

ASP.NET MVC 5 框架揭秘

◎蒋金楠 著



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

ASP.NET MVC 5

框架揭秘

◎蒋金楠 著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书以一个模拟 ASP.NET MVC 内部运行机制的“迷你版 MVC 框架”作为开篇，其目的在于将 ASP.NET MVC 真实架构的“全景”勾勒出来。接下来本书以请求消息在 ASP.NET MVC 框架内部的流向为主线将相关的知识点串联起来，力求将“黑盒式”的消息处理管道清晰透明地展示在读者面前。相信精读本书的读者一定能够将 ASP.NET MVC 从接收请求到响应回复的整个流程了然于胸，对包括路由、Controller 的激活、Model 元数据的解析、Action 方法的选择与执行、参数的绑定与验证、过滤器的执行及 View 的呈现等相关机制具有深刻的理解。

本书以实例演示的方式介绍了很多与 ASP.NET MVC 相关的最佳实践，同时还提供了一系列实用性的扩展，相信它们一定能够解决你在真实开发过程中遇到的很多问题。本书最后一章提供的案例不仅用于演示实践中的 ASP.NET MVC，很多架构设计方面的东西也包含其中。除此之外，本书在很多章节还从设计的角度对 ASP.NET MVC 的架构进行了深入分析，所以从某种意义上讲本书可以当成一本架构设计的书来读。

虽然与市面上任何一本相关的书相比，本书走得更远，并更加近距离地触及 ASP.NET MVC 框架的内核，但是就其内容本身来讲却没有涉及太多“高深莫测”的知识点，所以阅读本书不存在太高的门槛。如果你觉得自己对 ASP.NET MVC 所知甚少，可以利用此书来系统地学习 ASP.NET MVC；如果你觉得自己对 ASP.NET MVC 足够精通，一定能够在此书中找到相应的“盲点”。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目（CIP）数据

ASP.NET MVC 5 框架揭秘 / 蒋金楠著. —北京：电子工业出版社，2014.8

ISBN 978-7-121-23781-2

I. ①A… II. ①蒋… III. ①网页制作工具—程序设计 IV. ①TP393.092

中国版本图书馆 CIP 数据核字（2014）第 150027 号

策划编辑：张春雨

责任编辑：徐津平

特约编辑：赵树刚

印 刷：北京市京科印刷有限公司

装 订：三河市皇庄路通装订厂

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本：787×980 1/16 印张：41 字数：1050 千字

版 次：2014 年 8 月第 1 版

印 次：2014 年 8 月第 1 次印刷

定 价：99.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zltz@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

前言

ASP.NET MVC 是一个建立在 ASP.NET 平台上基于 MVC 模式的 Web 开发框架，它提供了一种与 Web Form 完全不同的开发方式。ASP.NET Web Form 借鉴了 Windows Form 基于控件和事件注册的编程模式，使 Web 应用的开发变得简单而快捷，但是它却使开发人员与 Web 的本质渐行渐远。ASP.NET MVC 是一种回归，它使开发人员可以真正地面向 Web 进行编程，我们面对的不再是拖拉到 Web 页面的控件，而是整个 HTTP 请求和响应的流程。

这不是一本传统意义上的入门书籍

在《ASP.NET MVC 4 框架揭秘》出版之后，读者通过多种渠道将他们对本书的看法、建议和意见反馈给笔者。有一些读者觉得本书对他们来说太“深”了，因为他们希望买到的是一本单纯的入门书籍，所以我不得不再次强调——“这不是一本传统意义上的入门书籍”。如果你之前根本没有接触过 ASP.NET MVC，并且希望有一本书能够让你在一两个星期内掌握 ASP.NET MVC 的基本编程模式，那么笔者觉得选择本书并不是明智的。

笔者个人认为掌握 ASP.NET MVC 具有 3 个层次。了解基本的编程模式，掌握 Controller 和 View 的定义方式，知道路由如何注册及验证规则如何定义，此为第一层次。第二层次要求我们对 ASP.NET MVC 框架本身从请求接收到响应回复的整个流程有一个清晰的认识，这包括请求如何被路由、目标 Controller 如何被激活、Model 元数据如何被解析、Action 方法如何被执行、View 如何呈现等。ASP.NET MVC 本身是一个极具可扩展的开发框架，合理利用其扩展性可以解决很多开发中的实际问题，而掌握 ASP.NET MVC 的最高层次就是凭着对框架本身的运行机制的了解准确地找到相应的扩展点，并创建相应的扩展来解决我们遇到的问题。

本书不是一本 ASP.NET MVC 入门书籍，而是一本让处于第一层次的读者快速进入第二和第三层次的书。如果你具有进入更高层次的需求，并且有充裕的时间来阅读，笔者坚信你一定能够通过此书获得其他同类书籍难以提供的东西。

不少人觉得本书过于“深入”，但笔者个人宁愿说本书“走得更远”。虽然本书比其他同类书籍更加近距离地触摸到了 ASP.NET MVC 框架的内核，但就其内容本身来说，笔者认为本书并没有什

么“深不可测”的东西。尽管本书不是一本纯粹意义上的入门书籍，但是笔者绝不会根据读者个人的技术水平将其排除在本书的读者范围之外。如果读者具有基本的.NET 方面的知识储备，并且从事过简单的 Web 开发，阅读本书不会有太大的障碍。

这是一本讲述 ASP.NET MVC 框架本质的书

很多.NET 开发人员都在抱怨微软开发技术过快的更新频率让他们无所适从，其实他们看到的只是单纯的版本升级而已，一些本质的东西一直是“稳定”的。微软推出.NET 战略已经十多年了，CLR 却只有 4 个版本而已。最新版本的 ASP.NET 虽然表面上已经看不到太多最初的影子，但是整个请求处理的管道一直未曾改变。对于一项开发技术，只要我們了解了它最根本性的一些东西，不但不应该惧怕其高频率的版本更替，而应该热烈拥抱它。本书力求将关于 ASP.NET MVC 框架最根本的东西带给大家，而不是罗列一些简单的编程技巧。

这本书与《ASP.NET MVC 4 框架揭秘》有何不同

本书针对最新版本的 ASP.NET MVC，所以 ASP.NET MVC 5 提供的新特性和对 ASP.NET MVC 4 现有特性的改变自然体现在本书之中。通过微软“官方”渠道发布的信息我们知道，“特性路由”和“认证过滤器”是 ASP.NET MVC 5 提供的两个新特性，我们在本书中会通过单独的章节（第 13 章“特性路由”）来对特性路由作详细讲解，针对认证过滤器的介绍则放到新增加的一章（第 12 章“过滤器”）中。

除了这些通过官方渠道了解到的改变之外，其实微软在 ASP.NET MVC 5 框架内部作了很多的改进，这些东西只有当我们深入去分析其源码时才有可能发现。这些“不为人知”的内容也包含在本书之中，如果你手中正好有一本《ASP.NET MVC 4 框架揭秘》，在阅读的时候对照一下就知道了。

本书除了反映 ASP.NET MVC 5 本身的改变之外，还对前版一些遗漏掉的知识点进行了补充。为了让读者更好地理解本书的内容并尽可能地迎合大多数人的思维方式，我们在写作过程中对整本书的结构作了调整，同时对一些表达方式作了相应的改变。

和大部分 ASP.NET MVC 的书籍一样，《ASP.NET MVC 4 框架揭秘》利用一个单独的章节来介绍 Web API，这种做法实际上给很多读者造成了一种误解，让他们觉得 ASP.NET Web API 仅仅是

ASP.NET MVC 的附属产品。实际上两者不但在运行上使用不同的消息处理管道，并且这两个管道在设计上也是截然不同的，所以本书中找不到任何关于 ASP.NET Web API 的内容。如果你希望深入系统地学习 ASP.NET Web API，可以考虑本书的姊妹篇《ASP.NET Web API 2 框架揭秘》。

这是一本实用的书吗？

可能有人觉得这本剖析 ASP.NET MVC 框架运行原理的书没有什么“实际”的意义，因为我们每天的日常工作就是编程，知道了 ASP.NET MVC 从请求接收到响应回复之间整个处理流程并不会对我们的工作造成实质性的改变，这是一种极端错误的想法。学习一种软件开发技术就像是练一门功夫，不仅要苦练攻敌招式，还得研习内功心法。编程模式是攻敌招式，框架背后的设计原理是内功心法，没有内功心法支撑的招式只能是花拳绣腿，而具有极高内功修为的高手却能“无招胜有招”。

除此之外，由于我们编写的程序最终是在 ASP.NET MVC 框架上运行的，程序的高效性决定了它是否能够最大限度地“迎合”框架的运行机制，所以了解 ASP.NET MVC 框架的运行原理有利于我们写出高质量的程序。

很多读者都问笔者为何有那么多时间和精力去深入学习多个不同的开发框架（比如 WCF、ASP.NET MVC 和 ASP.NET Web API 等），其实这和从事的工作有关。多年以来，笔者在公司所作的工作就是设计、开发和维护一套内部的开发框架，这个框架的根本目的在于让我们可以采用一种类似于流水线的项目开发方式，不仅可以提高开发效率，还能提高项目本身的质量并降低对最终开发人员的技能要求，实现的途径就是让最终的开发人员只关心具体业务功能的实现，所有非业务功能都由框架本身来完成。

为了完成这个使命，我们针对 .NET 平台上的开发技术作了很多扩展。为了能够精准地定位采用的扩展点，我们不得不对开发技术本身有一个深刻的认识，所以深入学习 .NET 平台主流开发技术是笔者的本质工作之一。为了将这些实用的扩展服务于大众，笔者将这些扩展放到了相关的书籍之中。笔者自然不会将我们做的东西直接共享出来（因为这是违法的），所以书中涉及的这些扩展都经过刻意简化，因为笔者希望的不是让读者直接使用这些扩展，而是根据其体现出来的原理来设计你们需要的扩展。

ASP.NET MVC 之外的一些东西

在笔者的周围存着这样的一些人，他们以刚毕业一两年的毕业生为主。他们大都工作勤奋、聪明好学，手中经常捧着 GoF 的《设计模式》，总是希望在将书中的设计模式应用到具体项目之中，或者希望通过项目的实践来印证他们在书本上的设计模式，但是理论和实践之间的距离总让他们感到困惑。

要从真实的项目或者产品中学习“实用”的软件架构设计知识，先得确定目标项目或者产品中采用的架构思想和设计模式是正确的，而我们参与的很多项目其实被“架构”得一塌糊涂。对于像 ASP.NET 这样的产品，其基础架构能够在很长一段时间内保持不变，本身就证明了应用在上面的架构设计的正确性，它们不正是我们学习架构设计最好的素材吗？本书对 ASP.NET MVC 框架的运行机制进行了深入剖析，实际上是将 ASP.NET MVC 的整个设计展示在读者面前，读者也许可以将本书作为一本“架构设计”的书来读。

很多读者向笔者咨询针对一种新的开发技术，如何才能更加有效地掌握其“精髓”。这是一个关于学习方法的问题，笔者个人采用的学习方法不可能适用于所有的人。尽管如此，本书也或多或少地体现了笔者个人采用的学习方法，所以在论述某个知识点的时候，不但会告诉读者“是什么”和“为什么”，还会告诉读者“如何证明是这样”。换句话说，笔者不仅仅将某个论点抛给你，还会为你展现整个论证的过程。

本书的写作特点

和其他同类书籍作者总是从“静态编程”作为切入点不同，笔者以“动态执行”的视角来审视 ASP.NET MVC 框架。本书以框架本身处理请求的流程为主线，力求将 ASP.NET MVC 框架从接收请求到回复响应的整个流程完整而清晰地展现在读者面前。在本书的第 1 章中，我们设计了一个“迷你版”的 ASP.NET MVC 框架，其目的在于让读者能够对 ASP.NET MVC 的执行管道有一个整体的认识，至于组成这个管道的每个环节，则通过后续的章节对其架构设计进行详细介绍，对其执行原理进行深入剖析。

本书具有一个与其他中文原创或者翻译书籍截然不同的特点，那就是几乎所有的术语都采用英文，比如 Controller、Action 和 Model 在本书中并没有翻译成中文“控制器”、“操作”和“模型”。

因为笔者认为很多术语其实很难找到一个语义完全等同的中文词组或短语与之对应，对于习惯了英文作为“开发语言”的读者来说，强行翻译其实是不必要的。

除此之外，这不是一本纯理论的书，而是一本“实证型”的书，在书中提供了一百多个可供单独下载的实例演示。这些实例在本书中具有不同的作用，有的是为了探测和证明对应的论点，有的是为了演示某种实用的编程技巧。

关于作者

蒋金楠（网名 Artech），《WCF 全面解析（上、下册）》、《ASP.NET MVC 4 框架揭秘》、《ASP.NET Web API2 框架揭秘》等多部畅销 IT 图书作者，现就职于一知名软件公司担任高级软件顾问。拥有一个访问量超过 200 万的个人博客（<http://www.cnblogs.com/artech>），2012 年度 51CTO IT 博客大赛十佳得主。2007—2013 年被连续 7 次评为微软 MVP，同时也是少数几个跨多领域（Solutions Architect、Connected System、Microsoft Integration 和 ASP.NET/IIS）的 MVP 之一。

致谢

本书得以出版，需要感谢博文视点的编辑，你们的专业水准和责任心是为本书提供的质量保证，多次合作产生的默契让我对下次合作充满期待。此外，最需要感谢的是我的老婆徐妍妍，只有我知道你在本书提交给出版社之前所作的校对工作有多么重要。

本书支持

本书针对最新版本的 ASP.NET MVC，同时涉及太多底层实现的内容，所以大部分内容是找不到任何现成参考资料的，这些内容大都来自于笔者对源码的分析和试验的证明。这些因素加上作者自身能力的限制，都可能造成一些无法避免的错误或者偏差，如果读者在阅读过程中发现了任何问题，希望能够加以反馈。如果读者遇到任何 ASP.NET MVC、ASP.NET Web API 或者 WCF 的问题，欢迎与我通过以下的方式进行交流。

VIII ■ 前 言

- 作者博客: <http://www.cnblogs.com/artech>
- 作者微博: <http://www.weibo.com/artech>
- 电子邮箱: jiangjinnan@gmail.com
- 微信公众账号: Artech1981

本书每一章节都会提供一系列实例演示,读者可以根据编号(比如 S101、S202 等)从下载的源代码压缩包中找到对应的实例。本书的附录部分给出了所有源代码可供下载的所有实例演示的列表和相关描述。

源代码下载地址: <https://onedrive.live.com/redirect?resid=5760EBEEB92818D2%21109>

目 录

第 1 章 ASP.NET + MVC	1
1.1 传统 MVC 模式	2
1.1.1 自治视图	2
1.1.2 什么是 MVC 模式	3
1.2 MVC 的变体	4
1.2.1 MVP	5
1.2.2 Model 2	13
1.2.3 ASP.NET MVC 与 Model 2	15
1.3 IIS/ASP.NET 管道	15
1.3.1 IIS 5.x 与 ASP.NET	16
1.3.2 IIS 6.0 与 ASP.NET	17
1.3.3 IIS 7.0 与 ASP.NET	19
1.3.4 ASP.NET 集成	20
1.3.5 ASP.NET 管道	22
1.4 ASP.NET MVC 是如何运行的	28
1.4.1 建立在“迷你版”ASP.NET MVC 上的 Web 应用	28
1.4.2 路由	31
1.4.3 Controller 的激活	37
1.4.4 Action 的执行	41
1.4.5 完整的流程	49
第 2 章 路由	51
2.1 ASP.NET 路由	52
2.1.1 请求 URL 与物理文件的分离	52
2.1.2 实例演示：通过路由实现请求地址与.aspx 页面的映射（S201）	53
2.1.3 Route 与 RouteTable	57

2.1.4	路由注册	65
2.1.5	根据路由规则生成 URL	77
2.2	ASP.NET MVC 路由	79
2.2.1	路由映射	79
2.2.2	路由注册 (S210)	80
2.2.3	缺省 URL 参数	83
2.2.4	基于 Area 的路由映射	85
2.2.5	链接和 URL 的生成	91
2.3	动态 HttpHandler 映射	98
2.3.1	UrlRoutingModule	99
2.3.2	PageRouteHandler 与 MvcRouteHandler	100
2.3.3	ASP.NET 路由系统扩展	101
2.3.4	实例演示：通过自定义 Route 对 ASP.NET 路由系统进行扩展 (S214)	102
第 3 章	Controller 的激活	107
3.1	Controller 激活系统全景展示	108
3.1.1	Controller	108
3.1.2	ControllerFactory	114
3.1.3	ControllerBuilder	115
3.1.4	Controller 的激活与路由	122
3.2	Controller 默认激活机制	125
3.2.1	Controller 类型的解析	125
3.2.2	Controller 类型的缓存	130
3.2.3	Controller 的释放和会话状态行为的控制	131
3.3	IoC 的应用	133
3.3.1	从 Unity 来认识 IoC	133
3.3.2	Controller 与 Model 的解耦	135
3.3.3	基于 IoC 的 ControllerFactory	137
3.3.4	基于 IoC 的 ControllerActivator	143
3.3.5	基于 IoC 的 DependencyResolver	145
第 4 章	Model 元数据的解析	149
4.1	Model 元数据	150

4.1.1	Model 元数据层次化结构	150
4.1.2	Model 元数据的定制	154
4.1.3	IMetadataAware 接口	171
4.2	模板化数据的呈现	176
4.2.1	实例演示：通过模板将布尔值显示为 RadioButton (S409)	176
4.2.2	预定义模板	178
4.2.3	针对数据类型的模板	185
4.2.4	数据类型名称 V.S. 模板名称	189
4.2.5	模板的获取与执行	193
4.2.6	实例演示：通过定制 Model 元数据和自定义模板 实现预定义列表的 呈现 (S412)	197
4.3	Model 元数据的提供机制	205
4.3.1	再谈 ModelMetadata	206
4.3.2	ModelMetadataProvider	210
4.3.3	Model 元数据提供系统的扩展	214
第 5 章	3 个重要的描述对象	217
5.1	ControllerDescriptor	218
5.1.1	ReflectedControllerDescriptor	219
5.1.2	ReflectedAsyncControllerDescriptor	228
5.2	ActionDescriptor	229
5.2.1	AsyncActionDescriptor	230
5.2.2	ReflectedActionDescriptor	231
5.2.3	ReflectedAsyncActionDescriptor	232
5.2.4	TaskAsyncActionDescriptor	233
5.3	ParameterDescriptor	234
第 6 章	Model 的绑定 (一)	236
6.1	源数据的提供	237
6.1.1	NameValueCollectionValueProvider	238
6.1.2	DictionaryValueProvider	246
6.1.3	ValueProviderFactory	254
6.1.4	ValueProviderFactories	255
6.2	ModelBinder 及其提供策略	259

6.2.1	ModelBinder	259
6.2.2	ModelBinderProvider	265
6.2.3	ModelBinders.....	268
6.2.4	CustomModelBinderAttribute	271
6.2.5	针对参数的 ModelBinder 是如何创建的	274
6.3	Model 绑定的实施.....	276
6.3.1	绑定上下文的初始化.....	277
6.3.2	绑定过程中对 ModelState 的设置	279
第 7 章	Model 的绑定 (二)	283
7.1	绑定简单对象	284
7.1.1	利用 ValueProvider 绑定简单对象	284
7.1.2	实例演示: 利用 MyDefaultModelBinder 绑定简单类型参数 (S701)	286
7.2	绑定复杂对象	289
7.2.1	复杂对象层次化结构.....	289
7.2.2	递归式绑定.....	291
7.2.3	实例演示: 利用 MyDefaultModelBinder 绑定复杂类型参数 (S702、S703)	294
7.3	绑定集合	297
7.3.1	针对同名数据项的集合绑定.....	297
7.3.2	针对索引的集合绑定.....	302
7.4	绑定字典	311
7.4.1	字典是一个复杂类型的集合	311
7.4.2	针对字典类型的 Model 绑定策略	312
7.4.3	实例演示: 利用 MyDefaultModelBinder 绑定字典类型参数 (S707)	316
第 8 章	Model 的验证 (一)	318
8.1	几种参数验证方式	319
8.1.1	ModelError.....	319
8.1.2	验证消息的呈现.....	320
8.1.3	手工验证绑定的参数.....	322
8.1.4	使用 ValidationAttribute 特性.....	327
8.1.5	让数据类型实现 IValidatableObject 接口	330
8.1.6	让数据类型实现 IDataErrorInfo 接口	332

8.2	ModelValidator 及其提供策略	334
8.2.1	ModelValidator 与 ModelValidatorProvider	334
8.2.2	DataAnnotationsModelValidator	337
8.2.3	ValidatableObjectAdapter	337
8.2.4	DataErrorInfoModelValidator	338
8.2.5	ClientModelValidator	339
8.2.6	CompositeModelValidator	341
8.3	Model 验证的实施	345
8.3.1	Model 绑定过程中的验证	346
8.3.2	实例演示: 模拟 Model 绑定中的验证 (S810)	347
8.3.3	针对“必需”数据成员的验证	351
第 9 章	Model 的验证 (二)	354
9.1	ValidationAttribute 特性	355
9.1.1	数据是如何被验证的	356
9.1.2	几个常用的 ValidationAttribute	358
9.1.3	应用 ValidationAttribute 特性的唯一性	360
9.2	DataAnnotationsModelValidator 及其提供策略	364
9.2.1	“适配”型 DataAnnotationsModelValidator	365
9.2.2	DataAnnotationsModelValidatorProvider	368
9.2.3	将 ValidationAttribute 特性应用到参数上	375
9.2.4	一种 Model 类型, 多种验证规则	382
9.3	客户端验证	389
9.3.1	jQuery 验证	390
9.3.2	基于 jQuery 的 Model 验证	394
9.3.3	自定义验证	398
第 10 章	Action 方法的执行	402
10.1	异步 Action 的定义	403
10.1.1	基于线程池的请求处理机制	403
10.1.2	两种异步 Action 方法的定义	404
10.1.3	AsyncManager	406
10.2	各种同步与异步组件	412
10.2.1	MvcHandler	412

10.2.2	Controller	413
10.2.3	ActionInvoker	414
10.2.4	ControllerDescriptor	420
10.2.5	ActionDescriptor	423
10.3	目标方法的执行	430
10.3.1	Action 方法并不以“反射”方式执行	430
10.3.2	实例演示：采用针对表达式树执行 Action 方法（S1010）	432
第 11 章	View 的呈现	437
11.1	ActionResult	438
11.1.1	EmptyResult	438
11.1.2	ContentResult	439
11.1.3	FileResult	446
11.1.4	JavaScriptResult	451
11.1.5	JsonResult	455
11.1.6	HttpStatusCodeResult	457
11.1.7	RedirectResult/RedirectToRouteResult	458
11.2	ViewResult 与 ViewEngine	461
11.2.1	View 引擎中的 View	461
11.2.2	ViewEngine	463
11.2.3	ViewResult 的执行	465
11.3	Razor 引擎	474
11.3.1	View 的编译原理	475
11.3.2	WebViewPage 与 WebViewPage<TModel>	480
11.3.3	RazorView	485
11.3.4	RazorViewEngine	495
第 12 章	过滤器	499
12.1	Filter 及其提供机制	500
12.1.1	Filter 与 FilterProvider	500
12.1.2	以特性方式注册过滤器	502
12.1.3	Controller 本身就是过滤器	504
12.1.4	过滤器的全局注册	504

12.1.5 实例演示：验证 Filter 的提供机制和执行顺序 (S1201, S1202, S1203)	506
12.2 AuthenticationFilter	511
12.2.1 AuthenticationFilter 的执行流程	512
12.2.2 实例演示：通过自定义 AuthenticationFilter 实现 Basic 认证 (S1204)	513
12.3 AuthorizationFilter	518
12.3.1 AuthorizeAttribute	518
12.3.2 RequireHttpsAttribute	520
12.3.3 ValidateInputAttribute	520
12.3.4 ValidateAntiForgeryTokenAttribute	523
12.3.5 ChildActionOnlyAttribute	527
12.4 ActionFilter	528
12.4.1 ActionFilter 的执行流程	529
12.4.2 ActionFilter 对 ActionResult 的设置	530
12.4.3 异常处理	532
12.5 ExceptionFilter	534
12.5.1 HandleErrorAttribute	535
12.5.2 实例演示：利用自定义的 ExceptionFilter 集成 Enterprise Library 进行 异常处理 (S1207, S1208, S1209)	537
12.6 ResultFilter 与 OverrideFilter	551
12.6.1 ResultFilter 的执行流程	552
12.6.2 屏蔽外围过滤器	553
第 13 章 特性路由	556
13.1 特性路由注册	557
13.1.1 RouteInfoProvider 特性	557
13.1.2 基本路由映射	558
13.1.3 让路由模板能够尽可能反映资源的层次结构	559
13.1.4 为路由变量设置约束	560
13.1.5 缺省路由变量	561
13.1.6 设置模板前缀	562
13.1.7 设置 Area 名称	563
13.2 约束表达式的解析	564
13.2.1 RangeRouteConstraint	565

13.2.2	InlineConstraintResolver	566
13.2.3	自定义约束.....	570
13.3	Route 的创建.....	574
13.3.1	特性路由注册的 Route 对象	574
13.3.2	Route 的生成机制	579
13.3.3	Controller 的激活与 Action 方法的选择	579
第 14 章	案例实践	581
14.1	功能简介	582
14.1.1	商品列表的呈现.....	582
14.1.2	订购商品.....	584
14.1.3	登录与错误页面.....	585
14.2	设计概述	586
14.2.1	Controller-Service-Repository	586
14.2.2	IoC 的应用.....	591
14.2.3	AOP 的应用.....	594
14.2.4	异常处理.....	601
14.3	编程实现	602
14.3.1	数据表的创建.....	603
14.3.2	Repository	604
14.3.3	Service	609
14.3.4	路由注册和布局.....	612
14.3.5	ProductController.....	616
14.3.6	OrderController.....	624
14.3.7	AccountController.....	630
附录 A	实例列表	635

第 1 章 ASP.NET + MVC

ASP.NET MVC 是一个全新的 Web 应用框架。将术语 ASP.NET MVC 拆分开来,即 ASP.NET + MVC。前者代表支撑该应用框架的技术平台,它表明 ASP.NET MVC 和传统的 Web Forms 应用框架一样,都是建立在 ASP.NET 平台之上;后者则表示该框架背后的设计思想,意味着 ASP.NET MVC 采用了 MVC 架构模式。

1.1 传统 MVC 模式

对于大部分面向最终用户的应用来说，它们都需要具有一个与用户进行交互的可视化 UI 界面，我们将这个 UI 称为视图（View）。在早期，我们倾向于将所有与 UI 相关的操作糅合在一起，这些操作包括 UI 界面的呈现、用户交互操作的捕捉与响应、业务流程的执行及对数据的存取等，我们将这种设计模式称为自治视图（Autonomous View，AV）。

1.1.1 自治视图

说到自治视图，很多人会感到陌生，但是我们（尤其是 .NET 开发人员）可能经常采用这种模式来设计我们的应用。Windows Forms 和 ASP.NET Web Forms 虽然分别属于 GUI 和 Web 应用开发框架，但是它们都采用了事件驱动的开发方式，所有与 UI 相关的逻辑都可以定义在针对视图（Windows Form 或者 Web Form）的后台代码（Code Behind）中，并最终注册到视图本身或者视图元素（控件）的相应事件上。

一个典型的人机交互应用具有 3 个主要的关注点，即数据在可视化界面上的呈现、UI 处理逻辑（用于处理用户交互式操作的逻辑）和业务逻辑。自治视图模式将三者混合在一起，势必会带来如下一些问题。

- 重用性。业务逻辑是与 UI 无关的，应该最大限度地被重用，但是若将业务逻辑定义在自治视图中，相当于使它完全与视图本身绑定在一起。除此之外，如果我们能够将 UI 的行为抽象出来，基于抽象化 UI 的处理逻辑也是可以被共享的，但是定义在自治视图中的 UI 处理逻辑也完全丧失了重用的可能。
- 稳定性。业务逻辑具有最强的稳定性，UI 处理逻辑次之，可视化界面上的呈现最差（比如我们经常会为了更好地呈现效果来调整 HTML）。如果将具有不同稳定性的元素混合为一体，那么具有最差稳定性的元素决定了整体的稳定性，这是“短板理论”在软件设计中的体现。
- 可测试性。任何涉及 UI 的组件都不易测试，因为 UI 是呈现给人看的，并且会与人进行交互，用机器来模拟活生生的人对组件实施自动化测试本就不是一件容易的事。

为了解决自治视图导致的这些问题，我们需要采用关注点分离（Seperation of Concerns, SoC）的原则将可视化界面呈现、UI 处理逻辑和业务逻辑三者分离出来，并且采用合理的交互方式将它们之间的依赖降到最低。将三者“分而治之”，自然也使 UI 逻辑和业务逻辑变得更易于测试，测试驱动设计与开发也得以实现。这里用于进行关注点分离的模式就是 MVC。

1.1.2 什么是 MVC 模式

MVC 的创建者是 Trygve M. H. Reenskau, 他是挪威的计算机专家, 同时也是奥斯陆大学的名誉教授。MVC 是他在 1979 年访问施乐帕克研究中心 (Xerox Palo Alto Research Center, Xerox PARC) 期间提出的一种主要针对 GUI 应用的软件架构模式。Trygve 最初对 MVC 的描述记录在 *Applications Programming in Smalltalk-80(TM):How to use Model-View-Controller (MVC)* 这篇论文中, 有兴趣的读者可以通过地址 <http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html> 阅读这篇论文。

MVC 体现了“关注点分离”这一基本的设计方针, 它将一个人机交互应用涉及的功能分为 Model、Controller 和 View 三部分, 它们各自具有如下的职责。

- Model 是对应用状态和业务功能的封装, 我们可以将它理解为同时包含数据和行为的领域模型 (Domain Model)。Model 接受 Controller 的请求并完成相应的业务处理, 在应用状态改变的时候可以向 View 发出相应的通知。
- View 实现可视化界面的呈现并捕捉最终用户的交互操作 (如鼠标和键盘操作)。
- View 捕获到用户交互操作后会直接转发给 Controller, 后者完成相应的 UI 逻辑。如果需要涉及业务功能的调用, Controller 会直接调用 Model。在完成 UI 处理之后, Controller 会根据需要控制原 View 或者创建新的 View 对用户交互操作予以响应。

图 1-1 揭示了 MVC 模式下 Model、View 和 Controller 之间的交互。对于传统的 MVC 模式来说, 很多人会认为 Controller 仅仅是 View 和 Model 之间的中介。实则不然, View 和 Model 之间存在直接的联系, View 不仅可以直接调用 Model 查询其状态信息, 当 Model 的状态发生改变的时候, 它也可以直接通知 View。比如在一个提供股票实时价位的应用中, 维护股价信息的 Model 在股价变化的情况下可以直接通知相关的 View 改变其显示信息。

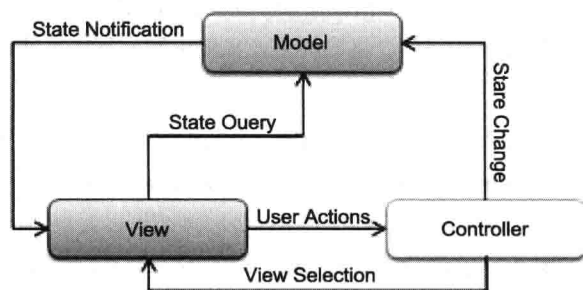


图 1-1 Model-View-Controller 之间的交互

从消息交换模式的角度来讲,不论是 Model 在应用状态发生改变时通知 View,还是 View 在捕捉到用户的交互操作后通知 Controller,消息都是以“单向(One-Way)”方式流动的,所以我们推荐采用事件机制来实现这两种类型的通知。从设计模式的角度来讲就是采用观察者(Observer)模式通过注册/订阅的方式来实现它们,具体来讲就是让 View 作为 Model 的观察者通过注册相应的事件来检测状态的改变,让 Controller 作为 View 的观察者通过注册相应的事件来处理用户的交互操作。

我们看到很多人将 MVC 和所谓的“三层架构”进行比较,其实两者并没有什么可比性。MVC 更不是分别对应着 UI、业务逻辑和数据存取 3 个层次,不过两者也不能说完全没有关系。Trygve M. H. Reenskau 提出 MVC 的时候是将其作为构建整个 GUI 应用的架构模式,这种情况下的 Model 实际上维护着整个应用的状态并实现了所有的业务逻辑,所以它更多地体现为一个领域模型。

对于多层架构来说(比如我们经常提及的三层架构),MVC 是被当成 UI 呈现层(Presentation Layer)的设计模式,而 Model 则更多地体现为访问业务层的入口(Gateway)。如果采用面向服务的设计,业务功能被定义成相应服务并通过接口(契约)的形式暴露出来,这里的 Model 还可以表示成进行服务调用的代理。

1.2 MVC 的变体

我们可以采用 MVC 模式将可视化 UI 元素的呈现、UI 处理逻辑和业务逻辑分别定义在 View、Controller 和 Model 中,但是 MVC 并没有对三者之间的交互进行严格的限制。这主要体现在它允许 View 和 Model 绕开 Controller 进行直接交互,不仅 View 可以通过调用 Model 获取需要呈现给用户的数据,Model 也可以直接通知 View 让其感知到应用状态的变化。当我们将 MVC 应用于具体的项目开发时,不论是基于 GUI 的桌面应用还是基于浏览器的 Web 应用,如果不对 Model、View 和 Controller 之间的交互作更为严格的约束,我们编写的程序可能比自治视图更加难以维护。

今天我们将 MVC 视为一种模式(Pattern),但是作为 MVC 最初提出者的 Trygve M. H. Reenskau 却将 MVC 视为一种范例(Paradigm),这可以从他在 *Applications Programming in Smalltalk-80(TM):How to use Model-View-Controller (MVC)* 中对 MVC 的描述可以看出来: *In the MVC paradigm the user input, the modeling of the external world, and the visual feedback to the user are explicitly separated and handled by three types of object, each specialized for its task.*

模式和范例的区别在于前者可以直接应用到具体的应用上,而后者则仅提供一些基本的

指导方针。在我看来，MVC 是一个很宽泛的概念，任何基于 Model、View 和 Controller 对 UI 应用进行分解的设计都可以称为 MVC。当我们采用 MVC 的思想来设计 UI 应用的时候，应该根据开发框架（比如 Windows Forms、WPF 和 Web Forms）的特点对 Model、View 和 Controller 设置一个明确的界限，同时为它们之间的交互制定一个更为严格的规则。

在软件设计的发展历程中出现了一些 MVC 的变体（Variation），它们遵循定义在 MVC 中的基本原则，但对于三元素之间的交互制定了更为严格的规范。我们现在就来简单地讨论几种常用的 MVC 变体。

1.2.1 MVP

MVP 是一种广泛使用的 UI 架构模式，适用于基于事件驱动的应用框架，比如 ASP.NET Web Forms 和 Windows Forms 应用。MVP 中的 M 和 V 分别对应于 MVC 的 Model 和 View，而 P（Presenter）则自然代替了 MVC 中的 Controller。但是 MVP 并非仅仅体现在从 Controller 到 Presenter 的转换，而是更多地体现在 Model、View 和 Presenter 之间的交互上。

MVC 模式中三元素之间“混乱”的交互主要体现在允许 View 和 Model 绕开 Controller 进行单独“交流”，这个问题在 MVP 模式中得到了彻底解决。如图 1-2 所示，能够与 Model 直接进行交互的仅限于 Presenter，View 只能通过 Presenter 间接地调用 Model。Model 的独立性在这里得到了真正的体现，它不仅仅与可视化元素的呈现（View）无关，与 UI 处理逻辑（Presenter）也无关。使用 MVP 的应用是用户驱动的而非 Model 驱动的，所以 Model 不需要主动通知 View 以提醒状态发生了改变。

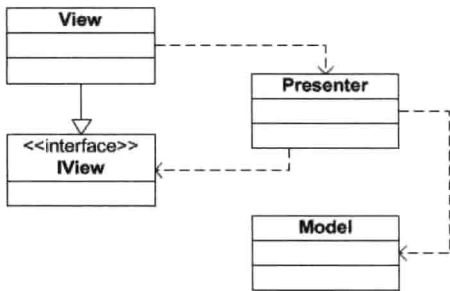


图 1-2 Model-View-Presenter 之间的交互

MVP 不仅仅避免了 View 和 Model 之间的深度耦合，更进一步地降低了 Presenter 对 View 的依赖。如图 1-2 所示，Presenter 依赖的是一个抽象化的 View，即具体 View 实现的接口 IView，这带来的最直接的好处就是使定义在 Presenter 中的 UI 处理逻辑变得易于测试。由于 Presenter

对 View 的依赖行为定义在接口 IView 中,我们只需要 Mock 一个实现了该接口的 View 就能对 Presenter 进行测试。

构成 MVP 三要素之间的交互体现在两个方面,即 View 与 Presenter 及 Presenter 与 Model 之间的交互。Presenter 和 Model 之间的交互很清晰,它仅仅体现为 Presenter 对 Model 的单向调用。View 和 Presenter 之间该采用怎样的交互方式是整个 MVP 的核心,MVP 针对关注点分离的初衷能否体现在具体的应用中,很大程度上取决于两者之间的交互方式是否正确。按照 View 和 Presenter 之间的交互方式,以及 View 本身的职责范围,Martin Folwer 将 MVP 分为 PV (Passive View) 和 SC (Supervising Controller) 两种模式。

1. PV 与 SC

解决 View 难以测试的最好办法就是让它无须测试。如果 View 不需要测试,其先决条件就是让它尽可能不涉及 UI 处理逻辑,这就是 PV 模式的目的所在。顾名思义,PV (Passive View) 是一个被动的 View,定义其中的针对 UI 元素(比如控件)的操作不是由 View 自身主动来控制,而是被动地交给 Presenter 来操控。

如果我们纯粹地采用 PV 模式来设计 View,意味着我们需要将 View 中的 UI 元素通过属性的形式暴露出来。具体来说,当我们在为 View 定义接口的时候,需要定义基于 UI 元素的属性使 Presenter 可以对 View 进行细粒度操作,但这并不意味着我们直接将 View 上的控件暴露出来。举个简单的例子,假设我们开发的 HR 系统中具有如图 1-3 所示的一个 Web 页面,我们通过它可以获取某个部门的员工列表。



图 1-3 员工查询页面

假设现在通过 ASP.NET Web Forms 应用来设计这个页面,我们来讨论一下如果采用 PV 模式 View 的接口该如何定义。对于 Presenter 来说,View 供它操作的控件有两个,一个是包含所有部门列表的 DropDownList,另一个则是显示员工列表的 GridView。在页面加载的时候,

Presenter 将部门列表绑定在 DropDownList 上,与此同时包含所有员工的列表被绑定到 GridView 上。当用户选择某个部门并单击“查询”按钮后,View 将包含筛选部门在内的查询请求转发给 Presenter,后者筛选出相应的员工列表之后将其绑定到 GridView。

如果为该 View 定义一个接口 IEmployeeView,我们不能按照如下所示的代码将上述这两个控件直接以属性的形式暴露出来。针对具体控件类型的数据绑定属于 View 的内部细节(比如说针对部门列表的显示,可以选择 DropDownList,也可以选择 ListBox),不能体现在表示用于抽象 View 的接口中。除此之外,理想情况下定义在 Presenter 中的 UI 处理逻辑应该是与具体的技术平台无关的,如果在接口中涉及控件类型,这无疑将 Presenter 与具体的技术平台绑定在了一起。

```
public interface IEmployeeView
{
    DropDownList      Departments { get; }
    GridView           Employees { get; }
}
```

正确的接口和实现该接口的 View(一个 Web 页面)应该采用如下的定义方式:Presenter 通过对属性 Departments 和 Employees 赋值来实现对相应 DropDownList 和 GridView 的数据绑定,同时通过属性 SelectedDepartment 得到用户选择的筛选部门。为了尽可能让接口只暴露必需的信息,我们还特意将对属性的读/写作了控制。

```
public interface IEmployeeView
{
    IEnumerable<string>      Departments { set; }
    string                  SelectedDepartment { get; }
    IEnumerable<Employee>    Employees { set; }
}

public partial class EmployeeView: Page, IEmployeeView
{
    //其他成员
    public IEnumerable<string> Departments
    {
        set
        {
            this.DropDownListDepartments.DataSource = value;
            this.DropDownListDepartments.DataBind();
        }
    }

    public string SelectedDepartment
    {
        get { return this.DropDownListDepartments.SelectedValue; }
    }
}
```

```
public IEnumerable<Employee> Employees
{
    set
    {
        this.GridViewEmployees.DataSource = value;
        this.GridViewEmployees.DataBind();
    }
}
```

PV 模式将所有的 UI 处理逻辑全部定义在 **Presenter** 上，意味着所有的 UI 处理逻辑都可以被测试，从可测试性的角度来看这是一种不错的选择。但是它要求将 **View** 中可供操作的 UI 元素定义在对应的接口中，对于一些复杂的富客户端（Rich Client）应用的 **View** 来说，接口成员的数量将可能会变得很多，这无疑会提升编程所需的代码量。从另一方面来看，由于 **Presenter** 需要在控件级别对 **View** 进行细粒度的控制，这无疑会提高 **Presenter** 本身的复杂度，往往会使原本简单的逻辑复杂化。在这种情况下我们往往采用 SC 模式。

