ASP.NET MVC的URL路由系统通过注册的路由表对HTTP请求进行解析从而得到一个用于封装路由数据的RouteData对象，而这个过程是通过自定义的UrlRoutingModule对HttpApplication的PostResolveRequestCache事件进行注册实现的。RouteData中已经包含了目标Controller的名称，现在我们来进一步分析真正的Controller对象是如何被激活的。我们首先需要了解一个类型为MvcRouteHandler的类型。

**一、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，如下的代码片断反映了这一点。

1: public class Route : RouteBase

2: {

3: //其他成员

4: public IRouteHandler RouteHandler { get; set; }

5: public Route()

6: {

7: //其他操作

8: this.RouteHandler = new MvcRouteHandler();

9: }

10: }

对于我们这个“迷你版”的ASP.NET MVC框架来说，MvcRouteHandler是一个具有如下定义的类型。在实现的GetHttpHandler方法中，它直接返回一个MvcHandler对象。

1: public class MvcRouteHandler: IRouteHandler

2: {

3: public IHttpHandler GetHttpHandler(RequestContext requestContext)

4: {

5: return new MvcHandler(requestContext);

6: }

7: }

**二、MvcHandler**

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

UrlRoutingModule在通过路由表解析HTTP请求得到一个用于封装路由数据的RouteData后，或调用其RouteHandler的GetHttpHandler方法得到HttpHandler对象并注册到当前的HTTP上下文。由于RouteData的RouteHandler来源于对应Route对象的RouteHandler，而后者在默认的情况下是一个MvcRouteHandler对象，所以默认情况下用于处理HTTP请求的就是这么一个MvcHandler对象。MvcHandler实现了对Controller对象的激活和对相应Action方法的执行。

下面的的代码片断体现了MvcHandler的整个定义，它具有一个类型为RequestContext的属性表示被处理的当前请求上下文，该属性在构造函数指定。在实现的ProcessRequest中实现了对Controller对象的激活和执行。

1: public class MvcHandler: IHttpHandler

2: {

3: public bool IsReusable

4: {

5: get{return false;}

6: }

7: public RequestContext RequestContext { get; private set; }

8: public MvcHandler(RequestContext requestContext)

9: {

10: this.RequestContext = requestContext;

11: }

12: public void ProcessRequest(HttpContext context)

13: {

14: string controllerName = this.RequestContext.RouteData.Controller;

15: IControllerFactory controllerFactory = ControllerBuilder.Current.GetControllerFactory();

16: IController controller = controllerFactory.CreateController(this.RequestContext, controllerName);

17: controller.Execute(this.RequestContext);

18: }

19: }

**三、Controller与ContrllerFactory**

我们为Controller定义了一个接口IController，如下面的代码片断所示，该接口具有唯一的方法Execute在MvcHandler的ProcessRequest方法中被执行，而传入该方法的参数时表示当前请求上下文的RequestContext对象。

1: public interface IController

2: {

3: void Execute(RequestContext requestContext);

4: }

从MvcHandler的定义我们可以看到Controller对象的激活是通过工厂模式实现的，我们为Controller工厂定义了一个具有如下定义的IControllerFactory接口。IControllerFactory通过CreateController方法根据传入的请求上下文和Controller的名称来激活相应的Controller对象。

1: public interface IControllerFactory

2: {

3: IController CreateController(RequestContext requestContext, string controllerName);

4: }

在MvcHandler的ProcessRequest方法中，它通过ControllerBuilder的静态属性Current得到当前的ControllerBuilder对象，并调用GetControllerFactory方法获得当前的ControllerFactory。然后通过从自己的RequestContext中提取的RouteData获得Controller的名称，最后将它连同RequestContext一起作为ContollerFactory的CreateController方法的参数进而创建具体的Controller对象。

ControllerBuilder的整个定义如下面的代码片断所示，表示当前ControllerBuilder的静态只读属性的Current在静态构造函数中被创建。SetControllerFactory和GetControllerFactory方法用于ContorllerFactory的注册和获取。而类型为HashSet<string>的DefaultNamespaces属性表示默认的命名空间列表，这是为了最终解析Controller类型的需要。

