[**第二节：各种路由约束(动态路由、静态路由、组合路由、正则约束、命名空间约束、区域内路由)**](https://www.cnblogs.com/yaopengfei/p/7885597.html)

一. 什么是路由

 　　路由是约束URL的一组规范，那么什么是URL呢？通俗的来说URL是一个地址，通过该地址，用户可以访问Web网站或者下载服务器上的文件。

 　　  比如下面就是两组URL：

　　　　http://www.cnblogs.com/yaopengfei/p/7828441.html

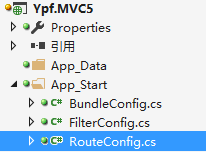
　　　　http://www.cnblogs.com/yaopengfei/p/7828441

　　显然我们喜欢第二组，省略.html，会使该地址看起来更加简洁，更加友好，利于SEO优化。

　　那么我们怎么实现这个简单的需求呢？

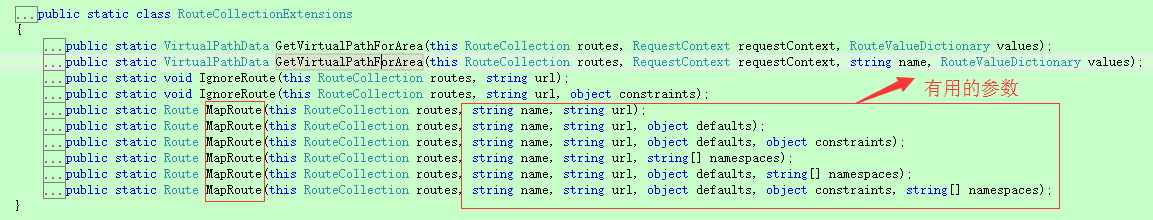
**答案是：通过【路由】配置，所以现在我们似乎有点清晰了，路由可以规定URL的特殊格式，使其达到特殊效果。**

　　在ASP.NET MVC框架中，通过路由配置URL，使用户的URL请求可以映射到Controller下的action方法中，执行相应操作，并接受URL中传过来的参数，在MVC5框架中，在【RouteConfig.cs】类中进行路由规则的配置，如下图：



二. 从源码的角度分析路由

 　　我们进入【RouteConfig】类中，发现核心代码是调用【RouteCollection】类下的MapRoute方法，F12看源码得知，MapRoute方法是RouteCollectionExtensions类的给【RouteCollection】类扩展方法的方法，并且有多个重载，如下图：



**下面分析一下参数的含义：**

　　　　(1) name:  要映射的路由的名称。

　　　　(2) url:  路由的 URL 模式，可以自定义路由的格式，可以写静态路由，也可以写动态路由、组合路由等。

　　　　(3) defaults:  一个包含默认路由值的对象，书写路由的默认值。

　　　　(4) constraints: 一组表达式，可以使用正则指定 url 参数值的约束。

　　　　(5) namespaces: 应用程序的一组命名空间，可以缩小检索路由对象匹配的范围。

**底层源码，有兴趣可以看一下：**

[复制代码](javascript:void(0);)

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

2 {

3 if (routes == null)

4 {

5 throw new ArgumentNullException("routes");

6 }

7 if (url == null)

8 {

9 throw new ArgumentNullException("url");

10 }

11 Route route = new Route(url, new MvcRouteHandler()) {

12 Defaults = CreateRouteValueDictionaryUncached(defaults),

13 Constraints = CreateRouteValueDictionaryUncached(constraints),

14 DataTokens = new RouteValueDictionary()

15 };

16 ConstraintValidation.Validate(route);

17 if ((namespaces != null) && (namespaces.Length > 0))

18 {

19 route.DataTokens["Namespaces"] = namespaces;

20 }

21 routes.Add(name, route);

22 return route;

[复制代码](javascript:void(0);)

三. MVC中的几类路由及其规则

1. 动态路由

1 　　　　　　routes.MapRoute(

2 name: "Default", //路由名称

3 url: "{controller}/{action}/{id}", //路由规则

4 defaults: new { controller = "First", action = "Index1", id = UrlParameter.Optional } //默认值,当Controller或action为空（省略）的时候调用

5 );

分析：路由规则为 {controller}/{action}/{id} ，其中 {controller}、{action}、{id}为三个参数，/ 为格式分割符号，defaults中声明的是默认路由，所以下面的测试结果：

http://localhost:7559/

http://localhost:7559/First

http://localhost:7559/First/index1

http://localhost:7559/First/Index1/2

都会跳转到Index1的页面。

　　变种：将上面的代码的URL分割符号稍微调整一下

1 　　　　　　routes.MapRoute(

2 name: "Default9", //路由名称

3 url: "{controller}/{action}-{id}", //路由规则

4 defaults: new { controller = "First", action = "Index1", id = UrlParameter.Optional } //默认值,当Controller或action为空（省略）的时候调用

