ASP.NET Core 核心特性学习笔记「上」

学习瘾发作最严重的一次，躺在床上，拼命念大悲咒，难受的一直抓自己眼睛，眼睛越来越大都要炸开了一样，真的不知道该怎么办，我真的想学习想得要发疯了。我每时每刻眼睛都直直地盯着电脑，像一台雷达一样扫视经过我眼睛的每一个文字，我真的觉得自己像中邪了一样，我的眼睛滚烫滚烫，我发病了，我魔怔了，我要疯狂的学习，我要狠狠的学习，我受不了了，学习，我要狠狠地学习！学习，我要狠狠地学习！ 

开个玩笑，工作之余一直在学习 SharpC2 这款 C2 的代码，其使用了 ASP.NET Core 来实现，尤其是 TeamServer，其实就是实现了一些遵循 RESTful api 规范的 HTTP 接口，供植入体与客户端通信使用；其中使用了很多关于 ASP.NET Core 的核心特性，比如依赖注入、MVC 等，颇有学习价值。

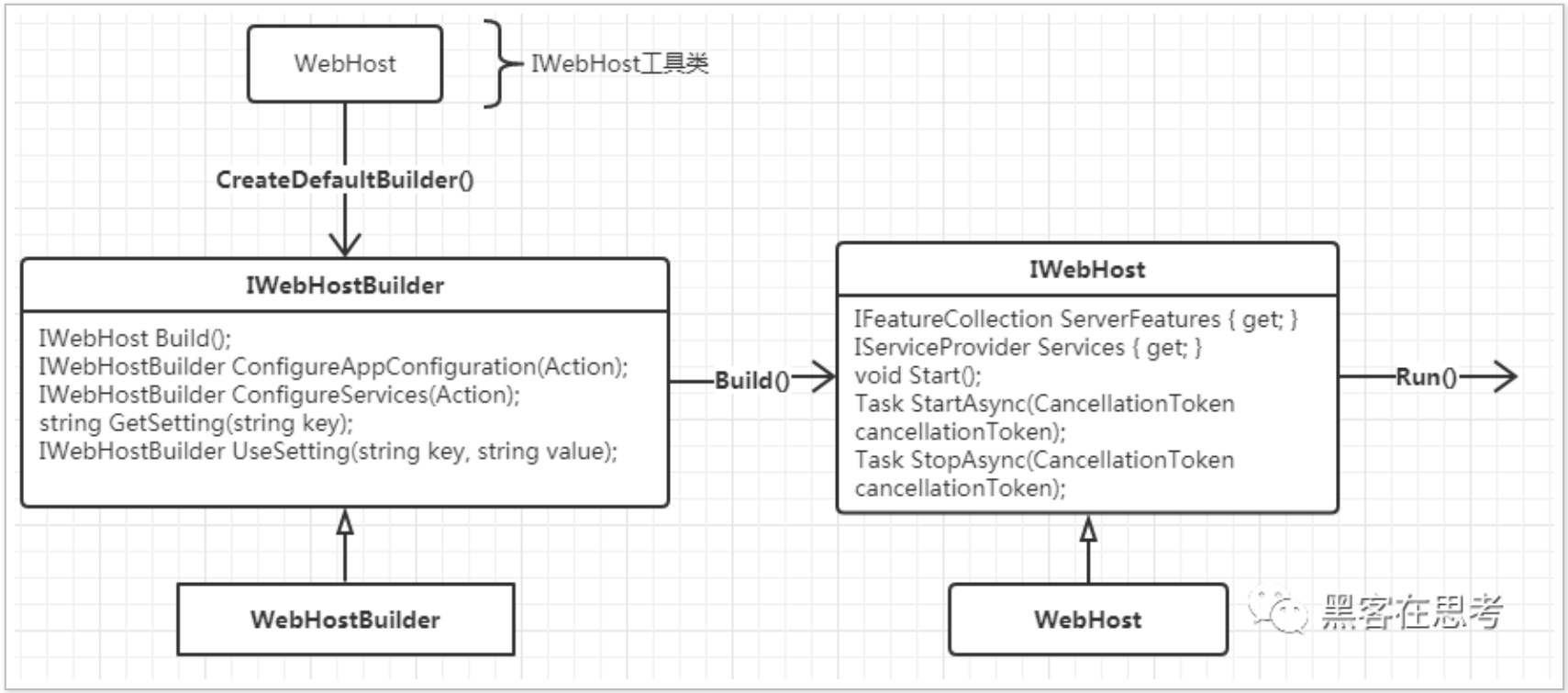
本文只是晚上回家的学习过程中记的笔记，大部分都是参照图书《ASP.NET Core 与 RESTful API 开发实战》第三章，没有太深入，旨在更好的理解 SharpC2 的代码，如果你也想学习 SharpC2 或是. NET5，那么正好。

