Autofac

以下例子使用的.NET SDK版本为6.0.100,开发工具使用Microsoft Visual Studio Enterprise 2022 (64位) 版本 17.0.1

Autofac版本为6.3.0

安装Autofac包

```
Install-Package Autofac -Version 6.3.0
```

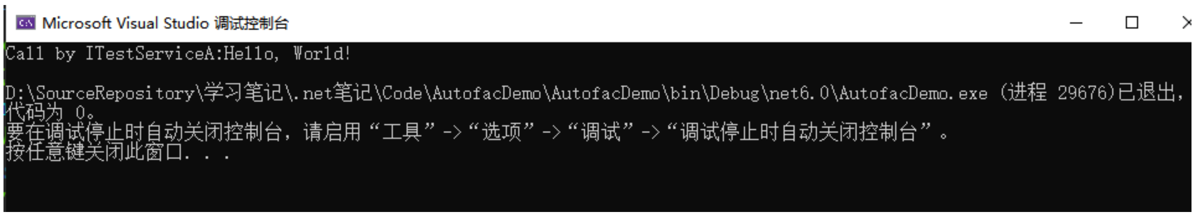
基本用法

代码路径..\学习笔记.net笔记\Code\AutofacDemo

`..\学习笔记\.net笔记\Code\AutofacDemo\AutofacDemo\Program.cs`

```
using Autofac;
using AutofacDemo.BLL;
using AutofacDemo.IBLL;

//实例化容器Builder
ContainerBuilder containerBuilder = new ContainerBuilder();
//注册服务
containerBuilder.RegisterType<TestServiceAimpl>().As<ITestServiceA>();
//创建容器
IContainer container = containerBuilder.Build();
//从容器中获取服务
ITestServiceA testServiceA = container.Resolve<ITestServiceA>();
//调用方法
testServiceA.Hello("Hello, World!");
```

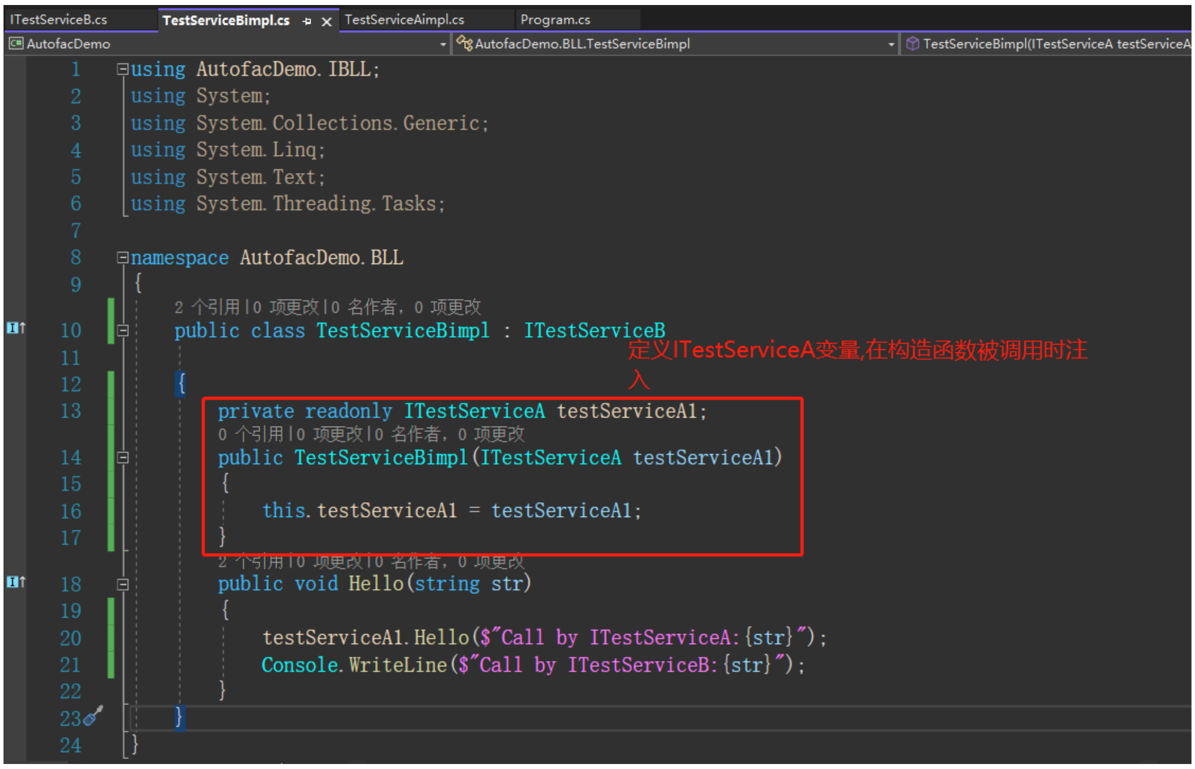
执行结果



```
Call by ITestServiceA:Hello, World!

D:\SourceRepository\学习笔记\.net笔记\Code\AutofacDemo\AutofacDemo\bin\Debug\net6.0\AutofacDemo.exe (进程 29676)已退出,
代码为 0。
要在调试停止时自动关闭控制台，请启用 “工具” -> “选项” -> “调试” -> “调试停止时自动关闭控制台” 。
按任意键关闭此窗口. . .
```

