# 控制器可扩展性

这部分主要研究一下如何配置控制器工厂和动作调用器，以便了解如何控制它们的行为；另外，也看看如何能够替代这些组件，并使用自己的逻辑。

下图是一个请求经过各组件时的基本流程，它演示了请求的处理过程：



调用一个动作方法

（一个请求在调用动作方法时的处理管道）

## 准备示例项目

为了解控制器的可扩展性，需要创建一个新项目。

项目名称：ControllerExtensibility

项目模板：Empty

基本结构：

* Models文件夹：Result.cs类——定义Result Model对象
* Views/ Shared文件夹：Result.cshtml视图——一个渲染控制器类中所有动作方法的视图（以Result类作为其模型，简单地显示ControllerName和ActionName）
* 控制器：Product、Customer

下面看看上述基本结构中各项的代码示例：

1、Result Model对象：

namespace ControllerExtensibility.Models

{

public class Result

{

public string ControllerName { get; set; }

public string ActionName { get; set; }

}

}

2、Result.cshtml视图：

@model ControllerExtensibility.Models.Result

@{

Layout = null;

}

<!DOCTYPE html>

<html>

<head>

<meta name="viewport" content="width=device-width" />

<title>Result</title>

</head>

<body>

<div>Controller: @Model.ControllerName</div>

<div>Action: @Model.ActionName</div>

</body>

</html>

3、Product控制器：

using ControllerExtensibility.Models;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace ControllerExtensibility.Controllers

{

public class ProductController : Controller

{

public ViewResult Index()

{

return View("Result", new Result { ControllerName = "Product", ActionName = "Index" });

}

public ViewResult List()

{

return View("Result", new Result { ControllerName = "Product", ActionName = "List" });

}

}

}

4、Customer控制器

using ControllerExtensibility.Models;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace ControllerExtensibility.Controllers

{

public class CustomerController : Controller

{

public ViewResult Index()

{

return View("Result", new Result { ControllerName = "Customer", ActionName = "Index" });

}

public ViewResult List()

{

return View("Result", new Result { ControllerName = "Customer", ActionName = "List" });

}

}

}

主要目的：研究MVC框架提供的对控制器和动作的管理进行定制的方式。

## 创建自定义控制器工厂

虽然实际项目中建议通过对内建的控制器工厂进行扩展，但现在需要理解其工作原理，所以，我们通过创建一个自定义控制器工厂进行探究。

控制器工厂是由IControllerFactory接口定义的，下面是接口的定义：

using System;

using System.Web.Routing;

using System.Web.SessionState;

namespace System.Web.Mvc

{

public interface IControllerFactory

{

IController CreateController(RequestContext requestContext, string controllerName);

SessionStateBehavior GetControllerSessionBehavior(RequestContext requestContext, string controllerName);

void ReleaseController(IController controller);

}

}

现在创建一个Infrastructure文件夹，并在其中建立一个简单的控制器工厂CustomControllerFactory：

using ControllerExtensibility.Controllers;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

using System.Web.Routing;

using System.Web.SessionState;

namespace ControllerExtensibility.Infrastructure

{

public class CustomControllerFactory : IControllerFactory

{

public IController CreateController(RequestContext requestContext, string controllerName)

{

Type targetType = null;

switch (controllerName)

{

case "Product":

targetType = typeof(ProductController);

break;

case "Customer":

targetType = typeof(CustomerController);

break;

default:

requestContext.RouteData.Values["controller"] = "Product";

targetType = typeof(ProductController);

break;

}

return targetType == null ? null : (IController)DependencyResolver.Current.GetService(targetType);

}

public SessionStateBehavior GetControllerSessionBehavior(RequestContext requestContext, string controllerName)

{

return SessionStateBehavior.Default;

}

public void ReleaseController(IController controller)

{

IDisposable disposable = controller as IDisposable;

if (controller != null)

{

disposable.Dispose();

}

}

}

}

上面代码中最重要的是CreateController方法，在MVC需要控制器对请求进行服务时调用。该方法的RequestContext类型参数能提供请求的细节内容；另一个字符串参数则提供的是controller值，该值是从URL那里得到的。

RequestContext属性：

* HttpContext：类型为HttpContextBase，用来提供HTTP请求的信息
* RouteData：类型为RouteData，提供与请求匹配的路由信息

如果在开发的时候像上面那样自行创建控制器工厂将是一件很麻烦的事——在Web程序中查找控制器类并对它们实例化是复杂的，如需要能够动态并一致地定位控制器类，并能处理其中各种潜在的问题（如：消除不同命名空间中同名类之间的歧义、构造器异常，以及其他一些问题）。

