第 1 章　ASP.NET + MVC

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

1.1 传统 MVC 模式

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

1.1.1 自治视图

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

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

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

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

1.1.2 什么是 MVC 模式

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

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

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

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

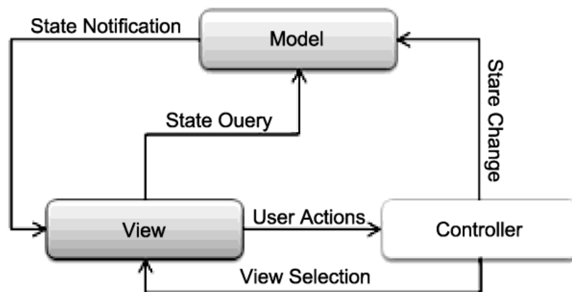

图 1-1 Model-View-Controller 之间的交互

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

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

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

1.2 MVC 的变体

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

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

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

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

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

1.2.1 MVP

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

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

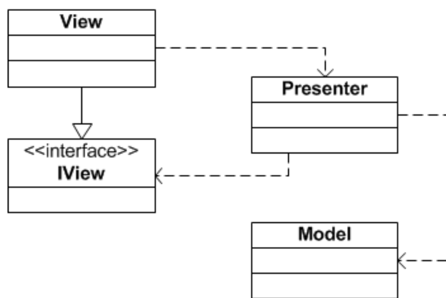

图 1-2　Model-View-Presenter 之间的交互

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

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

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

1．PV 与 SC

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

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

图 1-3 员工查询页面

假设现在通过 ASP.NET Web Forms 应用来设计这个页面，我们来讨论一下如果采用 PV 模式 View 的接口该如何定义。对于 Presenter 来说，View 供它操作的控件有两个，一个是包含所

有部门列表的 DropDownList，另一个则是显示员工列表的 GridView。在页面加载的时候，Presenter 将部门列表绑定在 DropDownList 上，与此同时包含所有员工的列表被绑定到 Grid View 上。当用户选择某个部门并单击“查询”按钮后，View 将包含筛选部门在内的查询请求转发给 Presenter，后者筛选出相应的员工列表之后将其绑定到 GridView。

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

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

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

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

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

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

```
    }

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

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

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

2．View 和 Presenter 交互的规则（针对 SC 模式）

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

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

对于 Presenter 处理用户交互请求的流程，如果中间环节需要涉及 Model，它会直接发起对 Model 的调用。如果需要 View 的参与（比如需要将 Model 最新的状态反映在 View 上），Presenter 会驱动 View 完成相应的工作。

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

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

3．实例演示：SC 模式的应用（S101）

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

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

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

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

```
        this.Gender       = gender;
        this.BirthDate    = birthDate;
        this.Department   = department;
    }
}
```

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

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

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

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

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

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

```
}

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

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

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

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

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

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

我们在构造函数中注册了View的DepartmentSelected事件，作为事件处理器的OnDepartmentSelected方法通过调用Repository（即Model）得到了用户选择部门下的员工列表，返回的员工列表通过调用View的BindEmployees方法绑定在View上。在Initialize方法中，通过调用Repository获取所有员工的列表，并通过调用View的BindEmployees方法将员工列

表显示在界面上，作为筛选条件的部门列表则通过调用 View 的 BindDepartments 方法绑定在 View 上。

最后我们来看看作为 View 的 Web 页面如何定义。如下所示的是组成该页面的 HTML，其核心部分是一个用于绑定筛选部门列表的 DropDownList 和一个绑定员工列表的 GridView[1]，所以无须对它多做介绍。

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

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

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

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

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

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

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

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

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

1.2.2 Model 2

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

为了让MVC 能够为 Web 应用提供原生的支持，另一个被称为 Model 2 的 MVC 变体被提出来，这是一种来源于 Java 阵营的 Web 应用架构模式。Java Web 应用具有两种基本的基于 MVC 的架构模式，分别被称为 Model 1 和 Model 2。Model 1 类似于我们前面提及的自治视图模式，它将数据的可视化呈现和用户交互操作的处理逻辑合并在一起。Model 1 适用于那些比较简单的 Web 应用，对于相对复杂的应用多采用 Model2。