5 );

　　请求地址就变成了：

http://localhost:7559/                                               【无法访问】

http://localhost:7559/First                                        【404找不到】

http://localhost:7559/First/index1                            【404找不到】

http://localhost:7559/First/Index1-1                         【可以访问】

2. 静态路由

1 　　　　routes.MapRoute(

2 name: "Default2", //路由名称

3 url: "Ypf", //路由规则，不区分大小写，当输入“ypf”时，会自动跳转到下面的地址

4 defaults: new { controller = "First", action = "Index1", id = UrlParameter.Optional } //默认值,当Controller或action为空的时候调用

5 );

静态路由：url中是一个静态值，访问的URL只有输入这个静态值，才能访问下面default中的默认值。

测试地址如下：

http://localhost:7559/                                            【无法访问】

http://localhost:7559/ypf                                      【 跳转到index1页面】

http://localhost:7559/First/index1                       【无法访问】

**补充一下：MapRoute方法是可以不需要写参数名的，就像正常的调用方法一样，所以上面的代码可以改写：**

1 routes.MapRoute(

2 "Default3", //路由名称

3 "Ypf", //路由规则，不区分大小写，当输入“Ypf”时，会自动跳转到下面的地址

4 new { controller = "First", action = "Index1", id = UrlParameter.Optional } //默认值,当Controller或action为空的时候调用

5 );

3. 组合路由

1 　　routes.MapRoute(

2 "Default4", //路由名称

3 "Ypf/{action}", //路由规则，不区分大小写，规则相符的时候，会自动跳转到下面的地址

4 new { controller = "First", action = "Index1" }

5 );

所谓的组合路由，就是静态路由和动态路由相互组合使用，测试地址如下：

　　http://localhost:7559/ 　　　　　　　　　　　　【无法访问】       **（分析：因为不满足路由规则，没有输入ypf）**

　　http://localhost:7559/ypf 　　　　　　　　　　   【 跳转到index1页面】

　　http://localhost:7559/ypf/index1 　　　　　　      【 跳转到index1页面】

　　http://localhost:7559/Ypf/hh 　　　　　　　　     【404找不到】        **（满足路由规则，但是没有hh这个action，所以404）**

　　http://localhost:7559/ypf/First/index1 　　　　　 【404找不到】       **（满足路由规则，但这里把First当做action，并没有这个action，所以404）**

4. 正则约束

1 　　　　　　routes.MapRoute(

2 "Default5",

3 "{controller}/{action}\_{Year}\_{Month}\_{Day}",

4 new { controller = "First", action = "Index1", id = UrlParameter.Optional },

5 new { Year = @"^\d{4}", Month = @"\d{2}", Day = @"\d{2}" }

6 );//正则路由

所谓的正则约束，是指可以对URL中的参数使用正则表达式进行约束，上述代码约束了Year必须是四位数字，Month和Day必须是两位数字。

测试地址：

　　http://localhost:7559/first/index1\_2018\_09\_01         【跳转到index1页面】

　　http://localhost:7559/first/index1 　　　　　　　　【无法访问】       **（分析：因为不满足路由规则，没有输入{Year}\_{Month}\_{Day} 的参数）**

　　http://localhost:7559/first/ 　　　　　　　　　　　【无法访问】       **（分析：因为不满足路由规则，没有输入{Year}\_{Month}\_{Day} 的参数）**

　　http://localhost:7559/ 　　　　　　　　　　　　   【无法访问】        **（分析：因为不满足路由规则，没有输入{Year}\_{Month}\_{Day} 的参数）**

5. 命名空间约束

1 　　　　 routes.MapRoute(

2 name: "Default6",

3 url: "{controller}/{action}/{id}",

4 defaults: new { controller = "Third", action = "Index", id = UrlParameter.Optional },

5 namespaces: new string[] { "Ypf.MVC5" }

6 );

所谓的命名空间约束，即限定匹配范围路由的检索范围，提高检索速度。

**特别注意：不能从外层控制器直接跳转到内层Area内的控制器！！**

测试地址：以下三个访问地址，都会跳转到index1页面

　　http://localhost:7559/

　　http://localhost:7559/First

　　http://localhost:7559/First/index1

6. Area区域内的路由

[复制代码](javascript:void(0);)

1 public override void RegisterArea(AreaRegistrationContext context)