我们的示例项目中只有两个控制器，且打算直接对它们进行实例化，即将类名强行写入控制器工厂——这在实际项目中是极其不明智的。

CreateController方法的目的是，创建能够对当前请求进行处理的控制器实例。至于具体做法是没有任何限制的。唯一的规则，作为该方法的结果，必修返回一个实现IController接口的对象。

在创建控制器工厂时可以遵循MVC框架的约定，也可以放弃这些约定而创建适合于自己项目需要的约定。如果只是单纯为了创建自己的约定，只能有助于理解MVC框架的灵活性，但这不是我们提倡的做法。

### 处理备用控制器

在示例项目中，当请求的控制器与任一控制器都不匹配，将以ProductController类为目标。这在实际项目中可能不是最好的做法，但它表明控制器工厂对请求的解析有充分的灵活性。但这需要我们了解MVC框架中的其他切入点是如何操作的。

默认情况下，MVC框架会根据路由数据中controller的值来选择视图，而不是控制器类的名称。所以，如果希望备用位置按照控制器名称组织的约定来使用视图，就需要改变controller路由属性的值，就像这样：

requestContext.RouteData.Values["controller"] = "Product";

这一改变将导致MVC框架搜索备用的控制器相关的视图，而不是用户请求的控制器。

这里有两个重要的切入点：

1. 控制器工厂不仅要独自负责将请求与控制器进行匹配，而且它还可以对请求进行修改，以改变请求处理管道中后续步骤的行为。这是MVC框架强有力的要素和关键特征。
2. 尽管在控制器工厂中可以自由选择尊重哪种约定，但仍需要了解MVC框架其他部分的约定。并且，因为其他组件也可以改用自定义代码。因此，遵循尽可能多的约定，以允许组件彼此独立地开发和使用是有意义的。

### 实例化控制器类

对控制器类的实例化没有硬性的规则，其中比较好的方法如通过依赖性解析器实例化。使用这种方式可以让我们在开发自定义控制器工厂时专注于请求与控制器类之间的映射，而将依赖性注入这样的问题留下来单独处理，并用于整个程序。如下面使用DependencyResolver类实现控制器的实例化的示例：

return targetType == null ? null : (IController)DependencyResolver.Current.GetService(targetType);

注意，上面示例代码没有检查最终返回的对象是否是IController的实现，但在实际项目中最好进行类型的检测。

### 实现其他接口方法

IcontrollerFactory接口中的另外两个方法如下：

* GetControllerSessionBehavior方法由MVC框架用来确定是否应该为控制器维护会话数据
* 当不再需要CreateController方法创建的控制器对象时，会调用ReleaseController方法。前面示例中检查了这个类是否实现IDisposable接口。如果是，则调用Dispose方法以释放那些可以释放的资源。

这两个方法的实现适用于大多数项目，且可以直接照搬。（一些具体的实现请看后续介绍）

### 注册自定义控制器工厂

通过ControllerBuilder类注册自定义控制器工厂。在程序启动时必须注册自定义工厂控制器，即在Global.asax.cs中使用Application\_Start方法，如：

**using ControllerExtensibility.Infrastructure;**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Http;

using System.Web.Mvc;

using System.Web.Routing;

namespace ControllerExtensibility

{

// 注意: 有关启用 IIS6 或 IIS7 经典模式的说明，

// 请访问 http://go.microsoft.com/?LinkId=9394801

public class MvcApplication : System.Web.HttpApplication

{

protected void Application\_Start()

{

AreaRegistration.RegisterAllAreas();

WebApiConfig.Register(GlobalConfiguration.Configuration);

FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);

RouteConfig.RegisterRoutes(RouteTable.Routes);

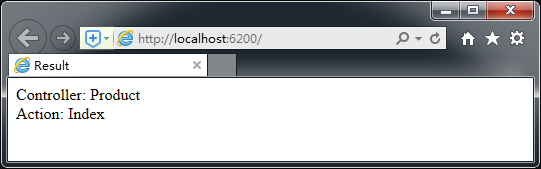
**ControllerBuilder.Current.SetControllerFactory(new CustomControllerFactory());**

}

}

}

此时启动程序将得到预期效果——浏览器将请求跟URL，并被路由系统映射到Home控制器。自定义工厂会通过创建ProductController类的实例来处理对Home控制器的请求，如图：



## 使用内建的控制器工厂

在了解了控制器工厂的工作机制后接着来看看内建的控制器工厂，对于大多数程序，内建的控制器工厂类——DefaultControllerFactory是完全足够的。当从路由系统收到一个请求时，该工厂会考察路由数据，找到controller属性的值，并试图在这个Web应用程序中找到满足以下条件的类（控制器类的条件）：

* 这个类必须是一个公共类（public类）。
* 这个类必须是具体类。
* 这个类必须没有泛型（Generic）参数。
* 类名必须以Controller结尾。
* 这个类必须实现IController接口。

