



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>

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 zlt@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

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（比如我们会经常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容易测试，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. Reenskaug，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。

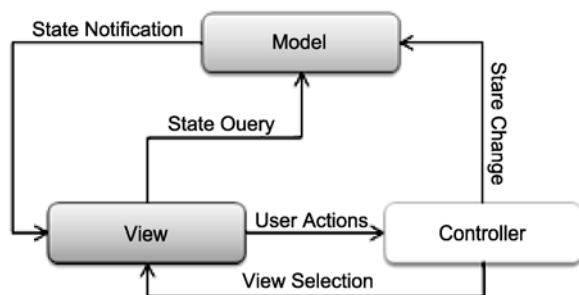


图 1-1 Model-View-Controller 之间的交互

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。

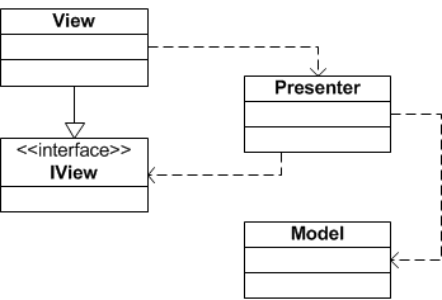


图 1-2 Model-View-Presenter 之间的交互

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。

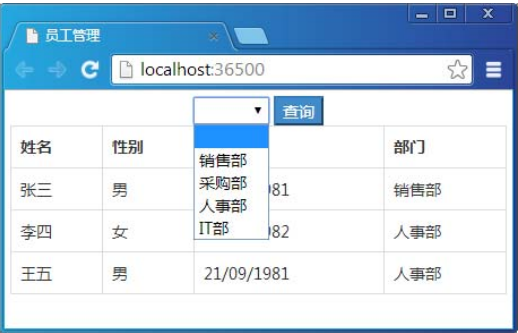


图 1-3 员工查询页面

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，现在我们来看看 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

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。

(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;

    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。

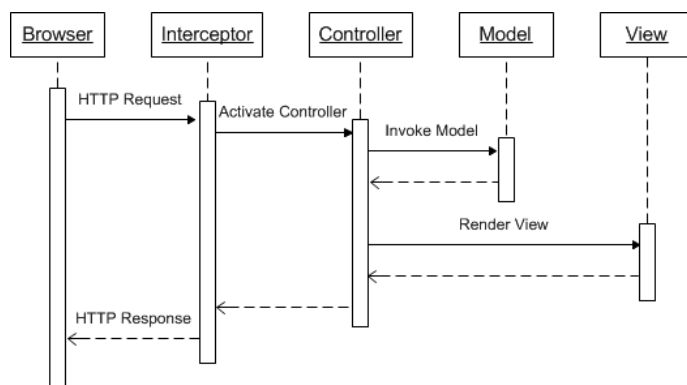


图 1-4 Model 2 交互流程

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。

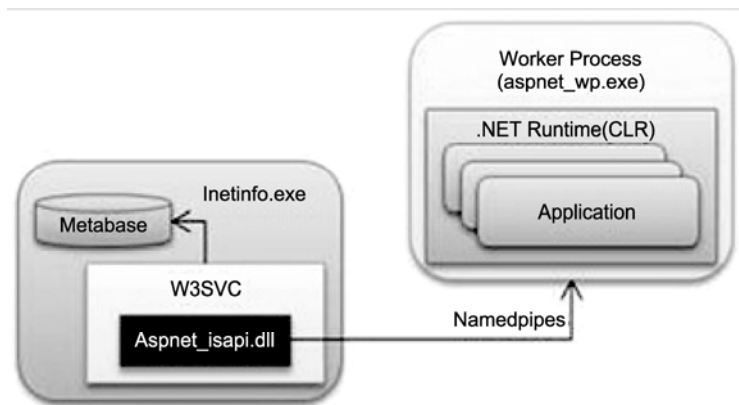


图 1-5 IIS 5.x 与 ASP.NET

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。

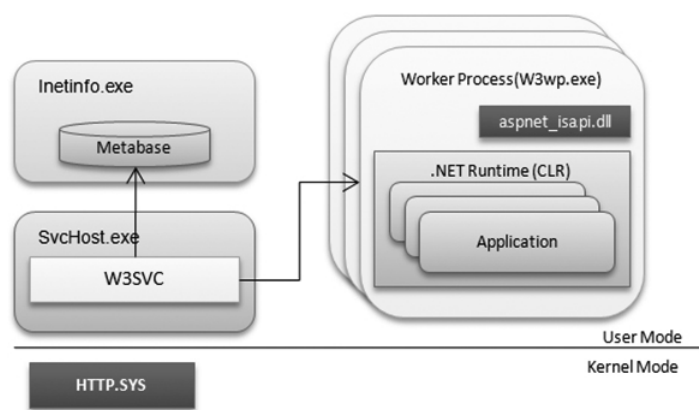


图 1-6 IIS 6.0 与 ASP.NET

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。

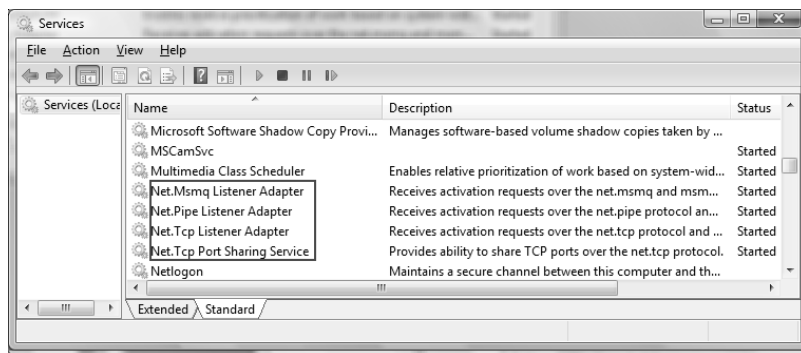


图 1-7 定义在 SMSVCHost.exe 中的 Windows Service

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的请求，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。

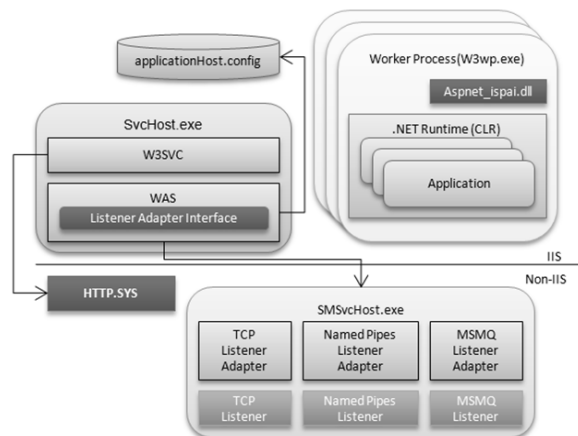


图 1-8 IIS 7.0 与 ASP.NET

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。

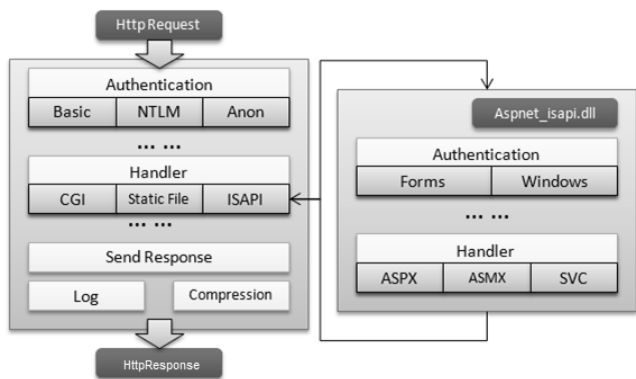


图 1-9 基于 IIS 6.0 与 ASP.NET 双管道设计

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。

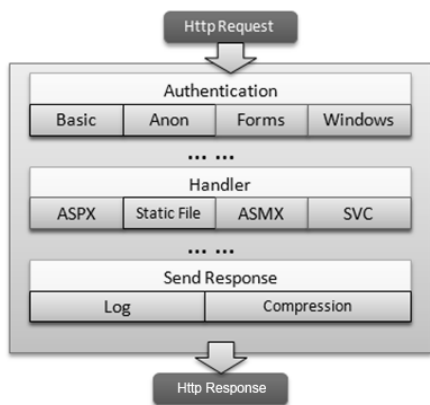


图 1-10 基于 IIS 7.0 与 ASP.NET 集成管道设计

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。

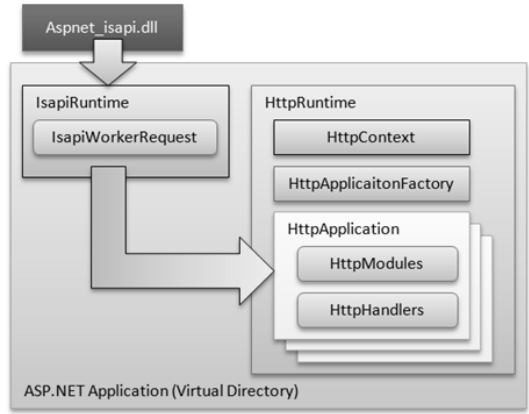


图 1-11 ASP.NET 处理管道

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的请求处理，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
<code>BeginRequest</code>	HTTP 管道开始处理请求时，会触发 <code>BeginRequest</code> 事件
<code>AuthenticateRequest</code> , <code>PostAuthenticateRequest</code>	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
<code>AuthorizeRequest</code> , <code>PostAuthorizeRequest</code>	ASP.NET 先后触发这两个事件，使安全模块对请求进行授权
<code>ResolveRequestCache</code> , <code>PostResolveRequestCache</code>	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）
<code>PostMapRequestHandler</code>	对于访问不同的资源类型，ASP.NET 具有不同的 <code>HttpHandler</code> 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 <code>HttpHandler</code> 类型，成功匹配后，该事件被触发
<code>AcquireRequestState</code> , <code>PostAcquireRequestState</code>	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 <code>SessionState</code>
<code>PreRequestHandlerExecute</code> , <code>PostRequestHandlerExecute</code>	ASP.NET 最终通过与请求资源类型相对应的 <code>HttpHandler</code> 实现对请求的处理，在实行 <code>HttpHandler</code> 前后，这两个事件被先后触发

续表

名 称	描 述
<code>ReleaseRequestState</code> ,	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

PostReleaseRequestState	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 HttpHandlerFactory 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 HttpHandler/HttpHandlerFactory 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 HttpHandler 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 HttpHandler 对象映射到当前 HTTP 请求。如果不曾通过调用该方法进行 HttpHandler 的显式映射，或者调用该方法时传入的参数为 Null，真正的 HttpHandler 对象的映射发生在 HttpApplication 的 PostMapRequestHandler 触发之前，默认进行 HttpHandler 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）

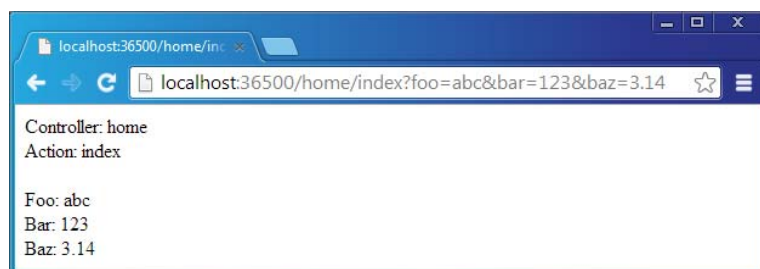


图 1-12 采用符合注册的路由地址模板的地址访问 Web 应用

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
{
```

ASP.NET MVC 5 框架揭秘

```

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 GetHandler 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。

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。

```

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

    public Route()
    {
        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 `HttpHandler` 对象。`UrlRoutingModule` 最后调用 `HttpContextWrapper` 对象的 `RemapHandler` 方法对得到的 `HttpHandler` 对象进行映射，那么针对当前 HTTP 请求的后续处理将由这个 `HttpHandler` 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 `GetHandler` 方法中它会直接返回一个 `MvcHandler` 对象。

```
public class MvcRouteHandler: IRouteHandler
{
    public IHttpHandler GetHandler(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 `IHttpHandler` 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 `IHttpHandler` 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 `GetHandler` 方法得到一个 `IHttpHandler` 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(HomeController).IsAssignableFrom(type)))
            {
                controllerTypes.Add(type);
            }
        }
    }

    public HomeController 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 (HomeController)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。

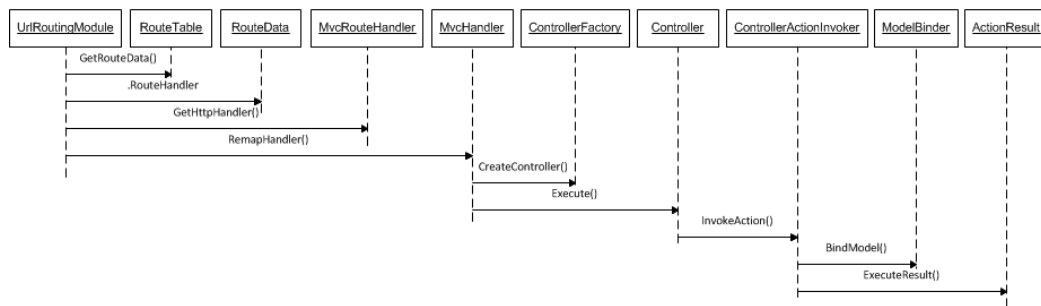


图 1-13 “接收请求”到“响应回复”的完整流程

由于 UriRoutingModule 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，UriRoutingModule 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，UriRoutingModule 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 `GetHandler` 方法得到一个 `Handler` 对象。对于 `MvcRouteHandler` 来说，它的 `GetHandler` 方法具体返回的是一个 `MvcHandler` 对象。`UrlRoutingModule` 随之调用当前 `HttpContext` 上下文的 `MapHandler` 方法对得到的 `Handler` 对象实施映射，那么此 `Handler` 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进行模式匹配。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" />
            <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。如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：前者是通过对请求 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 IHttpHandler 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 returnUrl,
    string physicalFile);
public Route MapPageRoute(string routeName, string returnUrl,
    string physicalFile, bool checkPhysicalUrlAccess);
public Route MapPageRoute(string routeName, string returnUrl,
    string physicalFile, bool checkPhysicalUrlAccess,
    RouteValueDictionary defaults);
public Route MapPageRoute(string routeName, string returnUrl,
    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` 属性用于控制是否对存在的物理文件实施路由，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针对物理文件的请求。

`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 的正常化（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 对象的数据Tokens 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 表示是否需要对表示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。

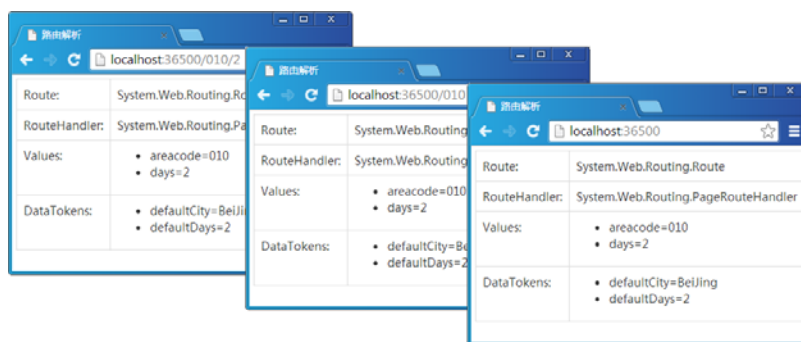


图 2-2 基于变量默认值的 URL 等效性

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 对象的数据 Tokens 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）

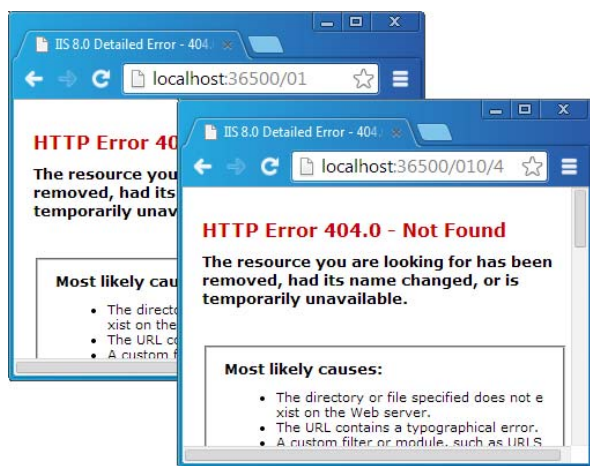


图 2-3 不满足正则表达式约束导致的 404 错误

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。该方法的 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）

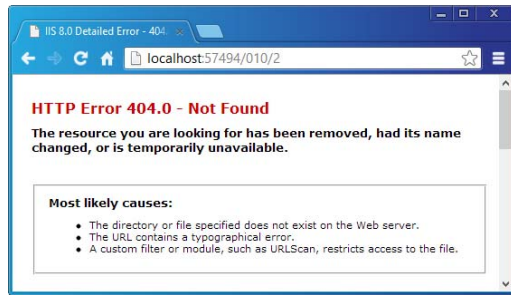


图 2-4 不满足 HTTP 方法约束（POST）导致的 404 错误

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）

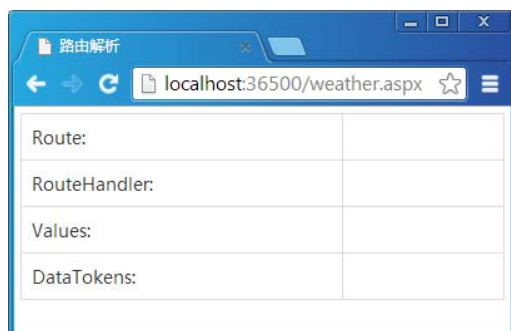


图 2-5 直接请求现存的物理文件（RouteExistingFiles = false）

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)

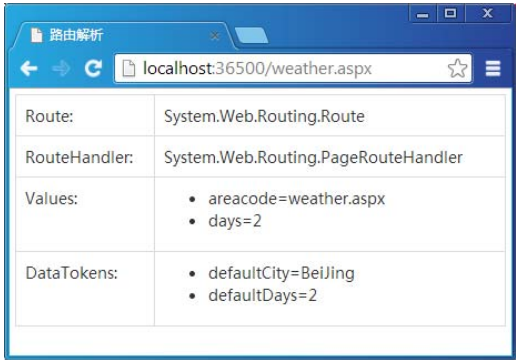


图 2-6 直接请求现存的物理文件（RouteExistingFiles = true）

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）

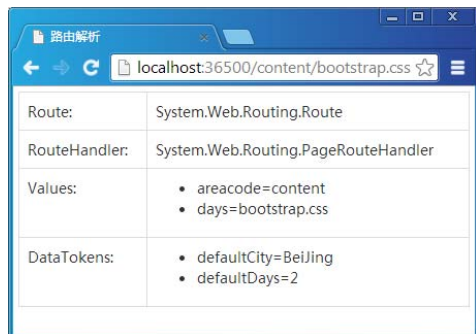


图 2-8 直接请求现存的.css 文件（RouteExistingFiles = true）

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)

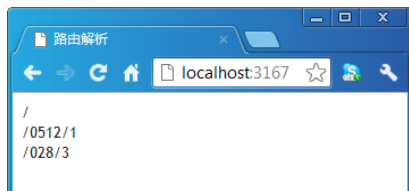


图 2-9 `GetVirtualPath` 方法接受不同参数生成的 URL

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>
```

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。

```

<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>

```

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。

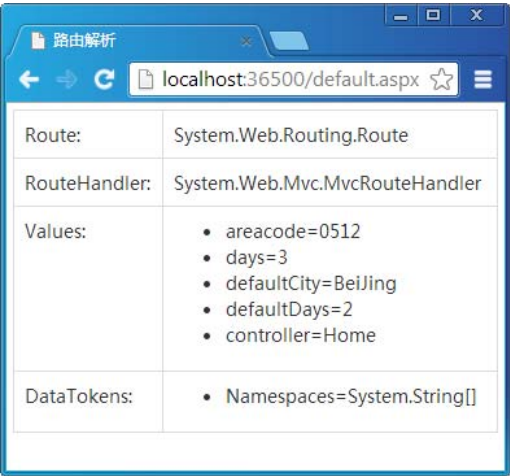


图 2-10 采用 ASP.NET MVC 路由映射得到的 RouteData

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, 它会为我们注册如下一个模板为 “{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 }
        );
    }
}
```

ASP.NET MVC 5 框架揭秘

}

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来。

```
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));
        }
    }
}
```

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。

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）

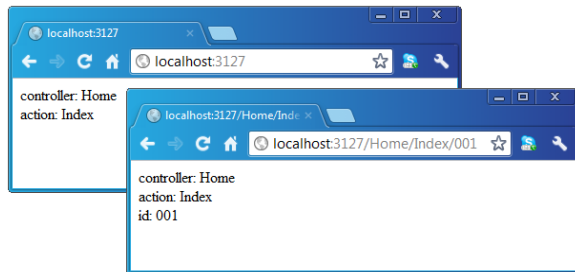


图 2-11 普通路由变量与缺省 URL 参数的路由变量之间的差别

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。

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。

- %Windir%\Microsoft.NET\Framework\v{version}\Temporary ASP.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。

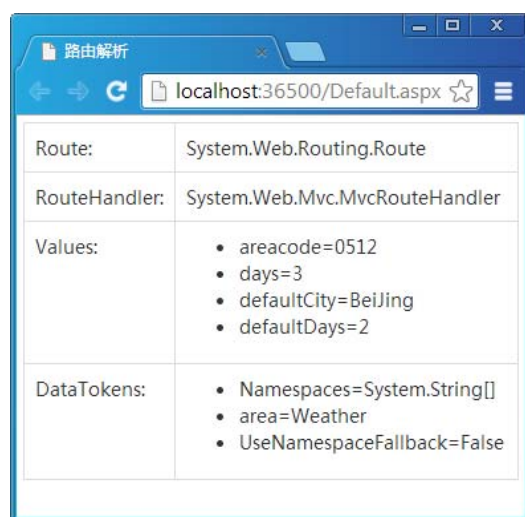
```
public partial class Default : System.Web.UI.Page
{
```

```

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，其余参数均是可以默认的。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。

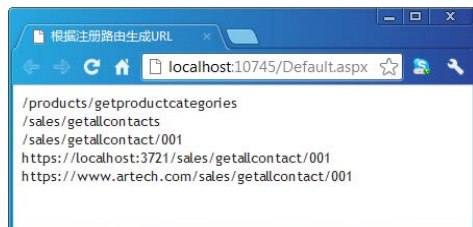


图 2-13 通过自定义 `RouteHelper` 生成的 URL

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。

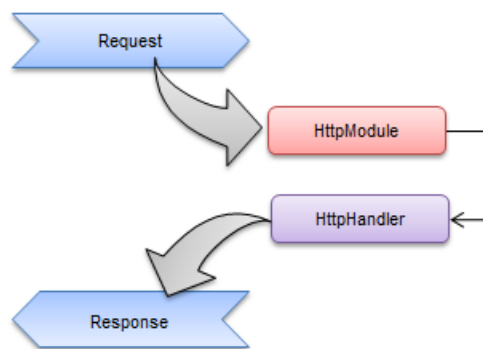


图 2-14 通过 HttpModule 实现针对 HttpHandler 的动态映射

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 `GetHandler` 方法得到这个需要被动态映射的 `Handler` 对象。定义在 `UrlRoutingModule` 类型的 `OnApplicationPostResolveRequestCache` 方法中的 `Handler` 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.GetHandler(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 `GetHandler` 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 `Handler` 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 : IHttpHandler
{
    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通过对 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);
    }
}
```