构造函数注入(默认方式)

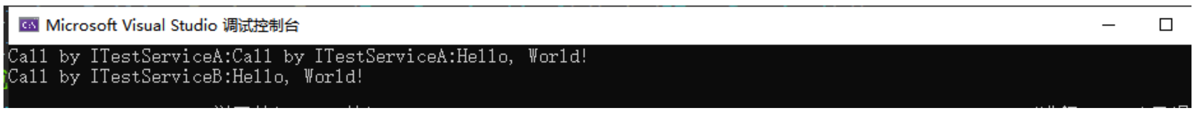
定义`TestServiceBimpl.cs`


ITestServiceB.cs
TestServiceBimpl.cs
TestServiceAimpl.cs
Program.cs
AutofacDemo
AutofacDemo.BLL.TestServiceBimpl
TestServiceBimpl(ITestServiceA testServiceA
定义ITestServiceA变量,在构造函数被调用时注入

```
using AutofacDemo.IBLL;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace AutofacDemo.BLL
{
    2 个引用|0 项更改|0 名作者，0 项更改
    public class TestServiceBimpl : ITestServiceB

    {
        private readonly ITestServiceA testServiceA1;
        0 个引用|0 项更改|0 名作者，0 项更改
        public TestServiceBimpl(ITestServiceA testServiceA1)
        {
            this.testServiceA1 = testServiceA1;
        }
        2 个引用|0 项更改|0 名作者，0 项更改
        public void Hello(string str)
        {
            testServiceA1.Hello($"Call by ITestServiceA:{str}");
            Console.WriteLine($"Call by ITestServiceB:{str}");
        }
    }
}
```

```
using Autofac;
using AutofacDemo.BLL;
using AutofacDemo.IBLL;

//实例化容器Builder
ContainerBuilder containerBuilder = new ContainerBuilder();
//注册服务
containerBuilder.RegisterType<TestServiceAimpl>().As<ITestServiceA>();
containerBuilder.RegisterType<TestServiceBimpl>().As<ITestServiceB>();
//创建容器
IContainer container = containerBuilder.Build();
//从容器中获取服务
//ITestServiceA testServiceA = container.Resolve<ITestServiceA>();
ITestServiceB testServiceB = container.Resolve<ITestServiceB>();
//调用方法
//testServiceA.Hello("Hello, World!");
testServiceB.Hello("Hello, World!");
```

