ASP.NET Core 核心特性学习笔记「下」

今天花时间看完了 ASP.NET Core 中剩下两块相对重要的内容，还剩下异常处理、配置和日志，就不写了，就看看自己能用到的。

下面这两块比较复杂内容也比较多，给我看麻了，需要结合动手写写才能理解。很多东西有个印象用到的时候再去查阅即可。

依赖注入

依赖注入（Dependency injection，DI）是一种实现对象及其合作者或依赖项之间松散耦合的技术。

ASP.NET Core 框架内部集成了自身的依赖注入容器，在 ASP.NET Core 中，所有被放入依赖注入容器的类型或组件成为服务。分为两类，第一种是框架服务，是 ASP.NET Core 框架的组成部分；另一种是应用服务，所有由用户放到容器中的服务都属于这一类。

在程序中使用服务需要向容器添加服务，然后通过构造函数以注入的方式注入所需要的类。若要添加服务，则需要使用 Startup 类的 ConfigureService 方法，该方法有一个 IServiceCollection 类型的参数，它位于 Microsoft.Extensions.DependencyInjection 命名空间，如下：

```
public void ConfigureServices(IServiceCollection services){  services.Add(new ServiceDescriptor(typeof(IBook), typ
eof(Book), ServiceLifetime.Scoped));}
```

使用了 IServiceCollection 的 Add 方法添加了一个 ServiceDescriptor，ServiceDescriptor 类用来描述一个服务和对应的实现，以及其生命周期；如上构造函数，前两个参数分别是接口及其实现的类型，第 3 个参数是其生命周期。

在 ASP.NET Core 中内置的依赖注入容器中，服务的生命周期有如下 3 种：

- Singleton：容器会创建并共享服务的单例，且会一直存在于应用程序的整个生命周期。
- Transient：每次服务被请求时，总会创建新实例
- Scoped：每一次请求时总会创建新实例，并在这个请求内一直共享这个示例。

当服务添加至容器中后，就可以在程序中使用了，例如在 Controller 中或 Startup 类的 Configure 方法中。使用的方式有以下几种：构造函数注入、方法注入和通过 HttpContext 手工获取。

别怀疑，我也看不懂，看完就俩字“你在说什么钩 8？”，这里引用一个别的例子，很形象：

> 软件设计原则中有一个依赖倒置原则（DIP），就是为了解耦；高层模块不应该依赖于底层模块。二者都应该依赖于抽象；抽象不应该依赖于细节，细节应该依赖于抽象；而依赖注入是实现这种原则的方式之一；
> 举个现实中例子：小明去行政领一节 5 号电池，然后行政给了小明一节黑象牌 5 号电池来分析 ；
> 小明只需要向行政领一节 5 号电池即可，小明不需要关心什么牌子的电池，电池从哪来的，电池的价格等等。他们俩共同需要关心的是一节 5 号电池即可；即使后期行政给了小明北孚电池，小明仍可以正常使用；他们只需要满足一个规则（5 号电池）即可；小明（高层模块）不应该依赖黑象牌电池（低层模块），两者应该都依赖 5 号电池（抽象）。如果小明直接获取到（new）黑象牌电池，如果后期业务变更提供的是北孚电池，那么我们就需要更改小明的代码；再如果公司有几百个小明，代码量可想而知；为了解决直接获取（new）黑象牌电池，简单说为了解耦，我们让每位员工通过行政领取（构造函数，属性，方法等），这种即使更改其他品牌，而小明压根不需要关心；

最常见的就是构造注入：

```
public class ProductService{    private readonly IProductRepository _productRepository;    public ProductService(IProductRepository productRepository)    {        _productRepository = productRepository;    }    public void Delete(int id)    {        _productRepository.Delete(id);    }}
```

ProductService 将 IProductRepository 作为依赖项注入其构造函数中，然后在 Delete 方法中使用它。

还有两种分别是属性注入和方法注入。属性注入是通过设置类的属性来获取所需要的依赖；方法注入是通过在方法的参数中传入所需要的依赖。

（能看懂构造注入就行...

