.NET5 从零基础到精通

一共60个视频课；

主讲：Richard老师

机构：朝夕教育

环境要求

https://dotnet.microsoft.com/download/dotnet/5.0

1.Visual Studio2019 16.80版本

2..NET5运行时---发布部署

如果是2020年11月11日之后下载安装的时候，可以直接基于Visual Studio2019 16.80版本做开发；

环境准备

一、Vs安装

Vs下载地址：**https://visualstudio.microsoft.com/zh-hans/vs/**

1.运行在下的安装工具

2.在线安装，可能出现有网络问题的；

3.选择的安装板块：Web网站开发

建议：安装，建议安装在C盘；如果之前有过其他版本的Vs安装，不用卸载，二者可以并存；

C#6语法

1.只读自动属性

2.using static

3.Null 条件运算符

3.字符串内插

4.异常筛选器

5.nameof 表达式

6.使用索引器初始化关联集合

C#7语法

1.out 变量

2.元组

3.弃元

4.模式匹配

5.ref 局部变量和返回结果

6.本地函数

7.更多的 expression-bodied 成员

8.throw 表达式

9.通用的异步返回类型

10.数字文本语法改进

C#8语法

1.Readonly 成员 待处理

2.默认接口方法

3.模式匹配增强功能：

4.Using 声明

5.静态本地函数

6.可处置的 ref 结构

7.可为空引用类型

8.异步流

9.异步可释放

10.索引和范围

11.Null 合并赋值

12.非托管构造类型

13.嵌套表达式中的 14.Stackalloc

15.内插逐字字符串的增强功能

C#9语法

1.记录

2.仅限 Init 的资源库

3.顶级语句

4.模式匹配增强功能

5.本机大小的整数

6.函数指针

7.禁止发出 localsinit 标志

8.目标类型的新表达式

9.静态匿名函数

10.目标类型的条件表达式

11.协变返回类型

12.扩展 GetEnumerator 支持 foreach 循环

13.Lambda 弃元参数

14.本地函数的属性

15.模块初始值设定项

16.分部方法的新功能

项目建立

1.Net5项目建立

2.项目介绍

3.多种传值方式

Log4Net组件使用

1.Nuget引入程序集：log4Net + Microsoft.Extensions.Logging.Log4Net.AspNetCore

2.准备配置文件

3.配置使用Log4Net 记录日志

IIS安装

1.打开控制面板

2.点击程序

3.点击启动或关闭Windows功能

4.勾选Internet infomation Service下的所有选项全部划√

5.确认

运行发布

IIS发布

IIS直接建立网站，目录指向项目根目录下的Debug/Net5文件----不行

把项目发布以后：目录指向项目发布目录 ---可以的

脚本启动

dotnet AspNetCore.Web.dll --urls="https://*:8079" --ip="127.0.0.1" --port= 8079

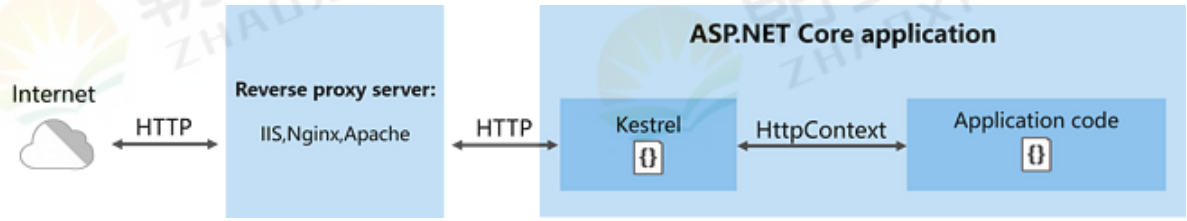
AspnetcoreModuleV2:安装：

下载路径

https://dotnet.microsoft.com/download/dotnet/5.0

下载安装：Hosting Bundle　需要根据自己电脑的系统位数来决定

ASPNET Core跨平台的原因：



静态文件读取+脚本参数

静态文件读取：

1.Nuget引入：Nuget引入Microsoft.Extensions.FileProviders　System.IO

2.配置读取静态文件的中间件

```
app.UseStaticFiles(new StaticFileOptions()
                {
                    FileProvider = new
PhysicalFileProvider(Path.Combine(Directory.GetCurrentDirectory(),"wwwroot"))  //执行文件下的wwwroot文件
夹
                });
```

脚本参数：

1.可以在启动dll的脚本之后放上参数，格式：--参数名称=参数值

2.控制器通过构造函数注入IConfiguration

3.可以在控制器中通过IConfiguration[参数名称]

配置文件读取：

第一种：

通过IConfiguration索引取值；

如果遇到对象，数组；

中间使用：分割，依次读取；

第二种：

1.定义一个和配置文件内容格式一致的类

2.在ConfigureServices 配置　services.Configure(Configuration.GetSection("ConnectionStrings"));

3.在使用的时候可以直接通过IOptions options注入

4.获取注入的Options.Value　就是我们获取的到配置文件的一个实体类型对象

MVC开发

1.什么是MVC?

V-视图------呈现给用户看到的内容（表现层）

C-控制器----控制业务逻辑计算，调用服务，选择返回什么内容，可以返回视图，JSON,字符串等等

M-视图模型---用作控制器和视图之间传递数据的载体

cshtml文件：其实是一个类文件；

2.修改视图，增加了Html代码，不能马上生效，那么办？必须要编译后方可生效的问题，如何解决?

```
Nuget引入：Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation
services.AddRazorPages().AddRazorRuntimeCompilation();
```

