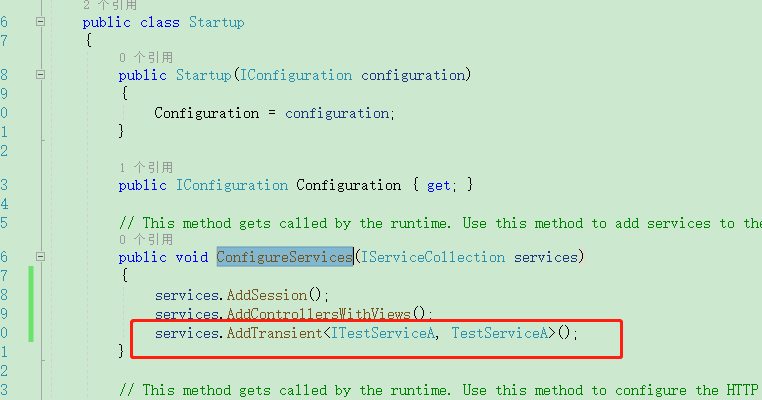
Demo:

1. 生成 iService接口 及其实现 Service
2. 项目Startup的ConfigureServices 注入该接口及其实现：services.AddTransient<ITestServiceA, TestServiceA>();

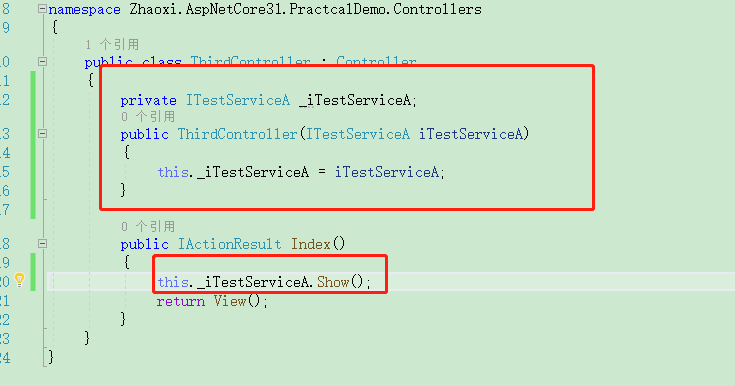


services.AddTransient<ITestServiceC, TestServiceC>();//瞬时，即时构造（常用）

services.AddSingleton<ITestServiceC, TestServiceC>();//单例-全程唯一(不建议太多单例，一般只在(配置文件，线程池 等)需要的地方才单例)

services.AddScoped<ITestServiceC, TestServiceC>();//作用域单例，本质是容器实例单例(一个http请求就是一个单例，这说明一次请求会创建一个容器实例)例如 事务,数据库连接，主要适用于一次请求只需要一个对象，多个请求需要多个对象