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

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

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

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

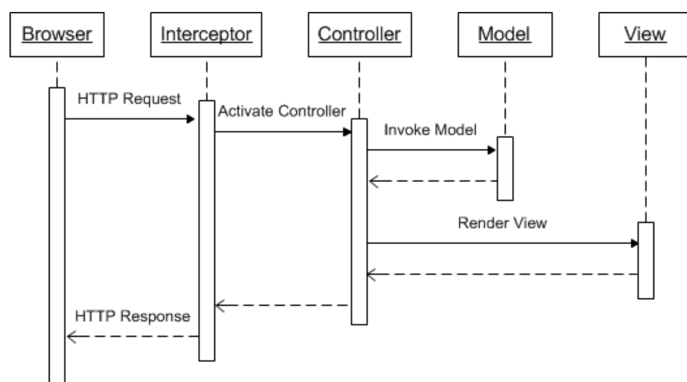

图 1-4　Model 2 交互流程

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

1.2.3 ASP.NET MVC 与 Model 2

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

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

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

1.3 IIS/ASP.NET 管道

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

1.3.1 IIS 5.x 与 ASP.NET

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

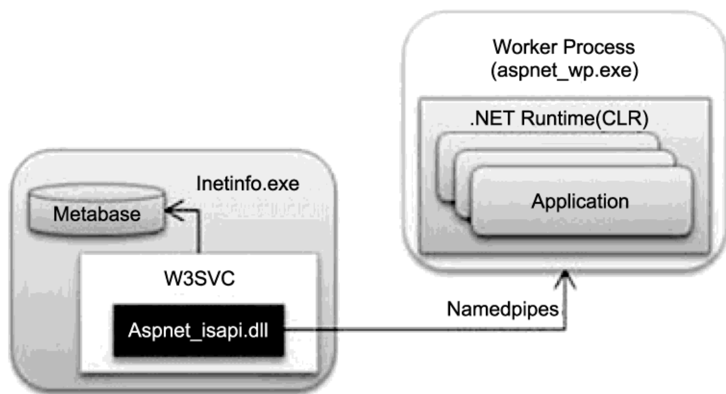

图 1-5 IIS 5.x 与 ASP.NET

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

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

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

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

1.3.2 IIS 6.0 与 ASP.NET

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

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

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

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

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

- 持续监听。由于 HTTP.SYS 是一个网络驱动程序，始终处于运行状态，所以对于用户的 HTTP 请求能够及时作出反应。
- 更好的稳定性。HTTP.SYS 运行在操作系统内核模式下，并不执行任何用户代码，所以其本身不会受到 Web 应用、工作进程和 IIS 进程的影响。
- 内核模式下数据缓存。如果某个资源被频繁请求，HTTP.SYS 会把响应的内容进行缓存，缓存的内容可以直接响应后续的请求。由于这是基于内核模式的缓存，不存在内核模式和用户模式的切换，响应速度将得到极大的改进。