Razor混编：

可以在cshtml 上写上后台C#代码+前台html代码；混合起来

Razor布局

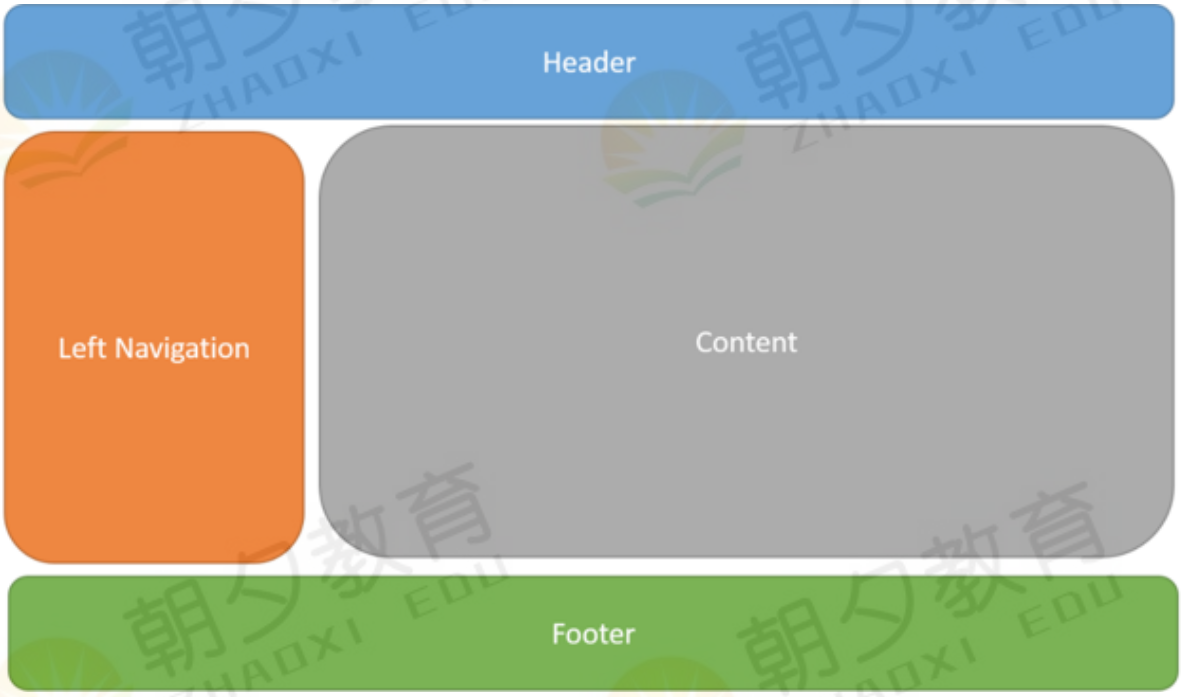
我们看到的页面组成到底有哪些内容?

包含了Layout的母版嵌套的返回的需要渲染的视图内容；

如何嵌套呢?

通过Layout中RenderBody()方法做了替换；把返回的视图替换到母版页中；形成了一整块的内容；

目的在于在每一次返回不同的页面的时候，能够把不变的部分视图，做以重用；这样就可以少写代码；



在母版页中使用css,js的时候，为了提高效率，会把css，引入在母版的上方，js引入在下方；

页面嵌套到母版页后，就会出现在引入js的上方调用js,js 其实是调用不了的；

解决方案：

1.在母版页中，标记 @await RenderSectionAsync("Scripts", required: false)

2.在返回页面中，标记@section Scripts{} 把js代码写到到 @section Scripts{ } 大括号中，就可以解决；

Razor扩展控件

第一种方式：

1.定义个静态类

2.定义静态扩展方法，扩展IHtmlHelper类型，返回IHtmlContent类型；

实例：扩展返回一个img html标签

本质：通过一个后台方法，返回一个**已经存在**的Html标签的字符串，浏览器在读取的时候，就读取成一个html标签；

```
public static IHtmlContent Img(this IHtmlHelper helper,string src,string @classs="")
{
        return new HtmlString($"<Img src='{src}' class='{@classs}'></Img>");
}
```

3.调用

引入扩展方法所在的命名空间

```
@Html.Img("扩展方法的参数")
```

第二种方式：

通过一个后台方法，返回一个**不存在**的Html标签的字符串，在读取的时候，通过后台方法，去渲染成我们需要的的标签和内容；

1.定义一个普通类，类名称建议以TagHelper结尾，建议给类标记特性[HtmlTargetElement（视图中在调用的时候使用的名称)],如果没有标记特性，视图中在调用的时候使用当前类名称去掉TagHelper后缀的到的字符串用来调用：

2.继承TagHelper抽象类、或者实现接口ITagHelper,二者均可；

3.覆写或者实现接口Process方法

4.实现方法，定义标签类型，指定属性；指定内容；

5.如果在调用的时候，需要传递参数，可以在定义这标签的类内部，定义属性；属性名称建议首字母大写； 在调用的时候保持和属性名称一致；

局部视图

1.可以增加代码的重用型

如何定义：

1.添加一个cshtml文件

2.在页面中调用局部视图：@html.Partial("局部视图的名称")

问题：局部是不没有去访问控制器中的任何Action；

视图组件

1.呈现页面响应的某一部分而不是整个响应

2.包括在控制器和视图之间发现的关注分离和可测试性优势

3.可以具有参数和业务逻辑

4.通常在页面布局中调用

如何自定义视图组件?

1.Razor组件对应的类需要以ViewComponent结尾

2.继承ViewComponent类

3.定义一个异步版本的InvokeAsync 可以自定义参数

4.InvokeAsync就是返回组件内容的方法

5.在Views文件夹下的Shard文件夹下建立Components 文件夹，再Conponents下创建一个和组件类名称相同的文件夹

6.创建视图

7.@await Component.InvokeAsync("List", new { maxPriority = 123, isDone = false })调用组件

IOC容器IServiceCollection

.NET5是内置IOC容器的；

什么IOC?

把对象的创建统一交给第三方容器来创见；

如何使用内置IOCIServiceCollection:

第一种方式：

1.在Startup中的ConfigureServices 方法中注册服务

```
services.AddTransient<ITestServiceA, TestServiceA>();
```

2.在需要使用的控制器中，通过构造函数，定义服务的抽象类型，作为参数，在运行时，自动得到服务的具体；

```
private readonly ITestServiceA _ITestServicA = null;
public FifthController(ITestServiceA iTestServicA)
{
                _ITestServicA = iTestServicA;
}
```

3.调用服务内部的方法

```
public IActionResult Index()
{
    _ITestServicA.Show();
    return View();
}
```

第二种方式：

1.在Startup中的ConfigureServices 方法中注册服务

```
services.AddTransient<ITestServiceA, TestServiceA>();
```

2.在需要使用的控制器中，通过构造函数，注入IServiceProvider

```
private readonly IServiceProvider _ServiceProvider = null;
public FifthController(IServiceProvider serviceProvider)
{
  _ServiceProvider=serviceProvider
}
```

3.通过_ServiceProvider获取到服务，然后通过服务实例调用服务内部的方法

```
public IActionResult Index()
{
    ITestServiceA testServiceA = (ITestServiceA)_ServiceProvider.GetService(typeof(ITestServiceA));
    testServiceA.Show();
}
```

第三种方式;

在注册服务后，视图中通过关键字@Inject 获取实例

```
@inject ITestServiceA iTestServicA     ---获取到服务实例
@{
     iTestServicA.Show();
}
```

DI依赖注入：IServiceCollection支持且仅支持构造函数注入

什么是依赖注入呢?

