

# ASP.NET3.5

## 开发大全



编者：

ASP.NET 是微软力推的 Web 开发编程技术  
也是当今最热门的 Web 开发编程之一

## 前言

随着互联网的不断发展和平台的多样性，越来越多的 **Web** 开发技巧呈现在用户面前，也是由于互联网的不断发展，越来越多的普通用户进入了互联网的范围开始了网络生活，这些网络生活随时随地的伴随着我们的生活，当我们使用银行的取款机进行取款时，我们就在与互联网打着交到，当我们收发电子邮件，在互联网上聊 **QQ**，同样也是在与互联网打着交到。在这些有趣的应用中，通常是通过一些 **Web** 编程语言进行实现的，这些语言包括 **ASP.NET**、**ASP**、**PHP** 等。**Web** 开发技巧不断的完善，更多更加丰富的应用程序也随之诞生，**ASP.NET** 使用 **.NET** 平台进行 **Web** 应用程序的开发有着先天性的优势，开发人员能够快速的使用 **ASP.NET** 提供的控件和开发方法进行复杂的应用程序开发，同时 **ASP.NET** 还为未来的云计算、多核化和多平台提供了基础，也为移动设配应用程序编程提供了保障。

为了方便广大读者学习，作者花费半年时间编制这本书，本书全面地介绍了 **ASP.NET** 技术，并介绍了能够与 **ASP.NET** 进行融合和跨平台的 **Web Server**、**WCF**、**WPF** 等，本书还附带大量的实例以及详细的注释，方便初学者进行深入学习。在学习完本书之后，读者能够具备基本的 **ASP.NET** 应用程序开发基础。

## 本书的特点

### 1. 循序渐进，深入浅出

为了能够方便读者的学习，本书前面几个章节详细的讲解了 **ASP.NET** 开发工具的安装，数据库系统的安装以及 **ASP.NET** 的基本知识。**ASP.NET** 使用的是面向对象的思想进行应用程序开发，本书还详细的讲解了面向对象的概念以及最新的开发模型。

### 2. 技术全面，内容充实

**ASP.NET** 应用程序的开发会遇到诸多问题，本书着手实际开发经验，在 **ASP.NET** 应用程序开发中详细的讲解了如何进行高效的 **ASP.NET** 应用程序开发，不仅如此，本书还详细的讲解了如何使用现有的互联网上的优秀的开源项目进行应用程序开发以提高开发效率，同时，读者还能够通过了解简单易懂的开源项目深入学习 **ASP.NET** 应用程序开发。

### 3. 分类讲解，理解深刻

本书通过将一些固定的知识进行分类讲解，举一反三，在本书的控件篇中，主要讲解基础控件和若干高级控件以及网站应用程序的配置方法，在数据篇中，详细的讲解数据源控件和数据绑定控件，以便读者能够详细的对知识进行分类，举一反三。

### 4. 案例精讲，深入剖析

在 **.NET** 应用平台下进行应用程序开发，无论是制作网站应用程序还是制作 **Windows** 应用程序都非常的简单，并且制作不同的应用程序所需要的知识也基本相同，本章在知识章节中配备了详细的例子进行讲解，包括 **MVC** 框架应用程序，**WCF** 以及 **WPF** 等，在本书的模块章节中，详细的讲解了 **ASP.NET** 应用程序模块开发的流程以及 **ASP.NET** 应用程序开发的技巧和规范，能够帮助读者学习到规范的应用程序开发技巧。

### 6. 最新技术前瞻

在 **.NET** 应用平台下进行应用程序开发，无需学习过多的新知识，包括 **MVC**、**WCF**、**WPF** 等应用程序开发都是基于 **.NET** 平台的，开发人员能够使用相同的开发方法进行不同的应用程序开发。本书详细的介绍了最新的技术以及技术走向，以便读者能够快速的为最新的技术做好准备而无需担心技术的淘汰。



### 7. 规范的开发，更多的技巧

本书在实例章节中，详细的介绍了如何进行规范的应用程序开发，包括设计需求分析文档以及编写类图等，规范的应用程序开发是非常重要的，同时本书还介绍了 **ASP.NET** 应用程序开发技巧，以便读者能够规范的、快速的编写高效的代码。

### 8. 配有多媒体光盘，加速学习

本书在光盘中配备了大量的实例，读者能够通过阅读实例代码进行实例的运行和学习，光盘中的实例与本书中的示例一一对应，读者能够进行书本的阅读并同时运行实例了解程序是如何运行的。

### 9. 提供完善的售后服务

为了方便读者的学习，读者可以访问作者的博客获取更多的帮助，作者还会在博客中不定期的发布视频和图文教程以便读者学习到本书之外的更多的 **ASP.NET** 应用程序和开发技巧，读者可以在 <http://www.shangducms.com/blog/uid115.html> 进入作者的博客查阅更多 **ASP.NET** 应用程序的开发技巧和进行本书的错误的反馈。

## 本书的内容

第 1 章：第一章详细的介绍了 **ASP.NET** 基础以及 **.NET** 平台的历史以及前瞻，在第一章中，读者能够学会如何安装 **Visual Studio 2008** 以及 **SQL Server 2005** 进行应用程序开发基础，第一章还讲解了开发环境的使用技巧以方便读者进行高效的应用程序开发。

第 2 章：在进行 **ASP.NET** 应用程序开发前，首先需要了解 **ASP.NET** 应用程序开发的最主要的编程语言 **C#**，由于 **ASP.NET** 应用程序是基于面向对象的思想的，所以 **C#** 编程语言也包括多种面向对象的特性，包括多态和继承等，本章讲解了 **C#** 编程语言的基本结构和技巧以便读者能够了解面向对象。

第 3 章：在了解了 **C#** 编程语言后，就需要深入的了解什么是面向对象，面向对象是应用程序开发中非常重要的思想，本章详细的讲解了 **C#** 编程语言中关于面向对象的技巧以及特性，以便读者能够高效的开发 **ASP.NET** 应用程序。

第 4 章：详细的介绍了 **ASP.NET** 网页代码模型和生命周期，了解网页代码模型和生命周期能够帮助读者高效的创建 **ASP.NET** 应用页面。

第 5 章：详细的介绍了 **ASP.NET** 应用程序中提供的控件，控件为开发人员提供了高效的应用程序开发方法，开发人员无需专业的知识就能够实现复杂的应用操作。

第 6 章：着重讲解了 **ASP.NET** 高级控件中的登陆控件的使用，并讲解了 **ASP.NET** 网站管理工具的使用方法和技巧。

第 7 章：主要讲解了数据库基础以及如何进行数据库中的相应操作，包括数据库的创建、数据库的删除、数据库表的创建以及数据库表的删除。数据库应用在当今的应用程序开发中必不可少，本章详细的讲解了如何进行数据库的开发。

第 8 章：主要讲解了如何使用 **ASP.NET** 提供的数据源控件和数据绑定控件进行高效的应用程序开发和数据开发。

第 9 章：在使用数据源控件和数据绑定控件进行数据操作时，并不能够非常灵活的进行应用程序开发，本章讲解了除了使用控件以外的使用类成员进行数据库开发。

第 10 章：本章详细的介绍了如何使用类成员进行不同的数据源的访问，这些数据源包括 **MySQL**、**Access**、**Excel**、**Txt** 以及 **SQLite**。

第 11 章：本章主要讲解了用户控件和自定义控件的编程方法，以便开发人员能够高效的进行功能的封装。

第 12 章：重点介绍了 **CSS** 和母版页对 **ASP.NET** 应用程序进行样式控制的方法和技巧。

第 13 章：本章详细的讲解了 **ASP.NET** 应用程序的内置对象和缓存等概念，**ASP.NET** 的内置对象维护了 **Web** 应用程序的状态，而通过使用缓存能够极大的提高应用程序的抗压性，提高网站的访问效率。

第 14 章：本章详细的讲解了 **ASP.NET** 应用程序和 **Web Service** 的概念，为了能够让读者更容易的理解

**Web Service**，本章详细的介绍了 XML 以及 XML 类成员是如何进行 XML 文件的读取和编写的。

第 15 章：介绍了如何使用 ASP.NET 中 .NET 应用程序框架的 GDI+ 进行 Web 应用程序的图形图像编程。

第 16 章：随着互联网的不断发展，无刷新应用 AJAX 也越来越多的被人们所关注，本章详细的介绍了如何在 ASP.NET 2.0 和 3.5 中进行 AJAX 应用程序的开发。

第 17 章：介绍了 ASP.NET MVC 框架的基本知识，ASP.NET MVC 框架是下一代 ASP.NET 应用程序框架，了解 ASP.NET MVC 基本知识能够为下一代 ASP.NET 应用程序开发做好准备。

第 18、19 章：介绍了 WCF 和 WPF 应用程序的开发，WCF 和 WPF 是 .NET 3.5 应用程序框架的新特性，使用 WCF 能够将桌面应用和 Web 应用进行整合，而 WPF 提供了高效的绚丽的桌面应用开发解决方案。

第 20 章：介绍了 LINQ 的基本知识以及 Lambda 表达式基础。

第 21 章：介绍了如何使用 LINQ 进行数据库操作，这其中包括数据的插入、更新和删除等。

第 22、23、24、25、26、27、28、29 章：最后篇幅通过多个模块以及综合实例开发和讲解，让读者有实际项目的体会，从而能够深刻的了解本书前面的知识并达到实战的能力。

## 适合的读者

- 网页专业设计人员
- 网页维护人员
- 网页制作爱好者
- 大中专院校的学生
- 社会培训学生
- ASP.NET 应用程序初学者
- .NET 应用开发入门者

编者

2008 年 11 月

## 内容提要

ASP.NET 是微软力推的 Web 开发编程技术，也是当今最热门的 Web 开发编程之一。本书深入浅出，循序渐进讲授读者如何使用 ASP.NET 进行系统开发。全书内容包括了解 ASP.NET、认识 C# 3.5、认识面向对象、ASP.NET 基础控件、ASP.NET 高级控件、数据库与 ADO.NET、数据库基础、ASP.NET 操作数据库、ASP.NET 访问其他数据源、ASP.NET MVC、LINQ 及 Lambda 表达式、WCF 应用开发、WPF 应用开发以及图形图像编程等内容。为了便于读者学习和理解 ASP.NET 的知识，本书最后几章进行了不同的小型模块的开发，以便读者能够深入的了解 ASP.NET 技术的基础开发，在了解了基本的模块开发后，还包括两章进行较大、较完整的系统开发。具体讲解了用户注册模块、登陆模块、投票模块、聊天模块以及留言本系统和校友录系统。

本书适合广大 Web 网站开发人员、网站管理维护人员和大专院校学生阅读，尤其是有一定 Internet /Intranet 编程技术的人员，同样本书也适合 .NET 平台的初学者以及热爱 .NET 技术的入门人员。

## 目 录

### 第一篇 .NET 基础

#### 第 1 章 认识 ASP.NET 3.5

- 1.1 什么是 ASP.NET
  - 1.1.1 .NET 历史与展望
  - 1.1.2 ASP.NET 与 ASP
  - 1.1.3 ASP.NET 开发工具
  - 1.1.4 ASP.NET 客户端
  - 1.1.5 ASP.NET 3.5 新增控件
  - 1.1.6 ASP.NET 3.5 AJAX
- 1.2 .NET 应用程序需框架
  - 1.2.1 什么是.NET 应用程序框架
  - 1.2.2 公共语言运行时 (CLR)
  - 1.2.3 .NET Framework 类库
- 1.3 安装 Visual Studio 2008
  - 1.3.1 安装 Visual Studio 2008
  - 1.3.2 主窗口
  - 1.3.3 文档窗口
  - 1.3.4 工具箱
  - 1.3.5 解决方案管理器
  - 1.3.6 属性窗口
  - 1.3.7 错误列表窗口
- 1.4 安装 SQL Server 2005
- 1.5 ASP.NET 应用程序基础
  - 1.5.1 创建 ASP.NET 应用程序
  - 1.5.2 运行 ASP.NET 应用程序
  - 1.5.3 编译 ASP.NET 应用程序
- 1.6 小结

#### 第 2 章 C# 3.0 程序设计基础

- 2.1 C#程序
  - 2.1.1 C#程序的结构
  - 2.1.2 C# IDE 的代码设置
- 2.2 变量
  - 2.2.1 定义
  - 2.2.2 值类型
  - 2.2.3 引用类型

- 2.3 变量规则
  - 2.3.1 命名规则和命名习惯
  - 2.3.2 声明并初始化变量
  - 2.3.3 数组
  - 2.3.4 声明并初始化字符串
  - 2.3.5 操作字符串
  - 2.3.6 创建和使用常量
  - 2.3.7 创建并使用枚举
  - 2.3.8 类型转换
- 2.4 编写表达式
  - 2.4.1 表达式和运算符
  - 2.4.2 运算符的优先级
- 2.5 使用条件语句
  - 2.5.1 **if** 语句的使用方法
  - 2.5.2 **switch** 选择语句的使用
- 2.6 使用循环语句
  - 2.6.1 **for** 循环语句
  - 2.6.2 **while** 循环语句
  - 2.6.3 **do while** 循环语句
  - 2.6.4 **foreach** 循环语句
- 2.7 异常处理语句
  - 2.7.1 **throw** 异常语句
  - 2.7.2 **try-catch** 异常语句
  - 2.7.3 **try-finally** 异常语句
  - 2.7.4 **try-catch-finally** 异常语句
- 2.8 小结

## 第 3 章 面向对象设计基础

- 3.1 什么是面向对象
  - 3.1.1 传统的面向过程
  - 3.1.2 面向对象的概念
  - 3.1.3 面向组件的概念
- 3.2 面向对象的 **C#** 实现
  - 3.2.1 定义
  - 3.2.2 创建一个类和其方法
  - 3.2.3 类成员
  - 3.2.4 构造函数和析构函数
- 3.3 对象的生命周期
  - 3.3.1 类成员的访问
  - 3.3.2 类的类型
  - 3.3.3 **.NET** 的垃圾回收机制
- 3.4 使用命名空间
  - 3.4.1 为什么要用命名空间
  - 3.4.2 创建命名空间
  - 3.4.3 分层设计中使用命名空间
- 3.5 类的方法
  - 3.5.1 编写方法

- 3.5.2 给方法传递参数
- 3.5.3 通过引用来传递参数
- 3.5.4 方法的重载
- 3.6 封装
  - 3.6.1 为什么要封装
  - 3.6.2 类的设计
- 3.7 属性
  - 3.7.1 语法
  - 3.7.2 只读/只写属性
- 3.8 继承
  - 3.8.1 继承的基本概念
  - 3.8.2 创建派生类
  - 3.8.3 对象的创建
  - 3.8.4 使用抽象类
  - 3.8.5 使用密封类
- 3.9 多态
  - 3.9.1 抽象方法
  - 3.9.2 覆盖
  - 3.9.3 虚方法的抽象类
  - 3.9.4 抽象属性
- 3.10 委托和事件
  - 3.10.1 委托
  - 3.10.2 声明事件
  - 3.10.3 引发事件
  - 3.10.4 订阅事件
  - 3.10.5 委托和事件
- 3.11 类命名
  - 3.11.1 命名空间的命名
  - 3.11.2 类的命名原则
  - 3.11.3 接口的命名原则
  - 3.11.4 属性的命名原则
  - 3.11.5 枚举的命名原则
  - 3.11.6 只读字段的命名原则
  - 3.11.7 参数名
  - 3.11.8 委托命名原则
- 3.12 小议设计模式
  - 3.12.1 什么是设计模式
  - 3.12.2 为什么要使用设计模式
  - 3.12.3 改装现有类
- 3.13 小结

## 第 4 章 ASP.NET 的网页代码模型及生命周期

- 4.1 ASP.NET 的网页代码模型
  - 4.1.1 创建 ASP.NET 网站
  - 4.1.2 单文件页模型
  - 4.1.3 代码隐藏页模型
  - 4.1.4 创建 ASP.NET Web Application



- 4.1.5 ASP.NET 网站和 ASP.NET 应用程序的区别
- 4.2 代码隐藏页模型的解释过程
- 4.3 代码隐藏页模型的事件驱动处理
- 4.4 ASP.NET 客户端状态
  - 4.4.1 视图状态
  - 4.4.2 控件状态
  - 4.4.3 隐藏域
  - 4.4.4 Cookie
  - 4.4.5 客户端状态维护
- 4.5 ASP.NET 页面生命周期
- 4.6 ASP.NET 生命周期中的事件
  - 4.6.1 页面加载事件 (Page\_PreInit)
  - 4.6.2 页面加载事件 (Page\_Init)
  - 4.6.3 页面载入事件 (Page\_Load)
  - 4.6.4 页面卸载事件 (Page\_Unload)
  - 4.6.5 页面指令
- 4.7 ASP.NET 网站文件类型
- 4.8 小结

## 第二篇 ASP.NET 窗体控件

### 第 5 章 Web 窗体的基本控件

- 5.1 控件的属性
- 5.2 简单控件
  - 5.2.1 标签控件 (Label)
  - 5.2.2 超链接控件 (HyperLink)
  - 5.2.3 图像控件 (Image)
- 5.3 文本框控件 (TextBox)
  - 5.3.1 文本框控件的属性
  - 5.3.2 文本框控件的使用
- 5.4 按钮控件 (Button, LinkButton, ImageButton)
  - 5.4.1 按钮控件的通用属性
  - 5.4.2 Click 单击事件
  - 5.4.3 Command 命令事件
- 5.5 单选控件和单选组控件 (RadioButton 和 RadioButtonList)
  - 5.5.1 单选控件 (RadioButton)
  - 5.5.2 单选组控件 (RadioButtonList)
- 5.6 复选框控件和复选组控件 (CheckBox 和 CheckBoxList)
  - 5.6.1 复选框控件 (CheckBox)
  - 5.6.2 复选组控件 (CheckBoxList)
- 5.7 列表控件 (DropDownList, ListBox 和 BulletedList)
  - 5.7.1 DropDownList 列表控件
  - 5.7.2 ListBox 列表控件
  - 5.7.3 BulletedList 列表控件

- 5.8 面板控件 (**Panel**)
- 5.9 占位控件 (**PlaceHolder**)
- 5.10 日历控件 (**Calendar**)
  - 5.10.1 日历控件的样式
  - 5.10.2 日历控件的事件
- 5.11 广告控件 (**AdRotator**)
- 5.12 文件上传控件 (**FileUpload**)
- 5.13 视图控件 (**MultiView** 和 **View**)
- 5.14 表控件 (**Table**)
- 5.15 向导控件 (**Wizard**)
  - 5.15.1 向导控件的样式
  - 5.15.2 导航控件的事件
- 5.16 XML 控件
- 5.17 验证控件
  - 5.17.1 表单验证控件 (**RequiredFieldValidator**)
  - 5.17.2 比较验证控件 (**CompareValidator**)
  - 5.17.3 范围验证控件 (**RangeValidator**)
  - 5.17.4 正则验证控件 (**RegularExpressionValidator**)
  - 5.17.5 自定义逻辑验证控件 (**CustomValidator**)
  - 5.17.6 验证组控件 (**ValidationSummary**)
- 5.18 导航控件
- 5.19 其他控件
  - 5.19.1 隐藏输入框控件 (**HiddenField**)
  - 5.19.2 图片热点控件 (**ImageMap**)
  - 5.19.3 静态标签控件 (**Lieral**)
  - 5.19.4 动态缓存更新控件(**Substitution**)
- 5.20 小结

## 第 6 章 Web 窗体的高级控件

- 6.1 登录控件
  - 6.1.1 登录控件 (**Login**)
  - 6.1.2 登录名称控件 (**LoginName**)
  - 6.1.3 登录视图控件 (**LoginView**)
  - 6.1.4 登录状态控件 (**LoginStatus**)
  - 6.1.5 密码恢复控件 (**PasswordRecovery**)
  - 6.1.6 密码更改控件 (**ChangePassword**)
  - 6.1.7 生成用户控件 (**CreateUserWizard**)
- 6.2 网站管理工具
  - 6.2.1 启动管理工具
  - 6.2.2 用户管理
  - 6.2.3 用户角色
  - 6.2.4 访问规则管理
  - 2.6.5 应用程序配置
- 6.3 使用登录控件
  - 6.3.1 生成用户控件 (**CreateUserWizard**)
  - 6.3.2 密码更改控件 (**ChangePassword**)
- 6.4 小结

### 第 7 章 数据库与 ADO.NET 基础

- 7.1 数据库基础
  - 7.1.1 结构化查询语言
  - 7.1.2 表和视图
  - 7.1.3 存储过程和触发器
- 7.2 使用 SQL Server 2005 管理数据库
  - 7.2.1 初步认识 SQL Server 2005
  - 7.2.2 创建数据库
  - 7.2.3 删除数据库
  - 7.2.4 备份数据库
  - 7.2.5 还原数据库
  - 7.2.6 创建表
  - 7.2.7 删除表
  - 7.2.8 创建数据库关系图
- 7.3 ADO.NET 连接 SQL 数据库
  - 7.3.1 ADO.NET 基础
  - 7.3.2 连接 SQL 数据库
  - 7.3.3 ADO.NET 过程
- 7.4 ADO 与 ADO.NET
  - 7.4.1 ADO 概述
  - 7.4.2 ADO.NET 与 ADO
- 7.5 ADO.NET 常用对象
- 7.6 Connection 连接对象
  - 7.6.1 连接 SQL 数据库
  - 7.6.2 连接 Access 数据库
  - 7.6.3 打开和关闭连接
- 7.7 DataAdapter 适配器对象
- 7.8 Command 执行对象
  - 7.8.1 ExecuteNonQuery 方法
  - 7.8.2 ExecuteNonQuery 执行存储过程
  - 7.8.3 ExecuteScalar 方法
- 7.9 DataSet 数据集对象
  - 7.9.1 DataSet 数据集基本对象
  - 7.9.2 DataTable 数据表对象
  - 7.9.3 DataRow 数据行对象
  - 7.9.4 DataView 数据视图对象
- 7.10 DataReader 数据访问对象
  - 7.10.1 DataReader 对象概述
  - 7.10.2 DataReader 读取数据库
  - 7.10.3 异常处理
- 7.11 连接池概述
- 7.12 参数化查询
- 7.13 小结

## 第 8 章 Web 窗体的数据控件

- 8.1 数据源控件
  - 8.1.1 SQL 数据源控件 (SqlDataSource)
  - 8.1.2 Access 数据源控件 (AccessDataSource)
  - 8.1.3 目标数据源控件 (ObjectDataSource)
  - 8.1.4 LINQ 数据源控件 (LinqDataSource)
  - 8.1.5 Xml 数据源控件 (XmlDataSource)
  - 8.1.6 站点导航控件 (SiteMapDataSource)
- 8.2 重复列表控件 (Repeater)
- 8.3 数据列表控件 (DataList)
- 8.4 数据列表控件 (GridView)
- 8.5 数据绑定控件 (FormView)
- 8.6 数据绑定控件 (DetailsView)
- 8.7 数据绑定控件 (ListView)
- 8.8 数据绑定控件 (DataPager)
- 8.9 小结

## 第 9 章 ASP.NET 操作数据库

- 9.1 使用 ADO.NET 操作数据库
  - 9.1.1 使用 ExecuteReader()操作数据库
  - 9.1.2 使用 ExecuteNonQuery()操作数据库
  - 9.1.3 使用 ExecuteScalar()操作数据库
  - 9.1.4 使用 ExecuteXmlReader()操作数据库
- 9.2 ASP.NET 创建和插入记录
  - 9.2.1 SQL INSERT 数据插入语句
  - 9.2.2 使用 Command 对象更新记录
  - 9.2.3 使用 DataSet 数据集插入记录
- 9.3 ASP.NET 更新数据库
  - 9.3.1 SQL UPDATE 数据更新语句
  - 9.3.2 使用 Command 对象更新记录
  - 9.3.3 使用 DataSet 数据集更新记录
- 9.4 ASP.NET 删除数据
  - 9.4.1 SQL DELETE 数据删除语句
  - 9.4.2 使用 Command 对象删除记录
  - 9.4.3 使用 DataSet 数据集删除记录
- 9.5 使用存储过程
  - 9.5.1 存储过程的优点
  - 9.5.2 创建存储过程
  - 9.5.3 调用存储过程
- 9.6 ASP.NET 数据库操作实例
  - 9.6.1 制作用户界面 (UI)
  - 9.6.2 使用 GridView 显示、删除、修改数据
  - 9.6.3 使用 DataList 显示数据
  - 9.6.4 DataList 分页实现
  - 9.6.5 使用 SQLHelper 操作数据库
- 9.7 小结

## 第 10 章 访问其他数据源

- 10.1 使用 ODBC .NET Data Provider
  - 10.1.1 ODBC .NET Data Provider 简介
  - 10.1.2 建立连接
- 10.2 使用 OLE DB.NET Data Provider
  - 10.2.1 OLE DB.NET Data Provider 简介
  - 10.2.2 建立连接
- 10.3 访问 MySql
  - 10.3.1 MySql 简介
  - 10.3.2 建立连接
- 10.4 访问 Excel
  - 10.4.1 Excel 简介
  - 10.4.2 建立连接
- 10.5 访问 txt
  - 10.5.1 使用 ODBE.NET Data Provider 连接 txt
  - 10.5.2 使用 OLE DB .NET Data Provider 连接 txt
  - 10.5.3 使用 System.IO 命名空间
- 10.6 访问 SQLite
  - 10.6.1 SQLite 简介
  - 10.6.2 SQLite 连接方法
- 10.7 小结

## 第四篇 ASP.NET 网络编程

## 第 11 章 用户控件和自定义控件

- 11.1 用户控件
  - 11.1.1 什么是用户控件
  - 11.1.2 编写一个简单的控件
  - 11.1.3 将 Web 窗体转换成用户控件
- 11.2 自定义控件
  - 11.2.1 实现自定义控件
  - 11.2.2 复合自定义控件
- 11.3 用户控件和自定义控件的异同
- 11.4 用户控件示例
  - 11.4.1 ASP.NET 登录控件
  - 11.4.2 ASP.NET 登录控件的开发
  - 11.4.3 ASP.NET 登录控件的使用
- 11.5 自定义控件实例
  - 11.5.1 ASP.NET 分页控件
  - 11.5.2 ASP.NET 分页控件的使用
- 11.6 小结

## 第 12 章 ASP.NET 的皮肤、主题和母版页

- 12.1 皮肤和主题
  - 12.1.1 CSS 简介
  - 12.1.2 CSS 基础
  - 12.1.3 CSS 常用属性
  - 12.1.4 将 CSS 应用在控件上
  - 12.1.5 主题和皮肤
  - 12.1.6 页面主题和全局主题
  - 12.1.7 应用和禁用主题
  - 12.1.8 用编程的方法控制主题
- 12.2 母版页
  - 12.2.1 母版页基础
  - 12.2.2 内容窗体
  - 12.2.3 母版页的运行方法
  - 12.2.4 嵌套母版页
- 12.3 Microsoft Expression 2
  - 12.3.1 Microsoft Expression 2 简介
  - 12.3.2 安装 Microsoft Expression 2
- 12.4 使用 Microsoft Expression Web 2 制作页面
  - 12.4.1 创建 ASPX 页面
  - 12.4.2 创建 CSS 层叠样式表
  - 12.4.3 创建框架集
- 12.5 小结

## 第 13 章 ASP.NET 内置对象,应用程序配置和缓存

- 13.1 ASP.NET 内置对象
  - 13.1.1 Request 传递请求对象
  - 13.1.2 Response 请求响应对象
  - 13.1.3 Application 状态对象
  - 13.1.4 Session 状态对象
  - 13.1.5 Server 服务对象
  - 13.1.6 Cookie 状态对象
  - 13.1.7 Cache 缓存对象
  - 13.1.8 Global.asax 配置
- 13.2 ASP.NET 应用程序配置
  - 13.2.1 ASP.NET 应用程序配置
  - 13.2.2 Web.config 配置文件
  - 13.2.3 ASP.NET 基本配置节
- 13.3 ASP.NET 缓存功能
  - 13.3.1 缓存概述
  - 13.3.2 页面输出缓存
  - 13.3.3 页面部分缓存
  - 13.3.4 应用程序数据缓存
  - 13.3.5 检索应用程序数据缓存对象
- 13.4 小结

## 第 14 章 ASP.NET XML 和 Web Service

- 14.1 XML 简介



- 14.2 读写 XML
  - 14.2.1 XML 与 HTML
  - 14.2.2 创建 XML 文档
  - 14.2.3 XML 控件
  - 14.2.4 XML 文件读取类 (XmlTextReader)
  - 14.2.5 XML 文件编写类 (XmlTextWriter)
  - 14.2.6 XML 文本文档类 (XmlDocument)
- 14.3 XML 串行化
  - 14.3.1 XmlSerializer 串行化类
  - 14.3.2 基本串行化
- 14.4 XML 样式表 XSL
  - 14.4.1 XSL 简介
  - 14.4.2 使用 XSLT
- 14.5 Web 服务 (Web Service)
  - 14.5.1 什么是 Web 服务
  - 14.5.2 Web 服务体系结构
  - 14.5.3 Web 服务协议栈
- 14.6 简单 Web Service 示例
- 14.7 自定义 Web Service
  - 14.7.1 创建自定义的 Web Service
  - 14.7.2 使用自定义的 Web Service
- 14.8 小结

## 第五篇 ASP.NET 3.5 高级编程

### 第 15 章 图形图像编程

- 15.1 图形图像基础
  - 15.1.1 图像布局
  - 15.1.2 GDI+简介
  - 15.1.3 绘制线条示例
  - 15.1.4 .NET Framework 绘图类
- 15.2 图形编程
  - 15.2.1 Graphics 类
  - 15.2.2 绘制基本图形
  - 15.2.3 图形绘制实例
- 15.3 绘制文字特效
  - 15.3.1 投影特效
  - 15.3.2 倒影特效
  - 15.3.3 旋转特效
- 15.4 绘制图片
  - 15.4.1 载入图像文件
  - 15.4.2 GDI+输出图像
- 15.5 图像特效处理

- 15.5.1 底片效果
- 15.5.2 浮雕效果
- 15.6 小结

## 第 16 章 ASP.NET 3.5 和 AJAX

- 16.1 AJAX 基础
  - 16.1.1 什么是 AJAX
  - 16.1.2 ASP.NET AJAX 入门
  - 16.1.3 ASP.NET 2.0 AJAX
  - 16.1.4 ASP.NET 3.5 AJAX
  - 16.1.5 AJAX 简单示例
- 16.2 ASP.NET 3.5AJAX 控件
  - 16.2.1 脚本管理控件 (ScriptManger)
  - 16.2.2 脚本管理控件 (ScriptMangerProxy)
  - 16.2.3 时间控件 (Timer)
  - 16.2.4 更新区域控件 (UpdatePanel)
  - 16.2.5 更新进度控件 (UpdateProgress)
- 16.3 AJAX 编程
  - 16.3.1 自定义异常处理
  - 16.3.2 使用母版页的 UpdatePanel
  - 16.3.3 母版页刷新内容窗体
- 16.4 小结

## 第 17 章 ASP.NET MVC 基础

- 17.1 了解 MVC
  - 17.1.1 MVC 和 Web Form
  - 17.1.2 ASP.NET MVC 的运行结构
- 17.2 ASP.NET MVC 基础
  - 17.2.1 安装 ASP.NET MVC
  - 17.2.2 新建一个 MVC 应用程序
  - 17.2.3 ASP.NET MVC 应用程序的结构
  - 17.2.4 运行 ASP.NET MVC 应用程序
- 17.3 ASP.NET MVC 原理
  - 17.3.1 ASP.NET MVC 运行流程
  - 17.3.2 ASP.NET MVC 工作原理
- 17.4 ASP.NET MVC 开发
  - 17.4.1 创建 ASP.NET MVC 页面
  - 17.4.2 ASP.NET MVC 数据呈现 (ViewData)
  - 17.4.3 ASP.NET MVC 跨页数据呈现 (TempData)
  - 17.4.4 ASP.NET MVC 页面重定向
  - 17.4.5 ASP.NET MVC URL 路由 (URLRouting)
  - 17.4.6 ASP.NET MVC 控件辅助工具 (Helper)
  - 17.4.7 ASP.NET MVC 表单传值
- 17.5 小结

## 第 18 章 WCF 开发基础

- 18.1 了解 WCF

- 18.1.1 什么是 WCF
- 18.1.2 为什么需要 WCF
- 18.2 WCF 基础
  - 18.2.1 服务
  - 18.2.2 地址
  - 18.2.3 契约
- 18.3 WCF 应用
  - 18.3.1 创建 WCF 应用
  - 18.3.2 创建 WCF 方法
- 18.4 WCF 消息传递
  - 18.4.1 消息传递
  - 18.4.2 消息操作
- 18.5 使用 WCF 服务
  - 18.5.1 在客户端添加 WCF 服务
  - 18.5.2 在客户端使用 WCF 服务
- 18.6 小结

## 第 19 章 WPF 开发基础

- 19.1 了解 WPF
  - 19.1.1 什么是 WPF
- 19.2 WPF 的应用范围
- 19.2 WPF 和 Microsoft Expression
  - 19.2.1 使用 Microsoft Expression Blend 设计 WPF
  - 19.2.2 WPF 控件样式
  - 19.2.3 浅谈 XAML
  - 19.2.4 WPF 控件层次
- 19.3 WPF 应用程序开发
  - 19.3.1 WPF 动画事件
  - 19.3.2 WPF 时间轴
  - 19.3.3 WPF 事件处理
- 19.4 WPF 系统开发
  - 19.4.1 WPF 系统需求
  - 19.4.2 WPF 界面开发
  - 19.4.3 WPF 动画制作
  - 19.4.4 WPF 事件编写
- 19.5 小结

## 第六篇 ASP.NET 3.5 与 LINQ

## 第 20 章 ASP.NET 3.5 与 LINQ

- 20.1 什么是 LINQ
  - 20.1.1 LINQ 起源
  - 20.1.2 LINQ 构架
  - 20.1.3 LINQ 与 Visual Studio 2008 新特性

- 20.2 LINQ 与 Web 应用程序
  - 20.2.1 创建使用 LINQ 的 Web 应用程序
  - 20.2.2 基本的 LINQ 数据查询
  - 20.2.3 IEnumerable 和 IEnumerable<T>接口
  - 20.2.4 IQueryProvider 和 IQueryable<T>接口
  - 20.2.5 LINQ 相关的命名空间
- 20.3 Lambda 表达式
  - 20.3.1 匿名方法
  - 20.3.2 Lambda 表达式基础
  - 20.3.3 Lambda 表达式格式
  - 20.3.4 Lambda 表达式树
  - 20.3.5 访问 Lambda 表达式树
- 20.4 小结

## 第 21 章 使用 LINQ 查询

- 21.1 LINQ 查询概述
  - 21.1.1 准备数据源
  - 21.1.2 使用 LINQ
  - 21.1.3 执行 LINQ 查询
- 21.2 LINQ 查询语法概述
- 21.3 基本子句
  - 21.3.1 from 查询子句
  - 21.3.2 where 条件子句
  - 21.3.3 select 选择子句
  - 21.3.4 group 分组子句
  - 21.3.5 orderby 排序子句
  - 21.3.6 into 连接子句
  - 21.3.7 join 连接子句
  - 21.3.8 let 临时表达式子句
- 21.4 LINQ 查询操作
  - 21.4.1 LINQ 查询概述
  - 21.4.2 投影操作
  - 21.4.3 筛选操作
  - 21.4.4 排序操作
  - 21.4.5 聚合操作
- 21.5 使用 LINQ 查询和操作数据库
  - 21.5.1 简单查询
  - 21.5.2 建立连接
  - 21.5.3 插入数据
  - 21.5.4 修改数据
  - 21.5.5 删除数据
- 21.6 LINQ 与 MVC
  - 21.6.1 创建 ASP.NET MVC 应用程序
  - 21.6.2 创建 LINQ to SQL
  - 21.6.3 数据查询
- 21.7 小结

## 第七篇 ASP.NET 3.5 模块开发

### 第 22 章 注册模块设计

- 22.1 学习要点
- 22.2 系统设计
  - 22.2.1 模块功能描述
  - 22.2.2 模块流程分析
- 22.3 数据库设计
  - 22.3.1 数据库的分析和设计
  - 22.3.2 数据表的创建
- 22.4 界面设计
  - 22.4.1 基本界面
  - 22.4.2 创建 CSS
- 22.5 代码实现
  - 22.5.1 验证控制
  - 22.5.2 过滤输入信息
  - 22.5.3 插入注册信息
  - 22.5.4 管理员页面
- 22.6 实例演示
- 22.7 小结

### 第 23 章 登录模块设计

- 23.1 学习要点
- 23.2 系统设计
  - 23.2.1 模块功能描述
  - 23.2.2 模块流程分析
- 23.3 数据库设计
  - 23.3.1 数据库设计分析
  - 23.3.2 数据库表的创建
- 23.4 界面设计
  - 23.4.1 基本界面
  - 23.4.2 创建 CSS
  - 23.4.3 发送密码页面
- 23.5 代码实现
  - 23.5.1 登录代码实现
  - 23.5.2 邮件发送页面
  - 23.5.3 根据不同的用户显示不同的内容
- 23.6 实例演示
- 23.7 小结

### 第 24 章 广告模块设计

- 24.1 学习要点
- 24.2 系统设计
  - 24.2.1 模块功能描述
  - 24.2.2 模块流程分析

- 24.3 数据库设计
  - 24.3.1 数据库设计分析
  - 24.3.2 数据库表的创建
- 24.4 界面设计
  - 24.4.1 发布广告界面
  - 24.4.2 发布广告页数据源配置
  - 24.4.3 修改广告界面
  - 24.4.4 管理广告界面
  - 24.4.5 分类管理界面
- 24.5 代码实现
  - 24.5.1 广告添加功能
  - 24.5.2 广告修改功能
  - 24.5.3 自定义控件的实现
- 24.6 实例演示
- 24.7 小结

## 第 25 章 新闻模块设计

- 25.1 学习要点
- 25.2 系统设计
  - 25.2.1 模块功能描述
  - 25.2.2 模块流程分析
- 25.3 数据库设计
  - 25.3.1 数据库设计
  - 25.3.2 数据表的创建
- 25.4 界面设计
  - 25.4.1 登录界面
  - 25.4.2 后台框架集
  - 25.4.3 新闻发布页面
  - 25.4.4 新闻修改页面
  - 25.4.5 新闻管理页面
  - 25.4.6 新闻分类管理页面
- 25.5 代码实现
  - 25.5.1 导航菜单配置
  - 25.5.2 身份验证页面
  - 25.5.3 新闻发布页面
  - 25.5.4 静态生成功能
  - 25.5.5 新闻显示页面
  - 25.5.6 静态模板编写
- 25.6 实例演示
- 25.7 小结

## 第 26 章 投票模块设计

- 26.1 学习要点
- 26.2 系统设计
  - 26.2.1 模块功能描述
  - 26.2.2 模块流程分析
- 26.3 数据库设计



- 26.3.1 数据库设计
- 26.3.2 数据表的创建
- 26.4 界面设计
  - 26.4.1 后台框架集
  - 26.4.2 投票管理页面
  - 26.4.3 投票发布页面
  - 26.4.4 投票修改页面
  - 26.4.5 投票删除页面
- 26.5 代码实现
  - 26.5.1 添加投票代码实现
  - 26.5.2 修改投票代码实现
  - 26.5.3 删除投票代码实现
  - 26.5.4 显示投票代码实现
  - 26.5.5 用户投票代码实现
- 26.6 实例演示
- 26.7 小结

## 第 27 章 聊天模块设计

- 27.1 学习要点
- 27.2 系统设计
  - 27.2.1 模块功能描述
  - 27.2.2 模块流程分析
- 27.3 界面设计
  - 27.3.1 登录界面设计
  - 27.3.2 登录界面 CSS
  - 27.3.3 聊天室显示界面
  - 27.3.4 聊天室界面 CSS
- 27.4 代码实现
  - 27.4.1 登录代码实现
  - 27.4.2 多人聊天代码实现
  - 27.4.3 单人聊天代码实现
  - 27.4.4 聊天记录保存实现
- 27.5 实例演示
- 27.6 小结

## 第八篇 ASP.NET 3.5 应用实例

### 第 28 章 制作一个 ASP.NET 留言本

- 28.1 系统设计
  - 28.1.1 需求分析
  - 28.1.2 系统功能设计
  - 28.1.3 模块功能划分
- 28.2 数据库设计
  - 28.2.1 数据库的分析和设计

- 28.2.2 数据表的创建
- 28.2.3 数据表关系图
- 28.3 系统公用模块的创建
  - 28.3.1 创建 CSS
  - 28.3.2 使用 SQLHepler
  - 28.3.3 配置 Web.config
- 28.4 系统界面和代码实现
  - 28.4.1 留言板用户控件
  - 28.4.2 管理员登录实现
  - 28.4.3 用户注册登录实现
  - 28.4.4 用户登录实现
  - 28.4.5 留言板界面布局
  - 28.4.6 留言功能实现
  - 28.4.7 回复功能实现
  - 28.4.8 删除功能的实现
  - 28.4.9 用户索引实现
- 28.5 用户体验优化
  - 28.5.1 AJAX 留言实现
  - 28.5.2 AJAX 数据重绑定
  - 28.5.3 系统导航实现
  - 28.5.4 侧边栏界面优化
- 28.6 用户功能实现
  - 28.6.1 用户信息界面
  - 28.6.2 用户信息修改实现
  - 28.6.3 用户信息删除实现
  - 28.6.4 用户注销
- 28.7 实例演示
  - 28.7.1 准备数据源
  - 28.7.2 基本实例演示
  - 28.7.3 用户功能演示
- 28.8 小结

## 第 29 章 制作一个 ASP.NET 校友录系统

### 第 29 章 制作一个 ASP.NET 校友录系统

- 29.1 系统设计
  - 29.1.1 需求分析
  - 29.1.2 系统功能设计
  - 29.1.3 模块功能划分
- 29.2 数据库设计
  - 29.2.1 数据库分析和设计
  - 29.2.2 数据表的创建
- 29.3 数据表关系图
- 29.4 系统公用模块的创建
  - 29.4.1 使用 Fckeditor
  - 29.4.2 使用 SQLHelper
  - 29.4.3 配置 Web.config
- 29.5 系统界面和代码实现

- 29.5.1 用户注册实现
- 29.5.2 用户登录实现
- 29.5.3 校友录页面规划
- 29.5.4 自定义控件实现
- 29.5.5 校友录页面实现
- 29.5.6 日志发布实现
- 29.5.7 日志修改实现
- 29.5.8 管理员日志删除
- 29.5.9 日志显示页面
- 29.5.10 用户索引页面
- 29.5.11 管理员用户删除
- 29.6 用户体验优化
  - 29.6.1 超链接样式优化
  - 29.6.2 默认首页优化
  - 29.6.3 导航栏编写
  - 29.6.4 AJAX 留言优化
  - 29.6.5 优化留言表情
- 29.7 高级功能实现
  - 29.7.1 后台管理页面实现
  - 29.7.2 日志管理实现
  - 29.7.3 日志修改和删除实现
  - 29.7.4 评论删除实现
  - 29.7.5 板报功能实现
  - 29.7.6 用户修改和删除实现
  - 29.7.7 用户权限管理
  - 29.7.8 权限及注销实现
- 29.8 实例演示
  - 29.8.1 准备数据源
  - 29.8.2 基本实例演示
  - 29.8.3 管理后台演示
- 29.9 小结

## 第一篇 .NET 基础

第 1 章 认识 ASP.NET 3.5

第 2 章 C# 3.0 程序设计基础

第 3 章 面向对象设计基础

第 4 章 ASP.NET 的网页代码模型及生命周期

## 第 1 章 ASP.NET 3.5 与开发工具

从本章开始，读者将能够系统的学习 **ASP.NET 3.5** 技术，相对于 **ASP.NET 2.0** 而言，在 **3.5** 版本的 **ASP.NET** 中并没有太多的变化，而更多的变化则在于 **C#** 编程语言中。而作为微软主推的编程语言，**ASP.NET 3.5** 能够使用 **C#** 的最新特性进行高效的开发，本章从基础讲解什么是 **ASP.NET**，以及开发工具的使用。

### 1.1 什么是 ASP.NET

**ASP.NET** 是微软推出的 **ASP** 的下一代 **Web** 开发技术。**ASP.NET** 顾名思义是基于 **.NET** 平台而存在的，在了解 **ASP.NET** 之前就需要了解 **.NET** 技术，了解 **.NET** 平台的相关技术才能够深入的了解 **ASP.NET** 是如何运作的。

#### 1.1.1 .NET 历史与展望

**.NET** 技术是微软近几年推出的主要技术，微软为 **.NET** 技术的推出可谓是不遗余力，在 **.NET** 平台下，微软有着极大的野心，**.NET** 技术的发展历程如下所示。

- ❑ 2000 年 6 月，微软公司总裁比尔·盖茨在“论坛 2000”的会议上向业内公布 **.NET** 平台并描绘了 **.NET** 的愿景。
- ❑ 2002 年 1 月，微软发布 **.NET Framework 1.0** 版本，以及 **Visual Studio .NET 2002** 进行 **.NET Framework 1.0** 应用程序的辅助开发。
- ❑ 2003 年 4 月，微软发布 **.NET Framework 1.1** 版本，以及针对 **.NET Framework 1.1** 版本的开发工具 **Visual Studio 2003**，**.NET Framework 1.1** 版本较之于 **.NET Framework 1.0** 而言有重大的改进。
- ❑ 2004 年 6 月，微软在 **TechEd Europe** 会议上发布 **.NET Framework 2.0 beta** 版本，以及 **Visual Studio 2005** 的 **beta** 版本，在 **Visual Studio 2005** 的 **beta** 版本中包含了多个精简版，以便不同的开发人员的需要。
- ❑ 2005 年 4 月，微软发布 **Visual Studio 2005** 的 **beta 2** 版本。
- ❑ 2005 年 11 月，微软发布 **Visual Studio 2005** 的正式版和 **SQL Server 2005** 的正式版。
- ❑ 2006 年 11 月，微软发布 **.NET Framework 3.0** 版本，在其中加入了一些新特性，以及语法特性，这些特性包括 **Windows Workflow Foundation**、**Windows Communication Foundation**、**Windows CardSpace** 和 **Windows Presentation Foundation**。
- ❑ 2007 年 11 月，微软发布 **.NET Framework 3.5** 版本，在其中加入了更多的新特性，包括 **LINQ**、**AJAX** 等，为下一代软件开发做出准备。
- ❑ 2008 年 11 月，微软向业界发布 **.NET Framework 4.0** 社区测试版，以及 **Visual Studio 2010** 社区测试版，标识着 **.NET 4.0** 的到来。

在 **.NET** 发展的 8 年时间里，**.NET** 技术在不断的改进。虽然在 2002 年微软发布了 **.NET** 技术的第一个版本，但是由于系统维护和系统学习的原因，**.NET** 技术当时并没有广泛的被开发人员和企业所接受。而自从 **.NET 2.0** 版本之后，越来越多的开发人员和企业已经能够接受 **.NET** 技术带来的革新。

而随着计算机技术的发展，越来越高的要求和越来越多的需求让开发人员不断的进行新技术的学习，这里包括云计算和云存储等新概念。**.NET** 平台同样为最新的概念和软件开发理念做出准备，这其中就包括**3.0**中出现并不断完善的 **Windows Workflow Foundation**、**Windows Communication Foundation**、**Windows CardSpace** 和 **Windows Presentation Foundation** 等应用。

在最新的操作系统 **Vista** 中，微软集成了**.NET** 平台，使用**.NET** 技术进行软件开发能够无缝的将软件部署在操作系统中，在进行软件的升级和维护中，基于**.NET** 平台的软件也能够快速升级。微软的**.NET** 野心不仅如此，微软的**.NET** 平台还在为多核化、虚拟化、云计算做准备。随着时间的推移，**.NET** 平台已经逐渐完善，学习**.NET** 平台以及**.NET** 技术对开发人员而言能够在未来的计算机应用中起到促进作用。

### 1.1.2 ASP.NET 与 ASP

对于 **ASP.NET** 而言，开发人员不可避免的会将 **ASP.NET** 与 **ASP** 进行比较，因为 **ASP.NET** 可以算是 **ASP** 的下一个版本。但是 **ASP.NET** 却与 **ASP** 完全不同，可以说微软重新将 **ASP** 进行编写和组织形成 **ASP.NET** 技术。

在传统的 **ASP** 开发中，开发人员可以在页面中进行 **ASP** 代码的编写，当服务器请求相应的页面时，服务器会解析 **ASP** 代码进行页面呈现。**ASP** 具有轻巧等特点，但是随着互联网的发展，**ASP** 也越来越多的呈现出其不足之处，这些不足之处包括 **ASP** 代码无法和 **HTML** 代码很好的分离，这就造成了页面代码混乱、维护性低等情况。当 **ASP** 中出现错误或者需要进行功能的添加，就需要多大部分的页面进行更改，这样就降低了 **ASP** 程序的复用性和维护性。

而随着互联网的不断发展，基于 **Web** 的应用程序诞生，**ASP** 已经不能满足日益增长的需求，于是诞生了 **ASP.NET**。**ASP.NET** 虽然同 **ASP** 都包含“**ASP**”这个词，但是 **ASP.NET** 与 **ASP** 完全是不同的编程模型，对于有 **ASP** 经验的人可以在页面中进行代码编写，而对于 **ASP.NET** 而言，**ASP** 的经验基本上不适用于 **ASP.NET** 的开发。**ASP.NET** 使用了软件开发的思维进行 **Web** 应用程序的编写，**ASP.NET** 是面向对象的开发模型，使用 **ASP.NET** 能够提高代码的重用性，降低开发和维护的成本。

而对于 **ASP** 而言，同样不能够满足日益增长的互联网需求，随着计算机科学与技术的发展，互联网和本地客户端的界限越来越模糊。一个 **Web** 应用程序可能是基于本地应用程序，而本地应用程序也可能基于服务器的服务进行开发的，这就对 **Web** 应用程序提出了更高的要求，相比之下，基于**.NET** 平台的 **ASP.NET** 却能够适应和解决复杂的互联网需求。

从历史发展的角度而言，不得不说 **ASP** 已经是过时的技术，但是并不代表 **ASP** 不会被使用，现在还有很多 **ASP** 应用程序，在小型的应用中，**ASP** 依旧是低成本的最佳选择。

### 1.1.3 ASP.NET 开发工具

相对于 **ASP** 而言，**ASP.NET** 具有更加完善的开发工具。在传统的 **ASP** 开发中，可以使用 **Dreamware**、**FrontPage** 等工具进行页面开发。当时使用 **Dreamware**、**FrontPage** 等工具进行 **ASP** 应用程序开发时，其效率并不能提升，并且这些工具对 **ASP** 应用程序的开发和运行也不会带来性能提升。

相比之下，对于 **ASP.NET** 应用程序而言，微软开发了 **Visual Studio** 开发环境提供给开发人员进行高效的开发，开发人员还能够使用现有的 **ASP.NET** 控件进行高效的应用程序开发，这些控件包括日历控件、分页控件、数据源控件和数据绑定控件。开发人员能够在 **Visual Studio** 开发环境中拖动相应的控件到页面中实现复杂的应用程序编写。

**Visual Studio** 开发环境在人机交互的设计理念上更加完善，使用 **Visual Studio** 开发环境进行应用程序开发能够极大的提高开发效率，实现复杂的编程应用，如图 1-1 所示。



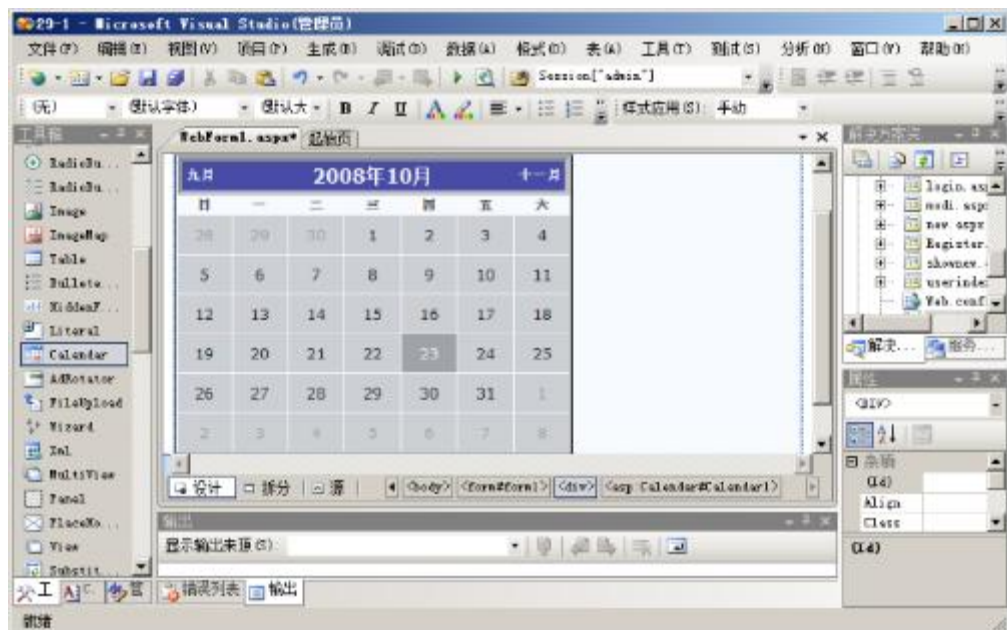


图 1-1 Visual Studio 开发环境

**Visual Studio** 开发环境为开发人员提供了诸多控件，使用这些控件能够实现在 **ASP** 中难以实现的复杂功能，极大的简化了开发人员的开发。如图 1-1 所示，在传统的 **ASP** 开发过程中需要实现日历控件是非常复杂和困难的，而在 **ASP.NET** 中，系统提供了日历控件用于日历的实现，开发人员只需要将日历控件拖动到页面中就能够实现日历效果。

使用 **Visual Studio** 开发环境进行 **ASP.NET** 应用程序开发还能够直接编译和运行 **ASP.NET** 应用程序。在使用 **Dreamware**、**FrontPage** 等工具进行页面开发时需要安装 **IIS** 进行 **ASP.NET** 应用程序运行，而 **Visual Studio** 提供了虚拟的服务器环境，用户可以像 **C/C++** 应用程序编写一样在开发环境中进行应用程序的编译和运行。

## 1.1.4 ASP.NET 客户端

**ASP.NET** 应用程序是基于 **Web** 的应用程序，所以用户可以使用浏览器作为 **ASP.NET** 应用程序的客户端进行 **ASP.NET** 应用程序的访问。浏览器已经是操作系统中必备的常用工具，包括 **IE 7**、**IE 8**、**Firefox**、**Opera** 等常用浏览器都可以支持 **ASP.NET** 应用程序的访问和使用。对于 **ASP.NET** 应用程序而言，由于其客户端为浏览器，所以 **ASP.NET** 应用程序的客户端部署成本低，可以在服务器端进行更新而无需进入客户端进行客户端的更新。

## 1.1.5 ASP.NET 3.5 新增控件

在 **ASP.NET 1.1** 初期，开发人员抱怨微软自带的 **ASP.NET** 控件过少，无法满足日益增长的应用程序开发，而到了 **ASP.NET 2.0** 版本中，微软增加了数十种服务器控件用于应用程序的开发。这些服务器控件不仅在一定程度上实现的复杂的功能，还提升了应用程序的可维护性、可扩展性，同时这些服务器控件也提高了 **ASP.NET** 应用程序的代码的复用性。

在 **ASP.NET 3.5** 中，微软虽然没有像 **ASP.NET 1.1** 到 **ASP.NET 2.0** 一样增加数十种服务器控件，但是微软增加了 **ListView** 控件和 **DataPager** 控件两个颇受欢迎的服务器控件。使用 **ListView** 控件和 **DataPager** 控件能够快速地进行页面数据的呈现和布局，同时还能轻松的实现分页和数据更新等操作。

### 1. ListView 控件

**ListView** 控件是 **ASP.NET 3.5** 中新增的数据绑定控件。**ListView** 控件是介于 **GridView** 控件和 **Repeater** 之间的另一种数据绑定控件，相对于 **GridView** 来说，它有着更为丰富的布局手段，开发人员可以在 **ListView** 控件的模板内写任何 **HTML** 标记或者控件。

### 2. DataPage 控件

**DataPager** 控件通过实现 .NET 框架中 **IPageableItemContainer** 接口实现了控件的分页。在 **ASP.NET 3.5** 中，**ListView** 控件可以使用 **DataPager** 控件进行分页操作。

要在 **ListView** 中使用 **DataPager** 控件需要在 **ListView** 的 **LayoutTemplate** 模板中加入 **DataPager** 控件，**DataPager** 控件包括两种样式，一种是“上一页/下一页”样式，第二种是“数字”样式，方便了开发人员实现不同的分页效果。同时，用户不仅能够使用微软为开发人员提供的服务器控件，**Visual Studio 2008** 还能够让开发人员创建用户控件和自定义控件，以满足应用程序中越来越大的开发需求并提供了可扩展、可自定义控件。

## 1.1.6 ASP.NET 3.5 AJAX

在 **Web** 应用程序的开发中，越来越多的网站能够实现用户操作的无刷新效果。网站页面的无刷新效果能够提高用户体验、提高网站应用的操作性并能够降低服务器与客户端之间的通信次数。在 **ASP.NET 3.5** 中，**Visual Studio** 开发环境提供了 **AJAX** 应用环境，开发人员能够使用 **Visual Studio 2008** 进行 **AJAX** 应用程序和 **AJAX** 控件的创建，如图 1-2 所示。

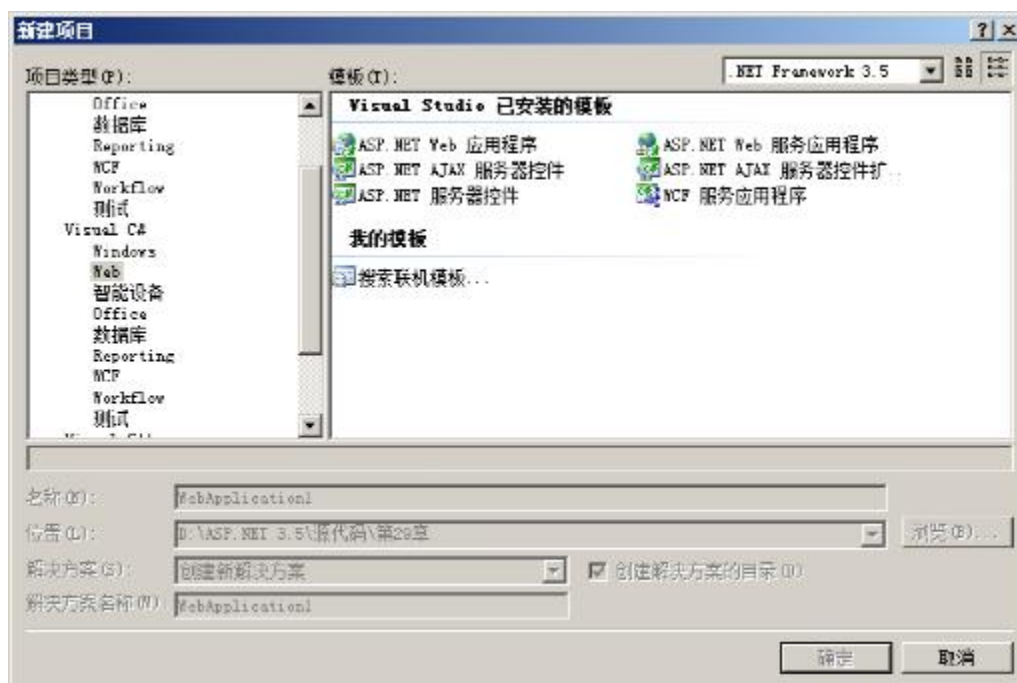


图 1-2 新增的 AJAX 服务器控件创建应用

用户可以创建 **ASP.NET AJAX** 服务器控件和服务器扩展控件用于实现 **ASP.NET AJAX** 应用程序中所需要使用的自定义控件。在 **ASP.NET 3.5** 中，**Visual Studio 2008** 还提供了默认的 **AJAX** 控件，这些控件包括脚本管理控件（**ScriptManger**）、脚本管理控件（**ScriptMangerProxy**）、时间控件（**Timer**）、更新区域控件（**UpdatePanel**）和更新进度控件（**UpdateProgress**）。使用 **AJAX** 控件能够同服务器控件一起使用从而实现服务器控件的无刷新。**ASP.NET 3.5** 为 **AJAX** 应用程序开发提供了原生环境，开发人员使用 **Visual Studio 2008** 和默认的服务器控件就能够轻松的实现 **AJAX** 效果。

## 1.2 .NET 应用程序需框架

无论是 **ASP.NET** 应用程序还是 **ASP.NET** 应用程序中所提供的控件，甚至是 **ASP.NET** 支持的原生的 **AJAX** 应用程序都不能离开 .NET 应用程序框架的支持。**.NET** 应用程序框架作为 **ASP.NET** 以及其应用程序的基础而存在，若需要使用 **ASP.NET** 应用程序则必须使用 .NET 应用程序框架。

### 1.2.1 什么是.NET 应用程序框架

.NET 框架是一个多语言组件开发和执行环境，无论开发人员使用的是 **C#** 作为编程语言还是使用 **VB.NET** 作为其开发语言都能够基于 .NET 应用程序框架而运行。.NET 应用程序框架主要包括三个部分，这三个部分分别为公共语言运行时、统一的编程类和活动服务器页面。

#### 1. 公共语言运行时

公共语言运行时在组件的开发及运行过程中扮演着非常重要的角色。在经历了传统的面向过程开发，开发人员寻找更多的高效的方法进行应用程序开发，这其中的发展成为了面向对象的应用程序开发，在面向对象程序开发的过程中，衍生了组件开发。

在组件运行过程中，运行时负责管理内存分配、启动或删除线程和进程、实施安全性策略、同时满足当前组件对其它组件的需求。在多层开发和组件开发应用中，运行时负责管理组件与组件之间的功能的需求。

#### 2. 统一的编程类

.NET 框架为开发人员提供了一个统一、面向对象、层次化、可扩展的类库集（API）。现今，**C++** 开发人员使用的是 **Microsoft** 基类库，**Java** 开发人员使用的是 **Windows** 基类库，而 **Visual Basic** 用户使用的又是 **Visual Basic API** 集，在应用程序开发中，很难将应用程序进行平台的移植，当出现了不同版本的 **Windows** 时，就会造成移植困难。

**注意：**虽然 **Windows** 包括不同的版本，而这些版本的基本类库相同，但是不同版本的 **Windows** 同样会有不同的 API，例如 **Windows 9x** 系列和 **Windows NT** 系列。

而 .NET 框架就统一了微软当前的各种不同类型的框架，.NET 应用程序框架是一个系统级的框架，对现有的框架进行了封装，开发人员无需进行复杂的框架学习就能够轻松使用 .NET 应用程序框架进行应用程序开发。无论是使用 **C#** 编程语言还是 **Visual Basic** 编程语言都能够进行应用程序开发，不同的编程语言所调用的框架 API 都是来自 .NET 应用程序框架，所以这些应用程序之间就不存在框架差异的问题，在不同版本的 **Windows** 中也能够方便移植。

**注意：**.NET 框架能够安装到各个版本的 **Windows** 中，当有多个版本的 **Windows** 时，只需安装了 .NET 框架，任何 .NET 应用程序就能够在不同的 **Windows** 中运行而不需要额外的移植。

#### 3. 活动服务器页面

.NET 框架还为 **Web** 开发人员提供了基础保障，**ASP.NET** 是使用 .NET 应用程序框架提供的编程类库构建而成的，它提供了 **Web** 应用程序模型，该模型由一组控件和一个基本结构组成，使用该模型让 **ASP.NET Web** 开发变得非常的容易。开发人员可以将特定的功能封装到控件中，然后通过控件的拖动进行应用程序的开发，这样不仅提高了应用程序开发的简便性，还极大的精简了应用程序代码，让代码更具复用性。

.NET 应用程序框架不仅能够安装到多个版本的 **Windows** 中，还能够安装其他智能设备中，这些设备包括智能手机、**GPS** 导航以及其他家用电器中。.NET 框架提供了精简版的应用程序框架，使用 .NET 应用程序框架能够开发容易移植到手机、导航器以及家用电器中的应用程序。**Visual Studio 2008** 还提供了智能电话应用程序开发的控件，实现了多应用、单平台的特点。

开发人员在使用 **Visual Studio 2008** 和 .NET 应用程序框架进行应用程序开发时，会发现无论是在原理上还是在控件的使用上，很多都是相通的，这样极大的简化了开发人员的学习过程，无论是 **Windows** 应用程序、**Web** 应用程序还是手机应用程序，都能够使用 .NET 框架进行开发。



## 1.2.2 公共语言运行时（CLR）

在前面的小结中可以看出，无论开发人员使用何种编程语言（如 **C#** 或 **Visual Basic**）都能够使用 **.NET** 应用程序框架进行应用程序的开发。那么何种原因使得开发人员使用任何 **.NET** 应用程序框架的支持的语言都能够使用 **.NET** 应用程序框架并实现相应的应用程序功能，这就要了解 **.NET** 中的公共语言运行库（**CLR**）。

公共语言运行时（**Common Language Runtime, CLR**）为托管代码提供各种服务，如跨语言集成、代码访问安全性、对象生存期管理、调试和分析支持。**CLR** 和 **Java** 虚拟机一样也是一个运行时环境，它负责资源管理（内存分配和垃圾收集），并保证应用和底层操作系统之间必要的分离。同时，为了提高 **.NET** 平台的可靠性，以及为了达到面向事务的电子商务应用所要求的稳定性和安全性级别，**CLR** 还要负责其他一些任务。

在公共语言运行时中运行的程序被称为托管程序。顾名思义，托管程序就是被公共语言运行时所托管的应用程序，公共语言运行时会监视应用程序的运行并在一定程度上监视应用程序的运行。当开发人员进行应用程序开发和运行时，例如出现了数组越界等错误都会被公共语言运行库所监控和捕获。

当开发人员进行应用程序的编写时，编写完成的应用程序将会被翻译成一种中间语言，中间语言在公共语言运行时中被监控并被解释成为计算机语言，解释后的计算机语言能够被计算机所理解并执行相应的程序操作。在程序开发中，使用的编程语言如果在 **CLR** 监控下就被称为托管语言，而语言的执行不需要 **CLR** 的监控就不是托管语言，被称为非托管语言。在托管语言在解释时的效率没有非托管语言迅速，因为托管的语言首先需要被解释成计算机语言，这也造成了性能问题。

虽然如此，但是 **CLR** 所带来的性能问题越来越不足以成为问题，因为随着计算机硬件的发展，当代计算机已经能够适应和解决托管程序所带来的效率问题。

## 1.2.3 .NET Framework 类库

**.NET Framework** 是支持生成和运行下一代应用程序和 **XML Web services** 的内部 **Windows** 组件。**.NET Framework** 类库包含了 **.NET** 应用程序开发中所需要的类和方法，开发人员可以使用 **.NET Framework** 类库提供的类和方法进行应用程序的开发。

**.NET Framework** 类库中的类和方法将 **Windows** 底层的 **API** 进行封装和重新设计，开发人员能够使用 **.NET Framework** 类库提供的类和方法方便的进行 **Windows** 应用程序开发，**.NET Framework** 还意图实现一个通用的编程环境。**.NET Framework** 想要实现的功能如下所示。

- ❑ 提供一个一致的面向对象的编程环境，无论这个代码是在本地执行还是在远程执行。
- ❑ 提供一个将软件部署和版本控制冲突最小化的代码执行环境以便于应用程序的部署和升级。
- ❑ 提供一个可提高代码执行安全性的代码执行环境，就算软件是来自第三方不可信任的开发商也能够提供可信赖的开发环境。
- ❑ 提供一个可消除脚本环境或解释环境的性能问题的代码执行环境，**.NET Framework** 将应用程序甚至是 **Web** 应用相关类编译成 **DLL** 文件。
- ❑ 使开发人员的经验在面对类型大不相同的应用程序时保持应用程序和数据的一致性，特别是使用面向服务开发和敏捷开发。
- ❑ 提供一个可以确保基于 **.NET Framework** 的代码可与任何其他代码开发、集成、移植的可靠环境。

**.NET Framework** 类库用于实现基于 **.NET Framework** 的应用程序所需要的功能，例如实现音乐的播放和多线程开发等技术都可以使用 **.NET Framework** 现有的类库进行开发。**.NET Framework** 类库相比 **MFC** 具有较好的命名方法，开发人员能够轻易阅读和使用 **.NET Framework** 类库提供的类和方法。

无论是基于何种平台或设备的应用程序都可以使用 **.NET Framework** 类库提供的类和方法。无论是基于 **Windows** 的应用程序和基于 **Web** 的 **ASP.NET** 应用程序还是移动应用程序，都可以使用现有的 **.NET Framework** 中的类和方法进行开发。在开发过程中，**.NET Framework** 类库中对不同的设备和平台提供类和方法基本相同，开发人员不需要进行重复学习就能够进行不同设备的应用程序的开发。

## 1.3 安装 Visual Studio 2008

使用.NET 框架进行应用程序开发的最好的工具莫过于 Visual Studio 2008，Visual Studio 系列产品被认为是世界上最好的开发环境之一。使用 Visual Studio 2008 能够快速构建 ASP.NET 应用程序并为 ASP.NET 应用程序提供所需要的类库、控件和智能提示等支持，本节会介绍如何安装 Visual Studio 2008 并介绍 Visual Studio 2008 中的窗口的使用和操作方法。

### 1.3.1 安装 Visual Studio 2008

在安装 Visual Studio 2008 之前，首先确保 IE 浏览器版本在 6.0 或更高，同时，可安装 Visual Studio 2008 开发环境的计算机配置要求如下所示。

- ❑ 支持的操作系统：Windows Server 2003; Windows Vista; Windows XP。
- ❑ 最低配置：1.6 GHz CPU，384 MB 内存，1024x768 显示分辨率，5400 RPM 硬盘。
- ❑ 建议配置：2.2 GHz 或更快的 CPU，1024 MB 或更大的内存，1280x1024 显示分辨率，7200 RPM 或更快的硬盘。
- ❑ 在 Windows Vista 上运行的配置要求：2.4 GHz CPU，768 MB 内存。

Visual Studio 2008 在硬件方面对计算机的配置要求如下所示。

- ❑ CPU：600MHz Pentium 处理器或 AMD 处理器或更高配置的 CPU。
- ❑ 内存：至少需要 128m 内存，推荐 256m 或更高。
- ❑ 硬盘：要求至少有 5G 空间进行应用程序的安装，推荐 10G 或更高。
- ❑ 显示器：推荐使用 800\*600 分辨率或更高。

当开发计算机满足以上条件后就能够安装 Visual Studio 2008，安装 Visual Studio 2008 的过程非常简单。

(1) 单击 Visual Studio 2008 的光盘或 MSDN 版的 Visual Studio 2008（90 天试用版）中的 setup.exe 安装程序进入安装程序，如图 1-3 所示。

(2) 进入 Visual Studio 2008 界面后，用户可以选择进行 Visual Studio 2008 的安装，单击【安装 Visual Studio 2008】按钮进行 Visual Studio 2008 的安装，如图 1-4 所示。



图 1-3 Visual Studio 2008 安装界面



图 1-4 加载安装组件

在进行 Visual Studio 2008 的安装前，Visual Studio 2008 安装程序首先会加载安装组件，这些组件为 Visual Studio 2008 的顺利安装提供了基础保障，安装程序在完成组件的加载前用户不能够进行安装步骤的选择。

(3) 在安装组件加载完毕后，用户可以单击【下一步】按钮进行 Visual Studio 2008 的安装，用户将进行 Visual Studio 2008 的安装路径的选择，如图 1-5 所示。

当用户选择安装路径后就能够进行 Visual Studio 2008 的安装。用户在选择路径前，可以选择相应的安



装功能，用户可以选择“默认值”、“完全”和“自定义”。选择“默认值”将会安装 **Visual Studio 2008** 提供的默认组件，选择“完全”将安装 **Visual Studio 2008** 的所有组件，而如果用户只需要安装几个组件，可以选择自定义进行组件的选择安装。

(4) 选择后，单击【安装】按钮就能够进行 **Visual Studio 2008** 的安装，如图 1-6 所示。



图 1-5 选择 Visual Studio 2008 安装路径

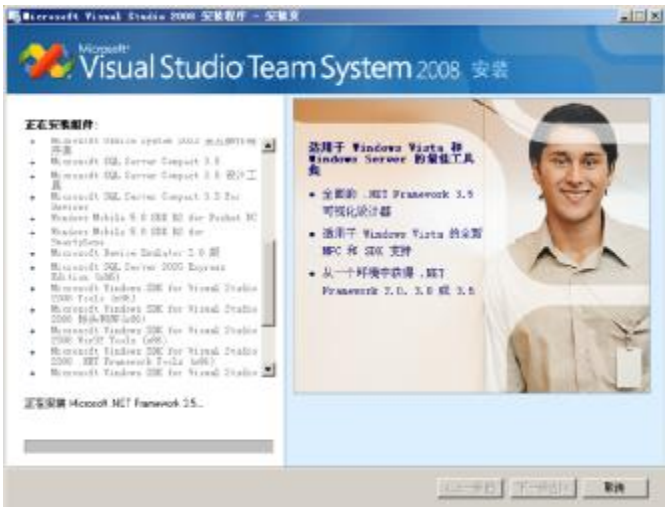


图 1-6 Visual Studio 2008 的安装

等待图 1-6 中的安装界面中左侧的安装列表的进度，当安装完毕后就会出现安装成功界面，说明已经在本地计算机中成功的安装了 **Visual Studio 2008**。

1.3.2 主窗口

在安装完成 **Visual Studio 2008** 后就能够进行.NET应用程序的开发，**Visual Studio 2008** 极大的提高了开发人员对.NET应用程序的开发效率，为了能够快速的进行.NET应用程序的开发，就需要熟悉 **Visual Studio 2008** 开发环境。当启动 **Visual Studio 2008** 后，就会呈现 **Visual Studio 2008** 主窗口，如图 1-7 所示。

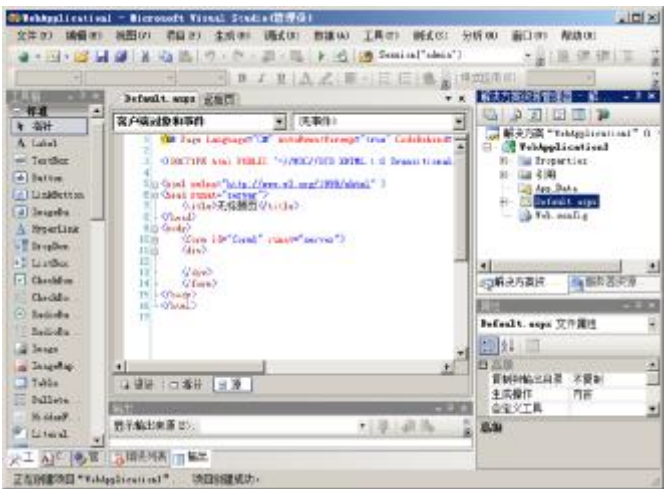


图 1-7 Visual Studio 2008 主界面

在图 1-7 中所示，**Visual Studio 2008** 主窗口包括其他多个窗口，最左侧的是工具箱，用于服务器控件的存放；中间是文档窗口，用于应用程序代码的编写和样式控制；中下方是错误列表窗口，用于呈现错误信息；右侧是资源管理器窗口和属性窗口，用于呈现解决方案，以及页面及控件的相应的属性。

1.3.3 文档窗口

文档窗口用于代码的编写和样式控制。当用户开发的是基于 **Web** 的 **ASP.NET** 应用程序时，文档窗口是以 **Web** 的形式呈现给用户，而代码视图则是以 **HTML** 代码的形式呈现给用户的，而如果用户开发的是基于 **Windows** 的应用程序，则文档窗口将会呈现应用程序的窗口或代码，如图 1-8、1-9 所示。





图 1-8 Windows 程序开发文档窗口

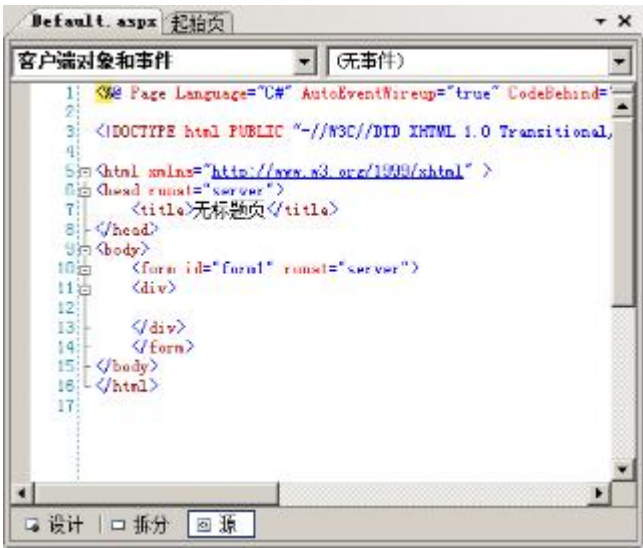


图 1-9 Web 程序开发文档窗口

当开发人员进行不同的应用程序开发时，文档窗口也会呈现为不同的样式以便开发人员进行应用程序开发。在 ASP.NET 应用程序中，其文档窗口包括三个部分，如图 1-10 所示。

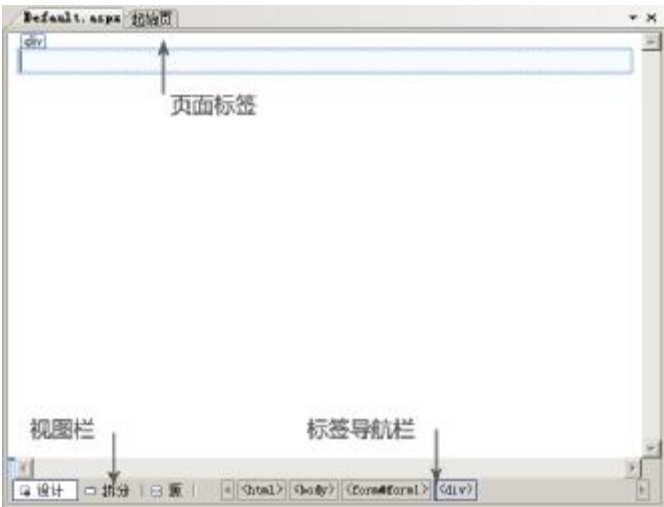


图 1-10 文档主窗口

正如图 1-10 所示，主文档窗口包括三个部分，开发人员可以通过使用这三个部分进行高效开发，这三个部分的功能如下所示。

- ❑ 页面标签：当进行多个页面进行开发时，会呈现多个页面标签，当开发人员需要进行不同页面的交替时可以通过页面标签进行页面替换。
- ❑ 视图栏：用户可以通过视图栏进行视图的切换，Visual Studio 2008 提供“设计”，“拆分”和“源代码”三种视图，开发人员可以选择不同的视图进行页面样式控制和代码的开发。
- ❑ 标签导航栏：标签导航栏能够进行不同的标签的选择，当用户需要选择页面代码中的<body>标签时，可以通过标签导航栏进行标签或标签内内容的选择。

开发人员可以灵活运用主文档窗口进行高效的应用程序开发，相比 Visual Studio 2005 而言，Visual Studio 2008 的视图栏窗口提供了拆分窗口，拆分窗口允许开发人员一边进行页面样式开发和代码编写。

注意：虽然 Visual Studio 2008 为开发人员提供了拆分窗口，但是只有在编写 Web 应用中文档主窗口才能够呈现拆分窗口。

1.3.4 工具箱

Visual Studio 2008 主窗口的左侧为开发人员提供了工具箱，工具箱中包含了 Visual Studio 2008 对 .NET 应用程序所支持的控件。对于不同的应用程序开发而言，在工具箱中所呈现的工具也不同。工具箱是 Visual Studio 2008 中的基本窗口，开发人员可以使用工具箱中的控件进行应用程序开发，如图 1-11 和图 1-12 所示。

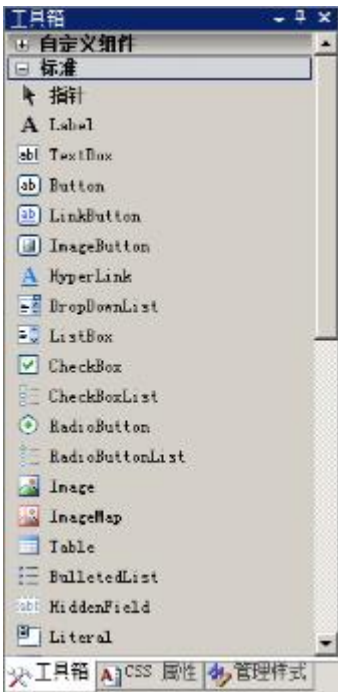


图 1-11 工具箱



图 1-12 选择类别

正如图 1-11 中所示，系统默认为开发人员提供了数十种服务器控件用于系统的开发，用户也可以添加工具箱选项卡进行自定义组件的存放。**Visual Studio 2008** 为开发人员提供了不同类别的服务器控件，这些控件被归为不同的类别，开发人员可以按照需求进行相应类别的控件的使用。开发人员还能够在工具箱中添加现有的控件。右击工具箱空白区域，在下拉菜单中选择【选择项】选项，系统会弹出窗口用于开发人员对自定义控件的添加，如图 1-13 所示。



图 1-13 添加自定义组件

组件添加完毕后就能够在工具箱中显式，开发人员能够将自定义组件拖放在主窗口中进行应用程序中相应的功能的开发而无需通过复杂编程实现。

**注意：**开发人员能够在互联网上下载其他人已经开发好的自定义组件进行.NET 应用程序开发，这样就无需通过编程实现重复的功能。

1.3.5 解决方案管理器

在 **Visual Studio 2008** 的开发中，为了能够方便开发人员进行应用程序开发，在 **Visual Studio 2008** 主窗口的右侧会呈现一个解决方案管理器。开发人员能够在解决方案管理器中进行相应的文件的选择，双击后相应文件的代码就会呈现在主窗口，开发人员还能够单击解决方案管理器下方的服务器资源管理器窗口进行服务器资源的管理，服务器资源管理器还允许开发人员在 **Visual Studio 2008** 中进行表的创建和修改。如图 1-14、1-15 所示。

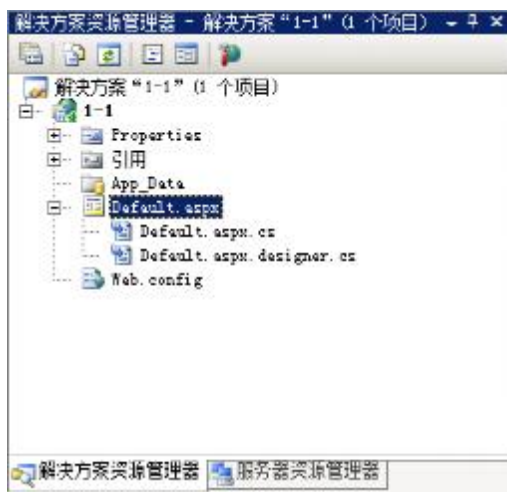


图 1-14 解决方案管理器

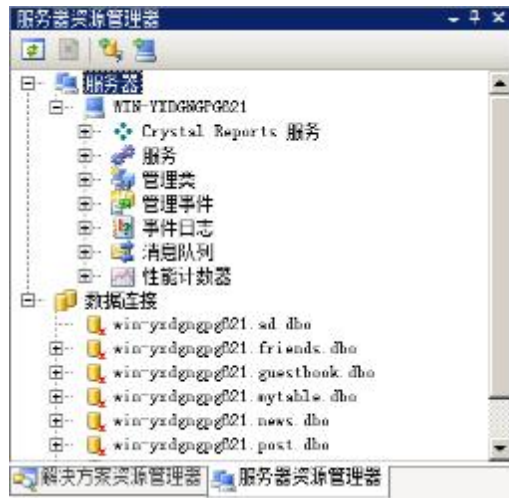


图 1-15 服务器资源管理器

解决方案管理器就是对解决方案进行管理，解决方案可以想象成是一个软件开发的整体方案，这个方案包括程序的管理、类库的管理和组件的管理。开发人员可以在解决方案管理器中双击文件进行相应的文件的编码工作，在解决方案管理器中也能够进行项目的添加和删除等操作，如图 1-16 所示。

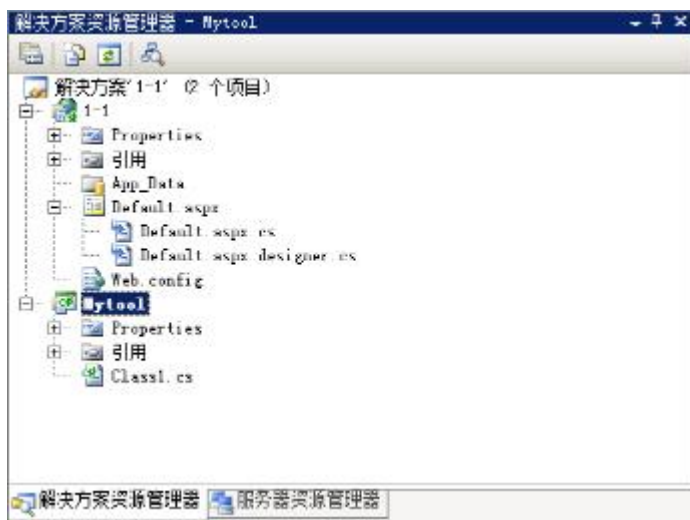


图 1-16 解决方案管理器

在应用程序开发中，通常需要进行不同的组件的开发，例如我开发用户界面，而我的一个同事进行后台开发，在开发中，如果将不同的模块分开开发或打开多个 **Visual Studio 2008** 进行开发是非常不方便的。解决方案管理器就能够解决这个问题。将一个项目看成是一个“解决方案”，不同的项目之间都在一个解决方案中进行互相的协调和相互的调用。

**注意：****Visual Studio 2008** 可能在默认情况下不会呈现解决方案管理器中的“解决方案‘1-1’这个标题”，开发人员可以在“工具”菜单栏的“选项”中的项目和解决方案中选择“总是显式解决方案”，如果没有项目和解决方案，则需要点击“显式所有设置”。

## 1.3.6 属性窗口

**Visual Studio 2008** 提供了非常多的控件，开发人员能够使用 **Visual Studio 2008** 提供的控件进行应用程序的开发。每个服务器控件都有自己的属性，通过配置不同的服务器控件的属性可以实现复杂的功能。服务器控件属性如图 1-17、1-18 所示。



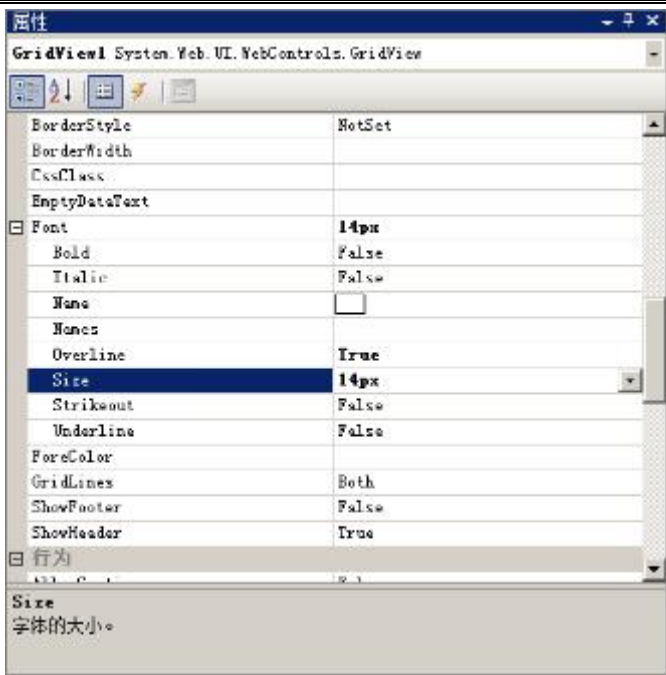


图 1-17 控件的样式属性

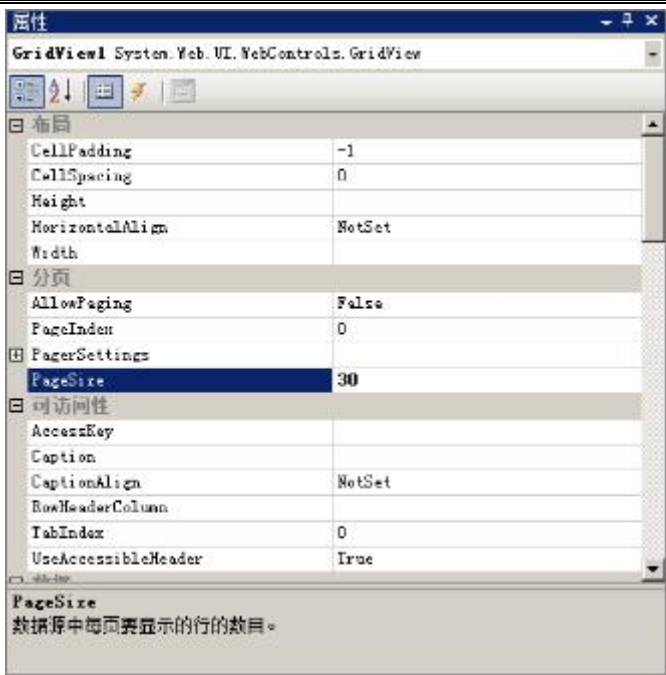


图 1-18 控件的数据属性

控件的属性配置中，可以为控件进行样式属性的配置，包括配置字体的大小、字体的颜色、字体的粗细、CSS 类等相关的控件所需要使用的样式属性，有些控件还需要进行数据属性的配置。这里使用了 **GirdView** 控件进行数据呈现并将 **PageSize** 属性（分页属性）设置为 **30**，则如果数据条目数大于 **30** 则该控件会自动按照 **30** 条目进行分页，免除了复杂的分页编程。

1.3.7 错误列表窗口

在应用程序的开发中，通常会遇到错误，这些错误会在错误列表窗口中呈现，开发人员可以单击相应的错误进行错误的跳转。如果应用程序中出现编程错误或异常，系统会在错误列表窗口呈现，如图 1-19 所示。



图 1-19 错误列表窗口

相对于传统的 **ASP** 应用程序编程而言，**ASP** 应用程序出现错误并不能良好的将异常反馈给开发人员。这在一方面是由于开发环境的原因，因为 **Dreamware** 等开发环境并不能原生的支持 **ASP** 应用程序的开发，另一方面也是由于 **ASP** 本身是解释型编程语言而无法进行良好的异常反馈。

对于 **ASP.NET** 应用程序而言，在应用程序运行前 **Visual Studio 2008** 会编译现有的应用程序并进行程序中错误的判断。如果 **ASP.NET** 应用程序出现错误，则 **Visual Studio 2008** 不会让应用程序运行起来，只有修正了所有的错误后才能够运行。

**注意：****Visual Studio 2008** 的错误处理并不能将应用程序中的逻辑错误检测出来，例如 **1 除以 0** 的错误是不会被检测出来，错误处理通常情况下处理的是语法错误而不是逻辑错误。

在错误列表窗口中包含错误、警告和消息选项卡，这些选项卡中的错误的安全级别不尽相同。对于错误选项卡中的错误信息，通常是语法上的错误，如果存在语法上的错误则不允许应用程序的运行，而对于警告和消息选项卡中信息安全级别较低，只是作为警告而存在，通常情况下不会危害应用程序的运行和使用。警告选项卡如图 1-20 所示。



图 1-20 警告选项卡

在应用程序中如果出现了变量未使用或者在页面布局中出现了布局错误，都可能会在警告选项卡中出现警告信息。双击相应的警告信息会跳转到应用程序中相应的位置，方便开发人员对于错误的检查。

注意：虽然警告信息不会造成应用程序运行错误，但是可能存在潜在的风险，推荐开发人员修正所有的错误和警告中出现的错误信息。

1.4 安装 SQL Server 2005

Visual Studio 2008 和 SQL Server 2005 都是微软为开发人员提供的开发工具和数据库工具，所以微软将 Visual Studio 2008 和 SQL Server 2005 紧密的集成在一起，使用微软的 SQL Server 进行.NET 应用程序数据开发能够提高.NET 应用程序的数据存储效率。

- (1) 打开 SQL Server 2005 安装盘，单击 SPLASH.HTA 文件进行安装，安装界面如图 1-21 所示。
- (2) 进入 SQL Server 2005 安装界面后就能够选择相应的平台选择，开发人员可以为相应的开发平台选择安装环境，如图 1-22 所示。



图 1-21 SQL Server 2005 安装界面



图 1-22 选择安装平台

(3) 开发人员可以选择相应的平台进行安装，现在大部分的操作系统都是基于 X86 平台进行应用，而 X64 平台虽然少，但是却有长足的发展前景。选择相应的开发平台后就能够进行进入安装选择界面，如图 1-23 所示。

在安装选择界面中开发人员可以进行安装准备，安装准备包括检查硬件和软件要求、阅读发行说明和安装 SQL Server 升级说明。在安装准备界面中的准备选项中开发人员可以检查自己所在的系统能否进行 SQL Server 2005 的安装，以及安装 SQL Server 2005 所需要遵守的协议。

(4) 在安装选择界面中需要选择【安装】连接可以进行 SQL Server 2005 应用程序的安装，可以选择【服务器组件、工具、联机丛书和示例】连接进行 SQL Server 2005 组件和应用程序的安装。单击【服务器组件、工具、联机丛书和示例】连接后如图 1-24 所示。



图 1-23 安装选择界面



图 1-24 所示 检查安装组件

(5) 在安装 SQL Server 2005 之前首先需要安装 SQL Server 2005 所必备的组件，这些组件包括.NET Framework 2.0 语言包，以及相应 SQL Server 2005 客户端组件，安装完成后就能够正式进入安装步骤，如图 1-25 所示。

SQL Server 2005 会进行应用程序的检查，检查包括系统的最低配置、IIS 功能要求、挂起的重新启动要求、ASP.NET 版本注册要求等等，这些要求系统会自行检查，如果 SQL Server 2005 安装程序提示安装成功则能够进行 SQL Server 2005 进一步的安装。

(6) 单击【下一步】按钮进行系统组件的安装，如图 1-26 所示。



图 1-25 系统配置检查



图 1-26 选择安装组件

(7) 选择相应的组件后单击【下一步】按钮就可以进行实例的选择，对于普通用户而言可以选择【默认实例】复选框进行 SQL Server 2005 的安装，如图 1-27 所示。



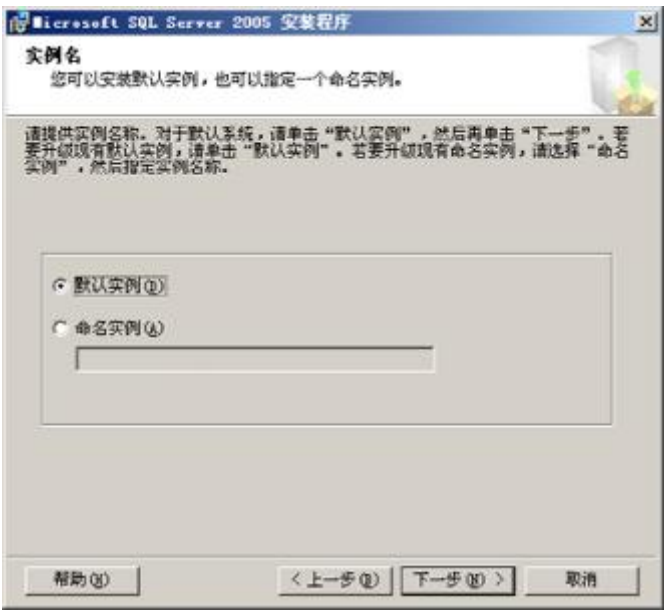


图 1-27 选择实例名称

(8) 在选择了【默认实例】复选框后就需要进行服务账户的配置，如果用户需要使用域用户账户可以选择【使用域用户账户】选项进行域配置，否则可以选择使用内置用户账户进行 SQL Server 2005 的安装并进行密码配置，如图 1-28 和图 1-29 所示。



图 1-28 选择服务账户



图 1-29 身份验证模式

(9) 单击【下一步】按钮进行身份验证模式选择，开发人员可以选择“Windows 身份验证模式”和“混合模式”，为了数据库服务器的安全，推荐使用“混合模式”进行身份验证。

注意：在有些操作系统上，例如 Windows Server 2003 和 Windows Server 2008 操作系统，可能需要强密码进行 SQL Server 2005 的安装。

(10) 在选择了身份验证模式后单击【下一步】按钮进行错误信息的配置和字符的配置，普通用户可以直接单击【下一步】按钮进行默认配置直至安装程序安装完毕。

## 1.5 ASP.NET 应用程序基础

使用 Visual Studio 2008 和 SQL Server 2005 能够快速的进行应用程序的开发，同时使用 Visual Studio 2008 和 SQL Server 2005 能够创建负载高的 ASP.NET 应用程序。通常情况下，Visual Studio 2008 负责 ASP.NET 应用程序的开发，而 SQL Server 2005 负责应用的数据存储。

## 1.5.1 创建 ASP.NET 应用程序

使用 **Visual Studio 2008** 能够进行 **ASP.NET** 应用程序的开发，微软提供了数十种服务器控件能够快速地进行应用程序开发。

(1) 打开 **Visual Studio 2008** 应用程序后如图 1-30 所示。

(2) 打开 **Visual Studio 2008** 初始界面后，可以单击菜单栏上的【文件】按钮，选择【新建项目】按钮创建 **ASP.NET** 应用程序，如图 1-31 所示。

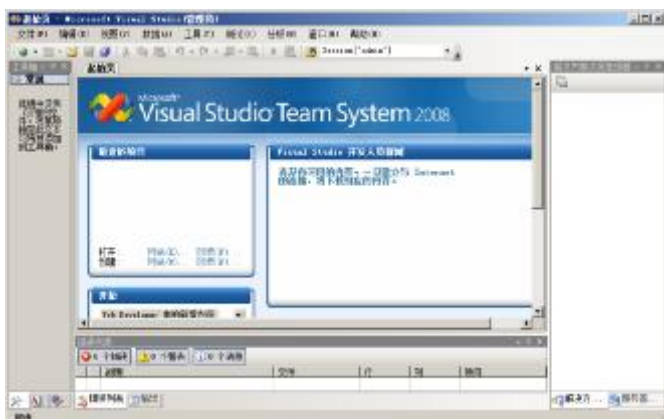


图 1-30 Visual Studio 2008 初始界面



图 1-31 创建 ASP.NET Web 应用程序

(3) 选择【**ASP.NET Web 应用程序**】选项，单击确定就能够创建一个最基本的 **ASP.NET Web** 应用程序。创建完成后系统会创建 **default.aspx**、**default.aspx.cs**、**default.aspx.designer.cs**、以及 **Web.config** 等文件用于应用程序的开发。

## 1.5.2 运行 ASP.NET 应用程序

创建 **ASP.NET** 应用程序后就能够进行 **ASP.NET** 应用程序的开发，开发人员可以在【资源管理器】中添加相应的文件和项目进行 **ASP.NET** 应用程序和组件开发。**Visual Studio 2008** 提供了数十种服务器控件以便开发人员进行应用程序的开发。

在完成应用程序的开发后，可以运行应用程序，单击【调试】按钮或选择【启动调试】按钮就能够调试 **ASP.NET** 应用程序。调试应用程序的快捷键为【F5】，开发人员也可以单击【F5】进行应用程序的调试，调试前 **Visual Studio 2008** 会选择是否启用 **Web.config** 进行调试，默认选择使用即可，如图 1-32 所示。

选择“修改 **Web.config** 文件以启动调试”进行应用程序的运行。在 **Visual Studio 2008** 中包含虚拟服务器，所以开发人员可以无需安装 **IIS** 进行应用程序的调试。但是一旦进入调试状态，就无法在 **Visual Studio 2008** 中进行 **cs** 页面，以及类库等源代码的修改，如图 1-33 所示。



图 1-32 启用调试配置

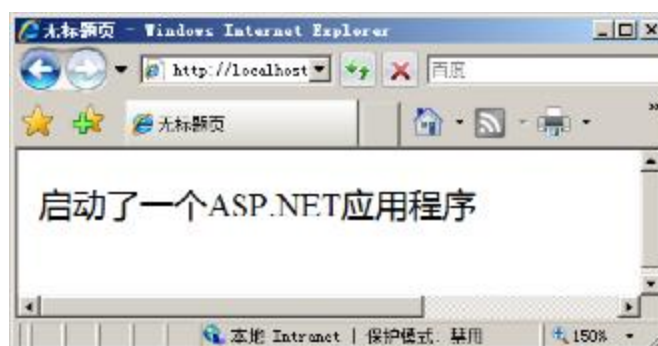


图 1-33 运行 ASP.NET 应用程序

**注意：**虽然 **Visual Studio 2008** 提供虚拟服务器，开发人员可以无需安装 **IIS** 进行应用程序调试，但是为了完好模拟 **ASP.NET** 网站应用程序，建议在发布网站前使用 **IIS** 进行调试。

## 1.5.3 编译 ASP.NET 应用程序

与传统的 **ASP** 应用程序开发不同的是, **ASP.NET** 应用程序能够将相应的代码编译成 **DLL** (动态链接库) 文件, 这样不仅能够提高 **ASP.NET** 应用程序的安全性, 还能够提高 **ASP.NET** 应用程序的速度。在现有的项目中, 打开相应的项目文件, 其项目源代码都可以进行读取, 如图 1-34 所示。

开发人员能够将源代码文件放置在服务器中进行运行, 但是将源代码直接运行会产生潜在的风险, 例如用户下载 **Default.aspx** 或其他页面进行源代码的查看, 这样就有可能造成源代码的泄露和漏洞的发现, 这样是非常不安全的。将 **ASP.NET** 应用程序代码编译成动态链接库能够提高安全性, 就算非法用户下载了相应的页面也无法看到源代码。

单击项目然后右击【项目图标】, 选择【发布】按钮发布 **ASP.NET** 应用程序, 系统会弹出发布对话框用户应用程序的发布, 如图 1-35 所示。



图 1-34 源代码文件



图 1-35 发布 Web

单击【发布】按钮后, **Visual Studio 2008** 就能够将网站编译并生成 **ASP.NET** 应用程序, 如图 1-36 所示。编译后的 **ASP.NET** 应用程序没有 **cs** 源代码, 因为编译后的文件会存放在 **bin** 目录下并编译成动态链接库文件, 如图 1-37 所示。



图 1-36 编译后的文件



图 1-37 动态链接库文件

正如图 1-36 所示, 在项目文件夹中只包含 **Default.aspx** 页面而并没有包含 **Default.aspx** 页面的源代码 **Default.aspx.cs** 等文件, 因为这些文件都被编译成为动态链接库文件。编译后的 **ASP.NET** 应用程序在第一次应用时会有些慢, 在运行后, 每次对 **ASP.NET** 应用程序的请求都可以直接从 **DLL** 文件中请求, 能够提高应用程序的运行速度。

## 1.6 小结

本章讲解了 **ASP.NET** 的基本概念, 以及 **.NET** 框架的基本概念。这些概念在初学 **ASP.NET** 时会觉得非



常的困难，但是这些概念会在今后的开发中逐渐清晰。虽然这些基本概念看上去没什么作用，但是在今后的 **ASP.NET** 应用开发中起着非常重要的作用，熟练掌握 **ASP.NET** 基本概念能够提高应用程序的适用性和健壮性。**Visual Studio 2008** 不仅提供了丰富的服务器控件还提供了属性、资源管理、错误列表窗口以便开发人员进行项目开发。本章还包括：

- **.NET 历史与展望**：包括.NET 应用程序的过去和未来以及发展前景。
- **ASP.NET 与 ASP**：讲解了 ASP.NET 与 ASP 的不同之处。
- **ASP.NET 开发工具**：讲解了 ASP.NET 开发工具的基本知识。
- **.NET 框架**：讲解了.NET 框架的基本知识。
- **公共语言运行时(CLR)**：讲解了.NET 框架的公共语言运行时。
- **.NET Framework 类库**：讲解了.NET 框架的.NET Framework 类库的基本知识。
- **安装 Visual Studio 2008**：讲解了如何安装 Visual Studio 2008。
- **安装 SQL Server 2005**：讲解了如何安装 SQL Server 2005。
- **ASP.NET 应用程序基础**：讲解了 ASP.NET 应用程序的安装，编译和运行。

本章着重讲解了 **Visual Studio 2008** 开发环境，以及如何安装 **SQL Server 2005** 以便于 **ASP.NET** 应用程序的数据存储。**Visual Studio 2008** 和 **SQL Server 2005** 的紧密集成能够提高 **ASP.NET** 应用程序的开发效率和运行效率。本章讲解了 **ASP.NET** 的基本知识，**ASP.NET** 使用的是 **C#** 语言进行开发的，了解 **C#** 编程语言是 **ASP.NET** 应用开发的第一步，下一章将会详细的讲解 **C#** 编程技术。

## 第 2 章 C# 3.0 程序设计基础

在第一章里，了解了 ASP.NET 3.5 的特性和一些基本的 .NET Framework 知识，不过如果要深入到 ASP.NET 3.5 应用程序开发，需要对开发语言有更加深入的了解。而在 .NET 平台上，微软主推的编程语言就是 C#，本章将会从 C# 的语法、结构和特性来讲解，以便读者能够深入的了解 C# 程序设计。

### 2.1 C# 程序

C# 程序有自己的程序结构。C# 编程语言类似 C++/Java 等面向对象编程语言，同样需要编写类、创建对象等。但是 C# 依旧有与其他面向对象编程语言不同的特性，使用这些特性能够快速正确的编写 C# 宿主语言的应用程序，如 ASP.NET、WinForm 等。

#### 2.1.1 C# 程序的结构

在开始学习和编写 C# 代码之前，首先应该了解 C# 编程语言的结构，下列代码说明了 C# 应用程序的基本结构。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;                                //使用命名空间
namespace mycsharp                                //程序代码命名空间
{
    class Program                                  //应用程序主类
    {
        static void Main(string[] args)           //入口方法
        {
            Console.WriteLine("Hello World");      //输出 Hello World
            Console.ReadKey();                      //等待用户输入
        }
    }
}
```

其中，using 关键字的用途是引用微软的 .NET 框架中现有的类库资源，该关键字出现在应用程序代码的开头，并使用在 cs 为后缀的文件中使用。using 关键字通常情况下会出现几次，其目的是引用类库中的各种资源，这些资源不仅包括代码中的 System, System.Collections.Generic, Linq，还包括其他 .NET 框架的资源。

System 命名空间提供了构建应用程序所需的各种系统功能，例如 LINQ 的类库包括了构建 LINQ 应用程序的各种类库资源。.NET 中提供大量的命名空间，以便开发人员能够使用现有的类库进行应用程序的开发。同时，在代码中也可以看到在其中包含一个 mycsharp 的一个命名空间，示例代码如 namespace mycsharp。在当前程序中声明该命名空间，可以在其他的程序中引用这个命名空间，并使用此命名空间下的类和方法。

另外，Program 是一个类名。在 C# 或其他的任何面向对象语言中（如 JAVA、C++）都需要编写类，类用于创建对象。在上述代码中，Program 是一个类的名称。

方法是用于描述类的行为。在上述示例第 9 行中，static void Main 是一个全局静态方法，它指示编译器从此处开始执行程序，相当于程序的入口，程序运行的时候会执行 Main 方法作为入口。在 C# Windows 编程中，大部分的应用程序必须在其组成程序的其中一个类中包含 Main 方法。

语句就是在 C# 应用程序中包含的指令，通过使用分号进行分割，编译器通过分号来区分它们。一些编

程语言只允许一行放置一条语句，但是 **C#** 允许放置多个语句，也可以将一个语句拆分成多行。虽然 **C#** 编译器支持这样的特性，但是还是推荐使用一行放置一个语句的，这样不仅提高了可读性，也便于书写。

括号“{”和“}”用来标识程序中代码的范围，如上述代码中 **Main** 方法囊括了 **Main** 方法的语句，**Program** 类囊括了类的方法，而 **namespace mycsharp** 命名空间囊括了此命名空间里的所有类。值得注意的是，**Visual Studio 2008** 为开发人员在编写程序的时候提供了诸多的智能提示，在完成一个类或一个变量时，系统会自动补全，而当鼠标放到一个大括号上的时候，编译器会指示开发人员此括号的范围，如图 2-1 所示。

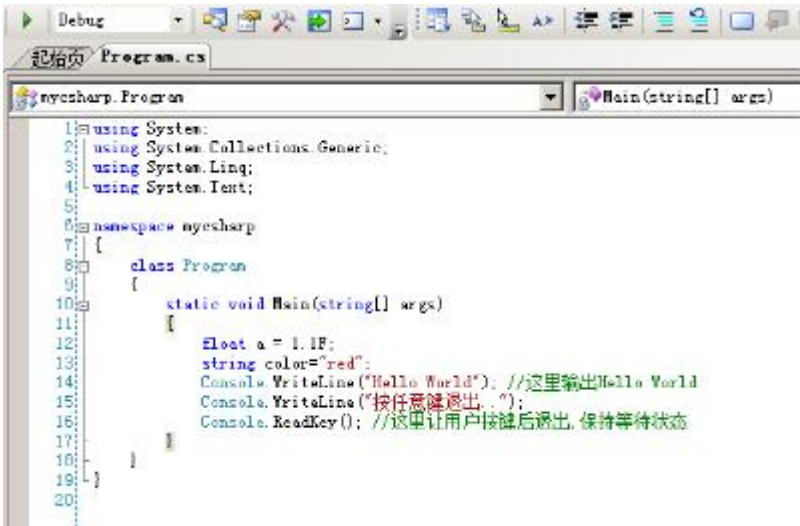


图 2-1 外围元素被标记

注意：在一个类内的所有方法都是独立的方法,所以每个大括号后面都不需要分号,同样对于命名空间里的所有类也是一样。

2.1.2 C# IDE 的代码设置

代码格式也是程序设计中一个非常重要的组成环节，他可以帮助用户组织代码和改进代码，也让代码具有可读性。具有良好可读性的代码能够让更多的开发人员更加轻松的了解和认知代码。按照约定的格式书写代码是一个非常良好的习惯，下面的代码示例说明了应用缩进、大小写敏感、空白区和注释等格式的原则。

```
using System;
using System.Collections.Generic;
using System.Linq;                                //使用 LINQ 命名空间
using System.Text;
namespace mycsharp                                //声明命名空间
{
    class Program                                  //主程序类
    {
        static void Main(string[] args)           //静态方法
        {
            Console.WriteLine("Hello World");      //这里输出 Hello World
            Console.WriteLine("按任意键退出.."); Console.ReadKey(); //这里让用户按键后退出,保持等待状态
        }
    }
}
```

1. 缩进

缩进可以帮助开发人员阅读代码，同样能够给开发人员带来层次感。读者可以从以上代码看出这一串代码让人能够很好的分辨区域，非常方便的就能找到 **Main** 方法的代码区域，这是因为括号都是有层次的。

缩进让代码保持优雅，同一语句块中的语句应该缩进到同一层次，这是一个非常重要的约定，因为它



直接影响到代码的可读性。虽然缩进不是必须的，同样也没有编译器强制，但是为了在不同人员的开发中能够进行良好的协调，这是一个值得去遵守的约定。

## 2. 大小写敏感

C#是一种对大小写敏感的编程语言。可能 **php** 等其他语言的开发人员不太适应大小写敏感，但是在 C#中，其语法规则的确是对字符串中字母的大小写敏感的，例如“C Sharp”、“c Sharp”、“c sHaRp”都是不同的字符串，在编程中应当注意。

## 3. 空白

C#编译器会忽略到空白。使用空白能够改善代码的格式，提高代码的可读性。但是值得注意的是，编译器不对引号内的任何空白做忽略，在引号内的空格作为字符串存在。

## 4. 注释

在 C/C++里，编译器支持开发人员编写注释，以便开发人员能够方便的阅读代码。当然，在 C#里也一样继承了这个良好的习惯。之所以这里说的是习惯，是因为编写注释同缩进一样，没有人强迫要编写注释，但是良好的注释习惯能够让代码更加优雅和可读，谁也不希望自己的代码在某一天过后自己也不认识了。

注释的写法是以符号“/\*”开始,并以符号“\*/”结束，这样能够让开发人员更加轻松的了解代码的作用，同时，也可以使用符号“//”双斜线来写注释，但是这样的注释是单行的，示例代码如下所示。

```
/*
 * 多行注释
 * 本例演示了在程序中写注释的方法
   在注释内也可以不要开头的*号
 */
//单行注释,一般对单个语句进行注释
```

## 5. 布局风格

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello World");           //这里输出 Hello World
        Console.WriteLine("按任意键退出.."); Console.ReadKey(); //这里让用户按键后退出,保持等待状态
    }
}
```

从以上代码可以看出，程序中使用了缩进、大小写敏感，空白区和注释等，但是这个代码风格依旧不是最好，可以修改代码让代码更加“好看”。这里能够将代码进行修正，修正后的示例代码如下所示。

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello World");           //这里输出 Hello World
        Console.WriteLine("按任意键退出..");
        Console.ReadKey();                           //这里让用户按键后退出,保持等待状态
    }
}
```

这种布局风格让开发人员感觉到耳目一新，这样更能方便更多的开发人员阅读源代码。如果打开一千行或更多代码量的源文件时，其编码格式都是标准的风格的话，不管是谁再接手去阅读，都能尽快上手。不仅如此，在软件开发当中，应该规定好每个人都使用同样的布局风格，让团队能够协调运作。

## 2.2 变量

在任何编程语言中，无论是传统的面向过程还是面向对象都必须使用变量。因此，变量都有自己的数

据类型，在使用变量的时候，必须使用相同的数据类型进行运算。在程序的运行中，计算中临时存储的数据都必须用到变量，变量的值也会放置在内存当中，由计算机运算后再保存到变量中，由此可见，变量在任何的应用程序开发中都是非常基础也是非常重要的。同样，在 **C#**中也需要变量对数据进行存储，本节将会介绍 **C#**的基本语法、数据类型、变量、枚举等。

2.2.1 定义

要声明一个变量就需要为这个变量找到一个数据类型，在 **C#**中，数据类型由**.NET Framework** 和 **C#**语言来决定，表 2-1 列举了一些预定义的数据类型。

表 2-1 预定义数据类型

预定义类型	定义	字节数
byte	0~255之间的整数	1
sbyte	-128~127之间的整数	1
short	-32768~32767之间的整数	2
ushort	0~65535之间的整数	2
int	-2147483648~2147483647之间的整数	4
uint	0~4294967259之间的整数	4
long	-9223372036854775808~9223372036854775807之间的整数	8
ulong	0~18445744073709551615之间的整数	8
bool	布尔值,true of false	1
float	单精度浮点值	4
double	双精度浮点值	8
decimal	精确的十进制值,有28个有效单位	12
object	其他所有类型的基类	N/A
char	0~65535之间的单个Unicode字符	2
string	任意长度的Unicode字符序列	N/A

一个简单的声明变量的代码如下所示：

```
int s; //声明整型变量
float myfloat; //声明浮点型变量
```

上述代码声明了一个整型的变量 **s**，同时也声明了一个单精度浮点型变量 **myfloat**。

2.2.2 值类型

这种类型的对象总是直接通过其值使用，不需要对它进行引用。基于值类型的变量直接包含值。并且，所有的 **C#**局部变量都需要初始化后才可以使使用，值类型同样如此，初始化代码如下所示。

```
int s; //声明整型变量
s = new int(); //声明整型变量
s = 3; //初始化变量
```

上式等同于如下代码。

```
int s; //声明整型变量
s = 3; //初始化变量
```

所有的值类型均隐式的派生自 **System.ValueType**，并且值类型不能派生出新的类。值的类型不能为 **null**，但是可空类型允许将 **null** 值赋给值类型，在上面的代码中，程序通过默认的构造函数给为变量 **s** 初始化并赋值。

2.2.3 引用类型

引用类型的变量又称为对象，是可存储对实际数据的引用。常见的引用类型有 **class**、**interface**、**delegate**、**object** 和 **string**。多个引用变量可以附加于一个对象，而且某些引用可以不附加于任何对象，如果声明了一

个引用类型的变量却不给他赋给任何对象，那么它的默认值就是 **null**。相比之下，值类型的值不能为 **null**。

2.3 变量规则

声明变量并不是随意声明的，变量的声明有自己的规则。在 **C#**中，应用程序包含许多关键字，包括 **int** 等是不能够声明为变量名的，如 **int int** 是不允许的，在进行变量的声明和定义时，需要注意变量名称是否与现有的关键字重名。

2.3.1 命名规则和命名习惯

命名规则就是给变量取名的一种规则，一般来说，命名规则就是为了让开发人员给变量或者命名空间取个好名，不仅要好记，还要说明一些特性。在 **C#**里面，有常用的一些命名的习惯如下。

- ❑ **Pascal** 大小写形式：所有单词的第一个字母大写，其他字母小写。
- ❑ **Camel** 大小写形式：除了第一个单词，所有单词的第一个字母大写，其他字母小写。

当然，在其他编程中，不同的开发人员可能遇到了一些不一样的命名规则和命名习惯，但是在 **C#**中，推荐使用常用的一些命名习惯，这样能保证代码的优雅性和可读性。同时，也应该避免使用相同名称的命名空间或与系统命名相同的变量，如以下代码所示：

```
string int; //系统会提示出错
```

运行上述代码时系统会提示错误，因为字符串“**int**”是一个关键字，当使用关键字做变量名时，编译器会混淆该变量是变量还是关键字，所以系统会提示错误。所以，在变量声明时应该避免变量名称与关键字重名，如果变量名称与关键字重名，编译器就会报错，**C#**中常用的关键字如表 2-2 所示：

表 2-2 不应使用的关键字名称

AddHandler	AddressOf	Alias	And	Ansi	As
Assembly	Auto	BitAnd	BitNot	BitOr	BitXor
Boolean	ByRef	Byte	ByVal	Call	Case
Catch	CBool	CByte	CChar	CDate	CDec
CDbl	Char	CInt	Class	CLng	CObj
Const	CShort	CSng	CStr	CType	Date
Decimal	Declare	Default	Delegate	Dim	Do
Double	Each	Else	Elseif	End	Enum
Erase	Error	Event	Exit	ExternalSource	False
Finally	For	Friend	Function	Get	GetType
Goto	Handles	If	Implements	Imports	In
Inherits	Integer	Interface	Is	Let	Lib
Like	Long	Loop	Me	Mod	Module
MustInherit	MustOverride	MyClass	Namespace	MyBase	New
Next	Not	Nothing	NotInheritable	NotOverridable	Object
On	Option	Optional	Or	Overloads	Overridable
Overrides	ParamArray	Preserve	Private	Property	Protected
Public	RaiseEvent	ReadOnly	ReDim	Region	REM
RemoveHandler	Resume	Return	Select	Set	Shadows
Shared	Short	Single	Static	Step	Stop
String	Structure	Sub	SyncLock	Then	Throw
To	True	Try	TypeOf	Unicode	Until
Variant	When	While	With	WithEvents	WriteOnly
Xor	eval	extends	instanceof	package	var

注意：标识符、参数名、函数名都不需要使用缩写。如果要使用缩写，超过两个字符以上的缩写都应该使用 **Camel** 大写格式。

2.3.2 声明并初始化变量

在程序代码编写中，需要大量的使用变量和读取变量的值，所以需要声明一个变量来表示一个值。这个变量可能描述是一个人的年龄，也可能是一辆车的颜色。在声明了一个变量之后，就必须给这个变量一个值，只有在给变量值之后能够说明这个变量被初始化。

1. 语法

声明变量的语法非常简单，即在数据类型之后编写变量名，如一个人的年龄（**age**）和一辆车的颜色（**color**），声明代码如下所示。

```
int age;           //声明一个叫 age 的整型变量,代表年龄
string color;      //声明一个叫 color 的字符串变量,代表颜色
```

上述代码声明了一个整型变量 **age** 和一个字符串型变量 **color**，由于年龄的值不会小于 **0** 也不会大于 **100**，所以在声明时可以使用数字类型进行声明。

2. 初始化变量

变量在声明后还需要初始化，例如“我年龄 **21** 岁，很年轻，我想买一辆红色的车”，那么就需要对相应的变量进行初始化，示例代码如下所示。

```
int age;           //声明一个叫 age 的整型变量,代表年龄
string color;      //声明一个叫 color 的字符串变量,代表颜色
age = 21;          //声明始化, 年龄 21 岁
color = "red";     //声明始化, 车的颜色为红色
```

上述代码也可以合并为一个步骤简化编程开发，示例代码如下所示。

```
int age=1;          //声明并初始化一个叫 age 的整型变量,代表年龄
string color="red"; //声明初始化
```

3. 赋值

在声明了一个变量之后，就可以给这个变量赋值了，但是当编写以下代码就会出错，示例代码如下。

```
float a = 1.1;      //错误的声明浮点类型变量
```

当运行了以上代码后会提示错误信息：不能隐式地将 **Double** 类型转换为“**float**”类型;请使用“**F**”后缀创建此类型。从错误中可以看出，将变量后缀增加一个“**F**”即可，示例代码如下所示。

```
float a = 1.1F;     //正确的声明浮点类型变量
```

运行程序，程序就能够编译并运行了。这是因为若无其他指定，**C#**编译器将默认所有带小数点的数字都是 **Double** 类型，如果要声明成其他类型，可以通过后缀来指定数据类型，如表 2-3 将展示一些可用的后缀，并且后缀可用小写。

表 2-3 可用的后缀表

后缀	描述
U	无符号
L	长整型
UL	无符号长整型
F	浮点型
D	双精度浮点型
M	十进制
L	长整型

4. 转义字符

在开发过程当中，如果需要将单引号或者双引号输出，或将单引号等字符作为字符串输出，就会发现在字符串中单引号或者双引号等字符是不能直接进行输出呈现。为了解决这一问题，于是引入了转义字符，常用的转义字符表如表 2-4 所示。

表 2-4 转义字符表

换码序列	字符名称
\'	单引号
\"	双引号



\\	反斜杠
\0	空字符
\a	警报符
\b	退格
\f	换页
\n	换行
\r	回车
\t	水平制表符
\v	垂直制表符

若在应用程序开发过程中，需要在程序里的字符串中编写一个双引号并进行输出，可以使用转义字符进行输出，示例代码如下所示。

```
string str="this is \" "; //使用转义字符
```

6. 设置断点

在 Visual Studio .NET 开发环境中，为用户提供了在开发应用程序时查看变量值的工具，只需要在查看的变量上设置断点，以调试模式运行应用程序，就可以在调试窗口中查看变量的值。在代码编辑窗口单机左边的空白处可直接设置断点。断点以红色圆点标识。也可以在调试菜单中单击【切换断点】按钮，或使用快捷键【F9】键来设置断点，如图 2-2 所示。

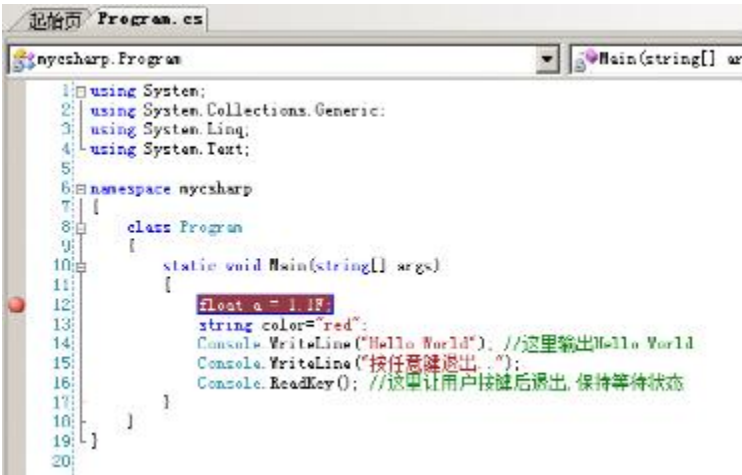


图 2-2 设置断点

按下【F5】键或在菜单栏中的调试菜单中单击【启动调试】按钮都可以运行程序。当程序开始运行，程序从 Main 入口运行并直至遇到断点，遇到断点后程序将停止运行，如图 2-3 所示。同时开发环境会高亮显示下一条即将执行的代码，同时调试查看窗口会显示，并呈现变量的当前值，如图 2-4 所示，。



图 2-3 运行到断点,提示下一步执行的代码

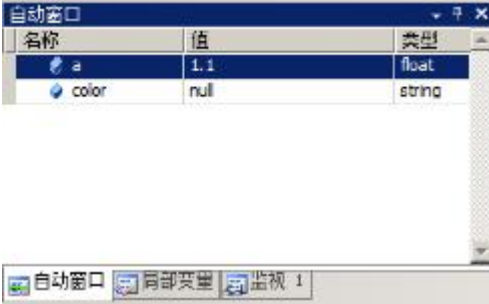


图 2-4 显示当前值

在调试完成后，可以通过快捷键【Shift+F5】停止调试，也可以在菜单栏中的【调试】菜单里的【停止调试】选项中停止应用程序的调试。如果需要继续执行，可以按下【F5】键或在调试菜单中选择【继续执行到下一个断点】选项进行执行。开发人员还可以通过使用快捷键【F10】，或在调试菜单中选择【逐过程】或【逐语句】每次只执行一条语句，方便对代码中变量变化的查看。

2.3.3 数组

数组是一个引用类型，开发人员能够声明数组并初始化数据进行相应的数组操作，数组是一种常用的数据存放方式。

1. 数组的声明

数组的声明方法是在数据类型和变量名之间插入一组方括号，声明格式如下所示。

```
string[] groups;                                     //声明数组
```

以上语句声明了一个变量名为 **groups** 的数组，其数据类型为 **string**。声明了一个数组之后，并没有为此数组添加内容初始化，需要对数组初始化，才能使用数组。

2. 数组的初始化

开发人员可以对数组进行显式的初始化，以便能够填充数组中的数据，初始化代码如下所示。

```
string[] groups={"asp.net","c#","control","mvc","wcf","wpf","linq"};    //初始化数组
```

值得注意的是，与平常的逻辑不同的是，数组的开始并不是 **1**，而是 **0**。以上初始化了 **groups** 数组，所以 **groups[0]** 的值应该是 “asp.net” 而不是 “c#”，相比之下，**group[1]** 的值才应该是 “c#”。

3. .NET 中数组的常用的属性和方法

在.NET 中，.NET 框架为开发人员提供了方便的方法来对数组进行运算，专注于逻辑处理的开发人员不需要手动实现对数组的操作。这些常用的方法如下所示。

- ❑ **Length** 方法用来获取数组中元素的个数。
- ❑ **Reverse** 方法用来反转数组中的元素，可以针对整个数组，或数组的一部分进行操作。
- ❑ **Clone** 方法用来复制一个数组。

对于数组的操作，可以使用相应的方法进行数据的遍历、查询和反转，示例代码如下所示。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;                                     //声明文本命名空间
namespace myArray                                     //主应用程序类
{
    class Program
    {
        static void Main(string[] args)
        {
            string[] groups={"asp.net","c#","control","mvc","wcf","wpf","linq"};    //初始化一个数组
            int count = groups.Length;                                           //获取数组的长度
            Console.WriteLine("-----数组长度-----");
            Console.WriteLine(count.ToString());                                //输出数组的长度
            Console.WriteLine("-----原数组元素值-----");
            for (int i = 0; i < count; i++)                                       //遍历输出数组元素
            {
                Console.WriteLine(groups[i]);                                   //输出数组中的元素
            }
        }
    }
}
```

按 **F5** 运行后运行结果如图 2-5 所示。





图 2-5 数组运行结果

从上述结果中可以看出，程序遍历了数组并将数组的内容全部输出。在进行数组中的内容输出时，需要使用循环语句进行输出数组的遍历和输出，循环语句的用法会在后面讲解。

2.3.4 声明并初始化字符串

字符串是计算机应用程序开发中常用的变量，在文本输出、字符串索引、字符串排序中都需要使用字符串。

1. 声明及初始化字符串

字符串类型（**string**）是程序开发中最常见的数据类型，如上一小节声明的数组中的任意一个元素都是一个字符串。由于数组也是有其数据类型的，所以声明的数组是一个字符串型的数组。字符串的声明方式和其他的数据类型声明方式相同，字符串变量的值必须在“ ”双引号之间，示例代码如下所示。

```
string str="Hello World!"; //声明字符串
```

在 2.3.2 中讲解了转义字符，当开发人员试图在字符串中间输入一些特殊符号的时候，会发现编译器报错，示例代码如下所示。

```
string str="Hello "World!";
```

在 **Visual Studio 2008** 中编写上述代码，会发现褐色的字符串被分开了，并且编译器报错“常量中有换行符”，因为字符串中的“ ”符号被当成是字符串的结束符号。为了解决这个问题，就需要用到转义字符。示例代码如下所示。

```
string str="Hello \"World!"; //使用转义字符
```

编译并运行，运行结果如图 2-6 所示。

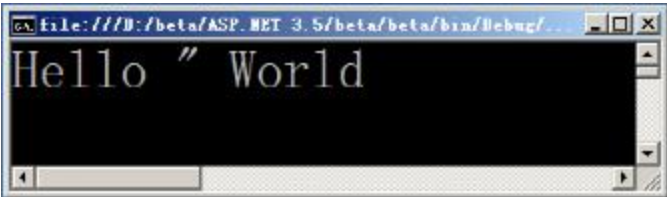


图 2-6 使用转义字符初始化字符串

在程序中的开发中，经常需要引用和打开某个文件，打开文件的操作必须要引用文件夹的地址。例如要打开我的文档里的内容，就必须在地址栏敲击 **D:\Users\Administrator\Documents**，在编写程序的时候，“\”字符却无法编写在字符串中，同样也需要转义字符，示例代码如下所示。

```
string str="D:\\Users\\Administrator\\Documents"; //使用转义字符
```

编译并运行，运行结果如图 2-7 所示。



图 2-7 使用转义字符初始化字符串

2. 使用逐字符串

如果字符串初始化为逐字符串，编译器会严格的按照原有的样式输出，无论是转义字符中的换行符还是制表符，都会按照原样输出。逐字符串的声明只需要在双引号前加上字符“@”即可，示例代码如下所示。

```
string str=@"文件地址:D:\Users\Administrator\Documents \t"; //逐字符串
```

编译并运行上述代码，运行结果如图 2-8 所示。

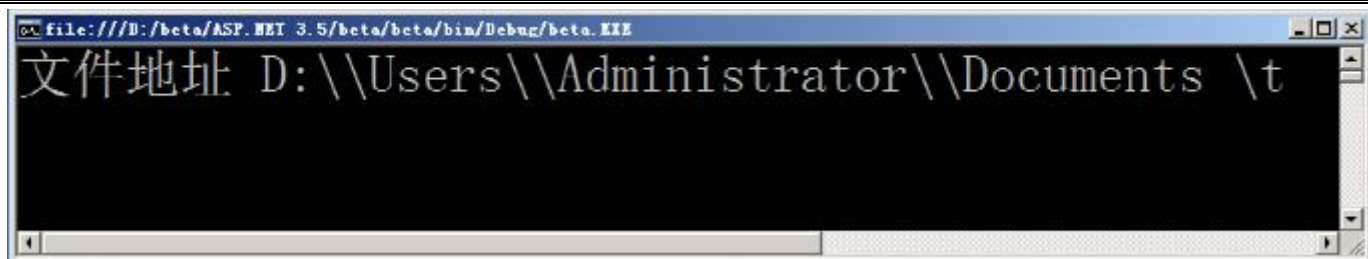


图 2-8 使用逐字符串

但是对于双引号而言，逐字符串依旧无法正确进行输出。若要使用逐字符串进行双引号的输出，则必须使用引号进行编写以便正确的输出双引号，示例代码如下所示。

```
string str=@"\"; //输出双引号
```

编译并运行，运行结果如图 2-9 所示。

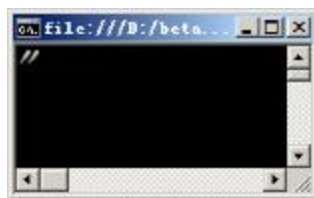


图 2-9 逐字符串输出双引号

## 3. 字符串格式化

在字符串操作时，很多地方需要用到字符串格式化，使用 **Console.WriteLine** 方法就能够实现字符串格式化，字符串格式化代码如下所示。

```
string str = "Guojing"; //声明字符串
Console.WriteLine("Hi! Myname is {0},I love C#",str); //字符串格式化输出
Console.ReadKey(); //等待用户按键
```

运行上述代码，其结果如图 2-10 所示。



图 2-10 字符串格式化

可以从运行结果看出，**Console.WriteLine** 方法中，前一个传递的参数中的{0}被后一个传递的参数 **str** 替换。例子中的“{0}”被称为占位符，用于标识一个参数，括号中的数字指定了参数的索引。同时，输出方法也可以格式化多个字符串，示例代码如下所示。

```
string str = "Guojing";
string str2 = "C#";
Console.WriteLine("Hi! Myname is {0},I love {1}",str,str2); //格式化多个字符串输出
```

运行结果如图 2-11 所示。



图 2-11 多个占位符格式化字符串

## 2.3.5 操作字符串

在 **C#**中，为字符串提供了快捷和方便的操作，使用 **C#**提供的类能够进行字符串的比较、字符串的连接、字符串的拆分等操作，方便了开发人员进行字符串的操作。

## 1. 比较字符串

如果需要比较字符串，有两种方式，一种是值比较，一种是引用比较。值比较可以直接使用运算符“==”进行比较，示例代码如下所示。

```
string str = "Guojing";           //声明字符串
string str2 = "C#";              //声明字符串
if (str == str2)                 //使用 “==” 比较字符串
{
    Console.WriteLine("字符串相等"); //输出不相等信息
}
else
{
    Console.WriteLine("字符串不相等"); //输出相等信息
}
```

当判断两个字符串是否指向同一个对象时，可以使用 **Compare** 方法判定两个字符串是否指向同一个对象，示例代码如下所示。

```
string str = "Guojing";           //声明字符串
string str2 = "C#";              //声明字符串
if (str.CompareTo(str2) > 0)      //使用 Compare 比较字符串
{
    Console.WriteLine("字符串不相等"); //输出不相等信息
}
else
{
    Console.WriteLine("字符串相等"); //输出相等信息
}
```

编译上述代码并运行，其结果如图 2-12 所示。



图 2-12 比较字符串

在上述代码运行后，如果字符串不相等，则输出“字符串不相等”字符，否则会输出“字符串相等”。

## 2. 连接字符串

当一个字符串被创建，对字符串的操作的方法实际上是对字符串对象的操作。其返回的也是新的字符串对象，与 **int** 等数据类型一样，字符串也可以使用符号“+”进行连接，代码如下所示。

```
string str = "Guojing is A C# "; //声明字符串
string str2 = "Programmer";      //声明字符串
Console.WriteLine(str+str2);      //连接字符串
```

在上述例子中，声明并初始化两个字符串型变量 **str** 和 **str2**，并输出 **str+str2** 的结果，运行结果如图 2-13 所示。

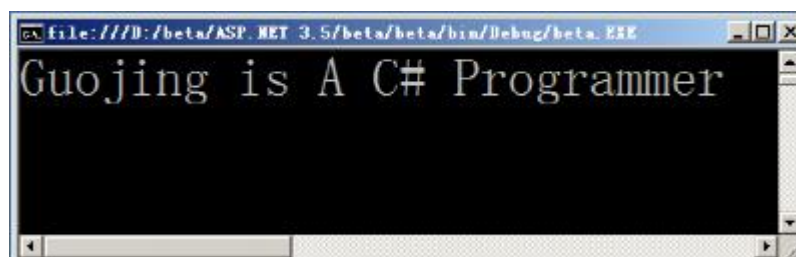


图 2-13 字符串连接

同样的，**str** 和 **str2** 也可以为新的字符串赋值，代码如下所示。

```
string mystr = str + str2; //连接字符串
```

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

```
//输出字符串
```

上述代码运行结果同样如图 2-13 所示，这里就不再运行演示。

注意：在上述代码中，**mystr**被初始化 **str** 和 **str2** 的“和”，但是在运算过程当中，**str** 和 **str2** 的值都没有被改变。

### 3. 拆分字符串

能够连接一个字符串，同样也可以拆分一个字符串。**.NET Framework** 提供了若干方法供拆分字符串，示例代码如下所示。

```
string str = "Guojing is A C# Programmer";
Console.WriteLine(str.IndexOf("is").ToString());
Console.ReadKey();
```

```
//声明字符串
```

```
//拆分字符串
```

编译运行后，可以看到返回的结果是 **8**，说明 **is** 是字符串从开始第 **8** 位才找到 **is**，若搜索不到查询的字符串，则返回 **-1**。当字符串拆分成子字符串之后，可以通过 **Split** 方法对字符串进行分割，代码如下所示。

```
string str = "BeiJing,Shanghai,GuangZhou,WuHan,ShenYang"; //初始化字符串
string[] p = str.Split(','); //使用 Split 方法分割并存入数组
for (int i = 0; i < p.Length; i++) //遍历显示
{
    Console.WriteLine(p[i]); //输出字符串
}
```

上述代码第一句声明并初始化了一个字符串，第二句使用 **Split** 函数按照逗号来分割字符串，并存入到数组 **p** 内，然后遍历显示数组元素。

注意：使用 **Split** 函数的时候，通常情况下只能使用字符对字符串进行分割，所以使用的是单引号。

### 4. 更改字符串大小写

在**.NET**中，系统为开发人员提供了将字符串更改为大写或小写的方法，这两个方法分别为 **ToUpper()** 和 **ToLower()**。使用该方法能够进行字符串的大小写转换，示例代码如下所示。

```
string str = "BeiJing,Shanghai,GuangZhou,WuHan,ShenYang"; //声明字符串
Console.WriteLine(str.ToUpper()); //转换成大写
Console.WriteLine(str.ToLower()); //转换成小写
Console.ReadKey(); //等待用户输入
```

### 5. 常用的字符串操作

在 **C#** 软件开发过程，字符串是使用率最高的数据类型之一，开发人员往往需要对字符串进行大量的操作。这里介绍一些经常使用的字符串操作如判断字符串是否为空，替换字符串中相应的字符等等。判断字符串是否为空会经常在程序中使用，以保证用户输入的完整性，示例代码如下所示。

```
string str = "BeiJing,Shanghai,GuangZhou,WuHan,ShenYang"; //声明字符串
if (String.IsNullOrEmpty(str)) //使用 String 类的静态方法
{
    Console.WriteLine("字符串为空"); //输出字符串为空的信息
}
else
{
    Console.WriteLine("字符串不为空"); //输出字符串不为空的信息
}
```

当需要对字符串执行替换操作时，可以使用 **Replace** 方法进行字符串的替换，示例代码如下所示。

```
string str = "BeiJing,Shanghai,GuangZhou,WuHan,ShenYang"; //声明字符串
str = str.Replace("ShenYang", "ShanXi"); //使用 Replace 方法
Console.WriteLine(str); //输出字符串
```

大多数应用程序都是对字符串进行操作，这里简单的介绍几种常用的字符串的操作，熟练掌握字符串的操作对应用程序开发有很大的好处。



2.3.6 创建和使用常量

常量是一般在程序开发当中不经常更改的变量，如  $\pi$  值、税率或者是数组的长度等。使用常量一般能够让代码更具可读性、更加健壮、便于维护。在程序开发当中，好的常量使用技巧对程序开发和维护都有好的影响，示例代码如下所示。

```
const double pi=3.1415926;           //常量 pi,  $\pi$ 
static void Main(string[] args)      //程序入口方法
{
    double r=2;                      //声明 double 类型常量
    double round = 2 * pi * r * r;    //使用常量
    Console.WriteLine(round.ToString()); //输出变量值
    Console.ReadKey();                //等待用户输入
}
```

上述代码非常简单，就是计算一个圆的圆周率。当代码非常长的时候，程序也会非常干练，容易阅读，如果在程序中出现了以下代码，也能够理解该表达式的作用。示例代码如下所示。

```
double Perimeter = 2 * pi * r;      //使用常量
```

就算是其他的开发人员阅读到上述代码，也能够轻易的了解该语句的作用就是求圆的周长，因为在前面定义了常量 **pi=3.1415926**；当程序中用到这个变量的时候，立刻就能够知道程序的作用。声明变量的方法，只需要在普通的变量格式前加上 **const** 关键字即可，声明代码如下所示。

```
const double pi=3.1415926;          //声明 const 变量
const int max = 500;                 //声明 const 变量
const long kilometer = 1000;        //声明 const 变量
```

使用 **const** 声明的变量能够在程序中使用，但是值得注意的是，使用 **const** 声明的变量不能够在后面的代码中对该变量进行重新赋值。

注意：使用 **const** 声明的变量如果在后面的代码中进行重新赋值或更改，则编译器会提示错误。**const** 修饰符通常用于不常更改的变量的修饰。

2.3.7 创建并使用枚举

枚举类型是一组已命名的常量，它是一种用户自定义类型，开发人员可以自行创建枚举类型，声明枚举变量并初始化。枚举变量和普通的变量相比，确保了只将预定的值赋予变量，让代码更加容易维护。在编写代码的时候，允许以简单容易辨认的名字作为变量名，使代码更具有可读性。同时，如果开发人员声明枚举变量，**Visual Studio 2008** 还能够提供的智能感知功能能够让代码更加方便的输入。

1. 声明及初始化枚举

如果需要创建枚举类型，就需要使用 **enum** 关键字，指定一个类型名称，如 **int** 等，然后列举出枚举可以使用的值，示例代码如下所示。

```
enum color {red,yellow,green,blue}; //声明枚举
```

上述代码创建了一个 **color** 类型，然后声明了这个类型的变量，并使用枚举成员赋值，示例代码如下所示。

```
class Program
{
    enum color {red,yellow,green,blue}; //声明枚举
    static void Main(string[] args)    //主程序入口方法
    {
        Console.WriteLine(color.green); //查看枚举成员变量 green
        Console.ReadKey();               //等待用户按键
    }
}
```

```
    }
```

编译并运行上述代码，其输出结果为 **green**，说明在程序中已经对枚举变量中的成员初始化了。

2. 使用枚举类型

枚举是用户自定义类型，所以在程序中可以引用用户的自定义类型进行自定义类型的变量的创建，示例代码如下所示。

```
color mycolor = color.green; //使用枚举类型
```

注意：枚举类型的定义只能在命名空间或类内声明，否则编译器会报错。

4. 枚举成员的赋值和常用类型

声明并初始化枚举类型，也可以对枚举成员的值进行初始化，示例代码如下所示。

```
enum color {red=1,yellow=2,green=3,blue=1}; //枚举成员的赋值
```

不仅可以为枚举成员初始化，也可以为枚举成员定义基本类型，示例代码如下所示。

```
enum color:int {red=1,yellow=2,green=3,blue=1}; //定义基本类型
```

2.3.8 类型转换

在应用程序开发当中，很多的情况都需要对数据类型进行转换，以保证程序的正常运行。类型转换是数据类型和数据类型之间的转换，在.NET 中，存在着大量的类型转换，常见的类型转换代码如下所示。

```
int i = 1; //声明整型变量
Console.WriteLine(i); //隐式转换输出
```

在上述代码中 **i** 是整型变量而 **WriteLine** 方法的参数是 **Object** 类型，但是 **WriteLine** 方法依旧能够正确输出是因为系统将 **i** 的类型在输出的时候转换成了字符型。在.NET 框架中，有隐式转换和显式转换，隐式转换是一种由 **CLR** 自动执行的类型转换，如上述代码中的，就是一种隐式的转换（开发人员不明确指定的转换），该转换由 **CLR** 自动的将 **int** 类型转换成了 **string** 型。在.NET 中，**CLR** 支持许多数据类型的隐式转换，**CLR** 支持的类型转换列表如表 2-5 所示。

表 2-5 CLR支持转换列表

从该类型	到该类型
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

显式转换是一种明确要求编译器执行的类型转换。在程序开发过程中，虽然很多地方能够使用隐式转换，但是隐式转换有可能存在风险，显式转换能够通过程序捕捉进行错误提示。虽然隐式也会提示错误，但是显式转换能够让开发人员更加清楚的了解代码中存在的风险并自定义错误提示以保证任何风险都能够及早避免，示例代码如下所示。

```
int i = 1; //声明整型变量 i
float j = (float)i; //显式转换为浮点型
```

上述代码说明了显式转换的基本语法格式，具体语法格式如下所示。

```
type variable1=(cast-type)variable2;
```

注意：显式的转换可能导致数据的部分丢失，如 3.1415 转换为整型的时候会变成 3。

除了隐式的转换和显式的转换，还可以使用.NET 中的 **Convert** 类来实现转换，即使是两种没有联系的类型也可以实现转换。**Convert** 类的成员函数都是静态方法，当调用 **Convert** 类的方法时无需创建 **Convert** 对象，当使用显式的转换的时候，若代码如下所示，则编译器会报错。

string i = "1";	//声明字符串变量
int j = (int)i;	//显式转换为整型
Console.WriteLine(j);	//隐式转换为字符串

但是明显的是，字符串变量 **i** 的值是有可能转换成整型变量值 **1** 的，**Convert** 类能够实现转换，示例代码如下所示。

string i = "1";	//声明字符串变量
int j = Convert.ToInt32(i);	//显式转换为整型
Console.WriteLine(j);	//隐式转换为字符串

上述代码编译通过并能正常运行。**Convert** 类提供了诸多的转换功能，每个 **Toxx** 方法都将变量的值转换成相应.NET 简单数据类型的值，如 **Int16**、**Int32**、**String** 等。但是值得注意的是，并不是每个变量的值都能随意转换，示例代码如下所示。

string i = "haha";	//声明字符串变量
int j = Convert.ToInt32(i);	//错误的转换

上述代码中，**i** 的值是字符串“**haha**”，很明显，该字符串是无法转换为整型变量的。运行此代码后系统会抛出异常提示字符串“**haha**”不能够转换成整型常量。

## 2.4 编写表达式

在了解了 **C#** 中的数据类型、变量的声明和初始化方式、以及类型转换等基本知识，就需要了解如何进行表达式的编写。表达式在 **C#** 应用程序开发中非常的重要，本节将说明如何使用运算符创建和使用表达式。

### 2.4.1 表达式和运算符

表达式和运算符是应用程序开发中最基本也是最重要的一个部分，表达式和运算符组成一个基本语句，语句和语句之间组成函数或变量，这些函数或变量通过某种组合形成类。

#### 1. 定义

表达式是运算符和操作符的序列。运算符是个简明的符号，包括实际中的加减乘除，它告诉编译器在语句中实际发生的操作，而操作数既操作执行的对象。运算符和操作数组成完整的表达式。

#### 2. 运算符类型

在大部分情况下，对运算符类型的分类都是根据运算符所使用的操作数的个数来分类的，一般可以分为三类，这三类分别如下所示。

- ❑ 一元运算符：只使用一个操作数，如 (!)，自增运算符 (++) 等等，如 **i++**。
- ❑ 二元运算符：使用两个操作数，如最常用的加减法，**i+j**。
- ❑ 三元运算符：三元运算符只有 (?:) 一个。

除了按操作数个数来分以外，运算符还可以按照操作数执行的操作类型来分，如下所示。

- ❑ 关系运算符。
- ❑ 逻辑运算符。
- ❑ 算术运算符。
- ❑ 位运算符。
- ❑ 赋值运算符。
- ❑ 条件运算符。
- ❑ 类型信息运算符。
- ❑ 内存访问运算符。
- ❑ 其他运算符。

在应用程序开发中，运算符是最基本也是最常用的，它表示着一个表达式是如何进行运算的。常用的运算符如表 2-6 所示。

表 2-6 常用的运算符

运算符类型	运算符
元运算符	(x),x.y,f(x),a[x],x++,x--,new,sizeof,checked,unchecked
一元运算符	+,~,!,++x,--x,(T)x,
算术运算符	+,*,/,%
位运算符	<<,>>,& ,^,~
关系运算符	<,>,<=,>=,is,as
逻辑运算符	& ,^
条件运算符	&&  ,?
赋值运算符	=,+=,-=,*=,/=,<<=,>>=,&=,^=, =

正如表 2-5 中所示，C#编程中所需要使用到的运算符都能够通过相应的类别进行相应的分类，但其分类的标准并不是唯一的。

3. 算术运算符

程序开发中常常需要使用算术运算符，算术运算符用于创建和执行数学表达式，以实现加、减、乘、除等基本操作，示例代码如下所示。

```
int a = 1; //声明整型变量
int b = 2; //声明整型变量
int c = a + b; //使用+运算符
int d = 1 + 2; //使用+运算符
int e = 1 + a; //使用+运算符
int f = b - a; //使用-运算符
int f = b / a; //使用/运算符
```

注意：当除数为 0，系统会抛出 DivideByZeroException 异常，在程序开发中应该避免出现逻辑错误，因为编译器不会检查逻辑错误，只有在运行中才会提示相应的逻辑错误并抛出异常。

在算术运算符中，运算符“%”代表求余数，示例代码如下所示。

```
int a = 10; //声明整型变量
int b = 3; //声明整型变量
Console.WriteLine((a%b).ToString()); //求 10 除以 3
```

上述代码实现了“求 10 除以 3”的功能，其运行结果为 1。在 C#的运算符中还包括自增和自减运算符，如“++”和“--”运算符。++和--运算符是一个单操作运算符，将目的操作数自增或自减 1。该运算符可以放置在变量的前面和变量的后面，都不会有任何的语法错误，但是放置的位置不同，实现的功能也不同，示例代码如下所示。

```
int a = 10; //声明整型变量
int a2 = 10; //声明整型变量
int b = a++; //执行自增运算
int c = ++a2; //执行自增运算
Console.WriteLine("a is " + a); //输出 a 的值
Console.WriteLine("b is " + b); //输出 b 的值
Console.WriteLine("c is " + c); //输出 c 的值
```

运行结果如图 2-14 所示。



图 2-14 ++运算符



由运行结果所示，变量 **a**、**a2** 为 **10**，在使用了++运算符后，**a** 和 **a2** 分别变为了 **11**，而 **b** 的赋值语句代码中使用的为后置自增运算符，示例代码如下所示。

```
int b = a++; //b=10
```

当执行了上述代码后，**b** 的值为 **10**，而 **a** 会自增为 **11**，因为上述代码首先会将变量 **a** 的值赋值给 **b** 变量，赋值后再进行自增。而 **c** 的赋值语句中使用的为前置自增运算符，示例代码如下所示。

```
int c = ++a2; //c=11
```

当执行了上述代码后，变量 **c** 的值为 **11**，是因为在执行自增操作时，首先进行自增，再将 **a2** 变量的值赋值给 **c**。当运算符在操作数后时，操作数会赋值给新的变量，然后再自增 **1**，当运算符在操作数前时，操作数会先进行自增或自减，然后再赋值给新变量。

4. 关系运算符

关系运算符用于创建一个表达式，该表达式用来比较两个对象并返回布尔值。示例代码如下所示。

```
string a="nihao"; //声明字符串变量 a
string b="nihao"; //声明字符串变量 b
if (a == b) //使用比较运算符
{
    Console.WriteLine("相等"); //输出比较相等信息
}
else
{
    Console.WriteLine("不相等"); //输出比较不相等信息
}
```

关系运算符如“>”，“<”，“>=”，“<=”等同样是比较两个对象并返回布尔值，示例代码如下所示。

```
string a="nihao"; //声明字符串变量 a
string b="nihao"; //声明字符串变量 b
if (a == b) //比较字符串，返回布尔值
{
    Console.WriteLine((a == b).ToString()); //输入比较后的布尔值
}
else
{
    Console.WriteLine((a == b).ToString()); //输入比较后的布尔值
}
```

编译并运行上述程序后，其输出为 **true**。若条件不成立，即如 **a** 不等于 **b** 的变量值，则返回 **false**。该条件所以可以直接编写在 **if** 语句中进行条件筛选和判断。

技巧：在使用判断的时候，可以直接使用表达式，只要表达式的返回值是布尔型的即可，同样也可以

使用类型转换 **Convert.ToBoolean** 方法转换。

初学者很容易错误的使用关系运算符中的“=”号，因为初学者通常会将等于运算符编写为“=”号，示例代码如下所示。

```
if (a = b) //使用布尔值布尔值
```

在这里，“=”号不等于“==”号，“=”号的意义是给一个变量赋值，而“==”号是比较两个变量的值是否相等，如果写成上述代码，虽然编译器不会报错，但是其运行过程就不是开发人员想象的流程。

5. 逻辑运算符

逻辑运算符和布尔类型组成逻辑表达式。**NOT** 运算符“!”使用单个操作数，用于转换布尔值，即取非，示例代码如下所示。

```
bool myBool = true; //创建布尔变量
bool notTrue = !myBool; //使用逻辑运算符
```

与其他编程语言相似的是，**C#**也使用 **AND** 运算符“&&”。该运算符使用两个操作数做与运算，当有一个操作数的布尔值为 **false** 时，则返回 **false**，示例代码如下所示。

```
bool myBool = true;           //创建布尔变量
bool notTrue = !myBool;      //使用逻辑运算符取反
bool result = myBool && notTrue; //使用逻辑运算符计算
```

同样，C#中也使用“||”运算符来执行 **OR** 运算，当有一个操作数的布尔值为 **true** 时，则返回 **true**。当使用“&&”运算符和“||”运算符时，他们是短路（**short-circuit**）的，这也就是说，当一个布尔值能够由前一个或前几个操作数决定结果，那么就不取使用剩下的操作数继续运算而直接返回结果，示例代码如下所示。

```
bool myBool = true;           //创建布尔变量
bool notTrue = !myBool;      //使用逻辑运算符取反
bool result = myBool && notTrue; //使用逻辑运算符计算
bool other = true;           //创建布尔变量
if (result&&other)             //短路操作
{
    Console.WriteLine("true"); //输出布尔值
}
else
{
    Console.WriteLine("false"); //输出布尔值
}
```

从上述代码可以看到，变量 **other** 的值为 **true**，而 **result** 的值为 **false**，那么 **result&&other** 语句中，会直接返回 **false**，说明条件失败。另外，在逻辑运算符中还包括 **XOR** 异或运算符“^”，该运算符确定是否操作数是否相同，若操作数的布尔值相同，则表达式将返回 **false**。

在 C#应用程序开发中，并不支持从整型直接转换为布尔值，虽然在 C/C++中能够直接编写数值进行逻辑判断，示例代码如下所示。

```
int result = 1;           //使用整型
if (result)                //c++的转换
{
    cout<<"true";
}
else
{
    cout<<"false";
}
```

而上述代码如果在 C#中运行，编译器会报错，说明 C#并不支持显式的直接从 **int** 类型到 **bool** 类型的转换，但是使用 **Convert** 对象的静态方法可以实现同样的效果，代码如下所示。

```
int result = 1;
if (Convert.ToBoolean(result)) //使用 Convert 静态对象
{
    Console.WriteLine("true"); //输出布尔值
}
else
{
    Console.WriteLine("false"); //输出布尔值
}
```

注意：在编程语言中，非 0 的数值都为 **true**，虽然 C#不支持隐式的转换 **int** 到 **bool** 类型，但是 **Convert.ToBoolean** 静态方法提供了实现，任何非 0 的参数都将返回 **true**。

6. 位运算符

位运算符通常使用为模式来操作整数类型，这些位运算符非常实用。位运算符包括“<<”、“>>”、**AND**、**OR** 和 **XOR**。左移位运算符“<<”将整型中的位左移指定位数，每一次左移，整型变量的值将乘以 2，代码如下所示。

```
int result = 4;
```

Console.WriteLine((result << 1).ToString());

//左移并输出

运行结果为 8，操作原理如图 2-15 所示。

当使用 “<<” 运算符时，左移操作将舍弃移出的所有位，并用 0 来填充移入的位。同样，右移运算符 “>>” 也将操作数右移，每一次右移，整型变量的值将除以 2。AND 位运算符 “&” 通过逐位执行逻辑 AND 运算，从而生成新的操作数，AND 运算中，两个对应的值，若有一个值为 0，则全部为 0，原理如图 2-16 所示。

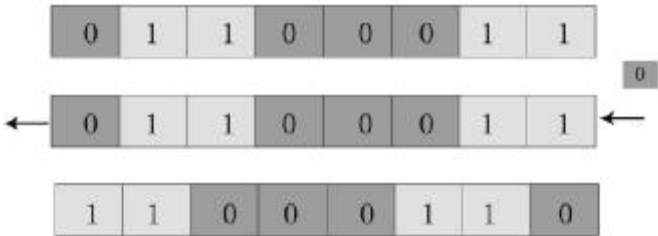


图 2-15 左移



图 2-16 AND 位运算

OR 位运算符 “|” 的是用方法和原理和 AND 位运算符 “&” 基本相同，其区别在于使用的是 OR 运算，当有一个值为 1，则结果为 1，其原理如图 2-17 所示。

XOR 位运算符 “^” 的用法和 AND 位运算符类似，其区别在于当两个值相同时，执行计算的结果为 0，否则为 1，其原理图如 2-18 所示。

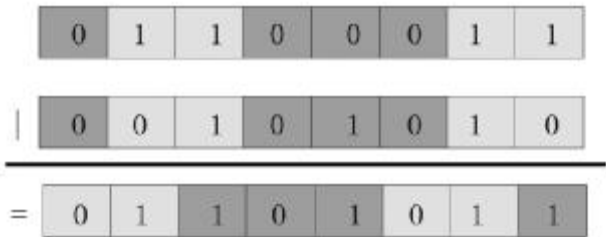


图 2-17 OR 位运算

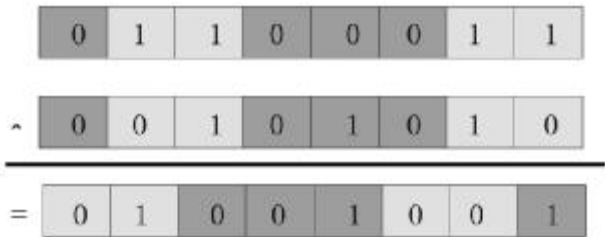


图 2-18 XOR 位运算

取补运算符 “~” 将生成整型类型的补码。如原值中的 1 将变为 0，而 0 则变为 1，原理图如 2-19 所示。



图 2-19 取补位运算

7. 赋值运算符

C#提供了几种类型的赋值运算符，最常见的就是 “=” 运算符。C#还提供了组合运算符，如 “+=”、“-=”、“\*=” 等。“=” 运算符通常用来赋值，示例代码如下。

int a,b,c;

a = b = c = 1;

//声明三个整型变量

//使用赋值运算符

上述代码声明并初始化 3 个整型变量 a、b、c 并初始化值这些变量的值为 1。加法赋值运算符 “+=” 将加法和赋值操作组合起来，先把第一个数值的值加上第二个数值的值再存放到第一个数值的，示例代码如下所示。

a += 1;

//进行自加运算

上述代码会将变量 a 的值加上 1 并再次赋值回 a，上述代码实现的功能和以下代码等效。

a = a + 1;

//不使用+=运算符

同样，“-=”，“\*=”，“/=” 都是将第一个数值的值与第二个数值的值做操作再存放到第一个数值，同样也支持位运算符，示例代码如下所示。

a <<= 1;

//移位运算

赋值运算符不仅支持加减乘除和位运算，同样也支持条件运算符，示例代码如下所示。

a &= 1;

//条件运算

8. 条件运算符

条件运算符“?:”需要三个操作数，示例代码如下所示。

```
bool ifisTrue=true;           //创建布尔值
string result = ifisTrue ? "true" : "false"; //使用三元条件运算符
Console.WriteLine(result.ToString()); //输出布尔值
```

上述代码中，使用了条件运算符“?:”，条件运算符“?:”会执行第一个条件，若条件成立，则返回“:”运算符前的一个操作数的数值，否则返回“:”运算符后的操作数的数值。上述代码中，变量 **ifisTrue** 为 **true**，则返回“**true**”。

技巧：当记忆条件运算符的时候，“?”可以被记忆为“条件成立吗”，如果成立，则执行第一个，否则执行第二个，如(1>2)?1:2 可以解释成 1 大于 2 吗，大于则返回 1，不大于则返回 2，这样有助于记忆和阅读。

2.4.2 运算符的优先级

开发人员需要经常创建表达式来执行应用程序的计算，简单的有加减法，复杂的有矩阵、数据结构等，在创建表达式时，往往需要一个或多个运算符。在多个运算符之间的运算操作时，编译器会按照运算符的优先级来控制表达式的运算顺序，然后再计算求值。例如在生活中也常常遇到这样的计算，如 **1+2\*3**。如果在程序开发中，编译器优先运算“+”运算符并进行计算就会造成错误的结果。

1. 运算顺序

表达式中常用的运算符的运算顺序如表 2-7 所示。

表 2-7 运算符优先级

运算符类型	运算符
元运算符	<b>X.y,f(x),a[x],x++,x--,new,typeof,checked,unchecked</b>
一元运算符	<b>+, -, !, ~, ++x, --x, (T)x</b>
算术运算符	<b>*, /, %</b>
位运算符	<b>&lt;&lt;, &gt;&gt;, &amp;,  , ^, ~</b>
关系运算符	<b>&lt;, &gt;, &lt;=, &gt;=, is, as</b>
逻辑运算符	<b>&amp;, ^,  </b>
条件运算符	<b>&amp;&amp;,   , ?</b>
赋值运算符	<b>=, +=, -=, *=, /=, &lt;&lt;=, &gt;&gt;=, &amp;=, ^=,  =</b>

当执行运算 **1+2\*3** 的时候，因为“+”运算符的优先级比“\*”运算符的优先级低，则当编译器编译表达式并进行运算的时候，编译器会首先执行“\*”运算符的乘法操作，然后执行“+”运算符的加法操作。当需要指定运算符的优先级，可以使用圆括号来告知编译器自定义运算符的优先级，示例代码如下所示。

```
c = a + b * c;           //先执行乘法
c = (a + b) * c;         //先执行加法
```

2. 左结合和右结合

所有的二元运算符都是右两个操作数，除了赋值运算符以外其他的运算符都是左结合的，而赋值运算符是右结合，示例代码如下所示。

```
a + b + c;           //结合方式为(a+b)+c
a = b = c;           //结合方式为 a=(b=c)
```

2.5 使用条件语句

程序开发中，开发人员经常遇到选择性的问题，如用户是否注册。如果用户已经注册则允许用户登陆，否则就跳转到注册页面。这个时候，就需要在程序中使用条件语句。**if** 是最常用的条件语句，同时，**if** 还包括 **if**、**if else**、**if else if** 等语句用于执行复杂的条件选择。



2.5.1 if 语句的使用方法

if 语句用于判断条件并按照相应的条件执行不同的代码块，if 语句包括多种呈现形式，这些形式分别是 if、if else、if else if。

1. 声明 if 语句

if 语句的语法如下所示。

```
if(布尔值) 程序语句
```

当布尔值为 true，则会执行程序语句，当布尔值为 false 时，程序会跳过执行的语句执行，示例代码如下所示。

```
if (true) //使用 if 语句
{
    console.WriteLine("ture"); //为 true 的代码块
}
```

上述代码首先会判断 if 语句的条件，因为 if 语句的条件为 true，所以 if 语句会执行大括号内的代码，程序运行会输出字符串 true，如果将 if 内的条件改为 false，那么程序将不会执行大括号内的代码，从而不会输出字符串 true。

2. 声明 if else 语句

if else 语句的语法如下所示。

```
if(布尔值) 程序语句 1 else 程序语句 2
```

同样，当布尔值为 true，则程序执行程序语句 1，但当布尔值为 false 时，程序则执行程序语句 2，示例代码如下所示。

```
if (true) //使用 if 语句判断条件
{
    console.WriteLine("ture"); //当条件为真时执行语句
}
else //如果条件不成立则执行
{
    console.WriteLine("false"); //当条件为假时执行语句
}
```

上述代码中 if 语句的条件为 true，所以 if 语句会执行第一个大括号中间的代码，而如果将 true 改为 false，则 if 语句会执行第二个大括号中的代码。

3. 声明 if else if 语句

当需要进行多个条件判断是，可以编写 if else if 语句执行更多条件操作，示例代码如下所示。

```
if (month == "12") //判断 month 是否等于 12
{
    console.WriteLine("winter"); //输出 winter
}
else if (month == "7") //判断 month 是否等于 7
{
    console.WriteLine("summer"); //输出 summer
}
else if (month == "3") //判断 month 是否等于 3
{
    console.WriteLine("spring"); //输出 spring
}
else //当都不成立时执行
{
    console.WriteLine("we don't have this month"); //输出默认情况
}
```

上述代码会判断相应的月份，如果月份等于 12，就会执行相应的大括号中的代码，否则会继续进行判断，如果判断该月份即不是 3 月也不是 7 月，说明所有的条件都不复合，则会执行最后一段大括号中的代

码。

4. 使用布尔值和布尔表达式

在 **if** 语句编写中，**if** 语句的条件可以使用布尔值或布尔表达式，以便能够显式的进行判断，示例代码如下所示。

```
if (true) //使用布尔值
{
    console.WriteLine("ture"); //输出 ture
}
```

**if** 语句的条件不仅能够由单个布尔值或表达式组成，同样也可以由多个表达式组成，示例代码如下所示。

```
if (true||false) //使用多个布尔值
{
    console.WriteLine("ture"); //输出 ture
}
```

同样在编写 **if** 语句的条件时，可以使用复杂的布尔表达式进行条件约束，示例代码如下所示。

```
if ((a==b)&&(b==c)) //使用复杂的布尔表达式
{
    console.WriteLine("a 和 b 相等,b 和 c 也相等"); //输出相等信息
}
```

5. 使用三元运算符

三元运算符(**?:**)是 **if else** 语句的缩略形式，比较后并返回布尔值，熟练使用该语句可以让代码变得更简练。

2.5.2 switch 选择语句的使用

**switch** 语句根据某个传递的参数的值来选择执行的代码。在 **if** 语句中，**if** 语句只能测试单个条件，如果需要测试多个条件，则需要书写冗长的代码。而 **switch** 语句能有效的避免冗长的代码并能测试多个条件。

1. 声明 **switch** 选择语句

**Switch** 语句的语法如下所示。

```
switch (参数的值)
{
    case 参数的对应值 1: 操作 1; break;
    case 参数的对应值 2: 操作 2; break;
    case 参数的对应值 3: 操作 3; break;
}
```

从上述语法格式中可以看出 **switch** 的语法格式。在 **switch** 表达式之后跟一连串 **case** 标记相应的 **switch** 块。当参数的值为某个 **case** 对应的值的时候，**switch** 语句就会执行对应的 **case** 的值后的操作，并以 **break** 结尾跳出 **switch** 语句。若没有对应的参数时，可以定义 **default** 条件，执行默认代码，示例代码如下所示。

```
int x;
switch (x) //switch 语句
{
    case 0: Console.WriteLine("this is 0"); break; //x=0 时执行
    case 1: Console.WriteLine("this is 1"); break; //x=1 时执行
    case 2: Console.WriteLine("this is 2"); break; //x=2 时执行
    default:Console.WriteLine("这是默认情况");break;
}
```

在上述代码中，当 **x** 等于 **0** 的时候，就会执行 **case 0** 的操作，就执行了 **Console.WriteLine("this is 0")**。如果 **x** 等于 **1**，语句就会执行 **case 1** 的操作。**switch** 不仅能够通过数字进行判断，还能够通过字符进行判断。

2. 使用 **break** 跳出语句

从上述代码中可以看出，每一个操作后面都使用了一个 **break** 语句。在 C/C++ 中，程序员可以被允许不写 **break** 而贯穿整个 **switch** 语句，但是在 C# 中不以 **break** 结尾是错误的，并且编译器不会通过。因为 C# 的 **switch** 语句不支持贯穿操作，因为 C# 中是希望避免在应用程序的开发中出现这样的错误。

注意：在 C# 中，可以使用 **goto** 语句模拟，继续执行下一个 **case** 或 **default**。尽管在程序中可以这样做，但是会降低代码的可读性，所以不推荐使用 **goto** 语句。

### 3. switch 的执行顺序

**switch** 语句能够对相应的条件进行判断，示例代码如下所示。

```
int x;
switch (x)                                     //switch 语句
{
    case 0: Console.WriteLine("this is 0"); break;           //x=0 时执行
    case 1: Console.WriteLine("this is 1"); break;           //x=1 时执行
    case 2: Console.WriteLine("this is 2"); break;           //x=2 时执行
    default: Console.WriteLine("no one"); break;              //都不满足时执行
}
```

从上述代码中分析，整型变量 **x** 被声明，并初始化，执行 **switch** 语句，当 **x** 的值为 **0** 的时候，**case 0** 之后的语句就会执行，即会执行 **Console.WriteLine("this is 0");** 执行完毕后，语句执行 **break** 跳出 **switch** 语句。当 **x** 的值等于 **3** 的，**switch** 操作会发现在 **case** 中没有与之相等的标记，则会执行 **default** 标记并执行默认的代码块。

注意：在 **switch** 语句中，**default** 语句并不是必须的，但是编写 **default** 是可以为条件设置默认语句。

### 4. 使用枚举类型

在 **switch** 表达式中，参数的类型必须为整型、字符型、字符串型、枚举类型或是能够隐式转换为以上类型的数据类型。在 **switch** 中常常会用到枚举类型，示例代码如下所示。

```
enum season { spring, summer, autumn, winter }           //声明枚举类型
static void Main(string[] args)
{
    season mySeason=season.summer;                         //初始化
    switch (mySeason)
    {
        case season.spring: Console.WriteLine("this is spring"); break; //mySeason=spring 时
        case season.summer: Console.WriteLine("this is summer"); break; //mySeason=summer 时
        case season.autumn: Console.WriteLine("this is autumn"); break; //mySeason=autumn 时
        case season.winter: Console.WriteLine("this is winter"); break; //mySeason=winter 时
        default: Console.WriteLine("no one"); break;
    }
    Console.ReadKey();                                     //等待用户按键
}
```

运行代码如图 2-20 所示。

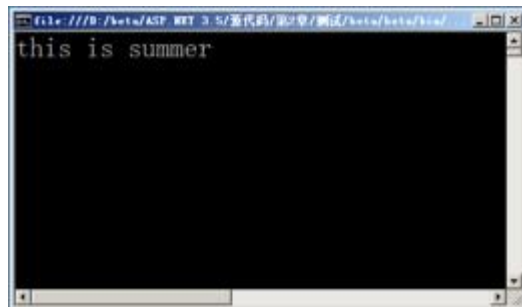


图 2-20 switch 中使用枚举类型

### 5. 组合 case 语句

在某些情况下，一些条件所达成的效果是相同，这就要求在 **switch** 中往往需要对多个标记使用同一语句。**switch** 语句能够实现多个标记使用同一语句，代码如下所示。

```
switch (mySeason)
{
    case season.spring:
    case season.summer: Console.WriteLine("this is spring and summer"); break; //组合 case
    case season.autumn:
    case season.winter: Console.WriteLine("this is autumn and winter"); break; //组合其他条件
    default: Console.WriteLine("no one"); break; //默认 case
}
```

## 2.6 使用循环语句

程序开发中，经常需要对某个代码块执行循环，使编译器能够重复执行某个代码块来完成计算。循环能够减少代码量，避免重复输入相同的代码行，也能够提高应用程序的可读性。常见的循环语句有 **for**、**while**、**do**、**for each**。

### 2.6.1 for 循环语句

**for** 循环一般用于已知重复执行次数的循环，是程序开发中常用的循环条件之一，当 **for** 循环表达式中的条件为 **true** 时，就会一直循环代码块。因为循环的次数是在执行循环语句之前计算的，所以 **for** 循环又称作预测式循环。当表达式中的条件为 **false** 时，**for** 循环会结束循环并跳出。**for** 循环语法格式如下所示。

```
for(初始化表达式,条件表达式,迭代表达式)
    循环语句
```

**for** 循环的优点就是 **for** 循环的条件都位于同一位置，同样，循环的条件可以使用复杂的布尔表达式表示。**for** 循环表达式包含三个部分，即初始化表达式、条件表达式和迭代表达式。当 **for** 循环执行时，将按照以下顺序执行。

- ❑ 在 **for** 循环开始时，首先运行初始化表达式。
- ❑ 初始化表达式初始化后，则判断表达式条件。
- ❑ 若表达式条件成立，则执行循环语句。
- ❑ 循环语句执行完毕后，迭代表达式执行。
- ❑ 迭代表达式执行完毕后，再判断表达式条件并循环。

**for** 语句循环示意图 2-21 所示。



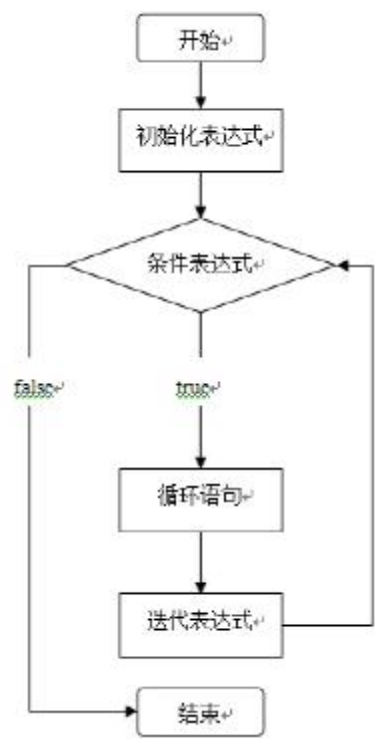


图 2-21 for 循环示意图

开发人员能够通过编写 **for** 循环语句进行代码块的重复，示例代码如下所示。

```
for (int i = 0; i < 100; i++) //循环 100 次
{
    Console.WriteLine(i); //输出 i 变量的值
}
```

技巧：**for** 循环即可做增量操作也可以做减量操作，如可以写为 `for(int i=10;i>0;i--)`，说明 **for** 循环的结构非常灵活，同样 **for** 循环的条件，迭代表达式也不仅仅局限与此。

**for** 循环还可以声明多个变量，在初始化表达式和迭代表达式中声明不只一个变量，示例代码如下所示。

```
for (int i = 0, j = 0; (i < 100) && (j < 100); i++, j++) //多个条件循环
{
    Console.WriteLine("i is" + i); //输出 i 变量的值
    Console.WriteLine("j is" + j); //输出 j 变量的值
}
```

2.6.2 while 循环语句

**while** 语句同 **for** 语句一样都可以执行循环，但是 **while** 的使用更加灵活，开发人员可以在代码块执行前判断条件，也可以在代码块执行一次后再行判断条件。

1. while 语句的语法

**while** 语句是除了 **if** 语句以外另一个常用语句，**while** 语句的使用方法基本上和 **if** 语句相同，其区别就在于，**if** 语句一般需要先知道循环次数，而 **while** 语句即便不知道循环次数也可以使用。**while** 语句基本语法如下所示。

```
while(布尔值)
    执行语句
```

**while** 语句包括两个部分，布尔值和执行语句，**while** 语句执行步骤一般如下所示。

- ❑ 判断布尔值。
- ❑ 若布尔值为 **true** 则执行语句，否则跳过。

**while** 语句循环示意图如图 2-22 所示。



图 2-22 while 语句循环示意图

while 语句示例代码如下所示。

```
x = 100; //声明整型变量
while (x != 1) //判断 x 不等于 1
{
    x--; //x 自减操作
}
```

上述代码，声明并初始化变量 **x** 等于 **100**，当判断条件 **x!=1** 成立时，则执行 **x—**操作，直到条件 **x!=1** 不成立时才跳过 **while** 循环。

2. continue 关键字：继续执行语句

在 **while** 语句中，可以使用 **continue** 语句来执行下一次迭代而不执行完所有的执行语句，示例代码如下所示。

```
int x, y; //声明整型变量
x = 10; //初始化 x
y = 10; //初始化 y
while (x != 1) //如果 x 不等于 1
{
    x--; //x 自减操作
    Console.WriteLine(x); //输出 x
    continue; //返回循环
    y--; //y 自减操作（但不执行）
    Console.WriteLine(y); //输出 y（但不执行）
}
```

上述代码声明了 **x**，**y** 两个整型变量，并初始化值为 **10**，当 **x** 不等于 **1** 时执行 **while** 循环。在 **while** 循环语句中，当执行到 **continue** 关键字时则跳出并继续执行 **while** 循环而不执行 **continue** 关键字后的语句，如 **y—**语句。

2. break 关键字：跳出循环语句

**break** 关键字允许程序在 **while** 循环中跳出并终止循环，从而能够继续执行循环后的语句，示例代码如下所示。

```
while (x != 1) //如果 x 不等于 1
{
    x--; //x 自减操作
    Console.WriteLine(x); //输出 x
    if (x == 5) //如果 x 等于 5
    {
        break; //跳出循环
    }
}
```

上述代码判断 **x** 是否等于 **5**。**x** 如果等于 **5**，则 **break** 语句会生效并跳出循环，继续执行 **while** 循环语句之后的代码。

## 2.6.3 do while 循环语句

**do while** 循环和 **while** 循环十分相似，区别在于 **do while** 循环会执行一次执行语句，然后再判断 **while** 中的条件。这种循环成为后测试循环，当程序需要执行一次语句再循环的时候，**do while** 语句是非常实用的。**do while** 语句语法格式如下所示。

```
do
{执行语句}
while(布尔值);
```

**do while** 语句包含两个部分，执行语句和布尔值。与 **while** 循环语句不同的时，执行步骤首先执行一次执行语句，具体步骤如下所示。

- ☐ 执行一次执行语句。
- ☐ 判断布尔值。
- ☐ 若布尔值为 **true**，则继续执行，否则跳出循环。

**while** 语句循环示意图如图 2-23 所示。

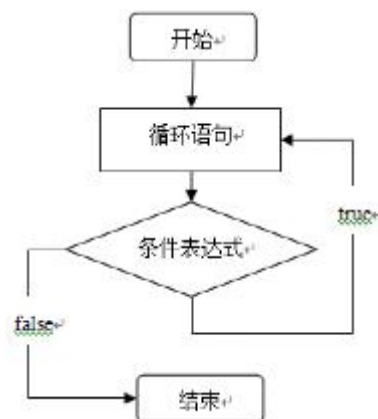


图 2-23 do while 语句循环示意图

**do while** 语句示例代码如下所示。

<code>int x=90;</code>	<code>//声明整型变量</code>
<code>do</code>	<code>//首先执行一次代码块</code>
<code>{</code>	
<code>    x ++;</code>	<code>//x 自增一次</code>
<code>    Console.WriteLine(x);</code>	<code>//输出 x 的值</code>
<code>}</code>	
<code>while (x &lt; 100);</code>	<code>//判断 x 是不是小于 100</code>

上述代码在运行时会执行一次大括号内的代码块，执行完毕后才进行相应的条件判断。

## 2.6.4 foreach 循环语句

**for** 循环语句常用的另一种用法就是对数组进行操作，**C#**还提供了 **foreach** 循环语句，如果想重复集合或者数组中的所有条目，使用 **foreach** 是很好的解决方案。**foreach** 语句语法格式如下

```
foreach (局部变量 in 集合)
    执行语句;
```

- ☐ **for each** 语句执行顺序如下所示。
- ☐ 集合中是否存在元素。
- ☐ 若存在，则用集合中的第一个元素初始化局部变量。
- ☐ 执行控制语句。
- ☐ 集合中是否还有剩余元素，若存在，则将剩余的第一个元素初始化局部变量。
- ☐ 若不存在，结束循环。

**foreach** 循环示意图如图 2-24 所示。

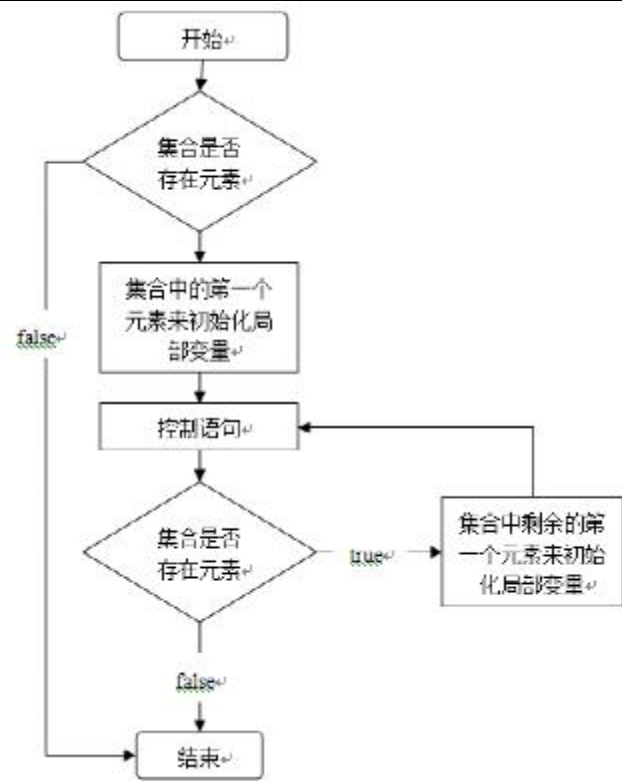


图 2-24 foreach 语句循环示意图

foreach 语句示例代码如下所示。

```
string[] str = { "hello", "world", "nice", "to", "meet", "you" }; //定义数组变量
foreach (string s in str) //如果存在元素则执行循环
{
    Console.WriteLine(s); //输出元素
}
```

上述代码声明了数组 **str**，并对 **str** 数组进行遍历循环。运行结果如图 2-25 所示。

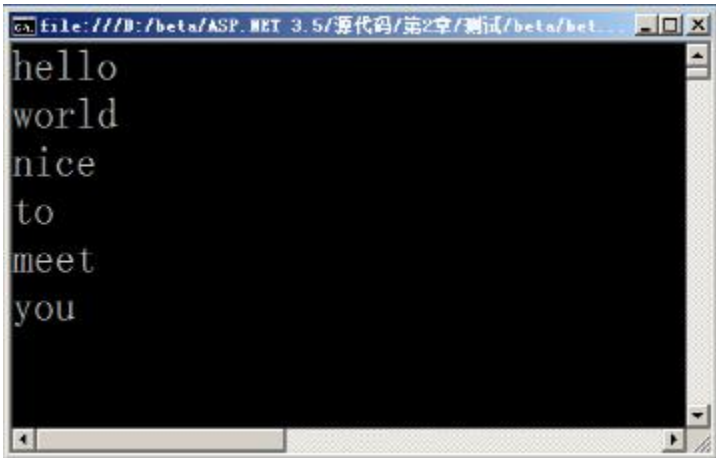


图 2-25 foreach 语句循环示例

注意：在使用 **foreach** 语句的时候，局部变量的数据类型应该与集合或数组的数据类型相同，否则编译器会报错。

## 2.7 异常处理语句

在传统的 **ASP** 开发过程中，要发现错误是非常复杂和困难的，常常错误发生后，很难找到错误的代码行。**C#**为处理程序执行期间可能出现的异常情况提供内置支持，这些异常由正常控制流之外的代码处理。常用的异常语句包括 **throw**，**try**，**catch** 等。



## 2.7.1 throw 异常语句

**throw** 语句用于发出在程序执行期间出现的异常情况的信号、引发异常的是一个对象，该对象的类是从 **System.Exception** 派生的。通常 **throw** 语句与 **try-catch** 或 **try-final** 语句一起使用。示例代码如下所示。

```
int x = 1; //声明整型变量 x
int y = 0; //声明整型变量 y
if (y == 0) //如果 y 等于 0
{
    throw new ArgumentException(); //抛出异常
}
Console.WriteLine("除数不能为 0"); //输出错误信息
```

上述代码使用 **throw** 语句引发异常并向用户输出了异常信息。

## 2.7.2 try-catch 异常语句

**try-catch** 语句由一个 **try** 和一个或多个 **catch** 子句构成，这些子句可以指定不同的异常处理应用程序。当 **try** 块中的代码异常，则会执行 **catch** 块的保护代码，在应用程序开发当中，**try-catch** 语句能够处理异常并返回给用户友好的错误提示，示例代码如下所示。

```
int x = 1; //声明整型变量 x
int y = 0; //声明整型变量 y
try //尝试处理代码块
{
    x = x / y; //出现异常
}
catch //捕获异常
{
    Console.WriteLine("除数不能为空"); //抛出异常
}
```

上述代码试图用一个整型变量除以一个值为 **0** 的整型变量，不使用 **try-catch** 捕捉异常，则系统会抛出异常跳转到开发环境或代码块。使用 **try-catch**，系统同样会抛出异常，但是开发人员能够通过程序捕捉异常并自定义输出异常。同样，它也可以接受从 **System.Exception** 派生的对象传递过来的参数，示例代码如下所示。

```
int x = 1; //声明整型变量
int y = 0; //声明整型变量
try //尝试处理代码块
{
    x = x / y; //进行除法计算
}
catch(Exception ee) //使用 Exception 对象
{
    Console.WriteLine("除数不能为空,具体错误信息如下所示\n"); //输出错误信息
    Console.WriteLine(ee.ToString()); //捕获代码
}
```

运行结果如图 2-26 所示。

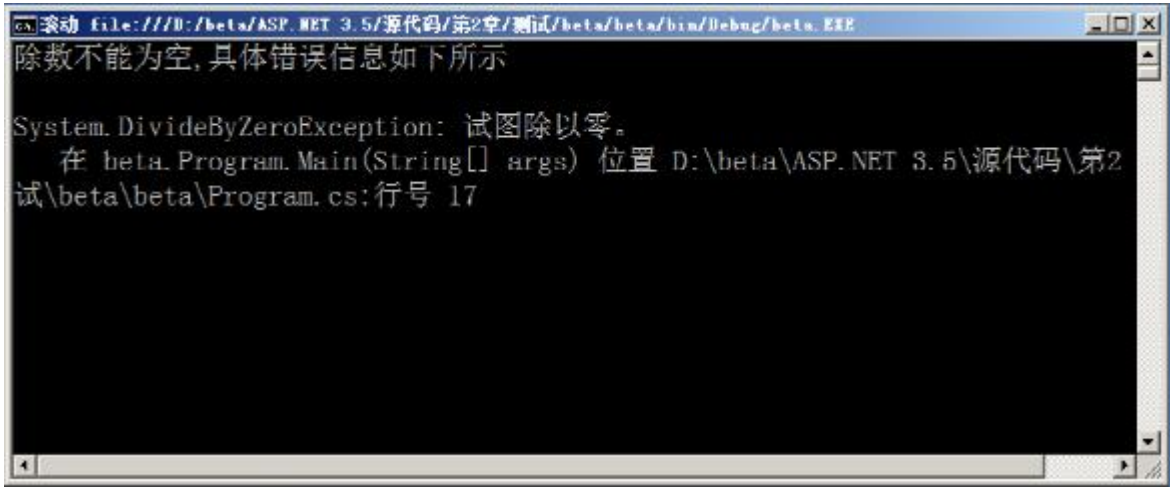


图 2-26 try-catch 语句使用示例

在运行结果中，程序详细的输出了异常的信息，此错误的信息由程序捕捉，并不会停止应用程序。

注意：try-catch 能够捕捉应用程序中的错误信息，但是 try-catch 会对程序的性能造成影响，在程序开发当中，应避免不必要的 try-catch 语句的出现。

2.7.3 try-finally 异常语句

catch 用于处理应用程序语句中出现的异常，而 finally 语句用于清除 try 块中分配的任何资源，以及运行应用程序中任何发生异常也必须执行的代码。finally 语句经常和 catch 语句搭配使用，示例代码如下所示。

```
int x = 1; //声明整型变量 x
int y = 0; //声明整型变量 y
try //尝试处理代码块
{
    x = x / y; //进行除法计算
}
finally //继续执行程序块
{
    Console.WriteLine("系统已自动停止"); //依旧输出错误信息
}
```

上述代码试图用一个整型变量除以一个值为0的整型变量，当异常发生时，系统会抛出异常，但是 finally 语句也被执行。

2.7.4 try-catch-finally 异常语句

try-catch-final 语句能够使应用程序更加健壮。try-finally 语句依旧会抛出异常，而 try-catch-finally 语句能够捕获异常并执行 finally 语句中的控制语句，try-catch-finally 语句结构和很灵活，示例代码如下所示。

```
int x = 1; //声明整型变量 x
int y = 0; //声明整型变量 y
try //尝试处理代码块
{
    x = x / y; //进行除法计算
}
catch (Exception ee) //捕获异常信息
{
    Console.WriteLine("除数不能为空,具体错误信息如下所示"); //抛出异常
    Console.WriteLine(ee.ToString()); //输出异常信息
}
```

finally	//继续执行程序块
{	
Console.WriteLine("系统已自动停止");	//继续执行程序
}	

上述代码试图用一个整型变量除以一个值为 **0** 的整型变量，当异常发生时，**catch** 捕获并抛出异常，捕获异常后，**finally** 语句也被执行。运行结果如图 2-27 所示。



图 2-27 try-catch-finally 语句运行示例

## 2.8 小结

本章介绍了 **C#**语言的基本知识，包括变量、变量规则、表达式、条件语句、循环语句以及异常处理，本章主要讲解了：

- ❑ 变量：介绍了变量的概念、变量的声明以及初始化。
- ❑ 变量规则：介绍了变量的命名、规则。
- ❑ 表达式：介绍了表达式的创建和使用方法。
- ❑ 条件语句：介绍了 **if**、**if else**、**if else if**、**switch** 等条件语句的使用方法。
- ❑ 循环语句：介绍了 **for**、**while**、**do while**、**foreach** 等循环语句的使用方法。
- ❑ 异常处理：介绍了异常以及 **throw**、**try-catch**、**try-finally**、**try-catch-finally** 语句的使用方法。

**C#**语言的特性参考了 **Java** 的技术规则，在 **C#**中也有一个虚拟机，叫公共语言运行环境(**CLR**)。**C#**的体系结构与 **Windows** 的体系结构十分相同，因此 **C#**很容易被开发人员使用并熟悉，能够很快的适应开发。**C#**和 **C++**也有很多的相似之处，学习 **C++**的开发人员也能够适应 **C#**的学习和开发。**C#**比 **C++**又有了更多的增强功能，如类型安全，事件处理，随便帐集，代码安全性，垃圾回收等。

**C#**是基于**.NET**体系的，也是**.NET**体系中的风云人物。学好 **C#**，读者不仅能够开发 **ASP.NET** 应用程序，也能够开发 **WinForm**、**WPF**、**WCF** 等应用程序。这些应用程序的开发原理上都是相通的，所以，掌握好 **C#**基础是非常必要的。

## 第 3 章 面向对象设计基础

第二章介绍了 **C#** 的基本语法，以及使用方法。**C#** 同 **Java**、**C++** 一样是面向对象的编程语言，同时 **C#** 更强化了面向对象的概念。本章将介绍面向对象的基础知识并介绍使用 **C#** 编写面向对象的应用程序，在 **C#** 中，面向对象的开发能够给系统设计、编码、维护提供更多的便利。

### 3.1 什么是面向对象

面向对象是应用程序开发中一个非常重要的技巧和概念，面向对象并不是什么高深的技术也不是负责的学习体系，面向对象主要是一种设计的思路。使用面向对象进行应用程序开发能够非常好的将现实中的物体进行抽象，这样就在一定程度上丰富了应用程序的结构，不仅如此，面向对象还包括继承、多态等特性以便能够快速构架应用程序。

#### 3.1.1 传统的面向过程

在传统应用程序开发领域（如 **C** 语言的开发），或者是早期的基于 **B/S** 领域的 **Web** 应用程序开发（如 **ASP**）都使用的是传统的面向过程的开发语言。而 **C++/JAVA/C#** 等开发都是使用的面向对象的开发语言。面向对象的开发语言更接近人类理解自然的语言，对开发人员来说更加通俗易懂，同时也对“对象”进行了较好的抽象。面向过程的一段 **C** 语言代码如下所示。

```
main()
{
    sum(x,y);
    get(x,y);
    print(x);
    print(y);
}
```

上述代码截取了 **C** 语言中的一个代码段，从上述代码中可以看出，**C** 语言中基本没有对象的概念。当执行一个主方法 **Main** 时，按照程序逻辑调用不同的函数，来达到运算的目的。传统的面向过程的开发必须规定每一个步骤，或者明确每一种函数，并在程序运行中调用不同的函数来实现运算的目的。面向过程的思想决定了其没有派生、覆盖、继承等特性，所以每当创建一个新的“对象”时，就有可能需要编写更多的代码，这在一定程度上造成了代码的编写和维护的困难。

#### 3.1.2 面向对象的概念

在面向过程的开发当中，开发人员发现在调用函数的时候，很难分清楚函数本身是属于哪个文件的，在代码的阅读上面，不同的开发人员会发现很难读懂其他的开发人员的代码。虽然注释和良好的命名都是必要的，但是还是给开发人员之间的交互造成了极大的困扰。为了解决这个问题，于是有了面向对象的概念。面向对象的一段 **C#** 代码如下所示。

class Program	//主程序类
{	
public int sum(int x, int y)	//sum 方法



```

    {
        return x + y;                                //返回值
    }
    static void Main(string[] args)                  //静态方法
    {
        int x = 1, y = 2;                            //声明整型变量
        Program sum = new Program();                 //创建对象
        Console.WriteLine(sum.sum(x, y));            //输出方法返回值
    }
}
```

上述代码中，**sum** 是一个 **Program** 对象，**sum** 有一个方法 **sum** 进行加法运算。读者可能会疑惑，相对于面向过程，面向对象的代码量好像变多了，而且执行的方法过程并没有太大的区别。其实面向对象解决了代码难以划分结构、代码可读性不高的问题，让程序变得更加容易组织和阅读。例如在.NET 中的 **Convert.ToInt32** 静态方法，当阅读到此代码的时候，开发人员能够比较清楚的知道这个方法做了什么操。而在调用面向过程中的函数的時候，如果函数的名称是 **CTi32**，而又分不清该函数在上下文中起到的作用时，会比较难以理解这个方法究竟做了什么。

### 3.1.3 面向组件的概念

面向组件其实是面向对象的另一种加强。在面向对象中，常常需要引用命名空间或者引用头文件（如 C++）来说明一个函数所在的对象与另一个同样名称的函数所在的对象是不同的。在传统的应用程序开发过程中，当客户更改了若干需求，往往只需要修改一个很小的功能，就要改动很多的代码，因此就出现了软件危机。

于是人们使用了分层的概念，将代码封装在一个类，然后对类进行组织协调，通过编译器对类或类库进行编译，形成 **DLL** 组件。在应用程序中使用 **DLL**，从而提高了代码的重用性。分层的概念也是设计模式的开端。面向组件的概念在平时就已经被读者广泛使用了，例如.NET 中的某些类库，还有 **COM** 组件等。面向组件的概念对开发人员在设计的思想上要求更高，这些要求不仅仅局限于编码。

## 3.2 面向对象的 C#实现

**C#**是面向对象的编程语言。在面向对象开发当中，不可避免的要创建一个类，创建类后还需要创建该类的属性和方法来描述对象，然后再创建这个类的对象进行实例化。创建后的对象能够通过类中的属性和方法完成相应的操作。

### 3.2.1 定义

什么是对象？世间万物皆对象，在生活中，可能是一只猫、一只狗，或者是饼干、一张订单、银行卡等等都是对象。对象描述了一个物体的特征，抽象一个实际的物体或一个方法。

类定义了对对象的特征，对对象进行了描述。这些特征包括对象的属性、对象的方法、对象的权限，以及如何访问该对象。在生活中就有很多例子，例如人类属于世界上的动物，并属于哺乳动物，同样，猫、狗、鸟也属于哺乳动物。所以，哺乳动物能够描述人类、猫、狗、鸟的一些基本特性。但是，人类会说话；猫会爬树；鸟会飞，不同的动物之间，实现的功能又不尽相同。所以这些对象之间也是有区别的。

程序开发人员能够为哺乳动物定义基本的相同的特性，如重量、体色、有没有毛之类，同样也能定义哺乳动物是否能够行走和飞行或者进行其他的操作。当然，如果定义了人类能够飞行，这是非常不符合逻辑的，类的设计同时也是需要符合逻辑。

注意：虽然在类设计中，能够自行设计相应的类和方法，以及属性，例如人类能够飞行，但是这个类的设计是不复合逻辑的，也是没有必要的。所以在类设计中，只要尽量完整的描述与现实中相同的属性即可。

3.2.2 创建一个类和其方法

上一节中了解了什么是面向对象，初学者可能还是对什么是面向对象、什么是对象、为什么要使用面向对象等概念很模糊。这里可以通过创建一个类可以深入了解面向对象的概念。在 **Visual Studio 2008** 中，系统提供了类的创建向导，如图 3-1 所示。

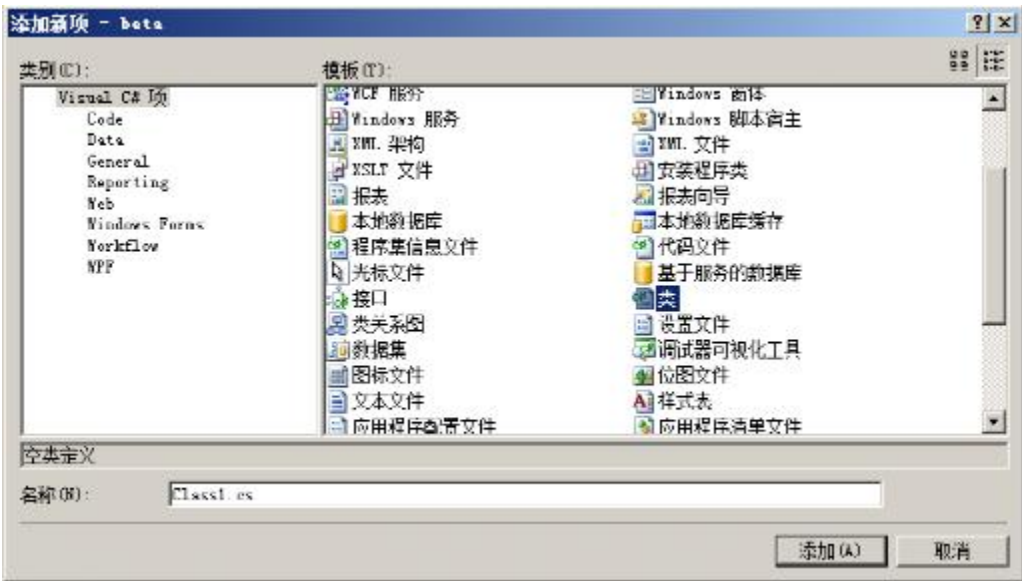


图 3-1 创建一个类

- 创建一个类的步骤如下所示。
- (1) 在 **Visual Studio 2008** 中打开一个项目。
  - (2) 右击该项目，在下拉菜单中选择【添加】选项，在【添加】下拉菜单中选择【新建项】选项。
  - (3) 在弹出对话框中选择【类】选项并，如图 3-1 所示，并给类一个名称。
  - (3) 单击【添加】按钮，类就添加到项目中了。

技巧：也可以在导航菜单栏中的【文件】菜单栏中选择【新建文件】创建一个类文件。

在创建了类文件之后，就能够编写代码创建类描述对象，为了能够描述哺乳动物，这里创建一个 **Animal** 类，示例代码如下所示。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace MyClass
{
    class Animal
    {
    }
}
```

//声明命名空间  
//不同的命名空间  
//声明命名空间  
  
//定义类

使用 **C#** 创建一个类，命名空间会包含在类文件中。默认命名空间通常与创建的项目名称相同，示例代码如下所示。

```
namespace MyClass
{
}
```

//当前程序的命名空间

类被创建之后，可以向类中添加方法、属性等，以便更加清晰的描述该对象，示例代码如下所示。

```
namespace MyClass
{
    public class Animal //创建对象
    {
        string color; //对象包含的字段
        public string get() //对象的方法
        {
            return color; //执行的方法
        }
    }
}
```

在主函数中，可以调用 **Animal** 类并创建该类的对象并执行相应的方法，这样就能够很好的描述一个对象并执行相应对象的动作，示例代码如下所示。

```
static void Main(string[] args)
{
    Animal an = new Animal(); //创建对象
    an.get(); //执行方法
}
```

上述代码首先创建了一个 **Animal** 对象 **an**，在创建对象后执行了对象的 **get** 方法进行该对象的颜色获取，从而得到了一个动物 **an** 的颜色。

3.2.3 类成员

在 **C#**中，类包含若干个组成成员，这些组成成员包括字段、属性、方法、事件等，这些组成成员能够彼此协调用于对象的深入描述。

1. 字段

“字段”是包含在类中的对象的值，字段使类可以封装数据，字段的存储可以满足类设计中所需要描述。例如上一节中 **Animal** 类中的字段 **color**，就是用来描述动物的颜色。当然，**Animal** 的特性不只颜色，可以声明多个字段描述 **Animal** 类的对象，示例代码如下所示。

```
class Animal
{
    public string color; //声明颜色字段
    public bool haveFeather; //声明是否含有羽毛字段
    public int age; //年龄字段
}
```

上述代码中，对 **Animal** 类声明了另外两个字段，用来描述是否有羽毛和年龄。当需要访问该类的字段的时候，需要声明对象，并使用点 “.” 操作符实现，**Visual Studio 2008** 中对 “.” 操作符有智能提示功能，示例代码如下所示。

```
Animal bird = new Animal(); //创建对象
bird.haveFeather = true; //鸟有羽毛
bird.color = "black"; //这是一只黑色的鸟
```

2. 属性

**C#**中，属性是类中可以像类的字段一样访问的方法。属性可以为字段提供保护，避免字段在用户创建的对象不知情的情况下被更改。属性机制非常灵活，提供了读取、编写或计算私有字段的值，可以像公共数据成员一样使用属性。

在 **C#**中，它们被称为“访问器”，为 **C#**应用程序中类的成员的访问提供安全性保障。当一个字段的权限为私有（**private**）时，不能通过对象的 “.” 操作来访问，但是可以通过“访问器”来访问，示例代码如下所示。

```
public class Animal
{
    private int _age; //定义私有变量
}
```

```
public int Age { get { return _age; } set { _age = value; } } //赋值属性
}
```

上述代码中为 **Animal** 类声明了一个属性 **Age**，在主程序中，同样可以通过 “.” 操作符来访问属性，示例代码如下所示。

```
Animal bird = new Animal(); //创建对象
bird.Age = 1; //Age 访问了 _age
```

在 **Visual Studio 2008** 中，属性的声明被简化，不再需要冗长的声明，示例代码如下所示。

```
public class Animal //创建类
{
    public int Age { get; set; } //简便的属性编写
}
```

**注意：**虽然在 **VS2008** 中，简化了代码，但是实现的过程依旧没有改变。

### 3. 方法

方法用来执行类的操作，方法是一段小的代码块。在 **C#**中，方法接收输入的数据参数，并通过参数执行函数体，返回所需的函数值，方法的语法如下所示。

```
私有级别 返回类型 方法名称(参数 1, 参数 2)
{
    方法代码块。
}
```

方法在类中声明。对方法的声明，需要指定访问级别、返回值、方法名称以及任何必要的参数。参数在方法名称后的括号中，多个参数用逗号分割，空括号表示无参数，示例代码如下所示。

```
public string output() //一个无参数传递的方法
{
    return "没有任何参数"; //返回字符串值
}
public string out_put(string output) //一个有参数传递的方法
{
    return output; //返回参数的值
}
```

上述代码中，创建了两个方法，一个是无参数传递方法 **output** 和一个参数传递的方法 **out\_put**，在主函数中可以调用该方法，调用代码如下所示。

```
Animal bird = new Animal(); //创建对象
bird.out_put(); //使用无参数的方法
string str = "我是一只鸟"; //创建字符串用于参数传递
bird.out_put(str); //使用有参数的方法
```

如上述代码所示，主函数调用了一个方法 **out\_put**，并传递了参数“我是一只鸟”。在使用类中的方法前，将“我是一只鸟”赋值给变量 **str**，传递给 **out\_put** 函数。在上述代码中，“我是一只鸟”或者 **str** 都可以作为参数。

在应用程序开发中，方法和方法之间也可以互相传递参数，一个方法可以作为另一个方法的参数，方法的参数还可以作为另一个方法的返回值，示例代码如下所示。

```
public string output() //一个无参数传递的方法
{
    return "没有任何参数"; //返回字符串
}
public string out_put() //使用其他方法返回值的方法
{
    string str = output(); //使用另一个方法的返回值
    return str; //返回方法的返回值
}
```

如上述代码所示，**out\_put** 使用了 **output** 方法，**output** 返回一个字符串“没有任何参数”。在 **out\_put** 方法中，使用了 **output** 方法，并将 **output** 方法的返回值赋给 **str** 局部变量，并返回局部变量。在方法的编写中，方法和方法之间可以使用同一个变量而互不影响，因为方法内部的变量是局部变量，示例代码如下所



示。

```
public string output() //一个无参数传递的方法
{
    string str = "没有任何参数"; //声明局部变量 str
    return str; //使用局部变量 str
}
public string out_put() //一个无参数传递的方法
{
    string str = "还是没有任何参数"; //声明局部变量 str
    return str; //使用局部变量 str
}
```

如上述代码所示，**output**和**out\_put**方法都没有任何参数，但是却使用了同一个变量**str**。**str**是局部变量，**str**的作用范围都在该变量声明的方法内，称作“作用域”。创建了一个方法，就必须指定该方法是否有返回值。如果有返回值，则必须指定返回值的类型，示例代码如下所示。

```
public int sum(int number1, int number2) //必须返回 int 类型的值
{
    return number1 + number2; //返回一个 int 类型的值
}
public void newsum(int number1, int number2) //void 表示无返回值
{
    int sum = number1 + number2; //没有返回值则不能返回值
}
```

上述代码中，声明了两个方法，分别为**sum**和**newsum**。**sum**方法中，声明了该方法是共有的返回值为**int**的方法，而**newsum**方法声明了是共有的无返回值方法。

4. 事件

事件是一个对象向其他对象提供有关事件发生的通知的一种方式。在**C#**中，事件是使用委托来定义和触发的。类或对象可以通过事件向其他类或对象通知发生的相关事情。发送或引发事件的类称为“发行者”，接收或处理事件的类称为“订阅者”。例如在**Web Form**中双击按钮的过程，就是一个事件，控件并不对过程做描述，只是负责通知一个事件是否发生。事件具有以下特点：

- ❑ 事件通常使用委托事件处理程序进行声明。
- ❑ 事件始终通知对象消息并指示需要执行某种操作的一种方式。
- ❑ 发行者确定何时引发事件，订阅者确定执行何种操作来响应该事件。
- ❑ 一个事件可以有多个订阅者。一个订阅者可处理来自多个发行者的多个事件。
- ❑ 没有订阅者的事件永远不会被调用。
- ❑ 事件通常用于通知用户操作，例如，图形用户界面中的按钮单击或菜单选择操作。
- ❑ 如果一个事件有多个订阅者，当引发该事件时，会同步调用多个事件处理程序，也可以使用异步处理多个事件。

声明委托和事件的示例代码如下所示。

```
public delegate void AnimalEventHandler(); //声明委托
public class Animal //创建类
{
    public event AnimalEventHandler OnFly; //声明事件
    public void Fly() //创建类的方法
    {
        OnFly(); //使用事件
    }
}
```

上述代码定义了一个委托，并针对相关委托声明了一个方法。关于委托和事件，会在后面的章节中讲到，上述代码在主函数调用代码如下所示。

```
Animal bird = new Animal(); //创建对象
bird.OnFly += new AnimalEventHandler(TestAnimal); //注册事件
bird.Fly(); //使用方法
```

3.2.4 构造函数和析构函数

构造函数和析构函数是面向对象中一个非常特别的函数，构造函数在对象初始化时被执行而析构函数在对象被销毁时被执行。开发人员无需显式的进行函数的编写，在 C#应用程序开发中能够为开发人员提供默认的构造函数和析构函数。

1. 构造函数

在变量中，常常需要初始化变量，而在类的声明中，也需要构造和初始化类。在类中，构造函数是在第一次创建对象时调用的方法。在任何时候，只要创建了一个对象，编译器就会默认的调用构造函数并执行构造函数。构造函数与类名相同，并且一个类可以有一个或多个构造函数，示例代码如下所示。

```
public class Animal //创建一个类
{
    public string AnimalName; //创建 AnimalName 名称字段
    public Animal() //使用构造函数
    {
        AnimalName = "动物"; //赋值私有变量
    }
}
```

上述代码声明了一个 **Animal** 类，并使用构造函数，构造函数与类同名。当声明一个对象时，系统默认使用此构造函数进行对象的构造。另外，构造函数也可以传递参数，示例代码如下所示。

```
public class Animal //创建一个类
{
    public string AnimalName; //创建 AnimalName 名称字段
    public Animal() //无参数的构造函数
    {
        AnimalName = "动物"; //赋值私有变量
    }
    public Animal(string name) //有参数的构造函数
    {
        AnimalName = name; //私有变量获取传递的参数
    }
}
```

构造函数可以传递参数，当声明一个对象时，若不指定构造函数，系统会默认使用无参数的构造函数。在初始化时，若指定构造函数，系统会按照指定的构造函数构造对象，示例代码如下所示。

```
Animal dog = new Animal("狗"); //通过构造函数创建对象
Console.WriteLine(dog.AnimalName); //输出对象的属性
```

在初始化中，直接将参数初始化并传递给构造函数则会在初始化中为对象中的字段初始化。构造函数方便了开发人员设置默认值、限制实例化来编写灵活并且便于阅读的代码。

2. 析构函数

析构函数是将当前对象从内存中移除时运行和调用的方法，析构函数的使用通常是为了确保需要释放的对象资源都得到了适当的处理。析构函数的函数名和类名基本相同，在方法前还需要“~”符号来声明该方法是类的析构函数。对于析构函数，有以下几个特点。

- ❑ 只能对类定义析构函数，结构不支持析构函数。
- ❑ 一个类只能有一个析构函数。
- ❑ 无法继承或重载析构函数。
- ❑ 无法调用析构函数，在对象注销时，系统会自动调用。
- ❑ 析构函数即没有修饰符也不能为它传递参数。

创建析构函数示例代码如下所示。

```
public class Animal //创建类
{
    public string AnimalName; //创建 AnimalName 名称字段
```

```
public Animal()                                //使用构造函数
{
    AnimalName = "动物";                       //赋值共有字段
}
~Animal()                                     //使用析构函数
{
    AnimalName = String.Empty;                 //将字符串清空
}
}
```

上述代码中，当 **Animal** 类的对象被销毁时，同时也将 **AnimalName** 设置为 **String.Empty**。构造函数隐式的对对象的基类调用 **Finalize**，所以开发人员无法控制在何时调用析构函数。在 **C++**中，当对象被销毁时，系统会调用析构函数并释放对象，而在 **C#**中，垃圾回收机制会自动清理对象资源。在确保了对象没有被任何应用程序使用后，**C#**的垃圾回收机制会自动清除不再使用的对象的资源。对于开发人员而言，虽然可以显式的使用 **Collect** 进行垃圾回收机制，但是会影响应用程序的性能。

3.3 对象的生命周期

在上一节中声明了类并说明了类成员，这些类成员包括字段、方法、事件、构造函数以及析构函数。类是对象的设计图（也称为模板），类用于描述对象。在创建对象后，对象就开始了其生命周期，只有在生命周期内的对象才能够被使用，否则无法使用相应的对象。

3.3.1 类成员的访问

类声明的方法是以 **class** 关键字开头，后面紧接着类名字，并以 “{” “}” 大括号包裹住类成员，示例代码如下所示。

```
访问权限 class 类名称
{
    类成员
}
```

例如在 **3.2.2** 中创建了一个 **Animal** 类，其中类名称就是 **Animal**。在实例化一个对象之后，在主程序或其他代码段中，需要对实例化的对象进行访问，即需要对类成员的访问。访问类成员的方法就是在对象后使用 “.” 号，并通过 **Visual Studio 2008** 智能提示选择相应的类成员，示例代码如下所示。

```
using System;
using System.Collections.Generic;
using System.Linq;                                //使用 LINQ 命名空间
using System.Text;                                //使用文本命名空间
namespace MyClass
{
    public class Animal                            //创建一个类
    {
        string type;                               //声明了类成员 string type
        public void SetType(string type)           //声明了类方法
        {
            this.type = type;                       //字段赋值
        }
    }
    class Program                                  //主程序类
    {
        static void Main(string[] args)            //程序入口方法
        {
            Animal bird = new Animal();             //创建了一个 bird 对象
        }
    }
}
```

```
bird.SetType("bird"); //引用了一个对象的成员
    }
}
}
```

上述代码中，创建了一个公共类 **Animal**，并创建了类成员字段 **type** 和方法 **SetType**。在主函数中，创建了一个 **Animal** 对象 **bird**，并通过 “.” 号访问了类成员 **SetType** 方法。在访问类或类成员时，可以通过关键字来限制类或类成员的访问权限，以便只有该类的派生类才能访问或者使用，同时也能够限制类成员的权限，提高类成员的安全性。这些关键字包括 **public**、**private**、**protected** 和 **internal**。

1. public 共有权限

**public** 字段具有最高访问级别，任何它的对象或者其他的类都能对 **public** 关键字所修饰的类或类成员进行访问，示例代码如下所示。

```
public class Animal //共有的类
{
    public string type; //共有的字段
    public void SetType(string type) //共有的方法
    {
        this.type = type; //赋值共有字段
    }
}
```

2. private 私有权限

**private** 字段具有最低的访问级别，它能够保证类和类成员的安全，却限制了其他类或对象对它的访问。私有成员只有在声明他们的类之后才能访问，示例代码如下所示。

```
public class Animal
{
    private int age; //私有成员
    string type; //默认的私有成员
    public void SetType(string type)
    {
        this.type = type; //赋值私有成员
    }
}
```

3. protected 保护权限

**protected** 字段具有保护类中字段的功能，能够保证类和类成员的安全性，也能够限制其他类或对象对它的访问。但是与 **private** 不同的是，**protected** 能够在类和类的的派生类中使用，比 **private** 具有更高的访问级别，又比 **public** 拥有更低的访问级别，保证了类的安全性，示例代码如下所示。

```
public class Animal
{
    protected string str; //受保护的成员
}
```

4. internal 程序集保护权限

**internal** 字段修饰的类或类成员只有在同一程序集的文件中内部类型或成员才可以访问，示例代码如下所示。

```
public class Animal
{
    internal string str; //受保护的程序集内的成员
}
```

这种程序集的文件中内部类型或成员才可以访问的修饰符通常是基于组件开发的，因为它能够使一组组件能够以私有方式进行合作，保证了组件的安全性。通常情况下，**ASP.NET** 中页面控件都是通过内部组件方式进行合作。另一方面，这些访问权限修饰符还能够组合使用，例如 **protected internal** 就可以进行组合使用，组合使用所修饰的对象只有该类和该类派生的类的成员才可以访问。



3.3.2 类的类型

每一个类的对象都是独立的对象，对象与对象之间有共同的属性，但是对象与对象之间不存在联系，虽然很多情况下类也可以引用类，示例代码如下所示。

```
public class Animal //创建类
{
    public string type; //创建字符串型共有变量
}
class Program //主程序类
{
    static void Main(string[] args) //程序入口方法
    {
        Animal bird = new Animal(); //bird 对象
        bird.type = "bird"; //初始化字段
        Animal cat = new Animal(); //cat 对象
        cat.type = "cat"; //初始化字段
    }
}
```

上述代码创建了两个对象，一个对象为 **bird**，另一个为 **cat** 对象。在初始化类成员时，为不同的对象的类成员赋了不同值，虽然这些类成员的名称相同，但是“.”号说明了该成员所在的对象是不同的。另外，由于类是引用类型，所以类的对象之间可以互相赋值，示例代码如下所示。

```
Animal newbird = new Animal(); //创建对象
newbird = bird; //对象之间互相赋值
```

上述代码将对象 **newbird** 初始化后并通过 **bird** 赋值，所以对象 **newbird** 中的 **type** 的值等于对象 **bird** 中的 **type** 值，因为 **newbird** 和 **bird** 都是引用的同一个对象。

3.3.3 .NET 的垃圾回收机制

当创建一个对象，**.NET** 对该对象初始化并在内存相应位置存储，当一个对象执行析构函数时，该对象被销毁并释放相关资源。在 **C++** 中，使用析构函数能够让开发人员显式的释放资源。而在 **.NET** 中，由于使用了垃圾回收机制（**GC**）从而导致开发人员无法控制析构函数是何时被运行的。

垃圾回收机制监视对象的生存周期，当一个对象没有被任何应用程序引用时，垃圾回收器就释放对象所占的内存以及资源。在基于 **.NET Framework** 编程时，开发人员无需像 **C++** 中显式的释放对象的资源也无需关心对象所占用的内存，因为 **.NET Framework** 的垃圾回收器能够监视对象并在相应的时候释放对象的资源。

垃圾回收机器没有固定的工作模式。它的工作间隔是不可预期的，一般情况下，当应用程序占用的内存不足的时候会启用垃圾回收器释放未被引用的对象的资源。在应用程序使用复杂并昂贵的外部资源的时候，**.NET** 机制提供接口能够让开发人员实现垃圾回收，以及资源释放机制，通过实现来自 **IDisposable** 接口的 **Dispost** 方法可以完成显式的资源释放。

在 **ASP.NET** 或者 **Win Form** 开发中，显式的使用 **Dispost** 方法能够提高应用程序的性能。同样，析构函数也是一种清理资源的方法，在对象析构时，可以用 **Dispose** 对对象的资源，以及连接信息进行清空从而对对象进行释放。

3.4 使用命名空间

在应用程序开发过程中，类和类成员的名称是丰富的，为了描述一个具体的对象，需要对类成员进行设计。在设计类和类成员过程中，不可避免类成员中的方法或者类的名称会出现相同的情况，这样就会

使类的使用变得复杂，代码的混乱造成可读性降低，使用命名空间可以解决此类难题。

### 3.4.1 为什么要用命名空间

正如引言中所述，在设计类和类成员的过程中，不可避免的，类或类成员使用的名称都是相同的，这样就让开发更加复杂，代码可读性降低。使用命名空间能够解决此类问题，示例代码如下所示。

```
namespace Programmer1                                //程序员 1 的命名空间
{
    public class Animal                                // Programmer1 的 Animal 类
    {
        public string type;                            //声明字段
    }
}
namespace Programmer2                                //程序员 2 的命名空间
{
    public class Animal                                // Programmer1 的 Animal 类
    {
        public string type;                            //声明字段
    }
}
```

上述代码中，创建了同样的两个类 **Animal** 以及两个类成员 **type**。在主函数中，开发人员很难区分到底是使用哪一个类进行对象的创建和初始化，因为通常情况下，每个程序员可能只负责该程序员的模块或者代码，当整合的时候，代码就会变得难以调用或难以维护。

正如某个学校有两个班级，每个班级里都有一个叫小明的人。如果在早操大会上点名小明，那么这两个小明都不知道点的是谁，如果指定一下，说是一班的小明，那么二班的小明就不会认为是自己了。命名空间就起到了这个区分的作用，在主函数中，可以显式的对类进行访问，示例代码如下所示。

```
namespace Programmer1                                //程序员 1 的命名空间
{
    public class Animal                                // Programmer1 的 Animal 类
    {
        public string type;                            //声明字段
    }
}
namespace Programmer2                                //程序员 2 的命名空间
{
    public class Animal                                // Programmer2 的 Animal 类
    {
        public string type;                            //声明字段
    }
}
namespace MyClass                                    //主程序的命名空间
{
    class Program                                        //主程序类
    {
        static void Main(string[] args)                //主程序入口方法
        {
            Programmer1.Animal bird = new Programmer1.Animal();//说明是程序员 1 的命名空间下的 Animal
            bird.type = "bird";                          //初始化字段
        }
    }
}
```

上述代码中很好的解决了类名称相同的情况下开发和维护的困难。在执行代码 **Programmer1.Animal bird = new Programmer1.Animal()**时，编译器就能够知道 **Animal** 类是属于命名空间 **Programmer1** 的，也就不会和

命名空间 **Programmer2** 的 **Animal** 类混淆。

## 3.4.2 创建命名空间

程序开发中，创建和良好的使用命名空间，对开发和维护都是有利的，命名空间语法格式如下所示。

```
namespace 命名空间
{
    类以及类成员
}
```

**namespace** 声明了一个命名空间，名称取命名空间的名称，在由“{”“}”大括号内引用的类成员来创建类。同样也可以创建两层或多层命名空间，示例代码如下所示。

```
namespace Programmer1                                //单层命名空间
{
    public class Animal                                // Programmer1 的 Animal 类
    {
        public string type;                            //声明字段
    }
}
namespace Programmer2                                //双层命名空间
{
    namespace Programmer3                            // Programmer2 的命名空间
    {
        public class Animal                            // Programmer3 的 Animal 类
        {
            public string type;                        //声明字段
        }
    }
}
```

同样，命名空间成员也通过“.”号访问，例如访问双层命名空间的类的字段时，可以通过 **Programmer2.Programmer3.Animal.type** 进行访问。

## 3.4.3 分层设计中使用命名空间

从上一节中可以看出，命名空间的使用可以对相同名称的类进行较好的规范。但是，在同一层代码块中，似乎很少使用命名空间来规范。而在分层设计中，命名空间的使用是非常必要的，虽然初学者不需要详细的了解分层设计，但是分层设计在软件开发过程当中是非常必要的，使用 **Visual Studio 2008** 能够轻松的创建分层构架软件。

在解决方案资源管理器中选择当前解决方案，右击【解决方案】项目，在下拉菜单中选择【添加】选项，在【添加】的下拉菜单中选择【新建项目】选项。若无法在解决方案管理器中看见解决方案，则可以在菜单栏的【工具】选项中选择【选项】菜单并在弹出窗口中找到【项目和解决方案】窗口，在窗口中选中【总是显示解决方案】复选框即可配置解决方案管理器，如图 3-2 所示。

为了能够在应用程序中进行分层开发，需要创建类库用于类的规划，创建一个“类库”项目，如图 3-3 所示。

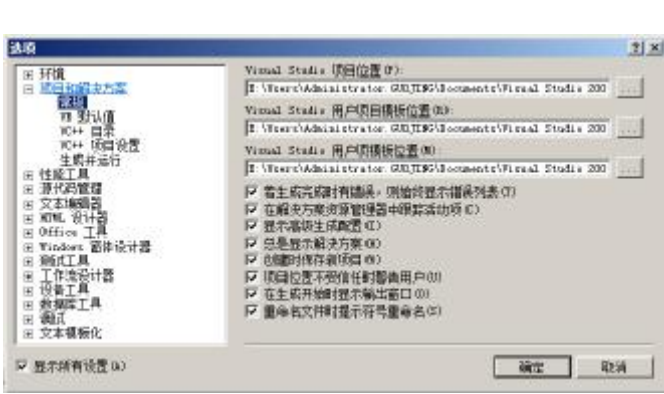


图 3-2 显示解决方案



图 3-3 创建类库

输入项目名称，创建了类库后，命名空间就是创建的项目名称，系统会自动创建一个类 **Class1**，读者可自行修改类名称，并创建字段和方法，示例代码如下所示。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace MyNamsSpace
{
    public class Class1
    {
        public string output()
        {
            return "另一个项目的命名空间";
        }
    }
}
```

//使用系统命名空间  
//当前程序命名空间  
  
//当前类名称  
  
//输出字符串方法  
  
//返回值

在原有的项目中，必须声明使用用户创建的命名空间，才可以访问命名空间类的类并创建对象。正如.NET 中为我们提供的系统命名空间一样，也是通过使用 **using** 来声明的。首先需要添加引用，右击项目，在下拉菜单中选择【添加引用】选项，在添加引用窗口选项卡上选择【项目】标签并引用相应类库，如图 3-4 所示。



图 3-4 添加引用

单击【确定】按钮后，在代码头部书写 **using** 命名空间则可以使用类库项目的命名空间并访问方法和成员，代码如下所示。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using MyNamsSpace;
namespace MyClass
{
    class Program
```

//系统命名空间的使用  
//自定义命名空间的使用



```
{
    static void Main(string[] args)
    {
        Class1 myclass = new Class1();           //创建对象，该类完整名称是 MyNamespace.MyClass
    }
}
```

上述代码中引用了 **MyNamespace** 命名空间，并访问了命名空间中的类，通过该类创建了一个对象。分层设计是软件设计中常用的设计方法，同设计模式一样，分层设计也是为了规范软件的开发和维护、降低软件开发成本、将软件模块化。但是过多的使用命名空间和分层设计也会造成层次过多无法维护的相反效果。

注意：当使用另一个命名空间的方法时，如果在本程序段中没有同名的方法，可以直接使用方法名称，而如果当前程序中包含了同名的方法，则需要指定命名空间名。

3.5 类的方法

创建了类，就需要创建类的字段，初始化字段。同样，创建了类之后也需要创建类的方法，来访问或者对字段进行操作。在类的对象的初始化后，对象能够使用方法进行对象的操作从而能够更加完整的描述一个对象（事务）。

3.5.1 编写方法

方法是指定名称的一组语句，每个方法都有一个方法名和一个方法体。方法名用来指定方法的名称，方法体用来描述该方法使用何种算法和结构来完成计算。对方法名的声明推荐具有一定的含义，例如 **GetUserPassword** 的大意就是获取用户密码，这样其他的开发人员也能够读懂该函数的作用，提高了编码效率。方法的声明语法如下所示。

```
描述 权限 返回值类型 方法名称(参数)
{
    方法体
}
```

上述伪代码说明了方法的声明语法，开发人员能够参照以上伪代码进行方法的编写，示例代码如下所示。

```
static public string GetUserName()           //返回值类型为 string
{
    string username = "guojing";             //定义字符串变量
    return username;                          //返回 string 类型
}
```

编写一个方法必须按照方法声明的语法来编写，在编写方法时需要指定描述、权限、返回值类型等，这些必要的条件作用如下所示。

- ❑ 描述：用来描述方法，例如是否是静态方法等。
- ❑ 权限：权限用来描述方法的访问性，确定外部对象是否能引用或直接访问此方法或类成员。
- ❑ 返回值类型：返回值类型用来描述方法体中所返回的值的类型。
- ❑ 方法名称：描述方法的命名称。
- ❑ 参数：参数用来向方法传递参数，可以为 0 个或多个参数，多个参数用逗号分割。
- ❑ 方法体：用来描述方法所要执行的操作。

如果创建了一个方法，并且说明了该方法必须有返回值，则该方法的必须有返回值并且返回值的类型

与声明的方法的返回值类型相同。若方法中的返回值和修饰符中返回值的修饰不相同，则编译器会报错，若方法无返回值，可使用 **void** 关键字修饰方法，示例代码如下所示。

```
static public void  GetUserName()                                //无返回值的方法
{
    string username = "guojing";                                //初始化字段
}
```

3.5.2 给方法传递参数

上一节中，了解了方法可以传递参数。通过参数的传递，开发人员能够知道方法的作用、方法的意义并通过传递参数对未知源代码的方法进行使用而无需阅读源代码。带参数传递的方法代码编写如下。

```
static public string  GetUserName(string name)                  //有返回值有参数的方法
{
    string username = name;                                     //初始化字段
    return username;                                           //返回一个返回值
}
```

上述代码创建了一个包含参数传递的方法。当传递了一个参数，方法接收了变量值的拷贝，然后使用拷贝的变量值进行操作。

1. 传递值

方法可以接收传递的值，并获取参数，示例代码如下所示。

```
class Program
{
    public void  GetUserName(string name)                        //包括参数传递的方法
    {
        string username = name;                                //内部变量赋值
    }
    static void Main(string[] args)
    {
        Program pro = new Program();                            //创建一个新对象
        pro.GetUserName("guojing");                             //传递了一个值
    }
}
```

上述代码创建了一个类，并在类中创建了一个 **GetUserName** 方法，包含一个参数传递。在创建了一个对象后，可以直接通过传递值来传递参数。

2. 传递对象

方法也可以接收对象，对象也可以看作是一个变量进行参数传递，示例代码如下所示。

```
class Program
{
    public string name;                                         //声明共有变量
    public void  GetUserName(Program pro)                       //声明共有方法
    {
        string username = pro.name;                             //初始化字段
    }
    static void Main(string[] args)
    {
        Program pro = new Program();
        pro.name="guojing"                                     //初始化字段
        pro.GetUserName(pro);                                   //传递了一个对象
    }
}
```

上述代码包含一个参数传递，该参数是一个对象。

3. this 关键字

**this** 关键字能够访问类成员，当参数名和类成员中字段名称相同时，可以使用 **this** 关键字，示例代码如下所示。

```
class Program
{
    public string name;                //声明共有变量
    public void  GetUserName(string name)
    {
        this.name = name;             //使用 this 关键字赋值私有变量
    }
    static void Main(string[] args)
    {
        Program pro = new Program();   //创建一个新对象
        pro.GetUserName("guojing");    //使用方法进行参数传递
    }
}
```

上述代码中，方法传递的参数名称为 **name**，同样类内有一个 **name** 为名称的字段，通过 **this** 关键字可以区分，**this.name** 就是类中的 **name** 字段。

### 3.5.3 通过引用来传递参数

通常情况下，方法只能返回一个值，但是在实际的开发当中，实际的情况可能需要返回多个值的方法。所以，传递一个变量的引用给一个方法即可实现。在平常使用的过程中，传递的参数在方法中是以参数的值的拷贝来运算的。当使用引用时，方法使用的是参数的实际数值，对参数的改变也会改变参数的实际值。

#### 1. 使用 **ref** 关键字

在描述一辆车的时候，可以创建一个 **Car** 类来描述车这个对象，示例代码如下所示。

```
class Car
{
    string carNumber="1001";
    string carType = "Car";
    public string GetCarNum(string num,string type)    //创建方法
    {
        num = carNumber;                               //获取私有变量的值
        type = carType;                                 //获取私有变量的值
        return num;
    }
}
class Program
{
    static void Main(string[] args)
    {
        Car myCar = new Car();                          //创建一个新对象
        string number="0";                               //初始化对象
        string type = String.Empty;                      //String.Empty 初始化空字符串
        myCar.GetCarNum(number, type);                   //改变字符串的值
        Console.WriteLine(number);                       //输出
        Console.WriteLine(type);
        Console.ReadKey();
    }
}
```

上述代码运行结果中，输出的 **number** 依旧是 **0**，而 **type** 依旧是空字符串。上述代码并没有为多个变量参数进行更改，所以需要使用 **ref** 关键字来对多个对象进行更改，示例代码如下所示。

```
class Car
{
```

```
string carNumber="1001";           //声明字符串变量
string carType = "Car";           //声明字符串变量
public string GetCarNum(ref string num,ref string type) //使用 ref 关键字
{
    num = carNumber;               //获取私有变量的值
    type = carType;                //获取私有变量的值
    return num;                    //返回字符串变量
}
}
class Program
{
    static void Main(string[] args) //程序的主入口方法
    {
        Car myCar = new Car();      //创建一个新对象
        string number="0";          //将 number 赋值为 0
        string type = String.Empty; //将 type 赋值为空
        myCar.GetCarNum(ref number,ref type); //使用 ref 关键字
        Console.WriteLine(number);  //输出字段
        Console.WriteLine(type);    //输出字段
        Console.ReadKey();          //等待用户按键
    }
}
```

上述代码使用了 **ref** 关键字，运行后，**number** 和 **type** 的值都被改变了。在声明的方法参数前，必须使用 **ref** 关键字，非常重要的一点是，在使用该方法时，同样需要使用 **ref** 关键字修饰参数。

技巧：在这里 **carNumber** 并不是一个 **int** 型变量，原因是虽然汽车编号大部分是以数字显示的，但是其实是字符串，一辆车的编号为 **1001** 并不代表这辆车是第 **1001** 辆车或其他。

2. 赋值明确

在 **C#**中，传递给方法的变量必须在传递前明确赋值，这样可以避免在执行方法时变量未赋值导致的不可预知的错误。因为有可能一个变量的值为 **null**，而 **null** 值不能转换到.NET 中的某些类型，例如上一小结的代码中，若不对 **number** 赋值为“0”，则编译器会报错。

3.5.4 方法的重载

在程序开发过程中，很多情况下，需要执行相同的函数体。例如在获取用户信息时，有的时候需要对用户的 **ID** 进行查询再获取用户信息，有的时候需要通过对用户的用户名来获取用户信息，可以用以下代码来实现。

```
public void GetUserInfor(string name) //一个方法
{
    //通过用户名获取用户信息
}
public void GetUserInforById(int id) //重载方法
{
    //通过 ID 获取用户信息
}
```

当一个功能需要加强的时候，就要写更多的方法。当这些方法的方法体大致上是相同的时，或者这些方法体都能够表示这个方法名的意义时，就可以使用重载来增强代码的可读性。只要传递的参数不同，就可以重载函数，示例代码如下所示。

```
public void GetUserInfor(string name) //原方法
{
    //通过用户名获取用户信息
}
```



```
}
public void GetUserInfor(int id)           //方法名相同但是参数不同
{
    //通过 ID 获取用户信息
}
```

上述代码创建了两个相同方法名的方法，但是传递的参数不同，编译器也可以通过编译。当调用方法时，指定具体的调用参数，则编译器会调用相应的方法，示例代码如下所示。

```
UserInformation user = new UserInformation();           //创建对象
user.GetUserInfor("guojing");                           //调用第一个方法
user.GetUserInfor(115);                                 //调用重载方法
```

当声明对象后，对象使用方法，根据传递的参数不同，编译器会自动识别，如传递的参数为字符串变量 **guojing** 时，则会使用 **GetUserInfor(sting name)**方法。同样，如传递的参数为整型变量 **115** 时，则会使用 **GetUserInfor(int di)**方法。在类设计中，可以将一些具有相同功能的方法设计为重载方法。使用重载的情况如下所示：

- ❑ 设计一些相同的方法时，如果只是参数不同，则使用重载。
- ❑ 如果为程序添加一个新功能，重载一个方法是不错的选择。
- ❑ 如要实现类中相似的功能，则可以考虑使用重载。

### 3.6 封装

在 **C#**中，封装就是将类成员中的字段、方法以及属性事件、委托等放在一个公共的结构中。按照一个公共的方法把数据和操作这些数据的方法进行组装（封装），同时为对象指定操作和属性，从而创建了新的数据类型提供给用户使用，而保证了私密的内容不会被用户察觉。

#### 3.6.1 为什么要封装

在应用程序开发，特别是面向组件开发的过程中，常常会将类成员中的字段、方法、属性及事件封装在一个类或命名空间内。当把数据和方法封装后，就可以指定方法和属性，对于外部使用者而言，他们无需知道类是怎样设计的，只需要关心如何实现信息，让用户成为类的使用对象。正如在.NET中的 **System.Web** 命名空间或者类型转换 **Convert.ToInt32** 一样，开发人员知道这个命名空间或者这个类的静态方法是做什么的，却不知道这个方法内部的代码。

这些方法的内部代码对开发人员是封闭的，保证内部代码的隐私和安全，开发人员只需要使用方法或者覆盖方法来达到自己编程的目的。例如一台电脑有显示器、主机等等外设，而相对于显示器而言，显示器是一个类，同样，机箱也是一个类，同样这些来还包括鼠标，键盘等。而显示器内部和机箱内部对一般的用户是不可见的，因为用户不知道怎样拆装显示器和机箱，但是用户知道怎样将插座相连，组成一个完整的计算机。封装能够让用户更加关注“计算机”本身，而不去关注“计算机”内部是怎样实现的。

这种抽象方便了封装人员便于改变内部实现细节，在改变了内部细节之后，用户通常情况下不会感到有任何的差异。

注意：例如.NET 中，在.NET 从 1.1 到 2.0 然后到 3.X，很多的细节被改变，因为类或方法有了改变。

但是大部分情况下，很多的细节改变了，而使用的方法没有改变，特别是从 2.0 到 3.5。

#### 3.6.2 类的设计

在设计一个类时，应该尽量的隐藏细节，只暴露那些开发人员或者类使用者需要知道的操作和数据。

这样就方便了代码的维护，实现了在使用者无需改变与类成员交互方式的前提下，对类的实现细节进行更改。

在 **C#** 中，没有对类成员或方法的公共属性进行设置，则默认的权限是 **private**（私有的）。而在类的设计中，尽量将使用的类成员设置为私有的权限，因为这样保证了代码的安全性，也让使用者更加关心怎样与类成员进行交互。而可以对一些共有的属性或者与方法交互的一些字段设置成为 **public**（共有的），除非认为字段是有用的，否则一般不予暴露。这种方式不仅保护了类成员，同样也让维护变得简单，示例代码如下所示。

```
class Car                                     //创建一个类
{
    public void GetCar()                     //创建类的方法
    {
        int i = 1;                          //声明整型变量
    }
}
class Program
{
    static void Main(string[] args)
    {
        Car myCar = new Car();              //使用类创建对象
        myCar.GetCar();                     //使用了对象的一个方法
    }
}
```

上述代码创建了一个汽车（**car**）类，在主方法中使用类，同样，当对 **Car** 类的类成员做修改时，不会影响到用户的使用，示例代码如下所示。

```
class Car                                     //创建一个类
{
    public void GetCar()                     //创建类的方法
    {
        int i = 1;                          //声明变量
        int j = 0;                          //更改了类内部的实现细节
    }
}
class Program
{
    static void Main(string[] args)
    {
        Car myCar = new Car();              //创建对象
        myCar.GetCar();                     //使用方法依旧没有改变
    }
}
```

同样，在对类进行了封装之后，用户不知道类是怎样实现的，但是同样能够创建类或对象，这如前面的章节中命名空间一样。通常情况下开发人员会建立一个类库，使用者需要引用类库，并调用类库中的代码，而无需知道类库中的代码是怎样实现的，示例代码如下所示。

```
class Program                                //应用程序主类
{
    static void Main(string[] args)         //应用程序入口方法
    {
        Car myCar = new Car();              //使用 Car 类，而使用者并不知道细节
        myCar.GetCar();                     //使用方法
    }
}
```

3.7 属性

在 3.2.3 类成员一节，简单的讲解了属性作为类成员的使用方法。属性是为了把对象的实现细节与使用者所能看见的元素隔离，同时通过定义类成员的作用域，从而控制关于对象数据的访问。虽然在前面的章节中，可以通过 **private**、**public** 等关键字限制类成员的访问权限，但是通过属性可以管理其他对象对类中类成员的访问。属性是一种类成员，提供了对对象或类中元素的访问方法。

3.7.1 语法

定义属性的语法通常使用 **public**、**private** 来修饰访问权限，同样包括类型、属性名、关键字以及由 “{” “}” 大括号包含的代码段，示例代码如下所示。

```
public string Str { get; set; } //编写属性
```

**get** 语句和 **set** 语句成为访问器。在 **Visual Studio 2005** 中，**get** 访问器的返回类型必须与属性类型相同，或者可以隐式的转换为属性类型。**set** 访问器等价于具有一个叫 **value** 的隐式参数的方法。而在 **Visual Studio 2008** 中，可以使用更加简单的代码，如上述代码所示。

1. 类成员实现

通过设计类成员，可以实现属性的原理，也能够更加方便的理解属性的实现，示例代码如下所示。

```
class Car //编写类
{
    private string _str; //私有的类成员，字段
    public string SetStr(string str) //共有的方法，设置类成员
    {
        _str = str; //设置相应属性
        return _str; //返回相应属性
    }
    public string GetStr(string str) //共有的方法，返回私有类成员值
    {
        return _str; //返回相应属性
    }
}
```

上述代码创建了一个汽车类，用来描述一辆汽车，同时也创建了一个私有的类成员 **\_str**，用来描述车的属性。当设置为私有时，对象不能通过 “.” 号来访问 **\_str**，于是需要通过类成员中的方法来实现访问。**SetStr** 是一个为 **\_str** 赋值的一个方法，代码非常简单，而 **GetStr** 是获取 **\_str** 值的方法，而这两个方法都公共方法，可以通过外部引用。

2. 属性的使用方法

从上述代码中可以看出，属性的使用无非就是为私有的变量的访问权限做了调整，通过属性来访问用户本不应该访问的私有变量。可以将上述类更改，使代码更加简洁和可读，示例代码如下所示。

```
class Car //编写类
{
    public string str { get; set; } //使用属性声明字符串变量
}
class Program //应用程序主类
{
    static void Main(string[] args) //应用程序入口方法
    {
        Car myCar = new Car(); //创建新变量
        myCar.str = "Car"; //访问类成员并赋值
    }
}
```

上述代码使用了属性，让代码更加可读。可以将属性的方法对用户隐藏，提高了代码的安全性，也增加了封装的特性。

注意：在 2.0 中，`get` 需要返回一个值，并且与 `get` 访问器的返回类型必须相同。而 `set` 访问器中，需要写变量名称的值等于 `value` 的隐式参数方法，但是在 3.5 中去掉了这样的声明方式，直接写 `get;set;` 即可实现。

3.7.2 只读/只写属性

只读属性和只写属性是 C# 属性中的一个特殊属性，只读属性和只写属性都只包含相应的访问器用于不同的属性的访问。只读属性和只写属性在一定程度上保证了属性的安全性和类的密封性，如果在应用程序开发中为了保证某个字段的安全性或可访问性，可以使用只读或只写属性。

1. 只读属性

在属性的声明中，允许声明只包含一个 `get` 访问器的属性。在这种情况下，声明的类成员只允许读，并在程序上下文中使用，示例代码如下所示。

```
class Car //编写类
{
    public string str { get { return "this is a str";}} //只读属性
}
```

上述代码中只使用了 `get` 访问器，在使用中，类成员变量 `str` 只能读而不能改。

2. 只写属性

同样，在属性的声明中，同样允许声明只包含一个 `set` 访问器的属性。在这种情况下，声明的类成员只允许写，但是却不能读，在程序的上下文中不允许使用，示例代码如下所示。

```
class Car //编写类
{
    private string _str; //声明私有变量
    public string str { set { _str = value; }} //只写属性
}
```

上述代码中只使用了 `set` 访问器，在使用中，类成员变量 `str` 只能写而不能读。

3.8 继承

在类的设计中，经常需要管理和开发相似功能。在设计这些类的时候，就可以使用继承的原则。在面向对象的应用程序开发中，允许创建一个抽象的类而不实现其具体方法，而需要通过继承、派生来实现方法。这样不仅优化了代码，提高了代码的可读性，而且在开发过程中，也让开发人员有比较明确的编码思想，使开发人员与开发人员之间更加容易协调。

3.8.1 继承的基本概念

在应用程序开发过程中，需要完成功能相近但是实现不同的类来抽象对象的时候，就需要用到继承。例如，要开发一个应用程序或者网站来统计全球动物的种类和基本信息的时候，就需要创建一些类，比如人类、猫、狗等等。相比之下，人类和猫和狗有相似之处，如它们都是哺乳动物，说的更高一点，就都是生物。而人，猫和狗却是不同的对象，还必须要区分三者之间的关系，错误的类设计代码如下所示。

```
public class Animal //编写类
{
```



```
public string type;           //动物的种类
public string color;         //动物的颜色
public string sound;         //动物叫的声音
}
class Program                 //应用程序主类
{
    static void Main(string[] args) //应用程序入口方法
    {
        Animal cat = new Animal(); //创建对象
        cat.type = "cat";          //猫科动物
        cat.sound = "miaomiao";    //猫叫声为 miaomiao
        Animal person = new Animal(); //创建对象
        person.type = "person";    //人类
    }
}
```

上述代码中，并没有语法错误，但是有逻辑误差或者说是开发困难。因为，如果要形容人类的国家，那么就得在 **Animal** 类中增加一个 **country**，但是猫却没有国家之分，这样就让整个类变得非常的混乱且庞大臃肿，使用派生就能解决这样的问题。派生类的优点如下所示。

- ❑ 提高重用性：派生提高了代码的重用性，不至于在创建一个新对象时再重新写一个新的类。
- ❑ 提高结构性：派生让程序有了结构，在程序开发过程，每一个派生类均继承上一个类的方法，且每个派生类除了可以使用公共的字段以外，可以专门为派生类增加字段和方法而不去影响到其他的派生类。

上述代码中可以看出，在设计类的时候，对于同一个类的重复使用，可能会使用户非常的迷惑。例如猫是派生自动物的，但是用户会希望猫的类型是 **Cat**，而同样，当创建一个人的类的时候，通常情况下用 **Animal** 描述是不太适合的，虽然人和猫都是属于动物，但是这样的表述往往让人迷惑不解，而派生类可以更好的实现。

3.8.2 创建派生类

通过使用派生类，可以让程序的代码的意义更加的明确和容易阅读，可以通过改造派生类去实现更多的方法，而不用修改基类，以免影响到其他的派生类。在.NET 中，当创建一个应用程序或者是 **Web Form** 应用程序时，其实都已经默认派生自一个系统提供的基类。而派生可以允许派生用户自定义的类，示例代码如下所示。

```
public class Animal           //创建基类
{
    public string type;       //创建基类成员
    public string color;      //创建基类成员
    public string sound;      //创建基类成员
}
public class People:Animal    //创建派生类
{
    public string country;    //创建派生类成员
}
```

使用 “:” 运算符说明该类是派生自一个基类，上述代码创建了一个新的类 **Person** 来描述人类这种高级动物，“:” 运算符说明了 **People** 类派生自 **Animal**。

创建了派生类，说明了该派生类继承了基类的共有或保护的方法和属性，在派生类中可以无需在声明变量。例如上述代码中，为 **People** 类增加了 **country** 字段来描述人类的国家。而人类同样有声音、肤色等，这些基类已经提供，就可以不需要再重新声明了，可以直接通过 “.” 运算符使用，因为基类的字段或方法已经被 “继承” 了，示例代码如下所示。

```
public class Animal           //创建基类
{
```

```
public string type;           //创建基类成员
public string color;          //创建基类成员
public string sound;          //创建基类成员
}
public class People:Animal     //创建派生类
{
    public string country;     //创建派生类成员
}
class Program
{
    static void Main(string[] args)
    {
        People person = new People(); //创建派生类对象
        person.country = "China";     //初始化派生类字段
        person.color = "yellow";      //使用基类字段
    }
}
```

上述代码中，**People** 类并没有声明字段 **color**，但是 **People** 类是继承自 **Animal** 类的，而 **Animal** 类包括字段 **color**，并且该字段是共有的，所以在 **People** 类中也可以使用 **color**。

注意：当基类的字段或者方法等访问修饰符为 **public** 或 **protected** 时，继承的派生类可以使用基类的字段或方法。但是当基类的字符按或方法等访问修饰符为 **private** 时，继承的派生类不能使用基类的 **private** 字段或方法。

3.8.3 对象的创建

当创建一个派生类的对象，派生类的对象可以使用基类中共有（**public**）或保护（**private**）的类成员。之所以派生类的对象能够使用基类的类成员，是因为在创建派生类的对象的时候，首先会执行基类的构造函数，然后再执行派生类的构造函数，最后一个对象才会被创建，示例代码如下所示。

```
public class Animal           //编写类
{
    public Animal()            //编写构造函数
    {
        Console.WriteLine("Animal 被构造"); //Animal 构造函数
    }
}
public class People:Animal     //编写派生类
{
    public string country;     //声明变量
    public People()            //编写构造函数
    {
        Console.WriteLine("People 被构造"); //People 构造函数
    }
}
class Program                  //应用程序主类
{
    static void Main(string[] args) //创建主方法
    {
        People person = new People(); //创建一个 Person 对象
        Console.ReadKey();           //等待用户按键
    }
}
```

```
    }
```

运行结果如图 3-5 所示。

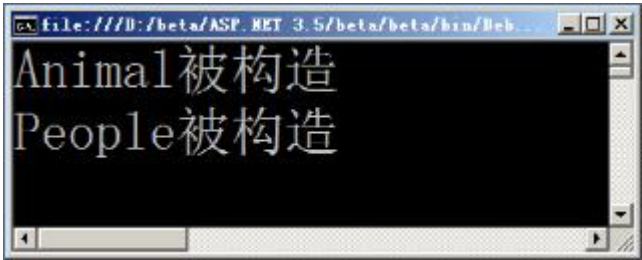


图 3-5 派生类的构造

从上述代码中可以看出，当创建了 **Person** 类的对象的时候，虽然没有创建 **Animal** 类的对象，但是还是构造了 **Animal** 类。**Animal** 的构造函数执行后，才开始执行 **Person** 的构造函数。当执行了 **Person** 构造函数，势必会执行 **Person** 基类的构造函数，如果一个派生类的基类有多个构造函数，而开发人员想指定构造函数时，必须使用 **base** 关键字，**Animal** 类示例代码如下所示。

```
public class Animal //编写类
{
    public Animal() //编写构造函数
    {
        Console.WriteLine("Animal 被构造"); //构造函数
    }
    public Animal(DateTime time) //另一个构造函数
    {
        Console.WriteLine("Animal 在" + DateTime.Now.ToString() + "被构造");//另一个代码块
    }
}
```

上述代码中，**Animal** 类具有两个构造函数，而要让派生类使用基类的某个构造函数，就必须指定派生类使用的基类的构造函数的方法，示例代码如下所示。

```
public class People:Animal //编写派生类
{
    public string country; //定义字符串型字段
    public People(DateTime time):base(time) //指定构造函数
    {
        Console.WriteLine("People 被构造"); //编写构造函数
    }
}
```

在指定了构造函数后，在主函数中也必须修改相应的对象的创建代码，示例代码如下所示。

```
static void Main(string[] args) //主入口方法
{
    People person = new People(DateTime.Now); //使用指定的构造函数
    Console.ReadKey(); //等待用户按键
}
```

运行结果如图 3-6 所示。

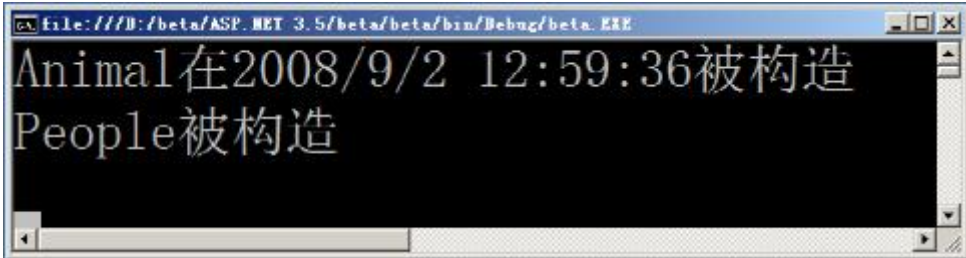


图 3-6 指定使用基类的构造函数

3.8.4 使用抽象类

在类被创建的时候，派生类的构造函数在执行前，会首先执行基类的构造函数。当声明或者设计一个类的结构时，基类往往是不完善的，也不应该把基类的类成员实例化。直接从 **Animal** 类创建对象是不正确的，因为基类的作用是为了整合派生类中公共的相同或相似的属性或字段，而对基类的成员赋值或者创建基类的对象，会使类的结构变得混乱。

例如在上面的例子中，**Person** 类从 **Animal** 类派生，当开发人员希望创建一个对象的时候，无论是从代码还是命名都能够更好的理解。虽然很多场景会不会使用人类的国家参数，但是从 **Person** 类创建对象还是比从 **Animal** 类创建对象更加好理解，示例代码如下所示。

```
People abc = new People();           //创建对象
Animal abcd = new Animal();          //创建对象
```

从上述代码中，阅读代码时，能够非常清楚的了解 **abc** 是一个人，而不是一只猫或者一只狗。而 **abcd** 却让人匪夷所思，只知道此对象是动物，而不知道具体是什么。虽然上述的例子中，**abc** 没有任何意义，在这里只是使用了开发中的某个场景，当代码过长的时候，命名也会变得困难，也会很难找到合适的命名，但是还是推荐使用有意义的名称来定义变量，例如 **person**。这里只是作为例子来说明创建对象时，好的类设计带来的好处。

创建对象时，如果对基类使用 **abstract** 关键字，那么编译器会阻止基类的直接实例化，从而可以强制的让开发人员使用正确的类让类层次结构正确并容易阅读。当用户创建 **Animal bird=new Animal()**时，编译器会报错并提示错误，示例代码如下所示。

```
public abstract class Animal          //创建抽象类
{
    public Animal()                   //创建构造函数
    {
        Console.WriteLine("Animal 被构造"); //编写构造函数
    }
}
```

当使用上述代码进行类的对象的创建时，系统会提示错误，因为该类是一个抽象类，在抽象类的基类中，系统是禁止基类的直接实例化的。

3.8.5 使用密封类

与抽象类相反的是，**C#**支持创建密封类，密封类是一种永远不能做基类的类。其他的类不能从此类派生，从而保证了密封类的密封性和安全性，使用 **sealed** 关键字能够创建密封类，示例代码如下所示。