MVC

MVC 是模型（Model）、视图（View）、控制器（Controller），是一种常见的 Web 应用程序的架构模式，

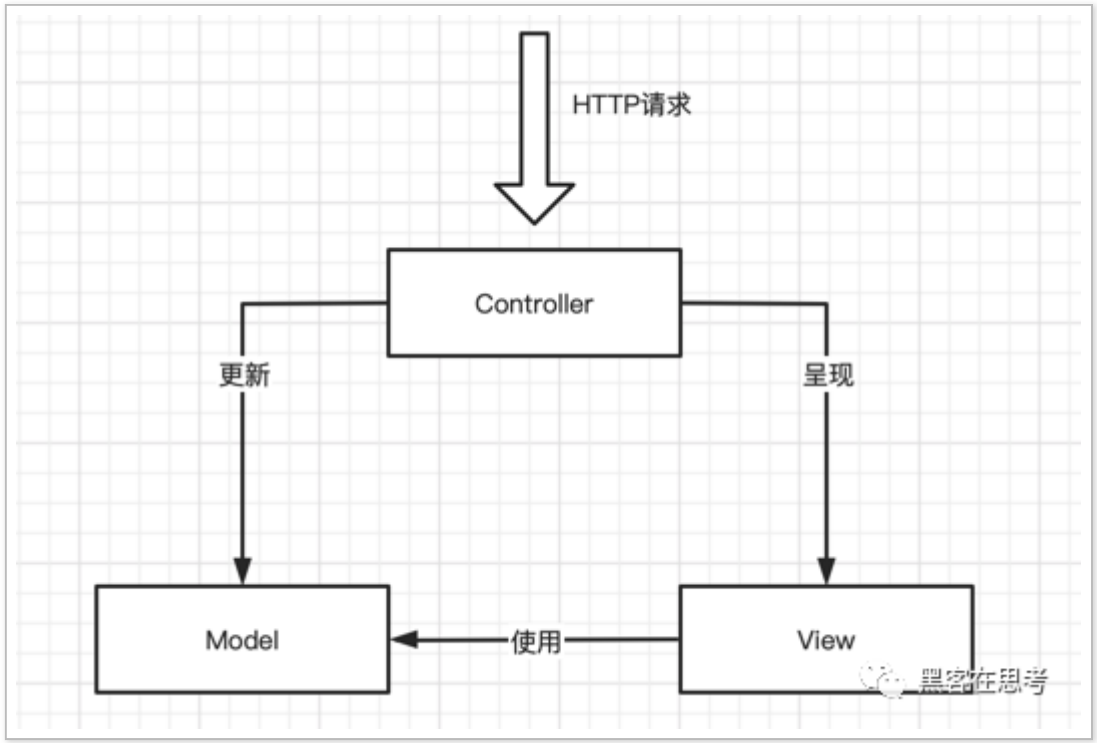


image-20211108194537720

Controller 的角色比较重要，介于 Model 与 View 之间，起到了入口点的作用。当应用程序收到请求时，ASP.NET Core MVC 会将请求路由到相应的 Controller，Controller 操作 Model 完成对数据的更改。不仅如此，Controller 还会将获取到的数据传给对应的 View。ASP.NET Core MVC 是构建在 ASP.NET Core 之上的，若使用需要添加中间件。

```
Public void ConfigureServices(IServiceCollection services){  services.AddMvc();}Public void Configure(IApplication Builder app, IHostingEnvironment env){  app.UseMvc();}
```

2.1 路由

路由负责从请求的 URL 中获取信息，并根据这些信息来定位或映射到对应的 Controller 与 Action

ASP.NET Core 中提供了创建路由和路由处理器的接口，要创建路由，首先要先添加与路由相关的服务，然后配置路由中间件。

```
public void ConfigureServices(IServiceCollection services){    services.AddRouting();}// This method gets called by the runtime. Use this method to configure the HTTP requpublic void Configure(IApplicationBuilder app, IWebHostEnvironment env){    var trackPackageRouteHandler = new RouteHandler(context =>    {        var routeValues = context.GetRouteData().Values;        return context.Response.WriteAsync($"路由值:{routeValues.ToString()}");    });    var routeBuilder = new RouteBuilder(app,trackPackageRouteHandler);    routeBuilder.MapRoute("Track Package Route", "package/{operation}/{id:int}");    routeBuilder.MapGet("hello/{name}", context =>    {        var name = context.GetRouteValue("name");        return context.Response.WriteAsync($"Hi {name}");    });    var routes = routeBuilder.Build();    app.UseRouter(routes);}
```