ASP.NET MVC 5 框架揭秘

```

        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。

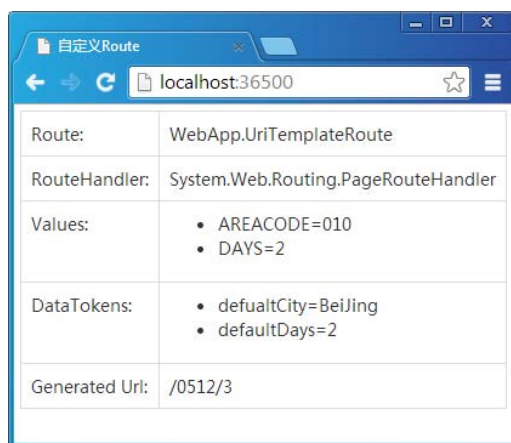
```
<form id="form1" runat="server">
    <div>
        <table>
            <tr>
                <td>Route:</td>
                <td><%=RouteData.Route != null?
                    RouteData.Route.GetType().FullName:"" %></td>
            </tr>
            <tr>
                <td>RouteHandler:</td>
                <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)
        {%>
            <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 Url:	/0512/3

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

第 8 章 Model 的验证 (上篇)

ASP.NET MVC 采用 Model 绑定为目标 Action 生成相应的参数列表，但是在真正执行目标 Action 方法之前，还需要对绑定的参数实施验证以确保其有效性，我们将针对参数的验证称为 Model 验证。ASP.NET MVC 的 Model 验证不仅直接帮助我们实现了服务端验证，还为客户端验证生成相应的验证规则。

8.1 几种参数验证方式

ASP.NET MVC 的 Model 绑定系统能够从请求中提取相应的数据并为目标 Action 方法生成相应的参数对象，在目标 Action 方法执行之前针对参数的验证是一项十分必要的操作。除了采用预定义的规则对 Model 绑定生成的参数实施验证之外，ASP.NET MVC 下的 Model 验证还不得不考虑一个问题，那就是如何将验证结果保存下来并呈现在 View 中。

8.1.1 ModelError

通过前面对“Model 绑定”的介绍我们知道，ASP.NET MVC 在针对目标 Action 方法的某个参数实施绑定过程中会将由 ValueProvider 提供的数据（通过一个 ValueProviderResult 对象封装）保存在当前 Controller 的 ModelState 中。具体来说，Controller 的 ModelState 是其 ViewData 的一部分，它返回一个 ModelStateDictionary 对象，这是一个 Key 和 Value 类型分别为 String 和 ModelState 的字典对象。这里的 Key 实际上就是在 Model 绑定执行过程中传递给 ValueProvider 用于提取相应原始数据的那个 Key。

ModelState 对象维护的 Model 状态具有两种形式：一种是上面所说的通过 ValueProvider 提供的 ValueProviderResult 对象，对应着属性 Value；另一种是通过 ModelErrorCollection 类型表示的错误信息。ModelErrorCollection 是一个元素类型为 ModelError 的集合，而 ModelError 通过属性 ErrorMessage 和 Exception 表述错误的消息和抛出的异常。如下面的代码片段所示，所有的这些类型都是可序列化的。

```
[Serializable]
public class ModelState
{
    public ModelErrorCollection    Errors { get; }
    public ValueProviderResult      Value { get; set; }
}

[Serializable]
public class ModelErrorCollection : Collection<ModelError>
{
    public void Add(Exception exception);
    public void Add(string errorMessage);
}

[Serializable]
public class ModelError
{
}
```



```

public ModelError(Exception exception);
public ModelError(string errorMessage);
public ModelError(Exception exception, string errorMessage);

public string ErrorMessage { get; }
public Exception Exception { get; }
}

```

如下面的代码片段所示, `ModelStateDictionary` 具有两个返回布尔值的方法。`IsValidField` 方法用于判断指定 `Key` 对应的 `ModelState` 是否有效, 具体采用的有效性判断逻辑很简单: 如果 `ModelState` 的 `Errors` 属性中包含一个或者多个 `ModelError` 则视为无效, 如果该属性对应的集合为空则视为有效。至于 `IsValid` 方法, 它帮助我们判断是否包含的所有 `ModelState` 均有效, 我们经常会调用此方法判断请求提交的数据是否均通过验证。

```

public class ModelStateDictionary : IDictionary<string, ModelState>,
{
    //其他成员
    public bool IsValidField(string key);
    public bool IsValid { get; }}
}

```

通过 `ModelError` 对象封装的错误可以是对绑定的参数实施验证得到的错误信息, 也可以是在执行 `Action` 方法过程中抛出的异常。如上面的代码片段所示, 我们可以分别通过提供的错误消息和异常对象来创建一个 `ModelError` 对象, `ModelErrorCollection` 类型提供两个 `Add` 方法重载, 它会根据提供的错误消息和异常对象来构建一个 `ModelError` 对象并添加到集合之中。

如下面的代码片段所示, `ModelStateDictionary` 为我们定义了两个 `AddModelError` 方法。当这两个方法被执行的时候, 它会先根据提供的 `Key` 去提取对应的 `ModelState` 对象, 并将创建的 `ModelError` 对象添加到它的 `Errors` 属性之中。如果这样的 `ModelState` 不存在, 该方法会根据这个 `Key` 创建新的 `ModelState` 对象, 创建的 `ModelError` 对象会被添加到它的 `Errors` 属性中。

```

public class ModelStateDictionary : IDictionary<string, ModelState>
{
    //其他成员
    public void AddModelError(string key, Exception exception);
    public void AddModelError(string key, string errorMessage);
}

```

8.1.2 验证消息的呈现

在很多情况下, 我们以表单的方式向服务端提交数据, `ASP.NET MVC` 会提取表单元素的值并采用 `Model` 绑定生成目标 `Action` 方法对应的参数。如果针对参数的验证失败, 相同的界面一般会再次呈现出来, 用户输入的数据不仅会保留下来, 相应的验证错误消息也会呈现出来。

那么当我们在进行 View 呈现的时候如何显示验证错误消息呢？

绑定的数据和验证错误均保存在当前 Controller 的 ModelState 中，后者是 ViewData 的一部分。所谓的 ViewData，实际上就是 Controller 传递给 View 的数据，所以我们在 View 中可以直接从当前 ModelState 中获取相应的验证错误。一旦得到了某个数据项对应的验证错误消息，我们就可以为它生成相应的 HTML 并最终将其呈现在 View 中。不过我们通常会调用帮助类 HtmlHelper 和 HtmlHelper<TModel>定义的扩展方法 ValidationMessage 和 ValidationMessageFor 来实现针对验证错误的呈现。

1 . ValidationMessage/ValidationMessageFor

验证消息在 View 中的呈现可以借助 HtmlHelper/HtmlHelper<TModel>来实现。如下面的代码所示，HtmlHelper 和 HtmlHelper<TModel> 分别提供了若干 ValidationMessage 和 ValidationMessageFor 扩展方法重载。

```
public static class ValidationExtensions
{
    public static MvcHtmlString ValidationMessage(this HtmlHelper htmlHelper,
        string modelName);
    public static MvcHtmlString ValidationMessage(this HtmlHelper htmlHelper,
        string modelName, IDictionary<string, object> htmlAttributes);
    //其他 ValidationMessage 方法重载

    public static MvcHtmlString ValidationMessageFor<TModel, TProperty>(
        this HtmlHelper<TModel> htmlHelper,
        Expression<Func<TModel, TProperty>> expression);
    public static MvcHtmlString ValidationMessageFor<TModel, TProperty>(
        this HtmlHelper<TModel> htmlHelper,
        Expression<Func<TModel, TProperty>> expression,
        string validationMessage);
    //其他 ValidationMessageFor 方法重载
}
```

HtmlHelper 的扩展方法 ValidationMessage 的参数 modelName 表示对应的 ModelState 在 ModelStateDictionary 中的 Key。如果针对这个 Key 找不到对应的 ModelState，或者对应的 ModelState 的 Errors 列表为空，意味着对应的数据成功通过验证，此时不会有任何 HTML 生成，否则该方法会生成一个元素来显示验证消息。

如果通过 validationMessage 显式指定了验证消息，那么该消息将会直接作为该元素的内部文本，否则 Errors 列表中第一个非空消息将会作为验证消息。除此之外，当我们调用扩展方法 ValidationMessage 时还可以通过参数 htmlAttributes 为这个元素设置相应的 HTML 属性。ValidationMessageFor 与 ValidationMessage 的不同之处在于它会通过指定的表达式来提取

ValidationMessage 方法中的参数 modelName。

2 . ValidationSummary

除了通过 ValidationMessageFor 与 ValidationMessage 这两个方法显示单条验证消息之外，我们还可以调用 HtmlHelper 的扩展方法 ValidationSummary 将所有的验证消息一并显示出来。如下面的代码片段所示，HtmlHelper 具有一系列 ValidationSummary 扩展方法重载，布尔类型的参数 excludePropertyErrors 表示是否需要排除基于属性的错误消息，而通过 message 参数可以为 ValidationSummary 指定一个作为标题的字符串。

```
public static class ValidationExtensions
{
    //其他成员
    public static MvcHtmlString ValidationSummary(
        this HtmlHelper htmlHelper);
    public static MvcHtmlString ValidationSummary(this HtmlHelper htmlHelper,
        bool excludePropertyErrors);
    public static MvcHtmlString ValidationSummary(this HtmlHelper htmlHelper,
        string message);
    //其他 ValidationSummary 方法重载
}
```

除了调用 ValidationMessage/ValidationMessageFor 和 ValidationSummary 显式控制错误消息在 View 中的呈现之外，当我们在一个强类型 View 中调用 HtmlHelper<TModel>的扩展方法 EditorForModel 将整个作为 Model 的数据对象以编辑模式呈现出来时，如果某个属性对应的 ModelState 具有相应的错误（通过 Errors 属性表示的 ModelError 集合不为空），错误消息也会一并呈现出来。当然，如果我们为 Model 类型定义了相应的模板就另当别论了。

8.1.3 手工验证绑定的参数

在定义具体 Action 方法的时候，对已经成功绑定的参数实施手工验证无疑是一种最为直接的编程方式，接下来我们通过一个简单的实例来演示如何将参数验证逻辑实现在对应的 Action 方法中，并在没有通过验证的情况下将错误信息响应给客户端。我们在一个 ASP.NET MVC 应用中定义了如下一个 Person 类作为被验证的数据类型，它的 Name、Gender 和 Age 属性分别表示一个人的姓名、性别和年龄。

```
public class Person
{
    [DisplayName("姓名")]
```

```

public string Name { get; set; }

[DisplayName("性别")]
public string Gender { get; set; }

[DisplayName("年龄")]
public int? Age { get; set; }
}

```

接下来我们定义了如下一个 `HomeController`。在针对 GET 请求的 Action 方法 `Index` 中，我们创建了一个 `Person` 对象并将其作为 `Model` 呈现在对应的 `View` 中。另一个支持 POST 请求的 `Index` 方法具有一个 `Person` 类型的参数，我们在该 Action 方法中先调用 `Validate` 方法对这个输入参数实施验证。如果验证成功（`ModelState.IsValid` 属性返回 `True`），则会返回一个内容为“输入数据通过验证”的 `ContentResult`，否则会将此参数作为 `Model` 呈现在对应的 `View` 中。

```

public class HomeController : Controller
{
    [HttpGet]
    public ActionResult Index()
    {
        return View(new Person());
    }

    [HttpPost]
    public ActionResult Index(Person person)
    {
        Validate(person);

        if (!ModelState.IsValid)
        {
            return View(person);
        }
        else
        {
            return Content("输入数据通过验证");
        }
    }

    private void Validate(Person person)
    {
        if (string.IsNullOrEmpty(person.Name))
        {
            ModelState.AddModelError("Name", "'Name' 是必需字段");
        }

        if (string.IsNullOrEmpty(person.Gender))
        {
            ModelState.AddModelError("Gender", "'Gender' 是必需字段");
        }
    }
}

```

```

    }
    else if (!new string[] { "M", "F", "m", "f" }.Any(
        g => string.Compare(person.Gender, g, true) == 0))
    {
        ModelState.AddModelError("Gender",
            "有效'Gender'必须是'M','F'之一");
    }

    if (null == person.Age)
    {
        ModelState.AddModelError("Age", "'Age'是必需字段");
    }
    else if (person.Age > 25 || person.Age < 18)
    {
        ModelState.AddModelError("Age", "有效'Age'必须在 18 到 25 周岁之间");
    }
}
}

```

如上面的代码片段所示，我们在 `Validate` 方法中对作为参数的 `Person` 对象的 3 个属性进行逐条验证，如果提供的数据没有通过验证，我们会调用当前 `ModelState` 的 `AddModelError` 方法将指定的验证错误消息转换为 `ModelError` 保存起来。我们采用的具体验证规则如下。

- `Person` 对象的 `Name`、`Gender` 和 `Age` 属性均为必需字段，不能为 `Null`（或者空字符串）。
- 表示性别的 `Gender` 属性的值必须是“M”（Male）或者“F”（Female），其余的均为无效值。
- `Age` 属性表示的年龄必须在 18 到 25 周岁之间。

如下所示的是 `Action` 方法 `Index` 对应 `View` 的定义，这是一个 `Model` 类型为 `Person` 的强类型 `View`，它包含一个用于编辑人员信息的表单。我们调用 `HtmlHelper<TModel>` 的扩展方法 `ValidationSummary` 对验证错误消息作统一显示。

```

@model Person
<html>
    <head>
        <title>编辑人员信息</title>
    </head>
    <body>
        @Html.ValidationSummary()
        @using (Html.BeginForm())
        {
            <div>@Html.LabelFor(m=>m.Name)</div>
            <div>@Html.EditorFor(m=>m.Name)</div>

            <div>@Html.LabelFor(m=>m.Gender)</div>
            <div>@Html.EditorFor(m => m.Gender)</div>
        }
    }
</body>
</html>

```

```

<div>@Html.LabelFor(m=>m.Age)</div>
<div>@Html.EditorFor(m => m.Age)</div>

<input type="submit" value="保存"/>
    }
</body>
</html>

```

直接运行该程序后，一个用于编辑人员基本信息的页面会被呈现出来，如果我们在输入不合法数据的情况下提交表单，相应的验证信息会以如图 8-1 所示的形式呈现出来。（S801）



图 8-1 以 ValidationSummary 形式呈现的验证错误信息

上面我们通过调用 `HtmlHelper<TModel>` 的扩展方法 `ValidationSummary` 将所有的验证错误消息统一显示在当前界面的某个地方，其实在很多情况下我们会选择将消息紧贴被验证输入元素显示，为此我们对 View 作了如下的改动。

```

@using (Html.BeginForm())
{
    <div>@Html.LabelFor(m=>m.Name)</div>
    <div>
        @Html.EditorFor(m=>m.Name)
        @Html.ValidationMessage("Name")
    </div>

    <div>@Html.LabelFor(m=>m.Gender)</div>

```

```

<div>
    @Html.EditorFor(m => m.Gender)
    @Html.ValidationMessage("Gender")
</div>

<div>@Html.LabelFor(m=>m.Age)</div>
<div>
    @Html.EditorFor(m => m.Age)
    @Html.ValidationMessage("Age")
</div>

<input type="submit" value="保存"/>
}

```

或者

```

@using (Html.BeginForm())
{
    <div>@Html.LabelFor(m=>m.Name)</div>
    <div>
        @Html.EditorFor(m=>m.Name)
        @Html.ValidationMessageFor(m=>m.Name)
    </div>

    <div>@Html.LabelFor(m=>m.Gender)</div>
    <div>
        @Html.EditorFor(m => m.Gender)
        @Html.ValidationMessageFor(m => m.Gender)
    </div>

    <div>@Html.LabelFor(m=>m.Age)</div>
    <div>
        @Html.EditorFor(m => m.Age)
        @Html.ValidationMessageFor(m => m.Age)
    </div>

    <input type="submit" value="保存"/>
}

```

如上面的代码片段所示，我们分别调用 `HtmlHelper` 的扩展方法 `ValidationMessage` 和 `HtmlHelper<TModel>` 的扩展方法 `ValidationMessageFor` 将验证错误信息显示在对应表单元素的右侧。如果运行该程序并在输入不合法数据的情况下提交表单，则会在浏览器中得到如图 8-2 所示的输出结果。(S802、S803)

图 8-2 以内联形式呈现的验证错误信息

我们还可以对 View 作相应的改动使之变得更加简单。如下面的代码片段所示，我们直接调用 `HtmlHelper<TModel>` 的扩展方法 `EditorForModel` 将作为 Model 的 `Person` 对象以编辑模式呈现在表单之中。运行此程序并在输入不合法数据的情况下提交表单，我们依然可以得到如图 8-2 所示的输出结果。（S804）

```
@using (Html.BeginForm())
{
    @Html.EditorForModel()
    <input type="submit" value="保存" />
}
```

8.1.4 使用 ValidationAttribute 特性

将针对输入参数的验证逻辑和业务逻辑定义在 `Action` 方法中并不是一种值得推荐的编程方式。在大部分情况下，同一个数据类型在不同的应用场景中具有相同的验证规则，如果我们能将验证规则与数据类型关联在一起，让框架本身来实施数据验证，那么最终的开发者就可以将关注点更多地放在业务逻辑的实现上面。

实际上这也是 ASP.NET MVC 的 Model 验证系统默认支持的编程方式。当我们在定义数据类型的时候，可以在类型及其数据成员上面应用相应的 `ValidationAttribute` 特性来定义默认采用的验证规则。“`System.ComponentModel.DataAnnotations`”命名空间定义了一系列具体的 `ValidationAttribute` 特性类型，它们大都可以直接应用在自定义数据类型的某个属性上对目标数据成员实施验证。这些预定义验证特性不是本章论述的重点，我们会在“下篇”中对它们作一个概括性的介绍。

1. 自定义 ValidationAttribute

常规验证可以通过预定义 `ValidationAttribute` 特性来完成,但是在很多情况下我们需要通过创建自定义的 `ValidationAttribute` 特性来解决一些特殊的验证。比如上面演示实例中针对 `Person` 对象的验证中,我们要求 `Gender` 属性指定的表示性别的值必须是“M/m”和“F/f”两者之一,这样的验证就不得不通过自定义的 `ValidationAttribute` 特性来实现。

针对“某个值必须在指定的范围内”这样的验证规则,我们定义了一个 `DomainAttribute` 特性。如下面的代码片段所示, `DomainAttribute` 为一个 `IEnumerable<string>` 类型的只读属性 `Values` 提供了一个有效值列表,该列表在构造函数中被初始化。具体的验证实现在重写的 `IsValid` 方法中,如果被验证的值在这个列表中,则视为验证成功并返回 `True`。为了提供一个友好的错误消息,我们重写了方法 `FormatErrorMessage`。

```
[AttributeUsage(AttributeTargets.Property | AttributeTargets.Field,
    AllowMultiple = false)]
public class DomainAttribute : ValidationAttribute
{
    public IEnumerable<string> Values { get; private set; }

    public DomainAttribute(string value)
    {
        this.Values = new string[] { value };
    }

    public DomainAttribute(params string[] values)
    {
        this.Values = values;
    }

    public override bool IsValid(object value)
    {
        if (null == value)
        {
            return true;
        }
        return this.Values.Any(item => value.ToString() == item);
    }

    public override string FormatErrorMessage(string name)
    {
        string[] values = this.Values.Select(value => string.Format("'{0}'",
            value)).ToArray();
        return string.Format(base.ErrorMessageString, name, string.Join(", ",
            values));
    }
}
```

2. 实例演示：针对 ValidationAttribute 特性的“自动化”验证 (S805)

由于 ASP.NET MVC 在进行参数绑定的时候会自动提取应用在目标参数类型或者数据成员上的 ValidationAttribute 特性并利用它们对提供的数据实施验证，所以我们不再需要像上面演示的实例一样自行在 Action 方法中实施验证，而只需要在定义参数类型 Person 的时候应用相应的 ValidationAttribute 特性将采用的验证规则与对应的数据成员相关联。

我们在数据类型或者数据成员上应用 ValidationAttribute 特性的时候可以直接指定验证失败时描述验证结果的错误消息。如下面的代码片段所示，ValidationAttribute 特性提供了一个 ErrorMessage 属性供我们直接指定作为错误消息的字符串。为了避免相同的错误消息重复指定，同时也让我们可以对错误消息进行单独维护并提供多语言的支持，ValidationAttribute 特性支持将错误信息定义在资源文件中。

如果我们将错误消息定义在资源文件中，在应用 ValidationAttribute 特性的时候需要利用它的 ErrorMessageResourceType 和 ErrorMessageResourceName 属性来指定对定义资源文件自动生成的类型和对应资源项的名称。在我们演示的这个实例中，采用资源文件来定义验证的错误消息。

```
public abstract class ValidationAttribute : Attribute
{
    //其他成员
    public string      ErrorMessage { get; set; }

    public Type        ErrorMessageResourceType { get; set; }
    public string      ErrorMessageResourceName { get; set; }
}
```

如下所示的是属性成员上应用了相关 ValidationAttribute 特性的 Person 类型的定义。我们在 3 个属性上均应用了 RequiredAttribute 特性将它们定义成必需的数据成员，Gender 和 Age 属性上则分别应用了 DomainAttribute 和 RangeAttribute 特性对有效属性值的范围作了相应限制。

```
public class Person
{
    [DisplayName("姓名")]
    [Required(ErrorMessageResourceName = "Required",
               ErrorMessageResourceType = typeof(Resources))]
    public string Name { get; set; }

    [DisplayName("性别")]
    [Required(ErrorMessageResourceName = "Required",
               ErrorMessageResourceType = typeof(Resources))]
    [Domain("M", "F", "m", "f", ErrorMessageResourceName = "Domain",
            ErrorMessageResourceType = typeof(Resources))]
    public string Gender { get; set; }
```

```

public string Gender { get; set; }

[DisplayName("年龄")]
[Required(ErrorMessageResourceName = "Required",
    ErrorMessageResourceType = typeof(Resources))]
[Range(18, 25, ErrorMessageResourceName = "Range",
    ErrorMessageResourceType = typeof(Resources))]
public int? Age { get; set; }
}

```

3 个 ValidationAttribute 特性采用的错误消息均定义在项目默认如图 8-3 所示的资源文件中 (我们可以采用这样的步骤创建这个资源文件: 右键选中项目文件, 并在上下文菜单中选择“属性”选项打开“项目属性”对话框, 然后在该对话框中选择“资源”Tab 页面, 通过单击页面中的链接创建一个资源文件)。

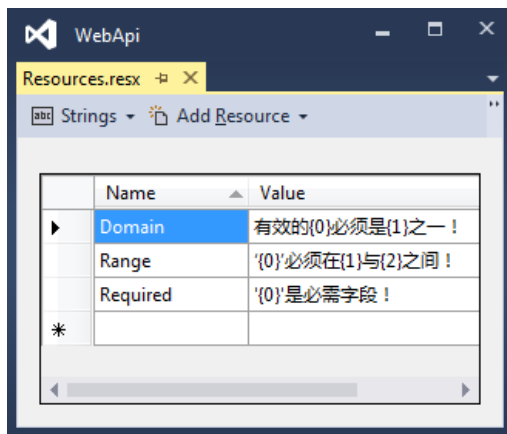


图 8-3 定义在资源文件中的错误消息

由于 ASP.NET MVC 会自动提取应用在绑定参数类型上的 ValidationAttribute 特性对绑定的参数实施自动化验证, 所以我们根本不需要在具体的 Action 方法中来对参数作手工验证。如下面的代码片段所示, 我们在 Action 方法 Index 中不再显式调用 Validate 方法, 但是运行该程序并在输入不合法数据的情况下提交表单后依然会得到如图 8-2 所示的输出结果。

```

public class HomeController : Controller
{
    // 其他成员
    [HttpPost]
    public ActionResult Index(Person person)
    {
        if (!ModelState.IsValid)
        {

```

```

        return View(person);
    }
    else
    {
        return Content("输入数据通过验证");
    }
}
}

```

8.1.5 让数据类型实现 IValidatableObject 接口

除了将验证规则通过 `ValidationAttribute` 特性直接定义在数据类型上并让 ASP.NET MVC 在进行参数绑定过程中据此来验证参数之外，我们还可以将验证逻辑直接定义在数据类型中。既然我们将验证操作直接实现在了数据类型上，就意味着对应的数据对象具有“自我验证”的能力，我们姑且将这些数据类型称为“自我验证类型”。这些自我验证类型是实现了具有如下定义的接口 `IValidatableObject`，该接口定义在“`System.ComponentModel.DataAnnotations`”命名空间下。

```

public interface IValidatableObject
{
    IEnumerable<ValidationResult> Validate(
        ValidationContext validationContext);
}

