

WebApi整合Autofac

实例代码路径 [.net笔记\Code\WebApi4Autofac\WebApi4Autofac](#)

以下例子使用的.NET SDK版本为6.0.100,开发工具使用Microsoft Visual Studio Enterprise 2022 (64位) 版本 17.0.1

Autofac版本为6.3.0

安装Autofac包

```
Install-Package Autofac -Version 6.3.0
```

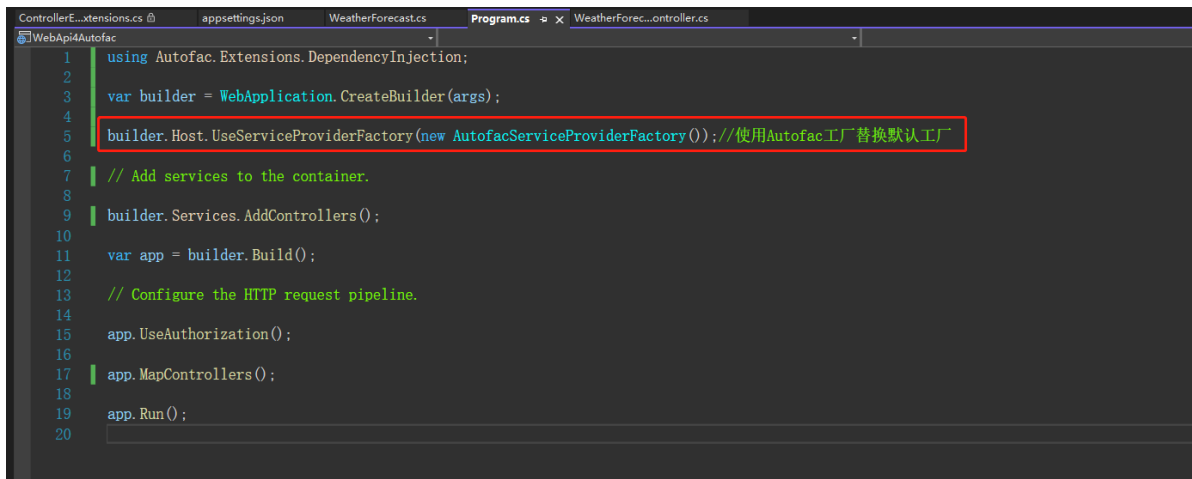
Autofac工厂替换默认工厂

安装Autofac.Extensions.DependencyInjection包

```
Install-Package Autofac.Extensions.DependencyInjection -Version 7.2.0
```

- 引用命名空间using Autofac.Extensions.DependencyInjection;

```
builder.Host.UseServiceProviderFactory(new AutofacServiceProviderFactory());//使用Autofac工厂替换默认工厂
```



```
1 using Autofac.Extensions.DependencyInjection;
2
3 var builder = WebApplication.CreateBuilder(args);
4 builder.Host.UseServiceProviderFactory(new AutofacServiceProviderFactory());//使用Autofac工厂替换默认工厂
5
6 // Add services to the container.
7
8 builder.Services.AddControllers();
9
10 var app = builder.Build();
11
12 // Configure the HTTP request pipeline.
13 app.UseAuthorization();
14
15 app.MapControllers();
16
17 app.Run();
18
19
20
```

- 引用命名空间using Autofac;
- 调用ConfigureContainer注册服务

```
builder.Host.ConfigureContainer<ContainerBuilder>((HostBuilderContext,
ContainerBuilder) => {
    ContainerBuilder.RegisterType<TestServiceAimpl>().As<ITestServiceA>();
});//调用ConfigureContainer注册服务
```

```
Program.cs
WebApi4Autofac

1 using Autofac;
2 using Autofac.Extensions.DependencyInjection;
3 using WebApi4Autofac.BLL;
4 using WebApi4Autofac.IBLL;
5
6 var builder = WebApplication.CreateBuilder(args);
7
8 builder.Host.UseServiceProviderFactory(new AutofacServiceProviderFactory()); //使用Autofac工厂替换默认工厂
9
10
11 builder.Host.ConfigureContainer<ContainerBuilder>((HostBuilderContext, ContainerBuilder) => {
12     ContainerBuilder.RegisterType<TestServiceAImpl>().As<ITestServiceA>();
13 }); //调用ConfigureContainer注册服务
14
15 // Add services to the container.
16
17 builder.Services.AddControllers();
18
19 var app = builder.Build();
20
21 // Configure the HTTP request pipeline.
22
23 app.UseAuthorization();
24
25 app.MapControllers();
26
27 app.Run();
28
```