在上述代码的 Configure 方法中，首先创建一个 RouteHandler, 即路由处理器，它会从请求的 URL 中获取路由信息，并将其输出；接着，创建一个 RouteBuilder; 并使用它的 MapRoute 方法来添加路由信息，这些信息包括路由名称以及要匹配的 URL 模板，在上面的实例中，URL 模板的值为 package/{operation}/{id:int}。除了调用 MapRoute 外，后面还可以使用 MapGet 方法添加仅匹配 GET 方法的请求，最后调用 IApplicationBuilder 的 UseRouter 扩展方法来添加路由中间件。

对于 ASP.NET Core MVC，定义路由的方法有以下两种。

- 基于约定的路由：基于约定的路由会根据一些约定来创建路由，它要在应用程序的 Startup 类中来定义，事实上，上面的实例就是基于约定的路由。
- 特性路由：使用 c# 特性对 Controller 和 Action 指定其路由信息。

.NET MVC 里面封装了上述逻辑

基本约定的路由

要使用基本约定的路由，首先定义一个或若干个路由约定，同时，只要保证所有定义的路由约定能够尽可能满足不同形式的映射即可。

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env){    app.UseMvc(routes =>    {        route
s.MapRoute(            template:$"{{controller}}/{action}");    });}
```

在大括号 {} 中的部分是路由参数，每一个参数都有一个名称，它们充当了占位符的作用，参数与参数之间以“/”分隔。

比如下面的 URL：

http://localhost:5001/home/index

http://localhost:5001/account/register

他们会分别映射到 HomeController 的 Index 方法以及 AccountController 的 register 方法。

举例子：

```
routes.MapRoute( template: "{controller}/{action}/{id?}",  defaults: new { controller = "Home", action = "Index"})
```

也可以写成：

```
routes.MapRoute(  "default": "{Controller=Home}/{action=Index}/{id?}")
```

最后一个参数 id（问号代表可选），会向 action 映射的方法同名参数传值，如上例子的 HomeController 可能就是：

```
public class HomeController : Controller{  public IActionResult Index(){    return Ok("Hello");  }     public IAct
ionResult Welcome(){    return Ok("Hello,Your id:" + id);  }}
```

当请求 URL 为 / Home/Welcome/1 时，URL 中的 1 会传递给 Welcome 方法的 id 参数，最后的 id 还可以通过别的函数限制，比如 length()、range() 和正则等。

特性路由

如下格式，只需要在 Controller 类或者 Action 方法上添加 Route 特性即可

```
public class HomeController: Controller{    [Route("")] [Route("Home/Index")] [Route("AnotherOne")] public IAction
Result Index() {    return Ok("Hello"); }}
```

下面三个 URL 都能映射到这个 Action 上

http://locahost:5001

http://localhost:5001/Home/Index

http://localhost:5001/AnotherOne](http://localhost:5001/AnotherOne)

如果要为 Action 方法传递固定参数的话，如下：

```
[Route("[action]/{name?}")]public IActionResult Welcome(string name){  ...}
```

指定 HTTP 方法的话：

```
[HttpGet("{id:int}")]public IActionResult Welcome(int id){  ...}
```

2.2 Controller 与 Action

在 ASP.NET Core MVC 中，一个 Controller 包括一个或多个 Action，而 Action 则是 Controller 中一些 public 类型的函数，它们可以接受参数、执行相关逻辑，最终返回一个结果，该结果作为 HTTP 响应返回给发起 HTTP 请求的客户端。对于 MVC 视图应用而言，Action 返回的是 View；而对于 Web API 应用程序来讲，则返回响应的资源或者 HTTP 状态码。

根据约定，Controller 通常应放在应用程序根目录下的 Controller 目录中，并且它继承自 using Microsoft.AspNetCore.Mvc; 命名空间下的 Controller 类，而这个 Controller 类由继承自自己的 ControllerBase 抽象类。此外，在类的命名上应以 Controller 结尾。