```

如上面的代码片段所示，`IValidatableObject` 接口具有唯一的方法 `Validate`，针对自身的验证就实现在该方法中。对于上面演示实例中定义的数据类型 `Person`，我们可以按照如下的形式将它定义成自我验证类型。

```

public class Person: IValidatableObject
{
    [DisplayName("姓名")]
    public string Name { get; set; }

    [DisplayName("性别")]
    public string Gender { get; set; }

    [DisplayName("年龄")]
    public int? Age { get; set; }

    public IEnumerable<ValidationResult> Validate(
        ValidationContext validationContext)
    {
        Person person = validationContext.ObjectInstance as Person;
        if (null == person)
        {

```

```

        yield break;
    }
    if (string.IsNullOrEmpty(person.Name))
    {
        yield return new ValidationResult("'Name'是必需字段",
            new string[] { "Name" });
    }

    if (string.IsNullOrEmpty(person.Gender))
    {
        yield return new ValidationResult("'Gender'是必需字段",
            new string[] { "Gender" });
    }
    else if (!new string[] { "M", "F" }.Any(
        g => string.Compare(person.Gender, g, true) == 0))
    {
        yield return new ValidationResult("有效'Gender'必须是'M','F'之一",
            new string[] { "Gender" });
    }

    if (null == person.Age)
    {
        yield return new ValidationResult("'Age'是必需字段",
            new string[] { "Age" });
    }
    else if (person.Age > 25 || person.Age < 18)
    {
        yield return new ValidationResult("'Age'必须在 18 到 25 周岁之间",
            new string[] { "Age" });
    }
}
}

```

如上面的代码片段所示，我们让 `Person` 类型实现了 `IValidatableObject` 接口。在实现的 `Validate` 方法中，我们从验证上下文中获取被验证的 `Person` 对象，并对其属性成员进行逐个验证。如果数据成员没有通过验证，我们通过一个 `ValidationResult` 对象封装错误消息和数据成员名称（属性名），该方法最终返回的是一个元素类型为 `ValidationResult` 的集合。在不对其他代码作任何改动的情况下，我们直接运行该程序并在输入不合法数据的情况下提交表单后依然会得到如图 8-2 所示的输出结果。（S806）

8.1.6 让数据类型实现 `IDataErrorInfo` 接口

上面我们让数据类型实现 `IValidatableObject` 接口并将具体的验证逻辑定义在实现的 `Validate` 方法中，这样的类型能够被 ASP.NET MVC 所识别，后者会自动调用该方法对绑定的数据对象实

施验证。如果我们让数据类型实现 `IDataErrorInfo` 接口，也能实现类似的自动化验证效果。

`IDataErrorInfo` 接口定义在“`System.ComponentModel`”命名空间下，它提供了一种标准的错误信息定制方式。如下面的代码片段所示，`IDataErrorInfo` 具有两个成员，只读属性 `Error` 用于获取基于自身的错误消息，而只读索引用于返回指定数据成员的错误消息。

```
public interface IDataErrorInfo
{
    string Error { get; }
    string this[string columnName] { get; }
}
```

同样是针对上面演示的实例，现在我们对需要被验证的数据类型 `Person` 进行了重新定义。如下面的代码片段所示，我们让 `Person` 实现了 `IDataErrorInfo` 接口。在实现的索引中，我们将索引参数 `columnName` 视为属性名称按照上面的规则对相应的属性成员实施验证，并在验证失败的情况下返回相应的错误消息。在不对其他代码作任何改动的情况下，我们直接运行该程序并在输入不合法数据的情况下提交表单后依然会得到如图 8-2 所示的输出结果。（S807）

```
public class Person : IDataErrorInfo
{
    [DisplayName("姓名")]
    public string Name { get; set; }

    [DisplayName("性别")]
    public string Gender { get; set; }

    [DisplayName("年龄")]
    public int? Age { get; set; }

    [ScaffoldColumn(false)]
    public string Error { get; private set; }

    public string this[string columnName]
    {
        get
        {
            switch (columnName)
            {
                case "Name":
                {
                    if (string.IsNullOrEmpty(this.Name))
                    {
                        return "'姓名'是必需字段";
                    }
                    return null;
                }
                case "Gender":
```

```

        {
            if (string.IsNullOrEmpty(this.Gender))
            {
                return "'性别'是必需字段";
            }
            else if (!new string[] { "M", "F" }.Any(
                g => string.Compare(this.Gender, g, true) == 0))
            {
                return "'性别'必须是'M','F'之一";
            }
            return null;
        }
        case "Age":
        {
            if (null == this.Age)
            {
                return "'年龄'是必需字段";
            }
            else if (this.Age > 25 || this.Age < 18)
            {
                return "'年龄'必须在 18 到 25 周岁之间";
            }
            return null;
        }
        default: return null;
    }
}
}
}
}

```

8.2 ModelValidator 及其提供策略

ASP.NET MVC 并不会对绑定的所有参数对象实施验证，真正能够被自动验证的仅限于复杂类型的参数。其实这也很好理解，不论是通过应用的 `ValidationAttribute` 特性来定义验证规则，还是将验证逻辑定义在实现了接口 `IValidatableObject/IDataErrorInfo` 的类型中，被验证的数据类型都是自定义的“复杂类型”。

8.2.1 ModelValidator 与 ModelValidatorProvider

虽然 Model 绑定的方式因被验证数据类型的差异而有所不同，但是 ASP.NET MVC 总是使用一个名为 `ModelValidator` 的对象来对绑定的数据对象实施验证。所有的 `ModelValidator` 类型

均继承自具有如下定义的抽象类 `ModelValidator`。它的 `GetClientValidationRules` 方法返回一个元素类型为 `ModelClientValidationRule` 的集合，而 `ModelClientValidationRule` 是对客户端验证规则的封装，我们会在客户端验证部分对其进行详细介绍。

```
public abstract class ModelValidator
{
    //其他成员
    public virtual IEnumerable<ModelClientValidationRule>
        GetClientValidationRules();
    public abstract IEnumerable<ModelValidationResult> Validate(
        object container);

    public virtual bool IsRequired { get; }
}
```

针对目标数据的验证是通过调用 `Validate` 方法来完成的，该方法的输入参数 `container` 表示的正是被验证的对象。正是因为被验证的总是一个复杂类型的对象，后者又被称作为一个具有若干数据成员的“容器”对象，所以对应的参数被命名为 `container`。`Validate` 方法表示验证结果的返回值并不是一个简单的布尔值，而是一个元素类型为具有如下定义的 `ModelValidationResult` 对象集合。

```
public class ModelValidationResult
{
    public string MemberName { get; set; }
    public string Message { get; set; }
}
```

`ModelValidationResult` 具有两个字符串类型属性 `MemberName` 和 `Message`，前者代表被验证数据成员的名称，后者表示错误消息。一般来说，如果 `ModelValidationResult` 对象来源于针对容器对象本身的验证，那么它的 `MemberName` 属性为空字符串。对于针对容器对象某个属性的验证来说，属性名称会作为返回的 `ModelValidationResult` 对象的 `MemberName` 属性。

`ModelValidationResult` 集合只有在验证失败的情况下才会返回。如果被验证的数据对象符合所有的验证规则，`Validate` 方法会直接返回 `Null` 或者一个空 `ModelValidationResult` 集合。值得一提的是，我们有时候会用 `ValidationResult` 的静态只读字段 `Success` 表示成功通过验证的结果，实际上该字段的值就是 `Null`。

```
public class ValidationResult
{
    //其他成员
    public static readonly ValidationResult Success;
}
```


ModelValidator 具有一个布尔类型的只读属性 `IsRequired`，表示该 ModelValidator 是否对目标数据进行“必需性”验证（即被验证的数据成员必须具有一个具体的值），该属性默认返回 `False`。我们可以通过应用 `RequiredAttribute` 特性将某个属性定义成“必需”的数据成员。

通过前面章节的介绍我们知道，ASP.NET MVC 大都采用 Provider 的模式来提供相应的组件，比如描述 Model 元数据的 `ModelMetadata` 通过对应的 `ModelMetadataProvider` 来提供，实现 Model 绑定的 `ModelBinder` 则可以通过对应的 `ModelBinderProvider` 来提供，用于实现 Model 验证的 `ModelValidator` 也不例外，它对应的提供者是 `ModelValidatorProvider`，对应的类型继承自具有如下定义的抽象类 `ModelValidatorProvider`。

```
public abstract class ModelValidatorProvider
{
    public abstract IEnumerable<ModelValidator> GetValidators(
        ModelMetadata metadata, ControllerContext context);
}
```

如上面的代码片段所示，`GetValidators` 方法具有两个参数，一个是用于描述被验证类型或者属性 Model 元数据的 `ModelMetadata` 对象，另一个是当前 `ControllerContext`。该方法返回的是一个元素类型为 `ModelValidator` 的集合。

ASP.NET MVC 通过静态类型 `ModelValidatorProviders` 对使用的 `ModelValidatorProvider` 进行注册。如下面的代码片段所示，`ModelValidatorProviders` 具有一个静态只读属性 `Providers`，对应的类型为 `ModelValidatorProviderCollection`，它表示基于整个 Web 应用范围的全局 `ModelValidatorProvider` 集合。

```
public static class ModelValidatorProviders
{
    public static ModelValidatorProviderCollection Providers { get; }
}

public class ModelValidatorProviderCollection :
    Collection<ModelValidatorProvider>
{
    public ModelValidatorProviderCollection();
    public ModelValidatorProviderCollection(
        IList<ModelValidatorProvider> list);
    public IEnumerable<ModelValidator> GetValidators(ModelMetadata metadata,
        ControllerContext context);
}
```

值得一提的是，用于描述 Model 元数据的 `ModelMetadata` 类型具有如下一个 `GetValidators` 方法，它返回的 `ModelValidator` 列表正是利用注册到 `ModelValidatorProviders` 静态属性 `Providers` 上的 `ModelValidatorProvider` 创建的。

```

public class ModelMetadata
{
    //其他成员
    public virtual IEnumerable<ModelValidator> GetValidators(
        ControllerContext context);
}

```

如图 8-4 所示的 UML 列出了组成 Model 验证系统的 3 个核心类型。具体的 Model 验证工作总是通过某个具体的 ModelValidator 来完成，作为 ModelValidator 提供者的 ModelValidatorProvider 注册在静态类型 ModelValidatorProviders 之上。

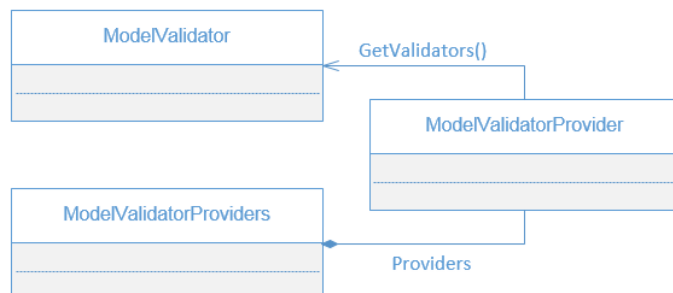


图 8-4 组成 Model 验证系统的 3 个核心类型

8.2.2 DataAnnotationsModelValidator

我们在上面一节中介绍了 3 种不同的“自动化验证”的编程方式，ASP.NET MVC 在内部会采用不同的 ModelValidator 来对绑定的参数实施验证。一个具体的 ModelValidator 通常由相应的 ModelValidatorProvider 来提供，在本节接下来的内容中我们将对 ASP.NET MVC 提供的原生的 ModelValidator 和对应的 ModelValidatorProvider 作详细的介绍。

对于上面提到的这 3 种验证编程方式，第一种（利用应用在数据类型或其数据成员上的 ValidationAttribute 特性来定义相应的验证规则）是最为常用的，这样的 Model 验证最终通过一个 DataAnnotationsModelValidator 对象来完成。一个 DataAnnotationsModelValidator 对象实际上是对一个 ValidationAttribute 特性的封装，这可以从如下所示的定义看出来。

```

public class DataAnnotationsModelValidator : ModelValidator
{
    public DataAnnotationsModelValidator(ModelMetadata metadata,
        ControllerContext context, ValidationAttribute attribute);
    public override IEnumerable<ModelClientValidationRule>
        GetClientValidationRules();

    public override IEnumerable<ModelValidationResult> Validate(

```

```

        object container);

    protected internal ValidationAttribute Attribute { get; }
    protected internal string                ErrorMessage { get; }
    public override bool                    IsRequired { get; }
}

```

`DataAnnotationsModelValidator` 的提供者是 `DataAnnotationsModelValidatorProvider`。采用应用的 `ValidationAttribute` 特性将验证规则直接与数据类型关联是我们最为常用的编程方式, 为了让读者对此具有深刻的认识, 我们会在第 9 章中对实现在 `DataAnnotationsModelValidator` 中的验证机制及 `DataAnnotationsModelValidatorProvider` 针对它的提供方式进行单独介绍。

8.2.3 ValidatableObjectAdapter

如果被验证的数据类型实现了 `IValidatable` 接口, ASP.NET MVC 会自动调用实现的 `Validate` 方法对其实施验证, 此时创建的 `ModelValidator` 是一个 `ValidatableObjectAdapter` 对象。`ValidatableObjectAdapter` 定义如下, 其 `Validate` 方法的实现逻辑很简单: 它直接调用被验证对象的 `Validate` 方法, 并将返回的 `ValidationResult` 对象转换成 `ModelValidationResult` 类型。

```

public class ValidatableObjectAdapter : ModelValidator
{
    public ValidatableObjectAdapter(ModelMetadata metadata,
        ControllerContext context);
    public override IEnumerable<ModelValidationResult> Validate(
        object container);
}

```

虽然 `ValidatableObjectAdapter` 继承自 `ModelValidator`, 但是 ASP.NET MVC 貌似没有将其视为一个真正意义上的 `ModelValidator`, 而是将其视为一个“适配器 (Adapter)”。ASP.NET MVC 也没有为 `ValidatableObjectAdapter` 定义单独的 `ModelValidatorProvider`, 它的提供者其实是上面提到过的 `DataAnnotationsModelValidatorProvider`, 至于它对 `ValidatableObjectAdapter` 采用的提供策略, 我们会在“下篇”中对其进行详细介绍。

8.2.4 DataErrorInfoModelValidator

如果我们让数据类型实现 `IDataErrorInfo` 接口, 可以利用实现的 `Error` 属性和索引提供针对自身及所属数据成员的验证错误信息。针对这样的数据类型, ASP.NET MVC 最终会创建一个 `DataErrorInfoModelValidator` 对象来对其实施验证, `DataErrorInfoClassModelValidator` 和 `DataErrorInfoPropertyModelValidator` 是两个具体的 `DataErrorInfoModelValidator`。

`DataErrorInfoClassModelValidator` 和 `DataErrorInfoPropertyModelValidator` 是两个内部类型。前者针对容器对象自身实施验证，所以它只需要从实现的 `Error` 属性中提取错误消息并将其转换成返回的 `ModelValidationResult` 对象。后者则专门验证容器对象的某个属性，它在实现的 `Validate` 方法中会利用属性名从实现的索引中提取相应的错误消息并将其转换成返回的 `ModelValidationResult` 对象。

```
internal sealed class DataErrorInfoClassModelValidator : ModelValidator
{
    public DataErrorInfoClassModelValidator(ModelMetadata metadata,
        ControllerContext controllerContext);
    public override IEnumerable<ModelValidationResult> Validate(
        object container);
}

internal sealed class DataErrorInfoPropertyModelValidator : ModelValidator
{
    public DataErrorInfoPropertyModelValidator(ModelMetadata metadata,
        ControllerContext controllerContext);
    public override IEnumerable<ModelValidationResult> Validate(
        object container);
}
```

ASP.NET MVC 最终利用具有如下定义的 `DataErrorInfoModelValidatorProvider` 来提供这两种类型的 `DataErrorInfoModelValidator`。对于其实现的 `GetValidators` 方法来说，如果被验证对象的类型实现了 `IDataErrorInfo` 接口，它会创建一个 `DataErrorInfoClassModelValidator` 对象并添加到返回的 `ModelValidator` 列表中。如果被验证的是容器类型的某个属性值，并且容器类型实现了 `IDataErrorInfo` 接口，它会创建一个 `DataErrorInfoPropertyModelValidator` 对象并添加到返回的 `ModelValidator` 列表中。

```
public class DataErrorInfoModelValidatorProvider : ModelValidatorProvider
{
    public override IEnumerable<ModelValidator> GetValidators(
        ModelMetadata metadata, ControllerContext context);
}
```

8.2.5 ClientModelValidator

ASP.NET MVC 不仅仅可以利用 `ModelValidator` 在服务端对绑定的参数对象实施验证，而且还对客户端验证提供了支持。虽然客户端验证是由 JavaScript 来实现的，但是针对被验证的表单元素的验证规则必须出现在最终生成的 HTML 中。

顾名思义，`ClientModelValidator` 是一种专门针对客户端验证的 `ModelValidator`，当我们调

用 `HtmlHelper<TModel>` 的模板方法 `EditorFor/EditorForModel` 将某个数据对象或其属性成员以编辑模式呈现在某个 View 中时, ASP.NET MVC 会利用对应的 `ClientModelValidator` 来提取客户端验证规则并将其写入到最终生成的 HTML 中。

`ClientModelValidator` 是定义在程序集 “System.Web.Mvc.dll” 中的内部类型。如下面的代码片段所示, 当我们通过调用构造函数创建一个 `ClientModelValidator` 的时候, 不仅需要指定描述被验证对象类型的 `ModelMetadata` 和当前 `ControllerContext`, 还需要以字符串的形式指定验证类型和错误消息。

```
internal class ClientModelValidator : ModelValidator
{
    public ClientModelValidator(ModelMetadata metadata,
        ControllerContext controllerContext, string validationType,
        string errorMessage);

    public sealed override IEnumerable<ModelClientValidationRule>
        GetClientValidationRules();
    public sealed override IEnumerable<ModelValidationResult> Validate(
        object container);
}
```

由于 `ClientModelValidator` 仅限于客户端验证, 用于实现服务端验证的 `Validate` 方法总是返回一个空的 `ModelValidationResult` 集合 (表示验证成功)。它的 `GetClientValidationRules` 方法返回的是一个元素类型为 `ModelClientValidationRule` 的集合, 表示需要出现在 HTML 中的客户端验证规则。

`ClientModelValidator` 具有两个继承者, 分别是针对数值类型和日期类型验证的 `NumericModelValidator` 和 `DateModelValidator`。如下面的代码片段所示, 这两个 `ClientModelValidator` 采用的验证类型的分别是 “number” 和 “date”。表示错误消息的字符串是从内部维护的资源文件中获取的, 这实际上带来了一个问题, 就是我们无法对错误消息进行定制。

```
internal sealed class NumericModelValidator : ClientModelValidator
{
    public NumericModelValidator(ModelMetadata metadata,
        ControllerContext controllerContext)
        : base(metadata, controllerContext, "number",
            ClientDataTypeModelValidatorProvider.GetFieldMustBeNumericResource(
                controllerContext))
    {}
}

internal sealed class DateModelValidator : ClientModelValidator
{
    public DateModelValidator(ModelMetadata metadata,
        ControllerContext controllerContext)
```

```

        : base(metadata, controllerContext, "date",
        ClientDataTypeModelValidatorProvider.GetFieldMustBeDateResource(
        controllerContext))
    {}
}

```

NumericModelValidator 和 DateModelValidator 这两种 ClientModelValidator 最终是通过具有如下定义的 ClientDataTypeModelValidatorProvider 来提供的。在实现的 GetValidators 方法中，ClientDataTypeModelValidatorProvider 会根据指定的 ModelMetadata 判断被验证类型是否属于数字或者 DateTime 类型，如果是则直接返回一个包含单个 NumericModelValidator 或者 DateModelValidator 对象的 ModelValidator 集合。在这里被视为数字的类型包括 byte、sbyte、short、ushort、int、uint、long、ulong、float、double 和 decimal 等。

```

public class ClientDataTypeModelValidatorProvider : ModelValidatorProvider
{
    public ClientDataTypeModelValidatorProvider();
    public override IEnumerable<ModelValidator> GetValidators(
        ModelMetadata metadata, ControllerContext context);
}

```

我们已经介绍了 3 种类型的 ModelValidator 和它们对应的 ModelValidatorProvider (ValidatableObjectAdapter 的提供者 DataAnnotationsModelValidatorProvider)，ASP.NET MVC 在默认情况下会使用哪些呢？如下所示的是用于注册 ModelValidatorProvider 的静态类型 ModelValidatorProviders 的完整定义，我们可以看出 3 个 ModelValidatorProvider 对象会被默认注册，其类型正是上面介绍的这 3 个。

```

public static class ModelValidatorProviders
{
    private static readonly ModelValidatorProviderCollection _providers;

    static ModelValidatorProviders()
    {
        ModelValidatorProviderCollection providers =
            new ModelValidatorProviderCollection();
        providers.Add(new DataAnnotationsModelValidatorProvider());
        providers.Add(new DataErrorInfoModelValidatorProvider());
        providers.Add(new ClientDataTypeModelValidatorProvider());
        _providers = providers;
    }

    public static ModelValidatorProviderCollection Providers
    {
        get
        {
            return _providers;
        }
    }
}

```

}

8.2.6 CompositeModelValidator

虽然 `CompositeModelValidator` 仅仅是定义在程序集 “`System.Web.Mvc.dll`” 中的一个私有类型而已, 但是它在 ASP.NET MVC 的 Model 验证系统中具有重要的地位, 可以说真正用于 Model 验证的 `ModelValidator` 就是这么一个对象。

```
private class CompositeModelValidator : ModelValidator
{
    public CompositeModelValidator(ModelMetadata metadata,
        ControllerContext controllerContext);
    public override IEnumerable<ModelValidationResult> Validate(
        object container);
}
```

从命名上可以看出, `CompositeModelValidator` 实际上并不是一个真正对数据对象实施验证的 `ModelValidator`, 而是一系列 `ModelValidator` 的组合。它根据描述数据类型或其属性的 `ModelMetadata` 动态地获取相应的 `ModelValidator` (通过调用 `ModelMetadata` 的 `GetValidators` 方法) 对目标数据实施验证。抽象类 `ModelValidator` 具有一个静态的 `GetModelValidator` 方法, 它会根据指定的 `ModelMetadata` 和 `ControllerContext` 得到相应的 `ModelValidator` 对象。如下面的代码片段所示, 它返回的正是一个 `CompositeModelValidator` 对象。

```
public abstract class ModelValidator
{
    //其他成员
    public static ModelValidator GetModelValidator(ModelMetadata metadata,
        ControllerContext context)
    {
        return new CompositeModelValidator(metadata, context);
    }
}
```

当 `CompositeModelValidator` 被用于验证某个容器对象的时候, 它会先验证其属性成员。针对容器对象自身的验证只有在所有属性值都通过验证的情况下才会进行。具体的逻辑是这样的: 它通过调用描述容器类型的 `ModelMetadata` 对象的 `Properties` 属性得到描述所有属性成员的 `ModelMetadata` 对象, 然后调用它们的 `GetValidators` 方法得到一组 `ModelValidator` 对相应的属性值实施验证, 验证得到的 `ModelValidationResult` 会被添加到最终返回的 `ModelValidationResult` 列表中。

如果在对所有属性实施验证之后该 `ModelValidationResult` 列表依然为空 (所有的属性均成功通过验证), `CompositeModelValidator` 才会获取针对容器类型的 `ModelMetadata` 对象, 并采用

调用其 `GetValidators` 方法获取的 `ModelValidator` 列表对容器对象本身实施验证。表示验证结果的 `ModelValidationResult` 对象被添加到最终返回的列表中。

实例演示：CompositeModelValidator 采用的验证行为 (S808, S809)