如果对象A依赖于对象B，对象B依赖于对象C，就可以先构造对象C,然后传递给对象B，然后把对象B传递给对象A，得到A的具体实例;

IServiceCollection可以支持无线层级的以来注入； 前提是都要先注入服务（注册抽象和具体的映射关系）

IServiceCollection生命周期

那么在创建对象的时候，不同的情况，需要让对象单利，每一次都创建新的对象实例；不同的作用于创建新的实例；

瞬时生命周期：每一次getService获取的实例都是不同的实例

单例生命周期，在整个进程中获取的都是同一个实例

作用域生命周期；同一个作用域，获取的是同一个对象的实例；不同的作用域，获取的是不同的对象实例

建议：开发工作中，一般情况下，都是一起请求一个对象的实例；更多的是瞬时生命周期的使用；

Autofa容器

初识：

Autofa也是一款很流行的IOC容器：那如何使用？ 第三方的IOC容器；

1.Nuget引入程序包

2.创建一个ContainerBuilder

3.注册抽象和实现关系

4.Build一下，得到IContainer容器

5.通过容器获取服务实例

6.使用服务

Autoafa多种注入方式：

1.构造函数注入

```
ContainerBuilder containerBuilder = new ContainerBuilder();
containerBuilder.RegisterType<TestServiceA>().As<ITestServiceA>();
containerBuilder.RegisterType<TestServiceB>().As<ITestServiceB>();
containerBuilder.RegisterType<TestServiceC>().As<ITestServiceC>();
IContainer container = containerBuilder.Build();
ITestServiceC testServiceC = container.Resolve<ITestServiceC>();//获取服务
testServiceC.Show();
```

2.属性注入

```
ContainerBuilder containerBuilder = new ContainerBuilder();
containerBuilder.RegisterType<TestServiceA>().As<ITestServiceA>();
containerBuilder.RegisterType<TestServiceB>().As<ITestServiceB>();
containerBuilder.RegisterType<TestServiceC>().As<ITestServiceC>();
containerBuilder.RegisterType<TestServiceD>().As<ITestServiceD>().PropertiesAutowired();
IContainer container = containerBuilder.Build();
ITestServiceD testServiceD = container.Resolve<ITestServiceD>();//获取服务
testServiceD.Show();
```

3.方法注入

```
ContainerBuilder containerBuilder = new ContainerBuilder();
containerBuilder.RegisterType<TestServiceA>().As<ITestServiceA>();
containerBuilder.RegisterType<TestServiceC>().As<ITestServiceC>();
containerBuilder.RegisterType<TestServiceD>().As<ITestServiceD>().PropertiesAutowired();
containerBuilder.RegisterType<TestServiceB>().OnActivated(e =>
e.Instance.SetService(e.Context.Resolve<ITestServiceA>())).As<ITestServiceB>();
IContainer container = containerBuilder.Build();
ITestServiceB testServiceB = container.Resolve<ITestServiceB>();//获取服务
testServiceB.Show();
```

Autofac生命周期

瞬时生命周期--每次获取实例都是全新的实例

```
ContainerBuilder containerBuilder = new ContainerBuilder();
containerBuilder.RegisterType<TestServiceA>().As<ITestServiceA>().InstancePerDependency();
IContainer container = containerBuilder.Build();
ITestServiceA testServiceA = container.Resolve<ITestServiceA>();//获取服务
ITestServiceA testServiceA1 = container.Resolve<ITestServiceA>();//获取服务
Console.WriteLine(object.ReferenceEquals(testServiceA, testServiceA1));
```

单例生命周期 --一个进程中没有都是同一个实例

```
ContainerBuilder containerBuilder = new ContainerBuilder();
containerBuilder.RegisterType<TestServiceA>().As<ITestServiceA>().SingleInstance();
IContainer container = containerBuilder.Build();
ITestServiceA testServiceA = container.Resolve<ITestServiceA>();//获取服务
ITestServiceA testServiceA1 = container.Resolve<ITestServiceA>();//获取服务
Console.WriteLine(object.ReferenceEquals(testServiceA, testServiceA1));
```

每个生命周期范围一个实例

```
ContainerBuilder containerBuilder = new ContainerBuilder();
containerBuilder.RegisterType<TestServiceA>().As<ITestServiceA>().InstancePerLifetimeScope();
IContainer container = containerBuilder.Build();

ITestServiceA testServiceA15 = null;
ITestServiceA testServiceA16 = null;
using (var scope1 = container.BeginLifetimeScope())
{
        ITestServiceA testServiceA11 = scope1.Resolve<ITestServiceA>();
        ITestServiceA testServiceA12 = scope1.Resolve<ITestServiceA>();
        Console.WriteLine(object.ReferenceEquals(testServiceA11, testServiceA12));
        testServiceA15 = testServiceA12;
}

using (var scope1 = container.BeginLifetimeScope())
{
        ITestServiceA testServiceA13 = scope1.Resolve<ITestServiceA>();
        ITestServiceA testServiceA14 = scope1.Resolve<ITestServiceA>();
        Console.WriteLine(object.ReferenceEquals(testServiceA13, testServiceA14));
        testServiceA16 = testServiceA14;
}
Console.WriteLine(object.ReferenceEquals(testServiceA15, testServiceA16));
```

每个【匹配生命周期范围一个实例

```
ContainerBuilder containerBuilder = new ContainerBuilder();
```

```
containerBuilder.RegisterType<TestServiceA>().As<ITestServiceA>
().InstancePerMatchingLifetimeScope("Zhaoxi");
IContainer container = containerBuilder.Build();

ITestServiceA testServiceA15 = null;
ITestServiceA testServiceA16 = null;
using (var scope1 = container.BeginLifetimeScope("Zhaoxi"))
{
        ITestServiceA testServiceA11 = scope1.Resolve<ITestServiceA>();
        using (var scope2 = scope1.BeginLifetimeScope())
        {
                ITestServiceA testServiceA12 = scope2.Resolve<ITestServiceA>();
                Console.WriteLine(object.ReferenceEquals(testServiceA11, testServiceA12));
        }
        testServiceA15 = testServiceA11;
}

using (var scope1 = container.BeginLifetimeScope("Zhaoxi"))
{
          ITestServiceA testServiceA13 = scope1.Resolve<ITestServiceA>();
          using (var scope2 = scope1.BeginLifetimeScope())
          {
                ITestServiceA testServiceA14 = scope2.Resolve<ITestServiceA>();
                Console.WriteLine(object.ReferenceEquals(testServiceA13, testServiceA14));
          }

        testServiceA16 = testServiceA13;
}
       Console.WriteLine(object.ReferenceEquals(testServiceA15, testServiceA16));
```