```
using Microsoft.AspNetCore.Mvc;public class HomeController : Controller{  }
```

每个 Action 都应返回 IActionResult 类型或 ActionResult 类型的值作为 HTTP 请求的结果。在 ASP.NET Core MVC 中，Action 的返回结果有几种比较常见的类型，包括状态码、包含对象的状态码、重定向和内容。

返回具体的状态码：StatusCode：`return StatusCode(403);`

更直观的方法是，使用 StatusCode 静态类，该类定义了所有可能的状态码常量：

```
return StatusCode(Microsoft.AspNetCore.Http.StatusCodes.Status100Continue);
```

```
return StatusCode(Microsoft.AspNetCore.Http.StatusCodes.Status100Continue);
```

包含对象的状态码，这一类结果继承自 ObjectResult，包括 OkObjectResult、CreatedResult 和 NotFoundObjectResult 等：

```
var result = new OkObjectResult(new { message ="操作成功",currentDate = DateTime.Now});return result;
```

重定向结果包括 RedirectResult、LoaclRedirectResult、RedirectToActionResult 和 RedirectToResult 等

```
//重定向到指定的URLreturn Redirect("http://www.microsoft.com/");//重新定向到当前应用程序中的另一个URLreturn LocalRedirect("/account/login");//重定向到指定的Actionreturn RedirectToAction("login");//重定向到指定的路由return RedirectToRoute("default",new { action="logion",controller = "account"});
```

内容结果包括 ViewResult、ParialViewResult、JsonResult 和 ContentResult 等，其中 ViewResult 和 PariaViewResult 在 MVC 试图应用中非常常见，用于返回响应的页面；JsonResult 用于返回 JSON 字符串，ContentResult 用于返回一个字符串。

```
return JsonResult(new { message="This is a JSON result.",data=DateTime.Now});return Content("返回字符串");
```

除了返回 IActionResult 外，当在 Action 要返回数据时，还可以使用 ActionResult 类，它既可以表示一个 ActionResult 对象，也可以表示一个具体类型。

```
[HttpGet("{id}")]public ActionResult<Employee> Get(long id){    if (id<=0)    {        return BadRequest();    }    var employee = new Employee(id);    if (employee == null)    {        return NotFound();    }    return employee;}
```

ActionResult 的优点在于更为灵活地为 Action 设置返回值，同时，当使用 OpenAPI(即 Swagger)为 API 生成文档时，Action 不需要使用 [Produces] 特性显示地指明其返回类型，因为其中的泛型参数 T 已经为 OpenAPI 指明了要返回的数据类型。

2.3 模型绑定

在 ASP.NET Core MVC 中，当一个 HTTP 请求通过路由定位到 Controller 中的某一个 Action 上时，HTTP 请求中的一部分信息会作为 Action 的参数。在 URL 中的 id 或 name 会传递给 Action 中的同名参数。将 HTTP 请求中数据映

射到 Action 中参数的过程称为模型绑定。

```
[Route("api/[controller]")]public class BlogsControlls : Controller{    [HttpGet("[action]/{keyword}")]    public
 IActionResult Search(string keyword, int top)    {        return NotFound();    }}
```

在上面的例子中：当请求的 URL 为 https://localhost:5001/api/blogs/search/web?top=10 时，其中的 web 和 10 会分别传递给 Search 方法的两个参数 keyword 和 top。MVC 在进行模型绑定时，会通过参数名在多个可能的数据源中进行匹配。第一个参数 keyword 是在路由中指定的，它的值会直接从 URL 中响应的部分解析得到；而第二个参数 top 并未在路由中定义，因此 ASP.NET Core MVC 会尝试从查询字符串中获取。

除了从路由以及查询字符串中获取数据以外，ASP.NET Core MVC 还会尝试从表单（Form）中获取数据来绑定到 Action 中的参数。因此，它主要使用以下 3 中数据源来为 Action 的参数提供数据，并且按照顺序来从以下每一种方式中获取：

- Form 值：HTTP POST 请求时表单中的数据
- 路由值：通过路由系统解析得到
- 查询字符串：从 URL 中的查询字符串中获取

像特性路由一样，ASP.NET Core MVC 也提供了用于模型绑定的特性，使用如下特性能够为 Action 的参数显示指定不同的绑定源。