2 {

3 //原路由

4 //context.MapRoute(

5 // "TestOne\_default",

6 // "TestOne/{controller}/{action}/{id}",

7 // new { action = "Index", id = UrlParameter.Optional }

8 //);

9

10 //结合命名空间进行路由改造

11 context.MapRoute(

12 this.AreaName + "\_default",

13 this.AreaName + "/{controller}/{action}/{id}",

14 new { area = this.AreaName, controller = "Sys\_Admin", action = "Index", id = UrlParameter.Optional },

15 new string[] { "Ypf.MVC5.Areas." + this.AreaName + ".Controllers" }

16 );

17

18 }

[复制代码](javascript:void(0);)

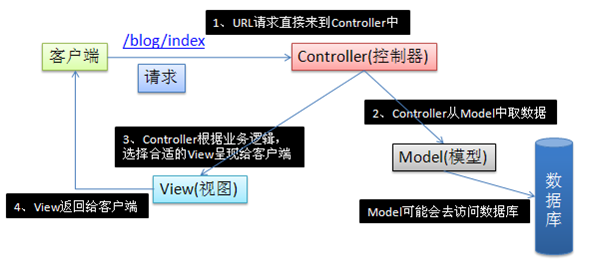
　　指MVC5框架中Area区域单独的一套路由规则，我们可以结合区域内的原路由，进行改造一番，如上述代码。

[**第三节：Action向View传值的四种方式(ViewData、ViewBag、TempData、Model)**](https://www.cnblogs.com/yaopengfei/p/7905648.html)

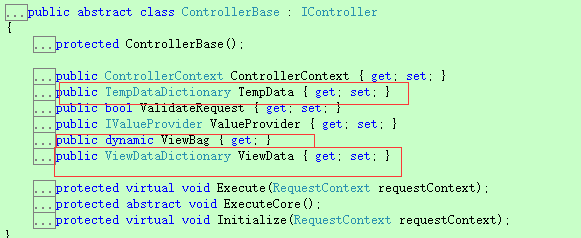
 简  介

 　　在前面的章节中，我们已经很清楚，MVC工作模型的流程，Controller中的Action接收到客户端的请求，处理后要将数据返回给View，那么Action中是如何将数据返回给View的，二者之间打通的桥梁又是什么呢？

　　这些问题正是本章节需要解决的，下面先复习一下MVC的请求模型，如下图：



　(一). 先解决第一个问题，Action向View中传值有四种方式：ViewData、ViewBag、TempData、Model，随意选中一个点击F12查看源码，源码如下:



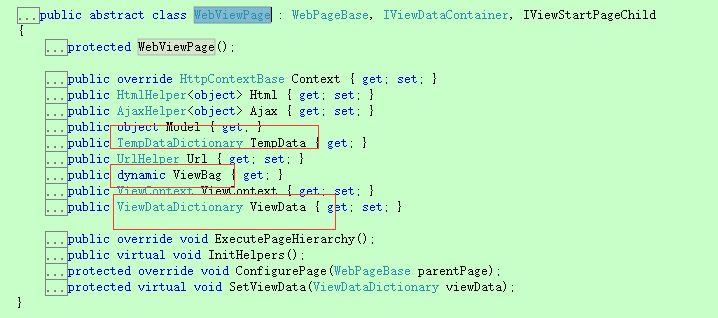
**分析：ViewData和TempData分别是ViewDataDictionary类型和TempDataDictionary类型，而这两种类型均实现了IDictionary接口，所以ViewData和TemData均为字典类型。**

**我们再看一下ViewDataDictionary和TempDataDictionary两个类型源码，以ViewDataDictionary为例，代码如下：**

****

**分析可知赋值方式有两种： ViewData[" "]=XXX   和  ViewData.Add("key", value) ，TempData赋值方式与此类似。**

(二). 接下来我们解决第二个问题，在前端页面选择ViewData点击F12，查看WebViewPage类源码，源码如下：这时候，应该都很清楚了。



　  (三). 总结一下结论：

　　A. ViewData：字典类型，在前端页面使用的时候，需要进行类型转换。

　　B. ViewBag：动态类型，运行时自动进行类型转换，不需要进行任何类型转换。

　　C：Model： 实质就是ViewData.Model，前端页面通过Model.XXX进行调用，页面需要using引入程序集。

　　D：TempData：字典类型，前端页面使用时候，需要进行类型转换，但该类型更多的是作为临时变量应用于后台Action直接的传值，它内部是基于Session实现的，它可以存储一次，**但是只能读取一次**，再次使用，将为空。

1. 测试四种方式向页面传值

[复制代码](javascript:void(0);)

1 　　public ActionResult PassValueIndex()

2 {

3 ViewData["num"] = 2;

4 ViewData.Add("num2", 2);

5

6 ViewBag.myNum = 2;

7 TempData["myNum2"] = 2;

8

9 Student stu = new Student()

10 {

11 id="123456",

12 name="ypf",

13 sex="男"

14 };

15 return View(stu);

16 }

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

1 @\*使用Model赋值，需要引入下面的命名空间\*@

2 @using Ypf.MVC5.Models;

3 @{

4 Layout = null;

5 }

6

7 <!DOCTYPE html>

8

9 <html>

10 <head>

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

12 <title>PassValueIndex</title>

13 </head>

14 <body>

15 <div>

16 <p>Model赋值：

17 @Model.id

18 @Model.name

19 @Model.sex

20 </p>

21 <p>ViewData(需要进行类型转换):@((int)ViewData["num"]+1) </p>

22 <p>ViewData(需要进行类型转换):@((int)ViewData["num2"] + 1) </p>

23

24 <p>ViewBag(不需要进行类型转换):@(ViewBag.myNum+1)</p>

25 <p>TempData(需要进行类型转换):@((int)TempData["myNum2"]+1)</p>

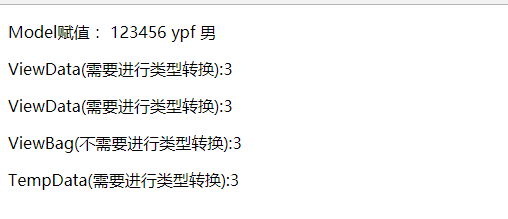
26 <a href="TestTempData1">第二次调用TempData</a>

27 </div>

28 </body>

29 </html>

[复制代码](javascript:void(0);)



2. 测试TempData的时效性

 　　从上面页面前端代码中点击，a标签，进入下面代码：

[复制代码](javascript:void(0);)

1 public ActionResult TestTempData1()

2 {

3 //第二次次测试使用TempData，因为PassValueIndex页面已经使用了一次，

4 //所以此处data1为空，很好的印证了TempData只能调用一次的结论

5 var data1 = TempData["myNum2"];

6 return Content("");

7 }

[复制代码](javascript:void(0);)

　　分析发现，这里的data1为null，印证了TempData使用一次后清空的结论。

[**第四节：MVC中AOP思想的体现(四种过滤器)并结合项目案例说明过滤器的实际用法**](https://www.cnblogs.com/yaopengfei/p/7910763.html)

一. 简介

 　　MVC中的过滤器可以说是MVC框架中的一种灵魂所在，它是MVC框架中AOP思想的具体体现，所以它以面向切面的形式无侵入式的作用于代码的业务逻辑，与业务逻辑代码分离，一经推出，广受开发者的喜爱。

　　那么过滤器到底是什么呢？它又有什么作用呢？

　　用户通过URL访问Web系统不一定都能得到相应的内容，一方面不同的用户权限不同，另一方面是为了保护系统，防止被攻击，这就是**过滤器的核心**所在，我们总计一下过滤器都有哪些作用：