每个请求一个实例(InstancePerRequest)

```
ContainerBuilder containerBuilder = new ContainerBuilder();
containerBuilder.RegisterType<TestServiceA>().As<ITestServiceA>().InstancePerRequest();
IContainer container = containerBuilder.Build();
using (var scope1 = container.BeginLifetimeScope())
{
        ITestServiceA testServiceA11 = scope1.Resolve<ITestServiceA>();
        ITestServiceA testServiceA12 = scope1.Resolve<ITestServiceA>();
        Console.WriteLine(object.ReferenceEquals(testServiceA11, testServiceA12));
}
```

Autofac支持配置文件

1.Nuget引入程序集：Autofac.Extensions.DependencyInjection

Autofac.Configuration

Autofac

2.准备配置文件

3.读取配置文件，根据配置文件信息，生成抽象和映射信息

Autofac整合.NET5 MVC

Autofac是一个第三方容器；

1.指定Autofac工厂替换默认工厂,Program指定

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
                Host.CreateDefaultBuilder(args)
                        .ConfigureWebHostDefaults(webBuilder =>
                        {
                              webBuilder.UseStartup<Startup>();
                        })
                        .UseServiceProviderFactory(new AutofacServiceProviderFactory());
```

2.在Startup类增加ConfigureContainer 方法， **注册关系**

```
public void ConfigureContainer(ContainerBuilder builder)
          {
                builder.RegisterType<TestServiceA>().As<ITestServiceA>();
                builder.RegisterType<TestServiceUpdate>().As<ITestServiceA>();
          }
```

3.通过控制器构造函数注入，获取实例

Autofac支持控制器属性注入

控制器是一个类，控制器的实例其实是IControllerActivator来创建的；

1.得让控制器使用容器来获取实例；

2.注册控制器抽象和具体的关系

```
var controllersTypesInAssembly = typeof(Startup).Assembly.GetExportedTypes()
                  .Where(type => typeof(ControllerBase).IsAssignableFrom(type)).ToArray();

builder.RegisterTypes(controllersTypesInAssembly).PropertiesAutowired(new AutowiredPropertySelector());
```

3.在控制器内定义属性

4.扩展，自己控制究竟哪些属性需要做依赖注入

Autofac抽象多实现的问题

1.一个抽象多个实例，都注册了，通过构造函数用抽象类型来获取实例，哪个后面注册就获取到哪一个；覆盖型；

2.一个抽象多个实例，都注册了,可以通过一个IEnumerable<抽象>,当做构造函数参数，可以获取到所有注册的具体的实例；

3.注册一个抽象的多个实例资源，如下方式注册，可以在控制器的构造函数中，使用具体实现类型作为参数类型，可以匹配到不同到具体类型实例

```
ontainerBuilder.RegisterSource(new AnyConcreteTypeNotAlreadyRegisteredSource(t =>
t.IsAssignableTo<ITestServiceA>()));
```

Autofac支持AOP-1

AOP面向切面编程；不用修改之前代码的基础上，可以动态的在某个动作之前加一些操作，动态在在某一个动作之后做点什么事儿

1.Nuget引入Castle.Core程序集+Autofac.Extras.DynamicProxy程序集

2.定注意切入者：如下，需要继承IInterceptor，实现方法

```
public class CustomAutofacAop : IInterceptor
    {
          public void Intercept(IInvocation invocation)
          {
                {
                      Console.WriteLine("方法执行前");
                }
                invocation.Proceed();//执行这句话就是去执行具体的实例的这个方法

                {
                      Console.WriteLine("方法执行后");
                }
          }
    }
```

2.在服务的抽象上标记[Intercept(typeof(CustomAutofacAop))]

3.注册支持AOP扩展的类

```
ontainerBuilder.RegisterType(typeof(CustomAutofacAop));
```

4.注册服务的时候，需要需要调用EnableInterfaceInterceptors，标记说明当前服务获取实例后可以支持AOP

Autofac支持AOP-2

EnableInterfaceInterceptors+抽象标记特性[Intercept(typeof(CustomAutofacAop))]，只要是实现了这个抽象就可以支持AOP

EnableClassInterceptors ＋ 实现类标记特性[Intercept(typeof(CustomAutofacAop))]，只有标记了这个特性的，才能够支持AOP

如果使用EnableClassInterceptors 来支持AOP，实现类中支持AOP的方法必须为虚方法；

```
    [Intercept(typeof(CustomAutofacAop))]
     public class TestServiceA : ITestServiceA
     {

          public TestServiceA()
          {
                Console.WriteLine($"{this.GetType().Name}被构造。。。");
          }

          public virtual void Show()
          {
                Console.WriteLine("A123456");
          }
    }
```

Autofac单抽象多实现构造函数注入

1.注册的时候可以指定一个标识

```
containerBuilder.RegisterType<TestServiceA>().Named<ITestServiceA>("TestServiceA");
containerBuilder.RegisterType<TestServiceUpdate>().Named<ITestServiceA>("TestServiceUpdate");
```

2.在控制器中获取的时候获取一个Autofac的上下文，通过上下文+标识，得到不同的实现的实例；

```
private readonly IComponentContext _ComponentContext = null;

        public SixThController(IComponentContext componentContext)
        {
            _ComponentContext = componentContext;
        }

