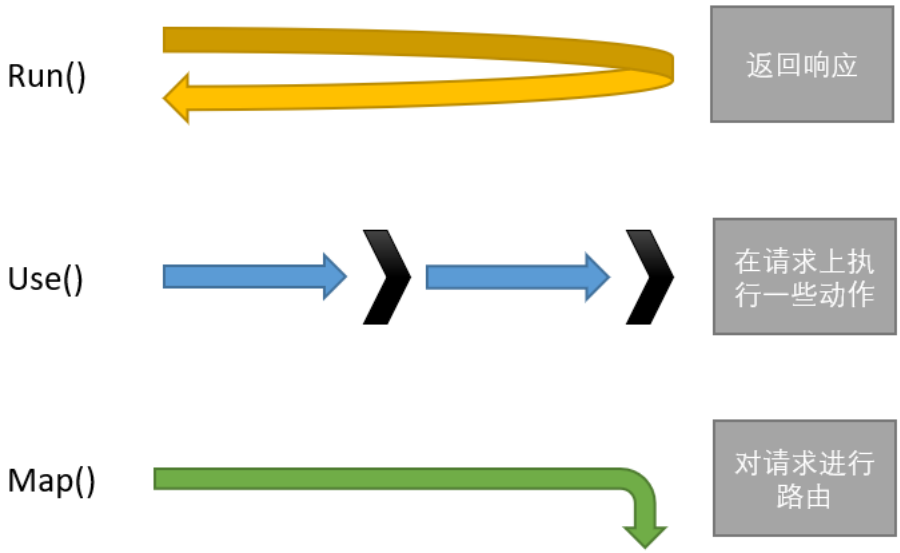
草根专栏

一个.NET Core开发者的博客

ASP.NET Core 3.x 中间件流程与路由体系

中间件分类



ASP.NET Core 中间件的配置方法可以分为以上三种，对应的Helper方法分别是：Run(), Use(), Map()。

- **Run()**，使用Run调用中间件的时候，会直接返回一个响应，所以后续的中间件将不会被执行了。
- **Use()**，它会对请求做一些工作或处理，例如添加一些请求的上下文数据，有时候甚至什么也不做，直接把请求交给下一个中间件。
- **Map()**，它会把请求重新路由到其它的中间件路径上去。

实际中呢，Use()这个helper方法用的最多。

Run():

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.Run(handler:async context ⇒
    {
        await context.Response.WriteAsync(text:"Hello world!");
    });
}
```

这是一个使用Run方法调用的中间件，Run方法会终止整个中间件管道，它应该返回某种类型的响应。

Use():

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.Use(async (context, next) ⇒
    {
        // 在这里写next中间件之前的业务逻辑

        await next.Invoke();

        // 在这里写next中间件之后的业务逻辑
    });

    app.Run(handler:async context ⇒
    {
        await context.Response.WriteAsync(text:"Response!");
    });
}
```

Use看起来和Run差不多，但是多了一个next参数。next可以用来调用请求管道中的下一个中间件。而当前的中间件也可以自己返回响应，这就忽略掉了next调用。

在next调用之前，我们可以写一些请求进来的逻辑，而在next调用之后，就相当于返回响应了，这时候也可以写一些逻辑。

在本例中，我们下面还使用了Run方法注册了另一个中间件。因为中间件会按照它们注册的顺序进行调用，所以在第一个Use方法里执行next.Invoke()的时候，就会执行下面Run所调用的中间件。

Map():

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.Map(pathMatch:"/jump", HereIAm);
}

