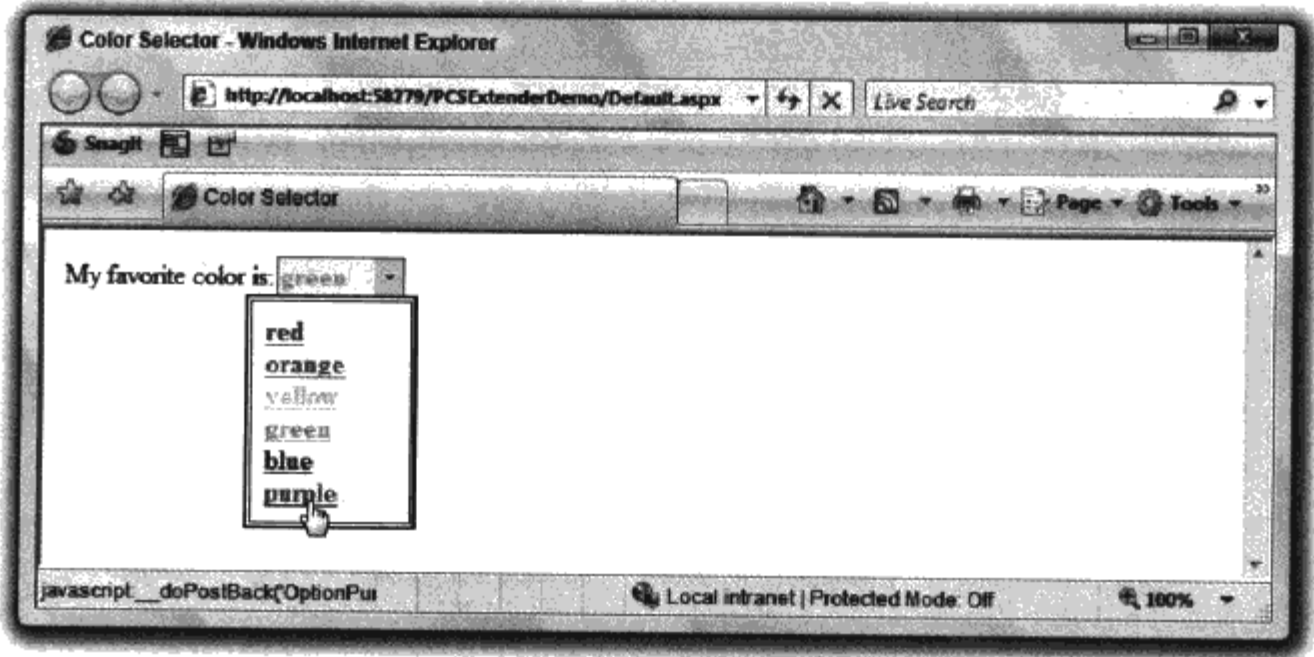


图 39-6

AJAX Control Toolkit 中的扩展器在不断增加和更新，所以请定期访问 <http://ajax.asp.net/ajaxtoolkit>。这个 Web 页面包含所有当前的扩展器的实时演示，可以看到它们的工作情况。

除了 AJAX Control Toolkit 提供的扩展器控件之外，还可以创建自己的扩展器控件。为了使这个过程尽可能简单，可以使用项目模板 ASP.NET AJAX Control Project。这个项目包含扩展器需要的所有基本功能，如用于扩展器的服务器端类和扩展器使用的客户端 JavaScript 文件。要创建有效的扩展器，必须使用 AJAX 库。

39.3.4 使用 AJAX 库

AJAX 库有许多可进一步增强 Web 应用程序的功能。但是，为了增强 Web 应用程序，至少需要了解 JavaScript 的基本知识。本节将介绍 AJAX 库提供的一些功能，但这不是一个全面的教程。

使用 AJAX 库的基本规则与在 Web 应用程序中添加任意类型的客户端脚本一样，仍使用核心语言 JavaScript，与 DOM 交互。但是，在许多方面，AJAX 库都使工作更容易完成。本节将学习这些内容，为用户进一步试验 AJAX 库、参考在线 AJAX 库文档打下基础。

本节介绍的技术都在 PCSLibraryDemo 项目中演示，该项目将贯穿本章的剩余内容。



### 1. 给 Web 页面添加 JavaScript

首先需要了解的是如何给 Web 页面添加客户端 JavaScript, 这里有三个选项:

- 使用<script>元素, 在 ASP.NET Web 页面上在线添加 JavaScript
- 将 JavaScript 添加到单独的 JavaScript 文件中, 其扩展名是.js, 再使用 ScriptManager 控件的<Scripts>子元素(首选)或从<script>元素中引用这些文件
- 从服务器端代码中生成 JavaScript, 例如后台代码或定制的扩展器控件

这些技术都有自己的优点。对于原型代码, 在线编码是无可替代的, 因为它非常快, 易于使用。将 HTML 元素的客户端事件处理程序和带客户端函数的服务器控件关联起来是很容易的, 因为所有的代码都在同一个文件中。

使用单独的文件有利于代码重用, 因为可以创建自己的类库, 这类似于已有的 AJAX 库 JavaScript 文件。

从后台代码中生成代码较难实现, 因为我们通常不能像使用 C#那样在编写 JavaScript 代码时访问 IntelliSense。但是, 可以动态生成代码, 以响应应用程序的状态, 有时这是完成任务的唯一方式。

可以用 AJAX Control Toolkit 创建的扩展器包含一个独立的 JavaScript 文件, 它用于定义操作, 解决将客户端代码显示到服务器上的一些问题。

本章将使用在线编码技术, 因为它最简单, 允许我们只关注 JavaScript 功能。

### 2. 全局实用函数

AJAX 库提供的一个最常用的特性是封装了其他功能的全局函数集, 包括:

- `$get()`: 这个函数可以获得 DOM 元素的一个引用, 它将其客户端 id 值提供为一个参数, 可选的第二个参数指定了要搜索的父元素。
- `$create()`: 这个函数可以创建指定 JavaScript 类型的对象, 同时进行初始化。可以给这个函数提供 1~5 个参数。第一个参数是要实例化的类型, 它一般是由 AJAX 库定义的一个类型。其他参数分别指定了属性的初始值、事件处理程序、其他组件的引用和对对象要关联的 DOM 对象。
- `$addHandler()`: 这个函数为给对象添加事件处理程序提供了一种缩写方式。

还有更多的全局函数, 但这些是最常用的全局函数, 尤其是`$create()`, 它可以大大减少初始化对象所需的代码量。

### 3. 使用 AJAX 库 JavaScript OOP 扩展

AJAX 库包含一个增强的架构, 它定义了使用基于 OOP 的系统的类型, 与 .NET Framework 技术紧密相关。可以创建命名空间, 给命名空间添加类型, 为类型添加构造函数、方法、属性和事件, 甚至可以在类型定义上使用继承和接口。

本节将介绍如何使用这个功能的基本内容, 但这里没有探讨事件和接口。这些结构超出了本章的范围。

#### (1) 定义命名空间

要定义命名空间, 应使用 `Type.registerNamespace()` 函数, 例如:

```
Type.registerNamespace("ProCSharp");
```

注册了命名空间后，就可以给它添加类型了。

## (2) 定义类

定义类需要三步。第一，定义构造函数，第二，添加属性和方法，第二，注册该类。

要定义构造函数，需要使用命名空间和类名来定义一个函数，例如：

```
ProCSharp.Shape = function(color, scaleFactor) {
    this._color = color;
    this._scaleFactor = scaleFactor;
}
```

这个构造函数带两个参数，使用它们设置本地字段(注意不一定要明确定义这些字段，只需设置它们的值)。

要添加属性和方法，应给它们赋予类的 `Prototype` 属性，如下所示：

```
ProCSharp.Shape.prototype = {
    getColor : function() {
        return this._color;
    },
    setColor : function(color) {
        this._color = color;
    },
    getScaleFactor : function() {
        return this._scaleFactor;
    },
    setScaleFactor : function(scaleFactor) {
        this._scaleFactor = scaleFactor;
    }
}
```

这段代码提供 `get` 和 `set` 存取器定义了两个属性：

要注册类，应调用其 `registerClass()` 函数：

```
ProCSharp.Shape.registerClass('ProCSharp.Shape');
```

## (3) 继承

派生类的方式与创建类相同，但有一些小区别。在构造函数中使用 `initializeBase()` 函数初始化基类，以数组的形式传送参数：

```
ProCSharp.Circle = function(color, scaleFactor, diameter) {
    ProCSharp.Circle.initializeBase(this, [color, scaleFactor]);
    this._diameter = diameter;
}
```

用前面的方式定义属性和方法：

```
ProCSharp.Circle.prototype = {
    getDiameter : function() {
        return this._diameter;
    }
}
```

```

    },

    setDiameter : function(diameter) {
        this._diameter = diameter;
    },

    getArea : function() {
        return Math.PI * Math.pow((this._diameter * this._scaleFactor) / 2, 2);
    },

    describe : function() {
        var description = "This is a " + this._color + " circle with an area of "
            + this.getArea();
        alert(description);
    }
}

```

注册类时，要把基类类型提供为第二个参数：

```
ProCSharp.Circle.registerClass('ProCSharp.Circle', ProCSharp.Shape);
```

将它们传送为其他参数，可以实现接口，但这里为了简单，没有提供其细节。

#### (4) 使用用户定义的类型

以这种方式定义了类之后，就可以通过简单的语法实例化和使用它们了。例如：

```
var myCircle = new ProCSharp.Circle('red', 1.0, 4.4);
myCircle.describe();
```

这段代码会显示一个 JavaScript 警报框，如图 39-7 所示。

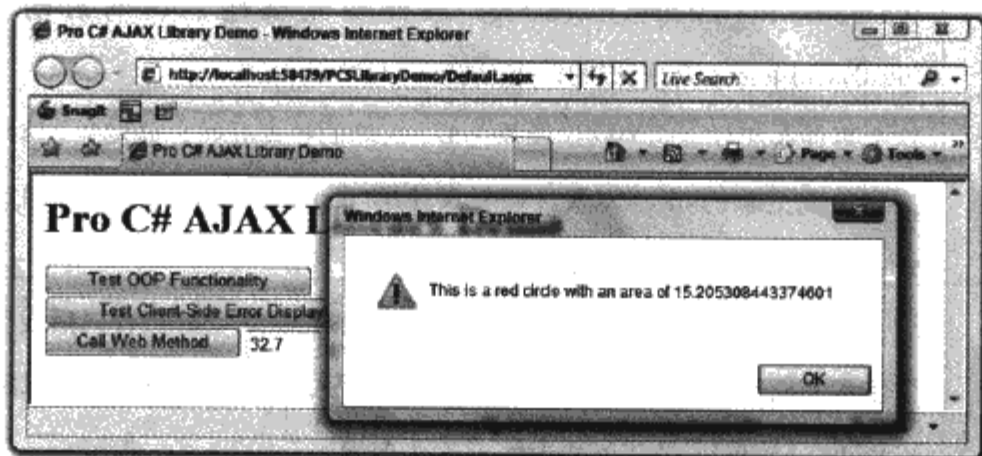


图 39-7

如果要测试一下，可以运行 PCSLibraryDemo 项目，单击 Text OOP Functionality 按钮。

#### 4. PageRequestManager 和 Application 对象

在 AJAX 库中，最有用的类是 PageRequestManager 和 Application。PageRequestManager 在 Sys.WebForms 命名空间中，Application 在 Sys 命名空间中。对于这两个类，重要的是它们提供的几个事件可以与 JavaScript 事件处理程序关联起来。这些事件在页面生存期(用于 Application)或部分页面回送过程(用于 PageRequestManager)中非常有趣的点发生，可以在这些关键时刻执行操作。

AJAX 库定义事件处理程序的方式类似于 .NET Framework 中的事件处理程序的定义方式。

每个事件处理程序都有类似的签名，带两个参数。第一个参数是对生成事件的对象的引用。第二个参数是 `Sys.EventArgs` 类的一个实例或派生自这个类的一个子类的实例。`PageRequestManager` 和 `Application` 提供的许多事件都包括专门的事件变元类，它可以用于确定事件的更多信息。表 39-4 按照事件的发生顺序列出了这些事件，先是加载页面，然后启动部分页面的回送操作，最后是关闭页面。

表 39-4

事 件	说 明
<code>Application</code> <code>.init</code>	这个事件在页面的生存期中是第一个发生的，它在加载了所有的 JavaScript 文件之后、创建应用程序中的对象之前发生
<code>Application</code> <code>.load</code>	这个事件在加载并初始化了应用程序中的对象后发生。这个事件经常关联一个事件处理程序，在页面第一次加载时执行操作。也可以为页面上的 <code>pageLoad()</code> 函数提供实现代码，该函数自动定义为这个事件的处理程序。使用 <code>Sys.ApplicationLoadEventArgs</code> 对象传送事件变元，该对象包含 <code>IsPartialLoad</code> 属性，用于确定是否已启动了部分页面的回送操作。用 <code>get_IsPartialLoad()</code> 存取器访问这个属性
<code>PageRequestManager</code> <code>.initializeRequest</code>	这个事件在部分页面的回送操作之前、创建请求对象之前发生。可以使用 <code>Sys.WebForms.InitializeRequestEventArgs</code> 事件变元属性，访问启动回送操作的元素 ( <code>postBackElement</code> ) 和底层的请求对象 ( <code>request</code> )
<code>PageRequestManager</code> <code>.beginRequest</code>	这个事件在部分页面的回送操作之前、创建请求对象之后发生。可以使用 <code>Sys.WebForms.BeginRequestEventArgs</code> 事件变元属性，访问启动回送操作的元素 ( <code>postBackElement</code> ) 和底层的请求对象 ( <code>request</code> )
<code>PageRequestManager</code> <code>.pageLoading</code>	这个事件在部分页面的回送操作之后、后续的处理开始之前发生。这个处理过程可以包含要删除或更新的 <code>&lt;div&gt;</code> 元素，该元素使用 <code>panelsDeleting</code> 和 <code>panelsUpdating</code> 属性，通过 <code>sys.WebForms.PageLoadingEventArgs</code> 对象来引用
<code>PageRequestManager</code> <code>.pageLoaded</code>	这个事件在部分页面的回送操作之后、处理 <code>UpdatePanel</code> 控件之后发生。这个处理过程可以包含要创建或更新的 <code>&lt;div&gt;</code> 元素，该元素使用 <code>panelsCreated</code> 和 <code>panelsUpdated</code> 属性，通过 <code>WebForms.PageLoadedEventArgs</code> 对象来引用
<code>PageRequestManager</code> <code>.endRequest</code>	这个事件完成部分页面的回送操作之后发生。传送给事件处理程序的 <code>System.WebForms.EndRequestEventArgs</code> 对象可以检测和处理服务器端错误(使用 <code>error</code> 和 <code>errorHandled</code> 属性)，通过 <code>response</code> 访问响应对象
<code>Application</code> <code>.unload</code>	这个事件在删除应用程序中的对象之前发生，以便执行最后的操作或清理任务

使用静态的 `add_xxx()` 方法，可以给 `Application` 对象的事件添加事件处理程序，例如：

```

Sys.Application.add_load(LoadHandler);

function LoadHandler(sender, args)
{

```



```
// Event handler code.
}
```

PageRequestManager 的过程与此类似, 但必须使用 `get_instance()` 函数获得当前对象的一个实例, 例如:

```
Sys.WebForms.PageRequestManager.getInstance().add_beginRequest(
    BeginRequestHandler);

function BeginRequestHandler(sender, args)
{
    // Event handler code.
}
```

在 PCSLibraryDemo 应用程序中, 为 `PageRequestManager.endRequest` 添加了一个事件处理程序。这个事件处理程序响应服务器端处理的错误, 在 id 为 `errorDisplay` 的 `<span>` 元素中显示一个错误消息。要测试这个方法, 可以单击 Test Client-Side Error Display 按钮, 如图 39-8 所示。

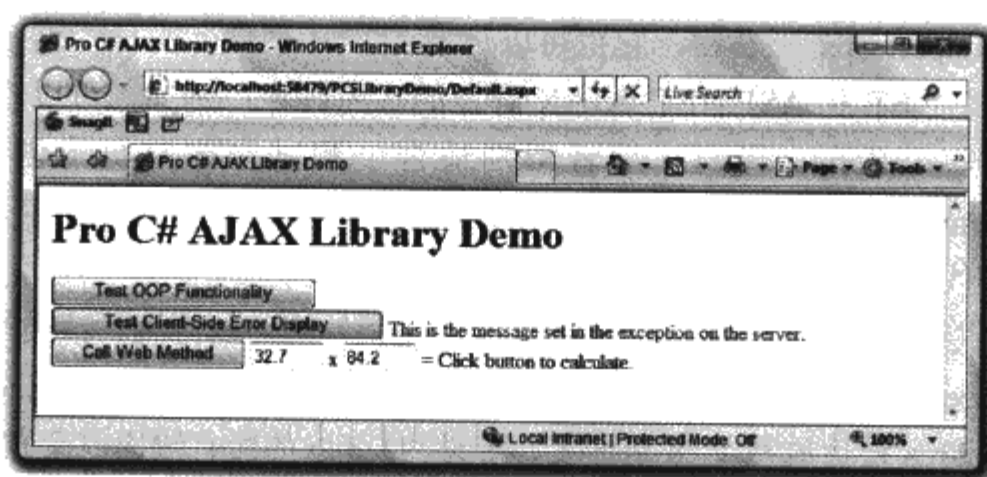


图 39-8

得到该结果的代码如下:

```
Sys.WebForms.PageRequestManager.getInstance().add_endRequest(EndRequestHandler);

function EndRequestHandler(sender, args)
{
    if (args.get_error() != undefined)
    {
        var errorMessage = args.get_error().message;
        args.set_errorHandled(true);
        $get('errorDisplay').innerHTML = errorMessage;
    }
}
```

注意 `EndRequestEventArgs` 对象的 `errorHandled` 属性设置为 `true`, 这会禁止执行默认操作, 即使用 JavaScript 的 `alert()` 函数在对话框中显示错误消息。

在服务器上抛出一个异常, 就生成了错误, 如下所示:

```
protected void testErrorDisplay_Click(object sender, EventArgs e)
{
    throw new ApplicationException(
        "This is the message set in the exception on the server.");
}
```

还有许多情形可以使用事件处理技术, 来处理 `PageRequestManager` 和 `Application` 的事件。



## 5. JavaScript 的调试

JavaScript 很难调试是出了名的。但这在 VS 的最新版本中得到了解决。现在可以像 C# 代码那样在 JavaScript 代码中添加断点，单步执行代码了。还可以在中断模式下查询典型状态，改变属性值等。编写 JavaScript 代码时使用的 IntelliSense 功能也在 VS 的最新版本中得到了很大改进。

有时还希望添加调试和跟踪代码，在代码执行过程中报告信息，例如使用 JavaScript 的 `alert()` 函数在对话框中显示信息。

有一些第三方工具可用于添加调试的客户端 UI，包括：

- **Fiddler**: 这个工具可以从 [www.fiddlertool.com](http://www.fiddlertool.com) 上获得，它可以记录计算机和 Web 应用程序之间的所有 HTTP 通信，包括部分页面的回送。还有一些工具可以查看在处理 Web 页面的过程中发生的事件的详细信息。
- **Nikhil 的 Web Development Helper**: 这个工具可以从 <http://projects.nikhilk.net/Projects/WebDevHelper.aspx> 上获得，它也可以记录 HTTP 通信。另外，这个工具包含许多专门用于 ASP.NET 和 ASP.NET AJAX 开发的实用程序，例如，可以查看视图状态，执行即时的 JavaScript 代码。后者特别适合于测试在客户机上创建的对象。Web Development Helper 还在发生 JavaScript 错误时显示其他错误信息，更便于跟踪 JavaScript 代码中的错误。

AJAX 库也提供了 `Sys.Debug` 类，给应用程序添加额外的调试特性。该类的一个最有用的特性是 `Sys.Debug.traceDump()` 函数，它可以分析对象，使用这个函数的一种方式是将一个 `id` 为 `TraceConsole` 的 `textarea` 控件放在 Web 页面上，接着，`Debug` 的所有输出就会发送到这个控件上。例如，可以使用 `traceDump()` 方法，将 `Application` 对象的信息输出到控制台上：

```

Sys.Application.add_load(LoadHandler);

function LoadHandler(sender, args)
{
    Sys.Debug.traceDump(sender);
}

```

这会得到如下输出：

```

traceDump (Sys.Application)
  _updating: false
  _id: null
  _disposing: false
  _creatingComponents: false
  _disposableObjects {Array}
  _components {Object}
  _createdComponents {Array}
  _secondPassComponents {Array}
  _loadHandlerDelegate: null
  _events {Sys.EventHandlerList}
  _list {Object}
    load {Array}
      [0] {Function}
  _initialized: true
  _initializing: true

```

在这个输出中,可以看到该对象的所有属性。这个属性特别适合于 ASP.NET AJAX 开发。

## 6. 异步调用 Web 方法

ASP.NET AJAX 的一个最强大的特性是可以从客户端脚本中调用 Web 方法,这就允许访问数据、服务器端处理和其他功能。

本书的第 42 章介绍了 Web 方法,所以这里不详细介绍它。但是要讨论一些基本知识。简言之,Web 方法是可以在 Web 服务中提供,能通过 Internet 访问远程资源的方法。在 ASP.NET AJAX 中,还可以将 Web 方法用作服务器端 Web 页面的后台代码中的静态方法。在 Web 方法中使用参数和返回值的方式与其他方法类型相同。

在 ASP.NET AJAX 中,Web 方法是异步调用的。给 Web 方法传送参数,定义一个回调函数,当 Web 方法调用完成时,就会调用这个回调函数。该回调函数用于处理 Web 方法的响应。也可以提供另一个回调函数,以处理调用失败的情况。

在 PCSLibraryDemo 应用程序中,单击 Call Web Method 按钮时,就调用一个 Web 方法,如图 39-9 所示。

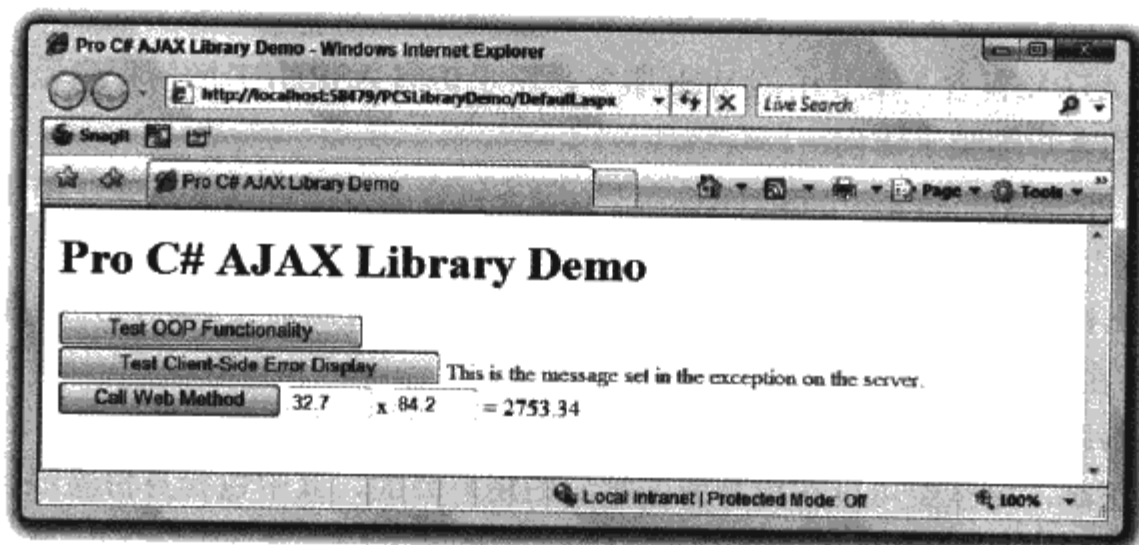


图 39-9

在客户端脚本中使用 Web 方法之前,必须生成一个客户端代理类,以进行通信。为此,最简单的方式是在 ScriptManager 控件中,引用包含 Web 方法的 Web 服务的 URL:

```
<asp:ScriptManager ID="ScriptManager1" runat="server">
  <Services>
    <asp:ServiceReference Path="~/SimpleService.asmx" />
  </Services>
</asp:ScriptManager>
```

ASP.NET Web 服务使用扩展名.asmx,如上面的代码所示。为了使用客户端代理访问 Web 服务中的 Web 方法,必须给 Web 服务应用 System.Web.Script.Services.ScriptService 属性。

对于 Web 页面的后台代码中的 Web 方法,不需要这个属性,或者 ScriptManager 中的这个引用,但必须使用静态方法,给方法引用 System.Web.Services.WebMethod 属性。

生成了客户端代理后,就可以通过名称访问 Web 方法了,它定义为类中与 Web 服务同名的一个函数。在 PCSLibraryDemo 中,Web 服务 SimpleService.asmx 的一个 Web 方法是 Multiply(),它对两个 double 参数执行相乘操作。从客户端代码中调用这个方法时,要传送方法需要的两个

参数(在例子中, 从 HTML `<input>` 元素中获得), 可以传送一个或两个回调函数引用。如果传送一个引用, 这个回调函数就在调用成功返回时使用这个引用。如果传送了两个引用, 第二个引用就在 Web 方法失败时使用。

在 PCSLibraryDemo 中, 使用了一个回调函数, 它提取 Web 方法调用的结果, 并将它赋予 id 为 `webMethodResult` 的 `<span>` 元素:

```
function callWebMethod()
{
    SimpleService.Multiply(parseFloat($get('xParam').value),
        parseFloat($get('yParam').value), multiplyCallBack);
}

function multiplyCallBack(result)
{
    $get('webMethodResult').innerHTML = result;
}
```

这个方法非常简单, 但演示了从客户端代码中异步调用 Web 服务的便利性。

## 7. ASP.NET 应用程序服务

ASP.NET AJAX 包含 3 个专用的 Web 服务, 用于访问 ASP.NET 应用程序服务。这些服务可以通过下面的客户端类来访问:

- `Sys.Services.AuthenticationService`: 这个服务包含的方法可以登录或注销用户, 或确定用户是否已登录。
- `Sys.Services.ProfileService`: 这个服务可以获取和设置当前登录的用户的配置属性。配置属性在应用程序的 `Web.config` 文件中配置。
- `Sys.Services.RoleService`: 这个服务可以确定当前登录的用户的角色成员。

如果使用正确, 这些类可以实现响应非常好的用户界面, 其中包含身份验证、配置经成员功能。

这些服务超出了本章的范围, 但应知道它们, 它们很值得研究。

## 39.4 小结

本章介绍了如何使用 ASP.NET AJAX 增强 ASP.NET Web 应用程序。ASP.NET AJAX 包含的丰富功能使 Web 应用程序的响应更好、更动态, 大大改进了用户体验。

首先学习了 Ajax 的概念、ASP.NET AJAX 的各个组件及其功能, 了解了 AJAX 扩展和 AJAX 库的区别, 这些组件联合起来提供 ASP.NET AJAX 核心功能的方式。还探讨了 AJAX Control Toolkit 和 ASP.NET 2.0 AJAX Futures CTP, 这些都添加到这个核心功能中。

接着, 论述了创建支持 ASP.NET AJAX 的 Web 应用程序的服务器端技术, 如何在 ASP.NET Web 应用程序的 `web.config` 文件中配置 ASP.NET AJAX, 如何使用 AJAX 扩展中的各种服务器控件, 特别是学习了 `ScriptManager`、`UpdatePanel`(和触发器)、`UpdateProgress` 和扩展器控件, 使用这些控件能快速、方便地给 Web 应用程序添加许多功能。

之后研究 AJAX 库。AJAX 库扩展并增强了 JavaScript, 提供了许多可以添加到应用程序中

的功能,但至少要了解 JavaScript 编程的基本知识。

我们学习了 AJAX 库给 JavaScript 添加的全局函数,如何使用 AJAX 库给 JavaScript 添加的 OOP 扩展,来定义命名空间和类。之后,研究了如何在页面的生存期和部分页面回送的过程中与客户端的事件交互,如何使用其中一个事件 `PageRequestManager.endRequest`,定制在部分页面回送的过程中发生的服务器错误在 Web 浏览器上的显示方式。

最后,陈述了客户端 Web 方法的调用,如何给这些方法调用使用异步模式,如何编写需要的代码,以调用简单的 Web 方法。还学习了通过 Web 服务访问 ASP.NET 应用程序服务(身份验证、配置和成员)的方式。

希望本章能使读者对这个新技术感兴趣。Ajax 在 Web 上非常流行,ASP.NET AJAX 是将 Ajax 功能与 ASP.NET 应用程序集成起来的绝佳方式。这个产品也得到了非常好的支持,基于团体的版本,如 AJAX Control Toolkit,提供了更酷的功能,这些功能可以在应用程序中免费使用。

尽管必须学习 JavaScript 语言,但这是值得的。使用 ASP.NET AJAX 比仅使用 ASP.NET 可以使 Web 应用程序更好、功能更强、更动态。在 VS 的最新版本中,有一些使 ASP.NET AJAX 更容易使用的工具。

下一章介绍 Web 开发,学习如何使用 VS 中的代码扩展 Microsoft Office 应用程序,例如 Word、Excel 和 Outlook。

# 第 40 章

## Visual Studio Tools for Office

Visual Studio Tools for Office(VSTO)技术可以使用.NET Framework 定制和扩展 Microsoft Office 应用程序和文档,它包含的工具还可以使这个定制在 Visual Studio 中更容易完成,例如用于 Office ribbon 控件的可视化设计器。

VSTO 是微软公司发布的一系列产品中的最新产品,可以定制 Office 应用程序。用于访问 Office 应用程序的对象模型已经随时间逐步演化了。如果读者过去曾使用过它,就会熟悉它的某些部分。如果读者以前为 Office 应用程序编写过 VBA 插件,就为本章讨论的技术做好了准备(VSTO 可以与 VBA 交互操作)。但 VSTO 通过 Office Primary Interop Assemblies(PIAs)提供的、与 Office 交互的类已经扩展到 Office 对象模型之外。例如,VSTO 类包括.NET 数据绑定功能。

在 Visual Studio 2008 推出之前,VSTO 一直是一个独立下载的软件包,如果要开发 Office 解决方案,就可以得到它。在 Visual Studio 2008 中,VSTO 集成到 Visual Studio IDE 中。VSTO 的这个版本也称为 VSTO 3,包含了对 Office 2007 的全部支持,还包括许多新特性,例如可以与 Word 内容控件交互,前面提及的 ribbon 可视化设计器、VBA 集成等。

本章不需要 VSTO 或其以前版本的任何预备知识。内容如下:

- 可以用 VSTO 创建的项目类型,在这些项目中可以包含的功能
- 应用于所有 VSTO 解决方案类型的基础技术
- 如何建立带定制 UI、VBA 交互操作功能和 ClickOnce 部署功能的 VSTO 解决方案

### 40.1 VSTO 概述

VSTO 包含如下组件:

- 一组项目模板,可用于创建各种类型的 Office 解决方案
- 设计器,支持 ribbons、动作面板和定制任务面板的可视化布局
- 建立在 Office Primary Interop Assemblies(PIAs)基础之上的类,它们还提供了扩展功能

VSTO 支持 Office 2003 和 2007 版。VSTO 类库有两种形式,各用于这两种 Office 版本,它们分别使用不同系列的程序集。由于它们比较简单(且功能集很丰富),所以本章主要介绍 2007 版。

VSTO 解决方案的一般体系结构如图 40-1 所示。



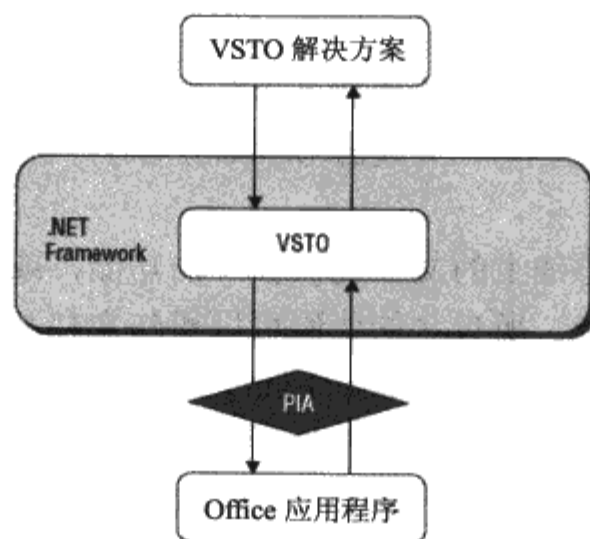


图 40-1

40.1.1 项目类型

图 40-2 显示了 Visual Studio 中的项目模板。

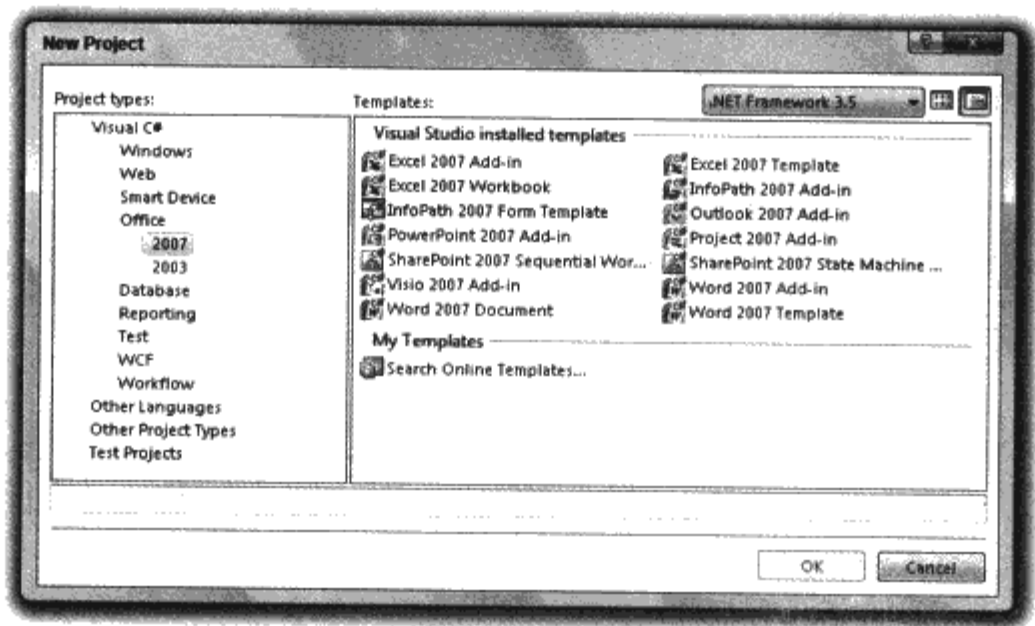


图 40-2

提示：

使用 VSTO 模板创建项目时，需要具备对 VBA 项目系统的访问权限。这是与 VBA 交互所必须的。

VSTO 项目模板可以分为如下类别：

- 文档级的定制
- 应用程序级的插件
- SharePoint 工作流模板
- InfoPath 窗体模板

一些项目类型有 2003 和 2007 版，但这里只介绍 2007 版。

本章主要讨论最常用的项目类型，即文档级的定制和应用程序级的插件。

## 1. 文档级的定制

创建这种类型的项目时，会生成一个链接到单个文档上的程序集，例如 Word 文档、Word 模板或 Excel 工作簿。加载该文档时，关联的 Office 应用程序会检测到定制，加载程序集，使 VSTO 定制可以使用。

这类项目可以给某个业务线上的文档提供额外的功能，或者在文档模板中添加定制功能，为这类文档添加额外功能。所包含的代码可以操作文档和文档的内容，包括内嵌的对象。还可以提供定制菜单，包括可以用 Visual Studio Ribbon 设计器创建的 ribbon 菜单。

创建文档级的项目时，可以选择创建新文档，或者复制已有的文档，作为开发的起点。也可以选择要创建的文档类型。例如，对于 Word 文档，就可以选择创建.docx(默认)、.doc 或.docm 文档(.docm 是支持宏的文档)。其对话框如图 40-3 所示。

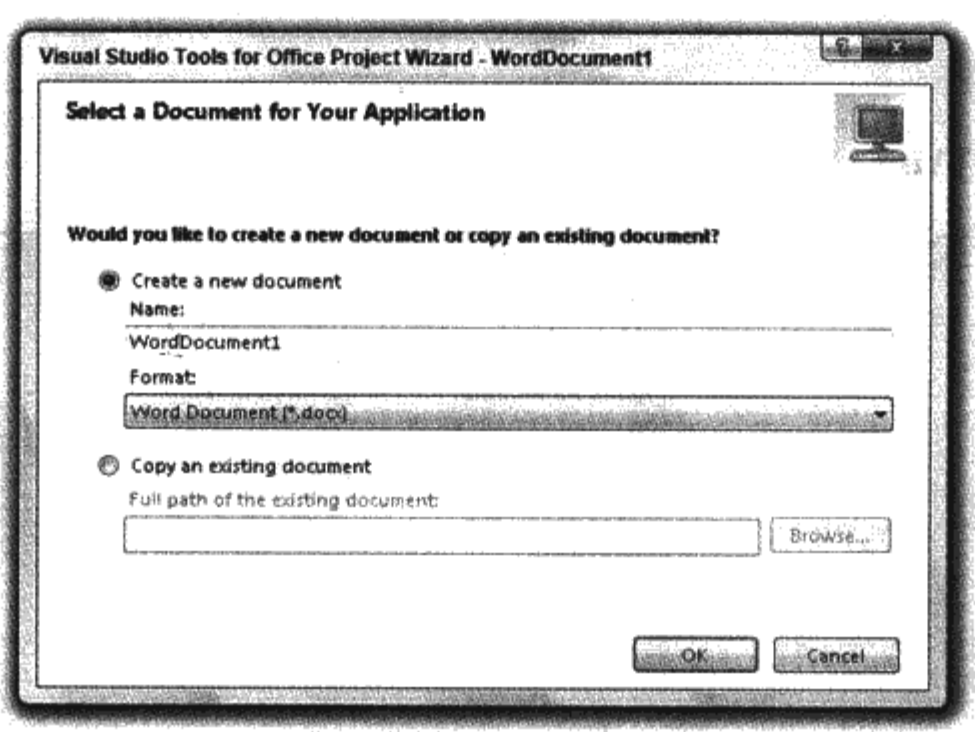


图 40-3

## 2. 应用程序级的插件

应用程序级的插件不同于文档级的定制，因为前者可用于整个目标 Office 应用程序。我们可以访问插件代码，其中可以包含菜单、文档操作等，而无论加载什么文档。

启动某个 Office 应用程序如 Word 时，它会寻找已在注册表中有数据项的关联插件，并加载需要的程序集。

## 3. SharePoint 工作流模板

这些项目提供了创建 SharePoint 工作流应用程序的模板。它们用于管理 SharePoint 进程中的文档流。创建了这类项目后，就可以在文档的生存期中，在重要的时刻执行定制代码。

## 4. InfoPath 窗体模板

这是用于 InfoPath 窗体的文档级定制的一种形式，但它们给 Word 和 Excel 文档定制使用略微不同的方法，所以通常要分为不同的类别。可以为 InfoPath 窗体创建模板，扩展 InfoPath

设计器的功能，为 InfoPath 窗体的设计人员和终端用户提供额外的功能和业务逻辑。  
创建 InfoPath 窗体模板时，可以利用向导指定要创建的项目类型，如图 40-4 所示。

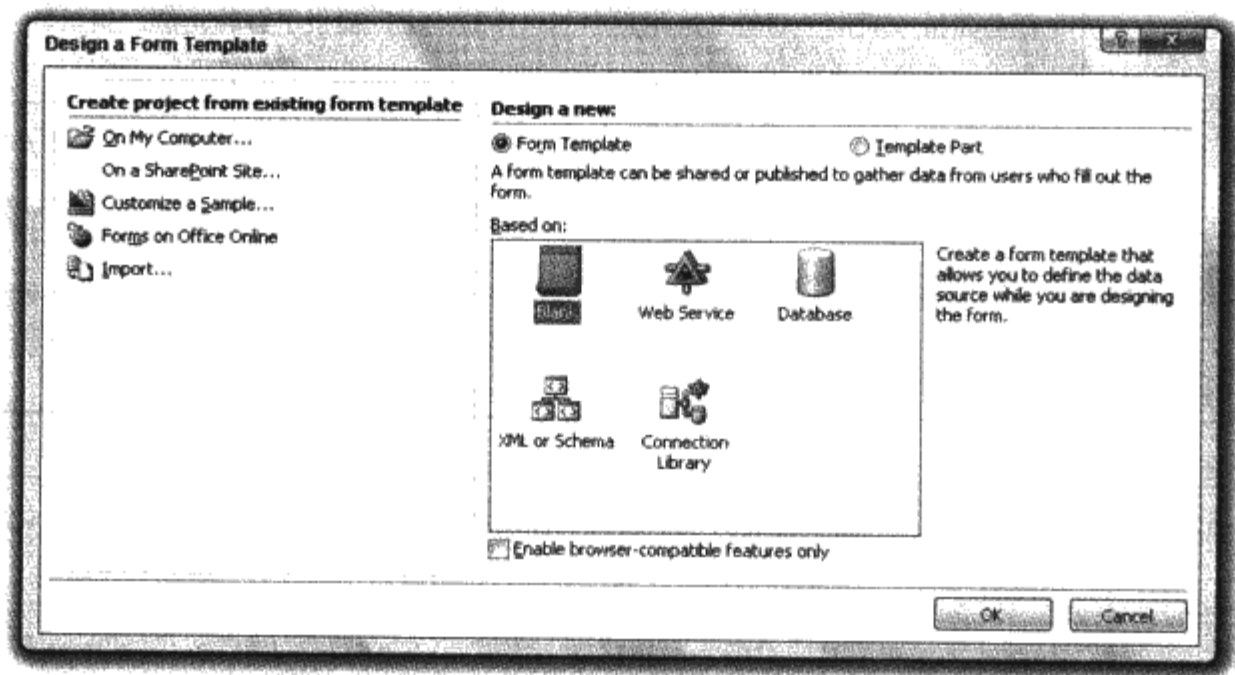


图 40-4

在图 40-4 中，这个向导为所创建的窗体提供了很大的灵活性：可以选择许多不同的起点(包括 SharePoint 站点上的窗体)。还可以创建完整的窗体或模板部分，把功能限制为与浏览器兼容的功能。

40.1.2 项目特性

在各种 VSTO 项目类型中有几个可以使用的特性，例如交互面板和控件。我们使用的项目类型决定了可用的特性。表 40-1 根据项目类型列出了这些特性。

表 40-1

特 性	说 明
动作面板	动作面板是保存在 Word 或 Excel 的动作面板中的对话框。可以在这里显示任意控件，这是扩展文档和应用程序的一种万能方式
数据高速缓存	数据的高速缓存可以在文档外部的高速缓存数据孤岛上存储在文档中使用的数据。这些数据孤岛可以从数据源中更新或手工更新，在数据源脱机或不可用时，允许 Office 文档访问数据
VBA 代码的端点	如前所述，VSTO 支持与 VBA 的交互操作。在文档级的定制中，可以提供从 VBA 代码中调用的端点方法
主机控件	主机控件是 Office 对象模型中已有控件的扩展封装器。可以操作这些对象，与它们建立数据绑定
智能标记	智能标记是嵌入在 Office 文档中、有类型化内容的对象。它们在 Office 文档的内容中自动检测，例如，应用程序检测到相应的文本时，就会自动添加股票报价智能标记。可以创建自己的智能标记类型，定义可以在该标记上执行的操作
可视化文档设计器	处理文档定制项目时，要使用 Office 对象模型创建一个可视化的设计界面，以交互式地布置控件。设计器中显示的工具栏和菜单(如本章后面所述)具有全面的功能

应用程序级的插件特性

特 性	说 明
定制任务面板	任务面板一般位于 Office 应用程序的一个边界上，提供了各种功能。例如，Word 的一个任务面板用于操作样式。与动作面板一样，它们也提供了很大的灵活性
跨应用程序的通信	为某个 Office 应用程序创建了插件后，就可以把这个功能提供给其他插件。例如可以在 Excel 中创建一个财务计算服务，再在 Word 中使用该服务——无需创建一个单独的插件
Outlook 窗体区域	可以创建在 Outlook 中使用的窗体区域

所有项目类型可用的特性

特 性	说 明
ClickOnce 部署	可以通过 ClickOnce 部署方法把自己创建的任意 VSTO 项目发布给终端用户，让用户检测对应用程序的程序集清单的变化，拥有文档级和应用程序级解决方案的最新版本
Ribbon 菜单	Ribbon 菜单在所有的 Office 应用程序中使用。VSTO 提供了创建定制 ribbon 菜单的两种方式，可以使用 XML 定义 ribbon，也可以使用 Ribbon 设计器，后者更容易使用，但采用 XML 版本可以保证向后兼容性

40.2 VSTO 基础

知道了 VSTO 包含的内容，下面该看看 VSTO 的特殊一面了，并学习如何建立 VSTO 项目。本节介绍的技术可以应用于所有的 VSTO 项目类型。

本节介绍如下内容：

- Office 对象模型
- VSTO 命名空间
- 主机项和主机控件
- 基本 VSTO 项目结构
- Globals 类
- 事件处理

40.2.1 Office 对象模型

Office 应用程序的 2007 套装通过一个 COM 对象模型提供其功能。可以在 VBA 中直接使用这个对象模型，来控制 Office 功能的任意方面。Office 对象模型在 Office 97 中引入，之后有了许多演变，Office 中的功能也有许多改变。

Office 对象模型有数量巨大的类，其中一些类在 Office 应用程序的套装中使用，一些类专门用于某些应用程序。例如，Word 2007 对象模型包含 Documents 集合，它表示当前加载的对象，每个对象都用一个 Document 对象表示。在 VBA 代码中，可以根据名称或索引访问文档，调用方法对它们执行操作。例如，下面的 VBA 代码关闭了名称为 My Document 的文档，且不

保存修改的内容：

```
Documents("My Document").Close SaveChanges:=wdDoNotSaveChanges
```

Office 对象模型包含命名的常量(例如上面代码中的 wdDoNotSaveChanges)和枚举，更便于使用。

40.2.2 VSTO 命名空间

VSTO 包含一个命名空间集合，该集合包含的类型可用于给 Office 对象模型编写程序。这些命名空间中的许多类和枚举直接映射到 Office 对象模型中的对象和枚举上。它们可以通过 Office PIAs 访问。VSTO 还包含不能直接映射的类型，或者与 Office 对象模型无关的类型。例如，有许多类用于 Visual Studio 中支持的设计器。

封装了 Office 对象模型中的对象或与它们通信的类型分别放在不同的命名空间中，这些命名空间包含了用于 Office 2003 和 2007 的类型。用于 Office 2007 开发的命名空间如表 40-2 所示。

表 40-2

命名空间	说明
Microsoft.Office.Core Microsoft.Office.Interop.*	这些命名空间包含 PIA 类的瘦封装器，所以提供了处理 Office 类的基本功能。 在 Microsoft.Office.Interop 命名空间中有几个嵌套的命名空间，用于每个 Office 产品
Microsoft.Office.Tools	这个命名空间包含的基本类型提供了 VSTO 功能和用于嵌套命名空间中的许多类的基类。例如，这个命名空间包含了实现文档级定制中的动作面板所需的类，以及应用程序级插件的基类
Microsoft.Office.Tools.Excel Microsoft.Office.Tools.Excel.*	这些命名空间包含的类型用于与 Excel 应用程序和 Excel 文档交互
Microsoft.Office.Tools.Outlook	这个命名空间包含的类型用于与 Outlook 应用程序交互
Microsoft.Office.Tools.Ribbon	这个命名空间包含的类型用于处理和创建 Ribbon 菜单
Microsoft.Office.Tools.Word Microsoft.Office.Tools.Word.*	这些命名空间包含的类型用于与 Word 应用程序和 Word 文档交互
Microsoft.VisualStudio.Tools.*	这些命名空间提供的 VSTO 基础体系可以在 Visual Studio 中开发 VSTO 解决方案时使用

40.2.3 主机项和主机控件

主机项和主机控件是扩展文档级定制的类，使之更容易与 Office 文档交互。这些类简化了代码，因为它们提供了 .NET 样式的事件，且进行了全面的管理。主机项和主机控件中的“主机”表示，这些类封装和扩展了通过 PIAs 访问的内部 Office 对象。

在使用主机项和主机控件时，常常需要使用底层的 PIA 交互操作类型。例如，如果创建了一个新的 Word 文档，就会接收到对交互操作 Word 文档类型的引用，而不是 Word 文档主机项。必须注意这一点，并据此编写代码。



Word 和 Excel 文档级定制都有主机项和主机控件。

1. Word

Word 只有一个主机项 Microsoft.Office.Tools.Word.Document。这表示一个 Word 文档。这个类有许多方法和属性，可用于与 Word 文档交互。

Word 有 12 个主机控件，如表 40-3 所示，所有主机控件都在 Microsoft.Office.Tools.Word 命名空间中。

表 40-3

控 件	说 明
Bookmark	这个控件表示 Word 文档中的一个位置，它可以是单个位置，或一个字符范围
XmlNode, XmlNodes	文档有一个关联的 XML 模式时使用这两个控件，它们允许通过文档内容的 XML 节点位置来引用文档内容。也可以用这两个控件操作文档的 XML 结构
ContentControl	这个类是本表中剩余 8 个控件的基类，允许处理 Word 内容控件。内容控件把内容表示为控件，或者启动文档中纯文本没有的功能
BuildingBlockGallery-ContentControl	这个控件允许添加和处理文档构造块，例如格式化的表、封面等
ComboBoxContentControl	这个控件表示格式化为组合框的内容
DatePickerContentControl	这个控件表示格式化为日期提取器的内容
DropDownListContentControl	这个控件表示格式化为下拉列表的内容
GroupContentControl	这个控件表示的内容是其他内容项的组合集合，包括文本和其他内容控件
PictureContentControl	这个控件表示一个图像
RickTextContentControl	这个控件表示一大块文本内容
PlainTextContentControl	这个控件表示一大块纯文本内容

2. Excel

Excel 有 3 个主机项和 4 个主机控件，它们都包含在 Microsoft.Office.Tools.Excel 命名空间中。

Excel 主机项如表 40-4 所示。

表 40-4

主 机 项	说 明
Workbook	这个主机项表示整个 Excel 工作簿，它可以包含多个工作表和图表
Worksheet	这个主机项用于工作簿中的单个工作表
Chartsheet	这个主机项用于工作簿中的单个图表

Excel 主机控件如表 40-5 所示。

表 40-5

控 件	说 明
Chart	这个控件表示嵌入到工作表中的图表
ListObject	这个控件表示工作表中的一个列表
NamedRange	这个控件表示工作表中的一个命名区域
XmlMappedRange	这个控件在 Excel 电子表格有关联的模式时使用，用于处理映射到 XML 模式元素上的范围

40.2.4 基本的 VSTO 项目结构

第一次创建 VSTO 项目时，系统创建的文件随项目类型的不同而不同，但有一些共同的特性。本节介绍 VSTO 项目的组成。

1. 文档级定制的项目结构

创建文档级定制的项目时，在 Solution Explorer 中有一项表示文档类型。它可以是：

- 表示 Word 文档的.docx 文件
- 表示 Word 模板的.dotx 文件
- 表示 Excel 工作簿的.xlsx 文件
- 表示 Excel 模板的.xltx 文件

每个文档类型都有一个设计器视图和一个代码文件，如果在 Solution Explorer 中展开该项，就会看到它们。Excel 模板还包含子项，它们表示整个工作簿和工作簿中的每个工作表。这个结构可以在每个工作表或工作簿的基础上提供定制功能。

如果查看上述项目类型的隐藏文件，会看到几个设计器文件，查看这些设计器文件，还会看到模板生成的代码。每个 Office 文档项都在 VSTO 命名空间中有关联的类，代码文件中的类派生于这些类。这些类定义为部分类，这样定制代码会与可视化设计器生成的代码分隔开，类似于 Windows 窗体应用程序的结构。

例如，Word 文档模板提供了一个派生自主机项 Microsoft.Office.Tools.Word.Document 的类，其代码包含在 ThisDocument.cs 中，如下所示：

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Xml.Linq;
using Microsoft.VisualStudio.Tools.Applications.Runtime;
using Office = Microsoft.Office.Core;
using Word = Microsoft.Office.Interop.Word;
namespace WordDocument1
{
    public partial class ThisDocument
    {
        private void ThisDocument_Startup(object sender, System.EventArgs e)
```

```

    {
    }
    private void ThisDocument_Shutdown(object sender, System.EventArgs e)
    {
    }
    #region VSTO Designer generated code
    /// < summary >
    /// Required method for Designer support - do not modify
    /// the contents of this method with the code editor.
    /// < /summary >
    private void InternalStartup()
    {
        this.Startup += new System.EventHandler(ThisDocument_Startup);
        this.Shutdown += new System.EventHandler(ThisDocument_Shutdown);
    }
    #endregion
}
}

```

这些模板生成的代码包含两个主要命名空间的别名，在为 Word 创建文档级的定制时，需要使用这两个命名空间。Microsoft.Office.Core 用于主要的 VSTO Office 类，Microsoft.Office.Interop.Word 用于和 Word 相关的类。注意如果要使用 Word 主机控件，还要为 Microsoft.Office.Tools.Word 命名空间添加一个 using 语句。模板生成的代码还定义了两个事件处理程序 ThisDocument\_Startup() 和 ThisDocument\_Shutdown()，用于在加载和卸载文档时执行代码。

每个文档级定制项目类型的代码文件都有类似的结构，还定义了命名空间别名以及 VSTO 类中各个 Startup 和 Shutdown 事件的处理程序。以此为起点，可以添加对话框、动作面板、ribbon 控件、事件处理程序和定制代码，来定义定制操作。

在文档级的定制中，还可以通过文档设计器定制文档。根据所创建的解决方案类型，这可能需要给模板添加样板文件，给文档添加交互式内容或其他内容。设计器是 Office 应用程序的高效主机版本，使用它们可以像在应用程序中那样输入内容。还可以在文档中添加控件，例如主机控件和 Windows 窗体控件，以及这些控件的代码。

## 2. 应用程序级插件的项目结构

创建应用程序级插件时，在 Solution Explorer 中没有文档，而有一项表示创建插件所使用的应用程序。如果展开该项，会看到一个文件 ThisAddIn.cs。这个文件包含类 ThisAddIn 的部分定义，该类是插件的入口点。这个类派生于 Microsoft.Office.Tools.AddIn，它提供了编写插件的功能，实现了 Microsoft.VisualStudio.Tools.Office.IOfficeEntryPoint 接口，这是一个基础体系接口。

例如，Word 插件模板生成的代码如下：

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Xml.Linq;
using Word = Microsoft.Office.Interop.Word;
using Office = Microsoft.Office.Core;
namespace WordAddIn1

```

```

{
    public partial class ThisAddIn
    {
        private void ThisAddIn_Startup(object sender, System.EventArgs e)
        {
        }
        private void ThisAddIn_Shutdown(object sender, System.EventArgs e)
        {
        }
        #region VSTO generated code
        /// < summary >
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// < /summary >
        private void InternalStartup()
        {
            this.Startup += new System.EventHandler(ThisAddIn_Startup);
            this.Shutdown += new System.EventHandler(ThisAddIn_Shutdown);
        }
        #endregion
    }
}

```

可以看出，这个结构非常类似于文档级定制使用的结构，它包含 `Microsoft.Office.Core` 和 `Microsoft.Office.Interop.Word` 命名空间的别名，提供了 `Startup` 和 `Shutdown` 事件的处理程序 (`ThisAddIn_Startup()` 和 `ThisAddIn_Shutdown()`)。这些事件与文档级定制略有不同，因为它们是在加载或卸载插件时触发，而不是在打开或关闭文档时触发。

定制应用程序级插件与文档级定制相同：添加 ribbon 控件、任务面板和其他代码。

#### 40.2.5 Globals 类

所有的 VSTO 项目类型都定义了 `Globals` 类，它提供了如下内容的全局访问权限：

- 对于文档级的定制，可以访问解决方案中的所有文档，这是通过其名称匹配文档类名的成员来实现的，例如 `Globals.ThisWorkbook` 和 `Globals.Sheet1`。
- 对于应用程序级的插件，可以访问插件对象。这是通过 `Globals.ThisAddIn` 实现的。
- 对于 Outlook 插件项目，可以访问所有的 Outlook 窗体区域。
- 可以访问解决方案中的所有 ribbon 控件，这是通过 `Globals.Ribbons` 属性定义的。

在后台，`Globals` 类是在解决方案的由各种设计器维护的代码文件中通过一系列部分定义来创建的。例如，在 Excel 工作簿项目中，默认的 `Sheet1` 工作表包含如下设计器生成的代码：

```

internal sealed partial class Globals
{
    private static Sheet1 _Sheet1;
    internal static Sheet1 Sheet1
    {
        get
        {
            return _Sheet1;
        }
        set
        {
            if ((_Sheet1 == null))

```

```
        {  
            _Sheet1 = value;  
        }  
        else  
        {  
            throw new System.NotSupportedException();  
        }  
    }  
}
```

这段代码把 Sheet1 成员添加到 Globals 类中。

#### 40.2.6 事件处理

本章前面介绍了主机项和主机控件类如何提供我们可以处理的事件。但交互操作类不是这样。我们只能使用几个事件，大多数事件都很难用于创建事件驱动的解决方案。为了响应事件，我们常常要关注主机项和主机控件提供的事件。

此处一个明显的问题是应用程序级的插件项目没有主机项和主机控件。在使用 VSTO 时，必须面对这个问题。但是，我们在插件中监听的大多数常用事件都关联到 ribbon 菜单和任务面板的交互操作中。我们用集成的 ribbon 设计器设计 ribbon 控件，响应 ribbon 控件生成的事件，使控件可以交互操作。任务面板常常实现为 Windows 窗体用户控件(也可以使用 WPF)，所以这里可以使用 Windows 窗体事件，来代替 PIA 交互操作事件。这表示，我们不会常常遇到如下情形：需要的功能没有可用的事件。

需要使用 PIA 提供的事件时，这些事件是通过 PIA 对象上的接口提供的。考虑一个 Word 插件项目。这个项目中的 ThisAddIn 类有一个属性 Application，利用它可以获得对 Office 应用程序的引用。这个属性的类型是 Microsoft.Office.Interop.Word.Application，它通过 Microsoft.Office.Interop.Word.ApplicationEvents4\_Event 接口提供事件。这个接口共提供了 29 个事件(对于像 Word 这样复杂的应用程序来说并不多)。我们可以处理 DocumentBeforeClose 事件，来响应 Word 文档的关闭请求。

### 40.3 建立 VSTO 解决方案

前面几节解释了 VSTO 项目的概念、结构和可以在各种项目类型中使用的特性。本节讨论如何实现 VSTO 解决方案。

图 40-5 列出了文档级定制解决方案的结构。

对于文档级的定制，至少要与一个主机项交互操作，该主机项一般包含几个主机控件。可以直接使用 Office 对象封装器，但在大多数情况下，应通过主机项和主机控件访问 Office 对象模型及其功能。



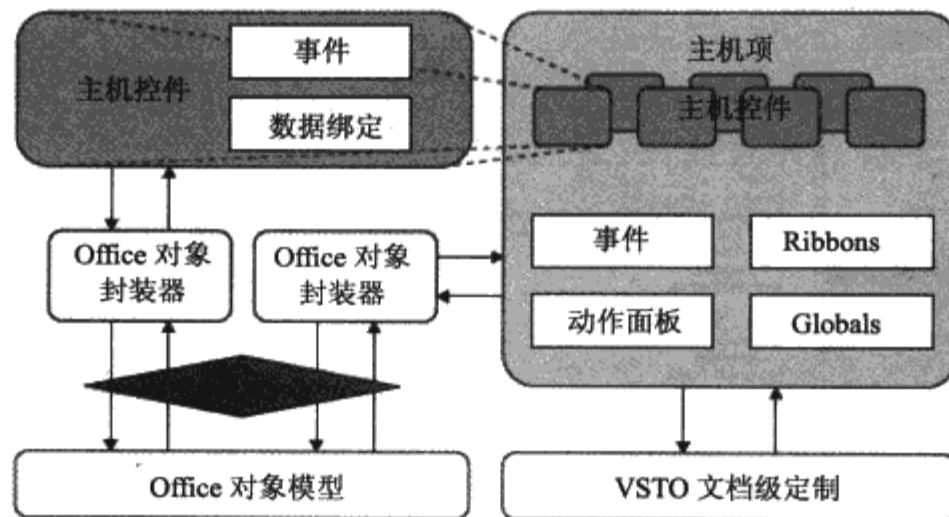


图 40-5

可以在代码中使用主机项和主机控件事件、数据绑定、ribbon 菜单、动作面板和全局对象。图 40-6 列出了应用程序级插件解决方案的结构。

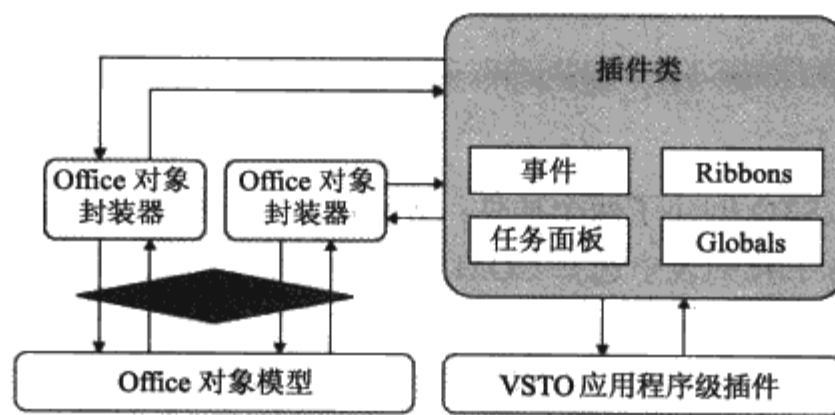


图 40-6

在这个略微简单的模型中，很可能要直接使用 Office 对象的瘦封装器，或者至少通过封装解决方案的插件类来使用。还要在代码中使用插件类的事件、ribbon 菜单、动作面板和全局对象。

本节介绍这两种应用程序类型以及如下主题：

- 管理应用程序级插件
- 与应用程序和文档交互操作
- UI 的定制

### 40.3.1 管理应用程序级插件

在创建应用程序级插件时，会发现 Visual Studio 执行了注册 Office 应用程序的插件所需的所有步骤。这表示，添加了注册表项，在 Office 应用程序启动时，会自动定位和加载程序集。如果以后要添加或删除插件，就必须浏览 Office 应用程序设置，或者手工操作注册表。

例如，在 Word 中，必须打开 Office Button 菜单，单击 Word Options，选择 Add-Ins 选项卡，如图 40-7 所示。

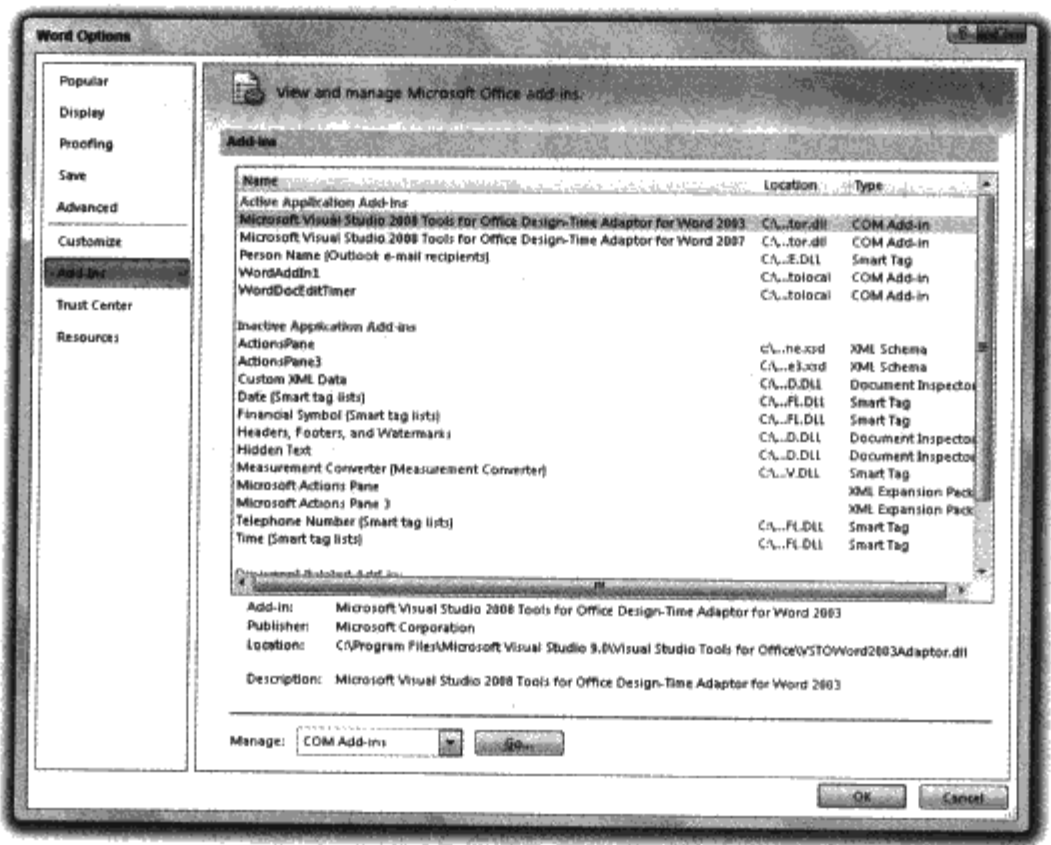


图 40-7

图 40-7 显示，用 VSTO 创建了两个插件：WordAddIn1 和 WordDocEditTimer。要添加或删除插件，必须在 Manage 下拉列表中选择 COM Add-Ins(默认选项)，单击 Go 按钮，打开如图 40-8 所示的对话框。

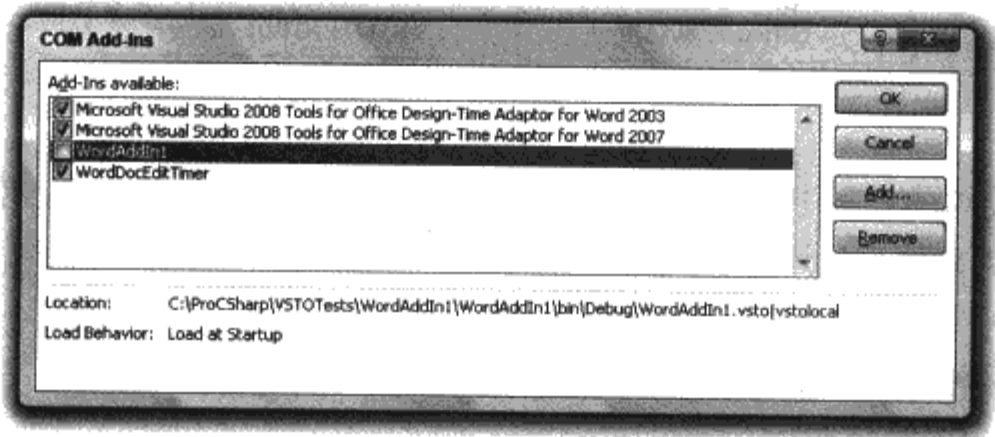


图 40-8

在 COM Add-Ins 对话框中取消对插件的选择，就会卸载插件，如图 40-8 所示。还可以使用 Add 和 Remove 按钮添加新插件或删除旧插件。

40.3.2 与应用程序和文档交互操作

无论创建什么类型的应用程序，都要与主机应用程序和/或主机应用程序中的文档交互操作。这包括使用下一节介绍的 UI 定制功能。还可能需

- DocumentOpen: 打开文档时触发
- NewDocument: 创建新文档时触发
- DocumentBeforeClose: 保存文档时触发

另外，Word 第一次启动时，会加载一个文档，它可以是空白的新文档，也可以是已加载的旧文档。

#### 提示：

本章的下载代码包含一个示例 WordDocEditTimer，它维护着 Word 文档的一个编辑时次数表。这个应用程序的部分功能是监控已加载的文档，其原因在后面解释。这个示例也使用了定制的任务面板和 ribbon 菜单，所以在介绍完这些主题后，再介绍这个示例。

在 Word 中，可以通过 `ThisAddIn.Application.ActivateDocument` 属性访问当前活动的文档，通过 `ThisAddIn.Application.Documents` 属性访问打开的文档集合。由于有了 Multiple Document Interface(MDI)，类似的属性也存在于其他 Office 应用程序中。可以通过 `Microsoft.Office.Interop.Word.Document` 类的属性操作文档的各个属性。

这里要注意，在开发 VSTO 解决方案时，必须处理的类和类成员的数量是相当大的。除非已经习惯了，否则很难找到需要的特性。例如，在 Word 中，当前活动的选择不是通过活动的文档获得的，而是通过应用程序获得的(利用 `ThisAddIn.Application.Selection` 属性)。其原因并不是很明显。

通过 `Range` 属性可以把选择应用于插入、读取或替换文本的操作。例如：

```
ThisAddIn.Application.Selection.Range.Text = "Inserted text";
```

但是，本章没有足够的篇幅来详细介绍对象库，读者可以在本章讨论到相关的内容时学习对象库。

### 40.3.3 UI 的定制

在 VSTO 的最新版本中，最重要的方面是定制 UI 的功能和插件的灵活性。可以给已有的 ribbon 菜单中添加内容，添加全新的 ribbon 菜单，定制动作面板，添加全新的动作面板，集成 Windows 窗体、WPF 窗体和控件。

本节介绍这些主题。

#### 1. Ribbon 菜单

可以在本章介绍的所有 VSTO 项目中添加 ribbon 菜单。添加 ribbon 菜单时，会看到如图 40-9 所示的设计器窗口。

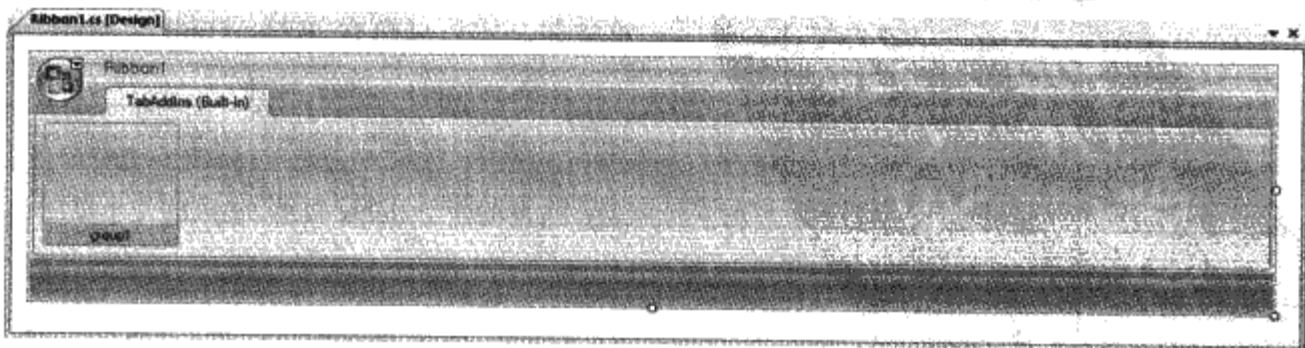


图 40-9

设计器可以给 Office 按钮菜单和 ribbon 菜单上的组添加控件(显示在图 40-9 的左上部)，来

定制这个 ribbon 菜单。也可以添加其他组。

ribbon 中使用的类在 Microsoft.Office.Tools.Ribbon 命名空间中。这包括用于创建 ribbon 的派生类 OfficeRibbon。这个类可以包含 RibbonTab 对象，每个 RibbonTab 对象都包含了单个选项卡的内容。选项卡则包含了 RibbonGroup 对象，例如图 40-9 中的 group1 组。这些选项卡都可以包含各种控件。

选项卡上的组可以位于目标 Office 应用程序的一个全新选项卡上，或者位于一个已有的选项卡上。组在何处显示取决于 RibbonTab.ControlId 属性。这个属性有一个 ControlIdType 属性，它可以设置为 RibbonControlIdType.Custom 或 RibbonControlIdType.Office。如果使用 Custom，还必须把 RibbonTab.ControlId.CustomId 设置为 String 值，这是选项卡的标识符。这里可以使用任意标识符。但如果给 ControlIdType 使用 Office，就必须把 RibbonTab.ControlId.OfficeId 设置为一个 String 值，该值匹配在当前 Office 产品中使用的标识符。例如，在 Excel 中，可以把这个属性设置为 TabHome，把组添加到 Home 选项卡上，设置为 TabInsert 把组添加到 Insert 选项卡上，等。插件的默认属性是 TabAddIns，它由所有的插件共享。

提示：

可以使用许多选项卡，尤其在 Outlook 中；可以从 [www.microsoft.com/downloads/details.aspx?FamilyID=4329D9E9-4D11-46A5-898D-23E4F331E9AE&displaylang=en](http://www.microsoft.com/downloads/details.aspx?FamilyID=4329D9E9-4D11-46A5-898D-23E4F331E9AE&displaylang=en) 中下载包含完整列表的一系列电子表格。

决定了在何处放置 ribbon 组后，就可以添加如表 40-6 所示的控件了。

表 40-6

控 件	说 明
RibbonBox	这是一个容器控件，可用于布置组中的其他控件。可以把 BoxStyle 属性改为 RibbonBoxStyle.Horizontal 或 RibbonBoxStyle.Vertical，在 RibbonBox 中水平或垂直布置控件
RibbonButton	这个控件可用于在组中添加大按钮或小按钮，在按钮的旁边可以有或没有文本标签。把 ControlSize 属性设置为 RibbonControlSize. RibbonControlSizeLarge 或 RibbonControlSize. RibbonControlSizeRegular，就可以控制大小。按钮的 Click 事件处理程序可用于响应交互操作。还可以设置定制图像或存储在 Office 系统中的图像(详见本表后面的内容)
RibbonButtonGroup	这是一个容器控件，表示一组按钮。它可以包含 RibbonButton、RibbonGallery、RibbonMenu、RibbonSplitButton 和 RibbonToggleButton 控件
RibbonCheckBox	复选框控件，有 Click 事件和 Checked 属性
RibbonComboBox	组合框(合并了文本项和下拉列表)。列表项使用 Items 属性，输入的文本使用 Text 属性，TextChanged 事件用于响应交互操作
RibbonDropDown	这个容器可以包含 RibbonDropDownItem 和 RibbonButton 项，它们分别在 Items 和 Buttons 属性中指定。按钮和列表项格式化到下拉列表中。使用 SelectionChanged 事件响应交互操作

(续表)

控 件	说 明
RibbonEditBox	文本框，用户可用于输入或编辑 Text 属性中的文本。这个控件有 TextChanged 事件
RibbonGallery	与 RibbonDropDown 相同，这个控件也可以包含 RibbonDropDownItem 和 RibbonButton 项，它们分别在 Items 和 Buttons 属性中指定。这个控件使用 Click 和 ButtonClick 事件，来替代 RibbonDropDown 控件的 SelectionChanged 事件
RibbonLabel	显示简单的文本，用 Label 属性设置
RibbonMenu	弹出菜单，在设计视图中打开时，可以用其他控件填充该菜单，例如 RibbonButton 和嵌套的 RibbonMenu 控件。处理菜单上的菜单项的事件
RibbonSeparator	一个简单的分隔符，用于定制组中的控件布局
RibbonSplitButton	合并了 RibbonButton 或 RibbonToggleButton 和 RibbonMenu 的控件。用 ButtonType 设置按钮的样式，该属性可以是 RibbonButtonType.Button 或 RibbonButtonType.ToggleButton。使用主按钮的 Click 事件或菜单中各按钮的 Click 事件来响应交互操作
RibbonToggleButton	一个按钮，可以处于选中或未选中状态，用 Checked 属性指定。这个控件也有 Click 事件

也可以设置组的 DialogBoxLauncher 属性，把一个图标显示在组的右下部。使用这个属性可以显示一个对话框，或者打开一个任务面板，或者执行其他操作。通过 GroupView Tasks 菜单可以添加或删除这个图标，如图 40-10 所示，该图还显示了表 40-5 中的其他一些控件，因为它们在设计视图中显示在 ribbon 上。

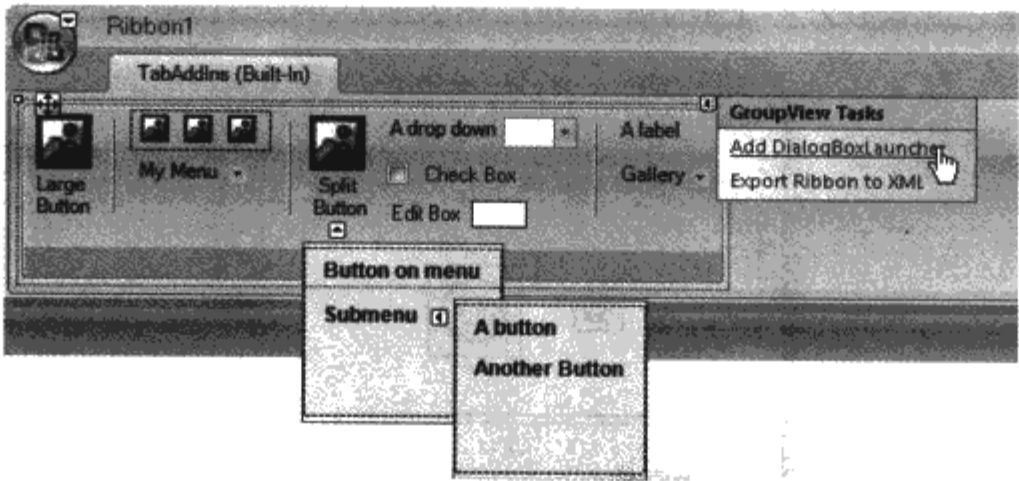


图 40-10

要给控件设置图像，例如给 RibbonButton 控件设置图像，就可以把 Image 属性设置为定制图像，ImageName 设置为图像名(以便在 OfficeRibbon.LoadImage 事件处理程序中优化图像的加载)，也可以使用内置的 Office 图像。为此，应把 OfficeImageId 属性设置为图像的 ID。

可以使用许多图像；还可以从 [www.microsoft.com/downloads/details.aspx?familyid=12b99325-93e8-4ed4-8385-74d0f7661318&displaylang=en](http://www.microsoft.com/downloads/details.aspx?familyid=12b99325-93e8-4ed4-8385-74d0f7661318&displaylang=en) 上下载包含这些图像的电子表格。图 40-11 显示了一个示例。



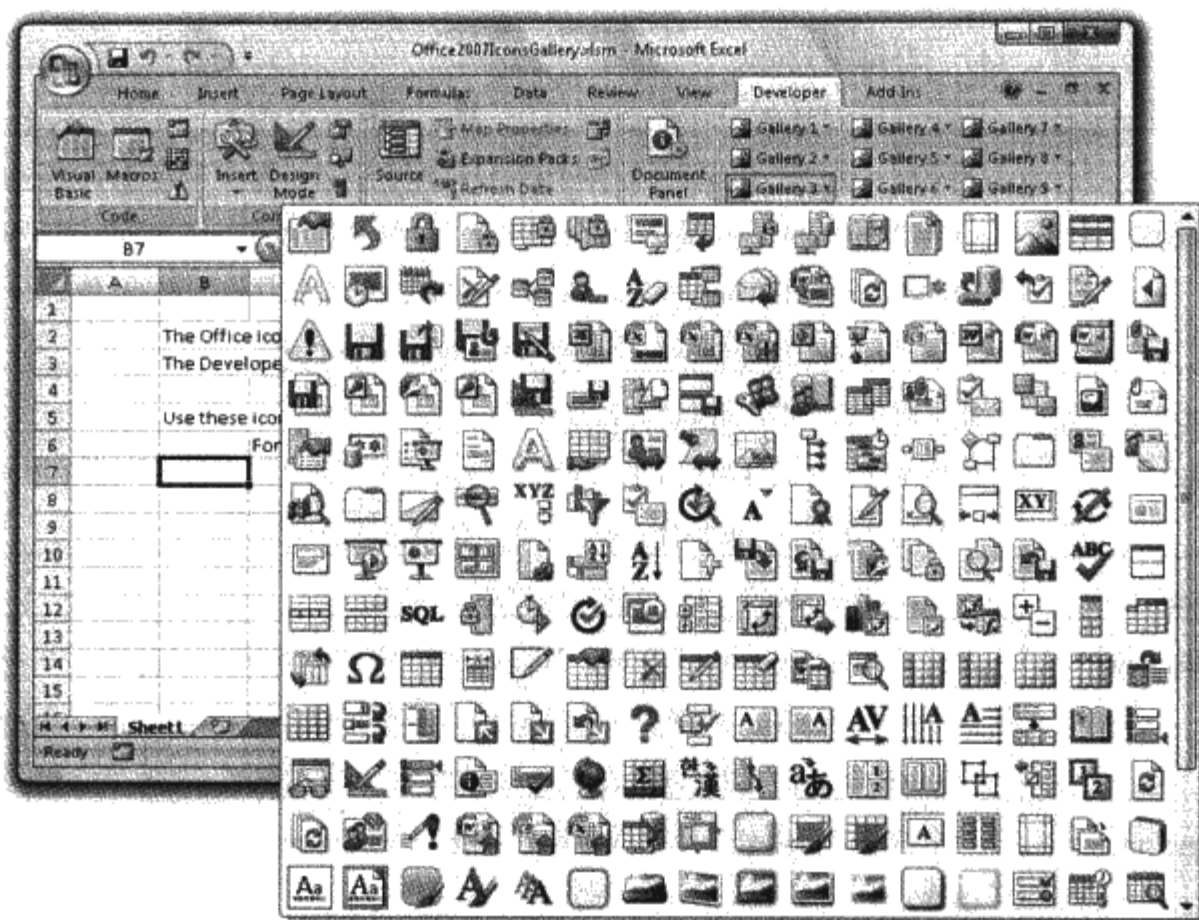


图 40-11

提示:

图 40-11 显示了 Developer ribbon 选项卡, 通过 Popular 选项卡上的 Excel Options 对话框中的 Office 按钮可以打开它。

单击一个图像, 就会打开一个对话框, 指出该图像的 ID 是什么, 如图 40-12 所示。



图 40-12

ribbon 设计器非常灵活, 还可以提供希望出现在 Office ribbon 上的许多额外功能。但是, 如果要进一步定制 UI, 就要使用动作和任务面板, 因为可以通过它们创建任意 UI 和功能。

2. 动作面板和定制的任务面板

使用动作和任务面板可以显示停放在 Office 应用程序界面的任务面板区域中的内容。任务面板在应用程序级的插件中使用, 动作面板在文档级的定制中使用。任务和动作面板都必须继

承自 `UserControl` 对象, 这表示应使用 Windows 窗体创建一个 UI。如果把 WPF 窗体保存在 `UserControl` 的 `ElementHost` 控件上, 还可以使用 WPF UI。这些控件的一个区别是可以通过 `New Item Wizard` 中的 `Action Pane Template`, 或使用简单的用户控件, 把动作面板添加到文档级的定制上。任务面板必须添加为一般的用户控件。

要把动作面板添加到文档级定制中的一个文档上, 应把动作面板类的一个实例添加到文档的 `ActionsPane` 属性的 `Controls` 集合中。例如:

```
public partial class ThisWorkbook
{
    Private ActionsPaneControl actionsPane;
    private void ThisWorkbook_Startup(object sender, System.EventArgs e)
    {
        actionsPane = new ActionsPaneControl();
        this.ActionsPane.Controls.Add(actionsPane);
    }
    ...
}
```

这段代码在加载文档(这里是 Excel 工作簿)时添加了动作面板。也可以在 `ribbon` 按钮事件处理程序中添加动作面板。

在应用程序级插件项目中, 定制的任务面板通过 `ThisAddIns.CustomTaskPanes.Add()` 方法属性添加。这个方法也允许命名任务窗口, 例如:

```
public partial class ThisAddIn
{
    Microsoft.Office.Tools.CustomTaskPane taskPane;
    private void ThisAddIn_Startup(object sender, System.EventArgs e)
    {
        taskPane = this.CustomTaskPanes.Add(new UserControl(),
            "My Task Pane");
        taskPane.Visible = true;
    }
    ...
}
```

注意 `Add()` 方法返回一个 `Microsoft.Office.Tools.CustomTaskPane` 类型的对象。可以通过这个对象的 `Control` 属性访问用户控件本身。还可以使用这个类型的其他属性, 例如上面代码中的 `Visible` 属性, 来控制任务面板。

此时, 应注意 Office 应用程序的一个不太寻常的特性, 尤其是 Word 和 Excel 之间的区别。由于历史的原因, 尽管 Word 和 Excel 都是 MDI 应用程序, 但这两个应用程序存储文档的方式是不同的。在 Word 中, 每个文档都有一个唯一的父窗口, 而在 Excel 中, 每个文档都共享同一个父窗口。

在调用 `CustomTaskPanes.Add()` 方法时, 默认操作是把任务面板添加到当前活动的窗口中。在 Excel 中, 这表示每个文档都显示该任务面板。因为它们都使用同一个父窗口。而在 Word 中, 情况就不同了。如果希望任务面板显示给每个文档, 就必须把它添加到包含文档的每个窗口中。

要把任务面板添加到特定的文档中, 应给 `Add()` 方法传送 `Microsoft.Office.Interop.Word.Windows` 类的一个实例, 作为第三个参数。通过 `Microsoft.Office.Interop.Word`

Document.ActiveWindow 属性可以获得关联了文档的窗口。

下一节介绍如何完成这个操作。

## 40.4 示例应用程序

如前所述，本章的示例代码包含一个应用程序 WordDocEditTimer，它维护着 Word 文档的一个编辑次数列表。本节将详细解释这个应用程序的代码，因为该应用程序演示了前面介绍的所有内容，还包含一些有益的提示。

这个应用程序的一般操作是只要创建或加载了文档，就启动一个链接到文档名称上的计时器。如果关闭文档，该文档的计时器就暂停。如果打开了以前计时的文档，计时器就恢复。另外，如果使用 Save As 把文档保存为另一个文件名，计时器就更新为使用新文件名。

这个应用程序是一个 Word 应用程序级的插件，使用一个定制任务面板和一个 ribbon 菜单。ribbon 菜单包含一个按钮和一个复选框，按钮用于开关任务面板，复选框用于暂停当前活动的文档的计时器。包含这些控件的组添加到 Home ribbon 选项卡的最后。任务面板显示一组活动的计时器。

这个用户界面如图 40-13 所示。

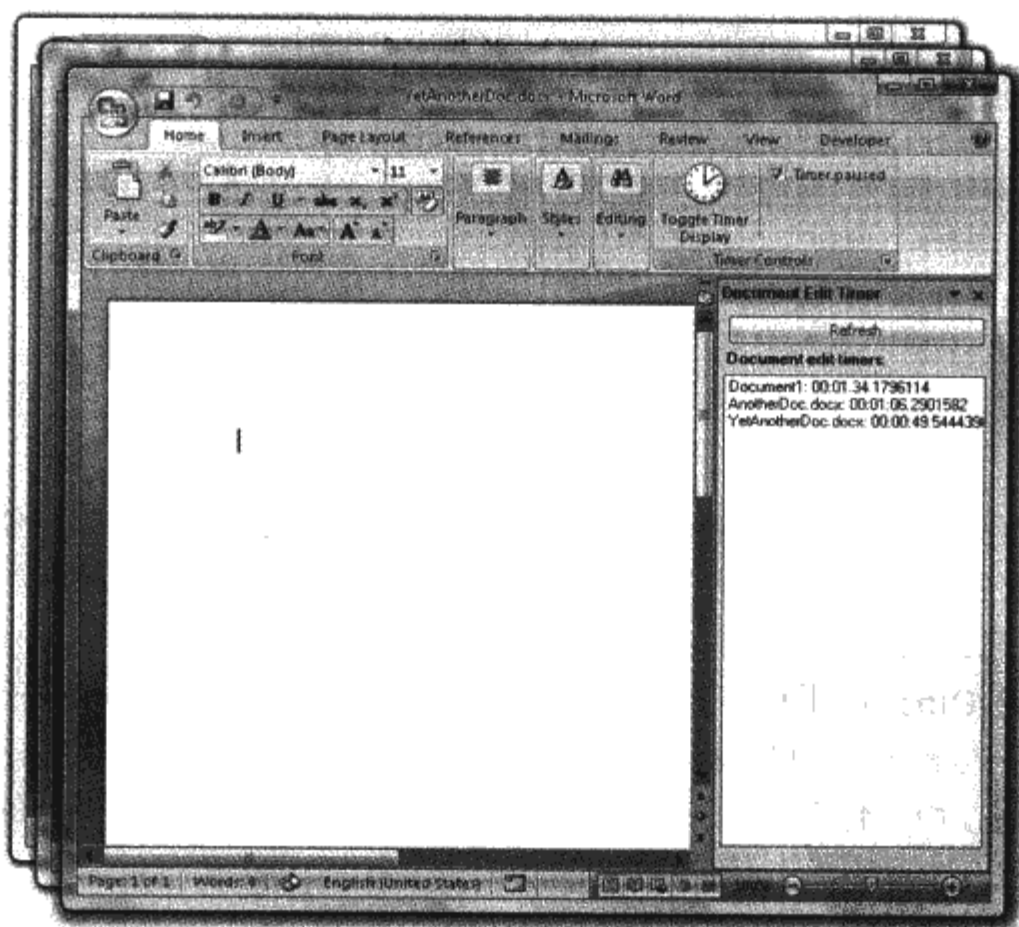


图 40-13

计时器通过 DocumentTimer 类来维护：

```
public class DocumentTimer
{
    public Word.Document Document { get; set; }
```

```

    public DateTime LastActive { get; set; }
    public bool IsActive { get; set; }
    public TimeSpan EditTime { get; set; }
}

```

这段代码保存了对 `Microsoft.Office.Interop.Word.Document` 对象的一个引用、总编辑时间、计时器是否激活，以及它上一次激活的时间。`ThisAddIn` 类维护这些对象的一个集合，这些对象与文档名关联起来：

```

public partial class ThisAddIn
{
    private Dictionary < string, DocumentTimer > documentEditTimes;

```

因此，每个计时器都可以通过文档引用或文档名来定位。这是必要的，因为文档引用可以跟踪文档名的变化(这里没有可用于监控文档名变化的事件)，文档名允许跟踪关闭、再次打开的文档。

`ThisAddIn` 类还维护一个 `CustomTaskPane` 对象列表(如前所述，Word 中的每个窗口都需要一个 `CustomTaskPane` 对象)：

```

private List < Tools.CustomTaskPane > timerDisplayPanels;

```

插件启动时，`ThisAddIn.Startup()` 方法执行了几个任务。首先它初始化两个集合：

```

private void ThisAddIn_Startup(object sender, System.EventArgs e)
{
    // Initialize timers and display panels
    documentEditTimes = new Dictionary < string, DocumentTimer > ();
    timerDisplayPanels = new
        List < Microsoft.Office.Tools.CustomTaskPane > ();

```

接着通过 `ApplicationEvents4_Event` 接口添加几个事件处理程序：

```

// Add event handlers
Word.ApplicationEvents4_Event eventInterface = this.Application;
eventInterface.DocumentOpen += new Microsoft.Office.Interop.Word
    .ApplicationEvents4_DocumentOpenEventHandler(
        eventInterface_DocumentOpen);
eventInterface.NewDocument += new Microsoft.Office.Interop.Word
    .ApplicationEvents4_NewDocumentEventHandler(
        eventInterface_NewDocument);
eventInterface.DocumentBeforeClose += new Microsoft.Office.Interop.Word
    .ApplicationEvents4_DocumentBeforeCloseEventHandler(
        eventInterface_DocumentBeforeClose);
eventInterface.WindowActivate += new Microsoft.Office.Interop.Word
    .ApplicationEvents4_WindowActivateEventHandler(
        eventInterface_WindowActivate);

```

这些事件处理程序用于监控文档的打开、创建和关闭，并确保 ribbon 上的 **Pause** 复选框保持最新状态。后一个功能是使用 `WindowsActivate` 事件跟踪窗口的激活状态来实现的。

在这个事件处理程序中，最后一个任务是开始监控当前文档，把定制的任务面板添加到包含文档的窗口中：

```

// Start monitoring active document
MonitorDocument(this.Application.ActiveDocument);
AddTaskPaneToWindow(this.Application.ActiveDocument.ActiveWindow);

```

}

**MonitorDocument()**实用方法为文档添加一个计时器:

```
internal void MonitorDocument(Word.Document Doc)
{
    // Monitor doc
    documentEditTimes.Add(Doc.Name, new DocumentTimer
    {
        Document = Doc,
        EditTime = new TimeSpan(0),
        IsActive = true,
        LastActive = DateTime.Now
    });
}
```

这个方法仅为文档创建了一个新的 **DocumentTimer** 对象。**DocumentTimer** 引用文档, 其编辑次数是 0, 且是在当前时间激活的。接着把这个计时器添加到 **documentEditTimes** 集合中, 并关联到文档名中。

**AddTaskPaneToWindow()**方法把定制任务面板添加到窗口中。这个方法首先检查已有的任务面板, 确保窗口中还没有任务面板。**Word** 中的另一个古怪的特性是如果在加载应用程序后, 立即打开一个旧文档, 默认的 **Document1** 文档就会消失, 且不触发关闭事件。在访问包含任务面板的文档窗口时, 这可能导致异常, 所以该方法还检查表示是否出现该异常的 **ArgumentNullException**:

```
private void AddTaskPaneToWindow(Word.Window Wn)
{
    // Check for task pane in window
    Tools.CustomTaskPane docPane = null;
    Tools.CustomTaskPane paneToRemove = null;
    foreach (Tools.CustomTaskPane pane in timerDisplayPanes)
    {
        try
        {
            if (pane.Window == Wn)
            {
                docPane = pane;
                break;
            }
        }
        catch (ArgumentNullException)
        {
            // pane.Window is null, so document1 has been unloaded.
            paneToRemove = pane;
        }
    }
}
```

如果抛出了一个异常, 就从集合中删除错误的任务面板:

```
// Remove pane if necessary
timerDisplayPanes.Remove(paneToRemove);
```

如果窗口中没有任务面板, 这个方法就添加一个:

```
// Add task pane to doc
if (docPane == null)
```



```

{
    Tools.CustomTaskPane pane = this.CustomTaskPanes.Add(
        new TimerDisplayPane(documentEditTimes),
        "Document Edit Timer",
        Wn);
    timerDisplayPanes.Add(pane);
    pane.VisibleChanged +=
        new EventHandler(timerDisplayPane_VisibleChanged);
}
}

```

添加的任务面板是 `TimerDisplayPane` 类的一个实例。稍后介绍这个类。它添加时使用的名称是 `Document Edit Timer`。另外，在调用 `CustomTaskPanes.Add()` 方法后，还为得到的 `CustomTaskPane` 的 `VisibleChanged` 事件添加了一个处理程序，这样在第一次显示任务面板时，可以刷新显示：

```

private void timerDisplayPane_VisibleChanged(object sender, EventArgs e)
{
    // Get task pane and toggle visibility
    Tools.CustomTaskPane taskPane = (Tools.CustomTaskPane)sender;
    if (taskPane.Visible)
    {
        TimerDisplayPane timerControl = (TimerDisplayPane)taskPane.Control;
        timerControl.RefreshDisplay();
    }
}

```

`TimerDisplayPane` 类有一个 `RefreshDisplay()` 方法，它在上面的代码中调用。这个方法刷新 `timerControl` 对象的显示。

接着的代码确保监控所有的文档。首先创建新文档时，调用 `eventInterface_New-Document()` 事件处理程序，调用 `MonitorDocument()` 和前面介绍过的 `AddTaskPaneTo-Window()` 方法监控文档。

```

private void eventInterface_NewDocument(Word.Document Doc)
{
    // Monitor new doc
    MonitorDocument(Doc);
    AddTaskPaneToWindow(Doc.ActiveWindow);
}

```

新文档在计时器运行时启动，此时这个方法还清除了 ribbon 菜单中的 `Pause` 复选框。这是通过一个实用方法 `SetPauseStatus()` 实现的，该方法在 ribbon 中定义：

```

// Set checkbox
Globals.Ribbons.TimerRibbon.SetPauseStatus(false);
}

```

在关闭文档之前，调用 `eventInterface_DocumentBeforeClose()` 事件处理程序。这个方法冻结了文档的计时器，更新了总编辑时间，清除了 `Document` 引用，删除了文档窗口中的任务面板(使用稍后介绍的 `RemoveTaskPaneFromWindow()` 方法)，之后关闭窗口。

```

private void eventInterface_DocumentBeforeClose(Word.Document Doc,
    ref bool Cancel)
{
    // Freeze timer

```

```

        documentEditTimes[Doc.Name].EditTime += DateTime.Now
        - documentEditTimes[Doc.Name].LastActive;
        documentEditTimes[Doc.Name].IsActive = false;
        documentEditTimes[Doc.Name].Document = null;
        // Remove task pane
        RemoveTaskPaneFromWindow(Doc.ActiveWindow);
    }

```

打开文档时，调用 `eventInterface_DocumentOpen()` 方法。该方法完成了许多工作，因为在监控文档之前，这个方法必须查看计时器的名称，确定文档是否已有计时器：

```

private void eventInterface_DocumentOpen(Word.Document Doc)
{
    if (documentEditTimes.ContainsKey(Doc.Name))
    {
        // Monitor old doc
        documentEditTimes[Doc.Name].LastActive = DateTime.Now;
        documentEditTimes[Doc.Name].IsActive = true;
        documentEditTimes[Doc.Name].Document = Doc;
        AddTaskPaneToWindow(Doc.ActiveWindow);
    }
}

```

如果还没有监控文档，就为文档配置一个新监控器：

```

else
{
    // Monitor new doc
    MonitorDocument(Doc);
    AddTaskPaneToWindow(Doc.ActiveWindow);
}
}

```

`RemoveTaskPaneFromWindow()` 方法用于从窗口中删除任务面板。其代码首先检查特定的窗口中是否有任务面板：

```

private void RemoveTaskPaneFromWindow(Word.Window Wn)
{
    // Check for task pane in window
    Tools.CustomTaskPane docPane = null;
    foreach (Tools.CustomTaskPane pane in timerDisplayPanes)
    {
        if (pane.Window == Wn)
        {
            docPane = pane;
            break;
        }
    }
}

```

如果找到了任务面板，就调用 `CustomTaskPanes.Remove()` 方法删除它。还要从任务面板引用的本地集合中删除它。

```

// Remove document task pane
if (docPane != null)
{
    this.CustomTaskPanes.Remove(docPane);
    timerDisplayPanes.Remove(docPane);
}
}

```

这个类中的最后一个事件处理程序是 `eventInterface_WindowActivate()`，在激活窗口时调用它。这个方法获得活动文档的计时器，选中 ribbon 菜单中的复选框，以更新文档的复选框：

```
private void eventInterface_WindowActivate(Word.Document Doc,
    Word.Window Wn)
{
    // Ensure pause checkbox in ribbon is accurate, start by getting timer
    DocumentTimer documentTimer =
        documentEditTimes[this.Application.ActiveDocument.Name];
    // Set checkbox
    Globals.Ribbons.TimerRibbon.SetPauseStatus(!documentTimer.IsActive);
}
```

`ThisAddIn` 的代码还包含两个实用方法。第一个方法 `ToggleTaskPaneDisplay()` 用于设置 `CustomTaskPanes.Visible` 属性，为当前活动的文档显示或隐藏任务面板。

```
internal void ToggleTaskPaneDisplay()
{
    // Ensure window has task window
    AddTaskPaneToWindow(this.Application.ActiveDocument.ActiveWindow);
    // toggle document task pane
    Tools.CustomTaskPane docPane = null;
    foreach (Tools.CustomTaskPane pane in timerDisplayPanes)
    {
        if (pane.Window == this.Application.ActiveDocument.ActiveWindow)
        {
            docPane = pane;
            break;
        }
    }
    docPane.Visible = !docPane.Visible;
}
```

上述代码中的 `ToggleTaskPaneDisplay()` 方法由 ribbon 控件上的事件处理程序调用，如后面所述。

最后，该类有另一个从 ribbon 菜单中调用的方法，它允许 ribbon 控件暂停或恢复文档的计时器：

```
internal void PauseOrResumeTimer(bool pause)
{
    // Get timer
    DocumentTimer documentTimer =
        documentEditTimes[this.Application.ActiveDocument.Name];
    if (pause & & documentTimer.IsActive)
    {
        // Freeze timer
        documentTimer.EditTime += DateTime.Now - documentTimer.LastActive;
        documentTimer.IsActive = false;
    }
    else if (!pause & & !documentTimer.IsActive)
    {
        // Resume timer
        documentTimer.IsActive = true;
        documentTimer.LastActive = DateTime.Now;
    }
}
```

这个类定义中的其他代码是 Shutdown 的空事件处理程序以及 VSTO 为关联 Startup 和 Shutdown 事件处理程序而生成的代码。

接着布置项目中的 ribbon，即 TimerRibbon，如图 40-14 所示。

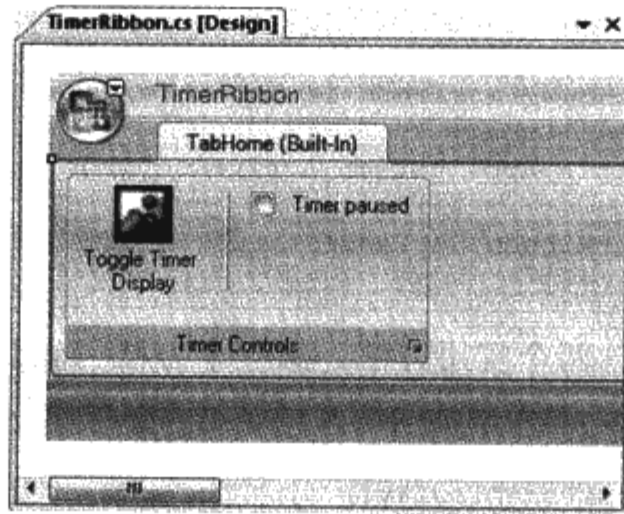


图 40-14

这个 ribbon 包含一个 RibbonButton、一个 RibbonSeparator、一个 RibbonCheckBox 和一个 DialogBoxLauncher。按钮使用大显示样式，其 OfficeImageId 设置为 StartAfterPrevious，显示如图 40-13 所示的钟表图像。(这些图像在设计期间不可见)。ribbon 使用 TabHome 选项卡类型，其内容追加到 Home 选项卡上。

ribbon 有 3 个事件处理程序，每个处理程序都调用前面介绍的 ThisAddIn 中的一个实用方法：

```
private void group1_DialogLauncherClick(object sender,
    RibbonControlEventArgs e)
{
    // Show or hide task pane
    Globals.ThisAddIn.ToggleTaskPaneDisplay();
}
private void pauseCheckBox_Click(object sender, RibbonControlEventArgs e)
{
    // Pause timer
    Globals.ThisAddIn.PauseOrResumeTimer(pauseCheckBox.Checked);
}
private void toggleDisplayButton_Click(object sender,
    RibbonControlEventArgs e)
{
    // Show or hide task pane
    Globals.ThisAddIn.ToggleTaskPaneDisplay();
}
```

ribbon 还包含自己的实用方法 SetPauseStatus()，如前所述，该方法由 ThisAddIn 中的代码调用，以选中复选框或取消复选框的选中。

```
internal void SetPauseStatus(bool isPaused)
{
    // Ensure checkbox is accurate
    pauseCheckBox.Checked = isPaused;
}
```

这个解决方案中的另一个组件是任务面板中使用的 TimerDisplayPane 用户控件，这个控件

的布局如图 40-15 所示。

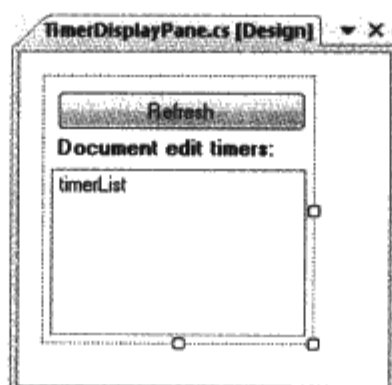


图 40-15

这个控件包含一个按钮、一个标签和一个列表框——这些都是很普通的显示控件，也可以用更漂亮的 WPF 控件替代它们。

该控件的代码保存了对文档计时器的一个本地引用，该引用在构造函数中设置：

```
public partial class TimerDisplayPane : UserControl
{
    private Dictionary < string, DocumentTimer > documentEditTimes;
    public TimerDisplayPane()
    {
        InitializeComponent();
    }
    public TimerDisplayPane(Dictionary < string, DocumentTimer >
        documentEditTimes) : this()
    {
        // Store reference to edit times
        this.documentEditTimes = documentEditTimes;
    }
}
```

按钮事件处理程序调用 RefreshDisplay() 方法刷新计时器的显示：

```
private void refreshButton_Click(object sender, EventArgs e)
{
    RefreshDisplay();
}
```

RefreshDisplay() 方法也从 ThisAddIn 中调用，如前所述。考虑到该方法的任务，这是一个相当复杂的方法，它还检查被监控文档的列表，与已加载文档的列表比较，并解决出现的问题。这段代码在 VSTO 应用程序中常常是必不可少的，因为 COM Office 对象模型的接口偶尔不能像期望的那样工作。这里的规则是防御式编码。

该方法首先清除 timerList 列表框中的当前计时器列表：

```
internal void RefreshDisplay()
{
    // Clear existing list
    this.timerList.Items.Clear();
}
```

接着检查监控器。这个方法迭代 Globals.ThisAddIn.Application.Documents 集合中的每个文档，确定文档是被监控、未被监控、或被监控了但在上次刷新时改变了文件名。

要找出被监控的文档，只需比较当前的文档名和键的 documentEditTimes 集合中的文档名：



```
// Ensure all docs are monitored
foreach (Word.Document doc in Globals.ThisAddIn.Application.Documents)
{
    bool isMonitored = false;
    bool requiresNameChange = false;
    DocumentTimer oldNameTimer = null;
    string oldName = null;
    foreach (string documentName in documentEditTimes.Keys)
    {
        if (doc.Name == documentName)
        {
            isMonitored = true;
            break;
        }
    }
}
```

如果文档名不匹配，就比较文档引用，以检测对文档名的修改，如下面的代码所示：

```
else
{
    if (documentEditTimes[documentName].Document == doc)
    {
        // Monitored, but name changed!
        oldName = documentName;
        oldNameTimer = documentEditTimes[documentName];
        isMonitored = true;
        requiresNameChange = true;
        break;
    }
}
```

对于未监控的文档，需要创建一个新的监控器：

```
// Add monitor if not monitored
if (!isMonitored)
{
    Globals.ThisAddIn.MonitorDocument(doc);
}
```

名称改变的文档需要通过用于旧文档的监控器重新关联起来：

```
// Rename if necessary
if (requiresNameChange)
{
    documentEditTimes.Remove(oldName);
    documentEditTimes.Add(doc.Name, oldNameTimer);
}
```

调整了文档编辑计时器后，生成一个列表。代码还会检测引用的文档是否加载了，对于没有加载的文档，把 IsActive 属性设置为 false，暂停该文档的计时器。这也是防御性编程方式：

```
// Create new list
foreach (string documentName in documentEditTimes.Keys)
{
    // Check to see if doc is still loaded
    bool isLoading = false;
    foreach (Word.Document doc in
        Globals.ThisAddIn.Application.Documents)
```

```

{
    if (doc.Name == documentName)
    {
        isLoading = true;
        break;
    }
}
if (!isLoading)
{
    documentEditTimes[documentName].IsActive = false;
    documentEditTimes[documentName].Document = null;
}

```

对于每个监控器，把一个列表项添加到列表框中，其中包含了文档名和总编辑时间：

```

// Add item
this.timerList.Items.Add(string.Format("{0}: {1}", documentName,
    documentEditTimes[documentName].EditTime +
    (documentEditTimes[documentName].IsActive ?
    (DateTime.Now - documentEditTimes[documentName].LastActive) :
    new TimeSpan(0))));
}
}

```

这就完成了这个例子中的代码。这个例子说明了如何使用 ribbon 和任务面板控件，如何维护多个 Word 文档中的任务面板，还演示了本章前面介绍的许多技术。

## 40.5 VBA 交互操作性

Office 系统已经推出了多年，所以读者很熟悉 VBA 代码，在已有的应用程序中还使用了 VBA。在 VSTO 解决方案中可以重写 VBA 代码，但这并不总是切合实际。看到 VSTO 的功能后，读者可能希望用托管的 VSTO 代码替代已有的 VBA 功能，或者添加新功能。