3． 控制器中使用ioc



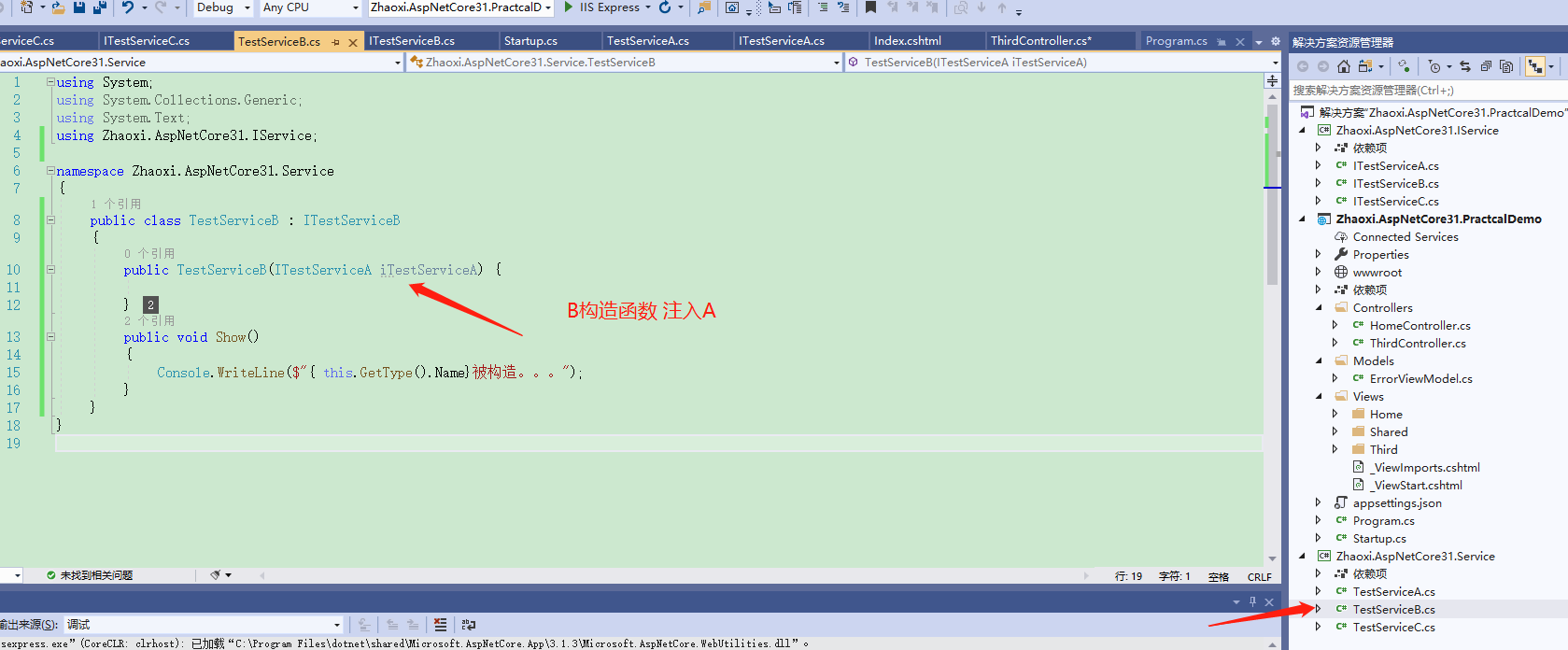
为什么使用Ioc

1. 去掉对细节的依赖，方便扩展，减少影响范围：甚至转移到配置文件依赖，只需要改配置文件就行了

一般写法：ITestServiceA testServiceA = new TestServiceA();

如果多个地方有同样的语句，一旦TestServiceA这个类需要更改名称，那所有的地方都需要修改， ioc只需要改StartUp就行了

1. 屏蔽细节



B构造函数注入A, C构造函数注入B ,可以直接调用，

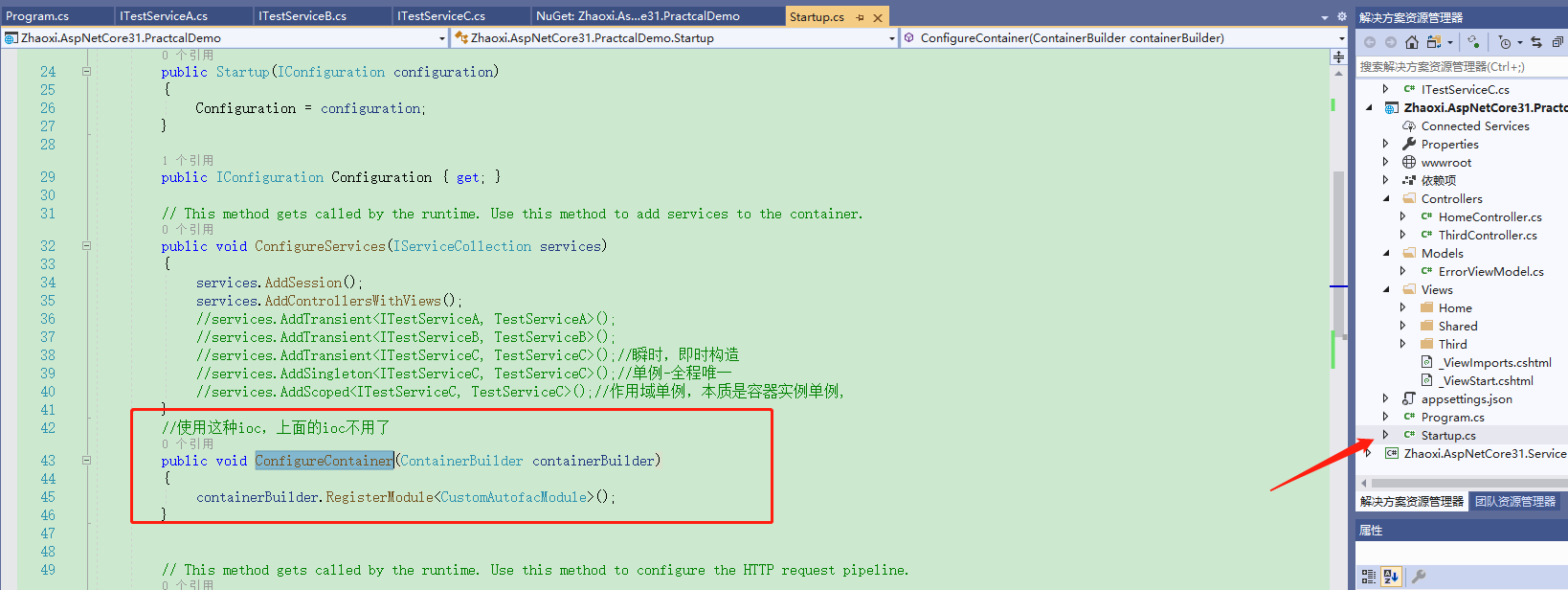
如果没有Ioc，使用C时，C就必须先构造B,B先构造A, 上级必须全部细节，而且麻烦