在 SC 模式下，为了降低 **Presenter** 的复杂度，我们倾向于将诸如数据绑定和显示数据格式化这样简单的 UI 处理逻辑转移到 **View** 中，这些处理逻辑会体现在 **View** 实现的接口中。尽管 **View** 从 **Presenter** 中接管了部分 UI 处理逻辑，但是 **Presenter** 依然是整个三角关系的驱动者，**View** 被动的地位依然没有改变。对于用户作用在 **View** 上的交互操作，**View** 本身并不进行响应，它只会将交互请求转发给 **Presenter**，后者在独立完成相应的处理流程（可能涉及针对 **Model** 的调用）之后会驱动 **View** 对用户交互请求进行响应。

2. View 和 Presenter 交互的规则（针对 SC 模式）

View 和 **Presenter** 之间的交互是整个 MVP 的核心，能否正确地应用 MVP 模式来架构我们的应用，主要取决于能否正确地处理 **View** 和 **Presenter** 两者之间的关系。在由 **Model**、**View** 和 **Presenter** 组成的三角关系中，核心元素不是 **View** 而是 **Presenter**，**Presenter** 不是 **View** 调用 **Model** 的中介，而是最终决定如何响应用户交互行为的决策者。

View 可以理解为 **Presenter** 委派到前端的客户代理，而作为客户的自然就是最终的用户。对于体现为鼠标/键盘操作的交互请求应该如何处理，作为代理的 **View** 并没有决策权，所以它只能将请求汇报给委托人 **Presenter**。**View** 向 **Presenter** 发送用户交互请求应该采用这样的口吻：“我现在将用户交互请求发送给你，你看着办，需要我的时候我会协助你”，而不应该是这样：“我现在处理用户交互请求了，我知道该怎么办，但是我需要你的支持，因为实现业务逻辑的 **Model** 只信任你”。

对于 **Presenter** 处理用户交互请求的流程，如果中间环节需要涉及 **Model**，它会直接发起对

Model 的调用。如果需要 View 的参与(比如需要将 Model 最新的状态反映在 View 上),Presenter 会驱动 View 完成相应的工作。

对于绑定到 View 上的数据,不应该是 View 从 Presenter 上“拉”回来的,而是 Presenter 主动“推”给 View 的。从消息流(或者消息交换模式)的角度来讲,不论是 View 向 Presenter 发送用户交互请求的通知,还是 Presenter 驱动 View 来对用户交互操作予以响应,都是单向的。反映在应用编程接口的定义上就意味着不论是定义在 Presenter 中被 View 调用的方法,还是定义在 IView 接口中被 Presenter 调用的方法,最好都没有返回值。如果不采用方法调用的形式,我们也可以通过事件注册的方式实现 View 和 Presenter 的交互,事件机制体现的消息流无疑就是单向的。

View 本身仅仅实现了单纯的、独立的 UI 逻辑,它处理的数据应该是 Presenter 实时推送给它的,所以 View 尽可能不维护数据状态。定义在 IView 的接口最好只包含方法,而不包含属性。Presenter 所需的 View 状态应该在接收到 View 发送的用户交互请求的时候一次得到,而不需要通过 View 的属性去获取。

3. 实例演示: SC 模式的应用(S101)

为了让读者对 SC 模式下的 MVP,尤其是该模式下的 View 和 Presenter 之间的交互方式有一个深刻的认识,我们现在来做一个简单的实例演示。本实例采用上面提及的关于员工查询的场景,并且采用 ASP.NET Web Forms 来建立这个简单的应用。前面已经演示了采用 PV 模式下的 IView 应该如何定义,现在来看看 SC 模式下的 IView 有何不同。

先来看看表示员工信息的数据类型如何定义。我们通过具有如下定义的数据类型 Employee 来表示一个员工。简单起见,我们仅仅定义了表示员工基本信息(ID、姓名、性别、出生日期和部门)的 5 个属性。

```
public class Employee
{
    public string      Id { get; private set; }
    public string      Name { get; private set; }
    public string      Gender { get; private set; }
    public DateTime    BirthDate { get; private set; }
    public string      Department { get; private set; }

    public Employee(string id, string name, string gender,
        DateTime birthDate, string department)
    {
        this.Id          = id;
        this.Name         = name;
        this.Gender       = gender;
        this.BirthDate    = birthDate;
    }
}
```

```

        this.Department = department;
    }
}

```

作为包含应用状态和状态操作行为的 Model, 通过如下一个简单的 `EmployeeRepository` 类型来体现。如代码所示, 表示所有员工列表的数据通过一个静态字段来维护, 而 `GetEmployees` 方法返回指定部门的员工列表。如果没有指定筛选部门或者指定的部门字符为空, 该方法直接返回所有的员工列表。

```

public class EmployeeRepository
{
    private static IList<Employee> employees;

    static EmployeeRepository()
    {
        employees = new List<Employee>();
        employees.Add(new Employee("001", "张三", "男",
            new DateTime(1981, 8, 24), "销售部"));
        employees.Add(new Employee("002", "李四", "女",
            new DateTime(1982, 7, 10), "人事部"));
        employees.Add(new Employee("003", "王五", "男",
            new DateTime(1981, 9, 21), "人事部"));
    }

    public IEnumerable<Employee> GetEmployees(string department = "")
    {
        if (string.IsNullOrEmpty(department))
        {
            return employees;
        }
        return employees.Where(e => e.Department == department).ToArray();
    }
}

```

接下来我们来看看作为 View 接口的 `IEmployeeView` 的定义。如下面的代码片段所示, 该接口定义了 `BindEmployees` 和 `BindDepartments` 两个方法, 分别用于绑定基于部门列表的 `DropDownList` 和基于员工列表的 `GridView`。除此之外, `IEmployeeView` 接口还定义了一个事件 `DepartmentSelected`, 该事件会在用户选择了筛选部门后单击“查询”按钮时触发。`DepartmentSelected` 事件参数类型为自定义的 `DepartmentSelectedEventArgs`, 属性 `Department` 表示用户选择的部门。

```

public interface IEmployeeView
{
    void BindEmployees(IEnumerable<Employee> employees);
    void BindDepartments(IEnumerable<string> departments);
    event EventHandler<DepartmentSelectedEventArgs> DepartmentSelected;
}

```

```
public class DepartmentSelectedEventArgs : EventArgs
{
    public string Department { get; private set; }
    public DepartmentSelectedEventArgs(string department)
    {
        this.Department = department;
    }
}
```

作为 MVP 三角关系核心的 **Presenter** 通过 **EmployeePresenter** 表示。如下面的代码片段所示，表示 **View** 的只读属性类型为 **IEmployeeView** 接口，而另一个只读属性 **Repository** 则表示作为 **Model** 的 **EmployeeRepository** 对象，两个属性均在构造函数中初始化。

```
public class EmployeePresenter
{
    public IEmployeeView View { get; private set; }
    public EmployeeRepository Repository { get; private set; }

    public EmployeePresenter(IEmployeeView view)
    {
        this.View = view;
        this.Repository = new EmployeeRepository();
        this.View.DepartmentSelected += OnDepartmentSelected;
    }

    public void Initialize()
    {
        IEnumerable<Employee> employees = this.Repository.GetEmployees();
        this.View.BindEmployees(employees);
        string[] departments =
            new string[] { "", "销售部", "采购部", "人事部", "IT 部" };
        this.View.BindDepartments(departments);
    }

    protected void OnDepartmentSelected(object sender,
        DepartmentSelectedEventArgs args)
    {
        string department = args.Department;
        var employees = this.Repository.GetEmployees(department);
        this.View.BindEmployees(employees);
    }
}
```

我们在构造函数中注册了 **View** 的 **DepartmentSelected** 事件，作为事件处理器的 **OnDepartmentSelected** 方法通过调用 **Repository**（即 **Model**）得到了用户选择部门下的员工列表，返回的员工列表通过调用 **View** 的 **BindEmployees** 方法绑定在 **View** 上。在 **Initialize** 方法中，通过调用 **Repository** 获取所有员工的列表，并通过调用 **View** 的 **BindEmployees** 方法将员工列表显示在界面上，作为筛选条件的部门列表则通过调用 **View** 的 **BindDepartments** 方法绑定在 **View** 上。

最后我们来看看作为 View 的 Web 页面如何定义。如下所示的是组成该页面的 HTML，其核心部分是一个用于绑定筛选部门列表的 DropDownList 和一个绑定员工列表的 GridView¹，所以无须对它多做介绍。

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>员工管理</title>
  </head>
  <body>
    <form id="form1" runat="server">
      <div id="page">
        <div class="top">
          <asp:DropDownList ID="DropDownListDepartments"
            runat="server" />
          <asp:Button ID="ButtonSearch" runat="server" Text="查询"
            OnClick="ButtonSearch_Click" />
        </div>
        <asp:GridView ID="GridViewEmployees" runat="server"
          AutoGenerateColumns="false" Width="100%">
          <Columns>
            <asp:BoundField DataField="Name" HeaderText="姓名" />
            <asp:BoundField DataField="Gender" HeaderText="性别" />
            <asp:BoundField DataField="BirthDate"
              HeaderText="出生日期"
              DataFormatString="{0:dd/MM/yyyy}" />
            <asp:BoundField DataField="Department" HeaderText="部门"/>
          </Columns>
        </asp:GridView>
      </div>
    </form>
  </body>
</html>
```

如下所示的是该 Web 页面的后台代码的定义，它实现了定义在 IEmployeeView 接口的方法（BindEmployees 和 BindDepartments）和事件（DepartmentSelected）。表示 Presenter 的同名只读属性在构造函数中被初始化。在页面加载的时候（Page_Load 方法）Presenter 的 Initialize 方法被调用，而在“查询”按钮被单击的时候（ButtonSearch_Click 方法）事件 DepartmentSelected 被触发。

```
public partial class Default : Page, IEmployeeView
{
    public EmployeePresenter Presenter { get; private set; }
    public event EventHandler<DepartmentSelectedEventArgs> DepartmentSelected;
```

¹ 为了尽可能地美化最终呈现出来的界面，我们会应用一些 CSS 样式，但是为了让文中的代码尽可能地简洁，我们并不会给出这些 CSS 的定义，所以本书的读者请不要纠结给出的 HTML 与最终呈现出来的界面样式不一致的问题。

```

public Default()
{
    this.Presenter = new EmployeePresenter(this);
}

protected void Page_Load(object sender, EventArgs e)
{
    if (!this.IsPostBack)
    {
        this.Presenter.Initialize();
    }
}

protected void ButtonSearch_Click(object sender, EventArgs e)
{
    string department = this.DropDownListDepartments.SelectedValue;
    DepartmentSelectedEventArgs eventArgs =
        new DepartmentSelectedEventArgs(department);
    if (null != DepartmentSelected)
    {
        DepartmentSelected(this, eventArgs);
    }
}

public void BindEmployees(IEnumerable<Employee> employees)
{
    this.GridViewEmployees.DataSource = employees;
    this.GridViewEmployees.DataBind();
}

public void BindDepartments(IEnumerable<string> departments)
{
    this.DropDownListDepartments.DataSource = departments;
    this.DropDownListDepartments.DataBind();
}
}

```

1.2.2 Model 2

Trygve M. H. Reenskau 当初提出的 MVC 是作为桌面应用的架构模式,所以并不太适合 Web 本身的特性(虽然 MVC 和 MVP 也可以直接用于 ASP.NET Web Forms 应用,但这是因为微软就是采用桌面应用的编程模式来设计 ASP.NET Web Forms 应用框架的)。Web 应用与桌面应用的主要区别在于用户是通过浏览器与应用进行交互,交互请求和响应是通过 HTTP 请求和响应来完成的。

为了让 MVC 能够为 Web 应用提供原生的支持,另一个被称为 Model 2 的 MVC 变体被提出来,这是一种来源于 Java 阵营的 Web 应用架构模式。Java Web 应用具有两种基本的基于 MVC

的架构模式，分别被称为 Model 1 和 Model 2。Model 1 类似于我们前面提及的自治视图模式，它将数据的可视化呈现和用户交互操作的处理逻辑合并在一起。Model 1 适用于那些比较简单的 Web 应用，对于相对复杂的应用多采用 Model 2。

为了让开发者采用相同的编程模式进行桌面应用和 Web 应用的开发，微软通过 ViewState 和 Postback 对 HTTP 请求和响应机制进行了封装，它使我们能够像编写 Windows Forms 应用一样采用事件驱动的方式进行 ASP.NET Web Forms 应用的编程。Model 2 则采用完全不同的设计，它让开发者直接面向 Web，关注 HTTP 的请求和响应，所以 Model 2 提供对 Web 应用原生的支持。

对于 Web 应用来说，和用户直接交互的 UI 界面由浏览器来呈现，用户交互请求通过浏览器以 HTTP 请求的方式发送到 Web 服务器，服务器对请求进行相应的处理并最终返回一个 HTTP 回复对请求予以响应。接下来我们详细讨论 Model 2 模式下作为 MVC 的三要素是如何相互协作最终完成对请求的响应的。

Model 2 中一个 HTTP 请求的目标是 Controller 中的某个 Action，具体体现为定义在 Controller 类型中的某个方法，所以对请求的处理最终体现在对目标 Controller 对象的激活和对目标 Action 方法的执行。一般来说，Controller 的类型和 Action 方法的名称及作为 Action 方法的部分参数可以直接通过请求的 URL 解析出来。

如图 1-4 所示，我们通过一个拦截器（Interceptor）对抵达 Web 服务器的 HTTP 请求进行拦截。一般的 Web 应用框架都提供了这样的拦截机制，对于 ASP.NET 来说，我们可以通过 HttpModule 的形式来定义这么一个拦截器。这个拦截器根据当前请求解析出目标 Controller 的类型和对应的 Action 方法的名称，随后目标 Controller 被激活，相应的 Action 方法被执行。

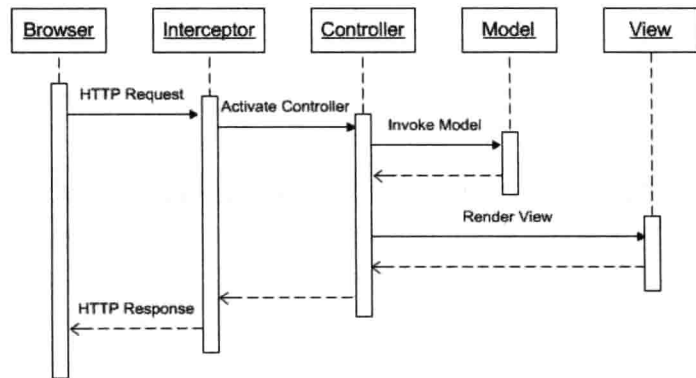


图 1-4 Model 2 交互流程

目标 Action 方法被执行过程中，它可以调用 Model 获取相应的数据或者改变其状态。在 Action 方法执行的最后阶段一般会创建一个 View，后者最终被转换成 HTML 以 HTTP 响应的形式返回到客户端并呈现在浏览器中。绑定在 View 上的数据来源于 Model 或者基于显示要求进行的简单逻辑计算，我们有时候将它们称为 VM（View Model），即基于 View 的 Model（这里的 View Model 与 MVVM 模式下的 VM 是完全不同的两个概念，后者不仅包括呈现在 View 中的数据，也包括数据操作行为）。

1.2.3 ASP.NET MVC 与 Model 2

ASP.NET MVC 就是根据 Model 2 模式设计的。对 HTTP 请求进行拦截以实现对目标 Controller 和 Action 名称的解析是通过一个自定义 HttpModule 来实现的，目标 Controller 的激活和 Action 方法的执行则通过一个自定义 HttpHandler 来完成。在本章的最后我们会通过一个例子来模拟 ASP.NET MVC 的工作原理。

在前面我们多次强调 MVC 的 Model 主要体现为维持应用状态并提供业务功能的领域模型，或者是多层架构中进入业务层的入口或业务服务的代理，但是 ASP.NET MVC 中的 Model 还是这个 Model 吗？稍微了解 ASP.NET MVC 的读者都知道，ASP.NET MVC 的 Model 仅仅是绑定到 View 上的数据而已，它和 MVC 模式中的 Model 并不是一回事。由于 ASP.NET MVC 中的 Model 是服务于 View 的，我们可以将其称为 View Model。

由于 ASP.NET MVC 只有 View Model，所以 ASP.NET MVC 应用框架本身仅仅关注 View 和 Controller，真正的 Model 及 Model 和 Controller 之间的交互体现在我们如何来设计 Controller。

1.3 IIS/ASP.NET 管道

我们在前面对 MVC 模式及其变体做了详细的介绍，其目的在于让读者充分地了解 ASP.NET MVC 框架的设计思想，接下来介绍支撑 ASP.NET MVC 的技术平台。顾名思义，ASP.NET MVC 就是建立在 ASP.NET 平台上基于 MVC 模式的 Web 应用框架，深刻理解 ASP.NET MVC 的前提是对 ASP.NET 管道式设计具有深刻的认识。由于 ASP.NET Web 应用大都寄宿于 IIS 上，所以我们将两者结合起来，力求让读者完整地理解请求在 IIS 和 ASP.NET 管道中是如何流动的。由于不同版本的 IIS 的处理方式具有很大的差异，接下来会介绍 3 个主要的 IIS 版本各自对 Web 请求的不同处理方式。

1.3.1 IIS 5.x 与 ASP.NET

我们先来看看 IIS 5.x 是如何处理基于 ASP.NET 资源（比如.aspx、.asmx 等）请求的。如图 1-5 所示，IIS 5.x 运行在进程 InetInfo.exe 中，该进程寄宿着一个名为 World Wide Web Publishing Service（简称 W3SVC）的 Windows 服务。W3SVC 主要负责 HTTP 请求的监听、激活和管理工作进程、加载配置（通过从 Metabase 中加载相关配置信息）等。

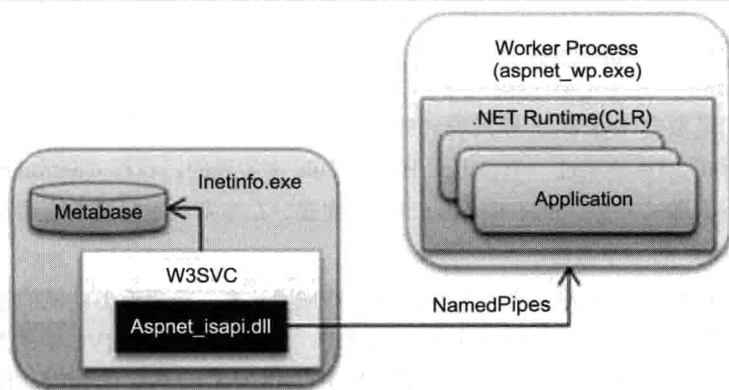


图 1-5 IIS 5.x 与 ASP.NET

当检测到某个 HTTP 请求时，IIS 先根据扩展名判断请求的是静态资源（比如.html、.img、.txt、.xml 等）还是动态资源。对于前者，IIS 会将文件的内容直接响应给客户端，对于动态资源（比如.aspx、.asp、.php 等）则通过扩展名从 IIS 的脚本映射（Script Map）中找到相应的 ISAPI 动态链接库（Dynamic Link Library，DLL）。

ISAPI (Internet Server Application Programming Interface) 是一套本地的 (Native) Win32 API，是 IIS 和其他动态 Web 应用或平台之间的纽带。ISAPI 定义在一个动态链接库 (DLL) 文件中，ASP.NET ISAPI 对应的 DLL 文件名称为 aspnet_isapi.dll，我们可以在目录 “%windir%\Microsoft.NET\Framework\{version no}\” 中找到它。ISAPI 支持 ISAPI 扩展 (ISAPI Extension) 和 ISAPI 筛选 (ISAPI Filter)，前者是真正处理 HTTP 请求的接口，后者则可以在 HTTP 请求真正被处理之前查看、修改、转发或拒绝请求，比如 IIS 可以利用 ISAPI 筛选进行请求的验证。

如果我们请求的是一个基于 ASP.NET 的资源类型，比如.aspx、.asmx、.svc 等，aspnet_isapi.dll 会被加载，ASP.NET ISAPI 随后会创建 ASP.NET 的工作进程（如果该进程尚未启动）。对于 IIS 5.x 来说，该工作进程为 aspnet.exe。IIS 进程与工作进程之间通过命名管道 (Named Pipes) 进

行通信。

在工作进程初始化过程中，.NET 运行时（CLR）会被加载以构建一个托管的环境。对于某个 Web 应用的初次请求，CLR 会为其创建一个应用程序域（Application Domain）。在应用程序域中，HTTP 运行时（HTTP Runtime）被加载并用以创建相应的应用。寄宿于 IIS 5.x 的所有 Web 应用都运行在同一个进程（工作进程 `aspnet_wp.exe`）的不同应用程序域中。

1.3.2 IIS 6.0 与 ASP.NET

以现在的眼光来审视 IIS 5.x，一定觉得它是一个很古老的版本，但是由于服役最长的 Windows XP 操作系统上搭载的就是这款产品，所以我们应该对它不会感到陌生。通过上面的介绍，我们可以看出 IIS 5.x 至少存在着如下两个方面的不足。

- ISAPI 动态链接库被加载到 `InetInfo.exe` 进程中，它和工作进程之间是一种典型的跨进程通信方式，尽管采用命名管道，但是仍然会带来性能的瓶颈。
- 所有的 ASP.NET 应用运行在相同进程（`aspnet_wp.exe`）的不同应用程序域中，基于应用程序域的隔离不能从根本上解决一个应用程序对另一个程序的影响，在更多的时候我们需要不同的 Web 应用运行在不同的进程中。

为了解决第一个问题，IIS 6.0 将 ISAPI 动态链接库直接加载到工作进程中。为了解决第二个问题，在 IIS 6.0 中引入了应用程序池（Application Pool）的机制。我们可以为一个或多个 Web 应用创建一个应用程序池，每一个应用程序池对应一个独立的工作进程（`w3wp.exe`），所以运行在不同应用程序池中的 Web 应用提供基于进程级别的隔离机制。

除了上面两点改进之外，IIS 6.0 还有其他一些值得称道的地方，其中最重要的一点就是创建了一个名为 HTTP.SYS 的 HTTP 监听器。HTTP.SYS 以驱动程序的形式运行在 Windows 的内核模式（Kernel Mode）下，它是 Windows TCP/IP 网络子系统的一部分，从结构上看它属于 TCP 之上的一个网络驱动程序。

严格地说，HTTP.SYS 已经不属于 IIS 的范畴了，所以 HTTP.SYS 的配置信息也没有保存在 IIS 的元数据库（Metabase），而是定义在注册表中。HTTP.SYS 的注册表项的路径为“`HKEY_LOCAL_MACHINE/SYSTEM/CurrentControlSet/Services/HTTP`”。HTTP.SYS 能够带来如下的好处。

- 持续监听。由于 HTTP.SYS 是一个网络驱动程序，始终处于运行状态，所以对于用户的 HTTP 请求能够及时作出反应。

- 更好的稳定性。HTTP.SYS 运行在操作系统内核模式下，并不执行任何用户代码，所以其本身不会受到 Web 应用、工作进程和 IIS 进程的影响。
- 内核模式下数据缓存。如果某个资源被频繁请求，HTTP.SYS 会把响应的内容进行缓存，缓存的内容可以直接响应后续的请求。由于这是基于内核模式的缓存，不存在内核模式和用户模式的切换，响应速度将得到极大的改进。

图 1-6 体现了 IIS 6.0 的结构和处理 HTTP 请求的流程。与 IIS 5.x 不同，W3SVC 在 IIS 6.0 中从 InetInfo.exe 进程脱离出来（对于 IIS 6.0 来说，InetInfo.exe 基本上可以看作单纯的 IIS 管理进程）运行在另一个进程 SvcHost.exe 中。不过 W3SVC 的基本功能并没有发生变化，只是在功能的实现上作了相应的改进。与 IIS 5.x 一样，元数据库（Metabase）依然存在于 InetInfo.exe 进程中。

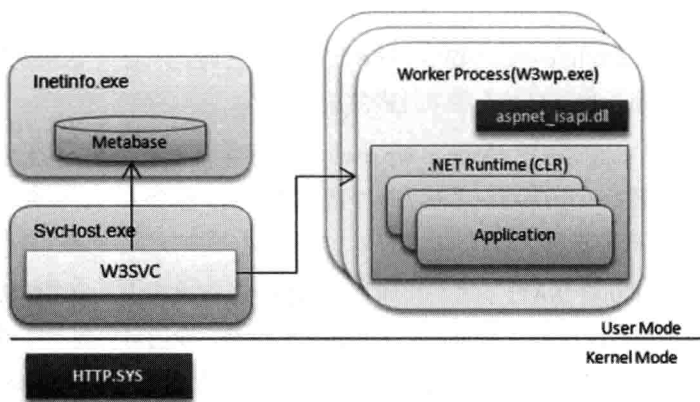


图 1-6 IIS 6.0 与 ASP.NET

当监听到 HTTP 请求时，HTTP.SYS 将其分发给 W3SVC，后者解析出请求的 UR，并根据从 Metabase 获取的 URL 与 Web 应用之间的映射关系得到目标应用，进而得到目标应用运行的应用程序池或工作进程。如果工作进程不存在（尚未创建或被回收），它为该请求创建新的工作进程。在工作进程的初始化过程中，相应的 ISAPI 动态链接库被加载，对于 ASP.NET 应用来说，被加载的 ISAPI.dll 为 aspnet_isapi.dll。ASP.NET ISAPI 负责进行 CLR 的加载、应用程序域的创建和 Web 应用的初始化等操作。

1.3.3 IIS 7.0 与 ASP.NET

IIS 7.0 在请求的监听和分发机制上又进行了革新性的改进,主要体现在引入 Windows 进程激活服务 (Windows Process Activation Service, WAS) 分流了原来 (IIS 6.0) W3SVC 承载的部分功能。通过上面的介绍我们知道, IIS 6.0 中的 W3SVC 主要承载着如下三大功能。

- HTTP 请求接收: 接收 HTTP.SYS 监听到的 HTTP 请求。
- 配置管理: 从元数据库 (Metabase) 中加载配置信息对相关组件进行配置。
- 进程管理: 创建、回收、监控工作进程。

IIS 7.0 将后两组功能实现到了 WAS 中,但接收 HTTP 请求的任务依然落在 W3SVC 头上。WAS 的引入为 IIS 7.0 提供了对非 HTTP 协议的支持,它通过监听适配器接口 (Listener Adapter Interface) 抽象出针对不同协议的监听器。具体来说,除了专门用于监听 HTTP 请求的 HTTP.SYS 之外, WAS 利用 TCP 监听器、命名管道监听器和 MSMQ 监听器提供基于 TCP、命名管道和 MSMQ 传输协议的监听支持。

与此 3 种监听器相对应的是 3 种监听适配器,它们提供监听器与 WAS 中的监听适配器接口之间的适配 (从这个意义上讲, IIS 7.0 中的 W3SVC 相当于 HTTP.SYS 的监听适配器)。这 3 种非 HTTP 监听器和监听适配器定义在程序集 SMSvcHost.exe 中,我们可以在目录 “%windir%\Microsoft.NET\Framework\v3.0\Windows Communication Foundation\” 中找到它们。

从程序集所在的目录名称可以看出,这 3 种监听器/监听适配器是为 WCF 设计的,它们以 Windows 服务的形式进行工作。虽然它们定义在一个程序集中,但我们依然可以通过服务管理器对其进行单独的启动、终止和配置。总的来说, SMSvcHost.exe 提供了 4 个重要的 Windows Service, 如图 1-7 所示为上述的 4 个 Windows 服务在服务控制管理器中的呈现。

- NetTcpPortSharing: 为 WCF 提供 TCP 端口共享,即同一个监听端口被多个进程共享。
- NetTcpActivator: 为 WAS 提供基于 TCP 的激活请求,包含 TCP 监听器和对应的监听适配器。
- NetPipeActivator: 为 WAS 提供基于命名管道的激活请求,包含命名管道监听器和对应的监听适配器。
- NetMsmqActivator: 为 WAS 提供基于 MSMQ 的激活请求,包含 MSMQ 监听器和对应的监听适配器。

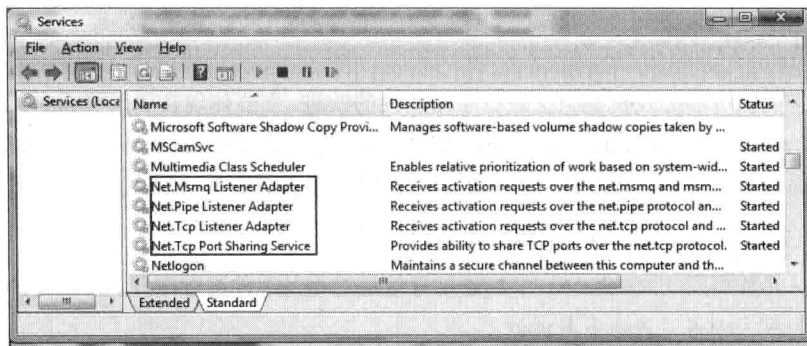


图 1-7 定义在 SMSvcHost.exe 中的 Windows Service

图 1-8 揭示了 IIS 7.0 的整体架构及整个请求处理流程。无论是从 W3SVC 接收到的 HTTP 请求，还是通过 WCF 提供的监听适配器接收到的针对其他传输协议请求，最终都会被传递到 WAS。如果相应的工作进程（针对单个应用程序池）尚未创建，则 WAS 会创建工作进程。WAS 在进行请求处理过程中通过内置的配置管理模块加载相关的配置信息，并对相关的组件进行配置。与 IIS 5.x 和 IIS 6.0 基于 Metabase 的配置信息存储不同的是，IIS 7.0 大都将配置信息存放于 XML 形式的配置文件中，基本的配置存放在 applicationHost.config 中。

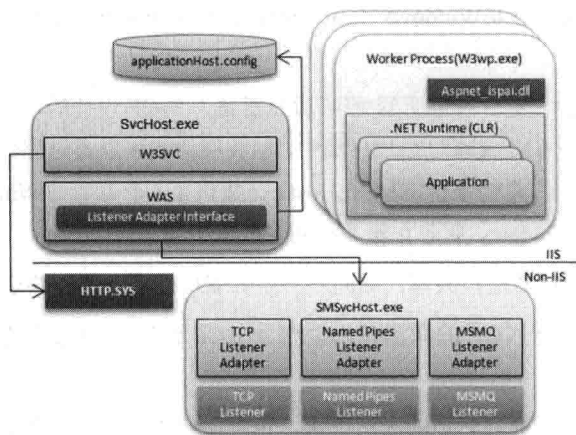


图 1-8 IIS 7.0 与 ASP.NET

1.3.4 ASP.NET 集成

从上面对 IIS 5.x 和 IIS 6.0 的介绍中我们不难发现，IIS 与 ASP.NET 是两个相互独立的管道

(Pipeline)。在各自管辖范围内，它们各自具有自己的一套机制对 HTTP 请求进行处理。两个管道通过 ISAPI 实现“连通”，IIS 是第一道屏障，当对 HTTP 请求进行必要的前期处理（比如身份验证等）后，IIS 通过 ISAPI 将请求分发给 ASP.NET 管道。当 ASP.NET 在自身管道范围内完成对 HTTP 请求的处理时，处理后的结果再返回到 IIS，IIS 对其进行后期处理（比如日志记录、压缩等）后生成 HTTP 回复对请求予以响应。图 1-9 反映了 IIS 6.0 与 ASP.NET 之间的桥接关系。

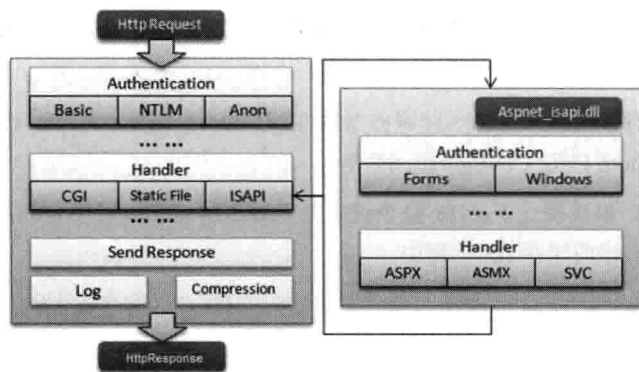


图 1-9 基于 IIS 6.0 与 ASP.NET 双管道设计

从另一个角度讲，IIS 运行在非托管的环境中，而 ASP.NET 管道则是托管的，所以说 ISAPI 还是连接非托管环境和托管环境的纽带。IIS 5.x 和 IIS 6.0 把两个管道进行隔离至少带来了下面的一些局限与不足。

- 相同操作的重复执行。IIS 与 ASP.NET 之间具有一些重复的操作，比如身份验证。
- 动态文件与静态文件处理的不一致。因为只有基于 ASP.NET 动态文件（比如.aspx、.asmx、.svc 等）的 HTTP 请求才能通过 ASP.NET ISAPI 进入 ASP.NET 管道，而对于一些静态文件（比如.html、.xml、.img 等）的请求则由 IIS 直接响应，那么 ASP.NET 管道中的一些功能将不能作用于这些基于静态文件的请求，比如我们希望通过 Forms 认证应用于基于图片文件的请求就做不到。
- IIS 难以扩展。对于 IIS 的扩展基本上就体现在自定义 ISAPI，但是对于大部分人来说，这不是一件容易的事情，因为 ISAPI 是基于 Win32 的非托管的 API，并非一种面向应用的编程接口。通常我们希望的是诸如定义 ASP.NET 的 HttpModule 和 HttpHandler 一样，通过托管代码的方式来扩展 IIS。

对于 Windows 平台下的 IIS 来讲，ASP.NET 无疑是一等公民，它们之间不应该是“井水不犯河水”而应该是“你中有我，我中有你”的关系，为此在 IIS 7.0 中实现了两者的集成，通过

集成可以获得如下的好处。

- 允许通过本地代码（Native Code）和托管代码（Managed Code）两种方式定义 IIS Module，这些 IIS Module 注册到 IIS 中形成一个通用的请求处理管道。由这些 IIS Module 组成的这个管道能够处理所有的请求，不论请求基于怎样的资源类型。例如，可以将 FormsAuthenticationModule 提供的 Forms 认证应用到基于.aspx、CGI 和静态文件的请求。
- 将 ASP.NET 提供的一些强大的功能应用到原来难以企及的地方，比如将 ASP.NET 的 URL 重写功能置于身份验证之前。
- 采用相同的方式去实现、配置、检测和支持一些服务器特性（Feature），比如 Module、Handler 映射、定制错误配置（Custom Error Configuration）等。

图 1-10 演示了在 ASP.NET 集成模式下，IIS 整个请求处理管道的结构。可以看到，原来 ASP.NET 提供的托管组件可以直接应用在 IIS 管道中。

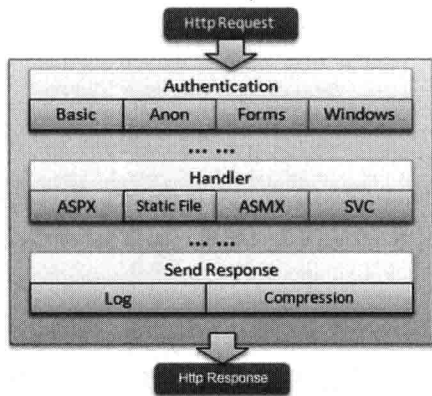


图 1-10 基于 IIS 7.0 与 ASP.NET 集成管道设计

1.3.5 ASP.NET 管道

以 IIS 6.0 为例，它在工作进程 w3wp.exe 中会利用 aspnet_isapi.dll 加载 .NET 运行时（如果 .NET 运行时尚未加载）。IIS 6.0 引入了应用程序池的概念，一个工作进程对应着一个应用程序池。一个应用程序池可以承载一个或多个 Web 应用，每个 Web 应用映射到一个 IIS 虚拟目录。与 IIS 5.x 一样，每一个 Web 应用运行在各自的应用程序域中。

如果 HTTP.SYS 接收到的 HTTP 请求是对该 Web 应用的第一次访问，在成功加载运行时后，IIS 会通过 AppDomainFactory 为该 Web 应用创建一个应用程序域。随后一个特殊的运行时

IsapiRuntime 被加载。IsapiRuntime 定义在程序集 System.Web.dll 中，对应的命名空间为“System.Web.Hosting”，被加载的 IsapiRuntime 会接管该 HTTP 请求。

接管 HTTP 请求的 IsapiRuntime 会首先创建一个 IsapiWorkerRequest 对象来封装当前的 HTTP 请求，随后将此对象传递给 ASP.NET 运行时 HttpRuntime。从此时起，HTTP 请求正式进入了 ASP.NET 管道。HttpRuntime 会根据 IsapiWorkerRequest 对象创建用于表示当前 HTTP 请求的上下文（Context）对象 HttpContext。

随着 HttpContext 的创建，HttpRuntime 会利用 HttpApplicationFactory 创建新的或获取现有的 HttpApplication 对象。实际上 ASP.NET 维护着一个 HttpApplication 对象池，HttpApplicationFactory 从池中选取可用的 HttpApplication 用于处理 HTTP 请求，处理完毕后将其释放到对象池中。HttpApplication 负责处理当前的 HTTP 请求。

在 HttpApplication 初始化过程中，ASP.NET 会根据配置文件加载并初始化注册的 HttpModule 对象。对于 HttpApplication 来说，在它处理 HTTP 请求的不同阶段会触发不同的事件（Event），而 HttpModule 的意义在于通过注册 HttpApplication 的相应事件，将所需的操作注入整个 HTTP 请求的处理流程。ASP.NET 的很多功能（比如身份验证、授权、缓存等）都是通过相应的 HttpModule 实现的。

最终完成对 HTTP 请求的处理实现在 HttpHandler 中，不同的资源类型对应着不同类型的 HttpHandler。比如.aspx 页面对应的 HttpHandler 类型为 System.Web.UI.Page，WCF 的.svc 文件对应的 HttpHandler 类型为 System.ServiceModel.Activation.HttpHandler。上面整个处理流程如图 1-11 所示。

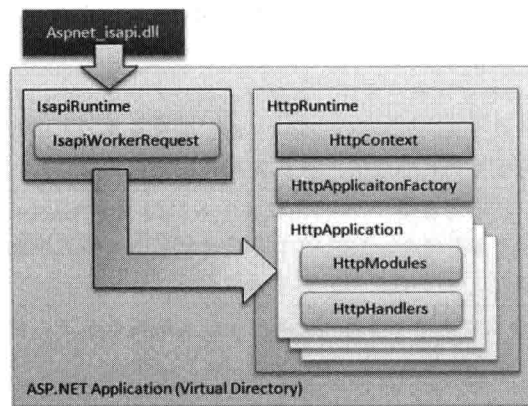


图 1-11 ASP.NET 处理管道

1. HttpApplication

HttpApplication 是整个 ASP.NET 基础架构的核心，它负责处理分发给它的 HTTP 请求。由于一个 HttpApplication 对象在某个时刻只能处理一个请求，只有完成对某个请求的处理后才能用于后续请求的处理，所以 ASP.NET 采用对象池的机制来创建或获取 HttpApplication 对象。

当第一个请求抵达时，ASP.NET 会一次创建多个 HttpApplication 对象，并将其置于池中，然后选择其中一个对象来处理该请求。处理完毕后，HttpApplication 不会被回收，而是释放到池中。对于后续的请求，空闲的 HttpApplication 对象会从池中取出。如果池中所有的 HttpApplication 对象都处于繁忙的状态，在没有超出 HttpApplication 池最大容量的情况下，ASP.NET 会创建新的 HttpApplication 对象，否则将请求放入队列等待现有 HttpApplication 的释放。

HttpApplication 处理请求的整个生命周期是一个相对复杂的过程，在该过程的不同阶段会触发相应的事件。我们可以注册相应的事件，将处理逻辑注入到 HttpApplication 处理请求的某个阶段。表 1-1 按照实现的先后顺序列出了 HttpApplication 在处理每一个请求时触发的事件名称。

表 1-1 HttpApplication 事件列表

名 称	描 述
BeginRequest	HTTP 管道开始处理请求时，会触发 BeginRequest 事件
AuthenticateRequest, PostAuthenticateRequest	ASP.NET 先后触发这两个事件，使安全模块对请求进行身份验证
AuthorizeRequest, PostAuthorizeRequest	ASP.NET 先后触发这两个事件，使安全模块对请求进行授权
ResolveRequestCache, PostResolveRequestCache	ASP.NET 先后触发这两个事件，以使缓存模块利用缓存的内容对请求直接进行响应（缓存模块可以将响应内容进行缓存，对于后续的请求，直接将缓存的内容返回，从而提高响应能力）
PostMapRequestHandler	对于访问不同的资源类型，ASP.NET 具有不同的 HttpHandler 对其进行处理。对于每个请求，ASP.NET 会通过扩展名选择匹配相应的 HttpHandler 类型，成功匹配后，该事件被触发
AcquireRequestState, PostAcquireRequestState	ASP.NET 先后触发这两个事件，使状态管理模块获取基于当前请求相应的状态，如 SessionState
PreRequestHandlerExecute, PostRequestHandlerExecute	ASP.NET 最终通过与请求资源类型相对应的 HttpHandler 实现对请求的处理，在实行 HttpHandler 前后，这两个事件被先后触发

续表

名 称	描 述
ReleaseRequestState, PostReleaseRequestState	ASP.NET 先后触发这两个事件，使状态管理模块释放基于当前请求相应的状态
UpdateRequestCache, PostUpdateRequestCache	ASP.NET 先后触发这两个事件，以使缓存模块将 HttpHandler 处理请求得到的内容得以保存到输出缓存中
LogRequest, PostLogRequest	ASP.NET 先后触发这两个事件为当前请求进行日志记录
EndRequest	整个请求处理完成后，EndRequest 事件被触发

对于一个 ASP.NET 应用来说，HttpApplication 派生于 Global.asax 文件，我们可以通过创建 Global.asax 文件对 HttpApplication 的请求处理行为进行定制。Global.asax 采用一种很直接的方式实现了这样的功能，这种方式不是我们常用的方法重写或事件注册，而是直接采用方法名匹配。在 Global.asax 中，我们按照 “Application_{Event Name}” 这样的方法命名规则进行事件注册。比如 Application_BeginRequest 方法用于处理 HttpApplication 的 BeginRequest 事件。如果通过 VS 创建一个 Global.asax 文件，将采用如下的默认定义。