　　①：判断用户是否登录以及不同用户对应不同的权限问题。

　　②：防盗链、防爬虫。

　　③：系统中语言版本的切换(本地化和国际化)。

　　④：权限管理系统中动态Action。

　　⑤：决策输出缓存。

　　知道到了过滤器的作用，那么过滤器分哪几类呢？如下图1：

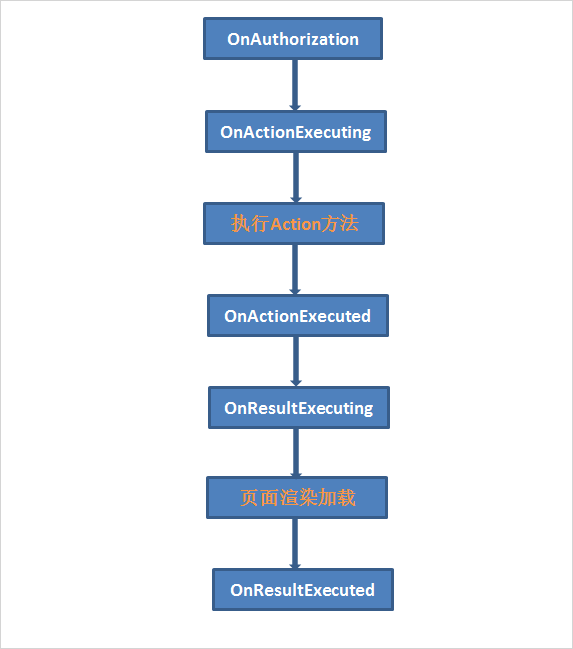


二. 执行顺序

 　　从上图①可知，过滤器分四类，总共重写了六个方法，在这六个方法里可以处理相应的业务逻辑，那么如果四种过滤器的六个重写方法同时存在，它们的执行顺序是什么呢？

 　　首先要将OnException方法除外，该方法不和其余五个方法参与排序问题，该方法独立存在，什么时间报错，什么时候调用。

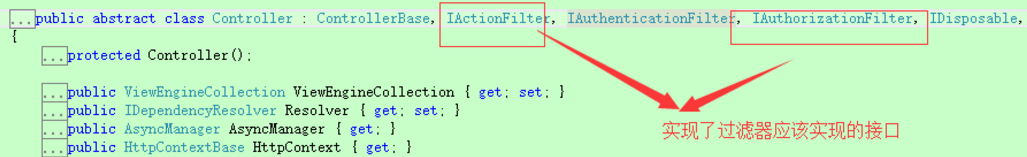
　   其余三种过滤器中的五个重写方法的执行顺序：

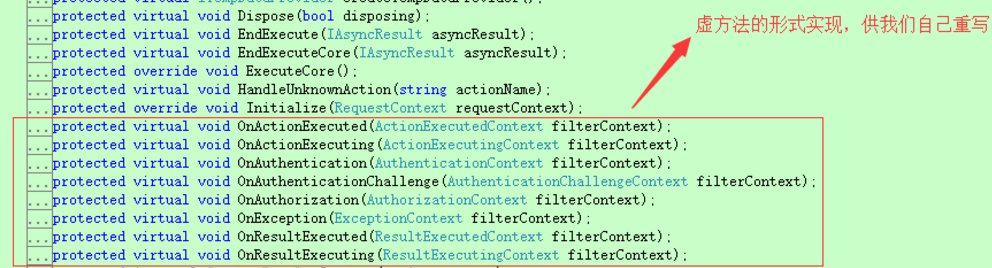


三. 自定义实现形式

1. 直接在控制器中重写方法或者利用控制器间的继承

 　　新建任何一个控制器，它均继承Controller类，F12进入Controller类中，发现Controller类中已经实现了过滤器需要实现的接口，并且提供虚方法供我们重写，代码如下：





　　基于以上原理，这样在控制器级别上我们就有两种思路来实现过滤器。

**方案一：直接在当前控制器重写相应的过滤器方法，则该过滤器的方法作用于当前控制器的所有Action。**



**方案二：新建一个父类控制器，在父类控制器中重写过滤器的方法，然后子类控制器继承该父类控制器，则该该过滤器作用于子类控制器中的所有Action。**

**【该方法和接下来以特性的形式作用于控制器的效果是一致的】**

[复制代码](javascript:void(0);)

1 /// <summary>

2 /// 控制器继承该控制器，和特性作用在控制器上效果一致

3 /// </summary>

4 public class MyBaseFilterController : Controller

5 {

6 //需要用protected类型，不能用public类型

7 protected override void OnAuthorization(AuthorizationContext filterContext)

8 {

9 //1.如果保留如下代码，则会运行.net framework定义好的身份验证，如果希望自定义身份验证，则删除如下代码

10 // base.OnAuthorization(filterContext);

11

12 //2.获取区域名字

13 // string strAreaName = filterContext.RouteData.DataTokens["area"].ToString().ToLower();

14

15 //3.获取控制器作用的Controller和action的名字

16 string controllerName = filterContext.ActionDescriptor.ControllerDescriptor.ControllerName.ToLower();

17 string actionName = filterContext.ActionDescriptor.ActionName.ToLower();

18 filterContext.HttpContext.Response.Write("身份验证过滤器作用于" + controllerName + "控制器下的" + actionName + "方法</br>");

19 }

20 }

[复制代码](javascript:void(0);)

2. 自定义类继承MVC中过滤器实现类或过滤器接口，特性的形式作用于控制器或Action

**特别补充：MVC框架中的AuthorizeAttirbute、ActionFilterAttribute、HandleErrorAttribute类已经实现了过滤器对应的接口，所以我们在自定义过滤器的时候，可以直接继承以上三个类；或者实现相应的接口：IAuthorizationFilter、IActionFilter、IResultFilter、IExceptionFilter。(该方案在实现相应接口的同时，需要继承FilterAttribute，使自定义的类成为一个特性)。**