VSTO 允许给 VBA 代码提供 VSTO 功能，以实现上述任务。为此，必须执行几个步骤，才能给 VBA 代码提供 COM 接口。这 9 步如下所示，还列出了下载代码的 ExcelVBAInterop 项目中的示例代码和屏幕图：

(1) 在开始给 VBA 提供 VSTO 代码之前，必须有一个包含 VBA 项目的文档。为了便于开发，最好在开始之前启动文档中的宏。接着，在 VSTO 中创建一个文档级的定制时，把该文档作为自己解决方案中文档的起点。

(2) 有了这个起点后，就可以用通常的方式编写访问应用程序和/或文档的代码了。这些代码不能通过 VSTO 项目访问，因为后面要提供一个 VBA 接口。所以应创建 VBA 可以调用的方法，例如：

```

public partial class ThisWorkbook : ExcelVBAInterop.IThisWorkbook
{
    ...
    public void NameSheet()
    {
        NamingDialog dlg = new NamingDialog();
        if (dlg.ShowDialog() == DialogResult.OK)
        {

```

```
        ((Excel.Worksheet)this.ActiveSheet).Name = dlg.SheetName;
    }
}
```

提示:

这些代码使用一个简单的定制对话框,稍后介绍它。这个对话框允许用户输入一个字符串,选择是否在表名中包含当前日期。

(3) 必须重写 GetAutomationObject()方法,为 VBA 代码的自动执行返回正确的对象,如下所示:

```
public partial class ThisWorkbook : ExcelVBAInterop.IThisWorkbook
{
    ...
    protected override object GetAutomationObject()
    {
        return this;
    }
}
```

(4) 因为 COM 系统通过接口来工作,所以必须通过接口提供要调用的方法。最简单的方法是右击代码,选择 Refactor | Extract Interface,接着选择要在接口中提供的方法,向导会完成剩余的工作,如图 40-16 所示。

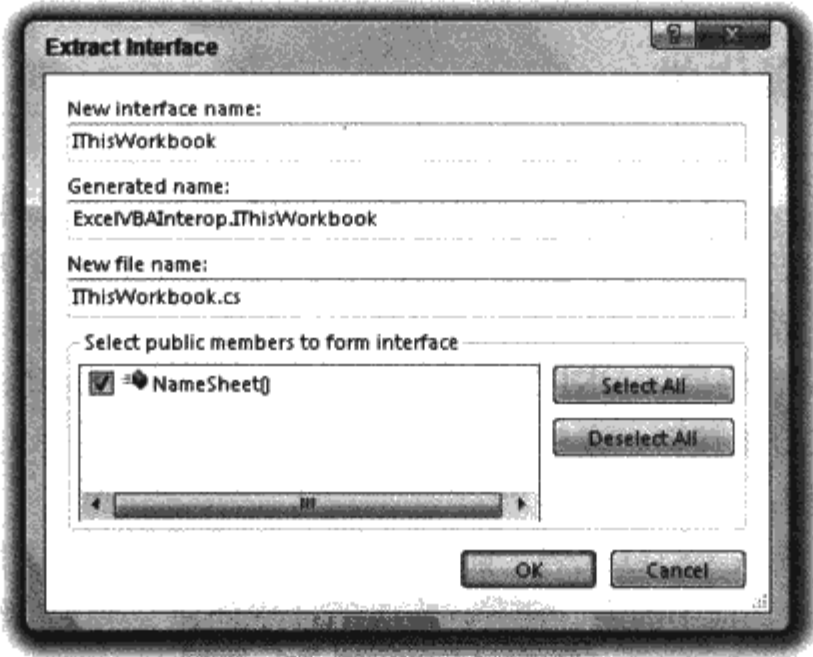


图 40-16

(5) 还必须把 System.Runtime.InteropServices 命名空间中的属性添加到类中,使类显示在 COM 上(参见第 24 章):

```
using System.Runtime.InteropServices;
namespace ExcelVBAInterop
{
    [ComVisible(true)]
    [ClassInterface(ClassInterfaceType.None)]
    public partial class ThisWorkbook : ExcelVBAInterop.IThisWorkbook
    {
```

```
...
}
}
```

(6) 生成的接口还需要 **ComVisible** 属性，该接口必须是公共的：

```
using System.Runtime.InteropServices;
namespace ExcelVBAInterop
{
    [ComVisible(true)]
    public interface IThisWorkbook
    {
        void NameSheet();
    }
}
```

(7) 把文档的 **ReferenceAssemblyFromVbaProject** 属性改为 **true**，如图 40-17 所示。如果文档不包含 VBA 代码，就不能修改这个属性。改变它时，会接收到一个警告，说明项目运行时，添加到项目中的 VBA 代码会丢失，所以应备份修改的 VBA 代码。

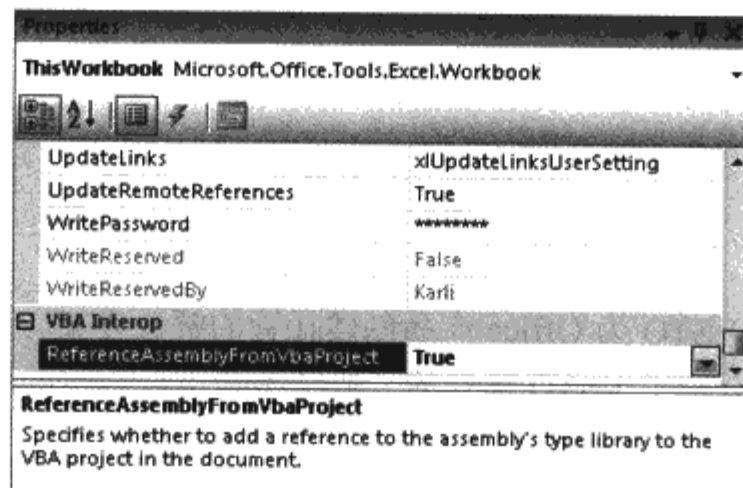


图 40-17

(8) 剩下的就是对文档中 VBA 代码的修改了。可以在此时运行项目，运行项目时可以通过 **Developer** 选项卡或按下 **Alt+F11**，来访问 VBA 代码。首先需要添加的是一个允许 VBA 访问 VSTO 代码的属性，如下所示(按添加了命名空间限制符的名称引用类)：

```
Property Get VSTOAssembly() As ExcelVBAInterop.ThisWorkbook
    Set VSTOAssembly = GetManagedClass(Me)
End Property
```

(9) 接着就可以通过这个属性调用 VSTO 方法了：

```
Public Sub RenameSheet()
    VSTOAssembly.NameSheet
End Sub
```

完成了这些步骤后，就可以添加代码，调用接口上的方法，或手工调用它，如图 40-18 所示。

如果代码包含一个 UI，如本例所示，就会显示该 UI，且可以使用它。示例项目中的 UI 如图 40-19 所示。



图 40-18

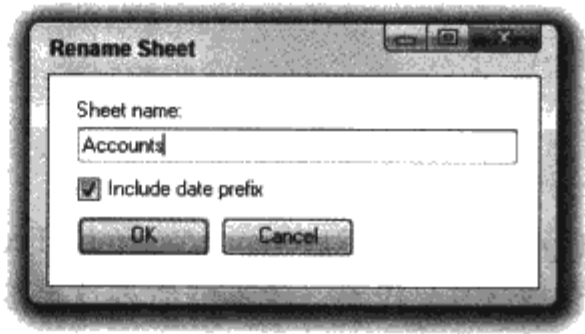


图 40-19

这样，就可以使 VSTO 代码用于 VBA 代码了。

40.6 小结

本章学习了如何使用 VSTO 为 Office 产品创建托管的解决方案。

在本章的第一部分，介绍了 VSTO 项目的一般结构和可以创建的项目类型，还探讨了便于 VSTO 编程的特性。

接下来详细论述了 VSTO 解决方案中的一些特性，讨论了如何与 Office 对象模型通信，介绍了 VSTO 中的命名空间和类型，陈述了如何使用这些类型实现各种功能。之后研究了 VSTO 项目的一些编码特性，以及如何使用这些特性获得希望的结果。

然后，进行了一些实践。我们学习了如何在 Office 应用程序中管理插件，如何与 Office 对象模型交互操作，如何用 ribbon 菜单、任务面板和动作面板定制应用程序的 UI。

接着开发了一个示例应用程序，演示了前面学习的 UI 和交互操作技术。这个示例包含许多代码，还包含有用的技巧，例如如何在多个 Word 文档窗口中管理任务面板。

最后介绍了与 VBA 代码的交互操作。这部分介绍了如何通过 COM 交互操作性把托管代码提供给 VBA，并用另一个示例演示了这些技术。

这是第 V 部分的最后一章。第 VI 部分的第一章将介绍如何在应用程序中使用 System.Net 命名空间中的类访问 Internet。



# 第VI部分

## 通 信

- 第 41 章 访问 Internet
- 第 42 章 Windows Communication Foundation
- 第 43 章 Windows Workflow Foundation
- 第 44 章 Enterprise Services
- 第 45 章 消息队列
- 第 46 章 目录服务
- 第 47 章 对等网络
- 第 48 章 Syndication

# 第 41 章

## 访问 Internet

第 37~39 章讨论了怎样使用 C#、ASP.NET 编写功能强、效率高的动态 Web 页。大多数情况下,访问 ASP.NET 页面的客户一般使用的是 Internet Explorer 或其他 Web 浏览器,如 Opera 或 FireFox。但是,有时需要把 Web 浏览特性添加到自己的应用程序中,或者需要让自己的应用程序从某个 Web 站点以编程方式获取信息,在后一种情况下,对于站点来说,比较好的解决方案通常是实现 Web 服务,但是,如果访问公共的 Internet 站点,就不能控制站点的执行方式。

本章将讨论通过 .NET 基类提供的工具,使用各种网络协议(尤其是 HTTP 和 TCP)访问网络和 Internet。本章将讨论的内容包括:

- 从 WWW 下载文件
- 在 Windows 窗体应用程序中使用 Web 浏览器控件
- 操纵 IP 地址,执行 DNS 查询
- 用 TCP、UDP 和套接字类进行套接字编程

本章将介绍通过 .NET Framework 获得执行协议的一些低级方式,使用下一章介绍的 WCF 技术还可以提供与这些协议通信的其他方式。

在网络环境下,我们最感兴趣的两个命名空间是 `System.Net` 和 `System.Net.Sockets`。`System.Net` 命名空间通常与较高层的操作有关,例如下载和上传文件,使用 HTTP 和其他协议进行 Web 请求等,而 `System.Net.Sockets` 命名空间包含的类通常与较低层的操作有关。如果要直接使用套接字(Socket)或 TCP/IP 之类的协议,这个命名空间中的类是非常有用的,这些类中的方法与 Windows 套接字(Winsock)API 函数(派生自 Berkeley 套接字)非常类似。本章介绍的一些对象在 `System.IO` 命名空间中。

本章将采取实用的方法,结合示例讨论相关理论和相应的网络概念。本章并不是计算机网络的指南,但介绍了如何使用 .NET Framework 进行网络通信。

我们还将介绍 Windows 窗体环境中新 Web 浏览器控件的使用,以及如何更方便地完成某些 Internet 访问任务。

首先,从最简单的示例开始,这个示例阐明了怎样给服务器发送请求和保存返回的信息(与其他章节一样,本章的示例代码也可以从 Wrox 网站 [www.wrox.com](http://www.wrox.com) 上下载)。

## 41.1 WebClient 类

如果只想从特定的 URI 请求文件, 则可以使用的最简单 .NET 基类就是 `System.Net.WebClient`。这个类是非常高层的类, 它只用一两个命令执行基本操作。 .NET Framework 目前支持以 `http:`、`https:`和 `file:`标识符开头的 URI。

注意:

术语 URL(统一资源定位符)在新的技术规范中已不再使用, 现在使用的是 URI(统一资源标识符)。URI 的含义大致与 URL 相同, 但 URI 更通用, 因为它不隐含正在使用的协议, 如 HTTP 或 FTP。

### 41.1.1 下载文件

使用 `WebClient` 类下载文件有两种方法, 具体使用哪一种方法取决于文件内容的处理方式。如果只想把文件保存到磁盘上, 就应该调用 `DownloadFile()`方法。这个方法有两个参数: 即文件的 URI 和保存所请求的数据的位置(路径和文件名):

```
WebClient Client = new WebClient();
Client.DownloadFile("http://www.reuters.com/", "ReutersHomepage.htm ");
```

更为常见的是, 应用程序需要处理从 Web 站点检索到的数据。为此, 要使用 `OpenRead()`方法, 这个方法返回一个 `Stream` 引用。然后, 就可以把数据从数据流中提取到内存中:

```
WebClient Client = new WebClient();
Stream strm = Client.OpenRead("http://www. reuters.com/ ");
```

### 41.1.2 基本的 Web 客户示例

第一个示例将阐述怎样使用 `WebClient.OpenRead()`方法。在这个示例中, 我们将把下载的页面显示在 `ListBox` 控件中。首先, 把项目创建为标准的 C# Windows 窗体应用程序, 添加一个名为 `listBox1` 的列表框, 将其 `docking` 属性设置为 `DockStyle.Fill`。在文件的开头, 需要在 `using` 指令中添加 `System.Net` 和 `System.IO` 命名空间引用, 然后对主窗体的构造函数进行以下改动:

```
public Form1()
{
    InitializeComponent();
    System.Net.WebClient Client = new WebClient();
    Stream strm = Client.OpenRead("http://www. reuters.com");
    StreamReader sr = new StreamReader(strm);
    string line;
    while ( (line=sr.ReadLine()) != null )
    {
        listBox1.Items.Add(line);
    }
    strm.Close();
}
```

在这个示例中，把 System.IO 命名空间的 StreamReader 类与网络数据流关联起来。这样，就可以使用高层方法，例如 ReadLine()方法，从数据流中以文本的形式获取数据。第 25 章讨论了把数据移动抽象化为数据流概念的优点，这个示例就充分体现出了这些优点。

这个示例的运行结果如图 41-1 所示。

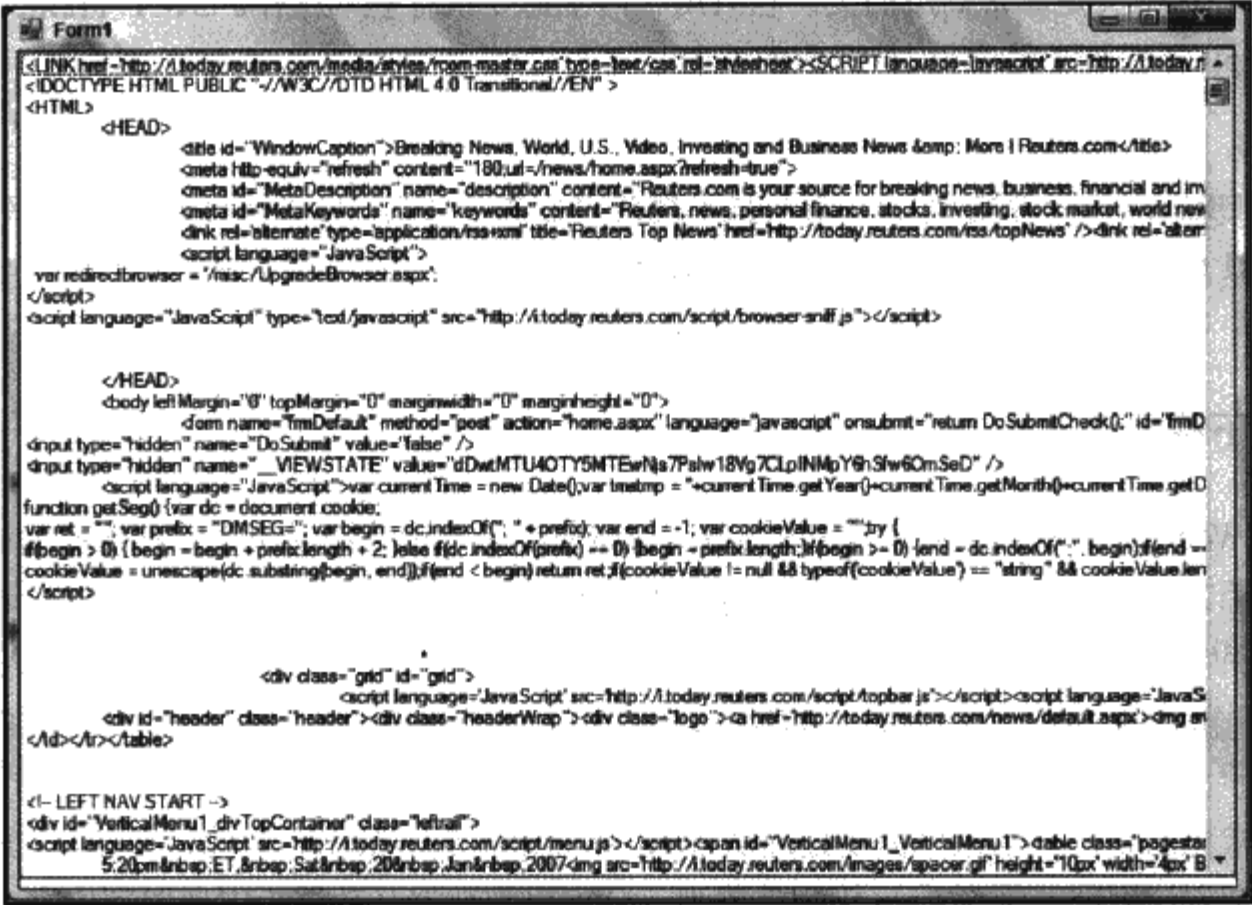


图 41-1

在 WebClient 类中还有一个方法 OpenWrite()，它可以返回一个可写的数据流，并把数据发送给 URI。也可以指定用于把数据发送给主机的方法；默认的方法是 POST。下面的代码段假定在本地机器上有一个可写的目录 accept，这段代码在该目录下创建文件 newfile.txt，其内容为 "Hello World"：

```
WebClient webClient = new WebClient();
Stream stream = webClient.OpenWrite("http://localhost/accept/newfile.txt", "PUT");
StreamWriter streamWriter = new StreamWriter(stream);
streamWriter.WriteLine("Hello World");
streamWriter.Close();
```

41.1.3 上传文件

WebClient 类还提供了 UploadFile()方法和 UploadData()方法。在需要传送 HTML 窗体或上传整个文件时，就可以使用这两个方法。UploadFile()方法用于把指定的文件上传到指定的位置，其中的文件名已经给出；而 UploadData()方法用于把二进制数据上传至指定的 URI，那些二进制数据是作为字节数组提供的(还有一个 DownloadData()方法，用于从 URI 中检索字节数组)：

```
WebClient client = new WebClient();
client.UploadFile("http://www.ourwebsite.com/NewFile.htm",
    "C:\\WebSiteFiles\\NewFile.htm");
byte [] image;
```

```
// code to initialise image so it contains all the binary data for
// some jpg file
client.UploadData("http://www.ourwebsite.com/NewFile.jpg", image);
```

## 41.2 WebRequest 类和 WebResponse 类

WebClient 类使用起来比较简单，但是它的功能非常有限，特别是不能使用它提供身份验证证书。这样，在上传数据时问题就出现了，许多站点都不会接受没有身份验证的上传文件。尽管可以给请求添加标题信息并检查响应中的标题信息，但这仅限于一般意义上的检查，对于任何一个协议，WebClient 没有具体的支持。由于 WebClient 是非常一般的类，可以使用任意协议发送请求和接收响应(例如 HTTP、FTP 等)。它不能处理任一协议的任何附加特性，例如专用于 HTTP 的 cookie。如果想利用这些特性，就需要使用 System.Net 命名空间中以 WebRequest 类和 WebResponse 类为基类的一系列类。

首先讨论怎样使用这些类下载 Web 页——这个示例与前面的示例一样，但使用 WebRequest 类和 WebResponse 类。在此过程中，将解释涉及到的类的层次结构，然后阐述怎样利用这个层次所支持的其他 HTTP 特性。

下面的代码是在 BasicWebClient 示例的基础上修改而成的，目的是让它使用 WebRequest 类和 WebResponse 类。

```
public Form1()
{
    InitializeComponent();

    WebRequest wrq = WebRequest.Create("http://www.reuters.com");
    WebResponse wrs = wrq.GetResponse();
    Stream strm = wrs.GetResponseStream();
    StreamReader sr = new StreamReader(strm);
    string line;
    while ( (line = sr.ReadLine()) != null)
    {
        listBox1.Items.Add(line);
    }
    strm.Close();
}
```

在这段代码中，首先对代表 Web 请求的对象进行实例化。但在此并不是使用构造函数来实例化对象，而是调用静态的 WebRequest.Create()方法，在下一小节中将解释这样做的原因。WebRequest 类是支持不同网络协议的类层次结构的一部分，为了给请求类型接收一个对正确对象的引用，需要一个工厂(factory)机制。WebRequest.Create()方法会为给定的协议创建合适的对象。

WebRequest 类代表要给某个 URI 发送信息的请求，URI 作为参数传送给 Create()方法。WebResponse 类代表从服务器获取的数据。调用 WebRequest.GetResponse()方法，实际上是把请求发送给 Web 服务器，创建一个 Response 对象，检查返回的数据。与 WebClient 对象一样，可以得到一个代表数据的数据流，但是，这里的数据流是使用 WebResponse.GetResponseStream()方法获得的。



## WebRequest 和 WebResponse 的其他特性

下面将讨论 WebRequest 和 WebResponse 和其他相关的类提供的良好支持。

### 1. HTTP 标题信息

HTTP 协议的一个重要方面就是能够利用请求和响应数据流发送扩展的标题信息。标题信息可以包括 cookies、以及发送请求的特定浏览器(用户代理)的一些详细信息。.NET Framework 为访问最重要的数据提供了支持。WebRequest 类和 WebResponse 类提供了读取标题信息的一些支持。而两个派生的类 HttpWebRequest 类和 HttpWebResponse 类提供了其他 HTTP 特定的信息。如后面所述,用 HTTP URI 创建 WebRequest 会生成一个 HttpWebRequest 对象实例。因为 HttpWebRequest 派生自 WebRequest,可以在需要 WebRequest 的任何地方使用新实例。另外,还可以把实例的类型强制转换为 HttpWebRequest 引用,访问 HTTP 协议特定的属性。同样,在使用 HTTP 时,GetResponse()方法调用会把 HttpWebResponse 实例返回为 HttpWebResponse 引用,也可以进行一个简单的强制转换,以访问 HTTP 特定的特性。

可以在 GetResponse()方法调用之前添加如下代码,检查两个标题属性:

```
WebRequest wrq = WebRequest.Create("http://www.wrox.com");
HttpWebRequest hwrq = (HttpWebRequest)wrq;

listBox1.Items.Add("Request Timeout (ms) = " + wrq.Timeout);
listBox1.Items.Add("Request Keep Alive = " + hwrq.KeepAlive);
listBox1.Items.Add("Request AllowAutoRedirect = " + hwrq.AllowAutoRedirect);
```

Timeout 属性的单位是毫秒,其默认值是 100 000。可以设置这个属性,以控制 WebRequest 对象在产生 WebException 之前花多长时间等待响应。可以检查属性 WebException.Status,看看产生异常的原因。这个枚举类型包括超时、连接失败、协议错误等的状态码。

KeepAlive 属性是对 HTTP 协议的特定扩展,所以可以通过 HttpWebRequest 引用访问这个属性。该属性允许多个请求使用同一个连接,在后续的请求中节省关闭和重新打开连接的时间。其默认值为 true。

AllowAutoRedirect 属性也是专用于 HttpWebRequest 类的,使用这个属性可以控制 Web 请求是否应自动跟随 Web 服务器上的重定向响应。其默认值也是 true。如果只允许次数有限的重定向,可以把 HttpWebRequest 的 MaximumAutomaticRedirections 属性设置为想要的数值。

请求和响应类把大多数重要的标题显示为属性,也可以使用 Headers 属性本身显示标题的总集合。在 GetResponse()方法调用的后面添加如下代码,把所有的标题都放在列表框中:

```
WebRequest wrq = WebRequest.Create("http://www.wrox.com");
WebResponse wrs = wrq.GetResponse();
WebHeaderCollection whc = wrs.Headers;
for(int i = 0; i < whc.Count; i++)
{
    listBox1.Items.Add("Header " + whc.GetKey(i) + " : " + whc[i]);
}
```

这个示例代码会产生如图 41-2 所示的标题列表。

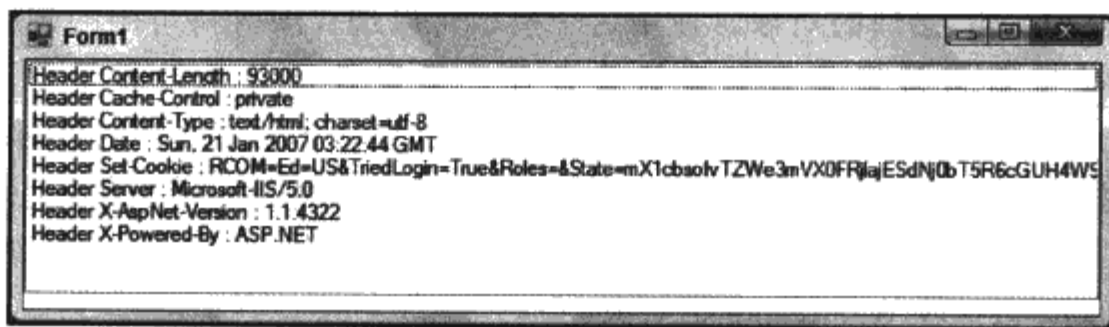


图 41-2

## 2. 身份验证

WebRequest 类中的另一个属性是 Credentials。如果需要把身份验证证书附带在请求中，就可以用用户名和密码创建 NetworkCredential 类(也在 System.Net 命名空间中)的一个实例。在调用 GetResponse()之前，添加下述代码：

```
NetworkCredential myCred = new NetworkCredential("myusername", "mypassword");
wrq.Credentials = myCred;
```

## 3. 使用代理

许多公司都需要使用代理服务器进行所有类型的 HTTP 或 FTP 请求。代理服务器常常使用某种形式的安全性(通常是用户名和密码)，路由公司的所有请求和响应。对于使用 WebClient 或 WebRequest 对象的应用程序，需要考虑这些代理服务器。与前面的 NetworkCredential 对象一样，在进行调用，执行请求之前，需要使用 WebProxy 对象。

```
WebProxy wp = new WebProxy("192.168.1.100", true);
wp.Credentials = new NetworkCredential("user1", "user1Password");
WebRequest wrq = WebRequest.Create("http://www.reuters.com");
wrq.Proxy = wp;
WebResponse wrs = wrq.GetResponse();
```

如果除了证书之外，还需要设计用户的域，就应在 NetworkCredential 实例上使用另一个签名：

```
WebProxy wp = new WebProxy("192.168.1.100", true);
wp.Credentials = new NetworkCredential("user1", "user1Password",
    "myDomain");
WebRequest wrq = WebRequest.Create("http://www.reuters.com");
wrq.Proxy = wp;
WebResponse wrs = wrq.GetResponse();
```

## 4. 异步页面请求

WebRequest 类的另一个特性就是可以异步请求页面。由于在给主机发送请求到接收响应之间有很长的延迟，因此，异步请求页面就显得比较重要。像 WebClient.DownloadData() 和 WebRequest.GetResponse() 等方法，在响应没有从服务器回来之前，是不会返回的。如果不希望在那段时间中应用程序处于等待状态，可以使用 BeginGetResponse() 方法和 EndGetResponse() 方法，BeginGetResponse() 方法可以异步工作，并立即返回。在底层，运行库会异步管理一个后台线程，从服务器上接收响应。BeginGetResponse() 方法不返回 WebResponse 对象，而是返回

一个执行 `IAsyncResult` 接口的对象。使用这个接口可以选择或等待可用的响应，然后调用 `EndGetResponse()` 搜集结果。

也可以把一个回调委托发送给 `BeginGetResponse()` 方法。该回调委托的目的地是一个返回类型为 `void` 并把 `IAsyncResult` 引用作为参数的方法，当工作线程完成了搜集响应的任务后，运行库就调用该回调委托，通知用户工作已完成。如下面的代码所示，在回调方法中调用 `EndGetResponse()` 可以接收 `WebResponse` 对象：

```
public Form1()
{
    InitializeComponent();

    WebRequest wrq = WebRequest.Create("http://www.reuters.com");
    wrq.BeginGetResponse(new AsyncCallback(OnResponse), wrq);
}

protected void OnResponse(IAsyncResult ar)
{
    WebRequest wrq = (WebRequest)ar.AsyncState;
    WebResponse wrs = wrq.EndGetResponse(ar);

    // read the response ...
}
```

注意可以把 `WebRequest` 对象传送为 `BeginGetResponse()` 的第二个参数，检索最初的 `WebRequest` 对象。第三个参数是一个对象引用，称为状态参数，在回调方法中，可以使用 `IAsyncResult` 的 `AsyncState` 属性检索相同的状态对象。

### 41.3 把输出结果显示为 HTML 页面

第一个示例说明了 .NET 基类可以从 Internet 上下载和处理数据。但是，迄今为止，从 Internet 上下载的文件都是以纯文本显示的。人们总是希望以 Internet Explorer 的界面样式查看 HTML 文件，以看到 Web 文档的实际面貌。遗憾的是，Microsoft 的 Internet Explorer 并没有 .NET 版本，但这并不意味着这个任务不能完成。在 .NET Framework 2.0 推出之前，可以引用封装了 Internet Explorer 的 COM 对象，使用 .NET 交互操作功能，把应用程序用作浏览器。现在有了 .NET Framework 2.0 和 3.5，就可以在 Windows 窗体应用程序中使用内置的 `WebBrowser` 控件。

`WebBrowser` 控件封装了 COM 对象，甚至可以更方便地完成以前复杂的任务。除了使用 `WebBrowser` 控件之外，另一个选项是使用编程功能，在代码中调用 Internet Explorer 实例。

如果不使用 `WebBrowser` 控件，可以使用 `System.Diagnostics` 命名空间中的 `Process` 类，编程打开 Internet Explorer 过程，导航到给定的 Web 页。

```
Process myProcess = new Process();
myProcess.StartInfo.FileName = "iexplore.exe";
myProcess.StartInfo.Arguments = "http://www.wrox.com";
myProcess.Start();
```

但是，上面的代码会把 IE 作为单独的窗口打开，而应用程序并没有与新窗口相连接，因此不能控制浏览器。

另一方面,使用 WebBrowser 控件,可以把浏览器作为应用程序的一个集成部分来显示和控制。WebBrowser 控件相当复杂,提供了许多方法、属性和事件。

### 41.3.1 在应用程序中进行简单的 Web 浏览

为了简单起见,首先创建一个 Windows 窗体应用程序,它只有一个 TextBox 控件和一个 WebBrowser 控件。建立该应用程序,让终端用户在文本框中输入一个 URL,按下回车键。WebBrowser 控件就会提取 Web 页面,显示得到的文档。

在 Visual Studio 2008 设计器中,应用程序如图 41-3 所示。

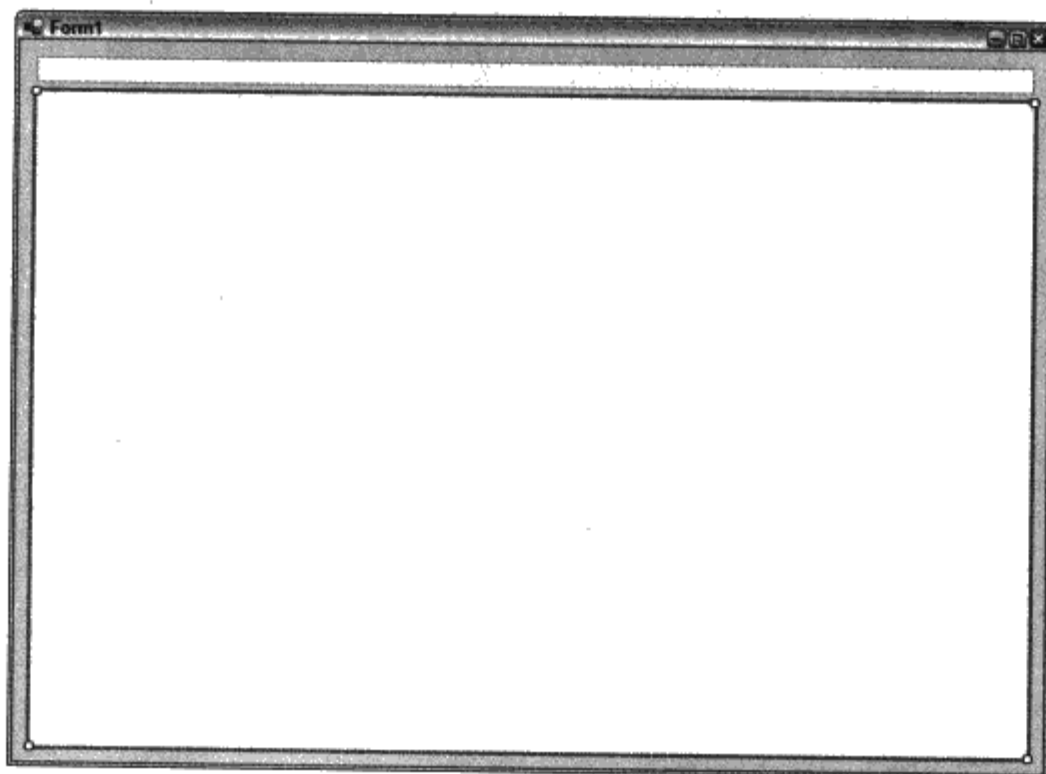


图 41-3

在这个应用程序中,终端用户输入 URL,按下回车键后,这个按键动作就会注册到应用程序中,WebBrowser 控件就会开始检索请求的页面,然后显示在该控件中。

该应用程序的代码如下所示:

```
using System;
using System.Windows.Forms;

namespace CSharpInternet
{
    partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void textBox1_KeyPress(object sender, KeyPressEventArgs e)
        {
            if (e.KeyChar == (char)13)
            {
                webBrowser1.Navigate(textBox1.Text);
            }
        }
    }
}
```



在这个示例中, 终端用户在文本框中按下的每个键都会被 `textBox1_KeyPress` 事件捕获, 如果输入的字符是一个回车键(按下回车键, 其键码是(char)13), 就用 `WebBrowser` 控件采取行动。使用 `WebBrowser` 控件的 `Navigate` 方法, 通过 `textBox1.Text` 属性指定 URL(指定为字符串), 最终结果如图 41-4 所示。



图 41-4

### 41.3.2 启动 Internet Explorer 实例

读者可能对上一节描述的把浏览器放在应用程序内部不感兴趣, 只对让用户在一般的浏览器中查找 Web 站点感兴趣(例如, 单击应用程序中的一个链接)。为了演示这个功能, 创建一个 Windows 窗体应用程序, 其中有一个 `LinkLabel` 控件。例如, 可以在窗体上放置一个 `LinkLabel` 控件, 显示“Visit our company website!”。

有了这个控件后, 就可以使用下面的代码, 在一个单独的浏览器中启动公司的 Web 站点, 而不是直接在应用程序的窗体中启动:

```
private void linkLabel1_LinkClicked(object sender, LinkLabelLinkClickedEventArgs e)
{
    WebBrowser wb = new WebBrowser();
    wb.Navigate("http://www.wrox.com", true);
}
```

在这个示例中, 用户单击 `LinkLabel` 控件时, 就会创建 `WebBrowser` 类的一个新实例。然后使用 `WebBrowser` 类的 `Navigate` 方法, 代码指定了 Web 页面的位置和一个布尔值, 该布尔值表示是在 Windows 窗体应用程序内部打开这个端点(其值为 `false`), 还是在一个单独的浏览器中打开这个端点(其值为 `true`)。它默认设置为 `false`。在前面的构造过程中, 当终端用户单击 Windows 应用程序中的链接时, 就实例化一个浏览器实例, 并立即加载 `www.Wrox.com` 站点。



### 41.3.3 给应用程序提供更多的 IE 类型特性

在前面的例子中，直接在 Windows 窗体应用程序中使用 WebBrowser 控件的实例时，单击页面上的链接，TextBox 控件中的文本不会更新，显示浏览过程的站点 URL。要更正这个错误，应监听 WebBrowser 控件中的事件，给控件添加处理程序。

为此，要用 HTML 页面的标题更新窗体的标题。只需使用 Navigated 事件，更新窗体的 Text 属性即可：

```
private void webBrowser1_Navigated(object sender, EventArgs e)
{
    this.Text = webBrowser1.DocumentTitle.ToString();
}
```

在这个示例中，WebBrowser 控件移动到另一个页面上时，就触发 Navigated 事件，并把窗体的标题改为所查看的页面的标题。在一些情况下，处理 Web 上的页面时，即使输入了指定的地址，也会被重定向另一个页面上。用户希望在窗体的文本框(地址栏)中反映这个变化。为此，应根据所查看页面的完整 URL 改变窗体的文本框。此时也可以使用 WebBrowser 控件的 Navigated 事件：

```
private void webBrowser1_Navigated(object sender, WebBrowserNavigatedEventArgs e)
{
    textBox1.Text = webBrowser1.Url.ToString();
    this.Text = webBrowser1.DocumentTitle.ToString();
}
```

这里，在 WebBrowser 控件下载完请求的页面后，触发 Navigated 事件。我们只需把 textBox1 控件的 Text 值更新为页面的 URL 即可。也就是说，页面加载到 WebBrowser 控件的 HTML 容器后，如果 URL 在这个过程中发生变化(例如，有一个重定向过程)，新的 URL 就会显示在文本框中。如果使用这些步骤导航到 Wrox 网站(<http://www.wrox.com>)，页面的 URL 会立即改为 <http://www.wrox.com/WileyCDA/>。这个过程也说明，如果终端用户单击了 HTML 视图中的一个链接，也会在文本框中显示新请求页面的 URL。

进行了这些修改后，运行应用程序。窗体的标题和地址栏会像 Microsoft 的 Internet Explorer 一样变化，如图 41-5 所示。

接着，创建 IE 样式的工具栏，让终端用户更多地控制 WebBrowser 控件。这样，就可以使用 Back、Forward、Stop、Refresh 和 Home 按钮了。

这里不使用 ToolBar 控件，而是在窗体顶部地址栏的上面添加一组 Button 控件。在控件的顶部添加 5 个按钮，如图 41-6 所示。

在这个示例中，修改按钮上的文本，以显示按钮的作用。当然，还可以使用屏幕捕捉功能，借用 IE 的按钮图像。按钮命名为 buttonBack、buttonForward、buttonStop、buttonRefresh 和 buttonHome。为了能重置大小，应把右边 3 个按钮的 Anchor 属性设置为 Top, Right。

开始时，buttonBack、buttonForward 和 buttonStop 应是禁用的，因为如果没有在 WebBrowser 控件中加载初始页面，就不能使用这些按钮。以后应告诉应用程序，根据用户在页面堆栈的位置，何时启用和禁用 Back 和 Forward 按钮。另外，在加载页面时，需要启用 Stop 按钮，在页面加载完毕后，需要禁用 Stop 按钮。在页面上再添加一个 Submit 按钮，用于提交所请求的 URL。

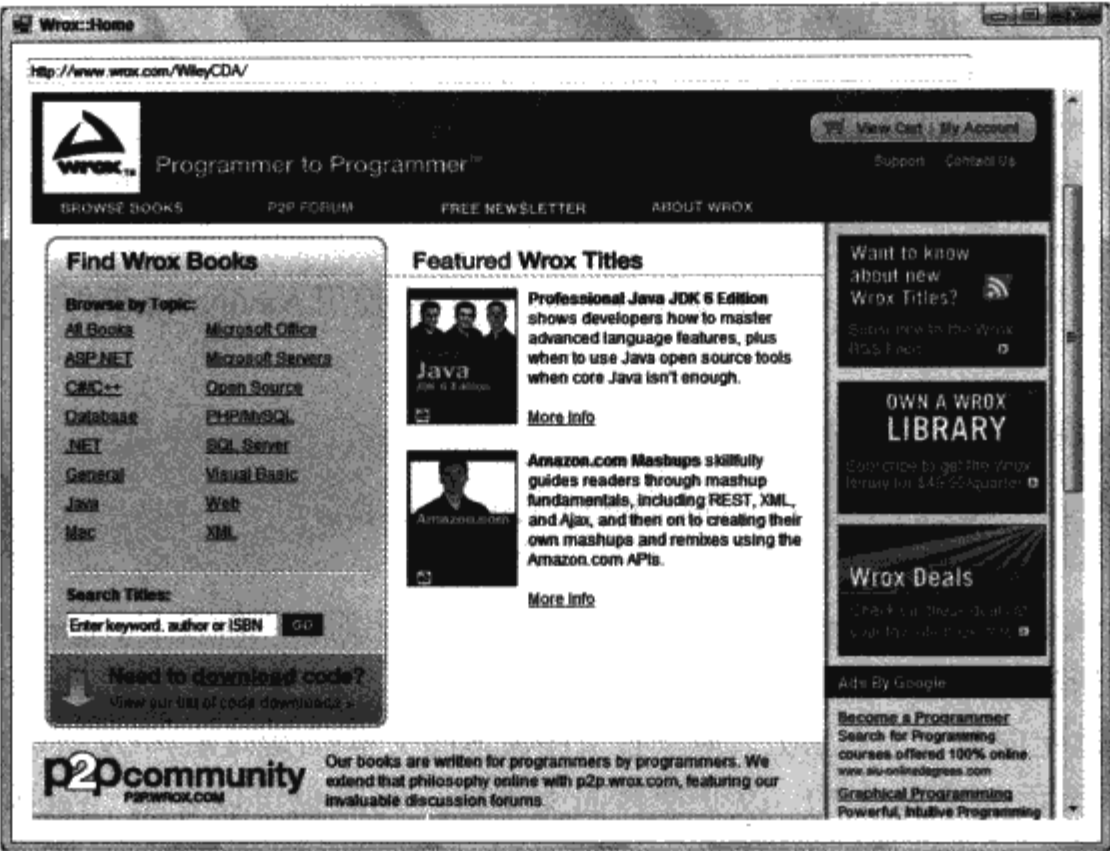


图 41-5

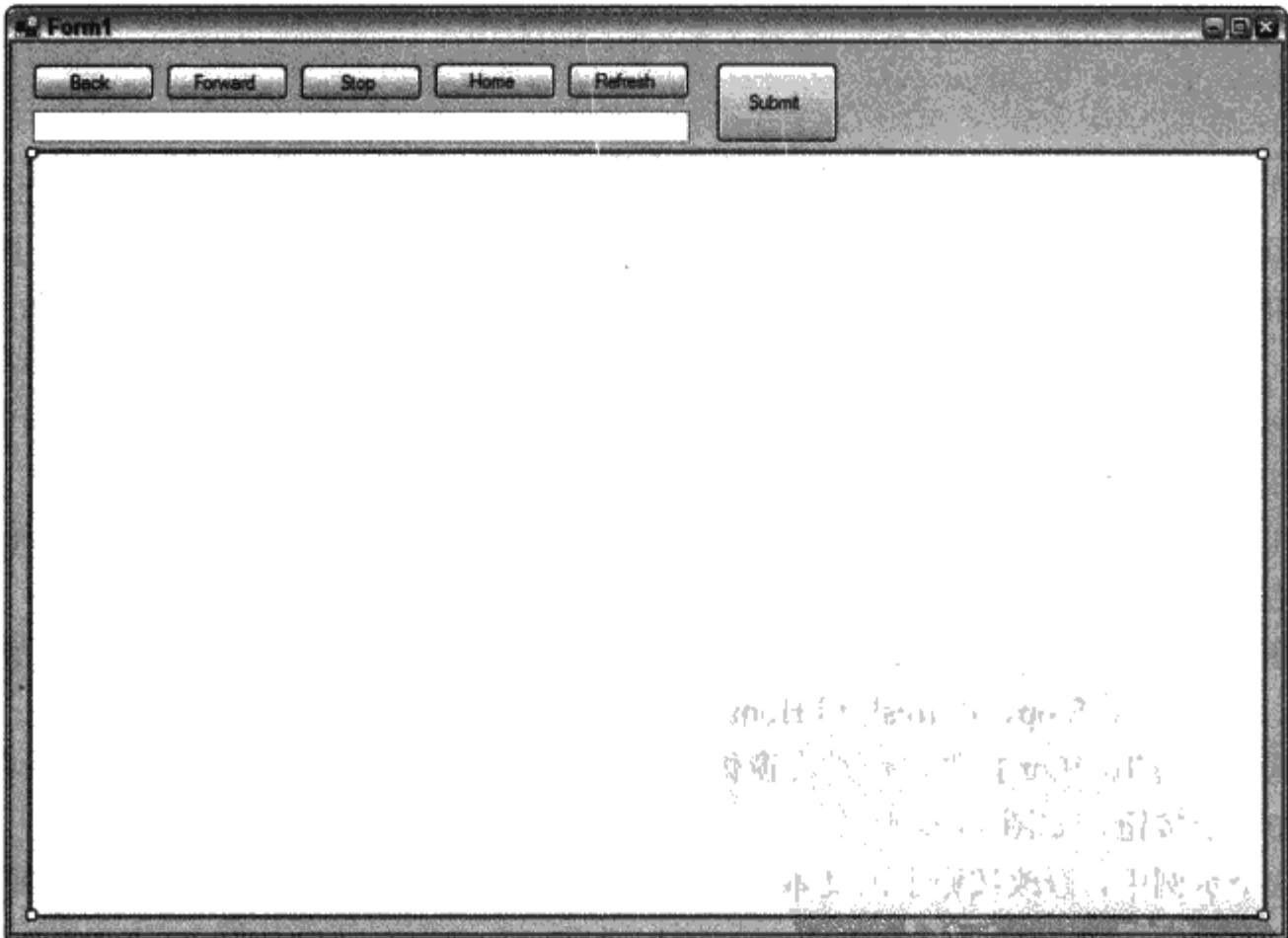


图 41-6

下面给按钮添加功能。WebBrowser 类有我们需要的所有方法，所以这是很简单的：

```
using System;
using System.Windows.Forms;

namespace CSharpInternet
```

```
{
    partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void textBox1_KeyPress(object sender, KeyPressEventArgs e)
        {
            if (e.KeyChar == (char)13)
            {
                webBrowser1.Navigate(textBox1.Text);
            }
        }

        private void webBrowser1_Navigated(object sender,
            WebBrowserNavigatedEventArgs e)
        {
            textBox1.Text = webBrowser1.Url.ToString();
            this.Text = webBrowser1.DocumentTitle.ToString();
        }
    }
}
```

```
private void Form1_Load(object sender, EventArgs e)
{
    buttonBack.Enabled = false;
    buttonForward.Enabled = false;
    buttonStop.Enabled = false;
}

private void buttonBack_Click(object sender, EventArgs e)
{
    webBrowser1.GoBack();
    textBox1.Text = webBrowser1.Url.ToString();
}

private void buttonForward_Click(object sender, EventArgs e)
{
    webBrowser1.GoForward();
    textBox1.Text = webBrowser1.Url.ToString();
}

private void buttonStop_Click(object sender, EventArgs e)
{
    webBrowser1.Stop();
}

private void buttonHome_Click(object sender, EventArgs e)
{
    webBrowser1.GoHome();
    textBox1.Text = webBrowser1.Url.ToString();
}

private void buttonRefresh_Click(object sender, EventArgs e)
{
    webBrowser1.Refresh();
}

private void buttonSubmit_Click(object sender, EventArgs e)
{
}
```

```

        webBrowser1.Navigate(textBox1.Text);
    }

    private void webBrowser1_Navigating(object sender,
        WebBrowserNavigatingEventArgs e)
    {
        buttonStop.Enabled = true;
    }

    private void webBrowser1_DocumentCompleted(object sender,
        WebBrowserDocumentCompletedEventArgs e)
    {
        buttonStop.Enabled = false;
        if (webBrowser1.CanGoBack)
        {
            buttonBack.Enabled = true;
        }
        else
        {
            buttonBack.Enabled = false;
        }
        if (webBrowser1.CanGoForward)
        {
            buttonForward.Enabled = true;
        }
        else
        {
            buttonForward.Enabled = false;
        }
    }
}

```

在这个示例中要执行许多不同的操作，因为终端用户在使用这个应用程序时，有那么多选项。首先，对于每个按钮单击事件，都有一个特定的 `WebBrowser` 类方法来执行操作。例如，对于窗体上的 `Back` 按钮，可以使用 `WebBrowser` 控件的 `GoBack()` 方法。其他按钮也一样，`Forward` 按钮要使用 `GoForward()` 方法，其他按钮要使用 `Stop()`、`Refresh()` 和 `GoHome()`。所以，很容易创建工具栏，其操作类似于 Microsoft 的 Internet Explorer。

在第一次加载窗体时，`Form1_Load` 事件禁用相应的按钮，此时，终端用户可以在文本框中输入 URL，单击 `Submit` 按钮，让应用程序检索相应的页面。

为了管理按钮的启用和禁用，必须建立一组事件。如前所述，只要开始下载，就需要启用 `Stop` 按钮。为此，只需给 `Navigating` 事件添加事件处理程序，启用 `Stop` 按钮：

```

private void webBrowser1_Navigating(object sender,
    WebBrowserNavigatingEventArgs e)
{
    buttonStop.Enabled = true;
}

```

接着，文档加载完毕后，再次禁用 `Stop` 按钮：

```

private void webBrowser1_DocumentCompleted(object sender,
    WebBrowserDocumentCompletedEventArgs e)
{
    buttonStop.Enabled = false;
}

```

为了启用和禁用相应的 Back 和 Forward 按钮, 应考虑在页面堆栈中后退和前进的功能。这是使用 CanGoForwardChanged() 和 CanGoBackChanged() 事件实现的:

```
private void webBrowser1_CanGoBackChanged(object sender, EventArgs e)
{
    if (webBrowser1.CanGoBack == true)
    {
        buttonBack.Enabled = true;
    }
    else
    {
        buttonBack.Enabled = false;
    }
}

private void webBrowser1_CanGoForwardChanged(object sender, EventArgs e)
{
    if (webBrowser1.CanGoForward == true)
    {
        buttonForward.Enabled = true;
    }
    else
    {
        buttonForward.Enabled = false;
    }
}
```

现在运行项目, 访问一个 Web 页面, 单击几个链接。还应能使用工具栏, 提升浏览过程。最终结果如图 41-7 所示。

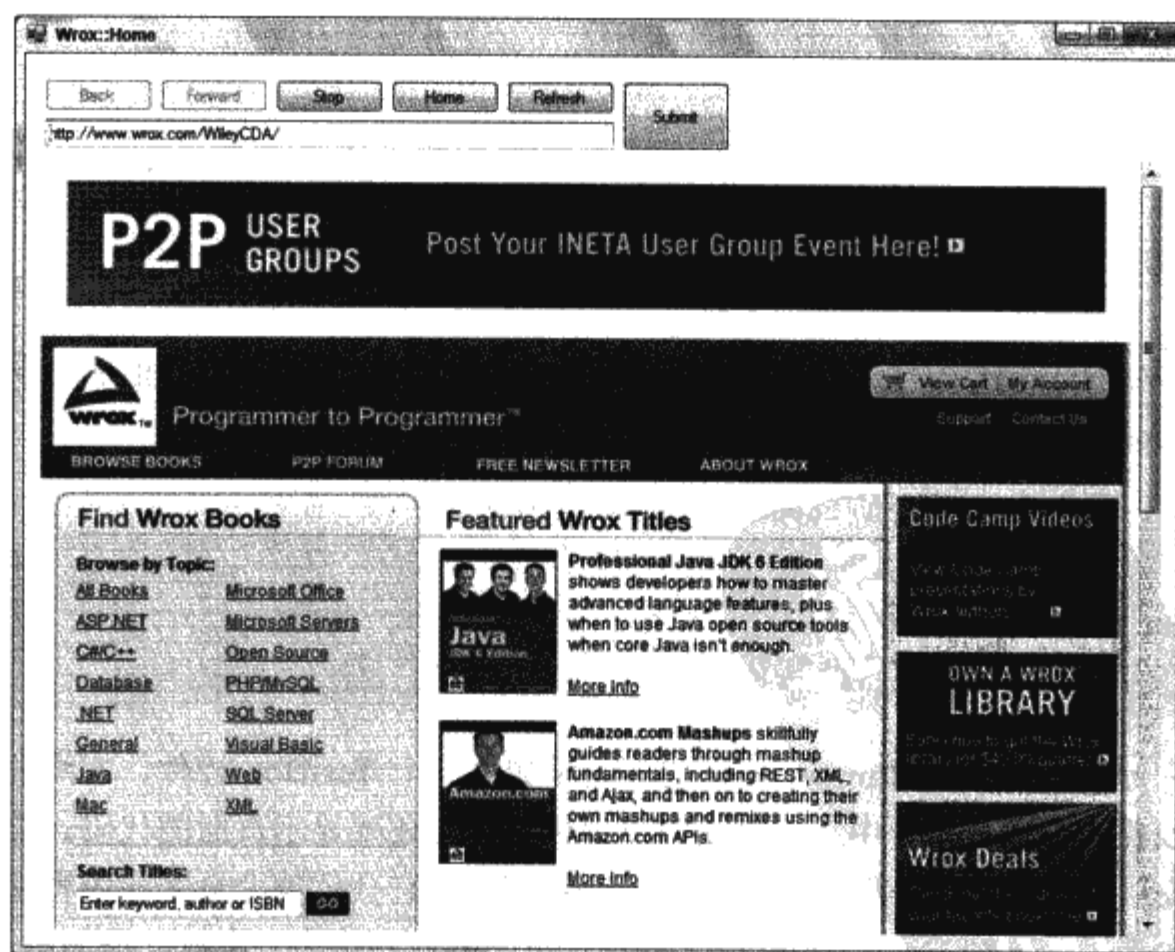


图 41-7



### 41.3.4 使用 WebBrowser 控件打印

用户不仅可以使⽤ WebBrowser 控件查看页面和文档,还可以使⽤ WebBrowser 控件把这些页面和文档发送到打印机上,进⾏打印。要打印在 WebBrowser 控件中查看的页面或文档,只需使⽤下面的构造代码:

```
webBrowser1.Print();
```

与以前相同,不必查看页面或文档,就可以打印它。例如,可以使⽤ WebBrowser 类加载 HTML 文档,并打印它,⽽无需显示加载的文档,其代码如下所示:

```
WebBrowser wb = new WebBrowser();
wb.Navigate("http://www.wrox.com");
wb.Print();
```

### 41.3.5 显示请求页面的代码

在本章的开头,我们使⽤ WebRequest 和 Stream 类获得一个远程页面,显示所请求页面的代码。下面的代码也可以完成这个任务:

```
public Form1()
{
    InitializeComponent();

    System.Net.WebClient Client = new WebClient();
    Stream strm = Client.OpenRead("http://www.reuters.com");
    StreamReader sr = new StreamReader(strm);
    string line;
    while ( (line=sr.ReadLine()) != null )
    {
        listBox1.Items.Add(line);
    }

    strm.Close();
}
```

引入了 WebBrowser 控件后,这个任务就更容易完成。只需修改本章前面开发的浏览器应用程序,在 Document\_Completed 事件中添加⼀行代码,如下所示:

```
private void webBrowser1_DocumentCompleted(object sender,
    WebBrowserDocumentCompletedEventArgs e)
{
    buttonStop.Enabled = false;
    textBox2.Text = webBrowser1.DocumentText.ToString();
}
```

在应用程序中,在 WebBrowser 控件的下面添加另一个 TextBox 控件。在终端用户请求页面时,不仅要在 TextBox 控件中显示页面的可视化部分,还要显示页面的代码。要显示页面的代码,只需使⽤ WebBrowser 控件的 DocumentText 属性,它会把整个页面的内容显示为一个字符串。另一个选项是使⽤ DocumentStream 属性把页面的内容作为一个流。添加第二个文本框,把页面的内容显示为字符串,结果如图 41-8 所示。

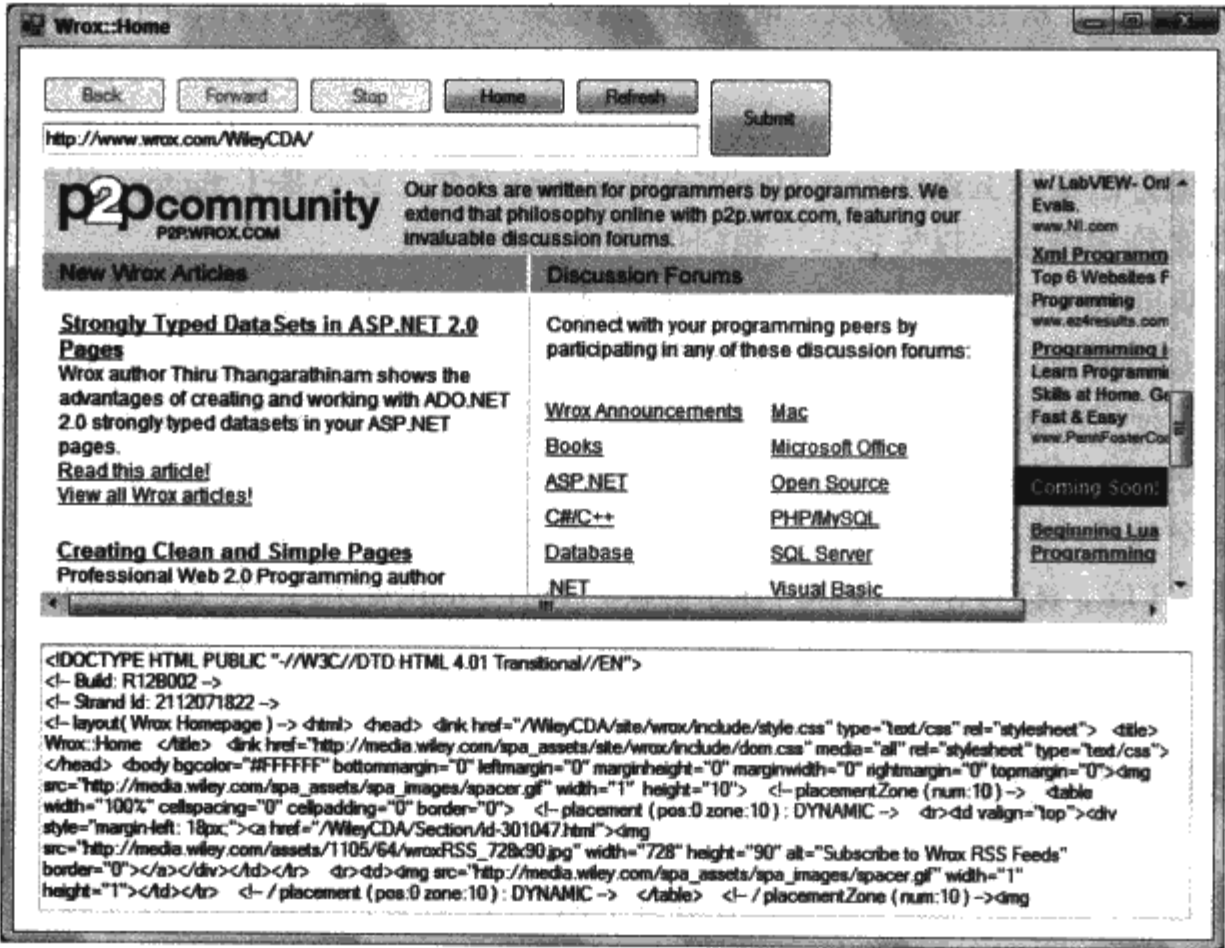


图 41-8

41.3.6 WebRequest 和 WebResponse 的层次结构

本节详细讨论 WebRequest 类和 WebResponse 类的底层体系结构。图 41-9 显示的是相关类的继承层次结构。

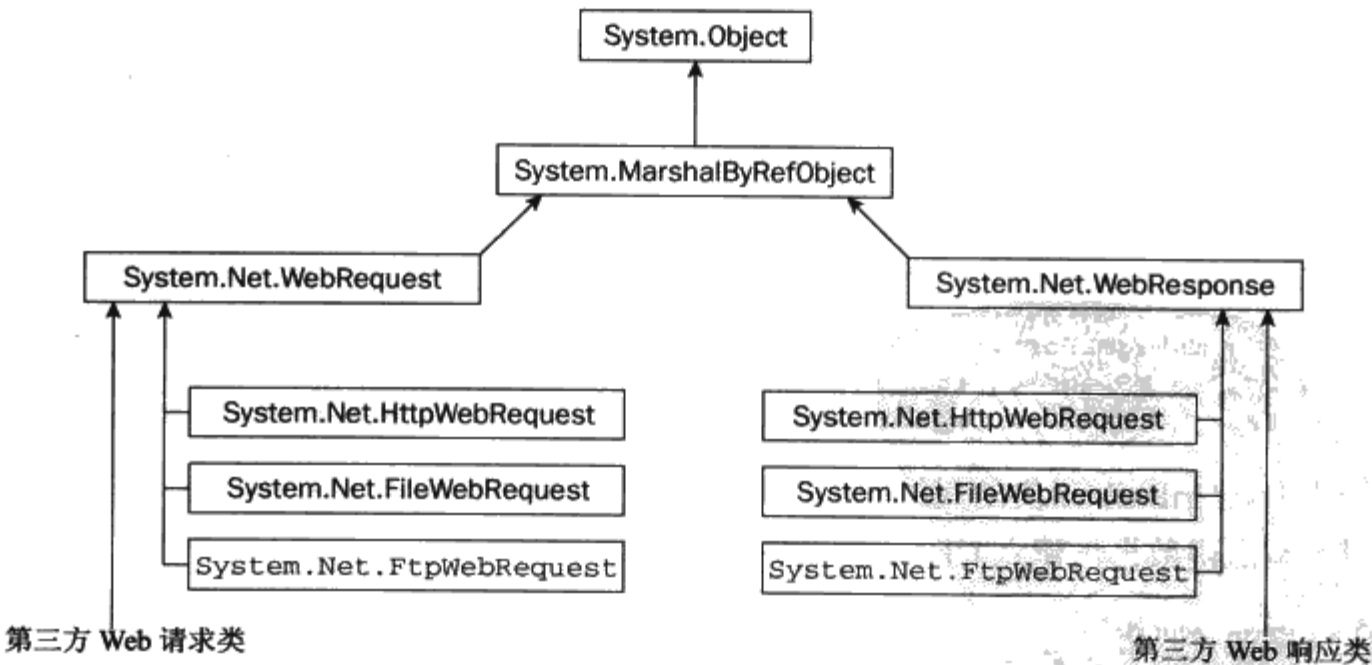


图 41-9

这个体系结构不仅仅包含刚才代码中使用的两个类。实际上，WebRequest类和WebResponse类都是抽象的，不能进行实例化。这些基类提供了用于处理Web请求和响应的通用功能，这些

功能独立于给定操作所使用的协议。请求总是通过某一协议(例如 HTTP、FTP、SMTP 等)实现的,并由为该协议编写的派生类处理。Microsoft 称之为“可插入的协议”。在前面的代码中,变量定义为对基类的引用,但是 `WebRequest.Create()` 实际上给出了一个 `HttpWebRequest` 对象, `GetResponse()` 方法实际上返回的是 `HttpWebResponse` 对象。这个基于 factory 的机制在客户机代码中隐藏了许多细节,以支持基于相同代码的各种协议。

有了 `WebRequest.Create()`,在 URI 中就不需要专门用于处理 HTTP 协议的对象。`WebRequest.Create()` 检查 URI 中的协议说明符,以实例化和返回一个适当类的对象。这样代码就不必了解所使用的派生类或特定协议的信息。在需要访问协议的特定功能时,应使用派生类的属性和方法,此时要把 `WebRequest` 或 `WebResponse` 的引用转换为派生类。

有了这个体系结构,就应能使用任一通用协议发送请求。但是,Microsoft 目前提供的派生类只适用于 HTTP、HTTPS、FTP 和 FILE 协议。.NET Framework 自从 2.0 版本以来支持 FTP。如果要利用其他的协议,例如 SMTP,则需要使用 WCF(替代了 Windows API)或 `SmtpClient` 类。

## 41.4 实用工具类

本节将讨论一些实用工具类,它们在处理 URI 和 IP 地址时可简化 Web 编程。

### 41.4.1 URI

`Uri` 和 `UriBuilder` 是 `System`(注意:不是 `System.Net`)命名空间中的两个类,它们都用于表示 URI。`UriBuilder` 允许把给定的字符串当作 URI 的组成部分,从而建立一个 URI,而 `Uri` 类允许分析、组合和比较 URI。

对于 `Uri` 类,构造函数需要一个完整的 URI 字符串:

```
Uri MSPage = new
    Uri("http://www.Microsoft.com/SomeFolder/SomeFile.htm?Order=true");
```

`Uri` 类给出了许多只读属性。当 `Uri` 对象构造出来之后,它就不能修改了。

```
string Query = MSPage.Query;           // Order=true;
string AbsolutePath = MSPage.AbsolutePath; // SomeFolder/SomeFile.htm
string Scheme = MSPage.Scheme;         // http
int Port = MSPage.Port;                // 80 (the default for http)
string Host = MSPage.Host;              // www.Microsoft.com
bool IsDefaultPort = MSPage.IsDefaultPort; // true since 80 is default
```

另一方面,`UriBuilder` 类的属性较少:只允许建立完整的 URI。这些属性是可读写的。可以给构造函数提供建立 URI 所需的各个组成部分:

```
Uri MSPage = new
    UriBuilder("http", "www.Microsoft.com", 80, "SomeFolder/SomeFile.htm")
```

或者把值赋给属性,建立 URI 的组成部分。

```
UriBuilder MSPage = new UriBuilder();
MSPage.Scheme = "http";
MSPage.Host = "www.Microsoft.com";
```

```
MSPage.Port = 80;
MSPage.Path = "SomeFolder/SomeFile.htm";
```

在完成 UriBuilder 的初始化后, 就可以使用 Uri 属性获得相应的 Uri 对象。

```
Uri CompletedUri = MSPage.Uri;
```

#### 41.4.2 IP 地址和 DNS 名称

在 Internet 上, 服务器和客户机都由 IP 地址或主机名(也称作 DNS 名称)标识。通常, 主机名是在 Web 浏览器的窗口中键入的友好名称, 例如 `www.wrox.com` 或 `www.microsoft.com` 等。另一方面, IP 地址是计算机用于互相标识的标识符, 它实际上是用于确保 Web 请求和响应到达相应机器的地址。计算机甚至可以有多个 IP 地址。

目前, IP 地址一般是一个 32 位值。例如 `192.168.1.100` 就是一个 32 位 IP 地址。IP 地址的这个格式称为 Internet Protocol 4。目前有许多计算机和其他设备在竞争 Internet 上的一个地点, 所以人们开发了一种新的地址 Internet Protocol 6。IPv6 提供了 64 位 IP 地址。IPv6 至多可以提供  $3 \times 10^{28}$  个不同的地址。.NET Framework 允许应用程序使用 IPv4 和 IPv6。

为了使这些主机名发挥作用, 首先必须发送一个网络请求, 把主机名翻译成 IP 地址, 翻译工作由一个或几个 DNS 服务器完成。

DNS 服务器中保存的一个表把主机名映射为它知道的所有计算机的 IP 地址, 以及用于其他 DNS 服务器在该表中查找它不知道的主机名的其他 IP 地址。本地计算机至少要知道一个 DNS 服务器。网络管理员在计算机启动时配置该信息。

在发送请求之前, 计算机首先应要求 DNS 服务器指出与键入的主机名相对应的 IP 地址。找到正确的 IP 地址后, 计算机就可以定位请求, 并通过网络发送它。这些工作一般都在后台发生, 用户仅浏览 Web 即可。

##### 1. 用于 IP 地址的 .NET 类

.NET Framework 提供了许多能够帮助寻找 IP 地址和主机信息的类。

##### (1) IPAddress 类

IPAddress 类代表 IP 地址。地址本身可以作为 GetAddressBytes 属性, 使用 ToString() 方法可以把 IP 地址转化为用小数点隔开的十进制格式。此外, IPAddress 也执行静态的 Parse() 方法, 这个方法的作用与 ToString() 方法正好相反, 把小数点隔开的十进制字符串转化为 IP 地址。

```
IPAddress ipAddress = IPAddress.Parse("234.56.78.9");
bytes[] address = ipAddress.GetAddressBytes();
string ipString = ipAddress.ToString();
```

在上面的示例中, byte 整型数 address 的值是 IP 地址的二进制表示, 字符串 ipString 的值为文本 "234.56.78.9"。

IPAddress 还提供了许多静态的常量字段, 以返回特殊的 IP 地址。例如, Loopback 地址允许机器给它自己发送消息, 而 Broadcast 允许多路传送到本地网络上。

```
// The following line will set loopback to "127.0.0.1".
// the loopback address indicates the local host.
string loopback = IPAddress.Loopback.ToString();
```

```
// The following line will set broadcast address to "255.255.255.255".
// the broadcast address is used to send a message to all machines on
// the local network.
string broadcast = IPAddress.Broadcast.ToString();
```

## (2) IPEndPoint 类

IPEndPoint 类用于封装与某台主机相关的信息。通过这个类的 HostName 属性(这个属性返回一个字符串), 可以使用主机名称; 通过 AddressList 属性返回一个 IPAddress 对象的数组。下一个示例 DnsLookupResolver 将使用 IPEndPoint 类。

## (3) Dns 类

Dns 类能够与默认的 DNS 服务器进行通信, 以检索 IP 地址。Dns 类有两个重要的静态方法: Resolve()方法和 GetHostByAddress()方法。给 Resolve()方法提供主机名称, Resolve()就可以使用 DNS 服务器获取主机的详细信息; 给 GetHostByAddress()方法提供 IP 地址, GetHostByAddress()也可以返回主机的详细信息。这两个方法都返回一个 IPEndPoint 对象。

```
IPEndPoint wroxHost = Dns.Resolve("www.wrox.com");
IPEndPoint wroxHostCopy = Dns.GetHostByAddress("208.215.179.178");
```

在这段代码中, 两个 IPEndPoint 对象将包含 Wrox.com 服务器的详细信息。

Dns 类与 IPAddress 类和 IPEndPoint 类的不同之处在于: Dns 可以与服务器进行通信, 以获取有关的信息; 而 IPAddress 类和 IPEndPoint 类只是包含许多便利属性的简单数据结构, 可以访问底层的数据。

## 2. DnsLookup 示例

下面通过查找 DNS 名称的示例 DnsLookup, 来阐明与 DNS 和 IP 相关的类, 如图 41-10 所示。

该示例让用户在主文本框中键入 DNS 名称, 当用户单击 Resolve 按钮时, 这个示例就使用 Dns.Resolve()方法检索 IPEndPoint 引用, 显示出主机名和 IP 地址。注意, 显示出的主机名也许与键入的名称不同, 如果一个 DNS 名称(www.microsoft.com)仅担当另一个 DNS 名称(www.microsoft.com.nsac.net)的代理, 就会发生这种情况。

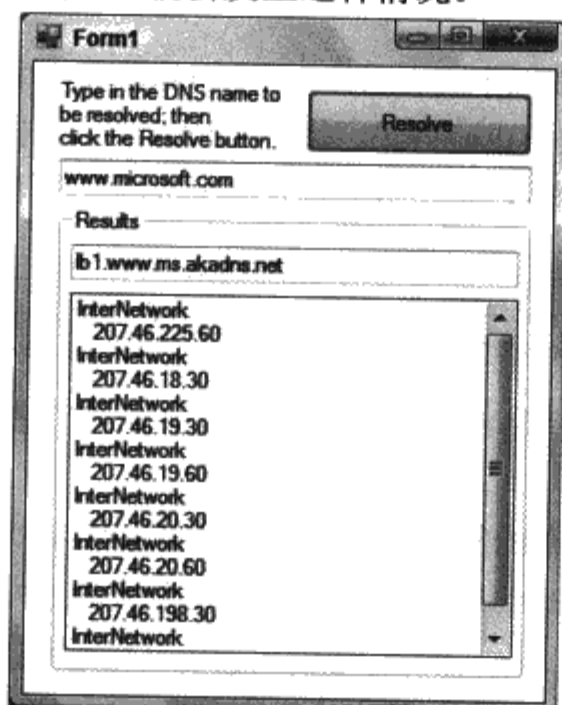


图 41-10



DnsLookup 应用程序是一个标准的 C# Windows 应用程序，给这个应用程序添加如图 41-10 所示的控件，这些控件分别命名为 `textBoxInput`、`btnResolve`、`textBoxHostName` 和 `listboxIPs`。然后，把下面的方法添加给 `Form1` 类，作为 `btnResolve` 单击事件的处理程序。

```
void btnResolve_Click (object sender, EventArgs e)
{
    try
    {
        IPEndPoint iphost = Dns.Resolve(textBoxInput.Text);
        foreach (IPAddress ip in iphost.AddressList)
        {
            string ipaddress = ip.AddressFamily.ToString();
            listBoxIPs.Items.Add(ipaddress);
            listBoxIPs.Items.Add(" " + ip.ToString());
        }
        textBoxHostName.Text = iphost.HostName;
    }
    catch (Exception ex)
    {
        MessageBox.Show("Unable to process the request because " +
            "the following problem occurred:\n" +
            ex.Message, "Exception occurred");
    }
}
```

注意，在这段代码中是如何捕获异常的。如果用户键入了无效的 DNS 名称，或者网络处于断开状态，就会产生异常。

在检索到 `IPEndPoint` 实例之后，使用它的 `AddressList` 属性获取包含 IP 地址的数组，再用 `foreach` 循环遍历该数组。在每次迭代中，都使用 `IPAddress.AddressFamily.ToString()` 方法把 IP 地址显示为整数和字符串。

## 41.5 较低层的协议

本节简要介绍一些在较低层次上进行通信的 .NET 类。

网络的通信分为几个不同的层次，本章迄今为止讨论的类都是工作在最高层，即处理某些命令的一层。如果考虑使用 FTP 传输文件，这个概念就非常容易理解。目前的 GUI 应用程序隐藏了许多 FTP 细节，但在命令行上执行 FTP 还是不久之前的事。在这个环境中，我们显式地键入一些要发送至服务器的命令，以下载、上传和列出文件。

FTP 并不是依赖于文本命令的唯一高层协议，HTTP、SMTP、POP 和其他的协议都基于相似的文本命令，许多现代的图形工具隐藏了命令的传输过程，因此用户一般意识不到这些命令的存在。例如，在 Web 浏览器中键入 URL 时、Web 请求发送给服务器时，浏览器实际上发送给服务器的是一个纯文本的 GET 命令，这个命令与 FTP 的 `get` 命令相似。此外，浏览器也可以发送 POST 命令，表示浏览器在请求上附有其他的数据。

但是，这些协议本身都不足以实现计算机之间的通信。即使客户和服务器都理解某个协议，例如 HTTP，它们仍然不能互相理解，除非另外有协议说明字符是如何传输的，使用的是什么二进制格式，什么电压用于代表二进制数据中的 0 和 1？这些问题都需要协议规定它们，网络

领域的开发人员和硬件工程师通常要查阅协议栈。在列出两个主机进行通信所需的各种协议和机制时，就创建了一个协议栈，其中既有最高层的协议，也有最低层的协议。这种方法利用模块化和分层的方式获得了很有效的通信。

幸运的是，对于大多数的开发工作而言，我们都不需要使用协议堆栈或处理电压级别。但是，如果要编写代码，以便在计算机之间进行高效率的通信，则需要编写的代码可以直接在计算机之间传送二进制数据包。这是 TCP 之类协议的领域，Microsoft 提供的许多类都允许方便地使用该层次上的二进制数据来工作。

低层类

System.Net.Sockets 命名空间包含一些相关类，允许直接发送 TCP 网络请求或在某个端口监听 TCP 网络请求。其中主要的类如表 41-1 所示。

表 41-1

类	用 途
Socket	这个低层的类用于管理连接。WebRequest、TcpClient 和 UdpClient 等类在内部使用这个类
NetworkStream	这个类是从 Stream 派生出来的，它表示来自网络的数据流
SmtpClient	允许通过 SMTP 发送消息(邮件)
TcpClient	允许创建和使用 TCP 连接
TcpListener	允许监听传入的 TCP 连接请求
UdpClient	用于为 UDP 客户创建连接(UDP 是 TCP 的一种替代协议，但没有得到广泛的使用，主要用于本地网络)

1. 使用 SmtpClient

SmtpClient 对象可以通过 SMTP 传送邮件消息。使用 SmtpClient 对象的一个简单示例如下：

```
SmtpClient sc = new SmtpClient("mail.mySmtpHost.com");
sc.Send("evjen@yahoo.com", "editor@wrox.com",
    "The latest chapter", "Here is the latest.");
```

在其最简单的形式中，使用了 SmtpClient 对象的一个实例。在这个例子中，该实例还提供给 SMTP 服务器的主机，在 Internet 上发送邮件消息。还可以使用 Host 属性完成相同的任务：

```
SmtpClient sc = new SmtpClient();
sc.Host = "mail.mySmtpHost.com";
sc.Send("evjen@yahoo.com", "editor@wrox.com",
    "The latest chapter", "Here is the latest.");
```

有了 SmtpClient 后，就可以调用 Send()方法，提供 From 地址、To 地址、主题以及邮件的消息体。

在许多情况下，邮件消息都比这里的示例复杂。为此，还可以给 Send()方法传送一个 MailMessage 对象：

```

SmtpClient sc = new SmtpClient();
sc.Host = "mail.mySmtpHost.com";
MailMessage mm = new MailMessage();
mm.Sender = new MailAddress("evjen@yahoo.com", "Bill Evjen");
mm.To.Add(new MailAddress("editor@wrox.com", "Katie Mohr"));
mm.To.Add(new MailAddress("marketing@wrox.com", "Wrox Marketing"));
mm.CC.Add(new MailAddress("publisher@wrox.com", "Joe Wikert"));
mm.Subject = "The latest chapter";
mm.Body = " < b > Here you can put a long message < /b > ";
mm.IsBodyHtml = true;
mm.Priority = MailPriority.High;
sc.Send(mm);

```

使用 **MailMessage** 可以细调建立邮件消息的方式。我们可以发送 HTML 消息、添加任意多个 **To** 和 **CC** 接收人，修改消息的优先级，使用消息编码，添加附件。添加附件的功能在下面的代码中定义：

```

SmtpClient sc = new SmtpClient();
sc.Host = "mail.mySmtpHost.com";
MailMessage mm = new MailMessage();
mm.Sender = new MailAddress("evjen@yahoo.com", "Bill Evjen");
mm.To.Add(new MailAddress("editor@wrox.com", "Katie Mohr"));
mm.To.Add(new MailAddress("marketing@wrox.com", "Wrox Marketing"));
mm.CC.Add(new MailAddress("publisher@wrox.com", "Joe Wikert"));
mm.Subject = "The latest chapter";
mm.Body = " < b > Here you can put a long message < /b > ";
mm.IsBodyHtml = true;
mm.Priority = MailPriority.High;
Attachment att = new Attachment("myExcelResults.zip",
    MediaTypeNames.Application.Zip);
mm.Attachments.Add(att);
sc.Send(mm);

```

这段代码创建了一个 **Attachment** 对象，使用 **Add()** 方法给 **MailMessage** 对象添加该对象，之后调用 **Send()** 方法。

## 2. 使用 TCP 类

传输控制协议(TCP)类为连接和发送两个点之间的数据提供了简单的方法。端点是 IP 地址和端口号的组合。现有的协议很好地定义了端口号，例如，HTTP 使用端口 80，而 SMTP 使用端口 25，Internet Assigned Number Authority(即 IANA，<http://www.iana.org/>)把端口号赋予这些已知的服务。除非执行某个已知的服务，否则应选择 1024 以上的端口号。

TCP 数据流构成了目前 Internet 上的主要传输流。TCP 通常是首选的协议，因为它提供了有保证的传输、错误校正和缓存。**TcpClient** 类封装了 TCP 连接，提供了许多属性来控制连接，包括缓存、缓存器的大小和超时。通过 **GetStream()** 方法请求 **NetworkStream** 对象时可以附带读写功能。

**TcpListener** 类用 **Start()** 方法监听传入的 TCP 连接。当连接请求到达时，可以使用 **AcceptSocket()** 方法返回一个套接字，以与远程机器通信，或使用 **AcceptTcpClient()** 方法通过高层的 **TcpClient** 对象进行通信。阐明 **TcpListener** 和 **TcpClient** 类如何工作的最简单的方式是举一个示例。

### 3. TcpSend 和 TcpReceive 示例

为了说明这两个类，需要建立两个应用程序。第一个应用程序是 `TcpSend`，如图 41-11 所示。这个应用程序打开一个到服务器的 TCP 连接，并为它自己发送 C#源代码。

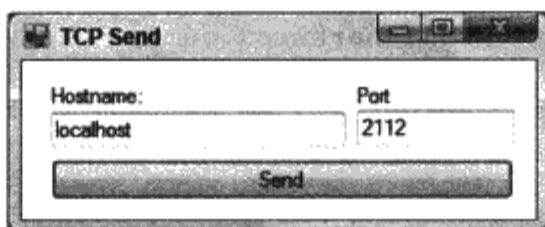


图 41-11

再创建一个 C# Windows 应用程序，其中的窗体包含两个文本框(`txtHost` 和 `txtPort`)，分别用于主机名和端口，该窗体还有一个按钮(`btnSend`)，单击它可以启动连接。首先，确保包含相关的命名空间：

```
using System;
using System.IO;
using System.Net.Sockets;
using System.Windows.Forms;
```

按钮的单击事件处理程序如下所示。

```
private void btnSend_Click(object sender, System.EventArgs e)
{
    TcpClient tcpClient = new TcpClient(txtHost.Text, Int32.Parse(txtPort.Text));
    NetworkStream ns = tcpClient.GetStream();
    FileStream fs = File.Open(Server.MapPath("form1.cs"), FileMode.Open);

    int data = fs.ReadByte();
    while(data != -1)
    {
        ns.WriteByte((byte)data);
        data = fs.ReadByte();
    }

    fs.Close();
    ns.Close();
    tcpClient.Close();
}
```

这个示例用主机名和端口号创建了 `TcpClient`。另外，如果有 `EndPoint` 类的一个实例，就可以把该实例传送给 `TcpClient` 构造函数。在得到 `NetworkStream` 类的一个实例后，打开源代码文件，开始读取字节。与许多二进制流一样，这里也需要将 `ReadByte()` 方法的返回值和 -1 相比较，以确定是否到达流的末尾。循环读取了所有的字节，并把它们发送给网络流后，就应关闭所有打开的文件、连接和流。

在连接的另一端，`TcpReceive` 应用程序显示传输完成后接收到的文件，如图 41-12 所示。

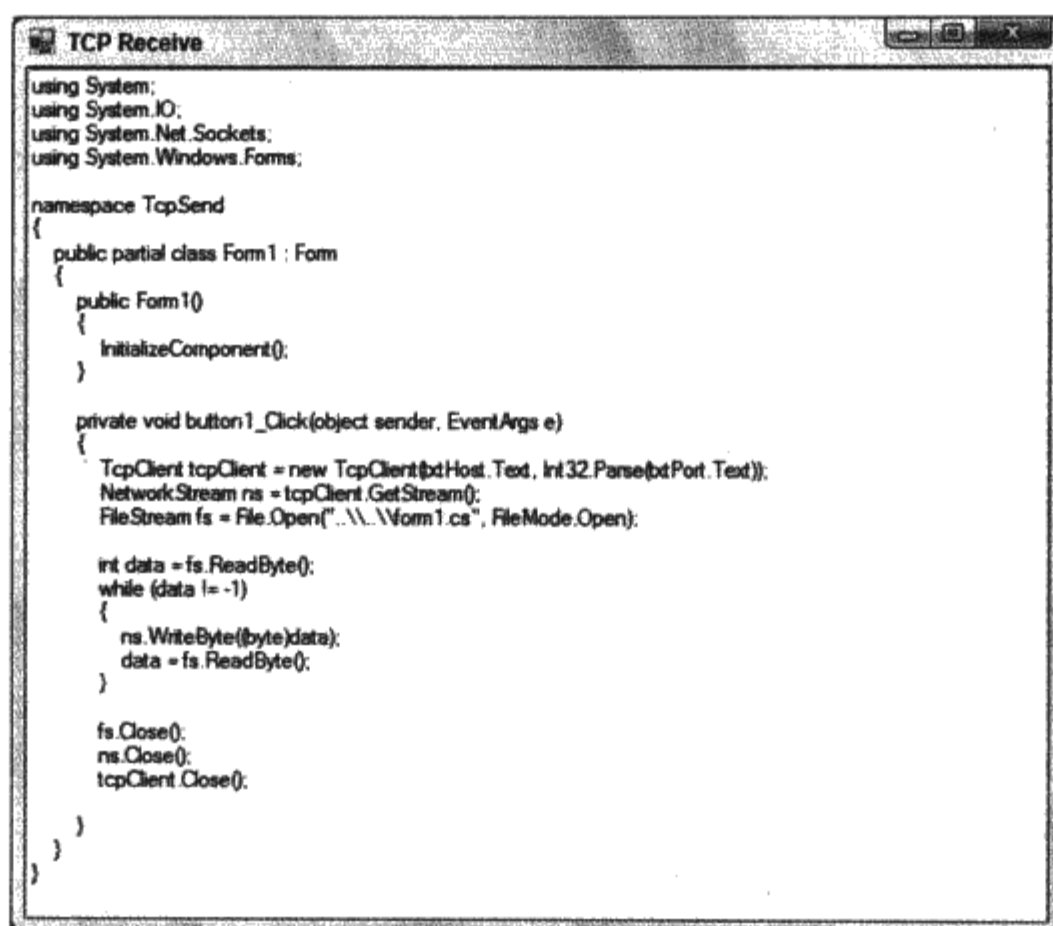


图 41-12

该窗体只包含一个 TextBox 控件 txtDisplay。TcpReceive 应用程序使用 TcpListener 等待进来的连接。为了避免应用程序界面的冻结，我们使用一个后台线程来等待，然后从连接中读取。因此还需要包含 System.Threading 命名空间：

```

using System;
using System.Net;
using System.IO;
using System.Net.Sockets;
using System.Windows.Forms;
using System.Threading;

```

在窗体的构造函数中，添加一个后台线程：

```

public Form1()
{
    InitializeComponent();

    Thread thread = new Thread(new ThreadStart(Listen));
    thread.Start();
}

```

其他重要的代码如下所示。

```

public void Listen()
{
    IPAddress localAddr = IPAddress.Parse("127.0.0.1");
    Int32 port = 2112;
    TcpListener tcpListener = new TcpListener(localAddr, port);
    tcpListener.Start();

    TcpClient tcpClient = tcpListener.AcceptTcpClient();
}

```



```

        NetworkStream ns = tcpClient.GetStream();
        StreamReader sr = new StreamReader(ns);
        string result = sr.ReadToEnd();
        Invoke(new UpdateDisplayDelegate(UpdateDisplay),
            new object[] {result} );

        tcpClient.Close();
        tcpListener.Stop();
    }

    public void UpdateDisplay(string text)
    {
        txtDisplay.Text= text;
    }

    protected delegate void UpdateDisplayDelegate(string text);

```

该线程在 Listen()方法中开始执行, 允许在不挂起界面的情况下对 AcceptTcpClient()进行调用。注意这里把 IP 地址 127.0.0.1 和端口号 2112 硬编码到应用程序中, 因此需要在客户应用程序中输入相同的端口号。

我们使用 AcceptTcpClient()返回的 TcpClient 对象打开一个新流, 进行读取。与本章前面的示例类似, 创建一个 StreamReader, 把进来的网络数据转换为字符串。在关闭客户机, 停止监听程序前, 更新窗体的文本框。我们不想从后台线程中直接访问文本框, 所以使用窗体的 Invoke()方法和一个委托, 把得到的字符串作为 object 参数数组的第一个元素来传送。Invoke()方法可确保调用正确编组到线程中, 以控制用户界面上的句柄。

#### 4. TCP 和 UDP

本节要介绍的另一个协议是 UDP(用户数据包协议)。UDP 是一个功能较少的简单协议, 但其开销也很小, 开发人员常常在速度和性能要求比可靠性更高的应用程序中使用 UDP, 例如视频流应用程序。相反, TCP 提供了许多功能来确保数据的传输, 它还提供了错误校正、当数据丢失或数据包损坏时重新传输它们的功能。最后, TCP 可缓存传入和传出的数据, 还保证在传输过程中, 在把数据包传送给应用程序之前, 重新编组杂乱的一系列数据包。即使有一些额外的开销, TCP 仍是在 Internet 上使用最广泛的协议, 因为它有非常高的可靠性。

#### 5. UDP 类

可以看出, 与 TcpClient 相比, UdpClient 类提供了一个较小、较简单的界面。这反映出 UDP 协议相对简单的本质。TCP 和 UDP 类都在后台使用套接字, 但 UdpClient 类不包含返回网络流以读写数据的方法。相反, 成员函数 Send()把一个字节数组作为参数, Receive()函数则返回一个字节数组。另外, 因为 UDP 是一个无连接的协议, 所以可以指定把通信的端点作为 Send() 和 Receive()方法的一个参数, 而不是在前面的构造函数或 Connect()方法中指定。也可以在某个后续的发送或接收过程中修改端点。

下面的代码段使用 UdpClient 类给回应服务(echo service)发送消息。带有回应服务的服务器在端口 7 处接收 TCP 或 UDP 连接。回应服务只把发送给服务器的数据再发送回客户机。这个服务可用于诊断和测试, 但许多系统管理员从安全的角度考虑, 不启用回应服务。

```

using System;
using System.Text;
using System.Net;
using System.Net.Sockets;
namespace Wrox.ProCSharp.InternetAccess.UdpExample
{
    class Class1
    {
        [STAThread]
        static void Main(string[] args)
        {
            UdpClient udpClient = new UdpClient();

            string sendMsg = "Hello Echo Server";
            byte [] sendBytes = Encoding.ASCII.GetBytes(sendMsg);

            udpClient.Send(sendBytes, sendBytes.Length, "SomeEchoServer.net", 7);

            IPEndPoint endPoint = new IPEndPoint(0,0);
            byte [] rcvBytes = udpClient.Receive(ref endPoint);
            string rcvMessage = Encoding.ASCII.GetString(rcvBytes,
                                                         0,
                                                         rcvBytes.Length);

            // should print out "Hello Echo Server"
            Console.WriteLine(rcvMessage);
        }
    }
}

```

Encoding.ASCII 类常常用于把字符串转换为字节数组，或把字节数组转换为字符串。还要注意，IPEndPoint 应按引用传送给 Receive() 方法。UDP 不是一个面向连接的协议，所以对 Receive() 的每次调用都会从不同的端点读取数据，Receive() 会用发送主机的 IP 地址和端口填充该参数。

UdpClient 和 TcpClient 在最低层的类 Socket 上提供了一个抽象层。

## 6. Socket 类

Socket 类提供了网络编程的最高级控制。说明该类的最简单方式是用 Socket 类重新编写 TcpReceive 应用程序。更新后的 Listen() 方法如下所示：

```

public void Listen()
{
    Socket listener = new Socket(AddressFamily.InterNetwork,
                                 SocketType.Stream,
                                 ProtocolType.Tcp);
    listener.Bind(new IPEndPoint(IPAddress.Any, 2112));
    listener.Listen(0);

    Socket socket = listener.Accept();
    Stream netStream = new NetworkStream(socket);
    StreamReader reader = new StreamReader(netStream);
}

```

```

        string result = reader.ReadToEnd();

        Invoke(new UpdateDisplayDelegate(UpdateDisplay),
            new object[] {result} );
        socket.Close();
        listener.Close();
    }

```

Socket 类需要再编写几行代码来完成相同的任务。对于初学者来说，构造函数的参数需要为使用 TCP 协议的流套接字指定 IP 寻址模式。这些参数只是可用于 Socket 类的许多组合中的一个，TcpClient 类会配置这些设置。接着把监听器的套接字绑定到一个端口上，开始监听传入的连接。当传入一个连接时，就可以使用 Accept() 方法创建一个新的套接字，来处理该连接。最后为套接字创建一个 StreamReader 实例，来读取传入的数据，其方式与前面的大致相同。

Socket 类也包含许多方法，用于异步接收、连接、发送和接收数据。使用这些方法和回调委托的方式与前面用 WebRequest 类请求异步页面的方式相同。如果确实需要了解套接字的内部情况，可以使用 GetSocketOption() 和 SetSocketOption() 方法，它们允许查看和配置各种选项，包括超时、生存期和其他低级选项。下面介绍另一个使用套接字的例子。

#### (1) 建立一个服务器控制台应用程序

为了进一步探讨 Socket 类，下一个例子创建一个控制台应用程序，作为传入套接字请求的服务器。之后创建第二个例子(另一个控制台应用程序)，它把一个消息传送给服务器控制台应用程序。

第一个应用程序是用作服务器的控制台应用程序，它会在指定的 TCP 端口上打开一个套接字，监听传入的消息。该控制台应用程序的代码如下：

```

using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

namespace SocketConsole
{
    class Program
    {
        static void Main()
        {
            Console.WriteLine("Starting: Creating Socket object");

            Socket listener = new Socket(AddressFamily.InterNetwork,
                SocketType.Stream,
                ProtocolType.Tcp);

            listener.Bind(new IPEndPoint(IPAddress.Any, 2112));
            listener.Listen(10);

            while (true)
            {
                Console.WriteLine("Waiting for connection on port 2112");

                Socket socket = listener.Accept();
                string receivedValue = string.Empty;

                while (true)
                {

```



```

        byte[] receivedBytes = new byte[1024];
        int numBytes = socket.Receive(receivedBytes);

        Console.WriteLine("Receiving ...");

        receivedValue += Encoding.ASCII.GetString(receivedBytes,
            0, numBytes);

        if (receivedValue.IndexOf("[FINAL]") > -1)
        {
            break;
        }
    }

    Console.WriteLine("Received value: {0}", receivedValue);

    string replyValue = "Message successfully received.";
    byte[] replyMessage = Encoding.ASCII.GetBytes(replyValue);

    socket.Send(replyMessage);
    socket.Shutdown(SocketShutdown.Both);

    socket.Close();
}

listener.Close();
}
}

```

这个例子使用 `Socket` 类建立了一个套接字。该套接字使用 TCP 协议，通过端口 2112 接收从任意 IP 地址传入的消息。通过打开的套接字接收到的值写入控制台屏幕。这个应用程序会继续接收字节，直到接收到 `[FINAL]` 字符串为止。这个 `[FINAL]` 字符串表示传入消息的末尾，之后就可以解释消息了。

从客户机上接收到消息的末尾后，就把一个回应消息传送给该客户机。之后，使用 `Close()` 方法关闭套接字，控制台应用程序继续等待接收新的消息。

## (2) 建立客户应用程序

下一步是建立一个客户应用程序，给第一个控制台应用程序发送消息。客户程序只要遵循某些建立好的规则，就可以给服务器控制台应用程序发送任意消息。第一个规则是服务器控制台应用程序只使用特定的协议。本例的服务器应用程序使用 TCP 协议监听消息。另一个规则是服务器应用程序只监听特定的端口，本例是端口 2112。最后一个规则是对于任意要发送的消息，最后必须以字符串 `[FINAL]` 结尾。

下面的客户控制台应用程序遵循这些规则：

```

using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

namespace SocketConsoleClient
{
    class Program
    {
        static void Main()
        {
            byte[] receivedBytes = new byte[1024];

```

```

IPHostEntry ipHost = Dns.Resolve("127.0.0.1");
IPAddress ipAddress = ipHost.AddressList[0];
IPEndPoint ipEndPoint = new IPEndPoint(ipAddress, 2112);

Console.WriteLine("Starting: Creating Socket object");

Socket sender = new Socket(AddressFamily.InterNetwork,
                           SocketType.Stream,
                           ProtocolType.Tcp);

sender.Connect(ipEndPoint);

Console.WriteLine("Successfully connected to {0}",
                  sender.RemoteEndPoint);

string sendingMessage = "Hello World Socket Test";
Console.WriteLine("Creating message: Hello World Socket Test");

byte[] forwardMessage = Encoding.ASCII.GetBytes(sendingMessage
          + "[FINAL]");

sender.Send(forwardMessage);
int totalBytesReceived = sender.Receive(receivedBytes);

Console.WriteLine("Message provided from server: {0}",
                  Encoding.ASCII.GetString(receivedBytes,
          0, totalBytesReceived));

sender.Shutdown(SocketShutdown.Both);
sender.Close();

Console.ReadLine();
}
}
}

```

在这个例子中，使用 localhost 的 IP 地址和服务器控制台应用程序要求的端口 2112 创建一个 IPEndPoint 对象。这里创建了一个套接字，调用了 Connect() 方法。打开套接字，连接到服务器控制台应用程序的套接字实例后，就使用 Send() 方法把一个文本字符串发送给服务器应用程序。由于服务器应用程序会返回一个消息，所以使用 Receive() 方法获取这个消息(把它放在一个字节数组中)。之后，将字节数组转换为一个字符串，显示在控制台应用程序上，最后关闭套接字。

运行这个应用程序，结果如图 41-13 所示。

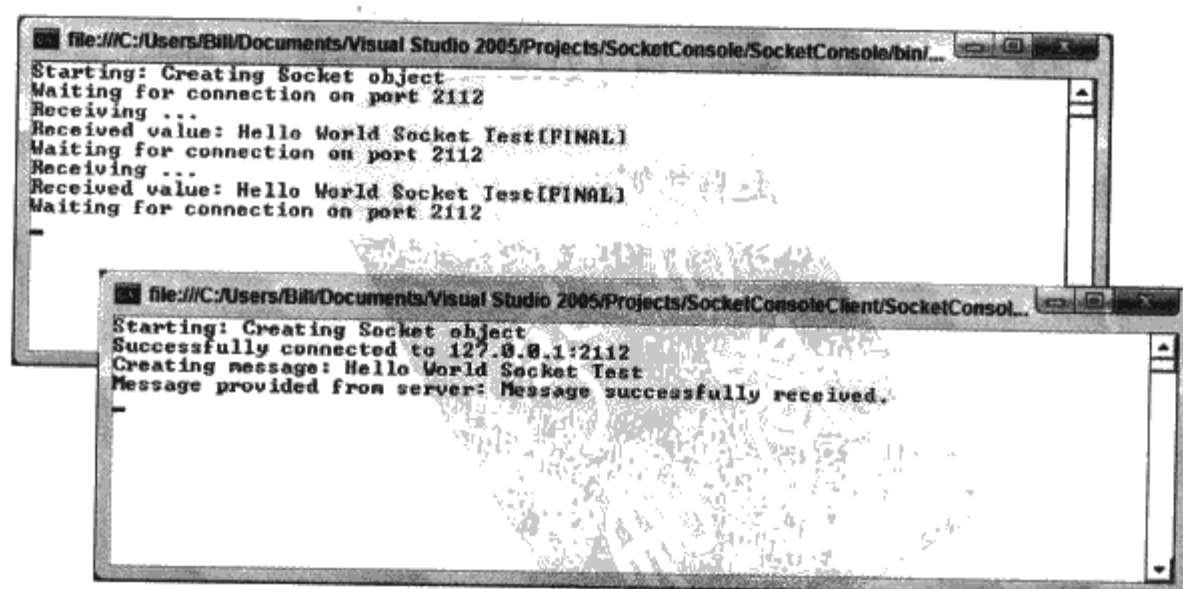


图 41-13



查看图 41-13 中的两个控制台应用程序，会发现服务器应用程序打开并等待传入的消息。传入的消息从客户程序中发送。接着，所发送的字符串由服务器应用程序显示。在接收和显示第一个消息后，服务器应用程序会等待其他消息的传入。关闭客户程序，然后再次运行它，就会看到这一点。接着会看到服务器应用程序再次显示接收到的消息。

## 41.6 小结

本章回顾了 System.Net 命名空间中用于网络通信的 .NET Framework 类。从中可了解到，哪些 .NET 基类可处理网络和 Internet 上打开的客户连接，如何给服务器发送请求和从服务器上接收响应，最常见的应用就是接收 HTML 页面。利用 .NET 3.5 中的 WebBrowser 控件，很容易在桌面应用程序中使用 Internet Explorer。

作为一般的规则，在使用 System.Net 命名空间中的类编程时，应使用最通用的类。例如，使用 TcpClient 类代替 Socket 类，可以把代码与许多低级套接字细节分离开来。更进一步，WebRequest 类允许利用 .NET Framework 中可插入的协议体系结构。代码应利用新的应用程序级协议，因为 Microsoft 和其他第三方引入了新功能。

最后，我们讨论了网络类中异步功能的使用，该功能给 Windows 窗体应用程序提供了用户响应界面的专业化外观。

下一章介绍 WCF。

# 第42章

## Windows Communication Foundation

在.NET 3.0 推出之前，一个企业解决方案需要好几个通信技术。对于独立于平台的通信，则使用 ASP.NET 的 Web 服务。对于比较高级的 Web 服务，可靠性、独立于平台的安全性和原子化事务处理、Web Services Enhancements 等技术给 ASP.NET Web 服务增加了复杂性。如果要求通信比较快，客户机和服务器都是.NET 应用程序，就应使用.NET Remoting。 .NET Enterprise Services 支持自动化的事务处理，它默认使用 DCOM 协议，比用.NET Remoting 快。DCOM 也是允许传送事务的唯一协议。所有这些技术都有不同的编程模型，都需要开发人员有许多技巧。

.NET Framework 3.0 提供了一个新的通信技术 WCF，它包含上述技术的所有特性，把它们合并到一个编程模型中。

本章讨论如下主题：

- WCF 概述
- 简单的服务和客户
- 合同
- 服务的实现
- 绑定
- 主机
- 客户机
- 双向通信

### 42.1 WCF 概述

WCF 合并了 ASP.NET Web 服务、.NET Remoting、消息队列和 Enterprise Services 的功能，WCF 的功能包括：

- 存储组件和服务：与联合使用定制主机、.NET Remoting 和 WSE 一样，也可以将 WCF 服务放在 ASP.NET 运行库、Windows 服务、COM+过程或 Windows 窗体应用程序中，进行对等计算。

- 声明操作：没有派生自基类的要求(这个要求存在于 .NET Remoting 和 Enterprise Services 中)，而可以使用属性定义服务。这类似于用 ASP.NET 开发的 Web 服务。
- 通信信道：在改变通信信道方面，.NET Remoting 非常灵活，WCF 也不错，因为它提供了相同的灵活性。WCF 提供了用 HTTP、TCP 和 IPC 信道进行通信的多个信道。也可以创建使用不同传输协议的定制信道。
- 安全架构：为了实现独立于平台的 Web 服务，必须使用标准化的安全环境。所提出的标准用 WSE 3.0 实现，这在 WCF 中被继承下来。
- 可扩展性：.NET Remoting 有丰富的扩展功能。它不仅能创建定制的信道、格式化标识符和代理对象，还能将功能插入客户机和服务器上的消息流。WCF 提供了类似的可扩展性。但是，WCF 的扩展性是用 SOAP 标题创建的。
- 支持以前的技术：要使用 WCF，根本不需要重写分布式解决方案，因为 WCF 可以与已有的技术集成起来。WCF 提供的信道使用 DCOM 与服务组件通信。也可以集成用 ASP.NET 开发的 Web 服务。

最终目标是通过进程或不同的系统、本地网络或 Internet 收发从客户机到服务的消息。如果需要以独立于平台的方式尽快收发消息，就应这么做。在远程视图上，服务提供了一个端点，它用合同、绑定和地址来描述。合同定义了服务提供的操作，绑定给出了协议和编码信息，地址是服务的位置。客户机需要一个兼容的端点来访问服务。

图 42-1 显示了参与 WCF 通信的组件。

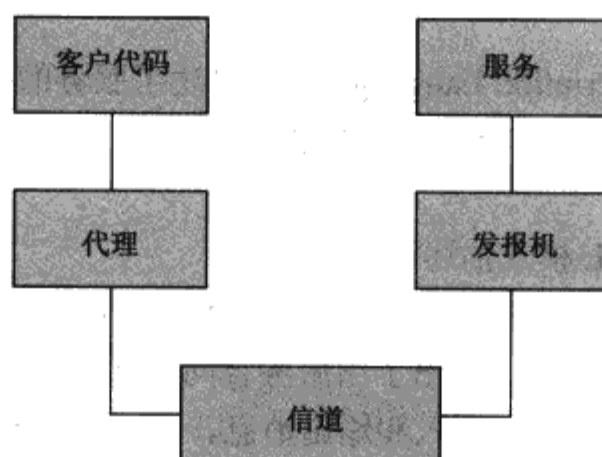


图 42-1

客户机调用代理上的一个方法。代理提供了与服务定义相同的方法，但把方法调用转换为一个消息，把该消息传送到信道上。信道有一个客户端部分和一个服务器端部分，它们通过一个联网协议来通信。在信道上，消息传送给发报机，发报机再把消息转换为用服务调用的方法。

WCF 支持几个通信协议。为了进行独立于平台的通信，需要支持 Web 服务标准。要在 .NET 应用程序之间通信，可以使用较快的通信协议，其开销较小。

下面几节介绍用于独立于平台的通信的核心服务的功能。

### 42.1.1 SOAP

为了进行独立于平台的通信，可以使用 SOAP 协议，它得到 WCF 的直接支持。SOAP 最初是 Simple Object Access Protocol 的缩写，但自从 SOAP 1.2 以来，就不再是这样了。SOAP 不再是一个对象访问协议，而可以发送用 XML 模式定义的消息。

服务从客户机中接收 SOAP 消息，返回一个 SOAP 响应消息。SOAP 消息包含封套，封套包含标题和消息体。

```
< s:Envelope xmlns:a="http://www.w3.org/2005/08/addressing"
  xmlns:s="http://www.w3.org/2003/05/soap-envelope" >
  < s:Header >
  < /s:Header >
  < s:Body >
    < ReserveRoom xmlns="http://www.wrox.com/ProCSharp/2008" >
      < roomReservation
        xmlns:d4pl="
          http://schemas.datacontract.org/2004/07/Wrox.ProCSharp.WCF"
        xmlns:i="http://www.w3.org/2001/XMLSchema-instance" >
        < d4pl:RoomName > Hawelka < /d4pl:RoomName >
        < d4pl:StartDate > 2007-06-21T08:00:00 < /d4pl:StartDate >
        < d4pl:EndDate > 2007-06-21T14:00:00 < /d4pl:EndDate >
        < d4pl:Contact > Georg Danzer < /d4pl:Contact >
        < d4pl:Event > White Horses < /d4pl:Event >
      < /roomReservation >
    < /ReserveRoom >
  < /s:Body >
< /s:Envelope >
```

标题是可选的，可以包含寻址、安全性和事务处理信息。消息体包含消息数据。

### 42.1.2 WSDL

WSDL(Web Services Description Language)文档描述了服务的操作和消息。WSDL 定义了服务的元数据，这些元数据可用于为客户应用程序创建代理。

WSDL 包含如下信息：

- 消息的类型：用 XML 模式描述。
- 从服务中收发的消息：消息的各部分是用 XML 模式定义的类型。
- 端口类型：映射服务合同，列出了用服务合同定义的操作。操作包含消息，例如，与请求和响应序列一起使用的输入和输出消息。
- 绑定信息：包含用端口类型列出的操作和用 SOAP 变体定义的操作
- 服务信息：把端口类型映射为端点地址。

**提示：**

在 WCF 中，WSDL 信息由 MEX(Metadata Exchange)端点提供。

### 42.1.3 JSON

除了发送 SOAP 消息之外，在 JavaScript 中访问服务最好使用 JSON(JavaScript Object Notation)。.NET 3.5 包含一个数据合同串行化器，可以用 JSON 记号创建对象。

JSON 的开销比 SOAP 小，因为它不是 XML，而是为 JavaScript 客户机进行了优化。这使之非常适用于 Ajax 客户机。Ajax 详见第 39 章。JSON 没有提供通过 SOAP 标题发送所具备的可靠性、安全性和事务处理特性，但这些通常是 JavaScript 客户机不需要的特性。

## 42.2 简单的服务和客户

在详细介绍 WCF 之前，首先看一个简单的服务。该服务用于预约会议室。

要存储会议室预约信息，应使用一个简单的 SQL Server 数据库，其中有一个表 RoomReservation。这个表及其属性如图 42-2 所示。可以下载这个数据库和本章的示例代码。