```
<%@ Application Language="C#" %>
<script runat="server">
    void Application_Start(object sender, EventArgs e){}
    void Application_End(object sender, EventArgs e){}
    void Application_Error(object sender, EventArgs e){}
    void Session_Start(object sender, EventArgs e){}
    void Session_End(object sender, EventArgs e){}
</script>
```

2. HttpModule

ASP.NET 拥有一个具有高度可扩展性的引擎，并且能够处理对于不同资源类型的请求。那麽是什么成就了 ASP.NET 的扩展性呢？HttpModule 功不可没。

当请求转入 ASP.NET 管道时，最终负责处理该请求的是与请求资源类型相匹配的 HttpHandler 对象，但是在 HttpHandler 正式工作之前 ASP.NET 会先加载并初始化所有配置的 HttpModule 对象。HttpModule 在初始化的过程中，会将一些回调操作注册到 HttpApplication 相应的事件中，在 HttpApplication 请求处理生命周期中的某个阶段，相应的事件会被触发，通过 HttpModule 注册的事件处理程序也得以执行。

所有的 HttpModule 都实现了具有如下定义的 System.Web.IHttpModule 接口，其 Init 方法实现了针对自身的初始化。该方法接受一个 HttpApplication 对象，有了这个对象，事件注册就很容易了。

```
public interface IHttpModule
{
    void Dispose();
    void Init(HttpApplication context);
}
```

ASP.NET 提供的很多基础功能都是通过相应的 `HttpModule` 实现的，下面列出了一些典型的 `HttpModule`。除了这些系统定义的 `HttpModule` 之外，我们还可以自定义 `HttpModule`，通过 `Web.config` 可以很容易地将其注册到 Web 应用中。

- `OutputCacheModule`：实现了输出缓存（Output Caching）的功能。
- `SessionStateModule`：在无状态的 HTTP 协议上实现了基于会话（Session）的状态保持。
- `WindowsAuthenticationModule` + `FormsAuthenticationModule` + `PassportAuthenticationModule`：实现了 Windows、Forms 和 Passport 这 3 种典型的身份认证方式。
- `UrlAuthorizationModule` + `FileAuthorizationModule`：实现了基于 URI 和文件 ACL（Access Control List）的授权。

3. `HttpHandler`

对于不同资源类型的请求，ASP.NET 会加载不同的 `Handler` 来处理，比如 `.aspx` 页面与 `.asmx` Web 服务对应的 `Handler` 是不同的。所有的 `HttpHandler` 都实现了具有如下定义的接口 `System.Web.IHttpHandler`，定义其中的方法 `ProcessRequest` 提供了处理请求的实现。另一个代表异步版本的 `HttpHandler` 的 `IHttpAsyncHandler` 接口继承自 `IHttpHandler`，它通过调用 `BeginProcessRequest/EndProcessRequest` 方法以异步的方式处理请求。

```
public interface IHttpHandler
{
    void ProcessRequest(HttpContext context);
    bool IsReusable { get; }
}

public interface IHttpAsyncHandler : IHttpHandler
{
    IAsyncResult BeginProcessRequest(HttpContext context, AsyncCallback cb,
        object extraData);
    void EndProcessRequest(IAsyncResult result);
}
```

某些 `HttpHandler` 具有一个与之相关的 `HttpHandlerFactory`，后者实现了具有如下定义的接口 `System.Web.IHttpHandlerFactory`，定义其中的方法 `GetHandler` 用于创建新的 `HttpHandler` 或者获取已经存在的 `HttpHandler`。

```
public interface IHttpHandlerFactory
{
    IHttpHandler GetHandler(HttpContext context, string requestType,
        string url, string pathTranslated);
    void ReleaseHandler(IHttpHandler handler);
}
```

HttpHandler 和 IHttpHandlerFactory 的类型都可以通过相同的方式配置到 Web.config 中。下面一段配置包含对.aspx、.asmx 和.svc 这 3 种典型的资源类型的 IHttpHandler/IHttpHandlerFactory 配置。

```
<configuration>
  <system.web>
    <httpHandlers>
      <add path="*.svc"
        verb="*"
        type="System.ServiceModel.Activation.HttpHandler,
          System.ServiceModel, Version=4.0.0.0, Culture=neutral,
          PublicKeyToken=b77a5c561934e089"
        validate="false"/>
      <add path="*.aspx"
        verb="*"
        type="System.Web.UI.PageHandlerFactory"
        validate="true"/>
      <add path="*.asmx"
        verb="*"
        type="System.Web.Services.Protocols.WebServiceHandlerFactory,
          System.Web.Services, Version=4.0.0.0, Culture=neutral,
          PublicKeyToken=b03f5f7f11d50a3a"
        validate="False"/>
    </httpHandlers>
  </system.web>
</configuration>
```

除了通过配置建立起 IHttpHandler 类型与请求路径模式之间的映射关系之外，我们还可以调用当前 HttpContext 具有如下定义的 RemapHandler 方法将一个 IHttpHandler 对象映射到当前 HTTP 请求。如果不曾通过调用该方法进行 IHttpHandler 的显式映射，或者调用该方法时传入的参数为 Null，真正的 IHttpHandler 对象的映射发生在 HttpApplication 的 PostMapRequestHandler 触发之前，默认进行 IHttpHandler 的依据就是上述的配置。

```
public sealed class HttpContext
{
    //其他操作
    public void RemapHandler(IHttpHandler handler)
}
```


换句话说，在调用当前 `HttpContext` 的 `RemapHandler` 方法时指定一个具体的 `HttpHandler` 对象，是为了让 ASP.NET 直接跳过默认的 `HttpHandler` 映射操作。此外，由于这个默认的 `HttpHandler` 映射发生在 `HttpApplication` 的 `PostMapRequestHandler` 事件触发之前，所以只有在这之前调用 `RemapHandler` 方法才有意义。通过阅读下一节的内容，我们就可以知道实现 ASP.NET MVC 框架的 `MvcHandler`（一个自定义的 `HttpHandler`）就是通过调用这个方法进行映射的。

1.4 ASP.NET MVC 是如何运行的

ASP.NET 由于采用了管道式设计，所以具有很好的扩展性，整个 ASP.NET MVC 应用框架就是通过扩展 ASP.NET 实现的。通过上面对 ASP.NET 管道设计的介绍我们知道，ASP.NET 的扩展点主要体现在 `HttpModule` 和 `HttpHandler` 这两个核心组件之上，整个 ASP.NET MVC 框架就是通过自定义的 `HttpModule` 和 `HttpHandler` 建立起来的。

为了使读者能够从整体上把握 ASP.NET MVC 框架的工作机制，接下来我们按照其原理通过一些自定义组件来模拟 ASP.NET MVC 的运行原理，也可以将此视为一个“迷你版”的 ASP.NET MVC。值得一提的是，为了让读者根据该实例从真正的 ASP.NET MVC 中找到对应的类型，本例完全采用了与 ASP.NET MVC 一致的类型命名方式。

1.4.1 建立在“迷你版”ASP.NET MVC 上的 Web 应用

在正式介绍我们自己创建的“迷你版”ASP.NET MVC 的实现原理之前，不妨来看看建立在该框架之上的 Web 应用如何来搭建。我们通过 Visual Studio 创建一个空的 ASP.NET Web 应用，注意不是 ASP.NET MVC 应用，我们也并不会引用“`System.Web.Mvc.dll`”这个程序集，所以在接下来的程序中看到的所谓 MVC 的类型都是我们自行定义的。

我们首先定义了如下一个 `SimpleModel` 类型，它表示最终需要绑定到 View 上的数据。为了验证针对目标 Controller 和 Action 的解析机制，`SimpleModel` 定义的两个属性分别表示当前请求的目标 Controller 和 Action。为了更好地演示 ASP.NET MVC 的参数绑定机制（Model 绑定），我们为 `SimpleModel` 定义了额外 3 个属性 `Foo`、`Bar` 和 `Baz`，并且让它们具有不同的数据类型。

```
public class SimpleModel
{
    public string Controller { get; set; }
    public string Action { get; set; }

    public string    Foo { get; set; }
    public int       Bar { get; set; }
```

```

    public double    Baz { get; set; }
}

```

与真正的 ASP.NET MVC 应用开发一样，我们需要定义 Controller 类。按照约定的命名方式（以字符“Controller”作为后缀），我们定义了如下一个继承自抽象类 ControllerBase 的 HomeController。定义在 HomeController 中的 Action 方法 Index 具有一个 SimpleModel 类型的输入参数，并以 ActionResult 作为返回类型。

```

public class HomeController : ControllerBase
{
    public ActionResult Index(SimpleModel model)
    {
        Action<TextWriter> callback = writer =>
        {
            writer.Write(string.Format(
                "Controller: {0}<br/>Action: {1}<br/><br/>",
                model.Controller, model.Action));
            writer.Write(string.Format(
                "Foo: {0}<br/>Bar: {1}<br/>Baz: {2}",
                model.Foo, model.Bar, model.Baz));
        };
        return new RawContentResult(callback);
    }
}

```

如上面的代码片段所示，我们让 Action 方法 Index 返回一个 RawContentResult 对象。顾名思义，RawContentResult 旨在将我们写入的内容原样呈现出来。一个 RawContentResult 对象是对一个 Action<TextWriter>委托的封装，它利用后者写入需要呈现的内容。在这里我们将作为参数的 SimpleModel 对象的两组属性（Controller/Action 和 Foo/Bar/Baz）的值显示出来。

ASP.NET MVC 根据请求的 URL 来解析目标 Controller 的类型和 Action 方法名称。具体来说，我们会注册一些包含 Controller 和 Action 名称作为占位符的路由模板。如果请求地址符合相应地址模板的模式，目标 Controller 和 Action 的名称就可以正确地解析出来。我们在 Global.asax 中注册了如下一个模板为“{controller}/{action}”的 Route 对象。除此之外，我们还注册了一个用于创建 Controller 对象的工厂 DefaultControllerFactory。

```

public class Global : System.Web.HttpApplication
{
    protected void Application_Start(object sender, EventArgs e)
    {
        RouteTable.Routes.Add("default",
            new Route{Url = "{controller}/{action}"});

        ControllerBuilder.Current.SetControllerFactory(
            new DefaultControllerFactory());
    }
}

```

路由实现了请求 URL 与目标 Controller/Action 之间的映射。ASP.NET MVC 的路由建立在 ASP.NET 自身的路由系统之上，后者则通过一个自定义的 `HttpModule` 来实现。在这个“迷你版”ASP.NET MVC 框架中，我们将其命名为 `UrlRoutingModule`，它与 ASP.NET 路由系统中对应的 `HttpModule` 类型同名。在运行 Web 应用之前，我们需要通过配置对该自定义 `HttpModule` 进行注册，下面是相关的配置。

```
//IIS 7.x Integrated 模式
<configuration>
  <system.webServer>
    <modules>
      <add name="UrlRoutingModule"
          type="WebApp.UrlRoutingModule, WebApp"/>
    </modules>
  </system.webServer>
</configuration>

//IIS 7.x Classical 模式或者之前的版本
<configuration>
  <system.web>
    <httpModules>
      <add name="UrlRoutingModule"
          type="WebApp.UrlRoutingModule, WebApp"/>
    </httpModules>
  </system.web>
</configuration>
```

到目前为止，所有的编程和配置工作已经完成。为了让定义在 `HomeController` 中的 `Action` 方法 `Index` 来处理针对该 Web 应用的访问请求，我们需要指定与之匹配的地址（符合定义在注册地址模板的路由模式）。如图 1-12 所示，由于在浏览器中输入的地址（“~/home/index?foo=abc&bar=123&baz=3.14”）正好对应着 `HomeController` 的 `Action` 方法 `Index`，所以对应的方法会被执行。除此之外，请求 URL 携带的 3 个查询字符串正好与 `Action` 方法参数类型 `SimpleModel` 的 3 个属性相匹配（忽略大小写），所以在进行参数绑定过程中能够对它们进行自动绑定，这可以从图 1-12 中看出来。（S102）

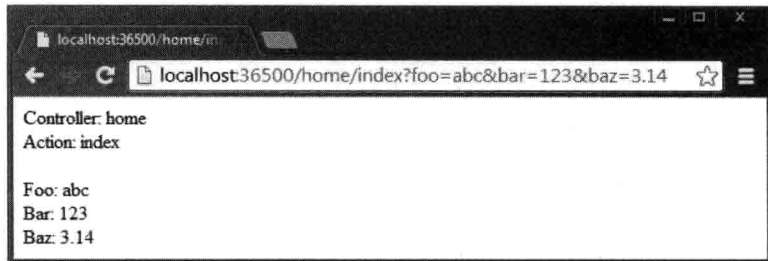


图 1-12 采用符合注册的路由地址模板的地址访问 Web 应用

上面演示了如何在自己创建的“迷你版”ASP.NET MVC 框架上搭建一个 Web 应用，从中可以看到它和创建一个真正的 ASP.NET MVC 应用别无二致。接下来我们就来逐步地分析这个自定义的 ASP.NET MVC 框架是如何建立起来的，它也代表了真正的 ASP.NET MVC 框架的基本工作原理。

总的来说，ASP.NET MVC 按照这样的流程来处理并响应请求：ASP.NET MVC 利用路由系统对请求 URL 进行解析进而得到目标 Controller 和 Action 的名称，以及其他相应的路由数据。它根据 Controller 的名称解析出目标 Controller 的真正类型，并将其激活（默认情况下就是根据类型以反射的机制创建 Controller 对象）。接下来，ASP.NET MVC 利用 Action 名称解析出定义在目标 Controller 类型中对应的方法，然后执行激活 Controller 对象的这个方法。Action 方法可以在执行过程中直接对当前请求予以响应，也可以返回一个 ActionResult 对象来响应请求。对于后者，ASP.NET MVC 在完成目标 Action 方法执行之后，会执行返回的 ActionResult 对象来对当前请求作最终的响应。

1.4.2 路由

对于一个 ASP.NET MVC 应用来说，针对 HTTP 请求的处理实现在目标 Controller 类型的某个 Action 方法中，每个 HTTP 请求不再像 ASP.NET Web Forms 应用一样是针对一个物理文件，而是针对某个 Controller 的某个 Action 方法。目标 Controller 和 Action 的名称由 HTTP 请求的 URL 来决定，当 ASP.NET MVC 接收到抵达的请求后，其首要任务就是通过当前 HTTP 请求的解析得到目标 Controller 和 Action 的名称，这个过程是通过 ASP.NET MVC 的路由系统来实现的。我们通过如下几个对象构建了一个简易的路由系统。

1. RouteData

ASP.NET 定义了一个全局的路由表，路由表中的每个 Route 对象包含一个路由模板。目标 Controller 和 Action 的名称可以通过路由变量以占位符（比如“{controller}”和“{action}”）的形式定义在模板中，也可以作为路由对象的默认值（无须出现在路由模板中）。对于每一个抵达的 HTTP 请求，路由系统会遍历路由表并找到一个具有与当前请求 URL 模式相匹配的 Route 对象，然后利用它解析出以 Controller 和 Action 名称为核心的路由数据。在我们自建的 ASP.NET MVC 框架中，通过路由解析得到的路由数据通过具有如下定义的 RouteData 类型表示。

```
public class RouteData
{
    public IDictionary<string, object> Values { get; private set; }
    public IDictionary<string, object> DataTokens { get; private set; }
    public IRouteHandler RouteHandler { get; set; }
```

```

public RouteBase Route { get; set; }

public RouteData()
{
    this.Values = new Dictionary<string, object>();
    this.DataTokens = new Dictionary<string, object>();
    this.DataTokens.Add("namespaces", new List<string>());
}

public string Controller
{
    get
    {
        object controllerName = string.Empty;
        this.Values.TryGetValue("controller", out controllerName);
        return controllerName.ToString();
    }
}

public string ActionName
{
    get
    {
        object actionName = string.Empty;
        this.Values.TryGetValue("action", out actionName);
        return actionName.ToString();
    }
}
}

```

如上面的代码片段所示, `RouteData` 定义了两个字典类型的属性 `Values` 和 `DataTokens`, 它们代表具有不同来源的路由变量, 前者由对请求 URL 实施路由解析获得。表示 `Controller` 和 `Action` 名称的属性 (`Controller` 和 `ActionName`) 直接从 `Values` 属性表示的字典中提取, 对应的 Key 分别为 “controller” 和 “action”。

我们之前已经提到过 ASP.NET MVC 本质上是由两个自定义的 ASP.NET 组件来实现的, 一个是自定义的 `HttpModule`, 另一个是自定义的 `HttpHandler`, 后者从 `RouteData` 对象的 `RouteHandler` 属性获得。`RouteData` 的 `RouteHandler` 属性类型为 `IRouteHandler` 接口, 如下面的代码片段所示, 该接口具有一个唯一的 `GetHttpHandler` 方法返回真正用于处理 HTTP 请求的 `HttpHandler` 对象。

```

public interface IRouteHandler
{
    IHttpHandler GetHttpHandler(RequestContext requestContext);
}

```

`IRouteHandler` 接口的 `GetHttpHandler` 方法具有一个类型为 `RequestContext` 的参数。顾名思义, `RequestContext` 表示当前(HTTP)请求的上下文, 其核心就是对当前 `HttpContext` 和 `RouteData`

的封装，这可以通过如下的代码片段看出来。

```
public class RequestContext
{
    public virtual HttpContextBase HttpContext { get; set; }
    public virtual RouteData RouteData { get; set; }
}
```

2. Route 和 RouteTable

承载路由变量的 `RouteData` 对象由路由表中与当前请求相匹配的 `Route` 对象生成，可以通过 `RouteData` 的 `Route` 属性获得这个 `Route` 对象，该属性的类型为 `RouteBase`。如下面的代码片段所示，`RouteBase` 是一个抽象类，它仅仅包含一个返回类型为 `RouteData` 的 `GetRouteData` 方法。

```
public abstract class RouteBase
{
    public abstract RouteData GetRouteData(HttpContextBase httpContext);
}
```

`RouteBase` 的 `GetRouteData` 方法具有一个类型为 `HttpContextBase` 的参数，它代表针对当前接收请求的 HTTP 上下文。当该方法被执行的时候，它会判断自身定义的路由规则是否与当前请求相匹配，并在成功匹配的情况下实施路由解析，并将得到的路由变量封装成 `RouteData` 对象返回。如果路由规则与当前请求不匹配，则该方法直接返回 `Null`。

我们定义了如下一个继承自 `RouteBase` 的 `Route` 类型来完成具体的路由工作。如下面的代码片段所示，一个 `Route` 对象²具有一个代表路由模板的字符串类型的 `Url` 属性。在实现的 `GetRouteData` 方法中，我们通过 `HttpContextBase` 获取当前请求的 URL，如果它与路由模板的模式相匹配，则创建一个 `RouteData` 对象作为该方法的返回值。对于返回的 `RouteData` 对象，其 `Values` 属性表示的字典对象包含直接通过 URL 解析出来的变量，而对于 `DataTokens` 字典和 `RouteHandler` 属性，则直接取自 `Route` 对象的同名属性。

```
public class Route : RouteBase
{
    public IRouteHandler RouteHandler { get; set; }
    public string Url { get; set; }
    public IDictionary<string, object> DataTokens { get; set; }

    public Route()
    {
    }
}
```

² 在本书中很多名词术语都是泛指，比如在大部分章节中的 `Controller` 是指实现了 `IController` 接口的某个类型的对象，而不是类型为 `Controller` 的某个对象。本书中的 `Route` 或者“`Route` 对象”在大部分情况下泛指继承自抽象类 `RouteBase` 的某个类型的对象，不过在这里却是指的具体类型为 `Route` 的某个对象。“泛指某类对象”和“具体某个类型的对象”在大部分情况下可以根据上下文来区分。

```

{
    this.DataTokens          = new Dictionary<string, object>();
    this.RouteHandler        = new MvcRouteHandler();
}

public override RouteData GetRouteData(HttpContextBase httpContext)
{
    IDictionary<string, object> variables;
    if (this.Match(httpContext.Request
        .AppRelativeCurrentExecutionFilePath.Substring(2), out variables))
    {
        RouteData routeData = new RouteData();
        foreach (var item in variables)
        {
            routeData.Values.Add(item.Key, item.Value);
        }
        foreach (var item in DataTokens)
        {
            routeData.DataTokens.Add(item.Key, item.Value);
        }
        routeData.RouteHandler = this.RouteHandler;
        return routeData;
    }
    return null;
}

protected bool Match(string requestUrl,
    out IDictionary<string, object> variables)
{
    variables          = new Dictionary<string, object>();
    string[] strArray1 = requestUrl.Split('/');
    string[] strArray2 = this.Url.Split('/');

    if (strArray1.Length != strArray2.Length)
    {
        return false;
    }

    for (int i = 0; i < strArray2.Length; i++)
    {
        if (strArray2[i].StartsWith("{") && strArray2[i].EndsWith("}"))
        {
            variables.Add(strArray2[i].Trim("{}").ToCharArray(), strArray1[i]);
        }
        else
        {
            if (string.Compare(strArray1[i], strArray2[i], true) != 0)
            {
                return false;
            }
        }
    }
    return true;
}
}

```

一个 Web 应用可以采用多种不同的 URL 模式，所以需要注册多个继承自 `RouteBase` 的 `Route` 对象，多个 `Route` 对象组成了一个路由表。在我们自定义的迷你版 ASP.NET MVC 框架中，路由表通过类型 `RouteTable` 表示。如下面的代码片段所示，`RouteTable` 仅仅具有一个类型为 `RouteDictionary` 的 `Routes` 属性表示针对整个 Web 应用的全局路由表。

```
public class RouteTable
{
    public static RouteDictionary Routes { get; private set; }
    static RouteTable()
    {
        Routes = new RouteDictionary();
    }
}
```

`RouteDictionary` 表示一个具名的 `Route` 对象的列表，我们直接让它继承自泛型的字典类型 `Dictionary<string, RouteBase>`，其中的 `Key` 表示 `Route` 对象的注册名称。在 `GetRouteData` 方法中，我们遍历集合找到与指定的 `HttpContextBase` 对象匹配的 `Route` 对象，并得到对应的 `RouteData`。

```
public class RouteDictionary: Dictionary<string, RouteBase>
{
    public RouteData GetRouteData(HttpContextBase httpContext)
    {
        foreach (var route in this.Values)
        {
            RouteData routeData = route.GetRouteData(httpContext);
            if (null != routeData)
            {
                return routeData;
            }
        }
        return null;
    }
}
```

在 `Global.asax` 中我们创建了一个基于指定路由模板（`{controller}/{action}`）的 `Route` 对象，并将其添加到通过 `RouteTable` 的静态只读属性 `Routes` 所表示的全局路由表中。

```
public class Global : System.Web.HttpApplication
{
    protected void Application_Start(object sender, EventArgs e)
    {
        RouteTable.Routes.Add("default",
            new Route{Url = "{controller}/{action}"});
        //其他操作
    }
}
```


3. UrlRoutingModule

路由表的作用是对当前的 HTTP 请求实施路由解析,进而得到一个以 Controller 和 Action 名称为核心的路由数据,即上面介绍的 RouteData 对象。整个路由解析工作是通过一个类型为 UrlRoutingModule 的自定义 IHttpModule 来完成的。

```
public class UrlRoutingModule: IHttpModule
{
    public void Dispose()
    {}

    public void Init(HttpApplication context)
    {
        context.PostResolveRequestCache += OnPostResolveRequestCache;
    }

    protected virtual void OnPostResolveRequestCache(object sender, EventArgs e)
    {
        HttpContextWrapper httpContext =
            new HttpContextWrapper(HttpContext.Current);
        RouteData routeData = RouteTable.Routes.GetRouteData(httpContext);
        if (null == routeData)
        {
            return;
        }
        RequestContext requestContext = new RequestContext {
            RouteData = routeData, HttpContext = httpContext };
        IHttpHandler handler =
            routeData.RouteHandler.GetHttpHandler(requestContext);
        httpContext.RemapHandler(handler);
    }
}
```

如上面的代码片段所示,我们让 UrlRoutingModule 实现了 IHttpModule 接口。在实现的 Init 方法中,我们注册了 HttpApplication 的 PostResolveRequestCache 事件。当代表当前应用的 HttpApplication 对象的 PostResolveRequestCache 事件触发之后,UrlRoutingModule 通过 RouteTable 的静态只读属性 Routes 得到表示全局路由表的 RouteDictionary 对象,然后根据当前 HTTP 上下文创建一个 HttpContextWrapper 对象(HttpContextWrapper 是 HttpContextBase 的子类),并将其作为参数调用 RouteDictionary 对象的 GetRouteData 方法。

如果方法调用返回一个具体的 RouteData 对象,UrlRoutingModule 会根据该对象本身和之前得到的 HttpContextWrapper 对象创建一个表示当前请求上下文的 RequestContext 对象,并将其作为参数传入 RouteData 的 RouteHandler 的 GetHttpHandler 方法得到一个 IHttpHandler 对象。UrlRoutingModule 最后调用 HttpContextWrapper 对象的 RemapHandler 方法对得到的 IHttpHandler 对象进行映射,那么针对当前 HTTP 请求的后续处理将由这个 IHttpHandler 来接手。

有人可能会问为什么 `UrlRoutingModule` 会选择注册代表当前应用的 `HttpApplication` 对象的 `PostResolveRequestCache` 事件来实施路由呢？实际上在 1.4.1 节已经回答了这个问题，因为 `UrlRoutingModule` 最终的目的是为当前请求映射一个 `HttpHandler` 对象，根据前面对 ASP.NET 管道的介绍我们知道，紧随 `PostResolveRequestCache` 事件被触发的另一个事件是 `PostMapRequestHandler`。如果再晚一步，`HttpHandler` 的动态映射就无法实现了。

1.4.3 Controller 的激活

ASP.NET MVC 的路由系统通过注册的路由表对当前 HTTP 请求实施路由解析，从而得到一个用于封装路由数据的 `RouteData` 对象，这个过程是通过自定义的 `UrlRoutingModule` 对 `HttpApplication` 的 `PostResolveRequestCache` 事件进行注册实现的。由于得到的 `RouteData` 对象中已经包含了目标 Controller 的名称，我们需要根据该名称激活对应的 Controller 对象。

1. MvcRouteHandler

通过前面的介绍我们知道，继承自 `RouteBase` 的 `Route` 类型具有一个类型为 `IRouteHandler` 接口的属性 `RouteHandler`，它主要的用途就是用于根据指定的请求上下文（通过一个 `RequestContext` 对象表示）来获取一个 `HttpHandler` 对象。当 `GetRouteData` 方法被执行后，`Route` 的 `RouteHandler` 属性值将反映在得到的 `RouteData` 的同名属性上。在默认的情况下，`Route` 的 `RouteHandler` 属性是一个 `MvcRouteHandler` 对象，如下的代码片段反映了这一点。

```
public class Route : RouteBase
{
    //其他成员
    public IRouteHandler RouteHandler { get; set; }
    public Route()
    {
        //其他操作
        this.RouteHandler = new MvcRouteHandler();
    }
}
```

对于我们这个“迷你版”的 ASP.NET MVC 框架来说，`MvcRouteHandler` 是一个具有如下定义的类型。如下面的代码片段所示，在实现的 `GetHttpHandler` 方法中它会直接返回一个 `MvcHandler` 对象。

```
public class MvcRouteHandler: IRouteHandler
{
    public IHttpHandler GetHttpHandler(RequestContext requestContext)
    {
```

```

        return new MvcHandler(requestContext);
    }
}

```

2. MvcHandler

在前面的内容中已经不止一次地提到，整个 ASP.NET MVC 框架是通过自定义的 `HttpModule` 和 `HttpHandler` 对 ASP.NET 进行扩展构建起来的。这个自定义的 `HttpModule` 类型已经介绍过了，它就是 `UrlRoutingModule`，而这个自定义的 `HttpHandler` 类型则是需要重点介绍的 `MvcHandler`。

`UrlRoutingModule` 在利用路由表对当前请求实施路由解析并得到封装路由数据的 `RouteData` 对象后，会调用其 `RouteHandler` 的 `GetHttpHandler` 方法得到一个 `HttpHandler` 对象，然后将其映射到当前的 HTTP 上下文。由于 `RouteData` 的 `RouteHandler` 来源于对应 `Route` 对象的 `RouteHandler`，而后者在默认的情况下是一个 `MvcRouteHandler` 对象，所以默认情况下用于处理 HTTP 请求的就是这么一个 `MvcHandler` 对象。`MvcHandler` 实现了对 `Controller` 对象的激活和对目标 `Action` 方法的执行。

如下面的代码片段所示，`MvcHandler` 具有一个类型为 `RequestContext` 的属性，它表示当前请求上下文，该属性在构造函数中指定。`MvcHandler` 在 `ProcessRequest` 方法中实现了对 `Controller` 对象的激活和执行。

```

public class MvcHandler: IHttpHandler
{
    public bool IsReusable
    {
        get{return false;}
    }

    public RequestContext RequestContext { get; private set; }

    public MvcHandler(RequestContext requestContext)
    {
        this.RequestContext = requestContext;
    }

    public void ProcessRequest(HttpContext context)
    {
        string controllerName = this.RequestContext.RouteData.Controller;
        IControllerFactory controllerFactory =
            ControllerBuilder.Current.GetControllerFactory();
        IController controller = controllerFactory.CreateController(
            this.RequestContext, controllerName);
        controller.Execute(this.RequestContext);
    }
}

```

Controller 与 ControllerFactory

我们为 Controller 定义了一个接口 `IController`。如下面的代码片段所示，该接口具有唯一的方法 `Execute` 表示对当前 Controller 对象的执行。该方法在 `MvcHandler` 的 `ProcessRequest` 方法中被调用，而传入该方法的参数是表示当前请求上下文的 `RequestContext` 对象。

```
public interface IController
{
    void Execute(RequestContext requestContext);
}
```

从 `MvcHandler` 的定义可以看到，Controller 对象的激活是通过工厂模式实现的，我们为激活 Controller 的工厂定义了一个 `IControllerFactory` 接口。如下面的代码片段所示，该接口具有唯一的方法 `CreateController`，该方法根据当前请求上下文和通过路由解析得到的目标 Controller 的名称激活相应的 Controller 对象。

```
public interface IControllerFactory
{
    IController CreateController(RequestContext requestContext,
                                string controllerName);
}
```

在 `MvcHandler` 的 `ProcessRequest` 方法中，它通过 `ControllerBuilder` 的静态属性 `Current` 得到当前的 `ControllerBuilder` 对象，并调用其 `GetControllerFactory` 方法获得当前的 `ControllerFactory`。接下来 `MvcHandler` 通过从 `RequestContext` 中提取的 `RouteData` 对象获得目标 Controller 的名称，最后将它连同 `RequestContext` 一起作为参数调用 `ControllerFactory` 的 `CreateController` 方法实现对目标 Controller 对象的创建。

`ControllerBuilder` 的整个定义如下面的代码片段所示，表示当前 `ControllerBuilder` 的静态只读属性 `Current` 在静态构造函数中被创建，其 `SetControllerFactory` 和 `GetControllerFactory` 方法用于 `ControllerFactory` 的注册和获取。

```
public class ControllerBuilder
{
    private Func<IControllerFactory> factoryThunk;
    public static ControllerBuilder Current { get; private set; }

    static ControllerBuilder()
    {
        Current = new ControllerBuilder();
    }

    public IControllerFactory GetControllerFactory()
    {
        return factoryThunk();
    }
}
```

```

    }

    public void SetControllerFactory(IControllerFactory controllerFactory)
    {
        factoryThunk = () => controllerFactory;
    }
}

```

再回头看看之前建立在自定义 ASP.NET MVC 框架的 Web 应用，我们就是通过当前的 **ControllerBuilder** 来注册 **ControllerFactory**。如下面的代码片段所示，注册的 **ControllerFactory** 的类型为 **DefaultControllerFactory**。

```

public class Global : System.Web.HttpApplication
{
    protected void Application_Start(object sender, EventArgs e)
    {
        //其他操作
        ControllerBuilder.Current.SetControllerFactory(
            new DefaultControllerFactory());
    }
}

```

作为默认 **ControllerFactory** 的 **DefaultControllerFactory** 类型定义如下代码片段所示。由于激活 **Controller** 对象的前提是能够正确解析出 **Controller** 的真实类型，作为 **CreateController** 方法输入参数的 **controllerName** 仅仅表示 **Controller** 的名称，所以我们需要加上 **Controller** 字符后缀作为类型名称。在 **DefaultControllerFactory** 类型被加载的时候（静态构造函数被调用），它通过 **BuildManager** 加载所有被引用的程序集，得到所有实现了接口 **IController** 的类型并将其缓存起来。在 **CreateController** 方法中，**DefaultControllerFactory** 根据 **Controller** 的名称从保存的 **Controller** 类型列表中得到对应的 **Controller** 类型，并通过反射的方式创建它。

```

public class DefaultControllerFactory : IControllerFactory
{
    private static List<Type> controllerTypes = new List<Type>();

    static DefaultControllerFactory()
    {
        foreach (Assembly assembly in BuildManager.GetReferencedAssemblies())
        {
            foreach (Type type in assembly.GetTypes().Where(
                type => typeof(IController).IsAssignableFrom(type)))
            {
                controllerTypes.Add(type);
            }
        }
    }

    public IController CreateController(RequestContext requestContext,
        string controllerName)

```

```

    {
        string typeName = controllerName + "Controller";
        Type controllerType = controllerTypes.FirstOrDefault(
            c => string.Compare(typeName, c.Name, true) == 0);
        if (null == controllerType)
        {
            return null;
        }
        return (IController)Activator.CreateInstance(controllerType);
    }
}

```

上面我们详细地介绍了 **Controller** 的激活原理，现在将关注点返回到 **Controller** 自身。我们通过实现 **IController** 接口为所有的 **Controller** 定义了一个具有如下定义的 **ControllerBase** 抽象基类，从中可以看到在实现的 **Execute** 方法中 **ControllerBase** 通过一个实现了接口 **IActionInvoker** 的对象完成了针对 **Action** 方法的执行。

```

public abstract class ControllerBase: IController
{
    protected IActionInvoker ActionInvoker { get; set; }

    public ControllerBase()
    {
        this.ActionInvoker = new ControllerActionInvoker();
    }

    public void Execute(RequestContext requestContext)
    {
        ControllerContext context = new ControllerContext {
            RequestContext = requestContext, Controller = this };
        string actionName = requestContext.RouteData.ActionName;
        this.ActionInvoker.InvokeAction(context, actionName);
    }
}

```

1.4.4 Action 的执行

作为 **Controller** 的基类 **ControllerBase**，它的 **Execute** 方法主要作用在于执行目标 **Action** 方法。如果目标 **Action** 方法返回一个 **ActionResult** 对象，它还需要执行该对象来对当前请求予以响应。在 **ASP.NET MVC** 框架中，两者的执行是通过一个叫作 **ActionInvoker** 的对象来完成的。

1. ActionInvoker

我们同样为 **ActionInvoker** 定义了一个接口 **IActionInvoker**。如下面的代码片段所示，该接口定义了唯一的方法 **InvokeAction** 用于执行指定名称的 **Action** 方法，该方法的第一个参数是一

个表示针对当前 Controller 上下文的 ControllerContext 对象。

```
public interface IActionInvoker
{
    void InvokeAction(ControllerContext controllerContext, string actionName);
}
```

ControllerContext 类型在真正的 ASP.NET MVC 框架中要复杂一些，在这里我们对它进行了简化，仅仅将它表示成对当前 Controller 和请求上下文的封装。如下面的代码片段所示，这两个要素分别通过 Controller 和 RequestContext 属性来表示。

```
public class ControllerContext
{
    public ControllerBase Controller { get; set; }
    public RequestContext RequestContext { get; set; }
}
```

ControllerBase 中表示 ActionInvoker 的同名属性在构造函数中被初始化。在 Execute 方法中，它通过作为方法参数的 RequestContext 对象创建一个 ControllerContext 对象，并通过包含在 RequestContext 中的 RouteData 得到目标 Action 的名称，最后将这两者作为参数调用 ActionInvoker 的 InvokeAction 方法。从前面给出的关于 ControllerBase 的定义中可以看到，在构造函数中默认创建的 ActionInvoker 是一个类型为 ControllerActionInvoker 的对象。

```
public class ControllerActionInvoker : IActionInvoker
{
    public IModelBinder ModelBinder { get; private set; }
    public ControllerActionInvoker()
    {
        this.ModelBinder = new DefaultModelBinder();
    }
    public void InvokeAction(ControllerContext controllerContext,
        string actionName)
    {
        //省略实现
    }
}
```

InvokeAction 方法的目的在于实现针对 Action 方法的执行，由于 Action 方法具有相应的参数，在执行 Action 方法之前必须进行参数的绑定。ASP.NET MVC 将这个机制称为 Model 绑定，而这又涉及另一个名为 ModelBinder 的对象。如上面的代码片段所示，ControllerActionInvoker 的 ModelBinder 属性返回这么一个 ModelBinder 对象。

2. ModelBinder

我们为 ModelBinder 实现的接口 IModelBinder 提供了一个简单的定义，这与在真正的

ASP.NET MVC 中的同名接口的定义不尽相同。如下面的代码片段所示，该接口具有唯一的 BindModel 方法，它根据 ControllerContext、Model 名称（在这里实际上是参数名称）和类型得到一个作为参数的对象。

```
public interface IModelBinder
{
    object BindModel(ControllerContext controllerContext, string modelName,
        Type modelType);
}
```

通过前面给出的关于 ControllerActionInvoker 的定义可以看到，在构造函数中默认创建的 ModelBinder 是一个 DefaultModelBinder 对象。由于我们仅仅是对 ASP.NET MVC 真实框架的简单模拟，定义在自定义的 DefaultModelBinder 中的 Model 绑定逻辑比真实 ASP.NET MVC 框架中的 DefaultModelBinder 要简单得多，很多复杂的 Model 绑定机制并未在我们自定义的 DefaultModelBinder 中体现出来。

如下面的代码片段所示，绑定到参数上的数据具有 4 个来源，即提交的表单、请求查询字符串、RouteData 的 Values 和 DataTokens 属性，它们都是字典结构的数据集合。如果参数类型为字符串或者简单的值类型，我们可以直接根据参数名称和 Key 进行匹配。对于复杂类型（比如本例中需要绑定的参数类型 SimpleModel），则先根据提供的数据类型采用反射的方式创建一个空对象，然后根据属性名与 Key 的匹配关系提供相应的数据并对属性进行赋值。

```
public class DefaultModelBinder : IModelBinder
{
    public object BindModel(ControllerContext controllerContext,
        string modelName, Type modelType)
    {
        if (modelType.IsValueType || typeof(string) == modelType)
        {
            object instance;
            if (GetValueTypeInstance(controllerContext, modelName, modelType,
                out instance))
            {
                return instance;
            };
            return Activator.CreateInstance(modelType);
        }

        object modelInstance = Activator.CreateInstance(modelType);
        foreach (PropertyInfo property in modelType.GetProperties())
        {
            if (!property.CanWrite || (!property.PropertyType.IsValueType &&
```



```

        property.PropertyType != typeof(string)))
    {
        continue;
    }
    object propertyValue;
    if (GetValueTypeInstance(controllerContext, property.Name,
        property.PropertyType, out propertyValue))
    {
        property.SetValue(modelInstance, propertyValue, null);
    }
}
return modelInstance;
}

private bool GetValueTypeInstance(ControllerContext controllerContext,
    string modelName, Type modelType, out object value)
{
    Dictionary<string, object> dataSource =
        new Dictionary<string, object>();

    //数据来源一: HttpContext.Current.Request.Form
    foreach (string key in HttpContext.Current.Request.Form)
    {
        if (dataSource.ContainsKey(key.ToLower()))
        {
            continue;
        }
        dataSource.Add(key.ToLower(),
            HttpContext.Current.Request.Form[key]);
    }

    //数据来源二: HttpContext.Current.Request.QueryString
    foreach (string key in HttpContext.Current.Request.QueryString)
    {
        if (dataSource.ContainsKey(key.ToLower()))
        {
            continue;
        }
        dataSource.Add(key.ToLower(),
            HttpContext.Current.Request.QueryString[key]);
    }

    //数据来源三: ControllerContext.RequestContext.RouteData.Values
    foreach (var item in
        controllerContext.RequestContext.RouteData.Values)
    {
        if (dataSource.ContainsKey(item.Key.ToLower()))
        {
            continue;
        }
    }
}

```

```

    }
    dataSource.Add(item.Key.ToLower(),
        controllerContext.RequestContext.RouteData.Values[item.Key]);
}