        public IActionResult Index()
        {

            ITestServiceA testServiceA = _ComponentContext.ResolveNamed<ITestServiceA>
("TestServiceA");
            ITestServiceA testServiceUpdate = _ComponentContext.ResolveNamed<ITestServiceA>
("TestServiceUpdate");
            iTestServiceAA.Show();
            return View();
        }
```

Autofac单抽象多实例属性注入

1.道理和上面一样，可以把Autofac 上下文当做属性来注入

2.在使用的时候，通过Autofac上下文+不同注册的标识，获取到不同的具体实例

.NET5 AOP 5个Filter

AOP：可以在不修改之前的代码为基础，可以动态的增加新功能；

AuthorizationFilter 鉴权授权

ResourceFilter 资源

ExceptionFilter 异常

ActionFilter 方法

ResultFilter 结果

ActionFilter的特点

自定义一个CustomActionFilterAttribute特性，继承Attribute，实现IActionFilter接口；实现方法，标记在Action上；

请求标记的有CustomActionFilterAttribute 的Action:执行顺序如下：

1.执行控制器构造函数

2.执行CustomActionFilterAttribute 内的OnActionExecuting方法

3.执行Action

4.执行CustomActionFilterAttribute 内的OnActionExecuted

ActionFilter的多种使用

1.通过实现IActionFilter接口来完成扩展

2.通过继承ActionFilterAttribute（系统提供的实现），根据自己的需要，覆写不同的方法，达到自己的诉求

3.异步版本的实现，通过实现IAsyncActionFilter接口来实现

ActionFilter应用+依赖注入

1.记录日志，action比较靠近Action，日志记录，可以记录到Action内部做的一些处理

2.通过Log4net完成日志记录，需简要注入Logger

Filter的多种注册

1.[CustomActionFilter]---Fitler必须有无参数构造函数

2.[TypeFilter(typeof(CustomActionFilterAttribute))]，可以没有无参数构造函数，可以支持依赖注入

3.[ServiceFilter(typeof(CustomActionFilterAttribute))]，可以没有无参数构造函数，可以支持依赖注入，但是必须要注册服务

以来注入是如何支持的？其实是是IOC来做到的。。。如何自定义。。。

FilterFactory扩展定制

为可以使用ServiceFilter/TypeFilter就可以支持依赖注入呢？----一定是IOC容器来完成；

1.自定义一个特性类，继承Attribute,实现接口 IFilterFactory；实现接口中的方法

2.通过构造函数传递需要实例化的特性的type类型

3.在实现接口中，通过Type类型获取到实例

4.标记在Action上面

Filter的生效范围和执行顺序

1.标记在Action上，就只对当前Action生效

2.标记在Controller上，就对Controller上中的所有Action生效

3.全局注册，对于当前整个项目中的Action都生效，在ConfigureServices中增加以下代码即可

```
services.AddMvc(option=> {
                    option.Filters.Add<CustomActionFilterAttribute>(); //全局注册：
                });
```

如果有三个actionFilter，分别注册全局，控制器、Action;执行顺序如何呢?

定义三个ActionFilter，分别注册全局，控制器，Action

执行顺序如下：

1.控制器实例化

2.全局注册的Filter-OnActionExecuting

3.控制器注册的Filter-OnActionExecuting

4.Actioin注册的Filter-OnActionExecuting

5.执行Action内部的逻辑算计

6.Action注册的Filter-OnActionExecuted

7.控制器注册的Filter-OnActionExecuted

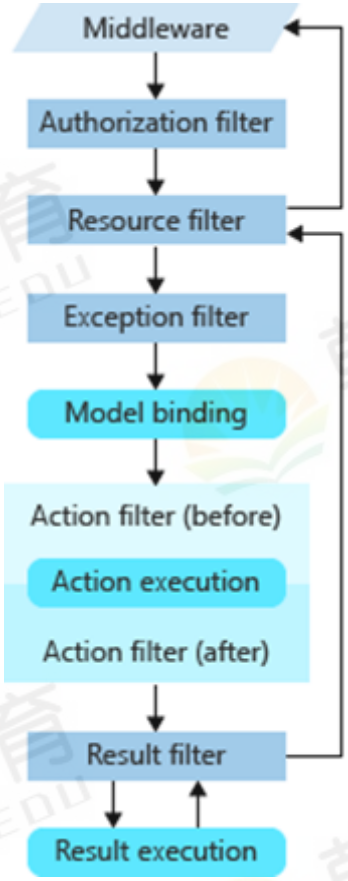
8.全局注册的Filter-OnActionExecuted

俄罗斯套娃~

执行顺序不能改吗？ 能改，在注册Filter的时候，指定Order的值，执行顺序，按照Order的值，从小到大执行；

ResourceFilter

就是为了缓存而存在。。。



Filter匿名

如果全局注册，Filter生效于所有的Acion；如果有部分Action我希望你不生效呢？怎么办呢？

匿名----可以避开Filter的检查；

如何支持自定义的Filter,匿名

1.自定义一个特性；

2.在需要匿名的Filter内部，检查是否需要匿名（检查是否标记的有匿名特性）,如果有就直接避开

ExceptionFilter

1.自定义一个CustomExceptionFilterAttribute ,实现IExceptionFilter接口，

2.实现方法，先判断，异常是否被处理过，如果没有被处理过，就处理；

3.分情况处理：1.如果是ajax请求，就返回JosnResult，如果不是Ajax请求，就返回错误页面

```
public class CustomExceptionFilterAttribute : Attribute, IExceptionFilter
    {
        private IModelMetadataProvider _modelMetadataProvider =null;

        public CustomExceptionFilterAttribute(IModelMetadataProvider modelMetadataProvider)
        {
            _modelMetadataProvider = modelMetadataProvider;
        }

        /// <summary>
        /// 当异常发生的时候触发到这儿来
        /// </summary>
        /// <param name="context"></param>
        public void OnException(ExceptionContext context)
        {
            if (!context.ExceptionHandled) //异常是否被处理过
            {
                //在这里处理  如果是Ajax请求===返回Json
                if (this.IsAjaxRequest(context.HttpContext.Request))//header看看是不是
XMLHttpRequest
                {
                    context.Result = new JsonResult(new
                    {
                        Result = false,
                        Msg = context.Exception.Message
                    });//中断式---请求到这里结束了，不再继续Action
                }
                else
                {
                    //跳转到异常页面
                    var result = new ViewResult { ViewName = "~/Views/Shared/Error.cshtml" };
                    result.ViewData = new ViewDataDictionary(_modelMetadataProvider,
context.ModelState);
                    result.ViewData.Add("Exception", context.Exception);

                    context.Result = result; //断路器---只要对Result赋值--就不继续往后了；
                }
                context.ExceptionHandled = true;
            }
        }

        private bool IsAjaxRequest(HttpRequest request)
        {
            string header = request.Headers["X-Requested-With"];
            return "XMLHttpRequest".Equals(header);
        }
    }
```

4.全局注册，在Starup中的ConfigureServices注册

```
services.AddMvc(option=> {
                    //option.Filters.Add<CustomActionFilterAttribute>(); //全局注册：
                    //option.Filters.Add<CustomGlobalActionFilterAtrribute>();

                    option.Filters.Add<CustomExceptionFilterAttribute>();

