




Broadview®  
www.broadview.com.cn

# ASP.NET MVC 4

# 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

针对最新版本的 ASP.NET MVC 4，深入剖析底层框架从请求接收到响应回复的整个处理流程（包括 URL 路由、Controller 的激活、Model 元数据的解析、Model 的绑定、Model 的验证、Action 的执行、View 的呈现和 ASP.NET Web API 等），并在此基础上指导读者如何通过对 ASP.NET MVC 框架本身的扩展解决应用开发中的实际问题。

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 4 框架揭秘 / 蒋金楠著. —北京：电子工业出版社，2013.1  
ISBN 978-7-121-19049-0

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（2012）第 281603 号

策划编辑：张春雨

责任编辑：葛 娜

印 刷：北京东光印刷厂

装 订：三河市鹏成印业有限公司

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：787×1092 1/16 印张：37 字数：855 千字

印 次：2013 年 1 月第 1 次印刷

印 数：3000 册 定价：89.00 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，意味着 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可视化的 UI 界面与用户进行交互，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、用于交互操作的捕捉与响应、业务流程的执行以及对数据的存取，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）可能经常在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 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 Forms 或者 Web Forms）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三个主要的关注点，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：

- 业务逻辑是与 UI 无关的，应该最大限度地被重用。由于业务逻辑定义在自治视图中，相当于完全与视图本身绑定在一起，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共享的。但是定义在自治视图中的 UI 处理逻辑完全丧失了重用的可能。
- 业务逻辑具有最强的稳定性，UI 处理逻辑次之，而可视化界面上的呈现最差（比如我们经常会为了更好地呈现效果来调整 HTML）。如果将具有不同稳定性的元素融为一体，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。
- 任何涉及 UI 的组件都不易测试。UI 是呈现给人看的，并且用于与人进行交互，用机器来模拟活生生的人来对组件实施自动化测试不是一件容易的事，自治视图严重损害了组件的可测试性。

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（Separation of Concerns, SoC）的方针将可视化界面呈现、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变成了可能。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) 期间提出一种主要针对 GUI 应用的软件架构模式。MVC 最初用于 SmallTalk，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，它将构成一个人机交互应用涉及的功能分为 Model、Controller 和 View 三部分，它们各自具有相应的职责。

- 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，在状态改变的时候向 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 (比如鼠标和键盘操作)。
- 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，很多人认为 Controller 仅仅是 View 和 Model 之间的中介，实则不然，View 和 Model 存在直接的联系。View 可以直接调用 Model 查询其状态信息。当 Model 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。

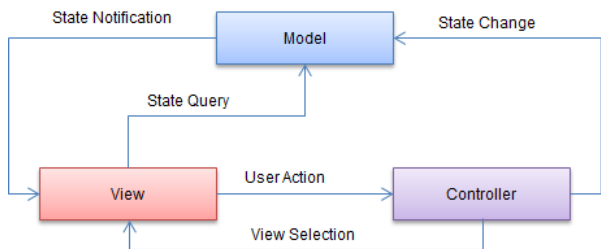


图 1-1 Model-View-Controller 之间的交互

从消息交换模式的角度来讲，Model 针对 View 的状态通知和 View 针对 Controller 的用户交互通知都是单向的，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，即 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，而 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三个层次，不过两者也不能说完全没有关系。Trygve M. H. Reenskau 当时提出 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。而对于多层架构来说（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

通过采用 MVC 模式，我们可以将可视化 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，但是对于三者之间的交互，MVC 并没有进行严格的限制。最为典型的就是允许 View 和 Model 绕开 Controller 进行直接交互，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 让其感知到状态的变化。当我们将 MVC 应用于具体的项目开发中，不论是基于 GUI 的桌面应用还是基于 Web UI 的 Web 应用，如果不对 Model、View 和 Controller 之间的交互进行更为严格的限制，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），这可以从它在 *Applications Programming in Smalltalk-80(TM):How to use Model-View-Controller (MVC)* 中对 MVC 的描述可以看出来：*In the MVC paradigm the user input, the modeling of the external world, and the visual feedback to the user are explicitly separated and handled by three types of object, each specialized for its task.*

模式和范例的区别在于前者可以直接应用到具体的应用上，而后者则仅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成为 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 的界限以及相互之间的交互设置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，我们现在来简单地讨论一些常用的 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，更多地体现在 Model、View 和 Presenter 之间的交互上。

MVC 模式中元素之间“混乱”的交互主要体现在允许 View 和 Model 绕过 Controller 进行单独“交流”，这在 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。

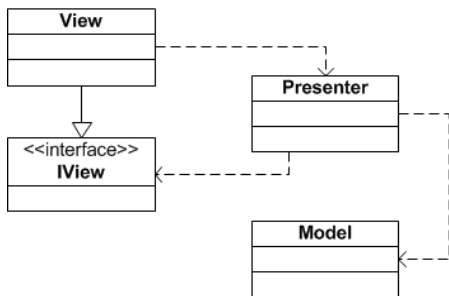


图 1-2 Model-View-Presenter 之间的交互

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 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/Presenter 和 Presenter/Model。Presenter 和 Model 之间的交互很清晰，仅仅体现在 Presenter 对 Model 的单向调用。而 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 可分为 PV (Passive View) 和 SC (Supervising Controller) 两种模式。

### PV 与 SC

解决 View 难以测试的最好的办法就是让它无需测试，如果 View 不需要测试，其先决

条件就是让它尽可能不涉及到 UI 处理逻辑,这就是 PV 模式目的所在。顾名思义,PV(Passive View)是一个被动的 View,包含其中的针对 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。

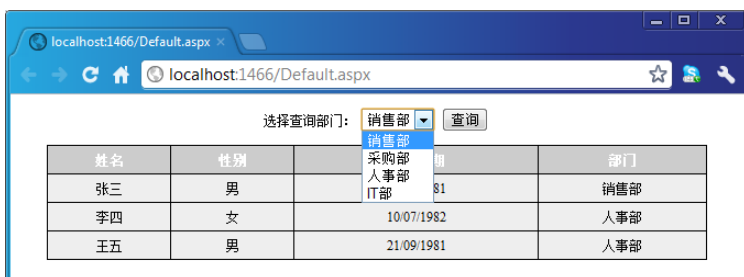


图 1-3 员工查询页面

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。当用户选择某个部门并点击“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。

如果我们为该 View 定义一个接口 IEmployeeSearchView,我们不能按照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(比如针对部门列表的显示,我们可以选择 DropDownList 也可以选择 ListBox),不能体现在表示用于抽象 View 的接口中。另外,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 也与具体的技术平台绑定在了一起。

```
public interface IEmployeeSearchView
{
    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 赋值进而实现对相应 DropDownList 和 GridView 的数



据绑定，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，我们特意将对属性的读/写作了控制。

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

public partial class EmployeeSearchView: Page, IEmployeeSearchView
{
    //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，所以从可测试性的角度来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）`View` 来说，接口成员将会变得很多，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，这无疑会提供 `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，我们将诸如数据绑定和格式化这样简单的 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，而是直接将交互请求转发给 `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` 或者创建新的 `View` 作为对用户交互操作的响应。

## 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。

打个比方, View 是 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。对于以鼠标/键盘操作体现的交互请求应该如何处理, 作为代理的 View 并没有决策权, 所以它会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到 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 最新的状态反应在 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, 应该是 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 完成针对用户交互请求的通知, 还是 Presenter 在进行交互请求处理过程中驱动 View 完成相应的 UI 操作, 都是单向 (One-Way) 的。反应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是单向的。

View 本身仅仅实现单纯的、独立的 UI 处理逻辑, 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, 而避免属性的定义, Presenter 所需的关于 View 的状态应该在接收到 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。

## 实例演示 : SC 模式的应用 ( S101 )

为了让读者对 MVP 模式, 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, 最终呈现出来的效果如图 1-3 所示。前面我们已经演示了采用 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` 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,则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();
    }
}
```

接下来我们来看作为 View 接口的 `IEmployeeSearchView` 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，IEmployeeSearchView 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点击“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 IEmployeeSearchView
{
    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 EmployeeSearchPresenter 表示。如下面的代码片段所示，表示 View 的只读属性类型为 IEmployeeSearchView 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 EmployeeSearchPresenter
{
    public IEmployeeSearchView View { get; private set; }
    public EmployeeRepository Repository { get; private set; }