//数据来源四: ControllerContext.RequestContext.RouteData.DataTokens
foreach (var item in
    controllerContext.RequestContext.RouteData.DataTokens)
{
    if (dataSource.ContainsKey(item.Key.ToLower()))
    {
        continue;
    }
    dataSource.Add(item.Key.ToLower(),
        controllerContext.RequestContext.RouteData
            .DataTokens[item.Key]);
}

if (dataSource.TryGetValue(modelName.ToLower(), out value))
{
    value = Convert.ChangeType(value, modelType);
    return true;
}
return false;
}
}

```

3. ControllerActionInvoker

实现了 `IActionInvoker` 接口的 `ControllerActionInvoker` 是默认使用的 `ActionInvoker`。如下面的代码片段所示，在实现的 `InvokeAction` 方法中，我们根据 `Action` 的名称得到用于描述对应方法的 `MethodInfo` 对象，进而得到描述所有参数的 `ParameterInfo` 列表。针对每个 `ParameterInfo` 对象，我们借助 `ModelBinder` 对象采用 `Model` 绑定的方式从当前请求中获取源数据并生成相应的参数对象。

```

public class ControllerActionInvoker : IActionInvoker
{
    public IModelBinder ModelBinder { get; private set; }
    public ControllerActionInvoker()
    {
        this.ModelBinder = new DefaultModelBinder();
    }

    public void InvokeAction(ControllerContext controllerContext,
        string actionName)
    {

```

```

MethodInfo methodInfo = controllerContext.Controller
    .GetType().GetMethods().First(
        m => string.Compare(actionName, m.Name, true) == 0);
List<object> parameters = new List<object>();
foreach (ParameterInfo parameter in methodInfo.GetParameters())
{
    parameters.Add(this.ModelBinder.BindModel(controllerContext,
        parameter.Name, parameter.ParameterType));
}
ActionExecutor executor = new ActionExecutor(methodInfo);
ActionResult actionResult = (ActionResult)executor.Execute(
    controllerContext.Controller, parameters.ToArray());
actionResult.ExecuteResult(controllerContext);
}
}

```

接下来我们创建一个类型为 `ActionExecutor` 的对象，并将激活的 `Controller` 对象（对应于当前 `ControllerContext` 的 `Controller` 属性）和通过 `Model` 绑定生成的参数列表作为输入参数调用这个 `ActionExecutor` 对象的 `Execute` 方法，目标 `Action` 方法最终得以执行。

4. ActionExecutor

目标 `Action` 方法的执行最终是由 `ActionExecutor` 来完成的，那么它具体采用怎样的方法执行策略呢？虽然 `ActionExecutor` 是根据描述目标 `Action` 方法的 `MethodInfo` 来创建的，它完全可以采用反射的方式来执行此方法。但是为了获得更高的性能，它并没有这么做。目标 `Action` 方法的执行最终是采用“表达式树”的方式来完成的。

我们可以利用表达式树将一段代码表示成一种树状的数据结构，这个表达式可以被编译成可执行代码。基于表达式树对目标 `Action` 方法的执行实现在 `ActionExecutor` 的 `Execute` 方法中。如下面的代码片段所示，我们根据描述被执行 `Action` 方法的 `MethodInfo` 对象来创建 `ActionExecutor` 对象，并在静态方法 `CreateExecutor` 中根据这个 `MethodInfo` 对象来构建用于执行目标方法的表达式树并对其进行编译生成一个 `Func<object, object[], object>` 类型的委托对象。目标 `Action` 方法的执行最终由此委托对象来完成。

```

internal class ActionExecutor
{
    private static Dictionary<MethodInfo, Func<object, object[], object>>
        executors =
        new Dictionary<MethodInfo, Func<object, object[], object>>();
    private static object syncHelper = new object();
    public MethodInfo MethodInfo { get; private set; }

    public ActionExecutor(MethodInfo methodInfo)

```

```

{
    this.MethodInfo = methodInfo;
}

public object Execute(object target, object[] arguments)
{
    Func<object, object[], object> executor;
    if (!executors.TryGetValue(this.MethodInfo, out executor))
    {
        lock (syncHelper)
        {
            if (!executors.TryGetValue(this.MethodInfo, out executor))
            {
                executor = CreateExecutor(this.MethodInfo);
                executors[this.MethodInfo] = executor;
            }
        }
    }
    return executor(target, arguments);
}

private static Func<object, object[], object> CreateExecutor(
    MethodInfo methodInfo)
{
    ParameterExpression target = Expression.Parameter(
        typeof(object), "target");
    ParameterExpression arguments = Expression.Parameter(
        typeof(object[]), "arguments");

    List<Expression> parameters = new List<Expression>();
    ParameterInfo[] paramInfos = methodInfo.GetParameters();
    for (int i = 0; i < paramInfos.Length; i++)
    {
        ParameterInfo paramInfo = paramInfos[i];
        BinaryExpression getElementByIndex =
            Expression.ArrayIndex(arguments, Expression.Constant(i));
        UnaryExpression convertToParameterType = Expression.Convert(
            getElementByIndex, paramInfo.ParameterType);
        parameters.Add(convertToParameterType);
    }

    UnaryExpression instanceCast = Expression.Convert(target,
        methodInfo.ReflectedType);
    MethodCallExpression methodCall =
        Expression.Call(instanceCast, methodInfo, parameters);
    UnaryExpression convertToObjectType = Expression.Convert(
        methodCall, typeof(object));
    return Expression.Lambda<Func<object, object[], object>>(

```

```

        convertToObjectType, target, arguments).Compile();
    }
}

```

`ActionExecutor` 对象的 `Execute` 方法执行之后返回的对象代表执行目标 `Action` 方法的返回值，假设这个返回值总是一个 `ActionResult` 对象（ASP.NET MVC 对 `Action` 方法的返回类型未作任何限制），所以我们会直接将其转换成 `ActionResult` 类型并调用其 `ExecuteResult` 方法对请求作最终的响应。

5. ActionResult

我们为具体的 `ActionResult` 定义了一个 `ActionResult` 抽象基类。如下面的代码片段所示，该抽象类具有一个参数类型为 `ControllerContext` 的抽象方法 `ExecuteResult`，我们最终对请求的响应就实现在该方法中。

```

public abstract class ActionResult
{
    public abstract void ExecuteResult(ControllerContext context);
}

```

在之前创建的例子中，`Action` 方法返回的是一个类型为 `RawContentResult` 的对象。顾名思义，`RawContentResult` 旨在将我们写入的内容原封不动地呈现出来。如下面的代码片段所示，`RawContentResult` 具有一个 `Action<TextWriter>` 类型的只读属性 `Callback`，我们利用它来写入需要呈现的内容。在实现的 `ExecuteResult` 方法中，我们对这个 `Action<TextWriter>` 对象予以执行，而作为参数的正是当前 `HttpResponse` 的 `Output` 属性表示的 `TextWriter` 对象，毫无疑问通过 `Action<TextWriter>` 对象写入的内容将最终作为响应返回到客户端。

```

public class RawContentResult : ActionResult
{
    public Action<TextWriter> Callback { get; private set; }

    public RawContentResult(Action<TextWriter> callback)
    {
        this.Callback = callback;
    }

    public override void ExecuteResult(ControllerContext context)
    {
        this.Callback(context.RequestContext.HttpContext.Response.Output);
    }
}

```

1.4.5 完整的流程

对于我们创建的这个迷你版本的 ASP.NET MVC 框架来说，虽然很多细节被直接忽略掉，但是它基本上能够展现整个 ASP.NET MVC 框架的全貌，支持这个开发框架的核心对象可以说是一个不少。接下来我们对通过这个模拟框架展现出来的 ASP.NET MVC 针对请求的处理流程作一个简单的概括。如图 1-13 所示的 UML 基本上展现了 ASP.NET MVC 从“接收请求”到“响应回复”的完整流程。

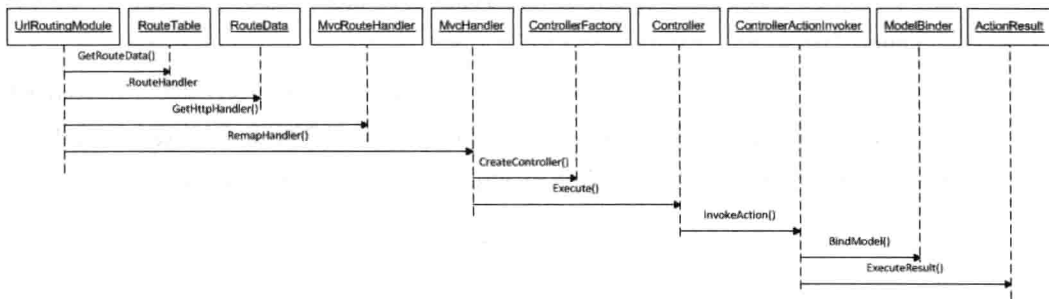


图 1-13 “接收请求”到“响应回复”的完整流程

由于 **UrlRoutingModule** 这个 **HttpModule** 被注册到 Web 应用中，所以对于每个抵达的请求来说，当代表当前应用的 **HttpApplication** 对象的 **PostResolveRequestCache** 事件被触发的时候，**UrlRoutingModule** 会利用 **RouteTable** 表示的路由表（实际上 **RouteTable** 的静态属性 **Routes** 返回的 **RouteDictionary** 对象代表这个路由表）针对当前请求实施路由解析。

具体来说，**UrlRoutingModule** 会调用代表路由表的 **RouteDictionary** 对象的 **GetRouteData** 方法，如果定义在某个 **Route** 对象上的路由规则与当前请求相匹配，那么该方法执行结束之后会返回一个 **RouteData** 对象，包含目标 **Controller** 和 **Action** 名称的路由变量被包含在这个 **RouteData** 对象之中。

接下来 **UrlRoutingModule** 通过 **RouteData** 对象的 **RouteHandler** 属性得到匹配 **Route** 对象采用的 **RouteHandler** 对象，在默认情况下这是一个 **MvcRouteHandler** 对象。**UrlRoutingModule** 随后会调用这个 **MvcRouteHandler** 对象的 **GetHttpHandler** 方法得到一个 **HttpHandler** 对象。对于 **MvcRouteHandler** 来说，它的 **GetHttpHandler** 方法具体返回的是一个 **MvcHandler** 对象。**UrlRoutingModule** 随之调用当前 HTTP 上下文的 **MapHttpHandler** 方法对得到的 **HttpHandler** 对象实施映射，那么此 **HttpHandler** 将最终接管当前请求的处理。

对于 **MvcHandler** 来说，当它被用来处理当前请求的时候，它会利用 **RouteData** 对象得到目

标 Controller 的名称，并借助于注册的 ControllerFactory 来激活对应的 Controller 对象。目标 Controller 被激活之后，它的 Execute 方法被 MvcHandler 调用。

如果被激活的 Controller 对象的类型是 ControllerBase 的子类，当它的 Execute 方法被执行的时候，它会调用 ActionInvoker 对象的 InvokeAction 方法来执行目标 Action 方法并对当前请求予以响应。默认采用的 ActionInvoker 是一个 ControllerActionInvoker 对象，当它的 InvokeAction 方法被执行的时候，它会利用注册的 ModelBinder 采用 Model 绑定的方式生成目标 Action 方法的参数列表，并利用 ActionExecutor 对象以“表达式树”的方式执行目标 Action 方法。

目标 Action 方法执行之后总是会返回一个 ActionResult（对于返回类型不是 ActionResult 的 Action 方法来说，ASP.NET MVC 总是会将执行的结果转换成一个 ActionResult 对象），ControllerActionInvoker 会通过执行此 ActionResult 对象来对请求作最终的响应。

第2章 路由

对于传统的 ASP.NET Web Forms 应用来说，用户请求总是指向某个具体的物理文件，目标文件的路径决定了访问请求的 URL。但是对于 ASP.NET MVC 应用来说，来自浏览器的请求总是指向定义在某个 Controller 类型中的某个 Action 方法，请求 URL 与目标 Controller/Action 之间的映射是通过“路由”来实现的。

2.1 ASP.NET 路由

由于来自客户端的请求总是指向定义在某个 Controller 类型中的某个 Action 方法，并且目标 Controller 和 Action 的名称由请求 URL 决定，所以必须采用某种机制根据请求 URL 解析出目标 Controller 和 Action 的名称，我们将这种机制称为“路由（Routing）”。但是路由系统并不是专属于 ASP.NET MVC 的，而是直接建立在 ASP.NET 上（实现路由的核心类型基本上定义在程序集“System.Web.dll”中）。路由机制同样可以应用在 Web Forms 应用中，它可以帮助我们实现请求地址与物理文件的分离。

2.1.1 请求 URL 与物理文件的分离

对于一个 ASP.NET Web Forms 应用来说，通常情况下一个有效的请求都对应着一个具体的物理文件。部署在 Web 服务器上的物理文件可以是静态的（比如图片和静态 HTML 文件等），也可以是动态的（比如.aspx 页面）。对于静态文件的请求，ASP.NET 会直接返回文件的原始内容，而针对动态文件的请求则会涉及相关代码的执行。这种将 URL 与物理文件紧密绑定在一起的方式并不是一种好的解决方案，它带来的局限性主要体现在如下几个方面。

- 灵活性。物理文件的路径决定了访问它的 URL，如果物理文件的路径发生了改变（比如改变了文件的目录结构或者文件名），原来访问该文件的 URL 将变得无效。
- 可读性。在很多情况下，URL 不仅仅具备基本的可用性（能够访问正确的网络资源），还需要具有很好的可读性。好的 URL 设计应该让我们一眼就能看出针对它访问的目标资源是什么，请求地址与物理文件紧密绑定让我们完全失去了设计高可读性 URL 的机会。
- SEO 优化。对于网站开发来说，为了迎合搜索引擎检索的规则，我们需要对 URL 进行有效的设计，使之能易于被主流的引擎检索收录。如果 URL 完全与物理地址关联在一起，这无异于失去了 SEO 优化的能力。

上述 3 个因素促使我们不得不采用一种更加灵活的映射机制来实现请求 URL 与目标文件路径的分离。那么有什么办法能够帮助实现两者之间的分离呢？可能很多人会想到一个叫作“URL 重写”的机制。为了使 Web 应用可以独立地设计用于访问应用资源的 URL，微软为 IIS 7 编写了一个 URL 重写模块。这是一个基于规则的 URL 重写引擎，它在 URL 被 Web 服务器处理之前根据定义的规则重定向某个物理文件。

URL 重写机制在 IIS 级别解决了 URL 与物理地址的分离，它的实现依赖于一个注册到 IIS 管道上的本地（Native）代码模块，所以它可以应用于寄宿在 IIS 中的所有 Web 应用类型。与 URL 重写机制不同，路由系统则是 ASP.NET 的一部分，并且是通过托管代码编写的。为了让读者对 ASP.NET 的路由系统具有一个感官的认识，我们来演示一个简单的实例。

2.1.2 实例演示：通过路由实现请求地址与.aspx 页面的映射（S201）

我们创建一个简单的 ASP.NET Web Forms 应用，并采用一套独立于.aspx 文件路径的 URL 来访问对应的 Web 页面，两者之间的映射通过路由来实现。我们依然沿用第 1 章关于员工管理的场景并创建一个页面来显示员工的列表和某个员工的详细信息，呈现效果如图 2-1 所示。



图 2-1 员工列表和员工详细信息页面

我们将关注点放到如图 2-1 所示的两个页面的 URL 上，用于显示员工列表的页面地址为“/employees”，当用户单击某个显示为姓名的链接后，用于显示所选员工详细信息的页面被呈现出来，其页面地址的 URL 模式为“/employees/{姓名}/{ID}”。对于后者，最终用户一眼就可以从 URL 中看出通过该地址获取的是哪个员工的信息。

有人可能会问，为什么我们要在 URL 中同时包含员工的姓名和 ID 呢？这是因为 ID（本例采用 GUID）的可读性不如员工姓名，但是员工姓名不具有唯一性，所以在这里我们使用的 ID 是为了逻辑处理的需要而提供的唯一标识，而姓名则是出于可读性的诉求。

我们将员工的所有信息(ID、姓名、性别、出生日期和所在部门)定义在如下所示的 `Employee` 类型中,它与我们在第1章“ASP.NET + MVC”中演示 Model 2 模式中的同名类型具有一致的定义。我们照例定义了如下一个 `EmployeeRepository` 类型来维护员工列表的数据。简单起见,员工列表通过静态字段 `employees` 表示。`EmployeeRepository` 的 `GetEmployees` 方法根据指定的 ID 返回对应的员工。如果指定的 ID 为 “*”,则会返回所有员工列表。

```
public class Employee
{
    public string      Id { get; private set; }
    public string      Name { get; private set; }
    public string      Gender { get; private set; }
    public DateTime    BirthDate { get; private set; }
    public string      Department { get; private set; }

    public Employee(string id, string name, string gender, DateTime birthDate,
        string department)
    {
        this.Id      = id;
        this.Name    = name;
        this.Gender  = gender;
        this.BirthDate = birthDate;
        this.Department = department;
    }
}

public class EmployeeRepository
{
    private static IList<Employee> employees;
    static EmployeeRepository()
    {
        employees = new List<Employee>();
        employees.Add(new Employee(Guid.NewGuid().ToString(), "张三", "男",
            new DateTime(1981, 8, 24), "销售部"));
        employees.Add(new Employee(Guid.NewGuid().ToString(), "李四", "女",
            new DateTime(1982, 7, 10), "人事部"));
        employees.Add(new Employee(Guid.NewGuid().ToString(), "王五", "男",
            new DateTime(1981, 9, 21), "人事部"));
    }
    public IEnumerable<Employee> GetEmployees(string id = "")
    {
        return employees.Where(e => e.Id == id || string.IsNullOrEmpty(id) ||
            id=="*");
    }
}
```

如图 2-1 所示的两个页面实际上对应着同一个.aspx 文件,即作为 Web 应用默认页面的 `Default.aspx`。要通过一个独立于物理路径的 URL 来访问该.aspx 页面,就需要利用路由来实现两者之间的映射。我们将实现映射的路由注册代码定义在 `Global.asax` 文件中。如下面的代码片

段所示，我们在 `Application_Start` 方法中通过 `RouteTable` 的 `Routes` 属性得到表示全局路由表的 `RouteCollection` 对象，并调用其 `MapPageRoute` 方法将 `Default.aspx` 页面的路径（`~/default.aspx`）与一个路由模板（`employees/{name}/{id}`）进行了映射。

```
public class Global : System.Web.HttpApplication
{
    protected void Application_Start(object sender, EventArgs e)
    {
        var defaults = new RouteValueDictionary{{"name", "*"}, {"id", "*"} };
        RouteTable.Routes.MapPageRoute("", "employees/{name}/{id}",
            "~/default.aspx", true, defaults);
    }
}
```

作为 `MapPageRoute` 方法最后一个参数的 `RouteValueDictionary` 对象用于提供定义在路由模板中两个路由变量（“`{name}`”和“`{id}`”）的默认值。如果我们为定义路由模板中的路由变量指定了默认值，在当前请求地址的后续部分缺失的情况下，路由系统会采用提供的默认值对该地址进行填充之后再行模式匹配。在如上所示的代码片段中，我们将 `{name}` 和 `{id}` 两个变量的默认值均指定为“*”。对于针对 URL 为“`/employees`”的请求，注册的 `Route` 会将其格式化成“`/employees/*/*`”，后者无疑与定义的路由模板的模式相匹配。

我们在 `Default.aspx` 页面中分别采用 `GridView` 和 `DetailsView` 来显示所有员工列表和某个列表的详细信息，下面的代码片段表示该页面主体部分的 HTML。`GridView` 模板中显示为员工姓名的 `HyperLinkField` 的链接采用了上面定义在模板（`employees/{name}/{id}`）中的模式。

```
<form id="form1" runat="server">
    <div id="page">
        <asp:GridView ID="GridViewEmployees"
            runat="server" AutoGenerateColumns="false" Width="100%">
            <Columns>
                <asp:HyperLinkField HeaderText="姓名" DataTextField="Name"
                    DataNavigateUrlFields="Name,Id"
                    DataNavigateUrlFormatString="~/employees/{0}/{1}" />
                <asp:BoundField DataField="Gender" HeaderText="性别" />
                <asp:BoundField DataField="BirthDate" HeaderText="出生日期"
                    DataFormatString="{0:dd/MM/yyyy}" />
                <asp:BoundField DataField="Department" HeaderText="部门" />
            </Columns>
        </asp:GridView>
        <asp:DetailsView ID="DetailsViewEmployee" runat="server"
            AutoGenerateRows="false" Width="100%">
            <Fields>
                <asp:BoundField DataField="ID" HeaderText="ID" />
                <asp:BoundField DataField="Name" HeaderText="姓名" />
            </Fields>
        </asp:DetailsView>
    </div>
</form>
```

```

        <asp:BoundField DataField="Gender" HeaderText="性别" />
        <asp:BoundField DataField="BirthDate" HeaderText="出生日期"
            DataFormatString="{0:dd/MM/yyyy}" />
        <asp:BoundField DataField="Department" HeaderText="部门" />
    </Fields>
</asp:DetailsView>
</div>
</form>

```

由于所有员工列表和单一员工的详细信息均体现在这个页面中，所以我们需要根据具体的请求地址来判断应该呈现怎样的数据，这是通过代表当前页面的 `Page` 对象的 `RouteData` 属性来实现的。`Page` 类型的 `RouteData` 属性返回一个 `RouteData` 对象，它表示路由系统对当前请求进行解析得到的路由数据。`RouteData` 的 `Values` 属性是一个存储路由变量的字典，其 `Key` 为变量名称。在如下所示的代码片段中，我们得到表示员工 `ID` 的路由变量（`RouteData.Values["id"]`），如果它是默认值（`*`）就表示当前请求是针对员工列表的，反之则是针对指定的某个具体员工。

```

public partial class Default : Page
{
    private EmployeeRepository repository;

    public EmployeeRepository Repository
    {
        get { return repository ?? (repository = new EmployeeRepository()); }
    }

    protected void Page_Load(object sender, EventArgs e)
    {
        if (this.IsPostBack)
        {
            return;
        }
        string employeeId = this.RouteData.Values["id"] as string;
        if (employeeId == "*" || string.IsNullOrEmpty(employeeId))
        {
            this.GridViewEmployees.DataSource = this.Repository.GetEmployees();
            this.GridViewEmployees.DataBind();
            this.DetailsViewEmployee.Visible = false;
        }
        else
        {
            var employees = this.Repository.GetEmployees(employeeId);
            this.DetailsViewEmployee.DataSource = employees;
            this.DetailsViewEmployee.DataBind();
            this.GridViewEmployees.Visible = false;
        }
    }
}

```

2.1.3 Route 与 RouteTable

ASP.NET 路由系统的核心是注册的 Route 对象，一个 Route 对象对应着一个路由模板。多个具有不同 URL 模式的 Route 对象可以注册到同一个 Web 应用中，它们构成了一个路由表。这个包含所有注册 Route 对象的路由表通过 RouteTable 类（该类型定义在命名空间“System.Web.Routing”下，如果未作特别说明，本节介绍的构成 ASP.NET 路由系统的所有类型均定义在此命名空间下）的静态属性 Routes 表示，该属性返回一个 RouteCollection 对象。在上面演示的实例中，我们正是通过调用此 RouteCollection 对象的 MapPageRoute 方法将某个物理文件路径映射到一个路由模板上。

1. RouteBase

我们所说的 Route 泛指的是继承自抽象类 RouteBase 的某个类型的对象。如下面的代码片段所示，RouteBase 具有两个返回类型分别为 RouteData 和 VirtualPathData 的方法 GetRouteData 和 GetVirtualPath，它们分别体现了针对两个“方向”的路由。实现在 GetRouteData 方法中的路由解析是为了获取路由数据，而 GetVirtualPath 方法则通过路由解析生成一个完整的虚拟路径。

```
public abstract class RouteBase
{
    public abstract RouteData GetRouteData(HttpContextBase httpContext);
    public abstract VirtualPathData GetVirtualPath(RequestContext requestContext,
        RouteValueDictionary values);

    public bool RouteExistingFiles { get; set; }
}
```

借助于路由，我们可以采用一个与路径无关的 URL 来访问某个物理文件。但是如果我们就是希望以物理路径的方式来访问对应的物理文件，那该怎么办呢？以上面演示的实例来说，我们注册了一个路由模板为“/employees/{name}/{id}”的 Route，但是我们在目录“/employees/hr/”下放置了一个名为 Default.aspx 的页面用于显示 HR 部门的员工信息。对于这样一个 URL“/employees/hr/default.aspx”，它与注册路由对象的模板是完全匹配的，如果 ASP.NET 总是对采用此 URL 的请求实施路由，则意味着我们不能以真实的物理路径来访问这个页面了。

为了解决这个问题，RouteBase 定义了一个布尔类型的属性 RouteExistingFiles，它表示是否对现有的物理文件实施路由。该属性的默认值为 True，意味着默认情况下在我们的实例中通过地址“/employees/hr/default.aspx”是访问不到 Default.aspx 页面文件的。

2. RouteData

我们现在来看看用于封装路由数据同时作为 `GetRouteData` 方法返回值的 `RouteData`。下面的代码片段所示, `RouteData` 具有一个类型为 `RouteBase` 的属性 `Route`, 该属性返回生成此 `RouteData` 的 `Route` 对象。不过这是一个可读/写的属性, 我们可以使用任意一个 `Route` 对象来对此属性进行赋值。

```
public class RouteData
{
    public RouteData();
    public RouteData(RouteBase route, IRouteHandler routeHandler);
    public string GetRequiredString(string valueName);

    public RouteBase Route { get; set; }
    public IRouteHandler RouteHandler { get; set; }
    public RouteValueDictionary DataTokens { get; }
    public RouteValueDictionary Values { get; }
}
```

`RouteData` 的 `Values` 和 `DataTokens` 属性都返回一个 `RouteValueDictionary` 的对象。如下面的代码片段所示, `RouteValueDictionary` 是一个实现了 `IDictionary<string, object>` 接口的字典。ASP.NET 路由系统利用此对象来保存路由变量, 字典元素的 `Key` 和 `Value` 分别表示变量的名称和值。存储于 `Values` 和 `DataTokens` 这两个属性中的路由变量的不同之处在于: 前者是通过请求 URL 进行解析得到的, 后者则是直接附加到路由对象上的自定义变量。

```
public class RouteValueDictionary :
    IDictionary<string, object>
{
    //省略成员
}
```

在某些路由场景中, 我们要求 `Route` 针对请求进行路由解析得到的变量集合 (`Values` 属性) 中必须包含某些固定名称的变量值 (比如 ASP.NET MVC 应用中表示 `Controller` 和 `Action` 名称的变量), `RouteBase` 的 `GetRequiredString` 方法用于获取它们的值。对于该方法的调用, 如果指定名称的变量在 `Values` 属性中不存在, 它会直接抛出一个 `InvalidOperationException` 异常。

`RouteData` 通过其 `RouteHandler` 属性返回一个 `RouteHandler` 对象。`RouteHandler` 在整个路由系统中具有重要的地位, 因为最终用于处理请求的 `HttpHandler` 对象由它来提供。所有的 `RouteHandler` 类型均实现了具有如下定义的 `IRouteHandler` 接口, `HttpHandler` 的提供实现在它的 `GetHttpHandler` 方法中。我们可以在构造函数中对 `RouteData` 的 `RouteHandler` 属性进行初始化, 也可以直接对这个可读/写的属性进行赋值。

```
public interface IRouteHandler
{
    IHttpHandler GetHttpHandler(RequestContext requestContext);
}
```

当请求被成功路由到某个.aspx 页面后，通过调用匹配 Route 对象的 GetRouteData 方法生成的 RouteData 被直接附加到目标页面对应的 Page 对象上。如下面的代码片段所示，Page 具有一个类型为 RouteData 的同名只读属性，它返回的正是这个 RouteData 对象。

```
public class Page : TemplateControl, IHttpHandler
{
    //其他成员
    public RouteData RouteData { get; }
}
```

3. VirtualPathData

介绍完 GetRouteData 方法的返回类型 RouteData 之后，我们接着介绍 RouteBase 的 GetVirtualPath 方法的返回类型 VirtualPathData。当 RouteBase 的 GetVirtualPath 方法被执行的时候，如果定义在路由模板中的变量与指定变量列表相匹配，它会使用指定的路由变量值去替换路由模板中对应的占位符并生成一个虚拟路径。生成的虚拟路径与 Route 对象最终被封装成一个 VirtualPathData 对象作为返回值，它们对应着这个返回的 VirtualPathData 对象的 VirtualPath 和 Route 属性。VirtualPathData 的 DataTokens 属性和 RouteData 的同名属性一样都是来源于附加到 Route 对象的自定义变量集合。

```
public class VirtualPathData
{
    public VirtualPathData(RouteBase route, string virtualPath);

    public RouteValueDictionary DataTokens {get; }
    public RouteBase Route { get; set; }
    public string VirtualPath { get; set; }
}
```

RouteBase 的 GetVirtualPath 方法具有一个类型为 RequestContext 的参数，一个 RequestContext 对象表示针对某个请求的上下文。从如下的代码片段中不难看出它实际上是对 HTTP 上下文和 RouteData 的封装。

```
public class RequestContext
{
    public RequestContext();
    public RequestContext(HttpContextBase httpContext, RouteData routeData);
}
```



```

    public virtual HttpContextBase    HttpContext { get; set; }
    public virtual RouteData           RouteData { get; set; }
}

```

4. Route

RouteBase 是一个抽象类，在 ASP.NET 路由系统的应用编程接口中，**Route** 类型是其唯一的直接继承者，在默认的情况下调用 **RouteCollection** 的 **MapPageRoute** 方法在路由表中添加的就是这么一个对象。如下面的代码片段所示，**Route** 类型具有一个字符串类型的属性 **Url**，它代表绑定在该路由对象上的路由模板。

```

public class Route : RouteBase
{
    public Route(string url, IRouteHandler routeHandler);
    public Route(string url, RouteValueDictionary defaults,
        IRouteHandler routeHandler);
    public Route(string url, RouteValueDictionary defaults,
        RouteValueDictionary constraints, IRouteHandler routeHandler);
    public Route(string url, RouteValueDictionary defaults,
        RouteValueDictionary constraints,
        RouteValueDictionary dataTokens, IRouteHandler routeHandler);

    public override RouteData GetRouteData(HttpContextBase httpContext);
    public override VirtualPathData GetVirtualPath(RequestContext requestContext,
        RouteValueDictionary values);

    public RouteValueDictionary Constraints { get; set; }
    public RouteValueDictionary Defaults { get; set; }
    public RouteValueDictionary DataTokens { get; set; }
    public IRouteHandler RouteHandler { get; set; }
    public string Url { get; set; }
}

```

在默认的情况下，针对请求的路由解析由路由表中的某个 **Route** 对象来完成，而某个 **Route** 对象是否会被选择取决于请求 **URL** 是否与对应的路由模板的模式相匹配。具体的匹配规则很简单，我们可以通过一个简单的例子来说明。假设我们具有如下一个路由模板表示获取某个地区（通过电话区号表示）未来 *N* 天的天气情况的 **URL**。

```

/weather/{areacode}/{days}

```

对于上述这个路由模板来说，我们通过分隔符“/”对其进行拆分得到 3 个基本的字符串，它们被称为“段（Segment）”。对于组成某个段的内容，又可以分为“变量（Variable）”和“字面量（Literal）”，前者通过采用花括号（“{ }”）对变量名的包装来表示（比如表示电话区号的“{areacode}”和天数的“{days}”），后者则代表单纯的静态文字（比如“weather”）。值得一提的是，路由解析过程中针对字符的比较是不区分大小写的，因为 **URL** 本来就不区分大小写。

对于一个具体的 URL 来说，匹配成功需要有两个基本的条件，即该 URL 包含的段的数量和 URL 模板相同，对应的文本段内容也一致。按照这个匹配规则，下面这个 URL 和上面我们定义的路由模板是相匹配的。

```
/weather/0512/2
```

除了用于表示路由模板的核心属性 `Url` 之外，`Route` 类型还具有一些额外属性。属性 `Constraints` 为定义在模板中的变量以正则表达式的形式设定一些约束条件，该属性类型为 `RouteValueDictionary`，其 `Key` 和 `Value` 分别表示变量名和作为约束的正则表达式。比如对于上面定义的这个 URL 模板来说，我们可以为两个变量指定相应的正则表达式使请求地址具有合法的区号和作为整数的未来天数。如果我们通过该属性为 `Route` 对象定义了基于某些变量的正则表达式，匹配成功的先决条件除了上述两个之外，被验证的 URL 中对应的段还必须通过对应的正则表达式的验证。除了采用正则表达式来定义约束之外，还可以直接创建一个 `RouteConstraint` 对象来表示约束。

`Route` 类型的另一个属性 `Defaults` 同样也返回一个 `RouteValueDictionary` 对象，它保存了为路由变量定义的默认值。值得一提的是，具有默认值的路由变量不一定要出现在路由模板之中。当某个 `Route` 对象针对某个 URL 实施路由解析的时候，如果 URL 只能匹配路由模板前面的部分，但是后边部分均为变量并且具有对应的默认值，这种情况下依然被视为成功匹配。

还是以前面给出的路由模板为例，如果我们将 “{areacode}” 和 “{days}” 这两个变量的默认值分别设置为 “010”（北京）和 “2”（未来两天），如下所示的 3 个 URL 都能和拥有此路由模板的 `Route` 对象匹配成功，并且它们可以被视为等效的 URL。

```
/weather/010/2
/weather/010
/weather/
```

关于定义在路由模板中的变量，我们并不要求它作为整个段的内容。换句话说，一个段可以同时包含静态文字和变量。除此之外，我们还可以采用 “{*<<variable>>}” 的形式来定义匹配 URL 的最后部分（可以包含多个段）的变量，姑且称之为“通配变量”。

```
/ {filename}. {extension} / { *pathinfo }
```

对于如上的这个路由模板来说，第一个段中包含两部分内容，即表示文件名称和扩展名的变量 “{filename}” 和 “{extension}”，以及作为两者分隔符的字面量 “.”，后边紧跟一个通配符变量 “{*pathinfo}”。这个路由模板与下面一个 URL 是可以成功匹配的，匹配后定义在模板中的 3 个变量（{filename}、{extension} 和 {pathinfo}）的值分别为 “default”、“aspx” 和 “abc/123”。

```
/default.aspx/abc/123
```

Route 类型的 DataTokens 属性在之前已经有所提及，它用于存储一些额外路由变量，这些变量不会参与针对请求的路由解析。对于调用 Route 类型的 GetRouteData 和 GetVirtualPath 方法分别得到的 RouteData 和 VirtualPathData 对象来说，它们的数据 Tokens 属性所包含的路由变量都来源于此。

5. RouteTable

对于一个 Web 应用来说，访问所有页面采用的 URL 不可能具有相同的模式，与之匹配的 Route 自然也不可能是唯一的。一个 Web 应用通过 RouteTable 类型的静态只读属性 Routes 维护一个全局的路由表，如下面的代码片段所示，该属性返回一个 RouteCollection 对象。

```
public class RouteTable
{
    //其他成员
    public static RouteCollection Routes { get; }
}
```

顾名思义，RouteCollection 就是一组 Route 对象的集合。如下面的代码片段所示，RouteCollection 直接继承自 Collection<RouteBase>，除了继承自基类用于操作集合相关的成员之外，它还定义了一些额外的属性和方法。

```
public class RouteCollection : Collection<RouteBase>
{
    //其他成员
    public RouteData GetRouteData(HttpContextBase httpContext);
    public VirtualPathData GetVirtualPath(RequestContext requestContext,
        RouteValueDictionary values);
    public VirtualPathData GetVirtualPath(RequestContext requestContext,
        string name, RouteValueDictionary values);

    //定义不需检查是否匹配路由的 URL 模式
    public void Ignore(string url);
    public void Ignore(string url, object constraints);

    //针对 Web Page 的路由映射
    public Route MapPageRoute(string routeName, string routeUrl,
        string physicalFile);
    public Route MapPageRoute(string routeName, string routeUrl,
        string physicalFile, bool checkPhysicalUrlAccess);
    public Route MapPageRoute(string routeName, string routeUrl,
        string physicalFile, bool checkPhysicalUrlAccess,
        RouteValueDictionary defaults);
    public Route MapPageRoute(string routeName, string routeUrl,
```

```

        string physicalFile, bool checkPhysicalUrlAccess,
        RouteValueDictionary defaults, RouteValueDictionary constraints);
    public Route MapPageRoute(string routeName, string returnUrl,
        string physicalFile, bool checkPhysicalUrlAccess,
        RouteValueDictionary defaults, RouteValueDictionary constraints,
        RouteValueDictionary dataTokens);

    public bool AppendTrailingSlash { get; set; }
    public bool LowercaseUrls { get; set; }
    public bool RouteExistingFiles { get; set; }
}

```

当我们调用 `RouteCollection` 的 `GetRouteData` 和 `GetVirtualPath` 方法的时候，该方法会遍历集合中的每一个 `Route` 对象。针对每个 `Route` 对象，同名的方法会被调用。如果方法返回一个具体的 `RouteData` 或者 `VirtualPathData` 对象，它们会直接作为方法的返回值。换言之，集合中第一个匹配的 `Route` 对象返回的 `RouteData` 或者 `VirtualPathData` 对象将作为整个方法的返回值。如果每个 `Route` 对象均返回 `Null`，那么整个方法的返回值就是 `Null`。

`RouteCollection` 的 `RouteExistingFiles` 属性用于控制是否存在对物理文件实施路由，也就是说在被解析的 URL 与某个物理文件的路径一致的情况下是否还需要对其实施路由。该属性默认值为 `False`，即注册的路由不会影响到物理文件的请求。

`AppendTrailingSlash` 和 `LowercaseUrls` 这两个布尔类型的属性与方法 `GetVirtualPath` 有关，它们决定了对 URL 的标准化（Normalization）行为。具体来说，`AppendTrailingSlash` 属性表示是否需要在生成的 URL 末尾添加 “/”（如果没有），而 `LowercaseUrls` 属性则表示是否需要将生成的 URL 转变成小写。

其实我们使用得最为频繁的还是 `MapPageRoute` 和 `Ignore` 这两个方法。前者用于注册某个物理文件（路径）与路由模板之间的映射，其本质就是在本集合中添加一个 `Route` 对象。后者则与此相反，用于注册一个路由模板使路由系统可以忽略具有对应模式的 URL。

当我们在调用 `MapPageRoute` 方法的时候，它会将 `routeName` 参数作为对应 `Route` 对象的注册名称。如下面的代码片段所示，`RouteCollection` 具有一个 `Dictionary<string, RouteBase>` 类型的字段，注册的 `Route` 对象和注册名称之间的映射关系就保存在这个字典对象中。我们可以通过这个注册名称从 `RouteCollection` 中提取对应的 `Route` 对象。

```

public class RouteCollection : Collection<RouteBase>
{
    //其他成员
    private Dictionary<string, RouteBase> _namedMap;
}

```

6. 线程安全

通过 `RouteTable` 的静态只读属性 `Routes` 表示的 `RouteCollection` 对象是针对整个应用的全局路由表。这个集合对象本身并不能提供线程安全的保证，所以同一个 `RouteCollection` 对象在多个线程中被同时操作就有可能造成意想不到的并发问题。为了解决这个问题，如下两个方法（`GetReadLock` 和 `GetWriteLock`）被定义在 `RouteCollection` 类型中，我们在对集合进行读取或者更新的时候可以分别调用它们获取读锁和写锁。

```
public class RouteCollection : Collection<RouteBase>
{
    //其他成员
    public IDisposable GetReadLock();
    public IDisposable GetWriteLock();
}
```

当执行 `GetReadLock` 方法的时候，只有在当前 `RouteCollection` 对象的写锁尚未被获取时才会将集合的读锁返回，否则会等待写锁的释放。当我们调用 `GetWriteLock` 方法试图获取某个 `RouteCollection` 对象写锁的时候，针对该集合的写锁只有在没有任何线程拥有读锁和写锁的情况下才会返回，否则会等待所有锁的释放。也就是说线程安全状态下的 `RouteCollection` 对象可以被多个线程同时读取，但是不允许在被某个线程读取的同时被另一个线程更新。集合在某个时刻只能被一个线程更新，此时其他线程针对集合的读取和更新都是不允许的。

`RouteCollection` 的 `GetReadLock` 和 `GetWriteLock` 方法的返回类型都是 `IDisposable` 接口，实际上返回值的类型分别是内嵌于 `RouteCollection` 中的两个私有类型（`ReadLockDisposable` 和 `WriteLockDisposable`），它们通过封装的 `ReaderWriterLockSlim` 对象实现了读/写锁的功能。`ReadLockDisposable` 和 `WriteLockDisposable` 实现了 `IDisposable` 接口，并在 `Dispose` 方法中完成对锁的释放，所以推荐的编程方式如下所示。

```
//读操作
using (IDisposable readLock = routeCollection.GetReadLock())
{
    //读取 RouteCollection
}

//写操作
using (IDisposable writeLock = routeCollection.GetWriteLock())
{
    //更新 RouteCollection
}
```

我们所说的路由注册本质上就是创建相应的 `Route` 对象并将其添加到通过 `RouteTable` 的静态属性 `Routes` 表示的全局路由表中。照理说不论我们调用 `RouteCollection` 的 `Add` 方法或者

MapPageRoute 都需要预先获取集合的写锁，但是在一般情况下路由注册发生在应用启动的时候（此时请求尚未抵达），能够确保集合对象此时只会被一个单一线程操作，所以在这种情况下我们无须调用 GetWriteLock 方法。值得一提的是，RouteCollection 的两个方法 GetRouteData 和 GetVirtualPath 在对集合进行遍历之前已经调用了 GetReadLock 方法获得读锁，所以这两个方法本身就是线程安全的。

2.1.4 路由注册

总的来说，我们可以通过 RouteTable 的静态属性 Routes 得到一个针对整个应用的全局路由表。通过上面的介绍我们知道这是一个 RouteCollection 对象，可以通过调用它的 MapPageRoute 方法注册某个物理文件的路径与某个路由模板的匹配关系。路由注册的核心在于根据提供的路由规则（路由模板、约束、默认值等）创建一个 Route 对象，并将其添加到这个全局路由表中。接下来我们通过实例演示的方式来说明路由注册的一些细节问题。

前面给出了一个获取天气预报信息的路由模板，现在我们在一个 ASP.NET Web 应用中创建一个 Weather.aspx 页面。不过我们并不打算在该页面中呈现任何天气信息，而是将相关的路由信息呈现出来。该页面主体部分的 HTML 如下所示，我们不仅将基于当前页面的 RouteData 对象的 Route 和 RouteHandler 属性类型输出，还将存储于 Values 和 DataTokens 属性的变量显示出来。