                });
```

ExceptionFilter能捕捉到哪些异常

1.控制器实例化异常　　----T

2.异常发生在Try-cache中　　---F

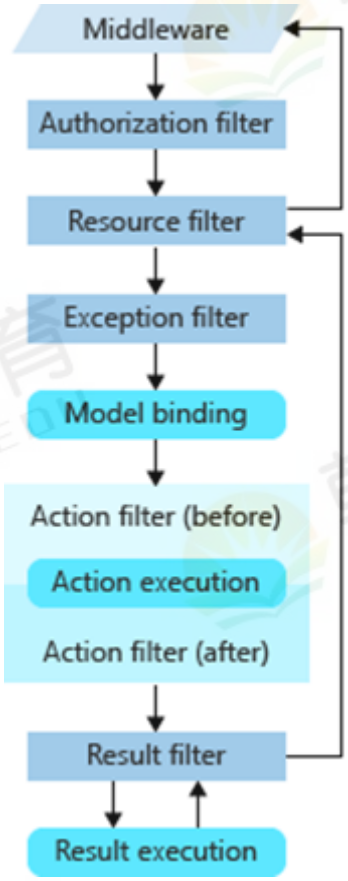
3.在视图中发生异常　　----F

4.Service层发生异常　　---T

5.在action中发生异常　　---T

6.请求错误路径异常　　---可以使用中间件来支持，只要不是200的状态，就都可以处理；

ResultFilter



1.自定义一个类，继承Attribute,实现IResultFilter接口，实现方法

2.标记在Action方法头上

3.执行顺序：视图执行前，渲染视图，视图执行后

ResultFilter的应用

双语言系统，其实就需要两个视图；要根据语言的不同，来选择不同的视图来渲染；

因为在渲染视图之前，会进入到OnResultExecuting方法，就可以在这个方法中确定究竟使用哪一个视图文件；

鉴权授权

为了拦截一些操作；

传统的授权方式；session/Cookies来完成；

1.在请求某个Action之前去做校验，验证当前操作者是否登录过，登录过就有权限

2.如果没有权限就跳转到登录页中去

AOP--Filter； ActionFilter:

传统的登陆，需要匿名；

.NET5鉴权授权

通过中间件来支持；

第一步：

在 app.UseRouting();之后，在app.UseEndpoints() 之前，增加鉴权授权；

鉴权： app.UseAuthentication();---检测用户是否登录

授权：app.UseAuthorization();//授权 检测有没有权限，是否能够访问后续的页面功能

第二步: 在ConfigureServices中中增加

```
 services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
             .AddCookie(options =>
             {
                 options.LoginPath = new PathString("/Eighth/Login"); //如果授权失败，就跳转到这个路径去中

//options.AccessDeniedPath = new PathString("/Home/Privacy");
             });//用cookie
```

第三步：指定哪些Action需要做鉴权授权

标记特性：标记在哪个Action上，哪个Action就能够支持鉴权授权

```
[Microsoft.AspNetCore.Authorization.Authorize]
```

也可以标记在控制器，全局；

.NET5鉴权授权角色授权

不同的用户，可能会存在不同的角色，不同的角色，可能在访问不同的页面的时候，需要做不同拦截；----角色授权其实就是通过角色不同，做不同的权限拦截；

保证上个视频的内容不变

```
var rolelist = new List<string>() {  //来自于数据库的角色
                "Admin",
                "Teacher",
                "Student"
            };
            //ClaimTypes.Role就是做权限认证的标识;
            var claims = new List<Claim>()//鉴别你是谁，相关信息
            {
                  new Claim(ClaimTypes.Role,"Admin"),
                  new Claim(ClaimTypes.Name,name),
                  new Claim("password",password),//可以写入任意数据
                  new Claim("Account","Administrator"),
                  new Claim("role","admin"),
                  new Claim("zhaoxi","zhaoxi"),
                  new Claim("User","zhaoxi")
            };
            foreach (var role in rolelist)
            {
                claims.Add(new Claim(ClaimTypes.Role, role));
            }
```

```
[Authorize(Roles = "Admin,Teacher,Student")] //标记的试试，通过逗号分隔不同的角色---只要是有一个角色符合就能够访问，角色之间是或者的关系

[Authorize(Roles = "Admin")]
[Authorize(Roles = "Teacher")]
[Authorize(Roles = "Student")] 多个特性标记--多个角色之前是且的关系，必须要包含所有的角色，才能够访问
```

.NET5鉴权授权策略鉴权

之前的角色授权是在代码中把角色定义死了；更希望能够自己来完成校验逻辑；

第一步：

增加CustomAuthorizationHandler-----专用做检验逻辑的； 要求继承自AuthorizationHandler<>泛型抽象类；

增加一个CustomAuthorizationRequirement，要求实现接口：IAuthorizationRequirement

第二步：让自定义逻辑生效：在ConfigureServices方法中注册进来

```
services.AddSingleton<IAuthorizationHandler, CustomAuthorizationHandler>();
```

第三步：支持多种策略

例：

```
services.AddAuthorization(options =>
            {
                options.AddPolicy("customPolicy", polic =>
                {
                    polic.AddRequirements(new CustomAuthorizationRequirement("Policy01"));
                });
            });
```

.NET5跨平台本质

1.Web应用程序是一个控制台；---Main程序的入口；跨平台的原因在于框架已经内置了主机，只要是程序启动，就是启动了主机，就可以监听端口；请求来了，只要是请求这个端口，主机就可以响应；无论是在Windows还是在Linux下开发，都是要第一步完成环境的；这样就不用拘泥于是在Windows还是在linux，----这就是能够跨平台的本质原因；

.NET5中间件

中间件的执行，是一个俄罗斯套娃；

1.先use 先执行 ，每一次Use一个中间件，其实就是在之前的基础上，套了一层；

2.请求来了以后，真正执行的时候，是一层一层的内部执行，然后再执行出来；----俄罗斯套哇

增加程序的扩展性---如果想要增加一层，直接增加一个中间件就可以来完成；

.NET5常用中间件

1.app.Run，中断式，只要是使用当前中间件，后面的中间件都不执行；

2.app.Map，判断路径中是否包含什么内容；

3.app.MapWhen判断式，两个委托，第一个委托作为判断条件内容，第二个委托，是要执行的逻辑；

中间件的多种引用方式

1.中间件的内容可以独立开，放入到一个独立的类中去---需要一个规则

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

        public FirstMiddleWare(RequestDelegate next)
        {
            _next = next;
        }
        public async Task Invoke(HttpContext context)
        {
            await context.Response.WriteAsync($"<p>{nameof(FirstMiddleWare)},Hello World1!</p>");
            await _next(context);
            await context.Response.WriteAsync($"<p>{nameof(FirstMiddleWare)},Hello World1!</p>");
        }
    }
```