    public EmployeeSearchPresenter(IEmployeeSearchView 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);
    }
}
```

在构造函数中我们注册了 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 方法实现了在 View 上的数据绑定。在 Initialize 方法中，我们通过调用 Repository 获取所有员工的列表，并通过 View 的 BindEmployees 方法显示在界面上。作为筛选条件的部门列表通过调用 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。如下所示的是作为页面主体部分的 HTML，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 GridView。

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>员工管理</title>
    <link rel="stylesheet" href="Style.css" />
  </head>
  <body>
    <form id="form1" runat="server">
      <div id="page">
        <div class="top">
          选择查询部门：
          <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 IEmployeeSearchView 接口的两个方法（BindEmployees 和 BindDepartments）和一个事件（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”按钮被点击的时候（ButtonSearch\_Click）事件 DepartmentSelected 被触发。

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

    public Default()
    {
        this.Presenter = new EmployeeSearchPresenter(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 是作为基于 GUI 的桌面应用的架构模式并不太适合 Web 本身的特性，虽然 MVC/MVP 也可以直接用于 ASP.NET Web Forms 应用，但这是因为微软就是基于桌面应用的编程模式来设计基于 Web Forms 的 ASP.NET 应用框架的。Web 应用不同于 GUI 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，这来源于基于 Java 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试图模式，

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应该采用 Model 2。

为了让开发者采用相同的编程模式进行 GUI 桌面应用和 Web 应用的开发，微软通过 ViewState 和 Postback 对 HTTP 请求和回复机制进行了封装，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。而 Model 2 采用完全不同的设计，它让开发者直接面向 Web，让他们关注 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 界面由浏览器来提供，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。图 1-4 所示的序列图体现了整个流程的全过程。

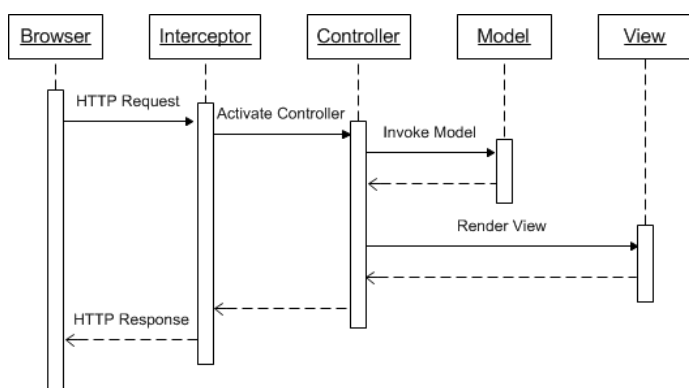


图 1-4 Model 2 交互流程

Model 2 中一个 HTTP 请求的目标是 Controller 中的某个 Action，后者体现为定义在 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 对象的激活和对相应 Action 方法的执行。一般来说，Controller 的类型和 Action 方法的名称以及作为 Action 方法的部分参数（针对 HTTP-GET）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。拦截器根据请求解析出目标 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。

在激活 Controller 对象的目标 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会选择相应的 View，整个 View 被最终转换成 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。对于 HTTP 请求的拦截以实现为目标 Controller 和 Action 的解析是通过一个自定义 `HttpModule` 来实现的，而对目标 Controller 的激活则通过一个自定义 `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。

在上面我们多次强调 MVC 的 Model 是维持应用状态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者业务服务的代理，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 是基于 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于 View 和 Controller，真正的 Model 以及 Model 和 Controller 之间的交互体现在我们如何来设计 Controller。我个人觉得将用于构建 ASP.NET MVC 的 MVC 模式成为 M (Model) -V (View) -VM (View Model) -C (Controller) 也许更为准确。

## 1.3 IIS/ASP.NET 管道

前面我们对 MVC 模式及其变体作了详细的介绍，其目的在于让读者充分地了解 ASP.NET MVC 框架的设计思想，接下来我们来介绍支撑 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 模式建立的 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 应用总是寄宿于 IIS 上，所以我们将两者结合起来介绍，力求让读者完整地理解请求在 IIS/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。整个过程基本上可以通过图 1-5 体现。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 的主要功能包括 HTTP 请求的监听、工作进程和配置管理（通过从 Metabase 中加载相关配置信息）等。



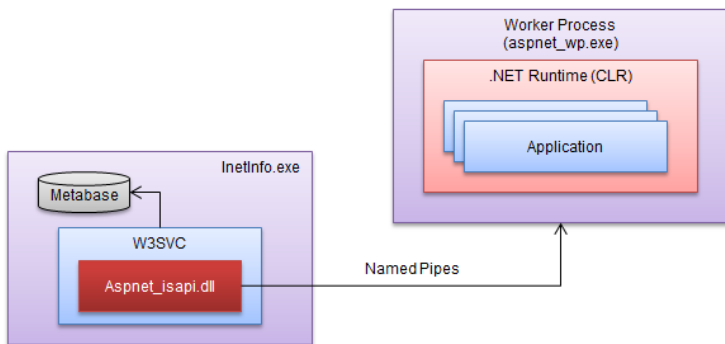


图 1-5 IIS 5.x 与 ASP.NET

当检测到某个 HTTP 请求时，先根据扩展名判断请求的是否是静态资源（比如.html、.img、.txt、.xml 等），如果是，则直接将文件内容以 HTTP 回复的形式返回；如果是动态资源（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 动态连接库（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 定义在一个动态连接库（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 和 .svc 等，aspnet\_isapi.dll 会被加载，而 ASP.NET ISAPI 扩展会创建 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）被加载进而构建了一个托管的环境。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

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 动态连接库被加载到 `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`) 中的不同的应用程序域中,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 动态连接库直接加载到工作进程中;为了解决第二个问题,引入了应用程序池 (Application Pool) 的机制。我们可以为一个或多个 Web 应用创建应用程序池,由于每一个应用程序池对应一个独立的工作进程,从而为运行在不同应用程序池中的 Web 应用提供基于进程的隔离级别。IIS 6.0 的工作进程名称为 `w3wp.exe`。

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 2003 的 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) 中,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,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 的结构和处理 HTTP 请求的流程。与 IIS 5.x 不同,W3SVC 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。

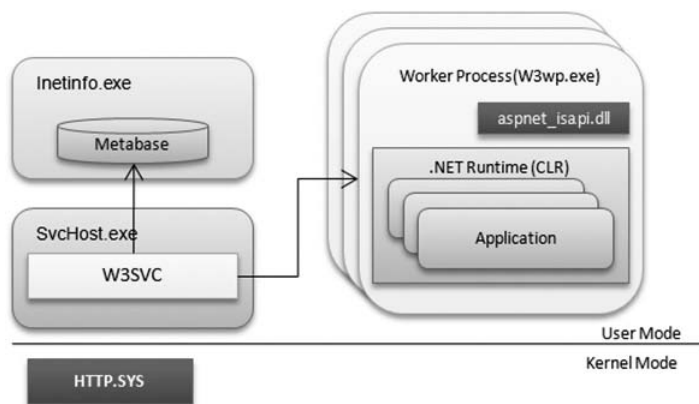


图 1-6 IIS 6.0 与 ASP.NET

当 HTTP.SYS 监听到用户的 HTTP 请求时将其分发给 W3SVC，W3SVC 解析出请求的 URL，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，并进一步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），则为该请求创建新的工作进程。我们将工作进程的这种创建方式称为请求式创建。在工作进程的初始化过程中，相应的 ISAPI 动态连接库被加载。对于 ASP.NET 应用来说，被加载的 ISAPI.dll 为 aspnet\_isapi.dll。ASP.NET ISAPI 再负责进行 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，主要体现在对于 Windows 进程激活服务（Windows Process Activation Service, WAS）的引入，将原来（IIS 6.0）W3SVC 承载的部分功能分流给了 WAS。通过上面的介绍，我们知道对于 IIS 6.0 来说 W3SVC 主要承载着 3 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，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。WAS 通过监听器适配器接口（Listener Adapter Interface）抽象出不同协议监听器。具体来说，除了基于网络驱动的 HTTP.SYS 提供 HTTP 请求监听功能外还提供了 TCP 监听器、命名管道监听器和 MSMQ 监听器以提供基于 TCP、命名管道和 MSMQ 传输协议的监听支持。

与此 3 种监听器相对的是 3 种监听适配器，它们提供监听器与 WAS 中的监听器适配

器接口之间的适配。从这个意义上讲，IIS 7.0 中的 W3SVC 更多地为 HTTP.SYS 起着监听适配器的作用。这 3 种非 HTTP 监听器和监听适配器定义在程序集 SMHost.exe 中，我们可以在目录 %windir%\Microsoft.NET\Framework\v3.0\Windows Communication Foundation\ 中找到它们。

WCF 提供的这 3 种监听器和监听适配器最终以 Windows 服务的形式体现。虽然它们定义在一个程序集中，我们依然可以通过服务工作管理器对其进行单独的启动、终止和配置。SMHost.exe 提供了 4 个重要的 Windows Service。

- NetTcpPortSharing: 为 WCF 提供 TCP 端口共享。关于端口共享在 WCF 中的应用，本人拙著《WCF 全面解析》(上册) 对此有详细的介绍。
- 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 为上述的 4 个 Windows 服务在服务控制管理器中的呈现。

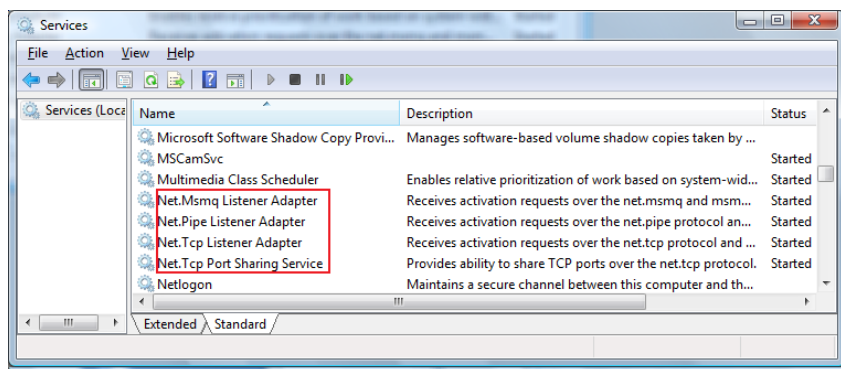


图 1-7 定义在 SMHost.exe 中的 Windows Service

图 1-8 揭示了 IIS 7.0 的整体构架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请求，最终都会传递到 WAS。如果相应的工作进程（或者应用程序池）尚未创建，则创建它，否则将请求分发给对应的工作进程进行后续的处理。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。

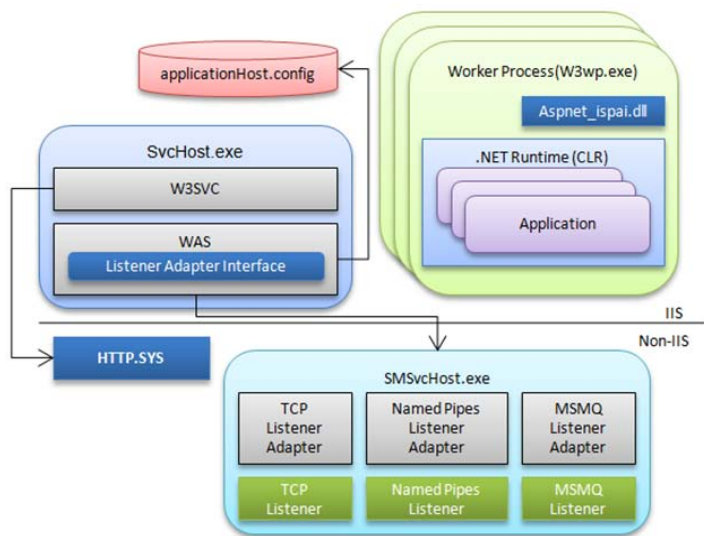


图 1-8 IIS 7.0 与 ASP.NET

## 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）时，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），最终生成 HTTP 响应。图 1-9 反映了 IIS 6.0 与 ASP.NET 之间的桥接关系。

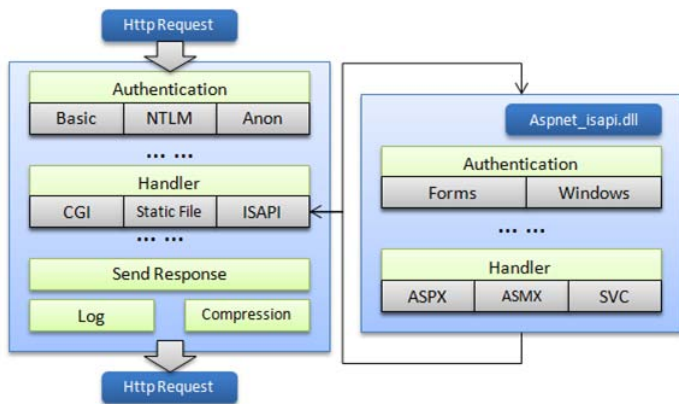


图 1-9 基于 IIS 6.0 与 ASP.NET 双管道设计

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，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。比如，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。

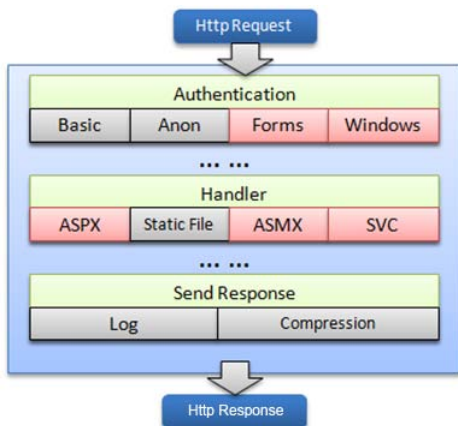


图 1-10 基于 IIS 7.0 与 ASP.NET 集成管道设计

### 1.3.4 ASP.NET 管道

以 IIS 6.0 为例，在工作进程 w3wp.exe 中，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，在成功加载了运行时后，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 中，对应的命名空间为 System.Web.Hosting，被加载的 IsapiRuntime 会接管该 HTTP 请求。

IsapiRuntime 会首先创建一个 IsapiWorkerRequest 对象，用于封装当前的 HTTP 请求，并将该 IsapiWorkerRequest 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 被成功创建，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，处理完毕后将将其释放到对象池中。HttpApplicationFactory 负责处理当前的 HTTP 请求。

在 HttpApplication 初始化过程中，会根据配置文件加载并初始化相应的 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 的相应的事件，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。对于不同的资源类型，具有不同的 `HttpHandler`。比如 `.aspx` 页面对应的 `HttpHandler` 为 `System.Web.UI.Page`，WCF 的 `.svc` 文件对应的 `HttpHandler` 为 `System.ServiceModel.Activation.HttpHandler`。上面整个处理流程如图 1-11 所示。

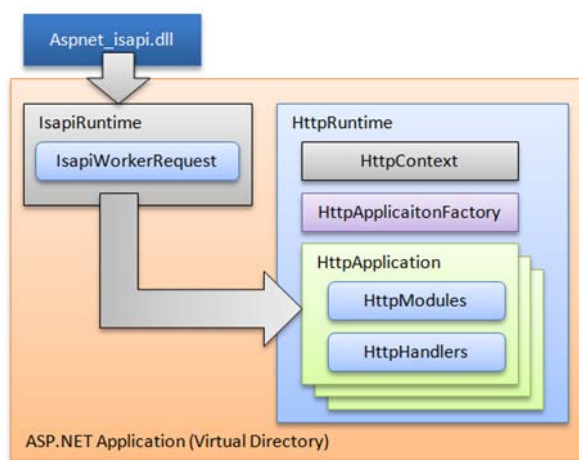


图 1-11 ASP.NET 处理管道

## 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，`HttpApplication` 才能用于后续请求的处理，所以 ASP.NET 采用对象池的机制来创建或获取 `HttpApplication` 对象。

当第一个请求抵达时，ASP.NET 会一次创建多个 `HttpApplication` 对象，并将其置于池中，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，ASP.NET 会创建新的 `HttpApplication` 对象。

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

表 1-1 `HttpApplication` 事件列表

名 称	描 述
-----	-----



BeginRequest	HTTP 管道开始处理请求时，会触发 BeginRequest 事件
AuthenticateRequest , PostAuthenticateRequest	ASP.NET 先后触发这两个事件，使安全模块对请求进行身份验证
AuthorizeRequest , PostAuthorizeRequest	ASP.NET 先后触发这两个事件，使安全模块对请求进程授权
ResolveRequestCache , PostResolveRequestCache	ASP.NET 先后触发这两个事件，以使缓存模块利用缓存的内容对请求直接进行响应（缓存模块可以将响应内容进行缓存，对于后续的请求，直接将缓存的内容返回，从而提高响应能力）
PostMapRequestHandler	对于访问不同的资源类型，ASP.NET 具有不同的 HttpHandler 对其进行处理。对于每个请求，ASP.NET 会通过扩展名选择匹配相应的 HttpHandler 类型，成功匹配后，该实现被触发
AcquireRequestState , PostAcquireRequestState	ASP.NET 先后触发这两个事件，使状态管理模块获取基于当前请求相应的状态，如 SessionState
PreRequestHandlerExecute , PostRequestHandlerExecute	ASP.NET 最终通过与请求资源类型相对应的 HttpHandler 实现对请求的处理，在实行 HttpHandler 前后，这两个实现被先后触发
ReleaseRequestState , PostReleaseRequestState	ASP.NET 先后触发这两个事件，使状态管理模块释放基于当前请求相应的状态
UpdateRequestCache , PostUpdateRequestCache	ASP.NET 先后触发这两个事件，以使缓存模块将 HttpHandler 处理请求得到的内容得以保存到输出缓存中
LogRequest , PostLogRequest	ASP.NET 先后触发这两个事件为当前请求进行日志记录
EndRequest	整个请求处理完成后，EndRequest 事件被触发

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

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

## 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，那么是什么成就了 ASP.NET 的高可扩展性呢？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 Handler 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，会将一些功能注册到 `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，其中 `Init` 方法用于实现 `HttpModule` 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`。

- `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。
- `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。

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。

## HttpHandler

对于不同资源类型的请求，ASP.NET 会加载不同的 Handler 来处理，也就是说.aspx 页面与.aspx 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`，方法 `ProcessRequest` 提供了处理请求的实现。

---

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

某些 `HttpHandler` 具有一个与之相关的 `HttpHandlerFactory`，它实现了具有如下定义的接口 `System.Web.IHttpHandlerFactory`，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` 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>
```

## 1.4 ASP.NET MVC 是如何运行的

ASP.NET 由于采用了管道式设计，具有很好的扩展性，而整个 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，实际上整个 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 的工作机制，接下来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 中找到对应的组件，本书完全采用了与 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 应用如何定义。通过 Visual Studio 创建一个空的 ASP.NET Web 应用（注意不是 ASP.NET MVC 应用）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 的组件都是我们自行定义的。

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 `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`。

---

```
public class SimpleModel
{
    public string Controller { get; set; }
    public string Action { 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 `HomeController`。`HomeController` 实现的抽象类型 `ControllerBase` 是我们自行定义的。以自定义的 `ActionResult` 作为返回类型的 `Index` 方法表示 `Controller` 的 `Action`，它接受一个 `SimpleModel` 类型的对象作为参数。该 `Action` 方法返回的 `ActionResult` 是一个 `RawContextResult` 对象，顾名思义，`RawContextResult` 就是将指定的内容进行原样返回。在这里我们将作为参数的 `SimpleModel` 对象的 `Controller` 和 `Action` 属性显示出来。

---