1 reference
private static void HereIAm(IApplicationBuilder app)
{
    app.Run(handler:async context ⇒ { await
        context.Response.WriteAsync(text:"Here I am."); });
}
```

Map方法可以把请求路由到其它的中间件上面。

在这里，如果请求的路径以/jump结尾，那么它所对应的handler方法，也就是HereIAm方法就会被调用，并返回一个响应。

而如果请求的路径不是以/jump结尾，那么HereIAm方法和里面的中间件就不会被调用。

中间件Class

上面的例子，我都是使用的inline写法的中间件。

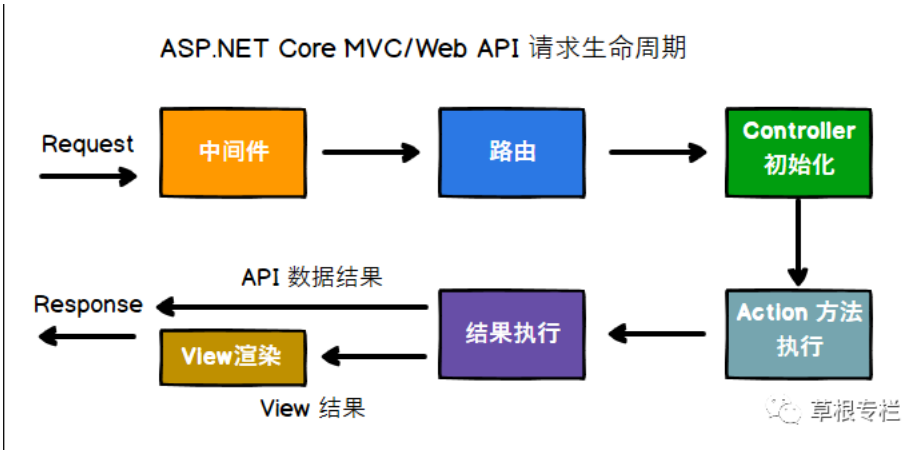
而实际上，中间件通常是自成一个类。中间件的类需要类似这样：

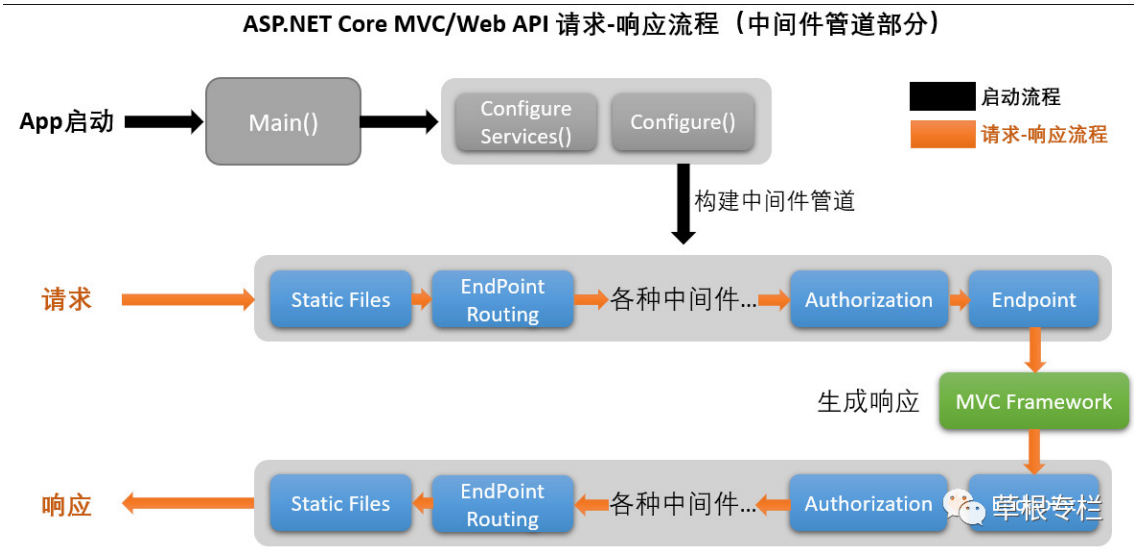
```
public class MyMiddleware
{
    private readonly RequestDelegate _next;