```
<form id="form1" runat="server">
    <div>
        <table>
            <tr>
                <td>Route:</td>
                <td><%=RouteData.Route != null?
                    RouteData.Route.GetType().FullName:"" %></td>
            </tr>
            <tr>
                <td>RouteHandler:</td>
                <td><%=RouteData.RouteHandler != null?
                    RouteData.RouteHandler.GetType().FullName:"" %></td>
            </tr>
            <tr>
                <td>Values:</td>
                <td>
                    <ul>
                        <%foreach (var variable in RouteData.Values)
                        {%>
                            <li>
```

```

                <%=variable.Key%>=<%=variable.Value%></li>
            <% }%>
        </ul>
    </td>
</tr>
<tr>
    <td>DataTokens:</td>
    <td>
        <ul>
            <%foreach (var variable in RouteData.DataTokens)
            {%>
                <li>
                    <%=variable.Key%>=<%=variable.Value%></li>
                <% }%>
            </ul>
        </td>
    </tr>
</table>
</div>
</form>

```

在添加的 `Global.asax` 文件中，我们将路由注册操作定义在 `Application_Start` 方法中。如下面的代码片段所示，映射到 `Weather.aspx` 页面的路由模板为 “`{areacode}/{days}`”。在调用 `MapPageRoute` 方法的时候，我们还为定义在路由模板中的两个变量指定了默认值及基于正则表达式的约束。除此之外，我们还在注册的 `Route` 对象的 `DataTokens` 属性中添加了两个路由变量，它们表示对变量默认值的说明（`defaultCity: BeiJing; defaultDays: 2`）。顺便说一下，`MapPageRoute` 方法中布尔类型的参数 `checkPhysicalUrlAccess` 表示是否需要对被路由的目标地址的 URL 实施授权（针对原请求地址的 URL 授权总是会执行）。

```

public class Global : System.Web.HttpApplication
{
    protected void Application_Start(object sender, EventArgs e)
    {
        var defaults = new RouteValueDictionary {
            { "areacode", "010" }, { "days", 2 } };
        var constraints = new RouteValueDictionary {
            { "areacode", @"0\d{2,3}" }, { "days", @"[1-3]" } };
        var dataTokens = new RouteValueDictionary {
            { "defaultCity", "BeiJing" }, { "defaultDays", 2 } };

        RouteTable.Routes.MapPageRoute("default", "{areacode}/{days}",
            "~/weather.aspx", false, defaults, constraints, dataTokens);
    }
}

```

1. 变量默认值

由于我们为定义在 URL 模板中表示区号和天数的变量定义了默认值 (areacode: 010; days: 2), 如果希望返回北京地区未来两天的天气, 可以直接访问应用根地址, 也可以只指定具体区号, 或者同时指定区号和天数。如图 2-2 所示, 我们在浏览器地址栏中输入上述 3 种不同的 URL 会得到相同的输出结果。

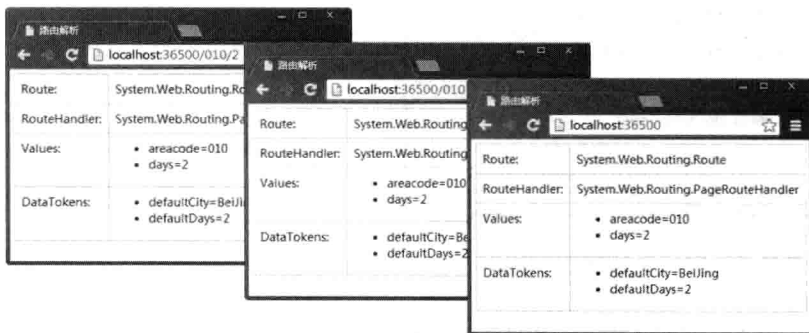


图 2-2 基于变量默认值的 URL 等效性

从图 2-2 所示的路由信息中可以看到, 默认情况下 `RouteData` 的 `Route` 属性返回的正是一个 `Route` 类型的对象, `RouteHandler` 属性返回的则是一个类型为 `PageRouteHandler` 的对象, 我们会在本章后续部分对 `PageRouteHandler` 进行详细介绍。针对请求 URL 实施路由解析得到的路由变量被保存在生成 `RouteData` 对象的 `Values` 属性中, 而在路由注册过程为 `Route` 对象的 `DataTokens` 属性指定的路由变量被转移到了 `RouteData` 的同名属性中。(S202)

2. 约束

我们以电话区号代表对应的城市, 为了确保用户在请求地址中提供有效的区号, 我们通过正则表达式 `(0\d{2,3})` 对其进行了约束。除此之外, 假设只能提供未来 3 天以内的天气情况, 我们同样通过正则表达式 `[1-3]` 对请求地址中表示天数的变量进行了约束。如果请求地址中的内容不能符合相关变量段的约束条件, 则意味着对应的路由对象与之不匹配。

对于本例来说, 由于只注册了唯一的路由对象, 如果请求地址不能满足我们定义的约束条件, 则意味着找不到一个具体目标文件, 此时会返回 404 错误。如图 2-3 所示, 由于在请求地址中指定了不合法的区号 (01) 和天数 (4), 我们直接在浏览器界面上得到一个 HTTP 404 错误。(S202)

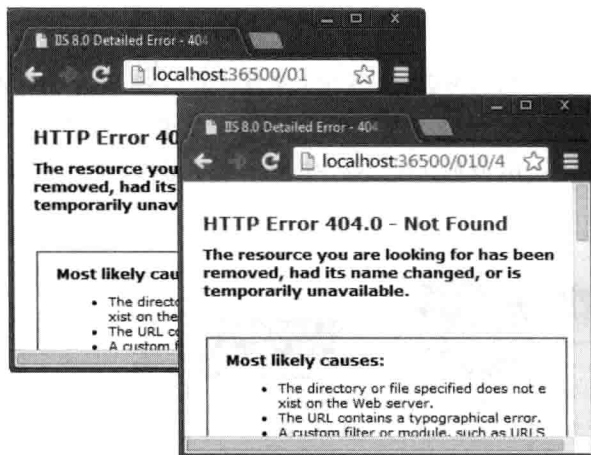


图 2-3 不满足正则表达式约束导致的 404 错误

对于约束，除了可以通过字符串的形式为某个变量定义相应的正则表达式之外，还可以指定一个 `RouteConstraint` 对象。所有的 `RouteConstraint` 类型均实现了 `IRouteConstraint` 接口，如下面的代码片段所示，该接口具有唯一的方法 `Match` 用于执行针对约束的检验。该方法 5 个参数分别表示当前 HTTP 上下文、当前 `Route` 对象、路由变量的名称（存储约束对象在 `RouteValueDictionary` 中对应的 `Key`）、用于替换定义在路由模板中占位符的变量集合及路由方向。

```
public interface IRouteConstraint
{
    bool Match(HttpContextBase httpContext, Route route, string parameterName,
        RouteValueDictionary values, RouteDirection routeDirection);
}

public enum RouteDirection
{
    IncomingRequest,
    UrlGeneration
}
```

所谓“路由方向”表明路由检验是针对请求匹配（入栈）还是针对 URL 的生成（出栈），分别通过如上所示的枚举类型 `RouteDirection` 的两个枚举值表示。`RouteBase` 类型的两个核心方法 `GetRouteData` 和 `GetVirtualPathData` 分别采用 `IncomingRequest` 和 `UrlGeneration` 作为路由方向。

ASP.NET 路由系统的应用编程接口中定义了如下一个实现了 `IRouteConstraint` 接口的 `HttpMethodConstraint` 类型。顾名思义，`HttpMethodConstraint` 提供针对 HTTP 方法（GET、POST、PUT、DELETE 等）的约束。如果我们通过 `HttpMethodConstraint` 为 `Route` 对象设置一个允许的 HTTP 方法列表，那么被路由的请求采用的 HTTP 方法必须在此列表中。这个被允许路由的 HTTP

方法列表对应着 `HttpMethodConstraint` 的只读属性 `AllowedMethods`, 该属性在构造函数中初始化。

```
public class HttpMethodConstraint : IRouteConstraint
{
    public HttpMethodConstraint(params string[] allowedMethods);

    bool IRouteConstraint.Match(HttpContextBase httpContext, Route route,
        string parameterName, RouteValueDictionary values,
        RouteDirection routeDirection);

    public ICollection<string> AllowedMethods { get; }
}
```

同样是针对前面演示的例子, 我们这次在进行路由注册的时候按照如下的方式将一个 `HttpMethodConstraint` 对象作为约束应用到被注册的 `Route` 对象上。此 `HttpMethodConstraint` 对象允许的 HTTP 方法列表中只具有 POST 这个唯一的 HTTP 方法, 意味着被注册的 `Route` 对象仅限于路由 POST 请求。

```
public class Global : System.Web.HttpApplication
{
    protected void Application_Start(object sender, EventArgs e)
    {
        var defaults = new RouteValueDictionary { { "areacode", "010" },
            { "days", 2 } };
        var constraints = new RouteValueDictionary { { "areacode", @"0\d{2,3}" },
            { "days", @"[1-3]{1}" },
            { "httpMethod", new HttpMethodConstraint("POST") } };
        var dataTokens = new RouteValueDictionary { { "defaultCity", "BeiJing" },
            { "defaultDays", 2 } };
        RouteTable.Routes.MapPageRoute("default", "{areacode}/{days}",
            "~/weather.aspx", false, defaults, constraints, dataTokens);
    }
}
```

现在我们采用与注册的模板相匹配的地址 (/010/2) 来访问 `Weather.aspx` 页面, 依然会得到如图 2-4 所示的 HTTP 404 错误。(S203)



图 2-4 不满足 HTTP 方法约束 (POST) 导致的 404 错误

3. 对现有物理文件的路由

在成功注册路由的情况下，如果我们按照传统的方式访问一个现存的物理文件（比如.aspx、.css 或者.js 等），在请求地址满足某个 Route 的路由规则的情况下，ASP.NET 是否还是正常实施路由呢？我们不妨通过实例来测试一下。为了让针对某个物理文件的访问地址也满足注册路由对象的路由模板采用的 URL 模式，我们需要按照如下的方式在进行路由注册时将表示约束的参数设置为 Null。

```
public class Global : System.Web.HttpApplication
{
    protected void Application_Start(object sender, EventArgs e)
    {
        var defaults = new RouteValueDictionary {
            { "areacode", "010" }, { "days", 2 } };
        var dataTokens = new RouteValueDictionary {
            { "defaultCity", "BeiJing" }, { "defaultDays", 2 } };
        RouteTable.Routes.MapPageRoute("default", "{areacode}/{days}",
            "~/weather.aspx", false, defaults, null, dataTokens);
    }
}
```

当通过传统的方式来访问存放于根目录下的 Weather.aspx 页面时会得到如图 2-5 所示的结果，从界面上的输出结果不难看出，虽然请求 URL 与注册 Route 对象的路由规则完全匹配，但是 ASP.NET 路由系统并没有对请求实施路由（如果中间发生了路由，基于页面的 RouteData 的各项属性都不可能为空）。(S204)

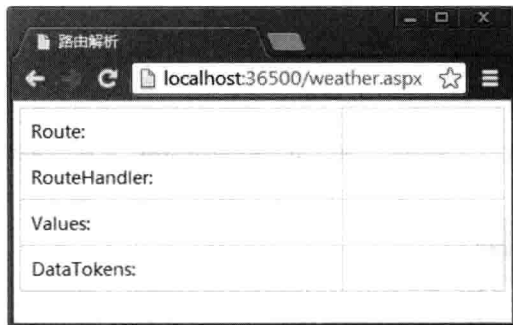


图 2-5 直接请求现存的物理文件（RouteExistingFiles = false）

如果请求 URL 对应着一个现存的物理文件的路径，ASP.NET 会不会总是自动忽略路由呢？实则不然，不对现有文件实施路由由仅仅是默认采用的行为而已，是否对现有文件实施路由取决于代表全局路由表的 RouteCollection 对象的 RouteExistingFiles 属性（该属性默认情况下为 False）。

我们可以将此属性设置为 `True` 使 ASP.NET 路由系统忽略现有物理文件的存在，让它总是按照注册的路由表进行路由。为了演示这种情况，我们对 `Global.asax` 文件作了如下改动，在进行路由注册之前将 `RouteTable` 的 `Routes` 属性代表的 `RouteCollection` 对象的 `RouteExistingFiles` 属性设置为 `True`。

```
public class Global : System.Web.HttpApplication
{
    protected void Application_Start(object sender, EventArgs e)
    {
        RouteTable.Routes.RouteExistingFiles = true;
        var defaults = new RouteValueDictionary {
            { "areacode", "010" }, { "days", 2 } };
        var dataTokens = new RouteValueDictionary {
            { "defaultCity", "BeiJing" }, { "defaultDays", 2 } };
        RouteTable.Routes.MapPageRoute("default", "{areacode}/{days}",
            "~/weather.aspx", false, defaults, null, dataTokens);
    }
}
```

依旧是针对 `Weather.aspx` 页面的访问却得到了不一样的结果。从图 2-6 中可以看到，针对页面的相对地址 `Weather.aspx` 不再指向具体的 Web 页面，在这里就是一个表示获取的天气信息对应的目标城市（`areacode=weather.aspx`）。（S205）

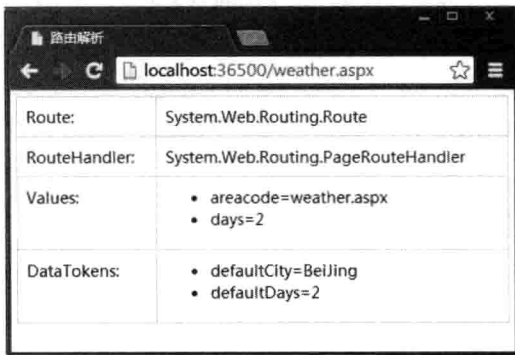


图 2-6 直接请求现存的物理文件（`RouteExistingFiles = true`）

通过上面的介绍我们知道，作为路由对象集合的 `RouteCollection` 和 `RouteBase` 均具有一个布尔类型的 `RouteExistingFiles` 属性，用以控制是否针对现有物理文件实施路由，它们的默认值分别是 `False` 和 `True`。由上面的实例演示我们知道，当请求与 `RouteCollection` 的某个 `Route` 对象的路由规则相匹配的情况下，`RouteCollection` 本身的 `RouteExistingFiles` 属性会改变 `GetRouteData` 的值，使该方法只有在 `RouteExistingFiles` 属性值为 `False` 的情况下才会返回一个 `RouteData` 对象，否则直接返回 `Null`。

我们现在需要讨论的是另一个问题：Route 对象自身的 RouteExistingFiles 属性值对于自身的 GetRouteData 方法及它所在的 RouteCollection 对象的 GetRouteData 又会造成什么样的影响呢？

对于一个 Route 对象来说，它自身的 GetRouteData 方法不受其 RouteExistingFiles 属性值的影响，也就是说 GetRouteData 能否返回一个具体的 RouteData 对象完全取决于定义的路由规则是否与请求相匹配。如果一个 RouteCollection 包含一个唯一的 Route 对象，那么它的 GetRouteData 方法只有同时满足如下 3 个条件才能返回一个具体的 RouteData 对象。

- RouteCollection 自身的 RouteExistingFiles 属性为 True。
- Route 对象的 RouteExistingFiles 也为 True。
- Route 对象的路由规则与请求相匹配。

也就是说 Route 自身的 RouteExistingFiles 属性对于自身的路由没有影响，该属性最终是给 RouteCollection 使用的，笔者个人觉得这样的设计有待商榷。为了让读者对 RouteCollection 和 Route 的 RouteExistingFiles 属性对它们各自路由解析产生的影响具有一个深刻的认识，我们不妨再做下面这个实例演示。

我们创建一个空的 ASP.NET 应用，并在添加的默认 Web 页面 Default.aspx 的后台文件中定义如下一个 GetRouteData 方法。该方法根据指定的参数返回一个 RouteData 对象，其中枚举类型的参数 routeOrCollection 决定返回的 RouteData 是调用 Route 对象的 GetRouteData 方法生成的还是调用 RouteCollection 的 GetRouteData 方法生成的，参数 routeExistingFiles4Collection 和 routeExistingFiles4Route 则分别控制着 RouteCollection 和 Route 对象的 RouteExistingFiles 属性。

```
public partial class Default : System.Web.UI.Page
{
    public enum RouteOrRouteCollection
    {
        Route,
        RouteCollection
    }

    public RouteData GetRouteData(RouteOrRouteCollection routeOrCollection,
        bool routeExistingFiles4Collection, bool routeExistingFiles4Route)
    {
        Route route = new Route("{areaCode}/{days}", new RouteValueDictionary
            { { "areacode", "010" }, { "days", 2 } }, null);
        route.RouteExistingFiles = routeExistingFiles4Route;
        HttpContextBase context = CreateHttpContext();

        if (routeOrCollection == RouteOrRouteCollection.Route)
        {
            return route.GetRouteData(context);
        }
    }
}
```

```

RouteCollection routes = new RouteCollection();
routes.Add(route);
routes.RouteExistingFiles = routeExistingFiles4Collection;
return routes.GetRouteData(context);
}

private static HttpContextBase CreateHttpContext()
{
    HttpRequest request = new HttpRequest("~/weather.aspx",
        "http://localhost:3721/weather.aspx", null);
    HttpResponse response = new HttpResponse(new StringWriter());
    HttpContext context = new HttpContext(request, response);
    HttpContextBase contextWrapper = new HttpContextWrapper(context);
    return contextWrapper;
}
}

```

在上面定义的这个 `GetRouteData` 方法中创建的 `Route` 对象采用我们熟悉的路由模板 “{areaCode}/{days}”，并且两个变量均具有默认值。如果需要返回 `Route` 对象自身生成的 `RouteData` 对象，我们直接调用其 `GetRouteData` 方法，否则创建一个仅仅包含该 `Route` 对象的 `RouteCollection` 对象并调用其 `GetRouteData` 方法。调用 `GetRouteData` 方法传入的参数是我们手工创建的 `HttpContextWrapper` 对象，它的请求 URL (“http://localhost:3721/weather.aspx”) 与 `Route` 对象的路由模板相匹配。

由于需要验证针对现有物理文件的路由，所以我们会在该应用的根目录下创建一个名为 `Weather.aspx` 的空页面，同时将应用发布的端口设置为 3721。然后我们在 `Default.aspx` 页面的主体部分定义如下的 HTML，它会将 `RouteCollection` 和 `Route` 的 `RouteExistingFiles` 属性在不同组合下调用各自 `GetRouteData` 方法的返回值通过表格的形式呈现出来（具体来说，如果返回值不为空则输出 “RouteData”，否则输出 “Null”）。

```

<form id="form1" runat="server">
  <table>
    <thead>
      <tr>
        <th>RouteCollection.RouteExistingFiles</th>
        <th colspan="2">True</th>
        <th colspan="2">False</th>
      </tr>
      <tr>
        <th>Route.RouteExistingFiles</th>
        <th>True</th>
        <th>False</th>
        <th>True</th>
        <th>False</th>
      </tr>
    </thead>
    <tbody>

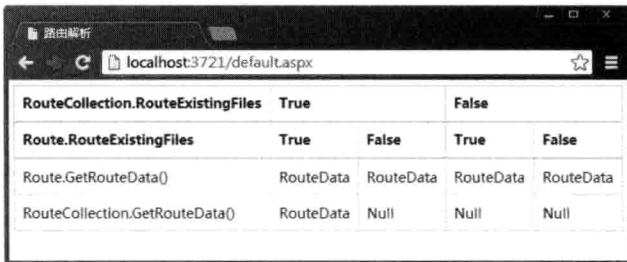
```

```

<tr>
  <td>Route.GetRouteData()</td>
  <td><%=this.GetRouteData(RouteOrRouteCollection.Route,true,true) ==
    null ? "Null":"RouteData" %></td>
  <td><%=this.GetRouteData(RouteOrRouteCollection.Route,true,false) ==
    null ? "Null":"RouteData" %></td>
  <td><%=this.GetRouteData(RouteOrRouteCollection.Route,false,true) ==
    null ? "Null":"RouteData" %></td>
  <td><%=this.GetRouteData(RouteOrRouteCollection.Route,false,false)
    == null ? "Null":"RouteData" %></td>
</tr>
<tr>
  <td>RouteCollection.GetRouteData()</td>
  <td><%=this.GetRouteData(RouteOrRouteCollection.RouteCollection,
    true,true) == null ? "Null":"RouteData" %></td>
  <td><%=this.GetRouteData(RouteOrRouteCollection.RouteCollection,
    true,false) == null ? "Null":"RouteData" %></td>
  <td><%=this.GetRouteData(RouteOrRouteCollection.RouteCollection,
    false,true) == null ? "Null":"RouteData" %></td>
  <td><%=this.GetRouteData(RouteOrRouteCollection.RouteCollection,
    false,false) == null ? "Null":"RouteData" %></td>
</tr>
</tbody>
</table>
</form>

```

运行这个程序后会在浏览器中呈现如图 2-7 所示的输出结果,这实际上证实了我们上面的结论,即 Route 对象的 GetRouteData 方法不会受到自身 RouteExistingFiles 属性的影响。在请求与路由规则匹配的情况下,RouteCollection 只有在自身 RouteExistingFiles 属性和 Route 对象的 RouteExistingFiles 属性同时为 True 的情况下才会返回一个具体的 RouteData 对象。(S206)



RouteCollection.RouteExistingFiles	True		False	
Route.RouteExistingFiles	True	False	True	False
Route.GetRouteData()	RouteData	RouteData	RouteData	RouteData
RouteCollection.GetRouteData()	RouteData	Null	Null	Null

图 2-7 RouteCollection 与 Route 的 RouteExistingFiles 属性对路由的影响

4. 注册路由忽略地址

RouteTable 的静态属性 Routes 返回的 RouteCollection 对象代表针对整个应用的全局路由表。如果我们将该对象的 RouteExistingFiles 属性设置为 True, ASP.NET 路由系统将会对所有

抵达的请求实施路由，但这同样会带来一些问题。

举个简单的例子，一个 Web 应用往往涉及很多静态文件，比如文本类型的 JavaScript 或者 CSS 文件和图片。如果一个 Web 应用寄宿于 IIS 下，对于 Classic 模式下的 IIS 7.x 及之前的版本，针对这些静态文件请求直接由 IIS 来响应，并不会进入 ASP.NET 的管道，所以由 ASP.NET 提供的路由机制并不会对针对它们的访问造成任何影响。

但是对于 Integrated 模式下的 IIS 7.5，如果采用与 ASP.NET 集成管道（关于 IIS 7.x 中集成 ASP.NET 管道，在本书第 1 章“ASP.NET MVC”中有详细介绍），所有类型的请求都将进入 ASP.NET 管道。在这种情况下，如果允许路由系统路由由现有物理文件，针对某个静态文件的请求就有可能被重定向到其他地方，这意味着我们将不能正常访问这些静态文件。除了采用基于 Integrated 模式下的 IIS 作为 Web 服务器，在采用 Visual Studio 提供的 ASP.NET Development Server 及 IIS Express（IIS Express 和 IIS 功能上面基本相同）的情况下这种问题依然存在。

我们就用上面的实例（S205）来演示这个问题。本书演示实例默认都是采用 IIS Express。为了让 ASP.NET 管道能够接管所有类型的访问请求，我们需要在 web.config 中添加如下一段配置。

```
<configuration>
  <system.webServer>
    <modules runAllManagedModulesForAllRequests="true" />
  </system.webServer>
</configuration>
```

我们在“/Content/”目录下放置了一个名为 bootstrap.css 的 CSS 文件来控制页面显示的样式，并在允许针对现有物理文件路由的情况下（RouteTable.Routes.RouteExistingFiles = true）通过浏览器来访问这个 CSS 文件。我们最终会在浏览器中得到如图 2-8 所示的输出结果。由于 CSS 文件的路径（/content/bootstrap.css）与注册 Route 的路由模板（{areacode}/{days}）是匹配的，所以对应的请求自然就被路由到 Weather.aspx 页面了。（S207）

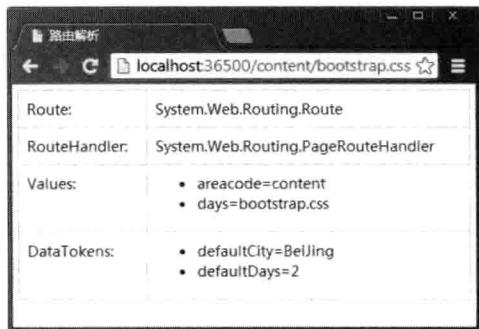


图 2-8 直接请求现存的.css 文件（RouteExistingFiles = true）

这是一个不得不解决的问题，因为它使我们无法正常地在页面中引用 JavaScript 和 CSS 文件。我们可以通过调用 `RouteCollection` 的 `Ignore` 方法来注册一些需要让路由系统忽略的 URL。从前面给出的关于 `RouteCollection` 的定义中可以看到它具有两个 `Ignore` 方法重载，除了指定与需要忽略的 URL 相匹配的路由模板之外，还可以对相关的变量定义约束正则表达式。为了让路由系统忽略针对 CSS 文件的请求，我们可以按照如下的方式在 `Global.asax` 中调用 `RouteTable` 的 `Routes` 属性的 `Ignore` 方法。值得一提的是，这样的方法调用应该放在路由注册之前，否则起不到任何作用。(S208)

```
public class Global : System.Web.HttpApplication
{
    protected void Application_Start(object sender, EventArgs e)
    {
        RouteTable.Routes.RouteExistingFiles = true;
        RouteTable.Routes.Ignore("content/{filename}.css/{*pathInfo}");
        //其他操作
    }
}
```

5. 直接添加路由对象

我们调用 `RouteCollection` 对象的 `MapPageRoute` 方法进行路由注册的本质就是在路由表中添加 `Route` 对象，所以我们完全可以调用 `Add` 方法添加一个手工创建的 `Route` 对象。如下所示的两种路由注册方式是完全等效的。如果需要添加一个继承自 `RouteBase` 的自定义路由对象，我们不得不采用手工添加的方式。

```
public class Global : System.Web.HttpApplication
{
    protected void Application_Start(object sender, EventArgs e)
    {
        var defaults = new RouteValueDictionary {
            { "areacode", "010" }, { "days", 2 } };
        var constraints = new RouteValueDictionary {
            { "areacode", @"0\d{2,3}" }, { "days", @"[1-3]{1}" } };
        var dataTokens = new RouteValueDictionary {
            { "defaultCity", "BeiJing" }, { "defaultDays", 2 } };

        //路由注册方式 1
        RouteTable.Routes.MapPageRoute("default", "{areacode}/{days}",
            "~/weather.aspx", false, defaults, constraints, dataTokens);

        //路由注册方式 2
        Route route = new Route("{areacode}/{days}", defaults, constraints,
            dataTokens, new PageRouteHandler("~/weather.aspx", false));
        RouteTable.Routes.Add("default", route);
    }
}
```

2.1.5 根据路由规则生成 URL

前面已经提到过 ASP.NET 的路由系统主要有两个方面的应用，一个是通过注册路由模板与物理文件路径的映射实现请求 URL 和物理地址的分离，另一个则是通过注册的路由规则生成一个完整的 URL，后者通过调用 `RouteCollection` 对象的 `GetVirtualPath` 方法来实现。

如下面的代码片段所示，`RouteCollection` 定义了两个 `GetVirtualPath` 方法重载，它们共同的参数 `requestContext` 和 `values` 分别表示请求上下文（`RouteData` 和 HTTP 上下文的封装）和用于替换定义在路由模板中的变量占位符的路由变量。另一个 `GetVirtualPath` 方法具有一个额外的字符串参数 `name`，它表示集合中具体使用的路由对象的注册名称（调用 `MapPageRoute` 方法时指定的第一个参数）。

```
public class RouteCollection : Collection<RouteBase>
{
    //其他成员
    public VirtualPathData GetVirtualPath(RequestContext requestContext,
        RouteValueDictionary values);
    public VirtualPathData GetVirtualPath(RequestContext requestContext,
        string name, RouteValueDictionary values);
}
```

如果调用 `GetVirtualPath` 方法时没有指定具体生成虚拟路径的 `Route` 对象，那么该方法会遍历整个路由表，直到找到一个路由模板与指定的路由参数列表相匹配的 `Route` 对象，并返回由它生成的 `VirtualPathData` 对象。具体来说，该方法会依次调用路由表中每个 `Route` 对象的 `GetVirtualPath` 方法，直到该方法返回一个具体的 `VirtualPathData` 对象为止。如果调用所有 `Route` 对象的 `GetVirtualPath` 方法的返回值均为 `Null`，那么整个方法的返回值也为 `Null`。

在调用 `GetVirtualPath` 方法的时候可以传入 `Null` 作为第一个参数（`requestContext`），在这种情况下它会根据当前 HTTP 上下文（对应于 `HttpContext` 的静态属性 `Current`）创建一个 `RequestContext` 对象作为调用 `Route` 对象 `GetVirtualPath` 方法的参数，该参数包含一个空的 `RouteData` 对象。如果当前 HTTP 上下文不存在，则该方法会直接抛出一个类型为 `InvalidOperationException` 的异常。

`Route` 对象针对 `GetVirtualPath` 方法而进行的路由解析只要求路由模板中定义的变量的值都能被提供，而这些变量值具有 3 种来源，分别是 `Route` 对象中为路由变量定义的默认值、指定 `RequestContext` 对象的 `RouteData` 中提供的变量值（`Values` 属性）和额外提供的变量值（通过 `values` 参数指定的 `RouteValueDictionary` 对象），这 3 种变量值具有由低到高的选择优先级。

同样以之前定义的关于获取天气信息的路由模板为例，我们在 `Weather.aspx` 页面的后台代

码中按照如下方法通过 `RouteTable` 的静态 `Routes` 得到代表全局路由表的 `RouteCollection` 对象，并调用其 `GetVirtualPath` 方法生成 3 个具体的 URL。

```
public partial class Weather : Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        RouteData routeData = new RouteData();
        routeData.Values.Add("areaCode", "0512");
        routeData.Values.Add("days", "1");
        RequestContext requestContext = new RequestContext();
        requestContext.HttpContext = new HttpContextWrapper(HttpContext.Current);
        requestContext.RouteData = routeData;

        RouteValueDictionary values = new RouteValueDictionary();
        values.Add("areaCode", "028");
        values.Add("days", "3");

        Response.Write(RouteTable.Routes.GetVirtualPath(null, null).VirtualPath
            + "<br/>");
        Response.Write(RouteTable.Routes.GetVirtualPath(requestContext,
            null).VirtualPath + "<br/>");
        Response.Write(RouteTable.Routes.GetVirtualPath(requestContext,
            values).VirtualPath + "<br/>");
    }
}
```

从上面的代码片段可以看到，第一次调用 `GetVirtualPath` 方法传入的 `requestContext` 和 `values` 参数均为 `Null`；第二次则指定了一个手工创建的 `RequestContext` 对象，其 `RouteData` 的 `Values` 属性具有两个变量（`areaCode=0512`；`days=1`），而 `values` 参数依然为 `Null`；第三次则同时为参数 `requestContext` 和 `values` 指定了具体的对象，后者包含两个参数（`areaCode=028`；`days=3`）。如果我们利用浏览器访问 `Weather.aspx` 页面会得到如图 2-9 所示的 3 个 URL，这充分证实了上面提到的关于变量选择优先级的结论。（S209）

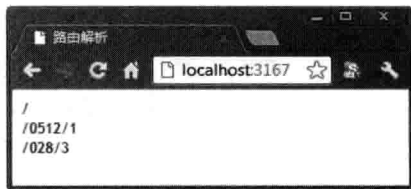


图 2-9 `GetVirtualPath` 方法接受不同参数生成的 URL

2.2 ASP.NET MVC 路由

ASP.NET 的路由系统旨在通过注册路由模板与物理文件路径之间的映射进而实现请求地址与文件路径之间的分离，但是对于 ASP.NET MVC 应用来说，请求的目标不再是一个具体的物理文件，而是定义在某个 Controller 类型中的 Action 方法。出于自身路由特点的需要，ASP.NET MVC 对 ASP.NET 的路由系统进行了相应的扩展。

2.2.1 路由映射

通过前面的介绍我们知道，RouteTable 的静态属性 Routes 返回的 RouteCollection 对象代表了针对整个应用的全局路由表，我们可以调用其 MapPageRoute 完成针对某个物理文件的路由。为了实现针对目标 Controller 和 Action 的路由，ASP.NET MVC 为 RouteCollection 类型定义了一系列的扩展方法，这些扩展方法定义在 RouteCollectionExtensions 类型中（该类型定义在“System.Web.Mvc”命名空间下，如果未作特别说明，本书涉及的与 ASP.NET MVC 相关的类型均定义在此命名下）。

如下面的代码片段所示，RouteCollectionExtensions 定义了两组方法。方法 IgnoreRoute 用于注册与需要被忽略的 URL 模式相匹配的路由模板，它对应于 RouteCollection 类型的 Ignore 方法。方法 MapRoute 帮助我们根据提供的路由规则（路由模板、约束和默认值等）进行路由注册，它对应于 RouteCollection 的 MapPageRoute 方法。

```
public static class RouteCollectionExtensions
{
    //其他成员
    public static void IgnoreRoute(this RouteCollection routes, string url);
    public static void IgnoreRoute(this RouteCollection routes, string url,
        object constraints);

    public static Route MapRoute(this RouteCollection routes, string name,
        string url);
    public static Route MapRoute(this RouteCollection routes, string name,
        string url, object defaults);
    public static Route MapRoute(this RouteCollection routes, string name,
        string url, string[] namespaces);
    public static Route MapRoute(this RouteCollection routes, string name,
        string url, object defaults, object constraints);
    public static Route MapRoute(this RouteCollection routes, string name,
        string url, object defaults, string[] namespaces);
    public static Route MapRoute(this RouteCollection routes, string name,
        string url, object defaults, object constraints, string[] namespaces);
}
```

由于 ASP.NET MVC 的路由注册与具体的物理文件无关, 所以 `MapRoute` 方法中并没有一个表示文件路径的 `physicalFile` 参数。与直接定义在 `RouteCollection` 中的 `Ignore` 和 `MapPageRoute` 方法不同的是, 表示默认路由变量值和约束的参数 `defaults` 和 `constraints` 不再是一个 `RouteValueDictionary` 对象, 而是一个普通的 `object`。这主要是为了编程上的便利, 这样的设计使我们可以通过匿名类型的方式来指定这两个参数值。该方法在内部会通过反射的方式得到指定对象的属性列表, 并将其转换为 `RouteValueDictionary` 对象, 指定对象的属性名和属性值将作为字典元素的 `Key` 和 `Value`。

对于 ASP.NET MVC 路由系统对请求 URL 进行路由解析后生成的 `RouteData` 对象来说, 包含在 `Values` 属性的路由变量集合中必须包含目标 `Controller` 的名称。由于 `Controller` 名称仅仅对应着类型的名称 (不含命名空间), 而目标 `Controller` 实例能够被激活的前提是我们能够正确地解析出它的真实类型, 所以如果一个应用中定义了多个同名的 `Controller` 类型, 我们不得不借助于类型所在的命名空间来对它们予以区分。

我们在调用 `MapRoute` 方法的时候可以通过字符串数组类型的参数 `namespaces` 来指定一个命名空间的列表。对于注册的命名空间, 我们可以指定一个代表完整命名空间的字符串, 也可以使用 “*” 作为通配符表示任意字符内容 (比如 “`Artech.Web.*`”)。添加的命名空间列表最终被存储于 `Route` 对象的 `DataTokens` 属性表示的 `RouteValueDictionary` 对象中, 对应的 `Key` 为 “`Namespaces`”。`MapRoute` 方法没有为初始化 `Route` 对象的 `DataTokens` 属性提供相应的参数, 如果没有指定命名空间列表, 所有通过该方法添加的 `Route` 对象的 `DataTokens` 属性总是一个空的 `RouteValueDictionary` 对象。

对于指向定义在 `Controller` 类型中某个 `Action` 方法的请求来说, 如果路由表与之匹配, 则具体匹配的 `Route` 对象的 `GetRouteData` 方法被调用并返回一个具体的 `RouteData` 对象。对请求实施路由解析得到的代表目标 `Controller` 和 `Action` 的名称的路由变量必须包含在该 `RouteData` 的 `Values` 属性中, 其对应的变量名分别为 “`controller`” 和 “`action`”。

2.2.2 路由注册 (S210)

ASP.NET MVC 通过调用代表全局路由表的 `RouteCollection` 对象的扩展方法 `MapRoute` 进行路由注册。为了让读者对此有一个深刻的认识, 我们来进行一个简单的实例演示。我们依然沿用之前关于获取天气信息的路由模板, 看看通过这种方式注册的 `Route` 对象针对匹配的请求将返回怎样一个 `RouteData` 对象。

我们在创建的空 ASP.NET Web 应用 (不是 ASP.NET MVC 应用, 所以需要人为地添加针对

程序集“System.Web.Mvc.dll”和“System.Web.WebPages.Razor.dll”的引用¹)中添加如下一个 Web 页面 (Default.aspx)，并按照之前的做法以内联代码的方式直接将 RouteData 的相关属性显示出来。需要注意的是，我们显示的 RouteData 是通过调用自定义的 GetRouteData 方法获取的，而不是当前页面的 RouteData 属性返回的 RouteData 对象。

```
<form id="form1" runat="server">
    <div>
        <table>
            <tr>
                <td>Route:</td>
                <td><%=GetRouteData().Route != null?
                    GetRouteData().Route.GetType().FullName:"" %></td>
            </tr>
            <tr>
                <td>RouteHandler:</td>
                <td><%=GetRouteData().RouteHandler != null?
                    GetRouteData().RouteHandler.GetType().FullName:"" %></td>
            </tr>
            <tr>
                <td>Values:</td>
                <td>
                    <ul>
                        <%=foreach (var variable in GetRouteData().Values)
                            {%>
                        <li>
                            <%=variable.Key%>=<%=variable.Value%></li>
                        <% }%>
                    </ul>
                </td>
            </tr>
            <tr>
                <td>DataTokens:</td>
                <td>
                    <ul>
                        <%=foreach (var variable in GetRouteData().DataTokens)
                            {%>
                        <li>
                            <%=variable.Key%>=<%=variable.Value%></li>
                        <% }%>
                    </ul>
                </td>
            </tr>
        </table>
    </div>
</form>
```

¹ 我们具有两种获取 ASP.NET MVC 相关的程序集。如果安装了 ASP.NET MVC 5，可以在目录“%ProgramFiles%\Microsoft ASP.NET\ASP.NET Web Stack 5\Packages”中找到这个程序集。也可以利用 Visual Studio 创建一个 ASP.NET MVC 应用的方式得到与 ASP.NET MVC 相关的所有程序集。

我们将 `GetRouteData` 方法定义在当前页面的后台代码中。如下面的代码片段所示，我们根据手工创建的 `HttpRequest`（请求 URL 为“`http://localhost/0512/3`”）和 `HttpResponse` 对象创建了一个 `HttpContext` 对象，然后以此创建一个 `HttpContextWrapper` 对象。接下来我们利用 `RouteTable` 的静态属性 `Routes` 获取代表全局路由表的 `RouteCollection` 对象，并将这个 `HttpContextWrapper` 对象作为参数调用其 `GetRouteData` 方法。这个方法实际上就是模拟注册的路由表针对相对地址为“`/0512/3`”的请求的路由解析。

```
public partial class Default : System.Web.UI.Page
{
    private RouteData routeData;

    public RouteData GetRouteData()
    {
        if (null != routeData)
        {
            return routeData;
        }
        HttpRequest request = new HttpRequest("default.aspx",
            "http://localhost/0512/3", null);
        HttpResponse response = new HttpResponse(new StringWriter());
        HttpContext context = new HttpContext(request, response);
        HttpContextBase contextWrapper = new HttpContextWrapper(context);

        return routeData = RouteTable.Routes.GetRouteData(contextWrapper);
    }
}
```

具体的路由注册依然定义在添加的 `Global.asax` 文件中。如下面的代码片段所示，我们利用 `RouteTable` 的静态属性 `Routes` 获取代表全局路由表的 `RouteCollection` 对象，然后调用其 `MapRoute` 方法注册了一个采用“`{areacode}/{days}`”作为路由模板的 `Route` 对象，并指定了变量的默认值、约束和命名空间列表。由于成功匹配的路由对象必须具有一个名为“`controller`”的路由变量，所以我们直接将 `controller` 的默认值设置为“`home`”。

```
public class Global : System.Web.HttpApplication
{
    protected void Application_Start(object sender, EventArgs e)
    {
        object defaults = new
        {
            areacode = "010",
            days = 2,
            defaultCity = "BeiJing",
            defaultDays = 2,
            controller = "home"
        };
        object constraints = new { areacode = @"0\d{2,3}", days = @"[1-3]" };
        string[] namespaces = new string[] {
```