```
public sealed class Animal             //创建密封类
{
    public Animal()                   //创建构造函数
    {
        Console.WriteLine("Animal 被构造"); //编写构造函数
    }
}
```

当 **Person** 再次从 **Animal** 派生时，编译器会提示出现错误，**Person** 类无法从密封类 **Animal** 派生。

注意：设计类的时候，通常情况下是不需要将类设置为密封类的，因为密封类会让类的扩展性非常的差，这个类也无法再次扩展和派生。但是，出于某种目的，当程序块只需要完成某些特定的功能或者在商业上为了保密，则可以使用密封类对类进行密封，保证类的可靠性。



## 3.9 多态

面向对象应用程序开发中，与传统的面向对象不同的是，面向对象具有很多的特性让开发变得简单和方便，代码便于阅读和维护，多态也是其中的重要的特性。多态可以分为两种，分别为动态多态和静态多态。上面章节中讲到的重载是多态的一种，重载是一种静态多态。

### 3.9.1 抽象方法

抽象方法是一个没有对类成员中方法进行具体实现的一种方法，抽象方法的实现必须让派生类实现。虽然抽象类中的所有方法不一定全是抽象方法，但是包含抽象方法的类被称作抽象类。抽象类可以使用 **abstract** 修饰符修饰类名称，抽象类中抽象方法的实现语法如下所示。

```
public abstract class Animal //编写类
{
    public abstract string Sound(); //创建抽象方法
}
```

上述代码中创建了抽象类 **abstract class Animal**，同时创建了抽象方法 **Sound()**。抽象方法不允许有方法体，同样不允许包含括号，只允许声明抽象方法。在派生类中，必须实现基类中的抽象方法，示例代码如下所示。

```
public class People:Animal //派生自 Animal 类
{
    public string country;
    public override string Sound() //实现抽象方法
    {
        return "language"; //返回值
    }
}
```

在派生类中，为了实现抽象的方法，就必须使用 **override** 关键字，来表示此方法是对基类的抽象方法的实现。使用抽象方法的好处在于，一位开发人员（可以是开发小组的组长或者软件构架设计师）可以创建一个或多个基类，来划分模块，或者按照功能划分和设计类，而小组的其他成员可以通过派生类来对基类进行实现，而设计基类的人员无需对类中方法的细节进行关心。同样，当修改代码时，也无需对基类进行修改，直接对派生类修改，防止当多个派生类派生于同一个基类时，出现不可预料的错误。

**注意：****Sound** 方法是一个抽象方法，在其他派生类中，如猫、狗，都有自己独特的叫声，而人类使用的是语言。不同的种别实现的方法千差万别，在基类中规定将属性规定死是非常不明智的做法。

### 3.9.2 覆盖

当基类创建了一个方法来描述类对象的时候，派生类的同一方法必须实现不同的细节，例如动物类中，初始化了方法中的声音的方法为“**miaomiao**”，那么在派生类中，可能声音的方法不是“**miaomiao**”，那么就可以对基类的方法进行覆盖，示例代码如下所示。

```
public class Animal //创建基类
{
    public string Sound() //基类中的 Sound 方法
    {
        return "miaomiao"; //返回值
    }
}
```

public class People:Animal	//创建派生类
{	
public string country;	//创建私有成员
public string Sound()	//覆盖派生类中的 Sound 方法
{	
return "language";	//返回值
}	
}	

派生类中使用了 **Sound** 方法，而基类中同样使用了此方法。当编译器编译代码的时候，会将派生类中的方法覆盖基类的方法，并在派生类中对象运行的时候，执行派生类中的方法，而不会执行基类中的方法。

注意：虽然派生类可以覆盖基类的方法，但是在设计类的时候，推荐使用抽象类或者接口来实现基类，因为这样便于阅读和维护，也提高了代码的安全性。

3.9.3 虚方法的抽象类

在类的设计中，可以使用 **abstract** 关键字修饰类为抽象类，那么在基类中就不需要实现抽象类，而必须在派生类中实现基类的方法。但是，如果在基类中，有多个方法来形容一个动物的特性，如飞行的特性可以用来形容鸟，而世界上很多动物都会飞行，但是人类是无法飞行的，所以在人类这个派生类中实现飞行的方法是没有必要，也是降低的代码的可读性的。这里就可以通过 **virtual** 关键字实现虚方法，让派生类能够选择是否实现该方法，示例代码如下所示。

public class Animal	//创建类
{	
public virtual string Fly()	//虚方法，飞行方法
{	
return "Most Of The Animal Can Fly";	//返回值
}	
}	
public class People:Animal	//创建派生类
{	
public string country;	//没有实现虚方法也可以
}	
public class Bird : Animal	//创建派生类
{	
public string FLY()	//鸟儿能飞行，实现一个虚方法
{	
return "It Can Fly";	//返回值
}	
}	

上述代码中，人类不能飞行，所以没有必要实现 **Fly()**方法，而鸟儿可以飞行，为了更好的描述一个对象，可以实现 **Fly()**方法让对象能够飞行。

技巧：在类设计中，对于多数的派生类使用的方法，可以考虑将方法放置在基类中，当一个方法，例如飞行，有一些派生类不需要使用，如人类，则可以将此方法设置为虚方法。而当一个方法，例如吃东西，是每个派生类都必须使用的，则最好将此方法设计为抽象方法，强制每个开发人员必须实现该方法。

3.9.4 抽象属性

同方法相同的是，属性也可以使用抽象属性。同样，基类的派生类也必须实现基类的抽象属性的方法，示例代码如下所示。

```
public class Animal                                     //创建基类
{
    public string Sound{get;set;}                       //创建 sound 属性
}
public class People:Animal                             //创建派生类
{
    public string country;
    public string Sound { get; set; }                  //覆盖基类属性
}
```

当抽象属性为只读或只写属性时，可以移除相应的访问器简化代码并提高相应字段的安全性。

3.10 委托和事件

委托让初学者觉得非常的疑惑和困难，委托其实就是一种引用方法的类型。委托通常与事件一起使用。通常情况下，如果为委托分配了方法，委托和声明的方法具有完全相同的功能，在 ASP.NET 应用程序的开发中，服务器控件就使用了委托和事件的思想进行了开发。

3.10.1 委托

委托的方法和其他所有的方法一样，具有参数以及返回值，委托用关键字 **delegate** 修饰，示例代码如下所示。

```
public delegate string MySound(string sound);           //声明一个委托
```

委托是一种安全的类型，并将方法安全的封装。在 C/C++应用程序开发中，C#中的委托和 C++的函数的指针类型类似，而稍有不同的是，C#中的委托是面向对象、类型安全的。委托的类型由委托的名称定义，如上述代码所示，代码中声明了 **MySound** 委托。

在委托对象被创建的时候，通常情况下提供委托包装的方法或匿名方法。实例化委托后，委托将把对它进行的方法调用传递给方法，委托允许将方法作为参数进行传递并定义回调的方法，调用方传递给委托的参数被传递给方法，相应的委托的方法的返回值返回给调用方，示例代码如下所示。

```
class Program                                           //应用程序主类
{
    public delegate void MyDel(string message);         //声明一个委托
    public static void DelegateMethod(string message)   //声明调用委托
    {
        Console.WriteLine("message is: " + message);   //输出字符串
    }
    static void Main(string[] args)                     //应用程序入口方法
    {
        MyDel del = DelegateMethod;                    //注意这里初始化
        del("Delegate Method");                         //使用委托
        Console.ReadKey();                              //等待用户按键
    }
}
```

当创建委托并实例化委托对象时，委托把对它进行的方法调用传递方法。上述代码中，将 **DelegateMethod** 方法传递，在实例化后，当使用委托时，调用方（**del**）传递给委托的参数（**Delegate Method**）被传递给了

方法（**DelegateMethod**），实现了方法，并调用、执行方法，运行结果如图 3-7 所示。

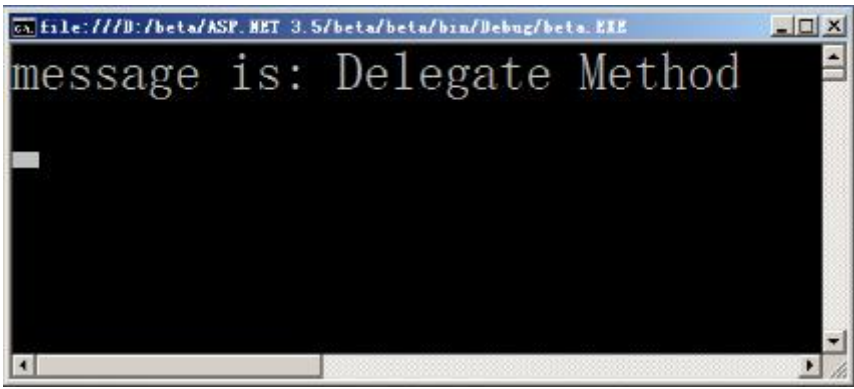


图 3-7 委托的定义和使用

3.10.2 声明事件

在 3.2.3 小结中，讲解了事件的基本概念，事件具有以下特点：

- ❑ 事件通常使用委托事件处理程序进行声明。
- ❑ 事件始终通知对象消息并指示需要执行某种操作的一种方式。
- ❑ 发行者确定何时引发事件，订阅者确定执行何种操作来响应该事件。
- ❑ 一个事件可以有多个订阅者。一个订阅者可处理来自多个发行者的多个事件。
- ❑ 没有订阅者的事件永远不会被调用。
- ❑ 事件通常用于通知用户操作，例如，图形用户界面中的按钮单击或菜单选择操作。
- ❑ 如果一个事件有多个订阅者，当引发该事件时，会同步调用多个事件处理程序，也可以使用异步处理多个事件。

事件和方法一样，通常事件和方法一起使用，事件和委托一样具有签名，但是事件的签名通过委托类型来定义，示例方法代码如下所示。

```
public delegate void MyDel(object sender, EventArgs e);           //声明一个委托
public class Event                                                //编写事件类
{
    public event MyDel EventTest;                                //声明一个事件
    public void EventTestMethod()                                //编写事件方法
    {
        Console.WriteLine("事件被使用");                        //输出字符串
    }
}
```

上述代码中，声明了一个委托，声明委托后，在 **Event** 类中声明了一个事件，这个事件绑定到委托 **MyDel**。在事件的签名中，第一个参数为引用事件源的对象，第二个参数为一个传送与事件相关的数据的类。在 **C#** 中，规范的代码编写能够让代码更具可读性。

3.10.3 引发事件

如果要引发事件，类可以调用委托，并传递所有与事件有关的参数。上面的章节中讲到，事件通常和委托一起使用，并且通过给委托发送信息，来引发事件。如果该事件没有任何处理程序，则此事件为空，所以在引发事件之前，必须先确定该事件不为空，否则会抛出 **NullReferenceException** 异常，引发事件代码如下所示。

```
public delegate void MyDel(object sender, EventArgs e);           //创建委托
public class Event                                                //编写事件类
{
    public event MyDel EventTest;                                //声明一个事件
    public void EventTestMethod()                                //声明一个事件所执行的方法
    {
    }
```



```
{
    MyDel OnLoad = EventTest;           //声明事件的方法
    if (OnLoad != null)                 //判断事件是否为空
    {
        OnLoad(this, EventArgs());      //不为空则使用委托
    }
}
```

每一个事件都可以分配多个程序来接收该事件，这样，事件自动调用每个接收器，当有多个接收器时，引发事件只需要调用一次事件。

### 3.10.4 订阅事件

事件可以像一个方法一样，若要接收某个事件的类，可以创建一个方法来接收该事件，接收事件的类像类事件自身添加方法的委托，这个被称作“订阅事件”，可以说，和平时上网中的 **RSS** 订阅整个过程很像。值得注意的是，接收器必须具有与事件自身相同的签名的方法，然后该方法才能采取适当的操作来响应事件，示例代码如下所示。

```
public class EventReceiver                //创建一个接收器
{
    public void EventTestReceiver(object sender, EventArgs e) //方法的签名必须相同
    {
        Console.WriteLine("从" + sender.ToString() + "引发了一个事件"); //执行方法体
    }
}
```

每一个事件可以分配多个程序来接收该事件，也就是说可以有多个接收器。多个接收器由源按照顺序调用。如果一个接收出现异常，则没有接收的接收器会接收事件。如果要订阅事件，接收器必须创建一个与事件具有同种类型的委托，并使用事件处理委托的目标，这也就是为什么事件通常情况下会与委托一起使用。示例代码如下所示。

```
public void EventTestSubscribe(Event eve)
{
    MyDel del = new MyDel(EventTestReceiver); //声明委托
    eve.EventTest += del;                     //增加事件
}
```

上述代码中，通过“+=”运算符订阅了一个事件，同样，也可以使用“-=”号取消订阅。示例代码如下所示。

```
public void EventTestSubscribe(Event eve)
{
    MyDel del = new MyDel(EventTestReceiver); //声明委托
    eve.EventTest -= del;                     //取消事件
}
```

### 3.10.5 委托和事件

上面几节中分开讲解委托和事件，对于初学者而言，委托和事件是很难学习的知识，但是当学习过委托和事件之后，会发现委托和事件非常的简单。在 **ASP.NET** 开发当中，很多控件都使用了委托和事件。例如当单击一个按钮控件时，按钮会发送信息指示“引发了一个按钮事件”，然后发送给相应的接收器，接收器接收了发来的信息从而引发相应的操作。在了解委托和事件的基本概念后，下列代码说明了怎样一步步的使用委托和事件。

为了实现广播喇叭功能（类似 **QQ** 的聊天窗口的系统信息），应用程序中不仅有用户的聊天窗口，也包括系统发送窗口。系统可以给用户的聊天窗口发送系统信息，在应用程序中，不仅需要广播用户的信息，

同样系统也能够广播系统信息。为了实现这一功能，首先，需要创建一个委托，示例代码如下所示。

```
public delegate void BetaDel string str;                                     //创建一个委托

在创建了委托后，就要为写方法，示例代码如下所示。

public delegate void BetaDel(string str);
public static void Output(string str)                                     //用户发送信息方法
{
    Console.WriteLine("用户发送给你一个消息");                           //输出用户提示信息
}
public static void SystemOutput(string str)                             //系统发送信息方法
{
    Console.WriteLine("系统发送给你一个消息");                           //输出用户提示信息
}
public static void OutputChoose(string str,BetaDel del)                  //使用委托变量
{
    del(str);
}
```

注意：在上述代码中，del 是一个委托变量，del(str)会按照方法的签名在委托的方法表中执行。上述代码，与 del(string str)签名相同的方法有 Output 和 SystemOutput，他们的方法签名相同。

在主函数中，可以通过委托来使用方法，示例代码如下所示。

```
static void Main(string[] args)
{
    OutputChoose("你好", Output);                                         //通过传递方法名称来使用方法
    Console.ReadKey();
}
```

上述代码中，使用了 OutputChoose 方法。值得注意的是，在 OutputChoose 方法中，其中的一个参数是方法名称。因为通过委托，可以将方法名称作为参数进行传递，从而执行了相应的方法。值得注意的是，在上述代码中，委托等方法全部都声明在一个类中，因为这样能够方便理解，但是这样就不具备面向对象的特点，面向对象的特性就是封装，封装能让代码具有结构性，于是可以使用事件。创建一个类，类名称叫 OutputChoose，示例代码如下所示。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;                                                         //使用系统命名空间
namespace beta
{
    class OutputChoose
    {
        public string message="你有新短消息,请注意查收";                //声明短消息字符串
        public delegate void BetaDel(string str);                        //定义委托注册事件
        public event BetaDel MyEvent;                                     //声明事件
        public void OnLoad()                                              //编写 OnLoad 方法注册事件
        {
            if (MyEvent != null)
            {
                MyEvent(message);                                         //当存在事件时，调用所有注册对象的方法
            }
        }
    }
}
```

上述代码将前面代码中的方法进行了封装作为委托。然后添加一个用户消息类，类名为 UserMessage，示例代码如下所示。

```
using System.Linq;
using System.Text;                                                         //使用文本处理命名空间
```

```
namespace beta //声明当前程序命名空间
{
    class UserMessage
    {
        public void Output(string str) //输出方法
        {
            Console.WriteLine("用户发送给你一个消息:" + str); //简单的输出
        }
    }
}
```

再添加一个系统消息类，类名称为 **SystemMessage**，示例代码如下所示。

```
using System.Linq;
using System.Text; //使用文本处理命名空间
namespace beta //声明当前程序命名空间
{
    class SystemMessage
    {
        public void SystemOutput(string str) //系统获取输出方法
        {
            Console.WriteLine("系统发送给你一个消息:" + str); //显式系统发送的消息
        }
    }
}
```

在主函数中，可以触发事件来，示例代码如下所示。

```
static void Main(string[] args)
{
    OutputChoose opc = new OutputChoose(); //声明一个类的对象
    SystemMessage msg = new SystemMessage();
    opc.MyEvent += msg.SystemOutput; //注册方法
    opc.OnLoad(); //开始自动调用所有注册的方法
    Console.ReadKey();
}
```

上述代码中，**OnLoad()**触发了之前注册的事件，并执行事件，运行结果如图 3-8 所示。



图 3-8 委托和事件的综合用例

运行结果显示，当创建了一个对象，对象可以注册声明事件，因为该对象没有实现该事件的方法的具体实现，但是在事件中增加了方法，类似于在该类中增加了一个方法，而在该类的编码实现中，定义了一个 **OnLoad** 方法来调用所有注册对象的方法。

## 3.11 类命名

.NET 框架系统中类的命名总是包含着各种含义，无论是命名空间还是类甚至是变量。良好的命名规范这能够让使用它的人非常容易理解并方便阅读和使用。在系统开发中，对于程序开发人员而言，也推荐统一并按照一定的规范来命名，这用同样为了方便阅读和维护。

3.11.1 命名空间的命名

在.NET 框架中，包含很多系统的命名空间，示例代码如下所示。

```
using System;                                //System 命名空间
using System.Collections.Generic;            // System.Collections.Generic 命名空间
using System.Linq;                          // System.Linq 命名空间
using System.Text;                          // System.Text 命名空间
```

上述代码中，开发人员能够非常方便的了解使用了什么命名空间，这个命名空间大概都能做什么操作，例如 **System.Linq** 就是为了支持 3.5 中 **LINQ** 的特性而提供的命名空间。当开发人员创建命名空间时，命名空间的名称应该避免与公司名称或其他品牌的名称相同，例如为了做 **Office** 开发扩展，可能会命名为 **Microsoft.Office**，但是此命名空间已经在.NET 框架中被使用了，所以编译器会报错。

技巧：尽量使用开发人员开发组或公司的名称作为命名空间，因为开发组或公司的名称能够表示这个程序或组件是来自哪里。不仅如此，开发组或公司的名称也能在一定程度上避免了重复，例如 **HuaWei.TcpIp.Class** 既能表示华为的 TCP/IP 研发小组，又可以很大程度上的避免重复。

3.11.2 类的命名原则

同样，类的命名也是有原则的，其原则基本上同命名空间一样，类名尽量使用 **Pascal** 大写，减少类名称的所写的使用量，并且不推荐使用前缀和下划线。正确的类命名如下所示。

```
class SystemMessage                          //良好的命名
{
}
```

上述代码命名了 **SystemMessage** 类，用来表示系统发出的消息，**SystemMessage** 类的名称能够让开发人员清晰的了解该类的基本用途。

3.11.3 接口的命名原则

接口的命名原则与类基本相同，不同的是，在接口的命名中，接口名前应加上大写字母 **I** 来表示这是一个接口而不是一个类。示例代码如下所示。

```
public interface ISystemMessage              //良好的接口命名
{
    void SystemOutput(string str);
}
public interface IUserMessage               //良好的接口命名
{
    void Output(string str);
}
```

3.11.4 属性的命名原则

属性的命名中，通常需要在属性名后加上 **Attribute**，来表示自定义属性类，说明该类用来封装属性，示例代码如下所示。

```
public class PersonInformationAttribute      //属性命名
{
}
```



}

3.11.5 枚举的命名原则

枚举的命名同样使用 **Pascal** 大写。枚举值的名称同样需要使用 **Pascal** 大写，并同样避免枚举名称中所写的使用量。枚举的命名不需要加入任何的后缀，示例代码如下所示。

```
public enum FileType
{
    Txt,                //定义枚举成员 Txt
    Mp3,                //定义枚举成员 Mp3
    Mp4,                //定义枚举成员 Mp4
    Doc,                //定义枚举成员 Doc
    Pdf,                //定义枚举成员 Pdf
    Html,               //定义枚举成员 Html
    Htm,                //定义枚举成员 Htm
}
```

如果需要使用数字值，这可以使用 **Flags** 对属性进行自定义，示例代码如下所示。

```
[Flags]                //使用 Flags 属性标记
public enum FileType
{
    Txt,                //定义枚举成员 Txt
    Mp3,                //定义枚举成员 Mp3
    Mp4,                //定义枚举成员 Mp4
    Doc,                //定义枚举成员 Doc
    Pdf,                //定义枚举成员 Pdf
    Html,               //定义枚举成员 Html
    Htm,                //定义枚举成员 Htm
}
```

在 **Win32 API** 中，这种类型是非常常见的，但是不同的是，**Win32 API** 中的枚举成员都是大写的，但是在.NET 中还是推荐使用 **Pascal** 大写。

3.11.6 只读字段的命名原则

只读字段在应用程序的开发中通常作为不可改变的变量而使用，只读字段一旦进行了声明，在代码中就无法对只读字段进行更改。只读字段通常用于声明那些在现实中规定好的变量，例如  $\pi$  的值、一千克的精确重量等等。

只读字段的命名同一般的变量的声明方法相同，但是只读字段推荐使用 **Pascal** 大写命名规则进行只读字段的变量命名。

3.11.7 参数名

参数名应该具有描述性，即一个参数能够描述当前参数的意义，例如 **computerUser** 能够描述一台电脑的使用者，而不应该命名为其他没有意义的字符，例如 **abc**，并推荐使用 **camel** 大写方式命名。参数的命名应该根据参数的意义来命名，而不是根据参数的种类，并且不提倡使用保留参数、下划线等。示例代码如下所示。

```
public void Set(string computerUser)    //参数声明
{ }
```

## 3.11.8 委托命名原则

对于委托，使用 **EventHandler** 作为后缀命名委托处理程序，示例代码如下所示。

```
public delegate void BetaDelHandler(string str); //委托命名
```

对于委托的参数，也有相应的命名规范，在参数中，使用名为 **sender** 和 **e** 两个参数，**sender** 参数代表提出委托的对象。**sender** 参数是一个类型的对象，为 **Object** 类的对象，**e** 代表与事件相关的状态，示例代码如下所示。

```
public delegate void BetaDelHandler(Object sender,EventArgs e); //委托的参数
```

## 3.12 小议设计模式

在应用程序开发中，良好的命名规范是为了规范代码，让代码更加容易维护和阅读。同样，设计模式是一种软件设计方法，也是为了开发出来的应用程序能够更好的使用和维护。在应用程序设计中，维护成本在软件设计占 **70%**或更高，所以良好的软件结构能够为后期开发和维护提供便利。

### 3.12.1 什么是设计模式

模式，就是一种规范，在新华字典上关于模式的解释如下。

□ 法式，规范，标准：模范。模式。楷模。模型。模本。模压。

□ 仿效：模仿（亦作“摹仿”）。模拟（亦作“摹拟”）。模写。

那么，所谓的设计模式，就是软件设计的一种范例。虽然定义看上去非常简单，看是实际上初学者是非常难以理解的，设计模式是一项长期的学习，需要不停的使用和学习才能掌握。

### 3.12.2 为什么要使用设计模式

设计模式是一种编码的范例。在软件开发过程中，不同的人对类的设计不同，命名的方法也不同，在类的结构上的设计也不同，通常会导致代码混乱，难以阅读和难以维护，以至于上个世纪出现了所谓的“软件危机”。于是人们在软件开发的经验中找到一种模式，所谓的模式，就是一种规律、一种规范，就例如现在的教育模式，大家都会从小学开始，慢慢的读到大学，是一种经典的模式。在软件的开发过程中，也存在这样的模式，就是通常情况下所说的设计模式。

设计模式让人们在软件设计中有据可循。例如在开发一套网站管理系统，首先系统设计师需要对网站进行需求分析，在需求分析之后，就需要规定接口，来说明开发中的类的规范。在规范制定好以后，开发人员需要按照规范来开发软件，并按照统一的类结构或风格编写代码。

设计模式基本上规定了类的结构，规定了类是怎样派生，以及怎样引用。常用的有 **28** 种设计模式，经常使用的成熟的设计模式在 **9** 种左右。设计模式的学习是一个长期的过程，初学者在理论上很难理解设计模式，是因为如果没有一定的编码基础，是很难理解为何要使用设计模式的，因为很短的代码段，基本上难以涉及到多人开发，以及市场的考验的经验。

### 3.12.3 改装现有类

使用设计模式，就需要对现有的代码不停的重构。例如不规范的命名，以及在实际项目过程中遇到的类的结构设计错误等，都需要改装，改装现有类并不是纯属为了类的“好看”，而是为了在实际的运用过程中，能够胜任新的系统要求，并且能够比原有的结构有很好的扩展能力，方便维护企业开发成本。例如，

在设计一个制造汽车的过程，设计了一个汽车类，示例代码如下所示。

```

class car                                     //设计汽车类
{
    string CarNumber;                         //设计汽车编号
    string CarType;                           //设计汽车类型
    public string ShowCar()                   //显式汽车方法
    {
        return "this is a car";
    }
}
class Program                                //主程序类
{
    static void Main(string[] args)           //入口方法
    {
        car car = new car();                 //创建一个新对象
        car.ShowCar();                       //使用对象的方法
    }
}
    
```

上述代码中，在同一个文件内声明了一个汽车类 **class car**，在 **Main** 方法中，创建了一个 **car** 的对象，并使用对象 **car** 的 **showcar** 方法。从上述代码中可以看出，这其中有几个非常不好的习惯，归纳如下所示。

- ☐ 命名没有按照规范，看起来没有层次。
- ☐ 命名中的字段没有按照规范。
- ☐ 命名中的字段推荐使用属性的方法。
- ☐ 类与使用对象的方法放在同一个文件中。

上面的习惯中，最后一条习惯可能读者会有疑问，会在想为何不能放在同一个文件中。其实，在语法上是没有错误的，不好的是，每当汽车类需要更改，开发人员就要重新修改 **car** 类并重新编译主方法，这样的效率非常的低。就好像当你做了一个网站，更新一个网页就必须更新整个网站一样，这样成本是非常高，而且不利于维护的。最好的做法是将类封装在类库中，存储在“后台”，在“前台”中创建类的对象，而更新细节的时候，只需要更新类库即可。

在设计模式的学习和使用过程中，是没有最好的设计模式的，而另一方面，虽然没有最好的设计模式但是有最适合的设计模式。设计模式的熟练使用，通常要求开发人员有较熟练的编码以及较好的编码习惯，并且在项目开发上有一定的经验。

## 3.13 小结

在这一章中，介绍了 **C#**面向对象的特性。面向对象在构建强大的应用程序中，起着重要的作用。同样，面向对象的设计思想为维护做了铺垫，为了让读者能够加深面向对象的概念，本章还讲解了基本的设计模式的概念。

- ☐ 什么是面向对象：介绍了面向对象的概念。
- ☐ 面向对象的 **C#**实现：使用 **C#**介绍面向对象。
- ☐ 对象的生命周期：讲解了构造函数、析构函数、垃圾回收机制等对象的生命周期的概念。
- ☐ 使用命名空间：讲解了使用命名空间，以及创建自定义命名空间。
- ☐ 类的方法：讲解了什么是类、类成员、字段、方法等基本知识。
- ☐ 封装：讲解了封装以及为何要封装。
- ☐ 属性：详细讲解了 **C#**中属性的概念，声明和使用方法。
- ☐ 继承：讲解了面向对象中的继承的概念，并用 **C#**实现。
- ☐ 多态：讲解了面向对象中的多态的概念，并用 **C#**实现。
- ☐ 委托和事件：讲解了委托和事件的概念，深入讲解了怎样使用委托。
- ☐ 类命名：讲解了命名规范，以及为何要规范命名。

- 小议设计模式：加深读者对面向对象的概念，从实际出发，讲解设计模式的基本概念，以及为何要使用设计模式进行软件开发。

本章讲解了 **C#** 中面向概念的特性，面向概念不仅局限于 **C#**，在其他面向对象的编程语言中，基本的面向对象的概念都是差不多的，而 **JAVA/C#** 可以说是纯面向对象的编程语言，在 **.NET 3.5** 中，**C#** 还加强了面向对象的特性。



## 第4章 ASP.NET 的网页代码模型及生命周期

从本章开始，就进入了 **ASP.NET** 应用程序开发的世界。在了解了 **C#** 的结构，以及面向对象的概念后，就可以从面向对象的思想开发 **ASP.NET** 应用程序。在 **ASP.NET** 中，能够使用面向对象思想和软件开发中的一些思想，例如封装、派生、继承以及高级的设计模式等。本章首先介绍 **ASP.NET** 中最重要的概念——网页代码模型。

### 4.1 ASP.NET 的网页代码模型

在 **ASP.NET** 应用程序开发中，微软提供了大量的控件，这些控件能够方便用户的开发以及维护。这些控件具有很强的扩展能力，在开发过程中无需自己手动编写。不仅如此，用户还能够创建自定义控件进行应用程序开发以扩展现有的服务器控件的功能。

#### 4.1.1 创建 ASP.NET 网站

在 **ASP.NET** 中，可以创建 **ASP.NET** 网站和 **ASP.NET** 应用程序，**ASP.NET** 网站的网页元素包含可视元素和页面逻辑元素，并不包含 **designer.cs** 文件。而 **ASP.NET** 应用程序包含 **designer.cs** 文件。创建 **ASP.NET** 网站，首先需要创建网站，单击【文件】按钮，在下拉菜单中选择【新建网站】选项，单击后会弹出对话框用于 **ASP.NET** 网站的创建，如图 4-1 所示。



图 4-1 新建 ASP.NET 网站

在【位置】选项中，旁边的【下拉菜单】可以按照开发的需求来写，一般选择文件系统，地址为本机的本地地址。语言为.NET 网站中使用的语言，如果选择 **Visual C#**，则默认的开发语言为 **C#**，否则为 **Visual Basic**。创建了 **ASP.NET** 网站后，系统会自动创建一个代码隐藏页模型页面 **Default.aspx**。**ASP.NET** 网页一般由三部分组成，这三个部分如下所示。

- ❑ 可视元素：包括 **HTML**，标记，服务器空间。
- ❑ 页面逻辑元素：包括事件处理程序和代码。
- ❑ **designer.cs** 页文件：用来为页面的控件做初始化工作，一般只有 **ASP.NET** 应用程序（**Web**

Application) 才有。

ASP.NET 页面中包含两种代码模型，一种是单文件页模型，另一种是代码隐藏页模型。这两个模型的功能完全一样，都支持控件的拖拽，以及智能的代码生成。

4.1.2 单文件页模型

单文件页模型中的所有代码，包括控件代码、事物处理代码以及 HTML 代码全都包含在.aspx 文件中。编程代码在 script 标签，并使用 runat="server"属性标记。创建一个单文件页模型，在【文件】按钮中选择【新建文件】选项，在弹出对话框中选择【Web 窗体】或在右击当前项目，在下拉菜单中选择【添加新建项】选项即可创建一个.aspx 页面，如图 4-2 所示。

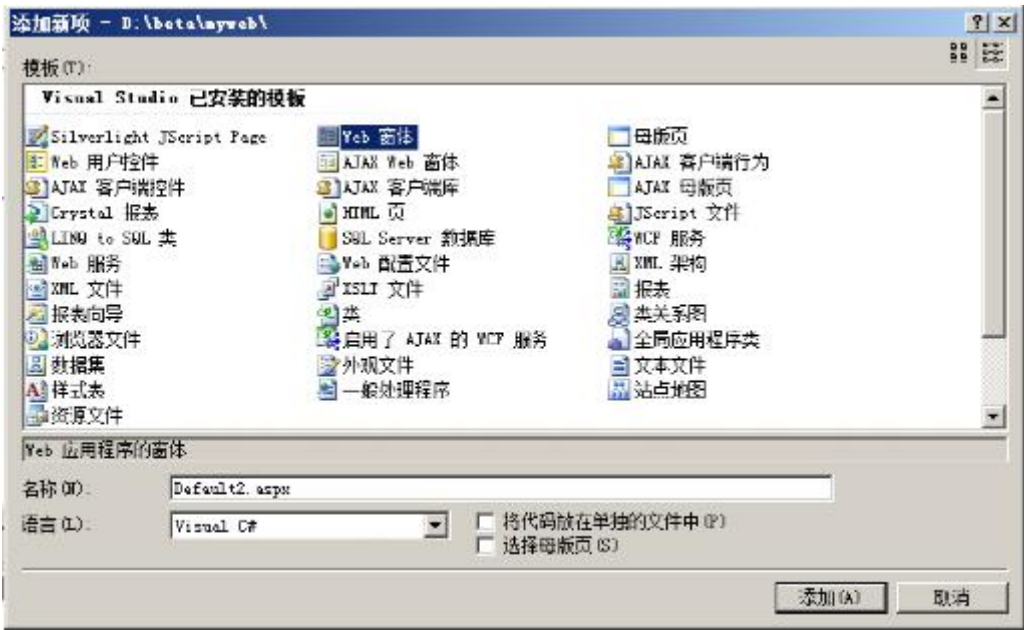


图 4-2 创建单文件页模型

在创建时，去掉【将代码放在单独的文件中】复选框的选择即可创建单文件页模型的 ASP.NET 文件。创建后文件会自动创建相应的 HTML 代码以便页面的初始化，示例代码如下所示。

```
<%@ Page Language="C#" %>
<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<script runat="server">
</script>
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>无标题页</title>
</head>
<body>
  <form id="form1" runat="server">
    <div>
    </div>
  </form>
</body>
</html>
```

编译并运行，即可看到一个空白的页面被运行了。ASP.NET 单文件页模型在创建并生成时，开发人员编写的类将编译成程序集，并将该程序集加载到应用程序域，并对该页的类进行实例化后输出到浏览器。可以说，.aspx 页面的代码也即将会生成一个类，并包含内部逻辑。在浏览器浏览该页面时，.aspx 页面的类实例化并输出到浏览器，反馈给浏览者。ASP.NET 单文件页模型运行示例图如图 4-3 所示。

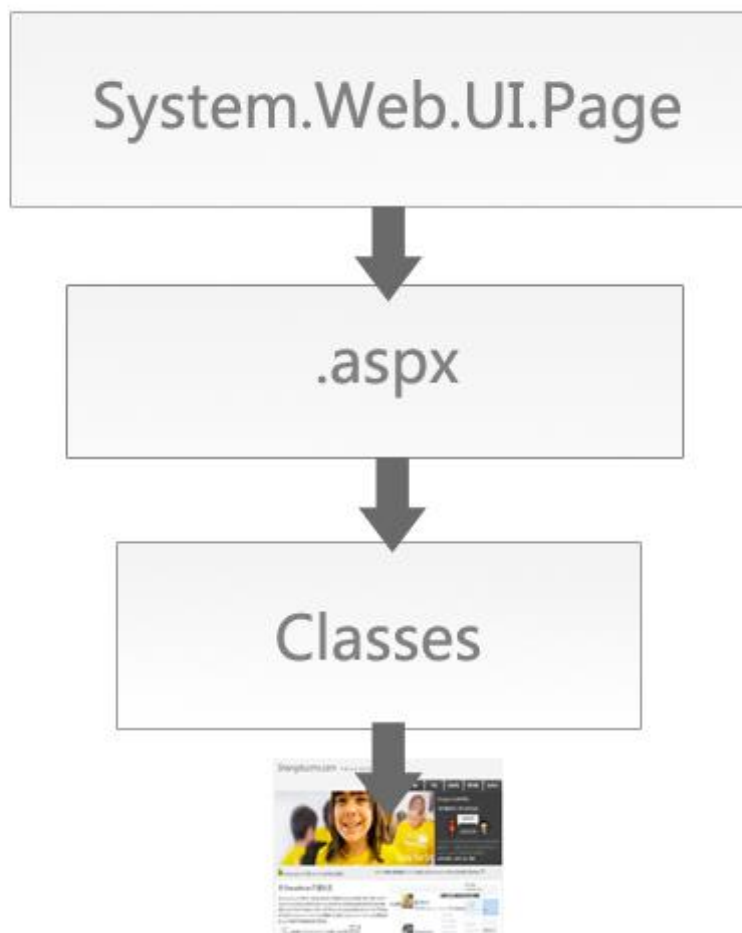


图 4-3 单文件页模型

## 4.1.3 代码隐藏页模型

代码隐藏页模型与单文件页模型不同的是，代码隐藏页模型将事物处理代码都存放在 **cs** 文件中，当 **ASP.NET** 网页运行的时候，**ASP.NET** 类生成时会先处理 **cs** 文件中的代码，再处理 **.aspx** 页面中的代码。这种过程被成为代码分离。

代码分离有一种好处，就是在 **.aspx** 页面中，开发人员可以将页面直接作为样式来设计，即美工人员也可以设计 **.aspx** 页面，而 **.cs** 文件由程序员来完成事务处理。同时，将 **ASP.NET** 中的页面样式代码和逻辑处理代码分离能够让维护变得简单，同时代码看上去也非常的优雅。在 **.aspx** 页面中，代码隐藏页模型的 **.aspx** 页面代码基本上和单文件页模型的代码相同，不同的是在 **script** 标记中的单文件页模型的代码默认被放在了同名的 **.cs** 文件中，**.aspx** 文件示例代码如下所示。

```
<%@ 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>无标题页</title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
        </div>
    </form>
</body>
</html>
```

从上述代码中可以看出，在头部声明的时候，单文件页模型只包含 **Language="C#"**，而代码隐藏页模型包含了 **CodeFile="Default.aspx.cs"**，说明被分离出去处理事物的代码被定义在 **Default.aspx.cs** 中，示例代码如下所示。

```
using System.Linq;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.HtmlControls;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
    }
}
```

//使用 HtmlControls  
 //使用 WebControls  
 //使用 WebParts  
 //继承自 System.Web.UI.Page

上述代码为 **Default.aspx.cs** 页面代码。从上述代码可以看出，其格式与类库、编写类的格式相同，这也说明了 **.aspx** 页面允许使用面向对象的特性，如多态、继承等。但是 **ASP.NET** 代码隐藏页模型的运行过程比单文件页模型要复杂，运行示例图如图 4-4 所示。

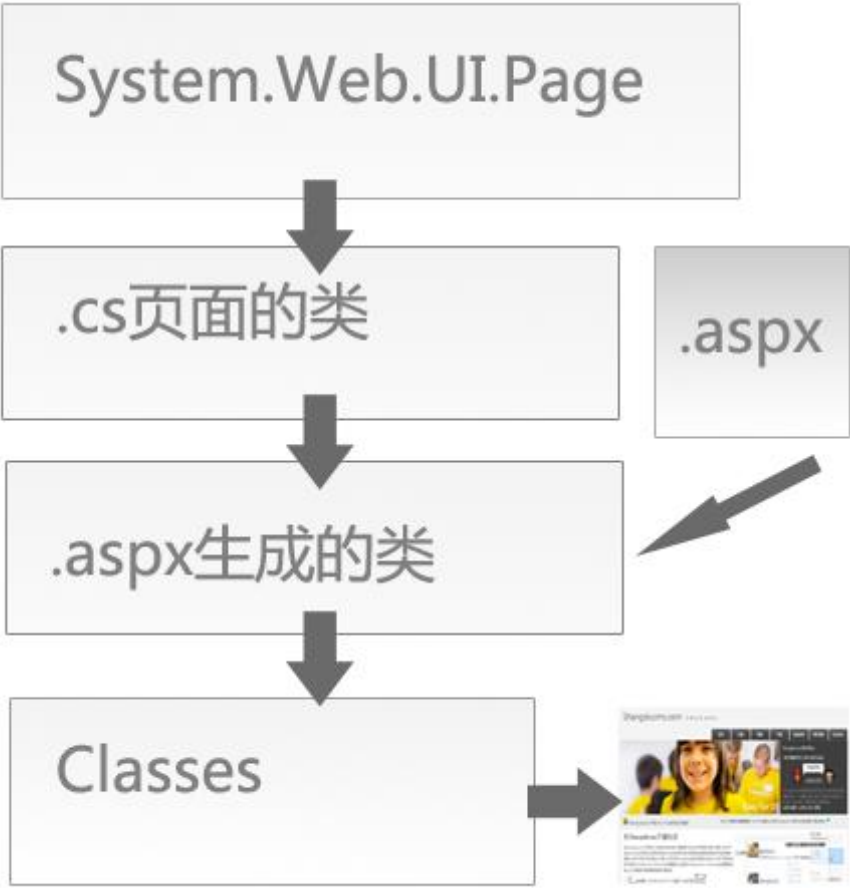


图 4-4 代码隐藏页模型

上述描述了代码隐藏类模型的页面生成模型。当页面被呈现之前，**ASP.NET** 应用程序会解释并编译相应的 **cs** 文件中的代码，与此同时，**ASP.NET** 应用程序还会将 **.aspx** 页面进行编译并生成 **.aspx** 页面对应的类。生成 **.aspx** 页面对应的类后会将该类与 **cs** 文件中的类进行协调生成新的类，该类会通过 **IIS** 在用户浏览页面时呈现在用户的浏览器中。

### 4.1.4 创建 ASP.NET Web Application

**ASP.NET** 网站有一种好处，就是在编译后，编译器将整个网站编译成一个 **DLL**（动态链接库），在更新的时候，只需要更新编译后的 **DLL**（动态链接库）文件即可。但是 **ASP.NET** 网站却有一个缺点，编译速度慢，并且类的检查不彻底。

相比之下，**ASP.NET Web Application** 不仅加快了速度，只生成一个程序集，而且可以拆分成多个项目进行管理。创建 **Application**，首先需要新建项目用于开发 **Web Application**，单击菜单栏上的【文件】按钮，在下拉菜单中选择【新建项目】选项，在弹出窗口中选择【**ASP.NET 应用程序**】选项，如图 4-5 所示。





图 4-5 创建 ASP.NET 应用程序

在创建了 ASP.NET 应用程序后，系统同样会默认创建一个 **Default.aspx** 页面，不同的是，多出了一个 **Default.aspx.designer.cs**，用来初始化页面控件，一般不需要修改。

## 4.1.5 ASP.NET 网站和 ASP.NET 应用程序的区别

在 ASP.NET 中，可以创建 ASP.NET 网站和 ASP.NET 应用程序，但是 ASP.NET 网站和 ASP.NET 应用程序开发过程和编译过程是有区别的。ASP.NET 应用程序主要有以下特点：

- ❑ 可以将 ASP.NET 应用程序拆分成多个项目以方便开发，管理和维护。
- ❑ 可以从项目中和源代码管理中排除一个文件或项目。
- ❑ 支持 VSTS 的 Team Build 方便每日构建。
- ❑ 可以对编译前后的名称，程序集等进行自定义。
- ❑ 对 App\_GlobalResources 的 Resource 强类支持。

ASP.NET WebSite 编程模型具有以下特点：

- ❑ 动态编译该页面，而不用编译整个站点。
- ❑ 当一部分页面出现错误不会影响到其他的页面或功能。
- ❑ 不需要项目文件，可以把一个目录当作一个 Web 应用来处理。

总体来说，ASP.NET 网站适用于较小的网站开发，因为其动态编译的特点，无需整站编译。而 ASP.NET 应用程序适应大型的网站开发、维护等。

## 4.2 代码隐藏页模型的解释过程

在 ASP.NET 的代码隐藏页模型中，一个完整的.aspx 页面包含两个页面，分别是以.aspx 和.cs 文件为后缀的文件，这两个文件在形成了整个 Web 窗体。在编译的过程中都被编译成由项目生成的动态链接库（.DLL），同时，.aspx 页面同样也会编译。但是与.cs 页面编译过程不同的是，当浏览者第一次浏览到.aspx 页面时，ASP.NET 自动生成该页的.NET 类文件，并将其编译成另一个.DLL 文件。

当浏览者再一次浏览该页面的时候，生成的.DLL 就会在服务器上运行，并响应用户在该页面上的请求或响应，ASP.NET 应用程序的解释过程图如 4-6 所示。

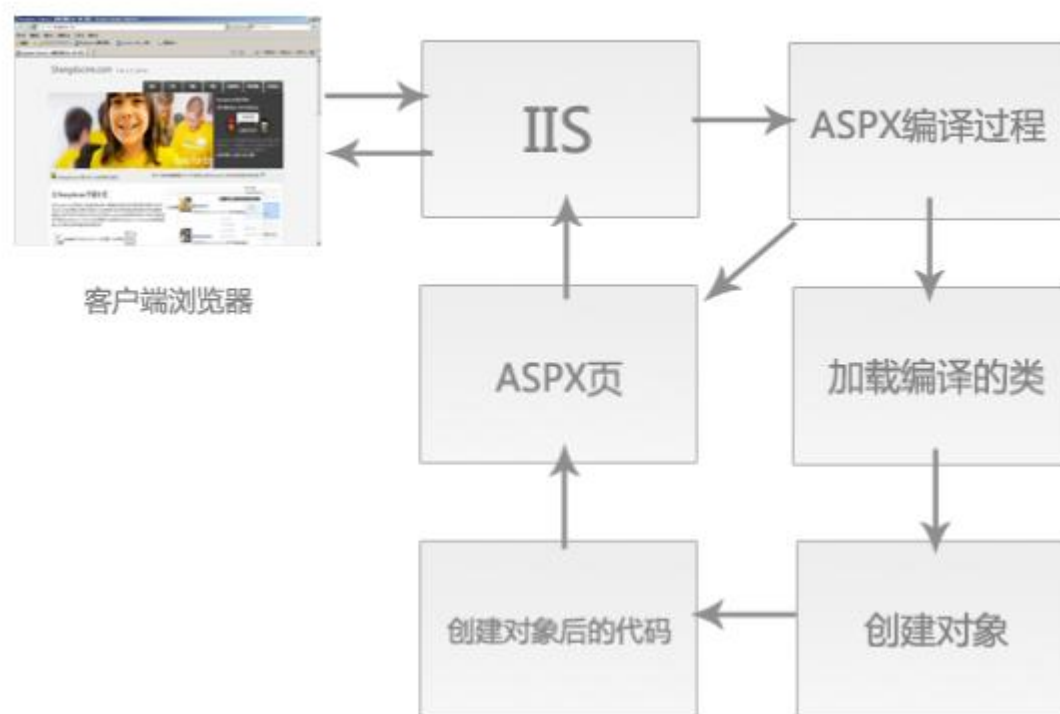


图 4-6 代码隐藏页模型页面的执行过程

在客户端浏览器访问该页面时，浏览器会给 IIS 发送请求消息，IIS 则会开始执行 ASP.NET 编译过程，如果不存在编译过后的 DLL 文件，则加载编译的类并创建对象。当创建对象完成，生成创建对象后的代码并生成一个 ASPX 页面代码，该页面代码反馈给 IIS，IIS 再反馈成 HTML 页面的形式给客户端。

## 4.3 代码隐藏页模型的事件驱动处理

在传统的 ASP 开发中，ASP 的事件都是按照网页的顺序来处理的，一般情况下，ASP 页面的事件都是从上到下处理事件，可以说 ASP 的开发是一个线性的处理模型。在用户的浏览操作中，每一次用户的操作都会导致页面重新被发送到服务器。因此，重复的操作必然导致客户端和服务器的往返过程，服务器必须重新创建页面，当创建页面后，服务器再按照原来的从上到下的顺序进行事件处理。

在 ASP.NET 中，通过使用模拟事件驱动模型的行为代替了 ASP 的线性处理模型。ASP.NET 页框架模型隐式的为用户建立了事件和事件处理程序的关联。ASP.NET 让用户可以为从浏览器传递的事件在服务器代码中设置相应的处理程序。假设某个用户正在浏览网站并与页面之间产生了某种交互，用户的操作就会引发事件，事件通过 HTTP 被传输到服务器。在服务器中，ASP.NET 框架解释信息，并触发事件与之对应的处理程序。该程序可以是.aspx 页面中的处理程序，也可以是开发者自定义的类库，或者 COM 组件等。事件驱动处理如图 4-7 所示。



图 4-7 页面框架的事件驱动处理模型

上图则说明了当一个浏览者通过浏览器触发 **ASPX** 页面时，浏览器、服务器和服务器返回页的过程。

## 4.4 ASP.NET 客户端状态

**Web** 开发不像软件开发，**Web** 应用实际上是没有状态的，这就说明 **Web** 应用程序不自动指示序列中的请求是否来自相同的浏览器或客户端，也无法判断浏览器是否一直在浏览一个页面或者一个站点，也无法判断用户执行了哪个操作并统计用户的喜好。

#### 4.4.1 视图状态

从上面的章节中可以知道，当服务器每次的往返过程，都将销毁页面并重新创建新的页面。如果一个页面中的信息超出了页面的生命周期，那么这个页面中的相关信息就不存在了。如果注销了页面的信息，那么用户的一些信息可能就不存在了。

在 **ASP.NET** 中，网页包含视图状态来保存用户的信息，视图状态在页面发回到自身时，跨页过程存储和用户自己的页面的特定值，视图状态的优点如下所示。

- ❑ 不需要任何服务器资源。
- ❑ 在默认情况下，对控件启用状态的数据进行维护，不会被破坏。
- ❑ 视图状态的值经过哈希运算和压缩保护，安全性更高。

❑ 在默认情况下，对控件启用状态的数据进行维护，不会被破坏。

❑ 视图状态的值经过哈希运算和压缩保护，安全性更高。

视图状态同样有一些缺点，缺点如下所示。

- ❑ 视图状态会影响性能，如果页面存储较大较多的值，则性能会有较大的影响。
- ❑ 在手机，移动终端上，可能无法保存视图状态中使用的值。
- ❑ 视图状态虽然安全性较高，但是还是有风险，如果直接查看页面代码，可以看到相应代码。

❑ 在手机，移动终端上，可能无法保存视图状态中使用的值。

❑ 视图状态虽然安全性较高，但是还是有风险，如果直接查看页面代码，可以看到相应代码。

### 4.4.2 控件状态

**ASP.NET** 中还提供了控件状态属性作为在服务器往返过程中存储自定义控件中的数据的方法。在页面控件中，如果有多个自定义控件使用多个不同的控件来显示不同的数据结构，为了让这些页面控件能够在页面上协调的工作，则需要使用控件状态来保护控件，同时，控件状态是不能被关闭的。同样，控件状态也有它的优点，优点如下所示。

- ☐ 与视图状态相同的是，不需要任何服务器资源。
- ☐ 控件状态是不能被关闭的，提供了控件管理的更加可靠的方法。
- ☐ 控件状态具有通用性。

### 4.4.3 隐藏域

在 **ASP** 中，通常使用隐藏域保存页面的信息。在 **ASP.NET** 中，同样具有隐藏域来保存页面的信息，作为维护页面状态的一种形式，但是隐藏域的安全性并不高，最好不要在隐藏域保存过多的信息。隐藏域具有以下优点。

- ☐ 不需要任何服务器资源。
- ☐ 支持广泛，任何客户端都支持隐藏域。
- ☐ 实现简单，隐藏域属于 **HTML** 控件，无需像服务器控件那样有需要编程知识。

而隐藏域具有一些不足，如下所示。

- ☐ 具有较高的安全隐患。
- ☐ 存储结构简单。
- ☐ 同样，如果存储了较多的较大的值，则会导致性能问题。
- ☐ 如果隐藏域过多，则在某些客户端中被禁止。
- ☐ 隐藏域将数据存储在服务器上，而不存储在客户端。

注意：如果开发中，页面的隐藏域过多，这些隐藏域被存储在服务器。当客户端浏览页面的时候，会有一些防火墙扫描页面，以保证操作系统的安全，如果页面的隐藏域过多，那么这些防火墙可能会禁止页面的某些功能。

### 4.4.4 Cookie

**Cookie** 在客户端用户保存网站的少量的用户信息，服务器可以通过编程的方法获取用户信息，**Cookie** 信息和页面请求通常一起发送到服务器，服务器对客户端传递过来的 **Cookie** 信息做处理。通常 **Cookie** 保存用户的登录状态、用户名等基本信息等等，在后面的章节会详细介绍使用 **ASP.NET** 操作 **Cookies**。

### 4.4.5 客户端状态维护

虽然使用某些客户端状态并不使用服务器资源，但是这些状态都具有潜在的安全隐患，如 **Cookie**。非法用户可以使用 **Cookie** 欺骗来攻击网站进行用户信息的获取，不过使用客户端状态能够使用客户端的资源从而提高服务器性能。使用客户端状态，虽然有安全隐患，但是具有良好的编程能力，以及基本的安全知识，能够较好的解决安全问题，同时也能够提高服务器性能。下面小结了一些客户端状态的优缺点。

- ☐ 视图状态：推荐当存储少量挥发到自身的页面的信息时使用。
- ☐ 控件状态：不需要任何服务器资源，控件状态是不能被关闭的，提供了控件管理的更加可靠和更通



用的方法。

- ❑ 隐藏域：实现简单，但是在应用程序中会造成一些安全隐患。
- ❑ **Cookie**：实现简单，同样也能够简单的获取用户的信息，但是 **Cookie** 有大小的限制，不适宜存储大量的代码。

## 4.5 ASP.NET 页面生命周期

**ASP.NET** 页面运行时，也同类的对象一样，有自己的生命周期。**ASP.NET** 页面运行时，**ASP.NET** 页面将经历一个生命周期，在生命周期内，该页面将执行一系列的步骤，包括控件的初始化，控件的实例化，还原状态和维护状态等，以及通过 **IIS** 反馈给用户呈现成 **HTML**。

**ASP.NET** 页面生命周期是 **ASP.NET** 中非常重要的概念，了解 **ASP.NET** 页面的生命周期，就能够在合适的生命周期内编写代码，执行事务。同样，熟练掌握 **ASP.NET** 页面的生命周期，可以开发高效的自定义控件。**ASP.NET** 生命周期通常情况下需要经历几个阶段，这几个阶段如下所示。

- ❑ 页请求：页请求发生在页生命周期开始之前。当用户请求一个页面，**ASP.NET** 将确定是否需要分析或者编译该页面，或者是否可以在不运行页的情况下直接请求缓存响应客户端。
- ❑ 开始：发生了请求后，页面就进入了开始阶段。在该阶段，页面将确定请求是发回请求还是新的客户端请求，并设置 **IsPostBack** 属性。
- ❑ 初始化：在页面开始后，进入了初始化阶段。初始化期间，页面可以使用服务器控件，并为每个服务器控件进行初始化。
- ❑ 加载：页面加载控件。
- ❑ 验证：调用所有的验证程序控件的 **Validate** 方法，来设置各个验证程序控件和页的属性。
- ❑ 回发事件：如果是回发请求，则调用所有事件处理的程序。
- ❑ 呈现：在呈现期间，视图状态被保存并呈现到页。
- ❑ 卸载：完全呈现页面后，将页面发送到客户端并准备丢弃时，将调用卸载。

## 4.6 ASP.NET 生命周期中的事件

在页面周期的每个阶段，页面将引发可运行用户代码进行处理事件。对于控件产生的事件，通过声明的方式执行代码，并将事件处理程序绑定到事件。不仅如此，事件还支持自动事件连接，最常用的就是 **Page\_Load** 事件了，除了 **Page\_Load** 事件以外，还有 **Page\_Init** 等其他事件，本节将会介绍此类事件。

### 4.6.1 页面加载事件（Page\_PreInit）

每当页面被发送到服务器时，页面就会重新被加载，启动 **Page\_PreInit** 事件，执行 **Page\_PreInit** 事件代码块。当需要对页面中的控件进行初始化时，则需要使用此类事件，示例代码如下所示。

```
protected void Page_PreInit(object sender, EventArgs e)           //Page_PreInit 事件
{
    Label1.Text = "OK";                                           //标签赋值
}
```

在上述代码中，当触发了 **Page\_PreInit** 事件时，就会执行该事件的代码，上述代码将 **Label1** 的初始文本值设置为“OK”。**Page\_PreInit** 事件能够让用户在页面处理中，能够让服务器加载时只执行一次而当网页被返回给客户端时不被执行。在 **Page\_PreInit** 中可以使用 **IsPostBack** 来实现，当网页第一次加载时 **IsPostBack** 属性为 **false**，当页面再次被加载时，**IsPostBack** 属性将会被设置为 **true**。**IsPostBack** 属性的使用能够影响到应用程序的性能。

### 4.6.2 页面加载事件 (Page\_Init)

**Page\_Init** 事件与 **Page\_PreInit** 事件基本相同，区别在于 **Page\_Init** 并不能保证完全加载各个控件。虽然在 **Page\_Init** 事件中，依旧可以访问页面中的各个空间，但是当页面回送时，**Page\_Init** 依然执行所有的代码并且不能通过 **IsPostBack** 来执行某些代码，示例代码如下所示。

```
protected void Page_Init(object sender, EventArgs e)           //Page_Init 事件
{
    if (!IsPostBack)                                           //判断是否第一次加载
    {
        Label1.Text = "OK";                                    //将成功信息赋值给标签
    }
    else
    {
        Label1.Text = "IsPostBack";                            //将回传的值赋值给标签
    }
}
```

### 4.6.3 页面载入事件 (Page\_Load)

大多数初学者会认为 **Page\_Load** 事件是当页面第一次访问触发的事件，其实不然，在 **ASP.NET** 页生命周期内，**Page\_Load** 远远不是第一次触发的事件，通常情况下，**ASP.NET** 事件顺序如下所示。

- ☐ 1. **Page\_Init()**。
- ☐ 2. **Load ViewState**。
- ☐ 3. **Load Postback data**。
- ☐ 4. **Page\_Load()**。
- ☐ 5. **Handle control events**。
- ☐ 6. **Page\_PreRender()**。
- ☐ 7. **Page\_Render()**。
- ☐ 8. **Unload event**。
- ☐ 9. **Dispose method called**。

**Page\_Load** 事件是在网页加载的时候一定会被执行的事件。在 **Page\_Load** 事件中，一般都需要使用 **IsPostBack** 来判断用户是否进行了操作，因为 **IsPostBack** 指示该页是否正为响应客户端回发而加载，或者它是否正被首次加载和访问，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)           //Page_Load 事件
{
    if (!IsPostBack)                                           //第一次执行的代码块
    {
        Label1.Text = "OK";
    }
    else
    {
        Label1.Text = "IsPostBack";                            //如果用户提交表单等
    }
}
```

上述代码使用了 **Page\_Load** 事件，在页面被创建时，系统会自动在代码隐藏页模型的页面中增加此方法。当用户执行了操作，页面响应了客户端回发，则 **IsPostBack** 为 **true**，于是执行 **else** 中的操作。

#### 4.6.4 页面卸载事件 (Page\_Unload)

在页面被执行完毕后，可以通过 **Page\_Unload** 事件用来执行页面卸载时的清除工作，当页面被卸载时，执行此事件。以下情况会触发 **Page\_Unload** 事件。

- ☐ 页面被关闭。
- ☐ 数据库连接被关闭。
- ☐ 对象被关闭。
- ☐ 完成日志记录或者其他的程序请求。

#### 4.6.5 页面指令

页面指令用来通知编译器在编译页面时做出的特殊处理。当编译器处理 **ASP.NET** 应用程序时，可以通过这些特殊指令要求编译器做特殊处理，例如缓存、使用命名空间等。当需要执行页面指令时，通常的做法是将页面指令包括在文件的头部，示例代码如下所示。

```
<%@ Page
Language="C#" AutoEventWireup="true" CodeBehind="Default.aspx.cs" Inherits="MyWeb._Default" %>
<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

上述代码中，就使用了 **@Page** 页面指令来定义 **ASP.NET** 页面分析器和编译器使用的特定页的属性。当代码隐藏页模型的页面被创建时，系统会自动增加 **@Page** 页面指令。

**ASP.NET** 页面支持多个页面指令，常用的页面指令如下所示。

- ☐ **@ Page:** 定义 **ASP.NET** 页分析器和编译器使用的页特定（.aspx 文件）属性，可以编写为 `<%@ Page attribute="value" [attribute="value"...]%>`。
- ☐ **@ Control:** 定义 **ASP.NET** 页分析器和编译器使用的用户控件（.ascx 文件）特定的属性。该指令只能为用户控件配置。可以编写为 `<%@ Control attribute="value" [attribute="value"...]%>`。
- ☐ **@ Import:** 将命名空间显示导入到页中，使所导入的命名空间的所有类和接口可用用户该页。导入的命名空间可以是 **.NET Framework** 类库或用户定义的命名空间的一部分。可以编写为 `<%@ Import namespace="value" %>`。
- ☐ **@ Implements:** 提示当前页或用户控件实现制定的 **.NET Framework** 接口。可以编写为 `<%@ Implements interface="ValidInterfaceName" %>`。
- ☐ **@ Reference:** 以声明的方式指示，应该根据在其中声明此指令的页对另一个用户控件或页源文件进行动态编译和链接。可以编写为 `<%@ Reference page | control="pathToFile" %>`。
- ☐ **@ Output Cache:** 以声明的方式空间 **ASP.NET** 页或页中包含的用户控件的输出缓存策略。可以编写为 `<%@ Output Cache Duration="#ofseconds" Location="Any | Client | Downstream | Server | None" Shared="True | False" VaryByControl="controlname" VaryByCustom="browser | customstring" VaryByHeader="headers" VaryByParam="parametername" %>`
- ☐ **@ Assembly:** 在编译过程中将程序集链接到当前页，以使程序集的所有类和接口都可用在该页上。可以编写为 `<%@ Assembly Name="assemblyname" %>` 或 `<%@ Assembly Src="pathname" %>` 的方式。
- ☐ **@ Register:** 将别名与命名空间以及类名关联起来，以便在自定义服务器控件语法中使用简明的表示法。可以编写为 `<%@ Register tagprefix=" tagprefix" Namespace="namespace" Assembly="assembly" %>` 或 `<%@ Register tagprefix=" tagprefix" Tagname="tagname" Src="pathname" %>` 的方式。

4.7 ASP.NET 网站文件类型

在 ASP.NET 中包含诸多的文件类型，这些类型的文件由 ASP.NET 支持和管理，而除了这些文件以外，其他的文件都由 IIS 托管。使用 VS2008 能够创建大部分可以使用 ASP.NET 托管运行的程序。同时，使用应用程序映射可以将文件类型映射到应用程序。当需要伪静态时，很可能需要将.html 后缀托管到 IIS 中的应用扩展，因为默认情况下 ASP.NET 不会处理 HTML 的操作。

技巧：现在的网站构架中，生成静态是一种降低网站压力的一种很好的解决方案。在某些情况下，服务器可能需要伪静态支持，就是将.aspx 页面后缀显式成.html 后缀，让搜索引擎能够更好的搜录。

1. ASP.NET 管理的文件类型

ASP.NET 管理的文件类型能够在 ASP.NET 应用程序中被 ASP.NET 应用程序的不同模块进行访问和调用，这些文件可能是用户能够直接访问的，也有可能是用户无法直接访问的。ASP.NET 管理的文件类型如表 4-1 所示。

表 4-1 ASP.NET管理的文件类型

文件类型	保存位置	描述
.asax	根目录。	Global.asax 文件。包含 HttpApplication 对象的派生代码，用于重新展示 Application 对象。
.ascx	根目录或子目录。	可重用的自定义 Web 控件。
.ashx	根目录或子目录。	处理器文件。包含实现 IHttpHandler 接口的代码，用于处理输入请求。
.asmx	根目录或子目录。	XML Web Services 文件。包含由 SOAP 提供给其他 Web 应用的类对象和功能。
.aspx	根目录或子目录。	ASP.NET Web 窗体。包含 Web 控件和其他业务逻辑。
.axd	根目录。	跟踪视图文件。通常是 Trace.axd。
.browser	App_Browsers 目录。	浏览器定义文件。用于识别客户端浏览器的可用特征。
.cd	根目录或子目录。	类图文件。
.compile	Bin 目录。	定位于适当汇编集中的预编译文件。可执行文件（.aspx，.ascx，.master，theme）预编译后放在 Bin 目录。
.config	根目录或子目录。	Web.config 配置文件。包含用于配置 ASP.NET 若干特征的 XML 元素集。
.cs, .jsl, vb	App_Code 目录。有些是 ASP.NET 的代码分离文件，位于与 Web 页面相同的目录。	运行时被编译的类对象源代码。类对象可以是 HTTP 模块，HTTP 处理器，或 ASP.NET 页面的代码分离文件。
.csproj, vbproj, vjsproj	Visual Studio 工程目录。	Visual Studio 客户工程文件。
.disco, .vsdisco	App_WebReferences 目录。	XML Web Services Discovery 文件。用于定位可用 Web Services。
.dsdgm, dsprototype	根目录或子目录。	分布式服务图表（DSD）文件。可添加到 Visual Studio 方案中，为反向引擎提供消耗 Web Services 时的交互性图表。
.dll	Bin 目录。	已编译类库文件。作为替代，可将类对象源代码保存到 App_Code 目录。
.licx, .webinfo	根目录或子目录。	许可协议文件。许可协议有助于保护控件开发者的知识产权，并对控件用户的使用权进行验证。
.master	根目录或子目录。	模板文件定义 Web 页面的统一布局，并在其他页面中得到引用。
.mdb, .ldb	App_Data 目录。	Access 数据库文件。
.mdf	App_Data 目录。	SQLServer 数据库文件。
.msgx, .svc	根目录或子目录。	Indigo Messaging Framework（MFx）服务文件。
.rem	根目录或子目录。	远程处理器文件。
.resources	App_GlobalResources 或 App_LocalResources 目录。	资源文件。包含图像，本地化文本，或其他数据的资源引用串。
.resx	App_GlobalResources 或 App_LocalResources 目录。	资源文件。包含图像，本地化文本，或其他数据的资源引用串。
.sdm, .	根目录或子目录。	系统定义模型（SDM）文件。



sdmDocument		
.sitemap	根目录。	网站地图文件。包含网站的结构。ASP.NET 通过默认的网站地图提供者，简化导航控件对网站地图文件的使用。
.skin	App_Themes 目录。	皮肤定义文件。用于确定显示格式。
.sln	Visual Web Developer 工程目录。	Visual Web Developer 工程的项目文件。
.soap	根目录或子目录。	SOAP 扩展文件。

注意：ASP.NET 管理的文件类型映射到 IIS 的 Aspnet\_isapi.dll。

2. IIS 管理的文件类型

在 ASP.NET 应用程序中，有些动态的文件如 asp 文件就不被 ASP.NET 应用程序框架管理，这些文件由 IIS 进行管理，由 IIS 管理的文件类型如表 4-2 所示。

表 4-2 IIS管理的文件类型

文件类型	保存位置	描述
.asa	根目录。	Global.asa 文件。包含 ASP 会话对象或应用程序对象生命周期中的各种事件处理。
.asp	根目录或子目录。	ASP Web 页面。包含 @ 指令和使用 ASP 内建对象的脚本代码。
.cdx	App_Data 目录。	Visual FoxPro 的混合索引文件。
.cer	根目录或子目录。	证明文件。用于对网站的授权。
.idc	根目录或子目录。	Internet Database Connector (IDC) 文件。被映射到 httpodbc.dll。 注意：由于无法为数据库连接提供足够的安全性，IDC 将不再被继续使用。 IIS 6.0 是最后一个支持 IDC 的版本。
.shtm, .shtml, .stm	根目录或子目录。	包含文件。被映射到 ssinc.dll。

注意：IIS 管理的文件类型被映射到 IIS 的 asp.dll

3. 静态文件类型

IIS 仅提供已注册 MIME 类型的静态文件服务，注册信息保存在 Mime Map IIS 元数据库中。如果某种文件类型已经映射到指定应用程序，在不需要作为静态文件的情况之下，无需再在 MIME 类型列表中进行包含。默认的静态文件类型如表 4-3 所示。

表 4-3 静态文件类型

文件类型	保存位置	描述
.css	根目录或子目录，以及 App_Themes 目录。	样式表文件。用于确定 HTML 元素的显示格式。
.htm, .html	根目录或子目录。	静态网页文件。由 HTML 代码编写。

注意：虽然 ASP.NET 的代码页面也能够手动添加到 MIME 类型列表中，但是这样操作浏览者就能够看到页面源代码，从而暴露 ASP.NET 页面源代码，相对于服务器而言是非常不安全的。

4.8 小结

本章介绍了 ASP.NET 页面生命周期，以及 ASP.NET 页面的几种模型。ASP.NET 页面生命周期是 ASP.NET 中非常重要的概念，熟练掌握 ASP.NET 生命周期能对 ASP.NET 开发，自定义控件开发起到促进作用。本章还介绍了：

- ❑ 代码隐藏页模型的解释过程。
- ❑ 代码隐藏页模型的事件驱动处理。
- ❑ ASP.NET 网页的客户端状态。
- ❑ ASP.NET 页面生命周期。
- ❑ ASP.NET 生命周期中的事件。

□ **ASP.NET** 网站文件类型。

上面的章节都分开的讲解了 **ASP.NET** 运行中的一些基本机制，在了解了这些基本运行机制后，就能够在 **.NET** 框架下做 **ASP.NET** 开发了。虽然这些都是基本概念，但是在今后的开发中，会起到非常重要的作用。

## 第二篇 ASP.NET 窗体控件

第 5 章 Web 窗体的基本控件

第 6 章 Web 窗体的高级控件

第 5 章 Web 窗体的基本控件

与 ASP 不同的是，ASP.NET 提供了大量的控件，这些控件能够轻松的实现一个交互复杂的 Web 应用功能。在传统的 ASP 开发中，让开发人员最为烦恼的是代码的重用性太低，以及事件代码和页面代码不能很好的分开。而在 ASP.NET 中，控件不仅解决了代码重用性的问题，对于初学者而言，控件还简单易用并能够轻松上手、投入开发。

5.1 控件的属性

每个控件都有一些公共属性，例如字体颜色、边框的颜色、样式等。在 Visual Studio 2008 中，当开发人员将鼠标选择了相应的控件后，属性栏中会简单的介绍该属性的作用。如图 5-1 所示。

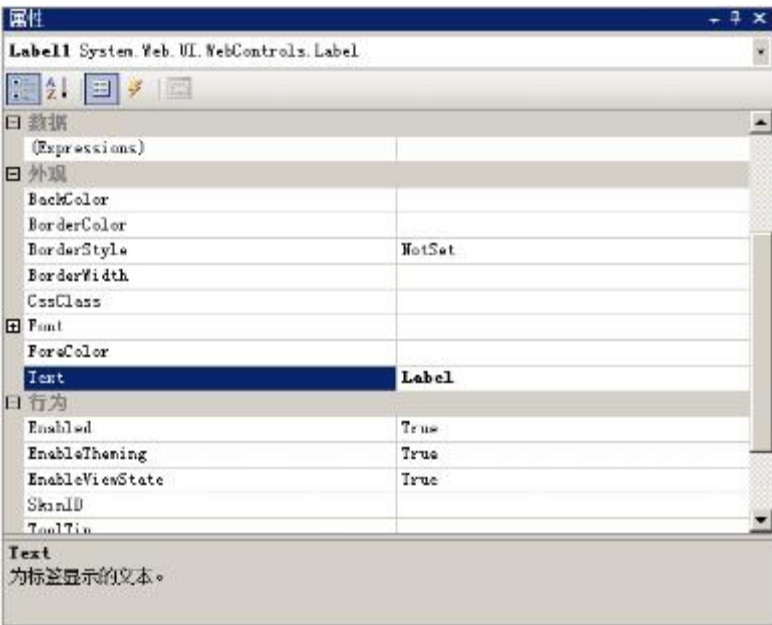


图 5-1 控件的属性

属性栏用来设置控件的属性，当控件在页面被初始化时，这些将被应用到控件。控件的属性也可以通过编程的方法在页面相应代码区域编写，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    Label1.Visible = false; //在 Page_Load 中设置 Label1 的可见性
}
```

上述代码编写了一个 Page\_Load（页面加载事件），当页面初次被加载时，会执行 Page\_Load 中的代码。这里通过编程的方法对控件的属性进行更改，当页面加载时，控件的属性会被应用并呈现在浏览器。

5.2 简单控件

ASP.NET 提供了诸多控件，这些控件包括简单控件、数据库控件、登录控件等强大的控件。在 ASP.NET 中，简单控件是最基础也是经常被使用的控件，简单控件包括标签控件（Label）、超链接控件（HyperLink）以及图像控件（Image）等。



### 5.2.1 标签控件（Label）

在 Web 应用中，希望显式的文本不能被用户更改，或者当触发事件时，某一段文本能够在运行时更改，则可以使用标签控件（Label）。开发人员可以非常方便的将标签控件拖放到页面，拖放到页面后，该页面将自动生成一段标签控件的声明代码，示例代码如下所示。

```
<asp:Label ID="Label1" runat="server" Text="Label"></asp:Label>
```

上述代码中，声明了一个标签控件，并将这个标签控件的 ID 属性设置为默认值 **Label1**。由于该控件是服务器端控件，所以在控件属性中包含 **runat="server"** 属性。该代码还将标签控件的文本初始化为 **Label**，开发人员能够配置该属性进行不同文本内容的呈现。

**注意：**通常情况下，控件的 ID 也应该遵循良好的命名规范，以便维护。

同样，标签控件的属性能够在相应的.cs 代码中初始化，示例代码如下所示。

```
protected void Page_PreInit(object sender, EventArgs e)
{
    Label1.Text = "Hello World"; //标签赋值
}
```

上述代码在页面初始化时为 **Label1** 的文本属性设置为“**Hello World**”。值得注意的是，对于 **Label** 标签，同样也可以显式 **HTML** 样式，示例代码如下所示。

```
protected void Page_PreInit(object sender, EventArgs e)
{
    Label1.Text = "Hello World<hr/><span style='color:red'>A Html Code</span>"; //输出 HTML
    Label1.Font.Size = FontUnit.XXLarge; //设置字体大小
}
```

上述代码中，**Label1** 的文本属性被设置为一串 **HTML** 代码，当 **Label** 文本被呈现时，会以 **HTML** 效果显式，运行结果如图 5-2 所示。



图 5-2 Label 的 Text 属性的使用

如果开发人员只是为了显示一般的文本或者 **HTML** 效果，不推荐使用 **Label** 控件，因为当服务器控件过多，会导致性能问题。使用静态的 **HTML** 文本能够让页面解析速度更快。

### 5.2.2 超链接控件（HyperLink）

超链接控件相当于实现了 **HTML** 代码中的“**<a href=""></a>**”效果，当然，超链接控件有自己的特点，当拖动一个超链接控件到页面时，系统会自动生成控件声明代码，示例代码如下所示。

```
<asp:HyperLink ID="HyperLink1" runat="server">HyperLink</asp:HyperLink>
```

上述代码声明了一个超链接控件，相对于 **HTML** 代码形式，超链接控件可以通过传递指定的参数来访问不同的页面。当触发了一个事件后，超链接的属性可以被改变。超链接控件通常使用的两个属性如下所

示：

- ❑ **ImageUrl**: 要显式图像的 URL。
- ❑ **NavigateUrl**: 要跳转的 URL。

## 1. ImageUrl 属性

设置 **ImageUrl** 属性可以设置这个超链接是以文本形式显式还是以图片文件显式，示例代码如下所示。

```
<asp:HyperLink ID="HyperLink1" runat="server"
    ImageUrl="http://www.shangducms.com/images/cms.jpg">
    HyperLink
</asp:HyperLink>
```

上述代码将文本形式显示的超链接变为了图片形式的超链接，虽然表现形式不同，但是不管是图片形式还是文本形式，全都实现的相同的效果。

## 2. Navigate 属性

**Navigate** 属性可以为无论是文本形式还是图片形式的超链接设置超链接属性，即即将跳转的页面，示例代码如下所示。

```
<asp:HyperLink ID="HyperLink1" runat="server"
    ImageUrl="http://www.shangducms.com/images/cms.jpg"
    NavigateUrl="http://www.shangducms.com">
    HyperLink
</asp:HyperLink>
```

上述代码使用了图片超链接的形式。其中图片来自 “<http://www.shangducms.com/images/cms.jpg>”，当点击此超链接控件后，浏览器将跳到 URL 为 “<http://www.shangducms.com>” 的页面。

## 3. 动态跳转

在前面的小结讲解了超链接控件的优点，超链接控件的优点在于能够对控件进行编程，来按照用户的意愿跳转到自己跳转的页面。以下代码实现了当用户选择 **QQ** 时，会跳转到腾讯网站，如果选择 **SOHU**，则会跳转到 **SOHU** 页面，示例代码如下所示。

```
protected void DropDownList1_SelectedIndexChanged(object sender, EventArgs e)
{
    if (DropDownList1.Text == "qq") //如果选择 qq
    {
        HyperLink1.Text = "qq"; //文本为 qq
        HyperLink1.NavigateUrl = "http://www.qq.com"; //URL 为 qq.com
    }
    else //选择 sohu
    {
        HyperLink1.Text = "sohu"; //文本为 sohu
        HyperLink1.NavigateUrl = "http://www.sohu.com"; //URL 为 sohu.com
    }
}
```

上述代码使用了 **DropDownList** 控件，当用户选择不同的值时，对 **HyperLink1** 控件进行操作。当用户选择 **qq**，则为 **HyperLink1** 控件配置连接为 <http://www.qq.com>。

注意：与标签控件相同的是，如果只是为了单纯的实现超链接，同样不推荐使用 **HyperLink** 控件，因为过多的使用服务器控件同样有可能造成性能问题。

## 5.2.3 图像控件（Image）

图像控件用来在 **Web** 窗体中显示图像，图像控件常用的属性如下：

- ❑ **AlternateText**: 在图像无法显式时显示的备用文本。

- ❑ **ImageAlign**: 图像的对齐方式。
- ❑ **ImageUrl**: 要显示图像的 URL。

当图片无法显示的时候，图片将被替换成 **AlternateText** 属性中的文字，**ImageAlign** 属性用来控制图片的对齐方式，而 **ImageUrl** 属性用来设置图像连接地址。同样，HTML 中也可以使用 `<img src="" alt="">` 来替代图像控件，图像控件具有可控性的优点，就是通过编程来控制图像控件，图像控件基本声明代码如下所示。

```
<asp:Image ID="Image1" runat="server" />
```

除了显示图形以外，**Image** 控件的其他属性还允许为图像指定各种文本，各属性如下所示。

- ❑ **ToolTip**: 浏览器显式在工具提示中的文本。
- ❑ **GenerateEmptyAlternateText**: 如果将此属性设置为 **true**，则呈现的图片的 **alt** 属性将设置为空。

开发人员能够为 **Image** 控件配置相应的属性以便在浏览时呈现不同的样式，创建一个 **Image** 控件也可以直接通过编写 HTML 代码进行呈现，示例代码如下所示。

```
<asp:Image ID="Image1" runat="server"
AlternateText="图片连接失效" ImageUrl="http://www.shangducms.com/images/cms.jpg" />
```

上述代码设置了一个图片，并当图片失效的时候提示图片连接失效。

**注意**：当双击图像控件时，系统并没有生成事件所需要的代码段，这说明 **Image** 控件不支持任何事件。

## 5.3 文本框控件（TextBox）

在 Web 开发中，Web 应用程序通常需要和用户进行交互，例如用户注册、登录、发帖等，那么就需要文本框控件（**TextBox**）来接受用户输入的信息。开发人员还可以使用文本框控件制作高级的文本编辑器用于 HTML，以及文本的输入输出。

### 5.3.1 文本框控件的属性

通常情况下，默认的文本控件（**TextBox**）是一个单行的文本框，用户只能在文本框中输入一行内容。通过修改该属性，则可以将文本框设置为多行/或者是以密码形式显示，文本框控件常用的控件属性如下所示。

- ❑ **AutoPostBack**: 在文本修改以后，是否自动重传
- ❑ **Columns**: 文本框的宽度。
- ❑ **EnableViewState**: 控件是否自动保存其状态以用于往返过程。
- ❑ **MaxLength**: 用户输入的最大字符数。
- ❑ **ReadOnly**: 是否为只读。
- ❑ **Rows**: 作为多行文本框时所显式的行数。
- ❑ **TextMode**: 文本框的模式，设置单行，多行或者密码。
- ❑ **Wrap**: 文本框是否换行。

#### 1. AutoPostBack（自动回传）属性

在网页的交互中，如果用户提交了表单，或者执行了相应的方法，那么该页面将会发送到服务器上，服务器将执行表单的操作或者执行相应方法后，再呈现给用户，例如按钮控件、下拉菜单控件等。如果将某个控件的 **AutoPostBack** 属性设置为 **true** 时，则如果该控件的属性被修改，那么同样会使页面自动发回到服务器。

#### 2. EnableViewState（控件状态）属性

**ViewState** 是 ASP.NET 中用来保存 Web 控件回传状态的一种机制，它是由 ASP.NET 页面框架管理的一个隐藏字段。在回传发生时，**ViewState** 数据同样将回传到服务器，ASP.NET 框架解析 **ViewState** 字符串并

为页面中的各个控件填充该属性。而填充后，控件通过使用 **ViewState** 将数据重新恢复到以前的状态。

在使用某些特殊的控件时，如数据库控件，来显示数据库。每次打开页面执行一次数据库往返过程是非常不明智的。开发人员可以绑定数据，在加载页面时仅对页面设置一次，在后续的回传中，控件将自动从 **ViewState** 中重新填充，减少了数据库的往返次数，从而不使用过多的服务器资源。在默认情况下，**EnableViewState** 的属性值通常为 **true**。

## 3. 其他属性

上面的两个属性是比较重要的属性，其他的属性也经常使用。

- ❑ **MaxLength**: 在注册时可以限制用户输入的字符串长度。
- ❑ **ReadOnly**: 如果将此属性设置为 **true**，那么文本框内的值是无法被修改的。
- ❑ **TextMode**: 此属性可以设置文本框的模式，例如单行、多行和密码形式。默认情况下，不设置 **TextMode** 属性，那么文本框默认为单行。

## 5.3.2 文本框控件的使用

在默认情况下，文本框为单行类型，同时文本框模式也包括多行和密码，示例代码如下所示。

```
<asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
<br />
<br />
<asp:TextBox ID="TextBox2" runat="server" Height="101px" TextMode="MultiLine"
    Width="325px"></asp:TextBox>
<br />
<br />
<asp:TextBox ID="TextBox3" runat="server" TextMode="Password"></asp:TextBox>
```

上述代码演示了三种文本框的使用方法，上述代码运行后的结果如图 5-3 所示。

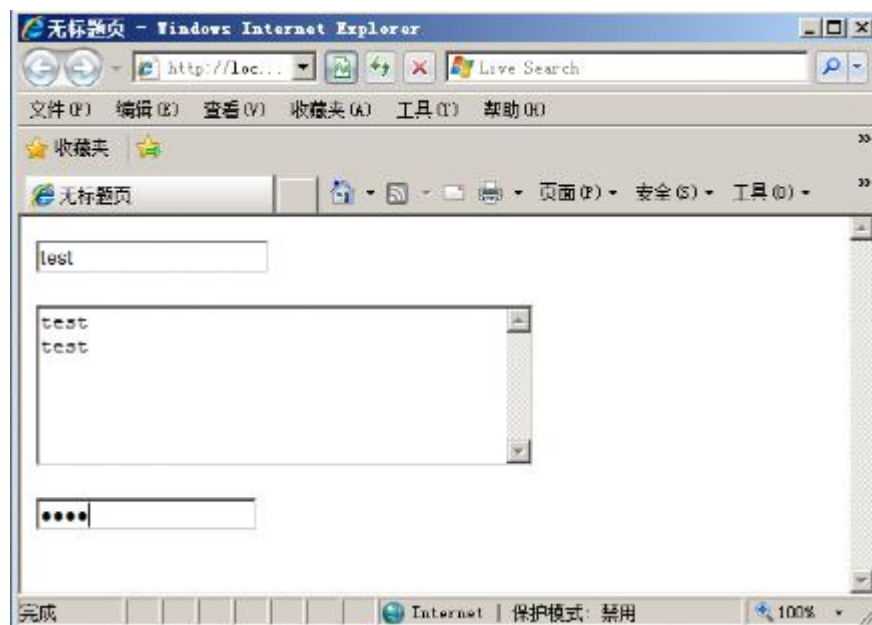


图 5-3 文本框的三种形式

文本框无论是在 **Web** 应用程序开发还是 **Windows** 应用程序开发中都是非常重要的。文本框在用户交互中能够起到非常重要的作用。在文本框的使用中，通常需要获取用户在文本框中输入的值或者检查文本框属性是否被改写。当获取用户的值的时候，必须通过一段代码来控制。文本框控件 **HTML** 页面示例代码如下所示。

```
<form id="form1" runat="server">
<div>
    <asp:Label ID="Label1" runat="server" Text="Label"></asp:Label>
    <br />
    <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
    <br />
    <asp:Button ID="Button1" runat="server" onclick="Button1_Click" Text="Button" />
</div>
</form>
```



```
<br />
</div>
</form>
```

上述代码声明了一个文本框控件和一个按钮控件，当用户单击按钮控件时，就需要实现标签控件的文本改变。为了实现相应的效果，可以通过编写 **cs** 文件代码进行逻辑处理，示例代码如下所示：

```
namespace _5_3                                //页面命名空间
{
    public partial class _Default : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)    //页面加载时触发
        {
        }
        protected void Button1_Click(object sender, EventArgs e) //双击按钮时触发的事件
        {
            Label1.Text = TextBox1.Text;                        //标签控件的值等于文本框中控件的值
        }
    }
}
```

上述代码中，当双击按钮时，就会触发一个按钮事件，这个事件就是将文本框内的值赋值到标签内，运行结果如图 5-4 所示。

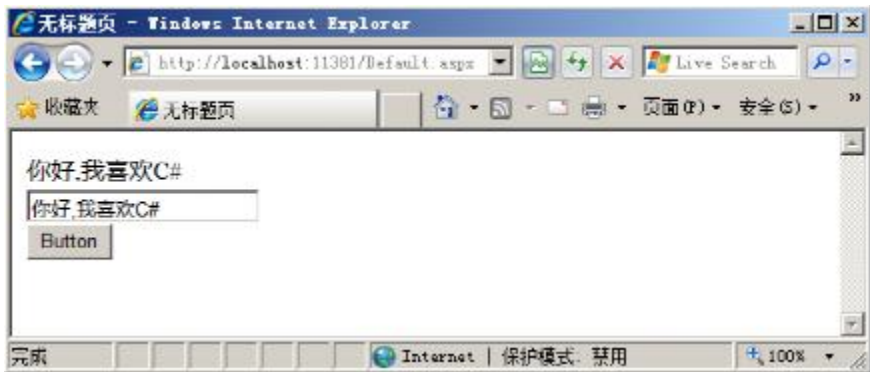


图 5-4 文本框控件的使用

同样，双击文本框控件，会触发 **TextChanged** 事件。而当运行时，当文本框控件中的字符变化后，并没有自动回传，是因为默认情况下，文本框的 **AutoPostBack** 属性被设置为 **false**。当 **AutoPostBack** 属性被设置为 **true** 时，文本框的属性变化，则会发生回传，示例代码如下所示。

```
protected void TextBox1_TextChanged(object sender, EventArgs e)    //文本框事件
{
    Label1.Text = TextBox1.Text;                                    //控件相互赋值
}
```

上述代码中，为 **TextBox1** 添加了 **TextChanged** 事件。在 **TextChanged** 事件中，并不是每一次文本框的内容发生了变化之后，就会重传到服务器，这一点和 **WinForm** 是不同的，因为这样会大大的降低页面的效率。而当用户将文本框中的焦点移出导致 **TextBox** 就会失去焦点时，才会发生重传。

## 5.4 按钮控件（Button，LinkButton，ImageButton）

在 **Web** 应用程序和用户交互时，常常需要提交表单、获取表单信息等操作。在这其间，按钮控件是非常必要的。按钮控件能够触发事件，或者将网页中的信息回传给服务器。在 **ASP.NET** 中，包含三类按钮控件，分别为 **Button**、**LinkButton**、**ImageButton**。

### 5.4.1 按钮控件的通用属性

按钮控件用于事件的提交，按钮控件包含一些通用属性，按钮控件的常用通用属性包括有：

- ❑ **Causes Validation:** 按钮是否导致激发验证检查。
- ❑ **CommandArgument:** 与此按钮管理的命令参数。
- ❑ **CommandName:** 与此按钮关联的命令。
- ❑ **ValidationGroup:** 使用该属性可以指定单击按钮时调用页面上的哪些验证程序。如果未建立任何验证组，则会调用页面上的所有验证程序。

下面的语句声明了三种按钮，示例代码如下所示。

```
<asp:Button ID="Button1" runat="server" Text="Button" />           //普通的按钮
<br />
<asp:LinkButton ID="LinkButton1" runat="server">LinkButton</asp:LinkButton> //Link 类型的按钮
<br />
<asp:ImageButton ID="ImageButton1" runat="server" />           //图像类型的按钮
```

对于三种按钮，他们起到的作用基本相同，主要是表现形式不同，如图 5-5 所示。



图 5-5 三种按钮类型

### 5.4.2 Click 单击事件

这三种按钮控件对应的事件通常是 **Click** 单击和 **Command** 命令事件。在 **Click** 单击事件中，通常用于编写用户单击按钮时所需要执行的事件，示例代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    Label1.Text = "普通按钮被触发";           //输出信息
}
protected void LinkButton1_Click(object sender, EventArgs e)
{
    Label1.Text = "连接按钮被触发";           //输出信息
}
protected void ImageButton1_Click(object sender, ImageClickEventArgs e)
{
    Label1.Text = "图片按钮被触发";           //输出信息
}
```

上述代码分别为三种按钮生成了事件，其代码都是将 **Label1** 的文本设置为相应的文本，运行结果如图 5-6 所示。

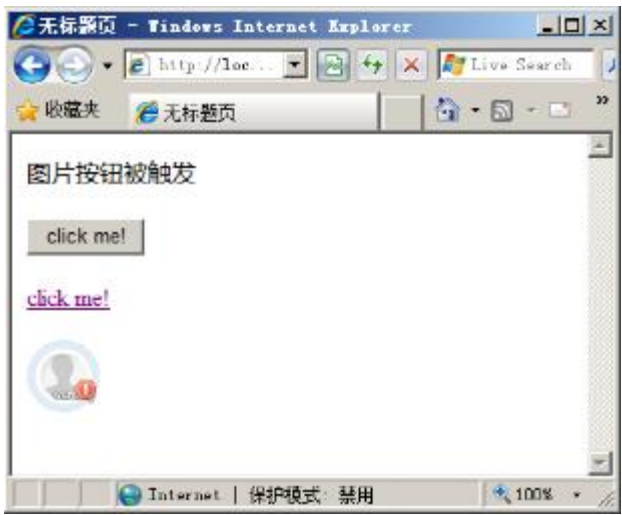


图 5-6 按钮的 Click 事件

5.4.3 Command 命令事件

按钮控件中，Click 事件并不能传递参数，所以处理的事件相对简单。而 Command 事件可以传递参数，负责传递参数的是按钮控件的 CommandArgument 和 CommandName 属性。如图 5-7 所示。

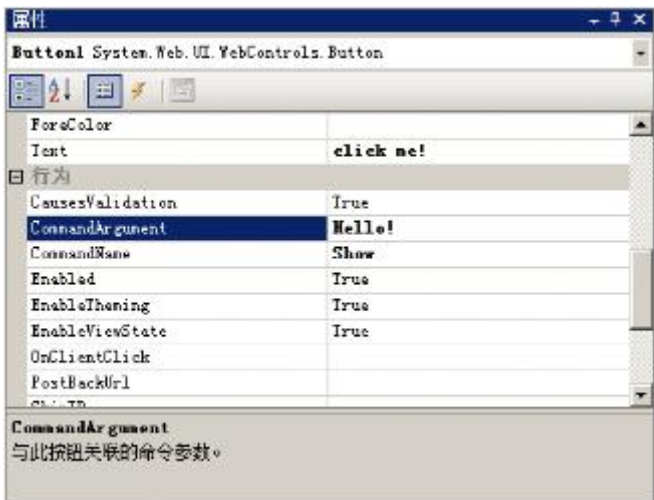



图 5-7 CommandArgument 和 CommandName 属性

将 CommandArgument 和 CommandName 属性分别设置为 Hello!和 Show，单击  创建一个 Command 事件并在事件中编写相应代码，示例代码如下所示。

```
protected void Button1_Command(object sender, CommandEventArgs e)
{
    if (e.CommandName == "Show")    //如果 CommandName 属性的值为 Show，则运行下面代码
    {
        Label1.Text = e.CommandArgument.ToString();//CommandArgument 属性的值赋值给 Label1
    }
}
```

注意：当按钮同时包含 Click 和 Command 事件时，通常情况下会执行 Command 事件。

Command 有一些 Click 不具备的好处，就是传递参数。可以对按钮的 CommandArgument 和 CommandName 属性分别设置，通过判断 CommandArgument 和 CommandName 属性来执行相应的方法。这样一个按钮控件就能够实现不同的方法，使得多个按钮与一个处理代码关联或者一个按钮根据不同的值进行不同的处理和响应。相比 Click 单击事件而言，Command 命令事件具有更高的可控性。

## 5.5 单选控件和单选组控件（RadioButton 和 RadioButtonList）

在投票等系统中，通常需要使用单选控件和单选组控件。顾名思义，在单选控件和单选组控件的项目中，只能在有限种选择中进行一个项目的选择。在进行投票等应用开发并且只能在选项中选择单项时，单选控件和单选组控件都是最佳的选择。

### 5.5.1 单选控件（RadioButton）

单选控件可以为用户选择某一个选项，单选控件常用属性如下所示。

- ❑ **Checked**: 控件是否被选中。
- ❑ **GroupName**: 单选控件所处的组名。
- ❑ **TextAlign**: 文本标签相对于控件的对齐方式。

单选控件通常需要 **Checked** 属性来判断某个选项是否被选中，多个单选控件之间可能存在着某些联系，这些联系通过 **GroupName** 进行约束和联系，示例代码如下所示。

```
<asp:RadioButton ID="RadioButton1" runat="server" GroupName="choose"
    Text="Choose1" />
<asp:RadioButton ID="RadioButton2" runat="server" GroupName="choose"
    Text="Choose2" />
```

上述代码声明了两个单选控件，并将 **GroupName** 属性都设置为“**choose**”。单选控件中最常用的事件是 **CheckedChanged**，当控件的选中状态改变时，则触发该事件，示例代码如下所示。

```
protected void RadioButton1_CheckedChanged(object sender, EventArgs e)
{
    Label1.Text = "第一个被选中";
}
protected void RadioButton2_CheckedChanged(object sender, EventArgs e)
{
    Label1.Text = "第二个被选中";
}
```

上述代码中，当选中状态被改变时，则触发相应的事件。运行结果如图 5-8 所示。

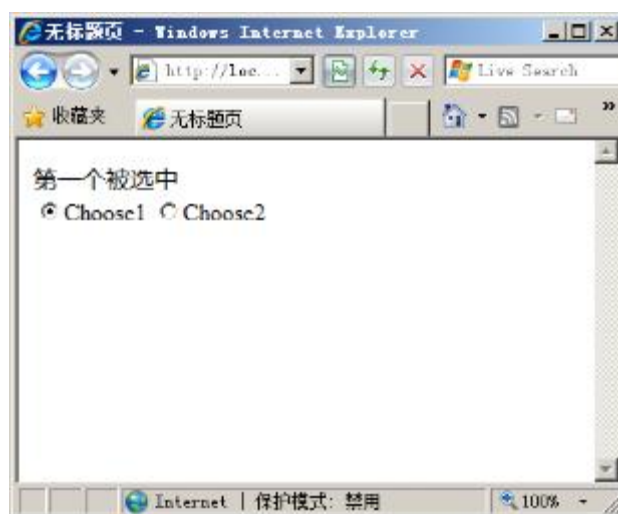


图 5-8 单选控件的使用

与 **TextBox** 文本框控件相同的是，单选控件不会自动进行页面回传，必须将 **AutoPostBack** 属性设置为 **true** 时才能在焦点丢失时触发相应的 **CheckedChanged** 事件。

### 5.5.2 单选组控件（RadioButtonList）

与单选控件相同，单选组控件也是只能选择一个项目的控件，而与单选控件不同的是，单选组控件没



有 **GroupName** 属性，但是却能够列出多个单选项目。另外，单选组控件所生成的代码也比单选控件实现的相对较少。单选组控件添加项如图 5-9 所示。



图 5-9 单选组控件添加项

添加项目后，系统自动在.aspx 页面声明服务器控件代码，代码如下所示。

```
<asp:RadioButtonList ID="RadioButtonList1" runat="server">
  <asp:ListItem>Choose1</asp:ListItem>
  <asp:ListItem>Choose2</asp:ListItem>
  <asp:ListItem>Choose3</asp:ListItem>
</asp:RadioButtonList>
```

上述代码使用了单选组控件进行单选功能的实现，单选组控件还包括一些属性用于样式和重复的配置。单选组控件的常用属性如下所示：

- ❑ **DataMember:** 在数据集用做数据源时做数据绑定。
- ❑ **DataSource:** 向列表填入项时所使用的数据源。
- ❑ **DataTextField:** 提供项文本的数据源中的字段。
- ❑ **DataTextFormat:** 应用于文本字段的格式。
- ❑ **DataValueField:** 数据源中提供项值的字段。
- ❑ **Items:** 列表中项的集合。
- ❑ **RepeatColumn:** 用于布局项的列数。
- ❑ **RepeatDirection:** 项的布局方向。
- ❑ **RepeatLayout:** 是否在某个表或者流中重复。

同单选控件一样，双击单选组控件时系统会自动生成该事件的声明，同样可以在该事件中确定代码。当选择一项内容时，提示用户所选择的内容，示例代码如下所示。

```
protected void RadioButtonList1_SelectedIndexChanged(object sender, EventArgs e)
{
    Label1.Text = RadioButtonList1.Text; //文本标签段的值等于选择的控件的值
}
```

5.6 复选框控件和复选组控件（CheckBox 和 CheckBoxList）

当一个投票系统需要用户能够选择多个选择项时，则单选框控件就不符合要求了。ASP.NET 还提供了复选框控件和复选组控件来满足多选的要求。复选框控件和复选组控件同单选框控件和单选组控件一样，都是通过 **Checked** 属性来判断是否被选择。

5.6.1 复选框控件（CheckBox）

同单选框控件一样，复选框也是通过 **Check** 属性判断是否被选择，而不同的是，复选框控件没有

**GroupName** 属性，示例代码如下所示。

```
<asp:CheckBox ID="CheckBox1" runat="server" Text="Check1" AutoPostBack="true" />
<asp:CheckBox ID="CheckBox2" runat="server" Text="Check2" AutoPostBack="true"/>
```

上述代码中声明了两个复选框控件。对于复选框空间，并没有支持的 **GroupName** 属性，当双击复选框控件时，系统会自动生成方法。当复选框控件的选中状态被改变后，会激发该事件。示例代码如下所示。

```
protected void CheckBox1_CheckedChanged(object sender, EventArgs e)
{
    Label1.Text = "选框 1 被选中";           //当选框 1 被选中时
}
protected void CheckBox2_CheckedChanged(object sender, EventArgs e)
{
    Label1.Text = "选框 2 被选中,并且字体变大";           //当选框 2 被选中时
    Label1.Font.Size = FontUnit.XXLarge;
}
```

上述代码分别为两个选框设置了事件，设置了当选择选框 **1** 时，则文本标签输出“选框 **1** 被选中”，如图 **5-10** 所示。当选择选框 **2** 时，则输出“选框 **2** 被选中，并且字体变大”，运行结果如图 **5-11** 所示。



图 5-10 选框 1 被选中

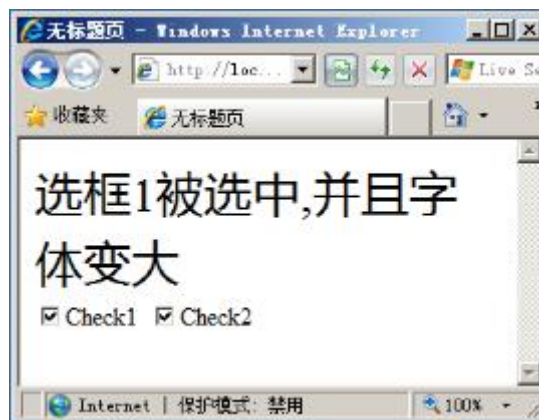


图 5-11 选框 2 被选中

对于复选框而言，用户可以在复选框控件中选择多个选项，所以就没有必要为复选框控件进行分组。在单选框控件中，相同组名的控件只能选择一项用于约束多个单选框中的选项，而复选框就没有约束的必要。

## 5.6.2 复选组控件（CheckBoxList）

同单选组控件相同，为了方便复选控件的使用，**.NET** 服务器控件中同样包括了复选组控件，拖动一个复选组控件到页面可以同单选组控件一样添加复选组列表。添加在页面后，系统生成代码如下所示。

```
<asp:CheckBoxList ID="CheckBoxList1" runat="server" AutoPostBack="True"
onselectedindexchanged="CheckBoxList1_SelectedIndexChanged">
    <asp:ListItem Value="Choose1">Choose1</asp:ListItem>
    <asp:ListItem Value="Choose2">Choose2</asp:ListItem>
    <asp:ListItem Value="Choose3">Choose3</asp:ListItem>
</asp:CheckBoxList>
```

上述代码中，同样增加了 **3** 个项目提供给用户选择，复选组控件最常用的是 **SelectedIndexChanged** 事件。当控件中某项的选中状态被改变时，则会触发该事件。示例代码如下所示。

```
protected void CheckBoxList1_SelectedIndexChanged(object sender, EventArgs e)
{
    if (CheckBoxList1.Items[0].Selected)           //判断某项是否被选中
    {
        Label1.Font.Size = FontUnit.XXLarge;       //更改字体大小
    }
    if (CheckBoxList1.Items[1].Selected)           //判断是否被选中
    {
```

```

        Label1.Font.Size = FontUnit.XLarge;           //更改字体大小
    }
    if (CheckBoxList1.Items[2].Selected)
    {
        Label1.Font.Size = FontUnit.XSmall;
    }
}

```

上述代码中，**CheckBoxList1.Items[0].Selected** 是用来判断某项是否被选中，其中 **Item** 数组是复选组控件中项目的集合，其中 **Items[0]** 是复选组中的第一个项目。上述代码用来修改字体的大小，如图 5-12 所示，当选择不同的选项时，字体的大小也不相同，运行结果如图 5-13 所示。



图 5-12 选择大号字体



图 5-13 选择小号字体

正如图 5-12、5-13 所示，当用户选择不同的选项时，**Label** 标签的字体的大小会随之改变。

注意：复选组控件与单选组控件不同的是，不能够直接获取复选组控件某个选中项目的值，因为复选组控件返回的是第一个选择项的返回值，只能够通过 **Item** 集合来获取选择某个或多个选中的项目值。

## 5.7 列表控件（DropDownList, ListBox 和 BulletedList）

在 **Web** 开发中，经常会需要使用列表控件，让用户的输入更加简单。例如在用户注册时，用户的所在地是有限的集合，而且用户不喜欢经常键入，这样就可以使用列表控件。同样列表控件还能够简化用户输入并且防止用户输入在实际中不存在的数据，如性别的选择等。

### 5.7.1 DropDownList 列表控件

列表控件能在一个控件中为用户提供多个选项，同时又能够避免用户输入错误的选项。例如，在用户注册时，可以选择性别是男，或者女，就可以使用 **DropDownList** 列表控件，同时又避免了用户输入其他的信息。因为性别除了男就是女，输入其他的信息说明这个信息是错误或者是无效的。下列语句声明了一个 **DropDownList** 列表控件，示例代码如下所示。

```

<asp:DropDownList ID="DropDownList1" runat="server">
    <asp:ListItem>1</asp:ListItem>
    <asp:ListItem>2</asp:ListItem>
    <asp:ListItem>3</asp:ListItem>
    <asp:ListItem>4</asp:ListItem>
    <asp:ListItem>5</asp:ListItem>
    <asp:ListItem>6</asp:ListItem>
    <asp:ListItem>7</asp:ListItem>
</asp:DropDownList>

```

上述代码创建了一个 **DropDownList** 列表控件，并手动增加了列表项。同时 **DropDownList** 列表控件也



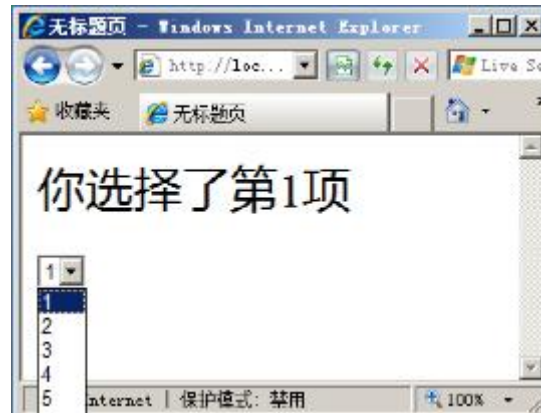
可以绑定数据源控件。**DropDownList** 列表控件最常用的事件是 **SelectedIndexChanged**，当 **DropDownList** 列表控件选择项发生变化时，则会触发该事件，示例代码如下所示。

```
protected void DropDownList1_SelectedIndexChanged1(object sender, EventArgs e)
{
    Label1.Text = "你选择了第" + DropDownList1.Text + "项";
}
```

上述代码中，当选择的项目发生变化时则会触发该事件，如图 5-14 所示。当用户再次进行选择时，系统会将更改标签 1 中的文本，如图 5-15 所示。



图 5-14 选择第三项



5-15 选择第一项

当用户选择相应的项目时，就会触发 **SelectedIndexChanged** 事件，开发人员可以通过捕捉相应的用户选中的控件进行编程处理，这里就捕捉了用户选择的数字进行字体大小的更改。

## 5.7.2 ListBox 列表控件

相对于 **DropDownList** 控件而言，**ListBox** 控件可以指定用户是否允许多项选择。设置 **SelectionMode** 属性为 **Single** 时，表明只允许用户从列表框中选择一个项目，而当 **SelectionMode** 属性的值为 **Multiple** 时，用户可以按住 **Ctrl** 键或者使用 **Shift** 组合键从列表中选择多个数据项。当创建一个 **ListBox** 列表控件后，开发人员能够在控件中添加所需的项目，添加完成后示例代码如下所示。

```
<asp:ListBox ID="ListBox1" runat="server" Width="137px" AutoPostBack="True">
    <asp:ListItem>1</asp:ListItem>
    <asp:ListItem>2</asp:ListItem>
    <asp:ListItem>3</asp:ListItem>
    <asp:ListItem>4</asp:ListItem>
    <asp:ListItem>5</asp:ListItem>
    <asp:ListItem>6</asp:ListItem>
</asp:ListBox>
```

从结构上看，**ListBox** 列表控件的 **HTML** 样式代码和 **DropDownList** 控件十分相似。同样，**SelectedIndexChanged** 也是 **ListBox** 列表控件中最常用的事件，双击 **ListBox** 列表控件，系统会自动生成相应的代码。同样，开发人员可以为 **ListBox** 控件中的选项改变后的事件做编程处理，示例代码如下所示。

```
protected void ListBox1_SelectedIndexChanged(object sender, EventArgs e)
{
    Label1.Text = "你选择了第" + ListBox1.Text + "项";
}
```

上述代码中，当 **ListBox** 控件选择项发生改变后，该事件就会被触发并修改相应 **Label** 标签中文本，如图 5-16 所示。

上面的程序同样实现了 **DropDownList** 中程序的效果。不同的是，如果需要使用户选择多个 **ListBox** 项，只需要设置 **SelectionMode** 属性为 “**Multiple**” 即可，如图 5-17 所示。





图 5-16 ListBox 单选

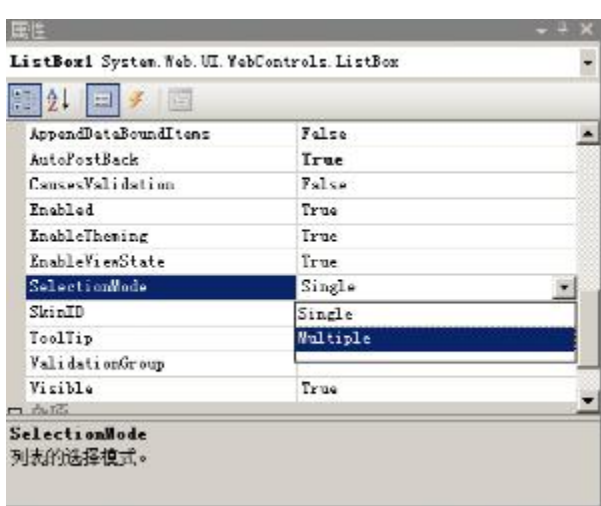


图 5-17 SelectionMode 属性

当设置了 **SelectionMode** 属性后，用户可以按住 **Ctrl** 键或者使用 **Shift** 组合键选择多项。同样，开发人员也可以编写处理选择多项时的事件，示例代码如下所示。

```
protected void ListBox1_SelectedIndexChanged1(object sender, EventArgs e)
{
    Label1.Text += ",你选择了第" + ListBox1.Text + "项";
}
```

上述代码使用了“+=”运算符，在触发 **SelectedIndexChanged** 事件后，应用程序将为 **Label1** 标签赋值，如图 5-18 所示。当用户每选一项的时候，就会触发该事件，如图 5-19 所示。

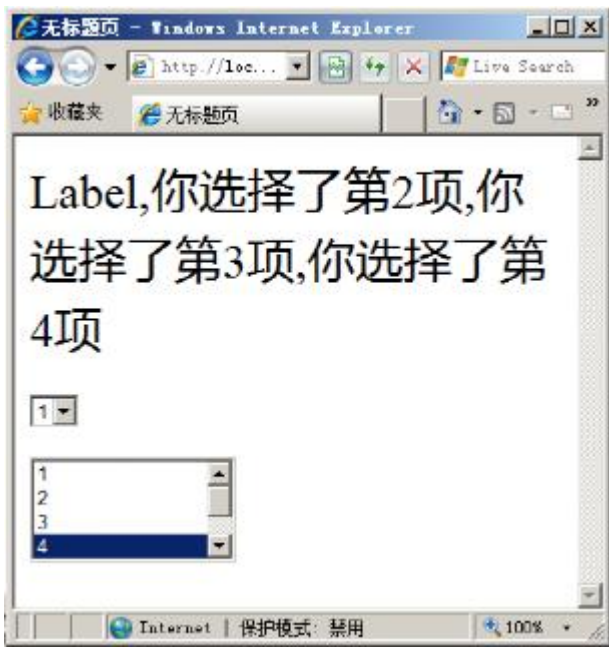


图 5-18 单选效果

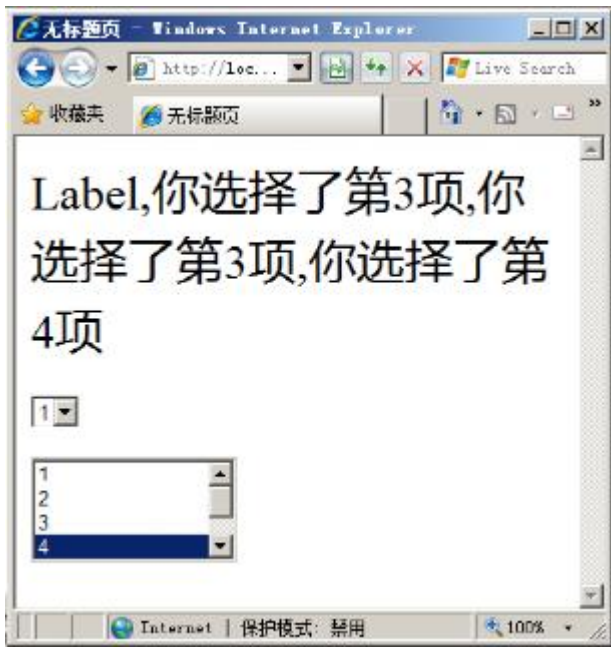


图 5-19 多选效果

从运行结果可以看出，当单选时，选择项返回值和选择的项相同，而当选择多项的时候，返回值同第一项相同。所以，在选择多项时，也需要使用 **Item** 集合获取和遍历多个项目。

5.7.3 BulletedList 列表控件

**BulletedList** 与上述列表控件不同的是，**BulletedList** 控件可呈现项目符号或编号。对 **BulletedList** 属性的设置为呈现项目符号，则当 **BulletedList** 被呈现在页面时，列表前端会则会显式项目符号或者特殊符号，效果如图 5-20 所示。

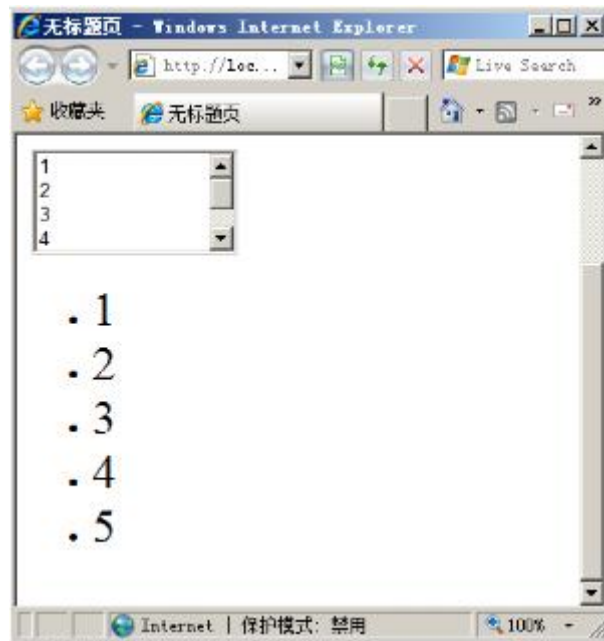


图 5-20 BulletedList 显示效果

**BulletedList** 可以通过设置 **BulletStyle** 属性来编辑列表前的符号样式，常用的 **BulletStyle** 项目符号编号样式如下所示。

- ☐ **Circle**: 项目符号设置为○。
- ☐ **CustomImage**: 项目符号为自定义图片。
- ☐ **Disc**: 项目符号设置为●。
- ☐ **LowerAlpha**: 项目符号为小写字母格式，如 a、b、c 等。
- ☐ **LowerRoman**: 项目符号为罗马数字格式，如 i、ii 等。
- ☐ **NotSet**: 表示不设置，此时将以 **Disc** 样式为默认样式。
- ☐ **Numbered**: 项目符号为 1、2、3、4 等。
- ☐ **Square**: 项目符号为黑方块■。
- ☐ **UpperAlpha**: 项目符号为大写字母格式，如 A、B、C 等。
- ☐ **UpperRoman**: 项目符号为大写罗马数字格式如 I、II、III 等。

同样，**BulletedList** 控件也同 **DropDownList** 以及 **ListBox** 相同，可以添加事件。不同的是生成的事件是 **Click** 事件，代码如下所示。

```
protected void BulletedList1_Click(object sender, BulletedListEventArgs e)
{
    Label1.Text += ",你选择了第" + BulletedList1.Items[e.Index].ToString() + "项";
}
```

**DropDownList** 和 **ListBox** 生成的事件是 **SelectedIndexChanged**，当其中的选择项被改变时，则触发该事件。而 **BulletedList** 控件生成的事件是 **Click**，用于在其中提供逻辑以执行特定的应用程序任务。

## 5.8 面板控件（Panel）

面板控件就好像是一些控件的容器，可以将一些控件包含在面板控件内，然后对面板控制进行操作来设置在面板控件内的所有控件是显示还是隐藏，从而达到设计者的特殊目的。当创建一个面板控件时，系统会生成相应的 **HTML** 代码，示例代码如下所示。

```
<asp:Panel ID="Panel1" runat="server">
</asp:Panel>
```

面板控件的常用功能就是显示或隐藏一组控件，示例 **HTML** 代码如下所示。

```
<form id="form1" runat="server">
    <asp:Button ID="Button1" runat="server" Text="Show" />
    <asp:Panel ID="Panel1" runat="server" Visible="False">
        <asp:Label ID="Label1" runat="server" Text="Name:" style="font-size: xx-large"></asp:Label>
```

```
<asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
<br />
This is a Panel!
</asp:Panel>
</form>
```

上述代码创建了一个 **Panel** 控件， **Panel** 控件默认属性为隐藏，并在控件外创建了一个 **Button** 控件 **Button1**，当用户单击外部的按钮控件后将显示 **Panel** 控件，cs 代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    Panel1.Visible = true;                //Panel 控件显示可见
}
```

当页面初次被载入时， **Panel** 控件以及 **Panel** 控件内部的服务器控件都为隐藏，如图 5-21 所示。当用户单击 **Button1** 时，则 **Panel** 控件可见性为可见，则页面中的 **Panel** 控件以及 **Panel** 控件中的所有服务器控件也都为可见，如图 5-22 所示。

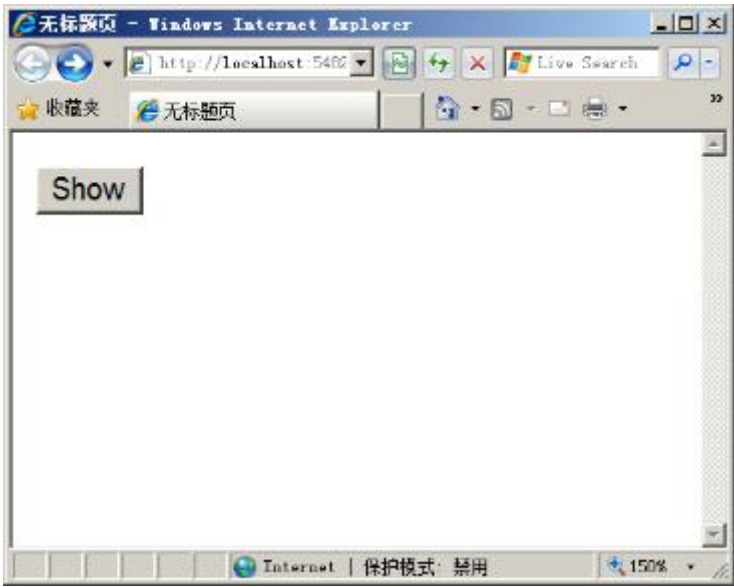


图 5-21 Panel 控件隐藏

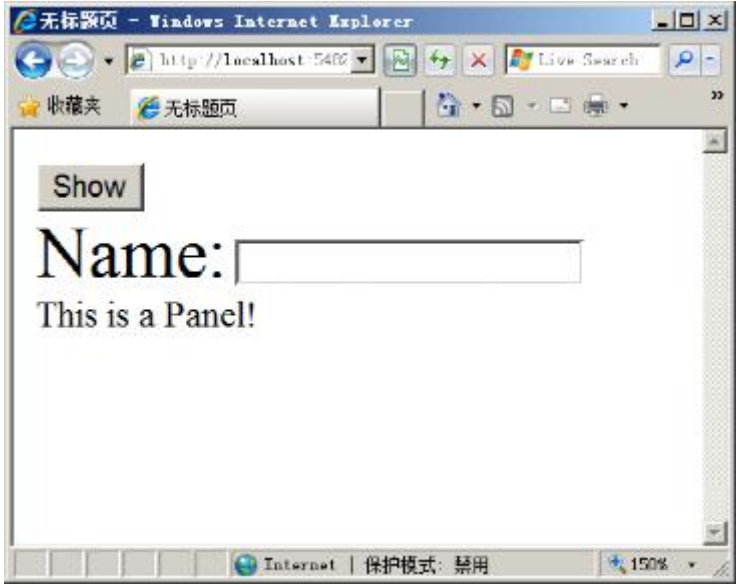


图 5-22 Panel 被显示

将 **TextBox** 控件和 **Button** 控件放到 **Panel** 控件中，可以为 **Panel** 控件的 **DefaultButton** 属性设置为面板中某个按钮的 **ID** 来定义一个默认的按钮。当用户在面板中输入完毕，可以直接按 **Enter** 键来传送表单。并且，当设置了 **Panel** 控件的高度和宽度时，当 **Panel** 控件中的内容高度或宽度超过时，还能够自动出现滚动条。

**Panel** 控件还包含一个 **GroupText** 属性，当 **Panel** 控件的 **GroupText** 属性被设置时， **Panel** 将会被创建一个带标题的分组框，效果如图 5-23 所示。

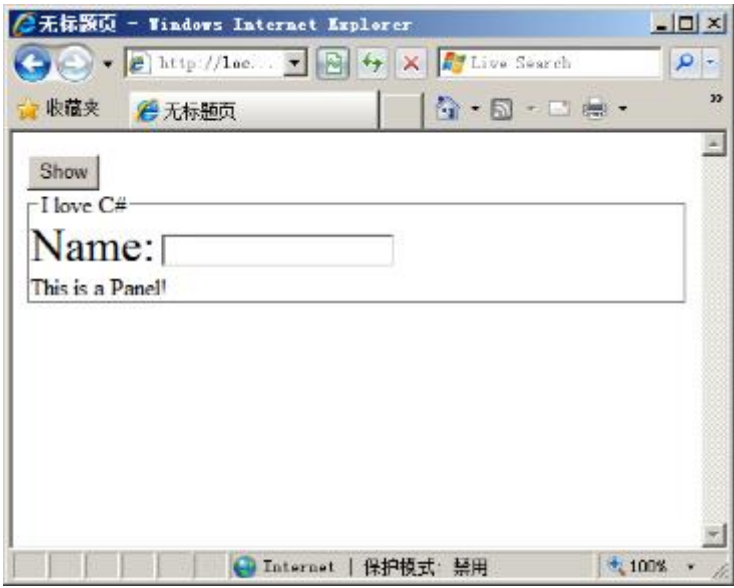


图 5-23 Panel 控件的 GroupText 属性

**GroupText** 属性能够进行 **Panel** 控件的样式呈现，通过编写 **GroupText** 属性能够更加清晰的让用户了解 **Panel** 控件中服务器控件的类别。例如当有一组服务器用于填写用户的信息时，可以将 **Panel** 控件的 **GroupText** 属性编写成为“用户信息”，让用户知道该区域是用于填写用户信息的。



## 5.9 占位控件（Placeholder）

在传统的 ASP 开发中，通常在开发页面的时候，每个页面有很多相同的元素，例如导航栏、GIF 图片等。使用 ASP 进行应用程序开发通常使用 **include** 语句在各个页面包含其他页面的代码，这样的方法虽然解决了相同元素的很多问题，但是代码不够美观，而且时常会出现问题。ASP.NET 中可以使用 **Placeholder** 来解决这个问题，与面板控件 **Panel** 控件相同的是，占位控件 **Placeholder** 也是控件的容器，但是在 HTML 页面呈现中本身并不产生 HTML，创建一个 **Placeholder** 控件代码如下所示。

```
<asp:Placeholder ID="Placeholder1" runat="server"></asp:Placeholder>
```

在 CS 页面中，允许用户动态的在 **Placeholder** 上创建控件，CS 页面代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    TextBox text = new TextBox();           //创建一个 TextBox 对象
    text.Text = "NEW";
    this.Placeholder1.Controls.Add(text);   //为占位控件动态增加一个控件
}
```

上述代码动态的创建了一个 **TextBox** 控件并显示在占位控件中，运行效果如图 5-24 所示。



图 5-24 Placeholder 控件的使用

开发人员不仅能够通过编程在 **Placeholder** 控件中添加控件，开发人员同样可以在 **Placeholder** 控件中拖动相应的服务器控件进行控件呈现和分组。

## 5.10 日历控件（Calendar）

在传统的 Web 开发中，日历是最复杂也是最难实现的功能，好在 ASP.NET 中提供了强大的日历控件来简化日历控件的开发。日历控件能够实现日历的翻页、日历的选取以及数据的绑定，开发人员能够在博客、OA 等应用的开发中使用日历控件从而减少日历应用的开发。

### 5.10.1 日历控件的样式

日历控件通常在博客、论坛等程序中使用，日历控件不仅仅只是显示了一个日历，用户还能够通过日历控件进行时间的选取。在 ASP.NET 中，日历控件还能够和数据库进行交互操作，实现复杂的数据绑定。开发人员能够将日历控件拖动在主窗口中，在主窗口的代码视图下会自动生成日历控件的 HTML 代码，示例代码如下所示。

```
<asp:Calendar ID="Calendar1" runat="server"></asp:Calendar>
```

ASP.NET 通过上述简单的代码就创建了一个强大的日历控件，其效果如图 5-25 所示。



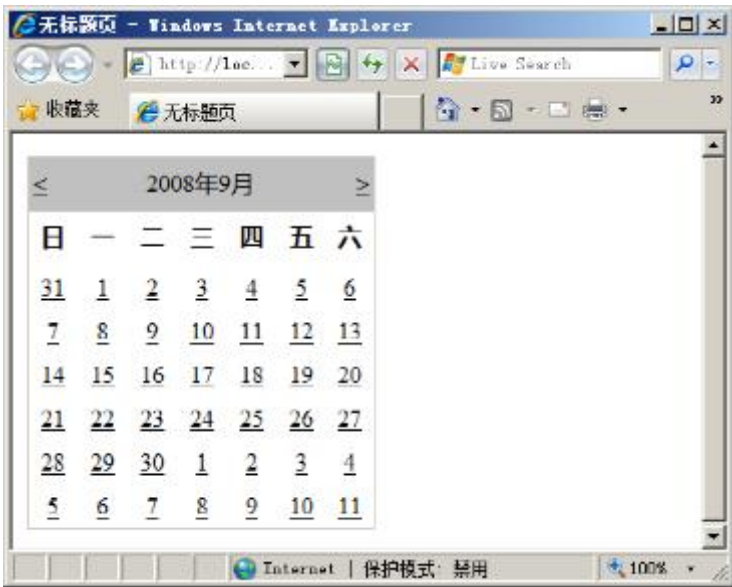


图 5-25 日历控件

日历控件通常用于显示月历，日历控件允许用户选择日期和移动到下一页或上一页。通过设置日历控件的属性，可以更改日历控件的外观。常用的日历控件的属性如下所示：

- ☐ **DayHeaderStyle:** 月历中显示一周中每一天的名称和部分的样式。
- ☐ **DayStyle:** 所显示的月份中各天的样式。
- ☐ **NextPrevStyle:** 标题栏左右两端的月导航所在部分的样式。
- ☐ **OtherMonthDayStyle:** 上一个月和下个月的样式。
- ☐ **SelectedDayStyle:** 选定日期的样式。
- ☐ **SelectorStyle:** 位于月历控件左侧，包含用于选择一周或整个月的连接的列样式。
- ☐ **ShowDayHeader:** 显示或隐藏一周中的每一天的部分。
- ☐ **ShowGridLines:** 显示或隐藏一个月中的每一天之间的网格线。
- ☐ **ShowNextPrevMonth:** 显示或隐藏到下一个月或上一个月的导航控件。
- ☐ **ShowTitle:** 显示或隐藏标题部分。
- ☐ **TitleStyle:** 位于月历顶部，包含月份名称和月导航连接的标题栏样式。
- ☐ **TodayDayStyle:** 当前日期的样式。
- ☐ **WeekendDayStyle:** 周末日期的样式。

Visual Studio 还为开发人员提供了默认的日历样式从而能够选择自动套用格式进行样式控制，如图 5-26 所示。

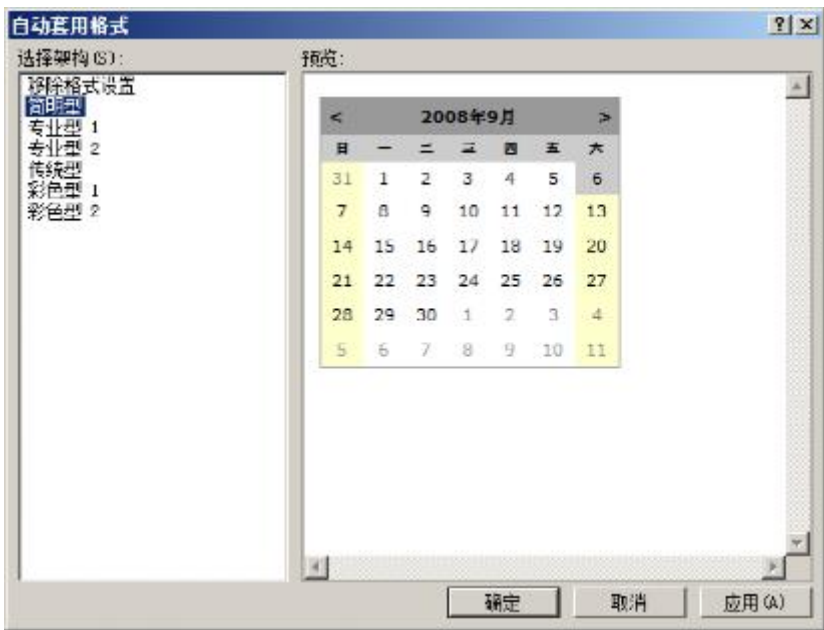


图 5-26 使用系统样式

除了上述样式可以设置以外，ASP.NET 还为用户设计了若干样式，若开发人员觉得设置样式非常困难，则可以使用系统默认的风格进行日历控件的样式呈现。

## 5.10.2 日历控件的事件

同所有的控件相同，日历控件也包含自身的事件，常用的日历控件的事件包括有：

- ❑ **DayRender:** 当日期被显示时触发该事件。
- ❑ **SelectionChanged:** 当用户选择日期时触发该事件。
- ❑ **VisibleMonthChanged:** 当所显示的月份被更改时触发该事件。

在创建日历控件中每个日期单元格时，则会触发 **DayRender** 事件。当用户选择月历中的日期时，则会触发 **SelectionChanged** 事件，同样，当双击日历控件时，会自动生成该事件的代码块。当对当前月份进行切换，则会激发 **VisibleMonthChanged** 事件。开发人员可以通过一个标签来接受当前事件，当选择月历中的某一天，则此标签显示当前日期，示例代码如下所示。

```
protected void Calendar1_SelectionChanged(object sender, EventArgs e)
{
    Label1.Text =
        "现在是:" + Calendar1.SelectedDate.Year.ToString() + "年"
        + Calendar1.SelectedDate.Month.ToString()+"月"
        + Calendar1.SelectedDate.Day.ToString()+"号"
        + Calendar1.SelectedDate.Hour.ToString()+"点";
}
```

在上述代码中，当用户选择了月历中的某一天时，则标签中的文本会变为当前的日期文本，如“现在是 **xx**”之类。在进行逻辑编程的同时，也需要对日历控件的样式做稍许更改，日历控件的 **HTML** 代码如下所示。

```
<asp:Calendar ID="Calendar1" runat="server" BackColor="#FFFFCC"
    BorderColor="#FFCC66" BorderWidth="1px" DayNameFormat="Shortest"
    Font-Names="Verdana" Font-Size="8pt" ForeColor="#663399" Height="200px"
    onselectionchanged="Calendar1_SelectionChanged" ShowGridLines="True"
    Width="220px">
    <SelectedDayStyle BackColor="#CCCCFF" Font-Bold="True" />
    <SelectorStyle BackColor="#FFCC66" />
    <TodayDayStyle BackColor="#FFCC66" ForeColor="White" />
    <OtherMonthDayStyle ForeColor="#CC9966" />
    <NextPrevStyle Font-Size="9pt" ForeColor="#FFFFCC" />
    <DayHeaderStyle BackColor="#FFCC66" Font-Bold="True" Height="1px" />
    <TitleStyle BackColor="#990000" Font-Bold="True" Font-Size="9pt"
        ForeColor="#FFFFCC" />
</asp:Calendar>
```

上述代码中的日历控件选择的是 **ASP.NET** 的默认样式，如图 5-27 所示。当确定了日历控件样式后，并编写了相应的 **SelectionChanged** 事件代码后，就可以通过日历控件获取当前时间，或者对当前时间进行编程，如图 5-28 所示。



图 5-27 日历控件

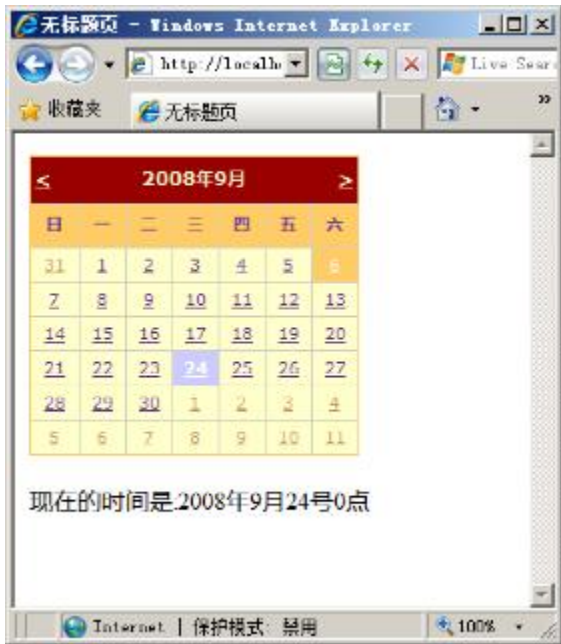


图 5-28 选择一个日期

5.11 广告控件（AdRotator）

在 Web 应用开发中，广告总是必不可少的。而 ASP.NET 为开发人员提供了广告控件为页面在加载时提供一个或一组广告。广告控件可以从固定的数据源中读取（如 XML 或数据源控件），并从中自动读取广告信息。当页面每刷新一次时，广告显示的内容也同样会被刷新。

广告控件必须放置在 Form 或 Panel 控件，以及模板内。广告控件需要包含图像的地址的 XML 文件。并且该文件用来指定每个广告的导航连接。广告控件最常用的属性就是 AdvertisementFile，使用它来配置相应的 XML 文件，所以必须首先按照标准格式创建一个 XML 文件，如图 5-29 所示。



图 5-29 创建一个 XML 文件

创建了 XML 文件之后，开发人员并不能按照自己的意愿进行 XML 文档的编写，如果要正确的被广告控件解析形成广告，就需要按照广告控件要求的标准的 XML 格式来编写代码，示例代码如下所示。

```
<?xml version="1.0" encoding="utf-8" ?>
<Advertisements>
  [<Ad>
    <ImageUrl></ImageUrl>
    <NavigateUrl></NavigateUrl>
    [<OptionalImageUrl></OptionalImageUrl>]*
    [<OptionalNavigateUrl></OptionalNavigateUrl>]*
    <AlternateText></AlternateText>
```

```
<Keyword></Keyword>
<Impression></Impression>
</Ad>]*
</Advertisements>
```

上述代码实现了一个标准的广告控件的 XML 数据源格式，其中各标签意义如下所示：

- ❑ **ImageUrl**: 指定一个图片文件的相对路径或绝对路径，当没有 **ImageKey** 元素与 **OptionalImageUrl** 匹配时则显示该图片。
  - ❑ **NavigateUrl**: 当用户单击广告时单没有 **NaavigateUrlKey** 元素与 **OptionalNavigateUrl** 元素匹配时，会将用户发送到该页面。
  - ❑ **OptionalImageUrl**: 指定一个图片文件的相对路径或绝对路径，对于 **ImageKey** 元素与 **OptionalImageUrl** 匹配时则显示该图片。
  - ❑ **OptionalNavigateUrl**: 当用户单击广告时单有 **NaavigateUrlKey** 元素与 **OptionalNavigateUrl** 元素匹配时，会将用户发送到该页面。
  - ❑ **AltemateText**: 该元素用来替代 **IMG** 中的 **ALT** 元素。
  - ❑ **KeyWord**: **KeyWord** 用来指定广告类别。
  - ❑ **Impression**: 该元素是一个数值，指示轮换时间表中该广告相对于文件中的其他广告的权重。
- 当创建了一个 XML 数据源之后，就需要对广告控件的 **AdvertisementFile** 进行更改，如图 5-30 所示。

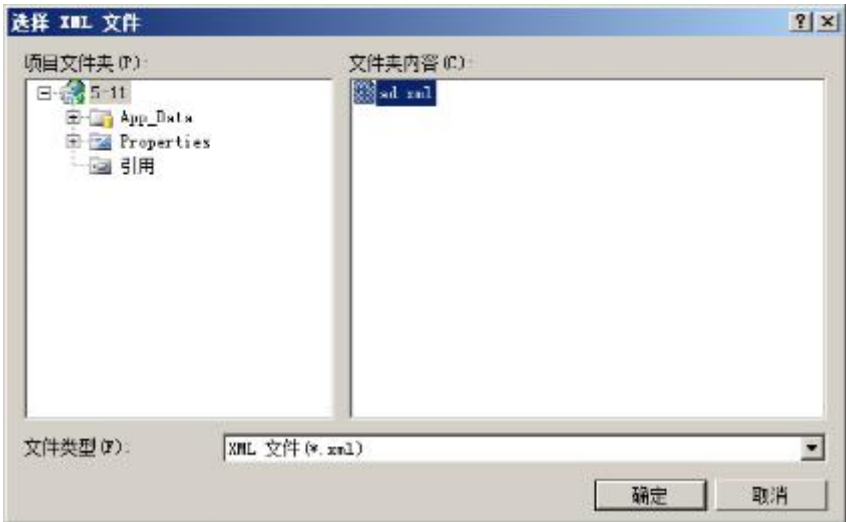


图 5-30 指定相应的数据源

配置好数据源之后，就需要在广告控件的数据源 XML 文件中加入自己的代码了，XML 广告文件示例代码如下所示。

```
<?xml version="1.0" encoding="utf-8" ?>
<Advertisements>
  <Ad>
    <ImageUrl>http://www.shangducms.com/images/cms.jpg</ImageUrl>
    <NavigateUrl>http://www.shangducms.com</NavigateUrl>
    <AltemateText>我的网站</AltemateText>
    <Keyword>software</Keyword>
    <Impression>100</Impression>
  </Ad>
  <Ad>
    <ImageUrl>http://www.shangducms.com/images/hello.jpg</ImageUrl>
    <NavigateUrl>http://www.shangducms.com</NavigateUrl>
    <AltemateText>我的网站</AltemateText>
    <Keyword>software</Keyword>
    <Impression>100</Impression>
  </Ad>
</Advertisements>
```

运行程序，广告对应的图像在页面每次加载的时候被呈现，如图 5-31 所示。页面每次刷新时，广告控件呈现的广告内容都会被刷新，如图 5-32 所示。





图 5-31 一个广告被呈现

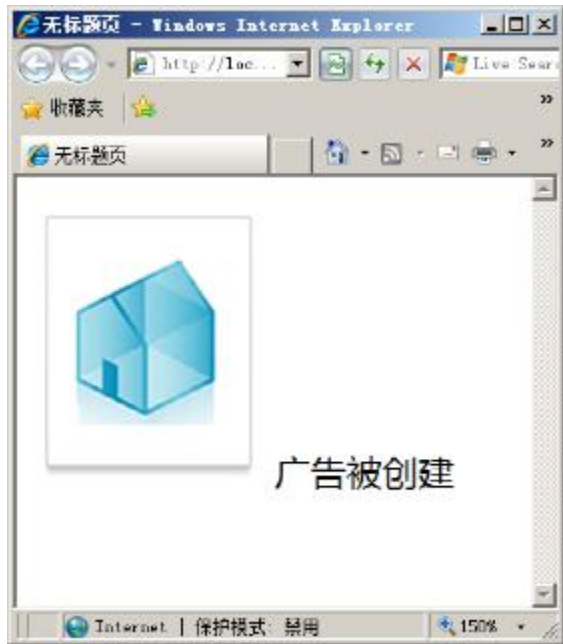


图 5-32 刷新后更换广告内容

注意：广告控件本身并不提供点击统计，所以无法计算广告是否被用户点击或者统计用户最关心的广告。

## 5.12 文件上传控件（FileUpload）

在网站开发中，如果需要加强用户与应用程序之间的交互，就需要上传文件。例如在论坛中，用户需要上传文件分享信息或在博客中上传视频分享快乐等等。上传文件在 ASP 中是一个复杂的问题，可能需要通过组件才能够实现文件的上传。在 ASP.NET 中，开发环境默认提供了文件上传控件来简化文件上传的开发。当开发人员使用文件上传控件时，将会显示一个文本框，用户可以键入或通过“浏览”按钮浏览和选择希望上传到服务器的文件。创建一个文件上传控件系统生成的 HTML 代码如下所示。

```
<asp:FileUpload ID="FileUpload1" runat="server" />
```

文件上传控件可视化设置属性较少，大部分都是通过代码控制完成的。当用户选择了一个文件并提交页面后，该文件作为请求的一部分上传，文件将被完整的缓存在服务器内存中。当文件完成上传，页面才开始运行，在代码运行的过程中，可以检查文件的特征，然后保存该文件。同时，上传控件在选择文件后，并不会立即执行操作，需要其他的控件来完成操作，例如按钮控件（Button）。实现文件上传的 HTML 核心代码如下所示。

```
<body>
  <form id="form1" runat="server">
    <div>
      <asp:FileUpload ID="FileUpload1" runat="server" />
      <asp:Button ID="Button1" runat="server" Text="选择好了，开始上传" />
    </div>
  </form>
</body>
```

上述代码通过一个 Button 控件来操作文件上传控件，当用户单击按钮控件后就能够将上传控件中选中的控件上传到服务器空间中，示例代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    FileUpload1.PostedFile.SaveAs(Server.MapPath("upload/beta.jpg")); //上传文件另存为
}
```

上述代码将一个文件上传到了 upload 文件夹内，并保存为 jpg 格式，如图 5-33 所示。打开服务器文件，可以看到文件已经上传了，如图 5-34 所示。



图 5-33 上传文件

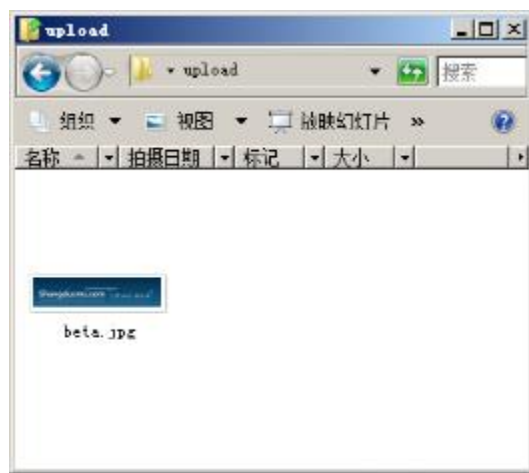


图 5-34 文件已经被上传

上述代码将文件保存在 **UPLOAD** 文件夹中, 并保存为 **JPG** 格式。但是通常情况下, 用户上传的并不全部都是 **JPG** 格式, 也有可能是 **DOC** 等其他格式的文件, 在这段代码中, 并没有对其他格式进行处理而全部保存为了 **JPG** 格式。同时, 也没有对上传的文件进行过滤, 存在着极大的安全风险, 开发人员可以将相应的文件上传的 **cs** 更改, 以便限制用户上传的文件类型, 示例代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    if (FileUpload1.HasFile) //如果存在文件
    {
        string fileExtension = System.IO.Path.GetExtension(FileUpload1.FileName); //获取文件扩展名
        if (fileExtension != ".jpg") //如果扩展名不等于 jpg 时
        {
            Label1.Text = "文件上传类型不正确, 请上传 jpg 格式"; //提示用户重新上传
        }
        else
        {
            FileUpload1.PostedFile.SaveAs(Server.MapPath("upload/beta.jpg")); //文件保存
            Label1.Text = "文件上传成功"; //提示用户成功
        }
    }
}
```

上述代码中决定了用户只能上传 **JPG** 格式, 如果用户上传的文件不是 **JPG** 格式, 那么用户将被提示上传的文件类型有误并停止用户的文件上传, 如果文件的类型为 **JPG** 格式, 用户就能够上传文件到服务器的相应目录中, 如图 5-35 所示。运行上传控件进行文件上传, 运行结果如图 5-36 所示。



图 5-35 文件类型错误



图 5-36 文件类型正确

值得注意的是, 上传的文件在 **.NET** 中, 默认上传文件最大为 **4M** 左右, 不能上传超过该限制的任何内容。当然, 开发人员可以通过配置 **.NET** 相应的配置文件来更改此限制, 但是推荐不要更改此限制, 否则可能造成潜在的安全威胁。

注意：如果需要更改默认上传文件大小的值，通常可以直接修改存放在 C:\WINDOWS\Microsoft.NET\Framework\V2.0.50727\CONFIG 的 ASP.NET 2.0 配置文件，通过修改文件中的 `maxRequestLength` 标签的值，或者可以通过 `web.config` 来覆盖配置文件。

## 5.13 视图控件（MultiView 和 View）

视图控件很像在 WinForm 开发中的 TabControl 控件，在网页开发中，可以使用 MultiView 控件作为一个或多个 View 控件的容器，让用户体验得到更大的改善。在一个 MultiView 控件中，可以放置多个 View 控件（选项卡），当用户点击到关心的选项卡时，可以显示相应的内容，很像 Visual Studio 2008 中的设计、视图、拆分等类型的功能。

无论是 MultiView 还是 View，都不会在 HTML 页面中呈现任何标记。而 MultiView 控件和 View 没有像其他控件那样多的属性，惟一需要指定的就是 ActiveViewIndex 属性，视图控件 HTML 代码如下所示。

```
<asp:MultiView ID="MultiView1" runat="server" ActiveViewIndex="0">
  <asp:View ID="View1" runat="server">
    abc<br />
    <asp:Button ID="Button1" runat="server" CommandArgument="View2"
      CommandName="SwitchViewByID" Text="下一个" />
  </asp:View>
  <asp:View ID="View2" runat="server">
    123<br />
    <asp:Button ID="Button2" runat="server" CommandArgument="View1"
      CommandName="SwitchViewByID" Text="上一个" />
  </asp:View>
</asp:MultiView>
```

上述代码中，使用了 Button 来对视图控件进行选择，通过单击按钮，来选择替换到【下一个】或者是【上一个】按钮，如图 5-37 所示。在用户注册中，这一步能够制作成 Web 向导，让用户更加方便的使用 Web 应用。当标签显式完毕后，会显式上一步按钮 5-38 所示。



图 5-37 第一个标签



图 5-38 第二个标签

注意：在 HTML 代码中，并没有为每个按钮的事件编写代码，是因为按钮通过 CommandArgument 和 CommandName 属性操作视图控件。

MultiView 和 View 控件能够实现 Panel 控件的任务，但可以让用户选择其他条件。同时 MultiView 和 View 能够实现 Wizard 控件相似的行为，并且可以自己编写实现细节。相比之下，当不需要使用 Wizard 提

供的方法时，可以使用 **MultiView** 和 **View** 控件来代替，并且编写过程更加“可视化”，如图 5-39 所示。

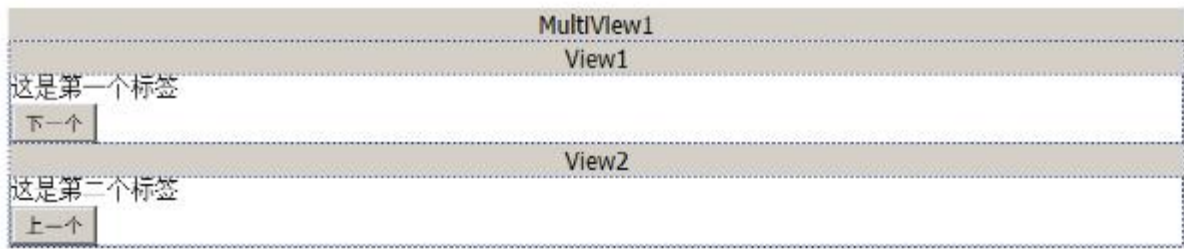


图 5-39 为每个 **View** 编写不同的应用

**MultiView** 和 **View** 控件也可以实现导航效果，可以通过编程指定 **MultiView** 的 **ActiveViewIndex** 属性显示相应的 **View** 控件。

注意：在 **MultiView** 控件中，第一个被放置的 **View** 控件的索引为 0 而不是 1，后面的 **View** 控件的索引依次递增。

### 5.14 表控件（Table）

在 **ASP.NET** 中，也提供了表控件（**Table**）来提供可编程的表格服务器控件。表中的行可以通过 **TableRow** 创建，而表中的列通过 **TableCell** 来实现，当创建一个表控件时，系统生成代码如下所示。

```
<asp:Table ID="Table1" runat="server" Height="121px" Width="177px">
</asp:Table>
```

上述代码自动生成了一个表控件代码，但是没有生成表控件中的行和列，必须通过 **TableRow** 创建行，通过 **TableCell** 来创建列，示例代码如下所示。

```
<asp:Table ID="Table1" runat="server" Height="121px" Width="177px">
<asp:TableRow>
<asp:TableCell>1.1</asp:TableCell>
<asp:TableCell>1.2</asp:TableCell>
<asp:TableCell>1.3</asp:TableCell>
<asp:TableCell>1.4</asp:TableCell>
</asp:TableRow>
<asp:TableRow>
<asp:TableCell>2.1</asp:TableCell>
<asp:TableCell>2.2</asp:TableCell>
<asp:TableCell>2.3</asp:TableCell>
<asp:TableCell>2.4</asp:TableCell>
</asp:TableRow>
</asp:Table>
```

上述代码创建了一个两行四列的表，如图 5-40 所示。



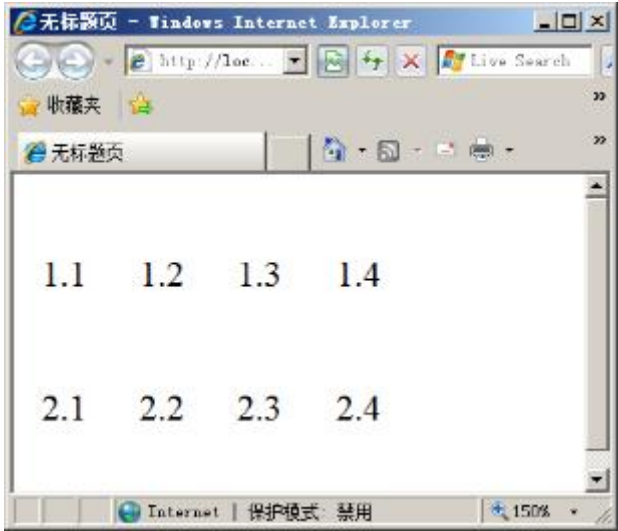


图 5-40 表控件

父 **Table** 控件支持一些控制整个表的外观的属性，例如字体、背景颜色等，如图 5-41 所示。**TableRow** 控件和 **TableCell** 控件也支持这些属性，同样可以用来指定个别的行或单元格的外观，运行后如图 5-42 所示。



图 5-41 Table 的属性设置



图 5-42 TableCell 控件的属性设置

表控件和静态表的区别在于，表控件能够动态的为表格创建行或列，实现一些特定的程序需求。**Web** 服务器控件中，**Table** 控件中的行是 **TableRow** 对象，**Table** 控件中的列是 **TableCell** 对象。可以声明这两个对象并初始化，可以为表控件增加行或列，实现动态创建表的程序，**HTML** 核心代码如下所示。

```
<body style="font-style: italic">
  <form id="form1" runat="server">
    <div>
      <asp:Table ID="Table1" runat="server" Height="121px" Width="177px"
        BackColor="Silver">
        <asp:TableRow>
          <asp:TableCell>1.1</asp:TableCell>
          <asp:TableCell>1.2</asp:TableCell>
          <asp:TableCell>1.3</asp:TableCell>
          <asp:TableCell BackColor="White">1.4</asp:TableCell>
        </asp:TableRow>
        <asp:TableRow>
          <asp:TableCell>2.1</asp:TableCell>
          <asp:TableCell BackColor="White">2.2</asp:TableCell>
          <asp:TableCell>2.3</asp:TableCell>
          <asp:TableCell>2.4</asp:TableCell>
        </asp:TableRow>
      </asp:Table>
      <br />
      <asp:Button ID="Button1" runat="server" onclick="Button1_Click" Text="增加一行" />
    </div>
  </form>
```

</body>

上述代码中，创键了一个二行一列的表格，同时创建了一个 **Button** 按钮控件来实现增加一行的效果，**cs** 核心代码如下所示。

```
namespace _5_14
{
    public partial class _Default : System.Web.UI.Page
    {
        public TableRow row = new TableRow(); //定义一个 TableRow 对象
        protected void Page_Load(object sender, EventArgs e)
        {
        }
        protected void Button1_Click(object sender, EventArgs e)
        {
            Table1.Rows.Add(row); //创建一个新行
            for (int i = 0; i < 4; i++) //遍历四次创建新列
            {
                TableCell cell = new TableCell(); //定义一个 TableCell 对象
                cell.Text = "3."+i.ToString(); //编写 TableCell 对象的文本
                row.Cells.Add(cell); //增加列
            }
        }
    }
}
```

上述代码动态的创建了一行并动态的在该行创建了四列，如图 5-43 所示。单击【增加一行】按钮，系统会在表格中创建新行，运行效果如图 5-44 所示。

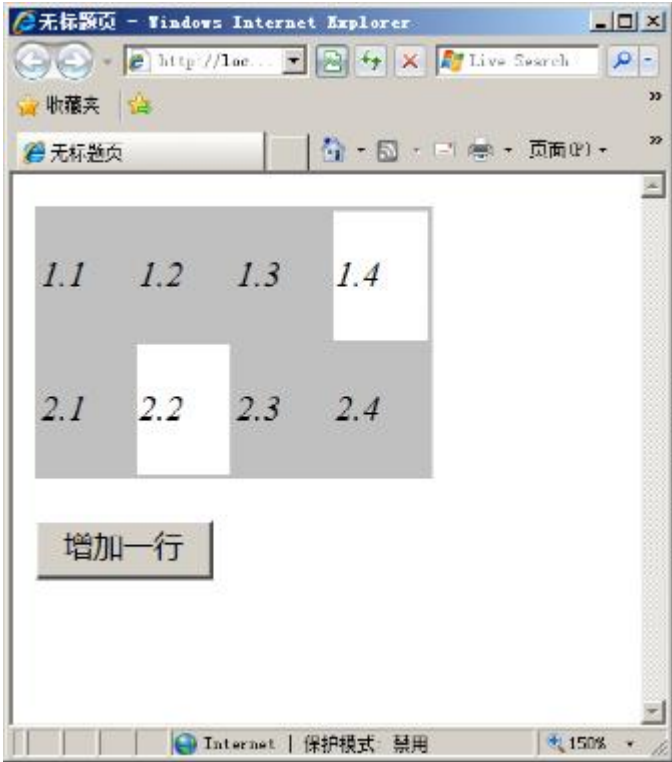


图 5-43 原表格

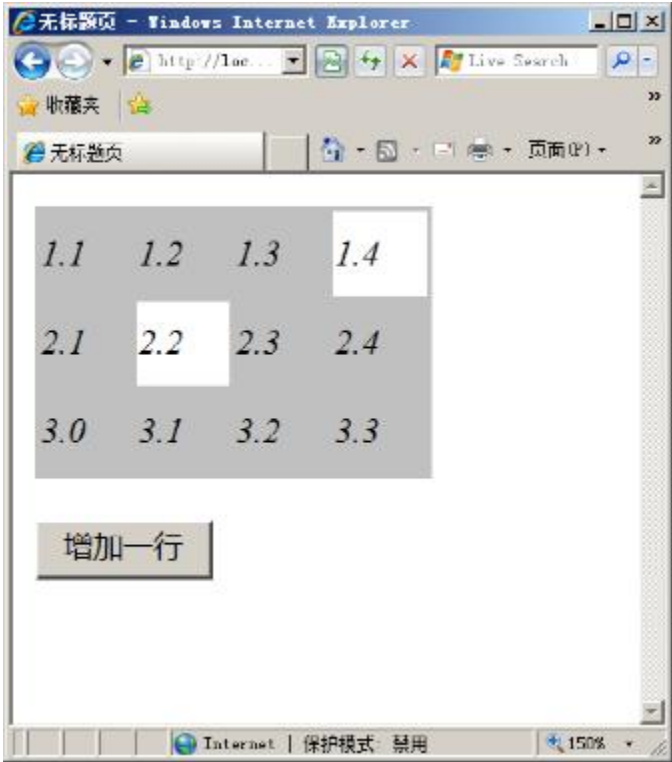


图 5-44 动态创建行和列

在动态创建行和列的时候，也能够修改行和列的样式等属性，创建自定义样式的表格。通常，表不仅用来显示表格的信息，还是一种传统的布局网页的形式，创建网页表格有如下几种形式：

- ❑ **HTML** 格式的表格：如<table>标记显示的静态表格。
- ❑ **HtmlTable** 控件：将传统的<table>控件通过添加 **runat=server** 属性将其转换为服务器控件。
- ❑ **Table** 表格控件：就是本节介绍的表格控件。

虽然创建表格有以上三种创建方法，但是推荐开发人员在使用静态表格，当不需要对表格做任何逻辑事物处理时，最好使用 **HTML** 格式的表格，因为这样可以极大的降低页面逻辑、增强性能。

## 5.15 向导控件（Wizard）

在 **WinForm** 开发中，安装程序会一步一步的提示用户安装，或者在应用程序配置中，同样也有向导提示用户，让应用程序安装和配置变得更加的简单。与之相同的是，在 **ASP.NET** 中，也提供了一个向导控件，便于在搜集用户信息、或提示用户填写相关的表单时使用。

### 5.15.1 向导控件的样式

当创建了一个向导控件时，系统会自动生成向导控件的 **HTML** 代码，示例代码如下所示。

```
<asp:Wizard ID="Wizard1" runat="server">
  <WizardSteps>
    <asp:WizardStep runat="server" title="Step 1">
    </asp:WizardStep>
    <asp:WizardStep runat="server" title="Step 2">
    </asp:WizardStep>
  </WizardSteps>
</asp:Wizard>
```

上述代码生成了 **Wizard** 控件，并在 **Wizard** 控件中自动生成了 **WizardSteps** 标签，这个标签规范了向导控件中的步骤，如图 5-45 所示。在向导控件中，系统会生成 **WizardSteps** 控件来显示每一个步骤，如图 5-46 所示。



图 5-45 向导控件



图 5-46 完成后的向导控件

在 **ASP.NET 2.0** 之前，并没有 **Wizard** 向导控件，必须创建自定义控件来实现 **Wizard** 向导控件的效果，如视图控件。而在 **ASP.NET 2.0** 之后，系统就包含了向导控件，同样该控件也保留到了 **ASP.NET 3.5**。向导控件能够根据步骤自动更换选项，如当还没有执行到最后一步时，会出现【上一步】或【下一步】按钮以便用户使用，当向导执行完毕时，则会显示完成按钮，极大的简化了开发人员的向导开发过程。

向导控件还支持自动显示标题和控件的当前步骤。标题使用 **HeaderText** 属性自定义，同时还可以配置 **DisplayCancelButton** 属性显示一个取消按钮，如图 5-47 所示。不仅如此，当需要让向导控件支持向导步骤的添加时，只需配置 **WizardSteps** 属性即可，如图 5-48 所示。

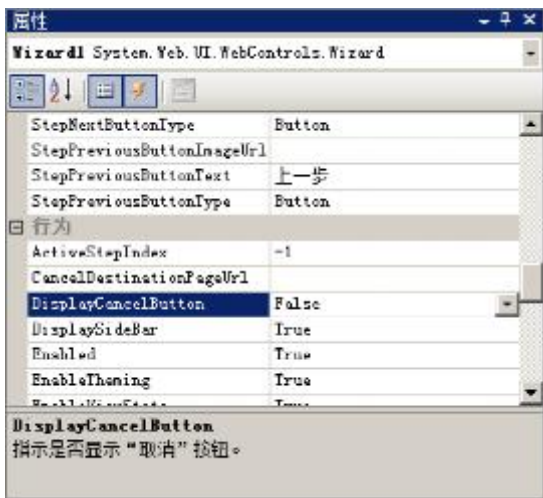


图 5-47 显式“取消”按钮

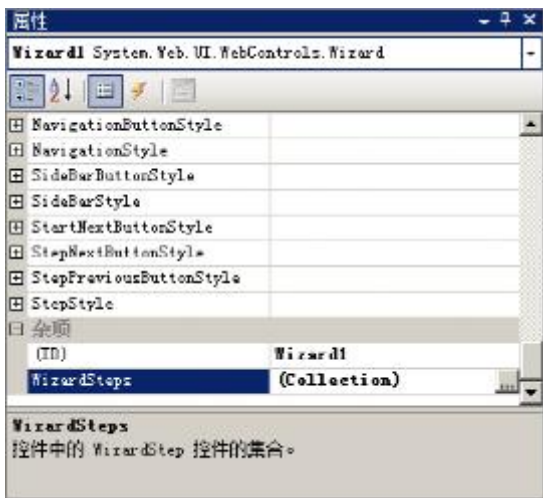


图 5-48 配置步骤

**Wizard** 向导控件还支持一些模板。用户可以配置相应的属性来配置向导控件的模板。用户可以通过编辑 **StartNavigationTemplate** 属性、**FinishNavigationTemplate** 属性、**StepNavigationTemplate** 属性以及 **SideBarTemplate** 属性来进行自定义控件的界面设定。这些属性的意义如下所示：

- ❑ **StartNavigationTemplate**：该属性指定为 **Wizard** 控件的 **Start** 步骤中的导航区域显示自定义内容。
- ❑ **FinishNavigationTemplate**：该属性为 **Wizard** 控件的 **Finish** 步骤中的导航区域指定自定义内容。
- ❑ **StepNavigationTemplate**：该属性为 **Wizard** 控件的 **Step** 步骤中的导航区域指定自定义内容。
- ❑ **SideBarTemplate**：该属性为 **Wizard** 控件的侧栏区域中指定自定义内容。

以上属性都可以通过可视化功能来编辑或修改，如图 5-49 所示。



图 5-49 导航控件的模板支持

导航控件还能够自定义模板来实现更多的特定功能，同时导航控件还能够为导航控件的其他区域定义进行样式控制，如导航列表和导航按钮等。

5.15.2 导航控件的事件

当双击一个导航控件时，导航控件会自动生成 **FinishButtonClick** 事件。该事件是当用户完成导航控件时被触发。导航控件页面 **HTML** 核心代码如下所示。

```
<body>
  <form id="form1" runat="server">
    <asp:Wizard ID="Wizard1" runat="server" ActiveStepIndex="2"
      DisplayCancelButton="True" onfinishbuttonclick="Wizard1_FinishButtonClick">
      <WizardSteps>
        <asp:WizardStep runat="server" title="Step 1">
          执行的是第一步</asp:WizardStep>
        <asp:WizardStep runat="server" title="Step 2">
          执行的是第二步</asp:WizardStep>
        <asp:WizardStep runat="server" Title="Step3">
          感谢您的使用</asp:WizardStep>
      </WizardSteps>
    </asp:Wizard>
  </div>
    <asp:Label ID="Label1" runat="server" Text="Label"></asp:Label>
  </div>
```



```
</form>
</body>
```

上述代码为向导控件进行了初始化，并提示用户正在执行的步骤，当用户执行完毕后，会提示感谢您的使用并在相应的文本标签控件中显示“向导控件执行完毕”。当单击向导控件时，会触发 **FinishButtonClick** 事件，通过编写 **FinishButtonClick** 事件能够为向导控件进行编码控制，示例代码如下所示。

```
protected void Wizard1_FinishButtonClick(object sender, WizardNavigationEventArgs e)
{
    Label1.Text = "向导控件执行完毕";
}
```

在执行的过程中，标签文本会显示执行的步骤，如图 5-50 所示。当运行完毕时，**Label** 标签控件会显示“向导控件执行完毕”，同时向导控件中的文本也会呈现“感谢您的使用”字样。运行结果如图 5-51 所示。



图 5-50 执行第二步

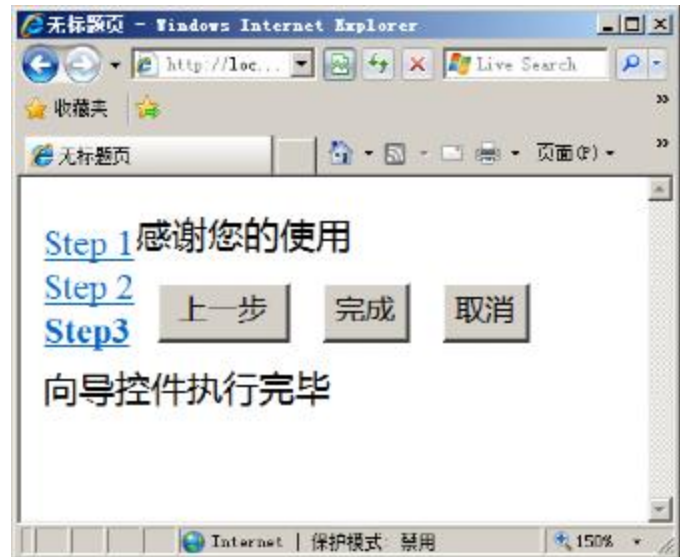


图 5-51 用户单击完成后执行事件

向导控件不仅能够使用 **FinishButtonClick** 事件，同样也可以使用 **PreviousButtonClick** 和 **FinishButtonClick** 事件来自定义【上一步】按钮和【下一步】按钮的行为，同样也可以编写 **CancelButtonClick** 事件定义单击【取消】按钮时需要执行的操作。

## 5.16 XML 控件

**XML** 控件可以读取 **XML** 并将其写入该控件所在的 **ASP.NET** 网页。**XML** 控件能够将 **XSL** 转换应用到 **XML**，还能够将最终转换的内容输出呈现在该页中。当创建一个 **XML** 控件时，系统会生成 **XML** 控件的 **HTML** 代码，示例代码如下所示。

```
<asp:Xml ID="Xml1" runat="server"></asp:Xml>
```

上述代码实现了简单的 **XML** 控件，**XML** 控件还包括两个常用的属性，这两个属性分别如下所示：

- ❑ **DocumentSource**: 应用转换的 **XML** 文件。
- ❑ **TransformSource**: 用于转换 **XML** 数据的 **XSL** 文件。

开发人员可以通过 **XML** 控件的 **DocumentSource** 属性提供的 **XML**，**XSL** 文件的路径来进行加载，并将相应的代码呈现到控件上，示例代码如下所示。

```
<asp:Xml ID="Xml1" runat="server" DocumentSource="~/XMLFile1.xml"></asp:Xml>
```

上述代码为 **XML** 控件指定了 **DocumentSource** 属性，通过加载 **XML** 文档进行相应的代码呈现，运行后如图 5-52 所示。

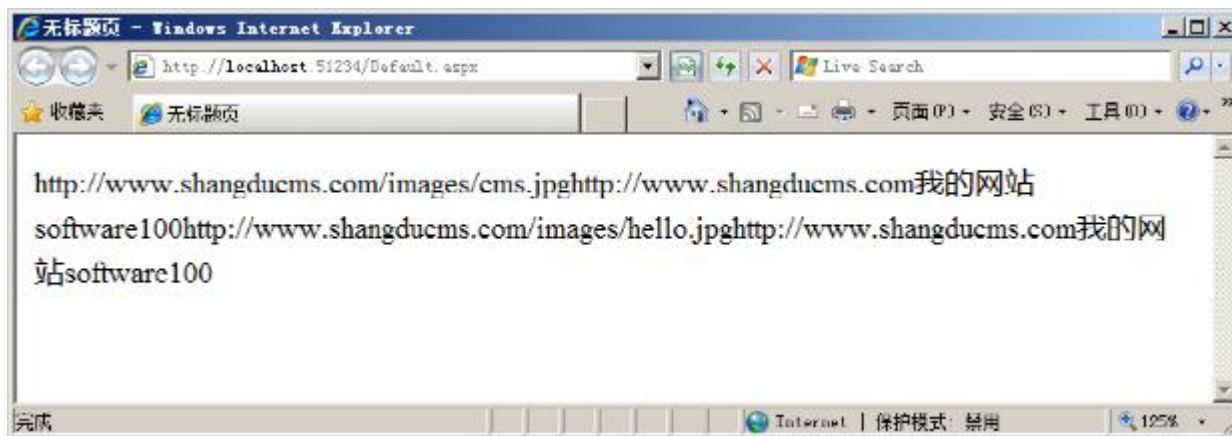


图 5-52 加载 XML 文档

XML 控件不仅能够呈现 XML 文档的内容，还能够进行相应的 XML 的文本操作。在本书的第 14 章中，会详细讲解如何使用 ASP.NET 进行操作 XML。

## 5.17 验证控件

ASP.NET 提供了强大的验证控件，它可以验证服务器控件中用户的输入，并在验证失败的情况下显示一条自定义错误消息。验证控件直接在客户端执行，用户提交后执行相应的验证无需使用服务器端进行验证操作，从而减少了服务器与客户端之间的往返过程。

### 5.17.1 表单验证控件（RequiredFieldValidator）

在实际的应用中，如在用户填写表单时，有一些项目是必填项，例如用户名和密码。在传统的 ASP 中，当用户填写表单后，页面需要被发送到服务器并判断表单中的某项 HTML 控件的值是否为空，如果为空，则返回错误信息。在 ASP.NET 中，系统提供了 **RequiredFieldValidator** 验证控件进行验证。使用 **RequiredFieldValidator** 控件能够指定某个用户在特定的控件中必须提供相应的信息，如果不填写相应的信息，**RequiredFieldValidator** 控件就会提示错误信息，**RequiredFieldValidator** 控件示例代码如下所示。

```
<body>
  <form id="form1" runat="server">
    <div>
      姓名:<asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
        <asp:RequiredFieldValidator ID="RequiredFieldValidator1" runat="server"
          ControlToValidate="TextBox1" ErrorMessage="必填字段不能为空"></asp:RequiredFieldValidator>
      <br />
      密码:<asp:TextBox ID="TextBox2" runat="server"></asp:TextBox>
      <br />
      <asp:Button ID="Button1" runat="server" Text="Button" />
    </div>
  </form>
</body>
```

在进行验证时，**RequiredFieldValidator** 控件必须绑定一个服务器控件，在上述代码中，验证控件 **RequiredFieldValidator** 控件的服务器控件绑定为 **TextBox1**，当 **TextBox1** 中的值为空时，则会提示自定义错误信息“必填字段不能为空”，如图 5-53 所示。

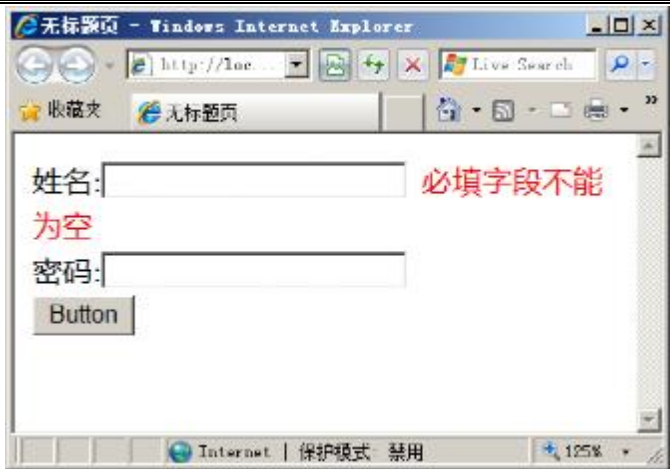


图 5-53 RequiredFieldValidator 验证控件

当姓名选项未填写时，会提示必填字段不能为空，并且该验证在客户端执行。当发生此错误时，用户会立即看到该错误提示而不会立即进行页面提交，当用户填写完成并再次单击按钮控件时，页面才会向服务器提交。

## 5.17.2 比较验证控件（CompareValidator）

比较验证控件对照特定的数据类型来验证用户的输入。因为当用户输入用户信息时，难免会输入错误信息，如当需要了解用户的生日时，用户很可能输入了其他的字符串。**CompareValidator** 比较验证控件能够比较控件中的值是否符合开发人员的需要。**CompareValidator** 控件的特有属性如下所示：

- ❑ **ControlToCompare**：以字符串形式输入的表达式。要与另一控件的值进行比较。
- ❑ **Operator**：要使用的比较。
- ❑ **Type**：要比较两个值的数据类型。
- ❑ **ValueToCompare**：以字符串形式输入的表达式。

当使用 **CompareValidator** 控件时，可以方便的判断用户是否正确输入，示例代码如下所示。

```
<body>
  <form id="form1" runat="server">
    <div>
      请输入生日:
      <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
      <br />
      毕业日期:
      <asp:TextBox ID="TextBox2" runat="server"></asp:TextBox>
      <asp:CompareValidator ID="CompareValidator1" runat="server"
        ControlToCompare="TextBox2" ControlToValidate="TextBox1"
        CultureInvariantValues="True" ErrorMessage="输入格式错误！请改正！"
        Operator="GreaterThan"
        Type="Date">
      </asp:CompareValidator>
      <br />
      <asp:Button ID="Button1" runat="server" Text="Button" />
      <br />
    </div>
  </form>
</body>
```

上述代码判断 **TextBox1** 的输入的格式是否正确，当输入的格式错误时，会提示错误，如图 5-54 所示。

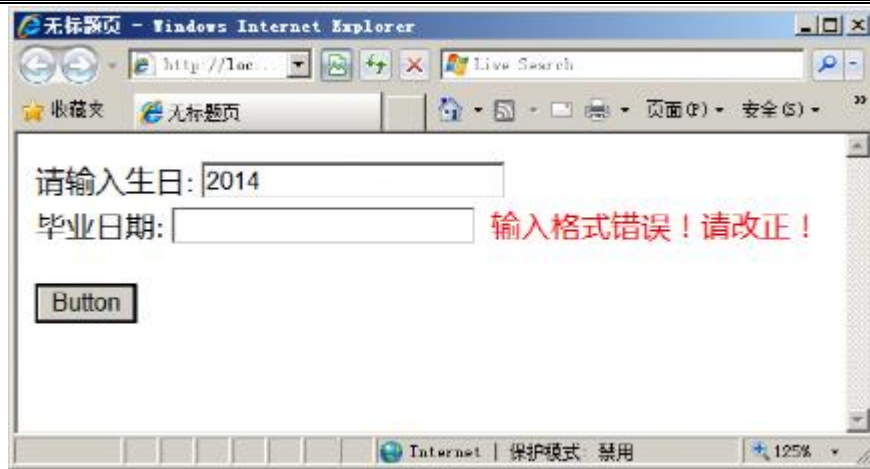


图 5-54 CompareValidator 验证控件

**CompareValidator** 验证控件不仅能够验证输入的格式是否正确，还可以验证两个控件之间的值是否相等。如果两个控件之间的值不相等，**CompareValidator** 验证控件同样会将自定义错误信息呈现在用户的客户端浏览器中。

## 5.17.3 范围验证控件（RangeValidator）

范围验证控件（**RangeValidator**）可以检查用户的输入是否在指定的上限与下限之间。通常情况下用于检查数字、日期、货币等。范围验证控件（**RangeValidator**）控件的常用属性如下所示。

- ❑ **MinimumValue**: 指定有效范围的最小值。
- ❑ **MaximumValue**: 指定有效范围的最大值。
- ❑ **Type**: 指定要比较的值的数据类型。

通常情况下，为了控制用户输入的范围，可以使用该控件。当输入用户的生日时，今年是 **2008** 年，那么用户就不应该输入 **2009** 年，同样基本上没有人的寿命会超过 **100**，所以对输入的日期的下限也需要进行规定，示例代码如下所示。

```
<div>
    请输入生日:<asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
    <asp:RangeValidator ID="RangeValidator1" runat="server"
        ControlToValidate="TextBox1" ErrorMessage="超出规定范围，请重新填写"
        MaximumValue="2009/1/1" MinimumValue="1990/1/1" Type="Date">
    </asp:RangeValidator>
    <br />
    <asp:Button ID="Button1" runat="server" Text="Button" />
</div>
```

上述代码将 **MinimumValue** 属性值设置为 **1990/1/1**，并能将 **MaximumValue** 的值设置为 **2009/1/1**，当用户的日期低于最小值或高于最高值时，则提示错误，如图 5-55 所示。



图 5-55 RangeValidator 验证控件



注意：**RangeValidator** 验证控件在进行控件的值的范围的设定时，其范围不仅仅可以是一个整数值，同样还能够是时间、日期等值。

## 5.17.4 正则验证控件（RegularExpressionValidator）

在上述控件中，虽然能够实现一些验证，但是验证的能力是有限的，例如在验证的过程中，只能验证是否是数字，或者是否是日期。也可能在验证时，只能验证一定范围内的数值，虽然这些控件提供了一些验证功能，但却限制了开发人员进行自定义验证和错误信息的开发。为实现一个验证，很可能需要多个控件同时搭配使用。

正则验证控件（**RegularExpressionValidator**）就解决了这个问题，正则验证控件的功能非常的强大，它用于确定输入的控件的值是否与某个正则表达式所定义的模式相匹配，如电子邮件、电话号码以及序列号等。

正则验证控件（**RegularExpressionValidator**）常用的属性是 **ValidationExpression**，它用来指定用于验证的输入控件的正则表达式。客户端的正则表达式验证语法和服务端的正则表达式验证语法不同，因为在客户端使用的是 **JScript** 正则表达式语法，而在服务器端使用的是 **Regex** 类提供的正则表达式语法。使用正则表达式能够实现强大字符串的匹配并验证用户的输入的格式是否正确，系统提供了一些常用的正则表达式，开发人员能够选择相应的选项进行规则筛选，如图 5-56 所示。



图 5-56 系统提供的正则表达式

当选择了正则表达式后，系统自动生成的 **HTML** 代码如下所示。

```
<asp:RegularExpressionValidator ID="RegularExpressionValidator1" runat="server"
    ControlToValidate="TextBox1" ErrorMessage="正则不匹配,请重新输入!"
    ValidationExpression="\d{17}[\d|X]|\d{15}">
</asp:RegularExpressionValidator>
```

运行后当用户单击按钮控件时，如果输入的信息与相应的正则表达式不匹配，则会提示错误信息，如图 5-57 所示。



图 5-57 **RegularExpressionValidator** 验证控件

同样，开发人员也可以自定义正则表达式来规范用户的输入。使用正则表达式能够加快验证速度并在

字符串中快速匹配，而另一方面，使用正则表达式能够减少复杂的应用程序的功能开发和实现。

注意：在用户输入为空时，其他的验证控件都会验证通过。所以，在验证控件的使用中，通常需要同  
表单验证控件（**RequiredFieldValidator**）一起使用。

## 5.17.5 自定义逻辑验证控件（CustomValidator）

自定义逻辑验证控件（**CustomValidator**）允许使用自定义的验证逻辑创建验证控件。例如，可以创建一个验证控件判断用户输入的是否包含“.”号，示例代码如下所示。

```
protected void CustomValidator1_ServerValidate(object source, ServerValidateEventArgs args)
{
    args.IsValid = args.Value.ToString().Contains(".");           //设置验证程序，并返回布尔值
}
protected void Button1_Click(object sender, EventArgs e)         //用户自定义验证
{
    if (Page.IsValid)                                             //判断是否验证通过
    {
        Label1.Text = "验证通过";                                //输出验证通过
    }
    else
    {
        Label1.Text = "输入格式错误";                            //提交失败信息
    }
}
```

上述代码不仅使用了验证控件自身的验证，也使用了用户自定义验证，运行结果如图 5-58 所示。



图 5-58 CustomValidator 验证控件

从 **CustomValidator** 验证控件的验证代码可以看出，**CustomValidator** 验证控件可以在服务器上执行验证检查。如果要创建服务器端的验证函数，则处理 **CustomValidator** 控件的 **ServerValidate** 事件。使用传入的 **ServerValidateEventArgs** 的对象的 **IsValid** 字段来设置是否通过验证。

而 **CustomValidator** 控件同样也可以在客户端实现，该验证函数可用 **VBScript** 或 **Jscript** 来实现，而在 **CustomValidator** 控件中需要使用 **ClientValidationFunction** 属性指定与 **CustomValidator** 控件相关的客户端验证脚本的函数名称进行控件中的值的验证。

## 5.17.6 验证组控件（ValidationSummary）

验证组控件（**ValidationSummary**）能够对同一页面的多个控件进行验证。同时，验证组控件（**ValidationSummary**）通过 **ErrorMessage** 属性为页面上的每个验证控件显式错误信息。验证组控件

(**ValidationSummary**) 的常用属性如下所示。

- ❑ **DisplayMode**: 摘要可显示为列表，项目符号列表或单个段落。
- ❑ **HeaderText**: 标题部分指定一个自定义标题。
- ❑ **ShowMessageBox**: 是否在消息框中显示摘要。
- ❑ **ShowSummary**: 控制是显示还是隐藏 **ValidationSummary** 控件。

验证控件能够显示页面的多个控件产生的错误，示例代码如下所示。

```
<body>
  <form id="form1" runat="server">
    <div>
      姓名:
      <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
      <asp:RequiredFieldValidator ID="RequiredFieldValidator1" runat="server"
        ControlToValidate="TextBox1" ErrorMessage="姓名为必填项">
      </asp: RequiredFieldValidator>
      <br />
      身份证:
      <asp:TextBox ID="TextBox2" runat="server"></asp:TextBox>
      <asp:RegularExpressionValidator ID="RegularExpressionValidator1" runat="server"
        ControlToValidate="TextBox1" ErrorMessage="身份证号码错误"
        ValidationExpression="\d{17}[\d|X]|\d{15}"></asp:RegularExpressionValidator>
      <br />
      <asp:Button ID="Button1" runat="server" Text="Button" />
      <asp:ValidationSummary ID="ValidationSummary1" runat="server" />
    </div>
  </form>
</body>
```

运行结果如图 5-59 所示。

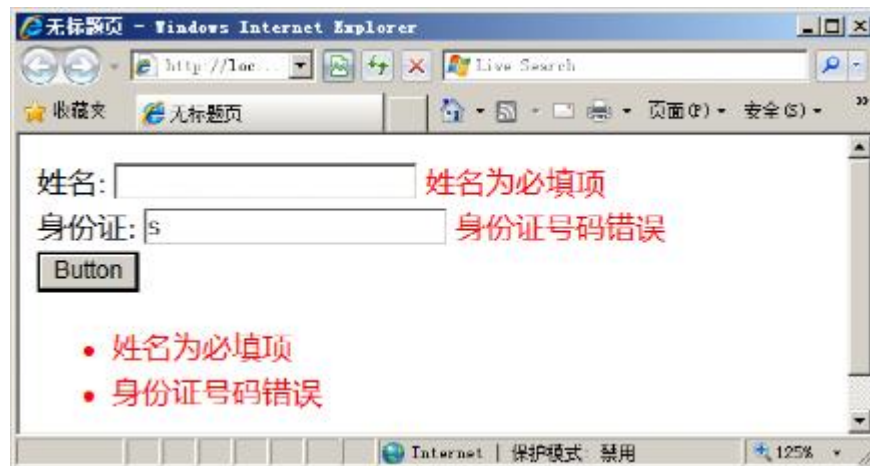


图 5-59 **ValidationSummary** 验证控件

当有多个错误发生时，**ValidationSummary** 控件能够捕获多个验证错误并呈现给用户，这样就避免了一个表单需要多个验证时需要使用多个验证控件进行绑定，使用 **ValidationSummary** 控件就无需为每个需要验证的控件进行绑定。

## 5.18 导航控件

在网站制作中，常常需要制作导航来让用户能够更加方便快捷的查阅到相关的信息和资讯，或能跳转到相关的版块。在 **Web** 应用中，导航是非常重要的。**ASP.NET** 提供了站点导航的一种简单的方法，即使用站点图形站点导航控件 **SiteMapPath**、**TreeView**、**Menu** 等控件。

导航控件包括 **SiteMapPath**、**TreeView**、**Menu** 三个控件，这三个控件都可以在页面中轻松建立导航。这三个导航控件的基本特征如下所示：



- ❑ **SiteMapPath**: 检索用户当前页面并显示层次结构的控件。这使用户可以导航回到层次结构中的其他页。**SiteMap** 控件专门与 **SiteMapProvider** 一起使用。
- ❑ **TreeView**: 提供纵向用户界面以展开和折叠网页上的选定节点，以及为选定像提供复选框功能。并且 **TreeView** 控件支持数据绑定。
- ❑ **Menu**: 提供在用户将鼠标指针悬停在某一项时弹出附加子菜单的水平或垂直用户界面。

这三个导航控件都能够快速的建立导航，并且能够调整相应的属性为导航控件进行自定义。

**SiteMapPath** 控件使用户能够从当前导航回站点层次结构中较高的页，但是该控件并不允许用户从当前页面向前导航到层次结构中较深的其他页面。相比之下，使用 **TreeView** 或 **Menu** 控件，用户可以打开节点并直接选择需要跳转的特定页。这些控件不会像 **SiteMapPath** 控件一样直接读取站点地图。**TreeView** 和 **Menu** 控件不仅可以自定义选项，也可以绑定一个 **SiteMapDataSource**。**TreeView** 和 **Menu** 控件的基本样式如图 5-60 和图 5-61 所示。

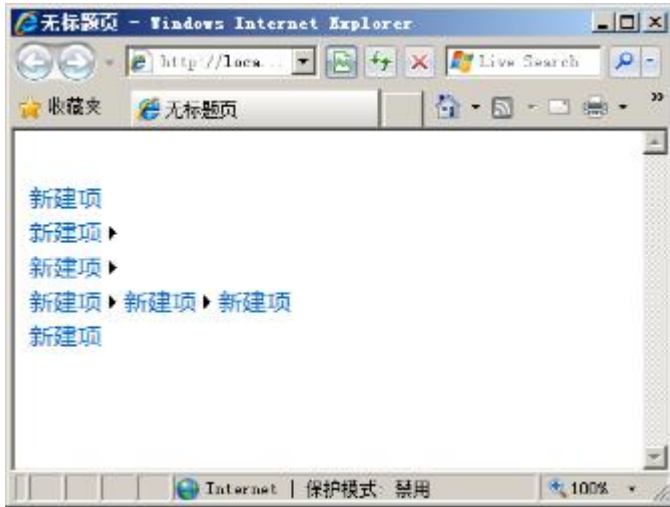


图 5-60 Menu 导航控件

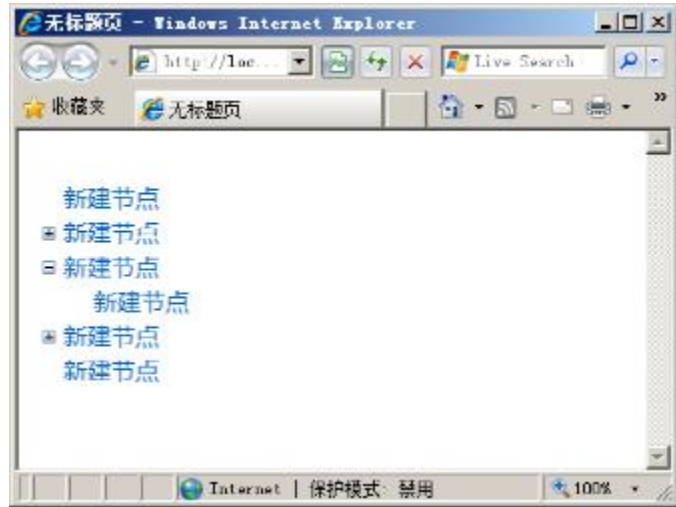


图 5-61 TreeView 导航控件

**TreeView** 和 **Menu** 控件生成的代码并不相同，因为 **TreeView** 和 **Menu** 控件所实现的功能也不尽相同。**TreeView** 和 **Menu** 控件的代码分别如下所示。

```
<asp:Menu ID="Menu1" runat="server">
  <Items>
    <asp:MenuItem Text="新建项" Value="新建项"></asp:MenuItem>
    <asp:MenuItem Text="新建项" Value="新建项">
      <asp:MenuItem Text="新建项" Value="新建项"></asp:MenuItem>
    </asp:MenuItem>
    <asp:MenuItem Text="新建项" Value="新建项">
      <asp:MenuItem Text="新建项" Value="新建项"></asp:MenuItem>
    </asp:MenuItem>
    <asp:MenuItem Text="新建项" Value="新建项">
      <asp:MenuItem Text="新建项" Value="新建项">
        <asp:MenuItem Text="新建项" Value="新建项"></asp:MenuItem>
      </asp:MenuItem>
    </asp:MenuItem>
    <asp:MenuItem Text="新建项" Value="新建项"></asp:MenuItem>
  </Items>
</asp:Menu>
```

上述代码声明了一个 **Menu** 控件，并添加了若干节点。

```
<asp:TreeView ID="TreeView1" runat="server">
  <Nodes>
    <asp:TreeNode Text="新建节点" Value="新建节点"></asp:TreeNode>
    <asp:TreeNode Text="新建节点" Value="新建节点">
      <asp:TreeNode Text="新建节点" Value="新建节点"></asp:TreeNode>
    </asp:TreeNode>
    <asp:TreeNode Text="新建节点" Value="新建节点">
      <asp:TreeNode Text="新建节点" Value="新建节点"></asp:TreeNode>
    </asp:TreeNode>
  </Nodes>
</asp:TreeView>
```



```
<asp:TreeNode Text="新建节点" Value="新建节点">
    <asp:TreeNode Text="新建节点" Value="新建节点"></asp:TreeNode>
</asp:TreeNode>
<asp:TreeNode Text="新建节点" Value="新建节点"></asp:TreeNode>
</Nodes>
</asp:TreeView>
```

上述代码声明了一个 **TreeView** 控件，并添加了若干节点。

从上面的代码和运行后的实例图可以看出，**TreeView** 和 **Menu** 控件有一些区别，这些具体区别如下所示：

- ☐ **Menu** 展开时，是弹出形式的展开，而 **TreeView** 控件则是就地展开。
- ☐ **Menu** 控件并不是按需下载，而 **TreeView** 控件则是按需下载的。
- ☐ **Menu** 控件不包含复选框，而 **TreeView** 控件包含复选框。
- ☐ **Menu** 控件允许编辑模板，而 **TreeView** 控件不允许模板编辑。
- ☐ **Menu** 在布局上是水平和垂直，而 **TreeView** 只是垂直布局。
- ☐ **Menu** 可以选择样式，而 **TreeView** 不行。

开发人员在网站开发的时候，可以通过使用导航控件来快速的建立导航，为浏览者提供方便，也为网站做出信息指导。在用户的使用中，通常情况下导航控件中的导航值是不能被用户所更改的，但是开发人员可以通过编程的方式让用户也能够修改站点地图的节点。

## 5.19 其他控件

在 **ASP.NET** 中，除了以上常用的一些基本控件以外，还有一些其他基本控件，虽然在应用程序开发中并不经常使用，但是在特定的程序开发中，还是需要使用到这些基本的控件进行特殊的应用程序开发和逻辑处理。

### 5.19.1 隐藏输入框控件（HiddenField）

**HiddenField** 控件就是隐藏输入框控件，用来保存那些不需要显示在页面上的对安全性要求不高的数据。隐藏输入框控件作为 `<input type="hidden">` 元素呈现在 **HTML** 页面。由于 **HiddenField** 隐藏输入框控件的值会呈现在客户端浏览器，所以对于安全性较高的数据，并不推荐将它保存在隐藏输入框控件中。隐藏输入框控件的值通过 **Value** 属性保存，同时也可以通过代码来控制 **Value** 的值，利用隐藏输入框对页面的值进行传递，示例代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    Label1.Text = HiddenField1.Value; //获取隐藏输入框控件的值
}
```

上述代码通过 **Value** 属性获取一个隐藏输入框的值，如图 5-62 所示。单击后如图 5-63 所示。

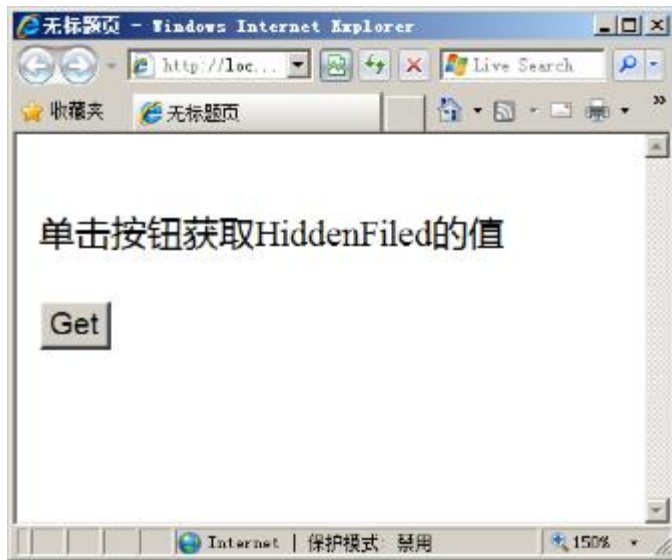


图 5-62 HiddenField 的值被隐藏

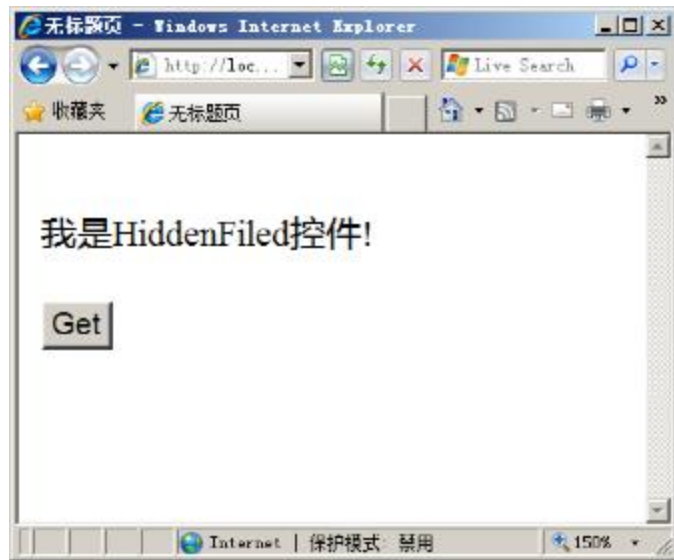


图 5-63 HiddenField 的值被获取

**HiddenField** 是通过 **HTTP** 协议进行参数传递的，所以当打开新的窗体或者使用 **method=get** 都无法使用 **HiddenField** 隐藏输入框控件。同时，隐藏输入框控件还能初始化或保存一些安全性不高的数据。当双击隐藏输入框控件时，系统会自动生成 **ValueChanged** 事件代码段，当隐藏输入框控件内的值被改变时则触发该事件，示例代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    HiddenField1.Value = "更改了值"; //更改隐藏输入框控件的值
}
protected void HiddenField1_ValueChanged(object sender, EventArgs e) //更改将触发此事件
{
    Label1.Text = "值被更改了,并被更改成\"" + HiddenField1.Value + "\"";
}
```

上述代码创建了一个 **ValueChanged** 事件，并当隐藏输入框控件的值被更改时，如图 5-64 所示。单击 **【change】** 按钮后会触发按钮事件，运行结果如图 5-65 所示。

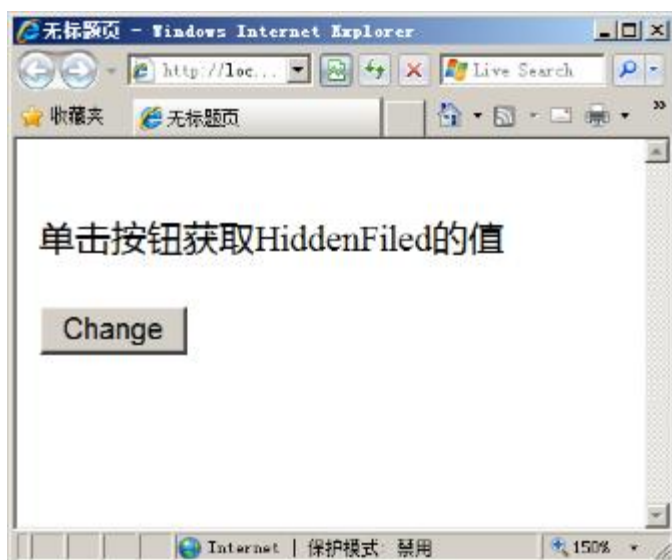


图 5-64 更新前

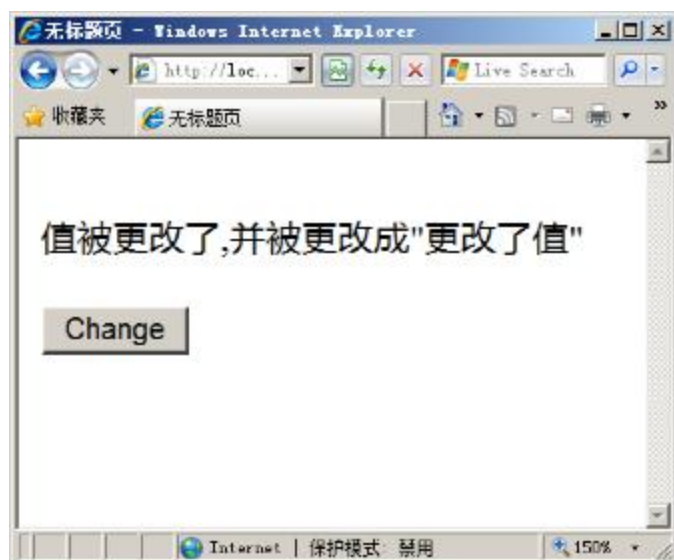


图 5-65 更新后

## 5.19.2 图片热点控件（ImageMap）

**ImageMap** 控件是一个让你可以在图片上定义热点（**HotSpot**）区域的服务器控件。用户可以通过点击这些热点区域进行回发（**PostBack**）操作或者定向（**Navigate**）到某个 **URL** 位址。该控件一般用在需要对某张图片的局部范围进行互动操作。**ImageMap** 控件主要由两个部分组成，第一部分是图像。第二部分是作用点控件的集合。其主要属性有 **HotSpotMode**、**HotSpots**，具体如下所示。

### 1. HotSpotMode（热点模式）常用选项

❑ **NotSet:** 未设置项。虽然名为未设置，但其实默认情况下会执行定向操作，定向到你指定的 **URL**

位址去。如果你未指定 **URL** 位址，那默认将定向到自己的 **Web** 应用程序根目录。

- ❑ **Navigate:** 定向操作项。定向到指定的 **URL** 位址去。如果你未指定 **URL** 位址，那默认将定向到自己的 **Web** 应用程序根目录。
- ❑ **PostBack:** 回发操作项。点击热点区域后，将执行后部的 **Click** 事件。
- ❑ **Inactive:** 无任何操作，即此时形同一张没有热点区域的普通图片。

## 2. HotSpots（图片热点）常用属性

该属性对应着 **System.Web.UI.WebControls.HotSpot** 对象集合。**HotSpot** 类是一个抽象类，它之下有 **CircleHotSpot**（圆形热区）、**RectangleHotSpot**（方形热区）和 **PolygonHotSpot**（多边形热区）三个子类。实际应用中，都可以使用上面三种类型来定制图片的热点区域。如果需要使用到自定义的热点区域类型时，该类型必须继承 **HotSpot** 抽象类。同时，**ImageMap** 最常用的事件有 **Click**，通常在 **HotSpotMode** 为 **PostBack** 时用到。当需要设置 **HotSpots** 属性时，可以可视化设置，如图 5-66 所示。



图 5-66 可视化设置 HotSpots 属性

当可视化完毕后，系统会自动生成 **HTML** 代码，核心代码如下所示。

```
<asp:ImageMap ID="ImageMap1" runat="server" HotSpotMode="PostBack"
    ImageUrl="~/images/mobile.jpg" onclick="ImageMap1_Click">
    <asp:CircleHotSpot Radius="15" X="15" Y="15" HotSpotMode="PostBack" PostBackValue="0" />
    <asp:CircleHotSpot Radius="100" X="15" Y="15" HotSpotMode="PostBack" PostBackValue="1" />
    <asp:CircleHotSpot Radius="300" X="15" Y="15" HotSpotMode="PostBack" PostBackValue="2" />
</asp:ImageMap>
```

上述代码还添加了一个 **Click** 事件，事件处理的核心代码如下所示。

```
protected void ImageMap1_Click(object sender, ImageMapEventArgs e)
{
    string str="";
    switch (e.PostBackValue) //获取传递过来的参数
    {
        case "0":
            str = "你点击了 1 号位置，图片大小将变为 1 号"; break;
        case "1":
            str = "你点击了 2 号位置，图片大小将变为 3 号"; break;
        case "2":
            str = "你点击了 3 号位置，图片大小将变为 3 号"; break;
    }
    Label1.Text = str;
    ImageMap1.Height =120*(Convert.ToInt32(e.PostBackValue)+1); //更改图片的大小
}
```

上述代码通过获取 **ImageMap** 中的 **CircleHotSpot** 控件中的 **PostBackValue** 值来获取传递的参数，如图 5-67 所示。当获取到传递的参数时，可以通过参数做相应的操作，如图 5-68 所示。





图 5-67 单击图片变大



图 5-68 单击图片变小

### 5.19.3 静态标签控件（Literal）

通常情况下 **Literal** 控件无需添加任何 **HTML** 元素即可将静态文本呈现在网页上。与 **Label** 不同的是，**Label** 控件在生成 **HTML** 代码时，会呈现 `<span>` 元素。而 **Literal** 控件不会向文本中添加任何 **HTML** 代码。如果开发人员希望文本和控件直接呈现在页面中而不使用任何附加标记时，推荐使用 **Literal** 控件。

与 **Label** 不同的是，**Literal** 控件有一个 **Mode** 属性，用来控制 **Literal** 控件中的文本的呈现形式。当 **HTML** 代码被输出到页面的时候，会以解释后的 **HTML** 形式输出。例如图片代码，在 **HTML** 中是以 `<img src="">` 的形式显示的，输出到 **HTML** 页面后，会被显示成一个图片。

在 **Label** 中，**Label** 可以作为一段 **HTML** 代码的容器，当输出时，**Label** 控件所呈现的效果是 **HTML** 被解释后的样式。而 **Literal** 可以通过 **Mode** 属性来选择输出的是 **HTML** 样式还是 **HTML** 代码，核心代码如下所示。

```
namespace _5_17
{
    public partial class Lieral : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            string str = "<span style=\"color:red\">大家好</span>，您现在查看的是 HTML 样式。";//HTML 字符
            Literal1.Text = str +
            "<div style=\"border-top:1px dashed #ccc;background:gray\">单击按钮查看 HTML 代码</div>";
            Label1.Text = str;                                     //赋值 Label
        }
        protected void Button1_Click(object sender, EventArgs e)
        {
            Literal1.Mode = LiteralMode.Encode;                  //转换显示的模式
        }
    }
}
```

上述代码将一个 **HTML** 形式的字符串分别赋值给 **Literal** 和 **Label** 控件，并通过转换查看赋值的源代码，运行结果如图 5-69 和图 5-70 所示。





图 5-69 Literal 控件直接显式 HTML 样式



图 5-70 Literal 控件显式 HTML 代码

当点击了按钮后, 更改了 **Literal** 的模式之后, **Literal** 中的 **HTML** 文本被直接显示, **Literal** 的具有三种模式, 具体如下:

- ☐ **Transform**: 添加到控件中的任何标记都将进行转换, 以适应请求浏览器的协议。
- ☐ **PassThrough**: 添加到控件中的任何标记都将按照原样输出在浏览器中。
- ☐ **Encode**: 添加到控件中的任何标记都将使用 **HtmlEncode** 方法进行编码, 该方法将把 **HTML** 编码转换为其文本表示形式。

注意: **PassThrough** 模式和 **Transform** 模式在通常情况下, 呈现的效果并没有区别。

## 5.19.4 动态缓存更新控件(Substitution)

在 **ASP.NET** 中, 缓存的使用能够极大的提高网站的性能, 降低服务器的压力。而通常情况下, 对 **ASP.NET** 整个页面的缓存是没有任何意义的, 这样经常会给用户带来疑惑。 **Substitution** 动态缓存更新控件允许用户在页上创建一些区域, 这些区域可以用动态的方式进行更新, 然后集成到缓存页。

**Substitution** 动态缓存更新控件将动态内容插入到缓存页中, 并且不会呈现任何 **HTML** 标记。用户可以将控件绑定到页上或父用户控件上的方法, 自行创建静态方法, 以返回要插入到页面中的任何信息。同时, 要使用 **Substitution** 控件, 则必须符合以下标准:

- ☐ 此方法被定义为静态方法。
- ☐ 此方法接受 **HttpContext** 类型的参数。
- ☐ 此方法返回 **String** 类型的值。

在 **ASP.NET** 页面中, 为了减少用户与页面的交互中数据库的更新, 可以对 **ASP.NET** 页面进行缓存, 缓存代码可以使用页面参数的 **@OutputCache**, 示例代码如下所示。

```
<%@ Page
Language="C#" AutoEventWireup="true" CodeBehind="Substitution.aspx.cs" Inherits="_5_17.Substitution" %>
<%@ OutputCache Duration="100" VaryByParam="none" %> //增加一个页面缓存
<!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>无标题页</title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            当前的时间为: <asp:Label ID="Label1" runat="server" Text="Label"></asp:Label>
```

```
(有缓存)<br />
当前的时间为: <asp:Substitution ID="Substitution1" runat="server" MethodName="GetTimeNow"/>
(动态更新)</div>
</form>
</body>
</html>
```

执行事件操作的 **cs** 页面核心代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    Label1.Text = DateTime.Now.ToString(); //页面初始化时, 当前时间赋值给 Label1 标签
}
protected static string GetTimeNow(HttpContext con) //注意事件的格式
{
    return DateTime.Now.ToString(); //Substitution 控件执行的方法
}
```

上述代码对 **ASP.NET** 页面进行了缓存, 当用户访问页面时, 除了 **Substitution** 控件的区域以外区域都会被缓存, 而使用了 **Substitution** 控件局部在刷新后会进行更新, 运行结果如图 5-71 所示。

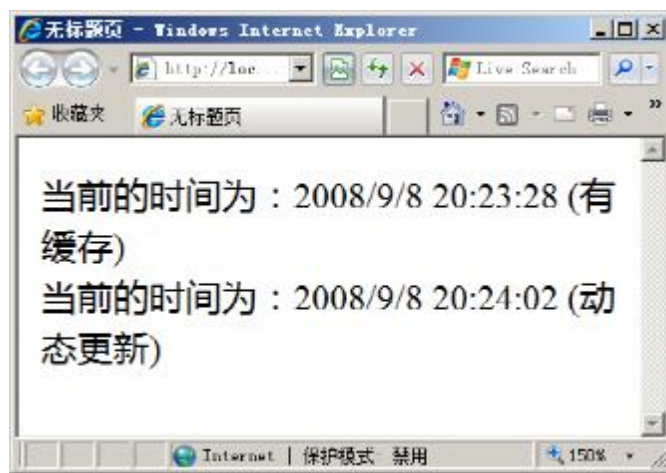


图 5-71 **Substitution** 动态更新

如运行结果可见, 没有使用 **Substitution** 控件的区域, 当页面再次被请求时, 会直接在缓存中执行。而 **Substitution** 控件区域内的值并不会缓存。在每次刷新时, 页面将进行 **Substitution** 控件的区域的局部的动态更新。

## 5.20 小结

本章讲解了 **ASP.NET** 中常用的控件, 对于这些控件, 能够极大的提高开发人员的效率, 对于开发人员而言, 能够直接拖动控件来完成应用的目的。虽然控件是非常的强大, 但是这些控件却制约了开发人员的学习, 人们虽然能够经常使用 **ASP.NET** 中的控件来创建强大的多功能网站, 却不能深入的了解控件的原理, 所以对这些控件的熟练掌握, 是了解控件的原理的第一步。本章还介绍了:

- ❑ 控件的属性: 介绍了控件的属性。
- ❑ 简单控件: 介绍了标签控件等简单控件。
- ❑ 文本框控件: 介绍了文本框控件。
- ❑ 按钮控件: 介绍了按钮控件的实现和按钮事件的运行过程。
- ❑ 单选控件和单选组控件: 介绍了单选控件和单选组控件。
- ❑ 复选框控件和复选组控件: 介绍了复选框控件和复选组控件。

这些控件为 **ASP.NET** 应用程序的开发提供了极大的便利, 在 **ASP.NET** 控件中, 不仅仅包括这些基本的服务器控件, 还包括高级的数据源控件和数据绑定控件用于数据操作, 但是在了解 **ASP.NET** 高级控件之前, 需要熟练的掌握基本控件的使用。

第 6 章 Web 窗体的高级控件

上一章中讲解了 ASP.NET 中常用的基本控件，ASP.NET 不仅提供了常用的基本控件如标签控件、文本框控件等，还提供了高级的 Web 窗体的控件。这些控件能够轻松实现更多在 ASP 开发中难以实现的效果。

6.1 登录控件

对于目前常用的网站系统而言，登录功能是必不可少的，例如论坛、电子邮箱、在线购物等。登录功能能够让网站准确的验证用户的身份。用户能够访问该网站时，可以注册并登录，登录后的用户还能够注销登录状态以保证用户资料的安全性。ASP.NET 就提供了一系列的登录控件方便登录功能的开发。

6.1.1 登录控件（Login）

登录控件是一个复合控件，它包含用户名和密码文本框，以及一个询问用户是否希望在下次访问该页面时记起其身份的复选框。当用户勾选此选项时，下次用户访问此网站后，将自动进行身份验证。创建一个登录控件代码，系统会自动生成相应的 HTML 代码，示例代码如下所示。

```
<asp:Login ID="Login1" runat="server">
</asp:Login>
```

上述代码则创建了一个登录控件，如图 6-1 所示。开发人员可以通过属性的设置更改登录控件的样式等，如图 6-2 所示。

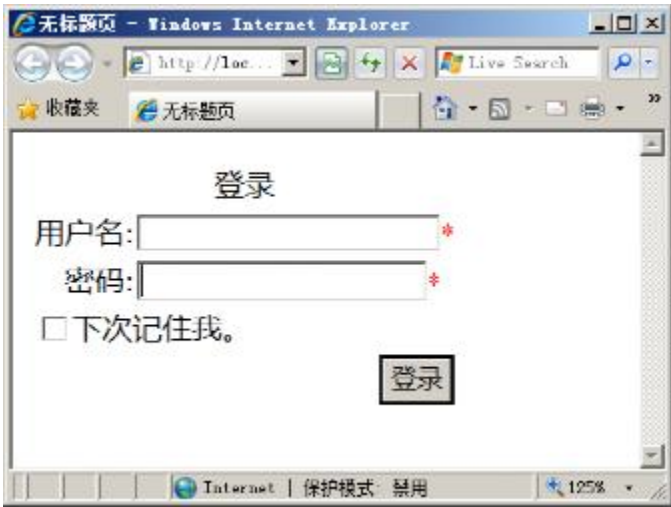


图 6-1 默认登录窗口

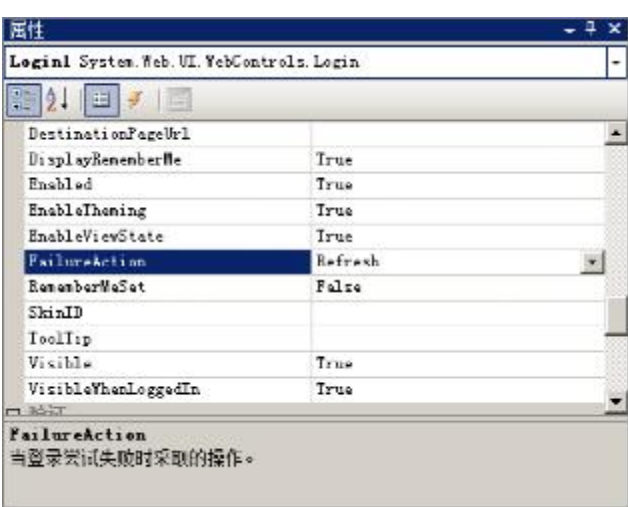


图 6-2 登录框属性的设置

开发人员能够使用登录控件执行用户登录操作而无需复杂的代码实现，登录控件常用的属性如下所示。

- ❑ **Orientation:** 控件的一般布局。
- ❑ **TextLayout:** 标签相对于文本框的布局。
- ❑ **CreateUserIconUrl:** 用户创建用户连接的图标的 URL。
- ❑ **CreateUserText:** 为“创建用户”连接显示的文本。
- ❑ **CreateUserUrl:** 创建用户页的 URL。
- ❑ **HelpPageIconUrl:** 用于帮助页连接的图标的 URL。
- ❑ **HelpPageText:** 为帮助连接显示的文本。
- ❑ **HelpPageUrl:** 帮助页的 URL。



- ❑ **PasswordRecoveryImageUrl**: 用于密码回复连接的图标的 URL。
- ❑ **PasswordRecoveryUrl**: 为密码回复连接显示的文本。
- ❑ **PasswordRecoveryText**: 密码回复页的 URL。
- ❑ **MembershipProvider**: 成员资格提供程序的名称。
- ❑ **FailuteText**: 当登录尝试失败时显示的文本。
- ❑ **InstructionText**: 为给出说明所显示的文本。
- ❑ **LoginButtonImageUrl**: 为“登录”按钮显示的图像的 URL。
- ❑ **LoginButtonText**: 为“登录”按钮显示的文本。
- ❑ **LoginButtonType**: “登录”按钮的类型。
- ❑ **PasswordLableText**: 密码标识文本框内的文本。
- ❑ **RememberMeText**: 为“记住我”复选框所显示的文本。
- ❑ **TitleText**: 为标题显示的文本。
- ❑ **UserName**: 用户名文本框内的初始值。
- ❑ **UserNameLableText**: 标识用户名文本框的文本。
- ❑ **DestinationPageUrl**: 用户成功登录时被定向到的 URL。
- ❑ **DisplayRememberMe**: 是否显示“记住我”复选框。
- ❑ **Enabled**: 控件是否处于启动状态。
- ❑ **RememberMeSet**: “记住我”复选框是否初始化被选中。
- ❑ **VisibleWhenLoggedIn**: 是否控件在用户登录时保持可见。
- ❑ **PasswordRequiredErrorMessage**: 密码为空时在验证摘要中显示的文本。
- ❑ **UserNameRequiredErrorMessage**: 用户名为空时在验证摘要中显示的文本。

同样，登录控件还包括许多常用的事件，登录控件常用的事件如下所示：

- ❑ **Authenticate**: 当用户使用登录控件登录到网站时，引发该事件。
- ❑ **LoggedIn**: 对用户进行身份验证后引发该事件。
- ❑ **LoggingIn**: 对用户进行身份验证前引发该事件。
- ❑ **LoginError**: 对用户进行用户身份验证失败时引发该事件。

开发人员能够在页面中拖动相应的登录控件实现登录操作，使用登录控件进行登录操作可以直接进行用户的信息的查询而无需复杂的登录实现。

## 6.1.2 登录名称控件（LoginName）

登录名称控件（**LoginName**）是一个用来显示已经成功登录的用户的控件。在 **Web** 应用程序开发中，开发人员常常需要在页面中通知相应的用户已经登录，如用户在商品网站上进行登录，登录成功后可以在相应的页面中提示“您已登录，您的用户名是 **XXX**”等，这样不仅能够提高用户的友好度，也能够让开发人员在 **Web** 应用程序中方便的对用户信息做收集整理。

开发人员能够方便的在应用程序中拖动 **LoginName** 控件用于用户名的呈现，拖动到页面中，系统生成的 **HTML** 代码如下所示。

```
<asp:LoginName ID="LoginName1" runat="server" />
```

上述代码则实现了一个登录名称控件，开发人员能够将该控件放置在页面中的任何位置进行页面呈现，当用户登录后，该控件能够获取用户的相应信息并呈现用户名在控件中。登录控件页面效果如图 6-3 所示。

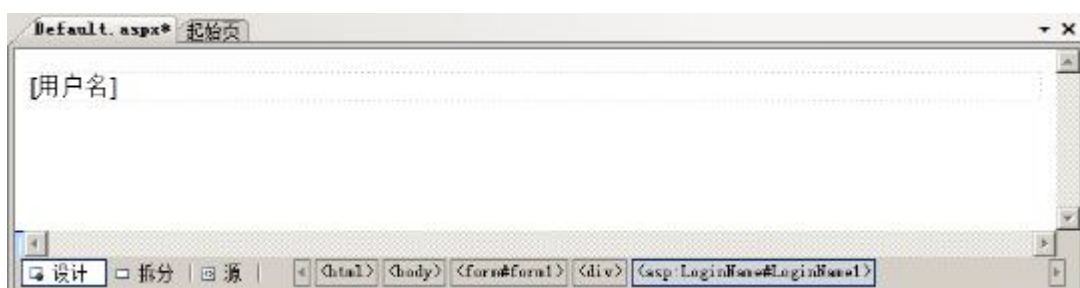




图 6-3 登录名称控件

注意：LoginName 控件只能够在<body>标记内的<form>标记中使用，该控件不能够使用于<title>、<style>等标记中。

在 LoginName 控件中，最常用的属性为 **FormatString** 属性，该属性用于格式化用户名输出。在控件的 **FormatString** 属性中，“{0}”字符串用于显式用户名，开发人员能够配置相应的字符串进行输出，例如配置成“您好，{0}，您已经登录！”，可以在相应的占位符中呈现相应的用户名，如图 6-4 所示。



图 6-4 格式化输出用户名

正如图 6-4 所示，当对 LoginName 进行格式化规定后，用户名能够被格式化输出，例如当用户 **soundbbg** 登录在 Web 应用后，该控件会呈现“您好，**soundbbg**，您已登录！”。开发人员只需要通过简单的配置就能够实现复杂的登录显示功能操作的实现。

## 6.1.3 登录视图控件（LoginView）

在应用程序的开发过程中，通常需要对不同的身份和权限的用户进行不同登录样式的呈现，开发人员可以为用户配置内置对象以呈现不同的页面效果。但是在页面请求时，还需要对用户的身份进行验证。在 ASP.NET 2.0 之后的版本中，系统提供了 **LoginView** 控件用于不同用户权限之间的视图的区分。

在开发一个应用程序时，开发人员希望应用程序能够实现功能当用户在网站中没有登录时，用户看到的视图是没有登录时的视图，包括网站的风格、系统的提示信息等。而当用户登录后，用户看到的视图是登录后的视图，同样包括网站的风格、系统的提示信息等。**LoginView** 控件为开发人员提供了不同权限的用户进行不同视图的查看的功能，开发人员能够拖动 **LoginView** 控件在页面中以编辑不同的页面进行开发。

拖动一个 **LoginView** 控件在页面中，开发人员能够通过编辑不同的模板进行不同权限的页面的编写，拖动 **LoginView** 控件后系统生成的 HTML 代码如下所示。

```
<asp:LoginView ID="LoginView1" runat="server">
  </asp:LoginView>
```

上述代码为默认的 **LoginView** 控件的代码，开发人员需要通过编写相应的模板以便不同的用户查看不同的页面，在 **LoginView** 控件中，包括两个最常用的模板，这两个模板及其作用分别如下所示。

- ❑ **AnonymousTemplate**: 匿名模板，当用户没有进行登录时，该模板会呈现在匿名用户面前。
- ❑ **LoggedInTemplate**: 已登录模板，当用户已经登录成功后，该模板会呈现在已经登录的用户面前。

开发人员可以通过编写相应的模板进行页面的呈现，当用户没有登录时，用户可以看见 **AnonymousTemplate** 模板中的内容而无法看见 **LoggedInTemplate** 模板的内容；而如果用户已经登录，则登录过后的用户能够看见 **LoggedInTemplate** 模板的内容而无法看见 **AnonymousTemplate** 模板的内容，如图 6-5 所示。

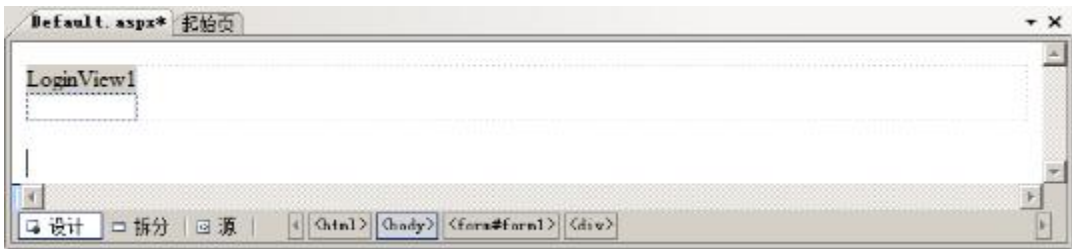


图 6-5 LoginView 控件

在 **AnonymousTemplate** 模板中，该模板通过获取和判断 **PageUser** 属性的 **Name** 属性进行判断。如果 **PageUser** 属性的 **Name** 属性为空时，**AnonymousTemplate** 模板则不会向通过身份验证的用户的呈现相应的页面。开发人员可以通过编写 **AnonymousTemplate** 模板和 **LoggedInTemplate** 模板进行不同用户的样式呈现，示例代码如下所示。

```
<body>
  <form id="form1" runat="server">
    <div>
      <asp:LoginView ID="LoginView1" runat="server">
        <LoggedInTemplate>
          这是一个登录用户可以访问的页面..
        </LoggedInTemplate>
        <AnonymousTemplate>
          这是一个匿名用户可以访问的页面..
        </AnonymousTemplate>
      </asp:LoginView>
    </div>
  </form>
</body>
```

上述代码为不同权限的用户配置了不同的模板，当不同权限的用户访问页面时，其看到的页面样式也是不同的。在 **LoginView** 控件中，还能够为不同权限和身份的用户配置不同的模板，开发人员能够为不同的用户分配不同的角色。当用户被分配了不同的角色后，用户能够通过相应的角色访问相应的模板，例如普通用户可以访问普通用户模板，**VIP** 用户可以访问 **VIP** 模板而管理员可以访问管理员模板。

在 **LoginView** 控件中，单击 **RoleGroup** 集合，可以添加相应的 **LoginView** 控件的 **RoleGroup** 集合，如图 6-6 所示。



图 6-6 添加 RoleGroup 集合

这里添加了两个 **RoleGroup** 集合，该 **RoleGroup** 集合分别包含 **admin** 和 **VIP** 两种用户类别，当用户为 **admin** 或 **VIP** 是，可以通过相应的权限绑定进行不同模板的访问，创建后示例代码如下所示。

```
<asp:LoginView ID="LoginView1" runat="server">
  <RoleGroups>
    <asp:RoleGroup Roles="admin">
      <ContentTemplate>
        这是一个管理员用户可以访问的页面..
      </ContentTemplate>
    </asp:RoleGroup>
```

```
<asp:RoleGroup Roles="vip">
    <ContentTemplate>
        这是一个 VIP 用户可以访问的页面
    </ContentTemplate>
</asp:RoleGroup>
</RoleGroups>
<LoggedInTemplate>
    这是一个登录用户可以访问的页面..
</LoggedInTemplate>
<AnonymousTemplate>
    这是一个匿名用户可以访问的页面..
</AnonymousTemplate>
</asp:LoginView>
```

当有不同身份的用户访问该控件时，控件能够通过用户的身份进行不同模板的呈现，这样就方便了开发人员对不同身份和权限的用户进行网站应用程序和模板的访问限制了。

注意：当一个用户拥有的身份或权限不在列表的权限中时，该用户会默认访问 **LoggedInTemplate** 模板，并且无论是 **LoggedInTemplate** 模板还是 **RoleGroup** 模板，都不会对匿名用户呈现。

## 6.1.4 登录状态控件（LoginStatus）

登录状态控件（**LoginStatus**）用于显示用户验证时的状态，**LoginStatus** 包括“登录”和“注销”两种状态，对于 **LoginStatus** 控件的状态是由相应的 **Page** 对象的 **Request** 属性中的 **IsAuthenticated** 属性进行决定。开发人员能够直接将 **LoginStatus** 控件拖放在页面中，从而让用户能够通过相应的状态进行登录或注销操作，**LoginStatus** 控件默认 HTML 代码如下所示。

```
<asp:LoginStatus ID="LoginStatus1" runat="server" />
```

上述代码就呈现了一个 **LoginStatus** 控件，**LoginStatus** 控件默认的呈现形式是以文本的形式呈现的，如图 6-7 所示。

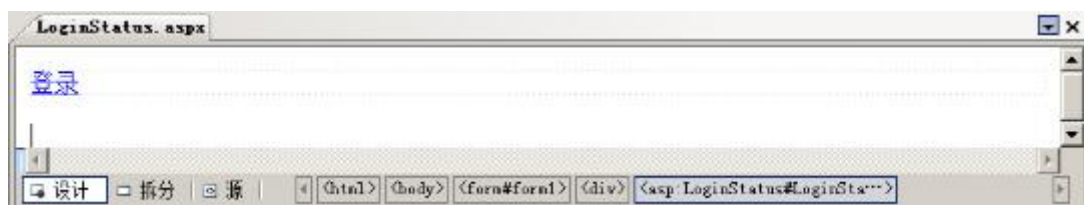


图 6-7 LoginStatus 控件呈现形式

正如图 6-7 所示，**LoginStatus** 控件默认的呈现形式是以文本的形式呈现的。当用户没有在网上进行登录操作时，该控件会呈现登录字样给用户以便用户进行登录操作，当用户登录后，**LoginStatus** 控件会为用户提供注销字样以便用户进行注销操作。开发人员还能够为 **LoginStatus** 控件指定以图片形式进行登录和注销，**LoginStatus** 控件常用的属性如下所示。

- ❑ **LoginImageUrl**: 设置或获取用于登录连接的图像 URL。
- ❑ **LoginText**: 设置或获取用于登录连接的文本。
- ❑ **LogoutAction**: 设置或获取一个值用于用户从网站注销时执行的操作。
- ❑ **LogoutImageUrl**: 设置或获取一个值用于登出图片的显示。
- ❑ **LogoutPageUrl**: 设置或获取一个值用于登出连接的图像 URL。
- ❑ **LogoutText**: 设置或一个值用于登出连接的文本。
- ❑ **TagKey**: 获取 **LoginStatus** 控件的 **HtmlTextWriterTag** 的值。

开发人员可以配置 **LoginImageUrl** 以及 **LogoutImageUrl** 属性进行登录、登出的图片显示，使用图片进行登录登出操作能够提高用户体验，示例代码如下所示。

```
<body>
```

```
<form id="form1" runat="server">
<div>
    <asp:LoginStatus ID="LoginStatus1" runat="server" LoginImageUrl="~/login.jpg"
        LogoutImageUrl="~/logout.jpg" />
</div>
</form>
</body>
```

上述代码指定了当用户没有登录时，相应的登录操作以图片的形式呈现在页面中，同样当用户登录后，注销操作也会以图片的形式呈现在页面中，如图 6-8 所示。



图 6-8 图片形式呈现

**LoginStatus** 控件还包括两个常用事件，这两个事件分别为 **LoggingOut** 和 **LoggedOut**。当用户单击注销按钮时会触发 **LoggingOut** 事件，开发人员能够在 **LoggingOut** 事件中编写相应的事件以清除用户的身份信息，这些信息包括 **Session**、**Cookie** 等。开发人员还能够在 **LoggedOut** 事件中规定在用户离开网站时所必须执行的操作。

6.1.5 密码恢复控件（PasswordRecovery）

当用户进行 **Web** 应用程序访问时，在有些情况下会丢失用户密码，这样就需要通过 **Web** 应用程序恢复自己的密码。在应用程序开发中，为了提高系统的安全性和用户信息的私密性，开发人员常常需要编写诸多代码来保存用户的信息并进行用户请求的检测。**ASP.NET** 中提供了密码恢复控件以便开发人员能够在 **Web** 应用中轻松的能够让用户自行进行密码回复。

开发人员能够拖动一个 **PasswordRecovery** 控件在页面中，系统能够在主窗口中创建一个 **PasswordRecovery** 控件所必须的声明，示例代码如下所示。

```
<asp:PasswordRecovery ID="PasswordRecovery1" runat="server">
</asp:PasswordRecovery>
```

开发人员能够使用 **PasswordRecovery** 控件进行相应的配置，包括自动套用格式、视图配置、转换成模板以及网站管理等，如图 6-9 所示。

对于 **PasswordRecovery** 控件而言，开发人员能够单击 **PasswordRecovery** 控件的属性进行相应的配置，例如选择自动套用格式，单击【自动套用格式】按钮进行格式的选取，如图 6-10 所示。



图 6-9 默认的 PasswordRecovery 控件

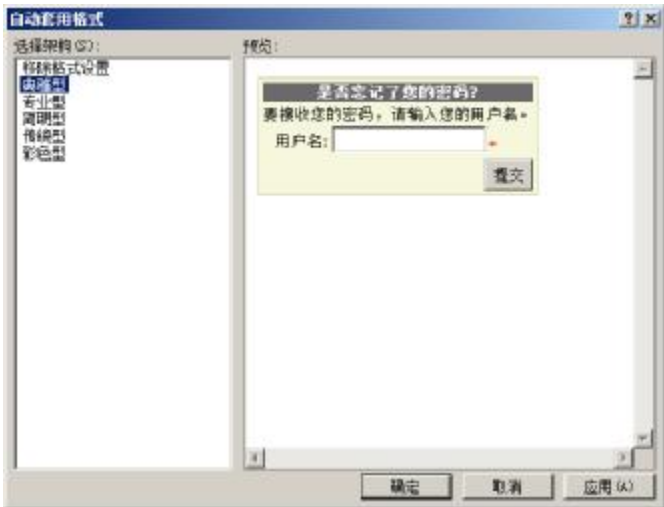


图 6-10 选择默认格式

开发人员可以选择自动套用格式进行模板的编写，以提高用户体验，开发人员还能够自行编写模板进



行 **PasswordRecovery** 控件的样式控制，选择相应的样式后，系统会自行生成样式控制代码，示例代码如下所示。

```
<asp>PasswordRecovery ID="PasswordRecovery1" runat="server" BackColor="#F7F7DE"
    BorderColor="#CCCC99" BorderStyle="Solid" BorderWidth="1px"
    Font-Names="Verdana" Font-Size="10pt">
    <TitleTextStyle BackColor="#6B696B" Font-Bold="True" ForeColor="#FFFFFF" />
</asp>PasswordRecovery>
```

开发人员能够通过修改上面的颜色进行样式控制。在 **PasswordRecovery** 控件中，除了能够自动套用和开发 **PasswordRecovery** 控件的格式外，开发人员还能够为 **PasswordRecovery** 控件相应的功能进行样式控制。**PasswordRecovery** 控件包括三个基本功能，分别为【用户名】、【密码提示问题】和【成功模板】。

在用户使用 **PasswordRecovery** 控件进行密码恢复时，首先需要输入用户名进行用户名的匹配。如果用户名匹配后 **PasswordRecovery** 控件要求用户进行问题答案的填写。如果答案正确，**PasswordRecovery** 控件能够为用户显示【成功模板】。

开发人员能够分别为三个功能进行模板创建。在默认情况下，开发人员不能够进行模板的编辑，开发人员可以选择 **PasswordRecovery** 控件中【管理】菜单中的【转换为模板】选项进行相应的模板转换，如图 6-11 所示。

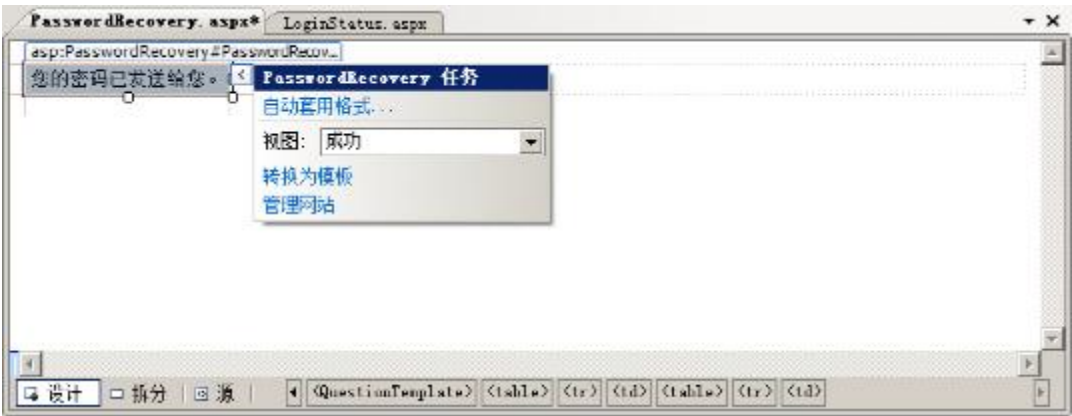


图 6-11 转换为模板

当转换为模板之后，开发人员就能够在模板中编写相应的文档或样式控制提高用户体验的友好度。在编写相应的模板后，该控件中的三个功能会分别被生成为模板形式而存在，示例代码如下所示。

```
<QuestionTemplate>
    <table border="0" cellpadding="1" cellspacing="0" style="border-collapse: collapse;">
        <tr>
            <td>
                <table border="0" cellpadding="0">
                    <tr>
                        <td align="center" colspan="2" style="color: White; background-color: #6B696B; font-weight: bold;">
                            标识确认</td>
                    </tr>
                    <tr>
                        <td align="center" colspan="2">要接收您的密码，请回答下列问题。只有当填写了相应的问题后，
                            您的用户密码才能够被恢复</td>
                    </tr>
                    <tr>
                        <td align="right">用户名:</td>
                        <td>
                            <asp:Literal ID="UserName" runat="server"></asp:Literal>
                        </td>
                    </tr>
                    <tr>
                        <td align="right">问题:</td>
                        <td>
                            <asp:Literal ID="Question" runat="server"></asp:Literal>
                        </td>
                    </tr>
                </table>
            </td>
        </tr>
    </table>
```

```
<tr>
  <td align="right">
    <asp:Label ID="AnswerLabel" runat="server" AssociatedControlID="Answer">答案:</asp:Label>
  </td>
  <td>
    <asp:TextBox ID="Answer" runat="server"></asp:TextBox>
    <asp:RequiredFieldValidator ID="AnswerRequired" runat="server"
      ControlToValidate="Answer" ErrorMessage="需要答案。"
      ToolTip="需要答案。" ValidationGroup="PasswordRecovery1">*</asp:RequiredFieldValidator>
  </td>
</tr>
<tr>
  <td align="center" colspan="2" style="color: Red;">
    <asp:Literal ID="FailureText" runat="server" EnableViewState="False"></asp:Literal>
  </td>
</tr>
<tr>
  <td align="right" colspan="2">
    <asp:Button id="SubmitButton" runat="server" commandname="Submit"
      text="提交" validationgroup="PasswordRecovery1" />
  </td>
</tr>
</table>
</td>
</tr>
</table>
</QuestionTemplate>
```

上述代码实现了【提问模板】中的模板信息和样式，当用户进入提问功能时会呈现该模板。当用户输入用户名时，系统会查找相应的用户信息并跳转到提问页面。如果用户回答自己提问的问题并回答正确后，**PasswordRecovery** 控件会将密码发送到相应的邮箱中，而如果用户回答出错，**PasswordRecovery** 控件就保留密码，以提高系统的安全性。

### 6.1.6 密码更改控件（ChangePassword）

在应用程序开发中，开发人员需要编写密码更改控件让用户能够快速的进行密码更改。在应用程序的使用中，用户会经常需要更改密码，更改密码有很多的可能性。例如用户进行登录后发现自己的用户信息可能被其他人改动过，就有可能怀疑密码泄露的问题，这样用户就可以通过更改密码进行密码的更换。另外，如果用户在注册时的密码是系统自动生成的密码，用户同样需要在密码更改控件中修改生成的密码以便用户记忆。

在 **ASP.NET** 中提供了密码更改控件以便开发人员能够轻易的完成密码更改功能。拖放一个密码更改控件在页面，系统会自动生成相应的 **HTML** 代码，示例代码如下所示。

```
<asp:ChangePassword ID="ChangePassword1" runat="server">
</asp:ChangePassword>
```

**ChangePassword** 控件包括密码、新密码和确认新密码，如图 6-12 所示。



图 6-12 ChangePassword 控件

当用户需要更改密码时，用户必须先填写旧密码进行密码的验证，如果用户填写的旧密码是正确的密码，则系统会将新密码替换旧密码以便用户下次登录时使用新密码。如果用户填写的旧密码不正确，则系统会认为可能是一个非法用户而不允许更改密码。**ChangePassword** 控件同样允许开发人员自动套用格式或者通过编写模板进行 **ChangePassword** 控件的样式布局，如图 6-13 所示。



图 6-13 自动套用格式

开发人员能够自动套用格式进行更改密码控件的呈现，不仅如此，开发人员还能够单击右侧的功能导航进行模板的转换，转换成模板后开发人员就能够进行模板的自定义。**ChangePassword** 控件可以使用 **Web.config** 中的 **membership** 配置节进行成员资格配置，所以 **ChangePassword** 控件能够实现不同场景的不同功能，这些功能如下所示。

- ❑ 用户登录情况：开发人员能够使用 **ChangePassword** 控件允许用户在不登录的情况下进行密码的更改。
- ❑ 更改用户密码：开发人员能够使用 **ChangePassword** 控件让一个登录的用户进行另一个用户的密码的更改。

在 **ChangePassword** 控件中，开发人员可以通过配置 **ChangePassword** 控件的相应属性进行 **ChangePassword** 控件的样式或者是功能的设置，这样能够保证在一定的安全范围内进行安全的用户信息操作。**ChangePassword** 控件常用的属性如下所示。

- ❑ **CancelButtonImageUrl**：配置取消按钮控件的图片文本，该属性可以为按钮控件指定一个图片按钮进行呈现。
- ❑ **CancelButtonStyle**：配置取消按钮控件的样式和外观的属性集。
- ❑ **ChangePasswordButtonType**：配置更改密码控件的类型。
- ❑ **ChangePasswordFailureText**：配置更改密码失败时所呈现的错误信息。
- ❑ **ConfirmNewPassword**：获取用户输入的重复密码的值。
- ❑ **ConfirmPasswordCompareErrorMessage**：当用户输入密码和输入验证密码出现错误时提示的错误消

息。

- ❑ **ConfirmPasswordRequiredErrorMessage:** 当用户没有输入“确认新密码”时在控件中提示的错误消息。
- ❑ **ContinueButtonImageUrl:** 为继续按钮配置一个图片文本，该属性可以为按钮控件指定一个图片按钮进行呈现。
- ❑ **ContinueButtonStyle:** 为继续按钮配置样式或属性集。

开发人员能够配置相应的 **ChangePassword** 控件的属性进行不同的 **ChangePassword** 控件的样式呈现，以及功能实现。在 **ChangePassword** 控件中，有许多属性都是包括按钮或表格的样式的呈现的属性，这里就不再一一列举。

## 6.1.7 生成用户控件（CreateUserWizard）

生成用户控件（**CreateUserWizard**）为 **MembershipProvider** 对象提供了用户界面，使用该控件能够方便的让开发人员在页面中生成相应的用户，同时当用户访问该应用程序时，用户能够通过使用 **CreateUserWizard** 控件的相应的功能进行注册，如图 6-14 所示。



图 6-14 CreateUserWizard 控件

正如图 6-14 所示，**CreateUserWizard** 控件默认包括多个文本框控件以便用户的输入，这里包括用户名、密码、确认密码、电子邮件、安全提示问题和问题答案等项目。其中用户名、密码、确认密码用于身份验证和数据插入为系统提供用户信息，而电子邮件和安全答案用于当用户忘记密码或更改密码时向用户发送相应的邮件以便提高系统身份认证的安全性。

开发人员能够将 **CreateUserWizard** 控件拖放在主窗口中进行页面呈现，这样就能够实现用户注册功能，当开发人员拖动 **CreateUserWizard** 在主窗口中是，系统会自动生成 **HTML** 代码，示例代码如下所示。

```
<asp:CreateUserWizard ID="CreateUserWizard1" runat="server">
  <WizardSteps>
    <asp:CreateUserWizardStep runat="server" />
    <asp:CompleteWizardStep runat="server" />
  </WizardSteps>
</asp:CreateUserWizard>
```

上述代码创建了一个 **CreateUserWizard** 控件进行用户注册功能的实现，开发人员还能够为 **CreateUserWizard** 控件中相应的模板进行样式控制。例如当用户注册完毕后，用户会跳转到一个页面提示“账户注册完毕，请登录”等等，这样就能提高用户体验。单击【自定义完成步骤】按钮或在快捷窗口下拉菜单中选择【完成】选项就能够进行完成模板的实现，如图 6-15 所示。



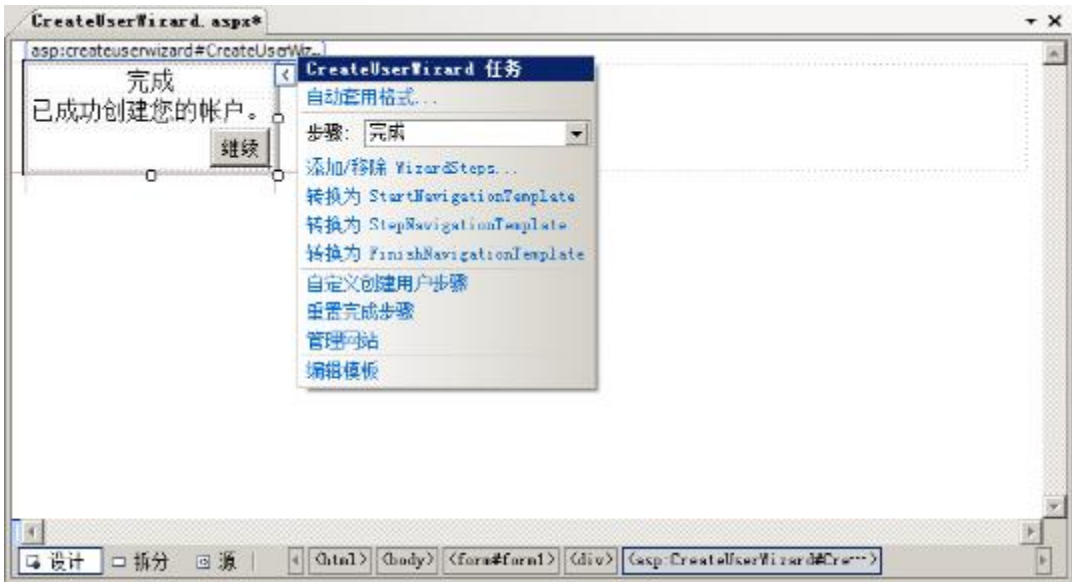


图 6-15 完成模板的编写

开发人员能够在完成步骤中编辑模板以便能够进行更多的提示和更好的用户体验，编辑完成模板后，系统会自动更改相应的代码，示例代码如下所示。

```
<asp:CreateUserWizard ID="CreateUserWizard1" runat="server" ActiveStepIndex="1">
  <WizardSteps>
    <asp:CreateUserWizardStep runat="server" />
    <asp:CompleteWizardStep runat="server">
      <ContentTemplate>
        <table border="0">
          <tr>
            <td align="center">恭喜您！注册完毕！</td>
          </tr>
          <tr>
            <td>已成功创建您的账户，请登录。</td>
          </tr>
          <tr>
            <td align="right">
              <asp:Button ID="ContinueButton" runat="server" CausesValidation="False"
                CommandName="Continue" Text="继续" ValidationGroup="CreateUserWizard1" />
            </td>
          </tr>
        </table>
      </ContentTemplate>
    </asp:CompleteWizardStep>
  </WizardSteps>
</asp:CreateUserWizard>
```

上述代码创建了一个完成注册的模板，开发人员还可以通过编写自定义创建用户模板以便更加方便的创建用户。**CreateUserWizard** 控件还包括其他模板，这些模板能够方便开发人员进行更高的页面呈现，这些模板及其说明如下所示。

- ❑ **HeadTemplate**: 获取或设置标题区的模板内容。
- ❑ **SideBarTemplate**: 获取或设置侧边栏的模板内容。
- ❑ **StartNavigationTemplate**: 获取或设置起始步骤中导航区域的模板内容。
- ❑ **StepNavigationTemplate**: 获取或设置不同步骤中导航区域的模板内容。
- ❑ **FinishNavigationTemplate**: 获取或设置结束步骤中导航区域的模板内容。
- ❑ **ContentTemplate**: 获取或设置在创建用户模板和完成模板中的模板内容。

开发人员还能够通过 **HeadTemplate**、**SideBarTemplate** 等模板进行高级的 **CreateUserWizard** 控件的页面呈现和样式控制，这样不仅能够提高用户体验和友好度，还能够清晰的让用户按照步骤执行操作，降低了错误的出现率。

## 6.2 网站管理工具

在使用高级用户控件时，开发人员需要使用网站管理工具进行相应的控件配置和网站管理，网站管理工具包括安全、应用程序配置和提供应用程序配置。开发人员能够在管理工具中进行用户访问的权限，以及应用程序配置等高级网站管理。

### 6.2.1 启动管理工具

在 ASP.NET 应用程序开发中，通常都是通过手动进行 Web.config 配置文件的更改。在 ASP.NET 应用程序中，系统提供了网站管理工具用于系统的用户、用户权限以及系统的配置的管理，开发人员能够很容易的进行 ASP.NET 应用程序的管理。

在应用程序中，特使需要使用到用户及网站管理的用户控件中，在侧边的快捷操作栏中都会包括一个【网站管理】选项，单击【网站管理】选项能够启动网站管理工具以便能够进行相应的网站管理，如果没有使用相应的控件进行 ASP.NET 网站管理，开发人员可以在导航菜单栏中右击当前项目，在下拉菜单中选择【ASP.NET 配置】选项进行网站管理，如图 6-16 所示。

当开发人员选择【ASP.NET 配置】选项后，Visual Studio 2008 会创建一个虚拟服务器用于管理工具的执行。在管理工具中，开发人员能够配置安全、应用程序配置和提供应用程序配置进行高级的 ASP.NET 应用程序配置。如图 6-17 所示。



图 6-16 选择 ASP.NET 配置

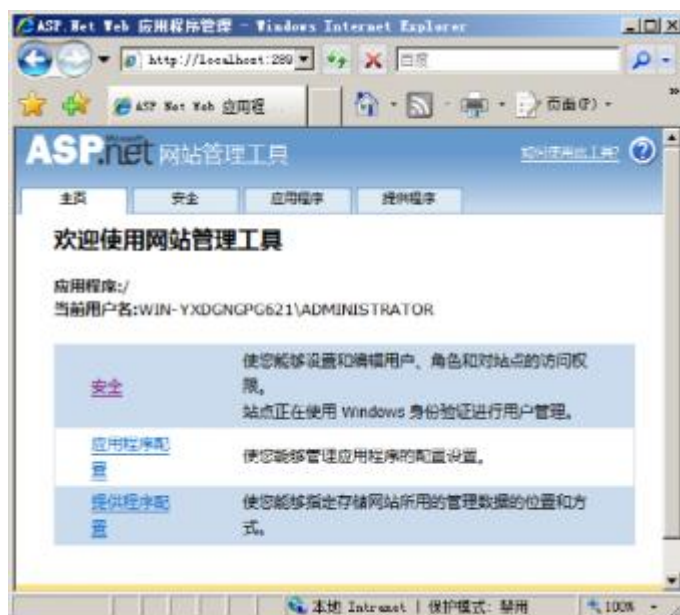


图 6-17 启动网站管理工具

**注意：**在使用 ASP.NET 管理工具进行网站管理时，推荐关闭 Web.config 文件或停止使用该文件，因为管理工具可能会在配置和运行中更改该文件。

### 6.2.2 用户管理

开发人员能够单击【安全】标签进行相应的应用程序安全的管理。安全管理包括用户管理、用户角色管理以及用户的访问规则管理，如图 6-18 所示。

使用网站管理工具能够进行用户管理，用户管理功能仅对表单验证有效。如果当前的验证方式是基于 Windows 默认的身份验证时，则会在用户栏中显式【当前身份验证类型为 Windows，因此禁用了此工具中的用户管理】文本，开发人员能够选择身份验证类型进行身份验证类型的配置，如图 6-19 所示。



图 6-18 用户管理



图 6-19 身份验证类型配置

在身份验证类型配置中，允许开发人员进行用户访问的配置，这里包括两个用户访问配置，这两个用户访问配置如下所示。

- ❑ 通过 Internet: 如果用户将通过公共 Internet 访问该网站时网站，可以选择此选项作为配置。用户需要使用 Web 窗体登录并且站点将使用 Forms 进行身份验证，从而根据存储在数据库中的用户信息来识别用户。
- ❑ 通过本地网络: 如果用户仅通过专用本地网络访问该的网站，可以选择此选项，站点将使用内置的 Microsoft Windows 身份验证来识别用户。

开发人员能够根据应用程序的功能的不同进行不同用户访问的配置，通常情况下可以选择【通过 Internet】选项进行用户的访问配置，单击【完成】按钮后，系统会呈现相应的用户管理信息，如图 6-20 所示。



图 6-20 用户管理

当配置【用户访问】选项为【通过 Internet】选项时，在用户管理中会生成相应的统计和功能，开发人员能够在用户选项卡中选择创建用户或管理用户，并在相应的选项卡中显示用户的统计。

6.2.3 用户角色

在 ASP.NET 应用程序开发中，需要对不同的用户进行用户角色的管理，例如该用户可能是一个学生，也可能是一个学生甚至是一个管理员。使用 ASP.NET 管理工具能够快速的创建用户角色以便管理不同角色的用户。在角色选项卡中选择【启动角色】选项即可启动角色，启动角色后，开发人员就能够在角色中创建和管理角色，单击【创建或管理角色】按钮进行角色管理，如图 6-21 所示。



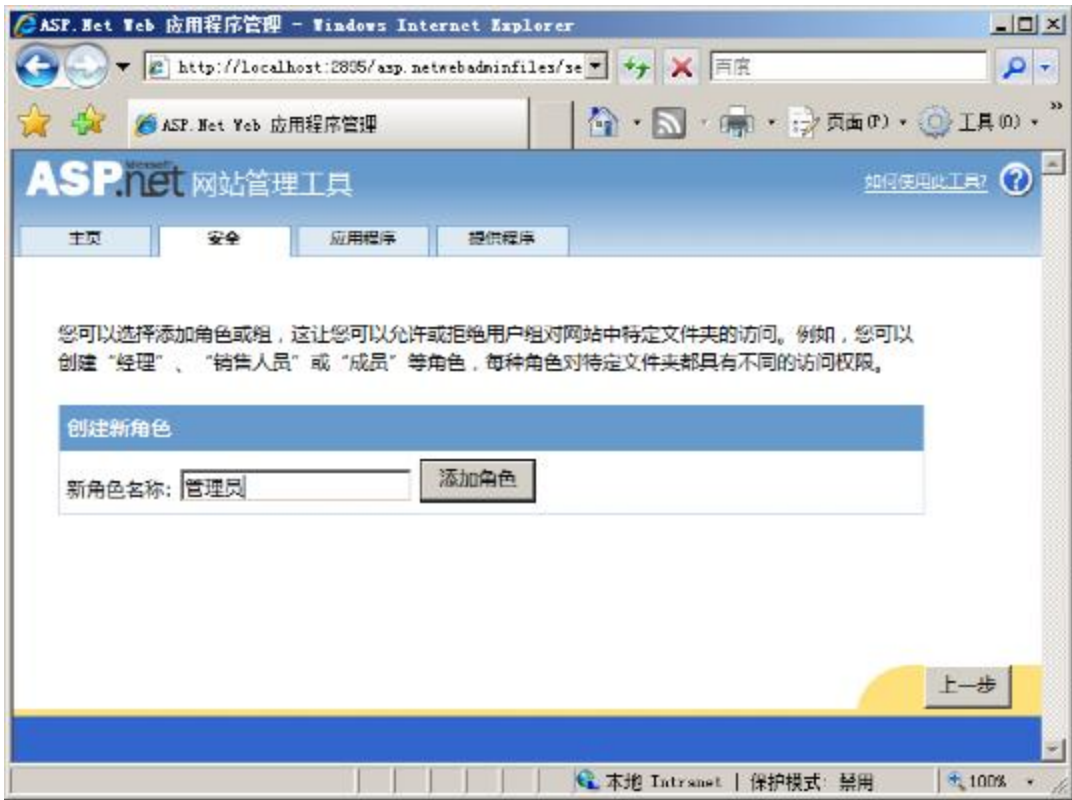


图 6-21 创建新角色

单击【添加角色】按钮就能够在 ASP.NET 应用程序中创建相应的角色，创建的角色可以在用户注册和用户登录时进行选择和管理，如图 6-22 所示。创建完成后，开发人员能够选择相应的用户角色进行用户角色的管理，如图 6-23 所示。



图 6-22 创建角色

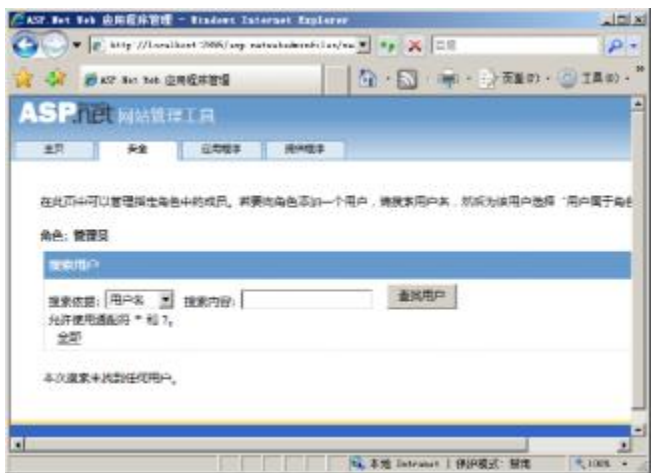


图 6-23 管理角色

开发人员能够为相应角色的用户进行信息的修改和删除。在进行用户管理前，开发人员还能够进行用户的搜索。ASP.NET 网站管理工具支持开发人员使用通配符进行用户搜索和筛选，这样就能够提高用户筛选的效率以便在大量用户前提下进行相应的用户角色的管理。

### 6.2.4 访问规则管理

在 ASP.NET 管理工具中，开发人员能够为用户进行访问规则的管理，这在应用程序开发中是非常必要的。在 ASP.NET 应用程序的开发中，开发人员通常是不允许用户进入到后台管理页面中的，这是非常重要的，一个普通用户无法进入到相应的后台中进行管理。而对于管理员而言，管理员能够进入到后台进行相应的管理。

在应用程序开发中，将管理员和用户分开开发是非常不明智的选择，同样这样开发也会造成应用程序维护困难。在 ASP.NET 管理工具中，可以为相应的用户角色配置相应的访问权限。选择【访问规则】选项卡，开发人员能够创建访问规则和管理访问规则，单击【创建访问规则】按钮可以进行访问规则的创建，如图 6-24 所示。



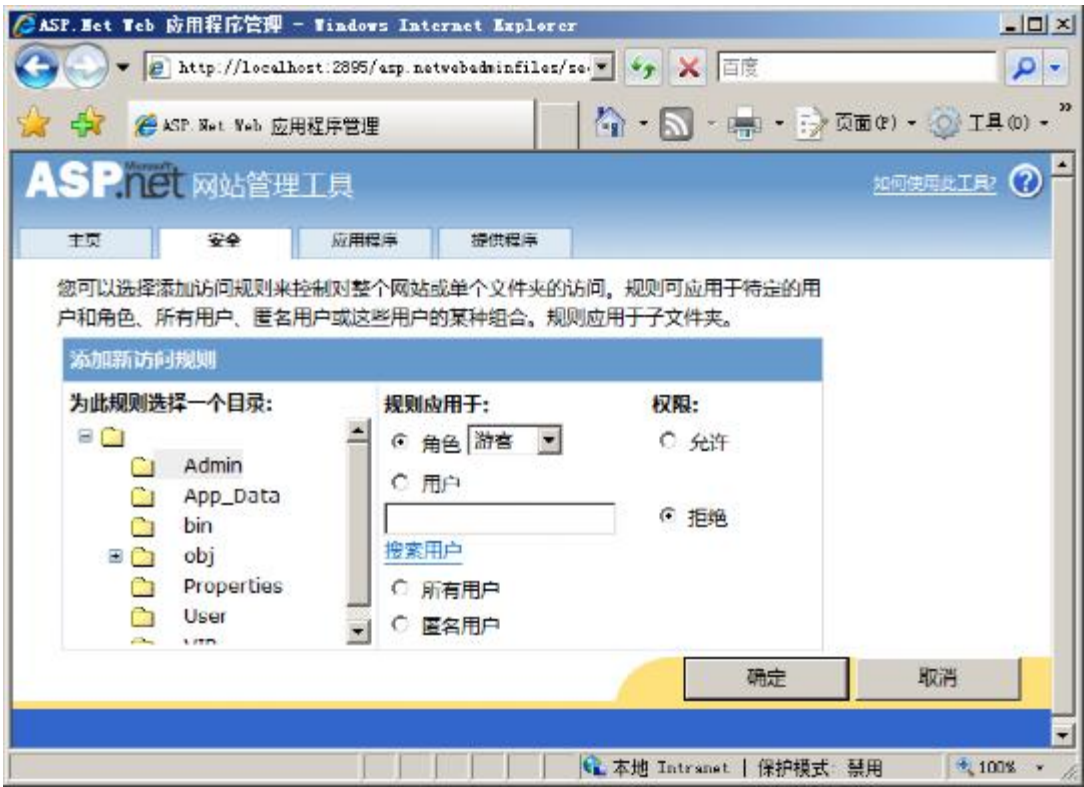


图 6-24 创建访问规则

开发人员能够在访问规则管理器中选择相应的目录进行访问规则的创建，正如图 6-24 所示。首先在左侧选择了 **Admin** 文件夹，由于该文件是一个机密文件夹，所以游客用户是不能够进行访问的。开发人员能够右侧的规则管理中的下拉菜单中选择【游客】选项并在【权限】选项中选择【拒绝】选项以禁止【游客】权限的用户的访问。

开发人员还能够为其他目录如 **VIP**、**User** 等目录进行访问权限的添加，如图 6-25 所示。在添加完成访问规则后，开发人员能够在选项卡中选择【管理访问规则】选项进行访问规则的管理，开发人员可以通过在左侧选择相应的文件夹目录进行访问规则的管理，如图 6-26 所示。



图 6-25 管理访问规则



图 6-26 选择访问规则

当开发人员选择不同的文件夹时，其访问规则呈现的也不同，开发人员能够在访问规则管理面板中删除相应的规则以修改角色的访问权限。

2.6.5 应用程序配置

在 **Web.config** 文件中，开发人员可以手动的进行应用程序管理和配置，在 **ASP.NET** 管理工具中，开发人员可以使用管理工具进行应用程序配置，如图 6-27 所示。



图 6-27 应用程序配置

使用应用程序配置能够创建应用程序设置和管理应用程序设置，创建的应用程序设置会保存在 **Web.config** 文件的 **appSettings** 配置节中，示例代码如下所示。