为了使读者对 `CompositeModelValidator` 的验证逻辑有一个深刻的理解，我们来演示一个具体的实例。我们在一个 ASP.NET MVC 应用中定义了如下一个名为 `AlwaysFailsAttribute` 的验证特性。如下面的代码片段所示，由于重写的 `IsValid` 方法总是返回 `False`，意味着针对数据的验证总是会失败。我们还重写了只读属性 `TypeId`，让它真正能够唯一标识一个 `AlwaysFailsAttribute` 特性实例（具体原因我们会在本章后续部分予以介绍）。

```
[AttributeUsage( AttributeTargets.Class| AttributeTargets.Property)]
public class AlwaysFailsAttribute : ValidationAttribute
{
    private object typeId;
    public override bool IsValid(object value)
    {
        return false;
    }
    public override object TypeId
    {
        get { return typeId ?? (typeId = new object()); }
    }
}
```

我们将 `AlwaysFailsAttribute` 特性应用到表示联系人的 `Contact` 类型上。如下面的代码片段所示，我们在 `Contact` 和 `Address` 的类型和各自的属性上都应用了该特性，并且指定了相应的错误消息。

```
[AlwaysFails(ErrorMessage = "Contact")]
public class Contact
{
    [AlwaysFails(ErrorMessage = "Contact.Name")]
    public string Name { get; set; }

    [AlwaysFails(ErrorMessage = "Contact.PhoneNo")]
    public string PhoneNo { get; set; }

    [AlwaysFails(ErrorMessage = "Contact.EmailAddress")]
    public string EmailAddress { get; set; }

    [AlwaysFails(ErrorMessage = "Contact.Address")]
    public Address Address { get; set; }
}

[AlwaysFails(ErrorMessage = "Address")]
```



```

public class Address
{
    [AlwaysFails(ErrorMessage = "Address.Province")]
    public string Province { get; set; }

    [AlwaysFails(ErrorMessage = "Address.City")]
    public string City { get; set; }

    [AlwaysFails(ErrorMessage = "Address.District")]
    public string District { get; set; }

    [AlwaysFails(ErrorMessage = "Address.Street")]
    public string Street { get; set; }
}

```

我们创建了一个具有如下定义的 `HomeController` 类，并在 `Action` 方法 `Index` 中使用当前注册的 `ModelMetadataProvider` 创建了一个描述 `Contact` 类型的 `ModelMetadata` 对象，然后将它和当前 `ControllerContext` 作为参数调用抽象类型 `ModelValidator` 的静态方法 `GetValidator` 创建一个 `CompositeModelValidator` 对象。我们利用该 `CompositeModelValidator` 来验证创建的 `Contact` 对象，并将表示验证结果的 `ModelValidationResult` 列表作为 `Model` 呈现在默认的 `View` 中。

```

public class HomeController : Controller
{
    public ActionResult Index()
    {
        Address address = new Address
        {
            Province = "江苏",
            City = "苏州",
            District = "工业园区",
            Street = "星湖街 328 号"
        };

        Contact contact = new Contact
        {
            Name = "张三",
            PhoneNo = "123456789",
            EmailAddress = "zhangsan@gmail.com",
            Address = address
        };

        ModelMetadata metadata = ModelMetadataProviders.Current
            .GetMetadataForType(() => contact, typeof(Contact));
    }
}

```

```

        ModelValidator validator = ModelValidator.GetModelValidator(metadata,
            ControllerContext);
        return View(validator.Validate(contact));
    }
}

```

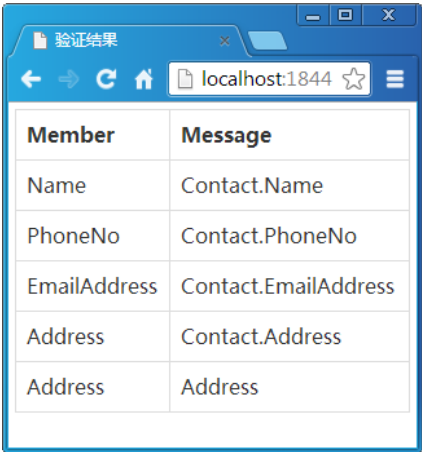
如下所示的是 Action 方法 Index 对应 View 的定义，这是一个 Model 类型为 IEnumerable<ModelValidationResult> 的强类型 View。我们在该 View 中将包含在集合中的每一个 ModelValidationResult 对象的成员名称和错误消息通过表格的形式呈现出来。

```

@model IEnumerable<ModelValidationResult>
<html>
    <head>
        <title>验证结果</title>
    </head>
    <body>
        <table >
            <tr><th>Member</th><th>Message</th></tr>
            @foreach (ModelValidationResult result in Model)
            {
                string propertyName = string.IsNullOrEmpty(result.MemberName) ?
                    "N/A" : result.MemberName;
                <tr><td>@propertyName</td><td>@result.Message</td></tr>
            }
        </table>
    </body>
</html>

```

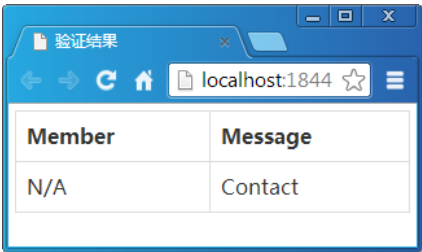
该程序运行后会在浏览器中呈现出如图 8-5 所示的结果，可以看出 CompositeModelValidator 对 Contact 对象实施验证得到的 5 个 ModelValidationResult 都来源于针对 4 个属性的验证，应用在 Contact 类型上的 AlwaysFailsAttribute 特性并没有参与验证。（S808）



Member	Message
Name	Contact.Name
PhoneNo	Contact.PhoneNo
EmailAddress	Contact.EmailAddress
Address	Contact.Address
Address	Address

图 8-5 CompositeModelValidator 的验证规则 (1)

按照前面介绍的“针对容器对象本身的验证只有在所有属性通过验证的情况下才会进行”的原理，为了让 Contact 的 4 个属性通过验证，我们将应用在 4 个属性和 Address 类型上的 AlwaysFailsAttribute 特性注释掉，只保留应用在 Contact 类型和 Address 4 个属性上的 AlwaysFailsAttribute 特性。再次运行我们的程序将会在浏览器中得到如图 8-6 所示的输出结果，不难看出输出的 ModelStateResult 来源于应用在 Contact 类型上的 AlwaysFailsAttribute 特性。(S809)



Member	Message
N/A	Contact

图 8-6 CompositeModelValidator 的验证规则 (2)

8.3 Model 验证的实施

Model 绑定解决了针对目标 Action 方法参数的初始化问题，而 Model 验证的目的在于对绑定的参数对象实施验证以确保输入数据的有效性，Model 验证是伴随着 Model 绑定进行的。在

上面一节中我们详细地介绍了真正用于 Model 验证的 `ModelValidator` 及相关的提供机制，接下来讨论在这个以 `ModelValidator` 为核心的 Model 验证系统中，针对通过 Model 绑定得到的数据对象的验证是如何实现的。

8.3.1 Model 绑定过程中的验证

在前面我们不止一次地提到，Model 验证可以看成是 Model 绑定的一个中间环节，默认情况下采用的 Model 绑定实现在 `DefaultModelBinder` 中。那么现在有这么一个问题：是 `DefaultModelBinder` 得到最终的参数对象后再递交给 `ModelValidator` 实施验证呢，还是它在实施 Model 绑定的过程中动态地调用 `ModelValidator` 对由 `ValueProvider` 提供的数据值实施验证？

实际上我们前面演示的两个实例已经回答了这个问题。通过前面演示的两个例子我们知道，`CompositeModelValidator` 这个默认 `ModelValidator` 在进行 Model 验证过程中并不是递归进行的（S808），但是从整个 Model 绑定过程来看，Model 验证却具有递归性，所以 Model 绑定和 Model 验证绝对不可能是一前一后的过程，唯一的可能是 `DefaultModelBinder` 在递归地进行 Model 绑定的过程中调用 `ModelValidator` 对提供的数据实施验证。

同样以针对 `Contact` 类型的 Model 绑定为例，当 `DefaultModelBinder` 通过得到一个被初始化的空 `Contact` 对象之后，会将描述 `Contact` 类型的 `ModelMetadata` 对象作为参数调用 `ModelValidator` 的静态方法 `GetModelValidator`，得到的 `CompositeModelValidator` 会被用于对 `Contact` 对象实施验证。由于 `CompositeModelValidator` 的 Model 验证不具有递归性，所以只有应用在 `Contact` 4 个属性（`Name`、`PhoneNo`、`Email` 和 `Address`）及其自身类型上的验证规则在本轮验证中有效。

由于 `Contact` 的 `Address` 属性是一个复杂类型，所以 `DefaultModelBinder` 在针对 `Contact` 类型的 Model 绑定过程中会递归地创建一个空 `Address` 对象作为 `Contact` 对象的 `Address` 属性。在完成对 `Address` 对象的绑定之后，又会调用 `ModelValidator` 的静态方法 `GetModelValidator` 根据描述 `Address` 类型的 `ModelMetadata` 得到一个 `CompositeModelValidator`，初始化后的 `Address` 对象将交给它验证。

描述 Model 元数据的 `ModelMetadata` 具有一个树形层次化结构，我们的验证规则可以应用到每一个节点上。`DefaultModelBinder` 就是在递归地绑定复杂对象的过程中对绑定后的对象实施验证，从而使各个层次上的验证得以实现。不过 `CompositeModelValidator` 只有在所有属性值都验证通过的情况下，才会使用应用在类型上的验证规则对数据对象实施验证，所以验证的结果也不能完全反映所有的验证规则。

8.3.2 实例演示：模拟 Model 绑定中的验证 (S810)

在第7章“Model 的绑定 (下)”中，我们自定义了一个 `MyDefaultModelBinder` 实现了针对简单类型、复杂类型、集合和字典的绑定。在本例中，我们在这个自定义的 `ModelBinder` 中引入 Model 验证部分。通过前面的介绍我们知道，真正实施 Model 验证的是通过 `ModelValidator` 的静态方法 `GetModelValidator` 创建的 `CompositeModelValidator` 对象，我们按照其采用的 Model 验证逻辑自定义了如下一个 `MyCompositeModelValidator` 类型。

```
public class MyCompositeModelValidator: ModelValidator
{
    public MyCompositeModelValidator(ModelMetadata metadata,
        ControllerContext controllerContext)
        : base(metadata, controllerContext)
    {}

    public override IEnumerable<ModelValidationResult> Validate(
        object container)
    {
        bool isPropertiesValid = true;
        foreach (ModelMetadata propertyMetadata in Metadata.Properties)
        {
            foreach (ModelValidator validator in
                propertyMetadata.GetValidators(this.ControllerContext))
            {
                IEnumerable<ModelValidationResult> results =
                    validator.Validate(this.Metadata.Model);
                if (results.Any())
                {
                    isPropertiesValid = false;
                }
                foreach (ModelValidationResult result in results)
                {
                    string key = (propertyMetadata.PropertyName ?? "") + "." +
                        (result.MemberName ?? "");
                    yield return new ModelValidationResult
                    {
                        MemberName = key,
                        Message = result.Message
                    };
                }
            }
        }

        if (isPropertiesValid)
        {

```

```

        foreach (ModelValidator validator in
            Metadata.GetValidators(this.ControllerContext))
        {
            IEnumerable<ModelValidationResult> results =
                validator.Validate(Metadata.Model);
            foreach (ModelValidationResult result in results)
            {
                yield return result;
            }
        }
    }
}

```

重写的 `Validate` 方法具体采用这样的验证策略：它先获取描述被验证数据类型或属性成员的 `ModelMetadata` 对象，然后遍历所有描述其属性成员的 `ModelMetadata` 对象。对于每一个针对属性成员的 `ModelMetadata` 对象，我们调用其 `GetModelValidators` 方法得到应用在该属性上的 `ModelValidator` 列表。我们接下来调用列表中每一个 `ModelValidator` 的 `Validate` 方法对属性值实施验证，并根据返回值创建相应的 `ModelValidationResult` 对象，该对象被添加到最终返回的 `ModelValidationResult` 集合中。

只有在所有属性通过验证的情况下，我们才根据当前 `ModelMetadata` 对象获取相应的 `ModelValidator` 列表对容器对象自身实施验证，验证的结果直接添加到最终返回的 `ModelValidationResult` 集合中。

在自定义的 `MyDefaultModelBinder` 中，我们定义了单独的方法来完成针对简单类型和复杂类型的 `Model` 绑定，但是只需要在进行针对复杂类型的 `Model` 绑定过程中利用 `CompositeModelValidator` 进行 `Model` 验证。`CompositeModelValidator` 能够完成针对容器对象所有属性及其自身的验证，虽然它不递归地去验证复杂类型属性的数据成员，但是由于 `Model` 绑定本身是一个递归的过程，所以从整个流程上看 `Model` 验证也是递归进行的。

针对复杂数据类型的 `Model` 验证实现在 `MyDefaultModelBinder` 的 `BindComplexModel` 方法中。如下面的代码片段所示，我们在目标容器对象生成之后会利用创建的 `CompositeModelValidator` 对象对它进行验证。对于返回的每一个 `ModelValidationResult`，我们将其错误消息添加到相应的 `ModelState` 之中。

```

public class MyDefaultModelBinder : IModelBinder
{
    private object BindComplexModel(ControllerContext controllerContext,
        ModelBindingContext bindingContext)
    {
        //其他操作
    }
}

```

```

        Type modelType = bindingContext.ModelType;
        object model = this.CreateModel(controllerContext, bindingContext,
            modelType);
        bindingContext.ModelMetadata.Model = model;
        ICustomTypeDescriptor modelTypeDescriptor =
            new AssociatedMetadataTypeTypeDescriptorProvider(modelType)
                .GetTypeDescriptor(modelType);
        PropertyDescriptorCollection propertyDescriptors =
            modelTypeDescriptor.GetProperties();
        foreach (PropertyDescriptor propertyDescriptor in
            propertyDescriptors)
        {
            this.BindProperty(controllerContext, bindingContext,
                propertyDescriptor);
        }

        //Model 验证
        ModelMetadata metadata = ModelMetadataProviders.Current
            .GetMetadataForType(() => model, modelType);
        MyCompositeModelValidator validator =
            new MyCompositeModelValidator(metadata, controllerContext);
        foreach (ModelValidationResult result in validator.Validate(null))
        {
            string key = (bindingContext.ModelName ?? "") + "." +
                (result.MemberName ?? "");
            controllerContext.Controller.ViewData.ModelState
                .AddModelError(key.Trim('.'), result.Message);
        }

        return model;
    }
}

```

在第7章“Model 的绑定 (下篇)”中已经验证过了自定义 `MyDefaultModelBinder` 的 Model 绑定功能, 现在我们通过一个简单的实例来验证刚刚增加的 Model 验证功能。我们直接使用上面实例中定义的 `Contact` 类型, 并且在 `Contact` 和 `Address` 类型和属性上应用自定义的 `AlwaysFailsAttribute` 特性。接下来我们定义了如下一个 `HomeController`, 针对 GET 请求的 Action 方法 `Index` 直接将一个空 `Contact` 对象呈现在默认的 View 中, 而针对 POST 请求的 Action 方法 `Index` 则将作为参数的 `Contact` 对象呈现在默认的 View 中。

```

public class HomeController : Controller
{
    public ActionResult Index()
    {
        return View(new Contact());
    }
}

```

```

[HttpPost]
public ActionResult Index(Contact contact)
{
    return View(contact);
}
}

```

如下所示的是 Action 方法 Index 对应 View 的定义, 这是一个 Model 类型为 Contact 的强类型 View。我们在该 View 中将作为 Model 的 Contact 对象的所有属性 (包括 Address 的所有属性) 以编辑模式呈现在一个表单之中, 该表单具有一个用于提交的“保存”按钮。

```

@model Contact
<html>
<head>
    <title>Model 绑定中的验证</title>
</head>
<body>
    @using (Html.BeginForm())
    {
        @Html.EditorForModel()
        @Html.EditorFor(m=>m.Address)
        <input type="submit" value="保存" />
    }
</body>
</html>

```

为了让 ASP.NET MVC 采用我们自定义的 MyDefaultModelBinder 来绑定 Action 方法 Index 中的参数, 我们在 Global.asax 中按照如下方法将创建的 MyDefaultModelBinder 对象注册到 Contact 和 Address 类型上。

```

public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        //其他成员
        ModelBinders.Binders.Add(typeof(Contact),
            new MyDefaultModelBinder());
        ModelBinders.Binders.Add(typeof(Address),
            new MyDefaultModelBinder());
    }
}

```

该程序运行之后会在浏览器中呈现一个“编辑联系人信息”的页面, 直接单击“保存”按钮会呈现出如图 8-7 所示的输出结果, 文本框右侧显示的文本正是应用在 Contact 和 Address

相应属性上的 AlwaysFailsAttribute 特性定义的错误消息。

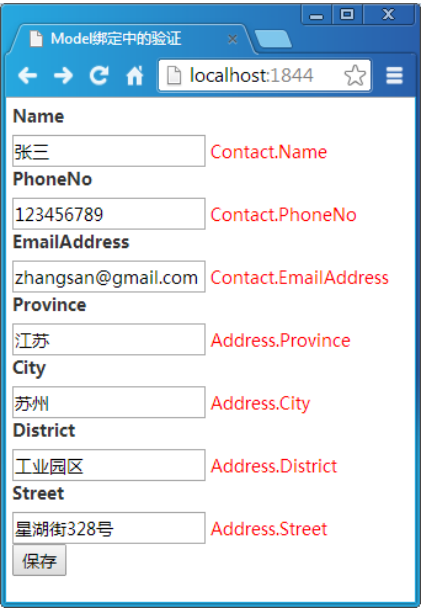


图 8-7 实现在自定义 MyDefaultModelBinder 中的 Model 验证

8.3.3 针对“必需”数据成员的验证

通过上面的介绍我们知道，作为默认 ModelBinder 的 DefaultModelBinder 会在对复杂类型实施 Model 绑定之后会对绑定的数据对象实施验证。换句话说，在每次针对复杂类型的绑定迭代中，DefaultModelBinder 会先生成一个完整的数据对象，再利用创建的 CompositeModelValidator 对其实施验证。但是在这之前，DefaultModelBinder 会对必需数据成员实施验证。

我们知道 DefaultModelBinder 针对复杂类型的 Model 绑定会先从创建空对象开始，然后再针对其属性实施 Model 绑定并得到对应的属性值。在对属性进行赋值之前，DefaultModelBinder 会利用注册的 ModelProvider 根据描述属性的 ModelMetadata 创建相应的 ModelValidator 列表，并从中筛选出 IsRequired 属性为 True 的 ModelValidator 对属性数值实施验证。

对于自定义的 MyDefaultModelBinder 来说，我们将针对容器对象属性成员的 Model 绑定实现在 BindProperty 方法中，现在我们将针对必需数据成员的验证实现在该方法中。如下面的代码片段所示，我们将针对必需数据成员的验证定义在 ValidateRequiredPropertyValue 方法中，在

对属性赋值之前调用其方法对绑定得到的属性值实施验证。

```

public class MyDefaultModelBinder : IModelBinder
{
    //其他成员
    private void BindProperty(ControllerContext controllerContext,
        ModelBindingContext bindingContext,
        PropertyDescriptor propertyDescriptor)
    {
        //将属性名附加到现有的前缀上
        string prefix = (bindingContext.ModelName ?? "") + "." +
            (propertyDescriptor.Name ?? "");
        prefix = prefix.Trim('.');

        //针对属性创建绑定上下文
        ModelMetadata metadata = bindingContext
            .PropertyMetadata[propertyDescriptor.Name];
        ModelBindingContext context = new ModelBindingContext
        {
            ModelName = prefix,
            ModelMetadata = metadata,
            ModelState = bindingContext.ModelState,
            ValueProvider = bindingContext.ValueProvider
        };

        //针对属性实施 Model 绑定并对属性赋值
        object propertyValue = ModelBinders.Binders
            .GetBinder(propertyDescriptor.PropertyType).BindModel(
                controllerContext, context);
        if (bindingContext.ModelMetadata.ConvertEmptyStringToNull &&
            object.Equals(propertyValue, string.Empty))
        {
            propertyValue = null;
        }
        context.ModelMetadata.Model = propertyValue;
        if (null == propertyValue)
        {
            this.ValidateRequiredPropertyValue(controllerContext,
                bindingContext, metadata, propertyValue);
        }
        propertyDescriptor.SetValue(bindingContext.Model, propertyValue);
    }

    private void ValidateRequiredPropertyValue(
        ControllerContext controllerContext,
        ModelBindingContext bindingContext,
        ModelMetadata propertyModelMetadata, object propertyValue)
    {
        string key = (bindingContext.ModelName ?? "") + "." +
            (propertyModelMetadata.PropertyName ?? "");
        key = key.Trim('.');
    }
}

```

```
ModelStateDictionary modelState = bindingContext.ModelState;

ModelValidator validator = ModelValidatorProviders.Providers
    .GetValidators(propertyModelMetadata, controllerContext)
    .FirstOrDefault(v => v.IsRequired);
if (null != validator)
{
    foreach (ModelValidationResult result in
        validator.Validate(bindingContext.Model))
    {
        modelState.AddModelError(key, result.Message);
    }
}
}
```

第 9 章 Model 的验证 (下篇)

在“上篇”中我们介绍了 3 种不同的自动化验证编程方式，ASP.NET MVC 会为之创建 3 种不同的 `ModelValidator` 对绑定的参数对象实施验证。对于这 3 种验证方式来说，最为常用的还是借助应用在数据类型或者属性成员上的 `ValidationAttribute` 特性来定义相应的验证规则。这种情况下对绑定参数实施验证的是一个 `DataAnnotationsModelValidator` 对象，它的提供者是 `DataAnnotationsModelValidatorProvider`，这两个对象是本章论述的核心。

9.1 ValidationAttribute 特性

应用在数据类型或者属性成员上的 `ValidationAttribute` 特性不仅仅用于定义相应的验证规则，其自身就具有验证的能力。一个 `DataAnnotationsModelValidator` 对象是对一个 `ValidationAttribute` 特性的封装，并直接利用它来对绑定的数据实施验证。所有的 `ValidationAttribute` 特性类型均继承自具有如下定义的抽象类型 `ValidationAttribute`，该类型定义在命名空间 “`System.ComponentModel.DataAnnotations`” 下。

```
public abstract class ValidationAttribute : Attribute
{
    public string      ErrorMessage { get; set; }
    public string      ErrorMessageResourceName { get; set; }
    public Type        ErrorMessageResourceType { get; set; }
    protected string  ErrorMessageString {get;}

    public virtual string FormatErrorMessage(string name);
    public virtual bool IsValid(object value);
    protected virtual ValidationResult IsValid(object value,
        ValidationContext validationContext)
    public void Validate(object value, string name);
    public ValidationResult GetValidationResult(object value,
        ValidationContext validationContext);
}
```

如上面的代码片段所示，`ValidationAttribute` 具有一个字符串类型的 `ErrorMessage` 属性用于指定错误消息。出于对本地化和对错误消息单独维护的需要，我们可以采用资源文件来存储错误消息，在这种情况下只需要通过 `ErrorMessageResourceName` 和 `ErrorMessageResourceType` 这两个属性指定错误消息所在资源项的名称和类型即可。如果我们通过 `ErrorMessage` 属性指定一个字符串作为验证错误消息，同时通过 `ErrorMessageResourceName` 和 `ErrorMessageResourceType` 属性指定错误消息资源项对应的名称和类型，则后者具有更高的优先级。`ValidationAttribute` 具有一个受保护的只读属性 `ErrorMessageString` 用于返回最终的错误消息文本。

对于错误消息的定义，我们可以指定完整的消息内容，比如“年龄必须在 18 至 25 之间”。但是对于像资源文件这种对错误消息进行独立维护的情况，为了让定义的资源文本能够最大限度地被重用，我们倾向于定义一个包含占位符的文本模板，比如“{DisplayName}必须在 {LowerBound}和{UpperBound}之间”，这样的消息适用于所有针对数值范围的验证。模板中的占位符可以在虚方法 `FormatErrorMessage` 中进行替换，该方法中的参数 `name` 实际上代表的是