在通过控制器的构造函数注入服务

```
private readonly ITestServiceA _testServiceA;

public WeatherForecastController(ILogger<WeatherForecastController> logger,
ITestServiceA testServiceA)
{
    _logger = logger;
    _testServiceA = testServiceA;
}
```

```
WeatherForecastController.cs
Program.cs
WebApi4Autofac
WebApi4Autofac.Controllers.WeatherForecastController
WeatherForecastController(ILogger<WeatherForecastController>)

1 using Microsoft.AspNetCore.Mvc;
2 using WebApi4Autofac.IBLL;
3
4 namespace WebApi4Autofac.Controllers
5 {
6     [ApiController]
7     [Route("[controller]")]
8     public class WeatherForecastController : ControllerBase
9     {
10         private static readonly string[] Summaries = new[]
11         {
12             "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm", "Balmy", "Hot", "Sweltering", "Scorching"
13         };
14
15         private readonly ILogger<WeatherForecastController> _logger;
16
17         private readonly ITestServiceA _testServiceA;
18
19         public WeatherForecastController(ILogger<WeatherForecastController> logger, ITestServiceA testServiceA)
20         {
21             _logger = logger;
22             _testServiceA = testServiceA;
23         }
24
25         [HttpGet]
26     }
```

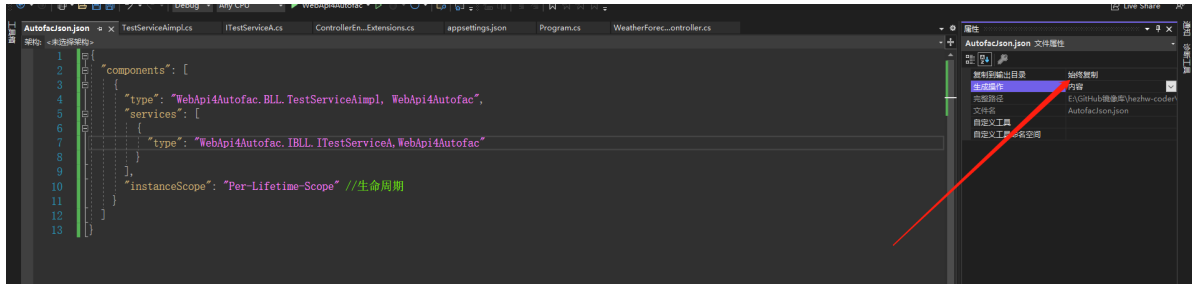
使用Autofac配置文件

nuget安装Autofac.Configuration配置扩展包(例子使用6.0.0版本)

```
Install-Package Autofac.Configuration -Version 6.0.0
```

创建 AutofacJson.json 文件配置

```
{
  "components": [
    {
      "type": "WebApi4Autofac.BLL.TestServiceAimpl, WebApi4Autofac",
      "services": [
        {
          "type": "WebApi4Autofac.IBLL.ITestServiceA, WebApi4Autofac"
        }
      ]
    },
    {
      "instanceScope": "Per-Lifetime-Scope" //生命周期
    }
  ]
}
```