```
public class HomeController: ControllerBase
{
    public ActionResult Index(SimpleModel model)
    {
```

```

        string content = string.Format("Controller: {0}<br/>Action:{1}",
            model.Controller, model.Action);
        return new RawContentResult(content);
    }
}

```

ASP.NET MVC 根据请求地址来解析出用于处理该请求的 Controller 的类型和 Action 方法名称。具体来说，我们预注册一些包含 Controller 和 Action 名称作为占位符的（相对）地址模板，如果请求地址符合相应地址模板的模式，Controller 和 Action 名称就可以正确地解析出来。和 ASP.NET MVC 应用类似，我们在 Global.asax 中注册了如下一个地址模板（{controller}/{action}）。我们还注册了一个用于创建 Controller 对象的工厂。RouteTable、ControllerBuilder 和 DefaultControllerFactory 都是我们自定义的类型。

---

```

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());
    }
}

```

正如上面所说的，ASP.NET MVC 是通过一个自定义的 HttpModule 实现的，在这个“迷你版”ASP.NET MVC 框架中我们也将起名为 UrlRoutingModule。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。

---

```

<configuration>
  <system.webServer>
    <modules>
      <add name="UrlRoutingModule"
          type="WebApp.UrlRoutingModule, WebApp"/>
    </modules>
  </system.webServer>
</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 URL 模式）。如图 1-12 所示，由于在浏览器中输入地址（http://.../Home/Index）正好对应着 HomeController 的 Action 方法 Index，所以对应的方法会被执行，而执行的结果就是将当前请求的目标 Controller 和 Action 的名称显示出来。（S102）

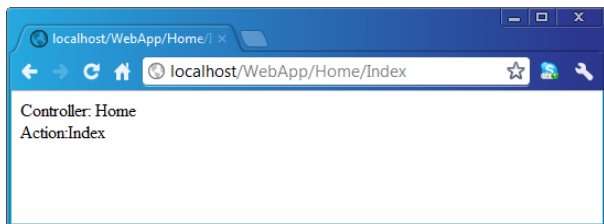


图 1-12 采用符合注册的路由地址模板的地址访问 Web 应用

上面演示了如何在我们自己创建的“迷你版”ASP.NET MVC 框架中创建一个 Web 应用，从中可以看到和创建一个真正的 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，而它也代表了真正的 ASP.NET MVC 框架的工作原理。

## 1.4.2 URL 路由

对于一个 ASP.NET MVC 应用来说，针对 HTTP 请求的处理实现在某个 Controller 类型的某个 Action 方法中，每个 HTTP 请求不再像 ASP.NET Web Forms 应用一样是对应着一个物理文件，而是对应着某个 Controller 的某个 Action。目标 Controller 和 Action 的名称包含在 HTTP 请求的 URL 中，而 ASP.NET MVC 的首要任务就是通过当前 HTTP 请求的解析得到正确的 Controller 和 Action 的名称，这个过程是通过 ASP.NET MVC 的 URL 路由机制来实现的。

### RouteData

ASP.NET 定义了一个全局的路由表，路由表中的每个路由对象包含一个 URL 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}”）定义在 URL 模板中，也可以作为路由对象的默认值。对于每一个抵达的 HTTP 请求，ASP.NET MVC 会遍历路由表找到一个具有与当前请求 URL 模式相匹配的路由对象，并最终解析出以 Controller 和 Action 名称为核心的路由数据。在我们自定义的 ASP.NET MVC 框架中，路由数据通过具有如下定义的 RouteData 类型表示。

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

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

```

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

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

```

如上面的代码片段所示，`RouteData` 定义了两个字典类型的属性 `Values` 和 `DataTokens`，前者代表直接从请求地址解析出来的变量列表，后者代表具有其他来源的变量列表。表示 `Controller` 和 `Action` 名称的同名属性直接从 `Values` 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`，而后者从 `RouteData` 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` 用于返回真正用于处理 HTTP 请求的 `HttpHandler` 对象。

---

```

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

```

`IRouteHandler` 接口的 `GetHttpHandler` 方法接受一个类型为 `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; }
}

```

## Route 和 RouteTable

`RouteData` 具有一个类型为 `RouteBase` 的 `Route` 属性，表示生成路由数据对应的路由对

象。如下面的代码片段所示，`RouteBase` 是一个抽象类，它仅仅包含一个 `GetRouteData` 方法。该方法判断是否与当前请求相匹配，并在匹配的情况下返回用于封装路由数据的 `RouteData` 对象。该方法接受一个表示当前 HTTP 上下文的 `HttpContextBase` 对象，如果与当前请求不匹配，则返回 `Null`。

---

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

ASP.NET MVC 提供的基于 URL 模板的路由机制是通过其子类 `Route` 实现的。如下面的代码片段所示，它具有一个代表 URL 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 URL 模板的模式相匹配则创建一个 `RouteData` 返回，否则返回 `Null`。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地址解析出来的变量，而对于 `DataTokens` 字典和 `RouteHandler` 属性，则直接取自 `Route` 对象的同名属性。

---

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

    public Route()
    {
        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]);
    }
}
return true;
}
}

```

由于同一个 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` 的路由对象，多个路由对象组成了一个路由表。在我们自定义 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` 表示一个具名的路由对象的列表，我们直接让它继承自泛型的字典类型 `Dictionary<string, RouteBase>`，其中的 `Key` 表示路由对象的注册名称。在 `GetRouteData` 方法中，我们遍历集合找到与指定的 `HttpContextBase` 对象匹配的路由对象，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 URL 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。

## UrlRoutingModule

路由表的作用是对当前的 HTTP 请求的 URL 进行解析，从而获取一个以 `Controller` 和 `Action` 名称为核心的路由数据，即上面介绍的 `RouteData` 对象。整个解析工作是通过一个类型为 `UrlRoutingModule` 的自定义 `HttpModule` 来完成的。如下面的代码片段所示，在实现了接口 `IHttpModule` 的 `UrlRoutingModule` 类型的 `Init` 方法中，我们注册了 `HttpApplicataion` 的

PostResolveRequestCache 事件。

```
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);
    }
}
```

当 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, 然后调用其 GetRouteData 方法并传入用于封装当前 HttpContext 的 HttpContextWrapper 对象 (HttpContextWrapper 是 HttpContextBase 的子类) 最终得到一个封装路由数据的 RouteData 对象。如果得到的 RouteData 不为 Null, 则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, 将其作为参数传入 RouteData 的 RouteHandler 的 GetHttpHandler 方法得到一个 IHttpHandler 对象。最后我们调用 HttpContextWrapper 对象的 RemapHandler 方法将得到的 IHttpHandler 进行映射使之用于对当前 HTTP 请求的处理。

### 1.4.3 Controller 的激活

ASP.NET MVC 的 URL 路由系统通过注册的路由表对 HTTP 请求进行解析从而得到一个用于封装路由数据的 RouteData 对象, 而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。RouteData 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。接下来进一步分析真正的 Controller 对象是如何被激活的。

## 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，在实现的 `GetHttpHandler` 方法中，它会直接返回一个 `MvcHandler` 对象。

---

```
public class MvcRouteHandler: IRouteHandler
{
    public IHttpHandler GetHttpHandler(RequestContext requestContext)
    {
        return new MvcHandler(requestContext);
    }
}
```

## MvcHandler

在前面的内容中已经提到整个 ASP.NET MVC 框架是通过自定义的 `HttpModule` 和 `HttpHandler` 对象 ASP.NET 进行扩展实现的。这个自定义 `HttpModule` 已经介绍过了，就是 `UrlRoutingModule`，而这个自定义的 `HttpHandler` 则是要重点介绍的 `MvcHandler`。

`UrlRoutingModule` 在通过路由表解析 HTTP 请求得到一个用于封装路由数据的 `RouteData` 后，会调用其 `RouteHandler` 的 `GetHttpHandler` 方法得到 `HttpHandler` 对象并注册到当前的 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` 对象的激活和对相应 `Action` 方法的执行。

下面的代码片段体现了整个 `MvcHandler` 的定义，它具有一个类型为 `RequestContext` 的属性，表示被处理的当前请求上下文，该属性在构造函数中指定。在实现的 `ProcessRequest`

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 Controller 的执行。该方法在 `MvcHandler` 的 `ProcessRequest` 方法中被执行，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 Controller 工厂定义了一个具有如下定义的 `IControllerFactory` 接口。`IControllerFactory` 通过 `CreateController` 方法根据传入的请求上下文和 Controller 的名称来激活相应的 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 `GetControllerFactory` 方法获得当前的 `ControllerFactory`。然后通过从 `RequestContext` 中提取的 `RouteData` 获得 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 的静态只读属性的 Current 在静态构造函数中被创建。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。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，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），通过 BuildManager 加载所有引用的程序集，并得到所有实现了接口 IController 的类型并将其缓存起来。在 CreateController 方法中根据 Controller 的名称和命名空间从保存的 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。通过实现 HomeController 接口我们为所有的 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: HomeController
{
    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 基类 ControllerBase 的 Execute 方法的核心在于对 Action 方法本身的执行和作为方法返回的 ActionResult 的执行，两者的执行是通过一个叫做 ActionInvoker 的组件来完成的。



## ActionInvoker

同样为 `ActionInvoker` 定义了一个接口 `IActionInvoker`，如下面的代码片段所示，该接口定义了一个唯一的方法 `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基于当前 `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，而这两个要素分别通过如下所示的 `Controller` 和 `RequestContext` 属性表示。

---

```
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，通过作为方法参数的 `RequestContext` 对象创建 `ControllerContext` 对象，并通过包含在 `RequestContext` 中的 `RouteData` 得到目标 `Action` 的名称，然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。如下所示的代码片段反映了整个 `ControllerActionInvoker` 的定义，`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` 的绑定，而这又涉及另一个重要的组件 `ModelBinder`。

---

```
public class ControllerActionInvoker : IActionInvoker
{
    public IModelBinder ModelBinder { get; private set; }

    public ControllerActionInvoker()
    {
        this.ModelBinder = new DefaultModelBinder();
    }

    public void InvokeAction(ControllerContext controllerContext,
        string actionName)
    {
        MethodInfo method = controllerContext.Controller.GetType().GetMethod()
            .First(m =>string.Compare(actionName, m.Name, true) == 0);
        List<object> parameters = new List<object>();
        foreach (ParameterInfo parameter in method.GetParameters())
        {
            parameters.Add(this.ModelBinder.BindModel(controllerContext,
                parameter.Name, parameter.ParameterType));
        }
    }
}
```

```

    }
    ActionResult actionResult = method.Invoke(controllerContext.Controller,
        parameters.ToArray()) as ActionResult;
    actionResult.ExecuteResult(controllerContext);
}
}

```

## ModelBinder

我们为 `ModelBinder` 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，根据 `ControllerContext` 和 `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` 对象是一个 `DefaultModelBinder` 对象，由于仅仅是对 ASP.NET MVC 的模拟，定义在自定义的 `DefaultModelBinder` 中的 `Model` 绑定逻辑比 ASP.NET MVC 的 `DefaultModelBinder` 要简单得多，很多复杂的 `Model` 机制并未在我们自定义的 `DefaultModelBinder` 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三个来源：`HTTP-POST Form`、`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（比如之前例子中定义的包含 `Controller` 和 `Action` 名称的数据类型 `SimpleModel`），则通过反射根据类型创建新的对象，并根据属性名称与 `Key` 的匹配关系对相应的属性进行赋值。

---

```

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;
        }
        objectpropertyValue;
        if (GetValueTypeInstance(controllerContext, property.Name,
            property.PropertyType, out propertyValue))
        {
            property.SetValue(modelInstance, propertyValue, null);
        }
    }
    returnmodelInstance;
}

private boolGetValueTypeInstance(ControllerContext controllerContext,
    stringmodelName, Type modelType, out object value)
{
    var form = HttpContext.Current.Request.Form;
    string key;
    if (null != form)
    {
        key = form.AllKeys.FirstOrDefault(k =>string.Compare(k,
            modelName, true) == 0);
        if (key != null)
        {
            value = Convert.ChangeType(form[key], modelType);
            return true;
        }
    }

    key = controllerContext.RequestContext.RouteData.Values
        .Where(item =>string.Compare(item.Key, modelName, true) == 0)
        .Select(item =>item.Key).FirstOrDefault();
    if (null != key)
    {
        value = Convert.ChangeType(controllerContext.RequestContext
            .RouteData.Values[key], modelType);
        return true;
    }

    key = controllerContext.RequestContext.RouteData.DataTokens
        .Where(item =>string.Compare(item.Key, modelName, true) == 0)
        .Select(item =>item.Key).FirstOrDefault();
    if (null != key)
    {
        value = Convert.ChangeType(controllerContext.RequestContext
            .RouteData.DataTokens[key], modelType);
        return true;
    }
    value = null;
    return false;
}
}

```

在 `ControllerActionInvoker` 的 `InvokeAction` 方法中，我们直接将传入的 `Action` 名称作为方法名从 `Controller` 类型中得到表示 `Action` 操作的 `MethodInfo` 对象，然后遍历 `MethodInfo` 的参数列表，对于每一个 `ParameterInfo` 对象，我们将它的 `Name` 和 `ParameterType` 属性表示

的参数名称和类型，连同创建的 `ControllerContext` 作为参数调用 `ModelBinder` 的 `BindModel` 方法并得到对应的参数值，最后通过反射的方式传入参数列表并执行 `MethodInfo`。

和真正的 ASP.NET MVC 一样，定义在 `Controller` 的 `Action` 方法返回一个 `ActionResult` 对象，我们通过执行它的 `Execute` 方法实现对请求的响应。

## 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` 将初始化时指定的内容（字符串）原封不动地写入针对当前请求的 HTTP 响应消息中，具体的实现如下所示。

---