创建一个空白的解决方案 RoomReservation，在其中添加一个新的组件库项目 RoomReservationData。第一个实现的项目只包含访问数据库的代码。LINQ to SQL 使数据库访问代码非常简单，所以这里使用 .NET 3.5 技术。

提示：

LINQ to SQL 参见第 27 章。

添加一个新项，即 LINQ to SQL 类，命名为 RoomReservation.dbml。在 LINQ to SQL 设计器中，打开 Server Explorer，把 RoomReservation 数据库表拖放到设计器上，如图 42-3 所示。这个设计器会创建一个实用类 RoomReservation，它为表中的每一列和 RoomReservationDataContext 类包含属性。RoomReservationDataContext 连接到数据库上。

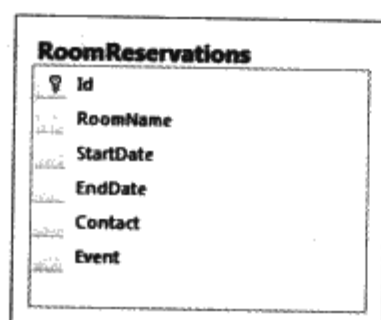


图 42-2

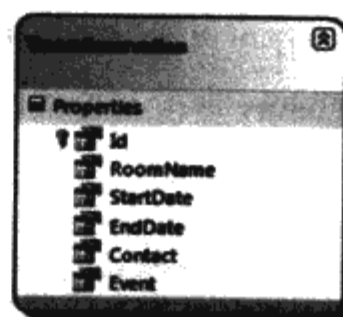


图 42-3

把 LINQ to SQL 设计器的 Serialization Mode 属性从 None 改为 Unidirectional。这样，生成的类 RoomReservation 会获得一个数据合同，以允许实用类通过 WCF 串行化。

要使用 LINQ to SQL 读写数据库中的数据，添加类 RoomReservationData。方法 ReserveRoom() 将一个会议室预约信息写入数据库。方法 GetReservations() 从指定的数据范围内返回一个会议室预约数组。

```
using System;
using System.Linq;
namespace Wrox.ProCSharp.WCF.Data
{
    public class RoomReservationData
    {
        public void ReserveRoom(RoomReservation roomReservation)
        {
            using (RoomReservationDataContext data =
                new RoomReservationDataContext())
            {
                data.RoomReservations.Add(roomReservation);
                data.SubmitChanges();
            }
        }
    }
}
```



```

public RoomReservation[] GetReservations(DateTime fromDate,
    DateTime toDate)
{
    using (RoomReservationDataContext data =
        new RoomReservationDataContext())
    {
        return (from r in data.RoomReservations
            where r.StartDate > fromDate & & r.EndDate < toDate
            select r).ToArray();
    }
}
}
}

```

现在开始创建服务。

### 42.2.1 服务合同

在解决方案中添加一个 WCF Service Library 类型的新项目，命名为 RoomReservationService。把生成的文件 IService1.cs 重命名为 IRoomService.cs，Service1.cs 重命名为 RoomReservationService.cs。把所生成的文件中的命名空间改为 Wrox.ProCSharp.WCF.Service。还需要引用程序集 RoomReservationData，使实用类型和 RoomReservationData 类可用。

服务提供的操作可以通过接口来定义。接口 IRoomService 定义了方法 ReserveRoom 和 GetRoomReservations。服务合同用 [ServiceContract] 属性定义。由服务定义的操作应用了属性 [OperationContract]。

```

using System;
using System.ServiceModel;

namespace Wrox.ProCSharp.WCF.Service
{
    [ServiceContract()]
    public interface IRoomService
    {
        [OperationContract]
        bool ReserveRoom(RoomReservation roomReservation);

        [OperationContract]
        RoomReservation[] GetRoomReservations(DateTime fromDate, DateTime toDate);
    }
}

```

### 42.2.2 服务的实现

服务类 RoomReservationService 实现了接口 IRoomService。实现服务时，只需调用类 RoomReservationData 的相应方法。

```

using System;
using System.ServiceModel;
using Wrox.ProCSharp.WCF.Data;
using Wrox.ProCSharp.WCF.Entities;

namespace Wrox.ProCSharp.WCF
{

```

```

public class RoomReservationService : IRoomService
{
    public bool ReserveRoom(RoomReservation roomReservation)
    {
        RoomReservationData data = new RoomReservationData();
        data.ReserveRoom(roomReservation);

        return true;
    }

    public RoomReservation[] GetRoomReservations(DateTime fromDate, DateTime toDate)
    {
        RoomReservationData data = new RoomReservationData();
        return data.GetReservations(fromDate, toDate);
    }
}

```

### 42.2.3 WCF 服务主机和 WCF 测试客户机

WCF Service Library 项目模板创建了一个应用程序配置文件 App.Config，它需要应用于新类和接口名。Service 元素引用了包含命名空间的服务类型 RoomReservationService，合同接口需要用 endpoint 元素定义。

```

< ?xml version="1.0" encoding="utf-8" ? >
< configuration >
  < system.serviceModel >
    < services >
      < service name="Wrox.ProCSharp.WCF.Services.RoomReservationService"
        behaviorConfiguration="RoomReservationsService.Service1Behavior" >
        < host >
          < baseAddresses >
            < add baseAddress =
              "http://localhost:8731/Design_Time_Addresses/RoomReservationService/" / >
          < /baseAddresses >
        < /host >
        <!-- Service Endpoints -->
        < endpoint address="" binding="wsHttpBinding"
          contract="Wrox.ProCSharp.WCF.Services.IRoomService" / >
        <!-- Metadata Endpoints -->
        < endpoint address="mex" binding="mexHttpBinding"
          contract="IMetadataExchange" / >
      < /service >
    < /services >
    < behaviors >
      < serviceBehaviors >
        < behavior name="RoomReservationsService.Service1Behavior" >
          < serviceMetadata httpGetEnabled="True" / >
          < serviceDebug includeExceptionDetailInFaults="False" / >
        < /behavior >
      < /serviceBehaviors >
    < /behaviors >
  < /system.serviceModel >
< /configuration >

```

提示:

服务地址 `http://localhost:8731/Design_Time_Addresses` 有一个关联的访问控制表(ACL), 它允许交互式用户创建一个监听端口。默认情况下, 非管理员用户不允许在监听模式下打开端口。使用命令行工具 `netsh http show urlacl` 可以查看 ACL, 用 `netsh http add url=http://+8080/ MyURI user=someUser` 添加新项。

在 VS2008 中启动这个库, 会启动 WCF 服务主机, 它显示为任务栏的注意区域中的一个图标。单击这个图标会打开如图 42-4 所示的对话框, 在其中可以查看服务的状态。项目属性定义了命令行参数/client: “WcfTestClient.exe”。 WCF 服务主机使用这个选项, 会启动 WCF 测试客户机, 如图 42-5 所示, 该测试客户机可用于测试应用程序。双击一个操作, 输入字段就会显示在应用程序的右边, 可以在其中填充要发送给服务的数据。单击 XML 选项卡, 可以看到已收发的 SOAP 消息。

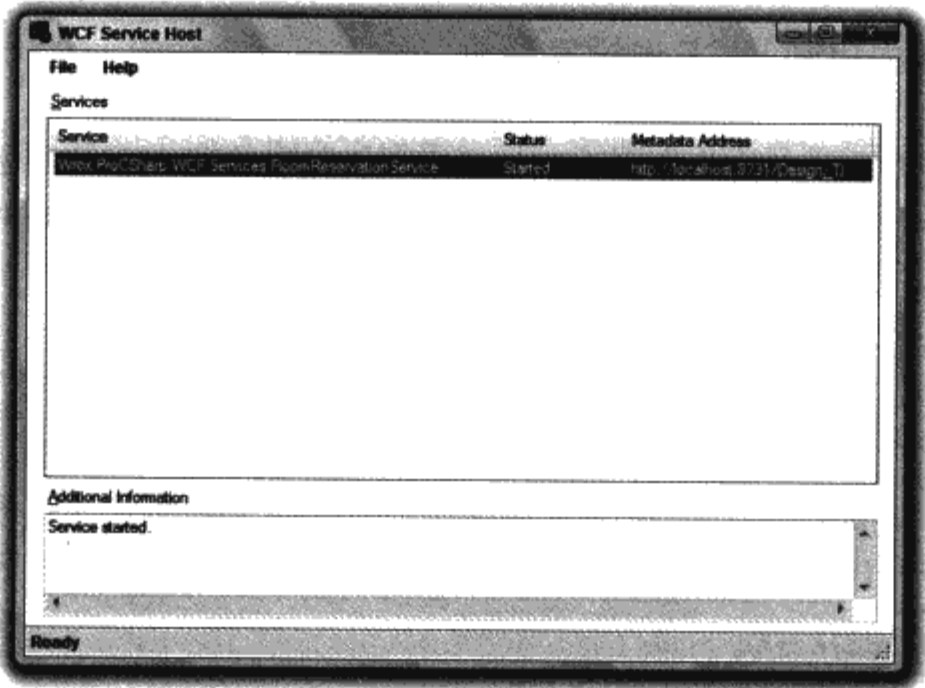


图 42-4

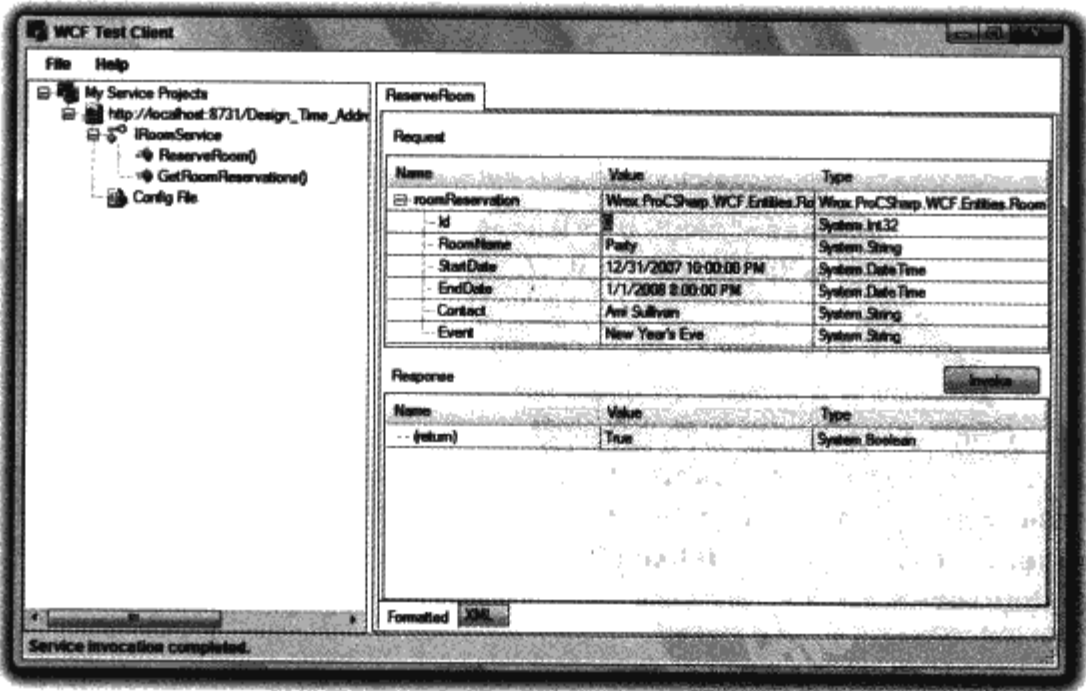


图 42-5

### 42.2.4 定制服务主机

使用 WCF 可以在任意主机上运行服务。可以为对等服务创建一个 Windows 窗体或 WPF 应用程序，创建一个 Windows 服务，或用 Windows Activation Services(WAS)实现该服务。控制台应用程序也适合于演示简单的主机。

使用服务主机必须引用 RoomReservationService 库。该服务从实例化和打开 ServiceHost 类型的对象开始。这个类在 System.ServiceModel 命名空间中定义。实现该服务的类 RoomReservationService 在构造函数中定义。调用 Open()方法会启动服务的监听器信道，该服务是用于监听请求的。Close()方法会停止信道。

```
using System;
using System.ServiceModel;
using Wrox.ProCSharp.WCF.Service;

namespace Wrox.ProCSharp.WCF
{
    class Program
    {
        internal static ServiceHost myServiceHost = null;

        internal static void StartService()
        {
            myServiceHost = new ServiceHost(typeof(RoomReservationService));

            myServiceHost.Open();
        }

        internal static void StopService()
        {
            if (myServiceHost.State != CommunicationState.Closed)
                myServiceHost.Close();
        }

        static void Main()
        {
            StartService();
            Console.WriteLine("Server is running. Press return to exit");
            Console.ReadLine();

            StopService();
        }
    }
}
```

对于 WCF 配置，需要把用服务库创建的应用程序配置文件复制到主机应用程序中。使用 WCF Service Configuration Editor 可以编辑这个配置文件，如图 42-6 所示。

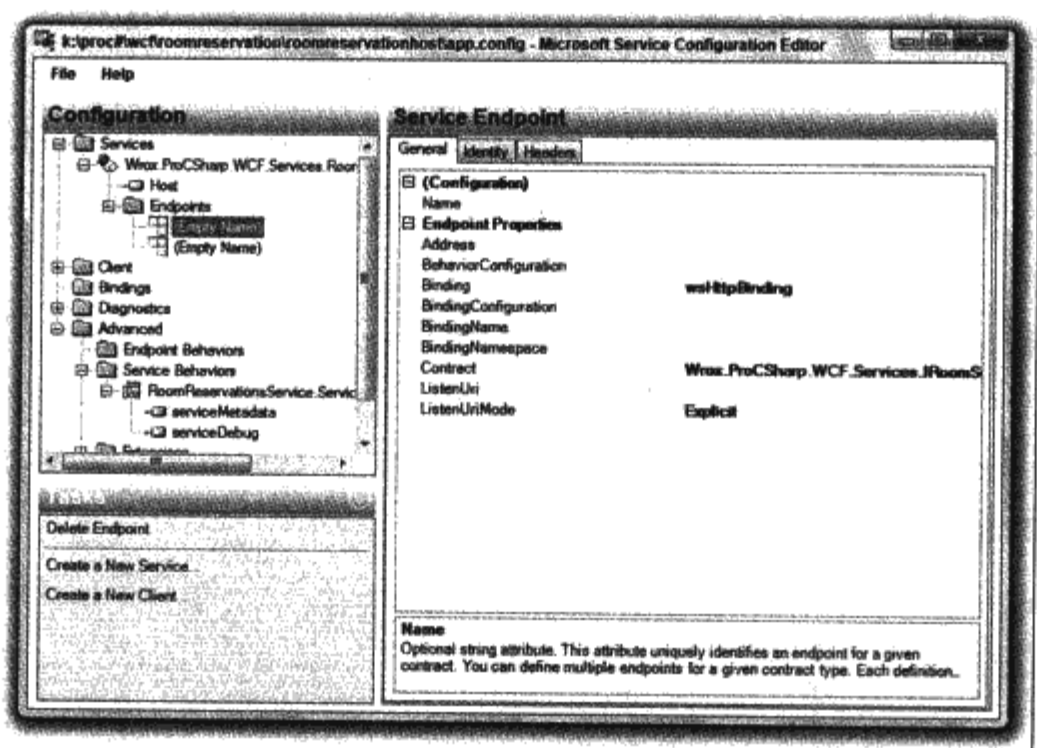


图 42-6

42.2.5 WCF 客户程序

对于客户程序，WCF 也可以灵活选择所使用的应用程序类型。客户程序可以是一个简单的控制台应用程序。但是，对于预约会议室，应创建一个包含控件的 Windows 窗体应用程序，如图 42-7 所示。

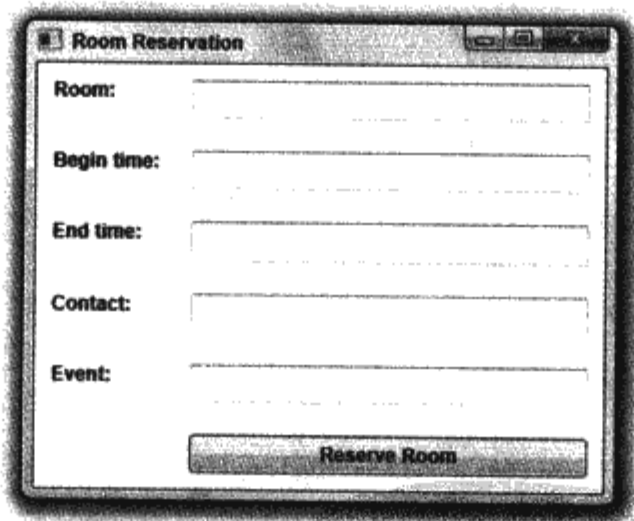


图 42-7

服务用绑定 mexHttpBinding 提供了一个 MEX 端点，元数据的访问是通过操作配置来启用的，所以可以从 Visual Studio 中添加一个服务引用。在添加服务引用时，会打开如图 42-8 所示的对话框。单击 Discover 按钮，可以在同一个解决方案中找到服务。

进入服务链接，将服务引用名设置为 RoomReservationService。服务引用名定义了所生成的代理类的命名空间。



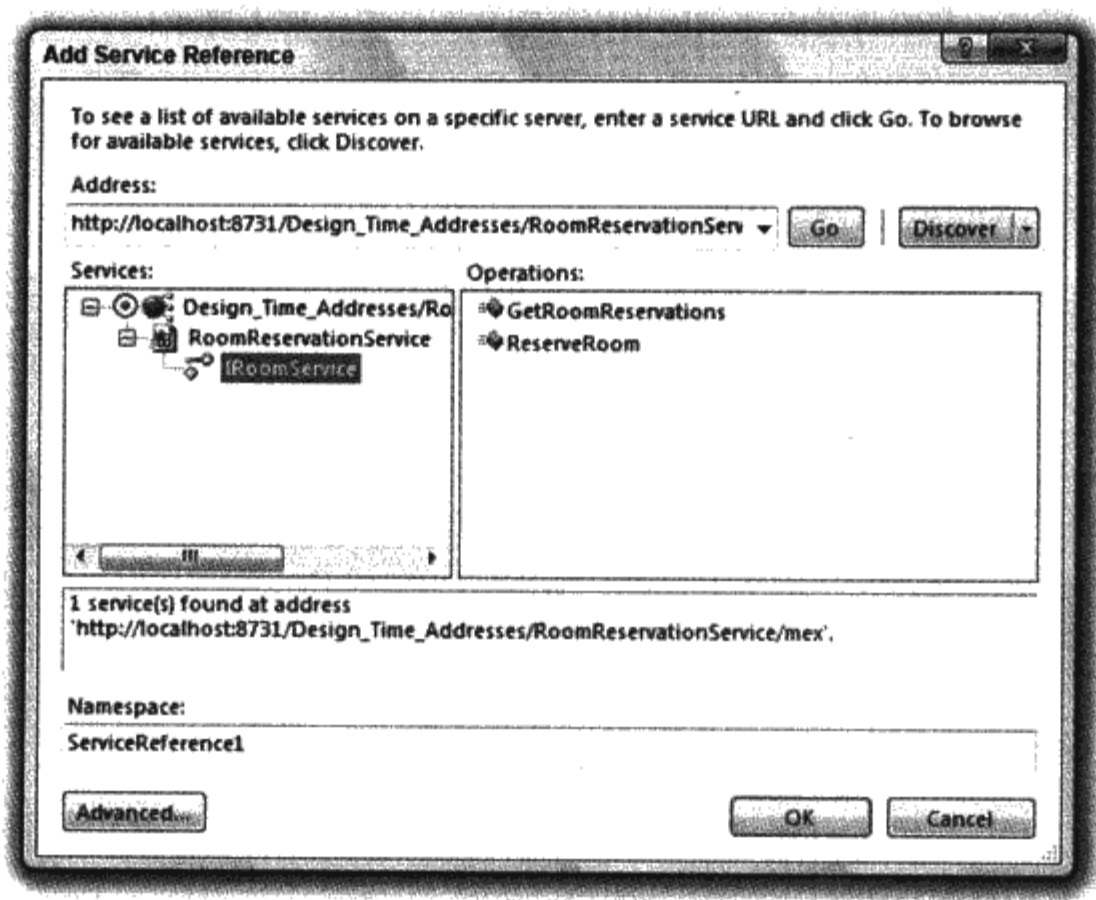


图 42-8

添加服务引用,会在服务中添加对程序集 `System.Runtime.Serialization` 和 `System.ServiceModel` 的引用,还会添加一个包含绑定信息和端点地址的配置文件。

在数据合同中,生成类 `RoomReservation`。这个类包含合同的所有 `[DataMember]` 元素。类 `RoomServiceClient` 是客户程序的代理,该客户程序包含由服务合同定义的方法。使用这个客户程序,可以将会议室预约信息发送给正在运行的服务。

```
private void OnReserveRoom(object sender, EventArgs e)
{
    RoomReservation reservation = new RoomReservation();
    reservation.Room = textRoom.Text;
    reservation.EventName = textEvent.Text;
    reservation.ContactName = textContact.Text;
    reservation.BeginDate = DateTime.Parse(textBeginTime.Text);
    reservation.EndDate = DateTime.Parse(textEndTime.Text);

    RoomServiceClient client = new RoomServiceClient();
    client.ReserveRoom(reservation);

    client.Close();
}
```

运行服务和客户程序,就可以将会议室预约信息添加到数据库中。

#### 42.2.6 诊断

运行客户和服务应用程序时,知道后台发生了什么非常有帮助。为此, WCF 使用一个需要配置的跟踪源。可以使用 `Service Configuration Editor`, 选择 `Diagnostics`, 启动 `Tracing and Message Logging`, 来配置跟踪。把跟踪源的跟踪级别设置为 `Verbose` 会生成非常详细的信息。

这个配置更改把跟踪源和监听器添加到应用程序配置文件中，如下所示：

```
< system.diagnostics >
  < sources >
    < source name="System.ServiceModel" switchValue="
      Verbose,ActivityTracing"propagateActivity="true" >
      < listeners >
        < add type="System.Diagnostics.DefaultTraceListener" name="Default" >
          < filter type="" / >
        < /add >
        < add name="ServiceModelTraceListener" >
          < filter type="" / >
        < /add >
      < /listeners >
    < /source >
    < source name="System.ServiceModel.MessageLogging"
      switchValue="Verbose,ActivityTracing" >
      < listeners >
        < add type="System.Diagnostics.DefaultTraceListener" name="Default" >
          < filter type="" / >
        < /add >
        < add name="ServiceModelMessageLoggingListener" >
          < filter type="" / >
        < /add >
      < /listeners >
    < /source >
  < /sources >
  < sharedListeners >
    < add initializeData="c:\logs\app_tracelog.svclog"
      type="System.Diagnostics.XmlWriterTraceListener, System,
      Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
      name="ServiceModelTraceListener" traceOutputOptions="Timestamp" >
    < filter type="" / >
    < /add >
    < add initializeData="c:\logs\app_messages.svclog"
      type="System.Diagnostics.XmlWriterTraceListener, System,
      Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
      name="ServiceModelMessageLoggingListener"
      traceOutputOptions="Timestamp" >
    < filter type="" / >
    < /add >
  < /sharedListeners >
< /system.diagnostics >
< system.serviceModel >
< diagnostics >
  < messageLogging logEntireMessage="true" logMalformedMessages="true"
    logMessagesAtServiceLevel="true" logMessagesAtTransportLevel="true" / >
< /diagnostics >
<!-- ... -->
```

提示:

WCF 类的执行代码使用跟踪源 `System.ServiceModel` 和 `System.ServiceModel.MessageLogging` 来写入跟踪消息。跟踪和配置跟踪源及监听器的更多内容详见第 18 章。

启动应用程序时,使用 `verbose` 跟踪设置的跟踪文件会很快变得很大。为了分析 XML 日志文件中的信息, .NET SDK 包含一个 `Service Trace Viewer` 工具 `svctraceviewer.exe`。图 42-9 显示了这个工具选择跟踪和消息日志文件后的视图。在默认配置下,可以看到互换了几个消息,其中的许多消息都与安全性相关。根据安全性需求,可以选择其他配置选项。

下面详细介绍 WCF 的细节和不同的选项。

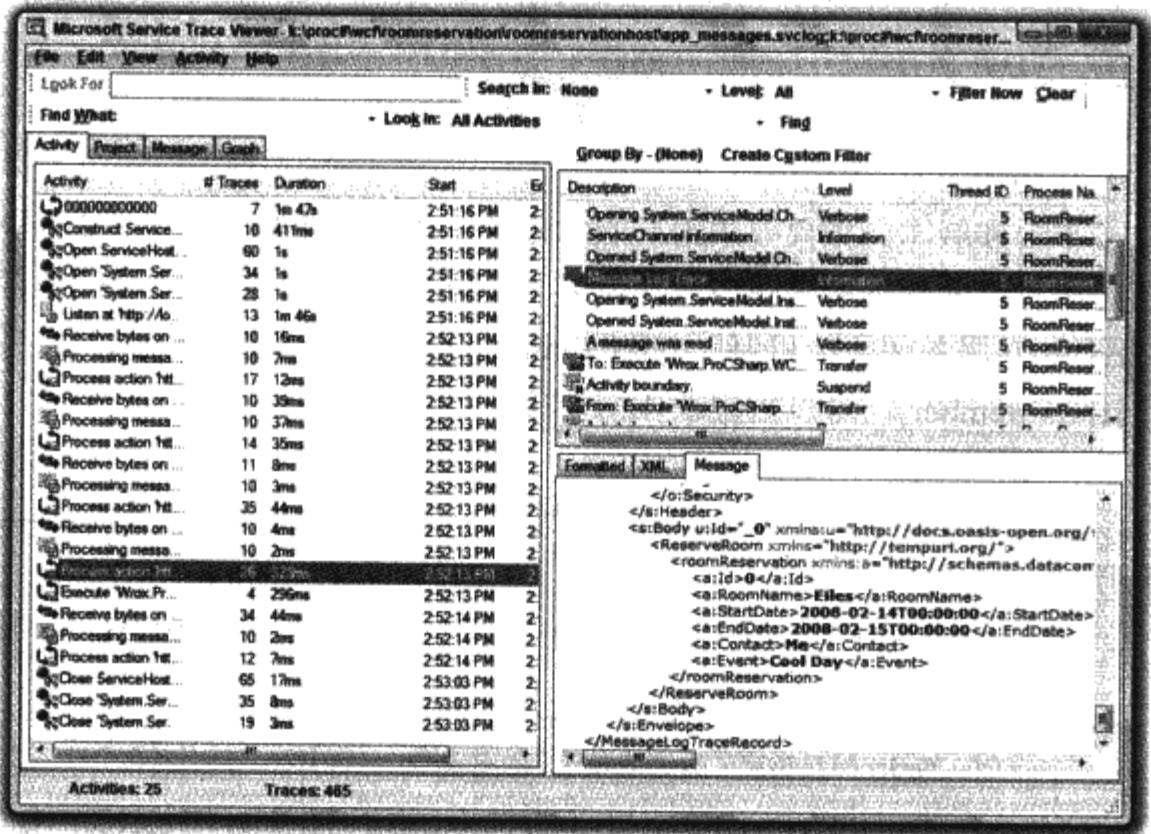


图 42-9

### 42.3 合同

合同定义了服务提供的功能和客户程序可以使用的功能。合同可以完全独立于服务的实现代码。

由 WCF 定义合同可以分为三种不同的类型: 数据合同、服务合同和消息合同。合同可以用 .NET 属性来指定:

- 数据合同: 数据合同定义了从服务中接收和返回的数据。用于收发消息的类关联了数据合同属性。
- 服务合同: 服务合同用于定义描述了服务的 WSDL。这个合同用接口或类定义。
- 消息合同: 如果需要完全控制 SOAP 消息, 消息合同就可以指定应放在 SOAP 标题中的数据以及放在 SOAP 主体中的数据。

下面详细介绍这些合同。

42.3.1 数据合同

在数据合同中，CLR 类型映射为 XML 模式。数据合同不同于其他.NET 串行化机制。在运行时串行化中，所有的字段都会串行化(包括私有字段)，而在 XML 串行化中，只串行化公共字段和属性。数据合同要求用[DataMember]属性明确标记要串行化的字段。无论字段是私有或公共的，还是应用于特性，都可以使用这个属性。

```
[DataContract(Namespace="http://www.thinktecture.com")]
public class RoomReservation
{
    [DataMember] public string Room;
    [DataMember] public DateTime StartDate;
    [DataMember] public DateTime EndDate;
    [DataMember] public string ContactName;
    [DataMember] public string EventName;
}
```

为了独立于平台和版本，如果要求用新版本修改数据，且不破坏旧客户程序和服务，使用数据合同是指定要传送的数据的最佳方式。还可以使用 XML 串行化和运行时串行化。XML 串行化是 ASP.NET Web 服务使用的机制，.NET Remoting 使用运行时串行化。

使用属性[DataMember]，可以指定表 42-1 中的特性。

表 42-1

[DataMember]特性	说 明
Name	串行化元素的名称默认与应用了属性[DataMember]的字段或特性名相同。使用 Name 特性可以修改该名称
Order	Order 特性指定了数据成员的串行化顺序
IsRequired	使用 IsRequired 特性，可以指定元素必须经过串行化，才能接收。这个特性可以用于解决版本问题  如果在已有的合同中添加了成员，合同不会被破坏，因为在默认情况下字段是可选的 (IsRequired=false)。将 IsRequired 设置为 true，就可以破坏已有的合同
EmitDefaultValue	EmitDefaultValue 指定有默认值的成员是否应串行化。如果 EmitDefaultValue 设置为 true，具有该类型默认值的成员就不串行化

42.3.2 版本问题

创建数据合同的新版本时，注意更改的种类，如果应同时支持新旧客户机和新旧服务，就应执行相应的操作。

在定义合同时，应使用 DataContractAttribute 的 Namespace 属性添加 XML 命名空间信息。如果创建了数据合同的新版本，破坏了兼容性，就应改变这个命名空间。如果只添加了可选的成员，合同就没有被破坏——这就是一个可兼容的改变。旧客户机仍可以给新服务发送消息，因为不需要其他数据。新客户机可以给旧服务发送消息，因为旧服务仅忽略额外的数据。

删除字段或添加需要的字段会破坏合同。此时还应改变 XML 命名空间。命名空间的名称可以包含年份和月份，例如 `http://thinktecture.com/SampleServices/2008/02`。每次做了破坏性的修改时，都要改变命名空间，例如把年份和月份改为实际值。

42.3.3 服务合同

服务合同定义了服务可以执行的操作。属性[ServiceContract]与接口或类一起使用，来定义服务合同。由服务提供的方法通过接口 `IRoomService` 应用了属性[OperationContract]，如下所示：

```
[ServiceContract]
public interface IRoomService
{
    [OperationContract]
    bool ReserveRoom(RoomReservation roomReservation);
}
```

用属性[ServiceContract]设置的特性如表 42-2 所示。

表 42-2

ServiceContract 特性	说 明
ConfigurationName	这个特性定义了配置文件中服务配置的名称
CallbackContract	当服务用于双向消息传送时，特性 CallbackContract 定义了是客户程序中实现的合同
Name	这个特性定义了 WSDL 中<portType>元素的名称
Namespace	Namespace 特性定义了 WSDL 中<portType>元素的 XML 命名空间
SessionMode	使用 SessionMode 特性，可以定义调用这个合同的操作所需的会话。其值用 SessionMode 枚举定义，包括 Allowed、NotAllowed 和 Required
ProtectionLevel	ProtectionLevel 特性确定了绑定是否必须能保护通信。其值用 ProtectionLevel 枚举定义，包括 None、Sign、EntryptAndSign

使用[OperationContract]可以定义如表 42-3 所示的特性。

表 42-3

OperationContract 特性	说 明
Action	WCF 使用 SOAP 请求的 Action 特性，把该请求映射到相应的方法上。Action 的默认值是合同 XML 命名空间、合同名和操作名的组合。如果是一个响应消息，就把 Response 添加到 Action 字符串中。指定 Action 特性可以重写 Action 值。如果指定值 “*”，服务操作就会处理所有的消息
ReplyAction	Action 设置了入站 SOAP 请求的 Action 名，而 ReplyAction 设置了回应消息的 Action 名
AsyncPattern	如果使用异步模式来实现操作，就把 AsyncPattern 特性设置为 true。异步模式详见第 19 章



(续表)

OperationContract 特性	说 明
IsInitiating IsTerminating	如果合同由一系列操作组成，且初始化操作指定 IsInitiating 特性，该系列的最后一个操作就需要 IsTerminating 特性。初始化操作启动一个新会话，服务器用中止操作来关闭会话
IsOneWay	设置 IsOneWay 特性，客户程序就不会等待回应消息。在发送请求消息后，单向操作的调用者无法直接检测失败
Name	操作的默认名称是指定了操作合同的方法名。使用 Name 特性可以修改该操作的名称
ProtectionLevel	使用这个特性可以确定消息是应只签名，还是应加密后签名

在服务合同中，也可以用属性 [DeliveryRequirements] 定义服务的传输要求。属性 RequireOrderedDelivery 指定所传送的消息必须以相同的顺序到达。使用属性 Queued-DeliveryRequirements 可以指定，消息以断开连接的方式传送，例如使用消息队列(参见第 45 章)。

42.3.4 消息合同

如果需要完全控制 SOAP 消息，就可以使用消息合同。在消息合同中，可以指定消息的哪些部分要放在 SOAP 标题中，哪些部分要放在 SOAP 主体中。下面的例子显示了 ProcessPersonRequestMessage 类的一个消息合同。该消息合同用属性 [MessageContract] 指定。SOAP 消息的标题和主体用属性 [MessageHeader] 和 [MessageBodyMember] 指定。指定 Position 属性，可以确定主体中的元素顺序。还可以为标题和主体字段指定保护级别。

```
[MessageContract]
public class ProcessPersonRequestMessage
{
    [MessageHeader]
    public int employeeId;

    [MessageBodyMember(Position=0)]
    public Person person;
}
```

ProcessPersonRequestMessage 类与用 IProcessPerson 接口定义的服务合同一起使用：

```
[ServiceContract]
public interface IProcessPerson
{
    [OperationContract]
    public PersonResponseMessage ProcessPerson(
        ProcessPersonRequestMessage message);
}
```

42.4 服务的实现

服务的实现代码用属性 [ServiceBehavior] 标记，如下面的 RoomReservationService 类所示：

```
[ServiceBehavior]
public class RoomReservationService : IRoomService
{
    public bool ReserveRoom(RoomReservation roomReservation)
    {
        // implementation
    }
}
```

属性[ServiceBehavior]用于描述 WCF 服务提供的操作，以获取所需功能的代码，如表 42-4 所示。

表 42-4

ServiceBehavior 特性	说 明
TransactionAutoComplete- OnSessionClose	当前会话正确完成时，就自动提交该事务。这类似于第 44 章讨论的 Enterprise Services 中的属性[AutoComplete]
TransactionIsolationLevel	要定义服务中事务处理的隔离级别，可以把属性 TransactionIsolation Level 设置为 IsolationLevel 枚举中的一个值。事务处理的隔离级别详见第 22 章
ReleaseServiceInstanceOn- TransactionComplete	完成了事务处理后，服务的实例可循环使用
AutomaticSessionShutdown	如果在客户关闭连接时没有关闭会话，就可以把属性 AutomaticSessionShutdown 设置为 false。在默认情况下，会关闭会话
InstanceContextMode	使用 InstanceContextMode 属性，可以确定应使用有状态的对象还是无状态的对象。默认设置为 InstanceContextMode.PerCall，为每个方法调用创建一个新对象。可以将它与.NET Remoting 中著名的 SingleCall 对象比较。其他设置有 PerSession 和 Single。这两个设置都使用有状态的对象。但是，PerSession 会为每个客户创建一个新对象，而 Single 允许在多个客户上共享同一个对象
ConcurrencyMode	因为有状态的对象可以由多个客户(或同一个客户的多个线程)使用，所以必须注意这种对象类型的并发问题。如果属性 ConcurrencyMode 设置为 Multiple，多个线程就可以访问对象，但必须处理同步问题。如果把该属性设置为 Single，一次就只有一个线程能访问对象，但不必处理同步问题，如果客户较多，可能出现可伸缩性问题。值 Reentrant 表示只有从调用返回的线程可以访问对象。对于无状态的对象，这个设置没有任何意义，因为每个方法调用都会实例化一个新对象，不共享状态
UseSynchronizationContext	Windows 窗体和 WPF 控件的成员都只能从创建线程中调用。如果服务位于 Windows 应用程序中，其方法调用了控件成员，就把 UseSynchronizationContext 设置为 true。这样，服务就运行在 SynchronizationContext 定义的线程中

(续表)

ServiceBehavior 特性	说 明
IncludeExceptionDetailInFaults	在.NET 中，错误被看作异常。SOAP 指定，SOAP 错误返回给客户，以防服务器出问题。出于安全考虑，最好不要把服务器端的异常细节返回给客户。因此，异常默认转换为未知错误。要返回特定的错误，可抛出 FaultException 类型的异常  为了便于调试，返回真实的异常信息是很有帮助的。此时应把 IncludeExceptionDetailInFaults 的设置改为 true，抛出 FaultException <TDetail>异常，其中原始异常包含了详细信息
MaxItemsInObjectGraph	使用 MaxItemsInObjectGraph 特性，可以限制要串行化的对象数
ValidateMustUnderstand	特性 ValidateMustUnderstand 设置为 true，表示必须理解 SOAP 标题(默认)

为了演示服务的操作，接口 IStateService 定义了一个服务合同，其中的两个操作用于获取和设置状态。有状态的服务合同需要一个会话。这就是把服务合同的 SessionMode 属性设置为 SessionMode.Required 的原因。服务合同还将 IsInitiating 和 IsTerminating 属性应用于操作合同，定义了启动和关闭会话的方法。

```
[ServiceContract(SessionMode=SessionMode.Required)]
public interface IStateService
{
    [OperationContract(IsInitiating=true)]
    void Init(int i);

    [OperationContract]
    void SetState(int i);

    [OperationContract]
    int GetState();

    [OperationContract(IsTerminating=true)]
    void Close();
}
```

服务合同由类 StateService 实现。服务的实现代码定义了 InstanceContextMode.Per-Session，使状态与实例保持同步。

```
[ServiceBehavior(InstanceContextMode=InstanceContextMode.PerSession)]
public class StateService : IStateService
{
    int i = 0;

    public void Init(int i)
    {
        this.i = i;
    }

    public void SetState(int i)
    {
        this.i = i;
    }
}
```

```

public int GetState()
{
    return i;
}

public void Close()
{
}
}

```

现在必须定义对地址和协议的绑定。其中，将 `basicHttpBinding` 赋予服务的端点：

```

<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      < services >
        < service behaviorConfiguration="StateServiceSample.Service1Behavior"
          name="Wrox.ProCSharp.WCF.StateService" >
          < endpoint address="" binding="basicHttpBinding"
            bindingConfiguration=""
            contract="Wrox.ProCSharp.WCF.IStateService" >
          < /endpoint >
          < endpoint address="mex" binding="mexHttpBinding"
            contract="IMetadataExchange" / >
        < /service >
      < /services >
      < host >
        < baseAddresses >
          < add baseAddress="http://localhost:8731/Design_Time_Addresses/
            StateServiceSample/Service1/" / >
        < /baseAddresses >
      < /host >
    < /services >
    < behaviors >
      < serviceBehaviors >
        < behavior name="StateServiceSample.Service1Behavior" >
          < serviceMetadata httpGetEnabled="True" / >
          < serviceDebug includeExceptionDetailInFaults="False" / >
        < /behavior >
      < /serviceBehaviors >
    < /behaviors >
  < /system.serviceModel>
</configuration>

```

如果用定义好的配置启动服务主机，就会抛出一个 `InvalidOperationException` 类型的异常。该异常的错误消息是“合同需要会话，但绑定 `BasicHttpBinding` 不支持它，或者没有正确配置为支持它。”

并不是所有的绑定都支持所有的服务。服务合同需要用 `[ServiceContract(ServiceMode=ServiceMode.Required)]` 指定一个会话，主机会失败，是因为所配置的绑定不支持会话。

只要修改对绑定的配置，使之支持会话(例如 `wsHttpBinding`)，服务器就会成功启动。

```

< endpoint address="" binding="wsHttpBinding"
  bindingConfiguration=""
  contract="Wrox.ProCSharp.WCF.IStateService" >
< /endpoint >

```

现在可以创建客户应用程序了。在前面的例子中，客户应用程序是通过添加一个服务引用

来创建的。除了添加服务引用之外，还可以直接访问包含合同接口的程序集，使用类 `ChannelFactory<TChannel>` 实例化连接服务的信道。

类 `ChannelFactory<TChannel>` 的构造函数接收绑定配置和端点地址这两个参数。绑定必须与用服务主机创建的绑定兼容，用 `EndpointAddress` 类定义的地址引用所运行的服务的 URI。

`CreateChannel()` 方法创建了一个连接服务的信道。接着调用服务的方法，可以看出，该服务实例是有状态的，直到调用指定了 `IsTerminating` 属性操作的 `Close()` 方法为止。

```
using System;
using System.ServiceModel;
namespace Wrox.ProCSharp.WCF
{
    class Program
    {
        static void Main()
        {
            WSHttpBinding binding = new WSHttpBinding();
            EndpointAddress address =
                new EndpointAddress("http://localhost:8731/" +
                    !Design_Time_Addresses/StateServiceSample/Service1/");
            ChannelFactory < IStateService > factory =
                new ChannelFactory < IStateService > (binding, address);

            IStateService channel = factory.CreateChannel();
            channel.Init(1);
            Console.WriteLine(channel.GetState());
            channel.SetState(2);
            Console.WriteLine(channel.GetState());
            channel.Close();

            factory.Close();
        }
    }
}
```

在服务的实现代码中，可以通过属性 `[OperationBehavior]` 将服务方法应用于如表 42-5 所示的特性。

表 42-5

OperationBehavior	说 明
AutoDisposeParameters	默认情况下，所有可删除的参数都自动删除。如果参数不应删除，就可以把 <code>AutoDisposeParameters</code> 特性设置为 <code>false</code> 。接着，发送者将负责删除该参数
Impersonation	使用 <code>Impersonation</code> 特性，可以模拟调用者，以调用者的身份运行方法
ReleaseInstanceMode	<code>InstanceContextMode</code> 使用服务操作设置定义了对象实例的生存期。使用操作行为设置，可以根据操作重写设置。 <code>ReleaseInstanceMode</code> 用 <code>ReleaseInstanceMode</code> 枚举定义了实例发布模式。其值 <code>None</code> 使用 <code>InstanceContextMode</code> 设置。值 <code>BeforeCall</code> 、 <code>AfterCall</code> 和 <code>BeforeAnd- AfterCall</code> 定义了操作的循环时间
TransactionScopeRequired	使用 <code>TransactionScopeRequired</code> 特性可以指定操作是否需要一个事务处理。如果需要，且调用者已经发出了一个事务处理，就使用这个事务处理。如果调用者没有发出事务处理，就创建一个新的事务处理



(续表)

OperationBehavior	说 明
TransactionAutoComplete	TransactionAutoComplete 特性指定事务处理是否自动完成。如果该特性设置为 true，在抛出异常的情况下就中止事务处理。如果这是一个根事务处理，且没有抛出异常，就提交事务处理

错误处理

默认情况下，在服务中出现的详细的异常消息不返回给客户应用程序。其原因是安全性。不应把详细的异常消息提供给使用服务的第三方。异常应记录到服务上(为此可以使用跟踪和事件日志功能)，包含有用信息的错误应返回给调用者。

可以抛出一个 `FaultException`，来返回 SOAP 错误。抛出 `FaultException` 会创建一个未类型化的 SOAP 错误。返回错误的首选方式是生成强类型化的 SOAP 错误。

应与强类型化的 SOAP 错误一起传送的信息用数据合同定义，如下面的 `StateFault` 类所示：

```
[DataContract]
public class StateFault
{
    [DataMember]
    public int BadState { get; set; }
}
```

SOAP 错误的类型必须用 `FaultContractAttribute` 和操作合同定义：

```
[FaultContract(typeof(StateFault))]
[OperationContract]
void SetState(int i);
```

在实现代码中，抛出了一个 `FaultException<TDetail>` 异常。在构造函数中，可以指定一个新的 `TDetail` 对象，在本例中就是 `StateFault`。另外，`FaultReason` 中的错误信息可以赋予构造函数。`FaultReason` 支持多种语言的错误信息。

```
public void SetState(int i)
{
    if (i == -1)
    {
        FaultReasonText[] text = new FaultReasonText[2];
        text[0] = new FaultReasonText("Sample Error",
            new CultureInfo("en"));
        text[1] = new FaultReasonText("Beispiel Fehler",
            new CultureInfo("de"));
        FaultReason reason = new FaultReason(text);

        throw new FaultException < StateFault > (
            new StateFault() { BadState = i }, reason);
    }
    else
    {
        this.i = i;
    }
}
```

在客户应用程序中，可以捕获 `FaultException<StateFault>` 类型的异常。出现该异常的原因由 `Message` 属性定义。`StateFault` 用 `Detail` 属性访问。

```
try
{
    channel.SetState(-1);
}
catch (FaultException < StateFault > ex)
{
    Console.WriteLine(ex.Message);
    StateFault detail = ex.Detail;
    Console.WriteLine(detail.BadState);
}
```

除了捕获强类型化的 SOAP 错误之外，客户应用程序还可以捕获 `FaultException- <Detail>` 的基类 `FaultException` 和 `CommunicationException` 的异常。通过 `Communication- Exception` 还可以捕获与 WCF 通信相关的其他异常。

42.5 绑定

绑定描述了服务的通信方式。使用绑定可以指定如下特性：

- 传输协议
- 安全要求
- 编码格式
- 事务处理要求
- 可靠性
- 形状变化
- 传输升级

绑定包含多个绑定元素，它们描述了所有绑定要求。可以创建定制的绑定，也可以使用表 42-6 中的预定义绑定：

表 42-6

标 准 绑 定	说 明
BasicHttpBinding	BasicHttpBinding 用于最广泛的交互操作的第一代 Web 服务。所使用的传输协议是 HTTP 或 HTTPS，其安全性仅由协议保证
WSHttpBinding	WSHttpBinding 用于下一代 Web 服务、用 SOAP 扩展确保安全、可靠性和事务处理的平台。所使用的传输协议是 HTTP 或 HTTPS，为了确保安全，使用了 WS-Security。使用 WS-Coordination、WS-AtomicTransaction 和 WS-Business Activity 规范支持事务处理，通过 WS-ReliableMessaging 支持可靠的消息传送。WSProfile 也支持用于发送附件的 MTOM 编码。WS-*标准的规范可访问 <a href="http://www.oasis-open.org">http://www.oasis-open.org</a>
WS2007HttpBinding	WS2007HttpBinding 派生于基类 WSHttpBinding，支持 OASIS 定义的安全性、可靠性和事务处理规范。这个类是 .NET 3.0 SP1 中新增的

(续表)

标 准 绑 定	说 明
WSHttpContextBinding	WSHttpContextBinding 派生于基类 WSHttpBinding, 支持没有使用 cookie 的环境。这个绑定会添加 ContextBindingElement, 交换环境信息
WebHttpBinding	这个绑定用于通过 HTTP 请求(而不是 SOAP 请求)提供的服务, 它可以用于脚本客户程序, 例如 ASP.NET AJAX
WSFederationHttpBinding	WSFederationHttpBinding 是一种安全、可交互操作的绑定, 支持在多个系统上共享身份, 以进行身份验证和授权
WSDualHttpBinding	与 WSHttpBinding 相反, 绑定 WSDualHttpBinding 支持消息的双向传送
NetTcpBinding	所有用 Net 作为前缀的标准绑定都使用二进制编码在 .NET 应用程序之间通信。这个编码比 WSxxx 绑定使用的文本编码快。绑定 NetTcpBinding 使用 TCP/IP 协议
NetTcpContextBinding	类似于 WSHttpContextBinding, NetTcpContextBinding 会添加 ContextBindingElement, 与 SOAP 标题交换环境信息
NetPeerTcpBinding	NetPeerTcpBinding 为对等通信提供了绑定
NetNamedPipeBinding	NetNamedPipeBinding 为在同一系统的不同进程之间的通信进行了优化
NetMsmqBinding	NetMsmqBinding 为 WCF 引入了排队通信。这里消息会发送到消息队列中
MsmqIntegrationBinding	MsmqIntegrationBinding 用于使用消息队列的已有应用程序。而 NetMsmqBinding 需要 WCF 应用程序位于客户机和服务器上
CustomBinding	使用 CustomBinding, 可以完全定制传输协议和安全要求

不同的绑定支持不同的特性。以 WS 开头的绑定是独立于平台的, 支持 Web 服务规范。以 Net 开头的绑定使用二进制格式, 使 .NET 应用程序之间的通信有很高的性能。其他特性有支持会话、可靠的会话、事务处理和双向通信。表 42-7 列出了支持这些特性的绑定。

表 42-7

特 性	绑 定
会话	WSHttpBinding、WSDualHttpBinding、WSFederationHttpBinding、NetTcpBinding、NetNamedPipeBinding
可靠的会话	WSHttpBinding、WSDualHttpBinding、WSFederationHttpBinding、NetTcpBinding
事务处理	WSHttpBinding、WSDualHttpBinding、WSFederationHttpBinding、NetTcpBinding、NetNamedPipeBinding、NetMsmqBinding、MsmqIntegrationBinding
双向通信	WSDualHttpBinding、NetTcpBinding、NetNamedPipeBinding、NetPeerTcpBinding

除了定义绑定之外, 服务还必须定义端点。端点依赖于合同、服务的地址和绑定。在下面的代码示例中, 实例化了一个 ServiceHost 对象, 将地址 `http://localhost:8080/RoomReservation`、一个 WSHttpBinding 实例和合同添加到服务的一个端点上。

```

static ServiceHost host;

static void StartService()
{
    Uri baseAddress = new Uri("http://localhost:8080/RoomReservation");
    host = new ServiceHost(
        typeof(RoomReservationService));

    WSHttpBinding binding1 = new WSHttpBinding();
    host.AddServiceEndpoint(typeof(IRoomService), binding1, baseAddress);
    host.Open();
}

```

除了以编程方式定义绑定之外，还可以在应用程序配置文件中定义它。WCF 的配置放在元素 `<system.serviceModel>` 中，`<service>` 元素定义了所提供的服务。同样，如代码所示，服务需要一个端点，该端点包含地址、绑定和合同信息。WSHttpBinding 的默认绑定配置用 XML 属性 `bindingConfiguration` 修改，该属性引用了绑定配置 `WSHttpConfig1`。这个绑定配置在 `<bindings>` 段中，它用于修改 WSHttpBinding 配置，以启用 `reliableSession`。

```

<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name="Wrox.ProCSharp.WCF.RoomReservationService">
        <endpoint address="http://localhost:8080/RoomReservation"
          contract="Wrox.ProCSharp.WCF.IRoomService"
          binding="wsHttpBinding" bindingConfiguration="wsHttpConfig1" />
      </service>
    </services>
    <bindings>
      <wsHttpBinding>
        <binding name="wsHttpConfig1">
          <reliableSession enabled="true" />
        </binding>
      </wsHttpBinding>
    </bindings>
  </system.serviceModel>
</configuration>

```

## 42.6 主机

在选择运行服务的主机时，WCF 是非常灵活的。主机可以是 Windows 服务、COM+ 应用程序、WAS 或 IIS、Windows 应用程序，或简单的控制台应用程序。在用 Windows 窗体或 WPF 创建定制的主机时，很容易创建对等的解决方案。

### 42.6.1 定制主机

先从定制主机开始。下面的示例代码列出了控制台应用程序中的服务主机。但在其他定制主机类型中，例如 Windows 服务或 Windows 应用程序，可以用相同的方式编写服务。

在 `Main()` 方法中，创建了一个 `ServiceHost` 实例。之后，读取应用程序配置文件，来定义绑定。也可以通过编程方式定义绑定，如前面所示。接着，调用 `ServiceHost` 类的 `Open()` 方法，

使服务接收客户调用。在控制台应用程序中，必须注意在关闭服务之前，不能关闭主线程。这里在调用 `Close()` 方法时，要求用户结束服务。

```
using System;
using System.ServiceModel;

public class Program
{
    public static void Main()
    {
        using (ServiceHost serviceHost = new ServiceHost())
        {
            serviceHost.Open();

            Console.WriteLine("The service started. Press return to exit");
            Console.ReadLine();

            serviceHost.Close();
        }
    }
}
```

要中止服务主机，可以调用 `ServiceHost` 类的 `Abort()` 方法。要获得服务的当前状态，`State` 属性会返回 `CommunicationState` 枚举定义的一个值，该枚举的值有 `Created`、`Opening`、`Opened`、`Closing`、`Closed` 和 `Faulted`。

#### 警告：

如果从 Windows 窗体或 WPF 应用程序中启动服务，该服务的代码调用了 Windows 窗体控件的方法，就必须注意，只有控件的创建线程才能访问该控件的方法和属性。在 WCF 中，设置特性 `[ServiceBehavior]` 的属性 `UseSynchronizationContext`，就可以实现这一点。

### 42.6.2 WAS 主机

在 WAS(Windows Activation Services)主机中，可以使用 WAS 工作进程中的特性，如自动激活服务、健康监控和循环处理。

要使用 WAS 主机，只需创建一个 Web 站点和一个 `.svc` 文件，其中的 `ServiceHost` 声明包含服务类的语言 and 名称。下面的代码使用了类 `Service1`。另外，还必须指定包含服务类的文件。这个类的实现方式与前面定义 WCF 服务库相同。

```
< %@ServiceHost language="C#" Service="Service1"
CodeBehind="Service1.svc.cs" % >
```

如果使用可在 WAS 主机中使用的 WCF 服务库，就可以创建一个 `.svc` 文件，它只包含对类的引用：

```
< %@ ServiceHost
Service="Wrox.ProCSharp.WCF.Services.RoomReservationService" % >
```

在 Windows Vista 和 Windows Server 2008 中，WAS 允许定义 .NET TCP 和 Message Queue 绑定。如果使用以前的版本，Windows Server 2003 和 Windows XP 中的 IIS6 或 IIS 5.1，就只能使用 HTTP 绑定从 `.svc` 文件中激活服务。



提示:

还可以将 WCF 服务添加到 Enterprise Service 组件中, 详见第 44 章。

## 42.7 客户程序

客户应用程序需要一个代理来访问服务。给客户程序创建代理有三种方式:

- Visual Studio 添加服务引用: 这个工具会从服务的元数据中创建代理类。
- ServiceModel 元数据实用工具(Svcutil.exe): 使用 SvcUtil 工具可以创建代理类。该工具从服务中读取元数据, 以创建代理类。
- ChannelFactory 类: 这个类由 Svcutil 生成的代理使用, 它也可以用于以编程方式创建代理。

在 Visual Studio 中添加服务引用, 需要访问 WSDL 文档。WSDL 文档是由 MEX 端点创建的, MEX 端点需要用服务配置。在下面的配置中, 带相对地址 mex 的端点使用 mexHttpBinding, 执行合同 IMetadataExchange。为了通过 HTTP GET 请求访问元数据, 应配置 behaviorConfiguration MexServiceBehavior。

```
< ?xml version="1.0" encoding="utf-8" ? >
< configuration >
  < system.serviceModel >
    < services >
      < service behaviorConfiguration=" MexServiceBehavior "
        name="Wrox.ProCSharp.WCF.Services.RoomReservationService" >
        < endpoint address="Test" binding="wsHttpBinding"
          contract="Wrox.ProCSharp.WCF.Services.IRoomService" / >
        < endpoint address="mex" binding="mexHttpBinding"
          contract="IMetadataExchange" / >
      < host >
        < baseAddresses >
          < add baseAddress=
            "http://localhost:8731/Design_Time_Addresses/RoomReservationService/" /
          >
        < baseAddresses >
        < /host >
      < /service >
    < /services >
    < behaviors >
      < serviceBehaviors >
        < behavior name="MexServiceBehavior" >
          <!-- To avoid disclosing metadata information,
            set the value below to false and remove the metadata endpoint above
            before deployment -->
          < serviceMetadata httpGetEnabled="True" / >
        < /behavior >
      < /serviceBehaviors >
    < /behaviors >
  < /system.serviceModel >
< /configuration >
```

类似于 Visual Studio 中的添加服务引用, 实用工具 Svcutil 需要元数据来创建代理类。Svcutil 工具可以从 MEX 元数据端点、程序集的元数据、WSDL 和 XSD 文档中创建代理。

```
svcutil http://localhost:8080/RoomReservation?wsdl /language:C# /out:proxy.cs
svcutil CourseRegistration.dll
svcutil CourseRegistration.wsdl CourseRegistration.xsd
```

生成的代理类需要在客户代码中实例化、调用方法，最后必须调用 `Close()` 方法：

```
RoomServiceClient client = new RoomServiceClient();
client.RegisterForCourse(roomReservation);
client.Close();
```

生成的代理类派生自基类 `ClientBase<TChannel>`，它封装了 `ChannelFactory<TChannel>` 类。除了使用生成的代理类之外，还可以直接使用 `ChannelFactory<TChannel>` 类。构造函数需要绑定和端点地址；之后，就可以创建信道，调用服务合同定义的方法了。最后，必须关闭该类。

```
WsHttpBinding binding = new WsHttpBinding();
EndpointAddress address =
    new EndpointAddress("http://localhost:8080/RoomService");

ChannelFactory<IRoomService> factory =
    new ChannelFactory<IRoomService>(binding, address);

IRoomService channel = factory.CreateChannel();
channel.ReserveRoom(roomReservation);

//...
factory.Close();
```

`ChannelFactory<TChannel>` 类有几个属性和方法，如表 42-8 所示。

表 42-8

ChannelFactory 成员	说 明
Credentials	Credentials 是一个只读属性，可以访问 ClientCredentials 对象，该对象被赋予信道，对服务进行身份验证。Credentials 可以用端点来设置
Endpoint	Endpoint 是一个只读属性，可以访问与信道相关的 ServiceEndpoint。端点可以在构造函数中指定
State	State 属性的类型是 CommunicationState，它返回信道的当前状态。CommunicationState 是一个枚举，其值是 Created、Opening、Opened、Closing、Closed 和 Faulted
Open()	该方法用于打开信道
Close()	该方法用于关闭信道
Opening、Opened、Closing、Closed 和 Faulted	可以指定事件处理程序，确定信道的状态变化。这些事件分别在信道打开前后、信道关闭前后和出错时发生

42.8 双向通信

下面的示例程序说明了如何在客户程序和服务程序之间直接进行双向通信。客户程序会启动与服务的连接。之后，服务就可以回调客户程序了。

为了进行双向通信，必须指定一个在客户程序中实现的合同。这里用于客户程序的合同由接口 `IMyMessageCallback` 定义。由客户程序实现的方法是 `OnCallback()`。操作使用了 `IsOneWay=true` 操作合同设置。这样，服务就不必等待方法在客户程序上成功调用了。在默认情况下，服务实例只能从一个线程中调用(服务操作的 `ConcurrencyMode` 属性默认设置为 `ConcurrencyMode.Single`)。

如果服务的实现代码回调了客户程序，等待获得客户程序的结果，则从客户程序中获得回应的线程就必须等待，直到得到服务对象的锁定为止。服务对象已经由客户程序的请求锁定，所以出现了死锁。WCF 检测到这个死锁，就抛出一个异常。为了避免这种情况，可以将 `ConcurrencyMode` 属性改为 `Multiple` 或 `Reentrant`。使用 `Multiple` 设置，多个线程可以同时访问实例。这里必须实现锁定，使用 `Reentrant` 设置，服务实例将只使用一个线程，但允许将回调请求的回应重新输入到上下文中。除了改变并发模式之外，还可以用操作合同指定 `IsOneWay` 属性。这样，调用者就不会等待回应了。当然，只有不需要返回值，才能使用这个设置。

服务合同由接口 `IMyMessage` 定义。回调合同用服务合同定义的 `CallbackContract` 属性映射到服务合同上。

```
public interface IMyMessageCallback
{
    [OperationContract(IsOneWay=true)]
    void OnCallback(string message);
}

[ServiceContract(CallbackContract=typeof(IMyMessageCallback))]
public interface IMyMessage
{
    [OperationContract]
    void MessageToServer(string message);
}
```

类 `MessageService` 实现了服务合同 `IMyMessage`。服务将来自客户的消息写入控制台。要访问回调合同，可以使用 `OperationContext` 类。`OperationContext.Current` 返回与客户中当前请求关联的 `OperationContext`。使用 `OperationContext` 可以访问会话信息、消息标题和属性，在双向通信的情况下还可以访问回调信道。泛型方法 `GetCallbackChannel()` 将信道返回客户实例。接着调用由回调接口 `IMyMessageCallback` 定义的 `OnCallback()`，使用这个信道将消息发送给客户。为了演示这些操作，还可以在方法环境中使用独立于服务的回调信道，创建一个接收回调信道的新线程。这个新线程再次使用回调信道，将消息发送给客户。

```
public class MessageService : IMyMessage
{
    public void MessageToServer(string message)
    {
        Console.WriteLine("message from the client: {0}", message);
        IMyMessageCallback callback =
            OperationContext.Current.GetCallbackChannel<IMyMessageCallback>();

        callback.OnCallback("message from the server");

        new Thread(ThreadCallback).Start(callback);
    }

    private void ThreadCallback(object callback)
```

```

    {
        IMyMessageCallback messageCallback = callback as IMyMessageCallback;
        for (int i = 0; i < 10; i++)
        {
            messageCallback.OnCallback("message " + i.ToString());
            Thread.Sleep(1000);
        }
    }
}

```

启动服务的方式与前面的例子相同，这里不再赘述。但是对于双向通信，必须配置一个支持双向通信的绑定。支持双向信道的一个绑定是 `WSDualHttpBinding`，它在应用程序的配置文件中配置。

```

<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name="Wrox.ProCSharp.WCF.MessageService">
        <endpoint contract="Wrox.ProCSharp.WCF.IMyMessage"
          binding="wsDualHttpBinding"/>
      </service>
    </services>
    <host>
      <baseAddresses>
        <add baseAddress="http://localhost:8731/Service1" />
      </baseAddresses>
    </host>
  </system.serviceModel>
</configuration>

```

在客户程序中，必须用 `ClientCallback` 类实现回调合同，该类实现了接口 `IMyMessageCallback`，如下所示：

```

class ClientCallback : IMyMessageCallback
{
    public void OnCallback(string message)
    {
        Console.WriteLine("message from the server: {0}", message);
    }
}

```

在双向信道中，不能像前面那样使用 `ChannelFactory` 启动与服务的连接。要创建双向信道，可以使用 `DuplexChannelFactory` 类。这个类有一个构造函数，除了绑定和地址配置之外，它还有一个参数，这个参数指定了 `InstanceContext`，它封装了 `ClientCallback` 类的一个实例。把这个实例传送给 `DuplexChannelFactory`，服务就可以通过信道调用对象了。客户只需使连接一直处于打开状态。如果连接关闭了，服务就不能通过它发送消息了。

```

WSDualHttpBinding binding = new WSDualHttpBinding();
EndpointAddress address =
    new EndpointAddress("http://localhost:8080/service1");

ClientCallback clientCallback = new ClientCallback();
InstanceContext context = new InstanceContext(clientCallback);

DuplexChannelFactory<IMyMessage> factory =
    new DuplexChannelFactory<IMyMessage>(context, binding, address);

```

```
IMyMessage messageChannel = factory.CreateChannel();  
messageChannel.MessageToServer("From the client");
```

启动服务主机和客户程序，就可以进行双向通信了。

## 42.9 小结

本章学习了如何使用 Windows Communication Foundation 在客户机和服务器之间通信。WCF 与 ASP.NET Web 服务一样，也独立于平台，但提供了与 .NET Remoting、Enterprise Services 和消息队列类似的功能。

WCF 主要利用服务合同、数据合同和消息合同，来简化客户程序和服务程序的独立开发，支持独立的平台。可以使用几个属性定义服务的操作。

我们还探讨了如何从服务提供的元数据中创建客户程序，如何使用 .NET 接口合同来创建客户程序。

本章介绍了不同绑定选项的特性。WCF 不仅提供了独立于平台的绑定，还提供了在 .NET 应用程序之间快速通信的绑定。本章还探讨了如何创建定制主机和如何使用 WAS 主机。如何定义回调接口，应用服务回调，在客户应用程序中执行回调合同，实现双向通信。

后面几章继续介绍 WCF 特性。第 44 章学习如何集成 Enterprise Services 和 WCF，第 45 章解释如何使用断开连接的 Message Queuing 特性和 WCF 绑定，第 43 章讨论 Windows Workflow Foundation，它使用 WCF 与 Workflow 实例通信。



# 第43章

## Windows Workflow Foundation

本章将概述 Windows Workflow Foundation (本章称之为 WF)，它提供了一个模型，在该模型中，可以使用一组构建块(称为活动)定义和执行过程。WF 还提供了一个设计器，在默认情况下，该设计器位于 Visual Studio 中，允许将工具箱中的活动拖放到设计界面上，创建一个工作流模板。

创建一个工作流实例，运行该实例，就可以执行这个模板。执行工作流的代码称为 Workflow Runtime，该对象也可以包含许多服务，它们可以访问正在运行的工作流。在任意时刻，均有几个工作流实例在执行，运行库会安排这些实例的运行，保存和恢复状态，还可以记录每个工作流实例的执行情况。

工作流是由许多活动构成的，这些活动由运行库执行。活动可以是发送电子邮件、更新数据库中的一行，或在后端系统上执行一个事务处理。有许多内置的活动，它们用于一般性的工作，也可以创建自己的定制活动，根据需要将它们放在工作流中。本章的主要内容如下：

- 可以创建的不同类型的工作流
- 描述一些内置的活动
- 如何创建定制活动

本章从一个规范的例子 Hello World 开始，每个人在面对一个新技术时都要使用这个例子，并描述在开发机器上使工作流运行起来所需要做的工作。

### 43.1 Hello World 示例

Visual Studio 2008 包含对创建工作流的支持。打开 New Project 对话框，会看到一系列工作流项目类型，如图 43-1 所示。

从可用的模板中选择 Sequential Workflow Console Application(它会创建一个包含工作流运行库的控制台应用程序)和默认的工作流，以后要给该工作流添加活动。

接着，把 Code 活动从工具箱拖放到设计界面上，就会得到如图 43-2 所示的工作流。

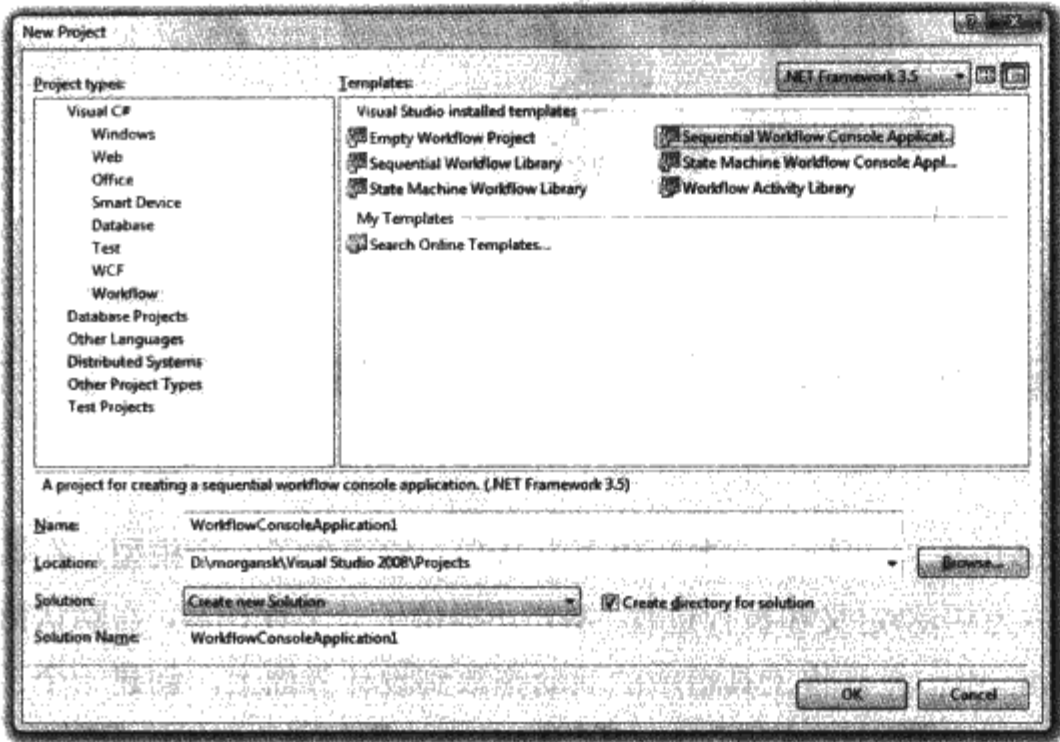


图 43-1

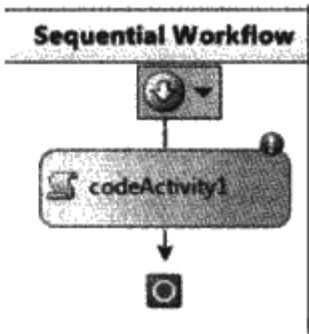


图 43-2

活动右上角的感叹号标记表示，没有定义该活动的强制属性，这里是 `ExecuteCode` 属性，它指定了活动执行时调用的方法。“活动有效性”一节将学习如何把自己的属性标记为强制属性。如果双击了 `CodeActivity`，就在后台编码类中创建一个方法，该方法使用 `Console.WriteLine` 输出字符串 `Hello World`，如下面的代码所示：

```
private void codeActivity1_ExecuteCode(object sender, EventArgs e)
{
    Console.WriteLine("Hello World");
}
```

如果建立并运行程序，就会在控制台上看到输出的文本。程序执行时，创建了一个 `WorkflowRuntime` 类型的实例，然后构建一个工作流实例，并执行它。在执行 `Code` 活动时，会调用定义好的方法，将字符串输出到控制台上。“工作流运行库”一节将详细描述如何执行运行库。上述示例代码在 `01 HelloWorldWorkflowWorld` 文件夹中。

43.2 活动

工作流中的内容是一个活动——该活动位于工作流中，其类型比较特殊，它允许在其中定义其他活动，这称为复合活动，本章的后面将介绍其他复合活动。活动是一个最终派生自 `Activity` 类的类。

`Activity` 类定义了许多可重写的方法，其中最重要的是 `Execute` 方法，如下所示：

```
protected override ActivityExecutionStatus Execute
( ActivityExecutionContext executionContext )
{
    return ActivityExecutionStatus.Closed;
}
```

在运行库安排活动的执行时，最终会调用 `Execute` 方法，在该方法中，可以编写定制代码，

提供活动的操作。在上一节的简单示例中，当 workflow 运行库在 `CodeActivity` 上调用 `Execute` 方法时，这个方法的实现代码就会执行在后台编码类中定义的方法，在控制台上显示消息。

`Execute` 方法需要一个 `ActivityExecutionContext` 类型的环境参数，本章的后面将探讨这个参数。该方法的返回值是 `ActivityExecutionStatus` 类型，该返回值由运行库用于确定活动是已成功完成、仍在处理，还是处于其他几个状态中，这几个状态可以向 workflow 运行库描述活动所处的状态。从这个方法中返回 `ActivityExecutionStatus.Closed`，表示活动已完成了工作，可以删除了。

WF 提供了许多标准活动，下面几节就介绍其中一些活动的例子，并说明使用这些活动的场合。活动的命名约定是在名称的后面加上 `Activity`。例如，图 43-2 中的代码活动就由 `CodeActivity` 类定义。

所有的标准活动都在 `System.Workflow.Activities` 命名空间中定义，该命名空间位于程序集 `System.Workflow.Activities.dll` 中。还有两个程序集 `System.Workflow.ComponentModel.dll` 和 `System.Workflow.Runtime.dll` 也是 WF 的组成部分。

### 43.2.1 IfElseActivity

顾名思义，这个活动的操作类似于 C# 中的 If-Else 语句。

将一个 `IfElseActivity` 拖放到设计界面上，会看到如图 43-3 所示的活动。`IfElseActivity` 是一个复合活动，它建立了两个分支（这两个分支也是活动，即 `IfElseBranchActivity`）。每个分支也都是派生自 `SequenceActivity` 的复合活动，这个类自上而下地执行每个活动。设计器添加了 `Drop Activities Here` 文本，指出可以把子活动添加到什么地方。

图 43-3 中的第一个分支包含一个符号，指定需要定义 `Condition` 属性。条件派生自 `ActivityCondition`，用于确定是否执行该分支。

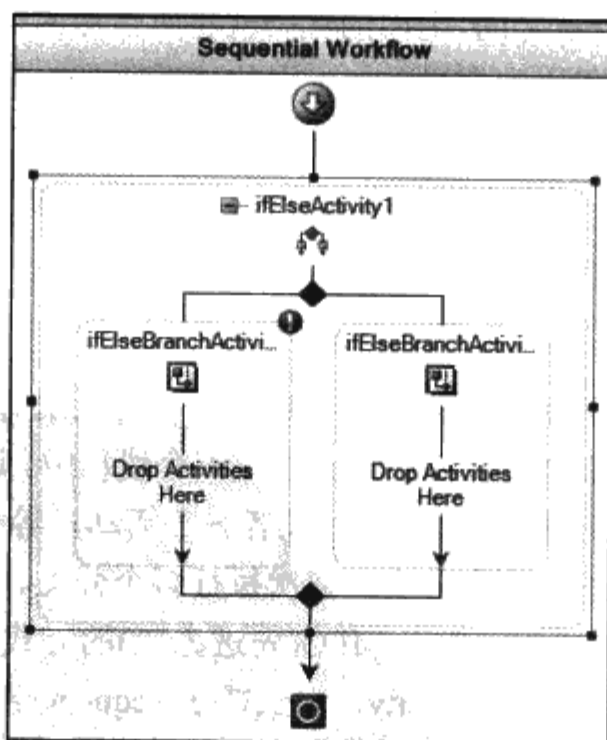


图 43-3

在执行 `IfElseActivity` 时，将计算第一个分支的条件，如果该条件等于 `true`，就执行该分支。如果条件等于 `false`，`IfElseActivity` 就尝试执行下一个分支，依此类推，直到活动中的最后一个分支为止。注意，`IfElseActivity` 可以有任意多个分支，每个分支都有自己的条件。最后一个分支

不能有条件，因为它相当于 If-Else 语句中的 else 部分。要添加一个新分支，可以显示活动的关联菜单，从菜单中选择 Add Branch——也可以在 Visual Studio 的 Workflow 菜单中选择它。在添加分支时，每个分支都有一个强制的条件，但最后一个分支例外。

WF 定义了两个标准的条件类型 CodeCondition 和 RuleConditionReference。CodeCondition 类在后台编码类上执行一个方法，返回 true 或 false。要创建 CodeCondition，可以显示 IfElseActivity 的属性表，将条件设置为 Code Condition，再为要执行的代码输入一个名称，如图 43-4 所示。

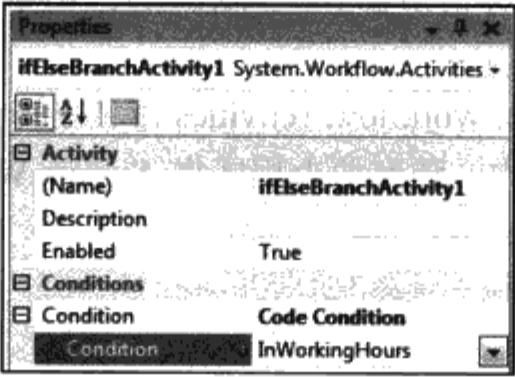


图 43-4

将方法名输入属性表时，设计器会在后台编码类上构建一个方法，如下面的代码所示：

```
private void InWorkingHours(object sender, ConditionalEventArgs e)
{
    int hour = DateTime.Now.Hour;

    e.Result = ((hour >= 9) && (hour <= 17));
}
```

如果当前时间在 9 am 和 5 pm 之间，上面的代码就将所传送的 ConditionalEventArgs 的 Result 属性设置为 true。条件可以在代码中定义，如上面的代码所示，另一种方法是根据以类似方式计算的 Rule 来定义条件。Workflow 设计器包含一个规则编辑器，它可以用于声明条件和语句(类似于上面的 If-Else 语句)。这些规则在运行期间根据工作流的当前状态进行计算。

43.2.2 ParallelActivity

这个活动可以定义同时执行的一系列活动，或者以伪并行的方式执行的一系列活动。当工作流运行库安排一个活动的执行时，会将该活动放在一个线程中。这个线程先执行第一个活动，再执行第二个活动，直到完成所有的活动为止(或者某个活动在等待某种形式的输入为止)。在执行 ParallelActivity 时，它会迭代每个分支，依次执行每个分支。工作流运行库为每个工作流实例维护一个已安排的活动队列，一般以 FIFO(先进先出)的方式执行它们。

假定有如图 43-5 所示的 ParallelActivity，它安排了 sequenceActivity1 和 sequenceActivity2 的执行。SequenceActivity 类型的工作方式是先用运行库执行第一个活动，这个活动执行完毕后，就安排第二个活动。这个安排/等待完成方法会遍历系列中的所有子活动，所有的子活动都执行完毕后，SequenceActivity 就完成了。

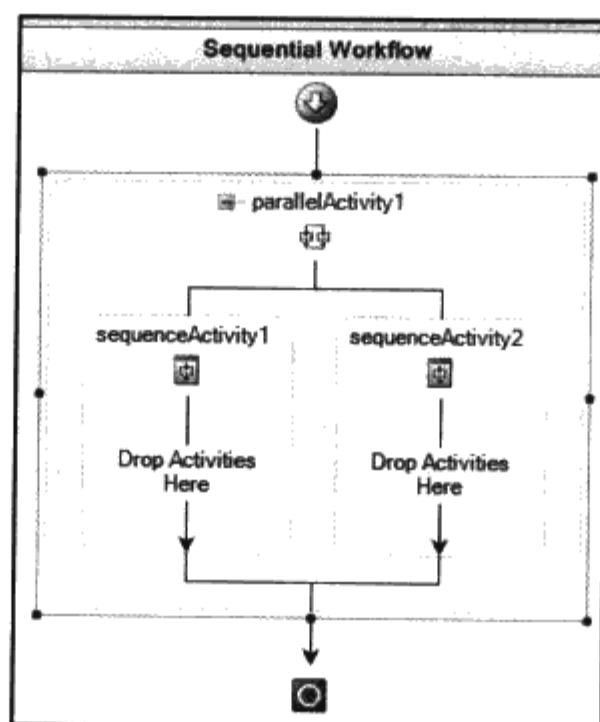


图 43-5

如果 `SequenceActivity` 一次安排执行一个活动，就表示 `WorkflowRuntime` 维护的队列会用可安排执行的活动连续不断地更新。假定有一个并行活动 `P1`，它包含两个系列 `S1` 和 `S2`，每个系列都有两个代码活动 `C1` 和 `C2`，这会在队列中生成如表 43-1 所示的项。

表 43-1

工作流队列	最初队列中没有活动
P1	在工作流运行时并发执行
S1, S2	执行 P1 时，添加到队列中
S2, S1.C1	S1 执行，并将 S1.C1 添加到队列中
S1.C1, S2.C1	S2 执行，并将 S2.C1 添加到队列中
S2.C1, S1.C2	S1.C1 完成，所以 S1.C2 进入队列
S1.C2, S2.C2	S2.C1 完成，所以 S2.C2 进入队列
S2.C2	队列中的最后一项

这里，队列处理第一项(并发活动 `P1`)，这将系列活动 `S1` 和 `S2` 添加到工作流队列中。在执行系列活动 `S1` 时，它会将其第一个子活动 `S1.C1` 放在队列的尾部，执行完这个活动后，就把第二个子活动添加到队列中。

从上面的例子可以看出，`ParallelActivity` 的执行并不真是并行的，而是在两个系列分支之间交叉执行。由此可以推断，最好的情况是活动的执行时间最短，因为每个工作流都只有一个线程为其安排的队列服务，所以运行时间较长的活动会妨碍队列中其他活动的执行。但是，活动的执行时间常常是任意的，所以必须采用某种方式将活动标记为“运行时间较长”，使其他活动有机会执行。为此，可以从 `Execute` 方法中返回 `ActivityExecutionStatus.Executing`，在活动结束时让运行库知道，以后还要调用它。这种活动的一个例子是 `DelayActivity`。



### 43.2.3 CallExternalMethodActivity

工作流一般需要调用工作流外部的的方法，这个活动可以定义一个接口和在该接口上调用的方法。`WorkflowRuntime` 维护了一个服务列表(其键为一个 `System.Type` 值)，使用传送给 `Execute` 方法的 `ActivityExecutionContext` 参数可以访问该服务列表。

可以定义自己的服务，并添加到这个集合中，再从自己的活动中访问这些服务。例如，可以构建一个数据访问层，用作一个服务接口，再为 `SQL Server` 和 `Oracle` 提供不同的服务实现代码。活动只是调用接口方法，所以从 `SQL Server` 到 `Oracle` 的切换对活动而言是不透明的。

将一个 `CallExternalMethodActivity` 添加到工作流中后，就定义两个强制属性 `InterfaceType` 和 `MethodName`。接口类型定义了在执行活动时运行库要使用的服务，方法名指定了要调用该接口的哪个方法。

在执行这个活动时，将查询该服务类型的执行环境，找出有指定接口的服务，然后调用该接口上的相应方法。也可以在工作流中给方法传送参数，这个内容将在本节后面的“给活动绑定参数”小节中讨论。

### 43.2.4 DelayActivity

业务处理常常需要等待一段时间才能完成——考虑使用工作流进行费用申请的过程。工作流给经理发送一封电子邮件，要求他批准某个费用申请。之后工作流进入等待状态，等待经理批准(或者不批准)，这里最好定义一个超时时间。如果在 1 天的时间内没有返回响应，费用申请就路由给命令链中的下一个经理。

`DelayActivity` 可以实现这个情形的一部分(另一个部分是下面定义的 `ListenActivity`)，其任务是等待指定的时间，之后继续执行工作流。定义延迟时间有两种方式：可以将延迟的 `TimeoutDuration` 属性设置为一个字符串，如“1.00:00:00”(1 天，没有指定小时、分钟和秒)；也可以提供一个方法，在执行活动时，调用该方法，在代码中将延迟时间设置为一个值。为此，需要为 `DelayActivity` 的 `InitializeTimeoutDuration` 属性定义一个值，这会在后台代码中创建一个方法，如下面的代码所示：

```
private void DefineTimeout(object sender, EventArgs e)
{
    DelayActivity delay = sender as DelayActivity;
    if (null != delay)
    {
        delay.TimeoutDuration = new TimeSpan(1, 0, 0, 0);
    }
}
```

这里的 `DefineTimeout` 方法给发送者传送了一个 `DelayActivity`，然后在代码中把 `TimeoutDuration` 属性设置为 `TimeSpan`。尽管这里硬编码了这个值，但可以在其他数据中构建它，例如传送给工作流的一个参数，或者从配置文件中读取的一个值。工作流的参数在本章后面的“工作流”一节中讨论。

### 41.2.5 ListenActivity

一个公共的编程结构是等待某个可能的事件，例如 `System.Threading.WaitHandle` 类的 `WaitAny` 方法。`ListenActivity` 是工作流中等待事件发生的一种方式，因为它可以定义任意多个分支，每个分支都把基于事件的活动作为该分支的第一个活动。

事件活动实现了在 `System.Workflow.Activities` 命名空间中定义的 `IEventActivity` 接口。WF 将三个这样的活动定义为标准活动：`DelayActivity`、`HandleExternalEventActivity` 和 `WebServiceInputActivity`。图 43-6 中的工作流在等待外部的输入，或者延迟——这是前面讨论的费用申请工作流示例。

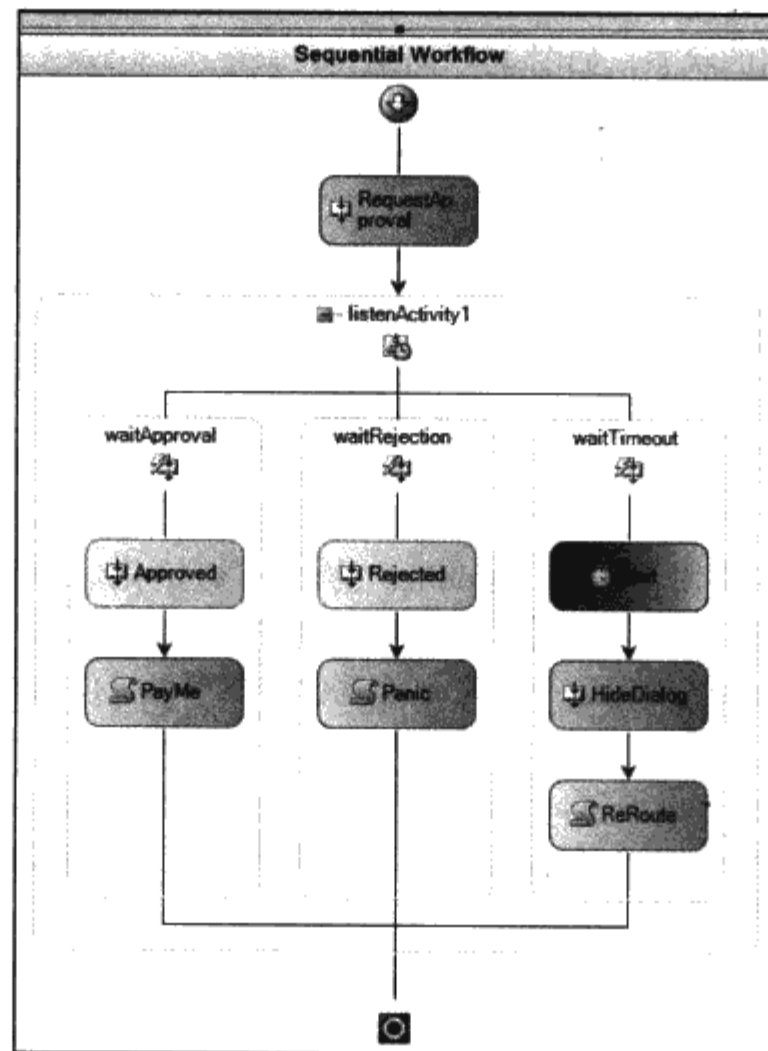


图 43-6

在这个例子中，`CallExternalMethodActivity` 用作工作流中的第一个活动，它会调用在服务接口上定义的方法，提请经理批准或不批准——因为这是一个外部服务，所以该提请可以是电子邮件、IM 消息或用其他方式通知经理，需要处理一个费用申请。之后，工作流执行 `ListenActivity`，等待这个外部服务的输入(批准或不批准)，同时也在延迟。

在执行 `ListenActivity` 时，会将一个等待操作放在每个分支的第一个活动中。在触发一个事件时，会取消其他所有的等待事件，并处理触发了事件的分支的其他活动。所以在前面的示例中，如果批准了费用报告，就引发 `Approved` 事件，安排 `PayMe` 活动。但如果经理没有批准该费用申请，就引发 `Rejected` 事件，然后执行 `Panic` 活动。

最后，如果 `Approved` 事件和 `Rejected` 事件都没有引发，`DelayActivity` 就会在延迟时间过

后完成，费用报告可以路由给下一个经理——可能在 Active Directory 中查找这个人。在本示例中，执行 RequestApproved 活动时，会给用户显示一个对话框。所以在这种情况下，执行 DelayActivity 时，还需要关闭这个对话框，而这就是图 43-6 中 HideDialog 活动的作用。

这个例子的代码在 02 Listen 目录下。该例使用了一些前面没有介绍的概念，例如 workflow 实例如何标识，事件如何引发，以返回 workflow 运行库，最终发送给正确的工作流实例。这些概念将在“工作流”一节中探讨。

### 43.2.6 活动的执行模型

到目前为止，本章仅讨论了运行库通过调用 Execute 方法来执行活动。实际上，活动在执行过程中会经历许多不同的状态，如图 43-7 所示。

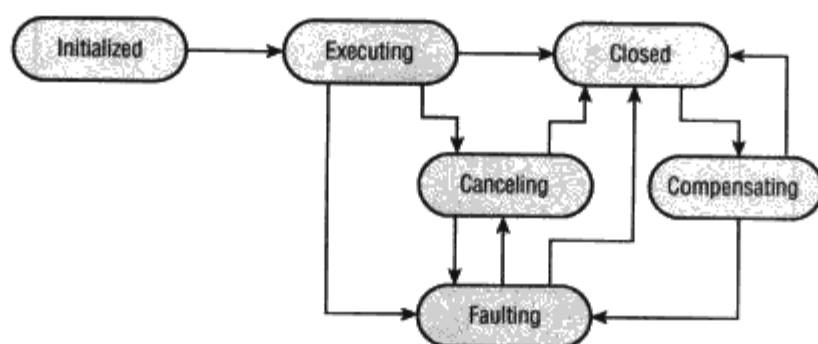


图 43-7

WorkflowRuntime 先调用活动的 Initialize 方法，初始化这个活动。给这个方法传送 IServiceProvider 实例，该实例映射了运行库中可用的服务。这些服务将在本章后面的“工作流服务”一节中讨论。大多数活动在这个方法中都是什么都不做，但这个方法会进行一些必要的设置。

接着，运行库调用 Execute 方法，活动就可以从 ActivityExecutionStatus 枚举中返回一个值了。一般应从 Execute 方法中返回 Closed，表示活动处理完毕；但是如果返回了其他的状态值，运行库就使用该值确定活动所处的状态。

从这个方法中可以返回 Executing，表示运行库还有额外的工作要做。一个典型的例子是有一个复合活动，它需要执行其子活动。在这种情况下，活动可以安排其每个子活动的执行，接着等待所有子活动执行完毕，之后通知运行库，活动已完成。

## 43.3 定制的活动

前面使用的都是 System.Workflow.Activities 命名空间中定义的活动。本节将学习如何创建定制的活动，扩展它们，为用户提供更好的设计和运行体验。

首先，编写 WriteLineActivity，将一行文本输出到控制台上。这是一个简单的例子，后面将扩展它，介绍定制活动的所有选项。在创建定制活动时，可以在 workflow 项目中构建一个类，但最好在一个独立的程序集中构建定制的活动，因为 Visual Studio 的设计环境(和工作流项目)会从独立的程序集中加载活动，并能在更新程序集时锁定它。所以，应创建一个简单的类库项目，在其中构建定制的活动。

简单的活动，如 `WriteLineActivity`，直接派生自 `Activity` 基类。下面的代码构建了一个活动类，定义了 `Message` 属性，在调用 `Execute` 方法时会显示该属性：

```
using System;
using System.ComponentModel;
using System.Workflow.ComponentModel;

namespace SimpleActivity
{
    /// <summary>
    /// A simple activity that displays a message to the console when it executes
    /// </summary>
    public class WriteLineActivity : Activity
    {
        /// <summary>
        /// Execute the activity - display the message on screen
        /// </summary>
        /// <param name="executionContext"></param>
        /// <returns></returns>
        protected override ActivityExecutionStatus Execute
            (ActivityExecutionContext executionContext)
        {
            Console.WriteLine(Message);

            return ActivityExecutionStatus.Closed;
        }

        /// <summary>
        /// Get/Set the message displayed to the user
        /// </summary>
        [Description("The message to display")]
        [Category("Parameters")]
        public string Message
        {
            get { return _message; }
            set { _message = value; }
        }

        /// <summary>
        /// Store the message displayed to the user
        /// </summary>
        private string _message;
    }
}
```

在 `Execute` 方法中，可以将消息写入控制台，再返回 `Closed` 状态，通知运行库，活动已完成。

也可以在 `Message` 属性上定义特性，因此在这个属性上定义其描述内容和类别，它们将在 Visual Studio 的属性表中使用，如图 43-8 所示。

本节用于创建活动的代码在 03 CustomActivities 解决方案中。如果编译该解决方案，就可以从工具箱的关联菜单中选择 `Choose Items` 菜单项，导航到包含活动的程序集所在的文件夹，将定制活动添加到 Visual Studio 的工具箱中。程序集中的所有活动都添加到工具箱中。



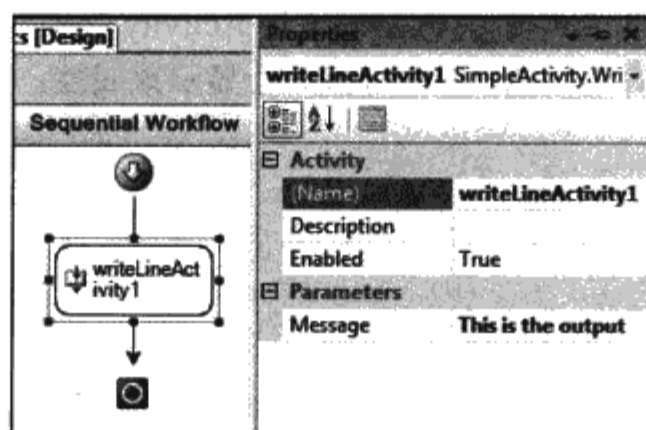


图 43-8

这个活动肯定是可以使用的，但是，有几种方式可以使这个活动更友好。与本章前面的 `CodeActivity` 一样，它有一些强制的属性，若没有定义，就会在设计界面上生成错误。为了使活动有相同的操作，需要构建一个派生自 `ActivityValidator` 的类，将这个类与定制活动关联起来。

### 43.3.1 活动的有效性验证

把活动放在设计界面上时， workflow 设计器就会在该活动上查找一个属性，它定义了对活动进行有效性验证的类。为了验证活动，需要检查是否设置了 `Message` 属性。

给活动实例传送一个定制的验证器，以确定哪些强制属性没有定义，并给设计器使用的 `ValidationErrorsCollection` 添加一个错误。之后 workflow 设计器读取这个集合，该集合中的错误会将一个错误消息添加到活动中，并将每个错误链接到需要注意的属性上。

```
using System;
using System.Workflow.ComponentModel.Compiler;

namespace SimpleActivity
{
    public class WriteLineValidator : ActivityValidator
    {
        public override ValidationErrorsCollection Validate(
            ValidationManager manager, object obj)
        {
            if (null == manager)
                throw new ArgumentNullException("manager");
            if (null == obj)
                throw new ArgumentNullException("obj");

            ValidationErrorsCollection errors = base.Validate(manager, obj);

            // Coerce to a WriteLineActivity
            WriteLineActivity act = obj as WriteLineActivity;

            if (null != act)
            {
                if (null != act.Parent)
                {
                    // Check the Message property
                    if (string.IsNullOrEmpty(act.Message))
                        errors.Add(ValidationErrors.GetNotSetValidationError("Message"));
                }
            }
        }
    }
}
```



```

    }
    return errors;
}
}
}

```

更新活动的任一部分时，将活动放在设计界面上时，设计器都会调用 `Validate` 方法。设计器调用 `Validate` 方法时，会把活动传送为未指定类型的 `obj` 参数。

在这个方法中，先验证传送进来的参数，再调用基类的 `Validate` 方法得到一个 `Validation-ErrorCollection`。尽管这不是严格必需的，但如果从有许多需要验证的属性的活动中派生，调用基类方法就能确保也检查这些属性。

将传送进来的 `obj` 参数强制转换为 `WriteLineActivity` 实例，检查活动是否有父活动。这个测试是必需的，因为 `Validate` 方法在编译活动的过程中调用(假定活动在一个 workflow 项目或活动库中)，此时并没有定义父活动。没有这个检查，就不能建立包含活动的程序集和验证器。如果项目的类型是类库，就不需要这个额外的步骤。

最后一步是检查 `Message` 属性是否设置了一个值，而不是空字符串——这使用 `ValidationError` 类的一个静态方法，它生成了一个错误，说明属性没有定义。

为了给 `WriteLineActivity` 添加验证支持，最后一步是将 `ActivityValidation` 属性添加到活动中，如下面的代码所示：

```

[ActivityValidator(typeof(WriteLineValidator))]
public class WriteLineActivity : Activity
{
    ...
}

```

如果编译应用程序，再将一个 `WriteLineActivity` 放在 workflow 上，就会看到如图 43-9 所示的验证错误；单击这个错误，就能在属性表中看到这个属性。

如果给 `Message` 属性输入一些文本，就会去除验证错误，之后就可以编译、运行应用程序了。

完成了活动的验证后，接下来就要修改活动的显示操作，给该活动添加填充色。为此，需要定义 `ActivityDesigner` 类和 `ActivityDesignerTheme` 类，如下一节所示。

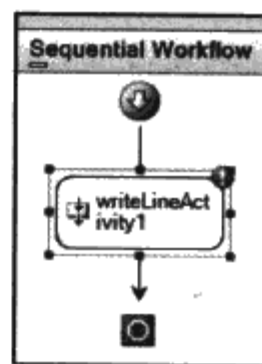


图 43-9

### 43.3.2 主题和设计器

活动的屏幕显示用 `ActivityDesigner` 类执行，这也可以使用 `ActivityDesignerTheme` 类执行。主题类用于在 workflow 设计器中对活动的显示操作进行简单的修改。

```

public class WriteLineTheme : ActivityDesignerTheme
{
    /// <summary>
    /// Construct the theme and set some defaults
    /// </summary>
    /// <param name="theme"></param>
    public WriteLineTheme(WorkflowTheme theme)
    {
    }
}

```

```

        : base(theme)
    {
        this.BackColorStart = Color.Yellow;
        this.BackColorEnd = Color.Orange;
        this.BackgroundStyle = LinearGradientMode.ForwardDiagonal;
    }
}

```

主题派生自 `ActivityDesignerTheme`，它有一个参数为 `WorkflowTheme` 的构造函数。在该构造函数中，为活动设置起始和结束颜色，再定义一个线性渐变笔刷，用于绘制背景。

`Designer` 类用于重写活动的显示操作——这里不需要重写，所以下面的代码就足够了：

```

[ActivityDesignerTheme(typeof(WriteLineTheme))]
public class WriteLineDesigner : ActivityDesigner
{
}

```

注意使用 `ActivityDesignerTheme` 属性将主题与设计器关联起来。

最后一步是用 `Designer` 属性装饰活动：

```

[ActivityValidator(typeof(WriteLineValidator))]
[Designer(typeof(WriteLineDesigner))]
public class WriteLineActivity : Activity
{
    ...
}

```

于是，活动的显示结果如图 43-10 所示。

添加了设计器和主题之后，活动现在看上去专业多了。主题上还有许多其他属性，例如用于绘制边框的笔、边框的颜色、边框的样式等。

重写 `ActivityDesigner` 类的 `OnPaint` 方法，可以完全控制活动的显示。这里最好练习一下，因为可以稍微走远一点儿，创建一个与工具箱中其他活动都不雷同的活动。

在 `ActivityDesigner` 类中，另一个有用的重写属性是 `Verbs`。它可以在活动的关联菜单中添加菜单项，用 `ParallelActivity` 的设计器将 `Add Branch` 菜单项插入活动的关联菜单和 `Workflow` 菜单。还可以重写设计器的 `PreFilterProperties` 方法，修改活动的属性列表，这是 `CallExternal MethodActivity` 的方法参数显示到属性表中的方式。如果需要对设计器进行这类扩展，就应运行 Lutz Roeder 的 `Reflector`(<http://www.aisto.com/dotnet>)，加载工作程序集，看看 Microsoft 是如何定义一些扩展属性的。

这个活动快完成了，现在需要定义显示活动时使用的图标，以及与活动关联起来的工具箱选项。

### 43.3.3 ActivityToolboxItem 和图标

为了完成定制活动，需要添加一个图标，还可以创建一个派生自 `ActivityToolboxItem` 的类，在 Visual Studio 的工具箱中显示活动时，要用到这个类。

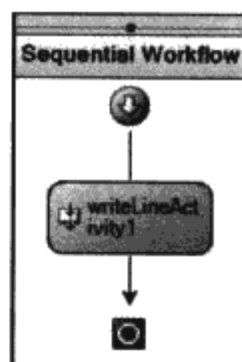


图 43-10

为了给活动定义图标，创建一个 16×16 像素的图像，将它包含到项目中，之后将图标的建立操作设置为 Embedded Resource。这会将该图像包含到程序集的清单资源中。可以给项目添加一个文件夹 Resources，如图 43-11 所示。

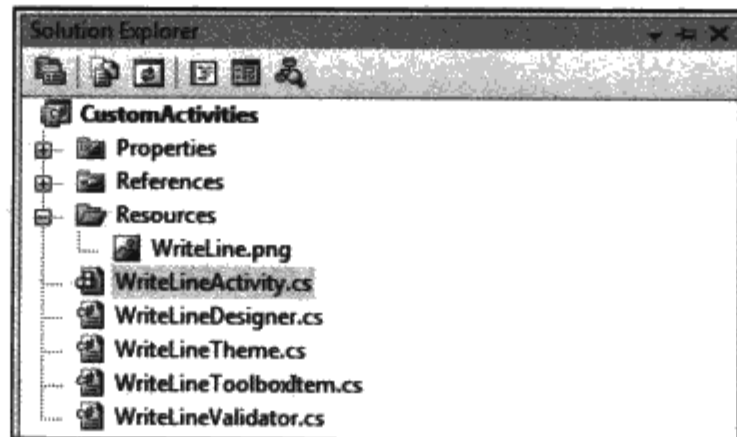


图 43-11

添加了图像文件，将其建立操作设置为 Embedded Resource 后，就可以设置活动的属性了，如下面的代码所示：

```
[ActivityValidator(typeof(WriteLineValidator))]
[Designer(typeof(WriteLineDesigner))]
[ToolboxBitmap(typeof(WriteLineActivity), "Resources.WriteLine.png")]
public class WriteLineActivity : Activity
{
    ...
}
```

ToolboxBitmap 属性定义了许多构造函数，这里使用的构造函数的参数是活动程序集中定义的类型和资源名称。将一个资源添加到文件夹中时，其名称设置为程序集的命名空间，之后是图像所在的文件夹名称，所以该资源的完全限定名称是 CustomActivities.Resources.WriteLine.png。给 ToolboxBitmap 属性使用的构造函数将类型参数所在的命名空间追加到传送为第二个参数的字符串上，这样在由 Visual Studio 加载时，这个名称会解析为相应的资源。

最后一个需要创建的类派生自 ActivityToolboxItem。当活动加载到 Visual Studio 工具箱中时，会使用这个类。它的一个典型应用是修改活动在工具箱上显示的名称——所有的内置活动都会修改其名称，从类型中删除“Activity”。在这个类中，要将 DisplayName 属性设置为 WriteLine，以修改活动的名称。

```
[Serializable]
public class WriteLineToolboxItem : ActivityToolboxItem
{
    /// <summary>
    /// Set the display name to WriteLine - i.e. trim off the 'Activity' string
    /// </summary>
    /// <param name="t"></param>
    public WriteLineToolboxItem(Type t)
        : base(t)
    {
        base.DisplayName = "WriteLine";
    }
    /// <summary>
    /// Necessary for the Visual Studio design time environment
```

```

/// </summary>
/// <param name="info"></param>
/// <param name="context"></param>
private WriteLineToolboxItem(SerializationInfo info, StreamingContext context)
{
    this.Deserialize(info, context);
}
}

```

这个类派生自 `ActivityToolboxItem`，重写了构造函数，以修改显示名称；它还提供了一个串行化构造函数，当将选项加载到工具箱上时，工具箱会使用这个串行化构造函数。没有这个构造函数，在试图将活动添加到工具箱上时，就会生成一个错误。注意这个类也标记为 `[Serializable]`。

工具箱中的选项使用 `ToolboxItem` 属性添加到活动中，如下面的代码所示：

```

[ActivityValidator(typeof(WriteLineValidator))]
[Designer(typeof(WriteLineDesigner))]
[ToolboxBitmap(typeof(WriteLineActivity), "Resources.WriteLine.png")]
[ToolboxItem(typeof(WriteLineToolboxItem))]
public class WriteLineActivity : Activity
{
    ...
}

```

所有这些修改都完成后，就可以编译程序集，创建一个新的 workflow 项目了。要把活动添加到工具箱中，可以打开一个 workflow，显示工具箱的关联菜单，单击 **Choose Items**。



图 43-12

接着找到包含活动的程序集，把活动添加到工具箱中，结果如图 43-12 所示。图标看起来不太漂亮，但它说明我们的工作已经完成了。

下一节将在定制复合活动中再次访问 `ActivityToolboxItem`，因为这个类有一些额外的功能，将复合活动添加到设计界面上时，需要这些功能。

### 43.3.4 定制的复合活动

活动有两种主要类型，派生自 `Activity` 的活动可以看作是能从 workflow 中调用的函数。派生自 `CompositeActivity` 的活动(如 `ParallelActivity`、`IfElseActivity` 和 `ListenActivity`)是其他活动的容器，它们在设计期间的操作完全不同于简单的活动，因为它们显示在设计器的一个区域中，可以在该区域中拖放子活动。

本节将创建一个活动，称为 `DaysOfWeekActivity`。这个活动可以根据当前的日期执行 workflow 的不同部分。例如，在 workflow 中，为周末到达的订单执行的操作需要与正常工作日到达的订单不同。在这个例子中，将学习许多高级 workflow 主题，很好地理解如何用自己的复合活动扩展系统。这个例子的代码也放在 03 CustomActivities 解决方案中。

首先，创建一个定制的活动，它的一个属性默认为当前日期/时间，可以将该属性设置为另一个值，该值来自于 workflow 中的另一个活动，或把该属性设置为一个执行 workflow 时传送给 workflow 的参数。这个复合活动包含许多用户定义的分支，每个分支都包含一个枚举常量，该常量定义了执行该分支的日期。下面的代码定义了该活动及其两个分支：



```

DaysOfWeekActivity
    SequenceActivity: Monday, Tuesday, Wednesday, Thursday, Friday
    <other activites as appropriate>
    SequenceActivity: Saturday, Sunday
    <other activites as appropriate>

```

这个例子需要一个定义一周中各天的枚举，其中包含[Flags]属性(这样就不能使用 System 命名空间中定义的内置枚举 DayOfWeek 了，因为它不包含[Flags]属性)。

```

[Flags]
[Editor(typeof(FlagsEnumEditor), typeof(UITypeEditor))]
public enum WeekdayEnum : byte
{
    None = 0x00,
    Sunday = 0x01,
    Monday = 0x02,
    Tuesday = 0x04,
    Wednesday = 0x08,
    Thursday = 0x10,
    Friday = 0x20,
    Saturday = 0x40
}

```

这个类型还包含一个定制编辑器，用于根据复选框修改枚举值，其代码可以下载。

定义了枚举类型后，就可以考虑活动的主干了。定制的复合活动一般派生自 CompositeActivity 类，因为该基类定义了 Activities 属性，它是所有后续活动的集合。

```

public class DaysOfWeekActivity : CompositeActivity
{
    /// <summary>
    /// Get/Set the day of week property
    /// </summary>
    [Browsable(true)]
    [Category("Behavior")]
    [Description("Bind to a DateTime property, set a specific date time, or leave blank for DateTime.Now")]
    [DefaultValue(typeof(DateTime), "")]
    public DateTime Date
    {
        get { return (DateTime)base.GetValue(DaysOfWeekActivity.DateProperty); }
        set { base.SetValue(DaysOfWeekActivity.DateProperty, value); }
    }

    /// <summary>
    /// Register the DayOfWeek property
    /// </summary>
    public static DependencyProperty DateProperty =
        DependencyProperty.Register("Date", typeof(DateTime),
            typeof(DaysOfWeekActivity));
}

```

Date 属性提供了一般的 get 和 set 存取器，本例还添加了许多标准特性，使 Date 属性能正确显示在属性浏览器中。其代码看起来与一般的.NET 属性有所不同，因为 get 和 set 存取器没有使用标准的字段存储其值，而是使用了 DependencyProperty。

Activity 类(和这个定制活动类，因为定制活动类最终派生自 Activity)派生自 DependencyObject 类，它定义了一个键为 DependencyProperty 的字典。这种使用 get 和 set 存取



器间接得到的属性值由 WF 用于支持绑定，即把一个活动的属性链接到另一个活动的属性上。例如，我们常常要在代码中传送参数，有时是按值传送，有时是按引用传送。WF 使用绑定将属性值链接在一起，所以本例在工作流上定义了一个 `DateTime` 属性，这个活动需要在运行期间绑定到该属性值上。本章的后面将列举一个绑定的例子。