DefaultControllerFactory类维护着程序中这些类的一个列表，且在请求到达时不需要每次都执行一次搜索。如果找到一个合适的类，便用控制器激活器（Controller Activator）创建一个实例，控制器的工作便完成了。如果没有匹配的控制器，那么便不能对该请求作进一步处理。

因为DefaultControllerFactory是遵循“约定优于配置”的，所以不需要在配置文件中注册控制器，需要做的全部工作仅仅是创建满足这个工厂查寻条件的类。

如果希望创建自定义控制器工厂的行为，可以对默认工厂的设置进行配置，或重写它的一些方法。这样，便能建立有用的“约定优于配置”行为，而不必向之前那样重新创建它。

### 命名空间优先排序

在创建路由时，做好对一个或多个命名空间的优先级设置可以解决控制器的多义性问题。且对命名空间列表处理并对其优先排序的就是这个控制器工厂：DefaultControllerFactory。

提示：Global优先级会被路由优先级所重写。即，可以定义一个全局策略，然后在必要时定制个别路由。

如果程序或项目中存在多个路由，那么全局性地指定命名空间优先级可能会更方便一些，以使这种优先级能够应用所有的这些路由。下面演示了如何在Global.asax文件的Application\_Start方法中实现这一功能（当然也可以使用App\_Start文件夹中的RouteConfig.cs文件）：

using ControllerExtensibility.Infrastructure;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Http;

using System.Web.Mvc;

using System.Web.Routing;

namespace ControllerExtensibility

{

// 注意: 有关启用 IIS6 或 IIS7 经典模式的说明，

// 请访问 http://go.microsoft.com/?LinkId=9394801

public class MvcApplication : System.Web.HttpApplication

{

protected void Application\_Start()

{

AreaRegistration.RegisterAllAreas();

WebApiConfig.Register(GlobalConfiguration.Configuration);

FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);

RouteConfig.RegisterRoutes(RouteTable.Routes);

**ControllerBuilder.Current.DefaultNamespaces.Add("MyControllerNamespace");**

**ControllerBuilder.Current.DefaultNamespaces.Add("MyProbject.\*");**

}

}

}

上面示例代码中加粗部分添加给予优先的命名空间，这些被添加的命名空间并不分优先级，所谓优先级只是相对未被添加的命名空间而言的。也就是说当控制器工厂在Add方法定义的命名空间中找不到合适的控制器类，那么将搜索整个程序。

提示：上面的第二条设置命名空间的语句使用了星号（\*）。这表明控制器工厂应该查找MyProject命名空间及其所包含的任意子命名空间。这里可以用“.\*”作为命名空间的结尾，但不能在Add方法中使用任何正则表达式语法。

### 定制DefaultControllerFactory的控制器实例化

定制DefaultControllerFactory类如何实例化控制器对象有许多方式。一般来讲对控制器工厂进行定制普遍的原因是为了添加对DI的支持。这些方法中哪种最合适这要取决于在程序中的其他地方如何使用DI。

1. 使用依赖性解析器

在依赖性解析器（Dependency Resolver）可用时，DefaultControllerFactory类将用它来创建控制器。

DefaultControllerFactory会调用IDependencyResolver. GetService方法，以请求控制器实例，这为解析并注入依赖性提供了机会。

1. 使用控制器激活器

也可以通过创建一个控制器激活器（Controller Activator）的方法，将DI引入到控制器中。通过实现IControllerActivator接口创建激活器，IControllerActivator接口如下：

using System;

using System.Web.Routing;

namespace System.Web.Mvc

{

public interface IControllerActivator

{

IController Create(RequestContext requestContext, Type controllerType);

}

}

该接口的Create方法需要一个描述请求的RequestContext对象和一个指定应该对哪个控制器类进行实例化的类型（Type）。如下示例：

using ControllerExtensibility.Controllers;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

using System.Web.Routing;

namespace ControllerExtensibility.Infrastructure

{

public class CustomControllerAtivator : IControllerActivator

{

public IController Create(RequestContext requestContext, Type controllerType)

{

if (controllerType == typeof(ProductController))

{

controllerType = typeof(CustomerController);

}

return (IController)DependencyResolver.Current.GetService(controllerType);

}

}

}

该示例很简单，如果请求的是ProductController类，将以CustomerController类的实例作为其响应。这里这么做仅仅是为了演示如何利用IControllerActivator接口在控制器工厂和依赖性解析器之间截取请求。

为了使用这个自定义的激活器，需要为DefaultControllerFactory的构造函数传递一个实现类的实例，而且需要在Global.asax文件的Application\_Start方法中进行注册，如：

using ControllerExtensibility.Infrastructure;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Http;

using System.Web.Mvc;

using System.Web.Routing;

namespace ControllerExtensibility

{

// 注意: 有关启用 IIS6 或 IIS7 经典模式的说明，

// 请访问 http://go.microsoft.com/?LinkId=9394801

public class MvcApplication : System.Web.HttpApplication

{

protected void Application\_Start()