```
public class RawContentResult: ActionResult
{
    public string RawData { get; private set; }
    public RawContentResult(string rawData)
    {
        RawData = rawData;
    }
    public override void ExecuteResult(ControllerContext context)
    {
        context.RequestContext.HttpContext.Response.Write(this.RawData);
    }
}
```

## 本章小结

ASP.NET MVC 是在现有的 ASP.NET 平台上基于 MVC 架构模式创建的 Web 应用开发框架。MVC 体现了界面呈现、UI 处理逻辑和业务逻辑之间的分离，传统的 MVC 并没有对 Model、View 和 Controller 之间的交互进行严格的约束。在软件设计的发展历程中出现了一些 MVC 的变体，它们遵循定义在 MVC 中的基本原则，并定义更加严格的交互规则，其中 MVP 和 Model 2 是两个典型的 MVC 变体，而 ASP.NET MVC 就是对 Model 2 的实现。

ASP.NET 采用极具扩展性的管道式设计，`HttpApplication` 是整个 ASP.NET 管道的核心，它定义了一系列的事件，它们会在请求处理过程中相应的阶段被触发。`HttpModule` 是成就 ASP.NET 可扩展的“头号功臣”，通过 `HttpModel` 注册 `HttpApplication` 相应的事件帮助我们

在某个阶段参与到对请求处理的整个流程之中，而请求的最终处理者是注册的 `HttpHandler`。`ASP.NET MVC` 实际上是通过自定义的 `HttpModule` 和 `HttpHandler` 构建的。

为了让读者对 `ASP.NET MVC` 对从“接收请求”到“回复响应”的整个处理流程有一个大致的了解，我们按照 `ASP.NET MVC` 本身的实现原理构建了一个模拟程序，该程序模拟了 URL 路由、Controller 的激活、Action 的执行和 View 的呈现，可以将此模拟程序看成是一个“迷你版”的 `ASP.NET MVC`。

## 第 3 章 Controller 的激活

ASP.NET MVC 应用中请求的目标不再是具体某个物理文件,而是定义在某个 Controller 中的 Action 方法。每个请求经过 ASP.NET URL 路由系统的拦截后,会生成以 Controller/Action 名称为核心的路由数据。ASP.NET MVC 据此解析出目标 Controller 的类型,并最终激活具体的 Controller 实例来处理当前的请求。

## 3.1 总体设计

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

### 3.1.1 Controller

我们知道作为 Controller 的类型直接或者间接实现了 `System.Web.Mvc.IController` 接口。如下面的代码片段所示，`IController` 接口仅仅包含一个参数类型为 `RequestContext` 的 `Execute` 方法，当一个 Controller 对象被激活之后，其核心的操作就是：从包含在当前请求上下文的路由数据中获取 Action 名称并据此解析出对应的方法，将通过 Model 绑定机制从当前请求上下文中提取相应的数据并调用 Action 方法生成对应的参数列表。所有这些后续操作都是间接地通过调用 Controller 的 `Execute` 方法来完成的。

---

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

定义在 `IController` 接口中的 `Execute` 是以同步的方式执行的。为了支持以异步方式对请求的处理，`IController` 接口的异步版本 `System.Web.Mvc.IAsyncController` 被定义出来。如下面的代码片段所示，实现了 `IAsyncController` 接口 Controller 的执行通过 `BeginExecute/EndExecute` 方法以异步的形式完成。

---

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

抽象类 `System.Web.Mvc.ControllerBase` 实现了 `IController` 接口。如下面的代码片段所示，`ControllerBase` 以“显式接口实现”的方式定义了 `Execute` 方法，该方法在内部直接调用受保护的 `Execute` 虚方法，而后者最终会调用抽象方法 `ExecuteCore` 方法。

---

```
public abstract class ControllerBase : IController
{
    //其他成员
    protected virtual void Execute(RequestContext requestContext);
    protected abstract void ExecuteCore();
    void IController.Execute(RequestContext requestContext);
}
```

```

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

```

ControllerBase 具有如下几个重要的属性：TempData、ViewBag 和 ViewData，它们用于存储从 Controller 向 View 传递的数据或者变量。其中 TempData 和 ViewData 具有基于字典的数据结构，Key 和 Value 分别表示变量的名称和值，两者的不同之处在于前者仅仅用于存储临时数据，并且设置的变量被第一次读取之后会被移除，换句话说通过 TempData 设置的变量只能被读取一次。ViewBag 和 ViewData 共享着相同的数据，它们之间的不同之处在于前者是一个动态对象，我们可以为其指定任意属性（动态属性名将作为数据字典的 Key）。

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

---

```

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

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

```

顾名思义，ControllerContext 就是基于某个 Controller 对象的上下文。从如上的代码可以看出一个 ControllerContext 对象实际上是对一个 Controller 对象和 RequestContext 的封装。这两个对象分别对应着 ControllerContext 中的同名属性，可以在构建 ControllerContext 的时候为调用的构造函数指定相应的参数来初始化它们。

通过 HttpContext 和 RouteData 属性返回的 HttpContextBase 和 RouteData 对象在默认情况下实际上就是 RequestContext 的核心组成部分。当 ControllerBase 的 Execute 方法被执行的时候，它会根据传入的 RequestContext 创建 ControllerContext 对象，后续的操作可以看成是在该上下文中进行。

通过 Visual Studio 的 Controller 创建向导创建的 Controller 类型实际上继承自抽象类 System.Web.Mvc.Controller，它是 ControllerBase 的子类。如下面的代码片段所示，除了直接继承 ControllerBase 之外，Controller 类型还显式地实现了 IController 和 IAsyncController 接口，以及代表 ASP.NET MVC 四大筛选器（AuthorizationFilter、ActionFilter、ResultFilter 和 ExceptionFilter）的 4 个接口（我们会在第 7 章“Action 的执行”中对筛选器进行详细介绍）。



---

```
public abstract class Controller :
    ControllerBase,
    IController,
    IAsyncController,
    IActionFilter,
    IAuthorizationFilter,
    IExceptionHandler,
    IResultFilter,
    IDisposable,
    ...
{
    //省略成员
}
```

### 同步还是异步

从抽象类 `Controller` 的定义可以看出它同时实现了 `IController` 和 `IAsyncController` 这两个接口，意味着它既可以采用同步的方式（调用 `Execute` 方法）执行，也可以采用异步的方式（调用 `BeginExecute/EndExecute` 方法）执行。但是即使执行 `BeginExecute/EndExecute` 方法，`Controller` 也不一定是以异步方式执行的。

如下面的代码片段所示，`Controller` 具有一个布尔类型的属性 `DisableAsyncSupport`，表示是否关闭对异步执行的支持。在默认的情况下该属性总是返回 `False`，即支持以异步方式执行 `Controller`。`BeginExecute` 方法会根据 `DisableAsyncSupport` 属性决定究竟是调用 `Execute` 方法以同步的方式执行，还是调用 `BeginExecuteCore/EndExecuteCore` 方法以异步的方式执行。换句话说，如果我们希望 `Controller` 总是以同步的方式来执行，可以将 `DisableAsyncSupport` 属性设置为 `True`。

---

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

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

}

现在我们通过一个简单的实例来演示属性 `DisableAsyncSupport` 对默认创建的 Controller 执行的影响。我们在一个 ASP.NET MVC 应用中定义了一个具有如下定义的默认 Home Controller，它重写了 `Execute`、`ExecuteCore`、`BeginExecute/EndExecute` 和 `BeginExecuteCore/EndExecuteCore` 六个方法，同时将相应的方法名写入响应并最终呈现在浏览器上。

---

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

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

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

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

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

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

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

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

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

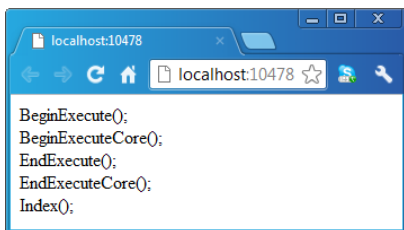


图 3-1 Controller 在默认情况下的异步执行方式

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

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

再次执行我们的程序将会得到如图 3-2 所示的输出结果, 可以看出由于 `HomeController` 间接地实现了 `IAsyncController` 接口, `Controller` 的执行总是以调用 `BeginExecute/EndExecute` 方法的方式来执行, 但是由于 `DisableAsyncSupport` 属性被设置为 `True`, `BeginExecute` 方法内部会以同步的方式调用 `Execute/ExecuteCore` 方法。(S302)

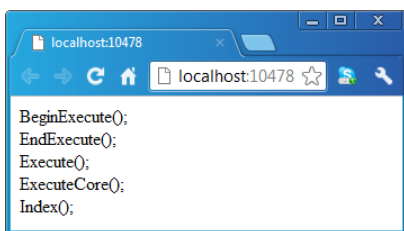


图 3-2 Controller 在 `DisableAsyncSupport` 属性为 `True` 的情况下的同步执行方式

ASP.NET MVC 应用编程接口中还定义了一个 `System.Web.Mvc.AsyncController` 类型, 从名称上看, `AsyncController` 是一个基于异步的 `Controller`, 但是这里的异步并不是指 `Controller` 的异步执行, 而是 `Action` 方法的异步执行。从如下的代码片段中可以看出, 这个直接继承自抽象类 `Controller` 的 `AsyncController` 是一个“空”类型(没有额外定义和重写基类的类型

成员)。在上一个版本中，以 XxxAsync/XxxCompleted 形式定义的异步 Action 方法均定义在继承自 AsyncController 的 Controller 类型中，考虑到向后兼容性，AsyncController 在新的版本中保留下来。

---

```
public abstract class AsyncController : Controller
{}
```

只有以传统方式（XxxAsync/XxxCompleted）定义的异步 Action 方法才需要定义在 AsyncController 中。ASP.NET MVC 4.0 提供了新的异步 Action 方法定义方式，使我们可以通过一个返回类型为 Task 的方法来定义以异步方式执行的 Action，这样的 Action 方法不需要定义在 AsyncController 中。

### 3.1.2 ControllerFactory

ASP.NET MVC 为 Controller 的激活定义相应的工厂，我们将其统称为 ControllerFactory，所有的 ControllerFactory 实现了接口 System.Web.Mvc.IControllerFactory 接口。如下面的代码片段所示，Controller 对象的激活最终通过 IControllerFactory 的 CreateController 方法来完成，该方法的两个参数分别表示当前请求上下文和从路由信息中获取的 Controller 的名称。

---

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

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

除了负责创建 Controller 处理请求之外，ControllerFactory 还需要在完成请求处理之后释放 Controller，对激活 Controller 对象的释放定义在 ReleaseController 方法中。IControllerFactory 的另一个方法 GetControllerSessionBehavior 返回一个 System.Web.SessionState.SessionStateBehavior 枚举。熟悉 ASP.NET 的读者对 SessionStateBehavior 应该不会感到陌生，它用于表示请求处理过程中会话状态支持的模式，它的四个枚举值分别具有如下的含义。

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

对于 Default 选项来说, ASP.NET 通过映射的 `HttpHandler` 类型是否实现了相关接口来决定具体的会话状态控制行为。在 `System.Web.SessionState` 命名空间下定义了 `IRequiresSessionState` 和 `IReadOnlySessionState` 接口, 如下面的代码片段所示, 这两个都是不具有任何成员的空接口 (我们一般称之为标记接口), 而 `IReadOnlySessionState` 继承自 `IRequiresSessionState`。如果 `HttpHandler` 实现了接口 `IReadOnlySessionState`, 则意味着采用 `ReadOnly` 模式, 如果只实现了 `IRequiresSessionState` 则采用 `Required` 模式。

---

```
public interface IRequiresSessionState
{
}

public interface IReadOnlySessionState : IRequiresSessionState
{
}
```

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

---

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

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

### 3.1.3 ControllerBuilder

用于激活 `Controller` 对象的 `ControllerFactory` 最终通过 `System.Web.Mvc.ControllerBuilder` 注册到 ASP.NET MVC 应用中。如下面的代码所示, `ControllerBuilder` 定义了一个静态只读属性 `Current` 返回当前 `ControllerBuilder` 对象, 这是针对整个 Web 应用的全局对象。两个 `SetControllerFactory` 方法重载用于注册 `ControllerFactory` 的类型或者实例, 而 `GetControllerFactory` 方法返回一个具体的 `ControllerFactory` 对象。

---

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

    public void SetControllerFactory(Type controllerFactoryType);
}
```

```

public void SetControllerFactory(IControllerFactory controllerFactory);

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

```

具体来说，如果我们注册的是 `ControllerFactory` 的类型，那么 `GetControllerFactory` 在执行的时候会通过对注册类型的反射（调用 `Activator` 的静态方法 `CreateInstance`）来创建具体的 `ControllerFactory`（系统不会对创建的 `Controller` 进行缓存）。如果注册的是一个具体的 `ControllerFactory` 对象，该对象直接从 `GetControllerFactory` 返回。

通过第2章“URL路由”的介绍我们知道，被 ASP.NET 路由系统进行拦截处理后会生成一个用于封装路由信息的 `RouteData` 对象，而目标 `Controller` 的名称就包含在通过该 `RouteData` 的 `Values` 属性表示的 `RouteValueDictionary` 对象中，对应的 `Key` 为“controller”。而在默认的情况下，这个作为路由数据的名称只能帮助我们解析出 `Controller` 的类型名称，如果在不同的命名空间下定义了多个同名的 `Controller` 类，会导致激活系统无法确定具体的 `Controller` 的类型从而抛出异常。

为了解决这个问题，我们必须为定义了同名 `Controller` 类型的命名空间设置不同的优先级。具体来说有两种提升命名空间优先级的方式。第一种方式就是在调用 `RouteCollection` 如下所示的扩展方法 `MapRoute` 时指定一个命名空间的列表。通过第2章“URL路由”的介绍我们知道，通过这种方式指定的命名空间列表会保存在 `Route` 对象的 `DataTokens` 属性表示的 `RouteValueDictionary` 字典中，对应的 `Key` 为“Namespaces”。