前言

.NET Core 是一个可以用来构建现代、可伸缩和高性能的跨平台软件应用程序的通用开发框架。可用于为 Windows、Linux 和 MacOS 构建软件应用程序。

ASP.NET Core 本质上是一个独立的控制台应用，它并不是必需在 IIS 内部托管且并不需要 IIS 来启动运行（而这正是 ASP.NET Core 跨平台的基石）

启动流程：



ASP.NET Core 总体启动流程

启动与宿主

1.1 启动

当 ASP.NET Core 应用程序启动时，他首先会配置并运行其宿主 (Host)，宿主主要用来启动、初始化应用程序，管理其生命周期。

可以看到 Program 类是程序的入口，与控制台应用程序一样。

```
public class Program{    public static void Main(string[] args)    {        CreateWebHostBuilder(args).Build().Run
();    }    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>        WebHost.CreateDefaultBuild
er(args)            .ConfigureWebHostDefaults(webBuilder =>            {                Builder.UseStartup<Startup
>();            });}
```

程序首先由 CreateWebHostBuilder 方法创建一个 IWebHostBuilder 对象，并调用 Build 方法得到 IWebHost 对象的实例，然后调用对象的 Run() 方法运行；还有 Start() 方法也可以运行，区别是前者以阻塞的方式运行。

CreateWebHostBuilder 方法内部，调用 WebHost 类的静态方法 CreateDefaultBuilder，返回 IWebHostBuilder 类型的对象，最后调用 UseStartup 方法进一步配置应用程序的启动。

用 CreateDefaultBuilder 方法创建 IWebHostBuilder 对象时所包含的主要默认选项如下：

- 配置 Kestrel 服务器作为默认的 Web 服务器来负责处理 Web 的请求与响应
- 使用当前目录作为应用程序的内容目录（ContentRoot），该目录决定了 ASP.NET Core 查找内容文件（如 MVC 视图等）的位置
- 从以 ASPNETCORE_ 开头的环境变量（如 ASPNETCORE_ENVIRONMENT）中以及命令行参数中加载配置项
- 从 appsettings.json、appsettings.{Environment}.json、用户机密（仅开发环境）、环境变量和命令行参数等位置加载应用设置
- 配置日志功能，默认添加控制台输出和调试输出
- 如果应用程序被托管在 IIS 中，启动 IIS 集成，它会配置应用程序的主机地址和端口，并允许捕获启动错误等