被验证数据成员的显示名称, 即 `ModelMetadata` 的 `DisplayName` 属性。

`FormatErrorMessage` 方法在 `ValidationAttribute` 中仅仅是按照如下方式调用 `String` 的静态方法 `Format` 并将参数 `name` 作为替换占位符的参数, 所以在默认情况下定义错误消息模板只允许包含一个针对显示名称的占位符 “{0}”。如果具有额外的占位符, 或者需要采用非序号 (“{0}”) 占位符定义方式 (比如采用类似于 “{DisplayName}” 这种基于文字的占位符更具可读性), 我们需要重写 `FormatErrorMessage` 方法。

```
public abstract class ValidationAttribute : Attribute
{
    //其他成员
    public virtual string FormatErrorMessage(string name)
    {
        return string.Format(CultureInfo.CurrentCulture,
            ErrorMessageString, new object[] { name });
    }
}
```

9.1.1 数据是如何被验证的

当我们通过继承抽象类 `ValidationAttribute` 创建自己的验证特性的时候, 可以将验证逻辑定义在任意一个重写的 `IsValid` 方法中。之所以能够通过重写任意一个 `IsValid` 方法来实现对目标数据的验证, 原因在于定义在 `ValidationAttribute` 的这两个 `IsValid` 方法之间存在相互调用的关系。很显然, 这种相互调用必然造成 “死循环”, 所以我们需要重写至少其中一个方法来避免 “死循环” 的发生。这里的 “死循环” 被加上引号, 是因为 `ValidationAttribute` 在内部作了处理, 当这种情况出现的时候会抛出一个 `NotImplementedException` 异常。

```
//调用公有 IsValid 方法
public class ValidatorAttribute : ValidationAttribute
{
    static void Main()
    {
        ValidatorAttribute validator = new ValidatorAttribute();
        validator.IsValid(new object());
    }
}

//调用受保护 IsValid 方法
public class ValidatorAttribute : ValidationAttribute
{
    static void Main()
    {

```

```

        ValidatorAttribute validator = new ValidatorAttribute();
        validator.IsValid(new object(), null);
    }
}

```

我们通过一个简单的实例来演示自定义 `ValidatorAttribute` 对两个 `IsValid` 方法进行重写的必要性。我们在一个控制台应用中分别编写了如上两段程序，继承自 `ValidatorAttribute` 自定义的 `ValidatorAttribute` 没有重写任何一个 `IsValid` 方法。当我们在 `Debug` 模式下分别运行这两段程序的时候，都会抛出如图 9-1 所示的 `NotImplementedException` 异常，提示“此类尚未实现 `IsValid(object value)`。首选入口点是 `GetValidationResult()`，并且类应重写 `IsValid(object value, ValidationContext context)`。”。

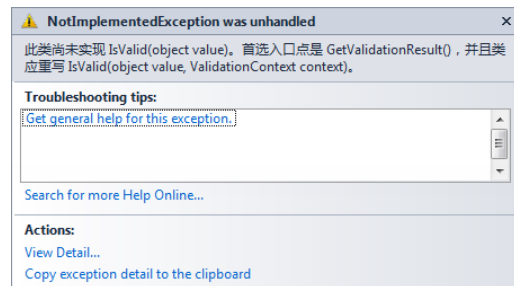


图 9-1 没有在自定义 `ValidatorAttribute` 中重写 `IsValid` 方法导致的异常

受保护的 `IsValid` 方法中除了包含一个表示被验证对象的参数 `value` 之外，还有一个类型为 `ValidationContext` 的参数。顾名思义，`ValidationContext` 旨在为当前的验证维护相应的上下文。如下面的代码片段所示，一个 `ValidationContext` 对象携带的验证上下文信息包括通过 `ObjectInstance` 和 `ObjectType` 属性表示的验证对象及其类型，以及通过 `MemberName` 和 `DisplayName` 属性表示的成员名称（一般指属性名称）和显示名称。

```

public sealed class ValidationContext
{
    //其他成员
    public ValidationContext(object instance);
    public ValidationContext(object instance,
        IDictionary<object, object> items);

    public string      DisplayName { get; set; }
    public string      MemberName { get; set; }
    public object      ObjectInstance { get; }
    public Type        ObjectType { get; }
}

```

受保护的 `IsValid` 方法返回值类型为具有如下定义的 `ValidationResult`。`ValidationResult` 与作

为 `ModelValidator` 验证结果的 `ModelValidationResult` 类型具有类似的定义, 它依然是错误消息和成员名称的组合。不过 `ModelValidationResult` 对应某个单一的成员名称, 而 `ValidationResult` 包含一组相关成员名称的列表。

```
public class ValidationResult
{
    //其他成员
    public ValidationResult(string errorMessage);
    public ValidationResult(string errorMessage,
        IEnumerable<string> memberNames);

    public string                ErrorMessage { get; set; }
    public IEnumerable<string>    MemberNames { get; }
}
```

`IsValid` 方法在验证失败的情况下会返回一个具体的 `ValidationResult` 对象, 如果指定的 `ValidationContext` 不为 `Null`, 那么其 `MemberName` 属性表示的成员名称将会包含在这个返回的 `ValidationResult` 对象的 `MemberNames` 列表中。`ValidationContext` 对象的 `DisplayName` 属性将会作为调用 `FormatErrorMessage` 的参数, 得到的格式化错误消息将会作为 `ValidationResult` 对象的 `ErrorMessage` 属性。如果成功通过验证, 该方法会直接返回 `Null`。

我们可以通过调用 `ValidationAttribute` 的方法 `GetValidationResult` 对指定的数据对象实施验证, 并得到以 `ValidationResult` 对象形式返回的验证结果, 得到的 `ValidationResult` 对象实际上就是调用受保护 `IsValid` 方法的返回值。我们也可以调用 `Validate` 方法直接验证某个指定的对象, 该方法在验证失败的情况下会直接抛出一个 `ValidationException` 异常, 通过调用 `FormatErrorMessage` 方法 (将参数 `name` 表示的字符串作为参数) 格式化后的错误消息将会作为该异常的消息。

9.1.2 几个常用的 ValidationAttribute

在 “`System.ComponentModel.DataAnnotations`” 命名空间下定义了一系列具体的验证特性, 它们大都直接应用在自定义数据类型的某个属性上对目标数据成员实施验证。这些预定义验证特性不是本章论述的重点, 所以在这里只是对它们作一个概括性的介绍。

- **RequiredAttribute:** 用于验证必需数据成员。
- **RangeAttribute:** 用于验证数据值是否在指定的范围之内。
- **StringLengthAttribute:** 用于验证字符串数据的长度是否在指定的范围之内。

- **MaxLengthAttribute/MinLengthAttribute:** 用于验证字符/数组数据长度是否小于/大于指定的上/下限。
- **RegularExpressionAttribute:** 用于验证字符串数据的格式是否与指定的正则表达式相匹配。
- **CompareAttribute:** 用于验证数据值是否与另一个成员一致，在用户注册场景中可以用于确认两次输入密码的一致性。
- **CustomValidationAttribute:** 指定一个用于验证目标成员的验证类型和验证方法。

除了上面这些常用的 **ValidationAttribute** 类型之外，还具有一个类型为 **DataTypeAttribute** 的验证特性帮助我们实现针对某种数据类型的验证，关于这个类型在本书第 4 章“Model 元数据的解析”中已经有过详细的介绍。**DataTypeAttribute** 还具有一系列子类，比如 **UrlAttribute**、**CreditCardAttribute**、**EmailAddressAttribute**、**EnumDataTypeAttribute**、**File-Extensions-Attribute** 和 **PhoneAttribute** 等，它们可以帮助我们实现针对某种具体数据类型（URL、信用卡号、电子邮箱地址、枚举成员、文件扩展名和电话号码）的验证。

在“**System.Web.Security**”命名空间下还具有一个类型为 **MembershipPasswordAttribute** 的验证特性。在进行用户注册时，我们可以利用它验证用户输入的密码是否符合预设的条件。我们可以利用 **MinRequiredPasswordLength** 和 **MinRequiredNonAlphanumericCharacters** 属性设置密码的最短长度和必须具有的非字母数字字符的个数，还可以通过设置属性 **PasswordStrengthRegularExpression** 指定一个正则表达式来控制密码的格式。它的额外 3 个属性（**MinNonAlphanumericCharactersError**、**MinPasswordLengthError** 和 **PasswordStrengthError**）分别代表相应的错误消息，如果我们通过其 **ResourceType** 指定了资源类型，那么这 3 个属性值将会作为相应资源项的名称。

```
[AttributeUsage(AttributeTargets.Parameter | AttributeTargets.Field |
    AttributeTargets.Property, AllowMultiple=false)]
public class MembershipPasswordAttribute : ValidationAttribute
{
    public override string FormatErrorMessage(string name);
    protected override ValidationResult IsValid(object value,
        ValidationContext validationContext);

    public int    MinRequiredNonAlphanumericCharacters { get; set; }
    public int    MinRequiredPasswordLength { get; set; }
    public string PasswordStrengthRegularExpression { get; set; }

    public string MinNonAlphanumericCharactersError { get; set; }
    public string MinPasswordLengthError { get; set; }
    public string PasswordStrengthError { get; set; }

    public Type   ResourceType { get; set; }
}
```

如果我们没有对 `MinRequiredPasswordLength`、`MinRequiredNonAlphanumericCharacters` 和 `PasswordStrengthRegularExpression` 属性作显式设置, 则它们的属性值来源于默认注册的 `MembershipProvider`。如下面的代码片段所示, `MembershipProvider` 类型具有对应的属性定义。

```
public abstract class MembershipProvider : ProviderBase
{
    //其他成员
    public abstract int MinRequiredNonAlphanumericCharacters { get; }
    public abstract int MinRequiredPasswordLength { get; }
    public abstract string PasswordStrengthRegularExpression { get; }
}
```

9.1.3 应用 `ValidationAttribute` 特性的唯一性

对于上面列出的这些预定义 `ValidationAttribute`, 它们都具有一个相同的特征, 那就是在同一个目标元素中只能应用一次, 这可以通过应用在它们上面的 `AttributeUsageAttribute` 特性的定义看出来。以如下所示的 `RequiredAttribute` 特性为例, 应用在该类型上的 `AttributeUsageAttribute` 特性的 `AllowMultiple` 属性被设置为 `False`。

```
[AttributeUsage(AttributeTargets.Parameter | AttributeTargets.Field |
    AttributeTargets.Property, AllowMultiple=false)]
public class RequiredAttribute : ValidationAttribute
{
    //省略成员
}
```

但这是否意味着如果我们在自定义 `ValidationAttribute` 的时候将 `AttributeUsageAttribute` 特性的 `AllowMultiple` 设置为 `True`, 它们就可以被多次应用到同一个属性或者类型上了呢? 我们不妨通过实例演示的方式来证实一下。

我们知道 `RangeAttribute` 特性可以帮助我们验证数据值的范围, 但是有时候我们需要进行“条件性范围验证”。举个例子, 我们现在对某个员工的薪水范围进行限定, 但是不同级别员工的薪水范围是不同的, 为此我们创建了一个名为 `RangeIfAttribute` 的验证特性帮助我们进行针对不同级别的薪水范围验证。如下面的代码片段所示, 我们将 3 个 `RangeIfAttribute` 特性应用到了表示薪水的 `Salary` 属性上, 分别针对 3 个级别 (G7、G8 和 G9) 的薪水范围作了相应的设定。

```
public class Employee
{
    public string Name { get; set; }
    public string Grade { get; set; }
```

```

[RangeIf("Grade", "G7", 2000, 3000)]
[RangeIf("Grade", "G8", 3000, 4000)]
[RangeIf("Grade", "G9", 4000, 5000)]
public decimal Salary { get; set; }
}

```

如下面的代码片段所示，`RangeIfAttribute` 直接继承自 `RangeAttribute`。`RangeIfAttribute` 根据被验证容器对象的另一个属性值来决定是否对当前属性实施验证，属性 `Property` 和 `Value` 就分别代表这个属性的名称和与之匹配的值。在重写的 `IsValid` 方法中，我们通过反射获取到了容器对象用于匹配的属性值，如果该值与 `Value` 属性值相匹配，则调用基类同名方法对指定对象进行验证，否则直接返回 `ValidationResult.Success (Null)`。应用在 `RangeIfAttribute` 上的 `AttributeUsageAttribute` 特性的 `AllowMultiple` 被设置为 `True`。

```

[AttributeUsage(AttributeTargets.Property, AllowMultiple = true)]
public class RangeIfAttribute: RangeAttribute
{
    public string Property { get; set; }
    public string Value { get; set; }

    public RangeIfAttribute(string property, string value, double minimum,
        double maximum) : base(minimum, maximum)
    {
        this.Property = property;
        this.Value = value ?? "";
    }

    protected override ValidationResult IsValid(object value,
        ValidationContext validationContext)
    {
        PropertyInfo property =
            validationContext.ObjectType.GetProperty(this.Property);
        object propertyValue =
            property.GetValue(validationContext.ObjectInstance, null);
        propertyValue = propertyValue ?? "";
        if (propertyValue.ToString() != this.Value)
        {
            return ValidationResult.Success;
        }
        return base.IsValid(value, validationContext);
    }
}

```

那么这样一个 `RangeIfAttribute` 特性真的能够按照我们期望的方式进行验证吗？为此我们在创建的空 ASP.NET MVC 应用中定义了如下一个 `HomeController`。我们在默认的动作方法 `Index` 中创建了用于描述 `Employee` 的 `Salary` 属性 `ModelMetadata` 对象，并通过调用其 `GetValidators` 方法得到用于验证该属性的 `ModelValidator` 列表，然后将这个 `ModelValidator` 列表

转化为数组作为 Model 呈现在对应的 View 中。

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        ModelMetadata employeeMetadata = ModelMetadataProviders.Current
            .GetMetadataForType(() => new Employee(), typeof(Employee));
        ModelMetadata salaryMetadata = employeeMetadata.Properties
            .FirstOrDefault(p => p.PropertyName == "Salary");
        IEnumerable<ModelValidator> validators = salaryMetadata
            .GetValidators(ControllerContext);
        return View(validators.ToArray());
    }
}
```

如下所示的是 Action 方法 Index 对应 View 的定义, 这是一个 Model 类型为 ModelValidator 数组的强类型 View。我们在该 View 中将所有 ModelValidator 对象的类型名称通过表格的形式呈现出来。

```
@model ModelValidator[]
<html>
<head>
    <title>ModelValidators</title>
</head>
<body>
    <table>
        @for(int i= 0; i<Model.Length; i++)
        {
            <tr><td>@(i+1)</td><td>@Model[i].GetType().Name</td></tr>
        }
    </table>
</body>
</html>
```

该程序运行之后会在浏览器中呈现出如图 9-2 所示的输出结果。由于 Employee 的 Salary 属性类型为非空值类型, 所以 ASP.NET MVC 会自动添加一个 RequiredAttributeAdapter 来进行必要性验证, 另一个用于数值验证的 NumericModelValidator 也是源于 Salary 属性的类型。只有第一个 DataAnnotationsModelValidator 是针对应用在 Salary 属性上的 RangeIfAttribute 特性而创建的, 换句话说, 应用在同一属性上的 3 个 RangeIfAttribute 特性只有一个是有效的, 具体原因何在呢? (S901)

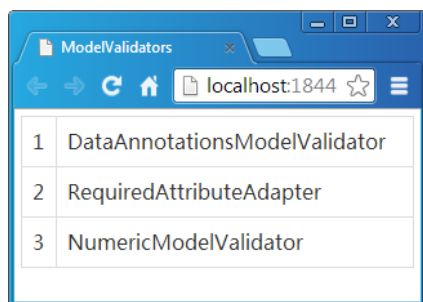


图 9-2 应用了多个同类验证特性的属性具有的 ModelValidator（默认）

我们知道 Attribute 具有一个 object 类型的 TypeId 属性，它默认返回代表自身类型的 Type 对象。ASP.NET MVC 在创建 DataAnnotationsModelValidator 对象的时候会根据该属性值对解析出来的所有特性进行分组，同一组只会选择一个特性。这就意味着对于多个应用到相同目标元素同类 ValidationAttribute，有且只有一个是有效的。

那么如何解决这个问题呢？其实很简单，既然 ASP.NET MVC 会根据 TypeId 属性对所有 ValidationAttribute 进行筛选，我们只需要通过重写 TypeId 属性使每个 ValidationAttribute 具有不同的属性值就可以了，为此我们按照如下方式在 RangeIfAttribute 中重写了 TypeId 属性。

```
[AttributeUsage( AttributeTargets.Field| AttributeTargets.Property,
    AllowMultiple = true)]
public class RangeIfAttribute: RangeAttribute
{
    //其他成员
    private object typeid;
    public override object TypeId
    {
        get{ return typeid?? (typeid= new object());}
    }
}
```

再次运行程序后将会在浏览器中得到如图 9-3 所示的输出结果，可以看到针对 3 个 RangeIfAttribute 特性的 3 个 DataAnnotationsModelValidator 被创建出来了。顺便说一下，通过重写 TypeId 属性使多个应用到相同元素上的同类 ValidationAttribute 生效的解决方案并不适合客户端验证，因为这会导致多组相同的验证规则被生成。（S902）

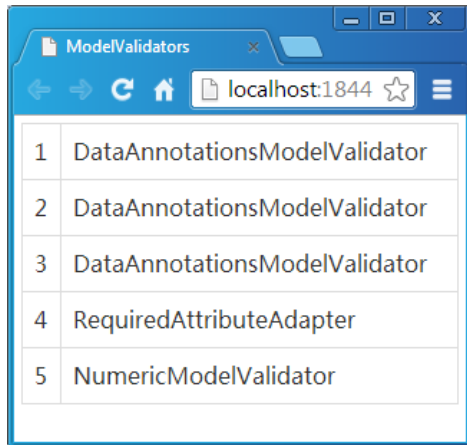


图 9-3 应用了多个同类验证特性的属性具有的 ModelValidator（重写 TypedId 属性）

9.2 DataAnnotationsModelValidator 及其提供策略

ModelValidator 是最终对绑定数据对象实施 Model 验证的对象，应用在数据类型或其属性成员上的 ValidationAttribute 特性最终会转换成相应的 ModelValidator 参与到针对目标数据的验证中。这个 ModelValidator 类型就是具有如下定义的 DataAnnotationsModelValidator，它的只读属性 Attribute 返回的就是对应的 ValidationAttribute 对象。

```
public class DataAnnotationsModelValidator : ModelValidator
{
    public DataAnnotationsModelValidator(ModelMetadata metadata,
        ControllerContext context, ValidationAttribute attribute);
    public override IEnumerable<ModelClientValidationRule>
        GetClientValidationRules();

    protected ValidationAttribute      Attribute { get; }
    protected string                   ErrorMessage { get; }
    public override bool                IsRequired { get; }

    public override IEnumerable<ModelValidationResult> Validate(
        object container) ;
}
```

下面的代码片段给出了用于实施验证的核心方法 Validate 的完整定义。该方法首先针对被验证容器对象创建出作为验证上下文的 ValidationContext 对象，并采用 ModelMetadata 的 DisplayName 属性作为该 ValidationContext 的同名属性值。真正的验证工作通过调用 ValidationAttribute 的 GetValidationResult 方法来完成，如果该方法返回值不为 Null

(ValidationResult.Success)，则将返回的 ValidationResult 转换成 ModelValidationResult 对象并添加到最终返回的 ModelValidationResult 集合中。

```
public class DataAnnotationsModelValidator : ModelValidator
{
    //其他成员
    public override IEnumerable<ModelValidationResult> Validate(
        object container)
    {
        ValidationContext validationContext = new ValidationContext(
            container ?? this.Metadata.Model, null, null)
        {
            DisplayName = this.Metadata.GetDisplayName()
        };
        ValidationResult validationResult =
            this.Attribute.GetValidationResult(
                this.Metadata.Model, validationContext);
        if (validationResult != ValidationResult.Success)
        {
            ModelValidationResult iteratorVariable2 =
                new ModelValidationResult
                {
                    Message = validationResult.ErrorMessage
                };
            yield return iteratorVariable2;
        }
        else
        {
            yield break;
        }
    }
}
```

顺便再说说定义在 DataAnnotationsModelValidator 中的另外两个受保护只读属性的逻辑。用于返回错误消息的 ErrorMessage 属性来源于调用验证特性的 FormatErrorMessage 方法的返回值，指定的参数就是当前 ModelMetadata 对象的 DisplayName 属性值。由于只有 RequiredAttribute 特性才会对被验证数据实施必要性验证，所以只有被封装的 ValidationAttribute 为 RequiredAttribute 时，对应 DataAnnotationsModelValidator 的 IsRequired 属性才返回 True。

9.2.1 “适配”型 DataAnnotationsModelValidator

ValidationAttribute 特性本身具有数据验证功能，并且 DataAnnotationsModelValidator 直接利用它实现服务端验证。如果被封装的 ValidationAttribute 没有实现 IClientValidatable 接口（实际上定义在“System.ComponentModel.DataAnnotations”命名空间下的 ValidationAttribute 类型

都没有实现这个接口), 对应 `DataAnnotationsModelValidator` 就不能提供客户端验证的支持。

出于客户端验证的需要, ASP.NET MVC 为一些 `ValidationAttribute` 提供了“适配”型的 `DataAnnotationsModelValidator`。这些作为适配器的 `DataAnnotationsModelValidator` 具有一个共同的特征, 那就是它们都通过重写的 `GetClientValidationRules` 方法返回相应的客户端验证规则。

1 . `DataTypeAttributeAdapter`

顾名思义, `DataTypeAttributeAdapter` 是为了实现针对某种数据类型的客户端验证而为 `DataTypeAttribute` 定义的适配器。如下面的代码片段所示, `DataTypeAttributeAdapter` 直接继承自 `DataAnnotationsModelValidator`。它的 `RuleName` 属性表示客户端验证规则名称, 可以在构造函数中通过参数 `ruleName` 指定, 重写的 `GetClientValidationRules` 方法根据此名称和指定的验证错误消息提供相应的验证规则。

```
internal class DataTypeAttributeAdapter : DataAnnotationsModelValidator
{
    public DataTypeAttributeAdapter(ModelMetadata metadata,
        ControllerContext context, DataTypeAttribute attribute,
        string ruleName);
    public override IEnumerable<ModelClientValidationRule>
        GetClientValidationRules();

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

2 . `DataAnnotationsModelValidator<TAttribute>`

ASP.NET MVC 为针对其他 `ValidationAttribute` 的适配器定义了一个具有如下定义的泛型基类 `DataAnnotationsModelValidator<TAttribute>`。它是 `DataAnnotationsModelValidator` 的子类, 泛型参数表示被封装的 `ValidationAttribute` 的类型。

```
public class DataAnnotationsModelValidator<TAttribute> :
    DataAnnotationsModelValidator where TAttribute: ValidationAttribute
{
    public DataAnnotationsModelValidator(ModelMetadata metadata,
        ModelBindingExecutionContext context, TAttribute attribute);
    protected TAttribute Attribute { get; }
}
```

ASP.NET MVC 为 4 个常用的验证特性 (`RequiredAttribute`、`RangeAttribute`、`RegularExpressionAttribute` 和 `StringLengthAttribute`) 定义了相应的适配器。如下面的代码片段所示, 它们都是泛型 `DataAnnotationsModelValidator<TAttribute>` 的子类, 并且通过重写的 `GetClientValidationRules` 方