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

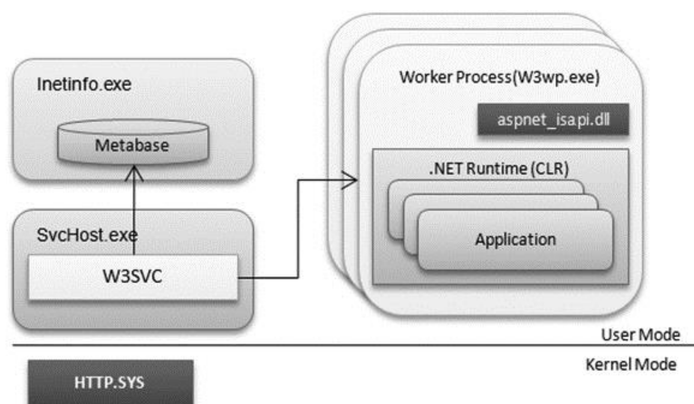

图 1-6　IIS 6.0 与 ASP.NET

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

1.3.3 IIS 7.0 与 ASP.NET

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

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

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

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

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

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

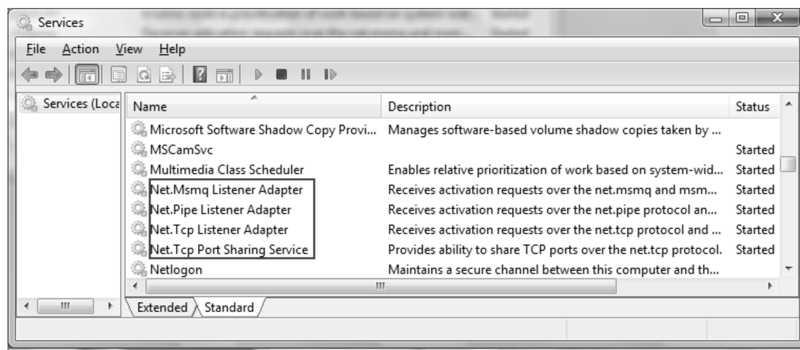

图 1-7 定义在 SMSVCHost.exe 中的 Windows Service

图 1-8 揭示了 IIS 7.0 的整体架构及整个请求处理流程。无论是从 W3SVC 接收到的 HTTP 请求，还是通过 WCF 提供的监听适配器接收到的针对其他传输协议的请求，最终都会被传递到 WAS。如果相应的工作进程（针对单个应用程序池）尚未创建，则 WAS 会创建工作进程。WAS 在进行请求处理过程中通过内置的配置管理模块加载相关的配置信息，并对相关的

组件进行配置。与 IIS 5.x 和 IIS 6.0 基于 Metabase 的配置信息存储不同的是，IIS 7.0 大都将配置信息存放于 XML 形式的配置文件中，基本的配置存放在 applicationHost.config 中。

图 1-8　IIS 7.0 与 ASP.NET

1.3.4 ASP.NET 集成

从上面对 IIS 5.x 和 IIS 6.0 的介绍中我们不难发现，IIS 与 ASP.NET 是两个相互独立的管道（Pipeline）。在各自管辖范围内，它们各自具有自己的一套机制对 HTTP 请求进行处理。两个管道通过 ISAPI 实现“连通”，IIS 是第一道屏障，当对 HTTP 请求进行必要的前期处理（比如身份验证等）后，IIS 通过 ISAPI 将请求分发给 ASP.NET 管道。当 ASP.NET 在自身管道范围内完成对 HTTP 请求的处理时，处理后的结果再返回到 IIS，IIS 对其进行后期处理（比如日志记录、压缩等）后生成 HTTP 回复对请求予以响应。图 1-9 反映了 IIS 6.0 与 ASP.NET 之间的桥接关系。

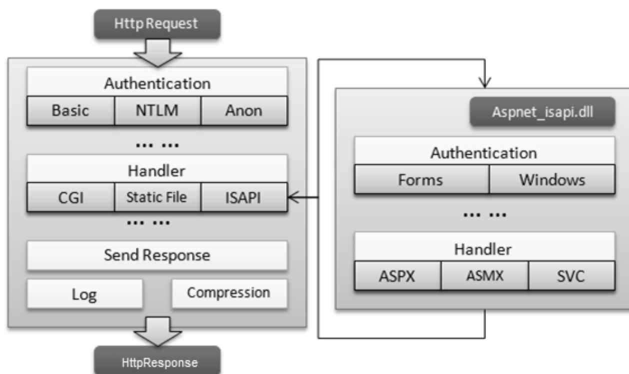

图 1-9 基于 IIS 6.0 与 ASP.NET 双管道设计

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

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

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

- 允许通过本地代码（Native Code）和托管代码（Managed Code）两种方式定义 IIS Module，这些 IIS Module 注册到 IIS 中形成一个通用的请求处理管道。由这些 IIS Module 组成的这个管道能够处理所有的请求，不论请求基于怎样的资源类型。例如，可以将 FormsAuthenticationModule 提供的 Forms 认证应用到基于.aspx、CGI 和静态文件的请求。

- 将 ASP.NET 提供的一些强大的功能应用到原来难以企及的地方，比如将 ASP.NET 的 URL 重写功能置于身份验证之前。
- 采用相同的方式去实现、配置、检测和支持一些服务器特性(Feature)，比如 Module、Handler 映射、定制错误配置（Custom Error Configuration）等。

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

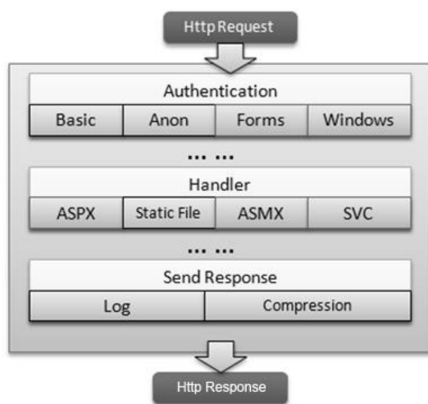

图 1-10　基于 IIS 7.0 与 ASP.NET 集成管道设计

1.3.5 ASP.NET 管道

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

如果 HTTP.SYS 接收到的 HTTP 请求是对该 Web 应用的第一次访问，在成功加载运行时后，IIS 会通过 AppDomainFactory 为该 Web 应用创建一个应用程序域。随后一个特殊的运行时 IsapiRuntime 被加载。IsapiRuntime 定义在程序集 System.Web.dll 中，对应的命名空间为“System.Web.Hosting”，被加载的 IsapiRuntime 会接管该 HTTP 请求。

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

请求的上下文（Context）对象 HttpContext。

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

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

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

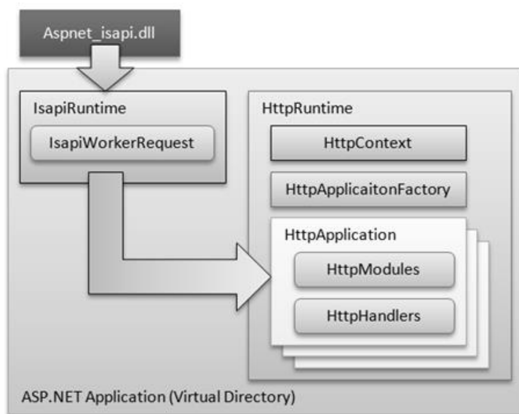

图 1-11 ASP.NET 处理管道

1．HttpApplication

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

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

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

表 1-1　HttpApplication 事件列表

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

续表

名　　称	描　　述
ReleaseRequestState， PostReleaseRequestState	ASP.NET 先后触发这两个事件，使状态管理模块释放基于当前请求相应的状态
UpdateRequestCache，	ASP.NET 先后触发这两个事件，以使缓存模块将 HttpHandler 处理

PostUpdateRequestCache	请求得到的内容得以保存到输出缓存中
LogRequest，PostLogRequest	ASP.NET 先后触发这两个事件为当前请求进行日志记录
EndRequest	整个请求处理完成后，EndRequest 事件被触发

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

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

2．HttpModule

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

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

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

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

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

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

3．HttpHandler

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

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

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

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

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

```
}
```

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

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

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

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

换句话说，在调用当前 HttpContext 的 RemapHandler 方法时指定一个具体的 HttpHandler 对象，是为了让 ASP.NET 直接跳过默认的 HttpHandler 映射操作。此外，由于这个默认的 HttpHandler 映射发生在 HttpApplication 的 PostMapRequestHandler 事件触发之前，所以只有在这之前调用

RemapHandler 方法才有意义。通过阅读下一节的内容，我们就可以知道实现 ASP.NET MVC 框架的 MvcHandler（一个自定义的 HttpHandler）就是通过调用这个方法进行映射的。

1.4 ASP.NET MVC 是如何运行的

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

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

1.4.1 建立在“迷你版”ASP.NET MVC 上的 Web 应用

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

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

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

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

与真正的 ASP .NET MVC 应用开发一样，我们需要定义 Controller 类。按照约定的命名方

式（以字符“Controller”作为后缀），我们定义了如下一个继承自抽象类 ControllerBase 的 HomeController。定义在 HomeController 中的 Action 方法 Index 具有一个 SimpleModel 类型的输入参数，并以 ActionResult 作为返回类型。

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

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

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

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

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

路由实现了请求 URL 与目标 Controller/Action 之间的映射。ASP.NET MVC 的路由建立在 ASP.NET 自身的路由系统之上，后者则通过一个自定义的 HttpModule 来实现。在这个“迷你

版”ASP.NET MVC 框架中，我们将其命名为 UrlRoutingModule，它与 ASP.NET 路由系统中对应的 HttpModule 类型同名。在运行 Web 应用之前，我们需要通过配置对该自定义 HttpModule 进行注册，下面是相关的配置。

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

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

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

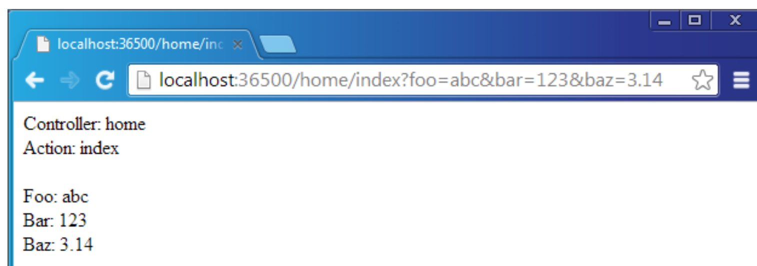

图 1-12　采用符合注册的路由地址模板的地址访问 Web 应用

上面演示了如何在自己创建的“迷你版”ASP.NET MVC 框架上搭建一个 Web 应用，从中可以看到它和创建一个真正的 ASP.NET MVC 应用别无二致。接下来我们就来逐步地分析这个

自定义的ASP.NET MVC框架是如何建立起来的，它也代表了真正的ASP.NET MVC框架的基本工作原理。

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

1.4.2 路由

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

1．RouteData

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

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

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

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

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

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

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

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

IRouteHandler 接口的 GetHttpHandler 方法具有一个类型为 RequestContext 的参数。顾名思义，RequestContext 表示当前(HTTP)请求的上下文，其核心就是对当前 HttpContext 和 RouteData 的封装，这可以通过如下的代码片段看出来。

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

2．Route 和 RouteTable

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

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

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

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

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

   public Route()
   {
      this.DataTokens          = new Dictionary<string, object>();
```

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

```
        this.RouteHandler       = new MvcRouteHandler();
    }

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

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

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

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

一个 Web 应用可以采用多种不同的 URL 模式，所以需要注册多个继承自 RouteBase 的

Route 对象，多个 Route 对象组成了一个路由表。在我们自定义的迷你版 ASP.NET MVC 框架中，路由表通过类型 RouteTable 表示。如下面的代码片段所示，RouteTable 仅仅具有一个类型为 RouteDictionary 的 Routes 属性表示针对整个 Web 应用的全局路由表。

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

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

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

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

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

3．UrlRoutingModule

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

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

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

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

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

如果方法调用返回一个具体的 RouteData 对象，UrlRoutingModule 会根据该对象本身和之前得到的 HttpContextWrapper 对象创建一个表示当前请求上下文的 RequestContext 对象，并将其作为参数传入 RouteData 的 RouteHandler 的 GetHttpHandler 方法得到一个 HttpHandler 对象。UrlRoutingModule 最后调用 HttpContextWrapper 对象的 RemapHandler 方法对得到的 HttpHandler

对象进行映射，那么针对当前 HTTP 请求的后续处理将由这个 HttpHandler 来接手。

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

1.4.3 Controller 的激活

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

1．MvcRouteHandler

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

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

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

```
public class MvcRouteHandler: IRouteHandler
{
```

```
    public IHttpHandler GetHttpHandler(RequestContext requestContext)
    {
        return new MvcHandler(requestContext);
    }
}
```

2．MvcHandler

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

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

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

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

    public RequestContext RequestContext { get; private set; }

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

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

```
        controller.Execute(this.RequestContext);
    }
}
```

Controller 与 ControllerFactory

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

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

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

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

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

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

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

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

```
    }

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

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

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

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

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

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

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

```
        }
    }

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

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

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

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

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

1.4.4 Action 的执行

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

1．ActionInvoker

我们同样为 ActionInvoker 定义了一个接口 IActionInvoker。如下面的代码片段所示，该接口定义了唯一的方法 InvokeAction 用于执行指定名称的 Action 方法，该方法的第一个参数是一个表示针对当前 Controller 上下文的 ControllerContext 对象。