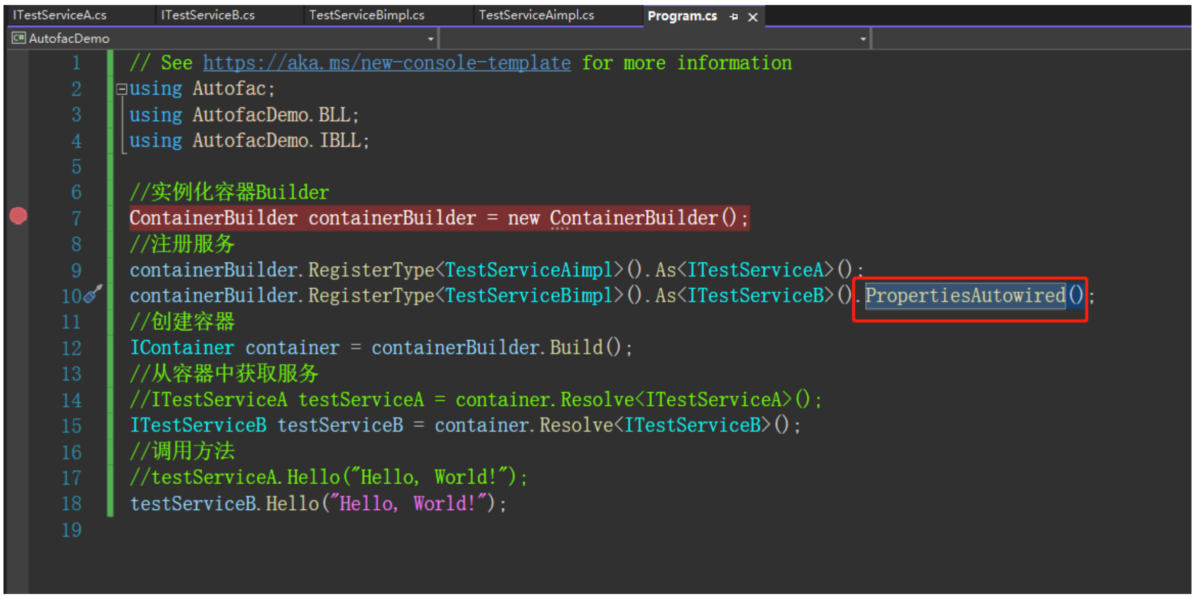
执行结果



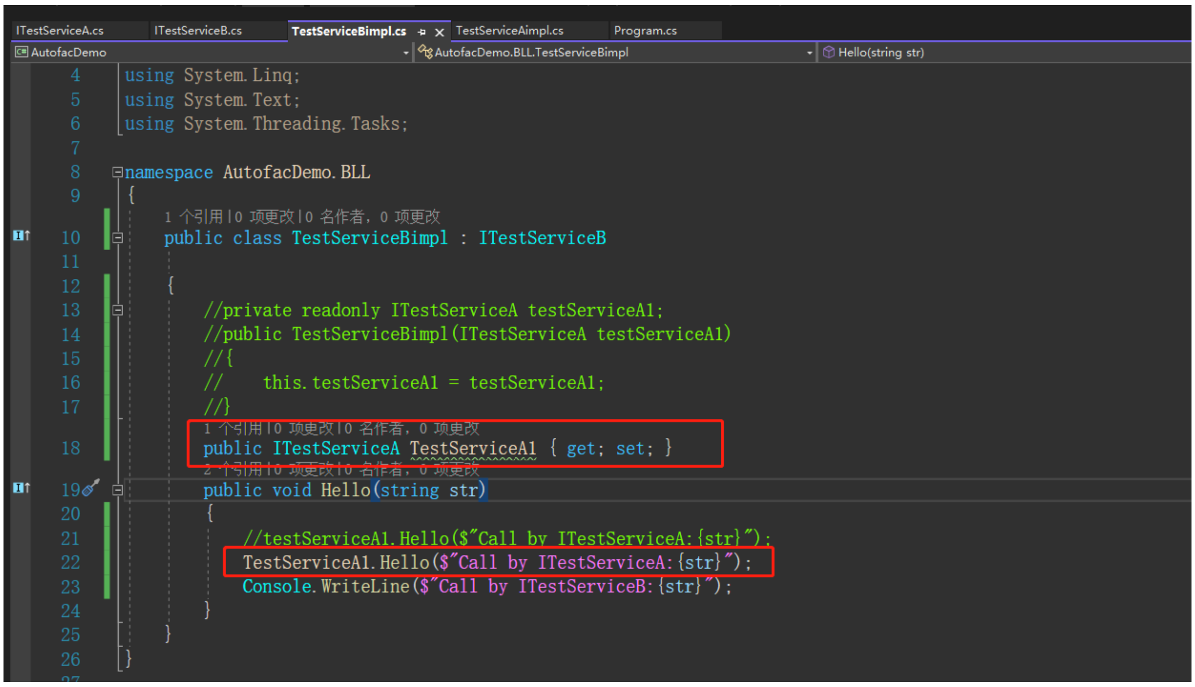
```
Call by ITestServiceA:Call by ITestServiceA:Hello, World!
Call by ITestServiceB:Hello, World!
```