    0 references
    public MyMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    0 references
    public async Task Invoke(HttpContext context)
    {
        await _next.Invoke(context);
    }
}
```

自定义的中间件类需要由这几部分组成：

- 接受一个RequestDelegate类型的参数next的构造函数。
- 按约定，还需要定义一个叫做Invoke的方法。该方法里会包含主要的业务逻辑，并且它会被请求管道所执行。Invoke方法可以忽略里面的_next调用，并返回一个响应；也可以调用_next.Invoke()把请求发送到管道的下一站。

中间件流程图





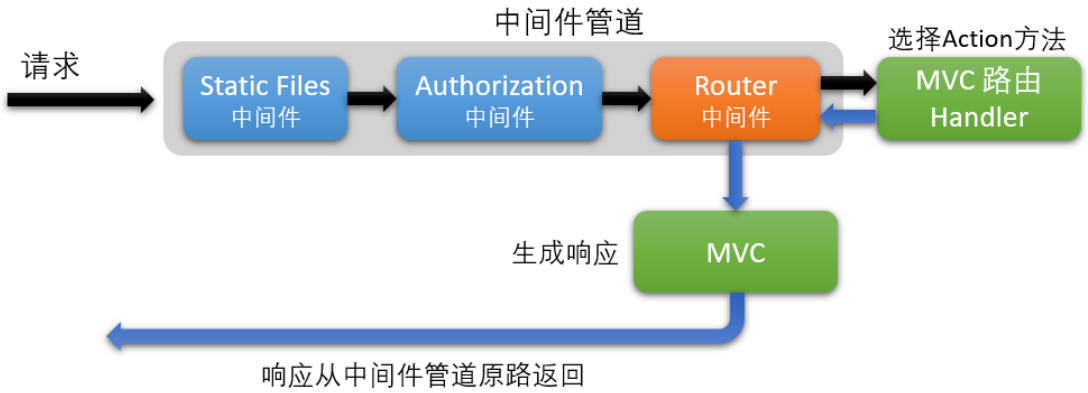
Endpoint Routing 路由系统

ASP.NET Core 3.x 使用了一套叫做 Endpoint Routing 的路由系统。这套路由系统在ASP.NET Core 2.2的时候就已经露面了。

这套Endpoint Routing路由系统提供了更强大的功能和灵活性，以便能更好的处理请求。

早期ASP.NET Core的路由系统

我们先回顾一下早期版本的ASP.NET Core的路由系统：



在早期的ASP.NET Core框架里，HTTP请求进入中间件管道，在管道的结尾处，有一个Router中间件，也就是路由中间件。这个路由中间件会把HTTP请求和路由数据发送给MVC的一个组件，它叫做MVC Router Handler。

这个MVC 路由 Handler就会使用这些路由数据来决定哪个Controller的Action方法应该来负责处理这个请求。

然后 Router中间件就会执行被选中的Action方法，并生成响应，而这个响应就会顺着中间件的管道原路返回。

问题出在哪?

为什么早期的这套路由系统被抛弃了？它有什么问题?

第一个问题就是，在被MVC处理之前，其它的中间件不知道最后哪个Action方法会被选中来处理这个请求。这对像Authorization（授权），Cors这样的中间件会造成很大的困扰，因为他们不能提前知道该请求会被送往哪里。