```

        "Artech.Web.Mvc", "Artech.Web.Mvc.Html" };
RouteTable.Routes.MapRoute("default", "{areacode}/{days}",
    defaults, constraints, namespaces);
    }
}

```

如果我们现在在浏览器中访问 `Default.aspx` 页面，会得到如图 2-10 所示的输出结果，从中可以得到一些有用的信息。首先，与调用 `RouteCollection` 的 `MapPageRoute` 方法进行路由映射不同，得到的这个 `RouteData` 对象的 `RouteHandler` 属性返回一个 `MvcRouteHandler` 对象。其次，在 `MapRoute` 方法中通过 `defaults` 参数指定的两个不参与路由解析的路由变量（`defaultCity=BeiJing`；`defaultDays=2`）会转移到 `RouteData` 的 `Values` 属性中。这意味着如果我们没有在路由模板中为 `Controller` 和 `Action` 的名称定义相应的变量（“`{controller}`”和“`{action}`”），则可以将它们定义成具有默认值的变量。第三，`DataTokens` 属性中包含一个名为“`Namespaces`”路由变量，不难猜出它的值对应着我们指定的命名空间列表。

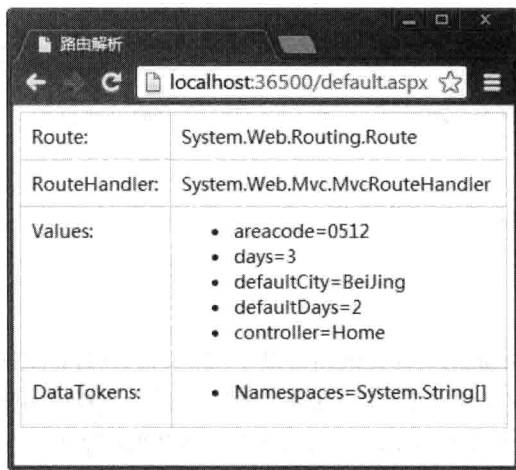


图 2-10 采用 ASP.NET MVC 路由映射得到的 `RouteData`

2.2.3 缺省 URL 参数

当通过 Visual Studio 的 ASP.NET MVC 项目模板创建一个 Web 应用后，它会为我们注册下一个模板为“`{controller}/{action}/{id}`”的默认 `Route` 对象。3 个路由变量（`{controller}`、`{action}` 和 `{id}`）均具有相应的默认值，但是变量名为 `id` 的默认值为 `UrlParameter.Optional`。按照字面的意思，我们将其称为可缺省 URL 参数。那么将路由变量的默认值进行如此设置与设置一个具体的默认值有什么区别呢？

```

public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        //其他操作
        routes.MapRoute(
            name: "Default",
            url: "{controller}/{action}/{id}",
            defaults: new { controller = "Home", action = "Index",
                           id = UrlParameter.Optional }
        );
    }
}

```

在介绍可缺省 URL 参数之前，我们不妨先来看看 `UrlParameter` 类型的定义。如下面的代码片段所示，`UrlParameter` 是一个不能被实例化的类型（它具有的唯一构造函数是私有的），唯一有用的就是它的静态只读字段 `Optional`。这是典型的单例编程模式，意味着多次注册的缺省 URL 参数引用着同一个 `UrlParameter` 对象。

```

public sealed class UrlParameter
{
    public static readonly UrlParameter Optional = new UrlParameter();

    private UrlParameter() {}

    public override string ToString()
    {
        return string.Empty;
    }
}

```

在进行路由解析的时候，默认值为 `UrlParameter.Optional` 的路由变量与其他具有默认值的路由变量并没有什么差别。它们之间的不同之处在于：如果将某个定义在路由模板中的变量的默认值设置为 `UrlParameter.Optional`，则只有请求 URL 真正包含具体变量值的情况下生成的 `RouteData` 的 `Values` 属性中才会包含相应的路由变量。

举个简单的例子，我们在 ASP.NET MVC Web 应用²中直接使用如上所示的默认注册的路由，并定义了如下一个 `HomeController`，定义其中的 `Action` 方法 `Index` 具有一个名为 `id` 的参数。我们在该方法中将包含在当前 `RouteData` 对象的 `Values` 属性中的所有路由变量的名称和值都输出出来。

2 本书用于实例演示而创建的 Web 应用，如果没有特殊说明就是通过 Visual Studio 的 ASP.NET MVC 项目模板创建的空 Web 应用。为了尽可能的简洁，我们会删除默认添加的大部分文件，只保留 `Global.asax`、`RouteConfig`（注册默认的 URL 路由）和 `web.config`。在必要的时候我们会添加一些 CSS 样式，但是具体的样式设置不会出现在给出的代码中。

```

public class HomeController : Controller
{
    public void Index(string id)
    {
        foreach (var variable in RouteData.Values)
        {
            Response.Write(string.Format("{0}: {1}<br/>",
                variable.Key, variable.Value));
        }
    }
}

```

我们直接运行该程序，并在浏览器的地址栏中输入不同的 URL 来访问 HomeController 的 Action 方法 Index，看看最终包含在 RouteData 的路由变量有何不同。如图 2-11 所示，当直接通过根地址访问的时候，RouteData 的 Values 属性中只包含 controller 和 action 这两个变量，被设置为 UrlParameter.Optional 的路由变量 id 只有在请求 URL 包含相应值的情况下才会出现在 RouteData 的 Values 属性中。（S211）

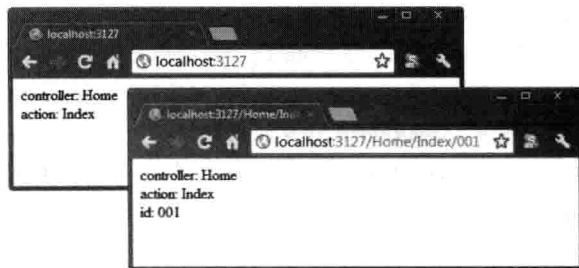


图 2-11 普通路由变量与缺省 URL 参数的路由变量之间的差别

2.2.4 基于 Area 的路由映射

对于一个较大规模的 Web 应用，我们可以从功能上通过 Area 将其划分为较小的单元。每个 Area 相当于一个独立的子系统，它们具有一套包含 Models、Views 和 Controller 在内的目录结构和配置文件。一般来说，每个 Area 具有各自的路由规则（路由模板上一般会包含 Area 的名称），而基于 Area 的路由映射通过 AreaRegistration 类型进行注册。

1. AreaRegistration 与 AreaRegistrationContext

针对 Area 的路由通过 AreaRegistration 来注册。如下面的代码片段所示，AreaRegistration 是一个抽象类，它的抽象只读属性 AreaName 返回当前 Area 的名称，而抽象方法 RegisterArea 用于实现基于当前 Area 的路由注册。

```

public abstract class AreaRegistration
{
    public static void RegisterAllAreas();
    public static void RegisterAllAreas(object state);

    public abstract void RegisterArea(AreaRegistrationContext context);
    public abstract string AreaName { get; }
}

```

AreaRegistration 定义了两个抽象的静态 **RegisterAllAreas** 方法重载，参数 **state** 表示传递给具体 **AreaRegistration** 的数据。当 **RegisterAllArea** 方法被执行的时候，所有被当前 Web 应用直接或者间接引用的程序集会被加载（如果尚未加载），ASP.NET MVC 会从这些程序集中解析出所有继承自 **AreaRegistration** 的类型，并通过反射创建相应的 **AreaRegistration** 对象。针对每个被创建出来的 **AreaRegistration** 对象，一个作为 **Area** 注册上下文的 **AreaRegistrationContext** 对象会被创建出来，它被作为参数调用这些 **AreaRegistration** 对象的 **RegisterArea** 方法进行针对相应 **Area** 的路由注册。

如下面的代码片段所示，**AreaRegistrationContext** 的只读属性 **AreaName** 表示 **Area** 的名称，属性 **Routes** 是一个代表路由表的 **RouteCollection** 对象，而 **State** 是一个用户自定义对象，它们均通过构造函数进行初始化。具体来说，**AreaRegistrationContext** 对象是在调用 **AreaRegistration** 的静态方法 **RegisterAllAreas** 时针对创建出来的 **AreaRegistration** 对象构建的，其 **AreaName** 来源于当前 **AreaRegistration** 对象的同名属性，**Routes** 则对应着 **RouteTable** 的静态属性 **Routes** 所表示的全局路由表。调用 **RegisterAllAreas** 方法指定的参数 **state** 将被作为调用 **AreaRegistrationContext** 构造函数的同名参数。

```

public class AreaRegistrationContext
{
    public AreaRegistrationContext(string areaName, RouteCollection routes);
    public AreaRegistrationContext(string areaName, RouteCollection routes,
        object state);

    public Route MapRoute(string name, string url);
    public Route MapRoute(string name, string url, object defaults);
    public Route MapRoute(string name, string url, string[] namespaces);
    public Route MapRoute(string name, string url, object defaults,
        object constraints);
    public Route MapRoute(string name, string url, object defaults,
        string[] namespaces);
    public Route MapRoute(string name, string url, object defaults,
        object constraints, string[] namespaces);

    public string AreaName { get; }
    public RouteCollection Routes { get; }
    public object State { get; }
    public ICollection<string> Namespaces { get; }
}

```

`AreaRegistrationContext` 的只读属性 `Namespaces` 表示一组需要优先匹配的命名空间（当多个同名的 `Controller` 类型定义在不同的命名空间的时候，定义在这些命名空间的 `Controller` 类型会被优先选用）。当针对某个具体 `AreaRegistration` 的 `AreaRegistrationContext` 对象被创建的时候，如果 `AreaRegistration` 类型定义在某个命名空间（比如 “`Artech.Controllers`”），则在这个命名空间基础上添加 “`*`” 后缀生成的字符串（比如 “`Artech.Controllers.*`”）会被添加到 `Namespaces` 集合中。换言之，对于多个定义在不同命名空间中的同名 `Controller` 类型，会优先选择包含在当前 `AreaRegistration` 所在命名空间下的 `Controller`。

`AreaRegistrationContext` 定义了一系列的 `MapRoute` 方法进行路由注册，方法的使用及参数的含义与 `RouteCollection` 类的同名扩展方法一致。在这里需要特别指出的是，如果 `MapRoute` 方法没有指定命名空间，通过属性 `Namespaces` 表示的命名空间列表会被使用；反之，该属性中包含的命名空间会被直接忽略。

当我们通过 Visual Studio 的 ASP.NET MVC 项目模板创建一个 Web 应用的时候，在 `Global.asax` 文件中会生成类似如下所示的代码，在这里通过调用 `AreaRegistration` 的静态方法 `RegisterAllAreas` 实现对所有 `Area` 的注册。也就是说，针对所有 `Area` 的注册发生在 Web 应用启动的时候。

```
public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        AreaRegistration.RegisterAllAreas();
        // 其他操作
    }
}
```

2. AreaRegistration 的缓存

`Area` 的注册（主要是基于 `Area` 的路由映射注册）通过具体的 `AreaRegistration` 来完成。在应用启动的时候，ASP.NET MVC 会遍历通过调用 `BuildManager` 的静态方法 `GetReferencedAssemblies`³得到的程序集列表，并从中找到所有 `AreaRegistration` 类型。如果一个应用涉及太多的程序集，则这个过程可能会耗费很多时间。为了提高性能，ASP.NET MVC 会对解析出来的所有 `AreaRegistration` 类型列表进行缓存。

ASP.NET MVC 对 `AreaRegistration` 类型列表的缓存是基于文件的。具体来说，当 ASP.NET MVC 框架通过程序集加载和类型反射得到了所有的 `AreaRegistration` 类型列表后，会对其序列

³ `BuildManager` 的静态方法 `GetReferencedAssemblies` 返回必须引用的程序集列表，这包括包含 `Web.config` 文件的 `<system.web>`/`<compilation>`/`<assemblies>` 配置节中指定的用于编译 Web 应用所使用的程序集和从 `App_Code` 目录中的自定义代码生成的程序集，以及其他顶级文件夹中的程序集。

化并将序列化的结果保存为一个物理文件中。这个名为“MVC-AreaRegistrationTypeCache.xml”的XML文件被存放在ASP.NET的临时目录下，具体的路径如下。其中第一个针对寄宿于Local IIS中的Web应用，后者针对直接通过Visual Studio Developer Server或者IIS Express作为宿主的应用。

- %Windir%\Microsoft.NET\Framework\v{version}\TemporaryASP.NET Files\{appname}\...\\UserCache\
- %Windir%\Microsoft.NET\Framework\v{version}\Temporary ASP.NET Files\root\...\\UserCache\

下面的XML片段体现了这个作为所有AreaRegistration类型缓存的XML文件的结构。我们从中可以看到所有的AreaRegistration类型名称，连同它所在的托管模块和程序集名称都被保存了下来。当AreaRegistration的静态方法RegisterAllAreas被调用之后，系统会试图加载该文件，如果该文件存在并且具有期望的结构，那么系统将不再通过程序集加载和反射来解析所有AreaRegistration的类型，而是直接对文件内容进行反序列化得到所有AreaRegistration类型的列表。

```
<?xml version="1.0" encoding="utf-8"?>
<!--This file is automatically generated. Please do not modify the contents of
this file.-->
<typeCache lastModified="3/3/2014 10:06:29 AM"
    mvcVersionId="72d59038-e845-45b1-853a-70864614e003">
  <assembly name="Artech.Admin, Version=1.0.0.0, Culture=neutral,
    PublicKeyToken=null">
    <module versionId="07be22a1-781d-4ade-bd22-34b0850445ef">
      <type>Artech.Admin.AdminAreaRegistration</type>
    </module>
  </assembly>
  <assembly name="Artech.Portal, Version=1.0.0.0, Culture=neutral,
    PublicKeyToken=null">
    <module versionId="7b0490d4-427e-43cb-8cb5-ac1292bd4976">
      <type>Artech.Portal.PortalAreaRegistration</type>
    </module>
  </assembly>
</typeCache>
```

如果这样的XML不存在，或者具有错误的结构（这样会造成针对AreaRegistration类型列表的反序列化失败），ASP.NET MVC框架会按照上述的方式重新解析出所有AreaRegistration类型列表，并将其序列化成XML保存到这个指定的文件中。值得一提的是，针对Web应用的重新编译会促使这些缓存文件的清除。

3. 实例演示：查看针对 Area 的路由信息（S212）

不同于一般的路由注册，通过 `AreaRegistration` 实现的针对 Area 的路由注册具有一些特殊的细节差异，我们可以通过实例演示的方式来对此予以说明。我们直接使用前面创建的演示实例（S210），并在项目中创建一个自定义的 `WeatherAreaRegistration` 类。如下面的代码片段所示，`WeatherAreaRegistration` 继承自抽象基类 `AreaRegistration`，表示 Area 名称的 `AreaName` 属性返回“`Weather`”。我们在 `RegisterArea` 方法中调用 `AreaRegistrationContext` 对象的 `MapRoute` 方法注册了一个模板为“`weather/{areacode}/{days}`”的 `Route` 对象，相应的默认变量值、约束也被提供。

```
public class WeatherAreaRegistration : AreaRegistration
{
    public override string AreaName
    {
        get { return "Weather"; }
    }

    public override void RegisterArea(AreaRegistrationContext context)
    {
        object defaults = new
        {
            areacode    = "010",
            days        = 2,
            defaultCity = "BeiJing",
            defaultDays = 2
        };
        object constraints = new { areacode = @"0\d{2,3}", days = @"[1-3]" };
        context.MapRoute("weatherDefault", "weather/{areacode}/{days}", defaults,
            constraints);
    }
}
```

我们可以在 `Global.asax` 的 `Application_Start` 方法中按照如下的方式调用 `AreaRegistration` 的静态方法 `RegisterAllAreas` 来实现对所有 Area 的注册。按照上面介绍的 Area 注册原理，`RegisterAllAreas` 方法的第一次调用会自动加载所有引用的程序集来获取所有的 `AreaRegistration` 类型（当然会包括我们上面定义的 `WeatherAreaRegistration`），最后通过反射创建相应的对象并调用其 `RegisterArea` 方法。

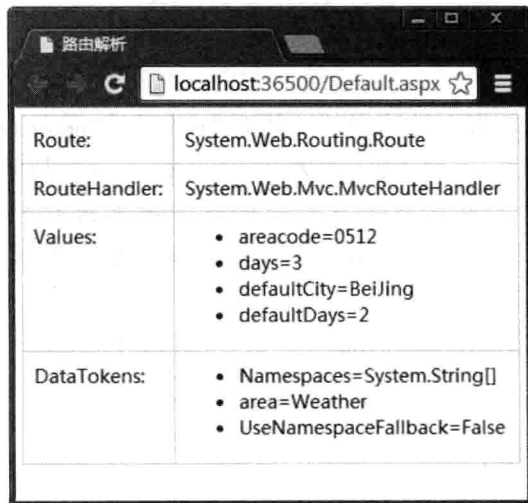
```
public class Global : System.Web.HttpApplication
{
    protected void Application_Start(object sender, EventArgs e)
    {
        AreaRegistration.RegisterAllAreas();
    }
}
```

在进行路由解析并生成 `RouteData` 对象的 `GetRouteData` 方法中，我们对创建的 `HttpRequest`

对象略加修改。如下面的代码片段所示，我们将请求 URL 设置为“/weather/0512/3”，正好与 WeatherAreaRegistration 注册的 Route 对象采用的路由模板（weather/{areacode}/{days}）相匹配。

```
public partial class Default : System.Web.UI.Page
{
    private RouteData routeData;
    public RouteData GetRouteData()
    {
        if (null != routeData)
        {
            return routeData;
        }
        HttpRequest request = new HttpRequest("default.aspx",
            "http://localhost/weather/0512/3", null);
        HttpResponse response = new HttpResponse(new StringWriter());
        HttpContext context = new HttpContext(request, response);
        HttpContextBase contextWrapper = new HttpContextWrapper(context);
        return routeData = RouteTable.Routes.GetRouteData(contextWrapper);
    }
}
```

在浏览器中访问 Default.aspx 页面会得到如图 2-12 所示的输出结果。通过 AreaRegistration 注册的 Route 对象生成的 RouteData 的不同之处主要反映在其 DataTokens 属性上。如图 2-12 所示，除了表示命名空间列表的元素之外，DataTokens 属性表示的 RouteValueDictionary 还具有两个额外的路由变量，其中一个名为“area”的变量代表 Area 的名称，另一个名为“UseNamespaceFallback”的变量表示是否需要使用后备的命名空间来解析 Controller 类型。



Route:	System.Web.Routing.Route
RouteHandler:	System.Web.Mvc.MvcRouteHandler
Values:	<ul style="list-style-type: none"> areacode=0512 days=3 defaultCity=Beijing defaultDays=2
DataTokens:	<ul style="list-style-type: none"> Namespaces=System.String[] area=Weather UseNamespaceFallback=False

图 2-12 采用 AreaRegistration 路由映射得到的 RouteData

如果调用 `AreaRegistrationContext` 的 `MapRoute` 方法是显式指定了命名空间，或者对应的 `AreaRegistration` 定义在某个命名空间下，这个名称为 “`UseNamespaceFallback`” 的 `DataToken` 元素的值为 `False`，反之则被设置为 `True`。进一步来说，如果在调用 `MapRoute` 方法时指定了命名空间列表，那么 `AreaRegistration` 类型所在的命名空间会被忽略。也就是说后者是前者的一个后备，前者具有更高的优先级。

`AreaRegistration` 类型所在命名空间也不是直接作为最终 `RouteData` 的 `DataTokens` 中的命名空间，而是在此基础上加上 “`.*`” 后缀。针对我们的实例来说，包含在 `RouteData` 的 `DataTokens` 集合中的命名空间为 “`WebApp.*`”（`WebApp` 是定义 `WeatherAreaRegistration` 的命名空间）。

2.2.5 链接和 URL 的生成

ASP.NET 路由系统通过注册的路由表旨在实现两个“方向”的路由解析，即针对入栈请求的路由和出栈 URL 的生成。前者通过调用代表全局路由表的 `RouteCollection` 对象的 `GetRouteData` 方法实现，后者则依赖于 `RouteCollection` 的 `GetVirtualPathData` 方法，但是最终还是落在具有某个 `Route` 对象的同名方法的调用上。

ASP.NET MVC 定义了两个名为 `HtmlHelper` 和 `UrlHelper` 的帮助类，可以通过调用它们的 `ActionLink/RouteLink` 和 `Action/RouteUrl` 方法根据注册的路由规则生成相应的链接或者 URL。从本质上讲，`HtmlHelper/UrlHelper` 实现的对 URL 的生成最终还是依赖于前面所说的 `GetVirtualPathData` 方法。

1. `UrlHelper` V.S. `HtmlHelper`

在介绍如何通过 `HtmlHelper` 和 `UrlHelper` 来生成链接或者 URL 之前，我们先来看看它们的基本定义。从下面给出的代码片段可以看出，一个 `UrlHelper` 对象实际上是对一个表示请求上下文的 `RequestContext` 对象和表示路由表的 `RouteCollection` 对象的封装，它们分别对应于只读属性 `RequestContext` 和 `RouteCollection`。如果在构造 `UrlHelper` 的时候没有通过参数指定 `RouteCollection` 对象，那么通过 `RouteTable` 的静态属性 `Routes` 表示的全局路由表将直接被使用。

```
public class UrlHelper
{
    //其他成员
    public UrlHelper(RequestContext requestContext);
    public UrlHelper(RequestContext requestContext,
        RouteCollection routeCollection);
```



```

    public RequestContext      RequestContext { get; }
    public RouteCollection      RouteCollection { get; }
}

```

再来看看如下所示的 `HtmlHelper` 的定义，它同样具有一个表示路由表的 `RouteCollection` 属性。和 `UrlHelper` 一样，如果我们在调用构造函数的时候没有通过参数来指定初始化此属性的 `RouteCollection` 对象，则 `RouteTable` 的静态属性 `Routes` 表示的 `RouteCollection` 对象将会用于初始化该属性。

```

public class HtmlHelper
{
    //其他成员
    public HtmlHelper(ViewContext viewContext,
        IViewDataContainer viewDataContainer);
    public HtmlHelper(ViewContext viewContext,
        IViewDataContainer viewDataContainer, RouteCollection routeCollection);

    public RouteCollection      RouteCollection { get; }
    public ViewContext          ViewContext { get; }
}

public class ViewContext : ControllerContext
{
    //省略成员
}

public class ControllerContext
{
    //其他成员
    public RequestContext      RequestContext { get; set; }
    public virtual RouteData    RouteData { get; set; }
}

```

由于 `HtmlHelper` 只是在 `View` 中使用，所以它具有一个通过 `ViewContext` 属性表示的针对 `View` 的上下文。对于 `ViewContext`，我们会在第 11 章“`View` 的呈现”中对其进行单独介绍，在这里只需要知道它的父类是表示 `Controller` 上下文的 `ControllerContext`，通过后者的 `RequestContext` 和 `RouteData` 属性可以获取代表当前请求上下文的 `RequestContext` 对象和通过路由解析生成的 `RouteData` 对象。

2. `UrlHelper.Action()` V.S. `HtmlHelper.ActionLink()`

`UrlHelper` 和 `HtmlHelper` 分别通过 `Action` 和 `ActionLink` 方法生成一个指向定义在某个 `Controller` 类型中的 `Action` 方法的 URL 和链接。下面的代码片段列出了 `UrlHelper` 的所有 `Action` 方法重载，参数 `actionName` 和 `controllerName` 分别代表 `Action` 和 `Controller` 的名称。object 或者 `RouteValueDictionary` 类型表示的 `routeValues` 参数表示替换路由模板中变量的参数列表。参

数 protocol 和 hostName 代表作为完整 URL 的传输协议（比如 http 和 https 等）和主机名。

```
public class UrlHelper
{
    //其他成员
    public string Action(string actionName);
    public string Action(string actionName, object routeValues);
    public string Action(string actionName, string controllerName);
    public string Action(string actionName, RouteValueDictionary routeValues);
    public string Action(string actionName, string controllerName,
        object routeValues);
    public string Action(string actionName, string controllerName,
        RouteValueDictionary routeValues);

    public string Action(string actionName, string controllerName,
        object routeValues, string protocol);
    public string Action(string actionName, string controllerName,
        RouteValueDictionary routeValues, string protocol, string hostName);
}
```

对于定义在 UrlHelper 中的众多 Action 方法来说，如果我们显式指定了传输协议（protocol 参数）或者主机名称，返回的是一个绝对地址，否则返回的是一个相对地址。如果我们没有显式地指定 Controller 的名称（controllerName 参数），那么当前 Controller 的名称会被采用。对于 UrlHelper 来说，通过 RequestContext 属性表示的当前请求上下文包含了相应的路由信息，即 RequestContext 的 RouteData 属性表示的 RouteData，它的 Values 属性中必须包含一个名为“controller”的路由变量，对应的变量值就代表当前 Controller 的名称。

ASP.NET MVC 为 HtmlHelper 定义了如下所示的一系列 ActionLink 扩展方法重载。顾名思义，ActionLink 不再仅仅返回一个 URL，而是生成一个链接（<a>...），但是其中作为目标 URL 的生成逻辑与 UrlHelper 是完全一致的。

```
public static class LinkExtensions
{
    //其他成员
    public static MvcHtmlString ActionLink(this HtmlHelper htmlHelper,
        string linkText, string actionName);
    public static MvcHtmlString ActionLink(this HtmlHelper htmlHelper,
        string linkText, string actionName, object routeValues);
    public static MvcHtmlString ActionLink(this HtmlHelper htmlHelper,
        string linkText, string actionName, string controllerName);
    public static MvcHtmlString ActionLink(this HtmlHelper htmlHelper,
        string linkText, string actionName, RouteValueDictionary routeValues);
    public static MvcHtmlString ActionLink(this HtmlHelper htmlHelper,
        string linkText, string actionName, object routeValues,
        object htmlAttributes);
    public static MvcHtmlString ActionLink(this HtmlHelper htmlHelper,
        string linkText, string actionName, RouteValueDictionary routeValues,
        IDictionary<string, object> htmlAttributes);
}
```

```

public static MvcHtmlString ActionLink(this HtmlHelper htmlHelper,
    string linkText, string actionName, string controllerName,
    object routeValues, object htmlAttributes);
public static MvcHtmlString ActionLink(this HtmlHelper htmlHelper,
    string linkText, string actionName, string controllerName,
    RouteValueDictionary routeValues,
    IDictionary<string, object> htmlAttributes);
public static MvcHtmlString ActionLink(this HtmlHelper htmlHelper,
    string linkText, string actionName, string controllerName,
    string protocol, string hostName, string fragment,
    object routeValues, object htmlAttributes);
public static MvcHtmlString ActionLink(this HtmlHelper htmlHelper,
    string linkText, string actionName, string controllerName,
    string protocol, string hostName, string fragment,
    RouteValueDictionary routeValues,
    IDictionary<string, object> htmlAttributes);
}

```

3. 实例演示：创建一个 RouteHelper 模拟 UrlHelper 的 URL 生成逻辑（S213）

为了让读者对 UrlHelper 如何利用 ASP.NET 路由系统生成 URL 的逻辑具有一个深刻认识，接下来我们会在一个空的 ASP.NET 应用中创建一个名为 RouteHelper 的等效帮助类。如下面的代码片段所示，RouteHelper 具有 RequestContext 和 RouteCollection 两个属性，前者在构造函数中指定，后者直接返回通过 RouteTable 的 Routes 静态属性表示的全局路由表。

```

public class RouteHelper
{
    public RequestContext    RequestContext { get; private set; }
    public RouteCollection   RouteCollection { get; private set; }

    public RouteHelper(RequestContext requestContext)
    {
        this.RequestContext    = requestContext;
        this.RouteCollection   = RouteTable.Routes;
    }

    public string Action(string actionName, string controllerName=null,
        object routeValues=null, string protocol=null, string hostName = null)
    {
        controllerName = controllerName ??
            this.RequestContext.RouteData.GetRequiredString("controller");
        RouteValueDictionary routeValueDictionary =
            new RouteValueDictionary(routeValues);
        routeValueDictionary.Add("action", actionName);
        routeValueDictionary.Add("controller", controllerName);

        string virtualPath = this.RouteCollection.GetVirtualPath(
            this.RequestContext, routeValueDictionary).VirtualPath;
    }
}

```

```

        if (string.IsNullOrEmpty(protocol) && string.IsNullOrEmpty(hostName))
        {
            return virtualPath.ToLower();
        }

        protocol = protocol ?? "http";
        Uri uri = this.RequestContext.HttpContext.Request.Url;
        hostName = hostName ?? uri.Host + ":" + uri.Port;
        return string.Format("{0}://{1}{2}", protocol,
            hostName, virtualPath).ToLower();
    }
}

```

RouteHelper 定义了一个 Action 方法根据指定的 Action 名称、Controller 名称、路由参数列表、网络协议前缀和主机名称来生成相应的 URL，除了第一个表示 Action 名称的参数，其余参数均是可以缺省的。具体的逻辑很简单：如果指定的 Controller 名称为 Null，我们会通过 RequestContext 获取当前 Controller 名称，然后将 Action 和 Controller 名称添加到表示路由变量的 RouteValueDictionary 对象中（routeValues 参数），对应的 Key 分别是“action”和“controller”。

然后我们调用 RouteCollection 的 GetVirtualPath 方法得到一个 VirtualPathData 对象。如果没有显式指定传输协议和主机名称，该方法直接返回 VirtualPathData 对象的 VirtualPath 属性表示的相对路径，否则通过添加传输协议前缀和主机名称生成一个完整的 URL。倘若没有显式指定主机名称，我们会采用当前请求的主机名称并使用当前的端口。如果没有指定传输协议，则直接使用“http”作为协议前缀。

接下来我们在添加的 Global.asax 中通过如下的代码注册一个路由模板为“{controller}/{action}/{id}”的路由对象。

```

public class Global : System.Web.HttpApplication
{
    protected void Application_Start(object sender, EventArgs e)
    {
        RouteTable.Routes.MapRoute(
            name: "Default",
            url: "{controller}/{action}/{id}",
            defaults: new
            {
                controller = "Home",
                action      = "Index",
                id          = UrlParameter.Optional
            }
        );
    }
}

```

我们在添加的 Web 页面 (Default.aspx) 中通过如下的代码利用自定义的 RouteHelper 生成 5 个 URL。在页面加载事件处理方法中, 我们根据手工创建的 HttpRequest (请求地址为 “http://localhost:3721/products/getproduct/001”) 和 HttpResponse 创建一个 HttpContext 对象, 并进一步创建 HttpContextWrapper 对象。接下来我们利用 RouteTable 的静态属性 Routes 得到表示全局路由表的 RouteCollection 对象, 并将这个 HttpContextWrapper 对象作为参数调用其 GetRouteData 方法。方法调用返回的 RouteData 对象和这个 HttpContextWrapper 对象进一步封装成一个 RequestContext 对象, RouteHelper 对象根据此对象被创建出来。

```
public partial class Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        HttpRequest request = new HttpRequest("default.aspx",
            "http://localhost:3721/products/getproduct/001", null);
        HttpResponse response = new HttpResponse(new StringWriter());
        HttpContext context = new HttpContext(request, response);
        HttpContextBase contextWrapper = new HttpContextWrapper(context);

        RouteData routeData = RouteTable.Routes.GetRouteData(contextWrapper);
        RequestContext requestContext = new RequestContext(
            contextWrapper, routeData);
        RouteHelper helper = new RouteHelper(requestContext);

        Response.Write(helper.Action("GetProductCategories") + "<br/>");
        Response.Write(helper.Action("GetAllContacts", "Sales") + "<br/>");
        Response.Write(helper.Action("GetAllContact", "Sales",
            new { id = "001" }) + "<br/>");
        Response.Write(helper.Action("GetAllContact", "Sales",
            new { id = "001" }, "https") + "<br/>");
        Response.Write(helper.Action("GetAllContact", "Sales",
            new { id = "001" }, "https", "www.artech.com") + "<br/>");
    }
}
```

运行该程序之后, 通过调用 RouteHelper 的 Action 方法生成的 5 个 URL 会以图 2-13 所示的方式出现在浏览器上。



图 2-13 通过自定义 RouteHelper 生成的 URL

4. UrlHelper.RouteUrl() V.S. HtmlHelper.RouteLink()

不论是 UrlHelper 的 Action 方法，还是 HtmlHelper 的 ActionLink，URL 都是通过表示路由表的 RouteCollection 对象生成出来的，在默认情况下这个对象就是通过 RouteTable 的静态属性 Routes 表示的全局路由表。换句话说，具体使用的总是路由表中第一个匹配的 Route 对象。但是有时候我们需要针对注册的某个具体的 Route 对象来生成 URL 或者链接，在这种情况下就需要用到 UrlHelper 和 HtmlHelper 的另外一组方法了。

如下面的代码片段所示，UrlHelper 定义了一系列的 RouteUrl 方法，除了第一个重载之外，后面的重载都接受一个表示 Route 注册名称的参数 routeName。与调用 UrlHelper 的 Action 方法一样，我们可以指定用于替换定义在 URL 模板中路由变量的参数（routeValues），以及传输协议名称（protocol）和主机名称（hostName）。

```
public class UrlHelper
{
    //其他成员
    public string RouteUrl(object routeValues);
    public string RouteUrl(string routeName);
    public string RouteUrl(RouteValueDictionary routeValues);
    public string RouteUrl(string routeName, object routeValues);
    public string RouteUrl(string routeName, RouteValueDictionary routeValues);
    public string RouteUrl(string routeName, object routeValues,
        string protocol);
    public string RouteUrl(string routeName, RouteValueDictionary routeValues,
        string protocol, string hostName);
}
```

对于没有显式指定 Route 注册名称的 RouteUrl 方法来说，它还是利用整个路由表进行 URL 的生成。如果显式指定了采用 Route 的注册名称，那么 ASP.NET MVC 会从路由表中获取相应的 Route 对象。如果该路由对象与指定的变量列表不匹配，则方法调用会返回 Null，否则会返回生成的 URL。

HtmlHelper 同样定义了类似的 RouteLink 方法重载用于实现基于指定路由对象的链接生成，具体的 RouteLink 方法定义如下。

```
public static class LinkExtensions
{
    //其他成员
    public static MvcHtmlString RouteLink(this HtmlHelper htmlHelper,
        string linkText, object routeValues);
    public static MvcHtmlString RouteLink(this HtmlHelper htmlHelper,
```

```

        string linkText, string routeName);
    public static MvcHtmlString RouteLink(this HtmlHelper htmlHelper,
        string linkText, RouteValueDictionary routeValues);
    public static MvcHtmlString RouteLink(this HtmlHelper htmlHelper,
        string linkText, object routeValues, object htmlAttributes);
    public static MvcHtmlString RouteLink(this HtmlHelper htmlHelper,
        string linkText, string routeName, object routeValues);
    public static MvcHtmlString RouteLink(this HtmlHelper htmlHelper,
        string linkText, string routeName, RouteValueDictionary routeValues);
    public static MvcHtmlString RouteLink(this HtmlHelper htmlHelper,
        string linkText, RouteValueDictionary routeValues,
        IDictionary<string, object> htmlAttributes);
    public static MvcHtmlString RouteLink(this HtmlHelper htmlHelper,
        string linkText, string routeName, object routeValues,
        object htmlAttributes);
    public static MvcHtmlString RouteLink(this HtmlHelper htmlHelper,
        string linkText, string routeName, RouteValueDictionary routeValues,
        IDictionary<string, object> htmlAttributes);
    public static MvcHtmlString RouteLink(this HtmlHelper htmlHelper,
        string linkText, string routeName, string protocol, string hostName,
        string fragment, object routeValues, object htmlAttributes);
    public static MvcHtmlString RouteLink(this HtmlHelper htmlHelper,
        string linkText, string routeName, string protocol, string hostName,
        string fragment, RouteValueDictionary routeValues,
        IDictionary<string, object> htmlAttributes);
}

```

2.3 动态 HttpHandler 映射

通过第1章“ASP.NET + MVC”对 ASP.NET 管道式设计的介绍,我们知道一般情况下一个请求最终是通过一个 **HttpHandler** 来处理的。表示一个 Web 页面的 **Page** 对象就是一个 **HttpHandler**, 它被用于最终处理针对某个.aspx 文件的请求。我们可以通过 **HttpHandler** 的动态映射来实现请求地址与物理文件路径之间的分离。

实际上 ASP.NET 路由系统就是采用了这样的实现原理。如图 2-14 所示, ASP.NET 的路由系统通过一个注册的 **HttpModule** 对象实现对请求的拦截, 然后为当前 HTTP 上下文动态映射了一个 **HttpHandler** 对象, 后者将会接管对当前请求的处理并最终对请求予以响应。

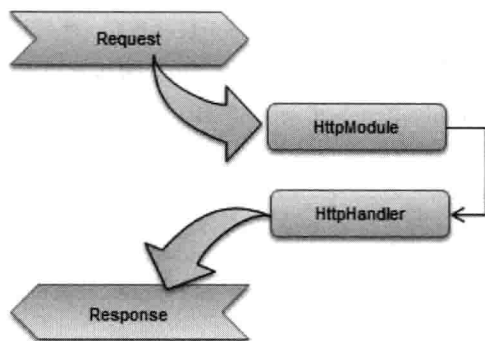


图 2-14 通过 HttpModule 实现针对 HttpHandler 的动态映射

2.3.1 UrlRoutingModule

在 ASP.NET 的路由系统中，图 2-14 所示的作为请求拦截器的 HttpModule 类型为 UrlRoutingModule。如下面的代码片段所示，UrlRoutingModule 对请求的拦截是通过注册代表当前应用的 HttpApplication 对象的 PostResolveRequestCache 事件实现的。

```

public class UrlRoutingModule : IHttpModule
{
    //其他成员
    public RouteCollection RouteCollection { get; set; }

    public void Init(HttpApplication context)
    {
        context.PostResolveRequestCache +=
            new EventHandler(this.OnApplicationPostResolveRequestCache);
    }

    private void OnApplicationPostResolveRequestCache(object sender,
        EventArgs e);
}
  
```

UrlRoutingModule 具有一个类型为 RouteCollection 的 RouteCollection 属性，在默认情况下该属性是对 RouteTable 的静态属性 Routes 的引用。HttpHandler 的动态映射就实现在 OnApplicationPostResolveRequestCache 方法中。当该方法被执行的时候，它会利用指定的 HttpApplication 得到表示当前 HTTP 上下文的 HttpContext 对象（对应于 HttpApplication 类型的 Context 属性），然后根据它创建一个 HttpContextWrapper 对象。

UrlRoutingModule 接下来将此 HttpContextWrapper 对象作为参数调用 RouteCollection 对象的 GetRouteData 方法对当前请求实施路由解析。如果方法调用返回一个具体的 RouteData 对象，

那么它会通过其 `RouteHandler` 属性得到对应 `Route` 采用的 `RouteHandler`，然后调用这个 `RouteHandler` 对象的 `GetHttpHandler` 方法得到这个需要被动态映射的 `HttpHandler` 对象。定义在 `UrlRoutingModule` 类型的 `OnApplicationPostResolveRequestCache` 方法中的 `HttpHandler` 动态映射逻辑基本上体现在如下所示的代码片段中。

```
public class UrlRoutingModule : IHttpModule
{
    //其他成员
    private void OnApplicationPostResolveRequestCache(object sender, EventArgs e)
    {
        HttpContext context = ((HttpApplication)sender).Context;
        HttpContextBase contextWrapper = new HttpContextWrapper(context);
        RouteData routeData = this.RouteCollection.GetRouteData(contextWrapper);
        RequestContext requestContext =
            new RequestContext(contextWrapper, routeData);
        IHttpHandler handler =
            routeData.RouteHandler.GetHttpHandler(requestContext);
        context.RemapHandler(handler);
    }
}
```

2.3.2 PageRouteHandler 与 MvcRouteHandler

通过前面的介绍我们知道，对于通过调用 `RouteCollection` 的 `GetRouteData` 方法获得的 `RouteData` 对象来说，其 `RouteHandler` 来源于创建它的 `Route` 对象；对于通过调用 `RouteCollection` 的 `MapPageRoute` 方法注册的 `Route` 来说，它的 `RouteHandler` 属性返回一个 `PageRouteHandler` 对象。

由于调用 `MapPageRoute` 方法的目的在于实现请求地址与某个 `.aspx` 页面文件之间的映射，我们最终还是要创建一个 `Page` 对象来处理该请求，`PageRouteHandler` 的 `GetHttpHandler` 方法最终返回的就是一个针对映射 `.aspx` 页面的 `Page` 对象。除此之外，`MapPageRoute` 方法还可以控制是否对物理文件地址实施授权，而授权检验在返回 `Page` 对象之前进行。

定义在 `PageRouteHandler` 中的 `HttpHandler` 映射逻辑基本上体现在如下的代码片段中。它的属性 `VirtualPath` 表示页面文件的虚拟路径，而 `CheckPhysicalUrlAccess` 属性则表示是否需要物理文件地址实施 URL 授权检验。这两个属性均在构造函数中被初始化，且最初来源于调用 `RouteCollection` 的 `MapPageRoute` 方法传入的参数。

```
public class PageRouteHandler : IRouteHandler
{
    public bool        CheckPhysicalUrlAccess { get; private set; }
    public string      VirtualPath { get; private set; }
}
```

