在一个ASP.NET MVC应用来说，针对HTTP请求的处理和相应定义Controller类型的某个Action方法中，每个HTTP请求的目标对象不再像ASP .NET Web Form应用一样是一个物理文件，而是某个Controller的某个Action。目标Controller和Action的名称包含在HTTP请求中，而ASP.NET MVC的首要任务就是通过当前HTTP请求的解析得到正确的Controller和Action的名称。这个过程是通过ASP.NET MVC的URL路由机制来实现的。

**一、RouteData**

ASP.NET定义了一个全局的路由表，路由表中的每个路由对象对应着一个将Controller和Action名称作为站位符的URL模板。对于每一个抵达的HTTP请求，ASP.NET MVC会遍历路由表找到一个URL模板的模式与请求地址相匹配的路有对象，并最终解析出以Controller和Action名称为核心的路由数据。在我们自定义的ASP.NET MVC框架中，路由数据通过具有如下定义的RouteData类型表示。

1: public class RouteData

2: {

3: public IDictionary<string, object> Values { get; private set; }

4: public IDictionary<string, object> DataTokens { get; private set; }

5: public IRouteHandler RouteHandler { get; set; }

6: public RouteBase Route { get; set; }

7:

8: public RouteData()

9: {

10: this.Values = new Dictionary<string, object>();

11: this.DataTokens = new Dictionary<string, object>();

12: this.DataTokens.Add("namespaces", new List<string>());

13: }

14: public string Controller

15: {

16: get

17: {

18: object controllerName = string.Empty;

19: this.Values.TryGetValue("controller", out controllerName);

20: return controllerName.ToString();

21: }

22: }

23: public string ActionName

24: {

25: get

26: {

27: object actionName = string.Empty;

28: this.Values.TryGetValue("action", out actionName);

29: return actionName.ToString();

30: }

31: }

32: public IEnumerable<string> Namespaces

33: {

34: get

35: {

36: return (IEnumerable<string>)this.DataTokens["namespaces"];

37: }

38: }

39: }

从上面的代码片断所示，RouteData定义了两个字典类型的属性Values和DataTokens，前者代表直接从请求地址解析出来的变量，后者代表其他类型的变量。表示Controller和Action名称的同名属性直接从Values字典中提取，对应的Key分别为controller和action。属性Namespaces表示辅助Controller类型的解析而设置的命名空间列表，该属性值从DataTokens字典中提取，对应的Key为namespaces。

我们之前已经提到过ASP.NET MVC本质上是两个自定义的ASP.NET组件来实现的，一个是自定义的HttpModule，另一个是自定义的HttpHandler，而后者从RouteData的RouteHandler属性获得。RouteData的RouteHandler属性类型为IRouteHandler接口，如下面的代码片断所示，该接口具有一个唯一的GetHttpHandler用于返回真正用于处理HTTP请求的HttpHandler对象。

1: public interface IRouteHandler

2: {

3: IHttpHandler GetHttpHandler(RequestContext requestContext);

4: }

IRouteHandler接口的GetHttpHandler方法接受一个类型为RequestContext的参数。顾名思义，RequestContext表示当前（HTTP）请求的上下文，其核心就是对当前HttpContext和RouteData的封装，这可以通过如下的代码片断看出来。

1: public class RequestContext

2: {

3: public virtual HttpContextBase HttpContext { get; set; }

4: public virtual RouteData RouteData { get; set; }

5: }

**二、Route和RouteTable**

RouteData具有一个类型为RouteBase的Route属性，表示当前路由表中与当前请求匹配的路由对象。换句话说，当前的RouteData就是通过该路由对象针对当前HTTP请求进行解析获得的。RouteBase是一个抽象类，如下面的代码片断所示，它仅仅包含一个GetRouteData方法，该方法通过对以HttpContextBase对象表示的当前HTTP上下文进行解析从而获取一个RouteData对象。

1: public abstract class RouteBase

2: {

3: public abstract RouteData GetRouteData(HttpContextBase httpContext);

4: }

ASP.NET MVC提供的基于URL模板的路由机制是通过具有如下定义的Route类型实现的。Route是RouteBase的子类，字符串类型的Url属性代表定义的URL模板 。在实现的GetRouteData方法中，通过HttpContextBase获取相对请求地址，如果该地址与定义在模板中的URL模式相匹配则创建一个RouteData返回；否则返回Null。对于返回的RouteData对象，其Values属性表示的字典包含直接通过地址解析出来的变量，而对于DataTokens字典和RouteHandler属性，则直接取自Route对象的同名属性。

1: public class Route : RouteBase

2: {

3: public IRouteHandler RouteHandler { get; set; }

4: public Route()

5: {

6: this.DataTokens = new Dictionary<string, object>();

7: this.RouteHandler = new MvcRouteHandler();

8: }

9: public override RouteData GetRouteData(HttpContextBase httpContext)

10: {

11: IDictionary<string, object> variables;

12: if (this.Match(httpContext.Request.AppRelativeCurrentExecutionFilePath.Substring(2), out variables))

13: {

14: RouteData routeData = new RouteData();

15: foreach (var item in variables)

16: {

17: routeData.Values.Add(item.Key, item.Value);