法提供对应的客户端验证规则。

```

public class RequiredAttributeAdapter :
    DataAnnotationsModelValidator<RequiredAttribute>
{
    public RequiredAttributeAdapter(ModelMetadata metadata,
        ControllerContext context, RequiredAttribute attribute);
    public override IEnumerable<ModelClientValidationRule>
        GetClientValidationRules();
}

public class RangeAttributeAdapter :
    DataAnnotationsModelValidator<RangeAttribute>
{
    public RangeAttributeAdapter(ModelMetadata metadata,
        ControllerContext context, RangeAttribute attribute);
    public override IEnumerable<ModelClientValidationRule>
        GetClientValidationRules();
}

public class RegularExpressionAttributeAdapter :
    DataAnnotationsModelValidator<RegularExpressionAttribute>
{
    public RegularExpressionAttributeAdapter(ModelMetadata metadata,
        ControllerContext context, RegularExpressionAttribute attribute);
    public override IEnumerable<ModelClientValidationRule>
        GetClientValidationRules();
}

public class StringLengthAttributeAdapter :
    DataAnnotationsModelValidator<StringLengthAttribute>
{
    public StringLengthAttributeAdapter(ModelMetadata metadata,
        ControllerContext context, StringLengthAttribute attribute);
    public override IEnumerable<ModelClientValidationRule>
        GetClientValidationRules();
}

```

上面这 4 个具体的 `DataAnnotationsModelValidator<TAttribute>` 类型均是公有类型，除此之外，ASP.NET MVC 还为其 3 种 `ValidationAttribute` 类型（`CompareAttribute`、`FileExtensionsAttribute` 和 `MembershipPasswordAttribute`）定义了对应的适配器，它们的类型分别是 `CompareAttributeAdapter`、`FileExtensionsAttributeAdapter` 和 `MembershipPasswordAttributeAdapter`。如下面的代码片段所示，它们均是内部类型。

```

internal class CompareAttributeAdapter :
    DataAnnotationsModelValidator<CompareAttribute>
{
    public CompareAttributeAdapter(ModelMetadata metadata,

```

```

        ControllerContext context, CompareAttribute attribute);
    public override IEnumerable<ModelClientValidationRule>
        GetClientValidationRules();
}

internal class FileExtensionsAttributeAdapter :
    DataAnnotationsModelValidator<FileExtensionsAttribute>
{
    public FileExtensionsAttributeAdapter(ModelMetadata metadata,
        ControllerContext context, FileExtensionsAttribute attribute);
    public override IEnumerable<ModelClientValidationRule>
        GetClientValidationRules();
}

internal class MembershipPasswordAttributeAdapter :
    DataAnnotationsModelValidator<MembershipPasswordAttribute>
{
    public MembershipPasswordAttributeAdapter(ModelMetadata metadata,
        ControllerContext context, MembershipPasswordAttribute attribute);
    public override IEnumerable<ModelClientValidationRule>
        GetClientValidationRules();
}

```

9.2.2 DataAnnotationsModelValidatorProvider

`DataAnnotationsModelValidator` 最终是通过对应的 `DataAnnotationsModelValidatorProvider` 创建的, 后者是 `AssociatedValidatorProvider` 的子类。类型名称前缀“Associated”表示与数据类型“关联”的特性, 所以 `AssociatedValidatorProvider` 实际上提供了一种根据应用在数据类型或者数据成员上的特性来创建 `ModelValidator` 的方式。

1. AssociatedValidatorProvider

如下面的代码片段所示, `AssociatedValidatorProvider` 是一个继承自 `ModelValidatorProvider` 的抽象类, 它定义了一个抽象的受保护方法 `GetValidators` 根据提供的特性列表来获取验证目标数据对象或其属性成员的 `ModelValidator` 列表。除了作为参数的特性列表之外, 该方法还具有额外两个参数, 它们分别代表描述被验证数据类型或者属性成员的 `ModelMetadata` 和当前 `ControllerContext`。

```

public abstract class AssociatedValidatorProvider : ModelValidatorProvider
{
    //其他成员
    public sealed override IEnumerable<ModelValidator>

```

```

        GetValidators(ModelMetadata metadata, ControllerContext context);

protected abstract IEnumerable<ModelValidator>
    GetValidators(ModelMetadata metadata, ControllerContext context,
        IEnumerable<Attribute> attributes);
}

```

AssociatedValidatorProvider 实现了定义在 ModelValidatorProvider 中的抽象方法 GetValidators (公有)。当该方法被执行的时候, 如果提供的 ModelMetadata 对象是对一个根容器类型的描述, 那么应用在数据类型上的特性会被提取出来。如果 ModelMetadata 对象是对容器类型某个属性成员的描述, 那么应用在数据类型和属性上的特性均会被提取出来。这些通过反射方式提取出来的特性列表会作为参数调用受保护的 GetValidators 方法, 方法最终返回的是一组 ModelValidator 列表。

2 . DataAnnotationsModelValidatorProvider 的 ModelValidator 提供策略

作为 AssociatedValidatorProvider 的子类, DataAnnotationsModelValidatorProvider 只需要在实现的抽象方法 GetValidators 中从提供的特性列表中筛选出继承自 ValidationAttribute 的特性并创建对应的 DataAnnotationsModelValidator 就可以了。接下来我们根据其类型定义来讨论一下实现在 DataAnnotationsModelValidatorProvider 中具体的 ModelValidator 提供机制。

首先来认识一下定义在 DataAnnotationsModelValidatorProvider 之中的如下两组静态字段, 第一组适用于根据 ValidationAttribute 特性来创建 ModelValidator, 另一组则适用于根据实现了 IValidatableObject 接口的数据类型来创建 ModelValidator。

```

public class DataAnnotationsModelValidatorProvider :
    AssociatedValidatorProvider
{
    //其他成员
    internal static Dictionary<Type, DataAnnotationsModelValidationFactory>
        AttributeFactories;
    internal static DataAnnotationsModelValidationFactory
        DefaultAttributeFactory;

    internal static DataAnnotationsValidatableObjectAdapterFactory
        DefaultValidatableFactory;
    internal static
        Dictionary<Type, DataAnnotationsValidatableObjectAdapterFactory>
        ValidatableFactories;
}

public delegate ModelValidator DataAnnotationsModelValidationFactory(
    ModelMetadata metadata, ControllerContext context,
    ValidationAttribute attribute);
public delegate ModelValidator

```

```
DataAnnotationsValidatableObjectAdapterFactory(ModelMetadata metadata,
ControllerContext context);
```

具体来说, 字典类型的字段 `AttributeFactories` 和 `ValidatableFactories` 分别维护着一组 `ValidationAttribute` 类型或者实现了 `IValidatableObject` 接口的数据类型与创建 `ModelValidator` 的委托对象之间的映射关系。如果给定的 `ValidationAttribute` 类型或者数据类型没有在这个映射关系中, 通过字段 `DefaultAttributeFactory` 和 `DefaultValidatableFactory` 表示的委托对象将用来创建对应的 `ModelValidator`。

当 `DataAnnotationsModelValidatorProvider` 类型被加载的时候, 上述这 4 个静态字段会被初始化。如下的代码片段反映了采用的默认 `ModelValidator` 提供策略: 如果被验证的数据类型或者属性成员上应用了 `ValidationAttribute` 特性, 则 `DataAnnotationsModelValidatorProvider` 会根据此特性创建一个 `DataAnnotationsModelValidator` 对象。如果被验证的数据类型实现了 `IValidatableObject` 接口, 则默认创建的 `ModelValidator` 是一个 `ValidatableObjectAdapter` 对象。

```
public class DataAnnotationsModelValidatorProvider :
    AssociatedValidatorProvider
{
    static DataAnnotationsModelValidatorProvider()
    {
        //其他操作
        DefaultAttributeFactory = (metadata, context, attribute) =>
            new DataAnnotationsModelValidator(metadata, context, attribute);
        DefaultValidatableFactory = (metadata, context) =>
            new ValidatableObjectAdapter(metadata, context);
        AttributeFactories = BuildAttributeFactoriesDictionary();
        ValidatableFactories = new Dictionary<Type,
            DataAnnotationsValidatableObjectAdapterFactory>();
    }
}
```

`DataAnnotationsModelValidatorProvider` 的静态字段 `AttributeFactories` 体现了它针对某些特殊 `ValidationAttribute` 类型采用的 `ModelValidator` 提供策略。在静态构造函数被执行的时候, `BuildAttributeFactoriesDictionary` 方法会被调用来初始化该字段。具体来说, 它会为如表 9-1 所示的 11 种类型的 `ValidationAttribute` 注册相应的创建 `ModelValidator` 的委托对象, 该表还列出了委托对象最终创建的 `ModelValidator` 类型。

表 9-1 11 种 ValidationAttribute 和对应的适配型 ModelValidator

ValidationAttribute	ModelValidator
RangeAttribute	RangeAttributeAdapter
RegularExpressionAttribute	RegularExpressionAttributeAdapter
RequiredAttribute	RequiredAttributeAdapter
StringLengthAttribute	StringLengthAttributeAdapter
MembershipPasswordAttribute	MembershipPasswordAttributeAdapter
CompareAttribute	CompareAttributeAdapter
FileExtensionsAttribute	FileExtensionsAttributeAdapter
CreditCardAttribute	DataTypeAttributeAdapter (RuleName="creditcard")
EmailAddressAttribute	DataTypeAttributeAdapter (RuleName="email")
PhoneAttribute	DataTypeAttributeAdapter (RuleName="phone")
UrlAttribute	DataTypeAttributeAdapter (RuleName="url")

上面介绍的这 4 个静态字段体现了 DataAnnotationsModelValidatorProvider 采用的 ModelValidator 提供策略，即针对具体某种类型的 ValidationAttribute 或者实现了 IValidatableObject 接口的数据类型应该创建怎样的 ModelValidator。如果默认的提供策略不能满足我们的需求，还可以调用如下 8 个静态方法对其进行任意定制。

```

public class DataAnnotationsModelValidatorProvider :
    AssociatedValidatorProvider
{
    //其他成员
    public static void RegisterAdapter(Type attributeType, Type adapterType);
    public static void RegisterAdapterFactory(Type attributeType,
        DataAnnotationsModelValidationFactory factory);
    public static void RegisterDefaultAdapter(Type adapterType);
    public static void RegisterDefaultAdapterFactory(
        DataAnnotationsModelValidationFactory factory);

    public static void RegisterValidatableObjectAdapter(Type modelType,
        Type adapterType);
    public static void RegisterValidatableObjectAdapterFactory(Type modelType
        ,DataAnnotationsValidatableObjectAdapterFactory factory);
    public static void RegisterDefaultValidatableObjectAdapter(
        Type adapterType);
    public static void RegisterDefaultValidatableObjectAdapterFactory(
        DataAnnotationsValidatableObjectAdapterFactory factory);
}

```

接下来我们来具体地谈谈 `DataAnnotationsModelValidatorProvider` 在实现的 `GetValidators` 方法中是如何根据提供的 `ModelMetadata` 和特性列表来提供 `ModelValidator` 的。当该方法被执行的时候, 所有 `ValidationAttribute` 特性会被从提供的特性列表中提取出来。针对每个具体的 `ValidationAttribute` 特性, 如果能够在 `AttributeFactories` 字段中找到对应的委托对象, 那么该委托对象会被用来创建相应的 `ModelValidator`, 否则用于创建 `ModelValidator` 的是通过静态字段 `DefaultAttributeFactory` 表示的委托对象。

```
public class DataAnnotationsModelValidatorProvider :
    AssociatedValidatorProvider
{
    //其他成员
    protected override IEnumerable<ModelValidator> GetValidators(
        ModelMetadata metadata, ControllerContext context,
        IEnumerable<Attribute> attributes);
}
```

如果被验证的数据类型实现了 `IValidatableObject` 接口, 并且能够从静态字段 `ValidatableFactories` 找到对应的委托对象, 那么该委托对象会被用来创建对应的 `ModelValidator`, 否则对应的 `ModelValidator` 将会由 `DefaultValidatableFactory` 字段表示的委托来创建。

如果绑定的目标数据为值类型, 在请求未曾提供相应数据的情况下, 默认值会作为它们最终的绑定值。但是这样的绑定行为带来一个问题, 即无法区分具有默认值的绑定数据是否真正由当前请求提供。在大部分情况下, 不论值类型的属性上是否应用了 `RequiredAttribute` 特性, ASP.NET MVC 都将其视为一个“必需”的数据成员, 并会为之创建一个 `RequiredAttributeAdapter` 对象作为对应的 `ModelValidator`。

如下面的代码片段所示, `DataAnnotationsModelValidatorProvider` 具有一个布尔类型的属性 `AddImplicitRequiredAttributeForValueTypes`, 表示是否需要为值类型数据成员显式添加一个 `RequiredAttribute` 特性。该属性默认返回 `True`, 意味着值类型的数据成员会默认视为必需的。

```
public class DataAnnotationsModelValidatorProvider :
    AssociatedValidatorProvider
{
    //其他成员
    public static bool AddImplicitRequiredAttributeForValueTypes { get; set; }
}
```

3. 实例演示: 解析针对不同属性成员创建的 `ModelValidator` (S903)

上面我们对实现在 `DataAnnotationsModelValidatorProvider` 中的 `ModelValidator` 的提供策略

进行了详细介绍，为了让读者对此有更加深刻的认识，我们来作一个简单的实例演示。我们在一个 ASP.NET MVC 应用中定义了如下一个数据类型 DemoModel。

```
public class DemoModel
{
    //第 1 组
    [Foobar]
    public object Foobar { get; set; }

    //第 2 组
    [Range(1,10)]
    public object Range { get; set; }

    [RegularExpression("...")]
    public object RegularExpression { get; set; }

    [Required]
    public object Required { get; set; }

    [StringLength(10)]
    public object StringLength { get; set; }

    [MembershipPassword]
    public object MembershipPassword { get; set; }

    [Compare("Foobar")]
    public object CompareAttribute { get; set; }

    [FileExtensions(Extensions = ".xml,.doc")]
    public object FileExtensions { get; set; }

    [CreditCard]
    public object CreditCard { get; set; }

    [EmailAddress]
    public object EmailAddress { get; set; }

    [Phone]
    public object Phone { get; set; }

    [Url]
    public object Url { get; set; }

    //第 3 组
    public ValidatableObject ValidatableObject { get; set; }
```

```

        //第4组
        public int ValueType { get; set; }
    }

    public class ValidatableObject : IValidatableObject
    {
        public IEnumerable<ValidationResult> Validate(
            ValidationContext validationContext)
        {
            throw new NotImplementedException();
        }
    }

    public class FoobarAttribute : ValidationAttribute{}

```

我们将定义在 DemoModel 中的若干属性成员划分为 4 组。包含在第一组的属性 Foobar 上应用了一个自定义的 FoobarAttribute 特性,第二组的属性上分别应用了表 9-1 中列出的 11 种类型的 ValidationAttribute,隶属于第三组的属性 ValidatableObject 的返回类型实现了 IValidatableObject 接口,划分为第四组的属性 ValueType 返回一个整数。

我们定义了如下一个 HomeController。在默认的动作方法 Index 中,我们解析出描述 DemoModel 所有属性的 ModelMetadata,并利用 DataAnnotationsModelValidatorProvider 根据它们的搭配验证对应属性成员采用的 ModelValidator 列表。我们将属性的名称和采用的 ModelValidator 列表组成一个 Dictionary<string, ModelValidator[]>对象,并作为 Model 呈现在对应的 View 中。

```

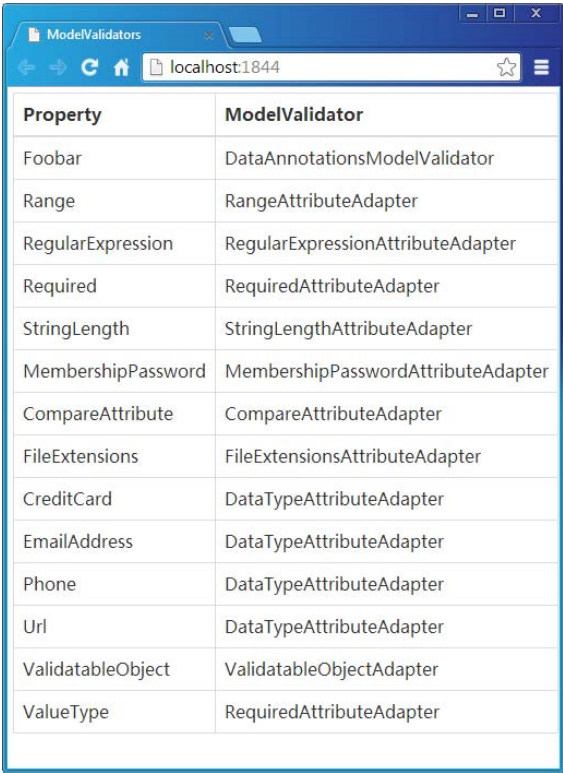
public class HomeController : Controller
{
    public ActionResult Index()
    {
        Dictionary<string, ModelValidator[]> modelValidators =
            new Dictionary<string, ModelValidator[]>();
        DataAnnotationsModelValidatorProvider validatorProvider =
            new DataAnnotationsModelValidatorProvider();
        foreach (ModelMetadata metadata in ModelMetadataProviders.Current
            .GetMetadataForProperties(new DemoModel(), typeof(DemoModel)))
        {
            modelValidators.Add(metadata.PropertyName,
                validatorProvider.GetValidators(metadata,
                    this.ControllerContext).ToArray());
        }
        return View(modelValidators);
    }
}

```


如下所示的是 Action 方法 Index 对应 View 的定义,这是一个 Model 类型为 IDictionary<string, ModelValidator[]> 的强类型 View。我们在该 View 中将属性的名称和采用的 ModelValidator 的类型以表格的形式呈现出来。

```
@model IDictionary<string, ModelValidator[]>
<html>
  <head>
    <title>ModelValidators</title>
  </head>
  <body>
    <table>
      <thead>
        <tr><th>Property</th><th>ModelValidator</th></tr>
      </thead>
      <tbody>
        @foreach(var item in Model)
        {
          <tr>
            <td rowspan="@item.Value.Length">@item.Key</td>
            @foreach (ModelValidator validator in item.Value)
            {
              <td>@validator.GetType().Name</td>
            }
          </tr>
        }
      </tbody>
    </table>
  </body>
</html>
```

运行该程序之后会在浏览器中得到如图 9-4 所示的输出结果,可以看到验证定义在 DemoModel 类型中各属性所采用的 ModelValidator 与 DataAnnotationsModelValidatorProvider 采用的 ModelValidator 提供策略是完全一致的。



The screenshot shows a web browser window with the title 'ModelValidators' and the address bar displaying 'localhost:1844'. The main content area contains a table with two columns: 'Property' and 'ModelValidator'. The table lists various properties and the specific validators used for each.

Property	ModelValidator
FooBar	DataAnnotationsModelValidator
Range	RangeAttributeAdapter
RegularExpression	RegularExpressionAttributeAdapter
Required	RequiredAttributeAdapter
StringLength	StringLengthAttributeAdapter
MembershipPassword	MembershipPasswordAttributeAdapter
CompareAttribute	CompareAttributeAdapter
FileExtensions	FileExtensionsAttributeAdapter
CreditCard	DataTypeAttributeAdapter
EmailAddress	DataTypeAttributeAdapter
Phone	DataTypeAttributeAdapter
Url	DataTypeAttributeAdapter
ValidatableObject	ValidatableObjectAdapter
ValueType	RequiredAttributeAdapter

图 9-4 验证 DemoModel 各个属性所采用的 ModelValidator

9.2.3 将 ValidationAttribute 特性应用到参数上

如果足够细心应该会发现我们常用的 ValidationAttribute 特性都可以直接应用到方法的参数上。以如下定义的 RangeAttribute 为例，应用在该类型上的 AttributeUsageAttribute 表明该特性可以直接标注到方法的参数和数据类型的字段及属性成员上。

```
[AttributeUsage(
    AttributeTargets.Parameter |
    AttributeTargets.Field |
    AttributeTargets.Property,
    AllowMultiple=false)]
public class RangeAttribute : ValidationAttribute
{
    //省略成员
}
```

但是对于 ASP.NET MVC 的 Model 验证来说,应用在 Action 方法参数上的 ValidationAttribute 特性起不到任何作用,因为用于进行 Model 验证的 ModelValidator 对象是通过描述参数类型的 ModelMetadata 创建的,它根本不会去解析应用在参数本身上的 ValidationAttribute 特性。

但在笔者看来直接针对 Action 方法参数来定义验证规则具有很高的实用价值。一方面,目前的 Model 验证仅限于针对容器对象本身及其属性的验证,如果目标 Action 方法的参数为 String、Int32 和 Double 等简单类型,则针对它们的验证只能通过手工的方式来完成。另一方面,具有相同参数类型的多个 Action 方法中可能具有不同的验证规则。如果我们能够将 ValidationAttribute 特性应用在参数上进行针对性的验证规则定义,这两个问题都将迎刃而解。

到目前为止,我们对 ASP.NET MVC 的 Model 验证系统已经有了一个全面的了解,现在我们试着通过对应的扩展使直接应用到参数上的 ValidationAttribute 特性能够生效。我们需要自定义一个 ModelValidatorProvider 来解析应用到参数上的验证特性,并据此生成对应的 ModelValidator。但在这之前需要解决的另一个问题是如何将应用于参数的特性提供给我们自定义的 ModelValidatorProvider,在这里我们将当前的 ControllerContext 作为载体。

Action 方法的执行是通过 ActionInvoker 来实现的,默认的 ControllerActionInvoker 和 AsyncControllerActionInvoker 都定义了一个受保护的虚方法 GetParameterValue,它会根据描述参数的 ParameterDescriptor 对象和当前的 ControllerContext 实施 Model 绑定得到对应的参数值。我们可以通过继承 ControllerActionInvoker/AsyncControllerActionInvoker 以重写该方法的方式将 ParameterDescriptor 保存到当前的 ControllerContext 中。

为此我们自定义了如下两个 ActionInvoker 类型,其中 ParameterValidationActionInvoker 继承自 ControllerActionInvoker,而 ParameterValidationAsyncActionInvoker 是 AsyncControllerActionInvoker 的子类。在重写的 GetParameterValue 方法中,我们在调用基类的同名方法之前将描述参数的 ParameterDescriptor 对象保存到当前 ControllerContext 中,具体来说放到了表示当前路由数据的 RouteDataDictionary 对象的数据 Tokens 集合中。在方法调用之后我们将其从 ControllerContext 中移除。

```
public class ParameterValidationActionInvoker : ControllerActionInvoker
{
    protected override object GetParameterValue(
        ControllerContext controllerContext,
        ParameterDescriptor parameterDescriptor)
    {
        try
        {
            controllerContext.RouteData.DataTokens.Add(
                typeof(ParameterDescriptor).FullName, parameterDescriptor);
            return base.GetParameterValue(controllerContext,
```

```

        parameterDescriptor);
    }
    finally
    {
        controllerContext.RouteData.DataTokens.Remove(
            typeof(ParameterDescriptor).FullName);
    }
}

public class ParameterValidationAsyncActionInvoker :
    AsyncControllerActionInvoker
{
    protected override object GetParameterValue(ControllerContext
        controllerContext, ParameterDescriptor parameterDescriptor)
    {
        try
        {
            controllerContext.RouteData.DataTokens.Add(
                typeof(ParameterDescriptor).FullName, parameterDescriptor);
            return base.GetParameterValue(controllerContext,
                parameterDescriptor);
        }
        finally
        {
            controllerContext.RouteData.DataTokens.Remove(
                typeof(ParameterDescriptor).FullName);
        }
    }
}

```

被 `ParameterValidationActionInvoker` 和 `ParameterValidationAsyncActionInvoker` 存放到当前 `ControllerContext` 中的 `ParameterDescriptor` 被自定义的 `ModelValidatorProvider` 提取出来用于创建相应的 `ModelValidator`。如下面的代码所示, 自定义的 `ParameterValidationModelValidatorProvider` 直接继承自 `DataAnnotationsModelValidatorProvider`, 我们在重写的 `GetValidators` 方法中将 `ParameterDescriptor` 从 `ControllerContext` 中提取出来, 并利用它得到应用在参数上的所有特性并与当前的特性列表进行合并, 最后将合并的特性列表作为参数调用基类的 `GetValidators` 方法。

```

public class ParameterValidationModelValidatorProvider :
    DataAnnotationsModelValidatorProvider
{
    protected override IEnumerable<ModelValidator> GetValidators(
        ModelMetadata metadata, ControllerContext context,
        IEnumerable<Attribute> attributes)
    {
        object descriptor;
        if (metadata.ContainerType == null && context.RouteData.DataTokens
            .TryGetValue(typeof(ParameterDescriptor).FullName,
                out descriptor))
        {