```
<appSettings>
  <add key="sql" value="0" />
</appSettings>
```

**appSettings** 配置节中的信息能够在应用程序中通过编程进行获取，这样就提高了应用程序的灵活性。除了能够配置 **appSettings** 配置节中的应用程序设置，开发人员还能够通过应用程序管理面板进行 **SMTP** 邮件配置，如图 6-28 和图 6-29 所示。



图 6-28 配置端口

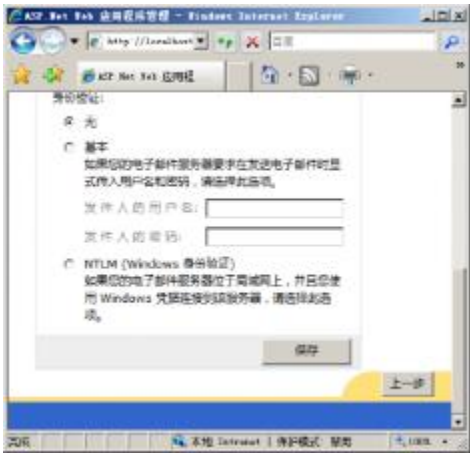


图 6-29 配置邮件

配置邮件后，登录等高级控件就能够通过该配置进行邮件的发送，当用户进行密码更改和密码索取时，相应的控件能够通过邮件配置进行密码和信息的发送。在 **ASP.NET** 应用程序配置中，还能够配置应用程序状态、配置调试和跟踪、定义默认错误页等功能而无需手动修改 **Web.config**，极大的方便了开发人员在 **ASP.NET** 应用程序开发中的应用程序配置，以及系统调配。

### 6.3 使用登录控件

使用登录控件前，需要进行相应的应用程序配置进行登录控件的使用，因为登录控件等高级控件的使用都是基于 **ASP.NET** 应用程序配置而存在的，这些控件不能够独立的运行。在实现相应的操作时，这些控件还需要使用默认的方法和配置信息进行方法操作，登录控件的使用非常简单，这里挑选两个重要的控件进行讲解。

### 6.3.1 生成用户控件（CreateUserWizard）

在用户访问网站时，需要通过注册才能够进行用户信息的保存和获取。在用户注册时，可以使用生成用户控件（**CreateUserWizard**）进行用户注册功能的实现，**CreateUserWizard** 控件 **HTML** 代码如下所示。

```
<body>
  <form id="form1" runat="server">
    <div>
      <asp:CreateUserWizard ID="CreateUserWizard1" runat="server" BackColor="#F7F6F3"
        BorderColor="#E6E2D8" BorderStyle="Solid" BorderWidth="1px"
        Font-Names="Verdana" Font-Size="0.8em">
        <SideBarStyle BackColor="#5D7B9D" BorderWidth="0px" Font-Size="0.9em"
          VerticalAlign="Top" />
        <SideBarButtonStyle BorderWidth="0px" Font-Names="Verdana" ForeColor="White" />
        <ContinueButtonStyle BackColor="#FFFBFF" BorderColor="#CCCCCC"
          BorderStyle="Solid" BorderWidth="1px" Font-Names="Verdana"
          ForeColor="#284775" />
        <NavigationButtonStyle BackColor="#FFFBFF" BorderColor="#CCCCCC"
          BorderStyle="Solid" BorderWidth="1px" Font-Names="Verdana"
          ForeColor="#284775" />
        <HeaderStyle BackColor="#5D7B9D" BorderStyle="Solid" Font-Bold="True"
          Font-Size="0.9em" ForeColor="White" HorizontalAlign="Center" />
        <CreateUserButtonStyle BackColor="#FFFBFF" BorderColor="#CCCCCC"
          BorderStyle="Solid" BorderWidth="1px" Font-Names="Verdana"
          ForeColor="#284775" />
        <TitleTextStyle BackColor="#5D7B9D" Font-Bold="True" ForeColor="White" />
        <StepStyle BorderWidth="0px" />
        <WizardSteps>
          <asp:CreateUserWizardStep runat="server" />
          <asp:CompleteWizardStep runat="server">
            <ContentTemplate>
              <table border="0">
                <tr>
                  <td align="center" colspan="2">恭喜您！注册完毕！</td>
                </tr>
                <tr>
                  <td>已成功创建您的账户，请登录。</td>
                </tr>
                <tr>
                  <td align="right" colspan="2">
                    <asp:Button ID="ContinueButton" runat="server" CausesValidation="False"
                      CommandName="Continue" Text="继续" ValidationGroup="CreateUserWizard1" />
                  </td>
                </tr>
              </table>
            </ContentTemplate>
          </asp:CompleteWizardStep>
        </WizardSteps>
      </asp:CreateUserWizard>
    </div>
  </form>
</body>
```

上述代码在页面中呈现了 **CreateUserWizard** 控件并在页面中进行样式控制。当用户进行注册时，用户可以单击该控件并进行注册操作，运行后如图 6-30 后台 6-31 所示。





图 6-30 创建用户



图 6-31 创建成功

注意：在创建用户时，可能会遇到“密码最短长度为 7,其中必须包含以下非字母数字字符 1”的错误，如果出现这个错误说明密码强度不够，密码中必须包含~!@#\$%^&\*()\_+字符串的一个，如果希望用户能够输入弱密码，可以修改 `minRequiredNonalphanumericCharacters` 的值为 0 即可。

当开发人员再次进入 ASP.NET 网站管理工具中时，开发人员能够在网站管理工具发现这两个用户已经被统计了并且可以为相应的用户进行管理操作，如图 6-32 所示。

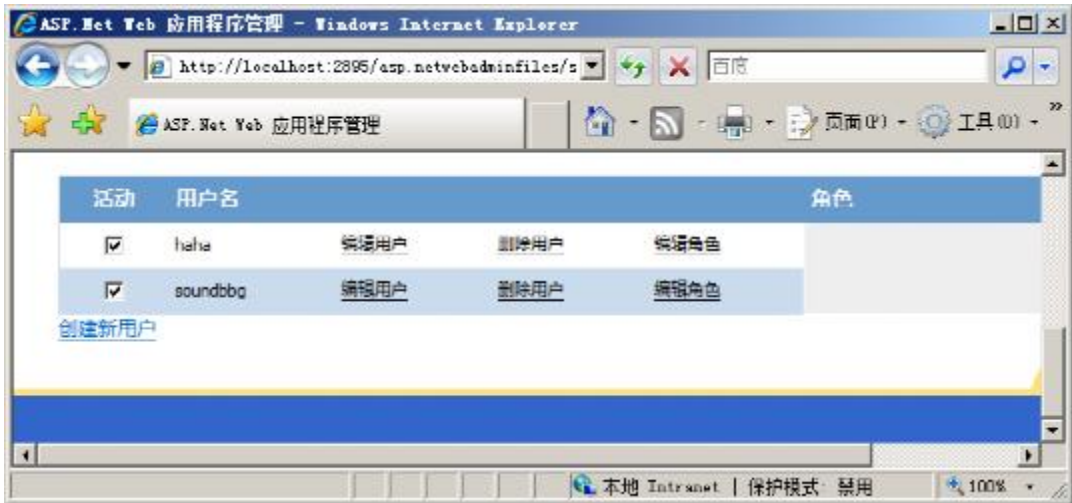


图 6-32 管理工具对用户的管理

在管理工具中，管理员可以对用户进行编辑、删除和角色管理，以便对注册的用户进行更加深入权限划分和信息编辑。

6.3.2 密码更改控件（ChangePassword）

当用户忘记密码后，可以通过使用密码控件进行密码的获取。在使用密码控件进行密码获取时，首先需要输入用户名进行用户身份验证，如图 6-33 所示。验证完成后系统会将相应的用户名匹配的问题呈现在用户界面中，用户需要填写相应的问题进行密码的获取，如图 6-34 所示。





图 6-33 输入用户名



图 6-34 标识确认

用户必须在标识确认前进行用户名的输入，输入用户名后才能够跳转到标识确认模板进行问题的回答，当用户回答正确后，系统将会发送一份邮件到用户的邮箱中提示用户已经找回了相应的密码。

**注意：**在更改密码控件中，必须在 ASP.NET 管理工具中配置 SMTP 邮件发送的相应项用于邮件的发送，否则系统不会发送邮件到用户页面。

## 6.4 小结

本章讲解了 ASP.NET 中的高级控件用于 ASP.NET 应用程序的开发，虽然 ASP.NET 高级控件能够极大的简化开发人员的应用程序开发并通过 ASP.NET 管理工具进行高级控件的配置便开发人员对复杂的应用的开发，但是 ASP.NET 高级控件同样包括一定的局限性，就是不够自主化。在后面的实例章节中会讲解如何通过手动创建一个登录、注册模块，以及如何在项目中使用模块。本章还包括：

- ❑ 启动管理工具：讲解了如何快速的启动管理工具。
- ❑ 访问规则管理：讲解了如何对用户的角色进行访问规则的管理。
- ❑ 应用程序配置：讲解了如何进行应用程序的配置。

在本章中详细讲解的登录控件中，必须在 ASP.NET 管理工具中开启相应的用户访问权限才能够让登录控件良好的运行。在最后一节中，演示了登录控件的使用和运行方法。

## 第三篇 数据操作篇

第 7 章 数据库与 ADO.NET 基础

第 8 章 Web 窗体的数据控件

第 9 章 ASP.NET 操作数据库

第 10 章 访问其他数据源

## 第 7 章 数据库与 ADO.NET 基础

数据库在任何应用程序开发中都非常的重⻱，特别在 ASP.NET 应用程序开发中，数据库通常被用来保存用户的信息、文章内容等数据，同时数据库也能够提供用户进行查询、搜索等操作。传统的纯静态 HTML 页面已经不能满足互联网的发展应用，使用数据库能够让网站与用户、新闻、投票等信息进行良好的整合。

### 7.1 数据库基础

要了解数据库，首先就要掌握数据库基础，数据库就是存放数据的仓库。当开发人员在应用程序的开发中，可以将任何可以抽象成数据的信息存放在数据库中，数据库的特点是数据能够按照数据模型组织进行存取，数据库是高度的结构化并且可以为多个用户共享的。

#### 7.1.1 结构化查询语言

结构化查询语言简称“SQL”，最早的是圣约瑟研究实验室为其关系数据库管理系统 SYSTEM R 开发的一种查询语言。现今的数据库，无论是大型的数据库，如 Oracle、Sybase、Informix、SQL server 这些大型的数据库管理系统，还是 Visual Foxpro、PowerBuilder 这些微机上常用的数据库开发系统，都支持 SQL 语言作为查询语言。

SQL 是高级的非过程化编程语言，允许用户在高层数据结构上工作，它不要求用户指定对数据的存放方法，也不需要用户了解具体的数据存放方式，所以具有完全不同的底层结构的不同数据库系统都可以使用相同的 SQL 语言作为数据输入与管理的接口。它以记录集作为操作对象，所有 SQL 语句接受集合作为输入，返回集合作为输出，这种集合特性允许一条 SQL 语句的输出作为另一条 SQL 语句的输入，所以 SQL 语言可以嵌套，这也使 SQL 语句具有极大的灵活性和强大的功能。在多数情况下，在其他语言中需要一大段程序实现的一个单独事件只需要一个 SQL 语句就可以达到目的，这也意味着用 SQL 语言可以写出非常复杂的语句。下面给出一组例子来演示 SQL 语句的使用方法。

##### 1. 查询表中所有记录

通过使用 select 关键字进行查询，示例代码如下所示。

```
SELECT * FROM NEWS
```

##### 2. 带条件的查询语句

通过使用 where 语句进行带条件的查询，示例代码如下所示

```
SELECT * FROM NEWS WHERE TITLE='新闻'
```

##### 3. 使用函数

语句中也可以使用内置函数，示例代码如下所示。

```
SELECT COUNT(*) AS MYCOUNT FROM NEWS
```

##### 4. 插入数据语句

通过使用 insert 进行插入数据库操作，示例代码如下所示。

```
INSERT INTO NEWS VALUES ('新闻','2008/9/9','新闻内容')
```

##### 5. 删除数据语句

通过使用 **delete** 关键字删除数据库中的数据，示例代码如下所示。

```
DELETE FROM NEWS WHERE ID=1
```

注意：当 **delete** 后面的条件没有限定时，则会删除该表的所有数据。

6. 更新数据语句

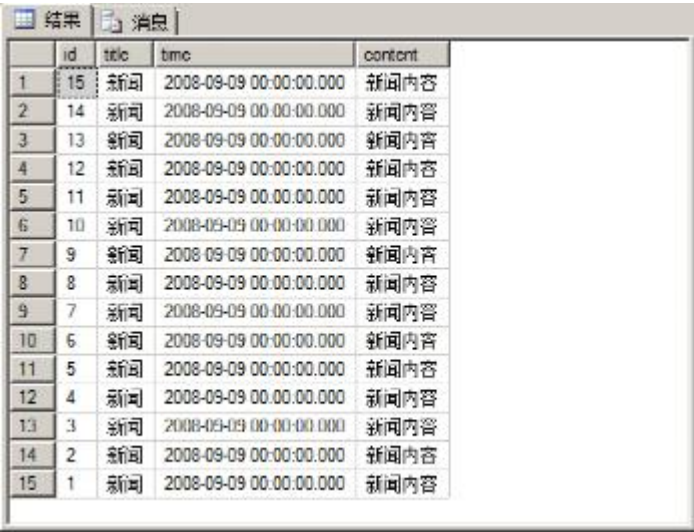
通过使用 **update** 关键字更新数据，示例代码如下所示。

```
UPDATE NEWS SET TITLE='新闻标题' WHERE ID='1'
```

注意：SQL 并不区分大小写，但是推荐使用大写来书写 SQL 语句，这样能够在应用程序中清晰的辨认。

7.1.2 表和视图

表是关系数据库中最主要的数据对象，开发人员通过创建表并向表中进行数据操作来存储和操作数据，表是用来存储和操作数据的一种逻辑结构。表通常以二维表形式呈现，在 **SQL Server Management Studio** 中可以看见表的结构，如图 7-1 所示。



	id	title	time	content
1	15	新闻	2008-09-09 00:00:00.000	新闻内容
2	14	新闻	2008-09-09 00:00:00.000	新闻内容
3	13	新闻	2008-09-09 00:00:00.000	新闻内容
4	12	新闻	2008-09-09 00:00:00.000	新闻内容
5	11	新闻	2008-09-09 00:00:00.000	新闻内容
6	10	新闻	2008-09-09 00:00:00.000	新闻内容
7	9	新闻	2008-09-09 00:00:00.000	新闻内容
8	8	新闻	2008-09-09 00:00:00.000	新闻内容
9	7	新闻	2008-09-09 00:00:00.000	新闻内容
10	6	新闻	2008-09-09 00:00:00.000	新闻内容
11	5	新闻	2008-09-09 00:00:00.000	新闻内容
12	4	新闻	2008-09-09 00:00:00.000	新闻内容
13	3	新闻	2008-09-09 00:00:00.000	新闻内容
14	2	新闻	2008-09-09 00:00:00.000	新闻内容
15	1	新闻	2008-09-09 00:00:00.000	新闻内容

图 7-1 表的表现形式

创建表可以使用 **SQL** 语句进行创建，下面是创建表的表脚本代码。

```
CREATE TABLE [dbo].[news](
    [id] [int] IDENTITY(1,1) NOT NULL,
    [title] [nvarchar](50) NULL,
    [time] [datetime] NULL,
    [content] [ntext] NULL,
)
```

上述代码创建了一个新闻表并且该表具有 4 个字段，这 4 个字段分别为 **id**、**title**、**time** 和 **content**，表是一个具体的表，用于数据的存放和读取。视图不同于表，视图并不是实际存在的表，视图是一种虚拟的表，视图将存在的表中按照一定的规则读取若干列，组成新的结果集，视图在物理上并不存在。当对视图进行操作时，系统会根据视图的定义去操作与视图相关联的基本表。视图有助于隐藏现有的表中的数据，创建视图代码如下所示。

```
CREATE VIEW myview as
SELECT title,[time] from news
```

上述代码创建了一个视图，是基于查询语句 **select title,[time] from news** 所查询的集合的。

注意：视图不是一个表，是一个虚拟的表，视图可以是多个表的集合、筛选形成的新表，视图是这些表的一个结果集。



### 7.1.3 存储过程和触发器

存储过程是一组为了完成特定功能的 **SQL** 语句集，在编写完成后，系统会编译代码并存储在数据库中。用户只需要指定存储过程的名字并给出传递的参数，就可以使用存储过程。存储过程的概念有点像应用程序开发中的方法。

#### 1. 存储过程

存储过程是数据库中一个非常重要的对象，使用好存储过程能够将数据库应用与程序应用相分离。当维护与数据库相关的功能的时候，只需要维护存储过程即可，另外使用存储过程能够提升性能，存储和过程会在运行中被编译，当没有显著的数据更新时，可以直接从编译后的文件中获取相应的结果。存储过程优点如下所示：

- ❑ 存储过程允许标准组件式编程。
- ❑ 存储过程的执行速度较快。
- ❑ 存储过程能够减少网络流量，降低应用程序读取数据库的次数。
- ❑ 存储过程比查询语句更加安全。

存储过程声明语法如下所示：

```
CREATEPROC[EDURE]procedure_name[:number]
    [{@parameterdata_type}
    [VARYING][=default][OUTPUT]
    ][,...n]
    [WITH
    {RECOMPILE|ENCRYPTION|RECOMPILE,ENCRYPTION}
    [FORREPLICATION]
    ASsql_statement[...n]
```

存储过程的各个参数的使用如下所示。

- ❑ **procedure\_name**: 新存储过程的名称，过程名必须符合标识符规则，且对于及其所有者必须惟一。
- ❑ **number**: 是可选的整数，用来对同名的过程分组，以便用一条 **DROPPROCEDURE** 语句即可将同组的过程一起除去。
- ❑ **@parameter**: 过程中的参数。在 **CREATEPROCEDURE** 语句中可以声明一个或多个参数。用户必须在执行过程时提供每个所声明参数的值。
- ❑ **data\_type**: 参数的数据类型。所有数据类型如 **text**、**ntext** 和 **image** 均可以用作存储过程的参数，而与之不同的是，**cursor** 数据类型只能用于 **OUTPUT** 参数。
- ❑ **VARYING**: 指定作为输出参数支持的结果集，其由存储过程动态构造，内容可以变化，**VARYING** 仅适用于游标参数。
- ❑ **default**: 参数的默认值。如果定义了默认值，不必指定该参数的值即可执行过程，默认值必须是常量或 **NULL**，如果过程将对该参数使用 **LIKE** 关键字，那么默认值中可以包含通配符（\*、\_、[]和[^]）。
- ❑ **OUTPUT**: 表明参数是返回参数。该选项的值可以返回给 **EXEC[UTE]**。使用 **OUTPUT** 参数可将信息返回给调用过程。
- ❑ **n**: 表示最多可以指定 **2100** 个参数的占位符。
- ❑ **{RECOMPILE|ENCRYPTION|RECOMPILE,ENCRYPTION}**: **RECOMPILE** 表明 **SQLSERVER** 不会缓存该过程的计划，该过程将在运行时重新编译；**ENCRYPTION** 表示 **SQLSERVER** 加密 **syscomments** 表中包含 **CREATEPROCEDURE** 语句文本的条目；使用 **ENCRYPTION** 可防止将过程作为 **SQLSERVER** 复制的一部分发布。

通过以上参数可以声明一个存储过程，示例代码如下所示。

```
CREATE PROCEDURE UpdatenewsInfo
    @ID int,
    @title nvarchar(50),
    @time datetime,
```

```
@content ntext,
AS
UPDATE [newsInfo]
Set NewsTitle=@title,NewsDatetime=@time
where [ID]=@ID
GO
```

上述代码创建了一个名为“**Updatenewsinfo**”的存储过程，该存储过程作用是修改新闻表中的相应的字段的值。

## 2. 触发器

触发器实际上也是一种存储过程，不过触发器是一种特殊的存储过程，当使用 **UPDATE**、**INSERT** 或 **DELETE** 的一种或多种对指定的数据库的相关表进行操作时，会触发触发器。触发器的语法格式如下所示。

```
CREATE TRIGGER trigger_name
ON { table | view }
[ WITH ENCRYPTION ]
{
  { { FOR | AFTER | INSTEAD OF } { [ INSERT ] [, ] [ UPDATE ] }
  [ WITH APPEND ]
  [ NOT FOR REPLICATION ]
AS
  [ { IF UPDATE ( column )
  [ { AND | OR } UPDATE ( column ) ]
  [ ...n ]
| IF ( COLUMNS_UPDATED ( ) { bitwise_operator } updated_bitmask )
{ comparison_operator } column_bitmask [ ...n ]
} ]
sql_statement [ ...n ]
}
}
```

其中，触发器的各个参数的使用如下所示。

- ❑ **trigger\_name**: 是触发器的名称。触发器名称必须符合标识符规则，并且在数据库中必须惟一，开发人员可以选择是否指定触发器所有者名称。
- ❑ **Table | view**: 是在其上执行触发器的表或视图，有时称为触发器表或触发器视图，可以选择是否指定表或视图的所有者名称。
- ❑ **WITH ENCRYPTION**: 加密 **syscomments** 表中包含 **CREATE TRIGGER** 语句文本的条目。使用 **WITH ENCRYPTION** 可防止将触发器作为 **SQL Server** 复制的一部分发布。
- ❑ **AFTER**: 指定触发器只有在触发 **SQL** 语句中指定的所有操作都已成功执行后才激发，所有的引用级联操作和约束检查也必须成功完成后，才能执行此触发器。
- ❑ **INSTEAD OF**: 指定执行触发器而不是执行触发 **SQL** 语句，从而替代触发语句的操作。
- ❑ **{ [DELETE] [,] [INSERT] [,] [UPDATE] }**: 是指定在表或视图上执行哪些数据修改语句时将激活触发器的关键字，必须至少指定一个选项。在触发器定义中允许使用以任意顺序组合的这些关键字。如果指定的选项多于一个，需用逗号分隔这些选项。
- ❑ **WITH APPEND**: 指定应该添加现有类型的其他触发器，只有当兼容级别是 **65** 或更低时，才需要使用该可选子句。
- ❑ **NOT FOR REPLICATION**: 表示当复制进程更改触发器所涉及的表时，不应执行该触发器。
- ❑ **AS**: 是触发器要执行的操作。
- ❑ **sql\_statement**: 是触发器的条件和操作，触发器条件指定其他准则，以确定 **DELETE**、**INSERT** 或 **UPDATE** 语句是否导致执行触发器操作。

触发器可以包含复杂的 **SQL** 语句，主要用于强制复杂的业务规则或要求。同时，触发器也能够维持数据库的完整性，当执行插入、更新或删除操作时，触发器会根据表与表之间的关系，强制保持其数据的完整性。

## 7.2 使用 SQL Server 2005 管理数据库

SQL Server 2005 是微软继 SQL Server 2000 后 5 年发布的一款新的数据库产品。SQL Server 2005 不仅增加了许多功能，同时也在 UI、管理工具、性能上做了很多的优化。使用 SQL Server 2005 管理网站数据库，不仅提高了开发中数据的存储和读写的效率，也更加方便了数据的管理。

### 7.2.1 初步认识 SQL Server 2005

相比于 SQL Server 2000，SQL Server 2005 在安装上更加的简单，基本上无需手动配置任何事情即可安装。在安装之前，SQL Server 2005 会检查宿主机器的配置是否适合安装 SQL Server 2005，如果机器的配置适合安装 SQL Server 2005，则会进入安装主界面。SQL Server 2005 的安装向导是基于 Windows 的安装程序，用户使用起来更加友好，并且在安装过程中为用户提供了可选方案，让用户选择自己需要的组件安装。

当安装完毕后，用户可以打开 SQL Server 2005 软件体系中的 SQL Server Management 来配置和管理 SQL Server 2005。并进行数据操作。在进入 SQL Server Management 时，对每个连接 SQL Server 2005 都要求一个连接实例，进行身份验证，如图 7-2 所示。

用户可以以 Windows 身份验证的方式登录到 SQL Server 2005 管理工具中，也可以使用 SQL Server 身份验证的方式登录到 SQL Server 2005 管理工具，相比之下，SQL Server 身份验证的方式更加安全。登入后 SQL Server Management 管理工具界面如图 7-3 所示。



图 7-2 SQL Server 2005 身份验证

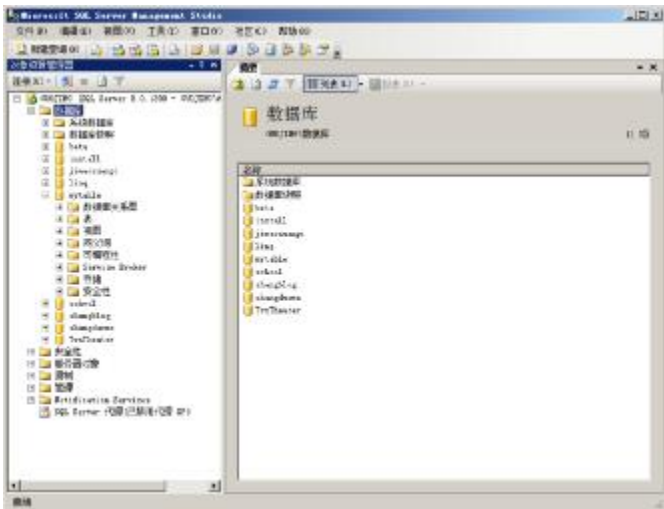


图 7-3 SQL Server Management 管理工具界面

在 SQL Server Management 管理工具中，表的操作与 SQL Server 2000 中并没有太大的差别，但是 SQL Server 2005 中没有了查询分析器，取而代之的是在 SQL Server 2005 中，可以直接在同一个窗口进行查询和数据操作，只需要单击导航栏上的【新建查询】按钮即可，如图 7-3 所示。

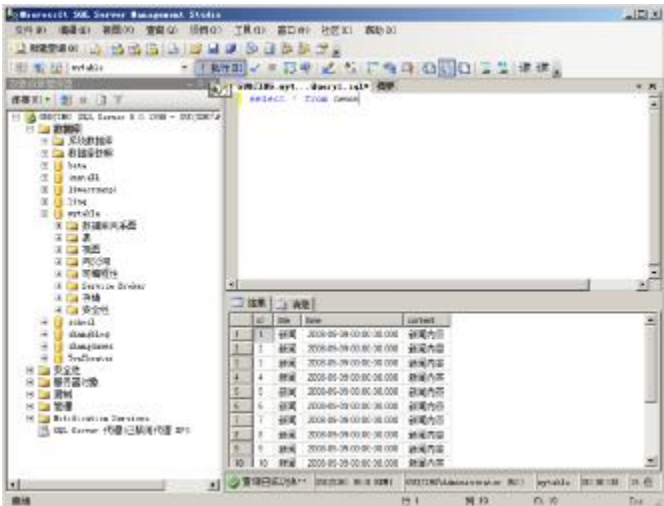


图 7-4 SQL Server Management 进行查询

对于普通的应用而言，SQL Server 2005 与 SQL Server 2000 并没有太大的区别。而对于高级的应，SQL



Server 2005 做了相应的优化，SQL Server 2005 的操作更加友好，在数据的存储等性能上也有较大的提升。

7.2.2 创建数据库

使用 SQL Server Management 管理工具可以快速的创建数据库，在 SQL Server Management 管理工具中左侧的【对象资源管理器】选项中单击【数据库】选项，右击相应数据库，在下拉菜单中选择【新建数据库】。选择后，系统会显示一个创建数据库的向导，如图 7-5 所示。

通常来说，对于一般的应用，只需要填写数据库的名称，而数据和日志逻辑名称系统会自己填写。当有其他需求时，用户也可以更改逻辑名称，以及数据库存放的物理地址。在数据库的创建过程中，可以选择数据库的初始大小，最大值为多少，并且设置增量。当单击【确定】按钮后系统就创建完成数据库“mytable”，如图 7-6 所示。

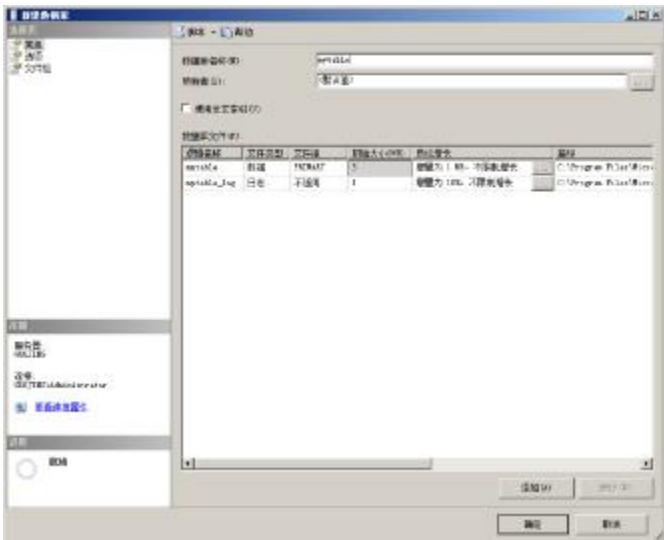


图 7-5 创建数据库



图 7-6 完成数据库的创建

对于任何可以使用 SQL Server Management 管理工具执行的操作，都可以通过 SQL 结构化查询语句来实现，同样，创建表的过程能够通过 SQL 语句来实现，示例代码如下所示。

```
CREATE DATABASE mytable
GO
```

在 SQL Server Management 管理工具中，新建查询，并将上述代码复制到代码块中，单击【执行】按钮，则会创建一个表 mytable。上述代码只是创建了一个简单的没有任何约束或功能的表，在 SQL 语句创建表语句中，使用 ON 子句可以设置数据库文件的属性，ON 子句的参数如下所示。

- ❑ PRIMARY: 设置主文件，ON 子句中只能出现一个 PRIMARY。
- ❑ NAME: 指定文件的逻辑名称。
- ❑ FILENAME: 指定文件的物理路径和名称。
- ❑ SIZE: 指定文件的初始大小。
- ❑ MAXSIZE: 指定文件大小的最大值。
- ❑ UNLIMITED: 指定文件将增长到磁盘变满位置。如果不指定此参数，当文件大小达到了 MAXSIZE 时，将存储为另外一个数据文件。
- ❑ FILEGROWTH: 定义文件的生长量。

当不指定以上参数时，系统会以默认方式创建数据库。若需要通过使用语句来自定义创建数据库，则可以使用 ON 子句并附上参数。示例代码如下所示。

```
CREATE DATABASE mytable
ON
PRIMARY (NAME=table1,
FILENAME='C:\PROGRAM FILES\MICROSOFT SQL SERVER\MSSQL\DATA\MYTABLEDAT1.MDF',
SIZE=100MB,MAXSIZE=200,FILEGROWTH=20)
GO
```



上述代码创建了一个 **mytable** 表，并指定了主文件为 **table1**，文件路径为 **C:\PROGRAM FILES\MICROSOFT SQL SERVER\MSSQL\DATA\MYTABLEDAT1.MDF**，并指定了初始大小为 **100m**，最大大小为 **200m**。

7.2.3 删除数据库

在 **SQL Server Management** 管理工具中，可以直接对数据库进行删除操作。在对象资源管理器中，选中需要删除的数据库，右击选中的数据库，在下拉菜单中选择【删除】选项，**SQL Server Management** 管理工具出现一个删除向导，如图 7-7 所示。

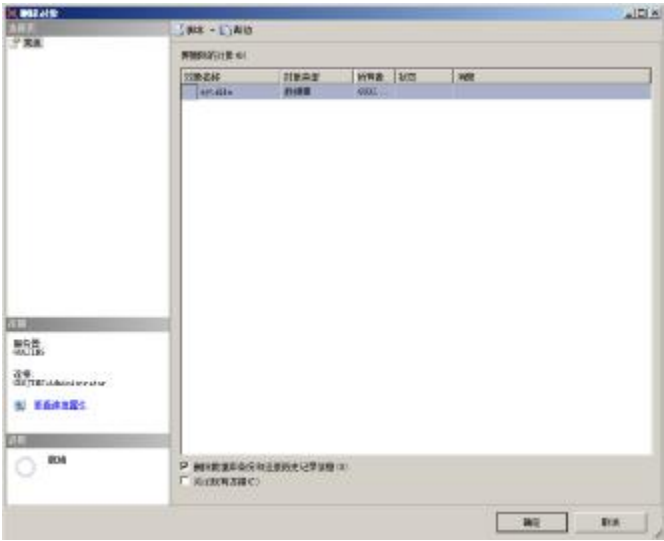


图 7-7 删除数据库

通常情况下，删除功能能够快速并安全的执行删除，但是有的时候，如数据库的连接正在被打开或数据库中的信息正被使用，那么就无法执行删除，必须勾选【关闭现有连接】复选框关闭现有连接。与创建数据库相同的是，删除数据库也可以使用 **SQL** 语句执行，删除数据库的 **SQL** 语法如下所示。

```
DROP DATABASE <数据库名>
```

当需要删除 **mytable** 数据库时，可以编写相应的 **SQL** 删除语句执行删除操作，示例代码如下所示。

```
DROP DATABASE mytable
GO
```

7.2.4 备份数据库

在数据库的使用中，通常会造成一些不可抗力或灾难性的损坏，如人工的操作失误，不小心删除了数据库，或出现了断电等情况，造成数据库异常或丢失。为了避免数据库中重要数据的丢失，就需要使用 **SQL Server Management** 管理工具来备份数据库。

**SQL Server Management** 管理工具备份数据库也非常的简单，在对象资源管理器中选择需要备份的数据库，右击需要备份的数据库，选择【任务】菜单，在【任务】菜单中单击【备份】按钮。单击【备份】按钮后，系统会出现一个备份向导，如图 7-8 所示。

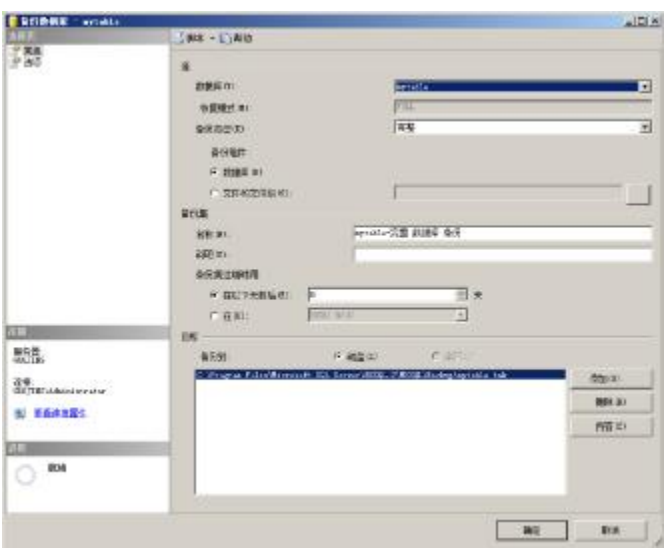


图 7-8 备份数据库

在备份数据库向导中，可以选择相应的备份选项，通常的备份选项有：

- ☐ 数据库：需要备份的数据库。
- ☐ 恢复模式：数据库的恢复模式。
- ☐ 备份类型：数据库的备份类型，通常有完全备份、差异备份、事物日志。
- ☐ 备份组件：通常可选数据库类型和文件类型。
- ☐ 名称：备份的名称。
- ☐ 说明：备份数据库所说的说明。
- ☐ 备份集过期时间：备份集过期的事件，可以设置过期时间。
- ☐ 备份到：选择备份的物理路径，可以选择备份到磁盘或磁带中。

如果有其他的数据库备份需求，则可以选择是备份数据库还是文件和文件组，并且可以配置数据库的备份模式。当配置好备份选项后，单击【确定】按钮，系统会提示备份成功。

7.2.5 还原数据库

当系统数据库出现故障时，就需要还原数据库，还原数据库的文件来自之前备份的数据库。在数据库还原之前，可以先将 mytable 数据库删除，通过还原来恢复数据库。在对象资源管理器中，右击相应的数据库，在下拉菜单中选择【恢复数据库】选项。系统会出现一个还原向导，如图 7-9 所示。

注意：这里的“数据库”是所有数据库的统称的，并不是某个数据库，是数据库的集合。

当还原数据库时，向导会要求用户填写目标数据库。目标数据库可以是一个现有的数据库，也可以是一个新的数据库。在【还原的源】选项中，可以选择【源数据库】选项进行恢复，也可以选择【源设备】选项进行恢复。这里可以选择【源设备】进行恢复，如图 7-10 所示。

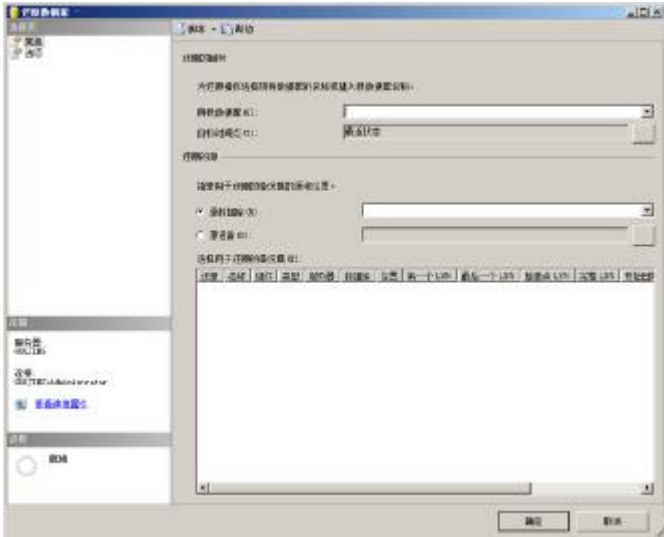


图 7-9 还原数据库



图 7-10 指定备份

单击【添加】按钮选择备份文件，如图 7-11 所示。

备份文件选择完毕后，可以直接单击确定，向导自动完成一些项目的填写，无需用户手动填写，如图 7-12 所示。

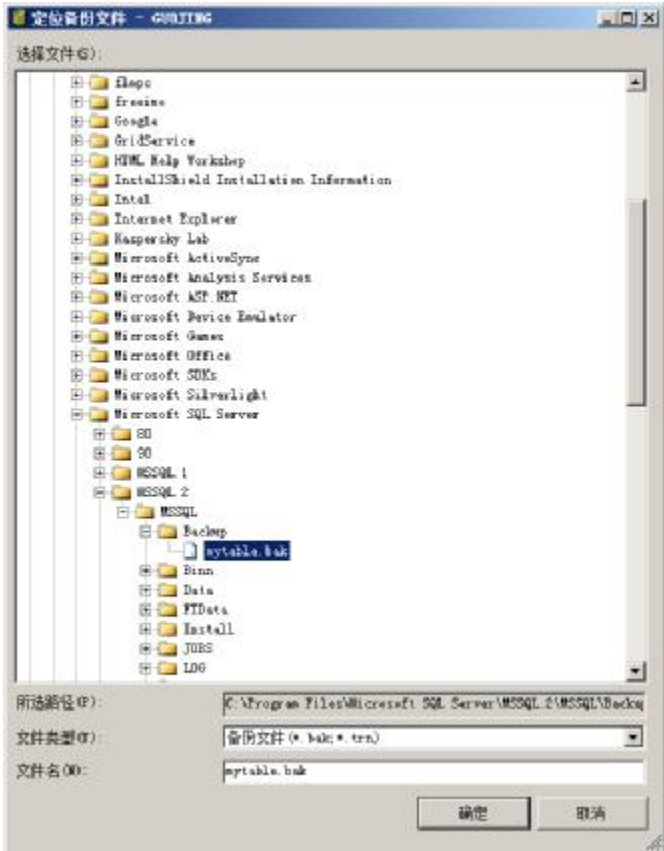


图 7-11 选择备份文件

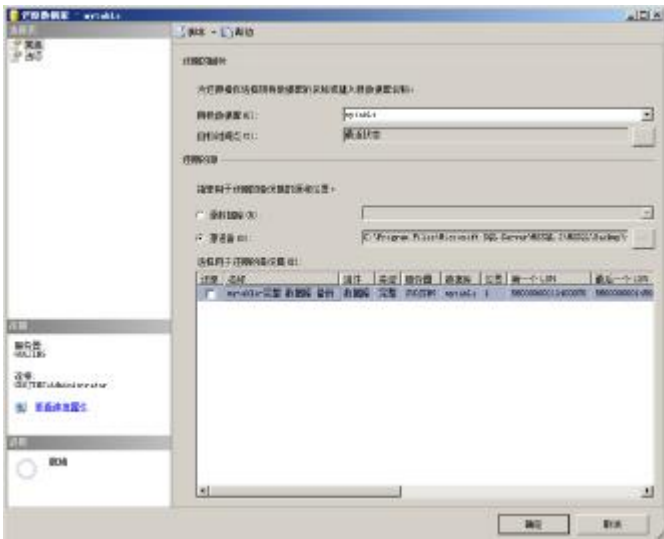


图 7-12 备份向导完成

单击【确定】按钮即可完成数据库的恢复，可以看见在对象资源管理器中，**mytable** 数据库又恢复了。备份数据库是一个非常良好的习惯，因为数据库保存着应用程序的所有信息，一旦数据丢失就会造成无法挽回的影响或亏损，经常备份数据库能够在数据丢失时进行数据的恢复，将应用程序的影响降低到最小。

7.2.6 创建表

在创建了数据库之后，就需要创建表来保存数据，**SQL Server Management** 管理工具可以可视化的为用户创建表操作。在定义表的结构中，需要说明表由哪些列组成，并且需要指定这些列的名称和数据类型。通过 **SQL Server Management** 管理工具可以可视化的创建表结构。在对象资源管理器中，选择相应数据库，右击相应的数据库，在下拉菜单中选择【新建表】选项，单击【新建表】按钮，系统会弹出一个新的 **TAB** 窗口，该窗口可以可视化的让用户创建表，如图 7-13 所示。

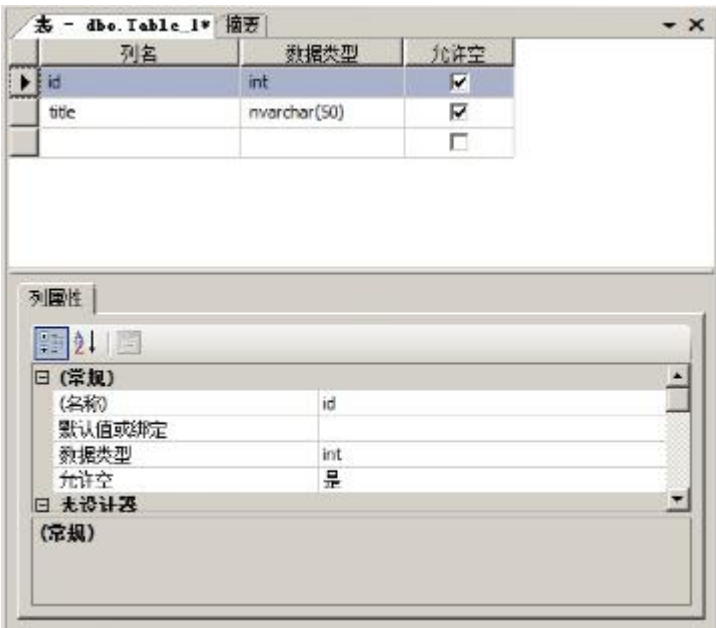


图 7-13 创建表

创建表中的列时，必须指定名称和数据类型。在上述创建表的过程中，创建了 **int** 数据类型的字段 **id** 和 **nvarchar** 数据类型的字段 **title**。

在表的结构中，有的列可以被设置为惟一的标识，如学生表中的学号，当设置了惟一的标识后，此列的数据在表中必须是惟一的、不能重复的。通常情况下，将表中的 **ID** 标识设置为主键。主键可以有效的约束添加到表中的值，被称为主键约束。为了保证约束主键和数据的完整性，定义的主键的字段将不允许插入空值。

**技巧：**在设计器中，在相应的字段上单击右键，选择“设置主键”即可将该字段设置为主键。

在应用程序开发中，通常需要将数据库中的编号设置为主键，通过编号来筛选内容。例如，当开发一个新闻系统中，新闻系统的编号是不应该重复的，所以可以设置为主键。同时，对 **int** 类型的主键可以设置为自动增长，当插入数据时，系统会根据相应的 **id** 号自动增长而不需要通过编程实现。在设计器中，可以为 **int** 类型的字段设置为自动增长，如图 7-14 所示。

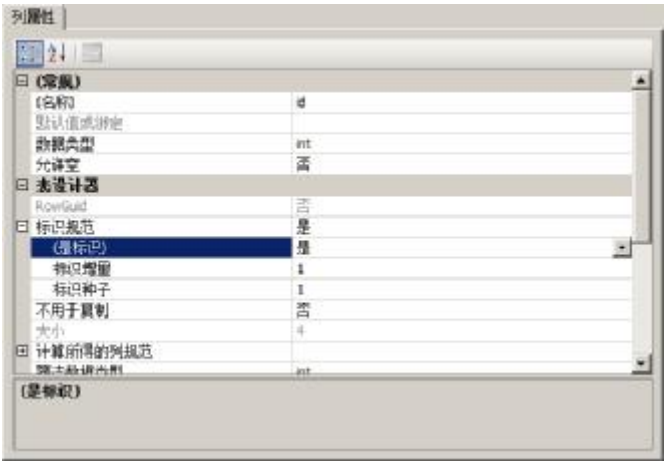


图 7-14 设置自动增长

将相应的字段设置了自动增长时，当插入一条数据，如果该表中没有任何数据，则表中的该字段为 **1**，当再次插入数据时，该字段则会自动增长到 **2**。在应用程序开发中，自动增长的字段经常被使用。

**注意：**当设置了主键，并配置了标识规范，就是配置自动增长，则在编写 SQL 的 **INSERT** 语句时无需向自动增长的列插入数据。

通过 **SQL Server Management** 管理工具可以创建表，也可以通过 **SQL** 语句创建表，创建表的语法结构如下所示。

```
CREATE TABLE 表名
(
    列名 数据类型,
    列名 1 数据类型
    ....
)
```

在创建表中的字段时，也可以使用关键字来约定字段。例如可以使用 **IDENTITY** 关键字来定义一个字段为自动增长列。也可以使用 **PRIMARY KEY** 关键字定义当前列为为主键。同样，当规定用户插入一个列时，必须填写字段，则可以使用 **NOT NULL** 关键字。示例代码如下所示。

```
CREATE TABLE mynews
(
    ID INT IDENTITY PRIMARY KEY,
    TITLE NVARCHAR(50)
)
```

**注意：**当使用语句创建数据库时，必须在导航栏中选择相应的数据库，默认的数据库为 **master**，在执行 **SQL** 语句前需选择相应操作的目标数据库。



7.2.7 删除表

使用 **SQL Server Management** 管理工具能够快速的删除表。在对象资源管理器中，右击相应表，在下拉菜单中选择【删除】选项即可，如图 7-15 所示。

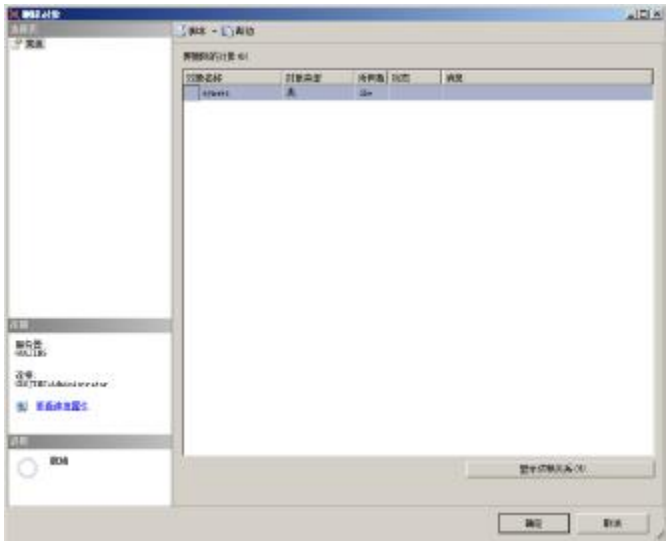


图 7-15 删除表

删除表与前面的删除数据库的操作非常的像，只需单击【确定】按钮即可删除该表。同样，在删除表时，也可以使用 **SQL** 语句删除表，语法结构如下所示。

```
DROP TABLE 表名称
```

当需要删除 **mynews** 表时，可以使用 **DROP** 语句来删除，示例代码如下所示。

```
DROP TABLE mynews
```

7.2.8 创建数据库关系图

在大型关系型数据库中，数据表很多，关系非常复杂。通过关系图，可以很清楚的分析数据库中表的关系。同时，通过这个关系图，你也可以对这些关系进行操作，可以说是一个图形化的关系操作入口。

在 **SQL Server Management** 管理工具中，单击相应的数据库，选择数据库关系图，单击右键，在下拉菜单中选择【新建数据库关系图】选项，系统会提示选择表来创建数据库关系图，如图 7-16 所示。选择需要的数据库中的表，单击【添加】按钮，则会出现关系图，如图 7-17 所示。

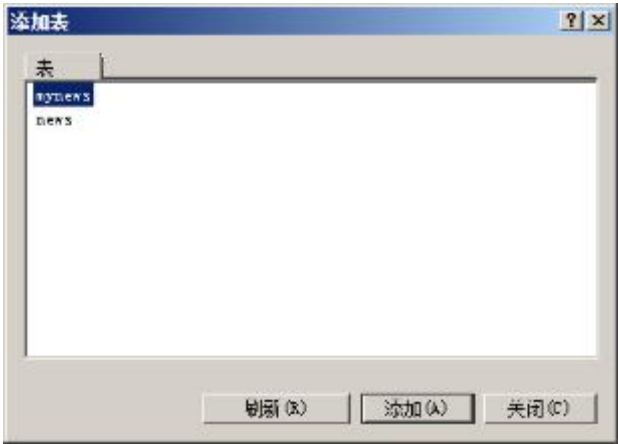


图 7-16 创建数据库关系图

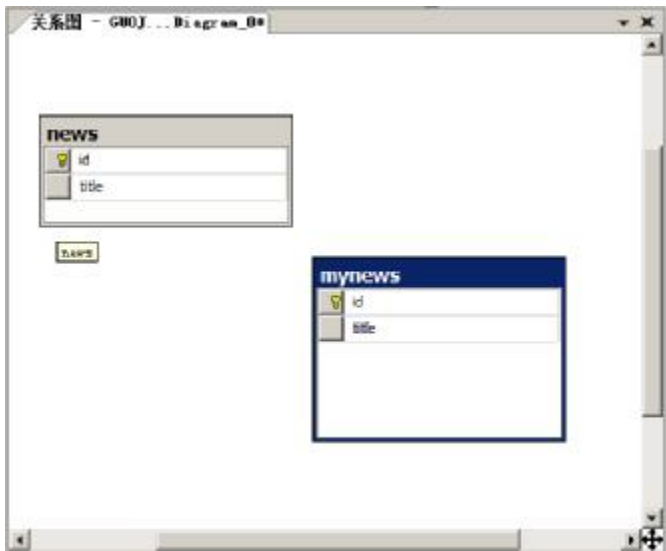


图 7-17 数据库关系图

在数据库关系图中，可以设置表与表之间的约束，可以拖动关系图中的字段，并建立关系，系统会自动建立表和列的关系，如图 7-18 所示。

选择了相应的列值之后，系统会提示填写表和列的规范，来规范外键的约束，建立约束，规范表与表

之间的关系，如图 7-19 所示。

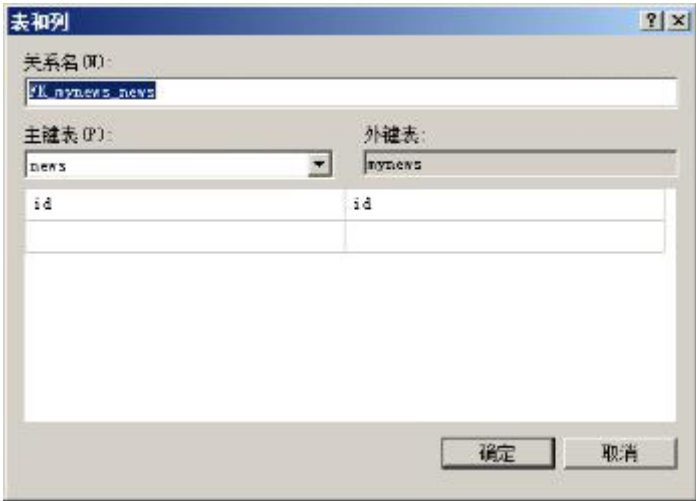


图 7-18 创建约束

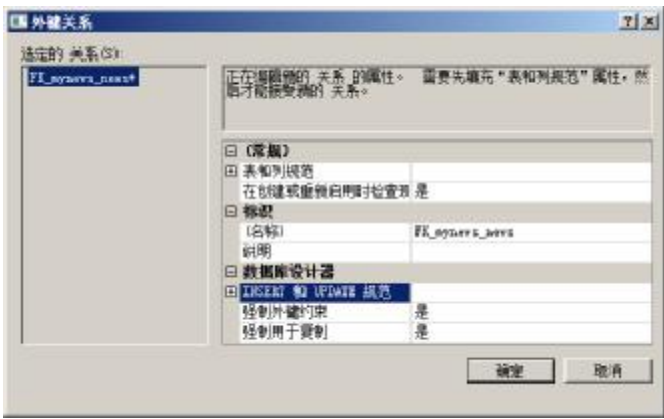


图 7-19 建立规范

单击【确定】按钮即可保存关系图。数据库关系图通过使用外键加强两个表数据之间的连接。外键（FOREIGN KEY）是用于建立和加强两个表数据之间的链接的一列或多列。通过将保存表中主键值的一列或多列添加到另一个表中，可创建两个表之间的链接，这个列就成为第二个表的外键。FOREIGN KEY 约束的主要目的是控制存储在外键表中的数据，但它还可以控制对主键表中数据的修改。

例如，当有一个学生管理系统，这个系统中有一个学生的学号为 20051183049，那么这个学生就会有一系列的数据，包括成绩、选修等。当这个学生毕业，注销该学生，或者这个学生经常不上课，要删除该学生的信息时，如果只删除这个学生的信息，而不删除这个学生的成绩、选修等信息就会会造成其他的表中的数据的完整性被破坏，在这时就需要使用外键约束即在删除学生信息时一同删除学生的其他信息以保证数据库的完整性。另外，通过数据库关系图，也可以良好的表达和操作表与表之间的关系。

### 7.3 ADO.NET 连接 SQL 数据库

ADO.NET 是 .NET Framework 中的一系列类库，它能够让开发人员更加方便的在应用程序中使用和操作数据。在 ADO.NET 中，大量的复杂的数据操作的代码被封装起来，所以当开发人员在 ASP.NET 应用程序开发中，只需要编写少量的代码即可处理大量的操作。ADO.NET 和 C#.NET、VB.NET 不同的是，ADO.NET 并不是一种语言，而是对象的集合。

#### 7.3.1 ADO.NET 基础

ADO.NET 是由微软编写代码，提供了在 .NET 开发中数据库所需要的操作的类。在 .NET 应用程序开发中，C# 和 VB.NET 都可以使用 ADO.NET。

ADO.NET 可以被看作是一个介于数据源和数据使用者之间的转换器。ADO.NET 接受使用者语言中的命令，如连接数据库、返回数据集之类，然后将这些命令转换成在数据源中可以正确执行的语句。在传统的应用程序开发中，应用程序可以连接 ODBC 来访问数据库，虽然微软提供的类库非常的丰富，但是开发过程却并不简单。ADO.NET 在另一方面，可以说简化了这个过程。用户无需了解数据库产品的 API 或接口，也可以使用 ADO.NET 对数据进行了操作。ADO.NET 中常用的对象有：

- ❑ SqlConnection: 该对象表示与数据库服务器进行连接。
- ❑ SqlCommand: 该对象表示要执行的 SQL 命令。
- ❑ SqlParameter: 该对象代表了一个将被命令中标记代替的值。
- ❑ SqlDataAdapter: 该对象表示填充命令中的 DataSet 对象的能力。
- ❑ DataSet: 表示命令的结果，可以是数据集，并且可以同 BulletedList 进行绑定。

通过使用上述的对象，可以轻松的连接数据库并对数据库中的数据进行操作。对开发人员而言，可以

使用 **ADO.NET** 对数据库进行操作，在 **ASP.NET** 中，还提供了高效的控件，这些控件同样使用了 **ADO.NET** 让开发人员能够连接、绑定数据集并进行相应的数据操作。

## 7.3.2 连接 SQL 数据库

**ADO.NET** 通过 **ADOConnection** 连接到数据库。和 **ADO** 的 **Connection** 对象相似的是，**ADOConnection** 同样包括 **Open** 和 **Close** 方法。**Open** 表示打开数据库连接，**Close** 表示关闭数据库连接。在每次打开数据库连接后，都需要关闭数据库连接。

### 1. 建立连接

在 **SQL** 数据库的连接中，需要使用 **.NET** 提供的 **SqlConnection** 对象来对数据库进行连接。在连接数据库前，需要为连接设置连接串，连接串就相当于告诉应用程序怎样找到数据库去进行连接，然后程序才能正确的与 **SQL** 建立连接，连接字串示例代码如下所示。

```
server='服务器地址';database='数据库名称';uid='数据库用户名';pwd='数据库密码';
```

上述代码说明了数据库连接字串的基本格式，如果需要连接到本地的 **mytable** 数据库，则编写相应的 **SQL** 连接字串进行数据库的连接，示例代码如下所示。

```
string strcon; //声明连接字串
strcon = "server=(local);database='mytable';uid='sa';pwd='sa';"; //设置连接字串
```

上述代码声明了一个数据库连接字串，**SqlConnection** 类将会通过此字串来进行数据库的连接。其中，**server** 是 **SQL** 服务器的地址，如果相对于应用程序而言数据库服务器是本地服务器，则只需要配置为 **(local)** 即可，而如果是远程服务器，则需要填写具体的 **ip**。另外，**uid** 是数据库登录时的用户名，**pwd** 是数据库登录时使用的密码。在声明了数据库连接字串后，可以使用 **SqlConnection** 类进行连接，示例代码如下所示。

```
string strcon; //声明连接字串
strcon = "server=(local);database='mytable';uid='sa';pwd='sa';"; //编写连接字串
SqlConnection con = new SqlConnection(strcon); //新建 SQL 连接
try
{
    con.Open(); //打开 SQL 连接
    Label1.Text = "连接数据库成功"; //提示成功信息
}
catch
{
    Label1.Text = "无法连接数据库"; //提示失败信息
}
```

上述代码连接了本地数据库服务器中的 **mytable** 数据库，如果连接成功，则提示“连接数据库成功”，出现异常时，则提示“无法连接数据库”。

**注意：**在使用 **SqlConnection** 类时，需要使用命名空间 **using System.Data.SqlClient**;而连接 **Access** 数据库时，需要使用命名空间 **using System.Data.OleDb**。

### 2. 填充 DataSet 数据集

**DataSet** 数据集表示来自一个或多个数据源数据的本地副本，是数据的集合，也可以看作是一个虚拟的表。**DataSet** 对象允许 **Web** 窗体半独立于数据源运行。**DataSet** 能够提高程序性能，因为 **DataSet** 从数据源中加载数据后，就会断开与数据源的连接，开发人员可以直接使用和处理这些数据，当数据发生变化并要更新时，则可以使用 **DataAdapter** 重新连接并更新数据源。**DataAdapter** 可以进行数据集的填充，创建 **DataAdapter** 对象的代码如下所示。

```
SqlDataAdapter da=new SqlDataAdapter("select * from news",con); //创建适配器
```

上述代码创建了一个 **DataAdapter** 对象并初始化 **DataAdapter** 对象，**DataAdapter** 对象的构造函数允许传递两个参数初始化，第一个参数为 **SQL** 查询语句，第二个参数为数据库连接的 **SqlConnection** 对象。初始化 **DataAdapter** 后，就需要将返回的数据的集合存放到数据集中，示例代码如下所示。



```
DataSet ds = new DataSet(); //创建数据集
da.Fill(ds, "tablename"); //Fill 方法填充
```

上述代码创建了一个 **DataSet** 对象并初始化 **DataSet** 对象，通过 **DataAdapter** 对象的 **Fill** 方法，可以将返回的数据存放到数据集 **DataSet** 中。**DataSet** 可以被看作是一个虚拟的表或表的集合，这个表的名称在 **Fill** 方法中被命名为 **tablename**。

3. 显式 DataSet

当返回的数据被存放到数据集中后，可以通过循环语句遍历和显示数据集中的信息。当需要显示表中某一行字段的值时，可以通过 **DataSet** 对象获取相应行的某一列的值，示例代码如下所示。

```
ds.Tables["tablename"].Rows[0]["title"].ToString(); //获取数据集
```

上述代码从 **DataSet** 对象中的虚表 **tablename** 中的第 0 行中获取 **title** 列的值，当需要遍历 **DataSet** 时，可以使用 **DataSet** 对象中的 **Count** 来获取行数，示例代码如下所示。

```
for (int i = 0; i < ds.Tables["tablename"].Rows.Count; i++) //遍历 DataSet 数据集
{
    Response.Write(ds.Tables["tablename"].Rows[i]["title"].ToString()+"<br/>");
}
```

**DataSet** 不仅可以通过编程的方法来实现显示，也可以使用 **ASP.NET** 中提供的控件来绑定数据集并显示。**ASP.NET** 中提供了常用的显示 **DataSet** 数据集的控件，包括 **Repeater**、**DataList**、**GridView** 等数据绑定控件。将 **DataSet** 数据集绑定到 **DataList** 控件中可以方便的在控件中显示数据库中的数据并实现分页操作，示例代码如下所示。

```
DataList1.DataSource = ds; //绑定数据集
DataList1.DataMember = "tablename";
DataList1.DataBind(); //绑定数据
```

上述代码就能够将数据集 **ds** 中的数据绑定到 **DataList** 控件中。**DataList** 控件还能够实现分页、自定义模板等操作，非常方便开发人员对数据开发。

7.3.3 ADO.NET 过程

从上一节中可以看出，在 **ADO.NET** 中对数据库的操作基本上需要三个步骤，即创建一个连接、执行命令对象并显式，最后再关闭连接。使用 **ADO.NET** 的对象，不仅能够通过控件绑定数据源，也可以通过程序实现数据源的访问。**ADO.NET** 的过程如图 7-20 所示。



图 7-20 ADO.NET 规范步骤

从上图中可以归纳出，**ADO.NET** 的规范步骤如下：

- ❑ 创建一个连接对象。
- ❑ 使用对象的 **Open** 方法打开连接。
- ❑ 创建一个封装 **SQL** 命令的对象。
- ❑ 调用执行命令的对象。
- ❑ 执行数据库操作。



❑ 执行完毕，释放连接。

掌握了这些初步的知识，就能够使用 **ADO.NET** 进行数据库开发。

### 7.4 ADO 与 ADO.NET

**ADO.NET** 相比于 **ADO** 有很大的改进。使用 **ADO.NET**，能够更加容易的进行数据库的开发，其中，一部分是针对开发人员做出的更改，包括易用性、适用性等，其次的更改让 **ADO.NET** 相比于 **ADO**，更加灵活、强大、易于升级使用。

#### 7.4.1 ADO 概述

微软公司的 **ADO** (**ActiveX Data Objects**) 是一个用于存取数据源的 **COM** 组件。它提供了编程语言和统一数据访问方式 **OleDb** 的一个中间层。允许开发人员编写访问数据的代码而不用关心数据库是如何实现的，而只用关心到数据库的连接。访问数据库的时候，关于 **SQL** 的知识不是必要的，但是特定数据库支持的 **SQL** 命令仍可以通过 **ADO** 中的命令对象来执行。

在开发过程中，**ADO** 为 **OleDb** 数据提供给予 **COM** 的应用程序级别的接口，对数据库的操作进行了封装，**ADO** 支持各种开发者的需要。同时，**ADO** 也能够像 **ADO.NET** 一样构建客户端记录集，使用松耦合记录集，并处理 **OleDb** 的数据整形集合。

相比于 **ADO.NET**，**ADO** 还支持一些特殊的方法，例如可滚动的服务器端游标 **MOVENEXT**。但是，使用服务器游标需要使用和保存数据库资源，所以当大量的游标在服务器端被使用时，则可能对应用程序的性能和可缩放性产生极大的负面影响。使用 **ADO**，还需要对防火墙进行配置以启用 **COM** 的发送请求才能够进行数据交互，这样可能造成一定的安全问题。**ADO** 编程模型如下所示：

- ❑ 连接数据源(**Connection**)，可选择开始事务。
- ❑ 可选择创建表示 **SQL** 命令的对象(**Command**)。
- ❑ 可选择指定列、表，以及 **SQL** 命令中的值作为变量参数(**Parameter**)。
- ❑ 执行命令 (**Command**、**Connection** 或 **Recordset**)。
- ❑ 如果命令以行返回，将行存储在存储对象中(**Recordset**)。
- ❑ 可选择创建存储对象的视图以便进行排序、筛选和定位数据(**Recordset**)。
- ❑ 编辑数据。可以添加、删除或更改行、列(**Recordset**)。
- ❑ 在适当情况下，可以使用存储对象中的变更对数据源进行更新(**Recordset**)。
- ❑ 在使用事务之后，可以接受或拒绝在事务中所做的更改。结束事务(**Connection**)。

从上述的编程模型可以看出，**ADO.NET** 在很多方面和 **ADO** 比较相近，但是 **ADO.NET** 并不是 **ADO** 的 **.NET** 版本，**ADO** 和 **ADO.NET** 是两种不同的数据访问方式。

#### 7.4.2 ADO.NET 与 ADO

**ADO.NET** 的名称起源于 **ADO** (**ActiveX Data Objects**)，**ADO** 用于在以往的 **Microsoft** 技术中进行数据的访问。所以微软希望通过使用 **ADO.NET** 名称来向开发人员表明，这是在 **.NET** 编程环境和 **Windows** 环境中优先使用的数据库访问接口。

**ADO.NET** 提供了平台互用性和可伸缩的数据访问，**ADO.NET** 增强了对非连接编程模式的支持，并支持 **RICH XML**。由于传送的数据都是 **XML** 格式的，因此任何能够读取 **XML** 格式的应用程序都可以进行数据处理。事实上，接受数据的组件不一定要是 **ADO.NET** 组件，它可以是基于一个 **Microsoft Visual Studio** 的解决方案，也可以是任何运行在其他平台上的任何应用程序。

可以说传统的 **ADO** 和 **ADO.NET** 是两种不同的数据访问方式，无论是在内存中保存数据，还是打开和

关闭数据库的操作模式都不尽相同。

### 1. 数据在内存中的存在形式

使用 **ADO.NET** 时，数据保存在内存中并且是以 **DataSet** 数据集的形式存放在内存中，而 **ADO** 中，则是以 **RecordSet** 记录集的形式存放在内存中。

在 **ASP** 的开发过程中，连接数据库后，通常将查询操作的数据的集合保存在记录集中，并使用 **MOVENEXT** 等方法进行遍历，也可以使用其他的方法进行查询，提取任意行。而在 **ASP.NET** 中，**ADO.NET** 提供的 **DataRelation** 对象维护有关主记录和详细资料记录的信息，并提供方法使用户可以获取与正在操作的记录相关的记录。

### 2. 数据的表现形式

在 **ADO** 中，记录集的表现形式像一个表。如果需要包含来自多个数据库的表的数据，就必须使用复杂的 **SQL** 语句中的 **JOIN** 查询将各个数据库的表的数据组合到单个记录集中。

而在 **ADO.NET** 中，数据集本身是一个表或多个表的集合。相对于记录集而言，数据集可以保存多个独立的表并维护有关表之间关系的信息。因此，**ADO.NET** 能够维护和保存数据结构复杂的表，模仿数据库的结构，例如表的自关联，以及有一对多或多对多的关系表。

### 3. 数据的连接和断开

**ADO** 是为连接的访问而设计的，相比之下，**ADO.NET** 打开连接的时间仅仅足够数据库的操作。数据集可以将行读入，然后断开与数据库的连接，再对数据集中记录进行更改。当需要将数据集中的资源更新到数据库时，**ADO.NET** 再与数据库连接并更新。

### 4. 数据共享

再 **ADO** 中，如果需要实现 **ADO.NET** 中断开数据连接传送数据的功能，则必须在 **COM** 组件之间互相传送，这样就造成了安全性问题。在杀毒软件中，一些杀毒软件可能会默认禁止 **COM** 组件之间的通信，这样就造成了开发人员的维护困难。而 **ADO.NET** 能够使用 **DataSet** 传送数据而不需要考虑防火墙的限制，是因为 **DataSet** 传送的数据集会被转换成 **XML** 流来传送。**ADO.NET** 相对于 **ADO** 再数据共享上，有如下优点：

- ❑ 突破 **COM** 数据类型的限制：由于 **ADO.NET** 基于 **XML** 流传送数据，所以对数据类型没有限制。
- ❑ 减少数据类型的转换：**ADO.NET** 相对于 **ADO** 而言，减少了大量的数据类型的转换，提高了性能。
- ❑ 可以穿透防火墙：基于 **XML** 流传送数据的方法能够轻松穿透防火墙。

### 5. 构架设计

在构架设计上 **ADO.NET** 与 **ADO** 也是不同的，**ADO.NET** 相对于 **ADO** 更加方便和简洁，从设计的角度上来说，**ADO.NET** 设计的更加完善。

在 **ADO** 中，通过使用 **ADORecordSet** 对象进行数据的连接和操作，**ADORecordSet** 对象是一个庞大的对象，它提供了多种类型的游标能力，例如快速的即时的游标到无连接的客户端游标。但是使用 **ADORecordSet** 对象，很难对数据的操作的方法进行自定义。而在 **ADO.NET** 中，**ADO.NET** 将 **ADO** 中 **ADORecordSet** 对象的方法进行了一个拆分，将其中的若干功能分成多个类，通过类之间的调用来实现。这样就方便了开发人员自定义数据的连接和操作。

注意：在应用程序中，**ADO** 与 **ADO.NET** 是可以共存的，因为在 **.NET** 中同样可以使用 **COM** 互操作服务使用 **ADO**。

## 7.5 ADO.NET 常用对象

**ADO.NET** 提供了一些常用对象来方便开发人员进行数据库的操作，这些常用的对象通常会使用在应用程序开发中，对于中级的开发人员而言，熟练的掌握这些常用的 **ADO.NET** 对象，能够自行封装数据库操作类，来简化开发。**ADO.NET** 的常用对象包括：

- ❑ **Connection** 对象。
- ❑ **DataAdapter** 对象。
- ❑ **Command** 对象。
- ❑ **DataSet** 对象。
- ❑ **DataReader** 对象。

上面的对象在.NET 应用程序操作数据中是非常重要的，它们不仅提供了数据操作的便利，同时，还提供了高级的功能给开发人员。为开发人员解决特定的需求。

## 7.6 Connection 连接对象

在.NET 开发中，通常情况下开发人员被推荐使用 **Access** 或者 **SQL** 作为数据源，若需要连接 **Access** 数据库，可以使用 **System.Data.OleDb.OleDbConnection** 对象来连接；若需要连接 **SQL** 数据库，则可以使用 **System.Data.SqlClient.SqlConnection** 对象来连接。使用 **System.Data.Odbc.OdbcConnection** 可以连接 **ODBC** 数据源，而 **System.Data.OracleClient.OracleConnecton** 提供了连接 **Oracle** 的一些方法。本章主要讨论连接 **Access** 和 **SQL** 数据库。

### 7.6.1 连接 SQL 数据库

如需要连接 **SQL** 数据库，则需要使用命名空间 **System.Data.SqlClient** 和 **System.Data.OleDb**。使用 **System.Data.SqlClient** 和 **System.Data.OleDb** 能够快速连接 **SQL** 数据库，因为 **System.Data.SqlClient** 和 **System.Data.OleDb** 都分别为开发人员提供了连接方法，示例代码如下所示。

```
using System.Data.SqlClient;           //使用 SQL 命名空间
using System.Data.OleDb                //使用 OleDb 命名空间
```

#### 1. 使用 System.Data.SqlClient

连接 **SQL** 数据库，则需要创建 **SqlConnection** 对象，**SqlConnection** 对象创建代码如下所示。

```
SqlConnection con = new SqlConnection();           //创建连接对象
con.ConnectionString = "server=(local);database='mytable';uid='sa';pwd='sa'"; //设置连接字符串
```

上述代码创建了一个 **SqlConnection** 对象，并且配置了连接字符串。**SqlConnection** 对象专门定义了一个专门接受连接字符串的变量 **ConnectionString**，当配置了 **ConnectionString** 变量后，就可以使用 **Open()** 方法来打开数据库连接，示例代码如下所示。

```
SqlConnection con = new SqlConnection();           //创建连接对象
con.ConnectionString = "server=(local);database='mytable';uid='sa';pwd='sa'";
try
{
    con.Open();           //尝试打开连接
    Label1.Text = "连接成功"; //提示打开成功
    con.Close();          //关闭连接
}
catch
{
    Label1.Text = "连接失败"; //提示打开失败
}
```

上述代码尝试判断是否数据库连接被打开，使用 **Open** 方法能够建立应用程序与数据库之间的连接。与之相同的是，可以使用默认的构造函数来对数据库连接对象进行初始化，示例代码如下所示。

```
string str = "server=(local);database='mytable';uid='sa';pwd='Sa'"; //设置连接字符串
SqlConnection con = new SqlConnection(str); //默认构造函数
```

上述代码与使用 **ConnectionString** 变量的方法等价，其默认的构造函数中已经为 **ConnectionString** 变量进行了初始化。



## 2. 使用 System.Data.OleDb

ADO.NET 中，具有相同功能的函数一般具有相同的参数和字段以及方法。所以，在.NET 开发中，开发人员能够很快的适应新的操作。同样 **System.Data.OleDb** 也提供了 **Open** 方法以及 **ConnectionString** 字段，示例代码如下所示。

```
OleDbConnection con= new OleDbConnection();           //创建连接对象
con.ConnectionString="Provider=SQLOLEDB;Data
Source=(local);Initial Catalog=mytable;uid=sa;pwd=sa"; //初始化连接字符串
try
{
    con.Open();           //尝试打开连接
    Label1.Text = "连接成功"; //提示连接成功
    con.Close();          //关闭连接
}
catch
{
    Label1.Text = "连接失败"; //提示连接失败
}
```

同样，**OleDbConnection** 也提供默认的构造函数来初始化连接变量，示例代码如下所示。

```
string str =
"Provider=SQLOLEDB;Data Source=(local);Initial Catalog=mytable;uid=sa;pwd=sa";
OleDbConnection con = new OleDbConnection(str);
```

上述代码通过使用构造函数初始化连接变量进行相应的 **ConnectionString** 变量的配置。值得注意的是，从上面代码可以看出，连接字符串一般都通过使用用户名和密码的形式连接，这样保证了连接的安全性。另外，连接字符串还可以使用 **Trusted\_Connection=Yes** 来声明这是一个值得信任的连接字符串，而不需要输入用户名和密码，示例代码如下所示。

```
string str2 =
"Provider=SQLOLEDB;Data Source=(local);Initial Catalog=mytable;Trusted_Connection=Yes";
OleDbConnection con = new OleDbConnection(str2);
```

### 7.6.2 连接 Access 数据库

**Access** 是一种桌面级数据库，虽然与 **SQL** 相比，**Access** 数据库的性能和功能都并不强大，但是 **Access** 却是最常用的数据库之一。对于小型应用和小型企业来说，**Access** 数据库也是开发中小型软件的最佳选择。

#### 1. 创建 Access 数据库

**Access** 是 **Office** 组件之一，当安装了 **Office** 后，就可以新建 **Access** 数据库，在桌面或任何文件夹中单击右键就能够创建 **Access** 数据库。创建完成后，双击数据库文件就能够打开数据库并建立表和字段，如图 7-21 所示。

同样，**Access** 数据库也需要创建表和字段，基本方法与 **SQL** 数据库相同，但是在数据类型上，自动增长编号作为单独的数据类型而存在。开发人员能够在表窗口中创建表 **mytable** 和相应字段，如图 7-22 所示。





图 7-21 创建 Access 数据库

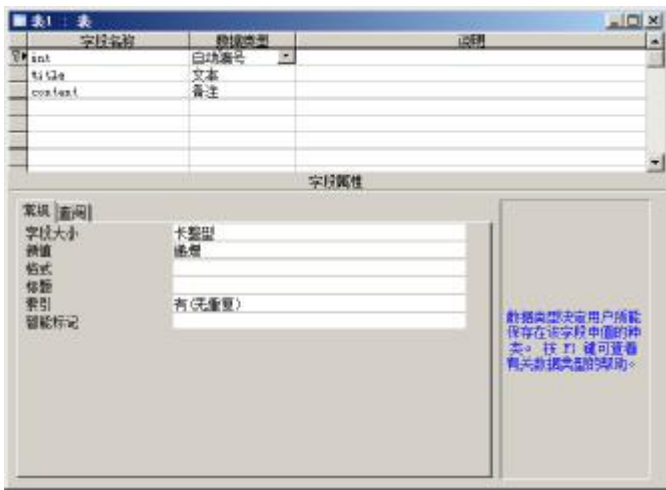


图 7-22 创建 Access 数据库的表

创建完成后可以使用 **System.Data.OleDb** 的对象进行数据库的连接和数据操作。

注意：Access 数据库是一个桌面级的数据库，其数据都会存放在一个文件中而不是存放在数据库服务器中。

2. 使用 **System.Data.OleDb**

在使用 **System.Data.OleDb** 时，只需要修改连接字符串即可。在这里需要强调一点的是，Access 数据库是一种桌面级的数据库，同文件类型的数据库类似，所以连接 Access 数据库时，必须指定数据库文件的路径，或者使用 **Server.MapPath** 来确定数据库文件的相对位置。示例代码如下所示。

```
string str = "provider=Microsoft.Jet.OLEDB.4.0 ;Data Source="
+ Server.MapPath("access.mdb") + "";
```

//使用相对路径

```
OleDbConnection con = new OleDbConnection(str);
```

//构造连接对象

```
try
{
    con.Open();
    Label1.Text = "连接成功";
    con.Close();
}
catch(Exception ee)
```

//抛出异常

```
{
    Label1.Text = "连接失败";
}
```

**Server.MapPath** 能够确定文件相对于当前目录的路径，如果不使用 **Server.MapPath**，则需要指定文件在计算机的路径，如“D:\服务器\文件夹\数据库路径”。但是这样会暴露数据库的物理路径，让程序长期处于不安全的状态。

7.6.3 打开和关闭连接

无论是使用 **System.Data.SqlClient** 还是 **System.Data.OleDb** 创建数据库连接对象，都可以使用 **Open** 方法来打开连接。同样，也可以使用 **Close** 方法来关闭连接，示例代码如下所示。

```
SqlConnection con = new SqlConnection(str);
```

//创建连接对象

```
OleDbConnection con2 = new OleDbConnection(str2);
```

//创建连接对象

```
con.Open();
```

//打开连接

```
con.Close();
```

//关闭连接

```
con2.Open();
```

//打开连接

```
con2.Close();
```

//关闭连接

注意：如果使用了连接池，虽然显式的关闭了连接对象，其实并不会真正的关闭与数据库之间的连接，这样能够保证再次进行连接时的连接性能。

## 7.7 DataAdapter 适配器对象

在创建了数据库连接后，就需要对数据集 **DataSet** 进行填充，在这里就需要使用 **DataAdapter** 对象。在没有数据源时，**DataSet** 对象对保存在 **Web** 窗体可访问的本地数据库是非常实用的，这样降低了应用程序和数据库之间的通信次数。然而 **DataSet** 必须要与一个或多个数据源进行交互，**DataAdapter** 就提供 **DataSet** 对象和数据源之间的连接。

为了实现这种交互，微软提供了 **SqlDataAdapter** 类和 **OleDbDataAdapter** 类。**SqlDataAdapter** 类和 **OleDbDataAdapter** 类各自适用情况如下。

- ❑ **SqlDataAdapter**: 该类专用于 **SQL** 数据库，在 **SQL** 数据库中使用该类能够提高性能，**SqlDataAdapter** 与 **OleDbDataAdapter** 相比，无需适用 **OLEDB** 提供程序层，可直接在 **SQL Server** 上使用。
- ❑ **OleDbDataAdapter**: 该类适用于由 **OLEDB** 数据提供程序公开的任何数据源，包括 **SQL** 数据库和 **Access** 数据库。

若要使一个使用 **DataAdapter** 对象的 **DataSet** 要能够和一个数据源之间交换数据，则可以使用 **DataAdapter** 属性来指定需要执行的操作，这个属性可以是一条 **SQL** 语句或者是存储过程，示例代码如下所示。

```
string str = "server=(local);database=mytable;uid=sa;pwd=sa";           //创建连接字符串
SqlConnection con = new SqlConnection(str);
con.Open();                                                             //打开连接
SqlDataAdapter da = new SqlDataAdapter("select * from news", con);       //DataAdapter 对象
con.Close();                                                            //关闭连接
```

上述代码创建了一个 **DataAdapter** 对象，**DataSet** 对象可以使用该对象的 **Fill** 方法来填充数据集。

## 7.8 Command 执行对象

**Command** 对象可以使用数据命令直接与数据源进行通信。例如，当需要执行一条插入语句，或者删除数据库中的某条数据的时候，就需要使用到 **Command** 对象。**Command** 对象的属性包括了数据库在执行某个语句的所有必要的信息，这些信息如下所示：

- ❑ **Name**: **Command** 的程序化名称。
- ❑ **Connection**: 对 **Connection** 对象的引用。
- ❑ **CommandType**: 指定是使用 **SQL** 语句或存储过程，默认情况下是 **SQL** 语句。
- ❑ **CommandText**: 命令对象包含的 **SQL** 语句或存储过程名。
- ❑ **Parameters**: 命令对象的参数。

通常情况下，**Command** 对象用于数据的操作，例如执行数据的插入和删除，也可以执行数据库结构的更改，包括表和数据库。示例代码如下所示。

```
string str = "server=(local);database=mytable;uid=sa;pwd=sa";           //创建数据库连接字符串
SqlConnection con = new SqlConnection(str);
con.Open();                                                             //打开数据库连接
SqlCommand cmd = new SqlCommand("insert into news values ('title')",con);//建立 Command 对象
```

上述代码使用了可用的构造函数并指定了查询字符串和 **Connection** 对象来初始化 **Command** 对象 **cmd**。通过指定 **Command** 对象的方法可以对数据执行具体的操作。

### 7.8.1 ExecuteNonQuery 方法

当指定了一个 SQL 语句，就可以通过 **ExecuteNonQuery** 方法来执行语句的操作。**ExecuteNonQuery** 不仅可以执行 SQL 语句，开发人员也可以执行存储过程或数据定义语言语句来对数据库或目录执行构架操作。而使用 **ExecuteNonQuery** 时，**ExecuteNonQuery** 并不返回行，但是可以通过 **Command** 对象和 **Parameters** 进行参数传递。示例代码如下所示。

```
string str = "server=(local);database=mytable;uid=sa;pwd=sa";           //创建数据库连接字符串
SqlConnection con = new SqlConnection(str);
con.Open();
SqlCommand cmd = new SqlCommand("insert into news values ('title'),con);
cmd.ExecuteNonQuery();                                                //执行 SQL 语句
```

运行上述代码后，会执行“**insert into news values (‘title’)**”这条 SQL 语句并向数据库中插入数据。值得注意的是，修改数据库的 SQL 语句，例如常用的 **INSERT**、**UPDATE** 以及 **DELETE** 并不返回行。同样，很多存储过程同样不返回任何行。当执行这些不返回任何行的语句或存储过程时，可以使用 **ExecuteNonQuery**。但是 **ExecuteNonQuery** 语句也会返回一个整数，表示受已执行的 SQL 语句或存储过程影响的行数，示例代码如下所示。

```
string str = "server=(local);database=mytable;uid=sa;pwd=sa";
SqlConnection con = new SqlConnection(str);                             //创建连接对象
con.Open();                                                             //打开连接
SqlCommand cmd = new SqlCommand("delete from mynews", con);           //构造 Command 对象
Response.Write("该操作影响了("+cmd.ExecuteNonQuery().ToString()+") 行");//执行 SQL 语句
```

上述代码执行了语句“**delete from mynews**”并将影响的行数输出到字符串中。开发人员能够使用 **ExecuteNonQuery** 语句进行数据库操作和数据库操作所影响行数的统计。

### 7.8.2 ExecuteNonQuery 执行存储过程

**ExecuteNonQuery** 不仅能够执行 SQL 语句，同样可以执行存储过程和数据定义语言来对数据库或目录执行构架操作如 **CREATE TABLE** 等。在执行存储过程之前，必须先创建一个存储过程，然后在 **SqlCommand** 方法中使用存储过程。在 **SQL Server** 管理器中可以新建查询创建存储过程，示例代码如下所示。

```
CREATE PROC getdetail
(
    @id int,
    @title varchar(50) OUTPUT
)
AS
SET NOCOUNT ON
DECLARE @newscount int
SELECT @title=mynews.title,@newscount=COUNT(mynews.id)
FROM mynews
WHERE (id=@id)
GROUP BY mynews.title
RETURN @newscount
```

上述存储过程返回了数据库中新闻的标题内容。“**@id**”表示新闻的 **id**，“**@title**”表示新闻的标题，此存储过程将返回“**@title**”的值，并且返回新闻的总数。上述代码可以直接在 **SQL** 管理器中菜单栏中单击【新建查询】后创建的 **TAB** 中使用，同样也可以使用 **SqlCommand** 对象进行存储过程的创建，示例代码如下所示。

```
string str = "CREATE PROC getdetail" +
"(" +
"@id int," +
"@title varchar(50) OUTPUT" +
```



```

    ")" +
    "AS" +
    "SET NOCOUNT ON" +
    "DECLARE @newscount int" +
    "SELECT @title=mynews.title,@newscount=COUNT(mynews.id)" +
    "FROM mynews" +
    "WHERE (id=@id)" +
    "GROUP BY mynews.title" +
    "RETURN @newscount";
    SqlCommand cmd = new SqlCommand(str, con);
    cmd.ExecuteNonQuery();           //使用 cmd 的 ExecuteNonQuery 方法创建存储过程

```

创建存储过程后，就可以使用 **SqlParameter** 调用命令对象 **Parameters** 参数的集合的 **Add** 方法进行参数传递，并指定相应的参数，示例代码如下所示。

```

string str = "server=(local);database=mytable;uid=sa;pwd=sa";
SqlConnection con = new SqlConnection(str);
con.Open();                                     //打开连接
SqlCommand cmd = new SqlCommand("getdetail", con); //使用存储过程
cmd.CommandType = CommandType.StoredProcedure; //设置 Command 对象的类型
SqlParameter spr;                               //表示执行一个存储过程
spr = cmd.Parameters.Add("@id", SqlDbType.Int); //增加参数 id
spr = cmd.Parameters.Add("@title", SqlDbType.NChar, 50); //增加参数 title
spr.Direction = ParameterDirection.Output; //该参数是输出参数
spr = cmd.Parameters.Add("@count", SqlDbType.Int); //增加 count 参数
spr.Direction = ParameterDirection.ReturnValue; //该参数是返回值
cmd.Parameters["@id"].Value = 1; //为参数初始化
cmd.Parameters["@title"].Value = null; //为参数初始化
cmd.ExecuteNonQuery(); //执行存储过程
Label1.Text = cmd.Parameters["@count"].Value.ToString(); //获取返回值

```

上述代码使用了现有的存储过程，并为存储过程传递了参数，当参数被存储过程接受并运行后，会返回一个存储过程中指定的返回值。当执行完毕后，开发人员可以通过 **cmd.Parameters** 来获取其中一个变量的值。

### 7.8.3 ExecuteScalar 方法

**Command** 的 **Execute** 方法提供了返回单个值的功能。在很多时候，开发人员需要获取刚刚插入的数据的 **ID** 值，或者可能需要返回 **Count(\*)**，**Sum(Money)**等聚合函数的结果，则可以使用 **ExecuteScalar** 方法。示例代码如下所示。

```

string str = "server=(local);database=mytable;uid=sa;pwd=sa"; //设置连接字符串
SqlConnection con = new SqlConnection(str); //创建连接
con.Open(); //打开连接
SqlCommand cmd = new SqlCommand("select count(*) from mynews", con); //创建 Command
Label1.Text = cmd.ExecuteScalar().ToString(); //使用 ExecuteScalar 执行

```

上述代码创建了一个连接，并创建了一个 **Command** 对象，使用了可用的构造函数来初始化对象。当使用 **ExecuteScalar** 执行方法时，会返回单个值。**ExecuteScalar** 方法同样可以执行 **SQL** 语句，但是与 **ExecuteNonQuery** 方法不同的是，当语句不为 **SELECT** 时，则返回一个没有任何数据的 **System.Data.SqlClient.SqlDataReader** 类型的集合。

**ExecuteScalar** 方法执行 **SQL** 语句通常是用来执行具有返回值的功能的 **SQL** 语句，例如上面所说的当插入一条新数据时，返回刚刚插入的数值的 **ID** 号。这种功能在自动增长类型的数据库设计中，经常被使用到，示例代码如下所示。

```

string str = "server=(local);database=mytable;uid=sa;pwd=sa"; //设置连接字符串
SqlConnection con = new SqlConnection(str); //创建连接
con.Open(); //打开连接
SqlCommand cmd = new SqlCommand("insert into mynews values ('this is a new title!')

```



SELECT @@IDENTITY as 'bh', con);	//打开连接
Label2.Text = cmd.ExecuteScalar().ToString();	//获取返回的 ID 值

上述代码使用了“**SELECT @@IDENTITY as**”语法，“**SELECT @@IDENTITY**”语法会自动获取刚刚插入的自动增长类型的值，例如，当表中有 **100** 条数据时，插入一条数据后数据量就成 **101**，为了不需要再次查询就获得 **101** 这个值，则可以使用“**SELECT @@IDENTITY as**”语法。运行结果如图 7-23 所示。

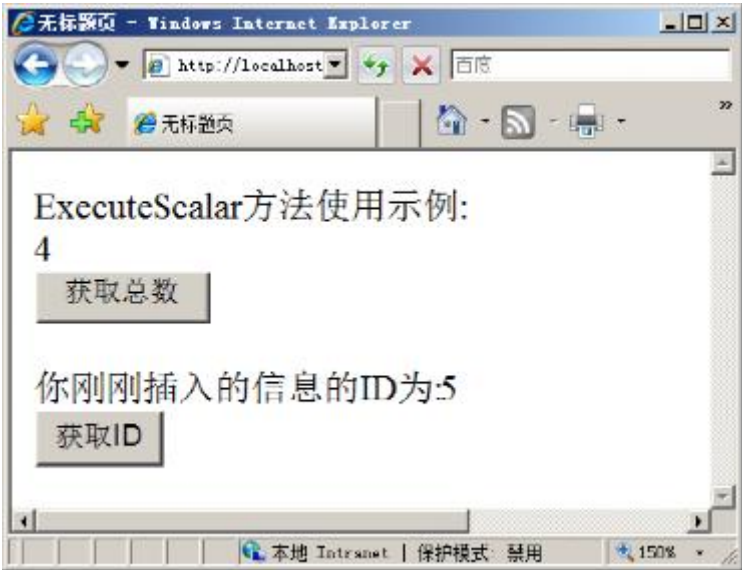


图 7-23 ExecuteScalar 方法示例

当使用了“**SELECT @@IDENTITY as**”语法进行数据操作时，**ExecuteScalar** 方法会返回刚刚插入的数据的 **ID**，这样就无需再次查询进行刚刚插入的数据的信息的获取。

## 7.9 DataSet 数据集对象

**DataSet** 是 **ADO.NET** 中的核心概念，作为初学者，可以把 **DataSet** 想象成虚拟的表，但是这个表不能用简单的表来表示，这个表可以想象成具有数据库结构的表，并且这个表是存放在内存中的。由于 **ADO.NET** 中 **DataSet** 的存在，开发人员能够屏蔽数据库与数据库之间的差异，从而获得一致的编程模型。

### 7.9.1 DataSet 数据集基本对象

**DataSet** 能够支持多表、表间关系、数据库约束等，可以模拟一个简单的数据库模型。**DataSet** 对象模型如图 7-24 所示。

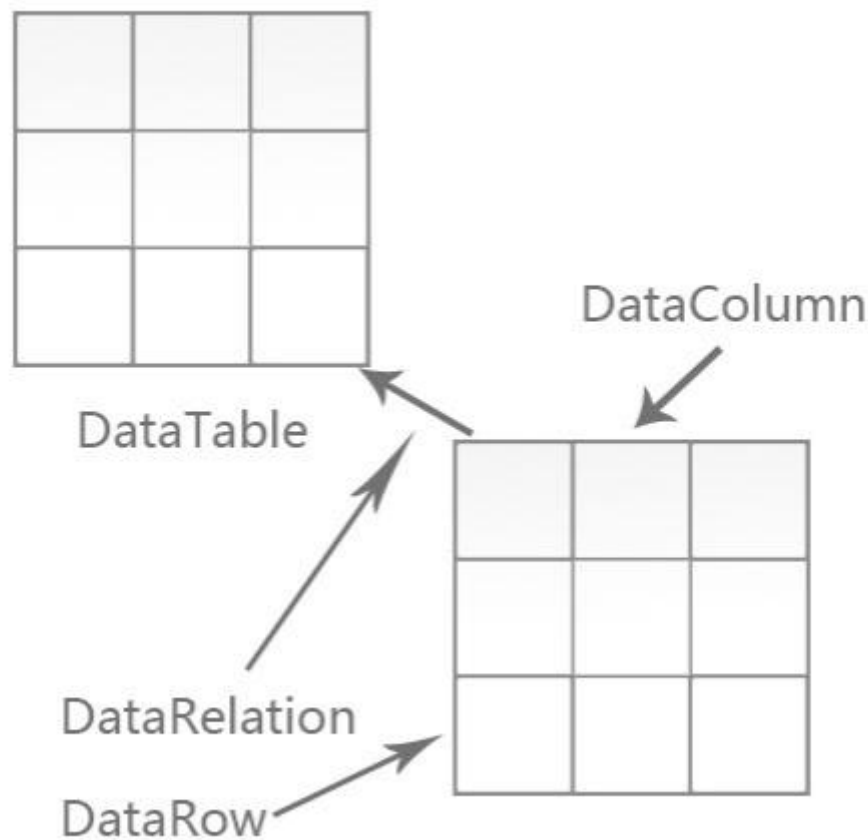


图 7-24 DataSet 对象模型

上图简要的介绍了常用对象之间的构架关系。在 **DataSet** 中，主要包括 **TablesCollection**、**RelationsCollection**、**ExtendedProperties** 几个重要对象：

### 1. TablesCollection 对象

在 **DataSet** 中，表的概念是用 **DataTable** 来表示的。**DataTable** 在 **System.Data** 中定义，它能够表示存储在数据库中的一张表。它包含一个 **ColumnsCollection** 的对象，代表数据表的各个列的定义。同时，它也包含 **RowsCollection** 对象，这个对象包含 **DataTable** 中的所有数据。

### 2. RelationsCollection 对象

在各个 **DataTable** 对象之间，是通过使用 **RelationsCollection** 来表达各个 **DataTable** 对象之间的关系。**RelationsCollection** 对象可以模拟数据库中的约束的关系。例如当一个包含外键的表被更新时，如果不满足主键-外键约束，这个更新操作就会失败，系统会抛出异常。

### 3. ExtendedProperties 对象

**ExtendedProperties** 对象能够配置特定的信息，例如 **DataTable** 的密码，更新时间等等。

## 7.9.2 DataTable 数据表对象

**DataTable** 是 **DataSet** 中的常用的对象，它和数据库中的表的概念十分相似。开发人员能够将 **DataTable** 想象成一个表。并且可以通过编程的方式创建一个 **DataTable** 表。示例代码如下所示。

```
DataTable Table = new DataTable("mytable");           //创建一个 DataTable 对象
Table.CaseSensitive = false;                         //设置不区分大小写
Table.MinimumCapacity = 100;                         //设置 DataTable 初始大小
Table.TableName = "newtable";                       //设置 DataTable 的名称
```

上述代码创建了一个 **DataTable** 对象，并为 **DataTable** 对象设置了若干属性，这些属性都是常用的属性，其作用分别如下所示。

- ❑ **CaseSensitive**: 此属性设置表中的字符串是否区分大小写，若无特殊情况，一般设置为 **false**，该属性对于查找，排序，过滤等操作有很大的影响。
- ❑ **MinimumCapacity**: 设置创建的数据表的最小的记录空间。
- ❑ **TableName**: 指定数据表的名称。

一个表必须有一个列，而 **DataTable** 必须包含列。当创建了一个 **DataTable** 后，就必须向 **DataTable** 中

增加列的。表中列的集合形成了二维表的数据结构。开发人员可以使用 **Columns** 集合的 **Add** 方法向 **DataTable** 中增加列，**Add** 方法带有两个参数，一个是表列的名称，一个是该列的数据类型。示例代码如下所示。

```

        DataTable Table = new DataTable("mytable");           //创建一个 DataTable
        Table.CaseSensitive = false;                         //设置不区分大小写
        Table.MinimumCapacity = 100;                        //设置 DataTable 初始大小
        Table.TableName = "newtable";                       //设置 DataTable 的名称
        DataColumn Colum = new DataColumn();                //创建一个 DataColumn
        Colum = Table.Columns.Add("id", typeof(int));        //增加一个列
        Colum = Table.Columns.Add("title", typeof(string));  //增加一个列
    
```

上述代码创建了一个 **DataTable** 和一个 **DataColumn** 对象，并通过 **DataTable** 的 **Columns.Add** 方法增加 **DataTable** 的列，这两列的列名和数据类型如下：

- ❑ 新闻 **ID**：整型，用于描述新闻的编号。
- ❑ 新闻标题 **TITLE**：字符型，用于描述新闻发布的标题。

注意：上述代码中，**DataTable** 的列的数据类型使用的只能是.net 中数据类型，因为其并不是真实的数据库，所以不能直接使用数据库类型，必须使用 **typeof** 方法把.net 中的数据类型转换成数据库类型。

## 7.9.3 DataRow 数据行对象

在创建了表和表中列的集合，并使用约束定义表的结构后，可以使用 **DataRow** 对象向表中添加新的数据库行，这一操作同数据库中的 **INSERT** 语句的概念类似。插入一个新行，首先要声明一个 **DataRow** 类型的变量。使用 **NewRow** 方法能够返回一个新的 **DataRow** 对象。**DataTable** 会根据 **DataColumnCollection** 定义的表的结构来创建 **DataRow** 对象。示例代码如下所示。

```

        DataRow Row = Table.NewRow();           //使用 DataTable 的 NewRow 方法创建一个新 DataRow 对象
    
```

上述代码使用 **DataTable** 的 **NewRow** 方法创建一个新 **DataRow** 对象，当使用该对象添加了新行之后，必须使用索引或者列名来操作新行，示例代码如下所示。

```

        Row[0] = 1;                               //使用索引赋值列
        Row[1] = "datarow";                       //使用索引赋值列
    
```

上述代码通过索引来为一行中个各列赋值。从数组的语法可以知道，索引都是从第 **0** 个位置开始。将 **DataTable** 想象成一个表，从左到右从 **0** 开始索引，直到数值等于列数减 **1** 为止。为了提高代码的可读性，也可以通过直接使用列名来添加新行，示例代码如下所示。

```

        Row["bh"] = 1;                             //使用列名赋值列
        Row["title"] = "datarow";                   //使用列名赋值列
    
```

通过直接使用列名来添加新行与使用索引添加新行的效果相同，但是通过使用列名能够让代码更加可读，便于理解，但是也暴露了一些机密内容（如列值）。在数据插入到新行后，使用 **Add** 方法将该行添加到 **DataRowCollection** 中，示例代码如下所示。

```

        Table.Rows.Add(Row);                       //增加列
    
```

## 7.9.4 DataView 数据视图对象

当需要显示 **DataRow** 对象中的数据时，可以使用 **DataView** 对象来显示 **DataSet** 中的数据。在显示 **DataSet** 中的数据之前，需要将 **DataTable** 中的数据填充到 **DataSet**。值得注意的是，**DataSet** 是 **DataTable** 的集合，可以使用 **DataSet** 的 **Add** 方法将多个 **DataTable** 填充到 **DataSet** 中去，示例代码如下所示。

```

        DataSet ds = new DataSet();                 //创建数据集
        ds.Tables.Add(Table);                       //增加表
    
```

填充完成后，可以通过 **DataView** 对象来显示 **DataSet** 数据集中的内容，示例代码如下所示。

```

        dv = ds.Tables["newtable"].DefaultView;     //设置默认视图
    
```

**DataSet** 对象中的每个 **DataTable** 对象都有一个 **DefaultView** 属性，该属性返回表的默认视图。上述代码访问了名为 **newtable** 表的 **DataTable** 对象。开发人员能够自定义 **DataView** 对象，该对象能够筛选表达式来设置 **DataView** 对象的 **RowFilter** 属性，筛选表达式的值必须为布尔值。同时，该对象能够设置 **Sort** 属性进行排序，排序表达式可以包括列名或一个算式，示例代码如下所示。

```

DataView dv = new DataView();           //创建数据视图对象
DataSet ds = new DataSet();             //创建数据集
ds.Tables.Add(Table);                   //增加数据表
dv = ds.Tables["newtable"].DefaultView; //设置默认视图
dv.RowFilter = "id" = "1";              //设置筛选表达式
dv.Sort = "id";                         //设置排序表达式

```

技巧：要显示 **DataSet** 中某项的值，可以使用语法 **ds.Tables[“表名称”].Rows[0][“列名称”].ToString()** 来显示，这种语法通常需要知道行的数目以免在访问数据时越界。

## 7.10 DataReader 数据访问对象

**DataSet** 的最大好处在于，能够提供无连接的数据库副本，**DataSet** 对象在表的生命周期内会为这些表进行内存的分配和维护。如果有多个用户同时对一台计算机进行操作，内存的使用就会变得非常的紧张。当对数据所需要进行一些简单的操作时，就无需保持 **DataSet** 对象的生命周期，可以使用 **DataReader** 对象。

### 7.10.1 DataReader 对象概述

当使用 **DataReader** 对象时，不会像 **DataSet** 那样提供无连接的数据库副本。**DataReader** 类被设计为产生只读，只进的数据流。这些数据流都是从数据库返回的。所以，每次的访问或操作只有一个记录保存在服务器的内存中。相比与 **DataSet** 而言，**DataReader** 具有较快的访问能力，并且能够使用较少的服务器资源，**DataReader** 具有以下快速的数据库访问、只进和只读、减少服务器资源等特色。

#### 1. 快速的数据库访问

**DataReader** 类是轻量级的，相比之下 **DataReader** 对象的速度要比 **DataSet** 要快。因为 **DataSet** 在创建和初始化时，可能是一个或多个表的集合，并且 **DataSet** 具有向前，向后读写和浏览的能力，所以当创建一个 **DataSet** 对象时，会造成额外的开销。

#### 2. 只进和只读

当对数据库的操作没有太大的要求时，可以使用 **DataReader** 显示数据。这些数据可以与单个 **list-bound** 控件绑定，也可以填充 **List** 接口。当不需要复杂的数据库处理时，**DataReader** 能够较快的完成数据显示。

#### 3. 减少服务器资源

因为 **DataReader** 并不是数据的内存的表示形式，所以使用 **DataReader** 对服务器占用的资源很少。

#### 4. 自定义数据库管理

**DataReader** 对象可以使用 **Read** 方法来进行数据库遍历，当使用 **Read** 方法时，可以以编程的方式自定义数据库中数据的显示方式，当开发自定义控件时，可以将这些数据整合到 **HTML** 中，并显示数据。

#### 5. 手动连接管理

**DataAdapter** 对象能够自动的打开和关闭连接，而 **DataReader** 对象需要用户手动的管理连接。**DataReader** 对象和 **DataAdapter** 对象很相似，都可以从 **SQL** 语句和一个连接中初始化。



### 7.10.2 DataReader 读取数据库

创建 **DataReader** 对象，需要创建一个 **SqlCommand** 对象来代替 **SqlDataAdapter** 对象。与 **SqlDataAdapter** 对象类似的是，**DataReader** 可以从 **SQL** 语句和连接中创建 **Command** 对象。创建对象后，必须显式的打开 **Connection** 对象。示例代码如下所示。

```
string str = "server=(local);database='mytable';uid='sa';pwd='sa'";
SqlConnection con = new SqlConnection(str);
con.Open();                                     //打开连接
SqlCommand cmd = new SqlCommand("select * from mynews", con); //创建 Command 对象
SqlDataReader dr = cmd.ExecuteReader();          //创建 DataReader 对象
con.Close();
```

上述代码创建了一个 **DataReader** 对象，从上述代码中可以看出，创建 **DataReader** 对象必须经过如下几个步骤：

- ☐ 创建和打开数据库连接。
- ☐ 创建一个 **Command** 对象。
- ☐ 从 **Command** 对象中创建 **DataReader** 对象。
- ☐ 调用 **ExecuteReader** 对象。

**DataReader** 对象的 **Read** 方法可以判断 **DataReader** 对象中的数据是否还有下一行，并将游标下移到下一行。通过 **Read** 方法可以判断 **DataReader** 对象中的数据是否读完。示例代码如下所示。

```
while (dr.Read())
```

通过 **Read** 方法可以遍历读取数据库中行的信息，当读取到一行时，需要获取某列的值只需要使用 “[” 和 “]” 运算符来确定某一列的值即可，示例代码如下所示。

```
while (dr.Read())
{
    Response.Write(dr["title"].ToString()+"<hr/>");
}
```

上述代码通过 **dr["title"]** 来获取数据库中 **title** 这一列的值，同样也可以通过索引来获取某一列的值，示例代码如下所示。

```
while (dr.Read())
{
    Response.Write(dr[1].ToString()+"<hr/>");
}
```

### 7.10.3 异常处理

在使用 **DataReader** 对象进行连接时，需要使用 **Try...Catch...Finally** 语句进行异常处理，以保证如果代码出现异常时连接能够关闭，否则连接将保持打开状态，影响应用程序性能。示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    string str = "server=(local);database='mytable';uid='sa';pwd='Bbg0123456#'";
    SqlConnection con = new SqlConnection(str);
    con.Open();
    SqlCommand cmd = new SqlCommand("select * from mynews", con);
    SqlDataReader dr;
    try
    {
        dr = cmd.ExecuteReader();
        while (dr.Read())
        {
            Response.Write(dr[1].ToString() + "<hr/>");
        }
    }
}
```

```

    }
    catch (Exception ee)                                //出现异常
    {
        Response.Write(ee.ToString());                  //出现异常则抛出错误语句
    }
    finally
    {
        dr.Close();                                     //强制关闭连接
        con.Close();                                    //强制关闭连接
    }
}

```

上述代码当出现异常时，则会抛出异常，并强制关闭连接。这样做就能够在程序发生异常时，依旧关闭连接应用程序与数据库的连接，否则大量的异常连接状态的出现会影响应用程序性能。

## 7.11 连接池概述

在应用程序与数据库交互中，建立和关闭数据库连接都是非常消耗资源的过程。如果一个应用程序需要大量的与数据库进行交互，则很有可能造成假死，以及崩溃的情况。使用连接池能够提高应用程序的性能。

连接池是 **SQL Server** 或 **OleDb** 数据源的功能，它可以使特定的用户重复使用连接，数据库连接池技术的思想非常简单，将数据库连接作为对象存储在一个 **Vector** 对象中，一旦数据库连接建立后，不同的数据库访问请求就可以共享这些连接，这样，通过复用这些已经建立的数据库连接，可以极大地节省系统资源和时间。连接池的主要操作如下所示：

- ☐ 建立数据库连接池对象。
- ☐ 对于一个数据库访问请求，直接从连接池中得到一个连接。如果数据库连接池对象中没有空闲的连接，且连接数没有达到最大，创建一个新的数据库连接。
- ☐ 存取数据库。
- ☐ 关闭数据库，释放所有数据库连接。
- ☐ 释放数据库连接池对象。

**注意：**在关闭数据库这一步中，并非真正的关闭，而是将其放入空闲队列中。如实际空闲连接数大于初始空闲连接数则释放连接。

当一个网站用户需要同数据库之间进行交互时，服务器会为网站用户建立一个业务对象，每个业务对象维护自身的连接，这些业务对象自身会创建连接。当用户无需该业务对象时，业务对象会释放连接，如图 7-25 所示。

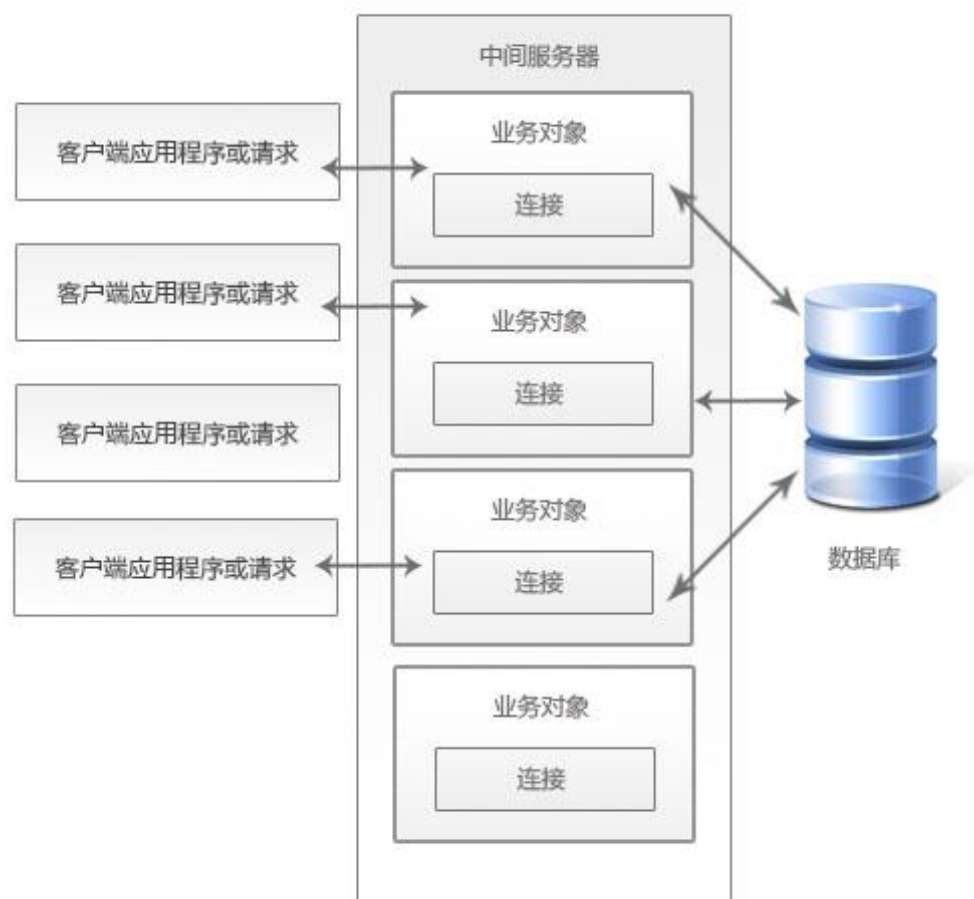


图 7-25 多层构架应用程序

当业务对数据库进行复杂的操作，并不停的打开和断开数据库连接，这样的操作会造成应用程序性能降低，因为重复的打开和断开数据库连接是非常消耗资源的，而使用连接池则可以避免这样的问题。连接池并不会真正的完全的关闭数据库与应用程序的连接，而将这些连接存放在应用程序连接池中。当一个新的业务对象产生时，会在连接池中检查是否已有连接，若无连接，则会创建一个新连接，否则会使用现有的匹配的连接，这样就提高了性能，如图 7-26 所示。

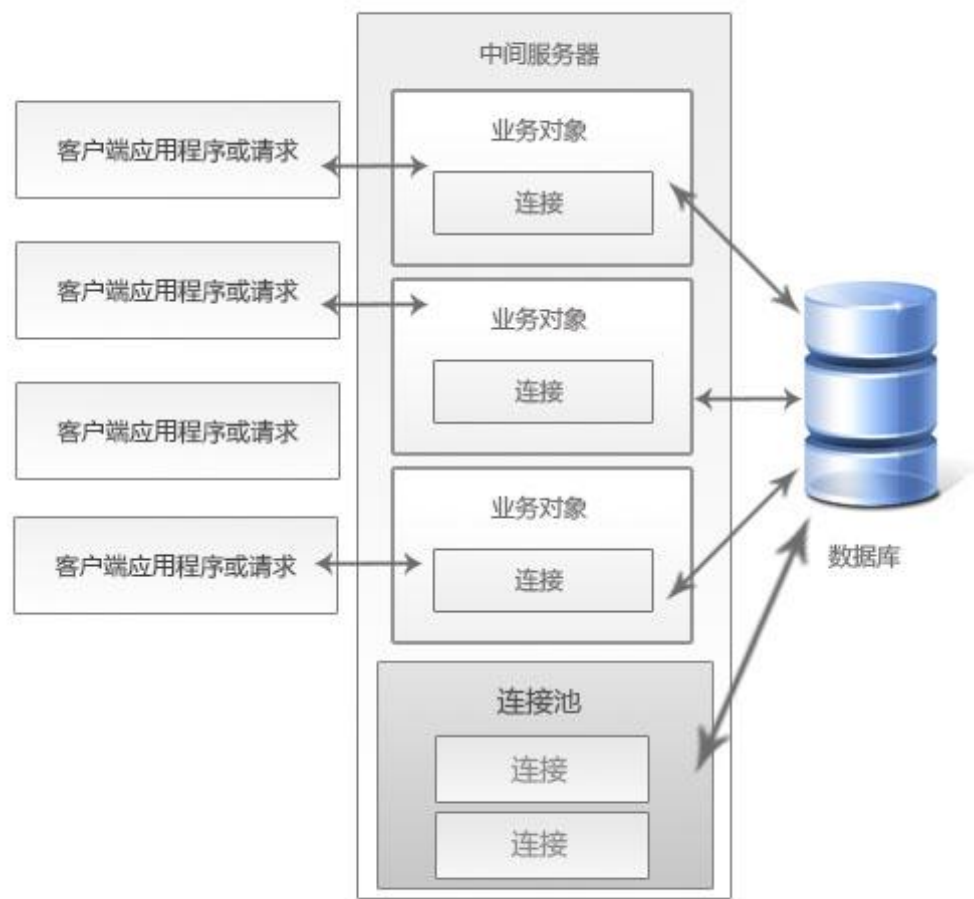


图 7-26 使用连接池

使用连接池能够提升应用程序的性能，特别是开发 **Web** 应用程序时，**Web** 应用程序通常需要频繁的与数据库之间进行交互，应用程序池能够解决 **Web** 引用中假死等情况，也能够节约服务器资源。但是，在创建连接时，良好的关闭习惯也是非常必要的。

## 7.12 参数化查询

在 **Web** 应用程序的开发过程中，**Web** 安全是非常重要的，现存的很多网站也都存在一些非常严重的安全漏洞，其中 **SQL** 注入是非常常见的漏洞，如果将查询语句进行参数化查询，可以减少 **SQL** 注入漏洞的概率，参数化查询示例代码如下所示。

```
string strsql = "select * from mynews where id= @id";
```

上述代码使用了参数化查询，在存储过程中，参数化是非常常见的，存储过程通过 **Command** 对象进行参数的添加和赋值。同样，参数化查询也可以通过 **Command** 对象进行添加和赋值，参数化查询过程如下所示。

- ❑ 创建一个 **Command** 对象。
- ❑ **Command** 对象增加一个参数。
- ❑ 通过索引对 **Command** 参数进行赋值。
- ❑ 执行 **ExecuteReader** 方法返回个 **DataReader** 对象。

通过 **Command** 对象可以为存储过程，以及参数化查询语句进行参数的添加，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    string str = "server=(local);database=mytable;uid='sa';pwd='sa'";
    SqlConnection con = new SqlConnection(str);
    con.Open();
    string strsql = "select * from mynews where id = @bh";
    SqlCommand cmd = new SqlCommand(strsql, con);           //创建 Command 对象
    cmd.Parameters.Add("@bh", SqlDbType.Int);               //增加参数@bh
    cmd.Parameters[0].Value = 4;                             //通过索引为参数赋值
    SqlDataReader dr = cmd.ExecuteReader();                 //执行后返回 DataReader 对象
    while (dr.Read())                                       //遍历 DataReader 对象
    {
        Response.Write(dr["title"].ToString()+"<hr>");
    }
}
```

参数化查询能够有效的解决一些安全问题，提高 **Web** 应用的安全性。同时，参数化查询能够极大的简化程序设计。只需要通过数值的更改而不需要修改 **SQL** 语句，极大的方便了应用程序的维护。

**注意：**如果未初始化 **Parameter** 数据类型的属性，但设置了 **Value** 属性，那么 **Parameter** 会自动选择合适的数据类型。

## 7.13 小结

本章接单的介绍了数据的基础知识，包括什么是数据库，数据库的作用。然后讲述了 **SQL Server 2005** 的数据库基本使用，并介绍了 **SQL Server Management** 管理工具的使用。通过介绍 **SQL Server Management** 管理工具，介绍了如何使用 **SQL Server Management** 管理工具和 **SQL** 语句创建表，删除表等过程。本章还包括：

- ❑ **ADO.NET** 连接 **SQL** 数据库：使用 **ADO.NET** 连接 **SQL** 数据库示例。



- ❑ **ADO.NET 与 ADO:** ADO 与 ADO.NET 发展史和利弊。
- ❑ **Connection 对象:** Connection 对象概述。
- ❑ **连接 SQL 数据库:** 使用 Connection 对象连接 SQL 数据库。
- ❑ **连接 Access 数据库:** 使用 Connection 对象连接 Access 数据库。
- ❑ **DataAdapter 对象:** 讲解了 DataAdapter 对象的使用。
- ❑ **Command 对象:** 讲解了 Command 对象的使用。
- ❑ **DataSet 对象:** 讲解了 DataSet 对象中常用的方法, 并高效使用 DataSet 开发。
- ❑ **DataReader 对象:** 讲解了 DataReader 对象。
- ❑ **连接池概述:** 讲解了连接池。
- ❑ **参数化查询:** 讲解了使用参数化查询提供安全性保证和简化开发。

在了解了基本的 ADO.NET 对象, 以及 ADO.NET 对象的作用后, 下一章将讲解如何使用数据源控件来显式和操作数据库。

## 第 8 章 Web 窗体的数据控件

在了解了 ADO.NET 基础后，就可以使用 ADO.NET 提供的对象进行数据库开发和操作。ASP.NET 还提供了一些 Web 窗体的数据控件，开发人员能够智能的配置与数据库的连接，而不需要手动的编写数据库连接。ASP.NET 不仅提供了数据源控件，还提供了能够显示数据的控件，简化了数据显示的开发，开发人员只需要简单的修改模板就能够实现数据显示和分页。

### 8.1 数据源控件

数据源控件很像 ADO.NET 中的 **Connection** 对象，数据源控件用来配置数据源，当数据控件绑定数据源控件时，就能够通过数据库源控件来获取数据源中的数据并显示。而无需通过程序实现数据源代码的编写。

#### 8.1.1 SQL 数据源控件 (SqlDataSource)

**SqlDataSource** 控件代表一个通过 ADO.NET 连接到 SQL 数据库提供者的数据源控件。并且 **SqlDataSource** 能够与任何一种 ADO.NET 支持的数据库进行交互，这些数据库包括 SQL Server、ACCESS、OleDb、Odbc 以及 Oracle。

**SqlDataSource** 控件能够支持数据的检索、插入、更新、删除、排序等，以至于数据绑定控件可以在这些能力被允许的条件下自动的完成该功能，而不需要手动的代码实现。并且 **SqlDataSource** 控件所属的页面被打开时，**SqlDataSource** 控件能够自动的打开数据库，执行 SQL 语句或存储过程，返回选定的数据，然后关闭连接。**SqlDataSource** 控件强大的功能极大的简化了开发人员的开发，缩减了开发中的代码。但是 **SqlDataSource** 控件也有一些缺点，就是在性能上不太适应大型的开发，而对于中小型的开发，**SqlDataSource** 控件已经足够了。

##### 1. 建立 SqlDataSource 控件

ASP.NET 提供的 **SqlDataSource** 控件能够方便的添加到页面，当 **SqlDataSource** 控件被添加到 ASP.NET 页面中时，会生成 ASP.NET 标签，示例代码如下所示。

```
<asp:SqlDataSource ID="SqlDataSource1" runat="server"></asp:SqlDataSource>
```

切换到视图模式下，点击 **SqlDataSource** 控件会显示【配置数据源.....】，单击【配置数据源.....】连接时，系统能够智能的提供 **SqlDataSource** 控件配置向导，如图 8-1 所示。

在新建数据源后，开发人员可以选择是否保存在 **web.config** 数据源中以便应用程序进行全局配置，通常情况下选择保存。由于现在没有连接，单击【新建连接】按钮选择或创建一个数据源。单击后，系统会弹出对话框用于选择数据库文件类型，如图 8-2 所示。

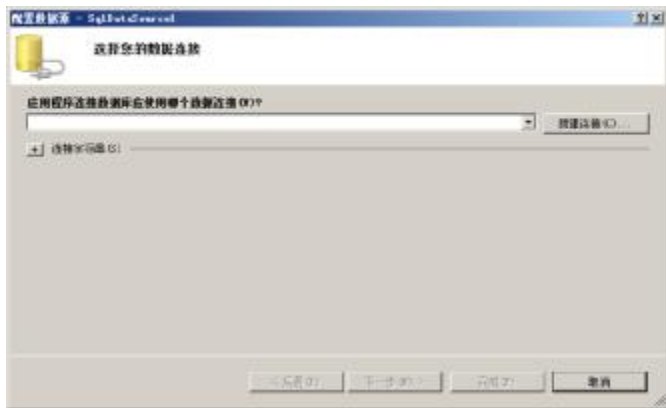


图 8-1 配置 SqlDataSource 控件

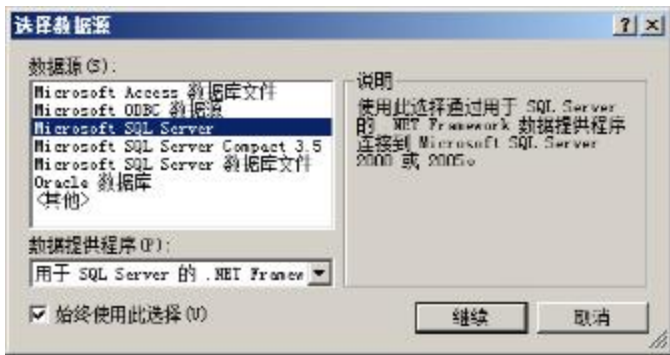


图 8-2 选择数据源

当选择完后，配置信息就会显示在 **web.config** 中。当需要对用户控件进行维护时，可以直接修改 **web.config**，而不需要修改每个页面的数据源控件，这样就方便了开发和维护。当选择了数据源后，需要对数据源的连接进行配置，这一步与 ADO.NET 中的 **Connection** 对象一样，就是要与数据库建立连接，当配置好连接后，可以单击【测试连接】按钮来测试是否连接成功，如图 8-3 和图 8-4 所示。



图 8-3 添加连接



图 8-4 测试连接

连接成功后，单击【确定】按钮，系统会自动添加连接，如图 8-5 所示。连接添加成功后，在 **web.config** 配置文件中，就有该连接的连接字串，代码如下所示。

```
<connectionStrings>
  <add name="mytableConnectionString" connectionString="Data
    Source=WIN-YXDGNPG621;Initial Catalog=mytable; Integrated Security=True"
    providerName="System.Data.SqlClient" />
</connectionStrings>
```

数据源控件可以指定开发人员所需要使用的 **Select** 语句或存储过程，开发人员能够在配置 **Select** 语句窗口中进行 **Select** 语句的配置和生成，如果开发人员希望手动编写 **Select** 语句或其他语句，可以单击【指定自定义 SQL 语句或存储过程】按钮进行自定义配置，**Select** 语句的配置和生成如图 8-6 所示。



图 8-5 成功添加连接

图 8-6 配置使用 Select 语句

对于开发人员，只需要勾选相应的字段，选择 **Where** 条件和 **Order By** 语句就可以配置一个 **Select** 语句。但是，通过选择只能够查询一个表，并实现简单的查询语。如果要想实现复杂的 **SQL** 查询语句，可以单击【指定自定义 **SQL** 语句或存储过程】进行自定义 **SQL** 语句或存储过程的配置，如图 8-7 所示，开发人员选择了一个 **getdetail** 的存储过程作为数据源。

单击【下一步】按钮，就需要对相应的字段进行配置，这些字段就像 **ADO.NET** 中的参数化查询一样。在数据源控件中，也是通过@来表示参数化变量，当需要配置相应的字段，例如配置 **WHERE** 语句等就需要对参数进行配置，如图 8-8 所示。



图 8-7 定义自定义语句或存储过程

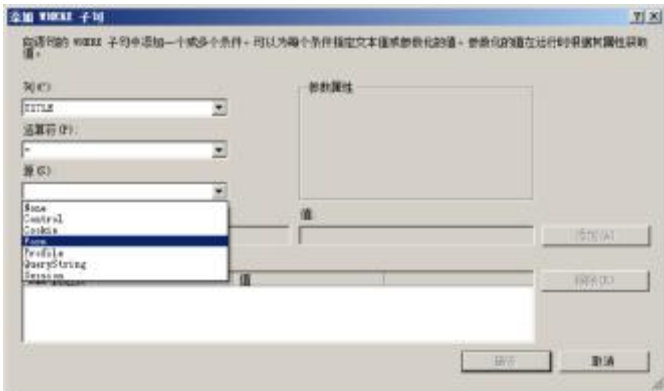


图 8-8 添加 WHERE 子句

添加 **WHERE** 子句时，**SQL** 语句中的值可以选择默认值、控件、**Cookie** 或者是 **Session** 等。当配置完成后，就可以测试查询，如果测试后显示的结果如预期一样，则可以单击完成，如图 8-9 所示。



图 8-9 测试查询并完成

完成后，**SqlDataSource** 控件标签代码如下所示。

```
<asp:SqlDataSource ID="SqlDataSource1" runat="server"
    ConnectionString="<%%$ ConnectionStrings:mytableConnectionString %>"
    SelectCommand="SELECT [TITLE], [ID] FROM [mynews]">
</asp:SqlDataSource>
```

2. 配置 **SqlDataSource** 控件属性

**SqlDataSource** 控件还包括一些可视化属性，这些属性包括删除查询（**DeleteQuery**）、插入查询（**InsertQuery**）、检索查询（**SelectQuery**）以及更新查询（**UpdateQuery**）。当需要使用可视化属性时，需选择【使用自定 **SQL** 语句或存储过程】复选框，在导航中可以使用查询生成器生成查询语句，如图 8-10 所示。





图 8-10 自定义语句或存储过程

选择【查询生成器】按钮，系统会提示选择相应的表并通过相应的表来生成查询语句，如图 8-11 和图 8-12 所示。

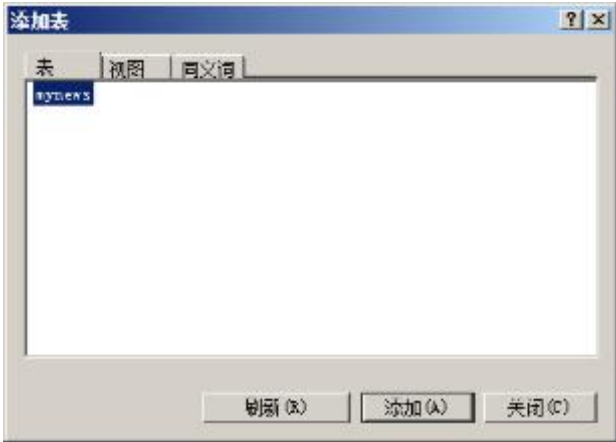


图 8-11 选择相应的表

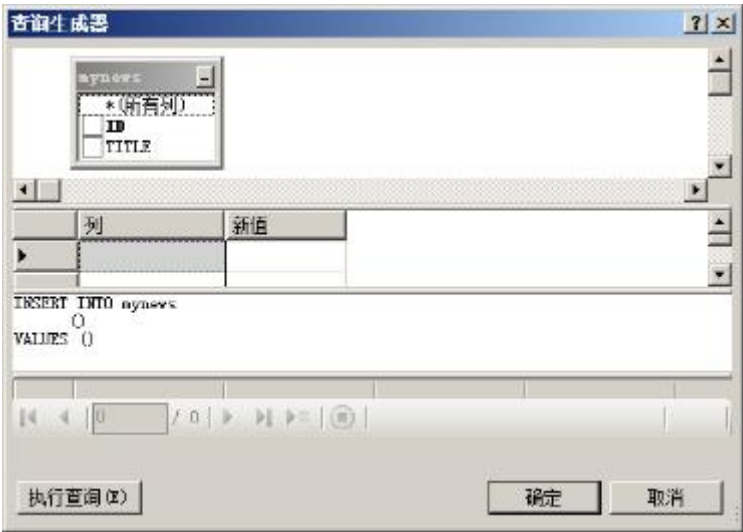


图 8-12 使用查询生成器

配置相应的查询语句后，**SqlDataSource** 控件的 **HTML** 代码如下所示。

```
<asp:SqlDataSource ID="SqlDataSource1" runat="server"
    ConnectionString="<%= $ ConnectionStrings:mytableConnectionString %>"
    InsertCommand="INSERT INTO mynews(ID) VALUES ('control title')"
    SelectCommand="SELECT [TITLE], [ID] FROM [mynews]">
</asp:SqlDataSource>
```

上述代码自动增加了一个 **InsertCommand** 并指定了 **Insert** 语句。开发人员可以为 **SqlDataSource** 控件指定四个命令参数：**SelectCommand**、**UpdateCommand**、**DeleteCommand** 和 **InsertCommand**。每个都是数据源控件的单一属性，开发人员可以配置相应的语句指定 **Select**、**Update**、**Delete** 以及 **Insert** 方法。

**SqlDataSource** 控件同时能够使用缓存来降低页面与数据库之间连接频率，这样可以避免开销很大的查询操作，以及建立连接和关闭连接操作。只要数据库是相对稳定不变的，则可以使用 **SqlDataSource** 控件的缓存属性（**EnableCaching**）来进行缓存。在默认情况下，缓存属性（**EnableCaching**）是关闭的，需要开发人员自行设置缓存属性。

8.1.2 Access 数据源控件（AccessDataSource）

在上一章中介绍了如何使用 **ADO.NET** 中 **OleDb** 来连接和读取 **Access** 数据库。**Access** 数据库是一种桌面级的数据库，当对应用程序性能，以及数据库性能要求不是很高，并且数据量不需很大时，可以考虑选择 **Access** 数据库。

**SqlDataSource** 能够与任何一种 **ADO.NET** 支持的数据源进行交互，这些数据源包括 **SQL Server**、**Access**、**OleDb**、**Odbc** 以及 **Oracle**。但是 **Access** 数据库有专门的数据源控件，就是 **AccessDataSource**。**AccessDataSource** 控件同配置 **SqlDataSource** 控件基本相同，如图 8-13 所示。

与 **SqlDataSource** 不同的是，**AccessDataSource** 主要采用的是 **ConnectionString** 属性连接数据库，而 **Access**

则采用的是 **AccessDataSource** 方式连接数据库。因为 **Access** 数据库是以文件的形式存在于系统中的，所以主要采用 **DataFile** 属性直接以文件地址的方式进行连接。要连接 **Access** 数据库，则必须选择 **Access** 数据库文件，如图 8-14 所示。



图 8-13 选择数据库



图 8-14 选择 Access 文件

在选择了 **Access** 数据库文件后，单击【确定】按钮，系统就会为开发人员配置连接字符串，在核对无误后，单击【下一步】按钮进入 **Select** 语句的配置。同 **SqlDataSource** 控件一样，同样能够配置 **Select** 语句或自定义存储过程，如图 8-15 所示。



同样 8-15 配置 Access 数据库的 Select 语句

其他步骤与 **SqlDataSource** 相同，当创建完成后，**AccessDataSource** 控件的 **HTML** 代码如下所示。

```
<asp:AccessDataSource ID="AccessDataSource1" runat="server"
    DataFile="~/acc.mdb"
    SelectCommand="SELECT [bh], [title] FROM [mytable]">
</asp:AccessDataSource>
```

当需要使用 **Access** 数据库，推荐将 **Access** 数据库文件保存在 **App\_Data** 文件夹中。以保证数据库文件是私有的，因为 **ASP.NET** 不允许直接请求 **App\_Data** 文件夹。

**注意：****AccessDataSource** 控件不支持访问受密码保护的 **Access** 数据库文件，如果需要访问受密码保护的 **Access** 数据库文件，则需要使用 **SqlDataSource** 控件。

8.1.3 目标数据源控件（ObjectDataSource）

大多数 **ASP.NET** 数据源控件，如 **SqlDataSource** 都是在两层应用程序层次结构中使用。在该层次结构中，表示层（**ASP.NET** 网页）可以与数据层（数据库和 **XML** 文件等）直接进行通信。但是，常用的应用

程序设计原则是将表示层与业务逻辑相分离，而将业务逻辑封装在业务对象中。这些业务对象在表示层和数据层之间形成一层，从而生成一种三层应用程序结构。**ObjectDataSource** 控件通过提供一种将相关页上的数据控件绑定到中间层业务对象的方法，为三层结构提供支持。在不使用扩展代码的情况下，**ObjectDataSource** 使用中间层业务对象以声明方式对数据执行选择、插入、更新、删除、分页、排序、缓存和筛选操作。

也就是说，**SqlDataSource** 是两层模型中使用的，页面通过直接访问数据库。而 **ObjectDataSource** 用于三层模型中，也就是将中间业务对象通过其访问数据库的。然后中间层业务对象再用在表示层中，例如在开发中使用的自定义控件。**ObjectDataSource** 的业务对象是可以用检索或更新数据的业务对象，例如 **Bin** 或 **App\_Code** 目录中定义的对象，选择业务对象如图 8-16 所示。



图 8-16 选择业务对象

可以创建一个类库，并在 **ASP.NET** 网站中添加引用，这样就可以通过 **ObjectDataSource** 对象选择该类库中的方法，如图 8-17 和图 8-18 所示。

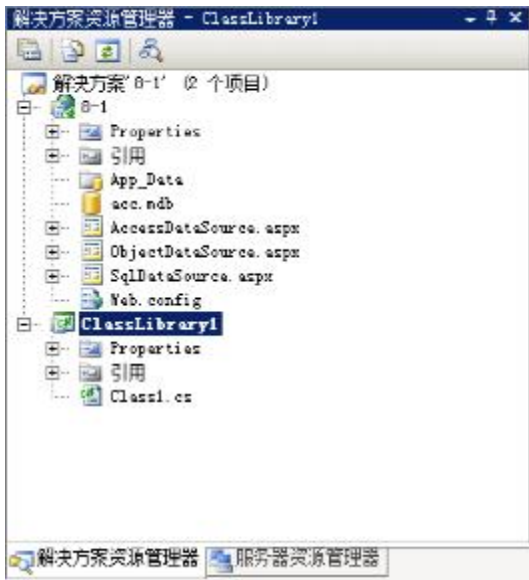


图 8-17 添加类库

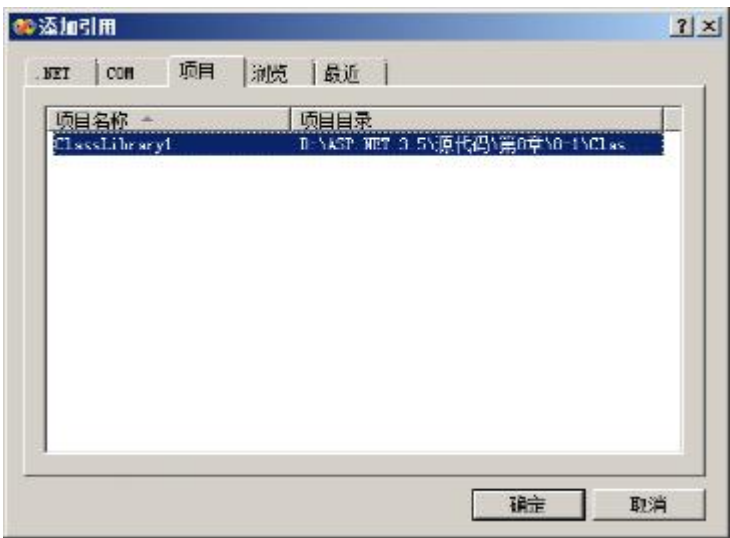


图 8-18 添加引用

**ObjectDataSource** 控件对象模型类似于 **SqlDataSource** 控件。**ObjectDataSource** 公开一个 **TypeName** 属性，该属性指定要实例化来执行数据操作的对象类型，也就是类的名称。与 **SqlDataSource** 的命令属性类似，同样 **ObjectDataSource** 包括四个重要属性，这四个属性分别为 **SelectMethod**、**UpdateMethod**、**InsertMethod** 和 **DeleteMethod**，分别用于指定要执行这些数据操作关联类型的方法。选择对象后，就可以配置 **SelectMethod**、**UpdateMethod**、**InsertMethod** 和 **DeleteMethod** 属性的方法。示例代码如下所示。

```
public class Class1 //创建类库
{
    public string GetTitle() //创建方法
}
```



```
{
    name = "title";
    return name;
}
public void InsertTitle()
{
    name = "insert";
}
public string name;
```

//变量赋值  
//返回 name  
//创建方法  
//变量赋值  
//创建共有变量 name

ObjectDataSource 控件可以使用 Class1 中的对象，如图 8-19 所示。



图 8-19 定义数据方法

ObjectDataSource 控件可以使开发人员将诸如 GridView 和 DropDownList 这样的用户界面控件绑定到一个中间层组件。能够无需编写任何代码即可绑定到一个组件，从而极大的简化用户界面。与其他的数据源控件相同，ObjectDataSource 控件在运行时可以接受参数，并在参数集合中对参数进行管理。每一项数据操作都有一个相关的参数集合。对于选择操作，可以使用 SelectParameters 集合，对于更新操作，可以使用 UpdateParameters 集合，而给予 InsertParameters、UpdateParameters、DeleteParameters 集合，需要分别确定相应操作所需调用的方法。

8.1.4 LINQ 数据源控件（LinqDataSource）

语言集成查询（LINQ）是一种查询语法，它可定义一组查询运算符，以便在任何基于.NET 的编程语言中以一种声明性的方式来表示遍历、筛选和投影操作。数据对象可以是内存中的数据集合，或者是表示数据库中数据的对象。无需为每个操作编写 SQL 命令，即可检索或修改数据。

使用 LinqDataSource 控件，开发人员可以通过在标记文本中设置属性从而在 ASP.NET 网页中使用 LINQ。LinqDataSource 控件使用 LINQ to SQL 来自动生成数据命令。LINQ 数据源可以是 LINQ 数据库或数组等以集合形式表现的数据库，有关 LINQ 的知识会有专门的章节讲解，在这里使用数组作为数据源，示例代码如下所示。

```
public string[] arr={"1","2","3","4"};
//创建数组
```

在 ASP.NET 页面中使用 LINQ 数据源控件可以对 LINQ 数据源进行查询，LINQ 数据源控件代码如下所示。

```
<asp:LinqDataSource ID="LinqDataSource1" runat="server">
</asp:LinqDataSource>
```

创建了 LINQ 数据源控件，同样单击【配置数据源.....】按钮可以进行 LINQ 数据源控件的数据源配置，如图 8-20 所示。



当选择上下文对象后，需要配置数据选择，LINQ 数据源控件同样支持 **Group** 和 **Where** 关键字，如图 8-21 所示。

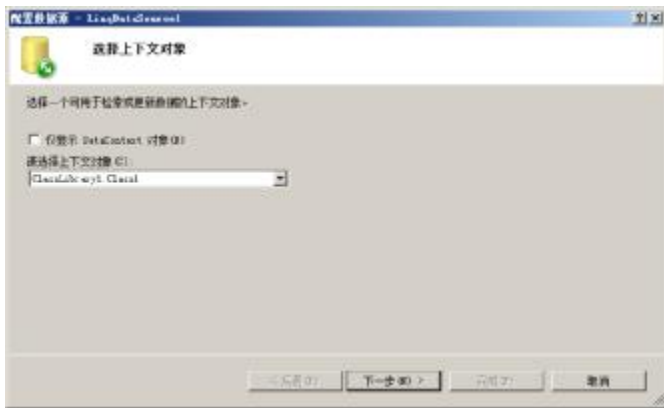


图 8-20 选择上下文对象

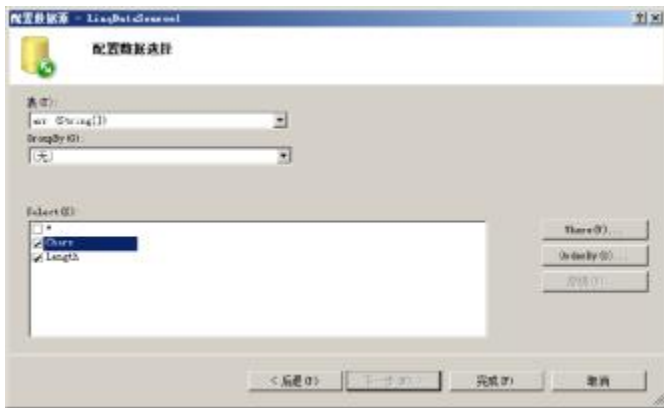


图 8-21 配置数据选择

配置完成后，LINQ 数据源控件 HTML 代码如下所示。

```
<asp:LinqDataSource ID="LinqDataSource1" runat="server"
    ContextTypeName="ClassLibrary1.Class1" Select="new (Length, Chars)"
    TableName="arr">
</asp:LinqDataSource>
```

当完成 LINQ 数据源控件（**LinqDataSource**）的配置后，就可以通过控件绑定 LINQ 数据源控件来获取 LINQ 数据库中的信息。**LinqDataSource** 控件按以下顺序应用数据操作：

- ☐ **Where:** 指定要返回的数据记录。
- ☐ **Order By:** 排序。
- ☐ **Group By:** 聚合共享值的数据记录。
- ☐ **Order Groups By:** 对分组数据进行排序。
- ☐ **Select:** 指定要返回的字段或属性。
- ☐ **Auto-sort:** 按用户选定的属性对数据记录进行排序。
- ☐ **Auto-page:** 检索用户选定的数据记录的子集。

LINQ 是 ASP.NET 3.5 中增加的一种语言集成查询，该控件的高级属性和方法在 ASP.NET 3.5 与 LINQ 中会详细讲解。

8.1.5 Xml 数据源控件（XmlDataSource）

Xml 数据源控件可以让数据绑定控件轻易的连接到 XML 数据源。在只读方案下通常使用 **XmlDataSource** 控件显示分层 XML 数据，但同样可以使用该控件显示分层数据和表格数据。

1. 建立 XmlDataSource 控件

与 **AccessDataScource** 相同的是，**XmlDataSource** 控件同样使用 **DataFile** 属性指定 XML 文件并加载 XML 数据，如图 8-22 所示。数据源是 XML 文件，单击【浏览】按钮选择数据文件，如图 8-23 所示。



图 8-22 配置数据源



图 8-23 选择 XML 数据源

选择数据源后，单击确定并完成数据源的配置即可，配置完成数据源后，**XmlDataSource** 控件的 HTML

代码如下所示。

```
<asp:XmlDataSource
    ID="XmlDataSource1" runat="server" DataFile="~/xmldata.xml">
</asp:XmlDataSource>
```

上述代码指定了 **DataFile** 属性的所属的文件，当配置完成后，**XmlDataSource** 控件就可以和数据绑定控件结合使用了。

2. XmlDataSource 控件的使用

当配置完成 **XmlDataSource** 后，就可以和数据绑定控件结合使用。在使用数据绑定控件前，先配置 **XML** 数据文件，示例代码如下所示。

```
<?xml version="1.0" encoding="utf-8" ?>
<news>
    <title>新闻标题 1</title>
    <time>2008</time>
    <author>guojing</author>
    <content>这是新闻正文</content>
    <title>新闻标题 2</title>
    <time>2008</time>
    <author>guojing</author>
    <content>这是新闻正文</content>
</news>
```

上述代码配置了 **XML** 数据文件，配置完成后，可以通过数据绑定控件来访问，可以使用 **TreeView** 控件，示例代码如下所示。

```
<asp:TreeView ID="TreeView1" runat="server" DataSourceID="XmlDataSource1">
</asp:TreeView>
```

上述代码只能够显示 **XML** 数据文件中各个节点的名称，并不能显示各个节点的值，必须为显示的节点做配置。在控件侧边单击【**TreeNode 数据绑定**】选项，并选择相应的列进行节点配置，如图 8-24 所示。



图 8-24 选择列配置 TextFiled

配置 **TextFiled** 后，各个节点的值会显示为 **XML** 数据中标签内的值，而 **XmlDataSource** 控件的 **HTML** 代码则会被系统自动替换，示例代码如下所示。

```
<asp:TreeView ID="TreeView1" runat="server" DataSourceID="XmlDataSource1"
    ImageSet="Contacts" NodeIndent="10">
    <ParentNodeStyle Font-Bold="True" ForeColor="#5555DD" />
    <HoverNodeStyle Font-Underline="False" />
    <SelectedNodeStyle Font-Underline="True" HorizontalPadding="0px"
        VerticalPadding="0px" />
    <DataBindings>
        <asp:TreeNodeBinding DataMember="title" Text="title" TextField="#InnerText" Value="title" />
    </DataBindings>
    <NodeStyle Font-Names="Verdana" Font-Size="8pt" ForeColor="Black">
```

```
HorizontalPadding="5px" NodeSpacing="0px" VerticalPadding="0px" />
</asp:TreeView>
```

运行后，相应的节点则会显示为标签的相应的值，如图 8-25 所示。



图 8-25 XmlDataSource 数据绑定

**XmlDataSource** 控件一般用于只读的数据方案。数据绑定控件显示 **XML** 数据，还可以通过 **XmlDataSource** 来编辑 **XML** 数据。但是当 **XmlDataSource** 控件加载时，必须使用 **DataFile** 属性加载，而不能从 **Data** 属性中指定的 **XML** 的字符串进行加载。

## 8.1.6 站点导航控件（SiteMapDataSource）

为了引导用户在站点的各个页面能够流畅跳转，需要在每个页面加入页面导航。在 **ASP** 的开发过程中，必须手动的为每个页面加入导航，这样不仅加大了开发的复杂度，也让代码的复用性变低。相对于手动加入导航更好的解决方法则是使用 **js** 在各个页面引用导航，但是一旦页面变得很多，可能会导致让 **js** 页面效率变低。而在 **ASP.NET 2.0** 以后的版本，微软提供了导航控件让导航菜单的创建、自定义和维护变得更加的简单。

**SiteMapDataSource** 控件包含来自站点地图的导航数据，这些数据包括有关网站中的页的信息，例如网站页面的标题、说明信息以及 **URL** 等。如果将导航数据存储在一个地方，则可以方便的在网站的导航菜单添加和删除项。站点地图提供程序中检索导航数据，然后将数据传递给可显示该数据的数据绑定控件，显示导航菜单。

如果需要使用 **SiteMapDataSource** 控件，用户必须在 **Web.sitemap** 文件中描述站点的结构，示例代码如下所示。

```
<?xml version="1.0" encoding="utf-8" ?>
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0" >
  <siteMapNode url="" title="根目录" description="根目录">
    <siteMapNode url="SqlDataSource.aspx" title="SqlDataSource.aspx" description="SQL 数据库" />
    <siteMapNode url="AccessDataSource" title="AccessDataSource" description="Access 数据库" />
    <siteMapNode url="LinqDataSource" title="LinqDataSource" description="Linq" />
    <siteMapNode url="ObjectDataSource" title="ObjectDataSource" description="Object" />
    <siteMapNode url="XmlDataSource" title="XmlDataSource" description="Xml" />
  </siteMapNode>
</siteMap>
```

上述代码描述了网站的目录结构，在文件中，必须有一个根为 **siteMapNode** 的元素作为 **siteMap** 元素的自己，并定义以下常用属性：

❑ **title**：为站点地图节点指定一个标题，该标题将显示为网页的连接文本。



- ❑ **Url:** 为网页指定 **URL**。支持相对或绝对路径。
- ❑ **Description:** 为站点地图的节点添加描述，当用户鼠标移动到该栏目时，则会显示描述信息。
- ❑ **StartFormCurrentNode:** 当设置为 **true** 时，则可以从该节点开始检索站点地图结构。
- ❑ **StartingNodeOffset:** 当属性设置为 **2** 时可以检索当前地图结构。

**SiteMapDataSource** 控件无需配置，拖放一个 **TreeView** 控件和一个 **SiteMapDataSource** 控件在页面，指定 **TreeView** 数据源即可，如图 8-26 所示。



图 8-26 配置数据源

配置完成后，数据绑定控件会自动读取 **Web.sitemap** 文件并生成导航。当使用了 **SiteMapDataSource** 控件后，数据绑定控件就能够绑定 **SiteMapDataSource** 控件并自动读取相应的值并生成导航，当需要对导航进行修改时，只需要修改 **Web.sitemap** 即可，方便了站点导航功能的使用和维护。运行后如图 8-27 所示。

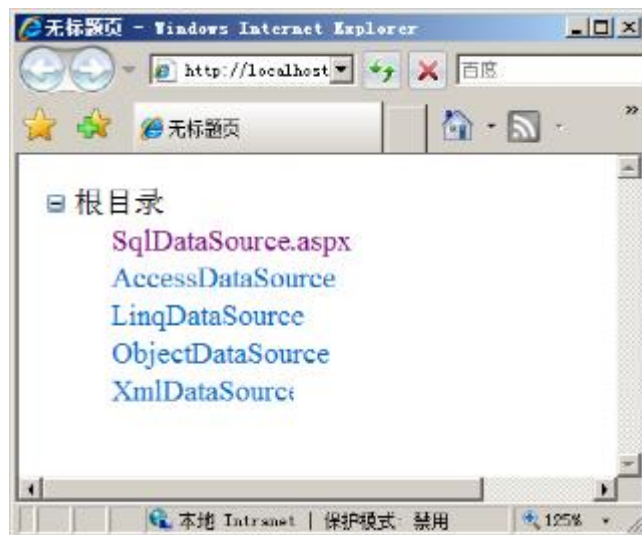


图 8-27 SiteMapDataSource 控件数据显示

## 8.2 重复列表控件（Repeater）

重复列表控件（**Repeater**）是一个可重复操作的控件。它能够通过使用模板显示一个数据源的内容，而且开发人员可以轻松的配置这些模板，**Repeater** 控件包括如标题和页脚这样的数据，它可以遍历所有的数据选项并将其应用到模板中。

重复列表控件并不是从 **WebControl** 派生出来，重复列表控件可以直接操控 **HTML** 文件或者样式表来编写模板和控制属性。重复列表控件支持 **5** 种模板，用来显示相应的界面信息，这 **5** 种模板的功能如下所示：

- ❑ **AlternatingItemTemplate:** 指定如何显示其他选项。
- ❑ **ItemTemplate:** 指定如何显示选项。
- ❑ **HeaderTemplate:** 建立如何显示标题。
- ❑ **FooterTemplate:** 建立如何显示页脚。
- ❑ **SeparatorTemplate:** 指定如何显示不同选项之间的分隔符。

在上面 **5** 种模板中，惟一需要使用的是 **ItemTemplate** 模板，其他的模板可以选用。示例代码如下所示。

```
<asp:Repeater ID="Repeater1" runat="server" DataSourceID="SqlDataSource1">
  <ItemTemplate>
    <%# Eval("title")%>
```



```
</ItemTemplate>
</asp:Repeater>
```

“<%#%>”符号之间的语句表示数据绑定表达式，可以直接使用数据源控件中查询出来字段。在 **Repeater** 中间，使用 **ItemTemplate** 制作模板，在 **ItemTemplate** 模板中可以直接使用 **HTML** 制作样式。在数据显示中，可以直接使用“<%#%>”绑定数据库中的列，例如当数据源控件中查询了一个 **title** 列时，则在 **Repeater** 控件中直接使用“<%#Eval(“title”)%>”方式显示 **title** 字段的值。

显示字段有几种方法，其中“<%#Eval(“字段名称”)%>”是最方便的显示字段的方法，能够方便的在模板中嵌入，其他方法还有使用“<%#DataBinder.Eval(Container.DataItem,“字段名称”)%>”方式来绑定相关的列。示例代码如下所示。

```
<asp:Repeater ID="Repeater1" runat="server" DataSourceID="SqlDataSource1">
  <ItemTemplate>
    <div style="border-bottom:1px dashed #ccc; padding:5px 5px 5px 5px;">
      <%# Eval("title")%>
    </div>
  </ItemTemplate>
</asp:Repeater>
```

上述代码自定义了一个 **HTML** 代码，增加了一个 **DIV** 标签，该标签设置了 **CSS** 属性 **border-bottom:1px dashed #ccc; padding:5px 5px 5px 5px;**。**Repeater** 控件能够自动的重复该模板。当数据库中的数据完毕后，则不再重复，运行结果如图 8-28 所示。

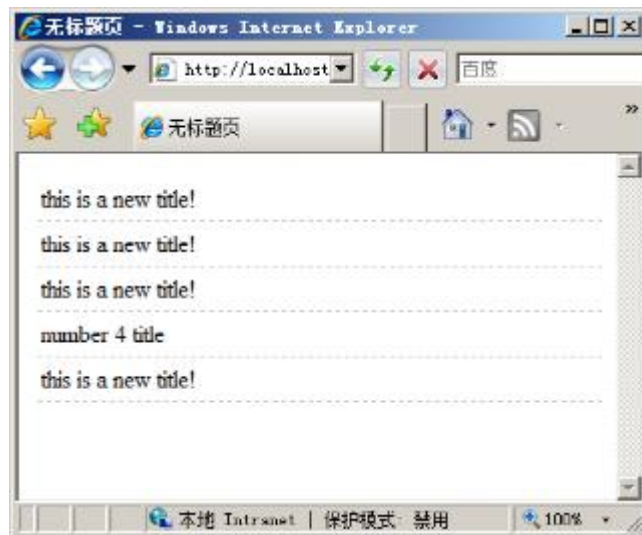


图 8-28 Repeater 控件

重复列表控件最常用的事件有 **ItemCommand**、**ItemCreated**、**ItemDataBound**。当创建一个项或者一个项被绑定到数据源时，将触发 **ItemCreated** 和 **ItemDataBound** 事件。当重复列表控件中有按钮被激发时，会触发 **ItemCommand** 事件。

在 **ItemCommand** 中，为了自定义按钮控件相应事件，开发人员必须指定 **RepeaterCommandEventArgs** 参数获取 **CommandArgument**、**CommandName** 和 **CommandSource** 三个属性对应的值，示例代码如下所示。

```
<asp:Repeater ID="Repeater1" runat="server" DataSourceID="SqlDataSource1"
onitemcommand="Repeater1_ItemCommand">
  <ItemTemplate>
    <div style="border-bottom:1px dashed #ccc; padding:5px 5px 5px 5px;">
      <%# Eval("title")%>
      <asp:Button ID="Button1" runat="server" Text="按钮"
        CommandArgument='<%# Eval("title")%>' />
    </div>
  </ItemTemplate>
</asp:Repeater>
```

上述代码增加了一个按钮控件，并配置按钮控件的命令参数为数据库中的 **title** 的值。当单击按钮控件时，则会触发 **ItemCommand**，示例代码如下所示。

```
protected void Repeater1_ItemCommand(object source, RepeaterCommandEventArgs e)
{
```

```
Label1.Text = "用户选择了" + e.CommandArgument.ToString(); //显式选择项
}
```

上述代码当指定了执行按钮控件触发的事件，运行结果如图 8-29 和图 8-30 所示。

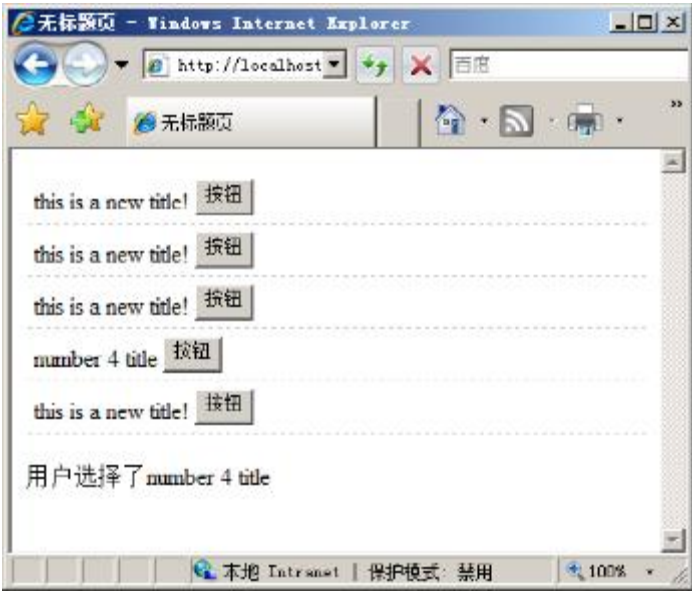


图 8-29 ItemCommand 事件



图 8-30 用户选择单击后

**Repeat** 控件需要一定的 **HTML** 知识才能显示数据库的相应信息，虽然增加了一定的复杂度，但是却增加了灵活性。**Repeat** 控件能够按照用户的想法显示不同的样式，让数据显示更加丰富。

### 8.3 数据列表控件（DataList）

**DataList** 控件支持各种不同的模板的样式，通过为 **DataList** 指定不同的样式，可以自定义 **DataList** 控件的外观。与 **Repeater** 控件相同的是，**DataList** 控件同样也支持自定义 **HTML**，但是 **DataList** 控件具备 **Repeater** 控件不具有的特性，**DataList** 控件常用属性如下所示。

- ❑ **AlternatingItemStyle**: 编写交替行的样式。
- ❑ **EditItemStyle**: 正在编辑的项的样式。
- ❑ **FooterStyle**: 列表结尾处的脚注的样式。
- ❑ **HeaderStyle**: 列表头部的标头的样式。
- ❑ **ItemStyle**: 单个项的样式。
- ❑ **SelectedItemStyle**: 选定项的样式。
- ❑ **SeparatorStyle**: 各项之间分隔符的样式。

通过修改 **DataList** 控件的相应的属性，能够实现复杂的 **HTML** 样式而不需要通过变成实现。而 **DataList** 控件能够套用自定义格式实现更多的效果，如图 8-31 所示。

通过属性生成器，同样可以通过勾选相应的项目来生成属性，这些属性能够极大的方便开发人员制作 **DataList** 控件的界面样式，如图 8-32 所示。

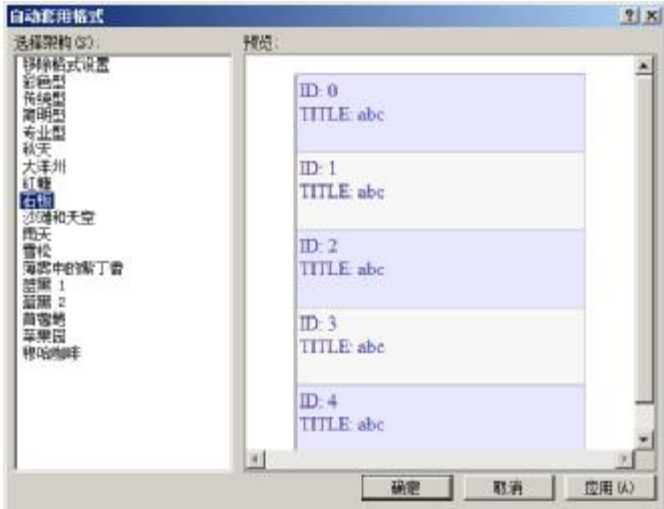


图 8-31 自动套用格式

图 8-32 属性生成器

**DataList** 控件经常在开发中使用，**DataList** 控件不仅能够支持 **Repeater** 控件中的 **ItemCommand**、**ItemCreated**、**ItemDataBound** 事件，还支持更多的服务器事件。对项中的按钮进行操作，如果按钮的 **CommandName** 属性为“**edit**”，则该按钮则可以引发 **EditorCommand** 事件，同样也可以配置不同的 **CommandName** 属性来实现不同的操作。编辑 **DataList** 控件，并编辑相应的 **HTML** 代码，让 **DataList** 控件包括按钮，并为按钮配置相应的 **CommandName** 属性，示例代码如下所示。

```
<asp:DataList ID="DataList1" runat="server" BackColor="White"
BorderColor="#E7E7FF" BorderStyle="None" BorderWidth="1px" CellPadding="3"
DataKeyField="ID" DataSourceID="SqlDataSource1" Font-Bold="False"
Font-Italic="False" Font-Overline="False" Font-Strikeout="False"
Font-Underline="False" GridLines="Horizontal" Width="100%"
ondeletecommand="DataList1_DeleteCommand">
    <FooterStyle BackColor="#B5C7DE" ForeColor="#4A3C8C" />
    <AlternatingItemStyle BackColor="#F7F7F7" />
    <ItemStyle BackColor="#E7E7FF" ForeColor="#4A3C8C" />
    <SelectedItemStyle BackColor="#738A9C" Font-Bold="True" ForeColor="#F7F7F7" />
    <HeaderStyle BackColor="#4A3C8C" Font-Bold="True" ForeColor="#F7F7F7" />
    <ItemTemplate>
        新闻 ID:
        <asp:Label ID="IDLabel" runat="server" Text='<%= Eval("ID") %>' />
        <br />
        新闻编号:
        <asp:Label ID="TITLELabel" runat="server" Text='<%= Eval("TITLE") %>' />
        <br />
        <asp:Button ID="Button1" runat="server" Text="删除"
        CommandName="delete" CommandArgument='<%= Eval("ID") %>' />
    </ItemTemplate>
</asp:DataList>
```

上述代码创建了一个 **DataList** 控件并配置了按钮控件，并将按钮控件的 **CommandName** 属性配置为“**delete**”，则触发该按钮则会引发 **DeleteCommand** 事件。在属性窗口中找到 **DeleteCommand** 事件，双击【**DeleteCommand**】连接系统会自动生成 **DeleteCommand** 事件相应的方法。当生成了 **DeleteCommand** 事件后，可以在代码段中编写相应的方法，示例代码如下所示。

```
protected void DataList1_DeleteCommand(object source, DataListCommandEventArgs e)
{
    Label1.Text = e.CommandArgument.ToString()+"被执行";
}
```

当用户单击了相应的按钮时会触发 **DeleteCommand** 事件。开发人员能够通过传递过来的参数，可以编写相应的方法，运行结果如图 8-33 所示。





图 8-33 触发 DeleteCommand 事件

程序运行后，当用户单击了相应的按钮时，开发人员可以通过获取传递的 **CommandArgument** 参数的值来编写相应的方法从而执行实现不同的应用。

## 8.4 数据列表控件（GridView）

**GridView** 是 ASP.NET 中功能非常丰富的控件之一，它可以以表格的形式显示数据库的内容并通过数据源控件自动绑定和显示数据。开发人员能够通过配置数据源控件对 **GridView** 中的数据进行选择、排序、分页、编辑和删除功能进行配置。**GridView** 控件还能够指定自定义样式，在没有任何数据时可以自定义无数据时的 UI 样式。

### 1. 建立 GridView 控件

**GridView** 控件为开发人员提供了强大的管理方案，同样 **GridView** 也支持内置格式，单击【自动套用格式】连接可以选择 **GridView** 中的默认格式，如图 8-34 所示。

**GridView** 是以表格为表现形式，**GridView** 包括行和列，通过配置相应的属性能够编辑相应的行的样式，同样也可以选择【编辑列】选项来编写相应的列的样式，如图 8-35 所示。

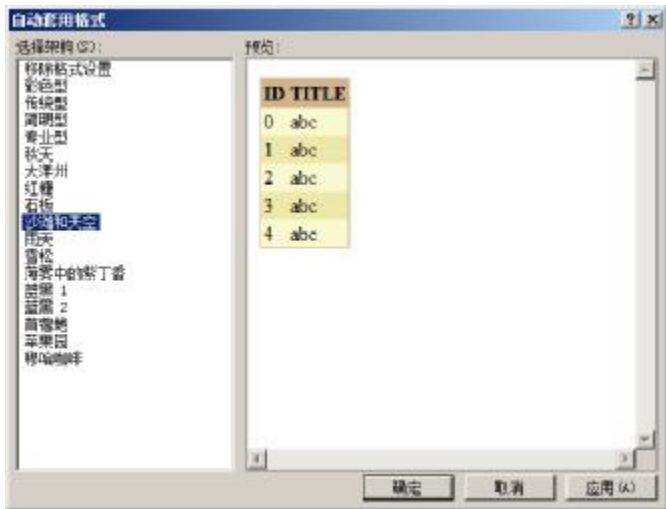


图 8-34 自动套用格式

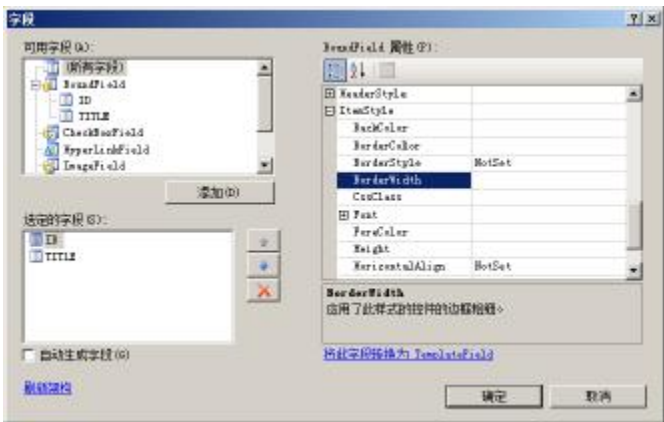


图 8-35 编辑列

**GridView** 控件提供两个用户绑定到数据的选项，其一是使用 **DataSourceID** 进行数据绑定，这种方法通常情况下是绑定数据源控件；而另一种则是使用 **DataSource** 属性进行数据绑定，这种方法能够将 **GridView** 控件绑定到包括 **ADO.NET** 数据和数据读取器内的各种对象。



使用 **DataSourceID** 进行数据绑定，可以让 **GridView** 控件能够自动的处理分页、选择等操作，如图 8-36 所示。而使用 **DataSource** 属性进行数据绑定，则需要开发人员通过编程实现分页等操作。**GridView** 控件能够自定义字段，单击【添加列】按钮，可以选择相应类型的列。在添加列选项中，**GridView** 控件支持多种列类型的列，包括复选框、图片、单选框、超链接等，如图 8-37 所示。



图 8-36 可选相应操作

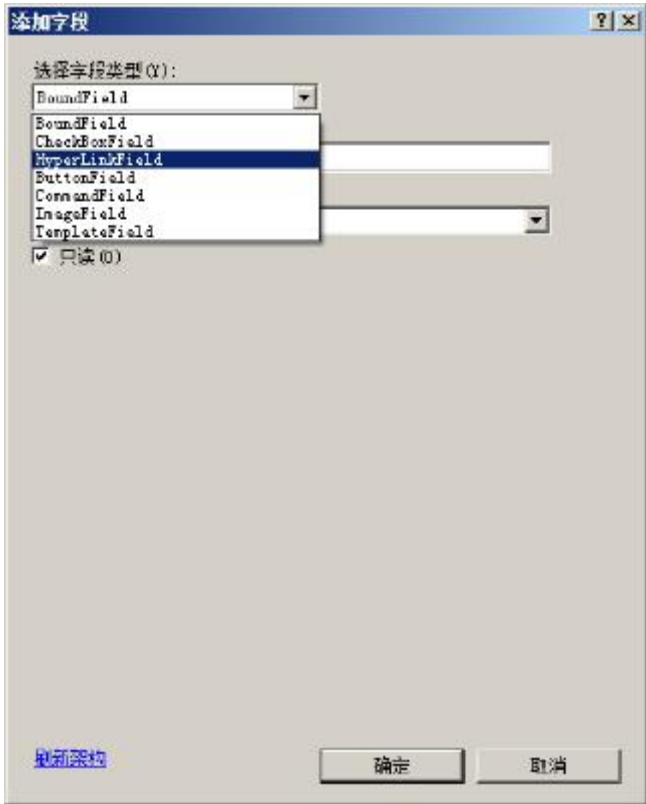


图 8-37 添加字段

添加自定义字段，**GridView** 控件支持从数据源中读取相应的数据源来配置相应的字段，来让开发人员自定义的读取数据源中的相应字段来自定义开发，如图 8-38 所示。当选择从数据源中获取文本，可以通过 **Format** 的形式编写相应的文本。例如，从数据源中获取 **title** 列，而显示文本为“这是一个标题：**title** 值”，则可以编写为“这是一个标题：**{0}**”，如图 8-39 所示。

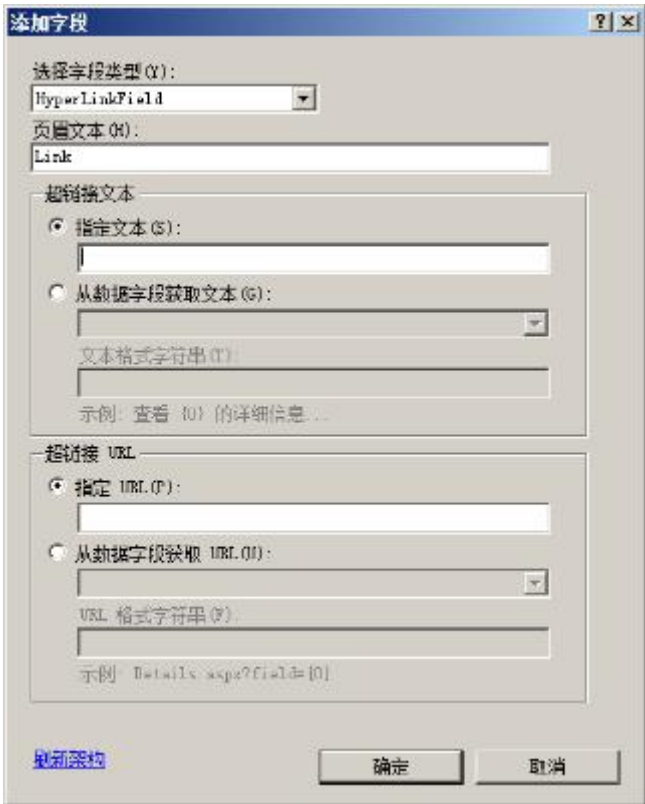


图 8-38 添加字段



图 8-39 格式化字符串输出

配置完成后，**GridView** 控件的 **HTML** 标签生成代码如下所示：

```
<asp:GridView ID="GridView1" runat="server" AllowPaging="True"
    AllowSorting="True" AutoGenerateColumns="False"
    BackColor="LightGoldenrodYellow" BorderColor="Tan" BorderWidth="1px"
```

```

CellPadding="2" DataKeyNames="ID" DataSourceID="SqlDataSource1"
ForeColor="Black" GridLines="None" Width="100%">
    <FooterStyle BackColor="Tan" />
    <Columns>
        <asp:BoundField DataField="ID" HeaderText="ID" InsertVisible="False"
            ReadOnly="True" SortExpression="ID" />
        <asp:BoundField DataField="TITLE" HeaderText="TITLE" SortExpression="TITLE" />
        <asp:HyperLinkField DataNavigateUrlFields="ID"
            DataNavigateUrlFormatString="Default.aspx?uid={0}" DataTextField="TITLE"
            DataTextFormatString="Title:{0}" HeaderText="Link" />
    </Columns>
    <PagerStyle BackColor="PaleGoldenrod" ForeColor="DarkSlateBlue"
        HorizontalAlign="Center" />
    <SelectedRowStyle BackColor="DarkSlateBlue" ForeColor="GhostWhite" />
    <HeaderStyle BackColor="Tan" Font-Bold="True" />
    <AlternatingRowStyle BackColor="PaleGoldenrod" />
</asp:GridView>

```

上述代码使用了一个默认格式，并新建了一个超链接文本类型的列，当单击超文本链接，则会跳转到另一个页面。

## 2. GridView 控件的常用事件

**GridView** 支持多个事件，通常对 **GridView** 控件进行排序、选择等操作时，同样会引发事件，当创建当前行或将当前行绑定至数据时发生的事件，同样，单击一个命令控件时也会引发事件。**GridView** 控件常用的事件如下所示。

- ❑ **RowCommand:** 在 **GridView** 控件中单击某个按钮时发生。此事件通常用于在该控件中单击某个按钮时执行某项任务。
- ❑ **PageIndexChanging:** 在单击页导航按钮时发生，但在 **GridView** 控件执行分页操作之前。此事件通常用于取消分页操作。
- ❑ **PageIndexChanged:** 在单击页导航按钮时发生，但在 **GridView** 控件执行分页操作之后。此事件通常用于在用户定位到该控件中不同的页之后需要执行某项任务时。
- ❑ **SelectedIndexChanging:** 在单击 **GridView** 控件内某一行的 **Select** 按钮（其 **CommandName** 属性设置为“**Select**”的按钮）时发生，但在 **GridView** 控件执行选择操作之前。此事件通常用于取消选择操作。
- ❑ **SelectedIndexChanged:** 在单击 **GridView** 控件内某一行的 **Select** 按钮时发生，但在 **GridView** 控件执行选择操作之后。此事件通常用于在选择了该控件中的某行后执行某项任务。
- ❑ **Sorting:** 在单击某个用于对列进行排序的超链接时发生，但在 **GridView** 控件执行排序操作之前。此事件通常用于取消排序操作或执行自定义的排序例程。
- ❑ **Sorted:** 在单击某个用于对列进行排序的超链接时发生，但在 **GridView** 控件执行排序操作之后。此事件通常用于在用户单击对列进行排序的超链接之后执行某项任务。
- ❑ **RowDataBound:** 在 **GridView** 控件中的某个行被绑定到一个数据记录时发生。此事件通常用于在某个行被绑定到数据时修改该行的内容。
- ❑ **RowCreated:** 在 **GridView** 控件中创建新行时发生。此事件通常用于在创建某个行时修改该行的布局或外观。
- ❑ **RowDeleting:** 在单击 **GridView** 控件内某一行的 **Delete** 按钮（其 **CommandName** 属性设置为“**Delete**”的按钮）时发生，但在 **GridView** 控件从数据源删除记录之前。此事件通常用于取消删除操作。
- ❑ **RowDeleted:** 在单击 **GridView** 控件内某一行的 **Delete** 按钮时发生，但在 **GridView** 控件从数据源删除记录之后。此事件通常用于检查删除操作的结果。
- ❑ **RowEditing:** 在单击 **GridView** 控件内某一行的 **Edit** 按钮（其 **CommandName** 属性设置为“**Edit**”的按钮）时发生，但在 **GridView** 控件进入编辑模式之前。此事件通常用于取消编辑操作。
- ❑ **RowCancelingEdit:** 在单击 **GridView** 控件内某一行的 **Cancel** 按钮（其 **CommandName** 属性设置

为“Cancel”的按钮）时发生，但在 **GridView** 控件退出编辑模式之前。此事件通常用于停止取消操作。

- ❑ **RowUpdating**: 在单击 **GridView** 控件内某一行的 **Update** 按钮（其 **CommandName** 属性设置为“Update”的按钮）时发生，但在 **GridView** 控件更新记录之前。此事件通常用于取消更新操作。
- ❑ **RowUpdated**: 在单击 **GridView** 控件内某一行的 **Update** 按钮时发生，但在 **GridView** 控件更新记录之后。此事件通常用来检查更新操作的结果。
- ❑ **DataBound**: 此事件继承自 **BaseDataBoundControl** 控件，在 **GridView** 控件完成到数据源的绑定后发生。

需要指定相应的事件，则必须添加一个 **RowCommand** 事件，**GridView** 控件 **HTML** 代码如下所示。

```
<asp:GridView ID="GridView1" runat="server" AllowPaging="True"
    AllowSorting="True" AutoGenerateColumns="False"
    BackColor="LightGoldenrodYellow" BorderColor="Tan" BorderWidth="1px"
    CellPadding="2" DataKeyNames="ID" DataSourceID="SqlDataSource1"
    ForeColor="Black" GridLines="None" onrowcommand="GridView1_RowCommand"
    Width="100%">
    <FooterStyle BackColor="Tan" />
    <Columns>
        <asp:BoundField DataField="ID" HeaderText="ID" InsertVisible="False"
            ReadOnly="True" SortExpression="ID" />
        <asp:BoundField DataField="TITLE" HeaderText="TITLE" SortExpression="TITLE" />
        <asp:HyperLinkField DataNavigateUrlFields="ID"
            DataNavigateUrlFormatString="Default.aspx?uid={0}" DataTextField="TITLE"
            DataTextFormatString="Title:{0}" HeaderText="Link" />
        <asp:ButtonField ButtonType="Button" CommandName="
            Select" HeaderText="选择按钮" ShowHeader="True" Text="按钮" />
    </Columns>
    <PagerStyle BackColor="PaleGoldenrod" ForeColor="DarkSlateBlue"
        HorizontalAlign="Center" />
    <SelectedRowStyle BackColor="DarkSlateBlue" ForeColor="GhostWhite" />
    <HeaderStyle BackColor="Tan" Font-Bold="True" />
    <AlternatingRowStyle BackColor="PaleGoldenrod" />
</asp:GridView>
```

上述代码创建了一个 **GridView** 控件，并增加了一个按钮控件，并且为按钮控件的 **CommandName** 属性赋值为 **Select**，当单击按钮控件时，则会触发 **RowCommand** 事件，**CS** 页面代码如下所示。

```
protected void GridView1_RowCommand(object sender, GridViewCommandEventArgs e)
{
    Label1.Text = e.CommandName + "事件被触发";
}
```

当单击按钮时，**GridView** 控件会选择相应的行。在 **GridView** 控件的 **RowCommand** 事件中，同样可以通过 **GridView** 控件的中按钮的 **CommandArgument** 属性获取相应的操作并执行相应代码。**GridView** 控件运行结果如图 8-40 和图 8-41 所示。





图 8-40 GridView 控件的事件



图 8-41 触发 Select 选择事件

注意：在执行其他事件时，如 RowDeleted、GridView 控件首先执行 RowDataBound 代码，然后执行 RowCommnad、RowDeleting 以及 RowDeleted 等事件。

8.5 数据绑定控件（FormView）

FormView 控件只能显示数据库中一行的数据，并且提供对数据的分页操作，FormView 控件可以以一种不规则的外观来将数据呈现给用户。FormView 控件同样支持模板，以方便开发人员自定义 FormView 控件的 UI，FormView 控件支持的模板如下所示：

- ❑ ItemTemplate: 用于在 FormView 种呈现一个特殊的记录。
- ❑ HeaderTemplate: 用于指定一个可选的页眉行。
- ❑ FooterTemplate: 用于指定一个可选的页脚行。
- ❑ EmptyDataTemplate: 当 FormView 的 DataSource 缺少记录的时候，EmptyDataTemplate 将会代替 ItemTemplate 来生成控件的标记语言。
- ❑ PagerTemplate: 如果 FormView 启用了分页的话，这个模板可以用于自定义分页的界面。
- ❑ EditItemTemplate / InsertItemTemplate: 如果 FormView 支持编辑或插入功能，那么这两种模板可以用于自定义相关的界面。

通过编辑 ItmTemplate，能够自定义 HTML 以呈现数据，这种情况很像 Repeater 控件。FormView 控件同样支持自动套用格式，选择【自动套用格式】选项就能够为 FormView 控件选择默认格式，选择后如图 8-42 所示。



图 8-42 自定义 FormView 控件

当 FormView 控件界面编写完成后，HTML 代码如下所示。



```
<asp:FormView ID="FormView1" runat="server" AllowPaging="True"
BackColor="White" BorderColor="#3366CC" BorderStyle="None" BorderWidth="1px"
CellPadding="4" DataKeyNames="ID" DataSourceID="SqlDataSource1"
GridLines="Both" Width="100%">
  <FooterStyle BackColor="#99CCCC" ForeColor="#003399" />
  <RowStyle BackColor="White" ForeColor="#003399" />
  <EditItemTemplate>
    ID:
    <asp:Label ID="IDLabel1" runat="server" Text='<%# Eval("ID") %>' /><br />
    TITLE:
    <asp:TextBox ID="TITLETextBox" runat="server" Text='<%# Bind("TITLE") %>' /><br />
    <asp:LinkButton ID="UpdateButton" runat="server" CausesValidation="True"
      CommandName="Update" Text="更新" />
    <asp:LinkButton ID="UpdateCancelButton" runat="server"
      CausesValidation="False" CommandName="Cancel" Text="取消" />
  </EditItemTemplate>
  <InsertItemTemplate>
    TITLE:
    <asp:TextBox ID="TITLETextBox" runat="server" Text='<%# Bind("TITLE") %>' /> <br />
    <asp:LinkButton ID="InsertButton" runat="server" CausesValidation="True"
      CommandName="Insert" Text="插入" />
    <asp:LinkButton ID="InsertCancelButton" runat="server"
      CausesValidation="False" CommandName="Cancel" Text="取消" />
  </InsertItemTemplate>
  <ItemTemplate>
    新闻编号:
    <asp:Label ID="IDLabel" runat="server" Text='<%# Eval("ID") %>' /><br />
    新闻标题:
    <asp:Label ID="TITLELabel" runat="server" Text='<%# Bind("TITLE") %>' /><br />
  </ItemTemplate>
  <PagerStyle BackColor="#99CCCC" ForeColor="#003399" HorizontalAlign="Left" />
  <HeaderStyle BackColor="#003399" Font-Bold="True" ForeColor="#CCCCFF" />
  <EditRowStyle BackColor="#009999" Font-Bold="True" ForeColor="#CCFF99" />
</asp:FormView>
```

上述代码创建了 **FormView** 控件，并为 **FormView** 控件自定义了若干模板。刚才只是编写了 **ItemTemplate** 模板，但是 **EdititemTemplate** 也已经在 **HTML** 标签中生成。

注意：**FormView** 控件模板中的相应数据字段也是通过数据绑定语法实现的，如<%# Eval("字段名称") %>。

**FormView** 控件同样支持对当前数据的更新、删除、选择等操作。当拖放一个按钮控件时，可以选择 **DataBindings** 来为按钮控件的属性做相应的配置，如图 8-43 所示。



图 8-43 DataBindings

当单击 **FormView** 中的控件时，会触发 **Command** 事件，要使用 **FormView** 控件进行更新等操作，必须在相应的模式下更新才行，例如当需要更新操作时，则必须在编辑模式下才能进行更新操作。当执行相应的操作时，例如更新操作，则必须在编辑模式下进行操作，并需要使用 **ItemUpdated** 事件来编写相应的更新事件。编写 **FormView** 控件中的 **ItemTemplate** 和 **EditItemTemplate**，生成的 **HTML** 代码如下所示。

```
<asp:FormView ID="FormView1" runat="server" AllowPaging="True"
    BackColor="White" BorderColor="#3366CC" BorderStyle="None" BorderWidth="1px"
    CellPadding="4" DataKeyNames="ID" DataSourceID="SqlDataSource1"
    GridLines="Both" Width="100%" onitemcommand="FormView1_ItemCommand"
    onitemupdated="FormView1_ItemUpdated">
    <FooterStyle BackColor="#99CCCC" ForeColor="#003399" />
    <RowStyle BackColor="White" ForeColor="#003399" />
    <EditItemTemplate>
        新闻编号:
        <asp:Label ID="IDLabel1" runat="server" Text='<%=# Eval("ID") %>' />
        <br />
        新闻标题:
        <asp:TextBox ID="TITLETextBox" runat="server" Text='<%=# Bind("TITLE") %>' />
        <br />
        <asp:LinkButton ID="UpdateButton" runat="server" CausesValidation="True"
            CommandName="Update" Text="更新" />
        &nbsp;<asp:LinkButton ID="UpdateCancelButton" runat="server"
            CausesValidation="False" CommandName="Cancel" Text="取消" />
    </EditItemTemplate>
    <InsertItemTemplate>
        TITLE:
        <asp:TextBox ID="TITLETextBox" runat="server" Text='<%=# Bind("TITLE") %>' />
        <br />
        <asp:LinkButton ID="InsertButton" runat="server" CausesValidation="True"
            CommandName="Insert" Text="插入" />
        <asp:LinkButton ID="InsertCancelButton" runat="server"
            CausesValidation="False" CommandName="Cancel" Text="取消" />
    </InsertItemTemplate>
    <ItemTemplate>
        新闻编号:
        <asp:Label ID="IDLabel" runat="server" Text='<%=# Eval("ID") %>' /><br />
        新闻标题:
        <asp:Label ID="TITLELabel" runat="server" Text='<%=# Bind("TITLE") %>' /><br />
    </ItemTemplate>
    <PagerStyle BackColor="#99CCCC" ForeColor="#003399" HorizontalAlign="Left" />
    <HeaderStyle BackColor="#003399" Font-Bold="True" ForeColor="#CCCCFF" />
    <EditRowStyle BackColor="#009999" Font-Bold="True" ForeColor="#CCFF99" />
</asp:FormView>
```

上述代码编写了 **FormView** 控件中的 **ItemTemplate** 和 **EditItemTemplate**。在页面中，增加了按钮来切换 **FormView** 控件的编辑模式，按钮控件代码如下所示。

```
<asp:Button ID="Button2" runat="server" onclick="Button2_Click" Text="Edit" />
```

当单击按钮时，**FormView** 控件会更改其编辑模式，示例代码如下所示。

```
protected void Button2_Click(object sender, EventArgs e)
{
    FormView1.ChangeMode(FormViewMode.Edit); //更改编辑模式
}
```

当更改了编辑模式后，**FormView** 控件允许在当前页面直接更改数据的值，并通过 **ItemUpdated** 进行更新，示例代码如下所示。

```
protected void FormView1_ItemUpdated(object sender, FormViewUpdatedEventArgs e)
{
    Label1.Text = "相应值被更新"; //提示已被更改
    FormView1.ChangeMode(FormViewMode.ReadOnly); //更改编辑模式
}
```

}

上述代码允许开发人员能够自定义数据操作，通过对对象 **e** 的值来获取相应的数据字段的值并进行更新，运行结果如图 8-44 和 8-45 所示。



图 8-44 视图模式

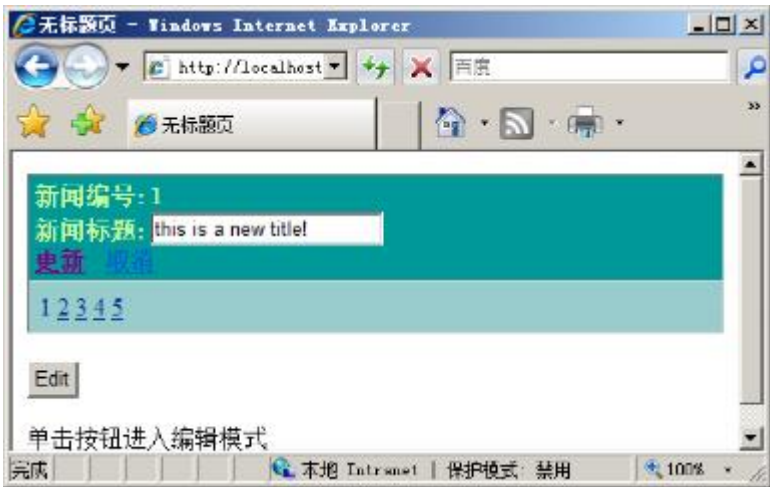


图 8-45 编辑模式

当单击了其中的更新，则会触发 **ItemUpdated** 事件，开发人员能够通过编写 **ItemUpdated** 事件来进行相应的更新操作。值得注意的是，通常情况下数据源控件必须支持更新操作才能够执行更新，在配置数据源时，需要为更新语句进行配置。在配置和生成 SQL 语句中必须选择【高级】选项、勾选【生成 **update**、**insert**、**delete** 语句】复选框才能够让数据源控件支持更新等操作，如图 8-46 所示。



图 8-46 高级数据源配置

如果数据绑定控件需要使用 **Insert** 等语句时，则数据源控件需配置高级 SQL 生成选项，开发人员还能够数据源控件的 **HTML** 代码中进行相应的 SQL 语句的更改已达到自定义数据源控件的目的。

## 8.6 数据绑定控件（DetailsView）

**DetailsView** 控件与 **FormView** 在很多情况下非常类似，**DetailsView** 控件通常情况下也只能够显示一行的数据，同 **FormView**，**DetailsView** 控件支持对数据源控件中的数据进行插入、删除和更新。但是 **DetailsView** 控件与 **FormView** 控件不同的是，**DetailsView** 控件不支持 **ItemTemplate** 模板，这也就是说，**DetailsView** 控件是以一种表格的形式所呈现的。

相比之下，**DetailsView** 控件能够支持 **Ajax**，因为 **FormView** 控件完全由模板驱动，但是 **FormView** 控件对验证控件的支持较好。而 **DetailsView** 控件可以通过选择是否包括更新，删除等操作，而无需手动的添加相应的事件，比 **FormView** 控件更加方便，如图 8-47 和图 8-48 所示。



图 8-47 配置 DetailsView 任务



图 8-48 减少任务配置

当选择了【启用分页】选项后 DetailsView 控件就能够自动进行分页。开发人员还可以配置 PagerSettings 属性允许自定义 DetailsView 控件生成分页用户界面的外观，它将呈现向前和向后导航的方向控件，PagerSettings 属性的常用模式有：

- ☐ NextPrevious: 以前一个，下一个形式显示。
- ☐ NextPreviousFirstLast: 以前一个，下一个，最前一个，最后一个形式显示。
- ☐ Numeric: 以数字形式显示。
- ☐ NumericFirstLast: 以数字，最前一个，最后一个形式显示。

当完成配置 DetailsView 控件后，DetailsView 控件无需通过外部控件来转换 DetailsView 控件的编辑模式，DetailsView 控件自动会显示更新、插入、删除等按钮来更改编辑模式，如图 8-49 所示。

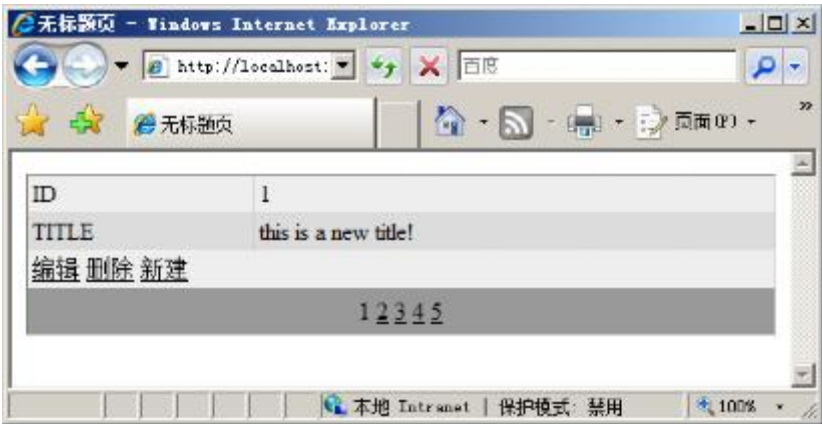


图 8-49 DetailsView 控件

编辑完成后，DetailsView 控件生成的 HTML 代码如下所示。

```
<asp:DetailsView ID="DetailsView1" runat="server" AllowPaging="True"
    AutoGenerateRows="False" BackColor="White" BorderColor="#999999"
    BorderStyle="None" BorderWidth="1px" CellPadding="3" DataKeyNames="ID"
    DataSourceID="SqlDataSource1" GridLines="Vertical" Height="50px" Width="100%">
  <FooterStyle BackColor="#CCCCCC" ForeColor="Black" />
  <RowStyle BackColor="#EEEEEE" ForeColor="Black" />
  <PagerStyle BackColor="#999999" ForeColor="Black" HorizontalAlign="Center" />
  <Fields>
    <asp:BoundField DataField="ID" HeaderText="ID" InsertVisible="False"
      ReadOnly="True" SortExpression="ID" />
    <asp:BoundField DataField="TITLE" HeaderText="TITLE" SortExpression="TITLE" />
    <asp:CommandField ShowDeleteButton="True" ShowEditButton="True"
      ShowInsertButton="True" />
  </Fields>
  <HeaderStyle BackColor="#000084" Font-Bold="True" ForeColor="White" />
  <EditRowStyle BackColor="#008A8C" Font-Bold="True" ForeColor="White" />
  <AlternatingRowStyle BackColor="#DCDCDC" />
</asp:DetailsView>
```

如上一节内容所讲，在数据源控件的配置中配置 SQL 语句，需要选择高级，勾选【生成 update、insert、delete 语句】复选框以支持自动生成更新、删除等语句的生成。当勾选了【生成 update、insert、delete 语句】



复选框后，数据源控件代码如下所示。

```
<asp:SqlDataSource ID="SqlDataSource1" runat="server"
    ConnectionString="<%= $ ConnectionStrings:mytableConnectionString %>"
    DeleteCommand="DELETE FROM [mynews] WHERE [ID] = @ID"
    InsertCommand="INSERT INTO [mynews] ([TITLE]) VALUES (@TITLE)"
    SelectCommand="SELECT * FROM [mynews]"
    UpdateCommand="UPDATE [mynews] SET [TITLE] = @TITLE WHERE [ID] = @ID">
    <DeleteParameters>
        <asp:Parameter Name="ID" Type="Int32" />
    </DeleteParameters>
    <UpdateParameters>
        <asp:Parameter Name="TITLE" Type="String" />
        <asp:Parameter Name="ID" Type="Int32" />
    </UpdateParameters>
    <InsertParameters>
        <asp:Parameter Name="TITLE" Type="String" />
    </InsertParameters>
</asp:SqlDataSource>
```

从上述代码可以看出，数据源控件自动生成了相应的 SQL 语句，如图 8-50 所示。当执行更新、删除等操作时，则会默认执行该语句。运行结果如图 8-51 所示。



图 8-50 更改相应字段的值

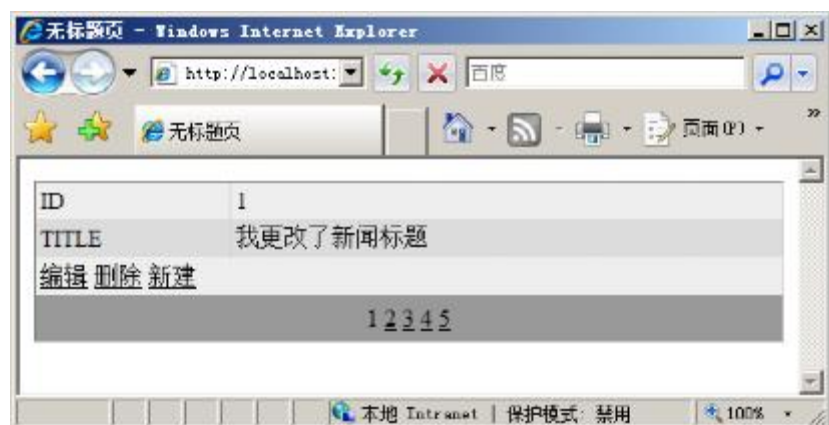


图 8-51 更改后的控件呈现

## 8.7 数据绑定控件（ListView）

**ListView** 控件是 ASP.NET 3.5 中新增的数据绑定控件，**ListView** 控件是介于 **GridView** 控件和 **Repeater** 之间的另一种数据绑定控件，相对于 **GridView** 来说，它有着更为丰富的布局手段，开发人员可以在 **ListView** 控件的模板内写任何 **HTML** 标记或者控件。相比于 **GridView** 和 **Repeater** 控件而言，**ListView** 支持的模板如下所示：

- ❑ **AlternatingItemTemplate**: 交替项目模板，用不同的标记显示交替的项目，便于查看者区别连续不断的项目。
- ❑ **EditItemTemplate**: 编辑项目模板，控制编辑时的项目显示。
- ❑ **EmptyDataTemplate**: 空数据模板，控制 **ListView** 数据源返回空数据时的显示。
- ❑ **EmptyItemTemplate**: 空项目模板，控制空项目的显示。
- ❑ **GroupSeparatorTemplate**: 组分隔模板，控制项目组内容的显示。
- ❑ **GroupTemplate**: 组模板，为内容指定一个容器对象，如一个表行、**div** 或 **span** 组件。
- ❑ **InsertItemTemplate**: 插入项目模板，用户插入项目时为其指定内容。
- ❑ **ItemSeparatorTemplate**: 项目分隔模板，控制项目之间内容的显示。
- ❑ **ItemTemplate** 项目模板：控制项目内容的显示。
- ❑ **LayoutTemplate**: 布局模板，指定定义容器对象的根组件，如一个 **table**、**div** 或 **span** 组件，它们包

装 **ItemTemplate** 或 **GroupTemplate** 定义的内容。

- ❑ **SelectedItemTemplate**: 已选择项目模板，指定当前选中的项目内容的显示。

其中最为常用的控件包括 **LayoutTemplate** 和 **ItemTemplate**，**LayoutTemplate** 为 **ListView** 控件指定了总的标记，而 **ItemTemplate** 指定的标记用于显示每个绑定的记录，用来编写 **HTML** 样式。**ListView** 控件能够自动套用 **HTML** 格式，如其他控件一样，可以选择默认模板，单击【配置 **ListView**】连接进行格式套用，如图 8-52 所示。



图 8-52 配置 **ListView**

开发人员能够选择相应的布局并选择相应的样式来确定 **ListView** 控件的界面，开发人员还可以通过选择【启用编辑】、【启用插入】等选项简化开发。

**注意：**当需要执行相应的数据操作时，数据源控件的高级选项都应该勾选。

当选择相应的布局方案和样式后，系统生成的 **ListView** 控件的 **HTML** 代码如下所示。

```
<asp:ListView ID="ListView1" runat="server" DataKeyNames="ID"
    DataSourceID="SqlDataSource1" InsertItemPosition="LastItem">
    <AlternatingItemTemplate>
        <li style="background-color: #FFF8DC;">ID:
            <asp:Label ID="IDLabel" runat="server" Text='<%# Eval("ID") %>' />
            <br />
            TITLE:
            <asp:Label ID="TITLELabel" runat="server" Text='<%# Eval("TITLE") %>' />
            <br />
            <asp:Button ID="EditButton" runat="server" CommandName="Edit" Text="编辑" />
            <asp:Button ID="DeleteButton" runat="server" CommandName="Delete" Text="删除" />
        </li>
    </AlternatingItemTemplate>
    <LayoutTemplate>
        <ul ID="itemPlaceholderContainer" runat="server"
            style="font-family: Verdana, Arial, Helvetica, sans-serif;">
            <li ID="itemPlaceholder" runat="server" />
        </ul>
        <div style="text-align: center;background-color: #CCCCCC;font-family: Verdana, Arial,
            Helvetica, sans-serif;color: #000000;">
            <asp:DataPager ID="DataPager1" runat="server">
                <Fields>
                    <asp:NextPreviousPagerField ButtonType="Button" ShowFirstPageButton="True"
                        ShowLastPageButton="True" />
                </Fields>
            </asp:DataPager>
        </div>
    </LayoutTemplate>
</asp:ListView>
```

```
</div>
</LayoutTemplate>
<InsertItemTemplate>
  <li style="">TITLE:
  <asp:TextBox ID="TITLETextBox" runat="server" Text='<%# Bind("TITLE") %>' />
  <br />
  <asp:Button ID="InsertButton" runat="server" CommandName="Insert" Text="插入" />
  <asp:Button ID="CancelButton" runat="server" CommandName="Cancel" Text="清除" />
  </li>
</InsertItemTemplate>
<SelectedItemTemplate>
  <li style="background-color: #008A8C;font-weight: bold;color: #FFFFFF;">ID:
  <asp:Label ID="IDLabel" runat="server" Text='<%# Eval("ID") %>' />
  <br />
  TITLE:
  <asp:Label ID="TITLELabel" runat="server" Text='<%# Eval("TITLE") %>' />
  <br />
  <asp:Button ID="EditButton" runat="server" CommandName="Edit" Text="编辑" />
  <asp:Button ID="DeleteButton" runat="server" CommandName="Delete" Text="删除" />
</li>
</SelectedItemTemplate>
<EmptyDataTemplate>
  未返回数据。
</EmptyDataTemplate>
<EditItemTemplate>
  <li style="background-color: #008A8C;color: #FFFFFF;">ID:
  <asp:Label ID="IDLabel1" runat="server" Text='<%# Eval("ID") %>' />
  <br />
  TITLE:
  <asp:TextBox ID="TITLETextBox" runat="server" Text='<%# Bind("TITLE") %>' />
  <br />
  <asp:Button ID="UpdateButton" runat="server" CommandName="Update" Text="更新" />
  <asp:Button ID="CancelButton" runat="server" CommandName="Cancel" Text="取消" />
  </li>
</EditItemTemplate>
<ItemTemplate>
  <li style="background-color: #DCDCDC;color: #000000;">ID:
  <asp:Label ID="IDLabel" runat="server" Text='<%# Eval("ID") %>' />
  <br />
  TITLE:
  <asp:Label ID="TITLELabel" runat="server" Text='<%# Eval("TITLE") %>' />
  <br />
  <asp:Button ID="EditButton" runat="server" CommandName="Edit" Text="编辑" />
  <asp:Button ID="DeleteButton" runat="server" CommandName="Delete" Text="删除" />
  </li>
</ItemTemplate>
<ItemSeparatorTemplate>
  <br />
</ItemSeparatorTemplate>
</asp:ListView>
```

上述代码定义了 **ListView** 控件，系统默认创建了相应的模板，开发人员能够编辑相应的模板样式来为不同的编辑模式显示不同的用户界面。同时，用户可以无需代码实现就能够实现删除，更新以及添加等操作，运行结果如图 8-53 所示。

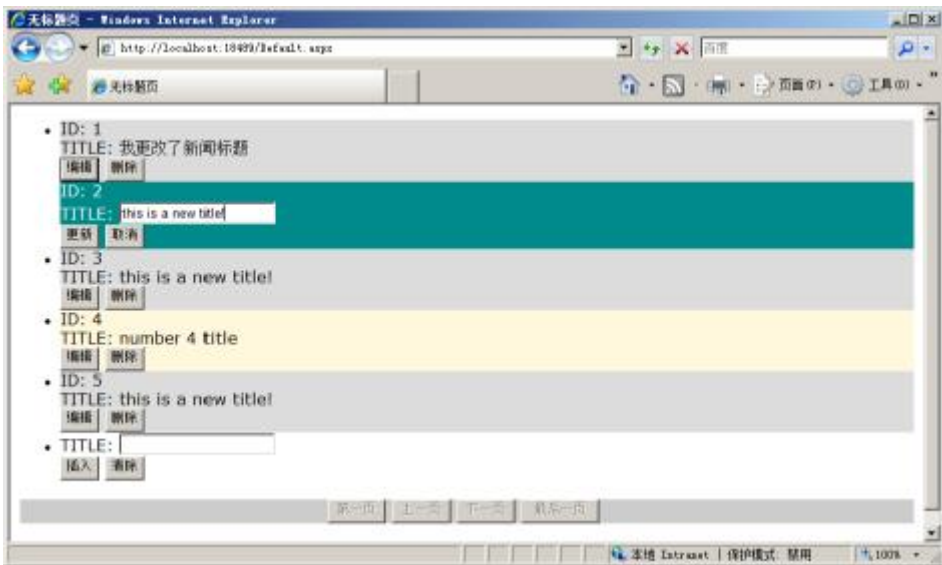


图 8-53 ListView 控件

**LayoutTemplate** 和 **ItemTemplate** 是标识定义控件的主要布局的根模板。通常情况下，它包含一个占位符对象，例如表行 **tr** 或 **div** 元素。此元素将由 **ItemTemplate** 模板或 **GroupTemplate** 模板中定义的内容替换。

如果需要定义自定义用户界面，则必须使用 **LayoutTemplate** 模板可以作为 **ListView** 控件的父容器。**LayoutTemplate** 模板是 **ListView** 控件所必需的。相同的是，**LayoutTemplate** 内容也需要包含一个占位符控件。占位符控件必须将包含 **runat="server"** 属性，并且将 **ID** 属性设置为 **ItemPlaceholderID** 或 **GroupPlaceholderID** 属性的值，示例代码如下所示。

```
<ItemTemplate>
    <td runat="server" style="background-color:#DCDCDC;color: #000000;">
        ID:
        <asp:Label ID="IDLabel" runat="server" Text='<%= Eval("ID") %>' /><br />
        TITLE:
        <asp:Label ID="TITLELabel" runat="server" Text='<%= Eval("TITLE") %>' /><br />
        <asp:Button ID="DeleteButton" runat="server" CommandName="Delete" Text="删除" /><br />
        <asp:Button ID="EditButton" runat="server" CommandName="Edit" Text="编辑" /><br />
    </td>
</ItemTemplate>
```

**ListView** 控件的事件和 **FormView** 控件的事件基本相同，同样可以为 **ListView** 控件执行更新、删除或添加等事件编写相应的代码。当执行更新前、更新时都可以触发相应的事件，示例代码如下所示。

```
protected void ListView1_ItemUpdated(object sender, ListViewEventArgs e)
{
    Label1.Text = "更新已经发生"; //触发更新事件
}
```

当运行后，则会触发 **ItemUpdated** 事件，运行结果如图 8-54 所示。

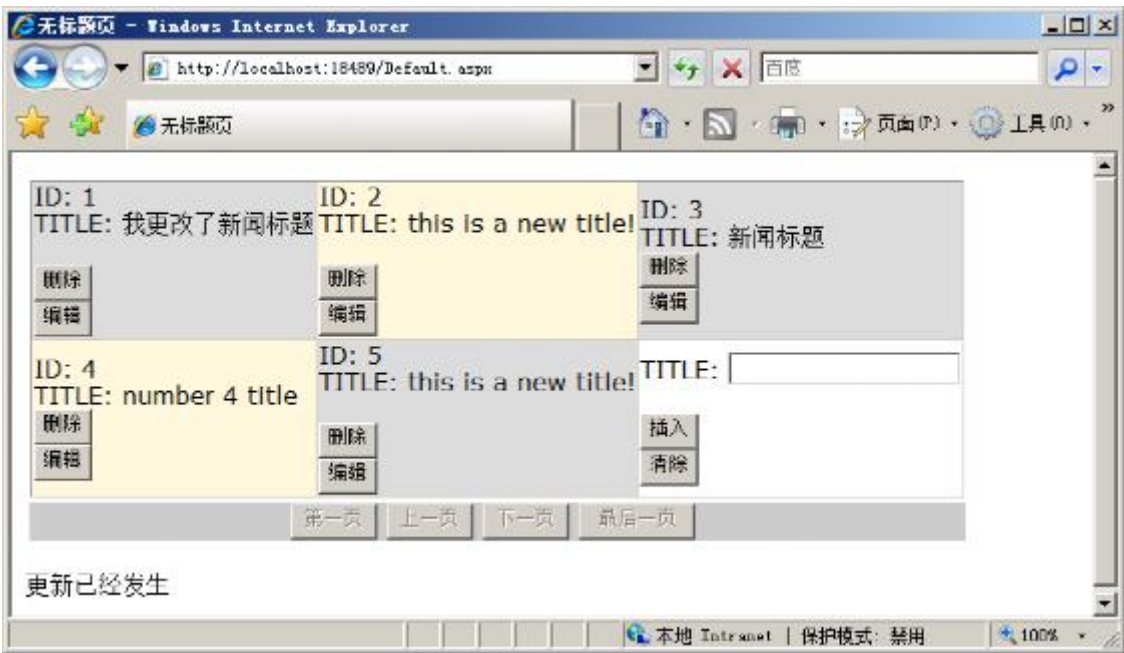


图 8-54 ItemUpdated 事件



**ListView** 控件不仅能够支持 **FormView** 控件的事件，而 **ListView** 控件具有更多的布局手段。**ListView** 控件能为开发人员在开发中提供极大的遍历，当如果需要进行相应的数据操作，又需要快捷的显式数据和添加数据时，**ListView** 控件是极佳的选择。

## 8.8 数据绑定控件（DataPager）

**DataPager** 控件通过实现 **IPageableItemContainer** 接口实现了控件的分页。在 **ASP.NET 3.5** 中，**ListView** 控件适合可以使用 **DataPager** 控件进行分页操作。要在 **ListView** 中使用 **DataPager** 控件只需要在 **LayoutTemplate** 模板中加入 **DataPager** 控件。**DataPager** 控件包括两种样式，一种是“上一页/下一页”样式，第二种是“数字”样式，如图 8-55 和图 8-56 所示。



图 8-55 文本样式

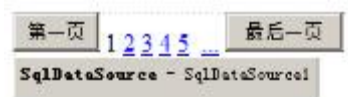


图 8-56 数字样式

当使用“上一页/下一页”样式时，**DataPager** 控件的 **HTML** 实现代码如下所示。

```
<asp:DataPager ID="DataPager1" runat="server">
  <Fields>
    <asp:NextPreviousPagerField ButtonType="Button" ShowFirstPageButton="True"
      ShowLastPageButton="True" />
  </Fields>
</asp:DataPager>
```

当使用“数字”样式时，**DataPager** 控件的 **HTML** 实现代码如下所示。

```
<asp:DataPager ID="DataPager1" runat="server">
  <Fields>
    <asp:NextPreviousPagerField ButtonType="Button" ShowFirstPageButton="True"
      ShowNextPageButton="False" ShowPreviousPageButton="False" />
    <asp:NumericPagerField />
    <asp:NextPreviousPagerField ButtonType="Button" ShowLastPageButton="True"
      ShowNextPageButton="False" ShowPreviousPageButton="False" />
  </Fields>
</asp:DataPager>
```

除了默认的方法来显示分页样式，还可以通过向 **DataPager** 中的 **Fields** 中添加 **TemplatePagerField** 的方法来自定义分页样式。在 **TemplatePagerField** 中添加 **PagerTemplate**，在 **PagerTemplate** 中添加任何服务器控件，这些服务器控件可以通过实现 **TemplatePagerField** 的 **OnPagerCommand** 事件来实现自定义分页。

## 8.9 小结

本章介绍了有关 **ASP.NET** 中绑定数据和数据源相关的控件，在 **ASP.NET** 中，这些控件强大的功能让开发变得更加的简单。在 **ASP.NET** 中，正是因为这些数据源控件和数据绑定控件，让开发人员在页面开发时，无需更多的操作即可实现强大的功能，解决了在传统的 **ASP** 中难以解决的问题。本章还包括：

- ❑ **ADO.NET**：讲解了 **ADO.NET**，并介绍了使用 **ADO.NET** 连接数据库。
- ❑ **数据源控件**：包括 **SqlDataSource** 等常用的数据源控件，并一步步的介绍了数据源控件的配置。
- ❑ **重复列表控件**：讲解了如 **Repeater** 之类的重复列表控件。
- ❑ **数据绑定控件**：讲解了常用的数据绑定控件并使用数据绑定控件对数据进行更新，删除等操作。

数据操作无论是在 **Web** 开发还是在 **Win Form** 开发中，都是要经常使用的，数据控件能够极大的简化开发人员对数据的操作，让开发更加迅速。

## 第 9 章 ASP.NET 操作数据库

通过对 **ADO.NET** 的基本讲解，以及讲解了一些数据源控件的基本用法后，本章将介绍一些 **ASP.NET** 操作数据库的高级用法，包括使用 **SQLHelper**，以及数据源控件对数据的操作。本章是对前面的数据库知识的一种补充和提升。

### 9.1 使用 ADO.NET 操作数据库

上一章中介绍了 **ADO.NET** 的基本概念、**ADO.NET** 的对象，以及如何使用 **ADO.NET**。使用 **ADO.NET** 能够极大的方便开发人员对数据库进行操作而无需关心数据库底层之间的运行，**ADO.NET** 不仅包括多个对象，同样包括多种方法，这些方法都可以用来执行开发人员指定的 **SQL** 语句，但是这些方法实现过程又不尽相同，本节将介绍 **ADO.NET** 中数据的操作方法。

#### 9.1.1 使用 ExecuteReader()操作数据库

使用 **ExecuteReader()**操作数据库，**ExecuteReader()**方法返回的是一个 **SqlDataReader** 对象或 **OleDbDataReader**对象。当使用 **DataReader**对象时，不会像 **DataSet**那样提供无连接的数据库副本，**DataReader**类被设计为产生只读、只进的数据流。这些数据流都是从数据库返回的。所以，每次的访问或操作只有一个记录保存在服务器的内存中。

相比与 **DataSet**而言，**DataReader**具有较快的访问能力，并且能够使用较少的服务器资源。**DataReader**对象提供了“游标”形式的读取方法，当从结果中读取了一行，则“游标”会继续读取到下一行。通过 **Read**方法可以判断数据是否还有下一行，如果存在数据，则继续运行并返回 **true**，否则返回 **false**。示例代码如下所示。

```
string str = "server=(local);database=mytable;uid='sa';pwd='sa'";
SqlConnection con = new SqlConnection(str);
con.Open();                                //打开连接
string strsql = "select * from mynews";     //SQL 查询语句
SqlCommand cmd = new SqlCommand(strsql, con); //初始化 Command 对象
SqlDataReader rd = cmd.ExecuteReader();     //初始化 DataReader 对象
while (rd.Read())
{
    Response.Write(rd["title"].ToString()); //通过索引获取列
}
```

**DataReader**可以提高执行效率，有两种方式可以提高代码的性能，一种是基于序号的查询；第二种情况则是使用适当的 **Get**方法来查询。一般来说，在数据库的设计中，需要设计索引键或主键来标识，在主键的设计中，自动增长类型是经常使用的，自动增长类型通常为整型，所以基于序号的查询可以使用 **DataReader**，示例代码如下所示。

```
string str = "server=(local);database=mytable;uid='sa';pwd='sa'"; //设置连接字符串
SqlConnection con = new SqlConnection(str);                       //创建连接对象
con.Open();                                                         //打开连接
string strsql = "select * from mynews where id=1 order by id desc"; //按标识查询
SqlCommand cmd = new SqlCommand(strsql, con);                     //创建 Command 对象
SqlDataReader rd = cmd.ExecuteReader();                             //创建 DataReader 对象
```

```
while (rd.Read()) //遍历数据库
{
    Response.Write(rd["title"].ToString()); //读取相应行的信息
}
```

当使用 **ExecuteReader()**操作数据库时，会遇到知道某列的名称而不知道某列的号的情况，这种情况可以通过使用 **DataReader** 对象的 **GetOrdinal()**方法获取相应的列号。此方法接收一个列名并返回此列名所在的列号，示例代码如下所示。

```
string str = "server=(local);database=mytable;uid=sa;pwd=sa"; //创建连接字符串
SqlConnection con = new SqlConnection(str); //创建连接对象
con.Open(); //打开连接
string strsql = "select * from mynews where id=1 order by id desc"; //创建执行 SQL 语句
SqlCommand cmd = new SqlCommand(strsql, con); //创建 Command 对象
SqlDataReader rd = cmd.ExecuteReader(); //创建 DataReader 对象
int id = rd.GetOrdinal("title"); //使用 GetOrdinal 方法获取 title 列的列号
while (rd.Read()) //遍历 DataReader 对象
{
    Label1.Text = "新闻 id 是" + rd["id"]; //输出对象的值
}
```

当完成数据库操作时，需要关闭数据库连接，**DataReader** 对象在调用 **Close()**方法即关闭与数据库的连接，如果在没有关闭之前又打开另一个连接，系统会抛出异常。示例代码如下所示。

```
rd.Close(); //关闭 DataReader 对象
```

**ExecuteReader()**可以执行相应的 **SQL** 语句，例如插入、更新以及删除等，当需要执行插入、更新或删除时，可以使用 **ExecuteReader()**进行数据操作，示例代码如下所示。

```
string str = "server=(local);database=mytable;uid=sa;pwd=sa"; //创建连接字符串
SqlConnection con = new SqlConnection(str); //创建连接对象
con.Open(); //打开连接
string strsql = "insert into mynews values ('执行更新后的标题')"; //创建执行 SQL 语句
SqlCommand cmd = new SqlCommand(strsql, con); //创建 Command 对象
SqlDataReader rd = cmd.ExecuteReader(); //使用 ExecuteReader()方法
while (rd.Read()) //读取数据库
{
    Response.Write(rd["title"].ToString() + "<hr/>");
}
rd.Close(); //关闭 DataReader 对象
Response.Redirect("ExecuteReader.aspx");
```

当执行了插入、删除等数据库操作时，**ExecuteReader** 返回为空的 **DataReader** 对象。当使用 **Read** 方法遍历读取数据库时，并不会显示相应的数据信息，因为不是查询语句，则返回一个没有任何数据的 **System.Data.OleDb.OleDbDataReader** 类型的集（EOF），但是 **ExecuteReader** 方法可以执行 **SQL** 语句。如图 9-1 所示。



图 9-1 ExecuteReader()执行查询和事务处理

使用 **ExecuteReader()**操作数据库，通常情况下是使用 **ExecuteReader()**进行数据库查询操作，使用 **ExecuteReader()**查询数据库能够提升查询效率，而如果需要进行数据库事务处理的话，**ExecuteReader()**方法并不是理想的选择。

### 9.1.2 使用 ExecuteNonQuery()操作数据库

使用 **ExecuteNonQuery()**操作数据库时，**ExecuteNonQuery()**并不返回 **DataReader** 对象，返回的是一个整型的值，代表执行某个 **SQL** 语句后，在数据库中影响的行数，示例代码如下所示。

```
string str = "server=(local);database='mytable';uid='sa';pwd='sa'"; //创建连接字符串
SqlConnection con = new SqlConnection(str); //创建连接对象
con.Open(); //打开连接
string strsql = "select top 5 * from mynews order by id desc";
SqlCommand cmd = new SqlCommand(strsql, con); //使用 ExecuteNonQuery
Label1.Text="该操作影响了"+cmd.ExecuteNonQuery()+"行"; //执行 SQL 语句并返回行
```

上述代码使用了 **SELECT** 语句，并执行语句，返回受影响的行数。运行后，发现返回的结果为-1，说明，当使用 **SELECT** 语句时，并没有对任何行有任何影响。**ExecuteNonQuery()**通常情况下为数据库事务处理的首选，当需要执行插入、删除、更新等操作时，首选 **ExecuteNonQuery()**。

对于更新、插入和删除的 **SQL** 句，**ExecuteNonQuery()**方法的返回值为该命令所影响的行数。对于“**CREATE TABLE**”和“**DROP TABLE**”语句，返回值为 0，而对于所有其他类型的语句，返回值为-1。**ExecuteNonQuery()**操作数据时，可以不使用 **DataSet** 直接更改数据库中的数据，示例代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    string str = "server=(local);database='mytable';uid='sa';pwd='sa'"; //创建连接字符串
    SqlConnection con = new SqlConnection(str); //创建连接对象
    con.Open(); //打开连接
    string strsql = "delete from mynews where id>4"; //编写执行删除的 SQL 语句
    SqlCommand cmd = new SqlCommand(strsql, con); //创建 Command 对象
    Label1.Text = "该操作影响了" + cmd.ExecuteNonQuery() + "行"; //返回影响行数
}
```

运行上述代码后，会执行删除 **id** 号大于 4 的数据事务，当执行删除并删除完毕后，则 **ExecuteNonQuery()**方法返回受影响的行数，如图 9-2 所示。





图 9-2 ExecuteNonQuery()方法

**ExecuteNonQuery()**操作主要进行数据库操作，包括更新、插入和删除等操作，并返回相应的行数。在进行数据库事务处理时或不需要 **DataSet** 为数据库进行更新时，**ExecuteNonQuery()**方法是数据操作的首选。因为 **ExecuteNonQuery()**支持多种数据库语句的执行。

注意：有些项目中，通过判断 **ExecuteNonQuery()**的返回值来判断 SQL 语句是否执行成功，这样是有失偏颇的，因为当使用创建表的语句时，就算执行成功也会返回-1。

### 9.1.3 使用 ExecuteScalar()操作数据库

**ExecuteScalar()**方法也用来执行 SQL 语句，但是 **ExecuteScalar()**执行 SQL 语句后的返回值与 **ExecuteNonQuery()**并不相同，**ExecuteScalar()**方法的返回值的数据类型是 **Object** 类型。如果执行的 SQL 语句是一个查询语句（**SELECT**），则返回结果是查询后的第一行的第一列，如果执行的 SQL 语句不是一个查询语句，则会返回一个未实例化的对象，必须通过类型转换来显示，示例代码如下所示。

```
string str = "server='(local)';database='mytable';uid='sa';pwd='sa'"; //创建连接字符串
SqlConnection con = new SqlConnection(str); //创建连接对象
con.Open(); //打开连接
string strsql = "select * from mynews order by id desc";
SqlCommand cmd = new SqlCommand(strsql, con);
Label1.Text = "查询出了 Id 为 " + cmd.ExecuteScalar(); //使用 ExecuteScalar 查询
```

通常情况下 **ExecuteNonQuery()**操作后返回的是一个值，而 **ExecuteScalar()**操作后则会返回一个对象，**ExecuteScalar()**经常使用于当需要返回单一值时的情况。例如当插入一条数据信息时，常常需要马上知道刚才插入的值，则可以使用 **ExecuteScalar()**方法。示例代码如下所示。

```
string str = "server='(local)';database='mytable';uid='sa';pwd='sa'"; //创建连接字符串
SqlConnection con = new SqlConnection(str); //创建连接对象
con.Open(); //打开连接
string strsql = "insert into mynews values ('刚刚插入的 id 是多少?')
SELECT @@IDENTITY as 'bh'"; //插入语句
SqlCommand cmd = new SqlCommand(strsql, con); //执行语句
Label1.Text = "刚刚插入的行的 id 是 " + cmd.ExecuteScalar(); //返回赋值
```

上述代码使用了 **SELECT @@IDENTITY** 语法获取刚刚执行更新后的 **id** 值，然后通过使用 **ExecuteScalar()**方法来获取刚刚更新后第一行第一列的值。

### 9.1.4 使用 ExecuteXmlReader()操作数据库

**ExecuteXmlReader()**方法用于操作 XML 数据库，并返回一个 **XmlReader** 对象，若需要使用 **ExecuteXmlReader()**方法，则必须添加引用 **System.Xml**。**XmlReader**类似于 **DataReader**，都需要通过 **Command** 对象的 **ExecuteXmlReader()**方法来创建 **XmlReader** 的对象并初始化，示例代码如下所示。

```
XmlReader xdr = cmd.ExecuteXmlReader(); //创建 XmlReader 对象
```

**ExecuteXmlReader()**返回 **XmlReader** 对象，**XmlReader** 特性如下所示：

- ❑ **XmlReader** 是面向流的，它把 **XML** 文档看作是文本数据流。
- ❑ **XmlReader** 是一个抽象类。
- ❑ **XmlReader** 使用 **pull** 模式处理流。
- ❑ 三个派生类：**XmlTextReader**、**XmlNodeReader** 和 **XmlValidatingReader**

下面代码实现了获取当前节点中属性的个数。

```
string str = "server=(local);database='mytable';uid='sa';pwd='sa';"      //创建连接字符串
SqlConnection con = new SqlConnection(str);                          //创建连接对象
con.Open();                                                            //打开连接
string strsql = "select * from mynews order by id desc FOR XML AUTO, XMLDATA";
SqlCommand cmd = new SqlCommand(strsql, con);                        //创建 Command 对象
XmlReader xdr = cmd.ExecuteXmlReader();                              //创建 XmlReader 对象
Response.Write(xdr.AttributeCount);                                   //读取节点个数
```

上述代码使用了 **SQL** 语言中的 **FOR XML AUTO**、**XMLDATA** 关键字，当执行 **ExecuteXmlReader()** 方法时，会返回 **XmlReader** 对象，若不指定 **FOR XML AUTO**、**XMLDATA** 关键字，则系统会抛出异常。

## 9.2 ASP.NET 创建和插入记录

在数据库操作中，经常需要对数据库中的内容进行插入操作。例如当有一个用户发布了评论，或者一个用户要购买某个商品，都需要插入记录来保存用户的相应的信息，以便当用户再次登录网站或应用时，能够及时获取自己购买的信息。

### 9.2.1 SQL INSERT 数据插入语句

使用 **SQL INSERT** 语句能够实现数据库的插入，**SQL** 语句必须遵照一些规范，**SQL INSERT** 语句的一般语法形式如下所示：

```
INSERT [INTO]
{table_name}
{
  [(column_list)]
  {
    VALUES({DEFAULT|NULL|expression} [...n])
  }
}
```

上述代码规范了 **INSERT** 语句的编写规范，其中：

- ❑ **INSERT** 是 **SQL** 插入关键字。
- ❑ **[INTO]** 是表名称之前能够包含的可选关键字。
- ❑ **Table\_name** 是相关的表名称。
- ❑ **column\_list** 是列的集合，如果有多个列可用都好隔开。
- ❑ **VALUES** 是相应的列的值。

如果需要向表 **mytables** 插入数据，而 **mytables** 里包括自动增长的主键 **id** 和 **title** 两列，则 **INSERT** 语句可以编写为如下代码。

```
INSERT INTO mytables VALUES ('新的新闻标题')
```

上述代码向表 **mytables** 中插入了一条新记录，并将 **title** 赋值为“新的新闻标题”。值得注意的是，在这条语句中，并没有编写列的集合，是因为当不编写 **column\_list** 时，则默认为每一个列插入数值。

**注意：**自动增长的主键类型的字段，无需向数据中插入数值，因为 **SQL Server** 会自动为该列赋值。

如果需要当插入数据时，需要指定插入相应的列的值，则可以将 **SQL** 语句代码编写如下。

```
INSERT INTO mytables (title) VALUES ('新的新闻标题')
```

上述代码指定了列 **title**，并对应了相应的值。若在表中存在多个列，列的顺序和列相应的值的顺序必须匹配。例如有 3 列并分别为 **number1**，**string2**，**datetime3**，当需要向其中插入数据时，则可以编写以下 SQL 语句。

```
INSERT INTO examtable (number1,string2,datetime3) VALUES (1,'this is a string','2008/9/18')
```

上述代码编写了 **INSERT** 语句以便数据的插入，同样在插入语句中如果需要插入所有的列，可以简化 **INSERT** 语句以便快速进行数据插入，示例代码如下所示。

```
INSERT INTO examtable VALUES (1,'this is a string','2008/9/18')
```

值得注意的是，无论按照何种方法编写 SQL 语句，值和列都应该相互匹配。

## 9.2.2 使用 Command 对象更新记录

编写了 SQL 语句后，必须执行 SQL 语句，在 ADO.NET 中，执行 SQL 语句有很多方法，其中推荐使用 **Command** 命令的 **ExecuteNonQuery()**。执行 SQL 语句的命令的必要步骤如下所示。

- ☐ 打开数据连接。
- ☐ 创建一个新的 **Command** 对象。
- ☐ 定义一个 SQL 命令。
- ☐ 执行 SQL 命令。
- ☐ 关闭连接。

从上面的步骤可以发现执行 SQL 语句是非常容易的，首先必须要打开到数据库的连接，示例代码如下所示。

```
string str = "server=(local);database=mytable;uid='sa';pwd='sa'";
SqlConnection con = new SqlConnection(str);           //创建连接对象
con.Open();                                           //打开连接
```

其中，**str** 是数据连接字符串，用来初始化 **Connection** 对象，说明如何连接数据库，当数据库连接完毕后，可以使用 **Open** 方法打开数据连接。完成数据库连接后，需创建一个新的 **Command** 对象，示例代码如下所示。

```
SqlCommand cmd = new SqlCommand("insert into mynews value ('插入一条新数据')", con);
```

**Command** 对象的构造函数的参数有两个，一个是需要执行的 SQL 语句，另一个是数据库连接对象。创建 **Command** 对象后，就可以执行 SQL 命令，执行后完成并关闭数据连接，示例代码如下所示。

```
cmd.ExecuteNonQuery();                               //执行 SQL 命令
con.Close();                                          //关闭连接
```

上述代码使用了 **ExecuteNonQuery()** 方法执行了 **SELECT** 语句的操作，当执行完毕后就需要对现有的连接进行关闭，以释放系统资源。

## 9.2.3 使用 DataSet 数据集插入记录

使用 **INSERT** 语句能够完成数据插入，使用 **DataSet** 对象也可以完成数据插入。为了将数据库的数据填充到 **DataSet** 中，则必须先使用 **DataAdapter** 对象的方法实现填充，当数据填充完成后，开发人员可以将记录添加到 **DataSet** 对象中，然后使用 **Update** 方法将记录插入数据库中。使用 **DataSet** 更新记录的步骤如下所示：

- ☐ 创建一个 **Connection** 对象。
- ☐ 创建一个 **DataAdapter** 对象。
- ☐ 初始化适配器。
- ☐ 使用数据适配器的 **Fill** 方法执行 **SELECT** 命令，并填充 **DataSet**。
- ☐ 使用 **DataTable** 对象提供的 **NewRow** 方法创建新行。
- ☐ 将数据行的字段设置为插入的值。



- ❑ 使用 **DataRowAdd** 类的 **Add** 方法将数据行添加到数据表中。
- ❑ 把 **DataAdapter** 类的 **InsertCommand** 属性设置成需要插入记录的 **INSERT** 语句。
- ❑ 使用数据适配器提供的 **Update** 方法将新记录插入数据库。
- ❑ 使用 **DataSet** 类提供的 **AcceptChanges** 方法将数据库与内存中的数据保持一致。

当使用 **DataSet** 插入记录前，需要创建 **Connection** 对象以保证数据库连接，示例代码如下所示。

```
string str = "server=(local);database=mytable;uid=sa;pwd=sa";           //创建连接字符串
SqlConnection con = new SqlConnection(str);                             //创建连接对象
con.Open();                                                             //打开连接
```

上述代码创建了一个数据库连接，并打开了数据库连接。完成数据连接后，就需要查询表中的数据并使用 **DataAdapter** 对象初始化适配器，示例代码如下所示。

```
string strsql = "select * from mynews";                                //编写 SQL 语句
SqlDataAdapter da = new SqlDataAdapter(strsql, con);                    //创建适配器
```

**DataAdapter** 对象默认构造函数包括两个参数，其中一个参数是需要执行的 **SQL** 语句，另一个是 **Connection** 对象。在初始化适配器后，需要对适配器的相应的属性做设置，使用 **SqlCommandBuilder** 对象可以让系统构造 **InsertCommand** 属性，示例代码如下所示。

```
SqlCommandBuilder build = new SqlCommandBuilder(da);                  //构造 SQL 语句
```

使用适配器的 **Fill** 方法能够填充 **DataSet** 数据集，示例代码如下所示。

```
DataSet ds = new DataSet();                                           //创建数据集
da.Fill(ds, "datatable");                                             //填充数据集
DataTable tb = ds.Tables["datatable"];                                //创建表
tb.PrimaryKey = new DataColumn[] { tb.Columns["id"] };              //创建表的主键
```

上述代码创建了一个 **DataSet** 数据集对象，被填充数据后，数据集中表的名称被命名为 **datatable**，该命名与数据库中的表的名称并不冲突。填充了 **DataSet** 数据对象后，需要使用 **DataRow** 对象为 **DataSet** 添加数据，示例代码如下所示。

```
DataRow row = ds.Tables["datatable"].NewRow();                        //创建 DataRow
row["title"] = "使用 DataSet 插入新行";                               //赋值新列
row["id"] = "15";
```

上述代码使用了 **NewRow** 方法创建新行返回 **DataRow** 对象，当 **DataRow** 对象中的相应的元素被赋值后，则需要使用 **Rows.Add** 方法增加新行，因为只对 **DataRow** 对象赋值，并不能自动的在数据库中增加新行。示例代码如下所示。

```
ds.Tables["datatable"].Rows.Add(row);                                 //添加新行
```

上述代码将数据更新到 **DataSet** 数据集中，为了保持数据集中的数据和数据库的数据的一致性，需使用 **Update** 方法，示例代码如下所示。

```
da.Update(ds, "datatable");                                           //更新数据
```

当执行了 **Update** 方法后，数据库中的数据就会同步 **DataSet** 数据集中的数据进行数据更新。

## 9.3 ASP.NET 更新数据库

在应用程序的开发中，常常会需要对数据库中现有的内容进行更新操作。**ADO.NET** 提供了若干不同的更新数据库中记录的方法，如果需要更新数据库中的某列的值或者某几列的值，则需要使用 **SQL UPDATE** 命令进行数据库更新。

### 9.3.1 SQL UPDATE 数据更新语句

使用 **SQL UPDATE** 语句能够实现数据库中数据的更新，**SQL UPDATE** 语句的一般语法格式如下所示。

```
UPDATE
{table_name}
{
```



```

SET column1_name=expression1,
  column2_name=expression2,
  .
  .
  columnN_name=expressionN
{WHERE condition1 AND|OR condition2}
}

```

上述代码规范了 **UPDATE** 语句的编写规范，其中：

- ❑ **UPDATE** 是 SQL 更新关键字。
- ❑ **table\_name** 是需要更新的表的名称。
- ❑ **columnN\_name** 是需要更新的列的名称。
- ❑ **expression** 是列相应的值。
- ❑ **WHERE** 是 SQL 语句关键字。
- ❑ **condition** 是条件。

如果需要更新表 **mytable** 中的某行的数据，则可以编写 **SQL UPDATE** 语句进行更新，示例代码如下所示。

```
UPDATE mytable SET title='修改后的数据' where id=3
```

上述代码更新了 **id** 为 **3** 的数据中的 **title**，并将 **title** 字段的值修改为“修改后的数据”。

### 9.3.2 使用 Command 对象更新记录

如需要执行 **UPDATE** 语句时，同样可以使用 **Command** 对象执行语句。**Command** 对象基本上能够执行所有需要进行数据更新的 **SQL** 语句。使用 **Command** 对象进行数据库操作的步骤基本如下所示。

- ❑ 创建数据库连接。
- ❑ 创建一个 **Command** 对象，并指定一个 **SQL UPDATE**（或存储过程）。
- ❑ 使用 **Command** 对象的 **ExecuteNonQuery()** 方法执行 **UPDATE**（或存储过程）。
- ❑ 关闭数据库连接。

当需要执行 **UPDATE** 语句时，首先必须要打开到数据库的连接，打开连接后，使用 **Command** 对象执行 **SQL** 语句，示例代码如下所示。

```

string str = "server=(local);database='mytable';uid='sa';pwd='sa'";
SqlConnection con = new SqlConnection(str);           //创建连接对象
con.Open();                                           //打开连接

```

其中，**str** 同样是数据连接字符串，用来初始化 **Connection** 对象，说明如何连接数据库，当数据库连接完毕后，可以使用 **Open** 方法打开数据连接。完成数据库连接后，需创建一个新的 **Command** 对象进行数据更新，示例代码如下所示。

```

SqlCommand cmd = new SqlCommand("UPDATE mynews SET title='修改后的数据'
where id=3", con);                                   //创建 Command 对象

```

**Command** 对象的构造函数的参数有两个，一个是需要执行的 **SQL** 语句，另一个是数据库连接对象。创建 **Command** 对象后，就可以执行 **SQL** 命令，执行后完成并关闭数据连接，示例代码如下所示。

```

cmd.ExecuteNonQuery();                             //执行 SQL 命令
con.Close();                                         //关闭连接

```

上述代码使用了 **ExecuteNonQuery()** 方法进行 **SQL UPDATE** 语句的执行，从而能够更新数据库中的相应数据。

### 9.3.3 使用 DataSet 数据集更新记录

**ADO.NET** 的 **DataSet** 对象提供了更好的编程实现数据库的更新功能。因为 **DataSet** 对象与数据库始终不是连接的，开发人员可以向脱离数据库的 **DataSet** 对象中增加列、删除列或更新列。当完成了修改后，则可

以通过将 **DataSet** 对象连接到 **DataAdapter** 对象来将记录传输给数据库。**DataSet** 更新记录的步骤如下所示。

- ☐ 创建一个 **Connection** 对象。
- ☐ 创建一个 **DataAdapter** 对象。
- ☐ 初始化适配器。
- ☐ 使用数据适配器的 **Fill** 方法执行 **SELECT** 命令，并填充 **DataSet**。
- ☐ 执行 **SqlCommandBuilder** 方法生成 **UpdateCommand** 方法。
- ☐ 创建 **DataTable** 对象并指定相应的 **DataSet** 中的表。
- ☐ 创建 **DataRow** 对象并查找需要修改的相应行。
- ☐ 更改 **DataRow** 对象中的列的值。
- ☐ 使用 **Update** 方法进行数据更新。

在更新记录前，首先需要查询出相应的数据，查询相应的数据后才能够填充 **DataSet**，示例代码如下所示。

```
string str = "server=(local);database='mytable';uid='sa';pwd='sa'";    //创建连接字符串
SqlConnection con = new SqlConnection(str);                          //创建连接对象
con.Open();                                                         //打开连接
string strsql = "select * from mynews";                             //执行查询
SqlDataAdapter da = new SqlDataAdapter(strsql, con);                 //使用 DataAdapter
DataSet ds = new DataSet();                                         //使用 DataSet
da.Fill(ds, "datatable");                                           //使用 Fill 方法填充 DataSet
```

上述代码将查询出来的数据集保存在名为 **datatable** 的 **DataSet** 记录集中，**DataSet** 记录集的表的名称可以按照开发人员的喜好来编写，从而区分内存中表的数据和真实的数据库的区别。当需要处理数据时，只需要处理相应名称的表即可，示例代码如下所示。

```
DataTable tb = ds.Tables["datatable"];
```

当需要执行更新时，可直接使用 **DataSet** 对象进行更新操作来修改其中的一行或多行记录，示例代码如下所示。

```
DataTable tb = ds.Tables["datatable"];
tb.PrimaryKey = new DataColumn[] { tb.Columns["id"] };
DataRow row = tb.Rows.Find(1);
row["title"] = "新标题";
```

当需要更新某个记录时，必须在更新之前查找到该行的记录。可以使用 **Rows.Find** 方法查找到相应的行，然后将数据集表中的该行的列值进行更新。使用 **DataAdapter** 的 **Update** 方法可以更新 **DataSet** 数据集，并保持数据集和数据库中数据的一致性，示例代码如下所示。

```
da.Update(ds, "datatable");
```

在执行以上代码，可能会抛出异常“当传递具有已修改行的 **DataRow** 集合时，更新要求有效的 **UpdateCommand**”。这是因为在更新时，并没有为 **DataAdapter** 对象配置 **UpdateCommand** 方法，可以通过 **SqlCommandBuilder** 对象配置 **UpdateCommand** 方法，示例代码如下所示。

```
SqlCommandBuilder build = new SqlCommandBuilder(da);
```

上述代码为 **DataAdapter** 对象自动配置了 **UpdateCommand**、**DeleteCommand** 等方法，当执行更新时，无需手动配置 **UpdateCommand** 方法。

## 9.4 ASP.NET 删除数据

当数据库中的数据过多，或需要对数据库进行数据优化时，则可能需要对数据库中的数据进行删除，例如用户的操作，长期不上线的用户资料，都可以删除。**ADO.NET** 提供多种数据库的删除方法，并且同样支持 **DataSet** 方法删除数据库。

### 9.4.1 SQL DELETE 数据删除语句

使用 **SQL DELETE** 语句能够实现数据库中数据的更新，**SQL DELETE** 语句的一般语法格式如下所示。

```
DELETE [FROM]
{table_name}
[WHERE condition1 AND|OR condition2]
```

上述代码规范了 **DELETE** 语句的编写规范，其中：

- ☐ **DELETE** 是 **SQL** 删除关键字。
- ☐ **FORM** 是一个可以选择的关键字。
- ☐ **table\_name** 是表的名称。
- ☐ **WHERE** 是一个 **SQL** 关键字。
- ☐ **conditionN** 是执行 **DELETE** 命令中需要达成的若干条件。

**SQL DELETE** 相对来说比较简单，当需要对某个表中的数据进行删除时，可以使用 **DELETE** 语句来执行删除操作，在编写 **DELETE** 语句时，需要指定表，并且指定相应的条件，示例代码如下所示。

```
DELETE FROM mynews WHERE ID=3
```

上述代码指定删除了 **mynews** 表中 **ID** 为 **3** 的行。如果不编写 **WHERE** 子句，则该表中所有的行都能够达成删除的条件，则会删除表中所有的行，示例代码如下所示。

```
DELETE FROM mynews
```

在编写删除语句时，可以通过编写相应的条件来提高执行的效率。

### 9.4.2 使用 Command 对象删除记录

当需要执行删除语句，可以使用 **Command** 对象来删除数据库中的记录。**Command** 对象的使用方法在前面的 **SQL** 语句介绍中已经讲的比较多了，在删除记录时，其使用方法基本相同。使用 **Command** 对象进行数据库操作的步骤基本如下所示。

- ☐ 创建数据库连接。
- ☐ 创建一个 **Command** 对象，并指定一个 **SQL DELETE**（或存储过程）。
- ☐ 使用 **Command** 对象的 **ExecuteNonQuery()** 方法执行 **DELETE**（或存储过程）。
- ☐ 关闭数据库连接。

当需要执行 **DELETE** 语句时，首先必须要打开到数据库的连接，打开连接后，使用 **Command** 对象执行 **SQL** 语句，示例代码如下所示。

```
string str = "server=(local);database='mytable';uid='sa';pwd='sa'";
SqlConnection con = new SqlConnection(str);
con.Open(); //打开连接
```

完成数据库连接后，需创建一个新的 **Command** 对象，示例代码如下所示。

```
SqlCommand cmd = new SqlCommand("Delete mynews where id=3", con);
```

**Command** 对象的构造函数的参数有两个，一个是需要执行的 **SQL** 语句，另一个是数据库连接对象。创建 **Command** 对象后，就可以执行 **SQL** 命令，执行后完成并关闭数据连接，示例代码如下所示。

```
cmd.ExecuteNonQuery(); //执行 SQL 命令
con.Close(); //关闭连接
```

### 9.4.3 使用 DataSet 数据集删除记录

使用 **DataSet** 删除记录和使用 **DataSet** 更新记录非常的相似，**DataSet** 删除记录的步骤如下所示。

- ☐ 创建一个 **Connection** 对象。
- ☐ 创建一个 **DataAdapter** 对象。
- ☐ 初始化适配器。

- ❑ 使用数据适配器的 **Fill** 方法执行 **SELECT** 命令，并填充 **DataSet**。
- ❑ 执行 **SqlCommandBuilder** 方法生成 **UpdateCommand** 方法。
- ❑ 创建 **DataTable** 对象并指定相应的 **DataSet** 中的表。
- ❑ 创建 **DataRow** 对象并查找需要修改的相应行。
- ❑ 使用 **Delete** 方法删除该行。
- ❑ 使用 **Update** 方法进行数据更新。

在删除记录前，首先需要创建连接，示例代码如下所示。

```
string str = "server=(local);database='mytable';uid='sa';pwd='sa'";
SqlConnection con = new SqlConnection(str);
con.Open();
string strsql = "select * from mynews";
```

上述代码创建了与数据库的连接，并编写 **SQL** 查询语句来填充 **DataSet**。填充 **DataSet** 对象需使用 **DataAdapter**，示例代码如下所示。

```
SqlDataAdapter da = new SqlDataAdapter(strsql, con);
SqlCommandBuilder build = new SqlCommandBuilder(da);
DataSet ds = new DataSet();
da.Fill(ds, "datatable");
```

编写完成后，需要创建 **DataTable** 对象对 **DataSet** 中相应的数据进行操作，其代码和更新记录基本相同，示例代码如下所示。

```
DataTable tb = ds.Tables["datatable"];
tb.PrimaryKey = new DataColumn[] { tb.Columns["id"] };
DataRow row = tb.Rows.Find(3);
```

在进行删除之前，同样需要找到相应的行，来指定删除语句所需要删除的行，示例代码如下所示。

```
row.Delete();
```

读者可以看到，**DataSet** 删除方法与更新方法不同的地方只操作语句的不同，在更新中使用的是 **Update()** 方法，而在删除中使用的是 **Delete()** 方法。

**注意：**当使用 **Delete** 方法删除记录行的时候，可以通过调用 **DataRow** 对象的 **RejectChanges** 方法取消对记录的删除，当使用该方法删除记录行时，该行的状态会恢复为 **Unchanged**。

在删除完毕后，同样需要保持 **DataSet** 中的数据和数据库中的数据的一致性，示例代码如下所示。

```
da.Update(ds, "datatable");
```

使用 **Update** 方法能够使 **DataSet** 中的数据和数据库中的数据保持一致性，在 **ASP** 中，这种方法也比较常见。

## 9.5 使用存储过程

存储过程在开发过程中经常被使用，因为存储过程能够将数据操作和程序操作在代码上分离，而且存储过程相对于 **SQL** 语句而言，具有更好的性能和安全性，使用存储过程能够提高应用程序的性能和安全性。

### 9.5.1 存储过程的优点

在数据库操作中，已经有了 **SQL** 语句，为何还需要存储过程。因为存储过程有 **SQL** 语句不能具备的特点和优点，以至于存储过程能在严格的数据库驱动的应用程序中起到重要的作用。存储和过程有点包括：

- ❑ 事务处理。
- ❑ 速度和性能。
- ❑ 过程控制。



- ☐ 安全性。
- ☐ 减少网络流量和通信。
- ☐ 模块化。

### 1. 事务处理

存储过程中，包括多个 **SQL** 语句，存储过程中的 **SQL** 语句属于事务处理的范畴。也就是说，存储过程类似于一个函数，当执行存储过程时，存储过程中的 **SQL** 语句要不都执行，要不都不执行。

### 2. 速度和性能

存储过程由数据库服务器编译和优化，优化包括使用存储过程在运行时所必须的特定数据库的结构信息，这样在执行过程中会节约很多时间。存储过程完全在数据库服务器上执行，避免了大量的 **SQL** 语句代码的传递，对于循环使用 **SQL** 语句而言，存储过程在速度和性能上都被优化。

### 3. 过程控制

在编写存储过程中，可以使用 **IF ELSE**、**FOR** 以及 **WHILE** 循环，这些语句并不能在 **SQL** 语句中编写，但是可以在存储过程中编写。当需要进行大量的和复杂的操作时，**SQL** 语句需要通过和编程语言一同编写才能实现，而且实现复杂。相比之下，存储过程可以对过程进行控制。

### 4. 安全性

存储过程也可以作为额外的安全层。开发人员或者用户，都只能对数据库中的存储过程进行使用，而无法直接对表进行数据操作，这样封装了数据操作，提高安全性。

### 5. 减少网络流量和通信

存储过程是在数据库服务器上运行的，在使用存储过程中，无需将大量的 **SQL** 语句代码传递给数据库服务器，而只需告诉数据库服务器执行哪个存储过程即可，而数据库服务器则会自行执行中间处理操作，而不会通过网络传递不必要的数据。

### 6. 模块化

正如代码编写规范和设计模式一样，通常情况下，开发团队或者公司需要严谨的代码编写风格和良好的协调能力，例如一个团队有人专门负责编码，有人专门负责数据库开发，那么可以让数据库开发人员负责数据库的开发，而编码的程序员只需要使用数据库开发人员设计的存储过程即可。在这种情况下，数据库操作和应用程序编码的操作被分开，在维护、管理中，也非常方便，如果数据库存储过程的代码出现问题，则只需要修改存储过程中的代码即可。

## 9.5.2 创建存储过程

存储过程可以通过 **SQL Server Management Studio** 创建，也可以使用 **.NET** 框架通过编程实现。**SQL Server Management Studio** 创建存储过程比较方便，右击【对象资源管理器】中的相应的数据库，在下拉菜单中选择【可编程性】选项并选择【存储过程】选项。单击右键，选择【新建存储过程】选项，系统会自动创建一个新的标签（**tab**）窗口，以提供输入存储过程语句，如图 9-3 所示。