如果建立这个活动，它并没有做太多的工作，甚至不允许添加子活动，因为还没有为该活动定义 `Designer` 类。

### 1. 添加设计器

与本章前面的 `WriteLineActivity` 一样，每个活动都有一个关联的 `Designer` 类，用于修改该活动在设计期间的操作。`WriteLineActivity` 中的 `Designer` 类是空的，但对于复合活动，需要重写两个方法，以添加某些特殊的处理。

```
public class DaysOfWeekDesigner : ParallelActivityDesigner
{
    public override bool CanInsertActivities
        (HitTestInfo insertLocation, ReadOnlyCollection<Activity> activities)
    {
        foreach (Activity act in activities)
        {
            if (!(act is SequenceActivity))
                return false;
        }

        return base.CanInsertActivities(insertLocation, activitiesToInsert);
    }

    protected override CompositeActivity OnCreateNewBranch()
    {
        return new SequenceActivity();
    }
}
```

这个 `Designer` 派生自 `ParallelActivityDesigner`，它在添加子活动时提供了很好的设计操作。如果所拖放的活动不是 `SequenceActivity`，就需要将 `CanInsertActivities` 重写为返回 `false`。如果所有的活动都使用了正确的类型，就可以调用基类方法，进一步检查定制活动所允许的活动类型。

还需要重写 `OnCreateNewBranch` 方法，它在用户选择 `Add Branch` 菜单项时调用。`Designer` 类使用 `[Designer]` 属性与活动关联起来，如下面的代码所示：

```
[Designer(typeof(DaysOfWeekDesigner))]
public class DaysOfWeekActivity : CompositeActivity
{
}
```

设计期间的操作就快完成了，还需给活动添加一个派生自 `ActivityToolboxItem` 的类，来指定将该活动的一个实例从工具箱中拖出时会发生什么。默认操作是构建一个新活动，但在本例中还希望创建两个默认分支。下面的代码显示了这个工具箱选项类：

```
[Serializable]
public class DaysOfWeekToolboxItem : ActivityToolboxItem
{
}
```

```

public DaysOfWeekToolboxItem(Type t)
    : base(t)
{
    this.DisplayName = "DaysOfWeek";
}

private DaysOfWeekToolboxItem(SerializationInfo info, StreamingContext context)
{
    this.Deserialize(info, context);
}

protected override IComponent[] CreateComponentsCore(IDesignerHost host)
{
    CompositeActivity parent = new DaysOfWeekActivity();
    parent.Activities.Add(new SequenceActivity());
    parent.Activities.Add(new SequenceActivity());

    return new IComponent[] { parent };
}

```

代码修改了活动的显示名称，实现了串行化的构造函数，重写了 `CreateComponentsCore` 方法。

这个方法在拖放操作的最后调用，在该方法中构建了 `DayOfWeekActivity` 的一个实例。在代码中还建立了两个子系列活动，为活动的用户提供了更好的设计体验。几个内置活动与 `DayOfWeekActivity` 有相同的作用，如把一个 `IfElseActivity` 实例拖放到设计界面上时，它的工具箱选项类也会添加两个分支。将 `ParallelActivity` 实例拖放到工作流上，也会添加两个分支。

串行化构造函数和 `[Serializable]` 属性是所有派生自 `ActivityToolboxItem` 的类都需要的。

最后，将这个工具箱选项类与活动关联起来：

```

[Designer(typeof(DaysOfWeekDesigner))]
[ToolboxItem(typeof(DaysOfWeekToolboxItem))]
public class DaysOfWeekActivity : CompositeActivity
{
}

```

之后，活动的 UI 就接近完成了，如图 43-13 所示。

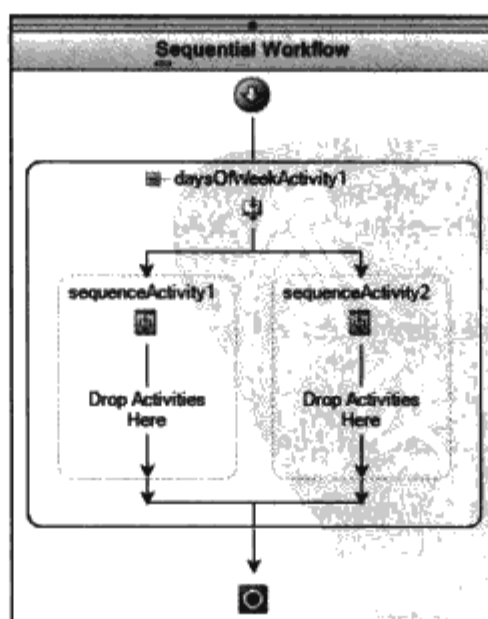


图 43-13

现在需要在图 43-13 的每个系列活动上定义一个属性，使用户能指定分支执行的日期。这在 Windows 工作流中有两种方式：可以创建 `SequenceActivity` 的一个子类，在该子类中定义该属性；也可以使用依赖属性的另一个特性 `Attached Properties`。

这里采用第二种方式，因为它不需要建立子类，但可以有效地扩展系列活动，而无需活动的源代码。

## 2. 附带属性

在注册依赖属性时，可以调用 `RegisterAttached` 方法，创建一个附带的属性。附带属性在一个类中定义，但在另一个类中显示。所以这里在 `DayOfWeekActivity` 中定义一个属性，但这个属性实际上显示在 UI 上，作为系列活动的一个附带属性。

下面的代码显示了一个 `WeekdayEnum` 类型的属性 `Weekday`，它会添加到复合活动的各系列活动中。

```
public static DependencyProperty WeekdayProperty =
    DependencyProperty.RegisterAttached("Weekday",
        typeof(WeekdayEnum), typeof(DaysOfWeekActivity),
        new PropertyMetadata(DependencyPropertyOptions.Metadata));
```

最后一行代码指定了属性的额外信息，这个例子指定它是一个 `Metadata` 属性。

`Metadata` 属性与一般的属性不同，它只能在运行期间读取。`Metadata` 属性类似于 C# 中的常量声明。在程序执行过程中不能修改常量，所以在执行工作流时也不能修改 `Metadata` 属性。

在这个例子中，要指定执行活动的日期，所以在设计器中将这个字段设置为“Saturday, Sunday”。在工作流的代码中，应有如下的声明(代码已重新设置了格式，以适应页面的大小)：

```
this.sequenceActivity1.SetValue
    (DaysOfWeekActivity.WeekdayProperty,
    ((WeekdayEnum)((WeekdayEnum.Sunday | WeekdayEnum.Saturday))));
```

除了定义依赖属性之外，还需要利用方法在活动中获取和设置这个值。这些方法一般定义为复合活动上的静态方法，如下面的代码所示：

```
public static void SetWeekday(Activity activity, object value)
{
    if (null == activity)
        throw new ArgumentNullException("activity");
    if (null == value)
        throw new ArgumentNullException("value");
    activity.SetValue(DaysOfWeekActivity.WeekdayProperty, value);
}

public static object GetWeekday(Activity activity)
{
    if (null == activity)
        throw new ArgumentNullException("activity");

    return activity.GetValue(DaysOfWeekActivity.WeekdayProperty);
}
```

还需要修改另外两个地方，才能使这个额外的属性显示为 `SequenceActivity` 的附带属性。第一处修改是创建一个扩展器提供程序，它告诉 Visual Studio 在系列活动中包含额外的属性。



第二处修改是注册这个提供程序，具体操作是重写活动设计器的 Initialize 方法，添加如下代码：

```
protected override void Initialize(Activity activity)
{
    base.Initialize(activity);

    IExtenderListService iels = base.GetService(typeof(IExtenderListService))
        as IExtenderListService;

    if (null != iels)
    {
        bool extenderExists = false;

        foreach (IExtenderProvider provider in iels.GetExtenderProviders())
        {
            if (provider.GetType() == typeof(WeekdayExtenderProvider))
            {
                extenderExists = true;
                break;
            }
        }

        if (!extenderExists)
        {
            IExtenderProviderService ieps =
                base.GetService(typeof(IExtenderProviderService))
                    as IExtenderProviderService;
            if (null != ieps)
                ieps.AddExtenderProvider(new WeekdayExtenderProvider());
        }
    }
}
```

在上述代码中对 GetService 的调用允许定制的设计器查询主机(这里是 Visual Studio)设置的服务。在 Visual Studio 中查询 IExtenderListService，它提供了枚举所有扩展器提供程序的一种方式。如果没有找到 WeekdayExtenderProvider 服务的实例，就查询 IExtender ProviderService，添加一个新的提供程序。

扩展器提供程序的代码如下所示：

```
[ProvideProperty("Weekday", typeof(SequenceActivity))]
public class WeekdayExtenderProvider : IExtenderProvider
{
    bool IExtenderProvider.CanExtend(object extendee)
    {
        bool canExtend = false;

        if ((this != extendee) && (extendee is SequenceActivity))
        {
            Activity parent = ((Activity)extendee).Parent;

            if (null != parent)
                canExtend = parent is DaysOfWeekActivity;
        }

        return canExtend;
    }

    public WeekdayEnum GetWeekday(Activity activity)
    {
        WeekdayEnum weekday = WeekdayEnum.None;
    }
}
```

```
        Activity parent = activity.Parent;

        if ((null != parent) && (parent is DaysOfWeekActivity))
            weekday = (WeekdayEnum)DaysOfWeekActivity.GetWeekday(activity);

        return weekday;
    }

    public void SetWeekday(Activity activity, WeekdayEnum weekday)
    {
        Activity parent = activity.Parent;

        if ((null != parent) && (parent is DaysOfWeekActivity))
            DaysOfWeekActivity.SetWeekday(activity, weekday);
    }
}
```

扩展器提供程序用它提供的属性来标识，对于这些属性，它都必须提供公共方法 `Get<Property>` 和 `Set<Property>`。这些方法的名称必须匹配属性的名称，并带有相应的 `Get` 或 `Set` 前缀。

对设计器进行了上述修改，添加了扩展器提供程序之后，单击设计器中的一个系列活动时，就会在 Visual Studio 中看到属性，如图 43-14 所示。

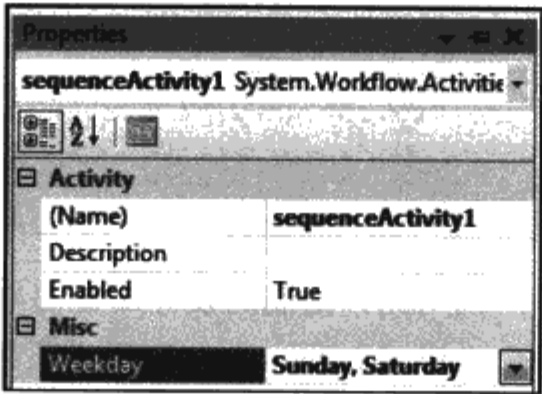


图 43-14

扩展器提供程序用于 .NET 中的其他特性，其中一个常见的特性是在 Windows 窗体项目中给控件添加工具提示。在给窗体添加工具提示控件时，会注册一个扩展器，为窗体上的每个控件添加 `Tooltip` 属性。

### 43.4 工作流

到目前为止，本章主要探讨了活动，但没有讨论工作流。工作流只是一个活动列表，实际上工作流本身是活动的另一个类型。使用这个模型简化了运行库引擎，因为运行库引擎只需知道如何执行一种对象——派生自 `Activity` 类的对象。

每个工作流实例都用其 `InstanceId` 属性唯一地标识，`InstanceId` 属性是一个可以由运行库指定的 `Guid`，这个 `Guid` 可以由代码提供给运行库，使用 `Guid` 的目的一般是将运行的工作流实例与在工作流外部维护的其他数据关联起来，例如数据库中的一行。使用 `WorkflowRuntime` 类的 `GetWorkflow(Guid)` 方法可以访问特定的工作流实例。

WF 中有两种工作流：系列工作流和状态机工作流。



### 43.4.1 系列工作流

系列工作流中的根活动是 `SequenceWorkflowActivity`。这个类派生自前面介绍的 `SequenceActivity`，它定义了两个需要提供处理程序的事件 `Initialized` 和 `Completed`。

系列工作流首先执行第一个子活动，之后执行第二个子活动，直到所有的子活动都执行完毕为止。在两种情况下，工作流不会继续执行这些活动：一种是执行工作流的过程中引发了异常，另一种是工作流中存在 `TerminateActivity`。

工作流不会在所有的情况下都执行，例如，当遇到 `DelayActivity` 时，工作流就进入等待状态，如果定义了工作流持续服务，工作流就会从内存中删除。工作流的持续在本章后面的“持续服务”中介绍。

### 43.4.2 状态机工作流

如果进程处于几种状态中的一种，只要给工作流传送数据，就可以使进程从一种状态进入另一种状态，此时就可以使用状态机工作流。

一个例子是工作流用于对大厦的访问控制。此时，可以建立一个 `door` 类，它可以关闭或打开，再建立一个 `lock` 类，它可以打开或锁上。在启动系统(或大厦)时，就进入了一个已知状态——为了便于讨论，假定所有的门都是关上的，且已上锁，所以某个门的状态是 `closed` 和 `locked`。

当雇员从前门输入其访问代码时，会把一个事件传送给工作流，其中包含的细节有输入的代码和用户 ID。接着需要访问数据库，提取信息，例如是否允许这个人在给定的这个时间打开选定的门，假定该访问获得了许可，工作流就会从其初始状态改为 `closed` 和 `unlocked` 状态。

在这个状态下，有两种可能的结果——雇员打开了门(这是因为门安装了打开/关闭感应器)；或者雇员发现把东西落在了汽车上，所以不打算进门了，于是，在过了延迟时间后要重新锁上门。因此这个门返回 `closed` 和 `locked` 状态，或进入 `open` 和 `unlocked` 状态。

现在假定雇员进入了大厦，并关上了门——接着要从 `open` 和 `unlocked` 状态进入 `closed` 和 `unlocked` 状态，在过了延迟时间后要重新锁上门。如果门处于 `open` 和 `unlocked` 状态的时间很长，还要引发警报。

在 Windows Workflow 中为这种情形建模是很简单的：需要定义系统的状态，再定义可以使工作流从一种状态进入另一种状态的事件。表 43-2 描述了系统的状态，提供了切换状态的细节以及改变状态的输入(外部或内部输入)。

表 43-2

状 态	切 换
Closed Locked	<p>这是系统的初始状态。</p> <p>为了响应用户的刷卡动作(且成功通过了访问检查)，状态会改为 <code>Closed Unlocked</code>，门锁会通过电子方式打开</p>
Closed Unlocked	<p>门处于这种状态时，会发生如下两个事件中的一个：</p> <p>(1) 用户打开了门——将状态切换为 <code>Closed Locked</code></p> <p>(2) 时间到，门返回 <code>Closed Locked</code> 状态</p>

(续表)

状 态	切 换
Open Unlocked	在这个状态下，工作流只能切换到 Closed Unlocked 状态
Fire Alarm	这是工作流的最后一个状态，其他三种状态都可以切换到这种状态

可以添加到系统中的另一个特性是响应火警。当发出警报时，应打开所有的房门，让所有人离开大厦，消防员进入大厦。可以把这个状态建立为房门工作流的最后一个状态，因为一旦取消警报，系统就可以从这个状态切换为其他状态。

图 43-15 中的工作流定义了这个状态机，显示了工作流的状态，说明了系统中可以进行的 状态切换。

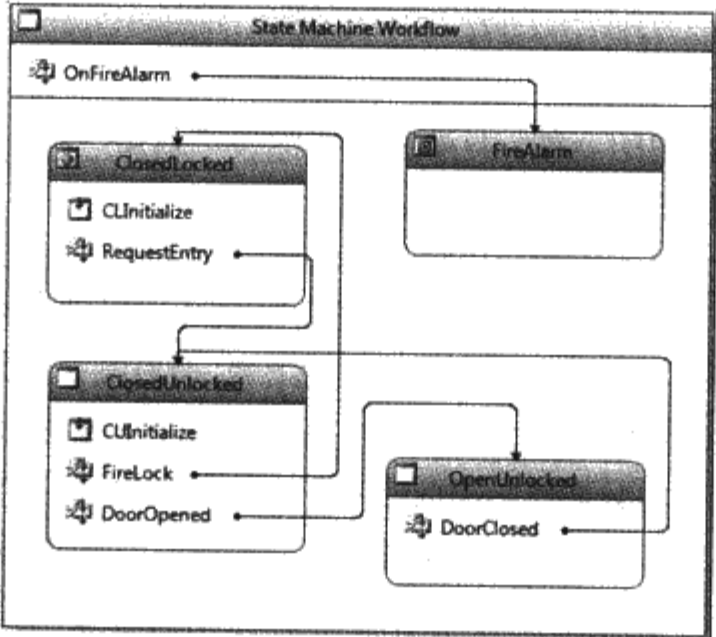


图 43-15

工作流的初始状态是用 ClosedLocked 活动建立的。这由一些初始化代码实现(锁上门)，然后是一个基于事件的活动，它在等待外部的一个事件，在这个例子中，是雇员输入了大厦的访问代码。状态图中显示的每个活动都由系列工作流组成，所以为系统的初始化定义一个工作流 (CLInitialize)，再定义一个工作流 RequestEntry，该工作流会响应当雇员输入其 PIN 时引发的外部事件。定义好的 RequestEntry 工作流如图 43-16 所示。

每个状态都由许多子工作流组成，每个子工作流的开始都有一个事件驱动的活动，之后是任意多个其他活动，它们构成了状态中的处理代码。在图 43-16 中，开头有一个 HandleExternalEventActivity，它在等待输入 PIN。之后工作流将检查 PIN，如果它是有效的，工作流就切换到 ClosedUnlocked 状态。

Closed Unlocked 状态由两个工作流组成，一个工作流响应门被打开的事件，将工作流切换到 OpenUnlocked 状态。另一个工作流包含一个延迟活动，用于将状态改为 Closed- Locked。状态驱动的活动与本章前面的 ListenActivity 采用相同的方式工作——状态由许多事件驱动的工作流组成，当发生一个事件时，就执行其中一个工作流。

为了支持工作流，需要引发系统中的事件，实现状态的改变。为此，需要使用一个接口和该接口的实现代码。这对对象称为外部服务。本章的后面将描述用于这个状态机的接口。

上述状态机例子的代码在 04 StateMachine 解决方案中，其中还包含一个用户界面，在该用

户界面上可以输入 PIN，通过两扇门中的一扇访问大厦。

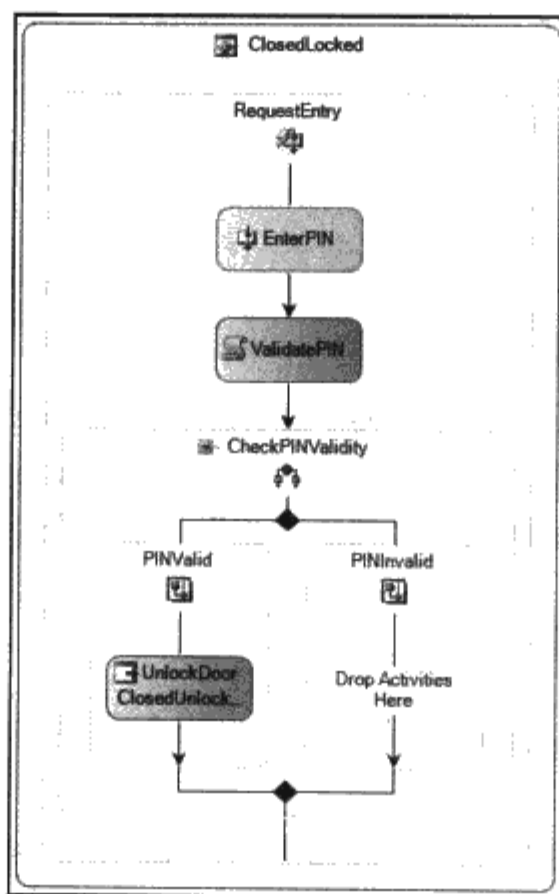


图 43-16

### 43.4.3 给工作流传送参数

工作流一般需要一些数据才能执行，例如订单处理工作流需要一个订单 ID，付费处理工作流需要一个顾客账户 ID，或者其他数据项。

工作流的参数传送机制与标准的.NET 类有所不同，在标准的.NET 类中，一般在方法调用中传送参数。而对于工作流，传送参数时，要把这些参数存储在一个名称/值对的字典中，在构建工作流时，要传送这个字典。

WF 安排工作流的执行时，会使用这些名称-值对设置工作流实例的公共属性。每个参数名都要用工作流的公共属性来检查。如果找到了匹配，就调用属性设置器，把该参数的值传送给设置器。如果将一个名称-值对添加到字典中，在该字典中，名称并不对应工作流上的一个属性，则在试图构建这个工作流时，会抛出一个异常。

例如，下面的工作流将 OrderID 属性定义为一个整数：

```

public class OrderProcessingWorkflow: SequentialWorkflowActivity
{
    public int OrderID
    {
        get { return _orderID; }
        set { _orderID = value; }
    }

    private int _orderID;
}
  
```

下面的代码说明了如何将订单 ID 参数传送给工作流的一个实例：

```

WorkflowRuntime runtime = new WorkflowRuntime ();

Dictionary<string,object> parms = new Dictionary<string,object>();
parms.Add("OrderID", 12345) ;

WorkflowInstance instance = runtime.CreateWorkflow(
    typeof(OrderProcessingWorkflow), parms);

instance.Start();

... Other code

```

在上面的示例代码中，构建了一个 `Dictionary<string, object>`，它包含要传送给工作流的参数，在构建工作流时要使用这个字典。上面的代码包含 `WorkflowRuntime` 和 `WorkflowInstance` 类，这里没有介绍它们，本章后面的“存储工作流”一节将介绍它们。

#### 43.4.4 从工作流中返回结果

工作流的另一个常见要求是返回输出参数，它们可能用于将数据记录到数据库或其他永久存储器中。

工作流是由工作流运行库执行的，所以调用工作流不是只使用标准的方法调用机制，而需要创建一个工作流实例，启动这个实例，等待它的完成。工作流完成后，工作流运行库就会引发 `WorkflowCompleted` 事件，给它传送工作流的相关信息，并获取从该工作流中返回的输出。

所以，要获取从该工作流中返回的输出参数，需要将一个事件处理程序关联到 `WorkflowCompleted` 事件上，该处理程序可以从工作流中提取输出参数。下面的代码是这方面的一个例子：

```

using(WorkflowRuntime workflowRuntime = new WorkflowRuntime())
{
    AutoResetEvent waitHandle = new AutoResetEvent(false);
    workflowRuntime.WorkflowCompleted +=
        delegate(object sender, WorkflowCompletedEventArgs e)
        {
            waitHandle.Set();
            foreach (KeyValuePair<string, object> parm in e.OutputParameters)
            {
                Console.WriteLine("{0} = {1}", parm.Key, parm.Value);
            }
        };

    WorkflowInstance instance = workflowRuntime.CreateWorkflow(typeof(Workflow1));
    instance.Start();

    waitHandle.WaitOne();
}

```

我们把一个委托关联到 `WorkflowCompleted` 事件上，在这个委托中，迭代传送给委托的 `WorkflowCompletedEventArgs` 类中的 `OutputParameters` 集合，并在控制台上显示输出参数。这个集合包含工作流的所有公共属性。工作流没有特定的输出参数。

43.4.5 将参数绑定到活动上

理解了如何将参数传送给工作流后，还需要明白如何把这些参数链接到活动上。这通过绑定机制来实现。在前面定义的 DaysOfWeekActivity 中，有一个 Date 属性，它可以硬编码，也可以绑定到工作流中的另一个值上。Bindable 属性显示在 Visual Studio 的属性表中，如图 43-17 所示。属性名右边的图标表示，这是一个可绑定的属性。

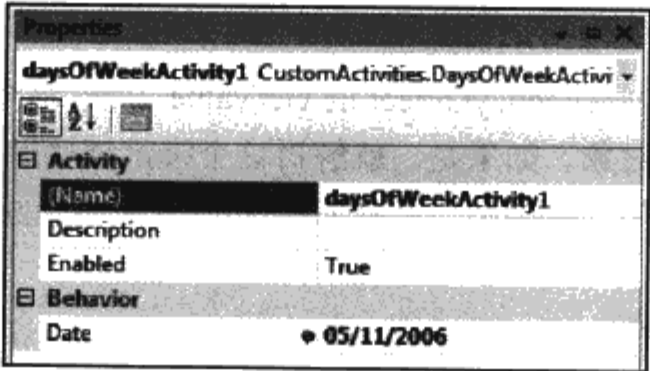


图 43-17

双击绑定图标，会显示如图 43-18 所示的对话框。它允许选择一个合适的属性，来链接 Date 属性。

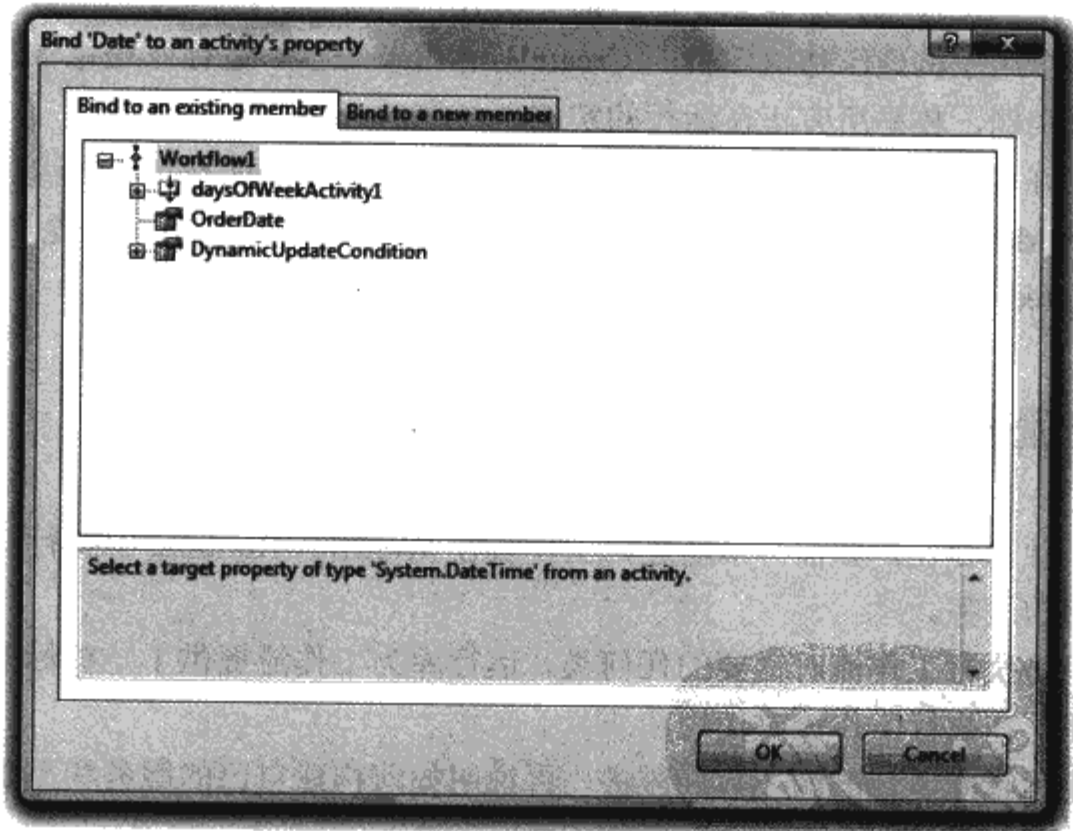


图 43-18

在图 43-18 中，选择了工作流的 OrderDate 属性(它定义为一个普通的.NET 属性，如前面的代码所示)。任何 Bindable 属性都可以绑定到定义活动的工作流的属性上，或者绑定到工作流中位于当前活动上方的活动的属性上。注意，被绑定的属性的数据类型必须匹配要绑定的属性的数据类型，该对话框不允许绑定不匹配的类型。

下面再次列出 Date 属性的代码，说明绑定的工作方式，并在后面详细解释。



```
public DateTime Date
{
    get { return (DateTime)base.GetValue(DaysOfWeekActivity.DateProperty); }
    set { base.SetValue(DaysOfWeekActivity.DateProperty, value); }
}
```

在绑定工作流中的一个属性时，会在后台构建一个 `ActivityBind` 类型的对象，它就是存储在依赖属性中的值。所以，需要给属性设置器传送一个 `ActivityBind` 类型的对象，它存储在这个活动的属性字典中。这个 `ActivityBind` 对象包含的数据描述了要绑定的活动和该活动中要在运行期间使用的属性。

在读取属性值时，调用 `DependencyObject` 的 `GetValue` 方法，这个方法会检查底层的属性值，确定它是否是 `ActivityBind` 对象。如果是，就解析这个绑定链接的活动，从该活动中读取属性值。但是，如果绑定的值是另一种类型，就从 `GetValue` 方法中返回该对象。

## 43.5 工作流运行库

为了启动工作流，需要创建 `WorkflowRuntime` 类的一个实例。这一般在应用程序中创建，这个对象通常定义为应用程序的一个静态成员，以便在应用程序的任意地方访问它。

在启动运行库时，它会从永久存储器中读取应用程序上次执行的工作流实例，再次加载它们。这需要使用一个持续服务，详见本节后面的内容。

运行库有 6 个用于构建工作流实例的 `CreateWorkflow` 方法。运行库还包含几个方法，可以重新加载工作流实例，枚举所有正在运行的实例。

运行库还有许多在执行工作流时引发的事件，例如 `WorkflowCreated`(在构建新的工作流实例时引发)、`WorkflowIdled`(在工作流等待输入时引发，例如前面的费用处理例子)和 `WorkflowCompleted`(在工作流完成时引发)。

## 43.6 工作流服务

工作流不能独自存在，如前一节所述，工作流在 `WorkflowRuntime` 中执行，这个运行库提供了执行工作流的服务。

该服务可以是执行工作流时需要的任何类。运行库为工作流提供了一些标准服务，也可以构建自己的服务，在执行工作流时使用。

本节将描述运行库提供的两个标准服务，再说明如何构建自己的服务和需要这么做的一些场合。

在执行活动时，要通过 `Execute` 方法的 `ActivityExecutionStatus` 参数给活动传送一些相关信息。

```
protected override ActivityExecutionStatus Execute
(ActivityExecutionContext executionContext)
{
    ...
}
```

可以在这个环境参数上使用的一个方法是 `GetService<T>`，在下面的代码中，它用于访问与工作流运行库关联的一个服务：

```
protected override ActivityExecutionStatus Execute
(ActivityExecutionContext executionContext)
{
    ICustomService myService = executionContext.GetService<ICustomService>();
    ... Do something with the service
}
```

在调用 `StartRuntime` 方法之前，运行库保存的服务会添加到运行库中，如果试图将服务添加到已启动的运行库中，就会引发一个异常。

有两种方式可以将服务添加到运行库中：可以在代码中构建服务，再调用 `AddService` 方法将它们添加到运行库中；也可以在应用程序配置文件中定义服务，这些服务构建后会自动添加到运行库中。

下面的代码说明了如何在代码中将服务添加到运行库中——所添加的服务将在本节的后面介绍。

```
using (WorkflowRuntime workflowRuntime = new WorkflowRuntime())
{
    workflowRuntime.AddService(
        new SqlWorkflowPersistenceService(conn, true, new TimeSpan(1,0,0),
                                           new TimeSpan(0,10,0)));
    workflowRuntime.AddService(new SqlTrackingService(conn));
    ...
}
```

这段代码构建了 `SqlWorkflowPersistenceService` 的实例，运行库使用这些实例存储工作流的状态。这段代码还构建了 `SqlTrackingService` 的一个实例，它记录了工作流执行时发生的事件。

要使用应用程序配置文件创建服务，需要为工作流运行库添加一个处理程序段，然后将服务添加到这个段上，如下所示：

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="WF"
      type="System.Workflow.Runtime.Configuration.WorkflowRuntimeSection,
      System.Workflow.Runtime, Version=3.0.00000.0, Culture=neutral,
      PublicKeyToken=31bf3856ad364e35" />
  </configSections>
  <WF Name="Hosting">
    <CommonParameters/>
    <Services>
      <add type="System.Workflow.Runtime.Hosting.SqlWorkflowPersistenceService,
        System.Workflow.Runtime, Version=3.0.00000.0, Culture=neutral,
        PublicKeyToken=31bf3856ad364e35"
        connectionString="Initial Catalog=WF;Data Source=.;
        Integrated Security=SSPI;"
        UnloadOnIdle="true"
        LoadIntervalSeconds="2"/>
      <add type="System.Workflow.Runtime.Tracking.SqlTrackingService,
        System.Workflow.Runtime, Version=3.0.00000.0, Culture=neutral,
        PublicKeyToken=31bf3856ad364e35"
```

```

        connectionString="Initial Catalog=WF;Data Source=.;
        Integrated Security=SSPI;"
        UseDefaultProfile="true"/>
    </Services>
</WF>
</configuration>

```

在配置文件中, 添加了 WF 处理程序段(这个名称并不重要, 但必须匹配其后配置段的名称), 之后为这个段创建需要的数据项。<Services>元素可以包含一个数据项列表, 其中包含一个.NET 类型和运行库构建服务时传送给服务的参数。

要从应用程序配置文件中读取配置设置, 可以调用运行库上的另一个构造函数, 如下所示:

```

using(WorkflowRuntime workflowRuntime = new WorkflowRuntime("WF"))
{
    ...
}

```

这个构造函数会实例化配置文件中定义的每个服务, 并把它们添加到运行库上的服务集合中。

下一节将介绍 WF 提供的一些标准服务。

### 43.6.1 持续服务

执行工作流时, 可能会进入等待状态——在执行延迟活动时, 或者在监听活动中等待外部输入时, 就会进入等待状态。此时, 工作流处于空闲状态, 等待继续执行。

假定先在服务器上执行 1000 个工作流, 之后每个工作流实例都进入空闲状态。此时, 不需要将这些实例的数据保存在内存中, 所以最好能卸载工作流, 释放它使用的资源。持续服务就用于这个目的。

当工作流进入空闲状态时, 工作流运行库会检查是否存在一个派生自 `WorkflowPersistenceService` 类的服务。如果存在, 就给它传送工作流实例, 接着服务就可以获得工作流的当前状态, 并将它存储在永久存储介质上。可以把工作流的状态存储在磁盘的一个文件或数据库中, 如 SQL Server 中。

工作流库包含持续服务的实现代码, 持续服务可以把数据存储在 SQL Server 数据库中, 即 `SqlWorkflowPersistenceService`。为了使用这个服务, 需要对 SQL Server 实例运行两个脚本, 其中一个脚本构建模式, 另一个脚本创建由持续服务使用的存储过程。这些脚本默认存储在目录 `C:\Windows\Microsoft.NET\Framework\v3.5\WindowsWorkflowFoundation\SQL\EN` 下。

在数据库上执行的脚本是 `SqlPersistenceService_Schema.sql` 和 `SqlPersistenceService_Logic.sql`。这些脚本需要按顺序执行: 先执行模式文件, 再执行逻辑文件。SQL 持续服务的模式包含两个表: `InstanceState` 和 `CompletedScope`: 这些都是不透明的表, 不在 SQL 持续服务的外部使用。

工作流在空闲时, 其状态是用二进制串行化机制来串行化, 接着把这些数据插入 `InstanceState` 表。重新激活工作流时, 从这一行中读取状态, 用于重新构建工作流实例。这一行用工作流实例 ID 来标识, 并在工作流完成时从数据库中删除。

SQL 持续服务可以由多个运行库同时使用——它实现了锁定机制, 所以一次只能由工作流

运行库的一个实例访问工作流。如果多个服务器都使用同一个永久存储器运行工作流，这个锁定机制就是非常有价值的。

为了查看添加到永久存储器中的内容，需要构建一个新的工作流项目，给运行库添加 `SqlWorkflowPersistenceService` 的一个实例。下面是使用声明代码的一个例子：

```
using(WorkflowRuntime workflowRuntime = new WorkflowRuntime())
{
    workflowRuntime.AddService(
        new SqlWorkflowPersistenceService(conn, true, new TimeSpan(1,0,0),
                                           new TimeSpan(0,10,0)));
    // Execute a workflow here...
}
```

接着，如果构建一个包含 `DelayActivity` 的工作流，将延迟时间设置为 10 秒，就可以查看 `InstanceState` 表中存储的数据。05 `WorkflowPersistence` 示例包含上述代码，其延迟时间设置为 20 秒。

持续服务的构造函数的参数如表 43-3 所示。

表 43-3

参 数	说 明	默 认 值
ConnectionString	由持续服务使用的数据库连接字符串	None
UnloadOnIdle	确定工作流在空闲时是否卸载。它应总是设置为 true，否则工作流就不会继续下去	False
InstanceOwnershipDuration	定义运行库拥有工作流实例的时间长度	None
LoadingInterval	在给数据库存储更新的永久记录时使用的时间间隔	2 分钟

这些值也可以在配置文件中定义。

43.6.2 跟踪服务

工作流执行时，需要记录执行了哪些活动。对于 `IfElseActivity` 和 `ListenActivity` 等复合活动，则执行其分支。这些数据可以用作工作流实例的一种审查踪迹，在以后的某个日期查看，以证明执行了哪些活动，在工作流中使用了什么数据。跟踪服务可以用于这类记录操作，并可以配置，根据需要记录尽可能少或尽可能多的工作流信息。

在 WF 中，跟踪服务实现为一个抽象类 `TrackingService`，很容易用自己的跟踪服务替换标准的跟踪代码。在工作程序集中，跟踪服务有一个具体的实现代码，即 `SqlTrackingService`。

要记录工作流的状态数据，就需要定义一个 `TrackingProfile`。它定义了应记录什么事件，例如，可以只记录工作流的开头和结束，忽略运行实例的其他所有数据。更一般的情况是，记录工作流的所有事件和其中的每个活动，以提供工作流的执行情况的完整描述。

运行库引擎安排工作流的执行时，引擎会检查工作流跟踪服务是否存在。如果找到了一个跟踪服务，就会要求该服务为正在执行的工作流提供一个跟踪描述，接着使用它记录工作流和活动数据。还可以定义用户跟踪数据，把它们存储在跟踪数据库中，这些操作不需要改变模式。



跟踪配置类如图 43-19 所示。这个类包含了活动、用户和工作流跟踪点的集合属性。跟踪点是一个对象(如 WorkflowTrackPoint)，它一般定义了一个匹配位置和一些额外的数据，在单击这个跟踪点时，就会记录这些额外的数据。匹配位置指定这个跟踪点在什么地方有效。例如，可以定义一个 WorkflowTrackPoint，它记录了创建工作流时的一些数据，再定义一个 WorkflowTrackPoint，来记录完成工作流时的一些数据。

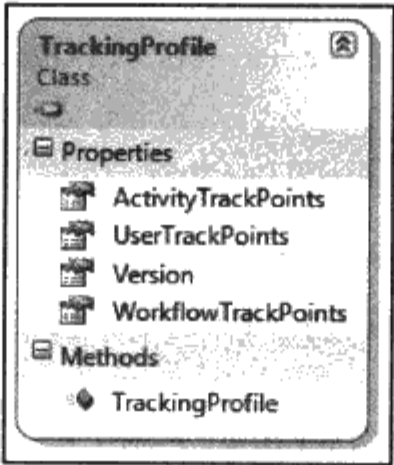


图 43-19

记录了这些数据后，就需要显示工作流的执行路径了，如图 43-20 所示。该图显示了执行的工作流，每个运行的服务都包含一个说明其执行的图标。这些数据从工作流实例的跟踪库中读取。

为了读取 SqlTrackingService 存储的数据，可以直接在 SQL 数据库上执行查询。Microsoft 还为此在 System.Workflow.Runtime.Tracking 命名空间中提供了 SqlTrackingQuery 类。下面的示例说明了如何检索在两个日期之间跟踪的所有工作流：

```

public IList<SqlTrackingWorkflowInstance> GetWorkflows
(DateTime startDate, DateTime endDate, string connectionString)
{
    SqlTrackingQuery query = new SqlTrackingQuery (connectionString);

    SqlTrackingQueryOptions queryOptions = new SqlTrackingQueryOptions();
    query.StatusMinDateTime = startDate;
    query.StatusMaxDateTime = endDate;

    return (query.GetWorkflows (queryOptions));
}
  
```

这段代码使用了 SqlTrackingQueryOptions 类，它定义了查询参数。可以定义这个类的其他属性，进一步约束要检索的工作流。

在图 43-20 中，可以看到所有的活动都执行了。但如果工作流仍在运行，或者在工作流中做了一些决策，要在执行过程中采用不同的路径，图 43-20 就不是这样了。跟踪数据包含执行了哪些活动等信息，这些数据与生成图 43-20 的活动相关。还可以在工作流执行时提取这些数据，用于为工作流的执行流生成审查轨迹。



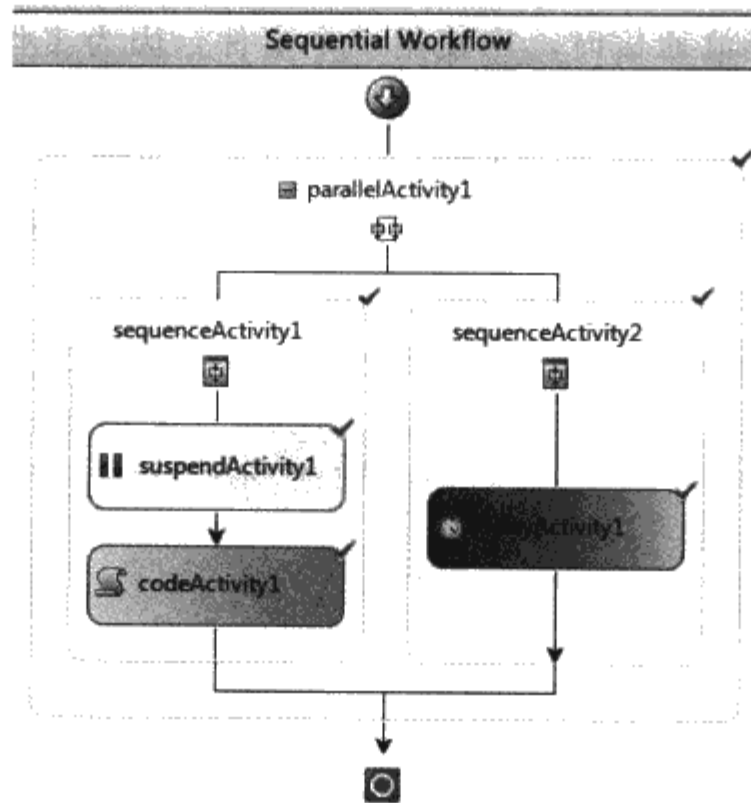


图 43-20

### 43.6.3 定制服务

除了持续服务和跟踪服务等内置服务之外，还可以在 `WorkflowRuntime` 维护的服务集合中添加自己的对象。这些服务一般用接口和实现代码来定义，以便在不记录工作流的情况下替换该服务。

本章前面的状态机利用了下面的接口：

```
[ExternalDataExchange]
public interface IDoorService
{
    void LockDoor();
    void UnlockDoor();

    event EventHandler<ExternalDataEventArgs> RequestEntry;
    event EventHandler<ExternalDataEventArgs> OpenDoor;
    event EventHandler<ExternalDataEventArgs> CloseDoor;
    event EventHandler<ExternalDataEventArgs> FireAlarm;

    void OnRequestEntry(Guid id);
    void OnOpenDoor(Guid id);
    void OnCloseDoor(Guid id);
    void OnFireAlarm();
}
```

这个接口中的方法由工作流用于调用服务，服务引发的事件由工作流使用。使用特性 `ExternalDataExchange` 是告诉工作流运行库，这个接口用于在运行的工作流和服务的实现代码之间通信。

在状态机中，有许多 `CallExternalMethodActivity` 的实例，它们用于在这个外部的接口上调用方法。例如，当门上锁或不上锁时，工作流需要执行对 `UnlockDoor` 或 `LockDoor` 方法的调用，服务的响应是发出一个命令，锁上门，或打开门。

当服务需要与工作流通信时，应使用一个事件，因为工作流运行库也包含一个 `ExternalDataExchangeService` 服务，它是这些事件的代理。这个代理在引发事件时使用，因为在引发事件时，工作流可能没有加载到内存中，所以事件会先路由给外部的数据交换服务，它检查工作流是否已加载，如果没有加载，就从永久存储器中重新加载工作流，再把事件传送给工作流。

下面的代码构建了 `ExternalDataExchangeService`，还为该服务定义的事件构建了代理：

```
WorkflowRuntime runtime = new WorkflowRuntime();
ExternalDataExchangeService edes = new ExternalDataExchangeService();

runtime.AddService(edes);
DoorService service = new DoorService();
edes.AddService(service);
```

这段代码建立了外部数据交换服务的一个实例，把它添加到运行库中，接着创建 `DoorService` 的一个实例(它本身实现了 `IDoorService`)，并把它添加到外部数据交换服务中。

`ExternalDataExchangeService.Add` 方法为定制服务定义的每个事件建立了一个代理，以便在事件发生之前加载已存储的工作流。如果没有把服务存储在外部数据交换服务中，在引发事件时，就不会监听这些事件了，它们也不会传送到正确的工作流。

事件使用 `ExternalDataEventArgs` 类，因为它包含接收事件的工作流实例 ID。如果需要将其他值从外部事件传送给工作流，就应从 `ExternalDataEventArgs` 中派生一个类，把这些值添加为该类的属性。

## 43.7 与 WCF 集成

.NET 3.5 中的两个新活动支持工作流和 WCF 之间的集成，这两个活动是 `SendActivity` 和 `ReceiveActivity`。`SendActivity` 称为 `CallActivity` 更恰当，因为它的工作是向 WCF 服务发送一个请求，并可以把结果显示为参数，绑定到调用工作流上。

但更有趣的是 `ReceiveActivity`。它允许工作流变成 WCF 服务的实现方式，所以现在工作流就是服务。下面例子中的服务使用一个工作流和一个新的服务测试工具，来测试服务，而无需编写独立的测试台程序。

在 VS2008 的 New Project 菜单中选择 WCF 节点，再选择 `Sequential Workflow Service Library` 项，如图 43-21 所示。

这会创建一个库，其中包含了一个工作流、一个应用程序配置文件和一个服务接口，如图 43-22 所示。

工作流显示了合同的 Hello 操作，还定义了传送给这个操作的参数属性，以及操作的返回值。接着只需添加代码，提供服务的执行操作，服务就完成了。

为此，把一个 `CodeActivity` 拖放到 `ReceiveActivity` 上，如图 43-23 所示。再双击该活动，提供服务的实现代码。

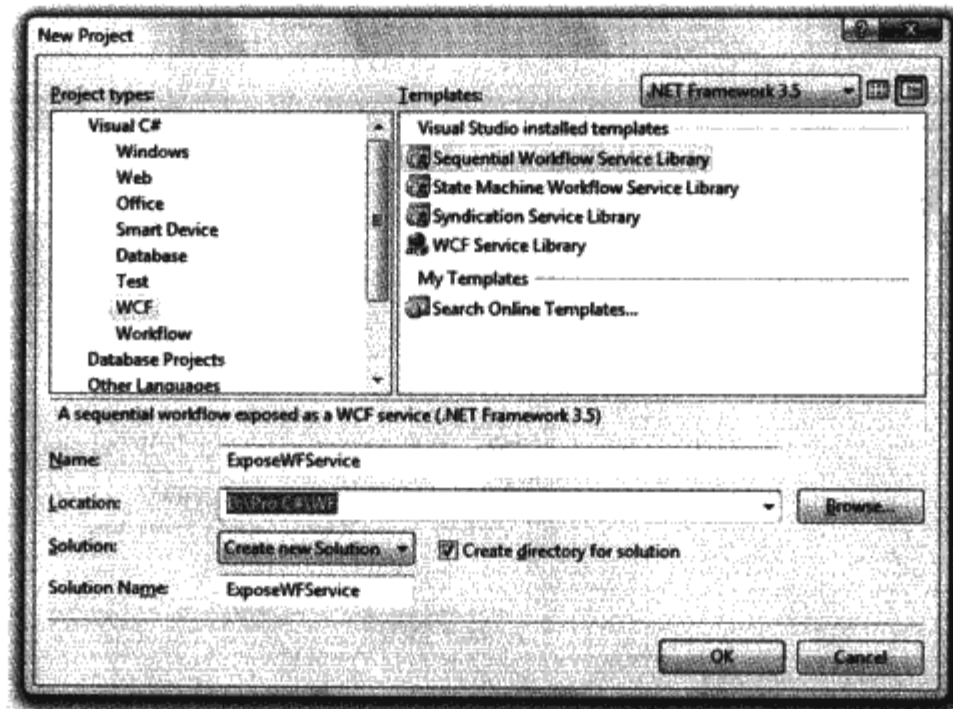


图 43-21

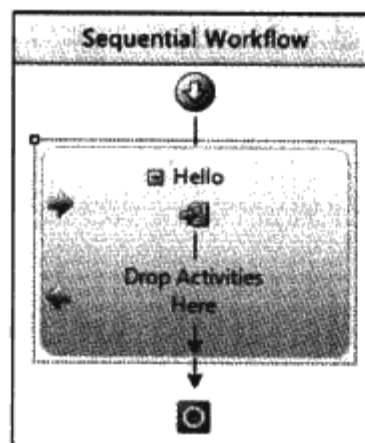


图 43-22

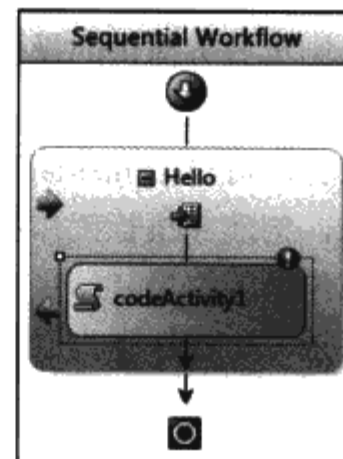


图 43-23

下面的代码就是这个服务的实现代码。

```
public sealed partial class Workflow1: SequentialWorkflowActivity
{
    public Workflow1()
    {
        InitializeComponent();
    }
    public String returnValue = default(System.String);
    public String inputMessage = default(System.String);

    private void codeActivity1_ExecuteCode(object sender, EventArgs e)
    {
        this.returnValue = string.Format("You said {0}", inputMessage);
    }
}
```

Hello 操作的服务合同包含参数 `inputMessage` 和返回值，所以它们都由工作流显示为公共字段。在代码中，把 `returnValue` 设置为一个字符串值，这就是从 WCF 服务的调用中返回的内容。

如果编译这个服务，按下 F5，就会注意到 VS2008 的另一个新特性：WCF 测试客户应用

程序，如图 43-24 所示。

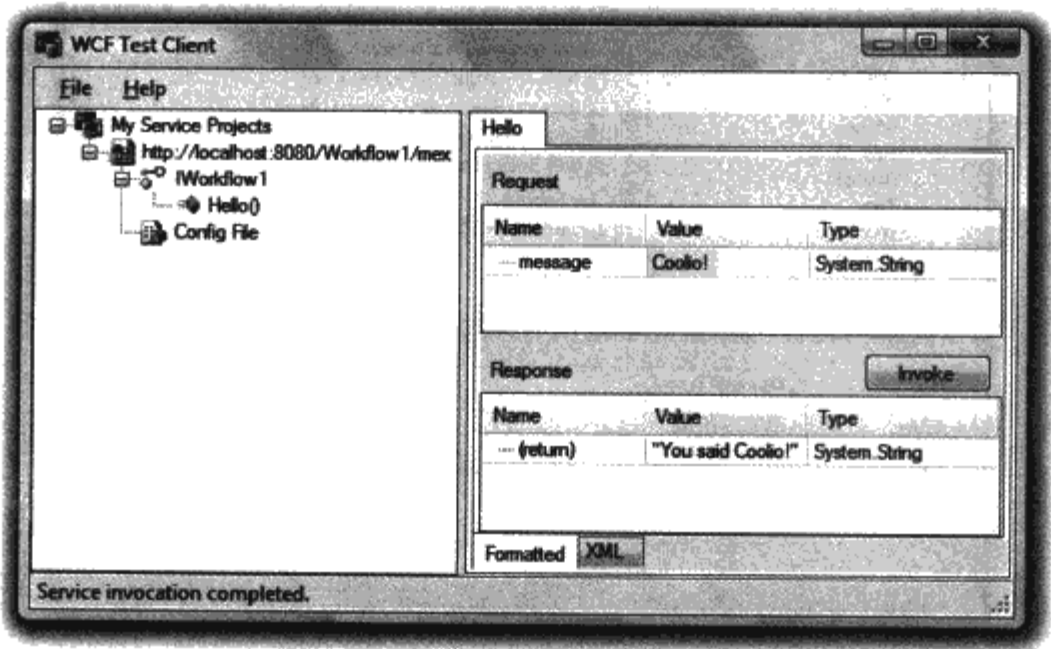


图 43-24

在这里可以浏览服务提供的操作，双击一个操作，会显示窗口的右半部分，其中列出了该服务使用的参数和返回值。

要测试该服务，可以给 message 属性输入一个值，单击 Invoke 按钮。这会通过 WCF 给服务发送一个请求，构建并执行工作流，调用代码活动，运行后台代码，最终给 WCF 测试客户应用程序返回工作流的结果。

如果要手工把工作流保存为服务，可以使用在 System.WorkflowServices 命名空间中定义的新类 WorkflowServiceHost。下面的代码显示了一个非常小的保存代码：

```
using (WorkflowServiceHost host = new WorkflowServiceHost
    (typeof(YourWorkflow)))
{
    host.Open();
    Console.WriteLine ("Press [Enter] to exit" );
    Console.ReadLine();
}
```

这里构建了 WorkflowServiceHost 的一个实例，把它传送给要执行的工作流。这类似于保存 WCF 服务时使用 ServiceHost 类。它读取配置文件，确定服务应监听哪个端点，并等待服务请求。

下一节介绍保存工作流的其他选项。

### 43.8 保存工作流

在进程中保存 WorkflowRuntime 的代码随应用程序的不同而不同。

对于 Windows 窗体应用程序或 Windows 服务，一般在应用程序的开头构建运行库，把它存储在主应用程序类的一个属性中。

为了响应应用程序中的一些输入(如用户单击了用户界面上的一个按钮)，需要建立工作流

的一个实例，在本地执行这个实例。工作流可能还需要与用户通信，例如，定义一个外部服务，在将订单发送给后端服务器之前，提示用户确认。

在 ASP.NET 中存储工作流时，一般不用消息框提示用户，而是导航到站点上请求确认的另一个页面上，再显示一个确认页面。在 ASP.NET 中存储运行库时，一般要重写 `Application_Start` 事件，建立工作流运行库的一个实例，以便在站点的其他部分访问它。运行库实例可以在一个静态属性中存储，但最好把它存储在应用程序状态中，再提供一个访问方法，从应用程序状态中提取工作流运行库，使之可用于应用程序的其他地方。

在 Windows 窗体和 ASP.NET 两种情况下，都要建立工作流运行库的一个实例，给它添加服务，如下所示：

```
WorkflowRuntime workflowRuntime = new WorkflowRuntime();
workflowRuntime.AddService(
    new SqlWorkflowPersistenceService(conn, true, new TimeSpan(1,0,0),
                                     new TimeSpan(0,10,0)));
// Execute a workflow here...
```

要执行工作流，需要使用运行库的 `CreateInstance` 方法，创建该工作流的一个实例。这个方法有许多重写版本，可用于建立基于代码的工作流实例或在 XML 中定义的工作流实例。

本章前面都把工作流看作 .NET 类，实际上这只是工作流的一个方面。还可以使用 XML 定义工作流，此时运行库会定义工作流的内存表示，在调用 `WorkflowInstance` 的 `Start` 方法时执行它。

在 Visual Studio 中，可以从 Add New Item 对话框中选择 Sequential Workflow(其代码是独立的)或 State Machine Workflow(其代码是独立的)，创建基于 XML 的工作流。这会创建一个扩展名为 .xaml 的 XML 文件，并将它加载到设计器中。

把活动添加到设计器中时，会将这些活动存储到 XML 中，元素的结构定义了活动之间的父子关系。下面的 XML 是一个简单的系列工作流，其中包含一个 `IfElseActivity` 和两个代码活动，分别用于 `IfElseActivity` 的每个分支。

```
<SequentialWorkflowActivity x:Class="DoorsWorkflow.Workflow1" x:Name="Workflow1"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/workflow">
  <IfElseActivity x:Name="ifElseActivity1">
    <IfElseBranchActivity x:Name="ifElseBranchActivity1">
      <IfElseBranchActivity.Condition>
        <CodeCondition Condition="Test" />
      </IfElseBranchActivity.Condition>
      <CodeActivity x:Name="codeActivity1" ExecuteCode="DoSomething" />
    </IfElseBranchActivity>
    <IfElseBranchActivity x:Name="ifElseBranchActivity2">
      <CodeActivity x:Name="codeActivity2" ExecuteCode="DoSomethingElse" />
    </IfElseBranchActivity>
  </IfElseActivity>
</SequentialWorkflowActivity>
```

在活动上定义的属性在 XML 中保存为特性，每个活动都保存为一个元素。从 XML 可以看出，该结构定义了父活动(如 `SequentialWorkflowActivity` 和 `IfElseActivity`)和子活动之间的关系。

执行基于 XML 的工作流与执行基于代码的工作流没有区别，只需使用 `CreateWorkflow` 方



法的一个重写版本，该版本的参数是 XMLReader 实例，接着调用 Start 方法启动该实例即可。

与基于代码的工作流相比，使用基于 XML 的工作流的一个优点是，很容易将工作流的定义存储在数据库中。之后可以在运行期间加载这个 XML，执行工作流。修改该工作流定义时，也不需要重新编译代码。

无论工作流是在 XML 中定义，还是在代码中定义，都可以在运行期间修改工作流：只需建立一个 WorkflowChanges 对象，它包含了要添加到工作流中的所有新活动。接着调用在 WorkflowInstance 类上定义的 ApplyWorkflowChanges 方法，来保存这些修改。这是非常有用的，因为业务需求时常修改，例如需要把这些修改应用于一个保险契约工作流，在续保日期的一个月之前给客户发送电子邮件，告诉客户他们的保险契约需要续保。这种修改要针对每个实例，所以，如果系统中有 100 个保险契约工作流，就需要对每个工作流进行这种修改。

## 43.9 工作流设计器

本章还有最后一个主题要讨论。用于设计工作流的工作流设计器不与 Visual Studio 关联，可以根据需要自己的应用程序中重新构建这个设计器。

这说明，可以发布一个包含工作流的系统，允许最终用户在没有 Visual Studio 的情况下定制系统。但构建设计器是相当复杂的，这个主题需要好几章的篇幅，但这超出了本章的范围。网络上有许多重新构建设计器的例子，建议读者在 <http://msdn2.microsoft.com/enus/library/aa480213.aspx> 上获得构建设计器的更多信息。

允许用户定制系统的传统方式是定义一个接口，再让客户实现这个接口，根据需要扩展处理代码。

有了 Windows 工作流，该扩展在整体上更富逻辑性，因为可以给用户提供一个空白的工作流作为模板，再提供一个工具箱，其中包含了适合于应用程序的定制活动。他们接着可以建立自己的工作流，添加他们自己编写的定制活动。

## 43.10 小结

Windows 工作流为应用程序的构建方式带来了根本性的改变。现在可以将应用程序的复杂部分都看作活动，让用户将活动拖放到工作流中，修改系统的处理方式。

几乎没有应用程序不能使用工作流，从简单的命令行工具，到包含上百个模块的最复杂的系统。现在 WCF 的通信功能和 WPF 的新 UI 功能使应用程序前进了一大步，添加 Windows 工作流使开发和配置应用程序的方式有了根本的改变。

如果只能试验 .NET Framework 3.0 中的一个新功能，最好考虑 Windows 工作流，希望工作流中的技术在不久的将来得到人们的高度关注。

下一章详细探讨企业服务。

# 第44章

## Enterprise Services

Enterprise Services 是 Microsoft 应用程序服务器技术的另一个名称, 它为分布式解决方案提供服务。Enterprise Services 基于已使用多年的 COM+ 技术。我们不但把 .NET 对象封装为 COM 对象, 以使用这些服务, 还对 .NET 进行了扩展, 使 .NET 组件可以直接利用这些服务。通过 .NET, 将很容易访问 .NET 组件的 COM+ 服务。

Enterprise Services 还可以与 WCF 集成。使用一个工具可以为服务组件自动创建 WCF 服务前端, 并从 COM+ 客户程序中调用 WCF 服务。

本章的主要内容如下:

- 何时使用 Enterprise Services
- 什么服务要与这个技术一起使用
- 如何创建服务组件来使用 Enterprise Services
- 如何部署 COM+ 应用程序
- 如何把事务处理和 Enterprise Services 一起使用
- 如何为 Enterprise Services 创建 WCF 前端
- 如何在 WCF 客户程序中使用 Enterprise Services

提示:

本章使用示例数据库 Northwind, 它可以从 Microsoft 下载页面 [www.microsoft.com/downloads](http://www.microsoft.com/downloads) 上下载。

### 44.1 概述

如果了解 Enterprise Services 的历史, 就很容易理解 Enterprise Services 的复杂性和不同的配置选项(如果解决方案的所有组件都是用 .NET 开发的, 则不需要许多配置选项)。所以本节首先介绍 Enterprise Services 的历史。之后, 概述这项技术提供的不同服务, 以探讨应用程序可以使用的特性。

本节主要内容如下:

- 历史
- 使用 Enterprise Services 的场合
- 环境
- 自动事务处理

- 分布式事务处理
- 对象池
- 基于角色的安全性
- 排队的组件
- 松耦合的事件

#### 44.1.1 Enterprise Services 简史

Enterprise Services 可以追溯到发布为一个 Windows NT 4.0 选项包的 Microsoft Transaction Server(MTS)。MTS 提供了 COM 对象的事务处理等服务,扩展了 COM。这个服务可以通过配置元数据来使用:组件的配置定义了是否需要事务处理。有了 MTS,就不再需要编程处理事务了。但是,MTS 有一个重要缺陷:COM 不是可扩展的,所以 MTS 在进行扩展时,要重写 COM 组件注册配置,把组件的实例化指向 MTS。在 MTS 中实例化 COM 对象时还需要一些特殊的 MTS API 调用。这些问题在 Windows 2000 中得到了解决。

Windows 2000 的一个最重要的新特性是在 COM+中集成了 MTS 和 COM。在 Windows 2000 中,COM+基本服务支持 COM+服务(以前的 MTS 服务)需要的环境,所以不再需要特殊的 MTS API 调用。在 COM+服务中,除了分布式事务处理之外,还增加了一些新的服务功能。

Windows 2000 包含 COM+ 1.0。COM+ 1.5 可以在 Windows XP 和 Windows Server 2003 中使用。COM+ 1.5 也新增了一些特性,以提高可伸缩性和可用性,包括应用程序池和循环,以及可配置的隔离级别。

.NET Enterprise Services 可以在 .NET 组件中使用 COM+服务,并为 Windows 2000 及其后续版本提供了支持。在 COM+应用程序中运行 .NET 组件,不需要使用 CCW(参阅第 24 章);而是作为 .NET 组件运行。在操作系统上安装 .NET 运行库程序,会给 COM+服务添加一些扩展。如果安装了两个带有 Enterprise Services 的 .NET 组件,且组件 A 使用组件 B,则不使用 COM 编组功能, .NET 组件可以直接彼此调用。

#### 44.1.2 使用 Enterprise Services 的场合

业务应用程序在逻辑上可以分为显示层、业务层和数据服务层。显示服务层(presentation service layer)负责用户交互,在该服务层上,用户可以与应用程序交互,输入和查看数据。在这个层上使用的技术是 Windows 窗体和 ASP.NET Web 窗体。业务服务层由业务规则和数据规则组成。数据服务层与持久存储器交互,在该层上可以通过组件来使用 ADO.NET。Enterprise Services 可以在业务服务层和数据服务层上使用。

图 44-1 显示了两个应用程序的情况。Enterprise Services 可以直接在使用 Windows 窗体或 WPF 的胖客户机上使用,或在运行 ASP.NET 的 Web 应用程序上使用。

Enterprise Services 也是一种可伸缩的技术。使用组件加载均衡技术,就可以在不同的系统上分布客户的负载。

还可以在客户系统上使用 Enterprise Services,因为该技术包含在 Windows XP 和 Windows Vista 中。

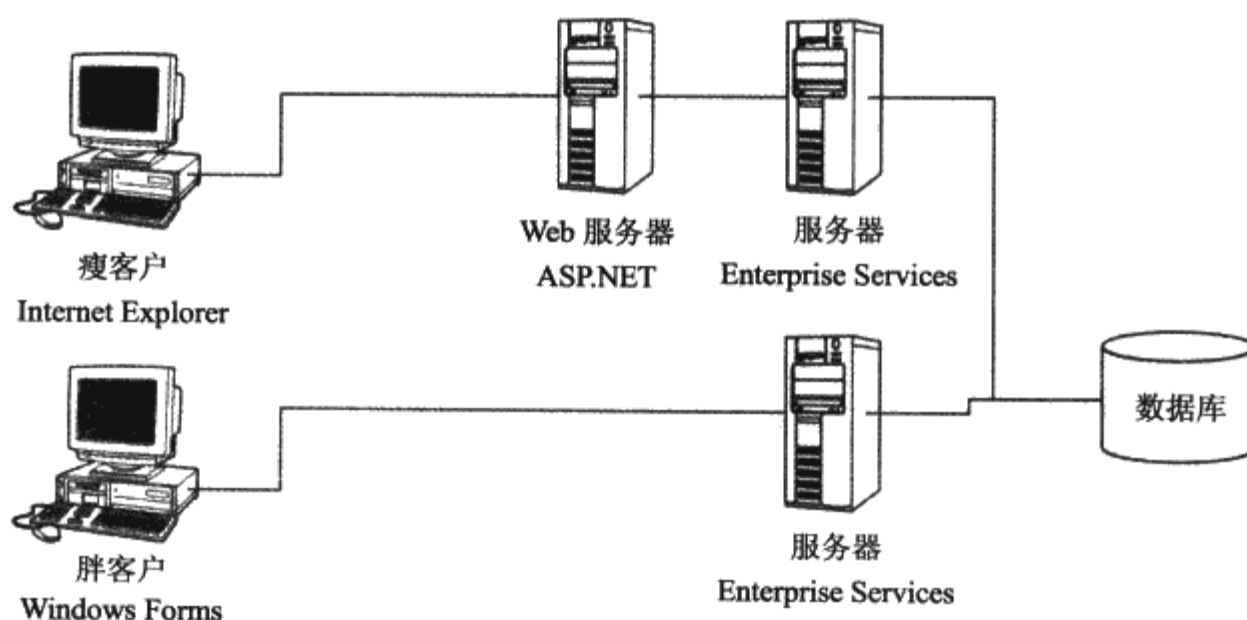


图 44-1

### 44.1.3 环境

Enterprise Services 的基本功能是提供环境(context)。该环境可以截取方法调用,在调用希望的方法之前执行某个服务功能。例如,在调用由组件实现的方法之前,可以创建事务处理或同步范围。

有了环境后,COM 组件和.NET 组件就可以参与同一个事务处理了。这要归功于基类 `ServicedComponent`,这个类本身派生于 `MarshalByRefObject`,集成了.NET 和 COM+环境。

### 44.1.4 自动的事务处理

Enterprise Services 最常用的特性是自动事务处理。使用这个特性,就不需要在代码中启动和执行事务处理了,而是可以把属性应用于一个类。使用 `[Transaction]` 属性和选项 `Required`、`Supported`、`RequiresNew`、`NotSupported`,就可以把一个类标记为需要有相关的事务处理。如果用选项 `Required` 标记属性,在方法开始时就会自动创建一个事务处理,并在事务处理的根组件完成时提交或停止事务处理。

在开发复杂的对象模型时,程序的这种声明方式与手工编写事务处理相比,具有特别的优势。例如,假定有一个 `Person` 对象,以及几个与 `Person` 对象相关的 `Address` 和 `Document` 对象。现在要把 `Person` 对象和所有的相关对象都存储在一个事务处理中。通过编程方式进行事务处理,就意味着把一个事务处理对象传送给所有相关的对象,使它们能参与同一个事务处理。明确使用事务处理,就不需要传送事务处理对象,因为这会通过环境在后台上进行。

### 44.1.5 分布式事务处理

Enterprise Services 不仅提供了自动的事务处理,事务处理还可以分布在多个数据库中。Enterprise Services 事务处理通过 `Distributed Transaction Coordinator(DTC)`来支持,DTC 支持使用 XA 协议的数据库,XA 协议是一种两阶段执行的协议,由 `SQL Server` 和 `Oracle` 支持。单个事务处理可以把数据写到 `SQL Server` 和 `Oracle` 数据库上。

分布式事务处理不仅对数据库有用，而且单个事务处理还可以把数据写到数据库和消息队列上，如果这两个操作中的一个失败，另一个操作就会回滚。消息排队详见第 45 章。

**提示：**

Enterprise Services 支持可升级的事务处理。如果使用 SQL Server 2005 或 2008，且在一个事务处理中只要一个激活的连接，就创建一个本地事务处理。如果在同一个事务处理中激活了另一个事务处理资源，该事务处理就升级为 DTC 事务处理。

本章的后面将讨论如何创建需要事务处理的组件。

#### 44.1.6 对象池

对象池是 Enterprise Services 提供的另一个特性。这些服务使用线程池来回应客户的请求。对象池可以用于初始化时间比较长的对象。使用对象池后，对象就会提前创建，这样客户就不需要等待对象初始化了。

#### 44.1.7 基于角色的安全性

使用基于角色的安全性，可以明确地定义角色，定义在什么角色中可以使用哪些方法或组件。系统管理员给用户或用户组赋予这些角色。在程序中，不需要处理访问控制表，而可以使用只是简单字符串的角色。

#### 44.1.8 排队的组件

排队的组件是消息队列的一个抽象层。客户机不是把消息传送给消息队列，而是通过一个记录器调用方法，该记录器提供的方法与在 Enterprise Services 中配置的.NET 类相同。该记录器再创建消息，通过消息队列把它们传送给服务器应用程序。

如果客户应用程序运行在断开连接的环境(例如不总是连接服务器的膝上计算机)下，或者发送给服务器的请求要在发送给另一个服务器(例如发送给商业伙伴的服务器)之前缓存，就可以使用排队的组件和消息队列。

#### 44.1.9 松散耦合的事件

第 7 章讨论了.NET 的事件模型，第 24 章讨论了如何在 COM 环境中使用事件。通过这两个事件机制，客户和服务器之间就会建立一个牢固的连接。这与松散耦合的事件(LCE)不同。在 LCE 中，COM+功能会插入客户和服务器之间(如图 44-2 所示)。发布者会通过定义一个事件类，用 COM+来注册它提供的事件。发布者不是把事件直接发送给客户，而是把事件发送给用 LCE 服务注册的事件类。LCE 服务会把事件转发给订阅者，订阅者是为事件注册了订阅的客户应用程序。



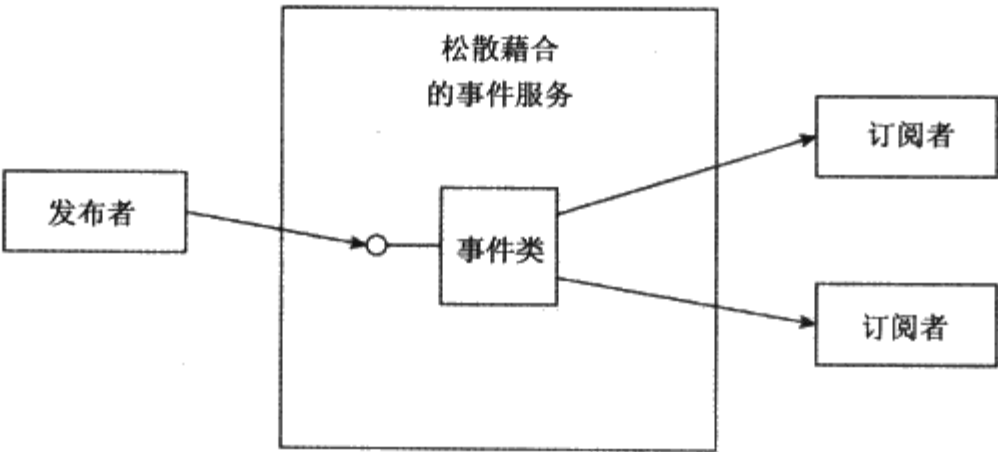


图 44-2

44.2 创建简单的 COM+应用程序

创建可以用 Enterprise Services 配置的.NET 类时，必须引用程序集 System.Enterprise Services，把命名空间 System.EnterpriseServices 添加到 using 声明中。这个应用程序所使用的最重要的类是 ServicedComponent。

第一个示例仅说明创建服务组件的基本要求。首先创建一个 C#库应用程序。所有的 COM+ 应用程序都必须编写为库应用程序，无论它们是运行在自己的进程中，还是运行在客户进程中，都是如此。把该库命名为 SimpleServer。引用程序集 System.EnterpriseServices，给 assemblyinfo.cs 文件和 class1.cs 文件添加声明 using System.EnterpriseServices;。

44.2.1 类 ServicedComponent

每个服务组件类都必须派生于基类 ServicedComponent。ServicedComponent 类本身派生于类 ContextBoundObject，所以实例会绑定到.NET Remoting 环境上。

类 ServicedComponent 包含一些可重写的受保护方法，如表 44-1 所示。

表 44-1

受保护的方法	说 明
Activate() Deactivate()	如果对象配置为使用对象池，就调用 Activate()和 Deactivate()方法。从对象池中取出对象时，就调用 Activate()方法。在对象放回对象池之前，调用 Deactivate()方法
CanBePooled()	这是对象池的另一个方法。如果对象的状态不一致，就可以在 CanBePooled()的重写实现代码中返回 false。这样对象就不会放回对象池，而是被销毁。而对象池会创建一个新对象
Construct()	这个方法在实例化时调用，实例化时将一个构造字符串传送给对象。构造字符串可以由系统管理员修改。本章后面将使用构造字符串定义数据库连接字符串

44.2.2 标记程序集

用 Enterprise Services 配置的库需要一个强名称。对于一些 Enterprise Services 特性来说，还需要在全局程序集缓存中安装程序集。强名称和全局程序集缓存在第 17 章讨论。

44.2.3 程序集的属性

还需要一些 Enterprise Services 属性。属性 ApplicationName 定义了应用程序在 Component Services 浏览器中显示的名称。Description 属性的值在应用程序配置工具中显示为描述。

ApplicationActivate 允许使用选项 ActivationOption.Library 或 ActivationOption.Server, 定义应用程序是应配置为库应用程序还是服务器应用程序。如果配置为库应用程序, 该应用程序就会在客户进程中加载。在这种情况下, 客户可能是 ASP.NET 运行库。如果配置为服务器应用程序, 就启动应用程序进程。进程的名称是 dllhost.exe。使用属性 ApplicationAccessControl 可以关闭安全功能, 这样每个用户就都可以使用组件了。

把 class1.cs 文件重命名为 SimpleComponent.cs, 在命名空间声明的外部添加这些属性:

```
[assembly: ApplicationName("Wrox EnterpriseDemo")]
[assembly: Description("Wrox Sample Application for Professional C#")]
[assembly: ApplicationActivation(ActivationOption.Server)]
[assembly: ApplicationAccessControl(false)]
```

表 44-2 列出了可以用 Enterprise Services 应用程序定义的、最重要的程序集属性。

表 44-2	
属 性	说 明
[ApplicationName]	属性[ApplicationName]定义 COM+应用程序的名称, 在配置组件后, 该名称显示在 Component Services 浏览器中
[ApplicationActivation]	属性[ApplicationActivation]确定应用程序是应运行在客户应用程序的一个库中, 还是应启动一个单独的进程。要配置的选项用枚举 ActivationOption 定义。ActivationOption.Library 指定在客户机的进程中运行应用程序; ActivationOption.Server 启动它自己的进程 dllhost.exe
[ApplicationAccessControl]	属性[ApplicationAccessControl]定义应用程序的安全配置。使用布尔值可以设置启用或禁用访问控制。使用 Authentication 属性可以设置私有级别: 即客户机是应在每个方法调用中验证, 还是仅在连接时验证, 还可以确定发送的数据是否应加密

44.2.4 创建组件

在文件 SimpleComponent.cs 中, 可以创建服务组件类。有了服务组件, 最好定义一个接口作为客户程序和组件之间的契约。这不是一个苛刻的要求, 但一些 Enterprise Services 特性(例如在方法或接口级别上设置基于角色的安全性)需要接口。用方法 Welcome()创建接口 IGreeting。可以在 Enterprise Services 特性中访问的服务组件和接口都需要应用[CpmVisible]特性:

```
using System;
using System.EnterpriseServices;
using System.Runtime.InteropServices;

namespace Wrox.ProCSharp.EnterpriseServices
{
    [ComVisible(true)]
```

```
public interface IGreeting
{
    string Welcome(string name);
}
```

类 SimpleComponent 派生于基类 SercivedComponent，实现接口 IGreeting。类 SercivedComponent 作为所有服务组件类的基类，为激活和构造过程提供了一些方法。把属性[EventTrackingEnabled]应用于这个类，使之能用 Component Sercives 浏览器监视对象。在默认情况下是禁止监视对象的，因为使用这个功能会降低性能。属性[Description]仅指定显示在浏览器上的文本：

```
[EventTrackingEnabled(true)]
[ComVisible(true)]
[Description("Simple Serviced Component Sample")]
public class SimpleComponent : SercivedComponent, IGreeting
{
    public SimpleComponent()
    {
    }
}
```

方法 Welcome()仅返回“Hello, ”和传送给参数的名称。要在 Component Sercives 浏览器中运行组件的同时查看一些结果，Thread.Sleep()模拟了一些处理时间：

```
public string Welcome(string name)
{
    //simulate some processing time
    System.Threading.Thread.Sleep(1000);
    return "Hello, " + name;
}
```

除了应用一些属性，使类派生于 SercivedComponent 之外，不需要对使用 Enterprise Services 特性的类做什么特别的工作。剩下的就是创建和部署客户应用程序了。

在第一个示例组件中设置了[EventTrackingEnabled]属性。有一些更常用的属性会影响服务组件的配置，如表 44-3 所示。

表 44-3

属 性 类	说 明
[EventTrackingEnabled]	设置[EventTrackingEnabled]属性，将允许使用 Component Services 浏览器监视组件。把这个属性设置为 true，会产生额外的开销，所以默认情况下关闭事件跟踪功能
[JustInTimeActivation]	使用这个属性，可以把组件配置为在调用程序实例化类时不激活，而是在调用第一个方法时激活。另外，使用这个属性，组件可以自动失效
[ObjectPooling]	如果与方法调用的时间相比，组件的初始化时间比较长，就可以用[ObjectPooling]属性配置对象池。使用这个属性，可以定义影响池中对象数的最大值和最小值
[Transaction]	[Transaction]属性定义组件的事务处理特性。这里组件定义了是否需要、支持或不支持事务处理

## 44.3 部署

拥有服务组件的程序集必须用 COM+配置。这个配置可以自动进行,或通过手工注册程序集来完成。

### 44.3.1 自动部署

如果启动了使用服务组件的.NET 客户应用程序,就会自动配置 COM+应用程序。所有派生于 `ServicedComponent` 的类都是这样。应用程序和诸如[`EventTrackingEnabled`]的类属性定义了配置的特性。

自动部署有一个重要的缺点。在自动部署时,客户应用程序需要管理权限。如果调用访问组件的客户应用程序是 ASP.NET 应用程序,ASP.NET 运行库一般没有管理权限。所以自动部署仅用于开发阶段。而在开发阶段,这是一个极佳的优势。在每次创建程序后,都不需要进行手工部署。

### 44.3.2 手工部署

手工部署程序集可以通过.NET 服务安装工具 `regsvcs.exe`(一种命令行实用工具)进行。输入下面的命令:

```
regsvcs SimpleServer.dll
```

就会把程序集 `SimpleServer` 注册为一个 COM+应用程序,并根据属性配置包含的组件,创建一个可以由访问.NET 组件的 COM 客户使用的类型库。

在配置好程序集后,就可以从 Windows XP 或 Windows Server 2003 的 Windows 菜单中选择 Administrative Tools | Component Services,启动 Component Services 浏览器,在 Windows Vista 上,必须启动 MMC,添加 Component Services 插件,才能看到 Component Services 浏览器。在应用程序的左边树图中选择 Component Services | Computers | My Computer | COM+ Application,验证应用程序是否已配置。

### 44.3.3 创建安装软件包

使用 Component Services 浏览器,可以为服务器或客户机系统创建安装软件包。服务器的安装软件包包含用于把应用程序安装到另一个服务器上的程序集和配置设置。如果在运行在另一个系统上的应用程序中调用服务组件,就必须在客户系统上安装一个代理对象。客户机的安装软件包包含代理对象的程序集和配置。

要创建安装软件包,可以启动 Component Services 浏览器,选择 COM+应用程序,选择菜单 Action | Export,单击第一个对话框中的 Next 按钮,打开如图 44-3 所示的对话框。在这个对

对话框中可以输出 Server application 或 Application proxy。在 Server application 选项中，还可以进行配置，输出带有角色的用户标识。这个选项只能在目标系统与创建安装软件包的系统在同一域中时使用，此时，配置的用户标识放在安装软件包中。使用 Application proxy 选项，会创建客户系统的安装软件包。

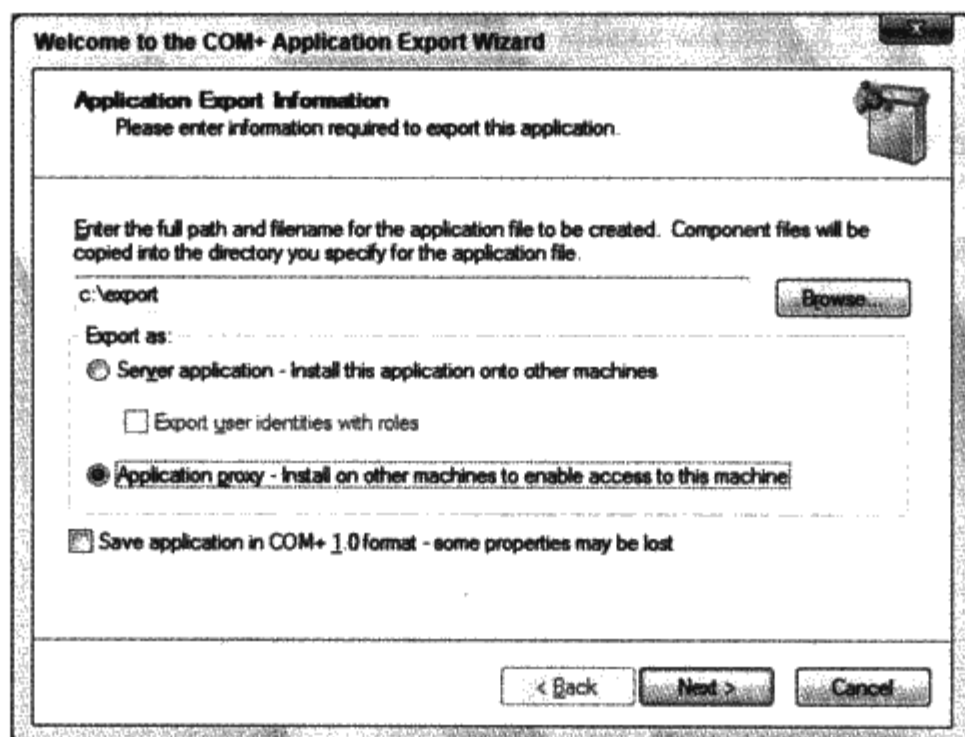


图 44-3

**注意：**

如果应用程序配置为库应用程序，就不能使用创建 Application proxy 的选项。

要安装代理对象，只需启动安装软件包中的 setup.exe。注意应用程序代理不能安装在应用程序所在的系统中。在安装完应用程序代理后，在 Component Services 浏览器中就有一项表示应用程序代理。在应用程序代理中，能配置的唯一选项是 Activation 选项卡中的服务器名，如下一节所述。

## 44.4 Component Services 浏览器

在成功配置后，就可以在 Component Services 浏览器的树图中看到 Wrox EnterPriseDemo 应用程序名。这个名称由属性[ApplicationName]设置。选择 Action | Properties，打开如图 44-4 所示的对话框。名称和描述都已使用属性配置了。在选择 Activations 标签时，可以看到该应用程序被配置为服务器应用程序，因为它使用[ApplicationActivation]属性定义，选择 Security 标签，其中没有选择 Enforce access checks for this application 选项，因为属性[ApplicationAccessControl]设置为 false。



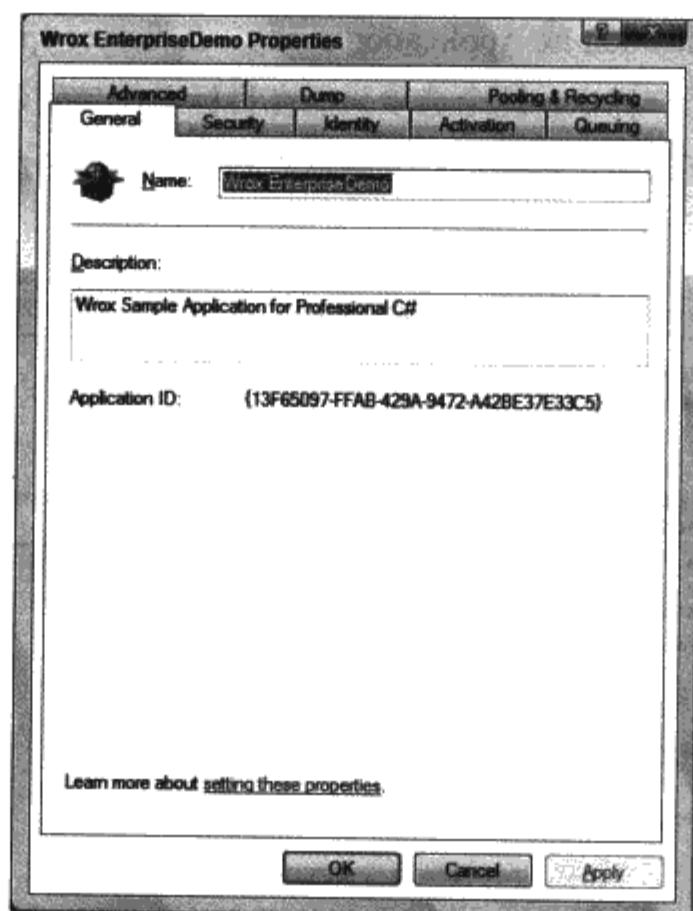


图 44-4

这个应用程序还可以设置其他一些选项：

- **Security:** 在安全配置中，可以启用或禁用访问检查。如果启用安全功能，就可以把访问检查设置为应用程序级别、组件级别、接口级别或方法级别。还可以把数据包私密性作为调用的识别级别，加密在网络上发送的消息。当然，这会增加开销。
- **Identity:** 在服务器应用程序中，使用 Identity 选项卡可以为驻留应用程序的进程配置要使用的用户账户。在默认情况下这是一个交互式的用户。这个设置在调试应用程序时非常有用，但如果应用程序在服务器上运行，该设置就不能在产品系统上使用，因为没有人能登录。在把应用程序安装到产品系统上之前，应为应用程序使用特定的用户，来测试应用程序。
- **Activation:** Activation 选项卡允许把应用程序配置为库或服务器应用程序。COM+ 1.5 的两个新选项是把应用程序作为 Windows 服务运行和使用 SOAP 访问应用程序。第 23 章将讨论 Windows 服务。选择 SOAP 选项，将在 Internet Information Server 中使用 .NET Remoting 配置来访问组件。本章的后面不使用 .NET Remoting，而使用 WCF 访问组件。WCF 已在第 42 章讨论过。
- 对于应用程序代理，选项 Remote Server Name 是可以配置的唯一选项。使用这个选项可以设置服务器名。在默认情况下，DCOM 协议用作网络协议。但是，如果服务器配置选择了 SOAP，就通过 .NET Remoting 进行通信。
- **Queuing:** 使用 Message Queuing 的服务组件需要 Queuing 配置。
- **Advanced:** 在 Advanced 选项卡中，可以指定应用程序是否应在客户处于未激活状态时的一段时间后停止。还可以指定是否锁定某个配置，这样就不会有人在无意中修改它了。

- **Dump:** 如果应用程序崩溃, 就可以指定应把垃圾存储在目录的什么地方。Dump 用于利用 C++ 开发的组件。
- **Pooling & Recycling:** 这是 COM+ 1.5 的一个新选项。利用这个选项可以配置应用程序是否根据生存期、需要的内存、调用的次数等重新启动(循环)。

使用 Component Services 浏览器还可以查看和配置组件本身。在打开应用程序的子元素时, 可以查看组件 Wrox.ProCSharp.EnterpriseServices.SimpleComponent。选择 Action | Properties 会打开如图 44-5 所示的对话框。

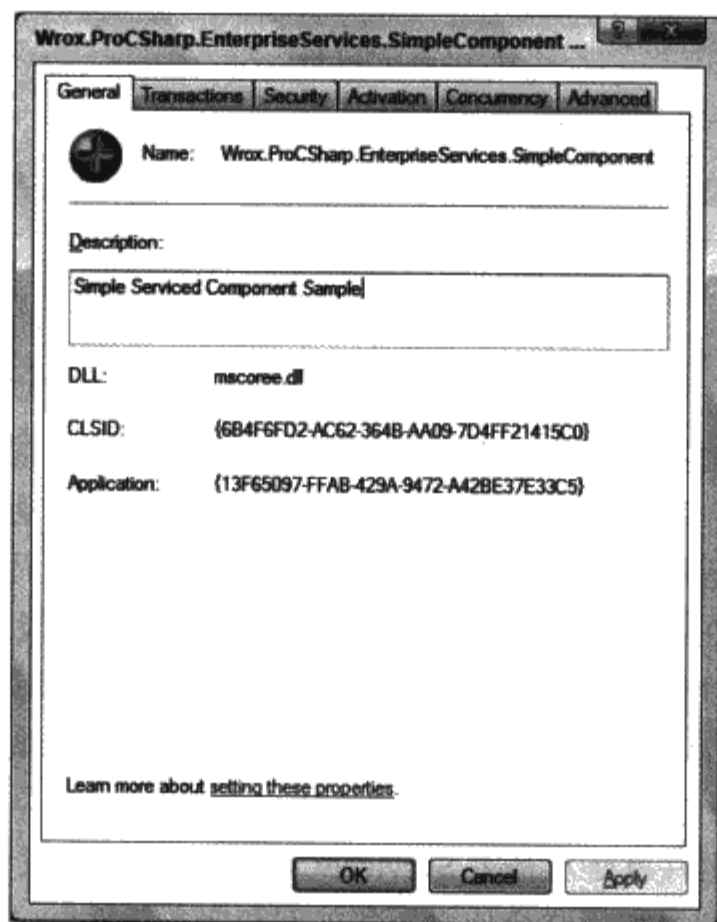


图 44-5

使用这个对话框, 可以配置如下选项:

- **Transactions:** 使用 Transactions 选项卡可以指定组件是否需要事务处理。下一个示例将使用这个特性。
- **Security:** 如果应用程序启用了安全功能, 利用这个配置就可以定义允许使用组件的角色。
- **Activation:** Activation 配置允许设置对象池, 指定构造字符串。
- **Concurrency:** 如果组件不是线程安全的, concurrency 就可以设置为 Required 或 Requires New。这样 COM+ 运行库就仅允许一个线程访问组件。

## 44.5 客户应用程序

在建立了服务组件库后, 就可以创建客户应用程序, 这可以是简单的 C# 控制台应用程序。在为客户创建了项目后, 必须引用服务组件中的程序集 SimpleServer 和程序集 System.EnterpriseServices。接着编写代码, 实例化一个新的 SimpleComponent 实例, 调用方法 Welcome()。

在下面的代码中调用方法 `Welcome()` 共 10 次。在垃圾收集器采取措施前，`using` 语句帮助释放分配给实例的资源。有了 `using` 语句，服务组件的 `Dispose()` 方法就会在 `using` 语句的最后调用。

```
using System;

namespace Wrox.ProCSharp.EnterpriseServices
{
    class Program
    {
        static void Main()
        {
            using (SimpleComponent obj = new SimpleComponent())
            {
                for (int i = 0; i < 10; i++)
                {
                    Console.WriteLine(obj>Welcome("Kathie"));
                }
            }
        }
    }
}
```

如果在配置服务器之前启动客户应用程序，服务器就会自动配置。服务器的自动配置是通过属性指定的值来进行的。下面进行测试。注销服务组件，再次启动客户程序。如果服务组件在客户应用程序的启动过程中配置，启动时间就会较长。这个特性只能用在开发阶段。自动部署还需要管理权限。如果在 VS 中启动应用程序，就应在有管理权限的情况下启动 VS。

在运行应用程序时，可以用 `Component Services` 浏览器监视服务组件。在树图中选择 `Components`，再选择 `View | Detail`，就可以查看设置属性 `[EventTrackingEnabled]` 后实例化对象的个数。

可以看出，创建服务组件就是从基类 `ServiceComponent` 中派生一个类，再设置一些属性，来配置应用程序。下面要学习如何联合使用事务处理和服务组件。

## 44.6 事务处理

自动事务处理是 `Enterprise Services` 中最常用的特性。使用 `Enterprise Services` 可以把组件标记为需要事务处理，接着在 `COM+` 运行库中创建事务处理。组件中所有支持事务处理的对象，例如 `ADO.NET` 连接，都在事务处理中运行。

提示：

事务处理的概念详见第 22 章。

### 44.6.1 事务处理的属性

服务组件可以用 `[Transaction]` 属性标记，以定义组件是否需要事务处理，以及如何进行事务处理。

图 44-6 显示了不同的事务处理配置的多个组件。客户调用组件 A。因为组件 A 是用 `TransactionOption.Required` 配置的，而且以前不存在事务处理，所以创建一个新的事务处理 1。

组件 A 调用组件 B，组件 B 调用组件 C。因为组件 B 是用 TransactionOption.Supported 配置的，而组件 C 的配置是 TransactionOption.Required，所以组件 A、B 和 C 都使用同一个事务处理环境。如果组件 B 是用 NotSupported 配置的，组件 C 就会得到一个新的事务处理。组件 D 是用 TransactionOption.RequiresNew 配置的，所以在组件 A 调用组件 D 时，会创建一个新的事务处理。

表 44-4 给出了 TransactionOption 值及其说明的列表。

表 44-4

TransactionOption 值	说 明
Required	把[Transaction]属性设置为 TransactionOption.Required, 表示组件在事务处理中运行。如果已经创建了一个事务处理，组件就运行在这个事务处理中。如果事务处理不存在，就创建一个
RequiresNew	TransactionOption.RequiresNew 总是会创建一个新的事务处理。组件从来不参与调用者所在的事务处理
Supported	使用 TransactionOption.Supported，组件就不需要事务处理了。但是，如果这些组件需要事务处理，事务处理就会位于调用者和被调用的组件中。
NotSupported	选项 TransactionOption.NotSupported 表示无论调用者是否有事务处理，组件从来都不运行在事务处理中
Disabled	TransactionOption.Disabled 表示忽略当前环境的事务处理

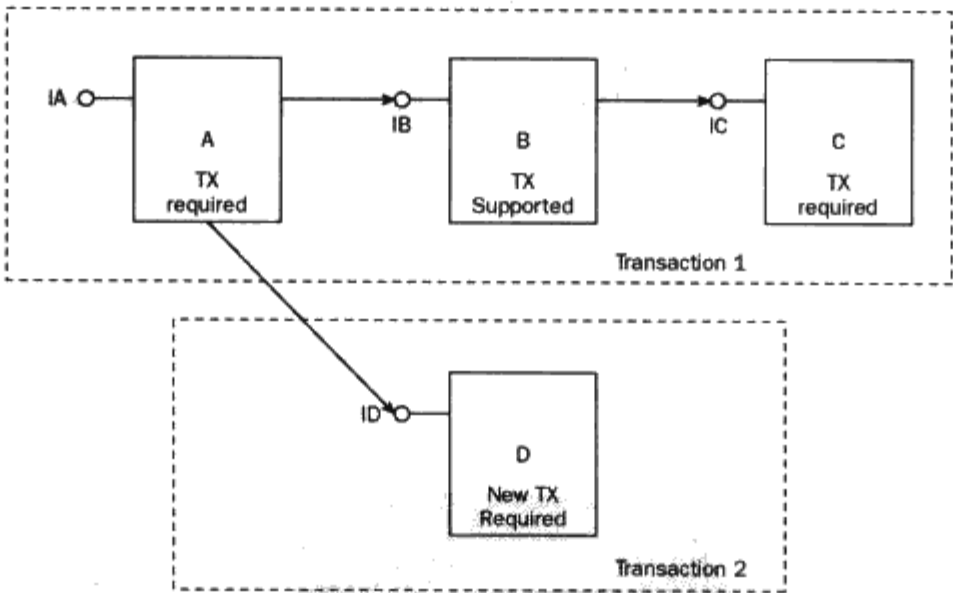


图 44-6

44.6.2 事务处理的结果

设置环境的 consistent 和 done 位会影响事务处理。如果 consistent 位设置为 true，就表示组件对事务处理的结果很满意。如果所有参与事务处理的组件都成功了，事务处理就完成了。如果 consistent 位设置为 false，则组件对事务处理的结果不满意，在启动事务处理的根对象完成时，就会中止事务处理。如果设置了 done 位，就可以在方法调用完成后释放对象，用下一个

方法调用创建新实例。  
使用 ContextUtil 类的 4 个方法可以设置 consistent 和 done 位，结果如表 44-5 所示。

表 44-5

ContextUtil 方法	consistent 位	done 位
SetComplete	true	true
SetAbort	false	true
EnableCommit	true	false
DisableCommit	false	false

在.NET 中，还可以对方法应用属性[AutoComplete]，来设置 consistent 和 done 位，而不是调用 ContextUtil 方法。使用这个属性，如果方法成功，就自动调用 ContextUtil.SetComplete() 方法。如果方法失败，并抛出一个异常，则通过属性[AutoComplete]调用 ContextUtil. SetAbort() 方法。

44.7 示例应用程序

在这个示例应用程序中，要模拟一个简化的场景：把一些新订单写到 Northwind 示例数据库中。如图 44-7 所示，多个组件和 COM+应用程序一起使用。在客户应用程序中调用类 OrderControl，创建新的订单。OrderControl 使用 OrderData 组件，该组件负责在 Northwind 数据库的 Order 表中创建一个新条目。OrderData 组件使用 OrderLineData 组件把 Order Detail 条目写到数据库中。OrderData 和 OrderLineData 组件都必须参与同一个事务处理。

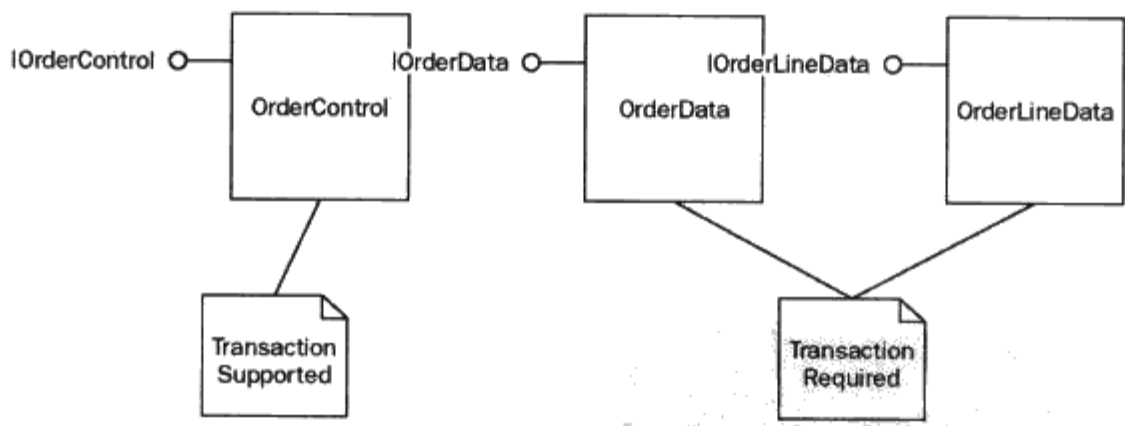


图 44-7

首先要创建一个 C#组件库，命名为 NorthwindComppnent。用密钥文件标记程序集，定义 Enterprise Services 应用程序属性，如下面的代码所示：

```
[assembly: ApplicationName("Wrox NorthwindDemo")]
[assembly: ApplicationActivation(ActivationOption.Library)]
[assembly: ApplicationAccessControl(false)]
```



### 44.7.1 实体类

接着添加一些实体类 Order 和 OrderLine, 表示 Northwind 数据库中 Order 表和 Order Detail 表中的列。实体类只是数据存储器, 表示对应用程序域很重要的数据, 在本例中就是下订单。类 Order 有一个静态方法 Create(), 它创建并返回类 Order 的一个新实例, 用传送给该方法的参数初始化这个实例。类 Order 还有一些只读属性, 以访问字段 orderId、customerId、orderDate、shipAddress、shipCity 和 shipCountry。orderId 在类 Order 的创建阶段是未知的, 但因为 Northwind 数据库中的 Order 表有一个自动递增的属性, 所以 orderId 仅在订单写入数据库中后才已知。方法 SetOrderId() 用于在订单写入数据库后设置相应的 id。因为这个方法由同一程序集中的类调用, 所以这个方法的访问级别设置为 internal。方法 AddOrderLine() 把订单细节添加到订单中。

```
using System;
using System.Collections.Generic;

namespace Wrox.ProCSharp.EnterpriseServices
{
    [Serializable]
    public class Order
    {
        public static Order Create(string customerId, DateTime orderDate,
                                   string shipAddress, string shipCity, string shipCountry)
        {
            return new Order()
            {
                CustomerId = customerId,
                OrderDate = orderDate,
                ShipAddress = shipAddress,
                ShipCity = shipCity,
                ShipCountry = shipCountry
            }
        }

        public Order()
        {
        }

        internal void SetOrderId(int orderId)
        {
            this.orderId = orderId;
        }

        public void AddOrderLine(OrderLine orderLine)
        {
            orderLines.Add(orderLine);
        }

        private List < OrderLine > orderLines = new List < OrderLine > ();
        public int OrderId { get; private set; }
        public string CustomerId { get; private set; }
        public DateTime OrderDate { get; private set; }
        public string ShipAddress { get; private set; }
        public string ShipCity { get; private set; }
        public string ShipCountry { get; private set; }

        public OrderLines[] OrderLines
    }
}
```

```

    {
        get
        {
            OrderLines[] ol = new OrderLine[orderliness.Count];
            orderlines.CopyTo(ol);
            return ol;
        }
    }
}

```

第二个实体类是 `OrderLine`，它有一个类似于 `Order` 类的静态方法 `Create()`，除此之外，这个类仅包含一些属性，用于访问字段 `productedId`、`unitPrice` 和 `quantity`。

```

using System;

namespace Wrox.ProCSharp.EnterpriseServices
{
    [Serializable]
    public class OrderLine
    {
        public static OrderLine Create(int productId, float unitPrice, int quantity)
        {
            return new OrderLine()
            {
                ProductId = productId,
                UnitPrice = unitPrice,
                Quantity = quantity
            };
        }
        public OrderLine()
        {
        }
        public int ProductId { get; set; }
        public float UnitPrice { get; set; }
        public int Quantity { get; set; }
    }
}

```

#### 44.7.2 OrderControl 组件

类 `OrderControl` 表示一个简单的业务服务组件。在这个示例中，接口 `IOrderControl` 中只定义了一个方法 `NewOrder()`。方法 `NewOrder()` 的实现代码只实例化数据服务组件 `OrderData` 的一个新实例，并调用方法 `Insert()` 把 `Order` 对象写入数据库。在比较复杂的情况下，这个方法可以扩展为把一个日志项写入数据库，或者调用排队的组件，把 `Order` 对象发送给消息队列。

```

using System;
using System.EnterpriseServices;
using System.Runtime.InteropServices;

namespace Wrox.ProCSharp.EnterpriseServices;
{
    [ComVisible(true)]
    public interface IOrderControl
    {
        void NewOrder(Order order);
    }
}

```

```

[Transaction(TransactionOption.Supported)]
[EventTrackingEnabled(true)]
[ComVisible(true)]
public class OrderControl : ServicedComponent, IOrderControl
{
    [AutoComplete()]
    public void NewOrder(Order order)
    {
        using (OrderData data = new OrderData())
        {
            data.Insert(order);
        }
    }
}

```

### 44.7.3 OrderData 组件

OrderData 类负责把 Order 对象的值写入数据库。接口 IOrderUpdate 定义了方法 Insert()。可以扩展这个接口，使之支持 Update()方法，该方法会更新数据库中已有的项。

```

using System;
using System.EnterpriseServices;
using System.Runtime.InteropServices;
using System.Data.SqlClient;

namespace Wrox.ProCSharp.EnterpriseServices
{
    [comvisible(true)]
    public interface IOrderUpdate
    {
        void Insert(Order order);
    }
}

```

类 OrderData 的属性[Transaction]值是 TransactionOption.Required。这表示组件在任何情况下都运行在事务处理中。如果事务处理由调用者创建，OrderData 就使用这个事务处理，否则就创建一个事务处理。这里会创建一个新的事务处理，因为调用组件 OrderControl 没有事务处理。

在服务组件中，只能使用默认的构造函数。但是，可以使用 Component Services 浏览器配置一个发送给组件的构造字符串(如图 44-8 所示)。选择组件配置的 Activation 选项卡，可以修改构造字符串。在设置属性[ConstructiounEnable]时，选项 Enable object constructioun 会打开，因为它与类 OrderData 一起使用。[ConstructiounEnable]的属性 Default 定义了默认的连接字符串，在注册程序集后，该字符串会显示在 Activation 设置中。设置这个属性还需要重载基类 ServicedComponent 中的方法 Construct()。这个方法由 COM+运行库在对象实例化时调用，同时把构造字符串作为参数传送。构造字符串设置为变量 connectionString，该变量在以后用于连接数据库。

```

[Transaction(TransactionOption.Required)]
[EventTrackingEnabled(true)]
[ConstructionEnabled(true, Default="server=(local);"+
    "database=northwind;trusted_connection=true")]
[comVisible(true)]

```



```

public class OrderData : ServicedComponent, IOrderUpdate
{
    private string connectionString = null;

    protected override void Construct(string s)
    {
        connectionString = s;
    }
}

```

方法 `Insert()` 是组件的核心。这里使用 ADO.NET 把 `Order` 对象写入数据库。ADO.NET 详见第 26 章。对于这个示例, 我们创建一个 `SqlConnection` 对象, 在该对象中, 使用方法 `Construct()` 设置连接字符串, 以初始化该对象。

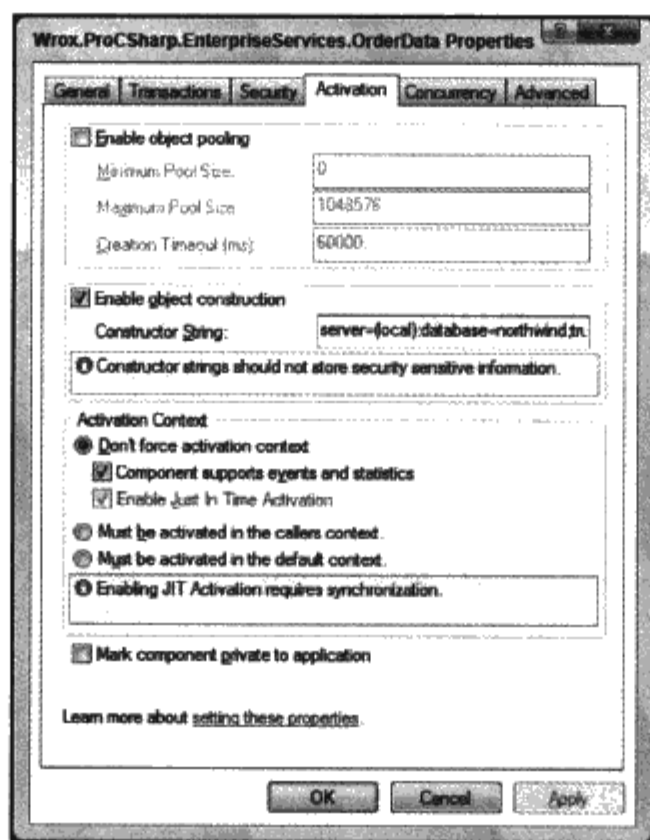


图 44-8

属性 `[AutoComplete()]` 应用于下面的方法, 进行前面论述的自动事务处理。