```

public PageRouteHandler(string virtualPath, bool checkPhysicalUrlAccess)
{
    this.VirtualPath = virtualPath;
    this.CheckPhysicalUrlAccess = checkPhysicalUrlAccess;
}

public IHttpHandler GetHttpHandler(RequestContext requestContext)
{
    if (this.CheckPhysicalUrlAccess)
    {
        //Check Physical Url Access
    }
    return (IHttpHandler)BuildManager.CreateInstanceFromVirtualPath(
        this.VirtualPath, typeof(Page)) ;
}
}

```

对于一个 ASP.NET MVC 应用来说, Route 对象是通过调用 RouteCollection 的扩展方法 MapRoute 进行注册的, 它的 RouteHandler 属性返回一个 MvcRouteHandler 对象。如下面的代码片段所示, MvcRouteHandler 提供的用于处理当前请求的 HttpHandler 是一个 MvcHandler 对象。MvcHandler 实现对 Controller 的激活、Action 方法的执行及对请求的响应。毫不夸张地说, 整个 MVC 框架就实现在这个 MvcHandler 之中。

```

public class MvcRouteHandler : IRouteHandler
{
    //其他成员
    public IHttpHandler GetHttpHandler(RequestContext requestContext)
    {
        return new MvcHandler(requestContext) ;
    }
}

```

2.3.3 ASP.NET 路由系统扩展

到此为止, 我们已经对 ASP.NET 的路由系统的实现进行了详细地介绍。总的来说, 整个路由系统是通过 HttpHandler 的动态注册的方式来实现的。具体来说, 注册的 UrlRoutingModule 通过对代表当前应用的 HttpApplication 对象的 PostResolveRequestCache 事件进行注册实现了对请求的拦截。

对于被拦截的请求, UrlRoutingModule 利用注册的路由表对其实施路由解析, 进而得到一个包含所有路由数据的 RouteData 对象, 并借助此 RouteData 对象的 RouteHandler 得到相应的 HttpHandler。该 HttpHandler 最终被 UrlRoutingModule 映射到当前 HTTP 上下文用以处理当前请求。从可扩展性的角度来讲, 可以通过如下 3 种方式来定制我们需要的路由。

- 通过继承抽象类 `RouteBase` 创建自定义 `Route` 类型定制路由逻辑。
- 通过实现接口 `IRouteHandler` 创建自定义 `RouteHandler` 定制 `HttpHandler` 提供机制。
- 通过实现 `IHttpHandler` 创建自定义 `HttpHandler` 来对请求进行处理并作最终的响应。

2.3.4 实例演示:通过自定义 `Route` 对 ASP.NET 路由系统进行扩展 (S214)

如果我们对 WCF REST 有一定的了解,应该知道它也具有自己的路由系统,它借助于一个 `UriTemplate` 对象实现针对模板的路由映射。现在我们通过一个实例来演示如何借助于一个自定义的 `Route` 利用 `UriTemplate` 来实现不一样的路由。

我们创建一个 ASP.NET Web 应用,并且添加针对程序集“`System.ServiceModel.dll`”的引用(`UriTemplate` 类型就定义在该程序集中)。我们在这个 Web 应用中定义如下一个针对 `UriTemplate` 的 `UriTemplateRoute` 类。

```
public class UriTemplateRoute:RouteBase
{
    public UriTemplate                UriTemplate { get; private set; }
    public IRouteHandler              RouteHandler { get; private set; }
    public RouteValueDictionary       DataTokens { get; private set; }

    public UriTemplateRoute(string template, string physicalPath,
        object dataTokens = null)
    {
        this.UriTemplate = new UriTemplate(template);
        this.RouteHandler = new PageRouteHandler(physicalPath);
        if (null != dataTokens)
        {
            this.DataTokens = new RouteValueDictionary(dataTokens);
        }
        else
        {
            this.DataTokens = new RouteValueDictionary();
        }
    }

    public override RouteData GetRouteData(HttpContextBase httpContext)
    {
        Uri uri = httpContext.Request.Url;
        Uri baseAddress = new Uri(string.Format("{0}://{1}",
            uri.Scheme, uri.Authority));
        UriTemplateMatch match = this.UriTemplate.Match(baseAddress, uri);
        if (null == match)
        {
            return null;
        }
    }
}
```

```

    }
    RouteData routeData = new RouteData();
    routeData.RouteHandler = this.RouteHandler;
    routeData.Route = this;
    foreach (string name in match.BoundVariables.Keys)
    {
        routeData.Values.Add(name, match.BoundVariables[name]);
    }
    foreach (var token in this.DataTokens)
    {
        routeData.DataTokens.Add(token.Key, token.Value);
    }
    return routeData;
}

public override VirtualPathData GetVirtualPath(RequestContext requestContext,
    RouteValueDictionary values)
{
    Uri uri = requestContext.HttpContext.Request.Url;
    Uri baseAddress = new Uri(string.Format("{0}://{1}",
        uri.Scheme, uri.Authority));
    Dictionary<string, string> variables = new Dictionary<string, string>();
    foreach (var item in values)
    {
        variables.Add(item.Key, item.Value.ToString());
    }

    //确定段变量是否被提供
    foreach (var name in this.UriTemplate.PathSegmentVariableNames)
    {
        if (!this.UriTemplate.Defaults.Keys.Any(
            key => string.Compare(name, key, true) == 0) &&
            !values.Keys.Any(key => string.Compare(name, key, true) == 0))
        {
            return null;
        }
    }

    //确定查询变量是否被提供
    foreach (var name in this.UriTemplate.QueryValueVariableNames)
    {
        if (!this.UriTemplate.Defaults.Keys.Any(
            key => string.Compare(name, key, true) == 0) &&
            !values.Keys.Any(key => string.Compare(name, key, true) == 0))
        {
            return null;
        }
    }

    Uri virtualPath = this.UriTemplate.BindByName(baseAddress, variables);
    string strVirtualPath = virtualPath.ToString().ToLower()

```

```

        .Replace(baseAddress.ToString().ToLower(), "");
        VirtualPathData virtualPathData = new VirtualPathData(this,
            strVirtualPath);
        foreach (var token in this.DataTokens)
        {
            virtualPathData.DataTokens.Add(token.Key, token.Value);
        }
        return virtualPathData;
    }
}

```

如上面的代码片段所示，继承自抽象类 `RouteBase` 的 `UriTemplateRoute` 具有 `UriTemplate`、`DataTokens` 和 `RouteHandler` 3 个只读属性，前两个属性通过构造函数的参数进行初始化，后者则是在构造函数中创建的 `PageRouteHandler` 对象。

在用于对入栈请求实施路由解析并生成路由数据的 `GetRouteData` 方法中，我们解析出应用的基地址并连同请求 URL 作为参数调用 `UriTemplate` 对象的 `Match` 方法。如果方法调用返回一个具体的 `UriTemplateMatch` 对象，则意味着路由模板的模式与请求 URL 匹配。在此情况下我们会针对解析出来的路由变量创建一个 `RouteData` 对象并返回。

至于用于生成出栈 URL 的 `GetVirtualPath` 方法，我们通过判断定义在路由模板中的变量是否存在于提供的 `RouteValueDictionary` 对象或者默认变量列表（通过属性 `Defaults` 表示）中来确定路由模板是否与提供的变量列表匹配。在匹配的情况下我们调用 `UriTemplate` 对象的 `BindByName` 方法得到一个完整的 URL。由于 `GetVirtualPath` 方法返回的是相对路径，所以我们需要将应用基地址剔除并最终创建返回的 `VirtualPathData` 对象。如果不匹配，则直接返回 `Null`。

在创建的 `Global.asax` 文件中采用如下的代码对自定义的 `UriTemplateRoute` 进行注册，选用的场景还是之前采用的天气预报的例子。笔者个人觉得基于 `UriTemplate` 的路由模板比针对 `Route` 的模板更好用，其中一点就是它定义默认值的方式更为直接。如下面的代码片段所示，可以直接将默认值定义在模板中（`{areacode=010}/{days=2}`）。

```

public class Global : System.Web.HttpApplication
{
    protected void Application_Start(object sender, EventArgs e)
    {
        UriTemplateRoute route = new UriTemplateRoute("{areacode=010}/{days=2}",
            "~/Weather.aspx", new { defaultCity = "BeiJing", defaultDays = 2 });
        RouteTable.Routes.Add("default", route);
    }
}

```

在注册的 `Route` 指向的目标页面 `Weather.aspx` 的后台代码中，我们定义了如下一个 `GenerateUrl` 方法，它会根据指定的区号（`areacode`）和预报天数（`days`）创建一个 URL。在该

方法中, 我们通过 `RouteTable` 的静态属性 `Routes` 得到代表全局路由表的 `RouteCollection` 对象, 并调用其 `GetVirtualPathData` 方法来生成最终返回的 URL。

```
public partial class Weather : System.Web.UI.Page
{
    public string GenerateUrl(string areacode, int days)
    {
        var values = new { areacode = areacode, days = days };
        RequestContext requestContext = new RequestContext();
        requestContext.HttpContext = new HttpContextWrapper(HttpContext.Current);
        requestContext.RouteData = RouteData;
        return RouteTable.Routes.GetVirtualPath(requestContext,
            new RouteValueDictionary(values)).VirtualPath;
    }
}
```

通过调用 `GenerateUrl` 方法生成的 URL(`areaCode=0512;days=3`)连同当前页面的 `RouteData` 的属性通过如下所示的 HTML 代码输出来。

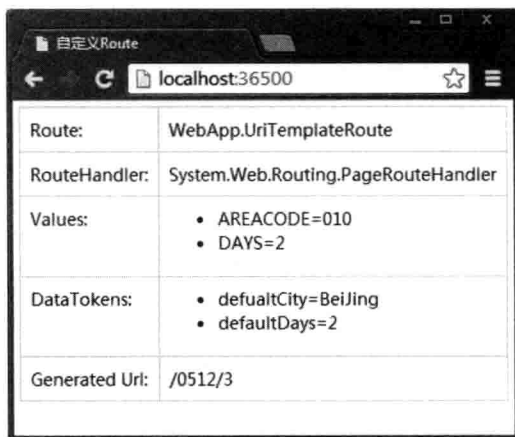
```
<form id="form1" runat="server">
    <div>
        <table>
            <tr>
                <td>Route:</td>
                <td><%=RouteData.Route != null?
                    RouteData.Route.GetType().FullName:"" %></td>
            </tr>
            <tr>
                <td>RouteHandler:</td>
                <td><%=RouteData.RouteHandler != null?
                    RouteData.RouteHandler.GetType().FullName:"" %></td>
            </tr>
            <tr>
                <td>Values:</td>
                <td>
                    <ul>
                        <%foreach (var variable in RouteData.Values)
                        {%>
                            <li>
                                <%=variable.Key%>=<%=variable.Value%></li>
                            <% }%>
                        </ul>
                    </td>
            </tr>
            <tr>
                <td>DataTokens:</td>
                <td>
                    <ul>
                        <%foreach (var variable in RouteData.DataTokens)
```

```

        {<%=variable.Key%>=<%=variable.Value%></li>
        <%=variable.Key%>=<%=variable.Value%></li>
        }%>
    </ul>
</td>
</tr>
<tr>
    <td>Generated Url:</td>
    <td>
        <%=GenerateUrl("0512",3)%>
    </td>
</tr>
</table>
</div>
</form>

```

由于注册的路由模板所包含的段均由具有默认值的变量构成,所以当我们请求应用的根地址时,请求会自动路由到 `Weather.aspx` 页面。如图 2-15 所示是我们在浏览器中访问应用根目录的截图,上面显示了注册的 `UriTemplateRoute` 生成的 `RouteData` 的信息和生成的 URL(`/0512/3`)。



Route:	WebApp.UriTemplateRoute
RouteHandler:	System.Web.Routing.PageRouteHandler
Values:	<ul style="list-style-type: none"> AREACODE=010 DAYS=2
DataTokens:	<ul style="list-style-type: none"> defaultCity=Beijing defaultDays=2
Generated Uri:	/0512/3

图 2-15 通过自定义 `UriTemplateRoute` 得到的 `RouteData` 和生成的 URL

第3章 Controller 的激活

不同于传统的 Web Forms 应用, ASP.NET MVC 应用中请求的目标不再是具体某个物理文件, 而是定义在 Controller 类型中的某个 Action 方法。请求经过路由解析之后, 目标 Controller 的名称会被作为路由变量出现在生成的 RouteData 对象中, ASP.NET MVC 会据此解析出目标 Controller 的类型, 进而激活对应的 Controller 对象。

3.1 Controller 激活系统全景展示

我们将整个 ASP.NET MVC 框架人为地划分为若干个子系统，那么针对请求上下文激活目标 Controller 对象的子系统可以称为 Controller 激活系统。在正式讨论 Controller 对象具体是如何被激活之前，我们先来了解一个 Controller 激活系统在 ASP.NET MVC 中的总体设计，看看它大体上由哪些对象构成。

3.1.1 Controller

本书范围内的 Controller 泛指实现了 IController 接口的某个类型的对象。Controller 是一个可执行的对象，它的执行体现在对其方法 Execute 的调用。如下面的代码片段所示，定义在 IController 接口的这个唯一的 Execute 方法具有一个 RequestContext 类型的参数。当目标 Controller 对象被激活之后，对请求的后续处理和最终响应均通过执行这个 Execute 方法来完成。

```
public interface IController
{
    void Execute(RequestContext requestContext);
}
```

定义在 IController 接口中的 Execute 方法是以同步的方式执行的。为了异步方式，另一个名为 IAsyncController 的接口被定义在“System.Web.Mvc.Async”命名空间下。如下面的代码片段所示，IAsyncController 接口派生于 IController 接口，Controller 的异步执行通过先后调用 BeginExecute/EndExecute 方法来完成。

```
public interface IAsyncController : IController
{
    IAsyncResult BeginExecute(RequestContext requestContext,
        AsyncCallback callback, object state);
    void EndExecute(IAsyncResult asyncResult);
}
```

默认作为所有 Controller 基类的 ControllerBase 实现了 IController 接口。如下面的代码片段所示，ControllerBase 是一个抽象类，它“显式”地实现了定义在 IController 接口的 Execute 方法。这个实现的 Execute 方法会调用受保护的虚方法 Execute，后者最终又会调用抽象方法 ExecuteCore。作为 ControllerBase 的继承者，必须通过实现这个抽象方法 ExecuteCore 来完成针对 Controller 的执行。

```
public abstract class ControllerBase : IController
{
    //其他成员
```

```

protected virtual void Execute(RequestContext requestContext);
protected abstract void ExecuteCore();
void IController.Execute(RequestContext requestContext);
protected virtual void Initialize(RequestContext requestContext)

public TempDataDictionary TempData { get; set; }
public object ViewBag { [return: Dynamic] get; }
public ViewDataDictionary ViewData { get; set; }
public ControllerContext ControllerContext { get; set; }
}

```

一般来说, 目标 Controller 在执行之后会呈现一个 View 来作为对请求的最终响应。在进行 View 呈现之前, Controller 需要向它传递一些变量, 这些变量可以存储在 ControllerBase 类型的 3 个属性 (TempData、ViewBag 和 ViewData) 中。

TempData 和 ViewData 属性均返回一个具有字典结构的数据容器, 其 Key 和 Value 分别代表变量的名称和值。两者的不同之处在于前者仅仅用于存储临时数据, 并且设置的变量在被第一次读取之后会被移除, 换句话说通过 TempData 设置的变量只能被读取一次。ViewBag 和 ViewData 属性是同一份数据的不同表现形式, 二者的不同之处在于前者是一个动态对象, 可以为其指定任意属性 (动态属性名将作为数据字典的 Key)。

在 ASP.NET MVC 中我们会陆续遇到一系列的上下文 (Context) 对象, 之前已经对表示请求上下文的 RequestContext (HttpContext + RouteData) 进行了详细的介绍, 我们现在来介绍另一个具有如下定义的上下文类型 ControllerContext。顾名思义, ControllerContext 就是基于某个 Controller 对象的上下文。

```

public class ControllerContext
{
    //其他成员
    public ControllerContext();
    public ControllerContext(RequestContext requestContext,
        ControllerBase controller);
    public ControllerContext(HttpContextBase httpContext,
        RouteData routeData, ControllerBase controller);

    public virtual ControllerBase Controller { get; set; }
    public virtual RequestContext RequestContext { get; set; }
    public virtual HttpContextBase HttpContext { get; set; }
    public virtual RouteData RouteData { get; set; }
}

```

从如上的代码可以看出, 一个 ControllerContext 对象实际上是对一个 Controller 对象和 RequestContext 对象的封装。ControllerContext 的属性 HttpContext 和 RouteData 分别返回当前的 HTTP 上下文和当前请求经过路由解析生成的 RouteData 对象, 实际上这两个对象也可以从表示请求上下文的 RequestContext 对象中提取。

从上面给出的针对 `ControllerBase` 类型的定义可以看出, `ControllerBase` 定义了一个受保护的虚方法 `Initialize` 来执行一些初始化操作。该方法具有一个类型为 `RequestContext` 的输入参数, 当此方法被执行的时候, 一个 `ControllerContext` 对象会根据这个 `RequestContext` 创建出来并作为其 `ControllerContext` 属性的值。对于定义在 `ControllerBase` 中的受保护 `Execute` 方法来说, 当它在执行 `ExecuteCore` 方法之前会执行这个 `Initialize` 方法。

通过 Visual Studio 的向导创建的 `Controller` 类型实际上继承自抽象类 `Controller`, 它是 `ControllerBase` 的子类。如下面的代码片段所示, 除了直接继承 `ControllerBase` 之外, `Controller` 类型还显式地实现了 `IController` 和 `IAsyncController` 接口, 以及代表 ASP.NET MVC 5 种过滤器的接口, 我们会在第 12 章“过滤器”中对过滤器进行详细介绍。

```
public abstract class Controller : ControllerBase,
    IAsyncController,
    IAuthenticationFilter,
    IAuthorizationFilter,
    IActionFilter,
    IExceptionFilter,
    IResultFilter,
    IDisposable,
    IAsyncManagerContainer
{
    //省略成员
}
```

抽象类 `Controller` 实现的另一个接口 `IAsyncManagerContainer` 提供一个 `AsyncManager` 对象为异步操作的执行提供参数传递、操作计数和超时控制等功能, 针对它的介绍放在第 10 章“Action 方法的执行”中。除此之外, `Controller` 还实现了 `IDisposable` 接口, ASP.NET MVC 的 `Controller` 激活系统在 `Controller` 执行结束之后会调用其 `Dispose` 方法以完成相应的资源回收工作。

同步还是异步

从抽象类 `Controller` 的定义可以看出它实现了 `IAsyncController` 接口, 而该接口继承自接口 `IController`, 意味着它既可以采用同步的方式(调用 `Execute` 方法)执行, 也可以采用异步的方式(调用 `BeginExecute/EndExecute` 方法)执行。但是即使调用 `BeginExecute/EndExecute` 方法, `Controller` 也不一定是以异步方式执行的。

如下面的代码片段所示, `Controller` 具有一个布尔类型的属性 `DisableAsyncSupport`, 表示是否关闭对异步执行的支持。在默认情况下该属性总是返回 `False`, 即支持以异步方式执行 `Controller`。当 `BeginExecute` 方法被执行的时候, 它会根据该属性决定究竟是调用 `Execute` 方法

以同步的方式执行 Controller，还是调用 BeginExecuteCore/EndExecuteCore 方法以异步的方式执行 Controller。换句话说，如果我们希望 Controller 总是以同步的方式来执行，可以将 DisableAsyncSupport 属性设置为 True。

```
public abstract class Controller: ...
{
    //其他成员
    protected virtual bool DisableAsyncSupport
    {
        get{return false;}
    }

    protected virtual IAsyncResult BeginExecute(RequestContext requestContext,
        AsyncCallback callback, object state)
    {
        if (this.DisableAsyncSupport)
        {
            //通过调用 Execute 方法同步执行 Controller
        }
        else
        {
            //通过调用 BeginExecuteCore/EndExecuteCore 方法异步执行 Controller
        }
    }
    protected virtual IAsyncResult BeginExecuteCore(AsyncCallback callback,
        object state);
    protected virtual void EndExecuteCore(IAsyncResult asynResult);
}
```

现在我们通过一个简单的实例来演示属性 DisableAsyncSupport 对执行 Controller 的影响。我们在一个 ASP.NET MVC 应用中定义了一个 HomeController 类型，它重写了 Execute、ExecuteCore、BeginExecute/EndExecute 和 BeginExecuteCore/EndExecuteCore 6 个方法。在这些重写的方法中，我们将相应的方法名写入响应并最终将其呈现在浏览器上。

```
public class HomeController : Controller
{
    public new HttpResponse Response
    {
        get { return System.Web.HttpContext.Current.Response; }
    }

    protected override void Execute(RequestContext requestContext)
    {
        Response.Write("Execute(); <br/>");
        base.Execute(requestContext);
    }

    protected override void ExecuteCore()
    {

```

```

        Response.Write("ExecuteCore(); <br/>");
        base.ExecuteCore();
    }

    protected override IAsyncResult BeginExecute(RequestContext requestContext,
        AsyncCallback callback, object state)
    {
        Response.Write("BeginExecute(); <br/>");
        return base.BeginExecute(requestContext, callback, state);
    }

    protected override void EndExecute(IAsyncResult asyncResult)
    {
        Response.Write("EndExecute(); <br/>");
        base.EndExecute(asyncResult);
    }

    protected override IAsyncResult BeginExecuteCore(AsyncCallback callback,
        object state)
    {
        Response.Write("BeginExecuteCore(); <br/>");
        return base.BeginExecuteCore(callback, state);
    }

    protected override void EndExecuteCore(IAsyncResult asyncResult)
    {
        Response.Write("EndExecuteCore(); <br/>");
        base.EndExecuteCore(asyncResult);
    }

    public ActionResult Index()
    {
        return Content("Index();<br/>");
    }
}

```

值得一提的是, 虽然抽象类 `Controller` 中定义了一个表示当前响应的属性 `Response`, 但是当 `BeginExecute` 方法执行的时候该属性尚未初始化, 所以上面代码中使用的 `Response` 属性是我们自行定义的。运行该程序后会在浏览器中呈现出如图 3-1 所示的输出结果, 从输出方法的调用顺序中不难看出, 在默认情况下 `Controller` 是以异步方式执行的。(S301)

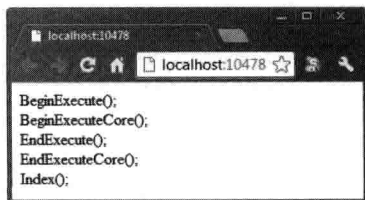


图 3-1 Controller 在默认情况下的异步执行方式

现在我们按照如下的方式重写虚属性 `DisableAsyncSupport`，使它直接返回 `True` 以关闭对 `Controller` 异步执行的支持。

```
public class HomeController : Controller
{
    //其他成员
    protected override bool DisableAsyncSupport
    {
        get{return true;}
    }
}
```

再次运行程序将会得到如图 3-2 所示的输出结果，我们可以看出，由于 `HomeController` 间接地实现了 `IAsyncController` 接口，所以 `Controller` 的执行总是通过调用 `BeginExecute/EndExecute` 方法以异步方式来完成。但是由于 `DisableAsyncSupport` 属性被设置为 `True`，所以 `BeginExecute` 方法内部会以同步的方式调用 `Execute/ExecuteCore` 方法。（S302）

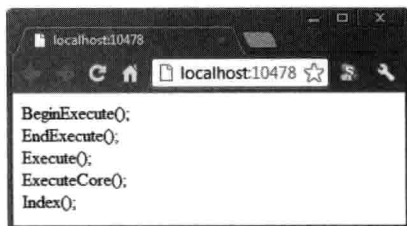


图 3-2 `Controller` 在 `DisableAsyncSupport` 属性为 `True` 的情况下的同步执行方式

在 ASP.NET MVC 应用编程接口中还定义了一个 `AsyncController` 类型。从名称上看，`AsyncController` 是一个异步 `Controller`，但是这里的异步并不是指 `Controller` 的异步执行，而是 `Action` 方法的异步执行。从如下的代码片段中可以看出，这个直接继承自抽象类 `Controller` 的 `AsyncController` 是一个“空”类型（没有额外定义和重写基类的类型成员）。在 ASP.NET MVC 3.0 中，异步执行的 `Action` 通过两个方法以 `XxxAsync/XxxCompleted` 的形式定义，以这种方式定义的异步 `Action` 方法必须定义在继承自 `AsyncController` 的类型中。考虑到向后兼容性，`AsyncController` 在 ASP.NET MVC 4.0 和 ASP.NET MVC 5.0 中均被保留了下来。

```
public abstract class AsyncController : Controller
{
    protected AsyncController() {}
}
```

值得强调的是，只有以传统方式（`XxxAsync/XxxCompleted`）定义的异步 `Action` 方法才需要定义在继承自 `AsyncController` 的 `Controller` 中。ASP.NET MVC 4.0 提供了新的异步 `Action` 方

法定义方式，它使我们可以通过一个返回类型为 `Task` 的方法来定义以异步方式执行的 `Action`，这样的 `Action` 方法并不需要定义在 `AsyncController` 中。

3.1.2 ControllerFactory

ASP.NET MVC 为 `Controller` 的激活定义了相应的工厂，我们将其称为 `ControllerFactory`，所有的 `ControllerFactory` 类型都实现了 `IControllerFactory` 接口。如下面的代码片段所示，`Controller` 对象的激活最终通过调用 `CreateController` 方法来完成，该方法两个参数分别表示当前请求上下文和通过路由解析得到的目标 `Controller` 的名称。

```
public interface IControllerFactory
{
    IController CreateController(RequestContext requestContext,
        string controllerName);
    SessionStateBehavior GetControllerSessionBehavior(
        RequestContext requestContext, string controllerName);
    void ReleaseController(IController controller);
}

public enum SessionStateBehavior
{
    Default,
    Required,
    ReadOnly,
    Disabled
}
```

除了负责创建 `Controller` 对象来处理并响应请求之外，`ControllerFactory` 还需要在请求处理结束之后释放这个 `Controller` 对象。`ControllerFactory` 对 `Controller` 对象的释放实现在方法 `ReleaseController` 中。`IControllerFactory` 的另一个方法 `GetControllerSessionBehavior` 返回一个类型为 `SessionStateBehavior` 的枚举值。熟悉 ASP.NET 的读者应该不会对 `SessionStateBehavior` 感到陌生，它表示请求处理过程中会话状态支持的模式，它的 4 个枚举值分别具有如下的含义。

- **Default:** 使用默认 ASP.NET 逻辑来确定请求的会话状态行为。
- **Required:** 为请求启用完全的读/写会话状态行为。
- **ReadOnly:** 为请求启用只读会话状态。
- **Disabled:** 禁用会话状态。

对于 `Default` 选项来说，ASP.NET 通过判断映射的 `HttpHandler` 类型是否实现了相关接口来决定具体的会话状态控制行为。在 “`System.Web.SessionState`” 命名空间下定义了 `IRequiresSessionState` 和 `IReadOnlySessionState` 两个接口。如下面的代码片段所示，这两个都是

不具有任何成员的空接口（一般称之为标记接口），接口 `IRedOnlySessionState` 继承自接口 `IRequiresSessionState`。如果 `HttpHandler` 实现了接口 `IRedOnlySessionState`，则意味着采用 `ReadOnly` 模式，如果只实现了 `IRequiresSessionState` 接口则采用 `Required` 模式。

```
public interface IRequiresSessionState {}
public interface IRedOnlySessionState : IRequiresSessionState {}
```

具体采用何种会话状态行为取决于当前 HTTP 上下文(通过 `HttpContext` 的静态属性 `Current` 表示)。对于 ASP.NET 3.0 及之前的版本来说，我们不能对当前 HTTP 上下文的会话状态行为模式进行动态的修改，ASP.NET 4.0 为 `HttpContext` 定义了如下一个 `SetSessionStateBehavior` 方法，使我们可以自由地选择会话状态行为模式。相同的方法同样定义在 `HttpContextBase` 中，它的子类 `HttpContextWrapper` 重写了这个方法，并在内部调用封装的 `HttpContext` 的同名方法来定制当前请求的会话模式。

```
public sealed class HttpContext : IServiceProvider, IPrincipalContainer
{
    //其他成员
    public void SetSessionStateBehavior(
        SessionStateBehavior sessionStateBehavior);
}

public class HttpContextBase: IServiceProvider
{
    //其他成员
    public void SetSessionStateBehavior(
        SessionStateBehavior sessionStateBehavior);
}
```

3.1.3 ControllerBuilder

用于激活 `Controller` 对象的 `ControllerFactory` 最终通过 `ControllerBuilder` 注册到 ASP.NET MVC 框架之中。如下面的代码所示，`ControllerBuilder` 定义了一个静态只读属性 `Current` 返回当前使用的 `ControllerBuilder` 对象。两个 `SetControllerFactory` 方法重载实现了针对 `ControllerFactory` 的注册，它们的不同之处在于前者注册的是 `ControllerFactory` 的类型，后者注册的则是一个具体的 `ControllerFactory` 对象。另一个方法 `GetControllerFactory` 返回一个具体的 `ControllerFactory` 对象。

```
public class ControllerBuilder
{
    public IControllerFactory GetControllerFactory();

    public void SetControllerFactory(Type controllerFactoryType);
```



```

public void SetControllerFactory(IControllerFactory controllerFactory);

public HashSet<string>           DefaultNamespaces { get; }
public static ControllerBuilder Current { get; }
}

```

如果我们注册的是 `ControllerFactory` 的类型，那么 `GetControllerFactory` 在执行的时候会通过对注册类型的反射（调用 `Activator` 的静态方法 `CreateInstance`）的方式来创建具体的 `ControllerFactory` 对象。也就是说针对 `GetControllerFactory` 方法的每次调用总是伴随目标 `ControllerFactory` 对象的实例化，ASP.NET MVC 并不会对创建的 `Controller` 进行缓存。如果注册的是一个具体的 `ControllerFactory` 对象，该对象直接从 `GetControllerFactory` 方法中返回。所以从性能方面考量，直接注册 `ControllerFactory` 对象是更好的选择。

通过第 2 章“路由”的介绍我们知道，路由系统对请求实施路由解析之后会生成一个 `RouteData` 对象，目标 `Controller` 名称对应的路由变量会包含在这个 `RouteData` 对象的 `Values` 属性表示的 `RouteValueDictionary` 对象中，对应的 `Key` 为“controller”。在默认情况下，目标 `Controller` 的名称只能帮助我们解析出不包括命名空间的类型名称，如果在不同的命名空间下定义了多个同名的 `Controller` 类，则会导致激活系统无法确定具体的 `Controller` 的类型，从而抛出异常。

为了解决这个问题，我们必须为定义了同名 `Controller` 类型的命名空间设置不同的优先级。ASP.NET MVC 的 `Controller` 激活系统提供了两种提升命名空间优先级的方式，第一种方式就是在调用 `RouteCollection` 如下所示的扩展方法 `MapRoute` 时指定一个命名空间的列表。通过第 2 章“路由”的介绍我们知道，通过这种方式指定的命名空间列表会保存在 `Route` 对象的 `DataTokens` 属性表示的 `RouteValueDictionary` 对象中，对应的 `Key` 为“Namespaces”。

```

public static class RouteCollectionExtensions
{
    //其他成员
    public static Route MapRoute(this RouteCollection routes, string name,
                                string url, string[] namespaces);
    public static Route MapRoute(this RouteCollection routes, string name,
                                string url, object defaults, string[] namespaces);
    public static Route MapRoute(this RouteCollection routes, string name,
                                string url, object defaults, object constraints, string[] namespaces);
}

```

另一种提升命名空间优先级的方式就是将其添加到当前的 `ControllerBuilder` 中的默认命名空间列表中。如下面的代码片段所示，`ControllerBuilder` 具有一个 `HashSet<string>` 类型的只读属性 `DefaultNamespaces`，默认命名空间列表被存储在这里。对于这两种不同的命名空间优先级提升方式，前者（通过路由注册）指定命名空间具有更高的优先级。

```

public class ControllerBuilder
{

```

```
//其他成员
public HashSet<string> DefaultNamespaces { get; }
}
```

1. 实例演示：如何提升命名空间的优先级（S303，S304，S305）

为了让读者对两种提升 Controller 所在命名空间优先级的方式具有一个深刻的印象，我们来进行一个简单的实例演示。我们在一个 ASP.NET MVC 应用中创建了两个同名的 HomeController 类。如下面的代码片段所示，这两个 HomeController 类分别定义在命名空间 “Artech.MvcApp” 和 “Artech.MvcApp.Controllers” 下，定义其中的默认 Action 方法 Index 直接返回当前 Controller 类型的全名。

```
namespace Artech.MvcApp.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            return this.Content(this.GetType().FullName);
        }
    }
}

namespace Artech.MvcApp
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            return this.Content(this.GetType().FullName);
        }
    }
}
```

现在我们直接运行该 Web 应用。由于多个 Controller 与注册的路由规则相匹配，这会导致 Controller 激活系统无法确定哪个类型的 Controller 应该被选用，所以会出现如图 3-3 所示的错误。（S303）

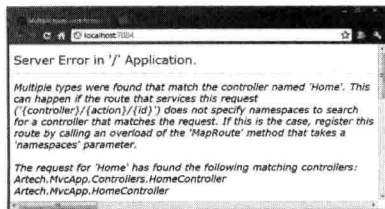


图 3-3 具有多个匹配 Controller 导致的异常

目前定义了 HomeController 类型的两个命名空间具有相同的优先级，我们现在将其中一个作为当前 ControllerBuilder 的默认命名空间以提升匹配优先级。如下面的代码片段所示，在 Global.asax 的 Application_Start 方法中，我们将命名空间 “Artech.MvcApp.Controllers” 添加到当前 ControllerBuilder 的 DefaultNamespaces 属性代表的默认命名空间列表中。

```
public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        //其他操作
        ControllerBuilder.Current.DefaultNamespaces
            .Add("Artech.MvcApp.Controllers");
    }
}
```

对于同时匹配注册的路由规则的两个 HomeController 来说，由于 “Artech.MvcApp.Controllers” 命名空间具有更高的匹配优先级，所有定义其中的 HomeController 会被选用，这可以通过如图 3-4 所示的运行结果看出来。(S304)

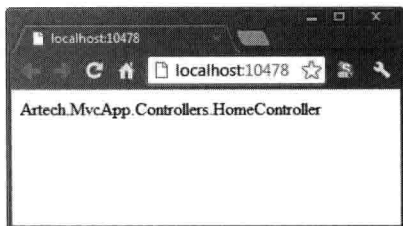


图 3-4 通过 ControllerBuilder 提升命名空间匹配优先级

为了检验在路由注册时指定的命名空间和作为当前 ControllerBuilder 的默认命名空间哪个具有更高匹配优先级，我们修改定义在 “App_Start/RouteConfig.cs” 中的路由注册代码。如下面的代码片段所示，我们在调用 RouteTable 的静态属性 Routes 的 MapRoute 方法进行路由注册的时候指定了命名空间 “Artech.MvcApp”。

```
public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        //其他操作
        routes.MapRoute(
            name      : "Default",
            url       : "{controller}/{action}/{id}",
            defaults  : new { controller = "Home", action = "Index",
```

```

        id = UrlParameter.Optional },
        namespaces : new string[] { "Artech.MvcApp" }
    );
}
}

```

再次运行程序会在浏览器中得到如图 3-5 所示的输出结果，从中可以看出定义在命名空间“Artech.MvcApp”中的 HomeController 被最终选用。可见较之作为当前 ControllerBuilder 的默认命名空间，在路由注册过程中指定的命名空间具有更高的匹配优先级，前者可以视为后者的一种后备。（S305）

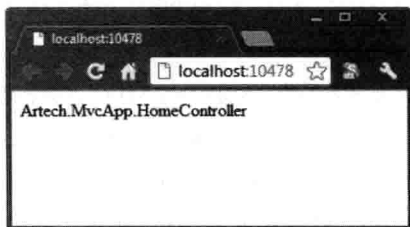


图 3-5 在路由注册时指定的命名空间具有更高的匹配优先级

在路由注册时指定的命名空间比当前 ControllerBuilder 的默认命名空间具有更高的匹配优先级，但是这两个集合中的所有命名空间却具有相同的匹配优先级。换句话说，用于辅助解析目标 Controller 类型的命名空间分为 3 个梯队，分别简称为“路由命名空间”、“ControllerBuilder 命名空间”和“Controller 类型自身的命名空间”。如果前一个梯队不能正确解析出 Controller 的类型，则后一个梯队的命名空间将作为后备。如果根据某个梯队的命名空间进行解析得到多个匹配的 Controller 类型，则会直接抛出如图 3-3 所示的异常。

2. 针对 Area 的路由对象的命名空间

针对某个 Area 的路由注册是通过相应的 AreaRegistration 对象实现的。具体来说，在应用启动的时候，AreaRegistration 的静态方法 RegisterAllAreas 被调用以实现针对所有 Area 的注册。在 RegisterAllAreas 方法被执行的过程中，定义在相关程序集中继承自抽象类型 AreaRegistration 的所有类型被成功解析出来。

针对每个具体的 AreaRegistration 类型，ASP.NET MVC 会采用反射的方式创建相应的实例。接下来，一个 AreaRegistrationContext 对象会根据当前 Area 的名称和全局路由表创建出来，它将成为参数调用此 AreaRegistration 对象的 RegisterArea 方法对相应的 Area 实施注册。

对于具体某个 AreaRegistration 类型来说，它会在重写的 RegisterArea 方法中调用作为参数

的 `AreaRegistrationContext` 对象的 `MapRoute` 方法实施针对当前 `Area` 的路由注册。如果我们在调用 `MapRoute` 方法中指定了一个命名空间列表，它将自动作为注册的 `Route` 对象的命名空间。如果这样的命名空间列表没有显式提供，则 `AreaRegistration` 类型所在的命名空间（会加上“.”后缀）会被添加到注册 `Route` 对象的命名空间列表。

这里所说的“`Route` 对象的命名空间”指的是一组表示命名空间的字符串数组，它以路由变量的形式存在于 `Route` 对象的 `DataTokens` 属性表示的 `RouteValueDictionary` 对象中，对应的 `Key` 为“`Namespaces`”。通过第2章“路由”的介绍我们知道，`Route` 对象的 `DataTokens` 属性包含的路由变量会转移到由它生成的 `RouteData` 的同名属性中，所以 `Controller` 激活系统可以从 `RouteData` 对象中获得这些命名空间。

除此之外，在调用 `AreaRegistrationContext` 的 `MapRoute` 方法时，ASP.NET MVC 还会在 `Route` 对象的 `DataTokens` 属性中添加一个 `Key` 为“`UseNamespaceFallback`”的路由变量，它表示是否采用后备命名空间来解析目标 `Controller` 类型。如果注册的 `Route` 对象具有命名空间（调用 `MapRoute` 方法时指定了命名空间或者对应的 `AreaRegistration` 类型定义在某个命名空间下），该变量的值为 `False`，否则为 `True`。

在解析目标 `Controller` 真实类型的过程中，ASP.NET MVC 会先使用 `RouteData` 包含的命名空间。如果解析失败，它会从 `RouteData` 对象的 `DataTokens` 属性中得到这个名为“`UseNamespaceFallback`”的变量值，并据此判断是否应该使用“后备”命名空间作进一步解析。具体来说，如果该值为 `True` 或者不存在，ASP.NET MVC 会先通过当前 `ControllerBuilder` 的命名空间进行解析，如果失败则直接忽略命名空间而采用类型名称进行匹配。如果该“`UseNamespaceFallback`”的变量存在并且值为 `False`，则 ASP.NET MVC 不会再使用“后备的”命名空间进行 `Controller` 类型的解析，而是直接抛出异常。

我们依然可以通过一个具体的例子来说明这个听起来貌似有些头晕的问题。我们在一个 ASP.NET MVC 应用中通过 `Area` 添加向导创建一个名称为“`Admin`”的 `Area`，此时 IDE 会默认为我们添加如下一个 `AdminAreaRegistration` 类型。

```
namespace MvcApp.Areas.Admin
{
    public class AdminAreaRegistration : AreaRegistration
    {
        public override string AreaName
        {
            get { return "Admin"; }
        }

        public override void RegisterArea(AreaRegistrationContext context)
        {
        }
    }
}
```

```

        context.MapRoute(
            "Admin_default",
            "Admin/{controller}/{action}/{id}",
            new { action = "Index", id = UrlParameter.Optional }
        );
    }
}

```

AdminAreaRegistration 类型定义在命名空间 “MvcApp.Areas.Admin” 下面。现在我们在该 Area 中添加如下一个 HomeController 类型。在默认 Action 方法 Index 中, 我们从当前 RouteData 的 DataTokens 属性中提取这个名为 “UseNamespaceFallback” 的变量值, 并将它和解析出来的 Controller 类型名称写入当前响应进而呈现在浏览器上。在默认情况下, 添加的 HomeController 类型被定义在 “MvcApp.Areas.Admin.Controllers” 命名空间下, 现在我们刻意将命名空间改为 “MvcApp.Areas.Controllers”。