- 如果应用程序被托管在 IIS 中，启动 IIS 失败，它会配置应用程序的主机地址和端口，并允许捕获启动错误等

CreateDefaultBuilder 的代码大致如下：

```
public static IWebHostBuilder CreateDefaultBuilder(string[] args)        {            var builder = new WebHostBuilder();            if (string.IsNullOrEmpty(builder.GetSetting(WebHostDefaults.ContentRootKey)))            {                builder.UseContentRoot(Directory.GetCurrentDirectory());            }            if (args != null)            {                builder.UseConfiguration(new ConfigurationBuilder().AddCommandLine(args).Build());            }            builder.ConfigureAppConfiguration((hostingContext, config) =>            {                var env = hostingContext.HostingEnvironment;                config.AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)                    .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true, reloadOnChange: true);                if (env.IsDevelopment())                {                    var appAssembly = Assembly.Load(new AssemblyName(env.ApplicationName));                    if (appAssembly != null)                    {                        config.AddUserSecrets(appAssembly, optional: true);                    }                }                config.AddEnvironmentVariables();                if (args != null)                {                    config.AddCommandLine(args);                }            })            .ConfigureLogging((hostingContext, loggingBuilder) =>            {                loggingBuilder.Configure(options =>                {                    options.ActivityTrackingOptions = ActivityTrackingOptions.SpanId                        | ActivityTrackingOptions.TraceId                        | ActivityTrackingOptions.ParentId;                });                loggingBuilder.AddConfiguration(hostingContext.Configuration.GetSection("Logging"));                loggingBuilder.AddConsole();                loggingBuilder.AddDebug();                loggingBuilder.AddEventSourceLogger();            }).            UseDefaultServiceProvider((context, options) =>            {                options.ValidateScopes = context.HostingEnvironment.IsDevelopment();            });            ConfigureWebDefaults(builder);            return builder;        }
```

里面的默认配置也能通过 IWebHostBuilder 接口提供的方法进行修改。

完整版 WebHost 代码：

https://github.com/dotnet/aspnetcore/blob/18a926850f7374248e687ee64390e7f10514403f/src/DefaultBuilder/src/WebHost.cs

1.2 Startup 类

IWebHostBuilder 接口有多个扩展方法，一个重要的方法就是 UseStartup 方法，主要功能是向应用程序提供用于配制启动的类，应该具有两个方法

- ConfigureServices：用于向 ASP.NET Core 的依赖注入容器添加服务
- Configure：用于添加中间件，配置请求管道

两个方法都会在运行时被调用，且在应用程序的整个生命周期内，只执行一次。其中 ConfigureServices 是可选的 Configure 是必选

- ConfigureServices 方法有一个参数，为 IServiceCollection 类型，使用它能够将应用程序级别的服务注册到 ASP.NET Core 默认的依赖注入容器中
- Configure 方法默认包函的参数类型为 IApplicationBuilder，通过这个可以添加中间件

比如一个 Startup 类：
https://github.com/dotnet/aspnetcore/blob/8b30d862de6c9146f466061d51aa3f1414ee2337/src/Mvc/test/WebSites/RazorPagesWebSite/Startup.cs

中间件

ASP.NET Core 中还有一个中间件的概念，所谓中间件，就是处理 HTTP 请求和响应的组件，它本质上是一段用来处理请求与响应的代码。多个中间件之间的链式关系使之形成了管道（Pipeline）或者请求管道。管道意味着请求将从一端进入，并按照一定的顺序由每一个中间件处理，最后由另一端出来。

ASP.NET Core 中内置了多个中间件，主要有 MVC、认证、错误、静态文件、HTTPS 和 CORS 等，也允许自定义中间件。

2.1 添加中间件

上一节提到的 Startup 类的 Configure 方法就是用来添加中间件的地方，通过调用 IApplicationBuilder 接口中以 Use 开头的扩展方法即可添加系统内置的中间件，比如：

```
public void Configure(IApplicationBuilder app)        {            app.Mvc();            app.UseAuthentication
(););        }
```

在这些系统内置中间件的内部实现中，每个中间件都是通过调用 IApplicationBuilder 接口的 Use 和 Run 方法添加到请求管道中的。

Run 方法，接受一个 RequestDelegate 类型的参数，它是一个委托，用来处理传入的 HTTP 请求，定义如下：

```
public delegate Task RequestDelegate(HttpContext context);
```

