# 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);

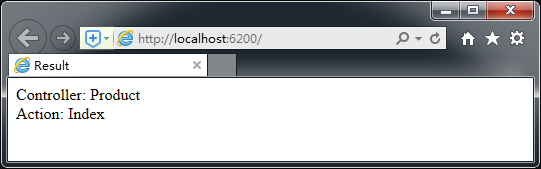
**ControllerBuilder.Current.SetControllerFactory(new CustomControllerFactory());**

}

}

}

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