**下面以继承MVC中实现类的形式来自定义四种过滤器：**

**A：身份验证过滤器**

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttps://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 /// <summary>

2 /// 身份验证过滤器

3 /// 1. 在非MVC框架项目中使用MVC过滤器，需要通过nuget把MVC的程序集添加进去

4 /// 2. 继承AuthorizeAttribute类，然后对OnAuthorization方法进行 override 覆写

5 /// 3. 在Action运行之前首先运行该过滤器

6 /// </summary>

7 public class MyAuthorize : AuthorizeAttribute

8 {

9 public override void OnAuthorization(AuthorizationContext filterContext)

10 {

11 //1.如果保留如下代码，则会运行.net framework定义好的身份验证，如果希望自定义身份验证，则删除如下代码

12 // base.OnAuthorization(filterContext);

13

14 //2.获取区域名字

15 // string strAreaName = filterContext.RouteData.DataTokens["area"].ToString().ToLower();

16

17 //3.获取控制器作用的Controller和action的名字

18 string controllerName = filterContext.ActionDescriptor.ControllerDescriptor.ControllerName.ToLower();

19 string actionName = filterContext.ActionDescriptor.ActionName.ToLower();

20 filterContext.HttpContext.Response.Write("身份验证过滤器作用于" + controllerName + "控制器下的" + actionName + "方法</br>");

21 }

22 }

[复制代码](javascript:void(0);)

View Code

**B:   行为过滤器**

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttps://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 /// <summary>

2 /// 行为过滤器

3 /// 1. 在非MVC框架项目中使用MVC过滤器，需要通过nuget把MVC的程序集添加进去

4 /// 2. 继承ActionFilterAttribute类，然后对OnActionExecuting方法和OnActionExecuted方法进行 override 覆写

5 /// 3. OnActionExecuting方法：在action方法运行之前，且OnAuthorization过滤器运行之后调用

6 /// OnActionExecuted方法：在action方法运行之后调用

7 /// </summary>

8 public class MyAction: ActionFilterAttribute

9 {

10

11 /// <summary>

12 /// 在action方法运行之前调用

13 /// </summary>

14 /// <param name="filterContext"></param>

15 public override void OnActionExecuting(ActionExecutingContext filterContext)

16 {

17 //1.如果保留如下代码，则会运行.net framework定义好的行为验证，如果希望自定义行为验证，则删除如下代码

18 // base.OnActionExecuting(filterContext);

19

20 //2.获取区域名字

21 // string strAreaName = filterContext.RouteData.DataTokens["area"].ToString().ToLower();

22

23 //3.获取控制器作用的Controller和action的名字

24 string controllerName = filterContext.ActionDescriptor.ControllerDescriptor.ControllerName.ToLower();

25 string actionName = filterContext.ActionDescriptor.ActionName.ToLower();

26 filterContext.HttpContext.Response.Write("行为过滤器OnActionExecuting作用于" + controllerName + "控制器下的" + actionName + "方法运行之前</br>");

27 }

28 /// <summary>

29 /// 在action方法运行之后调用

30 /// </summary>

31 /// <param name="filterContext"></param>

32 public override void OnActionExecuted(ActionExecutedContext filterContext)

33 {

34 //1.如果保留如下代码，则会运行.net framework定义好的行为验证，如果希望自定义行为验证，则删除如下代码

35 // base.OnActionExecuted(filterContext);

36

37 //2.获取区域名字

38 // string strAreaName = filterContext.RouteData.DataTokens["area"].ToString().ToLower();

39

40 //3.获取控制器作用的Controller和action的名字

41 string controllerName = filterContext.ActionDescriptor.ControllerDescriptor.ControllerName.ToLower();

42 string actionName = filterContext.ActionDescriptor.ActionName.ToLower();

43 filterContext.HttpContext.Response.Write("行为过滤器OnActionExecuted作用于" + controllerName + "控制器下的" + actionName + "方法运行之后</br>");

44 }

45 }

[复制代码](javascript:void(0);)

View Code

**C：结果过滤器**

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttps://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 /// <summary>

2 /// 结果过滤器

3 /// 1. 在非MVC框架项目中使用MVC过滤器，需要通过nuget把MVC的程序集添加进去

4 /// 2. 继承ActionFilterAttribute类，然后对OnResultExecuting方法和OnResultExecuted方法进行 override 覆写

5 /// 3. OnResultExecuting方法：在执行结果之后（action之后），页面渲染之前调用

6 /// OnResultExecuted方法：在页面渲染之后调用

7 /// </summary>

8 public class MyResult : ActionFilterAttribute

9 {

10

11 /// <summary>

12 /// action执行之后（OnActionExecuting之后），页面渲染之前调用

13 /// </summary>

14 /// <param name="filterContext"></param>

15 public override void OnResultExecuting(ResultExecutingContext filterContext)