{

AreaRegistration.RegisterAllAreas();

WebApiConfig.Register(GlobalConfiguration.Configuration);

FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);

RouteConfig.RegisterRoutes(RouteTable.Routes);

**ControllerBuilder.Current.SetControllerFactory(**

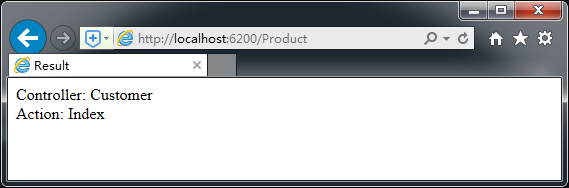
**new DefaultControllerFactory(new CustomControllerAtivator()));**

}

}

}

启动程序，并导航至/Product，就好看到这一自定义激活器的效果。路由将以Product控制器为目标，而DefaultControllerFactory会要求激活器对ProductController类进行实例化，但激活器截取了这一请求，反而创建了一个CustomerController实例对象进行替代。结果图如下：



1. 重写DefaultControllerFactory方法

还可以通过重写DefaultControllerFactory类中的方法来定制控制器的创建。下表给出了可以重写的三个方法，具体见表：

|  |  |  |
| --- | --- | --- |
| 方法 | 结果 | 描述 |
| CreateController | IController | IcontrollerFactory接口的CreateController方法的实现。默认情况下，这个方法调用GetControllerType来确定应该实例化哪一个类，然后通过将结果传递给GetControllerInstance方法，来获得一个控制器对象 |
| GetControllerType | Type | 检索指定名称和请求上下文的控制器类型 |
| GetControllerInstance | IController | 创建指定类型的一个实例 |

## 创建自定义动作调用器

控制器是通过Controller派生而来，而动作方法是由动作调用器（Action Invoker）调用的。动作调用器的接口为：

namespace System.Web.Mvc

{

public interface IActionInvoker

{

bool InvokeAction(ControllerContext controllerContext, string actionName);

}

}

该接口唯一成员InvokeAction的参数是一个ControllerContext对象和一个含有待调用动作名称的字符串。该成员方法返回的是一个布尔型结果，如果是“true”则表示找到并调用了这个动作；“false”表示控制器没有匹配的动作。

请注意，上面说的是“动作”，并未使用“方法”或“动作方法”这样的词汇来描述，是因为动作与方法之间的关联是严格可选的。虽然这是内建的动作调用器所采取的办法，但可以采取选择的任何方式随意地处理动作。如下面的实现：

注：动作是一种行为，而动作方法是实现这种行为的代码。动作调用器的作用是实现对一个动作的调用，而控制器中才是实现这个动作的动作方法。也就是说动作与动作方法的名是可以不同的，后面介绍到“使用自定动作名”的内容时将会进一步了解这些区别。

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace ControllerExtensibility.Infrastructure

{

public class CustomActionInvoker : IActionInvoker

{

public bool InvokeAction(ControllerContext controllerContext, string actionName)

{

if (actionName == "Index")

{

controllerContext.HttpContext.Response.Write("This is output from the Index action");

return true;

}

else

{

return false;

}

}

}

}

动作调用器实际上并不关心控制器类中的方法，具体的方法是其自行处理的。在示例代码中，如果是对Index动作的请求，那么该调用器直接将一条消息写到Response。如果是对其他动作的请求，则返回false，这样MVC框架将报告一个“404——未找到”的错误消息显示给用户。

与一个控制器相关联的动作调用器是通过Controller.ActionInvoker属性获得的。这样表明不同的控制器可以使用不同的动作调用器。现在添加一个名为ActionInvoker的新控制器来进行演示：

using ControllerExtensibility.Infrastructure;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace ControllerExtensibility.Controllers

{

public class ActionInvokerController : Controller

{

//

// GET: /ActionInvoker/

public ActionInvokerController()

{

this.ActionInvoker = new CustomActionInvoker();

}

}

}

在这个示例中我们依靠动作调用器去处理请求而非动作方法。启动程序，并导航至/ActionInvoker/Index，可以看到下面的效果，如果是导航至该控制的其他方法将会看到404的错误页面：



还是那句话，不建议自行实现动作调用器，而且，如果这么做也最好别安装示例这种做法。我们这里主要是为了探究其工作原理，另外内建的支持有一些非常有用的特性，最后在示例代码中存在的问题有缺乏可扩展性、贫乏的职责分离，而且缺乏对各种视图的支持。（当然在请求处理管道的几乎每一个方面都是可定制或完全可替换的）