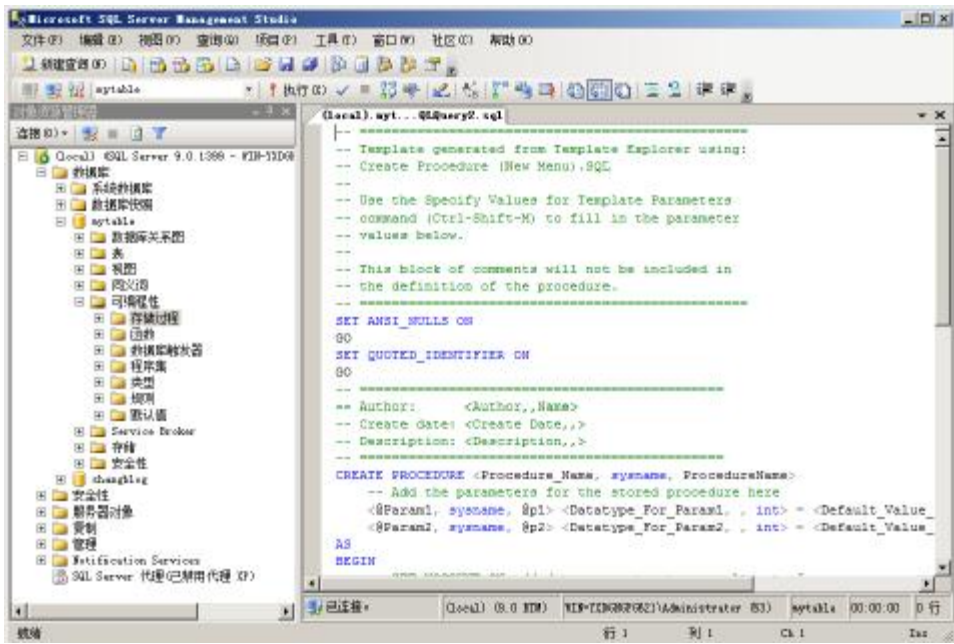


图 9-3 使用 Server Management Studio 创建存储过程

在 **tab** 窗口中输入存储过程，代码如下所示。

```
CREATE PROC myproc
(
    @id int,
    @title varchar(50) OUTPUT
)
AS
SET NOCOUNT ON
DECLARE @newscount int
SELECT @title=mynews.title,@newscount=COUNT(mynews.id)
FROM mynews
WHERE (id=@id)
GROUP BY mynews.title
RETURN @newscount
```

上述存储过程返回了数据库中新闻的标题内容。“@id”表示新闻的 **id**，@title 表示新闻的标题，此存储过程将返回 “@title” 的值，并且返回新闻的总数。在 **C#**中同样可以使用编程实现存储过程的创建，示例代码如下所示。

```
string str = "CREATE PROC myproc" +
"(" +
"@id int," +
"@title varchar(50) OUTPUT" +
")" +
"AS" +
"SET NOCOUNT ON" +
"DECLARE @newscount int" +
"SELECT @title=mynews.title,@newscount=COUNT(mynews.id)" +
"FROM mynews" +
"WHERE (id=@id)" +
"GROUP BY mynews.title" +
"RETURN @newscount";
SqlCommand cmd = new SqlCommand(str, con);
cmd.ExecuteNonQuery(); //使用 cmd 的 ExecuteNonQuery 方法创建存储过程
```

上述代码通过使用 **SqlCommand** 对象的 **ExecuteNonQuery()**方法在数据库中创建了一个存储过程，该存储过程用于返回了数据库中新闻的标题内容。

9.5.3 调用存储过程

创建存储过程之后，可以在.NET 应用程序中使用存储过程。存储过程可以看成是一个函数，可以对存储过程进行调用，传递参数，接受返回值。在调用存储过程前，首先要与数据库建立连接，示例代码如下所示。

```
string str = "server=(local);database='mytable';uid='sa';pwd='Sa'";
SqlConnection con = new SqlConnection(str);
con.Open(); //打开连接
```

建立与数据库连接后，需要使用 **Command** 对象使用存储过程，**Command** 对象接受的两个参数分别为 **SQL** 语句和 **Connection** 对象，在使用存储过程时，其中表示 **SQL** 语句的参数可以直接编写为存储过程名，代码如下所示。

```
SqlCommand cmd = new SqlCommand("getdetail", con); //使用存储过程
```

默认情况下，**Command** 对象的类型是 **SQL** 语句，必须将 **Command** 对象的 **CommandType** 属性设置为存储过程，系统才会调用存储过程，示例代码如下所示。

```
cmd.CommandType = CommandType.StoredProcedure; //设置 Command 对象的类型
```

设置执行类型后，需要为存储过程增加参数，示例代码如下所示。

```
SqlParameter spr; //表示执行一个存储过程
spr = cmd.Parameters.Add("@id", SqlDbType.Int); //增加参数 id
spr = cmd.Parameters.Add("@title", SqlDbType.NChar,50); //增加参数 title
spr.Direction = ParameterDirection.Output; //该参数是输出参数
spr = cmd.Parameters.Add("@count", SqlDbType.Int); //增加 count 参数
spr.Direction = ParameterDirection.ReturnValue; //该参数是返回值
cmd.Parameters["@id"].Value = 1; //为参数初始化
cmd.Parameters["@title"].Value = null; //为参数初始化
```

参数设置完毕后，执行 **ExecuteNonQuery** 方法能够执行存储过程，就相当于开始调用函数，示例代码如下所示。

```
cmd.ExecuteNonQuery(); //执行存储过程
```

当存储过程执行完毕后，能够获取参数和返回值，示例代码如下所示。

```
Label1.Text = cmd.Parameters["@count"].Value.ToString(); //获取返回值
```

使用 **SQL Server Management Studio** 同样能够执行存储过程，单击存储过程，单击右键，选择执行存储过程，系统会提示输入参数，如图 9-4 所示。输入相应的参数，单击确定，系统会执行存储过程并返回相应的值，如图 9-5 所示。

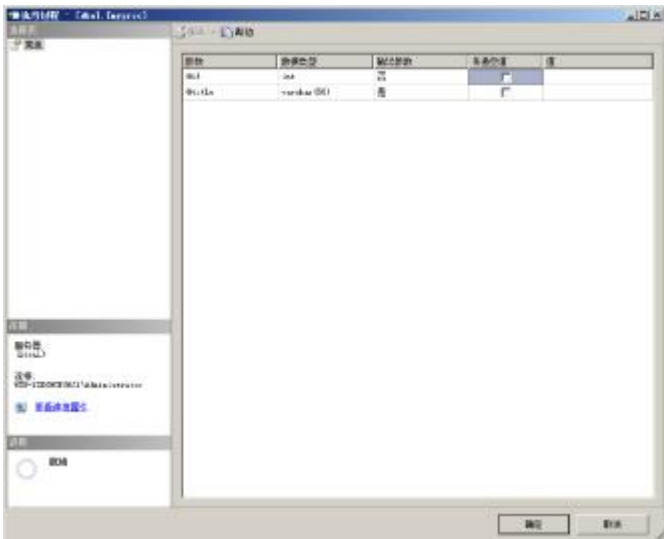


图 9-4 为存储过程传递参数



图 9-5 执行完成

使用 **SQL Server Management Studio** 能够快速的创建和使用存储过程，同样，能够通过编程的方法实现存储过程的创建、参数的传递以及执行。存储过程的优点就在于速度比较快，能够控制过程、减少网络通信和模块化，熟练的使用存储过程能够提高应用程序的性能和复用性。

9.6 ASP.NET 数据库操作实例

在了解了数据源控件和数据绑定控件的功能和使用方法，并且了解了 ADO.NET 的基本知识后，就可以使用控件和 ADO.NET 来操作数据库。ASP.NET 提供了强大的数据源控件和数据绑定控件，能够迅速的对数据库进行操作，同时，使用 ADO.NET 对数据进行操作，能够加深对 ADO.NET 的认识。

9.6.1 制作用户界面（UI）

使用数据控件和数据源控件显式数据，则需要为控件制作相应的用户界面，让数据控件对用户呈现的效果更好。首先，需要使用创建数据绑定控件 GridView 和数据源控件，并配置数据源控件，如图 9-6 所示。显然，对于用户而言，该数据源控件和数据绑定控件显然很不友好，这里就需要对数据绑定控件的界面进行修改。通过配置数据绑定控件的相应格式可以修改数据绑定控件的外观，如图 9-7 所示。



图 9-6 配置数据源控件和数据绑定控件



图 9-7 编辑数据绑定控件界面

开发人员能够自定义数据绑定控件的样式，并且修改某些列的顺序，这里使用了自动套用格式，并将数据绑定控件的 width 属性设置为 100%，这样编写宽度就能够适应浏览器的大小，从而随着浏览器的大小而改变。数据绑定控件配置完成后，值得注意的是，需要勾选 SQL 语句的高级选项，让数据绑定控件支持编辑、删除和选择，如图 9-8 所示。



图 9-8 SQL 高级选项

配置 SQL 高级选项后，数据源控件就会自动生成 INSERT、UPDATE、DELETE 语句，示例代码如下所示。

```
<asp:SqlDataSource ID="SqlDataSource1" runat="server"
    ConnectionString="<%= $ ConnectionStrings:mytableConnectionString %>"
    DeleteCommand="DELETE FROM [mynews] WHERE [ID] = @ID"
    InsertCommand="INSERT INTO [mynews] ([TITLE]) VALUES (@TITLE)"
    SelectCommand="SELECT * FROM [mynews]"
    UpdateCommand="UPDATE [mynews] SET [TITLE] = @TITLE WHERE [ID] = @ID">
    <DeleteParameters>
        <asp:Parameter Name="ID" Type="Int32" />
    </DeleteParameters>
```



```
<UpdateParameters>
  <asp:Parameter Name="TITLE" Type="String" />
  <asp:Parameter Name="ID" Type="Int32" />
</UpdateParameters>
<InsertParameters>
  <asp:Parameter Name="TITLE" Type="String" />
</InsertParameters>
</asp:SqlDataSource>
```

在完成用户界面的配置后，系统生成的 **HTML** 代码如下所示。

```
<asp:GridView ID="GridView1" runat="server" AllowPaging="True"
  AutoGenerateColumns="False" BackColor="White" BorderColor="#E7E7FF"
  BorderStyle="None" BorderWidth="1px" CellPadding="3" DataKeyNames="ID"
  DataSourceID="SqlDataSource1" GridLines="Horizontal" Width="100%">
  <FooterStyle BackColor="#B5C7DE" ForeColor="#4A3C8C" />
  <RowStyle BackColor="#E7E7FF" ForeColor="#4A3C8C" />
  <Columns>
    <asp:BoundField DataField="ID" HeaderText="ID" InsertVisible="False"
      ReadOnly="True" SortExpression="ID" />
    <asp:BoundField DataField="TITLE" HeaderText="TITLE" SortExpression="TITLE" />
  </Columns>
  <PagerStyle BackColor="#E7E7FF" ForeColor="#4A3C8C" HorizontalAlign="Right" />
  <SelectedRowStyle BackColor="#738A9C" Font-Bold="True" ForeColor="#F7F7F7" />
  <HeaderStyle BackColor="#4A3C8C" Font-Bold="True" ForeColor="#F7F7F7" />
  <AlternatingRowStyle BackColor="#F7F7F7" />
</asp:GridView>
```

开发人员可以编写以上 **HTML** 实现更多的效果，当确定用户界面编写完毕后，就可以为数据绑定控件选择操作了。

9.6.2 使用 GridView 显示、删除、修改数据

配置完成用户界面，则需要选择 **GridView** 控件的属性并配置 **GridView** 任务，如图 9-9 和图 9-10 所示。



图 9-9 默认 GridView 任务      图 9-10 选择 GridView 任务

**GridView** 控件支持分页、排序、编辑、删除和选定内容等操作。在 **GridView** 控件中，首先必须勾选【分页】复选框，然后再配置 **PageSize** 属性才能够让 **GridView** 控件支持分页功能。在 **GridView** 控件属性中如果勾选了【分页】复选框而不配置 **PageSize** 属性，则默认按 10 条数据分页。勾选了以启用分页、启用编辑、启用删除和启用选定内容后，**GridView** 控件的界面如图 9-11 所示。

因为在数据源控件配置的过程中，已经配置了支持编辑、删除和选择，所以在数据绑定控件中可以选择启用编辑，启用删除和选定内容等操作，并且系统默认支持更新、插入、删除等操作，运行后如图 9-12 所示。

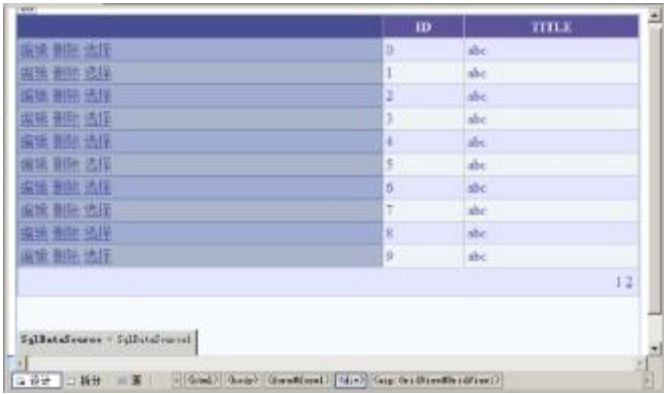


图 9-11 编辑数据绑定控件界面



图 9-12 GridView 控件显式

GridView 控件支持编辑、删除和选择，当单击编辑时，能够对选择的行进行数据编辑，如图 9-13 所示。编辑完成后，单击更新按钮则可以执行更新操作，而无需手动编写 UPDATE 操作，如图 9-14 所示。



图 9-13 编辑数据



图 9-14 执行更新操作

当单击删除时，则会执行 DELETE 命令，而无需手动编写 DELETE 命令。GridView 控件支持分页、排序、编辑、删除和选定内容，开发人员无需手动编写更新、删除、编辑、也无需手动编写分页，对 GridView 控件进行缓存设置能够提高应用程序性能，在对数据库的操作，编辑及更新中，GridView 控件能够方便开发人员，简化代码。

9.6.3 使用 DataList 显示数据

DataList 控件需要编辑 HTML 模板来显式数据，虽然在开发上，DataList 控件比 GridView 更加复杂，但是 DataList 控件能够实现更多效果。相比之下，DataList 控件比 GridView 控件更加灵活，能够进行复杂的事件编写和样式控制。选择【自动套用格式】复选框并将 DataList 控件的宽度设置为 100%，编辑基本的用户界面，如图 9-15 所示。

通过编辑 ItemTemplate 能够实现自定义模板，而无需像 GridView 一样，以表格形式呈现，编辑后运行如图 9-16 所示。



图 9-15 DataList 控件显式数据



图 9-16 编写 ItemTemplate 模板

DataList 控件执行数据操作基本上同 GridView 一样，DataList 控件与 GridView 相比只下，有着更灵活的模板方案，能够实现更多的显示效果。

## 9.6.4 DataList 分页实现

**DataList** 控件本身并不带分页实现，如果需要 **DataList** 能够实现分页效果，则需要通过代码实现 **DataList** 控件的分页。**DataList** 控件分页需要增加若干标签（**Label**）控件来显式“上一页”，“下一页”等分页所需要的连接，示例代码如下所示。

```
<asp:Label ID="Label4" runat="server" Text="Label"></asp:Label>
<asp:Label ID="Label3" runat="server" Text="Label"></asp:Label>
<asp:Label ID="Label2" runat="server" Text="Label"></asp:Label>
```

上述代码创建了三个 **Label** 控件，这三个控件并无需初始化，这三个控件通过编程实现上一页，下一页的分页形式。如果需要执行分页，则需要编写 **cs** 页面代码，**cs** 页面代码如下所示。

```
PagedDataSource objPds = new PagedDataSource();
objPds.DataSource = this.SqlDataSource1.Select(new DataSourceSelectArguments());
objPds.AllowPaging = true; //设置是否允许分页
objPds.PageSize = 3; //设置分页条目数
int CurPage; //设置当前页码
Label2.Visible = false; //隐藏标签
Label4.Visible = false;
```

上述代码初始化 **PagedDataSource** 对象，并将分页控件默认初始化属性 **Visible** 为 **false**。其中 **PagedDataSource** 是封装分页相关属性的类。

```
if (Request.QueryString["Page"] != null) //如果传递的页面不为空
{
    CurPage = Convert.ToInt32(Request.QueryString["Page"]); //获取传递的参数
}
else
{
    CurPage = 1; //页面的值为 1
}
objPds.CurrentPageIndex = CurPage - 1; //设置索引
Label2.Visible = true; //显式标签
Label4.Visible = true;
Label3.Text = "<a href='\"datalist.aspx\"'>首页</a>"; //编写分页
Label2.Text = "<a href='\"datalist.aspx?page=" + Convert.ToString(CurPage + 1) + "\">下一页</a>";
Label4.Text = "<a href='\"datalist.aspx?page=" + Convert.ToString(CurPage - 1) + "\">上一页</a>";
```

上述代码通过传递的 **Page** 的值进行分页操作，如果传递的 **Page** 的值为不为空，则从数据源控件中读取相应的数据，并显示到数据绑定控件中。

```
if (CurPage == 1) //如果只有一个页面
{
    Label4.Visible = false; //隐藏标签
}
if (objPds.IsLastPage)
{
    Label2.Visible = false;
}
DataList1.DataSourceID = ""; //重新绑定数据
DataList1.DataSource = objPds; //编写 DataList 的数据源
DataList1.DataBind(); //绑定数据源
```

上述代码通过 **PagedDataSource** 对象实现了分页效果，并且将分页条目数设置为 **3**，当数据超过 **3** 条时，则会实现分页。运行后如图 9-17 和图 9-18 所示。





图 9-17 实现下一页效果



图 9-18 实现上一页效果

**DataList** 控件虽然不支持分页，但是能够通过编程实现 **DataList** 控件的分页效果。**DataList** 控件在模板编辑和代码开发上虽然没有 **GridView** 方便，但是却提高了灵活性，能够自定义分页和数据显示。

9.6.5 使用 SQLHelper 操作数据库

使用控件，能够方便开发人员的开发和使用，但是很多情况下，不能使用控件来实现，所以很多情况都需要使用 **ADO.NET** 操作数据库中的数据，**SQLHelper** 是将 **ADO.NET** 中对数据操作的类和对象进行的封装的一个类库，使用 **SQLHelper** 能够提高数据库操作的效率。

1. 创建 SQLHelper

**SQLHelper** 类经常在数据库开发中使用，不仅封装了数据库操作，也提高了数据库操作的安全性，**SQLHelper** 在微软的开发中和 **DEMO** 中经常被使用，**SQLHelper** 通常用于多层设计，如果需要使用 **SQLHelper** 类，可以到微软官方下载最新的 **SQLHelper** 类，也可以自行编写 **SQLHelper** 类。如果自行创建 **SQLHelper** 类，则在解决方案管理器中新建一个类库，如图 9-19 所示。

创建类库后，删除自动生成的 **Class1** 类，并创建一个新类，类名为 **SQLHelper**，如图 9-20 所示。



图 9-19 添加类库



图 9-20 创建 SQLHelper 类

如果使用下载的 **SQLHelper** 类，则可以单击解决方案管理器，单击右键，选择添加现有项，然后选择现有项目添加即可。在 **SQLHelper** 类下对数据操作进行封装，开发人员能够使用自己封装的类进行数据操作，示例代码如下所示。

```
#region //数据库连接串
private static readonly string database = "数据库";           //配置数据库信息
private static readonly string uid = "用户名";               //配置用户名信息
private static readonly string pwd = "密码";                 //配置密码信息
private static readonly string server = "服务器";            //配置服务器信息
private static readonly string condb = "server='"+ server +";database='"+ database + ";uid=";
```



```
"" + uid + "";pwd="" + pwd + "";Max Pool Size=100000;Min Pool Size=0;
Connection Lifetime=0;packet size=32767;Connection Reset=false; async=true";//设置连接字符串
#endregion
#region//DataAdapter 方法 返回 DataSet 数据集
/// <summary>
/// DataAdapter 方法 返回 DataSet 数据集
/// </summary>
/// <param name="sqlCmd">SQL 语句</param>
/// <param name="command">操作参数 枚举类型</param>
/// <returns></returns>
public static DataSet DataAdapter(string sqlCmd, SDACmd command,           //实现适配器
string tabName, params SqlParameter[] paraList)
{
    SqlConnection con = new SqlConnection(condb);                        //创建连接对象
    SqlCommand cmd = new SqlCommand();                                  //创建 Command 对象
    cmd.Connection = con;                                              //使用连接对象
    cmd.CommandText = sqlCmd;                                          //配置连接字符串
    if (paraList != null)
    {
        cmd.CommandType = CommandType.Text;                          //配置 Command 类型
        foreach (SqlParameter para in paraList)                       //遍历参数
        { cmd.Parameters.Add(para); }                                  //添加参数
    }
    SqlDataAdapter sda = new SqlDataAdapter();                          //创建适配器
    switch (command)                                                    //查找条件
    {
        case SDACmd.select:                                           //如果为 select 执行
            sda.SelectCommand = cmd;
            break;
        case SDACmd.insert:                                           //如果为 insert 执行
            sda.InsertCommand = cmd;
            break;
        case SDACmd.update:                                           //如果为 update 执行
            sda.UpdateCommand = cmd;
            break;
        case SDACmd.delete:                                           //如果为 delete 执行
            sda.DeleteCommand = cmd;
            break;
    }
    DataSet ds = new DataSet();                                         //创建数据集
    sda.Fill(ds, tabName);                                             //填充数据集
    return ds;                                                         //返回数据集
}
```

在上述代码中，还需要通过一个枚举类型进行 **switch** 操作，枚举类型用于判断执行的操作，示例代码如下所示。

```
public enum SDACmd { select, delete, update, insert }           //定义枚举类型
```

定义的枚举类型用于在程序中进行筛选操作，用于指定 **SQL** 语句执行的操作。在 **SQLHelper** 类中，还需要封装 **DataReader** 方法进行 **DataReader** 的封装和实现，开发人员能够使用 **SQLHelper** 类中的 **DataReader** 方法进行数据库的读取，示例代码如下所示。

```
public static SqlDataReader ExecReader(string sqlcmd, params SqlParameter[] paraList)
{
    SqlConnection con = new SqlConnection(condb);                    //创建连接对象
    SqlCommand cmd = new SqlCommand();                                //创建 Command 对象
    cmd.Connection = con;                                            //使用连接
    cmd.CommandText = sqlcmd;                                         //配置 SQL 语句
    if (paraList != null)
```

```

    {
        cmd.CommandType = CommandType.Text;                //配置 Command 类型
        foreach (SqlParameter para in paraList)
        { cmd.Parameters.Add(para); }                        //添加参数
    }
    con.Open();                                              //打开连接
    SqlDataReader sdr = cmd.ExecuteReader(CommandBehavior.CloseConnection);
    return sdr;
}

```

上述代码实现了 **DataReader** 对象，使用 **DataReader** 能够填充 **SqlDataReader** 对象并进行数据的循环输出。在 **ADO.NET** 中，通常需要执行 **SQL** 语句进行数据库的操作，在 **SQLHelper** 类中，同样需要封装执行 **SQL** 语句的操作以便能够快速执行数据操作。

```

public static void ExecNonQuery(string sqlcmd, params SqlParameter[] paraList)
{
    using (SqlConnection con = new SqlConnection(condb))    //创建连接对象
    {
        SqlCommand cmd = new SqlCommand();                //创建 Command 对象
        cmd.Connection = con;                             //使用连接
        cmd.CommandText = sqlcmd;                          //配置执行类型
        if (paraList != null)
        {
            cmd.CommandType = CommandType.Text;          //配置执行类型
            foreach (SqlParameter para in paraList)
            { cmd.Parameters.Add(para); }                  //添加参数
        }
        con.Open();                                        //打开数据连接
        cmd.ExecuteNonQuery();                             //执行 SQL 语句
    }
}

```

上述代码编写了 **SQLHelper** 类操作数据库的函数，通过执行函数并传递参数，即可实现数据库的插入、更新和删除。

## 2. 使用 SQLHelper

创建完成 **SQLHelper** 类后，需要为应用程序配置 **SQLHelper** 的基本属性，代码如下所示。

```

private static readonly string database = "mytable";      //配置数据库
private static readonly string uid = "sa";               //配置用户名
private static readonly string pwd ="sa";               //配置用户会密码
private static readonly string server = "local";         //配置服务器的值

```

上述代码为 **SQLHelper** 类配置了属性，当使用 **SQLHelper** 类时，系统会自动连接数据库，在完成使用后，系统会自动关闭数据库。如果需要在当前项目使用 **SQLHelper** 类，则需要添加引用来使用 **SQLHelper** 类，右击现有项目，在下拉菜单中选择【添加】选项，在【添加】选项中选择【现有项】选项，在弹出窗口中选择【项目】标签栏，就可以添加相同项目的类库，如图 9-21 所示。



图 9-21 添加引用

引用添加完毕，在使用 **SQLHelper** 页面的 **CS** 页面中，需要添加命名空间，命名空间的名称和创建类库的名称相同，如果需要更改名称，可以通过修改类库的属性来修改。示例代码如下所示。

```
using MYSQL;
```

引用完毕后，就可以执行 **SQL** 语句，使用 **SQLHelper** 执行 **SQL** 语句非常方便，下面代码演示了如何执行插入、删除操作。

```
string strsql = "insert into mynews values ('SQLHelper 插入标题');           //编写 SQL 语句
SQLHelper.ExecNonQuery(strsql);                                           //执行 SQL 语句
```

上述代码运行后，则会执行插入操作，相比于 **ADO.NET**，封装后的代码更加简便易懂，删除操作代码如下所示。

```
string strsql2 = "delete form mynews where id=3";                         //编写 SQL 语句
SQLHelper.ExecNonQuery(strsql2);                                           //执行 SQL 语句
```

当需要执行 **SELECT** 语句时，可以通过 **SQLHelper.DataAdapter** 获取数据，示例代码如下所示。

```
string strsql = "select * from mynews where id=3";                       //编写 SQL 语句
DataSet ds = SQLHelper.DataAdapter(strsql, SQLHelper.SDACmd.select, "mydatatable");
```

上述代码通过 **SQLHelper.DataAdapter** 获取数据，并创建了一个 **mydatatable** 虚拟表，填充 **DataSet** 对象。当需要获取 **DataSet** 对象中的数据时，和普通的 **DataSet** 对象一样。**SQLHelper** 封装了 **ADO.NET** 中的许多方法，为开发人员提高了效率，同时也增加了安全性和模块化的特性。

**注意：**上述代码中的 **SQLHelper** 类能够执行的是 **SQL** 语句，如果需要执行存储过程，则需要更改 **SQLHelper** 类中的相应的 **CommandType** 属性的值。

## 9.7 小结

本章介绍了 **ADO.NET** 中操作数据库和执行数据库的一些方法，还介绍了如何编写和执行 **SQL** 语句，包括 **SQL INSERT**、**SQL UPDATE**、**SQL DELETE** 等数据操作语句，另外，本章还介绍了如何通过 **DataSet** 数据集实现插入、更新、删除等操作来深入了解 **ADO.NET**。本章通过演示使用控件更新和操作数据库，加强了控件操作数据库的示例，本章还包括：

- ❑ 使用 **ADO.NET** 操作数据库，介绍了 **ADO.NET** 操作数据库的方法。
- ❑ **ASP.NET** 创建和插入记录，介绍了 **SQL INSERT** 和数据操作。
- ❑ **ASP.NET** 更新数据库，介绍了 **SQL UPDATE** 和数据操作。
- ❑ **ASP.NET** 删除数据，介绍了 **SQL DELETE** 和数据操作。
- ❑ 使用存储过程，介绍了如何使用存储过程。
- ❑ 数据库操作实例，介绍了企业应用中常用的 **SQLHelper**，以及用户操作数据库的配置。

本章还介绍了如何创建和使用企业应用中常用的 **SQLHelper** 类、**SQLHelper** 类能够简化数据使用，提高开发效率。下一章中将会讲解如何连接其他数据库。

## 第 10 章 访问其他数据源

在 ADO.NET 体系中，非常重要的组件就是 **.NET Data Provider**，它负责建立与数据库之间的连接并执行数据操作。ADO.NET 提供了多种 **.NET Data Provider**，负责连接不同的数据库。在前面的章节中，通常使用的是 **SQL Server .NET Data Provider**，使用其他的 **.NET Data Provider** 能够访问其他类型的数据库。

### 10.1 使用 ODBC .NET Data Provider

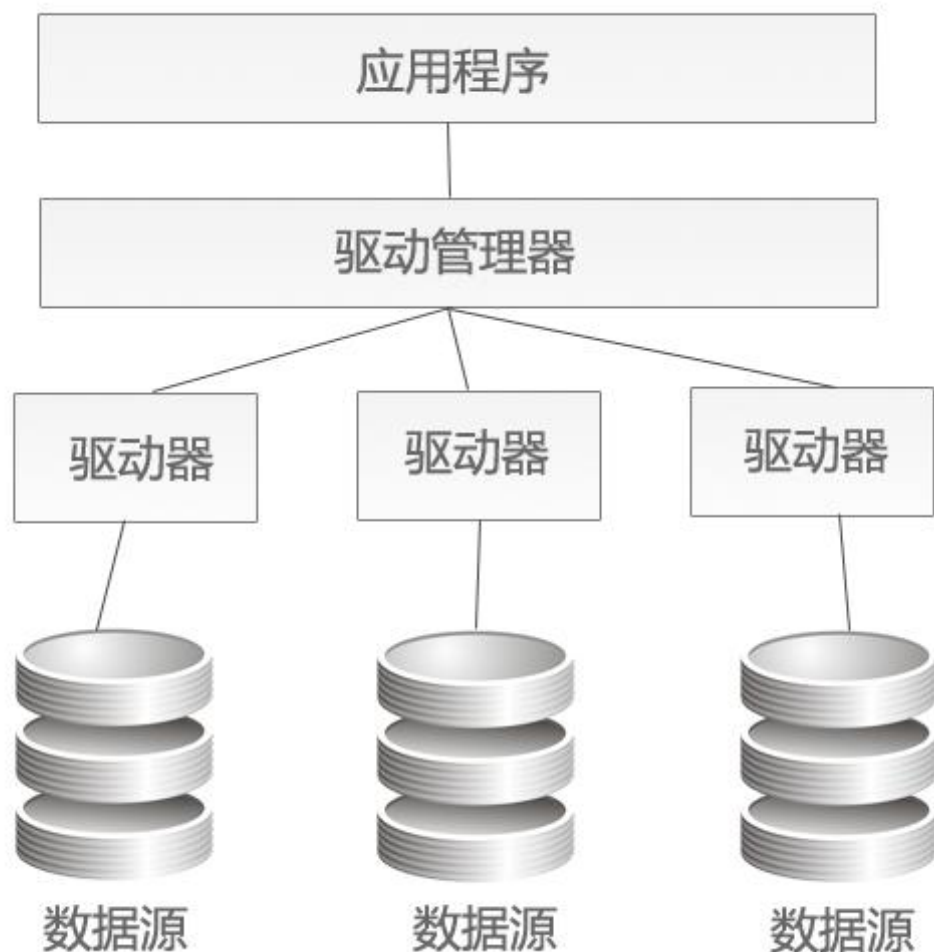
**ODBC (Open Database Connection)**，开放式数据互连)是访问数据库的一个统一的接口标准。在 C++ 开发中，经常使用 **ODBC** 来与数据库互连，**.NET** 同样提供了连接 **ODBC** 的方法。**ODBC** 可以让开发人员通过 **API** 来访问多种不同的数据库，包括 **SQL Server**、**Access**、**MySql** 等。

#### 10.1.1 ODBC .NET Data Provider 简介

**ODBC (Open Database Connection)**，开放式数据互连)是访问数据库的一个统一的接口标准，它允许开发人员使用 **ODBC API** (应用程序接口)来访问多种不同的数据源，并执行数据操作。

当使用应用程序时，应用程序首先通过使用 **ODBC API** 与驱动管理器进行通信。**ODBC API** 由一组 **ODBC** 函数调用组成，通过 **API** 调用 **ODBC** 函数提交 **SQL** 请求，然后驱动管理器通过分析 **ODBC** 函数并判断数据源的类型。驱动管理器会配置正确的驱动器，然后将 **ODBC** 函数调用传递给驱动器。最后，驱动器处理 **ODBC** 函数调用，把 **SQL** 请求发送给数据源，数据源执行相应操作后，驱动器返回执行结果，管理器再把执行结果返回给应用程序，如图 10-1 所示。





使用命名空间 **System.Data.Odbc** 才能够使用 **ODBC .NET Data Provider** 来访问 **ODBC** 数据源，并且支持对原有的 **ODBC** 驱动程序的访问。通过 **ODBC** 能够连接和执行数据操作，其访问方式和 **SQL Server .NET Data Provider** 相似，都需要先与数据源建立连接并打开连接，然后创建 **Command** 对象执行相应操作，最后关闭数据连接。

通过 **ODBC** 驱动程序访问数据源与 **SQL Server .NET Data Provider** 相同，**ODBC .NET Data Provider** 同样包含 **Connection**、**Command**、**DataReader** 等类为开发人员提供数据的遍历和存取等操作，这些类和功能如下所示。

- ❑ **OdbcConnection**: 建立与 **ODBC** 数据源的连接。
- ❑ **OdbcCommand**: 执行一个 **SQL** 语句或存储过程。
- ❑ **OdbcDataReader**: 与 **Command** 对象一起使用，读取 **ODBC** 数据源。
- ❑ **OdbcDataAdapter**: 创建适配器，用来填充 **DataSet**。
- ❑ **OdbcCommandBuilder**: 用来自动生成插入、更新、删除等操作的 **SQL** 语句。

上述对象在 **ADO.NET** 中经常遇到，在前面的章节中，**SQL Server .NET Data Provider** 同样包括这些对象，使用 **ODBC** 操作数据源的操作方法与 **SQL Server .NET Data Provider** 基本相同，使得开发人员无需额外的学习即可轻松使用。

## 10.1.2 建立连接

**ODBC .NET Data Provider** 连接数据库有两种方法，一种是通过 **DSN** 连接数据库，第二种就是使用 **OdbcConnection** 对象建立与数据库的连接。

### 1. 使用 **DSN** 的连接字符串进行连接

使用 **DSN** (**Data Source Name**, 数据源名) 连接数据库，必须首选创建 **ODBC** 数据源，当创建一个 **ODBC** 数据源时，需要在管理工具中配置。在开始菜单中找到并打开【控制面板】，然后在【控制面板】中选择【管理工具】选项，在【管理工具】选项中选择【数据源 (ODBC)】选项，选择后还需要选择【系统 DSN】标签用于系统 **DSN** 的配置，如图 10-2 所示。

单击【添加】按钮，在弹出的对话框中选择合适的驱动程序，由于这里需要使用 DSN 连接 ACCESS 数据库，就需要选择 ACCESS 数据库的相应的驱动，这里选择【Microsoft Access Driver(\*mdb)】选项，如图 10-3 所示。



图 10-2 数据源管理器



图 10-3 创建新数据源

单击【选择】按钮可以为需要使用数据库的应用程序选择相应的数据库，在选择完成相应的数据库后就能够在应用程序中使用 DSN 连接该数据源，如图 10-4 所示。



图 10-4 选择数据源

选择好相应的驱动程序后，系统会弹出【ODBC Microsoft Access 安装】对话框并为驱动设置数据源名和说明，如图 10-5 所示。单击【确定】按钮就完成了数据源的配置，如图 10-6 所示。



图 10-5 命名数据源



图 10-6 数据源配置完毕

当配置完成数据源后，就可以编写.NET应用程序来访问数据源。打开 Visual Studio 2008，选择【ASP.NET Web 应用程序】选项，如图 10-7 所示。



图 10-7 创建 ASP.NET 应用程序

当创建完成数据源之后，就可以使用 **OdbcConnection** 对象连接应用程序和数据库，与连接字串一样，**OdbcConnection** 对象需要使用 **Open** 方法才能打开与数据库之间的连接。在使用 **OdbcConnection** 对象同样需要使用命名控件 **System.Data.Odbc**，示例代码如下所示。

```
using System.Data.Odbc;

引用了 System.Data.Odbc 命名空间后，就可以创建 Connection 对象进行数据连接，示例代码如下所示。

string str = @"DSN=guojing"; //使用 ODBC 数据源
OdbcConnection con = new OdbcConnection(str); //创建 OdbcConnection 对象
con.Open(); //打开数据库连接
```

上述代码使用了 **ODBC** 数据源，数据源的名称和刚才创建的名称相同，数据库连接字串直接使用“**DSN=数据源名称**”即可。打开了与数据库的连接后，即可对数据库进行操作，操作方法同样和普通的方法没有区别。如果希望执行查询语句并填充数据集，则需要创建 **DataAdapter** 对象和 **DataSet** 对象，示例代码如下所示。

```
string strsql = "select * from mytable";
OdbcDataAdapter da = new OdbcDataAdapter(strsql,con); //创建 DataAdapter 对象
DataSet ds = new DataSet(); //创建 DataSet
da.Fill(ds, "tablename"); //填充数据集
```

若需要执行插入、更新、删除等操作，可以使用 **Command** 对象执行相应的操作，示例代码如下所示。

```
OdbcCommand cmd = new OdbcCommand("insert into mytable values ('title')",con);
cmd.ExecuteNonQuery(); //执行 SQL 语句
```

当需要对其他的数据源执行操作时，在配置 **DSN** 时配置其他数据源和数据源驱动程序即可，如图 10-8 所示。

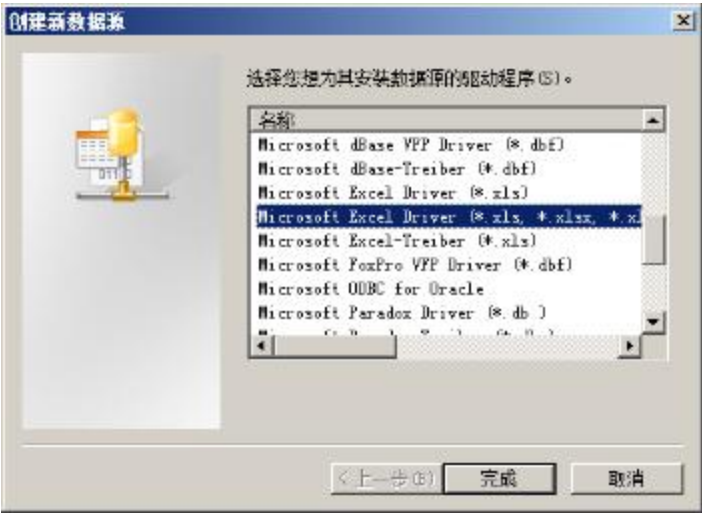


图 10-8 配置其他数据源

使用 **ODBC** 配置数据源有一些好处，就是能够为其他类型的数据库配置数据驱动而不用考虑驱动如何



进行手动方式连接。

## 2. 使用接字符串进行连接（ACCESS）

**ACCESS** 数据库是桌面级的数据库，是以一种文件形式保存的数据库。在使用 **ACCESS** 数据库时，很多情况下都不能依靠 **ODBC** 建立数据驱动来连接数据库，在这种情况下，需要使用连接字符串连接 **ACCESS** 数据库，示例代码如下所示。

```
string str = "provider=Microsoft.Jet.OLEDB.4.0 ;
Data Source=D:\ASP.NET 3.5\源代码\第 10 章\10-1\10-1\acc.mdb";           //配置数据库路径
```

上述代码使用了 **Micorsoft.Jet.OLEDB 4.0** 驱动进行 **ACCESS** 数据库的连接。在上述代码中，**ACCESS** 文件的地址为“**D:\ASP.NET 3.5\源代码\第 10 章\10-1\10-1**”。但是这样编写代码有若干坏处，最大的坏处就是暴露了物理路径，当非法用户访问或获取代码后，能容易的就能够获取数据库的信息并通过下载工具下载数据库，这样是非常不安全的。为了提高应用程序的安全性，开发人员可以使用 **Server.MapPath** 方法指定相对路径，示例代码如下所示。

```
string str = "provider=Microsoft.Jet.OLEDB.4.0 ;Data Source=" + Server.MapPath("acc.mdb") + "";
```

上述代码指定了数据库的文件，以及相对路径，当 **acc.mdb** 与文件夹路径相同时，系统会隐式的补完绝对路径，而不会轻易的暴露物理路径，如果 **acc.mdb** 在文件夹的上层路径，则只需要使用“**../**”来确定相对路径。当指定数据库文件地址后，可以使用 **Connection** 对象进行数据库连接操作并使用 **Open** 方法打开数据连接，示例代码如下所示。

```
OdbcConnection con = new OdbcConnection(str);           //配置连接对象
con.Open();           //打开连接
```

## 3. 使用接字符串进行连接（SQL Server）

**SQL Server** 数据库可以采用两种不同的连接方式，正如 **SQL Server Management Studio** 中注册连接一样，包括 **Windows** 安全认证和 **SQL Server** 验证两种验证方式，**SQL Server Management Studio** 中注册连接的方式如图 10-9 所示。



图 10-9 SQL Server 的两种连接方式

当使用 **Windows** 安全认证进行数据连接时，**SQL Server** 无需用户提供连接的用户名和密码即可连接，因为 **Windows** 安全认证是通过 **Windows** 登录时的账号来登录的。开发 **ASP.NET** 应用程序时，需要显式的声明这是一个安全的连接，示例代码如下所示。

```
@ " Driver={SQL Server}; Server=(local);Database=mytable;Trusted_Connection=Yes";
```

如果需要使用 **SQL Server** 验证方式连接数据库，就不能够使用 **Trusted\_Connection** 属性进行数据连接，而需要配置 **User=用户名;Password=密码**，示例代码如下所示。

```
@ "Driver={SQL Server}; Server=(local);Database=mytable;User ID=sa;Password= ";
```



## 10.2 使用 OLE DB.NET Data Provider

**OLE DB** 是访问数据库的另一个统一的接口标准，它建立在 **ODBC** 基础之上，通过 **OLE DB** 可以访问关系型数据库和非关系型数据库，**OLE DB** 不仅使应用程序和数据库之间的交互减少，还能够最大限度的提升数据库性能。

### 10.2.1 OLE DB.NET Data Provider 简介

**OLE DB** (**Object Link and Embedding Database**，对象连接与嵌套数据库) 是访问数据库的另一个统一的接口标准，它建立在 **ODBC** 基础之上，不仅能够提供传统的数据库访问，并且能够访问关系型数据库和非关系型数据库。

**OLE DB** 其本质就是一个封装数据库访问的一系列 **COM** 接口，使用 **COM** 接口不仅能够减少应用程序和数据库之间的通信和交互，也能够极大的提升数据库的性能，让数据库的访问和操作更加便捷。

**OLE DB.NET Data Provider** 是 **OLE DB** 数据源的托管数据提供者，**OLE DB.NET Data Provider** 能够在不更改 **COM** 组件的情况下，使用 **COM Interpol** 来使用 **OLE DB.NET Data Provider** 访问数据源，如果需要 **OLE DB.NET Data Provider** 访问数据源，则必须存在一个支持 **OLE DB.NET Data Provider** 的 **OLE DB Provider**。**OLE DB.NET Data Provider** 访问原理图如图 10-10 所示。

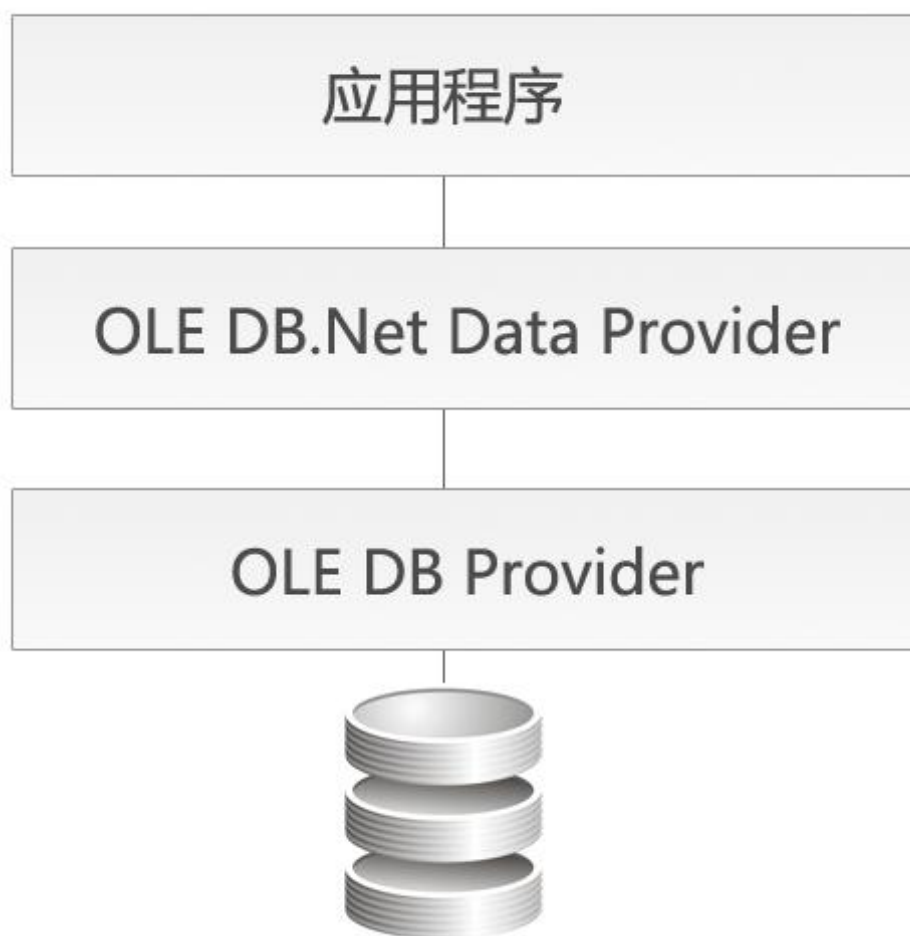


图 10-10 OLE DB.NET Data Provider 原理图

**OLE DB.NET Data Provider** 支持 **OLE DB Provider** 包括：

- ☐ **SQLOLEDB**：用来访问 **SQL Server** 数据库。
- ☐ **MSDAORA**：用来访问 **Oracle** 数据库。
- ☐ **Microsoft.Jet.OLEDB.4.0**：用于访问 **Access** 等使用 **Jet** 引擎的数据库。

使用 **OLE DB.NET Data Provider** 访问数据库同 **ODBC .NET Data Provider** 相同，具有 **Connection** 对象、**Command** 对象等用于数据库的连接和执行，以及数据存储等，常用对象如下所示。

- ❑ **OleDbConnection**: 建立与 ODBC 数据源的连接。
- ❑ **OleDbCommand**: 执行一个 SQL 语句或存储过程。
- ❑ **OleDbDataReader**: 与 **Command** 对象一起使用，读取 ODBC 数据源。
- ❑ **OleDbDataAdapter**: 创建适配器，用来填充 **DataSet**。
- ❑ **OleDbCommandBuilder**: 用来自动生成插入、更新、删除等操作的 SQL 语句。

这些常用的对象使用方法同其他 ADO.NET 数据操作类相同，使用 **Connection** 对象进行数据库连接，连接后，使用 **Command** 对象进行数据库操作，并使用 **Close** 方法关闭数据连接。

## 10.2.2 建立连接

使用 **Ole Db** 连接数据库基本只需要使用字符串连接方式即可，当需要使用 **Ole Db** 中的对象时，必须使用命名空间 **System.Data.OleDb**，示例代码如下所示。

```
using System.Data.OleDb; //使用 OleDb 命名空间
```

使用命名空间后，就能够创建 **Ole Db** 的 **Connection** 对象，以及 **Command** 对象对数据库进行连接和数据操作。

### 1. 使用接字符串进行连接（ACCESS）

**Access** 数据库是一种桌面级的数据库，同文件类型的数据库类似，所以连接 **Access** 数据库时，必须指定数据库文件的路径，或者使用 **Server.MapPath** 来确定数据库文件的相对位置。示例代码如下所示。

```
string str = "provider=Microsoft.Jet.OLEDB.4.0 ;
Data Source=" + Server.MapPath("access.mdb") + "" ; //设置连接字符串
OleDbConnection con = new OleDbConnection(str); //创建连接对象
try
{
    con.Open(); //打开连接
    Label1.Text = "连接成功"; //显示提示信息
    con.Close(); //关闭连接
}
catch(Exception ee) //抛出异常
{
    Label1.Text = "连接失败";
}
```

上述代码设置了 **Access** 数据库连接字符串，并使用 **OleDb** 的 **Connection** 方法创建数据连接，创建连接后打开数据连接，如果数据连接打开成功，则返回成功，否则返回连接失败。

### 2. 使用接字符串进行连接（SQL Server）

当需要连接 **SQL Server** 时，无需对连接、执行等操作进行修改，只需要对数据库连接字符串进行修改即可，示例代码如下所示。

```
OleDbConnection con= new OleDbConnection();
con.ConnectionString="Provider=SQLOLEDB;Data
Source=(local);Initial Catalog=mytable;uid=sa;pwd=sa"; //初始化连接字符串
try
{
    con.Open(); //尝试打开连接
    Label1.Text = "连接成功"; //显示连接信息
    con.Close(); //关闭连接
}
catch
{
    Label1.Text = "连接失败"; //抛出异常
}
```

上述代码只需修改连接字符串的格式，而无需修改其他 ADO.NET 中的对象，以及对象执行的方法即可。

## 10.3 访问 MySql

**MySql** 是一个开源的小型关系型数据库，**MySql** 数据库功能性强、体积小、运行速度快、成本低和安全性强，并且广泛的被中小型应用所接受。**MySql** 通常情况下和 **PHP** 一起开发使用，在 **ASP.NET** 中，同样能够使用 **MySql** 进行数据库的存储。

### 10.3.1 MySql 简介

**MySql** (<http://www.MySql.com>) 是一套开源的小型关系型数据库，**MySql** 能够执行标准的 **SQL** 语句进行数据操作。**MySql** 能够支持多种操作系统，包括 **Windows**、**Linux**、**Mac OS** 等等，是一种跨平台的数据库产品，并且 **MySql** 还为多种语言提供了 **API**，这些语言包括传统的 **C/C++** 也包括新近的 **Python** 和 **Ruby**，**MySql** 具有功能性强、体积小、运行速度快、成本低和安全性强等特点，**MySql** 还具有以下特性：

- ☐ **MySql** 具有客户端/服务器结构的分布式数据库管理系统。
- ☐ 支持多个操作系统。
- ☐ 为多种编程语言提供 **API**。
- ☐ 支持多用户，多线程操作数据库。
- ☐ 提供了 **TCP/IP**，**ODBC** 和 **JDBC** 等多种数据库连接方式。
- ☐ 优化 **SQL** 语句能力，提升性能。
- ☐ 支持 **ANSI** 标准的所有数据类型。
- ☐ 开放源代码，并且可以免费下载安装。

**MySql** 不仅具有良好的性能表现，同时 **MySql** 还能够执行复杂的 **SQL** 语句操作，但是 **MySql** 的缺点就在于无法创建和使用存储过程，缺少一些大型数据库所必备的功能。

### 10.3.2 建立连接

**ASP.NET** 应用程序需要使用 **ODBC .NET Data Provider** 连接 **MySql** 数据库。在连接数据库之前，**MySql** 数据库能够被 **.NET Data Provider** 识别和驱动必须首先安装 **MySql ODBC 驱动程序** (**MySql-connector-odbc-5.1.5**)，开发人员可以在官方网站免费获取 **MySql ODBC 驱动程序** 并免费下载 (<http://dev.MySql.com/downloads/connector/odbc/5.1.html#windows32>)。单击下载的安装程序，**MySql ODBC 驱动程序** 就会弹出安装向导，并提示安装。通常情况下，只需要选择典型安装即可如图 10-11 和图 10-12 所示。



图 10-11 安装 MySql ODBC 驱动程序



图 10-12 选择安装类型

选择完成后，单击【**Next**】按钮后即可完成安装。安装完成 **MySql ODBC 驱动程序**后，单击【**开始**】菜单找到【**控制面板**】，在【**控制面板**】的【**管理工具**】中选择【**数据源 ODBC**】选项，并在【**数据源 ODBC**



选项】对话框中选择【驱动程序】选项卡，如果【驱动程序】选项卡中已经存在 **MySQL ODBC Driver** 选项，则说明 **MySQL ODBC** 驱动程序已经安装完成。当安装完成后，需要新建 **DSN**，如图 10-13 和图 10-14 所示。



图 10-13 驱动程序已经安装完毕



图 10-14 新建 MySQL DSN

单击完成，系统会弹出对话框用于指定 **DNS** 名、**MySQL** 服务器名、数据库名、密码、端口等信息，如图 10-15 所示。



图 10-15 为 MySQL ODBC 配置 DSN

配置完成后，即可通过使用 **ODBC** 类库进行数据库操作，示例代码如下所示。

```
string str = @"DSN=guojing"; //设置 Connection 属性,使用 MySQL DSN
OdbcConnection con = new OdbcConnection(str); //设置 Connection 对象
con.Open();//打开连接
OdbcDataAdapter da = new OdbcDataAdapter("select * from mytables", con);//创建 DataAdapter
DataSet ds = new DataSet(); //创建 DataSet 对象
da.Fill(ds, "MySqltable"); //填充 DataSet 数据集
```

同样可以创建 **Command** 对象进行数据操作，示例代码如下所示。

```
OdbcCommand cmd = new OdbcCommand("insert into mytables values ('MySQL title')", con);
cmd.ExecuteNonQuery(); //执行 Command 方法
con.Close();
```

上述代码同连接和使用 **SQL** 数据库基本相同，其中 **str** 变量是配置的是 **DSN** 的属性。当需要脱离 **DSN** 连接 **MySQL** 数据库时，可以不需要配置 **DSN** 来访问 **MySQL** 数据库，在编写 **MySQL** 数据库连接字符串时，需要指定驱动程序名称、**IP** 地址或数据库名，常用的关键字如下所示。

- ❑ **Driver:** 设置驱动程序名。
- ❑ **Server:** 设置服务器的 **IP** 地址或者是本地主机。
- ❑ **Database:** 设置数据库名称。
- ❑ **Option:** 设置选项值。



- ❑ **UID**: 设置连接用户名。
- ❑ **PWD**: 设置连接密码。
- ❑ **Port**: 设置连接端口，默认值为 **3306**。

编写 **MySQL** 数据库连接字符串代码如下所示。

```
string strbase = @"Driver=MySQL ODBC 5.1 Driver;Server=localhost;Database=test;UID=guojing";
```

上述字符串能够连接 **MySQL** 数据库，开发人员在连接数据库后就能够使用 **Command** 对象进行相应的数据存储和操作。

## 10.4 访问 Excel

**Excel** 同 **Access** 数据库一样，都是 **Microsoft Office** 办公软件中的一个组件，**Excel** 主要用来处理电子表格，同时 **Excel** 也能够方便的进行数据存储，并提供强大的运算能力和统计功能，经常使用于办公环境。

### 10.4.1 Excel 简介

在办公环境中，大部分的办公人员都使用 **Excel** 进行报表处理，所以，**Excel** 中存储着大量的信息。这些信息对决策者或者是办公自动化管理而言，都比较重要，在办公室应用中，很多文档都是通过 **Excel** 保存在计算机中的，所以在编写应用程序时，常常需要访问 **Excel** 来访问和存储数据。但是，开发人员会发现通过应用程序访问 **Excel** 是一件非常困难的事情。

因为 **Excel** 并不是数据库，所以 **Excel** 不支持相关的数据结构，所以当开发人员需要对 **Excel** 进行数据访问时，会变得比较困难。但是从另一个角度来看，**Excel** 文件是由一张张工作表组成，其结构很像数据库中的表，所以，应该能够通过相应的手段让应用程序访问 **Excel**。

**ASP.NET** 提供了一些类和方法用于连接和访问 **Excel** 数据库，极大的方便了开发人员的开发，简化了开发代码。

### 10.4.2 建立连接

**ASP.NET** 访问 **Excel** 通常有两种方法，一种是使用 **ODBC .NET Data Provider** 进行访问，另一种则是使用 **OLE DB .NET Data Provider** 进行访问。这两种访问方式在原理上基本相同，同 **ADO.NET** 其他对象一样，访问和操作 **Excel** 文件时，都必须使用 **Connection** 对象进行连接，然后使用 **Command** 对象执行 **SQL** 命令。

#### 1. 使用 DSN 连接 Excel 数据源

首先，必须创建一个 **Excel** 数据源，例如 **data.xls**，并手动增加若干数据，如图 10-16 所示。

	A	B	C	D	E
1	编号	学号	年龄	姓名	性别
2	1	20051183049	21	小红	女
3	2	20051183050	21	小白	男
4	3	20051183051	20	小刘	男
5	4	20051183052	22	小赵	男
6					

图 10-16 创建 Excel 数据源

数据源创建完毕后，则需要在【数据源（ODBC）】中添加支持 **Excel** 的数据源，如图 10-17、10-18 所示。



图 10-17 创建数据源



图 10-18 数据源安装

添加数据源之后，需要配置数据源，如图 10-17 所示，则需要选择相应数据文件，单击【选择工作簿】按钮选择数据文件的位置，如图 10-19 所示，配置完毕后，单击【确定】按钮，即可配置成功。



图 10-19 选择数据文件

配置完成后，就可以通过 ASP.NET 应用程序访问 Excel 数据源，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    string str = @"DSN=myexcel"; //使用 ODBC 连接数据源
    OdbcConnection con = new OdbcConnection(str); //新建连接对象
    try
    {
        con.Open(); //尝试打开连接
        Label1.Text = "连接成功"; //显式连接信息
        con.Close(); //关闭连接
    }
    catch
    {
        Label1.Text = "连接失败"; //抛出异常
    }
}
```

执行数据连接后，就可以通过 SQL 语句执行数据源遍历，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    string str = @"DSN=myexcel";
    OdbcConnection con = new OdbcConnection(str);
    try
    {
        con.Open(); //打开连接
        Response.Write("连接成功<br/>"); //输出 HTML
        OdbcDataAdapter da = new OdbcDataAdapter("select * from [Sheet1$]",con); //创建适配器
        DataSet ds = new DataSet(); //创建 DataSet 数据集
        int count=da.Fill(ds, "exceltable"); //填充数据集
        for (int i = 0; i < count; i++) //遍历数据
        {
```

```

        Response.Write(ds.Tables["exceltable"].Rows[i]["姓名"].ToString()+"<hr/>"); //输出数据
    }
}
catch(Exception ee) //抛出异常
{
    Response.Write(ee.ToString());
}
}

```

上述代码使用了 SQL 对 Excel 数据源中的数据进行查询和遍历，运行结果如图 10-20 所示。

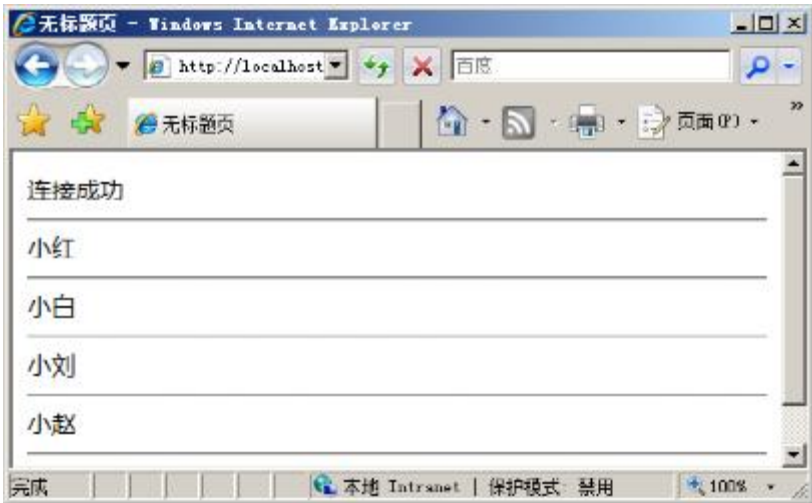


图 10-20 遍历 Excel 数据源

## 2. 使用 OLE DB .NET Data Provider 连接 Excel 数据源

使用 OLE DB .NET Data Provider 连接和操作 Excel 数据源，同其他 ADO.NET 数据源访问方法类似，同样也是使用 OleDbConnection 对象进行数据连接，使用 OleDbCommand 对象进行数据访问，示例代码如下所示。

```

protected void Page_Load(object sender, EventArgs e)
{
    string str =
        @"Provider=Microsoft.Jet.OleDb 4.0;Data
        Source="+Server.MapPath("data.xls")+";Extended Properties= Excel 8.0;"; //设置 Excel 连接串
    OleDbConnection con = new OleDbConnection(str); //创建连接对象
    try
    {
        con.Open(); //尝试打开连接
        Label1.Text = "连接成功"; //显式连接信息
        con.Close(); //关闭连接
    }
    catch
    {
        Label1.Text = "连接失败"; //抛出异常
    }
}

```

上述代码通过使用 OLE DB .NET Data Provider 连接字串进行 Excel 数据源的连接，在连接完成后，其数据操作的方法都与 ADO.NET 对象的操作方法相同。

## 10.5 访问 txt

文本文件（.txt）是一种最基本的文件类型，访问 txt 的方法比较多，不仅能够通过使用 ODBC .NET Data Provider 进行访问，或者使用 OLE DB .NET Data Provider 进行访问。而可以通过 System.IO 进行文本文件的访问。

10.5.1 使用 ODBE.NET Data Provider 连接 txt

使用 **ODBE.NET Data Provider** 建立与 **txt** 文件的连接需要在连接字符串中指定驱动器名，同样可以在管理工具中创建【数据源（ODBC）】来访问 **txt** 文本文件，如图 10-21 和图 10-22 所示。



图 10-21 建立数据源



图 10-22 完成配置数据源

上面创建了 **txt** 数据源的 **ODBC** 数据源，当连接 **txt** 数据源时，可使用 **DSN** 连接数据源，其中 **txt** 文本中的字符如下所示。

```
title
连接 2
连接 3
连接 4
连接 5
连接 6
连接 7
```

当通过 **ODBC** 连接 **txt** 数据源时，只需使用 **Connection** 对象即可，示例代码如下所示。

```
OdbcConnection con = new OdbcConnection(@"DSN=txtexample");
try
{
    con.Open(); //尝试打开连接
    Label1.Text = "连接成功"; //显式连接信息
    con.Close(); //关闭连接
}
catch
{
    Label1.Text = "连接失败"; //抛出异常
}
```

成功创建连接后，就可以对数据源进行操作了。与数据库的结构和 **Excel** 结构不同的是，**txt** 基本上没有类似与表的数据结构，所以在使用 **SQL** 语句时，基本上是通过查询 **txt** 文件来实现的，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    OdbcConnection con = new OdbcConnection(@"DSN=txtexample"); //创建连接
    try
    {
        con.Open(); //打开连接
        OdbcDataAdapter da = new OdbcDataAdapter("select * from data.txt", con); //创建适配器
        DataSet ds = new DataSet(); //创建数据集
        int count=da.Fill(ds, "txttable"); //填充数据集
        for (int i = 0; i < count; i++) //遍历输出
        {
            Response.Write(ds.Tables["txttable"].Rows[i]["title"].ToString()+"<hr/>"); //输出数据
        }
    }
}
```



```
    }
  }
  catch
  {
    Response.Write("连接失败");           //抛出异常
  }
}
```

上述代码遍历了 **txt** 中的数据，运行结果如图 10-23 所示。

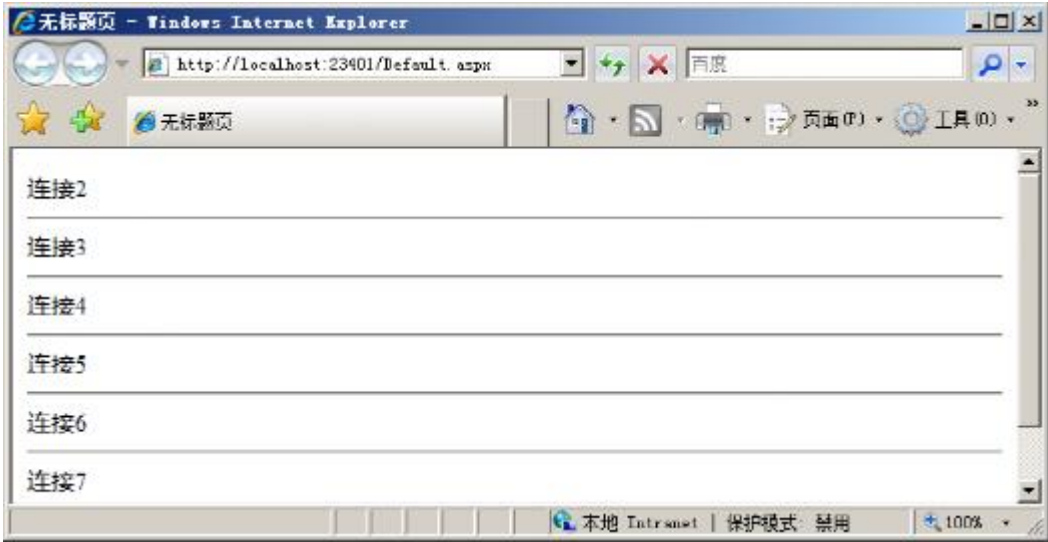


图 10-23 遍历 **txt** 中的数据

10.5.2 使用 OLE DB .NET Data Provider 连接 txt

使用 **OLE DB .NET Data Provider** 建立与 **txt** 文件的连接，只需要在连接字符串中指定提供程序名、数据源名、扩展属性、**HDR** 和 **FMT** 等参数即可，示例代码如下所示。

```
OleDbConnection olecon = new OleDbConnection(@"Provider=Microsoft.Jet.OLEDB 4.0;
Data Source=c:\sample\Extended Properties=text;HDR=yes;FMT=Delimited"); //创建连接对象
当完成连接后，在执行查询等操作时需要指定 txt 文件名，示例代码如下所示。
OdbcDataAdapter da = new OdbcDataAdapter("select * from data.txt", con); //创建适配器
DataSet ds = new DataSet(); //创建数据集
int count=da.Fill(ds, "txttable"); //填充数据集
for (int i = 0; i < count; i++) //遍历数据集
{
    Response.Write(ds.Tables["txttable"].Rows[i]["title"].ToString()+"<br/>"); //输出数据
}
```

注意：**txt** 文件的数据连接字符串中，数据库结构的中“数据库”的概念对于 **txt** 文件而言应该是文件所在的目录，而不是具体的某个文件。而具体的某个文件，相当于是数据库中“表”的概念。

10.5.3 使用 System.IO 命名空间

**System.IO** 能够创建文件，从而与 **txt** 文件进行交互，在使用 **System.IO** 命名空间时，通常需要使用各种类来进行文件操作，常用的类如下所示：

- ❑ **File**: 提供用于创建、复制、删除、移动和打开文件的静态方法。
- ❑ **FileInfo**: 提供创建、复制、删除、移动和代开文件的实例方法。
- ❑ **StreamReader**: 从数据流中读取字符。
- ❑ **StreamWriter**: 向数据流中写入字符。

在进行文件交互操作时，通常使用 **File** 类和 **FileInfo** 类来执行相应的操作。**File** 类和 **FileInfo** 类的基本用法基本相同，但是 **File** 类和 **FileInfo** 类有一些本质的区别。**File** 类和 **FileInfo** 类虽然基本用法相同，但是 **File** 类不用创建类的实例，而 **FileInfo** 类需要创建类的实例，所以 **File** 类可直接调用其类的静态方法执行文件操作，效率也比 **FileInfo** 类高。

**File** 类包含的主要方法如下所示：

- ❑ **OpenText**: 打开现有的文件进行读取。
- ❑ **Exists**: 判断一个文件是否存在。
- ❑ **CreateText**: 创建或打开一个文件以便写入文本字符串。
- ❑ **AppendText**: 将 **txt** 文本追加到现有的文本。

在执行 **txt** 操作时，首先需要判断文件是否存在，如果文件存在，则可用 **File** 类的 **OpenText** 方法打开 **txt** 文件。首先，需要创建一个 **txt** 文件并编写一些内容，内容如下所示。

```
something happend
in my restless dream,i see that place
silent hill
```

编写完成 **txt** 文件后，则可以通过 **File** 类读取 **txt** 文件内容，示例代码如下所示。

```
if (File.Exists(Server.MapPath("data.txt")))           //判断文件是否存在
{
    Response.Write("文件存在");                        //输出文件存在
    File.OpenText(Server.MapPath("data.txt"));          //打开文件
}
```

如果文件存在，并打开了文件，则需要创建 **StreamReader** 对象，使用 **StreamReader** 对象的 **ReadLine** 方法读取一行或 **ReadToEnd** 方法读取整个文本文件。示例代码如下所示。

```
StreamReader rd = File.OpenText(Server.MapPath("data.txt")); //StreamReader 对象
while (rd.Peek() != -1)                                     //如果没有读完
{
    Response.Write(rd.ReadLine() + "<hr/>");                //输出信息
}
```

上述代码中，**Peek** 方法用于返回指定的 **StreamReader** 对象流中的下一个字符，但是不会把这个字符从流中删掉。如果流中没有文本字符，则会返回-1，运行结果如图 10-24 所示。



图 10-24 读取 txt 文本文件

上述代码运行后，页面会从 **txt** 文件中读取内容，并循环遍历输出。

## 10.6 访问 SQLite

**SQLite** 是一款轻量级数据库，其类型在文件形式上很像 **Access** 数据库，但是相比之下 **SQLite** 操作更快。**SQLite** 也是一种文件型数据库，但是 **SQLite** 却支持多种 **Access** 数据库不支持的复杂的 **SQL** 语句，并且还支持事务处理。

10.6.1 SQLite 简介

SQLite 数据库具有小巧和轻量的特点，在 SQLite 数据库开发时，SQLite 是为嵌入式特别准备的，所以 SQLite 具有小巧、资源占用率低等特点。在嵌入式设备中，只需要几百 k 的内存即可。而同时 SQLite 支持多种操作系统，包括 Windows、Linux 等主流操作系统。

SQLite 能够与多种语言结合，包括.NET、PHP、Java 等。同样 SQLite 能够支持 ODBC 接口，相比于 MySQL 数据库而言，SQLite 执行效率更快。虽然 SQLite 小巧，但是 SQLite 同样能够支持 SQL 语句，这些支持的 SQL 语句相比其他的数据库产品，毫不逊色。SQLite 支持的常用 SQL 语句如下所示。

- ❑ CREATE INDEX: 创建索引。
- ❑ CREATE TABLE: 创建表。
- ❑ CREATE TRIGGER: 创建触发器。
- ❑ CREATE VIEW: 创建视图。
- ❑ DELETE: 执行删除操作。
- ❑ DROP INDEX: 删除索引。
- ❑ DROP TABLE: 删除表。
- ❑ DROP TRIGGER: 删除触发器。
- ❑ DROP VIEW: 删除视图。
- ❑ INSERT: 执行插入操作。
- ❑ SELECT: 执行选择，查询操作。
- ❑ UPDATE: 执行更新操作。

SQLite 不仅能够支持常用的 SQL 语句，SQLite 还包括其他 SQL 语句方便开发人员高效的执行数据库操作。

10.6.2 SQLite 连接方法

通过访问 [http://sourceforge.net/project/showfiles.php?group\\_id=132486](http://sourceforge.net/project/showfiles.php?group_id=132486) 页面下载 SQLite for ADO.NET，下载完成后，在应用程序中添加引用即可，如图 10-25 所示。



图 10-25 在项目中添加引用

在添加引用后，可以使用命名空间来为相应的操作提供支持，示例代码如下所示。

```
using System.Data.SQLite; //使用 SQLite 命名空间
```

使用命名控件后，就可以像 ADO.NET 其他对象一样，创建数据库并执行数据库操作。在连接 SQLite 数据库之前，首先需要创建 SQLite 数据库，通过编程的方法可以创建数据库，也可以手动创建 SQLite 数据库，示例代码如下所示。

```
SQLiteConnection.CreateFile(Server.MapPath("sqlite.db")); //创建数据库
```

SQLite 文件后缀可以直接指定，或者不为 SQLite 文件指定后缀，示例代码如下所示。

```
SQLiteConnection.CreateFile(Server.MapPath("sqlite")); //创建无后缀名的数据库
```

**SQLite** 创建成功后，就可以通过 **Connection** 对象连接 **SQLite**，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    SQLiteConnection con = new SQLiteConnection("Data Source="+Server.MapPath("sqlite.db"));
    try
    {
        con.Open();                                //打开连接
        Response.Write("连接成功");                //提示连接成功
    }
    catch(Exception ee)
    {
        Response.Write(ee.ToString());            //连接错误则抛出异常
    }
}
```

上述代码创建了与 **SQLite** 数据库的连接，并尝试打开连接。**SQLite** 数据库同样支持 **DataAdapter** 对象、**Command** 对象，其操作与 **ADO.NET** 其他对象基本上没有任何区别，**.NET** 平台下的开发人员能够很容易的上手 **SQLite**。

## 10.7 小结

本章介绍了 **ADO.NET** 访问其他数据源的知识，这些数据源包括 **MySQL**、**Excel**、**txt**、**SQLite** 等常用的数据源，这些数据源虽然在性能和功能上都与 **SQL Server** 有一段距离，但是在小型、轻便的数据操作和应用中，这些数据库都起着非常重要的作用。本章还介绍了如何使用 **ODBC.NET Data Provider** 连接数据库和使用 **OLE DB .NET Data Provider** 连接数据库，以及 **ODBC.NET Data Provider** 和 **OLE DB .NET Data Provider** 的作用和区别。本章还包括：

- ❑ 使用 **ODBC.NET Data Provider**: 讲解了如何使用 **ODBC.NET Data Provider** 进行数据库存储。
- ❑ 使用 **OLE DB.NET Data Provider**: 讲解了如何使用 **OLE DB.NET Data Provider** 进行数据库存储。。
- ❑ 访问 **MySQL**: 讲解了如何通过 **ADO.NET** 访问 **MySQL**。
- ❑ 访问 **Excel**: 讲解了如何通过 **ADO.NET** 访问 **Excel**。
- ❑ 访问 **txt**: 讲解了如何通过 **ADO.NET** 访问 **txt**。
- ❑ 访问 **SQLite**: 讲解了如何通过 **ADO.NET** 访问 **SQLite**。

通过本章的知识介绍，可以比较深入的了解 **ADO.NET** 和其他数据源的连接、访问、操作等方法，熟练掌握 **ADO.NET**，能够在 **.NET** 平台下的大部分的数据开发中事半功倍。



## 第四篇 ASP.NET 网络编程

第 11 章 用户控件和自定义控件

第 12 章 ASP.NET 的皮肤、主题和母版页

第 13 章 ASP.NET 内置对象,应用程序配置和缓存

第 14 章 ASP.NET XML 和 Web Service

## 第 11 章 用户控件和自定义控件

在 **ASP.NET** 中，系统自带的服务器控件为应用程序开发提供了诸多便利。在应用程序开发中，许多功能都需要重复使用，而如果在应用程序开发中重复的编写类似的代码是非常没有必要的。**ASP.NET** 让开发人员可以自行开发用户控件和自定义控件以提升代码的复用性，本章即将讲解用户控件和自定义控件的开发和使用。

### 11.1 用户控件

在 **ASP** 编程中，开发人员经常使用 **Include** 方式包含其他文件从而简化编程过程。而在 **ASP.NET** 中，控件能够提高应用程序中代码的复用性，不仅 **ASP.NET** 提供了服务器控件，**ASP.NET** 还支持用户自定义控件，从而提高了代码的复用性。

#### 11.1.1 什么是用户控件

用户控件使开发人员能够根据应用程序的需求，方便的定义和编写控件。开发所使用的编程技术将与编写 **Web** 窗体的技术相同，只要开发人员对控件进行修改，就可以将使用该控件的页面的所有控件都进行更改。为了确保用户控件不会被修改、下载，被当成一个独立的 **Web** 窗体来运行，用户控件的后缀名为 **.ascx**，当用户访问页面时，用户控件是不能被用户直接访问的。

**注意：**虽然 **.ascx** 文件会阻止用户的直接访问，但是一些常用的下载工具还是能够下载 **.ascx** 文件。

#### 11.1.2 编写一个简单的控件

用户控件是以 **.ascx** 为后缀名的，在 **Visual Studio 2008** 中，可以通过【添加新项】选项创建一个用户控件，如图 11-1 所示。

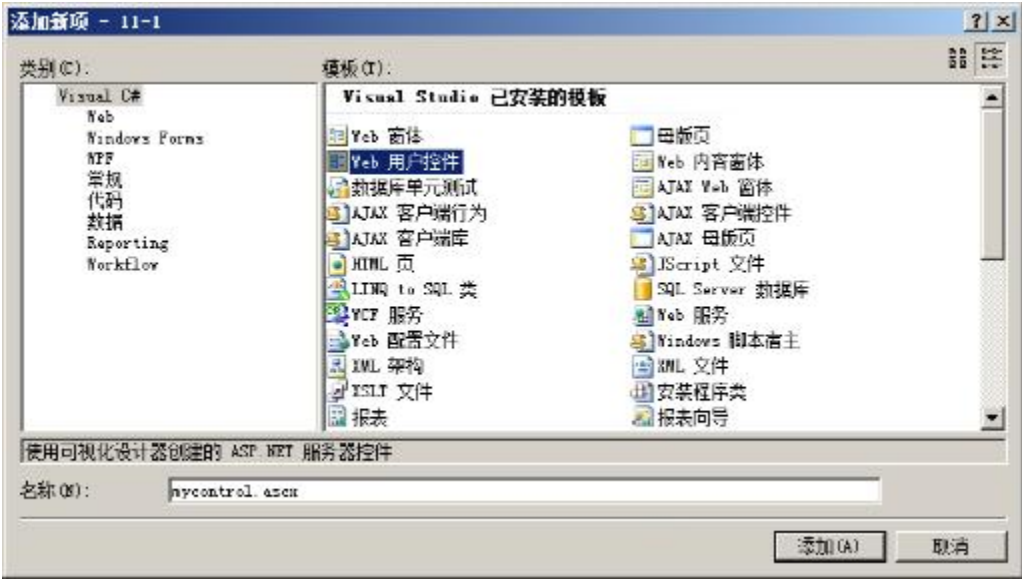


图 11-1 创建用户控件

用户控件创建完毕后，会生成一个`.ascx`页面。`.ascx`页面结构同`.aspx`页面基本没有什么区别。在解决方案管理器中可以打开`.aspx`页面和`.ascx`页面进行对比，其结构并没有太大的变化，如图 11-2 和图 11-3 所示。



图 11-2 创建一个用户控件

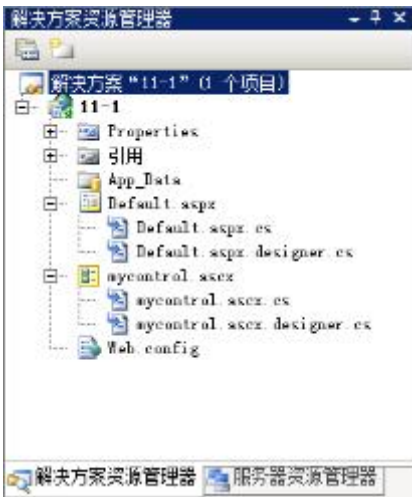


图 11-3 用户控件的结构

用户控件中并没有“`<html><body>`”等标记，因为`.ascx`页面作为控件被引用到其他页面，引用的页面（如`.aspx`页面）其中已经包含`<body><html>`等标记。而如果控件中使用这样的标记，可能会造成页面布局混乱。用户控件创建完成后，`.ascx`页面代码如下所示。

```
<%@ Control Language="C#" AutoEventWireup="true"
    CodeBehind="mycontrol.ascx.cs" Inherits="_11_1.mycontrol" %>
```

其中没有任何的“`<body><html>`”等标记，而`.ascx.cs`页面代码基本同`.aspx`相同，示例代码如下所示。

```
using System; //使用系统命名空间
using System.Collections;
using System.Configuration;
using System.Data;
using System.Linq;
using System.Web; //使用 Web 命名空间
using System.Web.Security;
using System.Web.UI; //使用 UI 命名控件
using System.Web.UI.HtmlControls; //使用 Html 控件命名空间
using System.Web.UI.WebControls; //使用 Web 控件命名空间
using System.Web.UI.WebControls.WebParts;
using System.Xml.Linq; //使用 LINQ 命名空间
namespace _11_1
{
    public partial class mycontrol : System.Web.UI.UserControl //从控件类派生
    {
        protected void Page_Load(object sender, EventArgs e) //页面加载方法
        {
        }
    }
}
```

```
}
}
}
```

用户控件能够提高复用性，前面介绍的服务器控件，从很多情况来说都可以看作是用户控件的一种。当网站需要登录框时，不可能在每个需要登录的地方都重新编写一个登录框，最好的方法是每个页面都能够引用一个登录框。当需要对登录框进行修改时，可以一次性的将所有的页面都修改完毕，而不需要对每个页面都修改登录框。

要达到这种目的，使用用户控件是最好不错的了。**.ascx** 页面允许开发人员拖动服务器控件，并编写相应的样式来实现用户控件，同时用户控件也能够支持事件、方法、委托等高级编程。编写一个用户登录窗口，可以通过几个 **TextBox** 控件和 **Button** 控件来实现，示例代码如下所示。

```
<%@ Control Language="C#"
AutoEventWireup="true" CodeBehind="mycontrol.ascx.cs" Inherits="_11_1.mycontrol" %>
<div style="border:1px solid #ccc; width:300px; background:#f0f0f0; padding:5px 5px 5px 5px; font-size:12px;">
    用户登录<br /><br />
    用户名 : <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox><br /><br />
    密码: <asp:TextBox ID="TextBox2" runat="server"></asp:TextBox><br /><br />
    <asp:Button ID="Button1" runat="server" Text="登录" />
    <asp:HyperLink ID="HyperLink1" runat="server">还没有注册?</asp:HyperLink>
</div>
```

上述代码创建了一个登录框界面。当用户进行网站访问时，网站希望用户能够注册和登录到网站从而提高网站的用户粘度、提升访问量。所以设置登录窗口是非常必要的，界面布局如图 11-4 所示。



图 11-4 编写用户登录界面

当界面布局完毕后，就需要为用户控件编写事件。当用户单击【登录】按钮时，就需要进行事件操作。同 **Web** 窗体一样，双击按钮同样会自动生成事件，示例代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    Label1.Text = "登录成功"; //显示登录信息
}
```

当单击【登录】按钮时，系统提示登录成功，当然这里只是一个简单的用户控件。如果来实现复杂的用户控件的登录窗口，还需要对用户登录进行验证、查询和判断等功能。当用户控件制作完毕后，就可以在其他页面引用用户控件，示例代码如下所示。

```
<%@ Register TagPrefix="Sample" TagName="Login" Src="~/mycontrol.ascx" %> //声明控件引用
```

在这段代码中，有几个属性是必须编写的，这些属性的功能如下所示：

- ❑ **TagPrefix**: 定义控件位置的命名控件。有了命名空间的制约，就可以在同一个页面中使用不同功能的同名控件。
- ❑ **TagName**: 指向所用的控件的名字。
- ❑ **Src**: 用户控件的文件路径，可以为相对路径或绝对路径，但不能使用物理路径。

了解了相关属性，就能够在其他页面中引用该控件了，示例代码如下所示。

```
<%@ Page Language="C#"
AutoEventWireup="true" CodeBehind="Default.aspx.cs" Inherits="_11_1._Default" %>
<%@ Register TagPrefix="Sample" TagName="Login" Src="~/mycontrol.ascx" %>
```



```
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
  <title>用户控件</title>
</head>
<body>
  <form id="form1" runat="server">
    <div>
      <Sample.Login runat="server" id="Login1"></Sample.Login>
    </div>
  </form>
</body>
</html>
```

上述代码声明了用户控件，并使用了用户控件，使用用户控件代码如下所示。

```
<Sample.Login runat="server" id="Login1"></Sample.Login> //使用用户控件
```

从上述代码可以看出，用户控件的格式为 **TagPrefix:TagName**，当声明了用户控件后，就可以使用 **TagPrefix:TagName** 的方式使用用户控件。这样一个用户控件就使用完毕了，如图 11-5 所示。



图 11-5 使用用户控件

运行 **Default.aspx** 页面，虽然在 **Default.aspx** 页面中没有使用制作和编写任何控件，以及代码，但是却已经运行了登录框，这说明用户控件已经被运行了，如图 11-6 所示。



图 11-6 运行用户控件

当需要对登录框进行修改，而无需对页面进行修改时，只需要修改相应的用户控件即可。当多个页面进行同样的用户控件的使用时，若需要对多个页面的控件进行样式或逻辑的更改只需要修改相应的控件，而不需要进行繁冗的多个页面的修正。

### 11.1.3 将 Web 窗体转换成用户控件

在编写用户控件时，会发现 **Web** 窗体的结构和用户控件的结构基本相同。如果开发人员已经开发了 **Web** 窗体，并在今后的需求中决定能够在应用程序全局中能够访问此 **Web** 窗体，那么就可以将 **Web** 窗体改成用户控件。如果需要将 **Web** 窗体更改为用户控件，首先需要对比 **Web** 窗体 and 用户控件的区别：

❑ **Web** 窗体中有 `<body><html><head>` 等标记，而用户控件没有。

❑ **Web** 窗体和用户控件所声明的方法不同。

在了解以上区别后，就可以很容易的将 **Web** 窗体转换成用户控件。首先，只需要删除 `<body><html><head>` 等标记即可。在删除标记后，好需要对两种窗体的声明方式进行更改，对于 **Web** 窗体，其标记方式如下代码所示。

```
<%@ Page Language="C#"
AutoEventWireup="true" CodeBehind="Default.aspx.cs" Inherits="_11_1._Default" %>
```

而对于用户控件，声明代码如下所示。

```
<%@ Control Language="C#"
AutoEventWireup="true" CodeBehind="mycontrol.ascx.cs" Inherits="_11_1.mycontrol" %>
```

在将 **Web** 窗体更改为用户控件时，只需要将 **Page Language** 更改为 **Control Language** 即可。这样就完成了 **Web** 窗体向用户控件的转换过程。

**注意** :有的时候 ,标记中还包括 **ClassName** 属性 ,当包含 **ClassName** 属性时 ,还需要修改相应的 **ClassName** 属性。

## 11.2 自定义控件

用户控件能够执行很多操作。并实现一些功能，但是在复杂的环境下，用户控件并不能够达到开发人员的要求，是因为用户控件大部分都是使用现有的控件进行组装，编写事件来达到目的。于是，**ASP.NET** 允许开发人员编写自定义控件实现复杂的功能。

### 11.2.1 实现自定义控件

自定义控件与用户控件不同，自定义控件需要定义一个直接或间接从 **Control** 类派生的类，并重写 **Render** 方法。在 **.NET** 框架中，**System.Web.UI.Control** 与 **System.Web.UI.WebControls.WebControl** 两个类是服务器控件的基类，并且定义了所有服务器控件共有的属性、方法和事件，其中最为重要的就是包括了控制控件执行生命周期的方法和事件，以及 **ID** 等共有属性。

实现自定义控件，必须创建一个自定义控件，自定义控件将会编译成 **DLL** 文件。创建自定义控件如图 11-7 所示。

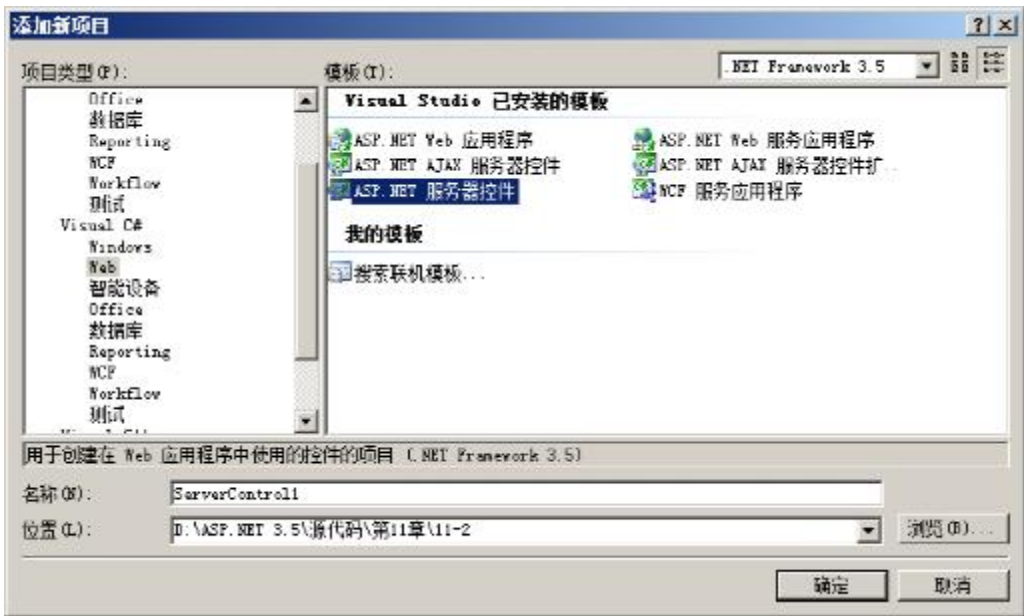


图 11-7 创建自定义控件

自定义控件创建完成后，会自动生成一个类，并在类中生成相应的方法，示例代码如下所示。

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Linq;
using System.Text;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls; //使用 UI 命名空间以便继承
namespace ServerControl1
{
    [DefaultProperty("Text")] //声明属性
    [ToolboxData("<{0}:ServerControl1 runat=server></{0}:ServerControl1>")] //设置控件格式
    public class ServerControl1 : WebControl
    {
        [Bindable(true)] //设置是否支持绑定
        [Category("Appearance")] //设置类别
        [DefaultValue("")] //设置默认值
        [Localizable(true)] //设置是否支持本地化操作
        public string Text //定义 Text 属性
        {
            get //获取属性
            {
                String s = (String)ViewState["Text"]; //获取属性的值
                return ((s == null) ? "[" + this.ID + "]" : s); //返回默认的属性的值
            }
            set //设置属性
            {
                ViewState["Text"] = value;
            }
        }
        protected override void RenderContents(HtmlTextWriter output) //页面呈现
        {
            output.Write(Text);
        }
    }
}
```

开发人员可以在源代码中编写和添加属性。当需要呈现给 **HTML** 页面输出时，只需要重写 **Render** 方法即可，示例代码如下所示。

```
protected override void RenderContents(HtmlTextWriter output)
```

```
{
    output.Write("定义的 Text 属性的值为:" + Text);           //输出为页面呈现
}
```

在使用服务器控件时，会发现控件有很多的属性，例如 **SqlConnection** 属性、**Color** 属性等，如图 11-8 所示。



图 11-8 控件的属性

为了实现服务器控件的智能属性配置，开发人员能够在源代码中编写属性，示例代码如下所示。

```
public string GuoJingString           //编写属性
{
    get { return (String)ViewState["GuoJingString"]; }           //获取属性
    set { ViewState["GuoJingString"] = value; }                 //设置属性
}
```

当自定义控件编写完毕后，需要在需要使用该控件的项目中添加引用。右击现有项目，选择【添加引用】选项，如果是同在一个解决方案下，则只要选择【项目】选项卡即可。而如果不在同一解决方案，则需要选择【浏览】选项卡浏览相应的 **DLL** 文件，如图 11-9 和图 11-10 所示。

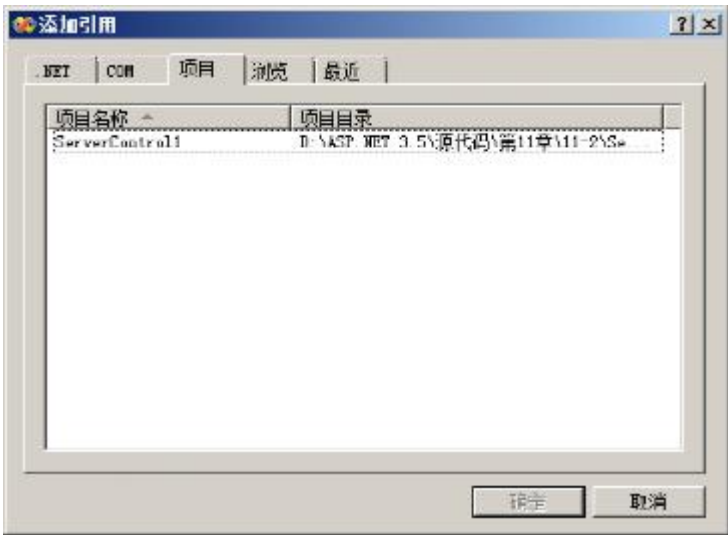


图 11-9 添加项目引用

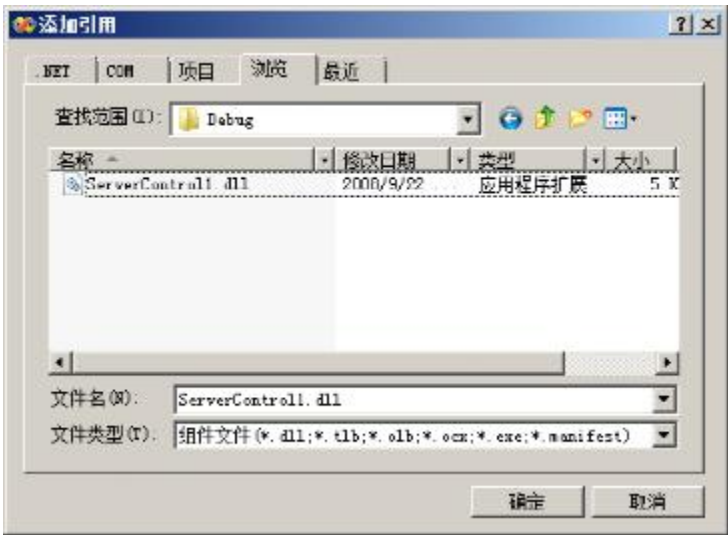


图 11-10 浏览 DLL

单击【确定】按钮完成引用的添加后，就可以在页面中使用此自定义控件。若需要在页面中需要使用此自定义控件，同样同用户控件一样需要在头部声明自定义控件，示例代码如下所示。

```
<%@ Register TagPrefix="MyControl" Namespace="ServerControl1" Assembly="ServerControl1" %>
```

上述代码向页面注册了自定义控件，自定义注册完毕后，就能够在页面中使用该控件。同时，在工具栏中也会呈现自定义控件，如图 11-11 所示。自定义控件呈现在工具箱之后，就可以直接拖动自定义控件到页面，并且配置相应的属性，如图 11-12 所示。





图 11-11 呈现自定义控件



图 11-12 配置自定义属性

正如图 11-12 所示，开发人员能够在自定义控件中编写属性，这些属性可以是共有属性也可以是用户自定义的属性，用户可以拖动自定义控件使用于自己的应用程序中并通过属性进行自定义控件的配置。用户拖动自定义页面到控件后，页面会生成相应的自定义控件的 **HTML** 代码，示例代码如下所示。

```
<form id="form1" runat="server">
  <div>
    <MyControl:ServerControl1 ID="ServerControl11" runat="server" />
  </div>
</form>
```

上述代码就在页面中使用了自定义控件。在 **ASP.NET** 服务器控件中，很多的控件都是通过自定义控件来实现的，开发人员能够开发相应的自定义控件并在不同的应用中使用而无需重复开发。

11.2.2 复合自定义控件

单单一个简单的控件并不能实现太多的效果，在实际开发中，可能需要更多的功能，这种复杂功能控件最常见的就是 **SqlDataSource** 控件。**SqlDataSource** 控件是数据源控件，通过 **SqlDataSource** 控件能够配置数据源，并且实现分页、插入、删除等功能。复合自定义控件就类似这样一个功能复杂的控件。编写复合自定义控件有以下几种方式：

- ❑ 创建用户控件，并使用用户控件封装的用户界面实现复合控件。
- ❑ 开发一个编译控件，封装一个按钮控件和文本框控件，通过重写 **Render** 方法呈现。
- ❑ 从现有的控件中派生出新控件。
- ❑ 从基本控件类之一派生来创建自定义控件。

通过编写复合控件，能够让控件开发更加灵活，控件的使用人员也能够更加方便的配置控件，例如，重写登录控件，前台页面制作人员使用该控件时，可以为控件配置验证等功能，方便前台人员配置和使用。如图 11-13 所示。

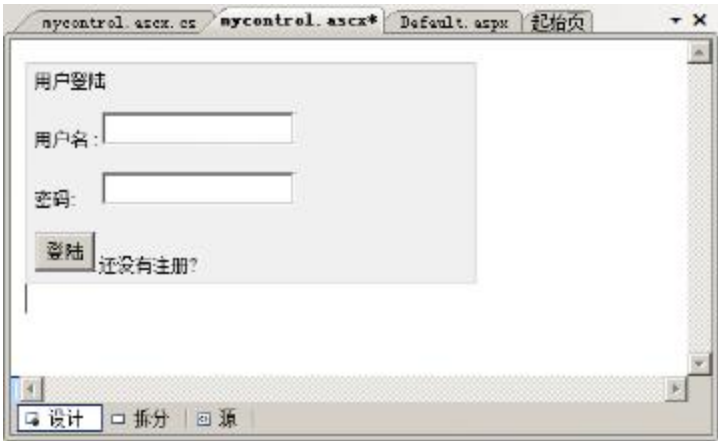


图 11-13 登录控件

为了实现登录控件，就必须在自定义控件中添加相应的服务器控件，在登录控件中，需要两个 **TextBox**

来让用户输入用户名和密码，填写完成后，必须单击登录按钮实现登录事件。在类中创建 **TextBox** 和 **Button** 代码如下所示。

```
public class ServerControl1 : WebControl
{
    //创建服务器控件
    public TextBox NameTextBox = new TextBox(); //创建 TextBox 控件
    public TextBox PasswordTextBox = new TextBox(); //创建密码控件
    public Button LoginButton = new Button(); //创建 Button 控件
    ...
    ...
}
```

上述代码创建了两个 **TextBox** 控件和一个 **Button** 控件。其中，**NameTextBox** 让用户能够输入用户名，而 **PasswordTextBox** 能够让用户输入密码。当用户单击 **LoginButton** 时，就需要实现登录操作，在这里就需要声明一个事件，示例代码如下所示。

```
public event EventHandler LoginClick; //声明事件
```

完成对控件和事件的声明，就需要进行属性的编写。在登录控件中，希望在前台开发人员在开发过程中，能够轻易的配置属性进行使用，从而提高代码的复用性。在图 11-13 所表示控件中，开发人员希望控件的使用人员能够配置背景颜色、边框粗细、内置距离、登录说明和跳转连接等。在代码中，可以分别为这些属性进行配置，示例代码如下所示。

```
[Bindable(true)] //设置是否支持绑定
[Category("Appearance")] //设置类别
[DefaultValue("")] //设置默认值
[Localizable(true)] //设置是否支持本地化操作
public string LoignBackColor
{
    get { return (String)ViewState["LoignBackColor"]; } //获取背景
    set { ViewState["LoignBackColor"] = value; } //设置背景
}
```

上述代码定义了一个属性，在属性定义前，可以对属性进行描述，如代码中 **Bindable**、**Category** 等，这些常用的描述意义如下所示：

- ❑ **Bindable**: 是否用于绑定。
- ❑ **Category**: 属性或事件显示在一个设置为“按分类顺序”的模式，如果不指定，则会显示在杂项中。
- ❑ **DefaultValue**: 指定属性的默认值。
- ❑ **Localizable**: 指定属性是否本地化。

编辑相应属性，在属性配置中就能够做相应的配置，如图 11-14 所示。

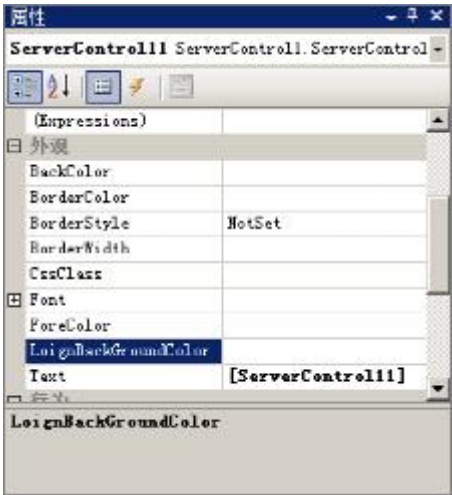


图 11-14 自定义属性

在代码中，将 **Category** 属性设置为 **Appearance**，这个属性就会在【外观】选项卡中出现。配置完成 **LoignBackColor** 后，就可以为其他的属性做相应的配置，示例代码如下所示。

```
[Bindable(true)] //设置是否支持绑定
[Category("Appearance")] //设置类别
```

[DefaultValue("")]	//设置默认值
[Localizable(true)]	//设置是否支持本地化操作
public string LoignBackColor	//设置背景颜色
{	
get { return (String)ViewState["LoignBackColor"]; }	//获取属性的值
set { ViewState["LoignBackColor"] = value; }	//设置属性默认值
}	
//登录边框粗细	
[Bindable(true)]	//设置是否支持绑定
[Category("Appearance")]	//设置类别
[DefaultValue("")]	//设置默认值
[Localizable(true)]	//设置是否支持本地化操作
public string LoginBorderWidth	//设置边框粗细
{	
get { return (String)ViewState["LoginBorderWidth"]; }	//获取边框属性的值
set { ViewState["LoginBorderWidth"] = value; }	//设置边框默认值
}	
//登录的内置距离	
[Bindable(true)]	//设置是否支持绑定
[Category("Appearance")]	//设置类别
[DefaultValue("")]	//设置默认值
[Localizable(true)]	//设置是否支持本地化操作
public string LoginPadding	//设置内置距离
{	
get { return (String)ViewState["LoginPadding"]; }	//获取内置距离的值
set { ViewState["LoginPadding"] = value; }	//设置默认值
}	
//登录说明	
[Bindable(true)]	//设置是否支持绑定
[Category("Appearance")]	//设置类别
[DefaultValue("")]	//设置默认值
[Localizable(true)]	//设置是否支持本地化操作
public string LoginInformation	//设置登录信息
{	
get { return (String)ViewState["LoginInformation"]; }	//获取登录信息的值
set { ViewState["LoginInformation"] = value; }	//设置默认登录信息值
}	
//登录跳转 URL	
[Bindable(true)]	//设置是否支持绑定
[Category("Appearance")]	//设置类别
[DefaultValue("")]	//设置默认值
[Localizable(true)]	//设置是否支持本地化操作
public string ResignURL	//设置登录跳转 URL
{	
get { return (String)ViewState["ResignURL"]; }	//获取 URL 的值
set { ViewState["ResignURL"] = value; }	//设置 URL 默认值
}	

编写完成属性后，就可以通过重写 **Render** 方法呈现不同的 **HTML**，示例代码如下所示。

protected override void RenderContents(HtmlTextWriter output)	//编写页面输出
{	
output.RenderBeginTag(HtmlTextWriterTag.Div);	//创建 Div 标签
output.RenderBeginTag(HtmlTextWriterTag.Tr);	//创建 Tr 标签
NameTextBox.RenderControl(output);	//添加控件
output.RenderBeginTag(HtmlTextWriterTag.Td);	//创建 Td 标签
output.RenderBeginTag(HtmlTextWriterTag.Br);	//创建 Br 标签
output.RenderBeginTag(HtmlTextWriterTag.Tr);	//创建 Tr 标签
PasswordTextBox.RenderControl(output);	//添加控件

```
output.RenderBeginTag(HtmlTextWriterTag.Td);           //输出 Td 标签
}
```

上述代码使用了 **HtmlTextWriter** 类，**HtmlTextWriter** 类能够动态的创建 **HTML** 标签。上述代码中使用了 **HtmlTextWriter** 类的对象的 **RenderBeginTag** 方法创建相应的 **HTML** 标记。重写 **Render** 方法以呈现不同的 **HTML** 后，用户就能够看到登录界面，当用户单击【登录】按钮后，应该执行登录事件，这里应该是个事件冒泡，编写按钮提交事件代码如下所示。

```
public void Submit_Click(object sender, EventArgs e)
{
    EventArgs arg = new EventArgs();           //编写按钮事件方法
    if (LoginClick != null)                     //判断事件冒泡是否为空
    {
        LoginClick(LoginButton, arg);          //触发事件
    }
}
```

编写按钮事件后，整个自定义控件就制作完毕了。相比之下，自定义控件的制作并不是那么难，反而自定义控件能够实现更多的效果，并呈现不同的样式，并且允许界面开发人员能够通过相应的配置呈现不同的样式。

## 11.3 用户控件和自定义控件的异同

对比用户控件和自定义控件，很多人或认为用户控件更加容易开发，而自定义控件的门槛较高，不方便应用程序的开发。其实不然，用户控件更适合创建内部的应用程序特定的控件，例如用户登录控件会在该项目中经常使用，所以创建用户控件能够极快的提高应用程序开发。而自定义控件通常应用到更适合创建通用的可再分发的控件，例如常用的开源 **HTML** 编辑器 **Fckeditor** 就可以说是一个优秀的自定义控件。

通常用户控件在一个项目中经常使用，而自定义控件用来在通用的程序中使用，当网站应用程序开发中，导航控件如果用用户控件实现，是非常方便的。但是通过自定义控件实现，可能并不能适合所有的应用场合，当需要适应其他场合时，可能需要重新开发和编译。具体的讲，用户控件和自定义控件可以从以下几个方面来说明它们的区别：

### 1. 使用率

在选择使用用户控件和自定义控件时，可以首先考虑使用率。如果开发的应用程序只是需要小范围的使用，则可以考虑用户控件，而如果开发的自定义控件能够在大部分的应用程序中被应用，则可以考虑自定义控件。

### 2. 创建技术

用户控件和自定义控件的创建技术是不相同的，并且用户控件和自定义控件创建的难度也不相同，用户控件是以 **.ascx** 形式声明并创建的，开发过程也比较简单，并且有设计器提供设计支持，而自定义控件是从 **System.Web.UI.Control** 派生而来的，开发过程稍微复杂，也没有设计器提供设计支持。

### 3. 生成方式

用户控件和自定义控件生成的方式不同，用户控件是以 **.ascx** 的形式呈现，而自定义控件是以 **DLL** 的形式呈现，通过添加引用，自定义控件能够在【工具箱】中显式，能够像服务器控件一样拖动到页面，并且能够通过编程开发增加自定义属性。而用户控件无法在工具箱显示，也不能够像自定义控件那样增加自定义属性。

### 4. 性能

虽然用户控件和自定义控件编写的过程不同，也遵循不同的创建模型，但是用户控件和自定义控件都是从 **System.Web.UI.Control** 直接或间接的派生的，在性能上没有很大的差别，主要是因为当用户控件在页面中第一次使用时，将作为普通的服务器控件被解析并编译进配件，第二次使用时，就和其他编译型控件一样。



## 11.4 用户控件示例

在用户控件一节中，介绍了如何创建和使用用户控件。创建用户控件能够为应用程序开发起到非常好的作用，并且提高代码的复用性，**ASP.NET** 允许开发人员创建用户控件和自定义控件，并在 **Visual Studio 2008** 中为开发人员提供了原生的开发环境，本节将一步步的进行用户控件的开发。

### 11.4.1 ASP.NET 登录控件

在应用程序开发过程中，登录是必不可少的，例如当用户初次访问该页面时，可以选择登录，也可以选择注册，但是需要有一个登录框作为指导，否则用户无法登录到该网站。当用户再次回访时，登录控件也能够让用户快速的进行登录访问。

作为登录控件，不仅需要对用户的身份进行判断，还需要对用户输入的字符串进行判断，例如，“'”号是 **SQL** 数据库中的关键字，如果非法用户利用了 **SQL** 关键字中的“'”号进行登录，则会出现错误，系统会提示异常，暴露数据库，这样是非常不安全的，所以登录控件还需要对输入的字符串进行判断，判断完成后，如果不是非法用户，则再继续对身份进行验证。

### 11.4.2 ASP.NET 登录控件的开发

**ASP.NET** 登录控件开发起来并不难，主要是需要理清几个基本概念：

- ☐ 用户是否已经注册过，注册过就可以直接登录。
- ☐ 用户如果没注册过，则需要跳转到注册页面。
- ☐ 如果用户输入的是非法字符或是没有任何输入，则先不对身份进行验证，先对输入框进行验证。
- ☐ 用户的验证是通过用户名和密码一起验证的。

当理清了以上思路之后，就能够进行 **ASP.NET** 登录控件的开发了。首先，在现有项目中添加一个新项并在弹出窗口中选择【Web 用户控件】项目，如图 11-15 所示。



图 11-15 添加新项

创建一个用户控件，并命名为 **LoginForm.ascx**，方便在今后的开发中进行识别。

**注意：**良好的页面命名习惯也是一种良好的开发习惯，当页面增多时，可以通过页面命名的含义快速的寻找到相应的页面，当用户控件增多时，良好的命名也可以帮助快速寻找到相应的控件。

当创建完成一个 **LoginForm** 的用户控件后，就需要对用户控件进行页面布局，布局步骤如下所示。

- ❑ 拖动两个 **TextBox**，一个作为用户名输入框，一个作为密码输入框。
  - ❑ 将密码输入框的 **TextMode** 属性设置为 **Password**。
  - ❑ 拖动一个 **Button** 按钮，当用户单击 **Button** 按钮时，进行登录操作。
  - ❑ 拖动一个 **LinkButton** 按钮，当用户单击 **LinkButton** 按钮时，跳转到登录页面。
- 当大概理清了控件的布局后，就可以针对控件的功能进行布局，如图 11-16 所示。



图 11-16 登录框控件初步布局

初步的对登录控件进行布局，发现这样的布局并不美观。对于访问者而言，看到一个并不美观的登录控件，很可能就没有想要登录或注册的想法。一个好的用户登录控件的布局，能够提升访问者的兴趣，于是就需要对控件进行布局更改，这里就需要借助于表格等布局工具，单击菜单栏中的【表】选项，单击下拉菜单中的【插入表】选项，系统会弹出默认的表格窗口。在这里，需要 3 行 2 列进行布局，可以在对话框中选择行数 3、列数 2 进行配置，如图 11-17 和图 11-18 所示。

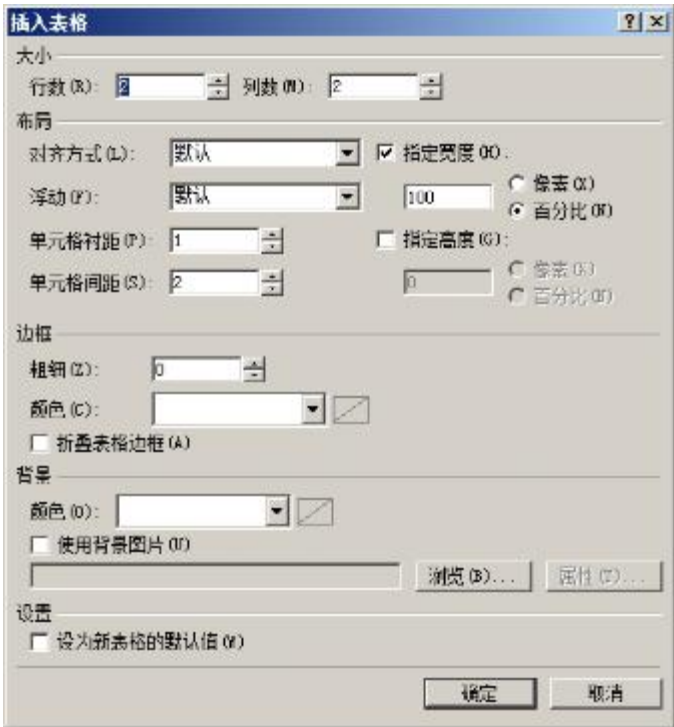


图 11-17 表格默认值



图 11-18 配置表格属性

技巧：配置表格属性后，可以勾选“设为新表格的默认值”，当下一次创建表格时，就无需再次配置，大量的需要同样格式的表格时，可优先考虑此方案。

在编写好表格后，只需要拖动表格进行用户控件的布局即可，布局后 **HTML** 代码如下所示。

```
<style type="text/css">
    .style1
    {
        width: 100%;
        font-size:12px;
    }
</style>
```

```
<div style="border:1px solid #ccc; background:#f0f0f0; font-size:12px;">
  <table class="style1">
    <tr>
      <td>
        用户名 :
      </td>
      <td>
        <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
      </td>
    </tr>
    <tr>
      <td>
        密码&nbsp;   :
      </td>
      <td>
        <asp:TextBox ID="TextBox2" runat="server"></asp:TextBox>
      </td>
    </tr>
    <tr>
      <td>
        <asp:Button ID="Button1" runat="server" Text="登录" />
      </td>
      <td>
        <asp:LinkButton ID="LinkButton1" runat="server">还没有注册?</asp:LinkButton>
      </td>
    </tr>
  </table>
</div>
```

上述代码所呈现的样式如图 11-19 所示。



图 11-19 更改样式后的登录控件

当登录控件的样式初步制作完毕后，就需要增加一些验证控件，并编写登录框的事件。增加验证控件和增加单击事件后的 **HTML** 代码如下所示。

```
<style type="text/css">
  .style1
  {
    width: 100%;
    font-size:12px;
  }
</style>
<div style="border:1px solid #ccc; background:#f0f0f0; font-size:12px;">
  <table class="style1">
    <tr>
      <td>
        用户名 :
```

```
<td>
<asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
<asp:RequiredFieldValidator ID="RequiredFieldValidator1" runat="server"
    ControlToValidate="TextBox1"
    ErrorMessage="用户名不能为空">
</asp:RequiredFieldValidator>
</td>
</tr>
<tr>
<td>
    密码 :</td>
<td>
<asp:TextBox ID="TextBox2" runat="server" TextMode="Password"></asp:TextBox>
<asp:RequiredFieldValidator ID="RequiredFieldValidator2" runat="server"
    ControlToValidate="TextBox2"
    ErrorMessage="密码不能为空">
</asp:RequiredFieldValidator>
</td>
</tr>
<tr>
<td>
<asp:Button ID="Button1" runat="server" Text="登录" onclick="Button1_Click" />
</td>
<td>
<asp:LinkButton ID="LinkButton1" runat="server" PostBackUrl="resign.aspx">
    还没有注册?
</asp:LinkButton>
<asp:Label ID="Label1" runat="server" style="color: #FF3300"></asp:Label>
</td>
</tr>
</table>
</div>
```

HTML 代码确定后，可以编写登录事件，示例代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    if (TextBox1.Text != "test" && TextBox2.Text != "test")           //如果用户名不匹配
    {
        Label1.Text = "登录失败";                                     //提示登录失败
    }
    else
    {
        Label1.Text = "登录成功";                                     //否则登录成功
    }
}
```

上述代码判断如果用户名如果不等于 **test** 并且密码不等于 **test** 的话，则提示登录失败，否则登录成功。

技巧：在这里可以使用 **ADO.NET** 对数据库中的用户表进行操作，通过 **SELECT** 语句查询相应的用户，

如果查询出的值返回值大于 **0**，则说明有这个用户，否则没有这个用户，判断“登录失败”。

11.4.3 ASP.NET 登录控件的使用

编写完成 **ASP.NET** 登录控件后，就可以使用登录控件进行登录页面的制作，在使用登录控件前，必须通过使用 **Register** 关键字向页面注册该用户控件，示例代码如下所示。



```
<%@ Register TagPrefix="Sample" TagName="Login" Src="~/LoginForm.ascx" %>
```

上述代码向页面注册了该控件。当页面被执行时，会通过 **TagPrefix** 以及 **TagName** 判断 **ASP.NET** 标签，并解析成相应的 **ASP.NET** 控件以呈现给页面，然后页面呈现给用户。当需要使用该控件时，只需要在页面中编写引用代码，示例代码如下所示。

```
<Sample:Login runat="server" id="Login1"></Sample:Login>
```

上述代码就在相应的位置显示了用户控件，如图 11-20 所示。

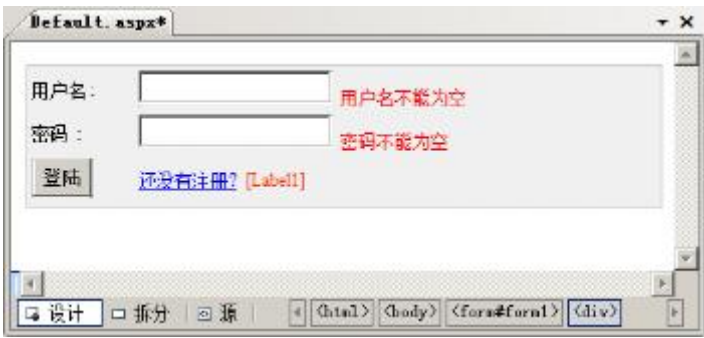


图 11-20 使用用户控件

对使用用户控件的页面进行页面布局，不会影响到用户控件的布局，对用户控件的布局，同样不会影响到页面的布局。相反的是，使用用户控件的页面无需考虑事件，也无需在该页面编写任何 **C#** 代码，这让页面变得非常的简洁，而事件的操作可以交付给用户控件，示例代码如下所示。

```
<%@ Page Language="C#"
AutoEventWireup="true" CodeBehind="Default.aspx.cs" Inherits="_11_4._Default" %>
<%@ Register TagPrefix="Sample" TagName="Login" Src="~/LoginForm.ascx" %>
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
<title>无标题页</title>
</head>
<body>
<form id="form1" runat="server">
<div>
<Sample:Login runat="server" id="Login1"></Sample:Login>
</div>
</form>
</body>
</html>
```

从上述代码可以看出，使用用户控件，并没有任何冗余的代码，这让页面代码显得非常的整洁和整齐。虽然从表面看上去只有 **HTML** 代码，但是并没有影响页面中程序的实现。运行后如图 11-21 所示。



图 11-21 使用用户控件

虽然在页面中并没有实现控件的布局 and 事件的处理，但是在页面依旧可以呈现相应的控件布局 and 事件处理。自定义控件的好处就在于能够将复杂的样式或事件存储在一个控件中，以便在不同的页面中进行使用。

## 11.5 自定义控件实例

虽然用户控件能够尽快的上手并运用在开发中，但是自定义控件的编写能够实现更多的效果。如分页效果在大部分的数据索引中，都需要使用分页。如果存在这么一个分页控件，只需要指定需要分页的表，那么可以自动分页，就能够更加方便应用程序开发了。

### 11.5.1 ASP.NET 分页控件

ASP.NET 能够编写自定义控件，并将自定义控件编译为 DLL 文件以保证在任何其他的项目中能够使用自定义控件。在 ASP.NET Web 应用程序开发中，对于数据的索引，通常情况下是不可能全部将数据索引到一个页面的，所以在显示数据时，就需要进行分页操作。

当用户打开一个页面时，如果将全部的数据一起显示在页面，不仅页面臃肿难看，并且用户很难找到自己需要的信息。对于页面数据的整理和索引，能够让用户更加方便的找到自己需要的信息。不仅如此，如果一次全部的将数据呈现到 HTML 页面，势必会造成 HTML 页面数据的冗长，在运行页面时，也会增加服务器的压力，让 Web 应用程序变得非常的缓慢。

#### 1. 属性设置

使用 ASP.NET 分页控件，能够让数据分开显示，让用户能够自行选择，并且能够自行选择页码，查看相应的数据，创建一个 MyPager 自定义控件，用来执行分页操作，如图 11-22 所示。



图 11-22 创建 MyPager 控件

创建完成后，就需要确定一些基本的属性，这些属性能够方便控件的使用者进行相应的配置，能够尽快的使用控件并完成编程目的。对于分页控件，通常需要确定的属性如下所示：

- ☐ **PageSize:** 用户希望一个页面呈现多少数据。
- ☐ **Server:** 数据库服务器的地址。
- ☐ **Database:** 数据库服务器的数据库。
- ☐ **Pwd:** 数据库服务器有效的密码。
- ☐ **Uid:** 数据库服务器有效的用户名。
- ☐ **Table:** 需要执行分页的表，如果不指定 **SqlCommand**，则自动生成语句。
- ☐ **SqlCommand:** 如果不指定表，则执行 **SqlCommand**。
- ☐ **IndexPage :** 一开始的索引页面。
- ☐ **PageName:** 当前页面的名称，用于跳转。

#### 2. 数据属性配置

在基本确定了以上属性后，就可以编写相应代码，首先需要为数据库连接和数据库的 SQL 语句的功能

实现编写相应的属性，示例代码如下所示。

[DefaultProperty("Text")]	//默认属性
[ToolboxData("<{0}:MyPager runat=server></{0}:MyPager>")]	//控件呈现代码
public class MyPager : WebControl	
{	
[Bindable(true)]	//设置是否支持绑定
[Category("Appearance")]	//设置类别
[DefaultValue("")]	//设置默认值
[Localizable(true)]	//设置是否支持本地化操作
public string Text	//Text 文本属性
{	
get	//获取属性
{	
String s = (String)ViewState["Text"];	//获取文本属性
return ((s == null) ? "[" + this.ID + "]" : s);	//设置文本属性默认值
}	
set	//设置属性
{	
ViewState["Text"] = value;	
}	
}	
[Bindable(true)]	//设置是否支持绑定
[Category("Data")]	//设置类别 Data
[DefaultValue(10)]	//设置默认值
[Localizable(true)]	//设置是否支持本地化操作
public int PageSize	//分页属性
{	
get	
{	
try	
{	
Int32 s = (Int32)ViewState["PageSize"];	//获取分页值
return ((s.ToString() == null) ? 10 : s);	//设置默认值
}	
catch	//如果用户输入异常
{	
return 10;	//返回分页数
}	
}	
set	
{	
ViewState["PageSize"] = value;	//设置分页
}	
}	

上述属性设置了分页属性，当开发人员使用该控件进行分页设置时，该控件会根据不同的分页属性进行分页设置并进行分页操作。在执行分页前，还需要进行数据库的连接，这就需要编写数据连接属性，示例代码如下所示。

[Bindable(true)]	//设置是否支持绑定
[Category("Data")]	//设置类别 Data
[DefaultValue("")]	//设置默认值
[Localizable(true)]	//设置是否支持本地化操作
public string Server	//服务器地址
{	
get	
{	
String s = (String)ViewState["Server"];	//设置服务器地址
return ((s == null) ? "(local)" : s);	//默认服务器地址
}	

```

    }
    set
    {
        ViewState["Server"] = value;
    }
}
[Bindable(true)]
[Category("Data")]
[DefaultValue("")]
[Localizable(true)]
public string DataBase
{
    get
    {
        String s = (String)ViewState["DataBase"];
        return ((s == null) ? ("none") : s);
    }
    set
    {
        ViewState["DataBase"] = value;
    }
}
[Bindable(true)]
[Category("Data")]
[DefaultValue("")]
[Localizable(true)]
public string Uid
{
    get
    {
        String s = (String)ViewState["Uid"];
        return ((s == null) ? ("uid") : s);
    }
    set
    {
        ViewState["Uid"] = value;
    }
}
[Bindable(true)]
[Category("Data")]
[DefaultValue("")]
[Localizable(true)]
public string Pwd
{
    get
    {
        String s = (String)ViewState["Pwd"];
        return ((s == null) ? ("none") : s);
    }
    set
    {
        ViewState["Pwd"] = value;
    }
}
[Bindable(true)]
[Category("Data")]
[DefaultValue("")]

```

//设置默认值

//设置是否支持绑定

//设置类别 Data

//设置默认值

//设置是否支持本地化操作

//数据库属性

//设置数据库属性

//设置默认值

//设置默认值

//设置是否支持绑定

//设置类别 Data

//设置默认值

//设置是否支持本地化操作

//数据库用户名

//设置 UID 属性

//设置默认值

//设置默认值

//设置是否支持绑定

//设置类别 Data

//设置默认值

//设置是否支持本地化操作

//数据库密码

//设置密码属性

//设置默认值

//设置默认值

//设置是否支持绑定

//设置类别 Data

//设置默认值



[Localizable(true)]	//设置是否支持本地化操作
public string Table	//分页的表
{	
get	
{	
String s = (String)ViewState["Table"];	//设置分页的表
return ((s == null) ? ("none") : s);	//设置默认值
}	
set	
{	
ViewState["Table"] = value;	//设置默认值
}	
}	
[Bindable(true)]	//设置是否支持绑定
[Category("Data")]	//设置类别 Data
[DefaultValue("")]	//设置默认值
[Localizable(true)]	//设置是否支持本地化操作
public string SqlCommand	//Sql 语句
{	
get	
{	
String s = (String)ViewState["SqlCommand"];	//设置 SQL 语句
return ((s == null) ? ("none") : s);	//设置默认值
}	
set	
{	
ViewState["SqlCommand"] = value;	//设置默认值
}	
}	

上述代码为数据库连接进行了属性配置，这些属性分别包括数据库连接字符串、数据库服务器 IP、数据库用户名、数据库密码等，这些属性用于配置不同的数据库以呈现不同的数据。

3. 页面属性配置

在编写了数据属性后，还需要编写相应的页面属性以便能够对页面进行控制，这些属性包括页面名称、索引等，示例代码如下所示。

[Bindable(true)]	//设置是否支持绑定
[Category("Data")]	//设置类别 Data
[DefaultValue("")]	//设置默认值
[Localizable(true)]	//设置是否支持本地化操作
public string IndexPage	//索引页面
{	
get	
{	
String s = (String)ViewState["IndexPage"];	//获取当前页面配置属性
return ((s == null) ? ("none") : s);	//返回默认值
}	
set	
{	
ViewState["IndexPage"] = value;	//设置默认配置属性
}	
}	
[Bindable(true)]	//设置是否支持绑定
[Category("Data")]	//设置类别 Data
[DefaultValue("")]	//设置默认值
[Localizable(true)]	//设置是否支持本地化操作
public string PageName	//页面名称
{	

```

get
{
    String s = (String)ViewState["PageName"];           //获取页面名称
    return ((s == null) ? ("none") : s);                //设置默认值
}
set
{
    ViewState["PageName"] = value;                      //返回默认值
}
}

```

在编写完成相应的属性后，就需要重写 **Render** 方法来执行 **HTML** 流输出，示例代码如下所示。

```

protected override void RenderContents(HtmlTextWriter output)
{
    string html = "";
    string str = "server=" + Server + ";database=" + DataBase + ";uid=" + Uid + ";pwd=" + Pwd + "";
    string strsql = "";
    SqlConnection con = new SqlConnection(str);           //创建连接对象
    try
    {
        con.Open();                                       //打开数据连接
        if (SqlCommand == "none" || SqlCommand == "")   //如果不自定义 Table 则自动生成
        {
            strsql = "select count(*) as mycount from " + Table + ""; //生成 SQL 语句
        }
        else
        {
            strsql = SqlCommand;                          //设置默认 SQL 语句
        }
        SqlDataAdapter da = new SqlDataAdapter(strsql, con); //创建适配器
        DataSet ds = new DataSet();                       //创建数据集
        int count = da.Fill(ds, "count");                 //填充数据集
        int page = 0;                                     //数据表中的行数
        int pageCount = 0;                                //分页数
        if (count > 0) //获取数据表中的行数
        {
            page = Convert.ToInt32(ds.Tables["count"].Rows[0]["mycount"].ToString());
        }
        if (page % PageSize > 0)                          //开始分页
        {
            pageCount = (page / PageSize) + 1;           //分页计算
        }
        else
        {
            pageCount = page / PageSize;
        }
        html += "<table><tr>";
        for (int i = 0; i < pageCount; i++)
        {
            if (IndexPage != i.ToString())                //如果查看的是当前页面，则高亮显示
            {
                html += "<td style=\"padding:5px 5px 5px 5px;background:#f0f0f0;border:1px dashed #ccc;\">"; //呈现相应的 HTML
            }
            else
            {
                html += "<td style=\"padding:5px 5px 5px 5px;background:Gray;border:1px dashed #ccc;\">"; //呈现相应的 HTML
            }
        }
    }
}

```

```
        }
        html += "<a href=\"\" + PageName + "?page=" + i + "\">" + i + "</a>";
        html += "</td>"; //完成 HTML
    }
    html += "</tr></table>"; //完成 HTML
}
catch(Exception ee) //出现错误抛出异常
{
    html = ee.ToString(); //输出异常
}
finally
{
    output.Write(html); //页面呈现
    con.Close();
}
}
```

上述代码会查询相应的数据库，例如 **Table**。当查询到了相应的数据库中数据的行数之后，再与 **PageSize** 属性进行对比，并执行分页查询，查询完成后将通过查询的条目数循环遍历输出 **HTML**，并将 **HTML** 呈现在页面中。

11.5.2 ASP.NET 分页控件的使用

自定义控件能够使用在各种其他的应用程序开发中。而任何其他的应用程序如果需要使用分页控件，则需要首先添加引用。右击当前项目，选择【添加引用】选项，单击【浏览】选项卡，浏览到项目的 **bin** 目录下，选择相应的 **DLL** 文件即可，如图 11-23 所示。

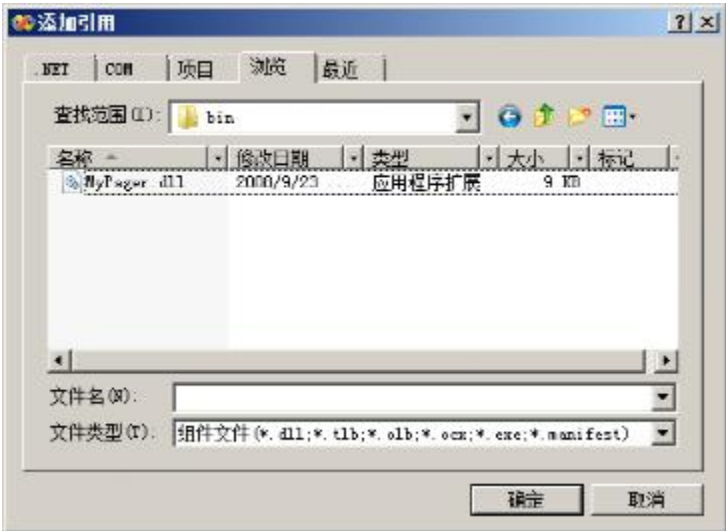


图 11-23 添加引用

添加引用后，同样需要在需要使用的页面进行注册，示例代码如下所示。

```
<%@ Register TagPrefix="MyControl" Namespace="MyPager" Assembly="MyPager" %>
```

添加注册后，在工具栏中就会显示自定义控件，拖动自定义控件到页面中就可以使用自定义控件并配置相应的属性，系统自动生成的 **HTML** 代码如下所示。

```
<%@ Register TagPrefix="MyControl" Namespace="MyPager" Assembly="MyPager" %>
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>无标题页</title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <MyControl:MyPager ID="MyPager1" runat="server" DataBase="mytable"
```

```
IndexPage="1" PageName="default.aspx" PageSize="1" Pwd="Bbg0123456#"
Table="mynews" Uid="sa" />
</div>
</form>
</body>
</html>
```

为了能够使分页控件能够自动进行分页，则需要对属性进行相应的配置，如图 11-24 所示。配置完毕后，控件就能够实现自动分页，如图 11-25 所示。



图 11-24 配置属性



图 11-25 分页控件

通过修改相应的属性，能够为不同的表，甚至不同的数据库进行分页操作，运行结果如图 11-26 所示。



图 11-26 运行分页控件

当配置 **PageSize** 属性为 **1** 时，系统则会按每页 **1** 个数据来进行分页，同样当配置 **PageSize** 属性为 **10** 时，则系统会按照每页 **10** 个数据来进行分页。

注意：在编写分页控件时，这里需要配置服务器信息，这样做是非常不安全的，这里的属性配置可以放在 **Web.config** 文件中，这样配置就更加的安全，具体可参考 **SqlDataSource** 的做法。

## 11.6 小结

本节在服务器控件的基础上，着重讲解了用户控件和自定义控件。使用用户控件和自定义控件的优势就在于，用户控件和自定义控件都能够非常简单的完成并且能够达到开发的需求，而无需重复的进行代码编写。

在传统的开发概念中，用户控件和自定义控件都比较复杂，而通过本章就能够了解到用户控件和自定义控件的开发并没有想象中的复杂，用户控件和自定义控件能够适应更多的应用场合，这些控件能够重复使用和自定义，极大的方便了应用程序的开发。本章还包括：

- ❑ 用户控件：包括什么是用户控件和如何创建用户控件。



- ☐ 将 **Web** 窗体转换成用户控件。
- ☐ 复合自定义控件。
- ☐ 用户控件和自定义控件的异同。

本章分别通过实例创建和使用用户控件和自定义控件，能够轻松的了解用户控件和自定义控件的异同和编程模型，对以后的开发有很大的帮助。

## 第 12 章 ASP.NET 的皮肤、主题和母版页

在 Web 应用程序开发中，一个好的 Web 应用程序界面能够让网站的访问者耳目一新，当用户访问 Web 应用时，网站的界面和布局能够提升访问者对网站的兴趣和继续浏览的耐心。ASP.NET 提供了皮肤、主题和模板页的功能增强了网页布局和界面优化的功能，这样即可轻松的实现对网站开发中界面的控制。

### 12.1 皮肤和主题

皮肤和主题是自 ASP.NET 2.0 就包括的内容，使用皮肤和主题，能够将样式和布局信息分解到单独的文件中，让布局代码和页面代码相分离。主题可以应用到各个站点，当需要更改页面主题时，无需对每个页面进行更改，只需要针对主题代码页进行更改即可。

#### 12.1.1 CSS 简介

在任何 Web 应用程序的开发过程中，CSS（Cascading Style Sheets，级联样式表）都是非常重要的页面布局方法，而且 CSS 也是最高效的页面布局方法。CSS 发展于 1994 年 10 月，是为了补救 HTML 3.2 语法中的不足，但是由于当时网络的发展的不足和浏览器的支持率较低，直到 1996 年底，才正式发表了 CSS 1.0 规格，也正是 1996 年之后，浏览器才开始正式的支持 CSS。

在网页布局中，CSS 经常被使用于页面样式布局和样式控制。熟练的使用 CSS 能够让网页布局更加的方便，在页面维护时，也能够减少工作量。通常 CSS 能够支持三种定义方式，一是直接将样式控制放置于单个 HTML 元素内，称为内联式；二是在网页的 head 部分定义样式，称为嵌入式；三是以扩展名为.css 文件保存样式，称为外联式。

这三种样式适用于不同的场合，内联式适用于对单个标签进行样式控制，这样的好处就在于开发方便，而在维护时，就需要针对每个页面进行修改，非常的不方便；而嵌入式可以控制一个网页的多个样式，当需要对网页样式进行修改时，只需要修改 head 标签中的 style 标签即可，不过这样仍然没有让布局代码和页面代码完全分离；而外联式能够将布局代码和页面代码相分离，在维护过程中，能够减少工作量。

#### 12.1.2 CSS 基础

CSS 能够通过编写样式控制代码来进行页面布局，在编写相应的 HTML 标签时，可以通过 Style 属性进行 CSS 样式控制，示例代码如下所示。

```
<body>
  <div style="font-size:14px;">这是一段文字</div>
</body>
```

上述代码使用内联式进行样式控制，并将属性设置为 font-size:14px，其意义就在于定义文字的大小为 14px；同样，如果需要定义多个属性时，可以同写在一个 style 属性中，示例代码如下所示。

```
<body>
  <div style="font-size:14px;">这是一段文字 1</div>
  <div style="font-size:14px; font-weight:bolder">这是一段文字 2</div>
  <div style="font-size:14px; font-style:italic">这是一段文字 3</div>
  <div style="font-size:14px; font-variant:small-caps">This is My First CSS code</div>
```

```
<div style="font-size:14px; color:red">这是一段文字 5</div>
</body>
```

上述代码分别定义了相关属性来控制样式，并且都使用内联式定义样式，这些 CSS 的属性的意义如下所示：

- ❑ 字体名称属性（**font-family**）：该属性设定字体名称，如 **Arial**、**Tahoma**、**Courier** 等，可以定义字体的名称。
- ❑ 字体大小属性（**font-size**）：该属性可以设置字体的大小。字体大小的设置可以有多种方式，最常用的就是 **pt** 和 **px**。
- ❑ 该属性有三个值可选：**normal**、**italic**、**oblique**、**normal** 是默认值，**italic**、**oblique** 都是斜体显示。
- ❑ 字体粗细属性（**font-weight**）：该属性常用值是 **normal** 和 **bold**，**normal** 是默认值，**bold** 是粗体。
- ❑ 字体变量属性（**font-variant**）：该属性有两个值 **normal** 和 **small-caps**，**normal** 是默认值。**small-caps** 表示字体将被显示成大写。
- ❑ 字体属性（**font**）：该属性是各种字体属性的一种快捷的综合写法。
- ❑ 字体颜色(**color**)：该属性用来控制字体颜色。

这些属性分别定义了字体属性，如图 12-1 所示。

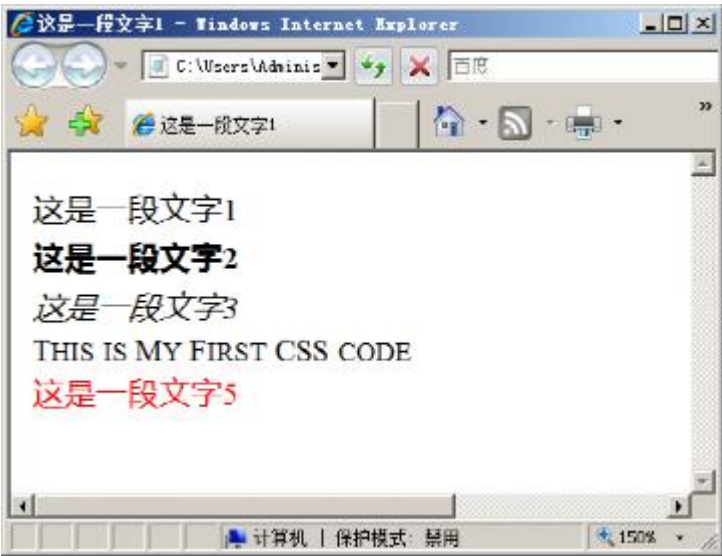


图 12-1 CSS 样式控制

用内联式的方法进行样式控制固然简单，但是在维护过程中却是非常的复杂和难以控制。当需要对页面中的布局进行更改时，则需要对每个页面的每个标签的样式进行更改，这样无疑增大的工作量，当需要对页面进行布局时，可以使用嵌入式的方法进行页面布局，示例代码如下所示。

```
<head>
  <meta content="text/html; charset=utf-8" http-equiv="Content-Type" />
  <title>这是一段文字 1</title>
  <style type="text/css">
    .font1
    {
      font-size:14px;
    }
    .font2
    {
      font-size:14px;
      font-weight:bolder;
    }
    .font3
    {
      font-size:14px;
      font-style:italic;
    }
    .font4
    {
      font-size:14px;
```

```

        font-variant:small-caps;
    }
    .font5
    {
        font-size:14px;
        color:red;
    }
</style>
</head>

```

上述代码分别定义了 5 种字体样式，这些样式都是通过“.”号加样式名称定义的，在定义了字体样式后，就可以在相应的标签中使用 **class** 属性来定义样式，示例代码如下所示。

```

<body>
    <div class="font1">这是一段文字 1</div>
    <div class="font2">这是一段文字 2</div>
    <div class="font3">这是一段文字 3</div>
    <div class="font4">This is My First CSS code</div>
    <div class="font5">这是一段文字 5</div>
</body>

```

其运行后的结果依然同 11-12 所示，但是这样编写代码在维护起来更加的方便，只需要找到 **head** 中的 **style** 标签，就可以对样式进行全局控制。虽然嵌入式能够解决单个页面的样式问题，但是这样只能针对单个页面进行样式控制，而在很多网站的开发应用中，大量的页面样式基本相同，只有少数的页面不尽相同，所以使用嵌入式还是有不足，这里就可以使用外联式。使用外联式，必须创建一个 **.css** 文件后缀的文件，并在当前页面中添加引用，**.css** 页面代码如下所示。

```

.font1
{
    font-size:14px;
}
.font2
{
    font-size:14px;
    font-weight:bolder;
}
.font3
{
    font-size:14px;
    font-style:italic;
}
.font4
{
    font-size:14px;
    font-variant:small-caps;
}
.font5
{
    font-size:14px;
    color:red;
}

```

在 **.css** 文件中，只需要定义如 **head** 标签中的 **style** 标签的内容即可，其编写方法也与内联式和内嵌式相同。在编写完成 **CSS** 文件后，需要在使用的页面的 **head** 标签中添加引用，示例代码如下所示。

```

<link href="css.css" type="text/css" rel="stylesheet"></link>

```

上述代码添加了一个 **css.css** 文件的引用，意在告诉浏览器当前页面的一些样式可以在 **css.css** 中找到并解析。在使用了外联式后，当前页面的 **HTML** 代码就能够变得简单和整洁，示例代码如下所示。

```

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta content="text/html; charset=utf-8" http-equiv="Content-Type" />

```



```
<title>这是一段文字 1</title>
<link href="css.css" type="text/css" rel="stylesheet"></link>
</head>
<body>
  <div class="font1">这是一段文字 1</div>
    <div class="font2">这是一段文字 2</div>
      <div class="font3">这是一段文字 3</div>
        <div class="font4">This is My First CSS code</div>
          <div class="font5">这是一段文字 5</div>
</body>
</html>
```

使用外联式能够很好的将页面布局的代码和 **HTML** 代码相分离，这样不仅能够让多个页面同时使用一个 **CSS** 样式表进行样式控制，同样在维护的过程中，只需要修改相应的 **CSS** 文件中的样式的属性即可实现该样式在所有的页面中都进行更新的操作。这样无疑是减少了工作量，提高的代码的可维护性。可见外联式能够让样式控制既方便又灵活。

### 12.1.3 CSS 常用属性

**CSS** 不仅能够控制字体的样式，**CSS** 还具有强大的样式控制功能，包括背景，边框，边距等属性，这些属性能够为网页布局提供良好的保障，熟练的使用这些属性能够极大的提高 **Web** 应用的友好度。

#### 1. CSS 背景属性

**CSS** 能够描述背景，包括背景颜色、背景图片、背景图片重复方向等属性，这些属性为页面背景的样式控制提供了强大的支持，这些属性包括如下所示：

- ❑ 背景颜色属性（**background-color**）：该属性为 **HTML** 元素设定背景颜色。
- ❑ 背景图片属性（**background-image**）：该属性为 **HTML** 元素设定背景图片。
- ❑ 背景重复属性（**background-repeat**）：该属性和 **background-image** 属性连在一起使用，决定背景图片是否重复。如果只设置 **background-image** 属性，没设置 **background-repeat** 属性，在缺省状态下，图片既 **x** 轴重复，又 **y** 轴重复。
- ❑ 背景附着属性（**background-attachment**）：该属性和 **background-image** 属性连在一起使用，决定图片是跟随内容滚动，还是固定不动。
- ❑ 背景位置属性（**background-position**）：该属性和 **background-image** 属性连在一起使用，决定了背景图片的最初位置。
- ❑ 背景属性（**background**）：该属性是设置背景相关属性的一种快捷的综合写法。

通过这些属性能够为网页背景进行样式控制，示例代码如下所示。

```
body
{
    background-color:green;
}
```

上述代码设置了网页的背景颜色为绿色，如图 12-2 所示。同样，设计人员能够使用 **background-image** 属性设置背景图片，并说明图片是否重复，如图 12-3 所示。



图 12-2 修改背景颜色



图 12-3 背景图片

当使用 **background-image** 属性设置背景图片时，还需要使用 **background-repeat** 属性进行循环判断，示例代码如下所示。

```
body
{
    background-image:url('bg.jpg');
    background-repeat:repeat-x;
}
```

上述代码将 **bg.jpg** 作为背景图片，并且以 **x** 轴重复，如果不编写 **background-repeat** 属性，则默认是即 **x** 轴重复也 **y** 轴重复。上述代码还可以简写，示例代码如下所示。

```
body
{
    background:green url('bg.jpg') repeat-x;
}
```

2. CSS 边框属性

CSS 还能够进行边框的样式控制，使用 CSS 能够灵活的控制边框，边框属性包括有：

- ❑ 边框风格属性（**border-style**）：该属性用来设定上下左右边框的风格。
- ❑ 边框宽度属性（**border-width**）：该属性用来设定上下左右边框的宽度。
- ❑ 边框颜色属性（**border-color**）：该属性设置边框的颜色。
- ❑ 边框属性（**border**）：该属性是边框属性的一个快捷的综合写法。

通过这些属性能够控制边框样式，示例代码如下所示。

```
.mycss
{
    border-bottom:1px black dashed;
    border-top:1px black dashed;
    border-left:1px black dashed;
    border-right:1px black dashed;
}
```

上述代码分别设置了边框的上部分、下部分、左部分、右部分的边框属性，来形式一个完整的边框，同样可以使用边框属性来整合这些代码，示例代码如下所示。

```
.mycss
{
    border:1px black dashed;
}
```

3. CSS 边距和间隙属性

CSS 的边距和间隙属性能够控制标签的位置，CSS 的边距属性使用的是 **margin** 关键字，而间隙属性使用的是 **padding** 关键字。CSS 的边距和间隙属性虽然都是一种定位方法，但是边距和间隙属性定位的对象不同，也就是参照物不同，如图 12-4 所示。

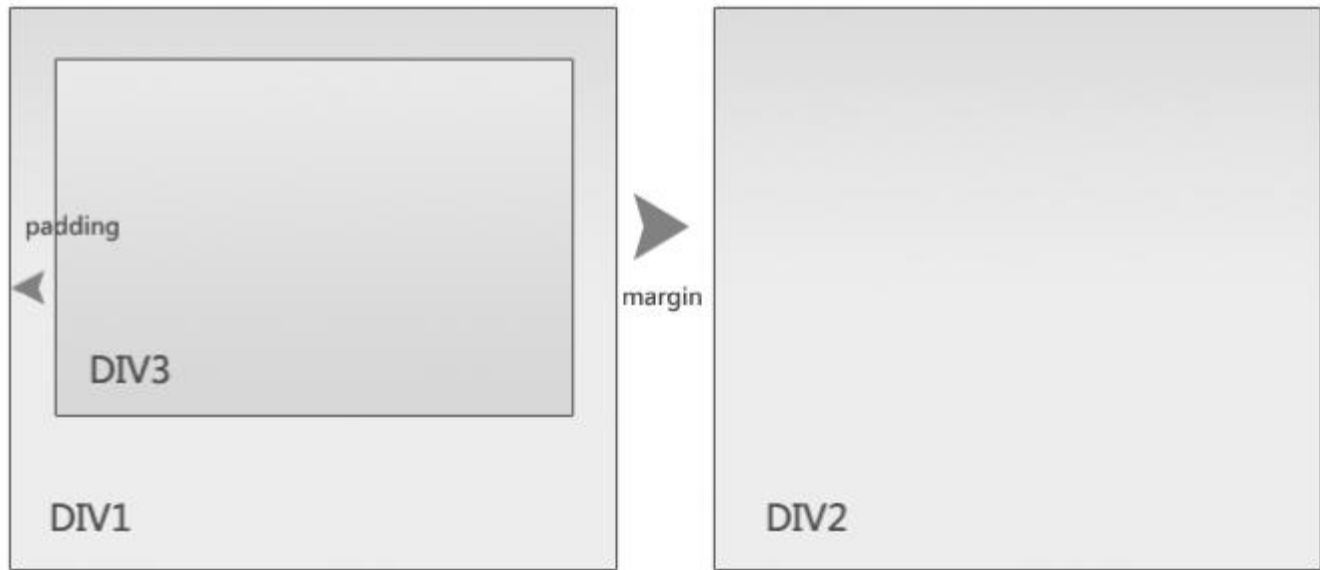


图 12-4 边距属性和间隙属性的区别

边距属性（**margin**）通常是设置一个页面中一个元素所占的空间的边缘到相邻的元素之间的距离，而间隙属性（**padding**）通常是设置一个元素中间的内容（或元素）到父元素之间的间隙（或距离）。对于边距属性（**margin**）有以下属性：

- ❑ 左边距属性（**margin-left**）：该属性用来设定左边距的宽度。
- ❑ 右边距属性（**margin-right**）：该属性用来设定右边距的宽度。
- ❑ 上边距属性（**margin-top**）：该属性用来设定上边距的宽度。
- ❑ 下边距属性（**margin-bottom**）：该属性用来设定下边距的宽度。
- ❑ 边距属性（**margin**）：该属性是设定边距宽度的一个快捷的综合写法，用该属性可以同时设定上下左右边距属性。

对于间隙属性，基本同边距属性，只是 **margin** 改为了 **padding**，其属性如下所示。

- ❑ 左间隙属性（**padding-left**）：该属性用来设定左间隙的宽度。
- ❑ 右间隙属性（**padding-right**）：该属性用来设定右间隙的宽度。
- ❑ 上间隙属性（**padding-top**）：该属性用来设定上间隙的宽度。示例如下：
- ❑ 下间隙属性（**margin-bottom**）：该属性用来设定下间隙的宽度。示例如下：
- ❑ 间隙属性（**padding**）：该属性是设定间隙宽度的一个快捷的综合写法，用该属性可以同时设定上下左右间隙属性。

使用边距属性和间隙属性能够进行页面布局，其中 **HTML** 页面代码如下所示。

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta content="text/html; charset=utf-8" http-equiv="Content-Type" />
  <title>这是一段文字 1</title>
  <link href="css.css" type="text/css" rel="stylesheet"></link>
</head>
<body>
  <div class="div1">
    DIV1
    <div class="div3">DIV3</div>
  </div>
  <div class="div2">DIV2</div>
</body>
</html>
```

**HTML** 代码制作完毕后，就可通过 **css.css** 文件为该页面编写样式，示例代码如下所示。

```
.div1
{
  float:left;
  margin-left:10px;                                     //和左边元素距离为 10px
  background:white url('bg.jpg') repeat-x;
```

```
border:1px solid #ccc;
width:300px;
height:200px;
padding:30px;                                //内部对齐 30px
}
.div2
{
    float:left;
    margin-left:20px;                        //和左边元素距离为 20px
    background:white url('bg.jpg') repeat-x;
    border:1px solid #ccc;
    width:300px;
    height:260px;
}
.div3
{
    background:white;                        //背景为白色
}
```

页面布局完成后，运行图如图 12-5 所示。

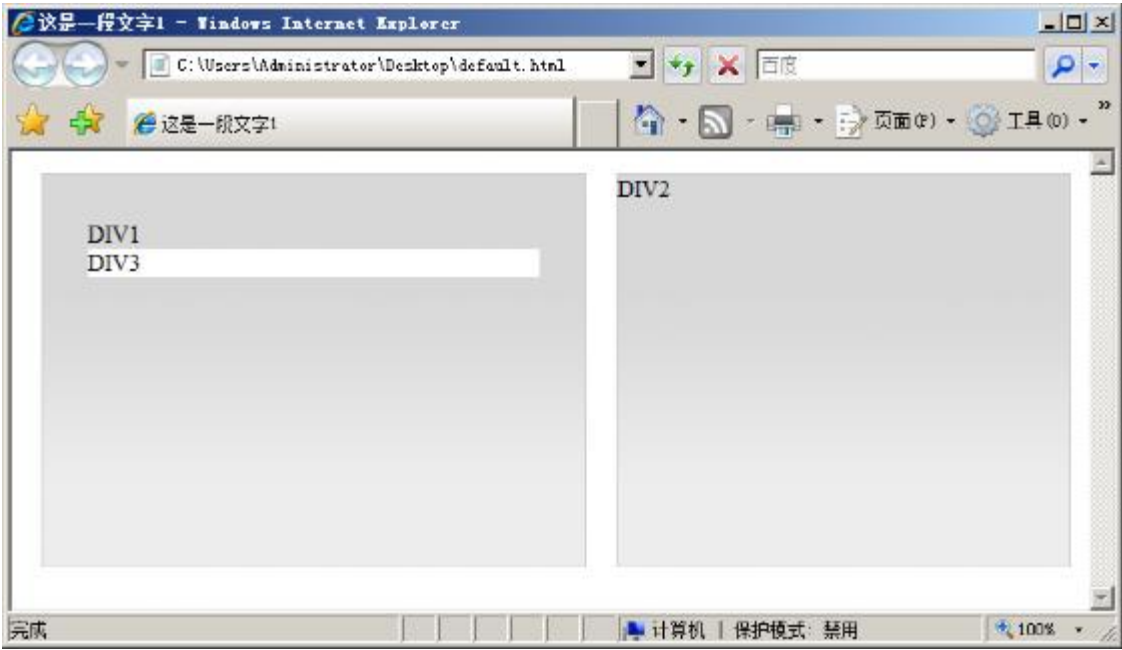


图 12-5 边距属性和间隙属性

CSS 不仅提供了诸如此类的强大布局功能，CSS 还提供了很多其他的布局功能，这些功能非常的多，能够为页面布局起到美化作用。CSS 还包括盒子模式、列表属性和伪类等高级技巧，这些技巧就不在本书中一一介绍了。

12.1.4 将 CSS 应用在控件上

CSS 不仅能够用来进行页面布局，同样也可以应用在控件中，使用 CSS 能够让控件更具美感。在空间上使用 CSS 基本和在页面上使用 CSS 的方法相同。在控件界面的编写中，可以使用控件的默认属性，例如 BackColor、ForeColor、BorderStyle 等，同样也可以通过 style 属性编写控件的属性，示例代码如下所示。

```
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>无标题页</title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
```



```
<asp:Button ID="Button1" runat="server" Text="Button" />
<br />
<br />
<asp:TextBox ID="TextBox2" runat="server" style="border:1px solid #ccc;"></asp:TextBox>
<asp:Button ID="Button2" runat="server" Text="Button"
style="border:1px solid #ccc; background:white; color:Black"/>
</div>
</form>
</body>
</html>
```

上述代码分别编写了 4 个控件，其中 2 个控件是输入框和按钮控件，另外 2 个控件也是输入框和按钮控件。不同的是，另外 2 个控件通过 **style** 属性进行了样式控制，运行后如图 12-6 所示。



图 12-6 控件的样式控制

除了通过 **style** 标签以外，控件自己还带有“样式”属性，通过配置相应的属性，即可为控件进行样式控制。其中最典型的包括日历控件，日历控件能够套用默认格式以呈现更加丰富的样式，示例代码如下所示。

```
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
  <title>无标题页</title>
  <style type="text/css">
    .style1
    {
      width: 100%;
    }
  </style>
</head>
<body>
  <form id="form1" runat="server">
    <div>
      <table class="style1">
        <tr>
          <td>
            默认样式</td>
          <td>
            选择样式</td>
        </tr>
        <tr>
          <td>
            <asp:Calendar ID="Calendar1" runat="server"></asp:Calendar>
          </td>
          <td>
            <asp:Calendar ID="Calendar2" runat="server"
            BackColor="#FFFFCC" BorderColor="#FFCC66"
            BorderWidth="1px" DayNameFormat="Shortest" Font-Names="Verdana" Font-Size="8pt"

```

```
ForeColor="#663399" Height="200px" ShowGridLines="True" Width="220px">
<SelectedDayStyle BackColor="#CCCCFF" Font-Bold="True" />
<SelectorStyle BackColor="#FFCC66" />
<TodayDayStyle BackColor="#FFCC66" ForeColor="White" />
    <OtherMonthDayStyle ForeColor="#CC9966" />
        <NextPrevStyle Font-Size="9pt" ForeColor="#FFFFCC" />
    <DayHeaderStyle BackColor="#FFCC66" Font-Bold="True" Height="1px" />
    <TitleStyle BackColor="#990000" Font-Bold="True" Font-Size="9pt"
        ForeColor="#FFFFCC" />
</asp:Calendar>
</td>
</tr>
</table>
</div>
</form>
</body>
</html>
```

上述代码通过属性为日历控件进行了样式控制，运行后如图 12-7 所示，默认的样式和配置了样式之后的差别巨大。

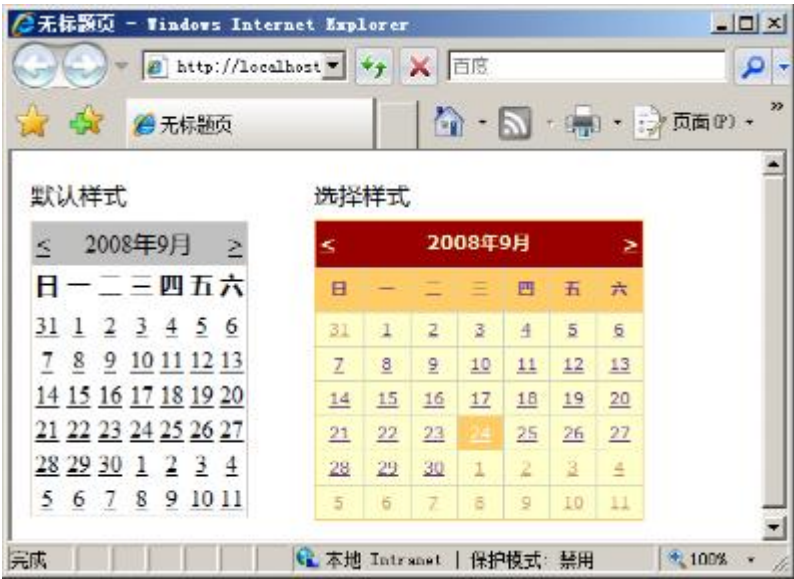


图 12-7 属性样式控制

通过编写样式能够让控件的呈现更加的丰富，让用户体验更加友好，当用户访问页面时，能够提高用户对网站的满意程度。控件的样式控制，不仅能够使用默认的属性进行样式控制，同样可以使用 **style** 属性进行样式控制，但是 **style** 属性的样式控制在很多地方不能操作，例如日历控件中的当前日期样式，而通过控件的属性的配置，却能够快速配置当前日期的样式。

12.1.5 主题和皮肤

主题是属性设置的集合，通过使用主题的设置能够定义页面和控件的样式，然后在某个 **Web** 应用程序中应用到所有的页面，以及页面上的控件，以简化样式控制。主题包括一系列元素，这些元素分别是皮肤、级联样式表（**CSS**）、图像和其他资源。主题文件的后缀名称为 **.skin**，创建主题后，主题文件通常保存在 **Web** 应用程序的特殊目录下，创建主题文件如图 12-8 所示。创建外观文件，**Visual Studio** 会提示是否将文件存放到特殊目录，如图 12-9 所示。



图 12-8 创建外观文件



图 12-9 将主题文件存放到特殊目录

单击【是】按钮后主题文件会存放到 **App\_Themes** 文件夹中。主题文件通常都保存在 **Web** 应用程序的特殊目录下，以便这些文件能够在页面中进行全局访问。在主题文件中编写代码可以对控件进行主题配置，示例代码如下所示。

```
<asp:Calendar runat="server" BackColor="White"
    BorderColor="Black" BorderStyle="Solid" CellSpacing="1" Font-Names="Verdana"
    Font-Size="9pt" ForeColor="Black" Height="250px" NextPrevFormat="ShortMonth"
    SkinID="blue" Width="330px">
    <SelectedDayStyle BackColor="#333399" ForeColor="White" />
    <TodayDayStyle BackColor="#999999" ForeColor="White" />
    <OtherMonthDayStyle ForeColor="#999999" />
    <DayStyle BackColor="#CCCCCC" />
    <NextPrevStyle Font-Bold="True" Font-Size="8pt" ForeColor="White" />
    <DayHeaderStyle Font-Bold="True" Font-Size="8pt" ForeColor="#333333"
    Height="8pt" />
    <TitleStyle BackColor="#333399" BorderStyle="Solid" Font-Bold="True"
    Font-Size="12pt" ForeColor="White" Height="12pt" />
</asp:Calendar>
<asp:Calendar runat="server" BackColor="White"
    BorderColor="#999999" CellPadding="4" DayNameFormat="Shortest"
    Font-Names="Verdana" Font-Size="8pt" ForeColor="Black" Height="180px"
    SkinID="now" Width="200px">
    <SelectedDayStyle BackColor="#666666" Font-Bold="True" ForeColor="White" />
    <SelectorStyle BackColor="#CCCCCC" />
    <WeekendDayStyle BackColor="#FFFFCC" />
    <TodayDayStyle BackColor="#CCCCCC" ForeColor="Black" />
    <OtherMonthDayStyle ForeColor="#808080" />
    <NextPrevStyle VerticalAlign="Bottom" />
    <DayHeaderStyle BackColor="#CCCCCC" Font-Bold="True" Font-Size="7pt" />
    <TitleStyle BackColor="#999999" BorderColor="Black" Font-Bold="True" />
</asp:Calendar>
```

上述代码创建了两种日历控件的主题，这两个日历控件的主题分别为 **SkinID=“blue”**和 **SkinID=“now”**。值得注意的是，**SkinID** 属性在主题文件中是必须且惟一的，因为这样才可以在相应页面中为控件配置所需要使用的主题，示例代码如下所示。

```
<asp:Calendar ID="Calendar1" runat="server" SkinID="blue"></asp:Calendar>
<asp:Calendar ID="Calendar2" runat="server" SkinID="now"></asp:Calendar>
```

上述控件并没有对控件进行样式控制，只是声明了 **SkinID** 属性，当声明了 **SkinID** 属性后，系统会自动在主题文件中找到相匹配的 **SkinID**，并将主题样式应用到当前控件。在使用主题的页面，必须声明主题，如果不声明主题，则页面无法找到页面中控件需要使用的主题，示例代码如下所示。

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Default.aspx.cs" Inherits="_12_1._Default"
Theme="Theme1"%>
```

在页面声明主题后，控件就能够使用 **.skin** 文件中的主题，通过 **SkinID** 属性，控件可以选择主题文件中

的主题。运行如图 12-10 所示。



图 12-10 选择不同的主题

主题还可以包括级联样式表（CSS 文件），将.css 放置在主题目录中，样式表则会自动的应用为主题的一部分，不仅如此，主题还可以包括图片和其他资源。

12.1.6 页面主题和全局主题

用户可以为每个页面设置主题，这种情况被称为“页面主题”。也可以为应用程序的每个页面都使用主题，在每个页面使用默认主题，这种情况被称为“全局主题”。

页面主题是一个主题文件夹，其中包括控件的主题、层叠样式表、图形文件和其他资源文件，这个文件夹是作为网站中的“\App\_Themes”文件夹和子文件夹创建的。每个主题都是“\App\_Themes”文件夹的一个子文件夹，如图 12-11 所示。

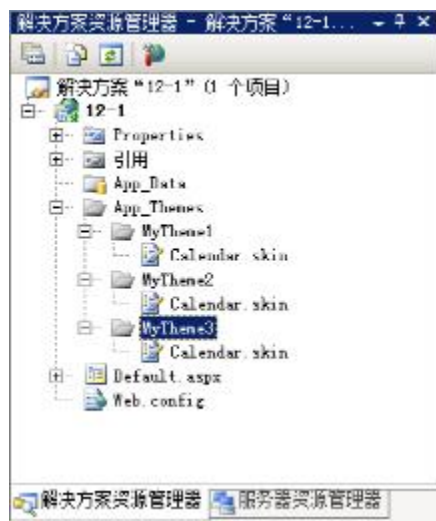


图 12-11 多个主题

使用全局主题，可以让应用程序中的所有页面都能够使用该主题，当维护同一个服务器上的多个网站时，可以使用全局主题定义应用程序的整体外观。当需要使用全局主题时，则可以通过修改 Web.config 配置文件中的<pages>配置节进行主题的全局设定。

使用全局主题和使用页面主题的方法基本相同，它们都包括属性设置、样式设置和图形。但是全局主题和页面主题不同的是，全局主题存放在服务器上的公共文件夹中，这个文件夹通常命名为 Theme。服务器上的任何 Web 应用程序都能够使用 Theme 文件夹中的主题。主题能够和 CSS 文件一样，进行页面布局 and 控件样式控制，但是主题和 CSS 文件的描述不同，所能够完成的功能也不同，其主要区别如下所示：

- ❑ 主题可以定义控件的样式，不仅能够定义样式属性，还能够定义其他样式，包括模板。
- ❑ 主题可以包括图形等其他主题元素文件。



- ❑ 主题的层叠方式与 CSS 文件的层叠方式不同。
- ❑ 一个页面只能应用与一个主题，而 CSS 可以被多个文件应用。

主题不仅能够进行控件的样式定义，还能够定义模板，这样减少了相同类型的控件的模板编写操作。但是主题也有缺点，一个页面只能应用一个主题，而无法应用多个主题。与之相反的是，一个页面能够应用多个 CSS 文件。

主题与 CSS 相比，主题在样式控制上还有很多不够强大的地方，而 CSS 页面布局的能力比主题更加强大，样式控制更加友好。

### 12.1.7 应用和禁用主题

通常情况下，可以在网站目录下的“App\_Themes”文件夹下定义主题，然后在页面中进行主题的使用声明，这样在页面中就能够使用主题了。制作主题的过程也非常简单，在“App\_Themes”文件夹下新建一个文件夹，则这个文件夹的名称就会作为主题名称在应用程序中保存。同样，开发人员能够在文件夹中可以新增“.skin”文件，以及“.css”文件和图形图像文件来修饰主题，这样一个主题就制作完毕并能够在页面中使用了。

在很多情况下，在 Web 开发中需要定义全局主题，这样 Web 应用程序就能够使用这个主题，全局主题通常放在一个特殊的目录下，放在这个目录下的主题能够被服务器上的任何网站，以及网站中的任何应用所引用。全局主题存放的目录如下所示。

```
isdefaultroot\aspnet_client\system_web\version\Themes
```

在全局主题目录下，可以创建任何主题文件，这样在网站上的其他 Web 应用也能够使用全局主题作为主题。在主题的编写过程中，通常需要以下几个步骤：

- ❑ 添加项目，包括.skin、css 以及其他文件。
- ❑ 创建皮肤，包括对控件属性的定义。
- ❑ 在页面中声明并使用皮肤。

通过以上三个步骤能够创建并使用皮肤，但是值得注意的是，在创建皮肤文件时，必须保存为.skin 文件并且主题中控件的定义必须包括 SkinID 属性且不能包括 ID。在皮肤中，对控件的属性的描述同样必须要包括 runat="server" 标记，这样才能够保证皮肤文件中控件的皮肤的正确和可读的，示例代码如下所示。

```
<asp:Calendar runat="server" BackColor="White"
    BorderColor="Black" BorderStyle="Solid" CellSpacing="1" Font-Names="Verdana"
    Font-Size="9pt" ForeColor="Black" Height="250px" NextPrevFormat="ShortMonth"
    SkinID="blue" Width="330px">
    <SelectedDayStyle BackColor="#333399" ForeColor="White" />
    <TodayDayStyle BackColor="#999999" ForeColor="White" />
    <OtherMonthDayStyle ForeColor="#999999" />
    <DayStyle BackColor="#CCCCCC" />
    <NextPrevStyle Font-Bold="True" Font-Size="8pt" ForeColor="White" />
    <DayHeaderStyle Font-Bold="True" Font-Size="8pt" ForeColor="#333333"
    Height="8pt" />
    <TitleStyle BackColor="#333399" BorderStyle="Solid" Font-Bold="True"
    Font-Size="12pt" ForeColor="White" Height="12pt" />
</asp:Calendar>
```

在定义了控件的皮肤后，就可以在单个页面进行皮肤的声明和使用，示例代码如下所示。

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Default.aspx.cs" Inherits="_12_1._Default"
Theme="MyTheme1"%>
```

同样也可以使用 StyleSheetTheme 属性进行页面主题的设置，示例代码如下所示。

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Default.aspx.cs" Inherits="_12_1._Default"
StylesheetTheme="MyTheme1"%>
```

如果需要使用全局主题，则需要在 Web.config 配置文件中定义全局主题，示例代码如下所示。

```
<system.web>
```

```
<pages theme="MyTheme1">
</pages>
</system.web>
```

在使用主题后，对于控件的属性的编写是没有任何效果的，示例代码如下所示。

```
<asp:Calendar ID="Calendar1" runat="server" SkinID="blue" BackColor="#FFFFCC"
    BorderColor="#FFCC66" BorderWidth="1px" DayNameFormat="Shortest"
    Font-Names="Verdana" Font-Size="8pt" ForeColor="#663399" Height="200px"
    ShowGridLines="True" Width="220px">
    <SelectedDayStyle BackColor="#CCCCFF" Font-Bold="True" />
    <SelectorStyle BackColor="#FFCC66" />
    <TodayDayStyle BackColor="#FFCC66" ForeColor="White" />
    <OtherMonthDayStyle ForeColor="#CC9966" />
    <NextPrevStyle Font-Size="9pt" ForeColor="#FFFFCC" />
    <DayHeaderStyle BackColor="#FFCC66" Font-Bold="True" Height="1px" />
    <TitleStyle BackColor="#990000" Font-Bold="True" Font-Size="9pt"
        ForeColor="#FFFFCC" />
</asp:Calendar>
```

上述代码编写了一个控件的属性，其中某些属性被主题覆盖。简单的说，局部的设置将会服从全局的设置，即页面上的控件已经具备自己的属性设置，但是当指定了 **SkinID** 属性后，部分属性将会服从全局属性设置，如图 12-12 和图 12-13 所示。

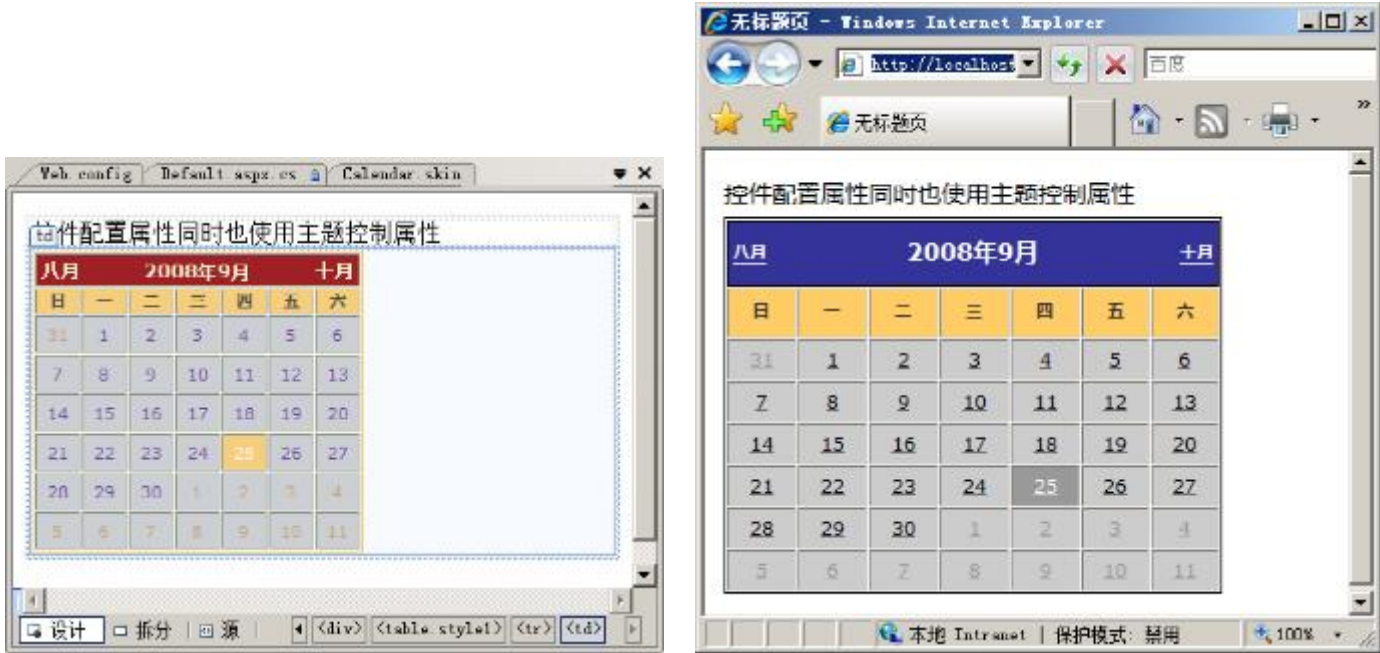


图 12-12 本地属性和全局属性



图 12-13 运行后的控件样式

虽然本地属性设置为另一种样式，但是运行后的控件样式却不是本地属性配置的样式，因为其中的某些属性已经被主题更改。在设置页面或者全局主题的 **StyleSheetTheme** 属性时，将主题作为样式表主题应用的话，本地页的设置将优先于主题中定义的设置，即局部设置将会覆盖全局设置。

对于主题而言，如果本地主题，以及全局主题都存在时，这种情况如控件本身的属性和使用的主题属性都存在一样，本身的属性将会被全局属性更改，全局属性中没有的属性将继续保留。而相对与 **CSS** 文件而言，如果本地 **CSS** 描述和全局 **CSS** 描述都存在，包括控件本身的 **CSS** 描述和内嵌式 **CSS** 文件的描述都一样时，相反的，本地 **CSS** 描述会替代全局的 **CSS** 描述。

对于有些情况，主题会重写也和控件外观的本地设置。当控件或页面已经定义了外观，而又不希望全局主题将本地主题进行重写和覆盖，可以禁用主题的覆盖行为。对于页面，可以用声明的方法进行禁用，示例代码如下所示。

```
<%@ Page Language="C#" AutoEventWireup="true" EnableTheming="false" %>

当页面需要某个主题的属性描述，而又希望单个控件不被主题描述时，同样可以通过 EnableTheming
属性进行主题禁止，示例代码如下所示。

<asp:Calendar ID="Calendar3" runat="server" EnableTheming="False">
</asp:Calendar>
```

这样就可以保证该控件不会被主题描述和控制，而页面和页面的其他元素可以使用主题描述中的相应

的属性。

### 12.1.8 用编程的方法控制主题

当主题被制作完成后，很多场合用户希望能够自行更改主题，这种方式非常的实用，通过编程手段，只需要更改 **StyleSheetTheme** 属性就能够对页面的主题进行更改。通过编程的方法不仅能够更改页面的主题，同样可以更改控件的主题，达到动态更改控件主题的效果。当需要更改页面的主题时，可以更改页面的 **StyleSheetTheme** 属性即可实现页面主题更改的效果，**StyleSheetTheme** 属性的更改代码只能编写在 **PreInit** 事件中，示例代码如下所示。

```
protected void Page_PreInit(object sender, EventArgs e)
{
    switch (Request.QueryString["theme"])           //获取传递的参数
    {
        case "MyTheme1":                             //判断主题
            Page.Theme = "MyTheme1"; break;           //更改主题
        case "MyTheme2":                             //判断主题
            Page.Theme = "MyTheme2"; break;           //更改主题
    }
}
```

上述代码则通过更改 **Page** 的 **StyleSheetTheme** 属性对页面的主题进行更改，在编程的过程中，同样可以使用更加复杂的编程方法实现主题的更改。在更改页面的代码中，必须首先重写 **StyleSheetTheme** 属性，然后通过其中的 **get** 访问器返回样式表的主题名称，示例代码如下所示。

```
public override String StyleSheetTheme
{
    get                                             //获取主题
    {
        return "MyTheme1";                       //返回主题名称
    }
}
```

对于控件，可以通过更改控件的 **SkinID** 属性来对控件的主题进行更改，示例代码如下所示。

```
protected void Page_PreInit(object sender, EventArgs e)
{
    Calendar3.SkinID = "blue";                   //更改 SkinID 属性
}
```

上述代码通过修改控件的 **SkinID** 属性修改控件的主题，在控件中，**SkinID** 属性是能够将控件与主题进行联系的关键属性。

## 12.2 母版页

在 **Web** 应用开发过程中，经常会遇到 **Web** 应用程序中的很多页面的布局都相同这种情况。在 **ASP.NET** 中，可以使用 **CSS** 和主题减少多页面的布局问题，但是 **CSS** 和主题在很多情况下还无法胜任多页面的开发，这时就需要使用母版页。

### 12.2.1 母版页基础

开发人员能够使用母版页定义某一组页面的呈现样式，甚至能够定义整个网站的页面的呈现样式，**Visual Studio 2008** 能够轻松的创建母版页文件，对网站的全部或部分页面进行样式控制。单击【添加项】选项，选择【母版页】项目，即可向项目中添加一个母版页，如图 12-14 所示。





图 12-14 添加母版页

母版页的后缀名为**.master**。母版页同 **Web** 窗体在结构上基本相同，与 **Web** 窗体不同的是，母版页的声明方法不是使用 **Page** 的方法声明，而是使用 **Master** 关键字进行声明，示例代码如下所示。

```
<%@ Master Language="C#"
AutoEventWireup="true" CodeBehind="MyMaster.master.cs" Inherits="_12_2.MyMaster" %>
```

母版页的结构基本同 **Web** 窗体，但是母版页通常情况下是用来进行页面布局。当 **Web** 应用程序中的很多页面的布局都相同，甚至中间需要使用的用户控件、自定义控件、样式表都相同时，则可以在一个母版页中定义和编码，对一组页面进行样式控制。编写母版页的方法非常简单，只需要像编写 **HTML** 页面一样就可以编写母版页。在编写网站页面时，首先需要确定通用的结构，并且确定需要使用控件或 **CSS** 页面，如图 12-15 所示。



图 12-15 母版页页面布局

在确定了母版页布局的通用结构后，就可以编写母版页的结构了。这里使用 **Table** 进行布局，在布局前，首选需要定义若干样式，示例代码如下所示。

```
<style type="text/css">
body
{
font-size:12px;
text-align:center;
}
.style1
{
width: 100%;
height: 129px;
}
```



```
.style2
{
    background:url('images/bg.jpg') repeat-x;
    height: 111px;
    text-align: center;
    font-size:18px;
    font-weight:bolder;
}
.style3
{
background:url('images/bg.jpg') repeat-x;
    height: 94px;
}
.style4
{
background:url('images/bg2.jpg') repeat-x;
    width: 129px;
}
.style5
{
background:url('images/bg2.jpg') repeat-x;
    width: 476px;
}
.style6
{
background:url('images/bg2.jpg') repeat-x;
}
</style>
</head>
```

这些样式规定了一些基本样式，用来 **Table** 以及页面的布局，整页布局代码如下所示。

```
<body>
    <form id="form1" runat="server">
    <div>
        <table class="style1">
            <tr>
                <td class="style2">
                    标题</td>
            </tr>
            <tr>
                <td>
                    <table class="style1">
                        <tr>
                            <td class="style4">
                                左侧</td>
                            <td class="style5">
                                中间</td>
                            <td class="style6">
                                右侧</td>
                        </tr>
                    </table>
                </td>
            </tr>
            <tr>
                <td class="style3">
                    底部说明
                </td>
            </tr>
        </table>
```

```
</table>
</div>
</form>
</body>
```

上述代码对页面进行了布局，并定位了头部、中部和底部三个部分，而中部又分为左侧、中间和右侧三个部分，布局完成后效果如图 12-16 所示。



图 12-16 母版页最终布局效果

通过编写 **HTML**，就能够进行母版页的布局，不仅如此，母版页还能够嵌入控件、用户控件和自定义控件，方便母版页中通用模块的编写。母版页提供一个对象模型，其他页面能够通过母版页快速的进行样式控制和布局，使用母版页具有以下好处。

- ☐ 母版页可以集中的处理页面的通用功能，包括布局 and 控件定义。
- ☐ 使用母版页可以定义通用性的功能，包括页面中某些模块的定义，这些模块通常由用户控件和自定义控件实现。
- ☐ 母版页允许控制占位符控件的呈现方式。
- ☐ 母版页能够为其他页面提供对项目型，其他页面能够使用母版页进行二次开发。

母版页能够将页面布局集中到一个或若干个页面中，这样无需在其他页面中过多的关心页面布局。

12.2.2 内容窗体

使用母版页的页面被称作内容窗体（也称内容页）。内容窗体不是专门负责设计的页面，它们只需要关注一般页面的布局、事件以及窗体结构即可，所以内容窗体无需过多的考虑页面布局。当用户请求内容窗体时，内容窗体将与母版页合并并且将母版页的布局和内容窗体的布局组合在一起呈现到浏览器。

创建内容窗体的方法基本同 **Web** 窗体一样，在 **Visual Studio 2005** 中创建 **Web** 窗体时，必须勾选【选择母版页】选项，而在 **Visual Studio 2008** 中，有单独的内容页可以选择，如图 12-17 所示。单击【添加】按钮，系统会提示选择相应的母版页，选择相应的母版页后，单击【确定】按钮即可创建内容窗体，如图 12-18 所示。



图 12-17 创建 Web 内容窗体



图 12-18 选择母版页

选择母版页后，系统会自动将母版页和内容整合在一起，如图 12-19 所示。



图 12-19 使用母版页

在使用母版页之后，内容窗体不能够修改母版页中的内容，也无法向母版页中新增 **HTML** 标签，在编写母版页时，必须使用容器让相应的位置能够在内容页中被填充。例如图 12-16，按照其方法编写母版页，内容窗体不能够对其中的文字进行修改，也无法在母版页中插入文字。在编写母版页，如果需要在某一区域能够允许内容窗体能够新增内容，就必须使用 **ContentPlaceHolder** 控件作占位，在母版页中，其代码如下所示。

```
<asp:ContentPlaceHolder ID="ContentPlaceHolder1" runat="server">
</asp:ContentPlaceHolder>
```

在母版页中无需编辑此控件，当内容窗体使用了相应的母版页后，则能够通过编辑此控件并向此占位控件中添加内容或控件。单击 **ContentPlaceHolder** 控件，并单击 **Content** 任务，可在占位控件中增加控件或自定义内容，如图 12-20 所示。

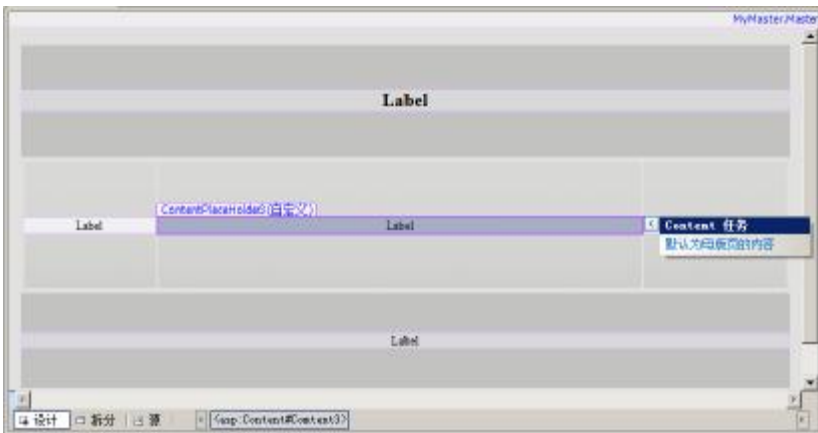


图 12-20 编辑内容窗体

编辑完成后，整个内容窗体就编写完毕了。内容窗体无需进行页面布局，也无法进行页面布局，否则会抛出异常。在内容窗体中，只需要按照母版页中的布局进行控件的拖放即可。

### 12.2.3 母版页的运行方法

在使用母版页时，母版页和内容页通常是一起协调运作的，母版页和内容也协调运作图如 12-21 所示。



图 12-21 母版页和内容窗体

在母版页运行后，内容窗体中 **ContentPlaceHolder** 控件会被映射到母版页的 **ContentPlaceHolder** 控件，并向母版页中的 **ContentPlaceHolder** 控件填充自定义控件。运行后，母版页和内容窗体将会整合形成结果页面，然后呈现给用户的浏览器。母版页运行的具体步骤为：

- ☐ 通过 **URL** 指令加载内容页面。
- ☐ 页面指令被处理。
- ☐ 将更新过内容的母版页合并到内容页面的控件树里。
- ☐ 单独的 **ContentPlaceHolder** 控件的内容被合并到相对的母版页中。
- ☐ 合并的页面被加载并显示给浏览器。

从浏览者的角度来说，母版页和内容窗体的运行并没有什么本质的区别，因为在运行的过程中，其 **URL** 是惟一的。而从开发人员的角度来说，实现的方法不同，母版页和内容窗体分别是单独而离散的页面，分别进行各自的工作，在运行后合并生成相应的结果页面呈现给用户。在内容页中使用，母版页无需存放在特殊的目录中，只需放在普通的目录文件中即可，内容页需要使用母版页时，只需要使用 **MasterPageFile** 属性即可，示例代码如下所示。

```
<%@ Page Language="C#"
MasterPageFile="~/MyMaster.Master"
AutoEventWireup="true" CodeBehind="Default.aspx.cs" Inherits="_12_2.Default" Title="无标题页" %>
```

使用 **MasterPageFile** 属性能够声明母版，**Page** 指令中的 **MasterPageFile** 属性会解析为一个 **.master** 页面，在运行时，就能够将母版页和内容窗体合并为一个 **Web** 窗体并呈现给浏览器。

### 12.2.4 嵌套母版页

母版页与母版页之间能够嵌套运行，让一个母版页作为另一个母版页的子母版，能够方便的将页面进行模块化。当编写 **Web** 应用时，可以使用母版页进行较大型的框架布局，对一个页面进行整体的样式控制。同样可是使用母版页进行嵌套，对细节的地方进行细分。

母版页的结构和 **Web** 窗体的结构十分相似，与任何母版页一样，母版页也可以包含母版页，被包含的母版页被称为子母版。子母版通常会包含一些控件，这些控件将映射到父母版上的内容占位符。在 **MyMaster** 页面中，可以编写相应的代码进行嵌套，示例代码如下所示。



```
<body>
  <form id="form1" runat="server">
    <div>
      <table class="style1">
        <tr>
          <td class="style2">
            <asp:ContentPlaceHolder ID="ContentPlaceHolder1" runat="server">
            </asp:ContentPlaceHolder>
          </td>
        </tr>
        <tr>
          <td>
            <table class="style1">
              <tr>
                <td class="style4">
                  <asp:ContentPlaceHolder ID="ContentPlaceHolder2" runat="server">
                  </asp:ContentPlaceHolder>
                </td>
                <td class="style5">
                  <asp:ContentPlaceHolder ID="ContentPlaceHolder3" runat="server">
                    Master 母版页:
                    <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
                  </asp:ContentPlaceHolder>
                </td>
                <td class="style6">
                  <asp:ContentPlaceHolder ID="ContentPlaceHolder4" runat="server">
                  </asp:ContentPlaceHolder>
                </td>
              </tr>
            </table>
          </td>
        </tr>
        <tr>
          <td class="style3">
            <asp:ContentPlaceHolder ID="ContentPlaceHolder5" runat="server">
            </asp:ContentPlaceHolder>
          </td>
        </tr>
      </table>
    </div>
  </form>
</body>
```

上述代码创建了 **MyMaster** 母版页，并使用了 **Content** 控件进行占位控件的编写。右击当前项目并单击【新建项】选项，创建一个 **Child.master** 母版页并为母版页编写相应的 **HTML** 代码，示例代码如下所示。

```
<%@ Master Language="C#" AutoEventWireup="true" CodeBehind="Child.master.cs" Inherits="_12_2.Child"
MasterPageFile="~/MyMaster.Master"%>
<asp:Content ID="ContentPlaceHolder4" ContentPlaceHolderID="ContentPlaceHolder4" runat="server">
子母版页:<asp:TextBox ID="TextBox2" runat="server"></asp:TextBox>
</asp:Content>
```

上述代码在子母版页中创建了一个文本框。在父母版页中则可以使用此母版页，使用方法同样也是使用 **MasterPageFile** 属性进行声明。在 **Child** 子母版中已经声明了 **MyMaster** 母版页，在使用和加载 **Child** 页面时，可以使用 **MasterPageFile='~/MyMaster.Master'** 语法对子母版页的母版进行声明，在上述代码中 **Child** 母版页使用了 **MyMaster** 母版页，并使用 **ContentPlaceHolderID="ContentPlaceHolder4"** 属性对该控件进行占位控件的填充，如图 12-22 所示。

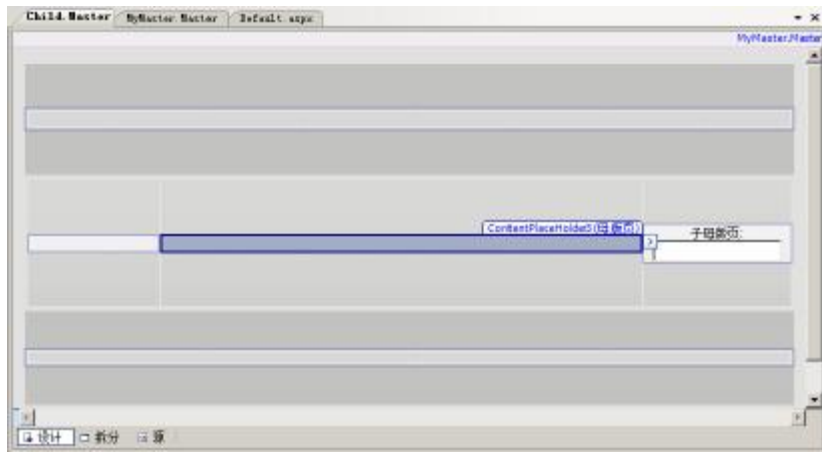


图 12-22 嵌套母版页

母版页嵌套完毕后，使用母版页的页面也应该进行相应的修改，在使用嵌套后，子母版页应该被声明到需要使用的页面，而不是母页面。简单的说，需要使用的页面应该声明的是子页面，而不是母母版页，在这里应该为 **Child.master**，示例代码如下所示。

```
<%@ Page Language="C#"
MasterPageFile="~/Child.Master"
AutoEventWireup="true" CodeBehind="Default.aspx.cs" Inherits="_12_2.Default" Title="无标题页" %>
```

上述代码声明了该页的母版页为 **Child.master**，运行结果如图 12-23 所示。

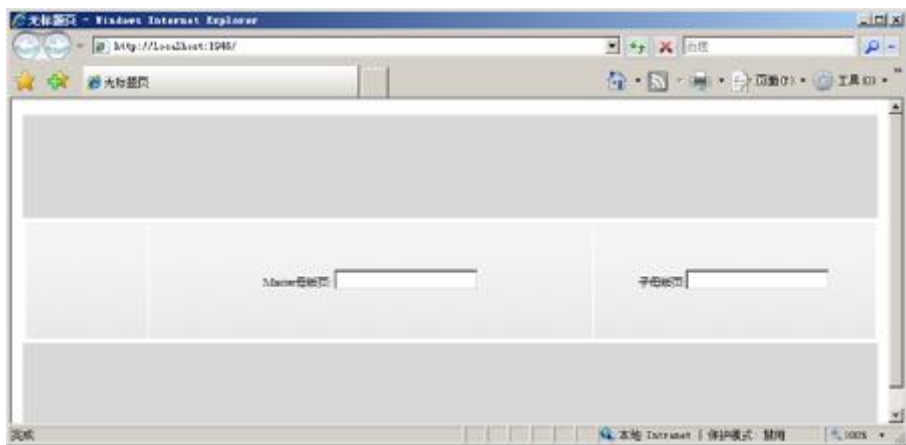


图 12-23 嵌套母版页

嵌套母版页之后，使用子母版页的页面将不能直接进行页面编辑，在 **Visual Studio 2008** 中，使用子母版页的页面将显示为空白，但并不表示页面显示将为空白。

## 12.3 Microsoft Expression 2

**Microsoft Expression 2** 是微软推出的一套专业的设计软件。**Microsoft Expression 2 Studio** 包括 **Expression Web 2**，**Expression Blend 2**，**Expression Design 2**，**Expression Media 2** 和 **Expression Encoder 2**，它们可以协调的同 **Visual Studio 2008** 一起协同合作，并支持 **Vista** 和 **Window Server 2008** 操作系统。

### 12.3.1 Microsoft Expression 2 简介

**Microsoft Expression 2** 是微软推出的强大的设计软件，不仅能够设计和进行网页布局，同时还支持 **XAML** 语言，能够进行 **Silverlight** 设计，**WPF** 设计，**Microsoft Expression 2 Studio** 包括的软件如下所示：

- ❑ **Microsoft Expression Web:** 网页设计工具，用于网页设计、页面布局。
- ❑ **Microsoft Expression Blend:** 交互设计工具，可以用于 **Silverlight**、**WPF** 的设计和开发。
- ❑ **Microsoft Expression Design:** 平面图形设计工具，可以对图像进行编辑和设计。
- ❑ **Microsoft Expression Media:** 多媒体编辑工具，可以对多媒体进行编辑、剪切和设计。

❑ **Expression Encoder:** 音频编辑工具，可以对音频进行编辑、剪切和设计。

**Microsoft Expression 2** 包含的软件为微软的产品开发和设计做出了强有力的保障，其中对于 **ASP.NET** 开发人员最常用的就包括 **Microsoft Expression Web**、**Microsoft Expression Blend** 和 **Microsoft Expression Design**。

**Microsoft Expression Web** 提供了对 **ASP.NET** 中控件的支持，这弥补了传统的 **Dreamware** 系列对 **ASP.NET** 的控件不支持，造成 **ASP.NET** 页面设计困难，**Microsoft Expression Web** 还提供了页面的调试环境，通过 **Microsoft Expression Web** 也能够进行基本的网页调试。在开源的影响力之下，**Microsoft Expression Web** 还支持 **php** 的脚本编写。

**Microsoft Expression Blend** 提供了对 **.NET** 中 **Silverlight**，以及 **WPF** 的设计和开发的支持。在 **Visual Studio 2005** 甚至是 **Visual Studio 2008** 中，**Silverlight** 以及 **WPF** 都不能很好的进行设计和可视化开发，因为 **Silverlight** 和 **WPF** 都是较新的技术，而 **Microsoft Expression Blend** 提供了对 **Silverlight** 和 **WPF** 的开发支持。

**Microsoft Expression Design** 用于平面设计，**Microsoft Expression Design** 不仅能够像传统的 **Photoshop** 一样设计和开发 **JPG**、**GIF** 格式的图片，也能够为 **Silverlight** 和 **WPF** 应用程序开发资源文件。这种资源文件可以是不规则的窗体，也可以是一段动画，**Microsoft Expression Design** 能够保存为资源文件所需要的文件类型。

**Microsoft Expression 2** 是微软推出的设计软件，在传统的开发过程中，虽然 **Visual Studio** 提供了可视化编程的解决方案，但是 **Visual Studio** 中可视化开发的效率依旧不高，对于计算机配置不是很高的用户更是如此。而另一方面，在开发过程中，开发小组很难将开发人员和设计人员完全的分离开，而使用 **Microsoft Expression 2**，可以使设计人员专注于设计，使开发人员专注于代码的编写。

## 12.3.2 安装 Microsoft Expression 2

**Microsoft Expression 2** 并不是免费的软件，但是开发人员可以在微软的官方主页上下载试用版本下载地址为 <http://www.microsoft.com/expression/try-it/default.aspx>，其中 **Expression® Studio 2** 包括了所有的软件包，如果无需其他软件包，可以选择单独的软件包进行下载。下载完成后，单击 **msi** 安装程序，即可安装 **Microsoft Expression 2**。系统会提取文件，当文件提取完毕后，即会加载进入安装界面，如图 12-24 所示。

等待安装程序初始化，安装程序会进入下一步，提示要求输入密钥并进行激活，安装程序会确定 **Expression Studio 2** 的安装状态，并继续安装。图 12-25 所示。



图 12-24 安装 Microsoft Expression 2



图 12-25 确定 Microsoft Expression 2 安装状态

按着安装程序的提示，基本上只需要单击下一步就能够将 **Microsoft Expression 2** 自行安装到本地计算机，**Microsoft Expression 2** 安装完毕后，就能够选择相应的应用程序做相应的开发。**Microsoft Expression 2** 的界面为黑色界面，看上去比较清新，但可能传统的用户很难适应这样的布局，如图 12-26 所示。



图 12-26 Microsoft Expression Web

## 12.4 使用 Microsoft Expression Web 2 制作页面

**Microsoft Expression Web 2** 是属于 **Microsoft Expression 2 Studio** 软件包中对 **ASP.NET** 开发人员来说最为强大的开发工具，**Microsoft Expression Web 2** 不仅提供了基本的网页布局功能，还支持 **ASP.NET** 中控件的拖动。

### 12.4.1 创建 ASPX 页面

通过 **Microsoft Expression Web 2** 能够快速的创建 **ASPX** 页面。在菜单栏中单击【文件】选项，单击【新建】按钮，可以选择创建相应的项目。**Microsoft Expression Web 2** 支持新建项目和新建网站，新建项目是为现有项目添加文件，也可以通过新建网站来新建另一个项目，在这里建立一个文件即可。单击【新建】按钮，系统会弹出对话框，用于创建新项目，如图 12-27 所示。

**Microsoft Expression Web 2** 不仅支持创建 **ASPX** 页面，也能够创建母版页、**XML**、动态 **Web** 模板，甚至能够支持创建 **PHP** 页面。这里可以选择一个 **ASPX** 页面进行创建，单击确定，创建一个 **ASPX** 页面。创建 **ASPX** 页面后，在 **Microsoft Expression Web 2** 的工具箱中，就可以看到 **Microsoft Expression Web 2** 为开发人员提供了 **HTML** 控件和 **ASPX** 控件，如图 12-28 所示。

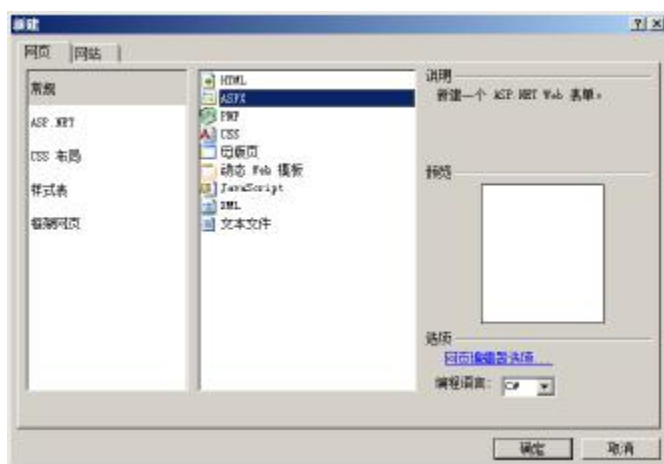


图 12-27 新建文件

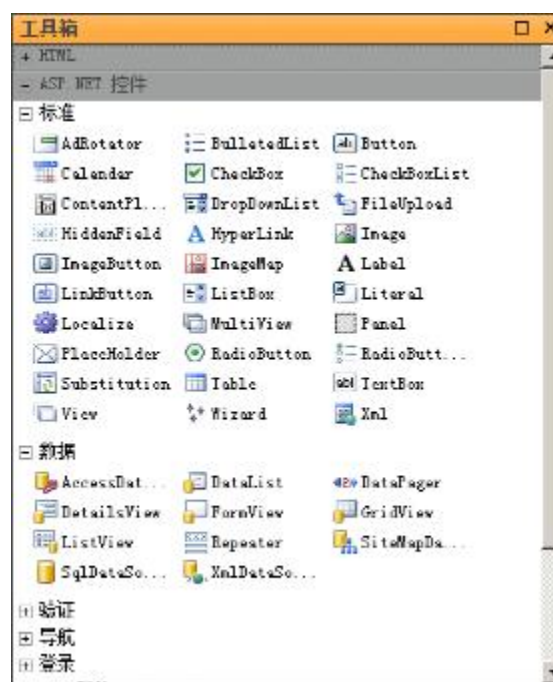


图 12-28 HTML 控件和 ASP.NET 控件

设计人员能够将页面布局进行设计，同时设计人员也能够拖动 **ASP.NET** 控件到页面布局中，这样就极大的方便了设计人员在前台界面的设计开发。而编程人员只需获取相应的页面，然后对页面进行逻辑代码的编写，即可组成一个完整的 **ASPX** 页面。**Microsoft Expression Web 2** 不仅能够支持设计人员对现有的页



面进行控件的拖放，还能够支持进行数据源配置和数据绑定，如图 12-29 所示。



图 12-29 配置数据源

在可视化开发中，**Microsoft Expression Web 2** 的效率比 **Visual Studio 2008** 较高，因为 **Microsoft Expression Web 2** 只负责页面布局，并负责配置相应的数据源和数据绑定，虽然 **Microsoft Expression Web 2** 不能负责页面逻辑的开发，但是对于 **ASP.NET** 页面设计的支持已经非常强大了。

12.4.2 创建 CSS 层叠样式表

**CSS** 层叠样式表是在网站设计和开发中必不可少的，通过 **Microsoft Expression Web 2** 同样能够创建和使用 **CSS** 层叠样式表，在菜单栏中找到并单击【文件】选项，在下拉菜单中单击【新建】选项，在弹出对话框中选择【**CSS**】选项，单击【确定】按钮就能够创建 **CSS** 层叠样式表。**CSS** 文件能够对现有的页面进行样式控制，开发人员可以在 **CSS** 层叠样式表中编写样式控制代码，示例代码如下所示。

```
body
{
    background:silver;
    font-size:12px;
}
.div1
{
    background-color:white;
    padding:10px 10px 10px;
    font-size:16px;
    font-weight:bolder;
}
```

上述代码编写了 **body** 的样式，以及 **class="div1"** 的样式。在相应的页面则能够通过此样式表对页面中的标签进行样式控制。在 **Microsoft Expression Web 2** 中，可以智能的添加 **CSS** 层叠样式表。单击菜单栏中的【格式】选项，选择【**CSS 样式**】，在下拉菜单中选择【附加样式表】选项则会弹出附加对话框，单击【浏览】按钮选择相应的 **css** 文件即可，如图 12-30 所示。

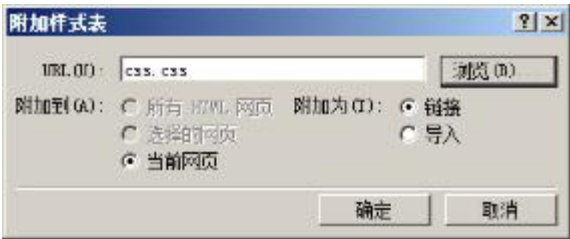


图 12-30 附加样式表

附加完成后，在页面的 **HTML** 代码中会自动增加 **CSS** 文件引用代码，示例代码如下所示。

```
<link href="css.css" rel="stylesheet" type="text/css" />
```

上述代码就为页面中声明的外联式 **CSS** 样式表，当声明了相应的外联式样式文件后，该页面就能够使用外联式 **CSS** 样式表提供的样式进行样式控制。

12.4.3 创建框架集

包含框架的页面被称为框架集。框架集是一个单独的文件，用于定义页面上所有框架的布局 and 属性，包括框架数量、框架的大小和位置，以及最初显示在每个框架中的页面的。框架集通常用于后台页面的开发，也用于帮助文档的开发，例如 MSDN 中对函数的查询，就是使用了框架集。

框架集是一个单独的页面，框架集是对页面中所有的框架的布局，每个框架都是另一个页面，框架集只是负责将页面组织并呈现在同一个页面中。单击【新建】按钮，在弹出窗口中选择框架集，如图 12-31 所示。

框架集能够预览，相应的框架集为不同的页面进行布局，这里创建一个目录类型的框架集，创建完成后，框架集页面会智能提示用户所需填充的页面，通过填充页面，能够填充相应的框架，如图 12-32 所示。

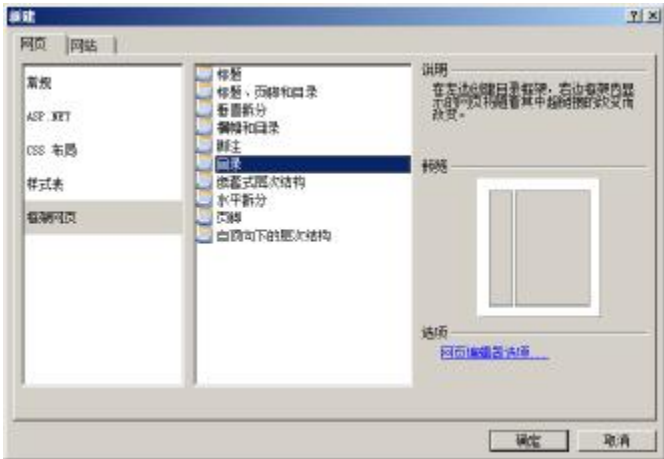


图 12-31 创建框架集



图 12-32 填充框架集

单击设置初始网页可以选择框架集所需要的网页，但是网页必须存在，若网页不存在，则可以单击新建网页填充框架集。左侧的框架可以用于导航，右侧的框架可以用于目录的显示，编写完成后如图 12-33 所示。

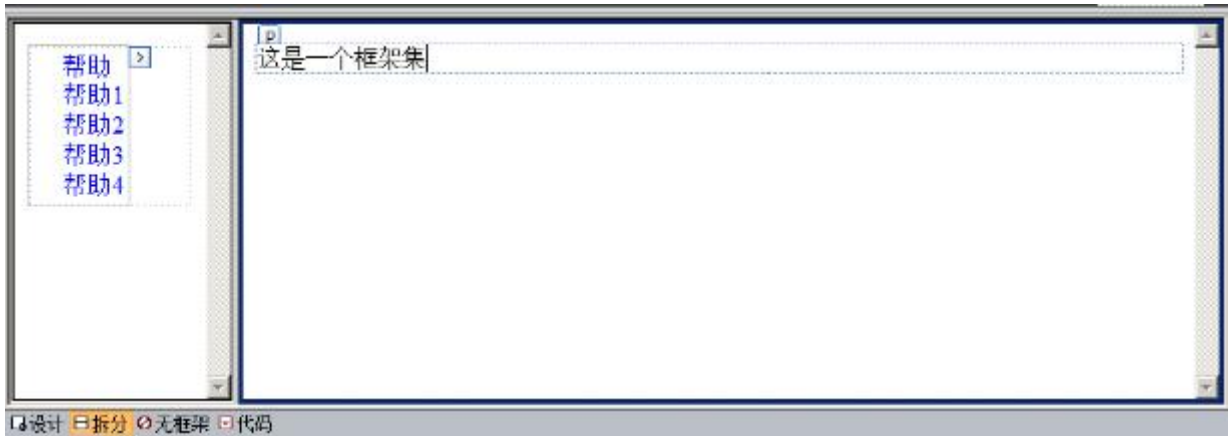


图 12-33 编写框架集

框架集创建后，框架集的页面 HTML 代码将自动生成，示例代码如下所示。

```
<html>
<head>
<meta content="text/html; charset=utf-8" http-equiv="Content-Type">
<title>无标题 1</title>
</head>
<frameset cols="150,*">
  <frame name="contents" src="无标题_3.aspx" target="main">
  <frame name="main" src="无标题_2.html">
  <noframes>
  <body>
  <p>此网页使用了框架，但您的浏览器不支持框架。</p>
  </body>
  </noframes>
```

```
</frameset>  
</html>
```

从上述代码可以看出，框架集就是将不同的页面呈现在同一页面的一种布局方法，上述代码中包括两个框架，这两个框架的页面分别为“无标题\_3.aspx”和“无标题\_2.html”。

注意：在使用框架集时，框架都包括 **name** 属性，这个属性在进行超链接时非常有用，通过编写超链接的 **target** 属性，能够指定相应的框架中的连接将能够对目标框架的页面进行更改。

## 12.5 小结

本章对 **CSS**，皮肤，主题做了详细的介绍，通过使用 **CSS**，能够优化网页代码布局，提高网页的友好度，增加用户粘度。同样，使用皮肤和主题能够控制控件的样式，并能够通过编程的方法动态的更改皮肤和主题，增强了代码的复用性。同时，本章还介绍了母版页，通过母版页能够将页面布局 and 控件进行分离，母版页只需对页面进行布局和样式控制，而内容窗体只需要镶嵌相应的控件即可。本章还包括：

- ☐ **CSS** 常用属性。
- ☐ 将 **CSS** 应用在控件上：在控件上使用 **CSS**。
- ☐ 应用和禁用主题：使用主题和禁用主题的方法。
- ☐ 母版页的运行方法：讲解了母版页是如何运行的。
- ☐ 嵌套母版页：讲解了如何进行母版页的嵌套。
- ☐ **Microsoft Expression 2** 简介。
- ☐ 使用 **Microsoft Expression Web 2** 制作页面。

本章还讲解了微软强大的设计工具——**Microsoft Expression 2**，通过 **Microsoft Expression 2** 能够为 **ASP.NET** 开发人员进行页面设计提供支持。

## 第 13 章 ASP.NET 内置对象, 应用程序配置和缓存

**Web** 应用程序在传统的意义上来说是无状态的, **Web** 应用不能像 **Win Form** 那样维持客户端状态, 所以在 **Web** 应用中, 通常需要使用内置对象进行客户端状态的保存。这些内置对象能够为 **Web** 应用程序的开发提供设置, 配置以及检索等功能。

### 13.1 ASP.NET 内置对象

在 **ASP** 的开发中, 这些内置对象已经存在, 这些内置对象包括 **Response**、**Request**、**Application** 等, 虽然 **ASP** 是一个可以称得上是“过时的”技术, 但是在 **ASP.NET** 开发人员中依旧可以使用这些对象。这些对象不仅能够获取页面传递的参数, 某些对象还可以保存用户的信息, 如 **Cookie**、**Session** 等。

#### 13.1.1 Request 传递请求对象

**Request** 对象是 **HttpRequest** 类的一个实例, **Request** 对象用于读取客户端在 **Web** 请求期间发送的 **HTTP** 值。**Request** 对象常用的属性如下所示。

- ❑ **QueryString**: 获取 **HTTP** 查询字符串变量的集合。
- ❑ **Path**: 获取当前请求的虚拟路径。
- ❑ **UserHostAddress**: 获取远程客户端 **IP** 主机的地址。
- ❑ **Browser**: 获取有关正在请求的客户端的浏览器功能的信息。

##### 1. QueryString: 请求参数

**QueryString** 属性是用来获取 **HTTP** 查询字符串变量的集合, 通过 **QueryString** 属性能够获取页面传递的参数。在超链接中, 往往需要从一个页面跳转到另外一个页面, 跳转的页面需要获取 **HTTP** 的值来进行相应的操作, 例如新闻页面的 **news.aspx?id=1**。为了获取传递过来的 **id** 的值, 则可以使用 **Request** 的 **QueryString** 属性, 示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!String.IsNullOrEmpty(Request.QueryString["id"]))           //如果传递的 ID 值不为空
    {
        Label1.Text = Request.QueryString["id"];                  //将传递的值赋予标签中
    }
    else
    {
        Label1.Text = "没有传递的值";                             //提示没有传递的值
    }
    if (!String.IsNullOrEmpty(Request.QueryString["type"]))        //如果传递的 TYPE 值不为空
    {
        Label2.Text = Request.QueryString["type"];               //获取传递的 TYPE 值
    }
    else
    {
        Label2.Text = "没有传递的值";                             //无值时进行相应的编码
    }
}
```



```
}
```

上述代码使用 **Request** 的 **QueryString** 属性来接受传递的 **HTTP** 的值，当通过访问页面路径为 “**http://localhost:29867/Default.aspx**” 时，默认传递的参数为空，因为其路径中没有对参数的访问。而当访问的页面路径为 “**http://localhost:29867/Default.aspx?id=1&type=QueryString&action=get**” 时，就可以从路径中看出该地址传递了三个参数，这三个参数和值分别为 **id=1**、**type=QueryString** 以及 **action=get**。

2. Path: 获取路径

通过使用 **Path** 的方法可以获取当前请求的虚拟路径，示例代码如下所示。

```
Label3.Text = Request.Path.ToString(); //获取请求路径
```

当在应用程序开发中使用 **Request.Path.ToString()** 时，就能够获取当前正在被请求的文件的虚拟路径的值，当需要对相应的文件进行操作时，可以使用 **Request.Path** 的信息进行判断。

3. UserHostAddress: 获取 IP 记录

通过使用 **UserHostAddress** 的方法，可以获取远程客户端 **IP** 主机的地址，示例代码如下所示。

```
Label4.Text = Request.UserHostAddress; //获取客户端 IP
```

在客户端主机 **IP** 统计和判断中，可以使用 **Request.UserHostAddress** 进行 **IP** 统计和判断。在有些系统中，需要对来访的 **IP** 进行筛选，使用 **Request.UserHostAddress** 就能够轻松的判断用户 **IP** 并进行筛选操作。

4. Browser: 获取浏览器信息

通过使用 **Browser** 的方法，可以判断正在浏览网站的客户端的浏览器的版本，以及浏览器的一些信息，示例代码如下所示。

```
Label5.Text = Request.Browser.Type.ToString(); //获取浏览器信息
```

这些属性能够获取服务器和客户端的相应信息，也可以通过 “?” 号进行 **HTTP** 的值的传递和获取，上述代码运行结果如图 13-1 所示。



图 13-1 Request 对象

**Request** 不仅包括这些常用的属性，还包括其他属性，例如用于获取当前目录在服务器虚拟主机中的绝对路径（如 **ApplicationPath**）。另外，开发人员也可使用 **Request** 中的 **Form** 属性进行页面中窗体的值集合的获取。

13.1.2 Response 请求响应对象

**Response** 对象是 **HttpResponse** 类的一个实例。**HttpResponse** 类用户封装页面操作的 **HTTP** 响应信息。**Response** 对象的常用属性如下所示。

- ❑ **BufferOutput**: 获取或设置一个值，该值指示是否缓冲输出，并在完成处理整个页面之后将其发送。
- ❑ **Cache**: 获取 **Web** 页面的缓存策略。
- ❑ **Charset**: 获取或设置输出流的 **HTTP** 字符集类型。

- ❑ **IsClientConnected**: 获取一个值，通过该值指示客户端是否仍连接在服务器上。
- ❑ **ContentEncoding**: 获取或设置输出流的 **HTTP** 字符集。
- ❑ **TrySkipListCustomErrors**: 获取或设置一个值，指定是否支持 **IIS 7.0** 自定义错误输出。

## 1. Response 常用属性

**BufferOutput** 的默认属性为 **True**。当页面被加载时，要输出到客户端的数据都暂时存储在服务器的缓冲期内并等待页面所有事件程序，以及所有的页面对象全部被浏览器解释完毕后，才将所有在缓冲区中的数据发送到客户端浏览器，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    Response.Write("缓冲区清除前..");           //输出缓冲区清除
}
```

上述代码在 **cs** 文件中重写了 **Page\_Load** 事件，该事件用于向浏览器输出一行字符串“缓冲区清除前”。在 **ASPX** 页面中，可以为页面增加代码以判断缓冲区的执行时间，示例代码如下所示。

```
<body>
    <form id="form1" runat="server">
        <div>
            <% Response.Write("缓冲区被清除"); %>           //输出字符串
        </div>
    </form>
</body>
```

上述代码在页面中插入了一段代码，并输出字符串“缓冲区被清除”。在运行该页面时，数据已经存放在缓冲区中。然后 **IIS** 才开始读取 **HTML** 组件的部分，读取完毕后才将结果送至客户端浏览器，所以在运行结果中可以发现，“缓冲期清除前”是在“缓冲区被清除”字符串之前出现，如图 13-2 所示。

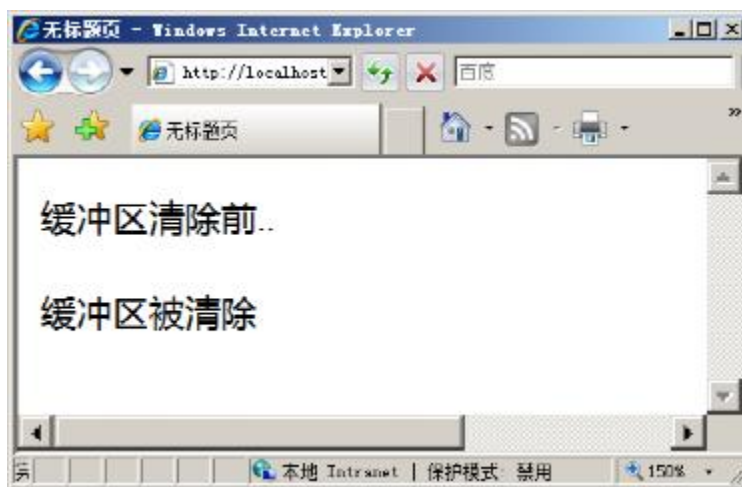


图 13-2 BufferOutput

因为 **BufferOutput** 属性默认为 **true**，所以上述代码并无法看到明显的区别，当在浏览器输出前清除缓冲区时，则可以看出区别。示例代码如下所示。

```
Response.Write("缓冲区清除前..");
Response.Clear();           //清除缓冲区
```

当使用 **Response** 的 **Clear** 方法时，缓冲区就被显式的清除了。在运行后，“缓冲区清除前”字符串被清除，并不会呈现给浏览器。当需要屏蔽 **Clear** 方法对缓冲区的数据清除，则可以指定 **BufferOutput** 的属性为 **False**，示例代码如下所示。

```
Response.BufferOutput = false;           //设置缓冲区属性
Response.Write("缓冲区清除前..");       //设置清除前字符
Response.Clear();                       //清除缓冲区
```

使用上述代码将指定 **BufferOutput** 的属性为 **False**，在运行时缓冲区数据不会被 **Clear** 方法清除。

## 2. Response 常用方法

**Response** 方法可以输出 **HTML** 流到客户端，其中包括发送信息到客户端和客户端 **URL** 重定向，不仅如此，**Response** 还可以设置 **Cookie** 的值以保存客户端信息。**Response** 的常用方法如下所示：

- ❑ **Write**: 向客户端发送指定的 **HTTP** 流。

- ❑ **End:** 停止页面的执行并输出相应的结果。
- ❑ **Clear:** 清除页面缓冲区中的数据。
- ❑ **Flush:** 将页面缓冲区中的数据立即显示。
- ❑ **Redirect:** 客户端浏览器的 **URL** 地址重定向。

在 **Response** 的常用方法中，**Write** 方法是最常用的方法，**Write** 能够向客户端发送指定的 **HTTP** 流，并呈现给客户端浏览器，示例代码如下所示。

```
Response.Write("<div style='font-size:18px;'>这是一串<span style='color:red'>HTML</span>流</div>");
```

上述代码则会向浏览器输出一串 **HTML** 流并被浏览器解析，如图 13-3 所示。



图 13-3 Response.Write 方法

当希望在 **Response** 对象运行时，能够中途进行停止时，则可以使用 **End** 方法对页面的执行过程进行停止，示例代码如下所示。

```
for (int i=0; i < 100; i++) //循环 100 次
{
    if (i < 10) //判断 i<10
    {
        Response.Write("当前输出了第" + i + "行<hr/>"); //i<10 则输出 i
    }
    else //否则停止输出
    {
        Response.End(); //使用了 End 方法停止执行
    }
}
```

上述代码循环输出 **HTML** 流“当前输出了第 **X** 行”，当输出到 **10** 行时，则停止输出，如图 13-4 所示。

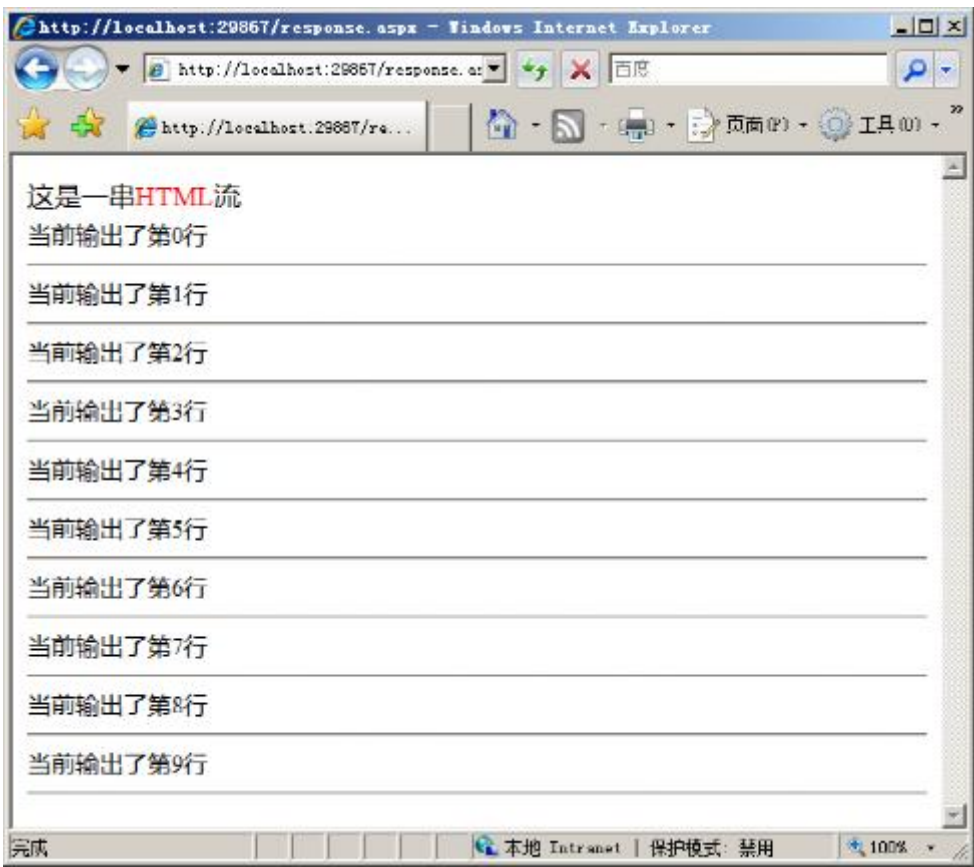


图 13-4 Response.End 方法

**Redirect** 方法通常使用于页面跳转，示例代码如下所示。

```
Response.Redirect("http://www.shangducms.com"); //页面跳转
```



执行上述代码，将会跳转到相应的 URL。

### 13.1.3 Application 状态对象

**Application** 对象是 **HttpApplication** 类的实例，将在客户端第一期从某个特定的 ASP.NET 应用程序虚拟目录中请求任何 URL 资源时创建。对于 Web 应用上的每个 ASP.NET 应用程序都要创建一个单独的实例。然后通过内部 **Application** 对象公开对每个实例进行引用。

#### 1. Application 对象的特性

对于 **Application** 对象有如下特性：

- ❑ 数据可以在 **Application** 对象之内进行数据共享，一个 **Application** 对象可以覆盖多个用户。
- ❑ **Application** 对象可以用 **Internet Service Manager** 来设置而获得不同的属性。
- ❑ 单独的 **Application** 对象可以隔离出来并运行在内存之中。
- ❑ 可以停止一个 **Application** 对象而不会影响到其他 **Application** 对象。

**Application** 对象常用的属性有：

- ❑ **AllKey**：获取 **HttpApplicationState** 集合中的访问键。
- ❑ **Count**：获取 **HttpApplicationState** 集合中的对象数。

其中 **Application** 对象的常用方法有：

- ❑ **Add**：新增一个 **Application** 对象变量。
- ❑ **Clear**：清除全部的 **Application** 对象变量。
- ❑ **Get**：通过索引关键字或变量名称得到变量的值。
- ❑ **GetKey**：通过索引关键字获取变量名称。
- ❑ **Lock**：锁定全部的 **Application** 对象变量。
- ❑ **UnLock**：解锁全部的 **Application** 对象变量。
- ❑ **Remove**：使用变量名称移除一个 **Application** 对象变量。
- ❑ **RemoveAll**：移除所有的 **Application** 对象变量。
- ❑ **Set**：使用变量名更新一个 **Application** 对象变量。

#### 2. Application 对象的使用

通过使用 **Application** 对象的方法，能够对 **Application** 对象进行操作，使用 **Add** 方法能够创建 **Application** 对象，示例代码如下所示。

```
Application.Add("App", "MyValue");           //增加 Application 对象
Application.Add("App1", "MyValue1");         //增加 Application 对象
Application.Add("App2", "MyValue2");         //增加 Application 对象
```

若需要使用 **Application** 对象，可以通过索引 **Application** 对象的变量名进行访问，示例代码如下所示：

```
Response.Write(Application["App1"].ToString()); //输出 Application 对象
```

上述代码直接通过使用变量名来获取 **Application** 对象的值。通过 **Application** 对象的 **Get** 方法也能够获取 **Application** 对象的值，示例代码如下所示。

```
for (int i = 0; i < Application.Count; i++) //遍历 Application 对象
{
    Response.Write(Application.Get(i).ToString()); //输出 Application 对象
}
```

**Application** 对象通常可以用来统计在线人数，在页面加载后可以通过配置文件使用 **Application** 对象的 **Add** 方法进行 **Application** 对象的创建，当用户离开页面时，可以使用 **Application** 对象的 **Remove** 方法进行 **Application** 对象的移除。当 Web 应用不希望用户在客户端修改已经存在的 **Application** 对象时，可以使用 **Lock** 对象进行锁定，当执行完毕相应的代码块后，可以解锁。示例代码如下所示。

```
Application.Lock();           //锁定 Application 对象
Application["App"] = "MyValue3"; //Application 对象赋值
Application.UnLock();         //解锁 Application 对象
```

上述代码当用户进行页面访问时，其客户端的 **Application** 对象被锁定，所以用户的客户端不能够进行



**Application** 对象的更改。在锁定后，也可以使用 **UnLock** 方法进行解锁操作。

## 13.1.4 Session 状态对象

**Session** 对象是 **HttpSessionState** 的一个实例，**Session** 是用来存储跨页程序的变量或对象，功能基本同 **Application** 对象一样。但是 **Session** 对象的特性与 **Application** 对象不同。**Session** 对象变量只针对单一网页的使用者，这也就是说各个机器之间的 **Session** 的对象不尽相同。

例如用户 **A** 和用户 **B**，当用户 **A** 访问该 **Web** 应用时，应用程序可以显式的为该用户增加一个 **Session** 值，同时用户 **B** 访问该 **Web** 应用时，应用程序同样可以为用户 **B** 增加一个 **Session** 值。但是与 **Application** 不同的是，用户 **A** 无法存取用户 **B** 的 **Session** 值，用户 **B** 也无法存取用户 **A** 的 **Session** 值。**Application** 对象终止于 **IIS** 服务停止，但是 **Session** 对象变量终止于联机机器离线时，也就是说当网页使用者关闭浏览器或者网页使用者在页面进行的操作时间超过系统规定时，**Session** 对象将会自动注销。

### 1. Session 对象的特性

**Session** 对象常用的属性有：

- ❑ **IsNewSession**: 如果用户访问页面时是创建新会话，则此属性将返回 **true**，否则将返回 **false**。
- ❑ **Timeout**: 传回或设置 **Session** 对象变量的有效时间，如果在有效时间内有没有任何客户端动作，则会自动注销。

注意：如果不设置 **Timeout** 属性，则系统默认的超时时间为 20 分钟。

**Session** 对象常用的方法有：

- ❑ **Add**: 创建一个 **Session** 对象。
- ❑ **Abandon**: 该方法用来结束当前会话并清除对话中的所有信息，如果用户重新访问页面，则可以创建新会话。
- ❑ **Clear**: 此方法将清除全部的 **Session** 对象变量，但不结束会话。

注意：**Session** 对象可以不需要 **Add** 方法进行创建，直接使用 **Session["变量名"]=变量值** 的语法也可以

进行 **Session** 对象的创建。

### 2. Session 对象的使用

**Session** 对象可以使用于安全性相比之下较高的场合，例如后台登录。在后台登录的制作过程中，管理员拥有一定的操作时间，而如果管理员在这段时间不进行任何操作的话，为了保证安全性，后台将自动注销，如果管理员需要再次进行操作，则需要再次登录。在管理员登录时，如果登录成功，则需要给管理员一个 **Session** 对象，示例代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    Session["admin"] = "guojing";           //新增 Session 对象
    Response.Redirect("Session.aspx");      //页面跳转
}
```

当管理员单击注销按钮时，则会注销 **Session** 对象并提示再次登录，示例代码如下所示。

```
protected void Button2_Click(object sender, EventArgs e)
{
    Session.Clear();                         //删除所有 Session 对象
    Response.Redirect("Session.aspx");
}
```

在 **Page\_Load** 方法中，可以判断是否已经存在 **Session** 对象，如果存在 **Session** 对象，则说明管理员当前的权限是正常的，而如果不存在 **Session** 对象，则说明当前管理员的权限可能是错误的，或者是非法用户正在访问该页面，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
```

```
{
    if (Session["admin"] != null)                                //如果 Session["admin"]不为空
    {
        if (String.IsNullOrEmpty(Session["admin"].ToString()))    //则判断是否为空字符串
        {
            Button1.Visible = true;                                //显式登录控件
            Button2.Visible = false;                                //隐藏注销控件
            //Response.Redirect("admin_login.aspx");                //跳转到登录页面
        }
        else
        {
            Button1.Visible = false;                                //显式注销控件
            Button2.Visible = true;                                //隐藏注销控件
        }
    }
}
```

上述代码当管理员没有登录时，则会出现登录按钮，如果登录了，存在 **Session** 对象，则登录按钮被隐藏，只显示注销按钮。其 **HTML** 代码如下所示。

```
<asp:Button ID="Button1" runat="server" onclick="Button1_Click" Text="登录" />
<asp:Button ID="Button2" runat="server" onclick="Button2_Click" Text="注销" />
```

上述代码运行后如图 13-5 和图 13-6 所示。



图 13-5 登录前



图 13-6 登录后

当再次单击【注销】按钮时则会清空 **Session** 对象，再次返回登录窗口时会呈现同图 13-5 所示。

13.1.5 Server 服务对象

**Server** 对象是 **HttpServerUtility** 的一个实例，该对象提供对服务器上的方法和属性进行访问。

1. Server 对象的常用属性

**Server** 对象的常用属性如下所示。

- ❑ **MachineName**: 获取远程服务器的名称。
- ❑ **ScriptTimeout**: 获取和设置请求超时。

通过 **Server** 对象能够获取远程服务器的信息，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    Response.Write(Server.MachineName);                            //输出服务器信息
}
```

上述代码运行后将会输出服务器名称，本例输出为“**WIN-YXDGNPG621**”，这个输出结果根据服务器的名称不同而不同。

2. Server 对象的常用方法

**Server** 对象的常用方法如下所示。

- ❑ **CreateObject**: 创建 **COM** 对象的一个服务器实例。
- ❑ **Execute**: 使用另一个页面执行当前请求。

- ❑ **Transfer:** 终止当前页面的执行，并为当前请求开始执行新页面。
- ❑ **HtmlDecode:** 对已被编码的消除 **Html** 无效字符的字符串进行解码。
- ❑ **HtmlEncode:** 对要在浏览器中显示的字符串进行编码。
- ❑ **MapPath:** 返回与 **Web** 服务器上的执行虚拟路径相对应的物理文件路径。
- ❑ **UrlDecode:** 对字符串进行解码，该字符串为了进行 **HTTP** 传输而进行编码并在 **URL** 中发送到服务器。
- ❑ **UrlEncode:** 编码字符串，以便通过 **URL** 从 **Web** 服务器到客户端浏览器的字符串传输。

在 **ASP.NET** 中，默认编码是 **UTF-8**，所以在使用 **Session** 和 **Cookie** 对象保存中文字符或者其他字符集时经常会出现乱码，为了避免乱码的出现，可以使用 **HtmlDecode** 和 **HtmlEncode** 方法进行编码和解码。**HTML** 页面代码如下所示。

```
<body>
  <form id="form1" runat="server">
    <p>
      HtmlDecode:
      <asp:Label ID="Label1" runat="server" Text="Label"></asp:Label>
    </p>
    <p>
      HtmlEncode:
      <asp:Label ID="Label2" runat="server" Text="Label"></asp:Label>
    </p>
  </form>
</body>
```

上述代码使用了两个文本标签控件用来保存并呈现编码后和解码后的字符串，在 **CS** 页面可以对字符串进行编码和解码操作，示例代码如下所示。

```
string str = "<p>(*^__^*) 嘻嘻.....</p>";           //声明字符串
Label1.Text = Server.HtmlEncode(str);                 //字符串编码
Label2.Text = Server.HtmlDecode(Label1.Text);         //字符串解码
```

上述代码将 **str** 字符串进行编码并存放在 **Label1** 标签中，**Label2** 标签将读取 **Label1** 标签中的字符串再进行解码，运行后如图 13-7 所示。



图 13-7 **HtmlEncode** 和 **HtmlDecode**

在使用了 **HtmlEncode** 方法后，编码后的 **HTML** 标注会被转换成相应的字符，如符号“<”会被转换成字符“&lt;”。在进行解码时，相应的字符会被转换回来，并呈现在客户端浏览器中。当需要让浏览器能够接受 **HTML** 字符时，**URL** 地址栏中对页面的参数的传递不能够包括空格，换行等符号，如果需要使用该符号，可以使用 **UrlEncode** 方法和 **UrlDecode** 方法进行变量的编码解码，示例代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    string str = Server.UrlEncode("错误信息 \n 操作异常");           //使用 UrlEncode 进行编码
    Response.Redirect("Server.aspx?str=" + str);                     //页面跳转
}
```

在 **Page\_Load** 方法中可以接收该字符串，示例代码如下所示。

```
if (Request.QueryString["str"] != "")
{
    Label3.Text = Server.UrlDecode(Request.QueryString["str"]);       //使用 UrlDecode 进行解码
}
```



}

当长字符串跳转和密封的信息在页面中进行发送和传递时，可以使用 **UrlEncode** 方法和 **UrlDecode** 方法进行变量的编码解码，以提高应用程序的安全性。

### 3. Server.MapPath 方法

在创建文件，删除文件或者读取文件类型的数据库时（如 **Access** 和 **SQLite**），都需要指定文件的路径并显式的提供物理路径执行文件的操作，如 **D:\Program Files**。但是这样做却暴露了物理路径，如果有非法用户进行非法操作，很容易就显示了物理路径，这样就造成了安全问题。

而如果在使用文件和创建文件时，如果非要为创建文件的保存地址设置一个物理路径，这样非常不便并且用户体验也不好。当用户需要上传文件时，用户不可能知道也不应该知道服务器路径。如果使用 **MapPath** 方法能够实现。**MapPath** 方法以 “/” 开头，则返回 **Web** 应用程序的根目录所在的路径，若 **MapPath** 方法以 “../” 开头，则会从当前目录开始寻找上级目录，如图 13-8 所示，而其实际服务器路径如图 13-9 所示。

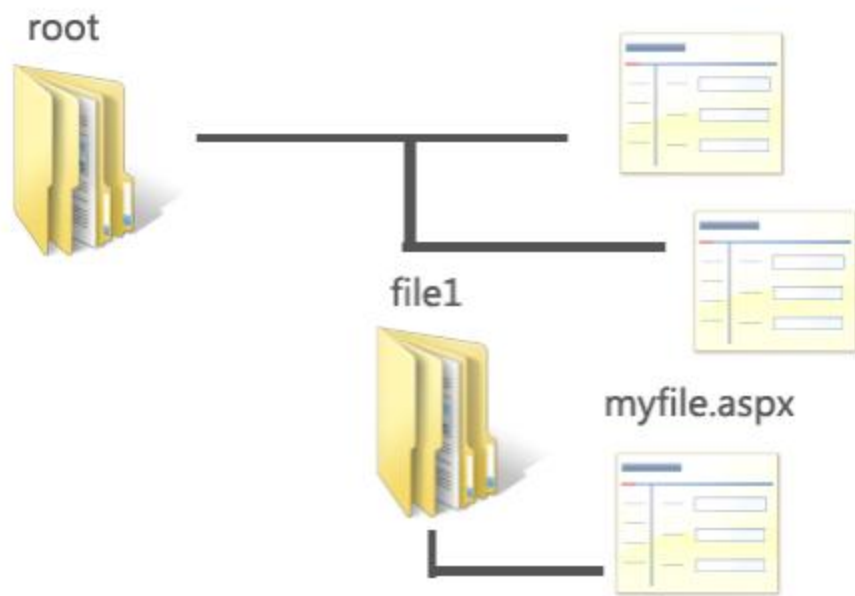


图 13-8 MapPath 示意图

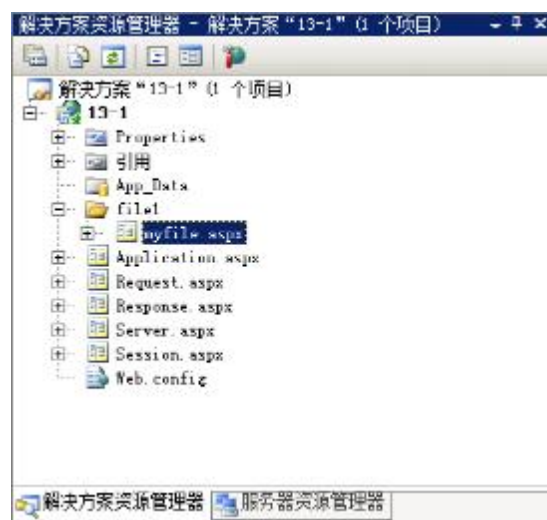


图 13-9 服务器路径

在图 13-8 所示，其中论坛根目录为 **root**，在根目录下有一个文件夹为 **file1**，在 **file1** 中的文件可以使用 **MapPath** 访问根目录中文件的方法有 **Server.MapPath("../文件名称")** 或 **Server.MapPath("/文件名称")**，示例代码如下所示。

```
string FilePath = Server.MapPath("../Default.aspx"); //设置路径
string FileRootPath = Server.MapPath("/Default.aspx"); //设置路径
```

**Server.MapPath** 其实返回的是物理路径，但是通过 **MapPath** 的封装，通过代码无法看见真实的物理路径，若需要知道真实的物理路径，只需输出 **Server.MapPath** 即可，示例代码如下所示。

```
Response.Write(Server.MapPath("../Default.aspx")); //输出路径
```

上述代码输出结果为 **D:\ASP.NET 3.5\源代码\第 13 章\13-1\13-1\Default.aspx**，该结果针对不同的物理路径而不同。

## 13.1.6 Cookie 状态对象

**Session** 对象能够保存用户信息，但是 **Session** 对象并不能够持久的保存用户信息，当用户在限定时间内没有任何操作时，用户的 **Session** 对象将被注销和清除，在持久化保存用户信息时，**Session** 对象并不适用。

### 1. Cookie 对象

使用 **Cookie** 对象能够持久化的保存用户信息，相比于 **Session** 对象和 **Application** 对象而言，**Cookie** 对象保存在客户端，而 **Session** 对象和 **Application** 对象保存在服务器端，所以 **Cookie** 对象能够长期保存。**Web** 应用程序可以通过获取客户端的 **Cookie** 的值来判断用户的身份来进行认证。

**ASP.NET** 内包含两个内部的 **Cookie** 集合。通过 **HttpRequest** 的 **Cookies** 集合来进行访问，**Cookie** 不是 **Page** 类的子类，所以使用方法和 **Session** 和 **Application** 不同。相比于 **Session** 和 **Application** 而言，**Cookie**



的优点如下所示。

- ❑ 可以配置到期的规则：**Cookie** 可以在浏览器会话结束后立即到期，也可以在客户端中无限保存。
- ❑ 简单：**Cookie** 是一种基于文本的轻量级结构，包括简单的键值对。
- ❑ 数据持久性：**Cookie** 能够在客户端上长期进行数据保存。
- ❑ 无需任何服务器资源：**Cookie** 无需任何服务器资源，存储在本地客户端中。

虽然 **Cookie** 包括若干优点，这些优点能够弥补 **Session** 对象和 **Application** 对象的不足，但是 **Cookie** 对象同样有缺点，**Cookie** 的缺点如下所示。

- ❑ 大小限制：**Cookie** 包括大小限制，并不能无限保存 **Cookie** 文件。
- ❑ 不确定性：如果客户端配置禁用 **Cookie** 配置，则 **Web** 应用中使用的 **Cookie** 将被限制，客户端将无法保存 **Cookie**。
- ❑ 安全风险：现在有很多的软件能够伪装 **Cookie**，这意味着保存在本地的 **Cookie** 并不安全，**Cookie** 能够通过程序修改为伪造，这会导致 **Web** 应用在认证用户权限时会出现错误。

**Cookie** 是一个轻量级的内置对象，**Cookie** 并不能将复杂和庞大的文本进行存储，在进行相应的信息或状态的存储时，应该考虑 **Cookie** 的大小限制和不确定性。

## 2. Cookie 对象的属性

**Cookie** 对象的属性如下所示：

- ❑ **Name**：获取或设置 **Cookie** 的名称。
- ❑ **Value**：获取或设置 **Cookie** 的 **Value**。
- ❑ **Expires**：获取或设置 **Cookie** 的过期的日期和事件。
- ❑ **Version**：获取或设置 **Cookie** 的符合 **HTTP** 维护状态的版本。

## 3. Cookie 对象的方法

**Cookie** 对象的方法如下所示：

- ❑ **Add**：增加 **Cookie** 变量。
- ❑ **Clear**：清除 **Cookie** 集合内的变量。
- ❑ **Get**：通过变量名称或索引得到 **Cookie** 的变量值。
- ❑ **Remove**：通过 **Cookie** 变量名称或索引删除 **Cookie** 对象。

## 4. 创建 Cookie 对象

通过 **Add** 方法能够创建一个 **Cookie** 对象，并通过 **Expires** 属性设置 **Cookie** 对象在客户端中所持续的时间，示例代码如下所示。

```
HttpCookie MyCookie = new HttpCookie("MyCookie ");
MyCookie.Value = Server.HtmlEncode("我的 Cookie 应用程序"); //设置 Cookie 的值
MyCookie.Expires = DateTime.Now.AddDays(5);                //设置 Cookie 过期时间
Response.AppendCookie(MyCookie);                            //新增 Cookie
```

上述代码创建了一个名称为 **MyCookie** 的 **Cookies**，上述代码通过使用 **Response** 对象的 **AppendCookie** 方法进行 **Cookie** 对象的创建，与之相同，可以使用 **Add** 方法进行创建，示例代码如下所示。

```
Response.Cookies.Add(MyCookie);
```

上述代码同样能够创建一个 **Cookie** 对象，当创建了 **Cookie** 对象后，将会在客户端的 **Cookies** 目录下建立文本文件，文本文件的内容如下所示。

```
MyCookie
MyCookie
```

注意：**Cookies** 目录在 **Windows** 下是隐藏目录，并不能直接对 **Cookies** 文件夹进行访问，在该文件夹中

可能存在多个 **Cookie** 文本文件，这是由于在一些网站中进行登录保存了 **Cookies** 的原因。

## 5. 获取 Cookie 对象

**Web** 应用在客户端浏览器创建 **Cookie** 对象之后，就可以通过 **Cookie** 的方法读取客户端中保存的 **Cookies** 信息，示例代码如下所示。

```

protected void Page_Load(object sender, EventArgs e)
{
    try
    {
        HttpCookie MyCookie = new HttpCookie("MyCookie ");           //创建 Cookie 对象
        MyCookie.Value = Server.HtmlEncode("我的 Cookie 应用程序");    //Cookie 赋值
        MyCookie.Expires = DateTime.Now.AddDays(5);                   //Cookie 持续时间
        Response.AppendCookie(MyCookie);                               //添加 Cookie
        Response.Write("Cookies 创建成功");                             //输出成功
        Response.Write("<hr/>获取 Cookie 的值<hr/>");
        HttpCookie GetCookie = Request.Cookies["MyCookie"];           //获取 Cookie
        Response.Write("Cookies 的值:" + GetCookie.Value.ToString() + "<br/>"); //输出 Cookie 值
        Response.Write("Cookies 的过期时间:" + GetCookie.Expires.ToString() + "<br/>");
    }
    catch
    {
        Response.Write("Cookies 创建失败");                             //抛出异常
    }
}

```

上述代码创建一个 **Cookie** 对象之后立即获取刚才创建的 **Cookie** 对象的值和过期时间。通过 **Request.Cookies** 方法可以通过 **Cookie** 对象的名称或者索引获取 **Cookie** 的值。

在一些网站或论坛中，经常使用到 **Cookie**，当用户浏览并登录在网站后，如果用户浏览完毕并退出网站时，**Web** 应用可以通过 **Cookie** 方法对用户信息进行保存。当用户再次登录时，可以直接获取客户端的 **Cookie** 的值而无需用户再次进行登录操作。

### 13.1.7 Cache 缓存对象

**Cache** 对象通过 **HttpContext** 对象的属性或 **Page** 对象的 **Cache** 属性来提供。**Cache** 对于每个应用程序域均创建该类的实例，只要相应的应用程序域是激活状态，则该实例则为有效状态。

#### 1. **Cache** 对象的属性

**Cache** 对象的属性如下所示：

- ☐ **Count**: 获取存储在缓存中的 **Cache** 对象的项数。
- ☐ **Item**: 获取或设置指定外键的缓存项。

#### 2. **Cache** 对象的方法

**Cache** 对象的方法如下所示。

- ☐ **Add**: 将指定的项添加到 **Cache** 对象，该对象具有依赖项，过期和优先级策略，以及一个委托。
- ☐ **Get**: 从 **Cache** 对象检索指定项。
- ☐ **Remove**: 从应用程序的 **Cache** 对象移除指定项。
- ☐ **Insert**: 向 **Cache** 对象插入一个新项。

#### 3. **Cache** 对象的使用

**Cache** 对象可以使用 **Get** 方法从相应的 **Cache** 对象中获取 **Cache** 对象的值，**Get** 方法能够通过 **Cache** 对象的名称和索引来获取 **Cache** 对象的值，示例代码如下所示。

```

protected void Button1_Click(object sender, EventArgs e)
{
    try
    {
        Cache.Get("Label1.Text");           //获取 Cache 对象的值
    }
    catch                                   //捕获异常，同 try 使用
    {
    }
}

```

```
Label2.Text = "获取 Cache 的值失败!";           //输出错误异常信息
    }
}
```

通过 **Cache** 的 **Count** 属性能够获取现有的 **Cache** 对象的项数，示例代码如下所示。

```
Response.Write("Cache 对象的项数有" + Cache.Count.ToString()); //输出 Cache 项数
```

13.1.8 Global.asax 配置

**Global.asax** 配置文件也称作 **ASP.NET** 应用程序文件，该文件是可选文件。该文件包含用于相应 **ASP.NET** 或 **HttpModule** 引发的应用程序级别事件的代码。**Global.asax** 配置文件主流在基于 **ASP.NET** 应用程序的根目录中，在应用程序运行时，首先编译器会分析 **Global.asax** 配置文件并将其编译到一个动态生成的 **.NET Framework** 类，该类是从 **HttpApplication** 基类派生的。**Global.asax** 配置文件不能够通过 **URL** 进行访问，以保证配置文件的安全性。

1. 创建 Global.asax 配置文件

**Global.asax** 配置文件通常处理高级的应用程序事件，如 **Application\_Start**、**Application\_End**、**Session\_Start** 等，**Global.asax** 配置文件通常不为个别页面或事件进行请求相应。创建 **Global.asax** 配置文件可以通过新建【全局应用程序类】文件来创建，如图 13-10 所示。



图 13-10 创建 Global.asax 配置文件

创建完成 **Global.asax** 配置文件，系统会自动创建一系列代码，开发人员只需要向相应的代码块中添加事务处理程序即可。

2. 应用域开始（Application\_Start）和应用域结束（Application\_End）事件

在 **Global.asax** 配置文件中，**Application\_Start** 事件会在 **Application** 对象被创建时触发，通常 **Application\_Start** 对象能够对应用程序进行全局配置。在统计在线人数时，通过重写 **Application\_Start** 方法可以实现实时在线人数统计，示例代码如下所示。

```
protected void Application_Start(object sender, EventArgs e)
{
    Application.Lock();           //锁定 Application 对象
    Application["start"] = "Application 对象被创建"; //创建 Application 对象
    Application.Unlock();         //解锁 Application 对象
}
```

当用户使用 **Web** 应用时，就会触发 **Application\_Start** 方法，而与之相反的是，**Application\_End** 事件在 **Application** 对象结束时被触发，示例代码如下所示。

```
protected void Application_End(object sender, EventArgs e)
{
    Application.Lock();           //锁定 Application 对象
}
```

```

Application["end"] = "Application 对象被销毁";
Application.UnLock();
}
//清除 Application 对象
//解锁 Application 对象

```

当用户离开当前的 **Web** 应用时，就会触发 **Application\_End** 方法，开发人员能够在 **Application\_End** 方法中清理相应的用户数据。

### 3. 应用域错误（Application\_Error）事件

**Application\_Error** 事件在应用程序发送错误信息时被触发，通过重写该程序，可以控制 **Web** 应用程序的错误信息或状态，示例代码如下所示。

```

protected void Application_Error(object sender, EventArgs e)
{
    Application.Lock();
    Application["error"] = "一个错误已经发生";
    Application.UnLock();
}
//锁定 Application 对象
//错误发生
//解锁 Application 对象

```

### 4. Session 开始（Session\_Start）和 Session 结束（Session\_End）事件

**Session\_Start** 事件在 **Session** 对象开始时被触发。通过 **Session\_Start** 事件可以统计应用程序当前访问的人数，同时也可以进行一些与用户配置相关的初始化工作，示例代码如下所示。

```

protected void Session_Start(object sender, EventArgs e)
{
    Session["count"] = 1;
}
//Session 开始执行

```

与之相反的是 **Session\_End** 事件，当 **Session** 对象结束时则会触发该事件，当使用 **Session** 对象统计在线人数时，可以通过 **Session\_End** 事件减少在线人数的统计数字，同时也可以对用户配置进行相关的清理工作，示例代码如下所示。

```

protected void Session_End(object sender, EventArgs e)
{
    Session["count"] = null;
    Session.Clear();
}
//设置 Session 为 null
//清除 Session 对象

```

上述代码当用户离开页面或者 **Session** 对象生命周期结束时被触发，在 **Session\_End** 中可以清除用户信息进行相应的统计操作。

**注意：****Session** 对象和 **Application** 对象都能够进行应用程序中在线人数或应用程序统计的统计和计算。

在选择对象时，可以按照应用要求（特别是对对象生命周期的要求）选择不同的内置对象。

## 13.2 ASP.NET 应用程序配置

**ASP.NET** 包含一个重要的特性，它为开发人员提供了一个非常方便的系统配置文件，就是常用的 **Web.config** 和 **Machine.config**。配置文件能够存用户或应用程序的储配置信息，让开发人员能够快速的建立 **Web** 应用环境，以及扩展 **Web** 应用配置。

### 13.2.1 ASP.NET 应用程序配置

**ASP.NET** 为开发人员提供了强大的灵活的配置系统，配置系统通常通过文件的形式存在于 **Web** 应用根目录下。这些配置文件通常包括两类，分别是 **Web.config** 和 **Machine.config**。**Machine.config** 是服务器配置文件。服务器配置信息通常存储在该文件中，该文件一般存储在系统目录中的“**systemroot**



\\Microsoft.NET\Framework\\VersionNumber\\CONFIG”目录下。一台服务器只有一个 **Machine.config** 文件，该文件描述了所有 **ASP.NET Web** 应用程序所需要的默认配置。

**Web.config** 是应用程序配置文件，该文件从 **Machine.config** 文件集成一部分基本配置，并且 **Web.config** 能够作为服务器上所有 **ASP.NET** 应用程序配置的跟踪配置文件。每个 **ASP.NET** 应用程序根目录都包含 **Web.config** 文件，所以对于每个应用程序的配置都只需要重写 **Web.config** 文件中的相应配置节即可。

在 **ASP.NET** 应用程序运行后，**Web.config** 配置文件按照层次结构为传入的每个 **URL** 请求计算惟一的配置设置集合。这些配置只会计算一次便缓存在服务器上。如果开发人员针对 **Web.config** 配置文件进行了更改，则很有可能造成应用程序重启。值得注意的是，应用程序的重启会造成 **Session** 等应用程序对象的丢失，而不会造成服务器的重启。

注意：如果针对 **Web.config** 文件中某些配置节（如 **processModel** 配置节）进行了更改，则可能需要重启 **IIS** 才能够让所做的应用程序配置立即生效。

## 13.2.2 Web.config 配置文件

**ASP.NET** 应用程序的配置信息都存放于 **Web.config** 配置文件中，**Web.config** 配置文件是基于 **XML** 格式的文件类型，由于 **XML** 文件的可伸缩性，使得 **ASP.NET** 应用配置变得灵活、高效、容易实现。同时，**ASP.NET** 不允许外部用户直接通过 **URL** 请求访问 **Web.config**，以提高应用程序的安全性。

### 1. Web.config 配置文件的优点

**Web.config** 配置文件使得 **ASP.NET** 应用程序的配置变得灵活、高效和容易实现，同时 **Web.config** 配置文件还为 **ASP.NET** 应用提供了可扩展的配置，使得应用程序能够自定义配置，不仅如此，**Web.config** 配置文件还包括以下优点。

- ❑ 配置设置易读性：由于 **Web.config** 配置文件是基于 **XML** 文件类型，所有的配置信息都存放在 **XML** 文本文件中，可以使用文本编辑器或者 **XML** 编辑器直接修改和设置相应配置节，相比之下，也可以使用记事本进行快速配置而无需担心文件类型。
- ❑ 更新的即时性：在 **Web.config** 配置文件中某些配置节被更改后，无需重启 **Web** 应用程序就可以自动更新 **ASP.NET** 应用程序配置。但是在更改有些特定的配置节时，**Web** 应用程序会自动保存设置并重启。
- ❑ 本地服务器访问：在更改了 **Web.config** 配置文件后，**ASP.NET** 应用程序可以自动探测到 **Web.config** 配置文件中的变化，然后创建一个新的应用程序实例。当浏览者访问 **ASP.NET** 应用时，会被重新定向到新的应用程序。
- ❑ 安全性：由于 **Web.config** 配置文件通常存储的是 **ASP.NET** 应用程序的配置，所以 **Web.config** 配置文件具有较高的安全性，一般的外部用户无法访问和下载 **Web.config** 配置文件。当外部用户尝试访问 **Web.config** 配置文件时，会导致访问错误。
- ❑ 可扩展性：**Web.config** 配置文件具有很强的扩展性，通过 **Web.config** 配置文件，开发人员能够自定义配置节，在应用程序中自行使用。
- ❑ 保密性：开发人员可以对 **Web.config** 配置文件进行加密操作而不会影响到配置文件中的配置信息。虽然 **Web.config** 配置文件具有安全性，但是通过下载工具依旧可以进行文件下载，对 **Web.config** 配置文件进行加密，可以提高应用程序配置的安全性。

使用 **Web.config** 配置文件进行应用程序配置，极大的加强了应用程序的扩展性和灵活性，对于配置文件的更改也能够立即的应用于 **ASP.NET** 应用程序中。

### 2. Web.config 配置文件的结构

**Web.config** 配置文件是基于 **XML** 文件类型的文件，所以 **Web.config** 文件同样包含 **XML** 结构中的树形结构。在 **ASP.NET** 应用程序中，所有的配置信息都存储在 **Web.config** 文件中的“<configuration>”配置节中。在此配置节中，包括配置节处理应用程序声明，以及配置节设置两个部分，其中，对处理应用程序的

声明存储在 **configSections** 配置节内，示例代码如下所示。