属性注入

如果要用属性注入，需要在注册时调用PropertiesAutowired方法



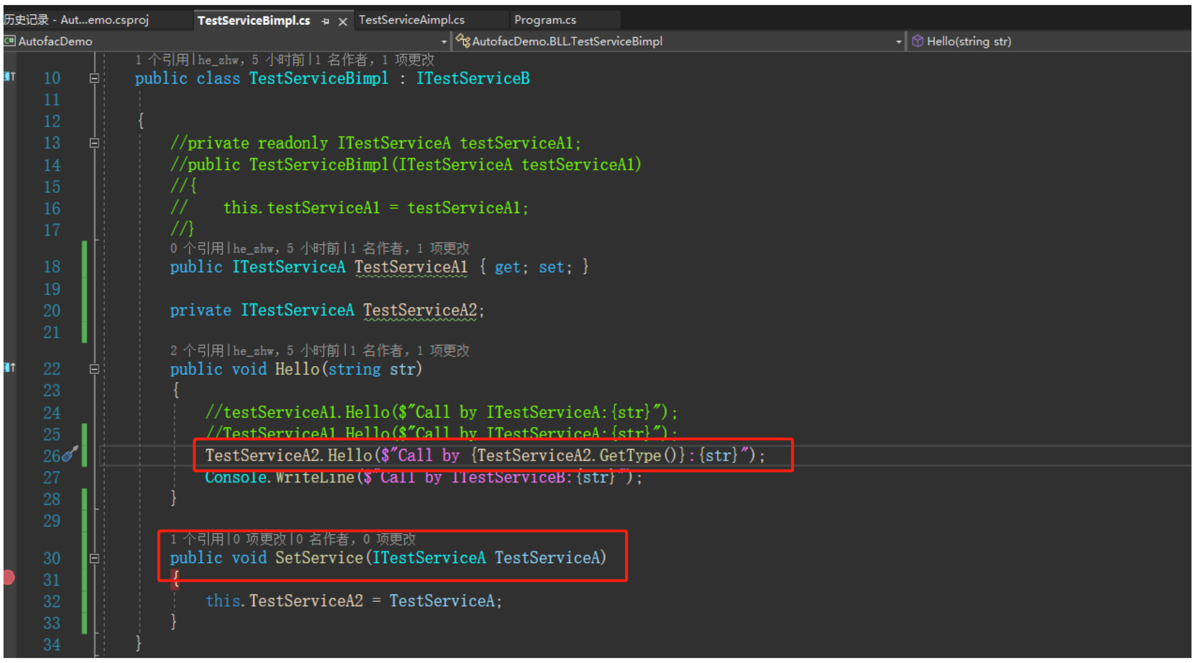
```csharp
// See https://aka.ms/new-console-template for more information
using Autofac;
using AutofacDemo.BLL;
using AutofacDemo.IBLL;

//实例化容器Builder
ContainerBuilder containerBuilder = new ContainerBuilder();
//注册服务
containerBuilder.RegisterType<TestServiceAimpl>().As<ITestServiceA>();
containerBuilder.RegisterType<TestServiceBimpl>().As<ITestServiceB>().PropertiesAutowired();
//创建容器
IContainer container = containerBuilder.Build();
//从容器中获取服务
//ITestServiceA testServiceA = container.Resolve<ITestServiceA>();
ITestServiceB testServiceB = container.Resolve<ITestServiceB>();
//调用方法
//testServiceA.Hello("Hello, World!");
testServiceB.Hello("Hello, World!");
```



```csharp
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace AutofacDemo.BLL
{
    1 个引用|0 项更改|0 名作者，0 项更改
    public class TestServiceBimpl : ITestServiceB

    {
        //private readonly ITestServiceA testServiceA1;
        //public TestServiceBimpl(ITestServiceA testServiceA1)
        //{
        //    this.testServiceA1 = testServiceA1;
        //}
        1 个引用|0 项更改|0 名作者，0 项更改
        public ITestServiceA TestServiceA1 { get; set; }
        2 个引用|0 项更改|0 名作者，0 项更改
        public void Hello(string str)
        {
            //testServiceA1.Hello($"Call by ITestServiceA:{str}");
            TestServiceA1.Hello($"Call by ITestServiceA:{str}");
            Console.WriteLine($"Call by ITestServiceB:{str}");
        }
    }
}
```