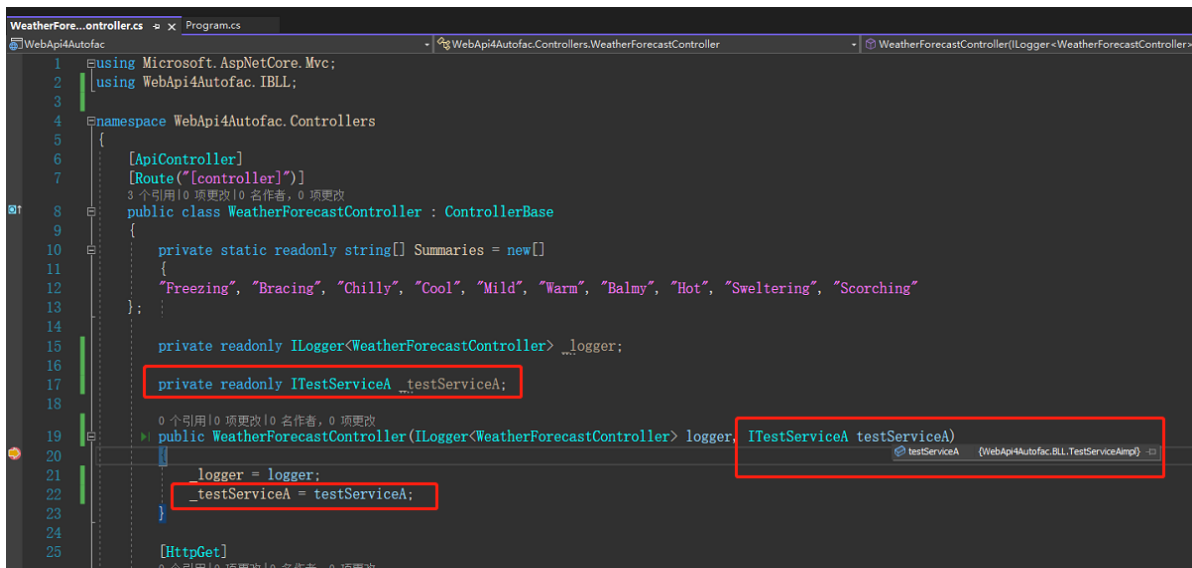


- 调用ConfigureContainer注册服务

```
builder.Host.ConfigureContainer<ContainerBuilder>((HostBuilderContext,
ContainerBuilder) => {
    // 实例化ConfigurationBuilder.
    var config = new Microsoft.Extensions.Configuration.ConfigurationBuilder();
    //使用Microsoft.Extensions.Configuration.Json读取json配置文件
    config.AddJsonFile("Conf/AutofacJson.json");

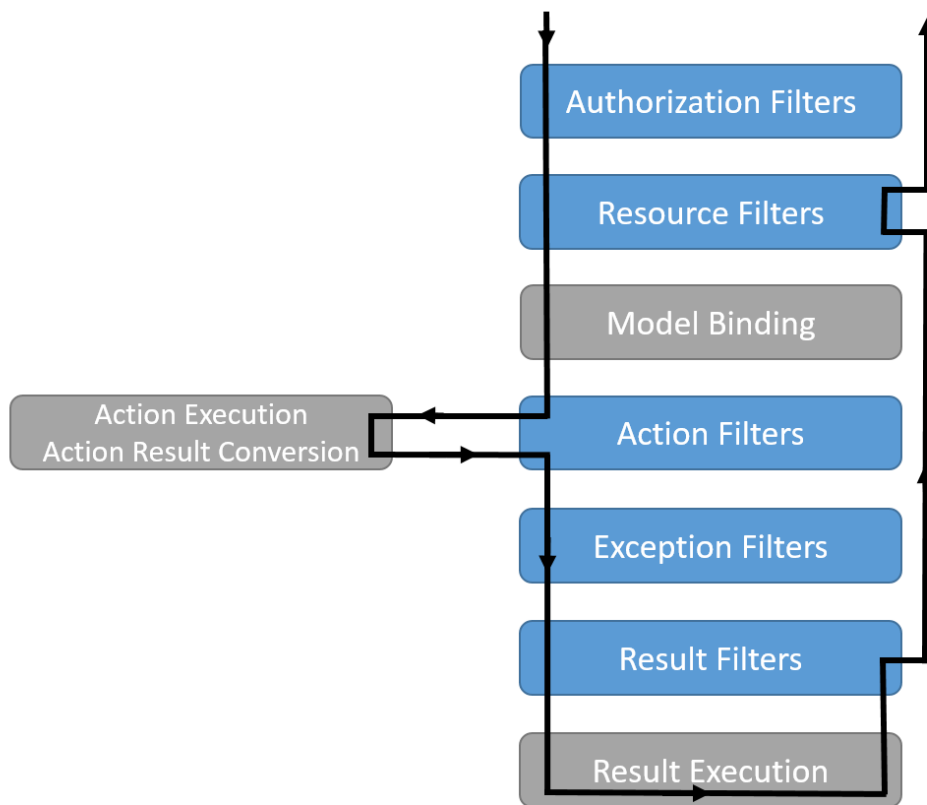
    // Register the ConfigurationModule with Autofac.
    var module = new
Autofac.Configuration.ConfigurationModule(config.Build()); //将配置文件加载至module
    ContainerBuilder.RegisterModule(module);
}); //调用ConfigureContainer注册服务
```