要求构造函数带有RequestDelegate参数类型---为了得到下一个中间件； 必须包含async Task Invoke方法，方法参数为HttpContext

2.在Starup中把这些中间件给引入进来

```
app.UseMiddleware<FirstMiddleWare>(); //使用某一个中间件
app.UseMiddleware<SecondMiddleWare>();
app.UseMiddleware<ThirdMiddleWare>();
```

EFCore

EF: 通过实体和数据库的映射，可以通过对实体的操作完成对数据的操作；以面向对象的思想来完成对数据的管理；

映射：1.从数据库到代码实体的映射　2.从代码到数据库的映射

数据库：ZhaoxiEduDataBase

EFCoreDBFirst

先有数据库，然后通过映射得到实体（和数据库的表对应）；

```
Install-Package Microsoft.EntityFrameworkCore
Install-Package Microsoft.EntityFrameworkCore.SqlServer
Install-Package Microsoft.EntityFrameworkCore.Tools

工具--nuget包管理器--程序包管理器控制台：命令执行：
Scaffold-DbContext "Data Source=DESKTOP-63QE7M1;Initial Catalog=ZhaoxiEduDataBase;User
ID=sa;Password=sa123" Microsoft.EntityFrameworkCore.SqlServer  -OutputDir  Entity -Force -Context
ZhaoxiDbContext  -ContextDir /

命令参数：
-OutputDir *** 实体文件所存放的文件目录
-ContextDir *** DbContext文件存放的目录
-Context *** DbContext文件名
-Schemas *** 需要生成实体数据的数据表所在的模式
-Tables *** 需要生成实体数据的数据表的集合
-DataAnnotations
-UseDatabaseNames 直接使用数据库中的表名和列名（某些版本不支持）
-Force 强制执行，重写已经存在的实体文件
```

EFCoreCodeFirst

(迁移)：代码先行，现有代码再有数据库；数据库随着业务变化迁移改变；

1.可以通过EfCore带有的api来完成从代码生成数据库

2.迁移

迁移命令：

迁移命令描述	CLI命令	PMC命令
创建迁移：migrationname 为迁移名称	dotnet ef migrations add migrationName	add-migration migrationName
移除迁移(删除最近的一次迁移)	dotnet ef migrations remove	remove-migration
应用所有的迁移(使迁移文件应用到数据库)	dotnet ef database update	update-database
指定版本进行迁移	dotnet ef database update migrationName	update-database migrationName
生成对应版本的脚本	dotnet ef migrations script	Script-Migration
查看迁移列表	dotnet ef migrations list	
查看数据库上下文信息	dotnet ef dbcontext info	

EFCore抓取Sql语句

一、日志输出

1.Nuget引入：

```
Microsoft.Extensions.Logging.Console
```

2.配置使用日志输出Sql语句

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            ///通过日志输出Sql语句
            optionsBuilder.UseLoggerFactory(LoggerFactory.Create(builder =>
            {
                builder.AddConsole();
            }));

        }
```

二、Profiler工具

EFCore支持Linq

```
 public static void Show()
        {
            #region 其他查询
            using (ZhaoxiDbContext dbContext = new ZhaoxiDbContext())
            {
                {
                    var idlist = new int[] { 1, 2, 3, 5, 7, 8, 9, 10, 11, 12, 14, 17 };//in查询
                    var list = dbContext.SysUsers.Where(u => idlist.Contains(u.Id));//in查询
                    foreach (var user in list)
                    {
                        Console.WriteLine(user.Name);
                    }
                }
```

```
                {
                    //没有任何差别，只有写法上的熟悉
                    var list = from u in dbContext.SysUsers
                               where new int[] { 1, 2, 3, 5, 7, 8, 9, 10, 11, 12, 14
}.Contains(u.Id)
                               select u;

                    foreach (var user in list)
                    {
                        Console.WriteLine(user.Name);
                    }
                }
                {
                    var list = dbContext.SysUsers.Where(u => new int[] { 1, 2, 3, 5, 7, 8, 9,
10, 11, 12, 14, 18, 19, 20, 21, 22, 23 }.Contains(u.Id))
                                                 .OrderBy(u => u.Id)
                                                 .Select(u => new
                                                 {
                                                     Name = u.Name,
                                                     Pwd = u.Password
                                                 }).Skip(3).Take(5);
                    foreach (var user in list)
                    {
                        Console.WriteLine(user.Pwd);
                    }
                }
                {
                    var list = (from u in dbContext.SysUsers
                                where new int[] { 1, 2, 3, 5, 7, 8, 9, 10, 11, 12, 14
}.Contains(u.Id)
                                orderby u.Id
                                select new
                                {
                                    Name = u.Name,
                                    Pwd = u.Password
                                }).Skip(3).Take(5);

                    foreach (var user in list)
                    {
                        Console.WriteLine(user.Name);
                    }
                }

                {
                    var list = dbContext.SysUsers.Where(u => u.Name.StartsWith("小")
                                                        && u.Name.EndsWith("村长"))
                                                 .Where(u => u.Name.EndsWith("长"))
                                                 .Where(u => u.Name.Contains("名
村"))
                                                 .Where(u => u.Name.Length < 5)
                                                 .OrderBy(u => u.Id);

                    foreach (var user in list)
                    {
                        Console.WriteLine(user.Name);
                    }

                    var list1 = from u in dbContext.SysUsers
```

```
                                        where u.Name.StartsWith("小")
                                          && u.Name.EndsWith("村长")
                                        where u.Name.EndsWith("长")
                                        select new { Name = u.Name, pwd = u.Password };

                        foreach (var user in list1)
                        {
                              Console.WriteLine(user.Name);
                        }
                  }
                  {

                        //dbContext.SysUsers.Join()
                        var list = (from u in dbContext.SysUsers
                                        join c in dbContext.SysUserRoleMappings on u.Id equals
c.SysUserId //条件不能写等号，要使用equals关键字
                                        where new int[] { 1, 2, 3, 4, 6, 7, 10 }.Contains(u.Id)
                                        select new
                                        {
                                              Name = u.Name,
                                              Pwd = u.Password,
                                              RoleId = c.SysRoleId.ToString(),
                                              UserId = u.Id
                                        }).OrderBy(u => u.UserId).Skip(3).Take(5);