---

```

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

```

另一种提升命名空间优先级的方式就是将其添加到当前的 `ControllerBuilder` 中的默认命名空间列表中。从上面的给出的 `ControllerBuilder` 的定义可以看出，它具有一个 `HashSet<string>` 类型的只读属性 `DefaultNamespaces` 代表了这么一个默认命名空间列表。对于这两种不同的命名空间优先级提升方式，前者（通过路由注册）指定命名空间具有更高的优先级。

### 实例演示：如何提升命名空间的优先级（S303，S304，S305）

为了让读者对如何提升命名空间优先级有一个深刻的印象，我们来进行一个简单的实例演示。在一个 ASP.NET MVC 应用创建两个同名的 `HomeController` 类，如下面的代码片段所

示，这两个 HomeController 类分别定义在命名空间 Artech.MvcApp 和 Artech.MvcApp.Controllers 之中，而 Index 操作返回的是一个将 Controller 类型全名作为内容的 System.Web.Mvc.ContentResult 对象。

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

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

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

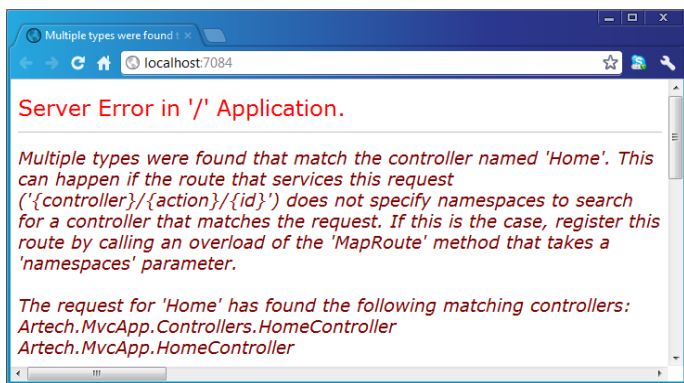


图 3-3 具有多个匹配 Controller 导致的异常

目前定义了 HomeController 的两个命名空间具有相同的优先级，现在将其中一个定义在当前 ControllerBuilder 的默认命名空间列表中以提升匹配优先级。如下面的代码片段所示，在 Global.asax 的 Application\_Start 方法中，将命名空间 “Artech.MvcApp.Controllers” 添加到当前 ControllerBuilder 的 DefaultNamespaces 属性所示的命名空间列表中。

---

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

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

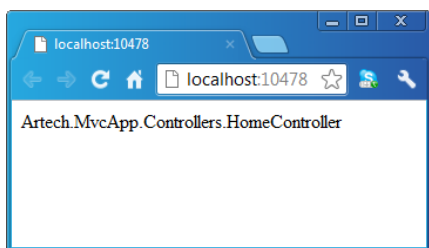


图 3-4 通过 ControllerBuilder 提升命名空间匹配优先级

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

---

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

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



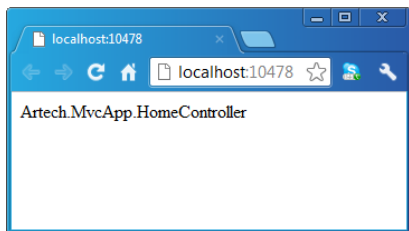


图 3-5 在路由注册时指定的命名空间具有更高的匹配优先级

在路由注册时指定的命名空间比当前 `ControllerBuilder` 的默认命名空间具有更高的匹配优先级，但是对于这两个集合中的所有命名空间却具有相同的匹配优先级。换句话说，用于辅助解析 `Controller` 类型的命名空间分为三个梯队，分别简称为路由命名空间、`ControllerBuilder` 命名空间和 `Controller` 类型命名空间。如果前一个梯队不能正确解析出目标 `Controller` 的类型，则后一个梯队的命名空间将作为后备，反之，如果根据某个梯队的命名空间进行解析得到多个匹配的 `Controller` 类型，会直接抛出异常。

### 针对 Area 的路由对象的命名空间

针对某个 Area 的路由映射是通过相应的 `AreaRegistration` 进行注册的，具体来说是在 `AreaRegistration` 的 `RegisterArea` 方法中调用 `AreaRegistrationContext` 对象的 `MapRoute` 方法进行注册的。如果在调用 `MapRoute` 方法中指定了表示命名空间的字符串，它将自动作为注册的路由对象的命名空间，否则会将 `AreaRegistration` 的命名空间加上“.”后缀得到的字符串作为路由对象的命名空间。

这里所说的“路由对象的命名空间”存在于 `Route` 对象的 `DataTokens` 属性表示的 `RouteValueDictionary` 对象中，对应的 `Key` 为“Namespaces”，`Value` 就是一个包含字符串数组的命名空间列表。通过第 2 章“URL 路由”的介绍，`Route` 对象的 `DataTokens` 属性包含的变量会转移到由它生成的 `RouteData` 的同名属性中。

除此之外，在调用 `AreaRegistrationContext` 的 `MapRoute` 方法时还会在注册 `Route` 对象的 `DataTokens` 属性中添加一个 `Key` 为“UseNamespaceFallback”的条目，它表示是否采用后备命名空间对 `Controller` 类型进行解析。如果注册的路由对象具有命名空间（调用 `MapRoute` 方法时指定了命名空间或者对应的 `AreaRegistration` 类型定义在某个命名空间下），该条目的值为 `False`，否则为 `True`。该条目同样反映在通过该 `Route` 对象生成的 `RouteData` 对象的 `DataTokens` 属性中。

在解析 `Controller` 真实类型的过程中，会先使用 `RouteData` 包含的命名空间。如果解析失败，则通过由 `RouteData` 的 `DataTokens` 属性得到的这个名为“UseNamespaceFallback”的变量值来判断是否使用“后备”命名空间进行解析。具体来说，如果该值为 `True` 或者不存在，则先通过当前 `ControllerBuilder` 的命名空间解析，如果失败则忽略命名空间直接采用类

型名称进行匹配，否则会因找不到匹配的 Controller 而直接抛出异常。

我们通过具体的例子来说明这个问题。在一个 ASP.NET MVC 应用中通过 Area 添加向导创建一个名称为 Admin 的 Area，此时 IDE 会默认为我们添加了如下一个 AdminAreaRegistration 类型。

---

```

NamespaceMvcApp.Areas.Admin
{
    public class AdminAreaRegistration : AreaRegistration
    {
        public override string AreaName
        {
            get{return "Admin";}
        }
        public override void RegisterArea(AreaRegistrationContext context)
        {
            context.MapRoute(
                "Admin_default",
                "Admin/{controller}/{action}/{id}",
                new { action = "Index", id = UrlParameter.Optional }
            );
        }
    }
}

```

AdminAreaRegistration 类型定义在命名空间 MvcApp.Areas.Admin 中。现在我们在该 Area 中添加如下一个 HomeController，在默认的动作方法 Index 中，我们从当前 RouteData 的 DataTokens 中提取这个名为“UseNamespaceFallback”的变量值，并将它和解析出来的 Controller 类型名称写入当前 HttpResponse 而最终呈现在客户端浏览器中。在默认情况下，添加的 HomeController 类型被定义在 MvcApp.Areas.Admin.Controllers 命名空间下，现在我们刻意将命名空间改为 MvcApp.Areas.Controllers。

---

```

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

```

现在我们在浏览器中通过匹配的 URL (/Admin/Home/Index) 来访问 Area 为 Admin 的 HomeController 的 Index 操作，会得到如图 3-6 所示的 HTTP 状态为“404, Not Found”的错误。这就是因为在对 Controller 类型进行解析的时候是严格按照对应的 AreaRegistration 所在的命名空间来进行的，很显然在这个范围内是不可能找得到对应的 Controller 类型的。

(S306)

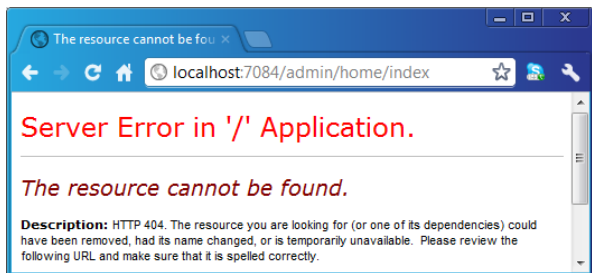


图 3-6 Controller 和 AreaRegistration 命名空间不匹配导致的 404 错误

但是如果我们去掉 `AdminAreaRegistration` 的命名空间，那么将会导致路由变量 `UseNamespaceFallback` 的值变为 `True`，这会促使 Controller 激活系统选择“后备”的命名空间。由于整个 Web 应用中仅仅定义了唯一匹配的 `MvcApp.Areas.Controllers.HomeController`，很显然这个 Controller 会被激活，如图 3-7 所示的程序运行结果也说明了这一点。（S307）

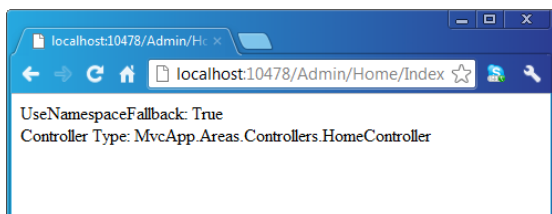


图 3-7 去掉 `AdminAreaRegistration` 命名空间以采用后备命名空间

### 3.1.4 Controller 的激活与 URL 路由

ASP.NET 路由系统是 HTTP 请求抵达服务端的第一道屏障，它根据注册的路由规则对拦截的请求进行匹配并解析包含目标 Controller 和 Action 名称的路由信息。而当前 `ControllerBuilder` 具有用于激活 Controller 对象的 `ControllerFactory`，现在看看两者是如何结合起来的。

通过第 2 章“URL 路由”的介绍我们知道，ASP.NET 路由系统的核心是一个叫做 `UrlRoutingModule` 的 `HttpModule`，路由的实现是它通过注册代表 `HttpApplication` 的 `PostResolveRequestCache` 事件对 `HttpHandler` 的动态映射来实现的。具体来说，它通过以 `RouteTable` 的静态属性 `Routes` 代表的全局路由表对请求进行匹配并得到一个 `RouteData` 对象。`RouteData` 具有一个实现了接口 `IRouteHandler` 的属性 `RouteHandler`，通过该属性的 `GetHttpHandler` 方法可以得到最终被映射到当前请求的 `HttpHandler` 对象。

对于 ASP.NET MVC 应用来说，`RouteData` 的 `RouteHandler` 属性类型为 `MvcRouteHandler`，实现在 `MvcRouteHandler` 中的 `HttpHandler` 提供机制基本上（不是完全等同）可以通过如下的代码来体现。`MvcRouteHandler` 维护着一个 `ControllerFactory` 对象，该对象可以在构造函数

数中指定，如果没有显示指定则直接通过调用当前 `ControllerBuilder` 的 `GetControllerFactory` 方法获取。

---

```
public class MvcRouteHandler : IRouteHandler
{
    private IControllerFactory _controllerFactory;
    public MvcRouteHandler(): this(ControllerBuilder.Current
        .GetControllerFactory())
    { }
    public MvcRouteHandler(IControllerFactory controllerFactory)
    {
        _controllerFactory = controllerFactory;
    }
    IHttpHandler IRouteHandler.GetHttpHandler(RequestContext requestContext)
    {
        string controllerName = (string)requestContext.RouteData
            .GetRequiredString("controller");
        SessionStateBehavior sessionStateBehavior = _controllerFactory
            .GetControllerSessionBehavior(requestContext, controllerName);
        requestContext.HttpContext.SetSessionStateBehavior(sessionStateBehavior);

        return new MvcHandler(requestContext);
    }
}
```

在用于提供 `HttpHandler` 的 `GetHttpHandler` 方法中，除了返回一个实现了 `IHttpHandler` 接口的 `MvcHandler` 对象之外，还需要对当前 HTTP 上下文的会话状态行为模式进行设置。具体的实现是：先通过包含在 `RequestContext` 的 `RouteData` 对象得到 `Controller` 的名称，该名称连同 `RequestContext` 对象一起传入 `ControllerFactory` 的 `GetControllerSessionBehavior` 方法得到一个类型为 `SessionStateBehavior` 的枚举。最后通过 `RequestContext` 得到当前 HTTP 上下文（实际上是一个 `HttpContextWrapper` 对象），并调用其 `SetSessionStateBehavior` 方法对会话状态行为进行设置。

通过第2章“URL 路由”的介绍我们知道，`RouteData` 中的 `RouteHandler` 属性最初来源于对应的路由对象，而当我们调用 `RouteCollection` 的扩展方法 `MapRoute` 方法时注册的 `Route` 对象对应的 `RouteHandler` 是一个 `MvcRouteHandler` 对象。由于在创建 `MvcRouteHandler` 对象时并没有显式指定 `ControllerFactory`，所以通过当前 `ControllerBuilder` 的 `GetControllerFactory` 方法得到的 `ControllerFactory` 默认被使用。

通过当前 `ControllerBuilder` 的 `GetControllerFactory` 方法得到的 `ControllerFactory` 仅仅用于获取会话状态行为模式，而 `MvcHandler` 真正将它用于创建 `Controller`。如下的代码片段基本上体现了 `MvcHandler` 的定义，它对请求处理的逻辑定义在 `BeginProcessRequest` 方法中。

