# 1 环境搭建到部署

**是一个控制台程序**

控制台程序（MVC代码 基于KESTREL 与网络监听，Kestrel 监听和数据交互）

不需要IIS

**添加中间件和服务实例**

|  |
| --- |
| public void ConfigureServices(IServiceCollection services)  {  services.AddSession(); //怎么用  }  public void Configure(IApplicationBuilder app, IWebHostEnvironment env)  {  App.UseSession(); //请求需要使用session  } |

**基于命令行托管**

Vs 控制台命令： dotnet 项目生成的dll --urls=”http://\*:5177” [--ip=”xxx” --port-5177]

dotnet D:\c#\WebApplication1\bin\Debug\netcoreapp3.1\WebApplication1.dll --urls=”http://\*:5177”

有时样式（在wwwroot文件夹下）会丢失，增加

|  |
| --- |
| app.UseStaticFiles(  new StaticFileOptions()  {  FileProvider = new PhysicalFileProvider(Path.Combine(Directory.GetCurrentDirectory(), @"wwwroot"))  }  ); |

**Kestrel**

精简高效的HttpServer,以包的形式提供，自身不能单独运行。内部封装对libuv的调用，作为I/O的底层，屏蔽各系统底层实现差异

做到真正的跨平台！

|  |
| --- |
| **public static IHostBuilder CreateHostBuilder(string[] args) =>**  **Host.CreateDefaultBuilder(args)**  **.ConfigureWebHostDefaults(webBuilder => //指定一个web服务器 --Kestrel**  **{**  **webBuilder.UseStartup<Startup>();**  **});** |

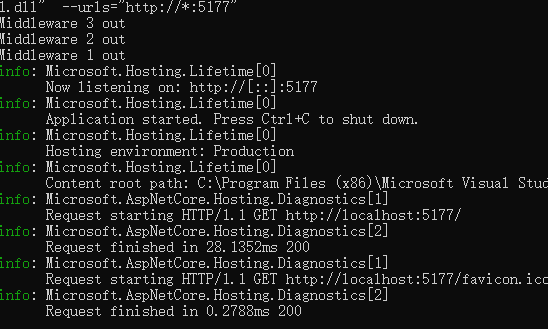
**标准中间件 （管道模型）**

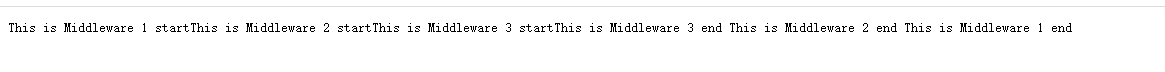
|  |
| --- |
| app.Use(next =>  {  return async c =>  {  await c.Response.WriteAsync("Hello word");  };  }); |

第一阶段是初始化，调用Buid 得到RequsetDelegate -middlewarel 管道，等待请求来

Context ---层层调用，然后就俄罗斯套娃（源码内是将这些中间件Resverse 倒叙）

|  |
| --- |
| //next 为下一个中间件app为参数，实现套娃  app.Use(  next => {  Console.WriteLine("Middleware 1 out");  return new RequestDelegate(  async context =>  {  await context.Response.WriteAsync("This is Middleware 1 start");  next.Invoke(context);  await context.Response.WriteAsync("This is Middleware 1 end ");  }  );  }  );  app.Use(  next => {  Console.WriteLine("Middleware 2 out");  return new RequestDelegate(  async context =>  {  await context.Response.WriteAsync("This is Middleware 2 start");  next.Invoke(context);  await context.Response.WriteAsync("This is Middleware 2 end ");  }  );  }  );  app.Use(  next => {  Console.WriteLine("Middleware 3 out");  return new RequestDelegate(  async context=>  {  await context.Response.WriteAsync("This is Middleware 3 start");  //next.Invoke(context); 源码内置返回404  await context.Response.WriteAsync("This is Middleware 3 end ");  }  );  }  ); |





**内置IOC**

控制反转，把对象的依赖换成对抽象的依赖

控制器里面都是对抽象的依赖，在程序开始时，对这个抽象进行具体化 services.AddTransient<ITestA, TestA>();

|  |
| --- |
| public class OneController : Controller  {  private readonly ITestA \_itestA;  public OneController(ITestA \_itestA)  {  this.\_itestA = \_itestA;  }  public IActionResult Index()  {  \_itestA.Show();  return View();  }  } |
| public void ConfigureServices(IServiceCollection services)  {  services.AddSession();  services.AddControllersWithViews();  services.AddTransient<ITestA, TestA>();  } |

作用：

1 可以去掉对细节的依赖。方便扩展，减小影响范围，甚至可以转移到配置文件的依赖

2 可以屏蔽细节，对象依赖注入（DI） 不在乎接口是怎么被构造的

DI: 构造对象，能自动把依赖的对象生成并传入支持递归无限级的技术手段叫依赖注入