方法注入

`TestServiceBimpl`类自定义一个方法，参数类别是`ITestServiceA`

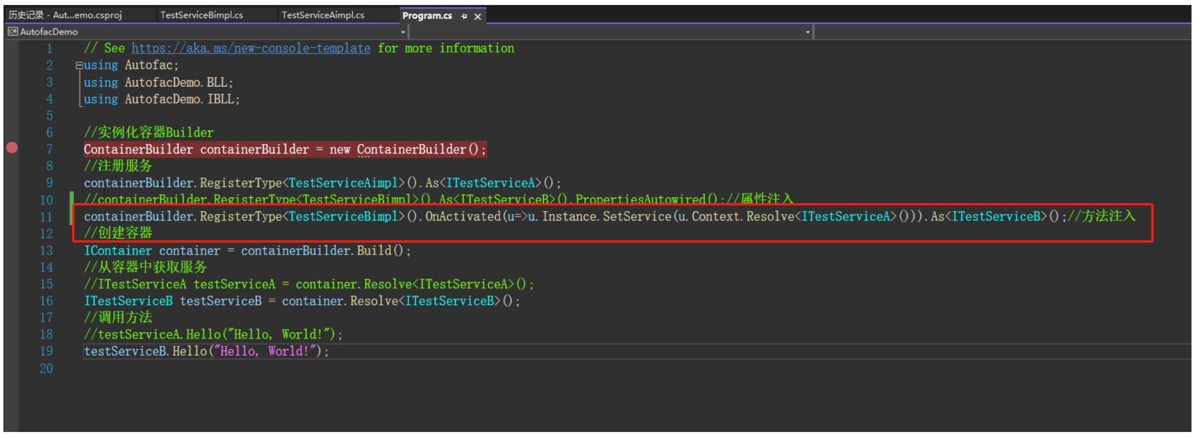


历史记录 - Aut...emo.csproj　TestServiceBimpl.cs　TestServiceAimpl.cs　Program.cs

AutofacDemo　AutofacDemo.BLL.TestServiceBimpl　Hello(string str)

```
1 个引用|he_zhw, 5 小时前|1 名作者, 1 项更改
public class TestServiceBimpl : ITestServiceB

{
    //private readonly ITestServiceA testServiceA1;
    //public TestServiceBimpl(ITestServiceA testServiceA1)
    //{
    //    this.testServiceA1 = testServiceA1;
    //}
    0 个引用|he_zhw, 5 小时前|1 名作者, 1 项更改
    public ITestServiceA TestServiceA1 { get; set; }

    private ITestServiceA TestServiceA2;

    2 个引用|he_zhw, 5 小时前|1 名作者, 1 项更改
    public void Hello(string str)
    {
        //testServiceA1.Hello($"Call by ITestServiceA:{str}");
        //TestServiceA1.Hello($"Call by ITestServiceA:{str}");
        TestServiceA2.Hello($"Call by {TestServiceA2.GetType()}:{str}");
        Console.WriteLine($"Call by ITestServiceB:{str}");
    }

    1 个引用|0 项更改|0 名作者, 0 项更改
    public void SetService(ITestServiceA TestServiceA)
    {
        this.TestServiceA2 = TestServiceA;
    }
}
```

注册服务使用方法注入



历史记录 - Aut...emo.csproj　TestServiceBimpl.cs　TestServiceAimpl.cs　Program.cs

AutofacDemo

```
// See https://aka.ms/new-console-template for more information
using Autofac;
using AutofacDemo.BLL;
using AutofacDemo.IBLL;

//实例化容器Builder
ContainerBuilder containerBuilder = new ContainerBuilder();
//注册服务
containerBuilder.RegisterType<TestServiceAimpl>().As<ITestServiceA>();