```

[AutoComplete()]
public void Insert(Order order)
{
    SqlConnection connection = new SqlConnection(connectionString);
}

```

方法 `connection.CreateCommand()` 创建一个 `SqlCommand` 对象, 在该对象中, 把 `CommandText` 属性设置为一个 SQL INSERT 语句, 用于把新记录添加到 `Orders` 表中。方法 `ExecuteNonQuery()` 执行这个 SQL 语句:

```

try
{
    SqlCommand command = connection.CreateCommand();
    command.CommandText = "INSERT INTO Orders (CustomerId, OrderDate, " +
        "ShipAddress, ShipCity, ShipCountry) " +
        "VALUES(@CustomerId, @OrderDate, @ShipAddress, @ShipCity, " + "@ShipCountry) ";
    command.Parameters.Add("@CustomerId", order.CustomerId);
    command.Parameters.Add("@OrderDate", order.OrderDate);
}

```

```

command.Parameters.Add("@ShipAddress", order.ShipAddress);
command.Parameters.Add("@ShipCity", order.ShipCity);
command.Parameters.Add("@ShipCountry", order.ShipCountry);

connection.Open();

command.ExecuteNonQuery();

```

因为 OrderId 定义为数据库中自动递增的值, 把 Order Details 写入数据库时需要这个 id, 所以 OrderId 使用 @@IDENTITY 读取。接着调用方法 SetOrderId() 把它设置为 Order 对象:

```

command.CommandText = "SELECT @@IDENTITY AS 'Identity' ";
object identity = command.ExecuteScalar();
order.SetOrderId(Convert.ToInt32(identity));

```

在订单写入数据库后, 订单上的所有订单行都使用 OrderLineData 组件写入数据库:

```

using (OrderLineData updateOrderLine = new OrderLineData())
{
    foreach (OrderLine orderLine in order.OrderLines)
    {
        updateOrderLine.Insert(order.OrderId, orderLine);
    }
}

```

最后, 无论 try 块中的代码成功执行, 还是发生了异常, 都会断开连接。

```

finally
{
    connection.Close();
}
}
}

```

#### 44.7.4 OrderLineData 组件

OrderLineData 组件的实现类似于 OrderData 组件的实现。使用属性 [ConstructionEnables] 定义数据库连接字符串:

```

using System;
using System.EnterpriseServices;
using System.Runtime.InteropServices;
using System.Data;
using System.Data.SqlClient;

namespace Wrox.ProCSharp.EnterpriseServices
{
    [ComVisible(true)]
    public interface IOrderLineUpdate
    {
        void Insert(int orderId, OrderLine orderDetail);
    }

    [Transaction(TransactionOption.Required)]
    [EventTrackingEnabled(true)]
    [ConstructionEnables(true, Default="server=(local); database=northwind; "+

```



```

        "trusted_connection=true"))]
[ComVisible(true)]
public class OrderLineData : ServicedComponent, IOrderLineUpdate
{
    private string connectionString = null;

    protected override void Construct(string s)
    {
        connectionString = s;
    }
}

```

在 OrderLineData 类的 Insert()方法中,不使用[AutoComplete]属性作为定义事务处理结果的另一种方式,而是说明如何使用 ContextUtil 类设置 consistent 和 done 位。在 Insert()方法的最后,根据在数据库中插入数据是否成功,来调用方法 SetComplete()。万一出现了错误,抛出了异常,方法 SetAbort()就把 consistent 位设置为 false,这样事务处理就和所有参与该事务处理的组件一起撤销。

```

public void Insert(int orderId, OrderLine orderDetail)
{
    SqlConnection connection = new SqlConnection(connectionString);
    try
    {
        SqlCommand command = connection.CreateCommand();
        command.CommandText = "INSERT INTO [Order Details] (OrderId, " +
            "ProductId, UnitPrice, Quantity) " +
            "VALUES(@OrderId, @ ProductId, @UnitPrice, @ Quantity) ";
        command.Parameters.AddWithValue("@OrderId", orderId);
        command.Parameters.AddWithValue("@ProductId", orderDetail.ProductId);
        command.Parameters.AddWithValue("@UnitPrice", orderDetail.UnitPrice);
        command.Parameters.AddWithValue("@Quantity", orderDetail.Quantity);

        connection.Open();

        command.ExecuteNonQuery();
    }
    catch (Exception)
    {
        ContextUtil.SetAbort();
        throw;
    }
    finally
    {
        connection.Close();
    }
    ContextUtil.SetComplete();
}
}

```

#### 44.7.5 客户应用程序

建立了组件后,就可以创建客户应用程序了。为了进行测试,可以创建一个控制台应用程序。在引用程序集 NorthwindComponent 和程序集 System.EnterpriseServices 后,就可以使用静态方法 Order.Create()创建一个新的 Order 对象。使用 Order.AddOrderLine()给 Order 表添加一个新订单行。OrderLine.Create()利用产品 ID、价格和数量来创建订单行。在真实的应用程序中,

添加 Product 类比使用产品 ID 会更好，但本例仅用于演示一般的事务处理。

最后，创建服务组件类 OrderControl，以调用方法 NewOrder()：

```
Order order = Order.Create("PICCO", DateTime.Today, "Georg Papps",
    "Salzburg", "Austria");
order.AddOrderLine(OrderLine.Create(16, 17.45F, 2));
order.AddOrderLine(OrderLine.Create(67, 14, 1));

using (OrderControl orderControl = new OrderControl())
{
    orderControl.NewOrder(order);
}
```

可以试着把不存在的产品写到 OrderLine(使用不在 Products 表中的产品 id)中。在这种情况下，事务处理会中止，不会把数据写到数据库中。

在事务处理处于激活状态时，通过在 Component Services 浏览器的树图中选择 Distributed Transaction Coordinator，可以看到事务处理，如图 44-9 所示。

#### 提示：

可以给 OrderData 类的 Insert()方法添加睡眠时间，以查看活动的事务处理，否则，事务处理就完成得太快，显示不出来。

#### 注意：

如果服务组件在事务处理中运行时对它进行调试，应注意对于该服务组件，默认的事务处理超时时间是 60 秒。在 Component Services 浏览器中单击 My Computer，选择 Action | Properties，打开 Options 选项卡，就可以为整个系统修改这个默认值。也可以为每个组件选择 Transaction 选项，修改每个组件的默认值。

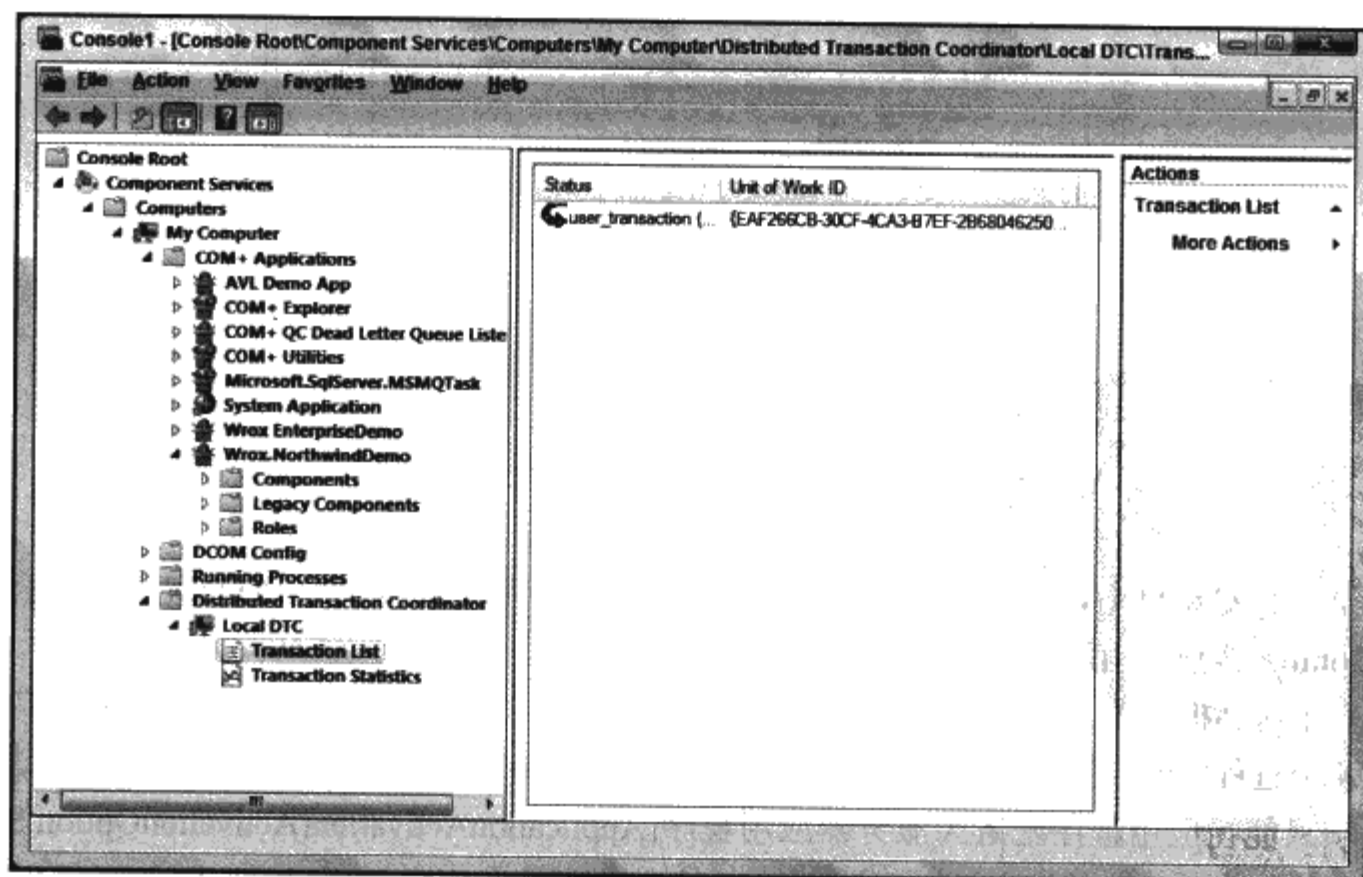


图 44-9

## 44.8 集成 WCF 和 Enterprise Services

Windows Communication Foundation (WCF)是.NET Framework 3.0 中的一个新通信技术，WCF 详见第 40 章。 .NET Enterprise Services 提供了一个与 WCF 集成的模型。

### 44.8.1 WCF 服务 Façade

给 Enterprise Services 应用程序添加 WCF façade，可以使用 WCF 客户程序访问服务组件。除了使用 DCOM 协议之外，还可以通过 WCF 使用不同的协议，如 HTTP、SOAP、TCP 和二进制格式。

选择 Tools | WCF SvcConfigEditor，可以在 Visual Studio 2008 中创建 WCF façade。使用这个工具要先选择 File | Integrate | COM+ Application，再选择 COM+应用程序 Wrox.NorthwindDemo、组件 Wrox.ProCSharp.EnterpriseServices.OrderControl 和接口 IOrderControl，如图 44-10 所示。

提示：

除了使用 Visual Studio 创建 WCF façade 之外，还可以使用命令行实用工具 comsvc-config.exe。这个实用工具在目录 < Windows > \Microsoft.NET\ Framework\v3.0\Windows Communication Foundation 下。

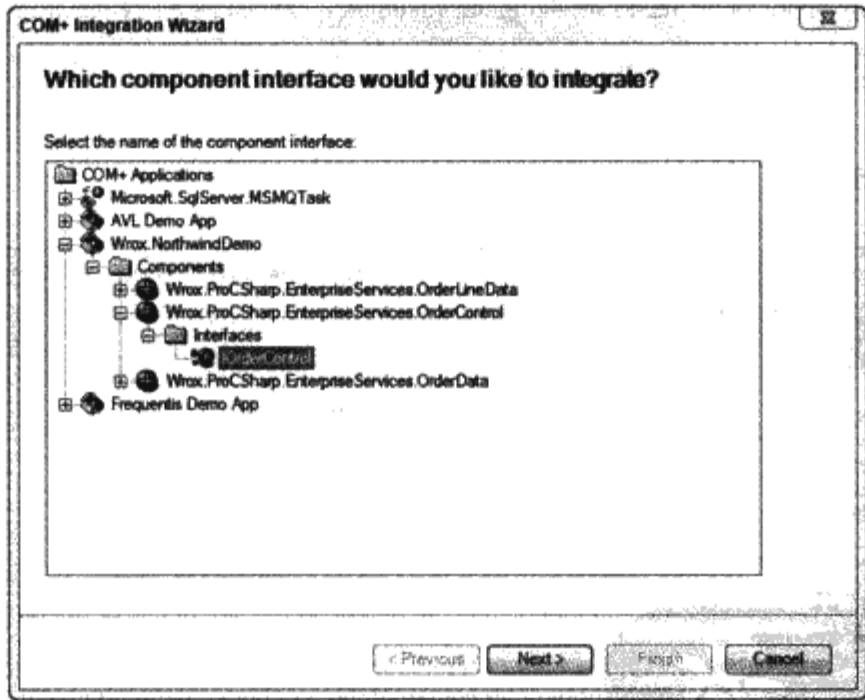


图 44-10

在下一个对话框中，可以选择 IOrderControl 接口中可用于 WCF 客户程序的所有方法。IOrderControl 接口只有一个方法 NewOrder()，如下所示。

下一个对话框如图 44-11 所示，它允许配置主机选项。在主机选项中，可以指定 WCF 服务运行在哪个进程中。选择 COM+ hosted 选项时，WCF façade 运行在 COM+的 dllhost.exe 进程中。这个选项只能在应用程序配置为服务器应用程序[ApplicationActivation(ActivationOption.Server)]时使用。Web hosted 选项指定 WCF 信道在 IIS 的一个进程或 WAS (Windows Activation Services) 工作进程中监听。WAS 是 Windows Vista 和 Windows Server 2008 中新增的。选择 Web hosted in -

process 表示 Enterprise Services 组件库运行在 IIS 或 WAS 工作进程中。这个配置只能在应用程序配置为库应用程序[ApplicationActivation (ActivationOption.Library)]时使用。

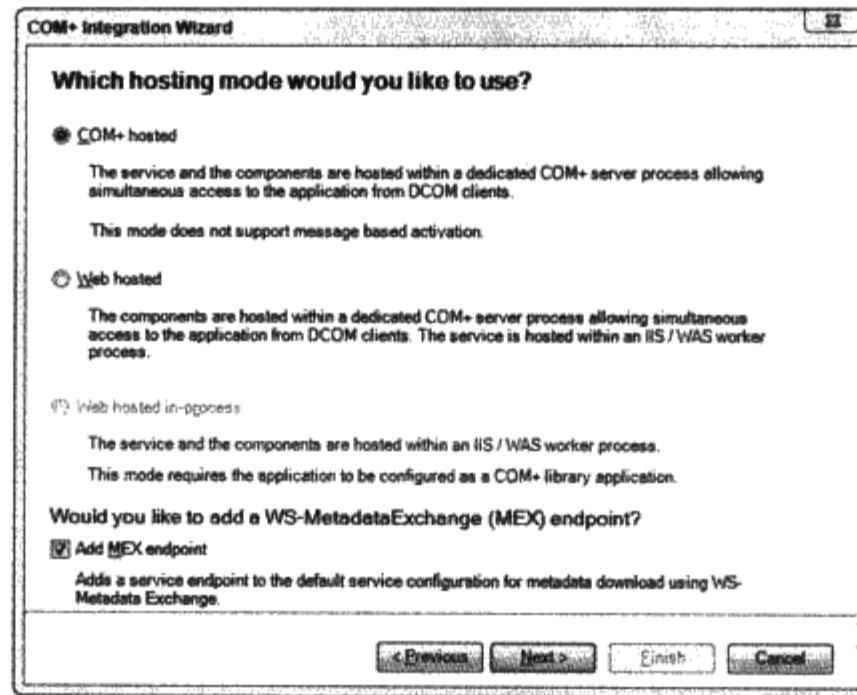


图 44-11

选择 Add MEX endpoint 选项, 会把一个 MEX (Metadata Exchange) 端点添加到 WCF 配置文件中, 这样客户程序员就可以使用 WS - Metadata Exchange 访问服务的元数据了。

**提示:**

MEX 参见第 42 章。

在如图 44-12 所示的下一个对话框中, 可以指定通信模式, 来访问 WCF façade。根据需要, 如果客户程序通过防火墙访问服务, 或者需要独立于平台的通信, HTTP 就是最佳选择。TCP 为 .NET 客户提供了机器之间的更快通信, 如果客户应用程序运行在与服务相同的系统上, Named Pipes 就是最快的选项。

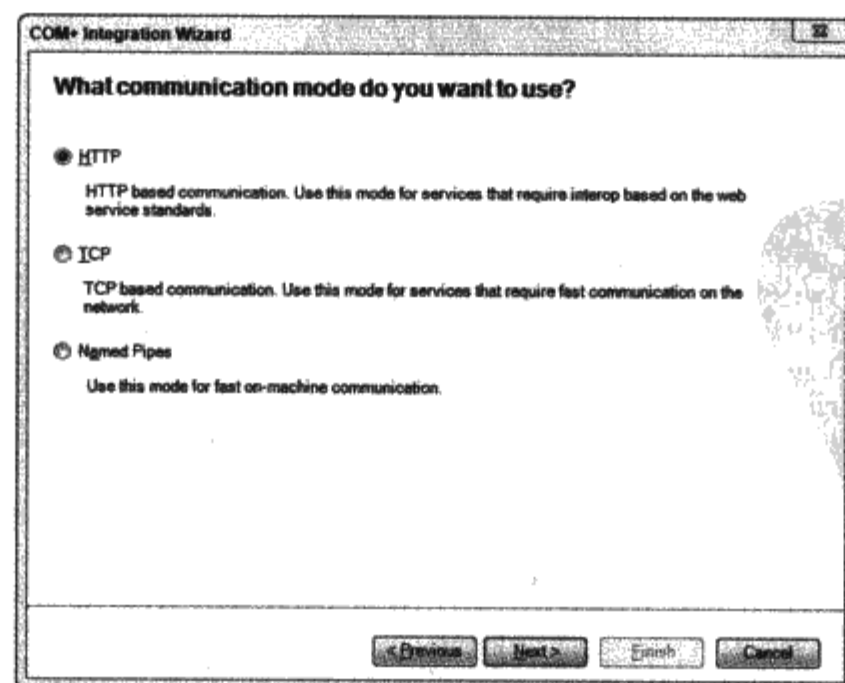


图 44-12

下一个对话框需要服务的基地址，这取决于所选择的通信协议，如图 44-13 所示。



图 44-13

最后一个对话框显示了端点配置的位置。配置的基目录是< Program Files > \ComPlus Applications，其后是应用程序的唯一 ID。在这个目录中，包含了 application.config 文件。这个配置文件列出了 WCF 的操作和端点。

<service>元素指定带端点配置的 WCF 服务。用 comTransactionalBinding 配置把绑定设置为 wsHttpBinding。这样事务就可以从调用程序流向服务组件。利用其他网络和客户要求，可以指定另一个绑定，详见第 42 章。

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.serviceModel>
    <behaviors>
      <serviceBehaviors>
        <behavior name="ComServiceMexBehavior">
          <serviceMetadata httpGetEnabled="false" />
          <serviceDebug />
        </behavior>
      </serviceBehaviors>
    </behaviors>
    <bindings>
      <netNamedPipeBinding>
        <binding name="comNonTransactionalBinding" />
        <binding name="comTransactionalBinding" transactionFlow="true" />
      </netNamedPipeBinding>
      <wsHttpBinding>
        <binding name="comNonTransactionalBinding" />
        <binding name="comTransactionalBinding" transactionFlow="true" />
      </wsHttpBinding>
    </bindings>
    <comContracts>
      <comContract contract="{E1B02E09-EE48-3B6B-946F-E6A8BAEC6340}"
        name="IOrderControl"
        namespace="http://tempuri.org/E1B02E09-EE48-3B6B-946F-E6A8BAEC6340"
        requiresSession="true">
        <exposedMethods>
```



```

        <add exposedMethod="NewOrder" />
    </exposedMethods>
</comContract>
</comContracts>
<services>
  <service behaviorConfiguration="ComServiceMexBehavior"
    name="{6AD42D20-9DB9-4BFB-927E-E5FB5099C8B5},
      {D30F79D7-6DE7-33DE-B3FC-C21F6B02A48D}">
    <endpoint address="IOrderControl" binding="netNamedPipeBinding"
      bindingConfiguration="comTransactionalBinding"
      contract="{E1B02E09-EE48-3B6B-946F-E6A8BAEC6340}" />
    <endpoint address="mex" binding="mexNamedPipeBinding"
      bindingConfiguration=""
      contract="IMetadataExchange" />
  </service>
</services>
</system.serviceModel>
</configuration>

```

在启动服务器应用程序之前，需要修改安全设置，让用户运行应用程序，注册监听端口。否则普通的用户就不能注册监听端口了。在 Windows Vista 上，可以用 `netsh` 命令来注册。选项 `http` 修改了 HTTP 协议的 ACL。端口号和服务名用 URL 定义，用户选项指定了启动监听服务的用户名。

```
netsh http urlacl url=http://+:8088/NorthwindService user=username
```

## 44.8.2 客户程序

创建一个新的控制台应用程序 `WCFClientApp`。由于服务提供了一个 MEX 端点，因此可以选择 Visual Studio 中的菜单 `Project | Add Service Reference...`，添加一个服务引用，如图 44-14 所示。

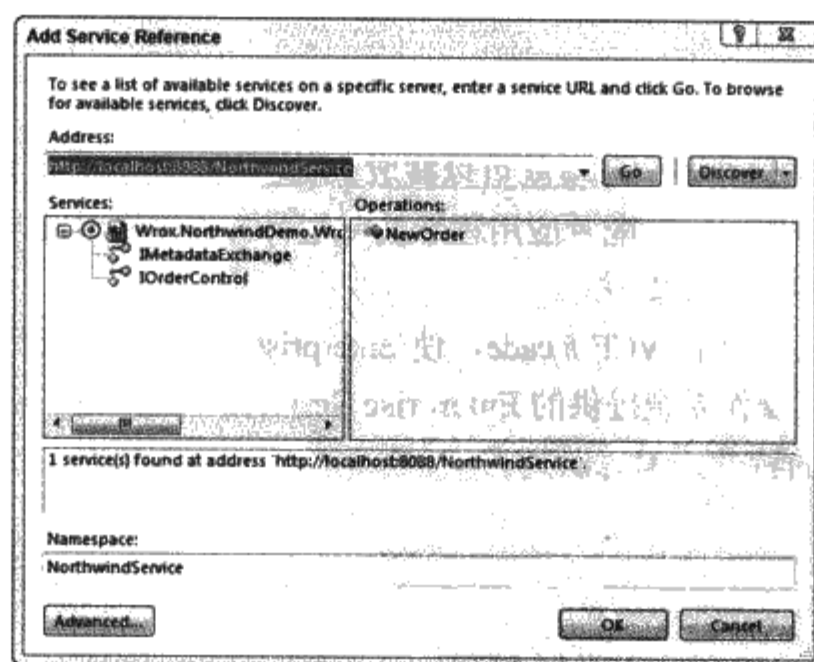


图 44-14

提示:

如果服务在 COM+ 中, 就必须在访问 MEX 数据之前启动应用程序。如果服务在 WAS 中, 应用程序会自动启动。

利用服务引用创建一个代理类, 引用 System.ServiceModel 和 System.Runtime.Serialization 程序集, 把引用服务的应用程序配置文件添加到客户程序中。

现在就可以使用生成的实体类和代理类 OrderControlClient, 给服务组件发送一个订单请求。

```
static void Main()
{
    Order order = new Order();
    order.customerId = "PICCO";
    order.orderDate = DateTime.Today;
    order.shipAddress = "Georg Pippis";
    order.shipCity = "Salzburg";
    order.shipCountry = "Austria";
    OrderLine line1 = new OrderLine();
    line1.productId = 16;
    line1.unitPrice = 17.45F;
    line1.quantity = 2;
    OrderLine line2 = new OrderLine();
    line2.productId = 67;
    line2.unitPrice = 14;
    line2.quantity = 1;
    OrderLine[] orderLines = { line1, line2 };
    order.orderLines = orderLines;

    OrderControlClient occ = new OrderControlClient();
    occ.NewOrder(order);
}
```

## 44.9 小结

本章讨论了 Enterprise Services 提供的丰富功能, 例如自动的事务处理、对象池、排队的组件和松散耦合的事件。

为了创建服务组件, 必须引用程序集 System.EnterpriseServices。所有服务组件的基类都是 ServicedComponent。有了这个类, 环境就可以截取方法调用。可以使用属性指定要使用的截取功能。我们还学习了如何使用属性配置应用程序及其组件, 如何使用属性 [Transaction] 管理事务处理, 以及指定组件的事务处理要求。

本章还讨论了如何创建一个 WCF façade, 使 Enterprise Services 与新通信技术集成起来。

本章介绍了如何使用操作系统提供的 Enterprise Services 功能, 下一章将讨论如何使用操作系统中另一个用于通信的特性: 消息队列。

# 第45章

## 消息队列

`System.Messaging` 命名空间包含的类可以用 Windows 操作系统的 Message Queuing 功能读写消息。消息传送功能可以在断开连接的环境下使用,在该环境下,客户机和服务器不需要同时运行。

本章主要内容如下:

- Message Queuing 概述
- Message Queuing 结构
- 消息队列管理工具
- 编程实现 Message Queuing
- 课程订单示例应用程序
- Message Queuing 和 WCF

### 45.1 概述

在开始给 Message Queuing 编程之前,本节先讨论消息传送的基本概念,将它与同步和异步编程比较。在同步编程中,调用一个方法时,调用者必须等待该方法执行完毕。在异步编程中,调用线程可以启动并行运行的方法。异步编程可以通过委托、支持异步方法的类库(例如 Web 服务代理, `System.Net` 和 `System.IO` 类),或使用定制的线程(详见第 19 章)来实现。在同步和异步编程中,客户机和服务器必须同时运行。

尽管 Message Queuing 是异步进行的,但因为客户机(发送者)不等待服务器(接收者)读取发送给它的的数据,所以 Messaging Queuing 与异步编程有很大的区别。Message Queuing 可以在断开连接的环境下进行。在传送数据时,接收者可以离线。在以后的某个时刻,接收者上线时,就会接收到数据,而无需发送应用程序的干预。

可以把打开连接的编程和断开连接的编程与给他人打电话和发送电子邮件做个比较。在给他人打电话时,电话两端的人都必须同时在线。这种通信是同步的。而利用电子邮件,发送者不能确定处理电子邮件的时间。使用这个技术的人利用的是断开连接的模式。当然,电子邮件可能从来不被处理,而只是被忽略。这就是断开连接的通信的本质。为了避免这个问题,可以要求发送一个回应,以确认已读取了电子邮件。如果在指定的时间内没有收到回应,就要处理这个“异常”。这也可以使用 Message Queuing 功能实现。

在某些方面,Message Queuing 就是应用程序之间通信的电子邮件,而不是人与人之间的

通信。但是，它提供了邮件服务没有的许多功能，例如有保证的交付、事务处理、确认、使用内存的快递模式等。如下一节所述，Message Queuing 有许多可用于应用程序之间通信的特性。

通过 Message Queuing 功能，可以在打开连接或断开连接的环境下发送、接收和路由消息。图 45-1 显示了使用消息的一种非常简单的方式。发送者给消息队列发送消息，接收者从队列中接收消息。

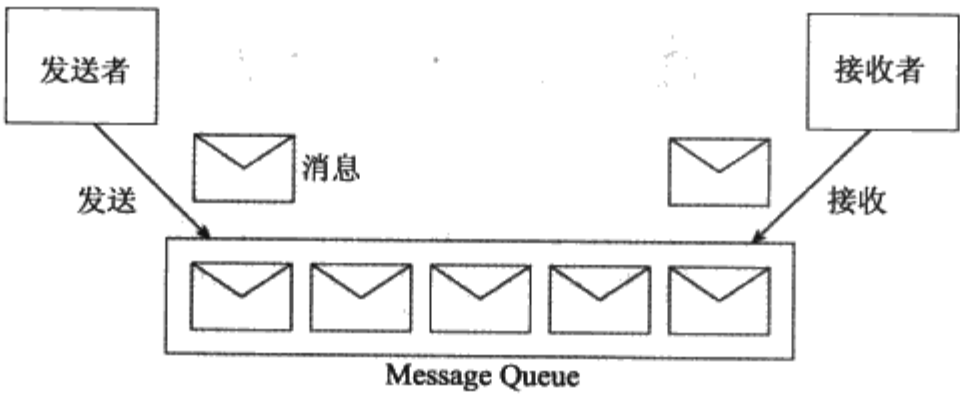


图 45-1

45.1.1 使用 Message Queuing 的场合

使用 Message Queuing 的一个场合是，客户应用程序常常从网络上断开(例如销售员在站点上访问顾客)。销售员可以直接在顾客的站点上输入订购数据。应用程序为每个订单给位于客户系统上的消息队列发送一个消息(如图 45-2 所示)。只要销售员回到办公室，订单就会自动从客户系统的消息队列传送到目标系统的消息队列上，处理消息。

除了使用笔记本电脑之外，销售员还可以使用支持 Message Queuing 的 Pocket Windows 设备。

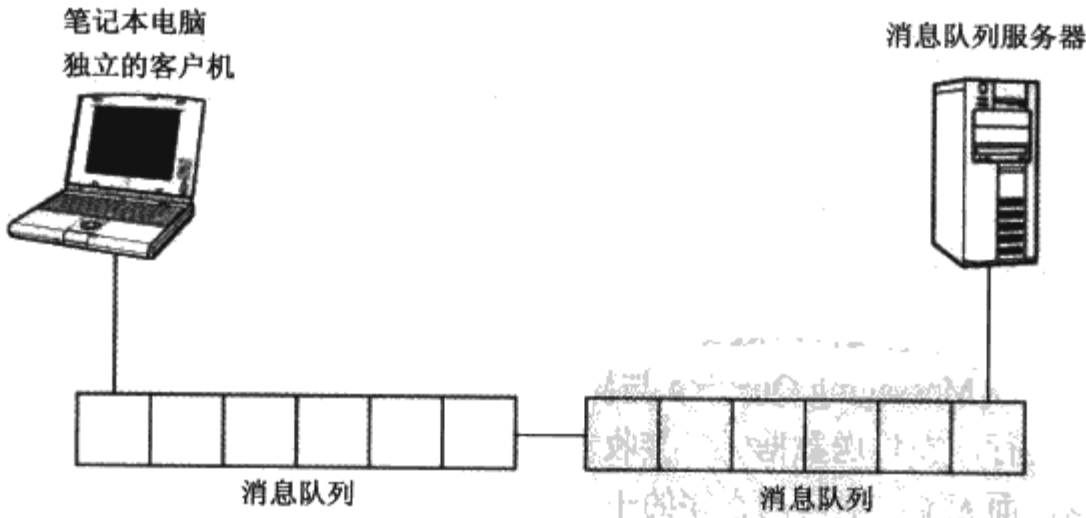


图 45-2

Message Queuing 还可以在打开连接的环境下使用。假定在一个电子商务站点上(如图 45-3 所示)，服务器在某些时刻(例如傍晚和周末)的订单事务负载是满的，但在晚上，其负载很低。一个解决方案是购买一台更快的服务器或在系统中添加更多的服务器，以处理高峰时的订单。还有一种成本较低的解决方案：把事务从高负载的时段移动到低负载的时段，即削峰平谷。在这种方案中，订单发送到消息队列中，接收端按对数据库系统有利的速度读取订单。现在，系统的负载被降低了，处理事务的服务器就可以比升级数据库系统便宜。

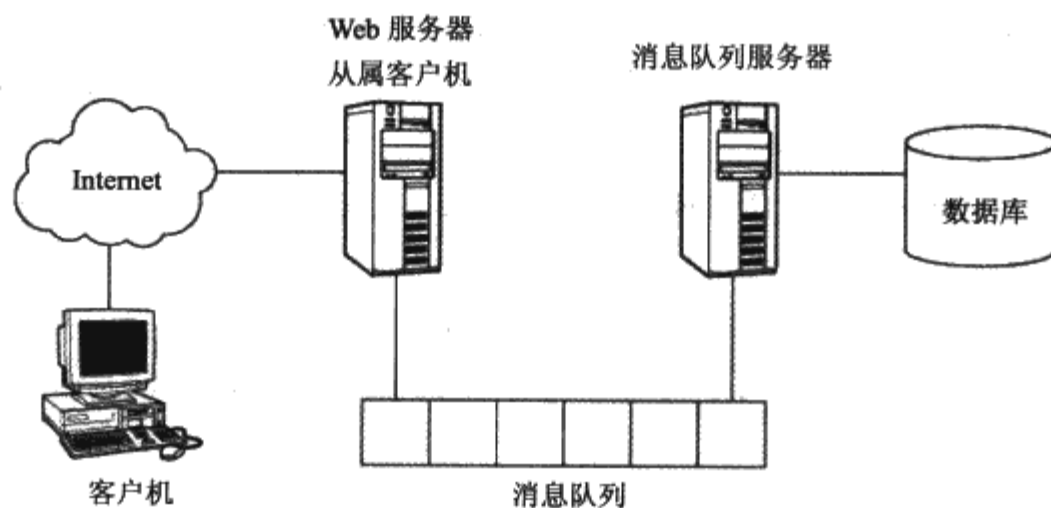


图 45-3

### 45.1.2 Message Queuing 特性

Message Queuing 是 Windows 操作系统的一部分。这个服务的主要功能如下：

- 消息可以在断开连接的环境下发送。不需要同时运行发送和接收应用程序。
- 使用快递模式，消息可以非常快地发送。在快递模式下，消息存储在内存中。
- 对于恢复机制，消息可以使用有保证的交付方式发送。可恢复的消息存储在文件中。在服务器重新启动时发送它们。
- 消息队列用访问控制表来保护，确定了哪些用户可以发送或接收队列中的消息。消息还可以加密，避免网络嗅探器读取其中的数据。消息在发送时可以指定优先级，更快地处理高优先级的项。
- Message Queuing 3.0 支持多播消息的发送。
- Message Queuing 4.0 支持带毒的消息。带毒的消息不能解析。可以在带毒的队列中移动带毒的消息。例如，如果从正常的队列中读取消息后，要把消息插入数据库，但消息不能插入数据库，因此该任务失败，消息就会发送到带毒的队列中。有人处理带毒的队列，这个人应以能解析带毒消息的方式来处理这些消息。

注意：

Message Queuing 是操作系统的一部分，所以不能在 Windows Server 2003 和 Windows XP 系统上安装 Message Queuing 4.0。Message Queuing 4.0 是 Windows Server 2008 和 Windows Vista 的一部分。

本章的剩余内容探讨这些功能的用法。

## 45.2 Message Queuing 产品

Message Queuing 4.0 是 Windows Server 2008 和 Windows Vista 的一部分。Windows 2000 使用 Message Queuing 2.0，它不支持 HTTP 协议和多播消息。Message Queuing 3.0 是 Windows



Server 2003 和 Windows XP 的一部分。使用 Windows Vista 的 Configuring Programs and Features 中的 Turn Windows Features on or off 链接时，其中有一个用于 Message Queuing 选项的部分。在这个部分中可以选择如下组件：

- Microsoft Message Queue (MSMQ) Server Core: Core 子组件是 Message Queuing 基本功能所必需的。
- Active Directory Domain Services Integration: 有了它，消息队列名就会写入 Active Directory。利用这个选项可以使用 Active Directory Integration 查找队列，用 Windows 用户和组保护队列。
- MSMQ HTTP Support: 它允许使用 HTTP 协议发送和接收消息。
- Triggers: 利用 Triggers 可以在接收新消息时实例化应用程序。
- Multicast Support: 有了它，就可以把一个消息发送给一组服务器。
- MSMQ DCOM Proxy: 有了 DCOM 代理，系统就可以使用 DCOM API 连接到远程服务器上。

在 Message Queuing 安装时，必须启动 Message Queuing 服务(如图 45-4 所示)。这个服务读写消息，与其他 Message Queuing 服务器通信，把消息路由到网络上。

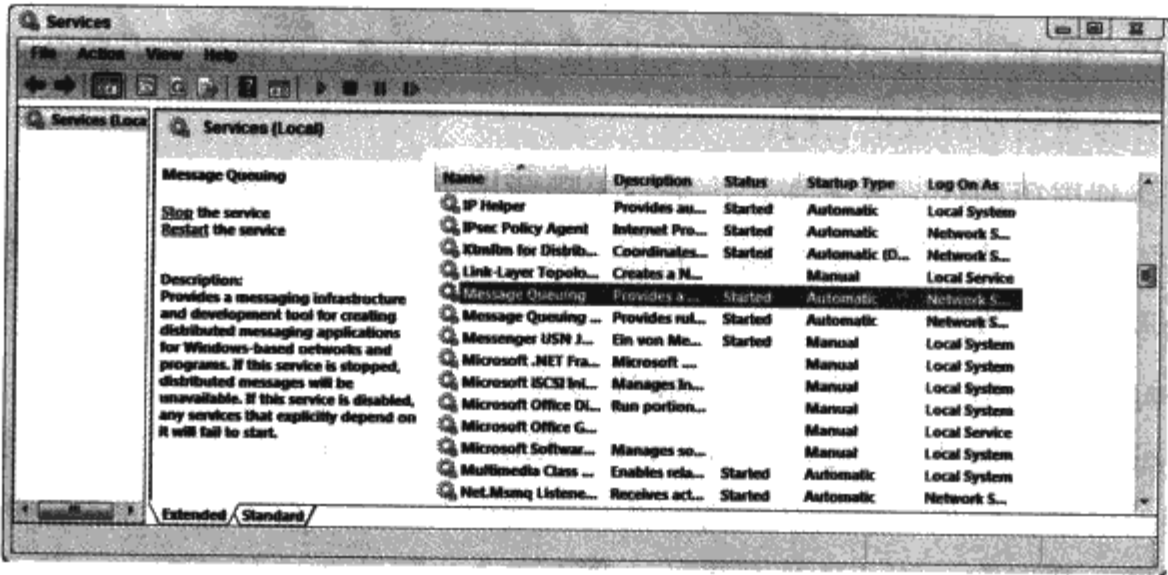


图 45-4

### 45.3 Message Queuing 结构

通过 Message Queuing，可以从消息队列中读写消息。消息和消息队列有几个属性需要进一步阐述。

#### 45.3.1 消息

消息会发送到消息队列中。消息包含消息体(包含要发送的数据)和消息标题。消息体中可以放置任意信息。在.NET 中，有几个格式化器转换要放在消息体中的数据。除了消息标题和消息体之外，消息还包含发送者、超时配置、事务 ID 或优先级等信息。

消息队列有几种类型的消息：

- 一般消息：由应用程序发送。
- 确认消息：报告一般消息的状态。确认消息发送到管理队列中，报告一般消息的发送是否成功。
- 响应消息：当最初的发送者需要某种回应时，由接收应用程序发送。
- 报告消息：由 Message Queuing 系统生成。测试消息和路由跟踪消息属于此类。

消息可以有优先级，定义消息从队列中读取出来的顺序。消息在队列中按照其优先级排序，所以下一个从队列中读取的消息就是优先级最高的那个消息。

消息有两种递送模式：快递模式和可恢复模式。快递消息的传送速度非常快，因为消息只使用内存来存储。可恢复消息在路由的每一阶段都要存储在文件中，直到消息传送到目的地为止。这样，即使计算机重新启动或网络失败，消息的递送也能得到保证。

事务消息是可恢复消息的一种特殊版本。在事务消息传送过程中，可以确保消息只到达目的地一次，且按照它们发送的顺序到达目的地。优先级不能在事务消息中使用。

### 45.3.2 消息队列

消息队列是一个消息库。存储在磁盘上的消息位于 `<windir>\system32\msmq\storage` 目录。公共或私有队列通常用于发送消息，还有其他队列类型：

- 公共队列：在 Active Directory 中发布。这些队列的信息通过 Active Directory 域复制。可以使用浏览和搜索功能获得这些队列的信息。即使不知道存储队列的计算机名，也可以访问公共队列。还可以把这种队列从一个系统移动到另一个系统上，而无需通知客户。但不能在 Workgroup 环境下创建公共队列，因为需要 Active Directory。Active Directory 详见第 46 章。
- 私有队列不在 Active Directory 中发布。这些队列只能在知道队列的完整路径名时访问。私有队列可以在 Workgroup 环境下使用。
- 日志队列：用于在发送或接收消息后，保存消息的副本。启动公共或私有队列的日志功能，就会自动创建一个日志队列。在日志队列中，可以有两种不同的队列类型：源日志队列和目标日志队列。源日志功能通过消息的属性打开，日志消息用源系统保存。目标日志功能用队列的属性打开，这些消息存储在目标系统的日志队列中。
- 死信队列：在同步编程中，错误会被立即检测出来，但使用 Message Queuing 处理错误的方式是不同的。如果消息没有在指定的时间期限前到达目标系统，该消息就存储在死信队列中。死信队列可以用于存储未到达目的地的消息。
- 管理队列：包含发送消息的确认。发送者可以指定一个管理队列，来接收消息是否成功发送的通知。
- 响应队列：如果需要把多个简单的确认消息用作接收端的回应，就可以使用响应队列。接收应用程序可以把响应消息发送回最初的发送者。

- 报告队列：用于测试消息。把公共或私有队列的类型改为预定义的 ID{55EE8F33-CCE9-11CF-B108-0020AFD61CE9}，就可以创建报告队列。报告队列可以用作跟踪路由中的消息的测试工具。
- 系统队列：是私有的，由 Message Queuing 系统使用。这些队列用于管理消息，存储确认消息，保证事务消息的正确顺序。

## 45.4 Message Queuing 管理工具

在介绍编程处理 Message Queuing 的方式之前，本节先讨论 Windows 操作系统中的管理工具，以创建和管理队列和消息。这里介绍的工具不仅能用于 Message Queuing。只有安装了 Message Queuing，才能使用这些工具的 Message Queuing 功能。

### 45.4.1 创建消息队列

消息队列可以使用 Computer Management MMC 插件创建。在 Windows Vista 系统上，用 Start | Control Panel | Administrative Tools | Computer Management 菜单启动 Computer Management MMC 插件。在树型视图面板上，Message Queuing 位于 Services and Applications 项的下面。选择 Private Queues 或 Public Queues，就可以从 Action 菜单中创建新队列，如图 45-5 所示。只有在 Active Directory 模式下配置 Message Queuing，才能使用公共队列。

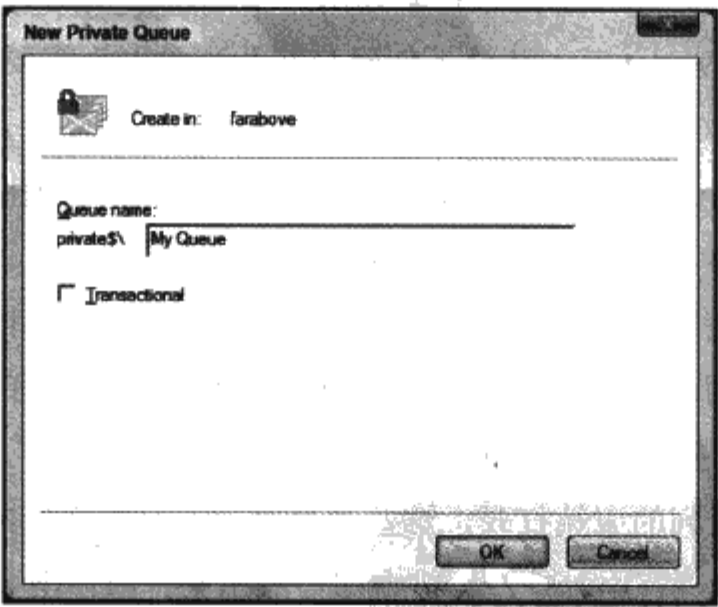


图 45-5

### 45.4.2 消息队列属性

在创建队列后，可以使用 Computer Management 插件，在树型面板上选择队列，选择 Action | Properties 菜单，来修改队列的属性，如图 45-6 所示。

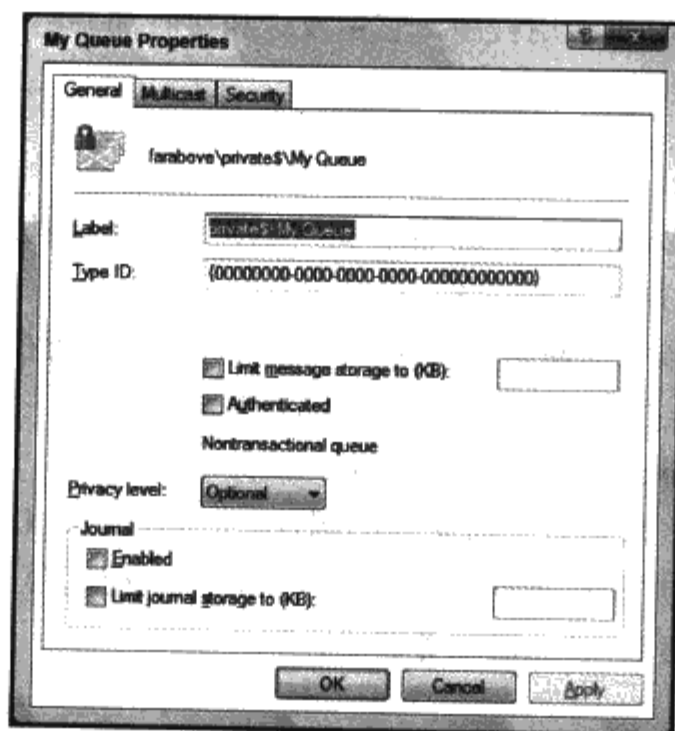


图 45-6

可以配置几个选项：

- Label 是队列的名称，可用于搜索队列。
- Type id 默认设置为 {00000000-0000-0000-0000-000000000000}，把多个队列映射为一个类别或类型。报告队列使用特定的类型 id，如前所述。类型 id 是全局唯一的 id(uuid) 或 GUID。

注意：

可以使用 uuidgen.exe 或 guidgen.exe 实用工具创建定制类型标识符。uuidgen.exe 是一个命令行实用工具，可创建唯一的 id，guidgen.exe 是创建 uuid 的图形化版本。

- 可以限制队列中所有消息的最大尺寸，使队列不至填满整个磁盘。
- Authenticated 选项只允许通过身份验证的用户读写队列中的消息。
- 使用 privacy level 选项，可以加密消息的内容。可以设置的值有 None、Optional 和 Body。None 表示不接收加密的消息，Body 只接收加密的消息，默认的 Optional 值接收两者。
- 使用 journal 设置可以配置目标日志功能。使用这个选项，可以把接收的消息副本存储到日志中。可以为队列的日志消息配置能使用的磁盘空间的最大尺寸。当到达这个最大尺寸时，就停止目标日志功能。
- 配置选项 Multicast 用于定义队列的多播 IP 地址。该多播 IP 地址可以用于网络上的不同节点，这样发送给一个地址的消息就可以用多个队列接收。

## 45.5 Message Queuing 的编程实现

理解了 Message Queuing 的结构之后，就可以探讨其编程了。下面的几节将学习如何创建和控制队列，如何发送和接收消息。

还要建立一个小型课程订单应用程序，它由发送和接收部分组成。

### 45.5.1 创建消息队列

前面了解了如何使用 Computer Management 实用工具创建消息队列。消息队列还可以用 MessageQueue 类的 Create() 方法编程创建。

在 Create() 方法中，必须传送新队列的路径。路径包括队列所在主机的名称和队列的名称。在下面的例子中，要在本地主机上创建队列 MyNewPublicQueue。为了创建私有队列，路径名必须包含 Private\$，例如 Private\$\MyNewPrivateQueue。

调用 Create() 方法之后，就可以修改队列的属性。例如，使用 Label 属性，把队列的标签设置为 Demo Queue。示例程序把队列的路径和格式名称写到控制台上。格式名称用 UUID 自动创建，UUID 可用于访问队列，且无需服务器名：

```
using System;
using System.Messaging;

namespace Wrox.ProCSharp.Messaging
{
    class Program
    {
        static void Main(string[] args)
        {
            using (MessageQueue queue = MessageQueue.Create(@".\MyNewPublicQueue"))
            {
                queue.Label = "Demo Queue";
                Console.WriteLine("Queue created:");
                Console.WriteLine("Path: {0}", queue.Path);
                Console.WriteLine("FormatName: {0}", queue.FormatName);
            }
        }
    }
}
```

注意：

创建队列时需要管理权限。通常不能让应用程序的用户拥有管理权限。这就是队列通常用安装程序创建的原因。本章的后面将介绍如何使用 MessageQueueInstaller 类创建消息队列。

### 45.5.2 查找队列

路径名和格式名可以用于标识队列。要查找队列，必须区分公共队列和私有队列。公共队列在 Active Directory 中发布。对于这些队列，无需知道它们所在的系统。私有队列只有在已知队列所在的系统名时才能找到。

在 Active Directory 域中搜索队列的标签、类别或格式名，就可以找到公共队列。还可以获得机器上的所有队列。MessageQueue 类的静态方法 GetPublicQueuesByLabel()、GetPublicQueuesByCategory() 和 GetPublicQueuesByMachine() 可以搜索队列。GetPublicQueues() 方法返回包含域中所有公共队列的数组。

```
using System;
```



```

using System.Messaging;

namespace Wrox.ProCSharp.Messaging
{
    class Program
    {
        static void Main
        {
            foreach (MessageQueue queue in MessageQueue.GetPublicQueues())
            {
                Console.WriteLine(queue.Path);
            }
        }
    }
}

```

GetPublicQueues()方法是重载的。它的一个重载版本允许传送 MessageQueueCriteria 类的一个实例。利用这个类可以搜索在某个时刻之前或之后创建或修改的队列，还可以查找队列的类别、标签或机器名。

私有队列可以使用静态方法 GetPrivateQueuesByMachine()搜索。这个方法返回指定系统中的所有私有队列。

### 45.5.3 打开已知的队列

如果队列名已知，就不需要搜索它。使用路径或格式名就可以打开队列。路径或格式名都在 MessageQueue 类的构造函数中设置。

#### 1. 路径名

路径指定了打开队列需要的机器名和队列名。下面的代码示例打开了本地主机上的队列 MyPublicQueue。为了确定队列是否存在，使用了静态方法 MessageQueue.Exists():

```

using System;
using System.Messaging;

namespace Wrox.ProCSharp.Messaging
{
    class Program
    {
        static void Main
        {
            if (MessageQueue.Exists(@".\MyPublicQueue"))
            {
                MessageQueue queue = new MessageQueue(@".\MyPublicQueue");
                //...
            }
            else
            {
                Console.WriteLine("Queue .\MyPublicQueue not existing");
            }
        }
    }
}

```

根据队列的类型，在打开队列时需要不同的标识符。表 45-1 列出了指定类型的队列名语法。

表 45-1

队 列 类 型	语 法
公共队列	MachineName\QueueName
私有队列	MachineName\Private\$\QueueName
日志队列	MachineName\QueueName\Journal\$
机器日志队列	MachineName\Journal\$
机器死信队列	MachineName\DeadLetter\$
机器事务死信队列	MachineName\XactDeadLetter\$

在使用路径名打开公共队列时，需要传送机器名。如果机器名未知，则可以使用格式名代替。私有队列的路径名只能在本地系统上使用，必须使用格式名远程访问私有队列。

2. 格式名

除了路径名之外，还可以使用格式名打开队列。格式名用于在 Active Directory 中搜索队列，获得队列所在的主机。在断开连接的环境下，消息不能发送到队列中，此时就需要使用格式名：

```
MessageQueue queue = new MessageQueue(
    @"FormatName:PUBLIC=09816AFF-3608-4c5d-B892-69754BA151FF");
```

格式名还有一些其他用途。它可以用于打开私有队列，指定要使用的协议：

- 要访问私有队列，必须给构造函数传送字符串 `FormatName:PRIVATE =Machine GUID\QueueNumber`。在创建队列时，会生成私有队列的队列号。队列号在<windows>\System32\msmq\storage\lqs 目录下。
- 使用 `FormatName:DIRECT=Protocol:MachineAddress\QueueName`，可以指定用于发送消息的协议。Message Queuing 3.0 支持 HTTP 协议。
- `FormatName:DIRECT=OS:MachineName\QueueName` 是使用格式名指定队列的另一种方式。此时不需要指定协议，但仍可以使用机器名和格式名。

45.5.4 发送消息

可以使用 MessageQueue 类的 Send 方法给队列发送消息。作为参数传送给 Send()方法的对象串行化到相关的队列上。Send()方法是重载的，这样才能传送标签和 MessageQueueTransaction 对象。Message Queuing 的事务处理在后面论述。

下面的代码示例先检查队列是否存在，如果不存在，就创建一个队列。接着打开队列，使用 Send()方法给队列传送消息 Sample Message。

路径名给服务器名指定“.”，表示它是本地系统。私有队列的路径名只能在本地使用。

```
using System;
using System.Messaging;

namespace Wrox.ProCSharp.Messaging
{
    class Program
    {
```

```
static void Main
{
    try
    {
        if (!MessageQueue.Exists(@".\Private$\MyPrivateQueue"))
        {
            MessageQueue.Create(@".\Private$\MyPrivateQueue");
        }
        MessageQueue queue = new MessageQueue(@".\Private$\MyPrivateQueue");
        queue.Send("Sample Message", "Label");
    }
    catch (MessageQueueException ex)
    {
        Console.WriteLine(ex.Message);
    }
}
```

图 45-7 显示了 Computer Management 管理工具，其中可以看到到达队列的消息。

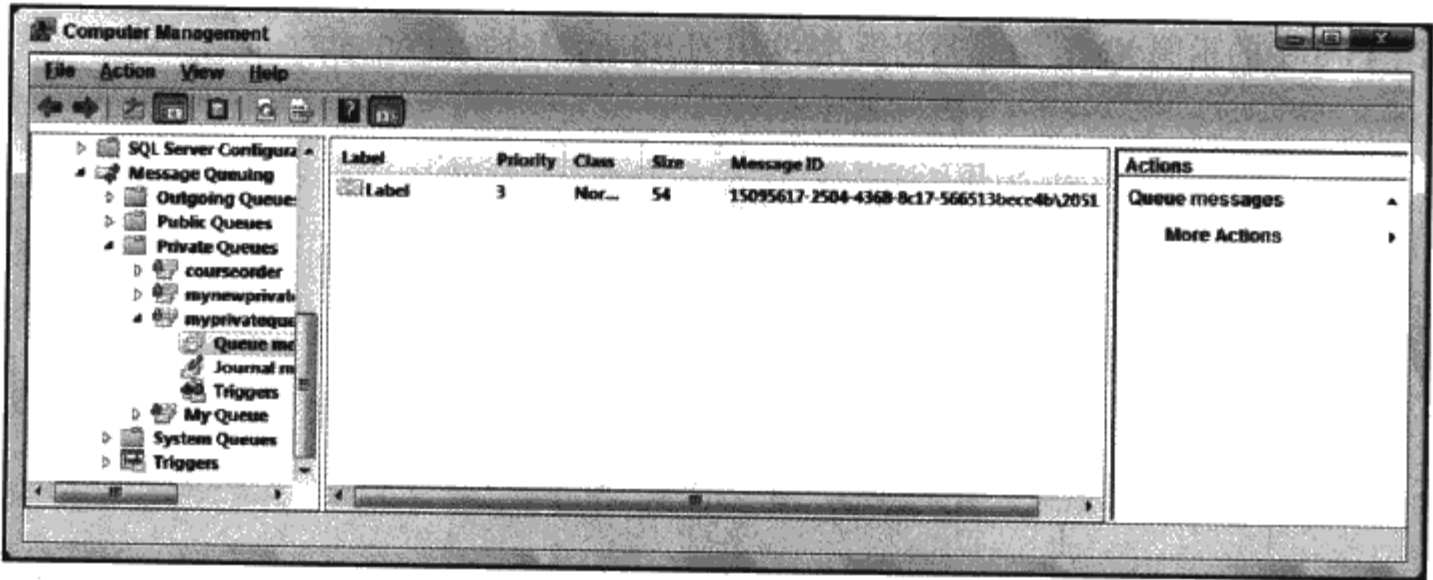


图 45-7

打开消息，选择对话框的 Body 选项卡(如图 45-8 所示)，就可以看到消息用 XML 格式化了。消息的格式化是消息队列附带的格式化器的功能。

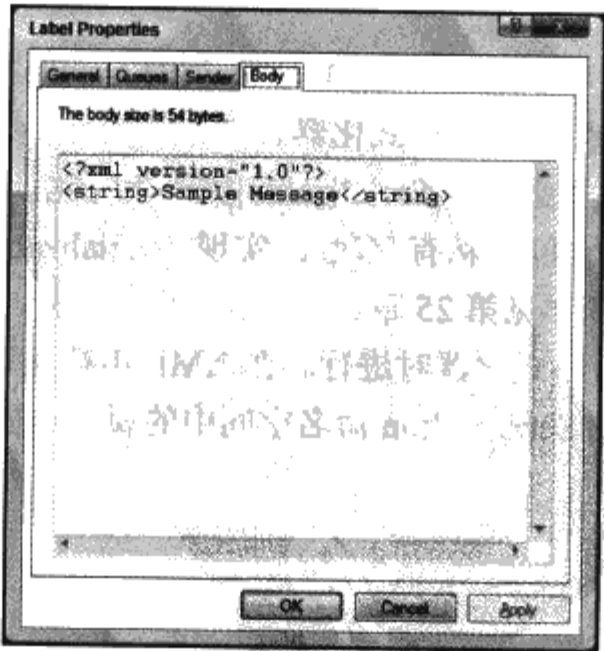


图 45-8

1. 消息格式化器

消息传送给队列的格式取决于格式化器。MessageQueue 类有一个 Formatter 属性，通过它可以指定格式化器。默认的格式化器 XmlMessageFormatter 会用 XML 语法格式化消息，如前面的例子所示。

消息格式化器实现了 IMessageFormatter 接口。System.Messaging 命名空间中有 3 个消息格式化器：

- XmlMessageFormatter 是默认的格式化器，它使用 XML 串行化对象，XML 的处理详见第 28 章。
- 使用 BinaryMessageFormatter，可以用二进制格式对消息进行串行化。这些消息比使用 XML 格式化的消息短。
- ActiveXMessageFormatter 是一个二进制格式化器，所以可以用 COM 对象读写消息。使用这个格式化器，可以用 .NET 类把消息写入队列，使用 COM 对象从队列中读取消息，反之亦然。

图 45-8 中的示例 XML 消息是用图 45-9 中的 BinaryMessageFormatter 格式化的。

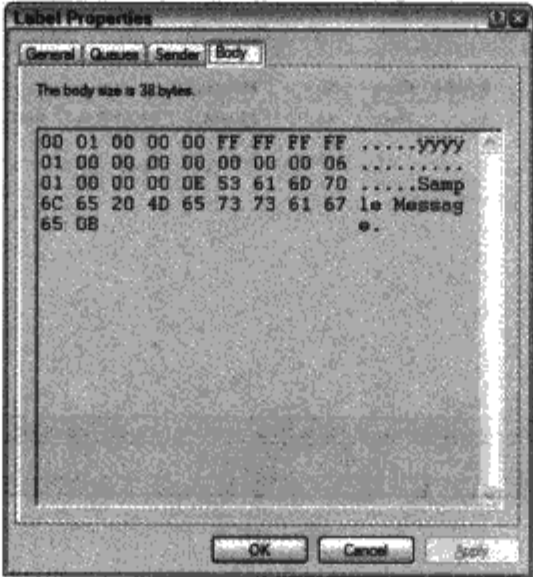


图 45-9

2. 发送复杂的消息

除了传送字符串之外，还可以给 MessageQueue 类的 Send()方法传送对象。该类的类型必须满足一些特殊的要求，但它们取决于格式化器。

对于二进制格式化器，该类必须用[Serializable]属性串行化。使用 .NET 运行库的串行化功能，对所有的字段进行串行化(包括私有字段)。实现 ISerializable 接口就可以定义定制串行化。 .NET 运行库的串行化功能详见第 25 章。

XML 串行化在使用 XML 格式化器时进行。在 XML 串行化过程中，会串行化所有的公共字段和属性。使用 System.Xml.Serialization 命名空间中的属性可以影响 XML 串行化。XML 串行化详见第 28 章。

45.5.5 接收消息

要读取消息，也可以使用 MessageQueue 类。通过 Receive()方法可以读取一个消息，再将

该消息从队列中删除。如果消息是使用不同的优先级发送的,就读取优先级最高的消息。读取优先级相同的消息时,第一个发送的消息不一定是第一个读取的,因为消息在网络中的传送顺序无法保证。要保证发送顺序和读取顺序相同,可以使用事务消息队列。

在下面的例子中,要从私有队列 `MyPrivateQueue` 中读取一个消息。之前把一个字符串传送给消息。在使用 `XmlMessageFormatter` 读取消息时,必须把要读取的对象类型传送给格式化器的构造函数。在本例中,将 `System.String` 类型传送给 `XmlMessageFormatter` 构造函数的参数数组。这个构造函数可以接收一个 `String` 数组,该数组包含把要传送为字符串的类型,也可以接收一个 `Type` 数组。

消息用 `Receive()` 方法读取,再把消息体写入控制台:

```
using System;
using System.Messaging;

namespace Wrox.ProCSharp.Messaging
{
    class Program
    {
        static void Main(string[] args)
        {
            MessageQueue queue = new MessageQueue(@".\Private$\MyPrivateQueue");
            queue.Formatter = new XmlMessageFormatter(
                new string[] { "System.String" });

            Message message = queue.Receive();
            Console.WriteLine(message.Body);
        }
    }
}
```

`Receive()` 方法将同步执行,如果队列中没有消息,就会等待队列中有消息时执行。

### 1. 枚举消息

除了使用 `Receive()` 方法逐个消息地读取之外,还可以使用枚举器迭代所有的消息。`MessageQueue` 类实现了 `IEnumerable` 接口,所以可以在 `foreach` 语句中使用。使用迭代器时,消息不会从队列中删除,但可以查看它们的内容:

```
MessageQueue queue = new MessageQueue(@".\Private$\MyPrivateQueue");
queue.Formatter = new XmlMessageFormatter(
    new string[] { "System.String" });

foreach (Message message in queue)
{
    Console.WriteLine(message.Body);
}
```

如果不使用 `IEnumerable` 接口,还可以使用 `MessageEnumerator` 类。`MessageEnumerator` 实现了 `IEnumerator` 接口,有更多的特性。实现 `IEnumerable` 接口,就表示不从队列中删除消息。`MessageEnumerator` 类的 `RemoveCurrent()` 方法可以从枚举器的当前光标位置删除消息。

在下面的例子中,使用 `MessageQueue` 类的 `GetMessageEnumerator()` 方法访问 `MessageEnumerator`。通过 `MessageEnumerator` 的 `MoveNext()` 方法,可以逐个访问消息。`MoveNext()` 方



法重载为把一个时间段作为参数。这是使用这个枚举器的一个主要优点。现在，线程可以在指定的时间段内等待消息进入队列，之后就不等待了。IEnumerator 接口定义的 Current 属性返回消息的一个引用：

```
MessageQueue queue = new MessageQueue(@".\Private$\MyPrivateQueue");
queue.Formatter = new XmlMessageFormatter(
    new string[] { "System.String" });

using (MessageEnumerator messages = queue.GetMessageEnumerator())
{
    while (messages.MoveNext(TimeSpan.FromMinutes(30)))
    {
        Message message = messages.Current;
        Console.WriteLine(message.Body);
    }
}
```

## 2. 异步读取

MessageQueue 类的 Receive 方法会等到队列中的消息可以读取为止。为了避免阻碍线程的执行，可以在 Receive 方法的一个重载版本中指定一个超时设置。要从超时后读取队列中的消息，必须再次调用 Receive() 方法。这里不是查询消息，而可以调用 BeginReceive() 异步方法。在使用 BeginReceive() 开始异步读取之前，应设置 ReceiveCompleted 事件。ReceiveCompleted 事件需要委托 ReceiveCompletedEventHandler，该委托引用在消息到达队列并可以读取时要调用的方法。在下面的例子中，把 MessageArrived 方法传送给委托 ReceivedCompletedEventHandler：

```
MessageQueue queue = new MessageQueue(@".\Private$\MyPrivateQueue");
queue.Formatter = new XmlMessageFormatter(
    new string[] { "System.String" });

queue.ReceiveCompleted +=
    new ReceiveCompletedEventHandler(MessageArrived);
queue.BeginReceive();
// thread does not wait
```

MessageArrived 处理程序需要两个参数。第一个参数是事件源 MessageQueue。第二个参数是 ReceiveCompletedEventArgs 类型，它包含消息和异步结果。在下面的例子中，调用了队列中的 EndReceive() 方法，以获得异步方法的结果，即消息：

```
public static void MessageArrived(object source,
    ReceiveCompletedEventArgs e)
{
    MessageQueue queue = (MessageQueue)source;
    Message message = queue.EndReceive(e.AsyncResult);
    Console.WriteLine(message.Body);
}
```

如果消息不应从队列中删除，那么 BeginPeek() 和 EndPeek() 方法就可以与异步 I/O 一起使用。

## 45.6 课程订单应用程序

为了演示 Message Queuing 的用法，本节将创建一个示例解决方案，用于订购课程。示例

解决方案由 3 个程序集组成：

- 组件库(CourseOrder)，它包含收发队列中的消息的实体类
- WPF 应用程序(CourseOrderSender)，给消息队列发送消息
- WPF 应用程序(CourseOrderReceiver)，从消息队列中接收消息

#### 45.6.1 课程订单类库

发送和接收应用程序都需要订单信息。所以，把实体类放在一个单独的程序集中。CourseOrder 程序集包含 3 个实体类：CourseOrder、Course 和 Customer。在示例应用程序中，并不像真正的应用程序那样实现所有的属性，而只实现能演示概念的属性。

在 Course.cs 文件中定义了 Course 类。这个类只有一个属性，表示课程的标题：

```
using System;

namespace Wrox.ProCSharp.Messaging
{
    public class Course
    {
        public string Title{ get; set; }
    }
}
```

文件 Customer.cs 包含 Customer 类，该类的属性用于表示公司和联系人姓名：

```
using System;

namespace Wrox.ProCSharp.Messaging
{
    public class Customer
    {
        public string Company{ get; set; }
        public string Contact{ get; set; }
    }
}
```

CourseOrder 类在 CourseOrder.cs 文件中，它映射订单中的一个顾客和一个课程，并确定订单是否有高优先级：

```
namespace Wrox.ProCSharp.Messaging
{
    public class CourseOrder
    {
        public Customer Customer{ get; set; }
        public Course Course{ get; set; }
    }
}
```

#### 45.6.2 课程订单消息发送程序

解决方案的第二部分是一个 Windows 应用程序 CourseOrderSender。在这个应用程序中，把课程订单发送给消息队列。必须引用 System.Messaging 和 CourseOrder 程序集。

这个应用程序的用户界面如图 45-10 所示。comboBoxCourses 组合框的项包括几个课程，

例如 Advanced .NET Programming、Programming with LINQ、和 Distributed Application Development using WCF。

图 45-10

单击 Submit the Order 按钮，就调用 buttonSubmit\_Click() 处理程序。在这个方法中，创建了一个 CourseOrder 对象，并用 TextBox 和 ComboBox 控件的内容填充它。接着创建一个 MessageQueue 实例，以打开带有格式名的公共队列。即使当前不能访问队列，也可以使用格式名发送消息。我们可以使用 Computer Management 插件获得格式名，来读取消息队列的 id。使用 Send() 方法，传送 CourseOrder 对象，用默认的 XmlMessageFormatter 串行化它，再把它写入队列：

```
private void buttonSubmit_Click(object sender, RoutedEventArgs e)
{
    try
    {
        CourseOrder order = new CourseOrder();
        order.Course = new Course()
        {
            Title = comboBoxCourses.SelectedItem.ToString()
        };
        order.Customer = new Customer()
        {
            Company = textCompany.Text,
            Contact = textContact.Text
        };
        using (MessageQueue queue = new MessageQueue(
            "FormatName:Public=D99CE5F3-4282-4a97-93EE-E9558B15EB13"))
        {
            queue.Send(order, String.Format("Course Order {{0}}",
                order.Customer.Company));
        }
        MessageBox.Show("Course Order submitted", "Course Order",
            MessageBoxButton.OK, MessageBoxImage.Information);
    }
    catch (MessageQueueException ex)
    {
        MessageBox.Show(ex.Message, "Course Order Error",
            MessageBoxButton.OK, MessageBoxImage.Error);
    }
}
```

### 45.6.3 发送优先级和可恢复的消息

通过设置 `Message` 类的 `Priority` 属性，就可以给消息指定优先级。如果消息是特别配置的，就必须创建一个 `Message` 对象，其中消息体在构造函数中传送。

在这个例子中，如果选中了 `checkBoxPriority` 复选框，优先级就设置为 `MessagePriority.High`。`MessagePriority` 是一个枚举，可以把值设置为从 `Lowest (0)` 到 `Highest (7)`，默认值 `Normal` 的优先级是 3。

为了使消息可以恢复，应把 `Recoverable` 属性设置为 `true`：

```
private void buttonSubmit_Click(object sender, RoutedEventArgs e)
{
    try
    {
        CourseOrder order = new CourseOrder();
        order.Course = new Course()
        {
            Title = comboBoxCourses.SelectedItem.ToString()
        };
        order.Customer = new Customer()
        {
            Company = textCompany.Text,
            Contact = textContact.Text
        };
        using (MessageQueue queue = new MessageQueue(
            "FormatName:Public=D99CE5F3-4282-4a97-93EE-E9558B15EB13"))
        using (Message message = new Message(order))
        {
            if (checkBoxPriority.IsChecked == true)
            {
                message.Priority = MessagePriority.High;
            }
            message.Recoverable = true;
            queue.Send(message, String.Format("Course Order {{{0}}}",
                order.Customer.Company));
        }
        MessageBox.Show("Course Order submitted");
    }
    catch (MessageQueueException ex)
    {
        MessageBox.Show(ex.Message, "Course Order Error",
            MessageBoxButton.OK, MessageBoxImage.Error);
    }
}
```

运行应用程序，就可以把课程订单添加到消息队列中，如图 45-11 所示。

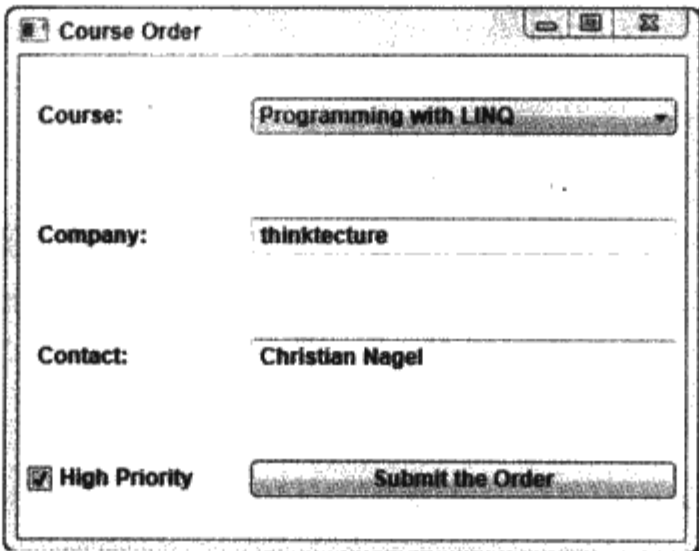


图 45-11

45.6.4 课程订单消息接收程序

课程订单接收程序从队列中读取消息，其设计视图如图 45-12 所示。这个应用程序在列表框 listOrders 中显示每个订单的标签。选中了一个订单后，订单的内容就会显示在应用程序右边的控件中。

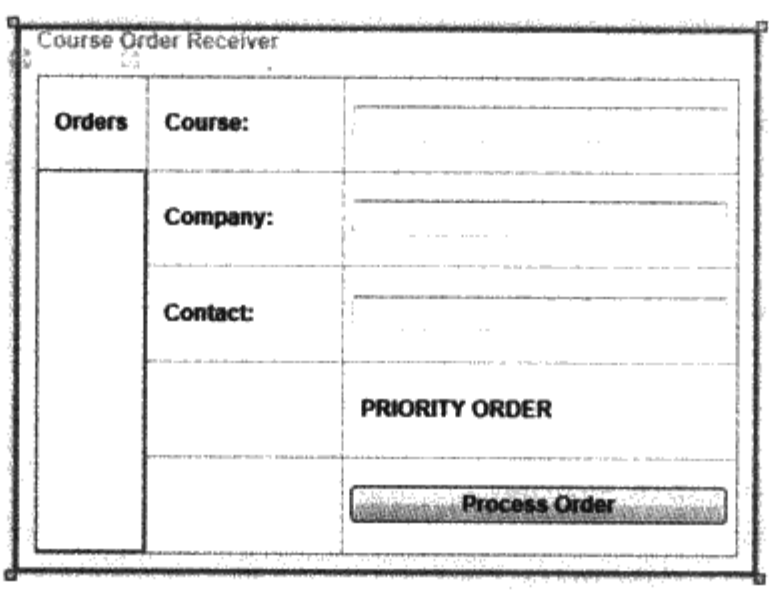


图 45-12

在窗体类 CourseOrderReceiverForm 的构造函数中，创建了一个 MessageQueue 对象，它引用由发送程序使用的队列。要读取消息，应使用 Formatter 属性把 XmlMessage Formatter、读取的类型和队列关联起来。

要在列表中显示可用的消息，应创建一个新线程，在后台上读取消息。该线程的主方法是 PeekMessages。

注意：  
线程的更多内容详见第 19 章。

```
using System;  
using System.Messaging;
```



```

using System.Threading;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Threading;

namespace Wrox.ProCSharp.Messaging
{
    public partial class CourseOrderReceiverWindow : Window
    {
        private MessageQueue orderQueue;

        public CourseOrderReceiverWindow()
        {
            InitializeComponent();

            string queueName =
                "FormatName:Public=D99CE5F3-4282-4a97-93EE-E9558B15EB13";

            orderQueue = new MessageQueue(queueName);
            System.Type[] types = new Type[]
            {
                typeof(CourseOrder),
                typeof(Customer),
                typeof(Course)
            };
            orderQueue.Formatter = new XmlMessageFormatter(types);

            // start the thread that fills the ListBox with orders
            Thread t1 = new Thread(PeekMessages);
            t1.IsBackground = true;
            t1.Start();
        }
    }
}

```

线程的主方法 `PeekMessages()` 使用消息队列的枚举器显示所有的消息。在 `while` 循环中，要检查队列中是否还有新消息。如果队列中没有消息，线程就等待新消息进入队列，等待 3 个小时后才会弹出。

要在列表框中显示队列中的每个消息，线程不能直接把文本写入列表框，而必须把调用传送给列表框的创建线程。Windows 窗体控件绑定到一个线程上，所以只能使用该创建线程访问方法和属性。`Invoke()` 方法把请求写入创建线程：

```

private delegate void MethodInvoker(LabelIdMapping labelIdMapping);

private void PeekMessages()
{
    using (MessageEnumerator messagesEnumerator =
        orderQueue.GetMessageEnumerator2())
    {
        while (messagesEnumerator.MoveNext(TimeSpan.FromHours(3)))
        {
            LabelIdMapping labelId = new LabelIdMapping()
            {
                Id = messagesEnumerator.Current.Id,
                Label = messagesEnumerator.Current.Label
            };
            Dispatcher.Invoke(DispatcherPriority.Normal,
                new MethodInvoker(AddListItem), labelId);
        }
    }
}

```

```

    }
    MessageBox.Show("No orders in the last 3 hours. Exiting thread",
        "Course Order Receiver", MessageBoxButton.OK,
        MessageBoxImage.Information);
}
private void AddListItem(LabelIdMapping labelIdMapping)
{
    listOrders.Items.Add(labelIdMapping);
}

```

ListBox 控件包含 LabelIdMapping 类的元素。这个类用于在列表框中显示消息的标签，但隐藏消息的 id。消息的 id 可以用于以后读取消息：

```

private class LabelIdMapping
{
    private string Label { get; set; }
    private string Id { get; set; }
    public override string ToString()
    {
        return label;
    }
}

```

ListBox 控件的 SelectedIndexChanged 事件与 OnOrderSelectionChanged()方法是关联的。这个方法从当前选中的项中获得 LabelIdMapping 对象，使用 id 通过 PeekById()方法再次访问消息。接着在 TextBox 控件中显示消息的内容。消息的优先级默认为不读取，所以属性 MessageReadPropertyFilter 必须设置为接收 Priority：

```

private void listOrders_SelectionChanged(object sender,
    RoutedEventArgs e)
{
    LabelIdMapping labelId = listOrders.SelectedItem as LabelIdMapping;
    if (labelId == null)
        return;

    orderQueue.MessageReadPropertyFilter.Priority = true;
    Message message = orderQueue.PeekById(labelId.Id);

    CourseOrder order = message.Body as CourseOrder;
    if (order != null)
    {
        textCourse.Text = order.Course.Title;
        textCompany.Text = order.Customer.Company;
        textContact.Text = order.Customer.Contact;
        buttonProcessOrder.IsEnabled = true;
        if (message.Priority > MessagePriority.Normal)
        {
            labelPriority.Visibility = Visibility.Visible;
        }
        else
        {
            labelPriority.Visibility = Visibility.Hidden;
        }
    }
    else
    {
        MessageBox.Show("The selected item is not a course order",
            "Course Order Receiver", MessageBoxButton.OK,

```

```

        MessageBoxImage.Warning);
    }
}

```

单击 **Process Order** 按钮时，会调用 `OnProcessOrder()` 处理程序。这里会再次引用列表框中当前选择的消息，并调用 `ReceiveById()` 方法从队列中删除该消息：

```

private void buttonProcessOrder_Click(object sender, RoutedEventArgs e)
{
    LabelIdMapping labelId = listOrders.SelectedItem as LabelIdMapping;
    Message message = orderQueue.ReceiveById(labelId.Id);

    listOrders.Items.Remove(labelId);
    listOrders.SelectedIndex = -1;
    buttonProcessOrder.Enabled = false;
    textCompany.Text = string.Empty;
    textContact.Text = string.Empty;
    textCourse.Text = string.Empty;

    MessageBox.Show("Course order processed", "Course Order Receiver",
        MessageBoxButton.OK, MessageBoxImage.Information);
}
}
}

```

图 45-13 显示了运行着的接收程序，此时队列中有 3 个订单，且当前选中了一个订单。

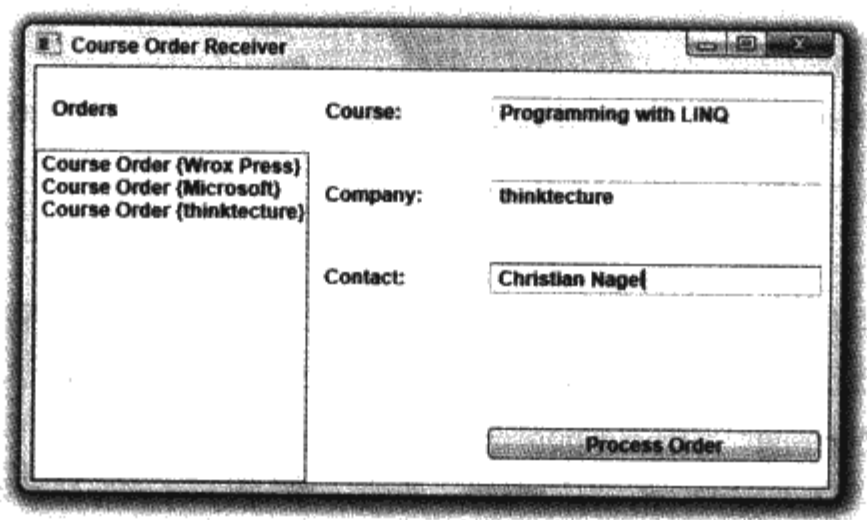


图 45-13

## 45.7 接收结果

在示例应用程序的当前版本中，发送程序并不知道消息是否已处理。为了得到接收程序的结果，可以使用确认队列或响应队列。

### 45.7.1 确认队列

使用确认队列，发送应用程序可以获得消息的状态信息。在确认消息中，可以确定是否要接收回应，来判断是一切正常，还是出错。例如，可以在消息到达目标队列或读取消息时，发送确认消息，或在指定的时间过后，消息未到达目标队列或未读取消息时，发送确认消息。

在下面的例子中，将 `Message` 类的 `AdministrationQueue` 设置为 `CourseOrderAck` 队列。这个队列的创建方式类似于一般队列，但它以另外一种方式使用：原发送程序接收到确认消息时使用它。`AcknowledgementType` 属性设置为 `AcknowledgementTypes.FullReceive`，会在读取消息时获得一个确认消息：

```
Message message = new Message(order);

message.AdministrationQueue =
    new MessageQueue(@".\CourseOrderAck");
message.AcknowledgementType = AcknowledgementTypes.FullReceive;

queue.Send(message, String.Format("Course Order {{0}}",
    order.Customer.Company);
queue.Send(message, "Course Order {" +
    order.Customer.Company + "}");

string id = message.Id;
```

关联 id 用于确定哪个确认消息属于什么消息。每个发送的消息都有一个 id，响应该消息的确认消息把源消息的 id 作为其关联 id。确认队列中的消息可以用 `MessageQueue.ReceiveByCorrelationId()` 方法读取，以接收相关的确认消息。

除了使用确认消息之外，还可以为没有到达目的地的消息使用死信队列。把 `Message` 类的属性 `UseDeadLetterQueue` 设置为 `true`，如果消息在指定的时间过后没有到达目标队列，该消息就会复制到死信队列中。

超时时间用 `Message` 的属性 `TimeToReachQueue` 和 `TimeToBeReceived` 设置。

### 45.7.2 响应队列

如果需要从接收程序中获得多个确认信息，就可以使用响应队列。响应队列类似于一般队列，但源发送程序把该队列用作接收器，源接收程序把响应队列用作发送器。

发送程序必须用 `Message` 类的 `ResponseQueue` 属性指定响应队列。下面的示例代码演示了接收程序如何使用响应队列返回一个响应消息。在响应消息 `responseMessage` 中，`CorrelationId` 属性设置为源消息的 id。这样，客户程序就知道回应属于哪个消息了。这类似于确认队列。响应消息用 `MessageQueue` 对象的 `Send()` 方法发送，`MessageQueue` 对象是从 `ResponseQueue` 属性中返回的：

```
public void ReceiveMessage(Message message)
{
    Message responseMessage = new Message("response");
    responseMessage.CorrelationId = message.Id;

    message.ResponseQueue.Send(responseMessage);
}
```

## 45.8 事务队列

对于可恢复的消息来说，不能保证消息的接收顺序，也不能保证消息只接收一次。网络失

败可能会使消息被接收多次，如果发送程序和接收程序安装了供 Message Queuing 使用的多个网络协议，也会发生这种情况。

在需要确保如下条件的情况下，可以使用事务队列：

- 消息的接收顺序与其发送顺序相同
- 消息只接收一次

对于事务队列，一个事务不能横跨消息的发送和接收过程。Message Queuing 的本质是发送和接收的时间间隔可能非常长。而事务处理是很短的。在 Message Queuing 中，第一个事务把消息发送到队列中，第二个事务把消息写入网络，第三个事务用于接收消息。

下面的例子演示了如何创建事务消息队列，如何使用事务发送消息。

给 MessageQueue.Create()方法的第二个参数传送 true，就创建了一个事务消息队列。

如果要在一个事务中把多个消息写入队列，就必须实例化 MessageQueueTransaction 对象，调用 Begin()方法。在发送完属于该事务的所有消息后，必须调用 MessageQueue Transaction 对象的 Commit()方法。要取消一个事务(且不把消息写入队列)，就必须在 catch 块中调用 Abort()方法，如下所示：

```
using System;
using System.Messaging;

namespace Wrox.ProCSharp.Messaging
{
    class Program
    {
        static void Main(string[] args)
        {
            if (!MessageQueue.Exists(@".\MyTransactionalQueue"))
            {
                MessageQueue.Create(@".\MyTransactionalQueue", true);
            }
            MessageQueue queue = new MessageQueue(@".\MyTransactionalQueue");
            MessageQueueTransaction transaction =
                new MessageQueueTransaction();
            try
            {
                transaction.Begin();
                queue.Send("a", transaction);
                queue.Send("b", transaction);
                queue.Send("c", transaction);
                transaction.Commit();
            }
            catch
            {
                transaction.Abort();
            }
        }
    }
}
```

## 45.9 消息队列和 WCF

第 42 章介绍了 WCF 的体系结构和核心功能。使用 WCF 可以配置一个使用 Windows



Message Queuing 体系结构的消息队列绑定。WCF 通过它为消息队列提供了一个抽象层。图 45-14 用一个简单的图解释了该体系结构。客户应用程序调用 WCF 代理的一个方法，给队列发送一个消息。该消息是由代理创建的。客户开发人员不需要知道消息发送给了队列，而只是调用代理的方法即可。代理抽象了处理 System.Messaging 命名空间中的类的过程，给队列发送一个消息。服务器端的 MSMQ 监听器信道从队列中读取消息，把它们转换为方法调用，再用服务调用方法。

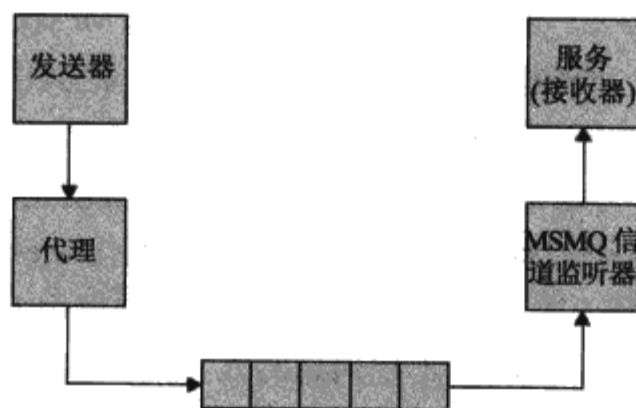


图 45-14

接着，从 WCF 的角度来看，课程预订应用程序转换为使用消息队列。在这个解决方案中，修改前面的 3 个项目，再添加一个程序集，其中包含 WCF 服务的合同。

- 组件库 (CourseOrder) 包含在线上发送消息的实体类。这些实体类改为满足数据合同的要求，与 WCF 进行串行化。
- 添加一个新库 (CourseOrderService)，它定义了服务提供的合同。
- 修改 WPF 发送程序 (CourseOrderSender)，不发送消息，而调用 WCF 代理的方法。
- 修改 WPF 接收程序 (CourseOrderReceiver)，使用实现了合同的 WCF 服务。

#### 45.9.1 带数据合同的实体类

在 CourseOrder 库中，修改类 Course、Customer 和 CourseOrder，以应用数据合同和属性 [DataContract] 及 [DataMember]。要使用这些属性，必须引用程序集 System.Runtime.Serialization，导入命名空间 System.Runtime.Serialization。

```
using System.Runtime.Serialization;

namespace Wrox.ProCSharp.Messaging
{
    [DataContract]
    public class Course
    {
        [DataMember]
        public string Title { get; set; }
    }
}
```

Customer 类还需要数据合同属性：

```
[DataContract]
public class Customer
{
```

```

    [DataMember]
    public string Company { get; set; }

    [DataMember]
    public string Contact { get; set; }
}

```

对于 `CourseOrder` 类, 不仅要添加数据合同属性, 还要添加 `ToString()` 方法的一个重写版本, 以包含这些对象的默认字符串表示:

```

[DataContract]
public class CourseOrder
{
    [DataMember]
    public Customer Customer { get; set; }

    [DataMember]
    public Course Course { get; set; }

    public override string ToString()
    {
        return String.Format("Course Order {{{0}}}", Customer.Company);
    }
}

```

### 45.9.2 WCF 服务合同

为了给服务提供 WCF 服务合同, 需要添加一个 WCF 服务库 `CourseOrderService-Contract`。这个合同由 `ICourseOrderService` 接口定义, 它需要属性 `[ServiceContract]`。如果仅允许把这个接口用于消息队列, 就可以应用属性 `[DeliveryRequirements]`, 并指定属性 `QueuedDeliveryRequirements`。枚举 `QueuedDeliveryRequirements` 的值有 `Required`、`Allowed` 和 `NotAllowed`。方法 `AddCourseOrder()` 是服务提供的。消息队列使用的方法只能有输入参数。发送器和接收器都可以彼此独立地运行, 所以发送器不能期望立即得到结果。使用 `[OperationContract]` 属性设置 `IsOneWay` 特性。这个操作的调用者不等待服务的回应:

```

using System.ServiceModel;

namespace Wrox.ProCSharp.Messaging
{
    [ServiceContract]
    [DeliveryRequirements(
        QueuedDeliveryRequirements=QueuedDeliveryRequirementsMode.Required)]
    public interface ICourseOrderService
    {
        [OperationContract(IsOneWay = true)]
        void AddCourseOrder(CourseOrder courseOrder);
    }
}

```

**提示:**

可以使用确认和响应队列给客户提供回应。

## 45.9.3 WCF 消息接收程序

WPF 应用程序 `CourseOrderReceiver` 现在修改为实现 WCF 服务，接收消息。它需要引用程序集 `System.ServiceModel` 和 WCF 合同程序集 `CourseOrderServiceContract`。

类 `CourseOrderService` 实现了接口 `ICourseOrderService`。在实现代码中，触发事件 `CourseOrderAdded`。WPF 应用程序注册这个事件，是为了接收 `CourseOrder` 对象。

WPF 控件绑定到一个线程上，所以属性 `UseSynchronizationContext` 用 `[ServiceBehavior]` 属性设置。这是 WCF 运行库的一个特性，可以把方法调用传送给 WPF 应用程序的同步环境定义的线程上。

```
using System.ServiceModel;

namespace Wrox.ProCSharp.Messaging
{
    public delegate void CourseOrderInfoHandler(CourseOrder courseOrder);

    [ServiceBehavior(UseSynchronizationContext=true)]
    public class CourseOrderService : ICourseOrderService
    {
        public static event CourseOrderInfoHandler CourseOrderAdded;

        public void AddCourseOrder(CourseOrder courseOrder)
        {
            if (CourseOrderAdded != null)
                CourseOrderAdded(courseOrder);
        }
    }
}
```

**提示：**

第 19 章介绍了同步环境。

在类 `CourseReceiverWindow` 的构造函数中，实例化了一个 `ServiceHost` 对象，并打开它，以启动监听器。监听器的绑定在应用程序配置文件中完成。

在构造函数中，订阅了 `CourseOrderReceiver` 的 `CourseOrderAdded` 事件。这里只需把接收到的 `CourseOrder` 对象添加到集合中，所以使用了一个简单的  $\lambda$  表达式。

**提示：**

第 7 章介绍了  $\lambda$  表达式。

这里使用的集合类是 `System.Collections.ObjectModel` 命名空间中的 `ObservableCollection<T>`。这个集合类实现了 `INotifyCollectionChanged` 接口，因此绑定到集合上的 WPF 控件知道列表的动态变化。

```
using System;
using System.Collections.ObjectModel;
using System.ServiceModel;
using System.Windows;

namespace Wrox.ProCSharp.Messaging
{
```

```

public partial class CourseOrderReceiverWindow : Window
{
    private ObservableCollection < CourseOrder > courseOrders =
        new ObservableCollection < CourseOrder > ();

    public CourseOrderReceiverWindow()
    {
        InitializeComponent();

        CourseOrderService.CourseOrderAdded +=
            courseOrder => courseOrders.Add(courseOrder);
        ServiceHost host = new ServiceHost(typeof(CourseOrderService));
        try
        {
            host.Open();
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
        this.DataContext = courseOrders;
    }
}

```

XAML 代码中的 WPF 元素现在使用了数据绑定。ListBox 绑定到数据环境中，单项控件绑定到数据环境的当前项的属性上。

```

< ListBox Grid.Row="1" x:Name="listOrders" ItemsSource="{Binding}"
    IsSynchronizedWithCurrentItem="true" / >

<!-- ... -->

< TextBox x:Name="textCourse" Grid.Row="0" Grid.Column="1"
    Text="{Binding Path=Course.Title}" / >
< TextBox x:Name="textCompany" Grid.Row="1" Grid.Column="1"
    Text="{Binding Path=Customer.Company}" / >
< TextBox x:Name="textContact" Grid.Row="2" Grid.Column="1"
    Text="{Binding Path=Customer.Contact}" / >

```

应用程序配置文件定义了 netMsmqBinding。为了可靠地传送消息，需要事务队列。要收发非事务队列中的消息，必须把 exactlyOnce 属性设置为 false。

#### 提示：

如果接收程序和发送程序都是 WCF 程序，就可以使用 netMsmqBinding 绑定。如果其中一个程序使用 System.Messaging API 发送或接收消息，或者其中一个程序是旧的 COM 应用程序，就可以使用 msmqIntegrationBinding。

```

< ?xml version="1.0" encoding="utf-8" ? >
< configuration >
    < system.serviceModel >
        < bindings >
            < netMsmqBinding >
                < binding name="NonTransactionalQueueBinding" exactlyOnce="false" >
                    < security mode="None" / >
                < /binding >
            < /netMsmqBinding >
        < /bindings >
    < /system.serviceModel >
< /configuration >

```

```

    < service name="Wrox.ProCSharp.Messaging.CourseOrderService" >
      < endpoint address="net.msmq://localhost/private/courseorder"
        binding="netMsmqBinding"
        bindingConfiguration="NonTransactionalQueueBinding"
        name="OrderQueueEP"
        contract="Wrox.ProCSharp.Messaging.ICourseOrderService" / >
    < /service >
  < /services >
< /system.serviceModel >
< /configuration >

```

**buttonProcessOrder** 按钮的 Click 事件处理程序从集合类中删除了选中的课程订单:

```

private void buttonProcessOrder_Click(object sender, RoutedEventArgs e)
{
    CourseOrder courseOrder = listOrders.SelectedItem as CourseOrder;
    courseOrders.Remove(courseOrder);
    listOrders.SelectedIndex = -1;
    buttonProcessOrder.IsEnabled = false;

    MessageBox.Show("Course order processed", "Course Order Receiver",
        MessageBoxButton.OK, MessageBoxImage.Information);
}

```

#### 45.9.4 WCF 消息发送程序

发送程序修改为使用 WCF 代理类。对于服务合同, 需要引用程序集 **CourseOrderServiceContract**, 而使用 WCF 类, 需要引用程序集 **System.ServiceModel**。

在 **buttonSubmit** 控件的 Click 事件处理程序中, **ChannelFactory** 类返回一个代理。该代理调用方法 **AddCourseOrder()**, 给队列发送一个消息:

```

private void buttonSubmit_Click(object sender, RoutedEventArgs e)
{
    try
    {
        CourseOrder order = new CourseOrder();
        order.Course = new Course()
        {
            Title = comboCourses.SelectedItem.ToString()
        };
        order.Customer = new Customer()
        {
            Company = textCompany.Text,
            Contact = textContact.Text
        };
        ChannelFactory < ICourseOrderService > factory =
            new ChannelFactory < ICourseOrderService > ("queueEndpoint");
        ICourseOrderService proxy = factory.CreateChannel();
        proxy.AddCourseOrder(order);
        factory.Close();

        MessageBox.Show("Course order submitted", "Course Order",
            MessageBoxButton.OK, MessageBoxImage.Information);
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message, "Course Order Error",

```



```

        MessageBoxButton.OK, MessageBoxImage.Error);
    }
}

```

应用程序配置文件定义了 WCF 连接的客户端部分。这里也使用了 `netMsmqBinding`：

```

< ?xml version="1.0" encoding="utf-8" ? >
< configuration >
  < system.serviceModel >
    < bindings >
      < netMsmqBinding >
        < binding name="nonTransactionalQueueBinding"
          exactlyOnce="false" >
          < security mode="None" / >
        < /binding >
      < /netMsmqBinding >
    < /bindings >
  < /system.serviceModel >
< /configuration >

```

现在启动应用程序，它的工作方式与以前类似。但不再需要使用 `System.Messaging` 命名空间中的类收发消息了。而是使用 TCP 或 HTTP 信道和 WCF 以类似方式编写应用程序。

但是，要创建消息队列，删除消息，仍需要 `MessageQueue` 类。WCF 只是收发消息的一个抽象。

#### 提示：

如果 `System.Messaging` 应用程序与 WCF 应用程序通信，就可以使用 `msmqIntegration-Binding` 替代 `netMsmqBinding`。这个绑定使用用于 COM 和 `System.Messaging` 的消息格式。

## 45.10 消息队列的安装

消息队列可以用 `MessageQueue.Create()` 方法创建。但运行应用程序的用户通常没有创建消息队列所需的管理权限。

消息队列一般可以用安装程序创建。对于安装程序，可以使用 `MessageQueueInstaller` 类。如果安装程序类是应用程序的一部分，那么命令行实用程序 `installutil.exe` (或 Windows 安装软件包) 就会调用安装程序的 `Install()` 方法。

Visual Studio 允许在 Windows 窗体应用程序中使用 `MessageQueueInstaller`。如果把 `MessageQueue` 组件从工具箱拖放到窗体上，该组件的智能标记就可以添加一个安装程序，其菜单项是 `Add Installer`。`MessageQueueInstaller` 对象可以使用属性编辑器配置，以定义事务队列、日志队列、格式化器的类型、基本优先级等。

注意：  
安装程序详见第 16 章。

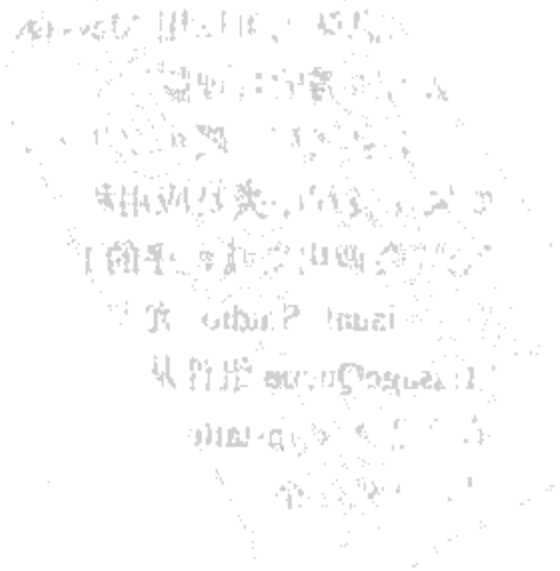
45.11 小结

本章介绍了 Message Queuing 的用法。Message Queuing 是一个重要的技术，不仅提供了异步通信，还提供了断开连接的通信。发送程序和接收程序可以在不同的时间内运行，此时智能客户程序就可以使用 Message Queuing, Message Queuing 还可以把负载分布到不同的时间段上，以充分利用服务器的潜能。

Message Queuing 的最重要的类是 Message 和 MessageQueue。MessageQueue 类可以发送、接收和查看消息，Message 类定义了发送的内容。

WCF 提供了消息队列的一个抽象。可以使用 WCF 提供的概念，调用代理的方法来发送消息，执行一个服务来接收消息。

下一章介绍目录服务，使用这些层次结构数据存储的方式和场合，以及连接这个服务的不同方式。



# 第46章

## 目录服务

Microsoft 的 Active Directory 是一种目录服务, 提供了用户信息、网络资源和服务等集中式的分层存储器。还可以扩展这个目录服务中的信息, 存储企业感兴趣的定制数据。例如, Microsoft 的 Exchange Server 和 Microsoft Dynamics 使用 Active Directory 来存储公共文件夹和其他项目。

在 Active Directory 发布之前, Exchange Server 使用它自己的私有存储器来存储对象。系统管理员必须为一个人配置两个用户 ID: Windows NT 域中的用户账户(以便登录), 和 Exchange Directory 中的用户账户。这是必需的, 因为需要用户的其他信息(例如电子邮件地址, 电话号码等), NT 域的用户信息不能扩展, 以添加需要的信息。目前系统管理员只需要在 Active Directory 上为一个人配置一个用户账户; 用户对象的信息可以扩展, 使之满足 Exchange Server 的要求。我们也可以扩展这些信息。例如, 可以在 Active Directory 上用一个技巧列表扩展用户信息, 然后就可以搜索需要的 C#技巧, 来跟踪 C#开发人员了。

本章介绍如何在 .NET Framework 中使用 System.DirectoryServices、System.DirectoryServices.AccountManagement 和 System.DirectoryServices.Protocols 命名空间中的类访问和处理目录服务中的数据。

### 注意:

本章使用带有 Active Directory 配置的 Windows Server 2008, 也可以使用 Windows 2003 Server 和其他目录服务。

本章主要内容如下:

- Active Directory 的体系结构, 包括特性和基本概念。
- 可用于管理 Active Directory 的一些工具, 及其在编程方面的优势。
- 如何读取和修改 Active Directory 中的数据。
- 搜索 Active Directory 中的对象。
- 账户管理
- 访问 DSML Web 服务, 搜索对象。

在讨论了体系结构和如何对 Active Directory 编程之后, 就要创建一个 Windows 应用程序, 指定一些属性和一个过滤器, 以搜索 user 对象。与其他章节一样, 本章的示例代码也可以从 Wrox 网站 [www.wrox.com](http://www.wrox.com) 上下载。

## 46.1 Active Directory 的体系结构

在开始编程前, 必须知道 Active Directory 的工作方式、用途, 以及什么数据可以存储在 Active Directory 中。

### 46.1.1 特性

Active Directory 的特性可以总结为:

- Active Directory 中的数据是以分层的方式组合的。对象可以存储在其他容器对象中。用户并不是放在一个大的用户列表中, 而是组合到组织单元中。组织单元可以包含其他组织单元, 以这种方式可以建立一个树形视图。
- Active Directory 使用多主机复制方式(multimaster replication)。在 Active Directory 域中, 主域控制器(DC)就是主机, 在多主机模型中, 更新可以应用于所有的 DC。与单主机模型相比, 这个模型的伸缩性比较高, 因为可以同时在不同的服务器上进行更新。该模型的缺点是复制起来比较复杂。本章后面会讨论复制问题。
- 复制拓扑(replication topology)非常灵活, 通过 WAN 中的慢速链接来支持复制。数据复制的频率由域管理员配置。
- Active Directory 支持开放标准。LDAP(Lightweight Directory Access Protocol, 轻型目录访问协议)是一个 Internet 标准, 用于访问许多不同的目录服务, 包括 Active Directory。在 LDAP 中, 也定义了一个编程接口 LDAP API。LDAP API 可以使用 C 语言来访问 Active Directory。在 Active Directory 中使用的另一个标准是 Kerberos, 它用于身份验证。Windows Server Kerberos 服务也可用于验证 Unix 客户的身份。
- Active Directory Service Interface(ADSI)定义了访问目录服务的 COM 接口。ADSI 可以访问 Active Directory 的所有功能。System.DirectoryServices 命名空间中的类封装了 ADSI COM 对象, 可以在 .NET 应用程序中访问目录服务。
- Directory Service Markup Language (DSML)是另一个访问目录服务的标准, 它独立于平台, 得到 OASIS 组的支持。
- 使用 Active Directory 可以得到非常高的安全性。每个存储在 Active Directory 中的对象都可以有一个相关的访问控制列表, 以确定谁可以对对象进行什么处理。

目录中的对象都是类型安全的, 这说明, 对象的类型是定义好的, 没有给对象添加未指定的属性。在模式中, 定义了对象类型和对象的部分(属性)。属性可以是必选的, 也可以是可选的。

### 46.1.2 Active Directory 的概念

在编程之前, 必须了解 Active Directory 的一些基本术语和定义。

#### 1. 对象

可以在 Active Directory 中存储对象。对象可以是一个用户、一台打印机或一个网络共享。

对象包含描述它们的必选属性和可选属性。例如 User 对象的属性包含姓、名、电子邮件地址和电话号码等。

图 46-1 显示了一个包含其他对象的容器对象 Wrox Press, 两个用户对象、一个联系人对象、一台打印机对象和一个用户组对象。

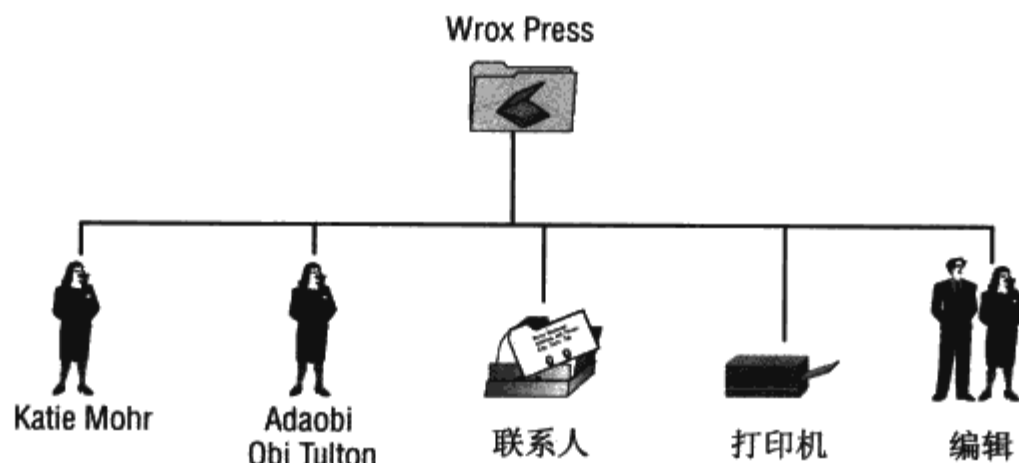


图 46-1

## 2. 模式

每个对象是类的一个实例，这个类是在模式中定义的。模式可以定义类型，模式本身存储在 Active Directory 的对象中。必须区分 classSchema 和 attributeSchema。对象的类型在 classSchema 中定义，其必选属性和可选属性也是在 classSchema 中定义的。AttributeSchema 则定义了属性的外观，以及该属性的语法。

我们可以定义定制类型和属性，把它们添加到模式中。但要注意，不能从 Active Directory 中删除新模式类型。可以把它标记为未激活，这样就不能再创建新对象了，但已有的对象可以是该类型，所以不能删除在模式中定义的类型或属性。

用户组 Administrator 没有足够的权限创建新模式项，此时需要用户组 Enterprise Admins。

## 3. 配置

除了对象和存储为对象的类定义之外，Active Directory 本身的配置也存储在 Active Directory 中。Active Directory 的配置存储了所有站点的信息，如复制间隔等。这些都是由系统管理员设置的。配置本身存储在 Active Directory 中，所以可以像对待 Active Directory 中的其他对象那样访问配置信息。

## 4. Active Directory 域

域是 Windows 网络的安全边界。在 Active Directory 域中，对象以分层的方式进行存储。Active Directory 本身由一个或多个域组成，图 46-2 显示了域中对象的分层顺序，其中域用一个三角形表示。容器对象如 Users、Computers 和 Books 都可以存储其他对象，每个椭圆表示一个对象，对象之间的线表示父-子关系。例如 Books 是 .NET 和 Java 的父对象，Pro C#、Beg C# 和 ASP.NET 是 .NET 对象的子对象。



5. 域控制器

一个域可以有多个域控制器，每个控制器存储域中的所有对象，没有主服务器，所有的DC都是同等的。这是一个多主机模型。对象可以在域的多个服务器之间复制。

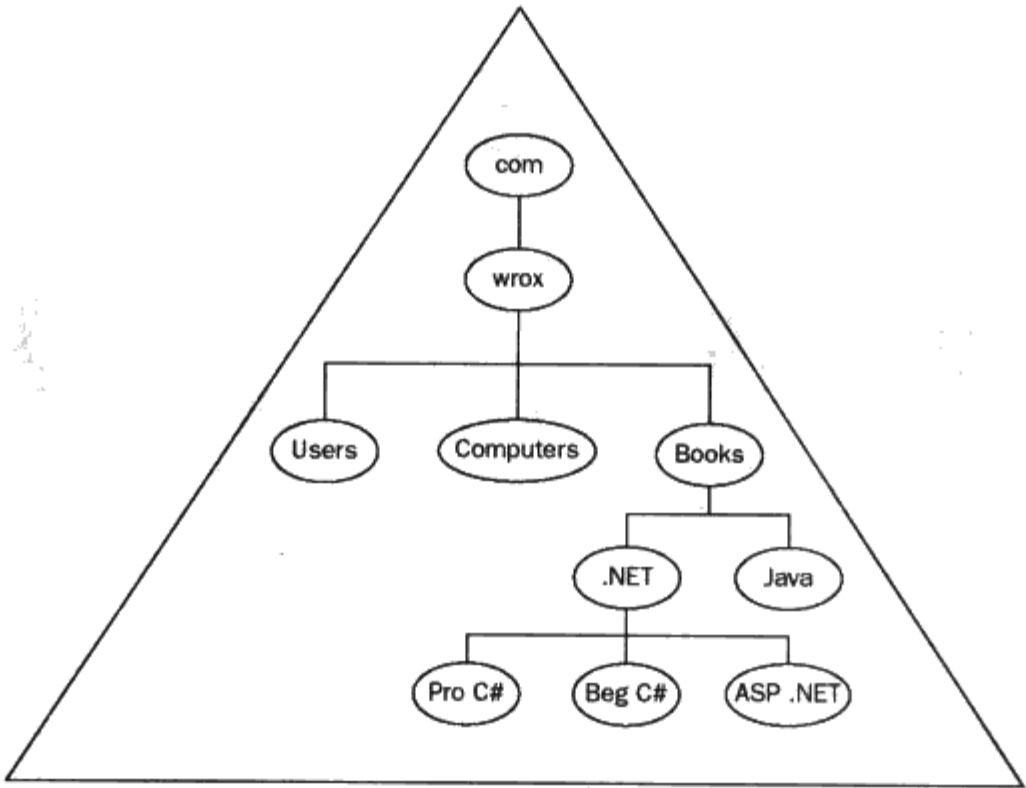


图 46-2

6. 站点

站点是网络中的一个位置，它至少拥有一个 DC。如果企业有多个办事处，用慢速网络连接来连接，就可以在一个域中使用多个站点。由于要考虑备份或可伸缩性，每个站点都有一个或多个 DC 在运行。在一个站点的服务器之间复制数据，其间隔比较短，这是因为网络连接比较快。在站点的服务器之间进行复制的时间间隔可以配置得较大，这取决于网络的速度。当然，这是由域管理员配置的。

7. 域树

可以通过信任关系连接多个域。这些域共享公共模式、公共配置和全局目录(后面将介绍全局目录)。公共模式和公共配置表示该数据可以在域之间复制。域树共享相同的类和属性模式。对象本身不能在域之间复制。

以这种方式连接的域就构成了域树。域树中的域有一个连续的、分层的命名空间。子域的域名是把子域的名称加到父域名上。在这些域之间，建立了使用 Kerberos 协议的信任关系。

例如，有一个根域 wrox.com，它是子域 india.wrox.com 和 uk.wrox.com 的父域。在父域和子域之间建立了信任关系，所以域中的账户可以由其他域验证身份。

8. 森林

使用公共模式、公共配置和全局目录来连接多个域树，但没有使用连续的命名空间，这称为一个森林(forest)。森林是一组域树。如果公司有使用不同域名的子公司，就可以使用森林。

例如,域 wiley.com 应相对独立于域 wrox.com,但应有一个公共的管理,wrox.com 的用户可以访问 wiley.com 域中的资源,反之亦然。使用森林,可以在多个域树之间建立信任关系。

## 9. 全局目录

对象的搜索可以跨越多个域。如果使用一些属性查找某个特定的 user 对象,必须搜索每个域。首先从 wrox.com 开始,接着搜索 uk.wrox.com 和 india.wrox.com,由于使用慢速链接,所以搜索的时间会比较长。

要加快搜索的速度,可以把所有的对象都复制到全局目录 GC 中。GC 将被复制到森林的每个域中。每个域至少有一个服务器包含 GC。出于性能和可伸缩性的考虑,一个域中可以有多个 GC 服务器。使用 GC,可以在一个服务器上搜索所有的对象。

全局目录是所有对象的只读缓存(read only cache)。目录只能用于搜索;必须使用域控制器更新。

并不是对象的所有属性都存储在 GC 中。可以定义属性是否和对象一起保存。属性是否存储在 GC 中,主要取决于该属性在搜索中使用的频率。如果属性在搜索中使用得很频繁,把它放在 GC 中,搜索就会比较快。用户图在 GC 中不是很有用,因为我们从来不会搜索该图。而在存储器中添加电话号码就比较有效。还可以定义属性是否被索引,如果进行索引,对该属性的查询就比较快。

## 10. 复制

程序员不喜欢配置复制,但它会影响存储在 Active Directory 中的数据,所以必须了解它如何工作。Active Directory 使用了多主机服务器结构。域中的每个域控制器都可以进行更新。复制等待时间定义了更新开始之前等待的时间。

- 如果某些属性发生变化,默认情况下,站点中每隔 5 分钟就发布一次改动通知,通知的内容是可配置的,发生改变的 DC 每隔 30 秒就通知另一个服务器。所以第 4 个 DC 可以在 7 分钟后得到改动通知。在默认情况下,在站点之间的改动通知设置为 180 分钟。站点之间和之内的复制可以配置为其他值。
- 如果没有改变,在站点内,每隔 60 分钟就进行一次预定复制。这将确保不漏掉一个改动通知。
- 对于安全性信息,例如账户被锁,会立即发出通知。

进行复制后,就只把改动的内容复制到 DC 中。在每次修改属性后,版本号(USN,更新序列号)和时间戳就会记录下来。如果更新不同服务器上的同一个属性,这可以用于解决冲突。

下面看一个示例。用户 John Doe 的移动电话属性的 USN 号为 47,这个值已经复制到所有域控制器上。一个系统管理员改变了电话号码,在服务器 DC1 上发生了改变后,服务器 DC1 上这个属性的新 USN 现在是 48。而其他域控制器的 USN 仍是 47。如果有人读取该属性,就会得到旧值,直到所有域控制器都进行了复制为止。

下面的情况是很少发生的:另一个管理员改变电话号码属性,选择另一个域控制器,因为这个管理员从服务器 DC2 上接收了一个比较快的响应,在服务器 DC2 上,这个属性的 USN 也改为 48。

在通知的间隔期间,发出通知的原因是属性的 USN 改变了,最后一次进行复制时,USN

的值是 47。使用复制机制可以检测到服务器 DC1 和 DC2 电话号码属性的 USN 都是 48。使用哪个服务器上的属性值并不重要,但必须使用其中一个服务器上的属性值。要解决这个冲突,就要使用改变的时间戳(timestamp)。因为 DC2 上的改变比较迟,所以会复制存储在 DC2 域控制器上的值。

**提示:**

在读取对象时,数据不一定存在。数据是否存在取决于复制等待时间。在更新对象时,另一个用户可以在更新后仍获取旧值,同时还可能进行另一个更新操作。

### 46.1.3 Active Directory 数据的特性

Active Directory 没有替代关系数据库或注册表,那么什么数据可以存储在 Active Directory 中?