- 在控制器的构造函数进行注入



WebApi中的过滤器(Filters)

过滤器管道



ActionFilter

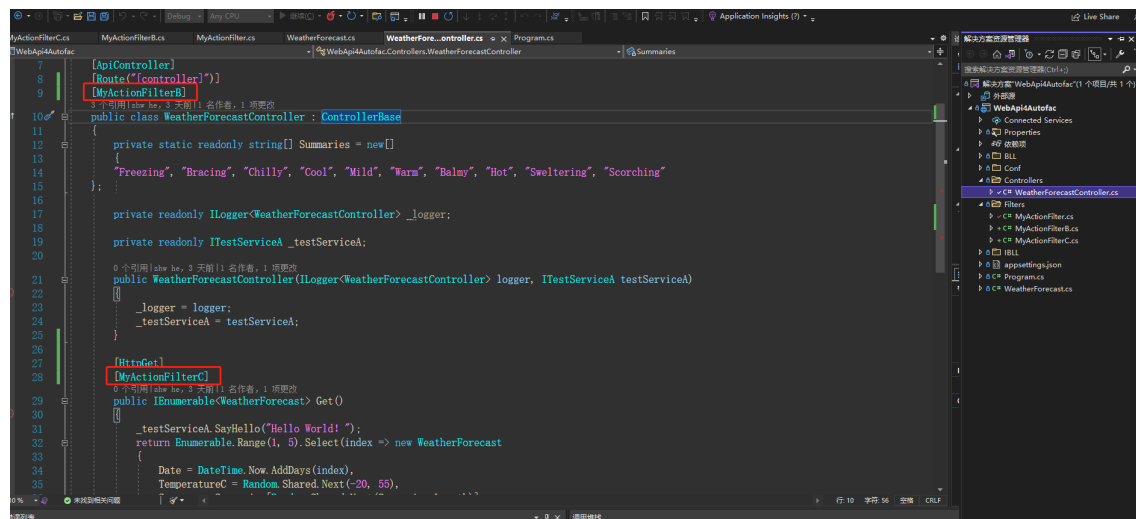
- 新建一个类继承ActionFilterAttribute,并重写里面的方法(建议重写异步的方法, 例子中使用的是同步方法)

```
MyActionFilter.cs
WebApi4Autofac
WebApi4Autofac.Filters.MyActionFilter
OnActionExecuting(ActionExecutingContext context)

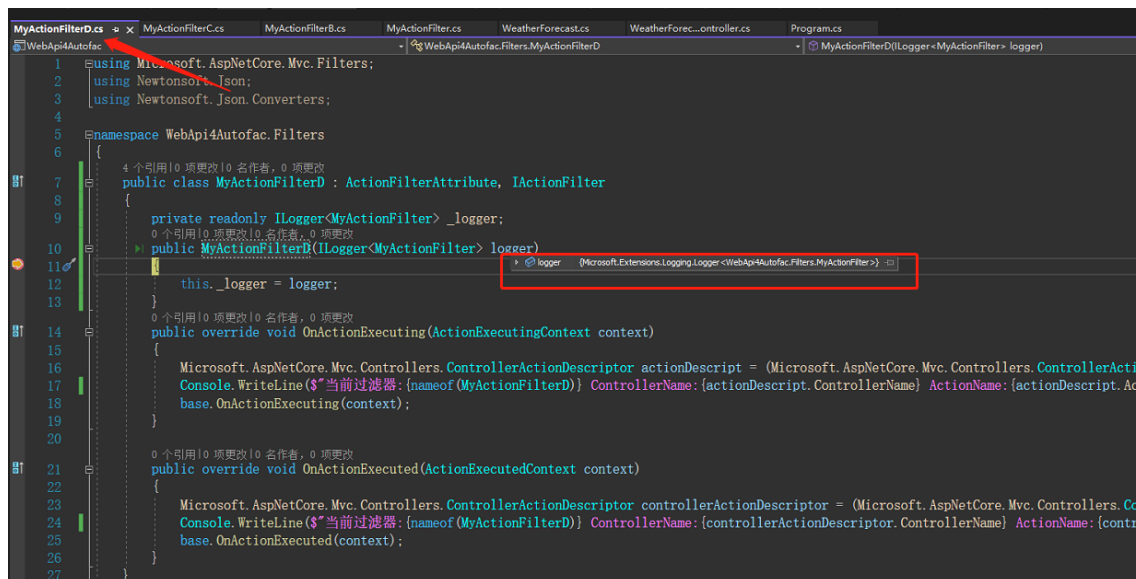
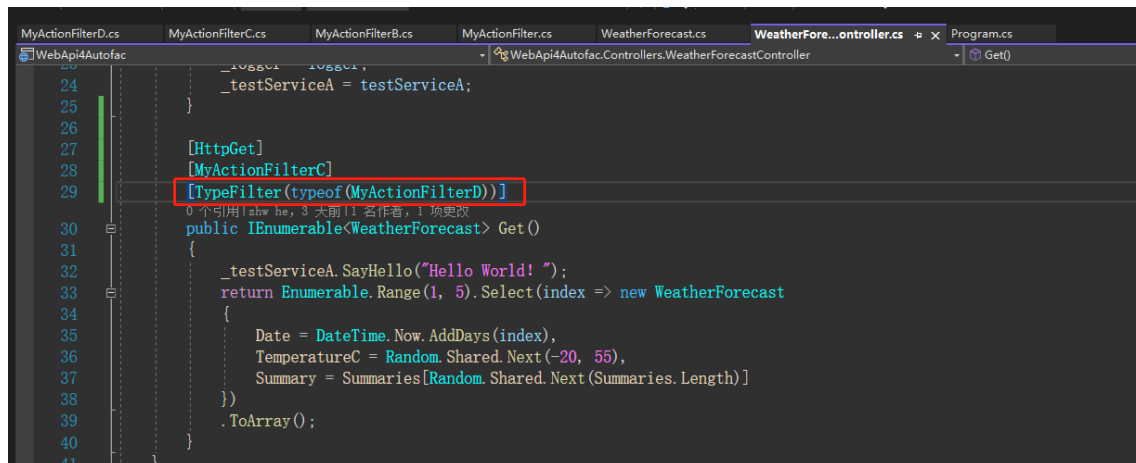
1 using Microsoft.AspNetCore.Mvc.Filters;
2 using Newtonsoft.Json;
3 using Newtonsoft.Json.Converters;
4
5 namespace WebApi4Autofac.Filters
6 {
7     public class MyActionFilter: ActionFilterAttribute, IActionFilter
8     {
9         private readonly ILogger<MyActionFilter> _logger;
10        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
11        public MyActionFilter(ILogger<MyActionFilter> logger)
12        {
13            this._logger = logger;
14        }
15
16        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
17        public override void OnActionExecuting(ActionExecutingContext context)
18        {
19            base.OnActionExecuting(context);
20        }
21
22        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
23        public override void OnActionExecuted(ActionExecutedContext context)
24        {
25            base.OnActionExecuted(context);
26        }
27    }
28 }
```