```

```

        ParameterDescriptor parameterDescriptor =
            (ParameterDescriptor)descriptor;
        DisplayAttribute displayAttribute = parameterDescriptor
            .GetCustomAttributes(true).OfType<DisplayAttribute>()
            .FirstOrDefault()
            ?? new DisplayAttribute
            { Name = parameterDescriptor.ParameterName };
        metadata.DisplayName = displayAttribute.Name;
        var addedAttributes =
            parameterDescriptor.GetCustomAttributes(true)
            .OfType<Attribute>();
        return base.GetValidators(metadata, context,
            attributes.Union(addedAttributes));
    }
    else
    {
        return base.GetValidators(metadata, context, attributes);
    }
}
}

```

值得一提的是，应用在参数上的特性是针对最外层的容器类型，而不是针对容器类型的属性。比如我们在类型为 `Contact` 的参数上应用一个 `ValidationAttribute` 特性，该特性应该与应用在 `Contact` 类型上的特性具有相同的效果，但是与 `Address` 属性无关。所以 `ParameterDescriptor` 的提取及特性的合并仅仅在当前 `ModelMetadata` 的 `ContainerType` 属性为 `Null` 的情况下才会进行。除此之外，我们还利用标注在参数上的 `DisplayAttribute` 特性对 `ModelMetadata` 的 `DisplayName` 属性作了相应的设置。

在默认情况下只有在针对复杂类型的 `Model` 绑定过程中才会进行 `Model` 验证。虽然我们通过 `ParameterValidationModelValidatorProvider` 能够根据应用在 `Action` 方法参数上的验证特性生成相应的 `ModelValidator`，但是如果 `ValidationAttribute` 特性是应用在一个简单类型的参数上，对应的 `ModelValidator` 也是不会真正用于参数验证的。为了使 `Model` 验证发生在针对简单类型的 `Model` 绑定过程中，我们不得不创建一个自定义的 `ModelBinder`。

我们定义了一个具有如下定义的 `ParameterValidationModelBinder`，它直接继承自默认使用的 `DefaultModelBinder`，针对简单类型的 `Model` 验证定义在重写的 `BindModel` 方法中。

```

public class ParameterValidationModelBinder : DefaultModelBinder
{
    public override object BindModel(ControllerContext controllerContext,
        ModelBindingContext bindingContext)
    {
        object model = bindingContext.ModelMetadata.Model =
            base.BindModel(controllerContext, bindingContext);
        ModelMetadata metadata = bindingContext.ModelMetadata;
        if (metadata.IsComplexType || null == model)
        {

```

```

        return model;
    }

    //针对简单类型的 Model 验证
    Dictionary<string, bool> dictionary =
        new Dictionary<string, bool>(StringComparer.OrdinalIgnoreCase);
    foreach (ModelValidationResult result in ModelValidator
        .GetModelValidator(metadata, controllerContext).Validate(null))
    {
        string key = bindingContext.ModelName;
        if (!dictionary.ContainsKey(key))
        {
            dictionary[key] = bindingContext.ModelState.IsValidField(key);
        }
        if (dictionary[key])
        {
            bindingContext.ModelState.AddModelError(key, result.Message);
        }
    }
    return model;
}
}

```

到此为止, 为了能够将验证特性直接应用于 Action 方法的参数, 我们创建了自定义的 ActionInvoker、ModelValidatorProvider 和 ModelBinder。为了验证它们是否能够最终实现期望的验证效果, 我们将它们应用到一个简单的 ASP.NET MVC 应用中。我们在一个 ASP.NET MVC 应用中创建了一个具有如下定义的 HomeController, 在重写的 CreateActionInvoker 方法中, 如果调用基类同名方法返回的 ActionInvoker 类型为 ControllerActionInvoker, 那么该方法将返回一个 ParameterValidationActionInvoker 对象, 否则返回一个 ParameterValidationAsyncActionInvoker 对象, 这样做的目的是与默认的同步/异步 Action 执行方式保持一致。

```

public class HomeController : Controller
{
    protected override IActionInvoker CreateActionInvoker()
    {
        IActionInvoker actionInvoker = base.CreateActionInvoker();
        if (actionInvoker is ControllerActionInvoker)
        {
            return new ParameterValidationActionInvoker();
        }
        else
        {
            return new ParameterValidationAsyncActionInvoker();
        }
    }

    public ActionResult Add(
        [Display(Name = "第 1 个操作数")]

```

```

        [Range(10, 20, ErrorMessage = "{0}必须在{1}和{2}之间!")]
        [ModelBinder(typeof(ParameterValidationModelBinder))]
        double operand1,

        [DisplayName(Name = "第 2 个操作数")]
        [Range(10, 20, ErrorMessage = "{0}必须在{1}和{2}之间!")]
        [ModelBinder(typeof(ParameterValidationModelBinder))]
        double operand2)
    {
        double result = 0.00;
        if (ModelState.IsValid)
        {
            result = operand1 + operand2;
        }
        return View(new OperationData
        {
            Operand1 = operand1,
            Operand2 = operand2,
            Operator = "Add",
            Result = result
        });
    }
}

public class OperationData
{
    [DisplayName("操作数 1")]
    public double Operand1 { get; set; }

    [DisplayName("操作数 2")]
    public double Operand2 { get; set; }

    [DisplayName("操作符")]
    public string Operator { get; set; }

    [DisplayName("运算结果")]
    public double Result { get; set; }
}

```

我们在 HomeController 中定义了一个进行加法运算的 Action 方法 Add, 传入的参数代表两个操作数。我们在两个参数上应用了 3 个特性, 其中 DisplayAttribute 特性用于设置显示名称, RangeAttribute 特性用于限制数值的范围 (10~20), 而 ModelBinderAttribute 特性则是为了让针对这两个参数的绑定采用我们自定义的 ParameterValidationModelBinder 来完成。我们在 Add 方法中会进行相应的运算, 并将相关的信息封装到一个 OperationData 对象, 此对象将作为 Model 呈现在对应的 View 中。

如下所示的是 Action 方法 Add 对应 View 的定义, 这是一个 Model 类型为 OperationData 的强类型 View。我们在该 View 中直接调用 HtmlHelper<TModel>的扩展方法 EditorForModel 将作为 Model 的 OperationData 对象以编辑模式呈现出来。

```
@model OperationData
<html>
  <head>
    <title>将验证特性应用在参数上</title>
  </head>
  <body>
    @Html.EditorForModel()
  </body>
</html>
```

为了让访问 Action 方法 Add 的请求能够直接将操作数放到请求的 URL 中, 我们在默认生成的 RouteConfig 类型中作了如下所示的路由注册。

```
public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        //其他操作
        routes.MapRoute(
            name      : "Add",
            url       : "{action}/{operand1}/{operand2}",
            defaults  : new { controller = "Home" }
        );
    }
}
```

我们在 Global.asax 中通过下面的代码对自定义的 ParameterValidationModel-ValidatorProvider 进行了注册, 在进行注册之前需要将现有的 ModelValidatorProvider 移除。

```
public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        //其他操作
        DataAnnotationsModelValidatorProvider validatorProvider =
            ModelValidatorProviders.Providers
                .OfType<DataAnnotationsModelValidatorProvider>()
                .FirstOrDefault();
        if (null != validatorProvider)
        {
            ModelValidatorProviders.Providers.Remove(validatorProvider);
        }
        ModelValidatorProviders.Providers.Add(
            new ParameterValidationModelValidatorProvider());
    }
}
```



```
}
}
```

现在运行我们的程序，并在浏览器中指定相应的 URL 访问定义在 HomeController 中的 Action 方法 Add。如果提供不合法的操作数（比如 “/Add/50/50”），验证消息将会以如图 9-5 所示的效果显示在相应的文本框旁边。（S904）

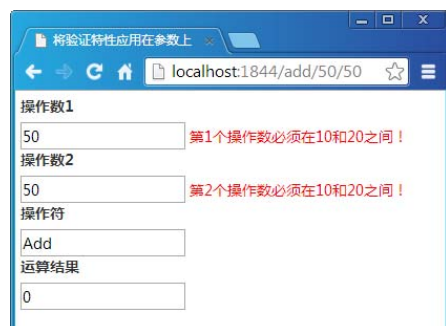


图 9-5 通过应用在 Action 方法参数的验证特性进行 Model 验证

9.2.4 一种 Model 类型，多种验证规则

理想的 Model 验证应该是场景驱动的，而不是类型驱动的，因为同一个数据类型在不同的使用场景中可能具有不同的验证规则。举个简单的例子，对于一个表示应聘者的数据对象来说，应聘的岗位不同，肯定对应聘者的年龄、性别、专业技能等方面具有不同的要求。

但是 ASP.NET MVC 的 Model 验证恰恰是类型驱动的，因为验证规则是通过应用在数据类型及其属性上的 `ValidationAttribute` 特性来定义的，这样的验证方式实际上限制了数据类型在基于不同验证规则的场景中被重用的可能性。上面的扩展将 `ValidationAttribute` 特性直接应用在参数上变成了可能，这在一定程度上解决了这个问题，但是也只能解决部分问题，因为应用到参数的 `ValidationAttribute` 特性只能用于针对参数类型级别的验证，而不能用于针对参数类型属性级别的验证。

现在我们利用对 ASP.NET MVC 的扩展来实现一种基于不同验证规则的 Model 验证。为了让读者对这种验证方式有一个直观的认识，我们先通过一个简单的实例来看看这个扩展最终实现了怎样的验证效果。我们在一个 ASP.NET MVC 应用中定义了如下一个 `Person` 类型。

```
public class Person
{
    [DisplayName("姓名")]
```

```

public string Name { get; set; }

[DisplayName("性别")]
public string Gender { get; set; }

[DisplayName("年龄")]
[RangeValidator(10, 20, RuleName="Rule1",
    ErrorMessage = "{0}必须在{1}和{2}之间!")]
[RangeValidator(20, 30, RuleName = "Rule2",
    ErrorMessage = "{0}必须在{1}和{2}之间!")]
[RangeValidator(30, 40, RuleName = "Rule3",
    ErrorMessage = "{0}必须在{1}和{2}之间!")]
public int Age { get; set; }
}

```

我们在表示年龄的 Age 属性上应用了 3 个自定义的 RangeValidatorAttribute (不是 RangeAttribute) 特性, 它们对应着通过 RuleName 属性指定的 3 种不同的验证规则。3 种验证规则 (Rule1、Rule2 和 Rule3) 分别要求年龄在 10~20、20~30 和 30~40 岁之间。

然后我们定义了如下一个 HomeController, 它具有 3 组 Action 方法 (Index、Rule1 和 Rule2)。方法 Rule1、Rule2 和 HomeController 类上应用了一个自定义的 ValidationRuleAttribute 特性表明当前采用的验证规则。用于指定验证规则的 ValidationRuleAttribute 特性可以同时应用于 Controller 类型和 Action 方法上, 后者具有更高的优先级。针对 HomeController 的定义, Action 方法 Index、Rule1 和 Rule2 分别采用的验证规则为 Rule3、Rule1 和 Rule2。

```

[ValidationRule("Rule3")]
public class HomeController : RuleBasedController
{
    public ActionResult Index()
    {
        return View("person", new Person());
    }
    [HttpPost]
    public ActionResult Index(Person person)
    {
        return View("person", person);
    }

    [ValidationRule("Rule1")]
    public ActionResult Rule1()
    {
        return View("person", new Person());
    }
    [HttpPost]
    [ValidationRule("Rule1")]
    public ActionResult Rule1(Person person)
    {

```

```

        return View("person", person);
    }

    [ValidationRule("Rule2")]
    public ActionResult Rule2()
    {
        return View("person", new Person());
    }
    [HttpPost]
    [ValidationRule("Rule2")]
    public ActionResult Rule2(Person person)
    {
        return View("person", person);
    }
}

```

定义在 HomeController 中的 6 个方法具有相同的操作，它们都将创建/接收的 Person 对象呈现到具有如下定义的 View 中。这是一个 Model 类型为 Person 的强类型 View，我们在该 View 中作为 Model 的 Person 对象以编辑模式呈现在一个表单中，并在表单中提供一个提交按钮。

```

@model Person
<html>
<head>
    <title>编辑个人信息</title>
    <style type="text/css">
        .field-validation-error{color: red;}
    </style>
</head>
<body>
    @using (Html.BeginForm())
    {
        @Html.EditorForModel()
        <input type="submit" value="保存" />
    }
</body>
</html>

```

现在运行我们的程序并通过在浏览器中指定相应的地址分别访问定义在 HomeController 中的 3 个 Action (Index、Rule1 和 Rule2)，一个用于编辑个人信息的表单会呈现出来。然后根据 3 个 Action 方法采用的验证规则输入不合法的年龄，单击“保存”按钮会看到输入的年龄按照对应的规则被验证，具体的验证效果如图 9-6 所示。(S905)

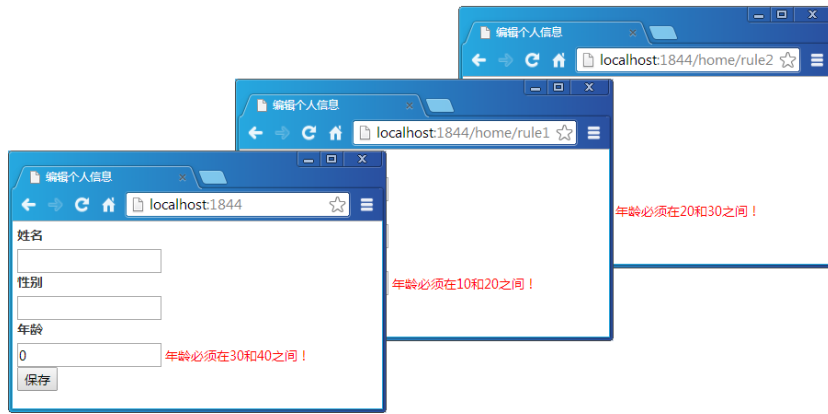


图 9-6 针对不同规则的 Model 验证

我们现在就来具体谈谈上面这个例子所展示的基于不同规则的 Model 验证是如何实现的。我们首先需要重建一套新的验证特性体系，因为需要指定具体的验证规则。我们定义了如下一个 `ValidatorAttribute` 类型，它是一个直接继承自 `ValidationAttribute` 的抽象类，其属性 `RuleName` 表示采用的验证规则名称。我们重写了 `TypeId` 属性，因为需要在相同的属性或者类型上应用多个同类的 `ValidatorAttribute` 特性。

```
[AttributeUsage( AttributeTargets.Class| AttributeTargets.Property,
    AllowMultiple = true)]
public abstract class ValidatorAttribute: ValidationAttribute
{
    private object typeId;
    public string RuleName { get; set; }
    public override object TypeId
    {
        get{return typeId ?? (typeId = new object());}
    }
}
```

上面演示实例中采用的 `RangeValidatorAttribute` 特性定义如下，可以看到它仅仅是对 `RangeAttribute` 的封装。`RangeValidatorAttribute` 具有与 `RangeAttribute` 一致的构造函数定义，并直接使用被封装的 `RangeAttribute` 实施验证。除了能够通过 `RuleName` 指定具体采用的验证规则之外，其他的使用方式与 `RangeAttribute` 完全一致。

```
[AttributeUsage( AttributeTargets.Property, AllowMultiple = true)]
public class RangeValidatorAttribute:ValidatorAttribute
{
    private RangeAttribute rangeAttribute;

    public RangeValidatorAttribute(int minimum, int maximum)
    {
```

```

        rangeAttribute = new RangeAttribute(minimum, maximum);
    }
    public RangeValidatorAttribute(double minimum, double maximum)
    {
        rangeAttribute = new RangeAttribute(minimum, maximum);
    }
    public RangeValidatorAttribute(Type type, string minimum, string maximum)
    {
        rangeAttribute = new RangeAttribute(type, minimum, maximum);
    }
    public override bool IsValid(object value)
    {
        return rangeAttribute.IsValid(value);
    }

    public override string FormatErrorMessage(string name)
    {
        return string.Format(CultureInfo.CurrentCulture,
            base.ErrorMessageString, new object[] {
                name, rangeAttribute.Minimum, rangeAttribute.Maximum });
    }
}

```

ValidatorAttribute 的 RuleName 属性仅仅指定了验证特性采用的验证规则名称, 当前应该采用的验证规则通过应用在 Action 方法或者 Controller 类型上的 ValidationRuleAttribute 特性指定。如下所示的就是这个 ValidationRuleAttribute 的定义, 验证规则名称通过 RuleName 属性表示。

```

[AttributeUsage( AttributeTargets.Class| AttributeTargets.Method)]
public class ValidationRuleAttribute: Attribute
{
    public string RuleName { get; private set; }
    public ValidationRuleAttribute(string ruleName)
    {
        this.RuleName = ruleName;
    }
}

```

对于这个用于实现针对不同验证规则的扩展来说, 其核心是如何将通过 ValidationRuleAttribute 特性设置的验证规则应用到 ModelValidator 的提供机制中, 从众多 ValidationAttribute 列表中筛选出与当前验证规则匹配的那一部分。在这里我们依然使用当前的 ControllerContext 来保存这个验证规则名称。细心的读者应该留意到了上面演示实例中创建的 HomeController 不是继承自 Controller, 而是继承自 RuleBasedController, 这个自定义的 Controller 基类定义如下。

```

public class RuleBasedController: Controller
{
    private static Dictionary<Type, ControllerDescriptor>
        controllerDescriptors = new Dictionary<Type, ControllerDescriptor>();
}

```

```

public ControllerDescriptor ControllerDescriptor
{
    get
    {
        ControllerDescriptor controllerDescriptor;
        if (controllerDescriptors.TryGetValue(this.GetType(),
            out controllerDescriptor))
        {
            return controllerDescriptor;
        }
        lock (controllerDescriptors)
        {
            if (!controllerDescriptors.TryGetValue(this.GetType(),
                out controllerDescriptor))
            {
                controllerDescriptor =
                    new ReflectedControllerDescriptor(this.GetType());
                controllerDescriptors.Add(this.GetType(),
                    controllerDescriptor);
            }
            return controllerDescriptor;
        }
    }
}

protected override IAsyncResult BeginExecuteCore(AsyncCallback callback,
    object state)
{
    SetValidationRule();
    return base.BeginExecuteCore(callback, state);
}

protected override void ExecuteCore()
{
    SetValidationRule();
    base.ExecuteCore();
}

private void SetValidationRule()
{
    string actionName = this.ControllerContext.RouteData
        .GetRequiredString("action");
    ActionDescriptor actionDescriptor = this.ControllerDescriptor
        .FindAction(this.ControllerContext, actionName);
    if (null != actionDescriptor)
    {
        ValidationRuleAttribute validationRuleAttribute =
            actionDescriptor.GetCustomAttributes(true)
                .OfType<ValidationRuleAttribute>().FirstOrDefault() ??
            this.ControllerDescriptor.GetCustomAttributes(true)
                .OfType<ValidationRuleAttribute>().FirstOrDefault() ??
            new ValidationRuleAttribute(string.Empty);
        this.ControllerContext.RouteData.DataTokens.Add(

```

```

        "ValidationRuleName", validationRuleAttribute.RuleName);
    }
}

```

在继承自 `Controller` 的 `RuleBasedController` 中, `ExecuteCore` 和 `BeginExecuteCore` 方法被重写。在调用基类的同名方法之前, 我们调用 `SetValidationRule` 方法提取应用在当前 `Action` 方法/`Controller` 类型上的 `ValidationRuleAttribute` 特性指定的验证规则名称, 并将其保存到当前的 `ControllerContext` 中。由于针对 `ValidationRuleAttribute` 特性的解析需要用到用于描述 `Controller` 的 `ControllerDescriptor` 对象, 出于性能考虑, 我们对该对象进行了全局缓存。

对于应用在同一个属性或者类型上的多个基于不同验证规则的 `ValidatorAttribute` 特性, 对应的验证规则名称并没有应用到具体的验证逻辑中。以上面定义的 `RangeValidatorAttribute` 为例, 具体的验证逻辑通过被封装的 `RangeAttribute` 来实现, 如果不作任何处理, 基于不同规则的 `RangeValidatorAttribute` 都将参与到最终的 `Model` 验证过程中。我们必须要做的是: 在根据特性创建 `ModelValidator` 的时候, 只选择那些与当前验证规则一致的 `ValidatorAttribute` 特性, 这样的操作实现在具有如下定义的 `RuleBasedValidatorProvider` 中。

```

public class RuleBasedValidatorProvider :
    DataAnnotationsModelValidatorProvider
{
    protected override IEnumerable<ModelValidator> GetValidators(
        ModelMetadata metadata, ControllerContext context,
        IEnumerable<Attribute> attributes)
    {
        object validationRuleName = string.Empty;
        context.RouteData.DataTokens.TryGetValue("ValidationRuleName",
            out validationRuleName);
        string ruleName = validationRuleName.ToString();
        attributes = this.FilterAttributes(attributes, ruleName);
        return base.GetValidators(metadata, context, attributes);
    }

    private IEnumerable<Attribute> FilterAttributes(
        IEnumerable<Attribute> attributes, string validationRule)
    {
        var validatorAttributes = attributes.OfType<ValidatorAttribute>();
        var nonValidatorAttributes =
            attributes.Except(validatorAttributes);
        List<ValidatorAttribute> validValidatorAttributes =
            new List<ValidatorAttribute>();

        if (string.IsNullOrEmpty(validationRule))
        {
            validValidatorAttributes.AddRange(validatorAttributes.Where(
                v => string.IsNullOrEmpty(v.RuleName)));
        }
        else

```

```

    {
        var groups = from validator in validatorAttributes
                      group validator by validator.GetType();
        foreach (var group in groups)
        {
            ValidatorAttribute validatorAttribute = group.Where(
                v => string.Compare(v.RuleName, validationRule, true) ==
                0).FirstOrDefault();
            if (null != validatorAttribute)
            {
                validValidatorAttributes.Add(validatorAttribute);
            }
            else
            {
                validatorAttribute = group.Where(
                    v => string.IsNullOrEmpty(v.RuleName))
                    .FirstOrDefault();
                if (null != validatorAttribute)
                {
                    validValidatorAttributes.Add(validatorAttribute);
                }
            }
        }
    }
    return nonValidatorAttributes.Union(validValidatorAttributes);
}
}

```

如上面的代码所示, `RuleBasedValidatorProvider` 继承自 `DataAnnotationsModelValidatorProvider`, 基于当前验证规则(从当前的 `ControllerContext` 中提取)对 `ValidatorAttribute` 的筛选及最终对 `ModelValidator` 的创建实现在重写的 `GetValidators` 方法中, 具体的筛选机制是:

- 如果当前的验证规则存在, 则选择与之具有相同规则名称的第一个 `ValidatorAttribute`。
- 如果这样的 `ValidatorAttribute` 找不到, 则选择第一个没有指定验证规则的 `ValidatorAttribute`。
- 如果当前的验证规则没有指定, 那么同样选择第一个没有指定验证规则的 `ValidatorAttribute`。

我们需要在 `Global.asax` 中通过如下方式对自定义的 `RuleBasedValidatorProvider` 进行注册, 然后我们的应用就能按照期望的方式根据指定的验证规则实施 `Model` 验证了。

```

public class MvcApplication : System.Web.HttpApplication
{
    //其他成员
    protected void Application_Start()
    {
        //其他操作
        DataAnnotationsModelValidatorProvider validator =
            ModelValidatorProviders.Providers

```



```

        .OfType<DataAnnotationsModelValidatorProvider>()
        .FirstOrDefault();
    if (null != validator)
    {
        ModelValidatorProviders.Providers.Remove(validator);
    }
    ModelValidatorProviders.Providers.Add(
        new RuleBasedValidatorProvider());
}
}