```
<configSections>
  <sectionGroup
    name="system.web.extensions"
    type="System.Web.Configuration.SystemWebExtensionsSectionGroup,
    System.Web.Extensions, Version=3.5.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35">
  <sectionGroup
    name="scripting"
    type="System.Web.Configuration.ScriptingSectionGroup,
    System.Web.Extensions, Version=3.5.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35">
    <section
      name="scriptResourceHandler"
      type="System.Web.Configuration.ScriptingScriptResourceHandlerSection,
      System.Web.Extensions, Version=3.5.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35"
      requirePermission="false" allowDefinition="MachineToApplication"/>
    <sectionGroup
      name="webServices"
      type="System.Web.Configuration.ScriptingWebServicesSectionGroup,
      System.Web.Extensions, Version=3.5.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35">
    </sectionGroup>
  </sectionGroup>
</sectionGroup>
</configSections>
```

配置节设置区域中的每个配置节都有一个应用程序声明。节处理程序是用来实现 **ConfigurationSection** 接口的 **.NET Framework** 类。节处理程序生命中包括了配置设置接的名称，以及用来处理该配置节中的应用程序的类名。

配置节设置区域位于配置节处理程序声明区域之后。对配置节的设置还包括子配置节的是配置，这些子配置节同父配置节一起描述一个应用程序的配置，通常情况下这些同父配置节由同一个配置节进行管理，示例代码如下所示。

```
<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>
```

虽然 **Web.config** 配置文件是基于 **XML** 文件格式的，但是在 **Web.config** 配置文件中并不能随意的自行添加配置节或者修改配置节的位置，例如 **pages** 配置节就不能存放在 **configSections** 配置节之中。在创建 **Web** 应用程序时，系统通常会自行创建一个 **Web.config** 配置文件在文件中，系统通常已经规定好了 **Web.config** 配置文件的结构。

## 13.2.3 ASP.NET 基本配置节

在 **Web.config** 配置文件中包括很多的配置节，这些配置节都用来规定 **ASP.NET** 应用程序的相应属性。

### 1. <configuration>：根配置节

所有 **Web.config** 的根配置节都存储与 **<configuration>** 标记中，在它内部封装了其他的配置节，示例代码如下所示。

```
<configuration>
  <syste.web>
```

```
.....
</configuration>
```

## 2. <configSections>: 处理声明配置节

该配置节主要用于自定义的配置节处理程序声明，该配置节由多个“<section>”配置节组成，示例代码如下所示。

```
<configSections>
  <sectionGroup
    name="system.web.extensions"
    type="System.Web.Configuration.SystemWebExtensionsSectionGroup,
    System.Web.Extensions, Version=3.5.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35">
    <sectionGroup name="scripting"
      type="System.Web.Configuration.ScriptingSectionGroup,
      System.Web.Extensions, Version=3.5.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35">
      <section name="scriptResourceHandler"
        type="System.Web.Configuration.ScriptingScriptResourceHandlerSection,
        System.Web.Extensions, Version=3.5.0.0, Culture=neutral,
        PublicKeyToken=31BF3856AD364E35"
        requirePermission="false" allowDefinition="MachineToApplication"/>
      </sectionGroup>
    </sectionGroup>
  </configSections>
```

其中<section>配置节包括 **name** 和 **type** 两种属性，**name** 属性指定配置数据配置节的名称，而 **type** 属性指定与 **name** 属性相关的配置处理程序类。

## 3. <appSettings>: 用户自定义配置节

“<appSettings>”配置节为开发人员提供 ASP.NET 应用程序的扩展配置，通过使用“<appSettings>”配置节能够自定义配置文件，示例代码如下所示。

```
<appSettings>
  <add key="Name" value="Guojing"/> //增加自定义配置节
  <add key="E-mail" value="soundbbg@live.cn"/>
</appSettings>
```

上述代码添加了两个自定义配置节，这两个自定义配置节分别为 **Name** 和 **E-mail**，用于定义该 Web 应用程序的开发者的信息，以便在其他页面中使用该配置节。

若需要在页面中使用该配置节，可以使用 **ConfigurationSettings.AppSettings**(“key 的名称”)方法来获取自定义配置节中的配置值，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    TextBox1.Text = ConfigurationSettings.AppSettings["name"].ToString(); //获取自定义配置节
}
```

“<appSettings>”配置节包括两个属性，分别为 **Key** 和 **Value**。**Key** 属性指定自定义属性的关键字，以方便在应用程序中使用该配置节，而 **Value** 属性则定义该自定义属性的值。

## 4. <customErrors>: 用户错误配置节

该配置节能够指定当出现错误时，系统自动跳转到一个错误发生的页面，同时也能够为应用程序配置是否支持自定义错误。“<customErrors>”配置节包括两种属性，这两种属性分别为 **mode** 和 **defaultRedirect**。其中 **mode** 包括 3 种状态，这三种状态分别为 **On**、**Off** 和 **RemoteOnly**。**On** 表示启动自定义错误；**Off** 表示不启动自定义错误；**RemoteOnly** 表示给远程用户显示自定义错误。另外：**defaultRedirect** 属性则配置了当应用程序发生错误时跳转的页面。

“<customErrors>”配置节还包括子配置节“<error>”，该标记用于特定状态的自定义错误页面，子标记“<error>”包括两个属性，分别为 **statusCode** 和 **redirect**，其中 **statusCode** 用于捕捉发生错误的状态码，而 **redirect** 指定发生该错误后跳转的页面，该配置节配置代码如下所示。

```
<customErrors mode="RemoteOnly" defaultRedirect="GenericErrorPage.htm">
  <error statusCode="403" redirect="NoAccess.htm" />
```



```
<error statusCode="404" redirect="FileNotFound.htm" />
</customErrors>
```

上述代码则在 **Web.config** 文件中配置了相应的 **customErrors** 信息。当出现 **404** 错误时，系统会自行跳转到 **FileNotFound.htm** 页面以提示 **404** 错误，开发人员能够编写 **FileNotFound.htm** 页面进行用户提示。

### 5. <globalization>: 全局编码配置节

“<globalization>”用于配置应用程序的编码类型，ASP.NET 应用程序将使用该编码类型分析 ASPX 等页面，常用的编码类型包括：

- ☐ **UTF-8: Unicode UTF-8 字节编码技术**，ASP.NET 应用程序默认编码。
- ☐ **UTF-16: Unicode UTF-16 字节编码技术**。
- ☐ **ASCII: 标准的 ASCII 编码规范**。
- ☐ **Gb2312: 中文字符 Gb2312 编码规范**。

在配置 “<globalization>” 配置节时，其编码类型可以参考上述编码类型，如果不指定编码类型，则 ASP.NET 应用程序默认编码为 **UTF-8**，示例代码如下所示。

```
<globalization fileEncoding="UTF-8" requestEncoding="UTF-8" responseEncoding="UTF-8"/>
```

### 6. <sessionState>: Session 状态配置节

<sessionState>配置节用于完成 ASP.NET 应用程序中会话状态的设置，<sessionState>配置节包括以下 5 种属性：

- ☐ **mode**: 指定会话状态的存储位置，一共有 **Off**、**Inproc**、**StateServer** 和 **SqlServer** 集中设置，**Off** 表示禁用该设置，**Inproc** 表示在本地保存会话状态，**StateServer** 表示在服务器上保存会话状态，**SqlServer** 表示在 **SQL Server** 保存会话设置。
- ☐ **stateConnectionString**: 用来指定远程存储会话状态的服务器名和端口号。
- ☐ **sqlConnectionString**: 用来连接 **SQL Server** 的连接字符串，当在 **mode** 属性中设置 **SqlServer** 时，则需要使用到该属性。
- ☐ **Cookieless**: 指定是否使用客户端 **cookie** 保存会话状态。
- ☐ **Timeout**: 指定在用户无操作时超时的时间，默认情况为 **20** 分钟。

<sessionState>配置节配置示例代码如下所示。

```
<sessionState mode="InProc" timeout="25" cookieless="false"></sessionState>
```

ASP.NET 不仅包括这些基本的配置节，还包括其他高级的配置节，高级的配置节通常用于指定界面布局样式，如母版页、默认皮肤、以及伪静态等高级功能。

## 13.3 ASP.NET 缓存功能

通常 **Web** 应用程序会处理大量的交互，在这些大量的交互中必然会造成频繁的数据处理。当 **Web** 应用程序中数据处理过于频繁时，会造成 **Web** 应用程序假死的状态，不仅如此，大量的重复请求还可能造成 **Web** 应用程序性能低下，这里就需要使用缓存减轻服务器压力。

### 13.3.1 缓存概述

为了防止不必要的数据处理，ASP.NET 允许开发人员将页面或数据进行缓存处理，当用户再次访问页面时，如果存在缓存则直接从缓存中获取用户信息或页面信息然后直接显示在客户端浏览器。这种缓存方式能够减少频繁的数据处理，减轻服务器压力。

#### 1. 应用程序缓存

应用程序缓存通过使用编程实现键/值对将任意数据存储在内存空间中。使用应用程序缓存与使用应用程序状态的方法类似。但是，与应用程序状态不相同，应用程序缓存中的数据是容易丢失的，应用程序缓存并不是在整个运行过程中都存在，应用程序缓存有一定的超时时间，当时间超过缓存设定的时间，缓存



会自动注销。

## 2. 页面输出缓存

在用户访问一个页面时，通常需要进行数据操作，例如网站的首页或其他信息页面，常常需要检索数据库中的数据并显示到页面中。如果每个用户的每次访问都需要进行一次数据操作的话，则当大量用户访问并进行数据操作时将会导致性能降低甚至造成死锁。

为了避免这种情况的出现，可以使用页面输出缓存。页面输出缓存允许开发人员将整页进行数据缓存。页面输出缓存对于那些不经常更改的但需要处理大量数据和事件的页面特别适用。

## 3. 缓存依赖

可以将缓存中的某一项的生存期配置为依赖于其他应用程序配置节，如某个文件或者数据库。当缓存依赖项被更改时，**ASP.NET** 将从缓存中移除对该项的数据缓存。例如网站静态页面，如果该页面数据需要进行缓存，可以保存为一个 **HTML** 文件，当页面被请求时，页面首先会判断是否有缓存依赖，如果存在，则执行相应的方法进行数据显示，当 **HTML** 文件被更改或被移除后，页面再次被请求，则会创建一个缓存依赖。

### 13.3.2 页面输出缓存

当用户访问页面时，整个页面将会被服务器保存在内存中，这样就对页面进行了缓存。当用户再次访问该页，页面不会再次执行数据操作，页面首先会检查服务器中是否存在缓存，如果缓存存在，则直接从缓存中获取页面信息，如果页面不存在，则创建缓存。

页面输出缓存适用于那些数据量较多，而不会进行过多的事件操作的页面，如果一个页面需要执行大量的事件更新，以及数据更新，则并不能使用页面输出缓存。使用 **@OutputCache** 指令能够声明页面输出缓存，示例代码如下所示。

```
<%@ OutputCache Duration="120" VaryByParam="none" %>
```

上述代码使用 **@OutputCache** 指令声明了页面缓存，该页面将被缓存 120 秒。**@OutputCache** 指令包括 10 个属性，通过这些属性能够分别为页面的不同情况进行缓存设置，常用的属性如下所示：

- ❑ **CacheProfile**: 获取或设置 **OutputCacheProfile** 名称。
- ❑ **Duration**: 获取或设置缓存项需要保留在缓存中的时间。
- ❑ **VaryByHeader**: 获取或设置用于改变缓存项的一组都好分隔的 **HTTP** 标头名称。
- ❑ **Location**: 获取或设置一个值，该值确定缓存项的位置，包括 **Any**、**Client**、**Downstream**、**None**、**Server** 和 **ServerAndClient**。默认值为 **Any**。
- ❑ **VaryByControl**: 获取或设置一簇分好分隔的控件标识符，这些标识符包含在当前页或用户控件内，用于改变当前的缓存项。
- ❑ **NoStore**: 获取或设置一个值，该值确定是否设置了 “**Http Cache-Control: no-store**” 指令。
- ❑ **VaryByCustom**: 获取输出缓存用来改变缓存项的自定义字符串列表。
- ❑ **Enabled**: 获取或设置一个值，该值指示是否对当前内容启用了输出缓存。
- ❑ **VaryByParam**: 获取查询字符串或窗体 **POST** 参数的列表。

通过设置相应的属性，可以为页面设置相应的缓存，当需要为 **Default.aspx** 设置缓存项时，可以使用 **VaryByParam** 属性进行设置，示例代码如下所示。

```
<%@ OutputCache Duration="120" VaryByParam="id" %>
```

上述代码使用了 **Duration** 属性和 **VarByParam** 属性设置了当前页的缓存属性。为一个页面进行整体的缓存设置往往是没有必要的，常常还会造成困扰，例如 **Default.aspx?id=1** 和 **Default.aspx?id=100** 在缓存时可能呈现的页面是相同的，这往往不是开发人员所希望的。通过配置 **VarByParam** 属性能够指定缓存参数，示例代码如下所示。

```
<%@ OutputCache Duration="120" VaryByParam="id" %>
```

上述代码则通过参数 **id** 进行缓存，当 **id** 项不同时，**ASP.NET** 所进行的页面缓存也不尽相同。这样保证了 **Default.aspx?id=1** 和 **Default.aspx?id=100** 在缓存时所显示的页面并不一致。**VarByHeader** 和 **VarByCustom**

主要用于根据访问页面的客户端对页面的外观或内容进行自定义。在 **ASP.NET** 中，一个页面可能需要为 **PC** 用户和 **MOBILE** 用户呈现输出，因此可以通过客户端的版本不同来缓存不同的数据，示例代码如下所示。

```
<%@ OutputCache Duration="120" VaryByParam="none" VaryByCustom="browser" %>
```

上述代码则为每个浏览器单独设置了缓存条目。

## 13.3.3 页面部分缓存

整页缓存在很多情况是不可行的，例如在留言本程序中。当用户第一次访问该页面，该页面就会被缓存在服务器内存中，当用户进行评论并进行刷新，当前页面还在缓存的生存周期中，页面不会立即显示刚才用户发表的评论，这种情况可能造成用户困扰。在这种情况下，就应该针对页面中不同的部分进行缓存，如对用户评论框进行缓存或数据绑定控件进行和 **HTML** 控件进行缓存，这样也可以减少数据操作次数。示例代码如下所示

```
<%@ OutputCache Duration="120" VaryByParam="*" %>
```

上述代码将缓存用户控件 **120** 秒，并且将针对查询字符串的每个变动进行缓存。

```
<%@ OutputCache Duration="120" VaryByParam="none" VaryByCustom="browser" %>
```

上述代码针对浏览器设置缓存条目并缓存 **120** 秒，当浏览器不同时，则会分别创建独立的缓存条目。

```
<%@ OutputCache Duration="120"
```

```
VaryByParam="none" VaryByCustom="browser" VaryByControl="TextBox1" %>
```

上述代码将服务器控件 **TextBox1** 的每个不同的值进行缓存处理。页面部分缓存是指输出存在页面的某些部分，而不是缓存整个页面的内容。页面部分缓存包括 **3** 种方法：

- ☐ 使用 **@OutputCatch** 指令声明方式为用户控件设置缓存功能。
- ☐ 在代码隐藏文件中使用 **PartialCachingAttribute** 类设置用户控件缓存。
- ☐ 使用 **ControlCachePolicy** 类以编程的方式指定用户缓存。

页面部分缓存与页面缓存在很多方面都很相似，其属性基本相同，页面部分缓存主要使用 **@OutputCache** 对象的属性进行缓存设置。

当对页面进行页面缓存时，通常需要针对不同的 **POST** 参数进行缓存，如果不针对参数单独的进行缓存，则形成应用程序错误，而这种错误不是逻辑上的错误，如 <http://localhost/Default.aspx?user=guojing> 所呈现的页面和 <http://localhost/Default.aspx?user=wujunmin> 会给用户呈现相同的页面，就会导致用户困扰。使用 **VaryByParam** 属性可以解决这个问题，示例代码如下所示。

```
<%@ OutputCache Duration="120" VaryByParam="name" %>
```

当存在多个 **POST** 参数时，可以用都好分隔，示例代码如下所示。

```
<%@ OutputCache Duration="120" VaryByParam="name" %>
```

当需要为每个执行的变动进行缓存时，可以使用 **\*** 进行缓存设置，示例代码如下所示。

```
<%@ OutputCache Duration="120" VaryByParam="*" %>
```

页面部分缓存通常应用于用户控件而不是 **Web** 窗体，除了 **Location** 属性，支持所有 **OutputCache** 属性。用户控件还支持名为 **VaryByControl** 的 **OutputCatch** 属性，该属性将根据用户控件的成员的值的改变该控件的缓存。如果一个用户控件不随应用程序中的页面而改变，则需要使用 **Shared="true"** 参数。

## 13.3.4 应用程序数据缓存

应用程序缓存是由 **Cache** 类实现的，每个应用程序对象可以专用一个缓存实例。缓存生存期依赖于应用程序的生存期，如果应用程序重启，则会重新创建 **Cache** 对象。

### 1. 创建 **Cache** 对象

应用程序数据缓存是由 **System.Web.Caching.Cache** 类实现。该类提供了简单的字典接口。通过这个接口可以设置缓存的有效期、依赖项以及优先级等特性。**Cache** 对象的属性如下所示：

- ☐ **Count**: 获取存储在缓存中的 **Cache** 对象的项数。
- ☐ **Item**: 获取或设置指定外键的缓存项。

Cache 对象的方法如下所示。

- ❑ **Add:** 将指定的项添加到 **Cache** 对象，该对象具有依赖项，过期和优先级策略，以及一个委托。
- ❑ **Get:** 从 **Cache** 对象检索指定项。
- ❑ **Remove:** 从应用程序的 **Cache** 对象移除指定项。
- ❑ **Insert:** 向 **Cache** 对象插入一个新项。

Cache 方法最简单的赋值方法就是使用一个键并为其赋值，示例代码如下所示。

```
Cache["key"] = "value"; //创建缓存项
```

增加缓存同样可以使用 **Cache** 对象的 **Add** 方法向缓存添加项。**Add** 方法能够设置与 **Insert** 方法相同的所有选项。使用 **Add** 方法将返回添加到缓存中的对象，如果在缓存中已经存在该项，则 **Add** 方法不会替换该项，所以该操作不会产生异常。

使用 **Add** 方法需要为 **Add** 方法传递参数，这些参数包括依赖项、过期时间和缓存的优先级策略等。若只需要实现创建 **Cache** 方法，推荐使用 **Cache** 对象的 **Insert** 方法，示例代码如下所示。

```
Cache.Insert("key", "value"); //插入缓存项
```

上述代码将在缓存中存储项，但是上述代码不带任何的依赖项，所以该缓存不会到期，除非缓存引擎为了给其他的缓存数据提供空间而将其删除，如果需要创建包含依赖项的缓存，则需要使用 **Add** 方法。

## 2. 创建带依赖项的 Cache 对象

创建 **Cache** 对象可以通过依赖项创建对象，使用 **Cache.Insert** 方法创建缓存不带任何依赖项，所以该缓存不会到期。如果需要创建带依赖项的 **Cache** 对象，可以使用 **Cache.Insert** 方法的重载函数，示例代码如下所示。

```
Cache.Insert("key", "xml file value",  
new System.Web.Caching.CacheDependency(Server.MapPath("XmlData.xml"))); //插入缓存项
```

上述代码将“**xml file value**”字符创插入缓存，无需在以后的请求中从文件读取缓存信息。

**CacheDependency** 的作用是确保缓存在文件更改后立即到期，从而能够从文件中提取最新的数据进行数据缓存，如果缓存数据来自若干个文件，还可以指定一个文件名数组，示例代码如下所示。

```
string[] dependencies = { "CacheItem1", "CacheItem2", "CacheItem3" }; //指定数组  
Cache.Insert("key", "xml file value",  
new System.Web.Caching.CacheDependency(null, dependencies)); //插入缓存项
```

创建带依赖项的 **Cache** 对象时，可以同时创建多个依赖项。该 **Cache** 对象向缓存中名为 **CacheItem1** 等数组配置节的另一项添加依赖项，同时也向 **XmlData.xml** 文件添加文件依赖项，示例代码如下所示。

```
System.Web.Caching.CacheDependency  
mydep1 = new System.Web.Caching.CacheDependency(Server.MapPath("XmlData.xml")); //创建缓存  
string[] Mydependencies = { "CacheItem1", "CacheItem2", "CacheItem3" }; //创建数组  
System.Web.Caching.CacheDependency  
mydep2 = new System.Web.Caching.CacheDependency(null, Mydependencies); //创建依赖  
System.Web.Caching.AggregateCacheDependency  
aggDep = new System.Web.Caching.AggregateCacheDependency(); //创建依赖  
aggDep.Add(mydep1); //增加依赖  
aggDep.Add(mydep2);  
Cache.Insert("Cache", "CacheItem", aggDep); //插入缓存
```

上述代码创建了 **Cache** 对象，并向 **CacheItem1** 等数组配置节的另一项添加依赖项，同时也向 **XmlData.xml** 文件添加文件依赖项。

## 13.3.5 检索应用程序数据缓存对象

要从缓存中检索数据，应指定存储缓存项的键。由于缓存中所存储的信息容易丢失，即该缓存信息有可能被 **ASP.NET** 移除，因此开发人员首先应该确定该项是否已经存在与缓存，如果不存在缓存，则首先应该创建一个缓存项，然后再检索该项。示例代码如下所示。

```
string cache = Cache["key"].ToString(); //获取缓存  
if (cache != null) //判断缓存  
{
```



```
        Response.Write(cache);                                //输出缓存
    }
    else
    {
        Cache["key"] = "value";                                //创建缓存
        Response.Write(Cache["key"].ToString());              //输出缓存
    }
}
```

上述代码从缓存项“key”中获取缓存的值，如果缓存的值为空置，则创建一个缓存。使用 **Cache** 类的 **Get** 方法也可以获取被缓存的数据对象，如果通过 **Get** 方法返回缓存中的数据对象，则返回的类型为 **object** 类型，示例代码如下所示。

```
    string cache = Cache["key"].ToString();                    //获取缓存
    if (cache != null)                                         //判断缓存
    {
        Response.Write(cache);                                //输出缓存
    }
    else
    {
        Cache["key"] = "value";                                //创建缓存
        Response.Write(Cache.Get("key").ToString());          //Get 缓存
    }
}
```

**Cache** 类的 **Get** 方法可以获取被缓存的数据对象，如果使用 **Cache** 类的 **GetEnumerator** 方法则返回一个 **IDictionaryEnumerator** 对象，该对象可以用于循环访问包含在缓存中的键值设置及其值的枚举类型。示例代码如下所示。

```
IDictionaryEnumerator cacheEnum = Cache.GetEnumerator();    //定义枚举
while (cacheEnum.MoveNext())                                  //遍历枚举
{
    cache = Server.HtmlEncode(cacheEnum.Current.ToString()); //输出缓存
    Response.Write(cache);
}
```

上述代码则通过使用 **Cache** 类的 **GetEnumerator** 方法则返回一个 **IDictionaryEnumerator** 对象给 **cacheEnum** 变量，并通过 **cacheEnum** 变量的 **MoveNext** 方法进行缓存遍历。

13.4 小结

本章讲解了 **ASP.NET** 内置对象，包括如何创建 **ASP.NET** 内置对象和使用 **ASP.NET** 内置对象。**Web** 应用程序从本质上来讲是无状态的，为了维持客户端的状态，必须使用 **ASP.NET** 内置对象进行客户端状态维护，这些状态包括 **Session**、**Cookies** 等。本章还包括：

- ❑ **ASP.NET** 内置对象：包括 **Session**、**Cookies** 等内置对象。
- ❑ **ASP.NET** 应用程序配置：包括 **ASP.NET** 应用程序配置。
- ❑ **Web.config** 配置文件：讲解了 **Web.config** 配置文件中常用的属性。
- ❑ 应用程序数据缓存：讲解了应用程序数据缓存。
- ❑ 检索应用程序数据缓存对象：讲解了从缓存中获取应用程序数据。

本章还讲解了缓存中的 **Cache** 对象和 **Web.config** 配置节，开发人员能够快速的构建 **ASP.NET** 应用环境并进行性能优化。



## 第 14 章 ASP.NET XML 和 Web Service

在上一章中讲到的 **Web.config** 配置文件就是基于 **XML** 文件格式的, **XML** (**Extensible Markup Language**, 可扩展标记语句) 是一种描述数据和数据结构的语言, **XML** 文本可以保存在任何存储文本中, 这就让 **XML** 具有了可扩展性、跨平台型以及传输与存储方面的优点。

### 14.1 XML 简介

标记语言 (**Markup Language**) 特指一系列约定好的标记来对电子文档进行标记, 以实现对电子文档的语义、结构以及格式的定义。在 **ASP.NET** 开发中, 最常用的标记语言就是 **HTML**, **HTML** 标记语言定义了 **HTML** 文档的语义、结构以及格式, 以便在不同的浏览器中所呈现的内容是一致的。**XML** 标记语言与 **SGML** 和 **HTML** 都属于标记语言, 标记语言的发展如图 14-1 所示。

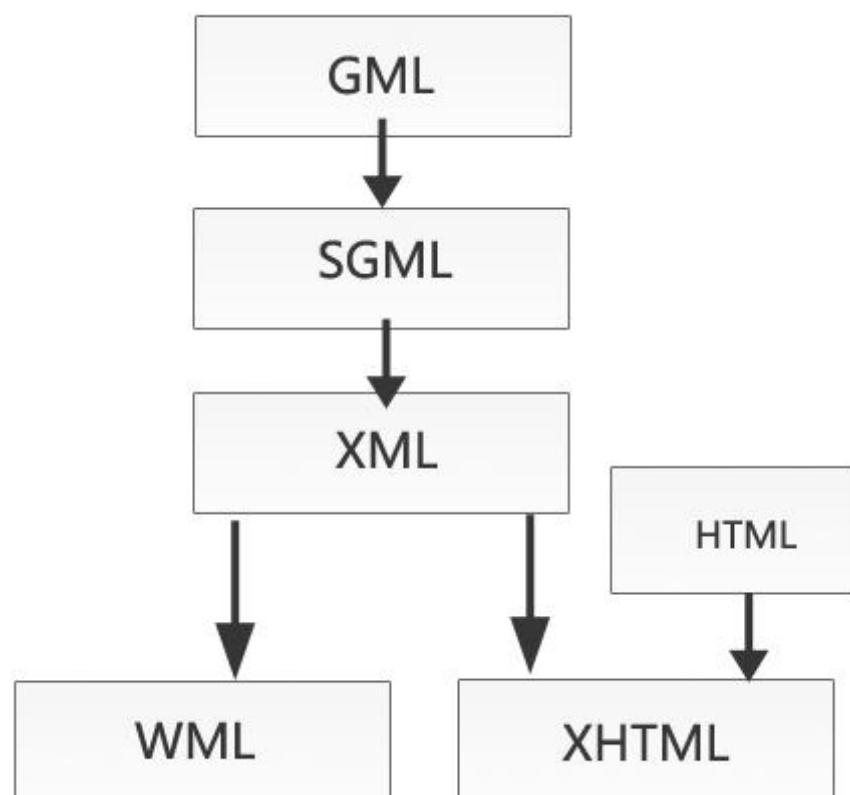


图 14-1 标记语言的发展史

为了更加方便的适应互联网的需求, 1996 年开始创建 **XML** 标记语言。**XML** 标记语言不仅具备了 **SGML** 标记语言强大的扩展性, 同样也具备 **HTML** 标记语言的易用性。不仅如此, **ASP.NET** 还将 **XML** 作为应用程序数据存储和传输的重要方法。

在当今互联网中, **Web** 应用已经成为一种分布式组件技术。传统的 **Web** 应用技术解决的问题是如何让人使用 **Web** 提供的应用, 而当今的 **Web** 应用技术是要解决如何让应用程序使用 **Web** 应用。由于 **Web** 应用能够跨平台、跨语言的为应用程序提供服务, 所以 **Web** 应用和 **XML** 应用的前景是非常广阔的。

**注意：**这里所说的 **Web** 应用的跨平台是针对浏览器而言, **Windows** 能够浏览一个 **Web** 应用, 而 **Linux** 同样可以浏览一个 **Web** 应用。

## 14.2 读写 XML

**XML** 和 **HTML** 都是基于 **SGML**（**Standard Generalized Markup Language**，标准通用标记语言）的，但是 **XML** 和 **HTML** 却有着很大的区别，这些区别不仅仅在于格式上的区别，还在于使用性、可扩展性等等。

### 14.2.1 XML 与 HTML

**XML** 标记语言和 **HTML** 标记语言有着极大的不同，在应用程序开发中，**XML** 标记语言能够适应于大部分的应用程序环境和开发需求。这些需求是 **HTML** 标记语言无法做到的，**XML** 标记语言和 **HTML** 标记语言的具体区别如下所示。

- ❑ **HTML** 标记是固定的，并且是没有层次的，在 **HTML** 文档中，用户无法自行创建标签，例如<book>这样的标签浏览器很可能解析不了，**HTML** 中标记的作用是描述数据的显示方式，这种方式只能交付给浏览器进行处理，而 **HTML** 文档中的标记都是独立存在的，没有层次。
- ❑ **XML** 的标记不是固定的且是有层次的，在 **XML** 文档中，用户可以自行创建标签，例如<day>这样的标签，**XML** 标记不能够描述网页的外观和内容，**XML** 只能够描述内容的数据结构和层次，在浏览器中浏览 **XML** 文档，也可以发现 **XML** 标记是有层次的。

在 **Visual Studio 2008** 中，**.NET Framework** 提供了 **System.XML** 命名空间，该命名空间提供了一组可扩展类使得开发人员能够轻松的读、写、以及编辑 **XML** 文本。

### 14.2.2 创建 XML 文档

使用 **Visual Studio 2008** 能够创建 **XML** 文档，创建和使用 **XML** 文档无需 **XML** 语法分析器来专门负责分析语法，在 **.NET Framework** 中已经集成了可扩展类。右击现有项，单击【添加新项】选项，选择 **XML** 文件，如图 14-2 所示。



图 14-2 创建 XML 文档

创建完成后，就需要向 **XML** 文档中编写 **XML** 标记，以下是一个完整的 **XML** 文档示例。

```
<?xml version="1.0" encoding="utf-8" ?>
<Root>
  <ShopInformation area="Asia">
    <Shop place="Wuhan">
      <Name>武汉电脑城</Name>
      <Phone>123456789</Phone>
      <Seller>J.Dan</Seller>
    </Shop>
  </ShopInformation>
</Root>
```

```
<Seller>Bill Gates</Seller>
</Shop>
<Shop place="ShangHai">
  <Name>武汉电脑城</Name>
  <Phone>123456789</Phone>
  <Seller>Bill Gates</Seller>
</Shop>
</ShopInformation>
<ShopInformation area="USA">
  <Shop place="S">
    <Name>PC STORE</Name>
    <Phone>123456789</Phone>
    <Seller>J.Dan</Seller>
    <Seller>Bill Gates</Seller>
  </Shop>
  <Shop place="S.K">
    <Name>Windows Mobile Store</Name>
    <Phone>123456789</Phone>
    <Seller>Bill Gates</Seller>
  </Shop>
</ShopInformation>
</Root>
```

上述 XML 文档使用了自定义标记对商城进行了描述，包括商城所在地、商城名称、电话号码以及负责人等。编写 XML 文档时，开发人员能够自定义标签进行文档描述，但是在 XML 文档的头部必须进行 XML 文档声明，示例代码如下所示。

```
<?xml version="1.0" encoding="utf-8" ?>
```

上述代码在 XML 文档头部进行了声明，表示该文档是一个 XML 文档，并且说明该文档的版本为 1.0 的 XML 文档，该文档还可以包含一个 encoding 属性，指明文档中的编码类型。声明该文档是一个 XML 文档后，则需要在 XML 文档中编写根标记，这个标记可以是开发人员自定义标记名称，在这里被命名为 Root，示例代码如下。

```
<Root>
<!-- 根标记内的所有内容 -->
</Root>
```

上述代码创建了一个根标记，在这里命名为 Root。在 XML 文档中，所有的标记都应该被包含在一个根标记中，这样不仅方便描述也方便查阅。XML 文档中的根标记不能够重复使用，如果重复使用则会提示异常。

在根标记内，应该编写需要描述的信息的标记。在这里，描述一个商城需要的一些属性，包括商城所在的州、所在地以及商城的主营类型等，通过 XML 标记语言可以自行创建标记来描述，示例代码如下所示。

```
<ShopInformation area="USA">                                     //地区描述
  <Shop place="S">                                               //位置描述
    <Name>PC STORE</Name>                                         //商城名称
    <Phone>123456789</Phone>                                       //商城电话
    <Seller>J.Dan</Seller>                                         //商城销售人员
    <Seller>Bill Gates</Seller>
  </Shop>
  <Shop place="S.K">
    <Name>Windows Mobile Store</Name>
    <Phone>123456789</Phone>
    <Seller>Bill Gates</Seller>
  </Shop>
</ShopInformation>
```

上述代码对商城的信息进行了描述，这些标签的意义如下所示：

- ❑ **ShopInformation:** 商城信息，包括 area 属性来描述所在州或板块，这里说明了是在 USA 地区。
- ❑ **Shop:** 商城在该板块的所在州、省市等信息。

- ❑ **Name:** 商城的名称。
- ❑ **Phone:** 商城的联系电话。
- ❑ **Seller:** 商城的销售人员。

这些标签都是用户自定义的，XML 文档允许开发人员自定义标签并，另外，XML 文档也不局限所要描述的对象格式。例如当上述代码也可以编写另外一种样式时，同样能够被 XML 所识别，示例代码如下所示。

```
<ShopInformation>
  <Area name="Usa">                                     //另一种地区表示方式
    <Shop>
      <Name>PC STORE</Name>
      <Phone>123456789</Phone>
      <Seller>J.Dan</Seller>
      <Seller>Bill Gates</Seller>
      <Place>S.K</Place>                                //地区直接放在描述中
    </Shop>
    <Shop>
      <Name>Windows Mobile Store</Name>
      <Phone>123456789</Phone>
      <Seller>Bill Gates</Seller>
      <Place>S</Place>
    </Shop>
  </Area>
</ShopInformation>
```

技巧：良好的缩进能够让 XML 文档更加方便阅读，同时 XML 文档是大小写敏感的，对于 XML 标记，标记头和标记尾的大小写规则必须匹配。

14.2.3 XML 控件

在 ASP.NET 中提供了针对 XML 读写的控件 XML 控件，XML 控件可以很好的解决 XML 文档的显示问题，如果需要浏览 XML 文档的数据，则只需要编写 XML 控件中的 DocumentSource 属性即可，示例代码如下所示。

```
<body>
  <form id="form1" runat="server">
    <div>
      <asp.Xml ID="xml1" runat="server" DocumentSource="~/XMLFile1.xml"></asp.Xml>    //使用 XML 控件
    </div>
  </form>
</body>
```

运行后如图 14-3 所示。

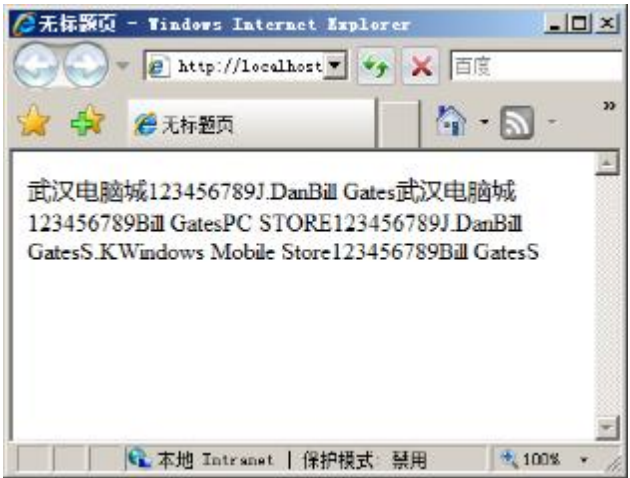




图 14-3 XML 控件

运行后会发现 XML 文档的内容都显示出来了，但是却没有层次感，因为 XML 控件并没有把记录分开，而是连续的呈现 XML 文档的内容。如果需要按照规范或开发人员的意愿呈现给浏览器，则必须使用 XSL 样式表。

14.2.4 XML 文件读取类（XmlTextReader）

XmlTextReader 类属于 System.Xml 命名空间，XmlTextReader 类提供对 XML 数据的快速、单项、无缓冲的数据读取功能，因为 XmlTextReader 类是基于流的，所以使用 XmlTextReader 类读取 XML 内容只能够从前向后读取，而不能逆向读取。

因为 XmlTextReader 类的流形式，节约了读取 XML 文档的时间，也大量的节约了读取 XML 所需花费的内存空间，当需要读取 XML 节点时，只需要使用 XmlTextReader 类的 Read()方法即可，示例代码如下所示。

```
XmlTextReader rd = new XmlTextReader(Server.MapPath("XMLFile1.xml"));           //构造函数
while (rd.Read())                                                                //遍历节点
{
    Response.Write("Node Type is : " + rd.NodeType + "&nbsp;<br/>");           //输出 Node
    Response.Write("Name is : " + rd.Name + "&nbsp;<br/>");                       //输出 Name
    Response.Write("Value is : " + rd.Value + "&nbsp;<br/>");                     //输出 Value
    Response.Write("<hr/>");
}
```

上述代码使用 XmlTextReader 类的构造函数创建了 XmlTextReader 对象，并通过使用 XmlTextReader 类的 Read()方法进行 XmlTextReader 对象的遍历。遍历 XML 文档后，需要使用 Close 方法进行 XmlTextReader 对象的关闭操作，这一点是非常重要的，如果不使用 XmlTextReader 类的 Close 方法，则相应的 XML 文件正在被进程使用，只有使用了 Close 方法才能将相应的文件关闭掉。示例代码如下所示。

```
rd.Close();                                                                    //关闭 Reader
```

XmlTextReader 类遍历 XML 文件运行结果如图 14-4 所示。

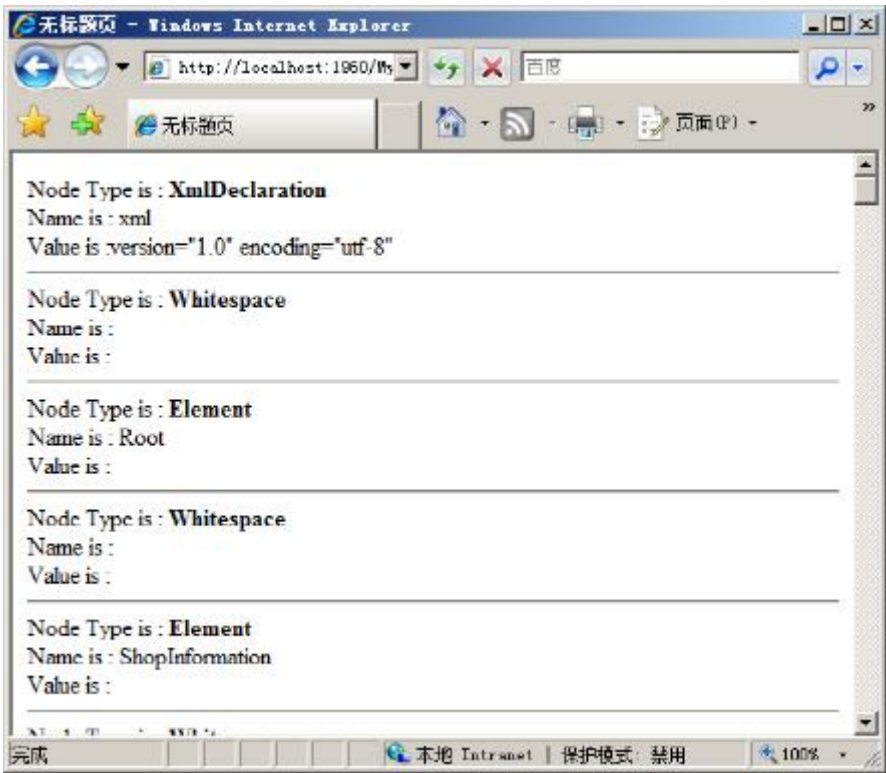


图 14-4 XmlTextReader 类遍历 XML 文件

在使用 XmlTextReader 类读取 XML 文件中相应的节点时，XmlTextReader 类的 NodeType 会检查节点的类型，而 XmlTextReader 类的 Name 和 Value 会分别检查节点的名称和值，相应的 XML 代码如下所示。

```
<Shop place="Wuhan">
    <Name>武汉电脑城</Name>
```

```
<Phone>123456789</Phone>
<Seller>J.Dan</Seller>
<Seller>Bill Gates</Seller>
</Shop>
```

上述代码中，使用 **XmlTextReader** 类进行读取，则 **Shop** 节点的 **NodeType** 为 **Element**，**Name** 的值为 **Shop**，**Value** 的值为空。**XML** 文档中不止以上几种节点类型，**XmlNodeType** 也包括其他节点类型，这些类型如下所示。

- ❑ **Attribut**: **XML** 元素的属性。
- ❑ **CDATA**: 用于转义文本块，避免将文本块识别为标记。
- ❑ **Comment**: **XML** 文档的注释。
- ❑ **Document**: 作为文档树的根的文档对象，可供每个 **XML** 文档进行访问。
- ❑ **DocumentType**: **XML** 文档类型的声明。
- ❑ **Element**: **XML** 元素。
- ❑ **EndElement**: 当 **XmlTextReader** 达到元素末尾时返回。
- ❑ **Entity**: 实体声明。
- ❑ **Text**: 元素的文本内容。
- ❑ **WhiteSpace**: 标记间的空白。
- ❑ **XmlDeclaration**: **XML** 节点声明，它是文档中的第一个节点。

在 **XML** 文档中，空白标记和根节点的节点类型是不相同的，**XmlTextReader** 类读取 **XML** 文件并遍历节点类型，根节点和空白节点遍历后结果如下所示。

```
Node Type is:XmlDeclaration
Nameis:xml
Value is:version="1.0" encoding="utf-8"
Node Type is:Whitespace
Nameis:
Value is:
```

其中根节点的节点类型为 **XmlDeclaration**，**Value** 值为 **version="1.0" encoding="utf-8"**。

## 14.2.5 XML 文件编写类（XmlTextWriter）

**XmlTextWriter** 类属于 **System.Xml** 命名空间，同 **XmlTextReader** 类相同的是，**XmlTextWriter** 类同样提供没有缓存，直向前的方式进行 **XML** 文件操作，但是与 **XmlTextReader** 类操作相反，**XmlTextWriter** 类执行的是写操作。**XmlTextWriter** 类的构造函数包括三种重载形式，分别为一个字符串、一个流对象和一个 **TextWriter** 对象。通过使用 **XmlTextWriter** 类可以动态的创建 **XML** 文档，示例代码如下所示。

```
XmlTextWriter wr = new XmlTextWriter("newXml.xml", null);           //读取 XML
try
{
    wr.Formatting = Formatting.Indented;                             //格式化输出
    wr.WriteStartDocument();                                           //开始编写文档
    wr.WriteStartElement("ShopInformation");                           //编写节点
    wr.WriteStartElement("Shop");                                     //编写节点
    wr.WriteAttributeString("place", "北京");                         //编写节点
    wr.WriteElementString("Name", "中关村");                          //编写节点
    wr.WriteElementString("Phone", "123456");                         //编写节点
    wr.WriteElementString("Seller", "Guojing");                       //编写节点
    wr.WriteEndElement();                                              //结束节点编写
    wr.WriteEndElement();                                              //结束节点编写
    Response.Write("操作成功");
}
catch
{
}
```

```
Response.Write("操作失败");  
}
```

上述代码创建了一个 **XmlTextWriter** 对象并通过 **XmlTextWriter** 对象编写 **XML** 文档，在使用 **XmlTextWriter** 类构造函数时，可以指定编码类型，或使用默认的编码类型，若使用默认的编码类型，参数传递 **null** 即可，默认编码类型将为 **UTF-8**，示例代码如下所示。

```
XmlTextWriter wr = new XmlTextWriter("newXml.xml", null); //创建写对象
```

使用了 **XmlTextWriter** 类创建对象后，则需要使用 **XmlTextWriter** 对象的 **Formatting** 方法指定输出的格式，示例代码如下所示。

```
wr.Formatting = Formatting.Indented; //格式化输出
```

指定了输出格式之后，则需要开始为 **XML** 文档创建节点，在创建节点前，首先需要声明 **XML** 文档，则必须输出 **<?xml version="1.0" encoding='utf-8' ?>** 声明，声明 **1.0** 版本的 **xml** 文档代码如下所示。

```
wr.WriteStartDocument(); //开始编写节点
```

声明文档后就可以使用 **WriteStartElement** 进行节点的创建，创建节点代码如下所示。

```
wr.WriteStartElement("Shop"); //开始编写节点
```

上述代码创建了 **Shop** 节点，如果需要为该节点创建 **place=“北京”** 属性则需要使用 **WriteAttributeString** 方法进行创建，示例代码如下所示。

```
wr.WriteAttributeString("place", "北京"); //编写属性
```

创建了父节点之后，可以通过 **WriteElementString** 方法创建子节点，示例代码如下所示。

```
wr.WriteElementString("Name", "中关村"); //创建子节点
```

节点全部创建完成后，需要使用 **WriteEndElement** 方法进行尾节点的编写，示例代码如下所示。

```
wr.WriteEndElement(); //结束节点编写
```

一个 **XML** 文档就编写完毕了，编写完成并不能自动的更新 **XML** 文档，还需要使用 **Flush** 方法进行数据更新，更新完毕后还需要关闭 **XmlTextWriter** 对象示例代码如下所示。

```
wr.Flush(); //更新文件  
wr.Close(); //结束写对象
```

使用 **Flush** 方法就能够将 **XML** 数据保存在文件中，运行后 **XML** 文档结构如下所示。

```
<?xml version="1.0" encoding="utf-8" ?>  
<ShopInformation>  
  <Shop place="北京">  
    <Name>中关村</Name>  
    <Phone>123456</Phone>  
    <Seller>Guojing</Seller>  
  </Shop>  
</ShopInformation>
```

14.2.6 XML 文本文档类（XmlDocument）

**XML** 文档在内存中是以 **DOM** 为表现形式的，**DOM**（**Document Object Model**）是对象化模型，**DOM** 是以树的节点形式来标识 **XML** 数据，14.2.2 中的 **XML** 文档读入 **DOM** 结构中，则在内存中的构造图如图 14-5 所示。

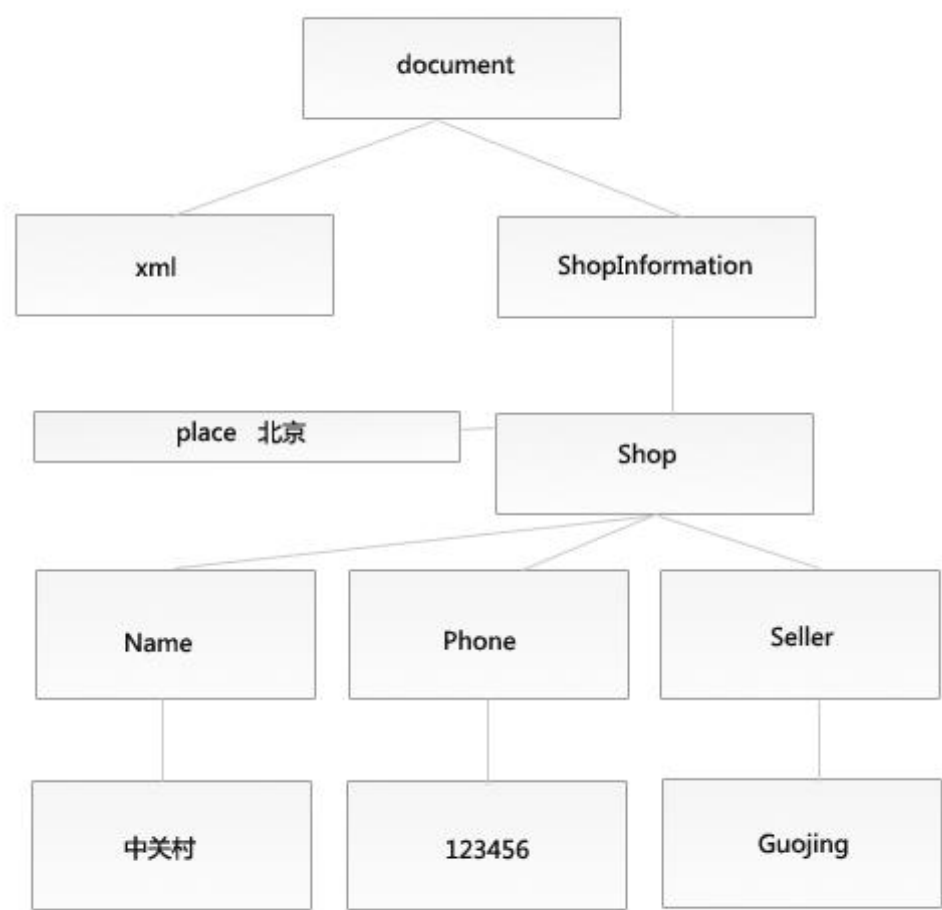


图 14-5 XML 文档构造

**XmlDocument** 类同样也属于命名空间 **System.Xml**，**XmlDocument** 类可以实现第一、第二级的 **W3C DOM**。它使用 **DOM** 以一个层次结构树的形式将整个 **XML** 数据加载到内存中，从而能够使开发人员能够对内存中的任意节点进行访问、插入、更新和删除。由于 **XmlDocument** 类，简化开发人员对 **XML** 文档进行访问、插入和删除等操作。

**XmlDocument** 类继承自 **System.Xml.XmlNode**，该抽象类表示单个节点并具有基本的属性和方法来操作节点。利用 **XmlDocument** 对象的 **DocumentElement** 属性能够表示单个节点并进行操作。**XmlDocument** 对象的 **DocumentElement** 返回一个指向文档元素的索引，可以通过读取给定的节点的 **HasChildNodes** 属性判断是否包括子节点。另外，使用 **XmlDocument** 对象的 **HasChildNodes** 和 **ChildNodes** 属性可以读取和遍历 **XML** 文件，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    XmlDocument doc = new XmlDocument();           //创建 XmlDocument 对象
    doc.Load(Server.MapPath("newXml.xml"));         //载入文件
    Response.Write("读取中..<hr/>");
    XmlNode node = doc.DocumentElement;            //读取节点
    output(node);                                   //使用输出函数
    Response.Write("读取完毕..<hr/>");             //输出 HTML 字符串
}
protected void output(XmlNode node)
{
    if (node != null)                               //如果节点不等于空
    {
        format(node);                               //格式化输出
    }
    if (node.HasChildNodes)                          //判断是否包括子节点
    {
        node = node.FirstChild;                     //获取子节点
        while (node != null)
        {
            output(node);                           //使用递归
        }
    }
}
```



```

        node = node.NextSibling;           //遍历节点值和信息
    }
}
protected void format(XmlNode node)
{
    if (!node.HasChildNodes)               //判断是否包括子节点
    {
        Response.Write("node name is" + node.Name);           //输出节点
        Response.Write("node value is" + node.Value);         //输出节点
        Response.Write("<br/>");
    }
    else
    {
        Response.Write(node.Name);
        if (XmlNodeType.Element == node.NodeType)             //遍历节点
        {
            XmlNamedNodeMap map = node.Attributes;            //遍历节点
            foreach (XmlNode att in map)
            {
                Response.Write("attrnode name is" + att.Name); //格式化输出节点
                Response.Write("attrnode value is" + att.Value); //格式化输出节点
                Response.Write("<br/>");
            }
        }
    }
}

```

上述代码通过使用 **XmlDocument** 类遍历节点，使用 **XmlDocument** 类遍历节点，首先需要创建一个 **XmlDocument** 对象，并使用 **Load** 方法加载一个 **XML** 文档，示例代码如下所示。

```

XmlDocument doc = new XmlDocument();           //创建 XmlDocument 对象
doc.Load(Server.MapPath("newXml.xml"));        //载入 XML 文件

```

创建对象之后，则需要使用递归的方法遍历显示每个节点。在遍历节点的过程中，需要对每个节点进行是否有子节点的判断，如果包括子节点，则先输出子节点，如果没有子节点则继续输出根节点。

**XmlDocument** 对象也可以向 **XML** 文档中添加一个新的元素，示例代码如下所示。

```

XmlDocument doc = new XmlDocument();           //创建 XmlDocument 对象
doc.Load(Server.MapPath("newXml.xml"));        //载入 XML 文件
XmlNode node = doc.DocumentElement;           //创建节点对象
node.RemoveChild(node.FirstChild);            //移除根节点

```

上述代码使用了 **XmlDocument** 对象的 **Load** 方法载入 **XML** 文档，当需要插入 **XML** 数据时，则先需要移除根节点，移除根节点之后就能够开始添加节点，示例代码如下所示。

```

XmlNode Shop = doc.CreateElement("Shop");      //创建节点 Shop
XmlNode shop1 = doc.CreateElement("Name");     //创建节点 Name
XmlNode shop2 = doc.CreateElement("Phone");    //创建节点 Phone
XmlNode shop3 = doc.CreateElement("Seller");   //创建节点 Seller
XmlNode NameText = doc.CreateTextNode("NameText"); //创建节点文本
XmlNode PhoneText = doc.CreateTextNode("PhoneText"); //创建节点文本
XmlNode SellerText = doc.CreateTextNode("SellerText"); //创建节点文本
shop1.AppendChild(NameText);                   //添加文本
shop2.AppendChild(PhoneText);                  //添加文本
shop3.AppendChild(SellerText);                 //添加文本
Shop.AppendChild(shop1);                      //添加 Shop 子节点
Shop.AppendChild(shop2);                      //添加 Shop 子节点
Shop.AppendChild(shop3);                      //添加 Shop 子节点
node.AppendChild(Shop);                       //添加 Shop 节点

```

上述代码分别为节点添加子节点，并为子节点添加文本，添加完成后，需要使用 **XmlDocument** 对象的 **Save** 方法进行保存，示例代码如下所示。

```
doc.Save("newXml.xml");
```

使用 **XmlDocument** 对象的 **Save** 方法即可将 **XML** 内容保存在 **XML** 文档中。使用 **XmlDocument** 对象不仅能够读取，新增 **XML** 文档，还支持修改、删除等操作，例如使用 **PrependChild** 和 **InsertBefore**，**InsertAfter** 等方法进行新增和删除节点和子节点操作。

## 14.3 XML 串行化

使用 **XML** 串行化能够方便 **XML** 的存储或传输，能够把一个对象的公共域和属性保存为一种串行格式的过程，反串行化则是使用串行的状态信息将对象从串行 **XML** 状态还原为初始状态的过程。**.NET Framework** 提供了命名空间来简化开发人员进行 **XML** 串行化。

### 14.3.1 XmlSerializer 串行化类

**.NET Framework** 提供了 **System.Runtime.Serialization** 和 **System.Xml.Serialization** 以提供串行化功能。而 **System.Xml.Serialization** 提供了一个 **XmlSerializer** 类，该类可以将一个对象串行和反串行化为 **XML** 格式。**XmlSerializer** 类虽然能够执行串行化和反串行化，但是串行化和反串行化有一个最大的区别，就是串行化调用 **Serialize** 方法，而反串行化调用 **Deserialize** 方法。

如果需要将一个对象进行 **XML** 串行化，可以在类前添加**[Serializable()]**标识，声明一个定制串行化属性，如果需要将整个类都能够支持串行化，则必须添加该属性。**XML** 串行化还包括以下属性：

- ❑ **[XmlRoot]**：用来识别作为 **XML** 文件根元素的类或结构，可以用此标记把一个元素的名称设置为根元素。
- ❑ **[XmlElement]**：共有的属性或字段可以作为一个元素被串行化到 **XML** 结构中。
- ❑ **[XmlAttribute]**：共有的属性或字段可以作为一个属性被串行化到 **XML** 结构中。
- ❑ **[XmlIgnore]**：共有的属性或字段不包括在串行化的 **XML** 结构中。
- ❑ **[XmlArray]**：共有的属性或字段可以作为一个元素数组被串行到 **XML** 结构中。
- ❑ **[XmlArrayItem]**：用来识别可以放到一个串行化数组中的类型。

### 14.3.2 基本串行化

基本串行化是指让**.NET Framework** 自动的对对象进行串行化操作。使用基本串行化进行操作，只要求对象拥有类属性**[Serializable()]**即可。如果类中的某些属性或字段不需要进行串行化操作，则使用**[NonSerializable()]**属性即可，示例代码如下所示。

```
using System.Xml.Serialization;
namespace Ser
{
    [Serializable()] //设置串行化属性
    class MySer
    {
        private string Shop { get; set; } //设置 Shop 属性
        private string Name { get; set; } //设置 Name 属性
        private string Phone { get; set; } //设置 Phone 属性
        private string Seller { get; set; } //设置 Seller 属性
        [NonSerialized()] //设置非串行化
        private string Age { get; set; } //设置 Age 属性
    }
}
```

上述代码使用了**[Serializable()]**属性声明一个类，则该类的成员都能够被串行化，而 **Age** 属性使用了

[NonSerializable()]属性声明了该属性不应被串行化。使用[NonSerializable()]能够精确的控制串行化。若需要将类进行 XML 串行，则可以通过使用[XmlAttribute]等属性进行 XML 串行化操作，示例代码如下所示。

```
using System.Xml.Serialization;
namespace Ser
{
    [Serializable()] //设置串行化
    [XmlRoot("SerXML")]
    class MySer
    {
        [XmlElement("Shop")] //设置节点属性
        private string Shop { get; set; } //设置 Shop 属性
        [XmlElement("Name")] //设置节点属性
        private string Name { get; set; } //设置 Name 属性
        [XmlElement("Phone")] //设置节点属性
        private string Phone { get; set; } //设置 Phone 属性
        [XmlElement("Seller")] //设置节点属性
        private string Seller { get; set; } //设置 Seller 属性
        [NonSerialized()] //设置非串行化
        [XmlElement("Age")] //设置节点属性
        private string Age { get; set; } //设置 Age 属性
    }
}
```

上述代码运行后的结果形式呈现为 XML 格式，其格式如下所示。

```
<Shop>this.Shop</Shop>
<Name>this.Name</Name>
<Phone>this.Phone</Phone>
<Seller>this.Seller</Seller>
//<Age></Age>
```

在串行化结果输出时，并没有输出 Age 标签，是因为在 Age 属性前定义了[NonSerializable()]以控制该字段不会被串行化输出。

## 14.4 XML 样式表 XSL

XSL 是 XML 的样式表语言，这种定义很像 HTML 中的 CSS。XSL 转换就是 XSLT，XSLT 是 XSL 标准中的重要组成部分，它可以把一个 XML 文档的数据以不同结构或格式转换为另一个文档，通过使用 XSL 能够将 XML 进行格式化输出。

### 14.4.1 XSL 简介

与 HTML 样式相比，XML 样式要更加复杂，HTML 中标记的含义都是固定的，而 XML 允许开发人员能够自行创建标签，所以用样式控制 XML 会比在 HTML 控制更加复杂。

#### 1. XSL 与 HTML

HTML 样式控制通常是使用 CSS 层叠样式表进行控制，而 XML 中的样式控制通常使用 XSL 文档进行控制，HTML 中的 CSS 和 XML 中的 XSL 有以下特征：

- ❑ CSS: 用于 HTML 样式控制，由于 HTML 标签事先是规定好的，所以浏览器知道如何显示 HTML 样式，如在 HTML 文档中，<table></table>标签能够以表格的形式呈现在客户端浏览器。
- ❑ XSL: 用于 XML 样式控制，由于 XML 的标签不是事先规定好的，所以 XML 文档中的标记并不能被浏览器理解，如 XML 文档中的<table></table>并不会被浏览器解释成表格。

相比于 CSS 而言，XSL 能够作为 XML 文件的样式表而存在，而 XSL 又不仅仅需要提供样式控制，还

需要提供 **XML** 文档的方法等，**XSL** 包括 **3** 部分，这 **3** 部分分别为：

- ❑ 转换 **XML** 文档的方法。
- ❑ 定义 **XML** 部分和模式的方法。
- ❑ 格式化 **XML** 文档的方法。

## 2. XSLT

**XSLT** 翻译为可扩展样式语言转换，**XSL** 包含三种语言，其中最重要的是 **XSLT**。**XSL** 转换实际上就是 **XSLT**，在 **Visual Studio 2008** 中，可以直接创建 **XSLT** 文件对 **XML** 文件进行样式控制，定义部分和模板方法。

**XSLT** 可以将一个 **XML** 文档的数据以不同的结构或格式转换为另一个文档格式，如 **HTML**。为了让 **XML** 文件能够格式化输出到浏览器并能够进行样式控制，**XSLT** 能够对 **XML** 进行样式控制，通过编写模板，以及简易的编程控制就能够读取 **XML** 中节点的数据并重新组织，当用户访问 **XML** 文件时，能够同 **HTML** 一样被浏览器解释并呈现到客户端浏览器。

## 3. XSLT 与 XSL

**XSLT** 是 **XSL** 中最重要的语言，也是 **XSL** 中最重要的组成部分，简而言之，**XSL** 通过 **XSLT** 将一个 **XML** 源中的数据重新组织并呈现成另一种 **XML** 样式。

### 14.4.2 使用 XSLT

使用 **XSL** 对 **XML** 进行样式控制和格式化 **XML** 文档，首先需要创建一个 **XML** 文档，这里 **XML** 文档代码如下所示。

```
<?xml version="1.0" encoding="utf-8" ?>
<Root>
  <ShopInformation area="Asia">
    <Shop place="Shanghai" value="Wuhan">
      <Name>上海电脑</Name>
      <Phone>123456789</Phone>
      <Seller>J.Dan</Seller>
      <Seller>Bill Gates</Seller>
    </Shop>
    <Shop place="Wuhan">
      <Name>广埠屯</Name>
      <Phone>123456789</Phone>
      <Seller>Bill Gates</Seller>
    </Shop>
  </ShopInformation>
</Root>
```

创建完成 **XML** 文档后则需要创建 **XSL** 文档，如图 14-6 所示。





图 14-6 创建 XSLT 文件

创建 **XSLT** 文件后，系统会自动生成代码，示例代码如下所示。

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:msxsl="urn:schemas-microsoft-com:xslt" exclude-result-prefixes="msxsl">
  <xsl:output method="xml" indent="yes"/>
  <xsl:template match="@* | node()">
    <xsl:copy>
      <xsl:apply-templates select="@* | node()"/>
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

上述代码将 **XSLT** 文件的输出方法设置为 **XML**，为了能够方便对 **XML** 页面进行样式控制，可以将输出方法设置为 **HTML**，**XSLT** 文件示例代码如下所示。

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:msxsl="urn:schemas-microsoft-com:xslt" exclude-result-prefixes="msxsl">
  <xsl:output method="html" indent="yes"/>
  <xsl:template match="/Root/ShopInformation">
    <head>
      <title>
        一个 XSLT 样例
      </title>
    </head>
    <body>
      <div style="border:1px solid #ccc; padding:5px 5px 5px 5px;font-size:14px;">一个 XSLT 样例</div>
      <div style="padding:5px 5px 5px 5px;font-size:12px;">
        <xsl:value-of select="Shop"/>
      </div>
    </body>
  </xsl:template>
</xsl:stylesheet>
```

上述代码使用了 **XSLT** 文件对 **XML** 文件进行样式控制，首先需要声明 **XSLT** 文件，因为 **XSLT** 文件同样是基于 **XML** 的，所以在文件头部必须进行声明，示例代码如下所示。

```
<?xml version="1.0" encoding="utf-8"?>
```

同样 **XSLT** 也需要进行声明，示例代码如下所示。

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:msxsl="urn:schemas-microsoft-com:xslt" exclude-result-prefixes="msxsl"
>
```

声明 **XSLT** 文件后，则可以编写 **XSLT** 文件的输出属性，创建时默认为 **XML**，如果需要通过 **XSLT** 文件进行 **XML** 样式控制，则需要更改为 **HTML**，示例代码如下所示。

```
<xsl:output method="html" indent="yes"/>
```

编写 **XSLT** 文件的输出属性后，就可以编写 **XSL** 的模板，模板可以自定义标签也可以使用 **HTML** 标签。在编写模板时，需要指定模板规则的作用点，通过配置 **match** 属性可以配置模板规则的作用点，示例代码如下所示。

```
<xsl:template match="/Root/ShopInformation">
```

从 **XML** 文件可以看出，根节点为 **Root**，根节点 **Root** 下有一个 **ShopInformation** 节点，为了显示 **Shop** 节点的数据，则需要在模板规则的作用点的 **match** 属性设置路径。模板中，在需要呈现 **XML** 文档中相应节点的值可以使用 **<xsl:value-of>** 元素进行呈现，**<xsl:value-of>** 元素将拷贝 **XML** 文档中相应的节点的值到该元素，并替换后呈现给浏览器，示例代码如下所示。

```
<xsl:value-of select="Shop"/>
```

上述代码首先会通过模板路径找到相应节点，在这里使用 **select** 属性声明所要找到的节点名称，如 **<xsl:value-of select="Shop"/>**，找到 **Shop** 节点后会将 **Shop** 节点的值替换 **<xsl:value-of select="Shop"/>**，呈现

给浏览器。在 XML 文档中，需要声明外部 XSLT 文件才能在访问 XML 页面时正确的解释标签，示例代码如下所示。

```
<?xml-stylesheet type="text/xsl" href="XSLTFile1.xslt"?>
```

直接在浏览器中浏览 XSLT 文件，则可以看到 XSLT 的结构树，如图 14-7 所示。XSLT 文件制作了 XML 页面呈现时所需要的样式，从另一个角度来说，当用户在 XML 页面中定义了标签后，浏览器并不能解释这个标签，而可以通过 XSLT 文件告知浏览器如何解释自定义标签并呈现到页面，XML 文件在浏览器中运行结果如图 14-8 所示。

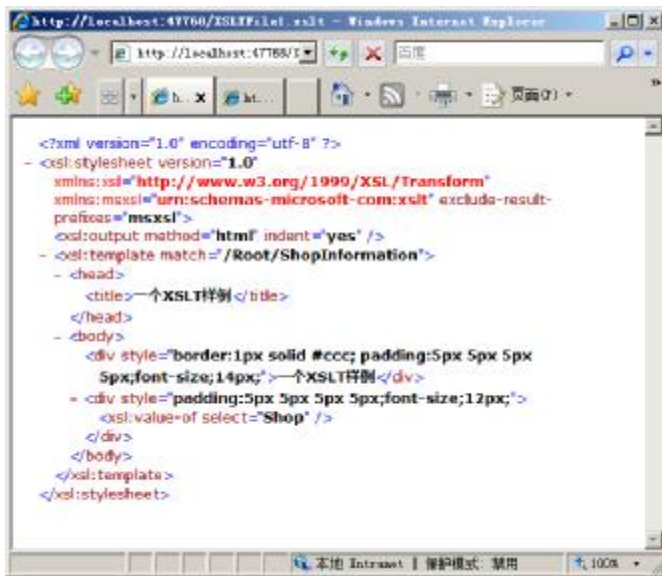


图 14-7 XSLT 结构树



图 14-8 XML 文件格式化输出

注意：IE 5.0 以下版本的浏览器很可能无法浏览结构树，如果需要在浏览器中直接浏览 XSLT 文件或 XML 文件，需要 IE 5.0 以上版本。

## 14.5 Web 服务（Web Service）

Web Service 是 Web 服务器上的一些组件，客户端应用程序可通过 Web 发出 HTTP 请求来调用这些服务。通过 ASP.NET 开发人员可以创建自定义的 Web Service 或使用内置的应用程序服务，并从任何客户端应用程序调用这些服务。

### 14.5.1 什么是 Web 服务

Web 服务（Web Service）可以被看作是服务器上的一个应用单元，它通过标准的 XML 数据格式和通用的 Web 协议为其他应用程序提供信息。Web Service 为其他应用程序提供接口从而能够实现特定的任务，其他应用程序可以使用 Web Service 提供的接口实现信息交换。

Web Service 的设计是为了解决不同平台，不同语言的技术层的差异，使用 Web Service 无论使用何种平台，何种语言都能够使用 Web Service 提供的接口，各种不同平台的应用程序也可以通过 Web Service 进行信息交互。

例如，当 Web 应用程序需要制作登录操作时，可以在 Web 页面进行登录操作设计，当 Web 应用逐渐壮大，当 Web 应用的某些应用可以发布到用户的操作系统时，就可以编写相应的应用程序来进行操作，如使用 QQ 类型的软件进行网站登录。但是这样做无疑产生了安全隐患，如果将服务器的用户名和地址等代码发布到本地，这样一些非法人员很可能能够通过反编译获取软件的信息，从而进行用户信息的盗取，而使用 Web Service，本地应用程序可以调用 Web 应用中相应的方法来实现本地登录功能，而这些方法是存在于 Web Service 中。Web Service 还具有以下特性：

- ❑ 实现了松耦合：应用程序与 **Web Service** 执行交互前，应用程序与 **Web Service** 之间的连接是断开的，当应用程序需要调用 **Web Service** 的方法时，应用程序与 **Web Service** 之间建立了连接，当应用程序实现了相应的功能后，应用程序与 **Web Service** 之间的连接断开。应用程序与 **Web** 应用之间的连接是动态建立的，实现了系统的松耦合。
- ❑ 跨平台性：**Web Service** 是基于 **XML** 格式并切基于通用的 **Web** 协议而存在的，对于不同的平台，只要能够支持编写和解释 **XML** 格式文件就能够实现不同平台之间应用程序的相互通信。
- ❑ 语言无关性：无论是用何种语言实现 **Web Service**，因为 **Web Service** 基于 **XML** 格式，只要该语言最后对于对象的表现形式和描述是基于 **XML** 的，不同的语言之间也可以共享信息。
- ❑ 描述性：**Web Service** 使用 **WSDL** 作为自身的描述语言，**WSDL** 具有解释服务的功能，**WSDL** 还能够帮助其他应用程序访问 **Web Service**。
- ❑ 可发现性：应用程序可以通过 **Web Service** 提供的注册中心查找和定位所需的 **Web Service**。

**Web Service** 也是使用和制作分布式所需的条件，使用 **Web Service** 能够让不同的应用程序之间进行交互操作，这样极大的简化了开发人员的平台的移植难度。

## 14.5.2 Web 服务体系结构

要讲到 **Web Service** 体系结构就不得不提到 **SOA**，**SOA**（**Service-Oriented Architecture**，面向服务的体系结构）是一个组件模型，它将应用程序的不同功能单元（称为服务）通过这些服务之间定义良好的接口和契约联系起来。

在 **SOA** 中，接口采用中立的方式定义，接口只声明开发人员如何继承和实现该接口，接口的声明应该是中立的、不依赖于平台、语言而实现的。接口相当于如何规定开发人员规范的进行 **Web Service** 中功能的实现。**SOA** 具有以下特点。

- ❑ **SOA** 服务具有平台独立的自我描述 **XML** 文档。**Web** 服务描述语言（**WSDL**，**Web Services Description Language**）是用于描述服务的标准语言。
- ❑ **SOA** 服务用消息进行通信，该消息通常使用 **XML Schema** 来定义（也叫做 **XSD**，**XML Schema Definition**）。

**Web Service** 体系结构则采用了 **SOA** 模型，**Web Service** 模型包含三个角色，这三个角色包括服务提供者、服务请求者和服务注册中心，如图 14-9 所示。

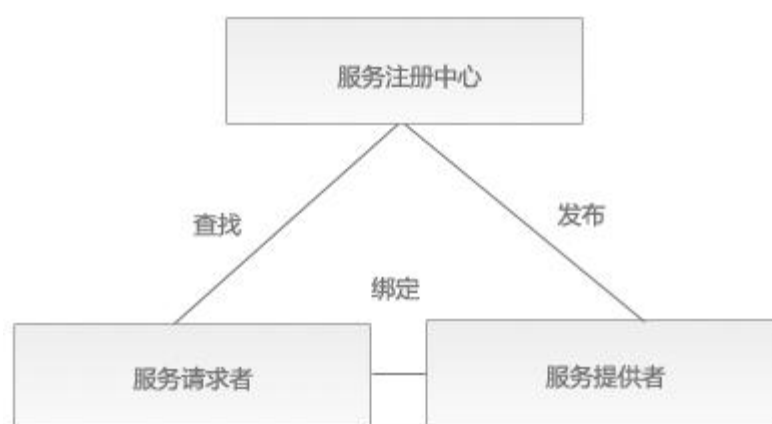


图 14-9 Web Service 体系结构

其中，服务提供者也可以称为服务的拥有者，它通过提供服务接口使 **Web Service** 在网络上是可用的。服务接口是可以被其他应用程序访问和使用的软件组件，如果服务提供者创建了服务接口，服务提供者会向服务注册中心发布服务，以注册服务描述。相对于 **Web Service** 而言，服务提供者可以看作访问服务的托管平台。

服务请求者也称为 **Web Service** 的使用者，服务请求者可以通过服务注册中心查找服务提供者，当请求者通过服务器中心查找到提供者之后，就会绑定到服务接口上，与服务提供者进行通信。相对于 **Web Service** 而言，服务请求者是寻找和调用提供者提供的接口的应用程序。



服务注册中心提供请求者和提供者进行信息通信，当服务提供者提供服务接口后，服务注册中心则会接受提供者发出的请求，从而注册提供者。而服务请求者对注册中心进行服务请求后，注册中心能够查找找到提供者并绑定到请求者。

14.5.3 Web 服务协议栈

在 Web Service 体系结构中，为了保证体系结构中的每个角色都能够正确和执行 Web Service 体系结构中的发布、查找和绑定操作，Web Service 体系必须为每一层标准技术提供 Web Service 协议栈。Web Service 协议栈如图 14-10 所示。



图 14-10 Web Service 协议栈示意图

在 Web Service 协议栈中，最为底层的是网络传输层，Web 服务协议是 Web Service 协议栈的基础。用户需要通过 Web 服务协议来调用服务接口。网络传输层可以使用多种协议，包括 HTTP、FTP 以及 SMTP。

在网络传输层上一层的则是消息传递层，消息传递层使用 SOAP 作为消息传递协议，以实现服务提供者，服务注册中心和服务请求者之间进行信息交换。

在消息传底层之上的是服务描述层，服务描述层使用 WSDL 作为消息协议，WSDL 使用 XML 语言来描述网络服务，在前面的章节中也讲到，WSDL 具有自我描述性，它能够提供 Web 服务的一些特定信息。服务描述层包括了 WSDL 文档，这些文档包括功能、接口、结构等定义和描述。

在服务描述层之上的是服务发布层，该层使用 UDDI 协议作为服务的发布/集成协议。UDDI 提供了 Web 服务的注册库，用于存储 Web 服务的描述信息。服务发布层能够为提供者提供发布 Web 服务的功能，也能够为服务请求者提供查询，绑定的功能。

当 Web Service 中触发了事件，如服务提供者发布服务接口、服务请求者请求服务等，服务提供者首先使用 WSDL 描述自己的服务接口，通过使用 UDDI 在服务器发布层向服务注册中心发布服务接口。服务注册中心则会返回 WSDL 文档。当服务请求者对服务注册中心执行服务请求，请求着通过 WSDL 文档的描述绑定相应的服务接口。

14.6 简单 Web Service 示例

在了解了 Web Service 基本的概念和协议栈的运行过程后，可以使用 Visual Studio 2008 进行 Web Service



应用程序的创建。单击菜单栏上的【文件】选项，在下拉菜单中选择【新建项目】选项，在新建项目窗口中选择【ASP.NET Web 服务应用程序】选项进行相应的应用程序创建，如图 14-11 所示。



图 14-11 创建 ASP.NET Web 服务应用程序

单击确定，系统则默认创建一个“Hello World” Web Service 应用程序，示例代码如下所示。

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Linq;
using System.Web;
using System.Web.Services; //使用 WebServer 命名空间
using System.Web.Services.Protocols; //使用 WebServer 协议命名空间
using System.Xml.Linq;
namespace _14_6
{
    /// <summary>
    /// Service1 的摘要说明
    /// </summary>
    [WebService(Namespace = "http://tempuri.org/")]
    [WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
    [ToolboxItem(false)]
    // 若要允许使用 ASP.NET AJAX 从脚本中调用此 Web 服务，请取消对下行的注释。
    // [System.Web.Script.Services.ScriptService]
    public class Service1 : System.Web.Services.WebService
    {
        [WebMethod] //声明为 Web 方法
        public string HelloWorld() //创建 Web 方法
        {
            return "Hello World";
        }
    }
}
```

在上述代码中，系统引入了默认命名空间，这些空间为 Web Service 应用程序提供基础保障，这些命名空间声明代码如下所示。

```
using System.Web;
using System.Web.Services; //使用 WebServer 命名空间
using System.Web.Services.Protocols; //使用 WebServer 协议命名空间
```

运行该 Web Service 应用程序，运行结果如图 14-12 所示。



图 14-12 Web Service 应用程序

在运行 Web Service 应用程序后，Web Service 应用程序将呈现一个页面。该页面显示了 Web Service 应用程序的名称，名称下面列举了 Web Service 应用程序中的方法。当开发人员增加方法时，Web Service 应用程序方法列表则会自动增加。创建 Web Service 应用程序方法代码如下所示。

```
[WebMethod]
public string PostMyTopic()
{
    return "Your Topic has been posted";
}
```

//声明为 Web 方法  
//创建 Web 方法  
  
//方法返回值

保存并运行后，Web Service 应用程序方法列表则会自动增加，如图 14-13 所示。单击该方法，Web Service 应用程序会跳转到另一个页面，该页面提供了方法的调用测试，以及 SOAP 各个版本请求和相应的示例，如图 14-14 所示。



图 14-13 Web Service 应用程序方法列表

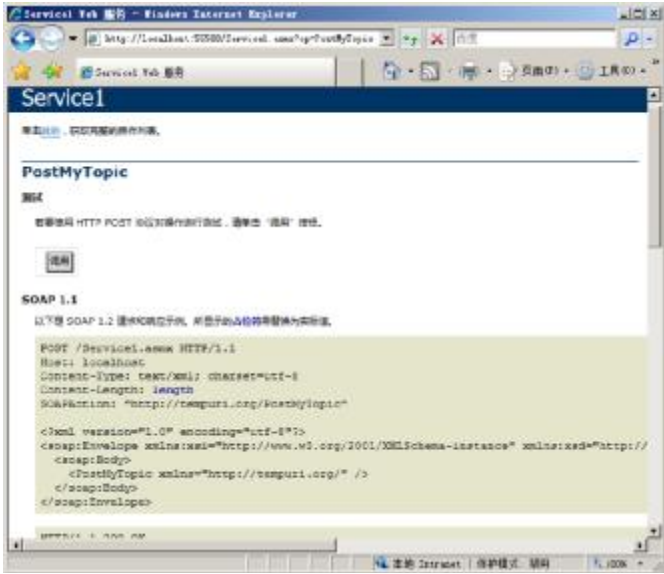


图 14-14 测试方法

单击【调用】按钮，则浏览器会通过 HTTP-POST 协议向 Web 服务递交请求信息，方法被执行完毕后，返回 XML 格式的结果，如图 14-15 所示。



图 14-15 返回结果

14.7 自定义 Web Service

在创建 Web Service 应用程序后，系统会自动创建 Web Service 应用程序并生成相关代码，通过修改自动生成的代码，能够快速创建和自定义 Web Service 应用程序，自定义 Web Service 应用程序能够让不同的应用程序引用 Web Service 提供的框架进行逻辑编程。

14.7.1 创建自定义的 Web Service

通过创建自定义 Web Service 能够进行应用程序开发，Web Service 同样支持带参数传递的方法，并能够在 Web Service 中进行数据查询等操作，保证了代码的安全性。创建一个 Web Service，并编写相应的查询方法，示例代码如下所示。

```
[WebMethod]
public string Search(string title)
{
    try
    {
        SqlConnection
        con = new SqlConnection("server=(local);database='mytable';uid='sa';pwd='sa'");
        con.Open(); //打开数据库连接
        string strsql = "select * from mynews where title like '%" + title + "%'"; //查询语句
        SqlDataAdapter da = new SqlDataAdapter(strsql, con); //创建适配器
        DataSet ds = new DataSet(); //创建数据集
        int i = da.Fill(ds, "mytable"); //填充数据集
        string result = ""; //初始化空字符串
        for (int j = 0; j < i; j++) //遍历循环数据集
        {
            result += ds.Tables["mytable"].Rows[j]["title"].ToString() + "\n"; //输出结果，生成字符串
        }
        return result; //返回结果
    }
    catch
    {
        string result = "没有任何结果";
        return result;
    }
}
```

上述代码通过创建了一个 Web Service 方法进行新闻查询，通过新闻的标题 title 字段查询数据库中索引类似信息，运行 Web Service 应用程序后，其界面如图 14-16 所示。

单击 Search 按钮进行 Web Service 应用程序测试，对于需要传递参数的方法，测试过程中可以输入方法进行测试，如图 14-17 所示。





图 14-16 自定义 Web Service 应用程序

图 14-17 输入参数

在文本框中输入 **title**，单击【调用】按钮，则会向方法中传递参数并执行方法，执行方法后将会返回 **string** 类型的值，如图 14-18 所示。



图 14-18 查询结果

在使用 **Web Service** 应用程序返回数据集时，可以直接返回 **DataSet** 对象，**Web Service** 应用程序执行后将会将 **DataSet** 对象转换为 **XML** 格式并返回 **XML** 格式的执行结果，**Search** 代码更改如下所示。

```
[WebMethod]
public DataSet Search(string title)
{
    SqlConnection con = new SqlConnection("server=(local);database='mytable';uid='sa';pwd='sa'");
    con.Open(); //打开数据连接
    string strsql = "select * from mynews where title like '%" + title + "%'"; //生成 SQL 语句
    SqlDataAdapter da = new SqlDataAdapter(strsql, con); //创建适配器
    DataSet ds = new DataSet(); //创建数据集
    int i = da.Fill(ds, "mytable"); //填充数据集
    return ds; //返回数据集
}
```

上述代码查询后直接返回 **ds** 记录集，当输入查询字串“t”后，运行结果如图 14-19 所示。

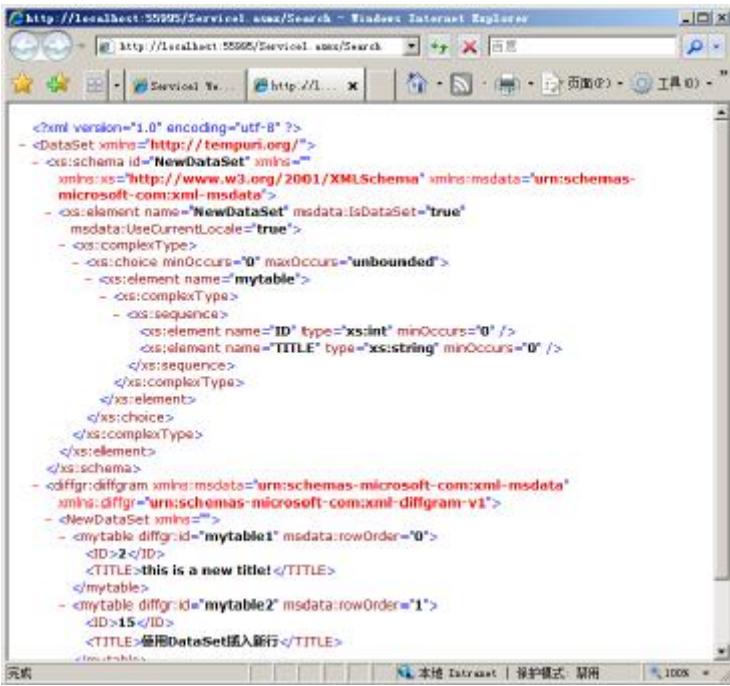


图 14-19 返回 DataSet 记录集

14.7.2 使用自定义的 Web Service

当 **Web** 应用程序需要使用 **Web Service** 应用程序并调用其方法时，只需要添加服务引用即可。右击 **Web** 应用程序，选择【添加服务引用】选项，在弹出的添加服务引用窗口中单击【发现】按钮查找服务，如图 14-20 所示。

选择相应的服务引用后并更改命名空间，再单击【确定】按钮确认添加，则服务引用添加成功，在解



决方案管理器中则会出现相应的服务引用，如图 14-21 所示。

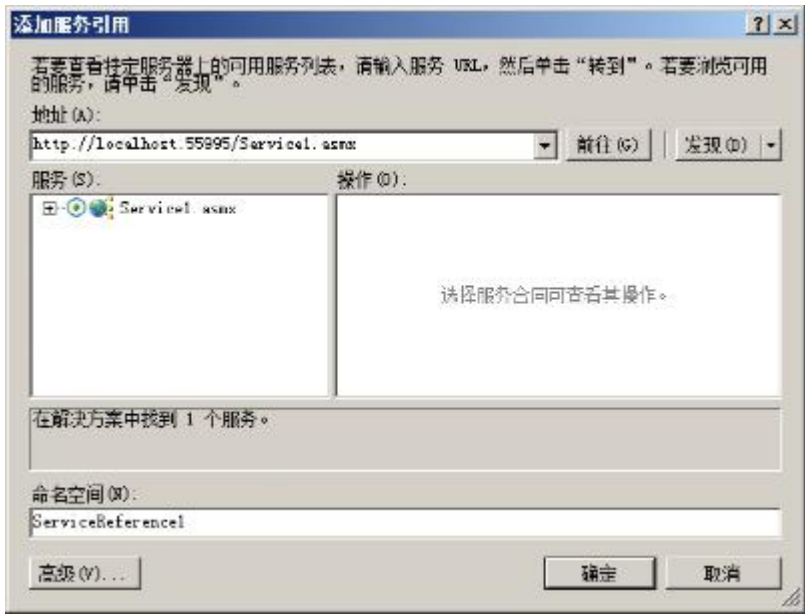


图 14-20 添加服务引用

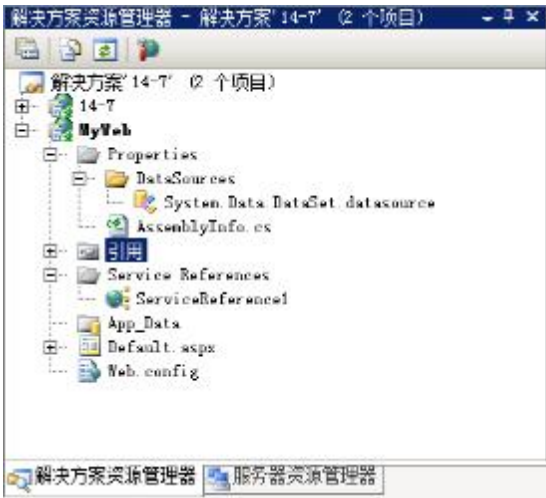


图 14-21 服务引用添加完成

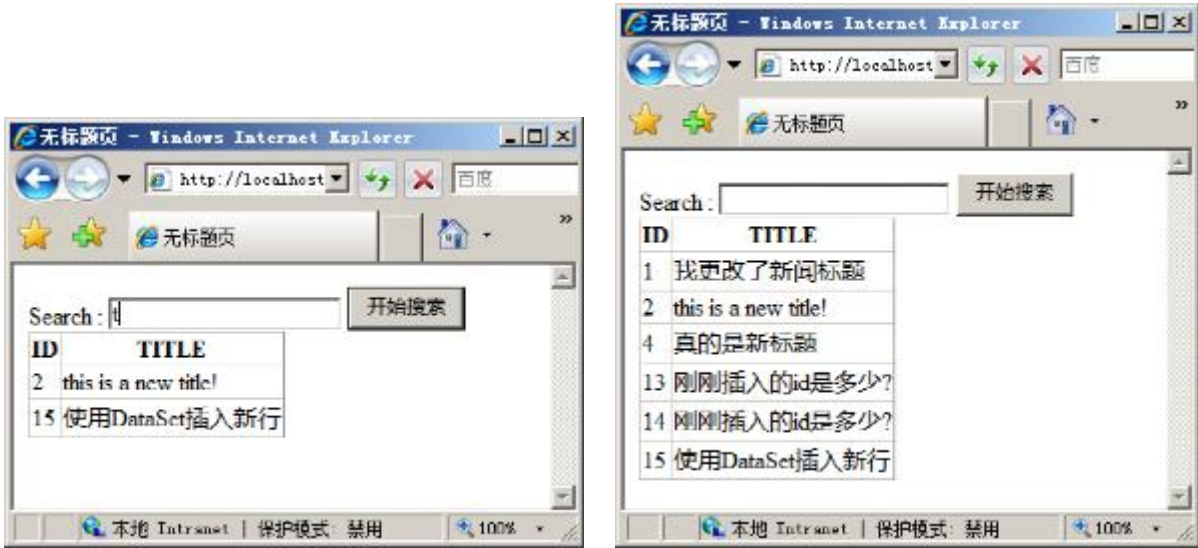
添加了服务引用之后，可以通过 **Web** 窗体使用和调用 **Web Service** 应用程序中的方法，**Web** 窗体中 **HTML** 代码如下所示。

```
<body>
  <form id="form1" runat="server">
    <div>
      Search :
      <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
      <asp:Button ID="Button1" runat="server" Text="开始搜索" onclick="Button1_Click"/>
      <br />
      <asp:GridView ID="GridView1" runat="server">
      </asp:GridView>
    </div>
  </form>
</body>
```

创建了 **Web** 应用后则需要为按钮单击事件编写代码，示例代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    ServiceReference1.Service1SoapClient
    cs = new MyWeb.ServiceReference1.Service1SoapClient()           ;//使用服务引用
    DataSet ds = new DataSet();                                     //创建数据集
    ds=cs.Search(TextBox1.Text);                                    //执行 Web Service 方法
    GridView1.DataSource = ds.Tables[0].DefaultView                 ;//绑定控件
    GridView1.DataBind();                                           //填充数据
}
```

上述代码使用了服务引用，通过服务引用进行方法的实现，运行后如图 14-22 和图 14-23 所示。



通过使用 **Web Service** 应用程序，从本地代码来看，隐藏了对数据的连接字串和查询操作代码，从应用程序角度来看，**Web Service** 应用程序保证了应用程序的封闭性和安全性。

## 14.8 小结

本章讲解了 **XML** 文件基础，以及 **Web Service** 基础，**XML** 作为 .NET 平台下微软强推的一种标记语言技术，其作用是不言而喻的。在 **SQL Server** 以及微软的其他应用软件中，也能够经常看到 **XML** 的影子，并且 **SQL Server 2005** 已经开始尝试支持 **XML** 数据类型，这说明 **XML** 在当今世界中的运用越来越广阔，也说明在未来的应用中，**XML** 技术包含着广大的前景。通过讲解 **Web Service** 的基本概念，包括什么是 **Web Service**，以及 **Web Service** 协议栈。同时，本章还包括：

- ❑ 创建 **XML** 文档：包括如何创建 **XML** 文档。
- ❑ **XML** 控件：演示了如何使用 **XML** 控件呈现 **XML** 数据。
- ❑ **XmlTextReader** 类：讲解了 **XmlTextReader** 类操作 **XML** 文档。
- ❑ **XmlTextWriter** 类：讲解了 **XmlTextWriter** 类操作 **XML** 文档。
- ❑ **XmlDocument** 类：讲解了 **XmlDocument** 类操作 **XML** 文档。
- ❑ **XML** 串行化：讲解了 **XML** 串行化基本功能。
- ❑ **XSL** 简介：介绍了 **XSL** 与 **HTML** 的异同。
- ❑ 使用 **XSLT**：介绍了 **XSLT** 语法并通过 **XSLT** 控制 **XML** 样式。
- ❑ 什么是 **Web Service**：讲解了 **Web Service** 基本概念。
- ❑ **Web Service** 协议栈：讲解了 **Web Service** 协议栈的基本概念，以及 **Web Service** 是如何运作的。

本章还简单的讲解了 **XML** 串行化功能和通过程序创建 **Web Service** 示例，虽然串行化和 **Web Service** 只做了基本的讲解，但是熟练的掌握这些基础能够为今后的分布式开发打下良好的基础。

## 第五篇 ASP.NET 3.5 高级编程

第 15 章 图形图像编程

第 16 章 ASP.NET 3.5 和 AJAX

第 17 章 ASP.NET MVC 基础

第 18 章 WCF 开发基础

第 19 章 WPF 开发基础

## 第 15 章 图形图像编程

在 **Web** 应用中，良好的图形图像的运用能够提升网站的友好度和易用性。**.NET Framework** 提供了图形图像编程的方法，开发人员可以运用图形图像编程技术进行良好的 **Web** 应用中图形图像布局和编程，也可以通过 **GDI+** 实现类似 **Photoshop** 的功能。

### 15.1 图形图像基础

使用图形图像可以进行良好的页面布局，在现有的很多 **Web** 应用中，其应用程序的页面布局经常需要使用图像，这样能够让页面整体效果更加友好。用户会对界面友好的应用程序印象深刻从而会进行回访。**ASP.NET** 不仅能够进行图形图像显示，还能够使用 **GDI+** 进行图形图像的绘制。

#### 15.1.1 图像布局

在页面布局中，很多设计人员喜欢使用 **CSS** 设计，这样能够简化页面代码，将页面布局代码和页面代码相分离，从而提高了维护性。虽然随着技术的发展，越来越多的动态生成页面布局，以及动态生成图像的方法也越来越多的被开发人员和设计人员所认知，但是开发人员和设计人员还是比较喜欢使用 **CSS** 和 **IMG** 标签进行页面布局，这是因为 **CSS** 和 **IMG** 标签都比较简单，可以说是“轻量级”的，即不需要页面进行逻辑处理也不需要动态生成。

##### 1. IMG 标签

**IMG** 标签是图像标签，**IMG** 标签属于 **HTML** 控件，在 **Web** 应用中可以看到在页面中包含大量的 **IMG** 标签用于图形图像显示，示例代码如下所示。

```
<body>
  
</body>
```

上述代码显示了一个名为 **autom** 的 **JPG** 图像到页面中，如图 15-1 所示。



图 15-1 插入图片



使用 **IMG** 标签能够轻松的为网页添加图片，**IMG** 标签包括以下常用属性：

- ❑ **Src**: 图片的地址，可以是图片的相对地址也可以是绝对地址。
- ❑ **Width**: 设定图片的宽度。
- ❑ **Height**: 设定图片的高度。
- ❑ **Alt**: 当图片显示不了时提示的字符。
- ❑ **Border**: 图片的边框的宽度。
- ❑ **Align**: 图片的周片文字的对齐方式。
- ❑ **Title**: 当鼠标放在图片上出现的提示字符。

开发人员能够通过编写 **Width** 和 **Height** 属性进行图像的大小控制，也可以编写 **Alt** 属性当图片显示不了时进行提示，如图 15-2 和图 15-3 所示。



图 15-2 Width 属性



图 15-3 Alt 属性

## 2. CSS

通过 **CSS** 能够使用图像进行页面布局和样式控制。当需要使背景呈现渐变效果时，无需使用 **JavaScript** 进行控制，可以直接使用 **CSS** 和图像进行搭配使用即可。**CSS** 背景属性包括：

- ❑ 背景颜色属性 (**background-color**)：该属性为 **HTML** 元素设定背景颜色。
- ❑ 背景图片属性 (**background-image**)：该属性为 **HTML** 元素设定背景图片。
- ❑ 背景重复属性 (**background-repeat**)：该属性和 **background-image** 属性连在一起使用，决定背景图片是否重复。如果只设置 **background-image** 属性，没设置 **background-repeat** 属性，在缺省状态下，图片既 **x** 轴重复，又 **y** 轴重复。
- ❑ 背景附着属性 (**background-attachment**)：该属性和 **background-image** 属性连在一起使用，决定图片是跟随内容滚动，还是固定不动。
- ❑ 背景位置属性 (**background-position**)：该属性和 **background-image** 属性连在一起使用，决定了背景图片的最初位置。
- ❑ 背景属性 (**background**)：该属性是设置背景相关属性的一种快捷的综合写法。

为了方便进行页面背景布局，可以使用 **CSS** 背景属性，示例代码如下所示。

```
<body style="background:#6094d7 url('bg.jpg') repeat-x;">
    <div style="width:800px; border:1px solid #333333; margin:0px auto; background:white">
        编写内容
    </div>
</body>
```

上述代码将 **BODY** 标签样式编写为背景颜色为#6094d7、背景图片编写为 **bg.jpg** 并且背景图片按照 **x** 轴重复。在 **body** 标签下方包含一个 **DIV** 标签，该标签作为主样式进行内容编写，编写内容后如图 15-4 所示。



图 15-4 使用图形进行布局

3. JavaScript 进行图像编程

HTML 图像控件支持 JavaScript 进行图像操作，可以为图像控件进行事件处理，JavaScript 代码如下所示。

```
<script type="text/javascript">
    function cut()
    {
        var pic=document.getElementById("pic1")
        pic.width=100;
        pic.height=100;
    }
</script>
```

//获取 ID 为 pic1 的图片的属性  
//设置图片的宽度  
//设置图片的高度

上述代码获取图片 ID 为 pic1 的图片属性，当触发该事件后，ID 为 pic1 的图片的宽度和高度将变为 100。为了让图片被单击时触发该事件，则应该在 IMG 标签中声明该事件，图片相应的代码如下所示。

```

```

上述代码定义了事件 onclick 并定义了图片 id 为 pic1，运行后如图 15-5 和图 15-6 所示。



图 15-5 原始图片



图 15-6 触发 JavaScript 事件后

使用 IMG 标签进行图形图像编程是非常简单的方法。IMG 标签和 CSS 配合能够进行页面布局，IMG 标签和 JavaScript 进行配合能够进行图形图像的处理，所以 IMG 标签是非常容易被学习和使用的。在 Web 应用开发中，大量的 IMG 被使用也说明 IMG 标签是一种高效率、门槛低的图形图像编程方法。

15.1.2 GDI+简介

虽然通过 IMG 标签和 CSS、JavaScript 相配合能够进行图形图像开发，但是其功能有限，并不能够进行

高级的图形图像开发。高级的图形图像开发有点类似与 **Photoshop** 中图形处理，在 **ASP.NET** 中，可以使用强大的 **GDI+** 进行图形图像开发，实现类似 **Photoshop** 中图形处理的功能。

**GDI+** 是 **Windows XP** 中的一个子系统，它主要负责在显示屏幕和打印设备输出有关信息，它是一组通过 **C++** 类实现的应用程序编程接口。**GDI+** 的前身是 **GDI**，在 **C++** 应用程序开发中，**C++** 开发人员经常需要使用 **GDI** 进行窗口的绘制与重绘，在 **Vista** 操作系统之后的操作系统中，微软对图形图像编程进行了更新，在 **Vista** 等系统中，大量的使用了半透明、渐变、边缘模糊化等效果，这就要求在编程中强化图形图像渲染，如图 15-7 所示。

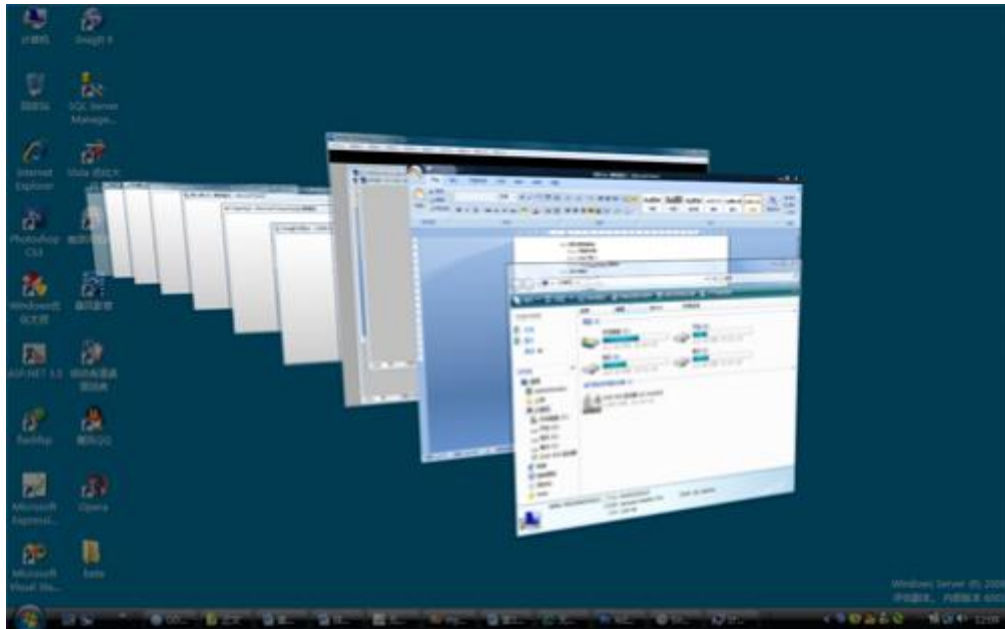


图 15-7 Vista 特效

为了适应更高的用户要求和用户体验的要求，微软对 **GDI** 进行了升级，就成为 **GDI+**。**GDI+** 不仅能够在 **C++** 开发当中使用，也能够使用 **GDI+** 强大的绘图效果。**GDI+** 相比与 **GDI**，进行了一些加强，这些加强功能如下所示。

- ❑ 渐变的画刷 (**Gradient Brushes**)：**GDI+** 允许开发人员使用渐变的画刷来绘制线条、图形以及外观。
- ❑ 基数样条函数 (**Cardinal Splines**)：**GDI+** 支持基数样条函数而 **GDI** 不支持，基数样条能够防止锯齿的出现，使得窗口以及图形的绘制能够平滑过渡。
- ❑ 持久路径对象 (**Persistent Path Objects**)：在 **GDI** 中，绘制路径在窗口更改需要通过重绘来保持图形的持久化，而在 **GDI+** 中，可以通过创建对象来持久化。
- ❑ 变形和矩阵对象 (**Transformations & Matrix Object**)：**GDI+** 提供了强大的矩阵对象，开发人员可以通过矩阵对象进行图形的翻转、平移和缩放。
- ❑ 可伸缩区域 (**Scalable Regions**)：**GDI+** 允许在一定的范围内进行任何图形变换。

**GDI+** 不仅包括这些新特性，还包括混合以及等多种图像类型支持等特性。**ASP.NET** 相对于 **ASP** 的强大之处就在于 **ASP.NET** 可以使用 **GDI+** 进行图形图像编程，实现不同的 **Web** 应用功能。

## 15.1.3 绘制线条示例

通过 **GDI+** 能够在 **Web** 应用中绘制线条，如果需要使用 **GDI+**，则首先需要引用命名空间 **System.Drawing**，示例代码如下所示。

```
using System.Drawing; //使用绘图命名空间
```

使用了命名空间后，就能够使用 **System.Drawing** 中的方法进行线条绘制，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    Bitmap MyImage = new Bitmap(400, 400); //创建 Bitmap 对象
    Graphics gr = Graphics.FromImage(MyImage); //创建绘图对象
    Pen pen = new Pen(Color.Green, 10); //创建画笔对象
    gr.Clear(Color.WhiteSmoke); //格式化画布
```



```
gr.DrawLine(pen, 50, 200, 400,20);           //绘制直线
MyImage.Save(Response.OutputStream,System.Drawing.Imaging.ImageFormat.Jpeg);    //输出
MyImage.Dispose();                             //释放对象
gr.Dispose();
}
```

上述代码绘制了使用 **GDI+**的 **Graphics** 类的对象进行线条的绘制，当页面被载入时则会开始绘制线条。当线条开始绘制时，首先需要创建一个 **Bitmap** 对象，**Bitmap** 对象处理由像素定义的图像对象，示例代码如下所示。

```
Bitmap MyImage = new Bitmap(400, 400);           //创建 Bitmap 对象
```

上述代码创建了一个 **Bitmap** 对象并定义该对象区域为 **400\*400**，**Bitmap** 就像画布。**Graphics** 对象能够在该区域内绘制图形图像，示例代码如下所示。

```
Graphics gr = Graphics.FromImage(MyImage);       //创建绘图对象
```

定义了 **Graphics** 对象后就需要创建 **Pen** 对象进行绘制，**Pen** 对象就像画笔，能够在画布上绘制图形，示例代码如下所示。

```
Pen pen = new Pen(Color.Green, 10);             //创建画笔对象
```

上述代码定义了画笔的颜色和宽度，定义画笔后就需要清除整个绘图面并进行背景颜色填充，示例代码如下所示。

```
gr.Clear(Color.WhiteSmoke);                     //格式化画布
```

填充背景后，就能够使用 **Graphics** 的 **DrawLine** 方法进行线条绘制，示例代码如下所示。

```
gr.DrawLine(pen, 50, 200, 400,20);             //绘制直线
```

上述代码则会在“画布”上绘制线条，运行后如图 15-8 所示。

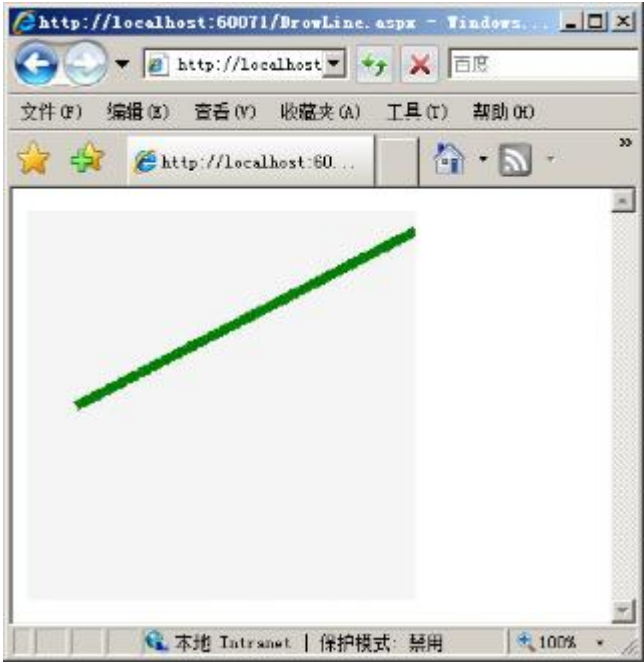


图 15-8 绘制线条

通过使用 **GDI+**的 **Graphics** 类能够在页面中绘制图形，**Graphics** 类还能够处理图片，实现锐化、底片等效果，**Graphics** 类将在后面的章节详细讲到。

15.1.4 .NET Framework 绘图类

**GDI+**包括很多的类，结构和枚举用于为开发人员提供快速进行图形图像开发提供保障和指导。

1. 命名空间

**GDI+**包括很多的类、结构和枚举用于为开发人员提供图形编程，这些类、结构和枚举都定义在命名空间中，这些命名控件如下所示。

- ❑ **System.Drawing**: 提供对 **GDI+**基本图形图像功能的访问，**Graphics** 包含在此命名空间中。
- ❑ **System.Drawing.Drawing2D**: 提供高级的二维和矢量图形功能。
- ❑ **System.Drawing.Imaging**: 提供高级的图像处理功能。



- ❑ **System.Drawing.Text**: 提供高级的文字处理及排版功能。
- ❑ **System.Drawing.Printing**: 提供图形打印所需要的类。
- ❑ **System.Drawing.Design**: 提供开发 UI 设计时所需要的类。

这些命名空间为开发人员提供了图形图像编程的基本保障，其中最常用的是 **System.Drawing**，该命名空间提供了 **Graphics** 类进行图形图像处理。**System.Drawing.Drawing2D** 提供了高级的二维图形和矢量图形的处理功能，使用 **System.Drawing.Drawing2D** 能够进行二维图形和二维游戏的开发和编写。

**System.Drawing.Imaging** 命名空间主要提供了图像处理的功能，例如将图像进行锐化处理，或者将图像变成黑白色或底片都可以通过使用 **System.Drawing.Imaging** 命名空间的方法。**System.Drawing.Text** 命名空间提供了文字处理能力，通过 **System.Drawing.Text** 类能够实现 Word 中艺术字的效果。

## 2. 类和方法

在 .NET Framework 中，包括诸多的命名空间以保证开发人员能够快速地进行图形图像开发，在这些命名空间中，最常用的是 **System.Drawing** 命名空间，该命名空间提供的类如下所示。

- ❑ **Bitmap**: 在 **Bitmap** 上使用图形工具，并在其中存储图形图像的绘图面板。
- ❑ **Graphics**: 提供直线、曲线、多边形等绘画方法，也提供对一些位图的处理，例如平移、缩放等。
- ❑ **Pen**: 提供直线、曲线等功能需要的画笔属性。
- ❑ **Brush**: 提供文本填充和图形绘画，可以填充图形如圆形、椭圆形和多边形。
- ❑ **Color**: 提供颜色的枚举，用于定义 **Pen** 和 **Brush** 的颜色。
- ❑ **Font**: 提供文本的字体属性，定义文本的字体类型、样式和大小等。
- ❑ **Point**: 用于定义有序的坐标对，这些坐标能够定义二维平面上的点。
- ❑ **Size**: 定义区域的大小。
- ❑ **Image**: 用于支持位图、指针和图标等文件类型。
- ❑ **Rectangle**: 用于定义矩形区域。
- ❑ **StringFormat**: 用于定义文本在位图上的对齐方式等属性。

简而言之，**Bitmap** 就相当于绘画时需要的纸，图形能够绘画到纸上面。而 **Graphics** 相当于绘画的人，因为人能够提供只写、曲线、多边形等绘画方法，而 **Pen** 和 **Brush** 相当于绘画工具，如铅笔、笔刷等，**Color** 就相当于是绘画所需要的颜料。

在绘画过程中，首先需要使用一张纸，固定到绘画板上，然后有一个人能够进行绘画，这个人能够进行素描、水彩等绘画。但是在绘画前，需要给这个人基本的工具，包括铅笔、笔刷和颜料盘等。在这些基本物质准备完毕后，就能够开始绘制了。**GDI+** 的绘制过程与之非常的相似，首先需要定义一个画布，并通过构造函数定义画布的大小，示例代码如下所示。

```
Bitmap MyImage = new Bitmap(500, 500); //创建 Bitmap 对象
```

上述代码定义了一个大小为 **500\*500** 的画布，定义了画布之后，就需要有一个人进行绘画操作，示例代码如下所示。

```
Graphics gr = Graphics.FromImage(MyImage); //创建 Graphics 对象
```

上述代码定义了一个 **Graphics** 对象并指定该对象在 **MyImage** 画布上进行绘画，定义了 **Graphics** 对象后，需要给该对象指定工具，示例代码如下所示。

```
Pen pen = new Pen(Color.Green, 10); //创建画笔
```

上述代码为对象指定了一个绿色的宽度为 **10** 的画笔，指定画笔后，这个“人”就能够进行绘画了。示例代码如下所示。

```
gr.DrawLine(pen, 50, 200, 400, 200); //绘制直线
MyImage.Save(Response.OutputStream, System.Drawing.Imaging.ImageFormat.Jpeg); //输出
```

通过 **DrawLine** 方法进行直线的绘画，绘画完毕需要执行 **Save** 方法把图片保存到文件中，或者使用 **Response.OutputStream** 将图片输出到 HTTP 流中在客户端浏览器中呈现。

## 15.2 图形编程

通过上面的章节了解了什么是 **GDI+**，以及如何使用 **GDI+** 进行图形图像编程。在介绍了 **.NET Framework** 绘图类所需要的命名空间和方法后，就需要了解如何使用相应的命名空间和方法进行图形图像的绘制和处理。

### 15.2.1 Graphics 类

**Graphics** 类在 **GDI+** 的开发过程中非常重要，**Graphics** 类封装了 **GDI+** 界面画图方法，以及图形显示设备，极大的简化了开发人员的编程过程。

#### 1. Graphics 类的属性

**Graphics** 类的属性如下所示。

- ❑ **DpiX**: 获取对象的水平分辨率。
- ❑ **DpiY**: 获取对象的垂直分辨率。
- ❑ **IsClipEmpty**: 为对象指定裁剪区域。
- ❑ **IsVisibleClipEmpty**: 判断裁剪区域是否为空。
- ❑ **TextGammaValue**: 返回一个提供文本灰度值的信息的整数值。
- ❑ **TextRenderingHint**: 获取或设置与该图形相关联的文本着色模式。

通过 **Graphics** 类的属性能够获取 **Graphics** 对象的水平分辨率和垂直分辨率，并能够为 **Graphics** 对象进行裁剪区域的选择和判断。

#### 2. Graphics 类的方法

**Graphics** 类提供的属性通常用于 **Graphics** 对象的信息获取，如果需要使用 **Graphics** 对象进行图形图像的绘制，则需要使用 **Graphics** 类提供的方法，**Graphics** 类的部分方法如下所示。

- ❑ **Dispose**: 删除图形并释放已分配的内存。
- ❑ **DrawArc**: 绘制弧线。
- ❑ **DrawBezier**: 绘制后三次贝塞尔曲线。
- ❑ **DrawClosedCurve**: 绘制封闭曲线。
- ❑ **DrawCurve**: 绘制曲线。
- ❑ **DrawEllipse**: 绘制椭圆。
- ❑ **DrawIcon**: 绘制图标图像。
- ❑ **DrawIconUnstretched**: 绘制图标图像，并可将图像缩放到指定大小。
- ❑ **DrawImage**: 绘制图像。
- ❑ **DrawImageUnscaled**: 绘制图像，并可将图像缩放到指定大小。
- ❑ **DrawImageUnscaledAndClipped**: 在不进行缩放的情况下进行图像绘制。
- ❑ **DrawLine**: 绘制线条。
- ❑ **DrawLines**: 绘制一系列线条组。
- ❑ **DrawPath**: 绘制 **GraphicsPath**。
- ❑ **DrawPie**: 绘制扇形。
- ❑ **DrawPolygon**: 绘制多边形。
- ❑ **DrawRectangle**: 绘制矩形。
- ❑ **DrawString**: 绘制字符串。
- ❑ **Equals**: 判断两个 **Object** 类型是否相同。
- ❑ **FillClosedCurve**: 填充封闭曲线的内部区域。
- ❑ **FillEllipse**: 填充椭圆内部。
- ❑ **FillPath**: 填充 **GraphicsPath** 内部。

- ❑ **FillPie**: 填充扇形内部。
- ❑ **GetHdc**: 获取图形上下文设备句柄。
- ❑ **Restore**: 恢复图形状态。
- ❑ **Save**: 保存图形。
- ❑ **SetClip**: 为对象设置剪辑区域。

**Graphics** 类还包括其他方法提供对图形图像的绘制和处理进行编程,开发人员能够通过.NET Framework 提供的 **Graphics** 类进行图形任何图形图像的编程。开发人员不仅能够可以绘制现有图形,还可以对图形进行渲染处理,这些渲染处理的效果能够应用到图形、文字和图表上。

15.2.2 绘制基本图形

通过使用 **Graphics** 类,开发人员能够绘制基本图形,这些基本图形包括线条、矩形、椭圆和多边形。

1. 绘制线条

在上面的章节中讲解了如何进行线条的绘制,这里再进行简单的介绍,能够加深对 **Graphics** 类的理解,示例代码如下所示。

```
Bitmap images = new Bitmap(200, 200);           //创建画纸
Graphics gr = Graphics.FromImage(images);        //创建 Graphics 对象
Pen pen = new Pen(Color.Red, 5);                //创建画笔
gr.Clear(Color.White);                          //设置画笔的颜色
gr.DrawLine(pen, 0, 0, 200, 200);               //开始绘画
images.Save(Response.OutputStream, System.Drawing.Imaging.ImageFormat.Jpeg);
gr.Dispose();                                   //释放绘图对象
images.Dispose();                               //释放图形对象
```

上述代码使用了 **DrawLine** 方法进行直线的绘制,其中 **DrawLine** 包括五个需要传递的参数,这五个此参数分别为起点的坐标和终点的坐标以及画笔。在上述代码中,图像大小为 **200\*200**,并在画布中从 **(0, 0)** 坐标开始绘画到终点坐标为 **(200, 200)** 的直线,绘制后的效果如图 15-9 所示。

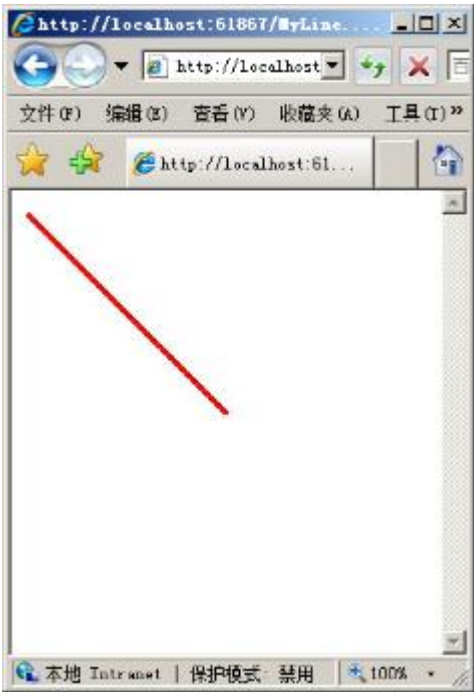


图 15-9 绘制线条形

2. 绘制矩形

绘制矩形的方法同绘制线条基本相同,但是绘制矩形不仅要指定矩形的坐标,还需要指定矩形的高度和宽度,示例代码如下所示。

```
Bitmap images = new Bitmap(400, 400);           //创建画纸
Graphics gr = Graphics.FromImage(images);        //创建 Graphics 对象
Pen pen = new Pen(Color.Red, 5);                //创建画笔
```

```
gr.Clear(Color.White); //设置画笔颜色
gr.DrawLine(pen, 0, 0, 200, 200); //绘制线条
gr.DrawRectangle(pen, 200, 200, 50, 50); //绘制矩形
images.Save(Response.OutputStream, System.Drawing.Imaging.ImageFormat.Jpeg);
gr.Dispose();
images.Dispose();
```

从上面的代码不难看出，绘制矩形的方法同绘制线条的方法基本相同，绘制矩形使用了 **DrawRectangle** 方法进行绘制，**DrawRectangle** 方法同样包括五个参数，这五个参数同绘制线条有一点不同。绘制线条需要指定绘制过程中开始坐标和终点坐标，而绘制矩形时需要指定开始坐标，矩形的高度和宽度以及所需的画笔，示例代码如下所示。

```
gr.DrawRectangle(pen, 200, 200, 50, 50); //绘制矩形
```

上述代码绘制了一个矩形，其开始坐标为（200，200），并且高度和宽度都为 50 的矩形，运行后如图 15-10 所示。

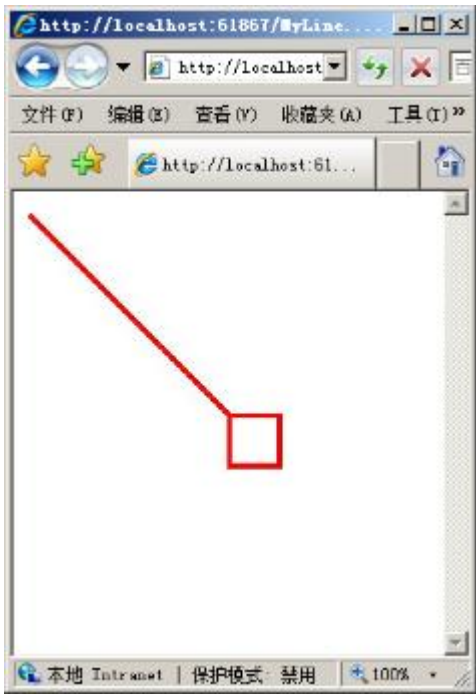


图 15-10 绘制矩形

3. 绘制圆形和椭圆

绘制椭圆的方法只需要使用 **DrawEllipse** 方法即可，示例代码如下所示。

```
gr.DrawEllipse(pen, 0, 0, 300, 200); //绘制椭圆
```

上述代码绘制了一个椭圆形，该椭圆形绘制的起点为（0，0），宽度为 300，高度为 200。**DrawEllipse** 方法同 **DrawRectangle** 方法基本相同，因为这两个方法都包括五个参数，这 5 个参数都需要指定绘制起点、宽度和高度。当需要绘制圆形时，只需要将宽度和高度设置相等即可，示例代码如下所示。

```
gr.DrawEllipse(pen, 0, 0, 200, 200); //绘制圆
```

当设置宽度和高度相等时，该椭圆就会以圆形呈现，上述代码就实现了圆形的绘制。

4. 绘制多边形

绘制多边形的方法只需要使用 **DrawPolygon** 方法即可，与绘制规则图形不同的是，绘制多边形需要指定多边形的各个节点，**DrawPolygon** 方法通过获取这些节点即可组成一个多边形，示例代码如下所示。

```
Point pt1 = new Point(50, 50); //设置节点
Point pt2 = new Point(150, 150); //设置节点
Point pt3 = new Point(200, 200); //设置节点
Point pt4 = new Point(350, 170); //设置节点
Point pt5 = new Point(90, 30); //设置节点
Point pt6 = new Point(180, 90); //设置节点
Point[] pts = { pt1, pt2, pt3, pt4, pt5, pt6 }; //设置节点组
gr.DrawPolygon(pen, pts); //绘制多边形
```

**Point** 对象表示一个二维坐标中的一个点，通过 **Point** 的默认构造函数能够在二维坐标中指定一个点，示例代码如下所示。



```
Point pt1 = new Point(50, 50); //设置节点
```

定义了平面上一个点之后，需要为多边形定义多个点，多个点能够组成一个多边形，这些点组成一个二维坐标的集合，示例代码如下所示。

```
Point[] pts = { pt1, pt2, pt3, pt4, pt5, pt6 }; //设置节点组
```

上述代码定义了一个 **Point** 类型的数组声明一个二维坐标的集合，通过 **DrawPolygon** 方法即可定义绘制多边形，示例代码如下所示。

```
gr.DrawPolygon(pen, pts); //绘制多边形
```

5. 绘制文字

通过使用 **DrawString** 方法能够绘制文字并呈现在图像中，示例代码如下所示。

```
Font font = new Font("宋体", 20); //创建文字对象
Brush brush=new SolidBrush(Color.Red); //创建笔刷对象
gr.DrawString("我的字符串", font, brush, 200,200); //绘制文字
```

使用 **DrawString** 方法，需要对 **DrawString** 方法进行参数传递，**DrawString** 方法需要五个参数，其中包括需要输出的字符串、文本格式对象、笔刷对象以及文字开始绘制的坐标。上述代码中，输出字符串为“我的字符串”。文本格式通过 **Font** 默认构造函数构造，并在坐标为(200,200)位置开始绘制。

15.2.3 图形绘制实例

上面一节讲解了如何分别绘制图形，使用相应的方法能够绘制相应的图形。**.NET Framework** 为图形图像开发进行了封装，使用**.NET Framework** 进行图形图像开发非常简单，下面绘制一些基本图形，并填充基本图形，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    Bitmap images = new Bitmap(400, 400); //创建画纸
    Graphics gr = Graphics.FromImage(images); //创建绘图类
    Pen pen = new Pen(Color.Red, 5); //创建画笔
    gr.Clear(Color.White); //绘制直线
    gr.DrawLine(pen, 0, 0, 200, 200); //绘制矩形
    gr.DrawRectangle(pen, 200, 200, 50, 50); //绘制椭圆
    gr.DrawEllipse(pen, 0, 0, 300, 200); //绘制多边形
    Point pt1 = new Point(50, 50); //设置节点
    Point pt2 = new Point(150, 150); //设置节点
    Point pt3 = new Point(200, 200); //设置节点
    Point pt4 = new Point(350, 170); //设置节点
    Point pt5 = new Point(90, 30); //设置节点
    Point pt6 = new Point(180, 90); //设置节点
    gr.DrawPolygon(pen, pts); //绘制文字
    Font font = new Font("宋体", 20); //设置字体大小
    Brush brush=new SolidBrush(Color.Red); //创建红色笔刷
    gr.DrawString("我的字符串", font, brush, 200,200); //填充矩形
    SolidBrush brush2 = new SolidBrush(Color.YellowGreen);
    gr.FillRectangle(brush2,new Rectangle(100,100,100,100)); //填充矩形
    images.Save(Response.OutputStream, System.Drawing.Imaging.ImageFormat.Jpeg);
    gr.Dispose();
    images.Dispose();
}
```

上述代码分别绘制了直线、矩形、椭圆、文字和多边形，并填充了矩形，运行后如图 15-11 所示。



图 15-11 图形绘制实例

正如图 15-20 所示，开发人员能够使用.NET Framework 提供的绘图类进行基本图形的绘制，不仅如此，.NET Framework 绘图类还提供了文字和图片的特效绘制方法。

### 15.3 绘制文字特效

在 Word 中，文本编辑人员经常使用艺术字进行 Word 编辑和排版，艺术字在很大程度上丰富了排版功能和艺术效果，通过使用.NET Framework 绘图类能够实现文字的艺术化效果从而丰富页面中的文本显示效果。

#### 15.3.1 投影特效

使用 System.Drawing.Drawing2D 和 System.Drawing.Text 能够进行文字投影特效。在制作文字投影特效前，首先需要使用命名空间 System.Drawing.Drawing2D 和 System.Drawing.Text。在实现投影效果前，首先需要了解如何制作投影。

投影特效的难度在于如何描述本体的影子。其实在画面上，影子是不可能像平常的描述一样呈现在图片上的，这也就是说，影子其实也是本体对象的另一种表现形式。首先，影子可以看作是本体的压缩和平移，在对本体进行压缩和平移后，从一定的角度上看就好像是本体的影子。其次，影子是没有颜色的，通常用灰色输出即可实现影子的效果。在制作投影特效时，需要使用到 Matrix 类，该类需要使用 System.Drawing.Drawing2D 和 System.Drawing.Text 命名空间，示例代码如下所示。

```
using System.Drawing;                                     //使用绘图类
using System.Drawing.Drawing2D;                           //使用绘图类
using System.Drawing.Text;
```

引用了命名空间后，就可以使用相应的方法进行特效的制作，示例代码如下所示。

```
Bitmap images = new Bitmap(600, 150);                     //创建 Bitmap 对象
Graphics gr = Graphics.FromImage(images);                 //创建 Graphics 对象
gr.Clear(Color.WhiteSmoke);                               //填充背景颜色
gr.TextRenderingHint = TextRenderingHint.ClearTypeGridFit; //设置文本输出质量
gr.SmoothingMode = SmoothingMode.AntiAlias;
Font newFont = new Font("宋体", 32);
Matrix matrix = new Matrix();                             //执行投射
matrix.Shear(-1.5f, 0.0f);                                //执行缩放
matrix.Scale(1, 0.5f);                                    //执行平移
```

```
matrix.Translate(130, 88); //执行坐标转换
gr.Transform = matrix;
SolidBrush grayBrush = new SolidBrush(Color.Gray);
SolidBrush colorBrush = new SolidBrush(Color.Red);
string text = "ASP.NET 3.5 开发大全"; //设置文字
gr.DrawString(text, newFont, grayBrush, new PointF(0, 30)); //绘制阴影
gr.ResetTransform(); //图形变形
gr.DrawString(text, newFont, colorBrush, new PointF(0, 40)); //绘制前景
images.Save(Response.OutputStream, System.Drawing.Imaging.ImageFormat.Jpeg);
```

上述代码使用了 **Matrix** 类进行了倒影效果的实现，**Matrix** 类封装表示几何变换的 **3 x 3** 仿射矩阵。**Matrix** 类中常用的方法有：

- ❑ **Shear**: 通过预先计算比例向量，将指定的比例向量应用到此矩阵。
- ❑ **Scale**: 通过预先计算切变向量将指定的切变向量应用到此矩阵。
- ❑ **Translate**: 通过预先计算转换向量，将指定的转换向量应用到此矩阵。

使用 **Matrix** 类能够进行对象的投射、缩放以及平移，并通过执行坐标转换呈现在图片中。作为投影特效，**Matrix** 类通过将现有的对象进行转换、压缩、平移，并通过 **Graphics** 对象的 **DrawString** 方法进行输出，使之看上去向文字的投影效果一样，如图 15-12 所示。



图 15-12 实现文字的投影效果

## 15.3.2 倒影特效

在 **Photoshop** 中，可以很容易的实现倒影效果，通过 **GDI+** 同样也能够快捷的实现倒影效果。在实现倒影效果时，首先需要理清倒影是如何实现的，倒影同影子一样。在编程中，没有倒影的概念，其实倒影就是本体的另一种表现形式，与投影不同的是，投影是将本体进行压缩或拉伸，从一个角度来看成为影子的效果，而倒影其实就是本地的垂直旋转。

在进行垂直旋转后，倒影通常情况下颜色要比本体要淡，这样才能真实的模拟倒影的效果，实现倒影效果示例代码如下所示。

```
Brush shadowBrush = Brushes.LightBlue; //创建倒影笔刷
Brush foreBrush = Brushes.Blue; //创建本体笔刷
Font font = new Font("微软雅黑", Convert.ToInt16(40), FontStyle.Italic); //配置字体
Bitmap images = new Bitmap(600, 150);
Graphics gr = Graphics.FromImage(images); //创建 Graphice
gr.Clear(Color.WhiteSmoke);
string text = "ASP.NET 3.5 开发大全"; //设置文字
SizeF size = gr.MeasureString(text, font); //设置矩形大小
int posX = (600 - Convert.ToInt16(size.Width)) / 2; //设置平移坐标
int posY = (150 - Convert.ToInt16(size.Height)) / 2; //设置平移坐标
gr.TranslateTransform(posX, posY); //执行转换
GraphicsState state = gr.Save(); //图形保存
gr.ScaleTransform(1, -1.0F); //图形变换
```



```
gr.DrawString(text, font, shadowBrush, 0, -120); //输出倒影
gr.Restore(state); //图形重置
gr.DrawString(text, font, foreBrush, 0, 0); //输出本体
images.Save(Response.OutputStream, System.Drawing.Imaging.ImageFormat.Jpeg);
```

倒影效果相对与投影效果来说比较简单，倒影效果只是将 **Graphics** 对象的矩形大小进行转换，然后通过平移操作进行翻转操作，翻转完成后直接输出倒影，而又因为倒影输出的文字比本体输出的文字要淡，看上去很像倒影的效果，如图 15-13 所示。



图 15-13 倒影效果

15.3.3 旋转特效

通过 **GDI+**能够通过非常简单的代码实现非常酷的效果，例如旋转特效就是一个非常酷的效果，在进行旋转特效编写前，首先需要了解如何实现旋转特效的。如果要实现旋转特效，首先需要获取一段文字，该文字进行通过平移坐标原点进行变换，当需要实现旋转时，则通过循环不停的实现旋转平移，示例代码如下所示。

```
Bitmap images = new Bitmap(400, 400); //创建 Bitmap 对象
Graphics gr = Graphics.FromImage(images); //创建绘图对象
gr.Clear(Color.WhiteSmoke); //格式化画布
gr.SmoothingMode = SmoothingMode.AntiAlias; //设置边缘
for (int i = 0; i <= 360; i += 20) //循环旋转
{
    gr.TranslateTransform(200, 200); //变形
    gr.RotateTransform(i); //按角度变形
    Brush brush = Brushes.Red; //创建画笔
    Font font = new Font("微软雅黑", 12); //创建文字
    gr.DrawString("ASP.NET 3.5 开发大全 ", font, brush, 0, 0); //绘制文字
    gr.ResetTransform(); //重置变形
}
images.Save(Response.OutputStream, System.Drawing.Imaging.ImageFormat.Jpeg);
```

上述代码运行后如图 15-14 和图 15-15 所示。



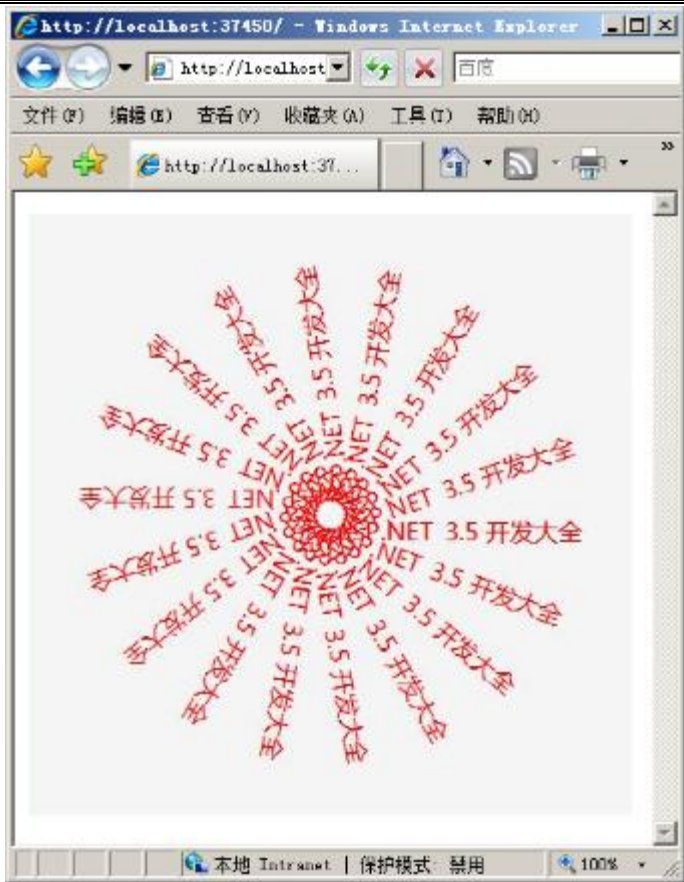


图 15-14 旋转特效

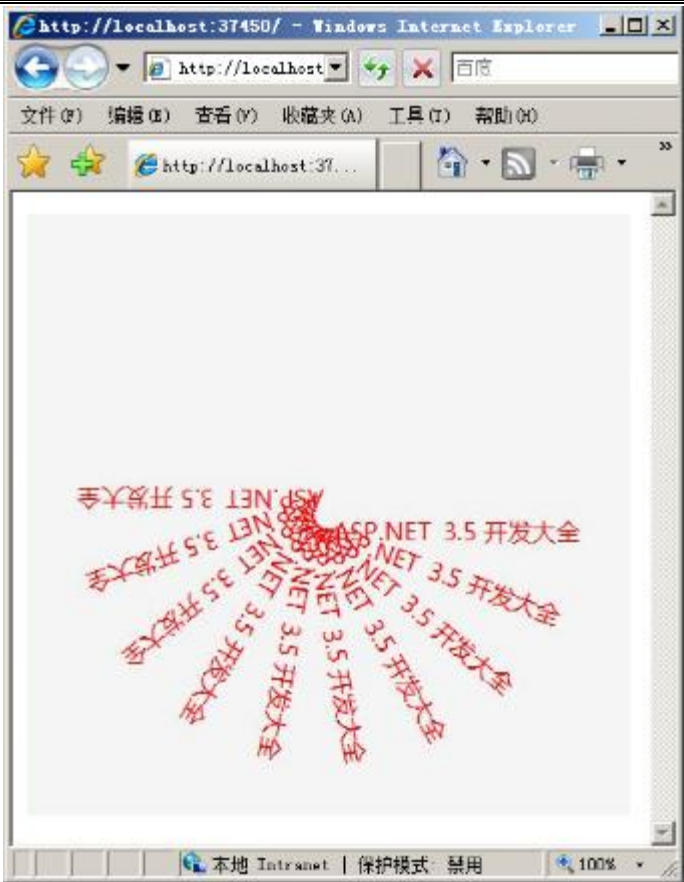


图 15-15 旋转半周

旋转特效就是按照循环不停的将本体进行循环并执行输出，在执行循环的过程中，因为圆的角度是 360 度的，所以循环时也需要循环 360 度，如果循环不被等于 360，则会出现半周的情况，如图 15-15 所示。

## 15.4 绘制图片

通过 **IMG** 标签能够插入图像，**IMG** 标签小巧而灵活，但是在如果需要使用 **GDI+** 实现图形图像的渲染，**IMG** 标签所呈现的图形显然是不行的，**ASP.NET** 提供了 **Image** 控件用来创建图片，并能够通过 **Image** 控件进行图片编程。

### 15.4.1 载入图像文件

使用 **Image** 控件能够载入图像文件，拖动一个 **Image** 控件到页面，页面会自动生成 **HTML** 代码，示例代码如下所示。

```
<body>
  <form id="form1" runat="server">
    <div>
      <asp:Image ID="Image1" runat="server" />
    </div>
  </form>
</body>
```

在控件的章节中，讲到 **Image** 控件包括以下常用属性：

- ❑ **AlternateText**: 在图像无法显示时显示的备用文本。
- ❑ **ImageAlign**: 图像的对齐方式。
- ❑ **ImageUrl**: 要显示图像的 URL。

通过配置以上三种属性能够呈现不同的图片效果，配置完成后的图像控件示例代码如下所示。

```
<asp:Image ID="Image1" runat="server" AlternateText="图片不存在"
  ImageUrl="~/autom.jpg" />
```

上述代码配置了 **Image** 控件的基本属性，包括 **Image** 控件所需要呈现的图像，以及 **Image** 控件呈现的

图像不存在时需要提示的信息，运行后如图 15-16 和图 15-17 所示。



图 15-16 Image 控件

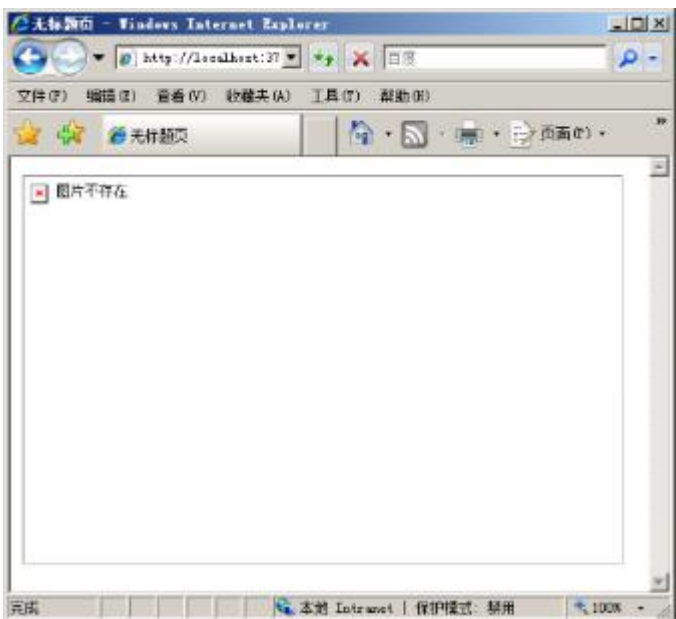


图 15-17 图片不存在

15.4.2 GDI+输出图像

使用 **Image** 控件能够快速的载入图形，但是 **Image** 控件并不支持 **Click** 事件，所以对 **Image** 控件是没有办法进行事件操作的，如果需要对图像进行事件操作并支持裁剪等高级方法时，可以采用 **GDI+**进行图形输出，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    Bitmap images = new Bitmap(Server.MapPath("autom.jpg"));           //读取现有图片
    images.Save(Response.OutputStream, images.RawFormat);             //格式化输出
    images.Dispose();                                                  //释放对象
}
```

上述代码使用了 **Bitmap** 类进行图形输出，**Bitmap** 类的默认构造方法能够载入现有的图片并执行输出。

注意：**Bitmap** 类的 **RawFormat** 属性能够直接返回现有文件的文件类型，在 **Bitmap** 的 **Save** 方法中可直接使用。

15.5 图像特效处理

相比与 **IMG** 标签而言，**ASP.NET** 能够通过 **GDI+**动态的创建图像并且进行图片特效处理。相对于文字处理而言，图片特效处理很像 **Photoshop** 中对图片的处理，开发人员能够实现不同的图片特效，如呈现底片效果、黑白效果等。

15.5.1 底片效果

通过 **Photoshop** 等软件能够快速的将图片制作成底片效果，但是在传统的图片处理领域中，只能通过软件进行图片效果的更改。在 **ASP.NET** 中，可以通过网页进行图片处理，包括底片、锐化等效果。

在进行底片效果制作前，首先需要了解底片效果是如何实现的，在图片显示中，其实是很多很多的点（像素）组成一个图片的，如果像素的数量很多，则图片显示的就清晰，如果像素数量较少，则图片看上去就不那么清晰。一个图片的组成是通过像素组成的，这也就是说，一个图片包括很多的小点进行组合，



最后组合成图片，如图 15-18 所示。

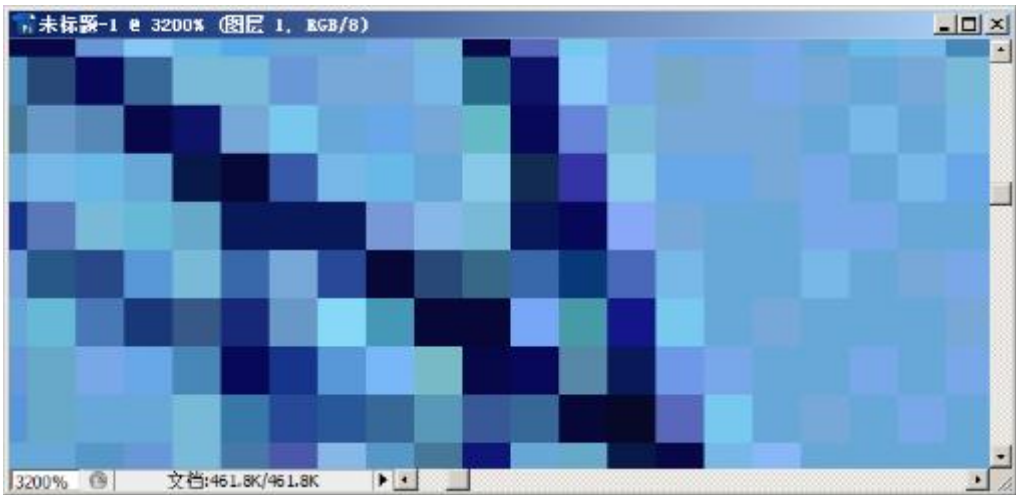


图 15-18 图片中的像素

在进行底片效果的制作时，只需要分别找到图片中的这些点，并获取这些点的像素的值，再取反保存即可，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    Bitmap images = new Bitmap(Server.MapPath("change.jpg"));           //载入图片
    for (int i = 0; i < images.Width; i++)                             //循环遍历宽
    {
        for (int j = 0; j < images.Height; j++)                         //循环遍历高度
        {
            Color pix = images.GetPixel(i, j);                         //获取图像像素值
            int r = 255 - pix.R;                                         //像素值取反
            int g = 255 - pix.G;                                         //转换颜色
            int b = 255 - pix.B;                                         //转换颜色
            images.SetPixel(i, j, Color.FromArgb(r, g, b));             //保存像素值
        }
    }
    images.Save(Response.OutputStream, System.Drawing.Imaging.ImageFormat.Jpeg);
    images.Dispose();
}
```

上述代码通过循环遍历像素值并进行像素值取反则能够实现底片效果，运行效果前后如图 15-19 和图 15-20 所示。



图 15-19 原图像

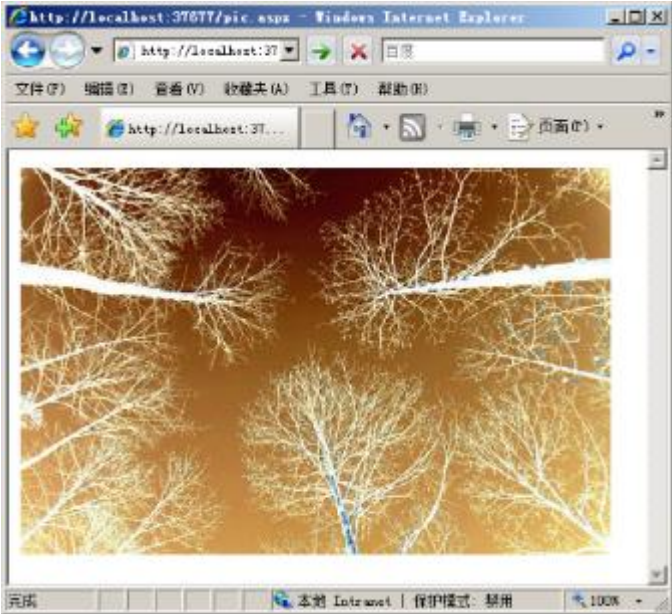


图 15-20 执行底片效果

在图像中，图像通常是使用 RGB 三原色进行图形渲染的，RGB 分别为红色、绿色和蓝色，图形通过协调这三原色进行图形渲染。RGB 在计算机中通常描述的范围是从 0-255 之间的某个数值的，所以当需要取反时，只需通过 255 减去现有的像素的色值即可。

## 15.5.2 浮雕效果

对于图片效果的更改和渲染都是通过修改像素的值来进行渲染的，当像素足够大时，人眼无法辨认其像素的多少，所以更改像素的大小对人眼来说是无法发觉的。

执行浮雕效果与底片效果实现手法非常类似，但是浮雕效果的实现与底片效果的实现中所需要使用的算法又不尽相同。实现浮雕效果通常是将图像上每个像素点与其对角线的像素点形成差值，使相似颜色值淡化，不同颜色值之间保持突出，从而形成纵深感，达到浮雕的效果。在程序开发中，可以讲像素点的像素值与周边的像素值相减后加上 **128**，则可以呈现浮雕效果，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    Bitmap images = new Bitmap(Server.MapPath("change.jpg"));           //载入图片
    for (int i = 0; i < images.Width-1; i++)                           //循环遍历宽
    {
        for (int j = 0; j < images.Height-1; j++)                       //循环遍历高度
        {
            Color pix1 = images.GetPixel(i, j);                       //获取图像像素值
            Color pix2 = images.GetPixel(i+1, j+1);                   //获取图像像素值
            int r = Math.Abs(pix1.R - pix2.R + 128);                  //实现浮雕效果
            int g = Math.Abs(pix1.G - pix2.G + 128);
            int b = Math.Abs(pix1.B - pix2.B + 128);
            r = check(r);                                             //判断是否溢出
            g = check(g);
            b = check(b);
            images.SetPixel(i, j, Color.FromArgb(r, g, b));          //设置像素值
        }
    }
    images.Save(Response.OutputStream, System.Drawing.Imaging.ImageFormat.Jpeg);
    images.Dispose();
}
```

在进行浮雕效果实现时，需要进行判断，判断转换后的值是否超过 **255** 或者小于 **0**，如果超过 **255** 则按照 **255** 进行处理，如果小于 **0** 则按照 **0** 进行处理。**check** 函数代码如下所示。

```
protected int check(int x)
{
    if (x > 255)                                                       //如果像素值大于 255
    {
        return 255;                                                  //返回 255
    }
    else if (x < 0)                                                    //如果像素值小于 0
    {
        return 0;                                                    //返回 0
    }
    else
    {
        return x;                                                     //直接返回
    }
}
```

程序运行后，图片渲染前后效果如图 15-21 和图 15-22 所示。





图 15-21 原图像

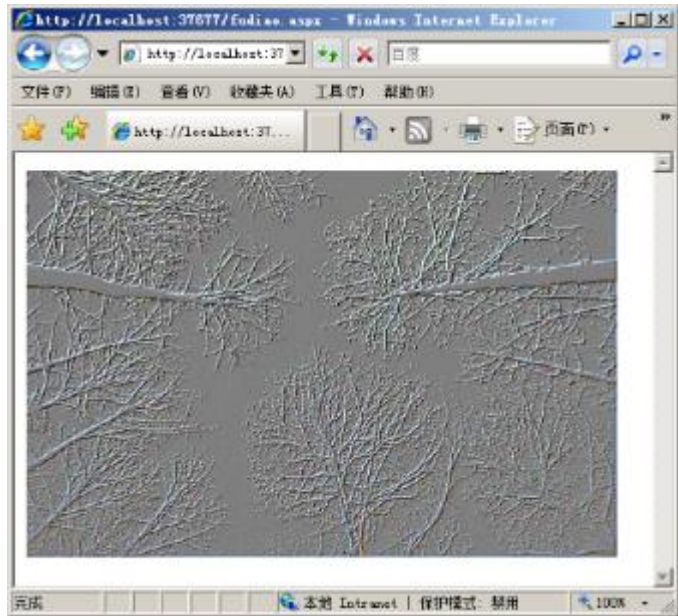


图 15-22 执行浮雕效果

## 15.6 小结

本章介绍了 **ASP.NET** 图形图像编程，通过 **ASP.NET** 图形图像编程能够在 **Web** 上执行图形图像的修改以及渲染。在页面中绘制图形图像包含很多方法，最简单的方法就是使用 **Graphics** 类中的方法进行图形的绘制，**Graphics** 类不仅提供了基本图形的绘制，还提供了图像、图标图像的绘制。**GDI+**看上去好像比较复杂，但是通过几个实例就能够了解其实 **GDI+**并不困难，在 **ASP.NET** 中，最常用的图形图像编程的方类属 **Graphics** 类了。本章还包括：

- ❑ 图形图像基础：介绍了图形图像编程基础。
- ❑ **GDI+**简介：介绍了 **GDI+**编程基础。
- ❑ **.NET Framework** 绘图类：介绍了 **.NET Framework** 绘图类。
- ❑ 图形绘制实例：进行图形图像编程，通过使用现有类进行基本图形的创建。
- ❑ 绘制文字特效：使用 **GDI+**进行文字绘制和特效渲染。
- ❑ 图像特效处理：使用 **GDI+**进行图像特效的处理。

本章还介绍了图像显示的基本原理，以及图像特效的实现思路。

## 第 16 章 ASP.NET 3.5 和 AJAX

现今，在 **Web** 开发领域最流行的就属 **AJAX**，**AJAX** 能够提升用户体验，更加方便的与 **Web** 应用程序进行交互。在传统的 **Web** 开发中，对页面进行操作往往需要进行回发，从而导致页面刷新，而使用 **AJAX** 就无需产生回发从而实现无刷新效果。

### 16.1 AJAX 基础

在 **C/S** 应用程序的开发过程中，很容易做到无“刷新”样式控制，因为 **C/S** 应用程序往往是安装在本地的，所以 **C/S** 应用程序能够维持客户端状态，对于状态的改变能够及时捕捉。相比之下，**Web** 应用属于一种无状态的应用程序，在 **Web** 应用程序操作过程中，需要通过 **POST** 等方法进行页面参数传递，这样就不可避免的会产生页面的刷新。

#### 16.1.1 什么是 AJAX

在传统的 **Web** 开发过程中，浏览者浏览一个 **Web** 页面，并进行相应的页面填写时，就需要使用表单向服务器提交信息。当用户提交表单时，就不可避免的会向服务器发送一个请求，服务器接受该请求后并执行相应的操作后将生成一个页面返回给浏览者。

然而，在服务器处理表单并返回新的页面的同时，浏览者第一次浏览时的页面（这里可以当作是旧的页面）和服务器处理表单后返回的页面在形式上基本相同，当大量的用户进行表单提交操作时，无疑是增加了网络的带宽，因为处理前和处理后的页面基本相同。

在 **C/S** 应用程序开发中，**C/S** 应用程序往往安装在本地，这样响应用户事件的时间非常的短，而且 **C/S** 应用程序可以算的上是有状态的应用程序，能够及时捕捉和相应用户的操作。而在 **Web** 端，由于每次的交互都需要向服务器发送请求，服务器接受请求和返回请求的过程就依赖于服务器的响应时间，所以给用户造成感觉要比在本地慢的多。

为了解决这一问题，通过在用户浏览器和服务器之间设计一个中间层——即 **AJAX** 层就能够解决这一问题，**AJAX** 改变了传统的 **Web** 中客户端和服务器的“请求——等待——请求——等待”的模式，通过使用 **AJAX** 应用向服务器发送和接收需要的数据，从而不会产生页面的刷新。

**AJAX** 应用通过使用 **SOAP** 或其他一些基于 **XML** 的 **web service** 接口，并在客户端采用 **JavaScript** 处理来自服务器的响应，从而减少了服务器和浏览器之间的“请求——回发”操作，从而减少了带宽。当服务器和客户端之间的信息通信减少之后，浏览者就会感觉到 **Web** 应用中的操作就更快了。

**AJAX** 将一些应用的处理交付给客户端，让服务器端原本应该运行的操作和需要处理的事务分布给客户端，这样服务器端的处理时间也减少了。

相对于传统的 **Web** 开发，**AJAX** 提供了更好的用户体验，**AJAX** 也提供了较好的 **Web** 应用交互的解决方案，相对于传统的 **Web** 开发而言，**AJAX** 技术也减少了网络带宽。**Ajax** 的核心是 **JavaScript** 对象 **XmlHttpRequest**。该对象在 **Internet Explorer 5** 中就被引入了，它是一种支持异步请求的技术。简而言之，**XmlHttpRequest** 使您可以使用 **JavaScript** 向服务器提出请求并处理响应，而不会影响客户端的信息通信。传统的 **Web** 应用和 **AJAX** 应用模型如图 16-1 所示。

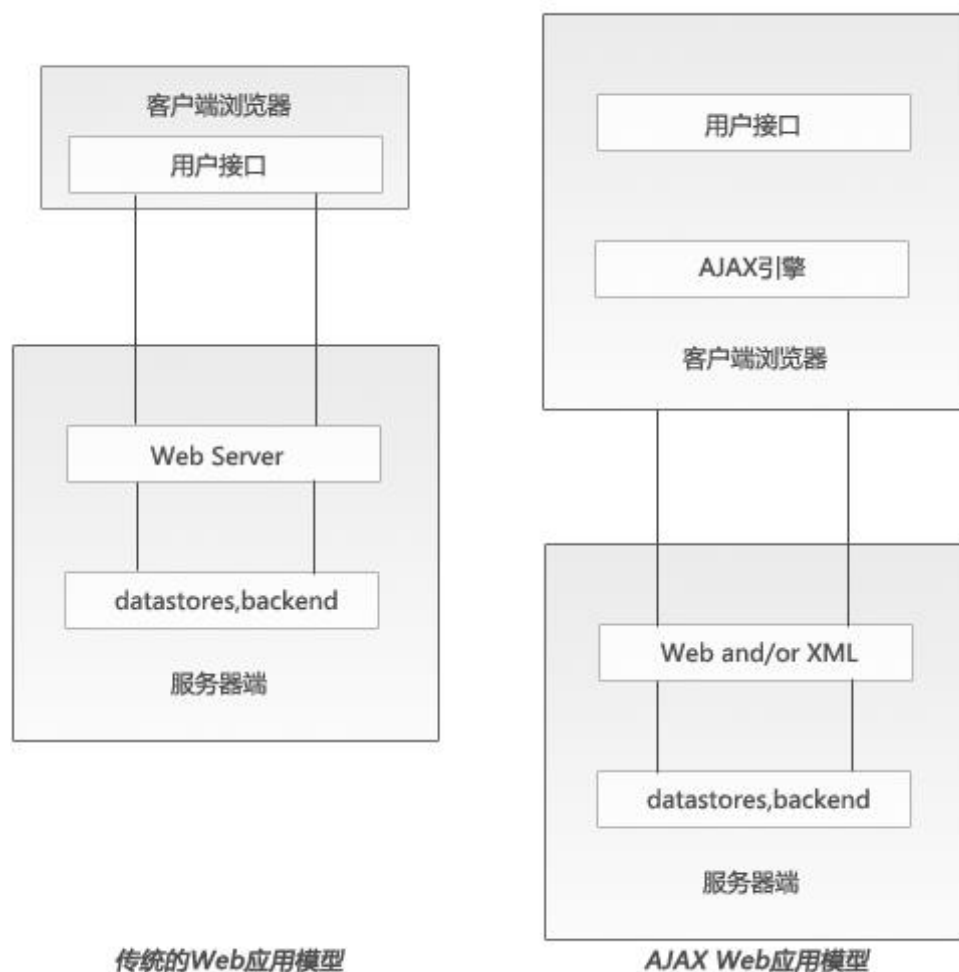


图 16-1 传统 Web 应用和 AJAX Web 应用模型

**AJAX Web** 应用模型的优点在于，无需进行整个页面的回发就能够进行局部的更新，这样能够使 **Web** 服务器能够尽快的响应用户的要求。而 **AJAX Web** 应用无需安装任何插件，也无需在 **Web** 服务器中安装应用程序，但是 **AJAX** 需要用户允许 **JavaScript** 在浏览器上执行，如图用户不允许 **JavaScript** 在浏览器上执行，则 **AJAX** 可能无法运行。但是随着 **AJAX** 的发展和客户端浏览器的发展，先进的所有的浏览器都能够支持 **AJAX**，包括最新的 **IE8**、**Firefox 4** 以及 **Opera** 等。

**AJAX** 包含诸多优点，同样也包含缺点。**AJAX** 无法维持刚刚的“历史”状态，当用户在一个页面进行操作后，**AJAX** 将破坏浏览器的功能中的“后退”功能，当用户执行了 **AJAX** 操作之后，当单击浏览器的后退按钮时，则不会返回到 **AJAX** 操作前的页面形式，因为浏览器仅仅能够记录静态页面的状态，而使用 **AJAX** 进行页面操作后，并不能改变本身页面的状态，所以单击后退按钮并不能返回操作前的页面状态。

在使用 **AJAX** 进行 **Web** 应用开发的过程中，另一个缺点就是容易造成用户体验变差。虽然 **AJAX** 能够极大的方便用户体验，但是当服务器需求变大时，当用户进行一个操作而 **AJAX** 无法及时相应时，可能会造成相反的用户体验。

例如用户阅读一个新闻时，当用户进行评论时，页面并没有刷新，但是评论这个操作已经在客户端和浏览器之间发生了，用户可能很难理解为什么页面没有显式也没有刷新，这样容易让用户变得急躁和不安，使得用户可能产生非法操作从而降低用户体验。为了解决这个问题，可以在页面明显的位置提示用户已经操作或提示请等待等操作，让用户知道页面正在运行。

相比于传统的 **Web** 应用，**AJAX** 的另一个缺点就是对移动设备的支持不够好。在当今 **iPhone** 和 **GPhone** 等智能移动设备逐渐普及的同时，**AJAX** 并不能很好的支持这些设备，这也需要等待 **AJAX** 技术的进一步发展。

## 16.1.2 ASP.NET AJAX 入门

**AJAX** 技术看似非常的复杂，其实 **AJAX** 并不是新技术，**AJAX** 只是一些老技术的混合体，**AJAX** 通过将这些技术进行一定的修改、整合和发扬，就形成了 **AJAX** 技术。这些老技术包括有：

- **XHTML**：基于 **XHTML1.0** 规范的 **XHTML** 技术。



- ❑ **CSS**: 基于 CSS2.0 的 CSS 布局的 CSS 编程技术。
- ❑ **DOM**: HTML DOM, XML DOM 等 DOM 技术。
- ❑ **JavaScript**: JavaScript 编程技术。
- ❑ **XML**: XML DOM、XSLT、XPath 等 XML 编程技术。

上面的这些技术并不是最新的技术，这些技术已经在现在的开发当中被普遍使用，包括 XHTML、CSS 和 DOM，开发人员能够使用 JavaScript 进行 Web 应用中 Web 编程和客户端状态维护，而通过使用 XML 技术能够进行数据保存和交换。

除了上面的一些老技术，AJAX 还包含另一个技术，这个技术就是 XMLHttpRequest。在 AJAX 中，最重要的就是 XMLHttpRequest 对象，XMLHttpRequest 对象是 JavaScript 对象，正式 XMLHttpRequest 对象实现了 AJAX 可以在服务器和浏览器之间通过 JavaScript 创建一个中间层，从而实现了异步通信。如图 16-2 所示。

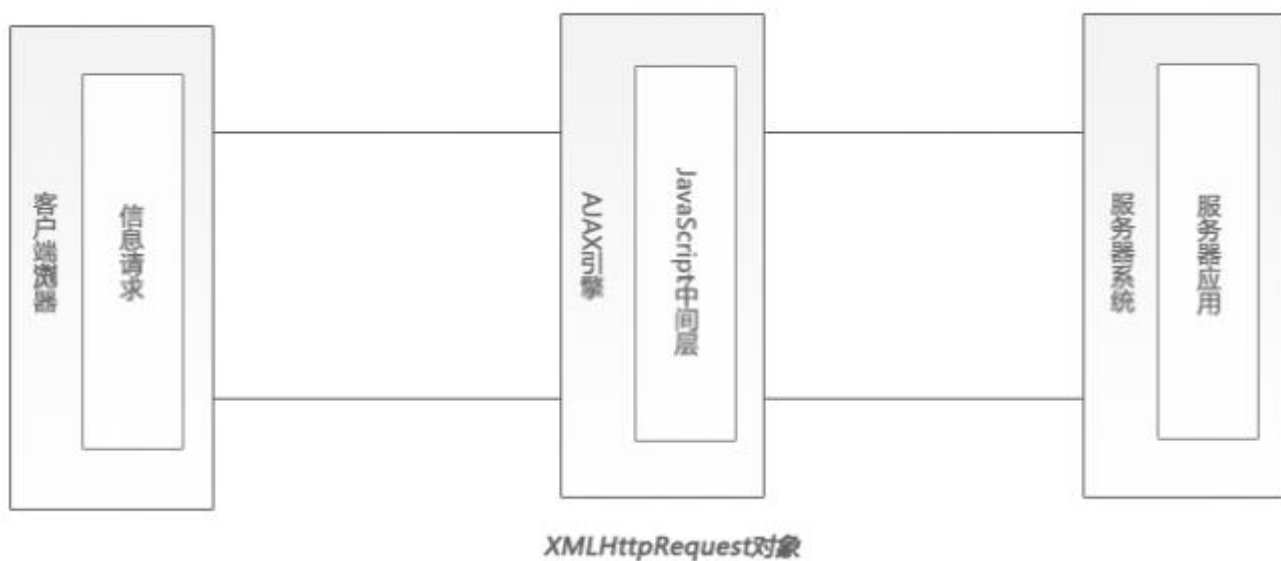


图 16-2 XMLHttpRequest 对象实现过程

AJAX 通过使用 XMLHttpRequest 对象实现异步通信，使用 AJAX 技术后，例如当用户填写一个表单，数据并不是直接从客户端发送到服务器，而是通过客户端发送到一个中间层，这个中间层可以被称为 AJAX 引擎。

开发人员无需知道 AJAX 引擎是如何将数据发送到服务器的。当 AJAX 引擎将数据发送到服务器时，服务器同样也不会直接将数据返回给浏览器，而是通过 JavaScript 中间层将数据返回给客户端浏览器。XMLHttpRequest 对象使用 JavaScript 代码可以自行与服务器进行交互。简而言之，AJAX 技术是通过使用 XHTML、CSS、DOM 等实现的，具体实现如下所示。

- ❑ 使用 XHTML+CSS 进行页面表示表示。
- ❑ 使用 DOM 进行动态显示和交互。
- ❑ 使用 XML 和 XSLT 进行数据交换。
- ❑ 使用 XMLHttpRequest 进行异步数据查询、检索。
- ❑ 使用 JavaScript 进行页面绑定。

## 16.1.3 ASP.NET 2.0 AJAX

在 ASP.NET 3.5 之前，ASP.NET 并不是原生的支持 AJAX 应用，所以在 ASP.NET 3.5 之前使用 ASP.NET AJAX 并不是一件容易的事情。同时由于国内服务器和主机的限制，导致很多应用无法直接基于 ASP.NET 3.5 而进行创建和开发。

ASP.NET 2.0 是现在国内最为成熟的 .NET 技术平台，在 ASP.NET 2.0 中同样可以使用 AJAX 技术实现无页面刷新效果。在这之前，首先需要下载一个 ASP.NET 2.0 AJAX 安装程序，通过此安装程序能够在应用程序中安装 AJAX 环境，包括 AJAX 技术以及 AJAX 控件。



技巧：ASP.NET 2.0 AJAX 安装程序可以通过访问 ASP.NET AJAX 中文站点进行下载，（<http://www.ajaxasp.net.cn/Download/Files/ASPAJAXExtSetup.msi>）

下载 ASP.NET 2.0 AJAX 安装程序后，双击安装程序即可安装，如图 16-3 所示。安装完成后会在 C:\Program Files\Microsoft ASP.NET\ASP.NET 2.0 AJAX Extensions 目录下生成文件 System.Web.Extensions.dll、System.Web.Extensions.Design.dll 以及 AJAXExtensionsToolbox.dll 等文件。这些文件都是在 ASP.NET 2.0 中使用 AJAX 的必要文件，开发人员能够将这些文件复制到 Web 应用的根目录下再添加引用即可，如图 16-4 所示。

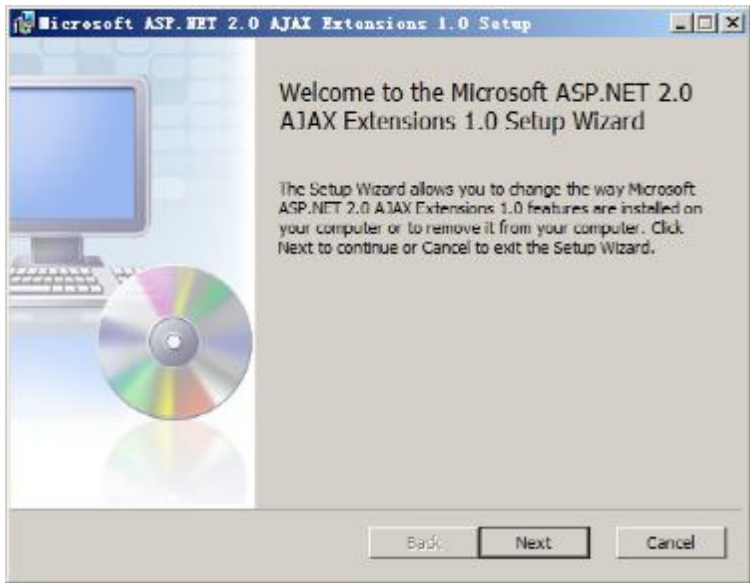


图 16-3 安装 ASP.NET 2.0 AJAX

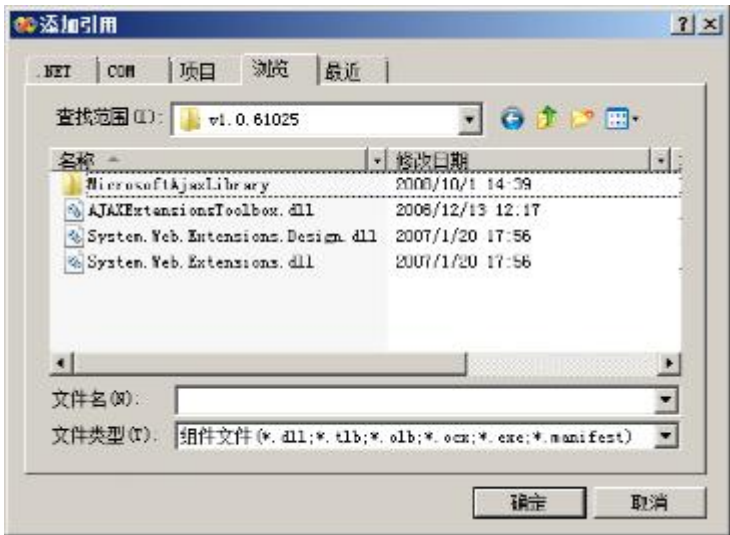


图 16-4 添加 ASP.NET 2.0 AJAX 引用

将这三个 DLL 文件添加引用到应用程序后，Web.config 文件将会更改，示例代码如下所示。

```
<pages validateRequest="false" buffer="true">
  <controls>
    <add tagPrefix="asp" namespace="System.Web.UI"
      assembly="System.Web.Extensions, Version=1.0.61025.0,
      Culture=neutral, PublicKeyToken=31bf3856ad364e35"/>
  </controls>
</pages>
```

添加引用后，可以通过添加工具栏将 AJAX 控件安装到默认工具栏中。右击工具栏空白区域，在下拉菜单中单击【选择项】选项，则会弹出【选择工具箱项】窗口，如图 16-5 所示。单击【浏览】选项，可以选择相应的 DLL 文件以添加工具箱，如图 16-6 所示。



图 16-5 选择工具箱

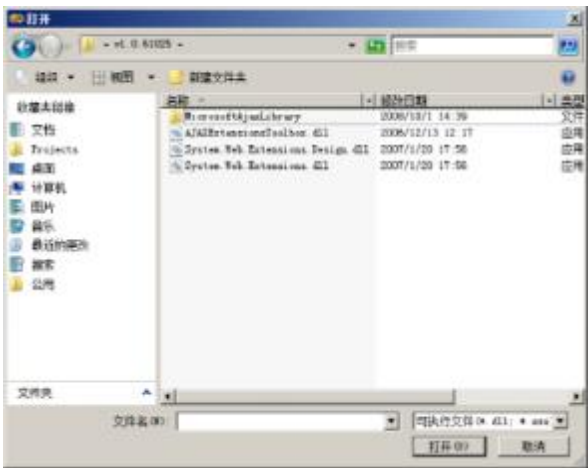


图 16-6 选择 DLL 文件

选择完成后，单击【确定】按钮就会发现在工具栏中已经包含了 AJAX 控件了。引用和工具栏添加完毕后，则需要修改 Web.config 文件从而实现 ASP.NET 2.0 对 AJAX 的支持。

注意：Web.config 文件已经保存在 ASP.NET 2.0 AJAX 应用程序下载安装包中，读者从上述连接或在官方网站下载 ASP.NET 2.0 AJAX 安装包后，其中就包含了 Web.config 文件。

## 16.1.4 ASP.NET 3.5 AJAX

在 ASP.NET 2.0 中，AJAX 需要下载和安装，开发人员还需要将相应的 DLL 文件分类存放并配置 Web.config 文件才能够实现 AJAX 功能。而在 ASP.NET 3.5 中，AJAX 已经成为 .NET 框架的原生功能。创建 ASP.NET 3.5 Web 应用程序就能够直接使用 AJAX 功能，如图 16-7 所示。



图 16-7 ASP.NET 3.5 AJAX

在 ASP.NET 3.5 中，可以直接拖动 AJAX 控件进行 AJAX 开发。AJAX 能够同普通控件一同使用，从而实现 ASP.NET 3.5 AJAX 中页面无刷新功能。在 ASP.NET 3.5 中，Web.config 文件已经被更改，并且声明了 AJAX 功能，示例代码如下所示。

```
<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>
```

在 ASP.NET 3.5 中，如果需要在 Internet 信息服务 7.0 中运行 ASP.NET AJAX 应用，则需要配置 System.webServer 配置节，示例代码如下所示。

```
<system.webServer>
  <validation validateIntegratedModeConfiguration="false"/>
  <modules>
    <remove name="ScriptModule"/>
    <add
      name="ScriptModule"
      preCondition="managedHandler"
      type="System.Web.Handlers.ScriptModule,
      System.Web.Extensions,
      Version=3.5.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35"/>
    </modules>
  <handlers>
    <remove name="WebServiceHandlerFactory-Integrated"/>
    <remove name="ScriptHandlerFactory"/>
    <remove name="ScriptHandlerFactoryAppServices"/>
    <remove name="ScriptResource"/>
    <add name="ScriptHandlerFactory" verb="*" path="*.asmx" preCondition="integratedMode"
```

```
type="System.Web.Script.Services.ScriptHandlerFactory,
System.Web.Extensions, Version=3.5.0.0, Culture=neutral,
PublicKeyToken=31BF3856AD364E35"/>
<add
    name="ScriptHandlerFactoryAppServices"
    verb="*" path="*_AppService.axd" preCondition="integratedMode"
    type="System.Web.Script.Services.ScriptHandlerFactory,
    System.Web.Extensions,
    Version=3.5.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35"/>
<add name="ScriptResource"
    preCondition="integratedMode"
    verb="GET,HEAD" path="ScriptResource.axd"
    type="System.Web.Handlers.ScriptResourceHandler, System.Web.Extensions,
    Version=3.5.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35"/>
</handlers>
</system.webServer>
```

注意：如果在 Internet 信息服务 7.0 下运行 ASP.NET AJAX 则必须使用 `system.webServer` 配置节。对早期版本的 IIS 来说则不需要配置此节。

### 16.1.5 AJAX 简单示例

虽然 AJAX 的原理听上去非常的复杂，但是 AJAX 的使用却是非常方便的。ASP.NET 3.5 提供了 AJAX 控件以便开发人员快速的进行 AJAX 应用程序开发。在进行 AJAX 页面开发时，首先需要使用脚本管理控件（**ScriptManger**），示例代码如下所示。

```
<asp:ScriptManager ID="ScriptManager1" runat="server">
</asp:ScriptManager>
```

开发人员无需对 **ScriptManger** 控件进行配置，只需保证 **ScriptManger** 控件在 **UpdatePanel** 控件之前即可。使用了 **ScriptManger** 控件之后，可以使用 **UpdatePanel** 用来确定需要进行局部更新的控件，示例代码如下所示。

```
<asp:UpdatePanel ID="UpdatePanel1" runat="server">
    <ContentTemplate>
        <asp:TextBox ID="TextBox1" runat="server" AutoPostBack="True"
            ontextchanged="TextBox1_TextChanged"></asp:TextBox>
        &nbsp;<asp:Button ID="Button1" runat="server" onclick="Button1_Click1"
            Text="Button" />
        <br />
        <asp:Label ID="Label1" runat="server" Text="Label"></asp:Label>
    </ContentTemplate>
</asp:UpdatePanel>
```

上述代码使用了 **UpdatePanel** 控件将服务器控件进行绑定，当浏览者操作 **UpdatePanel** 控件中的控件实现某种特定的功能时，页面只会针对 **UpdatePanel** 控件之间的控件进行刷新操作，而不会进行这个页面的刷新。为控件进行事件操作编写代码，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    Label2.Text = DateTime.Now.ToString();           //获取当前时间
}
protected void Button1_Click1(object sender, EventArgs e)
{
    TextBox1.Text = DateTime.Now.ToString();         //获取当前时间
}
protected void TextBox1_TextChanged(object sender, EventArgs e)
```



```
{
    Label1.Text = TextBox1.Text.Length.ToString();           //统计 TextBox1 中的字符串个数
}
```

当用户单击按钮控件时，**TextBox** 控件将会获得当前时间并呈现到 **TextBox** 控件中，当 **TextBox** 控件失去焦点时，则能够统计 **TextBox** 控件中字符串的个数。在传统的 **Web** 开发中，无论是单击按钮还是使用 **AutoPostBack** 属性都需要向服务器发送请求，服务器接收请求后执行请求，请求执行完毕再生成一个 **Web** 页面呈现在客户端。

当 **Web** 页面再次呈现到客户端时，用户能够很明显的感觉到页面被刷新。使用 **UpdatePanel** 控件后，页面只会针对 **UpdatePanel** 控件内的内容进行更新，而不会影响 **UpdatePanel** 控件外的控件，运行后如图 16-8 和图 16-9 所示。

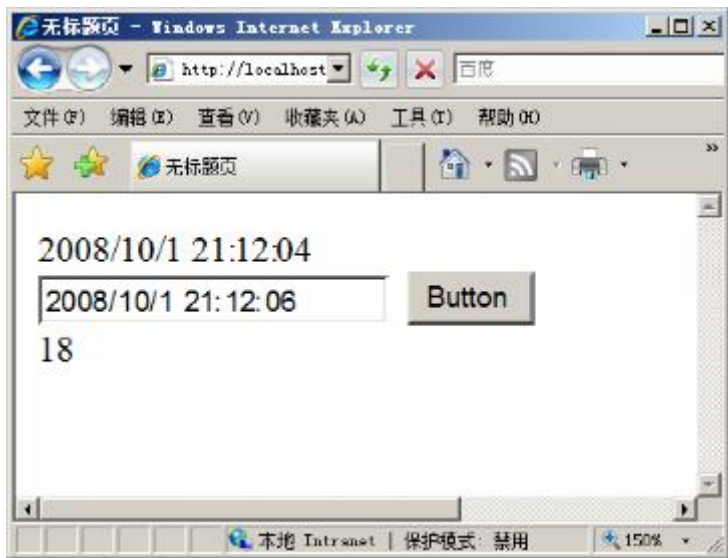


图 16-8 单击按钮获取时间



图 16-9 再次获取时间

当应用程序运行之后，单击按钮控件能够获取当前时间，当再次单击按钮控件之后，当前时间同样能够被获取并呈现在 **TextBox** 中，但是页面并没有再次被更新。在执行过程中，第一次获取的时间为 **2008/10/1 21:12:04**，当再次获取时间时，时间还是 **2008/10/1 21:12:04**，这说明 **UpdatePanel** 控件外的页面元素都没有再更新。

## 16.2 ASP.NET 3.5AJAX 控件

在 **ASP.NET 3.5** 当中，系统提供了 **AJAX** 控件以便开发人员能够在 **ASP.NET 3.5** 中进行 **AJAX** 应用程序开发，通过使用 **AJAX** 控件能够减少大量的代码开发，为开发人员提供了 **AJAX** 应用程序搭建和应用的绝佳环境。

### 16.2.1 脚本管理控件（ScriptManger）

脚本管理控件（**ScriptManger**）是 **ASP.NET AJAX** 中非常重要的控件，通过使用 **ScriptManger** 能够进行整个页面的局部更新的管理。**ScriptManger** 用来处理页面上局部更新，同时生成相关的代理脚本以便能够通过 **JavaScript** 访问 **Web Service**。

**ScriptManger** 只能在页面中被使用一次，这也就是说每个页面只能使用一个 **ScriptManger** 控件，**ScriptManger** 控件用来进行该页面的全局管理。创建一个 **ScriptManger** 控件后系统自动生成 **HTML** 代码，示例代码如下所示。

```
<asp:ScriptManager ID="ScriptManager1" runat="server">
</asp:ScriptManager>
```

**ScriptManger** 控件用户整个页面的局部更新管理，**ScriptManger** 控件的常用属性如下所示：

- ❑ **AllowCustomErrorRedirect**: 指明在异步回发过程中是否进行自定义错误重定向。



- ❑ **AsyncPostBackTimeout:** 指定异步回发的超时事件，默认为 **90** 秒。
- ❑ **EnablePageMethods:** 是否启用页面方法，默认值为 **false**。
- ❑ **EnablePartialRendering:** 在支持的浏览器上为 **UpdatePanel** 控件启用异步回发。默认值为 **True**。
- ❑ **LoadScriptsBeforeUI:** 指定在浏览器中呈现 **UI** 之前是否应加载脚本引用。
- ❑ **ScriptMode:** 指定要在多个类型时可加载的脚本类型，默认为 **Auto**。

在 **AJAX** 应用中，**ScriptManger** 控件基本不需要配置就能够使用。因为 **ScriptManger** 控件通常需要同其他 **AJAX** 控件搭配使用，在 **AJAX** 应用程序中，**ScriptManger** 控件就相当于一个总指挥官，这个总指挥官只是进行指挥，而不进行实际的操作。

## 1. 使用 ScriptManger

**ScriptManger** 控件在页面中相当于指挥的功能，如果需要使用 **AJAX** 的其他控件，就必须使用 **ScriptManger** 控件并且页面中只能包含一个 **ScriptManger** 控件。示例代码如下所示。

```
<body>
  <form id="form1" runat="server">
    <div>
      <asp:ScriptManager ID="ScriptManager1" runat="server">
      </asp:ScriptManager>
      <asp:UpdatePanel ID="UpdatePanel1" runat="server">
        <ContentTemplate>
          <asp:Label ID="Label1" runat="server" Text="这是一串字符" Font-Size="12px"></asp:Label>
          <br /><br />
          <asp:TextBox ID="TextBox1" runat="server" AutoPostBack="True"
            ontextchanged="TextBox1_TextChanged"></asp:TextBox>
            字符的大小(px)
        </ContentTemplate>
      </asp:UpdatePanel>
    </div>
  </form>
</body>
```

上述代码创建了一个 **ScriptManger** 控件和一个 **UpdatePanel** 控件用于 **AJAX** 应用开发。在 **UpdatePanel** 控件中，包含一个 **Label** 标签控件和一个 **TextBox** 文本框控件，当文本框控件的内容被更改时，则会触发 **TextBox1\_TextChanged** 事件。**TextChanged** 事件相应的 **CS** 代码如下所示。

```
protected void TextBox1_TextChanged(object sender, EventArgs e)
{
    try
    {
        Label1.Font.Size = FontUnit.Point(Convert.ToInt32(TextBox1.Text)); //改变字体
    }
    catch
    {
        Response.Write("错误"); //抛出异常
    }
}
```

上述代码通过文本框中的输入进行字体控制，当输入一个数字字符串并失去焦点时，则会触发改事件并执行相应的代码，运行后如图 16-10 和图 16-11 所示。

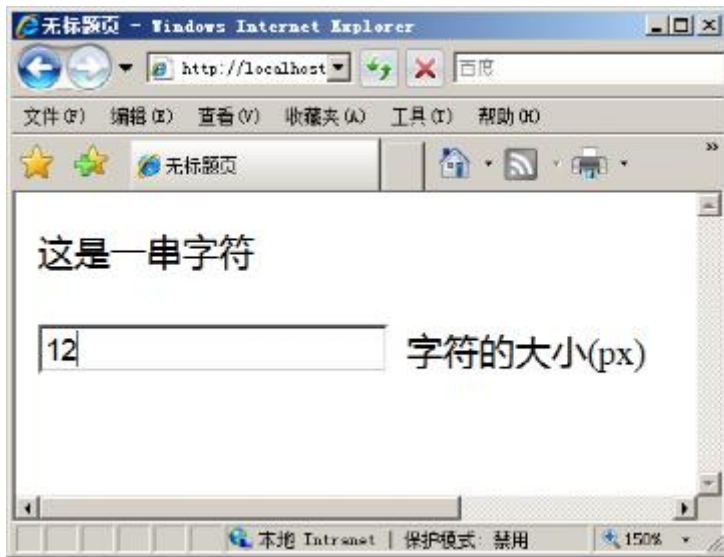


图 16-10 输入字符大小

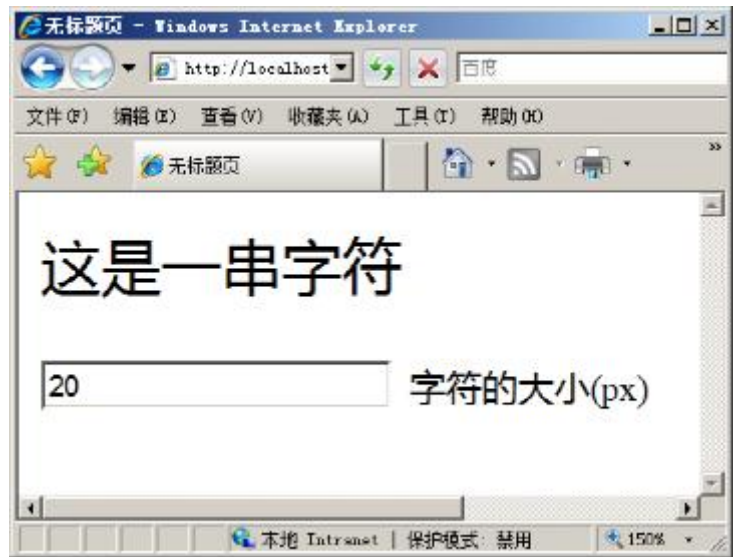


图 16-11 调整字体大小

## 2. 捕获异常

当页面回传发生异常时，则会触发 **AsyncPostBackError** 事件，示例代码如下所示。

```
protected void ScriptManager1_AsyncPostBackError(object sender, AsyncPostBackErrorEventArgs e)
{
    ScriptManager1.AsyncPostBackErrorMessage = "回传发生异常:" + e.Exception.Message;
}
```

**AsyncPostBackError** 事件的触发依赖于 **AllowCustomErrorsRedirect** 属性、**AsyncPostBackErrorMessage** 属性和 **Web.config** 中的 **<customErrors>** 配置节。其中，**AllowCustomErrorsRedirect** 属性指明在异步回发过程中是否进行自定义错误重定向，而 **AsyncPostBackErrorMessage** 属性指明当服务器上发生未处理异常时要发送到客户端的错误消息。示例代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    throw new ArgumentException(); //抛出异常
}
```

上述代码当单击按钮控件时，则会抛出一个异常，**ScriptManger** 控件能够捕获异常并输出异常，运行代码后系统会提示异常“回传发生异常:值不在预期范围内”。

### 16.2.2 脚本管理控件（ScriptMangerProxy）

**ScriptManger** 控件作为整个页面的管理者，**ScriptManger** 控件能够提供强大的功能以致开发人员无需关心 **ScriptManger** 控件是如何实现 **AJAX** 功能的，但是一个页面只能使用一个 **ScriptManger** 控件，如果在一个页面中使用多个 **ScriptManger** 控件则会出现异常。

在 **Web** 应用的开发过程中，常常需要使用到母版页。在前面的章节中提到，母版页和内容窗体一同组合成为一个新页面呈现在客户端浏览器，那么如果在母版页中使用了 **ScriptManger** 控件，而在内容窗体中也使用 **ScriptManger** 控件的话，整合在一起的页面就会出现错误。为了解决这个问题，就可以使用另一个脚本管理控件，**ScriptMangerProxy** 控件。**ScriptMangerProxy** 控件和 **ScriptManger** 控件十分相似，首先创建母版页，示例代码如下所示。

```
<body>
    <form id="form1" runat="server">
        <div style="width:300px; float:left; background:#f0f0f0; height:300px">
            <asp:ContentPlaceHolder ID="ContentPlaceHolder1" runat="server">
            </asp:ContentPlaceHolder>
            <asp:UpdatePanel ID="UpdatePanel2" runat="server">
                <ContentTemplate>
                    <asp:ScriptManager ID="ScriptManager1" runat="server">
                    </asp:ScriptManager>
                    <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
                </ContentTemplate>
            </asp:UpdatePanel>
        </div>
    </form>
</body>
```

```

        <asp:Button ID="Button1" runat="server" onclick="Button1_Click" Text="获取当前时间" />
    </ContentTemplate>
</asp:UpdatePanel>
</div>
<div style="width:300px; float:left; background:gray;color:White;height:300px">
    <asp:ContentPlaceHolder ID="ContentPlaceHolder2" runat="server">
        </asp:ContentPlaceHolder>
    </div>
</form>
</body>

```

上述代码创建了母版页，并且母版页中使用了 **ScriptMangerProxy** 控件为母版页中的控件进行 **AJAX** 应用支持，母版页中按钮控件的事件代码如下所示。

```

protected void Button1_Click(object sender, EventArgs e)
{
    TextBox1.Text = "母版页中的时间为" + DateTime.Now.ToString();           //获取母版页时间
}

```

在内容窗体中可以使用母版页进行样式控制和布局，内容窗体页面代码如下所示。

```

<%@ Page Language="C#"
MasterPageFile="~/Site1.Master"
AutoEventWireup="true"
CodeBehind="MyScriptMangerProxy.aspx.cs" Inherits="_16_2.MyScriptMangerProxy" Title="无标题页" %>
    <asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
    </asp:Content>
    <asp:Content ID="Content2" ContentPlaceHolderID="ContentPlaceHolder1" runat="server">
    </asp:Content>
    <asp:Content ID="Content3" ContentPlaceHolderID="ContentPlaceHolder2" runat="server">
        <asp:ScriptManagerProxy ID="ScriptManagerProxy1" runat="server">
        </asp:ScriptManagerProxy>
    <asp:UpdatePanel ID="UpdatePanel1" runat="server">
    <ContentTemplate>
        <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
        <asp:Button ID="Button1" runat="server" onclick="Button1_Click" Text="内容窗体时间" />
    </ContentTemplate>
    </asp:UpdatePanel>
    <br />
</asp:Content>

```

上述代码为内容窗体代码，在内容窗体中，使用了 **Site1.Master** 母版页作为样式控制，并且通过使用 **ScriptMangerProxy** 控件进行内容窗体 **AJAX** 应用的支持。运行后如图 16-12 所示。

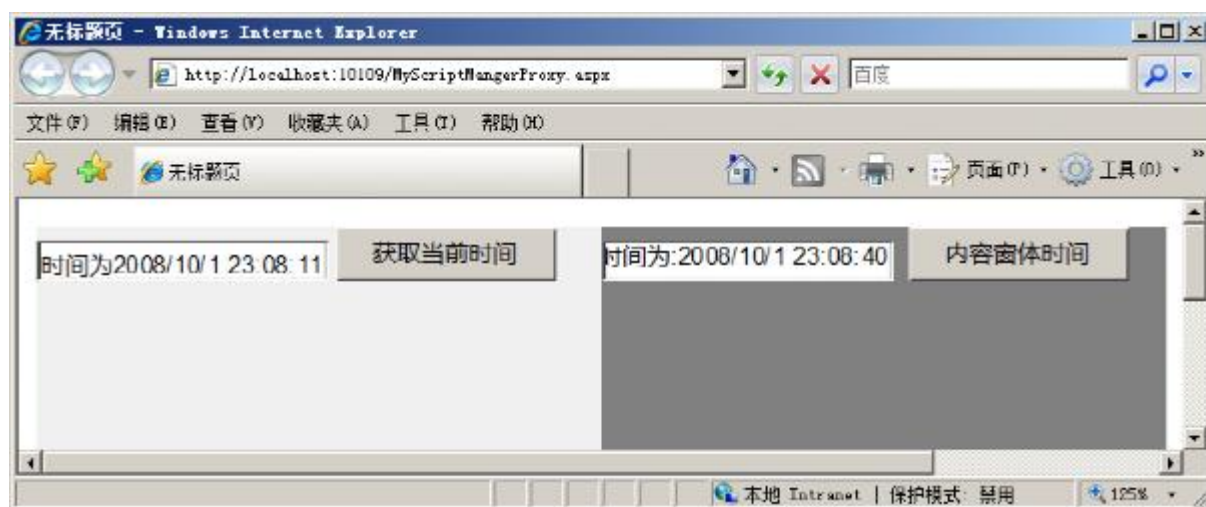


图 16-12 **ScriptMangerProxy** 控件

**ScriptMangerProxy** 控件与 **ScriptManger** 控件非常的相似，但是 **ScriptManger** 控件只允许在一个页面中使用一次。当 **Web** 应用需要使用母版页进行样式控制时，母版页和内容页都需要进行局部更新时 **ScriptManger** 控件就不能完成需求，使用 **ScriptMangerProxy** 控件就能够在母版页和内容页中都实现 **AJAX**

应用。

### 16.2.3 时间控件 (Timer)

在 C/S 应用程序开发中, **Timer** 控件是最常用的控件, 使用 **Timer** 控件能够进行时间控制。**Timer** 控件被广泛的应用在 **Windows WinForm** 应用程序开发中, **Timer** 控件能够在一定的时间内间隔的触发某个事件, 例如每隔 5 秒就执行某个事件。

但是在 **Web** 应用中, 由于 **Web** 应用是无状态的, 开发人员很难通过编程方法实现 **Timer** 控件, 虽然 **Timer** 控件还是可以通过 **JavaScript** 实现, 但是这样也是以复杂的编程的大量的性能要求为代价的, 这样就造成了 **Timer** 控件的使用困难。在 **ASP.NET AJAX** 中, **AJAX** 提供了一个 **Timer** 控件, 用于执行局部更新, 使用 **Timer** 控件能够控制应用程序在一段时间内进行事件刷新。**Timer** 控件初始代码如下所示。

```
<asp:Timer ID="Timer1" runat="server">
</asp:Timer>
```

开发人员能够配置 **Timer** 控件的属性进行相应事件的触发, **Timer** 的属性如下所示。

❑ **Enabled:** 是否启用 **Tick** 时间引发。

❑ **Interval:** 设置 **Tick** 事件之间的连续时间, 单位为毫秒。

通过配置 **Timer** 控件的 **Interval** 属性, 能够指定 **Time** 控件在一定时间内进行事件刷新操作, 示例代码如下所示。

```
<body>
  <form id="form1" runat="server">
    <div>
      <asp:ScriptManager ID="ScriptManager1" runat="server">
      </asp:ScriptManager>
      <asp:UpdatePanel ID="UpdatePanel1" runat="server">
        <ContentTemplate>
          <asp:Label ID="Label1" runat="server" Text="Label"></asp:Label>
          <asp:Timer ID="Timer1" runat="server" Interval="1000" ontick="Timer1_Tick">
          </asp:Timer>
        </ContentTemplate>
      </asp:UpdatePanel>
    </div>
  </form>
</body>
```

上述代码使用了一个 **ScriptManage** 控件进行页面全局管理, **ScriptManage** 控件是必须的。两外, 在页面中使用了 **UpdatePanel** 控件, 该控件用于控制页面的局部更新, 而不会引发整个页面刷新。在 **UpdatePanel** 控件中, 包括一个 **Label** 控件和一个 **Timer** 控件, **Label** 控件用于显示时间, **Timer** 控件用于每 1000 毫秒执行一次 **Timer1\_Tick** 事件, **Label1** 和 **Timer** 控件的事件代码如下所示。

```
protected void Page_Load(object sender, EventArgs e) //页面打开时执行
{
    Label1.Text = DateTime.Now.ToString(); //获取当前时间
}
protected void Timer1_Tick(object sender, EventArgs e) //Timer 控件计数
{
    Label1.Text = DateTime.Now.ToString(); //遍历获取时间
}
```

上述代码在页面被呈现时, 将当前时间传递并呈现到 **Label** 控件中, **Timer** 控件用于每隔一秒进行一次刷新并将当前时间传递并呈现在 **Label** 控件中, 这样就形成了一个可以计数的时间, 如图 16-13 和图 16-14 所示。





图 16-13 初始页面



图 16-14 刷新操作

**Timer** 控件能够通过简单的方法让开发人员无需通过复杂的 **JavaScript** 实现 **Timer** 控制。但是从另一方面来讲，**Timer** 控件会占用大量的服务器资源，如果不停的进行客户端服务器的信息通信操作，很容易造成服务器当机。

## 16.2.4 更新区域控件（UpdatePanel）

更新区域控件（**UpdatePanel**）在 **ASP.NET AJAX** 是最常用的控件，在上面几节控件的讲解中，已经使用到 **UpdatePanel** 控件，这已经说明 **UpdatePanel** 控件是非常重要的 **AJAX** 控件。

**UpdatePanel** 控件使用的方法同 **Panel** 控件类似，只需要在 **UpdatePanel** 控件中放入需要刷新的控件就能够实现局部刷新。使用 **UpdatePanel** 控件，整个页面中只有 **UpdatePanel** 控件中的服务器控件或事件会进行刷新操作，而页面的其他地方都不会被刷新。**UpdatePanel** 控件 **HTML** 代码如下所示。

```
<asp:UpdatePanel ID="UpdatePanel1" runat="server">
</asp:UpdatePanel>
```

**UpdatePanel** 控件可以用来创建局部更新，开发人员无需编写任何客户端脚本，直接使用 **UpdatePanel** 控件就能够进行局部更新，**UpdatePanel** 控件的属性如下所示。

- ❑ **RenderMode**: 该属性指明 **UpdatePanel** 控件内呈现的标记应为 **<div>** 或 **<span>**。
- ❑ **ChildrenAsTriggers**: 该属性指明来在 **UpdatePanel** 控件的子控件的回发是否导致 **UpdatePanel** 控件的更新，其默认值为 **True**。
- ❑ **EnableViewState**: 指明控件是否自动保存其往返过程。
- ❑ **Triggers**: 指明可以导致 **UpdatePanel** 控件更新的触发器的集合。
- ❑ **UpdateMode**: 指明 **UpdatePanel** 控件回发的属性，是在每次进行事件时进行更新还是使用 **UpdatePanel** 控件的 **Update** 方法再进行更新。
- ❑ **Visible**: **UpdatePanel** 控件的可见性。

**UpdatePanel** 控件要进行动态更新，必须依赖于 **ScriptManage** 控件。当 **ScriptManage** 控件允许局部更新时，它会以异步的方式发送到服务器，服务器接受请求后，执行操作并通过 **DOM** 对象来替换局部代码，其原理如图 16-15 所示。



图 16-15 UpdatePanel 控件异步请求示意图

**UpdatePanel** 控件包括 **ContentTemplate** 标签。在 **UpdatePanel** 控件的 **ContentTemplate** 标签中，开发人员能够放置任何 **ASP.NET** 控件这些控件到 **ContentTemplate** 标签中，这些控件就能够实现页面无刷新的更新操作，示例代码如下所示。

```
<asp:UpdatePanel ID="UpdatePanel1" runat="server">
  <ContentTemplate>
    <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
    <asp:Button ID="Button1" runat="server" Text="Button" />
  </ContentTemplate>
</asp:UpdatePanel>
```

上述代码在 **ContentTemplate** 标签加入了 **TextBox1** 控件和 **Button1** 控件，当这两个控件产生回发事件，并不会对页面中的其他元素进行更新，只会对 **UpdatePanel** 控件中的内容进行更新。**UpdatePanel** 控件还包括 **Triggers** 标签，**Triggers** 标签包括两个属性，这两个属性分别为 **AsyncPostBackTrigger** 和 **PostBackTrigger**。**AsyncPostBackTrigger** 用来指定某个服务器端控件，以及将其触发的服务器事件作为 **UpdatePanel** 异步更新的一种触发器，**AsyncPostBackTrigger** 属性需要配置控件的 **ID** 和控件产生的事件名，示例代码如下所示。

```
<asp:UpdatePanel ID="UpdatePanel1" runat="server">
  <ContentTemplate>
    <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
    <asp:Button ID="Button1" runat="server" Text="Button" />
  </ContentTemplate>
  <Triggers>
    <asp:AsyncPostBackTrigger ControlID="TextBox1" EventName="TextChanged" />
  </Triggers>
</asp:UpdatePanel>
```

而 **PostBackTrigger** 用来指定在 **UpdatePanel** 中的某个控件，并指定其控件产生的事件将使用传统的回发方式进行回发。当使用 **PostBackTrigger** 标签进行控件描述时，当该控件产生了一个事件，页面并不会异步更新，而会使用传统的方法进行页面刷新，示例代码如下所示。

```
<asp:PostBackTrigger ControlID="TextBox1" />
```

**UpdatePanel** 控件在 **ASP.NET AJAX** 中是非常重要的，**UpdatePanel** 控件用于进行局部更新，当 **UpdatePanel** 控件中的服务器控件产生事件并需要动态更新时，服务器端返回请求只会更新 **UpdatePanel** 控件中的事件而不会影响到其他的事件。

### 16.2.5 更新进度控件（UpdateProgress）

使用 **ASP.NET AJAX** 常常会给用户造成疑惑。例如当用户进行评论或留言时，页面并没有刷新，而是进行了局部刷新，这个时候用户很可能不清楚到底发生了什么，以至于用户很有可能会产生重复操作，甚至会产生非法操作。

更新进度控件（**UpdateProgress**）就用于解决这个问题，当服务器端与客户端进行异步通信时，需要使用 **UpdateProgress** 控件告诉用户现在正在执行中。例如当用户进行评论时，当用户单击按钮提交表单，系统应该提示“正在提交中，请稍后”，这样就提供了便利从而让用户知道应用程序正在运行中。这种方法不仅能够让用户操作更少的出现错误，也能够提升用户体验的友好度。**UpdateProgress** 控件的 **HTML** 代码如下所示：

```
<asp:UpdateProgress ID="UpdateProgress1" runat="server">
    <ProgressTemplate>
        正在操作中，请稍后 ...<br />
    </ProgressTemplate>
</asp:UpdateProgress>
```

上述代码定义了一个 **UpdateProgress** 控件，并通过使用 **ProgressTemplate** 标记进行等待中的样式控制。**ProgressTemplate** 标记用于标记等待中的样式。当用户单击按钮进行相应的操作后，如果服务器和客户端之间需要时间等待，则 **ProgressTemplate** 标记就会呈现在用户面前，以提示用户应用程序正在运行。完整的 **UpdateProgress** 控件和 **UpdatePanel** 控件代码如下所示。

```
<asp:ScriptManager ID="ScriptManager1" runat="server">
</asp:ScriptManager>
<asp:UpdatePanel ID="UpdatePanel1" runat="server">
    <ContentTemplate>
        <asp:UpdateProgress ID="UpdateProgress1" runat="server">
            <ProgressTemplate>
                正在操作中，请稍后 ...<br />
            </ProgressTemplate>
        </asp:UpdateProgress>
        <asp:Label ID="Label1" runat="server" Text="Label"></asp:Label>
        <asp:Button ID="Button1" runat="server" Text="Button"
            onclick="Button1_Click" />
    </ContentTemplate>
</asp:UpdatePanel>
```

上述代码使用了 **UpdateProgress** 控件用户进度更新提示，同时创建了一个 **Label** 控件和一个 **Button** 控件，当用户单击 **Button** 控件时则会提示用户正在更新，**Button** 更新事件代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    System.Threading.Thread.Sleep(3000);           //挂起 3 秒
    Label1.Text = DateTime.Now.ToString();          //获取时间
}
```

上述代码使用了 **System.Threading.Thread.Sleep** 方法指定系统线程挂起的时间，这里设置 **3000** 毫秒，这也就是说当用户进行操作后，在这 **3** 秒的时间内会呈现“正在操作中，请稍后...”几个字样，当 **3000** 毫秒过后，就会执行下面的方法，运行后如图 16-16 和图 16-17 所示。



图 16-16 正在操作中 图 16-17 操作完毕后

在用户单击后，如果服务器和客户端之间的通信需要较长时间的更新，则等待提示语会出现正在操作中。如果服务器和客户端之间交互的时间很短，基本上看不到 **UpdateProgress** 控件的显示。虽然如此，**UpdateProgress** 控件在大量的数据访问和数据操作中能够提高用户友好度，并避免错误的发生。

## 16.3 AJAX 编程

通过编程的方法实现 **AJAX** 高级功能，能够补充现有的 **AJAX** 功能。例如在执行局部更新时，如果出现了异常，则需要通过编程的方法实现错误信息提交，这样不仅能够提升用户体验的友好度，也能够提升应用程序的健壮性。

### 16.3.1 自定义异常处理

在 **AJAX** 应用程序开发和使用中，用户很容易输入错误信息的信息造成异常。例如在 **UpdatePanel** 控件中执行应用程序操作时，如果发生了错误，则会弹出一个对话框，这个对话框对用户来说非常晦涩并且极不友好，这里就需要自定义异常处理。在页面中，首先需要创建一个 **ScriptManage** 控件和 **UpdatePanel** 控件，示例代码如下所示。

```
<body>
  <form id="form1" runat="server">
    <div>
      <asp:ScriptManager ID="ScriptManager1" runat="server">
      </asp:ScriptManager>
      <asp:UpdatePanel ID="UpdatePanel1" runat="server">
      </asp:UpdatePanel>
    </div>
  </form>
</body>
```

上述代码创建了一个 **ScriptManage** 控件和 **UpdatePanel** 控件，在 **UpdatePanel** 控件中，开发人员可以拖放用户控件，以便进行页面局部更新，示例代码如下所示。

```
<asp:UpdatePanel ID="UpdatePanel1" runat="server">
  <ContentTemplate>
    <asp:Label ID="Label1" runat="server" Text="计算器"></asp:Label>
    <br />
    <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
    除以<asp:TextBox ID="TextBox2" runat="server"></asp:TextBox>
    等于<asp:TextBox ID="TextBox3" runat="server"></asp:TextBox>
    <asp:Button ID="Button1" runat="server" onclick="Button1_Click" Text="计算" />
  </ContentTemplate>
</asp:UpdatePanel>
```

上述代码编写了一个简单的计算器，用户能够通过输入相应的数字进行运算，CS 页面代码如下所示。



```
protected void Button1_Click(object sender, EventArgs e)
{
    try
    {
        int a, b;                                //创建整型变量
        float c;                                //创建浮点型变量
        a = Convert.ToInt32(TextBox1.Text);      //获取数值
        b = Convert.ToInt32(TextBox2.Text);      //获取数值
        c = a / b;                                //进行计算
        TextBox3.Text = c.ToString();            //结果呈现
    }
    catch(Exception ee)
    {
        ee.Data["error"] = "自定义错误";          //编写自定义错误
        throw ee;                                //抛出自定义错误
    }
}
```

上述代码描述了当用户单击按钮后，页面执行转换，即将文本框中的文本进行转换。转换完成后再相除，相除后输出到 **TextBox3** 中。但是这里会有一个问题，这个问题就是如果用户输入的不是数字，或者输入数字的除数是 **0**，都会导致异常。开发人员能够自定义异常并抛出异常，示例代码如下所示。

```
ee.Data["error"] = "自定义错误";                //编写自定义错误
throw ee;                                        //抛出自定义错误
```

当重新抛出异常后，**ScriptManage** 控件能够捕捉该异常。为了让 **ScriptManage** 控件捕捉异常，可以编写 **ScriptManage** 控件的 **AsyncPostBackError** 事件，示例代码如下所示。

```
protected void ScriptManager1_AsyncPostBackError(object sender, AsyncPostBackErrorEventArgs e)
{
    if (e.Exception.Data["error"] != null)        //判断自定义错误
    {
        ScriptManager1.AsyncPostBackErrorMessage =
            "发生了一个错误" + e.Exception.Data["error"].ToString();    //呈现自定义错误
    }
    else
    {
        ScriptManager1.AsyncPostBackErrorMessage = "发生了一个错误";    //默认系统错误
    }
}
```

上述代码通过 **ScriptManage** 控件的 **AsyncPostBackError** 事件捕获一个异常，如果抛出的异常被 **ScriptManage** 控件捕获，则会通过对话框的形式呈现在客户端浏览器，如图 16-18 所示。

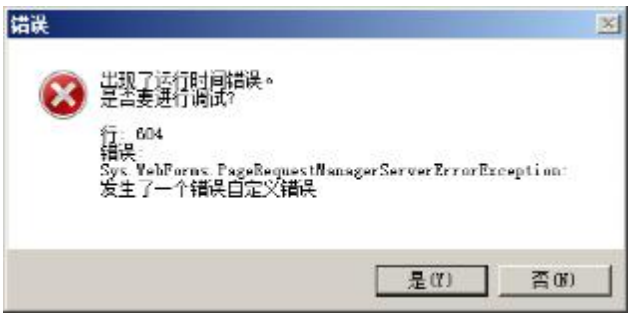


图 16-18 自定义异常处理

16.3.2 使用母版页的 UpdatePanel

在 **AJAX** 应用程序的开发中，常常需要制作大量的相同页面，这些相同的页面可以使用母版页进行样式控制，而内容页只需要进行控件布局即可。如果在母版页中需要完成和实现 **AJAX** 应用，则可以在母版页中使用 **UpdatePanel** 控件进行局部更新。母版页示例代码如下所示。

```
<body>
```

```
<form id="form1" runat="server">
<div style="width:300px; float:left; background:#f0f0f0; height:300px">
  <asp:ContentPlaceHolder ID="ContentPlaceHolder1" runat="server">
  </asp:ContentPlaceHolder>
  <asp:UpdatePanel ID="UpdatePanel2" runat="server">
    <ContentTemplate>
      <asp:ScriptManager ID="ScriptManager1" runat="server">
      </asp:ScriptManager>
      <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
      <asp:Button ID="Button1" runat="server" onclick="Button1_Click" Text="获取当前时间" />
    </ContentTemplate>
  </asp:UpdatePanel>
</div>
<div style="width:300px; float:left; background:gray;color:White;height:300px">
  <asp:ContentPlaceHolder ID="ContentPlaceHolder2" runat="server">
  </asp:ContentPlaceHolder>
</div>
</form>
</body>
```

上述代码在母版页中声明了一个 **ScriptManage** 控件和一个 **UpdatePanel** 控件，在母版页中可以通过向 **UpdatePanel** 控件拖动服务器控件以完成页面局部刷新。在编写内容窗体时，可以无需再创建 **ScriptManage** 控件也同样能够进行页面拒不更新，内容窗体示例代码如下所示。

```
<%@ Page Language="C#"
MasterPageFile="~/Site1.Master"
AutoEventWireup="true" CodeBehind="WebForm1.aspx.cs" Inherits="_16_4.WebForm1" Title="无标题页" %>
  <asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
  </asp:Content>
  <asp:Content ID="Content2" ContentPlaceHolderID="ContentPlaceHolder1" runat="server">
  </asp:Content>
  <asp:Content ID="Content3" ContentPlaceHolderID="ContentPlaceHolder2" runat="server">
  <asp:UpdatePanel ID="UpdatePanel3" runat="server">
    <ContentTemplate>
      <asp:TextBox ID="TextBox2" runat="server"></asp:TextBox>
      <asp:Button ID="Button2" runat="server" onclick="Button2_Click" Text="get time" />
    </ContentTemplate>
  </asp:UpdatePanel>
</asp:Content>
```

在内容窗体中，内容窗体使用了母版页，而母版页中已经包含了 **ScriptManage** 控件，所以当母版页和内容窗体整合在一起呈现一个新页面时，新的页面已经包含了 **ScriptManage** 控件。所以在对内容窗体中 **UpdatePanel** 控件中的控件进行更新时，就算内容窗体中没有 **ScriptManage** 控件，也能够进行局部更新。

注意：如果在内容窗体中使用 **ScriptManage** 控件，则会抛出异常，因为一个页面只允许使用一个 **ScriptManage** 控件，当需要在内容窗体中也使用 **ScriptManage** 控件时，可以使用 **ScriptManageProxy** 控件。

## 16.3.3 母版页刷新内容窗体

在母版页中使用 **ScriptManage** 控件能够方便的将整个页面进行 **AJAX** 全局控制。当 **Web** 应用中有很多相似页面，又需要执行 **AJAX** 应用时，可以在母版页中使用 **ScriptManage** 控件，在内容窗体中使用 **UpdatePanel** 控件，这样母版页中的 **ScriptManage** 控件也会整合到内容窗体中并进行整合输出。

同样，母版页也能够刷新内容窗体中的控件信息，在上一节中，母版页使用了 **ScriptManage** 控件进行全局控制，而内容窗体则通过本身的按钮实现时间的获取。在母版页中，可以创建一个按钮控件以执行内

容窗体中文本框控件的局部更新，母版页局部示例代码如下所示。

```
<asp:UpdatePanel ID="UpdatePanel2" runat="server">
    <ContentTemplate>
        <asp:ScriptManager ID="ScriptManager1" runat="server">
            </asp:ScriptManager>
        <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
        <asp:Button ID="Button1" runat="server" onclick="Button1_Click" Text="获取当前时间" />
        <br />
        <asp:Button ID="Button2" runat="server" Text="刷新子母版的值" />
    </ContentTemplate>
</asp:UpdatePanel>
```

上述代码在母版页中增加了一个按钮，通过该按钮控件能够刷新内容窗体中控件的值。但是如果要实现母版页刷新内容窗体的值，首先需要在母版页代码中注册这两个按钮为异步提交按钮，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    ScriptManager1.RegisterAsyncPostBackControl(Button2);           //注册异步操作
}
```

上述代码通过使用 **RegisterAsyncPostBackControl** 方法进行异步提交按钮控件的注册，注册完毕后就能够为控件编写相应的事件，示例代码如下所示。

```
protected void Button2_Click(object sender, EventArgs e)
{
    ((UpdatePanel)ContentPlaceHolder2.FindControl("UpdatePanel3")).Update(); //查找相应的控件
    TextBox tex = ((TextBox)ContentPlaceHolder2.FindControl("TextBox2"));      //创建 TextBox
    tex.Text = DateTime.Now.ToString();                                       //更改控件值
}
```

上述代码通过 **FindControl** 的方法进行控件的寻找和更改，找到目标控件后再进行目标控件的值的更改。在母版页注册异步提交按钮的方法并实现相应事件后，母版页并不能直接的进行内容窗体的更改，在内容窗体的 **UpdatePanel** 控件中，必须将 **UpdateMode** 属性更改为 **Conditional**，这样才能在内容窗体中接受母版页中进行局部更新的事件。运行后如图 16-19 所示。

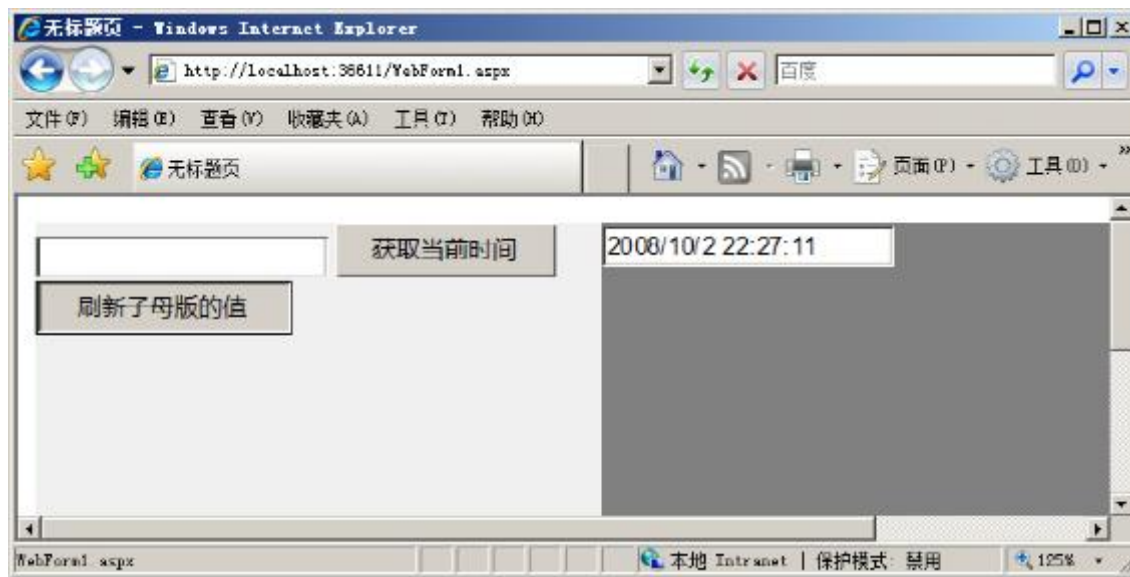


图 16-19 通过母版页进行内容窗体局部刷新

如果 **ASP.NET AJAX Web** 应用中很多的页面都需要执行相同的操作，而内容窗体同样需要执行这些操作，可以通过在母版页中注册异步传送控件以支持内容窗体中 **AJAX** 应用的需求，这样只需要在内容窗体中进行控件布局，而无需在内容窗体中再次创建局部更新控件和进行事件的编写。

### 16.4 小结

本章介绍了 **ASP.NET AJAX** 的一些控件和特性，并介绍了 **AJAX** 基础。在 **Web** 应用程序开发中，使用一定的 **AJAX** 技术能够提高应用程序的健壮性和用户体验的友好度。使用 **AJAX** 技术能够实现页面无刷新和异步数据处理，让页面中其他的元素不会随着“客户端——服务器”的通信再次刷新，这样不仅能够减少客户端服务器之间的带宽，也能够提高 **Web** 应用的速度。

虽然 **AJAX** 是当今热门的技术，但是 **AJAX** 并不是一个新技术，**AJAX** 是由一些老技术组合在一起，这些技术包括 **XML**、**JavaScript**、**DOM** 等，而且 **AJAX** 并不需要在服务器安装插件或安装应用程序框架，只需要浏览器能够支持 **JavaScript** 就能够实现 **AJAX** 技术的部署和实现。尽管 **AJAX** 包括如上诸多的好处，但是 **AJAX** 也有一些缺点，就是对多媒体的支持还没有 **Flash** 那么好，并且也不能很好的支持移动移动设备。本章除了介绍 **AJAX** 基础知识，还介绍了 **ASP.NET AJAX** 开发中必备的控件。本章还包括：

- ❑ **ASP.NET 2.0 AJAX**：讲解了如何在 **ASP.NET 2.0** 中实现 **AJAX** 功能。
- ❑ 脚本管理控件（**ScriptManger**）：讲解了如何使用脚本管理控件。
- ❑ 更新区域控件（**UpdatePanel**）：讲解了如何使用更新区域控件进行页面局部更新。
- ❑ 更新进度控件（**UpdateProgress**）：讲解了如何使用更新进度控件进行更新中进度的统计。
- ❑ 时间控件（**Timer**）：讲解了如何使用时间控件进行时间控制。
- ❑ 自定义异常处理：讲解了如何自定义 **AJAX** 异常。
- ❑ 使用母版页的 **UpdatePanel**：讲解了如何在内容窗体中使用母版页的局部更新控件。
- ❑ 母版页刷新内容窗体：讲解了如何在母版页中进行内容窗体控件的局部更新。

虽然 **AJAX** 包括诸多功能和特性，但是 **AJAX** 也增加了服务器负担，如果在服务器中大量使用 **AJAX** 控件的话，有可能造成服务器假死，熟练和高效的编写 **AJAX** 应用对 **AJAX Web** 应用程序开发是非常有好处的。



## 第 17 章 ASP.NET MVC 基础

在 **ASP.NET** 应用程序开发中，开发人员很难将 **ASP.NET** 应用程序进行良好分层并使相应的页面进行相应的输出，例如页面代码只进行页面布局和样式的输出而代码页面只负责进行逻辑的处理。为了解决这个问题，微软开发了 **MVC** 开发模式方便开发人员进行分层开发。

### 17.1 了解 MVC

**MVC** 是一个设计模式，**MVC** 能够将 **ASP.NET** 应用程序的视图、模型和控制器进行分开，开发人员能够在不同的层次中进行应用程序层次的开发，例如开发人员能够在视图中进行页面视图的开发，而在控制器中进行代码的实现。

#### 17.1.1 MVC 和 Web Form

在 **ASP.NET Web Form** 的开发当中，用户能够方便的使用微软提供的服务器控件进行应用程序的开发，从而提高开发效率。虽然 **ASP.NET Web Form** 提高了开发速度、维护效率和代码的复用性，但是 **ASP.NET** 现有的编程模型抛弃了传统的网页编程模型，在很多应用问题的解决上反而需要通过复杂的实现完成。

在 **ASP.NET MVC** 模型中，**ASP.NET MVC** 模型给开发人员的感觉仿佛又回到了传统的网页编程模型中（如 **ASP** 编程模型），但是 **ASP.NET MVC** 模型与传统的 **ASP** 同样是不同的编程模型，因为 **ASP.NET MVC** 模型同样是基于面向对象的思想进行应用程序的开发。

相比之下，**ASP.NET MVC** 模型是一种思想，而不是一个框架，所以 **ASP.NET MVC** 模型与 **ASP.NET Web Form** 并不具有可比性。同样 **ASP.NET MVC** 模型也不是 **ASP.NET Web Form 4.0**，这两个开发模型就好比一个是汽车一个是飞机，而两者都能够达到同样的目的。

**ASP.NET MVC** 模型是另一种 **Web** 开发的实现思路，其实现的过程并不像传统的 **ASP.NET** 应用程序一样。当用户通过浏览器请求服务器中的某个页面时，其实是实现了 **ASP.NET MVC** 模型中的一个方法，而不是具体的页面，这在另一种程度上实现了 **URL** 伪静态。当用户通过浏览器请求服务器中的某一个路径时，**ASP.NET MVC** 应用程序会拦截相应的地址并进行路由解析，通过应用程序中编程实现展现一个页面给用户，这种页面展现手法同传统的 **ASP.NET Web Form** 应用程序与其他的如 **ASP**，**PHP** 等应用程序都不相同。

同时，随着互联网的发展，搜索引擎在 **Web** 开发中起着重要的作用，这就对页面请求的地址有了更加严格的要求。例如百度、谷歌等搜索引擎会对目录形式的页面路径和静态形式的页面路径收录的更好，而对于动态的如 `abc.aspx?id=1&action=add&t=3` 这种样式的页面路径不甚友好。

另外，所有引擎又在一定程度上决定了 **Web** 应用的热度，例如当在百度中搜索“鞋”这个关键字时，如果搜索的结果中客户的网站在搜索结果的后几页，用户通常不会进行翻页查询，相比之下用户更喜欢在搜索结果中查看前几页的内容。

**ASP.NET MVC** 开发模型在用户进行页面请求时会进行 **URL** 拦截并通过相应的编程实现访问路径和页面的呈现，这样就能够更加方便的实现目录形式的页面路径和静态形式，对于 **Web** 应用动态的地址如 `abc.aspx?id=1&action=add&t=3` 可以以 `abc/action/id/add` 的形式呈现，这样就更加容易的被搜索引擎所搜录。

注意：ASP.NET MVC 模型和 ASP.NET Web Form 并不具备可比性，因为 ASP.NET MVC 模型和 ASP.NET Web Form 是不同的开发模型，而 ASP.NET MVC 模型和 ASP.NET Web Form 在各自的应用上都有有点和缺点，并没有哪个开发模型比另一个模型好之说。

## 17.1.2 ASP.NET MVC 的运行结构

在 ASP.NET MVC 开发模型中，页面的请求并不是像传统的 Web 应用开发中的请求一样是对某个文件进行访问，初学者可能会在一开始觉得非常的不适应。例如当用户访问 `/home/abc.aspx` 时，在服务器的系统目录中一定会存在 `abc.aspx` 这个页面，而对于传统的页面请求的过程也非常容易理解，因为在服务器上只有存在了 `home` 文件夹，在 `home` 文件夹下一定存在 `abc.aspx` 页面才能够进行相应的页面访问。

对于 ASP.NET MVC 开发模型而言，当请求 URL 路径为 `“/home/abc.aspx”` 时，也许在服务器中并不存在相应的 `abc.aspx` 页面，而可能是服务器中某个方法。在 ASP.NET MVC 应用程序中，页面请求的地址不能够按照传统的概念进行分析，要了解 ASP.NET MVC 应用程序的页面请求地址就需要了解 ASP.NET MVC 开发模型的运行结构。ASP.NET MVC 开发模型的运行结构如图 17-1 所示。

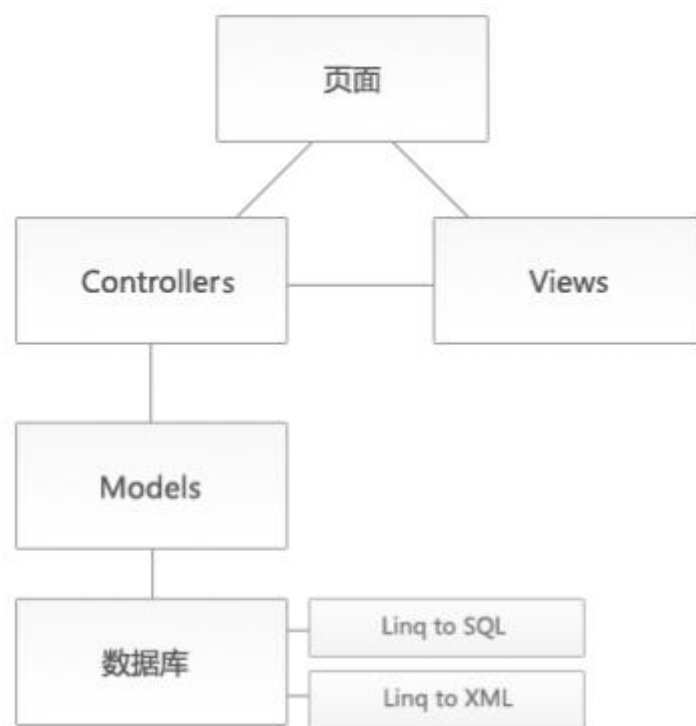


图 17-1 ASP.NET MVC 开发模型

正如图 17-1 所示，ASP.NET MVC 开发模型包括三个模块，这三个模块分别为 MVC 的 M、V、C，其中 M 为 Models（模型）、V 为 Views（视图）、C 为 Controllers（控制器），在 ASP.NET MVC 开发模型中，这三个模块的作用分别如下所示。

- ❑ **Models:** Models 负责与数据库进行交互，在 ASP.NET MVC 框架中，使用 LINQ 进行数据库连接和操作。
- ❑ **Views:** Views 负责页面的页面呈现，包括样式控制，数据的格式化输出等。
- ❑ **Controllers:** Controllers 负责处理页面的请求，用户呈现相应的页面。

与传统的页面请求和页面运行方式不同的是，ASP.NET MVC 开发模型中的页面请求首先会发送到 Controllers 中，Controllers 再通过 Models 进行变量声明和数据读取。Controller 通过页面请求和路由设置呈现相应的 View 给浏览器，用户就能够在浏览器中看到相应的页面。这里讲解 ASP.NET MVC 开发模型的工作流程可能会让读者感到困惑，具体 ASP.NET MVC 开发模型的工作流程会在后面详细讲解。

## 17.2 ASP.NET MVC 基础

ASP.NET MVC 开发模型和 ASP.NET Web Form 开发模型并不相同，ASP.NET MVC 为 ASP.NET Web 开发进行了良好的分层，ASP.NET MVC 开发模型和 ASP.NET Web Form 开发模型在请求处理和应用上都不尽相同，只有了解 ASP.NET Web Form 开发模型的基础才能够高效的开发 MVC 应用程序。

### 17.2.1 安装 ASP.NET MVC

ASP.NET MVC 是微软推出的最新的 ASP.NET Web 开发模型，开发人员可以在微软的官方网站上下载 ASP.NET MVC 安装程序，也能够使用光盘中附属的 ASP.NET MVC 安装程序进行安装，光盘中附带的是 ASP.NET MVC beta 版本，正式版同 beta 版本基本上没有任何区别，开发人员可以在官方网站下载最新的安装程序。单击下载或附录中的 AspNetMVCBeta-setup.msi 进行 ASP.NET MVC 开发模型的安装和相应示例的安装，如图 17-2 所示。

用户单击 ASP.NET MVC 安装界面中的【Next】按钮进入 ASP.NET MVC 安装的用户条款界面，单击【I accept the terms int the License Agreement】复选框同意 ASP.NET MVC 用户条款，如图 17-3 所示。同意后单击【Next】按钮进入 ASP.NET MVC 安装准备界面，进入安装界面后单击【Install】按钮进行安装。



图 17-2 ASP.NET MVC 安装界面

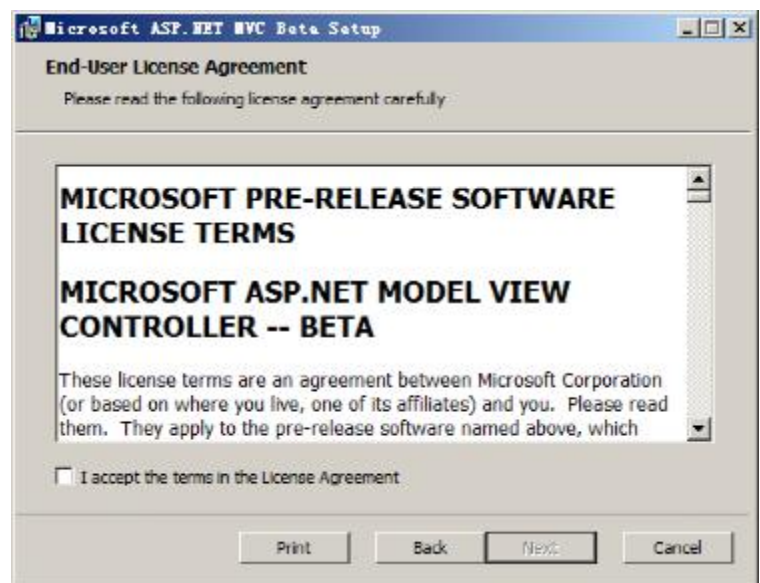


图 17-3 ASP.NET MVC 用户条款

**注意：**在安装 ASP.NET MVC 前必须安装 Visual Studio 2008 进行 ASP.NET MVC 应用程序的开发，安装完成 ASP.NET MVC 应用程序后就能够在 Visual Studio 2008 进行创建 ASP.NET MVC 应用程序。

单击【Install】按钮应用程序，系统就会在计算机中安装 ASP.NET MVC 开发模型和 Visual Studio 2008 中进行 ASP.NET MVC 程序开发所需要的必备组件以便在 Visual Studio 2008 为开发人员提供原生的 ASP.NET MVC 开发环境。安装完毕后，安装程序会提示 ASP.NET MVC 安装程序已经安装完毕，安装完毕后开发人员就能够使用 Visual Studio 2008 开发 ASP.NET MVC 应用程序。安装过程如图 17-4 和 17-5 所示。



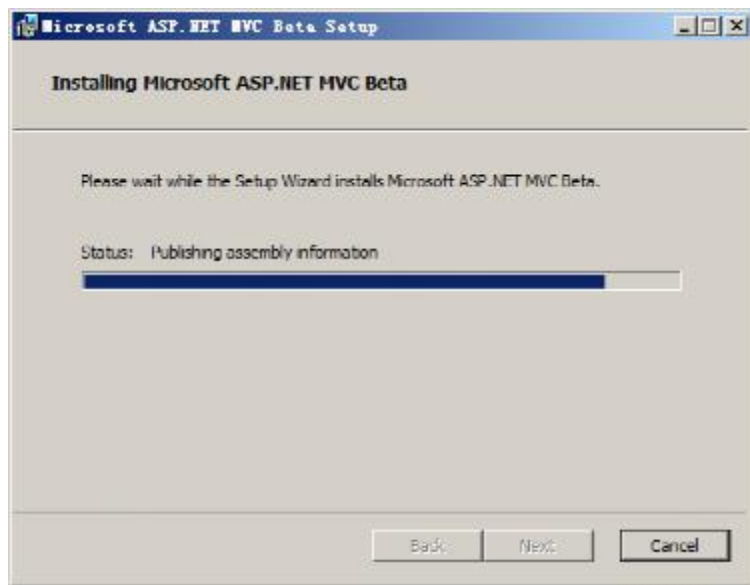


图 17-4 ASP.NET MVC 安装

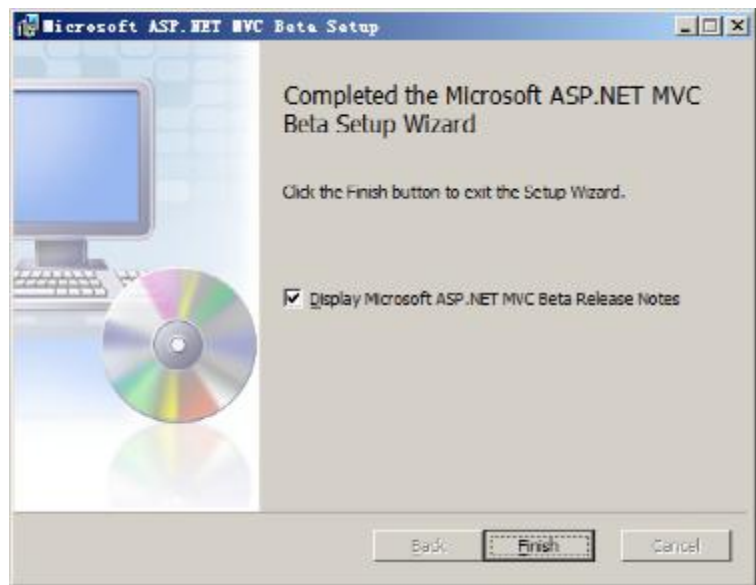


图 17-5 ASP.NET MVC 安装完毕

## 17.2.2 新建一个 MVC 应用程序

安装完成 ASP.NET MVC 开发模型后就能够在 Visual Studio 2008 中创建 ASP.NET MVC 应用程序进行 ASP.NET MVC 应用程序的开发，安装 ASP.NET MVC 开发模型后，Visual Studio 2008 就能够为 ASP.NET MVC 提供原生的开发环境。在菜单栏中选择【文件】选项，单击【文件】选项在下拉菜单中选择【新建项目】就能够创建 ASP.NET MVC 应用程序，如图 17-6 所示。

单击【确定】按钮后就能够创建 ASP.NET MVC 应用程序。Visual Studio 2008 为 ASP.NET MVC 提供了原生的开发环境，以及智能提示，开发人员进行 ASP.NET MVC 应用程序开发中，Visual Studio 2008 同样能够为 ASP.NET MVC 应用程序提供关键字自动补完、智能解析等功能以便开发人员高效的进行 ASP.NET MVC 应用程序的开发。创建 ASP.NET MVC 应用程序后，系统会自动创建若干文件夹和文件，如图 17-7 所示。

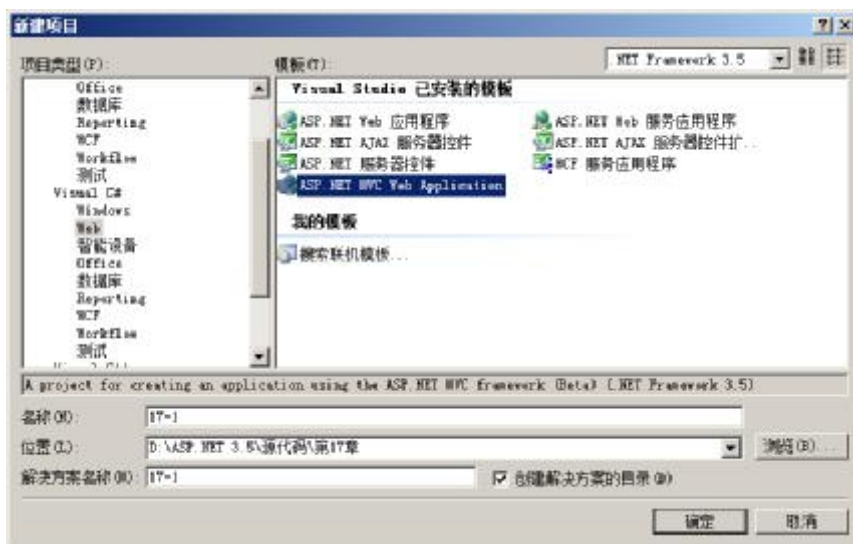


图 17-6 创建 ASP.NET MVC 应用程序



图 17-7 自动创建的文件

在自动创建的文件中，这些文件包括 ASP.NET MVC 应用程序中最重要的文件夹用于分层开发，这些文件夹分别为 Models、Views 和 Controllers，分别对应 ASP.NET MVC 开发模型的 Models（模型）、Views（视图）、Controller（控制器），开发人员能够在相应的文件夹中创建文件进行 ASP.NET MVC 应用程序的开发。

## 17.2.3 ASP.NET MVC 应用程序的结构

在创建完成 ASP.NET MVC 应用程序，系统会默认创建一些文件夹，这些文件夹不仅包括对应 ASP.NET



MVC 开发模型的 **Models**、**Views** 和 **Controllers** 文件夹，还包括配置文件 **Web.config**、**Global.aspx** 和 **Default.aspx**。

## 1. Default.aspx: 页面驱动

**Default.aspx** 用于 ASP.NET MVC 应用程序程序的驱动，当用户执行相应的请求时，**Default.aspx** 能够驱动 ASP.NET MVC 应用程序页面的处理和生成，**Default.aspx** 页面代码如下所示。

```
<%@ Page
Language="C#" AutoEventWireup="true" CodeBehind="Default.aspx.cs" Inherits="_17_1._Default" %>
```

**Default.aspx** 页面代码同传统的 ASP.NET Web Form 基本相同，但 **Default.aspx** 只是用于 MVC 应用程序的驱动。**Default.aspx** 使用 **IHttpHandler** 类获取和发送 HTTP 请求，**Default.aspx.cs** 页面代码如下所示。

```
using System.Web;
using System.Web.Mvc; //使用 Mvc 命名空间
using System.Web.UI;
namespace _17_1
{
    public partial class _Default : Page
    {
        public void Page_Load(object sender, System.EventArgs e)
        {
            HttpContext.Current.RewritePath(Request.ApplicationPath); //拦截虚拟目录根路径
            IHttpHandler httpHandler = new MvcHttpHandler();
            httpHandler.ProcessRequest(HttpContext.Current);
        }
    }
}
```

上述代码用于 ASP.NET MVC 应用程序的驱动。在 ASP.NET MVC 应用程序被运行时，会拦截虚拟目录的根路径将请求发送到 **Controllers** 实现。

## 2. Global.asax: 全局配置文件

**Global.asax** 是全局配置文件，在 ASP.NET MVC 应用程序中的应用程序路径是通过 **Global.asax** 文件进行配置和实现的，**Global.asax** 页面代码如下所示。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc; //使用 Mvc 命名空间
using System.Web.Routing; //使用 Mvc 命名空间
namespace _17_1
{
    // Note: For instructions on enabling IIS6 or IIS7 classic mode,
    // visit http://go.microsoft.com/?LinkId=9394801
    public class MvcApplication : System.Web.HttpApplication
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
            routes.MapRoute(
                "Default", //配置路由名称
                "{controller}/{action}/{id}", //配置访问规则
                new { controller = "Home", action = "Index", id = "" } //为访问规则配置默认值
            ); //配置 URL 路由
        }
        protected void Application_Start()
        {
            RegisterRoutes(RouteTable.Routes);
        }
    }
}
```

```
}
}
```

上述代码在应用程序运行后能够实现相应的 **URL** 映射，当用户请求一个页面时，该页面会在运行时启动并指定 **ASP.NET MVC** 应用程序中 **URL** 的映射以便将请求提交到 **Controllers** 进行相应的编程处理和页面呈现。

**Global.asax** 实现了伪静态的 **URL** 配置，例如当用户访问 **/home/guestbook/number** 服务器路径时，**Global.asax** 通过 **URLRouting** 够实现服务器路径 **/home/guestbook/number** 到 **number.aspx** 的映射。有关 **URLRouting** 的知识会在后面的小结中讲解。

注意：在 **ASP.NET MVC** 开发模型中，浏览器地址栏的 **URL** 并不能够被称为是伪静态，为了方便读者的理解可以暂时称为伪静态，但是最主要的是要理解访问的路径并不像传统的 **Web** 开发中那样是访问真实的某个文件。

### 3. Models、Views 和 Controllers 三层结构

**Models**、**Views** 和 **Controllers** 文件夹是 **ASP.NET MVC** 开发模型中最为重要的文件夹，虽然这里以文件夹的形式呈现在解决方案管理器中，其实并不能看作传统的文件夹。**Models**、**Views** 和 **Controllers** 分别用于存放 **ASP.NET MVC** 应用程序中 **Models**、**Views** 和 **Controllers** 的开发文件。在创建 **ASP.NET MVC** 应用程序后，系统会自行创建相应的文件，这里也包括 **ASP.NET MVC** 应用程序样例，如图 17-8 和图 17-9 所示。



图 17-8 Views 视图文件夹

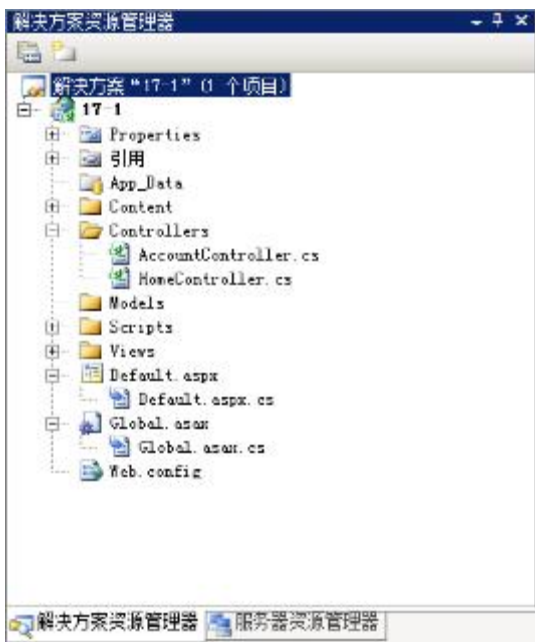


图 17-9 Controllers 控制器文件夹

正如图 17-8、17-9 所示，在样例中分别创建了若干 **Controllers** 控制器文件，以及 **Views** 页面文件。运行 **ASP.NET MVC** 应用程序后，用户的请求会发送到 **Controllers** 控制器中，**Controllers** 控制器接受用户的请求并通过编程实现 **Views** 页面文件的映射。

### 17.2.4 运行 ASP.NET MVC 应用程序

创建 **ASP.NET MVC** 应用程序后就能够直接运行 **ASP.NET MVC** 应用程序，默认的 **ASP.NET MVC** 应用程序已经提供了样例方便开发人员进行编程学习，单击 **【F5】** 运行 **ASP.NET MVC** 应用程序，运行后如图 17-10 所示。

在创建 **ASP.NET MVC** 应用程序后系统会创建样例，图 17-10 显式的就是 **ASP.NET MVC** 默认运行界面，单击旁边的 **【About Us】** 连接页面跳转到相应的页面，如图 17-11 所示。



图 17-10 ASP.NET MVC 应用程序初始界面

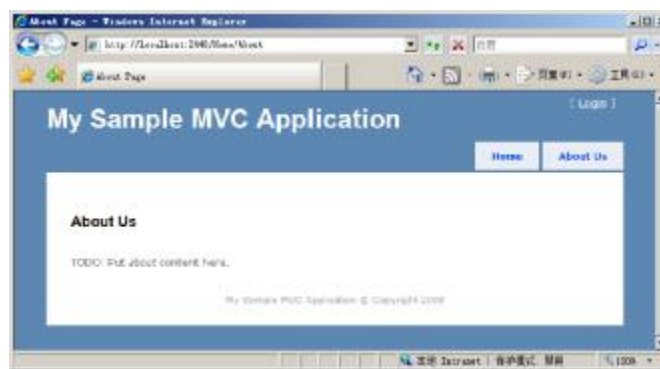


图 17-11 About 页面

当单击【About Us】链接后，页面会跳转到关于页面，页面 URL 为 `http://localhost:2448/Home/About`。在 ASP.NET MVC 应用程序中，URL 路径的请求方式与传统的 ASP.NET Web Form 应用程序不同，开发人员可以发现，在服务器文件中并没有 `/Home/About/index.aspx` 文件也没有 `/Home/About` 这个目录。

注意：在 ASP.NET MVC 应用程序中，这里再三强调，其 URL 并不是服务器中的某个文件而是一种地址映射。

在服务器中没有 `/Home/About/index.aspx` 文件也没有 `/Home/About` 这个目录，因为 `/Home/About` 中所呈现的页面是通过 **Controller** 控制器和 **Global.ascx** 进行相应的文件的路径的映射的，关于地址映射的内容会在后面的小结中详细讲解。

## 17.3 ASP.NET MVC 原理

运行了 ASP.NET MVC 应用程序后，就能够通过相应的地址访问不同的页面。在 ASP.NET MVC 应用程序中，应用程序中页面的 URL 并不是在服务器中实际存在的页面或目录而是访问了相应的方法，ASP.NET MVC 应用程序通过 **Global.ascx** 和 **Controllers** 实现了 URL 映射。

### 17.3.1 ASP.NET MVC 运行流程

在运行 ASP.NET MVC 应用程序后，会发现访问不同的 ASP.NET MVC 应用程序页面时，其 URL 路径并不会呈现相应的 `.aspx` 后缀。同样当访问相应的 ASP.NET MVC 应用程序页面，在服务器中并不存在对应的页面。为了了解如何实现页面映射，就需要了解 ASP.NET MVC 应用程序的运行流程。

在 ASP.NET MVC 程序中，应用程序通过 **Global.ascx** 和 **Controllers** 实现了 URL 映射。当用户进行 ASP.NET MVC 程序的页面请求时，该请求首先会被发送到 **Controllers** 控制器中，开发人员能够在控制器 **Controllers** 中创建相应的变量并将请求发送到 **Views** 视图中，**Views** 视图会使用在 **Controllers** 控制器中通过编程方式创建相应的变量并呈现页面在浏览器中。当用户在浏览器中对 Web 应用进行不同的页面请求时，该运行过程将会循环反复。

对于 **Models** 而言，**Controller** 通常情况下使用 **Models** 读取数据库。在 **Models** 中，**Models** 能够将传统的关系型数据库映射成面向对象的开发模型，开发人员能够使用面向对象的思想进行数据库的数据存取。**Controllers** 从 **Model** 中读取数据并存储在相应的变量中，如图 17-12 所示。

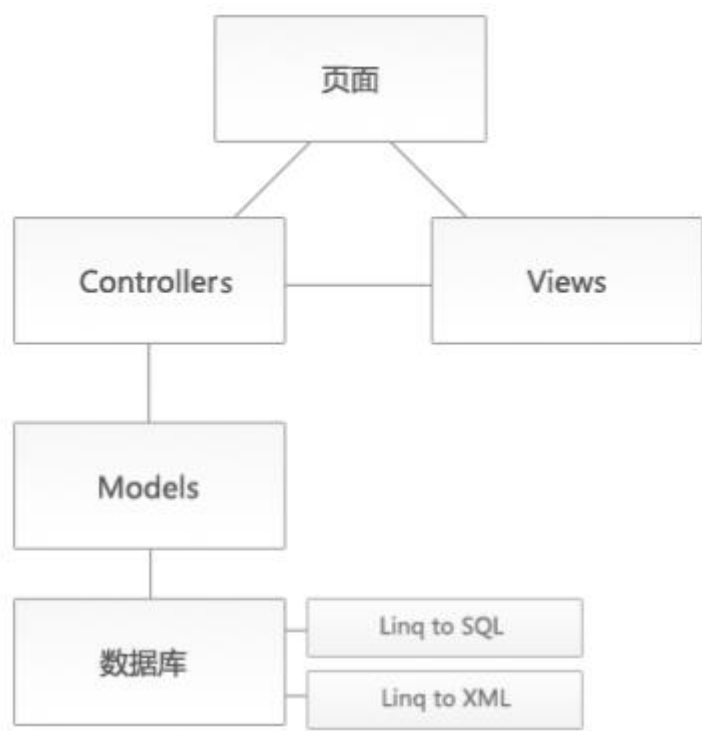


图 17-12 ASP.NET MVC 运行流程

正如图 17-12 所示，在用户进行页面请求时，首先这个请求会发送到 **Controllers** 中，**Controllers** 从 **Models** 中读取相应的数据并填充 **Controllers** 中的变量，**Controllers** 接受相应请求再将请求发送到 **Views** 中，**Views** 通过获取 **Controllers** 中的变量的值进行整合并生成相应的页面到用户浏览器中。

在 **Models** 中需要将数据库抽象成面向对象中的一个对象，开发人员能够使用 **LINQ** 进行数据库的抽象，这样就能够方便的将数据库中的数据抽象成相应的对象并通过对象的方法进行数据的存取和更新。

17.3.2 ASP.NET MVC 工作原理

正如上一节中讲解的 **ASP.NET MVC** 工作流程，在 **ASP.NET MVC** 应用程序中，系统默认创建了相应的文件夹进行不同层次的开发，在 **ASP.NET MVC** 应用程序的运行过程中，同样请求会发送到 **Controllers** 中，这样就对应了 **ASP.NET MVC** 应用程序中的 **Controllers** 文件夹，**Controllers** 只负责数据的读取和页面逻辑的处理。在 **Controllers** 读取数据时，需要通过 **Models** 中的 **LINQ to SQL** 从数据中读取相应的信息，读取数据完毕后，**Controllers** 再将数据和 **Controller** 整合并提交到 **Views** 视图中，整合后的页面将通过浏览器呈现在用户面前。

当用户访问 `http://localhost:2448/Home/About` 页面时，首先这个请求会发送到 **Controllers** 中，**Controllers** 通过 **Global.ascx** 文件中的路由设置进行相应的 **URL** 映射，**Global.ascx** 文件相应代码如下所示。

```
public static void RegisterRoutes(RouteCollection routes)           //注册路由
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
    routes.MapRoute(
        "Default",
        "{controller}/{action}/{id}",
        new { controller = "Home", action = "Index", id = "" }      //配置路由
    );
}
```

上述代码中实现了映射操作，具体是如何实现可以先无需关心，首先需要看看 **Controllers** 文件夹内的文件，以及 **Views** 文件夹的文件，如图 17-13 所示。



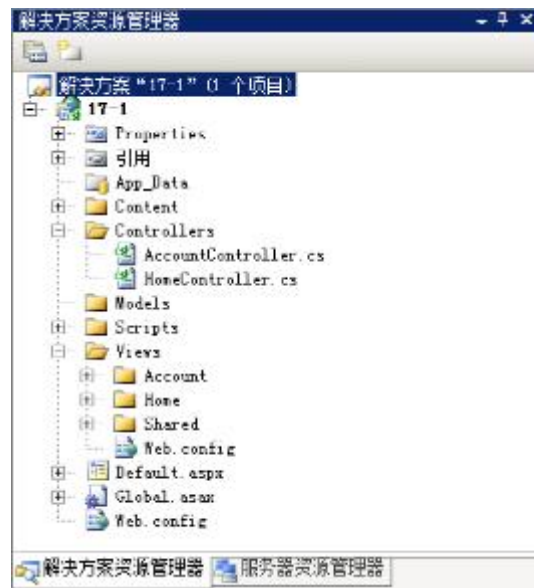


图 17-13 Controller 文件夹和 Views 文件夹

从图 17-13 中可以看出，在 **Views** 中包含 **Home** 文件夹，在 **Home** 文件夹中存在 **About.aspx** 和 **Index.aspx** 文件，而同样在 **Controllers** 文件夹中包含与 **Home** 文件夹同名的 **HomeController.cs** 文件。当用户访问 **http://localhost:2448/Home/About** 路径时，首先该路径请求会传送到 **Controller** 中。

注意：在 **Controllers** 文件夹中创建 **HomeController.cs** 文件同 **Home** 是同名文件，在 **Controllers** 中创建的文件，其文件名后的 **Controller.cs** 是不能更改的，所以 **HomeController.cs** 文件也可以看做是 **Home** 文件夹的同名文件。

在 **Controller** 中，**Controller** 通过 **Global.ascx** 文件和相应的编程实现路径的映射，示例代码如下所示。

```
[HandleError]
public class HomeController : Controller
{
    public ActionResult About()                                //实现 About 页面
    {
        ViewData["Title"] = "About Page";
        return View();                                         //返回视图
    }
}
```

上述代码实现了 **About** 页面的页面呈现，在运行相应的方法后会返回一个 **View**，这里默认返回的是与 **Home** 的 **About** 方法同名的页面，这里是 **about.aspx**，**about.aspx** 页面代码如下所示。

```
<%@ Page
    Language="C#"
    MasterPageFile="~/Views/Shared/Site.Master"
    AutoEventWireup="true" CodeBehind="About.aspx.cs" Inherits="_17_1.Views.Home.About" %>
<asp:Content ID="aboutContent" ContentPlaceHolderID="MainContent" runat="server">
    <h2>About Us</h2>
    <p>
        TODO: Put <em>about</em> content here.
    </p>
</asp:Content>
```

将 **about.aspx** 页面中的文字进行相应的更改，示例代码如下所示。

