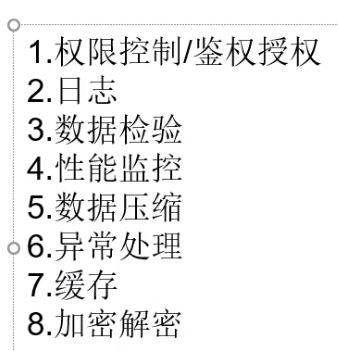
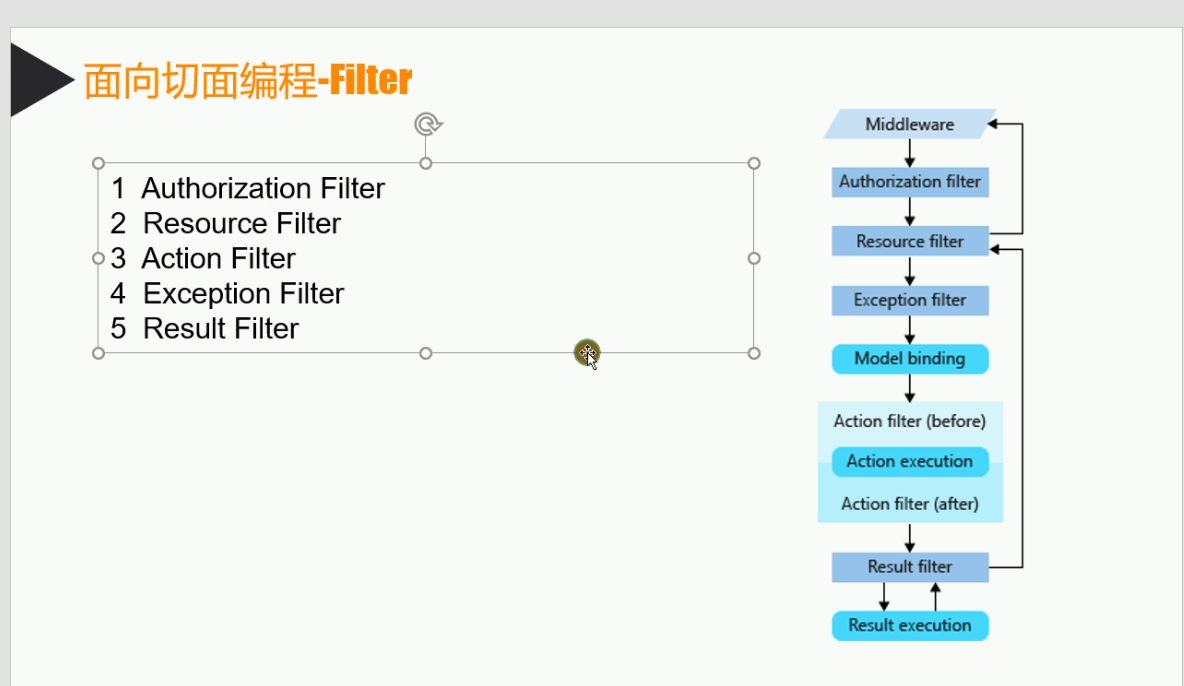
**Aop：面向切面编程，类似Fillter,过滤器，场景：权限， 异常处理，日志，缓存，事务等等**



1. 能在不破坏封装的前提下，额外扩展功能
2. 代码复用，集中管理。

生命周期



Filter执行顺序：

1. IResourceFilter.OnResourceExecuting（发生在控制器实例化之前）
2. IActionFilter. OnActionExecuting
3. IActionFilter. OnActionExecuted
4. IResultFilter. OnResultExecuting
5. IResultFilter. OnResultExecuted 视图替换环节
6. IResourceFilter. OnResultExecuted（发生在最后）