```

9.3 客户端验证

之前我们一直讨论的 Model 验证仅限于服务端验证，即在 Web 服务器端根据相应的规则对请求提交的数据实施验证。如果我们能够在客户端（浏览器）对用户输入的数据先进行验证，这样会减少针对服务器请求的频率，从而缓解 Web 服务器的访问压力。ASP.MVC 2.0 及其之前的版本采用 ASP.NET Ajax 进行客户端验证，在 ASP.NET MVC 3.0 中引入了 jQuery 验证框架。

9.3.1 jQuery 验证

Unobtrusive JavaScript 已经成为了 JavaScript 编程的一个指导方针，但是到目前为止貌似还没有一个针对它的确切定义，但是一些 Unobtrusive JavaScript 的基本原则却已经被广泛地接受。Unobtrusive JavaScript 体现了一种被称为“渐进式增强(Progressive Enhancement, PE)”的 Web 设计模式，它采用分层的方式实现了 Web 页面内容与功能的分离。用于实现某种功能的 JavaScript 不再内嵌于用于展现内容的 HTML 中，而是作为独立的层次建立在 HTML 之上。

我们就以验证为例，假设一个 Web 页面中具有如下一个表单，需要针对表单中 3 个文本框（foo、bar 和 baz）的输入进行验证。假设具体的验证操作实现在 validate 函数中，那么我们可以采用如下的 HTML 使相应的文本框在失去焦点的时候对输入的数据实施验证。

```

<form action="/">
    <input id="foo" type="text" onblur="validate()" />
    <input id="bar" type="text" onblur="validate()" />
    <input id="baz" type="text" onblur="validate()" />
    ...
</form>

```

但这不是一个好的设计，理想的方式是让 HTML 定义内容呈现的结构，让 CSS 控制内容呈现的样式，所有功能的实现定义在独立的 JavaScript 中，所以用于实现验证对 JavaScript 的调

用不应该以内联的方式出现在 HTML 中。按照 Unobtrusive JavaScript 的编程方式, 我们应该将以内联方式实现的事件注册 (onblur="validate()") 替换成如下形式。

```
<form action="/">
  <input id="foo" type="text"/>
  <input id="bar" type="text"/>
  <input id="baz" type="text" />
</form>

<script type="text/javascript">
  window.onload = function () {
    document.getElementById("foo").onblur = validate;
    document.getElementById("bar").onblur = validate;
    document.getElementById("baz").onblur = validate;
  }
</script>
```

Unobtrusive JavaScript 是一个很宽泛的话题, 我们不可能在本书中对它进行系统的介绍。

Unobtrusive JavaScript 在 jQuery 的验证中得到了很好的体现, 接下来我们就简单地介绍一下基于 jQuery 验证的编程。

1. 以内联的方式指定验证规则

jQuery 的验证实际上是对表单中的输入元素进行验证, 它支持一种内联的编程方式, 使我们可以直接将验证的规则定义在被验证表单元素的 class (表示 CSS 类型) 属性中。考虑到有一些读者对 jQuery 的验证框架可能不太熟悉, 为此我们来作一个简单的实例演示。我们在一个 ASP.NET MVC 应用中定义了如下一个 HomeController, 在 Action 方法 Index 中将默认的 View 呈现出来。

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        return View();
    }
}
```

我们将作为 Web 页面的整个 HTML 定义在 Action 方法 Index 对应的 View 中, 如下所示的代码片段是该 View 的定义。

```
<html>
  <head>
    <script type="text/javascript"
      src="@Url.Content("~/Scripts/jquery-1.10.2.js")"></script>
    <script type="text/javascript"
      src="@Url.Content("~/Scripts/jquery.validate.js")"></script>
```

```

<script type="text/javascript">
    $(document).ready(function () {
        $("form").validate();
    });
</script>
<title>编辑个人信息</title>
</head>
<body>
    <form action="/">
        <table>
            <tr>
                <td>姓名 : </td>
                <td>
                    <input class="required" id="name" name="name" type="text" />
                </td>
            </tr>
            <tr>
                <td>出生日期 : </td>
                <td>
                    <input class="required date" id="birthDate"
                        name="birthDate" type="text" />
                </td>
            </tr>
            <tr>
                <td>Blog 地址 : </td>
                <td>
                    <input class="required url" id="blogAddress"
                        name="blogAddress" type="text" />
                </td>
            </tr>
            <tr>
                <td>Email 地址 : </td>
                <td>
                    <input class="required email" id="emailAddress"
                        name="emailAddress" type="text" />
                </td>
            </tr>
            <tr>
                <td colspan="2">
                    <input type="submit" value="保存" />
                </td>
            </tr>
        </table>
    </form>
</body>
</html>

```

我们需要将两个必要的.js 文件包含进来，一个是 jQuery 的核心文件 jquery-1.10.2.js，另一个是实现验证的 jquery.validate.js。整个 HTML 文件的主体部分是一个表单，我们可以通过其

中的文本框输入一些个人信息(姓名、出生日期、Blog 地址和 E-mail 地址),最后单击“保存”按钮对输入的数据进行提交。

对于这 4 个文本框对应的<input>元素来说,其 class 属性在这里被用于进行验证规则的定义。其中 required 表示对应的数据是必需的,而 date、url 和 email 则对输入数据的格式进行验证以确保是一个合法的日期、URL 和 E-mail 地址。真正对输入实施验证体现在如下一段 JavaScript 调用中,在这里我们仅仅是调用<form>元素的 validate 方法而已。

```
<script type="text/javascript">
    $(document).ready(function () {
        $("form").validate();
    });
</script>
```

现在运行我们的程序,一个用于提交个人信息的页面会被呈现出来。当我们输入不合法的数据时(第一次验证发生在提交表单时,之后的验证会在被验证表单元素失去焦点时触发),对应的验证将会自动被触发,而预定义的错误消息将会显示在被验证表单元素的右侧。具体的显示效果如图 9-7 所示。(S906)

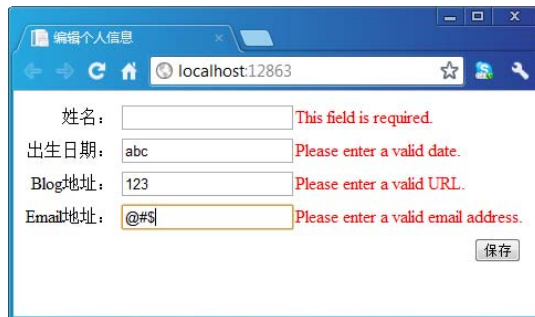


图 9-7 jQuery 验证及其默认错误消息的呈现

2. 单独指定验证规则和错误消息

验证规则其实可以不用以内联的方式定义在被验证表单元素对应的 HTML 中,我们可以直接将它们定义在用于实施验证的 validate 方法中,该方法不仅可以指定表单被验证的输入元素对应的验证规则,还可以指定验证消息,以及控制其他验证行为。

现在我们将上面演示实例中 View 的 HTML 进行相应的修改,将包含在表单中的 4 个文本框通过 class 属性设置的验证规则移除,然后在调用表单的 validate 方法实施验证的时候按照如下方式手工地为被验证输入元素指定相应的验证规则和错误消息。验证规则和错误消息与验证元素之间是通过 name 属性(不是 id 属性)进行关联的。

```

<script type="text/javascript">
$(document).ready(function () {
    $("form").validate({
        rules: {
            name      : { required: true },
            birthDate  : { required: true, date: true },
            blogAddress: { required: true, url: true },
            emailAddress: { required: true, email: true }
        },
        messages: {
            name      : { required: "请输入姓名" },
            birthDate : { required: "请输入出生日期",
                           date: "请输入一个合法的日期" },
            blogAddress: { required: "请输入 Blog 地址",
                           url: "请输入一个合法的 URL" },
            emailAddress: { required: "请输入 Email 地址",
                           email: "请输入一个合法的 Email 地址" }
        }
    });
});
</script>

```

再次运行程序后发现，定制的错误消息就会按照如图 9-8 所示的效果呈现出来。（S907）



图 9-8 jQuery 验证及其定制错误消息的呈现

9.3.2 基于 jQuery 的 Model 验证

在简单了解了 Unobtrusive JavaScript 形式的验证在 jQuery 中的实现之后，我们来具体讨论 ASP.NET MVC 是如何利用它实现客户端验证的。服务端验证最终实现在相应的 `ModelValidator` 中，最终的验证规则定义在相应的 `ValidationAttribute` 中，而客户端验证规则通过

HtmlHelper<TModel>相应的模板方法(比如 TextBoxFor、EditorFor 和 EditorForModel 等)输出到生成的 HTML 中。服务端验证和客户端验证必须采用相同的验证规则,通过应用 ValidationAttribute 特性定义的验证规则也同样体现在基于客户端验证规则的 HTML 上。

1. ValidationAttribute 与 HTML

ASP.NET MVC 默认采用基于验证特性的声明式验证,服务端验证最终实现在两个重写的 IsValid 方法中。对于客户端验证,ASP.NET MVC 对 jQuery 的验证插件进行了扩展,它将验证规则以表单元素属性的方式输出到最终生成的 HTML 中。为了让客户端和服务端采用相同的验证规则,应用在 Model 类型某个属性上的 ValidationAttribute 特性最终会反映在被验证表单元素对应的 HTML 上。

```
public class Contact
{
    [DisplayName("姓名")]
    [Required(ErrorMessage = "请输入{0}!")]
    [StringLength(8, ErrorMessage = "作为{0}字符串长度不能超过{1}!")]
    public string Name { get; set; }

    [DisplayName("电子邮箱地址")]
    [RegularExpression(@"^\w+@[a-zA-Z_]+?\.[a-zA-Z]{2,3}$",
        ErrorMessage = "请输入正确的电子邮箱地址!")]
    public string EmailAddress { get; set; }
}
```

假设我们具有如上一个数据类型 Contact, RequiredAttribute 和 StringLengthAttribute 特性应用到表示姓名的 Name 属性上用于确保用户必须输入一个不超过 8 个字符的字符串,而表示 E-mail 地址的 EmailAddress 属性应用了一个 RegularExpressionAttribute 用于确保用户输入一个合法的 E-mail 地址。在一个以此 Contact 为 Model 类型的 View 中,如果我们调用 HtmlHelper<TModel>的扩展方法 EditorForModel,最终会生成如下一段 HTML。

```
<div class="editor-label">
    <label for="Name">姓名</label>
</div>

<div class="editor-field">
    <input class="text-box single-line"
        data-val=""=""true""
        data-val-length=""作为姓名字符串长度不能超过 8!""
        data-val-length-max=""8""
```

```

        data-val-required    ="请输入姓名!"
        id="Name" name="Name" type="text" value="" />
    <span class="field-validation-valid" data-valmsg-for="Name"
        data-valmsg-replace="true"></span>
</div>

<div class="editor-label">
    <label for="EmailAddress">电子邮箱地址</label>
</div>

<div class="editor-field">
    <input class="text-box single-line"
        data-val              ="true"
        data-val-regex        ="请输入正确的电子邮箱地址!"
        data-val-regex-pattern ="^\\w+@[a-zA-Z_]+?\\.[a-zA-Z]{2,3}$"
        id="EmailAddress" name="EmailAddress" type="text" value="" />
    <span class="field-validation-valid" data-valmsg-for="EmailAddress"
        data-valmsg-replace="true"></span>
</div>

```

通过上面的这段 HTML 我们可以看到，Contact 的两个属性对应的<input>元素具有一个“data-val”属性和一系列以“data-val-”为前缀的属性，前者表示是否需要对用户输入的值进行验证，后者则代表相应的验证规则。具体来说，去除“data-val-”前缀后的属性名称就是 jQuery 验证规则名称。

一般来说，一个 ValidationAttribute 对应着一种验证类型和一系列可选的验证参数。比如 RequiredAttribute、StringLengthAttribute 和 RegularExpressionAttribute 对应的验证类型分别是“required”、“length”和“regex”，而 StringLengthAttribute 和 RegularExpressionAttribute 各自具有一个验证参数 length-max（表示允许的字符串最大长度）和 regex-pattern（正则表达式）。验证错误消息一般作为验证类型属性的值，而验证参数对应的属性值自然就是相应的参数值。

对于上面生成的 HTML 还有一点值得一提，对应着被验证属性的<input>元素会紧跟一个元素用于显示验证失败后的错误消息。该元素的 CSS 类型为“field-validation-valid”，当验证失败后被替换为“field-validation-error”，可以通过它来定制错误消息的显示样式。

2. 客户端验证规则的生成

ASP.NET MVC 在利用 jQuery 进行客户端验证的时候，虽然验证规则并没有采用其原生的方式通过被验证元素的 class 属性来提供，但是却可以通过“data-val-{rulename}”的命名模式提取相应的验证规则属性值，并最终得到对应的验证规则，ASP.NET MVC 只需要对此作简单的“适配”即可。我们现在关心的是当调用定义在 HtmlHelper<TModel>中相应的模板方法将

Model 对象的某个属性以表单元的形式进行呈现的时候, ASP.NET MVC 是如何生成这些以 “data-val-” 为前缀的验证属性的呢?

在这里我们需要涉及一个表示客户端验证规则的 `ModelClientValidationRule` 类型。如下面的代码所示, `ModelClientValidationRule` 具有 3 个属性, 字符串属性 `ErrorMessage` 和 `ValidationType` 表示验证错误消息和验证的类型, 类型为 `IDictionary<string, object>` 的只读属性 `ValidationParameters` 表示辅助客户端验证的参数, 其中 `Key` 和 `Value` 分别表示验证参数名和参数值。

```
public class ModelClientValidationRule
{
    public string ErrorMessage { get; set; }
    public string ValidationType { get; set; }
    public IDictionary<string, object> ValidationParameters { get; }
}

public abstract class ModelValidator
{
    //其他成员
    public virtual IEnumerable<ModelClientValidationRule>
        GetClientValidationRules();
    public abstract IEnumerable<ModelValidationResult> Validate(
        object container);
}
```

通过前面的介绍我们知道, 抽象类 `ModelValidator` 中具有一个虚方法 `GetClientValidationRules`, 用于创建一个 `ModelClientValidationRule` 对象的列表, 所有支持客户端验证的 `ModelValidator` 必须重写该方法以生成相应的客户端验证规则。

以用于进行范围验证的 `RangeAttribute` 特性对应的 `RangeAttributeAdapter` 为例, 如下面的代码片段所示, 它重写了 `GetClientValidationRules`, 返回的 `ModelClientValidationRule` 列表中包含一个 `ModelClientValidationRangeRule` 对象。该 `ModelClientValidationRangeRule` 对象的验证类型为 “range”, 采用 `RangeAttributeAdapter` 的 `ErrorMessage` 属性作为自身的错误消息。作为验证范围上、下限的两个属性 (`Maximum` 和 `Minimum`) 成为了该 `ModelClientValidationRule` 的两个验证参数, 参数名分别为 “max” 和 “min”。

```
public class RangeAttributeAdapter :
    DataAnnotationsModelValidator<RangeAttribute>
{
    //其他成员
    public override IEnumerable<ModelClientValidationRule>
        GetClientValidationRules()
    {
        string errorMessage = base.ErrorMessage;
```



```

        return new ModelClientValidationRangeRule[] {
            new ModelClientValidationRangeRule(errorMessage,
                base.Attribute.Minimum, base.Attribute.Maximum) };
    }
}

public class ModelClientValidationRangeRule : ModelClientValidationRule
{
    public ModelClientValidationRangeRule(string errorMessage,
        object minValue, object maxValue)
    {
        base.ErrorMessage = errorMessage;
        base.ValidationType = "range";
        base.ValidationParameters["min"] = minValue;
        base.ValidationParameters["max"] = maxValue;
    }
}

```

客户端验证在这里还涉及一个重要的接口 `IClientValidatable`，它具有唯一的方法 `GetClientValidationRules`，用来返回一个以 `ModelClientValidationRule` 对象表示的客户端验证规则列表。

```

public interface IClientValidatable
{
    IEnumerable<ModelClientValidationRule> GetClientValidationRules(
        ModelMetadata metadata, ControllerContext context);
}

```

所有支持客户端验证的 `ValidationAttribute` 都需要实现 `IClientValidatable` 接口，并通过实现 `GetClientValidationRules` 方法提供对应的验证规则，而生成的验证规则需要与通过重写的 `IsValid` 方法实现的服务端验证逻辑一致。`DataAnnotationsModelValidator` 重写了 `GetClientValidationRules` 方法，如果对应的 `ValidationAttribute` 实现了 `IClientValidatable` 接口，那么它（`ValidationAttribute`）的 `GetClientValidationRules` 方法会被调用并将返回的 `ModelClientValidationRule` 列表作为该方法的返回值。

当我们在某个 View 中调用 `HtmlHelper<TModel>` 的模板方法将 Model 对象的某个属性以表单元素呈现出来的时候，它会采用前面介绍的 `ModelValidator` 的提供机制根据目标属性对应的 `ModelMetadata` 创建相应的 `ModelValidator`，然后调用 `GetClientValidationRules` 方法得到一组表示客户端验证规则的 `ModelClientValidationRule` 列表。如果该列表不为空，它们将作为验证属性附加到目标属性对应的 `<input>` 元素中。

9.3.3 自定义验证

虽然在命名空间 “`System.ComponentModel.DataAnnotations`” 中具有一系列继承自

ValidationAttribute 的验证特性可以帮助我们完成基本的数据验证,但是在很多场景下的验证需要按照我们自定义的方式来进行,接下来以实例演示的方式来指导读者如何以自定义的方式实现服务端和客户端验证。

假设需要对表示出生日期的输入数据进行验证以确保其年龄在允许的范围内,为此我们创建了一个自定义的验证特性 AgeRangeAttribute。如下面的代码片段所示, AgeRangeAttribute 应用在表示出生日期的属性上,并且指定允许年龄范围(18~25)的上下限。

```
public class Person
{
    [DisplayName("姓名")]
    public string Name { get; set; }

    [DisplayName("出生日期")]
    [AgeRange(18, 25, ErrorMessage = "年龄必须在{0}到{1}周岁之间!")]
    [DisplayFormat(ApplyFormatInEditMode = true,
        DataFormatString = "{0:yyyy-MM-dd}")]
    public DateTime? BirthDate { get; set; }
}
```

为了简单起见,我们直接让 AgeRangeAttribute 继承自 RangeAttribute。如下面的代码片段所示,我们在构造函数中以整数的形式指定表示年龄范围的下限和上限。服务端验证体现在重写的 IsValid 方法中,而为了让格式化的错误消息是针对年龄而不是针对出生日期,我们重写了 FormatErrorMessage 方法。

```
[AttributeUsage( AttributeTargets.Property)]
public class AgeRangeAttribute: RangeAttribute, IClientValidatable
{
    public AgeRangeAttribute(int minimum, int maximum)
        : base(minimum, maximum)
    { }

    public override bool IsValid(object value)
    {
        DateTime? birthDate = value as DateTime?;
        if (null == birthDate)
        {
            return true;
        }
        DateTime age = new DateTime(DateTime.Today.Ticks -
            birthDate.Value.Ticks);
        return (int)this.Minimum <= age.Year && age.Year <= (int)this.Maximum;
    }

    public override string FormatErrorMessage(string name)
    {

```

```

        return string.Format(CultureInfo.CurrentCulture,
            this.ErrorMessageString,
            this.Minimum, this.Maximum);
    }

    public IEnumerable<ModelClientValidationRule> GetClientValidationRules(
        ModelMetadata metadata, ControllerContext context)
    {
        string errorMessage = FormatErrorMessage("");
        ModelClientValidationRule rule = new ModelClientValidationRule
            {ValidationType = "agerange", ErrorMessage = errorMessage };
        rule.ValidationParameters.Add("minage", this.Minimum);
        rule.ValidationParameters.Add("maxage", this.Maximum);
        yield return rule;
    }
}

```

AgeRangeAttribute 实现了 IClientValidatable 接口，在 GetClientValidationRules 方法中返回一个验证类型为“agerange”的 ModelClientValidationRule 对象。它具有两个验证参数 minage 和 maxage，分别表示年龄范围的下限和上限。通过调用 FormatErrorMessage 方法格式化后生成的消息成为该 ModelClientValidationRule 的错误消息。

ModelClientValidationRule 对象的 ValidationType 属性值最终表示 jQuery 验证插件的验证规则，而该规则通过一个对应的函数来实施验证。对于 AgeRangeAttribute 来说，其对应的客户端验证类型为“agerange”，为此我们在一个.js 文件中以此命名注册的验证函数，具体的定义如下所示。

```

jQuery.validator.addMethod("agerange",function (value, element, params) {
    value = value.replace(/(^\\s*)(\\s*$)/g, "");
    if (!value) {
        return true;
    }
    var minAge = params.minage;
    var maxAge = params.maxage;

    var birthDateArray = value.split("-");
    var birthDate = new Date(birthDateArray[0], birthDateArray[1],
        birthDateArray[2]);
    var currentDate = new Date();
    var age = currentDate.getFullYear() - birthDate.getFullYear();
    return age >= minAge && age <= maxAge;
});

jQuery.validator.unobtrusive.adapters.add("agerange", ["minage", "maxage"],
    function (options) {
        options.rules["agerange"] = {
            minage: options.params.minage,
            maxage: options.params.maxage
        };
        options.messages["agerange"] = options.message;
    });

```

```
});
```

如上面的代码片段所示, 我们调用全局对象 jQuery 的 validator 属性的 AddMethod 方法添加了一个用于进行年龄范围验证的函数, 该函数对应的验证规则为 “agerange”。验证函数具有 3 个参数, 其中 value 和 element 分别代表被验证的数据值 (表示出生日期的字符串) 和 HTML 元素, 而 params 参数则表示传入的验证参数 (表示年龄范围的上、下限)。最后我们调用 jQuery.validator.unobtrusive.adapters 的 add 方法对验证规则 agerange 进行注册, 并指定对应的验证参数名称列表。

我们在一个 ASP.NET MVC 应用中定义了以前面定义的 Person 类型为 Model 的 View。如下面的代码片段所示, 我们调用 HtmlHelper<TModel>的扩展方法 EditorForModel 将作为 Model 的 Person 对象以编辑模式呈现在一个表单之中, 通过<script>标签引用了 4 个.js 文件, 前 3 个是 ASP.NET MVC 原生脚本, 最后一个是我们自定义的脚本文件, 上面定义的 “agerange” 相关的扩展就定义在这个文件中。

```
@model Person
<html>
  <head>
    <script type="text/javascript"
      src="@Url.Content("~/Scripts/jquery-1.10.2.js")">
    </script>
    <script type="text/javascript"
      src="@Url.Content("~/Scripts/jquery.validate.js")">
    </script>
    <script type="text/javascript"
      src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.js")">
    </script>
    <script type="text/javascript"
      src="@Url.Content("~/Scripts/jquery.validator.extend.js")">
    </script>
    <title>编辑个人信息</title>
  </head>
  <body>
    @using(Html.BeginForm())
    {
      @Html.EditorForModel()
      <input type="submit" value="保存" />
    }
  </body>
</html>
```

运行程序并在浏览器中将该 View 呈现出来, 在输入违反年龄限制的出生日期的时候, 客户端验证将会生效并以图 9-9 所示的形式显示出错误消息。(S908)



图 9-9 自定义验证对错误消息的呈现

如果我们查看最终生成的 HTML，会发现表示出生日期的文本框具有如下所示的定义，高亮显示的 3 个以 “data-val-” 为前缀的属性内容正是根据 `AgeRangeAttribute` 特性的 `GetClientValidationRules` 方法返回的 `ModelClientValidationRule` 对象生成的。另一个名为 “data-val-date” 的属性是默认添加的针对日期格式的验证规则，由于没有引用相应的本地化 JavaScript 文件，所以错误消息被默认设置为英文。

```
<input class="text-box single-line"
       data-val="true"
       data-val-agerange      ="年龄必须在 18 到 25 周岁之间！"
       data-val-agerange-maxage ="25"
       data-val-agerange-minage ="18"
       data-val-date="The field 出生日期 must be a date."
       id="BirthDate" name="BirthDate" type="text" value="" />
```