ActionFilter的多种配置方式及特点

- 在ControllerBa或者Action上标记特性，但是定义的动作Filter必须有且只有一个无参的构造函数,并且不支持依赖注入



- 使用TypeFilter在ControllerBa或者Action上标记，定义的动作Filter可以没有无参的构造函数,支持依赖注入



- 定义特性实现IFilterFactory接口，可以没有无参数构造函数，可以支持依赖注入。但是要使用这种方式，要在【ConfigureContainer】方法中将过滤器注册到服务

```
MyFilterFactory.cs | MyActionFilterD.cs | MyActionFilterC.cs | MyActionFilterB.cs | MyActionFilter.cs | WeatherForecast.cs | WeatherFore...ontroller.cs | Program.cs
WebApi4Autofac
1 using Microsoft.AspNetCore.Mvc.Filters;
2
3 namespace WebApi4Autofac.Filters
4 {
5     public class MyFilterFactory : Attribute, IFilterFactory
6     {
7         private readonly Type _type;
8         /// <summary>
9         /// 构造传入的过滤器Type
10        /// </summary>
11        /// <param name="type"></param>
12        0 个引用 | 0 项更改 | 10 名作者, 0 项更改
13        public MyFilterFactory(Type type)
14        {
15            this._type = type;
16        }
17        /// <summary>
18        /// 是否可重用
19        /// </summary>
20        0 个引用 | 0 项更改 | 10 名作者, 0 项更改
21        public bool IsReusable => true;
22
23        0 个引用 | 0 项更改 | 10 名作者, 0 项更改
24        public IFilterMetadata CreateInstance(IServiceProvider serviceProvider)
25        {
26            object? oInstance = serviceProvider.GetService(_type);
27            return (IFilterMetadata)oInstance;
28        }
29    }
30 }
```

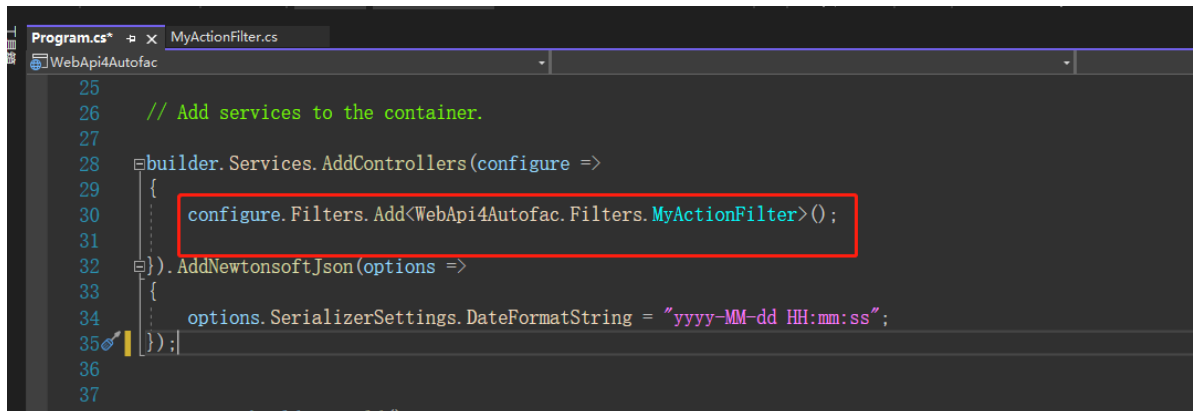
```
Program.cs
WebApi4Autofac
4 using WebApi4Autofac.IBLL;
5
6 var builder = WebApplication.CreateBuilder(args);
7
8 builder.Host.UseServiceProviderFactory(new AutofacServiceProviderFactory()); //使用Autofac工厂替换默认工厂
9
10 //builder.Host.ConfigureContainer<ContainerBuilder>((HostBuilderContext, ContainerBuilder) => {
11 //    ContainerBuilder.RegisterType<TestServiceAImpl>().As<ITestServiceA>();
12 //}); //调用ConfigureContainer注册服务
13
14 builder.Host.ConfigureContainer<ContainerBuilder>((HostBuilderContext, ContainerBuilder) => {
15     // 实例化ConfigurationBuilder.
16     var config = new Microsoft.Extensions.Configuration.ConfigurationBuilder();
17     //使用Microsoft.Extensions.Configuration.Json读取json配置文件
18     config.AddJsonFile("Conf/AutofacJson.json");
19
20     // Register the ConfigurationModule with Autofac.
21     var module = new Autofac.Configuration.ConfigurationModule(config.Build()); //将配置文件加载至module
22     ContainerBuilder.RegisterModule(module);
23     ContainerBuilder.RegisterType(typeof(WebApi4Autofac.Filters.MyActionFilterD));
24 }); //调用ConfigureContainer注册服务
25
26 // Add services to the container.
27
28
29 builder.Services.AddControllers(configure =>
30 {
31     configure.Filters.Add<WebApi4Autofac.Filters.MyActionFilter>();
32 })
```

```
WeatherFore...ontroller.cs | Program.cs
WebApi4Autofac
15 }
16
17 private readonly ILogger<WeatherForecastController> _logger;
18
19 private readonly ITestServiceA _testServiceA;
20
21 0 个引用 | chv he, 3 天前 | 1 名作者, 1 项更改
22 public WeatherForecastController(ILogger<WeatherForecastController> logger, ITestServiceA testServiceA)
23 {
24     _logger = logger;
25     _testServiceA = testServiceA;
26 }
27
28 [HttpGet]
29 [MyActionFilterC]
30 [TypeFilter(typeof(MyActionFilterD))]
31 [MyFilterFactory(typeof(MyActionFilterD))]
32 0 个引用 | chv he, 3 天前 | 1 名作者, 1 项更改
33 public IEnumerable<WeatherForecast> Get()
34 {
35     _testServiceA.SayHello("Hello World! ");
36     return Enumerable.Range(1, 5).Select(index => new WeatherForecast
37     {
38         Date = DateTime.Now.AddDays(index),
39         TemperatureC = Random.Shared.Next(-20, 55),
40         Summary = Summaries[Random.Shared.Next(Summaries.Length)]
41     })
42     .ToArray();
43 }
```

全局ActionFilter

在添加控制器的中间件注册全局过滤器

```
builder.Services.AddControllers(configure =>
{
    configure.Filters.Add<WebApi4Autofac.Filters.MyActionFilter>();
}).AddNewtonsoftJson(options =>
{
    options.SerializerSettings.DateFormatString = "yyyy-MM-dd HH:mm:ss";
});
```



Filter生效范围和控制执行顺序

- 标记在Action上，就只对当前Action生效。
- 标记在Controller上，就对改Controller上所有的Action生效。
- 全局注册，对于当前整个项目中的Action都生效

如果有三个ActionFilter，分别注册全局、控制器、Action,则执行顺序(类似中间件)如下

先执行全局的OnActionExecuting—>控制器的OnActionExecuting—>Action上标记的OnActionExecuting—>Action上标记的OnActionExecuted—>控制器的OnActionExecuted—>全局的OnActionExecuted

详细可参考[ASP.NET Core 中的筛选器 | Microsoft Docs](#)

过滤器设置短路

可以在

过滤器的Executing方法里给context的Result 属性赋值，这样Executing方法执行完后就不会往下执行其他过滤器，因为ResourceFilter 在管道中比较靠前，一般使用它来设置短路,详细可见[ASP.NET Core 中的筛选器 | Microsoft Docs](#)