- Active Directory 可以存储分层数据,容器可以存储其他容器和对象。容器本身也是对象。
- 数据应主要用于读取。因为在一定的时间间隔中会进行复制,所以不能确定可以读取到最新的数据。在应用程序中,必须注意读取的信息有可能不是最新的。
- 数据应是企业普遍感兴趣的数据。这是因为给模式添加一个新数据类型,会把该数据类型复制到企业的所有服务器上,如果只有一小部分用户对该数据类型感兴趣,企业的域管理员就不会安装新的模式类型。
- 数据应有合理的大小,因为这些数据是要被复制的。如果数据的大小是 100K,而且每星期修改一次数据,把它存储在目录中就不会出问题。但如果每小时修改一次数据,这个数据量就太大了。总是要考虑到数据复制到不同的服务器上、数据要传送到什么地方、复制的时间间隔等。如果数据量比较大,就要链接到 Active Directory 中,把数据存储在另一个地方。

总之,存储在 Active Directory 中的数据应分层组织,且具有合理的大小,这对企业来说非常重要。

### 46.1.4 模式

Active Directory 对象是类型安全的。模式定义了对对象的类型、必选属性和可选属性,属性的语法和约束。在模式中,必须区分类模式和属性模式的对象,类是属性的集合,有了类,就可以支持单一继承。从图 46-3 中可以看出, user 类派生自 organizationalPerson 类,而 organizationalPerson 是 person 的一个子类,它们的基类都是 top。classSchema 定义了一个类,它用 systemMayContain 属性描述了属性。

在图 46-3 中,只列出了几个 systemMayContain 值。使用 ADSI Edit 工具可以看到所有的值。下一节将介绍这个工具。在根类 top 中,每个对象都有公共的名称(cn)、displayName、objectGUID、whenChanged 和 whenCreated 属性。Person 类派生自 top。Person 对象也有 userPassword 和 telephoneNumber。OrganizationalPerson 派生自 Person。除了 Person 的属性外,它还有 manager、department、company 属性,以及 user 登录系统所必需的其他属性。

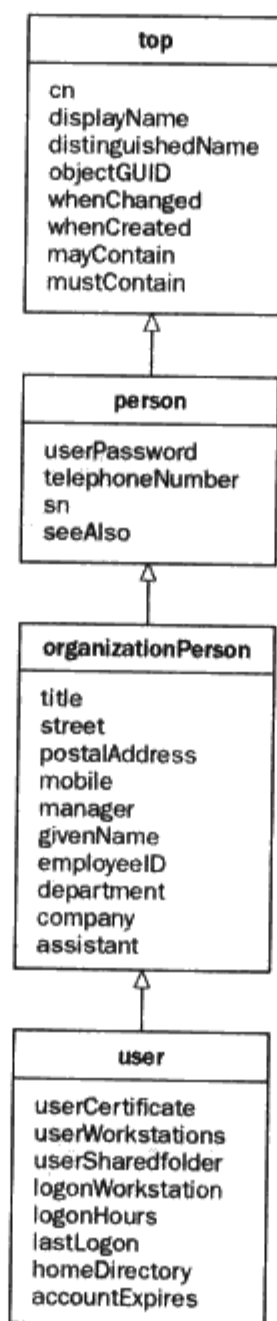


图 46-3

## 46.2 Active Directory 的管理工具

学习一些管理工具有助于理解 Active Directory、其中包含的数据，以及可编程完成的任务。系统管理员可以用许多工具输入新数据，更新数据和配置 Active Directory。

- Active Directory Users and Computers MMC 工具可以更新用户数据、输入新用户。
- Active Directory Sites and Services MMC 工具可以配置域中的站点，在这些站点之间复制数据。
- Active Directory Domains and Trusts MMC 工具可以在树的域之间建立信任关系。
- ADSI Edit 是 Active Directory 的编辑器，可以查看和编辑所有的对象。

提示：

要在 Windows Vista 或 Windows XP 上运行这些工具，需要安装 Windows Server 2003 Admin Pack。ADSI Edit 在 Windows Server 2003 Support 工具中。

下面几节介绍 Active Directory Users and Computers 和 ADSI Edit 工具的功能，因为这些工具对使用 Active Directory 创建应用程序非常重要。

46.2.1 Active Directory Users and Computers 工具

Active Directory Users and Computers 管理单元主要由系统管理员用来管理其用户。选择 Start | Programs | Administrative Tools | Active Directory Users and Computers，就会启动这个程序，得到如 46-4 所示的屏幕图。

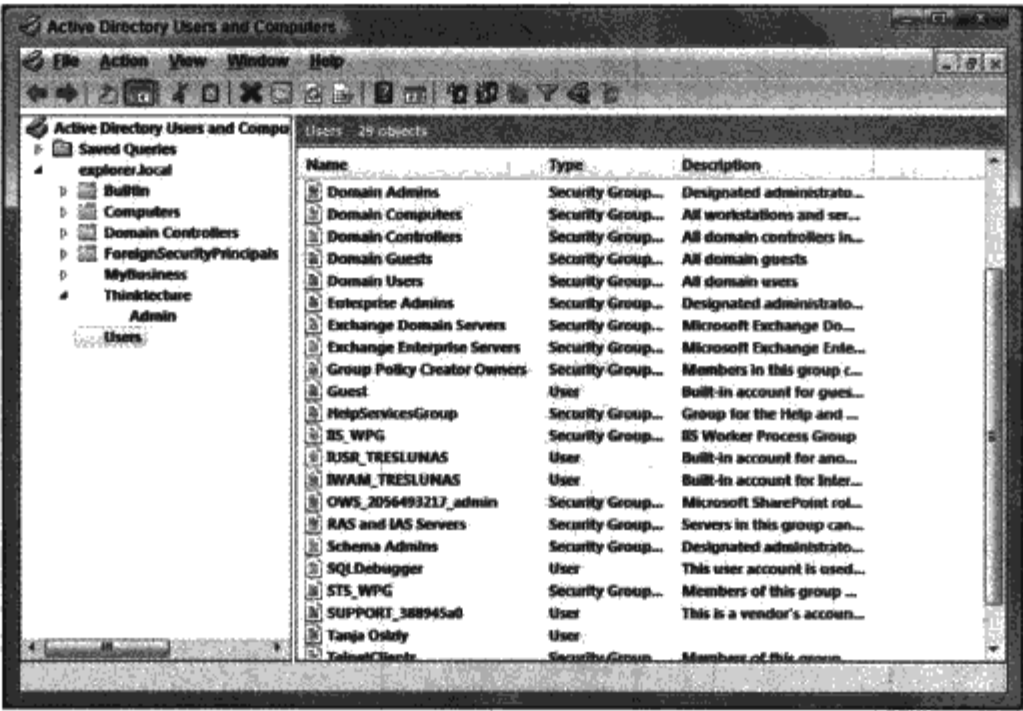


图 46-4

使用这个工具可以添加用户、组、联系人、组织单元、打印机、共享文件夹和计算机，修改已有的项。在屏幕图 46-5 中，可以为 user 对象输入属性：办公室、电话号码、电子邮件地址、Web 页面、公司信息、地址和组等。

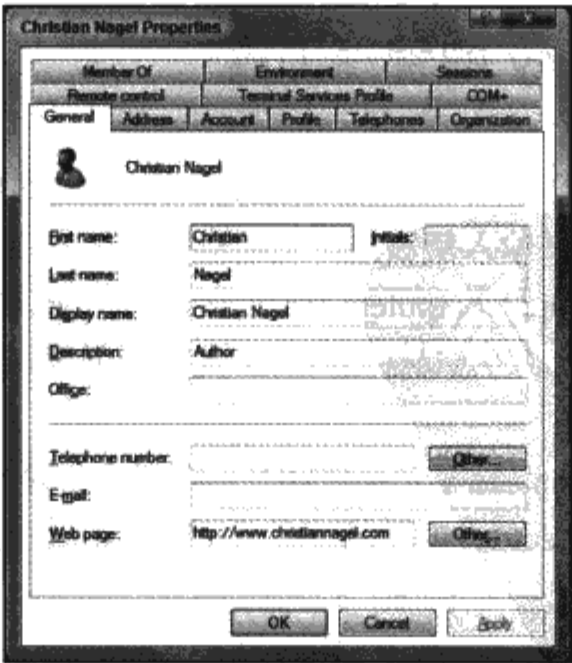


图 46-5

Active Directory Users and Computers 还可以用于有上百万对象的大公司。我们不必在有



1000 个对象的列表中查找，而可以使用一个定制过滤器，只显示某些对象，也可以执行 LDAP 查询，搜索公司中的对象。本章后面将讨论这些问题。

46.2.2 ADSI Edit 工具

ADSI Edit 是 Active Directory 的编辑器。这个工具不能自动安装。在 Windows Server 2003 CD 上有一个目录 Support Tools。安装了支持的工具后，执行程序 adsiedit.msc，就可以访问该工具。

ADSI Edit 提供的控制比 Active Directory Users and Computers 工具更多。使用 ADSI Edit，可以配置所有的一切，也可以查看模式和配置。但这个工具的使用并不是很容易，而且很容易输入错误的数据，如图 46-6 所示。

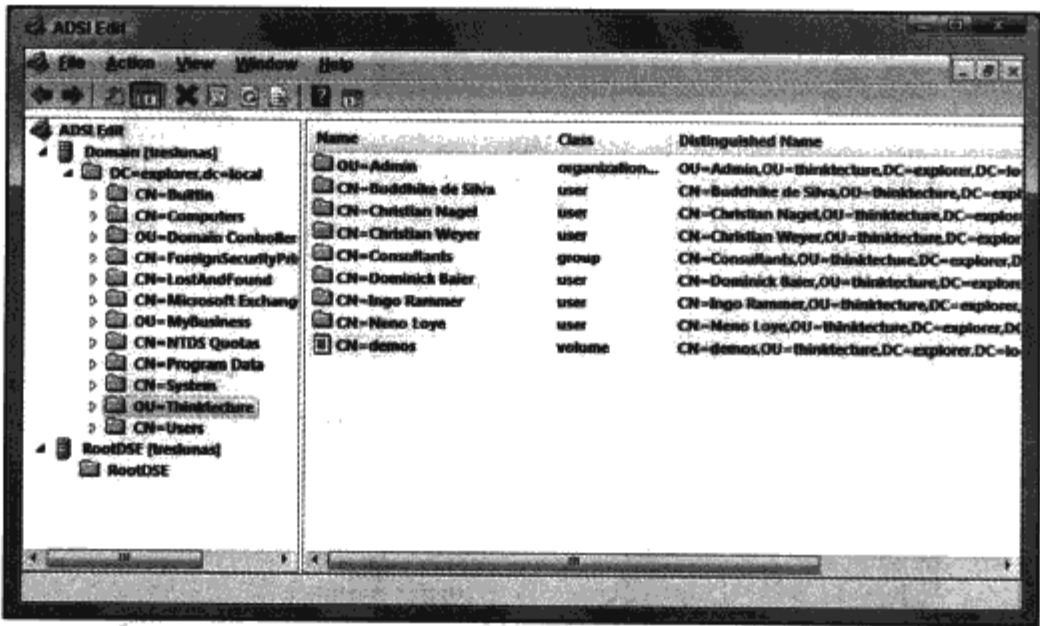


图 46-6

打开对象的 Properties 窗口，可以查看和修改 Active Directory 中对象的每个属性，例如必选属性、可选属性、属性的类型和值等，如图 46-7 所示。

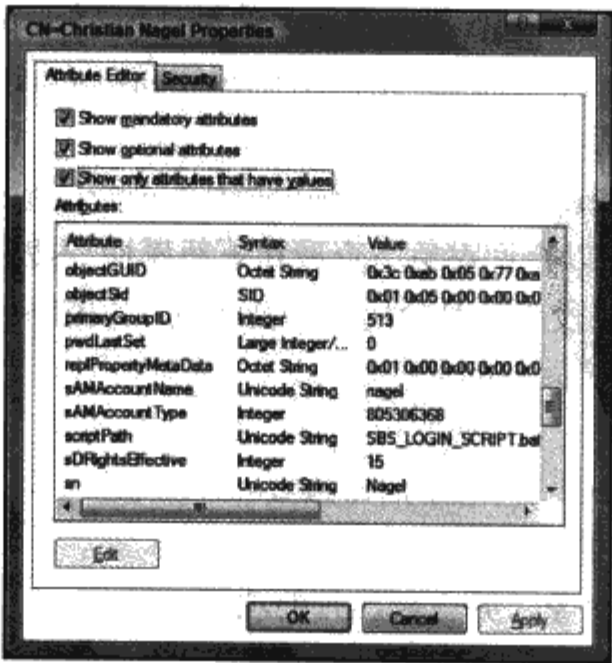


图 46-7

### 46.3 Active Directory 编程

要为 Active Director 开发程序，可以使用 System.DirectoryServices 或 System.DirectoryServices.Protocols 命名空间中的类。在 System.DirectoryServices 命名空间中，有封装了 Active Directory Service Interfaces (ADSI) COM 对象的类，它们可以访问 Active Directory。

ADSI 是一个目录服务的编程接口。ADSI 定义了一些由 ADSI 提供程序实现的 COM 接口。客户可以在相同的编程接口中使用不同的目录服务。System.DirectoryServices 命名空间中的 .NET Framework 类可以使用 ADSI 接口。

在图 46-8 中，有一些实现 COM 接口(如 IADs 和 IUnknown)的 ADSI 提供程序(LDAP、WinNT 和 NDS)。程序集 System.DirectoryServices 使用 ADSI 提供程序。

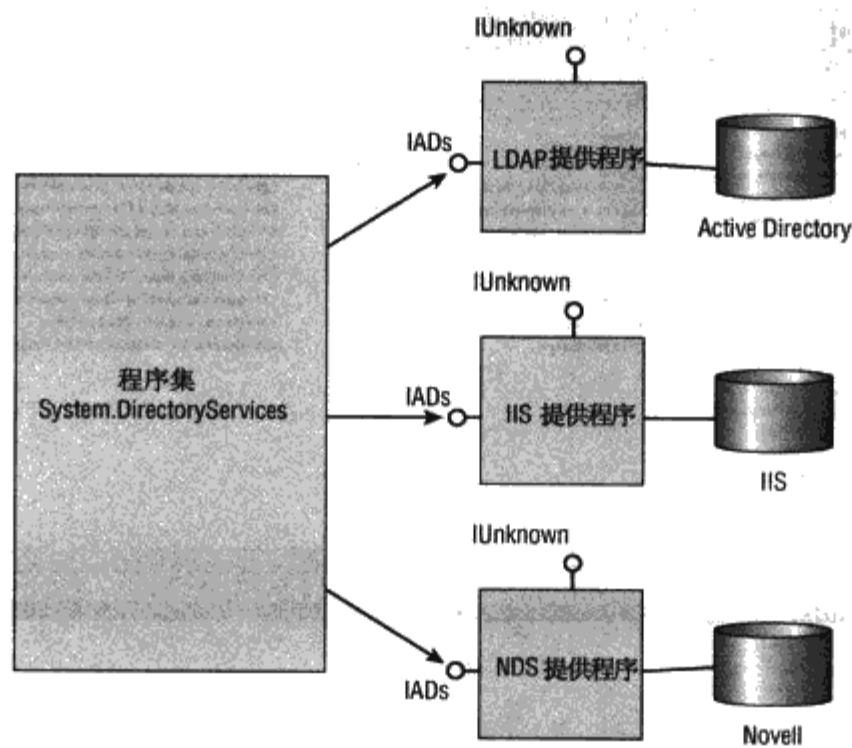


图 46-8

System.DirectoryServices.Protocols 命名空间中的类使用 Directory Services Markup Language (DSML) Services for Windows。OASIS 组(<http://www.oasis-open.org/committees/dsml>) 利用 DSML 定义了标准化的 Web 服务接口。

要使用 System.DirectoryServices 命名空间中的类，必须引用 System.DirectoryServices 程序集。使用这个程序集中的类可以查询对象、查看和更新属性、搜索对象、把对象移动到其他容器对象中。在本节后面的代码段中，将使用一个简单的 C#控制台应用程序，来说明如何使用 System.DirectoryServices 命名空间中的类。

本节将介绍：

- System.DirectoryServices 命名空间中的类。
- 连接 Active Directory 的处理方式：绑定。
- 获取目录项，创建新对象，并更新已有的项目。
- 搜索 Active Directory。

46.3.1 System.DirectoryServices 命名空间中的类

表 46-1 列出了 System.DirectoryServices 命名空间中的主要类。

表 46-1	
类	说 明
DirectoryEntry	这个类是 System.DirectoryServices 命名空间中的主类。这个类的对象表示 Active Directory 库中的一个对象。使用这个类可以绑定对象，查看和更新属性。对象的属性都放在 PropertyCollection 中。PropertyCollection 中的每个元素都有一个 PropertyValueCollection
DirectoryEntries	DirectoryEntries 是 DirectoryEntry 对象的一个集合。DirectoryEntry 对象的 Children 属性返回 DirectoryEntries 集合中的一个对象列表
DirectorySearcher	这个类主要用于用指定的属性搜索对象。要定义该搜索，可以使用 SortOption 类和枚举 SearchScope、SortDirection 和 ReferralChasingOption。搜索的结果是一个 SearchResult 或 SearchResultCollection。也可以得到 ResultProperty Collection 和 ResultPropertyValueCollection 对象

46.3.2 绑定

要获得 Active Directory 中一个对象的值，必须连接 Active Directory 服务。这个连接过程称为绑定，绑定路径如下所示。

```
LDAP:///dc01.thinktecture.com/OU=Development, DC=Thinktecture, DC=Com
```

在绑定过程中，可以指定下述内容：

- 协议(protocol)指定要使用的提供程序。
- 域控制器的服务器名(server name)。
- 服务器过程的端口号(port number)。
- 对象的显名(distinguunshed name)，以标识要访问的对象。
- 如果需要访问 Active Directory 的用户不是当前登录的账户，则提供用户名和密码。
- 如果需要加密，应指定 authentication 类型。

下面详细介绍这些内容。

1. 协议

绑定路径的第一部分指定 ADSI 提供程序。该提供程序是一个 COM 服务器；prog-id 的标识信息在注册表的 HKEY\_CLASSES\_ROOT 下。Windows Vista 附带的提供程序如表 46-2 所示。

2. 服务器名

在绑定路径中，服务器名在协议的后面。如果用户登录到 Active Directory 域上，服务器名就是可选的。如果不提供服务器名，就会发生无服务器绑定操作，此时 Windows Server 2008 会在域中查找与用户绑定过程相关的“最好的”域控制器。如果站点中没有服务器，就使用查

找到的第一个域控制器。  
无服务器的绑定如下所示：

```
LDAP://OU=Sales, DC=Thinktecture, DC=Local
```

表 46-2

提供程序	说 明
LDAP	LDAP 服务器，例如 Exchange 目录和 Windows 2000 Server 或 Windows Server 2003 Active Directory 服务器
GC	GC 用于访问 Active Directory 中的全局目录。它也可以用于快速查询
IIS	使用 IIS 的 ADSI 提供程序，可以在 IIS 目录中创建和管理新网站
NDS	这个 progID 用于和 Novell Directory Services 通信
NWCOMPAT	使用 NWCOMPAT 可以访问旧的 Novell 目录，例如 Novell Netware 3.x

3. 端口号

在服务器名的后面，可以指定服务器过程的端口号，其语法是:xxx。LDAP 服务器的默认端口号是端口 389: LDAP://dc01.sentinel.net:389。Exchange 服务器使用的端口号与 LDAP 服务器一样。如果在同一个系统上安装了 Exchange 服务器，例如用作 Active Directory 的域控制器，就可以配置另一个端口。

4. 显名

在路径中指定的第四部分是显名(distinguished name, DN)。显名是一个唯一的名称，标识要访问的对象。在 Active Directory 中，可以使用基于 X.500 的 LDAP 语法，指定对象的名称。例如有一个显名：

```
CN=Christian Nagel, OU=Consultants, DC= Thinktecture, DC=local
```

这个显名指定域 Thinktecture.local 中域组件(Domain Component, DC)Thinktecture 的组织单元(Organizational Unit, OU)Consultants 的公共名称(Common Name, CN)Christian Nagel。最右边的部分是域的根对象。该名称必须符合对象树中的分层方式。

显名的字符串表示的 LDAP 规范在 RFC 2253(<http://www.ietf.org/rfc/rfc2253.txt>)上。

(1) 相对显名

相对显名(RDN)用于引用容器对象中的对象。使用 RDN 时，不需要指定 OU 和 DC，有一个公共名称就足够了。CN=Christian Nagel 就是组织单元中的一个相对显名。如果已经引用了一个容器对象，要访问其子对象，就可以使用相对显名。

(2) 默认的命名环境

如果在路径中没有指定显名，绑定过程就会使用默认的命名环境(default naming context)。使用 rootDSE 可以读取默认命名环境。LDAP 3.0 把 rootDSE 定义为目录服务器中目录树的根。例如

```
LDAP://rootDSE
```

或:

```
LDAP://servername/rootDSE
```

通过枚举 rootDSE 的所有属性, 将获得没有指定名称时使用的 defaultNamingContext 信息。schemaNamingContext 和 configurationNamingContext 指定了用于访问模式和 Active Directory 库中配置所需要的名称。

通过下面的代码可获得 rootDSE 的所有属性:

```
try
{
    using (DirectoryEntry de = new DirectoryEntry())
    {
        de.Path = "LDAP://treslunas/rootDSE";
        de.Username = @"explorer\christian";
        de.Password = "password";

        PropertyCollection props = de.Properties;
        foreach (string prop in props.PropertyNames)
        {
            PropertyValueCollection values = props[prop];
            foreach (string val in values)
            {
                Console.Write(prop + ": ");
                Console.WriteLine(val);
            }
        }
    }
}
catch (COMException ex)
{
    Console.WriteLine(ex.Message);
}
```

这个程序显示了默认的命名环境(defaultNamingContext DC= explorer、DC=local), 用于访问模式的环境(CN=Schema、CN=Configuration、DC= explorer、DC=local)和配置的命名环境(CN=Configuration、DC= explorer、DC=local), 如下所示。

```
currentTime: 20071012063000.0Z
subschemaSubentry: CN=Aggregate,CN=Schema,CN=Configuration,DC=explorer, DC=local
dsServiceName: CN=NTDS Settings,CN=TRESLUNAS,CN=Servers,
CN=Default-First-Site-Name,CN=Sites,CN=Configuration,DC=explorer,DC=local
namingContexts: DC=explorer,DC=local
namingContexts: CN=Configuration,DC=explorer,DC=local
namingContexts: CN=Schema,CN=Configuration,DC=explorer,DC=local
namingContexts: DC=DomainDnsZones,DC=explorer,DC=local
namingContexts: DC=ForestDnsZones,DC=explorer,DC=local
defaultNamingContext: DC=explorer,DC=local
schemaNamingContext: CN=Schema,CN=Configuration,DC=explorer,DC=local
configurationNamingContext: CN=Configuration,DC=explorer,DC=local
rootDomainNamingContext: DC=explorer,DC=local
supportedControl: 1.2.840.113556.1.4.319
supportedControl: 1.2.840.113556.1.4.801
```

### (3) 对象标识符

每个对象都有一个全局唯一的标识符 GUID。GUID 是一个唯一的 128 位数字, 您可能已



经在 COM 开发中了解了它。使用 GUID 可以绑定一个对象。这样,即使对象移动到另一个容器中,也可以得到该对象。GUID 在创建对象时生成,且总是保持不变。

使用 `DirectoryEntry.NativeGuid` 可以得到 GUID 的字符串表示。这个字符串表示就可以用于绑定对象。

下面的示例显示了一个无服务器绑定的路径名称,它绑定到 GUID 代表的一个特定对象上:

```
LDAP://<GUID=14abbd652aae1a47abc60782dcfc78ea>
```

## 5. 用户名

有时,在访问目录时,必须使用一个非当前进程的用户名(也许这个用户没有访问 Active Directory 所必需的许可),此时必须为绑定过程显式指定用户证书(user credential)。Active Directory 提供了许多方式来设置用户名。

### (1) Downlevel 登录

使用 downlevel 登录,用户名可以用 Windows 2000 以前的域名来指定:

```
domain\username
```

### (2) 显名

也可以用 user 对象的显名来指定用户,例如:

```
CN=Administrator, CN=Users, DC=thinktexture, DC=local
```

### (3) User Principal Name (UPN)

对象的 UPN 用 `userPrincipalName` 属性来定义。系统管理员可以在 Active Directory Users and Computers 工具中 User 属性的 Account 选项卡上,用登录信息来指定 UPN,注意,这不是用户的电子邮件地址。

这些信息也唯一地标识了用户,可以用于登录:

```
Nagel@ thinktexture.local
```

## 6. 身份验证

为了给身份验证进行安全的加密,也可以指定身份验证(authentication)类型。身份验证可以用 `DirectoryEntry` 类的 `AuthenticationType` 属性设置。可以指定其值为 `Authentication Types` 枚举中的一个值。因为枚举是用属性[Flags]标识的,所以可以指定多个值。其可能的值有 `ReadonlyServer` 和 `Secure`。`ReadonlyServer` 对发送的数据进行加密,它指定只需要读取访问;`Secure` 表示安全的身份验证。

## 7. 用 DirectoryEntry 类绑定

`System.DirectoryServices.DirectoryEntry` 类可以用于指定所有的绑定信息。可以使用默认的构造函数,用 `Path`、`Username`、`Password` 和 `AuthenticationType` 属性定义绑定信息,或者把这些信息传递给构造函数:

```
DirectoryEntry de = new DirectoryEntry();
de.Path = "LDAP://platinum/DC= thinktexture, DC=local";
de.Username = "nagel@ thinktexture.local";
```

```
de.Password = "password";

// use the current user credentials
DirectoryEntry de2 = new DirectoryEntry(
    "LDAP://DC= thinktecture, DC=local");
```

即使成功地构造了 `DirectoryEntry` 对象，也并不意味着绑定成功了。在第一次读取属性时进行绑定，可以避免不必要的网络流量。对象是否存在，或者指定的用户证书是否正确，都可以在第一次访问该对象时确定。

### 46.3.3 获取目录项

前面介绍了如何指定 Active Directory 中对象的绑定属性，下面要读取对象的属性。在下面的示例中要读取用户对象的属性。

`DirectoryEntry` 类的一些属性可以提供对象的信息，即 `Name`、`Guid` 和 `SchemaClass Name` 属性。第一次访问 `DirectoryEntry` 对象的属性时，会执行绑定操作，并填充底层 ADSI 对象的缓存。后面将详细讨论这些。其他属性可以从缓存中读取，同一对象的数据不需要通过与服务器的通信来获得。

在本例中，用组织单元 Wrox Press 中的公共名称 Christian Nagel 访问一个用户对象：

```
using (DirectoryEntry de = new DirectoryEntry())
{
    de.Path = "LDAP://treslunas/CN=Christian Nagel, " +
        "OU= Thinktecture, DC=explorer, DC=local";

    Console.WriteLine("Name: " + de.Name);
    Console.WriteLine("GUID: " + de.Guid);
    Console.WriteLine("Type: " + de.SchemaClassName);
    Console.WriteLine();

    //...
}
```

#### 提示：

要在计算机上运行这段代码，必须修改要访问的对象路径，使之包含服务器名。

Active Directory 对象包含许多信息，这些信息取决于对象的类型。属性 `Properties` 将返回一个 `PropertyCollection`。每个属性本身就是一个集合，因为一个属性可以有多个值，例如，`user` 对象可以有多个电话号码。在本例中，用一个内部 `foreach` 循环查看这些值。从 `properties[name]` 返回的集合是一个 `object` 数组。属性值可以是字符串、数字或其他类型。使用 `ToString()` 方法就可以显示这些值：

```
Console.WriteLine("Properties: ");
PropertyCollection properties = de.Properties;
foreach (string name in properties.PropertyNames)
{
    foreach (object o in properties[name])
    {
        Console.WriteLine("{0}: {1}", name, o.ToString());
    }
}
```

在得到的结果中，包含了 user 对象 Christian Nagel 的所有属性，如下所示。otherTelephone 是一个多值属性，它包含许多电话号码。一些属性值只显示 System.\_ComObject 对象的类型，例如，lastLogoff、lastLogon 和 nTSecurityDescriptor。要得到这些属性的值，必须直接使用 System.DirectoryServices 命名空间的类中的 ADSI COM 接口。

```
Name: CN=Christian Nagel
GUID: 7705eb3c-d5aa-40a4-97f9-2649c7693f39
Type: user
Properties:
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: user
cn: Christian Nagel
sn: Nagel
description: Author
givenName: Christian
distinguishedName: CN=Christian Nagel,OU=thinktecture,DC=explorer,DC=local
instanceType: 4
whenCreated: 22.08.2004 13:31:10
whenChanged: 24.05.2005 12:26:05
displayName: Christian Nagel
uSNCreated: System._ComObject
uSNChanged: System._ComObject
company: Thinktecture
extensionName: 5717D53E-DD6D-4d1e-8A1F-C7BE620F65AA:L
WWWHomePage: http://www.christiannagel.com
name: Christian Nagel
objectGUID: System.Byte[]
userAccountControl: 514
badPwdCount: 0
```

### 直接通过名称访问属性

使用 DirectoryEntry.Properties 可以访问所有属性。如果已知属性名，就可以直接访问其值：

```
foreach (string homePage in de.Properties["WWWHomePage"])
    Console.WriteLine("Home page: " + homePage);
```

### 46.3.4 对象集合

对象在 Active Directory 中是分层存储的。容器对象包含子对象。使用 DirectoryEntry 类的 Children 属性可以枚举容器中的子对象。另一方面，使用 Parent 属性可以得到对象的容器。

user 对象没有子对象，所以下面的示例使用一个组织单元。非容器对象用 Children 属性返回一个空集合。下面在域 thinktecture.local 的组织单元 Wrox Press 中获得其所有的 user 对象。Children 属性返回一个 DirectoryEntries 集合，其中包含 DirectoryEntry 对象。迭代所有的 DirectoryEntry 对象，显示子对象的名称。

```
using (DirectoryEntry de = new DirectoryEntry())
{
    de.Path = "LDAP://treslunas/OU=thinktecture, " +
        "DC=explorer, DC=local";

    Console.WriteLine("Children of " + de.Name);
```

```
foreach (DirectoryEntry obj in de.Children)
{
    Console.WriteLine(obj.Name);
}
}
```

运行程序，会显示对象的公共名称：

```
Children of OU=thinktecture
OU=Admin
CN=Buddhike de Silva
CN=Christian Nagel
CN=Christian Weyer
CN=Consultants
CN=demos
CN=Dominick Baier
CN=Ingo Rammer
CN=Neno Loye
```

本例显示了组织单元中的所有对象：users、contacts、printers、shares 和其他对象。如果只需要查看某些对象类型，可以使用 DirectoryEntries 类的 SchemaFilter 属性。SchemaFilter 属性返回一个 SchemaNameCollection。有了这个 SchemaNameCollection，就可以使用 Add() 方法定义要查看的对象类型。在本例中，因为只需要查看 user 对象，所以把 user 添加到这个集合中：

```
using (DirectoryEntry de = new DirectoryEntry())
{
    de.Path = "LDAP:// treslunas /OU=Thibktectrue, " +
        "DC= explorer, DC=local";

    Console.WriteLine("Children of " + de.Name);
    de.Children.SchemaFilter.Add("user");
    foreach (DirectoryEntry obj in de.Children)
    {
        Console.WriteLine(obj.Name);
    }
}
```

结果，只显示组织单元中的 user 对象，如下所示。

```
Children of OU=thinktecture
CN=Buddhike de Silva
CN=Christian Nagel
CN=Christian Weyer
CN=Dominick Baier
CN=Ingo Rammer
CN=Neno Loye
```

### 46.3.5 缓存

为了减少网络流量，ADSI 使用缓存来存储对象属性。如前所述，在创建 DirectoryEntry 对象时是不访问服务器的。只要从目录库中读取第一个属性，所有的属性都会写到缓存中，这样，在读取下一个属性时，就不需要往返于服务器了。

写入对象的改变，只会改变已缓存的对象。设置属性不会产生网络流量。必须使用 DirectoryEntry.CommitChanges() 刷新缓存，把所有已改变的数据传送回服务器。要再次从目录



库中获取新写入的数据，可以使用 `DirectoryEntry.RefreshCache()` 读取属性。当然，如果没有调用 `CommitChanges()` 和 `RefreshCache()` 就修改了一些属性，所有的改变都会丢失，因为我们将再次使用 `RefreshCache()` 读取目录服务中的值。

把属性 `DirectoryEntry.UsePropertyCache` 设置为 `false`，就可以关闭这个属性缓存。但除非在调试，否则最好不要关闭它，因为这会与服务器间产生许多不必要的往返通信。

### 46.3.6 创建新对象

创建新 Active Directory 对象时，例如 `users`、`computers`、`printers` 和 `contacts` 等，可以使用 `DirectoryEntries` 类以编程的方式来完成创建工作。

要给目录添加新对象，首先必须绑定一个容器对象，例如组织单元，可以在其中插入新对象。不包含其他对象的对象是不能使用的。下面使用一个容器对象，其显名为 `CN=Users`，`DC=thinktecture`，`DC=local`：

```
DirectoryEntry de = new DirectoryEntry();
de.Path = "LDAP://treslunas/CN=Users, DC= explorer, DC=local";
```

使用 `DirectoryEntry` 的 `Children` 属性，可以得到一个 `DirectoryEntries` 对象：

```
DirectoryEntries users = de.Children;
```

`DirectoryEntries` 类提供的方法可以添加、删除和查找集合中的对象。下面创建一个新的 `user` 对象。使用 `Add()` 方法时，需要一个对象名和一个类型名。使用 ADSI Edit 很容易获得类型名：

```
DirectoryEntry user = users.Add("CN=John Doe", "user");
```

对象现在有了默认属性值。要指定特定的属性值，可以使用 `Properties` 属性的 `Add()` 方法添加属性。当然，所有的属性都必须存在于 `user` 对象的模式中。如果指定的属性不存在，就得到一个 `COMException`：“指定的目录服务属性或值不存在”：

```
user.Properties["company"].Add("Some Company");
user.Properties["department"].Add("Sales");
user.Properties["employeeID"].Add("4711");
user.Properties["samAccountName"].Add("JDoe");
user.Properties["userPrincipalName"].Add("JDoe@explorer.local");
user.Properties["givenName"].Add("John");
user.Properties["sn"].Add("Doe");
user.Properties["userPassword"].Add("someSecret");
```

最后，为了把数据写入 Active Directory，必须刷新缓存：

```
user.CommitChanges();
```

### 46.3.7 更新目录项

Active Directory 服务中对象的更新和读取一样简单。可以在读取对象后修改它们的值。要删除一个属性的所有值，可以调用 `PropertyValueCollection.Clear()` 方法。使用 `Add()`，可以把新值添加到属性中。`Remove()` 和 `RemoveAt()` 可以从属性集合中删除指定的值。

要修改一个值，可以把这个值设置为指定的值。下面的代码将通过 `PropertyValue Collection`



的一个索引符把移动电话号码设置为一个新值。使用该索引符，只能改变已存在的值。因此，应总是使用 `DirectoryEntry.Properties.Contains()` 来确定属性是否可用。

```
using (DirectoryEntry de = new DirectoryEntry())
{
    de.Path = "LDAP://treslunas /CN=Christian Nagel, " +
        "OU=thinktecture, DC= explorer, DC=local";
    if (de.Properties.Contains("mobile"))
    {
        de.Properties["mobile"][0] = "+43(664)3434343434";
    }
    else
    {
        de.Properties["mobile"].Add("+43(664)3434343434");
    }
    de.CommitChanges();
}
```

在本例的 `else` 部分，如果移动电话号码不存在新属性，就用方法 `PropertyValueCollection.Add()` 添加它。如果对已有的属性使用 `Add()` 方法，得到的结果将取决于属性的类型：单值属性或多值属性。对已有的单值属性使用 `Add()` 方法，会得到一个 `COMException`：违反了一个约束。对已有的多值属性使用 `Add()` 方法，则会成功地把另一个值添加到属性中。

`user` 对象的属性 `mobile` 定义为单值属性，所以不能添加其他移动电话的号码。但是用户可以有多个移动电话的号码。对于多个移动电话的号码，可以使用属性 `otherMobile`，它是一个多值属性，允许设置多个移动电话号码，所以可以调用 `Add()` 多次。注意，对多值属性要检查其唯一性。如果把第二个电话号码添加到同一个 `user` 对象上，也会得到一个 `COMException`：指定的目录服务属性或值已经存在。

#### 提示：

在创建或更新目录对象后，应调用 `DirectoryEntry.CommitChanges()`。否则只能更新缓存中的信息，改变的信息不会发送到目录服务上。

### 46.3.8 访问内部的 ADSI 对象

调用预定义的 ADSI 接口方法常常比搜索对象的属性名容易得多。一些 ADSI 对象还支持不能在 `DirectoryEntry` 类中直接使用的方法。例如 `IADsServiceOperations` 接口有启动和停止 Windows 服务的方法。Windows 服务在第 23 章讨论。

如前所述，`System.DirectoryServices` 命名空间的类使用底层的 ADSI COM 对象。`DirectoryEntry` 类支持直接使用 `Invoke()` 方法调用底层对象的方法。

`Invoke()` 的第一个参数是应在 ADSI 对象中调用的方法名，第二个参数的 `params` 关键字允许把数量可变的其他参数传送给 ADSI 方法：

```
public object Invoke(string methodName, params object[] args);
```

在 ADSI 文档中介绍了可以用 `Invoke()` 方法调用的方法。域中的每个对象都支持 `IADs` 接口的方法。前面创建的 `user` 对象也支持 `IADsUser` 接口的方法。

在下面的代码示例中，使用方法 `IADsUser.SetPassword()` 改变前面创建的 `user` 对象的密码：

```
using (DirectoryEntry de = new DirectoryEntry())
{
    de.Path = "LDAP://treslunas /CN=John Doe, " +
        "CN=Users, DC= explorer, DC=local";

    de.Invoke("SetPassword", "anotherSecret");
    de.CommitChanges();
}
```

如果不使用 Invoke(), 还可以直接使用底层的 ADSI 对象。要使用这些对象, 必须使用 Project | Add Reference 添加对 Active DS 类型库的引用, 如图 46-9 所示。这会创建一个包装器类, 在该类中可以访问 ActiveDS 命名空间中的这些对象。

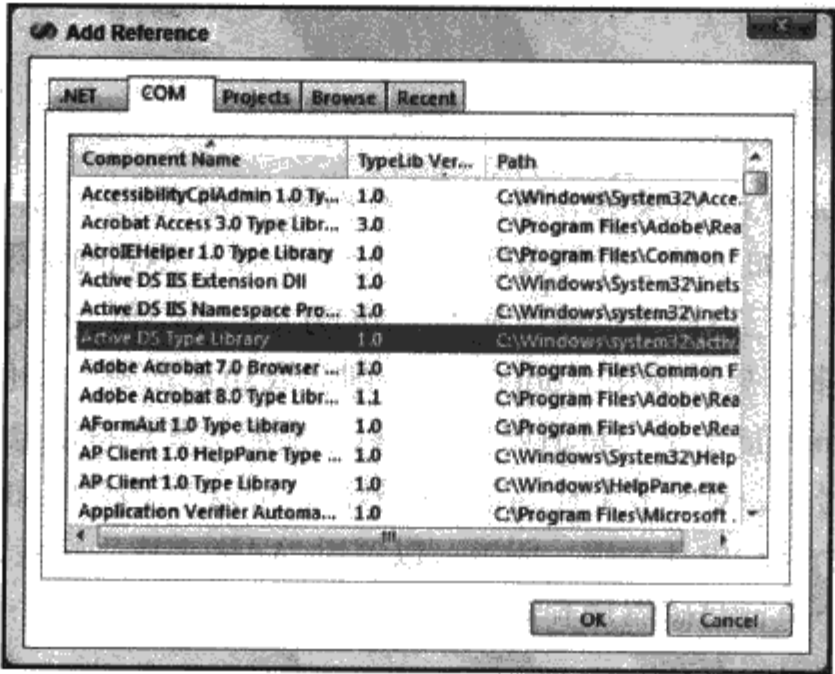


图 46-9

内部对象可以使用 DirectoryEntry 类的 NativeObject 属性来访问。在下面的示例中, 对象 de 是一个 user 对象, 所以可以把它强制转换为 ActiveDs.IADsUser。SetPassword()是在 IADsUser 接口中说明的方法, 所以可以直接调用它, 而不是使用 Invoke()方法来调用。把 IADsUser 的 AccountDisabled 属性设置为 false, 就可以激活账户。与前面的示例一样, 调用 CommitChanges() 和 DirectoryEntry 对象, 把改变的内容写到目录服务中:

```
ActiveDs.IADsUser user = (ActiveDs.IADsUser)de.NativeObject;
user.SetPassword("someSecret");
user.AccountDisabled = false;
de.CommitChanges();
```

46.3.9 在 Active Directory 中搜索

Active Directory 是一个数据库, 它为大多数读取访问进行了优化, 所以一般用来搜索一些值。要在 Active Directory 中搜索, 可以使用 .NET Framework 中的类 DirectorySearcher。

注意:

DirectorySearcher 只能和 LDAP 提供程序一起使用。它不能和 NDS 或 IIS 提供程序一起使用。

在类 `DirectorySearcher` 的构造函数中，可以定义搜索的 4 个重要部分。也可以使用默认构造函数，用属性定义搜索选项。

### 1. SearchRoot

`SearchRoot` 指定搜索应从什么地方开始。`SearchRoot` 的默认值是当前使用的域的根。`SearchRoot` 用 `DirectoryEntry` 对象的 `Path` 指定。

### 2. 过滤器

过滤器定义了希望获得的值。过滤器是一个必须放在括号中的字符串。

表达式中可以使用关系运算符，如 `<=`、`=`、`>=`。`(objectClass=contact)` 会搜索所有类型为 `contact` 的对象，`(lastName>=Nagel)` 会按字母表的顺序搜索 `lastName` 属性等于或大于 `Nagel` 的所有对象。

表达式可以和前缀运算符 `&` 和 `|` 组合使用。例如，`(&(objectClass=user)(description=Auth*))` 搜索类型为 `User`，其属性 `description` 以字符串 `Auth` 开头的对象。因为 `&` 和 `|` 运算符在表达式的开头，所以可以把多个表达式用一个前缀运算符组合在一起。

默认过滤器是 `(objectClass=*)`，所以将选择所有的对象。

注意：

过滤器语法在 RFC 2254 “LDAP 搜索过滤器的字符串表示” 中定义。这个 RFC 在 <http://www.ietf.org/rfc/rfc2254.txt> 中。

### 3. PropertiesToLoad

使用 `PropertiesToLoad`，可以定义需要的所有属性的 `StringCollection`。对象可以有许多属性，其中大多数对于搜索请求都不太重要。我们定义了应加载到缓存中的属性。如果没有指定属性，默认属性就应是对象的 `Path` 和 `Name` 属性。

### 4. SearchScope

`SearchScope` 是一个枚举，定义了搜索应延伸的深度：

- `SearchScope.Base` 只搜索对象中的属性，至多可以得到一个对象。
- `SearchScope.OneLevel` 表示在基对象的子集合中继续搜索。基对象本身是不搜索的。
- `SearchScope.Subtree` 定义了在整个树中搜索。

`SearchScope` 属性的默认值是 `SearchScope.Subtree`。

### 5. 搜索的限制

在目录服务中搜索特定的对象可以在几个域中进行。对搜索要限制对象的个数或搜索时间，可以定义其他几个属性(如表 46-3 所示)。

表 46-3

属 性	说 明
ClientTimeout	客户机等待服务器返回结果的最长时间。如果服务器没有响应，就不返回记录
PageSize	使用 paged search，服务器会返回用 PageSize 定义的许多对象，而不是所有的对象。这会减少客户获得第一个答案的时间和需要的内存，服务器把一个 cookie 送给客户机，客户机再把下一个搜索请求发送回服务器，这样搜索就可以在上一次结束的地方继续进行
ServerPageTimeLimit	对于 paged search，这个值定义了一个搜索时间，在该时间内返回用 PageSize 值定义的许多对象。如果这段时间在 PageSize 值之前到达，就会把此时查找到的对象返回给客户机。默认值为 -1，表示时间为无限
SizeLimit	定义搜索返回的最大对象个数。如果把它设置为比服务器定义的值(1000)还大，就使用服务器定义的限制
ServerTimeLimit	定义服务器搜索对象的最长时间。过了这段时间后，就会把此时查找到的所有对象返回给客户机。默认值是 120 秒，不能把搜索时间设置为较高的值
ReferralChasing	搜索可以在几个域中进行。如果用 SearchRoot 指定的根是一个父域或没有指定根，就会继续在子域中搜索。使用这个属性可以指定是否应在不同的服务器上继续搜索 ReferralChasingOption.None 表示不在不同的服务器上继续搜索 ReferralChasingOption.Subordinate 指定继续在子域中搜索，搜索从 DC=Wrox、DC=COM 开始时，服务器可以返回一个结果集，并引用 DC=France、DC=Wrox、DC=COM。客户机可以继续在子域中搜索 ReferralChasingOption.External 表示服务器可以让客户机搜索不在子域中的另一个独立服务器，这是默认选项 ReferralChasingOption.All 表示返回外部和从属的引用
Tombstone	如果将属性 Tombstone 设置为 true，那么返回所有匹配搜索的被删除对象
VirtualListView	如果搜索结果比较多，就可以使用 VirtualListView 属性定义一个搜索返回的子集。这个子集用 DirectoryVirtualListView 类定义

在搜索示例中，要搜索组织单元 Thinktecture 中 description 属性值为 Author 的所有 user 对象。

首先，绑定组织单元 Thinktecture，在该组织单元中开始搜索。创建一个 DirectorySearcher 对象，在其中设置 SearchRoot。过滤器定义为(&(objectClass=user)(description=Auth\*))，这样就可以搜索所有类型为 user、description 属性以 Auth 开头的对象。搜索的范围应在一个子树中，即在 Thinktecture 的子组织单元中搜索：

```
using (DirectoryEntry de =
    new DirectoryEntry("LDAP://OU=Thinktecture, DC= explorer, DC=local"))
using (DirectorySearcher searcher = new DirectorySearcher())
{
    searcher.SearchRoot = de;
    searcher.Filter = "(&(objectClass=user)(description=Auth*))";
    searcher.SearchScope = SearchScope.Subtree;
```

要在搜索结果中包含的属性有 name、description、givenName 和 wWWHomePage。



```
searcher.PropertiesToLoad.Add("name");
searcher.PropertiesToLoad.Add("description");
searcher.PropertiesToLoad.Add("givenName");
searcher.PropertiesToLoad.Add("wwwHomePage");
```

下面准备开始搜索。还应对结果进行排序。DirectorySearcher 有一个属性 Sort，它可以设置 SortOption。SortOption 构造函数的第一个参数定义了用于排序的属性，第二个参数定义了排序的方向。SortDirection 枚举的值是 Ascending 和 Descending。

要开始搜索，可以使用 FindOne() 方法查找第一个对象，或者使用 FindAll()。FindOne() 返回一个简单的 SearchResult，FindAll() 返回一个 SearchResultCollection。要得到所有的作者对象，就应使用 FindAll()：

```
searcher.Sort = new SortOption("givenName", SortDirection.Ascending);
SearchResultCollection results = searcher.FindAll();
```

使用一个 foreach 循环，可以访问 SearchResultCollection 中的每个 SearchResult。SearchResult 表示搜索缓存中的一个对象。Properties 属性返回一个 ResultPropertyCollection，使用属性名和索引符可以访问该集合中所有的属性：

```
SearchResultCollection results = searcher.FindAll();

foreach (SearchResult result in results)
{
    ResultPropertyCollection props = result.Properties;
    foreach (string propName in props.PropertyNames)
    {
        Console.WriteLine(propName + ": ");
        Console.WriteLine(props[propName][0]);
    }
    Console.WriteLine();
}
```

可以在搜索之后获得完整的对象：SearchResult 有一个方法 GetDirectoryEntry()，它返回查找到的对象的相应 DirectoryEntry。

得到的结果应显示 thinktecture 的列表，它们都有指定的属性，如下所示。

```
name: Christian Nagel
wwwhomepage: http://www.christiannagel.com
description: Author
givenname: Christian
adspath: LDAP://treslunas/CN=Christian Nagel,OU=thinktecture,DC=explorer,DC=local
name: Christian Weyer
description: Author
givenname: Christian
adspath: LDAP://treslunas/CN=Christian Weyer,OU=thinktecture,DC=explorer,DC=local
name: Ingo Rammer
wwwhomepage: http://www.thinktecture.com
description: Author
givenname: Ingo
adspath: LDAP://treslunas/CN=Ingo Rammer,OU=thinktecture,DC=explorer,DC=local
```



## 46.4 搜索用户对象

本节要创建一个 Windows Forms 应用程序 UserSearch。这个应用程序是非常灵活的，因为可以输入特定的域控制器、用户名和密码，以访问 Active Directory 或者使用运行进程的用户。在这个应用程序中，我们将访问 Active Directory 服务的模式，获得 user 对象的属性。该用户可以输入一个过滤器字符串，搜索域中的所有 user 对象。还可以设置应显示的 user 对象的属性。

### 46.4.1 用户界面

用户界面显示了一些已编好序号的步骤，说明如何使用该应用程序(如图 46-10 所示)。

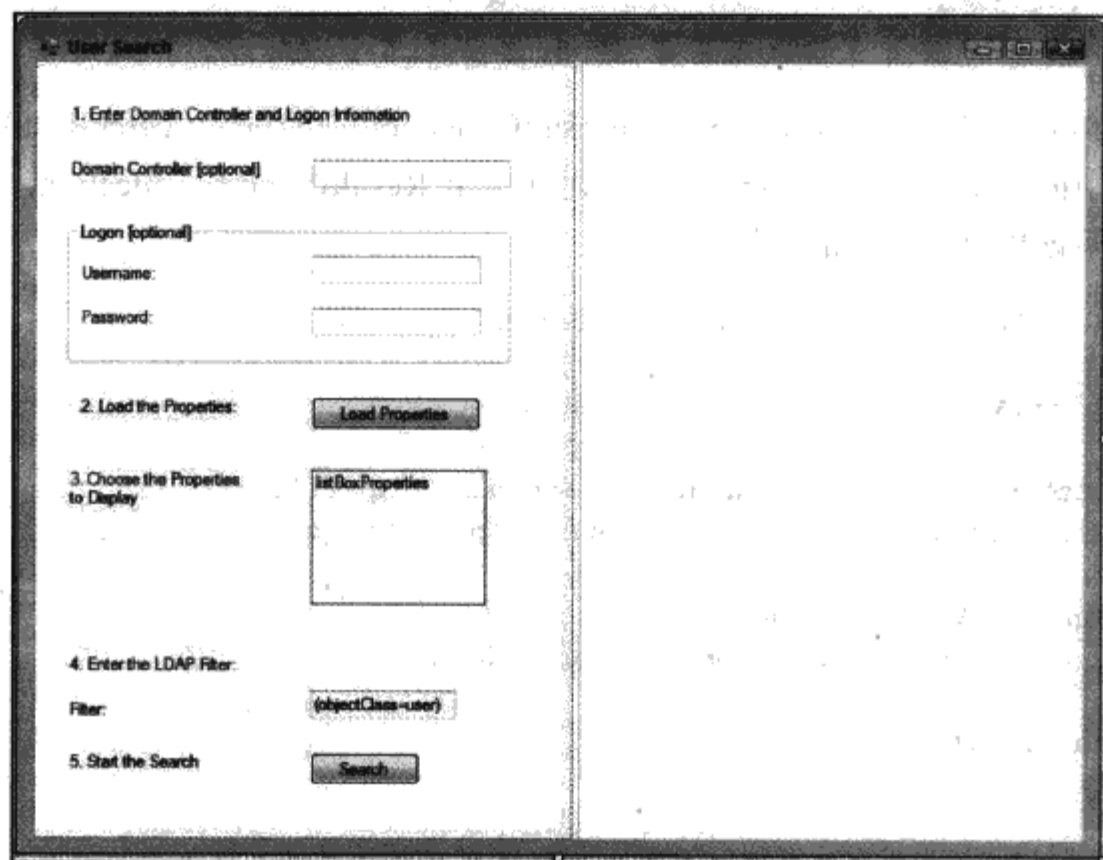


图 46-10

- 在第一步中，输入用户名、密码和域控制器。所有这些信息都是可选的。如果没有输入域控制器，就使用无服务器绑定进行连接。如果没有输入用户名，就使用当前用户的安全环境。
- 使用一个按钮，就可以把 User 对象的所有属性名动态加载到 listBoxProperties 列表框中。
- 在加载了属性名后，就可以选择要显示的属性。列表框的 SelectionMode 设置为 MultiSimple。
- 可以输入限制搜索的过滤器，在该对话框中设置的默认值是搜索所有的 user 对象：(objectClass=user)。
- 现在开始搜索。

### 46.4.2 获取模式命名环境

这个应用程序只有两个处理程序：按钮的第一个处理程序加载属性，第二个处理程序则在域中开始搜索操作。在第一部分，从模式中动态读取 User 类的属性，在用户界面中显示它。

在处理程序 `buttonLoadProperties_Click()` 中，使用 `SetLogonInformation()` 从对话框中读取用户名、密码和主机名，并存储在类的成员中。接着，`SetNamingContext()` 方法设置模式的 LDAP 名称和默认环境的 LDAP 名称。这个模式 LDAP 名称用于 `SetUserProperties()` 调用中，以设置列表框中的属性。

```
private void buttonLoadProperties_Click(object sender, System.EventArgs e)
{
    try
    {
        SetLogonInformation();
        SetNamingContext();

        SetUserProperties(schemaNamingContext);
    }
    catch (Exception ex)
    {
        MessageBox.Show("Check your inputs! " + ex.Message);
    }
}

protected void SetLogonInformation()
{
    username = (textBoxUsername.Text == "" ? null : textBoxUsername.Text);
    password = (textBoxPassword.Text == "" ? null : textBoxPassword.Text);
    hostname = textBoxHostname.Text;
    if (hostname != "") hostname += "/";
}
```

在帮助方法 `SetNamingContext()` 中，使用目录树的根来获得服务器的属性。这里只考虑两个属性的值：`schemaNamingContext` 和 `defaultNamingContext`。

```
protected string SetNamingContext()
{
    using (DirectoryEntry de = new DirectoryEntry())
    {
        string path = "LDAP://" + hostname + "rootDSE";
        de.Username = username;
        de.Password = password;
        de.Path = path;
        schemaNamingContext = de.Properties["schemaNamingContext"][0].ToString();
        defaultNamingContext =
            de.Properties["defaultNamingContext"][0].ToString();
    }
}
```

### 46.4.3 获取 User 类的属性名

使用 LDAP 名称可以访问模式。使用它可以访问目录，并读取属性。我们不仅要介绍 user 类的属性，还将介绍 user 的基类：Organizational-Person、Person 和 Top。在这个程序中，基类的名称是硬编码的。还可以使用 `subClassOf` 属性动态读取基类。

GetSchemaProperties()返回 IEnumerable < string >, 其中包含指定对象类型的所有属性名。这些属性名都添加到列表框中:

```
protected void SetUserProperties(string schemaNamingContext)
{
    var properties =
        from p in
            GetSchemaProperties(schemaNamingContext, "User").Concat(
                GetSchemaProperties(schemaNamingContext,
                    "Organizational-Person")).Concat(
                GetSchemaProperties(schemaNamingContext, "Person")).Concat(
                GetSchemaProperties(schemaNamingContext, "Top"))
        orderby p
        select p;

    listBoxProperties.Items.Clear();
    foreach (string s in properties)
    {
        listBoxProperties.Items.Add(s);
    }
}
```

在 GetSchemaProperties()中, 再次访问 Active Directory 服务。这次不使用 rootDSE, 而使用前面介绍的模式的 LDAP 名称。属性 systemMayContain 包含 objectType 类中的所有属性的一个集合:

```
protected string[] GetSchemaProperties(string schemaNamingContext,
                                       string objectType)
{
    string[] data;
    using (DirectoryEntry de = new DirectoryEntry())
    {
        de.Username = username;
        de.Password = password;

        de.Path = String.Format("LDAP://{0}CN={1},{2}", hostname, objectType,
                                schemaNamingContext);

        PropertyValueCollection values = de.Properties["systemMayContain"];
        data = from s in values.Cast < string > ()
                orderby s
                select s;
    }
    return data;
}
```

这样就完成了应用程序的第二步。Listbox 控件包含 User 对象的所有属性名。

#### 46.4.4 搜索用户对象

搜索按钮的处理程序只调用帮助方法 FillResult();

```
private void OnSearch(object sender, System.EventArgs e)
{
    try
    {
        FillResult();
    }
}
```

```

    }
    catch (Exception ex)
    {
        MessageBox.Show(String.Format("Check your input: {0}", ex.Message));
    }
}

```

在 FillResult() 中, 在整个 Active Directory 域中进行与前面一样的正常搜索。SearchScope 设置为 Subtree, Filter 设置为 TextBox 中的字符串, 应加载到缓存中的属性由用户在列表框中选择的值来设置。DirectorySearcher 的 PropertiesToLoad 属性是 StringCollection 类型, 应加载的属性可以使用 AddRange() 方法来添加, 该方法的参数是一个字符串数组。应加载的属性用 SelectedItems 属性从 listBoxProperties 列表框中读取。在设置了 DirectorySearcher 对象的属性后, 就调用 SearchAll() 方法搜索属性。SearchResultCollection 中的搜索结果用于生成写到文本框 textBoxResults 中的汇总信息。

```

protected void FillResult()
{
    using (DirectoryEntry root = new DirectoryEntry())
    {
        root.Username = username;
        root.Password = password;
        root.Path = String.Format("LDAP://{0}/{1}", hostname,
            defaultNamingContext);

        using (DirectorySearcher searcher = new DirectorySearcher())
        {
            searcher.SearchRoot = root;
            searcher.SearchScope = SearchScope.Subtree;
            searcher.Filter = textBoxFilter.Text;
            searcher.PropertiesToLoad.AddRange(
                listBoxProperties.SelectedItems.Cast < string > ().ToArray());

            SearchResultCollection results = searcher.FindAll();
            StringBuilder summary = new StringBuilder();
            foreach (SearchResult result in results)
            {
                foreach (string propName in
                    result.Properties.PropertyNames)
                {
                    foreach (string s in result.Properties[propName])
                    {
                        summary.AppendFormat(" {0}: {1}\r\n", propName, s);
                    }
                }
                summary.Append("\r\n");
            }
            textBoxResults.Text = summary.ToString();
        }
    }
}

```

启动该应用程序, 列出所有由过滤器选择的有效对象, 如图 46-11 所示。



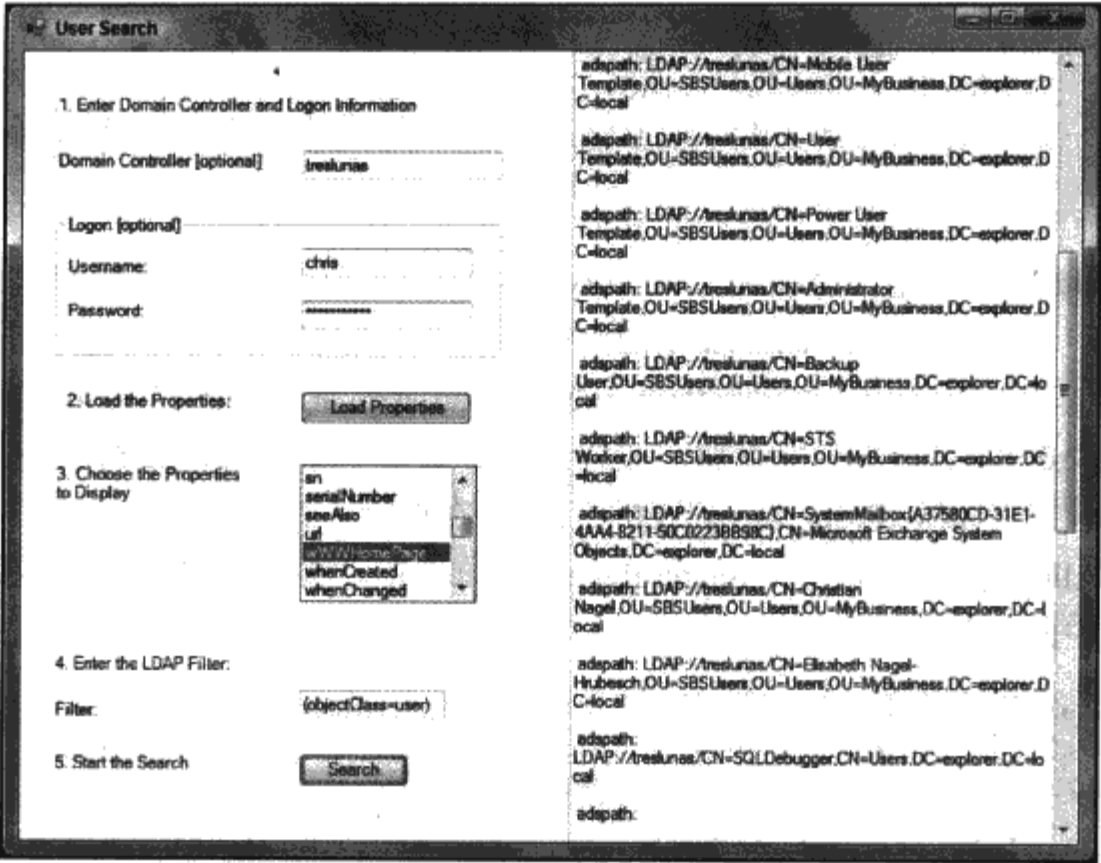


图 46-11

46.5 账户管理

在.NET 3.5 推出之前，很难创建和修改用户和组账户。以前使用的一种方式是利用命名空间 System.DirectoryServices 中的类或强类型化的本地 COM 接口。.NET 3.5 中新增了程序集 System.DirectoryServices.AccountManagement，它提供了 System.DirectoryServices 类的抽象，可以使用特定的方法和属性搜索、修改、创建、更新用户和组。

这些类及其功能如表 46-4 所示。

表 46-4

类	说 明
PrincipalContext	使用 PrincipalContext 可以配置账户管理的环境。这里可以确定是否使用 Active Directory 域、本地系统中的账户或应用程序目录。为此，应把 ContextType 枚举设置为 Domain、Machine 或 ApplicationDirectory。根据环境类型，还可以定义域名，指定用于访问的用户名和密码
Principal	Principal 是所有 Principal 的基类。使用静态方法 FindByIdentity()，可以获得一个 Principal 标识对象。通过 Principal 对象可以访问各种属性，例如名称、描述、显名以及模式中的对象类型。如果除了这个类的方法和属性之外，还需要对 Principal 的更多控制，方法 GetUnderlyingType()返回底层的 DirectoryEntry 对象
AuthenticablePrincipal	AuthenticablePrincipal 派生于 Principal，是所有可以验证身份的 Principal 的基类。有几个查找 Principal 的静态方法，例如按登录或停工时间、按不正确的密码尝试或按密码设置时间来查找。使用实例方法可以修改密码，释放账户



(续表)

类	说 明
UserPrincipal ComputerPrincipal	UserPrincipal 和 ComputerPrincipal 派生于 AuthenticablePrincipal 基类，因此拥有这个基类的所有属性和方法。UserPrincipal 对象映射为用户账户，ComputerPrincipal 映射为计算机账户。UserPrincipal 有许多属性可以获取和设置用户信息，例如 EmployeeId、EmailAddress、GivenName、VoiceTelephoneNumber
GroupPrincipal	组不能进行身份验证，因此 GroupPrincipal 直接派生于 Principal 类。在 GroupPrincipal 中，可以使用 Members 属性和 GetMembers()方法获得组的成员
PrincipalCollection	PrincipalCollection 包含一组 Principal 对象；例如，GroupPrincipal 类的 Members 属性返回一个 PrincipalCollection 对象
PrincipalSearcher	PrincipalSearcher 是 DirectorySearcher 类的一个抽象，专门用于账户管理。使用 PrincipalSearcher 不需要了解 LDAP 查询语法，因为这是自动创建的
PrincipalSearchResult<T>	PrincipalSearcher 和 Principal 类的搜索方法返回 PrincipalSearchResult<T>

下面几节介绍使用 System.DirectoryServices.AccountManagement 命名空间中的类的场合。

46.5.1 显示用户信息

UserPrincipal 类的 Current 静态属性返回一个 UserPrincipal 对象和当前登录用户的信息：

```
using (UserPrincipal user = UserPrincipal.Current)
{
    Console.WriteLine("Context Server: {0}",
        user.Context.ConnectedServer);
    Console.WriteLine(user.Description);
    Console.WriteLine(user.DisplayName);
    Console.WriteLine(user.EmailAddress);
    Console.WriteLine(user.GivenName);
    Console.WriteLine("{0:d}", user.LastLogon);
    Console.WriteLine(user.ScriptPath);
}
```

运行这个应用程序，会显示用户的信息：

```
Context Server: treslunas.explorer.local
Power User
Christian Nagel
Christian.Nagel@thinktecture.com
Christian
2007/10/14
SBS_LOGIN_SCRIPT.bat
```

46.5.2 创建用户

使用 UserPrincipal 类可以创建新用户。首先需要有一个 PrincipalContext 定义创建用户的环境。在 PrincipalContext 中，根据是使用目录服务、机器的本地账户还是应用程序目录，把 ContextType

设置为 Domain、Machine 或 ApplicationDirectory。如果当前用户不能在 Active Directory 中添加账户，还可以使用 PrincipalContext 设置用户和密码，以访问服务器。

接着，传送该 PrincipalContext，设置需要的属性，创建 UserPrincipal 的一个实例。这里设置了 GivenName 和 EmailAddress 属性。最后，必须调用 UserPrincipal 的 Save()方法，把新用户写入存储器：

```
using (PrincipalContext context =
    new PrincipalContext(ContextType.Domain, "explorer"))
using (UserPrincipal user = new UserPrincipal(context, "Tom",
    "P@ssw0rd", true)
{
    GivenName = "Tom",
    EmailAddress = "test@test.com"
})
{
    user.Save();
}
```

### 46.5.3 重置密码

要给已有的用户重置密码，可以使用 UserPrincipal 对象的 SetPassword()方法：

```
using (PrincipalContext context =
    new PrincipalContext(ContextType.Domain, "explorer"))
using (UserPrincipal user = UserPrincipal.FindByIdentity(
    context, IdentityType.Name, "Tom"))
{
    user.SetPassword("Pa$$w0rd");
    user.Save();
}
```

运行这些代码的用户需要有重置密码的权限。要把旧密码改为新密码，可以使用方法 ChangePassword()。

### 46.5.4 创建组

新组的创建方式与新用户的创建相同。这里仅使用 GroupPrincipal 类替代 UserPrincipal 类。与创建新用户一样，也要设置属性，调用 Save()方法：

```
using (PrincipalContext ctx =
    new PrincipalContext(ContextType.Domain, "explorer"))
using (GroupPrincipal group = new GroupPrincipal(ctx)
{
    Description = "Sample group",
    DisplayName = "Wrox Authors",
    Name = "WroxAuthors"
})
{
    group.Save();
}
```

### 46.5.5 在组中添加用户

要在组中添加用户，可以使用 `GroupPrincipal`，把 `UserPrincipal` 添加到组的 `Members` 属性中。要获得已有的用户和组，可以使用静态方法 `FindByIdentity()`：

```
using (PrincipalContext context =
    new PrincipalContext(ContextType.Domain))
using (GroupPrincipal group = GroupPrincipal.FindByIdentity(
    context, IdentityType.Name, "WroxAuthors"))
using (UserPrincipal user = UserPrincipal.FindByIdentity(
    context, IdentityType.Name, "Verena Oslzly"))
{
    group.Members.Add(user);
    group.Save();
}
```

### 46.5.6 查找用户

`UserPrincipal` 对象的静态方法可以根据某些预定义的条件查找用户。这里的示例说明了如何使用方法 `FindPasswordSetTime()` 查找在最近 30 天内未改变密码的用户。这个方法返回一个集合 `PrincipalSearchResult<UserPrincipal>`，迭代它就可以显示用户名、最后一次的登录时间和重置密码的时间：

```
using (PrincipalContext context =
    new PrincipalContext(ContextType.Domain, "explorer"))
using (PrincipalSearchResult < UserPrincipal > users =
    UserPrincipal.FindByPasswordSetTime(context,
    DateTime.Today - TimeSpan.FromDays(30), MatchType.LessThan))
{
    foreach (var user in users)
    {
        Console.WriteLine("{0}, last logon: {1}, " +
            "last password change: {2}", user.Name, user.LastLogon,
            user.LastPasswordSet);
    }
}
```

`UserPrincipal` 类中用于查找用户的其他方法有 `FindByBadPasswordAttempt()`、`FindByExpirationTime()`、`FindByLockoutTime()` 和 `FindByLogonTime()`。

使用 `PrincipalSearcher` 类可以更灵活地查找用户。这个类是 `DirectorySearcher` 类的一个抽象，且在后台使用 `DirectorySearcher` 类。在 `PrincipalSearcher` 类中，可以把任意 `Principal` 对象赋予 `QueryFilter` 属性。在下面的例子中，把带有属性 `Surname` 和 `Enabled` 的 `UserPrincipal` 对象赋予 `QueryFilter`。这样，就用 `PrincipalSearchResult` 集合返回了所有姓氏为 Nag、激活的用户对象。`PrincipalSearcher` 类创建了一个 LDAP 查询字符串，进行搜索。

```
PrincipalContext context = new PrincipalContext(ContextType.Domain);

UserPrincipal userFilter = new UserPrincipal(context);
userFilter.Surname = "Nag*";
userFilter.Enabled = true;

using (PrincipalSearcher searcher = new PrincipalSearcher())
```

```
{
    searcher.QueryFilter = userFilter;
    PrincipalSearchResult < Principal > searchResult =
    searcher.FindAll();
    foreach (var user in searchResult)
    {
        Console.WriteLine(user.Name);
    }
}
```

46.6 DSML

在命名空间 System.DirectoryServices.Protocols 中，可以通过 DSML(目录服务标记语言)访问 Active Directory。DSML 是由 OASIS 组(<http://www.oasis-open.org>)定义的一个标准，它允许通过 Web 服务访问目录服务。

要通过 DSML 访问 Active Directory，必须安装 Windows Server 2003 R2 或 DSML Services for Windows。DSML Services for Windows 可以从 Microsoft 网站上下载。

<http://www.microsoft.com/windowsserver2003/downloads/featurepacks/default.aspx>.

图 46-12 显示了 DSML 的一个配置。提供 DSML 服务的系统通过 LDAP 访问 Active Directory。在客户系统上，使用 System.DirectoryServices.Protocols 命名空间中的 DSML 类向 DSML 服务发出 SOAP 请求。

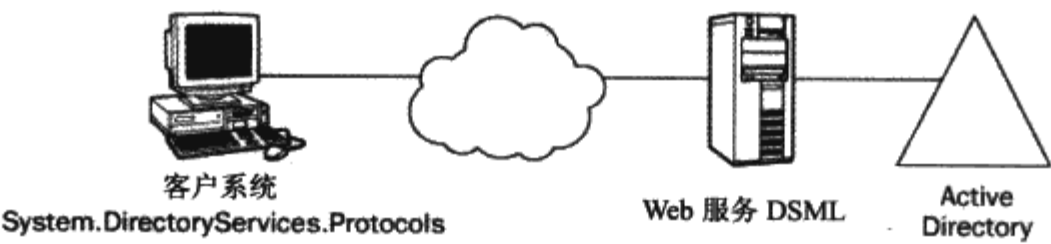


图 46-12

46.6.1 命名空间 System.DirectoryServices.Protocols 中的类

表 46-5 列出了命名空间 System.DirectoryServices.Protocols 中的主要类。

表 46-5

类	说 明
DirectoryConnection	DirectoryConnection 是可以定义目录服务连接的所有连接类的基类。派生于 DirectoryConnection 的类有 LdapConnection(使用 LDAP 协议)、DsmlSoapConnection 和 DsmlSoapHttpConnection。 使用 SendRequest 方法可以把消息发送给目录服务
DirectoryRequest	派生于基类 DirectoryRequest 的类可以把请求发送给目录服务。根据请求的类型，使用 SearchRequest、AddRequest、DeleteRequest 和 ModifyRequest 等类发送请求

(续表)

类	说 明
DirectoryResponse	用 SendRequest 返回的结果的类型派生于基类 DirectoryResponse。这种派生类有 SearchResponse、AddResponse、DeleteResponse 和 ModifyResponse

46.6.2 用 DSML 搜索 Active Directory 对象

本节用一个例子来说明如何搜索目录服务对象。在下面的代码中，首先实例化一个 DsmlSoapHttpConnection 对象，以定义与 DSML 服务的连接。该连接用包含 Uri 对象的 DsmlDirectoryIdentifier 类定义。还可以使用该连接设置用户证书：

```
Uri uri = new Uri("http://dsmlserver/dsml");
DsmlDirectoryIdentifier identifier = new DsmlDirectoryIdentifier(uri);

NetworkCredential credentials = new NetworkCredential();
credentials.UserName = "cnagel";
credentials.Password = "password";
credentials.Domain = "explorer";

DsmlSoapHttpConnection dsmlConnection =
    new DsmlSoapHttpConnection(identifier, credentials);
```

定义了连接后，就可以配置搜索请求了。搜索请求包含开始搜索的目录项、LDAP 搜索过滤器和应从搜索中返回的属性值定义。这里把过滤器设置为(objectClass=user)，以便从搜索中返回所有的用户对象。attributesToReturn 设置为 null，以读取所有带值的属性。SearchScope 是 System.DirectoryServices.Protocols 命名空间中的一个枚举，类似于 System.DirectoryServices 命名空间中的 SearchScope 枚举，它定义了搜索的深度。这里 SearchScope 设置为 Full，搜索完整的 Active Directory 树。

搜索过滤器可以用 LDAP 字符串定义，或使用包含在 XmlDocument 类中的一个 XML 文档定义：

```
string distinguishedName = null;
string ldapFilter = "(objectClass=user)";
string[] attributesToReturn = null;// return all attributes

SearchRequest searchRequest = new SearchRequest(distinguishedName,
    ldapFilter, SearchScope.Full, attributesToReturn);
```

用 SearchRequest 对象定义了搜索后，就调用 SendRequest 方法，把该搜索发送给 Web 服务。SendRequest 是 DsmlSoapHttpConnection 类的一个方法，它返回一个 SearchResponse 对象，从该对象中可以读取返回的对象。

如果不调用同步的 SendRequest 方法，DsmlSoapHttpConnection 类还提供了异步方法 BeginSendRequest 和 EndSendRequest。这些异步方法遵循异步.NET 模式。

提示：  
异步模式可参见第 19 章。



```
SearchResponse searchResponse =
    (SearchResponse)dsmlConnection.SendRequest(searchRequest);
```

返回的 Active Directory 对象可以在 SearchResponse 中读取。SearchResponse.Entries 包含用 SearchResultEntry 类型封装的所有项的集合。SearchResultEntry 类的 Attributes 属性包含所有的特性。每个属性都可以使用 DirectoryAttribute 类来读取。

在代码示例中，每个对象的显名都写入控制台。接着，访问组织单元的属性值，把组织单元的名称写入控制台。之后把 DirectoryAttribute 对象的所有值写入控制台。

```
Console.WriteLine("\r\nSearch matched {0} entries:",
    searchResponse.Entries.Count);
foreach (SearchResultEntry entry in searchResponse.Entries)
{
    Console.WriteLine(entry.DistinguishedName);

    // retrieve a specific attribute
    DirectoryAttribute attribute = entry.Attributes["ou"];
    Console.WriteLine("{0} = {1}", attribute.Name, attribute[0]);

    // retrieve all attributes
    foreach (DirectoryAttribute attr in entry.Attributes.Values)
    {
        Console.WriteLine("{0} =", attr.Name);

        // retrieve all values for the attribute
        // the type of the value can be one of string, byte[] or Uri
        foreach (object value in attr)
        {
            Console.WriteLine("{0} =", value);
        }
        Console.WriteLine();
    }
}
```

添加、修改和删除对象的操作与搜索对象类似。根据要执行的操作，使用相应的方法。

## 46.7 小结

本章介绍了 Active Directory 的体系结构，有关域、树和森林的重要概念。利用它，我们可以访问整个企业的信息。在编写访问 Active Directory 服务的应用程序时，必须注意读取的数据可能不是最新的，因为进行复制操作时有一定的等待时间。

使用 System.DirectoryServices 命名空间中的类，可以很容易地访问封装到 ADSI 提供程序中的 Active Directory 服务。DirectoryEntry 类可以直接读写数据库中的对象。

使用 DirectorySearcher 类可以进行复杂的搜索，定义过滤器、超时、加载的属性和范围等。使用全局目录，可以加快对整个企业中的对象的搜索，因为它在森林中存储了所有对象的只读版本。

DSML 是另一个允许通过 Web 服务接口访问 Active Directory 的 API。

System.DirectoryServices.AccountManagement 中的类提供了一个抽象层，很容易创建和修改用户、组和计算机账户。

下一章通过对等通信提供了联网的另一个视图。

## 对 等 网 络

对等网络常常称为 P2P，是近年来出现的最有用、也最受误解的技术。人们提到 P2P 时，通常会想到：共享音乐文件，这常常是非法的。这是因为文件共享应用程序如 BitTorrent，以令人吃惊的速度迅速流行开来，而这些应用程序使用 P2P 技术工作。

尽管 P2P 在文件共享应用程序中使用，但这并不是说其他应用程序就不使用这个技术。事实上，如本章所述，P2P 可以用于大量的应用程序，目前在我们生活的这个互联世界中变得越来越重要。本章的第一部分将概述 P2P 这个技术。

微软公司并没有忘记 P2P 的存在，而是开发了它自己的工具和技术，来使用 P2P。Microsoft Windows Peer-to-Peer Networking 平台可以用作 P2P 应用程序的通信框架。这个平台包含重要的组件 Peer Name Resolution Protocol(PNRP)和 People Near Me(PNM)。另外，.NET Framework 3.5 还包含一个新的命名空间 System.Net.PeerToPeer 和几个新类型及特性，它们可用于建立 P2P 应用程序。

本章的内容如下：

- P2P 概述
- Microsoft Windows Peer-to-Peer Networking 平台，包括 PNRP 和 PNM
- 如何用 .NET Framework 建立 P2P 应用程序
- 用 .NET Framework 建立的 P2P 应用程序示例

### 47.1 P2P 概述

对等网络是网络通信的另一种方式。为了理解 P2P 与网络通信的“标准”方法之间的区别，需要先回过头来看看客户机-服务器通信。目前，客户机-服务器通信在联网的应用程序中非常普遍。

#### 47.1.1 客户机-服务器体系结构

传统上，我们使用客户机-服务器体系结构，通过网络(包括内联网)与应用程序交互操作。Web 站点就是这方面的一个例子。在 Web 站点上，通过 Internet 给 Web 服务器发送一个请求，Web 服务器会返回我们需要的信息。如果要下载文件，就直接从 Web 服务器上下载。

同样，包含局域网或广域网连接的桌面应用程序一般连接到一个服务器上，例如数据库服务器或包含其他服务的服务器。

客户机-服务器体系结构的这个简单形式如图 47-1 所示。

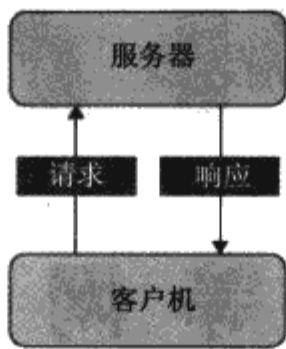


图 47-1

客户机-服务器体系结构本身并没有什么错误，而且在许多情况下，这就是我们需要的，但是它有一个安全性问题。图 47-2 显示了客户机-服务器体系结构如何通过其他客户机来扩展。

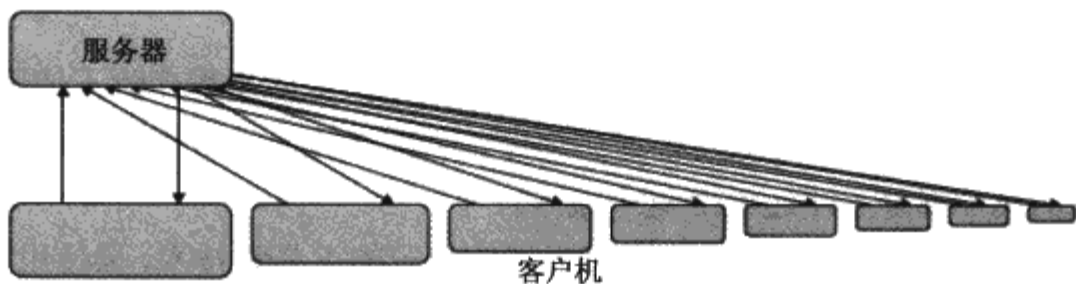


图 47-2

把每个添加了负载的客户机放在服务器上，服务器就必须与每个客户机通信。再看看 Web 站点的例子，这就是 Web 站点崩溃的方式。有太多的请求，服务器已无法响应了。

当然，要缓解这种状况，可以采用一些扩展选项。可以增加服务器的能力和资源，也可以添加更多的服务器，来扩展服务器。这种扩展当然受到可用技术和硬件成本的限制。扩展是比较灵活的，但需要一个额外的基础体系层，来确保客户机与各个服务器的通信，并独立于与它们通信的服务器来维护会话状态。这有许多解决方法，例如 Web 场或服务器场。

47.1.2 P2P 体系结构

对等方法完全不同于扩展方法。在 P2P 中，不是试图使服务器及其客户机之间的通信更流畅，而是考虑客户机之间的通信方式。

例如，客户机与之通信的 Web 站点是 wrox.com。在一个假想的情形中，Wrox 声称，本书的新版本要在 Web 站点 wrox.com 上发布，且可以免费下载，但在过了某一天后删除。在本书可以从该 Web 站点上得到之前，假定有许多人都想访问这个 Web 站点，刷新其浏览器，等待下载文件的出现。一旦文件出现在该 Web 站点上，每个人都试图同时下载它，这很可能使 wrox.com 的 Web 服务器在强大的压力下崩溃。

使用 P2P 技术可以避免这个 Web 服务器崩溃。P2P 技术不把文件直接从服务器发送到所有的客户机上，而是把文件仅发送给几个客户机。另外几个客户机可以从已经包含该文件的客户机上下载它，更多的客户机从这些二级客户机上下载该文件，以此类推。实际上，这个过程把文件分解成几个块，再把这些块分解到客户机上，一些客户机直接从服务器上下载文件，而一些客户机从其他客户机上下载文件，所以这个过程会更快完成。这就是文件共享技术如 BitTorrent 的工作方式，如图 47-3 所示。

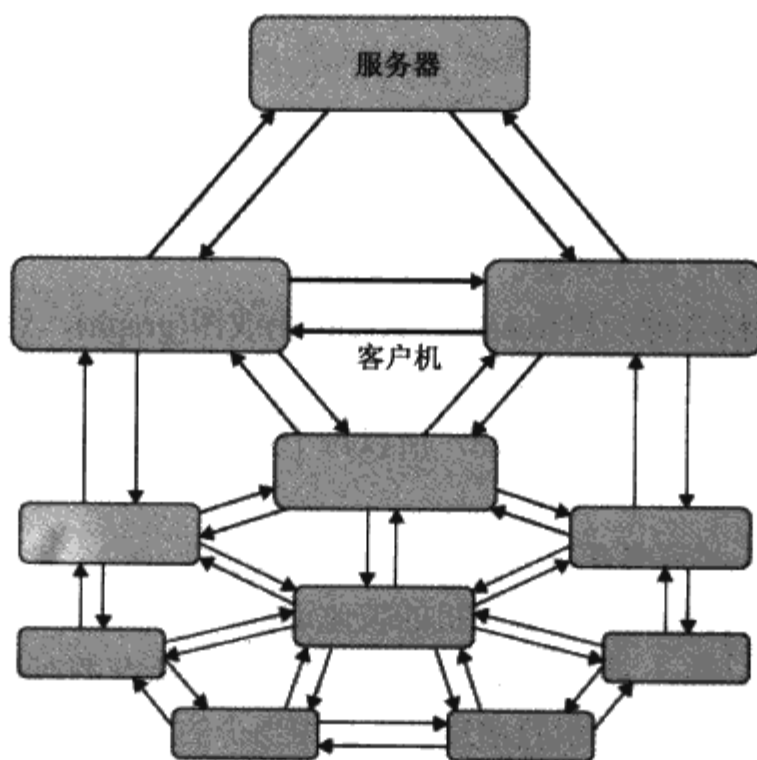


图 47-3

### 47.1.3 P2P 体系结构的挑战

在文件共享体系结构中，仍有一些问题需要解决。首先，客户机如何检测其他客户机的存在？如何定位其他客户机包含的文件块？另外，如何确保客户机之间的通信是按整个大陆来分割、优化的？

每个参与 P2P 网络应用程序的客户机都必须能执行如下操作，才能克服这些问题：

- 必须能发现其他客户机
- 必须能连接其他客户机
- 必须能与其他客户机通信

发现问题有两个明显的解决方案。可以在服务器上保存客户机的列表，这样客户机就可以获得这个列表，联系其他客户机了(称为对等机)；也可以使用一个基础体系(例如 PNRP，详见下一节)，让客户机可以直接找到其他客户机。大多数文件共享系统都使用“服务器上的列表”解决方案，并把服务器称为跟踪器。另外，在文件共享系统中，任何客户机都可以用作服务器，如图 47-3 所示，只有声明客户机有一个文件，并用跟踪器注册它即可。实际上，纯粹的 P2P 网络根本不需要服务器，只需要对等机。

连接问题比较棘手，需要考虑 P2P 应用程序使用的网络的整体结构。如果有一组客户机，它们都可以彼此通信，这些客户机之间的连接拓扑结构就会非常复杂。为了提高性能，常常需要有多组客户机，每组客户机都包含该组中客户机之间的连接，但不包含其他组中的客户机连接。如果在本地建立了这些组，性能就会得到极大的提升，因为客户机可以彼此通信，而不大需要联网计算机之间的连接。

通信问题不太重要，因为通信协议如 TCP/IP 已经建立好了，可以在这种情况下使用。但是，高端技术(例如可以使用 WCF 服务，因此使用 WCF 提供的所有功能)和低端协议(例如把数据同时发送给多个端点的多播协议)所提供的性能提升是有差异的。

发现、连接和通信是所有 P2P 实现方案的中心。本章介绍的实现方案使用 System.Net.PeerToPeer 类型和 PNM 进行发现，使用 PNRP 进行连接。如下面几节所述，这些技

术涵盖了上述的 3 个操作。

#### 47.1.4 P2P 术语

前面几节介绍了对等机的概念,这是客户机在 P2P 网络中的称呼。“客户机”在 P2P 网络中没有意义,因为客户机不需要服务器。

彼此连接的对等机的组合称为网(mesh)、云(cloud)或图(graph),这些术语可以互换。如果满足如下条件,给定的组就是连接好的:

- 每对对等机之间都有一个连接路径,所以每个对等机都可以根据需要连接到其他对等机上。
- 在任意一对对等机之间都有一个相对较小的连接数。
- 删除一个对等机不会妨碍其他对等机的彼此连接。

注意,这并不意味着,每个对等机都必须能连接到其他所有的对等机上。实际上,如果用数学方式分析网络,会发现一个对等机只需连接数量相当少的其他对等机,就可以满足这些条件。

另一个要注意的 P2P 概念是溢出(flooding)。溢出是单项数据通过网络传播到所有对等机上的方式,或者在网络中查询其他节点,以定位某项数据的方式。在未结构化的 P2P 网络中,先连接与自己最近的对等机,这个对等机再连接与它最近的对等机,以此类推,直到连接了网络中的所有对等机为止,这是一个相当随机的过程。也可以创建结构化的 P2P 网络,这样,查询和对等机之间的数据流就有定义好的路径了。

#### 47.1.5 P2P 解决方案

为 P2P 建立了基础体系后,不仅可以开发客户机-服务器应用程序的改进版本,还可以开发全新的应用程序。P2P 特别适合于下述类型的应用程序:

- 内容发布应用程序,包括前面讨论的文件共享应用程序。
- 合作应用程序,例如桌面共享和共享白板应用程序。
- 多用户通信应用程序,允许用户直接通信和交换数据,而不是通过服务器通信。
- 分布式处理应用程序,作为超级计算应用程序的另一种方式,处理海量数据
- Web 2.0 应用程序,在下一代动态 Web 应用程序中合并上述的一部分或全部功能

## 47.2 Microsoft Windows Peer-to-Peer Networking

Microsoft Windows Peer-to-Peer Networking 平台是微软的 P2P 技术实现方案。它是 Windows XP SP2 和 Windows Vista 的一部分,也可用作 Windows XP SP1 的一个插件。它包括两个技术,在创建.NET P2P 应用程序时可以使用这两个技术:

- Peer Name Resolution Protocol(PNRP),用于发布和解析对等机的地址
  - People Near Me(PNM)服务器,用于定位本地的对等机(目前只能在 Vista 中使用)
- 本节学习这两个技术。



### 47.2.1 Peer Name Resolution Protocol(PNRP)

当然,可以使用任意协议实现 P2P 应用程序,但如果在 Microsoft Windows 环境下工作(如果读者在阅读本书,就可能在 Microsoft Windows 环境下工作),考虑 PNRP 至少是有意义的。目前 PNRP 发布了两个版本。PNRP 1 包含在 Windows XP SP2、Windows XP Professional x64 Edition 和 Windows XP SP1 with the Advanced Networking Pack for Windows XP。PNRP 2 随 Windows Vista 一起发布,Windows XP SP2 用户可以通过一个独立的下载包使用它(参见 [support.microsoft.com/kb/920342](http://support.microsoft.com/kb/920342) 的 KB920342)。PNRP 1 和 2 版本不兼容,本章仅介绍版本 2。

PNRP 本身并没有提供创建 P2P 应用程序所需的功能,而只是一个可用于解析对等机地址的底层技术。PNRP 允许客户机注册一个端点(称为对等机名称),该端点自动在云中的对等机中循环。这个对等机名称封装在 PNRP ID 中。发现这个 PNRP ID 的对等机可以使用 PNRP 把它解析为实际的对等机名称,然后就可以直接与它通信。

例如,定义一个表示 WCF 服务端点的对等机名称。可以使用 PNRP 把云中的这个对等机名称注册为 PNRP ID。运行合适客户机应用程序的对等机使用一个发现机制,来标识为服务提供的对等机名称,接着这个对等机会发现这个 PNRP ID,之后对等机就使用 PNRP 定位 WCF 服务的端点,并使用该服务。

#### 提示:

重要的是,PNRP 没有假定对等机名称实际上表示什么。而是在发现对等机名称后,由对等机决定如何使用它们。在解析 PNRP ID 时,对等机从 PNRP 中获得的信息包括 ID 发布者的 IPv6(通常还有 IPv4)地址和一个端口号,有时还有少量其他数据。除非对等机知道对等机名称的含义,否则不可能利用该信息执行有意义的操作。

#### 1. PNRP ID

PNRP ID 是一个 256 位的标识符。低位上的 128 位用于唯一标识对等机,高位上的 128 位表示一个对等机名称。高位上的 128 位是发布对等机的散列公共键和一个至多 149 个字符的字符串的散列组合,该字符串表示对等机名称。散列公共键(称为授权)和这个字符串(称为分类符)统称为 P2P ID。也可以使用 0 代替散列的公共键,此时对等机名称是不安全的(与使用公共键的安全对等机名称相反)。

PNRP ID 的结构如图 47-4 所示。

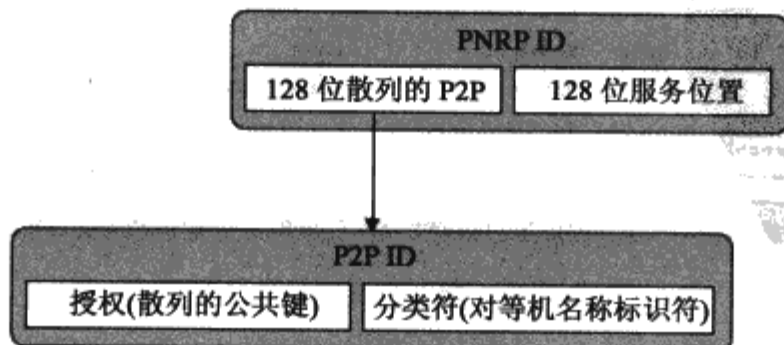


图 47-4

对等机上的 PNRP 服务负责维护一系列 PNRP ID,包括它发布的 PNRP ID 和通过云中其他地方的 PNRP 服务实例获得的 PNRP ID 高速缓存列表。对等机尝试解析 PNRP ID 时,PNRP 服务

会使用端点的一个高速缓存副本来解析发布 PNRP 的对等机，如果对等机的邻居能解析它，就请求它们解析。最终建立与发布对等机之间的连接，PNRP 服务能解析 PNRP ID。

注意，在进行这个操作时，不需要外界干预。我们只需确保对等机在用本地的 PNRP 服务解析了 PNRP ID 后，知道如何处理对等机名称。

对等机可以使用 PNRP 定位匹配某个 P2P ID 的 PNRP ID。使用这个功能可以为不安全的对等机名称实现最基本的发现机制。这是因为如果几个对等机发现了一个使用相同分类符的不安全对等机名称，P2P ID 就是相同的。当然，因为任意对等机都可以使用不安全的对等机名称，所以不能确保所连接的端点就是希望的端点类型，所以这只是在本地网中发现对等机的一种可行方法。

## 2. PNRP 云

在前面的讨论中，学习了 PNRP 如何注册和解析云中的对等机名称。云通过种子服务器维护，该服务器可以是运行 PNRP 服务的任意服务器，但至少要维护一个对等机记录。PNRP 服务可以使用两种类型的云：

- 本地链接：这类云包含连接到本地网络上的计算机。如果 PC 有多个网络适配器，就可以连接到多个本地链接的云上。
- 全局：这类云包含默认连接到 Internet 上的计算机，也可以定义一个私有的全局云。其区别是 Microsoft 为全局的 Internet 云维护种子服务器，而如果定义了私有的全局云，就必须使用自己的种子服务器。此时，必须配置策略设置，确保所有的对等机都连接到种子服务器上。

提示：

在 PNRP 的以前版本中，有第三种云：本地站点。现在不再使用它，所以本章不探讨它。

使用下面的命令可以确定自己连接了什么类型的云：

```
netsh p2p pnrp cloud show list
```

结果如图 47-5 所示。

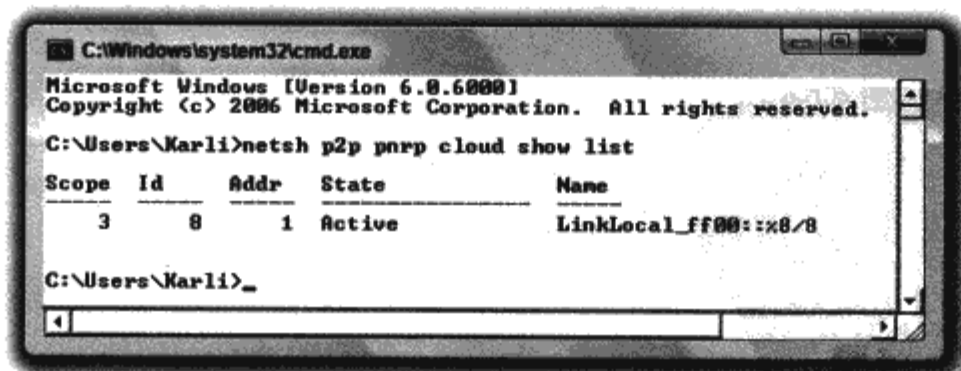


图 47-5

图 47-5 显示，只有一个云可用，它是一个本地链接云。这可以从名称和 Scope 值看出，本地链接云的 Scope 值是 3，而全局云的 Scope 值是 1。为了连接全局云，必须有一个全局的 IPv6 地址。用于获得图 47-5 的计算机没有全局的 IPv6 地址，所以只有一个本地云。

云的状态如下:

- **Active:** 如果云的状态是 active, 就可以使用它发布和解析对等机名称。
- **Alone:** 如果在云中查询的对等机没有连接到其他对等机上, 其状态就是 alone。
- **No Net:** 如果对等机没有连接到网络上, 云的状态就从 active 改为 no net。
- **Synchronizing:** 对等机连接到云上, 云的状态就是 synchronizing。这个状态可以很快变成其他状态, 因为连接不需要很长时间, 所以可能从来看不到云在这个状态下。
- **Virtual:** PNRP 服务仅根据需要, 通过对等机名称注册和解析连接到云上。如果云的连接没有激活的时间超过 15 分钟, 就会进入这个 virtual 状态。

如果读者遇到到网络连接问题, 就应检查防火墙, 以防它禁止通过 UDP 端口 3540 或 1900 的本地网络通信。UDP 端口 3540 由 PNRP 使用, UDP 端口 1900 由 SSDP(Simple Service Discovery Protocol)使用, SSDP 由 PNRP 服务(和 UPnP 设备使用)。

### 47.2.2 People Near Me

如上一节所述, PNRP 用于定位对等机。在考虑 P2P 应用程序的发现/连接/通信过程时, 这显然是一个重要的可用技术, 但 PNRP 本身并不是这些阶段的完整实现方案。People Near Me 服务是发现阶段的一个实现方案, 可以定位在本地区域(即我们连接的本地链接云)的 Window People Near Me 服务中签名的对等机。

我们可能要使用这个服务, 因为它内置在 Vista 中, 在 Windows Meeting Space 应用程序中使用, 该应用程序可用于在对等机中共享应用程序。通过“开始”菜单中如图 47-6 所示的对话框, 可以配置这个服务。

一旦在 PNM 服务中签名为, 就可以用于设置为使用该服务的应用程序。

编写本书时, PNM 只能在 Windows Vista 操作系统上使用。但未来的服务包或其他下载软件包可能在 Windows XP 上使用。

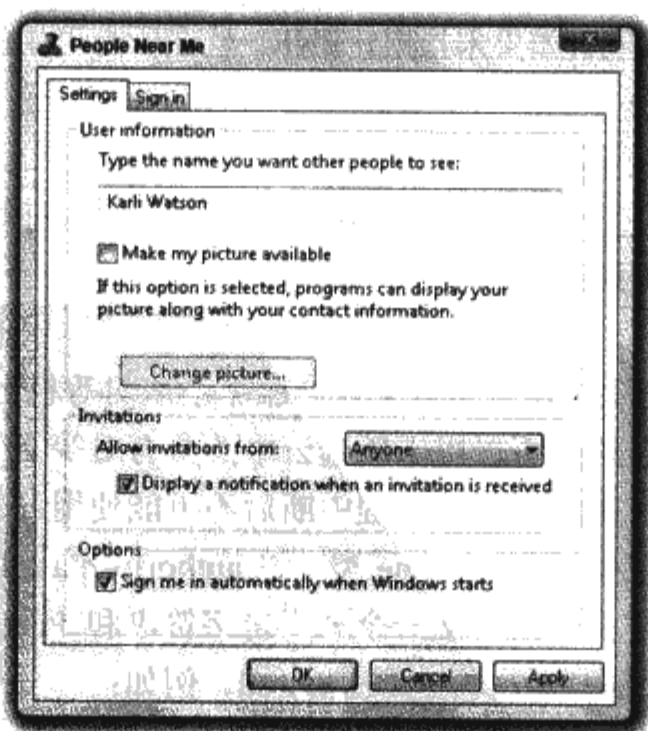


图 47-6

## 47.3 建立 P2P 应用程序

学习了 P2P 网络的概念和 .NET 开发人员可用于实现 P2P 应用程序的技术后, 就该讨论如何建立它们了。在前面的讨论中指出, 使用 PNRP 可以发布和解析对等机名称, 所以这里首先要了解如何使用 .NET 完成这个任务。接着陈述如何把 PNM 作为 P2P 应用程序的框架, 这是很有优势的, 因为我们不必实现自己的发现机制。

为了研究这些主题, 需要学习如下命名空间中的类:

- System.Net.PeerToPeer
- System.Net.PeerToPeer.Collaboration

要使用这些类, 必须引用 System.Net.dll 程序集。

### 47.3.1 System.Net.PeerToPeer

System.Net.PeerToPeer 命名空间中的类封装了 PNRP 的 API, 允许与 PNRP 服务交互操作。使用这些类可以完成两个主要任务:

- 注册对等机名称
- 解析对等机名称

在下面几节中, 除非特别指出, 否则所有的类型都来自 System.Net.PeerToPeer 命名空间。

#### 1. 注册对等机名称

要注册对等机名称, 必须执行如下步骤:

(1) 用特定的分类符创建一个安全或不安全的对等机名称。

(2) 为对等机名称配置一个注册项, 指定如下可选信息:

- TCP 端口号
- 用于注册对等机名称的云(如果未指定, PNRP 就在所有可用的云中注册对等机名称)
- 至多 39 个字符的注释
- 至多 4096 个字节的其它数据
- 是否自动为对等机名称生成端点(默认操作是, 端点应从对等机的 IP 地址和端口号(假设指定了端口号)中生成)
- 端点集合

(3) 使用对等机名称注册项, 通过本地 PNRP 服务注册对等机名称

在第 3 步之后, 对等机名称就可以用于所选云中的所有对等机了。对等机注册在明确停止后不再能使用, 或者在注册对等机名称的过程中断后不再能使用。

要创建对等机名称, 可以使用 PeerName 类。以 authority.classifier 的形式从 P2P ID 的字符串表示中创建这个类的一个实例, 或者从一个分类符字符串和 PeerNameType 中创建。可以使用 PeerNameType.Secured 或 PeerNameType.Unsecured。例如:

```
PeerName pn = new PeerName("Peer classifier", PeerNameType.Secured);
```

不安全的对等机名称使用的 authority 值是 0, 所以下面的代码与上述代码是等价的:

```
PeerName pn = new PeerName("Peer classifier", PeerNameType.Unsecured);
PeerName pn = new PeerName("0.Peer classifier");
```