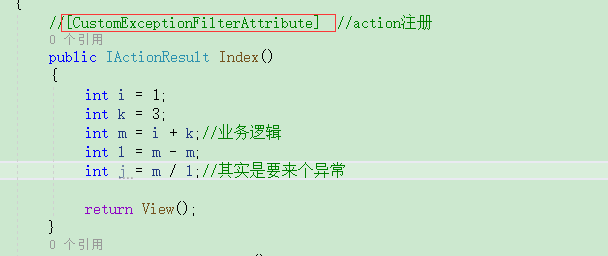
每一个环节都可以做缓存，只是效果不一样

1. 基于IresourceFilter的缓存，避免控制器创建和action执行，但是视图重新执行了，以action为单位执行，没法深入到业务逻辑层
2. [ResponseCache(Duration =60)]//直接缓存在客户端的硬盘上，不走sdn，反向代理和服务器了(在请求响应式，添加了一个responseheader),来设置浏览器缓存结果,
3. 等同于base.HttpContext.Response.Headers["Cache-Control"] = "public,max-age=60";

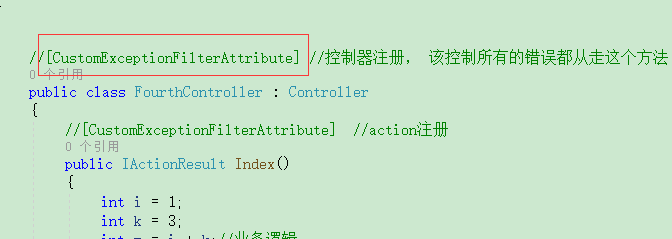
public IActionResult Index(){}

**Fillter 三种注册方式(不过404抓不到，因为没进去MVC流程)包含4中注入方式(控制器注入 1全局注册 2** ServiceFilter（需要在 startup中注册） **3 TypeFilter 4 IFilterFactory 2,3,4都是基于 IFilterFactory的，3 可以传参数 4可以做扩展 )：**