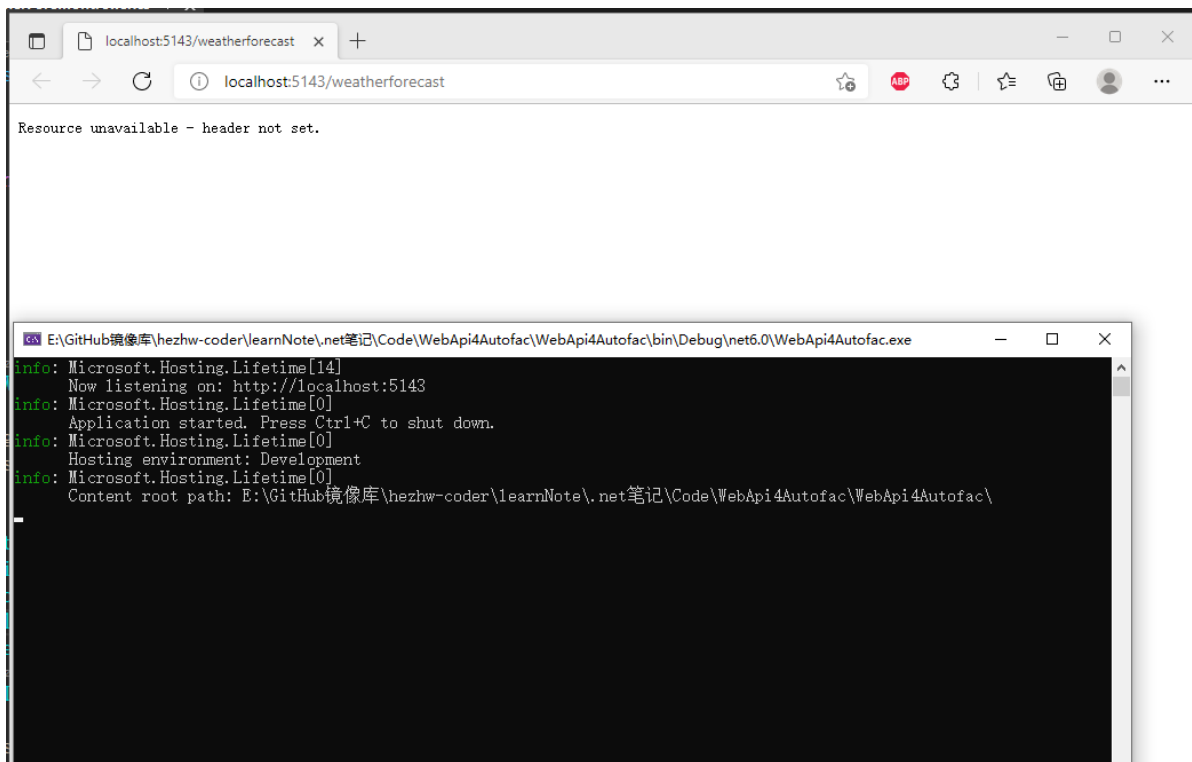
```
ShortCircuitingResourceFilterAttribute.cs | WeatherForecastController.cs
WebApi4Autofac | WebApi4Autofac.Filters.ShortCircuitingResourceFilterAttribute | OnResourceExecuting(ResourceExecutingContext context)

1 using Microsoft.AspNetCore.Mvc.Filters;
2
3 namespace WebApi4Autofac.Filters
4 {
5     public class ShortCircuitingResourceFilterAttribute : Attribute, IResourceFilter
6     {
7         public void OnResourceExecuting(ResourceExecutingContext context)
8         {
9             context.Result = new Microsoft.AspNetCore.Mvc.ContentResult()
10             {
11                 Content = "Resource unavailable - header not set."
12             };
13         }
14
15         public void OnResourceExecuted(ResourceExecutedContext context)
16         {
17         }
18     }
19 }
20
```

```
ShortCircuitingResourceFilterAttribute.cs | WeatherForecastController.cs
WebApi4Autofac | WebApi4Autofac.Controllers.WeatherForecastController | Get()

10 public class WeatherForecastController : ControllerBase
11 {
12     private static readonly string[] Summaries = new[]
13     {
14         "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm", "Balmy", "Hot", "Sweltering", "Scorching"
15     };
16
17     private readonly ILogger<WeatherForecastController> _logger;
18
19     private readonly ITestServiceA _testServiceA;
20
21     public WeatherForecastController(ILogger<WeatherForecastController> logger, ITestServiceA testServiceA)
22     {
23         _logger = logger;
24         _testServiceA = testServiceA;
25     }
26
27     [HttpGet]
28     [ShortCircuitingResourceFilter]
29     [MyActionFilterC(Order = 1)]
30     [TypeFilter(typeof(MyActionFilterD))]
31     [MyFilterFactory(typeof(MyActionFilterD))]
32     public IEnumerable<WeatherForecast> Get()
33     {
34         _testServiceA.SayHello("Hello World! ");
35         return Enumerable.Range(1, 5).Select(index => new WeatherForecast
36         {
37             Date = DateTime.Now.AddDays(index),
38             TemperatureC = Random.Shared.Next(-20, 55),
39             Summary = Summaries[Random.Shared.Next(Summaries.Length)]
40         })
41         .ToArray();
42     }
43 }
```

由下图执行结果可见,并不会执行全局的Action过滤器及标记的过滤器的内容



ResourceFilter

用途

- 因为在管道中仅位于授权过滤器之后，可用于使大部分管道短路
- 用于做缓存

官网介绍可见[ASP.NET Core 中的筛选器 | Microsoft Docs](#)

ExceptionHandler

- 控制器实例化异常——能捕捉
- 异常发生在Try-cache中——不能捕捉，因为异常已经被捕捉到了
- 在视图中发生异常——不能捕捉
- 在Service层（业务逻辑层）中发生异常——能捕捉
- 在Action中发生异常——能捕捉
- 请求错误路径异常——不能捕捉（但是可以通过中间件解决）

若要处理异常，请将 [ExceptionHandler](#) 属性设置为 `true`，或编写响应。这将停止传播异常。异常筛选器无法将异常转变为“成功”。只有操作筛选器才能执行该转变

建议使用中间件处理异常。基于所调用的操作方法，仅当错误处理不同时，才使用ExceptionHandler。例如，应用可能具有用于 API 终结点和视图/HTML 的操作方法。API 终结点可能返回 JSON 形式的错误信息，而基于视图的操作可能返回 HTML 形式的错误页。

官方文档详见[ASP.NET Core 中的筛选器 | Microsoft Docs](#)