```

namespace MvcApp.Areas.Controllers
{
    public class HomeController : Controller
    {
        public void Index()
        {
            Response.Write(string.Format("UseNamespaceFallback: {0}<br/>",
                RouteData.DataTokens["UseNamespaceFallback"]));
            Response.Write(string.Format("Controller Type: {0}<br/>",
                this.GetType().FullName));
        }
    }
}

```

现在利用浏览器中通过匹配的 URL (/admin/home/index) 来访问定义在 HomeController 中的 Action 方法 Index, 会得到如图 3-6 所示的 HTTP 状态为 “404, Not Found” 的响应, 这是因为目标 Controller 的真实类型是严格按照 AreaRegistration 所在的命名空间 “MvcApp.Areas.Controllers” 来匹配的, 很显然在这个命名空间下是不可能找到对应的 Controller 类型的。(S306)

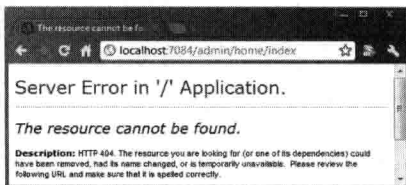


图 3-6 Controller 和 AreaRegistration 命名空间不匹配导致的 404 错误

但是如果我们去掉 `AdminAreaRegistration` 类型的命名空间, 那么将会导致路由变量 `UseNamespaceFallback` 的值变为 `True`, 这会促使 Controller 激活系统选择“后备”的命名空间。由于整个 Web 应用中仅仅定义了一个 `HomeController`, 很显然这个 Controller 会被激活, 如图 3-7 所示的程序运行结果也说明了这一点。(S307)

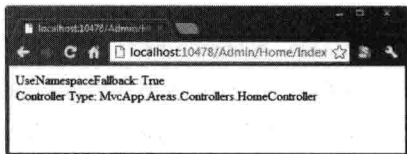


图 3-7 去掉 `AdminAreaRegistration` 命名空间以采用后备命名空间

3.1.4 Controller 的激活与路由

路由系统可以看成是 HTTP 请求被 IIS 分发给 ASP.NET 管道后的第一道屏障, 它利用注册的路由表对请求实施路由解析以生成包含目标 Controller 和 Action 名称在内的 `RouteData` 对象。目标 Controller 的激活是通过注册到当前 `ControllerBuilder` 上的 `ControllerFactory` 来完成的, 那么 Controller 的激活系统与路由系统两者是如何有机地结合起来的呢?

通过第 2 章“路由”的介绍我们知道, 整个 ASP.NET 的路由系统建立在一个叫作 `UrlRoutingModule` 的 `HttpModule` 之上, 针对请求的路由是它通过注册 `HttpApplication` 的事件 `PostResolveRequestCache` 为当前请求动态映射一个 `HttpHandler` 来实现的。具体来说, 它通过以 `RouteTable` 的静态属性 `Routes` 代表的全局路由表对请求实施路由解析并生成一个 `RouteData` 对象, 然后借助 `RouteData` 的 `RouteHandler` 属性得到最终被映射到当前请求的 `HttpHandler`。

默认情况下路由系统生成这个 `RouteData` 对象的 `RouteHandler` 是一个 `MvcRouteHandler` 对象。如下面的代码片段所示, `MvcRouteHandler` 维护着一个 `ControllerFactory` 对象, 该对象可以在构造函数中指定。如果在构造 `MvcRouteHandler` 对象的时候没有显式指定这个 `ControllerFactory` 对象, 那么它会调用当前 `ControllerBuilder` 对象的 `GetControllerFactory` 方法来得到这个 `ControllerFactory` 对象。

```
public class MvcRouteHandler : IRouteHandler
{
    private IControllerFactory _controllerFactory;

    public MvcRouteHandler(): this(ControllerBuilder.Current
        .GetControllerFactory())
    {}
}
```

```

public MvcRouteHandler(IControllerFactory controllerFactory)
{
    _controllerFactory = controllerFactory;
}

IHttpHandler IRouteHandler.GetHttpHandler(RequestContext requestContext)
{
    string controllerName = (string)requestContext.RouteData
        .GetRequiredString("controller");
    SessionStateBehavior sessionStateBehavior = _controllerFactory
        .GetControllerSessionBehavior(requestContext, controllerName);
    requestContext.HttpContext.SetSessionStateBehavior(sessionStateBehavior);

    return new MvcHandler(requestContext);
}
}

```

在用于提供 `HttpHandler` 的 `GetHttpHandler` 方法中，除了返回一个 `MvcHandler` 对象之外，`MvcRouteHandler` 还需要对当前 HTTP 上下文的会话状态行为模式进行设置。具体来说，该方法会先从包含在 `RequestContext` 的 `RouteData` 对象得到目标 `Controller` 的名称，此名称和 `RequestContext` 对象会作为参数调用 `ControllerFactory` 的 `GetControllerSessionBehavior` 方法得到一个类型为 `SessionStateBehavior` 的枚举。`MvcRouteHandler` 最后通过 `RequestContext` 对象得到当前 HTTP 上下文（实际上是一个 `HttpContextWrapper` 对象），并调用其 `SetSessionStateBehavior` 方法对会话状态行为进行设置。

通过第 2 章“路由”的介绍我们知道，`RouteData` 中的 `RouteHandler` 属性最初来源于对应的路由对象，对于调用 `RouteCollection` 的扩展方法 `MapRoute` 注册的 `Route` 对象来说，它对应的 `RouteHandler` 就是这么一个 `MvcRouteHandler` 对象。由于在创建 `MvcRouteHandler` 对象时并没有显式指定 `ControllerFactory`，所以通过调用当前 `ControllerBuilder` 对象的 `GetControllerFactory` 方法得到的 `ControllerFactory` 会默认被使用。

从上面的代码片段可以看出，通过当前 `ControllerBuilder` 的 `GetControllerFactory` 方法得到的 `ControllerFactory` 仅仅被 `MvcRouteHandler` 用来获取会话状态行为模式而已，只有 `MvcHandler` 才真正将它用于创建目标 `Controller` 对象。如下的代码片段基本上体现了 `MvcHandler` 的定义，它对请求处理的逻辑定义在 `BeginProcessRequest` 方法中。

```

public class MvcHandler :
    IHttpAsyncHandler,
    IHttpHandler,
    IRequiresSessionState
{
    //其他成员
    public RequestContext RequestContext { get; private set; }

    public bool IsReusable

```



```

    {
        get { return false; }
    }

    public MvcHandler(RequestContext requestContext)
    {
        this.RequestContext = requestContext;
    }

    IAsyncResult BeginProcessRequest(HttpContext context, AsyncCallback cb,
        object extraData)
    {
        IControllerFactory controllerFactory =
            ControllerBuilder.Current.GetControllerFactory();
        string controllerName =
            this.RequestContext.RouteData.GetRequiredString("controller");
        IController controller = controllerFactory
            .CreateController(this.RequestContext, controllerName);
        if (controller is IAsyncController)
        {
            {
                try
                {
                    //调用 BeginExecute/EndExecute 方法以异步的方式执行 Controller
                }
                finally
                {
                    controllerFactory.ReleaseController(controller);
                }
            }
        }
        else
        {
            {
                try
                {
                    //调用 Execute 方法以异步的方式执行 Controller
                }
                finally
                {
                    controllerFactory.ReleaseController(controller);
                }
            }
        }
    }
}

```

由于 `MvcHandler` 同时实现了 `IHandler` 和 `IAsyncHandler` 接口，所以它总是以异步的方式被执行（调用 `BeginProcessRequest/EndProcessRequest` 方法）。其 `BeginProcessRequest` 方法在执行的时候会通过 `RequestContext` 对象得到目标 `Controller` 的名称，然后利用当前 `ControllerBuilder` 提供的 `ControllerFactory` 对象激活目标 `Controller` 对象。如果 `Controller` 类型实现了 `IAsyncController` 接口，则以异步的方式执行 `Controller`，否则采用同步执行方式。在 `Controller` 对象被执行之后，`MvcHandler` 会调用 `ControllerFactory` 的 `ReleaseController` 方法对其实施释放清理工作。

3.2 Controller 默认激活机制

Controller 对象的激活最终通过注册的 ControllerFactory 来完成。如果没有调用当前 ControllerBuilder 的 SetControllerFactory 方法对 ControllerFactory 类型或者实例进行显式注册，则 Controller 激活系统默认会使用一个 DefaultControllerFactory 对象来激活目标 Controller，所以 DefaultControllerFactory 提供目标 Controller 对象的方式体现了 ASP.NET MVC 默认采用的 Controller 激活机制。

3.2.1 Controller 类型的解析

目标 Controller 对象能够被激活的根本前提是它的真实类型能够被成功解析出来。对于 DefaultControllerFactory 来说，用于解析目标 Controller 类型的辅助信息包括路由系统针对当前请求实施路由解析生成的 RouteData 对象（其中包含目标 Controller 的名称和命名空间）和包当前 ControllerBuilder 采用的默认命名空间。很多读者可能首先想到的 Controller 类型解析方案是：利用 Controller 名称得到对应的类型名称（Controller 的名称加上“Controller”后缀），并通过命名空间组成 Controller 类型的全名，最后遍历所有程序集并以此名称去加载相应的类型即可。

这貌似是一个“不错”的解决方案，实际上则完全行不通。不要忘了包含在 RouteData 中表示 Controller 名称的变量值是不区分大小写的，而类型名称则是大小写敏感的。除此之外，不论是注册路由时指定的命名空间还是当前 ControllerBuilder 采用的默认命名空间，有可能包含通配符(*)。基于这两点，我们根本不能通过给定的 Controller 名称和命名空间得到 Controller 的真实类型名称，自然就不可能通过名称去解析 Controller 的类型了。

默认采用的 DefaultControllerFactory 反其道而行之。它会先调用 BuildManager 的静态方法 GetReferencedAssemblies 得到所有可用的程序集，然后针对每个程序集通过反射的方式得到定义其中的所有实现了接口 IController 的类型，最后将 Controller 的名称和命名空间作为匹配条件去选择对应的 Controller 类型。在得到描述目标 Controller 的真实类型之后，DefaultControllerFactory 只需要采用反射的方式创建相应的对象即可。

实例演示：模拟 DefaultControllerFactory 的 Controller 激活机制（S308）

为了让读者对默认采用的 Controller 激活机制，尤其是目标 Controller 类型的解析机制有一个深刻的认识，我们通过一个自定义的 ControllerFactory 来模拟实现在 DefaultControllerFactory 中的 Controller 激活方式。由于需要采用反射的方式来创建 Controller 对象，所以我们将它命名为 ReflectedControllerFactory。

```

public class ReflectedControllerFactory : IControllerFactory
{
    //其他成员
    private static List<Type> controllerTypes;
    static ReflectedControllerFactory()
    {
        controllerTypes = new List<Type>();
        foreach (Assembly assembly in BuildManager.GetReferencedAssemblies())
        {
            controllerTypes.AddRange(assembly.GetTypes().Where(
                type => typeof(IController).IsAssignableFrom(type)));
        }
    }

    public IController CreateController(RequestContext requestContext,
        string controllerName)
    {
        Type controllerType = this.GetControllerType(requestContext.RouteData,
            controllerName);
        if (null == controllerType)
        {
            return null;
        }
        return (IController)Activator.CreateInstance(controllerType);
    }

    private static bool IsNamespaceMatch(string requestedNamespace,
        string targetNamespace)
    {
        if (!requestedNamespace.EndsWith(".*",
            StringComparison.OrdinalIgnoreCase))
        {
            return string.Equals(requestedNamespace, targetNamespace,
                StringComparison.OrdinalIgnoreCase);
        }
        requestedNamespace = requestedNamespace.Substring(0,
            requestedNamespace.Length - ".*".Length);
        if (!targetNamespace.StartsWith(requestedNamespace,
            StringComparison.OrdinalIgnoreCase))
        {
            return false;
        }
        return ((requestedNamespace.Length == targetNamespace.Length) ||
            (targetNamespace[requestedNamespace.Length] == '.'));
    }

    private Type GetControllerType(IEnumerable<string> namespaces,
        Type[] controllerTypes)
    {
        var types = (from type in controllerTypes
            where namespaces.Any(ns => IsNamespaceMatch(
                ns, type.Namespace))
            select type).ToArray();
    }
}

```

```

        switch (types.Length)
        {
            case 0: return null;
            case 1: return types[0];
            default: throw new InvalidOperationException("具有多个匹配的 Controller 类型");
        }
    }

    protected virtual Type GetControllerType(RouteData routeData,
        string controllerName)
    {
        //省略实现
    }
}

```

如上面的代码片段所示, `ReflectedControllerFactory` 具有一个静态的 `controllerTypes` 字段用于保存所有被解析出来的 `Controller` 的类型。在静态构造函数中, 我们调用 `BuildManager` 的 `GetReferencedAssemblies` 方法得到所有可用的程序集, 进而得到所有定义其中的实现了 `IController` 接口的类型, 这些类型全部被添加到通过静态字段 `controllerTypes` 表示的类型列表中。

目标 `Controller` 类型的解析实现在受保护的 `GetControllerType` 方法中。用于最终激活 `Controller` 对象的 `CreateController` 方法通过调用此方法得到与指定 `RequestContext` 和 `Controller` 名称相匹配的 `Controller` 类型, 并最终调用 `Activator` 的静态方法 `CreateInstance` 创建相应的 `Controller` 对象。

我们在 `ReflectedControllerFactory` 中定义了两个辅助方法, 其中 `IsNamespaceMatch` 用于判断 `Controller` 类型真正的命名空间是否与指定的命名空间 (可能包含通配符) 相匹配, 字符比较是忽略大小写的。私有方法 `GetControllerType` 根据指定的命名空间列表和类型名称从候选的 `Controller` 类型中选择一组完全匹配的 `Controller` 类型。如果得到多个匹配的类型, 它会直接抛出一个 `InvalidOperationException` 异常, 并提示具有多个匹配的 `Controller` 类型。如果找不到匹配类型则返回 `Null`。

在如下所示的用于解析 `Controller` 类型的 `GetControllerType` 方法中, 我们从预先得到的 `Controller` 类型候选列表中筛选出类型名称与传入的 `Controller` 名称相匹配的类型列表。接下来我们通过 `Route` 对象的命名空间对得到的 `Controller` 类型列表作进一步筛选, 如果能够找到一个唯一的类型, 则直接将其作为目标 `Controller` 类型。

```

public class ReflectedControllerFactory : IControllerFactory
{
    //其他成员
    protected virtual Type GetControllerType (RouteData routeData,
        string controllerName)
    {

```

```

//根据类型名称筛选
var types = controllerTypes.Where(type => string.Compare(
    controllerName + "Controller", type.Name, true) == 0).ToArray();
if (types.Length == 0)
{
    return null;
}

//通过路由对象的命名空间进行匹配
var namespaces = routeData.DataTokens["Namespaces"] as
    IEnumerable<string>;
namespaces = namespaces ?? new string[0];
Type contrllrType = this.GetControllerType(namespaces, types);
if (null != contrllrType)
{
    return contrllrType;
}

//是否允许采用后备命名空间
bool useNamespaceFallback = true;
if (null != routeData.DataTokens["UseNamespaceFallback"])
{
    useNamespaceFallback =
        (bool)(routeData.DataTokens["UseNamespaceFallback"]);
}

//如果不允许采用后备命名空间, 则返回 Null
if (!useNamespaceFallback)
{
    return null;
}

//通过当前 ControllerBuilder 的默认命名空间进行匹配
contrllrType = this.GetControllerType(
    ControllerBuilder.Current.DefaultNamespaces, types);
if (null != contrllrType)
{
    return contrllrType;
}

//如果只存在一个类型名称匹配的 Controller, 则返回之
if (types.Length == 1)
{
    return types[0];
}

//如果具有多个类型名称匹配的 Controller, 则抛出异常
throw new InvalidOperationException("具有多个匹配的 Controller 类型");
}
}

```

在不能根据 `Route` 对象命名空间成功解析出目标 `Controller` 类型的情况下，为了确定是否采用后备命名空间对 `Controller` 类型作进一步的解析，我们会尝试从作为参数的 `RouteData` 对象的 `DataTokens` 属性中提取路由变量 `UseNamespaceFallback`。如果该路由变量存在并且值为 `False`，则直接返回 `Null`。

如果路由变量 `UseNamespaceFallback` 不存在，或者它的值为 `True`，我们会采用当前 `ControllerBuilder` 的默认命名空间列表对 `Controller` 类型作进一步的解析，如果存在唯一的类型则直接当作目标 `Controller` 类型。如果通过两组命名空间均不能得到一个匹配的 `Controller` 类型，但是候选的 `Controller` 类型中只存在一个与传入的 `Controller` 名称（不含命名空间）相匹配的类型，则直接将该类型作为目标 `Controller` 类型。如果这样的类型具有多个，则直接抛出 `InvalidOperationException` 异常。

现在我们在一个 ASP.NET MVC 应用中创建如下一个简单的 `HomeController`，其默认的动作方法 `Index` 直接返回一个简单的指示行字符串。在 `Global.asax` 文件中，我们将针对 `ReflectedControllerFactory` 的注册定义在 `Application_Start` 方法中。如下面的代码片段所示，我们通过当前 `ControllerBuilder` 注册的是一个 `ReflectedControllerFactory` 对象而非其类型。

```
public class HomeController: Controller
{
    public string Index()
    {
        return "这是一个通过 ReflelctionControllerFactory 激活的 Controller! ";
    }
}

public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        //其他操作
        ControllerBuilder.Current.SetControllerFactory(
            new ReflectedControllerFactory());
    }
}
```

直接运行该程序会在浏览器中得到如图 3-8 所示的输出结果，由此可见我们注册的 `ReflectedControllerFactory` 正常地完成了其激活目标 `Controller` 对象的使命。有兴趣的读者可以进一步通过实例测试 `ReflectedControllerFactory` 针对不同命名空间的 `Controller` 解析机制，在这里我们就不一一演示了。



图 3-8 利用注册的 ReflectedControllerFactory 激活 Controller

3.2.2 Controller 类型的缓存

对于上面演示实例中用于模拟默认 Controller 激活机制的 `ReflectedControllerFactory` 类型来说,出于性能的考虑,我们对解析出来的所有有效的 Controller 类型作了全局缓存。针对 Controller 类型的缓存同样实现在 `DefaultControllerFactory` 中,而且它采用的缓存更加彻底,因为它会对缓存的 Controller 类型列表作持久化(被持久化的 Controller 类型列表在 ASP.NET MVC 应用重新启动之后依然有效)。

在本书第 2 章“路由”中,我们谈到了针对 Area 注册的实现原理。具体来说,Area 的注册是通过相应的 `AreaRegistration` 对象来完成的,为了避免频繁地解析所有 `AreaRegistration` 类型,ASP.NET MVC 会对解析出来 `AreaRegistration` 类型列表实施缓存,并将其持久化到一个临时文件中。与之类似,Controller 激活系统针对解析出来的所有 Controller 类型也采用相同的缓存策略。用于存储 Controller 类型列表的名为 `MVCControllerTypeCache.xml` 的文件保存在 ASP.NET 的临时目录下面,具体的路径如下。

- `%Windir%\Microsoft.NET\Framework\v{version}\Temporary ASP.NET Files\{appname}\...\UserCache\`
- `%Windir%\Microsoft.NET\Framework\v{version}\Temporary ASP.NET Files\root\...\UserCache\`

第一个针对寄宿于 Local IIS 中的 Web 应用,后者针对直接采用 Visual Studio Developer Server 或者 IIS Express 作为宿主的应用。用于保存所有 `AreaRegistration` 类型列表的 `MVC-AreaRegistrationTypeCache.xml` 文件也保存在这个目录下面。

当接收到 Web 应用被启动后的第一个请求时,Controller 激活系统会读取这个保存了所有 Controller 类型列表的 `ControllerTypeCache.xml` 文件,并对读取内容实施反序列化生成一个 `List<Type>` 对象。只有在该列表为空的时候才会通过遍历程序集和反射的方式得到所有实现了接口 `IController` 的类型,而被解析出来的 Controller 类型会被重新写入这个缓存文件中。

下面的 XML 片段反映了这个用于保存 Controller 类型列表的 MVC-ControllerTypeCache.xml 文件的结构,从中可以看出它包含了所有 Controller 类型的全名和所在程序集名称。

```
<?xml version="1.0" encoding="utf-8"?>
<!--This file is automatically generated. Please do not modify the contents of
this file.-->
<typeCache lastModified="3/4/2014 9:05:09 AM"
    mvcVersionId="72d59038-e845-45b1-853a-70864614e003">
  <assembly name="Artech.Admin, Version=1.0.0.0, Culture=neutral,
    PublicKeyToken=null">
    <module versionId="eb343e3f-2d63-4665-a12a-29fb30dceeed">
      <type>Artech.Admin.HomeController</type>
      <type>Artech.Admin.EmployeesController </type>
    </module>
  </assembly>
  <assembly name="Artech.Portal, Version=1.0.0.0, Culture=neutral,
    PublicKeyToken=null">
    <module versionId="3717F116-35EE-425F-A1AE-EB4267497D8C">
      <type>Artech.Portal.Controllers.HomeController</type>
      <type>Artech.Portal.ProductsController</type>
    </module>
  </assembly>
</typeCache>
```

3.2.3 Controller 的释放和会话状态行为的控制

用于激活目标 Controller 对象的 ControllerFactory 不仅仅用于创建目标 Controller 对象,还具有两个额外的功能,一个是通过 ReleaseController 方法对激活的 Controller 对象进行释放和回收,另一个则是通过 GetControllerSessionBehavior 方法返回用于控制当前会话状态行为的 SessionStateBehavior 枚举对象。

对于默认使用的 DefaultControllerFactory 来说,它对 Controller 对象的释放操作很简单:如果 Controller 类型实现了 IDisposable 接口,则直接调用其 Dispose 方法即可。如下面的代码片段所示,我们将这个逻辑也实现在了自定义的 ReflectedControllerFactory 类型中。

```
public class ReflectedControllerFactory : IControllerFactory
{
    //其他操作
    public void ReleaseController(IController controller)
    {
        IDisposable disposable = controller as IDisposable;
        if (null != disposable)
        {
            disposable.Dispose();
        }
    }
}
```


至于用于返回 `SessionStateBehavior` 枚举的 `GetControllerSessionBehavior` 方法，在默认情况下它的返回值为 `SessionStateBehavior.Default`。通过前面的介绍我们知道，在这种情况下具体的会话状态行为取决于映射的 `HttpHandler` 所实现的标记接口。对于 ASP.NET MVC 应用来说，默认使用的 `HttpHandler` 是一个 `MvcHandler` 对象。如下面的代码片段所示，类型 `MvcHandler` 实现了 `IRequiresSessionState` 接口，意味着默认情况下会话状态是可读可写的（相当于 `SessionStateBehavior.Required`）。

```
public class MvcHandler :
    IHttpAsyncHandler,
    IHttpHandler,
    IRequiresSessionState
{
    //其他成员
}
```

我们可以通过在 `Controller` 类型上应用 `SessionStateAttribute` 特性来具体控制会话状态行为。如下面的代码片段所示，`SessionStateAttribute` 具有一个 `SessionStateBehavior` 类型的只读属性 `Behavior`，该属性是在构造函数中被初始化的。

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple = false,
    Inherited = true)]
public sealed class SessionStateAttribute : Attribute
{
    public SessionStateAttribute(SessionStateBehavior behavior);
    public SessionStateBehavior Behavior { get; }
}
```

在实现的 `GetControllerSessionBehavior` 方法中，`DefaultControllerFactory` 会试着获取应用在 `Controller` 类型上的 `SessionStateAttribute` 特性，如果这样的特性存在则直接返回它的 `Behavior` 属性所表示的 `SessionStateBehavior` 枚举，否则返回 `SessionStateBehavior.Default`。具体的逻辑也反映在我们自定义的 `ReflectedControllerFactory` 的 `GetControllerSessionBehavior` 方法中。

```
public class ReflectedControllerFactory : IControllerFactory
{
    //其他成员
    public SessionStateBehavior GetControllerSessionBehavior(
        RequestContext requestContext, string controllerName)
    {
        Type controllerType = this.GetControllerType(requestContext.RouteData,
            controllerName);
        if (null == controllerType)
        {
            return SessionStateBehavior.Default;
        }
        SessionStateAttribute attribute = controllerType
            .GetCustomAttributes(true).OfType<SessionStateAttribute>()
            .FirstOrDefault();
    }
}
```

```

        .FirstOrDefault();
        attribute = attribute ??
            new SessionStateAttribute(SessionStateBehavior.Default);
        return attribute.Behavior;
    }
}

```

3.3 IoC 的应用

控制反转 (Inversion of Control, IoC)，简单地讲就是应用本身不负责依赖对象的创建和维护，而将此任务交给一个外部容器，这样控制权就由应用转移到了这个外部 IoC 容器，控制权就实现了所谓的反转。比如在类型 A 中需要使用类型 B 的实例，而 B 实例的创建并不由 A 来负责，而是通过外部容器来创建。通过 IoC 的方式实现针对目标 Controller 的激活具有重要的意义。

3.3.1 从 Unity 来认识 IoC

IoC 往往和另一个名词“依赖注入 (Dependency Injection, DI)”联系在一起。所谓依赖注入，就是由外部容器在运行时动态地将依赖的对象注入到组件之中。Martin Fowler 在那篇著名的文章 *Inversion of Control Containers and the Dependency Injection pattern* 中将具体的依赖注入划分为 3 种形式，即构造器注入、属性（设置）注入和接口注入，而笔者个人习惯将其划分为一种（类型）映射和 3 种注入。

- 类型映射 (Type Mapping)。虽然我们可以通过接口（或者抽象类）来调用某个对象的方法，但是对象本身却属于某个具体的类型，所以需要某种类型注册机制来解决接口/抽象类和实现类/具体子类之间的匹配关系。
- 构造器注入 (Constructor Injection)。IoC 容器会智能地选择和调用适合的构造函数以创建依赖的对象。如果被选择的构造函数具有相应的参数，IoC 容器会在调用构造函数之前解析注册的依赖关系并自行创建相应的参数对象。
- 属性注入 (Property Injection)。如果需要使用到被依赖对象的某个属性，在被依赖对象被创建之后 IoC 容器会自动初始化该属性。
- 方法注入 (Method Injection)。如果被依赖对象需要调用某个方法进行相应的初始化，在该对象创建之后 IoC 容器会自动调用该方法。

开源社区有很多流行的 IoC 框架，如 Castle、Unity、Spring.NET、StructureMap 和 Ninject

等。Unity 是微软 Patterns & Practices 部门开发的一个轻量级的 IoC 框架，该项目在 Codeplex 上的地址为 <http://unity.codeplex.com/>，我们可以下载相应的安装包和开发文档。在本书出版之时，Unity 的最新版本为 3.0。出于篇幅的限制，我们不可能对 Unity 进行详细的讨论，但是为了让读者了解 IoC 在 Unity 中的实现，我们写了一个简单的程序。

我们创建一个控制台程序，并在其中定义如下 4 个接口（IA、IB、IC 和 ID）和它们各自的实现类（A、B、C、D）。我们在类型 A 中定义了 B、C 和 D 3 个属性，其声明类型分别为接口 IB、IC 和 ID。属性 B 在构造函数中被初始化，意味着它会以构造器注入的方式被初始化。属性 C 上应用了 `DependencyAttribute` 特性，意味着这是一个需要以属性注入方式被初始化的依赖属性。属性 D 则通过方法 `Initialize` 初始化，该方法上应用了特性 `InjectionMethodAttribute`，意味着这是一个注入方法，它会在 A 对象被 IoC 容器激活的时候被自动调用。

```
namespace UnityDemo
{
    public interface IA {}
    public interface IB {}
    public interface IC {}
    public interface ID {}

    public class A : IA
    {
        public IB B { get; set; }
        [Dependency]
        public IC C { get; set; }
        public ID D { get; set; }

        public A(IB b)
        {
            this.B = b;
        }
        [InjectionMethod]
        public void Initialize(ID d)
        {
            this.D = d;
        }
    }
    public class B: IB{}
    public class C: IC{}
    public class D: ID{}
}
```

我们为该应用添加一个配置文件并定义如下一段关于 Unity 的配置。这段配置定义了一个名称为 `defaultContainer` 的 Unity 容器，并在其中完成了上面定义的接口和对应实现类之间的映射。

```
<configuration>
  <configSections>
    <section name="unity"
```

```

        type="Microsoft.Practices.Unity.Configuration.UnityConfigurationSection,
        Microsoft.Practices.Unity.Configuration"/>
    </configSections>
    <unity>
        <containers>
            <container name="defaultContainer">
                <register type="UnityDemo.IA, UnityDemo" mapTo="UnityDemo.A, UnityDemo"/>
                <register type="UnityDemo.IB, UnityDemo" mapTo="UnityDemo.B, UnityDemo"/>
                <register type="UnityDemo.IC, UnityDemo" mapTo="UnityDemo.C, UnityDemo"/>
                <register type="UnityDemo.ID, UnityDemo" mapTo="UnityDemo.D, UnityDemo"/>
            </container>
        </containers>
    </unity>
</configuration>

```

我们在作为程序入口的 Main 方法中创建一个代表 IoC 容器的 UnityContainer 对象，并加载配置信息对其进行初始化。随后调用它的泛型方法 Resolve 创建一个实现了泛型接口 IA 的对象。在将返回对象转变成类型 A 之后，我们检验其 B、C 和 D 属性是否为 Null。

```

static void Main(string[] args)
{
    IUnityContainer container = new UnityContainer();
    UnityConfigurationSection configuration =
        ConfigurationManager.GetSection(UnityConfigurationSection.SectionName)
        as UnityConfigurationSection;
    configuration.Configure(container, "defaultContainer");
    A a = container.Resolve<IA>() as A;
    if (null != a)
    {
        Console.WriteLine("a.B == null ? {0}", a.B == null ? "Yes" : "No");
        Console.WriteLine("a.C == null ? {0}", a.C == null ? "Yes" : "No");
        Console.WriteLine("a.D == null ? {0}", a.D == null ? "Yes" : "No");
    }
}

```

从如下给出的执行结果可以得到这样的结论：通过 Resolve<IA>方法返回的是一个类型为 A 的对象，该对象的 3 个属性被进行了有效的初始化。这个简单的程序分别体现了类型映射（通过相应的接口根据配置解析出相应的实现类型）、构造器注入（属性 B）、属性注入（属性 C）和方法注入（属性 D）。(S309)

```

a.B == null ? No
a.C == null ? No
a.D == null ? No

```

3.3.2 Controller 与 Model 的解耦

在第 1 章“ASP.NET + MVC”中我们谈到过 ASP.NET MVC 是根据 MVC 模式的变体 Model 2

设计的。ASP.NET MVC 所谓的 Model 仅仅表示绑定到 View 上的数据，所以我们一般称之为 View Model。值得再次强调的是，这里的 View Model 与 MVVM 模式中的 View Model 不是同一个概念，后者不仅仅包含绑定的数据，其绑定逻辑也实现在 View Model 之中。MVC 模式中的 Model 一般指维护应用状态和提供业务功能操作的领域模型、针对业务层的入口或者业务服务的代理。MVC 模式在 ASP.NET MVC 中的应用体现在如图 3-9 所示的 UML 中。

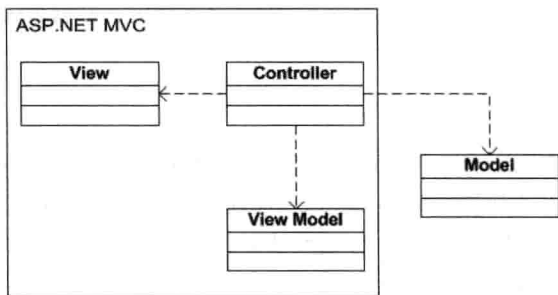


图 3-9 ASP.NET MVC + Model

对于一个 ASP.NET MVC 应用来说，用户交互请求直接发送给 Controller。如果涉及针对某项业务功能，Controller 会直接调用 Model。如果需要呈现业务数据，Controller 会通过 Model 获取相应业务数据并转换成 View Model 并通过 View 呈现出来。这样的交互协作方式反映了 Controller 针对 Model 的直接依赖。

如果我们在 Controller 激活系统中引入 IoC，并采用 IoC 的方式提供用于处理请求的 Controller 对象，那么 Controller 和 Model 之间的依赖程度可以在很大程度上被降低。我们甚至可以像图 3-10 所示的那样以接口的方式对 Model 进行抽象，让 Controller 依赖于这个抽象化的 Model 接口，而不是具体的 Model 实现。

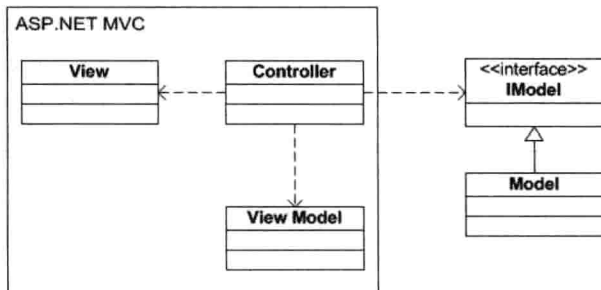


图 3-10 ASP.NET MVC + IModel + Model

3.3.3 基于 IoC 的 ControllerFactory

ASP.NET MVC 的 Controller 激活系统最终利用 ControllerFactory 来创建目标 Controller 对象，要将 IoC 引入 ASP.NET MVC 并通过对应的 IoC 容器实现对目标 Controller 的激活，我们很自然地会想到自定义一个基于 IoC 的 ControllerFactory。

我们可以直接实现 IControllerFactory 接口定义一个全新的 ControllerFactory 类型，但是这需要实现包括 Controller 类型解析、Controller 实例创建/释放及会话状态行为模式获取在内的所有功能。一般来说，Controller 实例的创建才需要 IoC 容器的控制¹，所以为了避免重新实现其他的功能，我们可以让自定义的 ControllerFactory 类型直接继承自 DefaultControllerFactory，这样的话只需重写 Controller 实例创建的逻辑即可。

实例演示：创建基于 Unity 的 ControllerFactory (S310)

现在我们通过一个简单的实例来演示如何通过自定义 ControllerFactory 利用 Unity 进行 Controller 的激活，所以我们将这个自定义 ControllerFactory 命名为 UnityControllerFactory。为了避免针对 Controller 类型解析、会话状态行为模式获取和对 Controller 对象的释放逻辑的重复实现，我们让 UnityControllerFactory 继承自 DefaultControllerFactory。

如下面的代码片段所示，UnityControllerFactory 具有一个只读的属性 UnityContainer 表示 Unity 框架下的 IoC 容器。我们仅仅重写了受保护的虚方法 GetControllerInstance，在该方法中我们将解析出来的目标 Controller 类型作为参数调用此 UnityContainer 对象的 Resolve 方法，进而得到最终被激活的 Controller 对象。

```
public class UnityControllerFactory: DefaultControllerFactory
{
    public IUnityContainer UnityContainer { get; private set; }

    public UnityControllerFactory(IUnityContainer unityContainer)
    {
        this.UnityContainer = unityContainer;
    }

    protected override IController GetControllerInstance(
        RequestContext requestContext, Type controllerType)
```

¹ 其实很多 IoC 框架不仅仅让 IoC 容器实现了目标对象的创建，还实现了针对目标对象的回收释放及管理对象生命周期的功能。比如我们可以让 IoC 容器获取某种类型对象的时候总是创建一个新的对象，也可以以单例模式让每次获取的均为同一对象。ASP.NET MVC 在默认情况下总是会创建一个全新的 Controller 对象来处理每一个请求。有人可能会问：“这样频繁地创建 Controller 对象是否会影响性能？可否采用对象池的方式来解决这个问题？”笔者个人觉得没有太大的必要，由于 Controller 的类型已经预先加载，所以通过类型反射创建 Controller 对象不会太耗时。此外，如果一个 Controller 被用于处理多个请求，这就要求我们必须创建“无状态”的 Controller，否则会引起并发问题。

```

    {
        if (null == controllerType)
        {
            return null;
        }
        return (IController)this.UnityContainer.Resolve(controllerType);
    }
}

```

整个自定义的 `UnityControllerFactory` 就这么简单。为了演示 IoC 在它身上的体现，我们在一个简单的 ASP.MVC 应用中来使用它。我们的演示实例沿用在第 2 章“路由”中使用过的关于“员工管理”的应用场景。如图 3-11 所示，本实例由两个页面（对应着两个 View）组成，一个用于显示员工列表，另一个用于显示基于某个员工的详细信息。



图 3-11 员工列表和员工详细信息页面

我们在一个 ASP.NET MVC 应用中添加对 Unity 的程序集 `Microsoft.Practices.Unity.dll` 的引用(如果读者不想安装 Unity, 可以通过下载本实例的源代码的方式获取该程序集), 然后在 `Models` 目录下定义如下一个表示员工信息的 `Employee` 类型。

```

public class Employee
{
    [Display(Name="ID")]
    public string Id { get; private set; }

    [Display(Name = "姓名")]
    public string Name { get; private set; }

    [Display(Name = "性别")]

```

```

public string Gender { get; private set; }

[Display(Name = "出生日期")]
[DataType(DataType.Date)]
public DateTime BirthDate { get; private set; }

[Display(Name = "部门")]
public string Department { get; private set; }

public Employee(string id, string name, string gender, DateTime birthDate,
    string department)
{
    this.Id = id;
    this.Name = name;
    this.Gender = gender;
    this.BirthDate = birthDate;
    this.Department = department;
}
}

```

接下来创建一个 `EmployeeRepository` 类型来模拟针对员工数据的存储，并为其定义了对应的接口 `IEmployeeRepository`。如下面的代码片段所示，`IEmployeeRepository` 接口仅仅具有一个返回 `Employee` 列表的方法 `GetEmployees` 用于获取指定 ID 的员工信息。如果指定的 ID 为空，则该方法会返回所有员工列表。`EmployeeRepository` 类型直接利用一个静态字段来存储员工列表。

```

public interface IEmployeeRepository
{
    IEnumerable<Employee> GetEmployees(string id = "");
}

public class EmployeeRepository : IEmployeeRepository
{
    private static IList<Employee> employees;

    static EmployeeRepository()
    {
        employees = new List<Employee>();
        employees.Add(new Employee("001", "张三", "男", new DateTime(1981, 8, 24),
            "销售部"));
        employees.Add(new Employee("002", "李四", "女", new DateTime(1982, 7, 10),
            "人事部"));
        employees.Add(new Employee("003", "王五", "男", new DateTime(1981, 9, 21),
            "人事部"));
    }

    public IEnumerable<Employee> GetEmployees(string id = "")
    {
        return employees.Where(e => e.Id == id || string.IsNullOrEmpty(id));
    }
}

```


我们创建了如下一个 `EmployeesController` 类型，它具有一个类型为 `IEmployeeRepository` 的属性 `Repository`，应用在上面的 `DependencyAttribute` 特性将它定义成一个“依赖属性”。当我们采用注册的 `UnityControllerFactory` 对象来激活 `EmployeesController` 对象的时候，作为 IoC 容器的 `UnityContainer` 对象会根据注册的类型映射解析出实现了 `IEmployeeRepository` 接口的类型，并创建相应的对象来初始化该属性。

```
public class EmployeesController : Controller
{
    [Dependency]
    public IEmployeeRepository Repository { get; set; }

    public ActionResult GetAllEmployees()
    {
        var employees = this.Repository.GetEmployees();
        return View("EmployeeList", employees);
    }

    public ActionResult GetEmployeeById(string id)
    {
        Employee employee = this.Repository.GetEmployees(id).FirstOrDefault();
        if (null == employee)
        {
            throw new HttpException(404, string.Format("ID为{0}的员工不存在", id));
        }
        return View("Employee", employee);
    }
}
```

`EmployeesController` 定义了两个基本的 Action 方法。`GetAllEmployees` 方法通过 `Repository` 获取所有员工列表并将其呈现在一个名为“EmployeeList”的 View 中。另一个 Action 方法 `GetEmployeeById` 则根据指定的 ID 获取相应的员工信息，最终用于呈现单个员工信息的 View 为“Employee”。如果根据指定的 ID 找不到相应的员工，则该方法会直接抛出一个状态为“404”的 `HttpException` 异常。

如下所示的是用于显示员工列表的 View (`EmployeeList`) 的定义，它是一个 Model 类型为 `IEnumerable<Employee>` 的强类型 View，我们在该 View 中通过一个表格来显示员工列表。值得一提的是，我们通过调用 `HtmlHelper` 的 `ActionLink` 方法将员工的名称显示为一个指向 Action 方法 `GetEmployeeById` 的链接。

```
@model IEnumerable<Employee>
<html>
    <head>
        <title>员工列表</title>
    </head>
    <body>
```

```

<table>
  <tr>
    <th>姓名</th>
    <th>性别</th>
    <th>出生日期</th>
    <th>部门</th>
  </tr>
  @{
    foreach(Employee employee in Model)
    {
      <tr>
        <td>
          @Html.ActionLink(employee.Name, "GetEmployeeById",
            new { name = employee.Name, id = employee.Id })
        </td>
        <td>@Html.DisplayFor(m=>employee.Gender)</td>
        <td>@Html.DisplayFor(m=>employee.BirthDate)</td>
        <td>@Html.DisplayFor(m=>employee.Department)</td>
      </tr>
    }
  </table>
</body>
</html>

```

用于显示单个员工信息的名为 `Employee` 的 `View` 定义如下，这是一个 `Model` 类型为 `Employee` 的强类型 `View`，我们在该 `View` 中依然通过表格的形式将员工的详细信息显示出来。

```

@model Employee
<html>
  <head>
    <title>@Model.Name</title>
  </head>
  <body>
    <table>
      <tr>
        <td>
          @Html.LabelFor(m=>m.Id)</td><td>@Html.DisplayFor(m=>m.Id)
        </td>
      </tr>
      <tr>
        <td>
          @Html.LabelFor(m=>m.Name)</td><td>@Html.DisplayFor(
            m=>m.Name)
          </td>
        </tr>
      <tr>
        <td>
          @Html.LabelFor(m=>m.Gender)</td><td>@Html.DisplayFor(
            m=>m.Gender)

```