第二个问题就是，这套流程会把MVC和路由的职责紧密的耦合在一起，而实际MVC的本职工作应该仅仅就是生成响应。

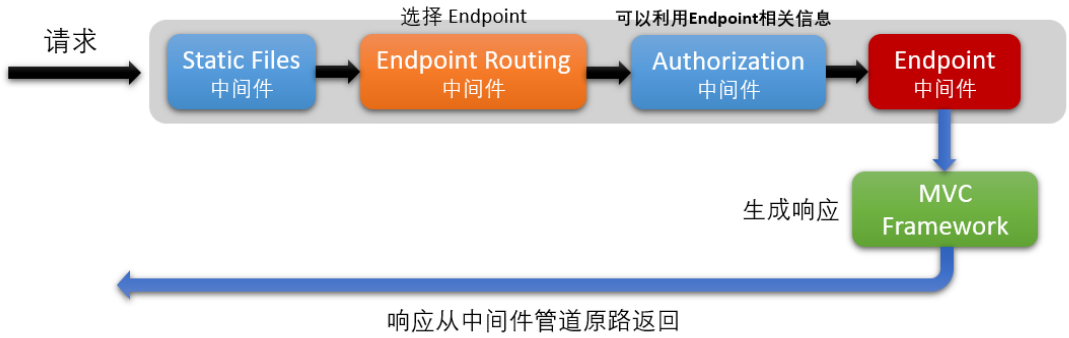
Endpoint Routing 路由系统前来营救

Endpoint routing 路由系统，它把MVC的路由功能抽象剥离出来，并放置到中间件管道里，从而解决了早期ASP.NET Core路由系统的那两个问题。

而在Endpoint Routing 路由系统里，其实一共有两个中间件，它们的名称有点容易混淆，但是你只要记住他们的职责即可：

1. **Endpoint Routing 中间件。**它决定了在程序中注册的哪个Endpoint应该用来处理请求。
2. **Endpoint** 中间件。它是用来执行选中的Endpoint，从而让其生成响应的。

所以，Endpoint Routing的流程图大致如下：



在这里，Endpoint Routing 中间件会分析进来的请求，并把它和在程序中注册的Endpoints进行比较。它会使用这些 Endpoints 上面的元数据来决定哪个是处理该请求的最佳人选。然后，这个选中的Endpoint 就会被赋给请求的对象，而其它后续的中间件就可以根据这个选中的Endpoint，来做一些自己的决策。在所有的中间件都执行完之后，这个被选中的Endpoint最终将被 Endpoint中间件所执行，而与之关联的Action方法就会被执行。

Endpoint是什么?

Endpoint是这样的一些类，这些类包含一个请求的委托（Request Delegate）和其它的一些元数据，使用这些东西，Endpoint类可以生成一个响应。

而在MVC的上下文中，这个请求委托就是一个包装类，它包装了一个方法，这个方法可以实例化一个Controller并执行选中的Action方法。

Endpoint还包含元数据，这些元数据用来决定他们的请求委托是否应该用于当前的请求，还是另有其它用途。

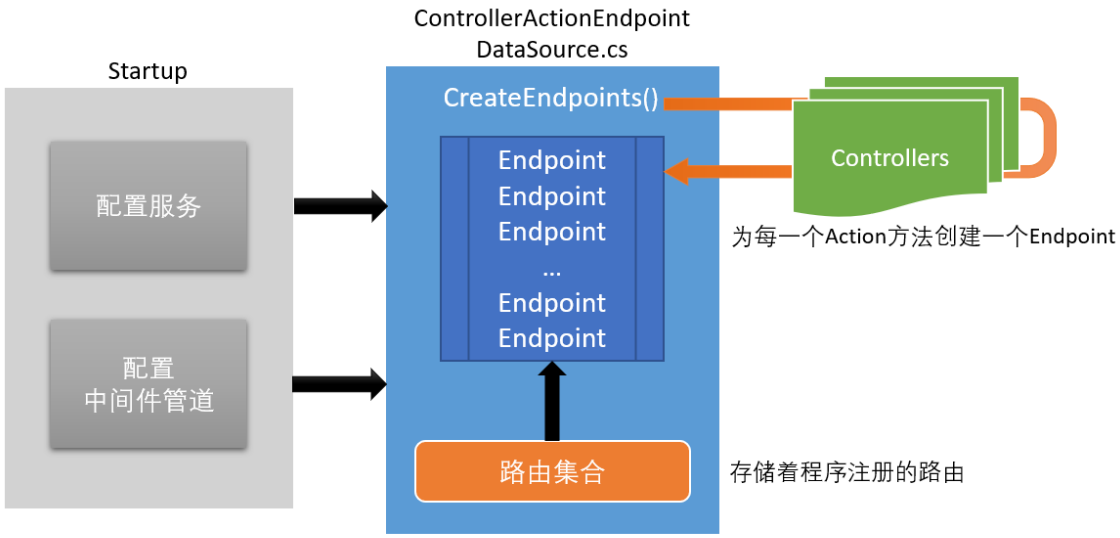
说起来可能有点迷糊，一会我们看看源码。