```
[FromHeader]特性:从HTTP请求的Header中获取参数的值
[FromQuery]特性：从查询字符串中获取参数的值
```

[FromServices]特性：从依赖注入容器中获取参数的值
[FromRoute]特性：从路由中获取参数的值
[FromForm]特性：从表单中获取该参数的值
[FromBody]特性：从HTTP请求的消息正文获取参数的值。

另外还有两个特性用于指明参数是否必须使用绑定

BinRequiredAttribute:如果没有值绑定到此参数，或绑定不成功，这个特性将添加一个ModelState错误。
BinNeverAttribute:在进行模型绑定时，忽略此参数。

比如：

```
[HttpGet("[action]/{keyword}")]public IActionResult Post([FromBody] string keyword, [FromHeader]int id){    ...}
```

要正确访问这个 Action 的话，请求如下：

```
POST /api/post1 HTTP/1.1
Host: localhost:5001
id: testid
Conten-Type: application/json

keyword=testkeyword
```

2.4 模型验证

模型验证是指数据被使用之前的验证过程，** 它发生在模型绑定之后。** 在 ASP.NET Core MVC 中，要实现对数据的验证，最方便的方式是使用数据注解（Data annotation），它使用特性为数据添加额外的信息。数据注解通常用于验证，只要为类的属性添加需要的数据注解即可，这些特性位于 `System.ComponentModel.DataAnnotations` 命名空间下。

```
public class BlogDto{    [Required]    public int Id { get; set; }    [Required, MinLength(10, ErrorMessage = "验证失败，自定义的错误提示信息")]    public string Title { get; set; }    [MaxLength(1000)]    public string Content { get
```