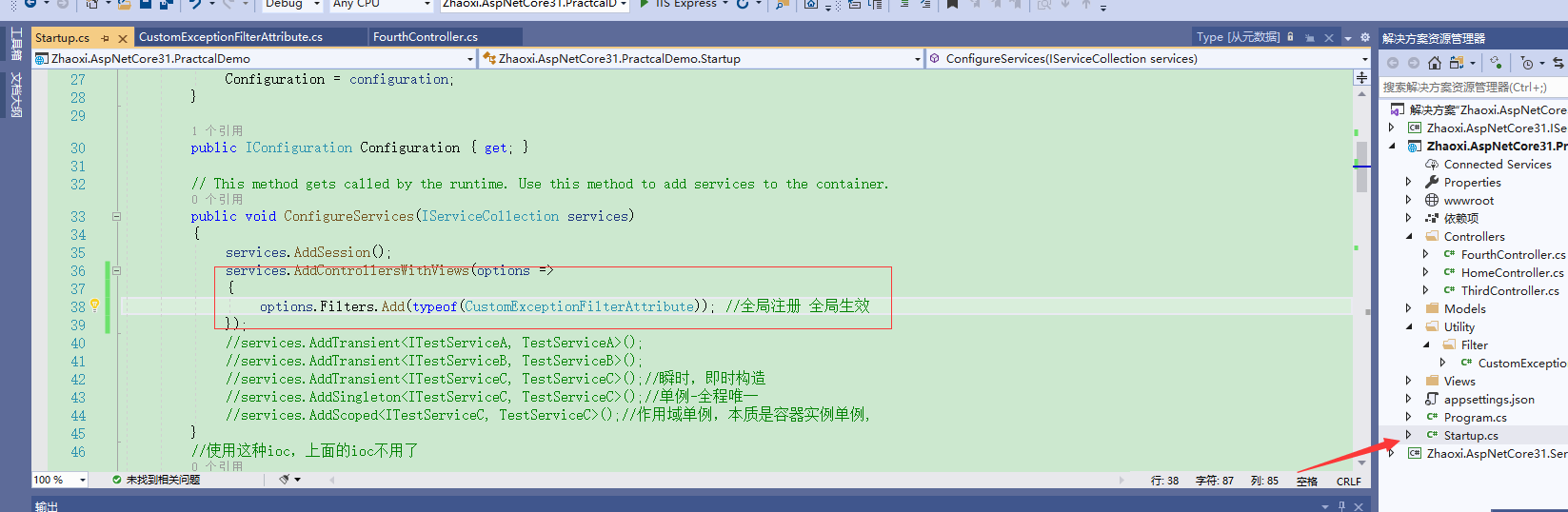
1 action注册（就是放在action的特性）



2. 控制器注册， 该控制所有的错误都从走这个方法



1. 全局注册，全局生效



//公共异常方法，重写ExceptionFilterAttribute的OnException方法

public class CustomExceptionFilterAttribute : ExceptionFilterAttribute

{

/// <summary>

/// 当异常发生时 会进来

/// </summary>

/// <param name="context"></param>

public override void OnException(ExceptionContext context)

{

if (!context.ExceptionHandled)

{

//当异常没有被处理时

Console.WriteLine($"{context.HttpContext.Request.Path} {context.Exception.Message}");

context.Result = new JsonResult(new

{

Result = false,

Msg = "发生异常，请联系管理员"

});

context.ExceptionHandled = true;

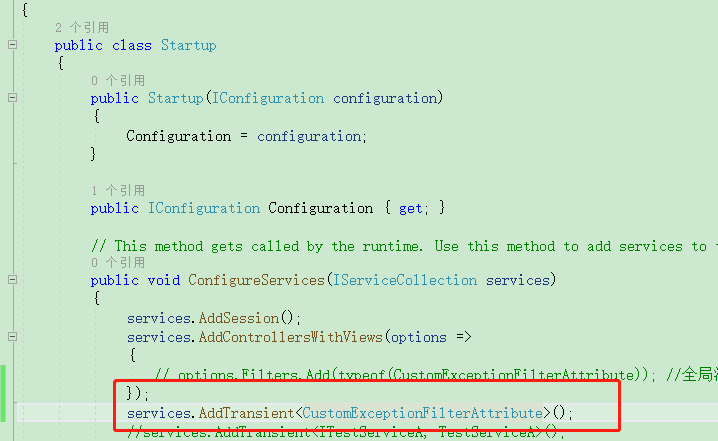
}

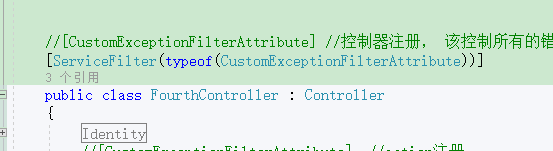
}

}

1. 控制器注册第2种写法

先在startup中注册，然后在控制器添加特性 [ServiceFilter(typeof(XXX))]

意思是XXX实例化



1. 控制器注册第3种写法

[TypeFilter(typeof(CustomExceptionFilterAttribute))]

public class FourthController : Controller

{

}

1. 控制器注册第4种写法

先在startup中注册，然后在控制器添加特性

[CustomFilterFactoryAttribute(typeof(CustomExceptionFilterAttribute))]

public class FourthController : Controller

{

}

public class CustomFilterFactoryAttribute :Attribute, IFilterFactory

{

private Type \_FilterType = null;

public CustomFilterFactoryAttribute(Type type)

{

this.\_FilterType = type;

}

public bool IsReusable => true;

/// <summary>

/// IServiceProvider 就是容器

/// 通过容器可以构造对象进行依赖注入

/// </summary>

/// <param name="serviceProvider"></param>

/// <returns></returns>

public IFilterMetadata CreateInstance(IServiceProvider serviceProvider)

{

return (IFilterMetadata)serviceProvider.GetService(this.\_FilterType);

}

}