有了 **PeerName** 实例后,就可以使用它和一个端口号初始化一个 **PeerNameRegistration** 对象:

```
PeerNameRegistration pnr = new PeerNameRegistration(pn, 8080);
```

另外,还可以在使用默认参数创建的 **PeerNameRegistration** 对象上设置 **PeerName** 和 **Port**(可选)属性。也可以把 **Cloud** 实例指定为 **PeerNameRegistration** 构造函数的第三个参数,或者通过 **Cloud** 属性来指定。从云的名称中可以得到一个 **Cloud** 实例,或者使用 **Cloud** 的如下静态成员:

- **Cloud.Global**: 这个静态属性获取全局云的一个引用。根据对等机策略配置,这可以是私有的全局云。
- **Cloud.AllLinkLocal**: 这个静态字段获取一个云,它包含了可用于对等机的所有本地链接云。
- **Cloud.Available**: 这个静态字段获取一个云,它包含了可用于对等机的所有云,其中包含本地链接云和全局云。

创建了 **Cloud** 实例后,可以根据需要设置 **Comment** 和 **Data** 属性。但应注意这些属性的限制。如果试图把 **Comment** 设置为超过 39 个 Unicode 字符的字符串,就会得到一个 **PeerToPeerException**,如果试图把 **Data** 设置为超过 4096 个字节的 **byte[]**,就会得到一个 **ArgumentOutOfRangeException**。还可以使用 **EndPointCollection** 属性添加端点。这个属性是 **System.Net.IPEndPoint** 对象的一个 **System.Net.IPEndPointCollection** 集合。如果使用 **EndPointCollection** 属性,还需要把 **UseAutoEndPointSelection** 属性设置为 **false**,以禁止自动生成端点。

准备注册对等机名称时,可以调用 **PeerNameRegistration.Start()**方法。要从 PNRP 服务中删除对等机名称注册项,可以使用 **PeerNameRegistration.Stop()**方法。

下面的代码注册了一个带注释的安全对等机名称:

```
PeerName pn = new PeerName("Peer classifier", PeerNameType.Unsecured);
PeerNameRegistration pnr = new PeerNameRegistration(pn, 8080);
pnr.Comment = "Get pizza here";
pnr.Start();
```

## 2. 解析对等机名称

要解析对等机名称,必须执行如下步骤:

- (1) 从已知的或通过发现技术得到的 P2P ID 中生成一个对等机名称。
- (2) 使用解析器解析对等机名称,得到一个对等机名称记录集合。可以把解析器限制为只用于某个云,和/或要返回的最大结果数。

- (3) 对于获得的每个对等机名称记录,需要获取对等机名称、端点、注释和额外的数据信息。

这个过程从一个看似对等机名称注册项的 **PeerName** 对象开始。这里的区别是使用通过一个或多个远程对等机注册的对等机名称。从本地链接云中获取活动对等机列表的最简单方式是给每个对等机注册一个分类符相同的、不安全的对等机名称,再在解析阶段使用同一个对等机名称。但是对于全局云,不推荐使用这个方法,因为不安全的对等机名称很容易被盗取。

要解析对等机名称,可以使用 **PeerNameResolver** 类。有了这个类的一个实例后,就可以使



用 `Resolve()` 方法同步解析对等机名称，或者使用 `ResolveAsync()` 方法异步解析对等机名称。

用一个 `PeerName` 参数可以调用 `Resolve()` 方法，也可以给该方法传送要解析的可选 `Cloud` 实例，或者传送要返回的最大对等机数(int)，或者传送这两个参数。这个方法返回一个 `PeerNameRecordCollection` 实例，这是 `PeerNameRecord` 对象的集合。例如，下面的代码解析所有本地链接云中的一个不安全的对等机名称，返回最多 5 个结果：

```
PeerName pn = new PeerName("0.Peer classifier");
PeerNameResolver pnres = new PeerNameResolver();
PeerNameRecordCollection pnrc = pnres.Resolve(pn, Cloud.AllLinkLocal, 5);
```

`ResolveAsync()` 方法使用一个标准的异步方法调用模式。给该方法传送一个唯一的 `userState` 对象，为要查找的对等机监听 `ResolveProgressChanged` 事件，在该方法停止时监听 `ResolveCompleted` 事件。可以用 `ResolveAsyncCancel()` 方法取消未决的异步请求。

`ResolveProgressChanged` 事件的处理程序使用 `ResolveProgressChangedEventArgs` 事件参数，它派生自标准的 `System.ComponentModel.ProgressChangedEventArgs` 类。可以使用在事件处理程序中接收到的事件参数对象的 `PeerNameRecord` 属性获得对查找到的对等机名称记录的引用。

同样，`ResolveCompleted` 事件也需要一个事件处理程序，它使用 `ResolveCompletedEventArgs` 类型的参数，该参数派生于 `AsyncCompletedEventArgs`。这个类型包含一个 `PeerNameRecordCollection` 参数，可用于获得查找到的对等机名称记录的完整列表。

下面的代码是这两个事件处理程序的实现代码：

```
private pnres_ResolveProgressChanged(object sender,
    ResolveProgressChangedEventArgs e)
{
    // Use e.ProgressPercentage (inherited from base event args)
    // Process PeerNameRecord from e.PeerNameRecord
}
private pnres_ResolveCompleted(object sender,
    ResolveCompletedEventArgs e)
{
    // Test for e.IsCancelled and e.Error (inherited from base event args)
    // Process PeerNameRecordCollection from e.PeerNameRecordCollection
}
```

有了一个或多个 `PeerNameRecord` 对象后，就可以处理它们了。这个 `PeerNameRecord` 类提供了 `Comment` 和 `Data` 属性，用于检查对等机名称注册项中设置的注释和数据，其 `PeerName` 属性可获取对等机名称记录的 `PeerName` 对象，最重要的是 `EndPointCollection` 属性。与 `PeerNameRecordRegistration` 一样，这个属性也是 `System.Net.IPEndPoint` 对象的一个 `System.Net.IPEndPointCollection` 集合。这个对象可用于以任意方式连接对等机提供的端点。

### 3. System.Net.PeerToPeer 中的代码访问安全性

`System.Net.PeerToPeer` 命名空间还包含如下两个用于 CAS(参见第 20 章)的类：

- `PnrpPermission`，它继承自 `CodeAccessPermission`
- `PnrpPermissionAttribute`，它继承自 `CodeAccessSecurityAttribute`

这两个类可以用通常的 CAS 方式给 PNRP 访问提供权限功能。

#### 4. 示例应用程序

本章的下载代码包含一个示例 P2P 应用程序(P2PSample),它使用了本节介绍的概念和命名空间。这是一个 WPF 应用程序,为对等机端点使用一个 WCF 服务。

该应用程序用一个应用程序配置文件来配置,在该文件中,可以指定对等机的名称和监听的端口,如下所示:

```
< ?xml version="1.0" encoding="utf-8" ? >
< configuration >
  < appSettings >
    < add key="username" value="Karli" / >
    < add key="port" value="8731" / >
  < /appSettings >
< /configuration >
```

建立了应用程序后,就可以把它复制到本地网络的其他计算机上,运行所有的实例来测试它,或者在一台计算机上运行多个实例,进行测试。如果选择后一个选项,就必须修改每个配置文件(复制本地计算机上 Debug 目录中的内容,依次编辑每个配置文件),以修改每个实例使用的端口。用这两种方式测试该应用程序时,如果还修改了每个实例的用户名,事情会更清楚。

运行对等机应用程序后,就可以使用 Refresh 按钮异步获得对等机列表。定位了一个对等机后,就可以单击对等机的 Message 按钮,发送一个默认信息。

图 47-7 显示了这个应用程序,它在一台机器上运行了 3 个实例。在图 47-7 中,一个对等机给另一个对等机发送了信息,这会生成一个对话框。

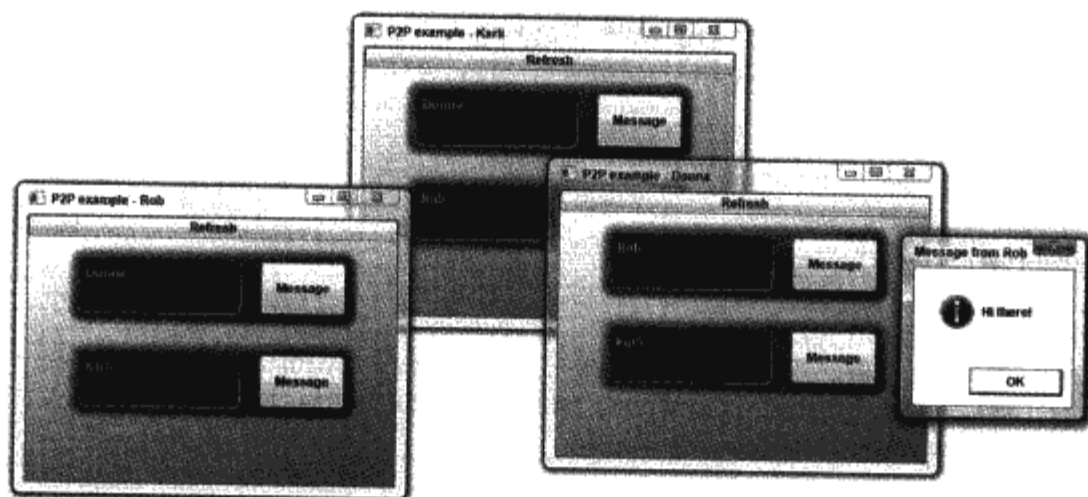


图 47-7

在这个应用程序中,大多数工作都在 Window1 窗口的 Window\_Loaded()事件处理程序中完成。这个方法首先加载配置信息,用用户名设置窗口的标题:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    // Get configuration from app.config
    string port = ConfigurationManager.AppSettings["port"];
    string username = ConfigurationManager.AppSettings["username"];
    string machineName = Environment.MachineName;
    string serviceUrl = null;

    // Set window title
    this.Title = string.Format("P2P example - {0}", username);
}
```

接着使用对等机主机地址和配置的端口确定要把 WCF 服务存储在哪个端点上。该服务使用 `NetTcpBinding` 绑定，所以端点的 URL 使用 `net.tcp` 协议：

```
// Get service url using IPv4 address and port from config file
foreach (IPAddress address in Dns.GetHostAddresses(Dns.GetHostName()))
{
    if (address.AddressFamily ==
        System.Net.Sockets.AddressFamily.InterNetwork)
    {
        serviceUrl = string.Format("net.tcp://{0}:{1}/P2PService",
            address, port);
        break;
    }
}
```

验证了端点的 URL 后，就注册并启动 WCF 服务：

```
// Check for null address
if (serviceUrl == null)
{
    // Display error and shutdown
    MessageBox.Show(this, "Unable to determine WCF endpoint.",
        "Networking Error", MessageBoxButton.OK, MessageBoxImage.Stop);
    Application.Current.Shutdown();
}

// Register and start WCF service.
localService = new P2PService(this, username);
host = new ServiceHost(localService, new Uri(serviceUrl));
NetTcpBinding binding = new NetTcpBinding();
binding.Security.Mode = SecurityMode.None;
host.AddServiceEndpoint(typeof(IP2PService), binding, serviceUrl);
try
{
    host.Open();
}
catch (AddressAlreadyInUseException)
{
    // Display error and shutdown
    MessageBox.Show(this, "Cannot start listening, port in use.",
        "WCF Error", MessageBoxButton.OK, MessageBoxImage.Stop);
    Application.Current.Shutdown();
}
```

该服务类的 `singleton` 实例用于启动主机应用程序和服务之间的简便通信(用于发送和接收信息)。另外要注意，在绑定配置中，为了简单起见，禁用了安全性。

之后，使用 `System.Net.PeerToPeer` 命名空间中的类注册对等机名称：

```
// Create peer name
peerName = new PeerName("P2P Sample", PeerNameType.Unsecured);

// Prepare peer name registration in link local clouds
peerNameRegistration = new PeerNameRegistration(peerName,
    int.Parse(port));
peerNameRegistration.Cloud = Cloud.AllLinkLocal;

// Start registration
peerNameRegistration.Start();
}
```

单击 Refresh 按钮时, RefreshButton\_Click() 事件处理程序使用 PeerNameResolveResolveAsync() 异步获得对等机:

```
private void RefreshButton_Click(object sender, RoutedEventArgs e)
{
    // Create resolver and add event handlers
    PeerNameResolver resolver = new PeerNameResolver();
    resolver.ResolveProgressChanged +=
        new EventHandler< ResolveProgressChangedEventArgs > (
            resolver_ResolveProgressChanged);
    resolver.ResolveCompleted +=
        new EventHandler< ResolveCompletedEventArgs > (
            resolver_ResolveCompleted);

    // Prepare for new peers
    PeerList.Items.Clear();
    RefreshButton.IsEnabled = false;

    // Resolve unsecured peers asynchronously
    resolver.ResolveAsync(new PeerName("0.P2P Sample"), 1);
}
```

剩余的代码负责显示对等机, 与对等机通信, 读者可以自己研究它们。

通过 P2P 云提供 WCF 端点是定位企业中的服务的一种好方法, 也是在对等机之间通信的好方法, 如本例所示。

### 47.3.2 System.Net.PeerToPeer.Collaboration

System.Net.PeerToPeer.Collaboration 命名空间中的类提供了一个框架, 可用于创建使用 PNM 服务和 P2P 协作功能的应用程序。如前所述, 只有使用 Windows Vista, 才能创建这类应用程序。

这个命名空间中的类可以与对等机和应用程序用许多方式交互操作, 包括:

- 签到和退出
- 发现对等机
- 管理各个联系人, 检测对等机的存在

还可以使用这个命名空间中的类邀请其他用户参与某个应用程序, 在用户和应用程序之间交换数据, 但是, 为此, 需要创建支持 PNM 的应用程序, 这超出了本章的范围。

在下面几节中, 除非特别指出, 否则所有的类型都来自 System.Net.PeerToPeer.Collaboration 命名空间。

#### 1. 签到和退出

System.Net.PeerToPeer.Collaboration 命名空间中的最重要的一个类是 PeerCollaboration 类。这是一个静态类, 提供了许多用于各种目的的静态方法, 如本节和下面几节所述。使用其中的两个方法 SignIn() 和 SignOut(), 可以签到和退出 PNM 服务。这两个方法都带一个 PeerScope 类型的参数, 该参数的值如下:

- PeerScope.None: 如果使用这个值, SignIn() 和 SignOut() 就没有任何作用。
- PeerScope.NearMe: 它表示签到和退出本地链接云。

- **PeerScope.Internet**: 它表示签到和退出全局云(连接当前不在本地子网上的联系人时, 需要使用这个值)。
- **PeerScope.All**: 它表示签到和退出所有的云。

如果需要, 调用 **SignIn()** 会打开 **People Near Me** 配置对话框。

签到了对等机后, 就可以把 **PeerCollaboration.LocalPresenceInfo** 属性设置为 **PeerPresenceInfo** 类型的一个值。这会启动标准的 IM 功能, 例如把状态设置为“离开”。**PeerPresenceInfo.DescriptiveText** 属性可以设置为至多 255 个字符的 Unicode 字符串, **PeerPresenceInfo.PresenceStatus** 属性可以设置为 **PeerPresenceStatus** 枚举的值。这个枚举的值如下:

- **PeerPresenceStatus.Away**: 对等机退出了。
- **PeerPresenceStatus.BeRightBack**: 对等机退出了, 但不久会返回。
- **PeerPresenceStatus.Busy**: 对等机在忙。
- **PeerPresenceStatus.Idle**: 对等机未激活。
- **PeerPresenceStatus.Offline**: 对等机离线。
- **PeerPresenceStatus.Online**: 对等机在线, 可以使用。
- **PeerPresenceStatus.OnThePhone**: 对等机在忙着通话。
- **PeerPresenceStatus.OutToLunch**: 对等机退出了, 但在午餐后会返回。

## 2. 发现对等机

如果登录到本地链接云上, 就可以获得附近的对等机列表。为此可以使用 **PeerCollaboration.GetPeersNearMe()** 方法。这个方法返回包含 **PeerNearMe** 对象的 **PeerNearMeCollection** 对象。

使用 **PeerNearMe** 对象的 **Nickname** 属性可以获得对等机的名称, 使用 **IsOnline** 属性可以确定对等机是否在线, 使用 **PeerEndpoints** 属性(用于低级操作)可以确定与对等机相关的端点。如果要确定 **PeerNearMe** 的在线状态, 也需要使用 **PeerEndpoints**。可以给 **GetPresenceInfo()** 方法传送一个端点, 获得一个 **PeerPresenceInfo** 对象, 如上一节所述。

## 3. 管理各种联系, 检测对等机的存在

联系是记住对等机的一种方式。可以添加通过 PNM 服务找到的对等机, 之后, 只要自己在线, 就可以连接该对等机。也可以通过本地链接云或全局云连接联系人(假定与 Internet 建立了 IPv6 连接)。

可以调用 **PeerNearMe.AddToContactManager()** 方法, 在对等机中添加已发现的联系人。在调用这个方法时, 可以选择把显示姓名、昵称和电子邮件地址关联到联系人上。但一般使用 **ContactManager** 类来管理联系人。

无论如何处理联系人, 都要处理 **PeerContact** 对象。这个对象与 **PeerNearMe** 一样, 也继承自 **Peer** 抽象基类。**PeerContact** 的属性和方法比 **PeerNearMe** 多, 例如, **PeerContact** 包括 **DisplayName** 和 **EmailAddress** 属性, 它们进一步描述了 PNM 对等机。这两个类型的另一个区别是 **PeerContact** 与 **System.Net.PeerToPeer.PeerName** 类有更明确的关系。通过 **PeerContact.PeerName** 属性可以从 **PeerContact** 中获得 **PeerName**。之后, 就可以使用前面介绍的技术与 **PeerName** 提供的任意端点通信了。



本地对等机的信息也可以通过 `ContactManager` 类的 `ContactManager.LocalContact` 静态属性来获得。该属性提供了一个 `PeerContact` 属性和本地对等机的详细信息。

使用 `ContactManager.CreateContact()` 或 `CreateContactAsync()` 方法, 可以把 `PeerNearMe` 对象添加到联系人的本地列表中。使用 `GetContact()` 方法可以获得 `PeerName` 对象。使用 `DeleteContact()` 方法可以删除 `PeerNearMe` 或 `PeerName` 对象表示的联系人。

最后, 可以使用一些事件来响应对联系人的修改。例如, 改变了 `ContactManager` 已知的任意联系人的存在状态时, 可以使用 `PresenceChanged` 事件来响应。

#### 4. 示例应用程序

这是本章下载代码中的第二个示例应用程序, 它演示了 `System.Net.PeerToPeer.Collaboration` 命名空间中的类的用法。这个应用程序类似于前一个示例, 但要简单许多。我们需要两台都可以签到 PNM 服务器上的计算机, 才能看到这个应用程序的执行情况, 因为它枚举并显示了本地子网中的 PNM 对等机。

在至少可以找到一个对等机的情况下运行应用程序, 结果如图 47-8 所示。

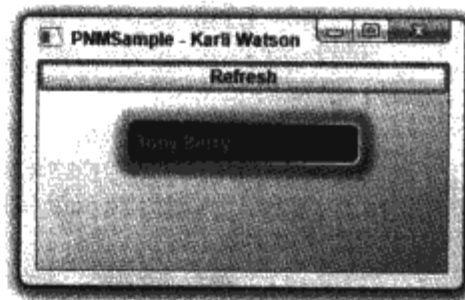


图 47-8

代码的结构与上一个例子相同, 所以如果阅读了前面的代码, 就会很熟悉这个例子的代码。这次在 `Window_Loaded()` 事件处理程序中, 除了签到之外, 并没有太多的工作要做, 因为没有初始化 WCF 服务, 也没有激活对等机名称注册项:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    // Sign in to PNM
    PeerCollaboration.SignIn(PeerScope.NearMe);
}
```

为了使结果更漂亮, 使用 `ContactManager.LocalContact.Nickname` 格式化窗口的标题:

```
// Get local peer name to display
this.Title = string.Format("PNMSample - {0}",
    ContactManager.LocalContact.Nickname);
}
```

在 `Window_Closing()` 中, 本地对等机自动退出 PNM:

```
private void Window_Closing(object sender,
    System.ComponentModel.CancelEventArgs e)
{
    // Sign out of PNM
    PeerCollaboration.SignOut(PeerScope.NearMe);
}
```

大多数工作都是在 `RefreshButton_Click()` 事件处理程序中完成的, 它使用 `PeerCollaboration.GetPeersNearMe()` 方法获得对等机的列表, 并使用项目中定义的 `PeerEntry` 类, 把这些对等机添加到显示结果中, 如果没有找到任何信息, 就显示一个失败信息。

```
private void RefreshButton_Click(object sender, RoutedEventArgs e)
{

```

```
// Get local peers
PeerNearMeCollection peersNearMe = PeerCollaboration.GetPeersNearMe();

// Prepare for new peers
PeerList.Items.Clear();

// Examine peers
foreach (PeerNearMe peerNearMe in peersNearMe)
{
    PeerList.Items.Add(
        new PeerEntry
        {
            PeerNearMe = peerNearMe,
            PresenceStatus = peerNearMe.GetPresenceInfo(
                peerNearMe.PeerEndpoints[0]).PresenceStatus,
            DisplayString = peerNearMe.Nickname
        });
}

// Add failure message if necessary
if (PeerList.Items.Count == 0)
{
    PeerList.Items.Add(
        new PeerEntry
        {
            DisplayString = "No peers found."
        });
}
}
```

从这个例子可以看出，使用前面介绍的类，可以非常方便地与 PNM 服务交互操作。

## 47.4 小结

本章介绍了如何在应用程序中使用 .NET 3.5 中新增的 P2P 类实现对等网(P2P)功能。

我们探讨了 P2P 的解决方案类型，这些解决方案的构造，如何使用 PNRP 和 PNM，如何使用 System.Net.PeerToPeer 和 System.Net.PeerToPeer.Collaboration 命名空间中的类型，还了解了把 WCF 服务提供为 P2P 端点的有用技术。

如果读者对开发 P2P 应用程序感兴趣，就应进一步研究 PNM。还应了解对等机信道，WCF 服务利用该信道在多个客户机上同时进行通信。

下一章介绍 Syndication，以及如何显示 RSS 和 Atom feeds 中的数据。

# 第 48 章

## Syndication

我们有时要提供一些结构化的、常常改变的数据。在许多 Web 站点上, RSS 或 Atom 符号允许通过 feed 阅读器订阅。Really Simple Syndication(RSS)是一个 XML 格式, 允许联合信息。RSS 随着博客的流行而非常流行。这个 XML 信息非常便于使用 RSS 阅读器订阅。

目前, RSS 不仅用于博客, 还用于许多不同的数据源, 例如在线新闻杂志。常常改变的数据都可以由 RSS 或其继任的协议 Atom 提供。IE7 和 Outlook 2007 提供了 RSS 和 Atom 阅读器, 它们集成到了产品中。

.NET 3.5 用联合功能扩展了 Windows Communication Foundation(WCF)。Syndication 类在 System.ServiceModel.Syndication 命名空间中定义。这个命名空间中的类可以用于读写 RSS 和 Atom feed。

本章介绍如何创建 Syndication 阅读器, 如何提供数据。内容如下:

- System.ServiceModel.Syndication 命名空间概述
- Syndication 阅读器
- Syndication Feed

### 48.1 System.ServiceModel.Syndication 命名空间概述

System.ServiceModel.Syndication 是 .NET 3.5 中的一个新命名空间, 可以提供 RSS 或 Atom 格式的数据。

RSS 2.0 版本发布后, RSS 现在已经是 Really Simple Syndication 的缩写。在以前的版本中, 它的名称是 RDF Site Summary 和 Rich Site Summary。RDF 是 Resource Description Framework 的缩写。第一个版本是由 Netscape 创建的, 描述了其门户网站的内容。《纽约时报》在 2002 年开始为其读者提供 RSS 新闻来源的订阅, 之后 RSS 变得非常成功。图 48-1 显示了 RSS 徽标。如果站点显示了这个徽标, 就提供了 RSS 来源。



图 48-1

Atom 是 RSS 的继任者, 并通过 RFC 4287 成为一个推荐标准([www.ietf.org/rfc/rfc4287.txt](http://www.ietf.org/rfc/rfc4287.txt))。RSS 和 Atom 之间的主要区别是可以用一个数据项定义的内容。在 RSS 上, 描述元素可以包含简单的文本或 HTML 元素, 阅读应用程序并不关心这些内容。而 Atom 要求用 type 属性为内容定义特定的类型, 还允许包含带指定命名空间的 XML 内容。

表 48-1 列出了.NET 3.5 中允许创建联合来源的类。这些类独立于联合类型 RSS 或 Atom。

表 48-1	
类	说 明
SyndicationFeed	SyndicationFeed 表示来源的顶级元素。在 Atom 中，顶级元素是<feed>，RSS 把<rss> 定义为顶级元素。使用静态方法 Load()可以通过 XmlReader 读取来源 这个类的属性 Authors、Categories、Contributors、Copyright、ImageUrl、Links、Title 和 Items 可以定义子元素
SyndicationPerson	SyndicationPerson 表示一个人，他的 Name、Email 和 Uri 可以赋予 Authors 和 Contributors 集合
SyndicationItem	来源由多个项组成。一项的属性有 Authors、Contributors、Copyright 和 Centent
SyndicationLink	SyndicationLink 表示来源或项的链接。这个类定义了属性 Title 和 Uri
SyndicationCategory	来源可以把项组合到类别中。类别的关键字可以设置为 SyndicationCategory 的 Name 和 Label 属性
SyndicationContent	SyndicationContent 是一个抽象基类，描述了项的内容。内容的类型可以是 HTML、纯 文本、XHTML、XML 或 URL，分别用具体的类 TextSyndication- Content、 UrlSyndicationContent 和 XmlSyndicationContent 来描述
SyndicationElement- Extension	对于扩展元素，可以添加额外的内容。SyndicationElementExtension 可用于给来源、分 类、人、链接和项添加信息

为了把来源转换为 RSS 和 Atom 格式，可以使用表 48-2 中的类。

表 48-2	
类	说 明
Atom10FeedFormatter Rss20FeedFormatter	Atom10FeedFormatter 和 Rss20FeedFormatter 派生自抽象基类 SyndicationFeedFormatter。 Atom10FeedFormatter 把 SyndicationFeed 串行化为 Atom 1.0 格式，Rss20FeedFormatter 把 SyndicationFeed 串行化为 RSS 2.0 格式
Atom10ItemFormatter Rss20ItemFormatter	Atom10ItemFormatter 和 Rss20ItemFormatter 派生自抽象基类 SyndicationItemFormatter。 Atom10ItemFormatter 把 SyndicationItem 串行化为 Atom 1.0 格式，Rss20ItemFormatter 把 SyndicationItem 串行化为 RSS 2.0 格式

48.2 Syndication 阅读器

第一个例子是一个 Syndication 阅读器应用程序，其用户界面用 WPF 开发，如图 48-2 所示。

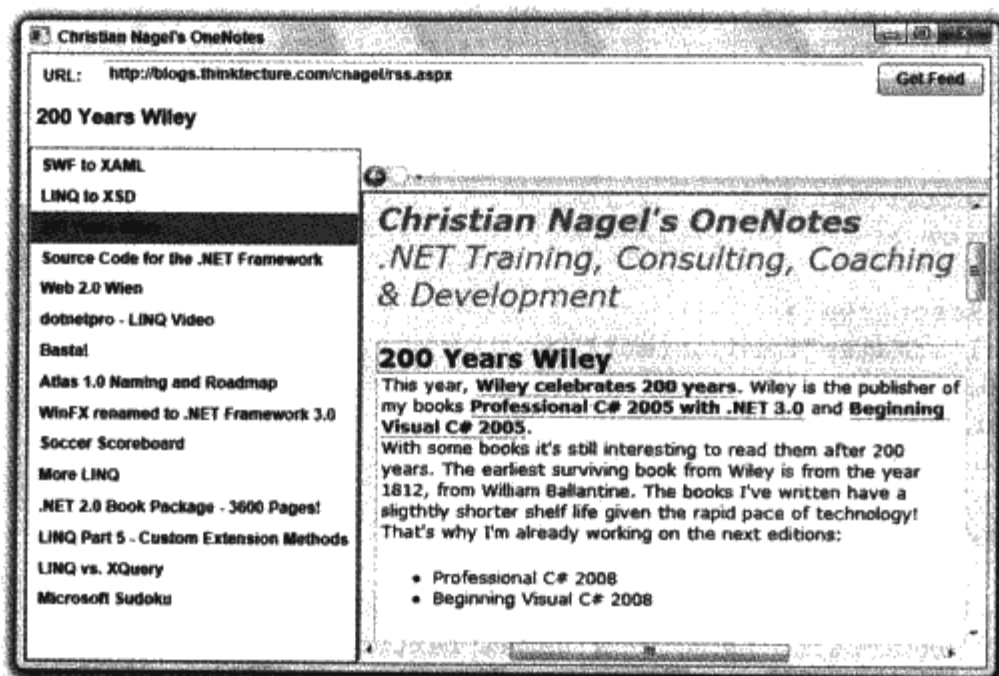


图 48-2

要使用 Syndication API，必须在应用程序中引用程序集 `System.ServiceModel.Web`。`OnGetFeed()` 事件处理程序关联到 `Get Feed` 按钮的 `Click` 事件上。读取应用程序所需的代码非常简单。首先，把 RSS 来源中的 XML 内容读入 `System.Xml` 命名空间的 `XmlReader` 类中。`Rss20FeedFormatter` 在 `ReadFrom()` 方法中接收一个 `XmlReader` 参数。对于数据绑定，把返回 `SyndicationFeed` 的 `Feed` 属性赋予 `Window` 的 `DataContext`，返回 `IEnumerable<SyndicationItem>` 的 `Feed.Items` 属性赋予 `DockPanel` 容器的 `DataContext`。

```
private void OnGetFeed(object sender, RoutedEventArgs e)
{
    XmlReader reader = XmlReader.Create(textUrl.Text);
    Rss20FeedFormatter formatter = new Rss20FeedFormatter();
    formatter.ReadFrom(reader);
    reader.Close();
    this.DataContext = formatter.Feed;
    this.feedContent.DataContext = formatter.Feed.Items;
}
```

定义用户界面的 XAML 代码如下所示。`Window` 类的 `Title` 属性绑定到 `SyndicationFeed` 的 `Title.Text` 属性上，以显示来源的标题。

在 XAML 代码中，定义了 `DockPanel` 容器 `heading`，它包含一个绑定到 `Title.Text` 上的标签和一个绑定到 `Description.Text` 上的标签。这些标签都包含在 `DockPanel` 容器 `feedContent` 中，而 `feedContent` 绑定到 `Feed.Item` 属性上，所以这些标签获得了当前选中项的标题和描述信息。

一个项列表显示在列表框中，该列表框使用 `ItemTemplate` 把标签绑定到 `Title` 上。

`DockPanel` 容器 `content` 包含一个 `Frame` 元素，该元素把 `Source` 属性绑定到项的第一个链接上。之后，`Frame` 控件使用 `Web` 浏览器控件显示链接中的内容，如图 48-2 所示。

```
< Window x:Class="RSSReader.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="{Binding Path=Title.Text}" Height="300" Width="345" >
    < Window.Resources >
        < Style x:Key="listTitleStyle" TargetType="{x:Type ListBox}" >
```



```

    < Setter Property="ItemTemplate" >
      < Setter.Value >
        < DataTemplate >
          < Label Content="{Binding Title.Text}" / >
        < /DataTemplate >
      < /Setter.Value >
    < /Setter >
  < /Style >
< /Window.Resources >
< DockPanel x:Name="feedContent" >
  < Grid DockPanel.Dock="Top" >
    < Grid.ColumnDefinitions >
      < ColumnDefinition Width="50" / >
      < ColumnDefinition Width="*" / >
      < ColumnDefinition Width="90" / >
    < /Grid.ColumnDefinitions >
    < Label Grid.Column="0" Margin="5" > URL: < /Label >
    < TextBox Grid.Column="1" x:Name="textUrl" MinWidth="150"
      Margin="5" > http://blogs.thinktecture.com/cnagel/rss.aspx
    < /TextBox >
    < Button Grid.Column="2" Margin="5" MinWidth="80"
      Click="OnGetFeed" > Get Feed < /Button >
    < /Grid >
  < DockPanel DockPanel.Dock="Top" x:Name="heading" >
    < Label DockPanel.Dock="Top" Content="{Binding Path=Title.Text}"
      FontSize="16" / >
    < Label DockPanel.Dock="Top"
      Content="{Binding Path=Description.Text}" / >
  < /DockPanel >
  < ListBox DockPanel.Dock="Left" ItemsSource="{Binding}"
    Style="{StaticResource listTitleStyle}"
    IsSynchronizedWithCurrentItem="True" / >
  < DockPanel x:Name="content" >
    < Label DockPanel.Dock="Top"
      Content="{Binding Path=Description.Text}" > < /Label >
    < Frame Source="{Binding Path=Links[0].Uri}" >
    < /Frame >
  < /DockPanel >
< /DockPanel >
< /Window >

```

### 48.3 提供 SyndicationFeed

读取 Syndication 来源时可以使用 Syndication API。Syndication API 还可以给 RSS 和 Atom 客户提供 Syndication 来源。

为此，Visual Studio 2008 提供了可用作起点的 Syndication Service Library 模板。这个模板定义了对 System.ServiceModel.Web 库的引用，并添加了一个应用程序配置文件，来定义 WCF 端点。

为了给 Syndication 来源提供数据，可以使用 LINQ 提供程序 LINQ to SQL。在示例应用程序中，使用了 Formula 1 数据库，它可以和本书的示例应用程序一起从 Wrox 网站上下载 (www.wrox.com)。名为 Formula1 的 LINQ to SQL Classes 项添加到项目中。其中，表 Racers、RaceResults、Races 和 Circuits 映射到实用类 Racer、RaceResult、Race 和 Circuit 上，如图 48-3 所示。

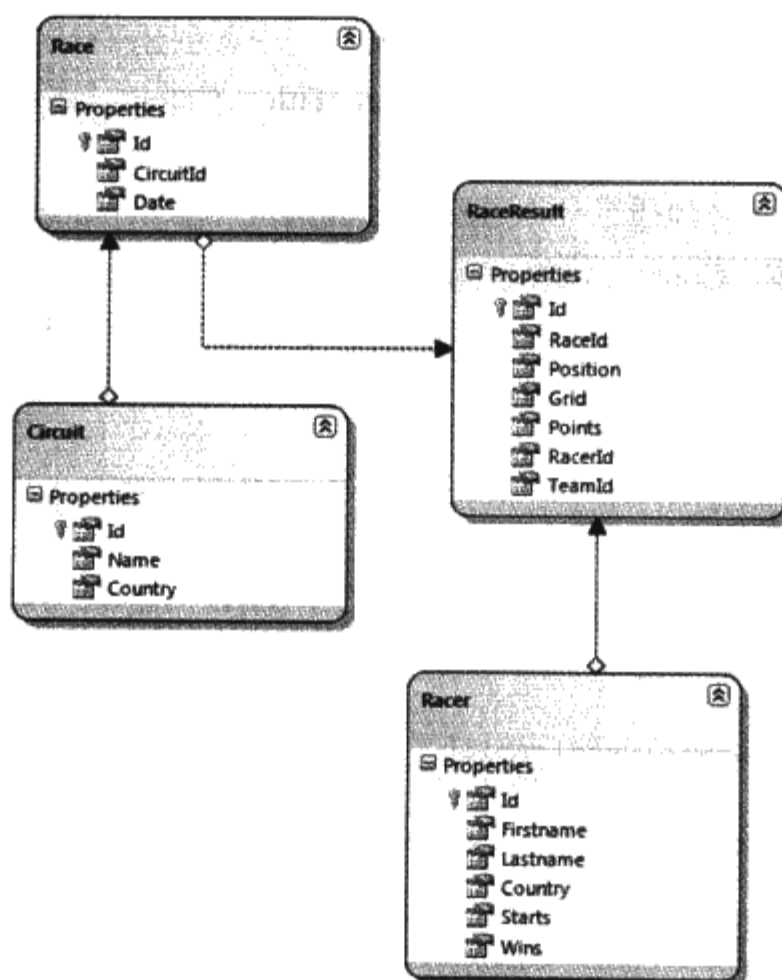


图 48-3

提示:

LINQ to SQL 详见第 27 章。

项目模板创建了一个文件 IService1.cs, 它包含 WCT 服务的合同。该接口包含 CreateFeed() 方法, 这个方法返回一个 SyndicationFeedFormatter。因为 SyndicationFeed-Formatter 是一个抽象类, 所返回的实际类型是 Atom10FeedFormatter 或 Rss20FeedFormatter, 而这些类型用 ServiceKnownTypeAttribute 列出, 所以串行化时类型是已知的。

属性 WebGet 定义了可以从一个简单的 HTTP GET 请求中调用的操作, 该操作可用于请求 Syndication 来源。WebMessageBodyStyle.Bare 指定, 发送结果(Syndication 来源中的 XML)时, 就好像这些 XML 没有添加 XML 封装元素一样。

```

using System.ServiceModel;
using System.ServiceModel.Syndication;
using System.ServiceModel.Web;
namespace Wrox.ProCSharp.Syndication
{
    [ServiceContract]
    [ServiceKnownType(typeof(Atom10FeedFormatter))]
    [ServiceKnownType(typeof(Rss20FeedFormatter))]
    public interface IFormula1Feed
    {
        [OperationContract]
        [WebGet(UriTemplate = "*", BodyStyle = WebMessageBodyStyle.Bare)]
        SyndicationFeedFormatter CreateFeed();
    }
}

```

该服务的实现代码在类 `Formula1Feed` 中定义。其中，创建了一个 `SyndicationFeed` 项，这个类还指定了各个属性，如 `Generator`、`Language`、`Title`、`Categories` 和 `Authors`。`Items` 属性填充了一个 LINQ to SQL 查询，该查询请求 2007 年一级方程式大奖赛的获奖者。在这个查询的 `select` 子句中，为每个获奖者创建了一个 `SyndicationItem`。在 `SyndicationItem` 中，给 `Title` 属性赋予包含举办国的纯文本。`Content` 属性使用 LINQ to XML 来填充。`XElement` 类用于创建能由浏览器解释的 XHTML 代码。这个内容显示了比赛的日期、举办国和获奖者的姓名。

根据请求 `Syndication` 的查询字符串，用 `Atom10FeedFormatter` 或 `Rss20FeedFormatter` 格式化 `SyndicationFeed`。

```
using System;
using System.Linq;
using System.ServiceModel.Syndication;
using System.ServiceModel.Web;
using System.Xml.Linq;
namespace Wrox.ProCSharp.Syndication
{
    public class Formula1Feed : IFormula1Feed
    {
        public SyndicationFeedFormatter CreateFeed()
        {
            // Create a new Syndication Feed.
            SyndicationFeed feed = new SyndicationFeed();
            feed.Generator = "Pro C# 2008 Sample Feed Generator";
            feed.Language = "en-us";
            feed.LastUpdatedTime = new DateTimeOffset(DateTime.Now);
            feed.Title = SyndicationContent.CreatePlaintextContent(
                "Formula1 results");
            feed.Categories.Add(new SyndicationCategory("Formula1"));
            feed.Authors.Add(new SyndicationPerson("web@christiannagel.com",
                "Christian Nagel", "http://www.christiannagel.com"));
            feed.Description = SyndicationContent.CreatePlaintextContent(
                "Sample Formula 1");
            Formula1DataContext data = new Formula1DataContext();
            feed.Items = from racer in data.Racers
                from raceResult in racer.RaceResults
                where raceResult.Race.Date >
                    new DateTime(2007, 1, 1) &&
                    raceResult.Position == 1
                orderby raceResult.Race.Date
                select new SyndicationItem()
            {
                Title =
                    SyndicationContent.CreatePlaintextContent(
                        String.Format("G.P. {0}",
                            raceResult.Race.Circuit.Country)),
                Content = SyndicationContent.CreateXhtmlContent(
                    new XElement("p",
                        new XElement("h3", String.Format("{0}, {1}",
```

```

        raceResult.Race.Circuit.Country,
        raceResult.Race.Date.
        ToShortDateString()),
    new XElement("b", String.Format(
        "Winner: {0} {1}",
        racer.Firstname,
        racer.Lastname))).ToString())
    };
// Return ATOM or RSS based on query string
// rss - >
// http://localhost:8731/Design_Time_Addresses/SyndicationService/Feed1/
// atom - >
// http://localhost:8731/Design_Time_Addresses/SyndicationService/
// Feed1/?format=atom
string query =
    WebOperationContext.Current.IncomingRequest.UriTemplateMatch.
    QueryParameters["format"];
SyndicationFeedFormatter formatter = null;
if (query == "atom")
{
    formatter = new Atom10FeedFormatter(feed);
}
else
{
    formatter = new Rss20FeedFormatter(feed);
}
return formatter;
}
}
}

```

在 Visual Studio 2008 中启动该服务时，WCF Service Host 会启动，执行该服务，在 IE 中显示格式化的来源结果，如图 48-4 所示。

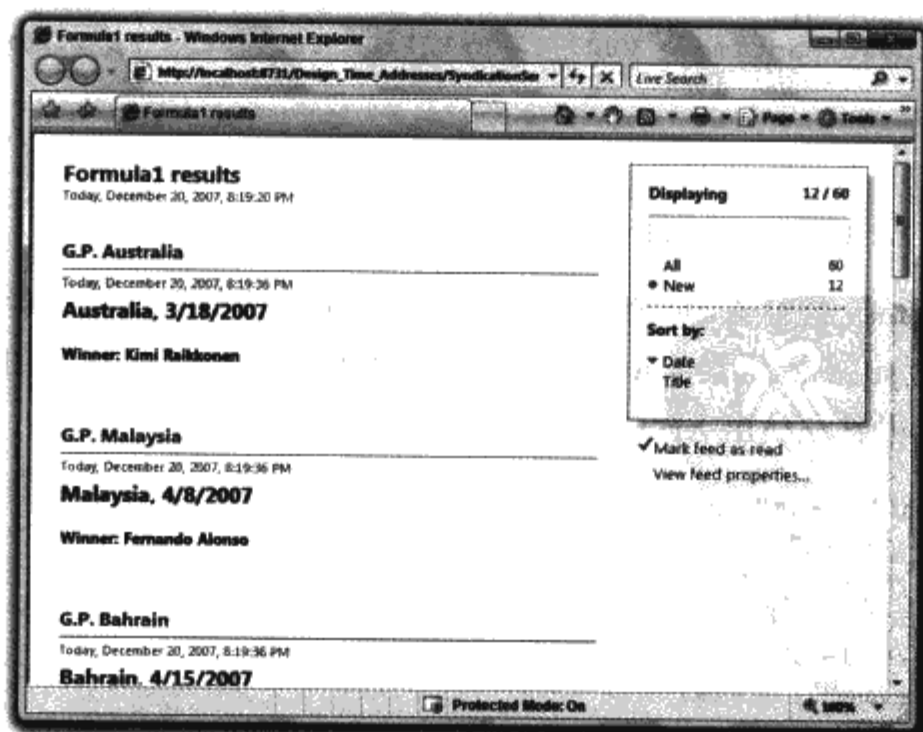


图 48-4

在对服务的默认请求中，返回了 RSS 来源。之后使用 rss 根元素提取 RSS 来源。在 RSS

中, Title 属性转换为 title 元素, Description 属性变成 description 元素。SyndicationFeed 的 Authors 属性包含 SyndicationPerson, 它仅使用电子邮件地址创建 managingEditor 元素。要在来源中添加更多的信息, 格式化器还要把一些 Atom 元素放在 RSS 来源中。RSS 来源中的 Atom 元素是提供不由 RSS 定义的信息的常见方式。

```
< ?xml version="1.0" encoding="utf-8"? >
< rss version="2.0" xmlns:atom="http://www.w3.org/2005/Atom"
  xmlns:cf="http://www.microsoft.com/schemas/rss/core/2005"
  xmlns:al0="http://www.w3.org/2005/Atom" >
  < channel
    xmlns:cfi="http://www.microsoft.com/schemas/rss/core/2005/internal"
    cfi:lastdownloaderror="None" >
    < title cf:type="text" > Formulal results < /title >
    < description cf:type="text" > Sample Formula 1 < /description >
    < language > en-us < /language >
    < managingEditor > web@christiannagel.com < /managingEditor >
    < atom:author >
      < atom:email > web@christiannagel.com < /atom:email >
    < /atom:author >
    < lastBuildDate > Tue, 04 Dec 2007 21:07:48 GMT < /lastBuildDate >
    < atom:updated > 2007-12-04T21:07:48Z < /atom:updated >
    < category > Formulal < /category >
    < generator > Pro C# 2008 Sample Feed Generator < /generator >
    < item >
      < title xmlns:cf="http://www.microsoft.com/schemas/rss/core/2005"
        cf:type="text" > G.P. Australia < /title >
      < description xmlns:cf="http://www.microsoft.com/schemas/rss/core/2005"
        cf:type="html" > &
        lt;p & gt; & lt;h3 & gt;Australia, 18.03.2007 & lt;/h3 & gt;
        & lt;b & gt;Winner: Kimi Raikkonen & lt;/b & gt; & lt;/p & gt;
      < /description >
      < cfi:id > 47 < /cfi:id > < cfi:read > true < /cfi:read >
      < cfi:downloadurl >
        http://localhost:8731/Design_Time_Addresses/SyndicationService/Feed1/
      < /cfi:downloadurl >
      < cfi:lastdownloadtime > 2007-12-04T21:05:16.486Z < /cfi:lastdownloadtime >
    < /item >
    < item >
      <!-- ... -->
    < /channel >
  < /rss >
```

用查询?format=atom 返回 Atom 格式化的来源, 结果如下所示。根元素现在是 feed 元素, Description 属性变成 subtitle 元素, Authors 属性的值现在完全不同于前面的 RSS 来源。Atom 允许不解码内容。也很容易找出 XHTML 元素。

```
< feed xml:lang="en-us" xmlns="http://www.w3.org/2005/Atom" >
  < title type="text" > Formulal results < /title >
  < subtitle type="text" > Sample Formula 1 < /subtitle >
  < id > uuid:c19284e7-aa40-4bc2-9be8-f1960b0f747e;id=1 < /id >
  < updated > 2007-12-05T00:46:35+01:00 < /updated >
  < category term="Formulal"/ >
  < author >
    < name > Christian Nagel < /name >
    < uri > http://www.christiannagel.com < /uri >
    < email > web@christiannagel.com < /email >
  < /author >
```



```

< generator > Pro C# 2008 Sample Feed Generator < /generator >
< entry >
  < id > uuid:c19284e7-aa40-4bc2-9be8-f1960b0f747e;id=2 < /id >
  < title type="text" > G.P. Australia < /title >
  < updated > 2007-12-04T23:46:43Z < /updated >
  < content type="xhtml" >
    < p > < h3 > Australia, 18.03.2007 < /h3 > < b > Winner: Kimi Raikkonen < /b
> < /p >
  < /content >
< /entry >
< entry >
  < id > uuid:c19284e7-aa40-4bc2-9be8-f1960b0f747e;id=3 < /id >
  < title type="text" > G.P. Malaysia < /title >
  < updated > 2007-12-04T23:46:43Z < /updated >
  < content type="xhtml" >
    < p > < h3 > Malaysia, 08.04.2007 < /h3 > < b > Winner: Fernando Alonso < /b
> < /p >
  < /content >
< /entry >
<!-- ... -->
< /feed >

```

## 48.4 小结

本章介绍了.NET 3.5中新增的System.ServiceModel.Syndication命名空间中的类如何用于创建接收来源的应用程序和提供来源的应用程序。Syndication API支持RSS 2.0和Atom 1.0。在这些标准制定时,可以使用新的格式化器。SyndicationXXX类独立于所生成的格式。抽象类SyndicationFeedFormatter的具体实现方式定义了使用什么属性,它们如何转换为特定的格式。

本章完成了本书的通信部分。我们学习了直接使用套接字和所提供的抽象层的通信技术。Windows Communication Foundation(WCF)技术在几章中做了讨论。WCF利用第45章介绍的消息队列技术,提供了断开连接的通信模型。第44章介绍了WCF和已有COM+应用程序的集成。

本书介绍了C#的语言特性,包括C# 3.0中的新特性,例如扩展方法和LINQ查询。还演示了C# 3.0特性的使用,并介绍了.NET Framework的核心特性、对数据库和XML的数据访问、用户界面、Windows Forms、Windows Presentation Foundation、ASP.NET和Microsoft Office。

还有许多相关的内容。附录介绍了可以把对象映射到关系数据库上的ADO.NET实体映射技术、Windows Vista和Windows Server 2008的应用程序,还比较了C#、VB和C++/CLI。

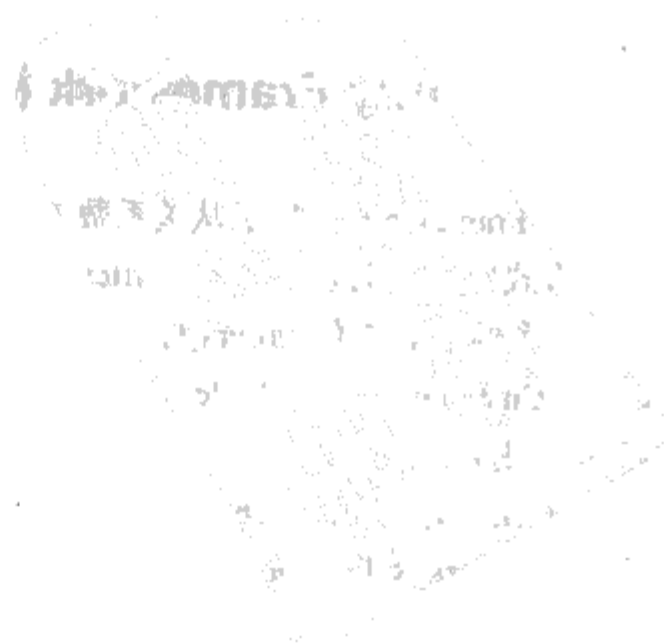
# 第Ⅶ部分

## 附 录

附录 A ADO.NET Entity Framework

附录 B C#、Visual Basic 和 C++/CLI

附录 C Windows Vista 和 Windows Server 2008



# ADO.NET Entity Framework

ADO.NET Entity Framework 是一个基于 .NET 3.5 的对象-关系的映射框架。第 27 章介绍了通过 LINQ to SQL 的对象-关系映射。LINQ to SQL 为关联和继承提供了简单的映射特性。ADO.NET Entity Framework 则为关联和继承提供了更多的选项。LINQ to SQL 和 ADO.NET Entity Framework 的另一个区别是，ADO.NET Entity Framework 是一个基于提供程序的模型，允许其他数据库供应商使用它。

本附录的内容如下：

- ADO.NET Entity Framework
- Entity Framework 层
- 实体
- 对象环境
- 关系
- 对象查询
- 更新
- LINQ to Entities

提示：

本附录基于这个框架的 Beta 3 版本，这是因为它在 .NET 3.5 产品发布后的几个月才发布，所以一些类和方法的名称与这里提到的不同。

本附录使用 Books、Formula1 和 Northwind 数据库。Northwind 数据库可以从 [msdn.microsoft.com](http://msdn.microsoft.com) 上下载，Books 和 Formula1 数据库包含在代码示例的下载包中。

## A.1 ADO.NET Entity Framework 概述

ADO.NET Entity Framework 提供了从关系数据库模式到对象的映射。关系数据库和面向对象的语言用不同的方式定义了关系。例如，Microsoft 示例数据库 Northwind 包含 Customers 和 Orders 表。要访问某个顾客的所有 Orders 行，需要执行一个 SQL join 语句。在面向对象的语言中，则常常定义一个 Customer 和一个 Order 类，使用 Customer 类的 Orders 属性访问顾客的订单。

对于对象-关系映射，自从 .NET 1.0 以来，就可以使用 DataSet 类和类型化的数据集。DataSet 非常类似于数据库的结构，它包含 DataTable、DataRow、DataColumn 和 DataRelation 类。ADO.NET Entity Framework 支持直接定义完全独立于数据库结构的实体类，并把它们映射到数

数据库的表和关系上。在应用程序中使用对象，应用程序就可以独立于数据库的修改。

ADO.NET Entity Framework 使用 Entity SQL 定义基于实体的数据库查询。LINQ to Entities 允许使用 LINQ 语法来查询数据。

对象环境保存了变化的实体信息，在把实体写回数据库时，提供这些信息。

包含 ADO.NET Entity Framework 中的类的命名空间如表 A-1 所示。

表 A-1

命名空间	说明
System.Data	这是用于 ADO.NET 的主要命名空间。在 ADO.NET Entity Framework 中，这个命名空间包含了与实体相关的异常类，例如 MappingException 和 QueryException
System.Data.Common	这个命名空间包含了由 .NET 数据提供程序共享的类。类 DbProviderServices 是一个抽象类，必须由 ADO.NET Entity Framework 提供程序实现
System.Data.Common .CommandTrees	这个命名空间包含建立表达式树的类
System.Data.Entity.Design	这个命名空间包含由设计器用于创建 Entity Data Model(EDM)文件的类
System.Data.EntityClient	这个命名空间中的类由 .NET Framework 数据提供程序访问 ADO.NET Entity Framework。EntityConnection、EntityCommand 和 EntityReader 可用于访问 Entity Framework
System.Data.Objects	这个命名空间包含了查询和更新数据库的类。类ObjectContext封装了与数据库的连接，用作创建、读取、更新和删除方法的网关。类ObjectQuery表示对数据库的一个查询。CompileQuery是一个高速缓存的查询
System.Data.Objects .DataClasses	这个命名空间包含实体需要的类和接口

A.2 Entity Framework 层

ADO.NET Entity Framework 提供了几个把数据库表映射到对象上的层。可以从一个数据库模式开始，使用 Visual Studio 项模板创建完整的映射。还可以先用设计器设计实体类，再把它映射到数据库上，在该数据库中，表和表之间的关系可以有完全不同的结构。

需要定义的层如下：

- 逻辑层：这个层定义了关系数据
- 概念层：这个层定义了 .NET 类
- 映射层：这个层定义了从 .NET 类到关系表和关联的映射。

下面先从一个简单的数据库模式开始，如图 A-1 所示，其中包含表 Books 和 Authors，以及关联表 BookAuthors，它把作者映射到图书上。

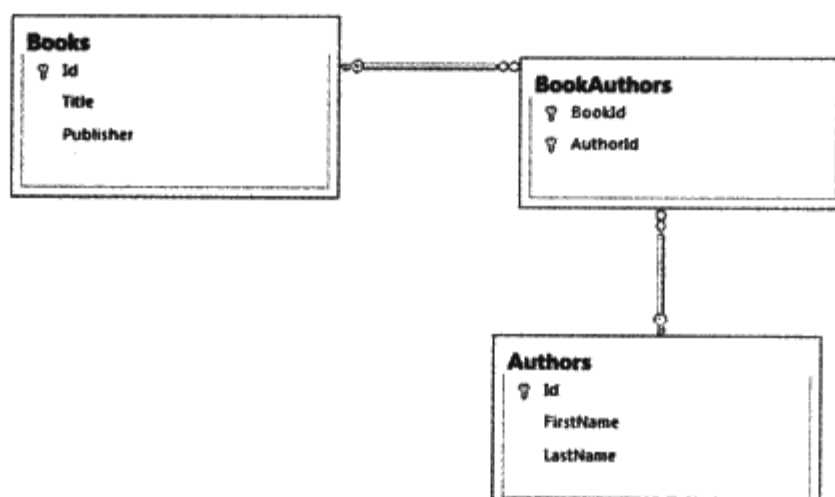


图 A-1

### A.2.1 逻辑层

逻辑层是由 Store Schema Definition Language(SSDL)定义的,描述了数据库表及其关系的结构。

下面的代码使用 SSDL 描述了 3 个表: Books、Authors 和 BookAuthors。EntityType 元素描述了所有带 EntitySet 元素的表和带 AssociationSet 元素的表。表的部分用 EntityType 元素定义。在 EntityType Books 中,列 Id、Title 和 Publisher 是用 Property 元素定义的。Property 元素包含了定义数据类型的 XML 属性。Key 元素定义了表的键。

```

< Schema Namespace="BookEntities.Store" Alias="Self"
  ProviderManifestToken="09.00.3054"
  xmlns="http://schemas.microsoft.com/ado/2006/04/edm/ssdl" >
  < EntityContainer Name="dbo" >
    < EntitySet Name="Authors"
      EntityType="Wrox.ProCSharp.Entities.Store.Authors" / >
    < EntitySet Name="BookAuthors"
      EntityType=" Wrox.ProCSharp.Entities.Store.BookAuthors" / >
    < EntitySet Name="Books" EntityType=" Wrox.ProCSharp.Entities.Store.Books" /
    >
    < AssociationSet Name="FK_BookAuthors_Authors"
      Association=" Wrox.ProCSharp.Entities.Store.FK_BookAuthors_Authors" >
      < End Role="Authors" EntitySet="Authors" / >
      < End Role="BookAuthors" EntitySet="BookAuthors" / >
    < /AssociationSet >
    < AssociationSet Name="FK_BookAuthors_Books"
      Association="BookDemoEntities.Store.FK_BookAuthors_Books" >
      < End Role="Books" EntitySet="Books" / >
      < End Role="BookAuthors" EntitySet="BookAuthors" / >
    < /AssociationSet >
  < /EntityContainer >
  < EntityType Name="Authors" >
    < Key > < PropertyRef Name="Id" / > < /Key >
    < Property Name="Id" Type="int" Nullable="false"
      StoreGeneratedPattern="Identity" / >
    < Property Name="FirstName" Type="nvarchar" Nullable="false"
      MaxLength="50" / >
    < Property Name="LastName" Type="nvarchar" Nullable="false"
      MaxLength="50" / >
  < /EntityType >
  < EntityType Name="BookAuthors" >
    < Key > < PropertyRef Name="BookId" / >
  
```



```

    < PropertyRef Name="AuthorId" / > < /Key >
    < Property Name="BookId" Type="int" Nullable="false" / >
    < Property Name="AuthorId" Type="int" Nullable="false" / >
  < /EntityType >
  < EntityType Name="Books" >
    < Key > < PropertyRef Name="Id" / > < /Key >
    < Property Name="Id" Type="int" Nullable="false"
      StoreGeneratedPattern="Identity" / >
    < Property Name="Title" Type="nvarchar" Nullable="false"
      MaxLength="50" / >
    < Property Name="Publisher" Type="nvarchar" Nullable="false"
      MaxLength="50" / >
  < /EntityType >
  < Association Name="FK_BookAuthors_Authors" >
    < End Role="Authors"
      Type=" Wrox.ProCSharp.Entities.Store.Authors" Multiplicity="1" / >
    < End Role="BookAuthors"
      Type=" Wrox.ProCSharp.Entities.Store.BookAuthors"
      Multiplicity="*" / >
    < ReferentialConstraint >
      < Principal Role="Authors" > < PropertyRef Name="Id" / >
        < /Principal >
      < Dependent Role="BookAuthors" > < PropertyRef Name="AuthorId" / >
        < /Dependent >
    < /ReferentialConstraint >
  < /Association >
  < Association Name="FK_BookAuthors_Books" >
    < End Role="Books" Type=" Wrox.ProCSharp.Entities.Store.Books"
      Multiplicity="1" / >
    < End Role="BookAuthors"
      Type=" Wrox.ProCSharp.Entities.Store.BookAuthors"
      Multiplicity="*" / >
    < ReferentialConstraint >
      < Principal Role="Books" > < PropertyRef Name="Id" / >
        < /Principal >
      < Dependent Role="BookAuthors" > < PropertyRef Name="BookId" / >
        < /Dependent >
    < /ReferentialConstraint >
  < /Association >
< /Schema >

```

### A.2.2 概念层

概念层定义了.NET类。这个层是用 Conceptual Schema Definition Language(CSDL)定义的。图 A-2 显示了用 ADO.NET Entity 数据模型设计器定义的实体 Author 和 Book。

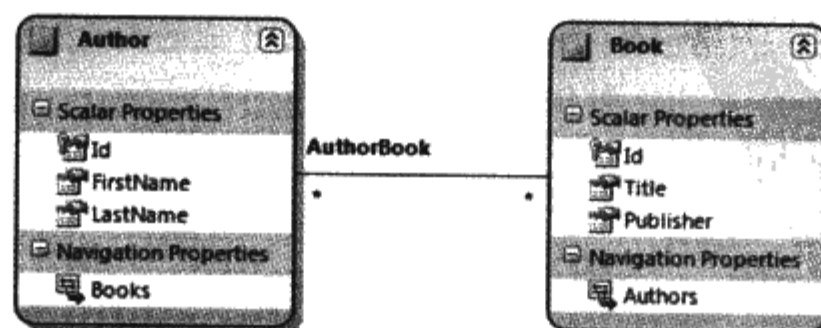


图 A-2

下面是定义实体类型 Author 和 Book 的 CSDL 内容。它们是在 Books 数据库中创建的。

```
< Schema Namespace="BookEntities" Alias="Self"
  xmlns="http://schemas.microsoft.com/ado/2006/04/edm" >
  < EntityContainer Name="BookEntities" >
    < EntitySet Name="Authors"
      EntityType="Wrox.ProCSharp.Entities.Author" / >
    < EntitySet Name="Books"
      EntityType="Wrox.ProCSharp.Entities.Book" / >
    < AssociationSet Name="BookAuthors"
      Association=" Wrox.ProCSharp.Entities.BookAuthors" >
      < End Role="Authors" EntitySet="Authors" / >
      < End Role="Books" EntitySet="Books" / >
    < /AssociationSet >
  < /EntityContainer >
  < EntityType Name="Author" >
    < Key >
      < PropertyRef Name="Id" / >
    < /Key >
    < Property Name="Id" Type="Int32" Nullable="false" / >
    < Property Name="FirstName" Type="String" Nullable="false"
      MaxLength="50" / >
    < Property Name="LastName" Type="String" Nullable="false"
      MaxLength="50" / >
    < NavigationProperty Name="Books" Relationship="BookDemoEntities
      .BookAuthors" FromRole="Authors" ToRole="Books" / >
  < /EntityType >
  < EntityType Name="Book" >
    < Key >
      < PropertyRef Name="Id" / >
    < /Key >
    < Property Name="Id" Type="Int32" Nullable="false" / >
    < Property Name="Title" Type="String" Nullable="false"
      MaxLength="50" / >
    < Property Name="Publisher" Type="String" Nullable="false"
      MaxLength="50" / >
    < NavigationProperty Name="Authors"
      Relationship=" Wrox.ProCSharp.Entities.BookAuthors"
      FromRole="Books" ToRole="Authors" / >
  < /EntityType >
  < Association Name="BookAuthors" >
    < End Type=" Wrox.ProCSharp.Entities.Author" Role="Authors"
      Multiplicity="*" / >
    < End Type=" Wrox.ProCSharp.Entities.Book" Role="Books"
      Multiplicity="*" / >
  < /Association >
< /Schema >
```

实体用 `EntityType` 元素定义，它包含 `Key`、`Property` 和 `NavigationProperty` 元素，描述了所创建的类的属性。`Property` 元素包含的属性描述了设计器生成的类的.NET 属性的名称和类型。`Association` 元素连接了类型 `Author` 和 `Book`。`Multiplicity="**"`表示一个作者可以编写多本图书，一本图书可以由多个作者编写。

### A.2.3 映射层

映射层使用 `Mapping Specification Language(MSL)`把 `CSDL` 中的实体类型定义映射到 `SSDL` 上。下面的规范包含一个 `Mapping` 元素，该元素包含的 `EntityTypeMapping` 元素引用了 `CSDL` 的 `Book` 类型，定义了 `MappingFragment`，来引用 `SSDL` 中的 `Authors` 表。`ScalarProperty` 把带 `Name`

属性的.NET 类的属性映射到带 ColumnName 属性的数据库表的列上。

```
< Mapping Space="C-S"
  xmlns="urn:schemas-microsoft-com:windows:storage:mapping:CS" >
  < EntityContainerMapping StorageEntityContainer="dbo"
    CdmEntityContainer="BookEntities" >
    < EntitySetMapping Name="Authors" >
      < EntityTypeMapping
        TypeName="IsTypeOf(Wrox.ProCSharp.Entities.Author)" >
        < MappingFragment StoreEntitySet="Authors" >
          < ScalarProperty Name="LastName" ColumnName="LastName" / >
          < ScalarProperty Name="FirstName" ColumnName="FirstName" / >
          < ScalarProperty Name="Id" ColumnName="Id" / >
        < /MappingFragment >
      < /EntityTypeMapping >
    < /EntitySetMapping >
    < EntitySetMapping Name="Books" >
      < EntityTypeMapping
        TypeName="IsTypeOf(Wrox.ProCSharp.Entities.Book)" >
        < MappingFragment StoreEntitySet="Books" >
          < ScalarProperty Name="Publisher" ColumnName="Publisher" / >
          < ScalarProperty Name="Title" ColumnName="Title" / >
          < ScalarProperty Name="Id" ColumnName="Id" / >
        < /MappingFragment >
      < /EntityTypeMapping >
    < /EntitySetMapping >
    < AssociationSetMapping Name="AuthorBook"
      TypeName=" Wrox.ProCSharp.Entities.AuthorBook"
      StoreEntitySet="BookAuthors" >
      < EndProperty Name="Book" >
        < ScalarProperty Name="Id" ColumnName="BookId" / >
      < /EndProperty >
      < EndProperty Name="Author" >
        < ScalarProperty Name="Id" ColumnName="AuthorId" / >
      < /EndProperty >
    < /AssociationSetMapping >
  < /EntityContainerMapping >
< /Mapping >
```

## A.3 实体

用设计器和 CSDL 创建的实体类一般派生于基类 EntityObject，如下面代码中的 Book 类。

这个类派生于基类 EntityObject，定义了了在 set 访问器中触发信息的改变的属性。所创建的是一个部分类，可以在新的源文件中扩展，这个新的源文件定义了同一个命名空间中的同一个类。在 set 访问器中调用的方法，如 OnTitleChanging()和 OnTitleChanged()也是部分方法，所以可以在类的定制扩展中实现它们。Authors 属性使用 RelationshipManager 给作者返回 Books。

```
[EdmEntityTypeAttribute
  (NamespaceName="Wrox.ProCSharp.Entities", Name="Book")]
[DataContractAttribute()]
[Serializable()]
public partial class Book : global::System.Data.Objects.DataClasses.EntityObject
{
  public static Book CreateBook(int ID, string title, string publisher)
  {
```



```
        this.OnPublisherChanged();
    }
}
private string _Publisher;
partial void OnPublisherChanging(string value);
partial void OnPublisherChanged();
[EdmRelationshipNavigationPropertyAttribute("BookDemoEntities", "AuthorBook",
"Author")]
[XmlIgnoreAttribute()]
[SoapIgnoreAttribute()]
[BrowsableAttribute(false)]
public EntityCollection < Author > Authors
{
    get
    {
        return ((IEntityWithRelationships)(this)).RelationshipManager.
            GetRelatedCollection < Author >
                ("WroxProCSharp.Entities.AuthorBook", "Author");
    }
}
}
```

与实体类相关的重要类和接口如表 A-2 所示。除了 INotifyPropertyChanging 和 INotifyPropertyChanged 之外，所有的类型都是在 System.Data.Objects.DataClasses 命名空间中定义的。

表 A-2

类 或 接 口	说 明
StructuralObject	StructuralObject 是类 EntityObject 和 ComplexObject 的基类。这个类实现了接口 INotifyPropertyChanging 和 INotifyPropertyChanged
INotifyPropertyChanging INotifyPropertyChanged	这些接口定义了 PropertyChanging 和 PropertyChanged 事件，允许在对象的状态变化时订阅信息。这两个接口与其他类和接口不同，因为它们是在 System.ComponentModel 命名空间中定义
EntityObject	这个类派生于 StructuralObject，实现了接口 IEntityWithKey、IEntityWithChangeTracker 和 IEntityWithRelationships。EntityObject 是一个常用的基类，用于映射到数据库表上的对象，包含一个键和对其他对象的关系
ComplexObject	这个类可以用作没有键的实体对象的基类，它派生于 StructuralObject，但没有实现与 EntityObject 类相同的其他接口
IEntityWithKey	这个接口定义了 EntityKey 属性，允许快速访问对象
IEntityWithChangeTracker	这个接口定义了方法 SetChangeTracker()，其中，实现了接口 IChangeTracker 的变化跟踪器可以从对象中获得状态变化的信息
IEntityWithRelationships	这个接口定义了只读属性 RelationshipManager，它返回一个用于在对象之间导航的 RelationshipManager 对象

提示：  
实体类不一定派生于基类 EntityObject 或 ComplexObject，而可以实现需要的接口。



Book 实体类很容易使用对象环境类 BookEntities 来访问。Book 属性返回一个可以迭代的 Book 对象集合：

```
BookEntities data = new BookEntities();
foreach (var book in data.Books)
{
    Console.WriteLine("{0}, {1}", book.Title, book.Publisher);
}
```

运行程序，就在控制台上显示从数据库中查询到的图书：

```
Professional C# 2008, Wrox Press
Beginning Visual C# 2008, Wrox Press
Working with Animation in Silverlight 1.0, Wrox Press
Professional WPF Programming, Wrox Press
```

## A.4 对象环境

要从数据库中检索数据，需要使用ObjectContext类。这个类定义了从实体对象到数据库的映射。在ADO.NET中，这个类可以和填充DataSet的数据适配器相比较。

设计器创建的BookEntities类派生于基类ObjectContext。这个类添加了构造函数，来传送连接字符串。在默认的构造函数中，连接字符串从配置文件中读取，也可以把一个已经打开的连接以EntityConnection实例的形式传送给构造函数。如果把没有打开的连接传送给构造函数，对象环境就会打开和关闭该连接。如果传送了打开的连接，也需要关闭它。

所创建的类定义了Books和Authors属性，它们返回一个ObjectQuery，该类还提供了添加作者和图书的方法AddToAuthors()和AddToBooks()。

```
public partial class BookEntities : ObjectContext
{
    public BookEntities() :
        base("name=BookEntities", "BookEntities") { }
    public BookEntities(string connectionString) :
        base(connectionString, "BookEntities") { }
    public BookEntities(EntityConnection connection) :
        base(connection, "BookEntities") { }
    [BrowsableAttribute(false)]
    public ObjectQuery < Author > Authors
    {
        get
        {
            if ((this._Authors == null))
            {
                this._Authors = base.CreateQuery < Author > ("[Authors]");
            }
            return this._Authors;
        }
    }
    private ObjectQuery < Author > _Authors;
    [BrowsableAttribute(false)]
    public ObjectQuery < Book > Books
    {
        get
        {
```

```
        if ((this._Books == null))
        {
            this._Books = base.CreateQuery < Book > ("[Books]");
        }
        return this._Books;
    }
}
private ObjectQuery < Book > _Books;
public void AddToAuthors(Author author)
{
    base.AddObject("Authors", author);
}
public void AddToBooks(Book book)
{
    base.AddObject("Books", book);
}
}
```

给 BookEntities 类的构造函数传送连接字符串时，EntityConnection 类型的连接字符串会定义关键字 Metadata，它有 3 个必选参数：映射文件的定界列表，不变的提供程序名 Provider(该提供程序用于访问数据源)和指定提供程序依赖的连接字符串的 Provider connection string。

```
EntityConnection conn = new EntityConnection(
    "Metadata=./BookModel.csdl|./BookModel.ssdl|./BookModel.msl;" +
    "Provider=System.Data.SqlClient;" +
    "Provider connection string=\\\"Data Source=(local);\" +
    "Initial Catalog=EntitiesDemo;Integrated Security=True\\\"");
```

ObjectContext 类给调用者提供了几个服务：

- 跟踪已经检索出来的实体对象。如果再次查询该对象，就从对象环境中提取。
- 保存实体的状态信息。可以获得已添加、修改和删除对象的信息。
- 更新对象环境中的实体，把改变的内容写入底层数据库。

ObjectContext 类的方法和属性如表 A-3 所示。

表 A-3

ObjectContext 类的方法和属性	说 明
Connection	返回一个与对象环境关联的 DbConnection 对象
MetadataWorkspace	返回一个 MetadataWorkspace 对象，该对象可用于读取元数据和映射信息
QueryTimeout	使用这个属性可以获取和设置对象环境查询的超时值
ObjectStateManager	这个属性返回一个 ObjectStateManager，该 ObjectStateManager 会跟踪检索出来的实体对象和该对象在对象环境中的变化
CreateQuery()	这个方法返回一个 ObjectQuery，从数据库中获取数据。前面的 Books 和 Authors 属性就使用这个方法返回一个 ObjectQuery
GetObjectByKey() TryGetObjectByKey()	这两个方法根据键从对象状态管理器或底层的数据库中返回对象。如果键不存在，GetObjectByKey() 就抛出一个 ObjectNotFoundException 类型的异常，而 TryGetObjectByKey()返回 false

(续表)

ObjectContext 类的方法和属性	说 明
AddObject()	这个方法在对象环境中添加一个新的实体对象。这个方法由方法 AddToAuthors()和 AddToBooks()调用
DeleteObject()	这个方法从对象环境中删除一个对象
Detach()	这个方法解除实体对象与对象环境的关联，使该对象不再在出现变化时被跟踪
Attach() AttachTo()	Attach()方法把解除关联的对象关联到数据库上。把对象关联回对象环境上，需要实体对象实现 IEntityWithKey 接口。AttachTo()方法不需要对象带有键，但需要一个实体集名称，需要关联的实体对象就放在这个实体集中
ApplyPropertyChanges()	如果对象从对象环境中解除了关联，就修改解除关联的对象，在把进行的修改应用于对象环境中的对象之前，就可以调用 ApplyPropertyChanges()方法应用这些修改。当解除关联的对象从 Web 服务中返回，在客户机上修改，再传送给 Web 服务时，就可以使用这个方法
Refresh()	实体对象存储在对象环境中时，数据库中的数据可以修改。为了用数据库中的变化进行刷新，可以使用 Refresh()方法。在这个方法中，可以传送一个 RefreshMode 枚举值。如果用于对象的值在数据库和对象环境中不同，就传送 ClientWins，修改数据库中的数据，若传送 StoreWins 值，就修改对象环境中的数据
SaveChanges()	在对象环境中添加、修改和删除对象，不会改变底层数据库中的对象。使用 SaveChanges()方法可以把这些修改保存到数据库中
AcceptAllChanges()	这个方法把对象环境中的对象状态改为未修改。SaveChanges()隐式调用这个方法

A.5 关系

实体类型 Book 和 Author 相互关联。一本书可以由一个或多个作者编写，一个作者也可以编写一本或多本图书。其关系基于关联类型的个数和多样性。ADO.NET Entity Framework 的第一个版本支持 Table per Type(TPT)和 Table per Hierarchy(TPH)。多样性可以是一对一、一对多和多对多。

A.5.1 Table per Hierarchy

在 TPH 中，数据库中的一个表对应实体类的一个层次结构。数据库表 Payments(如图 A-3 所示)包含的列对应于实体类型的一个层次结构。一些列在该层次结构中的所有实体中共享，例如 Id 和 Amount，CreditCard 列仅用于信用卡付费。

全部映射到同一个 Payments 表的实体类如图 A-4 所示。Payment 是一个抽象基类，它包含的属性用于该层次结构中的所有类型。派生于 Payment 的具体类有 CreditCardPayment、CashPayment 和 ChequePayment。除了基类的属性之外，CreditCardPayment 还有一个 CreditCard

属性, ChequePayment 还有一个 BankName 属性。

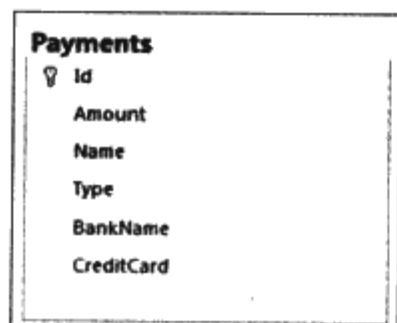


图 A-3

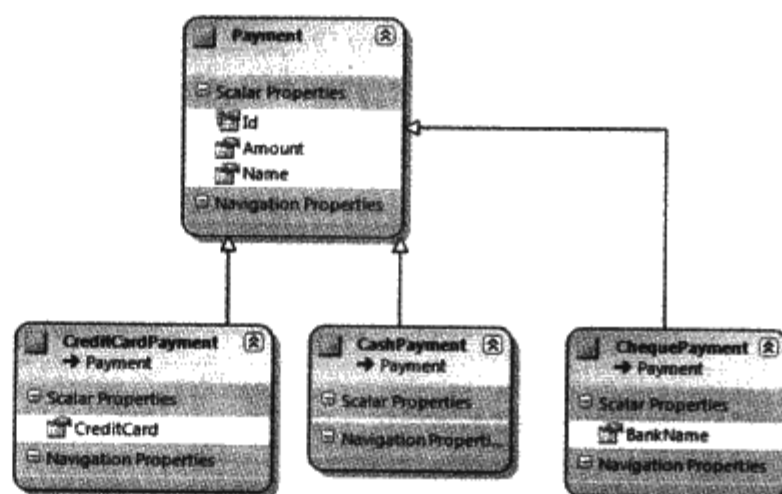


图 A-4

具体类的类型选择基于一个 Condition 元素, 如下面的 MSL 文件所示。其中, 类型根据 Type 列的值来选择。还可以使用其他用于选择类型的选项, 例如可以验证列是否不为空。

```

< Mapping Space="C-S"
  xmlns="urn:schemas-microsoft-com:windows:storage:mapping:CS" >
  < EntityContainerMapping StorageEntityContainer="dbo"
    CdmEntityContainer="EntitiesDemoEntities" >
    < EntitySetMapping Name="Payments" >
      < EntityTypeMapping TypeName="Wrox.ProCSharp.Entities.Payment" >
        < MappingFragment StoreEntitySet="Payments" >
          < ScalarProperty Name="Id" ColumnName="Id" / >
          < ScalarProperty Name="Amount" ColumnName="Amount" / >
          < ScalarProperty Name="Name" ColumnName="Name" / >
        < /MappingFragment >
      < /EntityTypeMapping >
      < EntityTypeMapping
        TypeName="Wrox.ProCSharp.Entities.CashPayment" >
        < MappingFragment StoreEntitySet="Payments" >
          < ScalarProperty Name="Id" ColumnName="Id" / >
          < Condition ColumnName="Type" Value="CASH" / >
        < /MappingFragment >
      < /EntityTypeMapping >
      < EntityTypeMapping
        TypeName="Wrox.ProCSharp.Entities.CreditCardPayment" >
        < MappingFragment StoreEntitySet="Payments" >
          < ScalarProperty Name="Id" ColumnName="Id" / >
          < ScalarProperty Name="CreditCard" ColumnName="CreditCard" / >
          < Condition ColumnName="Type" Value="CREDIT" / >
        < /MappingFragment >
      < /EntityTypeMapping >
      < EntityTypeMapping
        TypeName="Wrox.ProCSharp.Entities.ChequePayment" >
        < MappingFragment StoreEntitySet="Payments" >
          < ScalarProperty Name="Id" ColumnName="Id" / >
          < ScalarProperty Name="BankName" ColumnName="BankName" / >
          < Condition ColumnName="Type" Value="CHEQUE" / >
        < /MappingFragment >
      < /EntityTypeMapping >
    < /EntitySetMapping >
  < /EntityContainerMapping >
< /Mapping >
  
```

现在，可以迭代 Payments 表中的数据，根据映射返回不同的类型：

```
PaymentEntities data = new PaymentEntities();
foreach (var p in data.Payments)
{
    Console.WriteLine("{0}, {1} - {2:C}", p.GetType().Name,
        p.Name, p.Amount);
}
```

运行应用程序，会从数据库中返回两个 CashPayment 和一个 CreditCardPayment 对象：

```
CreditCardPayment, Gustav - $22.00
CashPayment, Donald - $0.50
CashPayment, Dagobert - $80,000.00
```

## A.5.2 Table per Type

在 TPT 中，一个表仅映射一个类型。Northwind 数据库的模式是表 Customers、Orders 和 Order Details(如图 A-5 所示)。表 Orders 用外键 CustomerID 映射表 Customers，表 Order Details 用外键 OrderID 映射表 Orders。

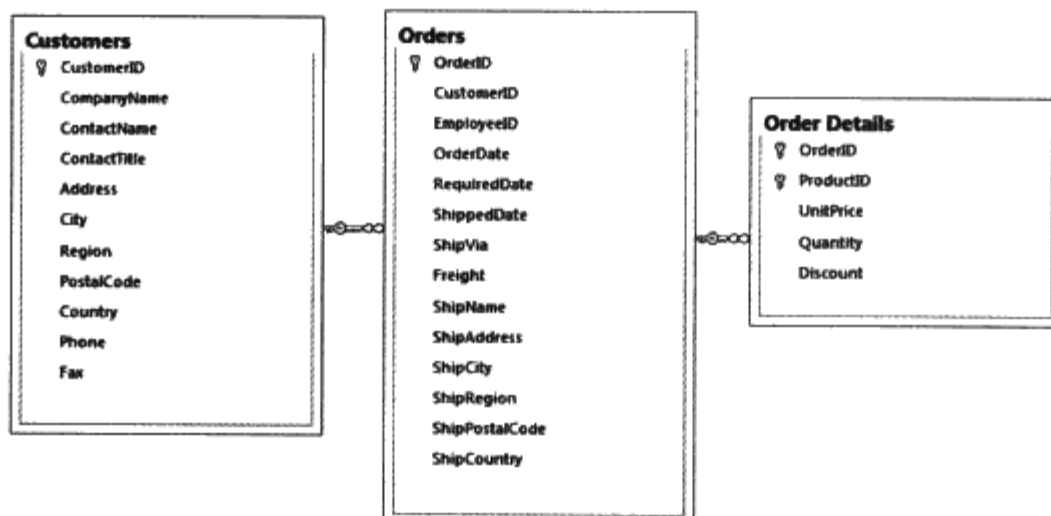


图 A-5

图 A-6 显示了实体类型 Customer、Order 和 OrderDetail。Customer 和 Order 有 0 对多或一对多的关系，Order 和 OrderDetail 有一对多关系。Customer 和 Order 有 0 对多或一对多的关系，是因为在数据库模式中，Orders 表中的 CustomerID 定义为 Nullable。

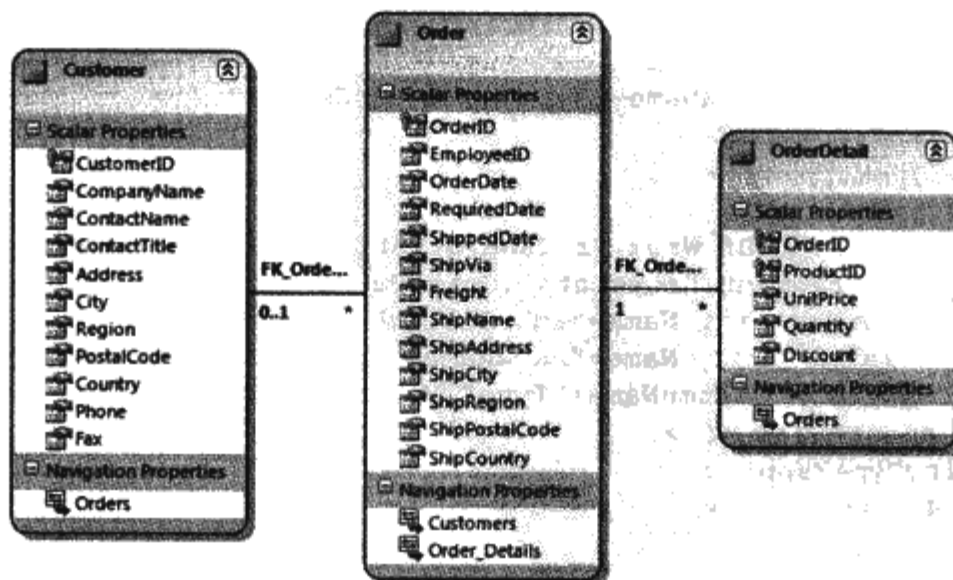


图 A-6



用两个迭代访问顾客及其订单。首先访问 Customer 对象，把 CompanyName 属性的值写入控制台。接着使用 Customer 类的 Orders 属性访问所有的订单。因为相关的订单默认没有加载到对象环境中，所以从 Orders 属性中返回 EntityCollection<Order>对象的 Load()方法。

```
NorthwindEntities data = new NorthwindEntities();
foreach (Customer customer in data.Customers)
{
    Console.WriteLine("{0}", customer.CompanyName);
    if (!customer.Orders.IsLoaded)
        customer.Orders.Load();
    foreach (Order order in customer.Orders)
    {
        Console.WriteLine("{0} {1:d}", order.OrderID, order.OrderDate);
    }
}
```

在后台，使用 RelationshipManager 类访问关系。把实体对象的类型强制转换为接口 IEntityWithRelationships，就可以访问 RelationshipManager 实例。这个接口由类 EntityObject 显式实现。RelationshipManager 属性返回一个 RelationshipManager，它与一端的实体对象关联起来。另一端调用方法 GetRelatedCollection() 来定义。第一个参数 NorthwindModel.FK\_Orders\_Customers 是关系的名称，第二个参数 Orders 定义了目标角色的名称。

```
RelationshipManager rm =
    ((IEntityWithRelationships)customer).RelationshipManager;
EntityCollection < Order > orders =
    rm.GetRelatedCollection < Order > (
        "NorthwindModel.FK_Orders_Customers", "Orders");
```

关系的加载延迟了。EntityCollection 类的 Load()方法从数据库中获取数据。Load()方法的一个重载版本接受 MergeOption 枚举。该枚举的值如表 A-4 所示。

提示：

默认情况下，关系都是延迟加载的。例如，如果定义了 Customers 和 Orders 表之间的关系，并需要查询顾客，就不加载该顾客的 Orders 记录。这里使用的术语是“延迟加载”，因为订单可以在需要时加载。与延迟加载相反，也可以“即时提取(eager fetch)”记录，这表示在访问顾客记录时，也加载该顾客的订单。

表 A-4

MergeOption 值	说 明
AppendOnly	这是默认值。追加新实体，对象环境中的已有实体不修改
NoTracking	不修改 ObjectStateManager，它跟踪实体对象的变化
OverwriteChanges	实体对象的当前值用数据库中的值替代
PreserveChanges	实体对象在对象环境中的初始值用数据库中的值替代

A.6 对象查询

查询对象是 ADO.NET Entity Framework 提供的一个服务。查询可以使用 LINQ to Entities、

Entity SQL 和创建 Entity SQL 的 Query Builder 方法来进行。LINQ to Entities 在本附录的后面介绍，这里先介绍其他两个选项。

下面几节使用了 Formula 1 数据库，其中包含从设计器中创建的实体，如图 A-7 所示。

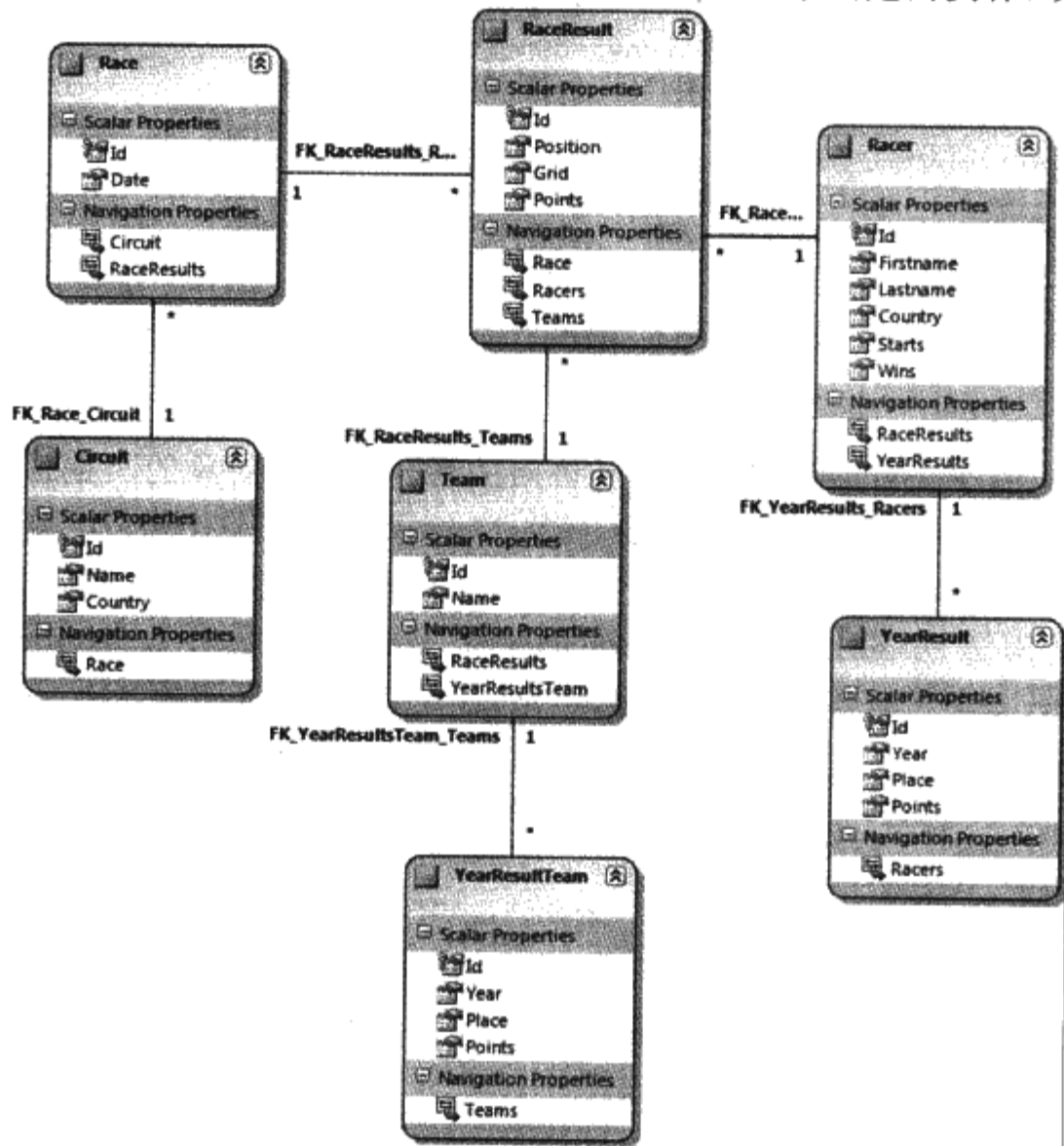


图 A-7

查询可以用 `ObjectQuery<T>` 类定义。下面先用一个简单的查询访问所有的 `Racer` 实体。在这个例子中，连接已经打开，所以给对象环境 `Formula1Entities` 传送该连接。这样就可以用 `ToTraceString()` 方法检索给 `ObjectQuery<Racer>` 类生成的 SQL 语句。这个方法需要一个打开的连接。

```
ConnectionStringSettings connSettings =
    ConfigurationManager.ConnectionStrings["Formula1Entities"];
EntityConnection connection =
    new EntityConnection(connSettings.ConnectionString);
connection.Open();
using (Formula1Entities data = new Formula1Entities(connection))
{
    ObjectQuery < Racer > racers = data.Racers;
    Console.WriteLine(racers.CommandText);
    Console.WriteLine(racers.ToTraceString());
    connection.Close();
}
```

从 `CommandText` 属性中返回的 Entity SQL 语句如下：

```
[Racers]
```

这个生成的 SELECT 语句从 ToTraceString() 方法显示的数据库中检索记录:

```
SELECT
  [Extent1].[Id] AS [Id],
  [Extent1].[Firstname] AS [Firstname],
  [Extent1].[Lastname] AS [Lastname],
  [Extent1].[Country] AS [Country],
  [Extent1].[Starts] AS [Starts],
  [Extent1].[Wins] AS [Wins]
FROM [dbo].[Racers] AS [Extent1]
```

除了在对象环境中访问 Racers 属性之外, 还可以用 CreateQuery() 方法创建一个查询:

```
ObjectQuery < Racer > racers = data.CreateQuery < Racer > ("[Racers]");
```

这类似于使用 Racers 属性, 实际上, Racers 属性的实现代码会以这种方式创建一个查询。

下面根据条件过滤赛手。这可以使用 ObjectQuery<T> 类的 Where() 方法来完成。Where() 方法是一个 Query Builder 方法, 它可以创建 Entity SQL 语句。这个方法需要把一个谓词作为字符串, 其可选参数的类型是 ObjectParameter。这里的谓词指定只返回来自巴西的赛手。it 指定结果项, Country 是列 Country。ObjectParameter 构造函数的第一个参数引用谓词的 @Country 参数, 但不带 @ 符号。

```
string country = "Brazil";
ObjectQuery < Racer > racers = data.Racers.Where(
    "it.Country = @Country",
    new ObjectParameter("Country", country));
```

it 的作用可以通过访问查询的 CommandText 属性来了解。在 Entity SQL 中, SELECT VALUE it 声明 it 访问列。

```
SELECT VALUE it
FROM (
  [Racers]
) AS it
WHERE
  it.Country = @Country
```

方法 ToTraceString() 显示生成的 SQL 语句:

```
SELECT
  [Extent1].[Id] AS [Id],
  [Extent1].[Firstname] AS [Firstname],
  [Extent1].[Lastname] AS [Lastname],
  [Extent1].[Country] AS [Country],
  [Extent1].[Starts] AS [Starts],
  [Extent1].[Wins] AS [Wins]
FROM [dbo].[Racers] AS [Extent1]
WHERE [Extent1].[Country] = @Country
```

当然, 也可以指定完整的 Entity SQL:

```
string country = "Brazil";
ObjectQuery < Racer > racers = data.CreateQuery < Racer > (
    "SELECT VALUE it FROM ([Racers]) AS it WHERE it.Country = @Country",
    new ObjectParameter("Country", country));
```

类 `ObjectQuery<T>` 提供了几个 Query Builder 方法，如表 A-5 所述。其中的许多方法非常类似于第 11 章介绍的 LINQ 扩展方法。

表 A-5

ObjectQuery<T>类的 Query Builder 方法	说 明
Where()	这个方法可以根据条件过滤结果
Distinct()	这个方法创建唯一结果的查询
Except()	这个方法返回的结果不满足 except 过滤器指定的条件
GroupBy()	这个方法创建一个新查询，根据指定的条件组合实体
Include()	相关的项是延迟加载的，所以需要调用 EntityCollection<T>类的 Load()方法，把相关的实体放在对象环境中。如果不使用 Load()方法，还可以用 Include()方法指定一个查询，即时提取相关的实体
OfType()	这个方法指定只返回特定类型的实体，使用 TPH 关系很有帮助
OrderBy()	这个方法定义实体的排列顺序
Select() Selectvalue()	这些方法返回结果的一个投射。Select()以 DbDataRecord 的形式返回结果项；SelectValue()根据泛型参数 TResultType，把值返回为标量类型或复杂类型
Skip() Top()	这些方法可用于分页。Skip()可以跳过某个数量的项，提取 Top()方法指定数量的项
Intersect() Union() UnionAll()	这些方法用于合并两个查询。Intersect()返回的查询只包含两个查询都有的结果。Union()合并查询，返回没有重复的完整结果，UnionAll()则包含重复的结果

下面用一个例子说明如何使用这些 Query Builder 方法。其中，赛手用 `Where()` 方法过滤，只返回来自美国的赛手；`OrderBy()` 方法指定先根据获奖者人数进行降序排列，再根据参加比赛的人数进行降序排列。最后，使用 `Top()` 方法仅在结果中显示前 3 个赛手：

```
using (FormulalEntities data = new FormulalEntities())
{
    string country = "USA";
    ObjectQuery < Racer > racers = data.Racers.Where("it.Country = @Country",
        new ObjectParameter("Country", country))
        .OrderBy("it.Wins DESC, it.Starts DESC")
        .Top("3");
    foreach (var racer in racers)
    {
        Console.WriteLine("{0} {1}, wins: {2}, starts: {3}",
            racer.Firstname, racer.Lastname, racer.Wins, racer.Starts);
    }
}
```

这个查询的结果如下：

```
Mario Andretti, wins: 12, starts: 128
Dan Gurney, wins: 4, starts: 87
Phil Hill, wins: 3, starts: 48
```

## A.7 更新

读取、搜索和过滤数据库中的数据仅是数据密集型应用程序通常需要执行的一部分操作。把改变的数据写回数据库是另一个要执行的操作。

下面几节介绍如下主题：

- 对象跟踪
- 改变信息
- 与实体建立关联和解除关联
- 存储实体的变化

### A.7.1 对象跟踪

为了修改和保存从数据库中读取的数据，必须在加载实体后跟踪它们。这也要求对象环境注意实体是否已从数据库中加载了。如果多个查询同时访问同一个记录，对象环境就需要返回已经加载的实体。

`ObjectStateManager` 由对象环境用于跟踪加载到环境中的实体。

下面的示例演示了，如果两个不同的查询从数据库中返回相同的记录，状态管理器就会注意到这一点，因此不创建新实体，而是返回相同的实体。与对象环境关联的 `ObjectStateManager` 实例可以用 `ObjectStateManager` 属性访问。`ObjectStateManager` 类定义了 `ObjectStateManagerChanged` 事件，每次在对象环境中添加或删除对象时，就触发这个事件。这里，把 `ObjectStateManager_ObjectStateManagerChanged` 方法赋予该事件，以获得改变的信息。

两个不同的查询用于返回一个实体对象。第一个查询获得来自奥地利、姓氏为 `Lauda` 的第一个赛手。第二个查询请求来自奥地利的赛手，按照赢得比赛的次数排列赛手，并获取第一个结果。事实上，这是同一个赛手。为了验证返回了同一个实体对象，使用 `Object.ReferenceEquals()` 方法验证两个对象引用是否指向同一个实例。

```
static void Tracking()
{
    using (Formula1Entities data = new Formula1Entities())
    {
        data.ObjectStateManager.ObjectStateManagerChanged +=
            ObjectStateManager_ObjectStateManagerChanged;
        Racer nikil = data.Racers.Where(
            "it.Country='Austria' & & it.Lastname='Lauda'").First();
        Racer niki2 = data.Racers.Where("it.Country='Austria'")
            .OrderBy("it.Wins DESC").First();
        if (Object.ReferenceEquals(nikil, niki2))
        {
            Console.WriteLine("the same object");
        }
    }
}

static void ObjectStateManager_ObjectStateManagerChanged(object sender,
    CollectionChangeEventArgs e)
{
    Console.WriteLine("Object State change - action: {0}", e.Action);
    Racer r = e.Element as Racer;
```



```

    if (r != null)
        Console.WriteLine("Racer {0}", r.Lastname);
}

```

运行这个应用程序，会看到 `ObjectStateManager` 的 `ObjectStateManagerChanged` 事件只触发了一次，引用 `niki1` 和 `niki2` 是相同的：

```

Object state change - action: Add
Racer Lauda
The same object

```

## A.7.2 改变信息

对象环境也会注意到实体的改变。下面的示例添加并修改对象环境中的一个赛车手，并获得修改的信息。首先，使用 `Formula1Entities` 类的 `AddToRacers()` 方法添加一个新赛车手，这个设计器生成的方法调用基类 `ObjectContext` 的 `AddObject()` 方法。`AddObject()` 方法用 `EntityState.Added` 信息添加一个新实体。接着查询 `Lastname` 为 `Alonso` 的赛车手。在这个实体类中，递增 `Starts` 属性，用 `EntityState.Modified` 信息标记实体。在后台，通知 `ObjectStateManager`：实现了接口 `INotifyPropertyChanged` 的对象有了状态改变。这个接口在实体基类 `StructuralObject` 中实现。`ObjectStateManager` 关联到 `PropertyChanged` 事件上，这个事件会因每个属性改变而触发。

为了获得所有添加或修改的实体对象，可以调用 `ObjectStateManager` 的 `GetObjectStateEntries()` 方法，并传送一个 `EntityState` 枚举值。这个方法返回一个 `ObjectStateEntry` 对象集合，其中保存了实体的信息。帮助方法 `DisplayState` 迭代这个集合，获得详细信息。

也可以把 `EntityKey` 传送给 `GetObjectStateEntry()` 方法，获得单个实体的状态信息。`EntityKey` 属性可以用实现了 `IEntityWithKey` 接口的实体对象来获得，即派生自基类 `EntityObject` 的实体对象。返回的 `ObjectStateEntry` 对象提供了方法 `GetModifiedProperties()`，在该方法中，可以读取已改变的所有属性值，也可以用 `OriginalValues` 和 `CurrentValues` 索引器访问属性的初始值和当前信息。

```

static void ChangeInformation()
{
    using (Formula1Entities data = new Formula1Entities())
    {
        Racer sebastien = new Racer()
        {
            Firstname = "Sébastien",
            Lastname = "Bourdais",
            Country = "France",
            Starts = 0
        };
        data.AddToRacers(sebastien);
        Racer fernando = data.Racers.Where("it.Lastname='Alonso']").First();
        fernando.Starts++;
        DisplayState(EntityState.Added.ToString(),
            data.ObjectStateManager.GetObjectStateEntries(
                EntityState.Added));
        DisplayState(EntityState.Modified.ToString(),
            data.ObjectStateManager.GetObjectStateEntries(
                EntityState.Modified));
        ObjectStateEntry stateOfFernando =
            data.ObjectStateManager.GetObjectStateEntry(fernando.EntityKey);
        Console.WriteLine("state of Fernando: {0}",
            stateOfFernando.State.ToString());
    }
}

```

```

foreach (string modifiedProp in
stateOfFernando.GetModifiedProperties())
{
    Console.WriteLine("modified: {0}", modifiedProp);
    Console.WriteLine("original: {0}",
        stateOfFernando.OriginalValues[modifiedProp]);
    Console.WriteLine("current: {0}",
        stateOfFernando.CurrentValues[modifiedProp]);
}
}
static void DisplayState(string state,
    IEnumerable < ObjectStateEntry > entries)
{
    foreach (var entry in entries)
    {
        Racer r = entry.Entity as Racer;
        if (r != null)
        {
            Console.WriteLine("{0}: {1}", state, r.Lastname);
        }
    }
}
}

```

运行这个应用程序，会显示添加和修改了的赛手，改变了的属性的初始值和当前值：

```

Added: Bourdais
Modified: Alonso
state of Fernando: Modified
modified: Starts
original: 95
current: 96

```

### A.7.3 与实体建立关联和解除关联

把实体数据返回给调用者，对于在对象环境中解除对象的关联是很重要的。例如，如果实体对象从 Web 服务中返回，这就是必须的。这里，如果实体对象在客户机上改变了，对象环境并不知道。

在示例代码中，ObjectContext 类的 Detach()方法解除了实体 fernando 的关联，因此对象环境不知道对这个实体进行了什么修改。如果改变了的实体对象从客户机应用程序传送给服务，可以再次关联它。把它关联到对象环境上还不够，因为这并没有给出信息，说明这个对象已经修改了。而最初的对象必须在对象环境中可用。最初的对象可以使用 GetObjectByKey()方法和键在数据库中访问。如果实体对象已经在对象环境中，就使用已有的实体，否则就从数据库中提取新实体。调用方法 ApplyPropertyChanges()，把修改过的实体对象传送给对象环境，如果实体对象有变化，就在已有的实体中用对象环境中的同一个键进行这个修改，再把 EntityState 设置为 Modified。方法 ApplyPropertyChanges()需要对象存在于对象环境中，否则就用 EntityState.Added 添加新实体对象。

```

using (FormulalEntities data = new FormulalEntities())
{
    data.ObjectStateManager.ObjectStateManagerChanged +=
        ObjectStateManager_ObjectStateManagerChanged;
    ObjectResult < Racer > racers = data.Racers.Where("it.Lastname='Alonso'");
    Racer fernando = racers.First();
    EntityKey key = fernando.EntityKey;
}

```

```

data.Detach(fernando);
// Racer is now detached and can be changed independent of the
// object context
fernando.Starts++;
Racer originalObject = (Racer)data.GetObjectByKey(key);
data.ApplyPropertyChanges(key.EntitySetName, fernando);
}

```

#### A.7.4 存储实体的变化

在 `ObjectStateManager` 的帮助下, 根据所有的变化信息, 可以使用 `ObjectContext` 类的 `SaveChanges()` 方法, 把添加、删除和修改的实体对象写到数据库中。要验证对象环境中的变化, 可以把一个处理程序赋予 `ObjectContext` 类的 `SaveChanges` 事件。这个事件在数据写入数据库之前触发, 所以可以添加一些验证逻辑, 确定是否应进行修改。`SaveChanges()` 方法返回写入的实体对象数。

如果数据库中用实体类表示的记录在读取之后发生了变化, 该怎么办? 答案取决于用模型设置的 `ConcurrencyMode` 属性。在实体对象的每个属性中, 都可以把 `ConcurrencyMode` 配置为 `Fixed` 或 `None`。值 `Fixed` 表示属性在写入时验证, 以确定值是否没有变化。默认值 `None` 表示忽略任何改变。如果某些属性配置为 `Fixed` 模式, 且在读写实体对象时数据有变化, 就抛出 `OptimisticConcurrencyException` 异常。调用 `Refresh()` 方法把数据库中的信息读入对象环境, 就可以处理这个异常。这个方法接收 `RefreshMode` 枚举值配置的两个刷新模式: `ClientWins` 和 `StoreWins`。`StoreWins` 表示从数据库中提取信息, 并设置为实体对象的当前值。`ClientWins` 表示数据库信息设置为实体对象的初始值, 因此数据库值在下一个 `SaveChanges` 事件中被覆盖。`Refresh()` 方法的第二个参数是实体对象的集合或单个实体对象。可以为每个实体确定刷新操作:

```

static void ChangeInformation()
{
    //...
    int changes = 0;
    try
    {
        changes = data.SaveChanges();
    }
    catch (OptimisticConcurrencyException ex)
    {
        data.Refresh(RefreshMode.ClientWins, ex.StateEntries);
        changes = data.SaveChanges();
    }
    Console.WriteLine("{0} entities changed", changes);
}

```

## A.8 LINQ to Entities

本书的几章介绍了 LINQ to Query 对象、数据库和 XML。当然, LINQ 也能用于查询实体。

在 LINQ to Entities 中, LINQ 查询的数据源是 `ObjectQuery<T>`。因为 `ObjectQuery<T>` 实现了接口 `IQueryable`, 所以用于查询的扩展方法是用 `System.Linq` 命名空间中的 `Queryable` 类定义的。用这个类定义的扩展方法有一个参数 `Expression<T>`, 这就是编译器把表达式树写入程序集的原因。第 11 章介绍了表达式树, 表达式树是从 `ObjectQuery<T>` 类中解析到 SQL 查询中的。

可以使用如下简单的 LINQ 查询返回赢得超过 40 场比赛的赛手:

```
using (FormulalEntities data = new FormulalEntities())
{
    var racers = from r in data.Racers
        where r.Wins > 40
        orderby r.Wins descending
        select r;
    foreach (Racer r in racers)
    {
        Console.WriteLine("{0} {1}", r.Firstname, r.Lastname);
    }
}
```

访问 Formula 1 数据库的结果如下:

```
Michael Schumacher
Alain Prost
Ayrton Senna
```

还可以定义一个 LINQ 查询访问关系, 如下所示。变量 *r* 表示赛手, 变量 *rr* 表示所有的比赛结果。过滤器用 **where** 子句定义, 只检索榜上有名的瑞士赛手。要完成这个排行榜, 应组合结果, 并计算排行榜的个数。排序根据排行榜的完成情况进行:

```
using (FormulalEntities data = new FormulalEntities())
{
    var query = from r in data.Racers
        from rr in r.RaceResults
        where rr.Position <= 3 && rr.Position >= 1 &&
            r.Country == "Switzerland"
        group r by r.Id into g
        let podium = g.Count()
        orderby podium descending
        select new { Racer = g.FirstOrDefault(), Podiums = podium };
    foreach (var r in query)
    {
        Console.WriteLine("{0} {1} {2}", r.Racer.Firstname,
            r.Racer.Lastname, r.Podiums);
    }
}
```

运行这个应用程序, 会返回瑞士的 3 个赛手姓名:

```
Clay Regazzoni 28
Jo Siffert 6
Rudi Fischer 2
```

## A.9 小结

本章介绍了 ADO.NET Entity Framework 的特性, 与第 27 章介绍的 LINQ to SQL 不同, 这个框架提供了一个基于提供程序的映射, 其他应用程序供应商也可以实现它们自己的提供程序。