```
<asp:Content ID="aboutContent" ContentPlaceHolderID="MainContent" runat="server">
    <h2>About Us</h2>
    <p>
        <span style="color:red">这是一个关于页面</span>
    </p>
</asp:Content>
```

运行 `about.aspx` 页面，运行后如图 17-14 所示。

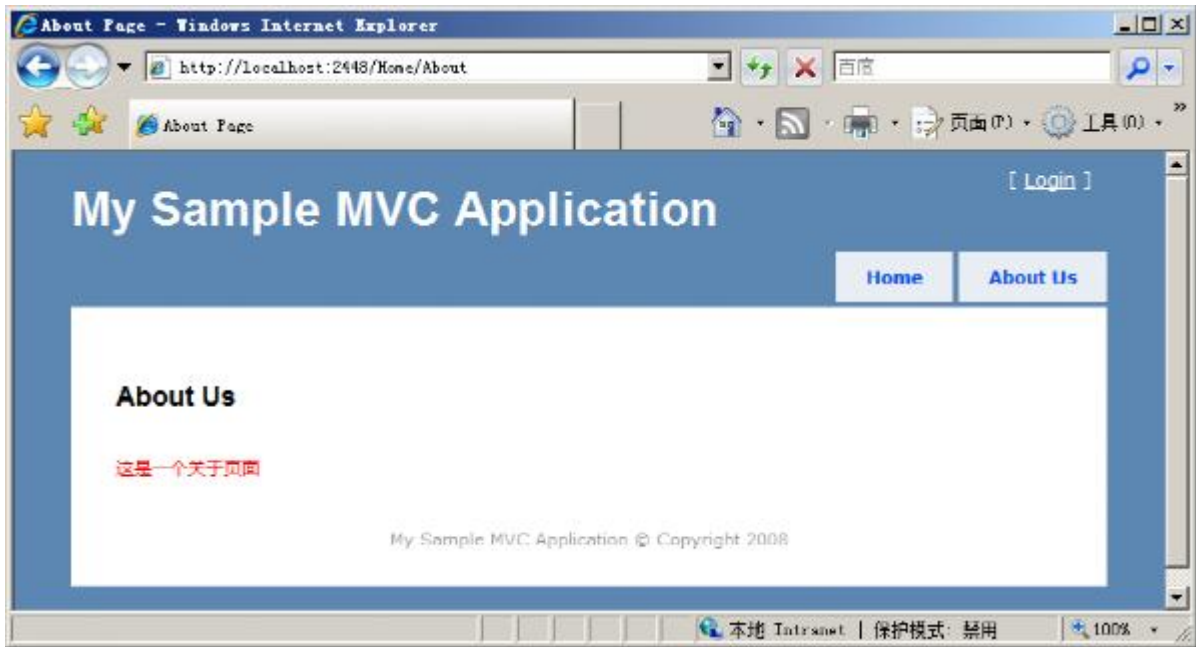


图 17-14 修改后的 About Us 页面

从上述代码可以看出，**Controllers** 与 **Global.ascx** 用于 **URL** 的映射，而 **Views** 用于页面的呈现。从这里可以看出，当用户访问 `http://localhost:2448/Home/About` 页面时，访问的并不是服务器中的 `/Home/About` 页面，而访问的是 **Controllers** 中的 **HomeControllers** 的 **About** 方法。

注意：ASP.NET MVC 应用程序中的 **URL** 路径访问的并不是一个页面，而是一个方法，例如访问 `/Home/About` 页面就是访问的是 **HomeControllers** 中的 **About** 方法，而访问 `/Account/Login` 页面就是访问的是 **AccountControllers** 中的 **Login** 方法。

在 ASP.NET MVC 应用程序中，ASP.NET MVC 应用程序的对应关系如图 17-15 所示。

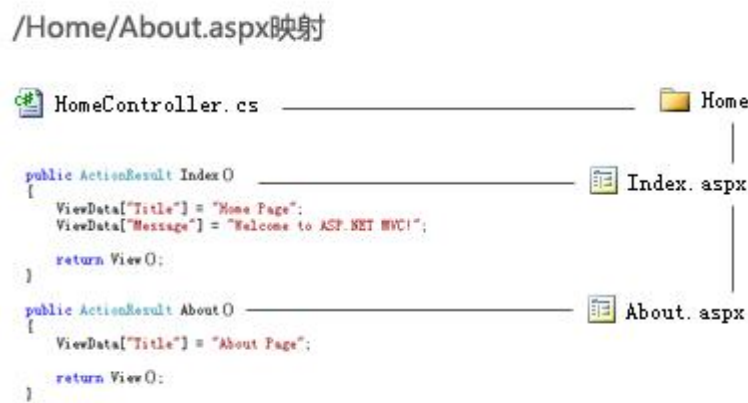


图 17-15 ASP.NET MVC 应用程序关系图

在 ASP.NET MVC 应用程序中，**HomeController.cs** 对应 **Views** 的 **Home** 文件夹，而其中的 **Index** 方法和 **About** 方法对应 `Index.aspx` 文件和 `About.aspx` 文件。

注意：在命名时，默认情况下 `XXXController.cs` 对应 **Views** 的 **XXX** 文件夹，而其中 `XXXController.cs` 中的 `YYY()`方法对应 **XXX** 文件夹中的 `YYY.aspx`，而访问路径为 `XXX/YYY` 是访问的是 `XXXController.cs` 中的 `YYY()`方法。

实现相应的 **URL** 映射需要通过修改 **Global.ascx** 文件进行实现，如何通过修改 **Global.ascx** 文件进行不同的 **URL** 映射将在后面的小结中讲解。

17.4 ASP.NET MVC 开发

在了解了 ASP.NET MVC 工作原理和 workflows，以及 ASP.NET MVC 中的 URL 映射基础原理，就能够进行 ASP.NET MVC 应用程序的开发，在进行 ASP.NET MVC 应用程序开发的过程中可以深入的了解 ASP.NET MVC 应用程序模型和 URL 映射原理。

17.4.1 创建 ASP.NET MVC 页面

ASP.NET MVC 应用程序包括 MVC 三个部分，其中 Models 是用于进行数据库抽象，Views 是用于进行视图的呈现而 Controllers 是用于控制器和逻辑处理，在创建 ASP.NET MVC 应用程序时，可以为 ASP.NET MVC 应用程序分别创建相应的文件。首先在 Views 文件夹中创建一个文件夹，这里创建一个 Beta 文件夹。创建文件夹后单击 Beta 文件夹，右击文件夹，在下拉菜单中选择【添加】选项，在【添加】选项中单击【新建项】选项，单击后系统会弹出对话框用于 View 文件的创建，如图 17-16 所示。



图 17-16 创建 View 文件

在 Views 中可以创建 MVC View Page 用于 Views 文件的创建，从而用于在 ASP.NET MVC 应用程序中呈现相应页的视图，在 Index.aspx 中可以编写相应的代码用于视图的呈现，Index.aspx 页面代码如下所示。

```
<%@ Page
Language="C#"
AutoEventWireup="true" CodeBehind="Beta.aspx.cs" Inherits="_17_1.Views.Beta.Beta" %>
    <h2>About Us</h2>
    <p>
        <span style="color:red">这是一个测试页面</span>
    </p>
```

Index.aspx 页面用于视图的呈现，在一个传统的 ASP.NET 应用程序窗体中，ASP.NET 应用程序窗体是派生自 System.Web.UI.Page 的，而 ASP.NET MVC 应用程序页面代码需要派生自 ViewPage，Index.aspx 的 cs 文件代码在创建时与传统的 ASP.NET 应用程序窗体不同，示例页面代码如下所示。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc; //使用 MVC 命名空间
namespace _17_1.Views.Beta
{
    public partial class Index : ViewPage //派生自 ViewPage
    {
```

```
}  
}
```

在完成 **Beta.aspx** 的创建后，在 **ASP.NET MVC** 应用程序开发模型中还需要创建 **Controllers** 用于接受用户请求和 **Beta.aspx** 页面同名的方法实现。单击 **Controllers** 文件夹，右击 **Controllers** 文件夹，在下拉菜单中选择【添加】选项，在【添加】选项中单击【新建项】选项。这里可以创建一个同名的类文件，如图 17-17 所示。

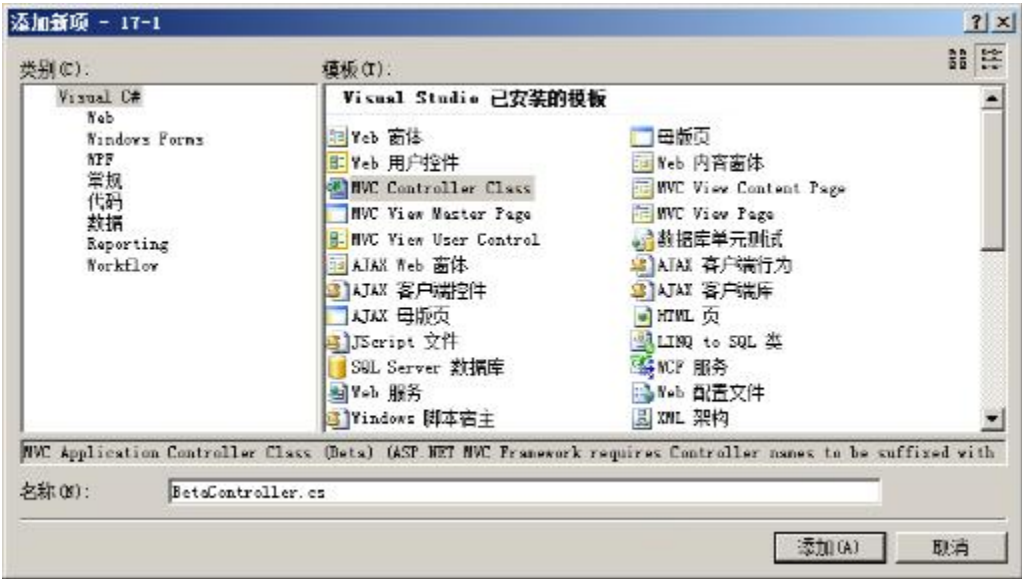


图 17-17 创建 **Controllers** 文件

创建 **Controllers** 类文件时，创面的类文件的名称必须为 **Views** 文件夹中相应的视图文件夹的名称加上 **Controllers.cs**，正如图 17-17 所示，如创建的是“**Beta**”文件夹，在创建 **Controllers** 时必须创建 **BetaControllers.cs**，在创建相应的类文件后才能够拦截相应的 **URL** 并进行地址映射，创建后的 **Controllers** 类文件代码如下所示。

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Web;  
using System.Web.Mvc; //使用 MVC 命名空间  
using System.Web.Mvc.Ajax; //使用 MVC 命名空间  
namespace _17_1.Controllers  
{  
    [HandleError]  
    public class BetaController : Controller  
    {  
        public ActionResult Index() //实现 Index 方法  
        {  
            return View(); //返回 Index 视图  
        }  
    }  
}
```

这里值得注意的是，仅仅创建一个 **Index.aspx** 页面并不能够在浏览器中浏览 **Index.aspx** 页面，必须在相应的 **Controllers** 类文件中实现与 **Index.aspx** 页面文件同名的方法 **Index()**才能够实现 **Index.aspx** 页面的访问。**Views** 中的 **Index.aspx** 页面能够使用 **Controllers** 类文件中的 **Index** 方法中的变量进行数据呈现。单击【F5】运行页面，运行后如图 17-18 所示。



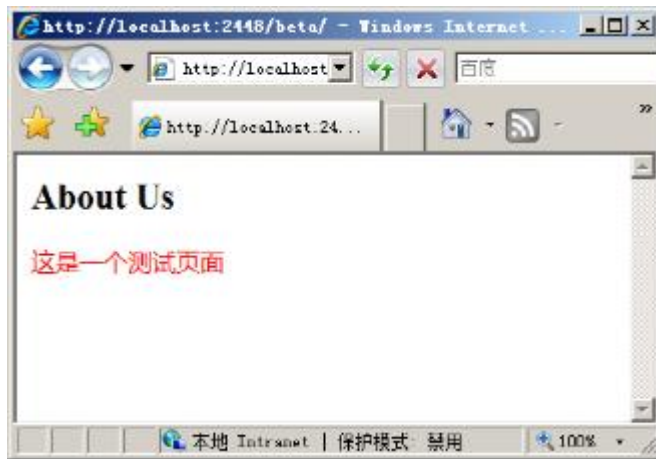


图 17-18 MVC 页面运行

这里讲解了如何手动创建 ASP.NET MVC 页面。在某些版本的 Visual Studio 中，安装了 ASP.NET MVC 开发包应用程序后，可能不会存在 MVC 文件的创建，这时只能通过创建 ASP.NET Web Form 再通过编码实现。

如果希望能够创建 ASP.NET MVC 模板而不使用手动创建可以在 C:\Program Files\Microsoft ASP.NET\ASP.NET MVC Beta\Temp 目录下将压缩包拷贝到相应的 Visual Studio 安装目录 X:\Microsoft Visual Studio 9.0\Common7\IDE\ItemTemplates\CSharp\Web\2052\中，拷贝后在开始菜单中选择“运行”，在窗口中输入 cmd，就会弹出一个黑色的命令行窗口，在命令行输入 cd X:\Microsoft Visual Studio 9.0\Common7\IDE\ItemTemplates\CSharp\Web\2052\进入目录，输入 devenv.exe /setup 进行模板的安装，安装完成后就能够在添加新项中选择 MVC 应用程序模板。

## 17.4.2 ASP.NET MVC 数据呈现（ViewData）

在 ASP.NET MVC 应用程序中，Controllers 负责数据的读取而 Views 负责界面的呈现，在界面的呈现中 Views 通常不进行数据的读取和逻辑运算，数据的读取和逻辑运算都交付给 Controllers 负责。为了能够方便的将 Controllers 与 Views 进行整合并在 Views 中呈现 Controllers 中的变量，可以使用 ViewData 整合 Controllers 与 Views 从而进行数据读取和显示。

在 ASP.NET MVC 应用程序的 Views 中，其值并不是固定的，而是通过 Controllers 传递过来的，在 Controllers 类文件中的页面实现代码中，可以使用 ViewData 进行值的传递，BetaControllers.cs 中 Index.aspx 实现的 Index()的方法示例代码如下所示。

```
[HandleError]
public class BetaController : Controller
{
    public ActionResult Index()                //实现 Index 方法
    {
        ViewData["beta"] = "这是一个 ViewData 字符串";    //使用 ViewData
        return View();                                //返回视图
    }
}
```

上述代码使用 ViewData 存储数据，ViewData 的声明和赋值方式与 Session 对象相同，直接通过编写 ViewData[键值的名称]=XXX 进行相应的键值的赋值。如果需要在页面中进行相应的值的呈现，只需要输出 ViewData[键值的名称]即可。

在 ASP.NET MVC 应用程序中，字符输出都需要呈现在 Views 视图中，在 Controllers 中进行 ViewData 变量的赋值，就需要在 Views 中输出相应的变量，BetaControllers.cs 中的 Index()方法实现的是 Index.aspx 页面，在 Index.aspx 可以使用 ViewData["beta"]变量，示例代码如下所示。

```
<h2>About Us</h2>
<p>
    <span style="color:Red">这是一个测试页面</span><br/>
    <span style="color:Green"><%=ViewData["beta"] %></span>
```

</p>

上述代码中在运行后会输出 ViewData["beta"]变量中存储的值，运行后如图 17-19 所示。

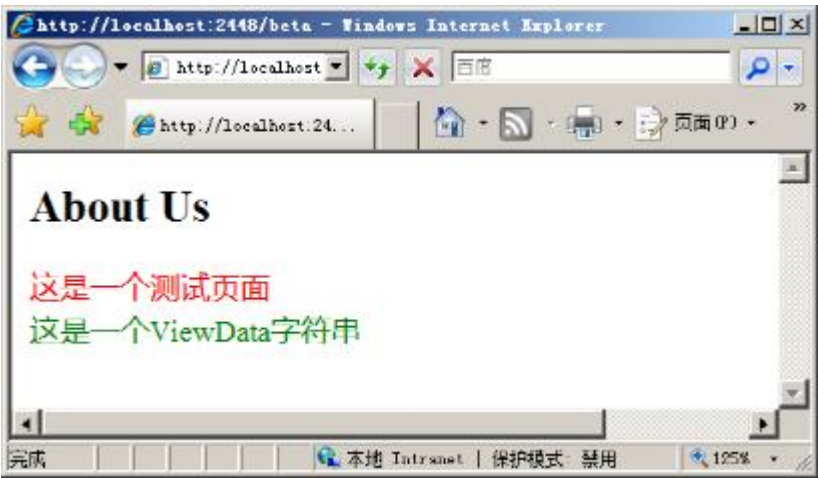


图 17-19 输出 ViewData

**ViewData** 不仅可以为某个具体的值，**ViewData** 还可以是一个泛型变量，示例代码如下所示。

```
[HandleError]
public class BetaController : Controller
{
    public ActionResult Index()
    {
        List<string> str = new List<string>();
        str.Add("str 字符串 1<hr/>");
        str.Add("str 字符串 2<hr/>");
        str.Add("str 字符串 3<hr/>");
        str.Add("str 字符串 4<hr/>");
        ViewData["beta"] = str;
        return View();
    }
}
```

//创建泛型变量  
//添加成员  
//添加成员  
//添加成员  
//添加成员  
//赋值 ViewData  
//返回视图

在为 **ViewData** 赋值泛型变量后，在相应的 **View** 页面中也可以输出 **ViewData** 的值，示例代码如下所示。

```
<h2>About Us</h2>
<p>
    <span style="color:Red">这是一个测试页面</span><br/>
    <% foreach(string str in ViewData["beta"] as List<string>) %>
    <% = str%>
</p>
```

上述代码通过使用 **foreach** 进行 **ViewData** 变量中相应键值的值的遍历，运行后如图 17-20 所示。

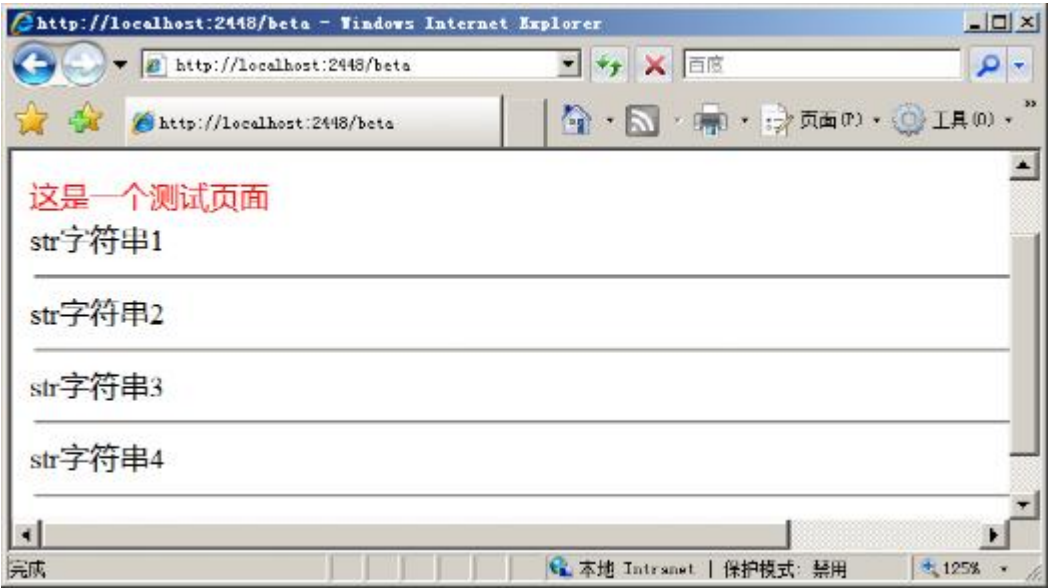


图 17-20 遍历 ViewData 变量的值

使用 **List** 类能够将数据库中的数据存放在泛型列表类中，开发人员能够将数据库中的数据遍历并存放在 **Controllers** 类文件中的页面实现的类的 **ViewData** 变量中，当需要遍历数据进行呈现时，例如新闻列表或者是论坛列表等，可以通过遍历 **ViewData** 变量的值遍历输出数据库中的数据。

## 17.4.3 ASP.NET MVC 跨页数据呈现（TempData）

**ASP.NET MVC TempData** 同 **ASP.NET MVC ViewData** 一样，是在 **Controllers** 中声明的变量以便在 **Views** 中进行调用，示例代码如下所示。

```
[HandleError]
public class BetaController : Controller
{
    public ActionResult Index()
    {
        TempData["beta"] = "TempData 字符串";
        return View();
    }
}
```

上述代码在 **Controllers** 中声明了 **TempData**，在 **Views** 中的相应页面可以使用此 **TempData** 进行变量的输出，示例代码如下所示。

```
<%@ Page
Language="C#"
AutoEventWireup="true" CodeBehind="Beta.aspx.cs" Inherits="_17_1.Views.Beta.Index" %>
    <h2>About Us</h2>
    <p>
        <%=TempData["beta"] %>
    </p>
```

上述代码呈现了 **TempData** 变量的值，运行后如图 17-21 所示。



图 17-21 显示 TempData 变量

在数据呈现上，**TempData** 变量同 **ASP.NET MVC ViewData** 基本相同，但是 **TempData** 能够在跳转中保存值。当用户访问一个页面时，该页面的 **Controllers** 中包含 **TempData** 变量。当这个页面通过 **Redirect** 跳转到另一个页面时，另一个页面能够使用跳转页面的 **TempData** 变量。在跳转页面中，在跳转前可以编写 **TempData** 变量保存相应的值，示例代码如下所示。

```
[HandleError]
public class BetaController : Controller
{
    public ActionResult Index()
    {
        TempData["Index"] = "这是跳转页面的字符串哦..";
        Response.Redirect("/Beta/Get");
        return View();
    }
}
```

//编写 TempData  
 //页面跳转  
 //返回视图

```
}
```

上述代码编写了一个 **TempData** 变量并将页面跳转到 **Get.aspx**，这里在 **Beta** 文件夹下创建一个 **Get.aspx** 页面读取相应的 **TempData** 变量的值。创建完成后，编写 **HTML** 代码如下所示。

```
<%@ Page
Language="C#"
AutoEventWireup="true" CodeBehind="Get.aspx.cs" Inherits="_17_1.Views.Beta.Get" %>
    <h2>接受传递的参数</h2>
    <p>
        <%=TempData["Index"] %>
    </p>
```

编写了页面代码后还不能对页面进行访问，由于 **MVC** 编程模型中路径是访问的 **Controller** 中的方法，所以还需要在 **Controller** 中实现 **Get** 页面的方法，示例代码如下所示。

```
public ActionResult Get()
{
    return View(); //返回默认视图
}
```

上述代码返回默认视图，当不给 **View** 方法进行参数传递，将会返回与方法同名的 **.aspx** 页面文件。这里没有对 **View** 方法传递参数，则返回的是 **Get.aspx** 页面视图，当用户访问 **/Beta/** 路径时，代码会创建一个 **TempData** 变量并跳转到 **/Beta/Get** 路径，在 **/Beta/Get** 路径相应的文件中可以获取跳转前的 **TempData** 变量的值，运行后如图 17-22 所示。



图 17-22 接受 **TempData** 变量的值

在 **Get.aspx** 页面相应的实现代码中并没有声明任何的 **TempData** 变量，而是在跳转前的页面中声明的 **TempData** 变量，与 **ViewData** 相比跳转后的页面能够获取 **TempData** 变量的值而不能获取 **ViewData** 的值，这在某些应用场合如抛出异常等情况下非常适用。

**注意：****TempData** 变量在跳转中能够跨页面进行数据读取，但是跨页面跳转后 **TempData** 变量只能呈现一次。简单的说就是跳转的第一次能过获取跳转前页面的 **TempData** 变量的值，而再次操作时就无法使用跳转前页面的 **TempData** 变量值。

## 17.4.4 ASP.NET MVC 页面重定向

在 **ASP.NET Web Form** 中，可以通过 **Response.Redirect(页面)** 的方法进行页面的重定向，在 **ASP.NET MVC** 编程模型中，也可以通过 **Response.Redirect(页面)** 的方法进行页面重定向。不仅如此，**ASP.NET MVC** 编程模型还支持多种页面重定向方法，传统的页面重定向可以使用 **Response.Redirect(页面)** 方法，示例代码如下所示。

```
public ActionResult Index()
{
```



```
Response.Redirect("/Beta/Get");           // Response.Redirect(页面)
return View();                             //返回视图
}
```

在 MVC 应用程序框架中，开发人员不仅能够使用传统的 **Response.Redirect(页面)** 的方法进行页面重定向，MVC 还支持直接返回重定向参数进行重定向，示例代码如下所示。

```
public ActionResult Index()
{
    return Redirect("/Beta/Get");           //返回重定向参数
}
```

上述代码通过使用重定向参数进行页面重定向。由于 MVC 编程模型是通过 **Controllers** 进行页面的呈现，而 MVC 编程模型同样是基于面向对象的，当用户访问某个路径实际上是访问相应的 **Controllers** 的方法。对于相同的页面而言，开发人员能够使用 MVC 编程模型中提供的 **RedirectToAction** 进行页面重定向，示例代码如下所示。

```
public ActionResult Index()
{
    return RedirectToAction("Get");         //通过方法重定向页面
}
public ActionResult Get()
{
    return View();                         //返回默认视图
}
```

上述代码只能使用在同一个 **Controllers** 中，当需要跳转到不同的 **Controllers** 中时，同样可以通过此方法进行页面重定向，示例代码如下所示。

```
public ActionResult Index()
{
    return RedirectToAction("Login","Account"); //跨 Controllers 跳转
}
```

上述代码同样使用 **RedirectToAction** 方法进行重定向，重载的 **RedirectToAction** 方法前一个参数是相应的页面的方法，而后一个参数是相应的页面所在的 **Controllers**。

17.4.5 ASP.NET MVC URL 路由（URLRouting）

在 ASP.NET MVC 编程模型中，除了 M、V、C 三个模块，MVC 编程模型中最为重要的就是 ASP.NET MVC URLRouting 的概念。运行 ASP.NET MVC 应用程序，其 URL 如图 17-23 所示。

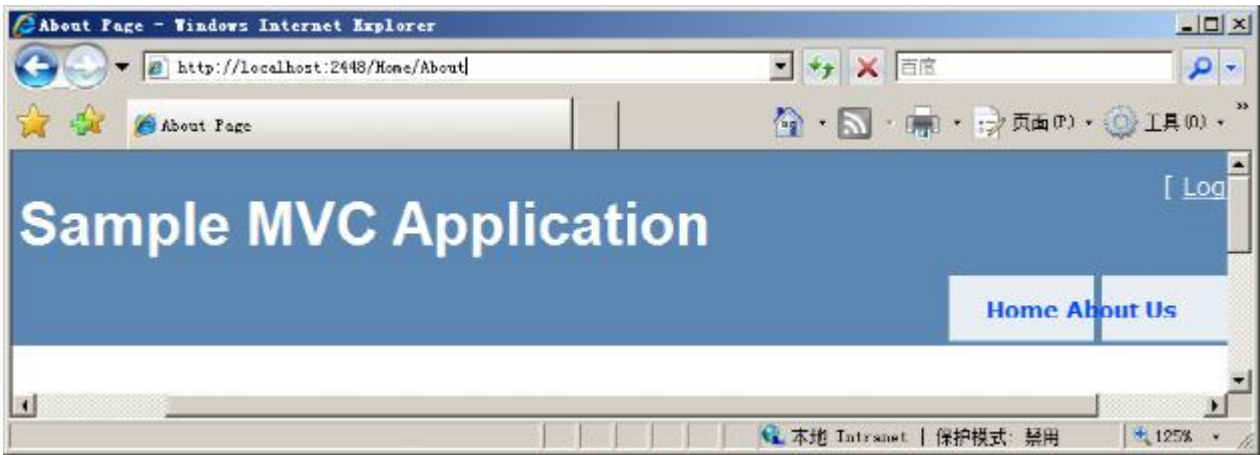


图 17-23 ASP.NET MVC 应用程序 URL 路径

从途中可以看出 URL 路径为 **http://localhost:2448/Home/About**，从前面的小结中可以知道，当访问了该路径时，实际上是访问了 **HomeController.cs** 中的 **About** 方法，而 **About** 方法通过 **About.aspx** 页面进行视图呈现，视图中所使用的数据是在 **About** 方法中声明的 **ViewData**，这样才组成了一个 MVC 应用程序页面。

ASP.NET MVC 应用程序中实现上述过程，即将 **/Home/About** 映射到相应的 **Controllers** 的相应方法中就必须使用到 ASP.NET MVC URLRouting，ASP.NET MVC URLRouting 定义在 **Global.ascx** 文件中，**Global.ascx**

文件代码如下所示。

```
namespace _17_1
{
    public class MvcApplication : System.Web.HttpApplication
    {
        public static void RegisterRoutes(RouteCollection routes)           //注册路由
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");               //注册路径
            routes.MapRoute(
                "Default",                                                  //设置默认名称
                "{controller}/{action}/{id}",                               //设置路由规则
                new { controller = "Home", action = "Index", id = "" }      //实现默认规则
            );
        }
        protected void Application_Start()                                  //应用程序执行
        {
            RegisterRoutes(RouteTable.Routes);                             //实现方法
        }
    }
}
```

上述代码通过 **URLRouting** 实现了 **URL** 地址的映射，在 **Global.ascx** 中，最为重要的是 **RegisterRoutes** 方法，该方法实现了相应映射规则，示例代码如下所示。

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");                     //注册路径
    routes.MapRoute(
        "Default",                                                         //设置默认名称
        "{controller}/{action}/{id}",                                     //设置路由规则
        new { controller = "Home", action = "Index", id = "" }           //实现路由规则
    );
}
```

上述代码中使用了 **RouteCollection** 对象的 **MapRoute** 进行地址配置，其中为 **ASP.NET MVC** 应用程序 **URL** 的配置的规则为 **"{controller}/{action}/{id}"**，这也就是说其映射会按照 **controller** 名称、方法名称和 **ID** 进行映射，所以 **/Home/About** 路径就映射到了 **HomeController.cs** 中的 **About** 方法。在了解了基本的 **URLRouting** 实现 **URL** 地址映射后，开发人员能够修改相应的映射规则进行更改，更改后的规则如下所示。

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
    routes.MapRoute(
        "Default",                                                         //设置默认名称
        "{controller}/{action}.html/{id}",                                //修改规则
        new { controller = "Home", action = "Index", id = "" }           //实现默认规则
    );
}
```

上述代码在相应的规则中进行了修改，修改规则是访问相应的 **Controllers** 中的方法名称加 **.html** 进行页面访问，这样 **http://localhost:2448/Home/About** 就不能够进行访问，因为 **URL Routing** 规则进行了更改。如果要访问相应的页面，则必须访问 **http://localhost:2448/Home/About.html** 进行页面访问，运行后如图 17-24 所示。



图 17-24 修改 URL Routing

正如图 17-24 所示，在访问页面时，原有的页面再不能够进行访问，而必须通过访问 `/Home/About.html` 进行页面访问。

## 17.4.6 ASP.NET MVC 控件辅助工具（Helper）

在 ASP.NET MVC 开发模型中，由于将页面进行分层开发和呈现，开发人员在视图开发中通常是不推荐使用服务器控件的，因为在 ASP.NET MVC 页面是派生自 `ViewPage` 而 ASP.NET WebForm 是派生自 `System.Web.UI.Page` 的，同样为了规范 ASP.NET MVC 开发模型中页面的呈现和运行，使用服务器控件也不是最好的选择。为了能够方便的呈现控件和进行 URL 操作，ASP.NET MVC 开发模型提供了 **Helper** 进行控件的呈现和 URL 操作，Helper 包括 `HtmlHelper` 和 `UrlHelper`。

### 1. HTML 辅助工具（HtmlHelper）

由于在 ASP.NET MVC 开发模型中不推荐使用服务器控件，这就会提高 ASP.NET 页面编程的复杂性，使用 `HtmlHelper` 能够减少相应的编程复杂性。使用 `HtmlHelper` 能够创建 HTML 控件并进行控件编程，在 MVC 编程模型中，其执行过程很像传统的 ASP 的执行过程。使用 `HtmlHelper` 创建 HTML 控件的代码如下所示。

```
<h2>HtmlHelper</h2>
<p>
    请输入用户名:<% =Html.TextBox("Name") %>                                //使用 TextBox
<br/>
    请输入密码:<% =Html.Password("Name") %>                                //使用 Password
<br/>
    <input id="Submit1" type="submit" value="submit" />
</p>
```

上述代码通过 `HtmlHelper` 创建了 HTML 控件，`HtmlHelper` 方法创建控件只能够在 **Views** 中使用而不能在 **Controllers** 中使用。上述代码运行后如图 17-25 所示。

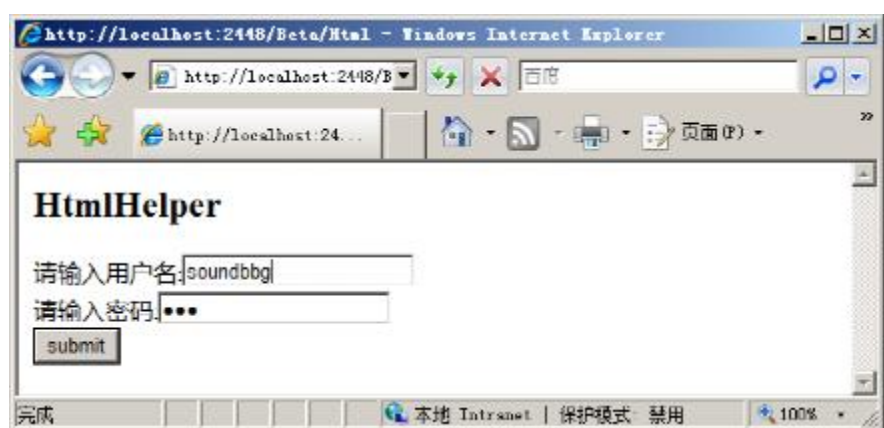


图 17-25 HtmlHelper 创建的 HTML 控件

注意：这里的 `TextBox` 控件和 `Password` 控件并不是 ASP.NET 控件，`TextBox` 控件和 `Password` 控件分别生成的是 HTML 控件。

## 2. URL 辅助工具（UrlHelper）

UrlHelper 在 MVC 开发框架中比较简单，UrlHelper 是用来呈现相应的 URL 路径的，UrlHelper 使用的示例代码如下所示。

```
<h2>HtmlHelper</h2>
<p>
    <%=Url.Action("Index","Beta") %>
</p>
```

上述代码通过使用 UrlHelper 的 Action 方法进行 URL 的呈现，在 Action 方法中，其参数分别为方法名和 Controller，上述代码中用于显示 BetaController 中的 Index 页面 URL，运行后如图 17-26 所示。



图 17-26 UrlHelper

### 17.4.7 ASP.NET MVC 表单传值

ASP.NET 的运行模型很像传统的 ASP，在 ASP.NET MVC 开发模型中，由于无法使用 runat="server" 进行表单传值，开发人员只能自己编写表单进行传值。进行表单传值有两种方法，一种是编写 form 进行表单传值，一种是通过 HtmlHelper 进行表单生成和传值。编写 form 的表单传值方法示例代码如下所示。

```
<h2>HtmlHelper</h2>
<p>
<form id="form1" method="post" action="<%=Html.AttributeEncode(Url.Action("Html","Beta")) %>">
    请输入用户名:<%=Html.TextBox("Name") %>
    <br/>
    请输入密码:<%=Html.Password("Name") %>
    <br/>
    <input id="Submit1" type="submit" value="submit" />
    <%=Url.Action("Index","Beta") %>
    <%= ViewData["p"] %>
</form></p>
```

上述代码通过传统的方法在 HTML 中编写 form 标签，而 form 标签的属性 action 需要通过使用 HtmlHelper 进行指定，这里指定的是 BetaControllers 中的 Html 方法进行参数传递处理。在 Html 方法中，可以通过编程实现相应的表单传值处理，示例代码如下所示。

```
public ActionResult Html()
{
    if (Request.HttpMethod != "POST") //判断传递方法是否为 Post
    {
        ViewData["p"] = "表单还没被传递"; //填充相应的 ViewData
    }
    else
    {
        ViewData["p"] = "表单已被传递"; //填充相应的 ViewData
    }
    return View(); //返回视图
}
```



上述代码首先会判断传递的方法是否为 **POST**，如果为 **POST**，说明表单已经传递，否则表单还没有传递，上述代码运行后如图 17-27 和图 17-28 所示。

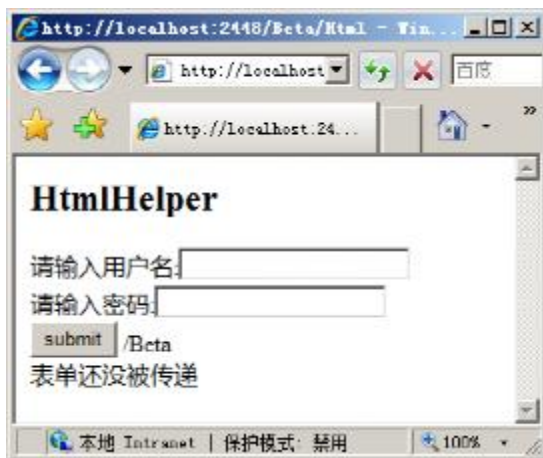


图 17-27 表单没传递

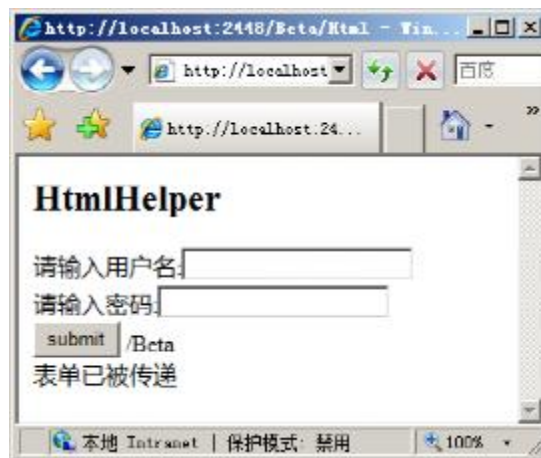


图 17-28 表单已传递

当用户单击按钮控件时，就会进行表单的传递。在 **Html** 方法中，通过编程进行了表单传递后的操作。在 **MVC** 编程中，可以通过 **HttpMethod** 判断表单是否传递，如果传递表单，则可以通过编程的方法进行数据判断和数据操作等其他操作。在 **ASP.NET MVC** 中，还可以使用 **HtmlHelper** 进行表单传值，示例代码如下所示。

```
<h2>HtmlHelper</h2>
<p>
<% using (Html.BeginForm("Html", "Beta"))
{ %>
    请输入用户名:<% =Html.TextBox("Name")%>
    <br/>
    请输入密码:<% =Html.Password("Name")%>
    <br/>
    <input id="Submit1" type="submit" value="submit" />
    <%=Url.Action("Index", "Beta")%>
    <br/>
    <%= ViewData["p"]%>
<%} %>
</p>
```

在表单创建时，**HtmlHelper** 的 **BeginForm** 方法能够创建一个 **Form** 进行表单生成，**BeginForm** 方法包括两个参数，第一个参数为方法，第二个参数为 **Controllers**。当运行该页面时，**BeginForm** 方法能够创建一个 **Form** 进行表单传值。

**注意：**使用 **BeginForm** 方法创建表单，而表单的结束并不是使用 **EndForm**，使用 **BeginForm** 方法创建需要使用 **using{}** 的方法进行 **Form** 中内容容器。

当 **ASP.NET MVC** 应用程序能够进行表单传值后，就能够通过 **HttpMethod** 进行判断，如果用户进行了提交，开发人员能够通过编程实现数据更新的插入和删除等操作。在 **ASP.NET MVC** 应用程序中，其数据通常使用 **LINQ** 进行描述和操作，关于 **LINQ** 的知识会在后面专门讲解。

## 17.5 小结

本章讲解了 **ASP.NET MVC** 开发模型，以及工作原理，在创建 **ASP.NET MVC** 应用程序时，系统会自行创建若干文件和文件夹。**ASP.NET MVC** 开发模型和 **ASP.NET Web Form** 极不相同，所以创建的文件夹和文件也不相同，要了解 **ASP.NET MVC** 开发模型就首先需要了解这些文件和文件夹的作用。本章还讲解了 **ASP.NET MVC** 的工作原理和工作流程，包括 **ASP.NET MVC** 中的 **Controllers**、**Model** 以及 **Views** 是如何

形成一个完整的页面呈现在客户端浏览器中。本章还包括：

- ❑ **安装 ASP.NET MVC:** 讲解了如何在 **Visual Studio 2008** 中安装 **ASP.NET MVC** 应用程序开发包进行 **ASP.NET MVC** 应用程序开发。
- ❑ **新建一个 MVC 应用程序:** 讲解了如何创建一个新的 **ASP.NET MVC** 进行应用程序开发。
- ❑ **ASP.NET MVC 应用程序的结构:** 讲解了 **ASP.NET MVC** 应用程序的基本结构, 以及 **ASP.NET MVC** 中 **M**、**V**、**C** 的概念。
- ❑ **创建 ASP.NET MVC 页面:** 讲解了如何创建 **ASP.NET MVC** 页面。
- ❑ **ASP.NET MVC ViewData:** 讲解了 **ASP.NET MVC** 中 **ViewData** 的作用和使用方法。
- ❑ **ASP.NET MVC TempData:** 讲解了 **ASP.NET MVC** 中 **TempData** 的作用和使用方法。
- ❑ **ASP.NET MVC 页面重定向:** 讲解了 **ASP.NET MVC** 中页面重定向的方法。
- ❑ **ASP.NET MVC URLRouting:** 讲解了什么是 **ASP.NET MVC URLRouting**, 以及 **URLRouting** 基本原理。
- ❑ **ASP.NET MVC Helper:** 讲解了基本的 **ASP.NET MVC** 中 **HtmlHelper**, 以及 **UrlHelper** 的作用。
- ❑ **ASP.NET MVC 表单传值:** 讲解了如何使用 **ASP.NET MVC** 应用程序的 **HtmlHelper**, 以及传统的 **HTML** 方式进行表单传值。

在 **ASP.NET MVC** 应用程序中, 使用数据操作通常是使用 **LINQ** 进行数据操作的, 当用户提交了表单, 开发人员能够通过判断进行数据的相应操作, 开发人员能够使用 **LINQ** 进行数据操作, 有关 **LINQ** 的知识将在后面的章节讲解。

## 第 18 章 WCF 开发基础

**WCF (Windows Communication Foundation)** 是 **.NET Framework** 的扩展，**WCF** 提供了创建安全的、可靠的、事务服务的统一框架，**WCF** 整合和扩展了现有分布式系统的开发技术，如 **Microsoft .NET Remoting**、**Web Services**、**Web Services Enhancements (WSE)** 等等，来开发统一的可靠的应用程序系统。

### 18.1 了解 WCF

**WCF** 是 **.NET Framework** 的扩展，同时 **WCF** 提供了一种在 **Windows** 环境下进行客户端开发和服务端开发的 **SDK**，并且为服务提供了运行环境。**WCF** 提供了创建安全的、可靠的、事务服务的统一框架，整合了现有的分布式技术，开发人员能够使用 **WCF** 快速创建基于服务的应用程序。

#### 18.1.1 什么是 WCF

**WCF** 是基于 **Windows** 平台下开发和部署服务的软件开发包 (**Software Development Kit, SDK**)。**WCF** 提供了服务的运行环境，这样就让开发人员能够将 **CLR** 类型公开为服务，也能够通过使用 **CLR** 类型来使用服务。**WCF** 框架模型如图 18-1 所示。

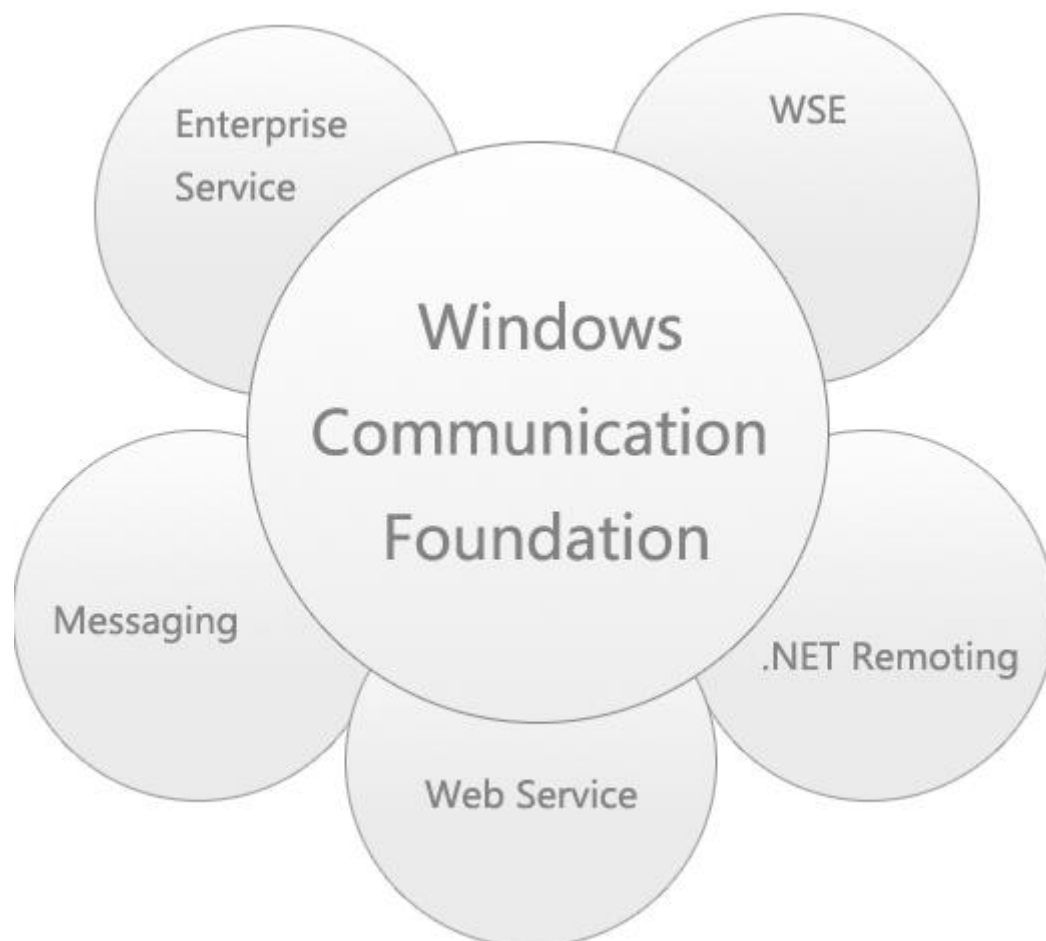


图 18-1 WCF 框架模型

**WCF** 提供了创建安全的、可靠的、事务服务的统一框架，**WCF** 整合和扩展了现有分布式系统的开发技术，如 **Microsoft .NET Remoting**、**Web Services**、**Web Services Enhancements (WSE)** 等等，来开发统一的

可靠系统。**WCF** 简化了 **SOA** 框架的应用，同时也统一了 **Enterprise Services**、**Messaging**、**.NET Remoting**、**Web Services**、**WSE** 等技术，极大的方便了开发人员进行 **WCF** 应用程序的开发和部署，同时也降低了 **WCF** 应用开发的复杂度。

**WCF** 支持大量的 **Web Service** 标准，这些标准包括 **XML**、**XSD**、**SOAP**、**Xpath**、**WSDL** 等标准和规范，所以对于现有的标准，开发人员能够方便的进行移植。同时 **WCF** 可以使用 **Attribute** 属性进行 **WCF** 应用程序配置，提高了 **WCF** 应用的灵活性。**WCF** 遵循客户端/服务器模型在应用程序之间进行通信，客户端程序能够通过服务器端提供的 **EndPoint** 端直接访问服务，如图 18-2 所示。

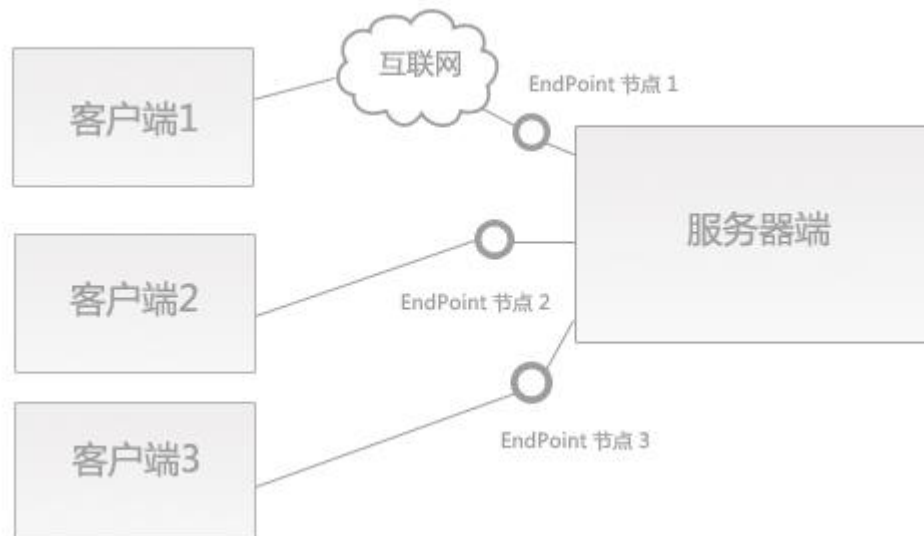


图 18-2 WCF 通信模型

虽然开发人员需要实现服务可以不使用 **WCF**，但是 **WCF** 封装了现有的类和结构，提供了服务实现的便捷手段，通过使用 **WCF** 能够快速的实现服务并让其他的应用程序使用服务。**WCF** 是微软提供的一系列协议的标准，包括服务交互、类型转换等。

**WCF** 中绝大部分的实现和功能都包含在一个单独的程序集 **System.ServiceModel.dll** 中，命名空间为 **System.ServiceModel**。通过使用 **System.ServiceModel** 命名空间能够快速搭建 **WCF** 应用程序环境。**WCF** 是 **.NET 3.0** 的一部分，但是 **.NET 3.0** 是基于 **.NET 2.0** 为基础而存在的，如果需要搭建和使用 **WCF** 应用，则服务器应该具备 **.NET 3.0** 环境。

### 18.1.2 为什么需要 WCF

在传统的应用程序开发中，例如在为麦当劳开发一个餐饮统计的应用程序，这个应用程序能够统计麦当劳的餐饮系统，包括每天客户购买的餐饮、餐饮的价格以及当天的餐饮统计。这个应用程序通常是安装在麦当劳店面主机中的，但是有很多的应用程序将需要对此餐饮统计应用程序进行访问和数据提取，这些应用程序有的是基于 **.NET** 的，有的是基于 **J2EE** 的，另一些可能是基于 **ASP.NET** 的 **Web** 应用，这样就造成了应用程序访问困难。如图 18-3 所示。



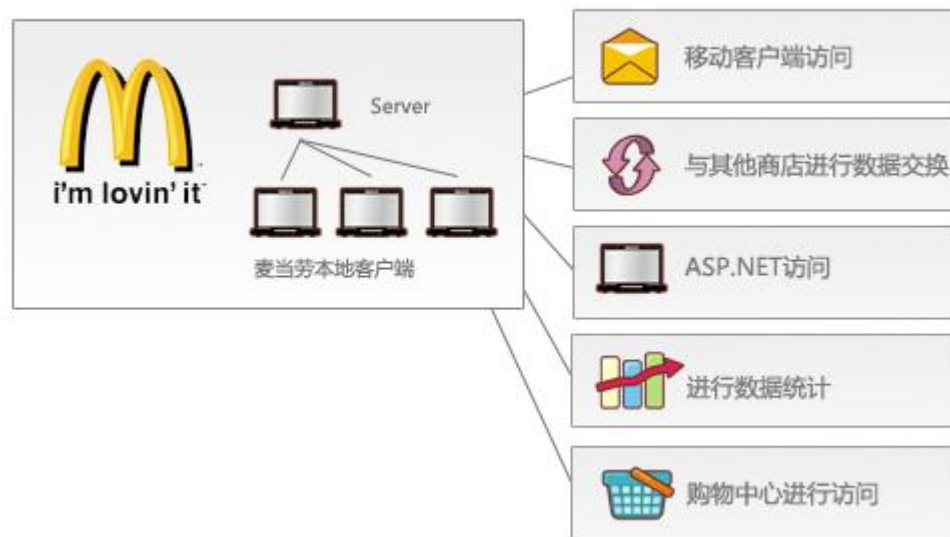


图 18-3 麦当劳业务模拟图

如图 18-3 中所示，麦当劳的餐饮业务也许需要支持很多其他的设备，在现在智能手机发达的今天，很多客户可能可以从移动客户端访问麦当劳的餐饮业务，这些移动客户端可能是 **PDA**、**Windows Mobile**、**GPhone** 或者 **IPhone**。在其他的客户端访问时，例如总部可能需要提取分部的数据，用户可以从网站中购买餐饮，分部经理需要对当天的数据进行统计，或者购物中心应用程序访问餐饮应用程序以增删数据，这些流程都必须考虑到平台、协议和通信等诸多因素。

**WCF** 可以看作是 **ASMX**、**.NET Remoting**、**Enterprise Service**、**WSE**、**MSMQ** 这些技术的并集，虽然在复杂度上 **WCF** 很可能比这些技术更加复杂，因为 **WCF** 是面向服务构架的，所以对于上述的麦当劳餐饮业务的例子，如果使用 **WCF** 就能够很好的实现不同平台，不同设备之间的安全性、可依赖性、互操作性等特性，而因为 **WCF** 对现有技术的封装，开发人员可以无需关心 **ASMX**、**Net Remoting** 这些技术的实现细节。

## 18.2 WCF 基础

在了解了 **WCF** 的概念和通信原理，以及为什么要使用 **WCF** 之后，就能够明白 **WCF** 在现在的应用程序开发中所起到的作用，**WCF** 能够实现不同技术和平台之间的安全性、可依赖性和用户操作性的实现，对大型应用程序开发起到促进作用。

### 18.2.1 服务

服务是一组公开的功能的集合。在软件开发领域，从传统的面向过程，到面向对象，然后历经了面向组件的开发一致发展到当今的面向服务开发。

#### 1. WCF 服务

面向服务开发也并不是什么新技术，面向服务开发只是之前的面向过程、面向对象、组件开发和面向服务开发一种补充。面向服务开发有如下优点：

- ❑ 重用性：面向服务的开发提升了应用程序的重用性，通过创建可用于服务的接口能够实现不同应用程序中使用相同或类似程序实现的代码。
- ❑ 注重效率：面向服务的开发可以使用现有的服务的集合，这样能够让开发人员能够快速地进行数据交换和开发，而无需关注底层服务的实现。
- ❑ 松耦合：面向服务的程序是独立于服务执行环境的程序，这样就让程序成为一个松耦合的应用。
- ❑ 职责划分：通过使用面向服务的开发能够进行职责的划分，例如经理和业务人员只需关心业务和统计数据即可，开发人员能够关注应用程序的开发。

一个面向服务的应用程序会将众多的服务集成到一起，形成单个逻辑单元，如图 18-4 所示。

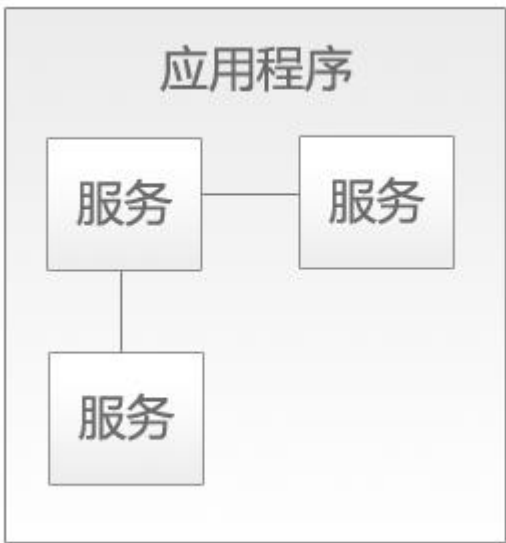


图 18-4 面向服务的应用

**WCF** 中的服务可以是本地的，也可以使用远程的服务。对于客户端而言，客户端只需要通过使用服务来实现应用程序功能，这些客户端也可以是不同的类型，包括 **Windows** 应用程序，**ASP.NET** 应用程序甚至是移动终端。

对于客户端而言，客户端是通过使用消息与服务器进行通信。消息可以直接在客户端与服务之间进行传递，也可以通过中间方进行传递。在服务器和客户端之间的消息是通过 **SOAP** 进行通信的，**SOAP** 与 **Web** 应用开发中不同的是，**Web** 应用通常需要某个具体的协议进行相应功能的实现，例如 **HTTP**、**FTP** 协议等，而在 **WCF** 中，**WCF** 服务可以在不同的协议中进行传递，并不局限于某个协议。正是因为如此，客户端与服务器之间的要求往往不是必须的，这也就是说，**WCF** 客户端可以与一个非 **WCF** 服务器进行信息通信，而一个非 **WCF** 客户端也可以与一个 **WCF** 服务器进行信息通信。

为了保障 **WCF** 服务器的安全性，**WCF** 服务器不允许直接对服务的调用。对于 **WCF** 客户端，只允许使用代理（**Proxy**）将调用信息转发给服务器。代理向客户端公开的操作和服务器端的操作相同。

2. 服务的执行边界

**WCF** 能够让客户端跨越执行边界与 **WCF** 服务进行通信，**WCF** 客户端和 **WCF** 服务器进行通信必须使用带来与服务进行通信，即使是与本地服务进行通信，如图 18-5 所示。

图 18-5 展示了 **WCF** 客户与本机服务进行通信，**WCF** 不仅能够支持不同应用程序域之间的服务的访问，也能够支持不同进程之间的服务的访问。这就让 **WCF** 客户端可以调用一个应用程序中的服务，也可以调用不同应用程序甚至不同进程中的 **WCF** 服务。不仅如此，**WCF** 还支持客户端对远程计算机的中服务的调用，在远程服务调用中，**WCF** 允许客户端可以跨越 **Intranet** 或 **Internet** 的边界进行远程服务的访问和调用，如图 18-6 所示。

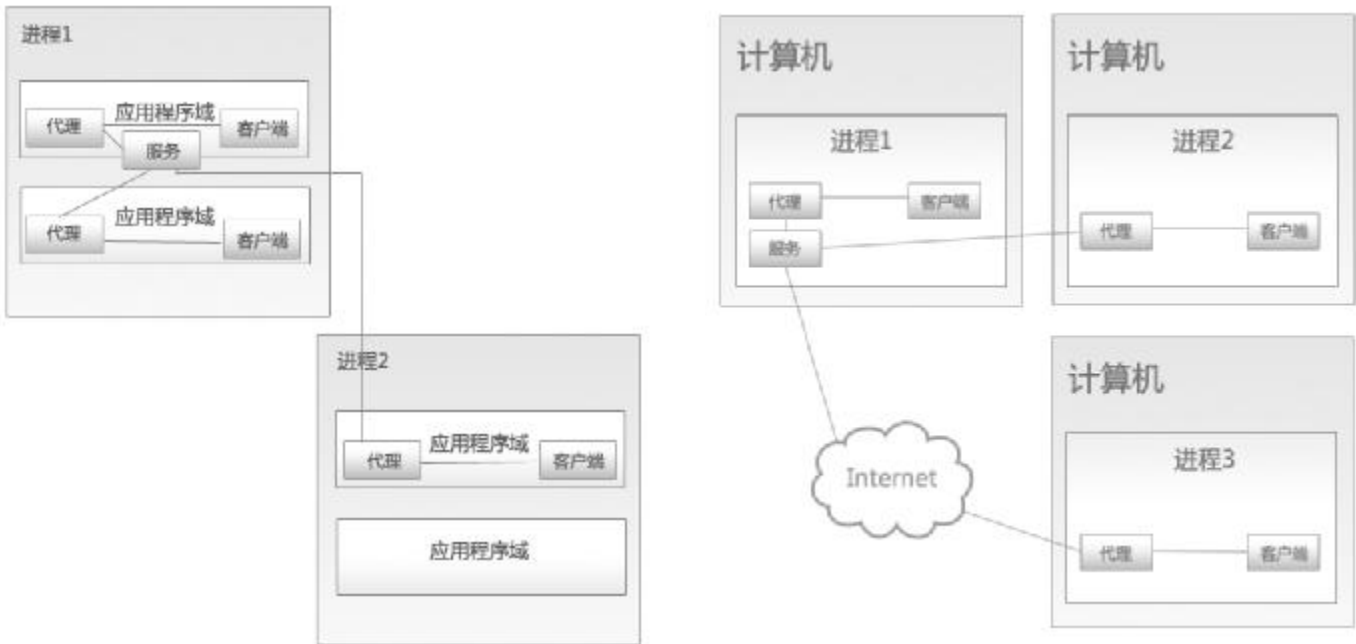


图 18-5 WCF 与本机服务进行通信

图 18-6 WCF 与远程服务进行通信

图 18-6 展示了 WCF 客户端与远程服务进行通信，无论 WCF 客户端是与远程服务进行通信还是与本地进程进行通信，都需要使用代理。

## 18.2.2 地址

在 Internet 中，为了标识每个计算机，就需要使用 IP 进行地址划分，在生活中也有此实例，例如每个家庭都有一个门牌号，为了方便找到某个人，则必须通过门牌号找到这个人，同样对于 WCF 服务而言，每个 WCF 服务都有一个自己的地址。

### 1. WCF 地址

WCF 地址包含两个元素，服务位置与传输协议，服务位置包括目标机器名、站点或网络、通信端口、管道或队列，以及一个可选的特定路径或者 URI。WCF 地址也可以是用于服务通信的传输样式。WCF 支持的传输样式包括：

- HTTP：超文本传输协议。
- TCP：传输控制协议。
- Peer network：对等网。
- IPC：基于命名管道的内部进程通信协议。
- MSMQ：微软消息队列。

地址通常通过[基地址]/[可选的 URI]的格式进行 WCF 地址描述，示例地址如下所示。

```
http://localhost:8731
http://localhost:8731/18-2
net.tcp://localhost:8731/server/18-2
net.pipe://localhost/18-2
net.msmq://localhost/18-2
```

其中关于 **http://localhost:8731** 这个地址可以称作使用 http 协议，访问计算机为 localhost 的端口 8731 正在等待客户端的调用。而对于 **http://localhost:8731/18-2** 这个地址可以称作使用 http 协议，访问计算机为 localhost 的端口为 8731 的 18-2 服务正在等待客户端的调用。

### 2. TCP 地址

TCP 地址使用 TCP 传输控制协议作为通信协议，使用 TCP 地址的示例地址如下所示。

```
net.tcp://localhost:8731/server/18-2
```

如果端口号没有指定，则 TCP 会使用默认端口号 808 作为其默认端口，示例地址如下所示。

```
net.tcp://localhost/server/18-2
```

### 3. HTTP 地址

HTTP 地址使用 HTTP 传输控制协议作为其通信协议，使用 HTTP 地址的示例地址如下所示。

```
http://localhost:8731/18-2
```

如果端口号没有指定，则 HTTP 会使用默认的端口号 80 作为其默认端口。

**注意：**无论是 TCP 协议还是 HTTP 协议，不同的服务可以公用相同的端口号。

### 4. IPC 和 MSMQ 地址

IPC 地址使用 net.tcp 作为通信协议，使用 net.tcp 地址的示例地址如下所示。

```
net.pipe://localhost/18-2
```

正是因为 IPC 地址使用 net.pipe 进行传输，所以 IPC 地址将使用 Windows 的命名管道机制。在 WCF 中，如果服务使用命名管道，则该服务只能接收来自同一台客户端计算机的调用。因此，在使用时必须明确的指定 WCF 提供服务的计算机名，从而为管道名提供一个惟一的标识字符串。而 MSMQ 地址使用 net.msmq 进行传输，即使用了微软消息队列机制，MSMQ 地址的示例地址如下所示。

```
net.msmq://localhost/18-2
```

18.2.3 契约

在 WCF 中，所有的 WCF 服务都会被公开成为契约。契约是服务的功能的标准描述方式，通常情况下 WCF 包含四种类型的契约，这些契约如下所示。

- ❑ 服务契约（**Service Contract**）：服务契约定义了客户端能够执行的操作，服务契约是 WCF 中使用最为广泛的一种契约。
- ❑ 数据契约（**Data Contract**）：数据契约定义了客户端与服务器之间交互的数据类型。
- ❑ 错误契约（**Fault Contract**）：错误契约定义了操作中出现的异常，包括定义服务出现的错误并传递返回给客户端。
- ❑ 消息契约（**Message Contract**）：消息契约允许服务直接与消息交互，但是 WCF 很少使消息契约。

WCF 使用特性 **ServiceContractAttribute** 标识服务契约，而使用 **OperationContractAttribute** 标识服务方法。示例代码如下所示。

```
[ServiceContract]                                     //标识服务契约
public interface IService1                             //实现接口
{
    [OperationContract]                               //方法声明
    string GetData(int value);                         //创建方法
    [OperationContract]
    CompositeType GetDataUsingDataContract(CompositeType composite);
    // 任务: 在此处添加服务操作
}
```

上述代码使用 **ServiceContractAttribute** 标识服务契约，而使用 **OperationContractAttribute** 标识服务方法，**OperationContract** 只能用于方法，指明客户端是否能够调用此方法。使用 **OperationContract** 标识可以标识私有方法以使用 **SOA** 的方式进行构架，虽然这样是可以实现客户端调用，但是作为面向对象的设计是不推荐使用该方法的。由于能够使用 **ServiceContractAttribute** 来标识服务契约，开发人员能够自定义标识指定相应的方法是否能够被客户端调用，示例代码如下所示。

```
[OperationContract]
CompositeType GetDataUsingDataContract(CompositeType composite); //标识方法
string Post(string content);
```

在上述代码中的 **Post** 方法不会成为契约。WCF 允许开发人员使用 **DataContractAttribute**、**DataMemberAttribute** 来标识自定义数据类型和属性，示例代码如下所示。

```
[DataMember]                                     //设置 DataMember
string stringValue = "Hello ";                  //创建 string 变量
[DataMember]                                     //设置 DataMember
public bool BoolValue                           //设置属性
{
    get { return boolValue; }
    set { boolValue = value; }                  //设置属性默认值
}
[DataMember]                                     //设置 DataMember
public string StringValue                       //设置属性
{
    get { return stringValue; }
    set { stringValue = value; }               //设置属性默认值
}
```

上述代码使用了 **DateMember** 定义了属性和相应的字段，这样就可以在服务方法中传递复杂的数据体了。



18.3 WCF 应用

在了解了基本的 WCF 概念后，先不用着急继续了解 WCF 应用体系，通过创建 WCF 应用可以深入的了解服务、地址和契约的概念。WCF 还允许开发人员创建和声明契约，通过契约的声明，客户端可以通过远程调用以实现自身的程序。

18.3.1 创建 WCF 应用

在 Visual Studio 2008 中，可以方便的创建 WCF 应用。在菜单栏中选择【文件】选项，在下拉菜单中单击【新建项目】选项，在弹出的【新建项目】窗口中选择 WCF，如图 18-7 所示。



图 18-7 创建 WCF 服务库

创建 WCF 服务库后，应用程序会自动生成 Server1.cs 和 IServer1.cs 接口，IServer1.cs 接口示例代码如下所示。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.Serialization;
using System.ServiceModel;
using System.Text;
namespace _18_2
{
    // 注意: 如果更改此处的接口名称“IService1”，也必须更新 App.config 中对“IService1”的引用。
    [ServiceContract]
    public interface IService1
    {
        [OperationContract]
        string GetData(int value);
        [OperationContract]
        CompositeType GetDataUsingDataContract(CompositeType composite);
        // 任务: 在此处添加服务操作
    }
    // 使用下面示例中说明的数据协定将复合类型添加到服务操作
    [DataContract]
    public class CompositeType
    {
    }
```

```
{
    bool boolValue = true;                                //创建字段
    [DataMember]
    string stringValue = "Hello ";                        //声明 string 变量
    [DataMember]
    public bool BoolValue                                //创建属性
    {
        get { return boolValue; }
        set { boolValue = value; }
    }
    [DataMember]
    public string StringValue                            //创建属性
    {
        get { return stringValue; }
        set { stringValue = value; }
    }
}
```

上述代码创建了一个 **IServer1** 接口，并通过 **ServiceContractAttribute** 标识服务契约，同样也通过 **DataContractAttribute**、**DataMemberAttribute** 来标识自定义数据类型和属性，示例代码如下所示。

```
[ServiceContract]                                //标识服务契约
public interface IService1                        //创建接口
{
    [DataContract]
    public class CompositeType
```

创建接口后则需要实现接口，接口的实现在 **Server.cs** 文件内，示例代码如下所示。

```
public class Service1 : IService1                //实现接口
{
    public string GetData(int value)              //实现接口方法
    {
        return string.Format("You entered: {0}", value); //返回 string 值
    }
    public CompositeType GetDataUsingDataContract(CompositeType composite) //实现接口方法
    {
        if (composite.BoolValue)                 //实现方法代码
        {
            composite.StringValue += "Suffix";    //添加相应数据
        }
        return composite;                        //返回值
    }
}
```

上述代码实现了 **IServer1** 接口中的方法，单击【运行】按钮或快捷键【F5】，**WCF** 应用程序就能够运行，如图 18-8 所示。

从图 18-8 中可以看出，在 **Server1.cs** 文件中实现的方法都能够在测试客户端中看到，单击测试客户端中相应的方法就能够在客户端测试调用服务器端的方法，如图 18-9 所示。



图 18-8 服务测试客户端

图 18-9 测试方法调用

双击 **GetData** 方法后，在右侧选项卡中就会分为两层，这两层分别为请求和响应。在请求框中可以在值那一栏编写需要传递的值，编写完毕后单击【调用】按钮就能够实现服务器端方法的调用并在响应框中呈现相应的值。

18.3.2 创建 WCF 方法

一个简单的 **WCF** 应用程序运行后，就会发现其实 **WCF** 并没有想象中的复杂。**WCF** 允许开发人员通过使用 **ServiceContractAttribute** 标识服务契约，同样开发人员还能够创建服务契约以提供客户端的方法的调用。在 **IServer1** 接口中首先需要定义该方法，示例代码如下所示。

```
int GetSum(DateTime time); //定义接口方法
[OperationContract] //标识调用
string GetShopInformation(string address);
```

上述代码声明了两个方法，这两个方法分别为 **GetSum** 和 **GetShopInformation**，**GetSum** 用于获取某一天麦当劳餐厅中出售总量，而 **GetShopInformation** 用于获取麦当劳地址和一些商店的信息，**GetSum** 和 **GetShopInformation** 具体实现如下所示。

```
public int GetSum(DateTime time) //实现方法
{
    int BreadNum = 10; //声明必要字段
    int Milk = 5; //声明必要字段
    int HotDryNuddle = 20; //声明必要字段
    int today = BreadNum + Milk + HotDryNuddle; //实现计算
    return today; //返回值
}
public string GetShopInformation(string address) //实现方法
{
    if (address == "武汉") //判断地址
    {
        return "武汉麦当劳连锁店"; //返回相应结果
    }
    else if (address == "北京") //判断地址
    {
        return "北京麦当劳连锁店"; //返回相应结果
    }
    else if (address == "上海") //判断地址
    {
        return "上海麦当劳连锁店"; //返回相应结果
    }
    else
    {
        return "没有该连锁店"; //返回默认结果
    }
}
```

在 **GetSum** 方法中，用于获取当天的销售总量，而 **GetShopInformation** 实现了商店信息的反馈，运行后如图 18-10 所示。

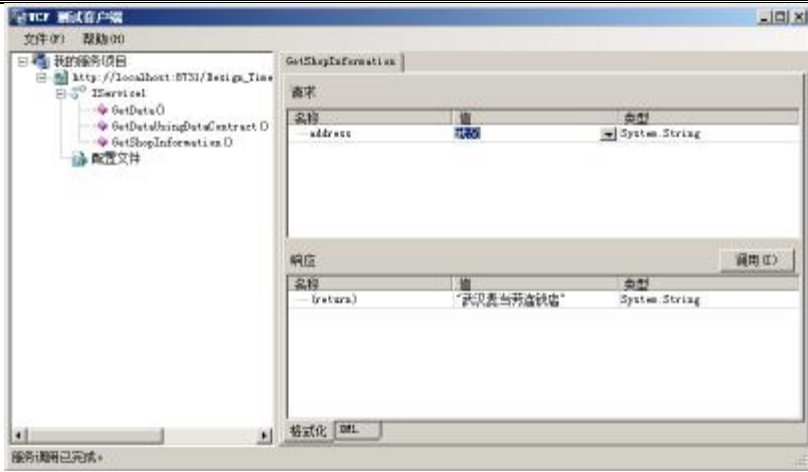


图 18-10 创建方法

从图 18-10 中可以看出，**GetShopInformation** 已经在测试客户端中服务器项目中显式了，并且输入了“武汉”这个信息，就能够返回“武汉麦当劳连锁店”。而 **GetSum** 方法并没有呈现在服务器项目中，这是因为 **GetSum** 方法并没有使用 **[OperationContract]** 标识进行声明，所以不会作为契约的一部分，若需要在客户端调用 **GetSum** 方法就必须使用 **[OperationContract]** 标识进行声明，示例代码如下所示。

```
[OperationContract]
int GetSum(DateTime time);                                     //标识客户端方法
```

WCF 应用程序运行后如图 18-11 所示。

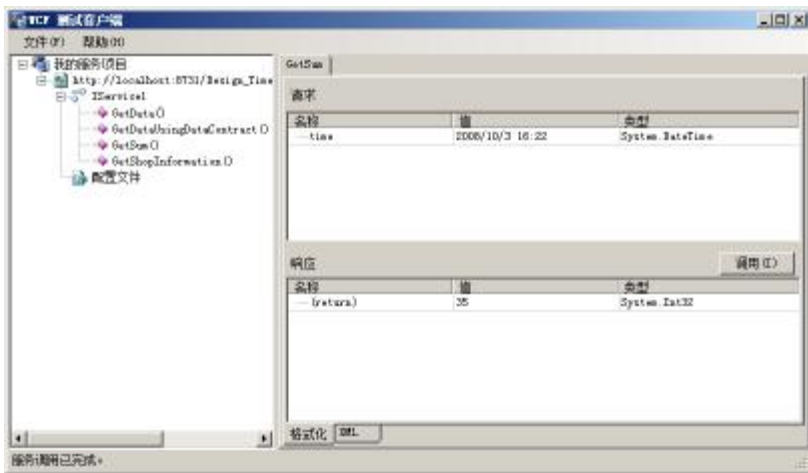


图 18-11 使用 **[OperationContract]** 标识

开发人员能够使用 **WCF** 快速的创建 **WCF** 应用程序并从客户端调用该方法，这样就能够在客户端隐藏服务器方法并且让客户端只关注逻辑实现而不需要关注底层是如何进行消息通信的。

## 18.4 WCF 消息传递

通过了解了 **WCF** 的一些基本概念并创建和编写 **WCF** 应用中的相应方法，实现了 **WCF** 服务和客户端之间的调用，就能够理解 **WCF** 应用是如何进行通信的。了解了一些基本的 **WCF** 概念后，还需要深入了解 **WCF** 消息的概念。

### 18.4.1 消息传递

客户端与服务器之间是通过消息进行信息通信的，通过使用消息，客户端和服务端之间能够通过使用消息交换来实现方法的调用和数据传递。

#### 1. Request/Reply 消息传递模式

**Request/Reply** 模式是默认的消息传递模式，该模式调用服务器的方法后需要等待服务的消息返回，从而获取服务器返回的值。**Request/Reply** 模式是默认模式，在声明时无需添加其模式的声明，示例代码如下所示。



[OperationContract]  
string GetShopInformation(string address);  
//默认模式

上述代码就使用了一个默认的 **Request/Reply** 模式进行消息传递，**GetShopInformation** 方法同样需要实现，示例代码如下所示。

```
public string GetShopInformation(string address)
{
    if (address == "武汉") //判断地址
    {
        return "武汉麦当劳连锁店"; //返回相应结果
    }
    else if (address == "北京") //判断地址
    {
        return "北京麦当劳连锁店"; //返回相应结果
    }
    else if (address == "上海") //判断地址
    {
        return "上海麦当劳连锁店"; //返回相应结果
    }
    else
    {
        return "没有该连锁店"; //返回默认结果
    }
}
```

**GetShopInformation** 方法返回一个 **string** 的值给客户端，客户端调用服务器的方法时，首先会向服务器发送消息，以告诉服务器客户端需要调用一个方法，当服务器接收消息后会返回消息给客户端。在这一段过程中，客户端会等待服务器端的相应，当客户端接受到服务器的相应后，则会呈现在客户端应用程序中。如图 18-12 所示。

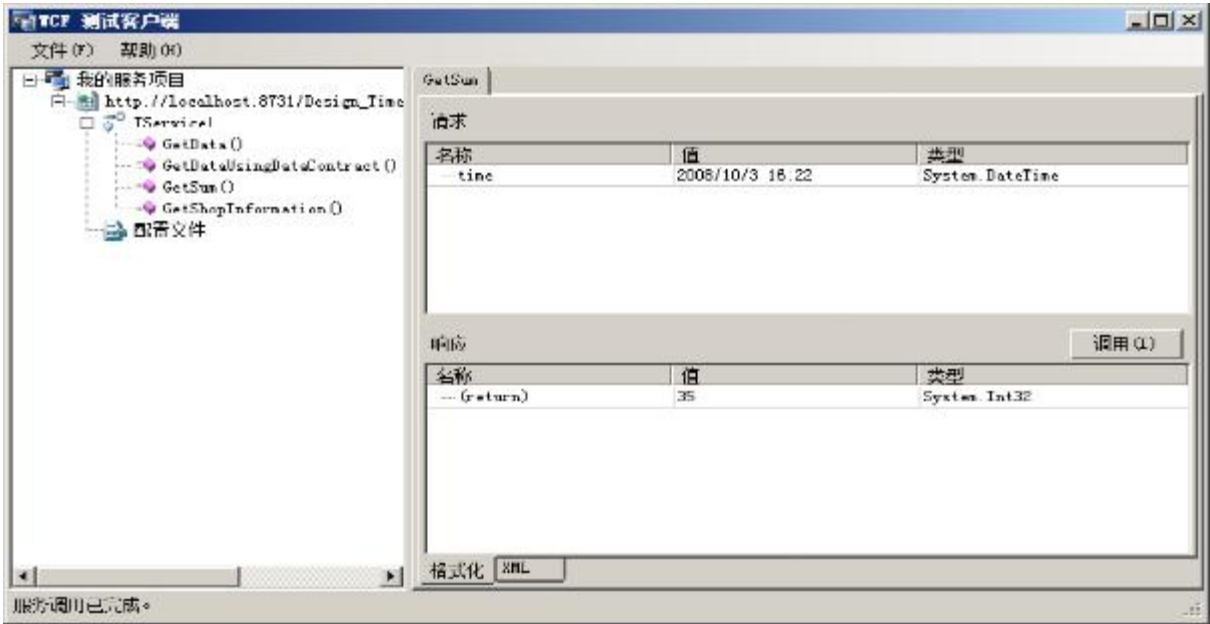


图 18-12 Request/Reply 模式

2. One-way 消息传递模式

**One-way** 模式和 **Request/Reply** 模式不同的是，如果使用 **One-way** 模式定义一个方法，该方法被调用后会立即返回。使用 **One-way** 模式修饰的方法必须是 **void** 方法，如果该方法不是 **void** 修饰的方法或者包括 **out/ref** 等参数，则不能使用 **One-way** 模式进行修饰，示例代码如下所示。

[OperationContract(IsOneWay = true)]  
void OutputString();  
//标识 One-way 模式  
//定义方法

该方法使用了 **One-way** 模式，则不能有参数的输出，只允许 **void** 关键字修饰该方法，**OutputString** 方法的具体实现如下所示。

```
public void OutputString() //实现方法
{
    Console.WriteLine("IsOneWay=true");
}
```

}  
运行 WCF 应用后，执行 **OutputString** 方法后结果如图 18-13 所示。

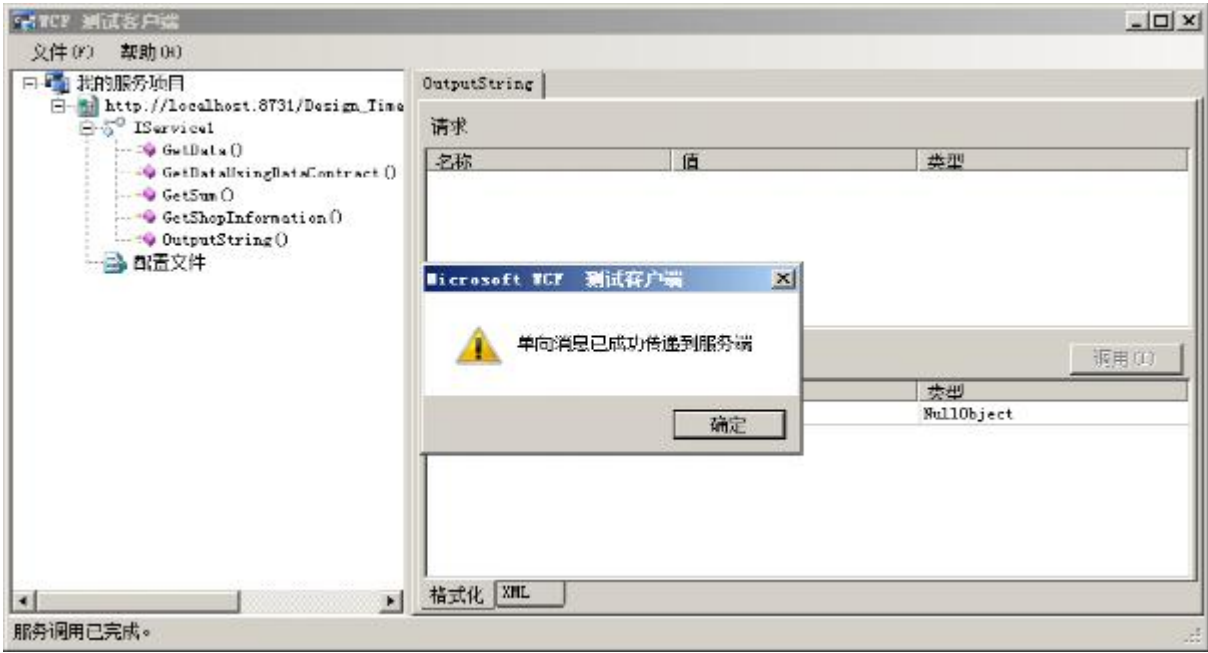


图 18-13 One-way 模式

WCF 的消息传递模式不仅包括这两种模式，还包括 **duplex** 模式，**duplex** 是 WCF 消息传递中比较复杂的一种模式，由于篇幅限制，本书不再进行详细的介绍。

18.4.2 消息操作

由于 WCF 的客户端和服务端之间都是通过消息响应和通信的，那么在 WCF 应用的运行过程中，消息是如何在程序之间进行操作的，这就需要通过 XML 文档来获取相应的结果。在客户端和服务端之间出现信息通信，并且客户端调用了服务器的方法时，就会产生消息，如 **GetSum** 方法。**GetSum** 方法在接口中的代码如下所示。

```
[OperationContract]                                     //标识方法
int GetSum(DateTime time);                                //定义方法
```

在 **GetSum** 方法的实现过程中，只需要进行简单的操作即可，示例代码如下所示。

```
public int GetSum(DateTime time)                            //实现方法
{
    int BreadNum = 10;                                     //声明必要字段
    int Milk = 5;                                           //声明必要字段
    int HotDryNuddle = 20;                                  //声明必要字段
    int today = BreadNum + Milk + HotDryNuddle;             //实现计算
    return today;                                           //返回值
}
```

上述代码执行后，客户端会调用服务器的 **GetSum** 方法，服务器接受响应再返回给客户端相应的值，如图 18-14 和图 18-15 所示。

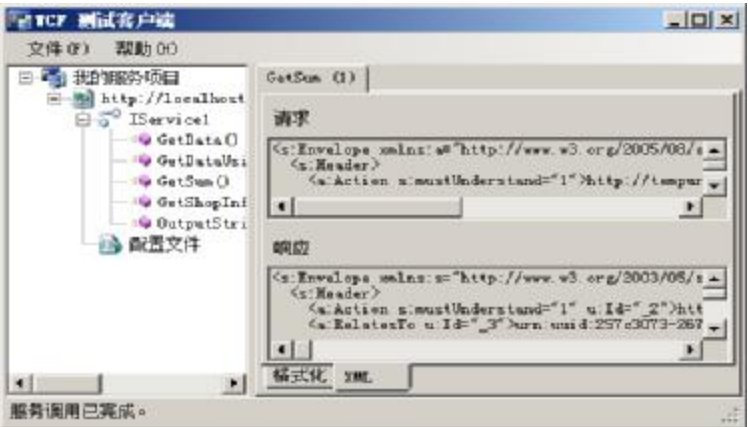
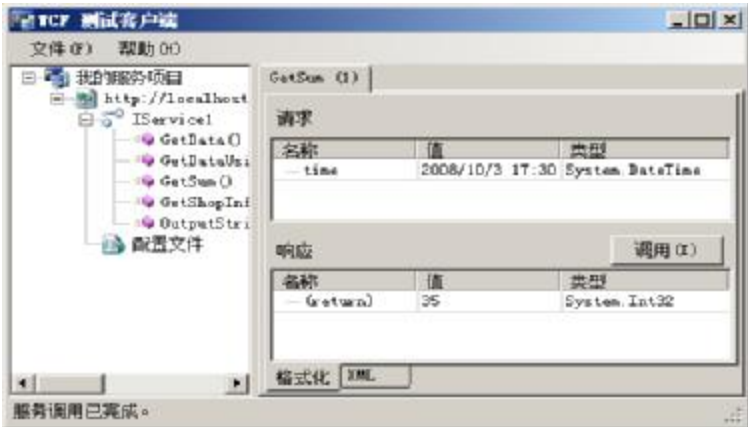


图 18-14 执行服务器方法

图 18-15 返回的 XML 格式文档

在运行后，测试客户端能够获取请求时和响应时的 XML 文档，其中请求时产生的 XML 文档如下所示。

```
<s:Envelope
  xmlns:a="http://www.w3.org/2005/08/addressing" xmlns:s="http://www.w3.org/2003/05/soap-envelope">
  <s:Header>
    <a:Action s:mustUnderstand="1">http://tempuri.org/IService1/GetSum</a:Action>
    <a:MessageID>urn:uuid:dcc8a76e-deaf-45c4-a80c-2034b965d001</a:MessageID>
    <a:ReplyTo>
      <a:Address>http://www.w3.org/2005/08/addressing/anonymous</a:Address>
    </a:ReplyTo>
  </s:Header>
  <s:Body>
    <GetSum xmlns="http://tempuri.org/">
      <time>2008-10-03T17:30:00</time>
    </GetSum>
  </s:Body>
</s:Envelope>
```

从上述代码可以看到在 **Action** 节中，使用了相应的方法 **GetSum**，在 **WCF** 服务库编程中可以通过使用 **OperationContract.Action** 捕获相应的 **Action** 消息，示例代码如下所示。

```
[OperationContract(Action = "GetSum", ReplyAction = "GetSum")]
```

```
Message MyProcessMessage(Message m);
```

**MyProcessMessage** 实现示例代码如下所示。

```
public Message MyProcessMessage(Message m)
{
    CompositeType t = m.GetBody<CompositeType>();           //获取消息
    Console.WriteLine(t.StringValue);                       //输出消息
    return Message.CreateMessage(MessageVersion.Soap11,
        "Add", new CompositeType("Hello World!"));         //返回消息
}
```

上述代码将操作转换为消息后发送，开发人员可以通过 **Windows** 应用程序或 **ASP.NET** 应用程序获取修改后消息的内容。在进行消息的操作时，**WCF** 还允许开发人员使用 **MessageContractAttribute** / **MessageHeaderAttribute** 来控制消息格式，这比 **DataContractAttribute** 要更加灵活。

## 18.5 使用 WCF 服务

创建了一个 **WCF** 服务之后，为了能够方便的使用 **WCF** 服务，就需要在客户端远程调用服务器端的 **WCF** 服务，使用 **WCF** 服务提供的方法并将服务中方法的执行结果呈现给用户，这样保证了服务器的安全性和代码的隐秘性。

### 18.5.1 在客户端添加 WCF 服务

为了能够方便的在不同的平台，不同的设备上使用执行相应的方法，这些方法不仅不能够暴露服务器地址，同样需要在不同的客户端上能呈现相同的效果，这些方法的使用和创建不能依赖本地的应用程序，为了实现跨平台的安全应用程序开发就需要使用 **WCF**。

创建了 **WCF** 服务，客户端就需要进行 **WCF** 服务的连接，如果不进行 **WCF** 服务的连接，则客户端无法知道在哪里找到 **WCF** 服务，也无法调用 **WCF** 提供的方法。首先需要创建一个客户端，客户端可以是 **ASP.NET** 应用程序也可以是 **WinForm** 应用程序。右击解决方案管理器，单击【项目】，在下拉菜单中选择【添加新项】，分别为该项目添加一个 **ASP.NET** 应用程序和一个 **WinForm** 应用程序，如图 18-16 和图 18-17 所示。



图 18-16 添加 Win Form 应用程序



图 18-17 添加 ASP.NET 应用程序

添加完成后在项目中就会出现这两个项目，分别为这两个项目添加 **WCF** 引用，右击当前项目，在下拉菜单中单击【添加服务引用】选项，在弹出窗口中单击【发现】按钮，即可发现 **WCF** 服务，如图 18-18 所示。添加完成后 **WCF** 服务就会被挂起，等待客户端对 **WCF** 服务中的方法进行调用，如图 18-19 所示。

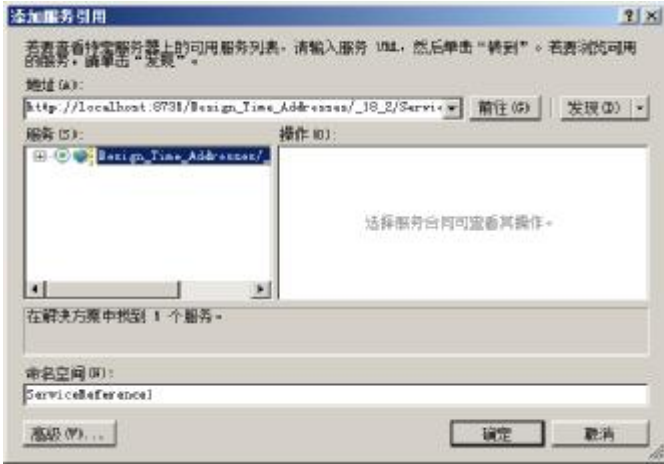


图 18-18 添加服务引用

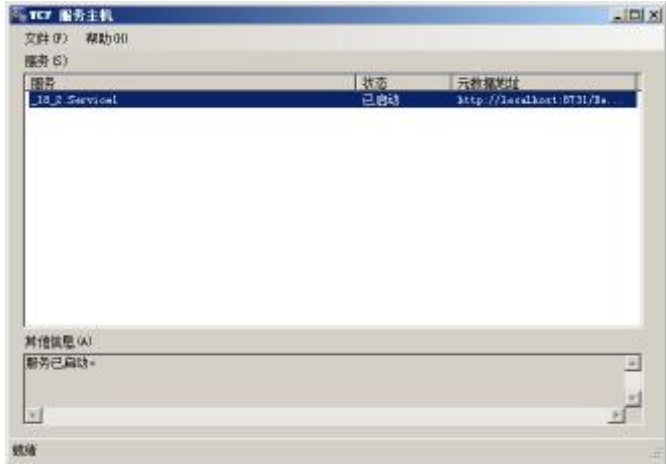


图 18-19 WCF 服务主机已经启动

分别为 **ASP.NET** 应用程序和 **Win Form** 应用程序添加 **WCF** 引用后，就可以在相应的应用程序中使用 **WCF** 服务提供的方法了。

18.5.2 在客户端使用 WCF 服务

当客户端添加了 **WCF** 服务的引用后，就能够非常方便的使用 **WCF** 服务中提供的方法进行应用程序开发。在客户端应用程序的开发中，几乎看不到服务器端提供的方法的实现，只能够使用服务器端提供的方法。对于客户端而言，服务器端提供的方法是不透明的。

1. ASP.NET 客户端

在 **ASP.NET** 客户端中，可以使用 **WCF** 提供的服务实现相应的应用程序开发，例如通过地名获取麦当劳的商店的信息，而不想要在客户端使用数据库连接字符串等容易暴露服务器端的信息，通过使用 **WCF** 服务提供的方法能够非常方便的实现这一点。Aspx 页面看代码如下所示。

```
<body>
  <form id="form1" runat="server">
    <div>
      输入地名 : <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
      <br />
      获得的结果: <asp:TextBox ID="TextBox2" runat="server"></asp:TextBox>
      <br />
      <asp:Button ID="Button1" runat="server" onclick="Button1_Click" Text="检索" />
    </div>
  </form>
</body>
```



上述代码在页面中拖放了两个 **Textbox** 控件分别用于用户输入和用户结果的返回，并拖放了一个按钮控件用于调用 **WCF** 服务中的方法并返回相应的值。**.cs** 页面代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    if (!String.IsNullOrEmpty(TextBox1.Text))
    {
        //开始使用 WCF 服务
        ServiceReference1.Service1Client ser = new Web.ServiceReference1.Service1Client();
        TextBox2.Text = ser.GetShopInformation(TextBox1.Text);           //实现方法
    }
    else
    {
        TextBox2.Text = "无法检索,字符串为空";                        //输出异常提示
    }
}
```

上述代码创建了一个 **WCF** 服务所提供的类的对象，通过调用该对象的 **GetShopInformation** 方法进行本地应用程序开发。上述代码运行后如图 18-20 和图 18-21 所示。



图 18-20 实现检索功能



图 18-21 实现异常处理

## 2. Win Form 客户端

在 **Win Form** 客户端中使用 **WCF** 提供的服务也非常的方便，其使用方法基本同 **ASP.NET** 相同，这也说明了 **WCF** 应用的开发极大的提高了开发人员在不同客户端之间的开发效率，节约了开发成本。在 **Win Form** 客户端中拖动一些控件作为应用程序开发提供基本用户界面，示例代码如下所示。

```
private void InitializeComponent()
{
    this.textBox1 = new System.Windows.Forms.TextBox();           //创建 textBox
    this.dateTimePicker1 = new System.Windows.Forms.DateTimePicker(); //创建 TimePicker
    this.SuspendLayout();
    //
    // textBox1
    //
    this.textBox1.Location = new System.Drawing.Point(13, 13);    //实现 textBox 属性
    this.textBox1.Name = "textBox1";                             //实现 textBox 属性
    this.textBox1.Size = new System.Drawing.Size(144, 21);       //实现 textBox 属性
    this.textBox1.TabIndex = 0;                                  //实现 textBox 属性
    //
    // dateTimePicker1
    //
    this.dateTimePicker1.Location = new System.Drawing.Point(166, 13); //实现 TimePicker 属性
    this.dateTimePicker1.Name = "dateTimePicker1";               //实现 TimePicker 属性
    this.dateTimePicker1.Size = new System.Drawing.Size(114, 21); //实现 TimePicker 属性
    this.dateTimePicker1.TabIndex = 1;                           //实现 TimePicker 属性
    this.dateTimePicker1.ValueChanged
```

```

+= new System.EventHandler(this.dateTimePicker1_ValueChanged);
//
// Form1
//
this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 12F);           //实现 Form 属性
this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;               //实现 Form 属性
this.ClientSize = new System.Drawing.Size(292, 62);                     //实现 Form 属性
this.Controls.Add(this.dateTimePicker1);                              //添加 Form 控件
this.Controls.Add(this.textBox1);                                       //添加 Form 控件
this.Name = "Form1";                                                    //实现 Form 属性
this.Text = "Form1";                                                    //实现 Form 属性
this.ResumeLayout(false);
this.PerformLayout();
}

```

上述代码在 **Win From** 窗体中创建了一个 **TextBox** 控件和一个 **DateTimePicker** 控件，并向窗体注册了 **dateTimePicker1\_ValueChanged** 事件，当 **DateTimePicker** 控件中的值改变后，则会输出相应天数的销售值。在前面的 **WCF** 服务中，为了实现销售值统计，创建了一个 **GetSum** 方法，在 **Win From** 窗体中无需再实现销售统计功能，只需要调用 **WCF** 服务提供的方法即可，示例代码如下所示。

```

private void dateTimePicker1_ValueChanged(object sender, EventArgs e)
{
    ServiceReference1.Service1Client ser = new WindowsForm.ServiceReference1.Service1Client();
    textBox1.Text = ser.GetSum(Convert.ToDateTime(dateTimePicker1.Text)).ToString();
}

```

上述代码使用了 **WCF** 服务中提供的 **GetSum** 方法进行了相应天数的销售额的统计，运行后如图 18-22 所示。



图 18-22 Win From 客户端使用 WCF 服务

创建和使用 **WCF** 服务不仅能够实现不同客户端之间实现相同的功能，还通过 **WCF** 应用提供了一个安全性、可依赖、松耦合的开发环境，对于其中任何一种客户端的实现，都不会暴露服务器中的私密信息，并且对于其中的某个客户端进行任何更改，也不会影响其他客户端，更不会影响到 **WCF** 服务器，这对应用程序开发和健壮性提供了良好的环境。

## 18.6 小结

本章简单的介绍了 **WCF** 的基本知识，包括什么是 **WCF** 和为什么需要 **WCF**。**WCF** 在现在的中大型应用程序开发中起到了非常重要的作用，使用 **WCF** 技术能够实现分布式的应用程序开发和管理，**WCF** 为应用程序开发提供了安全、可依赖和松耦合的开发环境。本章还包括：

- ❑ **WCF 基础**：讲解了基本的 **WCF** 知识，包括 **WCF** 技术的组成和为何需要 **WCF**。
- ❑ **WCF 应用**：通过实例讲解了如何创建一个 **WCF** 应用，并修改和创建相应的方法实现 **WCF** 服务中的方法。
- ❑ **WCF 消息传递**：简单的介绍了 **WCF** 消息传递的几种模式和消息操作。
- ❑ **使用 WCF 服务**：介绍了如何在客户端中添加 **WCF** 引用并使用 **WCF** 引用。

本章只是简单的讲解了 **WCF** 的基础知识，本书并不是专门进行 **WCF** 知识的讲解，而且 **WCF** 知识需要一定的开发经验和编程水平，所以本书并没有详细的讲解 **WCF** 技术。但是在 **ASP.NET** 开发中，一些中大型的项目同样需要使用 **WCF** 进行分布式应用，从而实现不同的客户端对 **ASP.NET** 应用的访问，了解基本的 **WCF** 知识在今后的大型项目开发中会起到良好的作用。

## 第 19 章 WPF 开发基础

在 **Vista** 和 **Windows Seven** 火热发布的今天，很多用户都被 **Vista** 的特效所吸引，**Vista** 和的 **Windows Seven** 的 3D 特效，以及毛玻璃等效果给操作系统带来了更新更好的用户体验，在这一系列功劳的背后，**WPF** 占据着不小的功劳。

### 19.1 了解 WPF

**WPF (Windows Presentation Foundation)** 原代号为“**Avalon**”，是微软的新一代图形系统。**WPF** 基于 **.NET 3.0** 构架，为开发人员进行 **Windows** 应用程序开发和 **2D/3D** 图形和多媒体提供了统一的描述方法。对于开发人员而言，**WPF** 开发非常的简单，只要开发人员有一定的 **.NET** 基础，都能够快速上手 **WPF** 应用程序开发。

#### 19.1.1 什么是 WPF

**WPF (Windows Presentation Foundation)** 是微软的新一代图形系统，为用户界面、**2D/3D** 图形、文档和媒体提供了统一的描述和操作方法。基于 **DirectX 9** 和 **Direct 10** 技术的 **WPF** 不仅带来了非常绚丽的 **3D** 界面，而且其图形向量渲染引擎也大大改进了传统的 **2D** 界面，使得传统的 **2D** 界面可以模拟毛玻璃、**3D** 等特效。

对于开发人员而言 **WPF** 提供了统一的 **Windows Form** 应用程序开发方法，并且开发人员通过使用 **WPF** 技术，能够使得 **Windows Form** 应用程序像动画一样展现在用户面前，用户能够得到良好的用户体验。**WPF** 包含两个部分，这两个部分分别为引擎和编程框架。

##### 1. WPF 引擎

**WPF** 引擎为开发人员和设计人员提供了统一的设计文档，开发人员能够像普通的 **Windows Form** 应用程序一样进行逻辑编程，设计人员能够通过使用 **XAML** 语言描述 **Windows Form** 应用程序中各个控件的风格，以实现动画效果。

**WPF** 引擎还为设计人员提供了基于浏览器的体验、基于窗体的应用程序、图形、视频、音频和文档提供了一个单一的运行时库，**WPF** 让传统的 **Windows Form** 应用程序能够利用起现有的硬件软件资源，充分的利用 **Direct** 功能和硬件的编码解码功能进行窗体和控件的渲染。

##### 2. WPF 框架

**WPF** 框架为媒体、用户界面设计和文档提供的解决方案比开发人员现有的解决方案都要好，**WPF** 框架在设计时考虑了可扩展性和可维护性，开发人员能够在 **WPF** 中创建自己的控件，还可以通过对现有的 **WPF** 控件进行改造创建新的 **WPF** 控件。

**WPF** 框架是用于形状、图像、视频、动画、文档、三维，以及用于放置控件和内容的面板的一系列控件，这些控件和内容的面板的一系列控件是 **WPF** 框架的核心。**WPF** 应用程序提供了若干 **WPF** 应用程序开发所需要的控件，开发人员同样能够对控件进行拖放操作实现应用程序布局 and 开发。

##### 3. XAML 基本概念

**WPF** 应用程序引入了 **XAML**，**XAML** 是基于 **XML** 文档格式的一种标记语言，**XAML** 能够描述 **Windows** 应用程序和用户界面。开发人员和设计人员能够使用 **XAML** 语言进行代码和界面布局的可重用性控制。而



对于 **Web** 开发者而言，**XAML** 是基于标记语言的，**XAML** 同样包括属性描述，对于 **Web** 开发者，也能够轻松的使用 **XAML** 描述 **WPF** 应用程序。

**WPF**（**Windows Presentation Foundation**）为开发人员和设计人员提供了统一的图形、图像、界面、文档等设计和开发的统一的运行和操作方法，**WPF** 使现有的 **Window** 应用程序能够充分的利用硬件软件的资源进行应用程序窗口渲染和优化，给用户以全新的 **Windows** 窗体应用程序体验。

## 19.2 WPF 的应用范围

在现有的 **Window** 应用程序中，对于已经成熟的传统的 **WinForm** 应用程序而言，为何还要抛弃现有的成熟技术而使用 **WPF** 技术开发 **Window** 应用程序呢？在传统 **Window** 应用程序开发中，应用程序的表现形式往往是非常死板的，应用程序窗体很难实现像 **Web** 应用和 **Flash** 中的渲染效果，例如图形图像的渲染和文本的渲染。虽然现今对渲染的方法有很多其他的解决方案，包括遨游等浏览器的 **JavaScript** 渲染，但是这些都是将 **Window** 应用程序和 **Web** 应用程序整合的解决方案，并没有完全的解决 **Window** 应用程序中对窗体本身的渲染的困难问题。

在 **Vista** 应用程序开发中，**Vista** 将应用程序窗体进行了效果的渲染，并没有使用 **Web** 应用的解决方案，直接通过 **WPF** 进行窗体和控件的渲染，实现了半透明等效果，让用户耳目一新，提高了用户体验。如图 19-1 所示。



图 19-1 Windows 窗体图形渲染

随着互联网和硬件的发展，显卡等硬件已经能够辅助 **CPU** 的运算实现动态解码，让 **CPU** 的使用率变得更低，让 **CPU** 专注处理内核运算，从而能够让网络游戏等大型的图形操作和运算的应用程序能够使用显卡的解码技术流畅运行。

使用 **WPF** 也能够使用显卡的硬件进行应用程序渲染加速，这也能够让 **WPF** 应用程序不会占用过多的 **CPU** 资源，**WPF** 应用程序能够基于 **Direct9/10** 进行图形图像编程，而使用显卡加速能够充分的利用 **Direct9/10** 的资源提升应用程序的用户体验。

## 19.2 WPF 和 Microsoft Expression

在进行 **WPF** 应用程序的开发中，需要编写相应的 **XAML** 文档进行窗体的布局和渲染，在 **Visual Studio 2008** 中，并没有提供很好的支持 **WPF** 应用程序设计所需要的功能，例如动画操作和图形渲染。微软提供了 **Microsoft Expression** 软件套装，在 **Microsoft Expression** 软件套装中可以使用 **Microsoft Expression Blend 2** 进行 **WPF** 应用程序窗体的布局和渲染。



## 19.2.1 使用 Microsoft Expression Blend 设计 WPF

Microsoft Expression Studio 2 软件套装中，微软提供了 Microsoft Expression Blend 2，用于提供 WPF 应用程序和 Silverlight 应用程序的图形开发和渲染，双击 Microsoft Expression Blend 2 图标打开 Microsoft Expression Blend 2 应用程序，如图 19-2 所示。

Microsoft Expression Blend 2 和 Microsoft Expression Studio 2 软件套装一样，其的界面也是以黑色为主的界面。对于设计人员而言，设计人员更加偏好黑色界面以便将图形图像突出的显式在屏幕中。单击【新建】按钮，Microsoft Expression Blend 2 会弹出一个新建框，开发人员能够选择相应的应用程序进行开发，如图 19-3 所示。



图 19-2 Microsoft Expression Blend 2

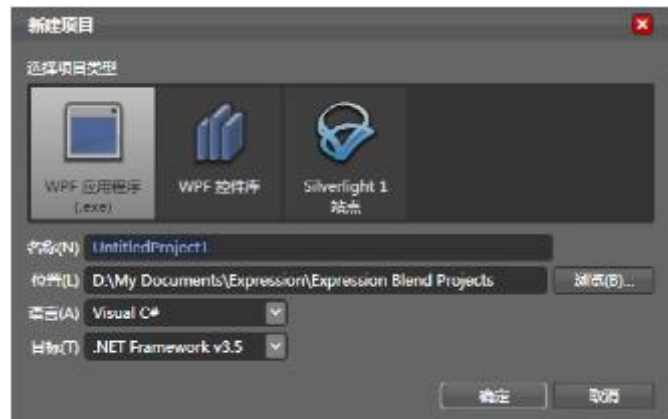


图 19-3 创建新建项

单击【WPF 应用程序】按钮，并选择相应的位置就能够创建 WPF 应用程序。WPF 应用程序创建后，对于开发人员和设计人员而言，其 Windows 应用程序开发窗口很像一张画布。在 WPF 应用程序中，窗体可以想象成是一个画布，这个画布能够承载多媒体、图形、图像甚至是动画。WPF 应用程序提供了默认控件，这些控件包括最常用的 Button 按钮控件，下拉框控件以便开发人员提供用户交互功能，如图 19-4 所示。

从图 19-4 中可以看出，WPF 应用程序提供了 Border、ListBox、Button、Label 等常用控件，在 Microsoft Expression Blend 2 中，同样可以直接拖放到窗体中进行窗体布局，如图 19-5 所示。



图 19-4 WPF 资源库



图 19-5 WPF 应用程序窗体布局

Microsoft Expression Blend 2 同 Visual Studio 2008 相似，Microsoft Expression Blend 2 允许开发人员直接向窗体中拖动控件以实现控件的布局，但是 Microsoft Expression Blend 2 并不支持控件事件的响应。当开发人员在 Microsoft Expression Blend 2 中双击 Button 按钮控件时，并不会在相应的代码中自动创建方法，Microsoft Expression Blend 2 仅仅为 WPF 应用程序布局提供了良好的支持，若需要为 WPF 应用程序编写事件，需要同 Visual Studio 2008 相配合。

## 19.2.2 WPF 控件样式

使用 Microsoft Expression Blend 2 进行 WPF 应用程序开发很类似与 Photoshop 中进行图形图像编程。在 Photoshop 中进行图形图像编程时，可以针对某一个图形进行渲染，包括半透明、颜色和渐变等。在传统

的 **Windows** 应用程序的开发中，如果需要让应用程序的背景或者某个按钮控件像动画一样呈现出渐变和半透明效果是非常困难的，在 **WPF** 中可以进行类似 **Photoshop** 的操作对 **WPF** 应用程序中的控件进行样式控制，如图 19-6 和图 19-7 所示。



图 19-6 属性控制面板



图 19-7 外观控制

图 19-6 和图 19-7 都是针对一个控件进行样式控制。使用 **Microsoft Expression Blend 2** 进行控件样式开发的过程中，每一个控件都包含一个属性面板，属性面板用于 **WPF** 应用程序中控件的样式的控制。在 **WPF** 中，窗体都是基于 **XAML** 文档进行编写和样式控制的，如果需要使用 **XAML** 文档进行样式开发和控制，不得不记住很多属性，这样就让 **WPF** 应用程序的开发变得非常困难。在 **Microsoft Expression Blend 2** 中使用属性控制面板能够快捷的定义相应控件的属性。

使用 **Microsoft Expression Blend 2** 进行应用程序中进行样式控制非常容易。在 **Photoshop** 中对图形图像的编程可以直接使用画笔或渐变等工具进行样式控制，同样在 **Microsoft Expression Blend** 中可以像在 **Photoshop** 中一样进行属性配置就可以实现控件的不同样式的布局，如图 19-8 所示。



图 19-8 Button 控件样式控制

在图 19-8 中，可以通过样式选择选择不同的控件样式，从左到右分别为渐变样式、平铺样式、纯色样式和半透明样式。使用 **XAML** 进行样式控制示例代码如下所示。

```
</Button>
<Button d:LayoutOverrides="VerticalAlignment" HorizontalAlignment="Right"
  Margin="0,0,519.113,65" VerticalAlignment="Bottom" Width="104.887"
  Height="54.837" Content="Button">
  <Button.Background>
    <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
      <GradientStop Color="#FFC8C8C8" Offset="0"/>
      <GradientStop Color="#FFFFFFFF" Offset="1"/>
    </LinearGradientBrush>
  </Button.Background>
</Button>
```

上述 **XAML** 代码就描述了一个渐变控件，其中为控件定义了基本的属性，包括宽度和高度。在子节点中，通过使用 **LinearGradientBrush**, **GradientStop** 等属性实现了控件的效果。这些样式在传统的 **Windows** 应用程序开发中要实现是非常繁琐和困难的，而在 **WPF** 应用程序中能够方便的实现渐变和图形控件。

在属性控制面板中还包括很多其他的属性配置，这些属性包括不透明度、宽度、高度、文本对齐方式、窗体状态等，极大的方便了开发人员在开发过程中对样式的控制和封装，简化了开发人员对于窗体界面的

开发。

### 19.2.3 浅谈 XAML

使用 **Visual Studio 2008** 打开项目，就会发现 **WPF** 应用程序是通过使用 **XAML** 文档进行描述的。**XAML** 是 **eXtensible Application Markup Language** 的英文缩写，其中文名称为可扩展应用程序标记语言，它是微软公司为构建应用程序用户界面而创建的一种新的描述性语言。**WPF** 应用程序中大量的使用了 **XAML** 对应用程序窗体进行描述。

**XAML** 是一种基于 **XML** 文档格式的标记语言。在 **WPF** 应用程序中，**XAML** 用于描述窗体的样式、规则，以及事件的声明等，这种开发模型和 **ASP.NET** 中的代码隐藏页模型十分类似，这样就让 **XAML** 提供了一种便于扩展和定位的语法来定义和程序逻辑分离的用户界面的开发模型。

由于 **XAML** 是一种标记语言，所以开发人员能够很快的学习 **XAML** 并将 **XAML** 投入到项目开发中。学习 **XAML** 只需要开发人员具备一定的 **HTML** 知识就能够快速的学习 **XAML** 的语法格式和掌握 **XAML** 语法规范。

使用 **XAML** 进行窗体样式开发能够更加方便的协调开发人员和设计人员的工作，设计人员能够专注于窗体的样式控制使得开发人员能够专注于代码的编写，这样就有利于应用程序的开发。在 **WPF** 应用程序开发中，**XAML** 能够方便的对应用程序中的控件，样式进行描述，示例代码如下所示。

```
<Button HorizontalAlignment="Left" Margin="86,0,0,65" VerticalAlignment="Bottom"
Width="104.887" Height="54.837" Content="Button">
  <Button.Background>
    <DrawingBrush Stretch="Fill" TileMode="None" Viewbox="0,0,20,20" ViewboxUnits="Absolute">
      <DrawingBrush.Drawing>
        <DrawingGroup>
          <GeometryDrawing Brush="#FFD3D3D3">
            <GeometryDrawing.Geometry>
              <RectangleGeometry Rect="0,0,20,20"/>
            </GeometryDrawing.Geometry>
          </GeometryDrawing>
          <GeometryDrawing Brush="#FF000000">
            <GeometryDrawing.Geometry>
              <EllipseGeometry Center="0,0" RadiusX="10" RadiusY="10"/>
            </GeometryDrawing.Geometry>
          </GeometryDrawing>
          <GeometryDrawing Brush="#FF000000">
            <GeometryDrawing.Geometry>
              <EllipseGeometry Center="20,20" RadiusX="10" RadiusY="10"/>
            </GeometryDrawing.Geometry>
          </GeometryDrawing>
          <GeometryDrawing Brush="#FFFFFFFF">
            <GeometryDrawing.Geometry>
              <EllipseGeometry Center="20,0" RadiusX="10" RadiusY="10"/>
            </GeometryDrawing.Geometry>
          </GeometryDrawing>
          <GeometryDrawing Brush="#FFFFFFFF">
            <GeometryDrawing.Geometry>
              <EllipseGeometry Center="0,20" RadiusX="10" RadiusY="10"/>
            </GeometryDrawing.Geometry>
          </GeometryDrawing>
        </DrawingGroup>
      </DrawingBrush.Drawing>
    </DrawingBrush>
  </Button.Background>
</Button>
```

上述代码通过使用 **XAML** 语法编写了使用平铺画笔进行渲染的按钮控件，其格式是按照 **XML** 文档的格式进行声明和编写的。创建一个富媒体的半透明控件，同样可以通过 **XAML** 进行控制，免除了 **C++/Java** 中复杂的编程实现，示例代码如下所示。

```
<Button Margin="0,114.419,134.797,51.255" Content="Button"
    Background="#FFF7F7" HorizontalAlignment="Right" Width="148.887" Opacity="0.4"/>
```

上述代码就创建了一个半透明控件，该控件属性分别包括 **Margin**、**Content**、**Background**、**HorizontalAlignment**、**Width** 和 **Opacity** 等，其中 **Opacity** 属性用于控制控件的半透明度。使用 **XAML** 文档进行 **WPF** 应用程序的样式控制可以实现可扩展的样式控制，设计人员可以通过直接对 **XAML** 文档的修改实现不同的样式控制。**WPF** 应用程序中的任何控件都包括一些属性，这些属性能够方便的对控件进行样式控制，免除了复杂的编码实现。

### 19.2.4 WPF 控件层次

同 **Photoshop** 中的画布相同，**WPF** 应用程序中也有层次分别，每一个控件都包含一个自己的层次，如图 19-9 所示。

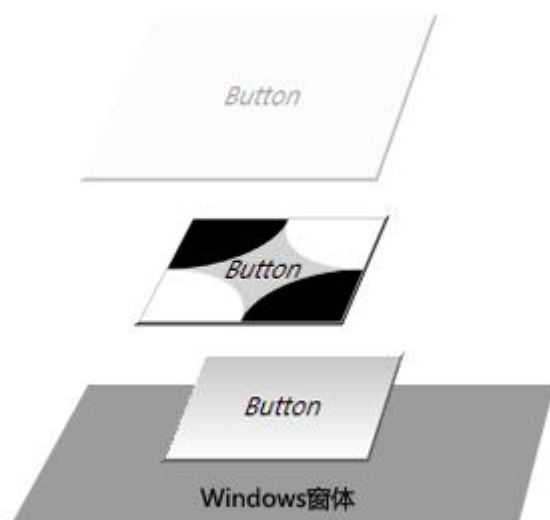


图 19-9 WPF 样式层次结构

无论是在 **WPF** 应用还是在 **Win From** 应用开发中，控件之间都有层次结构，如图 19-10 中所示，窗体的层次最低，用于承载控件。在窗体上的控件都是按照一个的层次堆叠在一起的，层次高的控件会遮住层次低的控件，在 **WPF** 布局中需要注意层次之间的控件的布局，如图 19-10 所示。



图 19-10 两个不同层次叠放的控件

在 **Microsoft Expression Blend 2** 中，包含层次管理器方便对层次进行管理。创建 **WPF** 应用程序后，系统会默认创建一个层，该层作为基层而存在，而控件都是基于该层而存在的，如图 19-11 所示。





图 19-11 层次管理



图 19-12 显式和隐藏层

在层次管理器中，可以设置相应的控件是否显式或隐藏，以便在大量的控件中准确的选取相应的控件，如图 19-12 所示。使用层次管理能够方便管理控件所在的层次并且为设计人员提供了控件选取的遍历，在层次管理中还可以选择锁定层或解除层，当一个层被锁定后就无法在窗体的设计中拖动相应的层，这样能够遍历的对不需要经常更改的层进行方便的管理。

### 19.3 WPF 应用程序开发

**WPF** 不仅提供了强大的布局功能和窗体渲染功能，在 **WPF** 应用程序开发中，还能够实现如 **Flash** 一样的动画效果，这就使得在 **Windows** 窗体中能够实现 **Flash** 动画效果，**Microsoft Expression Blend 2** 提供了动画轴，动画事件处理面板，方便了开发人员在 **WPF** 中实现动画效果。

#### 19.3.1 WPF 动画事件

**WPF** 可以像 **Flash** 一样支持动画开发，与普通的事件不同的是，**WPF** 包括一个动画事件，这个动画事件描述的是当用户执行某个操作时所触发的动画事件。首先需要创建一个动画对象，这个对象可以是一个图片，也可以是一个控件，其 **XAML** 文档如下所示。

```
<Button HorizontalAlignment="Left" Margin="267,103,0,0" VerticalAlignment="Top"
Content="会变的按钮" Height="52.687" Width="86" Opacity="0.4"/>
```

上述代码创建了一个按钮控件，并为按钮控件配置了相应的属性，这些属性包括对齐方式，大小，文本，以及不透明度。如果开发人员希望当用户的鼠标单击该控件时，该控件的宽度和高度都会变化，并且不透明度也会变化，在 **Microsoft Expression Blend 2** 中的交互控制面板可以完成该事件的配置，如图 19-13 所示。

单击【+事件】按钮可以为 **WPF** 应用程序添加动画事件，**Microsoft Expression Blend 2** 能够智能的识别 **WPF** 应用中的控件并为相应的控件选择方法，为了实现开发人员所希望实现的效果。在下拉菜单中，这里可以选择【Click】事件如图 19-14 所示。



图 19-13 交互面板

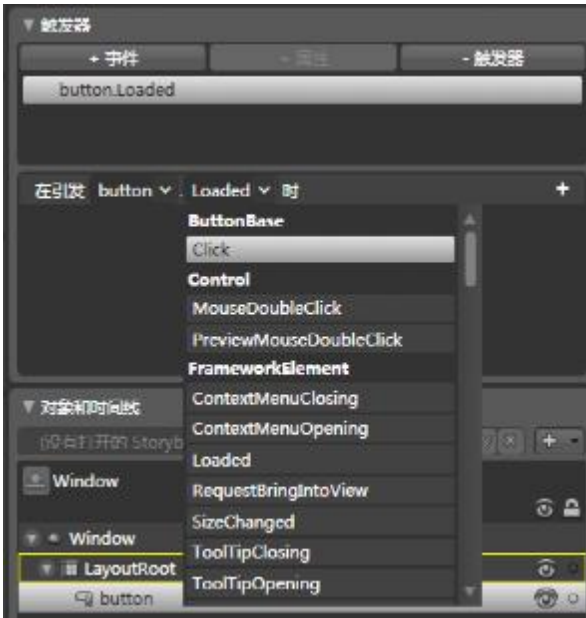


图 19-14 选择事件

选择事件后单击【时】按钮旁边的加号能够为动画事件添加新操作，如果 **WPF** 应用程序没有创建时间轴时，系统会提示是否添加一个时间轴，单击【确定】按钮即可创建一个默认的时间轴以供开发人员进行动画开发，如图 19-15 所示。



图 19-15 添加时间轴

**注意：**Story board 可以翻译成节目播出表，其概念同时间轴基本相同，都会规定对象的播放顺序和方法。

19.3.2 WPF 时间轴

在 **Flash** 动画的制作中，有一个时间轴的概念。时间轴是用来控制在动画运行中相应的时间时，某个或某些对象所需要执行的操作。例如一个 **Flash** 动画，在动画运行后，第 5 秒的时候有一个小人出现在动画中，并且拿出了一朵花。在这个过程中，就需要在第 5 秒的时候对相应的对象（这里包括人，花）进行相应的操作。

在 **WPF** 应用程序中同样包括一个时间轴，这个时间轴用于定制 **WPF** 动画事件中某一个时刻所需要实现的动画效果，开发人员能够通过使用 **WPF** 时间轴快速的实现动画效果。例如上一节中讲到的开发人员希望实现一个按钮的动画效果，则可以通过时间轴编写相应的事件，时间轴如图 19-16 所示。

在对象和时间轴面板中，可以选择相应的对象进行动画事件的操作。在时间轴面板中，可以看到在时间轴上方包括从 0 开始的数字，这些数字就是时间控制。当触发某个动画事件时，时间轴就开始运算，从 0 开始向右移动，分别执行相应的路径以实现动画效果。如果开发人员希望用户单击按钮时能够实现按钮形状和透明度的变化，可以通过时间轴方便的生成动画。在进行时间轴操作前，可以通过样式控制控件的相应位置，样式和内容等，如图 19-17 所示。

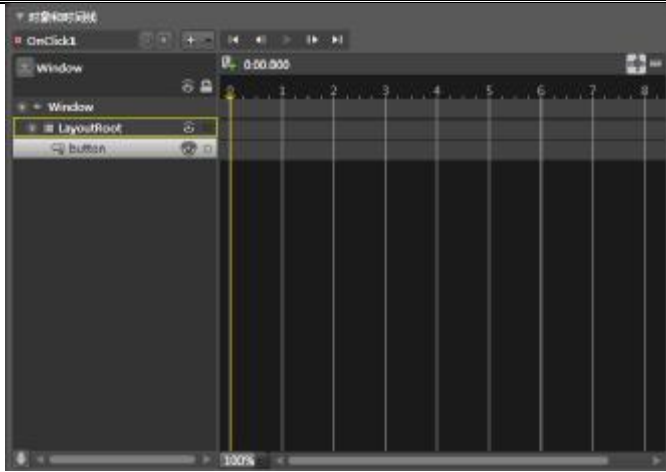


图 19-16 对象和时间轴



图 19-17 控件初始状态

初始状态确定后 XAML 文档代码如下所示。

```
<Button HorizontalAlignment="Left" Margin="267,103,0,0" VerticalAlignment="Top" Content="会 变 的 按 钮 "
Height="52.687" Width="86" Opacity="0.4" x:Name="button" RenderTransformOrigin="0.5,0.5">
  <Button.RenderTransform>
    <TransformGroup>
      <ScaleTransform ScaleX="1" ScaleY="1"/>
      <SkewTransform AngleX="0" AngleY="0"/>
      <RotateTransform Angle="0"/>
      <TranslateTransform X="0" Y="0"/>
    </TransformGroup>
  </Button.RenderTransform>
</Button>
```

上述代码初始化了一个按钮控件并声明。在确定了初始状态后就需要拖动时间轴来确定动画播放的顺序，如图 19-18 所示。

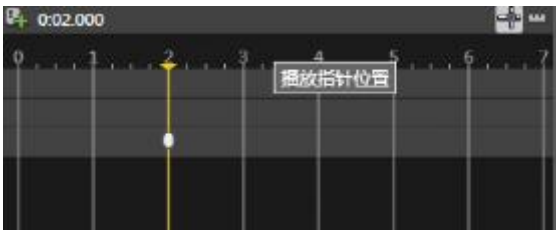


图 19-18 拖动时间轴

如图 19-18 所示，可以将时间轴拖放在 2 秒的位置。拖放后，可以直接在当前位置修改控件的属性。修改后当触发动画事件后，时间轴开始移动并且会随着时间轴进行控件属性的更改。当时间轴的时间指针移动到 2 秒位置时，属性就会更改成 2 秒时设置的样式，如图 19-19 和图 19-20 所示。



图 19-19 初始状态



图 19-20 实现动画

通过使用时间轴能够快速定义 WPF 动画效果，开发人员能够使用时间轴进行相应的动画操作而无需通过编程实现，这样就简化了开发人员对底层动画实现的复杂的操作，节约了开发周期。另外，设计人员也能够设计动画事件并专注与 WPF 动画的实现，而开发人员能够专注逻辑处理，可以将动画事件的实现交付给设计人员，形成明确的分工。

19.3.3 WPF 事件处理

在 **Microsoft Expression Blend 2** 中只能控制 **WPF** 应用程序的样式，却无法进行事件处理开发，若需要进行 **WPF** 应用程序开发，就必须使用 **Visual Studio 2008**。使用 **Visual Studio 2008** 打开 **Microsoft Expression Blend 2** 创建的解决方案，能够进行 **WPF** 应用程序事件开发，如图 19-21 所示。

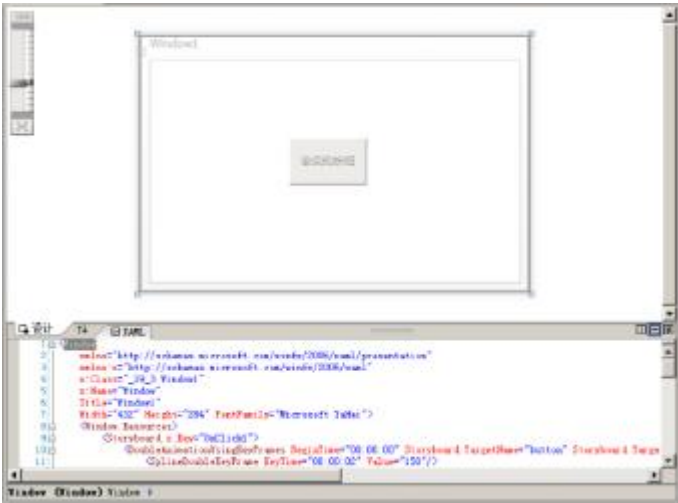


图 19-21 使用 **Visual Studio 2008** 打开解决方案

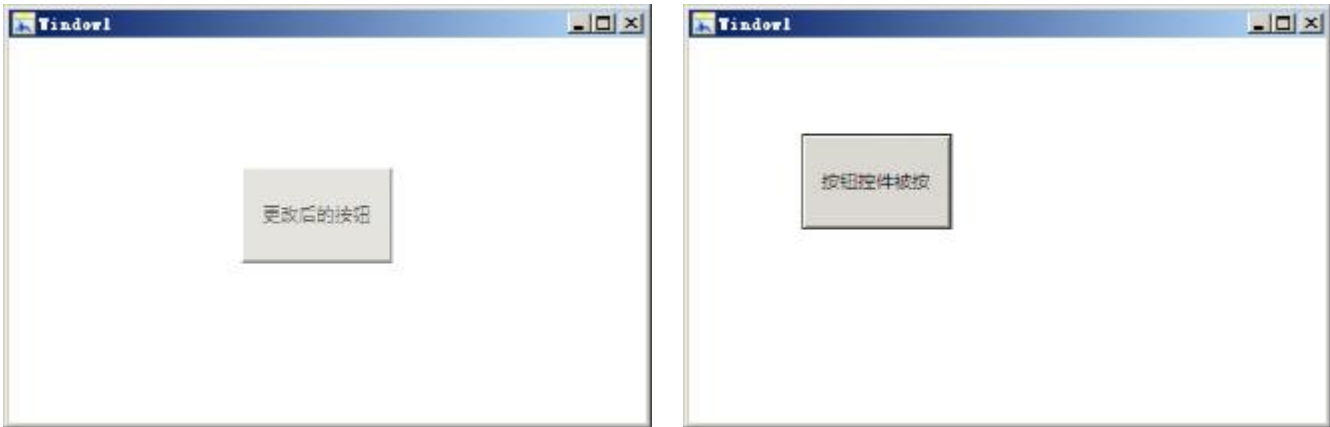
在 **Visual Studio 2008** 中进行 **WPF** 应用程序开发会呈现两个窗口，一个窗口用于直接进行 **Windows** 窗体布局，另一个窗口用于呈现相应的 **XAML** 文档。在 **Visual Studio 2008** 中，可以直接修改 **XAML** 文档进行 **WPF** 样式控制，示例代码如下所示。

```
<Grid x:Name="LayoutRoot">
  <Button Margin="155,86,169,107" Content="更改后的按钮" Opacity="0.6"
    x:Name="button" RenderTransformOrigin="0.5,0.5">
    <Button.RenderTransform>
      <TransformGroup>
        <ScaleTransform ScaleX="1" ScaleY="1"/>
        <SkewTransform AngleX="0" AngleY="0"/>
        <RotateTransform Angle="0"/>
        <TranslateTransform X="0" Y="0"/>
      </TransformGroup>
    </Button.RenderTransform>
  </Button>
</Grid>
```

上述代码直接修改 **XAML** 代码就可以实现 **Windows** 窗体样式的控制。与 **Microsoft Expression Blend 2** 不同的是，在 **Visual Studio 2008** 中双击按钮控件，在就会在 **cs** 文件中自动创建相应的事件代码，开发人员可以在相应的区域中编写代码，示例代码如下所示。

```
private void button_Click(object sender, RoutedEventArgs e)
{
    button.Content = "按钮控件被按"; //触发事件
}
```

上述代码运行后如图 19-22 和图 19-23 所示。





注意：WPF 应用程序中的一些属性可能和 Win From 和 ASP.NET 中的一些属性不同，例如在 Win From 和 ASP.NET 中按钮控件上的文本是通过 Text 属性控制的，而在 WPF 中使用的是 Content 属性。

WPF 应用的开发和 Win Form 应用程序的开发没有特殊的区别，但是 WPF 应用提供了更好的用户体验。WPF 不仅能够提供动画事件同样也能够执行 Win Form 应用程序开发中所需要的事件。

## 19.4 WPF 系统开发

WPF 能够开发用户体验更好的 Windows 应用程序，通过使用 WPF 技术，能够实现可扩展的容易维护并且用户体验友好的 Windows 应用程序。在微软本身的产品中，很多应用也使用了 WPF 技术，包括 Vista 以及 Expression。

### 19.4.1 WPF 系统需求

在 Windows 应用程序开发中，常常需要进行数据查询，例如一个图书管理系统，借读的读者往往很难在诸多图书当中寻找一个适合自己的书，例如如果读者希望借一本关于 ASP.NET 的书，但是图书馆中包含了很多关于 ASP.NET 的书，读者曾经看过了一本关于 ASP.NET 3.5 的书，希望能够找到该书，但是在图书馆中找了半天都找不到这本书，读者就会想“如果能够查询该书就好了”。

开发人员可以很快的进行图书管理系统的编码并进行查询分析，现在读者可以在图书馆电脑中查询 ASP.NET 3.5 开发大全了，但是查询出来的结果显示的并不那么友好，而且界面颜色单调，这就需要 Windows 应用程序具有较好的用户体验。WPF 应用程序就能够实现较好的用户体验，同样也可以实现普通 Windows 应用程序所能够完成的需求。

### 19.4.2 WPF 界面开发

为了实现较好的用户体验，首先需要进行良好的 WPF 界面开发和布局。WPF 支持 PNG, JPG 等图片资源作为 WPF 应用程序的背景，所以 WPF 应用程序能够实现半透明等多种渲染效果，WPF 系统登录界面和查询界面如图 9-24 和图 9-25 所示。



图 9-24 图书系统初步布局



图 9-25 用户查询界面布局

**WPF** 能够支持 **PNG**，**JPG** 等格式的图片文件，所以在 **WPF** 窗体开发中能够使用渐变效果填充窗体并可以直接使用 **PNG** 图片进行窗体渲染。登录窗体包含了一个图片文件，图片文件的 **XAML** 代码如下所示。

```
<Image Margin="0,0,2,23" Width="490" Height="450" Source="bg.png" Stretch="Fill"/>
```

注意：**PNG** 图片支持透明效果，而其他的图片格式的文件可能不支持半透明效果，**WPF** 支持半透明图片作为资源文件。

通过 **XAML** 文档能够定义图片文件并定义一些常用控件，为了实现以上的 **WPF** 界面布局，**WPF** 应用程序窗体的 **XAML** 代码如下所示。

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  x:Class="_19_4.Window1"
  x:Name="Window"
  Title="图书管理系统"
  Width="500"
  Height="500"
  FontFamily="Microsoft YaHei"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006" mc:Ignorable="d">
  .....
</Window>
```

上述代码通过 **XAML** 实现了一个基本的 **Windows** 窗体，该窗体的头部信息为图书管理系统，并且定义了窗体的高度和宽度为 **500**，窗体内的字体样式为微软雅黑。在 **WPF** 窗体中，还需要定义 **Label** 控件和 **TextBox** 等控件，用于实现基本的人机交互，其 **XAML** 代码如下所示。

```
<Grid x:Name="LayoutRoot">
  <Image Margin="0,0,2,23" Width="490" Height="450" Source="bg.png" Stretch="Fill"/>
  <Label FontSize="16" FontWeight="Bold" Margin="132,37,116.11,0"
    VerticalAlignment="Top" Height="30.117" Content="读者您好,欢迎查阅我图书馆资料"/>
  <Image HorizontalAlignment="Left" Margin="58,25,0,0" VerticalAlignment="Top"
    Width="54" Height="54" Source="hello.png" Stretch="Fill"/>
  <TabControl IsSynchronizedWithCurrentItem="True" Margin="12,95,21,23">
    <TabItem Header="登录">
      <Grid/>
    </TabItem>
  </TabControl>
</Grid>
```

```
<TabItem Header="查询">
    <Grid/>
</TabItem>
</TabControl>
<Label HorizontalAlignment="Left" Margin="110,227,0,0" VerticalAlignment="Top" Content="用户名:"/>
<Label HorizontalAlignment="Left" Margin="122,0,0,173" VerticalAlignment="Bottom" Content="密码:"/>
<TextBox d:LayoutOverrides="HorizontalAlignment" HorizontalAlignment="Left" Margin="175,0,0,213.163"
    VerticalAlignment="Bottom" Text="TextBox" TextWrapping="Wrap" Width="192.89"/>
<PasswordBox Margin="175,0,116.11,179.037" VerticalAlignment="Bottom"/>
<Button HorizontalAlignment="Left" Margin="198,0,0,100" VerticalAlignment="Bottom" Content="登录"
    Height="44.837" Width="75.887"/>
</Grid>
```

上述代码实现了 **WPF** 窗体的基本布局，在 **WPF** 窗体中包含三个 **Label** 标签控件，用于显示相应的提示信息，如“用户名”，“密码”等。该窗体还包含了两个 **TextBox** 控件，其中一个 **TextBox** 控件用于用户用户名的输入，而另一个 **TextBox** 控件用于密码的输入。编辑完成登录窗体后，就需要进一步对搜索窗体进行样式控制，搜索窗体 **XAML** 文档代码如下所示。

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="_19_4.Window2"
    x:Name="Window"
    Title="Window2"
    Width="500" Height="500" FontFamily="Microsoft YaHei">
    .....
</Window>
```

上述代码同样创建了一个宽度和高度都为 **500** 的窗体，在窗体中包括一个图片，一个搜索框和一个搜索结果框，这组控件 **XAML** 代码如下所示。

```
<Grid x:Name="LayoutRoot">
    <Image Margin="0,0,2,23" Source="bg.png" Stretch="Fill" Width="495" Height="450"/>
    <Image Margin="80,29,0,0" Source="ok.png" Stretch="Fill" Width="91"
        Height="91" VerticalAlignment="Top" HorizontalAlignment="Left"/>
    <Label HorizontalAlignment="Left" Margin="194,54,0,0" VerticalAlignment="Top"
        Content="输出相应书籍名称" FontWeight="Bold"/>
    <TextBox HorizontalAlignment="Left" Margin="194,83.837,0,0" VerticalAlignment="Top"
        Text="" TextWrapping="Wrap" Width="246.487"/>
    <TextBox Margin="44,137,31.513,36" Text="" TextWrapping="Wrap"/>
</Grid>
```

窗体基本布局完成后，就可以为窗体中的控件进行动画事件的编写，创建动画事件能够提高用户的体验并且使应用程序更加绚丽。

### 19.4.3 WPF 动画制作

在图书管理系统中，希望读者首先登录，如果登录成功了，就能够进行查询；如果登录没有成功，则不允许用户开始查询，只有用户登录成功后才有查询权限。在读者单击登录按钮时，应用程序可以播放一段动画以提示用户正在登录，如图 9-26 和图 9-27 所示。



图 9-26 登录框位置下移



图 9-27 登录框位置上移

当用户单击登录按钮进行登录时，登录框会上下移动以提示用户该应用程序正在处理。在动画处理代码中，必须为其中的每一个控件进行动画处理描述，而写控件的动画处理的 **XAML** 文档基本相同，示例代码如下所示。

```
<Window.Resources>                                     //窗体资源文件
    <Storyboard x:Key="OnClick1">                         //定义了动画事件
        <DoubleAnimationUsingKeyFrames
            BeginTime="00:00:00"
            Storyboard.TargetName="label" Storyboard.TargetProperty=
            "(UIElement.RenderTransform).(TransformGroup.Children)[3].(TranslateTransform.Y)">
            <SplineDoubleKeyFrame KeyTime="00:00:01" Value="-85"/>
            <SplineDoubleKeyFrame KeyTime="00:00:02" Value="49"/>
            <SplineDoubleKeyFrame KeyTime="00:00:03" Value="-86"/>
        </DoubleAnimationUsingKeyFrames>
    </Storyboard>
```

上述代码定义了动画处理中变换的操作，在 **XAML** 文档中，动画处理都会被作为窗体资源而存在，而动画事件作为窗体触发器而存在，示例代码如下所示。

```
<Window.Triggers>                                       //窗体触发器
    <EventTrigger RoutedEvent="ButtonBase.Click" SourceName="button">
        <BeginStoryboard Storyboard="{StaticResource OnClick1}"/>
    </EventTrigger>
</Window.Triggers>
```

上述代码定义了窗体触发器，当用户操作 **OnClick1** 事件后则会触发动画处理事件。开发人员能够在 **<Storyboard x:Key="OnClick1">** 标记中定义控件动画事件的其他内容以扩展 **WPF** 动画事件。

19.4.4 WPF 事件编写

在 **WPF** 应用程序控件动画制作中，不能为了实现绚丽的动画而放弃了实用的功能。该应用程序希望用户能够进行登录并对用户的身份进行验证操作，如果验证成功则能够执行操作，而如果身份验证不成功，则无法执行搜索操作。在 **Visual Studio 2008** 中，双击按钮控件以进行登录验证操作，示例代码如下所示。

```
private void button_Click_1(object sender, RoutedEventArgs e)
{
    if ((textBox.Text == "admin") && (passwordBox.Password == "admin")) //如果是管理员
    {
        Window2 w2 = new Window2(); //打开新窗口
        w2.ShowDialog();
    }
}
```



```
    }
}
```

上述代码定义了用户如果输入了用户名和密码分别为 **admin/admin** 时，则验证成功，就会呈现搜索框，如果用户名和密码不正确，则无法验证进行搜索。进入搜索窗口时，用户可以在书籍搜索框中输入相应信息，当用户输入信息后，结果框就能够及时反映相应的搜索结果，示例代码如下所示。

```
public string[] books = { "ASP.NET 开发大全", "ASP 开发指南", ".NET 应用程序", "组件开发指南",
                          "PHP 新手入门", "C++学习" };

private void TextBox_TextChanged(object sender, TextChangedEventArgs e)           //用户查询
{
    if (!String.IsNullOrEmpty(search.Text))                                     //如果输入不为空
    {
        result.Clear();                                                         //清空结果
        for (int i = 0; i < books.Length; i++)                                 //遍历书籍
        {
            if (books[i].Contains(search.Text))                               //如果匹配则输出
            {
                result.Text += books[i].ToString() + "\n";                     //填充结果控件
            }
        }
    }
}
```

上述代码定义了一个数组以存储书籍的相应信息，当用户在搜索框中输入相应的信息时，系统就会遍历数据库进行书籍查询，运行结果如图 9-28 和图 9-29 所示。



图 9-28 搜索 C++相关书籍



图 9-29 清空结果后再搜索

## 19.5 小结

本章简单的讲解了 **WPF** 的基础知识，包括 **WPF** 和 **WPF** 的适用范围，**WPF** 是微软近几年力推的技术，随着 **Vista** 的普及，**WPF** 应用已经被越来越多的个人和企业接受，了解 **WPF** 技术在今后的项目开发中会起到很好的作用。本章还包括：

- ❑ 什么是 **WPF**：讲解了什么是 **WPF**，以及 **WPF** 引擎和 **WPF** 构架。
- ❑ 使用 **Microsoft Expression Blend** 设计 **WPF**：讲解了如何使用 **Microsoft Expression Blend**，以及如何使用 **Microsoft Expression Blend** 设计 **WPF** 应用程序。
- ❑ **XAML** 文档：讲解了 **XAML** 基本概念，以及如何通过 **XAML** 进行样式控制。
- ❑ **WPF** 控件层次：讲解了 **WPF** 中控件的层次概念。
- ❑ **WPF** 动画事件：讲解了 **WPF** 动画事件的概念，以及如何使用 **Microsoft Expression Blend** 开发动画事件。
- ❑ **WPF** 时间轴：讲解了时间轴的概念，以及使用时间轴进行动画开发。
- ❑ **WPF** 事件处理：讲解了如何使用 **WPF** 进行事件处理。

由于本书并不详细的讲解 **WPF** 应用开发，本书只是对 **WPF** 进行了简单的介绍。**WPF** 应用程序的开发和 **Win From** 开发基本相同，但是 **WPF** 提供了更好的开发和布局方案，使用 **WPF** 能够开发用户体验更好的应用程序。

## 第六篇 ASP.NET 3.5 与 LINQ

第 20 章 ASP.NET 3.5 与 LINQ

第 21 章 使用 LINQ 查询

## 第 20 章 ASP.NET 3.5 与 LINQ

对于长期发展的面向对象编程模型而言，其发展基本处于一个比较稳定的阶段，可是面向对象的编程模型并没有解决数据的访问和整合的复杂问题。对于数据库的访问和 XML 的访问，面向对象方法论无法从根本意义上解决其复杂度和难度，而 LINQ 提供了一种更好的解决方案。

### 20.1 什么是 LINQ

任何技术都不可能凭空搭建起来，为了解决工业生产中某个实际问题，当现有的技术已经无法很好的完成工业的要求，就会促发新技术的诞生。LINQ 就是为了解决复杂的数据访问和整合而出现的一种新技术。

#### 20.1.1 LINQ 起源

从传统的意义上来说，面向过程的编程模型在数据访问和整合的能力上有一定的限度。因为面向过程的编程方法不能很好的描述一个事务，必须通过不同函数之间的调用来描述一个现有的对象，而且面向过程的编程方法在代码复用性上比较低，所以当面向过程的编程语言需要对数据库进行访问时，就需要编写大量的额外代码。虽然面向过程的编程模型可以通过良好的函数引用和编码提高复用性，但是并没有解决面向过程编程模型中对数据的访问和整合的复杂度。

随着计算机和编程模型的发展，人们发现了另一个更好的编程模型，这就是现在最常用的面向对象编程模型。相比面向过程的编程模型而言，面向对象的编程模型能够更好的描述一个事务，事务能够通过面向对象中的属性、字段和方法很好的模拟实际的事务，而面向对象编程模型中的派生、继承等特性同样能够极大的提高代码的复用性，提升开发效率。

但是面向对象的编程模型同样没有解决复杂的数据库访问和数据整合，开发人员还是需要通过繁琐的手段进行数据库的访问和数据整合。在 .NET 3.0 框架或更早，LINQ 就已经被提及，LINQ 是一种能够快速对大部分数据源进行访问和数据整合的一种技术，LINQ 解决了复杂的数据应用中开发人员需要面对和解决的问题。

虽然面向对象的数据库已经在几年前就被提及并且各大 IT 公司投入了对面向对象的数据库的研发，但是传统的关系型数据库在当今还是应用最为广泛的。关系型数据库中将数据整合和呈现成为一张张的表的形式，开发人员和数据库管理人员能够通过 SQL 管理工具提供的 SQL 语句进行数据的查询和整理。但是在开发过程中，开发人员不能够像使用 SQL 语句一样对数据集进行查询和处理。任何数据库中的数据都会以一种数据集的形式反馈给用户，这种数据集的形式可以反映成为数学中的集合的概念，其实在数据库早期的发展中，数据是以集合的概念呈现的，而随着数据库的发展，集合的概念依旧是数据库最基本的概念。

正式因为如此，开发人员不能够方便的是从一个集合中查询数据，这里不仅仅是一个数据库，还包括其他能够以数据库形式存在的文件，例如 ACCESS、TXT 等，当在开发中需要使用到多个数据库或者数据描述形式的文件时，更多的情况是将这些数据填充到数据集中并通过遍历来访问数据，这样却造成了更多的数据访问问题和麻烦。

LINQ 能够很方便的进行数据的查询，使用 LINQ 对数据集进行查询的形式很像使用 SQL 语句对数据库中的表进行查询，而与之不同的是，LINQ 能够面向更多的对象，这些对象包括数组、集合以及数据库，LINQ



对数组的查询示例代码如下所示。

```
static void Main(string[] args)
{
    string[] str = { "你好","今天的","天气真不错","生活很阳光"};           //创建数组
    var s = from n in str select n;                                           //编写查询字符串
    foreach (var n in s)                                                       //遍历查询对象
    {
        Console.WriteLine(n.ToString());                                     //输出对象值
    }
    Console.ReadKey();                                                         //等待用户按键
}
```

上述代码对数组 **str** 进行了查询，这种方式很像 **SQL** 语句。的确 **LINQ** 的查询方式和 **SQL** 语句很像，其语法和基本内容都没有什么太大的差别，但是 **LINQ** 提供了更好的查询的解决方案。**LINQ** 能够查询更多对象（例如上述代码中的数组）而无法使用 **SQL** 语句进行查询。另外，**LINQ** 查询语句还能够使用 **WHERE** 等关键字进行查询，示例代码如下所示。

```
static void Main(string[] args)
{
    string[] str = { "你好","今天的","天气真不错","生活很阳光"};           //创建数组
    var s = from n in str where n.Length > 3 select n;                       //使用条件查询
    foreach (var n in s)                                                       //遍历查询对象
    {
        Console.WriteLine(n.ToString());                                     //输出对象值
    }
    Console.ReadKey();                                                         //等待用户按键
}
```

上述代码修改了 **LINQ** 查询语句，为 **LINQ** 查询语句增加了条件查询，该条件的意义为查询字符串长度大于 **3** 的字符串，运行后如图 **20-1** 所示。

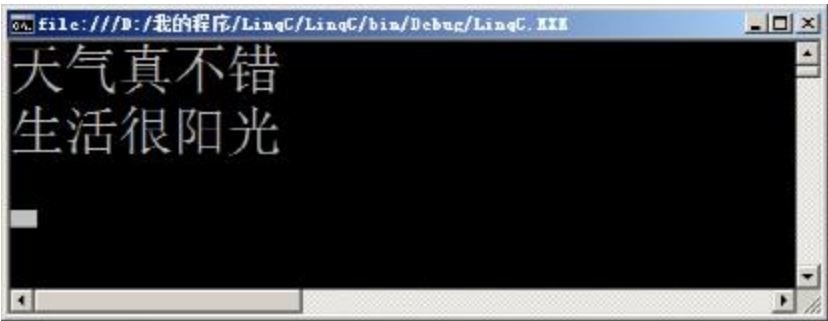


图 20-1 LINQ 查询语句

从上图可以看出，能够使用类似于 **SQL** 语句的形式进行数据集的查询，很大程度上方便了开发人员对于数据库中数据的访问和整理。**LINQ** 可以使用条件语句进行筛选，并且能够使用 **.NET** 提供的语法进行判断，这样就简化了开发人员对于数据集中的数据的筛选。有关 **LINQ** 的语句，会在后面的章节中详细的讲解。

20.1.2 LINQ 构架

在 **.NET 3.5** 中，**LINQ**（**Language Integrated Query**）已经成为了编程语言的一部分，开发人员已经能够使用 **Visual Studio 2008** 创建使用 **LINQ** 的应用程序。**LINQ** 对基于 **.NET** 平台的编程语言提供了标准的查询操作。在 **.NET 3.5** 中，**LINQ** 的基本构架如图 **20-2** 所示。

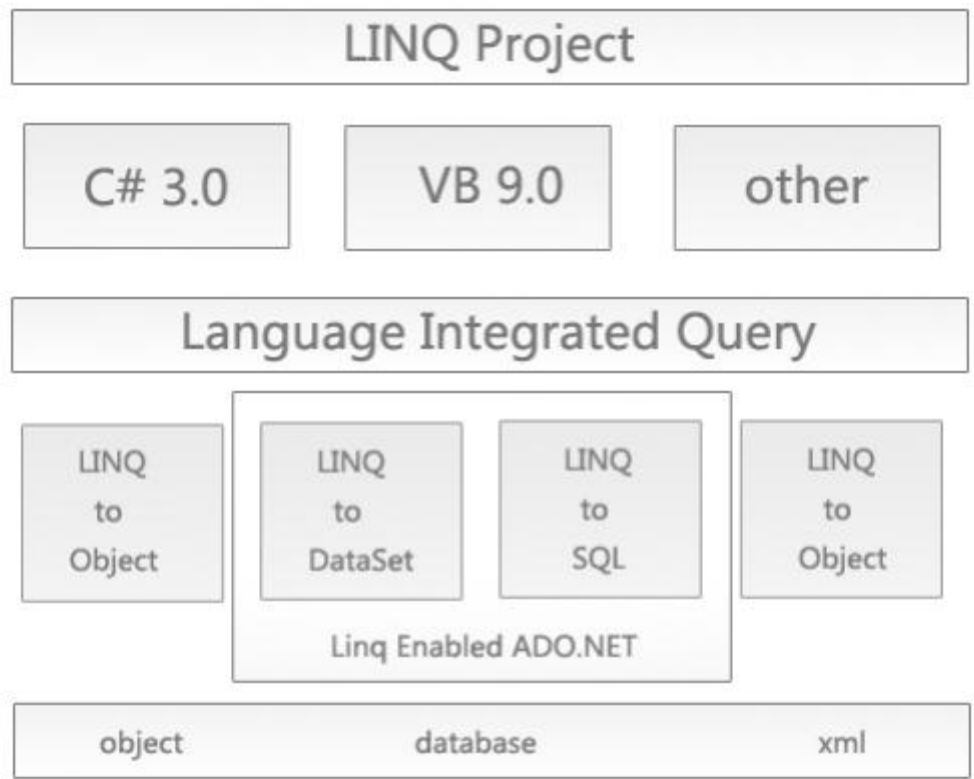


图 20-2 LINQ 基本构架

如图 20-2 所示，LINQ 能够对不同的对象进行查询。在.NET 3.5 中，微软提供了不同的命名空间以支持不同的数据库配合 LINQ 进行数据查询。在 LINQ 框架中，处于最上方的就是 LINQ 应用程序，LINQ 应用程序基于.NET 框架而存在的，LINQ 能够支持 C#、VB 等.NET 平台下的宿主语言进行 LINQ 查询。在 LINQ 框架中，还包括 Linq Enabled ADO.NET 层，该层提供了 LINQ 查询操作并能够提供数据访问和整合功能。

LINQ 包括五个部分，这五个部分分别是 LINQ to Objects、LINQ to DataSet、LINQ to SQL、LINQ to Entities、LINQ to XML，在.NET 开发中最常用的是 LINQ to SQL 和 LINQ to XML，本书也详细介绍 LINQ 的这两个部分。

LINQ to SQL 提供了对 SQL Server 中数据库的访问和整合功能，同时能够以对象的形式进行数据库管理，前面已经提到，现在的数据库依旧以关系型数据库为主，在面向对象开发过程中，很难通过对象的方法描述数据库，而 LINQ 提供了通过对象的形式对数据库进行描述。LINQ to XML 提供了对 XML 中数据集的访问和整合功能，LINQ to XML 使用 System.Xml.Linq 命名控件，为 XML 操作提供了高效易用的方法。

## 20.1.3 LINQ 与 Visual Studio 2008 新特性

讲到 LINQ 就不得不讲解 Visual Studio 2008 的新特性，LINQ 作为 Visual Studio 2008 中的一部分，Visual Studio 2008 为 LINQ 提供很好的编程环境，LINQ 也使用到了 C# 编程语言中的很多特性，以提高开发人员的开发效率。

- ❑ **Visual Studio 2008 重定向：**使用 Visual Studio 2008 与 Visual Studio 2005 不同的是，Visual Studio 2008 支持多个版本.NET 框架的共存，在 Visual Studio 2008 中可以选择基于.NET 2.0 或.NET 3.X 版本的框架来开发不同的应用程序，当选择不同的应用程序基础框架时，Visual Studio 2008 能够智能的提供不同的命名空间。
- ❑ **Visual Studio 2008 AJAX：**在 ASP.NET 2.0 开发中，需要使用 ASP.NET AJAX 1.0 作为 AJAX 开发必备的工具，在 Visual Studio 2008 中已经集成了对 AJAX 的支持，创建 ASP.NET 3.5 应用程序已经能够非常方便的使用 AJAX 功能。
- ❑ **Visual Studio 2008 可视化操作：**在 Visual Studio 2008 中，微软提供了可视化操作，开发人员能够选择不同的视图进行页面分离形式的开发，在 Visual Studio 2008 中开发人员可以选择视图，拆分，代码三种视图进行不同的开发体验。
- ❑ **Visual Studio 2008 集成 LINQ：**这是 Visual Studio 2008 中比较值得期待的功能，Visual Studio 2008 将 LINQ 作为编程语言中的一部分，为开发人员提供了 LINQ 开发的原生环境。

在 LINQ 与 Visual Studio 2008 中，开发人员最为期待的新特性还是 Visual Studio 2008 对 LINQ 的原生支持，使用 LINQ 能够快速地进行数据库的访问和整合，这样在一定的意义上降低了开发难度，LINQ 在 .NET Framework 3.5 中的位置如图 20-3 所示。

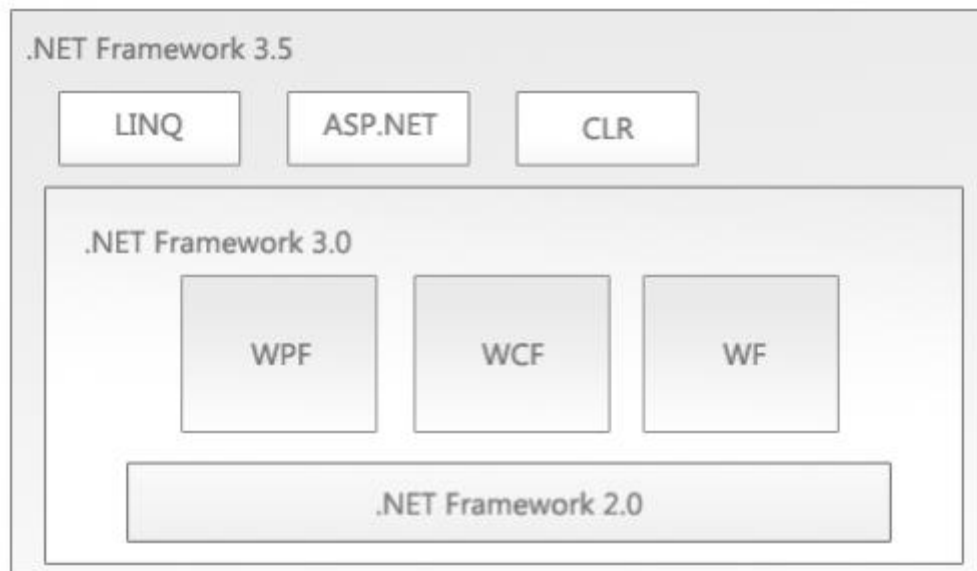


图 20-3 .NET 框架中的 LINQ

正如图 20-3 所示，.NET 2.0 后面几个版本的框架都是基于 .NET Framework 2.0 而存在的，在 .NET Framework 3.0 中，微软已经增加了 WPF，WCF，WF 等新特性以提供快速的面向服务的开发和完善的用户体验解决方案。而 LINQ 是作为 .NET Framework 3.5 存在于 .NET Framework 中的，这也就是说只有在 .NET Framework 3.5 框架中才能够使用 LINQ 技术。由于 .NET Framework 3.5 版本的框架基于 .NET Framework 3.0 版本，开发人员可以使用 LINQ 特性进行分布式开发和面向服务的开发，这样就能够更进一步的提高代码的复用性和安全性。

## 20.2 LINQ 与 Web 应用程序

在 ASP.NET 应用程序开发中，常常需要涉及到数据的显式和整合，使用 ASP.NET 2.0 中提供的控件能够编写用户控件，开发人员还能够选择开发自定义控件进行数据显示和整合，但是在数据显示和整合过程中，开发人员往往需要大量的连接、关闭连接等操作，而且传统的方法也破坏了面向对象的特性，使用 LINQ 能够方便的使用面向对象的方法进行数据库操作。

### 20.2.1 创建使用 LINQ 的 Web 应用程序

创建 LINQ 的 Web 应用程序非常的容易，只要创建 Web 应用程序时选择的平台是基于 .NET Framework 3.5 的就能够创建使用 LINQ 的 Web 应用程序，如图 20-4 所示。



图 20-4 选择.NET Framework 3.5

当创建一个基于系统.NET Framework 3.5 的应用程序，系统就能够自动为应用程序创建 LINQ 所需要的命名空间，示例代码如下所示。

```
using System.Xml.Linq; //使用 LINQ 命名空间
using System.Linq; //使用 LINQ 命名空间
```

上述命名空间提供了应用程序中使用 LINQ 所需要的基础类和枚举，在 ASP.NET 应用程序中就能够使用 LINQ 查询语句进行查询，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    string[] str = { "我爱 C#", "我喜欢 C#", "我做 C#开发", "基于.NET 平台", "LINQ 应用" }; //数据集
    var s = from n in str where n.Contains("C#") select n; //执行 LINQ 查询
    foreach (var t in s) //遍历对象
    {
        Response.Write(t.ToString() + "<br/>"); //输出查询结果
    }
}
```

上述代码在 ASP.NET 页面中执行了一段 LINQ 查询，查询字符串中包含“C#”的字符串，运行后如图 20-5 所示。



图 20-5 ASP.NET 执行 LINQ 查询

在 ASP.NET 中能够使用 LINQ 进行数据集的查询，Visual Studio 2008 已经将 LINQ 整合成为编程语言中的一部分，基于.NET Framework 3.5 的应用程序都可以使用 LINQ 特性进行数据访问和整合。



20.2.2 基本的 LINQ 数据查询

使用 LINQ 能够对数据集进行查询，在 ASP.NET 中，可以创建一个新的 LINQ 数据库进行数据集查询，右击现有项目，单击【添加新项】选项，选择【LINQ to SQL 类】选项，如图 20-6 所示。

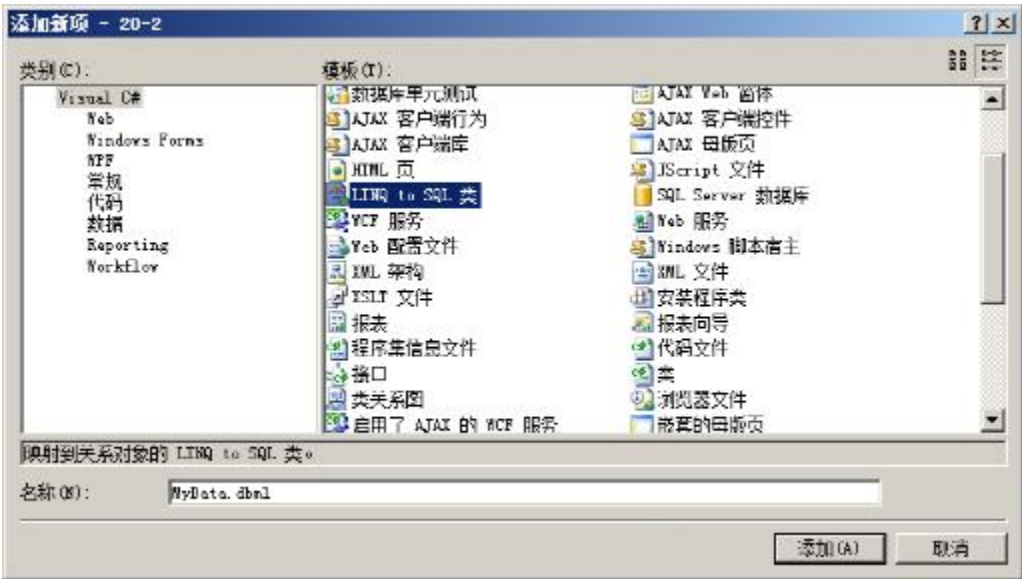


图 20-6 创建 LINQ to SQL 类

创建一个 LINQ to SQL 类，能够映射一个数据库，实现数据对象的创建，如图 20-7 所示。创建一个 LINQ to SQL 类后，可以直接在服务资源管理器中拖动相应的表到 LINQ to SQL 类文件中，如图 20-8 所示。



图 20-7 服务资源管理器



图 20-8 拖动一个表

开发人员能够直接将服务资源管理器中的表拖动到 LINQ to SQL 类中，在 LINQ to SQL 类文件中就会呈现一个表的视图。在视图中，开发人员能够在视图添加属性和关联，并且能够在 LINQ to SQL 类文件中可以设置多个表，进行可视化关联操作。

创建一个 LINQ to SQL 类文件后，LINQ to SQL 类就将数据进行对象化，这里的对象化就是以面向对象的思想针对一个数据集建立一个相应的类，开发人员能够使用 LINQ to SQL 创建的类进行数据库查询和整合操作，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    MyDataDataContext data = new MyDataDataContext();           //使用 LINQ 类
    var s = from n in data.mynews where n.ID==1 select n;        //执行查询
    foreach (var t in s)                                         //遍历对象
    {
        Response.Write(t.TITLE.ToString() + "<br/>");           //输出对象
    }
}
```

上述创建了一个 MyData.dbml 的 LINQ to SQL 文件，开发人员能够直接使用该类的对象提供数据操作。

上述代码使用了 LINQ to SQL 文件提供的类进行数据查询，LINQ 查询语句示例代码如下所示。

```
var s = from n in data.mynews where n.ID==1 select n; //编写查询语句
```

上述代码使用了 LINQ 查询语句查询了一个 mynews 表中 ID 为 1 的行，使用 LINQ to SQL 文件提供的对象能够快速的进行数据集中对象的操作。创建一个 MyData.dbml 的 LINQ to SQL 文件，其中 MyDataDataContext 为类的名称，该类提供 LINQ to SQL 操作方法，示例代码如下所示。

```
MyDataDataContext data = new MyDataDataContext(); //使用 LINQ 类
```

上述代码使用了 LINQ to SQL 文件提供的类创建了一个对象 data，data 对象包含数据中表的集合，通过 “.” 操作符可以选择相应的表，示例代码如下所示。

```
data.mynews //选择相应表
```

使用 LINQ 查询后运行结果如图 20-9 所示。



图 20-9 LINQ 执行数据库查询

使用 LINQ 技术能够方便的进行数据库查询和整合操作，LINQ 不仅能够实现类似 SQL 语句的查询操作，还能够支持 .NET 编程方法进行数据查询条件语句的编写。使用 LINQ 技术进行数据查询的顺序如下所示：

- ❑ 创建 LINQ to SQL 文件：创建一个 LINQ to SQL 类文件进行数据集封装。
- ❑ 拖动数据表：将数据表拖动到 LINQ to SQL 类文件中，可以进行数据表的可视化操作。
- ❑ 使用 LINQ to SQL 类文件：使用 LINQ to SQL 类文件提供的数据集的封装进行数据操作。

使用 LINQ to SQL 类文件能够极快的创建一个 LINQ 到 SQL 数据库的映射并进行数据集对象的封装，开发人员能够使用面向对象的方法进行数据集操作并提供快速开发的解决方案。

20.2.3 IEnumerable 和 IEnumerable<T>接口

IEnumerable 和 IEnumerable<T>接口在 .NET 中是非常重要的接口,它允许开发人员定义 foreach 语句功能的实现并支持非泛型方法的简单的迭代，IEnumerable 和 IEnumerable<T>接口是 .NET Framework 中最基本的集合访问器，这两个接口对于 LINQ 的理解是非常重要的。

在面向对象的开发过程中，常常需要创建若干对象，并进行对象的操作和查询，在创建对象前，首先需要声明一个类为对象提供描述，示例代码如下所示。

```
using System;
using System.Collections.Generic;
using System.Linq; //使用 LINQ 命名控件
using System.Text;
namespace IEnumeratorSample
{
    class Person //定义一个 Person 类
    {
        public string Name; //定义 Person 的名字
        public string Age; //定义 Person 的年龄
        public Person(string name, string age) //为 Person 初始化（构造函数）
        {
```

```
Name = name;                                //配置 Name 值
Age = age;                                  //配置 Age 值
    }
}
```

上述代码定义了一个 **Person** 类并抽象一个 **Person** 类的属性，这些属性包括 **Name** 和 **Age**。**Name** 和 **Age** 属性分别用于描述 **Person** 的名字和年龄，用于数据初始化。初始化之后的数据就需要创建一系列 **Person** 对象，通过这些对象的相应属性能够进行对象的访问和遍历，示例代码如下所示。

```
class Program
{
    static void Main(string[] args)
    {
        Person[] per = new Person[2]           //创建并初始化 2 个 Person 对象
        {
            new Person("guojing","21"),        //通过构造函数构造对象
            new Person("muqing","21"),          //通过构造函数构造对象
        };
        foreach (Person p in per)              //遍历对象
            Console.WriteLine("Name is " + p.Name + " and Age is " + p.Age);
        Console.ReadKey();
    }
}
```

上述代码创建并初始化了 **2** 个 **Person** 对象，并通过 **foreach** 语法进行对象的遍历。但是上述代码是在数组中进行查询的，就是说如果要创建多个对象，则必须创建一个对象的数组，如上述代码中的 **Per** 变量，而如果需要直接对对象的集合进行查询，却不能够实现查询功能。例如增加一个构造函数，该构造函数用户构造一组 **Person** 对象，示例代码如下所示。

```
private Person[] per;
public Person(Person[] array)                //重载构造函数,迭代对象
{
    per = new Person[array.Length];          //创建对象
    for (int i = 0; i < array.Length; i++)    //遍历初始化对象
    {
        per[i] = array[i];                  //数组赋值
    }
}
```

上述构造函数动态的构造了一组 **People** 类的对象，那么应该也能够使用 **foreach** 语句进行遍历，示例代码如下所示。

```
Person personlist = new Person(per);         //创建对象
foreach (Person p in personlist)             //遍历对象
{
    Console.WriteLine("Name is " + p.Name + " and Age is " + p.Age);
}
```

在上述代码的 **foreach** 语句中，直接在 **Person** 类的集合中进行查询，系统则会报错“**ConsoleApplication1.Person**”不包含“**GetEnumerator**”的公共定义，因此 **foreach** 语句不能作用于“**ConsoleApplication1.Person**”类型的变量，因为 **Person** 类并不支持 **foreach** 语句进行遍历。为了让相应的类能够支持 **foreach** 语句执行遍历操作，则需要实现派生自类 **IEnumerable** 并实现 **IEnumerator** 接口，示例代码如下所示。

```
public IEnumerator GetEnumerator()            //实现接口
{
    return new GetEnumerator(_people);
}
```

为了让自定义类型能够支持 **foreach** 语句，则必须对 **Person** 类的构造函数进行编写并实现接口，示例代码如下所示。

```
class Person:IEnumerable                     //派生自 IEnumerable,同样定义一个 PersonI 类
```

```

{
    public string Name;                //创建字段
    public string Age;                //创建字段
    public Person(string name, string age) //字段初始化
    {
        Name = name;                //配置 Name 值
        Age = age;                //配置 Age 值
    }
    public IEnumerator GetEnumerator() //实现接口
    {
        return new PersonEnum(per); //返回方法
    }
}

```

上述代码重构了 **Person** 类并实现了接口，接口实现的具体方法如下所示。

```

class PersonEnum : IEnumerator //实现 foreach 语句内部,并派生
{
    public Person[] _per; //实现数组
    int position = -1; //设置 “指针”
    public PersonEnum(Person[] list)
    {
        _per = list; //实现 list
    }
    public bool MoveNext() //实现向前移动
    {
        position++; //位置增加
        return (position < _per.Length); //返回布尔值
    }
    public void Reset() //位置重置
    {
        position = -1; //重置指针为-1
    }
    public object Current //实现接口方法
    {
        get
        {
            try
            {
                return _per[position]; //返回对象
            }
            catch (IndexOutOfRangeException) //捕获异常
            {
                throw new InvalidOperationException(); //抛出异常信息
            }
        }
    }
}

```

上述代码实现了 **foreach** 语句的功能，当开发 **Person** 类初始化后就可以直接使用 **Person** 类对象的集合进行 **LINQ** 查询，示例代码如下所示。

```

static void Main(string[] args)
{
    Person[] per = new Person[2] //同样初始化并定义 2 个 Person 对象
    {
        new Person("guojing","21"), //构造创建新的对象
        new Person("muqing","21"), //构造创建新的对象
    };
    Person personlist = new Person(per); //初始化对象集合
    foreach (Person p in personlist) //使用 foreach 语句
        Console.WriteLine("Name is " + p.Name + " and Age is " + p.Age);
}

```



```
        Console.ReadKey();
    }
}
```

从上述代码中可以看出，初始化 **Person** 对象时初始化的是一个对象的集合，在该对象的集合中可以通过 **LINQ** 直接进行对象的操作，这样做即封装了 **Person** 对象也能够让编码更加易读。在.NET Framework 3.5 中，**LINQ** 支持数组的查询，开发人员不必自己手动创建 **IEnumerable** 和 **IEnumerable<T>**接口以支持某个类型的 **foreach** 编程方法，但是 **IEnumerable** 和 **IEnumerable<T>**是 **LINQ** 中非常重要的接口，在 **LINQ** 中也大量的使用 **IEnumerable** 和 **IEnumerable<T>**进行封装，示例代码如下所示。

```
public static IEnumerable<TSource> Where<TSource>
(this IEnumerable<TSource> source,Func<TSource, Boolean> predicate)    //内部实现
{
    foreach (TSource element in source)                               //内部遍历传递的集合
    {
        if (predicate(element))
            yield return element;                                     //返回集合信息
    }
}
```

上述代码为 **LINQ** 内部的封装，从代码中可以看到，在 **LINQ** 内部也大量的使用了 **IEnumerable** 和 **IEnumerable<T>**接口实现 **LINQ** 查询。**IEnumerable** 原本就是.NET Framework 中最基本的集合访问器，而 **LINQ** 是面向关系（有序 **N** 元组集合）的，自然也就是面向 **IEnumerable<T>**的，所以了解 **IEnumerable** 和 **IEnumerable<T>**对 **LINQ** 的理解是有一定帮助的。

## 20.2.4 IQueryableProvider 和 IQueryable<T>接口

**IQueryable** 和 **IQueryable<T>**同样是 **LINQ** 中非常重要的接口，在 **LINQ** 查询语句中，**IQueryable** 和 **IQueryable<T>**接口为 **LINQ** 查询语句进行解释和翻译工作，开发人员能够通过重写 **IQueryable** 和 **IQueryable<T>**接口以实现用不同的方法进行不同的 **LINQ** 查询语句的解释。

**IQueryable<T>**继承于 **IEnumerable<T>**和 **IQueryable** 接口，在 **IQueryable** 中包括两个重要的属性，这两个属性分别为 **Expression** 和 **Provider**。**Expression** 和 **Provider** 分别表示获取与 **IQueryable** 的实例关联的表达式目录树和获取与数据源关联的查询提供程序，**Provider**作为其查询的翻译程序实现 **LINQ** 查询语句的解释。通过 **IQueryable** 和 **IQueryable<T>**接口，开发人员能够自定义 **LINQ Provider**。

注意：**Provider** 可以被看做是一个提供者，用于提供 **LINQ** 中某个语句的解释工具，在 **LINQ** 中通过编程的方法能够实现自定义 **Provider**。

在 **IQueryable** 和 **IQueryable<T>**接口中，还需要另外一个接口，这个接口就是 **IQueryProvider**，该接口用于分解表达式，实现 **LINQ** 查询语句的解释工作，这个接口也是整个算法的核心。**IQueryable<T>**接口在 **MSDN** 中的定义如下所示。

```
public interface IQueryable<T> : IEnumerable<T>, IQueryable, IEnumerable
{
}
public interface IQueryable : IEnumerable
{
    Type ElementType { get; }                                     //获取元素类型
    Expression Expression { get; }                                 //获取表达式
    IQueryProvider Provider { get; }                               //获取提供者
}
```

上述代码定义了 **IQueryable<T>**接口的规范，用于保持数据源和查询状态，**IQueryProvider**在 **MSDN** 中定义如下所示。

```
public interface IQueryProvider
{
    IQueryable CreateQuery(Expression expression);                //创建可执行对象
}
```

```

    IQueryable<TElement> CreateQuery<TElement>(Expression expression);           //创建可执行对象
    object Execute(Expression expression);                                         //计算表达式
    TResult Execute<TResult>(Expression expression);                             //计算表达式
}

```

**IQueryProvider** 用于 **LINQ** 查询语句的核心算法的实现，包括分解表达式和表达式计算等。为了能够创建自定义 **LINQ Provider**，可以编写接口的实现。示例代码如下所示。

```

public IQueryable<TElement> CreateQuery<TElement>(Expression expression)
{
    query.expression = expression;                                               //声明表达式
    return (IQueryable<TElement>)query;                                         //返回 query 对象
}

```

上述代码用于构造一个可用来执行表达式计算的 **IQueryable** 对象，在接口中可以看到需要实现两个相同的执行表达式的 **IQueryable** 对象，另一个则是执行表达式对象的集合，其实现代码如下所示。

```

public IQueryable CreateQuery(Expression expression)
{
    return CreateQuery<T>(expression);                                           //返回表达式的集合
}

```

而作为表达式解释和翻译的核心接口，则需要通过算法实现相应 **Execute** 方法，示例代码如下所示。

```

public TResult Execute<TResult>(Expression expression)
{
    var exp = expression as MethodCallExpression;                               //创建表达式对象
    var data = ((exp.Arguments[0] as ConstantExpression).Value as MyQuery<T>).Data;
    var func = (exp.Arguments[1] as UnaryExpression).Operand as Expression
    <System.Func<T, bool>>;
    var lambda = Expression.Lambda<Func<T, bool>>(func.Body, func.Parameters[0]);
    var r = data.Where(lambda.Compile());                                         //编译表达式
    return (TResult)r.GetEnumerator();
}

```

上述代码通过使用 **lambda** 表达式进行表达式的计算，实现了 **LINQ** 中查询的解释功能。在 **LINQ** 中，对于表达式的翻译和执行过程都是通过 **IQueryProvider** 和 **IQueryable<T>** 接口来实现的。**IQueryProvider** 和 **IQueryable<T>** 实现用户表达式的翻译和解释，在 **LINQ** 应用程序中，通常无需通过 **IQueryProvider** 和 **IQueryable<T>** 实现自定义 **LINQ Provider**，因为 **LINQ** 已经提供强大表达式查询和计算功能。了解 **IQueryProvider** 和 **IQueryable<T>** 接口有助于了解 **LINQ** 内部是如何执行的。

## 20.2.5 LINQ 相关的命名空间

**LINQ** 开发为开发人员提供了便利，可以让开发人员以统一的方式对 **IEnumerable<T>** 接口的对象、数据库、数据集以及 **XML** 文档进行访问。从整体上来说，**LINQ** 是这一系列访问技术的统称，对于不同的数据库和对象都有自己的 **LINQ** 名称，例如 **LINQ to SQL**、**LINQ to Object** 等等。当使用 **LINQ** 操作不同的对象时，可能使用不同的命名空间。常用的命名空间如下所示。

- ❑ **System.Data.Linq**: 该命名空间包含支持与 **LINQ to SQL** 应用程序中的关系数据库进行交互的类。
- ❑ **System.Data.Linq.Mapping**: 该命名空间包含用于生成表示关系数据库的结构和内容的 **LINQ to SQL** 对象模型的类。
- ❑ **System.Data.Linq.SqlClient**: 该命名空间包含与 **SQL Server** 进行通信的提供程序类，以及包含查询帮助器方法的类。
- ❑ **System.Linq**: 该命名空间提供支持使用语言集成查询 (**LINQ**) 进行查询的类和接口。
- ❑ **System.Linq.Expression**: 该命名空间包含一些类、接口和枚举，它们使语言级别的代码表达式能够表示为表达式树形式的对象。
- ❑ **System.Xml.Linq**: 包含 **LINQ to XML** 的类，**LINQ to XML** 是内存中的 **XML** 编程接口。

**LINQ** 中常用的命名空间为开发人员提供 **LINQ** 到数据库和对象的简单的解决方案，开发人员能够通过这些命名空间提供的类进行数据查询和整理，这些命名空间统一了相应的对象的查询方法，如数据集和数

数据库都可以使用类似的 LINQ 语句进行查询操作。

## 20.3 Lambda 表达式

**Lambda** 表达式是一种高效的类似于函数式编程的表达式，**Lambda** 简化了开发中需要编写的代码量。**Lambda** 表达式是由 .NET 2.0 演化过来的，也是 LINQ 的基础，熟练掌握 **Lambda** 表达式能够快速的上手 LINQ 应用开发。

### 20.3.1 匿名方法

在了解 **Lambda** 表达式之前，需要了解什么是匿名方法，匿名方法简单的说就是没有名字的方法，而通常情况下的方法定义是需要名字的，示例代码如下所示。

```
public int sum(int a, int b) //创建方法
{
    return a + b; //返回值
}
```

上面这个方法就是一个常规方法，这个方法需要方法修饰符（**public**）、返回类型（**int**）方法名称（**sum**）和参数列表。而匿名方法可以看作是一个委托的扩展，是一个没有命名的方法，示例代码如下所示。

```
delegate int Sum(int a,int b); //声明匿名方法
protected void Page_Load(object sender, EventArgs e)
{
    Sum s = delegate(int a,int b) //使用匿名方法
    {
        return a + b; //返回值
    };
}
```

上述代码声明了一个匿名方法 **Sum** 但是没有实现匿名方法的操作的实现，在声明匿名方法对象时，可以通过参数格式创建一个匿名方法。匿名方法能够通过传递的参数进行一系列操作，示例代码如下所示。

```
Response.Write(s(5,6).ToString());
```

上述代码使用了 **s(5,6)** 方法进行两个数的加减，匿名方法虽然没有名称，但是同样可以使用 “（” “）” 号进行方法的使用。

**注意：**虽然匿名方法没有名称，但是编译器在编译过程中，还是会为该方法定义一个名称，只是在开发过程中这个名称是不被开发人员所看见的。

除此之外，匿名方法还能够使用一个现有的方法作为其方法的委托，示例代码如下所示。

```
delegate int Sum(int a,int b); //方法委托
public int retSum(int a, int b) //普通方法
{
    return a + b;
}
```

上述代码声明了一个匿名方法，并声明了一个普通的方法，在代码中使用匿名方法代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    Sum s = retSum; //使用匿名方法
    int result = s(10, 10);
}
```

从上述代码中可以看出，匿名方法是一个没有名称的方法，但是匿名方法可以将方法名作为参数进行传递，如上述代码中变量 **s** 就是一个匿名方法，这个匿名方法的方法体被声明为 **retSum**，当编译器进行编



译时，匿名方法会使用 **retSum** 执行其方法体并进行运算。匿名方法最明显的好处就是可以降低常规方法编写时的工作量，另外一个好处就是可以访问调用者的变量，降低传参数的复杂度。

### 20.3.2 Lambda 表达式基础

在了解了匿名方法后，就能够开始了解 **Lambda** 表达式，**Lambda** 表达式在一定程度上就是匿名方法的另一种表现形式。为了方便对 **Lambda** 表达式的解释，首先需要创建一个 **People** 类，示例代码如下所示。

```
public class People
{
    public int age { get; set; }           //设置属性
    public string name { get; set; }       //设置属性
    public People(int age,string name)     //设置属性（构造函数构造）
    {
        this.age = age;                   //初始化属性值 age
        this.name = name;                 //初始化属性值 name
    }
}
```

上述代码定义了一个 **People** 类，并包含一个默认的构造函数能够为 **People** 对象进行年龄和名字的定义。在应用程序设计中，很多情况下需要创建对象的集合，创建对象的集合有利于对对象进行操作和排序等操作，以便在集合中筛选相应的对象。使用 **List** 进行泛型编程，可以创建一个对象的集合，示例代码如下所示。

```
List<People> people = new List<People>(); //创建泛型对象
People p1 = new People(21,"guojing");    //创建一个对象
People p2 = new People(21, "wujunmin");  //创建一个对象
People p3 = new People(20, "muqing");    //创建一个对象
People p4 = new People(23, "lupan");     //创建一个对象
people.Add(p1);                          //添加一个对象
people.Add(p2);                          //添加一个对象
people.Add(p3);                          //添加一个对象
people.Add(p4);                          //添加一个对象
```

上述代码创建了 4 个对象，这 4 个对象分别初始化了年龄和名字，并添加到 **List** 列表中。当应用程序需要对列表中的对象进行筛选时，例如需要筛选年龄大于 20 岁的人时，就需要从列表中筛选，示例代码如下所示。

```
IEnumerable<People> results = people.Where(delegate(People p) { return p.age > 20; });//匿名方法
```

上述代码通过使用 **IEnumerable** 接口创建了一个 **result** 集合，并且该集合中填充的是年龄大于 20 的 **People** 对象。细心的读者就能够发现在这里使用了一个匿名方法进行筛选，因为该方法没有名称，该匿名方法通过使用 **People** 类对象的 **age** 字段进行筛选。

虽然上述代码中执行了筛选操作，但是使用匿名方法往往不太容易理解和阅读，而 **Lambda** 表达式相比于匿名方法而言更加容易理解和阅读，示例代码如下所示。

```
IEnumerable<People> results = people.Where(People => People.age > 20); //Lambda
```

上述代码同样返回了一个 **People** 对象的集合给变量 **results**，但是其编写的方法更加容易阅读，这里可以看出 **Lambda** 表达式在编写的格式上和匿名方法非常相似。其实当编译器开始编译并运行，**Lambda** 表达式最终也表现为匿名方法。

使用匿名方法实际上并不是创建了没有名称的方法，实际上编译器会创建一个方法，这个方法对于开发人员来说是看不见的，该方法会将 **People** 类的对象中符合 **p.age>20** 条件的对象返回并填充到集合中。相同的是，使用 **Lambda** 表达式，当编译器编译时，**Lambda** 表达式同样会被编译成一个匿名方法进行相应的操作，但是相比于匿名方法而言，**Lambda** 表达式更容易阅读，**Lambda** 表达式的格式如下所示。

```
(参数列表)=>表达式或者语句块
```

如上述代码中，参数列表就是 **People** 类，表达式和语句块就是 **People.age>20**，使用 **Lambda** 表达式能够让人很容易的理解该语句究竟是如何执行的，虽然匿名方法提供了同样的功能，却并不容易理解。相比



之下 **People** => **People.age > 20** 却能够很好的理解为“返回一个年纪大于 **20** 的人”。其实 **Lambda** 表达式并没有什么高深的技术，**Lambda** 表达式可以看作是匿名方法的另一种表现形式。其实 **Lambda** 表达式经过反编译后，与匿名方法并没有什么区别。

20.3.3 Lambda 表达式格式

**Lambda** 表达式是匿名方法的另一种表现形式。比较 **Lambda** 表达式和匿名方法，在匿名方法中，“（”，“）”内是方法的参数的集合，这就对应了 **Lambda** 表达式中“(参数列表)”，而匿名方法中“{”，“}”内是方法的语句块，这也对应了 **Lambda** 表达式“=>”符号右边的表达式和语句块项。由于 **Lambda** 表达式是一种匿名方法，所以 **Lambda** 表达式也包含一些基本格式，这些基本格式如下所示。

**Lambda** 表达式可以有多个参数，一个参数，或者无参数。其参数类型可以隐式或者显式。示例代码如下所示：

(x, y) => x * y	//多参数，隐式类型=> 表达式
x => x * 5	//单参数， 隐式类型=>表达式
x => { return x * 5; }	//单参数，隐式类型=>语句块
(int x) => x * 5	//单参数，显式类型=>表达式
(int x) => { return x * 5; }	//单参数，显式类型=>语句块
() => Console.WriteLine()	//无参数

上述格式都是 **Lambda** 表达式的合法格式，在编写 **Lambda** 表达式时，可以忽略参数的类型，因为编译器能够根据上下文直接推断参数的类型，示例代码如下所示。

(x, y) => x + y	//多参数，隐式类型=> 表达式
-----------------	------------------

**Lambda** 表达式的主体可以是表达式也可以是语句块，这样就节约了代码的编写。

注意：**Lambda** 表达式与匿名方法的另一个不同是，**Lambda** 表达式的主体可以是表达式也可以是语句块，而匿名方法中不能包含表达式。

**Lambda** 表达式中的表达式和表达式体都能够被转换成表达式树，这在表达式树的构造上会起到很好的作用，表达式树也是 **LINQ** 中最基本最重要的概念。

20.3.4 Lambda 表达式树

**Lambda** 表达式树也是 **LINQ** 中最重要的一个概念，**Lambda** 表达式树允许开发人员像处理数据一样对 **Lambda** 表达式进行修改。理解 **Lambda** 表达式树的概念并不困难，**Lambda** 表达式树就是将 **Lambda** 表达式转换成树状结构，在使用 **Lambda** 表达式树之前还需要使用 **System.Linq.Expressions** 命名空间，示例代码如下所示。

using System.Linq.Expressions;	//使用命名空间
--------------------------------	----------

**Lambda** 表达式树的基本形式有两种，这两种形式代码如下所示。

Func<int, int> func = pra => pra * pra;	//创建表达式树
Expression<Func<int, int>> expression = pra => pra * pra;	//创建表达式树

**Lambda** 表达式树就是将 **Lambda** 表达式转换成树状结构，示例代码如下所示。

Func<int, int> func = (pra => pra * pra);	//创建表达式
Response.Write(func(8).ToString());	//执行表达式
Response.Write("<hr/>");	

上述代码直接用 **Lambda** 表达式初始化 **Func** 委托，运行后返回的结果为 **64**，同样使用 **Expression** 类也可以实现相同的效果，示例代码如下所示。

Expression<Func<int, int>> expression = pra => pra * pra;	//创建表达式
Func<int, int> func1 = expression.Compile();	//编译表达式
Response.Write(func1(8).ToString());	

上述代码运行后同样返回 64，运行后如图 20-10 所示。使用 **Func** 类和 **Expression** 类创建 **Lambda** 表达式运行结果基本相同，但是 **Func** 方法和 **Expression** 方法是有区别的，如 **Lambda** 表达式 `pra => pra * pra`，**Expression** 首先会分析该表达式并将表达式转换成树状结构，如图 20-11 所示。



图 20-10 Lambda 表达式树

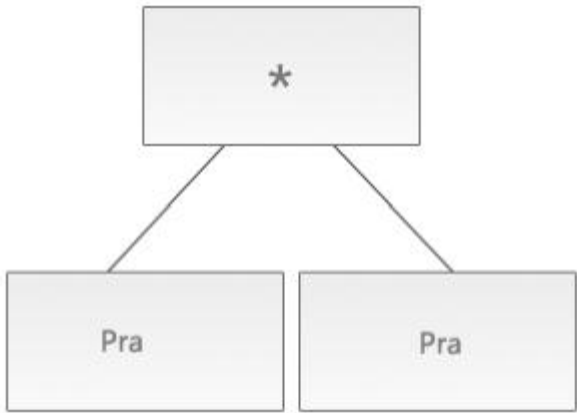


图 20-11 Lambda 表达式树格式

当编译器编译 **Lambda** 表达式时，如果 **Lambda** 表达式使用的是 **Func** 方法，则编译器会将 **Lambda** 表达式直接编译成匿名方法，而如果 **Lambda** 表达式使用的是 **Expression** 方法，则编译器会将 **Lambda** 表达式进行分析、处理然后得到一种数据结构。

20.3.5 访问 Lambda 表达式树

既然在 **LINQ** 应用开发中常常需要解析 **Lambda** 表达式，则就不能避免的对 **Lambda** 表达式树进行访问，访问 **Lambda** 表达式的方法非常简单，直接将表达式输出即可，示例代码如下所示。

```
Expression<Func<int, int>> expression = pra => pra * pra;           //创建表达式树
Func<int, int> func1 = expression.Compile();                       //表达式树编译
Response.Write(func1(8).ToString());                             //执行表达式
Response.Write(expression.ToString());                            //执行表达式
```

上述代码直接使用 **Expression** 类的对象进行表达式输出，这时候读者可能会想到，是否能够像 **Expression** 类的对象一样直接将 **Func** 对象进行输出，答案是否定的，而如果直接使用 **Func** 对象是不能够输出表达式的，如图 20-12 所示。

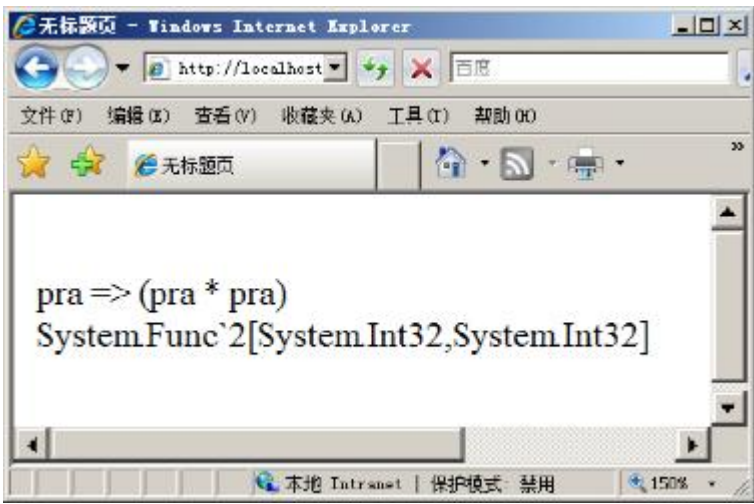


图 20-12 Lambda 表达式输出

正如图 20-12 所示，直接输出 **Expression** 类的对象的值能够进行表达式的输出，而如果输出 **Func** 类的对象只会输出系统信息，如 `System.Func`2[System.Int32,System.Int32]`，这并不是期待的结果。这也从另一个角度说明了 **Func** 方法和 **Expression** 方法是有区别的。

### 20.4 小结

本章介绍了 **LINQ** 的起源，包括什么是 **LINQ**，以及 **LINQ** 在 **.NET 3.5 Framework** 中的位置，本章还介绍了 **LINQ** 基础，包括在 **LINQ** 中常用的接口和类，以及使用 **LINQ** 需要的命名空间。本章还包括：

- 创建使用 **LINQ** 的 **Web** 应用程序：简单的介绍了使用 **LINQ** 实现 **Web** 应用程序中的查询功能。
- 基本的 **LINQ** 数据查询：介绍了 **LINQ** 基本查询功能。
- **Lambda** 表达式基础：介绍了 **Lambda** 表达式基础，以及何为 **Lambda** 表达式。
- **Lambda** 表达式格式：介绍了 **Lambda** 表达式书写中需要使用的格式。
- **Lambda** 表达式树：介绍了 **Lambda** 表达式树。

在介绍 **LINQ** 之前还介绍了 **Lambda** 表达式，**Lambda** 表达式是 **LINQ** 的基础，加深对 **Lambda** 表达式的理解能够更加容易的理解 **LINQ** 的内部机制和原理，实际上，**LINQ** 的使用并不是非常难，关于 **LINQ** 查询将会在下一章中讲到。

## 第 21 章 使用 LINQ 查询

了解了基本的 **LINQ** 基本概念, 以及 **Lambda** 表达式基础后, 就能够使用 **LINQ** 进行应用程序开发。**LINQ** 使用了 **Lambda** 表达式, 以及底层接口实现了对集合的访问和查询, 开发人员能够使用 **LINQ** 对不同的对象, 包括数据库、数据集和 **XML** 文档进行查询。

### 21.1 LINQ 查询概述

**LINQ** 可以对多种数据源和对象进行查询, 如数据库、数据集、**XML** 文档甚至是数组, 这在传统的查询语句中是很难实现的。如果有一个集合类型的值需要进行查询, 则必须使用 **Where** 等方法进行遍历, 而使用 **LINQ** 可以仿真 **SQL** 语句的形式进行查询, 极大的降低了难度。

#### 21.1.1 准备数据源

既然 **LINQ** 可以查询多种数据源和对象, 这些对象可能是数组, 可能是数据集, 也可能是数据库, 那么在使用 **LINQ** 进行数据查询时首先需要准备数据源。

##### 1. 数组

数组中的数据可以被 **LINQ** 查询语句查询, 这样就省去了复杂的数组遍历。数组数据源示例代码如下所示。

```
string[] str = { "学习", "学习 LINQ", "好好学习", "生活很美好" };  
int[] inter = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

数组可以看成是一个集合, 虽然数组没有集合的一些特性, 但是从另一个角度上来说可以看成是一个集合。在传统的开发过程中, 如果要筛选其中包含“学习”字段的某个字符串, 则需要遍历整个数组。

##### 2. SQL Server

在数据库操作中, 同样可以使用 **LINQ** 进行数据库查询。**LINQ** 以其优雅的语法和面向对象的思想能够方便的进行数据库操作, 为了使用 **LINQ** 进行 **SQL Server** 数据库查询, 可以创建两个表, 这两个表的结构如下所示。**Student** (学生表):

- ❑ **S\_ID**: 学生 ID。
- ❑ **S\_NAME**: 学生姓名。
- ❑ **S\_CLASS**: 学生班级。
- ❑ **C\_ID**: 所在班级的 ID。

上述结构描述了一个学生表, 可以使用 **SQL** 语句创建学生表, 示例代码如下所示。

```
USE [student]  
GO  
SET ANSI_NULLS ON  
GO  
SET QUOTED_IDENTIFIER ON  
GO  
  
CREATE TABLE [dbo].[Student](  
    [S_ID] [int] IDENTITY(1,1) NOT NULL,  
    [S_NAME] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,  
    [S_CLASS] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
```



```
[C_ID] [int] NULL,
CONSTRAINT [PK_Student] PRIMARY KEY CLUSTERED
(
    [S_ID] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
```

为了更加详细的描述一个学生所有的基本信息，就需要创建另一个表对该学生所在的班级进行描述，班级表结构如下所示。**Class**（班级表）：

- ❑ **C\_ID**：班级 ID。
- ❑ **C\_GREAD**：班级所在的年级。
- ❑ **C\_INFOR**：班级专业。

上述代码描述了一个班级的基本信息，同样可以使用 **SQL** 语句创建班级表，示例代码如下所示。

```
USE [student]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Class](
    [C_ID] [int] IDENTITY(1,1) NOT NULL,
    [C_GREAD] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
    [C_INFOR] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
    CONSTRAINT [PK_Class] PRIMARY KEY CLUSTERED
(
    [C_ID] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
```

上述代码在 **Student** 数据库中创建了一个班级表，开发人员能够向数据库中添加相应的信息以准备数据源。

## 3. 数据集

**LINQ** 能够通过查询数据集进行数据的访问和整合；通过访问数据集，**LINQ** 能够返回一个集合变量；通过遍历集合变量可以进行其中数据的访问和筛选。在第 9 章中讲到了数据集的概念，开发人员能够将数据库中的内容填充到数据集中，也可以自行创建数据集。

数据集是一个存在于内存的对象，该对象能够模拟数据库的一些基本功能，可以模拟小型的数据库系统，开发人员能够使用数据集对象在内存中创建表，以及模拟表与表之间的关系。在数据集的数据检索过程中，往往需要大量的 **if**、**else** 等判断才能检索相应的数据。

使用 **LINQ** 进行数据集中数据的整理和检索可以减少代码量并优化检索操作。数据集可以是开发人员自己创建的数据集也可以是现有数据库填充的数据集，这里使用上述 **SQL Server** 创建的数据库中的数据进行数据集的填充。

### 21.1.2 使用 LINQ

在传统对象查询中，往往需要很多的 **if**、**else** 语句进行数组或对象的遍历，例如在数组中寻找相应的字段，实现起来往往比较复杂，而使用 **LINQ** 就简化了对象的查询。由于前面已经准备好了数据源，那么就能够分别使用 **LINQ** 语句进行数据源查询。

#### 1. 数组

在前面的章节中，已经创建了一个数组作为数据源，数组示例代码如下所示。

```
int[] inter = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

上述代码是一个数组数据源，如果开发人员需要从其中的元素中搜索大于 **5** 的数字，传统的方法应该遍历整个数组并判断该数字是否大于 **5**，如果大于 **5** 则输出，否则不输出，示例代码如下所示。

```
using System;
using System.Collections.Generic;
using System.Linq; //使用必要的命名空间
using System.Text;
namespace _21_1
{
    class Program
    {
        static void Main(string[] args)
        {
            string[] str = { "学习", "学习 LINQ", "好好学习", "生活很美好" }; //定义数组
            int[] inter = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
            for (int i = 0; i < inter.Length; i++) //遍历数组
            {
                if (inter[i] > 5) //判断数组元素的值是否大于 5
                {
                    Console.WriteLine(inter[i].ToString()); //输出对象
                }
            }
            Console.ReadKey();
        }
    }
}
```

上述代码非常简单，将数组从头开始遍历，遍历中将数组中的的值与 **5** 相比较，如果大于 **5** 就会输出该值，如果小于 **5** 就不会输出该值。虽然上述代码实现了功能的要求，但是这样编写的代码繁冗复杂，也不具有扩展性。如果使用 **LINQ** 查询语句进行查询就非常简单，示例代码如下所示。

```
class Program
{
    static void Main(string[] args)
    {
        string[] str = { "学习", "学习 LINQ", "好好学习", "生活很美好" }; //定义数组
        int[] inter = { 1, 2, 3, 4, 5, 6, 7, 8, 9 }; //定义数组
        var st = from s in inter where s > 5 select s; //执行 LINQ 查询语句
        foreach (var t in st) //遍历集合元素
        {
            Console.WriteLine(t.ToString()); //输出数组
        }
        Console.ReadKey();
    }
}
```

使用 **LINQ** 进行查询之后会返回一个 **IEnumerable** 的集合。在上一章讲过，**IEnumerable** 是.NET 框架中最基本的集合访问器，可以使用 **foreach** 语句遍历集合元素。使用 **LINQ** 查询数组更加容易被阅读，**LINQ** 查询语句的结构和 **SQL** 语法十分类似，**LINQ** 不仅能够查询数组，还可以通过.NET 提供的编程语言进行筛选。例如 **str** 数组变量，如果要查询其中包含“学习”的字符串，对于传统的编程方法是非常冗余和繁琐的。由于 **LINQ** 是.NET 编程语言中的一部分，开发人员就能通过编程语言进行筛选，**LINQ** 查询语句示例代码如下所示。

```
var st = from s in str where s.Contains("学习") select s;
```

## 2. 使用 SQL Server

在传统的数据库开发中，如果需要筛选某个数据库中的数据，可以通过 **SQL** 语句进行筛选。在 **ADO.NET** 中，首先需要从数据库中查询数据，查询后就必须将数据填充到数据集中，然后在数据集中进行数据遍历，示例代码如下所示。

```
try
```

```

{
    SqlConnection
    con = new SqlConnection("server=(local);database='student';uid='sa';pwd='sa'"); //创建连接
    con.Open(); //打开连接
    string strsql = "select * from student,class where student.c_id=class.c_id"; //SQL 语句
    SqlDataAdapter da = new SqlDataAdapter(strsql, con); //创建适配器
    DataSet ds = new DataSet(); //创建数据集
    int j = da.Fill(ds, "mytable"); //填充数据集
    for (int i = 0; i < j; i++) //遍历集合
    {
        Console.WriteLine(ds.Tables["mytable"].Rows[i]["S_NAME"].ToString()); //输出对象
    }
}
catch
{
    Console.WriteLine("数据库连接错误"); //抛出异常
}

```

上述代码进行数据库的访问和查询。在上述代码中，首先需要创建一个连接对象进行数据库连接，然后再打开连接，打开连接之后就要编写 **SELECT** 语句进行数据库查询并填充到 **DataSet** 数据集中，并在 **DataSet** 数据集中遍历相应的表和列进行数据筛选。如果要查询 **C\_ID** 为 1 的学生的所有姓名，有三个办法，这三个办法分别是：

- ☐ 修改 **SQL** 语句。
- ☐ 在循环内进行判断。
- ☐ 使用 **LINQ** 进行查询。

修改 **SQL** 语句是最方便的方法，直接在 **SELECT** 语句中添加查询条件 **WHERE C-ID=1** 就能够实现，但是这个方法扩展性非常的低，如果有其他需求则就需要修改 **SQL** 语句，也有可能造成其余代码填充数据集后数据集内容不同步。

在循环内进行判断也是一种方法，但是这个方法当循环增加时会造成额外的性能消耗，并且当需要扩展时，还需要修改循环代码。最方便的就是使用 **LINQ** 进行查询，在 **Visual Studio 2008** 中提供了 **LINQ to SQL** 类文件用于将现有的数据抽象成对象，这样就符合了面向对象的原则，同时也能够减少代码，提升扩展性。创建一个 **LINQ to SQL** 类文件，直接将服务资源管理器中的相应表拖放到 **LINQ to SQL** 类文件可视化窗口中即可，如图 21-1 所示。



图 21-1 创建 LINQ to SQL 文件

创建了 **LINQ to SQL** 类文件后，就可以直接使用 **LINQ to SQL** 类文件提供的类进行查询，示例代码如下所示。

```

linqtosqlDataContext lq = new linqtosqlDataContext();
var mylq = from l in lq.Student from cl in lq.Class where l.C_ID==cl.C_ID select l; //执行查询
foreach (var result in mylq) //遍历集合
{
    Console.WriteLine(result.S_NAME.ToString()); //输出对象
}

```

上述代码只用了很短的代码就能够实现数据库中数据的查询和遍历，并且从可读性上来说也很容易理解，因为 **LINQ** 查询语句的语法基本与 **SQL** 语法相同，只要有一定的 **SQL** 语句基础就能够非常容易的编写

**LINQ** 查询语句。

## 3. 数据集

**LINQ** 同样对数据集支持查询和筛选操作。其实数据集也是集合的表现形式，数据集除了能够填充数据库中的内容以外，开发人员还能够通过对数据集的操作向数据集中添加数据和修改数据。前面的章节中已经讲到，数据集可以看作是内存中的数据库。数据集能够模拟基本的数据库，包括表、关系等。这里就将 **SQL Server** 中的数据填充到数据集即可，示例代码如下所示。

```
try
{
    SqlConnection
    con = new SqlConnection("server=(local);database='student';uid='sa';pwd='sa'"); //创建连接
    con.Open(); //打开连接
    string strsql = "select * from student,class where student.c_id=class.c_id"; //执行 SQL
    SqlDataAdapter da = new SqlDataAdapter(strsql, con); //创建适配器
    DataSet ds = new DataSet(); //创建数据集
    da.Fill(ds, "mytable"); //填充数据集
    DataTable tables = ds.Tables["mytable"]; //创建表
    var dslq = from d in tables.AsEnumerable() select d; //执行 LINQ 语句
    foreach (var res in dslq)
    {
        Console.WriteLine(res.Field<string>("S_NAME").ToString()); //输出对象
    }
}
catch
{
    Console.WriteLine("数据库连接错误");
}
```

上述代码使用 **LINQ** 针对数据集中的数据进行筛选和整理，同样能够以一种面向对象的思想进行数据集中数据的筛选。在使用 **LINQ** 进行数据集操作时，**LINQ** 不能直接从数据集对象中查询，因为数据集对象不支持 **LINQ** 查询，所以需要使用 **AsEnumerable** 方法返回一个泛型的对象以支持 **LINQ** 的查询操作，示例代码如下所示。

```
var dslq = from d in tables.AsEnumerable() select d; //使用 AsEnumerable
```

上述代码使用 **AsEnumerable** 方法就可以让数据集中的表对象能够支持 **LINQ** 查询。

## 21.1.3 执行 LINQ 查询

从上一节可以看出 **LINQ** 在编程过程中极大的方便了开发人员对于业务逻辑的处理代码的编写，在传统的编程方法中复杂、冗余、难以实现的方法在 **LINQ** 中都能很好的解决。**LINQ** 不仅能够像 **SQL** 语句一样编写查询表达式，**LINQ** 最大的优点也包括 **LINQ** 作为编程语言的一部分，可以使用编程语言提供的特性进行 **LINQ** 条件语句的编写，这就弥补了 **SQL** 语句中的一些不足。在前面的章节中将一些复杂的查询和判断的代码简化成 **LINQ** 应用后，就能够执行应用程序判断 **LINQ** 是否查询和筛选出了所需要的值。

### 1. 数组

在数组数据源中，开发人员希望能够筛选出大于 **5** 的元素。开发人员将传统的代码修改成 **LINQ** 代码并通过 **LINQ** 查询语句进行筛选，示例代码如下所示。

```
var st = from s in inter where s > 5 select s; //执行 LINQ 查询
```

上述代码将查询在 **inter** 数组中的所有元素并返回其中元素的值大于 **5** 的元素的集合，运行后如图 21-2 所示。





图 21-2 遍历数组

**LINQ** 执行了条件语句并返回了元素的值大于 5 的元素。**LINQ** 语句能够方便的扩展，当有不同的需求时，可以修改条件语句进行逻辑判断，例如可以筛选一个平方数为偶数的数组元素，直接修改条件即可，**LINQ** 查询语句如下所示。

```
var st = from s in inter where (s*s)%2==0 select s; //执行 LINQ 查询
```

上述代码通过条件  $(s*s)\%2==0$  将数组元素进行筛选，选择平方数为偶数的数组元素的集合，运行后如图 21-3 所示。

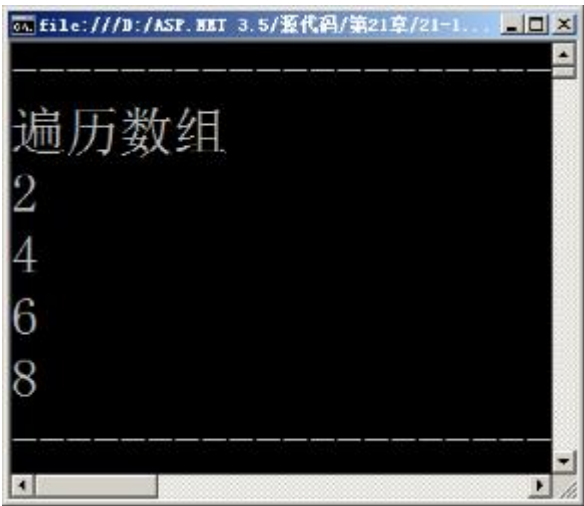


图 21-3 更改筛选条件

2. 使用 SQL Server

在 **LINQ to SQL** 类文件中，**LINQ to SQL** 类文件已经将数据库的模型封装成一个对象，开发人员能够通过面向对象的思想访问和整合数据库。**LINQ to SQL** 也对 **SQL** 做了补充，使用 **LINQ to SQL** 类文件能够执行更强大的筛选，**LINQ** 查询语句代码如下所示。

```
var mylq = from l in lq.Student from cl in lq.Class where l.C_ID==cl.C_ID select l; //执行 LINQ 查询
```

上述代码从 **Student** 表和 **Class** 表中筛选了 **C\_ID** 相等的学生信息，这很容易在 **SQL** 语句中实现。**LINQ** 作为编程语言的一部分，可以使用更多的编程方法实现不同的筛选需求，例如筛选名称中包含“郭”字的学生名称在传统的 **SQL** 语句中就很难通过一条语句实现，而在 **LINQ** 中就能够实现，示例代码如下所示。

```
var mylq = from l in lq.Student from cl in lq.Class where l.C_ID==cl.C_ID where l.S_NAME.Contains("郭") select l; //执行 LINQ 条件查询
```

上述代码使用了 **Contains** 方法判断一个字符串中是否包含某个字符或字符串，这样不仅方便阅读，也简化了查询操作，运行后如图 21-4 和图 21-5 所示。



图 21-4 简单查询

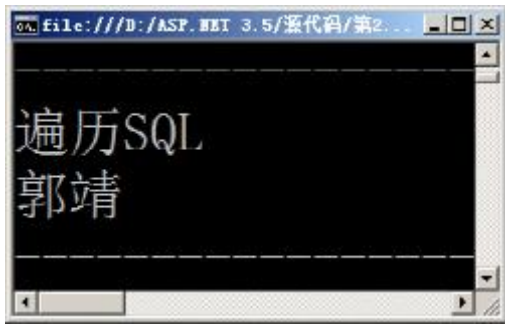


图 21-5 条件查询

**LINQ** 返回了符合条件的元素的集合，并实现了筛选操作。**LINQ** 不仅作为编程语言的一部分，简化了开发人员的开发操作，从另一方面讲，**LINQ** 也补充了在 **SQL** 中难以通过几条语句实现的功能的实现。从上面的 **LINQ** 查询代码可以看出，就算是不同的对象、不同的数据源，其 **LINQ** 基本的查询语法都非常相似，并且 **LINQ** 还能够支持编程语言具有的特性从而弥补 **SQL** 语句的不足。在数据集的查询中，其查询语句也可以直接使用而无需大面积修改代码，这样代码就具有了更高的维护性和可读性。

## 21.2 LINQ 查询语法概述

从上面的章节中可以看出，**LINQ** 查询语句能够将复杂的查询应用简化成一个简单的查询语句，不仅如此，**LINQ** 还支持编程语言本有的特性进行高效的数据访问和筛选。虽然 **LINQ** 在写法上和 **SQL** 语句十分相似，但是 **LINQ** 语句在其查询语法上和 **SQL** 语句还是有出入的，**SQL** 查询语句如下所示。

```
select * from student,class where student.c_id=class.c_id //SQL 查询语句
```

上述代码是 **SQL** 查询语句，对于 **LINQ** 而言，其查询语句格式如下所示。

```
var mylq = from l in lq.Student from cl in lq.Class where l.C_ID==cl.C_ID select l; //LINQ 查询语句
```

上述代码作为 **LINQ** 查询语句实现了同 **SQL** 查询语句一样的效果，但是 **LINQ** 查询语句在格式上与 **SQL** 语句不同，**LINQ** 的基本格式如下所示。

```
var <变量> = from <项目> in <数据源> where <表达式> orderby <表达式>
```

**LINQ** 语句不仅能够支持对数据源的查询和筛选，同 **SQL** 语句一样，还支持 **ORDER BY** 等排序，以及投影等操作，示例查询语句如下所示。

```
var st = from s in inter where s==3 select s; //LINQ 查询
var st = from s in inter where (s * s) % 2 == 0 orderby s descending select s; //LINQ 条件查询
```

从结构上来看，**LINQ** 查询语句同 **SQL** 查询语句中比较大的区别就在于 **SQL** 查询语句中的 **SELECT** 关键字在语句的前面，而在 **LINQ** 查询语句中 **SELECT** 关键字在语句的后面，在其他地方没有太大的区别，对于熟悉 **SQL** 查询语句的人来说非常容易上手。

## 21.3 基本子句

既然 **LINQ** 查询语句同 **SQL** 查询语句一样，能够执行条件、排序等操作，这些操作就需要使用 **WHERE**、**ORDERBY** 等关键字，这些关键字在 **LINQ** 中是基本子句。同 **SQL** 查询语句中的 **WHERE**、**ORDER BY** 操作一样，都为元素进行整合和筛选。

### 21.3.1 from 查询子句

**from** 子句是 **LINQ** 查询语句中最基本也是最关键的子句关键字，与 **SQL** 查询语句不同的是，**from** 关键

字必须在 **LINQ** 查询语句的开始。

## 1. from 查询子句基础

后面跟随着项目名称和数据源，示例代码如下所示。

```
var linqstr = from lq in str select lq; //form 子句
```

**from** 语句指定项目名称和数据源，并且指定需要查询的内容，其中项目名称作为数据源的一部分而存在，用于表示和描述数据源中的每个元素，而数据源可以是数组、集合、数据库甚至是 **XML**。值得一提的是，**from** 子句的数据源的类型必须为 **IEnumerable**、**IEnumerable<T>** 类型或者是 **IEnumerable**、**IEnumerable<T>** 的派生类，否则 **from** 不能够支持 **LINQ** 查询语句。

在 **.NET Framework** 中泛型编程中，**List**（可通过索引的强类型列表）也能够支持 **LINQ** 查询语句的 **from** 关键字，因为 **List** 实现了 **IEnumerable**、**IEnumerable<T>** 类型，在 **LINQ** 中可以对 **List** 类进行查询，示例代码如下所示。

```
static void Main(string[] args)
{
    List<string> MyList = new List<string>(); //创建一个列表项
    MyList.Add("guojing"); //添加一项
    MyList.Add("wujunmin"); //添加一项
    MyList.Add("muqing"); //添加一项
    var linqstr = from l in MyList select l; //LINQ 查询
    foreach (var element in linqstr) //遍历集合
    {
        Console.WriteLine(element.ToString()); //输出对象
    }
    Console.ReadKey();
}
```

上述代码创建了一个列表项并向列表中添加若干项进行 **LINQ** 查询。由于 **List<T>** 实现了 **IEnumerable**、**IEnumerable<T>**，所以 **List<T>** 列表项可以支持 **LINQ** 查询语句的 **from** 关键字，如图 21-6 所示。

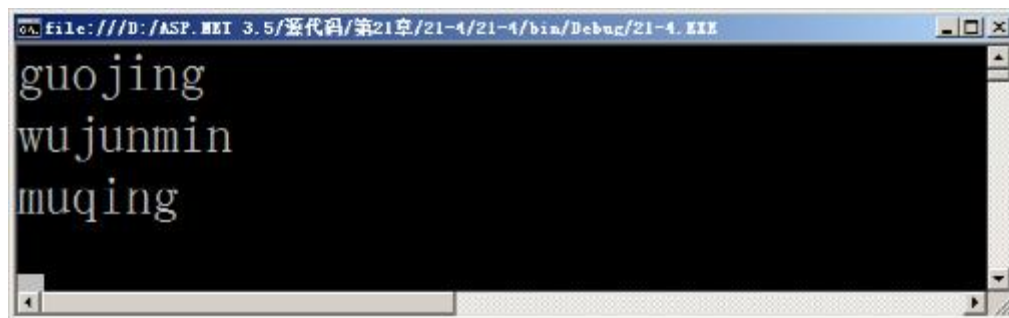


图 21-6 from 子句

顾名思义，**from** 语句可以被理解为“来自”，而 **in** 可以被理解为“在哪个数据源中”，这样 **from** 语句就很好理解了，如 **from l in MyList select l** 语句可以翻译成“找到来自 **MyList** 数据源中的集合 **l**”，这样就能够更加方便的理解 **from** 语句。

## 2. from 查询子句嵌套查询

在 **SQL** 语句中，为了实现某一功能，往往需要包含多个条件，以及包含多个 **SQL** 子句嵌套。在 **LINQ** 查询语句中，并没有 **and** 关键字为复合查询提供功能。如果需要进行复杂的复合查询，可以在 **from** 子句中嵌套另一个 **from** 子句即可，示例代码如下所示。

```
var linqstr = from lq in str from m in str2 select lq; //使用嵌套查询
```

上述代码就使用了一个嵌套查询进行 **LINQ** 查询。在有多个数据源或者包括多个表的数据需要查询时，可以使用 **LINQfrom** 子句嵌套查询，数据源示例代码如下所示。

```
List<string> MyList = new List<string>(); //创建一个数据源
MyList.Add("guojing"); //添加一项
MyList.Add("wujunmin"); //添加一项
MyList.Add("muqing"); //添加一项
MyList.Add("yuwen"); //添加一项
```

```
List<string> MyList2 = new List<string>(); //创建另一个数据源
MyList2.Add("guojing's phone"); //添加一项
MyList2.Add("wujunmin's phone "); //添加一项
MyList2.Add("muqing's phone "); //添加一项
MyList2.Add("lupan's phone "); //添加一项
```

上述代码创建了两个数据源，其中一个数据源存放了联系人的姓名的拼音名称，另一个则存放了联系人的电话信息。为了方便的查询在数据源中“联系人”和“联系人电话”都存在并且匹配的数据，就需要使用 **from** 子句嵌套查询，示例代码如下所示。

```
var linqstr = from l in MyList from m in MyList2 where m.Contains(l) select l; //from 子句嵌套查询
foreach (var element in linqstr) //遍历集合元素
{
    Console.WriteLine(element.ToString()); //输出对象
}
Console.ReadKey();
```

上述代码使用了 **LINQ** 语句进行嵌套查询，嵌套查询在 **LINQ** 中会被经常使用到，因为开发人员常常遇到需要面对多个表多个条件，以及不同数据源或数据源对象的情况，使用 **LINQ** 查询语句的嵌套查询可以方便的在不同的表和数据源对象之间建立关系。

21.3.2 where 条件子句

在 **SQL** 查询语句中可以使用 **where** 子句进行数据的筛选，在 **LINQ** 中同样包括 **where** 子句进行数据源中数据的筛选。**where** 子句指定了筛选的条件，这也就是说在 **where** 子句中的代码段必须返回布尔值才能够进行数据源的筛选，示例代码如下所示。

```
var linqstr = from l in MyList where l.Length > 5 select l; //编写 where 子句
```

**LINQ** 查询语句可以包含一个或多个 **where** 子句，而 **where** 子句可以包含一个或多个布尔值变量，为了查询数据源中姓名的长度在 6 之上的姓名，可以使用 **where** 子句进行查询，示例代码如下所示。

```
static void Main(string[] args)
{
    List<string> MyList = new List<string>(); //创建 List 对象
    MyList.Add("guojing"); //添加一项
    MyList.Add("wujunmin"); //添加一项
    MyList.Add("muqing"); //添加一项
    MyList.Add("yuwen"); //添加一项
    var linqstr = from l in MyList where l.Length > 6 select l; //执行 where 查询
    foreach (var element in linqstr) //遍历集合
    {
        Console.WriteLine(element.ToString()); //输出对象
    }
    Console.ReadKey();
}
```

上述代码添加了数据源之后，通过 **where** 子句在数据源中进行条件查询，**LINQ** 查询语句会遍历数据源中的数据并进行判断，如果返回值为 **true**，则会在 **linqstr** 集合中添加该元素，运行后如图 21-7 所示。



图 21-7 where 子句查询



当需要多个 **where** 子句进行复合条件查询时，可以使用 “&&” 进行 **where** 子句的整合，示例代码如下所示。

```
static void Main(string[] args)
{
    List<string> MyList = new List<string>();           //创建 List 对象
    MyList.Add("guojing");                             //添加一项
    MyList.Add("wujunmin");                             //添加一项
    MyList.Add("muqing");                             //添加一项
    MyList.Add("guomoruo");                             //添加一项
    MyList.Add("lupan");                             //添加一项
    MyList.Add("guof");                             //添加一项
    var linqstr = from l in MyList where (l.Length > 6 && l.Contains("guo"))||l=="lupan" select l; //复合查询
    foreach (var element in linqstr)                   //遍历集合
    {
        Console.WriteLine(element.ToString());         //输出对象
    }
    Console.ReadKey();
}
```

上述代码进行了多条件的复合查询，查询姓名长度大于 6 并且姓名中包含 **guo** 的姓或者姓名是 “**lupan**” 的人，运行后如图 21-8 所示。

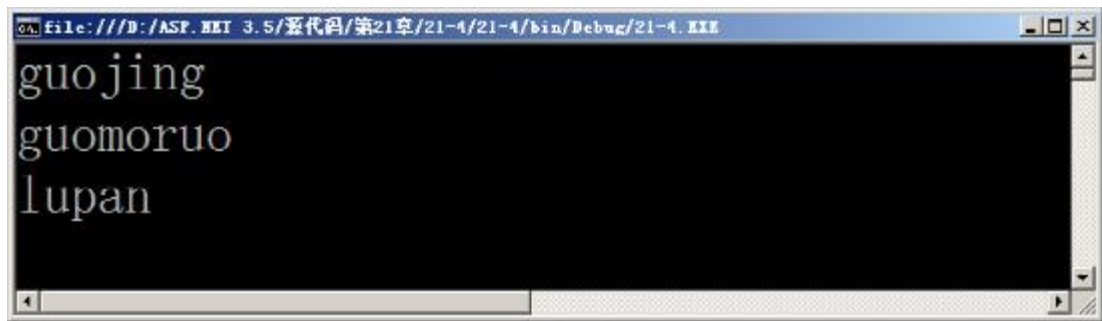


图 21-8 复合 **where** 子句查询

复合 **where** 子句查询通常用于同一个数据源中的数据查询，当需要在同一个数据源中进行筛选查询时，可以使用 **where** 子句进行单个或多个 **where** 子句条件查询，**where** 子句能够对数据源中的数据进行筛选并将复合条件的元素返回到集合中。

## 21.3.3 select 选择子句

**select** 子句同 **from** 子句一样，是 **LINQ** 查询语句中必不可少的关键字，**select** 子句在 **LINQ** 查询语句中是必须的，示例代码如下所示。

```
var linqstr = from lq in str select lq;           //编写选择子句
```

上述代码中包括三个变量，这三个变量分别为 **linqstr**、**lq**、**str**。其中 **str** 是数据源，**linqstr** 是数据源中满足查询条件的集合，而 **lq** 也是一个集合，这个集合来自数据源。在 **LINQ** 查询语句中必须包含 **select** 子句，若不包含 **select** 子句则系统会抛出异常（除特殊情况外）。**select** 语句指定了返回到集合变量中的元素是来自哪个数据源的，示例代码如下所示。

```
static void Main(string[] args)
{
    List<string> MyList = new List<string>();           //创建 List
    MyList.Add("guojing");                             //添加一项
    MyList.Add("wujunmin");                             //添加一项
    MyList.Add("guomoruo");                             //添加一项
    List<string> MyList2 = new List<string>();           //创建 List
    MyList2.Add("guojing's phone");                     //添加一项
    MyList2.Add("wujunmin's phone ");                 //添加一项
}
```

```
MyList2.Add("lupan's phone ");           //添加一项
var linqstr = from l in MyList from m in MyList2 where m.Contains(l) select l;    //select l 变量
foreach (var element in linqstr)          //遍历集合
{
    Console.WriteLine(element.ToString()); //输出集合内容
}
Console.ReadKey();                         //等待用户按键
}
```

上述代码从两个数据源中筛选数据，并通过 **select** 返回集合元素，运行后如图 21-9 所示。

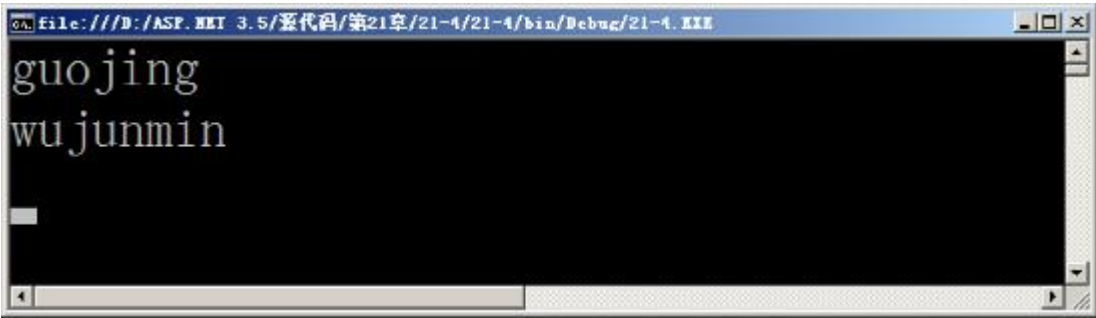


图 21-9 select 子句

如果将 **select** 子句后面的项目名称更改，则结果可能不同，更改 **LINQ** 查询子句代码如下所示。

```
var linqstr = from l in MyList from m in MyList2 where m.Contains(l) select m;    //使用 select
```

上述 **LINQ** 查询子句并没有 **select l** 变量中的集合元素，而是选择了 **m** 集合元素，则返回的应该是 **MyList2** 数据源中的集合元素，运行后如图 21-10 所示。

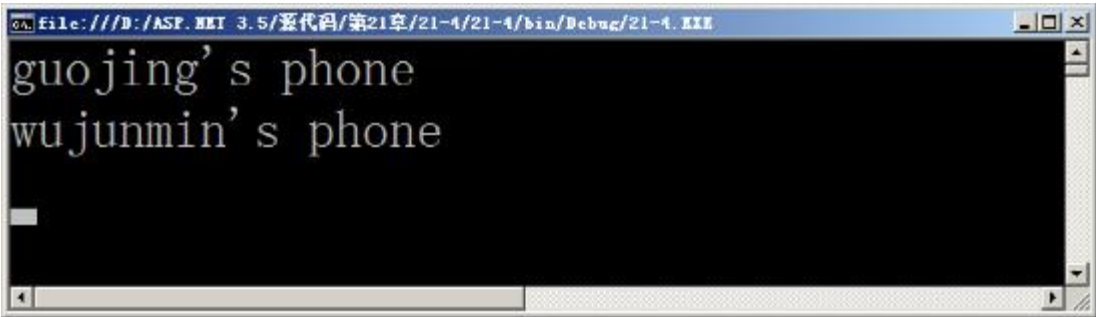


图 21-10 select 子句

对于不同的 **select** 对象返回的结果也不尽相同，当开发人员需要进行复合查询时，可以通过 **select** 语句返回不同的复合查询对象，这在多数据源和多数据对象查询中是非常有帮助的。

21.3.4 group 分组子句

在 **LINQ** 查询语句中，**group** 子句对 **from** 语句执行查询的结果进行分组，并返回元素类型为 **IGrouping<TKey,TElement>** 的对象序列。**group** 子句支持将数据源中的数据进行分组。但进行分组前，数据源必须支持分组操作才可使用 **group** 语句进行分组处理，示例代码如下所示。

```
public class Person
{
    public int age;           //分组条件
    public string name;       //创建姓名字段
    public Person(int age,string name) //构造函数
    {
        this.age = age;      //构造属性值 age
        this.name = name;    //构造属性值 name
    }
}
```

上述代码设计了一个类用于描述联系人的姓名和年级，并且按照年级进行分组，这样数据源就能够支

持分组操作。

注意：虽然数组也可以进行分组操作，因为其绝大部分数据源都能够支持分组操作，但是数组等数据源进行分组操作可能是没有意义的。

这里同样可以通过 **List** 列表以支持 **LINQ** 查询，示例代码如下所示。

```
static void Main(string[] args)
{
    List<Person> PersonList = new List<Person>();
    PersonList.Add(new Person(21,"limusha"));           //通过构造函数构造新对象
    PersonList.Add(new Person(21, "guojing"));           //通过构造函数构造新对象
    PersonList.Add(new Person(22, "wujunmin"));           //通过构造函数构造新对象
    PersonList.Add(new Person(22, "lupan"));             //通过构造函数构造新对象
    PersonList.Add(new Person(23, "yuwen"));             //通过构造函数构造新对象
    var gl = from p in PersonList group p by p.age;      //使用 group 子句进行分组
    foreach (var element in gl)                          //遍历集合
    {
        foreach (Person p in element)                   //遍历集合
        {
            Console.WriteLine(p.name.ToString());       //输出对象
        }
    }
    Console.ReadKey();
}
```

上述代码使用了 **group** 子句进行数据分组，实现了分组的功能，运行后如图 21-11 所示。

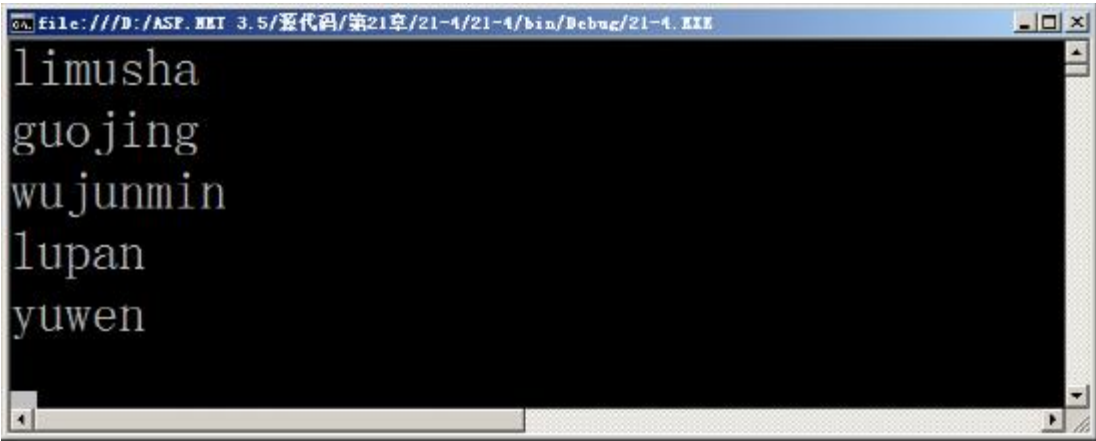


图 21-11 group 子句

正如图 21-11 所示，**group** 子句将数据源中的数据进行分组，在遍历数据元素时，并不像前面的章节那样直接对元素进行遍历，因为 **group** 子句返回的是元素类型为 **IGrouping<TKey,TElement>** 的对象序列，必须在循环中嵌套一个对象的循环才能够查询相应的数据元素。

在使用 **group** 子句时，**LINQ** 查询子句的末尾并没有 **select** 子句，因为 **group** 子句会返回一个对象序列，通过循环遍历才能够在对象序列中找到相应的对象的元素，如果使用 **group** 子句进行分组操作，可以不使用 **select** 子句。

21.3.5 orderby 排序子句

在 **SQL** 查询语句中，常常需要对现有的数据元素进行排序，例如注册用户的时间，以及新闻列表的排序，这样能够方便用户在应用程序使用过程中快速获取需要的信息。在 **LINQ** 查询语句中同样支持排序操作以提取用户需要的信息。在 **LINQ** 语句中，**orderby** 是一个词组而不是分开的，**orderby** 能够支持对象的排序，例如按照用户的年龄进行排序时就可以使用 **orderby** 关键字，示例代码如下所示。

```
public class Person                                     //创建对象
```

```
{
    public int age;                                //创建字段
    public string name;                            //创建字段
    public Person(int age,string name)             //构造函数
    {
        this.age = age;                           //赋值字段
        this.name = name;
    }
}
```

上述代码同样设计了一个 **Person** 类，并通过 **age**、**name** 字段描述类对象。使用 **LINQ**，同样要使用 **List** 类作为对象的容器并进行其中元素的查询，示例代码如下所示。

```
class Program
{
    static void Main(string[] args)
    {
        List<Person> PersonList = new List<Person>();           //创建对象列表
        PersonList.Add(new Person(21,"limusha"));              //年龄为 21
        PersonList.Add(new Person(23, "guojing"));              //年龄为 23
        PersonList.Add(new Person(22, "wujunmin"));              //年龄为 22
        PersonList.Add(new Person(25, "lupan"));                //年龄为 25
        PersonList.Add(new Person(24, "yuwen"));                //年龄为 24
        var gl = from p in PersonList orderby p.age select p;    //执行排序操作
        foreach (var element in gl)                             //遍历集合
        {
            Console.WriteLine(element.name.ToString());          //输出对象
        }
        Console.ReadKey();
    }
}
```

上述代码并没有按照顺序对 **List** 容器添加对象，其中数据的显示并不是按照顺序来显示的。使用 **orderby** 关键字能够指定集合中的元素的排序规则，上述代码按照年龄的大小进行排序，运行后如图 21-12 所示。

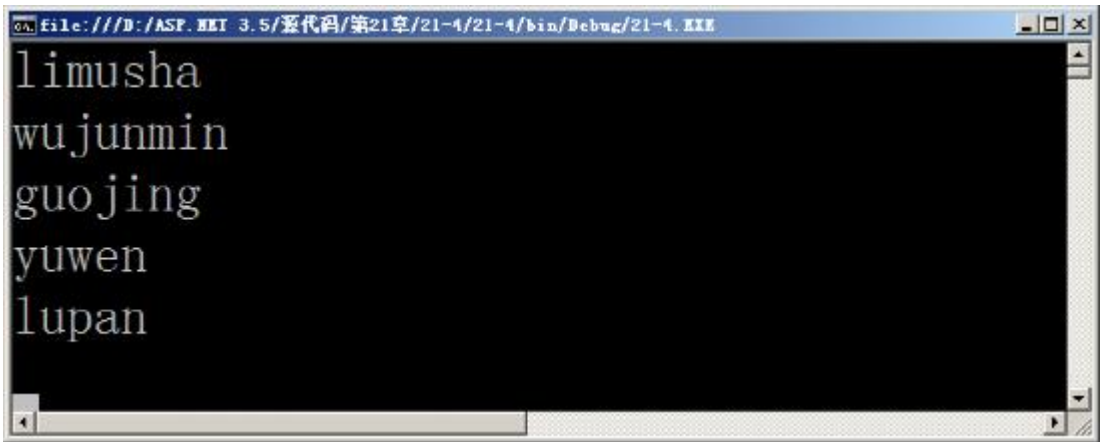


图 21-12 orderby 子句

**orderby** 子句同样能够实现倒序排列，倒序排列在应用程序开发过程中应用的非常广泛，例如新闻等。用户关心的都是当天的新闻而不是很久以前发布的某个新闻，如果管理员发布了一个新的新闻，显示在最上方的应该是最新的新闻。在 **orderby** 子句中可以使用 **descending** 关键字进行倒序排列，示例代码如下所示。

```
var gl = from p in PersonList orderby p.age descending select p;           //orderby 语句
```

上述代码将用户的信息按照其年龄的大小倒序排列，运行如图 21-13 所示。



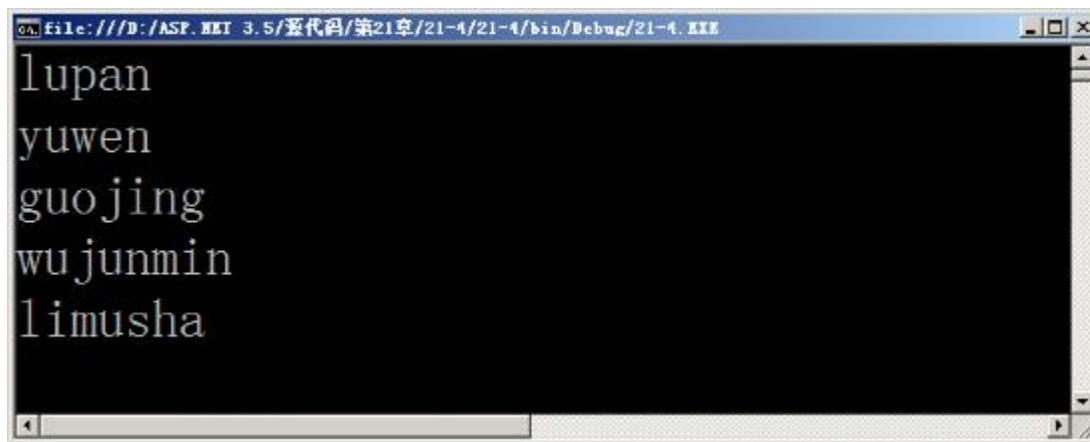


图 21-13 orderby 子句倒序

**orderby** 子句同样能够进行多个条件排序，如果需要使用 **orderby** 子句进行多个条件排序，只需要将这些条件用 “,” 号分割即可，示例代码如下所示。

```
var gl = from p in PersonList orderby p.age descending,p.name select p; //orderby 语句
```

### 21.3.6 into 连接子句

**into** 子句通常和 **group** 子句一起使用，通常情况下，**LINQ** 查询语句中无需 **into** 子句，但是如果需要对分组中的元素进行操作，则需要使用 **into** 子句。**into** 语句能够创建临时标识符用于保存查询的集合，示例代码如下所示。

```
static void Main(string[] args)
{
    List<Person> PersonList = new List<Person>();           //创建对象列表
    PersonList.Add(new Person(21, "limusha"));              //通过构造函数构造新对象
    PersonList.Add(new Person(21, "guojing"));               //通过构造函数构造新对象
    PersonList.Add(new Person(22, "wujunmin"));              //通过构造函数构造新对象
    PersonList.Add(new Person(22, "lupan"));                  //通过构造函数构造新对象
    PersonList.Add(new Person(23, "yuwen"));                  //通过构造函数构造新对象
    var gl = from p in PersonList group p by p.age into x select x; //使用 into 子句创建标识
    foreach (var element in gl)                               //遍历集合
    {
        foreach (Person p in element)                         //遍历集合
        {
            Console.WriteLine(p.name.ToString());            //输出对象
        }
    }
    Console.ReadKey();
}
```

上述代码通过使用 **into** 子句创建标识，从 **LINQ** 查询语句中可以看出，查询后返回的是一个集合变量 **x** 而不是 **p**，但是编译能够通过并且能够执行查询，这说明 **LINQ** 查询语句将查询的结果填充到了临时标识符对象 **x** 中并返回查询集合给 **gl** 集合变量，运行结果如图 21-14 所示。

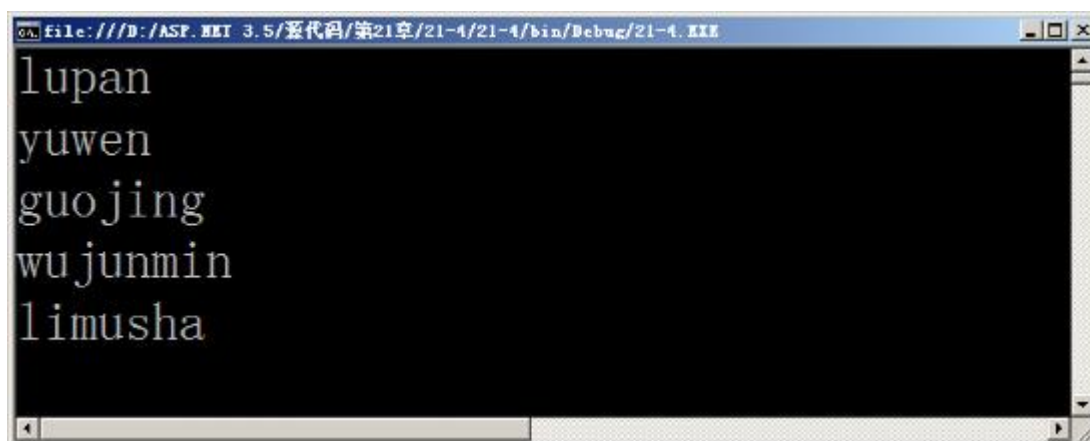


图 21-14 into 子句

注意：into 子句必须以 select、group 等子句作为结尾子句，否则会抛出异常。

21.3.7 join 连接子句

在数据库的结构中，通常表与表之间有着不同的联系，这些联系决定了表与表之间的依赖关系。在 LINQ 中同样也可以使用 join 子句对有关系的数据源或数据对象进行查询，但首先这两个数据源必须要有一定的联系，示例代码如下所示。

```
public class Person                                     //描述“人”对象
{
    public int age;                                     //描述“年龄”字段
    public string name;                                //描述“姓名”字段
    public string cid;                                  //描述“车 ID”字段
    public Person(int age,string name,int cid)          //构造函数
    {
        this.age = age;                               //初始化
        this.name = name;                             //初始化
        this.cid = cid;
    }
}
public class CarInformaion                             //描述“车”对象
{
    public int cid;                                     //描述“车 ID”字段
    public string type;                                //描述“车类型”字段
    public CarInformaion(int cid,string type)          //初始化构造函数
    {
        this.cid = cid;                               //初始化
        this.type = type;                             //初始化
    }
}
```

上述代码创建了两个类，这两个类分别用来描述“人”这个对象和“车”这个对象，CarInformation 对象可以用来描述车的编号以及车的类型，而 Person 类可以用来描述人购买了哪个牌子的车，这就确定了这两个类之间的依赖关系。而在对象描述中，如果将 CarInformation 类的属性和字段放置到 Person 类的属性中，会导致类设计臃肿，同时也没有很好的描述该对象。对象创建完毕就可以使用 List 类创建对象，示例代码如下所示。

```
List<Person> PersonList = new List<Person>();          //创建 List 类
PersonList.Add(new Person(21, "limusha",1));          //购买车 ID 为 1 的人
PersonList.Add(new Person(21, "guojing",2));          //购买车 ID 为 2 的人
PersonList.Add(new Person(22, "wujunmin",3));         //购买车 ID 为 3 的人
List<CarInformaion> CarList = new List<CarInformaion>();
CarList.Add(1, "宝马");                               //车 ID 为 1 的基本信息
CarList.Add(2, "奇瑞");
```

上述代码分别使用了 List 类进行对象的初始化，使用 join 子句就能够进行不同数据源中数据关联的操作和外连接，示例代码如下所示。

```
static void Main(string[] args)
{
    List<Person> PersonList = new List<Person>();      //创建 List 类
    PersonList.Add(new Person(21, "limusha",1));      //购买车 ID 为 1 的人
    PersonList.Add(new Person(21, "guojing",2));      //购买车 ID 为 2 的人
    PersonList.Add(new Person(22, "wujunmin",3));     //购买车 ID 为 3 的人
    List<CarInformaion> CarList = new List<CarInformaion>(); //创建 List 类
    CarList.Add(new CarInformaion(1,"宝马"));         //车 ID 为 1 的车
```

```
CarList.Add(new CarInformaion(2, "奇瑞"));           //车 ID 为 2 的车
var gl = from p in PersonList join car in CarList on p.cid equals car.cid select p; //使用 join 子句
foreach (var element in gl)                          //遍历集合
{
    Console.WriteLine(element.name.ToString());      //输出对象
}
Console.ReadKey();
}
```

上述代码使用 **join** 子句进行不同数据源之间关系的创建，其用法同 **SQL** 查询语句中的 **INNER JOIN** 查询语句相似，运行后如图 21-15 所示。

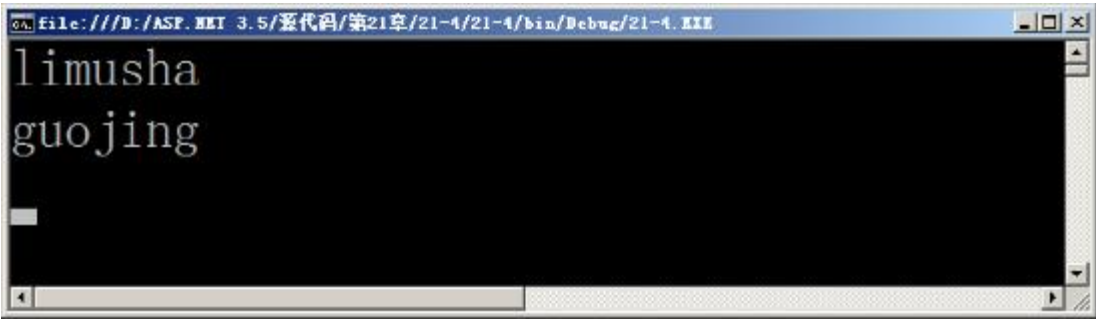


图 21-15 join 查询子句

21.3.8 let 临时表达式子句

在 **LINQ** 查询语句中，**let** 关键字可以看作是在表达式中创建了一个临时的变量用于保存表达式的结果，但是 **let** 子句指定的范围变量的值只能通过初始化操作进行赋值，一旦初始化之后就无法再次进行更改操作。示例代码如下所示。

```
static void Main(string[] args)
{
    List<Person> PersonList = new List<Person>();           //创建 List 类
    PersonList.Add(new Person(21, "limusha",1));           //购买车 ID 为 1 的人
    PersonList.Add(new Person(21, "guojing",2));           //购买车 ID 为 2 的人
    PersonList.Add(new Person(22, "wujunmin",3));           //购买车 ID 为 3 的人
    List<CarInformaion> CarList = new List<CarInformaion>(); //创建 List 类
    CarList.Add(new CarInformaion(1,"宝马"));              //车 ID 为 1 的车
    CarList.Add(new CarInformaion(2, "奇瑞"));              //车 ID 为 2 的车
    var gl = from p in PersonList let car = from c in CarList select c.cid select p; //使用 let 语句
    foreach (var element in gl)                            //遍历集合
    {
        Console.WriteLine(element.name.ToString());        //输出对象
    }
    Console.ReadKey();
}
```

**let** 就相当于是一个中转变量，用于临时存储表达式的值，在 **LINQ** 查询语句中，其中的某些过程的值可以通过 **let** 进行保存。而简单的说，**let** 就是临时变量，如 **x=1+1**、**y=x+2** 这样，其中 **x** 就相当于是一个 **let** 变量，上述代码运行后如图 21-16 所示。



图 21-16 let 子句

## 21.4 LINQ 查询操作

前面介绍了 LINQ 的一些基本的语法，以及 LINQ 常用的查询子句进行数据的访问和整合，甚至建立数据源对象和数据源对象之间的关联，使用 LINQ 查询子句能够实现不同的功能，包括投影、排序和聚合等，本节开始介绍 LINQ 的查询操作。

### 21.4.1 LINQ 查询概述

LINQ 不仅提供了强大的查询表达式为开发人员对数据源进行查询和筛选操作提供遍历，LINQ 还提供了大量的查询操作，这些操作通过实现 `IEnumerable<T>` 或 `IQueryable<T>` 提供的接口实现了投影、排序、聚合等操作。通过使用 LINQ 提供的查询方法，能够快速实现投影、排序等操作。

由于 LINQ 查询操作实现了 `IEnumerable<T>` 或 `IQueryable<T>` 接口，所以 LINQ 查询操作能够通过接口中特定的方法进行查询和筛选，可以直接使用数据源对象变量的方法进行操作。在 LINQ 查询操作的方法中，需要大量的使用 **Lambda** 表达式实现委托，这就从另一个方面说明了 **Lambda** 表达式的重要性。示例代码如下所示。

```
int[] inter = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };           //创建数组
var lint = inter.Select(i => i);                                           //使用 Lambda
```

上述代码使用了 **Select** 方法进行投影操作，在投影操作的参数中使用 **Lambda** 表达式表示了如何实现数据筛选。LINQ 查询操作不仅包括 **Select** 投影操作，还包括排序、聚合等操作，LINQ 常用操作如下所示。

- ❑ **Count**: 计算集合中元素的数量，或者计算满足条件的集合的元素的数量。
- ❑ **GroupBy**: 实现对集合中的元素进行分组的操作。
- ❑ **Max**: 获取集合中元素的最大值。
- ❑ **Min**: 获取集合中元素的最小值。
- ❑ **Select**: 执行投影操作。
- ❑ **SelectMany**: 执行投影操作，可以为多个数据源进行投影操作。
- ❑ **Where**: 执行筛选操作。

LINQ 不只提供上述这些常用的查询操作方法，还提供更多的查询方法，由于本书篇幅有限，只讲解一些常用的查询方法。

### 21.4.2 投影操作

投影操作和 SQL 查询语句中的 **SELECT** 基本类似，投影操作能够指定数据源并选择相应的数据源，在 LINQ 中常用的投影操作包括 **Select** 和 **SelectMany**。

#### 1. Select 选择子句

**Select** 操作能够将集合中的元素投影到新的集合中去，并能够指定元素的类型和表现形式，示例代码如下所示。

```
static void Main(string[] args)
{
    int[] inter = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };           //创建数组
    var lint = inter.Select(i => i);                                           //Select 操作
    foreach (var m in lint)                                                    //遍历集合
    {
        Console.WriteLine(m.ToString());                                       //输出对象
    }
}
```



```

    }
    Console.ReadKey();
}

```

上述代码将数据源进行了投影操作，使用 **Select** 进行投影操作非常简单，其作用同 **SQL** 语句中的 **SELECT** 语句十分相似，上述代码将集合中的元素进行投影并将符合条件的元素投影到新的集合中 **lint** 去。

## 2. SelectMany 多重选择子句

**SelectMany** 和 **Select** 的用法基本相同，但是 **SelectMany** 与 **Select** 相比可以选择多个序列进行投影，示例代码如下所示。

```

static void Main(string[] args)
{
    int[] inter = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };           //创建数组
    int[] inter2 = { 21, 22, 23, 24, 25, 26};                                   //创建数组
    List<int[]> list = new List<int[]>();                                       //创建 List
    list.Add(inter);                                                            //添加对象
    list.Add(inter2);                                                            //添加对象
    var lint = list.SelectMany(i => i);                                         //SelectMany 操作
    foreach (var m in lint)                                                     //遍历集合
    {
        Console.WriteLine(m.ToString());                                       //输出对象
    }
    Console.ReadKey();
}

```

上述代码通过 **SelectMany** 方法将不同的数据源投影到一个新的集合中，运行结果如图 21-17 所示。

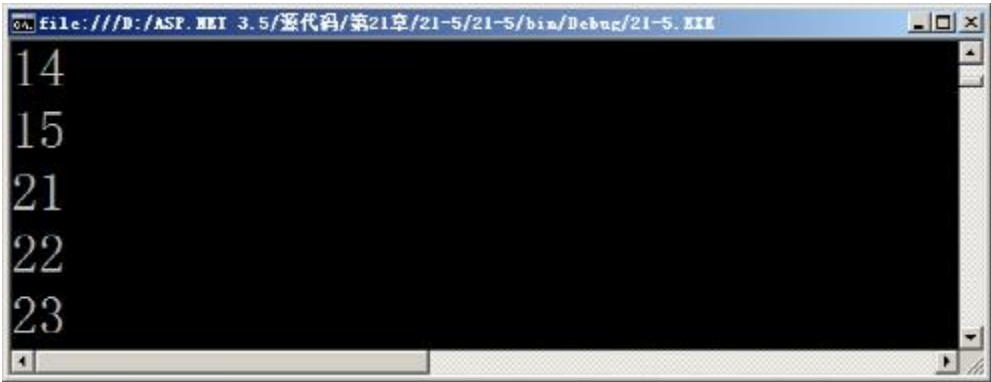


图 21-17 SelectMany 投影操作

### 21.4.3 筛选操作

筛选操作使用的是 **Where** 方法，其使用方法同 **LINQ** 查询语句中的 **where** 子句使用方法基本相同，筛选操作用于筛选符合特定逻辑规范的集合的元素，示例代码如下所示。

```

public static void WhereQuery()
{
    int[] inter = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };           //创建数组
    var lint = inter.Where(i => i > 5);                                           //使用 where 进行筛选操作
    foreach (var m in lint)                                                       //遍历集合
    {
        Console.WriteLine(m.ToString());                                       //输出对象
    }
    Console.ReadKey();
}

```

上述代码通过 **Where** 方法和 **Lambda** 表达式实现了对数据源中数据的筛选操作，其中 **Lambda** 表达式筛选了现有集合中所有值大于 **5** 的元素并填充到新的集合中，使用 **LINQ** 查询语句的子查询语句同样能够实现这样的功能，示例代码如下所示。

```
var lint = from i in inter where i > 5 select i; //执行筛选操作
```

上述代码同样实现了 LINQ 中的筛选操作 **Where**，但是使用筛选操作的代码更加简洁，上述代码运行后如图 21-18 所示。

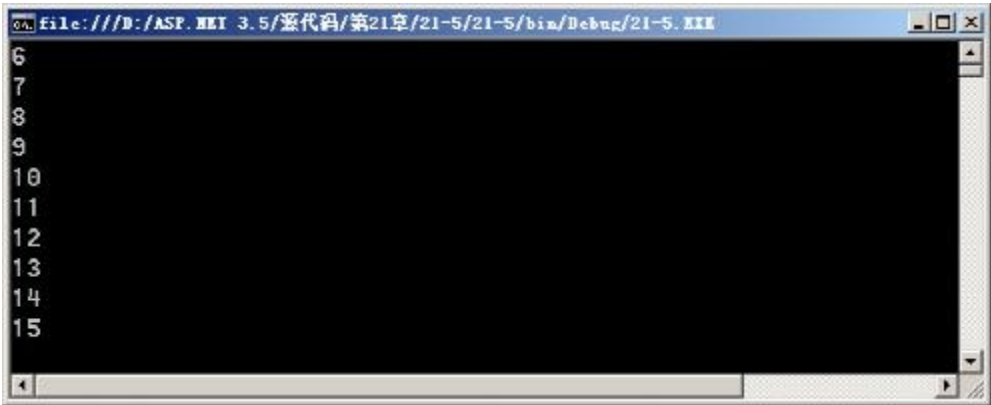


图 21-18 筛选操作

21.4.4 排序操作

排序操作最常使用的是 **OrderBy** 方法，其使用方法同 LINQ 查询子句中的 **orderby** 子句基本类似，使用 **OrderBy** 方法能够对集合中的元素进行排序，同样 **OrderBy** 方法能够针对多个参数进行排序。排序操作不仅提供了 **OrderBy** 方法，还提供了其他的方法进行高级排序，这些方法包括：

- ❑ **OrderBy** 方法：根据关键字对集合中的元素按升序排列。
- ❑ **OrderByDescending** 方法：根据关键字对集合中的元素按倒序排列。
- ❑ **ThenBy** 方法：根据次要关键字对序列中的元素按升序排列。
- ❑ **ThenByDescending** 方法：根据次要关键字对序列中的元素按倒序排列。
- ❑ **Reverse** 方法：将序列中的元素的顺序进行反转。

使用 LINQ 提供的排序操作能够方便的进行排序，示例代码如下所示。

```
public static void OrderByQuery()
{
    int[] inter = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 }; //创建数组
    var lint = inter.OrderByDescending(i => i); //使用倒序方法
    foreach (var m in lint) //遍历集合
    {
        Console.WriteLine(m.ToString()); //输出对象
    }
    Console.ReadKey();
}
```

上述代码使用了 **OrderByDescending** 方法将数据源中的数据进行倒排，除此之外，还可以使用 **Reverse** 将集合内的元素进行反转，示例代码如下所示。

```
public static void OrderByQuery()
{
    int[] inter = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 }; //创建数组
    var lint = inter.Reverse(); //反转集合
    foreach (var m in lint) //遍历集合
    {
        Console.WriteLine(m.ToString()); //输出对象
    }
    Console.ReadKey();
}
```

上述代码使用了 **Reverse** 元素将集合内的元素进行反转，运行结果如图 21-19 所示。

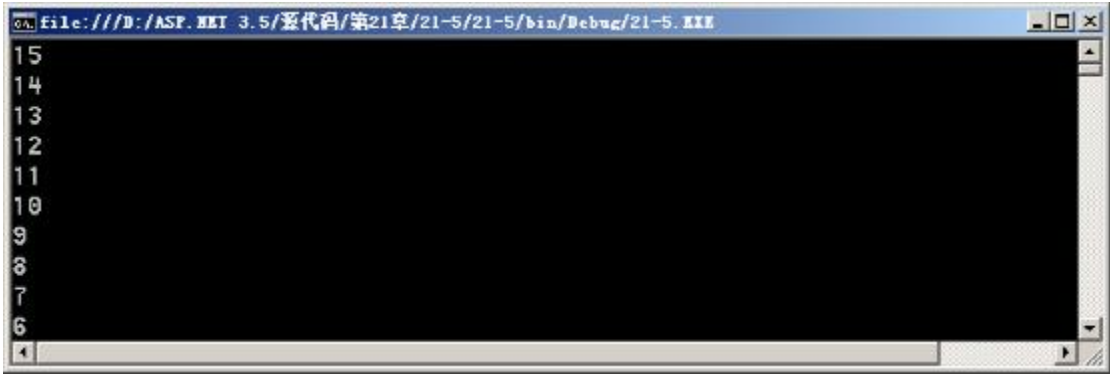


图 21-19 排序操作

注意：排序和反转并不相同，排序是将集合中的元素进行排序，可以是正序也可以是倒序，而反转并没有进行排序，只是讲集合中的元素从第一个放到最后一个，依次反转而已。

21.4.5 聚合操作

在 SQL 中，往往需要统计一些基本信息，例如今天有多少人留言，今天有多少人访问过网站，这些都可以通过 SQL 语句进行查询。在 SQL 查询语句中，支持一些能够进行基本运算的函数，这些函数包括 **Max**、**Min** 等。在 LINQ 中，同样包括这些函数，用来获取集合中的最大值和最小值等一些常用的统计信息，在 LINQ 中，这种操作被称为聚合操作。聚合操作常用的方法有：

- ❑ **Count** 方法：获取集合中元素的数量，或者获取满足条件的元素数量。
- ❑ **Sum** 方法：获取集合中元素的总和。
- ❑ **Max** 方法：获取集合中元素的最大值。
- ❑ **Min** 方法：获取集合中元素的最小值。
- ❑ **Average** 方法：获取集合中元素的平均值。
- ❑ **Aggregate** 方法：对集合中的元素进行自定义的聚合计算。
- ❑ **LongCount** 方法：获取集合中元素的数量，或者计算序列满足一定条件的元素的数量。一般计算大型集合中的元素的数量。

1. **Max**、**Min**、**Count**、**Average** 内置方法

通过 LINQ 提供的聚合操作的方法能够快速获取统计信息，如要找到数据源中数据的最大值，可以使用 **Max** 方法，示例代码如下所示。

```
public static void CountQuery()
{
    int[] inter = { 20, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };           //创建数组
    var Maxlint = inter.Max();                                                       //获取最大值
    var Minlint = inter.Min();                                                       //获取最小值
    Console.WriteLine("最大值是" + Maxlint.ToString());                             //输出最大值
    Console.WriteLine("最小值是" + Minlint.ToString());                             //输出最小值
    Console.ReadKey();
}
```

上述代码在获取最大值和最小值时并没有使用 **Lambda** 表达式，因为数据源中并没有复杂的对象，所以可以默认不使用 **Lambda** 表达式就能够返回相应的值，如果要编写 **Lambda** 表达式，可以编写相应代码如下所示。

```
var Maxlint = inter.Max(i => i);                                                     //获取最大值
var Minlint = inter.Min(i => i);                                                     //获取最小值
```

聚合操作还能够获取平均值和获取集合中元素的数量，示例代码如下所示。

```
public static void CountQuery2()
{
```

```
int[] inter = { 20, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };           //创建数组
var Countlint = inter.Count(i => i > 5);                                       //获取元素数量
var Arrlint = inter.Average(i => i);                                           //获取平均值
Console.WriteLine("复合条件的集合有" + Countlint.ToString()+"项");           //输出项数
Console.WriteLine("平均值为" + Arrlint.ToString());                           //输出平均值
Console.ReadKey();
}
```

上述代码通过 **Count** 方法获得符合相应条件的元素的数量，并通过 **Average** 方法获取平均值，运行后如图 21-20 所示。

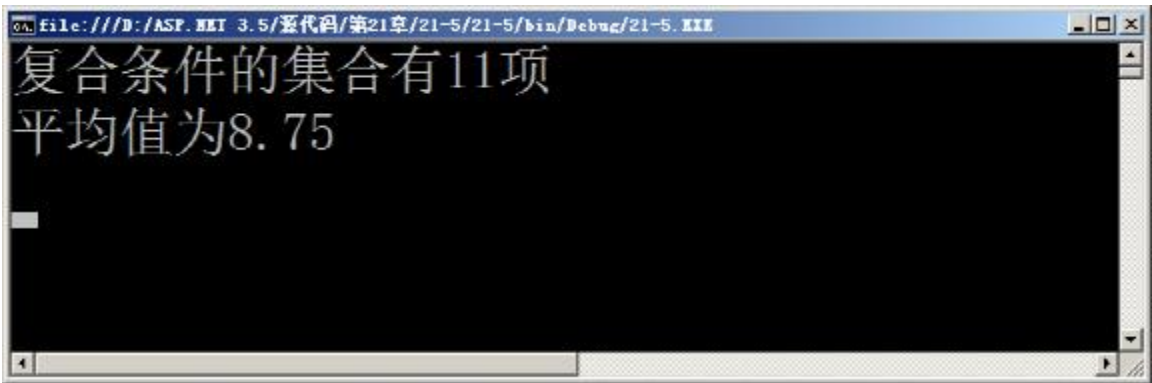


图 21-20 Count 和 Average 方法

在编写查询操作时，可以通过编写条件来规范查询范围，例如上述代码使用 **Count** 的条件就编写了 **i => i > 5** 的 **Lambda** 表达式，该表达式会返回符合该条件的集合再进行方法运算。

2. Aggregate 聚合方法

**Aggregate** 方法能够对集合中的元素进行自定义的聚合计算，开发人员能够使用 **Aggregate** 方法实现类似 **Sum**、**Count** 等聚合计算，示例代码如下所示。

```
public static void AggregateQuery()
{
    int[] inter = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };           //创建数组
    var aq = inter.Aggregate((x,y)=>x+y);                                       //使用 Aggregate 方法
    Console.WriteLine(aq.ToString());                                           //实现 Sum 方法
    Console.ReadKey();
}
```

上述代码通过编写 **Lambda** 表达式实现了数据源中所有数据的加法，也就是实现了 **Sum** 聚合方法，运行后如图 21-21 所示。

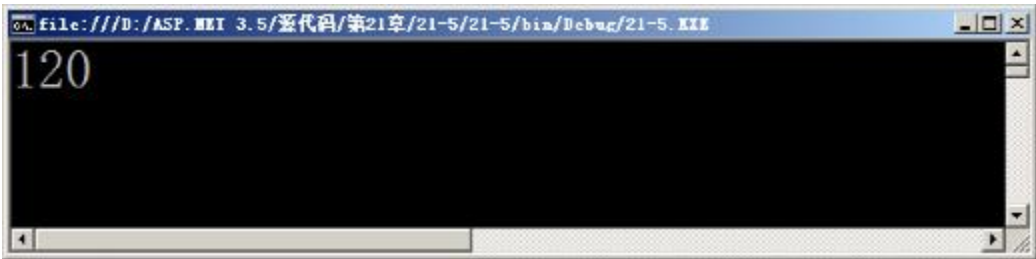


图 21-21 自定义聚合操作

**LINQ** 不仅仅包括这些查询操作方法，**LINQ** 还包括集合操作，删除集合中重复的元素，也能够计算集合与集合之间的并集差集等。**LINQ** 查询操作不仅提供了最基本的投影、筛选、聚合等操作，还能够极大的简化集合的开发，实现集合和集合中元素的操作。

21.5 使用 LINQ 查询和操作数据库

讲解了关于 **LINQ** 的基本知识，就需要使用 **LINQ** 进行数据库操作，**LINQ** 能够支持多个数据库并为每种数据库提供了便捷的访问和筛选方案，本书主要使用 **SQL Server 2005** 作为数据源进行 **LINQ** 查询和操作



数据示例数据库。

21.5.1 简单查询

LINQ 提供了快速查询数据库的方法，这个方法非常的简单，在前面的章节中已经讲到，这里使用 21.1.1 中准备的 **student** 数据库作为数据源，其表结构和数据都已经创建完毕，只需要进行简单查询即可。首先创建一个 LINQ to SQL 文件，名称为 **MyLinq.dbml**，并将需要查询的表拖动到视图中，这里需要拖动 **Class** 表和 **Student** 表作为数据源，如图 21-22 所示。



图 21-22 数据库关系图

创建了文件并拖动了相应的数据库关系图后，就可以保存并编写相应的代码进行查询了，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    MyLinqDataContext dc = new MyLinqDataContext();           //创建对象
    var StudentList = from d in dc.Student orderby d.S_ID descending select d; //执行查询
    foreach (var stu in StudentList)                          //遍历元素
    {
        Response.Write("学生姓名为" + stu.S_NAME.ToString()+"<br/>"); //输出 HTML 字符串
    }
}
```

上述代码直接使用 LINQ to SQL 文件提供的类进行数据查询和筛选，运行后如图 21-23 所示。

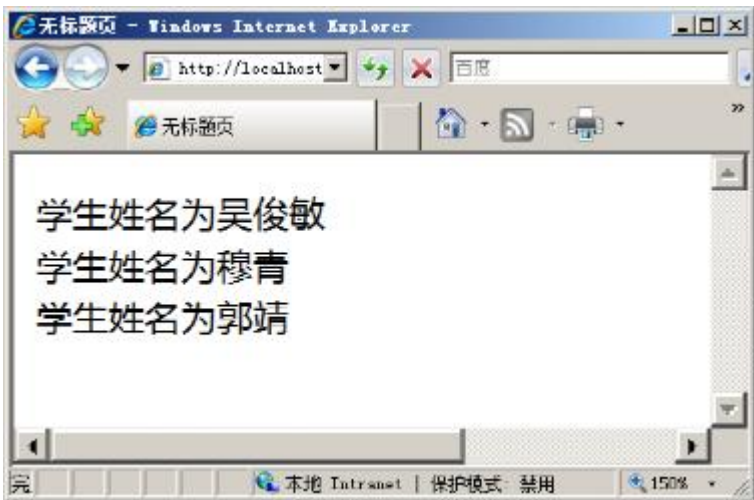


图 21-23 简单查询

查询的原理很简单，在 21.1.1 中就已经讲解了如何创建 LINQ 的 Web 应用，但是那个时候并没有涉及到 LINQ 查询子句，现在回过头再看就会发现其实使用 LINQ 进行数据库访问也并不困难，这里不再作过多解释。

## 21.5.2 建立连接

上一节中讲解了使用 LINQ 快速的建立数据库之间的连接。在 LINQ to SQL 中，.NET Framework 同样像 ADO.NET 一样为 LINQ 提供了 LINQ 数据库连接类和枚举用于自定义数据连接。建立与 SQL 数据库的连接，就需要使用 DataContext 类，示例代码如下所示。

```
DataContext db = new DataContext("Data Source=(local);
Initial Catalog=student;Persist Security Info=True;User ID=sa;Password=sa");//建立连接
```

上述代码通过 DataContext 类进行数据连接。当数据库连接后，就可以获取数据库相应的表显示数据，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    DataContext db = new DataContext("Data Source=(local);
Initial Catalog=student;Persist Security Info=True;User ID=sa;Password=sa");//建立连接
    try
    {
        Table<Student> stu = db.GetTable<Student>(); //获取相应表的数据
        var StudentList = from d in stu orderby d.S_ID descending select d; //执行 LINQ 查询
        foreach (var stud in StudentList) //遍历集合
        {
            Response.Write("学生姓名为" + stud.S_NAME.ToString() + "<br/>");//输出对象
        }
    }
    catch
    {
        Response.Write("数据库连接失败"); //抛出异常
    }
}
```

上述代码使用 DataContext 类进行了数据库连接的建立，建立连接后可以使用 Table 类获取数据库中的表并填充数据到表里面，这样就无需像 ADO.NET 一样首先建立连接、然后再填充数据集这样进行繁冗的数据操作。开发人员可以直接使用 LINQ 查询语句对数据进行筛选。

## 21.5.3 插入数据

创建了 DataContext 类对象之后，就能够使用 DataContext 的方法进行数据插入、更新和删除操作。相比 ADO.NET，使用 DataContext 对象进行数据库操作更加方便和简单。使用 LINQ to SQL 类进行数据插入的操作步骤如下。

- ❑ 创建一个包含要提交的列数据的新对象。
- ❑ 将这个新对象添加到与数据库中的目标表关联的 LINQ to SQL Table 集合。
- ❑ 将更改提交到数据库。

上面三个步骤就能够实现数据的插入操作，对数据库的连接可以使用 LINQ to SQL 类文件或者自己创建连接字符串。示例代码如下所示。

```
public void InsertSQL()
{
    Student stu = new Student { S_NAME="xixi",C_ID=1,S_CLASS="0502" }; //创建一个数据对象
    MyLinqDataContext dc = new MyLinqDataContext(); //创建一个数据连接
    dc.Student.InsertOnSubmit(stu); //执行插入数据操作
    dc.SubmitChanges(); //执行更新操作
}
```

上述代码使用了前面创建的 LINQ to SQL 类文件 MyLinq.dbml，使用该类文件快速的创建一个连接。在 LINQ 中，LINQ 模型将关系型数据库模型转换成一种面向对象的编程模型，开发人员可以创建一个数据

对象并为数据对象中的字段赋值，再通过 **LINQ to SQL** 类执行 **InsertOnsubmit** 方法进行数据插入就可以完成数据插入，运行后如图 21-24 所示。

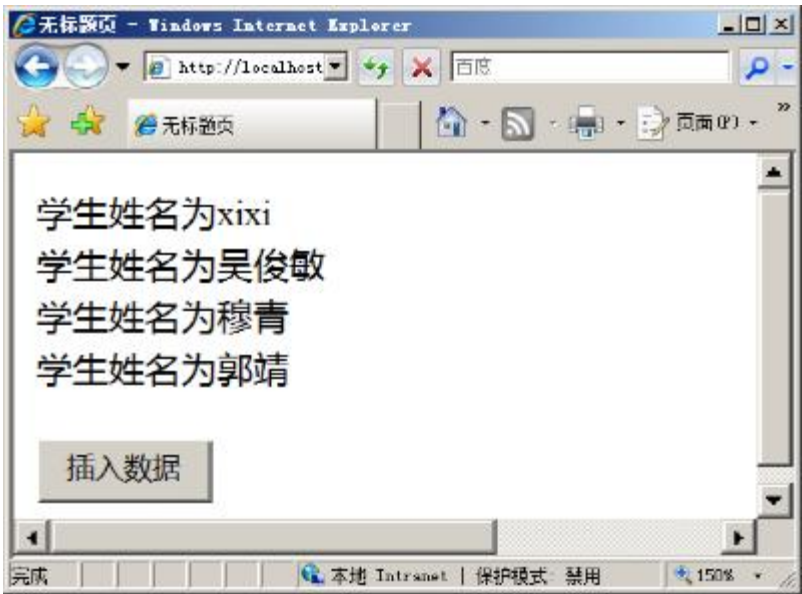


图 21-24 插入数据

使用 **LINQ** 进行数据插入比 **ADO.NET** 操作数据库使用的代码更少，而其思想更贴近了面向对象的概念。

21.5.4 修改数据

**LINQ** 对数据库的修改也是非常的简便的，执行数据库中数据的更新的基本步骤如下所示。

- ❑ 查询数据库中要更新的行。
- ❑ 对得到的 **LINQ to SQL** 对象中的成员值进行所需的更改。
- ❑ 将更改提交到数据库。

上面三个步骤就能够实现数据的修改更新，示例代码如下所示。

```
public void UpdateSQL()
{
    MyLinqDataContext dc = new MyLinqDataContext();
    var element = from d in dc.Student where d.S_ID == 4 select d;           //查询
    foreach (var e in element)                                             //遍历集合
    {
        e.S_NAME = "xixi2";                                               //修改值
        e.S_CLASS = "0501";                                               //修改值
    }
    dc.SubmitChanges();                                                    //更新
}
```

在修改数据库中一条数据之前，必须要查询出这个数据。查询可以使用 **LINQ** 查询语句和 **where** 子句进行筛选查询，也可以使用 **Where** 方法进行筛选查询。筛选查询出数据之后，就能够修改相应的值并使用 **SubmitChanges()** 方法进行数据更新，运行后如图 21-25 所示。

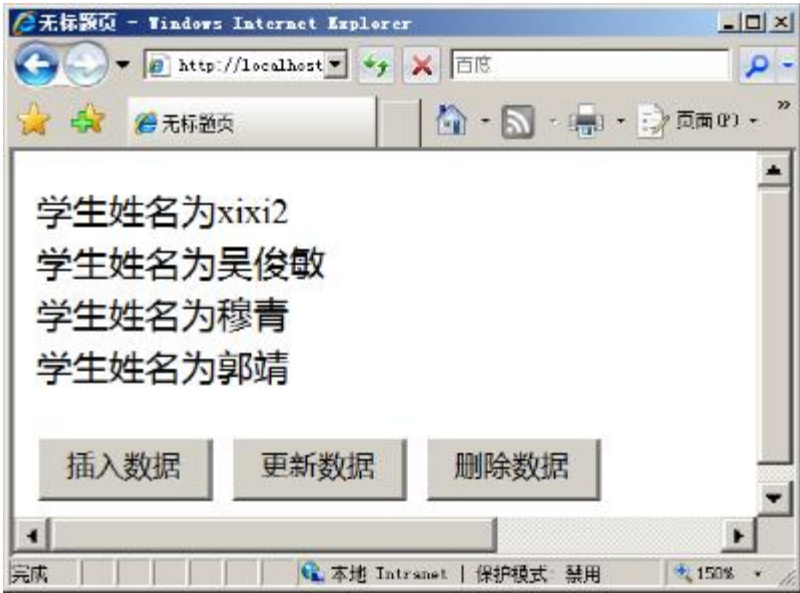


图 21-25 更新 xixi 为 xixi2

21.5.5 删除数据

使用 LINQ 能够快速的删除行，删除行的基本步骤如下所示。

- ❑ 在数据库的外键约束中设置 **ON DELETE CASCADE** 规则。
- ❑ 使用自己的代码首先删除阻止删除父对象的子对象。

只需要上面两个步骤就能够实现数据的删除，示例代码如下所示。

```
public void DeleteSQL()
{
    MyLinqDataContext dc = new MyLinqDataContext();           //连接数据源
    var del = from d in dc.Student where d.S_ID == 4 select d;  //查询要删除的行
    foreach (var e in del)                                     //遍历集合
    {
        dc.Student.DeleteOnSubmit(e);                         //执行删除操作
    }
}
```

上述代码使用 LINQ 执行了数据库删除操作，运行后如图 21-26 所示。

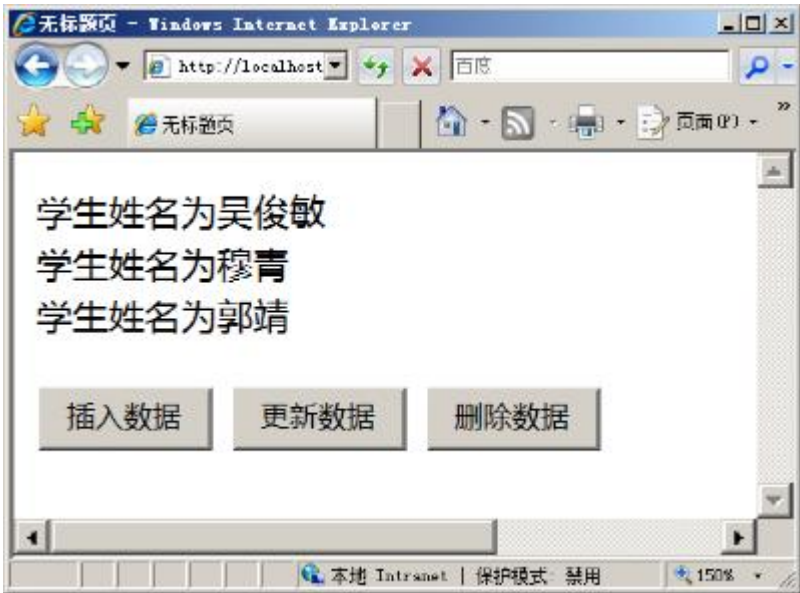


图 21-26 删除数据

在进行数据中表的删除过程时，有些情况需要判断数据库中表与表之间是否包含约束关系，如果包含了子项，首先必须删除子项否则不能删除父项。例如在删除 **Class** 表时，在 **Student** 表中有很多项都包含 **Class** 表的元素，例如 **C\_ID** 等于 **1** 的元素，当要删除 **Class** 表中 **C\_ID** 为 **1** 的元素时，就需要先删除 **Student** 表中包含 **C\_ID** 为 **1** 的元素，以保持数据库约束，示例代码如下所示。



```
public void DeleteSQL()
{
    MyLinqDataContext dc = new MyLinqDataContext();
    var delf = from d in dc.Class where d.C_ID == 1 select d;           //查询父表
    var del = from d in dc.Student from f in dc.Class where d.S_ID ==
    4 && f.C_ID==1&&d.S_ID==f.C_ID select d;                           //进行约束查询
    foreach (var e in del)                                             //删除子表
    {
        dc.Student.DeleteOnSubmit(e);                                //删除对象
        dc.SubmitChanges();                                           //更新删除
    }
    foreach (var f in delf)                                           //删除父表
    {
        dc.Class.DeleteOnSubmit(f);                                  //删除对象
        dc.SubmitChanges();                                           //更新删除
    }
}
```

当数据库包含外键，以及其他约束条件时，在执行删除操作时必须小心进行，否则会破坏数据库约束，也有可能抛出异常。

## 21.6 LINQ 与 MVC

在 **ASP.NET MVC** 应用程序中，**Models** 层通常用于抽象数据库中的表使之成为开发人员能够方便操作的对象，在 **Models** 层中，开发人员能够使用 **LINQ** 进行数据库的抽象并通过 **LINQ** 筛选和查询数据库中的数据用于页面呈现。

### 21.6.1 创建 ASP.NET MVC 应用程序

在前面的章节中讲到了 **ASP.NET MVC** 开发模型，在 **ASP.NET MVC** 应用程序中，开发人员能够很好的将页面进行分离，这样不同的开发人员就能够只关注自身的开发而无需进行页面整合。在 **ASP.NET MVC** 应用程序中，包括三个基本的模块，这三个模块分别是 **Models**、**Controllers** 和 **Views**。

**Controllers** 用于实现与 **Models** 的交互和 **Views** 的交互，在 **Controllers** 与 **Models** 交互时，**Controllers** 主要是用于从 **Models** 中进行数据的获取，而 **Models** 主要关注与数据库进行交互，在 **Controllers** 与 **Views** 交互时，**Controllers** 中的方法同 **Views** 中的页面一起用于页面呈现。在进行 **ASP.NET MVC** 应用程序的开发时，在应用程序中读取数据库则需要在 **Models** 中创建 **LINQ** 文件与数据进行交互，在创建 **LINQ** 文件时，首先需要创建 **ASP.NET MVC** 应用程序，如图 21-27 所示。

单击【确定】按钮创建 **ASP.NET MVC** 应用程序，系统会默认创建若干文件和文件夹，删除 **Views** 和 **Controllers** 文件夹下的文件，自行创建页面进行 **ASP.NET MVC** 应用程序的开发。首先创建 **Views** 文件，在创建 **Views** 文件时首先需要创建一个文件夹，这里创建一个 **Blog** 文件夹，创建后在该文件夹内创建 **Views** 文件 **Index.aspx**，如图 21-28 所示。



图 21-27 创建 ASP.NET MVC 应用程序



图 21-28 创建 Views 文件

这里创建一个 **Index.aspx** 的 **Views** 文件用于页面呈现，**Index.aspx** 页面代码如下所示。

```
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title><%=ViewData["title"]%></title>
</head>
<body>
    <div>
        数据库中的数据为:<br/>
        <%=ViewData["contents"]%>
    </div>
</body>
</html>
```

上述代码在 **Index.aspx** 中使用了两个 **ViewData**，这两个 **ViewData** 分别用于呈现标题和数据内容。在 **Views** 文件中，需要通过 **Controllers** 文件进行 **ViewData** 变量的获取，这里还需要创建一个 **Controllers** 文件。创建该文件时，应该与相应 **Views** 页面的文件夹同名，并在名称后加入 **Controllers.cs**，如图 21-29 所示。

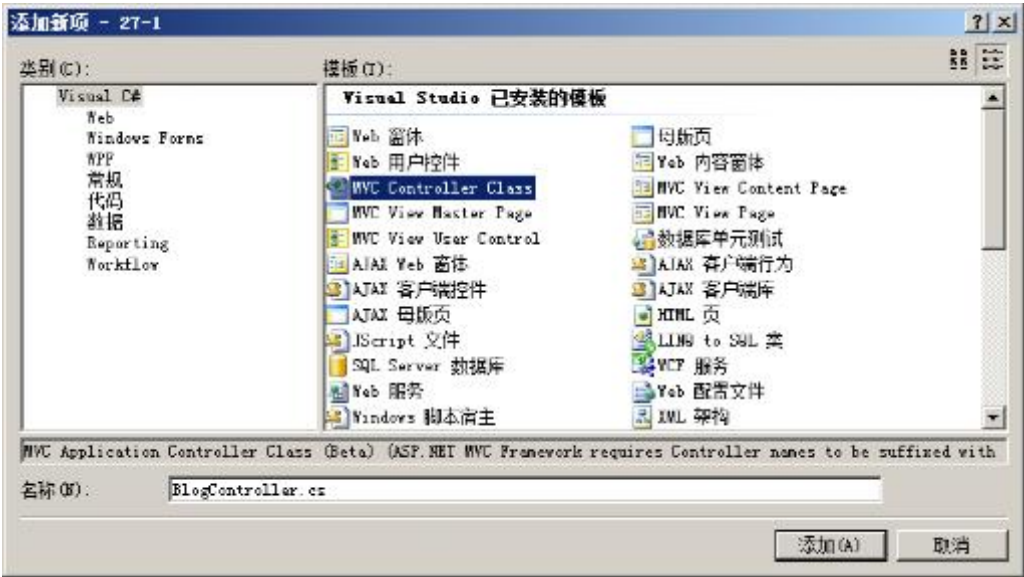


图 21-29 创建 Controllers 文件

由于创建的文件是 **Blog**，所以 **Controllers** 文件的名称应该为 **BlogControllers.cs**，创建完成后，为了让用户能够访问 **Index.aspx**，还需要实现 **Index** 方法，示例代码如下所示。

```
namespace _27_1.Controllers
{
    public class BlogController : Controller //继承自 Controller
    {
        public ActionResult Index() //实现方法
        {
            ViewData["title"] = "MVC 和 LINQ"; // ViewData["title"]
            ViewData["contents"] = "数据内容"; //ViewData["contents"]
            return View(); //返回视图
        }
    }
}
```

```
}  
}
```

上述代码分别为 ASP.NET MVC 应用程序添加了两个 ViewData，这两个 ViewData 分别用于呈现标题和数据内容，在 Views 相应的文件中能够使用这两个 ViewData 进行数据呈现。

21.6.2 创建 LINQ to SQL

在创建了 ASP.NET MVC 应用程序后并创建了相应的 Views 和 Controllers 文件用于页面呈现和数据获取，在 Controllers 中 ViewData[“content”]是获取数据库中的数据，这里可以在 Models 中创建 LINQ to SQL 类进行数据辅助操作，如图 21-30 所示。



图 21-30 创建 LINQ to SQL 类

创建完成后就能够在服务器资源管理器中拖动相应的表进行 LINQ to SQL 类的创建，如图 21-31 和图 21-32 所示。



图 21-31 服务器资源管理器



图 21-32 LINQ to SQL 类文件

添加完成并配置 LINQ to SQL 类后，还需要在 Models 中创建相应的类文件，示例代码如下所示。

```
namespace _27_1.Models  
{  
    public class GetData  
    {  
        public string build()  
        {  
            //实现方法  
        }  
    }  
}
```

```
string build="";
DataClasses1DataContext dcd = new DataClasses1DataContext();
var d = from dc in dcd.mynews select dc;
foreach (var myd in d)
{
    build += myd.TITLE.ToString()+"<br/>";
}
return build;
}
}
```

//返回字符串  
//使用 LINQ 类  
//执行查询  
//遍历集合  
//输出字符串

上述类文件在 **Models** 中进行数据查询并返回相应的数据，在 **Controllers** 中，开发人员可以使用该类进行数据查询和操作。

注意：在 **Controllers** 中需要使用 **Models** 命名空间，这也就能够直接在 **Controllers** 中进行 LINQ 数据查询和操作，但是为了层次分明和简便，推荐在 **Models** 中进行相应的数据操作类文件编写。

21.6.3 数据查询

在创建了相应的数据操作类后，就能够在 **Controllers** 中查询数据并将数据呈现在页面中，**Controllers** 中 **Index** 方法需要从数据库中进行数据读取，示例代码如下所示。

```
using _27_1.Models;
namespace _27_1.Controllers
{
    public class BlogController : Controller
    {
        public ActionResult Index()
        {
            ViewData["title"] = "MVC 和 LINQ";
            GetData da=new GetData();
            ViewData["contents"]=da.build();
            return View();
        }
    }
}
```

//使用 Model 命名空间  
//继承自 Controller  
//实现方法  
//定义 ViewData  
//创建对象  
//赋值给 ViewData  
//返回默认视图

上述代码则使用了 **Models** 中的类进行数据呈现，在创建对象后，可以使用对象中的 **build** 方法进行数据获取。在运行前，还需要修改 **URL Routing** 的默认值进行默认页面的呈现，示例代码如下所示。

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
    routes.MapRoute(
        "Default",
        "{controller}/{action}/{id}",
        new { controller = "Blog", action = "Index", id = "" }
    );
}
```

//编写路由规则  
//编写默认值

上述代码编写了路由规则和默认值，当用户访问网站时，如果 **Controllers** 和方法都没有指定，则会访问 **Blog/Index** 方法。修改路由规则和默认值后，就能够运行 ASP.NET MVC 应用程序，运行后如图 21-33 所示。



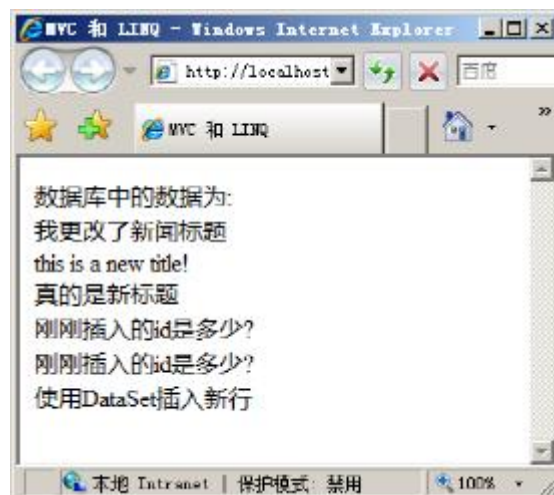


图 21-33 运行 MVC 应用程序

## 21.7 小结

**LINQ**是.NET 3.5框架里的新特性,使用**LINQ**能够极大的方便开发人员进行数据操作。不仅如此,**LINQ**还支持多种数据源中数据的筛选和查询,这些数据源可能是数组、数据库、数据集甚至是**XML**文档。本章着重的讲解了**LINQ**查询语法,以及**LINQ**查询子句,可以由浅入深的了解**LINQ**查询语句是如何编写的。**LINQ**查询语句的语法非常简单,熟悉**SQL**查询语法的人在一定程度上很容易就能够上手投入到开发中。本章还包括:

- ❑ 准备数据源: 准备了 3 种类型的数据源以演示如何使用**LINQ**进行多种数据源查询。
- ❑ 基本子句: 讲解了 **from**、**where**、**let**、**join** 等基本子句,基本子句在**LINQ**中是非常基础也是非常重要的,熟练的编写基本子句不仅能够提高性能也能够方便筛选。
- ❑ 投影操作: 讲解了如何使用**LINQ**提供的 **Select** 方法进行投影操作。
- ❑ 排序操作: 讲解了如何使用**LINQ**提供的 **Where** 方法进行排序操作。
- ❑ 聚合操作: 讲解了如何使用**LINQ**提供的 **Sum**、**Count** 等方法进行聚合操作。
- ❑ 建立连接: 讲解了如何不使用**LINQ to SQL**提供的类而使用方法建立与**SQL**数据源的连接。
- ❑ 数据操作: 讲解了如何使用**LINQ**进行数据插入、修改和删除等操作。

本章在最后几节中详细的讲解了如何使用**LINQ**进行数据插入、修改和删除等操作以及对比了**LINQ**与**ADO.NET**的优劣,使用**LINQ**能够减少数据操作的代码量,使代码更像是使用面向对象的思想进行开发的,并再与**ASP.NET MVC**应用程序进行应用整合。**LINQ**技术现在在国内是一门新技术,但是发展也有一定的时间了,熟练掌握**LINQ**基础能够在未来的开发潮流中占有一席之地。

## 第七篇 ASP.NET 3.5 模块开发

第 22 章 注册模块设计

第 23 章 登录模块设计

第 24 章 广告模块设计

第 25 章 新闻模块设计

第 26 章 投票模块设计

第 27 章 聊天模块设计

## 第 22 章 注册模块设计

注册模块在网站开发中是一个必不可少的模块，注册模块让用户能够在网站上注册自己的信息，以便在以后的访问中可以直接登录，网站也可以通过注册模块保存用户信息，让用户能够在网站上随时查阅自己的信息和聚合内容。

### 22.1 学习要点

注册模块需要涉及到一些 **ASP.NET 3.5** 的基本知识，如果要仔细学习注册模块的开发，需要详细了解本书的一些章节知识，这些章节如下所示：

- ❑ **ASP.NET** 的网页代码模型。
- ❑ **Web** 窗体基本控件。
- ❑ 数据库基础。
- ❑ **ADO.NET** 常用对象。
- ❑ **Web** 窗体数据控件。

基本了解了以上章节的知识点后，就能够熟练学习和开发此模块。

### 22.2 系统设计

在进行系统开发时，无论是模块开发还是整体规划，都需要进行系统设计，系统设计不仅能够方便开发人员的系统开发，同样也节约了在后期维护中所需的时间和成本。系统设计就好像是一张软件制造计划书，通过计划书能够高效的进行软件开发和软件维护。

#### 22.2.1 模块功能描述

注册模块是网站中最常用也是必不可少的模块，对于注册模块的开发，首先需要确定一个基本的用户流程图，如图 **22-1** 所示。

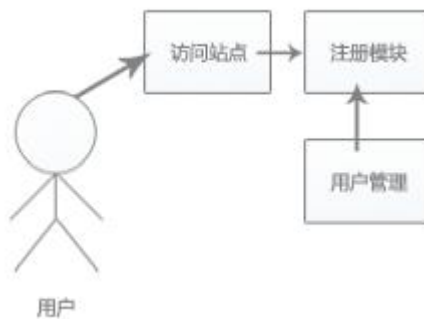


图 22-1 注册模块基本用户流程图

从注册模块的基本用户流程图可以看出，用户进行注册这个动作非常的简单。首先用户需要访问网站，

访问网站后就会选择是否进行注册，如果需要注册则网站提供一个注册模块给用户，用户就能够进行注册。在用户完成注册后，用户信息还应该被管理员管理，管理员能够通过用户管理页面进行页面管理。从上述用户流程图可以基本规划以下几个页面：

- ❑ 注册页面：提供用用户注册操作。
- ❑ 管理页面：提供管理员管理页面。

在基本规划了 **Web** 应用中需要制作的模块，可以为这些模块进行模块的流程分析。

## 22.2.2 模块流程分析

在对业务进行了基本的划分之后，可以为模块进行基本的流程分析，包括这个模块中最基本的函数，以及这些函数在页面中是如何执行的。

对于注册页面而言，首先需要确定用户需要提供哪些注册内容，如果 **Web** 应用希望用户提供真实的信息，例如校内网这样的 **SNS**，那么就需要用户提供真实的信息，以及提供应用程序验证用户的真实性。如果 **Web** 应用无所谓用户提供的信息是真实的或者是虚假的，那么就无所谓应用程序的开发，那么应用程序的开发就只需要进行入库即可。

对于管理页面而言，管理人员需要对用户信息进行操作，包括修改和删除。在 **ASP.NET 3.5** 中，可以使用 **SQL** 数据源控件和 **SQL** 数据绑定控件完成功能。既然了解了基本的模块流程和制作，就可以模拟模块流程分析图，如图 22-2 所示。

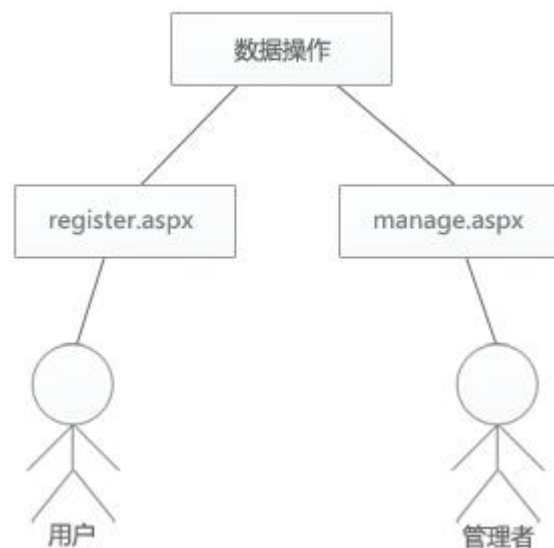


图 22-2 基本模块流程图

用户注册直接进入 **register.aspx** 页面进行注册，注册完成后进行数据操作，将用户信息加入到数据库中。管理人员进入 **manage.aspx** 对用户的注册信息管理进行数据操作即可。

## 22.3 数据库设计

数据库设计是软件设计中最为重要的一部分，当数据库的设计完成后，软件开发过程中如果对于数据库模型的更改则会引起很多的变动，如果对于数据库其中的一个字段的更改，很可能就需要将大部分代码中的 **SQL** 语句进行更改，良好的数据库设计是非常必要的。

### 22.3.1 数据库的分析和设计

用户在网站上进行登录，首先要确定对网站而言需要用户的哪些基本信息，这些基本信息可以暂时归



纳如下：

- ❑ 用户名：用于保存用户的用户名，当用户登录时可以通过用户名验证。
- ❑ 密码：用于保存用户的密码，当用户使用登录时可以通过密码验证。
- ❑ 性别：用于保存用户的性别。
- ❑ 头像：用于保存用户的个性头像。
- ❑ QQ/MSN：用于保存用户的 QQ/MSN 等信息。
- ❑ 个性签名：用于展现用户的个性签名等资料。
- ❑ 备注：用于保存用户的备注信息。
- ❑ 用户情况：用于保存用户的状态，可以设置为通过审批和未通过等。

对数据库的基本分析完成后，就可以创建数据库表来存储用户注册的信息。这里需要创建一个 **Register** 数据库，创建完成后就能够在 **Register** 数据库中创建表。

22.3.2 数据表的创建

创建表可以通过 **SQL Server Management Studio** 视图进行创建也可以通过 **SQL Server Management Studio** 查询使用 **SQL** 语句进行创建，本书两者都介绍。这个模块的数据库设计比较简单，为了保存用户信息，可以创建一个 **Register** 表并为数据库分析中的基本信息创建字段，如图 22-3 所示。

	列名	数据类型	允许空
🔑	id	int	<input type="checkbox"/>
	username	nvarchar(50)	<input checked="" type="checkbox"/>
	password	nvarchar(50)	<input checked="" type="checkbox"/>
▶	sex	int	<input checked="" type="checkbox"/>
	picture	nvarchar(MAX)	<input checked="" type="checkbox"/>
	IM	nvarchar(50)	<input checked="" type="checkbox"/>
	information	nvarchar(MAX)	<input checked="" type="checkbox"/>
	others	nvarchar(MAX)	<input checked="" type="checkbox"/>
	ifuser	int	<input checked="" type="checkbox"/>
			<input type="checkbox"/>

图 22-3 数据库表结构

正如图 22-3 中所示，表为用户的基本信息创建了字段，这些字段的意义分别为：

- ❑ **id**：用于标识用户的 **ID** 号，并为自动增长的主键。
- ❑ **username**：用于标识用户名。
- ❑ **password**：用于标识用户密码。
- ❑ **sex**：用于标识用户性别。
- ❑ **picture**：用于标识用户头像。
- ❑ **IM**：用于标识用户的 **IM** 信息，包括 **QQ/MSN** 等。
- ❑ **information**：用于标识用户的个性签名。
- ❑ **others**：用于标识用户的备注信息。
- ❑ **ifuser**：用于标识用户是否为合法用户。

创建数据表的 **SQL** 查询语句代码如下所示。

```
USE [Register]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Register](
    [id] [int] IDENTITY(1,1) NOT NULL,
    [username] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
    [password] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
    [sex] [int] NULL,
    [picture] [nvarchar](max) COLLATE Chinese_PRC_CI_AS NULL,
    [IM] [nvarchar](50) NULL,
    [information] [nvarchar](max) NULL,
    [others] [nvarchar](max) NULL,
    [ifuser] [int] NULL
) ON [PRIMARY]
```

//创建数据库

```
[IM] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
[information] [nvarchar](max) COLLATE Chinese_PRC_CI_AS NULL,
[others] [nvarchar](max) COLLATE Chinese_PRC_CI_AS NULL,
[ifisuser] [int] NULL,
CONSTRAINT [PK_Register] PRIMARY KEY CLUSTERED
(
    [id] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
```

上述代码创建了一个数据库并将 **ID** 设为自动增长的主键，在用户注册时，可以不向该字段进行数据操作。

22.4 界面设计

良好的界面设计是吸引用户的基本，在注册页面将页面设计的丰富多彩，可以吸引用户的注册和登录，并提高回头率。在进行页面设计时，可以使用 **CSS** 也可以使用表格进行页面布局，相比之下 **CSS** 具有更高的灵活性。

22.4.1 基本界面

在进行页面布局前，只需要创建一个基本页面以满足应用程序的需求即可。注册模块需要一些基本的控件，这些控件包括 **TextBox** 控件、**Label** 控件和按钮控件，示例代码见光盘中源代码\第 22 章\22-1\22-1\Default.aspx 所示。

上述代码创建了一个头部信息层、一个注册信息层和一个底部信息层，这三个层分别负责头部图片的显示、注册信息的样式控制和底部版权说明，在没有 **CSS** 控制时，其效果如图 22-4 所示。

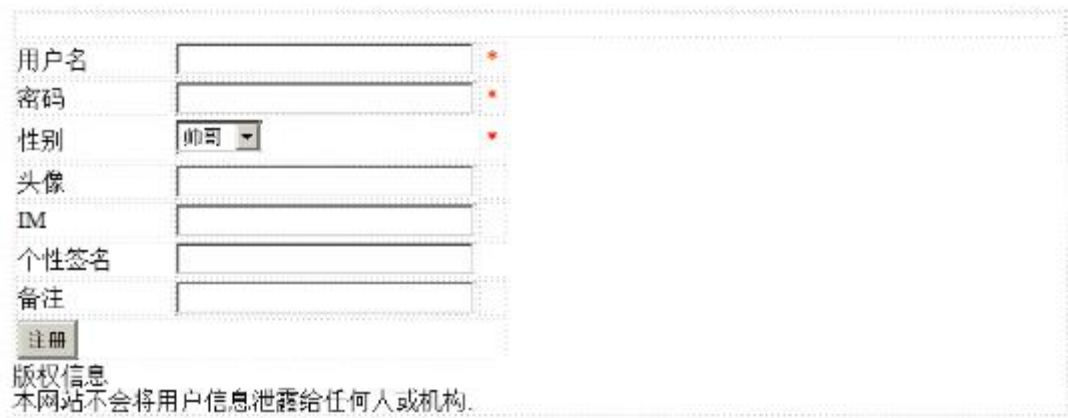


图 22-4 基本样式

在基本样式中，注册信息层使用表格进行排版，使用表格能够快速的进行页面的布局控制，表格同样可以使用 **CSS** 进行样式控制。

22.4.2 创建 CSS

使用 **CSS** 进行网页布局能够极大的加强网页布局的灵活度，同样在网页布局中也提高了代码的复用性并将 **HTML** 页面代码与 **CSS** 代码相分离，**CSS** 页面代码如下所示。

```
body //设置页面样式
{
```

```
font-size:12px;
font-family:Geneva, Arial, Helvetica, sans-serif;
margin:0px 0px 0px 0px;
}
.top //设置头部样式
{
    background:white url(top.png) no-repeat top center;
    height:200px;
    margin:0px auto;
    width:800px;
}
.register //设置注册样式
{
    margin:0px auto;
    width:800px;
}
.end //设置底部样式
{
    background:#f9fbfd;
    margin:0px auto;
    width:800px;
    text-align:center;
    padding:10px 10px 10px 10px;
}
```

在 CSS 页面文件样式编写完毕后，就需要在相应的页面进行引用，示例代码如下所示。

```
<link href="css.css" rel="stylesheet" type="text/css" />
```

在使用了 CSS 文件后，页面样式如图 22-5 所示。



图 22-5 CSS 样式控制后的页面

上述页面在 CSS 的样式控制下显得非常的友好，用户在进行注册时，会感觉到应用程序是在用心制作的情况下上线的，提高了用户的回头率。

## 22.5 代码实现

在完成基本的控件布局和 CSS 样式布局之后，页面就能够呈现在客户端浏览器中。但是如果用户想要在页面中执行逻辑操作，就需要进行代码实现完成应用程序所需要执行的页面逻辑，以保证用户注册功能

能够良好的运行。

22.5.1 验证控制

在用户进行注册操作时，需要对用户进行用户验证控制，例如用户没有输入密码的情况下单击了注册控件，数据是不应该被插入到数据库中的。如果没有对数据进行验证则会插入很多空数据，影响数据库功能。若要实现验证控制，可以使用现有的验证控件进行验证控制，示例代码见关盘中源代码\第 22 章\22-1\22-1\Default.aspx。

上述代码使用了 **RequiredFieldValidator** 控件进行了基本的验证，如果用户输入的用户名和密码以及性别为空，则会说明用户名和密码以及性别为空，请重新输入，如图 22-6 所示。



图 22-6 验证控制

进行验证控制后，就能够防止非法用户或用户疏忽所造成的空数据库问题，也方便了数据维护的进行。

22.5.2 过滤输入信息

在进行数据操作之前，并不能只凭用户输入的信息是否为空就能够判断用户是否是合法用户，在 Web 应用中包括很多的坏的信息，例如黄色淫秽名称或者是特殊的字符串，都有可能对网站造成危害。

注意：不仅仅是黄色淫秽的名称会对网站造成危害，特殊的字符串还有可能造成 SQL 注入等更大的危害。

在用户单击按钮控件时会执行数据插入操作，在数据插入之前就需要对信息进行过滤，示例代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    if (Check(TextBox1.Text) || Check(TextBox2.Text) || Check(TextBox4.Text) ||
        Check(TextBox5.Text) || Check(TextBox6.Text) || Check(TextBox7.Text))           //判断
    {
        Label8.Text = "用户信息中不能够包含特殊字符如<, >, ', //, \\等, 请审核";      //输出信息
    }
    else
    {
        //注册代码
    }
}
```

上述代码使用了 **Check** 函数对文本框控件进行了用户资料的判断，**Check** 函数的实现如下所示。

```
protected bool Check(string text)                                                    //判断实现
{
    ...
}
```



```

        if (text.Contains("<") || text.Contains(">") || text.Contains("\"") ||
            text.Contains("/") || text.Contains("\\"))           //检查字符串
        {
            return true;                                         //返回真
        }
        else
        {
            return false;                                       //返回假
        }
    }

```

**Check** 函数定义了基本的判断方式，如果文本框信息中包含“<”，“>”，“'”，“/”，“\”等字符串时，该方法将会返回 **true**，否则会返回 **false**。这也就是说，如果字符串中包含了这些字符，则会返回 **true**。在 **Button1\_Click** 函数中就会判断包含非法字符，并进行提示，否则会执行注册代码。对关键字的过滤是非常必要的，这样能够保证应用程序的完整性并提高应用程序健壮性，同时也对数据库中的完整性进行了保护。

## 22.5.3 插入注册信息

当用户单击按钮控件时，如果对用户进行了非空验证和关键字过滤后，就能够进行数据的插入，用户可以使用 **ADO.NET** 进行数据操作，示例代码如下所示。

```

protected void Button1_Click(object sender, EventArgs e)
{
    if (Check(TextBox1.Text) || Check(TextBox2.Text) || Check(TextBox4.Text) ||
        Check(TextBox5.Text) || Check(TextBox6.Text) || Check(TextBox7.Text)) //检查字符串
    {
        Label8.Text = "用户信息中不能够包含特殊字符如<, >, ', /, \等, 请审核"; //输出信息
    }
    else
    {
        try
        {
            SqlConnection con =
                new SqlConnection("server=(local);database='Register';uid='sa';pwd='sa'"); //建立连接
            con.Open(); //打开连接
            string strSql =
                "insert into register (username,password,sex,picture,im,information,others,ifisuser) values (" +
                TextBox1.Text + "," + TextBox2.Text + "," + DropDownList1.Text + "," +
                TextBox4.Text + "," + TextBox5.Text + "," + TextBox6.Text + "," + TextBox7.Text + ",0)";
            SqlCommand cmd = new SqlCommand(strsql, con); //创建执行
            cmd.ExecuteNonQuery(); //执行 SQL
            Label8.Text = "注册成功,请牢记您的信息"; //提示成功
        }
        catch
        {
            Label8.Text = "出现错误信息,请返回给管理员"; //抛出异常
        }
    }
}

```

上述代码通过 **ADO.NET** 实现了数据的插入，但是上述代码有一个缺点，如果用户注册了一个用户并且名称为 **abc**，当这个用户注销并再注册一个用户名称为 **abc** 时，如果依旧将数据插入到数据库则会出现错误。值得注意的是，这个错误并不是逻辑错误，但是这个错误会造成不同的用户可能登录了同一个用户信息并产生信息错误。为了避免这种情况的发生，在用户注册前首先需要执行判断，示例代码如下所示。

```

string check = "select * from register where username='" + TextBox1.Text + "'";

```

```
SqlDataAdapter da = new SqlDataAdapter(check,con);           //创建适配器
DataSet ds = new DataSet();                               //创建数据集
da.Fill(ds, "table");                                     //填充数据集
if (da.Fill(ds, "table") > 0)                             //判断同名
{
    Label8.Text = "注册失败,有相同用户名";               //输出信息
}
else
{
    SqlCommand cmd = new SqlCommand(strsql, con);         //创建执行对象
    cmd.ExecuteNonQuery();                               //执行 SQL
    Label8.Text = "注册成功,请牢记您的信息";              //输出成功
}
```

在用户注册时，首先从数据库查询出是否已经包含这个用户名的信息，如果包含则不允许用户注册，如果没有，则说明用户是一个新用户，可以进行注册。

22.5.4 管理员页面

管理员页面作为管理页面，其功能非常简单，只需要对数据进行删除和修改即可，无需进行任何的数据操作，使用 **ASP.NET** 本身的数据源控件和数据绑定控件就能够实现管理员页面的编写和制作。作为数据的呈现，可以使用 **GridView** 控件进行呈现，同时 **GridView** 控件还支持编辑和删除功能，示例代码见光盘中源代码\第 22 章\22-1\22-1\Manage.aspx 所示。

上述代码编写了 **GridView** 控件的样式并且为 **GridView** 控件配置了数据源，同时也配置 **GridView** 控件能够支持编辑和删除等操作，在数据源配置时，需要新建一个连接字符串，如图 22-7 所示。

建立连接字符串并保存连接字符串到 **Web.config** 文件中，单击【下一步】按钮，可以生成 **SQL** 语句，在生成 **SQL** 语句时，为了方便管理，管理员通常都是对最新注册用户进行管理，如图 22-8 所示。

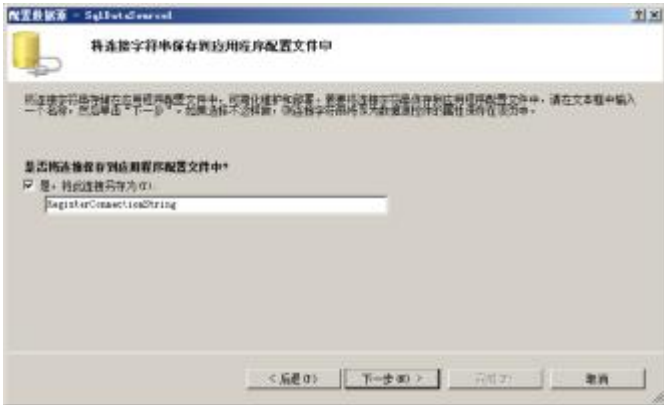


图 22-7 建立连接字符串

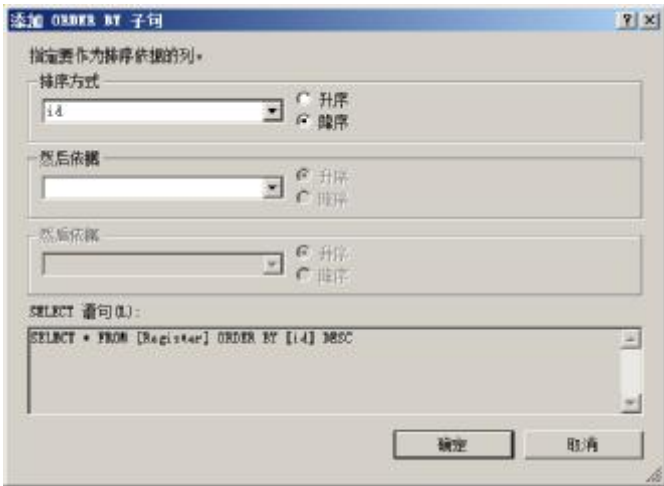


图 22-8 选择排序方式

选择按照 **id** 的方式进行倒序，能够让管理员快速的管理最新的注册用户，并进行编辑和删除等操作，为了能够让数据源自动支持编辑和删除操作，必须进行数据源高级配置，如图 22-9 所示。



图 22-9 生成数据操作语句

勾选【生成 INSERT、UPDATE 和 DELETE 语句】选项，以支持数据源控件自动进行编辑和删除等操作，单击【确定】按钮并完成，就将数据源控件配置完成，数据源控件配置后代码见光盘中源代码第 22 章 \22-1\22-1\Manage.aspx 所示。

从上述代码可以看出数据源控件中生成的 SQL 语句，使用数据源控件能够简化开发人员对数据的开发。

22.6 实例演示

编写完成页面代码和逻辑代码后，就可以进行注册和管理操作了，单击【运行】按钮运行模块，就能够实现用户的注册操作。用户在注册页面可以填写相应的用户注册信息，这些信息能够保存用户在相应网站上所需要的个人信息，同时网站还能够使用这些信息对用户数据进行归纳和整合，如图 22-10 所示。

用户可以在该页面填写用户信息，并进行注册。用户填写相关选项时，系统都为相关的选项进行了验证和关键字符的过滤，如果用户是一个非法用户，系统通过非法的方法（如 SQL 注入和字符注入）时，就能够验证当前用户注册方法是不正常的方法，就会提示用户注册中所填的项目是错误的。同样如果用户忘记填写相应的选项时，系统也会提示用户必须填写相应的项目，否则不予注册。

这种做法是基于 Web 应用的安全考虑的，不仅提高了网站的健壮性，也让数据库中避免了过多的非法信息，也保证了 Web 应用的其他模块能够正常的运行。当用户注册信息错误时，系统会提示信息错误，如图 22-11 所示。



图 22-10 注册模块运行示例



图 22-11 注册信息异常

如果用户注册成功，开发人员能够控制用户跳转到登录页面，登录页面会在后面的模块中讲解。对于 Web 应用而言，需要对现有的用户信息进行管理，所以管理员能够在后台管理页面中管理相应的用户，进行编辑和删除等操作，如图 22-12 所示。

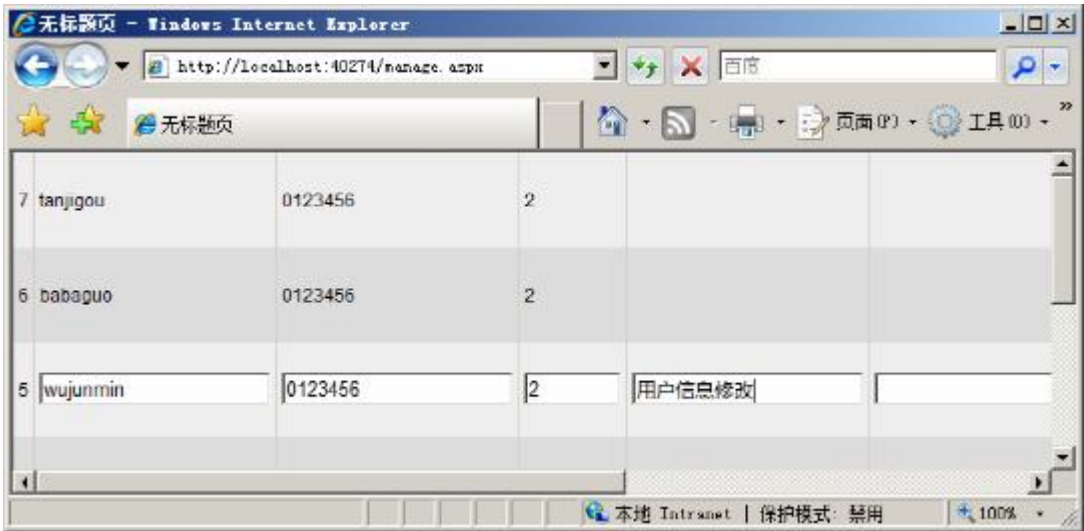


图 22-12 编辑相应用户

管理员可以在管理页面对用户数据库中的相应字段进行修改和增加，如果用户忘记了自己的密码，也可以通过联系管理员获取和修改自己的密码，对于恶意注册的用户，管理员可以轻易的删除相应的注册用户。

### 22.7 小结

本章介绍了注册模块的开发流程和核心代码，注册模块是网站应用中非常重要的模块，通过注册模块能够实现网站和用户的信息交流，本章从案例分析，数据库设计到代码编写都进行了讲解。本章还巩固了：

- ❑ ASP.NET 的网页代码模型。
- ❑ Web 窗体基本控件。
- ❑ 数据库基础。
- ❑ ADO.NET 常用对象。
- ❑ Web 窗体数据控件。

通过本章的学习能够巩固和强化对本书中的这些章节的理解。



## 第 23 章 登录模块设计

登录模块能够配合注册模块让网站应用能够同用户进行信息交互，当用户在网站进行注册后，就需要登录模块进行用户登录，登录模块虽然看上去比较容易，但是要比注册模块复杂一些，如身份处理，这些复杂的地方需要使用 **ASP.NET** 内置对象。

### 23.1 学习要点

登录模块需要涉及到一些 **ASP.NET 3.5** 的基本知识，如果要仔细学习注册模块的开发，需要详细了解本书的一些章节知识，这些章节如下所示：

- ☐ **ASP.NET** 的网页代码模型。
- ☐ **Web** 窗体基本控件。
- ☐ 数据库基础。
- ☐ **ADO.NET** 常用对象。
- ☐ **Web** 窗体数据控件。
- ☐ **ASP.NET** 内置对象。

基本了解了以上章节的知识点后，就能够熟练学习和开发此模块。

### 23.2 系统设计

登录模块需与注册模块不同的地方在于登录模块面向的用户有两种情况，一种是用户已经注册了，另一种是用户还没有注册，对于没有注册的用户需要引导到注册页面，而对于没注册的非法用户必须进行登录限制。

#### 23.2.1 模块功能描述

登录模块是配合注册模块的另一个非常重要的模块，相比之下，登录模块需要考虑更多的情况，例如用户是否注册，以及用户是否是合法用户，如果是合法用户忘记密码了怎么办，如果是非法用户，登录了多次是否要进行限制等等。登录模块的功能基本可以描述如图 **23-1** 所示。

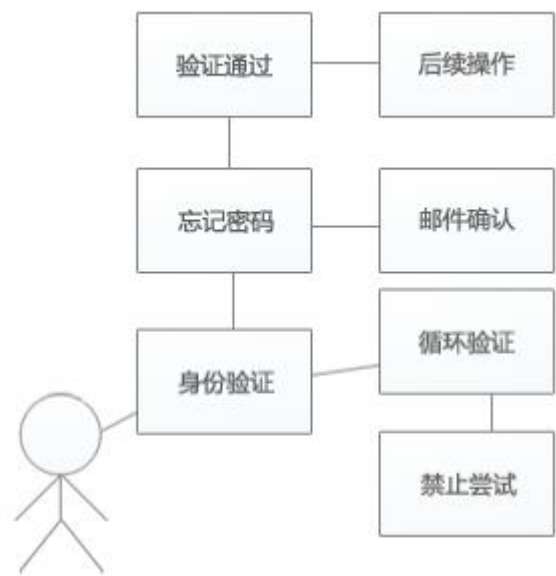


图 23-1 登录模块基本用户流程图

从登录模块可以看出，当用户进行身份验证后，可能会出现几种情况，包括验证通过、忘记密码和循环身份验证。如果用户是一个正常的用户，可以说一次就能够通过验证，那么这个用户就可以进行后续操作；如果用户已经是注册的用户，但是却忘记了密码，可以通过邮件确认进行密码的索要；如果用户是非法用户，在不断的进行尝试，那么就要禁止非法用户的不断尝试。从上述流程基本上可以规划以下几个页面：

- ❑ 登录页面：提供用户的主页面。
- ❑ 忘记密码页面：提供用户索取密码后提示的页面。
- ❑ 用户信息页面：提供用户登录成功后的个人信息页面。

在这其中最主要的是登录页面和忘记密码页面，其中很多的函数的实现都需要在这个页面实现，而其他页面主要是作为提示页面存在的。该模块需要使用 **ASP.NET** 内置对象对用户的操作进行保存和限制。

22.2.2 模块流程分析

在对业务进行了基本的划分之后，可以为模块进行基本的流程分析，包括这个模块中最基本的函数，以及这些函数在页面中是如何执行的。首先是登录模块需要提供哪些登录信息，登录模块中最重要的就是用户名和密码，登录模块通常情况下通过用户名和密码进行用户权限的判断。

如果用户登录成功，那么用户就是一个合法用户，可以进行后续的操作，如果用户登录失败，则需要让用户选择是否继续登录或者说明忘记密码，如果用户反复尝试则可以认为这个用户可能是非法用户，需要禁止该用户继续进行登录。在了解了基本的模块流程分析后，就可以进行函数和页面的划分，如图 23-2 所示。

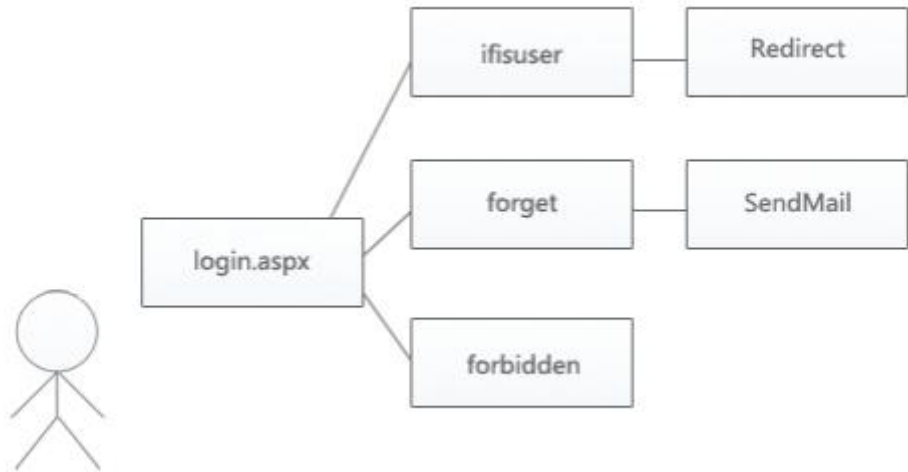


图 23-2 基本页面的函数分析

正如图 23-2 所示，这里主要起到作用的就是 **login.aspx** 页面，这个页面主要包括三个函数 **ifisuser**、**forget**

和 **forbidden**，分别作为判断用户是否为正常用户，以及判断用户是否忘记密码和非法用户等操作。在用户正常登录后，可以使用 **Redirect** 方法进行页面跳转，如果用户忘记了密码，需要使用发送邮件函数进行邮件发送，如果用户是非法用户，则需要禁止用户的登录。

23.3 数据库设计

对于登录表同样需要进行数据库设计，而登录表的数据库设计比较简单，只需要一个简单的用户表就能够进行登录设计。通常情况下注册模块和登录模块是一起协调合作的，登录模块读取用户表的信息而注册模块用于数据的索引和插入。

23.3.1 数据库设计分析

对于数据库设计分析，只需要简单的进行用户信息表的设计就可以了，但是这里需要使用用户信息表中的邮箱信息进行验证，所以数据库中表的字段可以归纳如下：

- ❑ 用户名：用户的用户名，用于登录使用。
- ❑ 密码：用户的密码，用于登录中输入密码。
- ❑ email：用户的 **E-mail**，用于发送邮件，如果用户忘记了密码就可以发送到该邮件。
- ❑ QQ/MSN：用户的 **QQ** 或 **MSN**，用于连接。
- ❑ 是否通过：用户的情况，用户保存用户信息，判断用户是否已经被通过。

这里最主要的字段是 **email** 和 **password**，这两个字段用于发送邮件到用户和判断用户是否被通过。如果用户忘记了密码，可以封锁该用户的用户信息然后发送邮件到用户的邮箱中，通过激活提示用户密码。

23.3.2 数据库表的创建

创建表可以通过 **SQL Server Management Studio** 视图进行创建也可以通过 **SQL Server Management Studio** 查询使用 **SQL** 语句进行创建。登录模块的数据库设计比较简单，这里创建一个 **Login** 数据库并创建一个表，如图 23-3 所示。

列名	数据类型	允许空
id	int	<input type="checkbox"/>
username	nvarchar(50)	<input checked="" type="checkbox"/>
password	nvarchar(50)	<input checked="" type="checkbox"/>
email	nvarchar(50)	<input checked="" type="checkbox"/>
msn	nvarchar(50)	<input checked="" type="checkbox"/>
passed	nvarchar(50)	<input checked="" type="checkbox"/>
ask	nvarchar(50)	<input checked="" type="checkbox"/>
answer	nvarchar(50)	<input checked="" type="checkbox"/>
		<input type="checkbox"/>

图 23-3 数据库表结构

正如图 23-3 中所示，表为用户的基本信息创建了字段，这些字段的意义分别为：

- ❑ **id**：用于标识用户的 **ID** 号，并为自动增长的主键。
- ❑ **username**：用于标识用户名。
- ❑ **password**：用于标识用户密码。
- ❑ **email**：用于标识用户 **E-mail** 信息。
- ❑ **msn**：用于标识用户的 **MSN** 等信息。
- ❑ **passed**：用于标识用户是否通过审核。
- ❑ **ask**：用于保存用户提示信息的问题。
- ❑ **answer**：用于保存用户提示信息的答案。

上述字段描述了相应的字段在实际应用中的意义，创建表的 **SQL** 语句如下所示。

```
USE [Login]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Login](                                //创建 Login 表
    bh] [int] IDENTITY(1,1) NOT NULL,
    username] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
    password] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
    email] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
    msn] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
    passed] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
    ask] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
    answer] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
    CONSTRAINT [PK_Login] PRIMARY KEY CLUSTERED
(
    bh] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
```

上述代码创建了一个数据库并将 **ID** 设为自动增长的主键，该数据库用于保存用户的基本信息，本模块通常不会更改数据库的信息，只是对数据库进行调用而已。所以在调用之前必须插入若干新数据，示例代码如下所示。

```
INSERT
    NTO
    [Login]
    (username,password,email,msn,passed,ask,answer)
    alues
    ('guojing','123321','soundbbg@live.cn','hellome@hotmail.com',1,"你好吗?","我很好")
```

上述代码在数据库中插入了一条用户名为 **guojing**，密码为 **123321** 的用户信息，并且这个用户的邮箱为 **soundbbg@live.cn**，当用户忘记密码时，就会通过这个邮箱发送确认信息。

## 23.4 界面设计

登录界面也能够吸引用户眼球，在登录界面也可以进行广告推广，因为一个网站的良好表现能够让用户大量的在登录页面停驻，在登录页面进行良好的设计可以使登录页面具有广告效应也能够提高用户体验。

### 23.4.1 基本界面

由于登录模块可能要考虑到很多的扩展，包括广告位之类的，登录页面也可以单独进行一个页面的制作，这些页面包括基本的 **TextBox** 和 **Label** 控件用于呈现基本的页面信息，示例代码见光盘中源代码\第 23 章\23-1\23-1\Default.aspx 所示。

上述代码在页面中使用了三个 **Label** 控件，用于显示用户登录必须的信息，包括指引用户如何填写相应的名称，以及提示是否存在该用户，该页面还包括两个 **TextBox** 控件用于用户填写相关的信息，并且为了验证用户是否输入正确，在页面中使用了验证控件对用户输入进行控制，示例代码如下所示。

```
<asp:RequiredFieldValidator ID="RequiredFieldValidator1" runat="server"
    ControlToValidate="TextBox1"
    ErrorMessage="用户名不能为空"></asp:RequiredFieldValidator>
```



```
<asp:RequiredFieldValidator ID="RequiredFieldValidator2" runat="server"
    ErrorMessage="密码不能为空"></asp:RequiredFieldValidator>
```

在注册控件已经说明了，验证控件能够验证用户是否输入的是合法的信息，如果用户输入的信息不合法或者输入的信息为空，那么就不应该让操作继续进行，而需要让用户再次进行信息输入。在没有 **CSS** 样式控制的情况下，使用了表格进行基本的布局，如图 23-4 所示。



图 23-4 基本界面布局

23.4.2 创建 CSS

为了更好的为页面进行页面布局，可以使用 **CSS** 进行页面的样式控制。在登录页面中，可以为页面和控件进行样式控制，**CSS** 示例代码如下所示。

```
body //定义全局
{
    font-size:12px;
    font-family:Geneva, Arial, Helvetica, sans-serif;
    margin:0px 0px 0px 0px;
    background:gray;
}
.top //定义头部
{
    background:white;
    margin:0px auto;
    margin-top:50px;
    padding-top:10px;
    padding-bottom:10px;
    padding-left:10px;
    width:490px;
    font-size:18px;
}
.login //定义登录
{
    background:white;
    margin:0px auto;
    width:500px;
}
.end //定义底部
{
    background:#f9fbfd;
    margin:0px auto;
    width:480px;
    text-align:center;
    padding:10px 10px 10px 10px;
}
```

上述代码定义了全局页面的字体大小和字体属性，并定义了头部样式、登录主样式和底部样式，定义完成后如图 23-5 所示。



图 23-5 CSS 样式控制后的页面

上述页面的布局非常鲜明，让用户一下就知道登录窗口在哪里，但是这个布局并不方便扩展，也不方便广告位的布局。这里不详细讲解如何进行广告位布局，只是介绍如何对登录页面进行样式布局。

23.4.3 发送密码页面

对于登录控件而言，需要两个提示页面，这两个提示页面包括发送密码页面和错误信息页面。发送密码页面主要是用于发送忘记密码的用户的密码到用户的邮箱中，这样用户就能够获取相应的信息以登录网站；而错误信息页面主要是用于提示用户输入的次数超过限定的次数，禁止用户再次输入。在这两个页面中，需要进行事务处理的页面只有发送密码页面，发送密码页面需要向指定的用户的邮箱发送邮件，而在发送邮件前，必须让用户输入用户名才能够发送。

**注意：**在用户忘记密码后，必须让用户输入用户信息，然后在数据库中查询相应的用户的邮箱的信息，而不是直接让用户填写邮箱。

发送页面示例代码见光盘中源代码\第 23 章\23-1\23-1\Mail.aspx 所示。其中，代码创建了一个发送邮件页面，当用户填写用户名后，系统会在数据库中查找相应的用户名的用户信息，查找完成后会发送相应的信息到用户的邮箱中，如果用户邮箱正确或者用户提示信息正确，那么系统会发送信息到相应邮箱，如果用户邮箱不正确或者用户提示信息不正确，系统则不会将密码信息发送到用户邮箱。页面布局完成后如图 23-6 所示。



图 23-6 发送密码页面

发送密码页面需要进行业务处理，在发送密码时，必须填写用户名和用户提示问题以及答案，才能够保证此用户是一个安全合法的用户。

## 23.5 代码实现

在完成基本的 **CSS** 页面布局后，就需要进行代码实现，登录模块的代码实现比较复杂，不仅需要查询相应的用户是否是合法用户，当用户忘记密码后，还需要通过邮件进行密码的索取，所以在代码实现中还需要实现邮件发送等功能。

### 23.5.1 登录代码实现

在用户进行登录时，必须验证用户是否已经登录，如果已经登录则不需要再次登录，如果没有登录，则允许用户进行登录操作，当用户单击【登录】按钮时，首先会验证用户是否填写信息，如果没有填写则提示用户填写，如果已经填写了，则判断用户是否是合法用户。

```
protected void Button1_Click(object sender, EventArgs e)
{
    string str = "server=(local);database='login';uid='sa';pwd='sa'";           //连接数据库
    SqlConnection con = new SqlConnection(str);                             //创建连接
    con.Open();                                                             //打开连接
    string strsql =
        "select * from login where username='"+TextBox1.Text+"' and password='"+TextBox2.Text+"'";
    SqlDataAdapter da = new SqlDataAdapter(strsql, con);                     //创建适配器
    DataSet ds = new DataSet();                                             //创建数据集
    int count=da.Fill(ds, "table");                                         //填充数据集
    if (count > 0)                                                          //登录成功
    {
        Session["name"] = TextBox1.Text;                                   //赋予 Session
        Session["password"] = TextBox2.Text;                               //赋予 Session
        Session["login"] = "yes";                                           //赋予 Session
    }
    else
    {
        Label3.Text = "登录失败";                                           //登录失败
    }
}
```

当需要判断一个用户是否为合法用户时，只需要在数据库中查询出该用户即可，如果查询出该用户，则说明这个用户是存在的；如果查询不出该用户，则说明这个用户是不存在的。查询用户可以使用 **ADO.NET** 的 **DataSet** 对象，示例代码如下所示。

```
"select * from login where username='"+TextBox1.Text+"' and password='"+TextBox2.Text+"'";
SqlDataAdapter da = new SqlDataAdapter(strsql, con);                       //创建适配器
DataSet ds = new DataSet();                                                //创建数据集
int count=da.Fill(ds, "table");                                           //填充数据集
```

上述代码使用 **DataSet** 对象和 **SqlDataAdapter** 对象进行数据填充，**DataSet** 对象的 **Fill** 方法会返回受影响的行数，当执行查询语句时，如果返回受影响的行数大于 **0**，则说明存在这个用户，如果受影响的行数小于等于 **0**，则说明不存在该用户。

**注意：**在验证用户时一定要同时进行用户名和密码的判断，如果不这样判断，很可能非法用户会猜出用户名就能够进行登录。

如果查询出的结果大于 **0**，则说明用户是合法用户，可以为用户赋予 **ASP.NET** 内置对象，以保存用户状态，示例代码如下所示。

```
Session["name"] = TextBox1.Text;                                           //赋予 Session
Session["password"] = TextBox2.Text;                                       //赋予 Session
```

Session["login"] = "yes";

//赋予 Session

上述代码当用户登录成功时，给每个用户一个 **Session** 对象，如果在一定时间内不进行操作或者用户关闭了浏览器进程，系统就会注销该用户。为了保证用户无法重复多次进行登录，可以在登录页面添加一个计数器，这里可以使用一个 **Label** 控件进行计数控制，**Label** 控件可以设置为不可见，初始值为 **0**，示例代码如下所示。

<asp:Label ID="Label4" runat="server" Text="0" Visible="False"></asp:Label>

在执行登录代码时，首先要判断该控件的值。这里设置登录 4 次后就无法登录了，示例代码如下所示。

```
if (Convert.ToInt32(Label4.Text) < 4)
{
    //登录操作
    //判断操作如下
    if (count > 0)
    {
        Session["name"] = TextBox1.Text;
        Session["password"] = TextBox2.Text;
        Session["login"] = "yes";
    }
    else
    {
        Label3.Text = "登录失败";
        int times = Convert.ToInt32(Label4.Text);
        Label4.Text = (times + 1).ToString();
    }
}
else
{
    Label3.Text = "您已经被禁止登录,请稍后再登录";
}
```

//判断登录次数

//赋予 Session

//赋予 Session

//赋予 Session

//提示登录失败

//登录次数

//登录次数加一

//静止登录

上述代码首先会判断计数器中的值是不是小于 **4**，如果小于 **4**，则可以进行登录操作，否则就会禁止用户登录。在登录失败时，必须让计数器的值加 **1**，否则计数器的值永远小于 **4**。

23.5.2 邮件发送页面

在用户需要索取自己的密码时，系统对用户进行邮件发送功能的实现和使用，这样就保证了用户信息的机密性，而用户可以在自己的邮箱中获取密码。邮件发送示例代码如下所示。

protected void TextBox1\_TextChanged(object sender, EventArgs e)

{

string str = "server=(local);database='login';uid='sa';pwd='sa'";

SqlConnection con = new SqlConnection(str);

con.Open();

string strsql = "select \* from login where username='" + TextBox1.Text + "'";

SqlDataAdapter da = new SqlDataAdapter(strsql, con);

DataSet ds = new DataSet();

int count = da.Fill(ds, "table");

if (count > 0)

{

Label5.Text = ds.Tables["table"].Rows[0]["ask"].ToString();

Label2.Text = "";

}

else

{

Label2.Text = "没有这个用户";

}

}

//创建连接字符串

//创建连接对象

//打开连接

//编写 SQL 语句

//创建适配器

//创建数据集

//填充数据集

//查找用户

//提示用户信息

//清空错误信息

//提示用户信息



当用户填写用户名并失去焦点时，系统会在数据库中查询相关的用户信息，如果包括该用户，则会提示这个用户的提问信息；如果没有这个用户，则提示没有这个用户，如图 23-7 所示。

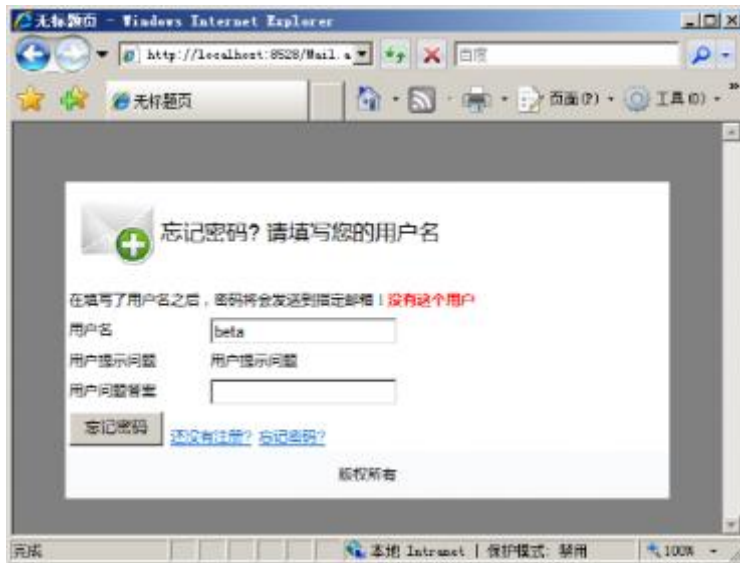


图 23-7 搜索用户信息

当用户填写完用户名和用户提示问题答案后，系统会判断用户答案是否正确，如果用户的答案是正确的，就会发送邮件到用户邮箱；如果用户答案不正确，则会提示用户再次输入答案，示例代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    string str = "server=(local);database='login';uid='sa';pwd='sa'"; //创建连接字符串
    SqlConnection con = new SqlConnection(str); //创建连接对象
    con.Open(); //打开连接
    string strsql = "select * from login where username='" + TextBox1.Text + "'"; //配置 SQL 语句
    SqlDataAdapter da = new SqlDataAdapter(strsql, con); //创建适配器
    DataSet ds = new DataSet(); //填充数据集
    int count = da.Fill(ds, "table"); //获取数据
    if (count > 0) //如果存在用户
    {
        if (TextBox2.Text != ds.Tables["table"].Rows[0]["answer"].ToString()) //对比问题
        {
            Label2.Text = "提示问题答案回答出错,请再次输入答案.."; //出现错误
        }
        else
        {
            SendUserMail(ds.Tables["table"].Rows[0]["email"].ToString(), //发送邮件
                ds.Tables["table"].Rows[0]["password"].ToString()); //实现邮件发送
        }
    }
    else
    {
        Label5.Text = "没有这个用户"; //声明没有用户
    }
}
```

上述代码会判断用户回答的问题是否和本身用户在注册时设置的问题相同，如果相同，就执行 **SendUserMail** 函数。**SendUserMail** 函数实现代码如下所示。

```
private bool SendUserMail(string recevie, string password)
{
    try
    {
        System.Net.Mail.SmtpClient client = new System.Net.Mail.SmtpClient();
        client.Host = "SMTP 服务器"; //SMTP 服务器信息
        client.UseDefaultCredentials = false;
        client.EnableSsl = false;
        client.Credentials = new System.Net.NetworkCredential("邮件发送邮箱", "发送邮箱密码");
    }
}
```

```

client.DeliveryMethod = System.Net.Mail.SmtpDeliveryMethod.Network;
System.Net.Mail.MailMessage message =
new System.Net.Mail.MailMessage("邮件发送邮箱", recevie);
message.Subject = "获取密码信息";                                //邮件的标题
message.Body = "您的密码为:"+password;                        //邮件的密码
message.BodyEncoding = System.Text.Encoding.UTF8;              //邮件的编码形式
message.IsBodyHtml = true;                                      //邮件内容的形式
try
{
    client.Send(message);                                        //发送邮件
    return true;                                                //返回真
}
catch (Exception ex)                                           //抛出异常
{
    return false;                                              //返回假
}
}
catch                                                         //不存在邮件服务器
{
    return false;                                              //返回假
}
}

```

上述代码实现了邮件的发送功能，这个函数的参数为接受者的地址和密码。在使用这个函数时，**SMPT** 服务器信息可以编写服务器所需要的邮件服务器 **SMPT** 服务器，例如 **126** 邮箱的 **SMPT** 服务器就是 **SMTP.126.COM**。编写完成 **SMPT** 服务器之后，只需要在上述代码中编写发送邮箱地址和发送邮箱密码就能够实现邮箱的发送。

注意：如果邮箱需要使用 **SSL** 安全证书才能够进行发送和接受，那么就需要配置 **System.Net.Mail.SmtpClient** 对象的 **EnableSsl** 为 **true**。例如 **Gmail** 就需要配置 **EnableSsl**。

### 23.5.3 根据不同的用户显示不同的内容

为了方便不同的用户显示不同的内容，可以使用 **ASP.NET** 提供的内置对象进行编程和判断，例如在登录时，如果登录成功，则系统会为用户配置一个 **Session** 内置对象。**Session** 内置对象寄宿在用户浏览器进程内，如果用户浏览器进程关闭或者用户长时间没有操作，则 **Session** 内置对象就会注销。

在 **Session** 生命周期内，可以使用 **Session** 内置对象对不同的用户进行页面编程，这样就能够实现不同的用户显示不同的内容。例如当用户登录后，会跳转到一个个人界面，这个界面可能是通用界面，但是需要不同的用户在当前界面操作。创建一个个人界面，当不同的用户访问该界面时显示的效果也不同，该页面为 **logged.aspx**，示例代码见光盘中源代码第 23 章\23-1\23-1\Logged.aspx 所示。

其中，代码在页面中添加了一个 **Label** 控件和一个 **Image** 控件，这两个控件分别对不同的用户呈现不同的效果。例如当用户“**soundbbg**”登录后，则系统应该提示说“感谢您 **soundbbg** 的登录”，而如果是用户“**wujunmin**”登录，则系统应该提示“感谢您 **wujunmin** 的登录”而不是原来的提示信息，同时也可以为相应的用户显示不同的用户头像，示例代码如下所示。

```

protected void Page_Load(object sender, EventArgs e)
{
    if (String.IsNullOrEmpty(Session["name"].ToString()))        //判断 Session
    {
        Response.Redirect("default.aspx");                        //页面跳转
    }
    else
    {

```

```
Label1.Text = Session["name"].ToString();           //获取 Session
if (Session["name"].ToString() == "guojing")        //执行编程
{
    Image1.ImageUrl = "mail.png";                   //获取图像
}
}
```

上述代码可以使用 **Session** 对象进行判断和编程，如果 **Session** 对象为空，则说明用户并没有登录或者用户为非法用户，那么就必须跳转到登录页面进行登录。如果用户是合法用户，并且用户是一些例如 **VIP** 等用户，就需要对特定的用户进行编程以呈现不同的样式。

在用户登录后，不仅可以使**Session**对象进行用户信息和权限的判断，也可以使用 **Cookie** 对象对用户信息和权限进行判断。使用 **Session** 对象和 **Cookie** 对象能够非常方便的进行用户信息的获取和存储。网站应用中，使用 **Session** 对象和 **Cookie** 对象是最常用的用户信息的获取和存储的方法，开发人员能够根据不同的使用 **Session** 对象和 **Cookie** 对象的值进行相应的编程。

23.6 实例演示

编写完成后就能够运行模块，这个模块包含登录、用户密码发送和用户个人页面，运行后用户登录页面如下图 23-8 所示。

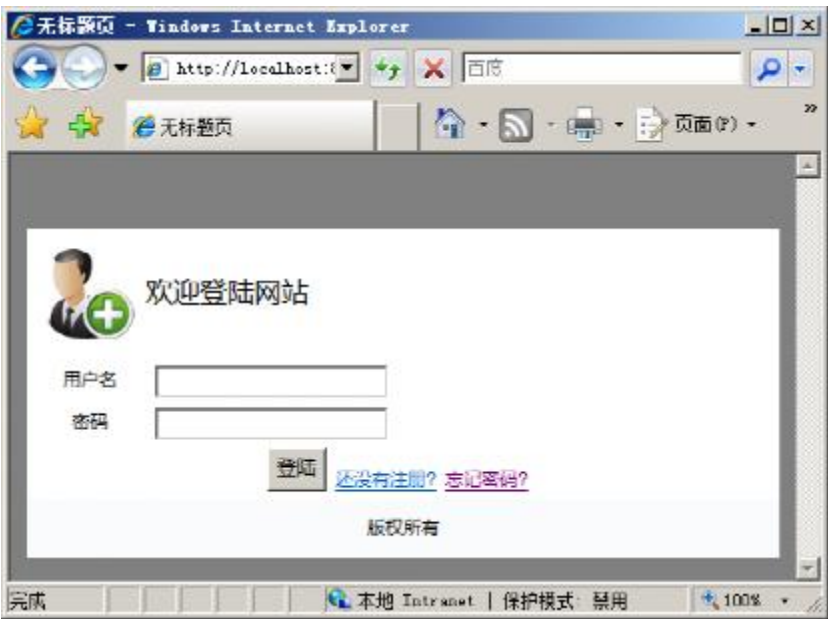


图 23-8 登录页面

当用户进行登录操作后，计数器就会开始计数，如果用户登录失败，用户还有 **3** 次机会能够再次进行登录。如果用户多次登录都没有成功，这就说明用户可能忘记了密码或者用户是一个非法用户，对于这样的用户必须禁止再次登录，如图 23-9 所示。

如果用户登录成功，则可以在相同的页面进行不同的用户信息的呈现，如果用户是 **VIP** 用户或者某个特殊的用户，就可以使用编程的方法为用户进行相应的页面编程，如图 23-10 所示，其中用户登录成功，登录成功后的用户头像已经被程序编制成 **mail.png**，否则是默认的头像显示。



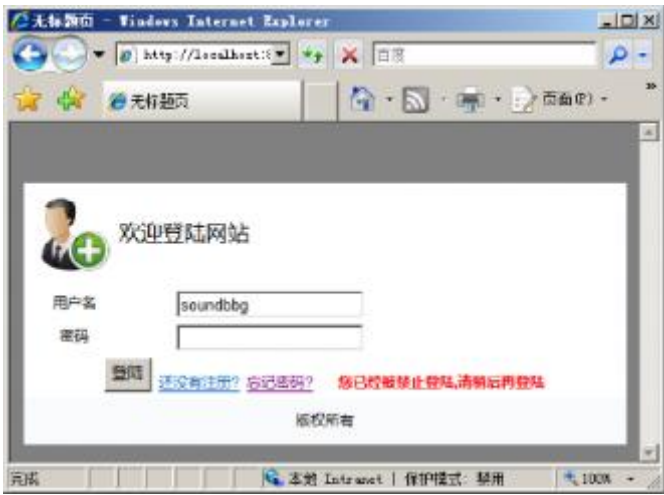


图 23-9 禁止再次登录



图 23-10 用户登录成功

如果用户多次登录都出现错误被禁止登录，用户可以单击忘记密码来让系统发送密码到用户邮箱中，在向用户邮箱中发送邮件前，必须要判断用户是否为合法用户，否则任何人都能够发送邮件到相应的邮箱中，如图 23-11 和图 23-12 所示。



图 23-11 不存在用户



图 23-12 存在一个用户

在发送邮件到一个用户邮箱之前，必须要检测是否包含这个用户，正如图 23-11 所示，如果不存在这个用户，那么系统就会提示不存在该用户。如果存在这个用户，则系统会提示一个用户提示问题，如图 23-12 所示，其中就提示了问题“你好吗”。用户必须在问题答案文本框中输入答案信息，如果答案正确的话就能够发送邮件到相应的邮箱，如果答案不正确就会提示用户答案不正确，需要再次填写答案并确认，如图 23-13 所示。

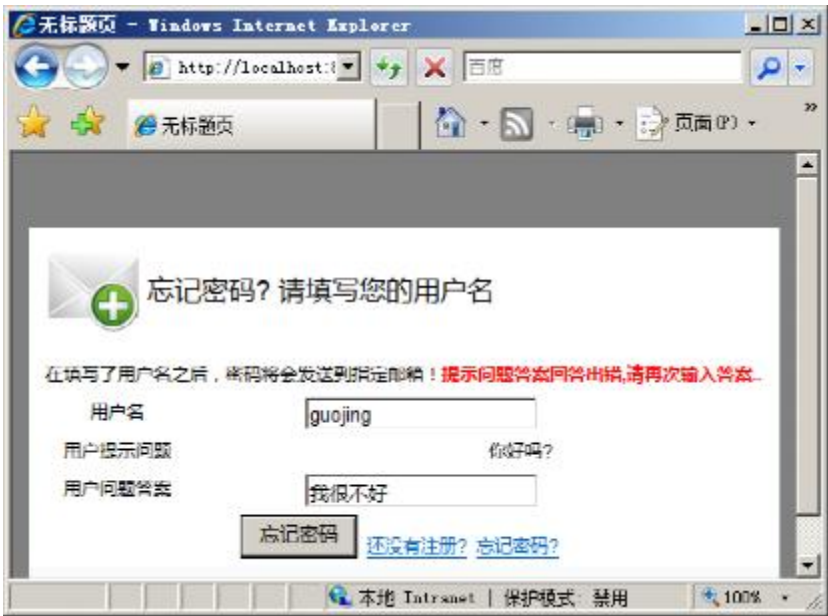


图 23-13 提示用户答案错误

如果用户答案错误，当用户单击【忘记密码】按钮时，就需要提示用户回答的答案错误，如果用户答案正确，就可以发送相应的邮件到用户的用户信箱中，用户可以通过查看用户信箱获取自己的密码。登录



成功的用户，系统会跳转到 **logged.aspx**，用户可以在该页面进行相应的操作，开发人员可以在该页面进行不同用户权限的获取和判断甚至是为某个或某些用户进行编程，这样就能够方便的利用 **ASP.NET** 提供的内置对象对用户进行编程操作。

### 23.7 小结

本章通过编程的方法实现了登录模块的编写，登录模块通常包括用户登录页面、忘记密码页面以及个人信息页面。登录模块通常情况下是和注册模块一起使用的，因为登录模块所需要的数据库也是注册模块所需要使用的数据库，当用户注册后，注册信息会存放在数据库中，登录模块只是在数据库中索取相应的信息，而不会对其中的信息进行操作。本章还巩固了：

- ❑ **ASP.NET** 的网页代码模型。
- ❑ **Web** 窗体基本控件。
- ❑ 数据库基础。
- ❑ **ADO.NET** 常用对象。
- ❑ **Web** 窗体数据控件。
- ❑ **ASP.NET** 内置对象。

通过本章的学习能够巩固和强化对本书中的这些章节的理解。

## 第 24 章 广告模块设计

广告能够为网页带来很多的增色功能效果和盈利，广告模块的设计对网站来说非常重要，一个网站不可能只有一个广告或者网站的广告还需要手动增加和删除。广告模块需要随机的获取系统广告或者能够在相应的位置增加广告来实现更多广告效果。

### 24.1 学习要点

广告模块需要涉及到一些 **ASP.NET 3.5** 的基本知识，如果要仔细学习广告模块的开发，需要详细了解本书的一些章节知识，这些章节如下所示：

- ☐ **ASP.NET** 的网页代码模型。
- ☐ **Web** 窗体基本控件。
- ☐ 数据库基础。
- ☐ **ADO.NET** 常用对象。
- ☐ **Web** 窗体数据控件。
- ☐ **ASP.NET** 内置对象。
- ☐ 用户控件
- ☐ 自定义控件

广告模块制作的是一个或多个自定义控件，这样在多个不同的页面中就能够快速的使用控件进行广告开发，在基本了解了以上章节的知识点后，就能够熟练学习和开发此模块。

### 24.2 系统设计

广告系统是一个非常重要的系统，随着网站的发展，如果网站需要实现盈利，就可以通过发布和获取广告信息来得到更多的盈利。管理人员希望在后台管理中，添加广告信息，包括文字广告和图片广告，这些广告能够按照一定的顺序随机的进行展现，并且当用户访问网站时，应该能够在页面中寻找到相应的广告并且点击广告。

#### 24.2.1 模块功能描述

在网站系统的发展过程中，广告投放是必不可少的一部分，因为现在的绝大部分网站都需要通过广告来盈利。不仅如此，广告还能够绝佳的展现网站现有的信息，如果网站是一个商城类型的网站，那么广告在这个时候还能够为自己的网站展现广告以达到宣传自己网站的目的。

广告的展现过程需要分几个类型的广告展现，最常见的是文字广告，文字广告是最常用也是最基本的广告类型，但是文字广告比较多的时候会引起用户的反感。除了文字广告还有图片广告，图片广告能够引起网站用户或者是消费者的兴趣，因为图片广告一目了然并且容易被网站用户发现和查看，如果图片广告制作的比较精良，那么会极大的提高用户的兴趣并提高点击量。在广告模块的设计中，考虑到有不同的广告类型和展现方式，通常情况下有以下三种广告展现方式。

- ☐ 文字广告：仅向用户展现文字广告。

- ❑ 图片广告：仅向用户展现图片广告。
- ❑ 图文广告：随机的向用户展现文字或图片广告。

文字广告和图片广告都是单纯的广告形式，只向页面中展现文字或者图片，提供一个超连接即可，而图文广告能够随机的展现文字广告或图片广告，当用户将页面刷新或者缓存更新时，对用户展现的广告是不同的。

由于广告模块是网站的管理者进行发布的，开发人员可以将广告模块制作成为自定义控件，使用自定义控件可以允许管理者或开发人员进行管理或发布，对于不同的广告类型，可以考虑不同的自定义控件。而对于管理员而言，广告发布流程是比较简单的，如图 24-1 所示。

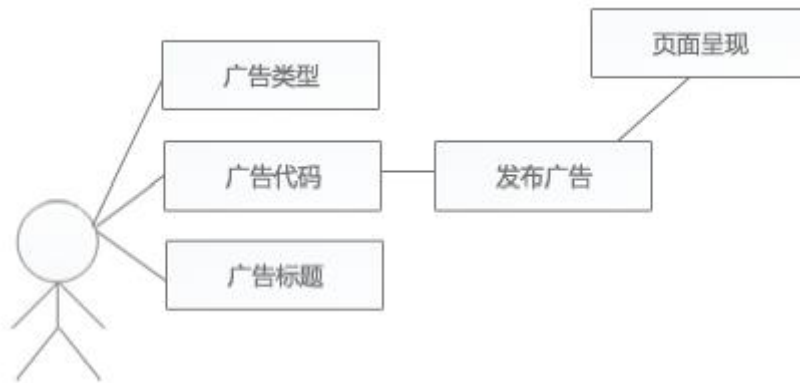


图 24-1 广告发布流程

正如图 24-1 所示，对于管理人员并无需做复杂的操作，管理人员只需要在后台编写相应的广告类型、广告代码和广告标题等广告模块需要的字段，然后进行广告的发布就能够在页面呈现了，但是对于开发人员来说，要让这个过程能够顺利的进行需要在页面呈现中进行筛选。如果是文字广告，就需要筛选出文字广告和广告说明，如果是图片广告，就需要筛选出图片和超链接，对于不同的广告类型开发人员必须筛选出不同的广告并呈现在页面中。所以对于开发人员可以选择如下两种方案进行广告模块的开发。

- ❑ 自定义控件：开发人员可以为不同的广告类型进行自定义控件的开发，对于页面编程人员可以拖动自定义控件到相应的位置进行广告的呈现，但是这样制作就有可能在页面中呈现多个相同的广告。
- ❑ 固定投放位置：开发人员可以固定投放位置，例如网站头部广告、网站底部广告和网站侧面广告，虽然这样做能够降低网站的重复广告的频率，但是这样制作无疑只能固定死网站的 **HTML** 代码，也显得不够灵活。

虽然网站广告投放没有最好的解决方案，通常开发人员也会通过页面的修改进行网站广告的投放，但是这里还是选择一个折中的方案，就是自定义控件的开发，自定义控件的开发可以通过编程的方法在页面中进行广告的控制和筛选，相比固定投放位置而言，在维护过程中更加的方便。从上述流程中可以基本规划几个自定义控件：

- ❑ 文字广告自定义控件：专门用于呈现文字广告自定义控件。
- ❑ 图片广告自定义控件：专门用于呈现图片广告自定义控件。
- ❑ 图文广告自定义控件：专门用于呈现图文广告自定义控件。
- ❑ 高级广告呈现控件：可以通过属性进行控制广告的呈现。
- ❑ 广告发布页面：管理员可以通过该页面进行广告发布。
- ❑ 广告呈现页面：管理员发布的广告能够在一个或多个页面进行呈现。
- ❑ 广告管理页面：管理员能够为不需要使用的广告进行管理。

通过编写多个自定义控件进行广告控制，也可以通过编写一个广告控件进行广告控制，同时管理员能够方便在后台进行广告发布并能够轻松的呈现在前台页面。

## 24.2.2 模块流程分析

在对业务进行了基本的划分之后，可以为模块进行基本的流程分析，包括这个模块中最基本的函数，以及这些函数在页面中是如何执行的。其中广告模块中需要开发广告发布页面，广告发布页面能够让管理员快速的发布广告，并选择广告发布的类型和位置，这样就能够轻松进行广告的发布，同时管理员也应该

能够进行广告的管理，包括修改和删除，如图 24-2 所示。

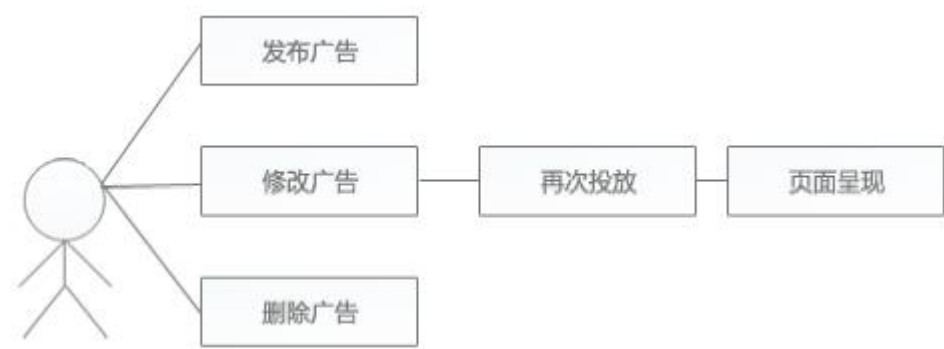


图 24-2 管理员发布流程

管理员能够在广告后台发布广告、修改广告和删除广告，当对广告进行修改后可以选择对广告再次进行投放，投放完成后就能够在页面进行呈现。对于上述流程可以分别开发若干个页面进行功能整合，如图 24-3 所示。

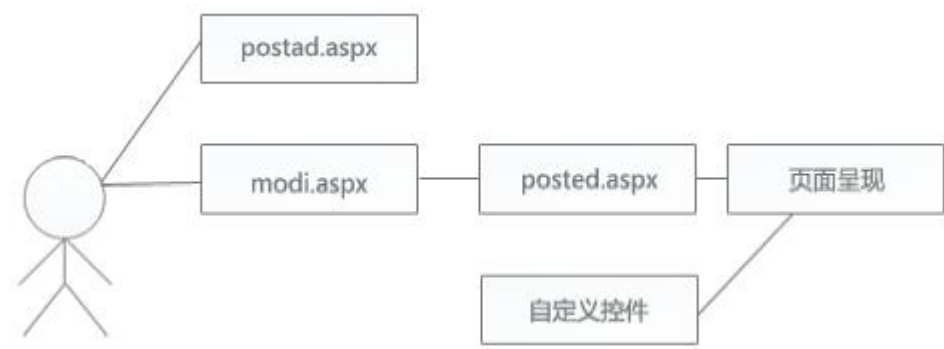


图 24-3 管理员操作页面划分

管理员在后台只需要进行广告管理等操作，而对于页面呈现，只需要从自定义控件中选取相应的数据并通过编程的方法进行整合呈现即可，而其中最重要的是如何进行页面呈现，只有灵活的将同类广告进行页面呈现才能够避免重复广告的出现。

## 24.3 数据库设计

对于广告模块的数据库设计可能比较的复杂，其复杂并不在数据库设计的本身上，数据库本质上就是一种存储数据的容器，而如何进行数据筛选在广告模块中是一个最为重要的过程，对于数据库的设计就需要考虑到广告模块中的数据筛选。

### 24.3.1 数据库设计分析

对于广告模块的数据库设计，需要加强数据条目的筛选功能，例如数据库中对广告的类型进行筛选，以选择不同类型的广告的不同呈现方式。在广告设计中，需要设计 3 个表，这三个表分别为 **ads**、**type** 和 **adclass**。其中 **ads** 表用于存放广告数据，其字段如下所示。

- ❑ 广告编号：表示广告的 ID 号，为自动增长的主键。
- ❑ 发布时间：表示广告发布的时间。
- ❑ 结束时间：表示广告发布结束的时间。
- ❑ 广告名称：作为广告的标识而存在，用于表示广告的名称。



- ❑ 广告内容：作为广告的内容而存在，可以是文字也可以是 **HTML** 代码。
- ❑ 广告备注：作为广告的备注而存在，用于标识备注信息。
- ❑ 广告图片：作为图片广告的图片连接。
- ❑ 广告连接：作为外部连接的广告的地址。
- ❑ 广告标题：作为广告的标题。
- ❑ 广告 **html**：作为广告呈现的 **HTML** 代码，可以为 **JavaScript** 代码。
- ❑ 广告类型：作为广告的类型而存在，类型没描述在 **type** 表中。
- ❑ 聚合类型：作为广告的广告 **ID** 而存在，用于归纳同类广告。

其中 **type** 表用于存放广告的类型数据，其字段如下所示。

- ❑ 分类编号：表示广告类型的 **ID** 号，为自动增长的主键。
- ❑ 分类名称：表示广告类型的描述，例如文字、图片等。

其中 **adclass** 表示广告显示的类型，使用 **ads** 表的 **adid** 表示可以表示广告在页面中呈现的归纳，其字段如下所示。

- ❑ 聚合分类编号：表示广告类型的 **ID** 号，为自动增长的主键。
- ❑ 分类名称：表示广告存放类型的描述，例如头部广告、底部广告。

对于广告模块来说，其数据表比较多，为了方便维护和扩展，就必须要让一些需要长期修改的字段进行外部连接。这样就能够极大的加强数据库中数据的健壮性和低耦合性。

注意：良好的数据库设计可能需要同时创建多个表进行一个功能的描述，虽然在数据库设计时这样的方法比较麻烦，但是在维护和开发中，这样会带来很多的便利。

24.3.2 数据库表的创建

创建表可以通过 **SQL Server Management Studio** 视图进行创建也可以通过 **SQL Server Management Studio** 查询使用 **SQL** 语句进行创建。广告模块需要创建多个表进行广告的描述，在创建表之前首先需要创建一个 **ad** 数据库，数据库创建完成后就能够在数据库中创建表了。这里首先需要创建一个 **ads** 表，该表用于存储广告模块中的广告信息，如图 24-4 所示。

列名	数据类型	允许空
id	int	<input type="checkbox"/>
time	datetime	<input checked="" type="checkbox"/>
endtime	datetime	<input checked="" type="checkbox"/>
name	nvarchar(50)	<input checked="" type="checkbox"/>
[content]	nvarchar(MAX)	<input checked="" type="checkbox"/>
infor	nvarchar(MAX)	<input checked="" type="checkbox"/>
picture	nvarchar(500)	<input checked="" type="checkbox"/>
url	nvarchar(500)	<input checked="" type="checkbox"/>
title	nvarchar(500)	<input checked="" type="checkbox"/>
html	nvarchar(MAX)	<input checked="" type="checkbox"/>
type	int	<input checked="" type="checkbox"/>
adid	int	<input checked="" type="checkbox"/>
		<input type="checkbox"/>

图 24-4 ads 表结构

正如图 23-4 所示，其中的字段意义如下所示。

- ❑ **id**：表示广告的 **ID** 号，为自动增长的主键。
- ❑ **time**：用于标识广告的开始时间
- ❑ **endtime**：用于标识广告的结束时间，当时间到达该时间后，广告将不再被呈现。
- ❑ **name**：用于标识广告的名称，这个名称在后台管理中可以进行辨认。
- ❑ **content**：作为广告的内容而存在，管理员能够在该字段进行广告内容的编写。
- ❑ **infor**：作为广告的备注而存在，管理员和管理员之间能够通过备注阅读该广告是什么广告。
- ❑ **picture**：作为图片广告的图片连接。

- ❑ **url**: 作为外部连接的广告的地址，用户单击广告时能够跳转到相应的连接。
  - ❑ **title**: 作为广告的标题，呈现在页面之中。
  - ❑ **html**: 作为广告呈现的 **HTML** 代码，可以为 **JavaScript** 代码，当广告为文字广告时，将呈现 **HTML**。
  - ❑ **type**: 作为广告的类型而存在，类型没描述在 **type** 表中。
  - ❑ **adid**: 作为广告的广告 **ID** 而存在，用于归纳同类广告，一个页面可以呈现一种或多种类型的广告。
- 上述字段描述了相应的字段在实际应用中的意义，创建表的 **SQL** 语句如下所示。

```
USE [ad]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[ads](
    [id] [int] IDENTITY(1,1) NOT NULL,
    [time] [datetime] NULL,
    [endtime] [datetime] NULL,
    [name] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
    [content] [nvarchar](max) COLLATE Chinese_PRC_CI_AS NULL,
    [infor] [nvarchar](max) COLLATE Chinese_PRC_CI_AS NULL,
    [picture] [nvarchar](500) COLLATE Chinese_PRC_CI_AS NULL,
    [url] [nvarchar](500) COLLATE Chinese_PRC_CI_AS NULL,
    [title] [nvarchar](500) COLLATE Chinese_PRC_CI_AS NULL,
    [html] [nvarchar](max) COLLATE Chinese_PRC_CI_AS NULL,
    [type] [int] NULL,
    [adid] [int] NULL,
    CONSTRAINT [PK_ads] PRIMARY KEY CLUSTERED
(
    [id] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
//创建 ads 表
```

上述代码创建了一个 **ads** 表用于存储广告数据，其中的 **type** 字段和 **adid** 字段都是其他表的外键，这三个表一起完成整个广告模块的数据描述，**type** 表创建的 **SQL** 语句如下所示。

```
USE [ad]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[type](
    [id] [int] IDENTITY(1,1) NOT NULL,
    [classname] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
    CONSTRAINT [PK_type] PRIMARY KEY CLUSTERED
(
    [id] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
//创建 type 表
```

**type** 表用于描述广告的类型，而 **adclass** 表用于描述广告呈现的类型，这两个表是有区别的。**type** 主要描述的是广告的类型，包括图片广告、文字广告等，是系统类型，通常情况下是不会更改的。而 **adclass** 用于描述的是广告呈现时所需要的类型，例如头部广告和底部广告，这些广告通过 **adclass** 表进行筛选和整合。**adclass** 表创建的 **SQL** 语句如下所示。

```
USE [ad]
GO
SET ANSI_NULLS ON
```

```
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[adclass](                                //创建 adclass 表
    [id] [int] IDENTITY(1,1) NOT NULL,
    [classname] [nvarchar](10) COLLATE Chinese_PRC_CI_AS NULL,
    CONSTRAINT [PK_adclass] PRIMARY KEY CLUSTERED
(
    [id] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
    ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
```

上述代码创建了一个 **adclass** 表，使用该表能够将广告进行分类并呈现到相应的页面中，可以极大程度的避免同种类型的广告的呈现。数据库的设计是非常重要的，也是在软件开发过程中一个非常重要的环节。在广告模块中，必须先规定好，以及规划好广告模块的数据库设计，否则数据库的更改会带来很多的不便，例如如果将 **adclass** 表和 **type** 表整合在 **ads** 表中，如果要修改一个字段的值，例如修改图片类型的广告，有可能需要更改一个或多个数据，这样就非常的不方便，也会导致数据的混乱，所以数据库设计在任何模块甚至是系统的开发过程中都是非常重要的一个环节。

## 24.4 界面设计

对于广告模块的界面设计，并不像前面两个模块一样对界面的要求很高，也同样没有对用户体验进行要求。但是广告模块的界面设计也并不是很简单，由于广告模块的呈现需要使用自定义控件进行 **HTML** 代码的呈现，其界面设计反而要求开发人员有较熟练的 **HTML** 编码能力。

### 24.4.1 发布广告界面

发布广告界面作为管理员进行广告发布的页面，这个页面无需特别复杂的呈现，因为管理员最终期望的是能够快速地进行广告的发布，而不是花哨的界面，不过虽然这样，还是需要进行一定的用户体验的开发，发布广告界面代码见光盘中源代码\第 24 章\24-1\24-1\Postad.aspx 所示。

其中的代码编写了广告信息的基本控件，管理员能够填写相应的广告信息用于广告的识别。在广告发布中，还需要填写广告发布代码，以及图片连接用于高级的广告信息的呈现，示例代码见光盘中源代码第 24 章\24-1\24-1\Postad.aspx 所示。

在该页面中，使用了若干控件，这些控件都分别为广告中的数据输入进行准备，这些控件包括 **TextBox** 文本框控件、日历控件和下拉菜单控件。下拉菜单作为数据绑定控件用于数据绑定，提供给管理人员选择相应的广告分类。

### 24.4.2 发布广告页数据源配置

在发布广告页面中使用了数据源控件进行数据源的呈现。在页面中，需要对数据源进行配置、筛选和生成才能够在发布页面中进行数据选择。单击【配置数据源】按钮，选择【新建连接】选项，在新建连接窗口中进行数据源配置，如图 24-5 所示。

拖放一个数据源控件到页面，用于配置 **adclass** 数据连接和数据绑定，创建数据连接后，选择【将数据连接保存到 **Web.config**】选项，在项目里就可以使用该连接进行数据连接和绑定，如图 24-6 所示。



图 24-5 创建新数据连接



图 24-6 创建连接

创建连接后，就可以自动生成 **SELECT** 语句填充数据绑定控件，方便开发，如图 24-7 所示。在完成 **SELECT** 语句的配置，就可以在相应的控件中使用数据源呈现的数据，例如在广告类型的下拉菜单中就可以使用数据源控件进行数据显示，如图 24-8 所示。

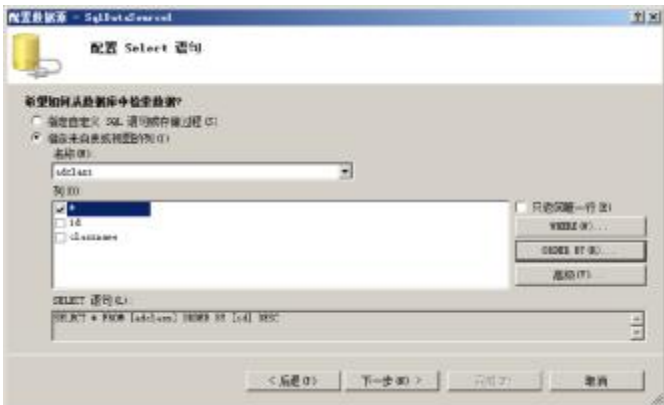


图 24-7 配置 select 语句



图 24-8 选择数据源呈现数据

在选择了数据源之后，就可以为另一个下拉菜单进行数据源配置，配置过程与上面的代码相同。配置完成后，页面增加了数据源控件的代码和数据绑定控件代码，示例代码见光盘中源代码\第 24 章\24-1\24-1\Postad.aspx 所示。

24.4.3 修改广告界面

修改广告界面同发布广告界面相同，但是修改广告界面在加载时必须接受一个传递的参数 **id** 来查询相应的广告信息，加载完成后就要填充到修改广告页面的控件中。这也就是说，当页面加载时，加载之后的修改广告页面应该先获取广告信息提供给管理人员修改，修改广告界面代码见光盘中源代码\第 24 章\24-1\24-1\modi.aspx。

修改广告界面基本同添加广告界面相同，因为修改广告界面只需要进行广告的读取和修改即可，而广告中所需修改的字段同广告添加字段基本相同，所以在广告修改中只需要进行字段的显示和更新就能够实现广告修改页面的制作。



24.4.4 管理广告界面

管理广告界面可以使用现有的 **ASP.NET** 数据源控件和 **ASP.NET** 数据绑定控件实现，**ASP.NET** 数据源控件和数据绑定控件能够快速的提供数据的更新、删除等功能。由于这里使用的是自定义更新页面，就不能够使用数据源控件本身提供的数据更新功能，对于管理广告界面，只需要进行数据删除操作的支持即可，数据源示例代码见光盘中源代码\第 24 章\24-1\24-1\Manage.aspx。

上述代码配置了数据源控件的高级模式以支持数据绑定控件中的更新、删除等操作，这里只需要使用删除操作就能够实现广告的管理，更新操作无需使用自带的更新而使用自定义页面。单击【数据绑定】控件，在菜单中单击【功能模块】按钮，选择【添加新列】选项，在【选择字段类型】选项中选择【HyperLinkFiled】选项并填写 **HyperLinkFiled** 类型字段中提供的相应的数据列和数据显示策略，如图 24-9 所示。

在数据绑定控件中能够使用【更新】连接进行页面跳转功能的实现，如图 23-10 所示，其中就包括了系统自带的删除操作和开发人员自定义的更新操作。



图 24-9 添加字段

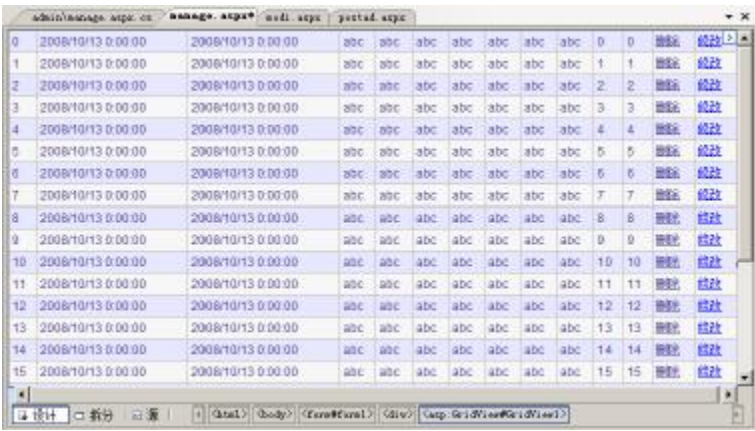


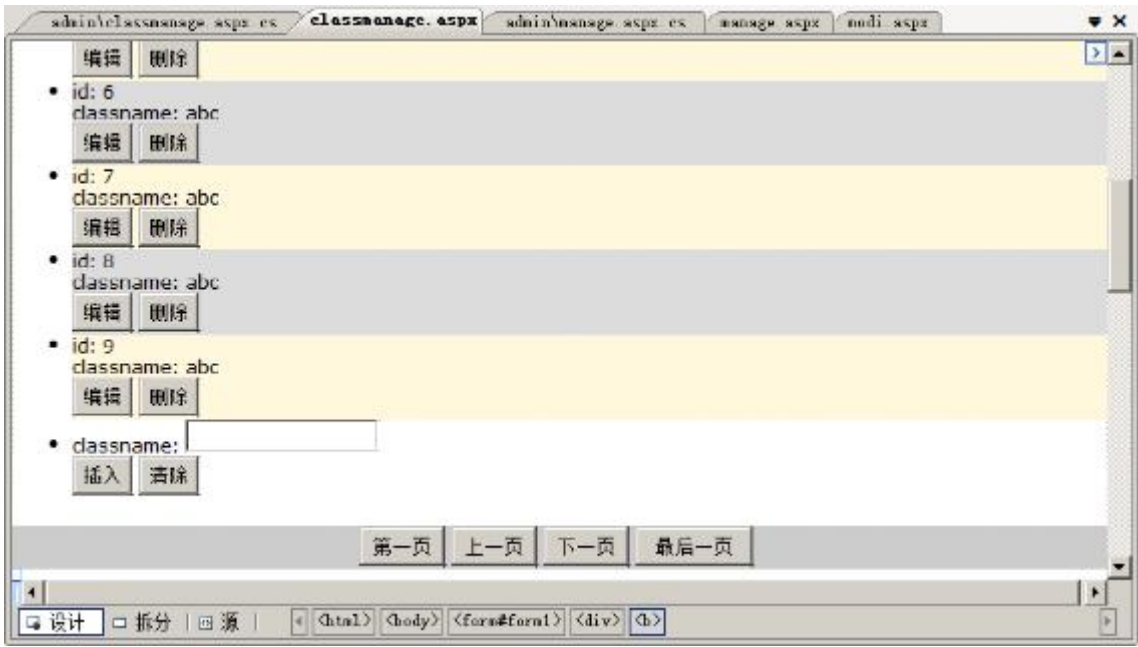
图 24-10 数据绑定控件 Grid View

其中数据绑定控件选择了自动套用格式让管理界面看上去更加的友好，管理人员能够在该界面查看相应的广告信息并且删除相应的信息，如果管理人员要修改相应的数据，可以单击【修改】按钮在自定义页面中进行广告的修改。

24.4.5 分类管理界面

分类管理界面比较的简单，因为分类管理表中的字段非常的少，所以分类管理界面就能够使用现有的控件，如 **Grid View** 控件进行数据插入、删除和更新，在分类管理界面中，可以直接使用控件进行操作，这样就能够多个页面进行复杂的管理，示例代码见光盘中源代码\第 24 章\24-1\24-1\ClassManage.aspx。

上述代码使用了 **ListView** 控件并自动套用格式，使管理员在操作的时候更加方便和简单，**ListView** 控件能够直接进行数据的插入、更新和删除，更加简便的进行了数据管理，如图 24-11 所示。



与 24-11 分类管理页面效果

分类管理页面是广告模块中一个比较容易实现的模块，但在功能上却是非常重要的模块，因为在广告的分类管理是非常重要的，在自定义控件的开发过程中，可以通过广告的分类管理进行广告的筛选，以及整合，通过广告的分类可以在网站的不同页面进行不同的广告的呈现，以及不同广告的筛选，避免了广告的重复。

## 24.5 代码实现

虽然控件为开发提供了良好的支持，但是控件毕竟样式死板、界面布局有限，而且代码实现也有限，所以很多情况下都需要使用自定义页面进行应用程序的开发，使用控件虽然能够方便和快速的进行功能开发，但是却无法避免死板的界面布局和有限的功能。

### 24.5.1 广告添加功能

广告添加功能可以使用 **ADO.NET** 进行广告添加，**ADO.NET** 可以执行 **INSERT** 语句进行数据库中的数据插入，在广告添加页面，管理员在填写完相应的项目时，可以单击按钮控件进行数据插入。在 **postad.aspx** 页面中，制作完成页面并双击【控件】按钮，**Visual Studio 2008** 能够自动生成相应的事件，开发人员可以在该事件中使用 **ADO.NET** 进行数据操作代码的编写，示例代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    try
    {
        SqlConnection con = new SqlConnection("Data Source=(local);Initial Catalog=ad;Integrated
                                                Security=True");           //创建连接
        con.Open();                               //打开连接
        string strsql = "insert into ads (time,endtime,name,content,infor,picture,url,title,html,type,adid)
                        values ('" + Convert.ToDateTime(Calendar1.SelectedDate).ToString() + "','"
                        + Convert.ToDateTime(Calendar2.SelectedDate).ToString() + "','" +
                        TextBox1.Text + "','" + TextBox3.Text + "','" + TextBox4.Text + "','" +
                        TextBox6.Text + "','" + TextBox7.Text + "','" + TextBox2.Text + "','" +
                        TextBox5.Text + "','" + DropDownList1.Text + "','" + DropDownList2.Text + "')";
        SqlCommand cmd = new SqlCommand(strsql, con);           //创建执行
        cmd.ExecuteNonQuery();                                   //执行 SQL
        Response.Redirect("manage.aspx");                         //页面跳转
    }
}
```

```

    }
    catch(Exception ee)
    {
        Response.Write(ee.ToString());           //抛出异常
    }
}

```

广告添加过程非常的容易，正如上述代码所示，直接对数据库中的数据进行插入操作就能够插入一条新广告，对于自定义控件，可以从数据库中获取广告和筛选广告进行呈现。

## 24.5.2 广告修改功能

广告修改页面是广告模块中的自定义页面，这个页面使用的是控件进行组合开发，当页面被加载时，首先需要通过传递的参数进行查询，查询后填充到控件中，示例代码如下所示。

```

protected void Page_Load(object sender, EventArgs e)
{
    try
    {
        if (!IsPostBack)                                     //判断加载
        {
            if (Request.QueryString["id"] == "")             //获取参数
            {
                Response.Redirect("manage.aspx");           //页面跳转
            }
            SqlConnection con =
            new SqlConnection("Data Source=(local);Initial Catalog=ad;Integrated Security=True");
            con.Open();                                       //打开连接
            string strsql = "select * from ads where id='" + Request.QueryString["id"].ToString() + "'";
            SqlDataAdapter da = new SqlDataAdapter(strsql, con); //创建适配器
            DataSet ds = new DataSet();                     //创建数据集
            int count = da.Fill(ds, "table");                //填充数据集
            if (count > 0)                                    //判断数据
            {
                TextBox1.Text = ds.Tables["table"].Rows[0]["name"].ToString(); //初始化控件
                TextBox2.Text = ds.Tables["table"].Rows[0]["title"].ToString(); //初始化控件
                TextBox3.Text = ds.Tables["table"].Rows[0]["content"].ToString(); //初始化控件
                TextBox4.Text = ds.Tables["table"].Rows[0]["infor"].ToString(); //初始化控件
                TextBox5.Text = ds.Tables["table"].Rows[0]["html"].ToString(); //初始化控件
                TextBox6.Text = ds.Tables["table"].Rows[0]["picture"].ToString(); //初始化控件
                TextBox7.Text = ds.Tables["table"].Rows[0]["url"].ToString(); //初始化控件
                Calendar1.SelectedDate = ds.Tables["table"].Rows[0]["time"].ToString(); //初始化控件
                Calendar2.SelectedDate = ds.Tables["table"].Rows[0]["endtime"].ToString();
                DropDownList1.Text = ds.Tables["table"].Rows[0]["type"].ToString(); //初始化控件
                DropDownList2.Text = ds.Tables["table"].Rows[0]["adid"].ToString(); //初始化控件
                Label1.Text = ds.Tables["table"].Rows[0]["id"].ToString(); //初始化控件
            }
            else
            {
                Response.Redirect("manage.aspx");           //页面跳转
            }
        }
    }
    catch
    {
        Response.Redirect("manage.aspx");                 //错误页跳转
    }
}

```



}

当页面被加载时就会执行上述代码，上述代码仅仅是在数据库中查询相应的数据，并呈现在相应的控件中，这样管理员在加载页面时就能够修改现有的数据内容进行更新。当管理员更新完毕后，单击相应的事件按钮就能够执行数据更新操作，数据更新操作示例代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    SqlConnection con = new SqlConnection("Data Source=(local);
                                         Initial Catalog=ad;Integrated Security=True");           //创建连接字符串
    con.Open();                                     //打开连接
    string strsql = "update ads set time='" + Calendar1.SelectedDate + "',endtime='" +
        Calendar2.SelectedDate + "',name='" + TextBox1.Text + "',title='" + TextBox2.Text +
        "',content='" + TextBox3.Text + "',infor='" + TextBox4.Text + "',html='" + TextBox5.Text +
        "',picture='" + TextBox6.Text + "',url='" + TextBox7.Text + "',type='" + DropDownList1.Text +
        "',adid='" + DropDownList2.Text + "' where id='" + Label1.Text + "'";           //更新 SQL
    SqlCommand cmd = new SqlCommand(strsql, con);           //创建执行
    cmd.ExecuteNonQuery();           //执行 SQL 语句
    Response.Redirect("manage.aspx");           //页面跳转
}
```

上述代码通过 ADO.NET 进行数据更新，从上面代码可以看出使用 ADO.NET 进行数据更新非常的简单，只需要打开数据库连接，在连接过程中使用 SqlCommand 对象执行 ExecuteNonQuery 方法进行数据的插入和删除操作就能够对数据库进行操作。

24.5.3 自定义控件的实现

在增加广告、删除广告和广告管理等页面制作完毕后，这也就意味着后台基本制作完毕，管理员可以在后台进行广告的增加和删除以及管理，也可以对广告类别进行管理。后台制作完毕后就需要在前台呈现广告，前台广告的呈现可以通过制作自定义控件进行呈现。

右击现有解决方案管理，在下拉菜单中选择【添加新项】选项，在【添加新项目】窗口中选择【自定义控件】项，这里创建一个名为 Ad 的自定义控件，如图 24-12 所示。



图 24-12 新增自定义控件

自定义控件用于筛选广告和呈现广告，筛选过程可以使用自定义控件的属性和方法完成，在编写完成自定义控件之后，就能够通过向页面拖动自定义控件和属性配置进行广告呈现，自定义控件属性编写示例代码如下所示。

```
[Bindable(true)]           //设置允许绑定
[DefaultValue("")]         //默认值为空
[Localizable(true)]        //允许本地化
public string type          //设置广告类型
```



```

{ get; set; }
[Bindable(true)] //设置允许绑定
[DefaultValue("")] //默认值为空
[Localizable(true)] //允许本地化
public string adid
{ get; set; }
[Bindable(true)]
[DefaultValue("Data Source=(local);Initial Catalog=ad;Integrated Security=True")] //设置连接字符串
[Localizable(true)]
public string SqlConnectionString
{ get; set; }
[Bindable(true)] //设置允许绑定
[DefaultValue("")] //默认值为空
[Localizable(true)] //允许本地化
public bool text //设置是否为文字
{ get; set; }
[Bindable(true)] //设置允许绑定
[DefaultValue("")] //默认值为空
[Localizable(true)] //允许本地化
public string CssStyle //设置 CSS 样式
{ get; set; }
[Bindable(true)] //设置允许绑定
[DefaultValue("")] //默认值为空
[Localizable(true)] //允许本地化
public string TitleCssStyle //设置标题 CSS
{ get; set; }
[Bindable(true)] //设置允许绑定
[DefaultValue("")] //默认值为空
[Localizable(true)] //允许本地化
public string ContentCssStyle //设置内容 CSS
{ get; set; }
[Bindable(true)] //设置允许绑定
[DefaultValue("")] //默认值为空
[Localizable(true)] //允许本地化
public int ShowNumber //设置显示个数
{ get; set; }

```

上述代码为自定义控件设置了属性，管理员可以使用此控件并设置属性进行控件的编写和调用相应的广告代码，自定义控件 **HTML** 页面实现代码如下所示。

```

protected override void RenderContents(HtmlTextWriter output)
{
    try
    {
        string constring = "Data Source=(local);Initial Catalog=ad;Integrated Security=True";
        if (SqlConnectionString != null) //获取连接字符串
        {
            constring = SqlConnectionString;
        }
        SqlConnection con = new SqlConnection(constring); //创建连接对象
        con.Open(); //打开连接
        string strsql = "select * from ads order by id desc"; //默认 SQL 语句
        if (type != null&&adid!=null) //筛选 SQL 语句
        {
            strsql = "select * from ads where type='" + type + "' and adid='" + adid + "' order by id desc";
        }
        else if (type != null) //筛选 SQL 语句
        {
            strsql = "select * from ads where type='" + type + "' order by id desc";
        }
    }
}

```

```

    }
    else if (adid != null)                                     //筛选 SQL 语句
    {
        strsql = "select * from ads where adid=" + adid + " order by id desc";
    }
    SqlDataAdapter da = new SqlDataAdapter(strsql,con);         //创建适配器
    DataSet ds = new DataSet();                               //创建数据集
    int count=da.Fill(ds, "table");                           //填充数据集
    if (count > 0)                                             //判断项数
    {
        if (ShowNumber < count)                               //判断生成条目
        {
            count = ShowNumber;                               //获取用户设置
        }
        StringBuilder build = new StringBuilder();           //创建 String 对象
        //开发人员可以在这里使用属性中的样式
        build.Append("<div style=\"padding:10px 10px 10px 10px;border:1px dashed #ccc;\">");
        for (int i = 0; i < count; i++)                       //遍历输出
        {
            build.Append("<div style=\"font-size:14px;border-bottom:1px dashed #ccc;\">
            <a href=\"" + ds.Tables["table"].Rows[i]["url"].ToString() + "\"> " +
            ds.Tables["table"].Rows[i]["title"].ToString() + "</a></div>");
            build.Append("<div
            style=\"font-size:12px;\"> " + ds.Tables["table"].Rows[i]["content"].ToString() +
            "</div>");                                       //输出 HTML
        }
        build.Append("</div>");                             //输出 HTML
        Text = build.ToString();                               //呈现广告内容
    }
    else
    {
        Text = "暂时没有任何投放的广告";                     //提示没广告
    }
    output.Write(Text);                                       //输出 HTML
}
catch(Exception ee)
{
    Text = ee.ToString();                                     //抛出异常
    output.Write(Text);                                       //输出异常信息
}
}
```

上述代码通过配置不同的属性进行 SQL 语句生成，例如当用户配置了 **type** 属性和 **adid** 属性，SQL 语句就能够生成筛选 **type** 和 **adid** 属性的 SQL 语句，当开发人员使用自定义控件时，只需要在页面中拖动相应的控件并配置相应的属性即可。

在需要使用广告控件的页面中，必须要引用控件，单击需要引用的项目，单击右键，单击添加引用，在项目选项卡中选择项目或在浏览选项卡中选择相应的 **DLL** 文件就能够添加引用，引用添加完毕后能够通过控件拖动生成广告代码，如图 24-13 所示。



图 24-13 自定义控件

当开发人员使用自定义控件并拖动到页面时，可以配置一些基本属性就能够呈现不同的广告类型。开发人员还能够进行多次开发和扩展进行自定义控件编程，通过获取和判断数据库中的字段进行不同类型广告的呈现。

24.6 实例演示

使用广告模块能够快速地进行广告的投放和使用，开发人员还能够使用自定义控件在页面中进行自定义控件的属性配置以呈现不同的广告的显示形式。作为管理人员可以直接在后台进行广告发布，在发布之前，需要创建一些广告类型，如图 24-14 和图 24-15 所示。

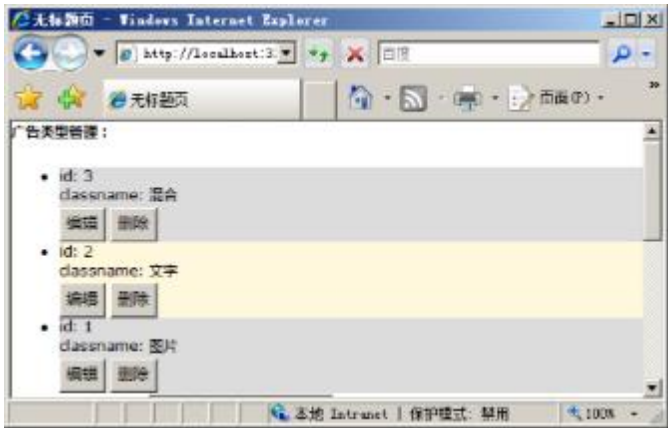


图 24-14 创建广告类型

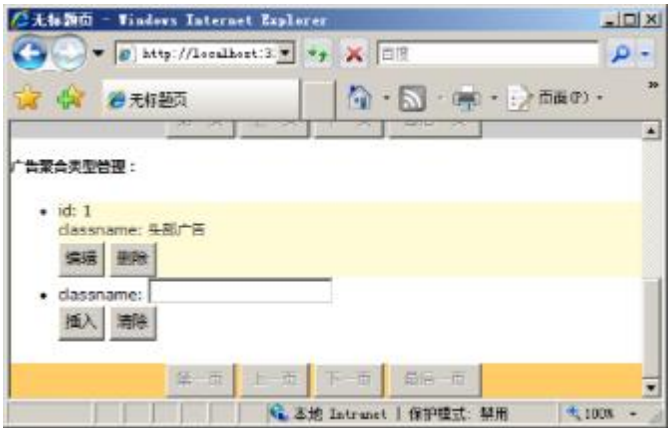


图 24-15 创建广告聚合类型

广告类型一般都不会更改，广告类型包括文字广告、图片广告和图文广告，开发人员可以配置属性显示文字广告、图片广告还是混合广告。如果显示的广告类型不同，那么呈现在相应页面的样式也不同，而创建广告聚合类型是能够避免广告的重复，例如选择头部广告、底部广告和侧边广告，可以选择一个广告只显示在头部而不显示在底部，避免了广告的重复。广告分类添加后就能够添加广告，如图 24-16 和图 24-17 所示。

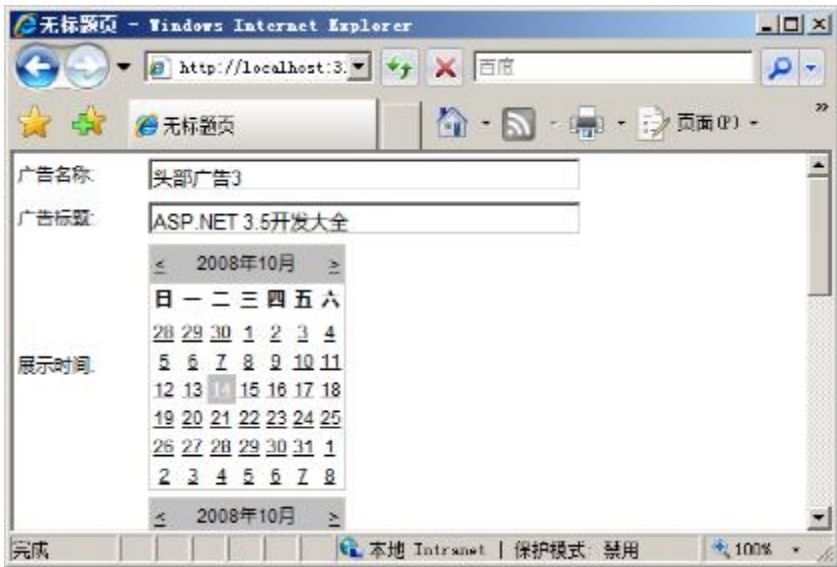


图 24-16 配置广告基本信息

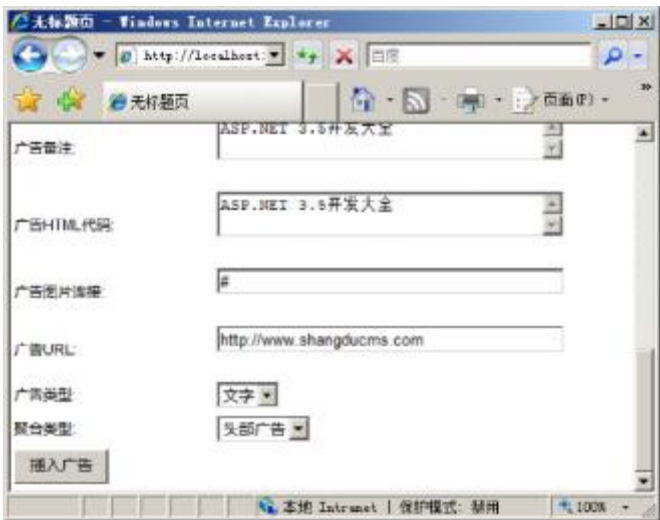


图 24-17 填写广告类型信息

插入完成后就能够在管理页面进行管理了，管理页面可以对广告进行修改、删除等操作，修改广告在自定义页面中进行修改，修改页面并没有使用现有的数据绑定控件进行修改，而是使用了自定义控件进行编程，通过使用 ADO.NET 获取数据并从数据中存放到自定义页面的控件中。

当管理员修改完成后，可以通过单击按钮进行数据更新，这时候广告就能够在各个页面被重新投放并且各个广告的样式和内容也会更改。如果开发人员为了提高广告的性能，可以在广告中使用缓存进行广告的缓存处理，限于篇幅的限制，这里就不再详细的讲解如何使用缓存进行性能提高。

在管理页面，管理员可以快速的删除广告信息，当管理员删除了广告信息后，广告信息就不会在相应的页面中进行投放，广告管理页面如图 24-18 所示。



图 24-18 广告管理页面

开发人员可以选择修改和删除相应的信息以在页面呈现不同的广告内容，开发人员在相应的页面可以使用自定义控件进行广告呈现，示例代码如下所示。

```
<body>
  <form id="form1" runat="server">
    <div>
      <cc1:ads ID="ads1" runat="server" ShowNumber="10"/>           //ShowNumber 为必填项
    </div>
  </form>
</body>
```

上述代码仅仅是简单的将控件拖动到页面中，就能够呈现广告内容，这里呈现的是文字广告内容，运行后如图 24-19 所示。

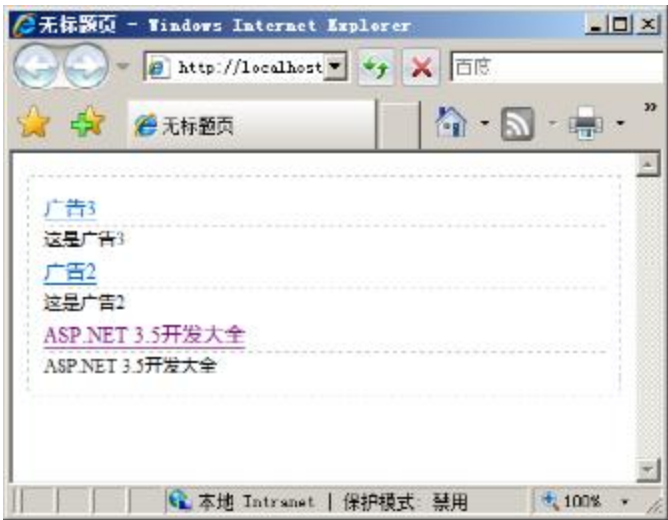


图 24-19 广告控件

当广告开发人员将广告控件拖放到页面中后，广告就能够呈现在相应的页面并提供给用户。管理人员在后台添加相应的类型的广告就能够在页面中呈现不同的广告内容。管理员能够配置自定义控件的属性进行不同的广告内容的控制，如果管理员将相应类型的广告全部删除，广告页面就会提示并没有呈现相应的广告。

由于篇幅的限制，这里的自定义控件并没有单独的制作图片广告的呈现方法，因为在图片文件的呈现和混合呈现的 **HTML** 代码生成中，需要筛选更多的数据或者生成更多不同种类的 **HTML** 代码，这里只是简单的进行了文字广告的呈现，在使用自定义控件的过程中，必须要配置相应的属性让控件能够使用数据库中的数据。

## 24.7 小结

本章通过编程的方法实现了广告模块的编写，广告模块是一个非常灵活和重要的模块，在网站的制作



过程中，广告模块能够加强网站与用户的交互性本章在编写广告模块的过程中还巩固了：

- ☐ **ASP.NET** 的网页代码模型。
- ☐ **Web** 窗体基本控件。
- ☐ 数据库基础。
- ☐ **ADO.NET** 常用对象。
- ☐ **Web** 窗体数据控件。
- ☐ **ASP.NET** 内置对象。
- ☐ 用户控件
- ☐ 自定义控件

本章使用了自定义控件进行了扩展开发，自定义控件是一个难度比较高的知识点，虽然自定义控件的开发并不困难，但是对于初学者而言，自定义控件很容易让人变得混乱。自定义控件能够实现更多更强大的自定义功能，熟练的掌握自定义控件能够简化开发，提高应用程序的灵活性。

## 第 25 章 新闻模块设计

现在的大部分网站都需要使用新闻模块进行网站信息交流，新闻模块是网站之中最传统的交流模块。管理人员能够通过后台进行新闻的发布和修改，用户就能够在前台页面中进行新闻的访问和评论，新闻模块是网站必不可少的模块，例如新浪、腾讯、搜狐等大型网站都离不开新闻模块。

### 25.1 学习要点

新闻模块需要涉及到一些 **ASP.NET 3.5** 的基本知识，如果要仔细学习新闻模块的开发，需要详细了解本书的一些章节知识，这些章节如下所示：

- ☐ **ASP.NET** 的网页代码模型。
- ☐ **Web** 窗体基本控件。
- ☐ 数据库基础。
- ☐ **ADO.NET** 常用对象。
- ☐ **Web** 窗体数据控件。
- ☐ **ASP.NET** 内置对象。
- ☐ 生成静态的概念

基本了解了以上章节的知识点后，就能够熟练学习和开发此模块。

### 25.2 系统设计

新闻模块对于网站而言是非常重要的，虽然今天 **Web 2.0** 的概念大行其道，但是新闻还是作为网站应用的基础内容而存在，新闻能够提供最简单的用户信息交互，对于新闻信息的筛选和投放同样能够吸引访问者。

#### 25.2.1 模块功能描述

新闻模块对于网站开发而言是最简单也是最重要的，对于网站而言，作为一个信息媒体，需要向用户，也就是网站的使用者进行信息传递。现在的各大门户网站，如新浪、腾讯和搜狐等，依旧使用的是新闻作为网站主导，而对于大行其道的 **Web 2.0**，同样也是基于新闻模块的形式进行信息呈现。

新闻模块的开发相对于广告模块而言从技术上实现比较的简单，并没有广告模块实现起来复杂和繁琐，也不需要使用自定义控件。但是新闻模块如果要制作好，还是有一定的难度的，其最主要的难度就在于生成静态和伪静态化。

对于不需要生成静态或伪静态化的新闻，其功能模块抽象起来比较的简单，在新闻使用之前，管理员可以在后台添加新闻分类，用于分类新闻。在添加新闻分类完毕后，就可以添加新闻并选择相应的分类进行新闻分类，分类后的新闻将能够呈现在不同的页面中以显示不同的分类的新闻。从一定的意义上来说，新闻模块的功能对于管理员而言，就只是添加分类和发布新闻，如图 **25-1** 所示。

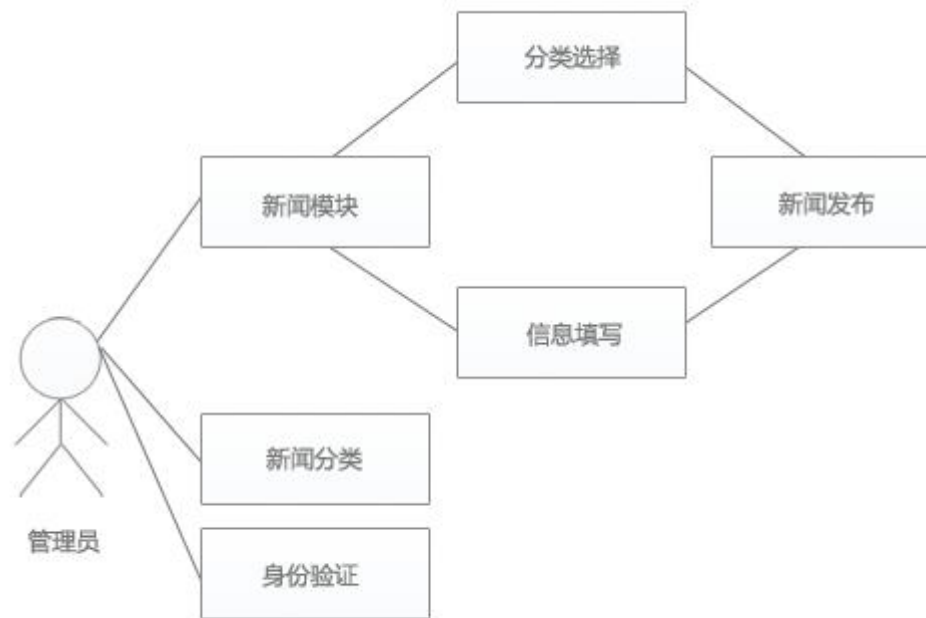


图 25-1 新闻模块基本流程分析

正如图 25-1 所示，管理员能够在后台进行新闻模块中的分类选择和信息填写进行新闻发布，管理员还可以对新闻分类进行管理。但是在管理员进行操作前，首先需要对管理员进行身份验证，以判断管理员是否有合法的权限进行身份验证。

身份验证可以使用登录模块进行身份验证，但是这里的登录模块没有网站的登录控件复杂。这里只需要实现对管理员进行判断，如果判断是管理员则能够通过，如果不是管理员则不允许通过的功能即可。从上述模块功能描述中可以规划成以下几个页面：

- ❑ 登录页面：管理员登录页面，为管理员提供身份验证。
- ❑ 新闻分类添加页面：为管理员提供新闻添加功能。
- ❑ 新闻分类管理页面：为管理员提供新闻分类的添加和管理。
- ❑ 新闻页面：用于显示新闻。
- ❑ 首页调用：用于进行新闻列表的显示，方便用户进行新闻查阅。

这些页面能够为管理员的新闻发布和更新进行操作提供，管理员首先需要在登录页面进行登录操作并进行身份验证。如果验证通过，就能够在新闻分类页面和新闻页面进行新闻分类操作和新闻操作，管理员可以通过新闻分类操作和新闻操作进行新闻的发布和归类，这样有助于在前台的页面中进行调用。

在前台显示中，同样还需要新闻显示页面和首页，新闻显示页面用于显示单个新闻，而首页用于显示新闻相应的列表，如在新浪、腾讯等网站的首页，都是调用最新的一些新闻列表来呈现的，这样有助于用户对新闻信息的筛选和分类。

### 25.2.2 模块流程分析

在各种类型的网站中，例如腾讯，都可以看到首页被各种新闻版块内容所填充，包括时事、体育、娱乐等等，这些新闻和内容版块都是在后台相关人员进行采编并纳入数据库和页面中的。可以想象，一个大型的门户网站每天会有多少的访问量，如果每次的用户访问都需要从数据库中读取数据，那么一天下来可能有几百万的读取次数，这样无疑会对 Web 应用带来极大的挑战。

可以观察各种门户的新闻，可以看得出来这些门户的新闻的 URL 地址的后缀都是 .html 或者是 .shtml 的，那么是不是这些网站的开发人员和采编人员当有一条新闻时就手动进行页面编写呢？显然答案是否定的，新闻网站可以将一些新闻静态化，这样就能够保证服务器只需要承受较少的压力依旧可以承担百万级的访问量。

生成静态就是将数据库中的数据或相应的字段进行静态化，例如将 .aspx 页面的文件进行静态化生成成为 .html 页面。 .html 是静态页面，当用户访问 .html 页面时无需进行数据操作和逻辑操作，对于服务器而言只需要将 .html 文本发送到浏览器就能够显示页面的内容。这样无疑增加了访问速度。如果网站要生成静态，其基本模块流程如图 25-2 所示。

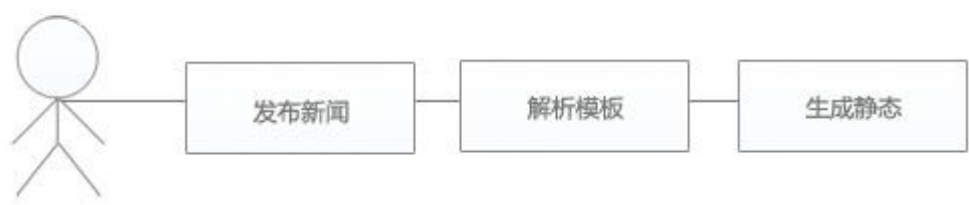


图 25-2 生成静态解决方案

虽然静态化能够降低服务器的压力，但是静态化同样会牺牲很多的空间。如果将新闻数据中的每个数据进行静态化，这也就是说每一条新闻就会生成一个.html 页面，那么有十万条新闻就会生成十万个.html 页面，这对服务器操作系统和 I/O 读写也有更高的要求，如果文件太多，打开文件夹的速度还不如读取数据库。虽然静态化是一个解决方案，但是很多情况下也可以不使用静态化。如果系统不使用静态化，可以使用非静态化的解决方案，如图 25-3 所示。



图 25-3 非静态化的解决方案

相比之下，非静态化的解决方案在实现上来说更加容易，因为静态化的实现方案还需要解析模板。在新闻模块的编写中，可以事先考虑是选择静态化的解决方案还是选择非静态化的解决方案，静态化的解决方案和非静态化的解决方案在开发过程中虽然可以替换，但是也有一定的开发风险。

而对于管理员而言，无需关心是否是静态化的解决方案还是非静态化的解决方案。在后台的操作过程中，管理员只关心自己如何能够快速的进行添加新闻和修改新闻等操作，在执行了相应的操作后，管理员就能够在前台进行新闻显示。

## 25.3 数据库设计

新闻模块同样需要多个表进行新闻描述和新闻操作，同样，为了安全起见和模块的可扩展性，还需要其他的表进行数据存储，这些表能够进行新闻的存储、身份验证、新闻分类的增删以及静态化生成保存等操作。

### 23.3.1 数据库设计

在新闻模块设计中，需要多个表进行新闻描述，同时为了保证管理用户的安全性，还需要设计管理员表，这些表包括 **news**、**newsclass** 和 **admin** 三个表，这三个表分别存储新闻、新闻分类和管理员信息。在创建表之前，首先需要创建数据库 **news**，创建完成后就能够创建相应的表。在对新闻模块进行流程分析之后，就能够大概的设计出这三个表中所需要的字段，其中 **news** 表所包含的字段如下所示。

- ❑ 新闻编号：用于标识新闻，为自动增长的主键。
- ❑ 新闻标题：用于表示新闻的标题。
- ❑ 发布时间：用于表示新闻发布的事件。
- ❑ 新闻作者：用于表示新闻的作者。
- ❑ 新闻内容：用于表示新闻的内容。
- ❑ 发布天气：用于表示新闻发布的天气。
- ❑ 新闻等级：用于表示新闻的等级。
- ❑ 阅读次数：用于表示新闻的阅读次数。



- ❑ 新闻分类：用于表示新闻的分类，为整型字段。
- 对于新闻分类表而言，可以使用少数字段进行新闻分类的描述，新闻分类表的字段如下所示。
- ❑ 分类编号：用于标识新闻的分类，为自动增长的主键。
- ❑ 分类名称：用于显示新闻分类的名称。

在管理员进行新闻操作之前，首先需要验证身份，如果管理员是合法用户则通过验证，否则就不允许进行后续操作，管理员的身份验证和登录模块基本相同，但是其功能要比登录模块少很多，**admin** 表结构中的字段如下所示。

- ❑ 管理员编号：用于标识管理员信息，为自动增长的主键。
- ❑ 管理员用户名：用于标识管理员用户名。
- ❑ 管理员密码：用于标识管理员的密码，通常情况下和管理员用户名一起进行身份验证。

对于新闻表而言，其功能并不是十分的复杂，而新闻模块的难度不在于数据库的设计上，而在于前台显示和静态生成，静态生成主要是要利用模板解析技术进行静态生成，模板解析技术可以使用编程的方法进行编写也可以使用数据库进行模板技术的支持，这里使用 **htm** 文本作为数据库进行技术解析处理，将在后面的章节进行介绍。

25.3.2 数据表的创建

创建表可以通过 **SQL Server Management Studio** 视图进行创建也可以通过 **SQL Server Management Studio** 查询使用 **SQL** 语句进行创建。新闻模块同样需要创建多个表进行模块功能的实现，首先最重要的是 **news** 表，**news** 表的字段如下所示。

- ❑ **id**：用于标识新闻，为自动增长的主键。
- ❑ **title**：用于表示新闻的标题。
- ❑ **time**：用于表示新闻发布的事件。
- ❑ **author**：用于表示新闻的作者。
- ❑ **content**：用于表示新闻的内容。
- ❑ **weather**：用于表示新闻发布的天气。
- ❑ **level**：用于表示新闻的等级。
- ❑ **hits**：用于表示新闻的阅读次数。
- ❑ **classname**：用于表示新闻的分类，为整型字段。

确定好 **news** 表的各个字段后，就能够创建一个 **news** 表，**news** 表结构如图 25-4 所示。

	列名	数据类型	允许空
PK	id	int	<input type="checkbox"/>
	title	nvarchar(50)	<input checked="" type="checkbox"/>
	time	datetime	<input checked="" type="checkbox"/>
	author	nvarchar(50)	<input checked="" type="checkbox"/>
	[content]	nchar(10)	<input checked="" type="checkbox"/>
	weather	nvarchar(50)	<input checked="" type="checkbox"/>
	[level]	int	<input checked="" type="checkbox"/>
	hits	int	<input checked="" type="checkbox"/>
	classname	int	<input checked="" type="checkbox"/>
			<input type="checkbox"/>

图 25-4 news 表结构

图中的字段描述了相应的字段在实际应用中的意义，创建表的 **SQL** 语句如下所示。

```
USE [news]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[news](
    [id] [int] IDENTITY(1,1) NOT NULL,
```

//创建 news 表

```
[title] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
[time] [datetime] NULL,
[author] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
[content] [nvarchar](3000) COLLATE Chinese_PRC_CI_AS NULL,
[weather] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
[level] [int] NULL,
[hits] [int] NULL,
[classname] [int] NULL,
CONSTRAINT [PK_news] PRIMARY KEY CLUSTERED
(
    [id] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
```

**news** 表中的 **classname** 字段为整型字段，这也就是说 **classname** 字段为另一个表的外键，另一个表 **newsclass** 用于描述新闻的分类的信息，**newsclass** 字段如下所示。

- ❑ **id**: 用于标识新闻的分类，为自动增长的主键。
- ❑ **classname**: 用于显示新闻分类的名称。

上述字段描述了 **newsclass** 表中需要使用的字段，可以使用 **SQL** 语句进行表和字段的创建，创建 **newsclass** 表的 **SQL** 语句如下所示。

```
USE [news]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[newsclass](                                //创建 newsclass 表
    [id] [int] IDENTITY(1,1) NOT NULL,
    [classname] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
    CONSTRAINT [PK_newsclass] PRIMARY KEY CLUSTERED
(
    [id] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
```

上述代码创建了 **newsclass** 表，创建完成后，还需要创建 **admin** 表，通过上述字段描述可以了解 **admin** 表只需要保存管理员的用户名和密码即可，则其字段可以描述为如下所示。

- ❑ **id**: 用于标识管理员信息，为自动增长的主键。
- ❑ **admin**: 用于标识管理员用户名。
- ❑ **password**: 用于标识管理员的密码，通常情况下和管理员用户名一起进行身份验证。

上述字段描述了 **admin** 表中需要使用的字段，可以使用 **SQL** 语句进行表和字段的创建，创建 **newsclass** 表的 **SQL** 语句如下所示。

```
USE [news]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[admin](                                    //创建 admin 表
    [id] [int] IDENTITY(1,1) NOT NULL,
    [admin] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
    [password] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
    CONSTRAINT [PK_admin] PRIMARY KEY CLUSTERED
(
    [id] ASC
```

```
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
```

上述代码创建了 **admin** 表，用于进行管理员的身份验证，创建完成后的 **admin** 表和 **newsclass** 表如图 25-5 和图 25-6 所示。

	列名	数据类型	允许空
🔑	id	int	<input type="checkbox"/>
	admin	nvarchar(50)	<input checked="" type="checkbox"/>
	password	nvarchar(50)	<input checked="" type="checkbox"/>
▶			<input type="checkbox"/>

图 25-5 admin 表结构

	列名	数据类型	允许空
🔑	id	int	<input type="checkbox"/>
	classname	nvarchar(50)	<input checked="" type="checkbox"/>
▶			<input type="checkbox"/>

图 25-6 newsclass 表结构

创建完成 **admin** 表之后就需要插入一个管理员，在 **SQL** 中可以新建查询并执行 **SQL** 语句进行管理员表中数据的插入，示例代码如下所示。

```
INSERT INTO admin (admin,password) VALUES ('guojing','0123456')
```

执行上述代码就能够进行 **admin** 表的数据插入，插入一个新管理员之后，就能够在后面的登录操作中使用该表的管理员信息。

## 25.4 界面设计

新闻模块包括众多的页面，这些页面包括登录页面、后台框架集、新闻发布页面、新闻删除页面等页面，这些页面都需要进行界面设计。在后台的开发过程中，虽然对后台的界面设计并没有苛刻的要求，但同样需要良好的用户体验，本章使用 **Microsoft Expression Web 2** 进行页面设计。

### 25.4.1 登录界面

登录界面用于进行管理员的身份验证，管理员可以在后台进行登录执行相应的新闻操作，如果管理员为合法用户，则允许进行新闻操作，否则不允许进行新闻操作，登录界面 **HTML** 代码见光盘中源代码\第 25 章\25-1\25-1\admin\login.aspx。

上述代码使用了 **TextBox** 控件以及验证控件和按钮控件，这些控件用于验证用户输入的是否正确并且判断用户是否为合法管理员，管理员可以通过该页面进行登录操作。如果登录成功，系统会跳转到后台管理框架集中，如果登录不成功，则会提示相应的错误信息。

### 25.4.2 后台框架集

后台操作中，为了提高页面的友好度，可以使用框架集进行后台开发，框架集是多个网页组成的一个页面，使用框架集能够在不刷新的情况下进行页面跳转，使用 **Microsoft Expression Web 2** 可以制作框架集。在 **Microsoft Expression Web 2** 中，单击【文件】选项，在下拉菜单中单击【新建】选项，单击【网站】选项，在弹出窗口中选择框架集，如图 25-7 所示。

框架集可以将多个页面放置在同一个页面，在 **Microsoft Expression Web 2** 中可以创建框架集并为框架集中的页面进行指定或新建，如图 25-8 所示。

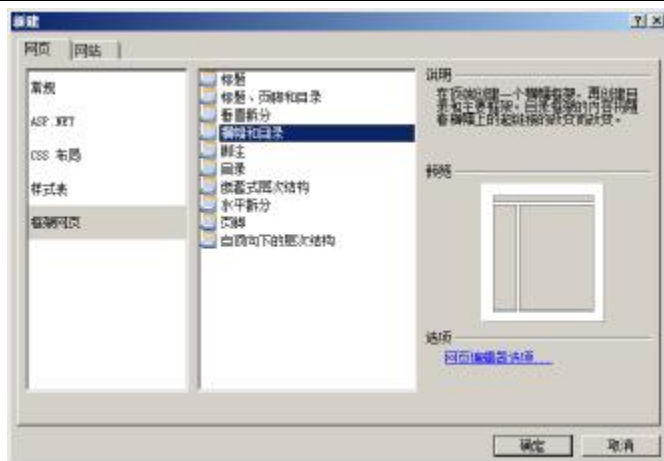


图 25-7 创建框架集

图 25-8 设置初始网页或新建网页

开发人员可以在框架集中创建网页或选择设置初始网页，这里创建三个网页，头部的网页用于显示后台管理的基本信息，包括这是什么后台管理系统；左侧的边栏用于显示操作，这里使用 **TreeView** 控件进行显示；中间为主操作区，该操作区用于后台中主要的页面操作。设置完成后示例代码见光盘中源代码\第 25 章\25-1\25-1\admin\default.aspx。

页面中的代码使用了一个框架集。在该框架集中包括三个页面，这三个页面分别为 **top.aspx**、**left.aspx** 和 **center.aspx**，其中 **top.aspx** 用于显示相应的信息，主要是用来作为导航或者后台提示，**left.aspx** 用于显示导航，使用 **TreeView** 控件能够为该页面制作相应的导航，而 **center.aspx** 用于呈现相应的操作页面，在这里可以被成为主工作区。

开发人员能够在不同的页面进行布局，控件拖动和事件等操作，当用户访问框架集时，各个页面之间互不影响，可以在框架集之间进行页面跳转，其中 **left.aspx** 代码见光盘源代码\第 25 章\25-1\25-1\admin\left.aspx。

**Left.aspx** 页面代码使用了 **TreeView** 控件在 **left.aspx** 页面中添加了导航信息，但是上述代码并没有配置 **TreeView** 控件中相应字段的 **URL** 属性，开发人员可以通过 **TreeView** 控件的属性进行配置。这里只提供 **left.aspx** 代码，对于其他页面的代码可以自行布局显示。如图 25-9 所示。



图 25-9 框架集布局

### 25.4.3 新闻发布页面

新闻发布页面是新闻系统中最为重要的页面，新闻发布页面主要使用 **ADO.NET** 进行新闻的发布和提交等操作，管理员能够在该页面进行新闻填写、新闻分类选择，然后管理员就能够进行新闻数据操作，新闻发布页面示例代码见光盘中源代码\第 25 章\25-1\25-1\admin\news\_add.aspx。

**News\_add.aspx** 页面代码使用了基本的文本框控件用于文本的输入。在一些用户数据输入时，为了保证用户输入的是完整的、符合规范的以及安全的数据，就需要使用下拉菜单控件进行数据呈现，示例代码见光盘源代码\第 25 章\25-1\25-1\admin\news\_add.aspx 中下拉菜单所示。

**News-add.aspx** 页面代码声明了多个文本框控件和下拉菜单控件用于文本的输入和呈现，管理员还需要通过数据源控件进行数据绑定并通过按钮控件进行数据提交。按钮控件用于数据提交，而数据绑定控件主



要用于绑定下拉菜单方便管理员选择，示例代码见光盘中源代码\第 25 章\25-1\25-1\admin\news\_add.aspx 中数据源配置代码所示。

新闻发布页面使用了数据源控件进行新闻分类的绑定，这也就是说明了在新闻添加之前，必须要选择新闻分类，否则新闻分类没有被填写，系统就会提示错误。在新闻页面设计中，使用 **TextBox** 控件和验证控件对管理员的操作进行验证和控制，如果管理员没有填写相应的信息，则系统会提示管理员填写，当管理员填写完成后，就可以单击控件进行提交。

25.4.4 新闻修改页面

新闻修改页面可以使用控件进行编写，新闻修改页面的数据获取同样需要从传递的参数中进行选择和判断，在 **ASP.NET 3.5** 中提供了一些数据绑定控件能够进行相应的数据的查询和更新，这里使用 **DetailsView** 控件，示例代码见光盘中源代码\第 25 章\25-1\25-1\admin\modi.aspx。

上述代码使用 **DetailsView** 控件进行数据绑定并能够使用 **DetailsView** 控件自带的更新功能进行数据更新。由于新闻更改页面需要通过获取的参数进行查询和更新，在配置 **DetailsView** 控件使用的数据源时，**SELECT** 查询语句必须配置参数，如图 25-10 所示。

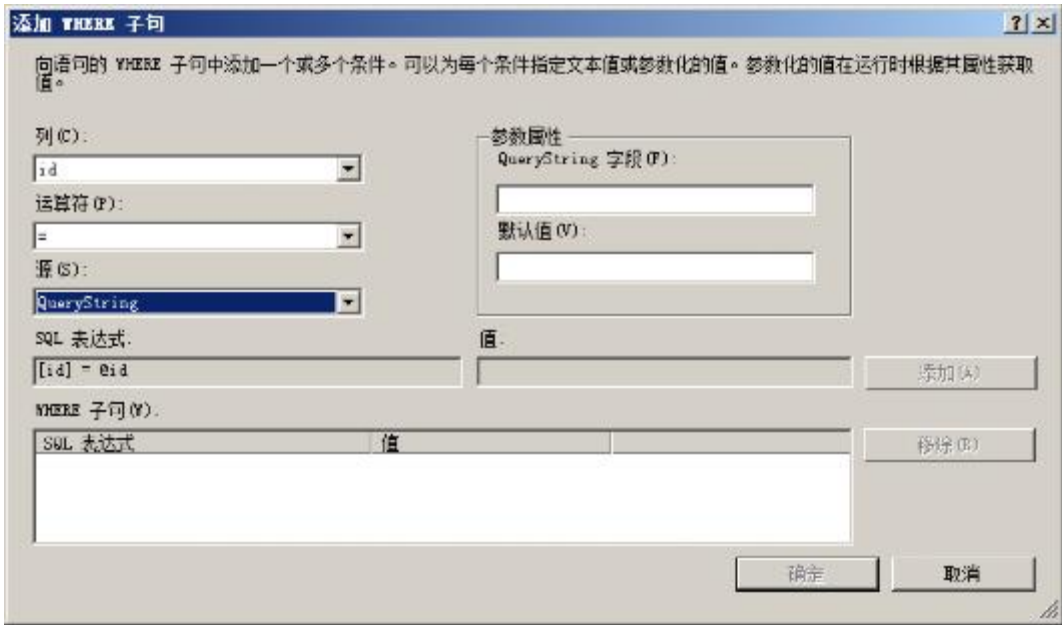


图 25-10 配置 WHERE 子句

配置 **WHERE** 子句就能够让数据源通过传递的参数进行相应的数据更新而不会涉及到其他的新闻数据，数据源源代码见光盘中源代码\第 25 章\25-1\25-1\admin\modi.aspx 中数据源配置代码。

在配置数据源时，同样需要配置能够自动生成“插入、更新、删除”等操作，这样数据绑定控件才能够支持数据的插入、更新和删除。在新闻修改页面，只需要进行新闻的更新即可，在配置数据绑定控件时，无需选择“插入、删除”等操作。

25.4.5 新闻管理页面

新闻管理页面可以使用 **GridView** 控件进行编程，这样不仅能够简化开发人员的开发操作，还能够提高开发效率，因为 **GridView** 控件能够支持数据的更新和删除，使用 **GridView** 控件能够直接执行数据的更新和删除操作。由于新闻修改页面是一个单独的页面，而且在单独的页面中进行新闻修改能够提高用户体验，所以在新闻管理页面中就不再使用修改功能，**GridView** 控件示例代码见光盘中源代码\第 25 章\25-1\25-1\admin\manage.aspx。

使用 **GridView** 控件可以进行相应字段的筛选，在管理页面中，并不需要每个字段都显示，例如 **content** 新闻内容字段。如果在管理页面同样要呈现 **content** 字段的话，那么当 **content** 字段的数据很多，例如是一篇很长的文章，那么页面就会被压缩的很难看，甚至变形。所以在新闻管理页面可以选择显示相应的字段而不显示一些不常用的字段，**GridView** 控件配置后如图 25-11 所示。

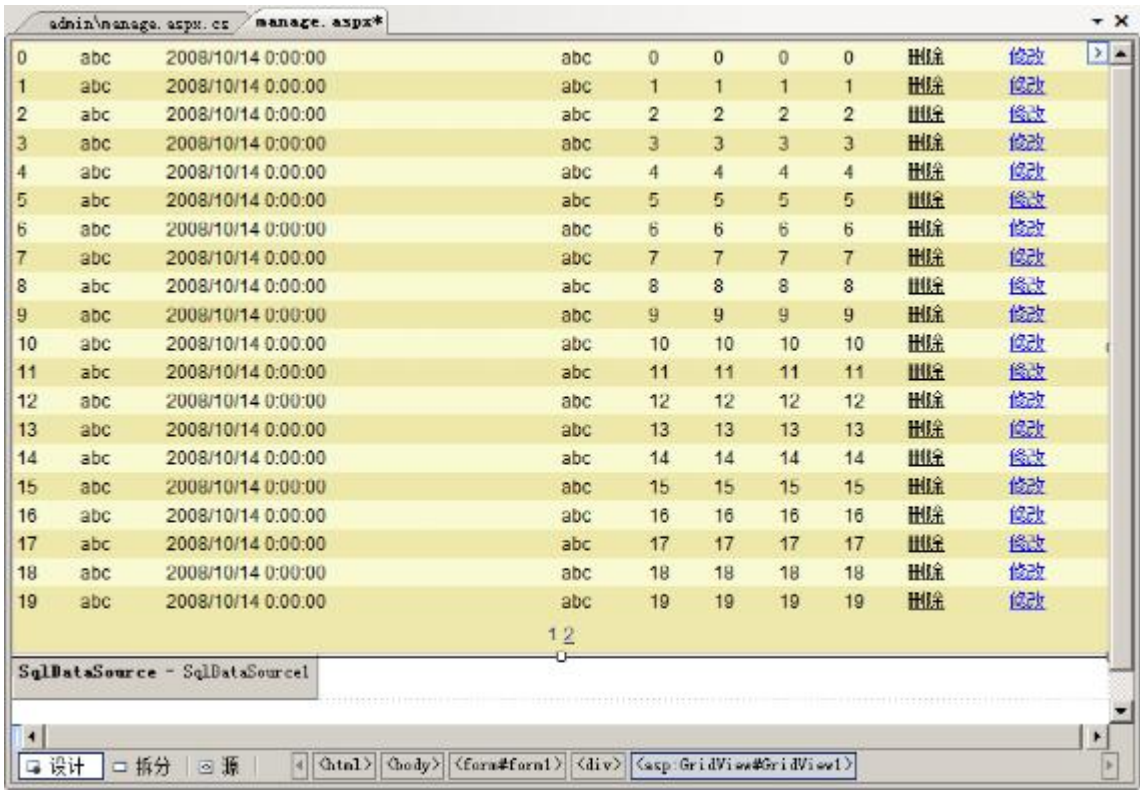


图 25-11 GridView 控件

GridView 控件自己能够支持删除操作。在 GridView 控件中，需要添加 HypeLink 列进行页面跳转。当管理人员单击【修改】连接时，就能够跳转到新闻修改页面进行新闻修改，而无需关心新闻修改页面的开发和维护。

25.4.6 新闻分类管理页面

新闻分类管理页面用于管理新闻分类，管理员可以在新闻分类管理页面进行新闻分类的添加和删除，在新闻分类的管理和添加页面中，同样可以使用 ASP.NET 3.5 提供的 ListView 控件进行分页、添加、修改、删除等操作，示例代码见光盘中源代码\第 25 章\25-1\25-1\admin\classmanage.aspx。

ListView 控件能够提供数据的插入、更新和删除等功能，对于简单的数据操作可以使用 ListView 控件进行功能实现，而如果需要复杂的数据操作和页面布局，使用 ListView 控件就不能很好的完成。ListView 控件通常情况下可以使用到数据较少，数据字段较短的情况下，如果对页面布局要求不是很高，也可以使用 ListView 控件。

25.5 代码实现

在新闻模块开发中包含了很多的页面，这些页面有些可以使用控件进行实现，而有些需要使用编程的方法进行代码实现，例如登录时的身份验证，新闻模块中的新闻发布甚至是新闻显示和生成静态等功能。这些功能没有现有的控件提供功能实现，而需要开发人员进行代码实现。

25.5.1 导航菜单配置

在后台管理的框架集中，可以配置导航菜单进行页面跳转，管理人员可以通过导航菜单进行页面的管理和跳转操作。当管理员单击导航菜单上的新闻管理时，就应该在主工作区中间显示新闻管理页面而不会刷新其他页面，同样当管理员点击其他操作时，会跳转到不同的页面。导航菜单示例代码见光盘中源代码\第 25 章\25-1\25-1\admin\left.aspx。

Left.aspx 页面代码使用了 TreeView 控件中节点的 NavigateUrl 属性配置了单击该节点时应该跳转的页

面，并使用了节点的 **Target** 属性进行了连接应该跳转的框架集。

注意：在超链接中包含 **target** 属性，而 **TreeView** 控件同样包含 **Target** 属性。在框架集中，需要指定相应的连接所需要跳转的框架，否则跳转的框架会在自身页面中执行。

## 25.5.2 身份验证页面

在管理员操作之前，首先需要进行身份验证，如果管理员是合法的用户，那么系统就能够使管理员进行添加和删除等操作。如果系统验证操作人员不是合法用户，那么就不应该为用户赋予权限，并阻止用户的登录和管理操作。打开登录页面，双击按钮编写相应的登录事件，示例代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    try
    {
        SqlConnection con = new SqlConnection("Data Source=(local);
                                                Initial Catalog=news;Integrated Security=True"); //创建连接
        con.Open();
        string strsql = "select * from admin where admin='" + TextBox1.Text + "' and password='"
                        + TextBox2.Text + "'"; //创建 SQL
        SqlDataAdapter da = new SqlDataAdapter(strsql, con); //创建适配器
        DataSet ds = new DataSet(); //创建数据集
        int count = da.Fill(ds, "table"); //填充数据集
        if (count > 0) //如果存在用户
        {
            Session["admin"] = TextBox1.Text; //配置一个 Session
            Response.Redirect("default.aspx"); //页面跳转
        }
        else
        {
            Label1.Text = "无法登录,请检查用户名和密码"; //提示无法登录
        }
    }
    catch
    {
        Label1.Text = "无法进行数据连接"; //抛出异常
    }
}
```

上述代码使用了 **ADO.NET** 中的数据对象进行数据操作，其操作和登录控件一样，在数据库中查询相应的信息，如果查询的结果大于 **0** 则说明该查询在数据库中是有效的，也就是说存在这样一个管理员能够进行登录，而如果查询的结果小于 **0** 则说明不存在相应的管理员。

在管理页面的各个页面都需要进行用户身份判断，当管理员登录成功后，系统会配置一个 **Session** 对象给用户，如果管理员操作超时或者操作者是一个非法用户，那么就没有 **Session** 对象。在各个页面判断 **Session** 对象是否存在，就能够判断是否是合法的管理员，示例代码如下所示。

```
if (Session["admin"] == null) //如果不为管理员
{
    Response.Redirect("login.aspx"); //登录跳转
}
```

如果相应的 **Session** 对象为空，就说明正在操作的用户不具备管理权限，则应该跳转到 **login.aspx** 页面进行重新进行登录操作。



### 25.5.3 新闻发布页面

在新闻发布页面，管理员只需要在相应的字段进行新闻内容的填写，包括新闻的标题、内容、作者等就能够进行新闻的发布。对于非静态生成的新闻发布而言，只需要进行数据的插入即可，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    if (Session["admin"] == null)                //如果不为管理员
    {
        Response.Redirect("login.aspx");          //登录跳转
    }
    TextBox2.Text = DateTime.Now.ToString();       //初始化字段
}
protected void Button1_Click(object sender, EventArgs e)
{
    try
    {
        SqlConnection con = new SqlConnection("Data Source=(local);
                                                Initial Catalog=news;Integrated Security=True"); //创建连接
        con.Open();                               //打开连接
        string strsql = "insert into news (title,time,author,content,weather,level,hits,classname) values
                        (" + TextBox1.Text + "," + TextBox2.Text + "," + TextBox3.Text + "," +
                        TextBox5.Text + "," + TextBox4.Text + "," + DropDownList1.Text + ",0," +
                        DropDownList2.Text + ")";    //SQL 语句
        SqlCommand cmd = new SqlCommand(strsql, con); //创建 Command
        cmd.ExecuteNonQuery();                     //执行 SQL 语句
        Response.Redirect("manage.aspx");          //页面跳转
    }
    catch(Exception ee)
    {
        Response.Write(ee.ToString());            //抛出异常
    }
}
```

新闻发布页面只需要执行相应的 **SQL** 语句进行数据插入即可，如果新闻发布只需要进行动态读取，那么只需要进行静态插入新闻即可，而不需要进行模板解析操作；如果需要生成静态页面，那么就需要进行模板编写再生成纯静态页面。

### 25.5.4 静态生成功能

静态生成听上去非常的复杂，但是其实静态生成非常的简单。当管理员发布一条新闻，就会在数据库中插入数据，数据插入后就应该解析模板进行静态生成。静态生成的是一个文件，这个文件可以是一个 **.html** 文件或者 **.shtml** 文件。在生成文件之前，可以运行以下代码，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)                                //判断加载
    {
        string str = "<b>*title*</b>";              //模板代码
        string database = "新闻标题";                //数据库字段
        string output = str.Replace("*title*", database); //替换操作
        Response.Write(output);                      //输出字符串
    }
}
```



上述代码简单的定义了一个模板代码，其中 **str** 变量被定义为标签，**database** 变量被定义为新闻标题，当需要生成静态时，可以将模板中代码的关键字进行替换，例如这里的 **\*title\*** 替换成为 **database** 变量中的标签，替换完成后输出替换后的字符串生成即可。在模板代码中，模板代码如下所示。

```
string str = "<b>*title*</b>";
```

其中 **\*title\*** 是定义的标签，用于替换关键字，**\*title\*** 可以替换新闻标题数据进行呈现。在了解了静态生成的原理之后，就能够使用模板解析进行静态生成，示例代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    try
    {
        SqlConnection con = new SqlConnection("Data Source=(local);
                                                Initial Catalog=news;Integrated Security=True");           //创建连接
        con.Open();                                           //打开连接
        string strsql = "insert into news (title,time,author,content,weather,level,hits,classname) values
                        (" + TextBox1.Text + "," + TextBox2.Text + "," + TextBox3.Text + "," +
                        TextBox5.Text + "," + TextBox4.Text + "," + DropDownList1.Text + ",0," +
                        DropDownList2.Text + ")";               //SQL 语句
        SqlCommand cmd = new SqlCommand(strsql, con);         //创建执行对象
        cmd.ExecuteNonQuery();                               //执行 SQL
        StreamReader aw = File.OpenText(Server.MapPath("template.htm")); //打开模板
        string template = aw.ReadToEnd();                     //读取模板
        aw.Close();                                           //关闭对象
        template = template.Replace("[新闻标题]", TextBox1.Text); //替换标签
        template = template.Replace("[发布时间]", TextBox2.Text); //替换标签
        template = template.Replace("[新闻作者]", TextBox3.Text); //替换标签
        template = template.Replace("[新闻天气]", TextBox4.Text); //替换标签
        template = template.Replace("[新闻内容]", TextBox5.Text); //替换标签
        StreamWriter sw = File.CreateText(Server.MapPath("../html/" + DateTime.Now.Year.ToString()
                                                                + DateTime.Now.Month.ToString() + DateTime.Now.Day.ToString()
                                                                + DateTime.Now.Hour.ToString() + DateTime.Now.Second.ToString()
                                                                + ".htm")); //生成文件
        sw.Write(template);                                   //编写内容
        sw.Close();                                          //关闭对象
        Response.Redirect("manage.aspx");                    //页面跳转
    }
    catch(Exception ee)
    {
        Response.Write(ee.ToString());                       //抛出异常
    }
}
```

上述代码首先使用 **ADO.NET** 进行数据插入操作，在数据操作成功后，就进行静态生成。静态生成的模板为 **template.htm**，通过编写 **template.htm** 文件能够为静态文件编辑相应的模板。在模板读取之后，系统会将一些关键字进行替换，例如将 “[新闻标题]” 这个字符串替换为数据库中的新闻标题，当 **template.htm** 模板中包含 “[新闻标题]” 字符串时，就能够通过程序将该字符串替换成相应的数据库中的数据。

在使用 **File** 类之前，需要使用 **System.IO** 命名空间，以提供对文件的读取操作。当读取模板并替换了关键字字符串之后，就能够通过 **File** 类的 **CreateText** 进行文件的存储，示例代码如下所示。

```
StreamWriter sw = File.CreateText(Server.MapPath("../html/" + DateTime.Now.ToString() + ".htm"));
sw.Write(template); //编写内容
sw.Close();         //关闭对象
```

上述代码生成了一个以时间为文件名的 **htm** 静态文件，当文件生成后，就会保存在 **html** 文件夹中。在编写静态标签时，需要注意的是就是静态标签应该比较复杂，例如不能够将 **title** 作为标签进行替换，因为可能文章中的很多字段都包含 **title**，而 **html** 页面本身就包含 **title** 字符串，如果将 **title** 进行替换，很有可能会将不应该替换的字符串替换成数据库文件，这样很有可能会造成模板输出错误。

在制作模板时，尽量使用一些符号进行标签规则，例如标题可以编写成为 **{ \$title\$ }** 或者 **[\*title\*]** 等，就不

会与现有的字符串中的字符进行冲突，替换了不该替换的字符串。

## 25.5.5 新闻显示页面

新闻显示页面的作用在于显示新闻，当用户单击一条新闻时就会跳转到新闻显示页面。例如在新浪网站上点击一个新闻，打开的页面就是新闻显示页面。新闻显示页面可以使用 **CSS** 和 **HTML** 布局，新闻显示页面的设计能够提高用户体验、吸引用户，新闻显示页面代码见光盘中源代码\第 25 章\25-1\25-1\articles.aspx。

上述代码编写了一个 **articles.aspx** 文件，当用户访问新闻时，会打开该文件并通过参数获取新闻的值，然后将值填充到 **Label** 等控件中。如图 25-12 所示。



图 25-12 新闻显示页面

在参数获取和填充的过程可以在 **Page\_Load** 方法中编写相应代码，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    if (Request.QueryString["id"] != "")
    {
        SqlConnection con = new SqlConnection("Data Source=(local);
                                                Initial Catalog=news;Integrated Security=True"); //创建连接
        con.Open(); //打开连接
        string strsql = "select * from news where id = " + Request.QueryString["id"] + ""; //查询数据
        SqlDataAdapter da = new SqlDataAdapter(strsql, con);
        DataSet ds = new DataSet(); //填充数据
        int count = da.Fill(ds, "table");
        if (count > 0)
        {
            Label1.Text = ds.Tables["table"].Rows[0]["title"].ToString(); //填充控件
            Label2.Text = ds.Tables["table"].Rows[0]["content"].ToString(); //填充控件
            Label3.Text = ds.Tables["table"].Rows[0]["author"].ToString(); //填充控件
            Label4.Text = ds.Tables["table"].Rows[0]["time"].ToString(); //填充控件
        }
        else
        {
            Response.Redirect("default.aspx"); //页面跳转
        }
    }
    else
    {
        Response.Redirect("default.aspx");
    }
}
```

新闻显示页面的 **HTML** 代码使用了 **Label** 控件进行页面文字的呈现，当用户打开页面时，会通过传递过来的 **id** 参数进行数据查询。如果数据查询成功，就会填充到相应的控件中，如上述代码所示，填充后就会显示相应的文本给用户，用户就能够看到新闻。

### 25.5.6 静态模板编写

如果新闻模块不需要静态生成，那么就可以通过 **id** 参数进行 **articles.aspx** 页面的访问，例如可以通过 **articles.aspx?id=1** 这样规则的地址进行访问。但是动态的访问不仅数据访问量大，还不能很好的被搜索引擎搜录，如果选择新闻模块要生成静态，就需要编写相应的模板。由于新闻模块在添加新闻时就需要生成静态文件，其静态文件的模板读取的是 **template.htm**，所以编写 **template.htm** 的 **HTML** 代码就能够编写模板文件，**template.htm** 示例代码见光盘中源代码\第 25 章\25-1\25-1\admin\template.htm。

上述代码编写了模板文件，其中[新闻标题]、[新闻作者]等都是模板标签。当生成静态文件时，系统会根据标签替换数据库中的数据，例如把[新闻标题]替换成“新闻标题”，然后生成到 **html** 文件夹中形成一个静态文件。

当用户单击新闻列表的时候，就可以跳转到相应的静态文件进行静态访问而无需动态读取文件，这样就能够快速的提高读取效率并降低服务器压力。

## 25.6 实例演示

使用新闻模块能够快速的发布新闻和显示新闻，如果新闻模块选择生成静态，那么就需要对模板进行编写和解析，管理员可以在后台快速的进行新闻发布，前台就能够显示相应的新闻。在发布新闻之前，首先需要进行登录，如图 25-13 所示。



图 25-13 登录验证

登录完成后，就能够跳转到后台框架集，后台框架集为管理员提供了遍历的操作，如图 25-14 和图 25-15 所示，其中侧边是导航栏，右边大块的区域是工作区。





图 25-14 主框架集

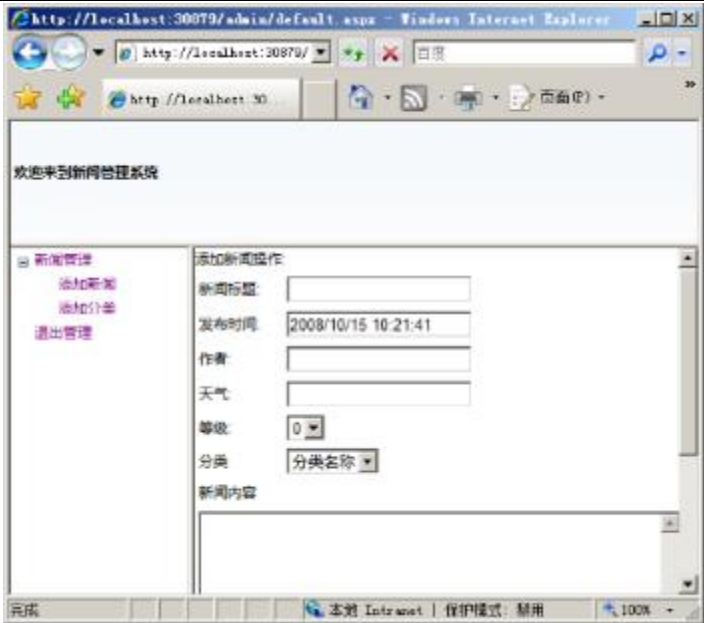


图 25-15 添加新闻

框架集能够让管理员方便的进行导航和功能操作，管理员可以在侧边栏中点击相应的菜单，例如单击【添加新闻】和【添加分类】选项进行相应页面的跳转。如果管理员单击的是【添加新闻】选项，则在工作区就能够呈现新闻添加页面；如果管理员单击的是【添加分类】选项，则主工作区就会呈现添加分类页面，而对其他的页面丝毫不会造成影响。单击【添加分类】按钮，首先要为系统添加一个分类，如图 25-16 所示。

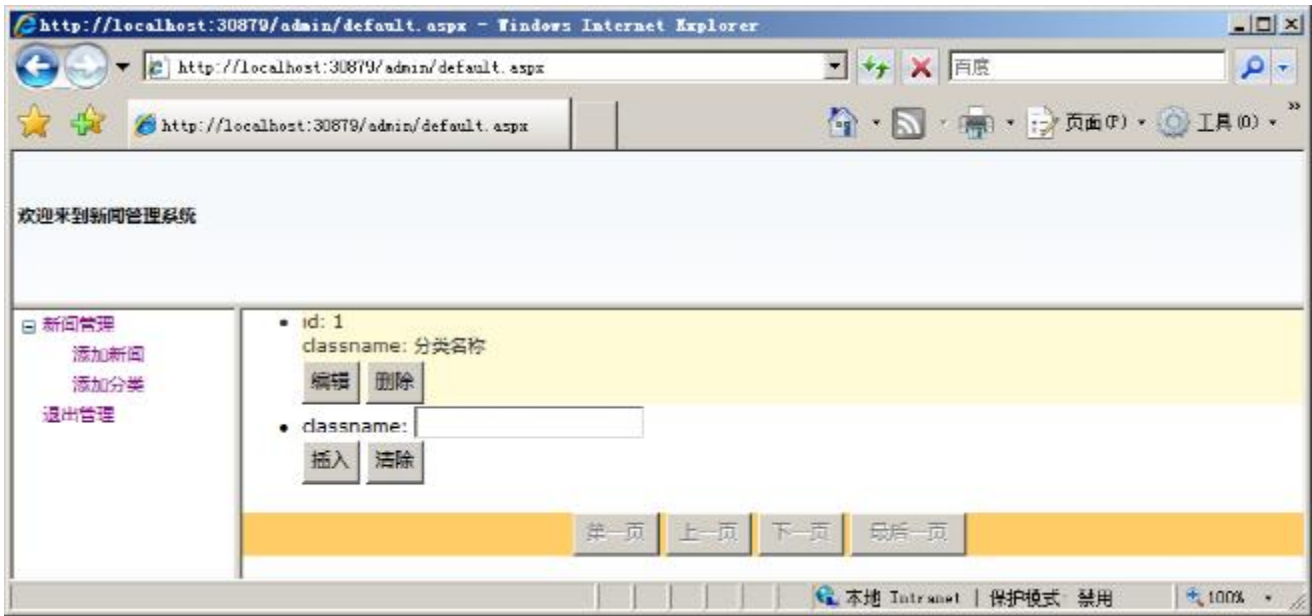


图 25-16 添加分类

分类添加完成后才能够添加新闻，如果在添加新闻之前已经有了相关分类，则没有必要再行添加，如果没有则必须添加分类。在添加分类后就能够添加新闻，这里使用了生成静态的方法生成了静态文件，虽然生成了静态文件，但是同样可以通过参数进行新闻数据的访问。添加新闻和管理新闻操作如图 25-17 和图 25-18 所示。



图 25-17 添加新闻

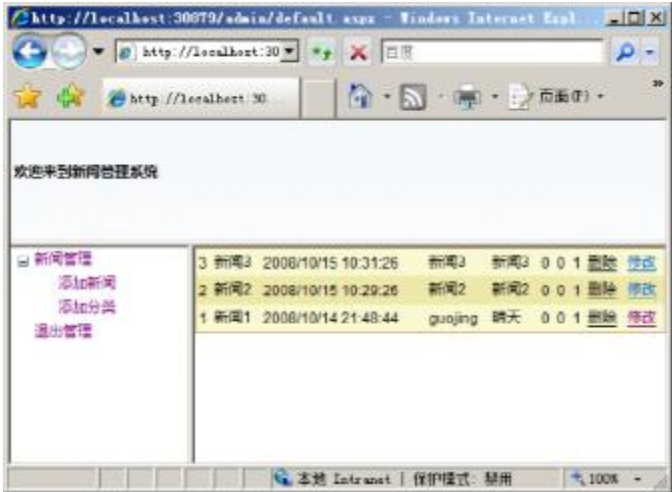


图 25-18 管理新闻



在添加完成后，就能够进行新闻的管理，新闻管理能够修改和删除新闻，修改和删除新闻都能够进行新闻模块中新闻数据的更改。当新闻被修改后，在前台所呈现的新闻样式也不同，当新闻被删除时，当用户访问该新闻则会跳转到首页。

新闻添加后，会在 **html** 文件夹中生成相应的静态文件，通过访问静态文件可以极大的减少服务器的压力，但是会增加服务器的 **IO** 读写负担，生成静态文件后同样可以通过参数的方式进行数据的访问。在 **articles.aspx** 文件中，通过传递的 **id** 参数进行数据查询和显示，虽然有些情况下需要进行数据库的操作，但是从另一方面来说减少了 **IO** 负担，如图 25-19 所示。

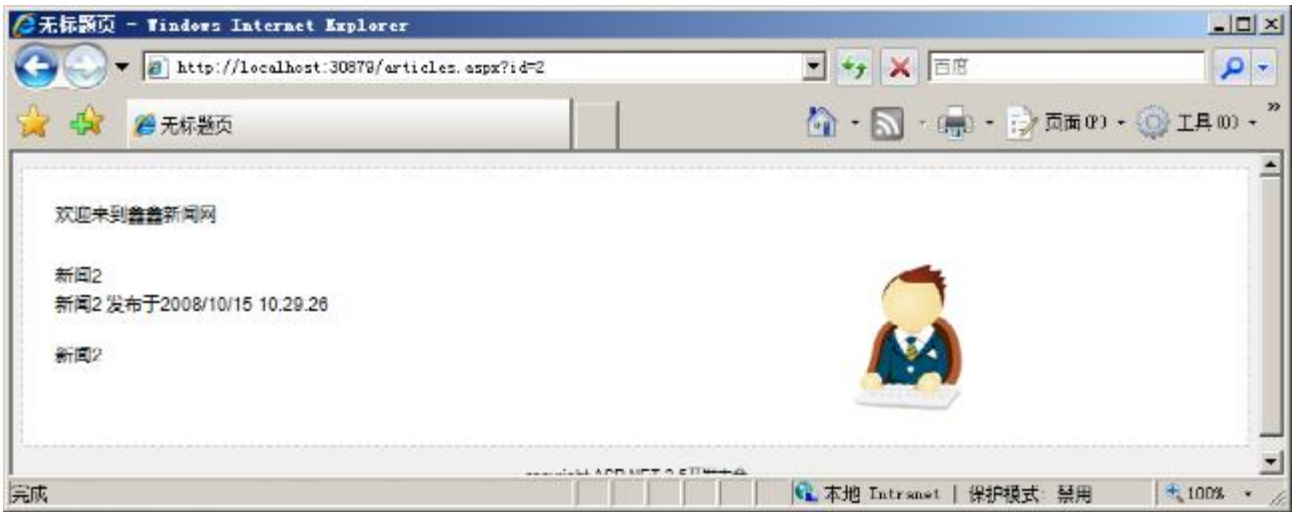


图 25-19 动态访问新闻

动态访问新闻要从数据库中读取数据，而静态访问不需要，新闻模块将静态数据文件存放在 **html** 文件夹中，访问 **html** 文件夹中的静态文件就能够访问相应的新闻，如图 25-20 所示。

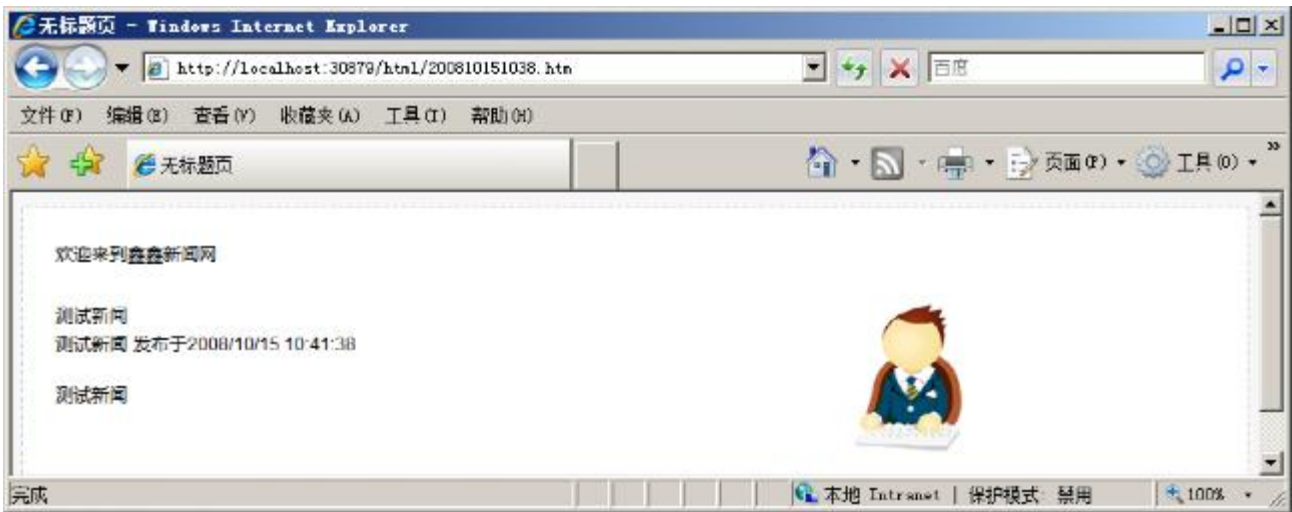


图 25-20 生成静态页面

正如图 25-20 中的地址栏所示，用户访问的是一个 **.htm** 页面而不是 **.aspx** 页面，这说明系统已经将数据进行替换并生成在 **html** 文件中，当用户访问相应的静态文件就能够访问相应的页面和数据而无需从数据中进行读取。

## 25.7 小结

本章通过讲解新闻模块的编写讲解了新闻模块的设计，动态读取和静态生成等功能。在网站中，新闻模块是必不可少的模块，对于大部分网站应用而言，新闻模块能够吸引用户，进行基础的用户信息交互。通过编写新闻模块巩固了：

- ❑ ASP.NET 的网页代码模型。
- ❑ Web 窗体基本控件。
- ❑ 数据库基础。
- ❑ ADO.NET 常用对象。

- ❑ Web 窗体数据控件。
- ❑ ASP.NET 内置对象。
- ❑ 生成静态的概念

本章讲解了生成静态的概念，以及如何生成静态，还讲解了基本的模板解析技术，在大型的网站应用开发过程中，静态生成是非常重要的也是必不可少的。但是对于中小型的网站而言生成静态有可能会造成额外的负担，所以生成静态并不是万能的，开发人员需要针对不同的应用情况进行静态生成的选择。

本章讲解了静态生成的概念但是没有详细的介绍首页中如何调用新闻列表，调用新闻列表可以使用自定义控件进行调用，这一点可以参考广告模块中广告模块的自定义控件的显示，同样也能够使用静态化的方式进行新闻列表的生成，由于篇幅限制，这里不再进行过多的讲解，其基本原理和静态生成和模板解析相同。

## 第 26 章 投票模块设计

在一些网站的应用中，为了加强用户和网站之间的交互，常常开发投票模块让用户能够参与到网站的活动，网站还能够通过投票模块进行用户信息的统计和调查，使用投票模块能够更好的让网站与用户进行交互。

### 26.1 学习要点

投票模块需要涉及到一些 **ASP.NET 3.5** 的基本知识，如果要仔细学习投票模块的开发，需要详细了解本书的一些章节知识，这些章节如下所示：

- ☐ **Web** 窗体基本控件。
- ☐ 数据库基础。
- ☐ **ADO.NET** 常用对象。
- ☐ **Web** 窗体数据控件。
- ☐ **ASP.NET** 内置对象。
- ☐ 生成静态的概念
- ☐ 自定义控件和用户控件。

基本了解了以上章节的知识点后，就能够熟练学习和开发此模块。

### 26.2 系统设计

投票模块开发起来对技术的要求并不是很高，但是投票的表设计和样式呈现都是有技巧的。在网站应用中，有些应用就需要使用投票模块，例如网站信息统计和网站信息调查等，投票模块还能够进行热点调查等。

#### 26.2.1 模块功能描述

投票模块能够加强用户与网站之间的交互，投票模块能够加强用户和网站信息之间的互动，网站可以使用投票功能进行网站内容的调查，例如调查用户是否满意网站的设计或者是否满意网站改版等等。同时投票模块还能够进行热点事件的调查，例如“你怎么看待 **XX** 事件”等等，都可以使用投票模块进行统计。

投票模块在设计上来说比较的简单，在后台的投票发布中，只需要进行相应的投票项目的发布即可，而在呈现时，需要调用多个表进行投票的呈现。例如在投票表的设计中，不能够将一个投票和选项设计在一起，那么一个投票有可能有单个选项或多个选项，不同的投票之间可能选项不同，如果将这些字段设计在一起，那么就会造成很大的数据浪费。投票模块的功能模块比较容易和简单，模块图如图 **26-1** 所示。

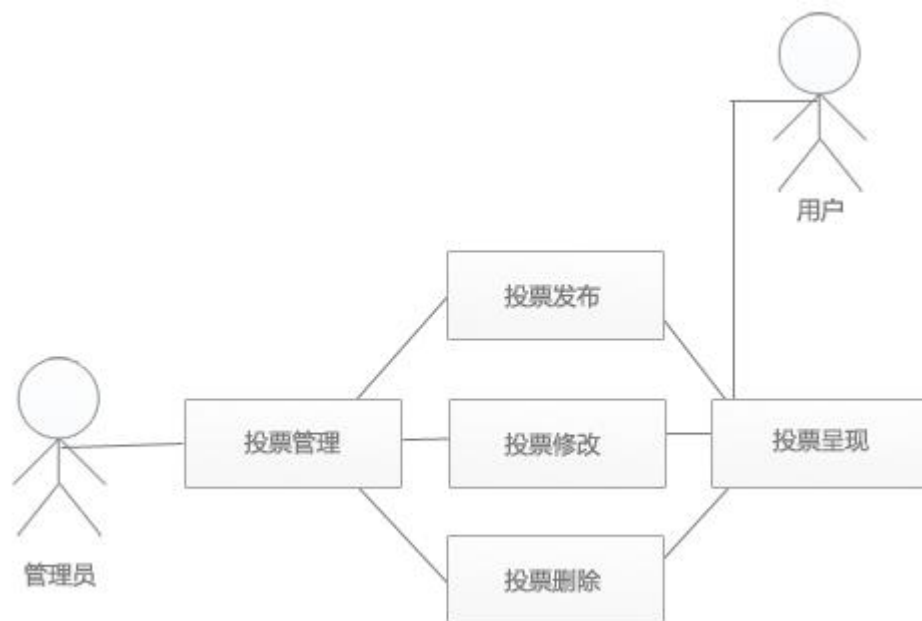


图 26-1 投票模块功能描述

正如图 26-1 所示，投票模块的功能大概可以描述为投票管理和投票呈现，管理员可以在后台进行投票管理，包括投票发布、投票修改和投票删除。投票发布或修改完毕后，用户就能够在前台进行相应的操作，包括投票和查看结果等。在前台的用户投票页面中，可以使用 **ASP.NET 3.5 AJAX** 进行无刷新操作实现用户的无刷新进行投票和结果统计。从上述模块功能描述中可以规划成以下几个页面和控件：

- ❑ 登录页面：管理员登录页面，为管理员提供身份验证。
- ❑ 后台框架集：用于管理员的管理操作。
- ❑ 投票发布页面：管理员用于投票的发布。
- ❑ 投票删除页面：管理员用于删除投票。
- ❑ 投票修改页面：管理员用于投票的修改。

其中登录页面用于制作后台的登录窗口，其作用同新闻模块中的登录一样。框架集用于制作后台管理界面，能够方便管理员在不同的页面和功能之间进行切换。

管理员首先需要进行身份验证，当身份验证通过后管理员就能够在后台进行投票的发布、删除和修改，管理员在后台进行投票操作后，用户能够在前台查看相应的投票并进行操作，投票控件可以用用户控件制作，这样能够为用户投票提供操作。而投票的查看可以使用自定义控件制作，使用自定义控件能够呈现更多的投票效果，这些效果包括统计、百分比等。

## 26.2.2 模块流程分析

虽然投票模块不是网站开发过程中最重要的模块，但是投票模块在网站开发过程中也非常的实用。现在可以看到在各个大型网站中都会有投票模块的存在，因为投票模块可以提高用于和网站之间的交互，也能够提高用户与用户之间的交互。

当出现了一个热门话题时，例如“您对番茄花园作者被抓有什么看法”时，使用新闻模块能够进行网站和用户之间的交流，但是却很难明显的看出用户的意愿，也无法统计用户的观点的信息。使用投票功能能够良好的解决这个问题，投票能够很好的进行统计，非常直观的呈现百分比，如图 26-2 所示。





图 26-2 投票模块

从图 26-2 中可以看出不同的用户的不同的观点，也能够快速的统计出大流的用户管理，例如“喜欢读书还是工作”这个问题上，可以看出很多人选择了“我现在工作了,我想回到学校去读书”这一选项。在新闻模块中虽然能够通过评论来进行用户交互，但是很多时候的效果并没有投票好。投票可以快速的统计群体对于某观点的意向，但是在投票模块中，管理员和用户进行的操作流程都是不同的，投票模块相应的流程如图 26-3 和图 26-4 所示所示。

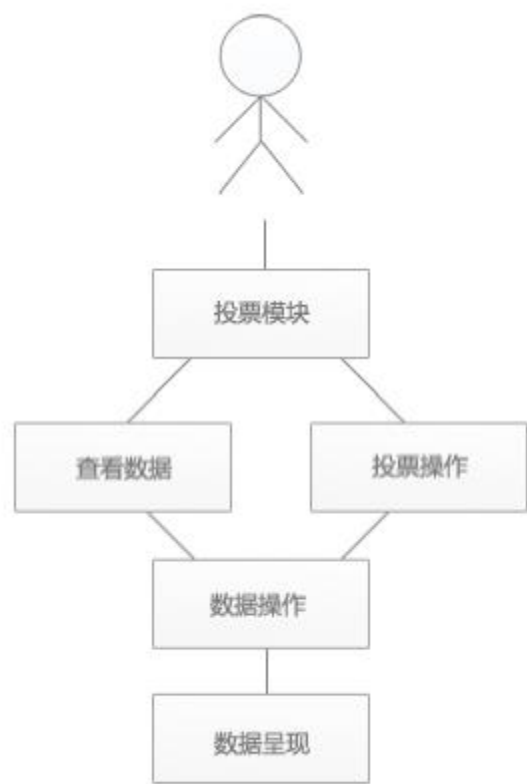


图 26-3 用户操作流程图



图 26-4 管理员操作流程图

对于用户而言，用户需要访问投票模块，在访问投票模块时可以查看投票数据然后进行投票操作，当用户进行了投票操作就会更新投票数据和相应的统计信息，系统在对数据进行重更新后再呈现在页面。而对于管理员而言，需要在后台进行投票发布和修改，在发布和修改后系统根据投票进行数据的初始化或统计并呈现在相应的页面中。

### 26.3 数据库设计

在投票模块的数据库设计中，需要多个表进行投票数据的描述。同样为了系统运行的安全，在后台同样需要登录操作，只有管理员进行登录后才能够发布投票。而对于用户，进行不同的操作时，还需要对数据库中的相应字段进行更新。

26.3.1 数据库设计

在投票模块中，需要多个表进行投票模块的描述，其中最重要的数据库就是投票问题表和投票选项表。投票问题表用于存放和投票相关的问题的数据，而投票选项表用于存放和投票选项相关的数据，在对投票模块和流程分析后，可以为几个表进行字段规划，其中投票表字段可以归纳如下。

- ❑ 投票编号：用于标识投票，为自动增长的主键。
- ❑ 投票标题：用于显示标题，作为投票模块的标题。
- ❑ 投票内容：用于解释投票信息的一些内容。

投票问题表的字段非常的少，其主要是用于索引，而投票选项表需要同投票问题表一起整合使用，一同描述一个投票模块。

- ❑ 投票选项编号：用于标识投票选项，为自动增长的主键。
- ❑ 投票选项统计：用于标识该投票被选择的次数。
- ❑ 投票描述：用于描述投票中的一个选项。
- ❑ 投票选项 ID：用于标识该投票选项是隶属于哪个投票问题，为投票问题表的外键。

管理员表用于管理员登录和验证操作，其作用同登录模块和新闻模块中的登录模块基本相同，其字段如下所示。

- ❑ 管理员编号：用于标识管理员信息，为自动增长的主键。
- ❑ 管理员用户名：用于标识管理员用户名。
- ❑ 管理员密码：用于标识管理员的密码，通常情况下和管理员用户名一起进行身份验证。

其中投票问题表和投票选项表一同描述投票模块，如图 26-5 所示。

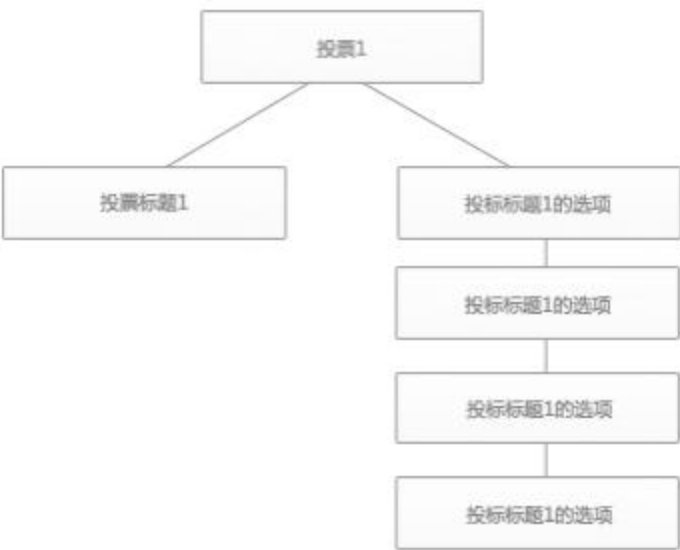


图 26-5 投票模块的形成

投票问题表和投票选项表一同描述投票模块，在呈现一个投票项目时，在投票问题表中只需要显示投票的问题，而投票的选项则需要通过投票问题的标识（ID）号来进行筛选，在投票选项表中有一个投票选项 ID 字段，该字段就是用于标识这个选项是属于哪个投票问题的。只有将这两个表中的数据进行整合才能够完整的实现一个投票所需要的数据表。投票模块的数据设计并不复杂，而投票模块中将不同的表的数据进行呈现却比数据设计更加复杂的。

26.3.2 数据表的创建

创建表可以通过 SQL Server Management Studio 视图进行创建也可以通过 SQL Server Management Studio 查询使用 SQL 语句进行创建。在创建表之前首先需要创建数据库 post，在 post 数据库中只需要创建 3 个表就能够实现投票项目的描述，其中投票表（posttitle）的字段如图 26-6 所示。

	列名	数据类型	允许空
	id	int	<input type="checkbox"/>
	title	nvarchar(50)	<input checked="" type="checkbox"/>
	[content]	nvarchar(MAX)	<input checked="" type="checkbox"/>
			<input type="checkbox"/>

图 26-6 投票表结构图

从投票表的结构图可以看出只需要创建三个字段就能够表述投票问题，这三个字段的意义如下所示。

- ❑ **id:** 用于标识投票，为自动增长的主键。
- ❑ **title:** 用于显示标题，作为投票模块的标题显示。
- ❑ **content:** 用于解释投票信息的一些内容。

创建表的 SQL 语句如下所示。

```
USE [post]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[posttitle](                                //创建 posttitle 表
    [id] [int] IDENTITY(1,1) NOT NULL,
    [title] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
    [content] [nvarchar](max) COLLATE Chinese_PRC_CI_AS NULL,
    CONSTRAINT [PK_posttitle] PRIMARY KEY CLUSTERED
(
    [id] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
    ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
```

上述代码创建了投票表并创建了相应的字段，为了配合投票问题表，还需要创建投票选项表，投票选项表的字段如下所示。

- ❑ **id:** 用于标识投票选项，为自动增长的主键。
- ❑ **hits:** 用于标识该投票被选择的次数。
- ❑ **content:** 用于标识投票项及其描述。
- ❑ **askid:** 用于标识该投票选项是隶属于哪个投票问题，为投票问题表的外键。

这里值得注意的是，选项表中的 **askid** 字段同投票表一同描述投票项，投票选项表创建的 SQL 语句如下所示。

```
USE [post]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[postchoose](                                //创建 postchookse 表
    [id] [int] IDENTITY(1,1) NOT NULL,
    [hits] [int] NULL,
    [content] [nvarchar](max) COLLATE Chinese_PRC_CI_AS NULL,
    [askid] [int] NULL,
    CONSTRAINT [PK_postchoose] PRIMARY KEY CLUSTERED
(
    [id] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
    ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
```

上述代码创建了 **postchookse** 表用于描述投票的选项，其中 **asdid** 为投票问题表的 **id**，该键为外键，用于筛选和整合两个表中的数据。在投票的发布中，为了系统的安全性，还需要创建系统表用于管理员登录，**admin** 表只需要保存管理员的用户名和密码即可，则其字段可以描述为如下所示。

- ❑ **id**: 用于标识管理员信息，为自动增长的主键。
- ❑ **admin**: 用于标识管理员用户名。
- ❑ **password**: 用于标识管理员的密码，通常情况下和管理员用户名一起进行身份验证。

上述字段描述了 **admin** 表中需要使用的字段，可以使用 **SQL** 语句进行表和字段的创建，创建 **newsclass** 表的 **SQL** 语句如下所示。

```
USE [post]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[admin](                                //创建 admin 表
    [id] [int] IDENTITY(1,1) NOT NULL,
    [admin] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
    [password] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
    CONSTRAINT [PK_admin] PRIMARY KEY CLUSTERED
(
    [id] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
    ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
```

上述代码创建了 **admin** 表，用于进行管理员的身份验证，在后台的管理员登录中需要使用到该表，在表创建完成后，需要向数据库中添加管理员，添加管理员代码如下所示。

```
INSERT INTO admin (admin,password) VALUES ('guojing','0123456')
```

执行上述代码就能够进行 **admin** 表的数据插入，插入一个新管理员之后，就能够在后面的登录操作中使用该表的管理员信息进行登录和投票操作。

## 26.4 界面设计

投票模块包括众多的页面，这些页面包括发布投票页面、修改投票页面和删除投票页面，这其中还包括投票管理页面。在投票模块中，不适用进行 **ASP.NET** 自带的控件进行操作，在投票模块的页面设计中需要自行开发自定义页面。

### 26.4.1 后台框架集

在 **Microsoft Expression Web 2** 中，选择菜单栏中的【文件】选项，在下拉菜单中选择【新建】选项，单击【新建】选项中的【网站】选项创建网站，在弹出窗口中选择框架集，如图 26-7 所示。



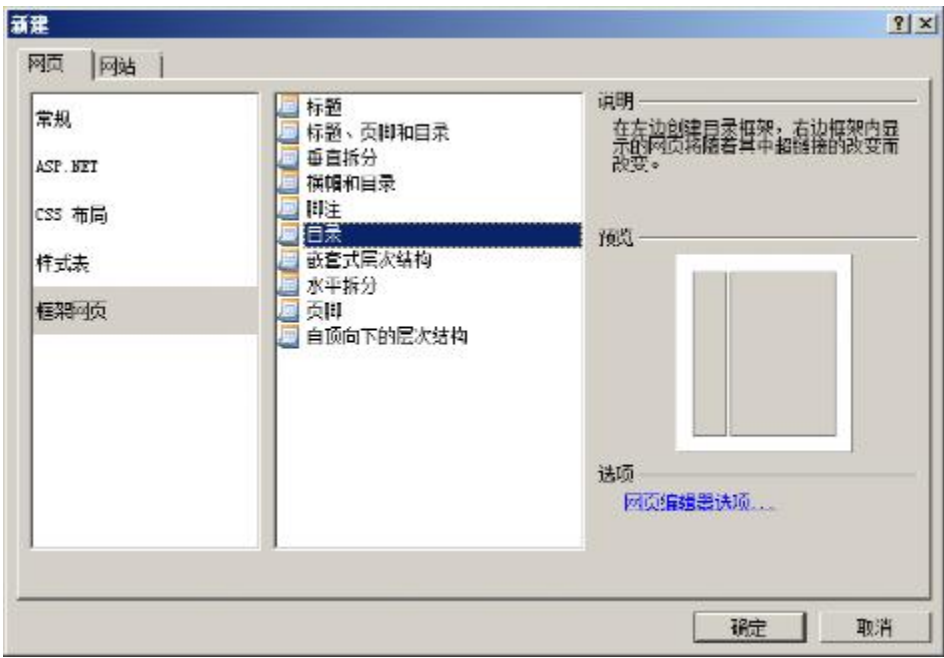


图 26-7 创建框架集

框架集可以将多个页面放置在同一个页面，在 **Microsoft Expression Web 2** 中可以创建框架集并为框架集中的页面进行指定或新建，这里可以创建一个目录类型的框架集，创建后框架集代码见光盘中源代码第 26 章\26-1\26-1\admin\default.aspx。

开发人员可以在框架集中创建网页或选择设置初始网页，这里创建两个网页，一个用于显示导航，此页面为 **sidebar.aspx**，管理员能够在该页面进行导航处理。另一个则是主工作区，用于管理员操作，导航栏初始化代码见光盘中源代码第 26 章\26-1\26-1\admin\sidebar.aspx。

投票管理的操作并不多，所以在后台中管理员导航的节点也并不多，管理员在后台主要执行投票管理和添加投票两个操作，投票管理操作包括了投票的修改和删除。

26.4.2 投票管理页面

投票管理页面的数据显示可以使用 **ASP.NET 3.5** 提供的 **GridView** 控件，由于投票模块其功能的复杂性，这里并不能使用 **ASP.NET 3.5** 自带的控件进行修改，所以在投票管理页面只能够使用 **GridView** 控件进行数据罗列，并添加相应的超链接，示例代码见光盘中源代码第 26 章\26-1\26-1\admin\manage.aspx。

在管理页面中，其代码使用了 **GridView** 控件进行数据罗列，但在 **GridView** 控件中添加的是两个数据绑定的超链接分别用于实现修改和删除功能，如图 26-8 和图 26-9 所示。



图 26-8 添加修改超链接



图 26-9 添加删除超链接

可以看出修改和删除超链接都会通过参数传递跳转到另一个页面，其中修改超链接跳转的是 **modi.aspx**？

id=编号，而删除超链接跳转的是 delete.aspx?id=编号。通过参数的传递，开发人员能够在相应的页面进行业务逻辑处理。

注意：这种情况很像 ASP 的开发过程，为了能够让开发人员更好的理解控件的制作，可以使用类似 ASP 的开发流程进行过程开发。

由于页面只需要进行数据的呈现，所以数据源并不需要支持数据操作，数据源代码如下所示。

```
<asp:SqlDataSource ID="SqlDataSource1" runat="server"
    ConnectionString="<%= $ConnectionStrings:postConnectionString %>"
    SelectCommand="SELECT * FROM [posttitle] ORDER BY [id] DESC">
</asp:SqlDataSource>
```

从上述代码可以看出，数据源中并没有自动支持插入、更新、删除等数据操作，因为对于投票管理页面的 GridView 控件而言，其作用只是用于数据的呈现，而无需进行数据操作。

26.4.3 投票发布页面

投票发布页面用于管理员的投票发布，管理员可以在投票发布页面进行投票信息的填写，在投票发布页面，管理员需要填写投票信息和投票选项信息，这就涉及到多个表的数据操作。在管理员填写投票选项信息时，需要向投票选项表中重复插入数据，因为投票选项往往是多项，而投票问题只是一项。投票发布页面涉及到两个表和程序筛选，将在后面的章节中讲到，而投票发布页面的设计只需要进行基础控件布局即可，示例代码见光盘中源代码第 26 章\26-1\26-1\admin\post.aspx。

投票代码使用了 TextBox 控件用于管理员的信息输入，管理员能够在相应的控件中填写投票信息并通过按钮控件提交数据。在投票发布页面中，投票选项同样使用的是 TextBox 控件。再次回到流程分析中，用户在前台进行投票。投票同样包含多个选项，有的投票包含 1 个或 3 个选项，有的投票包含 2 个或 4 个选项，那么在投票功能中就不能规定死投票选项的个数。

在投票发布时，管理员需要按照投票的选项个数进行投票发布，同时在投票中又需要降低数据库的使用率，这里就需要智能筛选数据。这里使用的是回车筛选，即一个回车就是一个选项。为投票功能设计页面以便管理员添加投票项目，页面设计后如图 26-10 所示。



图 26-10 投票页面设计

26.4.4 投票修改页面

投票修改页面同投票发布页面相同，在投票页面被加载时，投票的基本信息需要载入并存储在投票修改页面中的控件里，当管理员单击【修改投票】按钮时，就能够进行数据筛选和修改，投票修改页面示例

代码见光盘中源代码\第 26 章\26-1\26-1\admin\modi.aspx。

投票修改页面在加载时接受传递过来的参数，使用传递的参数获取数据库中投票的相应信息进行页面中控件的文本填充。管理员可以通过修改页面中相应的信息进行数据更改，当更改完毕后管理员可以进行数据操作更新相应的数据选项。

## 26.4.5 投票删除页面

投票删除页面可以不进行页面布局的处理，因为投票删除页面主要作用为删除数据。当管理员在投票管理页面进行投票删除选择时，会跳转到投票删除页面，投票删除页面通过获取传递过来的参数进行投票的删除，删除完毕后再次返回投票管理页面，所以在投票删除页面中只需要进行事务的处理而不需要进行页面布局和控件处理。

## 26.5 代码实现

投票模块的页面整体设计比较简单，但是投票功能的实现还需要掌握一些难点。投票模块的难点就在于一个投票的问题和多个投票选项是如何整合，并且在发布投票时如何进行多项投票发布，在修改投票时同样修改选项和筛选选项都是一个难题。

### 26.5.1 添加投票代码实现

添加投票时，管理员在投票选项中可以用回车进行分割，系统能够使用回车进行多个选项筛选和数据插入。在代码实现中，首先要将投票选项控件中的文本进行分割，这里使用回车进行分割，每个回车就会创建一个新的投票项。添加投票的具体实现思路为：

- ❑ 管理员在投票控件中按回车进行多个选项添加。
- ❑ 管理员单击按钮控件进行数据添加。
- ❑ 数据添加时按照回车进行选项分割，当有多个回车时说明有多个选项。
- ❑ 添加投票问题表，添加完成后获取刚刚添加的投票的编号。
- ❑ 循环添加投票选项，并为选项添加相应的 ID。

从上述实现思路可以编写相应的代码，示例代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    SqlConnection con = new SqlConnection("Data Source=(local);
        Initial Catalog=post;Integrated Security=True"); //创建连接
    con.Open(); //打开连接
    string trim = TextBox3.Text.Replace("\n", "|"); //回车替换成字符
    string[] count = trim.Split('|'); //替换后分拆
    string strsql = "insert into posttitle (title,content) values ('"+TextBox1.Text+"','"+TextBox2.Text+"')
        SELECT @@IDENTITY as 'bh' "; //编写 SQL 语句
    SqlCommand cmd = new SqlCommand(strsql, con); //创建执行对象
    int id=Convert.ToInt32(cmd.ExecuteScalar()); //执行数据插入
    for (int i = 0; i < count.Length; i++) //循环添加
    {
        string contentinsert = "insert into postchoose (hits,content,askid) values
            ('0','" + count[i].ToString() + "','" + id + "')"; //遍历插入项目
        SqlCommand command = new SqlCommand(contentinsert, con); //创建执行对象
        command.ExecuteNonQuery(); //执行遍历
    }
}
```



```
con.Close();                                     //关闭连接
}
```

上述代码中将管理员输入的投票选项通过回车分割，添加选项方法如图 26-11 所示。



图 26-11 输入投票选项

管理员输入投票选项时，可以以回车的形式进行投票选项的分割，如图 26-11 所示，其中管理员添加了 6 个选项，在数据插入前，首先需要将不同的投票选项之间分割，示例代码如下所示。

```
string trim = TextBox3.Text.Replace("\n", "|");           //替换字符
string[] count = trim.Split('|');                         //分割字符串
```

上述代码将回车字符串替换成为“|”符号，然后通过 Split 将字符串进行分割，在图 26-11 中输入的字符串会被分割为“投票选项 1|投票选项 2|投票选项 3..”，Split 函数能够将这个字符串进行分割并放置在数组中，如“count[0]=投票选项 1，count[1]=投票选项 2”，这样就能够通过数组循环进行数据插入，示例代码如下所示。

```
for (int i = 0; i < count.Length; i++)                   //遍历执行
{
    string contentinsert =
        "insert into postchoose (hits,content,askid) values ('0','" + count[i].ToString() + "','" + id + "')";
    SqlCommand command = new SqlCommand(contentinsert, con); //SQL 语句
    command.ExecuteNonQuery();                             //执行 SQL
}
```

使用 ADO.NET 中数据操作的 ExecuteScalar 方法和 SQL 语句的“SELECT @@IDENTITY as”语法能够返回刚才数据插入的值，在进行选项遍历插入之前，首先需要得到刚才插入的投票问题的编号。使用 ExecuteScalar 方法能够获取刚刚插入的数据的编号，获取后就需要在遍历操作中为相应的选项指定其问题的 ID 号。执行一次操作后其表之间的关系如图 26-12 和图 26-13 所示。

	id	title	content
▶	1	投票1	投票简介
✕	NULL	NULL	NULL

图 26-12 投票问题表

	id	hits	content	askid
	1	0	投票选项1	1
	2	0	投票选项2	1
	3	0	投票选项3	1
	4	0	投票选项4	1
	5	0	投票选项5	1
	6	0	投票选项6	1
*	NULL	NULL	NULL	NULL

图 26-13 投票选项表

从图 26-12，26-13 可以看出投票问题表和投票选项表之间一起描述了一个投票项目，通过回车的方式能够使不同的投票项目中的投票问题和投票选项之间分离开，这样投票项目与投票项目之间就没有选项等约束，一个投票可以有多个选项，不同的投票之间投票选项个数可以不同，其统计和描述也可以不同。但是这样进行数据库设计同样也会有一些缺点，其缺点就是在投票选项表中可能会占用过多的数据。



## 26.5.2 修改投票代码实现

在修改投票代码页面载入时，首先同样需要和投票插入一样进行数据筛选。一个投票项目需要两个数据表进行描述，在页面加载时需要加载多个表中的数据，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack) //判断是否第一次加载
    {
        TextBox3.Text = ""; //清空控件值
        Label1.Text = ""; //清空控件值
        SqlConnection con = new SqlConnection("Data Source=(local);
                                             Initial Catalog=post;Integrated Security=True"); //创建连接
        con.Open(); //打开连接
        string strsql = "select * from posttitle where id=" + Request.QueryString["id"] + "";
        SqlDataAdapter da = new SqlDataAdapter(strsql, con);
        DataSet ds = new DataSet(); //创建数据集
        da.Fill(ds, "table"); //填充投票数据
        string strchoose = "select * from postchoose where askid=" + Request.QueryString["id"] + "";
        SqlDataAdapter ch = new SqlDataAdapter(strchoose, con);
        int count = ch.Fill(ds, "choosetable"); //填充投票选项
        TextBox1.Text = ds.Tables["table"].Rows[0]["title"].ToString(); //填充控件
        TextBox2.Text = ds.Tables["table"].Rows[0]["content"].ToString(); //填充控件
        for (int i = 0; i < count; i++) //循环获取数据
        {
            if (i == count - 1) //判断是否循环完
            {
                TextBox3.Text += ds.Tables["choosetable"].Rows[i]["content"].ToString().Replace("\r", "");
                Label1.Text += ds.Tables["choosetable"].Rows[i]["id"].ToString(); //填充数据
            }
            else
            {
                TextBox3.Text += ds.Tables["choosetable"].Rows[i]["content"].ToString() + "\n";
                Label1.Text += ds.Tables["choosetable"].Rows[i]["id"].ToString() + "|"; //填充数据
            }
        }
        con.Close(); //关闭连接
    }
}
```

上述代码实现了当页面加载时其相应的数据被加载的相应的控件的功能，其中比较复杂的就在于两个数据表中的数据如何整合在一起。在投票选项中，其选项需要整合在一起放置在控件中，同时还需要像添加投票一样一行一个投票项目。

不仅如此，还需要将这些投票项目的编号进行统计，如果不进行统计则不能够使用编号进行相应的数据字段的更新，只有保存相应的选项的数据的编号才能够循环更新。在添加投票时，将选项与选项之间进行回车分割，在载入页面时，同样需要将数据呈现为选项与选项之间的回车形式，示例代码如下所示。

```
for (int i = 0; i < count; i++) //循环获取数据
{
    if (i == count - 1) //判断是否循环完
    {
        TextBox3.Text += ds.Tables["choosetable"].Rows[i]["content"].ToString().Replace("\r", "");
        Label1.Text += ds.Tables["choosetable"].Rows[i]["id"].ToString(); //填充数据
    }
    else
    {
        TextBox3.Text += ds.Tables["choosetable"].Rows[i]["content"].ToString() + "\n";
    }
}
```

```
        Label1.Text += ds.Tables["choosetable"].Rows[i]["id"].ToString() + "|"; //填充数据
    }
}
```

上述代码将投票选项的数据和相应的编号进行填充，在进行更新时同样可以使用控件中的数据数据进行数据分割操作然后进行数据更新，更新代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    SqlConnection con = new SqlConnection("Data Source=(local);
        Initial Catalog=post;Integrated Security=True"); //创建连接
    con.Open(); //打开连接
    string trim = TextBox3.Text.Replace("\n", "|"); //替换字符
    string[] count = trim.Split('|'); //分割字符
    string[] count2 = Label1.Text.Split('|'); //计算分割
    if (count != count2) //判断项数
    {
        Label2.Text = "修改不能修改投票项数";
    }
    else
    {
        string strsql = "update posttitle set title='" + TextBox1.Text + "',content='" + TextBox2.Text + "'
            where bh='" + Request.QueryString["id"] + "'"; //编写更新语句
        SqlCommand cmd = new SqlCommand(strsql, con); //创建执行对象
        cmd.ExecuteNonQuery(); //执行对象
        for (int i = 0; i < count.Length; i++) //遍历更新
        {
            strsql = "update postchoose set content='" + count[i].ToString() + "' where id='" +
                count2[i].ToString() + "'"; //生成 SQL 语句
            SqlCommand cmd1 = new SqlCommand(strsql, con); //更新数据
            cmd1.ExecuteNonQuery(); //执行更新
        }
        con.Close(); //关闭连接
    }
}
```

在 **Page\_Load** 代码中，将选项的编号都存放到 **Label** 控件中，在执行更新时，需要分别循环遍历选项控件和 **Label** 控件进行数据更新。在数据更新中，并不能修改投票的项数，这也就是说在发布投票时有多少选项则在修改时只能够修改相应的选项而不能增加选项或删除选项。

### 26.5.3 删除投票代码实现

在投票项目删除时，并不能像前面的模块一样只对模块问题项目进行删除。在删除模块问题表时，同样需要删除投票的选项，这样就能够保证数据库的完整性。因为如果对投票的问题进行删除而不对投票的选项进行删除的话，会造成大部分的垃圾数据，同样也会影响到程序的性能。删除投票通过传递的参数进行两个表中相应的数据的删除，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    SqlConnection con = new SqlConnection("Data Source=(local);
        Initial Catalog=post;Integrated Security=True"); //创建连接
    con.Open(); //打开连接
    string deletetitle = "delete posttitle where id='"+Request.QueryString["id"]+"'"; //删除
    string deletechoose = "delete postchoose where askid='"+Request.QueryString["id"]+"'"; //删除
    SqlCommand cmd = new SqlCommand(deletetitle, con);
    cmd.ExecuteNonQuery(); //执行删除
    SqlCommand cmd1 = new SqlCommand(deletechoose, con);
    cmd1.ExecuteNonQuery(); //执行删除
}
```

```
Response.Redirect("manage.aspx");
```

```
//页面跳转
```

```
}
```

上述代码分别删除了两个表中的数据，在进行删除操作时首先需要对数据库中的投票问题表中的数据删除，删除后还需要删除与投票问题表相关的投票选项，其中投票选项中的 **askid** 字段用于标识选项所对应的问题，只有执行了两个表中相应的数据的删除才能够真正删除一个投票项目。

#### 26.5.4 显示投票代码实现

当管理员发布代码后，就需要进行代码的显示，代码显示同样需要获取两个表中相应的投票项的信息，在显示过程，还需要对投票中的数据进行显示和图表设计。

在投票显示之前，首先需要遍历投票的选项并且计算投票选项的总和，计算完成投票的总和后，再分别计算每个选项被用户选择的次数。例如有一个投票项目名为“你怎么看待上大学”，其中选项有“上学好，比工作轻松”和“上学没自由”等等选项，首先就需要统计所有的选数的总和，假设是 **100** 票，然后再统计选择“上学好，比工作轻松”的票数，进行相应的统计。在统计时，需要遍历数据库中的数据进行计算，计算完成后可以通过 **HTML** 呈现给用户，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    SqlConnection con = new SqlConnection("Data Source=(local);
                                           Initial Catalog=post;Integrated Security=True");
    con.Open();
    string title;
    string id = Request.QueryString["id"];
    string width = "500px";
    string str = "select * from posttitle where id='" + id + "'";
    SqlDataAdapter da = new SqlDataAdapter(str, con);
    DataSet ds = new DataSet();
    da.Fill(ds, "table");
    string str2 = "select * from postchoose where askid='" + id + "'";
    SqlDataAdapter da2 = new SqlDataAdapter(str2, con);
    DataSet ds2 = new DataSet();
    da2.Fill(ds2, "table2");
    title = ds.Tables["table"].Rows[0]["title"].ToString();
    int sum = 0;
    for (int i = 0; i < ds2.Tables["table2"].Rows.Count; i++)
    {
        sum += Convert.ToInt32(ds2.Tables["table2"].Rows[i]["hits"].ToString());
    }
    Response.Write("<table style='width:" + width + "'><tr><td style='border: #4a95c9
                    1px solid;background-color: #b7d8ed;padding:5px 5px 5px 5px;'>");
    Response.Write("<strong>" + title + "</strong><br />");
    for (int i = 0; i < ds2.Tables["table2"].Rows.Count; i++)
    {
        Response.Write("<br/><span style='font-size:12px;'>" + (i + 1) +
            ".<a href='newvote.aspx?id=" + ds2.Tables["table2"].Rows[i]["id"].ToString()
            + "&askid=" + id.ToString() + "'#vote' target='_blank'>" +
            ds2.Tables["table2"].Rows[i]["content"].ToString().Replace("\n", "").Replace("\r", "") +
            "</a>票数:" + ds2.Tables["table2"].Rows[i]["hits"].ToString() + "百分比:");
        if (sum != 0)
        {
            Response.Write(Convert.ToSingle(ds2.Tables["table2"].Rows[i]["hits"].ToString())
                / Convert.ToSingle(sum) * 100 + "%");
        }
        else
        {

```

```

        Response.Write("0%"); //输出 0%
    }
    Response.Write("</span><br/><br/> <div style='height:10px; background-color:White;
border:1px solid #ccc;'><div style='height:10px; background-color:gray; width:"); //输出边框
    if (sum != 0) //总和不为 0
    {
        Response.Write(Convert.ToInt32(Convert.ToSingle(ds2.Tables["table2"].Rows[i]["hits"].ToString())
/ Convert.ToSingle(sum) * 100) + "%"); //输出百分比
    }
    else
    {
        Response.Write("0"); //输出票数
    }
    Response.Write("<</div></div>"); //输出 div
}
Response.Write("<br /><strong>总票数</strong>: " + sum + ""); //输出总票数
Response.Write("</td></tr></table>");
}

```

当用户通过访问 **vote.aspx** 页面时，就能够查看到相应的投票信息，投票信息页面中包括投票名称、投票选项、票数和百分比，这些信息能够方便的让投票者或用户了解到大众的相关信息。上述代码首先读取投票的基本信息，包括投票的标题和投票的描述，读取完毕后再进行投票选项的遍历，遍历过后进行票数的统计和百分比的计算，然后再通过 **HTML** 代码呈现在页面中。

## 26.5.5 用户投票代码实现

用户可以看见投票项目和投票统计，同样用户也应该能够进行投票，当用户投票之后就需要记录用户对投票进行的操作，才能够让用户不能重复投票。如果用户重复投票就会造成投票的不真实性，从显示的投票页面可以看出，投票页面使用了参数进行另一个页面的传递，示例代码如下所示。

```

Response.Write("<br/><span style='font-size:12px;'>" + (i + 1) +
".<a href='newvote.aspx?id=" + ds2.Tables["table2"].Rows[i]["id"].ToString() + "&askid="
+ id.ToString() + "#vote' target='_blank'>" +
ds2.Tables["table2"].Rows[i]["content"].ToString().Replace("\n", "").Replace("\r", "") + "</a>票数:"
+ ds2.Tables["table2"].Rows[i]["hits"].ToString() + "百分比:");

```

从上述代码中可以看出，当用户单击项目进行投票时，系统会通过参数 **id** 和 **askid** 进行传递跳转到另一个 **newvote.aspx** 页面并在 **newvote.aspx** 页面进行逻辑处理。逻辑处理完毕后再跳转回该页面，**newvote.aspx** 页面代码如下所示。

```

protected void Page_Load(object sender, EventArgs e)
{
    try
    {
        if (Request.Cookies[Request.QueryString["id"]] == null) //判断是否已经投票
        {
            SqlConnection con =
            new SqlConnection("Data Source=(local);Initial Catalog=post;Integrated Security=True");
            con.Open(); //打开连接
            string askid = Request.QueryString["askid"]; //获取 askid
            string id = Request.QueryString["id"]; //获取 id
            string str = "select * from postchoose where id='" + id + "'"; //生成 SQL 语句
            SqlDataAdapter da = new SqlDataAdapter(str, con); //创建适配器
            DataSet ds = new DataSet(); //创建数据集
            da.Fill(ds, "table"); //填充数据集
            int hits = Convert.ToInt32(ds.Tables["table"].Rows[0]["hits"].ToString()); //获取点数
            string strsql = "update postchoose set hits='" + (hits + 1) + "' where id='" + id + "'"; //增加点击
            SqlCommand cmd = new SqlCommand(strsql, con); //创建执行对象

```



```
cmd.ExecuteNonQuery(); //更新数据操作
HttpCookie cookies_votenum = new HttpCookie(id); //创建 Cookie
cookies_votenum.Value = id; //设置 Cookie 值
cookies_votenum.Expires = DateTime.Now.AddDays(1); //设置持续事件
Response.AppendCookie(cookies_votenum); //添加 Cookie
Response.Redirect("vote.aspx?id=" + askid + ""); //跳转投票
}
else
{
    Label1.Text = "您已经参与投票"; //输出用户信息
}
}
catch(Exception ee)
{
    Response.Write(ee.ToString()); //抛出异常
}
}
```

上述代码通过传递的参数进行数据更新，该页面进行投票的事务处理，当用户单击投票系统中的项目时，系统会跳转到该页面进行数据更新并跳转回相应的投票项目，返回后用户就能够查看相应的数据信息。

当用户投票后，系统会为用户提供一个 **Cookie** 对象，用户再次执行投票后首先会检查这个 **Cookie** 对象，如果存在 **Cookie** 对象说明用户已经投过票，则不允许用户再次投票，如果没有 **Cookie** 对象则用户能够进行投票操作。

26.6 实例演示

当使用投票模块进行网站和用户之间的交互时，管理员能够在后台发布投票信息并在前台进行投票信息的呈现，用户能够在前台投票页面中进行投票和数据查看。投票能够清晰明显的显示项目，让用户能够较直观的获取信息。管理员可以在后台页面中进行投票发布，当投票项有多个选项时，管理员可以用回车进行分割，如图 26-14 所示。

管理员添加了一个“你喜欢本书吗”的调查，并通过回车创建了三个选项，这三个选项分别为“喜欢, 本书言简意赅”，“还可以,就是知识不够广泛”和“不喜欢,写的太烂了”，创建后这三个选项都会被添加到相应的表中。单击按钮控件，管理员能够创建投票，管理员可以在后台的投票管理页面中进行投票信息的管理，如图 26-15 所示。



图 26-14 添加投票信息



图 26-15 投票管理

管理员能够在投票管理页面进行投票的修改和删除，修改投票可以修改其中的选项、投票标题和简介。如果管理员发布的投票信息并没有什么错误的话，可以无需修改投票信息就可以进行页面呈现，这里可以访问 **vote.aspx?id=相应 ID** 页面进行投票信息的访问，如图 26-16 所示。

用户能够访问页面进行投票信息的查看并进行投票，当用户对一个项目进行投票后，就不能够再对某个项目进行投票，如果用户投票后，系统会给用户一个 **Cookie** 对象，当用户投票前首先会检查 **Cookie** 对象，如果不存在 **Cookie** 对象则允许用户进行投票。投票后的页面如图 26-17 所示。

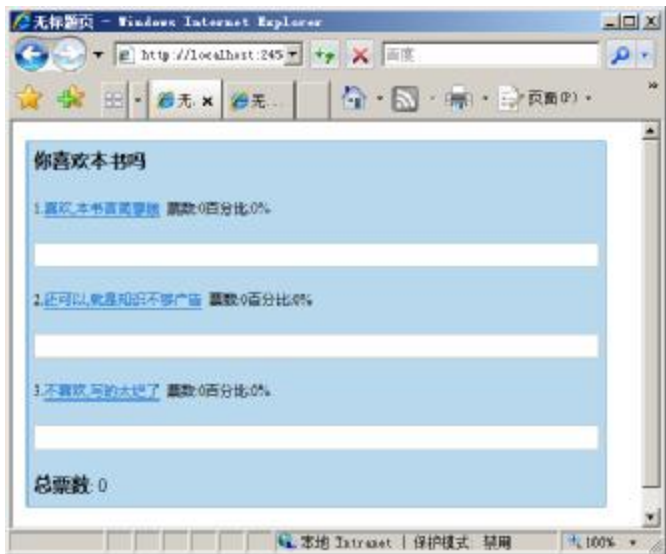


图 26-16 投票信息

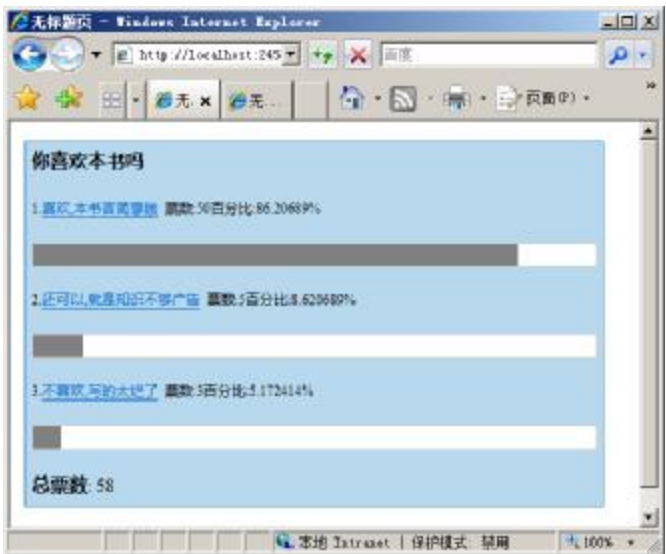


图 26-17 投票统计

当用户进行投票后，投票页面会统计用户投票的信息，并以图表的形式呈现给用户，用户能够非常直观的查看到其他用户的投票信息，例如这里“喜欢,本书言简意赅”选项能够一眼就看出选择这个用户的人数比较多，这样就能够非常清除的了解“你喜欢本书吗”这个问题的结果。

26.7 小结

本章通过投票模块的编写了解了数据库的设计，以及基本的投票设计开发，在投票模块的开发过程中，使用了两个数据库进行投票项目的描述，当对投票模块的数据插入、更新和删除时，都需要考虑到数据的完整性。本章还巩固了：

- ❑ Web 窗体基本控件。
- ❑ 数据库基础。
- ❑ ADO.NET 常用对象。
- ❑ Web 窗体数据控件。
- ❑ ASP.NET 内置对象。
- ❑ 生成静态的概念
- ❑ 自定义控件和用户控件。

开发人员能够将投票页面更改成为 **JavaScript**，这样就能够不同的页面进行调用，投票是网站开发中一个比较常用的功能，现在的很多的大型网站都包含投票模块用于与用户进行信息交互。

本章还包括了管理员登录等页面，由于前面的章节中详细的讲解了管理员登录的开发方式和登录控件的开发，这里就不再详细的介绍，管理员登录等基本功能在后台开发中是非常重要的，只要在后台包含管理员登录操作，就需要开发验证或后台登录页面。

## 第 27 章 聊天模块设计

在一些网站应用中，常常需要在线聊天模块的设计，例如网站的客户服务。聊天模块的设计往往不需要使用到数据库，因为在线聊天是一种即时的行为，当用户离开在线聊天页面时，可能无需进行用户信息的保存。

### 27.1 学习要点

聊天模块需要涉及到一些 **ASP.NET 3.5** 的基本知识，如果要仔细学习聊天模块的开发，需要详细了解本书的一些章节知识，这些章节如下所示：

- **Web** 窗体基本控件。
- **Web** 窗体数据控件。
- **ASP.NET** 内置对象。
- 生成静态的概念
- 自定义控件和用户控件。
- **ASP.NET 3.5** 与 **AJAX**

基本了解了以上章节的知识点后，就能够熟练学习和开发此模块。

### 27.2 系统设计

聊天模块的作用在于能够实现在线同网站的其他人员聊天。聊天不仅可以与网站的其他用户进行聊天，也可以同网站的客服人员进行聊天。所以聊天模块不仅仅局限于单个用户和单个用户的聊天，可以是单个用户和单个用户，也可能是单个用户对多个用户。

#### 27.2.1 模块功能描述

聊天模块能够在一定程度上加强用户与用户之间的信息沟通，另外聊天模块也能够加强客户与用户之间的交互，对用户的信息进行及时的反馈。聊天模块的功能并没有复杂的模块和过多的页面，对于聊天模块而言，只需要进行页面中的数据处理即可，也无需保存数据。

当用户选择进行在线聊天时，用户需要进行登陆操作，这个登陆操作无需从数据库中进行提取和验证，这里的用户登陆操作只是给用户一个名称，以便于在页面聊天中方便被识别。当用户登陆完成后就能够进入主登陆窗口进行聊天操作。用户能够同多个人进行群聊也可以同单个用户进行私聊，当用户进行群聊时，群聊的信息会发布到相应的群聊文本框中；而当用户进行私聊时，其信息并不会发布到相应的群聊文本框，而发布到私聊文本框。

对于聊天的用户而言，可以选择是否将聊天记录进行保存，如果将聊天记录进行保存，系统可以将用户的相应的聊天记录保存为 **txt** 文档存放在用户的个人电脑中，以保存相应的数据而无需进行数据库的数据存储。同时，为了能够提高用户体验，对于聊天模块而言可以使用 **AJAX** 进行无刷新实现，聊天模块用户流程图如图 27-1 所示。

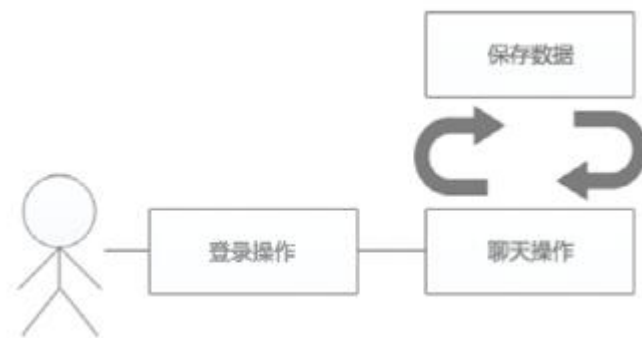


图 27-1 聊天模块功能描述

正如图 27-1 所示，用户在进行聊天操作之前首先需要进行登陆操作，在进行登陆操作后，系统会为用户分配一个 **Session** 对象用户描述用户的基本信息。当用户进行聊天时，其 **Session** 对象的相应值能够描述用户信息。

当用户执行完成聊天操作之后，可以选择是否保存数据并继续进行聊天。如果用户选择保存数据，用户的聊天信息全都保存为 **txt** 文档存放在本地，如果用户并不希望进行数据保存，可以继续聊天或直接关闭浏览器。从上述模块功能描述中可以规划成以下几个页面：

- ☐ 登陆操作页面：用于用户的登陆操作。
- ☐ 选择聊天室：用户选择喜爱的聊天室分类。
- ☐ 聊天操作页面：用于用户的聊天操作，可以进行单人聊天或多人聊天。

其中登陆操作主要是用于用户的登陆和 **Session** 对象的分配，而聊天操作页面就是一个在线聊天的主页面。聊天模块中并没有涉及到数据库的存储，而更多的是编程的技巧，聊天模块没有数据读取也就不存在数据库设计和性能了。

## 27.2.2 模块流程分析

聊天模块并不是网站应用中非常重要的模块，但是在很多业务情况下聊天模块也是非常必要的模块。特别是在用户咨询等情况下作为网站的客户服务就非常需要及时的信息获取和反馈，聊天模块能够非常好的完成这项任务。

在聊天模块的开发过程中，聊天模块并不涉及到数据的存储和读取，所以聊天模块属于即时模块，也就是说当用户关闭了浏览器之后除非用户保存聊天记录到文件中，否则用户的大部分信息全部都会丢失。当用户需要进行数据的存储或管理，可以选择保存聊天记录以保存操作的数据，操作数据将以 **txt** 文本文档的形式存储在用户的个人电脑中。从以上的流程分析中可以归纳出大部分用户在聊天模块中执行的操作的顺序，在流程分析中可以归纳如下：

- ☐ 用户登陆：用户登陆并进行信息初始化。
- ☐ 访问聊天页面：用户访问相应的聊天页面进行聊天操作。
- ☐ 页面初始化：聊天页面能够初始化用户信息，包括有多少个用户在此聊天室。
- ☐ 执行聊天操作：用户和一个用户或多个用户进行聊天操作。
- ☐ 存储聊天信息：用户可以选择存储聊天信息或直接离开。
- ☐ 离开聊天室：离开聊天室后需要刷新聊天室信息。

上述流程如图 27-2 所示。



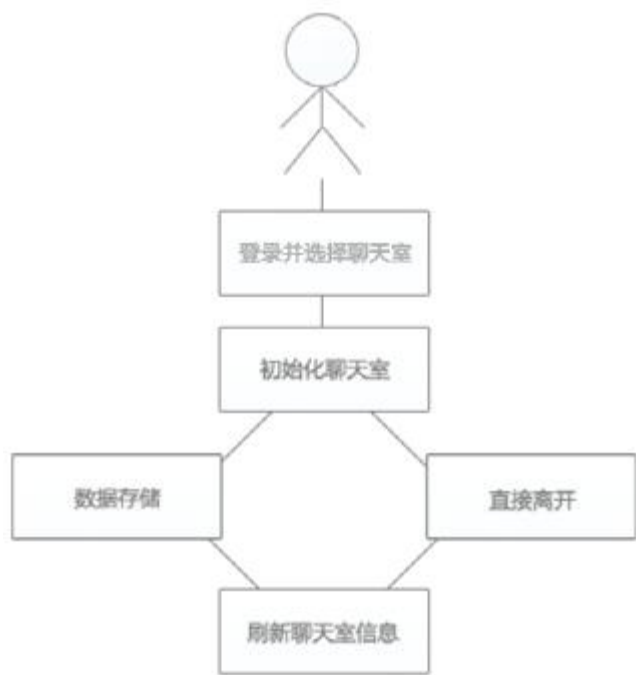


图 27-2 聊天操作流程图

在用户进入聊天室之前，首先需要初始化聊天室信息，例如已经在此页面的用户数量和用户的遍历。当用户离开聊天室时，需要刷新聊天室信息，因为一个用户离开了聊天室，那么聊天室信息中的用户数量和用户名称都已经被改变。无论聊天室中用户数量是增加还是减少，都需要进行数据的刷新。

## 27.3 界面设计

聊天室是用户与用户之间信息沟通的页面，聊天室的界面设计能够加强用户的粘度以使用户的再次回访。提高用户体验能够让用户在聊天应用中感觉到舒适，从而提高网站的口碑让更多的用户参与到在线聊天中。

### 27.3.1 登陆界面设计

用户能够在登陆界面进行登陆和聊天室的选择，在登陆操作和聊天室选择中可以使用 **ASP.NET 3.5 AJAX** 进行无刷新验证，登陆界面设计核心代码见光盘中源代码\第 27 章\27-1\27-1\login.aspx。

其中，代码使用了 **TextBox** 控件用于用户的昵称编写，用户必须填写昵称进行聊天，昵称是用户的一个标识，当用户进入聊天页面进行聊天时昵称就能够显示相应的用户信息，以便聊天中用户身份的区别。上述代码还使用 **RadioButtonList** 控件进行聊天室的选择，用户在填写昵称后需要选择相应的聊天室进行聊天。如果用户不选择聊天室同样不能够进行聊天，只有选择了相应的聊天室才能够聊天，例如用户可以选择“谈天说地”聊天室进行聊天，在该聊天室中的所有用户都是基于“谈天说地”这个话题进行聊天的。上述代码使用了 **AJAX** 进行无刷新效果实现，其布局如图 27-3 所示。

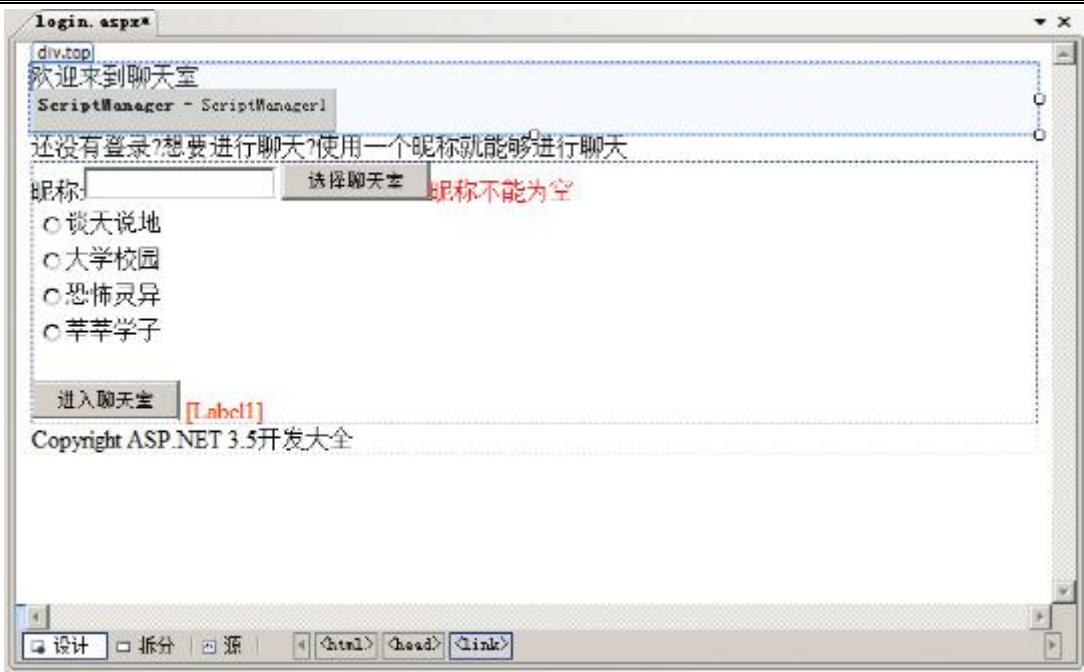


图 27-3 聊天登陆页面基本布局

在页面被初始化时，其中的 **RadioButtonList** 控件和进入聊天室按钮是不会显示的，当用户填写了昵称并单击选择聊天室才能够显示 **RadioButtonList** 控件和进入聊天室按钮。如图 27-3 所示，其登陆页面的友好度显然不够，可以使用 **CSS** 进行样式控制让该页面更加友好和丰富。

27.3.2 登陆界面 CSS

如上一节中的登陆界面可以看出登陆界面并不够友好，友好的登陆界面能够让用户喜欢这个网站并长期进行访问，这样就能够提高网站的访问量和提高用户的粘度。为了让页面的友好度更高可以使用 **CSS** 进行样式控制，**CSS** 代码如下所示。