```
; set; }    [Url]    public string Url { get; set; }    [Range(1,5)]    public int Level { get; set; }}
```

补充，上一小节的模型绑定不仅可以处理基本数据类型，也可以是上面这种，类似于 C 语言结构体一样，很多个属性。

在 Controller 内的 Action 中，要检查某一个对象是否满足指定的条件，只要调用 ModelState.IsValid 属性，其中 ModelState 是 ControllerBase 类的属性

2.5 过滤器

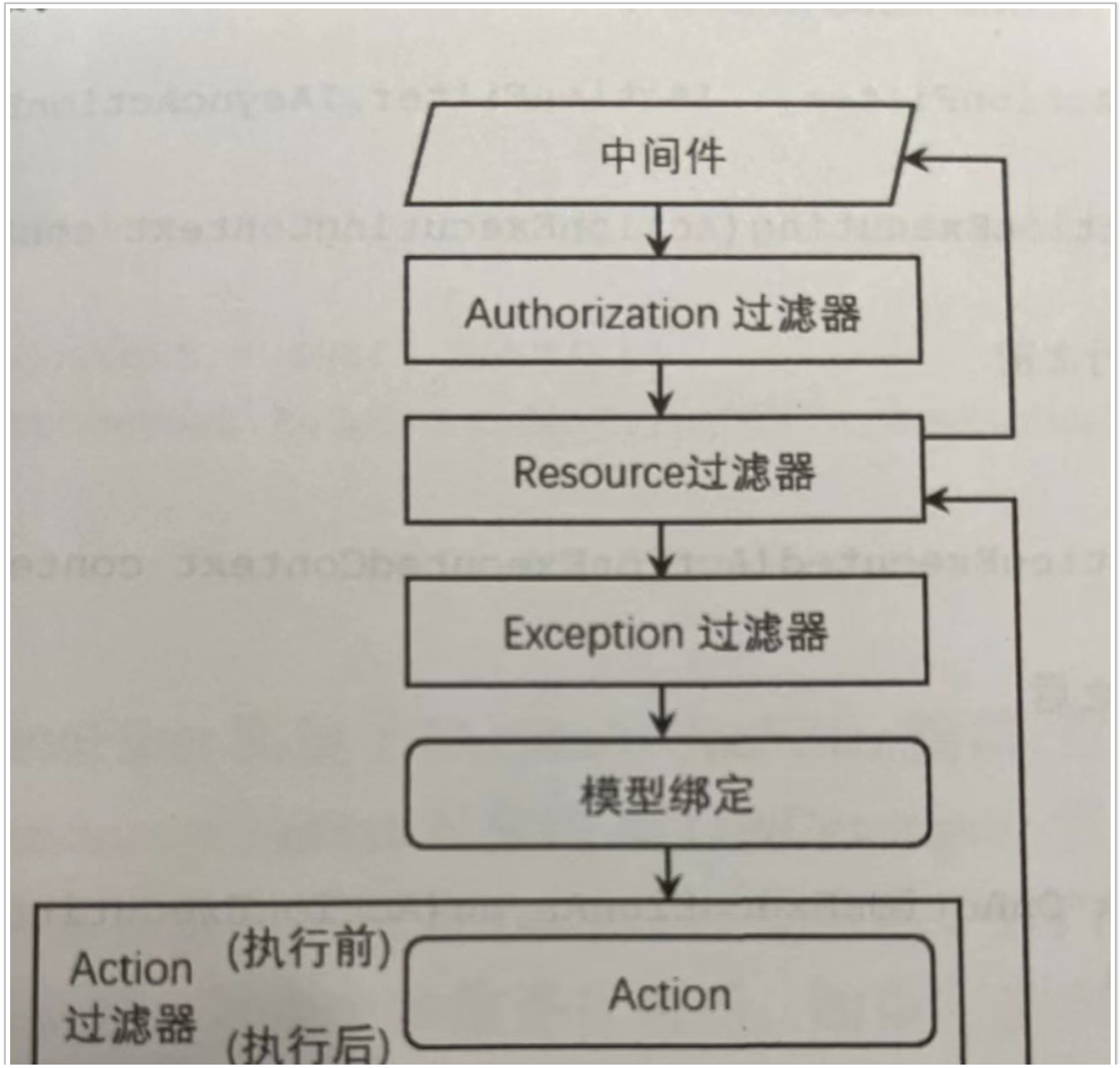
过滤器和中间件很相似，在 ASP.NET Core MVC 中它们能够在某些功能前后执行，由此而形成一个管道。如果，在 Action 方法开始执行前与执行后运行，因此它能够极大地减少代码重复，如果一些代码需要每个 Action 之前执行，那么只要使用一个 Action 过滤器即可，而无需添加重复的代码。

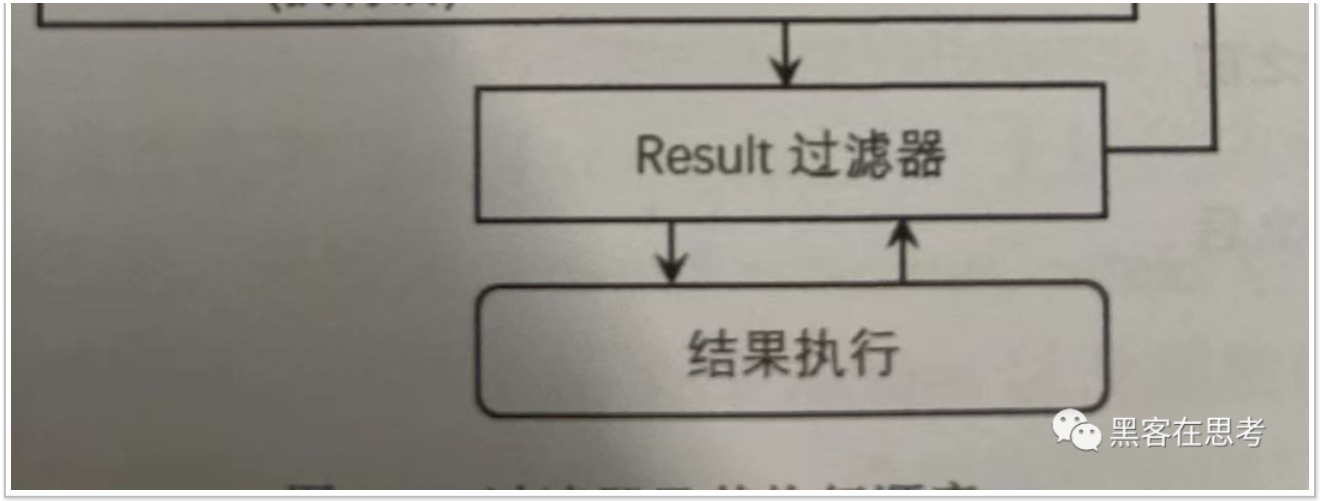
ASP.NET Core MVC 提供了以下 5 种类型的过滤器。

- Authorization 过滤器：最先执行，用于判断用户是否授权，如果未授权，则直接结束当前请求，这种类型的过滤器实现了 IAsyncAuthorizationFilter 或 IAuthorizationFilter 接口。
- Resource 过滤器：在 Authorization 过滤器后执行，并在执行其他过滤器（除 Authorization 过滤器外）之前和之后执行，由于它在 Action 之前执行，因而可以用来对请求判断，根据条件来决定是否继续执行 Action，这种类型过滤器实现了 IAsyncResourceFilter 或 IResourceFilter 接口
- Action 过滤器：在 Action 执行的前后执行，与 Resource 过滤器不一样，它在模型绑定后执行，这种类型的过滤器实现了 IAsyncActionFilter 或 IActionFilter 接口
- Exception 过滤器：用于捕获异常，这种类型的过滤器实现了 IAsyncExceptionFilter 或 IExceptionFilter 接口
- Result 过滤器：在 IActionResult 执行的前后执行，使用它能够控制 Action 的执行结果，比如格式化结果等。需要注意的是，它只有在 Action 方法成功执行后才会运行，这种类型的过滤器实现了 IAsyncResultFilter 或

要注意的是，它只有在 Action 方法成功执行后才会运行，这样类型的过滤器实现了 IAsyncResultFilter 或 IRsultFilter 接口。

工作顺序：





当要创建过滤器的时候，应该实现 IXXXFilter 或 IAsyncXXXFilter，这两个接口的区别是前者同步、后者异步。ASP.NET Core MVC 会首先检查异步实现，如果没有实现异步方式，则继续检查同步实现，因此在创建过滤器的时，不需要同步和异步接口都实现。

以 IAsyncActionFilter 和 IActionFilter 为例，这两个接口的定义分别如下：