//containerBuilder.RegisterType<TestServiceBimpl>().As<ITestServiceB>().PropertiesAutowired();//属性注入
containerBuilder.RegisterType<TestServiceBimpl>().OnActivated(u=>u.Instance.SetService(u.Context.Resolve<ITestServiceA>())).As<ITestServiceB>();//方法注入
//创建容器
IContainer container = containerBuilder.Build();
//从容器中获取服务
//ITestServiceA testServiceA = container.Resolve<ITestServiceA>();
ITestServiceB testServiceB = container.Resolve<ITestServiceB>();
//调用方法
//testServiceA.Hello("Hello, World!");
testServiceB.Hello("Hello, World!");
```

容器中对象的生命周期

瞬时生命周期InstancePerDependency(默认)

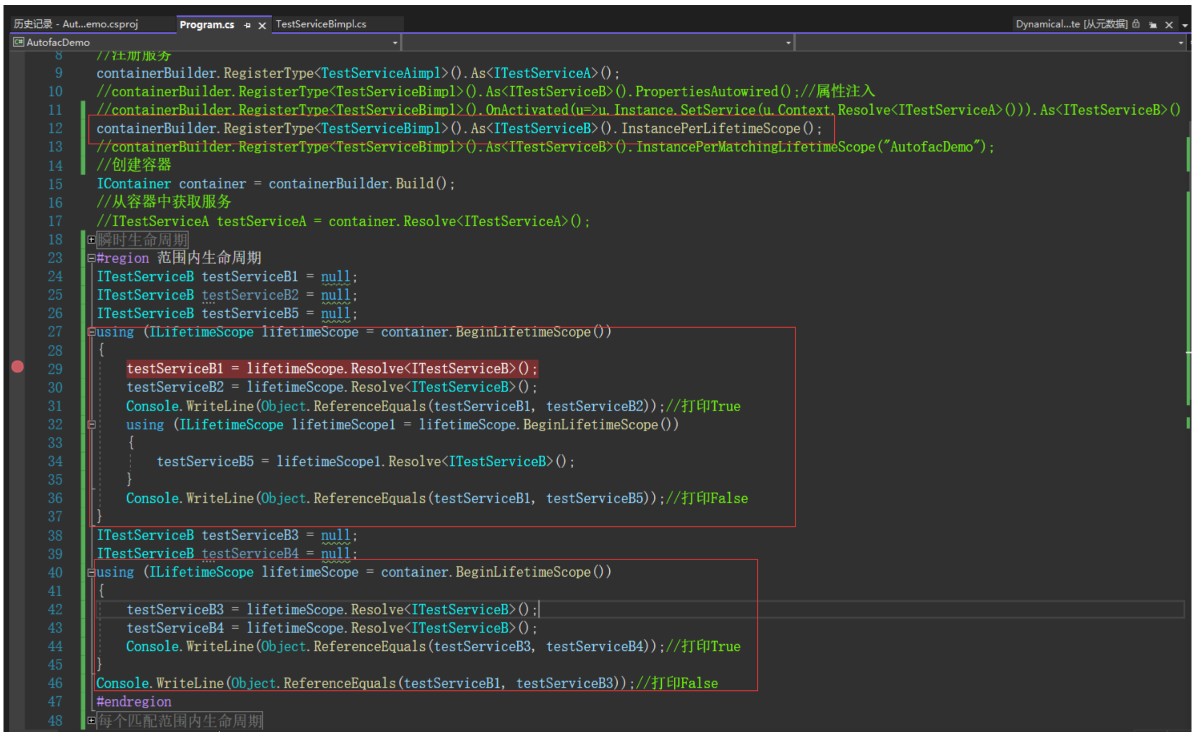
瞬时生命周期：每一次从容器中获取对象都是一个全新的实例，默认的生命周期。

```
using Autofac;
using AutofacDemo.BLL;
using AutofacDemo.IBLL;

//实例化容器Builder
ContainerBuilder containerBuilder = new ContainerBuilder();
//注册服务
containerBuilder.RegisterType<TestServiceAimpl>().As<ITestServiceA>();
//containerBuilder.RegisterType<TestServiceBimpl>().As<ITestServiceB>
().PropertiesAutowired();//属性注入
containerBuilder.RegisterType<TestServiceBimpl>
().OnActivated(u=>u.Instance.SetService(u.Context.Resolve<ITestServiceA>
())).As<ITestServiceB>();//方法注入
//创建容器
IContainer container = containerBuilder.Build();
//从容器中获取服务
//ITestServiceA testServiceA = container.Resolve<ITestServiceA>();
ITestServiceB testServiceB = container.Resolve<ITestServiceB>();
ITestServiceB testServiceB1 = container.Resolve<ITestServiceB>();
Console.WriteLine(Object.ReferenceEquals(testServiceB, testServiceB1));//最后打印的是False
```