Use 方法不同的是，它会在处理完请求之后还会将请求传入下一个中间件做处理。

例子：

```
app.Use(async (context, next) =>{    Console.WriteLine("中间件 A：开始");    await next();// 下一个中间件    Console.W
riteLine("中间件 A：结束");});app.Run(async (context) =>{    Console.WriteLine("中间件 B");    await context.Respons
e.WriteAsync("Hello, world");});
```

运行结果

```
中间件 A：开始
中间件 B
中间件 A：结束
```

除了 Run 和 Use 外，IApplicationBuilder 接口还提供了其他方法，可以指定一些条件来判断是否进入下一个分支管道，比如有 Map、MapWhen 和 UseWhen。

比如，Map 会根据是否匹配指定的请求路径来决定是否在一个新的分支上继续执行后续的中间件；MapWhen 会对传入的 HttpContext 对象进行更细致的判断，比如是否包含制定的消息头等。

下面只举一个 Map 的例子

```
app.Use(async (context, next) =>{    Console.WriteLine("中间件 A：开始");    await next();// 下一个中间件    Console.W
riteLine("中间件 A：结束");});app.Map(new PathString("/maptest"),    a => a.Use(async (context, next) =>{    Consol
e.WriteLine("中间件 B：开始");    await next(); // 下一个中间件    Console.WriteLine("中间件 B：结束");}));app.Run(asyn
c context =>{    Console.WriteLine("中间件 C");    await context.Response.WriteAsync("Hello, world");});
```

访问 https://localhost:5001/maptest

```
中间件 A：开始
中间件 B：开始
中间件 B：结束
中间件 A：结束
```

可以看到新分支管道执行完毕后不会回到原来的管道，而剩下两个是会继续回去执行。

2.2 自定义中间件

创建自定义中间件需要一个特定的构造函数和一个名为 Invoke 的方法。对于构造函数应包含一个 RequestDelegate 类型的参数，该参数表示在管道中的下一个中间件；而对于 Invoke 方法，应包含一个 HttpContext 类型的参数，并返回 Task 类型。

比如创建自定义中间件让应用程序只接受 GET 和 HEAD 方法：

```
public class HttpMethodCheckMiddleware{    // 在管道中的下一个中间件    private readonly RequestDelegate _nex
t;    // 构造函数中可以得到下一个中间件，并且还可以注入需要的服务，比如 IHostEnvironment    public HttpMethodCheckMiddleware
(RequestDelegate requestDelegate, IHostEnvironment environment)    {        this._next = requestDelegat
```

```
e;    }    // 对 HTTP 请求方法进行判断，如果符合条件则继续执行下一个中间件    // 否则返回 400 Bad Request 错误，并在响应中添加自定义消息头用于说明错误原因    public Task Invoke(HttpContext context)    {        var requestMethod = context.Request.Method.ToUpper();        if (requestMethod == HttpMethods.Get || requestMethod == HttpMethods.Head)        {            return _next(context);        }        else        {            context.Response.StatusCode = 400;            context.Response.Headers.Add("X-AllowHTTPWeb", new[] {"GET,HEAD"});            context.Response.WriteAsync("只支持 GET、HEAD 方法");            return Task.CompletedTask;        }    }}
```

在 Startup 类的 Configure 方法中添加中间件：

```
app.UseMiddleware<HttpMethodCheckMiddleware>();
```

或者创建扩展方法

```
public static class CustomMiddlewareExtensions{    public static IApplicationBuilder UseHttpMethodCheckMiddleware(this IApplicationBuilder builder)    {        return builder.UseMiddleware<HttpMethodCheckMiddleware>();    }}
```

调用扩展方法添加中间件

```
app.UseHttpMethodCheckMiddleware();
```

到这里上篇就结束了，下篇也写了一部分，但是夜色较晚，打算去看会 B 站，所以等下一次某天晚上有时间会写完。所以，你**还不来**我的知识星球？