

第 2 章 路由

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

2.1 ASP.NET 路由

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

2.1.1 请求 URL 与物理文件的分离

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

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

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

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

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

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



图 2-1 员工列表和员工详细信息页面

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

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

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

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

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

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

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

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

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

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

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

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

```

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

```

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

```

public partial class Default : Page
{
    private EmployeeRepository repository;

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

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

```

2.1.3 Route 与 RouteTable

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

1. RouteBase

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

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

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

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

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

2 . RouteData

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

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

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

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

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

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

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

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

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

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

3 . VirtualPathData

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

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

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

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

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

```

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

```

4. Route

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

```

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

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

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

```

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

```

/weather/{areacode}/{days}

```

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

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

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

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

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

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

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

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

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

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

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

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

5. RouteTable

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

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

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

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

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

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

```

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

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

```

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

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

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

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

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

```

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

```

6. 线程安全

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

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

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

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

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

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

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

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

2.1.4 路由注册

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

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

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

```

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

```

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

```

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

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

```


1. 变量默认值

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

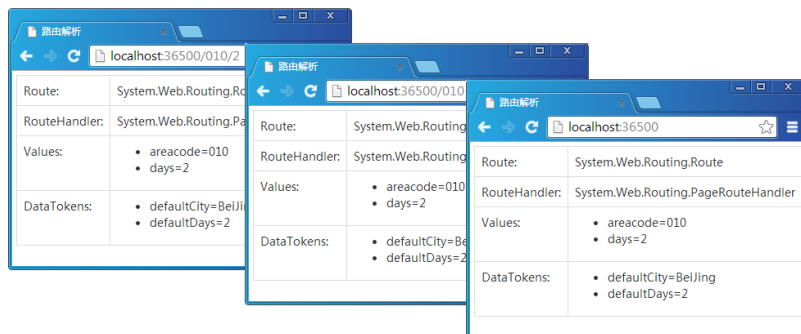


图 2-2 基于变量默认值的 URL 等效性

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

2. 约束

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

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

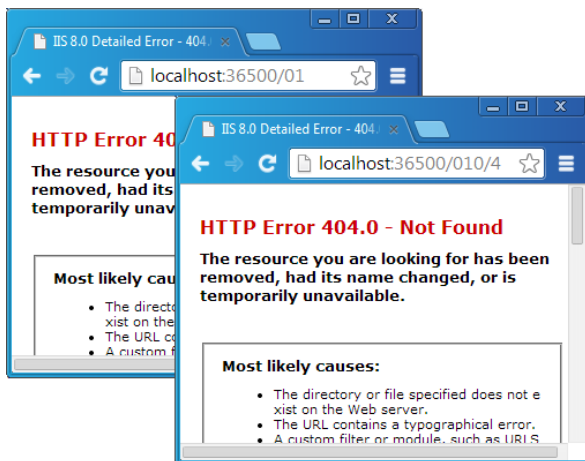


图 2-3 不满足正则表达式约束导致的 404 错误

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

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

public enum RouteDirection
{
    IncomingRequest,
    UrlGeneration
}
```

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

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

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

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

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

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

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

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

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

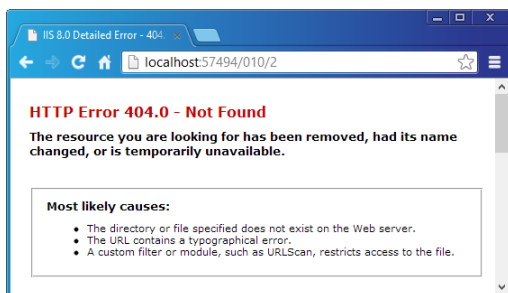


图 2-4 不满足 HTTP 方法约束 (POST) 导致的 404 错误

3. 对现有物理文件的路由

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

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

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

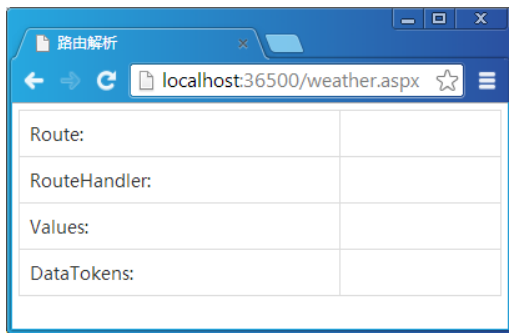


图 2-5 直接请求现存的物理文件（RouteExistingFiles = false）

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

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

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

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

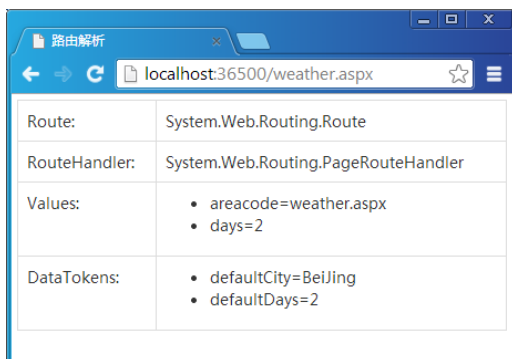


图 2-6 直接请求现存的物理文件（`RouteExistingFiles = true`）

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

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

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

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

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

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

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

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

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

```

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

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

```

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

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

```

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

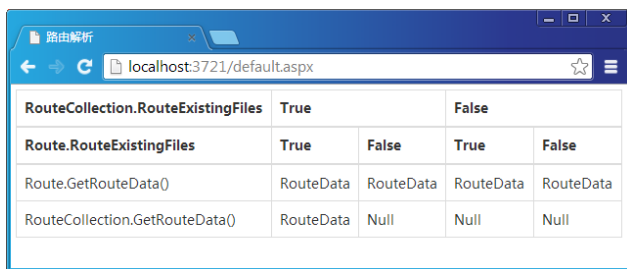
```

```

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

```

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



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

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

4. 注册路由忽略地址

`RouteTable` 的静态属性 `Routes` 返回的 `RouteCollection` 对象代表针对整个应用的全局路由

表。如果我们将该对象的 `RouteExistingFiles` 属性设置为 `True`，ASP.NET 路由系统将会对所有抵达的请求实施路由，但这同样会带来一些问题。

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

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

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

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

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

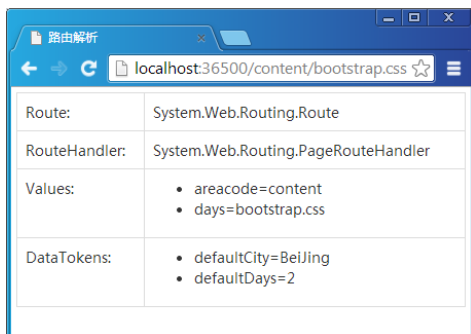


图 2-8 直接请求现存的.css 文件 (RouteExistingFiles = true)

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

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

5. 直接添加路由对象

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

```
public class Global : System.Web.HttpApplication
{
    protected void Application_Start(object sender, EventArgs e)
    {
        var defaults = new RouteValueDictionary {
```

```

    { "areacode", "010" }, { "days", 2 } };
    var constraints = new RouteValueDictionary {
        { "areacode", @"0\d{2,3}" }, { "days", @"[1-3]{1}" } };
    var dataTokens = new RouteValueDictionary {
        { "defaultCity", "BeiJing" }, { "defaultDays", 2 } };

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

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

```

2.1.5 根据路由规则生成 URL

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

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

```

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

```

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

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

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

同样以之前定义的关于获取天气信息的路由模板为例，我们在 `Weather.aspx` 页面的后台代码中按照如下方法通过 `RouteTable` 的静态 `Routes` 得到代表全局路由表的 `RouteCollection` 对象，并调用其 `GetVirtualPath` 方法生成 3 个具体的 URL。

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

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

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

从上面的代码片段可以看到，第一次调用 `GetVirtualPath` 方法传入的 `requestContext` 和 `values` 参数均为 `Null`；第二次则指定了一个手工创建的 `RequestContext` 对象，其 `RouteData` 的 `Values`

属性具有两个变量（areaCode=0512； days=1），而 values 参数依然为 Null；第三次则同时为参数 requestContext 和 values 指定了具体的对象，后者包含两个参数（areaCode=028； days=3）。如果我们利用浏览器访问 Weather.aspx 页面会得到如图 2-9 所示的 3 个 URL，这充分证实了上面提到的关于变量选择优先级的结论。（S209）

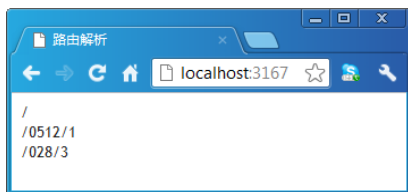


图 2-9 GetVirtualPath 方法接受不同参数生成的 URL

2.2 ASP.NET MVC 路由

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

2.2.1 路由映射

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

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

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

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

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

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

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

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

2.2.2 路由注册 (S210)

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

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

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

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

```

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

```

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

```

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

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

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

```

具体的路由注册依然定义在添加的 `Global.asax` 文件中。如下面的代码片段所示，我们利用

RouteTable 的静态属性 Routes 获取代表全局路由表的 RouteCollection 对象，然后调用其 MapRoute 方法注册了一个采用 “{areacode}/{days}” 作为路由模板的 Route 对象，并指定了变量的默认值、约束和命名空间列表。由于成功匹配的路由对象必须具有一个名为 “controller” 的路由变量，所以我们直接将 controller 的默认值设置为 “home”。

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

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

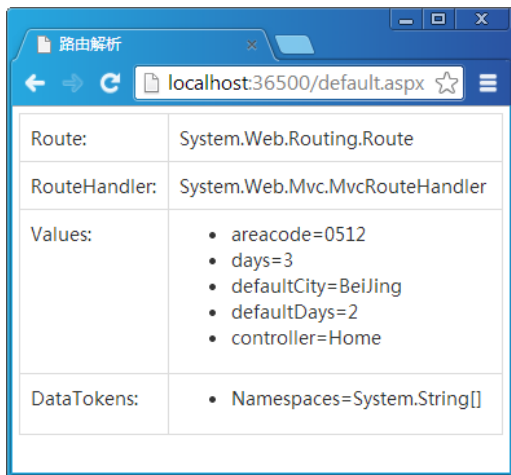


图 2-10 采用 ASP.NET MVC 路由映射得到的 RouteData

2.2.3 缺省 URL 参数

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

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

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

URL 参数引用着同一个 `UrlParameter` 对象。

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

    private UrlParameter() {}

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

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

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

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

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

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

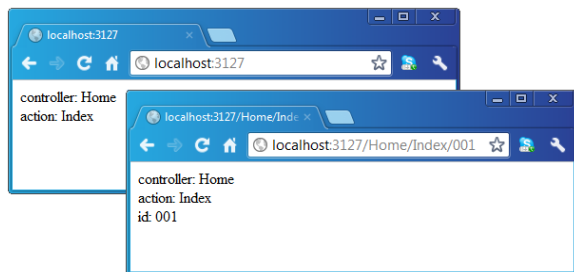


图 2-11 普通路由变量与缺省 URL 参数的路由变量之间的差别

2.2.4 基于 Area 的路由映射

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

1. AreaRegistration 与 AreaRegistrationContext

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

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

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

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

Area 的路由注册。

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

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

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

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

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

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

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

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

2. AreaRegistration 的缓存

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

ASP.NET MVC 对 `AreaRegistration` 类型列表的缓存是基于文件的。具体来说，当 ASP.NET MVC 框架通过程序集加载和类型反射得到了所有的 `AreaRegistration` 类型列表后，会对其序列化并将序列化的结果保存为一个物理文件中。这个名为“MVC-AreaRegistrationTypeCache.xml”的 XML 文件被存放在 ASP.NET 的临时目录下，具体的路径如下。其中第一个针对寄宿于 Local IIS 中的 Web 应用，后者针对直接通过 Visual Studio Developer Server 或者 IIS Express 作为宿主的应用。

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

下面的 XML 片段体现了这个作为所有 `AreaRegistration` 类型缓存的 XML 文件的结构。我们从其中可以看到所有的 `AreaRegistration` 类型名称，连同它所在的托管模块和程序集名称都被保

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

存了下来。当 `AreaRegistration` 的静态方法 `RegisterAllAreas` 被调用之后，系统会试图加载该文件，如果该文件存在并且具有期望的结构，那么系统将不再通过程序集加载和反射来解析所有 `AreaRegistration` 的类型，而是直接对文件内容进行反序列化得到所有 `AreaRegistration` 类型的列表。

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

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

3. 实例演示：查看针对 Area 的路由信息 (S212)

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

```
public class WeatherAreaRegistration : AreaRegistration
{
    public override string AreaName
    {
```

```

        get { return "Weather"; }
    }

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

```

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

```

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

```

在进行路由解析并生成 `RouteData` 对象的 `GetRouteData` 方法中，我们对创建的 `HttpRequest` 对象略加修改。如下面的代码片段所示，我们将请求 URL 设置为 `"/weather/0512/3"`，正好与 `WeatherAreaRegistration` 注册的 `Route` 对象采用的路由模板（`weather/{areacode}/{days}`）相匹配。

```

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

```



```

        HttpContextBase contextWrapper = new HttpContextWrapper(context);
        return routeData = RouteTable.Routes.GetRouteData(contextWrapper);
    }
}

```

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

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

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

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

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

2.2.5 链接和 URL 的生成

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

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

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

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

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

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

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

```
public class HtmlHelper
{
    //其他成员
    public HtmlHelper(ViewContext viewContext,
```

```

        IViewDataContainer viewDataContainer);
    public HtmlHelper(ViewContext viewContext,
        IViewDataContainer viewDataContainer, RouteCollection routeCollection);

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

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

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

```

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

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

`UrlHelper` 和 `HtmlHelper` 分别通过 `Action` 和 `ActionLink` 方法生成一个指向定义在某个 `Controller` 类型中的 `Action` 方法的 URL 和链接。下面的代码片段列出了 `UrlHelper` 的所有 `Action` 方法重载，参数 `actionName` 和 `controllerName` 分别代表 `Action` 和 `Controller` 的名称。`object` 或者 `RouteValueDictionary` 类型表示的 `routeValues` 参数表示替换路由模板中变量的参数列表。参数 `protocol` 和 `hostName` 代表作为完整 URL 的传输协议（比如 `http` 和 `https` 等）和主机名。

```

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

```

```

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

```

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

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

```

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

```

```

        IDictionary<string, object> htmlAttributes);
    }

```

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

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

```

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

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

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

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

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

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

```

RouteHelper 定义了一个 Action 方法根据指定的 Action 名称、Controller 名称、路由参数

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

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

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

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

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

```
public partial class Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
```

```

{
    HttpRequest request = new HttpRequest("default.aspx",
        "http://localhost:3721/products/getproduct/001", null);
    HttpResponse response = new HttpResponse(new StringWriter());
    HttpContext context = new HttpContext(request, response);
    HttpContextBase contextWrapper = new HttpContextWrapper(context);

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

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

```

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

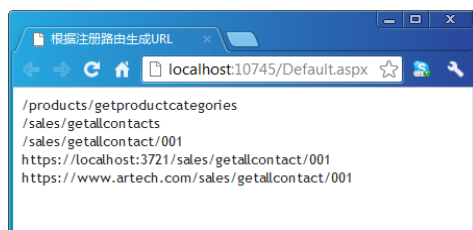


图 2-13 通过自定义 `RouteHelper` 生成的 URL

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

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

如下面的代码片段所示，`UrlHelper` 定义了一系列的 `RouteUrl` 方法，除了第一个重载之外，

后面的重载都接受一个表示 Route 注册名称的参数 `routeName`。与调用 `UrlHelper` 的 `Action` 方法一样，我们可以指定用于替换定义在 URL 模板中路由变量的参数（`routeValues`），以及传输协议名称（`protocol`）和主机名称（`hostName`）。

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

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

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

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



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

2.3 动态 HttpHandler 映射

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

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

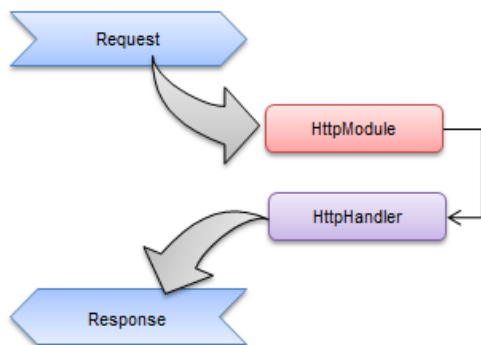


图 2-14 通过 HttpModule 实现针对 HttpHandler 的动态映射

2.3.1 UrlRoutingModule

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

```

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

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

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

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

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

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

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

2.3.2 PageRouteHandler 与 MvcRouteHandler

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

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

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

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

```

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

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

```

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

```

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

```

2.3.3 ASP.NET 路由系统扩展

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

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

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

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

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

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

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

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

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

```

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

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

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

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

    Uri virtualPath = this.UriTemplate.BindByName(baseAddress, variables);
}

```

```

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

```

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

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

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

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

```

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

```

在注册的 `Route` 指向的目标页面 `Weather.aspx` 的后台代码中，我们定义了如下一个

GenerateUrl 方法，它会根据指定的区号（areacode）和预报天数（days）创建一个 URL。在该方法中，我们通过 RouteTable 的静态属性 Routes 得到代表全局路由表的 RouteCollection 对象，并调用其 GetVirtualPathData 方法来生成最终返回的 URL。

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

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

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

```

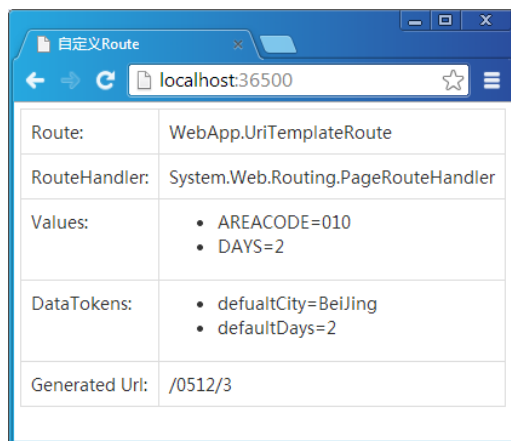


```

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

```

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



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

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