范围内生命周期(InstancePerLifetimeScope)

`某个范围内获取的都是同一个实例`

在注册实例时调用`InstancePerLifetimeScope`方法

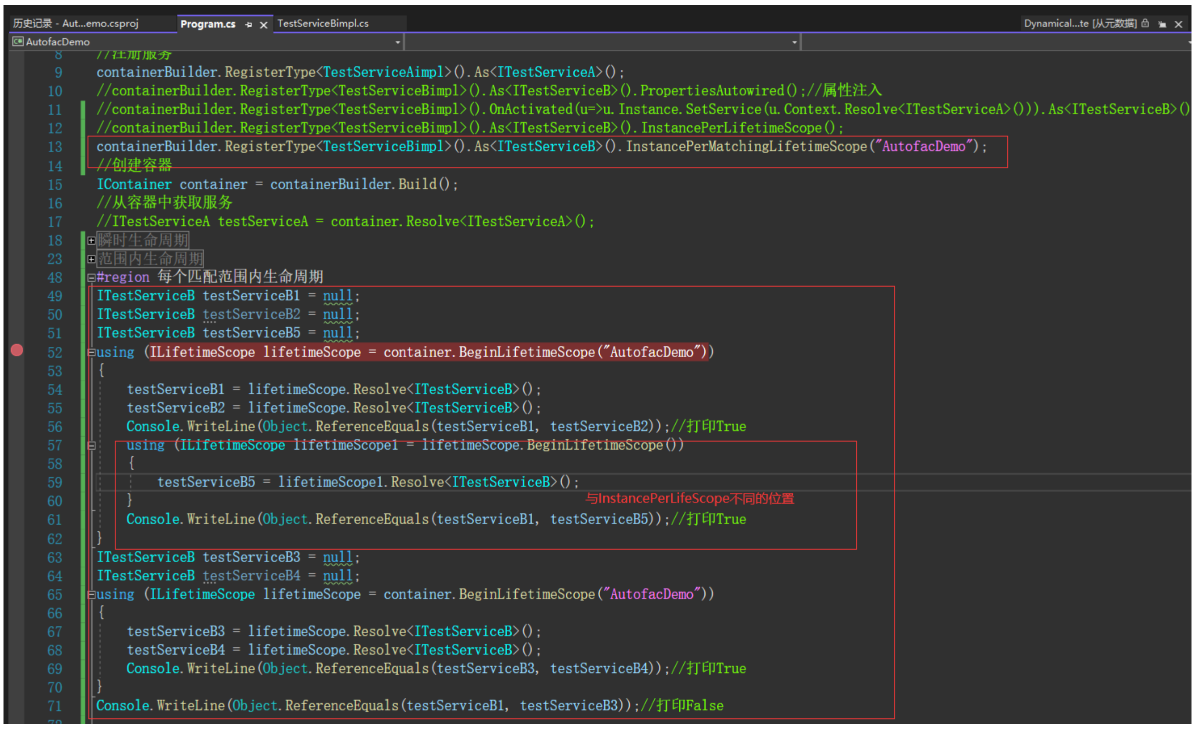


```
//注册服务
containerBuilder.RegisterType<TestServiceAimpl>().As<ITestServiceA>();
//containerBuilder.RegisterType<TestServiceBimpl>().As<ITestServiceB>().PropertiesAutowired();//属性注入
//containerBuilder.RegisterType<TestServiceBimpl>().OnActivated(u=>u.Instance.SetService(u.Context.Resolve<ITestServiceA>())).As<ITestServiceB>()
containerBuilder.RegisterType<TestServiceBimpl>().As<ITestServiceB>().InstancePerLifetimeScope();
//containerBuilder.RegisterType<TestServiceBimpl>().As<ITestServiceB>().InstancePerMatchingLifetimeScope("AutofacDemo");
//创建容器
IContainer container = containerBuilder.Build();
//从容器中获取服务
//ITestServiceA testServiceA = container.Resolve<ITestServiceA>();
瞬时生命周期
#region 范围内生命周期
ITestServiceB testServiceB1 = null;
ITestServiceB testServiceB2 = null;
ITestServiceB testServiceB5 = null;
using (ILifetimeScope lifetimeScope = container.BeginLifetimeScope())
{
    testServiceB1 = lifetimeScope.Resolve<ITestServiceB>();
    testServiceB2 = lifetimeScope.Resolve<ITestServiceB>();
    Console.WriteLine(Object.ReferenceEquals(testServiceB1, testServiceB2));//打印True
    using (ILifetimeScope lifetimeScope1 = lifetimeScope.BeginLifetimeScope())
    {
        testServiceB5 = lifetimeScope1.Resolve<ITestServiceB>();
    }
    Console.WriteLine(Object.ReferenceEquals(testServiceB1, testServiceB5));//打印False
}
ITestServiceB testServiceB3 = null;
ITestServiceB testServiceB4 = null;
using (ILifetimeScope lifetimeScope = container.BeginLifetimeScope())
{
    testServiceB3 = lifetimeScope.Resolve<ITestServiceB>();
    testServiceB4 = lifetimeScope.Resolve<ITestServiceB>();
    Console.WriteLine(Object.ReferenceEquals(testServiceB3, testServiceB4));//打印True
}
Console.WriteLine(Object.ReferenceEquals(testServiceB1, testServiceB3));//打印False
#endregion
每个匹配范围内生命周期
```

```
// See https://aka.ms/new-console-template for more information
using Autofac;
using AutofacDemo.BLL;
using AutofacDemo.IBLL;

//实例化容器Builder
ContainerBuilder containerBuilder = new ContainerBuilder();
//注册服务
containerBuilder.RegisterType<TestServiceAimpl>().As<ITestServiceA>();
//containerBuilder.RegisterType<TestServiceBimpl>().As<ITestServiceB>().PropertiesAutowired();//属性注入
//containerBuilder.RegisterType<TestServiceBimpl>().OnActivated(u=>u.Instance.SetService(u.Context.Resolve<ITestServiceA>())).As<ITestServiceB>();//方法注入
containerBuilder.RegisterType<TestServiceBimpl>().As<ITestServiceB>().InstancePerLifetimeScope();
//创建容器
IContainer container = containerBuilder.Build();
//从容器中获取服务
//ITestServiceA testServiceA = container.Resolve<ITestServiceA>();
#region 瞬时生命周期
//ITestServiceB testServiceB = container.Resolve<ITestServiceB>();
//ITestServiceB testServiceB1 = container.Resolve<ITestServiceB>();
//Console.WriteLine(Object.ReferenceEquals(testServiceB, testServiceB1));
#endregion
#region 范围内生命周期
```