```
body //控制全局样式
{
    font-size:12px;
    font-family:Geneva, Arial, Helvetica, sans-serif;
    margin:0px 0px 0px 0px;
    background:white url(images/background.gif);
}
.all //控制登陆框样式
{
    margin:50px auto;
    width:520px;
    background:white;
    border:1px solid #ccc;
}
.top //控制头部样式
{
    margin:0px auto;
    width:500px;
    padding:10px 10px 10px 10px;
    background:white url(images/bg.png) repeat-x;
}
.center //控制登陆样式
{
    margin:0px auto;
    width:500px;
    padding:10px 10px 10px 10px;
}
.end //控制底部样式
```

```
{
    margin:0px auto;
    width:500px;
    padding:10px 10px 10px 10px;
}
```

上述 CSS 分别为登陆页面进行了样式控制，其中定义了全局样式并定义了页面的字体大小，定义了全局样式后为其他的 DIV 层定义了样式，包括外对齐、宽度和内对齐等。在 CSS 文件中使用了图片让页面看上去更加友好，良好的背景图片和导航的图片的应用能够提升网站的设计效果，CSS 样式控制后的页面效果如图 27-4 所示。

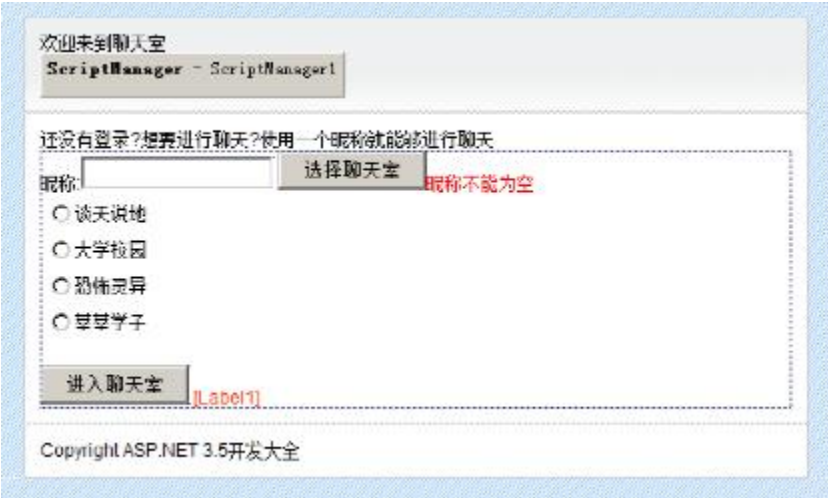


图 27-4 布局后的效果

通过 CSS 进行样式布局后的页面效果明显好很多，当用户访问该页面时会感觉到页面设计的友好度，提高了用户体验，增强了网站对用户的粘度和可信度。

27.3.3 聊天室显示界面

聊天室主窗口包含一些常用的窗口，这些窗口用于呈现相应的文本，包括用户的聊天发布窗口和用户的对话窗口。在聊天室面板中还包含用户列表，这些列表能够为用户提供私聊服务，聊天室显示界面核心代码见光盘中源代码\第 27 章\27-1\27-1\room.aspx。

在页面中的上部分代码实现的是群聊窗口，当用户不指定私聊对象时，其发布的聊天信息将会呈现在群聊窗口。如果用户希望与某个用户进行私聊，就需要使用私聊窗口，示例代码见光盘中源代码\第 27 章\27-1\27-1\room.aspx 中私聊窗口所示。

聊天窗口中为了防止页面的重复刷新，可以使用 AJAX 控件实现无刷新功能。为了能够让页面在指定的时间内进行局部刷新，就需要使用 Timer 控件进行实现，示例代码如下所示。

```
<asp:Timer ID="Timer1" runat="server" Interval="10000" ontick="Timer1_Tick">
</asp:Timer>
```

上述代码实现了聊天页面中的主窗口，其中主窗口包括群聊窗口、私聊窗口、发言窗口和发言人窗口，当用户单击其中的按钮控件进行聊天时，会根据其中的发言窗口和发言人窗口在群聊窗口和私聊窗中显示相应的数据，如图 27-5 所示。

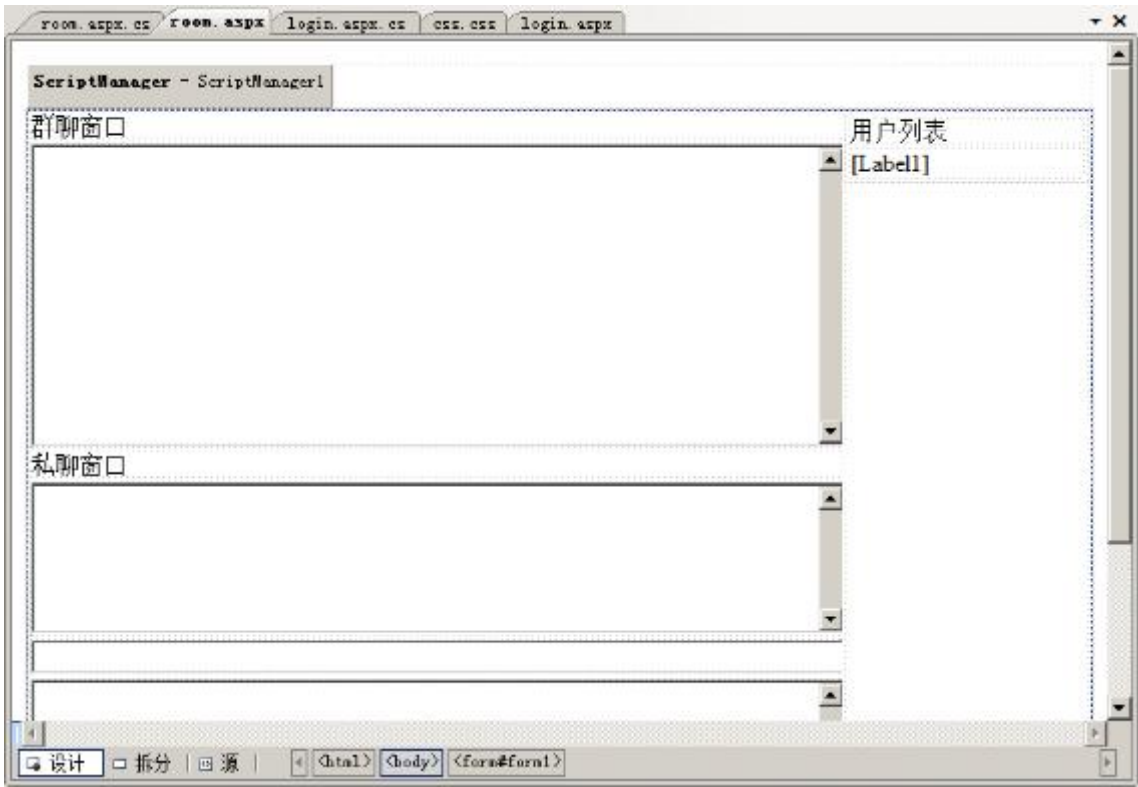


图 27-5 聊天页面窗口

在聊天页面的右侧有一个用户列表，当加载该页面时会初始化页面信息并载入用户列表，当用多个用户时用户列表就会呈现为多个用户，用户可以单击用户列表与相应的用户进行聊天。

27.3.4 聊天室界面 CSS

同样该聊天室窗口不太人性化也没有任何的用户体验，使用 CSS 能够提高用户体验，不同的页面的 CSS 都可以放置在同一个 CSS 文件中，这些 CSS 文件能够被单个或多个页面使用，减少了冗余代码，CSS 代码如下所示。

```
.room
{
    margin:10px auto;
    width:800px;
    background:white;
    border:1px solid #ccc;
}
.banner
{
    width: 536px;
    background:white url(images/bg.png) repeat-x;
}
```

由于两个页面使用的是同一个 CSS 文件所以很多样式控制可以无需再次编写，只需要对该页面中需要使用的样式进行样式编写，聊天室界面需要控制其主聊天窗口的长度和宽度，为了提高用户友好度，也可以使用图片进行样式控制，如图 27-6 所示。



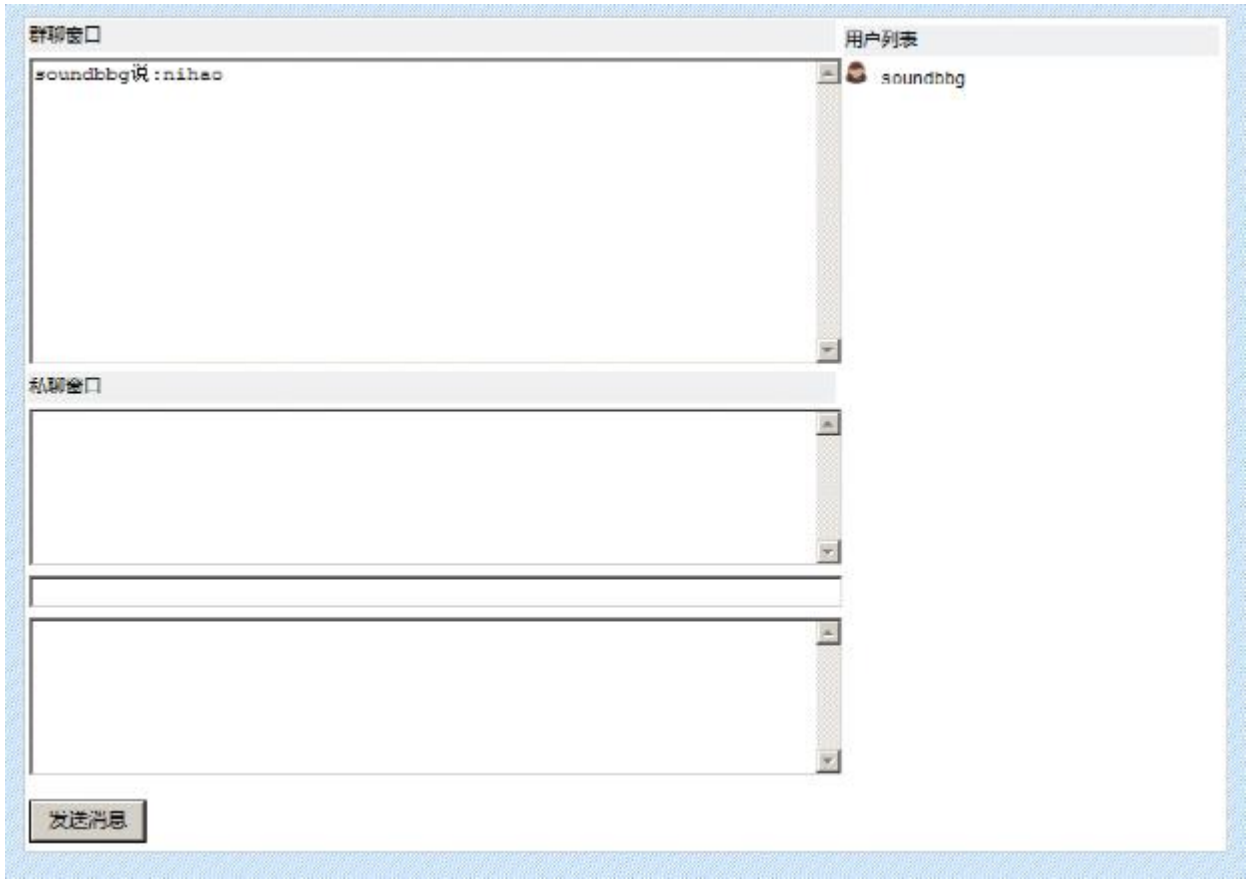


图 27-6 布局后的聊天窗口

正如图 27-6 所示，在使用 CSS 进行样式控制后的页面具有更好的友好度，在用户列表区域，当有多个用户时会呈现多个不同的用户列表。在用户列表中用户可以选择相应的用户并与用户发送私密消息，当用户发送私密消息时，其消息不会呈现在群聊窗口，而是呈现在私聊窗口。

## 27.4 代码实现

聊天模块不需要进行数据存储和读取，在聊天模块中，当用户发送一个信息到页面中需要进行聊天时，在相应的窗口就应该显示用户发布的信息。对于不同的用户而言，不同的用户所看到的页面是不同的，这里必须使用 **Application** 对象进行跨页存储。

### 27.4.1 登陆代码实现

当用户进行页面访问时，页面中的一些控件初始化是无法看见的，只有当用户填写了用户名昵称并选择了相应的聊天室才能够进行登陆操作，在用户选择聊天室之前，必须填写用户名。用户填写用户名之后就能够在单击【选择聊天室】按钮进入聊天室。示例代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    RadioButtonList1.Visible = true;           //显示控件
    Button2.Visible = true;                     //显示控件
    TextBox1.ReadOnly = true;                   //锁定用户名
}
```

当用户填写了用户名之后并单击选择聊天室按钮进入聊天室选择，在选择聊天室时就不能够再修改用户名，因为一旦用户确定了用户名并进行聊天室选择时就无法再修改自己的用户名。在这里其实也可以让用户能够修改用户名，只是这样做会造成网站应用的不安全。当用户进行聊天室选择后就可以进入聊天室，进入聊天室按钮代码实现如下所示。

```
protected void Button2_Click(object sender, EventArgs e)
{
    if (RadioButtonList1.SelectedIndex == -1) //判断单选列表
```

```
{
    Label1.Text = "请选择一个聊天室";           //提示选择
}
else
{
    Session["roomid"] = RadioButtonList1.SelectedItem.Value;           //赋予 Session 值
    Session["username"] = TextBox1.Text;
    Response.Redirect("room.aspx?id="+ RadioButtonList1.SelectedItem.Value + ""); //跳转
}
}
```

上述代码当用户单击选择聊天室按钮后就能够进行聊天室的选择，用户必须选择一个自己感兴趣的聊天室进行聊天，否则系统会提示“请选择一个聊天室”。如果用户选择了聊天室并进行登陆，系统会为用户赋予两个 **Session** 值，这两个 **Session** 值分别为 **roomid** 和 **username**，其中 **roomid** 为聊天室的 ID 号，而 **username** 用于存储进行聊天的用户名。当聊天页面被载入时，用户的 **Session** 对象的 **roomid** 值会与传递的参数进行判断，如果是相应的聊天室就允许用户进行聊天，如果不是相应的聊天室则不允许用户聊天。

### 27.4.2 多人聊天代码实现

多人聊天相对比较简单，用户可以在页面中直接进行信息输入发布聊天信息。多人聊天时，用户发布的信息能够被所有人看见，这也就是说用户的信息能够呈现在多人聊天窗口。当多人聊天时，不同的用户所打开的页面是不相同的，这样就造成可能信息呈现的时间不一致，为了保证信息的一致性，这里使用了 **AJAX** 的 **Time** 控件进行刷新。

在聊天页面加载时，首先需要判断用户是否包含 **Session** 值或者聊天室是否为用户选择的聊天室，否则会跳回登陆页面，不仅如此，当页面被初次加载时还需要进行一些初始化工作，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    if (Session["roomid"] == null || Session["username"] == null)           //判断是否登陆
    {
        Response.Redirect("login.aspx");           //页面跳转
    }
    if (Request.QueryString["id"] != Session["roomid"].ToString())           //判断是否匹配
    {
        Response.Redirect("login.aspx");
    }
    Label1.Text = "";           //清空控件的值
    for(int i=0;i<Session.Count/2;i++)           //添加用户列表
    {
        if (Session[i * 2] == Session["roomid"])           //配置 Session
        {
            Label1.Text += "<img src=\"images/p.png\"> &nbsp; " + (Session[i * 2 + 1] + "<br/>");
        }
    }
    if (Application["char"] != null)           //初始化聊天信息
    {
        TextBox3.Text = Application["char"].ToString();           //获取 Application
    }
}
```

上述代码在页面加载时被执行，当用户访问页面时，首先会对用户的身份进行判断，判断用户是否已经登陆，如果用户没有登陆则跳转到登陆页面进行登陆。如果用户已经登陆，还需要判断用户登陆是否所属对应的房间，如果不是对应的房间则会被认为是非法进入房间，需要重新进行登陆操作。当用户身份验证通过后，就需要进行初始化操作清除相应的控件的值并循环添加用户列表项，在添加用户列表时，需要遍历 **Session** 对象的值进行列表的初始化，示例代码如下所示。

```

for(int i=0;i<Session.Count/2;i++)
{
    if (Session[i * 2] == Session["roomid"])           //判断 Session
    {
        Label1.Text += "<img src=\"images/p.png\"> &nbsp; " + (Session[i * 2 + 1] + "<br/>");
    }
}

```

在用户登陆后，会给用户分配两个 **Session** 对象，一个用户记录聊天室的 **ID**，另一个用于记录用户名。这也就是说当遍历 **Session** 时，会有多个 **Session** 对象，即一个用户有 2 个 **Session** 对象，当遍历用户列表时需要筛选这些 **Session** 值，去掉聊天室 **ID** 的 **Session** 的值而呈现用户的 **Session** 值。在用户登陆完成后，在用户之前已经有很多人进行聊天了，在用户进入聊天室后，需要加载这些聊天信息，示例代码如下所示。

```

if (Application["char"] != null)
{
    TextBox3.Text = Application["char"].ToString();           //加载聊天信息
}

```

上述代码使用了 **Application** 对象进行跨页的数值存储，**Application** 对象是页面中的公共对象，就算是不同的用户之间也能够共享 **Application** 对象，在页面进行聊天信息发布时，可以将值添加到 **Application** 对象中被其他用户读取。当用户单击按钮控件进行消息发布时，需要在页面中的相应位置进行呈现，在多人聊天代码实现中，可以直接将信息内容增加到文本框中，示例代码如下所示。

```

TextBox3.Text += Session["username"] + "说:" + TextBox2.Text + "\n";
TextBox2.Text = "";                                           //清空窗口
Application["char"] = TextBox3.Text;                         //添加公共对象

```

当用户进行消息发布时，其中多人聊天窗口的信息要增加刚才用户发布的信息，如上述代码所示，其中多人聊天窗口 **TextBox3** 的信息增加了“**XX** 说：...”的字符串。由于页面使用了 **AJAX** 控件进行无刷新的实现，用户基本看不出来页面被刷新。当用户单击按钮控件时就能够进行局部刷新，实现多人聊天。

单个用户进行聊天信息的发布并不能被其他用户阅读，当两个人打开一个页面，其中一个人进行的操作不会呈现给另一个人。在这里不仅要使用 **Application** 对象还需要进行页面刷新，这里使用了 **AJAX** 控件中的 **Timer** 控件实现，当 **Timer** 控件执行更新时，页面中的对话框的数据也会进行更新，示例代码如下所示。

```

protected void Timer1_Tick(object sender, EventArgs e)
{
    TextBox3.Text = Application["char"].ToString();           //更新数据
}

```

当 **AJAX** 中的 **Timer** 控件执行刷新操作后，其多人聊天窗口的文本值就会等于 **Application** 对象的相应值，这样在其他页面中就能够查看现有的数据。

### 27.4.3 单人聊天代码实现

当用户需要进行单人聊天时，其聊天的内容不能够被多人聊天窗口捕获和呈现，单人聊天的信息必须呈现在私人聊天窗口中。在聊天页面中，与多人聊天不同的是，多人聊天时无论信息是怎样发布的其信息都会呈现在多人聊天窗口中，这样也没有谁发送给谁之说。而在单人聊天时，当用户 **soundbbg** 发送信息给 **guojing** 时，就需要判断是否有这个用户并且提示用户接收信息。

多人聊天时无需考虑到用户是否需要接受，这就和广播一样，广播台无需关心用户是否在接听广播，广播台只需要负责发送，用户可以选择接收或不接收。而单人聊天时需要判断是否有这个用户，如果有这个用户，不仅在发送者的聊天窗口中需要添加字符串“你给 **XX** 发送了 **XX** 信息”，同样在接收者中需要添加“**XX** 给你发送了 **XX** 信息”。在显示信息时，需要遍历 **Application** 对象进行判断，实现代码如下所示。

```

TextBox3.Text += "你对" + TextBox4.Text + "说:" + TextBox2.Text + "\n"; //输出聊天记录
TextBox1.Text += "你对" + TextBox4.Text + "说:" + TextBox2.Text + "\n"; //输出聊天记录
Application[Session["username"].ToString()] = TextBox1.Text;           //增加 Application 对象
for (int i = 0; i < Application.Count; i++)                             //遍历 Application 对象
{
    if (Application[TextBox4.Text] != null)                             //判断 Application

```

```
{
    Application[TextBox4.Text] += TextBox4.Text + "对你" + TextBox4.Text + "说:"
    + TextBox2.Text + "\n";
    //增加相应聊天记录
}
}
TextBox2.Text = "";
//清空窗口
```

上述代码在用户发送相应的信息后，系统会遍历 **Application** 对象查找是否有这个用户，如果没有这个用户则不在相应的用户信息框中呈现信息，如果存在这个用户，则在该用户的 **Application** 对象中添加字符串“**XX** 对你说 **XX** 信息”。

但是上述代码有一定的缺陷，就是当用户和用户之间发送私密信息时，同样还是会将信息发送到群聊文本框中，这就需要进行判断。如果没有填写相应的用户名，则说明这个信息是一个群发信息，进行广播，在群聊文本框中就会显示该信息，如果填写了用户名，则说明这个信息是一个私聊信息，就不会进行广播，示例代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    if (String.IsNullOrEmpty(TextBox4.Text))
    {
        //判断是否是群发
        TextBox3.Text += Session["username"] + "说:" + TextBox2.Text + "\n"; //群发窗口添加记录
        TextBox2.Text = "";
        //清空窗口
        Application["char"] = TextBox3.Text;
        //更新 Application 对象
    }
    else
    {
        //TextBox3.Text += "你对" + TextBox4.Text + "说:" + TextBox2.Text + "\n";
        TextBox1.Text += "你对" + TextBox4.Text + "说:" + TextBox2.Text + "\n";
        Application[Session["username"].ToString()] = TextBox1.Text;
        //获取 Application 对象
        for (int i = 0; i < Application.Count; i++)
        {
            //遍历 Application 对象
            if (Application[TextBox4.Text] != null)
            {
                //查找 Application 对象
                Application[TextBox4.Text] += TextBox4.Text + "对你" +
                    TextBox4.Text + "说:" + TextBox2.Text + "\n";
                //修改相应用户记录
            }
        }
        TextBox2.Text = "";
        //清空窗口
    }
}
```

上述代码在发送信息前，首先会判断是否是群发，如果是群发则不进行其他的任何操作，直接进行文本框中的数据的呈现，并将相应的值添加到公共对象 **Application["char"]** 中去。如果用户进行的是私聊，那么就需要遍历 **Application** 对象进行信息的发布，发布信息的同时不仅要修改发布者的 **Application** 对象，同样需要修改接收者的 **Application** 对象，当 **AJAX** 的 **Timer** 控件进行刷新时，用户就能够看到发送者的信息。

单人聊天实现的过程比多人聊天的过程更加复杂，在多人聊天时只需要将数据添加到公用对象中，而无需考虑发送者和接受者，当进行单人聊天时，不仅需要修改发送者的 **Application** 对象，还需要遍历 **Application** 对象进行接受者的 **Application** 对象的修改。

#### 27.4.4 聊天记录保存实现

当用户希望保存聊天记录时，可以单击相应的文本框旁边的保存记录进行聊天记录保存。聊天记录可以保存为 **txt** 文本文档到用户的目录中，当用户希望查阅聊天记录时，可以打开相应的文件进行查阅，示例代码如下所示。

```
protected void LinkButton1_Click(object sender, EventArgs e)
{
```



```
if (!Directory.Exists("C:\\chat\\group")) //保存群聊记录
{
    Directory.CreateDirectory("C:\\chat\\group"); //创建路径
    //开始通过编写文本保存路径创建文件
    StreamWriter sw = File.CreateText("C:\\chat\\group\\" + DateTime.Now.Year +
        DateTime.Now.Month + DateTime.Now.Day +
        DateTime.Now.Hour + DateTime.Now.Second + ".txt");

    sw.Write(TextBox3.Text); //编写文本
    sw.Close(); //关闭对象
}
else
{
    StreamWriter sw = File.CreateText("C:\\chat\\group\\" + DateTime.Now.Year +
        DateTime.Now.Month + DateTime.Now.Day +
        DateTime.Now.Hour + DateTime.Now.Second + ".txt");

    sw.Write(TextBox3.Text); //编写文本
    sw.Close(); //关闭对象
}
LinkButton1.Text = "已经保存"; //提示保存信息
}
```

上述代码首先在用户的计算机的 C 盘中进行文件夹判断，如果存在这个目录，就直接进行 **txt** 文件的创建，如果不存在则首先创建相应的目录然后再进行文件的存储。用户私聊记录同样可以保存在用户计算机中，其代码实现基本相同，示例代码如下所示。

```
protected void LinkButton2_Click(object sender, EventArgs e)
{
    if (!Directory.Exists("C:\\chat\\pri")) //保存私聊记录
    {
        Directory.CreateDirectory("C:\\chat\\pri"); //创建文件
        StreamWriter sw = File.CreateText("C:\\chat\\pri\\" + DateTime.Now.Year +
            DateTime.Now.Month + DateTime.Now.Day +
            DateTime.Now.Hour + DateTime.Now.Second + ".txt");

        sw.Write(TextBox2.Text); //编写内容
        sw.Close(); //关闭对象
    }
    else
    {
        StreamWriter sw = File.CreateText("C:\\chat\\pri\\" + DateTime.Now.Year +
            DateTime.Now.Month + DateTime.Now.Day +
            DateTime.Now.Hour + DateTime.Now.Second + ".txt");

        sw.Write(TextBox2.Text); //编写内容
        sw.Close(); //关闭对象
    }
    LinkButton1.Text = "已经保存";
}
```

上述代码将用户的私聊记录进行保存，同样会判断目录的存在性，如果不存在目录则会创建相应的目录进行文件保存。

## 27.5 实例演示

聊天模块能够在网站开发过程中进行用户和用户的信息的交互，也能够进行用户和客户服务之间的交互，也能够实时的进行问题的反应和反馈，在用户进行聊天之前需要选择用户名并选择相应的聊天，如图 27-7 和图 27-8 所示。



图 27-7 填写用户名

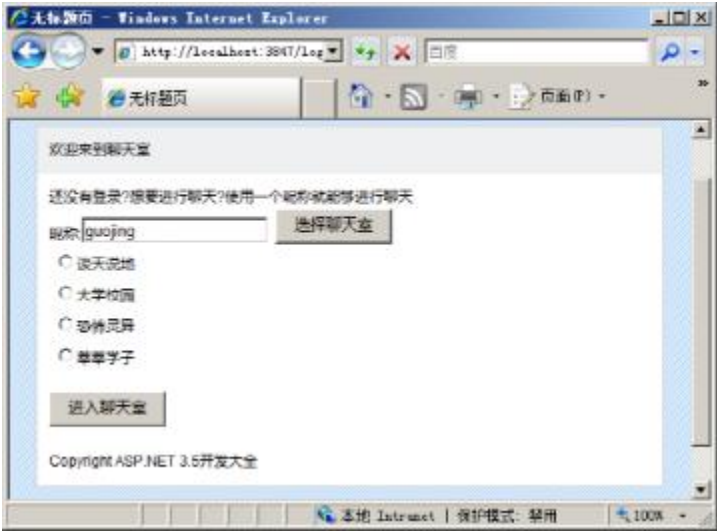


图 27-8 选择聊天室

用户在选择聊天室之前必须输入用户名，如果不输入用户名就不能够进行聊天室选择和登陆，当用户选择了相应的用户名之后就能够选择聊天室进行聊天，当单击【进入聊天室】按钮后就能够跳转到相应的聊天室页面，如图 27-9 所示。

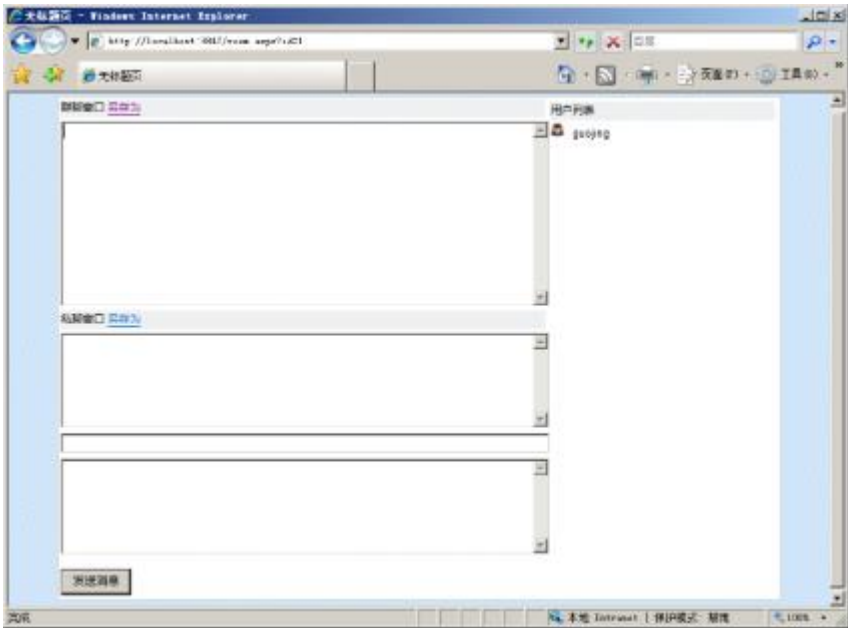


图 27-9 聊天室页面

在聊天室页面能够进行聊天，用户可以在最下面的对话框中输入信息，输入的信息将作为群发信息发送到群消息中，如果用户在用户名窗口中输入用户名则消息会发送到私聊窗口而不会发送到群发窗口，如图 27-10 和图 27-11 所示。



图 27-10 发送群发消息



图 27-11 发送私聊信息

用户可以选择相应的用户名进行聊天，用户可以在聊天窗口中进行聊天信息的发送和接收，当用户聊天之后希望将聊天记录保存时，可以在相应的位置选择“另存为”就能够将记录保存在相应的目录下。群体记录保存在 C:\chat\group 目录下而私聊记录保存在 C:\chat\pri 目录下。这些记录都会按照时间进行保存，

如图 27-12 所示。



图 27-12 生成目录

正如图 27-12 所示，用户能够将聊天记录保存在计算机的目录中也可以随时将记录删除，当用户需要使用到聊天记录时可以调出系统中的相应日期的 **txt** 文件就能够进行聊天记录的查看。

## 27.6 小结

本章通过聊天模块的开发对 **ASP.NET** 内置对象进行了详细的讲解。在聊天模块中，虽然没有对数据库进行存储和读取，但是需要更多的编程技巧进行 **Web** 应用中用户状态的保存和编程。聊天模块是网站开发中一个基本模块，但是聊天模块并不是常用的模块，虽然聊天模块并不是常用模块，但是通过聊天模块可以更加深入的了解 **ASP.NET** 内置对象。本章还巩固了：

- ❑ **Web** 窗体基本控件。
- ❑ **Web** 窗体数据控件。
- ❑ **ASP.NET** 内置对象。
- ❑ 生成静态的概念
- ❑ 自定义控件和用户控件。
- ❑ **ASP.NET 3.5** 与 **AJAX**

**ASP.NET** 内置对象在 **ASP.NET** 应用开发过程中是非常重要的，也是维持 **Web** 状态的一个重要的方法，虽然聊天模块在现在的网站开发中并不常用，但是也是学习 **ASP.NET** 内置对象的最佳方法。在进行聊天模块的开发中，本章讲解的开发方法并不是最好的方法，如果要能够异步进行跨页信息发送和接受，可以开发服务器端和客户端分别尽心信息发送和接收，由于篇幅限制和难度限制，本书不对服务器端/客户端的聊天开发方法进行详细的介绍。

## 第八篇 ASP.NET 3.5 应用实例

第 28 章 制作一个 ASP.NET 留言本

第 29 章 制作一个 ASP.NET 校友录系统



## 第 28 章 制作一个 ASP.NET 留言本

在了解了一些基本的模块的开发之后就能够开发一些基本的应用，这些应用可以看作是很多的模块组成应用，在开发过程中可以应用现有的模块进行应用的开发。留言本是最基础 **Web** 应用，也是初学者最常学习的 **Web** 应用。

### 28.1 系统设计

系统设计在项目开发中是非常重要的，在系统设计中，需求分析也是最为重要的。需求分析规定了开发小组或团队以何种方式进行模块的开发和编码，也规定了客户最基本的需求，如果连客户最基本的需求都没有弄清楚，那么这个系统必然是失败的。

#### 28.1.1 需求分析

需求分析是系统设计中最为核心的组成，在任何系统的开发中都需要进行需求分析，虽然 **ASP.NET** 留言本是一个很小的项目，但是还是需要进行需求分析。需求分析并不因为项目的大小而有任何区别，需求分析更多的任务是告诉开发团队客户想要的是什么、客户需要的是什么、团队怎样进行模块划分和开发等等。

虽然在 **ASP.NET** 留言本开发中需求分析显得微不足道，但是随时保持编写需求分析是一个非常良好的习惯。需求分析是软件工程中的一个概念，指的是在建立一个新的或改变一个现存的电脑系统时描写新系统的目的、范围和定义时所要做的所有的工作。简单的说需求分析也就是分析客户要的是什么、怎么做、做完了怎么办。对于 **ASP.NET** 留言本项目而言，其需求分析可以编写如下：

##### 1. 目录

需求分析通常情况下是一个单独的需求分析文档，需求分析文档的格式很像一本书或论文的格式，其示例目录如下所示。

- 1. 引言：通常是需求分析文档的引言，用户描述为何编写需求分析文档。
- 1.1 编写目的：编写目的用户描述为何编写需求分析文档。
- 1.2 项目背景：编写相应的项目背景。
- 1.3 定义缩写词和符号：编写在需求分析文档中定义的缩写词或符号等。
- 1.4 参考资料：用户描述在需求分析文档中所参考的资料。
- 2. 任务描述：定义任务，通常情况下用于描述完成何种任务。
- 2.1 开发目标：定义开发目标，包括为何要进行开发。
- 2.2 应用目标：定义应用目标，包括系统应用人员要实现什么功能，以及有哪些应用等。
- 2.3 软件环境：用于定义软件运行的环境。
- 3. 数据描述：用户进行数据库中数据设计开发的描述。

这里只是简单的介绍了需求分析文档编写中的目录的一些基本格式，需求分析文档通常是一个单独的文档，而需求分析文档需要解决客户需要的是什么，如何进行协调开发等。对于不同的项目其需求分析文档其目录并不限于此格式，对于小型的项目的需求分析可以灵活更改。

##### 2. 引言

在需求分析文档中，通常需要编写引言用户描述为何编写需求分析文档和需求分析文档的作用，对于 **ASP.NET** 留言本而言，其引言可以编写如下：

在对客户现有的应用模块的调查和了解的基础上，用户希望能够在现有的应用中加上留言本的功能以便能够及时的和用户进行信息反馈和调查。

此规格说明书在详细的调查了客户现有的应用模块和基本的操作流程后进行编写，对留言本功能进行了详细的规划、设计，明确了软件开发中应具有的功能、性能使得系统的开发人员和维护人员能够详细清楚的了解软件是如何开发和进行维护的，并在此基础上进一步提出概要设计说明书和完成后续设计与开发工作。本规格说明书的预期读者包括客户、业务或需求分析人员、测试人员、用户文档编写者、项目管理人员等。

### 3. 项目背景

项目背景用于描述该项目在何种背景或何种条件下进行开发的，以及为何要进行现有的项目的开发或升级，**ASP.NET** 留言本的项目背景可以编写如下：

由于现在信息化的迅猛发展，原有的软件项目已经不能满足现今越来越多的需求，更多的厂商都将软件应用基于互联网进行开发和使用。相对于原有的 **C/S** 软件开发而言，基于互联网的软件开发具有部署快、成本低、维护性低的特性，对于企业而言可以使用基于互联网的应用进行信息的发布和反馈。

对于原有的系统而言，用户必须下载客户端才能够与企业内部数据进行通信，这样难免会造成使用不便和安全性的问题，因为用户需要进行软件下载。如果用户并没有连接到网络，就不能够及时的了解用户的信息也无法下载现有的程序，如果用户将现有的程序进行反编译等操作也会造成安全性的问题。

随着互联网的发展，越来越多的用户已经可以使用互联网进行信息交互，也促成了越来越多的基于浏览器的应用程序，企业可以使用服务器/客户端的开发模型进行系统的开发，**ASP.NET** 留言本就是为了解决信息交互复杂和交互困难的问题的而诞生的。为了解决现有的企业中企业与用户信息反馈困难等情况，让企业能够更加方便的同用户进行信息交互，在征求了多方意见的情况下进行此 **ASP.NET** 留言本的开发，以便解决现有的企业难题。

### 4. 任务描述

任务描述用于描述客户的任务，以及基本的如何完成任务的描述，**ASP.NET** 留言本的任务描述可以编写为如下所示：

为了加强现有的企业和用户之间的信息交互，也解决企业和用户的沟通不便的情况，现开发基于 **.NET** 平台的留言本应用程序，用户能够使用留言本进行信息的反馈和调查，能够及时的获取用户的相关意见或信息的数据。

**注意：**任务描述作为章节的概述，在软件规格说明书中，该章节的其他章节将需要对概述进行更详细的说明。

### 5. 开发目标

**ASP.NET** 留言本的开发目标是为了加强现有的企业和用户之间的信息交互，解决企业和用户的沟通不便的情况，进行企业和用户之间的数据整合和交互。

### 6. 应用目标

**ASP.NET** 留言本的应用目标是为了能够让企业能够获取用户的信息，这些信息包括用户的意见、反馈的信息以及用户数据等，同时企业也能够通过留言本进行基础的意见调查。

需求分析是系统设计中最为重要的一部分，如果在系统设计初期需求分析设计的非常好也就方便了后期的开发和维护。

## 28.1.2 系统功能设计

**ASP.NET** 留言本是企业内部的一个信息交互平台，用户可以在相应的主题的留言本之内进行信息发布

和反馈，用户还能够通过留言本进行信息的交互。在留言本的开发过程中需要确定基本的系统功能，这些基本的系统功能包括如下：

1. 留言信息浏览

用户可以在相应的留言页面进行留言信息的浏览，包括对企业产品的意见以及功能反馈等，在留言页面中按照用户的习惯可以进行按回复查看，按时间查看等选项。用户还能够通过导航栏进行不同留言板的跳转。管理员可以在留言信息浏览页面进行信息回复，可以对用户的疑问和意见进行反馈，管理员还能够删除不良的留言和屏蔽相关用户等操作。

2. 注册登录功能

在用户进行留言之前，必须进行注册和登录等操作，如果用户没有登录就不能够进行留言操作，用户登录或注册后可以通过留言索引自己的留言并进行留言修改或增加。

3. 用户留言索引

登录的用户可以索引自己的留言，对于自己较早的留言能够进行查看，这样就方便了用户进行信息整合，管理员也能够通过用户的索引相应的用户信息并进行用户管理。

4. 管理员留言管理

管理员对于不良的留言进行删除、屏蔽等操作，当用户进行了不良信息发布后管理员能够在留言页面进行删除操作。

28.1.3 模块功能划分

当介绍了系统所需实现的功能模块后并执行了相应的功能模块的划分和功能设计，可以编写相应的模块操作流程和绘制模块图，ASP.NET 留言本总体模块划分如图 28-1 所示。

图 28-1 描述了系统的总体模块功能划分，其中包括前面章节中讲到的留言信息浏览、用户信息注册、用户登录操作、用户留言索引以及管理员留言管理等操作，其中可以将用户注册、登录、信息浏览和索引等操作进行划分，如图 28-2 所示。

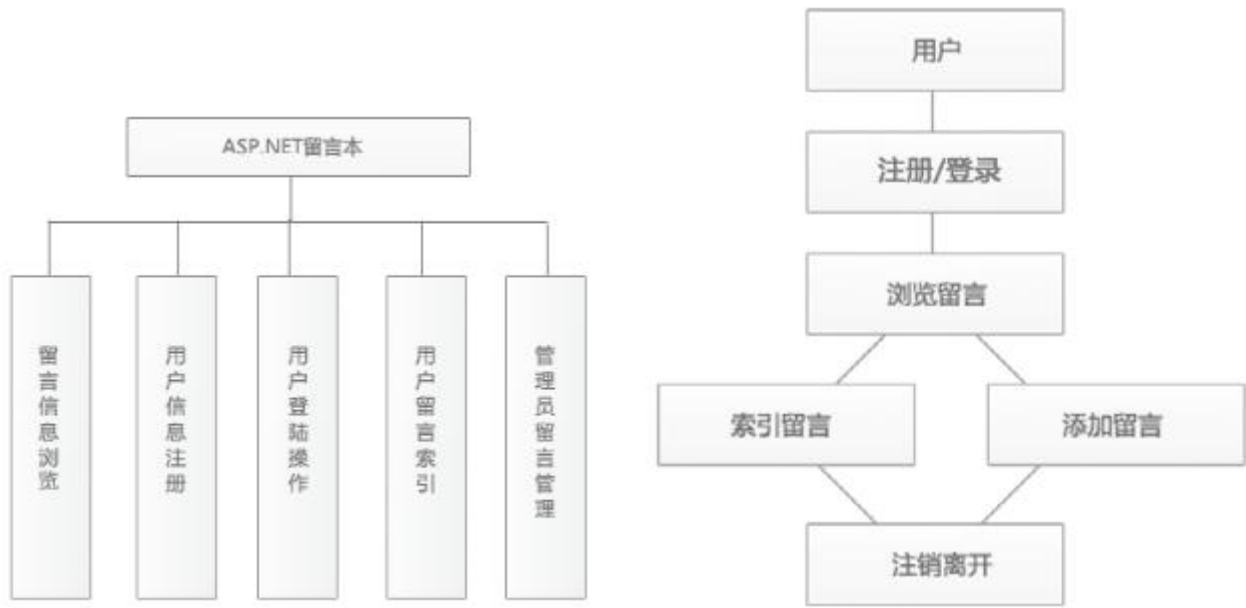


图 28-1 系统总体模块功能划分



图 28-2 用户操作模块流程图

用户在进行页面访问时，可以呈现相关的留言信息，当用户进行留言时就必须登录，如果用户事先没有任何账号信息可以进行注册，注册完成后会跳转到登录页面进行登录操作，如果用户已经存在账号就能够直接登录进行操作。

在用户注册或登录后就可以进行留言的索引和留言的添加，留言的索引能够方便用户查询长时间之前的自己的留言信息，例如用户进行留言后一个月再次访问企业网站，就会很难搜索到自己的留言，而通过索引能够方便的索引到自己的留言信息。如果用户没有任何留言可以选择添加留言，添加后的留言能够提交给管理员进行审核并回复。

对于管理员而言，管理员需要查看留言并进行留言的管理，在管理员管理之前也需要进行登录操作，以验证管理员身份的正确性和权限。当管理员验证通过后可以进行管理操作，管理员操作流程图如 28-3 所示。



图 28-3 管理员操作模块流程图

在管理员进行管理之前同样需要进行身份验证，否则会造成系统的安全性问题，管理员只有在身份验证之后才能够进行管理回复和留言的删除操作。

## 28.2 数据库设计

在 ASP.NET 留言本的功能模块描述中，可以看出在数据库的设计中包括多个表，这些表包括留言表、留言分类表、用户信息表、管理员信息表等表，这些表用于实现前面小节中系统设计所规划的系统功能。

### 28.2.1 数据库的分析和设计

在前面的系统设计中已经详细的了解了系统的功能，在数据库的设计中，需要充分的了解系统的功能并进行合理的抽象再进行数据库设计，数据库设计图如图 28-4 所示。

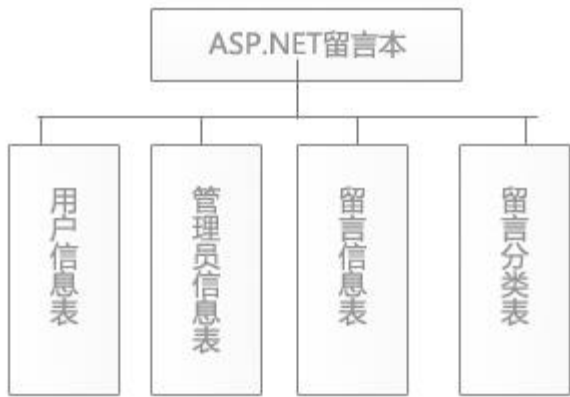


图 28-4 数据库设计图

其中初步的为数据库中的表进行设计，这里包括四个表，作用分别如下：

- ❑ 用户信息表：用于存放用户的信息并进行用户信息的管理。
- ❑ 管理员信息表：用于存放管理员的信息并在管理员登录时进行数据验证。
- ❑ 留言信息表：用于存放留言信息。
- ❑ 留言分类表：用于进行留言分类。



其中留言信息表和留言的分类表用于描述留言项目，一个企业网站不只包含一个留言页面，当企业有多个产品时，需要考虑到留言模块的扩展性，使用留言分类可以进行相应的功能的扩展。在 **ASP.NET** 留言本中最为重要的就是留言信息表和留言分类表，其中留言信息表的字段可以归纳如下。

- ☐ 留言编号：用于标识留言本的编号进行索引，为自动增长的主键。
- ☐ 留言标题：用户留言的标题。
- ☐ 留言名称：用户的名称。
- ☐ 留言时间：用户留言的时间。
- ☐ 留言内容：用户留言的内容。
- ☐ 回复标题：管理员回复留言的标题。
- ☐ 管理员名称：管理员的名称。
- ☐ 回复时间：管理员回复留言的时间。
- ☐ 回复内容：管理员回复的内容。
- ☐ 所属留言分类：用户留言所属的分类。
- ☐ 所属用户：留言所属的用户 **ID**。

上述表用于描述用户留言信息，用户留言表中的信息为最主要的数据，在进行留言呈现时呈现的就是此数据表中的数据。同时为了能够将留言数据进行分类，需要创建留言分类表，其字段可以描述如下所示。

- ☐ 分类编号：用于标识留言本分类的编号，为自动增长的主键。
- ☐ 分类名称：用于描述分类的名称，例如“客户服务”等。

留言分类和留言表一起描述留言项目，这样能够增加留言本系统的扩展性，管理员可以创建多个留言分类进行留言管理。例如现在企业有一个“皮鞋”产品，可以创建一个主题为“皮鞋”的留言本，但是如果企业有一天多了一个产品，那么就需要重新制作留言本，重新制作的留言本在数据和管理上都需要重新操作。而如果将一个留言本进行分类处理，当有新产品出现时就能够很好的扩展。

在进行留言前，用户必须要登录，如果没有登录就必须要注册，这里可以使用注册模块进行注册功能的实现，注册模块中用户表的字段可以归纳如下。

- ☐ 用户名：用于保存用户的用户名，当用户登录时可以通过用户名验证。
- ☐ 密码：用于保存用户的密码，当用户使用登录时可以通过密码验证。
- ☐ 性别：用于保存用户的性别。
- ☐ 头像：用于保存用户的个性头像。
- ☐ **QQ/MSN**：用于保存用户的 **QQ/MSN** 等信息。
- ☐ 个性签名：用于展现用户的个性签名等资料。
- ☐ 备注：用于保存用户的备注信息。
- ☐ 用户情况：用于保存用户的状态，可以设置为通过审批和未通过等。

用户在注册后就能够进行登录，登录后的用户将能够通过留言页面进行信息发布和反馈。在用户发布信息后，管理员可以对这些信息进行管理，同样管理员在进行管理时也需要进行登录操作，其字段可以归纳如下。

- ☐ 管理员编号：用于标识管理员信息，为自动增长的主键。
- ☐ 管理员用户名：用于标识管理员用户名。
- ☐ 管理员密码：用于标识管理员的密码，通常情况下和管理员用户名一起进行身份验证。

管理员要进行操作前需要进行登录并通过数据进行身份验证，验证通过后才能够进行相应的数据更改，在对数据库进行基本的分析后，就能够创建相应的数据表进行数据存储。

### 28.2.2 数据表的创建

创建表可以通过 **SQL Server Management Studio** 视图进行创建也可以通过 **SQL Server Management Studio** 查询使用 **SQL** 语句进行创建。

#### 1. 事务表

在创建数据表之前首先需要创建 **guestbook** 数据库，创建完成后就能够进行其中的表的创建。在 **ASP.NET** 留言本中最为重要的是留言表，留言表和留言分类表可以统称为事务表，其中留言表结构如图 28-5 所示。

	列名	数据类型	允许空
?	id	int	<input type="checkbox"/>
	title	nvarchar(50)	<input checked="" type="checkbox"/>
	name	nvarchar(50)	<input checked="" type="checkbox"/>
	time	datetime	<input checked="" type="checkbox"/>
	[content]	nvarchar(MAX)	<input checked="" type="checkbox"/>
	reptitle	nvarchar(50)	<input checked="" type="checkbox"/>
	admin	nvarchar(50)	<input checked="" type="checkbox"/>
	reptime	datetime	<input checked="" type="checkbox"/>
	repcontent	nvarchar(MAX)	<input checked="" type="checkbox"/>
	classid	int	<input checked="" type="checkbox"/>
	userid	int	<input checked="" type="checkbox"/>
▶			<input type="checkbox"/>

图 28-5 留言表结构

从数据库中可以看出留言表中的字段信息，这些字段意义如下所示：

- ☐ **id**: 用于标识留言本的编号进行索引，为自动增长的主键。
- ☐ **title**: 用户留言的标题。
- ☐ **name**: 用户的名称。
- ☐ **time**: 用户留言的时间。
- ☐ **content**: 用户留言的内容。
- ☐ **reptitle**: 管理员回复留言的标题。
- ☐ **admin**: 管理员的名称。
- ☐ **reptime**: 管理员回复留言的时间。
- ☐ **repcontent**: 管理员回复的内容。
- ☐ **classid**: 用户留言所属的分类。
- ☐ **userid**: 留言所属的用户 ID。

创建数据表的 **SQL** 查询语句代码如下所示。

```
USE [guestbook]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[gbook](
    [id] [int] IDENTITY(1,1) NOT NULL,
    [title] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
    [name] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
    [time] [datetime] NULL,
    [content] [nvarchar](max) COLLATE Chinese_PRC_CI_AS NULL,
    [reptitle] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
    [admin] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
    [reptime] [datetime] NULL,
    [repcontent] [nvarchar](max) COLLATE Chinese_PRC_CI_AS NULL,
    [classid] [int] NULL,
    [userid] [int] NULL,
    CONSTRAINT [PK_gbook] PRIMARY KEY CLUSTERED
(
    [id] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
```

//创建 gbook 表

**gbook** 表为用户留言表，其中所有的留言数据都会存放在此表中。为了提高留言本程序的可扩展性，需要创建留言分类表进行留言程序的分类，其中留言分类表字段如下所示。

- ❑ **id:** 用于标识留言本分类的编号，为自动增长的主键。
- ❑ **classname:** 用于描述分类的名称，例如“客户服务”等。

创建数据表的 SQL 查询语句代码如下所示。

```
USE [guestbook]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[gbook_class](
    [id] [int] IDENTITY(1,1) NOT NULL,
    [classname] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
    CONSTRAINT [PK_gbook_class] PRIMARY KEY CLUSTERED
(
    [id] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
```

### 3. 验证表

验证表包括管理员验证表和用户注册表。用户注册表通常进行用户的验证包括注册和登录等操作，用户也能够通过注册表进行索引，而管理员验证表用于验证管理员的信息，验证后的管理员可以进行回复、删除等数据操作。管理员表字段如下所示。

- ❑ **id:** 用于标识管理员信息，为自动增长的主键。
- ❑ **username:** 用于标识管理员用户名。
- ❑ **password:** 用于标识管理员的密码，通常情况下和管理员用户名一起进行身份验证。

创建数据表的 SQL 查询语句代码如下所示。

```
USE [guestbook]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[admin](
    [id] [int] IDENTITY(1,1) NOT NULL,
    [admin] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
    [password] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
    CONSTRAINT [PK_admin] PRIMARY KEY CLUSTERED
(
    [id] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
//创建 admin 表
```

管理员表用于管理员的身份验证，以及确定操作人员的权限，对于用户而言需要一个表将用户的信息进行存储和读取，其字段如下所示。

- ❑ **id:** 用于标识用户 ID，为自动增长的主键。
- ❑ **username:** 用于保存用户的用户名，当用户登录时可以通过用户名验证。
- ❑ **password:** 用于保存用户的密码，当用户使用登录时可以通过密码验证。
- ❑ **sex:** 用于保存用户的性别。
- ❑ **pic:** 用于保存用户的个性头像。
- ❑ **IM:** 用于保存用户的 QQ/MSN 等信息。
- ❑ **information:** 用于展现用户的个性签名等资料。
- ❑ **others:** 用于保存用户的备注信息。

❑ ifisuser: 用于保存用户的状态，可以设置为通过审批和未通过等。

创建数据表的 SQL 查询语句代码如下所示。

```
USE [guestbook]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Register](
    [id] [int] IDENTITY(1,1) NOT NULL,
    [username] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
    [password] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
    [sex] [int] NULL,
    [picture] [nvarchar](max) COLLATE Chinese_PRC_CI_AS NULL,
    [IM] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
    [information] [nvarchar](max) COLLATE Chinese_PRC_CI_AS NULL,
    [others] [nvarchar](max) COLLATE Chinese_PRC_CI_AS NULL,
    [ifisuser] [int] NULL,
    CONSTRAINT [PK_Register] PRIMARY KEY CLUSTERED
(
    [id] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
```

//创建注册表

上述查询语句创建了注册表以存储用户的注册信息。在进行留言时，系统将使用用户注册表进行用户的身份验证，验证通过后的用户可以进行留言、索引等操作。

28.2.3 数据表关系图

系统数据库中需要进行约束，需要约束的表包括用户表、留言表和留言分类表，其约束可以使用 SQL Server Management Studio 视图进行编写，如图 28-6 所示。在创建数据表关系图后系统将弹出对话框进行关系的保存，如图 28-7 所示。

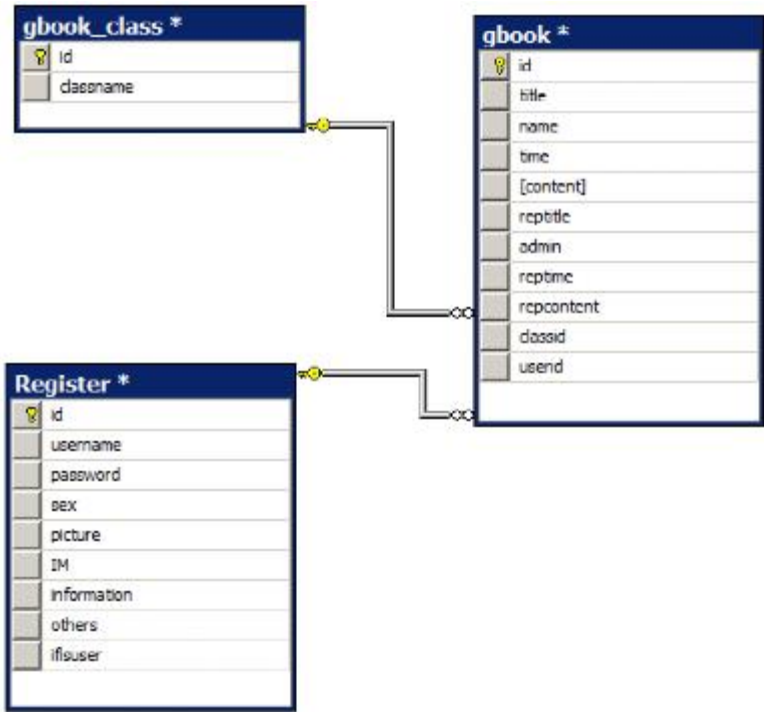


图 28-6 数据表关系图



图 28-7 保存关系

在进行关系的保存之后系统表就会被关系进行更改，其中的数据库创建的 SQL 语句也会相应的更改。作为数据库设计人员可以一开始设计毫无关系的数据表，然后在 SQL Server Management Studio 视图状态中



进行关系的设计，而无需编写复杂的数据表创建 SQL 语句。

注意：当进行了数据表的更改后，其数据库中的结构和数据库中表的一些信息就会被更改，为了保证其数据的完整性和规范性，必须按照规范进行数据操作。

## 28.3 系统公用模块的创建

在系统开发中，为了保证其系统的可扩展性和可维护性，通常将需要经常使用的部分创建成为系统的公用模块，系统的公用模块可以被系统中的任何页面或者类库进行调用，当需要进行更改时，可以修改通用模块进行低成本维护。

### 28.3.1 创建 CSS

CSS 作为页面布局的全局文件，可以进行 ASP.NET 留言本全局的布局的样式控制，通过使用 CSS 能够将页面代码和布局代码相分离，这样就能够方便的进行系统样式维护。右击现有项目，在下拉菜单中选择【添加】选项，然后在【添加】选项的下拉菜单中单击【新建项】选项以创建 CSS 样式表，如图 28-8 所示。



图 28-8 创建样式表

样式表可以统一存放在一个文件夹中，该文件夹能够进行样式表的统一存放和规划，以便系统可以使用不同的样式表。虽然样式表能够存放在系统的任何位置，但是为了文件的整洁，以及文件系统的可维护性建议存放在统一的文件夹中。在 CSS 文件中可以编写代码进行样式控制，示例代码如下所示。

```
body
{
    font-size:12px;
    font-family:Geneva, Arial, Helvetica, sans-serif;
    margin:0px 0px 0px 0px;
}
```

上述代码只是定义了一个 **body** 标签的样式，在后续开发过程中，如果需要进行样式控制可以随时更改此文件。

## 28.3.2 使用 SQLHepler

**SQLHepler** 是一个数据库操作的封装，使用 **SQLHepler** 类能够快速地进行数据的插入、查询、更新等操作而无需使用大量的 **ADO.NET** 代码进行连接。使用 **SQLHelper** 类为开发人员进行数据操作提供了极大的便利。在现有的系统中，在解决方案管理器中可以选择添加现有项添加现有的类库的引用，也可以通过自行创建类进行引用。在这里用户可以无需自行创建 **SQLHelper** 类，本书提供了 **SQLHelper** 类常用的精简版本，开发人员能够使用 **SQLHelper** 类进行高效的开发。

**注意：**本书提供的 **SQLHelper** 类在“源代码/附-SQLHELPER 类”文件夹中可以找到。

在使用现有的 **SQLHelper** 类之前，需要进行项目的创建，以便进行类的维护。在 **SQLHelper** 类中包含诸多静态方法，可以无需创建对象就能够使用 **SQLHelper** 类进行数据的插入、查询、更新、删除等操作，使用 **SQLHelper** 类也无需进行 **ADO.NET** 数据连接。**SQLHelper** 类是一个类，而不是项目，在 **Visual Studio 2008** 中可以创建新的类库项目进行分层开发，右击【解决方案管理器】选项，在下拉菜单中选择【添加新项目】选项，在弹出窗口中选择【类库】选项，如图 28-9 所示。

为了方便和容易阅读，新建类库项目的名称也被称为 **SQLHelper**，创建类库项目后可以删除自行创建的 **Class1.cs** 文件，在项目中添加现有项目，在计算机中找到需要的 **SQLHelper** 类即可。在使用 **SQLHelper** 类时，还需要添加命名空间 **System.Collections** 的引用，创建完成后的 **SQLHelper** 类库如图 28-10 所示。



图 28-9 创建类库项目

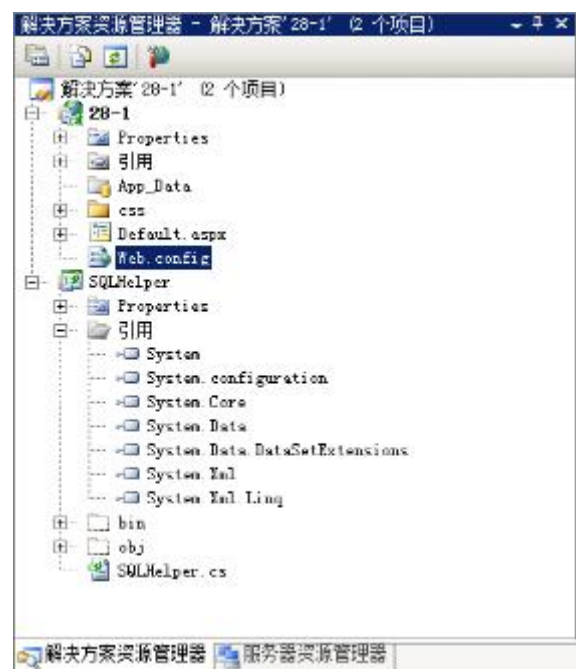


图 28-10 使用 SQLHelper

**注意：**为了更加方便的使用 **SQLHelper**，其类库的命名和命名空间的命名可以直接使用 **SQLHelper** 命名，通常情况下无需修改，如果开发人员需要修改，则需要在类库的属性和代码中修改相应的名称。

## 28.3.3 配置 Web.config

**Web.config** 文件为系统的全局配置文件，在 **ASP.NET** 中 **Web.config** 文件提供了自定义可扩展的系统配置，在 **Web.config** 文件中的 `<appSettings/>` 配置节可以配置自定义信息。当使用 **SQLHelper** 类进行数据辅助操作时，其中包含连接字符串代码，示例代码如下所示。

```
private static readonly string database = ConfigurationManager.AppSettings["database"].ToString();
private static readonly string uid = ConfigurationManager.AppSettings["uid"].ToString();
private static readonly string pwd = ConfigurationManager.AppSettings["pwd"].ToString();
private static readonly string server = ConfigurationManager.AppSettings["server"].ToString();
private static readonly string condb = "server=" + server + ";database=" + database + ";uid=" + uid
```

```
+ "";pwd="" + pwd + """;
```

```
//配置连接字符串
```

**ConfigurationManager.AppSettings** 能够获取 **Web.config** 文件中<appSettings/>配置节的相应的字段的值，上述代码分别获取<appSettings/>配置节中 **database**、**uid**、**pwd**、**server** 的值进行数据连接。在 **Web.config** 文件中，可以配置<appSettings/>配置节以使用 **SQLHelper**，示例代码如下所示。

```
<appSettings>
  <add key="server" value="(local)"/>
  <add key="database" value="guestbook"/>
  <add key="uid" value="sa"/>
  <add key="pwd" value="sa"/>
</appSettings>
```

```
//编辑 server 项
//编辑 guestbook 项
//编辑 uid 项
//编辑 pwd 项
```

在配置了 **Web.config** 中<appSettings/>配置的信息后，**SQLHelper** 类就能够使用<appSettings/>配置节中的字段和值。

**注意：**在使用 **SQLHelper** 类时，也可以自定义数据库连接字符串，在 **SQLHelper** 类中直接使用。当需要在项目中使用 **SQLHelper** 类时，还需要添加 **SQLHelper** 的引用。

## 28.4 系统界面和代码实现

在进行了系统设计和数据库设计之后就能够进行编码的实现，编码实现包括系统界面的编码实现和逻辑编码的实现，系统界面代码可以使用 **CSS** 进行全局样式控制，而逻辑编码实现需要在页面中进行逻辑控制。

### 28.4.1 留言板用户控件

在留言板的数据显示中，可以编写用户控件进行数据显示并在相应的页面中使用该用户控件，用户控件能够极大的方便开发人员进行开发维护。

用户控件是使用现有的服务器控件进行控件的制作，相比于自定义控件而言，用户控件更加容易和方便的进行开发和使用。自定义控件使用现有的服务器控件进行组合，而使用现有的数据源控件能够方便的进行分页、更新、删除等操作，所以在留言板中可以选择用户控件进行数据呈现。

右击现有项目，选择【添加】选项，在下拉菜单中单击【新建项】选项，在弹出窗口中选择【Web 用户控件】选项进行用户控件的创建，创建完成后可以进行用户控件的布局和样式控制并拖动数据源控件和数据绑定控件进行数据呈现和操作，示例代码见光盘中源代码\第 28 章\28-1\28-1\control\GbookList.aspx。

其中，代码使用了 **DataList** 数据控件进行数据呈现，**DataList** 控件可以不使用数据源控件中的更新、删除、插入等操作进行数据操作，在留言本控件中，**DataList** 控件只需要执行数据的呈现和分页即可。分页代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    PagedDataSource objPds = new PagedDataSource();
    objPds.DataSource = this.SqlDataSource1.Select(new DataSourceSelectArguments());
    objPds.AllowPaging = true;
    objPds.PageSize = 20;
    int CurPage;
    Label2.Visible = false;
    Label4.Visible = false;
    if (Request.QueryString["Page"] != null)
    {
        CurPage = Convert.ToInt32(Request.QueryString["Page"]);
    }
```

```
//使用分页类
//设置是否分页
//设置分页个数
//设置页码
//隐藏导航
//隐藏导航
//获取传递参数
//获取参数
```

```
    }
    else
    {
        CurPage = 1; //设置默认页数
    }
    objPds.CurrentPageIndex = CurPage - 1; //设置页码
    Label2.Visible = true; //显示导航
    Label4.Visible = true; //显示导航
    Label3.Text = "<a href=\"Gbook.aspx?cid=\" + Request.QueryString["cid"] + \"\">首页</a>";
    Label2.Text = "<a href=\"Gbook.aspx?page=\" + Convert.ToString(CurPage + 1) + \" &cid=\"
        + Request.QueryString["cid"] + \"\">下一页</a>"; //显示分页
    Label4.Text = "<a href=\"Gbook.aspx?page=\" + Convert.ToString(CurPage - 1) + \" &cid=\"
        + Request.QueryString["cid"] + \"\">上一页</a>"; //显示分页
    if (CurPage == 1) //是否只有一页
    {
        Label4.Visible = false; //屏蔽下一页
    }
    if (objPds.IsLastPage) //是否最后一页
    {
        Label2.Visible = false; //屏蔽上一页
    }
    DataList1.DataSourceID = ""; //清空绑定
    DataList1.DataSource = objPds; //选择数据源
    DataList1.DataBind(); //数据重绑定
}
```

使用数据分页类能够对 **ListView** 控件进行分页操作，当数据库中的信息过多时用户可以使用分页进行留言的查看，当管理员进行留言本的回复时，也可以使用分页操作对原来未进行回复的操作进行选择和回复。在创建自定义控件时，其数据源控件可以不必支持数据更新、插入和删除等操作。另外，在 **SqlDataSource** 数据源控件中还使用了 **Web.config** 中的数据连接字符串进行数据呈现和操作的支持。

### 28.4.2 管理员登录实现

管理员登录的实现可以使用前面开发的登录模块进行管理员身份的验证开发，这里也可以简单的使用服务器控件进行管理员登录的实现。管理员登录页面 **HTML** 代码见光盘中源代码\第 28 章\28-1\28-1\admin\login.aspx。

页面代码中包含了一个样式类为 **admin\_login** 的 **DIV** 层，该层用于管理员登录界面中控件的存放和样式控制，其中 **CSS** 样式控制代码如下所示。

```
.admin_login
{
    margin:100px auto;
    width:500px;
    border:1px solid #ccc;
    background:#f0f0f0;
}
```

上述代码定义了该层的背景颜色、边框粗细、宽度以及外间距，布局后如图 28-11 所示。

图 28-11 管理员登录页面布局



当管理员单击【Login】按钮时会进行身份验证，如果管理员身份正确则会给管理员分配一个 **Session** 对象，在其他的页面中需要使用 **Session** 对象进行身份验证。示例代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    string strsql = "select * from admin where admin='" + TextBox1.Text + "' and password='"
        + TextBox2.Text + "'"; //使用 SQL 语句
    SqlDataReader sdr = SQLHelper.SQLHelper.ExecReader(strsql); //使用 SqlDataReader
    if (sdr.Read()) //判断是否有数据
    {
        Session["admin"] = TextBox1.Text; //给予 Session
        Response.Redirect("../default.aspx"); //页面跳转
    }
    else
    {
        Label1.Text = "无法登录,用户名或密码错误"; //提示错误
    }
}
```

上述代码使用了 **SqlDataReader** 类和 **SQLHelper** 类进行数据查询，从上述代码可以看出使用 **SQLHelper** 类减少了大量的代码，使用 **ADO.NET** 进行数据操作还需要进行数据连接、创建适配器、进行数据集填充和数据集行数的判断，而使用 **SQLHelper** 类精简了该过程，极大的简化了开发人员的开发。

## 28.4.3 用户注册登录实现

用户在留言之前需要进行登录，如果用户没有账号就需要在登录之前进行注册操作，注册操作可以使用注册模块进行数据操作，这里只需要简单的进行数据插入即可，注册页面 **HTML** 核心代码见光盘中源代码\第 28 章\28-1\28-1\login.aspx。

其中的页面代码实现了用户进行注册时所必要的控件以便用户能够进行快速注册，在用户注册时，用户名和密码为必填项，而其他为选填项目，当用户单击按钮控件进行注册时，会执行相应的注册事件进行数据库操作，示例代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    try
    {
        string strsql =
            "insert into register (username,password,sex,picture,IM,information,others,ifisuser) values ('" +
                TextBox1.Text + "','" + TextBox2.Text + "','" + DropDownList1.Text + "','" +
                TextBox3.Text + "','" + TextBox4.Text + "','" + TextBox5.Text + "','" +
                TextBox6.Text + "',1)"; //编写 INSERT 语句
        SQLHelper.SQLHelper.ExecNonQuery(strsql); //执行数据操作
        if (!String.IsNullOrEmpty(Request.QueryString["id"])) //选择跳转
        {
            Response.Redirect("Gbook.aspx?id=" + Request.QueryString["id"]);
        }
        else
        {
            Response.Redirect("default.aspx"); //跳转到默认页
        }
    }
    catch
    {
        Response.Redirect("default.aspx"); //错误也跳转到默认页
    }
}
```

上述代码执行数据操作只用了一条语句，并没有进行 **ADO.NET** 中的数据连接、**Command** 对象的创建以及操作，只使用了一行代码就简化了数据操作。当用户进行注册时，可以在留言页面进行注册，也可以在首页进行注册，如果在留言页面进行注册跳转，希望注册后还能够直接跳回留言页面，如果用户在首页进行注册，注册完毕后应跳转回首页然后进行留言本的选择。

## 28.4.4 用户登录实现

用户登录实现同管理员登录相同，管理员登录时进行管理员用户名和密码的查询，如果存在数据则说明存在管理员，并赋予管理员相应的 **Session** 对象。对于用户登录而言，其代码基本相同，登录事件处理代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    string strsql = "select * from register where username='" + TextBox1.Text + "' and password='"
        + TextBox2.Text + "'"; //查询 SQL
    SqlDataReader sdr = SQLHelper.SQLHelper.ExecReader(strsql); //执行查询
    if (sdr.Read()) //判断用户
    {
        Session["username"] = TextBox1.Text; //给予权限
        Session["userid"] = sdr["id"].ToString(); //给予权限
        if (!String.IsNullOrEmpty(Request.QueryString["id"])) //选择跳转
        {
            Response.Redirect("Gbook.aspx?id=" + Request.QueryString["id"]); //跳转
        }
        else
        {
            Response.Redirect("default.aspx"); //默认跳转
        }
    }
    else
    {
        Label1.Text = "无法登录,用户名或密码错误"; //抛出异常
    }
}
```

上述代码执行了查询，将查询填充到 **SqlDataReader** 对象中，如果数据库中包含相应的查询数据，则 **SqlDataReader** 对象的 **Read** 方法会返回 **true**，否则返回 **false**。

用户登录界面可以同管理员界面一样，由于篇幅限制，这里就不再进行登录界面的样式布局和样式控制，直接使用管理员登录界面样式进行用户登录的样式控制。同样为了提高用户体验，如果用户在登录前已经打开了某个留言本，则在登录后能够直接跳转到该留言本，而如果用户在首页进行登录操作，则用户依旧会跳转回首页。

## 28.4.5 留言本界面布局

留言本界面是留言本项目中最为重要的页面，在该页面用户能够进行留言操作并可以查看管理员对自己留言的回复，而管理员能够在该页面进行数据管理和回复。留言本页面 **HTML** 核心代码见光盘中源代码\第 28 章\28-1\28-1\Gbook.aspx。

留言本界面使用了 **DIV+CSS** 让页面的样式更加丰满、用户体验更加友好，在留言本界面中使用了前面制作的控件，并且能够支持分页等操作。在页面以及自定义控件中定义了 **CSS** 样式，通过编写 **CSS** 样式文件能够进行控件和页面的样式控制，**CSS** 代码如下所示。

```
.gbook_main_title //定义留言本标题界面
{
```

```
margin:0px auto;
margin-top:50px;
width:800px;
border:1px solid #ccc;
background:white url(..images/top.png);
height:200px;
}
.gbook_main                                     //定义留言本主界面
{
    margin:5px auto;
    width:800px;
    border:1px solid #ccc;
    background:white;
}
.gbook_banner                                   //定义留言本导航界面
{
    margin:5px auto;
    width:790px;
    border:1px solid #ccc;
    background:white;
    padding:5px 5px 5px 5px;
}
.left                                           //定义留言本侧边界面
{
    width:200px;
    float:left;
}
.right                                          //定义留言本主界面
{
    width:579px;
    margin-left:10px;
    float:left;
    border-left:1px dashed #ccc;
    padding:5px 5px 5px 5px;
}
.copyright                                     //定义版权界面
{
    margin:5px auto;
    width:800px;
    border:1px solid #ccc;
    background:white;
    font-size:10px;
    text-align:center;
}
```

上述 **CSS** 代码定义了页面中的样式，能够让页面的样式更加友好。在留言本页面中，使用了前面制作的 用户控件，用户控件的模板中同样使用 **CSS** 进行了定义，通过编写相应的 **CSS** 代码能够控制控件中数据的呈现方式，**CSS** 代码如下所示。

```
.g_title                                       //定义控件标题
{
    background:#f0f0f0;
    padding:5px 5px 5px 5px;
}
.g_content                                    //定义控件内容界面
{
    padding:5px 5px 5px 5px;
}
.g_reply                                     //定义控件回复界面
```

```
{
    padding:5px 5px 5px 5px;
}
.g_table //定义控件循环表格
{
    border:1px solid #ccc;
    margin:5px 5px 5px 5px;
    width:98%;
}
```

上述代码定义了留言本中用户控件的样式，在数据呈现的过程中，系统会将样式和页面代码整合在一起呈现给用户，留言本界面布局如图 28-12 所示。



图 28-12 ASP.NET 留言本主界面布局

28.4.6 留言功能实现

当用户单击【留言】按钮后，用户就能够进行留言操作，使用 **SQLHelper** 类只需要编写 **INSERT** 语句就能够实现数据插入，示例代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    try
    {
        //编写执行插入的 INSERT 语句
        string strsql =
            "insert into gbook (title,name,time,content,reptitle,admin,reptime,repcontent,classid,userid)
            values ('" + TextBox2.Text + "','" + Session["username"].ToString() + "','" + DateTime.Now + "','"
            + TextBox1.Text + "','" + DateTime.Now + "','" + Request.QueryString["cid"] + "','" +
            Session["userid"].ToString() + "')"; //编写 INSERT
        SQLHelper.SQLHelper.ExecNonQuery(strsql); //执行 SQL 语句
        Response.Redirect("Gbook.aspx?cid=" + Request.QueryString["cid"]); //页面跳转
    }
    catch
    {
        //编写错误处理 //自定义错误处理
    }
}
```

在进行留言功能的实现前，首先需要判断用户是否注册或登录，如果用户没有注册或登录，那么用户就只能查看相应的留言而无权进行留言；如果进行注册并登录，则用户在留言本页面能够进行留言操作。所以在页面加载时就必须对用户的身份进行验证判断，示例代码如下所示。



```
protected void Page_Load(object sender, EventArgs e)
{
    if (Session["username"] == null || Session["userid"] == null)           //如果不是用户
    {
        Panel1.Visible = false;                                           //留言区域不可见
    }
}
```

上述代码在页面加载时被执行，如果页面加载时发现执行页面操作的用户并不是网站的用户就会隐藏留言区域，反之将会呈现留言区域以供用户进行留言操作。

### 28.4.7 回复功能实现

当管理员进行留言查看时，对于相应的留言可以选择对用户的留言进行回复，管理员能够通过页面中的回复超链接进行回复。留言页面 **HTML** 代码如下所示。

```
<body style="background:white url(..images/bg.png) repeat-x;">
    <form id="form1" runat="server">
        <div class="gbook_main">
            回复留言:<br />
            <asp:TextBox ID="TextBox1" runat="server" Height="150px" TextMode="MultiLine"
            Width="100%"></asp:TextBox>
            <br />
            <asp:Button ID="Button1" runat="server" Text="回复留言" />
        </div>
    </form>
</body>
```

当管理员单击【回复】超链接时，会跳转到该回复页面，该页面能够执行相应的留言的回复，当用户单击【回复】超链接时，必须在回复页面进行判断。如果是管理员，则允许进行回复，如果不是管理员，则跳转回留言页面，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    if (Session["admin"] == null)                                           //如果不是管理员
    {
        Response.Redirect("../Gbook.aspx?cid=" + Request.QueryString["cid"] + ""); //跳回页面
    }
}
```

如果身份验证后判断确实是管理员，则能够打开回复页面进行回复。当管理员回复后，单击按钮控件就能够进行回复并跳转到相应的页面，示例代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    try
    {
        string strsql = "update gbook set repcontent='" + TextBox1.Text + "',reptime='" + DateTime.Now
            + "',admin='" + Session["admin"].ToString() + "' where id='" +
            Request.QueryString["id"] + "'";                                //编写更新
        SQLHelper.SQLHelper.ExecNonQuery(strsql);                        //执行更新
        Response.Redirect("../Gbook.aspx?cid=" + Request.QueryString["cid"] + ""); //页面跳转
    }
    catch
    {
        //异常处理
    }
}
```

上述代码通过传递的参数进行更新操作，在用户控件中，每一列数据都包含回复和删除两个操作的超

链接，这两个操作所在的页面需要通过获取控件中传递过来的参数进行数据的查询和操作，页面通过获取 **cid** 参数进行页面的跳转，而通过获取 **id** 的参数进行相应的数据处理。

### 28.4.8 删除功能的实现

在执行回复和删除操作时，页面通过获取 **cid** 参数进行页面的跳转，而通过获取 **id** 的参数进行相应的数据处理。删除页面并不需要进行数据的呈现或 **HTML** 的呈现，删除页面只需要进行数据的处理。当管理员执行删除操作时，同样需要进行身份验证，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    if (Session["admin"] == null) //如果不是管理员
    {
        Response.Redirect("../Gbook.aspx?cid=" + Request.QueryString["cid"] + ""); //跳回页面
    }
}
```

如果验证通过，则能够执行删除操作，删除操作通过传递过来的 **id** 参数进行数据删除，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    if (Session["admin"] == null) //管理员权限判断
    {
        Response.Redirect("../Gbook.aspx?cid=" + Request.QueryString["cid"] + ""); //页面跳转
    }
    else if
        (String.IsNullOrEmpty(Request.QueryString["cid"]))
        || String.IsNullOrEmpty(Request.QueryString["id"])) //如果参数不为空
    {
        Response.Redirect("../default.aspx"); //页面跳转
    }
    else
    {
        try
        {
            string strsql = "delete from gbook where id=" + Request.QueryString["id"] + "";
            SQLHelper.SQLHelper.ExecNonQuery(strsql); //执行删除
            Response.Redirect("../Gbook.aspx?cid=" + Request.QueryString["cid"] + ""); //页面跳转
        }
        catch
        {
            Response.Redirect("../Gbook.aspx?cid=" + Request.QueryString["cid"] + ""); //页面跳转
        }
    }
}
```

在删除页面中可以直接进行数据删除而无需通过 **HTML** 呈现或页面显示进行数据删除，当数据删除后可以直接跳转到相应分类的页面。

注意：在执行删除操作时，需要判断传递过来的参数是否存在或安全，例如这里判断传递的 **cid** 和 **id** 的值是否存在，如果不存在则视用户的操作为非法操作，即跳转到安全的页面。

### 28.4.9 用户索引实现

当用户进行留言后，可能会在很长一段时间内不会再次进行网站的访问和登录，当经过很长的时间后，用户再次返回该网站进行访问，就会比较关心自己的留言是否被回复。而留言页面中的数据排序是根据留言 **ID** 进行倒序的，在经过很长一段时间后，用户的留言很可能就在很多页之后了。为了方便用户能够在网站中进行留言的检索，可以为用户创建索引，方便用户进行较早留言的查看，用户索引页面 **HTML** 代码见光盘中源代码\第 28 章\28-1\28-1\userindex.aspx。

上述代码同样使用 **ListView** 控件进行数据呈现和显示，而不同的是数据源控件的配置。在配置时，数据源控件需要获取传递的 **uid** 参数进行查询，数据源控件代码如下所示。

```
<asp:SqlDataSource ID="SqlDataSource1" runat="server"
    ConnectionString="<%%$ ConnectionStrings:guestbookConnectionString %>"
    SelectCommand="SELECT gbook.* FROM gbook,register WHERE (gbook.userid = @userid) AND
        (register.id=gbook.userid) ORDER BY gbook.id DESC">
    <SelectParameters>
        <asp:QueryStringParameter Name="userid" QueryStringField="uid" />
        <!--获取传递的参数进行查询--!>
    </SelectParameters>
</asp:SqlDataSource>
```

数据源控件查询的返回的值同留言本中查询的返回类型相同，但是索引页面查询的结果与留言本中查询的条件不同，留言本中的查询条件是按照留言本分类进行查询，而用户的索引是通过用户的 **ID** 号进行查询。当数据填充 **ListView** 控件后，同样需要让索引页面的 **ListView** 控件支持分页操作，分页代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    ..... //前面基本相同
    .....
    Label3.Text = "<a href=\"UserIndex.aspx?uid=\" + Request.QueryString[\"uid\"] + \"\">首页</a>";
    Label2.Text = "<a href=\"UserInde.aspx?page=\" + Convert.ToString(CurPage + 1) + \"&uid=\"
        + Request.QueryString[\"uid\"] + \"\">下一页</a>"; //页面和参数不同
    Label4.Text = "<a href=\"UserInde.aspx?page=\" + Convert.ToString(CurPage - 1) + \"&uid=\"
        + Request.QueryString[\"uid\"] + \"\">上一页</a>"; //页面和参数不同
    if (CurPage == 1) //是否只有一页
    {
        Label4.Visible = false; //屏蔽下一页
    }
    if (objPds.IsLastPage) //是否最后一页
    {
        Label2.Visible = false; //屏蔽上一页
    }
    DataList1.DataSourceID = ""; //清空绑定
    DataList1.DataSource = objPds; //选择数据源
    DataList1.DataBind(); //数据重绑定
}
```

**ListView** 控件的分页操作代码基本相同，开发人员值需要将页面的从 **Gbook.aspx** 改为 **UserIndex.aspx**，然后将传递的参数更改即可。

## 28.5 用户体验优化

在基本的系统功能模块编写完毕后，还需要对现有的功能模块进行优化。优化不仅仅包括应用程序代码的优化，还包括系统的界面以及用户体验的优化。为了能够提升用户体验，这里就需要使用 **AJAX** 对系统进行功能进行优化。

## 28.5.1 AJAX 留言实现

在 ASP.NET 3.5 中，系统已经为 AJAX 提供了原生的支持，开发人员能够直接拖动控件进行 AJAX 应用程序的开发而无需复杂的功能实现。AJAX 应用有一个优点，就是能够让页面进行无刷新的数据交互，这样就能够提高用户体验度。

### 1. AJAX 留言页面

打开留言页面进行 AJAX 控件布局。在留言页面中，最主要的数据操作就是留言的实现，这里的留言实现所需要的控件都需要使用 AJAX 控件进行“包裹”和“承载”。留言控件已经存放在 Panel 控件中，用于不同身份权限的 HTML 呈现。同样，AJAX 控件也可以陈放在 Panel 控件中，示例代码见光盘中源代码\第 28 章\28-1\28-1\Gbook.aspx。

其中，代码在 Panel 控件中加入了 AJAX 控件以保证页面数据能够局部刷新而不会造成页面的其他数据的刷新。在进行留言时，还需要对用户进行提示以告知用户正在留言。为了实现这样的功能，就需要在 UpdatePanel 控件中加入 UpdateProgress 控件，示例代码如下所示。

```
<asp:UpdateProgress ID="UpdateProgress1" runat="server">
    <ProgressTemplate>
        
    </ProgressTemplate>
</asp:UpdateProgress>
```

上述代码在 UpdateProgress 控件中插入了“等待”图片，该图片会在系统执行过程中呈现。当用户在留言时，局部的数据会发送到数据库进行数据发送和回传，在这个过程中，无疑是需要时间的。使用一个“等待”图片或者一段“等待”字样能够提示用户，从而提高用户体验。UpdateProgress 控件如图 28-13 所示。

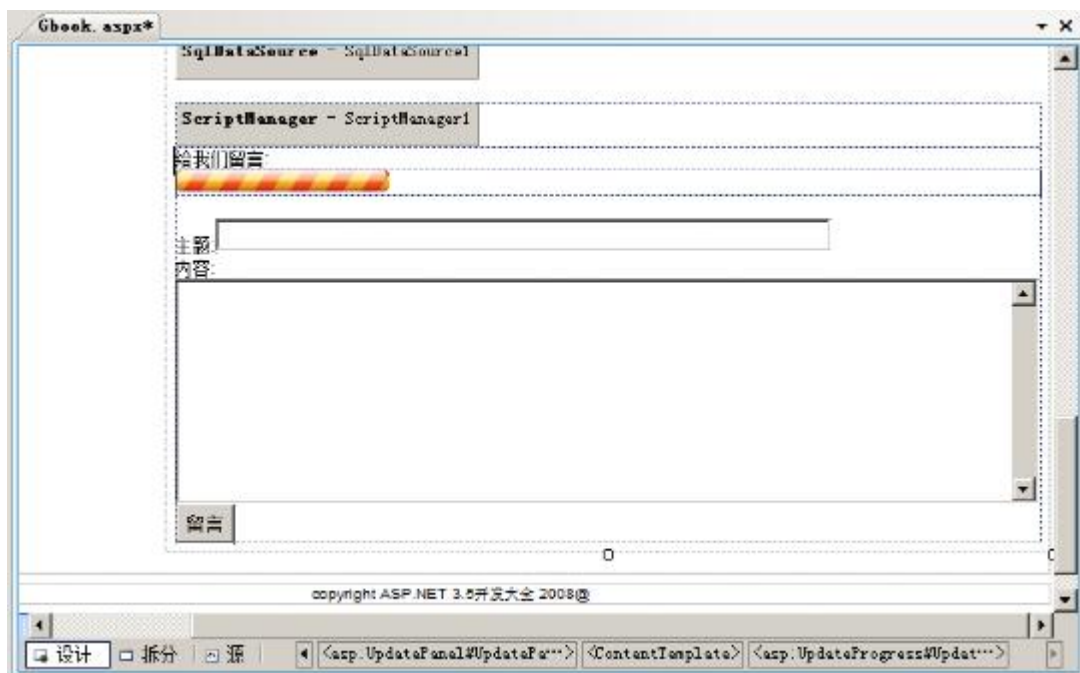


图 28-13 UpdateProgress 控件的使用

在留言过程中，如果留言本的数据量非常的小，那么留言本中的 UpdateProgress 控件就不会呈现在用户面前。因为当留言本的数据非常小时，其数据的交互和页面的往返过程也会变得非常的简单，这样就会造成 UpdateProgress 控件无法呈现。

为了让 UpdateProgress 控件呈现在用户面前，开发人员可以使用线程进行控制。在用户单击【留言】按钮时，系统会停止 3 秒以便模拟大量数据时等待的过程。示例代码如下所示。

```
try
{
    System.Threading.Thread.Sleep(3000); //系统休眠
    string strsql
    = "insert into gbook (title,name,time,content,reptitle,admin,reptime,repcontent,classid,userid)
    values ('" + TextBox2.Text + "','" + Session["username"].ToString() + "','" + DateTime.Now + "','"
    + TextBox1.Text + "','" + DateTime.Now + "','" + Request.QueryString["cid"] + "','" +
```



```
Session["userid"].ToString() + "");                                //编写 SQL 语句
SQLHelper.SQLHelper.ExecNonQuery(strsql);                        //执行 SQL 语句
Response.Redirect("Gbook.aspx?cid=" + Request.QueryString["cid"]); //页面跳转
    }
```

上述代码通过使用 **System.Threading.Thread.Sleep** 方法进行系统休眠以达到系统等待的目的，从而能够让 **UpdateProgress** 控件呈现在浏览器当中以提示用户操作正在执行。

注意：在该代码段中使用 **System.Threading.Thread.Sleep** 方法进行系统休眠在实际的应用程序开发中是不可取的，这里使用 **System.Threading.Thread.Sleep** 方法进行系统休眠是为了模拟大量数据时页面的信息提示。

在对页面进行了 **AJAX** 控件的布局后，还需要对操作进行修改。这里值得一题的是，当使用了 **AJAX** 控件执行数据操作，**AJAX** 控件在执行数据的过程中的状态都是“激活”的，如图 28-14 所示。

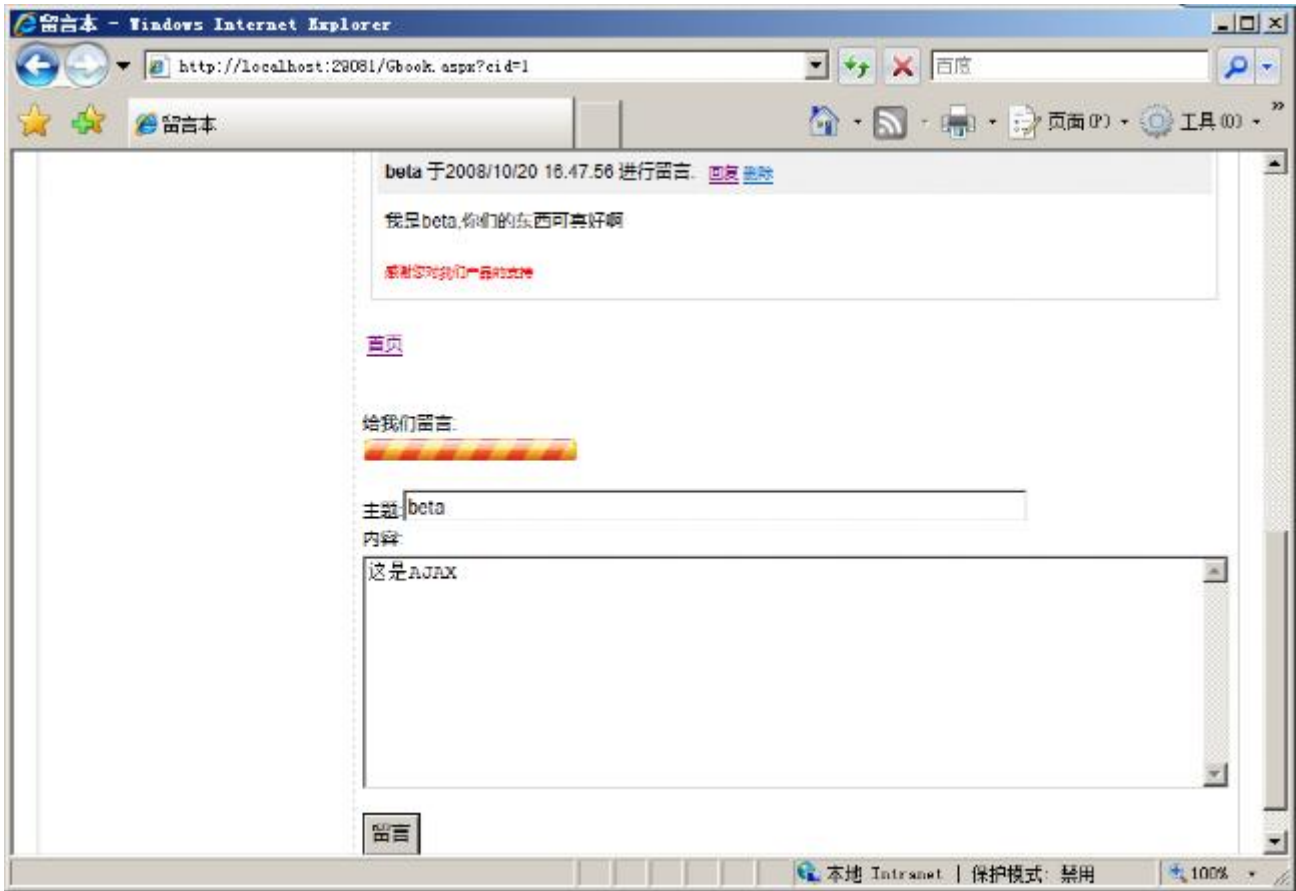


图 28-14 激活的控件状态

从图 28-14 中可以看出，当用户单击了【留言】按钮后，系统会执行相应的数据操作进行数据的增加，但是其中【留言】按钮依旧可以点击。在“等待”过程中，如果用户重复的单击【留言】按钮，则会造成数据的重复操作，当用户在执行过程中单击多个【留言】按钮时，系统也会添加同样多的数据，这不仅会造成数据冗余还会降低用户体验。

## 2. AJAX 留言页代码控制

为了解决以上问题，在 **AJAX** 留言页面中需要通过代码控制进行数据冗余的防止。这里可以暂时归纳成以下操作步骤：

- ❑ 用户单击前：用户单击前控件的状态是“激活”的，这也就是说用户可以单击按钮控件进行数据的操作。
- ❑ 用户单击时：用户单击时控件的状态应该是“非激活”的，这也就是说如果用户正在提交留言，那么就应该屏蔽控件的信息防止用户重复按键。
- ❑ 用户单击后：当用户单击后，其控件状态应该恢复成“激活”状态，这样用户就能够再次进行数据的提交。

从以上步骤可以看出，用户在执行单击时，相应的控件应该是“非激活”状态。这里最主要的是按钮

控件。当用户单击【留言】按钮时，相应的数据会执行并产生页面回传，在这个过程中，应该使用操作屏蔽【留言】按钮防止用户再次进行提交，示例代码如下所示。

```
StringBuilder sb = new StringBuilder(); //声明字符串
sb.Append("if (typeof(Page_ClientValidate) == 'function')
{ if (Page_ClientValidate() == false) { return false; }};"); //编写 javascript 脚本
sb.Append("this.disabled = true;"); //编写函数体
sb.Append(this.ClientScript.GetPostBackEventReference(Button1, "")); //判断是否回传
sb.Append(";");
sb.Append("SetUIStyle();"); //设置 UI 的样式
Button1.Attributes.Add("onclick", sb.ToString()); //控件添加属性
```

上述代码在代码页面中为控件添加了属性。当用户单击【留言】按钮并触发了相应的数据插入事件时，在回传过程中【留言】按钮的激活状态都会被修改为“非激活”，以阻止用户进行重复的单击操作，如图 28-15 所示。



图 28-15 控件激活状态

从图 28-15 可以看出，当用户单击【留言】按钮后，【留言】按钮变灰，即无法单击。如果用户多次单击【留言】按钮，系统也会将多次单击认定为是一次单击，从而只执行一次数据操作。这样不仅避免了数据冗余，同样还提高了用户体验。

注意：上述代码段可以直接编写在 Page\_Load 事件中。当页面初次被加载时，就需要对相应的控件进行属性的添加。这样在页面的执行中，该控件都会具备相应的属性。

28.5.2 AJAX 数据重绑定

在优化了 AJAX 留言功能之后可以发现当用户单击【留言】控件后，系统会跳转到该页面以便数据的重绑定，这样同样造成了页面全局刷新。当用户单击【留言】控件后，系统希望页面能够不跳转而重新加载页面即可执行数据的重绑定，这里同样需要使用 AJAX 进行局部刷新。

为了能够让数据能够进行局部刷新，这里需要将留言显示控件放置在 UpdatePanel 控件中，示例见光盘源代码\第 28 章\28-1\28-1\Gbook.aspx。

将留言显示控件放置在 UpdatePanel 控件中之后，就能够进行页面的局部刷新。值得注意的时，页面数据的重绑定的过程比较简单，只需要将控件进行重新加载即可。示例代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
```

```
{
    try
    {
        System.Threading.Thread.Sleep(3000); //模拟过程
        string strsql
        = "insert into gbook (title,name,time,content,replytitle,admin,replytime,replycontent,classid,userid)
        values ('" + TextBox2.Text + "','" + Session["username"].ToString() + "','" + DateTime.Now + "','"
        + TextBox1.Text + "','" + DateTime.Now + "','" + Request.QueryString["cid"] + "','" +
        Session["userid"].ToString() + "')"; //生成 SQL 语句
        SQLHelper.SQLHelper.ExecNonQuery(strsql); //执行 SQL 语句
        GbookList1.Reload(); //执行重加载方法
        //Response.Redirect("Gbook.aspx?cid=" + Request.QueryString["cid"]);
    }
    catch
    {
        //编写错误处理
    }
}
```

上述代码使用了 **GbookList** 控件的重加载方法，该方法必须要在 **GbookList** 控件中进行编写实现才能够调用。**GbookList** 控件中的 **Reload** 方法编写如下所示。

```
public void Reload()
{
    PagedDataSource objPds = new PagedDataSource(); //设置分页数据
    objPds.DataSource = this.SqlDataSource1.Select(new DataSourceSelectArguments());
    objPds.AllowPaging = true; //设置分页属性为 true
    objPds.PageSize = 20; //设置分页数
    int CurPage; //设置页码
    Label2.Visible = false; //重新清空统计
    Label4.Visible = false; //重新清空统计
    if (Request.QueryString["Page"] != null) //判断传递参数
    {
        CurPage = Convert.ToInt32(Request.QueryString["Page"]); //参数的显式转换
    }
    else
    {
        CurPage = 1; //设置默认页面
    }
    objPds.CurrentPageIndex = CurPage - 1; //设置分页索引
    Label2.Visible = true; //开始统计信息
    Label4.Visible = true; //开始统计信息
    Label3.Text = "<a href='\"Gbook.aspx?cid=" + Request.QueryString["cid"] + "\">首页</a>";
    Label2.Text = "<a href='\"Gbook.aspx?page=" + Convert.ToString(CurPage + 1) + "&cid=" +
    Request.QueryString["cid"] + "\">下一页</a>"; //设置分页
    Label4.Text = "<a href='\"Gbook.aspx?page=" + Convert.ToString(CurPage - 1) + "&cid=" +
    Request.QueryString["cid"] + "\">上一页</a>"; //设置分页
    if (CurPage == 1) //判断是否为第一页
    {
        Label4.Visible = false;
    }
    if (objPds.IsLastPage) //判断是否为最后一页
    {
        Label2.Visible = false;
    }
    DataList1.DataSourceID = ""; //清空绑定数据源
    DataList1.DataSource = objPds; //重新绑定数据源
    DataList1.DataBind(); //数据源重绑定
}
```

上述代码实现了用户控件的数据重绑定，当执行该方法时，数据绑定控件中的数据将会被“刷新”并呈现在客户端浏览器中。在使用该用户控件的页面中，可以使用该用户控件的 **Reload** 方法进行数据刷新。当用户单击【留言】按钮时，**AJAX** 控件能够完成页面的局部刷新，从而实现了用户无需进行页面跳转即可进行数据查看的功能，如图 28-16 所示。

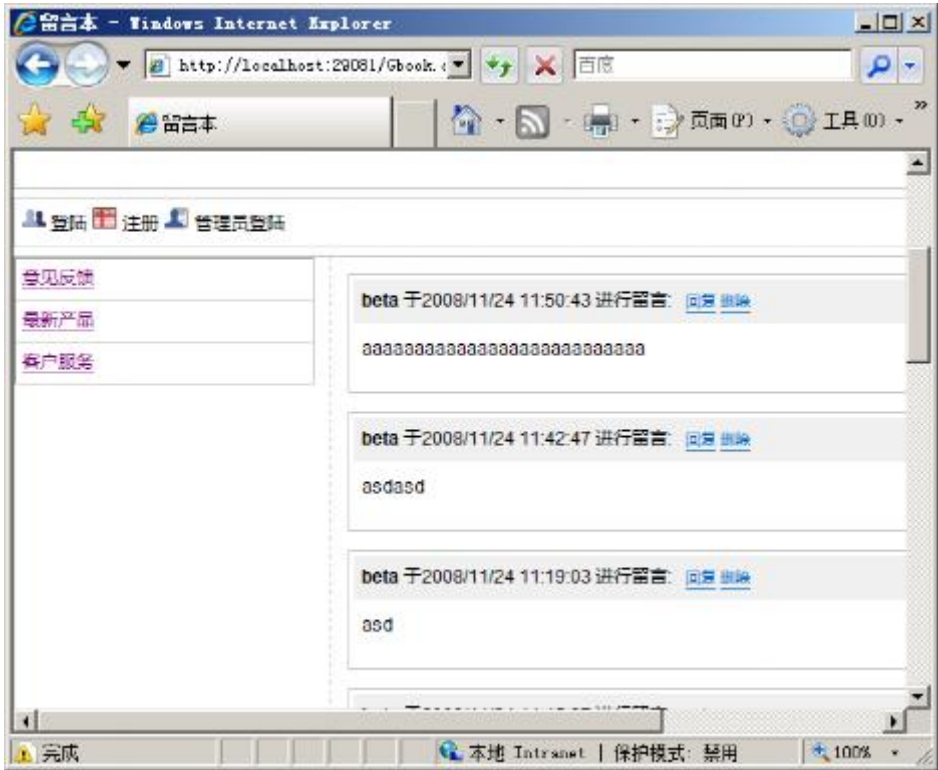


图 28-16 无刷新数据重绑定

注意：在执行无刷新数据重绑定时，依旧推荐能够使用 **UpdateProgress** 控件进行执行状态的呈现以提高用户体验。

28.5.3 系统导航实现

导航在系统中非常的重要，良好的导航能够方便的让用户选择【登录】、【注册】等操作而无需手动的编写地址，这样能够提高网站的用户体验。对于留言本系统而言，留言本需要进行登录操作才能够在留言本中执行数据操作，另外，留言本系统中还包括不同的用户权限，例如用户和管理员。对于不同的用户权限所需要展示的导航也不尽相同。

对于不同的权限，系统希望权限不同的用户所看到的导航也不尽相同。在前面的代码实现中，实现了固定的导航，实现了登录、注册、管理员登录等操作。值得注意的是，如果用户是一个管理员，那么这个导航应该是合理的，而如果用户不是管理员而是一个普通用户，【管理员登录】选项是不应该被呈现的。系统导航可以规划如下：

- ❑ 管理员：管理员导航包括【管理员名称】选项、【管理员注销】选项。
- ❑ 用户：用户导航包括【用户名称】选项、【用户索引导航】选项、【用户注销】选项。
- ❑ 未登录用户：未登录用户包括【登录】选项、【注册】选项和【管理员登录】选项。

在 **Web** 开发中，导航基本是每个页面都需要的，在每个页面手动的添加导航是非常不现实的。在应用程序编写过程中，可以使用动态页面进行数据的呈现并通过 **JS** 的方式进行调用。右击当前项目，选择【新建文件夹】选项，在项目根目录中创建【js】文件夹。在 **js** 文件夹中创建 **banner.aspx** 用于导航。删除 **banner.aspx** 文件中生成的代码而只保留声明代码，示例代码如下所示。

```
<%@ Page
Language="C#" AutoEventWireup="true" CodeBehind="banner.aspx.cs" Inherits="_28_1.js.banner" %>
```

上述代码是一段声明代码，开发人员能够在代码下使用 **javascript** 脚本的形式用于页面的数据的呈现，示例代码如下所示。



```
<%@ Page
Language="C#" AutoEventWireup="true" CodeBehind="banner.aspx.cs" Inherits="_28_1.js.banner" %>
<%
    if (Session["admin"] != null)
    {
%>
        document.write('
        你好:<% Response.Write(Session["admin"].ToString()); %>&nbsp;');
        document.write('
        <a href="../admin/logout.aspx">注销</a>&nbsp;');
    }
    else if (Session["username"] != null && Session["userid"] != null)
    {
%>
        document.write('
        你好:<a href="../personal.aspx?uid=<% Response.Write(Session["userid"].ToString()); %>">
        <% Response.Write(Session["username"].ToString()); %></a>&nbsp;');
        document.write('
        <a href="../userindex.aspx?uid=<% Response.Write(Session["userid"].ToString()); %>">
        我的留言</a>&nbsp;');
        document.write('
        <a href="../admin/logout.aspx">注销</a>&nbsp;');
    }
    else if (Session["username"] == null && Session["userid"] == null && Session["username"] == null)
    {
%>
        document.write('
        <a href="../login.aspx">登录</a>&nbsp;');
        document.write('
        <a href="../register.aspx">注册</a>&nbsp;');
        document.write('
        <a href="../admin/login.aspx">管理员登录</a>&nbsp;');
    }
%>
```

上述代码制作了一个用于呈现导航的 **js**，通过调用此 **js** 文件能够呈现不同的导航，示例代码如下所示。

```
<div class="gbook_banner">
    <script src="js/banner.aspx" type="text/javascript"></script>
</div>
```

上述代码为留言本页面中的导航页面，在该页面使用制作的 **js** 文件能够快速的呈现相应的导航并实现逻辑判断。当有多个页面需要使用该导航时，可以直接使用该 **js** 文件而不需要额外的在每个页面中进行逻辑判断和事务处理，这样就能够有效降低应用程序之间的耦合度。

**注意：**在编写 **js** 文件的过程中，需要以 **document.write** 的方式进行 **HTML** 标签以及数据的输出。任何

动态页面都能够作为 **js** 文件的进行编写和调用。

该导航实现了不同的用户权限的不同视图呈现，对于不同的用户而言，其导航也不应该相同。而针对不同的用户提示用户进行不同的操作是非常必要的，例如这里的用户进行【我的索引】选项的选择，就会跳转到该用户的索引页面而不需用户手动编写。

28.5.4 侧边栏界面优化

侧边栏也是导航的另一种形式，但是在留言本应用程序中，侧边栏用于不同的留言本之间的跳转。开发人员能够优化侧边栏界面以便用户能够更加方便的进行留言本的跳转和索引。现有的侧边栏界面如图 28-17 所示。



图 28-17 现有侧边栏界面

从现有的侧边栏界面可以看出，其界面是非常不友好的。由于这里使用了 **GridView** 控件，开发人员能够自动套用格式以便数据的呈现，如图 28-18 所示，开发人员还能够自定义控件以便数据的呈现，如图 28-19 所示。

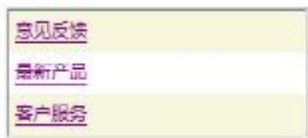


图 28-18 自动套用格式

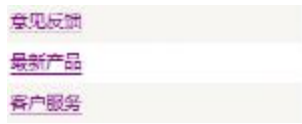


图 28-19 自定义格式

开发人员可以使用不同的格式以便侧边栏能够更加友好。从图 28-18 和图 28-19 中的侧边栏呈现形式可以看出，及时对侧边栏进行样式控制依旧不能对其中的超链接进行样式控制。因为超链接的样式控制需要在 **CSS** 文件中编写，示例代码如下所示。

```
a:link
{
    text-decoration: none;color: #000000;
}
a:active
{
    text-decoration: none;color: #000000;
}
a:visited
{
    text-decoration: none;color: #000000;
}
```

上述代码控制了超链接文本的样式。其中 **a:link** 的意义是链接的样式，**a:active** 是激活状态的超链接样式，而 **a:visited** 是已访问过的超链接文本样式。除了这几种超链接文本样式以外，还包括 **a:hover** 超链接样式，**a:hover** 是当鼠标移动到超链接文本上时，超链接文本的呈现样式。示例代码如下所示。

```
a:hover
{
    color:white;
    background:red;
}
```

上述代码则通过编写 **a:hover** 样式进行超链接文本样式的丰富。当超链接呈现在用户浏览器中时，其呈现方式首先以 **a:link** 样式呈现，如图 28-20 所示。当用户的鼠标放置在连接上时（并不是点击，而是放置），其样式会以 **a:hover** 的形式呈现，如图 28-21 所示。

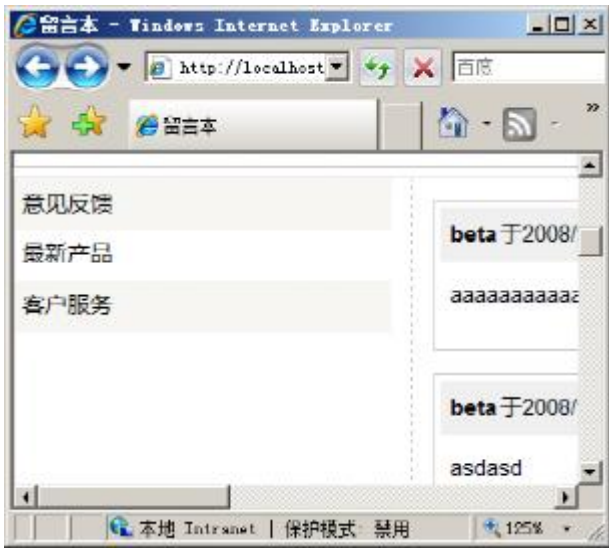


图 28-20 a:link 样式



图 28-21 a:hover 样式

丰富侧边栏样式以及编写超链接文本样式能够让页面样式更加简约，编写后的样式能够提高用户体验的友好度并加强应用程序的交互性。

## 28.6 用户功能实现

用户功能是 **Web** 应用中另一个非常重要的功能，该功能能够让用户方便的查看自己注册的信息并通过修改模块进行用户信息的修改以便呈现不同的用户信息。用户功能的实现主要是为了能够更好的管理网站的注册用户，对于注册并不使用的用户，管理员还能够进行用户的修改、删除等操作。

### 28.6.1 用户信息界面

在用户登录后，用户能够通过不同的导航栏进行相应的操作。在 **Web** 应用中，用户可以索引自己的用户信息，另外用户还能够修改自己的信息。为了方便用户对自己信息的查看，这里可以制作用户信息界面。在该界面，用户不仅能够查看自己的用户信息，还能够查看自己留言，以及回复信息的统计。示例代码见光盘中源代码\第 28 章\28-1\28-1\personal.aspx。

其中，页面代码实现了用户信息的呈现。在该代码中，使用了多个 **Label** 标签控件以便呈现用户信息。在呈现用户信息时，页面的其他信息同样需要呈现，页面布局如图 28-22 所示。



图 28-22 个人信息页面布局

注意：在进行页面布局时，其样式可以直接使用个人留言索引页面的布局和样式控制。值得注意的是，

对于多个不同功能却需要使用相同的布局样式的页面而言，可以通过 CSS 进行整体的样式控制。

为了能够方便的让用户查看统计信息，开发人员能够通过 SQL 语句进行数据库中信息的查询进行数据的呈现和统计。在用户信息页面，用户能够查看信息、查看统计和修改信息，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e) //页面加载执行
{
    if (!String.IsNullOrEmpty(Request.QueryString["uid"])) //获取传递的 uid
    {
        string uid = Request.QueryString["uid"]; //参数值获取
        string strsql = "select * from register where id='"+uid+"'"; //编写查询语句
        SqlDataReader da=
        SQLHelper.SQLHelper.ExecReader(strsql); //初始化适配器
        while (da.Read()) //查询数据
        {
            Label1.Text = da["username"].ToString(); //显示用户名
            if (da["sex"].ToString() == "1") //判断用户性别
                Label2.Text = "男"; //显示性别
            else if (da["sex"].ToString() == "2") //判断用户性别
                Label2.Text = "女"; //显示性别
            else
                Label2.Text = "保密"; //显示默认
            Label3.Text = da["im"].ToString(); //输出 im 值
            Label4.Text = da["information"].ToString(); //输出用户信息
            Label5.Text = da["others"].ToString(); //输出备注

            string strsql1 = "select count(*) as mycount from gbook where name='"+Label1.Text+"'";
            SqlDataReader da1 =
            SQLHelper.SQLHelper.ExecReader(strsql1); //查询统计信息
            while (da1.Read())
            {
                Label6.Text = da1["mycount"].ToString(); //输出统计信息
            }
        }
    }
}
```

上述代码在页面被加载时执行。当页面加载时，该代码段会获取传递的用户 **id** 的信息进行数据查询。在查询完成后，该代码将查询的数据值填充到控件中以便能够呈现在客户端浏览器中。在查询了用户数据后，还需要通过用户的用户名进行数据库中留言数据的统计，以便用户能够快速查看自己的留言统计信息。

## 28.6.2 用户修改实现

当用户进行信息的查看后，如果需要修改自己的信息，则可以通过用户修改页面进行修改功能的实现。为了方便用户进行信息的修改，用户可以在个人信息页面进行用户信息的修改。另外，为了保证用户信息的安全性，在用户查看信息时，如果用户查看的信息是自己的信息，则允许用户对信息的修改，否则该用户不是可被允许的用户，则该用户不能够进行信息的修改。判断用户身份代码如下所示。

```
if (Session["username"] != null && Session["userid"] != null) //判断是否登录
{
    if (Session["username"].ToString() == Label1.Text) //如果登录并且是当前用户
    {
        Label7.Text = "<a href='\"modi.aspx?uid=\" + Request.QueryString[\"uid\"] + \"'>";
        修改资料</a>"; //修改资料
    }
}
```



```
    }
    else //如果登录但不是当前用户
    {
        Label7.Text = "<a href=\"personal.aspx?uid=\" + Session[\"userid\"].ToString() + \"\">
        我的信息</a>"; //可以选择跳转到“我的信息”
    }
}
```

上述代码在用户信息界面实现并在页面加载时执行。用户通常不允许直接跳转到用户修改页面，用户能够在自己的用户信息页面选择是否修改自己的信息。当一个用户登录后，该用户就具备了查看其他用户信息的权限，当用户查看其他用户信息时，该用户不能够修改其他用户的信息，如果当前用户访问的是自己的信息页面时，这个而用户就被认为是“可操作的”用户，即当前用户能够修改用户信息。

用户能够单击【修改资料】超链接跳转到修改资料页面中，在修改资料页面加载时，还需要进一步的进行用户权限的判断。如果当前用户是“可操作的”用户（包括管理员），则当前用户能够继续进行操作，示例代码如下所示。

```
    if (Session[\"username\"] != null && Session[\"userid\"] != null) //判断是否登录
    {
        if (Session[\"username\"].ToString() != Label1.Text) //判断是否是当前用户
        {
            Response.Redirect(\"default.aspx\"); //不是用户则跳转
        }
        else
        {
            Response.Redirect(\"default.aspx\"); //未登录用户跳转
        }
    }
}
```

上述代码在执行加载时同样再次判断用户权限，如果用户权限正确，则允许从页面的呈现并允许用户进行数据更新操作，示例代码如下所示。

```
    string uid = Request.QueryString[\"uid\"]; //获取传递的参数
    string strsql = \"select * from register where id=\" + uid + \"\"; //编写查询语句
    SqlDataReader da =
    SQLHelper.SQLHelper.ExecReader(strsql); //创建和初始化适配器
    while (da.Read()) //读取数据
    {
        Label1.Text = da[\"username\"].ToString(); //进行数据填充
        Label2.Text = da[\"password\"].ToString(); //填充密码
        DropDownList1.Text = da[\"sex\"].ToString(); //填充性别
        TextBox4.Text = da[\"im\"].ToString(); //填充 im
        TextBox5.Text = da[\"information\"].ToString(); //填充用户信息
        TextBox6.Text = da[\"others\"].ToString(); //填充用户备注
    }
}
```

当用户访问自己的页面时，用户可以进行数据的查看和更新，如图 28-23 所示，当用户单击【提交】按钮控件时，用户能够执行数据操作。

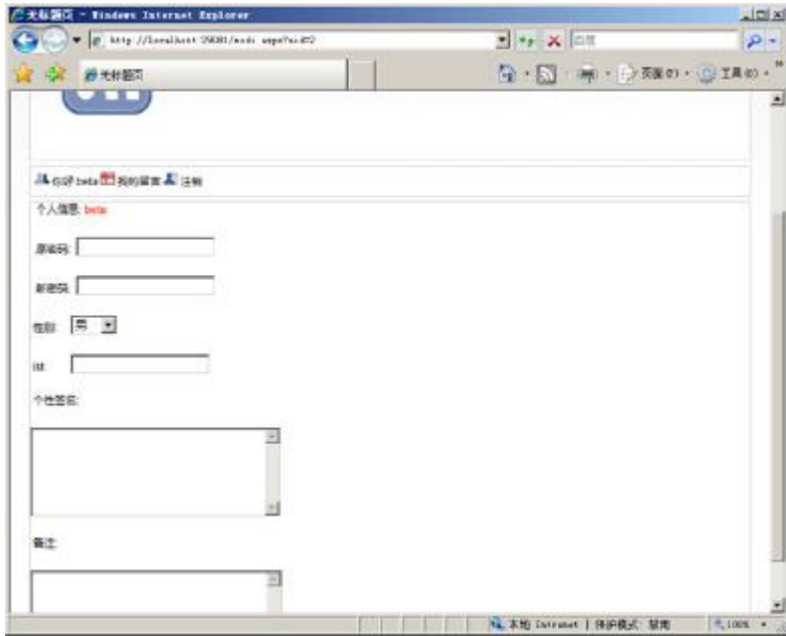


图 28-23 查看基本信息

正如图 28-23 所示，当用户进行注册时，往往不会填写注册的信息，如果用户希望在今后的操作中进行信息的更正，用户可以填写或清空相应的信息。值得注意的是，如果用户需要修改自己的信息，首先需要填写原密码进行身份的验证，如果填写的原密码信息等于数据库中的密码，则说明该用户能够进行修改操作，示例代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    if (TextBox1.Text != Label2.Text)                                     //判断密码是否正确
    {
        Label3.Text = "原密码不正确,无法修改!";                       //不正确则停止修改
    }
    else
    {
        if (!String.IsNullOrEmpty(TextBox2.Text))                     //判断密码输入框
        {
            string strsql = "update register set password='" + TextBox2.Text + "',sex='" +
                DropDownList1.Text + "',im='" + TextBox4.Text + "',information='" + TextBox5.Text +
                "',others='" + TextBox6.Text + "' where id='" + Request.QueryString["uid"] + "'";
            SQLHelper.SQLHelper.ExecNonQuery(strsql);                   //如果输入密码则更新
            Response.Redirect("personal.aspx?uid="+Request.QueryString["uid"]); //页面跳转
        }
        else
        {
            string strsql = "update register set sex='" + DropDownList1.Text + "',im='" + TextBox4.Text
                + "',information='" + TextBox5.Text + "',others='" + TextBox6.Text + "' where id='" +
                Request.QueryString["uid"] + "'";                       //未输入密码则不更新
            SQLHelper.SQLHelper.ExecNonQuery(strsql);                   //执行 SQL 语句
            Response.Redirect("personal.aspx?uid=" + Request.QueryString["uid"]);
        }
    }
}
```

在用户信息更改页面中，如果用户填写了【新密码】文本框，则在执行更新时会更改用户的密码。如果用户在更新过程中没有填写【新密码】文本框，则系统不会更改用户的密码，当用户再次登录时，依旧可以使用原密码进行信息查询和留言。

**注意：**在执行数据库更新操作时，页面需要使用 **IsPostBack** 属性进行判断，否则可能不会更新数据。

### 28.6.3 用户信息删除实现

用户能够查看自己的信息并修改自己的信息，但是用户无法对自己的信息进行删除。在留言本系统中，只有管理员能够对用户的信息进行删除。值得注意的是，为了保证用户信息的完整度，在删除用户信息时，还需要进行用户相关数据的删除。

在用户信息查看页面，管理员能够进行用户信息的删除。在用户信息查看页面被加载时，需要进行管理员权限的判断，如果一个用户的权限是管理员，那么在页面加载时就应该加载【删除用户】连接以便管理员进行删除操作，示例代码如下所示。

```
if (Session["admin"] != null)
{
    Label8.Text = "<a href=\"delete.aspx?uid="+Request.QueryString["uid"]+"\">删除用户</a>";
    //呈现连接
}
```

当管理员进行用户信息页面访问时，由于该用户是一个管理员，则该用户能够执行删除操作。删除操作在“**delete.aspx**”页面实现，示例代码如下所示。

```
namespace _28_1
{
    public partial class delete : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)           //页面加载事件
        {
            string strsql = "delete from register where id='"+Request.QueryString["uid"]+"'"; //删除用户信息
            string strsql1 = "delete from gbook where userid='"+Request.QueryString["uid"]+"'"; //删除留言
            SQLHelper.SQLHelper.ExecNonQuery(strsql1);                    //执行留言删除
            SQLHelper.SQLHelper.ExecNonQuery(strsql);                    //执行信息删除
            Response.Redirect("default.aspx");                            //页面跳转
        }
    }
}
```

当执行上述代码时，系统将删除用户的信息，以及和用户相关的留言信息以保证数据库系统的完整性。当删除用户信息后，用户的所有信息都会从数据库中删除，当该用户访问 **Web** 应用时，无法再使用原来的账号进行登录。

**注意：**在某些系统设计中，当需要进行数据库的删除时，需要判断数据库中数据的约束性和完整性。

当进行某些数据的删除时，首先需要考虑数据的完整性，然后再执行删除操作。

### 28.6.4 用户注销

当用户登录完毕后并执行了相应的操作后，用户可以选择注销操作进行注销以保证用户信息的安全。在主要页面中，不仅是普通用户能够执行注销操作，管理员也同样能够执行注销操作。当执行了注销操作后，用户的所有信息将会被清除，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    Session["username"] = null;           //清除用户信息
    Session["userid"] = null;             //清除用户信息
    Session["admin"] = null;              //清除管理员信息
    Response.Redirect("../default.aspx"); //页面跳转
}
```

用户注销的过程十分简单，从上述代码即可看出。当用户执行注销操作时，系统只需要将相应的 **Session**

对象的值赋值为 **null** 即可。该操作将删除用户的信息，当系统的某些页面被加载时，会判断用户未登录。

28.7 实例演示

在编码完成后还需要对现有的项目进行测试，在内部进行演示是一个很好的方法，在进行实例演示之前，需要准备一些数据，这些数据能够提供基本的数据操作所必备的元素从而能够为后面的应用程序运行提供基本保障。

28.7.1 准备数据源

由于篇幅的限制，**ASP.NET** 留言本中没有针对管理员和聊天室分类进行数据添加、数据操作等页面的制作，这些页面的制作在前面的模块中已经有所涉及，所以就不再重复讲解。在进行演示前，需要执行相应的数据操作以保证基本的数据是存在的。在执行操作前，需要添加管理员，管理员的添加可以在 **SQL Server Management Studio** 视图状态下添加也可以执行 **SQL** 语句进行添加。执行 **SQL** 语句添加管理员代码如下所示。

```
INSERT INTO admin (admin,password) VALUES ('admin','admin')
```

上述代码创建了一个名字为 **admin**，密码为 **admin** 的管理员，使用该用户名和密码能够进行页面操作。除了需要准备管理员数据以外，还需要准备留言分类数据，其 **SQL** 语句如下所示。

```
INSERT INTO gbook_class (classname) VALUES ('客户服务')
INSERT INTO gbook_class (classname) VALUES ('最新产品')
INSERT INTO gbook_class (classname) VALUES ('意见反馈')
```

上述代码使用 **SQL** 语句进行留言本分类数据的插入，当用户浏览不同的留言本时所看到的留言也不相同。

28.7.2 基本实例演示

当用户进行网站的访问时，可以选择相应的留言本分类进行留言本的访问，不同的分类其留言也不相同，如图 **28-24** 所示。

在用户选择了相应的留言本分类后，单击 **【Go!】** 按钮控件就能够访问相应的留言本，跳转到相应的页面，如图 **28-25** 所示。



图 28-24 选择留言本



图 28-25 留言本页面

用户能够在左侧的导航栏中选择感兴趣的留言本进行留言查看，也可以查看当前选择的留言。正如图 **28-25** 所示，用户并没有进行登录操作，所以无法看到留言本中的留言控件进行留言操作，未登录的用户可以看见留言本中的留言并进行翻页。如果用户希望在留言本中进行留言，就需要进行登录操作，如果在登



录操作前没有注册，则还需要进行注册，这里注册一个用户名和密码都为 **beta** 的用户，注册页面如图 28-26 所示。

单击【立即注册】按钮就能够进行注册操作，用户注册完成后依旧不能够进行留言操作，只有进行了登录操作的用户才能够进行留言，打开 **login.aspx** 页面进行登录操作，如图 28-27 所示。

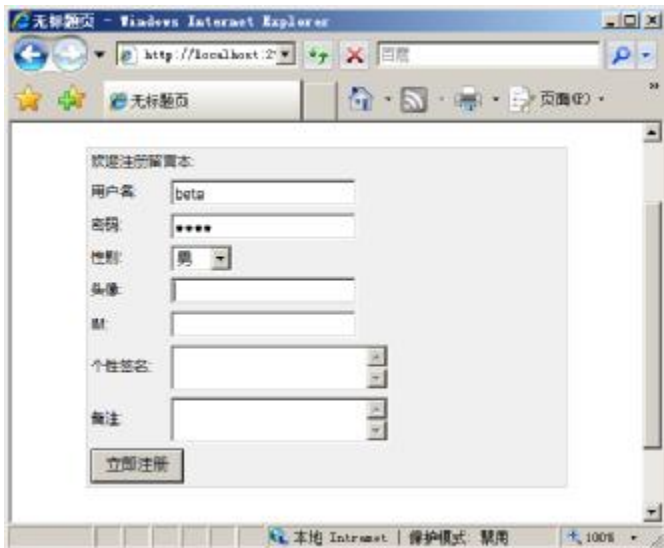


图 28-26 注册页面

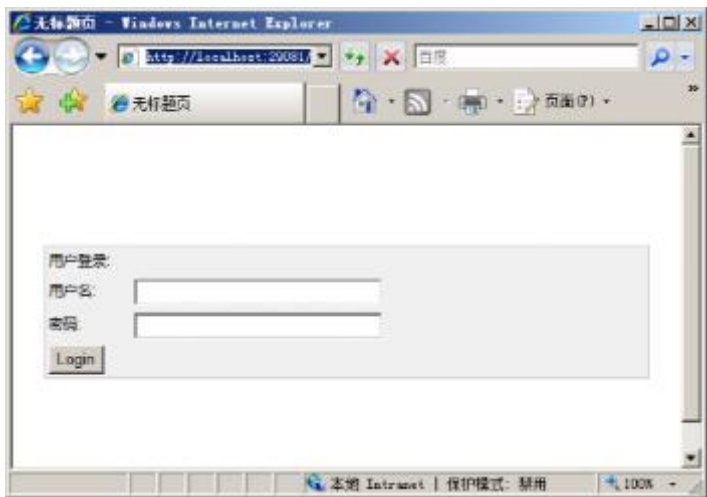


图 28-27 登录操作

登录完成后，系统会给登录的用户一个 **Session** 对象，以便进行留言。登录后再次选择留言本就能够进行留言了，如图 28-28 所示。

从图 28-28 中可以看出，登录后的用户可以进行留言，留言后会根据留言的 **ID** 进行倒序，以确保最新的留言数据在最前端。如果留言本中的留言需要管理员进行回复，管理员可以单击【回复】按钮进行回复操作。在进行回复操作之前，管理员需要进行登录，登录页面在 **admin/login.aspx** 中，可以使用 **admin/admin** 进行登录，登录完成后可以选择相应的聊天室进行回复，如图 28-29 所示。



图 28-28 执行留言操作

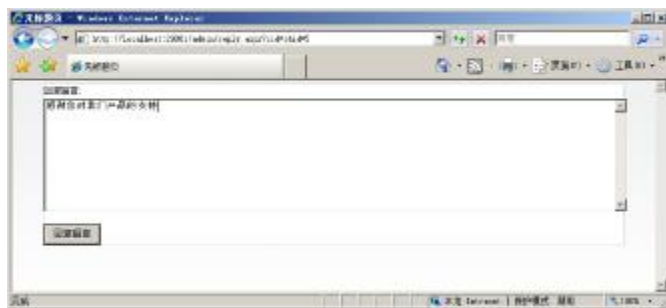


图 28-29 管理员回复

管理员发表回复后系统会跳转到相应的留言分类的页面，如图 28-30 所示。



图 28-30 管理员回复

管理员同样可以在留言页面进行留言的删除操作。当执行删除操作后，系统会跳转到 **delete.aspx** 页面并通过参数进行删除操作的执行，当执行完毕后会跳转回留言页面，而留言页面的数据会被同步更新。在留言本页面中，可以选择【导航】选项进行不同留言本中留言的查看和管理，用户也能够通过导航在不同的分类的留言本内进行留言。

28.7.3 用户功能演示

上一小节演示了基本的用户注册、登录以及留言等操作，在这一节中将演示用户功能。当用户登录后，用户就可以选择相应的留言模块进行留言操作，如图 28-31 所示。

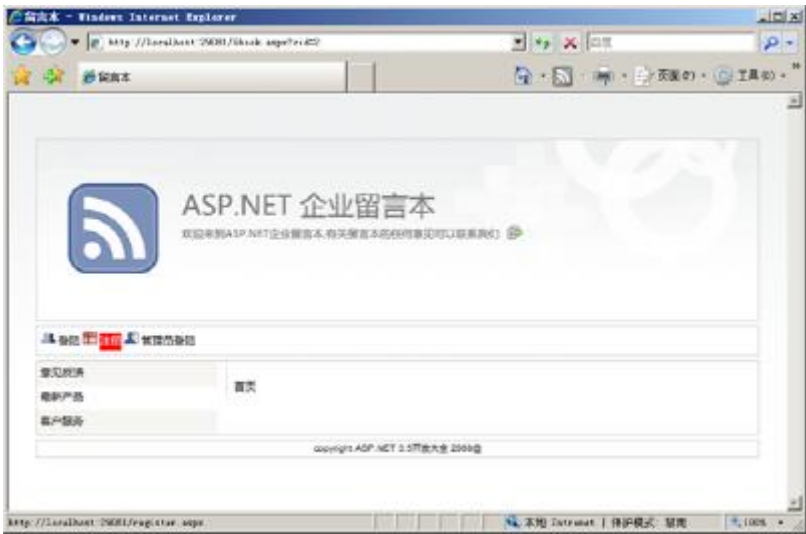


图 28-31 进行登录、注册选择

当用户选择登录并登录后，用户可以在导航栏中进行相应的用户操作，如图 28-32 所示。



图 28-32 登录后进行导航选择

当用户登录完成后，就能够在导航中进行用户信息的选择和留言的索引。单击【用户名】超链接能够进入用户信息页面。在用户信息页面，用户能够查看自己的用户信息，如图 28-33 所示。不仅如此，用户还能够通过相应的 id 信息访问其他用户的用户信息，如图 28-34 所示。



图 28-33 查看当前用户信息



图 28-34 查看其他用户信息

正如图 28-33 和图 28-24 所示，当用户访问自己的信息时，其页面呈现的就是【修改资料】字样，当用户访问其他的用户信息时，则呈现【我的信息】字样。单击【修改资料】连接即可修改用户资料，如图 28-35 所示。

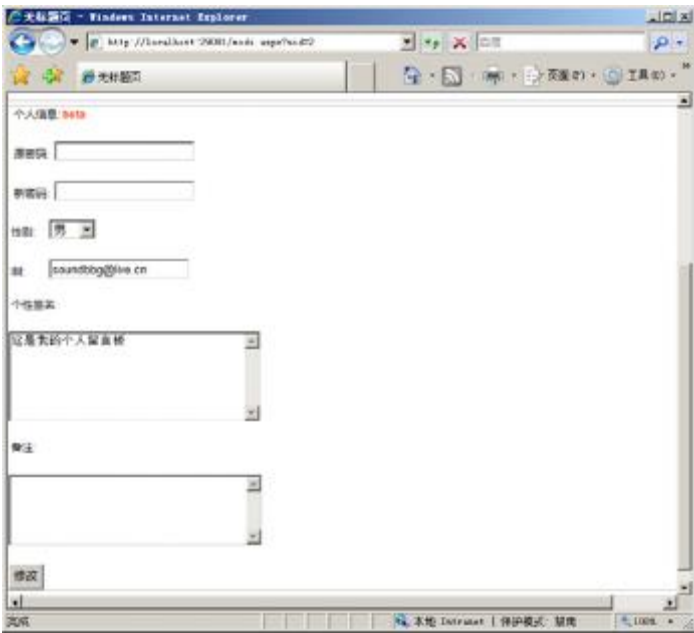


图 28-35 修改用户信息

用户可以在文本框控件中编写相应的信息进行数据的更新。在更新数据前，用户还需要填写【原密码】以便能够执行用户信息的更新验证。当用户填写了正确的原密码后，系统将允许用户执行更新操作，否则系统将提示用户错误信息并重新进行填写，如图 28-36 所示。

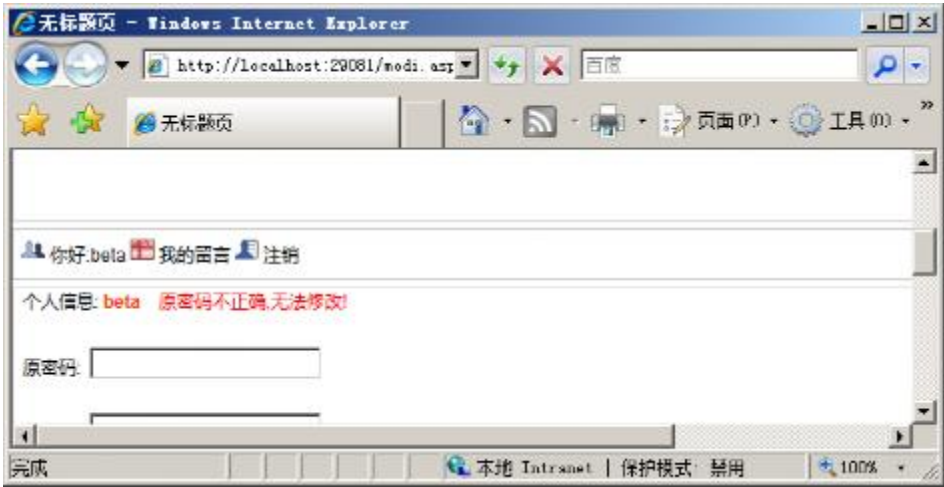


图 28-36 错误的修改密码

当用户修改密码后，用户必须使用新的密码进行登录，原密码将被更新。如果一个用户长时间没有进行更新或者留言等出道作，管理员能够将用户进行删除。管理员或用户可以单击【管理员登录】超链接进行管理员登录操作，登录后，管理员能够打开用户界面进行用户信息的删除，如图 28-37 所示。

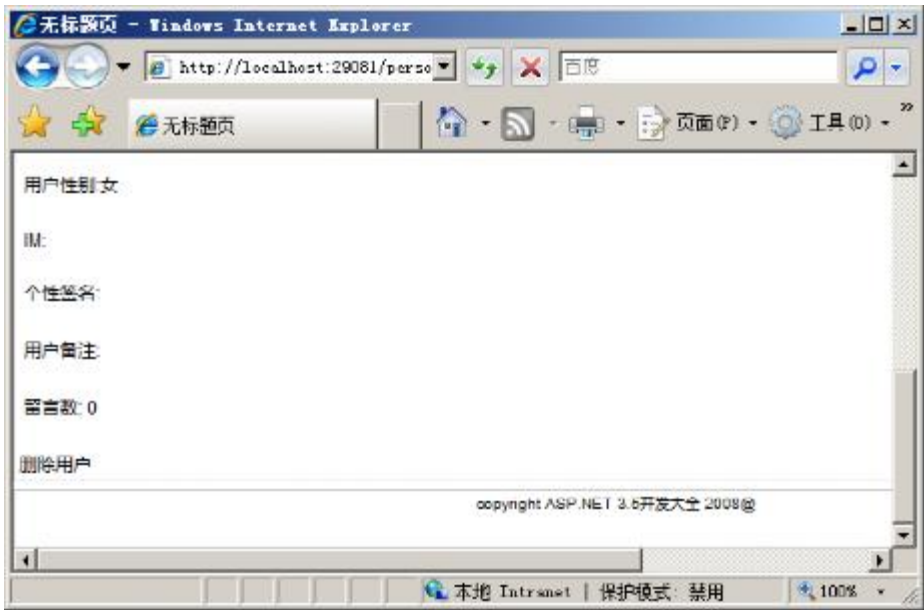




图 28-37 删除用户

在删除用户时，必须以管理员身份进行登录，如果不以管理员身份登录，则无法呈现【删除用户】超链接。当管理员单击【删除用户】超链接时，系统会跳转到删除页面并执行删除操作，在删除过程中，删除功能会删除用户的留言和用户的信息。在删除用户前，留言本的留言如图 28-38 所示。在执行了删除操作后，留言本的留言如图 28-39 所示。

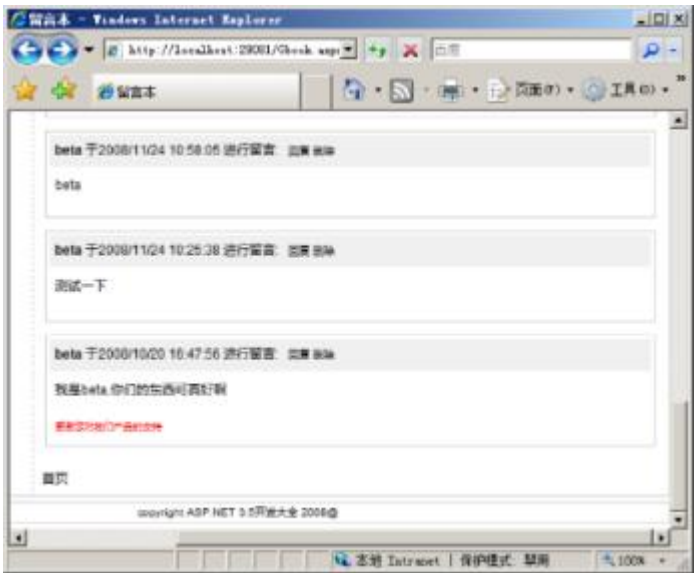


图 28-38 删除用户前



图 28-39 用户删除后

在执行删除后，用户的评论也会随之删除，这样不仅保证了数据的完整性，另外也保证了数据库的约束性。

注意：在执行信息删除操作时，首先执行的是约束条件的删除。例如这里首先必须删除用户的留言信息，当留言信息被删除完毕后，才能够进行用户信息的删除。如果不删除留言信息而进行用户信息的删除，则在数据库设计时的约束条件会被破坏，系统会抛出异常。

## 28.8 小结

本章通过留言本实例的讲解讲解了软件开发过程中的基础步骤，这些步骤包括系统设计、需求分析文档、数据库设计等等。软件项目管理在软件开发过程中是非常重要的，良好的软件项目管理能力能极大的简化和协调开发人员对产品的开发。

本章还使用了 **SQLHelper** 类进行数据操作，**SQLHelper** 类简化了开发人员对数据的操作，只需要使用一行或几行代码就能够实现 **ADO.NET** 中的复杂代码实现，**SQLHelper** 类还能够不同的项目中使用，这些项目包括 **Web** 应用、类库和自定义控件。本章还包括：

- ❑ 功能模块划分：讲解了 **ASP.NET** 留言本中的功能模块的划分。
- ❑ 使用 **SQLHelper**：讲解了如何使用 **SQLHelper** 类进行数据操作。
- ❑ 配置 **Web.config**：讲解了如何使用 **Web.config** 配置文件配置可扩展元素。
- ❑ 留言板用户控件：讲解了如何开发留言板的用户控件。
- ❑ 留言功能实现：讲解了如何实现留言本需要的一系列操作。

留言本是系统开发中比较基础的系统，但是很多系统模块都是基于留言本的制作流程的，例如论坛、小型社区、博客等都是类似于留言本的。这些模块在很久之前都是从留言本系统中演化而来的，对于掌握的较熟练的读者可以将本章中的留言本系统结合前面章节中的新闻模块进行博客系统的开发。



## 第 29 章 制作一个 ASP.NET 校友录系统

在现在的网络应用中，用户是网络应用的中心，如现今最风靡的校内网都是把用户放到了网络应用的第一位。而校内网的成功和风靡在很大程度上是因为它是一个真实的社交网络，校友录系统也是利用了真实的社交网络进行设计和开发的。

### 29.1 系统设计

在编写校友录系统前，首先需要确定校友录系统所需要的一些功能模块和适用场景，例如校友录是以何种形式呈现给用户的，如何判断这个用户是不是一个真实的用户等等，这些功能都是需要在开发初级进行设计和规划的。

#### 29.1.1 需求分析

在上一章 **ASP.NET** 留言本中，通过一个简单的 **ASP.NET** 留言本项目对需求分析进行介绍，需求分析是在系统设计中一个最为重要的组成部分，良好的需求分析设计能够极大的方便在后续过程中的软件开发以及软件维护。

##### 1. 目录

需求分析通常情况下是一个单独的需求分析文档，为了模拟在软件开发过程中的顺序，以及软件开发的步骤，这里模拟基本的需求分析文档并为相关的部分进行描述。

- 1. 引言：通常是需求分析文档的引言，用户描述为何编写需求分析文档。
- 1.1 编写目的：编写目的用户描述为何编写需求分析文档。
- 1.2 项目背景：编写相应的项目背景。
- 1.3 定义缩写词和符号：编写在需求分析文档中定义的缩写词或符号等。
- 1.4 参考资料：用户描述在需求分析文档中所参考的资料。
- 2. 任务描述：定义任务，通常情况下用于描述完成何种任务。
- 2.1 开发目标：定义开发目标，包括为何要进行开发。
- 2.2 应用目标：定义应用目标，包括系统应用人员要实现什么功能，以及有哪些应用等。
- 2.3 软件环境：用于定义软件运行的环境。
- 3. 数据描述：用户进行数据库中数据设计开发的描述。

对于需求分析文档而言，其格式很像论文或软件开发说明书，所以在需求分析文档前通常会有一个目录方便客户和开发人员进行文档的阅读。上述目录描述了引言、编写文档的目录、项目背景等，当客户进行文档的翻阅时可以很方便的进行文档的查询。

##### 2. 引言

对于 **ASP.NET** 校友录系统而言，其作用是为了增加同学之间的友情，在需求分析文档的引言部分可以简单的编写为何要开发该系统以及相应的背景。引言编写如下所示：

随着互联网的发展，越来越多的交流社区应用被广泛的接受，这些社区的存在都是为了能够加强人与人之间的交流。在针对现有的系统进行调查，拟开发一套校友录系统进行校友联络，这样不仅方便校友之间的联络，也能够加强老校友和新校友的感情。

此规格说明书在详细的调查了客户现有的应用模块和基本的操作流程后进行编写，对校友录系统以及其功能进行了详细的规划、设计，明确了软件开发中应具有的功能、性能使得系统的开发人员和维护人员能够详细清楚的了解软件是如何开发和进行维护的，并在此基础上进一步提出概要设计说明书和完成后续设计与开发工作。本规格说明书的预期读者包括客户、业务或需求分析人员、测试人员、用户文档编写者、项目管理人员等。

### 3. 项目背景

由于互联网的迅猛发展，越来越多的用户希望在互联网上能够即时的，快速的与家人或朋友进行联络，相对于传统的 C/S（客户端/服务器）模式的软件开发而言，其成本较高、难以维护，虽然能够即时的与家人和朋友发送消息，但是无法与家人和朋友分享生活和照片等。

而由于互联网的发展，越来越多的用户已经能够适应基于浏览器的应用程序，即 **Web** 应用，也有越来越多的用户尝试在 **Web** 服务上进行自己的应用，包括 **QQ** 空间、博客、个人日志等，都是基于浏览器的应用程序。

为了解决 C/S 模式的应用程序中日志、照片、音乐等难以交互的情况，现开发 **ASP.NET** 校友录系统用于进行校友之间的交流和通信，方便校友与校友之间进行通信。校友与校友之间不仅能够分享日志，还能够进行身边信息的分享，这样就加强了人与人之间的交互。

### 4. 任务描述

任务描述用于描述客户的任务，以及基本的讲述如何完成任务的描述，**ASP.NET** 校友录系统的任务描述可以编写为如下所示：

为了解决传统的 C/S 应用程序中程序的信息交互不够的问题，并加强用户与用户之间的信息交互，现开发基于 **.NET** 平台的校友录应用程序，用户能够使用校友录进行信息的通信和分享，不仅能够加强校友与校友之间的感情，也能够增强现有的社交。

### 5. 开发目标

**ASP.NET** 校友录系统的开发目标是为了加强现有的用户和用户之间的信息交互，解决传统的用户和用户沟通不便和沟通内容不够丰富的问题，进行用户和用户之间的数据整合和交互。

开发 **ASP.NET** 校友录系统可以为现有学校所使用，也可以被班级或个人进行使用，适用性广泛，不仅能够在大型应用中使用，同样也能够适用于小型应用。

### 6. 应用目标

**ASP.NET** 校友录是为了能够让校友之间进行真实的交互，用于加强校友与校友之间的感情，同时也能够收集校友的信息。

## 29.1.2 系统功能设计

**ASP.NET** 校友录是学校内的一个交流平台，用于校友与校友之间的信息交互，校友能够在校友录系统进行注册，注册完毕后管理员审核相应的用户并进行相应的用户操作，当用户的审核通过后，用户就能够在校友录中进行新鲜事的分享。在 **ASP.NET** 校友录系统的开发过程中需要确定基本的系统功能，这些基本的系统功能包括如下：

#### 1. 用户注册功能

当用户访问 **Web** 页面时需要进行注册，如果用户不进行注册就不能够发表和回复留言，也不能够分享相应的信息。管理员可以配置是否需要进行登录才能够查看校友录的内容，如果管理员设置需要登录查看，则用户不登录就不能够查看相应的内容。

#### 2. 用户登录功能

用户注册之后就需要实现用户的登录，登录的用户可以进行信息的发表、回复以及相应内容的分享。登录的用户的操作也会被记录在日志中，用户可以通过自己的 **ID** 进行校友录中的功能或文章的索引。

#### 3. 用户日志功能

用户注册和登录后就能够在校友录中进行日志分享，发表关于自己觉得的最新事件，其他人能够查阅该日志并进行相应的日志操作。

### 4. 用户留言功能

用户可以查看校友录中日志并进行相应的评论，不仅如此，用户还能够在回复中发布表情，进行文字处理等操作让留言功能更加丰富，用户还能够在校友录系统中对校友录的日志进行评分。

### 5. 管理员审核功能

当用户注册后，需要对用户进行身份的审核，管理员可以审核已知的用户的身份，如果用户不是校友录系统的指定用户，则管理员可以不允许用户进行身份验证和登录，以确保校友录系统中的用户的身份都是真实的。

### 6. 文章管理功能

管理员需要对校友发布的相应的信息进行管理，如果校友发布了反动、黄色、淫秽等文章，管理员有权进行修改、屏蔽和删除等操作。

### 7. 留言管理功能

管理员需要对校友发布的相应的留言进行管理，如果校友发布了反动、黄色、淫秽或广告的留言，管理员可以进行相应的留言的删除操作。

### 8. 用户管理功能

当用户进行了非法操作或者用户注册后发布了太多的反动、黄色、淫秽等内容，管理员可以将用户进行删除，在删除的同时系统数据库中的数据也会被删除。

### 9. 板报/公告等功能

管理员在校友录系统中还可以进行板报、公告等发布和管理，让页面看上去更像学生时代课堂的样子，这样提高了用户友好度也能够及时的将相应的信息反馈给校友，以便校友能够获取该校友录活动等最新消息。

## 29.1.3 模块功能划分

**ASP.NET** 校友录系统中的模块非常的多，这些模块包含最基本的注册、登录等模块，还包括文章管理、用户管理、用户管理等模块，这些模块都在不同程度上进行系统的协调。当介绍了系统所需实现的功能模块后并执行了相应的功能模块的划分和功能设计，可以编写相应的模块操作流程和绘制模块图，**ASP.NET** 校友录总体模块划分如图 29-1 所示。

图 29-1 描述了 **ASP.NET** 校友录系统的总体的模块划分，用户在校友录系统中需要进行注册登录等操作。对于用户而言，用户在 **ASP.NET** 校友录中必须要进行注册和登录操作，如果用户不进行登录操作就无法进行 **ASP.NET** 校友录中校友的信息的查看，**ASP.NET** 校友录中用户的模块流程图如图 29-2 所示。

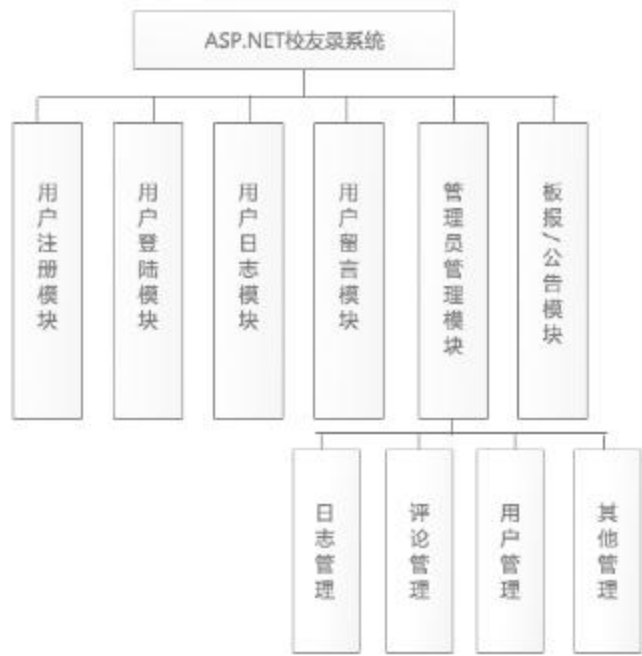


图 29-1 ASP.NET 校友录系统模块划分

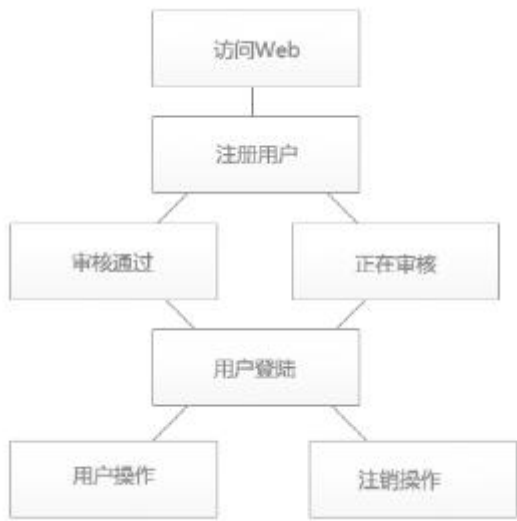


图 29-2 用户登录模块流程图

用户访问 **Web** 应用并能够在 **Web** 应用中进行注册，在用户注册后，并不能够立即进行相应的操作，如果用户没有被管理员审核，那么用户只能对校友录中的数据和信息进行查看，并不能进行修改等操作，如果管理员对用户进行了身份审核并通过相应的用户，则说明用户是一个可以被认为是真实的用户，那么用户就能够执行相应操作。

对于管理员而言，管理员不仅能够作为用户的一部分进行用户的活动，包括编写日志等，还应该具备管理功能，这些管理功能包括用户的审核、帖子的审核和用户的管理等等，管理员模块流程图如图 29-3 所示。

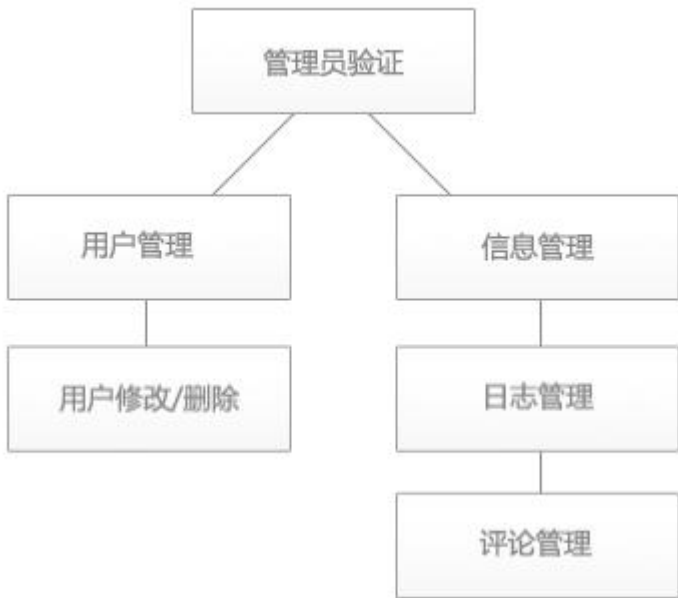


图 29-3 管理员模块流程图

正如图 29-3 所示，管理员在进行操作时同样需要对管理员进行身份验证，由于管理员也是用户的一部分，所以在进入后台管理时，需要判断用户是否有该权限进行管理，如果没有就不允许用户进行操作，如果有管理权限，管理员就能够在后台进行相应的管理操作。

对管理员进行身份验证后，管理员主要进行两大块管理，一个是用户管理，另一个是信息管理。对于用户管理而言，管理员主要是进行用户的删除、积分等操作，而对于信息管理而言，主要是用于不良的日志、评论进行修改和删除管理。

**注意：**由于管理员是用户的一部分，而一个校友录可以有多个管理员，这些管理员可以是用户，所以在数据库设计中需要额外的字段进行描述。



29.2 数据库设计

ASP.NET 校友录比 ASP.NET 留言本更加的复杂，在数据库设计上也更加复杂，不同的表之间还包含着连接。在这些数据表中，单个表或多个表都用来描述校友录的相应功能，在数据库设计中，还需要考虑到数据的约束和完整性约束以便数据库的维护。

29.2.1 数据库分析和设计

在前面的系统设计中已经非常仔细对功能和模块进行划分并对相应的用户（校友、管理员）进行了模块流程分析，在进行了模块划分和流程分析后就能够对数据库进行设计。从模块中可以看出 ASP.NET 校友录包含了更多的功能，这些功能都能够让校友用户在网站上分享自己的照片、音乐、视频等，所以在数据库的设计上，其表的数量和表与表之间的关系也比原有的模块或系统更加复杂。针对现有的模块以及模块流程图可以归纳数据库中相应的表，数据库设计图如图 29-4 所示。



图 29-4 数据库设计图

其中初步的为数据库中的表进行设计，这里包括四个表，分别作用如下：

- ❑ 用户注册表：用于存放用户的注册信息，以便登录时使用。
- ❑ 日志表：用户可以发布相应的日志，这些日志都存放在日志表中。
- ❑ 日志评论表：用户可以对相应的日志进行评论。
- ❑ 日志分类表：用户可以选择自己喜欢的分类进行日志发布，但日志分类由管理员管理。
- ❑ 公告信息表：管理员可以在校友录中发布最新的信息。

其中用户在发布日志时可以选择相应的分类，例如选择“最近心情”或“好歌欣赏”等，用户还能够进行相应的分类日志的索引。在 ASP.NET 校友录系统中最为重要的就是日志表和与之相关的表，用户在校友录系统中主要通过日志进行信息交换和分享。其中日志表的字段可以归纳如下。

- ❑ 日志 ID：日志的 ID，为自动增长的主键。
- ❑ 日志标题：日志的标题，用于显示日志标题的信息。
- ❑ 日志作者：日志的作者，用于显示是谁发布了日志。
- ❑ 日志发布时间：日志发布时间，用于显示日志发布的日期。
- ❑ 日志内容：日志内容，用于呈现日志的内容，包括音乐、图片等信息。
- ❑ 日志打分：日志打分，对于其他用户而言可以为该日志进行评分。
- ❑ 日志所属分类：日志所属分类，用于显示日志所属于的分类。
- ❑ 日志阅读次数：用于表示阅读被访问的次数。
- ❑ 日志所属用户 ID：日志所属用户 ID 用于标识该日志所属的用户信息。

日志表能够描述日志的基本信息，而日志分类表和日志所属用户表用户描述整个日志的其他信息，这些信息是日志的分类、日志发布作者的个性签名等等。日志分类表可以规划如下。

- ❑ 分类编号：用于标识留言本分类的编号，为自动增长的主键。
- ❑ 分类名称：用于描述分类的名称，例如“阳光男孩”等。

一个日志可以有一个分类进行描述，当对日志的分类进行描述后，用户可以通过索引相应的分类的日志，例如有某个用户对“阳光男孩”这个分类特别感兴趣，那么用户就能够索引这个分类的所有文章，而暂时关闭对其他文章的浏览。注册模块在前面的章节中都有设计，这里同样需要注册模块，注册模块的字段可以描述如下所。

- ❑ 用户名：用于保存用户的用户名，当用户登录时可以通过用户名验证。
- ❑ 密码：用于保存用户的密码，当用户使用登录时可以通过密码验证。
- ❑ 性别：用于保存用户的性别。
- ❑ 头像：用于保存用户的个性头像。
- ❑ QQ/MSN：用于保存用户的 QQ/MSN 等信息。
- ❑ 个性签名：用于展现用户的个性签名等资料。
- ❑ 备注：用于保存用户的备注信息。
- ❑ 用户情况：用于保存用户的状态，可以设置为通过审批和未通过等。
- ❑ 用户权限：用户区分是管理员还是普通用户。

与前面的用户注册不同的是，这里多了一个用户权限字段，由于管理员也能够进行普通的用户的操作，所以需要另一个字段进行用户权限的描述。当用户进行登录后，可以对相应的日志进行评论。同样，当管理员进行管理登录后，管理员可以对日志的评论进行删除，日志评论表字段如下所示。

- ❑ 评论 ID：用于标识评论，是自动增长的主键。
- ❑ 评论标题：用于表示评论的标题。
- ❑ 评论时间：用于表示评论的时间。
- ❑ 评论内容：用于表示评论的内容。
- ❑ 用户 ID：用于标识评论的用户 ID，可以通过该 ID 进行多表连接查询。
- ❑ 日志 ID：用于标识评论的所在的日志，可以通过该 ID 进行多表连接查询。

这些表就能够实现校友录的基本信息，在校友录首页就能够通过查询相应的数据进行校友录中的用户和数据的查看。

29.2.2 数据表的创建

创建表可以通过 SQL Server Management Studio 视图进行创建也可以通过 SQL Server Management Studio 查询使用 SQL 语句进行创建。

1. 事务表

在创建日志表之前首先需要创建 friends 数据库，创建完成后就能够进行其中的表的创建。在 ASP.NET 校友录系统中最为重要模块的就是日志模块，日志模块的表结构分别如图 29-5 和图 29-6 所示。

	列名	数据类型	允许空
	id	int	<input type="checkbox"/>
	title	nvarchar(500)	<input checked="" type="checkbox"/>
	author	nvarchar(50)	<input checked="" type="checkbox"/>
	time	datetime	<input checked="" type="checkbox"/>
	[content]	nvarchar(MAX)	<input checked="" type="checkbox"/>
	marks	int	<input checked="" type="checkbox"/>
	classid	int	<input checked="" type="checkbox"/>
	userid	int	<input checked="" type="checkbox"/>
	hits	int	<input checked="" type="checkbox"/>
			<input type="checkbox"/>

图 29-5 日志表结构

	列名	数据类型	允许空
	id	int	<input type="checkbox"/>
	classname	nvarchar(50)	<input checked="" type="checkbox"/>
			<input type="checkbox"/>

图 29-6 日志分类表结构

从数据库中可以看出留言表中的字段信息，日志表中的字段意义如下所示：

- ❑ **id:** 日志的 **ID**，为自动增长的主键。
- ❑ **title:** 日志的标题，用于显示日志标题的信息。
- ❑ **author:** 日志的作者，用于显示是谁发布了日志。
- ❑ **time:** 日志发布时间，用于显示日志发布的日期。
- ❑ **content:** 日志内容，用于呈现日志的内容，包括音乐、图片等信息。
- ❑ **marks:** 日志打分，对于其他用户而言可以为该日志进行评分。
- ❑ **classid:** 日志所属分类，用于显示日志所属于的分类。
- ❑ **hits:** 用于表示阅读被访问的次数。
- ❑ **userid:** 日志所属用户 **ID** 用于标识该日志所属的用户信息。

创建数据表的 **SQL** 查询语句代码如下所示。

```
USE [friends]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[diary](
    [id] [int] IDENTITY(1,1) NOT NULL,
    [title] [nvarchar](500) COLLATE Chinese_PRC_CI_AS NULL,
    [author] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
    [time] [datetime] NULL,
    [content] [nvarchar](max) COLLATE Chinese_PRC_CI_AS NULL,
    [marks] [int] NULL,
    [classid] [int] NULL,
    [userid] [int] NULL,
    [hits] [int] NULL,
    CONSTRAINT [PK_diary] PRIMARY KEY CLUSTERED
(
    [id] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
```

同样日志分类表的字段如下所示。

- ❑ **id:** 用于标识留言本分类的编号，为自动增长的主键。
- ❑ **classname:** 用于描述分类的名称，例如“阳光男孩”等。

创建数据表的 **SQL** 查询语句代码如下所示。

```
USE [friends]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[diaryclass](
    [id] [int] IDENTITY(1,1) NOT NULL,
    [classname] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
    CONSTRAINT [PK_diaryclass] PRIMARY KEY CLUSTERED
(
    [id] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
```

在用户发布日志后，用户可以对相应的日志进行评论，评论表和日志表的连接也是非常重要的，在相应的日志下需要筛选出相应的日志的评论进行呈现，用户也可以在相应的日志中添加自己的评论，评论表字段可以归纳如下。

- ❑ **id**: 用于标识评论，是自动增长的主键。
- ❑ **title**: 用于表示评论的标题。
- ❑ **time**: 用于表示评论的时间。
- ❑ **content**: 用于表示评论的内容。
- ❑ **userid**: 用于标识评论的用户 **ID**，可以通过该 **ID** 进行多表连接查询。
- ❑ **diaryid**: 用于标识评论的所在的日志，可以通过该 **ID** 进行多表连接查询。

创建数据表的 **SQL** 查询语句代码如下所示。

```
USE [friends]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[diarygbook](
    [id] [int] IDENTITY(1,1) NOT NULL,
    [title] [nvarchar](500) COLLATE Chinese_PRC_CI_AS NULL,
    [time] [datetime] NULL,
    [content] [nvarchar](max) COLLATE Chinese_PRC_CI_AS NULL,
    [userid] [int] NULL,
    [diaryid] [int] NULL,
    CONSTRAINT [PK_diary_gbook] PRIMARY KEY CLUSTERED
(
    [id] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
```

评论表需要同多个表进行连接，其中 **userid** 需要与注册表进行连接用于查询用户的信息，而 **diaryid** 用于同日志表进行连接查询所在的日志。在进行日志呈现时，同样需要连接日志评论表进行相应的评论的筛选。

## 2. 验证表

在用户注册中，增加了对管理员身份验证的字段，用户注册表字段如下所示。

- ❑ **id**: 用于标识用户 **ID**，为自动增长的主键。
- ❑ **username**: 用于保存用户的用户名，当用户登录时可以通过用户名验证。
- ❑ **password**: 用于保存用户的密码，当用户使用登录时可以通过密码验证。
- ❑ **sex**: 用于保存用户的性别。
- ❑ **pic**: 用于保存用户的个性头像。
- ❑ **IM**: 用于保存用户的 **QQ/MSN** 等信息。
- ❑ **information**: 用于展现用户的个性签名等资料。
- ❑ **others**: 用于保存用户的备注信息。
- ❑ **ifisuser**: 用于保存用户的状态，可以设置为通过审批和未通过等。
- ❑ **userroot**: 用于验证用户是管理员还是普通用户。

创建数据表的 **SQL** 查询语句代码如下所示。

```
USE [friends]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Register](
    [id] [int] IDENTITY(1,1) NOT NULL,
    [username] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
    [password] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
```



```
[sex] [int] NULL,
[picture] [nvarchar](max) COLLATE Chinese_PRC_CI_AS NULL,
[IM] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
[information] [nvarchar](max) COLLATE Chinese_PRC_CI_AS NULL,
[others] [nvarchar](max) COLLATE Chinese_PRC_CI_AS NULL,
[ifisuser] [int] NULL,
[userroot] [int] NULL,
CONSTRAINT [PK_Register] PRIMARY KEY CLUSTERED
(
    [id] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
```

上述代码创建了一个可以进行身份判断的用户表，开发人员可以通过 **userroot** 字段进行管理员身份的判  
断，其结构如图 29-7 所示。

	列名	数据类型	允许空
🔑	id	int	<input type="checkbox"/>
	username	nvarchar(50)	<input checked="" type="checkbox"/>
	password	nvarchar(50)	<input checked="" type="checkbox"/>
	sex	int	<input checked="" type="checkbox"/>
	picture	nvarchar(MAX)	<input checked="" type="checkbox"/>
	IM	nvarchar(50)	<input checked="" type="checkbox"/>
	information	nvarchar(MAX)	<input checked="" type="checkbox"/>
	others	nvarchar(MAX)	<input checked="" type="checkbox"/>
	ifisuser	int	<input checked="" type="checkbox"/>
	userroot	int	<input checked="" type="checkbox"/>
▶			<input type="checkbox"/>

图 29-7 注册表结构

3. 公告数据

公告数据可以不使用数据库进行存储，在这里可以使用 **txt** 文档进行数据存储，这样不仅可以减轻数据库服务器的压力，也能够增加公告中文本的可扩展性。

注意：对于公告的数据直接存储在 **txt** 文档中，当首页需要调用公告时，可以直接从 **txt** 文档中读取数  
据进行 **HTML** 呈现。

29.3 数据表关系图

系统数据库中需要进行约束，需要约束的表包括用户表、留言表和留言分类表，其约束可以使用 **SQL Server Management Studio** 视图进行编写。在 **ASP.NET** 同学录系统中，包括一些数据约束用于保持数据库中数据的完整性，数据表关系图如图 29-8 所示。

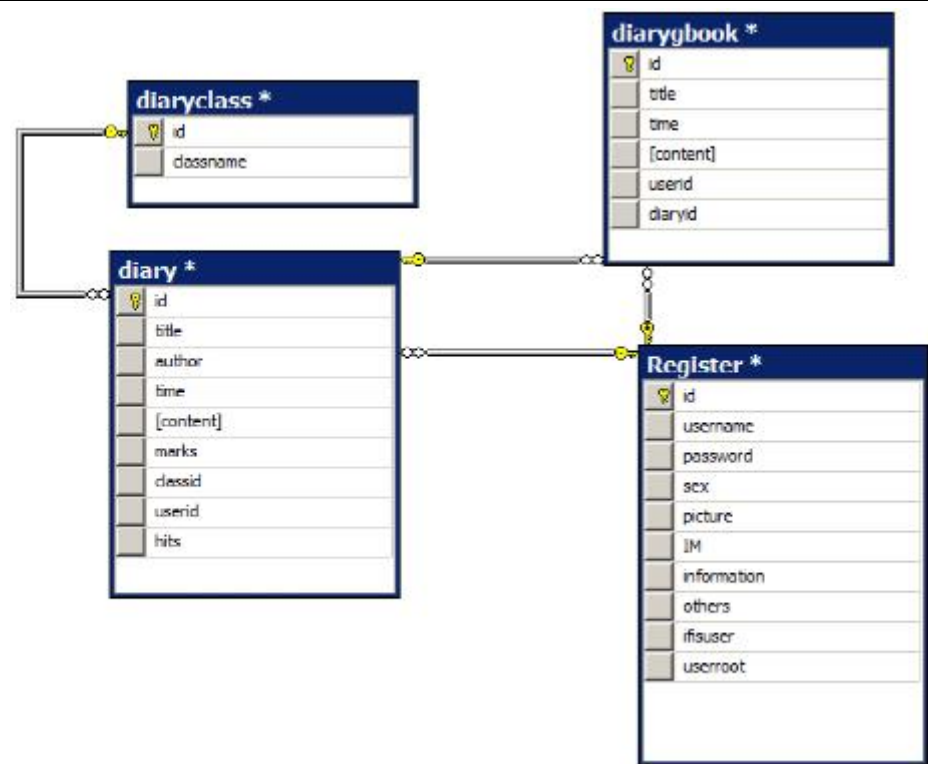


图 29-8 数据库关系图

图 29-8 中说明了数据库表之间的关系，进行表之间数据的约束。当进行数据插入时，就会判断数据库中表的约束情况和完整性，如果用户表没有任何数据而进行日志表中数据的插入是不被允许的。在进行关系的保存之后系统表就会被关系进行更改，其中的数据库创建的 SQL 语句也会相应的更改，就不会是原来的简单的创建语句，而更改后的语句包含了键值的约束关系。当对数据库中的表之间进行约束，则在进行数据操作时就应该按照规范来进行插入、删除等操作。

## 29.4 系统公用模块的创建

在 ASP.NET 校友录系统中，用户能够分享自己的日志就需要使用 HTML 编辑器，HTML 编辑器是系统的公用模块，可以通过 HTML 编辑器进行富文本编写和呈现。同样为了简化数据操作，也可以使用 SQLHelper 类进行数据操作。

### 29.4.1 使用 Fckeditor

Fckeditor 是现在最热门的开源 HTML 编辑器，使用 Fckeditor 能够像 Word 一样进行页面排版和布局，Fckeditor 还能够使用表情、进行拼写检查等。

在留言本系统开发中，并没有使用 Fckeditor 进行文本提交是因为在留言本系统中不需要进行复杂的文字呈现，而对于校友录系统而言，用户可以分享自己的日志并且进行日志布局和排版，这就需要进行复杂的 HTML 代码进行页面呈现。Fckeditor 能够完成这一系列复杂的操作。Fckeditor 在“附-Fckeditor 编辑器”中，可以在项目中使用 Fckeditor 进行文本框制作和二次开发。

在项目中添加 Fckeditor 的引用，首先需要将 Fckeditor 文件夹拷贝到项目中。由于 ASP.NET 应用程序的关系，Fckeditor 并不会在解决方案管理器中呈现，单击【解决方案管理器】上方的【显示所有文件】小图标可以进行所有文件的显示，如图 29-9 所示。

显示所有文件后就能够看到 Fckeditor 编辑器文件夹，右击 Fckeditor 文件夹，选择【添加到项目】选项就能够将文件夹中的所有文件批量添加到项目中，如图 29-10 所示。

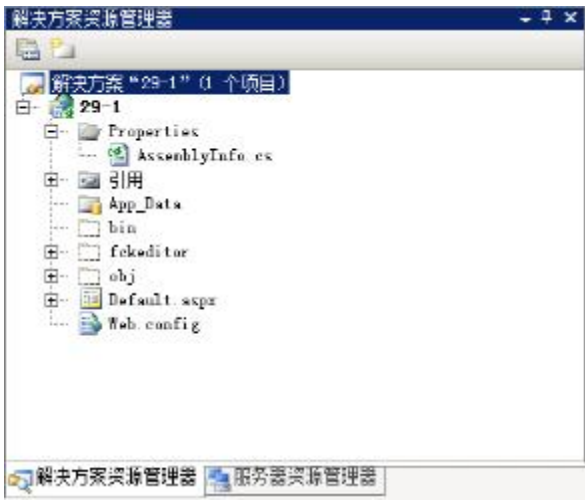


图 29-9 显示所有文件

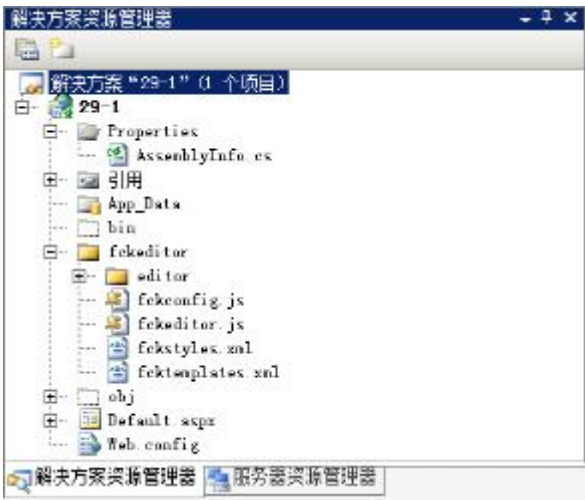


图 29-10 添加 Fckeditor

添加文件后还需要添加相应的 DLL 文件，以便在程序开发中使用 **Fckeditor** 编辑器进行文本框开发。右击现有项目，在下拉菜单中选择【添加现有项】选项，在标签中选择【浏览】选项卡，找到【附-Fckeditor 编辑器】目录中的 **bin** 目录并添加到项目中，如图 29-11 所示。

添加引用后就能够在开发中使用 **Fckeditor** 编辑器进行富文本编辑，开发人员还能够在工具栏中添加 **Fckeditor** 编辑器。单击【工具箱】的空白区域，单击右键，在下拉菜单中选择【选择项】选项，选择刚才添加的 **DLL** 文件。选择后单击【确定】按钮即可添加控件。控件添加完毕后就会在工具栏中呈现相应的控件，如图 29-12 所示。



图 29-11 添加引用

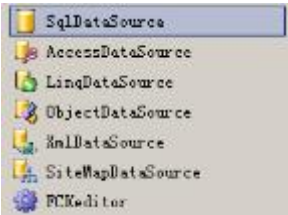


图 29-12 添加后的 Fckeditor 编辑器控件

开发人员能够将 **Fckeditor** 编辑器控件拖动到其他的区域，以适合自己的开发方式。在控件添加完成后，就可以向页面中添加控件，示例代码如下所示。

```
<body>
  <form id="form1" runat="server">
    <div>
      <FCKeditorV2:FCKeditor ID="FCKeditor1" runat="server">
      </FCKeditorV2:FCKeditor>
    </div>
  </form>
</body>
```

上述代码使用了 **Fckeditor** 编辑器控件进行富文本操作，运行后如图 29-13 所示。



图 29-13 Fckeditor 编辑器

使用 **Fckeditor** 编辑器可以更快的进行富文本的编辑，如果开发人员从头开发 **HTML** 编辑器会花费大量的时间，使用 **Fckeditor** 编辑器能够进行样式的布局、文本格式化等操作而无需从头进行开发。对于 **Fckeditor** 编辑器而言，**Fckeditor** 编辑器是免费和开源的，开发人员能够免费的下载 **Fckeditor** 编辑器并进行二次开发，极大的简化富文本功能的开发。

### 29.4.2 使用 SQLHelper

**SQLHepler** 是一个数据库操作的封装，使用 **SQLHelper** 类能够快速的进行数据的插入、查询、更新等操作而无需使用大量的 **ADO.NET** 代码进行连接，使用 **SQLHelper** 类为开发人员进行数据操作提供了极大的遍历，在现有的系统中，在解决方案管理器中可以选择添加现有项添加现有的类库的引用，也可以通过自行创建类进行引用。

在前面的 **ASP.NET** 留言本中详细的讲解了如何使用 **SQLHepler** 类进行数据操作，使用 **SQLHepler** 类能够无需自己创建 **ADO.NET** 对象进行复杂的数据操作。

### 29.4.3 配置 Web.config

**Web.config** 文件为系统的全局配置文件，在 **ASP.NET** 中 **Web.config** 文件提供了自定义可扩展的系统配置，这里同样可以通过配置 `<appSettings/>` 配置节配置自定义信息，示例代码如下所示。

```
<appSettings>
  <add key="server" value="(local)"/>           //编辑 server 项
  <add key="database" value="guestbook"/>       //编辑 guestbook 项
  <add key="uid" value="sa"/>                   //编辑 uid 项
  <add key="pwd" value="sa"/>                   //编辑 pwd 项
  <add key="look" value="false"/>               //编辑 look 项
</appSettings>
```

上述代码对配置文件 **Web.config** 进行了相应的配置，`<appSettings/>` 配置节的配置信息能够在程序中通过 **ConfigurationManager.AppSettings** 获取，在 **SQLHelper** 类中就使用 **ConfigurationManager.AppSettings** 进行相应的自定义配置节的参数值的获取。这些配置节的相应的意义如下所示。

- ❑ **server:** **server** 项，用于配置数据库服务器的服务器地址。
- ❑ **database:** **database** 项，用于配置数据库服务器的数据库名称。
- ❑ **uid:** **uid** 项，用于配置数据库服务器的用户名。
- ❑ **pwd:** **pwd** 项，用于配置数据库服务器的密码。
- ❑ **look:** **look** 项，用于配置用户是否需要登录才能进行查看。

在配置了 **Web.config** 中 `<appSettings/>` 配置的信息后，不仅 **SQLHelper** 类能够进行相应参数的获取，在



应用程序中也能够获取 **Web.config** 中<appSettings/>配置节的参数值。

## 29.5 系统界面和代码实现

在 **ASP.NET** 校友录系统中使用了 **Fckeditor** 以及 **SQLHelper** 简化了 **HTML** 编辑器的开发和数据操作，在系统界面编写和代码实现上也更加容。，对于校友录系统而言，具有比较多的页面，这些页面用于注册、登录、发布日志和管理。

### 29.5.1 用户注册实现

在用户进行校友录系统登录前必须进行注册，对于注册而言，本书的前面的模块章节以及 **ASP.NET** 留言本项目都有比较详细的介绍，这里就不在做过多的介绍，用户注册只需要将数据插入到数据库即可，注册页面 **HTML** 核心代码见光盘中源代码\第 29 章\29-1\29-1\register.aspx。

上述代码进行了用户注册页面的基本布局，当用户打开校友录页面时，系统会提示用户必须要进行登录操作，如果用户没有用户惟一则必须先进行注册，注册页面如图 29-14 所示。



图 29-14 注册页面

当用户进行注册时，需要将数据插入到数据库中，使用 **SQLHelper** 类能够简化数据操作，示例代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    try
    {
        string strsql
        = "insert into register (username,password,sex,picture,IM,information,others,ifisuser,userroot)
        values ('" + TextBox1.Text + "','" + TextBox2.Text + "','" + DropDownList1.Text + "','" +
        TextBox3.Text + "','" + TextBox4.Text + "','" + TextBox5.Text + "','" + TextBox6.Text + "','0,0)";
        SQLHelper.SQLHelper.ExecNonQuery(strsql); //执行 SQL 语句
        Response.Redirect("login.aspx"); //注册后跳转到登录页面
    }
    catch
    {
        Response.Redirect("default.aspx"); //出错后跳转到首页
    }
}
```

当用户执行注册后，如果注册成功系统就会跳转到登录页面进行登录操作，如果没有注册成功（抛出异常），则系统会认定用户执行了非法操作，会跳转到首页。在进行注册时，默认情况下 **ifisuser** 字段为 **0**，用户注册后并不能够立即通过，需要管理员进行身份验证。

注意：在进行注册时首先需要进行查询，查询是否已经有现有的用户，这里可以参考注册模块，由于前面已经讲解了很多关于注册的操作，这里就不再详细讲解如何实现。

29.5.2 用户登录实现

用户登录操作在前面的章节中讲的非常的多，并且在模块篇中还详细的介绍了用户登录模块的开发，这里可以使用简单的登录模块进行登录操作即可而无需实现复杂的登录控制。登录页面 **HTML** 核心代码见光盘中源代码\第 29 章\29-1\29-1\login.aspx。

用户注册完成后就会跳转到登录页面，登录页面能够给用户配置相应的 **Session** 对象以存储用户状态，登录界面布局后如图 29-15 所示。

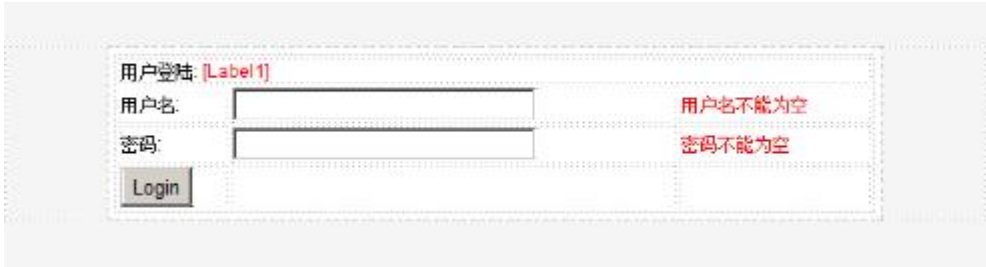


图 29-15 登录界面布局

当用户单击 **【Login】** 按钮时进行登录，使用 **SQLHelper** 类能够快速进行查询，示例代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    string strsql = "select * from register where username='" + TextBox1.Text + "' and password='" +
        TextBox2.Text + "'"; //编写 SQL
    SqlDataReader sdr = SQLHelper.SQLHelper.ExecReader(strsql); //执行查询
    if (sdr.Read())
    {
        Session["username"] = TextBox1.Text; //用户名
        Session["userid"] = sdr["id"].ToString(); //用户 ID
        Session["admin"] = sdr["userroot"].ToString(); //管理员判断
        Response.Redirect("friends.aspx"); //页面跳转
    }
    else
    {
        Label1.Text = "无法登录,用户名或密码错误"; //提示错误登录
    }
}
```

当用户进行登录后，系统会为用户赋予三个 **Session** 对象，这个三个对象的意义为用户名、用户 **ID** 和管理员判断。在用户表中，其字段 **userroot** 用于判断是否为管理员，如果 **userroot** 的值为 **0**，说明不是管理员，否则说明该用户是一个管理员用户。

29.5.3 校友录页面规划

校友录页面是校友录系统中最丰富的页面，大部分用户都会在校友录页面中停留较长的时间，在校友录页面不仅要呈现相应的日志，还需要呈现黑板报、公告、管理员等信息，在开发校友录页面时，首先需要进行页面规划。页面规划如图 29-16 所示。



图 29-16 校友录页面规划

正如图 29-16 所示，校友录页面的基本规划可以分为 5 块，这 5 块内容如下所示。

- ❑ 1：显示头部信息，包括 logo 等。
- ❑ 2：背景，用户填充背景颜色。
- ❑ 3：显示公告，管理员等。
- ❑ 4：显示最新发布的日志。
- ❑ 5：显示最新加入的同学，最热门的日志等等。

为了提高用户的友好度，可以将管理员命名为“值日生”，而将用户命名为“同学”，更加有校园的感觉。

29.5.4 自定义控件实现

在进行了页面规划后，就需要使用自定义控件进行相应的数据成呈现，例如呈现值日生、最新加入的同学等。在页面进行数据呈现的自定义控件并没有多大的难度，但是自定义控件能够方便开发和维护，当进行管理员显示的开发时，只需要修改相应的自定义控件即可。

1. 值日生控件

值日生控件用于显示校友录系统的管理员，管理员在校友录系统中被称为“值日生”，这样具有更好的友好度，值日生控件实现代码如下所示。

```
namespace DiaryAdmins
{
    [ToolboxData("<{0}:Myadmins runat=server></{0}:Myadmins>")] //控件呈现形式
    public class Myadmins : WebControl
    {
        protected override void RenderContents(HtmlTextWriter output)
        {
            try
            {
```

```
StringBuilder str = new StringBuilder(); //使用 String
string strsql = "select * from register where userroot=1 order by id desc"; //创建 SQL 语句
SqlDataReader sdr = SQLHelper.SQLHelper.ExecReader(strsql); //查询内容
while (sdr.Read()) //遍历对象
{
    str.Append("<span style=\"color:white\"><a href=\"userindex.aspx?uid=\" + sdr[\"id\"] + \"\">"
        + sdr[\"username\"] + "</span></a><br/>"); //输出 HTML
}
output.Write(str); //呈现 HTML
}
catch
{
    output.Write(""); //输出空
}
}
```

上述代码编写了值日生控件，当使用值日生控件，能够将校友录的管理员呈现在页面中。

## 2. 加入校友控件

加入校友控件用于呈现最新加入的校友，校友能够关注最新加入的校友并和他成为好友，加入校友控件代码如下所示。

```
namespace AddFriends
{
    [ToolboxData("<{0}:NewFriends runat=server></{0}:NewFriends>")] //控件呈现形式
    public class NewFriends : WebControl
    {
        protected override void RenderContents(HtmlTextWriter output)
        {
            try
            {
                StringBuilder str = new StringBuilder(); //使用 String
                string strsql = "select top 10 * from register where userroot=0 order by id desc"; //编写 SQL
                SqlDataReader sdr = SQLHelper.SQLHelper.ExecReader(strsql); //执行查询
                while (sdr.Read()) //遍历对象
                {
                    str.Append("<span style=\"color:white\"><a href=\"userindex.aspx?uid=\" + sdr[\"id\"] + \"\">"
                        + sdr[\"username\"] + "</span></a><br/>"); //输出 HTML
                }
                output.Write(str); //呈现 HTML
            }
            catch
            {
                output.Write(""); //输出空串
            }
        }
    }
}
```

上述代码与值日生控件不同的是，值日生控件通过遍历用户表中的 **userroot** 为 **1** 的用户，而校友控件是遍历用户表中 **userroot** 为 **0** 的用户，**userroot** 字段用于辨别用户身份，可以通过 **userroot** 字段进行筛选。

### 29.5.5 校友录页面实现

29.5.3 中的图片可以作为页面布局的规范，页面布局人员能够使用该图片作为页面布局的蓝本进行页面布局，校友录页面布局头部 **HTML** 代码如下所示。



```
<div class="top">
    
</div>
```

在校友录界面头部布局实现中，需要使用 **logo** 进行页面呈现，这里可以使用 **HTML** 控件进行图片呈现。在显式了 **logo** 之后，就需要呈现 **banner** 标签。**banner** 标签的样式在 **CSS** 文件中进行编写，在实际的校友录系统中，可以直接使用，示例代码如下所示。

```
<div class="banner">
</div>
```

在实现了 **banner** 标签后，就需要实现校友录页面中最重要的页面，即标签 **center** 内的内容。标签 **center** 中包括 **main\_board**、**main\_site** 以及 **main\_right** 标签，示例代码见光盘中源代码\第 29 章\29-1\29-1\friends.aspx。

在编写了校友录页面的主窗体后，就能够编写校友录底部信息，示例代码如下所示。在 **end** 标签中，开发人员能够进行版权的声明和编写。

```
<div class="end">
    校友录系统由 xx 开发完成
</div>
```

在校友录页面中使用了 **GridView** 控件和自定义控件，**GridView** 控件主要是用于呈现日志数据，其排序方式是按照最后回复时间进行排序，而自定义控件包含值日生控件和加入校友控件，用户呈现相应的用户数据。

注意：在使用自定义控件时，可能会提示 **SQLHelper** 类异常，开发人员可以不予理会。开发人员也能够自己编辑异常处理进行错误信息处理。

29.5.6 日志发布实现

在日志发布页面，用户能够使用 **HTML** 编辑器进行富文本编辑，这样就提高了交互性。对于用户而言，也能够使用 **HTML** 编辑器编写更多丰富的内容，包括音乐分享和文件上传。日志发布页面只需要将数据插入到相应的表即可，日志发布页面示例 **HTML** 核心代码见光盘中源代码第 29 章\29-1\29-1\new.aspx。

上述代码使用了同日志显示页面相同的 **CSS** 和样式进行布局，在一些相同的应用中使用相同的样式和布局能够提高用户的熟悉程度，让用户能够尽快适应。当用户填写了相应的日志之后，就能够进行日志的提交，日志提交代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    try
    {
        string strsql = "insert into diary (title,author,time,content,marks,classid,userid,hits) values ('" +
            TextBox1.Text + "','" + Session["username"].ToString() + "','" + DateTime.Now +
            "','" + FCKeditor1.Value + "','0,'" + DropDownList1.Text + "','" +
            Session["userid"].ToString() + "','0)"; //编写 SQL 语句
        SQLHelper.SQLHelper.ExecNonQuery(strsql); //执行插入
        Response.Redirect("friends.aspx"); //页面跳转
    }
    catch
    {
        Label3.Text = "出现错误,请检查日志"; //提示错误信息
    }
}
```

在页面载入时，首先需要进行用户的身份的判断才能够进行相应的操作，如果用户没有登录则不允许进行日志操作，在载入时进行判断需要使用 **Page\_Load** 方法，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
```

```
{
    if (Session["username"] == null || Session["userid"] == null)           //如果未登录
    {
        Response.Redirect("login.aspx");                                     //跳转到登录页
    }
}
```

如果用户没有登录或者登录超时，则会跳转到登录页面重新进行登录操作。

### 29.5.7 日志修改实现

日志修改页面基本同日志添加页面相同，这里就不再重复 **HTML** 代码，日志修改页面中的控件基本同日志添加页相同，而在日志修改页面中需要使用控件进行传递的参数的存放，当需要进行修改等操作时可以使用传递的控件进行日志修改。

在页面加载时，需要通过传递的参数进行日志的查询和控件中数据的填充，而当用户进行修改时，需要判断用户是否是作者，否则不运行修改功能，日志页面加载时实现代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        if (Session["username"] == null || Session["userid"] == null)           //判断状态
        {
            Response.Redirect("login.aspx");                                     //页面跳转
        }
        else
        {
            string strsql = "select * from diary where id=" + Request.QueryString["id"] + "";
            SqlDataReader sdr = SQLHelper.SQLHelper.ExecReader(strsql);           //编写 SQL
            if (sdr.Read())                                                         //存在该文章
            {
                if (sdr["userid"].ToString() == Session["userid"].ToString()
                    || Session["admin"].ToString() == "1")                       //判断权限
                {
                    TextBox1.Text = sdr["title"].ToString();                     //控件值初始化
                    Label1.Text = sdr["author"].ToString();                     //控件值初始化
                    Label2.Text = sdr["time"].ToString();                       //控件值初始化
                    FCKeditor1.Value = sdr["content"].ToString();               //控件值初始化
                    DropDownList1.Text = sdr["classid"].ToString();              //控件值初始化
                }
                else
                {
                    Response.Redirect("error/cmodi.aspx?id=" + sdr["id"]);        //跳转错误
                }
            }
            else
            {
                Response.Redirect("login.aspx");                                 //登录跳转
            }
        }
    }
}
```

当页面加载时会判断用户是否登录，如果用户没有登录则会跳转到登录页面，否则进行数据库查询判断该文章是否为当前用户所能够操作的文章，如果不能够操作文章，则跳转到错误页面。当用户进行修改时，执行 **UPDATE** 语句对数据库中的数据进行更改，示例代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
```

```
{
    try
    {
        string strsql = "update diary set title='" + TextBox1.Text + "',content='" + FCKeditor1.Value + "' w
                                here id='" + Request.QueryString["id"] + "'"; //更新语句
        SQLHelper.SQLHelper.ExecNonQuery(strsql); //执行更新
        Response.Redirect("news?id=" + sdr["id"]); //页面跳转
    }
    catch
    {
        Label3.Text = "出现错误,请检查日志"; //抛出异常
    }
}
```

当用户进行日志更新时，只需要进行 **UPDATE** 语句的执行就能够执行更新，更新完成后可以跳转到相应的页面。

注意：管理员也能够进行日志的修改，通过判断 **userroot** 字段的值进行管理员权限判断，如果 **userroot** 的值为 1 则说明该用户是管理员，可以无条件的对日志进行更改。

### 29.5.8 管理员日志删除

对于删除页面而言，同前面的章节一样无需实现 **HTML** 页面的呈现，只需要进行相应的逻辑实现即可，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    if (Session["admin"] == null) //判断有没有登录
    {
        Response.Redirect("friends.aspx"); //跳转到校友录
    }
    else
    {
        if (Session["admin"].ToString() == "1") //判断是不是管理员
        {
            string strsql = "delete from diary where id='" + Request.QueryString["id"] + "'";
            SQLHelper.SQLHelper.ExecNonQuery(strsql); //执行删除操作
            Response.Redirect("friends.aspx"); //页面跳转
        }
        else
        {
            Response.Redirect("errors/cdelete.aspx?id=" + Request.QueryString["id"]);
        }
    }
}
```

当用户进行删除操作时，页面需要对用户进行身份验证。如果用户是管理员，则允许执行操作，否则会跳转到错误页面，并提示不运行进行相应的操作。

### 29.5.9 日志显示页面

在校友发布了日志后，就能够进行日志显示，其他的校友也能够进行相应的页面进行日志的查看和评论。校友进行日志查看，也能够对自己的日志进行编辑处理，管理员能够对校友的相关日志进行查看、删

除、编辑等操作。

在日志显示时，包括多个小模块进行数据显示，这些小模块包括日志显示模块、评论显示模块以及评论模块。日志显示页面 **HTML** 代码基本同日志发布页面相同，而各个模块根据其显示的效果而不同。日志显示模块 **HTML** 代码见光盘中源代码\第 29 章\29-1\29-1\shownew.aspx。

其中，页面代码使用了 **DataList** 控件和 **SqlDataSource** 控件进行日志显示，通过传递的页面参数 **id** 进行数据查询并呈现在相应的 **DataList** 控件中。当用户评论后，日志显示页面还需要对相应的评论进行显示，评论显示页面 **HTML** 代码见光盘中源代码\第 29 章\29-1\29-1\shownew.aspx。

显式评论代码用于呈现用户的评论并按照一定的 **HTML** 格式输出。当用户阅读日志后并希望能够进行相应的评论，可以在页面的评论模块中进行评论操作，评论模块 **HTML** 代码见光盘中源代码\第 29 章\29-1\29-1\shownew.aspx。当用户填写完成标题和内容后就能够通过按钮控件进行评论的提交，评论提交代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    string strsql = "insert into diarygbook (title,time,content,userid,diaryid) values"
        ("+" + TextBox1.Text + "+","+" + DateTime.Now + "+","+" + TextBox2.Text + "+",""
        + Session["userid"].ToString() + "+","+" + Request.QueryString["id"] + "+");    //编写 SQL
    SQLHelper.SQLHelper.ExecNonQuery(strsql);    //执行 SQL
    Response.Redirect("shownew.aspx?id=" + Request.QueryString["id"]);    //页面跳转
}
```

当用户进行留言操作后会跳转到同样的页面并呈现用户的留言信息。

### 29.5.10 用户索引页面

当用户发布日志后，用户可以通过索引页面索引自己的日志并查看相关的日志。不仅是一个用户，而且校友和管理员都能够进行用户索引的查看，用户索引页面 **HTML** 核心代码见光盘中源代码\第 29 章\29-1\29-1\userindex.aspx。

管理员可以在用户索引页面进行用户的管理，用户也能够在用户索引页面进行相应的用户的信息的查看，例如一个校友用户对他的另一个校友感兴趣，可以点击用户名跳转到用户索引页面查看该用户曾经发布的信息。

### 29.5.11 管理员用户删除

由于数据库中数据表的约束关系，在删除用户时需要对多个表进行操作，用户删除页面同样不需要呈现相应的 **HTML** 代码而可以直接执行逻辑处理，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    if (Session["admin"] == null)    //判断是否登录
    {
        Response.Redirect("friends.aspx");    //页面跳转
    }
    else
    {
        if (Session["admin"].ToString() == "1")    //判断是否为管理员
        {
            string strsql = "delete from diary where userid=" + Request.QueryString["uid"] + "";
            string strsql1 = "delete from diarygbook where userid=" + Request.QueryString["uid"] + "";
            string strsql2 = "delete from register where id=" + Request.QueryString["uid"] + "";
            SQLHelper.SQLHelper.ExecNonQuery(strsql);    //删除用户日志
            SQLHelper.SQLHelper.ExecNonQuery(strsql1);    //删除用户留言
            SQLHelper.SQLHelper.ExecNonQuery(strsql2);    //删除用户信息
        }
    }
}
```



```
        Response.Redirect("friends.aspx");
    }
    else
    {
        Response.Redirect("errors/cdelete.aspx?id=" + Request.QueryString["id"]);
    }
}
}
```

在进行删除用户操作时，首先删除用户的日志中的数据，清空其发布的日志，清空后再删除用户的评价数据，清空其对日志发布的评论，清空后才能够进行用户信息的删除。

注意：当数据库中表与表之间包含约束关系，由于数据完整性的原因，如果操作不按照数据规范进行操作，则系统会抛出异常。

29.6 用户体验优化

在前面的小结中只是将应用程序所需要的功能简单的实现了，而简单的实现并不能够满足现今越来越丰富的应用程序要求。用户体验优化也是现在应用程序开发的必经阶段，提高用户体验有助于快速的加入和使用应用程序。

29.6.1 超链接样式优化

超链接样式是用户体验优化中一个非常简单却非常重要的部分。超链接显示着不同连接之间的样式，用户能够通过超链接进行跳转。单击【F5】快捷键运行现有的应用程序，如图 29-17 所示。



图 29-17 应用程序运行

从图 29-17 中可以看出，其超链接文本样式为默认文本样式，这样就显得非常的不友好。在进行样式控制时，可以在 CSS 文件中进行超文本连接样式的控制。为了能够更好的配合校友录系统，这里将超文本链接样式设置为蓝色链接样式，示例代码如下所示。

```
a:link
{
    text-decoration: none;
    color: #3b5888;
```

```
}
a:active
{
text-decoration: none;
color: #3b5888;
}
a:visited
{
text-decoration: none;
color: #3b5888;
}
/*设置超链接鼠标移动样式*/
a:hover
{
text-decoration:underline;
color:White;
background: #3b5888;
}
```

上述代码使用了超链接文本控制样式进行样式控制。其中包括了 **a:link**、**a:active**、**a:visited** 和 **a:hover**。在 **CSS** 层叠样式表中，样式是能够继承的。在校友录应用程序开发中，可以定义一个全局超链接样式表，另外，也可以为单独的某个样式进行超链接文本样式控制。示例代码如下所示。

```
.main_right a:link
{
text-decoration: none;
color: #3b5888;
}
.main_right a:active
{
text-decoration: none;
color: #3b5888;
}
.main_right a:visited
{
text-decoration: none;
color: #3b5888;
}
.main_right a:hover
{
text-decoration:underline;
color:#3b5888;
font-weight:bolder;
background:white;
}
```

**main\_right** 是样式表中用于控制侧边栏的样式。在校友录系统中，系统希望右侧的边栏的超链接样式与全局的超链接样式不同，那么开发人员就能够通过继承的方法将相应的层中的样式的超链接文本样式进行覆盖，从而呈现另一种超链接文本样式。

在定义了一个全局超链接文本样式后，全局的超链接文本样式都会被更改成全局样式，如图 29-18 所示。而局部定义的超链接文本样式会被局部样式覆盖，如图 29-19 所示。



图 29-18 全局样式



图 29-19 局部样式

29.6.2 默认首页优化

在前面的代码实现中，并没有制作默认首页，这样就会导致用户访问网站时找不到网站页面。默认首页对网站整体应用而言是非常重要的，当用户访问网站时，默认首页会首先展示在用户面前。在默认首页加载时，应该首先跳转到校友录页面。示例代码如下所示。

```
public partial class _default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        Response.Redirect("friends.aspx"); //跳转到校友录页面
    }
}
```

当系统载入首页时，首先会执行首页的 **Page\_Load** 事件。上述代码在 **Page\_Load** 事件中进行了页面跳转。当用户打开并访问页面时，页面即会跳转到 “**friend.aspx**” 页面。

29.6.3 导航栏编写

导航栏用于用户操作的指引。当用户进入 **Web** 系统时，通常需要通过导航栏进行应用程序功能的查找。校友录系统的导航栏同样需要编写事务逻辑判断以便不同身份权限的用户查看的信息是不相同的。在校友录系统中，包括以下两种用户权限。

- ❑ 普通校友：该用户是普通校友，该用户能够进行日志的发表和用户信息的索引。
- ❑ 校友录管理员：该用户是校友录管理员，不仅如此，管理员也是校友录中校友用户的一份子，该用户也能够进行日志的发布和留言的发布。

在导航栏中进行逻辑事务判断并编写成为 **js** 文件，通过其他文件的引用能够在多个不同页面中进行相同的功能的实现。**js** 文件示例代码如下所示。

```
<%
if (Session["admin"].ToString() == "1")
{
%>
document.write('<div style="margin:5px 5px 5px 5px;padding:5px 5px 5px 5px;border:1px dashed #ccc">');
document.write('');
你好:<% Response.Write(Session["username"].ToString()); %> <br/>
 你的身份是
<a href="admin/default.aspx"> <span style="color:Red">管理员</span></a>&nbsp;&nbsp;&nbsp;');
document.write('<br/>');
<a href=" ../logout.aspx">注销</a>&nbsp;&nbsp;&nbsp;');
document.write('</div>');
```

```
<%
    }
    else
    {
%>
    document.write('<div style="margin:5px 5px 5px 5px;padding:5px 5px 5px 5px;border:1px dashed #ccc">');
    document.write('');
    document.write('你好:<% Response.Write(Session["username"].ToString()); %> <br/>');
    document.write('你的身份是<span style="color:Red">普通用户</span>&nbsp;&nbsp;&nbsp;');
    document.write('<br/>');
    document.write('<a href="../../logout.aspx">注销</a>&nbsp;&nbsp;&nbsp;');
    document.write('</div>');
<%
    }
%>
```

上述代码不仅简单的实现了导航栏的编写，还实现了用户信息的简约查看，如图 29-20 所示。如果用户是管理员，则会以管理员的导航样式形式进行呈现，如图 29-21 所示。



图 29-20 用户信息导航栏    图 29-21 管理员导航样式

上述代码制作了一个用于呈现导航的 **js**，通过调用此 **js** 文件能够呈现不同的导航，示例代码如下所示。

```
<div class="main_right">
    <script src="js/banner.aspx" type="text/javascript"></script><br/>
    <cc2:NewFriends ID="NewFriends1" runat="server" />
</div>
```

在需要使用导航的页面添加 **js** 调用，则相应的页面就能够呈现 **js** 导航信息。**js** 导航并不局限于网站页面或功能的导航，在很多情况下，**js** 导航还能够制作用户控制面板、网站页头、网站页尾等通用模块。

**注意：****js** 导航不仅仅能够制作带有逻辑的页面引用，**js** 还能够编写 **HTML** 进行页面题头、题尾等通用模块的编写，这样不仅能够在多个页面中使用，也方便了系统的维护。但如果在网站中大量的使用 **js** 页面进行逻辑判断，也可能会造成性能问题。

29.6.4 AJAX 留言优化

**AJAX** 能够提高应用程序的用户体验，在前面的留言本章节中就使用了 **AJAX** 用于无刷新应用程序的实现。在校友录系统的实现中，同样可以使用 **AJAX** 进行无刷新实现，示例代码见光盘中源代码\第 29 章\29-1\29-1\shownew.aspx。

上述代码将留言进行呈现，为了能够使用 **AJAX** 进行无刷新功能的实现，需要将此控件防止在 **AJAX** 的局部更新控件中。在进行了数据绑定控件的模板配置后，还需要配置数据源，示例代码见光盘中源代码\第 29 章\29-1\29-1\shownew.aspx 中数据源的配置。

其中，页面代码进行了数据源的配置。值得注意的是，数据在 **AJAX** 应用中也是非常重要的。当用户执行数据更新时，数据绑定控件还需要通过数据源重绑定进行数据更新。当用户单击【留言】按钮后，系统会执行数据库插入操作并跳转到当前页面进行页面重加载。使用 **AJAX** 进行页面局部更新就不需要进行页面跳转。



在应用程序代码控制中，值需要进行数据绑定控件的数据重绑定即可。在数据绑定控件中，大部分的数据绑定控件都具有 **DataSourceID** 属性。该属性用于指定当前用户使用的数据源，在指定了数据源之后，数据并不会自动进行绑定，还需要使用 **DataBind** 方法进行数据绑定，示例代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    string strsql = "insert into diarygbook (title,time,content,userid,diaryid) values
    ('"+TextBox1.Text+"','"+DateTime.Now+"','"+TextBox2.Text+"','"+Session["userid"].ToString()+"','"+
    Request.QueryString["id"]+"')"; //生成 SQL 语句
    SQLHelper.SQLHelper.ExecNonQuery(strsql); //执行 SQL 语句
    DataList2.DataSourceID = "SqlDataSource2"; //配置数据源属性
    DataList2.DataBind(); //数据源重绑定
}
```

上述代码使用了 **DataBind** 方法进行数据重绑定。当执行了数据重绑定后，数据绑定控件将能够直接进行数据呈现而无需再次页面跳转进行页面呈现，如图 29-22 所示。

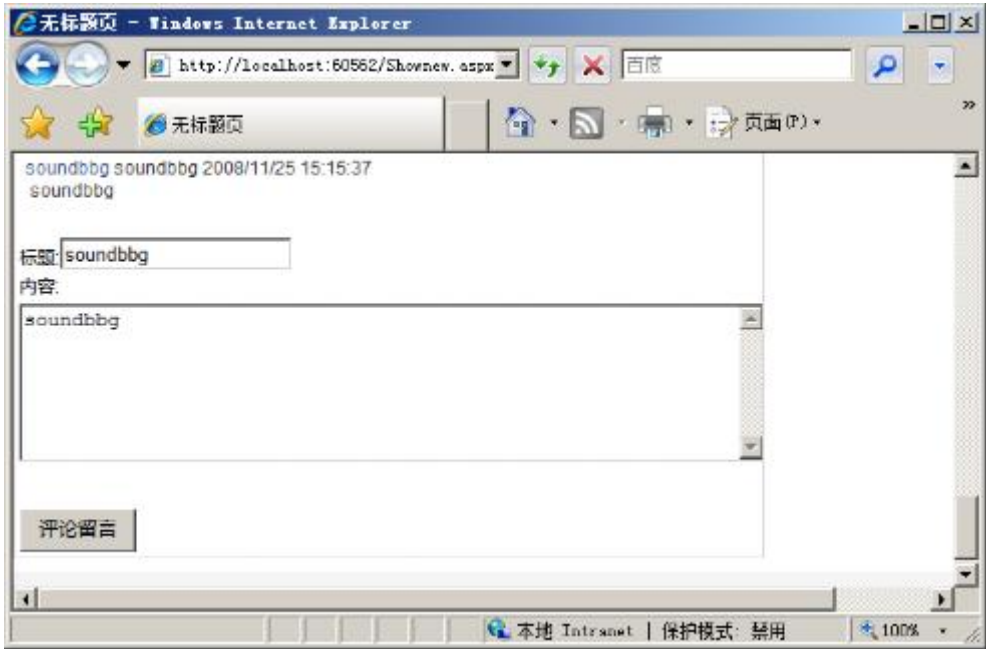


图 29-22 无刷新页面重绑定

值得注意的是，在使用 **AJAX** 控件进行留言等数据操作时，其控件的状态依旧会保持原有的数据状态，例如图 29-22 中所示。当向某个日志进行评论后，文本框控件中的文本并没有被清楚。在执行操作时还需要进行文本清除，示例代码如下所示。

```
TextBox1.Text = "";
TextBox2.Text = "";
```

注意：由于 ASP.NET 对 **AJAX** 的封装，**AJAX** 应用程序的开发已经非常容易，但是在 **AJAX** 应用程序开发时，还需要注意 **AJAX** 运行过程中的控件状态。

29.6.5 优化留言表情

在很多应用程序中，留言和聊天的文本中都会出现表情，如 **QQ** 就可以使用表情，如图 29-23 所示。同样，在 **Web** 应用程序中也可以使用表情，最常见的就是论坛、博客和新闻评论了，如图 29-24 所示。在校友录系统中，只有正文能够使用表情，而该表情是通过 **Fckeditor** 进行实现的。为了让校友录系统更加丰富，可以使用 **C#**和 **js** 共同实现表情功能。

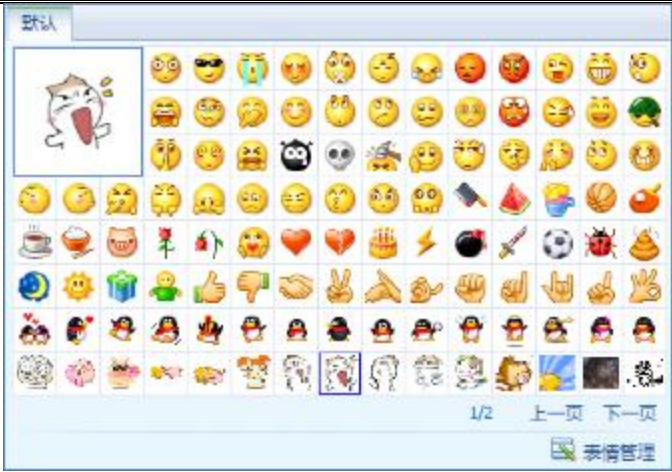


图 29-23 QQ 表情



图 29-24 网站应用表情

在制作和实现表情功能时，首先需要考虑表情是如何呈现的。从现有的应用中可以看出，当单击了表情之后，表情会以一种字符的形式呈现在相应的控件中（主要是文本框控件），如图 29-25 所示。当用户单击发布按钮进行留言发布后，该字符会被转义成表情图片进行呈现。所以，表情功能实现的第一步就是表情的呈现。



图 29-25 表情呈现

1. 表情呈现和选择

正如图 29-25 所示，微笑的表情是字符“:)”。当用户单击【笑脸】表情时，首先笑脸表情会转义成字符串“:)”呈现在文本框控件中。表情可以使用图片按钮控件进行呈现，当单击相应的【表情】按钮时，会触发方法进行文本框中文本的呈现。

值得注意的是，这种方法当有多个表情时会使代码变得非常的复杂和冗长，不仅如此，这种方法也增加了页面的控件数量。当页面载入时，ASP.NET 托管程序首先会将控件进行转义和呈现并生成新的页面模型，当控件的数量增大时，难免会页面性能问题。为了解决这个问题，可以使用 JavaScript 进行按钮控件的事件模拟。

右击现有项，在下拉菜单中选择【新建文件夹】选项，在新建文件夹选项中选择【添加新项】，选项，在弹出菜单中选择【JScript】文件，如图 29-26 所示。



图 29-26 新建 JScript 文件

**JScript** 文件用于编写 **JavaScript** 函数，该函数能够在其他页面进行调用。添加表情函数示例代码如下所示。

```
function add_smile(smile)
{
    var str=document.getElementById("TextBox2");
    str.value+=smile;
}
```

上述代码使用了 **JavaScript** 创建了一个添加表情函数，其过程非常简单。该函数拥有一个参数，这个参数是表情的字符，如 “:)” 字符串。当执行了函数时，该函数首先会查找 **ID** 为 **TextBox2** 的文本框，查找后会将传递的字符串添加到相应的文本框控件中。编写了函数后就需要在页面中进行函数的引用，示例代码如下所示。

```
<script src="js/JScript1.js" type="text/javascript"></script>
```

上述代码声明了一个 **JavaScript** 页面的引用，当引用了该 **JavaScript** 页面后，该页面的标签就能够使用该页面提供的函数。在表情的呈现过程中，使用图片控件或图片按钮控件都是不合适的，这里可以直接使用 **HTML** 图片并通过使用 **add\_smile** 函数实现表情，示例代码如下所示。

```


')"/>

')"/>




```

上述代码呈现了若干表情，并编写了 **HTML** 控件的 **onclick** 事件。该事件通过传递参数添加到文本框控件中。例如当单击 **URL** 路径为 “smiles/0.gif” 的图片时，会触发 **add\_smile(':)')** 事件，该事件会传递一个 “:)” 字符串到文本框 **TextBox2** 中，如图 29-27 所示。

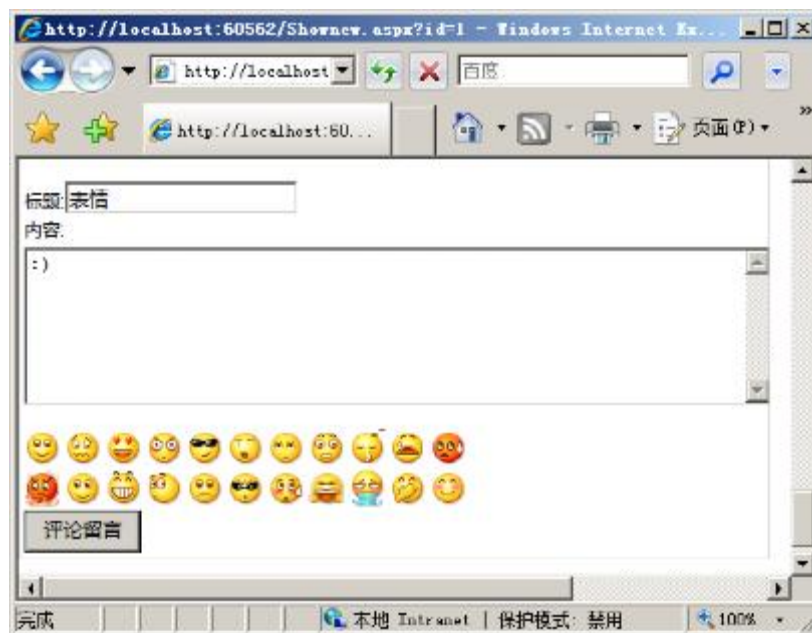


图 29-27 表情呈现和选择

## 2. 表情转义

如果直接单击【评论留言】按钮，系统将会进行数据插入。但是值得注意的是，“:)” 字符串并不会以表情的形式呈现，如果需要以表情的形式呈现，在执行数据插入前，还需要将表情转换成 **HTML** 代码，核心代码如下所示。

```
public string opFormatsmiles(object input)
{
    string data = input.ToString(); //获取传递的参数
```



```

data = data.Replace("&","&"); //替换特殊符号
data = data.Replace(""","/"); //替换特殊符号
data = data.Replace("'",""); //替换特殊符号
data = data.Replace("<","<"); //替换特殊符号
data = data.Replace(">",">"); //替换特殊符号
data = data.Replace(":o)", "<img src=\"smiles/13.gif\" ale=\"大笑\"/>"); //替换成图片
data = data.Replace(":)", "<img src=\"smiles/0.gif\" ale=\"我得意的笑\"/>"); //替换成图片
data = data.Replace(":s", "<img src=\"smiles/1.gif\" ale=\"委屈的很\"/>"); //替换成图片
data = data.Replace(":>", "<img src=\"smiles/2.gif\" ale=\"色色的笑\"/>"); //替换成图片
data = data.Replace(":~)", "<img src=\"smiles/3.gif\" ale=\"啊哦..呜呜..\"/>"); //替换成图片
data = data.Replace(":->", "<img src=\"smiles/4.gif\" ale=\"嘿嘿..\"/>"); //替换成图片
data = data.Replace(":<", "<img src=\"smiles/5.gif\" ale=\"我哭哭了..\"/>"); //替换成图片
data = data.Replace(":)", "<img src=\"smiles/6.gif\" ale=\"媚眼..\"/>"); //替换成图片
data = data.Replace(":o", "<img src=\"smiles/7.gif\" ale=\"有点小小的惊讶\"/>"); //替换成图片
data = data.Replace(":zz", "<img src=\"smiles/8.gif\" ale=\"睡觉觉咯..\"/>"); //替换成图片
data = data.Replace(":(", "<img src=\"smiles/9.gif\" ale=\"大哭特哭..\"/>"); //替换成图片
data = data.Replace(":..", "<img src=\"smiles/10.gif\" ale=\"..\"/>"); //替换成图片
data = data.Replace(":xx", "<img src=\"smiles/11.gif\" ale=\"我恼火的很\"/>"); //替换成图片
data = data.Replace(":p", "<img src=\"smiles/12.gif\" ale=\"笑笑\"/>"); //替换成图片
data = data.Replace(":ma", "<img src=\"smiles/14.gif\" ale=\"惊讶\"/>"); //替换成图片
return data;
}

```

上述代码将相应的字符串进行转换，从而呈现相应的表情的 **HTML** 图片。例如 “:)” 字符串会在应用程序执行时被替换成字符串 “<img src=\\smiles/0.gif\\ ale=\\我得意的笑\\/>”。当页面呈现时，该字符串会以 **HTML** 的形式呈现，这样用户看到的字符串就是一个表情而不是一个文本字串。在执行数据插入之前，还需要通过该函数进行转换，示例代码如下所示。

```

string strsql = "insert into diarygbook (title,time,content,userid,diaryid) values ('" + TextBox1.Text + "','" +
DateTime.Now + "','" + opFormatsmiles(TextBox2.Text) + "','" + Session["userid"].ToString() + "','" +
Request.QueryString["id"] + "')"; //转换后添加
SQLHelper.SQLHelper.ExecNonQuery(strsql); //执行 SQL 语句

```

上述代码在插入数据前使用了 **opFormatsmiles** 方法进行文本中字符串的替换，替换完成后就能够以 **HTML** 的形式呈现在留言信息中，如图 29-28 所示。

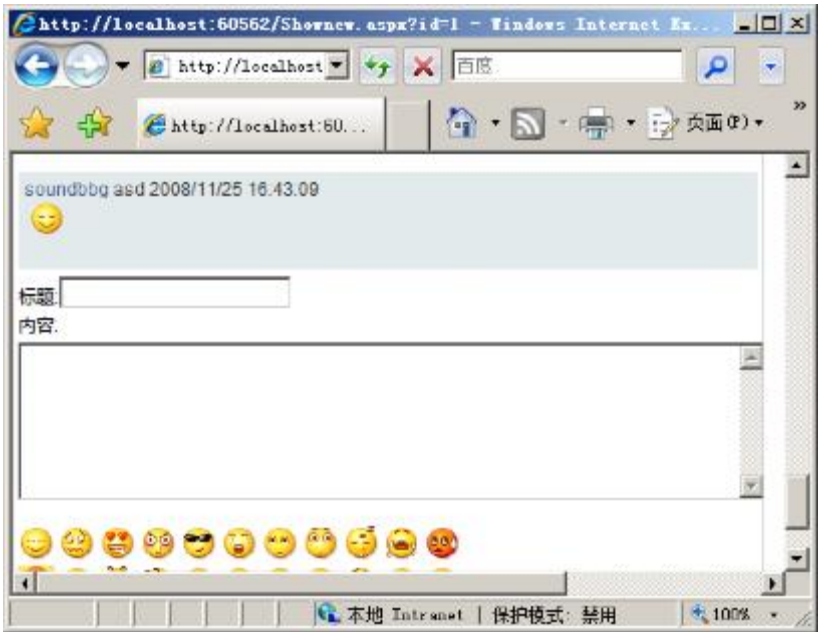


图 29-28 表情显示

注意：在表情功能的实现中，还可以不进行替换直接将表情字串添加到数据库中。当呈现留言或评论数据时，可以使用编程控制数据的呈现方式，即在呈现表情时进行字符串替换。



## 29.7 高级功能实现

在前面的功能实现的章节中，只是制作了基本的应用开发所需要的功能，其中还有板报、管理员管理、后台留言管理以及关键字过滤等功能没有实现。虽然这些功能的实现并不复杂，但是这些功能都是健壮的应用程序所必备的。

### 29.7.1 后台管理页面实现

虽然管理员能够在前台进行数据的管理，但是前台的数据管理往往非常不方便。复杂的前台管理功能不仅影响了用户体验，并且还暴露了管理功能和管理路径。后台管理页面不仅能够保护管理功能防止非法用户进行管理的尝试，后台还为管理员提供了统一的管理界面和管理工具，管理员能够在后台管理页面方便的进行数据管理。

单击【开始】菜单，在菜单中选择【程序】选项，找到【Microsoft Expression】选项并在 Microsoft Expression 选项中选择【Microsoft Expression Web 2】选项打开 Microsoft Expression Web 2 用于框架集的制作。在制作框架集之前首先需要确定框架的作用，这里可以创建一个【横幅和目录】形式的框架用于系统的管理，如图 29-29 所示。

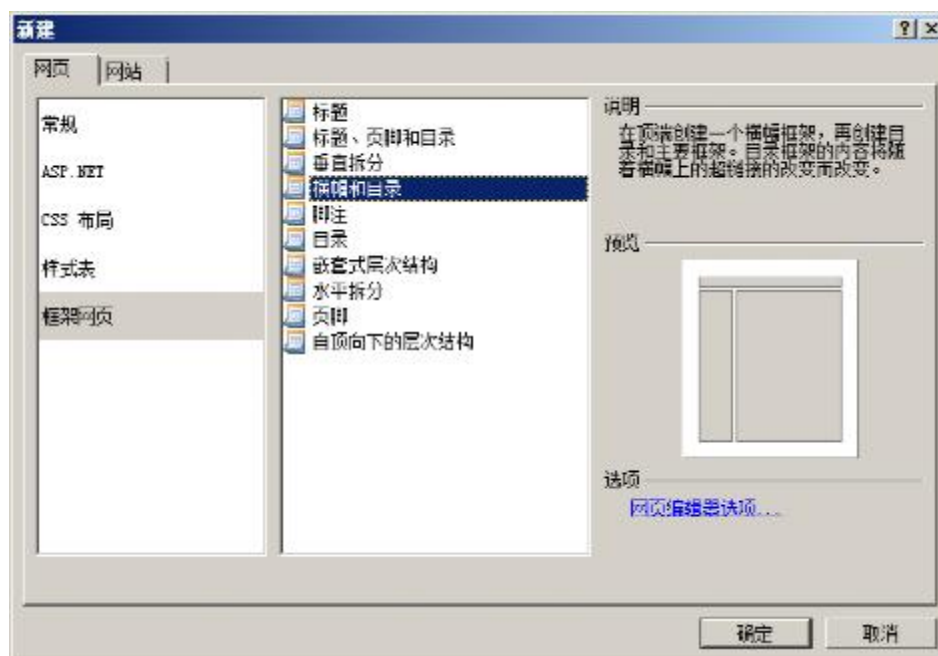


图 29-29 选择框架集

【横幅和目录】形式的框架包括三个窗口，从图 29-29 中可以看出，这三个窗口分别位于主窗口的上方、下方左侧和下方右侧。这里分别命名为 **head.aspx**、**left.aspx** 和 **center.aspx** 并将框架集保存为 **default.aspx**。这三个框架集的作用分别如下所示。

- ❑ **head.aspx**: 用于呈现头部信息，通常情况下该框架集用于美工作用，并不呈现实际的作用。
- ❑ **left.aspx**: 侧边栏用于快捷方式的存放，开发人员能够选择侧边栏进行相应的功能的设置。
- ❑ **center.aspx**: 中间部分用于呈现主工作区，当开发人员在侧边栏中选择了相应的快捷方式后，主工作区用于呈现工作所必须呈现的界面。

在确定了三个框架的基本作用后，可以编写相应的页面进行框架的呈现，示例代码如下所示。

```
<body style="background:white url('images/bg.png') repeat-x;">
<p></p>
</body>
```

上述代码实现了框架集头部代码，该代码用于实现头部布局。为了让管理人员方便的进行系统的管理和操作，可以在侧边栏使用 **TreeView** 控件进行导航，示例代码见光盘中源代码\第 29 章\29-1\29-1\admin\left.aspx。

其中，页面代码编写了一个 **TreeView** 控件用于后台系统的导航，**TreeView** 控件包括管理首页、日志管

理、用户管理和退出管理几个大块。这几个模块的作用如下所示。

- ❑ 管理首页：主要是用于全局配置，包括关键字管理等。
- ❑ 日志管理：包括日志的修改和删除，管理员能够在后台管理日志并进行日志的删除操作。
- ❑ 用户管理：用户管理包括用户密码的修改、信息的修改以及用户的删除。
- ❑ 退出管理：注销管理主要用于管理员的退出操作。

在确定了基本的管理模块后就可以针对管理模块进行后台页面的开发。

29.7.2 日志管理

虽然管理员能够在前台页面进行日志的管理操作，但是前台的操作毕竟十分有限，在后台管理页面中，由于已经确认了管理员的身份，则管理员能够对数据进行修改和删除操作。在进行修改和删除操作时，日志数据中的任何字段都能够被管理员修改。

日志管理页面需要展示日志数据，并提供管理员快捷的进行日志的修改和删除操作。在日志管理页面，可以使用 **GridView** 控件用于数据呈现。**GridView** 控件能够创建自定义连接进行高级的数据操作，这里可以自行创建【修改】选项和使用系统默认的【删除】选项，示例代码如下所示。

```
<asp:HyperLinkField DataNavigateUrlFields="id"
    DataNavigateUrlFormatString="dmodi.aspx?id={0}" Text="修改">
    <ItemStyle Width="25px" />
</asp:HyperLinkField>
<asp:HyperLinkField DataNavigateUrlFields="id"
    DataNavigateUrlFormatString="del.aspx?id={0}" Text="删除">
    <ItemStyle Width="25px" />
</asp:HyperLinkField>
```

上述代码只是 **GridView** 控件的一部分，用于呈现自定义修改超链接和删除超链接。在代码中，系统创建了【修改】选项和【删除】选项，【修改】选项是自定义选项，当管理员单击【修改】超链接时，系统会跳转到 **dmodi.aspx** 页面并进行数据的呈现。管理员能够在 **dmodi.aspx** 页面进行数据的修改和更新。当管理员单击【删除】超链接时，系统会删除相应的新闻信息。

日志管理的数据源配置不需要使用自动生成 **SQL** 语句选项进行智能的 **SQL** 操作的支持，因为在数据源操作的过程中，其中的【修改】选项和【删除】选项所需要实现的数据操作都是通过自定义模块进行实现。日志管理页面的数据源只需要连接数据即可，示例代码如下所示。

```
<asp:SqlDataSource ID="SqlDataSource1" runat="server"
    ConnectionString="<%= $ ConnectionStrings:friendsConnectionString %>"
    SelectCommand="SELECT * FROM [diary] ORDER BY [id] DESC">
</asp:SqlDataSource>
```

上述代码只使用了 **SqlDataSource** 控件的 **SelectCommand** 属性进行数据的呈现，如图 29-30 所示，管理员只需要对其中的数据进行查看和筛选即可。

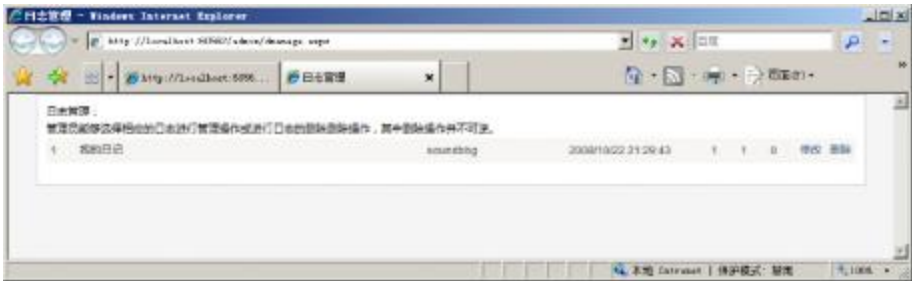


图 29-30 日志管理页面

29.7.3 日志修改和删除实现

在前台页面中，已经实现了日志的修改和删除。在后台页面中，日志的修改和删除操作也基本相同。

另外，管理员在修改日志时具备比前台修改更多的权限，包括前台不能够修改的字段，管理员也能够后台修改。

### 1. 日志修改实现

后台管理页面中的日志修改可以修改不同的用户、不同的字段，相比之下，从后台进行日志修改能够更加方便的进行多个保密字段的修改，日志修改页面 **HTML** 核心代码见光盘中源代码\第 29 章\29-1\29-1\admin\dmodi.aspx。

在前台页面中，管理员能够修改用户的基本信息，但是无法修改用户日志的发布时间和阅读次数。在后台系统中，管理员能够修改用户日志的发布时间以便能够修正用户的日志信息。另外，管理员还能够修改阅读次数，示例代码见光盘中源代码\第 29 章\29-1\29-1\admin\dmodi.aspx。

由于用户发布的日志是基于 **Fckeditor** 编辑器进行发布的，所以在后台日志管理页面中，同样需要使用 **Fckeditor** 进行日志修改，示例代码如下所示。

```
<td colspan="2">
    <FCKEditorV2:FCKEditor ID="FCKEditor1" runat="server" Height="300px">
    </FCKEditorV2:FCKEditor>
</td>
```

当管理员修改了相应的选项后，可以单击【修改】按钮进行日志的修改。当单击【修改】按钮后，系统还需要进行字段的检查才能够进行数据更新，示例代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    try
    {
        string strsql = "update diary set title='" + TextBox1.Text + "',content='" + FCKEditor1.Value + "',
        time='"+TextBox2.Text+"',hits='"+TextBox3.Text+" where id='" +
        Request.QueryString["id"] + "'"; //更新数据库
        SQLHelper.SQLHelper.ExecNonQuery(strsql); //执行更新语句
        Response.Redirect("dmanage.aspx"); //跳转到管理页面
    }
    catch
    {
        Label3.Text = "出现错误,请检查日志"; //提示异常信息
    }
}
```

当管理员单击【修改】按钮进行数据更新时，管理员所填写的字段都会更新到数据库中。用户能够在前台的相应页面进行查看。

### 2. 日志删除实现

日志删除实现的过程比较容易，但是同留言本系统一样，在执行数据的删除时同样要注意数据的约束性和完整性，删除操作代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    string strsql = "delete form diarygbook where diaryid='"+Request.QueryString["id"]+"'"; //删除评论
    string strsql1 = "delete from diary where id='"+Request.QueryString["id"]+"'"; //删除新闻
    SQLHelper.SQLHelper.ExecNonQuery(strsql); //执行删除
    SQLHelper.SQLHelper.ExecNonQuery(strsql1); //执行删除
    Response.Redirect("dmanage.aspx"); //页面跳转
}
```

执行了上述代码后，系统会将日志以及与日志有关的评论全部删除。但是在执行删除操作时，首先需要删除与目的数据相关的所有其他数据后才能够删除目的数据。

**注意：**在这几章实例章节中都强调了数据完整性需求，当删除某个数据前，一定要检查数据的约束性和完整性，以便能够正确删除数据。



### 29.7.4 评论删除实现

评论删除功能的实现非常简单，可以直接使用系统数据源控件提供的删除功能即可实现评论的删除。示例代码见光盘中源代码\第 29 章\29-1\29-1\admin\gmanage.aspx。

其中，页面代码配置了 **GridView** 控件以便该控件能够进行数据的呈现和删除功能的选择。由于该控件需要数据源支持删除操作，则数据源必须支持智能的生成插入、更新、删除的 **SQL** 语句，示例代码见光盘中源代码\第 29 章\29-1\29-1\admin\gmanage.aspx 中数据源的配置。

其中，页面中数据源控件的代码实现了数据的插入、更新和删除所需要使用的 **SQL** 语句的生成。当数据绑定控件执行了删除操作时，会触发数据源控件的 **DeleteCommand** 属性进行数据删除。

### 29.7.5 板报功能实现

在校友录的每个页面中，都有一个板报用于呈现相应的板报信息。校友能够通过板报了解最近的校友录的最新动态，包括日志、管理员信息以及一些周边八卦等。在设计数据库时，并没有为板报功能专门设计数据库信息，所以板报可以通过 **js** 进行实现。在板报功能制作直线，首先需要知道板报功能是如何工作的，板报功能的实现需要两个文件，这两个文件分别为 **txt** 文件和以 **.aspx** 为结尾的 **js** 文件，这两个文件的作用如下所示。

❑ **txt** 文件：用于存放数据，显示板报内容。

❑ **js** 文件：用于调用 **txt** 文件中的板报数据。

**txt** 文件用于存放数据。在板报功能模块中，板报主要用于存放字符串数据，而且字符串数据通常只是若干字符而已，最多显示一些图片，所以板报只需要打开 **txt** 文件进行文件内容的增删即可。当板报管理页面加载时，首先需要加载 **txt** 文本文件的内容，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        try
        {
            StreamReader aw = File.OpenText(Server.MapPath("banbao.txt")); //打开文本文件
            TextBox1.Text = aw.ReadToEnd(); //读文本文件
            aw.Close(); //关闭文本对象
        }
        catch
        {
            TextBox1.Text = "公告文本文件读取错误";
        }
    }
}
```

在载入页面文件后，管理员能够在文本框中填写板报文本字段，在填写了之后，单击【保存公告】按钮就能够进行板报的发布，示例代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    StreamWriter sw1 = File.CreateText(Server.MapPath("banbao.txt")); //创建文本文件
    sw1.Write(TextBox1.Text); //输出文本内容
    sw1.Close(); //关闭输出对象
    Response.Redirect("manage.aspx"); //页面跳转
}
```

保存的板报数据会保存到文本文档“**banbao.txt**”中，当需要页面中读取“**banbao.txt**”文档的文本时，同样可以使用 **StreamReader** 类进行读取。另外，还可以使用 **JS** 进行文本文档中内容的读取，示例代码如下所示。



```
<%@ Page Language="C#"
AutoEventWireup="true" CodeBehind="banbao.aspx.cs" Inherits="_29_1.js.banbao" %>
document.write('<% Response.Write(str); %>');
```

上述代码输出了共有字符串 **str**，在页面逻辑代码实现中，可以从文本中读取字符串并赋值给 **str** 变量进行文本输出，示例代码如下所示。

```
public partial class banbao : System.Web.UI.Page
{
    public string str = ""; //声明共有变量以便输出
    protected void Page_Load(object sender, EventArgs e)
    {
        try
        {
            StreamReader aw = File.OpenText(Server.MapPath("../admin/banbao.txt")); //读取文本
            str = aw.ReadToEnd(); //读取文本
            aw.Close(); //关闭读取对象
        }
        catch
        {
            str = "暂时没有任何公告"; //抛出异常
        }
    }
}
```

上述代码声明了一个共有的字符串型变量 **str**，该共有变量能够在页面中直接进行输出。值得注意的是，由于该 **js** 文件保存在根目录的 **js** 文件夹下，所以读取文本的路径也应该随之改变。如果无法读取文本，但为了提高用户的体验度，系统就不应该输出异常信息，而应该直接输出“暂时没有任何公告”。在需要使用公告的页面可以通过 **js** 调用进行数据呈现，示例代码如下所示。

```
<div class="main_board_font">
    <script src="js/banbao.aspx" type="text/javascript"></script>
</div>
```

注意：在使用 **JavaScript** 形式呈现数据时，要过滤“”等符号，因为 **JavaScript** 无法显示某些关键字或特殊符号，如果字符串中包含了这些符号，则可能无法呈现字符串。

## 29.7.6 用户修改和删除实现

管理员能够在前台进行用户信息的访问和用户索引的查看，在后台的操作中，管理员还能够对用户信息进行删除。删除用户信息时，为了保证用户数据的约束性和完整性，还需要对用户评论和用户数据中的数据进行删除。当页面初次被载入时，首先需要从数据库中读取相应的数据，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        string str = "select * from register where id=" + Request.QueryString["id"] + ""; //执行查询
        SqlDataReader da = SQLHelper.SQLHelper.ExecReader(str); //填充适配器
        while (da.Read()) //读取数据
        {
            Label1.Text = da["username"].ToString(); //填充控件
            TextBox2.Text = da["password"].ToString(); //填充控件
            DropDownList1.Text = da["sex"].ToString(); //填充控件
            TextBox3.Text = da["picture"].ToString(); //填充控件
            TextBox4.Text = da["im"].ToString(); //填充控件
            TextBox5.Text = da["information"].ToString(); //填充控件
        }
    }
}
```

```
        TextBox6.Text = da["others"].ToString(); //填充控件
    }
}
```

上述代码当页面被载入时执行，首先会将数据库中的数据查询并呈现在用户控件中。当管理员进行用户信息的填写后，可以单击【修改】按钮进行数据更改，示例代码如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    if (String.IsNullOrEmpty(TextBox2.Text)) //判断密码
    {
        string str = "update register set sex=" + DropDownList1.Text + ",picture=" + TextBox3.Text +
            ",im=" + TextBox4.Text + ",information=" + TextBox5.Text + ",others=" + TextBox6.Text + "
            where id=" + Request.QueryString["id"] + """; //生成 SQL 语句
        SQLHelper.SQLHelper.ExecNonQuery(str); //执行 SQL 语句
    }
    else
    {
        string str = "update register set password=" + TextBox2.Text + ",sex=" + DropDownList1.Text +
            ",picture=" + TextBox3.Text + ",im=" + TextBox4.Text + ",information=" + TextBox5.Text +
            ",others=" + TextBox6.Text + " where id=" + Request.QueryString["id"] + """; //生成 SQL 语句
        SQLHelper.SQLHelper.ExecNonQuery(str); //执行 SQL 语句
    }
}
```

在修改用户信息时，管理员可以填写用户的密码进行用户密码的更改，如果管理员不填写用户密码，那么在执行更新时不会更新用户的密码。管理员管理用户时，对于长期不上线的用户可以进行删除操作，删除用户信息还需要删除与用户相关的所有数据。在校友录系统中，与注册用户信息相关的数据包括评论数据和日志数据，在执行用户信息删除前首先要删除这些数据，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    string strsql1 = "delete from diarygbook where userid=" + Request.QueryString["uid"] + """;
    string strsql2 = "delete from diary where userid=" + Request.QueryString["uid"] + """;
    string strsql3 = "delete from register where id=" + Request.QueryString["uid"] + """;
    SQLHelper.SQLHelper.ExecNonQuery(strsql1); //删除日志评论
    SQLHelper.SQLHelper.ExecNonQuery(strsql2); //删除日志信息
    SQLHelper.SQLHelper.ExecNonQuery(strsql3); //删除用户信息
}
```

上述代码首先删除日志评论以保证日志数据的约束性和完整性，然后再删除日志以保证用户信息的约束性和完整性，当删除了以上数据后，最后删除用户信息。上述代码分别进行数据的删除，在执行删除时，还可以通过编写复杂的删除 SQL 语句进行数据的删除，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    string strsql1 = "delete from diarygbook,diary,register where diarygbook.userid=diary.userid and
        diarygbook.userid=register.id and diarygbook.userid=" + Request.QueryString["uid"] + """;
    //上述代码进行复杂的 SQL 语句删除多个表
    SQLHelper.SQLHelper.ExecNonQuery(strsql1); //执行数据删除
}
```

当运行上述代码时，系统会删除用户所有相关的信息并保证了用户数据的约束性和完整性。系统管理页面如图 29-31 所示。

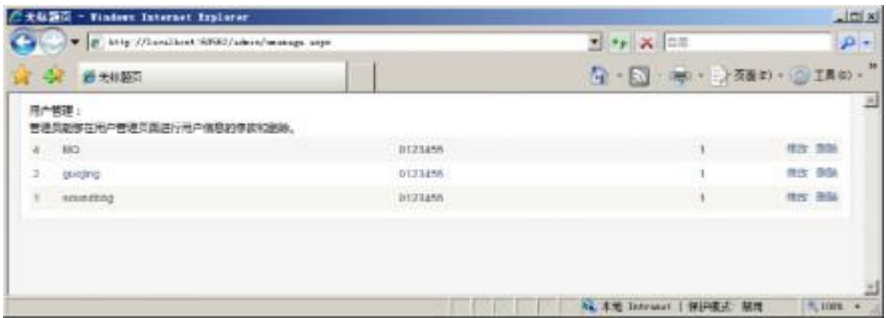


图 29-31 用户管理界面

29.7.7 用户权限管理

在数据库设计时，用户的权限通过 **userroot** 字段进行描述的。如果用户的 **userroot** 字段值为 **0** 时，那么这个用户就是一个普通的校友用户，如果 **userroot** 字段的值为 **1** 时，则用户会在系统中被判断为管理员。用户权限的管理就是用户信息的管理，如果需要将用户权限进行修改，可以直接使用用户修改功能进行实现。在修改模块中添加一个 **DropDownList** 控件用于权限的选择，示例代码如下所示。

```
<asp:DropDownList ID="DropDownList2" runat="server">
    <asp:ListItem Value="0">普通用户</asp:ListItem>
    <asp:ListItem Value="1">管理员</asp:ListItem>
</asp:DropDownList>
```

在用户权限管理中，还需要在页面加载时载入数据，示例代码如下所示。

```
DropDownList2.Text = da["userroot"].ToString();
```

在执行更新时，还需要将用户权限进行更新，示例代码如下所示。

```
if (String.IsNullOrEmpty(TextBox2.Text))
{
    string str = "update register set sex='" + DropDownList1.Text + "',picture='" + TextBox3.Text +
        "',im='" + TextBox4.Text + "',information='" + TextBox5.Text + "',others='" + TextBox6.Text +
        "',userroot='" + DropDownList2.Text + "' where id='" + Request.QueryString["id"] + "'"; //更新
    SQLHelper.SQLHelper.ExecNonQuery(str); //执行 SQL
}
else
{
    string str = "update register set password='" + TextBox2.Text + "',sex='" + DropDownList1.Text
        + "',picture='" + TextBox3.Text + "',im='" + TextBox4.Text + "',information='" + TextBox5.Text +
        "',others='" + TextBox6.Text + "',userroot='" + DropDownList2.Text + "' where id='" +
        Request.QueryString["id"] + "'"; //更新 SQL 语句
    SQLHelper.SQLHelper.ExecNonQuery(str); //执行 SQL 语句
}
```

当更改了 **SQL** 语句后，用户信息的更新也会更新用户权限。如果管理员希望升级某个校友用户的身份，管理员可以使用下拉菜单控件进行身份的选择，如图 29-32 所示。



图 29-32 用户权限管理

29.7.8 权限及注销实现

在校友录系统登录中，无论是校友用户还是管理员都是从前台登录，并且校友用户和管理员都是使用同一个表，惟一能够区分校友用户和管理员的就是 **userroot** 字段。在后台登录中，可以使用标识 **userroot** 的 **Session** 对象进行权限的判断，示例代码如下所示。

```
if (Session["admin"] == null)                                //判断是否登录
{
    Response.Redirect("../login.aspx");                       //未登录跳转
    if (Session["admin"].ToString() != "1")                   //判断是否为管理员
    {
        Response.Redirect("../friends.aspx");                 //非管理员跳转
    }
}
```

当管理员在前台登录后，管理员就能够管理员面板进入后台。当管理员完成管理并离开系统时，可以选择【退出管理】选项进行注销操作，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    Session["username"] = null;                                //清空用户名信息
    Session["userid"] = null;                                  //清空用户 ID 信息
    Session["admin"] = null;                                    //清空管理员信息
}
```

注意：在后台管理中，所有需要且只需要管理员操作的页面都需要进行页面权限判断。

29.8 实例演示

在编码完成后还需要对现有的项目进行测试，测试时需要准备数据源并进行数据操作，同时还需要进行程序中的页面测试，对错误的布局和逻辑进行修正。在数据源的准备中，需要考虑到所有的应用场景进行相应的数据插入，而在对页面进行测试时，不仅需要测试页面的显示，同样还需要对页面逻辑进行测试。



29.8.1 准备数据源

在 ASP.NET 校友录系统中，现有的代码并没有为校友录中日志的分类进行添加、删除管理等操作，可以使用 SQL 语句进行数据库中的数据添加，以便用户在校友录中添加日志时选择分类，添加分类 SQL 语句如下所示。

```
insert into diaryclass (classname) values ('生活日记')
insert into diaryclass (classname) values ('青青校园')
insert into diaryclass (classname) values ('天下驴友')
insert into diaryclass (classname) values ('社会时事')
```

上述代码在数据库中添加了 4 个分类，当用户进行日志发布时可以选择相应分类进行归类。

注意：由于篇幅限制，校友录系统并没有为日志分类的添加和删除进行开发，校友录系统的分类的系统开发在前面的新闻模块章节中都有所涉及。

29.8.2 实例演示

当用户进行 friend.aspx 页面访问时，系统会判断用户是否登录。如果用户没有登录，则会跳转到 login.aspx 登录页面进行登录，如果用户没有账户则需要进行注册。用户进行注册后就会跳转到登录页面，用户必须要进行登录才能够进行校友录系统的访问，如图 29-33 和图 29-34 所示。



图 29-33 用户注册页面



图 29-34 用户登录页面

当用户登录完成后，系统会跳转到校友录首页面，这里在校友录系统中还没有任何的日志，系统会提示用户发布日志，在侧边栏中会显示用户列表，用户列表控件的实现就是通过加入校友控件实现的，如图 29-35 所示。

当有多个用户注册时，侧边的用户列表就会显示多个用户，校友能够点击相应的用户进入用户索引。当用户进入 new.aspx 时，可以发布日志，如图 29-36 所示。



图 29-35 加入校友控件

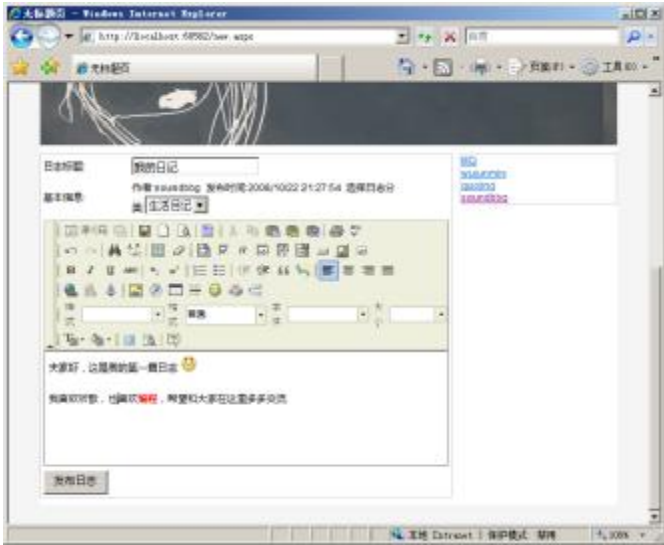


图 29-36 添加日志

发布日志时可以进行文本编辑进行富文本编写，用户还能够添加表情、进行编写检查等操作，这样就能够提高日志的可读性，丰满了日志。查看日志页面如图 29-37 所示。

当用户访问相应的日志时，还能够对日志进行评论，评论的内容会随着相应新闻而呈现在页面中，对于不同的新闻而言所呈现的留言内容也不同，如图 29-38 所示。



图 29-37 浏览日志



图 29-38 日志评论

对于感兴趣的用户，如图 29-38 中的 wujunmin，可以单击其用户名跳转到相应的索引页面，如果访问的用户为管理员，可以进行删除操作。管理员还能够在日志或校友录主界面进行相应的用户的访问，如图 29-39 所示。



图 29-39 删除用户操作

管理员能够单击相应的用户跳转到用户界面并对用户进行删除操作，删除用户后会删除该用户的所有日志和留言，并删除用户相关的信息。管理员还能够修改和删除日志，删除日志后，与日志相应的评论也会被删除。

注意：无论是在系统开发还是系统测试，都需要遵守数据的完整性，否则可能造成大量的垃圾数据。

29.8.3 管理后台演示

管理员在前台登录后，可以通过用户面板进行后台的访问，如图 29-40 所示。



图 29-40 用户面板

单击红色的【管理员】超链接字样，系统将会跳转到后台页面。在后台界面，管理员能够进行日志管理、评论管理以及用户管理，如图 29-41 所示。

单击【日志管理】跳转到日志管理，管理员能够在日志管理页面进行日志的修改和删除。在修改日志时，管理员能够修改前台用户不能修改的字段，如图 29-42 所示。



图 29-41 管理侧边栏

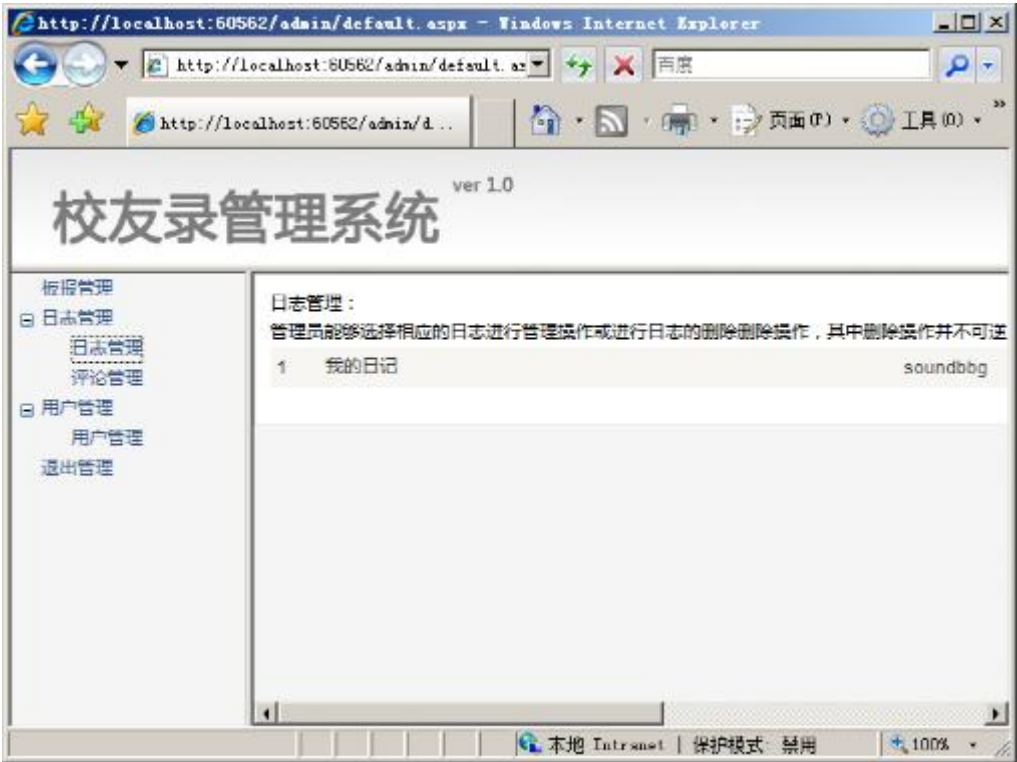


图 29-42 日志管理

管理员能够在日志管理中进行日志的修改，如图 29-43 所示，修改完成后，管理员可以单击【修改】按钮进行数据更新。



图 29-43 日志修改

单击【评论管理】超链接能够对用户评论进行管理。当前台校友发布了不良的信息时，可以通过【评论管理】选项管理评论并删除相应的评论，如图 29-44 所示。

单击【用户管理】超链接能够对用户进行管理。如果一个用户是非法用户或者用户并不是校友录中的相关用户（如这个校友录是给某个班级做的，而这个用户又不是这个班级的人），那么管理员就能够在【用户管理】选项中进行用户的删除操作。管理员还能够提升一个用户为管理员或将一个管理员降级为普通用户，如图 29-45 所示。

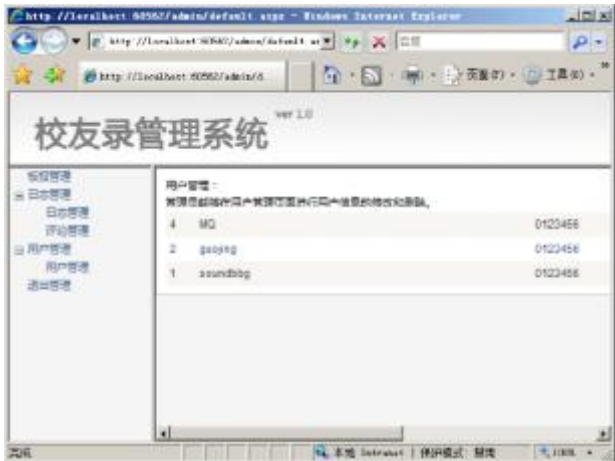


图 29-44 用户管理



图 29-45 用户权限管理

当管理员操作完毕后，可以单击【退出管理】超链接进行注销操作，注销完成后，系统会跳出后台并提示管理员进行登录操作，如图 29-46 所示。





图 29-46 注销后台

注销后台后，管理员在前台的用户信息也会被注销，如果管理员希望在前台登录或者继续在后台进行管理，管理员还需要进行登录操作。

## 29.9 小结

本章通过开发 **ASP.NET** 校友录系统进行系统开发讲解，这其中包括了系统设计、模块划分、文档编写和数据设计等，由于篇幅的限制，在 **ASP.NET** 校友录系统中还有一些功能没有实现，但是这些功能在前面的模块章节中已经实现，对于开发人员而言已经不是很难的问题。

对于系统开发而言，其过程是非常复杂的，不仅从一开始的系统设计还是从开发中的界面设计和编码实现，都是非常复杂的过程，对于初学者而言可能是一个页面的代码实现，而在系统的开发中是将系统模块化，进行分层开发，这就对开发人员有了更高的要求。

本章不仅从系统规划入手，着手基本的分层开发，使用 **SQLHelper** 类库和自定义控件都能够方便维护中维护成本的降低，本章不仅包括 **ASP.NET** 校友录系统的开发，还包括开源 **HTML** 编辑器的使用。本章还包括：

- ❑ 数据表关系图：绘制了数据库关系图，保障数据库中的约束条件。
- ❑ 使用 **Fckeditor**：讲解了如何使用 **Fckeditor** 进行富文本编程。
- ❑ 校友录页面规划：讲解了在开发前如何对页面进行页面规划。
- ❑ 校友录页面实现：讲解了校友录页面的实现和自定义控件的实现。
- ❑ 日志发布实现：讲解了如何使用 **Fckeditor** 实现日志发布。
- ❑ 日志显示页面：讲解了如何进行多表查询进行日志显示。

校友录系统使用了前面模块章节中讲到的模块，这些模块包括注册模块、登录模块和新闻模块，将这些模块进行整合就能够开发出复杂的系统。但是在模块整合的过程中同样会遇到很多问题，这些问题还需要开发人员进行二次开发和完善。