                        foreach (var user in list)
                        {
                              Console.WriteLine("{0} {1}", user.Name, user.Pwd);
                        }
                  }
                  {
                        Console.WriteLine("**********************Linq左连接
**************************");
                        //Linq左连接：
                        {
                              //Linq中只有左连接
                              {
                                    var list = from u in dbContext.SysUsers
                                                join c in dbContext.SysUserRoleMappings on
u.Id equals c.SysUserId
                                                into ucList
                                                from uc in ucList.DefaultIfEmpty()
                                                where new int[] { 1, 2, 3, 4, 6, 7, 10
}.Contains(u.Id)
                                                select new
                                                {
                                                      Account = u.Name,
                                                      Pwd = u.Password,
                                                      UserId = u.Id
                                                };
                                    foreach (var user in list)
                                    {
                                          Console.WriteLine("{0} {1}", user.Account, user.Pwd);
                                    }
                              }
                              {

                                    var list = from c in dbContext.SysUserRoleMappings
```

```
                                                join u in dbContext.SysUsers on c.SysUserId
equals u.Id
                                                into ucList
                                                from uc in ucList.DefaultIfEmpty()
                                                where new int[] { 1, 2, 3, 4, 6, 7, 10
}.Contains(c.Id)
                                                select new
                                                {
                                                      Account = uc.Name,
                                                      Pwd = uc.Password,
                                                      Id = uc.Id.ToString()
                                                };
                                    foreach (var user in list)
                                    {
                                          Console.WriteLine("{0} {1}", user.Account, user.Pwd);
                                    }

                              }
                        }

                  }
            }

            //如果遇到非常复杂的查询----建议直接写Sql语句;
            using (ZhaoxiDbContext dbContext = new ZhaoxiDbContext())
            {
                  {
                        try
                        {
                              string selectSql = "select * from  JD_Commodity_001 where id<@Id";
                              SqlParameter parameter1 = new SqlParameter("@Id", 500);
                              var query = dbContext.JdCommodity001s.FromSqlRaw<JdCommodity001>
(selectSql, parameter1);

                              foreach (var jd in query)
                              {
                                    Console.WriteLine(jd.Title);
                              }
                        }
                        catch (Exception ex)
                        {
                              Console.WriteLine(ex.Message);
                        }

                  }
                  {
                        string sql = "Update [JD_Commodity_001] Set Title='小新' WHERE Id=@Id";
                        SqlParameter parameter1 = new SqlParameter("@Id", 500);
                        int flg = dbContext.Database.ExecuteSqlRaw(sql, parameter1);
                  }
            }
            #endregion
      }
```

EFCoreState

增删改的动作是统一又SaveChanges以后才落实到数据库中去；

数据库的增改动作都是统一由SaveChanges之后，统一提交到数据库；是通过状态跟踪，任何一个增删改查的操作都会记录一个状态在内存中；增删改查的状态；一旦SaveChanges，就根据状态去落实到数据中去的；

状态跟踪实现了增删改便捷，但是也会有性能损耗；

```
        //
        // 摘要:
        //     上下文未跟踪实体。
        Detached = 0,
        //
        // 摘要:
        //     实体正在被上下文跟踪，并且存在于数据库中。它的财产
        //     数据库中的值没有更改
        Unchanged = 1,
        //
        // 摘要:
        //     实体正在被上下文跟踪，并且存在于数据库中。是的
        //     已标记为从数据库中删除。
        Deleted = 2,
        //
        // 摘要:
        //     实体被上下文跟踪并存在于数据库中。一些或
        //     它的所有属性值都已修改。
        Modified = 3,
        //
        // 摘要:
        //     实体正在被上下文跟踪，但在数据库中尚不存在。
        Added = 4
```

EFCore事务

SaveChange就是保证事务的；多个对于数据库的操作，统一SaveChanges，就是开启了一个事务；

.NET5 MVC+EFCore+分层整合

没有分层:

1.职责不清晰

2.如果有一处修改，会导致全部需要重新修改，就需要全部重新测试

分层架构:

UI:表现给用户查看

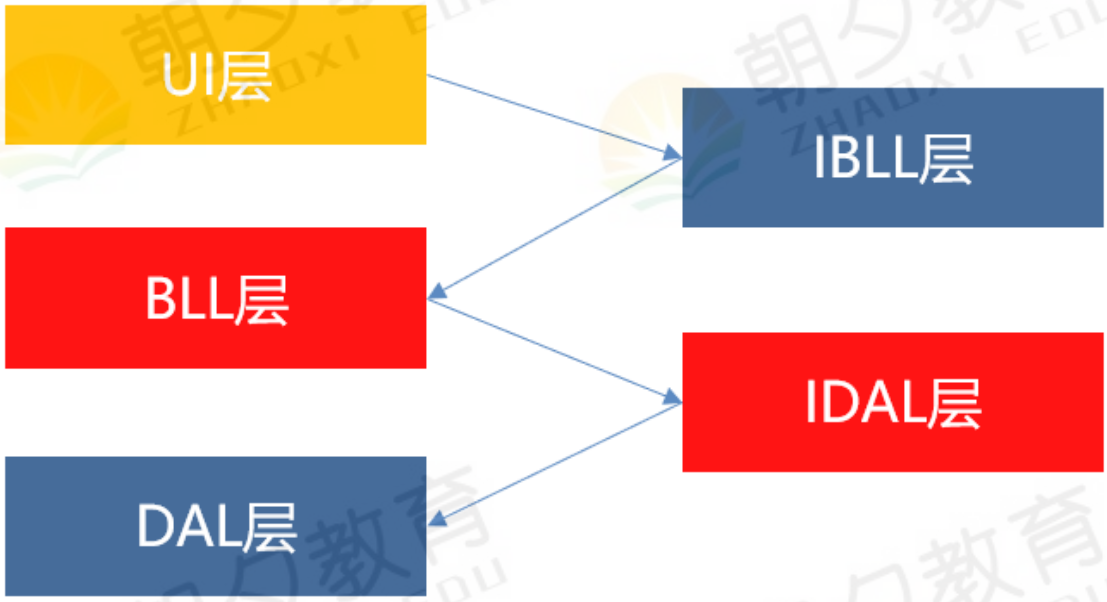
Bll：业务逻辑计算

Dal:数据库访问

1.职责更加清晰

2.需求的变更不用修改全部代码；

3.人员更好调配---让更加专业的人来做专业的事儿



分层以后：要求不能跨层调用，UI层----Bll层---Bll---Dal层

.NET5 MVC+EFCore+分层整合

因为每一个数据库的表，一般情况都是独立的来一个Service,那类似于一些基础操作；每个Service中都会存在；代码有很多重复；

支持依赖注入：必须有抽象，依赖抽象而不是依赖于细节；