---

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

    public bool IsReusable
```

```

    {
        get { return false; }
    }

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

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

```

由于 `MvcHandler` 同时实现了 `IHandler` 和 `IHandlerAsync` 接口，所以它总是以异步的方式被执行（调用 `BeginProcessRequest/EndProcessRequest` 方法）。`BeginProcessRequest` 方法通过 `RequestContext` 对象得到目标 `Controller` 的名称，然后利用当前 `ControllerBuilder` 创建的 `ControllerFactory` 激活 `Controller` 对象。如果 `Controller` 类型实现了 `IAsyncController` 接口，则以异步的方式执行 `Controller`，否则采用同步执行方式。在被激活 `Controller` 对象被执行之后，`MvcHandler` 会调用 `ControllerFactory` 的 `ReleaseController` 对其进行释放清理工作。

## 3.2 默认实现

Controller 激活系统最终通过注册的 `ControllerFactory` 创建相应的 `Controller` 对象，如果没有对 `ControllerFactory` 类型或者实例进行注册（通过调用当前 `ControllerBuilder` 的 `SetControllerFactory` 方法），默认使用的 `ControllerFactory` 类型为 `System.Web.Mvc.DefaultControllerFactory`。我们现在就来讨论实现在 `DefaultControllerFactory` 中的默认 `Controller` 激活机制。

### 3.2.1 Controller 类型的解析

激活目标 `Controller` 对象的前提是能够正确解析出 `Controller` 类型。对于 `DefaultControllerFactory` 来说，用于解析目标 `Controller` 类型的辅助信息包括：通过与当前请求匹配的路由对象生成的 `RouteData`（其中包含 `Controller` 的名称和命名空间）和包含在当前 `ControllerBuilder` 中的命名空间。很多读者可能首先想到的是通过 `Controller` 名称得到对应的类型，并通过命名空间组成 `Controller` 类型的全名，最后遍历所有程序集并以此名称去加载相应的类型即可。

这貌似是一个不错的解决方案，实际上则完全行不通。不要忘了作为请求地址 `URL` 一部分的 `Controller` 名称是不区分大小写的，而类型名称则是大小写敏感的。此外，不论是注册路由时指定的命名空间还是当前 `ControllerBuilder` 的默认命名空间，有可能包含统配符（\*）。由于我们不能通过给定的 `Controller` 名称和命名空间得到 `Controller` 的真实类型名称，自然就不可能通过名称去解析 `Controller` 的类型了。

`ASP.NET MVC` 的 `Controller` 激活系统则反其道而行之，它先遍历通过 `BuildManager` 的静态方法 `GetReferencedAssemblies` 方法得到所有引用程序集，通过反射的方式得到定义在它们中的所有实现了接口 `IController` 的类型，最后通过 `Controller` 的名称和命名空间作为匹配条件去选择对应的 `Controller` 类型。

#### 实例演示：创建一个自定义 `ControllerFactory` 模拟 `Controller` 默认激活机制（S308）

为了让读者对默认采用的 `Controller` 激活机制，尤其是 `Controller` 类型的解析机制有一个深刻的认识，通过一个自定义的 `ControllerFactory` 来模拟其中的实现。由于采用反射的方式来创建 `Controller` 对象，所以将该自定义 `ControllerFactory` 起名为 `ReflectedControllerFactory`。

```
public class ReflectedControllerFactory : IControllerFactory
{
    //其他成员
    private static List<Type> controllerTypes;
    static ReflectedControllerFactory()
    {
```

```

        controllerTypes = new List<Type>();
        foreach (Assembly assembly in BuildManager.GetReferencedAssemblies())
        {
            controllerTypes.AddRange(assembly.GetTypes().Where(
                type => typeof(IController).IsAssignableFrom(type));
        }
    }

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

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

    private Type GetControllerType(IEnumerable<string> namespaces,
        Type[] controllerTypes)
    {
        var types = (from type in controllerTypes
            where namespaces.Any(ns => IsNamespaceMatch(
                ns, type.Namespace))
            select type).ToArray();
        switch (types.Length)
        {
            case 0: return null;
            case 1: return types[0];
            default: throw new InvalidOperationException("具有多个匹配的 Controller
                类型");
        }
    }

    protected virtual Type GetControllerType(RouteData routeData,
        string controllerName)

```

```

    {
        //省略实现
    }
}

```

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

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

`ReflectedControllerFactory` 中定义了两个辅助方法, 其中 `IsNamespaceMatch` 用于判断 `Controller` 类型真正的命名空间是否与指定的命名空间(可能包含统配符)相匹配, 进行字符比较是忽略大小写的。私有方法 `GetControllerType` 根据指定的命名空间列表和类型名称匹配的类型数组得到一个完全匹配的 `Controller` 类型。如果得到多个匹配的类型, 直接抛出 `InvalidOperationException` 异常, 并提示具有多个匹配的 `Controller` 类型, 如果找不到匹配类型, 则返回 `Null`。

在如下所示的用于解析 `Controller` 类型的 `GetControllerType` 方法中, 从预先得到的所有 `Controller` 类型列表中筛选出类型名称与传入的 `Controller` 名称相匹配的类型。首先通过路由对象的命名空间对之前得到的类型列表进行进一步筛选, 如果能够找到一个唯一的类型, 则直接将其作为 `Controller` 的类型返回。为了确定是否采用后备命名空间对 `Controller` 类型进行解析, 可以从作为参数的 `RouteData` 对象中得到其 `DataTokens` 属性, 并从中获取路由变量 `UseNamespaceFallback` 的值。如果该路由变量存在并且值为 `False`, 则直接返回 `Null`。

---

```

public class ReflectedControllerFactory : IControllerFactory
{
    //其他成员
    protected virtual Type GetControllerType (RouteData routeData,
        string controllerName)
    {
        //根据类型名称筛选
        var types = controllerTypes.Where(type => string.Compare(
            controllerName + "Controller", type.Name, true) == 0).ToArray();
        if (types.Length == 0)
        {
            return null;
        }

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

```



```

if (null != contrllerType)
{
    return contrllerType;
}

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

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

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

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

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

```

如果 RouteData 的 DataTokens 中不存在这样一个 UseNamespaceFallback 路由变量, 或者它的值为 True, 则先采用当前 ControllerBuilder 的默认命名空间列表进一步对 Controller 类型进行解析, 如果存在唯一的类型则直接当作目标 Controller 类型返回。如果通过两组命名空间均不能得到一个匹配的 ControllerType, 并且只存在唯一一个与传入的 Controller 名称相匹配的类型, 则直接将该类型作为目标 Controller 返回。如果这样的类型具有多个, 则直接抛出 InvalidOperationException 异常。

## 3.2.2 Controller 类型的缓存

为了避免频繁地遍历所有程序集对目标 Controller 类型进行解析, ASP.NET MVC 对解析出来的 Controller 类型进行了缓存以提升性能。与针对用于 Area 注册的 AreaRegistration

类型的缓存类似，Controller 激活系统同样采用基于文件的缓存策略，用于保存 Controller 类型列表的名为 MVCControllerTypeCache.xml 的文件保存在 ASP.NET 的临时目录下面。具体的路径如下：

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

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

当接收到 Web 应用被启动后的第一个请求时，Controller 激活系统会读取这个用于缓存所有 Controller 类型列表的 ControllerTypeCache.xml 文件并反序列化成一个 List<Type>对象。只有在该列表为空的时候才会通过遍历程序集和反射的方式得到所有实现了接口 IController 的类型，而被解析出来的 Controller 类型重新被写入这个缓存文件中。这个通过读取缓存文件或者重新解析出来的 Controller 类型列表被保存到内存中，在 Web 应用活动期间内被 Controller 激活系统使用。

下面的 XML 片段反映了这个用于 Controller 类型列表缓存的 MVC-ControllerTypeCache.xml 文件的结构，从中可以看出它包含了所有的 Controller 类型的全名和所在的程序集和托管模块的名称。

---

```
<?xml version="1.0" encoding="utf-8"?>
<!--This file is automatically generated. Please do not modify the contents of
this file.-->
<typeCache lastModified="3/22/2012 1:18:49 PM"
    mvcVersionId="80365b23-7a1d-42b2-9e7d-cc6f5694c6d1">
  <assembly name="Artech.Admin, Version=1.0.0.0, Culture=neutral,
    PublicKeyToken=null">
    <module versionId="eb343e3f-2d63-4665-a12a-29fb30dceed">
      <type>Artech.Admin.HomeController</type>
      <type>Artech.Admin.EmployeeController </type>
    </module>
  </assembly>
  <assembly name="Artech.Portal, Version=1.0.0.0, Culture=neutral,
    PublicKeyToken=null">
    <module versionId=" 3717F116-35EE-425F-A1AE-EB4267497D8C">
      <type>Artech.Portal.Controllers.HomeController</type>
      <type>Artech.Portal.ProductsController</type>
    </module>
  </assembly>
</typeCache>
```

### 3.2.3 Controller 的释放和会话状态行为的控制

作为激活 Controller 对象的 ControllerFactory 不仅仅用于创建目标 Controller 对象，还具有两个额外的功能，即通过 ReleaseController 方法对激活的 Controller 对象进行释放和回收，以及通过调用 GetControllerSessionBehavior 方法返回用于控制当前会话状态行为的 SessionStateBehavior 枚举对象。

对于默认使用的 DefaultControllerFactory 来说，它对 Controller 对象的释放操作很简单，即如果 Controller 类型实现了 IDisposable 接口，则直接调用其 Dispose 方法即可。我们将这个逻辑也实现在了我们自定义的 ReflectedControllerFactory 中。

---

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

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

---

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

可以通过在 Controller 类型上应用 System.Web.Mvc.SessionStateAttribute 特性来具体控制会话状态行为。如下面的代码片段所示，SessionStateAttribute 具有一个 SessionStateBehavior 类型的只读属性 Behavior 用于返回具体行为设置的会话状态行为选项，该属性是在构造函数中被初始化的。

---

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple = false, Inherited = true)]
public sealed class SessionStateAttribute : Attribute
{
```

```

public SessionStateAttribute(SessionStateBehavior behavior);
public SessionStateBehavior Behavior { get; }
}

```

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

---

```

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

```

## 3.3 IoC 的应用

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

### 3.3.1 从 Unity 来认识 IoC

有时又将 IoC 称为依赖注入 (Dependency Injection, DI)。所谓依赖注入，就是由外部容器在运行时动态地将依赖的对象注入到组件之中。Martin Fowler 在那篇著名的文章 *Inversion of Control Containers and the Dependency Injection pattern* 中将具体的依赖注入划分为三种形式，即构造器注入、属性（设置）注入和接口注入，而我个人习惯将其划分为一种（类型）匹配和三种注入。

- **类型匹配 (Type Mapping)**: 虽然我们通过接口 (或者抽象类) 来进行服务调用, 但是服务本身还是实现在某个具体的服务类型中, 这就需要某个类型注册机制来解决服务接口和服务类型之间的匹配关系。
- **构造器注入 (Constructor Injection)**: IoC 容器会智能地选择和调用适合的构造函数以创建依赖的对象。如果被选择的构造函数具有相应的参数, IoC 容器在调用构造函数之前解析注册的依赖关系并自行获得相应参数对象。
- **属性注入 (Property Injection)**: 如果需要使用到被依赖对象的某个属性, 在被依赖对象被创建之后, IoC 容器会自动初始化该属性。
- **方法注入 (Method Injection)**: 如果被依赖对象需要调用某个方法进行相应的初始化, 在该对象创建之后, IoC 容器会自动调用该方法。

开源社区具有很有流行的 IoC 框架, 如 Castle Windsor、Unity、Spring.NET、StructureMap 和 Ninject 等。Unity 是微软 Patterns& Practices 部门开发的一个轻量级的 IoC 框架, 该项目在 Codeplex 上的地址为 <http://unity.codeplex.com/>, 我们可以下载相应的安装包和开发文档。在本书出版之时, Unity 的最新版本为 2.1。出于篇幅的限制, 我们不可能对 Unity 进行详细的讨论, 但是为了让读者了解 IoC 在 Unity 中的实现, 我们写了一个简单的程序。

创建一个控制台程序, 定义如下几个接口 (IA、IB、IC 和 ID) 和它们各自的实现类 (A、B、C、D)。在类型 A 中定义了 B、C 和 D3 个属性, 其类型分别为接口 IB、IC 和 ID。属性 B 在函数中被初始化, 意味着它会以构造器注入的方式被初始化; 属性 C 上应用了 `Microsoft.Practices.Unity.DependencyAttribute` 特性, 意味着这是一个需要以属性注入方式被初始化的依赖属性; 属性 D 则通过方法 `Initialize` 初始化, 该方法上应用了特性 `Microsoft.Practices.Unity.InjectionMethodAttribute`, 意味着这是一个注入方法, 它会在 A 对象被 IoC 容器创建的时候会被自动调用。

---

```
namespace UnityDemo
{
    public interface IA { }
    public interface IB { }
    public interface IC { }
    public interface ID { }

    public class A : IA
    {
        public IB B { get; set; }
        [Dependency]
        public IC C { get; set; }
        public ID D { get; set; }

        public A(IB b)
        {
            this.B = b;
        }
        [InjectionMethod]
        public void Initialize(ID d)
        {
```

```

        this.D = d;
    }
}
public class B: IB{}
public class C: IC{}
public class D: ID{}
}

```

然后为该应用添加一个配置文件，并定义如下一段关于 Unity 的配置。这段配置定义了一个名称为 defaultContainer 的 Unity 容器，并在其中完成了上面定义的接口和对应实现类之间映射的类型匹配。

---

```

<configuration>
  <configSections>
    <section name="unity"
      type="Microsoft.Practices.Unity.Configuration.UnityConfigurationSection,
        Microsoft.Practices.Unity.Configuration"/>
  </configSections>
  <unity>
    <containers>
      <container name="defaultContainer">
        <register type="UnityDemo.IA, UnityDemo" mapTo="UnityDemo.A, UnityDemo"/>
        <register type="UnityDemo.IB, UnityDemo" mapTo="UnityDemo.B, UnityDemo"/>
        <register type="UnityDemo.IC, UnityDemo" mapTo="UnityDemo.C, UnityDemo"/>
        <register type="UnityDemo.ID, UnityDemo" mapTo="UnityDemo.D, UnityDemo"/>
      </container>
    </containers>
  </unity>
</configuration>