```
public interface IAsyncActionFilter : IFilterMetadata{    Task OnActionExecutionAsync(ActionExecutedContext contex
t,ActionExecutionDelegate next);}public interface IActionFilter : IFilterMetadata{    void OnActionExecuted(Action
ExecutedContext context);    void OnActionExecuting(ActionExecutingContext context);}
```

在 IActionFilter 接口中包括两个方法，分别表示 Action 执行前与执行后要执行的方法；在 IAsyncActionFilter 接口中仅有一个 OnActionExecutionAsync 方法，该方法的第二个参数 ActionExecutionDelegate 表示要执行的 Action，它是一个**委托类型**，因此在这个方法的内部可以直接调用 next()，并在 next() 前后执行相应的代码。

下面的代码展示了一个自定义过滤器同时实现了异步与同步的 Action 过滤器接口：

```
public class CustomActionFilter : IActionFilter, IAsyncActionFilter{    public void OnActionExecuting(ActionExecut
ingContext context)    {        //Action执行前    }    public void OnActionExecuted(ActionExecutedContext contex
```

```
t)    {        //Action执行后    }    public Task OnActionExecutionAsync(ActionExecutedContext context, ActionExecu
tionDelegate next)    {        //Action执行前        next();        //Action执行后        return Task.CompletedTas
k;    }}
```

参考

《第三章 ASP.NET Core 核心特性》

https://www.debugger.wiki/article/html/1585477416849943

https://www.jianshu.com/p/cc95657a2a2a