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

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

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

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

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

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

2．ModelBinder

我们为 ModelBinder 实现的接口 IModelBinder 提供了一个简单的定义，这与在真正的 ASP.NET MVC 中的同名接口的定义不尽相同。如下面的代码片段所示，该接口具有唯一的 BindModel 方法，它根据 ControllerContext、Model 名称（在这里实际上是参数名称）和类型得到一个作为参数的对象。

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

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

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

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

        object modelInstance = Activator.CreateInstance(modelType);
```

```
    foreach (PropertyInfo property in modelType.GetProperties())
    {
        if (!property.CanWrite || (!property.PropertyType.IsValueType &&
            property.PropertyType != typeof(string)))
        {
            continue;
        }
        object propertyValue;
        if (GetValueTypeInstance(controllerContext, property.Name,
            property.PropertyType, out propertyValue))
        {
            property.SetValue(modelInstance, propertyValue, null);
        }
    }
    return modelInstance;
}

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

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

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

    //数据来源三: ControllerContext.RequestContext.RouteData.Values
    foreach (var item in
        controllerContext.RequestContext.RouteData.Values)
    {
```

```
            if (dataSource.ContainsKey(item.Key.ToLower()))
            {
                continue;
            }
            dataSource.Add(item.Key.ToLower(),
              controllerContext.RequestContext.RouteData.Values[item.Key]);
        }

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

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

3．ControllerActionInvoker

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

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

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

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

4．ActionExecutor

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

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

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

```
public MethodInfo MethodInfo { get; private set; }

public ActionExecutor(MethodInfo methodInfo)
{
    this.MethodInfo = methodInfo;
}

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

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

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

    UnaryExpression instanceCast = Expression.Convert(target,
        methodInfo.ReflectedType);
    MethodCallExpression methodCall =
        Expression.Call(instanceCast, methodInfo, parameters);
```

```
        UnaryExpression convertToObjectType = Expression.Convert(
            methodCall, typeof(object));
        return Expression.Lambda<Func<object, object[], object>>(
            convertToObjectType, target, arguments).Compile();
    }
}
```

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

5．ActionResult

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

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

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

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

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

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

```
}
```

1.4.5 完整的流程

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

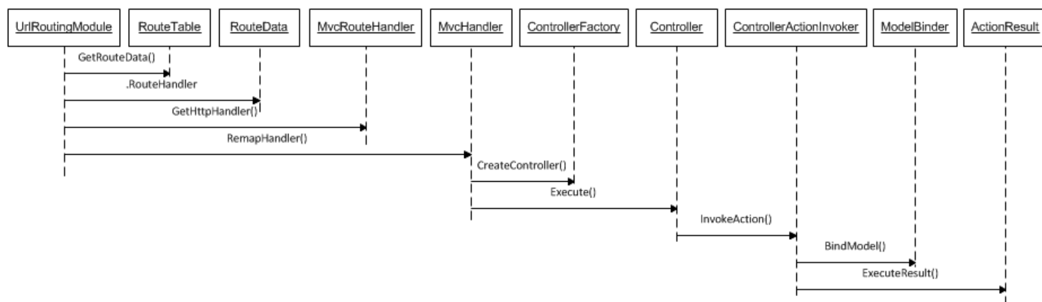

图 1-13 “接收请求”到“响应回复”的完整流程

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

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

接下来 UrlRoutingModule 通过 RouteData 对象的 RouteHandler 属性得到匹配 Route 对象采用的 RouteHandler 对象，在默认情况下这是一个 MvcRouteHandler 对象。UrlRoutingModule 随后会调用这个 MvcRouteHandler 对象的 GetHttpHandler 方法得到一个 HttpHandler 对象。对于 MvcRouteHandler 来说，它的 GetHttpHandler 方法具体返回的是一个 MvcHandler 对象。UrlRoutingModule 随之调用当前 HTTP 上下文的 MapHttpHandler 方法对得到的 HttpHandler 对

象实施映射，那么此 HttpHandler 将最终接管当前请求的处理。

对于 MvcHandler 来说，当它被用来处理当前请求的时候，它会利用 RouteData 对象得到目标 Controller 的名称，并借助于注册的 ControllerFactory 来激活对应的 Controller 对象。目标 Controller 被激活之后，它的 Execute 方法被 MvcHandler 调用。

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

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