18: }

19: foreach (var item in DataTokens)

20: {

21: routeData.DataTokens.Add(item.Key, item.Value);

22: }

23: routeData.RouteHandler = this.RouteHandler;

24: return routeData;

25: }

26: return null;

27: }

28: public string Url { get; set; }

29: public IDictionary<string, object> DataTokens { get; set; }

30: protected bool Match(string requestUrl, out IDictionary<string,object> variables)

31: {

32: variables = new Dictionary<string,object>();

33: string[] strArray1 = requestUrl.Split('/');

34: string[] strArray2 = this.Url.Split('/');

35: if (strArray1.Length != strArray2.Length)

36: {

37: return false;

38: }

39:

40: for (int i = 0; i < strArray2.Length; i++)

41: {

42: if(strArray2[i].StartsWith("{") && strArray2[i].EndsWith("}"))

43: {

44: variables.Add(strArray2[i].Trim("{}".ToCharArray()),strArray1[i]);

45: }

46: }

47: return true;

48: }

49: }

由于同一个Web应用可以采用多种不同的URL模式，所以也需要注册多个继承自RouteBase的路由对象对它们进行解析，多个路由对象组成了一个路由表。在我们自定义ASP.NET MVC框架中，路由表通过类型RouteTable表示。如下面的代码片断所示，RouteTable仅仅具有一个类型为RouteDictionary的Routes属性表示针对真个Web应用的全局路由表。

1: public class RouteTable

2: {

3: public static RouteDictionary Routes { get; private set; }

4: static RouteTable()

5: {

6: Routes = new RouteDictionary();

7: }

8: }

RouteDictionary表示一个具名的路由对象的列表，我们直接让它继承自Dictionary<string, RouteBase>类型，其中的Key表示注册的路由对象的名称。在GetRouteData方法中，我们遍历集合找到与指定的HttpContextBase对象匹配的路由对象，并得到对应的RouteData。

1: public class RouteDictionary: Dictionary<string, RouteBase>

2: {

3: public RouteData GetRouteData(HttpContextBase httpContext)

4: {

5: foreach (var route in this.Values)

6: {

7: RouteData routeData = route.GetRouteData(httpContext);

8: if (null != routeData)

9: {

10: return routeData;

11: }

12: }

13: return null;

14: }

15: }

在Global.asax中我们创建了一个基于指定URL模板的Route对象，并将其添加到通过RouteTable的静态只读属性Routes表示的全局路由表中。

**三、UrlRoutingModule**

路由表的目的在于对当前的HTTP请求进行解析从而获取一个以Controller和Action名称为核心的路由数据，即上面介绍的RouteData，而整个解析工作是通过一个类型为UrlRoutingModule的自定义HttpModule来完成的。如下面的代码片断所示，在实现了接口IHttpModule的UrlRoutingModule类型的Init方法中，我们注册了HttpApplicataion的PostResolveRequestCache事件。

1: public class UrlRoutingModule: IHttpModule

2: {

3: public void Dispose()

4: {}

5: public void Init(HttpApplication context)

6: {

7: context.PostResolveRequestCache += OnPostResolveRequestCache;

8: }

9: protected virtual void OnPostResolveRequestCache(object sender, EventArgs e)

10: {

11: HttpContextWrapper httpContext = new HttpContextWrapper(HttpContext.Current);

12: RouteData routeData = RouteTable.Routes.GetRouteData(httpContext);

13: if (null == routeData)

14: {

15: return;

16: }

17: RequestContext requestContext = new RequestContext { RouteData = routeData, HttpContext = httpContext };

18: IHttpHandler handler = routeData.RouteHandler.GetHttpHandler(requestContext);

19: httpContext.RemapHandler(handler);

20: }

21: }

当PostResolveRequestCache事件触发之后，UrlRoutingModule通过RouteTable的静态只读属性Routes得到表示全局路由表的RouteDictionary对象，然后调用其GetRouteData方法并传入用于封装当前HttpContext的HttpContextWrapper对象（HttpContextWrapper是HttpContextBase的子类）并得到一个封装了路由数据的RouteData对象。如果得到的RouteData不为空，根据该对象本身和和之前得到的HttpContextWrapper对象创建一个表示当前请求上下文的RequestContext对象，将其作为参数传入RouteData的RouteHandler的GetHttpHandler方法得到一个HttpHandler对象。最后我们调用HttpContextWrapper对象的RemapHandler方法对得到HttpHandler进行映射，这意味着该HttpHandler将最终用于处理当前的HTTP请求。

[ASP.NET MVC是如何运行的[1]: 建立在“伪”MVC框架上的Web应用](http://www.cnblogs.com/artech/archive/2012/03/11/mvc-how-to-work-01.html)   
[ASP.NET MVC是如何运行的[2]: URL路由](http://www.cnblogs.com/artech/archive/2012/03/11/mvc-how-to-work-02.html)   
[ASP.NET MVC是如何运行的[3]: Controller的激活](http://www.cnblogs.com/artech/archive/2012/03/12/mvc-how-to-work-03.html)   
[ASP.NET MVC是如何运行的[4]: Action的执行](http://www.cnblogs.com/artech/archive/2012/03/12/mvc-how-to-work-04.html)