1: public class ControllerBuilder

2: {

3: private Func<IControllerFactory> factoryThunk;

4: static ControllerBuilder()

5: {

6: Current = new ControllerBuilder();

7: }

8: public ControllerBuilder()

9: {

10: this.DefaultNamespaces = new HashSet<string>();

11: }

12: public static ControllerBuilder Current { get; private set; }

13: public IControllerFactory GetControllerFactory()

14: {

15: return factoryThunk();

16: }

17: public void SetControllerFactory(IControllerFactory controllerFactory)

18: {

19: factoryThunk = () => controllerFactory;

20: }

21: public HashSet<string> DefaultNamespaces { get; private set; }

22: }

在回头看看我们之前建立在我们自定义ASP.NET MVC框架的Web应用，我们就是通过当前的ControllerBuilder进行ControllerFactory的注册和默认命名空间的指定的。如下面的代码片断所示，我们注册的ControllerFactory的类型为DefaultControllerFactory。

1: public class Global : System.Web.HttpApplication

2: {

3: protected void Application\_Start(object sender, EventArgs e)

4: {

5: //其他操作

6: ControllerBuilder.Current.SetControllerFactory(new DefaultControllerFactory());

7: ControllerBuilder.Current.DefaultNamespaces.Add("WebApp");

8: }

9: }

作为默认ControllerFactory的DefualtControllerFactory类型定义如下。激活Controller类型的前提是能够正确解析出Controller的真实类型。作为CreateController方法输入参数的controllerName仅仅表示Controller的名称，我们需要加上Controller字符后缀作为类型名称。此外我们还需要得到类型的命名空间，而命名空间具有两个来源，即RouteData和当前ControllerBuilder。在DefualtControllerFactory初始化过程中，我们通过BuildManager加载所有应用的程序集，并加载所有实现了接口IController的类型并保存起来，而在CreateController方法中根据Controller的名称和命名空间从保存的Controller类型列表中得到对应的Controller类型，并通过反射的方式创建它。

1: public class DefaultControllerFactory : IControllerFactory

2: {

3: private List<Type> controllerTypes = new List<Type>();

4: public DefaultControllerFactory()

5: {

6: foreach (Assembly assembly in BuildManager.GetReferencedAssemblies())

7: {

8: foreach (Type type in assembly.GetTypes().Where(type => typeof(IController).IsAssignableFrom(type)))

9: {

10: controllerTypes.Add(type);

11: }

12: }

13: }

14: public IController CreateController(RequestContext requestContext, string controllerName)

15: {

16: string typeName = controllerName + "Controller";

17: List<string> namespaces = new List<string>();

18: namespaces.AddRange(requestContext.RouteData.Namespaces);

19: namespaces.AddRange(ControllerBuilder.Current.DefaultNamespaces);

20: foreach (var ns in namespaces)

21: {

22: string controllerTypeName = string.Format("{0}.{1}", ns, typeName);

23: Type controllerType = controllerTypes.FirstOrDefault(type => string.Compare(type.FullName, controllerTypeName, true) == 0);

24: if (null != controllerType)

25: {

26: return (IController)Activator.CreateInstance(controllerType);

27: }

28: }

29: return null;

30: }

31: }

上面我们详细地介绍了Controller的激活原理，我们现在讲关注点返回到Controller自身。通过实现IContrller接口，我们为具有的Controller定义了一个具有如下定义的ControllerBase抽象基类。从中我们可以看到在实现的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。

1: public abstract class ControllerBase: IController

2: {

3: protected IActionInvoker ActionInvoker { get; set; }

4: public ControllerBase()

5: {

6: this.ActionInvoker = new ControllerActionInvoker();

7: }

8: public void Execute(RequestContext requestContext)

9: {

10: ControllerContext context = new ControllerContext { RequestContext = requestContext, Controller = this };

11: string actionName = requestContext.RouteData.ActionName;

12: this.ActionInvoker.InvokeAction(context, actionName);

13: }

14: }

[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击激活](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)