ADO.NET Entity Framework 基于 CSDL、MSL 和 SSDL 定义的映射, 这些 XML 信息描述了实体、映射和数据库模式。使用这个映射技术, 可以创建不同的关系类型, 把实体类映射到数据库表上。

本章还介绍了对象环境如何保存所检索和更新的实体信息, 如何把改变的内容写入数据库。

LINQ to Entities 仅是 ADO.NET Entity Framework 的一个部分, 它允许使用新的查询语法访问实体。

## C#、Visual Basic 和 C++/CLI

C#是专为.NET 设计的编程语言。编写.NET 应用程序可以使用 50 多种语言，例如 Eiffel、Smalltalk、COBOL、Haskell、Pizza、Pascal、Delphi、Oberon、Prolog、Ruby 等。Microsoft 还发布了 C#、Visual Basic、C++/CLI、J#和 JScript.NET。

每种语言都有优缺点；一些任务很容易用一种语言完成，但用另一种语言完成就很复杂。.NET Framework 中的类总是相同的，但语言的语法从.NET Framework 中抽象出了各种特性。例如，C#的 using 语句很便于使用实现了 IDisposable 接口的对象。其他语言要实现该功能就需要较多代码。

最常用的.NET 语言是 C#和 Visual Basic。C#是专为.NET 设计的新语言，其理念来自于 C++、Java、Pascal 和其他语言。Visual Basic 植根于 Visual Basic 6，用.NET 的面向对象特性进行了扩展。

C++/CLI 是 C++的一种扩展，基于 ECMA 标准(ECMA 372)。C++/CLI 的一大优点是可以将内置代码与托管代码混合起来。我们可以扩展已有的 C++应用程序，添加.NET 功能，将.NET 类添加到内置库中，使它们能用于其他.NET 语言，如 C#。还可以用 C++/CLI 编写完全托管的应用程序。

本章介绍如何将.NET 应用程序从一种语言转换为另一种语言。Visual Basic 或 C++/CLI 示例代码可以映射为 C#，反之亦然。

本章的主要内容如下：

- 命名空间
- 定义类型
- 方法
- 数组
- 控制语句
- 循环
- 异常处理
- 继承
- 资源管理
- 委托
- 事件
- 泛型
- LINQ 查询
- C++/CLI 混合内置代码和托管代码



提示:

要理解本章的内容,读者应知道 C#,并已阅读了本书的前几章。但不必了解 Visual Basic 和 C++/CLI。

## B.1 命名空间

.NET 类型组织到命名空间中。这三种语言定义和使用命名空间的语法完全不同。

为了导入命名空间,C#使用 `using` 关键字。C++/CLI 完全基于 C++语法,使用 `using namespace` 语句, Visual Basic 指定用 `Imports` 关键字导入命名空间。

在 C#中,可以给类或其他命名空间指定别名。在 C++/CLI 和 Visual Basic 命名空间中,别名只能引用其他命名空间,不能引用类。C++需要用 `namespace` 关键字来定义别名,这个关键字也用于定义命名空间。Visual Basic 再次使用 `Imports` 关键字。

要定义命名空间,这三种语言都使用了 `namespace` 关键字,但有一个区别。在 C++/CLI 中,不能用一个命名空间语句定义带层次结构的命名空间,而必须嵌套命名空间。项目设置有一个重要的区别:在 C#的项目设置中定义命名空间,就是定义默认的命名空间,该命名空间会显示在添加到项目中的所有新项的代码中。在 Visual Basic 的项目设置中,定义的是由项目中所有项使用的根命名空间。在源代码中声明的命名空间只定义了根命名空间中的子命名空间。

```
// C#
using System;
using System.Collections.Generic;
using Assm = Wrox.ProCSharp.Assemblies;

namespace Wrox.ProCSharp.Languages
{
}
```

```
// C++/CLI
using namespace System;
using namespace System::Collections::Generic;
namespace Assm = Wrox.ProCSharp.Assemblies;

namespace Wrox
{
    namespace ProCSharp
    {
        namespace Languages
        {
        }
    }
}
```

```
'Visual Basic
Imports System
Imports System.Collections.Generic
Imports Assm = Wrox.ProCSharp.Assemblies

Namespace Wrox.ProCSharp.Languages
End Namespace
```

## B.2 定义类型

.NET 区分了引用类型和值类型。在 C# 中，引用类型用类定义，值类型用结构定义。除了创建引用类型和值类型的方式之外，本章还介绍如何定义接口(引用类型)和枚举(值类型)。

### B.2.1 引用类型

要声明引用类型，C# 和 Visual Basic 使用 `class` 关键字。在 C++/CLI 中，类和结构基本相同，不能像 C# 和 Visual Basic 那样区分引用类型和值类型。C++/CLI 有一个 `ref` 关键字，它定义了托管类。定义 `ref class` 或 `ref struct` 可以创建引用类型。

在 C# 和 C++/CLI 中，类用花括号括起来。C++/CLI 要求在类声明的最后加上分号。Visual Basic 在类的最后使用 `End Class` 语句。

```
// C#
public class MyClass
{
}

// C++/CLI
public ref class MyClass
{
};

public ref struct MyClass2
{
};
```

```
'Visual Basic
Public Class MyClass
End Class
```

在使用引用类型时，需要声明一个变量，该对象必须分配托管堆上的空间。在声明引用类型的句柄时，C++/CLI 会定义句柄操作符`^`，它有点类似于 C++ 指针`*`。`gcnew` 操作符分配托管堆上的空间。使用 C++/CLI 还可以在本地声明变量，但对于引用类型，该对象仍分配托管堆上的空间。在 Visual Basic 中，变量声明以 `Dim` 语句开头，其后是变量名。对于 `new` 和对象类型，需要分配托管堆上的空间。

```
// C#
MyClass obj = new MyClass();
```

```
// C++/CLI
MyClass^ obj = gcnew MyClass();

MyClass obj2;
```

```
'Visual Basic
Dim obj as New MyClass()
```

如果引用类型没有引用内存，这三种语言都使用其他关键字：C# 定义了 `null` 字面量，C++/CLI 使用 `nullptr`(`NULL` 仅对内置对象有效)，Visual Basic 使用 `Nothing`。

表 B-1 列出了预定义的引用类型。C++/CLI 没有像其他两种语言那样定义对象和字符串类

型。当然，可以使用 Framework 定义的类。

表 B-1

.NET 类型	C#	C++/CLI	Visual Basic
System.Object	Object	未定义	Object
System.String	String	未定义	String

B.2.2 值类型

要声明值类型，C#使用 struct 关键字；C++/CLI使用 value 关键字，Visual Basic 使用 Structure。

```
// C#
public struct MyStruct
{
}
```

```
// C++/CLI
public value class MyStruct
{
};
```

```
'Visual Basic
Public Structure MyStruct
End Structure
```

在 C++/CLI 中，可以把值类型分配到堆栈上、内置堆上(使用 new 操作符)或托管堆上(使用 gcnew 操作符)。C#和 Visual Basic 没有这些选项，但用 C++/CLI 混合内置代码和托管代码时，这些选项就变得非常重要了。

```
// C#
MyStruct ms;
```

```
// C++/CLI
MyStruct ms1;
MyStruct* pms2 = new MyStruct();
MyStruct^ hms3 = gcnew MyStruct();
```

```
'Visual Basic
Dim ms as MyStruct
```

表 B-2 列出了这三种语言中预定义的值类型。在 C++/CLI 中，char 类型的大小为 1 字节，用于存储 ASCII 字符。在 C#中，char 类型的大小为 2 字节，用于存储 Unicode 字符，而 C++/CLI 为此使用了 wchar\_t 类型。C++的 ANSI 标准只定义了 short <= int <= long。在 32 位机器上，int 和 long 的大小都是 32 位。要在 C++中定义 64 位变量，需要使用 long long。

表 B-2

.NET 类型	C#	C++/CLI	Visual Basic	大小
Char	char	wchar_t	Char	2 字节
Boolean	bool	bool	Boolean	1 字节，包含 true 或 false

(续表)

.NET 类型	C#	C++/CLI	Visual Basic	大小
Int16	short	short	Short	2 字节
UInt16	ushort	unsigned short	Ushort	2 字节, 无符号
Int32	int	int	Integer	4 字节
UInt32	uint	unsigned Int	UInteger	4 字节, 无符号
Int64	long	long long	Long	8 字节
UInt64	ulong	unsigned long long	ULong	8 字节, 无符号

B.2.3 类型推断

C# 3.0 允许在定义本地变量时，不显式声明数据类型，而使用 var 关键字。类型是从指定的初始值中推断出来的。在 VB 中，只要打开了 Option infer，就可以使用 Dim 关键字推断变量的类型。这个功能需要使用编译器设置 /optioninfer+，或者在 VS 中使用项目配置页面：

```
// C#
var x = 3;
' Visual Basic
Dim x = 3
```

B.2.4 接口

这三种语言在定义接口方面非常类似，它们都使用关键字 interface：

```
// C#
public interface IDisplay
{
    void Display();
}

// C++/CLI
public interface class IDisplay
{
    void Display();
};

'Visual Basic
Public Interface IDisplay
    Sub Display
End Interface
```

实现接口的方式则不同。C#和 C++/CLI 在类名后面加上冒号，之后是接口名，并实现用该接口定义的方法。在 C++/CLI 中，方法必须声明为 virtual。Visual Basic 使用 Implements 关键字来实现接口，接口定义的方法也需要加上 Implements 关键字。

```
// C#
public class Person : IDisplay
{
```



```

public:
    void Display()
    {
    }
}
// C# explicit interface implementation
public class Person : IDisplay
{
    void IDisplay.Display()
    {
    }
}

```

```

// C++/CLI
public ref class Person : IDisplay
{
public:
    virtual void Display();
};

```

```

'Visual Basic
Public Class Person
    Implements IDisplay

    Public Sub Display Implements IDisplay.Display
    End Sub
End Class

```

### B.2.5 枚举

这三种语言在定义枚举方面非常类似，它们都使用关键字 `enum` (只是 Visual Basic 使用新的一行代码，而不是用逗号分隔元素)。

```

// C#
public enum Color
{
    Red, Green, Blue
}

```

```

// C++/CLI
public enum class Color
{
    Red, Green, Blue
};

```

```

'Visual Basic
Public Enum Color
    Red
    Green
    Blue
End Enum

```

## B.3 方法

方法总是在类中声明。C++/CLI 的语法非常类似于 C#。只是访问修饰符不包含在方法声明中，而是写在方法声明之前。访问修饰符必须用冒号结束。在 Visual Basic 中，使用 `Sub` 关键



字定义方法。

```
// C#
public class MyClass
{
    public void Foo()
    {
    }
}
```

```
// C++/CLI
public ref class MyClass
{
public:
    void Foo()
    {
    }
};
```

```
'Visual Basic
Public Class MyClass
    Public Sub Foo
    End Sub
End Class
```

### B.3.1 方法的参数和返回类型

在 C# 和 C++/CLI 中，传送给方法的参数在括号中定义。参数的类型在变量名的前面声明。如果从方法中返回一个值，该方法就用返回值的类型定义，而不是 `void`。

Visual Basic 使用 `Sub` 语句声明没有返回值的方法，用 `Function` 语句声明有返回类型的方法。返回类型放在方法名和括号的后面。参数中的变量声明和类型在 Visual Basic 中的顺序也不同：类型在变量的后面，而在 C# 和 C++/CLI 中，变量在类型的后面。

```
// C#
public class MyClass
{
    public int Foo(int i)
    {
        return 2 * i;
    }
}
```

```
// C++/CLI
public ref class MyClass
{
public:
    int Foo(int i)
    {
        return 2 * i;
    }
};
```

```
'Visual Basic
Public Class MyClass
    Public Sub Foo1(ByVal i as Integer)
    End Sub
```

```

    Public Function Foo(ByVal i As Integer) As Integer
        Return 2 * i
    End Sub
End Class

```

### B.3.2 参数的修饰符

在默认情况下，值类型按值传送，引用类型按引用传送。如果传送为参数的值类型要在调用方法中修改，C#允许使用参数修饰符 `ref`。

C++/CLI 定义了一个托管的引用操作符 `%`。这个操作符类似于 C++ 引用操作符 `&`，但 `%` 可以用于托管类型，垃圾收集器可以跟踪这些对象，以防它们移动到托管堆中。

Visual Basic 使用 `ByRef` 关键字按引用传送参数。

```

// C#
public class ParameterPassing
{
    public void ChangeVal(ref int i)
    {
        i = 3;
    }
}

```

```

// C++/CLI
public ref class ParameterPassing
{
public:
    int ChangeVal(int% i)
    {
        i = 3;
    }
};

```

```

'Visual Basic
Public Class ParameterPassing
    Public Sub ChangeVal(ByRef i as Integer)
        i = 3
    End Sub
End Class

```

调用带引用参数的方法时，只有 C# 语言需要使用参数修饰符。C++/CLI 和 Visual Basic 在调用带或不带参数修饰符的方法方面没有区别。这方面 C# 是有优势的，因为可以立即看出，参数值可以在调用方法中修改。

由于调用语法没有区别，因此 Visual Basic 不允许在重载方法时仅改变修饰符。C++/CLI 编译器允许在重载方法时仅改变修饰符，但不能编译调用方法，因为这个方法是模糊的。在 C# 中，允许重载并使用只有参数修饰符不同的方法，但这不是一个好的编程习惯。

```

// C#
ParameterPassing obj = new ParameterPassing();
int a = 1;
obj.ChangeVal(ref a);
Console.WriteLine(a); // writes 3

```

```

// C++/CLI
ParameterPassing obj;

```

```
int a = 1;
obj.ChangeVal(a);
Console.WriteLine(a); // writes 3
```

```
' Visual Basic
Dim obj as new ParameterPassing()
Dim i as Integer = 1
obj.ChangeVal(i)
Console.WriteLine(i) // writes 3
```

#### 提示:

当一个参数从方法中返回时, C#还定义了 `out` 关键字。C++/CLI 和 Visual Basic 都没有这个选项。只要调用方法和被调用的方法在相同的应用程序域中, `out` 和 `ref` 在后台就没有区别。用 C# `out` 参数修饰符声明的方法也可以在 C++/CLI 和 Visual Basic 中以与 `ref` 参数修饰符相同的方式调用。如果方法要在多个应用程序域或进程中使用, C++/CLI 和 Visual Basic 就使用属性 `[out]`。

### B.3.3 构造函数

在 C# 和 C++/CLI 中, 构造函数与类同名。Visual Basic 使用 `New` 过程。`this` 和 `Me` 关键字用于访问这个实例的成员。在一个构造函数中调用另一个构造函数时, C# 需要初始化一个成员。C++/CLI 和 Visual Basic 可以将构造函数调用为方法。

```
// C#
public class Person
{
    public Person()
        : this("unknown", "unknown")
    { }

    public Person(string firstname, string lastname)
    {
        this.firstname = firstname;
        this.lastname = lastname;
    }

    private string firstname;
    private string lastname;
}
```

```
// C++/CLI
public ref class Person
{
public:
    Person()
    {
        Person("unknown", "unknown");
    }

    Person(String^ firstname, String^ lastname)
    {
        this->firstname = firstname;
        this->lastname = lastname;
    }
private:
    String^ firstname;
    String^ lastname;
}
```



```
};

'Visual Basic
Public Class Person
    Public Sub New()
        Me.New("unknown", "unknown")
    End Sub

    Public Sub New(ByVal firstname As String, ByVal lastname As String)
        Me.MyFirstname = firstname
        Me.MyLastname = lastname
    End Sub

    Private MyFirstname As String
    Private MyLastname As String
End Class
```

### B.3.4 属性

要定义属性，C#只需在属性块中编写 `get` 和 `set` 存取器。在 `set` 存取器中，C#编译器会自动创建变量的值。C# 3.0 中还有一种新的缩写方式：如果 `get` 和 `set` 存取器只返回或设置变量，就不需要实现代码。这与 C++/CLI 和 Visual Basic 不同，这两种语言都使用 `property` 关键字，且需要用 `set` 存取器定义变量的值。C++/CLI 还需要用 `get` 存取器指定返回类型，用 `set` 存取器指定参数类型。

C++/CLI 还可以用一个简短版本来编写属性。使用 `property` 关键字，只需定义属性和类型的名称，`get` 和 `set` 存取器由编译器自动创建。如果除了设置和返回变量之外不需要做其他工作，就可以使用这个简短版本。如果存取器的实现代码还需要做其他工作，例如检查变量或刷新，就必须使用属性的完整语法。C# 3.0 设计器从 C++/CLI 中学到了这个简短版本。

```
// C#
public class Person
{
    private string firstname;

    public string Firstname
    {
        get { return firstname; }
        set { firstname = value; }
    }

    public string LastName { get; set; }
}
```

```
// C++/CLI
public ref class Person
{
private:
    String^ firstname;
public:
    property String^ Firstname
    {
        String^ get()
        {
            return firstname;
        }
        void set(String^ value)
```

```

    {
        firstname = value;
    }
}

property String^ Lastname;
};

```

```

'Visual Basic
Public Class Person
    Private myFirstname As String

    Public Property Firstname()
        Get
            Return myFirstname
        End Get
        Set(ByVal value)
            myFirstname = value
        End Set
    End Property

    Private myLastname As String
    Public Property Lastname()
        Get
            Return myLastname
        End Get
        Set(ByVal value)
            myLastname = value
        End Set
    End Property
End Class

```

在 C# 和 C++/CLI 中，只读属性只有 get 存取器。在 Visual Basic 中，还必须指定 `ReadOnly` 修饰符，只写属性必须用 `WriteOnly` 修饰符和 set 存取器定义。

```

' Visual Basic

Public ReadOnly Property Name()
    Get
        Return myFirstname & " " & myLastname
    End Get
End Property

```

### B.3.5 对象初始化器

在 C# 3.0 和 VB 中，属性可以使用对象初始化器进行初始化。属性可以使用类似于数组初始化器的花括号来初始化。C# 和 VB 中的语法非常类似，VB 仅使用 `With` 关键字：

```

// C#
Person p = new Person() { FirstName = "Tom", LastName = "Turbo" };

' Visual Basic
Dim p As New Person With { .FirstName = "Tom", .LastName = "Turbo" }

```

### B.3.6 扩展方法

扩展方法是 LINQ 的基础。在 C# 和 VB 中，可以创建扩展方法。但是其语法不同。C# 在第一个参数中用 `this` 关键字标记扩展方法，VB 用属性 `<Extension>` 标记扩展方法。



```
// C#
public static class StringExtension
{
    public static void Foo(this string s)
    {
        Console.WriteLine("Foo {0}", s);
    }
}

' Visual Basic
Public Module StringExtension
    < Extension() > _
    Public Sub Foo(ByVal s As String)
        Console.WriteLine("Foo {0}", s)
    End Sub
End Module
```

## B.4 静态成员

静态字段只为某种类型的所有对象实例化一次。C#和 C++/CLI 都使用 `static` 关键字，Visual Basic 则使用 `Shared` 关键字。

使用静态成员的方法是：指定类名，其后是“.”操作符和静态成员名。C++/CLI 使用：`::`操作符来访问静态成员。

```
// C#
public class Singleton
{
    private static SomeData data = null;

    public static SomeData GetData()
    {
        if (data == null)
        {
            data = new SomeData();
        }
        return data;
    }
}

// use:
SomeData d = Singleton.GetData();
```

```
// C++/CLI
public ref class Singleton
{
private:
    static SomeData^ hData;

public:
    static SomeData^ GetData()
    {
        if (hData == nullptr)
        {
            hData = gcnew SomeData();
        }
        return hData;
    }
};
```

```
// use:
SomeData^ d = Singleton::GetData();
```

```
'Visual Basic
Public Class Singleton
    Private Shared data As SomeData

    Public Shared Function GetData() As SomeData
        If data is Nothing Then
            data = new SomeData()
        End If
        Return data
    End Function
End Class
```

```
'Use:
Dim d as SomeData = Singleton.GetData()
```

## B.5 数组

数组在第 5 章讨论过了。Array 类总是后台的 .NET 数组。声明数组时，编译器会创建一个派生于 Array 基类的类。在设计 C# 时，采用了 C++ 数组的括号语法，并用数组初始化器进行了扩展。

```
// C#
int[] arr1 = new int[3] {1, 2, 3};
int[] arr2 = {1, 2, 3};
```

如果在 C++/CLI 中使用括号，就会创建一个内置的 C++ 数组，而不是基于 Array 类的数组。为了创建 .NET 数组，C++/CLI 引入了 array 关键字。这个关键字使用类似于泛型的语法，即使用尖括号。在尖括号中指定元素的类型。C++/CLI 支持数组初始化器的语法与 C# 相同。

```
// C++/CLI
array<int>^ arr1 = gcnew array<int>(3) {1, 2, 3};
array<int>^ arr2 = {1, 2, 3};
```

Visual Basic 给数组使用括号。它要求在数组声明中指定最后一个元素号，而不是数组中的元素个数。在每种 .NET 语言中，数组都以元素号 0 开始，Visual Basic 也是如此。为了使之更清楚，Visual Basic 9 在数组声明中引入了 0 To number 表达式。它总是从 0 开始，To 使该语法更容易理解。

如果数组用 new 操作符初始化，Visual Basic 还支持数组初始化器。

```
'Visual Basic
Dim arr1(0 To 2) As Integer()
Dim arr2 As Integer() = New Integer(0 To 2) {1, 2, 3};
```

## B.6 控制语句

控制语句指定应运行什么代码。C# 定义了 if 和 switch 语句，以及条件操作符。

### B.6.1 if 语句

C#的 if 语句与 C++/CLI 版本相同，Visual Basic 使用 If-Then/Else/End If 来替代花括号。

```
// C# and C++/CLI
if (a == 3)
{
    // do this
}
else
{
    // do that
}
```

```
'Visual Basic
If a = 3 Then
    'do this
Else
    'do that
End If
```

### B.6.2 条件操作符

C#和 C++/CLI 支持条件操作符，它是 if 语句的一个轻型版本。在 C++/CLI 中，这个操作符称为三元操作符。第一个参数必须等于布尔值，如果结果为 true，就计算第一个表达式，否则就计算第二个表达式。Visual Basic 在 Visual Basic Runtime Library 中提供了有类似功能的 IIf 函数。

```
// C#
string s = a > 3 ? "one" : "two";
```

```
// C++/CLI
String^ s = a > 3 ? "one" : "two";
```

```
' Visual Basic
Dim s As String = IIf(a > 3, "one", "two")
```

### B.6.3 switch 语句

C#和 C++/CLI 中的 switch 语句看起来很类似，但它们有重要的区别。C#支持在 case 选项中使用字符串，但 C++不支持。在 C++中，必须使用 if-else。C++/CLI 支持从一个 case 选项向下一个 case 选项移动。但在 C#中，如果没有 break 或 goto 语句，编译器就会发出警告。只有 case 中没有语句时，C#才支持从一个 case 选项向下一个 case 选项移动。

Visual Basic 用 Select/Case 语句替代了 switch/case。它不需要也不允许使用 break 语句。即使 Case 中没有任何语句，也不能从一个 case 选项向下一个 case 选项移动。Case 可以用 And、Or 和 To 合并，例如 3 To 5。

```
// C#
string GetColor(Suit s)
{
    string color;
    switch (s)
    {
```

```

        case Suit.Heart:
        case Suit.Diamond:
            color = "Red";
            break;
        case Suit.Spade:
        case Suit.Club:
            color = "Black";
            break;
        default:
            color = "Unknown";
            break;
    }
    return color;
}

```

```

// C++/CLI
String^ GetColor(Suit s)
{
    String^ color;
    switch (s)
    {
        case Suit::Heart:
        case Suit::Diamond:
            color = "Red";
            break;
        case Suit::Spade:
        case Suit::Club:
            color = "Black";
            break;
        default:
            color = "Unknown";
            break;
    }
    return color;
}

```

```

'Visual Basic
Function GetColor(ByVal s As Suit) As String
    Dim color As String = Nothing
    Select Case s
        Case Suit.Heart And Suit.Diamond
            color = "Red"
        Case Suit.Spade And Suit.Club
            color = "Black"
        Case Else
            color = "Unknown"
    End Select

    Return color
End Function

```

## B.7 循环

使用循环，代码会重复执行，直到满足一个条件为止。C#中的循环详见第2章，包括 `for`、`while`、`do..while` 和 `foreach`。C#和 C++/CLI 的这些语句非常类似，而 Visual Basic 定义了不同的语句。



### B.7.1 for 语句

C#和 C++/CLI 的 for 语句很类似。在 Visual Basic 中，不能在 For/To 语句中初始化变量，而必须提前初始化变量。For/To 不需要使用 Step 语句，因为默认使用 Step 1。只有不打算将递增量设置为 1，For/To 才需要使用 Step 关键字。

```
// C#
for (int i = 0; i < 100; i++)
{
    Console.WriteLine(i);
}
```

```
// C++/CLI
for (int i = 0; i < 100; i++)
{
    Console::WriteLine(i);
}
```

```
'Visual Basic
Dim count as Integer
For count = 0 To 99 Step 1
    Console.WriteLine(count)
Next
```

### B.7.2 while 和 do..while 语句

while 和 do..while 语句在 C#和 C++/CLI 中是相同的。Visual Basic 中的 Do While/Loop 和 Do/Loop While 的结构与它们非常类似。

```
// C#
int i = 0;
while (i < 3)
{
    Console.WriteLine(i++);
}
```

```
i = 0;
do
{
    Console.WriteLine(i++);
} while (i < 3);
```

```
// C++/CLI
int i = 0;
while (i < 3)
{
    Console::WriteLine(i++);
}
```

```
i = 0;
do
{
    Console::WriteLine(i++);
} while (i < 3);
```

```
'Visual Basic
Dim num as Integer = 0
Do While (num < 3)
```



```

        Console.WriteLine(num)
        num += 1
    Loop

    num = 0
    Do
        Console.WriteLine(num)
        num += 1
    Loop While (num < 3)

```

### B.7.3 foreach 语句

foreach 语句使用 IEnumerable 接口。foreach 语句在 ANSI C++ 中不存在，但它是 ANSI C++/CLI 中的一个扩展。与 C# 的 foreach 语句不同，在 C++/CLI 中，for 和 each 之间有一个空格。Visual Basic 的 For Each 语句不允许在循环内部声明迭代变量的类型，该类型必须提前声明。

```

// C#
int[] arr = {1, 2, 3};
foreach (int i in arr)
{
    Console.WriteLine(i);
}

```

```

// C++/CLI
array<int>^ arr = {1, 2, 3};
for each (int i in arr)
{
    Console::WriteLine(i);
}

```

```

'Visual Basic
Dim arr() As Integer = New Integer() {1, 2, 3}
Dim num As Integer
For Each num In arr
    Console.WriteLine(num)
Next

```

#### 提示：

foreach 很容易迭代集合，而 C# 允许使用 yield 语句创建枚举。Visual Basic 和 C++/CLI 不能使用 yield 语句，而必须手工实现 IEnumerable 和 IEnumerator 接口。yield 语句参见第 5 章。

## B.8 异常处理

异常处理详见第 14 章。这三种语言的异常处理非常类似。它们都使用 try/catch/finally 来处理异常，用 throw 关键字创建异常：

```

// C#
public void Method(Object o)
{
    if (o == null)
        throw new ArgumentException("Error");
}

public void Foo()

```

```

{
    try
    {
        Method(null);
    }
    catch (ArgumentException ex)
    { }
    catch (Exception ex)
    { }
    finally
    { }
}

```

```

// C++/CLI
public:
    void Method(Object^ o)
    {
        if (o == nullptr)
            throw gcnew ArgumentException("Error");
    }

    void Foo()
    {
        try
        {
            Method(nullptr);
        }
        catch (ArgumentException^ ex)
        { }
        catch (Exception^ ex)
        { }
        finally
        { }
    }
}

```

```

'Visual Basic
Public Sub Method(ByVal o As Object)
    If o = Nothing Then
        Throw New ArgumentException("Error")
    End Sub
Public Sub Foo()
    Try
        Method(Nothing)
    Catch ex As ArgumentException
    '
    Catch ex As Exception
    '
    Finally
    '
    End Try
End Sub

```

## B.9 继承

第 4 章讨论了许多关键字，用于定义多态操作、覆盖或隐藏方法；以及访问修饰符，以允许访问或不允许访问类的成员。C#、C++/CLI 和 Visual Basic 的功能非常类似，但关键字不同。

B.9.1 访问修饰符

C++/CLI 和 Visual Basic 的访问修饰符非常类似于 C#，但有一些显著的区别。Visual Basic 使用 Friend 访问修饰符替代 internal，来访问同一个程序集中的类型。C++/CLI 还有一个访问修饰符 protected private。internal protected 允许访问同一个程序集中的成员，还可以访问其他程序集中派生自基类的类型。C#和 Visual Basic 不允许访问同一个程序集中的派生类型，但在 C++/CLI 中这可以使用 protected private 来实现。其中 private 表示在程序集的外部不能访问，但在程序集的内部可以进行受保护的访问。其顺序即 protected private 或 private protected 并不重要。访问修饰符总是在程序集中允许访问的内容较多，在程序集的外部允许访问的内容较少。见表 B-3。

表 B-3

C#	C++/CLI	Visual Basic
public	Public	Public
protected	Protected	Protected
private	Private	Private
internal	Internal	Friend
internal protected	internal protected	Protected Friend
无	protected private	无

B.9.2 关键字

对继承很重要的关键字如表 B-4 所示。

表 B-4

C#	C++/CLI	Visual Basic	功 能
:	:	Implements	实现一个接口
:	:	Inherits	继承一个基类
virtual	virtual	Overridable	声明一个支持多态性的方法
overrides	overrides	Overrides	重写一个虚方法
new	new	Shadows	隐藏基类中的方法
abstract	abstract	MustInherit	抽象类
sealed	sealed	NotInheritable	密封类
abstract	abstract	MustOverride	抽象方法
sealed	sealed	NotOverridable	密封方法
this	this	Me	引用当前对象
base	Classname::	MyBase	引用基类



关键字的放置顺序在各种语言中是很重要的。在代码示例中，抽象基类 **Base** 有一个抽象方法和一个已实现了的虚方法。类 **Derived** 派生自 **Base**，它实现了抽象方法，重写了虚方法。

```
// C#
public abstract class Base
{
    public virtual void Foo()
    {
    }
    public abstract void Bar();
}

public class Derived : Base
{
    public override void Foo()
    {
        base.Foo();
    }
    public override void Bar()
    {
    }
}
```

```
// C++/CLI
public ref class Base abstract
{
public:
    virtual void Foo()
    {
    }
    virtual void Bar() abstract;
};

public ref class Derived : public Base
{
public:
    virtual void Foo() override
    {
        Base::Foo();
    }
    virtual void Bar() override
    {
    }
};
```

```
'Visual Basic
Public MustInherit Class Base
    Public Overridable Sub Foo()
    End Sub
    Public MustOverride Sub Bar()
End Class

Public Class Derived
    Inherits Base
    Public Overrides Sub Foo()
        MyBase.Foo()
    End Sub
    Public Overrides Sub Bar()
    End Sub
End Class
```

## B.10 资源管理

第 12 章介绍了资源管理，实现了 `IDisposable` 接口和一个终结器。本节介绍这些功能在 C++/CLI 和 Visual Basic 中如何实现。

### B.10.1 IDisposable 接口的实现

为了释放资源，`IDisposable` 接口定义了 `Dispose()` 方法。使用 C# 和 Visual Basic 时，必须实现 `IDisposable` 接口。在 C++/CLI 中，也实现了 `IDisposable` 接口，但如果只编写了一个析构函数，这将由编译器完成。

```
// C#
public class Resource : IDisposable
{
    public void Dispose()
    {
        // release resource
    }
}
```

```
// C++/CLI
public ref class Resource
{
public:
    ~Resource()
    {
        // release resource
    }
};
```

```
'Visual Basic
Public Class Resource
    Implements IDisposable

    Public Sub Dispose() Implements IDisposable.Dispose
        'release resource
    End Sub
End Class
```

#### 警告：

在 C++/CLI 中，`Dispose()` 方法使用 `delete` 语句调用。

### B.10.2 using 语句

C# 的 `using` 语句使用“获取/使用/释放”模式释放不再使用的资源，甚至在出现异常的情况下也是如此。编译器创建一个 `try/finally` 语句，在 `finally` 中调用 `Dispose()` 方法。Visual Basic 9 支持类似于 C# 的 `using` 语句。C++/CLI 为这个问题提供了更好的方法。如果引用类型在本地声明，编译器就创建一个 `try/finally` 语句，在该块的最后调用 `Dispose()` 方法。

```
// C#
using (Resource r = new Resource())
```



```
{
    r.Foo();
}
```

```
// C++/CLI
{
    Resource r;
    r.Foo();
}
```

```
' Visual Basic
Using r As New Resource
    r.Foo()
End Using
```

### B.10.3 重写 Finalize()

如果类包含必须释放的内置资源，该类就必须重写 `Object` 类中的 `Finalize()` 方法。在 C# 中，这只需编写一个析构函数。C++/CLI 采用一种特殊的语法：用 “!” 前缀定义终结器。在终结器中，不允许删除有终结器的对象，以确保终结器的执行顺序。这就是 `Dispose` 模式定义一个带 `Boolean` 参数的 `Dispose()` 方法的原因。在 C++/CLI 中，不需要在代码中采用这个模式，因为这是由编译器完成的。C++/CLI 析构函数实现了两个 `Dispose()` 方法。在 Visual Basic 中，两个 `Dispose()` 方法和终结器都必须手工实现。但是大多数 Visual Basic 类都不直接使用内置资源，而是借助于其他类。在 Visual Basic 中，通常不需要重写 `Finalize()` 方法，但一般需要执行 `Dispose()` 方法。

#### 提示：

在 C# 中，编写析构函数来重写基类的 `Finalize()` 方法；C++/CLI 的析构函数实现了 `IDisposable` 接口。

```
// C#
public class Resource : IDisposable
{
    ~Resource // override Finalize
    {
        Dispose(false);
    }

    protected virtual void Dispose(bool disposing)
    {
        if (disposing) // dispose embedded members
        {
        }
        // release resources of this class
        GC.SuppressFinalize(this);
    }

    public void Dispose()
    {
        Dispose(true);
    }
}

// C++/CLI
public ref class Resource
{
public:
```

```

~Resource() // implement IDisposable
{
    this->!Resource();
}
!Resource() // override Finalize
{
    // release resource
}
};

```

```

'Visual Basic
Public Class Resource
    Implements IDisposable

    Public Sub Dispose() Implements IDisposable.Dispose
        Dispose(True)
        GC.SuppressFinalize(Me)
    End Sub

    Protected Overridable Sub Dispose(ByVal disposing)
        If disposing Then
            'Release embedded resources
        End If
        'Release resources of this class
    End Sub

    Protected Overrides Sub Finalize()
        Try
            Dispose(False)
        Finally
            MyBase.Finalize()
        End Try
    End Sub
End Class

```

## B.11 委托

第 7 章讨论的委托是方法类型安全的指针。在这三种语言中，都可以使用 `delegate` 关键字定义委托，但使用委托的方式有区别。

示例代码使用的类 `Demo` 有一个静态方法 `Foo()` 和一个实例方法 `Bar()`。这两个方法都由 `DemoDelegate` 类型的委托实例来调用。`DemoDelegate` 声明为调用一个返回类型为 `void`、带 `int` 参数的方法。

C#使用委托来支持委托推断，编译器会创建一个委托实例，传送方法的地址。在 C#和 C++/CLI 中，可以使用 `+` 操作符将两个委托合并为一个：

```

// C#
public delegate void DemoDelegate(int x);

public class Demo
{
    public static void Foo(int x) { }
    public void Bar(int x) { }
}

Demo d = new Demo();
DemoDelegate dl = Demo.Foo;

```

```
DemoDelegate d2 = d.Bar;
DemoDelegate d3 = d1 + d2;
d3(11);
```

C++/CLI 不支持委托推断。C++/CLI 需要创建一个委托类型的新实例，将方法的地址传送给构造函数。

```
// C++/CLI
public delegate void DemoDelegate(int x);

public ref class Demo
{
public:
    static void Foo(int x) { }
    void Bar(int x) { }
};

Demo^ d = gcnew Demo();
DemoDelegate^ d1 = gcnew DemoDelegate(&Demo::Foo);
DemoDelegate^ d2 = gcnew DemoDelegate(d, &Demo::Bar);
DemoDelegate^ d3 = d1 + d2;
d3(11);
```

与 C++/CLI 类似，Visual Basic 也不支持委托推断。必须创建一个委托类型的新实例，传送方法的地址。Visual Basic 使用 `AddressOf` 操作符传送方法的地址。

Visual Basic 没有为委托重载+操作符，所以需要调用 `Delegate` 类的 `Combine()` 方法。`Delegate` 类写在括号中，因为 `Delegate` 是一个 Visual Basic 关键字，不能使用同名的类。给 `Delegate` 加上括号，可确保使用类，而不是 `Delegate` 关键字。

```
'Visual Basic
Public Delegate Sub DemoDelegate(ByVal x As Integer)

Public Class Demo
    Public Shared Sub Foo(ByVal x As Integer)
    ,
    End Sub
    Public Sub Bar(ByVal x As Integer)
    ,
    End Sub
End Class

Dim d As New Demo()
Dim d1 As New DemoDelegate(AddressOf Demo.Foo)
Dim d2 As New DemoDelegate(AddressOf d.Bar)
Dim d3 As DemoDelegate = [Delegate].Combine(d1, d2)
d3(11)
```

## B.12 事件

使用 `event` 关键字可以实现基于委托的订阅机制。这三种语言都定义了 `event` 关键字，提供类中的事件。下面的类 `EventDemo` 引发了 `DemoDelegate` 类型的事件 `DemoEvent`。

在 C# 中，引发事件的语法类似于事件的方法调用。只要没有人注册事件，事件变量就是 `null`。所以在引发事件之前要检查事件变量是否为 `null`。处理程序方法在注册时，要使用 `+=` 操作符，在委托推断的帮助下传送处理方法的地址。



```
// C#
public class EventDemo
{
    public event DemoDelegate DemoEvent;

    public void FireEvent()
    {
        if (DemoEvent != null)
            DemoEvent(44);
    }
}

public class Subscriber
{
    public void Handler(int x)
    {
        // handler implementation
    }
}

//...
EventDemo evd = new EventDemo();
Subscriber subscr = new Subscriber();
evd.DemoEvent += subscr.Handler;
evd.FireEvent();
```

C++/CLI 与 C# 非常类似，但引发事件不需要先检查事件变量不为 null，这由编译器创建的 IL 代码自动完成。

#### 提示：

C# 和 C++/CLI 使用 += 操作符来注销事件。

```
// C++/CLI
public ref class EventDemo
{
public:
    event DemoDelegate^ DemoEvent;

    public void FireEvent()
    {
        DemoEvent(44);
    }
}

public class Subscriber
{
public:
    void Handler(int x)
    {
        // handler implementation
    }
}

//...
EventDemo^ evd = gcnew EventDemo();
Subscriber^ subscr = gcnew Subscriber();
evd->DemoEvent += gcnew DemoDelegate(subscr, &Subscriber::Handler);
evd->FireEvent();
```

Visual Basic 使用不同的语法。事件用 Event 关键字声明，这与 C# 和 C++/CLI 相同，但是，事

件用 `RaiseEvent` 语句引发。`RaiseEvent` 语句检查事件变量是否用订阅器初始化了。用于注册处理程序的 `AddHandler` 语句与 C# 中的 `+=` 操作符有相同的功能。`AddHandler` 需要两个参数：第一个参数定义事件，第二个参数定义处理程序的地址。`RemoveHandler` 语句用于从事件中注销处理程序。

```
'Visual Basic
Public Class EventDemo
    Public Event DemoEvent As DemoDelegate

    public Sub FireEvent()
        RaiseEvent DemoEvent(44);
    End Sub
End Class

Public Class Subscriber
    Public Sub Handler(ByVal x As Integer)
        'handler implementation
    End Sub
End Class

'...
Dim evd As New EventDemo()
Dim subscr As New Subscriber()
AddHandler evd.DemoEvent, AddressOf subscr.Handler
evd.FireEvent()
```

Visual Basic 提供的另一种语法不能用于其他两种语言：对订阅事件的方法使用 `Handles` 关键字。它要求用 `WithEvents` 关键字定义一个变量：

```
Public Class Subscriber
    Public WithEvents evd As EventDemo

    Public Sub Handler(ByVal x As Integer) Handles evd.DemoEvent
        'Handler implementation
    End Sub

    Public Sub Action()
        evd = New EventDemo()
        evd.FireEvent()
    End Sub
End Class
```

## B.13 泛型

这三种语言都支持泛型的创建和使用。泛型参见第 9 章。

为了使用泛型，C# 借用了 C++ 模板的语法，用尖括号定义泛型类型。C++/CLI 使用相同的语法。在 Visual Basic 中，泛型类型用括号中的 `Of` 关键字定义。

```
// C#
List<int> intList = new List<int>();
intList.Add(1);
intList.Add(2);
intList.Add(3);
```

```
// C++/CLI
List<int>^ intList = gcnew List<int>();
intList->Add(1);
```



```
intList->Add(2);
intList->Add(3);
```

```
'Visual Basic
Dim intList As List(Of Integer) = New List(Of Integer)()
intList.Add(1)
intList.Add(2)
intList.Add(3)
```

因为类声明使用了尖括号，所以编译器知道要创建一个泛型类型。约束用 **where** 子句定义。

```
public class MyGeneric<T>
    where T : IComparable<T>
{
    private List<T> list = new List<T>();
    public void Add(T item)
    {
        list.Add(item);
    }

    public void Sort()
    {
        list.Sort();
    }
}
```

用 C++/CLI 定义泛型类型与用 C++ 定义模板类似。但 C++/CLI 不使用 **template** 关键字，而使用 **generic** 关键字。**where** 子句类似于 C# 中的 **where** 子句，但 C++/CLI 没有构造函数的限制。

```
generic <typename T>
where T : IComparable<T>
ref class MyGeneric
{
private:
    List<T>^ list;

public:
    MyGeneric()
    {
        list = gcnew List<T>();
    }

    void Add(T item)
    {
        list->Add(item);
    }

    void Sort()
    {
        list->Sort();
    }
};
```

Visual Basic 用 **Of** 关键字定义泛型类，条件用 **As** 定义。

```
Public Class MyGeneric(Of T As IComparable(Of T))
    Private myList = New List(Of T)

    Public Sub Add(ByVal item As T)
        myList.Add(item)
    End Sub
```

```

Public Sub Sort()
    myList.Sort()
End Sub
End Class

```

## B.14 LINQ 查询

语言集成的查询是 C# 3.0 和 VB 9.0 的一个特性。这两种语言的语法非常类似：

**提示：**

LINQ 参见第 11 章。

```

// C#
var query = from r in racers
             where r.Country == "Brazil"
             orderby r.Wins descending
             select r;

' Visual Basic
Dim query = From r in racers
             Where r.Country = "Brazil"
             Order By r.Wins Descending
             Select r

```

**提示：**

C++/CLI 不支持 LINQ 查询。

## B.15 C++/CLI 混合内置代码和托管代码

C++/CLI 的一大优点是混合内置代码和托管代码。使用 C# 中的内置代码需要一种机制，称为平台调用。平台调用参见第 24 章。在 C++/CLI 中使用内置代码称为 “It just works”。

在托管类中，可以使用内置代码和托管代码，如下所示。内置类也是如此。可以在一个方法中混合使用内置代码和托管代码。

```

#pragma once
#include <iostream> // include this header file for cout

using namespace std; // the iostream header defines the namespace std
using namespace System;

public ref class Managed
{
public:
    void MixNativeAndManaged()
    {
        cout << "Native Code" << endl;
        Console::WriteLine("Managed Code");
    }
};

```

在托管类中，还可以声明内置类型的字段或指针。但不能在内置类中声明托管类型的字段或指针。必须注意，托管类型的实例可以在垃圾收集器清理内存时被删除。

为了将托管类用作内置类中的成员，C++/CLI 定义了关键字 `gcroot`，它在头文件 `gcroot.h`

中定义。gcroot 封装了一个 CGHandle，用于跟踪内置引用中的 CLR 对象。

```
#pragma once
#include "gcroot.h"

using namespace System;

public ref class Managed
{
public:
    Managed() {}

    void Foo()
    {
        Console::WriteLine("Foo");
    }
};

public class Native
{
private:
    gcroot<Managed^> m_p;

public:
    Native()
    {
        m_p = gcnew Managed();
    }

    void Foo()
    {
        m_p->Foo();
    }
};
```

## B.16 C#的特殊功能

一些 C#语法特性没有在本附录中介绍。C#定义了 yield 语句，它便于创建枚举器。这个语句不能用于 C++/CLI 和 VB。在这些语言中，枚举器必须手工实现。另外，C#还为可空类型定义了特殊的语法，而在其他语言中，必须使用泛型结构 Nullable<T>。

C#允许使用不安全的代码块，在不安全的代码块中可以使用指针和指针算术。这个特性非常有利于调用内部库中的方法。VB 没有这个功能，这是 C#的一个优势。C++/CLI 不需要使用 unsafe 关键字来定义不安全的代码块，C++/CLI 本身就混合了内部代码和托管代码。

## B.17 小结

本章学习了如何将 C#的语法映射到 Visual Basic 和 C++/CLI 上。C++/CLI 定义了对 C++的扩展，以编写 .NET 应用程序，描绘了根据 C#进行的语法扩展。C#和 C++/CLI 尽管根源相同，但有许多重要的区别。Visual Basic 没有使用花括号，但比较饶舌。

在语法映射上，本章探讨了如何把 C#语法映射到 Visual Basic 和 C++/CLI 上，介绍了其他两种语言的语法，如何定义类型、方法、属性，哪些关键字用于 OO 特性，资源管理如何进行，委托、事件和泛型在三种语言中如何实现。

尽管可以映射大多数语法，但这些语言的功能仍有区别。

# Windows Vista 和 Windows Server 2008

本附录介绍为 Windows Vista 和 Windows Server 2008 开发应用程序所需要了解的内容，以及如何在 .NET 应用程序中使用新的 Windows 特性。本章不介绍对 Windows Vista 用户或 Windows Server 2008 管理员有用的特性，而只讨论对开发人员很重要的特性。

如果应用程序不只面向 Windows Vista，就应注意，WPF、WCF 和 WF 也可以用于 Windows XP，但本章不讨论这些主题。如果仍使用 Windows XP，还要考虑在 Windows Vista 上运行应用程序的问题和相关的注意事项。此时，应特别关注用户账户控制和目录的变化。

本章的主要内容如下：

- Vista Bridge
- 用户账户控制
- 目录结构
- 新控件和对话框
- 搜索

## C.1 Vista Bridge

.NET 3.5 发布之后，Windows Vista 和 Windows Server 2008 上许多新的 Windows API 调用不能用于 .NET Framework。但是，Windows SDK 包含一个名为 Vista Bridge 的示例，它封装了新的 API 调用，使它们可用于 .NET 库。可以在 Windows 窗体或 WPF 应用程序中使用这个库。

安装了 Windows SDK 后，会发现 Vista Bridge 示例在 <program files>\Microsoft SDKs\Windows\v6.0\Samples\CrossTechnologySamples.zip 文件中。在这个 .zip 文件中有三个项目 VistaBridgeLibrary、VistaBridgeControls 和 VistaBridgeDemoApp。VistaBridgeLibrary 项目包含几个类和控件。

## C.2 用户账户控制

作为开发人员，应知道用户账户控制(UAC)是 Windows Vista 和 Windows Server 2008 中的



一个特性。尽管 Windows 规则总是提及这个问题，但许多应用程序仍需要运行在管理员账户下。例如，一般用户不允许直接将数据写入 `program files` 目录，这需要管理权限。许多应用程序都不能在没有管理权限的情况下运行(但程序的功能并不需要管理权限)，所以许多用户都使用管理员账户登录系统。当然，这会带来安装木马程序的高风险。

Windows Vista 避免了这个问题，管理员在默认情况下没有管理权限。这个过程有两个相关的安全标志，一个用于一般用户权限，另一个用于管理权限(此时登录到管理员账户上)。对于需要管理权限的应用程序，用户可以以管理员的身份运行它，为此，可以使用关联菜单 `Run as Administrator`，或者在应用程序的 `Compatibility` 属性中，将应用程序配置为总是需要管理权限，如图 C-1 所示。这个设置会在注册表的 `HKCU\Software\Microsoft\Windows NT\CurrentVersion\AppCompatFlags\Layers` 上添加应用程序兼容性标志，其值为 `RUNASADMIN`。

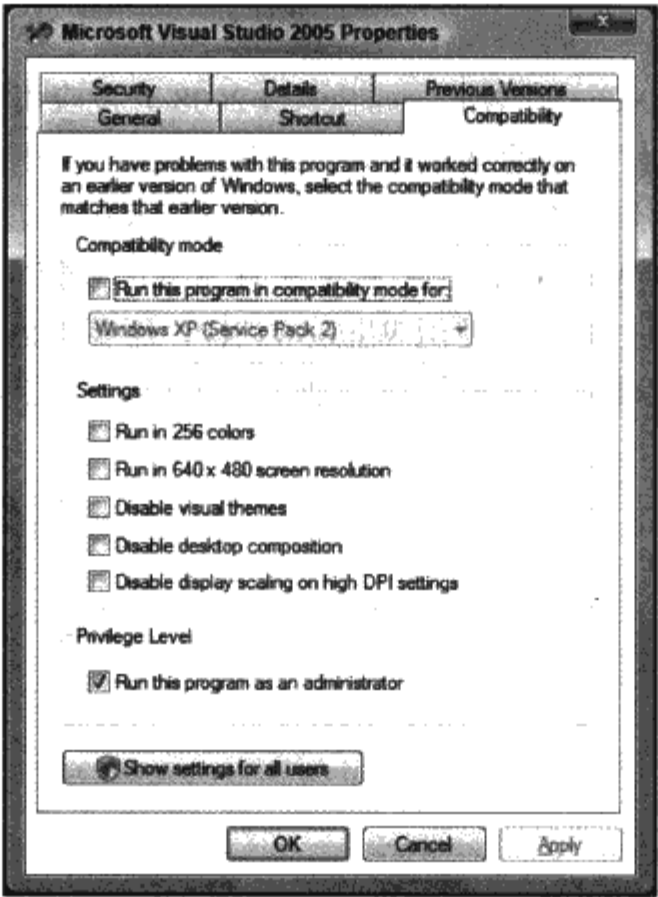


图 C-1

C.2.1 需要管理权限的应用程序

对于需要管理权限的应用程序，还可以添加一个应用程序清单。VS2008 有一个新的项模板，可以把应用程序清单添加到应用程序中。为此，可以给应用程序添加一个清单文件，或者在程序集中嵌入一个 Win32 资源文件。在 VS 项目中添加清单文件后，该清单文件就添加到项目的资源中。例如在项目的属性中，在 `Resources` 类别中选择 `Application` 选项卡，在此处添加该项，这会把清单嵌入为程序集的一个 Win32 资源。

应用程序清单是一个类似于应用程序配置文件的 XML 文件。应用程序配置文件的扩展名是 `.config`，而清单文件以 `.manifest` 结尾。该文件的名称必须设置为应用程序的名称，包括 `exe` 文件扩展名，其后是 `.manifest`。清单文件包含如下所示的 XML 数据。根元素是 `<assembly>`，它包含子元素 `<trustInfo>`。管理员要求用 `<requestedExecutionLevel>` 元素的 `level` 特性来定义。



```

< ?xml version="1.0" encoding="UTF-8"? >
< asmv1:assembly manifestVersion="1.0"
  xmlns="urn:schemas-microsoft-com:asm.v1"
  xmlns:asmv1="urn:schemas-microsoft-com:asm.v1"
  xmlns:asmv2="urn:schemas-microsoft-com:asmv2"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" >
  < assemblyIdentity version="1.0.0.0" name="MyApplication.app" / >
  < trustInfo xmlns="urn:schemas-microsoft-com:asm.v2" >
    < security >
      < requestedPrivileges xmlns="urn:schemas-microsoft-com:asm.v3" >
        < requestedExecutionLevel level="requireAdministrator"
          uiAccess="false"/ >
      < /requestedPrivileges >
    < /security >
  < /trustInfo >
< /asmv1:assembly >

```

以这种方式启动应用程序，就会显示一个提示，询问用户是否允许在管理权限下运行应用程序。

`requestedExecutionLevel` 设置可以指定为 `requireAdministrator`、`highestAvailable` 和 `asInvoker`。`highestAvailable` 表示，只有得到用户的允许，应用程序才能获得用户拥有的权限。`requireAdministrator` 需要管理员权限。如果用户没有以管理员的身份登录系统，就显示一个登录对话框，用户可以以应用程序管理员的身份登录。`asInvoker` 表示应用程序运行在用户的安全标志下。

`uiAccess` 属性指定，应用程序是否需要输入到桌面上一个权限级别较高的窗口上。例如，屏幕上的键盘需要将输入放在桌面上的其他窗口中，因此对于显示屏幕软键盘的应用程序，这个设置应为 `true`。不支持 UI 访问的应用程序应将这个属性设置为 `false`。

#### 警告：

给应用程序授予管理权限的另一个选项是编写 Windows 服务。因为 UAC 仅用于交互式过程，所以 Windows 服务可以获得管理权限。还可以编写一个无权限的 Windows 应用程序，使用 WCF 或另一个通信技术与有权限的 Windows 服务通信。

#### 提示：

Windows 服务参见第 23 章，WCF 参见第 42 章。

### C.2.2 保护图标

如果应用程序或其中的任务需要管理权限，应通过一个容易识别的保护图标通知用户。保护图标附着在需要提高权限级别的控件上。用户在单击带保护的项时，会看到一个提高权限级别提示。图 C-2 和图 C-3 显示了正在使用的保护图标。任务管理器需要提高权限级别，才能查看所有用户的进程。对于用户账户，需要提高权限级别，才能修改账户类型，让其他用户访问计算机。

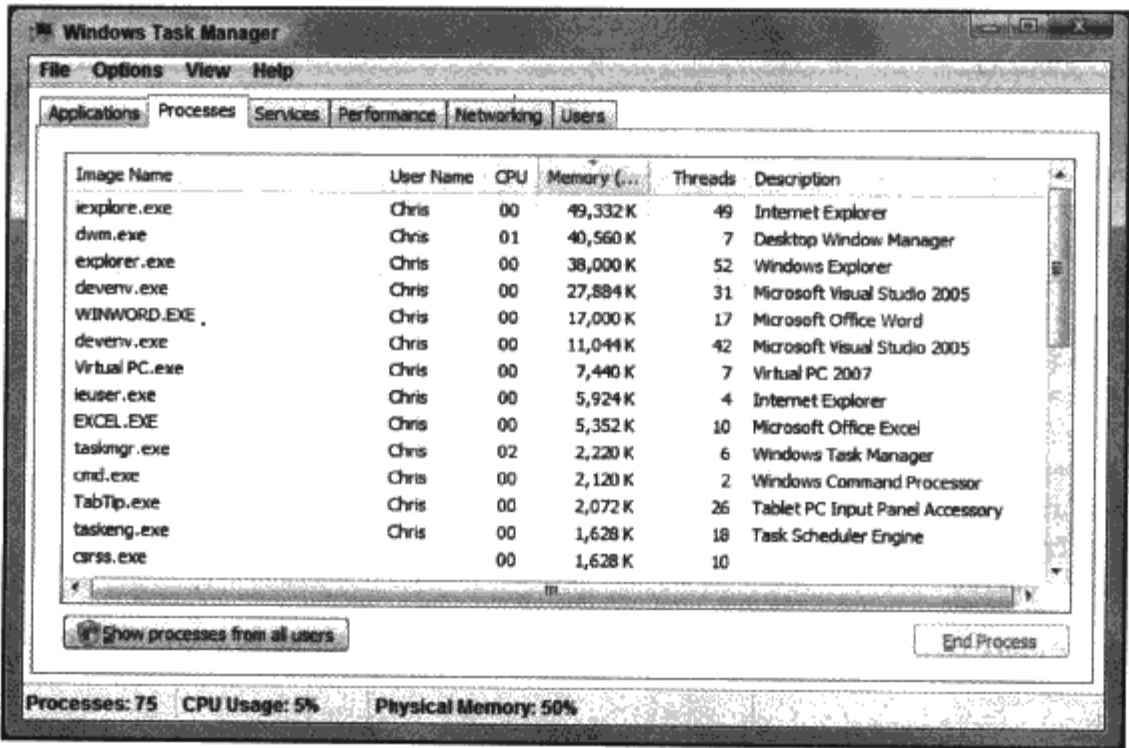


图 C-2

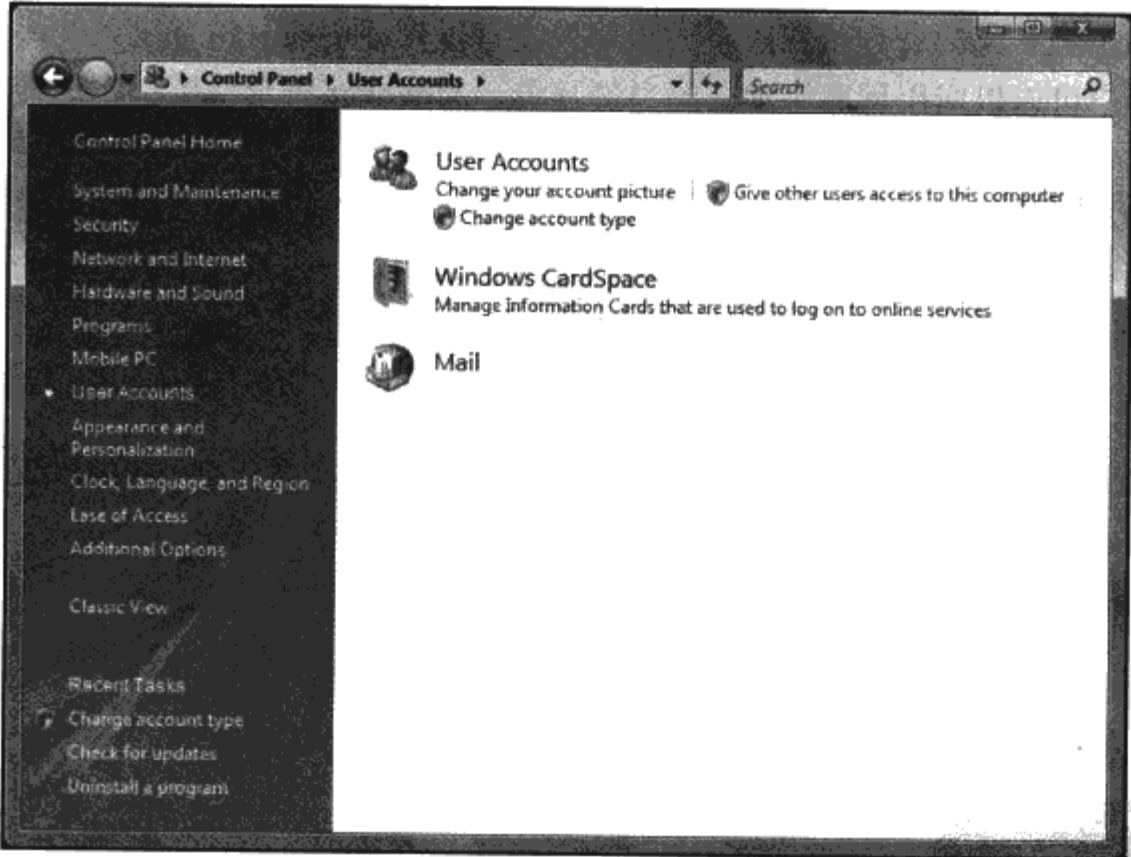


图 C-3

使用本章后面介绍的新命令链接控件，就可以在应用程序中创建保护图标。

当用户单击带保护图标的控件时，会显示一个提高权限级别的提示。提高权限级别的提示根据应用程序类型的不同而不同。

- Windows 需要许可才能继续运行。这个提示用于通过 Windows 发布的应用程序。
- 程序需要许可才能继续运行。这个提示用于包含证书，提供了出版商信息的应用程序。
- 未标识的程序要访问计算机。这个提示用于不包含证书的应用程序。

C.3 目录结构

Windows 的目录结构在 Windows Vista 中有变化。不再有 c:\Documents and Settings\

如果遵循不在程序中使用硬编码的路径值这个简单规则，文件夹在什么地方就不重要。文件夹在不同的 Windows 语言中是不同的。对于特定的文件夹，应使用 Environment 类和 SpecialFolder 枚举：

```
string folder = Environment.GetFolderPath(
    Environment.SpecialFolder.Personal);
```

SpecialFolder 枚举定义的一些文件夹如表 C-1 所示。

表 C-1

内 容	SpecialFolder 枚举	Windows Vista 默认目录
用户的专用文档	Personal	c:\Users\ <user&gt;\documents< td=""></user&gt;\documents<>
漫游用户的专用数据	ApplicationData	c:\Users\ <user&gt;\appdata\roaming< td=""></user&gt;\appdata\roaming<>
本地系统用户的专用数据	LocalApplicationData	c:\Users\ <user&gt;\appdata\local< td=""></user&gt;\appdata\local<>
程序文件	ProgramFiles	c:\Program Files
在不同程序中共享的程序文件	CommonProgramFiles	c:\Program Files\Common Files
所有用户共享的应用程序数据	CommonApplicationData	c:\ProgramData

提示：

注销时，漫游目录的内容会复制到服务器上，所以如果用户登录到另一个系统上，就会复制相同的内容，并可以用于该用户访问的所有系统。

使用特定文件夹时必须小心，因为一般用户不能对 Program Files 目录进行直接的写入访问。可以把应用程序中的用户专用数据写入 LocalApplicationData，对于漫游用户则写入 ApplicationData。应由不同用户共享的数据可以写入 CommonApplicationData。

许多应用程序都将内容写入 Program Files 目录，所以它们在没有管理权限的情况下不能运行在 Windows Vista 上。Windows Vista 为处理这些程序提供了一个解决方案：将文件夹重定向为一个虚拟库，应用程序可以在这个虚拟库上读写数据，不会生成错误。这个技术称为文件虚拟化。

下面验证一下：编写一个简单的程序，将一个文件写入 Program Files 目录的一个子目录 WroxSampleApp。使用 Environment.GetFolderPath()和 SpecialFolder 枚举值 ProgramFiles，返回 Program Files 文件夹；这个文件夹根据所使用的 Windows 语言而不同。Program Files 文件夹与目录 WroxSampleApp 合并，在这个目录中写入文件 samplefile.txt。

```
string programFiles = Environment.GetFolderPath(
```

```
Environment.SpecialFolder.ProgramFiles);  
string appDir = Path.Combine(programFiles, "WroxSampleApp");  
  
if (!Directory.Exists(appDir))  
{  
    Directory.CreateDirectory(appDir);  
}  
  
string demoFile = Path.Combine(appDir, "samplefile.txt");  
File.WriteAllText(demoFile, "test content");
```

在未提高权限级别的情况下运行应用程序，文件是不会写入目录 `c:\Program Files\WroxSampleApp` 的。这个文件在目录 `c:\Users\<Username> \AppData \Local\Virtual Store\Program Files\WroxSampleApp` 中。

可以看出，数据存储为用户专用的目录中，没有在同一系统的不同用户之间共享。如果有这个要求，就必须在提高权限级别的模式下启动应用程序。在提高权限级别的 Visual Studio 进程中运行应用程序，文件会写入 Program Files 文件夹，而不是虚拟库，因为在提高权限级别的进程中启动应用程序，应用程序的权限级别也会提高。

对于文件的读取，需要另一种机制。因为安装程序可以将内容写入 Program Files 文件夹，所以程序可以从 Program Files 文件夹中读取数据。只要程序在没有提高权限级别的情况下将数据写入这个文件夹，文件夹就会重定向为一个虚拟库。在程序读取写入的内容时，重定向操作和读取操作会同时发生。

虚拟化技术不仅用于文件夹，还用于注册表项。如果应用程序将注册表键 `Software` 写入 `HKEY_LOCAL_MACHINE`，它就会重定向到 `HKEY_CURRENT_USER`。它不是写入 `HKLM\Software\{Manufacturer}`，而是写入 `HKCU\Software\Classes\VirtualStore\MACHINE\SOFTWARE\{Manufacturer}`。

文件和注册表的虚拟化都只能用于 32 位应用程序，不能用于 Windows Vista 上的 64 位应用程序。

#### 警告：

不要将文件和注册表的虚拟化用作应用程序的一个特性。最好修改应用程序，而不是在不提高用户权限的情况下，将数据写入 Program Files 文件夹和 HKLM 注册表。重定向只是修复崩溃的应用程序的一种临时方式。

## C.4 新控件和对话框

Windows Vista 发布了几个新控件。命令链接控件是对 Button 控件的扩展，与其他几个控件一起使用。任务对话框是下一代的 MessageBox，还有用于打开和保存文件的新对话框。

### C.4.1 命令链接

命令链接控件是对 Windows 按钮控件的扩展。命令链接包含一个可选的图标和一个注意文本。这个控件常常用于任务对话框和向导。图 C-4 显示了两个命令链接控件，它们提供的信息

比带 OK 和 Cancel 的按钮控件更多。

在.NET 应用程序中,可以使用 Vista Bridge 示例库创建命令链接控件。如果将项目 Vista-BridgeLibrary 添加到解决方案中,就可以将工具箱中的 CommandLinkWinForms 控件添加到 Windows 窗体应用程序中。类 CommandLinkWinForms 派生自 System.Windows.Forms.Button 类。命令链接是对内置的 Windows 按钮控件的一种扩展,它定义了额外的 Windows 消息和一个配置按钮的新样式。封装类 CommandLinkWinForms 发送 Windows 消息 BCM\_SETNOTE 和 BCM\_SETSHIELD, 设置了样式 BS\_COMMANDLINK。除了 Button 类的成员之外,该类提供的公共方法和属性还有 NoteText 和 ShieldIcon。

下面的代码创建了一个新的命令链接控件,它设置了 NoteText 和 ShieldIcon。图 C-5 显示了运行期间配置的命令链接。

```
this.commandLinkDemo =
    new Microsoft.SDK.Samples.VistaBridge.Library.CommandLinkWinForms();

this.commandLinkDemo.NoteText =
    "The application deletes important files on your system";

this.commandLinkDemo.ShieldIcon = true;
this.commandLinkDemo.Size = new System.Drawing.Size(275, 68);
this.commandLinkDemo.Text = "Give access to this computer";
this.commandLinkDemo.UseVisualStyleBackColor = true;

this.Controls.Add(commandLinkDemo);
```

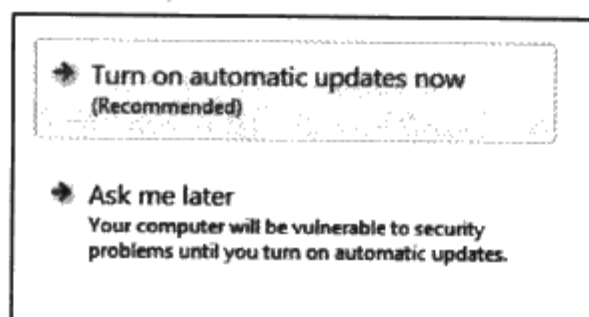


图 C-4

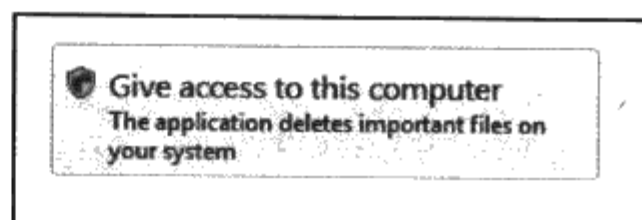


图 C-5

## C.4.2 任务对话框

任务对话框是替代旧消息框的新一代对话框,它是新命令控件的一部分。Windows API 定义了函数 TaskDialog 和 TaskDialogIndirect 来创建任务对话框。TaskDialog 可以创建简单的对话框,TaskDialogIndirect 用于创建比较复杂的对话框,其中包含了命令链接控件和扩展内容。

使用 Vista Bridge 示例库,对 TaskDialogIndirect 的内部 API 调用封装在 PInvoke 中:

```
[DllImport(ExternDll.ComCtl32, CharSet = CharSet.Auto, SetLastError = true)]
internal static extern HRESULT TaskDialogIndirect(
    [In] NativeMethods.TASKDIALOGCONFIG pTaskConfig,
    [Out] out int pnButton,
    [Out] out int pnRadioButton,
    [Out] out bool pVerificationFlagChecked);
```

TaskDialogIndirect() 的第一个参数定义为 TASKDIALOGCONFIG 类,它映射了内部 API 调用的相同结构:



```
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Auto, Pack = 4)]
internal class TASKDIALOGCONFIG
{
    internal uint cbSize;
    internal IntPtr hwndParent;
    internal IntPtr hInstance;
    internal TASKDIALOG_FLAGS dwFlags;
    internal TASKDIALOG_COMMON_BUTTON_FLAGS dwCommonButtons;
    [MarshalAs(UnmanagedType.LPWStr)]
    internal string pszWindowTitle;
    internal TASKDIALOGCONFIG_ICON_UNION MainIcon;
    [MarshalAs(UnmanagedType.LPWStr)]
    internal string pszMainInstruction;
    [MarshalAs(UnmanagedType.LPWStr)]
    internal string pszContent;
    internal uint cButtons;
    internal IntPtr pButtons; // Ptr to TASKDIALOG_BUTTON structs
    internal int nDefaultButton;
    internal uint cRadioButtons;
    internal IntPtr pRadioButtons; // Ptr to TASKDIALOG_BUTTON structs
    internal int nDefaultRadioButton;
    [MarshalAs(UnmanagedType.LPWStr)]
    internal string pszVerificationText;
    [MarshalAs(UnmanagedType.LPWStr)]
    internal string pszExpandedInformation;
    [MarshalAs(UnmanagedType.LPWStr)]
    internal string pszExpandedControlText;
    [MarshalAs(UnmanagedType.LPWStr)]
    internal string pszCollapsedControlText;
    internal TASKDIALOGCONFIG_ICON_UNION FooterIcon;
    [MarshalAs(UnmanagedType.LPWStr)]
    internal string pszFooter;
    internal PFTASKDIALOGCALLBACK pfCallback;
    internal IntPtr lpCallbackData;
    internal uint cxWidth;
}
```

Vista Bridge 中用于显示任务对话框的公共类是 `TaskDialog`。要显示简单的任务对话框，只需调用静态方法 `Show()`。简单的对话框如图 C-6 所示。

```
TaskDialog.Show("Simple Task Dialog");
```

为了获得 `TaskDialog` 类的更多特性，应设置 `Caption`、`Content`、`StandardButtons` 和 `MainIcon` 属性。结果如图 C-7 所示。

```
TaskDialog dlg1 = new TaskDialog();
dlg1.Caption = "Title";
dlg1.Content = "Some Information";
dlg1.StandardButtons = TaskDialogStandardButtons.OkCancel;
dlg1.MainIcon = TaskDialogStandardIcon.Information;
dlg1.Show();
```



图 C-6

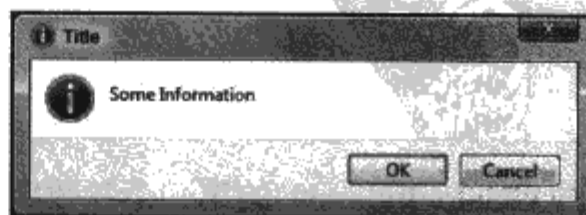


图 C-7

在任务对话框中，可以设置保护图标，它与命令链接一起显示。另外，还可以设置 `ExpansionMode` 属性来扩展它。使用 `TaskDialogExpandedInformationLocation` 枚举，可以指定是否应扩展内容或脚标。图 C-8 显示了折叠模式下的任务对话框，图 C-9 显示了展开模式的任务对话框。

```
TaskDialog dlg2 = new TaskDialog();
dlg2.Caption = "Title";
dlg2.Content = "Some Information";
dlg2.StandardButtons = TaskDialogStandardButtons.YesNo;
dlg2.MainIcon = TaskDialogStandardIcon.Shield;
dlg2.ExpandedText = "Additional Text";
dlg2.ExpandedControlText = "More information";
dlg2.CollapsedControlText = "Less information";
dlg2.ExpansionMode = TaskDialogExpandedInformationLocation.ExpandContent;
dlg2.FooterText = "Footer Information";
dlg2.FooterIcon = TaskDialogStandardIcon.Information;
dlg2.Show();
```

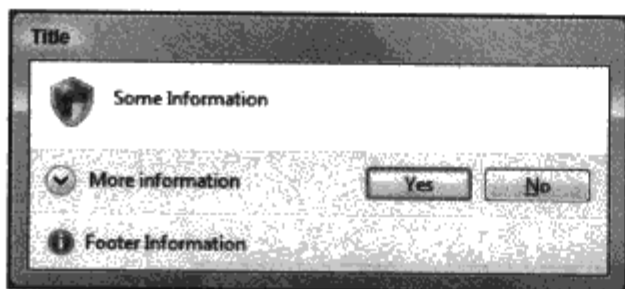


图 C-8

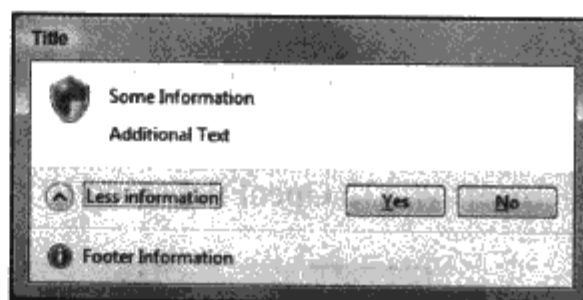


图 C-9

任务对话框也可以包含其他控件。在下面的代码中，创建了一个任务对话框，它包含两个单选按钮、一个命令链接和一个保护控件。上一节介绍了命令链接，其实，命令链接在任务对话框中使用得非常频繁。图 C-10 在内容区域中显示了带控件的任务对话框。当然，还可以将展开模式与控件组合起来。

```
TaskDialogRadioButton radiol = new TaskDialogRadioButton();
radiol.Name = "radiol";
radiol.Text = "One";
TaskDialogRadioButton radio2 = new TaskDialogRadioButton();
radio2.Name = "radio2";
radio2.Text = "Two";

TaskDialogCommandLink commandLink = new TaskDialogCommandLink();
commandLink.Name = "link1";
commandLink.ShowElevationIcon = true;
commandLink.Text = "Information";
commandLink.Instruction = "Sample Command Link";

TaskDialogMarquee marquee = new TaskDialogMarquee();
marquee.Name = "marquee";
marquee.State = TaskDialogProgressBarState.Normal;

TaskDialog dlg3 = new TaskDialog();
dlg3.Caption = "Title";
dlg3.Instruction = "Sample Task Dialog";

dlg3.Controls.Add(radiol);
dlg3.Controls.Add(radio2);
```

```
dlg3.Controls.Add(commandLink);
dlg3.Controls.Add.marquee);
dlg3.Show();
```

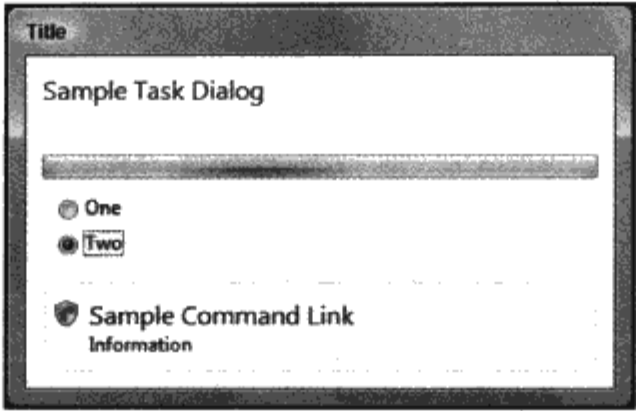


图 C-10

C.4.3 文件对话框

打开和保存文件的对话框有了变化。图 C-11 显示了传统的文件打开对话框，它封装在 Windows 窗体类 System.Windows.Forms.OpenFileDialog 和程序集 PresentationFramework 的 WPF 封装类 Microsoft.Win32.OpenFileDialog 中。

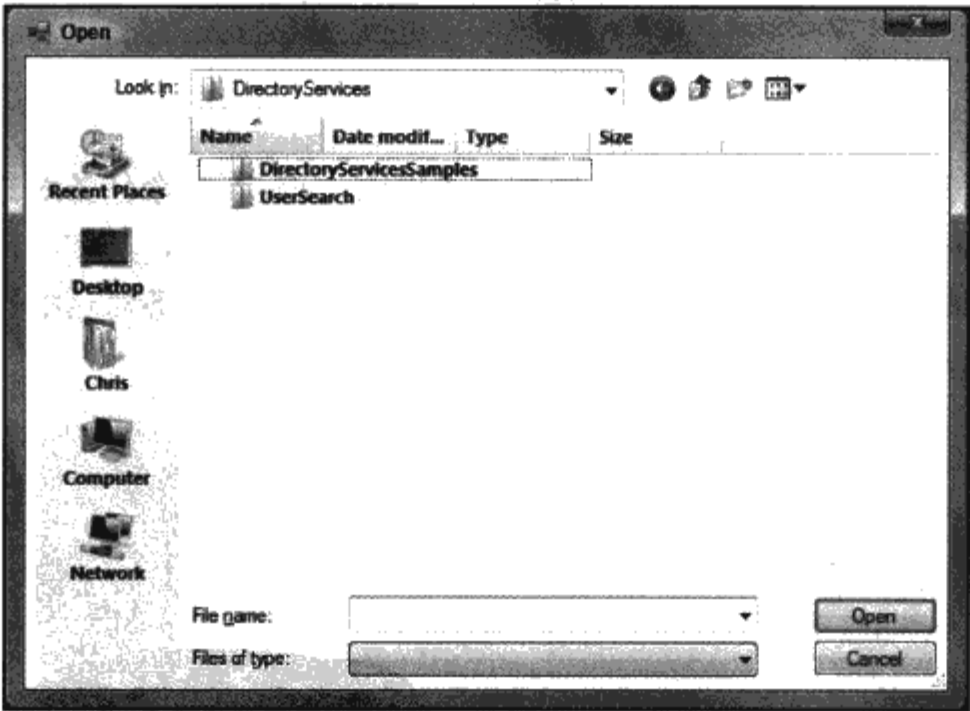


图 C-11

新的 Windows Vista 对话框如图 C-12 所示。这个对话框中的 Navigation、Details 和 Preview 面板可以在 Organize | Layout 菜单中配置。这个对话框还包含搜索功能，是可以完全定制的。在 Vista Bridge 库中，这个对话框封装在 CommonOpenFileDialog 类中。

```
CommonOpenFileDialog dlg = new CommonOpenFileDialog();
dlg.ShowDialog();
```

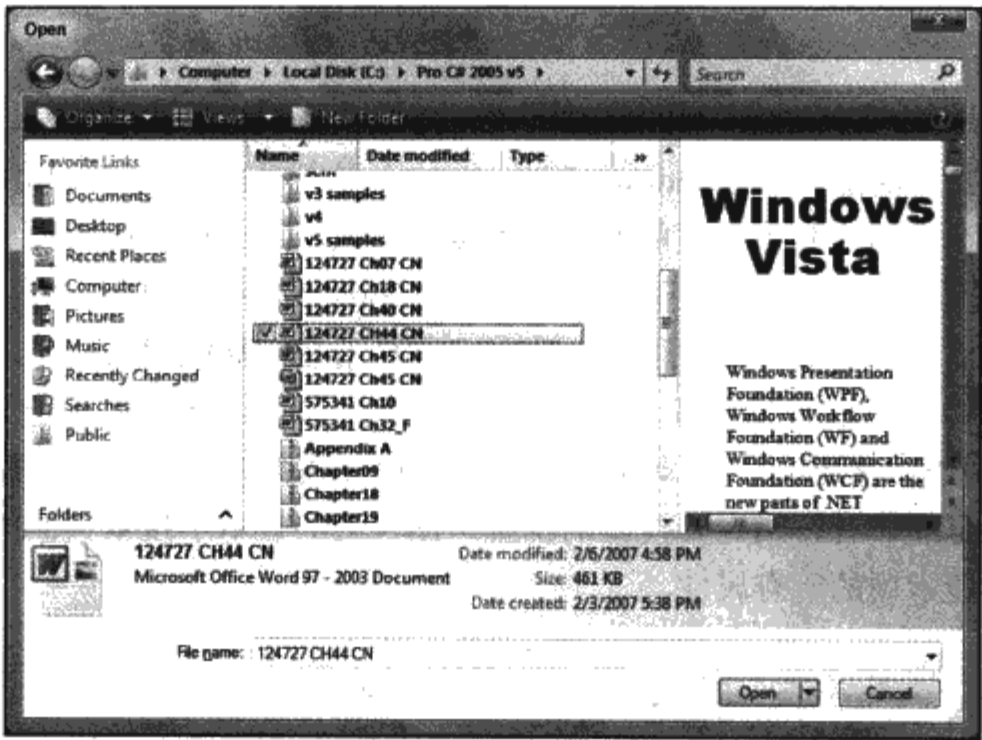


图 C-12

保存文件的新 Windows Vista 对话框也是可以定制的。它默认定义了折叠(如图 C-13 所示)和展开模式(如图 C-14 所示)。这个对话框封装在 CommonSaveDialog 类中。

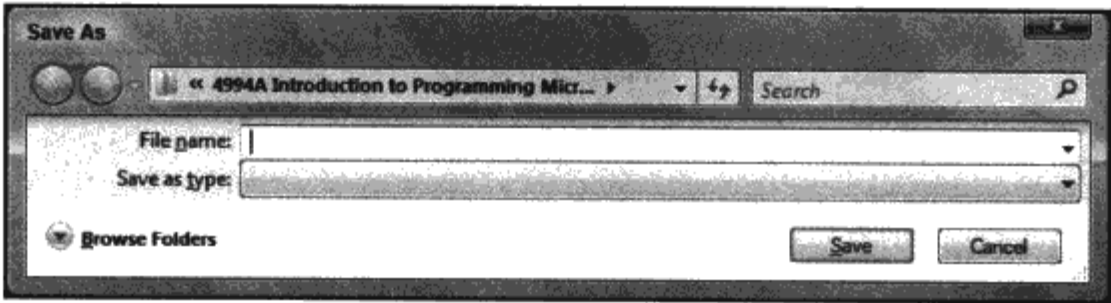


图 C-13

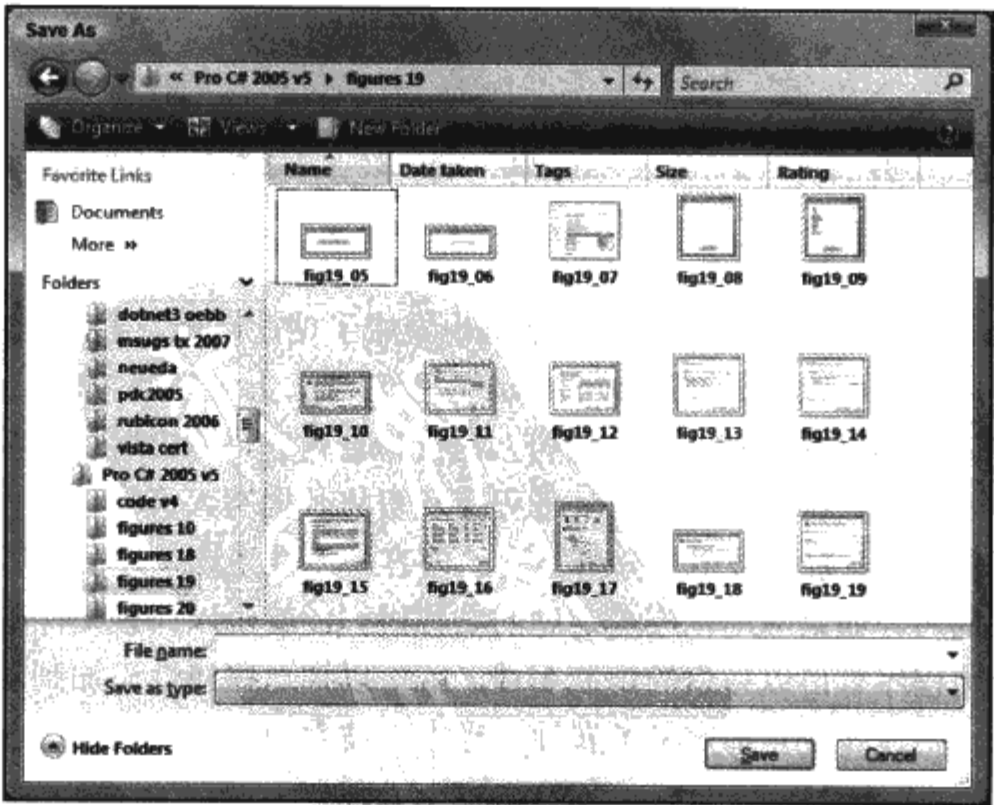


图 C-14



C.5 搜索

搜索是一个重要的功能，在 Windows Vista 中的许多应用程序、工具和实用程序中都提供了这个功能。Windows 的“开始”菜单就提供了搜索功能。这里可以搜索要启动的程序。在使用这个搜索功能一段时间后，就离不开它了。在 Windows XP 中，如果安装了许多应用程序，就很难从“开始”按钮中找到需要的程序。现在很容易用搜索功能解决这个问题。

选择 Search 菜单，可以查找许多文档，如电子邮件、文档、图片、音乐等。在简单的搜索中，只需在搜索框中输入搜索短语，即可在指定的位置查找数据项。高级搜索(如图 C-15 所示)可以输入名称、标记或作者，指定要搜索的位置。图 C-16 显示了搜索页面的详细视图，在其中可以选择显示被搜索项的所有属性。

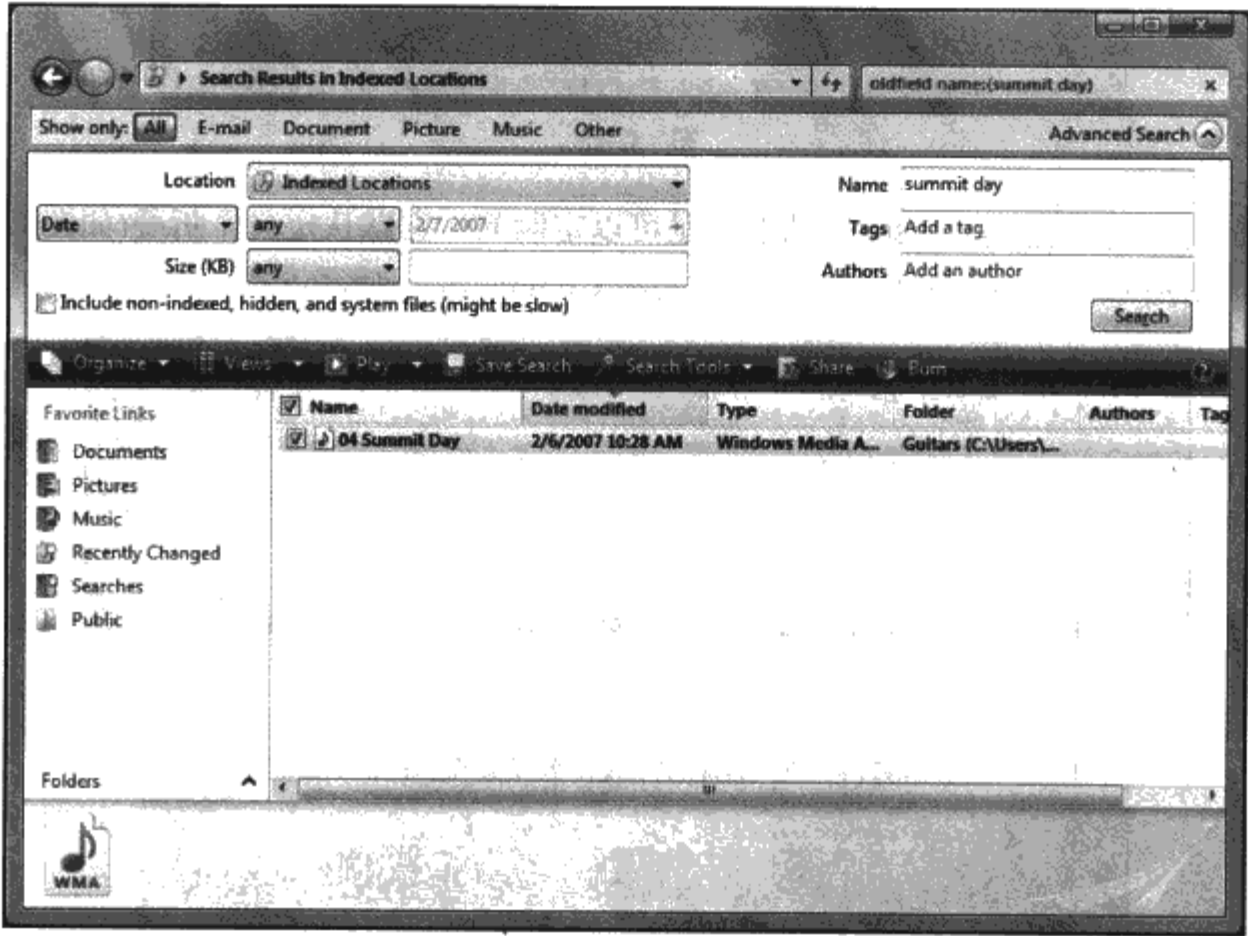


图 C-15

Windows Vista 的文件打开和文件保存对话框也集成了搜索功能。搜索功能可以集成到自己的应用程序中。应用程序可以充分利用 Windows 搜索功能。为了理解 Windows 搜索功能的架构，请参考图 C-17。搜索功能的核心是索引符，它会检查内容，将它写入内容索引。对于每个存储器(文件系统，MAPI)，都有一个协议处理程序负责将数据写入索引符。协议处理程序实现了接口 Ifilter，这个接口由索引符用于分析索引的内容。属性系统描述了可以搜索的属性。属性通过属性模式来描述。如果应用程序有定制的文件格式，就可以为该文件格式实现一个属性处理程序。如果应用程序有可以搜索的定制属性，就可以把这些属性添加到属性系统中。一般文件、Office 文档、图片和视频都定义了属性。当内容被索引时，就调用属性处理程序，来分析内容的属性。



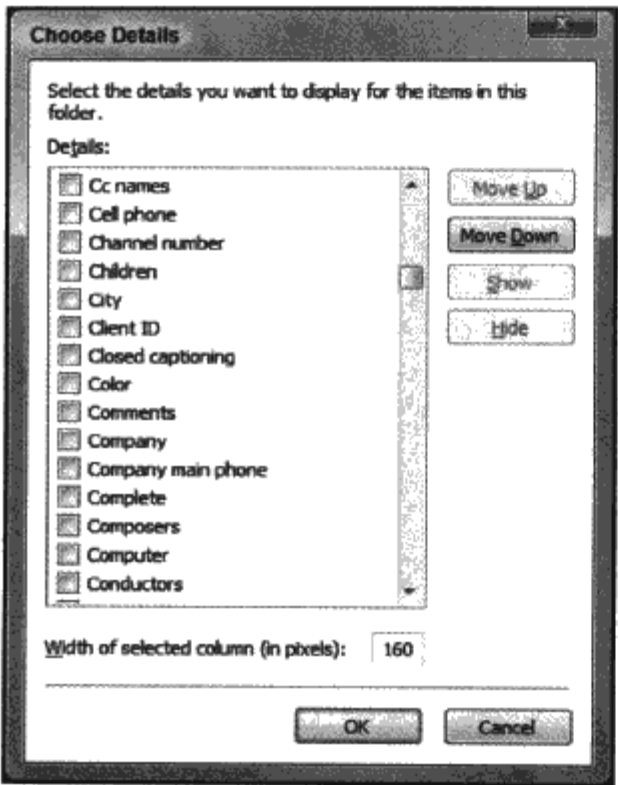


图 C-16

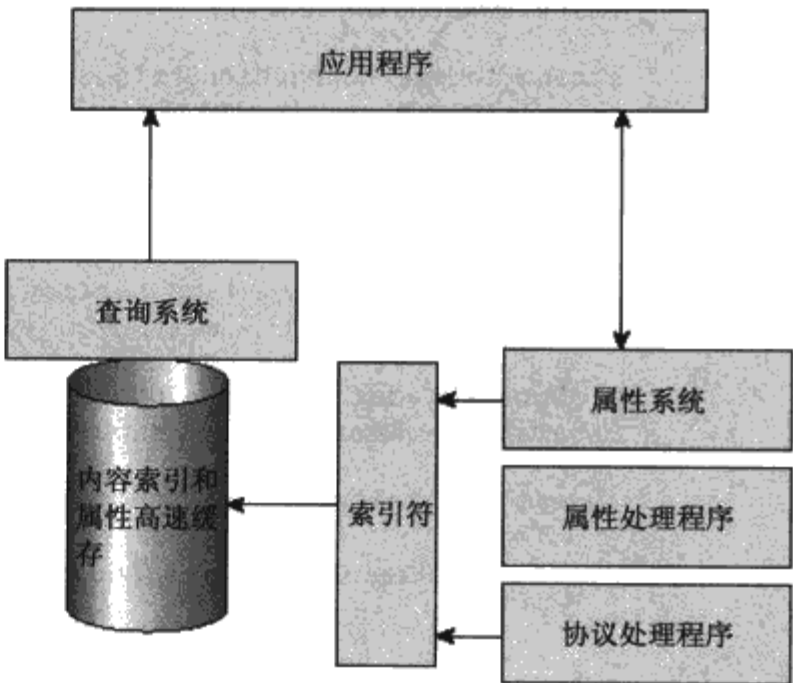


图 C-17

下面将搜索功能内置于应用程序，以利用查询系统。

C.5.1 OLE DB 提供程序

使用 OLE DB 提供程序可以将搜索功能集成到应用程序中，以搜索索引中的项。创建一个简单的 Windows 窗体应用程序，其中的文本框允许用户输入一个查询，按钮控件用于启动查询，ListView 控件用于显示结果，如图 C-18 所示。将 ListView 控件的 View 属性改为 Details，显示用户输入到查询中的所有信息。

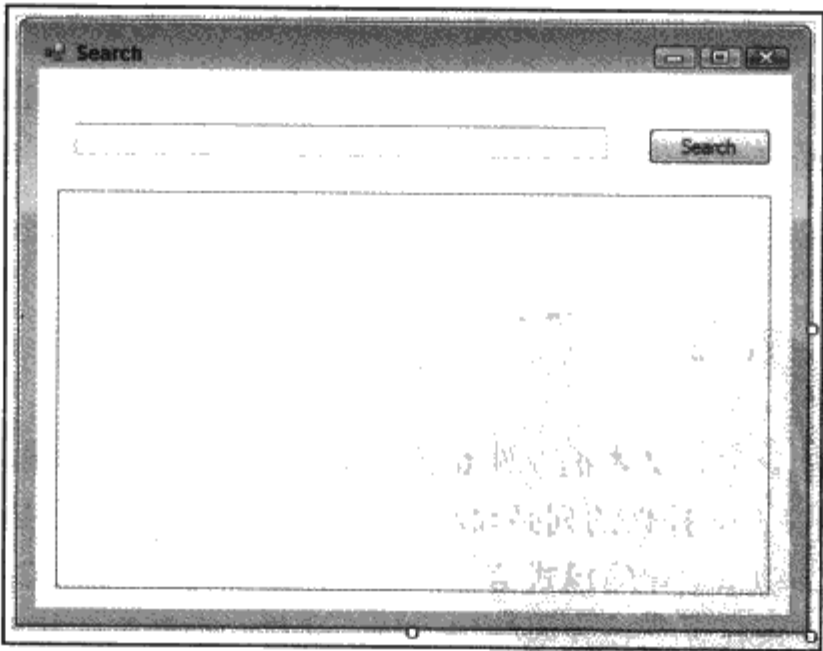


图 C-18

导入命名空间 System.Data.OleDb，在 Search 按钮的 Click 事件中输入如下代码：

```
private void buttonSearch_Click(object sender, EventArgs e)
```

```

{
    try
    {
        listViewResult.Clear();

        string indexerConnectionString = "provider=Search.CollatorDSO.1;" +
            "EXTENDED PROPERTIES='Application=Windows'";
        OleDbConnection connection = new OleDbConnection(
            indexerConnectionString);
        connection.Open();

        OleDbCommand command = connection.CreateCommand();
        command.CommandText = textBoxQuery.Text;

        OleDbDataReader reader = command.ExecuteReader();
        DataTable schemaTable = reader.GetSchemaTable();

        foreach (DataRow row in schemaTable.Rows)
        {
            listViewResult.Columns.Add(row[0].ToString());
        }

        while (reader.Read())
        {
            ListViewItem item = new ListViewItem(reader[0].ToString());
            for (int i = 1; i < reader.FieldCount; i++)
            {
                item.SubItems.Add(reader[i].ToString());
            }
            listViewResult.Items.Add(item);
        }

        connection.Close();
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}

```

下面详细分析代码。索引符给 OLE DB 提供程序提供了 Search.CollatorDSO。在 OLE DB 连接字符串中，可以传送这个提供程序的信息，打开与索引符的连接。

```

string indexerConnectionString = "provider=Search.CollatorDSO.1;" +
    "EXTENDED PROPERTIES='Application=Windows'";
OleDbConnection connection = new OleDbConnection(
    indexerConnectionString);
connection.Open();

```

与索引符一起使用的查询从文本框控件 textBoxQuery 中读取。在编译期间，不知道用户会选择什么属性，所以 ListView 控件的列必须动态添加。OleDbDataReader 的 GetSchemaTable() 方法返回与查询相关的、动态创建的模式信息。每一行都描述了 SELECT 语句中的一项，其中第一列给出了项的名称。迭代返回模式中的每一行，就将一个新列添加到 ListView 控件中，这一列的标题设置为项的名称。

```

OleDbCommand command = connection.CreateCommand();
command.CommandText = textBoxQuery.Text;

OleDbDataReader reader = command.ExecuteReader();

```

```

DataTable schemaTable = reader.GetSchemaTable();
foreach (DataRow row in schemaTable.Rows)
{
    listViewResult.Columns.Add(row[0].ToString());
}

```

接着读取 OleDbDataReader 的每一行。第一列创建了一个新的 ListViewItem。结果集中的每一列都会添加一个子项，与 ListView 的信息一起显示。

```

while (reader.Read())
{
    ListViewItem item = new ListViewItem(reader[0].ToString());
    for (int i = 1; i < reader.FieldCount; i++)
    {
        item.SubItems.Add(reader[i].ToString());
    }

    listViewResult.Items.Add(item);
}

```

现在可以启动应用程序，输入一个查询，所得的结果如图 C-19 所示。

```

SELECT System.ItemName, System.ItemTitle, System.Size FROM SYSTEMINDEX
WHERE System.Size > 1024

```

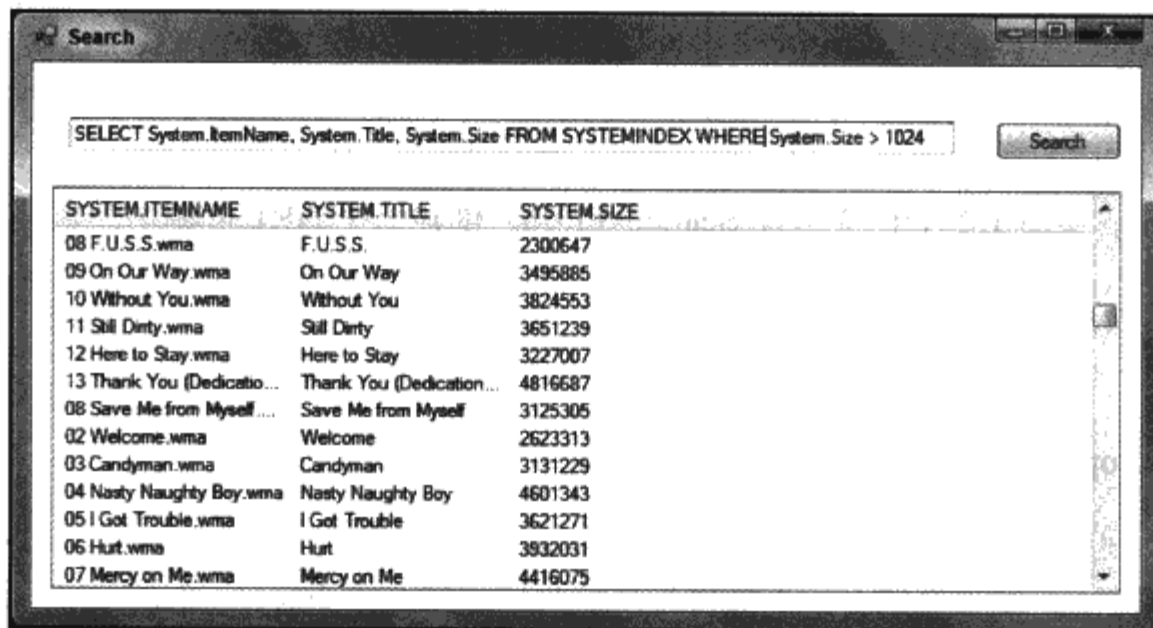


图 C-19

在查询的 SELECT 语句中，指定了应返回的属性。System.ItemName、System.ItemTitle 和 System.Size 是预定义的属性，其他预定义的属性可参见 MSDN，Windows Desktop Search 3.0 属性可参见 TechNet 文档。一般文件属性有 System.Author、System.Category、System.Company、System.DateCreated、System.DateModified、System.FileName、System.ItemName、System.ItemUrl 和 System.Keywords。对于音频文件、数字照片、图像文件、媒体文件、Office 文档、音乐文件和 Outlook 日历项，还定义了其他属性，例如 System.Photo.Orientation、System.Photo.DateTaken、System.Music.Artist、System.Music.BeatsPerMinute、System.Music.Mood、System.Calender.Location、System.Calender.Duration。

使用 WHERE 子句可以定义谓词，例如用于比较字面量值的 <、>、= 和 LIKE；用于全文搜索的 CONTAINS 和 FREETEXT，以及指定搜索深度的 SCOPE 和 DIRECTORY。

### C.5.2 高级查询语法

用户在搜索时可能不希望指定像上例那样的 SELECT 语句，此时可以创建一个用户界面，让用户指定特定的项，以编程方式建立 SELECT 语句。让用户执行自己的查询的另一种方式是使用 Advanced Query Syntax(AQS)。

Advanced Query Syntax 可以指定搜索项，根据属性限定搜索。例如，查询 Wrox date: past week 搜索包含字符串 Wrox、上个星期修改的所有项。Wrox date: past week kind: documents 进一步将搜索限定为只接收文档。

下面的示例说明了如何限定搜索：

- 可以定义存储器，来限定搜索的范围。例如，store:outlook 仅在 Outlook 中搜索。store:file 在文件系统中搜索。
- 使用搜索功能，可以指定结果中应包含什么类型的项，例如 kind:text, kind:tasks, kind:contacts, kind:emails 和 kind:folders。
- 布尔操作符可用于限定搜索。例如，Wrox OR Wiley 中的 OR 操作符。date:>11/25/07 搜索 2007 年 11 月 25 日之后的项。在两个日期之间的项可以使用 11/25/06..11/27/07 来搜索。
- 可以使用一些项的属性来搜索。例如，webpage:www.wrox.com, birthday:2/14/65, firstname:Christian。

不必将 AQS 手工转换为 SELECT 查询，有一个 COM 对象可以完成这个任务。在 Windows SDK Lib 目录中，有一个文件 SearchAPI.tlb。它是一个类型库，描述了用于进行 AQS 转换的 COM 对象。通过 COM Interop，可以在 .NET 中使用 COM 对象。

使用 tlbimp 实用工具导入类型库 SearchAPI.tlb，创建一个 .NET 可调用的封装器：

```
tlbimp c:\Program Files\Microsoft SDKs\Windows\v6.0\Lib\SearchAPI.tlb
/out:Interop.SearchAPI.dll
```

提示：

COM Interop 详见第 24 章。

在前面创建的 Windows 窗体项目中引用生成的 interop 程序集，就可以在 .NET 应用程序中使用 SearchAPI 了。因为类型库导入器用生成的程序集定义了 Interop.SearchAPI 命名空间，所以从应用程序中导入这个命名空间，给 Windows 窗体类添加 GetSql() 方法。

类 CSearchManager、CSearchCatalogManager 和 CSearchQueryHelper 在 tlbimp 实用工具中生成，以调用 COM 对象。GetCatalog() 方法定义了所查询的类别，并返回 catalog-Manager。有了 catalogManager 实例，就从 GetQueryHelper() 方法中返回查询帮助对象。将一个 AQS 字符串传送给 GenerateSQLFromUserQuery() 方法，会返回一个 SELECT 查询，该查询可以用 OLE DB 提供程序执行：

```
private string GetSql(string aqs)
{
    CSearchManager searchManager = new CSearchManager();
    CSearchCatalogManager catalogManager =
        searchManager.GetCatalog("SystemIndex");
    CSearchQueryHelper queryHelper = catalogManager.GetQueryHelper();
    return queryHelper.GenerateSQLFromUserQuery(aqs);
}
```

现在，还需要修改按钮控件的 Click 处理程序，以调用 GetSql() 方法，将 AQS 转换为应用

程序使用的 SELECT 查询。

```
private void buttonSearch_Click(object sender, EventArgs e)
{
    try
    {
        listViewResult.Clear();

        string indexerConnectionString = "provider=Search.CollatorDSO.1;" +
            "EXTENDED PROPERTIES='Application=Windows'";
        OleDbConnection connection = new
            OleDbConnection(indexerConnectionString);
        connection.Open();
        OleDbCommand command = connection.CreateCommand();
        command.CommandText = GetSql(textBoxQuery.Text);
        OleDbDataReader reader = command.ExecuteReader();

        //...
```

启动应用程序，传送一个 AQS 查询，如图 C-20 所示。

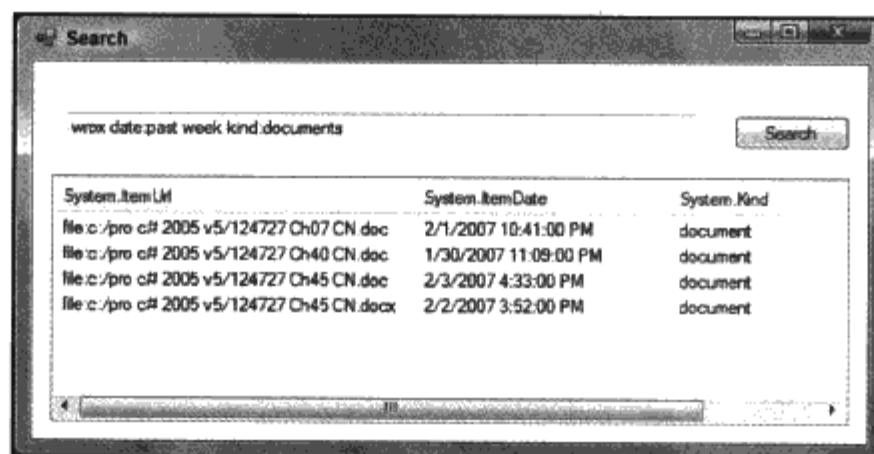


图 C-20

## C.6 小结

本章介绍了 Windows Vista 和 Windows Server 2008 中的各种特性，这些特性对应用程序的开发非常重要。

Microsoft 多年来制定了规则，指出非管理应用程序不需要管理权限。许多应用程序都没有遵循这个规则，所以操作系统现在限制了 UAC。用户必须明确地给应用程序提供管理权限。本章介绍了这些以及文件夹和注册表虚拟化如何影响应用程序。

本章还介绍了 Windows Vista 中的几个新对话框，以获得更好的用户交互操作，包括新的文件打开和文件保存对话框，替代消息框的新任务对话框，以及扩展了按钮控件的命令链接。

本章还学习了 Windows 查询系统，其中的 Advanced Query Syntax 和可扩展的属性系统能将搜索功能集成到应用程序中。

其他章节还介绍了只能用于 Windows Vista 和 Windows Server 2008 的更多特性：

- 第 18 章讨论了新的事件记录特性 Event Tracing for Windows (ETW)
- 第 20 章给出了 Cryptography Next Generation (CNG) 的信息，这是一个新的 Crypto API。
- 第 22 章介绍了基于文件和基于注册表的事务处理。
- 第 42 章使用 Windows Activation Services (WAS) 存储 WCF 服务。