Startup.cs

之前我们见过，ASP.NET Core里面的Startup.cs里面有两个方法，分别是ConfigureServices()和Configure()，它们的职责就是注册应用的一些服务和构建中间件请求管道。

而Startup.cs同时也是应用的路由以及Endpoint作为其它步骤的一分部进行注册的地方。

看图：



在ASP.NET Core应用程序启动的时候，一个叫做ControllerActionEndpointDataSource的类作为应用程序级别的服务被创建了。

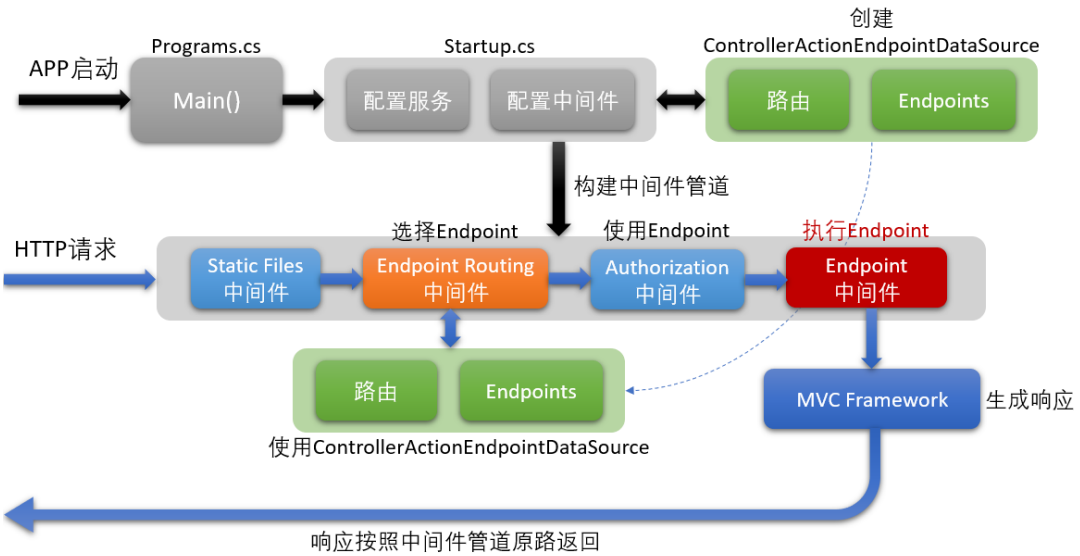
这个类里面有一个叫做CreateEndpoints()的方法，它会获取所有Controller的Action方法。

然后针对每个Action方法，它会创建一个Endpoint实例。这些Endpoint实例就是包装了Controller和Action方法的执行的请求委托（Request Delegate）。

ControllerActionEndpointDataSource里面包存储着在应用程序里注册的路由模板。

而针对每个Endpoint，它要么与某个按约定的路由模板相关联，要么与某个Controller Action上的Attribute路由信息相关联。而这些路由在稍后就会被用来将Endpoint与进来的请求进行匹配。

从Endpoint的角度查看请求-响应流程图



App启动那部分就不说了。

第一个HTTP请求进来的时候，Endpoint Routing中间件就会把请求映射到一个Endpoint上。它会使用之App启动时创建好的EndpointDataSource，来遍历查找所有可用的Endpoint，并检查和它关联的路由以及元数据，来找到最匹配的Endpoint。

一旦某个Endpoint实例被选中，它就会被附加在请求的对象上，这样它就可以被后续的中间件所使用了。

最后在管道的尽头，当 Endpoint中间件运行的时候，它就会执行Endpoint所关联的请求委托。这个请求委托就会触发和实例化选中的Controller和Action方法，并产生响应。最后响应再从中间件管道原路返回。

我的.NET Core公众号: