### **筛选器类型**

每种筛选器类型都在筛选器管道中的不同阶段执行。

[授权筛选器](https://docs.microsoft.com/zh-cn/aspnet/core/mvc/controllers/filters?view=aspnetcore-2.1" \l "authorization-filters)：最先运行，用于确定是否已针对当前请求为当前用户授权。 如果请求未获授权，它们可以让管道短路。

[资源筛选器](https://docs.microsoft.com/zh-cn/aspnet/core/mvc/controllers/filters?view=aspnetcore-2.1" \l "resource-filters)：是授权后最先处理请求的筛选器。 出于性能方面的考虑，可以使用它们来实现缓存或以其他方式让筛选器管道短路。 它们在模型绑定之前运行，所以可以影响模型绑定。

[操作筛选器](https://docs.microsoft.com/zh-cn/aspnet/core/mvc/controllers/filters?view=aspnetcore-2.1" \l "action-filters)：可以在调用单个操作方法之前和之后立即运行代码。 它们可用于处理传入某个操作的参数以及从该操作返回的结果。

[异常筛选器](https://docs.microsoft.com/zh-cn/aspnet/core/mvc/controllers/filters?view=aspnetcore-2.1" \l "exception-filters)：用于在向响应正文写入任何内容之前，对未经处理的异常应用全局策略。

[结果筛选器](https://docs.microsoft.com/zh-cn/aspnet/core/mvc/controllers/filters?view=aspnetcore-2.1" \l "result-filters)：可以在执行单个操作结果之前和之后立即运行代码。 仅当操作方法成功执行时，它们才会运行。 对于必须围绕视图或格式化程序的执行的逻辑，它们很有用。

**筛选器添加**

services.AddMvc (options => {

    options.Filters.Add (typeof (MemberIdFilter));

})

## **授权筛选器**

实现IAsyncAuthorizationFilter接口

public class ApiAuthorizationAttribute : Attribute, IAsyncAuthorizationFilter {

    public async Task OnAuthorizationAsync (AuthorizationFilterContext context) {

        ....

        if (!authorizationResult.Succeeded) {

            // 禁止访问

            context.Result = new ForbidResult ();

        }

    }

}

## **资源筛选器**

实现 IResourceFilter 或 IAsyncResourceFilter 接口，

如果需要使某个请求正在执行的大部分工作短路，资源筛选器会很有用。 例如，如果响应在缓存中，则缓存筛选器可以绕开管道的其余阶段。

## **操作筛选器**

实现 IActionFilter 或 IAsyncActionFilter 接口。

下面是一个操作筛选器示例：

public class MemberIdFilter : IAsyncActionFilter {

    private IHttpContextAccessor \_httpContextAccessor { get; set; }

    public MemberIdFilter (IHttpContextAccessor httpContextAccessor) {

        \_httpContextAccessor = httpContextAccessor;

}

    public async Task OnActionExecutionAsync (

        ActionExecutingContext context,

        ActionExecutionDelegate next) {

        // 已认证

        if (\_httpContextAccessor.HttpContext.User != null) {

            .......

        }

        await next ();

    }

}

## **异常筛选器**

异常筛选器可实现 IExceptionFilter 或 IAsyncExceptionFilter 接口。

下面的异常筛选器示例使用自定义开发人员错误视图，显示在开发应用时发生的异常的相关详细信息：

public class CustomExceptionFilterAttribute : ExceptionFilterAttribute

{

    private readonly IHostingEnvironment \_hostingEnvironment;

    private readonly IModelMetadataProvider \_modelMetadataProvider;

    public CustomExceptionFilterAttribute(

        IHostingEnvironment hostingEnvironment,

        IModelMetadataProvider modelMetadataProvider)

    {

        \_hostingEnvironment = hostingEnvironment;

        \_modelMetadataProvider = modelMetadataProvider;

    }

    public override void OnException(ExceptionContext context)

    {

        if (!\_hostingEnvironment.IsDevelopment())

        {

            // do nothing

            return;

        }

        var result = new ViewResult {ViewName = "CustomError"};

        result.ViewData = new ViewDataDictionary(\_modelMetadataProvider,context.ModelState);

        result.ViewData.Add("Exception", context.Exception);

        // TODO: Pass additional detailed data via ViewData

        context.Result = result;

    }

}

若要处理异常，请将 ExceptionContext.ExceptionHandled 属性设置为 true，或编写响应。 这将停止传播异常。

## **结果筛选器**

实现 IResultFilter 或 IAsyncResultFilter 接口。

下面是一个添加 HTTP 标头的结果筛选器示例。

public class AddHeaderFilterWithDi : IResultFilter {

    private ILogger \_logger;

    public AddHeaderFilterWithDi (ILoggerFactory loggerFactory) {

        \_logger = loggerFactory.CreateLogger<AddHeaderFilterWithDi> ();

    }

    public void OnResultExecuting (ResultExecutingContext context) {

        var headerName = "OnResultExecuting";

        context.HttpContext.Response.Headers.Add (

            headerName, new string[] { "ResultExecutingSuccessfully" });

        \_logger.LogInformation ($"Header added: {headerName}");

}

    public void OnResultExecuted (ResultExecutedContext context) {

        // Can't add to headers here because response has already begun.

    }

}

**关于context参数的数据**

1.获取控制器名称代码

MVC Core中需强行转下

var controllerName = ((Microsoft.AspNetCore.Mvc.Controllers.ControllerActionDescriptor)filterContext.ActionDescriptor).ControllerName;

2.获取Action名称代码

var actionName = filterContext.ActionDescriptor.ActionName;

 3.获取Action参数名称

//获取参数数组

var arrParameter = filterContext.ActionDescriptor.GetParameters();

//根据索引获取对应参数名

var paramName = arrParameter[0].ParameterName;

4.获取参数值

var parameterValue = filterContext.Controller.ValueProvider.GetValue(paramName).RawValue;

如果可以确定参数名称可以直接用ActionParameters通过Key来获取，Key指参数名称

var parameterValue = filterContext.ActionParameters["KeyName"];