```

最后在作为程序入口的 Main 方法中创建一个代表 IoC 容器的 UnityContainer 对象，并加载配置信息对其进行初始化。然后调用它的泛型方法 Resolve 创建一个实现了泛型接口 IA 的对象。最后将返回对象转变成类型 A，并检验其 B、C 和 D 属性是否为 Null。

---

```

static void Main(string[] args)
{
    IUnityContainer container = new UnityContainer();
    UnityConfigurationSection configuration =
        ConfigurationManager.GetSection(UnityConfigurationSection.SectionName)
        as UnityConfigurationSection;
    configuration.Configure(container, "defaultContainer");
    A a = container.Resolve<IA>() as A;
    if (null != a)
    {
        Console.WriteLine("a.B == null ? {0}", a.B == null ? "Yes" : "No");
        Console.WriteLine("a.C == null ? {0}", a.C == null ? "Yes" : "No");
        Console.WriteLine("a.D == null ? {0}", a.D == null ? "Yes" : "No");
    }
}

```

从如下给出的执行结果可以得到这样的结论：通过 Resolve<IA>方法返回的是一个类型为 A 的对象，该对象的三个属性被进行了有效的初始化。这个简单的程序分别体现了接口注入（通过相应的接口根据配置解析出相应的实现类型）、构造器注入（属性 B）、属性注入（属性 C）和方法注入（属性 D）。（S309）

```

a.B == null ? No
a.C == null ? No
a.D == null ? No

```

### 3.3.2 Controller 与 Model 的分离

在第 1 章“ASP.NET + MVC”中我们谈到过 ASP.NET MVC 是基于 MVC 的变体 Model2 设计的。ASP.NET MVC 所谓的 Model 仅仅表示绑定到 View 上的数据，我们一般称之为 View Model。而真正的 Model 一般意义上指维护应用状态和提供业务功能操作的领域模型，或者是针对业务层的入口或者业务服务的代理。真正的 MVC 在 ASP.NET MVC 中的体现如图 3-8 所示。

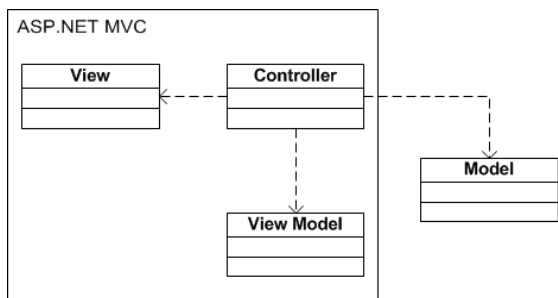


图 3-8 ASP.NET MVC + Model

对于一个 ASP.NET MVC 应用来说，用户交互请求直接发送给 Controller。如果涉及针对某项业务功能的调用，Controller 会直接调用 Model。如果需要呈现业务数据，Controller 会通过 Model 获取相应业务数据并转换成 View Model，最终通过 View 呈现出来，这样的交互协议方式反映了 Controller 针对 Model 的直接依赖。

如果我们在 Controller 激活系统中引入 IoC，并采用 IoC 的方式提供用于处理请求的 Controller 对象，那么 Controller 和 Model 之间的依赖程度在很大程度上被降低了，甚至可以像图 3-9 所示的一样，以接口的方式对 Model 进行抽象，让 Controller 依赖于这个抽象化的 Model 接口，而不是具体的 Model 实现。

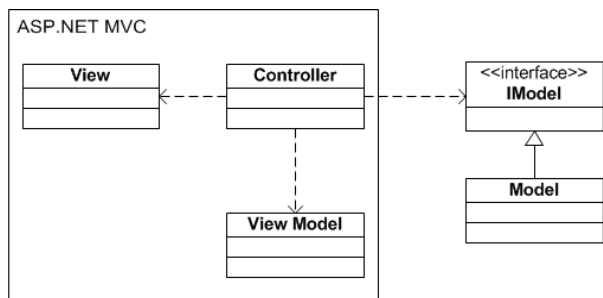


图 3-9 ASP.NET MVC + IModel + Model

### 3.3.3 基于 IoC 的 ControllerFactory

ASP.NET MVC 的 Controller 激活系统最终通过 ControllerFactory 来创建目标 Controller 对象，要将 IoC 引入 ASP.NET MVC 并通过对应的 IoC 容器实现对目标 Controller 的激活，我们很自然地会想到自定义一个基于 IoC 的 ControllerFactory。

对于自定义 ControllerFactory，可以直接实现 IControllerFactory 接口创建一个全新的 ControllerFactory 类型，这要实现包括 Controller 类型的解析、Controller 实例的创建与释放以及会话状态行为选项的获取在内的所有功能。一般来说，Controller 实例的创建才需要 IoC 容器的控制，为了避免重新实现其他的功能，可以直接继承 DefaultControllerFactory，重写 Controller 实例创建的逻辑。

#### 实例演示：创建基于 Unity 的 ControllerFactory ( S310 )

现在我们通过一个简单的实例演示如何通过自定义 ControllerFactory 利用 Unity 进行 Controller 的激活。为了避免针对 Controller 类型解析、会话状态行为选项的获取和对 Controller 对象的释放逻辑的重复定义，我们直接继承 DefaultControllerFactory。将该自定义 ControllerFactory 命名为 UnityControllerFactory。如下面的代码片段所示，UnityControllerFactory 仅仅重写了受保护的虚方法 GetControllerInstance，将成功解析的 Controller 类型作为调用 UnityContainer 的 Resolve 方法的参数，而返回值就是需要被激活的 Controller 实例。

```
public class UnityControllerFactory: DefaultControllerFactory
{
    public IUnityContainer UnityContainer { get; private set; }
    public UnityControllerFactory(IUnityContainer unityContainer)
    {
        this.UnityContainer = unityContainer;
    }
    protected override IController GetControllerInstance(
        RequestContext requestContext, Type controllerType)
    {
        if (null == controllerType)
        {
            return null;
        }
        return (IController)this.UnityContainer.Resolve(controllerType);
    }
}
```

整个自定义的 UnityControllerFactory 就这么简单。为了演示 IoC 在它身上的体现，我们在一个简单的 ASP.MVC 实例中来使用它。我们沿用在第 2 章“URL 路由”中使用过的关于“员工管理”的场景，如图 3-10 所示，本实例由两个页面（对应着两个 View）组成，一个用于显示员工列表，另一个用于显示基于某个员工的详细信息。



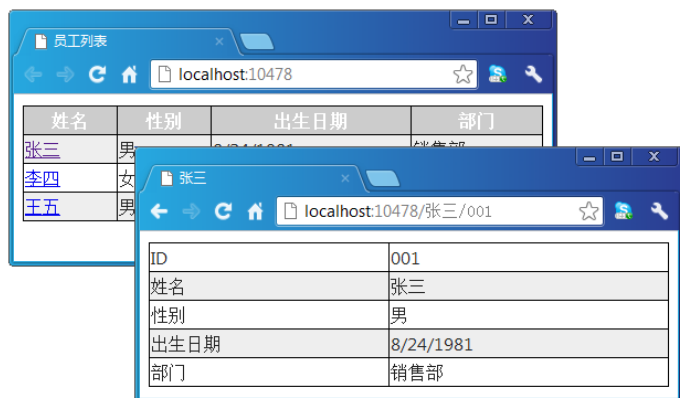


图 3-10 员工列表和员工详细信息页面

在一个 ASP.NET MVC 应用中添加对 Unity 的程序集 Microsoft.Practices.Unity.dll 的引用（如果读者不想安装 Unity，可以通过下载本实例的源代码的方式获取该程序集），然后在 Models 目录下定义如下一个表示员工信息的 Employee 类型。

```
public class Employee
{
    [DisplayName="ID"]
    public string Id { get; private set; }

    [DisplayName = "姓名"]
    public string Name { get; private set; }

    [DisplayName = "性别"]
    public string Gender { get; private set; }

    [DisplayName = "出生日期"]
    [DataType(DataType.Date)]
    public DateTime BirthDate { get; private set; }

    [DisplayName = "部门"]
    public string Department { get; private set; }

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

我们创建一个 EmployeeRepository 对象来进行数据的获取，并为它定义了对应的接口 IEmployeeRepository。如下面的代码片段所示，IEmployeeRepository 仅仅具有一个返回

Employee 列表的唯一方法 `GetEmployees`，用于获取指定 ID 的员工信息。如果指定的 ID 为空，则返回所有员工列表。`EmployeeRepository` 直接利用一个静态字段模拟对数据的存储。

---

```
public interface IEmployeeRepository
{
    IEnumerable<Employee> GetEmployees(string id = "");
}

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

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

    public IEnumerable<Employee> GetEmployees(string id = "")
    {
        return employees.Where(e => e.Id == id || string.IsNullOrEmpty(id));
    }
}
```

我们创建了一个具有如下定义的 `EmployeeController`，它具有一个类型为 `IEmployeeRepository` 的属性 `Repository`，应用在上面的 `DependencyAttribute` 特性告诉我们这是一个“依赖属性”。当我们采用 `UnityContainer` 来激活 `EmployeeController` 对象的时候，会根据注册的类型映射来实例化一个实现了 `IEmployeeRepository` 的类型的实例来初始化该属性。

---

```
public class EmployeeController : Controller
{
    [Dependency]
    public IEmployeeRepository Repository { get; set; }

    public ActionResult GetAllEmployees()
    {
        var employees = this.Repository.GetEmployees();
        return View("EmployeeList", employees);
    }

    public ActionResult GetEmployeeById(string id)
    {
        Employee employee = this.Repository.GetEmployees(id).FirstOrDefault();
        if (null == employee)
        {

```

```

        throw new HttpException(404, string.Format("ID 为{0}的员工不存在", id));
    }
    return View("Employee", employee);
}
}

```

EmployeeController 定义了两个基本的 Action 方法。GetAllEmployees 通过 Repository 获取所有员工列表并将其通过名为 EmployeeList 的 View 呈现出来。另一个 Action 方法 GetEmployeeById 根据指定的 ID 获取相应的员工信息，最终用于呈现单个员工信息的 View 为 Employee。如果根据指定的 ID 找不到相应的员工，直接抛出一个状态为“404”的 HttpException 异常。

如下所示的是用于显示员工列表的 View (EmployeeList) 的定义，它的 Model 类型为 IEnumerable<Employee>。在该 View 中，通过一个表格来显示员工列表，值得一提的是，可以通过调用 HtmlHelper 的 ActionLink 方法将员工的名称显示为一个指向 Action 方法 GetEmployeeById 的链接。

---

```

@model IEnumerable<Employee>
<html>
  <head>
    <title>员工列表</title>
  </head>
  <body>
    <table>
      <tr>
        <th>姓名</th>
        <th>性别</th>
        <th>出生日期</th>
        <th>部门</th>
      </tr>
      @{
        foreach(Employee employee in Model)
        {
          <tr>
            <td>@Html.ActionLink(employee.Name, "GetEmployeeById",
              new { name = employee.Name, id = employee.Id })
            </td>
            <td>@Html.DisplayFor(m=>employee.Gender)</td>
            <td>@Html.DisplayFor(m=>employee.BirthDate)</td>
            <td>@Html.DisplayFor(m=>employee.Department)</td>
          </tr>
        }
      }
    </table>
  </body>
</html>

```

用于显示单个员工信息的名为 Employee 的 View 定义如下，这是一个 Model 类型为 Employee 的强类型的 View，通过表格的形式将员工的详细信息显示出来。

```
@model Employee
<html>
  <head>
    <title>@Model.Name</title>
  </head>
  <body>
    <table>
      <tr>
        <td>
          @Html.LabelFor(m=>m.Id)</td><td>@Html.DisplayFor(m=>m.Id)
        </td>
      </tr>
      <tr>
        <td>
          @Html.LabelFor(m=>m.Name)</td><td>@Html.DisplayFor(
            m=>m.Name)
          </td>
        </tr>
      <tr>
        <td>
          @Html.LabelFor(m=>m.Gender)</td><td>@Html.DisplayFor(
            m=>m.Gender)
          </td>
        </tr>
      <tr>
        <td>
          @Html.LabelFor(m=>m.BirthDate)</td><td>@Html.DisplayFor(
            m=>m.BirthDate)
          </td>
        </tr>
      <tr>
        <td>
          @Html.LabelFor(m=>m.Department)</td><td>@Html.DisplayFor(
            m=>m.Department)
          </td>
        </tr>
      </table>
    </body>
</html>
```

我们对两个页面的 URL 进行了相应的设计，主页用于显示所有员工列表，它指向 EmployeeController 的 Action 方法 GetAllEmployees。用于显示单个员工详细信息的页面的 URL 的结构为“/{员工姓名}/{员工 ID}”（比如“/李四/002”），它自然指向另一个 Action 方法 GetEmployeeById，为此我们在自动生成的 RouteConfig 类型中按照如下的方式注册两个路由。

```
public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        //其他操作
        routes.MapRoute(
            name: "Home",
            url: "",
```

```

        defaults: new { controller = "Employee", action = "GetAllEmployees" }
    );

    routes.MapRoute(
        name: "Detail",
        url: "{name}/{id}",
        defaults: new { controller = "Employee", action = "GetEmployeeById" }
    );
}
}

```

自定义的 `ControllerFactory` (`UnityControllerFactory`) 在 `Global.asax` 中通过如下的代码进行注册。用于创建 `UnityControllerFactory` 的 `UnityContainer` 对象注册了 `IEmployeeRepository` 和 `EmployeeRepository` 之间的映射关系。

---