16 {

17 //1.如果保留如下代码，则会运行.net framework定义好的结果验证，如果希望自定义结果验证，则删除如下代码

18 // base.OnResultExecuting(filterContext);

19

20 //该方法中无法获取是哪个控制器后

21 filterContext.HttpContext.Response.Write("结果过滤器OnResultExecuting作用于action运行之后，页面加载之前");

22 }

23 /// <summary>

24 /// 页面渲染之后调用

25 /// </summary>

26 /// <param name="filterContext"></param>

27 public override void OnResultExecuted(ResultExecutedContext filterContext)

28 {

29 //1.如果保留如下代码，则会运行.net framework定义好的结果验证，如果希望自定义结果验证，则删除如下代码

30 // base.OnResultExecuted(filterContext);

31

32 //该方法中无法获取是哪个控制器后

33 filterContext.HttpContext.Response.Write("结果过滤器OnResultExecuted作用于页面渲染之后");

34 }

35 }

[复制代码](javascript:void(0);)

View Code

**D：异常过滤器**

**使用自定义异常处理，需要在web.config中为system.web添加<customErrors mode="On" />节点**

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttps://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 /// <summary>

2 /// 异常过滤器

3 /// 需要注意的点：

4 /// ①：如果自定义异常过滤器且需要有作用于全局，需要把FilterConfig中的 filters.Add(new HandleErrorAttribute());注释掉，

5 /// 然后把自定义的异常过滤器添加到FilterConfig中。

6 /// ②：使用自定义异常处理，需要在web.config中为system.web添加<customErrors mode="On" />节点

7 /// </summary>

8 public class MyException: HandleErrorAttribute

9 {

10 public override void OnException(ExceptionContext filterContext)

11 {

12 //调用框架本身异常处理器的方法

13 base.OnException(filterContext);

14

15 //获取异常信息(可以根据实际需要写到本地或数据库中)

16 var errorMsg = filterContext.Exception;

17

18 //跳转指定的错误页面

19 filterContext.Result = new RedirectResult("/error.html");

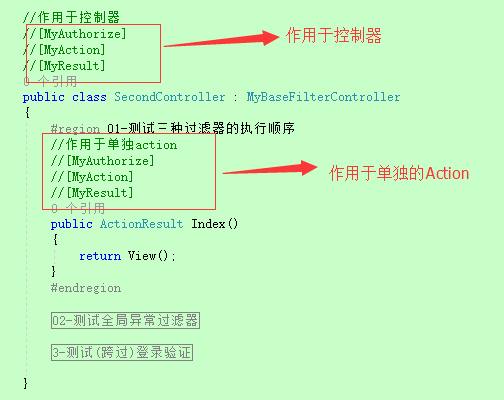
20 }

21 }

[复制代码](javascript:void(0);)

View Code

**下面展示以特性的形式作用于控制器或控制器中的Action：**

****

3. 自定义类继承MVC中实现类或接口，全局注册，作用于全部控制器

 　　如果以上两种方式均不能满足你的过滤器的使用范围，你可以在App\_Start文件夹下的FilterConfig类中进行全局注册，使该过滤器作用于所有控制器中所有Action方法。

　  特别注意的一点是：自定义异常过滤器，需要把系统默认的filters.Add(new HandleErrorAttribute());注释掉。

**全局注册的代码如下：**

[复制代码](javascript:void(0);)

1 public class FilterConfig

2 {

3 public static void RegisterGlobalFilters(GlobalFilterCollection filters)

4 {

5 //如果自定义异常过滤器，需要把默认的异常过滤器给注释掉

6 //filters.Add(new HandleErrorAttribute());

7

8 //自定义异常过滤器

9 filters.Add(new MyException());

10

11 //全局注册身份验证、行为、结果过滤器

12 //filters.Add(new MyAuthorize());

13 //filters.Add(new MyAction());

14 //filters.Add(new MyResult());

15

16 //全局注册登录验证（暂时注释，使用的时候要打开）

17 //filters.Add(new CheckLogin());

18 }

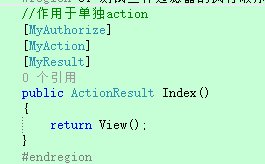
19 }

[复制代码](javascript:void(0);)

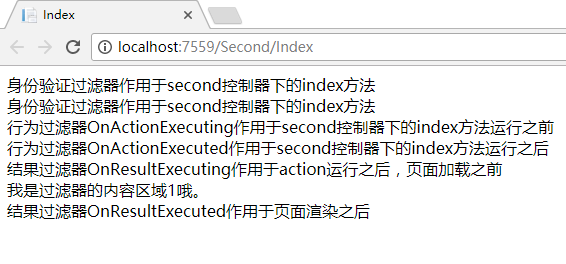
四. 结合实际案例进行代码测试

1. 测试过滤器的执行顺序

 　　将上面的身份验证过滤器、行为过滤器、结果过滤器以特性的形式作用于Action上，通过断点监控或者查看最后的输出结果：



**结果：**



**符合：OnAuthorization→OnActionExecuting-> Action方法执行 ->OnActionExecuted->OnResultExecuting/ -> Render View() (页面渲染加载)->OnResultExecuted() 这一顺序。**

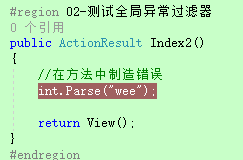
2. 全局捕获异常，记录错误日志案例

步骤1：编写异常过滤器，通过 var errorMsg = filterContext.Exception; 获取异常信息，可以写入文本、存入数据库、或者是Log4Net错误日志框架进行处理。代码在上面。

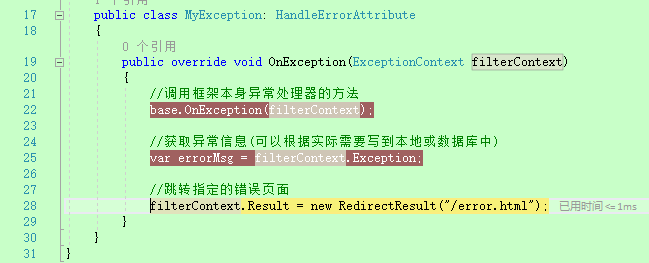
步骤2：在web.config中为system.web添加<customErrors mode="On" />节点。

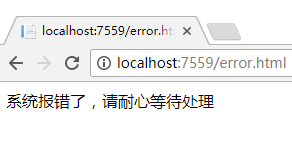
步骤3：添加到全局注册文件中进行捕获。

步骤4：在自定义的异常过滤器中添加断点，并且自己制造一个错误。



　　捕获到错误，进行页面跳转。





3. 登录验证案例

 　　业务背景：在90%以上的Web系统中，很多页面都是登录成功以后才能看到的，当然也有很多页面不需要登录，对于需要登录才能看到的页面，即使你知道了访问地址，也是不能访问的，会退出到登录页面提示让你登录，对于不需要登录的页面通过URL地址可以直接访问。

**分析：针对以上背景，过滤器对于大部分Action是需要过滤的，需要做登录验证，对于一小部分是不需要过滤的。**

**解决思路：**

**①：自定义一个身份验证过滤器，进行全局注册。**

**②：自定义一个Skip特性，将该特性加到不需要身份验证的Action上。**

**③：重点说一下身份证过滤器中的逻辑：**

**a. 先判断该Action上是否又Skip特性，如果有，停止不继续执行；如果没有，继续下面的验证逻辑。**

**b. 从Session中或Redis中读取当前用户，查看是否为空，如果为空，表明没有登录，返回到登录页面；如果不为空，验证通过，进行后面的业务逻辑。**

**代码段如下：**

[复制代码](javascript:void(0);)

1 /// <summary>

2 /// 校验系统是否登录的过滤器

3 /// 使用身份验证过滤器进行编写

4 /// </summary>

5 public class CheckLogin: AuthorizeAttribute

6 {

7 public override void OnAuthorization(AuthorizationContext filterContext)

8 {

9 //1. 校验是否标记跨过登录验证

10 if (filterContext.ActionDescriptor.IsDefined(typeof(skipAttribute), true)

11 || filterContext.ActionDescriptor.ControllerDescriptor.IsDefined(typeof(skipAttribute), true))

12 {

13 //表示该方法或控制器跨过登录验证

14 return;

15 }

16 //2. 校验是否登录

17 //可以使Session或数据库或nosql

18 //这里只是测试，所有统统当做没有登录来处理

19 var sessionUser = HttpContext.Current.Session["CurrentUser"];//使用session

20 if (sessionUser == null)

21 {

22 HttpContext.Current.Session["CurrentUrl"] = filterContext.RequestContext.HttpContext.Request.RawUrl;

23 //如果没有登录，则跳转到错误页面

24 filterContext.Result = new RedirectResult("/error.html");

25 }

26 }

27 }

[复制代码](javascript:void(0);)