this.\_iTestServiceA.Show();

this.\_iTestServiceB.Show();

this.\_iTestServiceC.Show();

Ioc 使用方法（第三方扩展，包含aop）

1. 安装 AutoFac和Autofac.Extensions.DependencyInjection、Autofac.Extras.DynamicProxy包
2. 配置Program
3. public static IHostBuilder CreateHostBuilder(string[] args) =>
4. Host.CreateDefaultBuilder(args)
5. .UseServiceProviderFactory(new AutofacServiceProviderFactory())//把默认的Ioc容器 ServicesCollections替换成 Autofac
6. .ConfigureWebHostDefaults(webBuilder =>
7. {
8. webBuilder.UseStartup<Startup>();
9. });
10. 配置Startup ，添加ConfigureContainer和以前代码
11. 

public class CustomAutofacModule:Autofac.Module

{

protected override void Load(ContainerBuilder containerBuilder)

{

var assembly = this.GetType().GetTypeInfo().Assembly;

var builder = new ContainerBuilder();

var manager = new ApplicationPartManager();

manager.ApplicationParts.Add(new AssemblyPart(assembly));

manager.FeatureProviders.Add(new ControllerFeatureProvider());

var feature = new ControllerFeature();

manager.PopulateFeature(feature);

builder.RegisterType<ApplicationPartManager>().AsSelf().SingleInstance();

builder.RegisterTypes(feature.Controllers.Select(ti => ti.AsType()).ToArray()).PropertiesAutowired();

containerBuilder.Register(c => new CustomAutofacAop());//aop注册

containerBuilder.RegisterType<TestServiceA>().As<ITestServiceA>().SingleInstance().PropertiesAutowired();

containerBuilder.RegisterType<TestServiceC>().As<ITestServiceC>();

containerBuilder.RegisterType<TestServiceB>().As<ITestServiceB>();

containerBuilder.RegisterType<A>().As<IA>().EnableInterfaceInterceptors();

}

}

public class CustomAutofacAop : IInterceptor

{

private static Dictionary<string, object> CustomAutofacAopDictionary = new Dictionary<string, object>();

public void Intercept(IInvocation invocation)

{

Console.WriteLine($"invocation.Methond={invocation.Method}");

Console.WriteLine($"invocation.Arguments={string.Join(",", invocation.Arguments)}");

string key = $"{invocation.Method}\_{string.Join(",", invocation.Arguments)}";

if (!CustomAutofacAopDictionary.ContainsKey(key))

{

invocation.Proceed(); //继续执行

CustomAutofacAopDictionary.Add(key, invocation.ReturnValue);

}

else

{

invocation.ReturnValue = CustomAutofacAopDictionary[key];

}

Console.WriteLine($"方法{invocation.Method}执行完成了");

}

}

//AOP缓存时 实现类里的有两个方法，但是想在使用时只缓存一个方法返回值怎么做--就得区分下---不想修改代码但是又想区分--加一个特性--然后判断下

public interface IA

{

void Show(int id, string name);

DateTime PlusTime(int i, int k);

}

/// <summary>

/// autofac是5.0

/// </summary>

[Intercept(typeof(CustomAutofacAop))]

public class A : IA

{

public void Show(int id, string name)

{

Console.WriteLine($"This is A {id} \_{name}");

}

public DateTime PlusTime(int i, int k)

{

Console.WriteLine("PlusTime");

return DateTime.Now;

}

}