```

public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        //其他操作
        UnityContainer unityContainer = new UnityContainer();
        unityContainer.RegisterType<IEmployeeRepository, EmployeeRepository>();
        UnityControllerFactory controllerFactory =
            new UnityControllerFactory(unityContainer);
        ControllerBuilder.Current.SetControllerFactory(controllerFactory);
    }
}

```

除此之外，我们还为该实例应用定义相应的布局文件和 CSS 样式，在这里就不一一介绍了。这个例子旨在演示通过自定义 `ControllerFactory` 实现以 IoC 的方式激活目标 `Controller` 对象，这样可以最大限度地降低 `Controller` 和其他组件之间的依赖关系，因为这些依赖会被用于激活 `Controller` 的 IoC 容器动态注入。

### 3.3.4 基于 IoC 的 ControllerActivator

除了通过自定义 `ControllerFactory` 的方式引入 IoC 之外，在使用默认 `DefaultControllerFactory` 情况下也可以通过一些扩展使基于 IoC 的 `Controller` 激活成为可能。不过这就需要我们具体了解现在 `DefaultControllerFactory` 内部的 `Controller` 激活机制了。

`DefaultControllerFactory` 针对目标 `Controller` 的激活其实是通过另一个名为 `ControllerActivator` 的组件来完成的，所有的 `ControllerActivator` 实现了 `System.Web.Mvc.IControllerActivator` 接口。如下面的代码片段所示，`IControllerActivator` 定义了唯一的用于创建 `Controller` 对象的 `Create` 方法，而 `DefaultControllerFactory` 使用的 `ControllerActivator` 可以直接通过构造函数参数的方式来指定。

---

```

public interface IControllerActivator
{
    IController Create(RequestContext requestContext, Type controllerType);
}

```

```

}

public class DefaultControllerFactory : IControllerFactory
{
    //其他成员
    public DefaultControllerFactory();
    public DefaultControllerFactory(IControllerActivator controllerActivator);
}

```

### 实例演示：创建基于 Ninject 的 ControllerActivator ( S311 )

如果我们基于一个 ControllerActivator 对象来创建一个 DefaultControllerFactory，它最终会被用于 Controller 对象的激活，那么可以通过自定义 ControllerActivator 的方式将 IoC 引入 Controller 激活系统。接下来自定义的 ControllerActivator 基于另一个 IoC 框架 Ninject，较之 Unity，Ninject 是一个更加轻量级也更适合 ASP.NET MVC 的 IoC 框架，将自定义的 ControllerActivator 起名为 NinjectControllerActivator。如下面的代码所示，针对目标 Controller 的创建是通过一个 StandardKernel 对象来完成的，为了方便实现类型的映射，我们定义了一个泛型的 Register 方法。

---

```

public class NinjectControllerActivator : IControllerActivator
{
    public IKernel Kernel { get; private set; }

    public NinjectControllerActivator()
    {
        this.Kernel = new StandardKernel();
    }
}

```

```

    }

    public IController Create(RequestContext requestContext, Type controllerType)
    {
        return (IController)this.Kernel.TryGet(controllerType);
    }

    public void Register<TFrom, TTo>() where TTo: TFrom
    {
        this.Kernel.Bind<TFrom>().To<TTo>();
    }
}

```

接下来我们使用的还是之前演示过的关于员工管理的例子，前面我们演示了属性注入的方式在激活 `EmployeeController` 的时候对 `Repository` 进行初始化，现在来演示另一种依赖注入形式——构造器注入。如下面的代码片段所示，只读的 `Repository` 是在构造函数中通过指定的参数初始化的，而该参数的类型是 `IEmployeeRepository`。

---

```

public class EmployeeController : Controller
{
    //其他成员
    public IEmployeeRepository Repository { get; private set; }

    public EmployeeController(IEmployeeRepository repository)
    {
        this.Repository = repository;
    }
}

```

为了让 ASP.NET MVC 的 Controller 激活系统采用我们自定义的 `ControllerActivator` 来创建目标 Controller，我们需要创建并注册一个相应的 `DefaultControllerFactory` 对象。如下面的代码片段所示，我们在 `Global.asax` 中创建一个 `NinjectControllerActivator` 对象，并注册了接口 `IEmployeeRepository` 和实现类型 `EmployeeRepository` 之间的匹配关系。最后据此创建一个 `DefaultControllerFactory` 对象，通过当前的 `ControllerBuilder` 进行注册。

---

```

public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        //其他成员
        NinjectControllerActivator controllerActivator =
            new NinjectControllerActivator();
        controllerActivator.Register<IEmployeeRepository, EmployeeRepository>();
        DefaultControllerFactory controllerFactory =
            new DefaultControllerFactory(controllerActivator);
        ControllerBuilder.Current.SetControllerFactory(controllerFactory);
    }
}

```

再次运行我们的程序，依然会得到如图 3-10 所示的结果，其实自定义 `ControllerActivator`

实现 IoC 的方式并不是很常用，接下来我们介绍第三种更加常用的 IoC 实现方式。

### 3.3.5 基于 IoC 的 DependencyResolver

如果在构建 `DefaultControllerFactory` 的时候没有显式指定采用 `ControllerActivator`，它默认使用的是一个类型为 `DefaultControllerActivator` 的对象。如下面的代码片段所示，这只是一个实现了 `IControllerActivator` 接口的私有类型，不能直接通过编程的方式使用它。

---

```
private class DefaultControllerActivator : IControllerActivator
{
    public DefaultControllerActivator();
    public DefaultControllerActivator(IDependencyResolver resolver);
    public IController Create(RequestContext requestContext,
        Type controllerType);
}
```

即使 `DefaultControllerFactory` 采用了默认的 `DefaultControllerActivator`，依然可以将 IoC 引入到 Controller 的激活系统中，而这就需要进一步了解实现在 `DefaultControllerActivator` 的 Controller 激活逻辑了。

其实 `DefaultControllerActivator` 完成对 Controller 的激活依赖于另一个名为 `DependencyResolver` 的对象。`DependencyResolver` 是一个非常重要的组件，可以将其视为 ASP.NET MVC 框架内部使用的 IoC 容器。它不只是用于针对 Controller 的激活，框架内部很多组件的提供最终都依赖于它。`DependencyResolver` 实现了具有如下定义的 `System.Web.Mvc.IDependencyResolver` 接口，`GetService` 和 `GetServices` 方法分别用于根据指定的类型获取单个和所有实例。

---

```
public interface IDependencyResolver
{
    object GetService(Type serviceType);
    IEnumerable<object> GetServices(Type serviceType);
}
```

整个 Web 默认使用的 `DependencyResolver` 可以通过 `System.Web.Mvc.DependencyResolver` 类型进行注册。如下面的代码片段所示，`DependencyResolver` 类型具有一个静态的 `Current` 属性表示当前 `DependencyResolver`，具体对 `DependencyResolver` 的注册通过调用静态方法 `SetResolver` 来完成。顺便说一下，`DependencyResolver` 类型并没有实现 `IDependencyResolver` 接口，并不是真正意义上的 `DependencyResolver`。

---

```
public class DependencyResolver
{
    //其他成员
    private static DependencyResolver _instance;

    public void InnerSetResolver(object commonServiceLocator);
    public void InnerSetResolver(IDependencyResolver resolver);
    public void InnerSetResolver(Func<Type, object> getService,
```



```

        Func<Type, IEnumerable<object>> getServices);

    public static void SetResolver(object commonServiceLocator);
    public static void SetResolver(IDependencyResolver resolver);
    public static void SetResolver(Func<Type, object> getService,
        Func<Type, IEnumerable<object>> getServices);

    public static IDependencyResolver Current { get; }
    public IDependencyResolver InnerCurrent { get; }
}

```

这个被封装的 `DependencyResolver`（指实现了接口 `IDependencyResolver` 的某个类型的对象，不是指 `DependencyResolver` 类型的对象，对于后者我们会采用“`DependencyResolver` 类型对象”的说法）通过只读属性 `InnerCurrent` 表示，而三个 `InnerSetResolver` 方法重载用于初始化该属性。静态字段 `_instance` 表示当前的 `DependencyResolver` 类型对象，静态只读属性 `Current` 则表示该对象内部封装的 `DependencyResolver` 对象，而它通过三个静态的 `SetResolver` 进行初始化。

如果没有对 `DependencyResolver` 进行显式注册，系统默认使用的是一个类型为 `DefaultDependencyResolver` 的对象。如下面的代码片段所示，这是一个私有类型，用于根据类型提供“服务实例”的 `GetService` 方法直接以反射的方式根据类型创建并返回对应的实例。对于类型为接口/抽象类，或者不曾定义默认公有构造函数的类型，我们直接返回 `Null`。也就是说在默认的情况下，`Controller` 的激活最终是通过对 `Controller` 类型的反射来实现的。`DefaultDependencyResolver` 的另一个 `GetServices` 方法直接返回一个空的对象列表。

---

```

private class DefaultDependencyResolver : IDependencyResolver
{
    public object GetService(Type serviceType)
    {
        if (serviceType.IsInterface || serviceType.IsAbstract)
        {
            return null;
        }
        try
        {
            return Activator.CreateInstance(serviceType);
        }
        catch
        {
            return null;
        }
    }

    public IEnumerable<object> GetServices(Type serviceType)
    {
        return Enumerable.Empty<object>();
    }
}

```

上面介绍的类型 `DefaultControllerFactory`、`IControllerActivator`、`DefaultControllerActivator`、`IDependencyResolver`、`DefaultDependencyResolver` 和 `DependencyResolver` 之前的关系基本上

可以通过如图 3-11 所示的类图来体现。

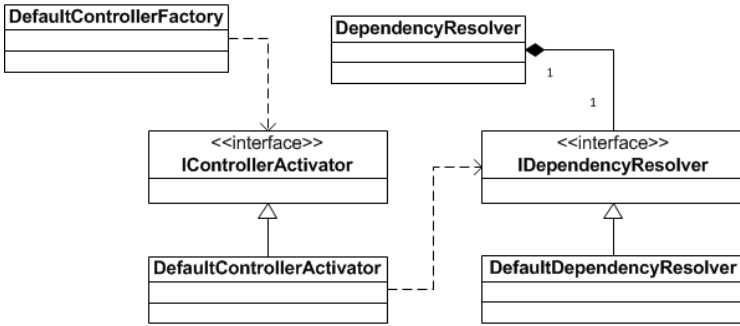


图 3-11 DefaultControllerFactory + ControllerActivator + DependencyResolver

### 实例演示：创建基于 Ninject 的 DependencyResolver ( S312 )

通过前面的介绍我们知道，当调用构造函数创建一个 DefaultControllerFactory 的时候，如果调用的时候默认无参构造函数，后者将作为参数的 ControllerActivator 对象设置为 Null，那么默认请求用于激活 Controller 实例的是通过 DependencyResolver 类型的静态属性 Current 表示的 DependencyResolver 对象，换言之，我们可以通过自定义 DependencyResolver 的方式来实现基于 IoC 的 Controller 激活。

同样是采用 Ninject，我们定义了一个具有如下定义的 NinjectDependencyResolver。与上面定义的 NinjectControllerActivator 类似，NinjectDependencyResolver 具有一个 IKernel 类型的只读属性 Kernel，该属性在构造函数中被初始化为一个 StandardKernel 对象。对于实现的 GetService 和 GetServices 方法，直接调用 Kernel 的 TryGet 和 GetAll 返回指定类型的实例和实例列表。为了方便进行类型映射，我们定义了泛型的 Register<TFrom, TTo> 方法。

---

```
public class NinjectDependencyResolver : IDependencyResolver
{
    public IKernel Kernel { get; private set; }

    public NinjectDependencyResolver()
    {
        this.Kernel = new StandardKernel();
    }

    public void Register<TFrom, TTo>() where TTo: TFrom
    {
        this.Kernel.Bind<TFrom>().To<TTo>();
    }

    public object GetService(Type serviceType)
    {
        return this.Kernel.TryGet(serviceType);
    }

    public IEnumerable<object> GetServices(Type serviceType)
    {
        return this.Kernel.GetAll(serviceType);
    }
}
```

我们只需要创建一个自定义的 NinjectDependencyResolver 对象并将其作为当前的 DependencyResolver 即可。如下面的代码片段所示，我们创建了一个 NinjectDependencyResolver 对象并注册了 IEmployeeRepository 和 EmployeeRepository 之间的映射关系，然后调用 DependencyResolver 的静态方法 SetResolver 将创建的 NinjectDependencyResolver 注册为当前的 DependencyResolver 对象。再次运行我们的程序，依然会得到如图 3-10 所示的效果。

```
public class MvcApplication : System.Web.HttpApplication
{
    //其他成员
    protected void Application_Start()
    {
        //其他操作
        NinjectDependencyResolver dependencyResolver =
            new NinjectDependencyResolver();
        dependencyResolver.Register<IRepository, IRepository>();
        DependencyResolver.SetResolver(dependencyResolver);
    }
}
```

## 本章小结

当目标 Controller 的名称通过 URL 路由被解析出来之后，ASP.NET MVC 利用注册的 ControllerFactory 根据该名称实现对目标 Controller 的激活。除了完成对 Controller 的激活之外，ControllerFactory 还负责对 Controller 的释放工作，以及获取用于控制会话状态行为的 SessionStateBehavior 枚举。ControllerFactory 的注册通过 ControllerBuilder 来完成。

ASP.NET MVC 默认使用的 ControllerFactory 类型为 DefaultControllerFactory，它在 Controller 类型进行解析的时候对所有 Controller 类型采用了基于文件的缓存以提升性能。在 DefaultControllerFactory 内部，它将解析得到的 Controller 类型递交给 ControllerActivator 对象对 Controller 实施最终的激活。默认使用 DefaultControllerActivator 内部利用了当前注册的 DependencyResolver 来提供具体的 Controller 对象。如果没有对 DependencyResolver 进行显式注册，默认提供的 DependencyResolver 将采用对提供类型的反射方式创建相应的实例

将 IoC 应用到 Controller 的激活过程中具有重要的意义，可以极大地降低 Controller 和其他组件的依赖关系。通过对 Controller 激活流程的分析，我们提供了三种实现方法，即自定义 ControllerFactory、ControllerActivator 和 DependencyResolver。