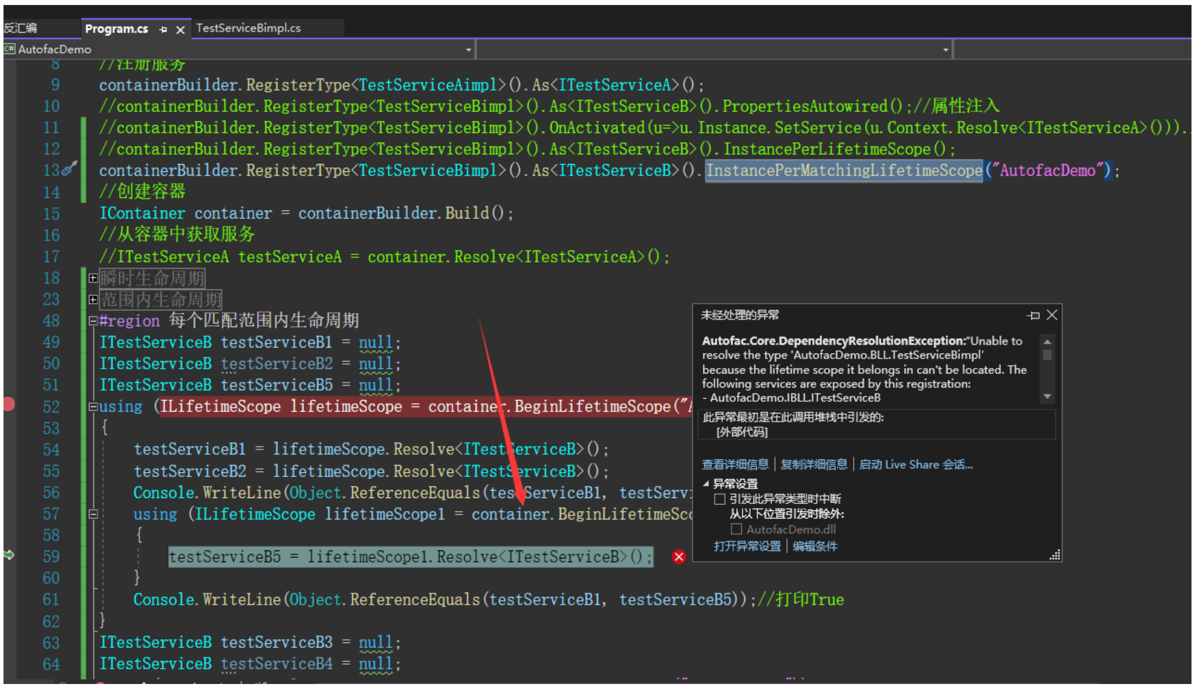
```
ITestServiceB testServiceB1 = null;
ITestServiceB testServiceB2 = null;
ITestServiceB testServiceB5 = null;
using (ILifetimeScope lifetimeScope = container.BeginLifetimeScope())
{
     testServiceB1 = lifetimeScope.Resolve<ITestServiceB>();
     testServiceB2 = lifetimeScope.Resolve<ITestServiceB>();
     Console.WriteLine(Object.ReferenceEquals(testServiceB1, testServiceB2));//打
印True
    using (ILifetimeScope lifetimeScope1 = container.BeginLifetimeScope())
    {
        testServiceB5 = lifetimeScope1.Resolve<ITestServiceB>();
    }
    Console.WriteLine(Object.ReferenceEquals(testServiceB1, testServiceB5));//打
印False
}
ITestServiceB testServiceB3 = null;
ITestServiceB testServiceB4 = null;
using (ILifetimeScope lifetimeScope = container.BeginLifetimeScope())
{
    testServiceB3 = lifetimeScope.Resolve<ITestServiceB>();
    testServiceB4 = lifetimeScope.Resolve<ITestServiceB>();
    Console.WriteLine(Object.ReferenceEquals(testServiceB3, testServiceB4));//打
印True
}
Console.WriteLine(Object.ReferenceEquals(testServiceB1, testServiceB3));//打印
False

#endregion
//调用方法
//testServiceA.Hello("Hello, World!");
//testServiceB.Hello("Hello, World!");
```

每个匹配生命周期范围一个实例 (InstancePerMatchingLifetimeScope)


历史记录 - Aut...emo.csproj
Program.cs
TestServiceBimpl.cs
Dynamical...te [从元数据]
AutofacDemo
与InstancePerLifeScope不同的位置

```
//注册服务
containerBuilder.RegisterType<TestServiceAimpl>().As<ITestServiceA>();
//containerBuilder.RegisterType<TestServiceBimpl>().As<ITestServiceB>().PropertiesAutowired();//属性注入
//containerBuilder.RegisterType<TestServiceBimpl>().OnActivated(u=>u.Instance.SetService(u.Context.Resolve<ITestServiceA>())).As<ITestServiceB>()
//containerBuilder.RegisterType<TestServiceBimpl>().As<ITestServiceB>().InstancePerLifetimeScope();
containerBuilder.RegisterType<TestServiceBimpl>().As<ITestServiceB>().InstancePerMatchingLifetimeScope("AutofacDemo");
//创建容器
IContainer container = containerBuilder.Build();
//从容器中获取服务
//ITestServiceA testServiceA = container.Resolve<ITestServiceA>();
瞬时生命周期
范围内生命周期
#region 每个匹配范围内生命周期
ITestServiceB testServiceB1 = null;
ITestServiceB testServiceB2 = null;
ITestServiceB testServiceB5 = null;
using (ILifetimeScope lifetimeScope = container.BeginLifetimeScope("AutofacDemo"))
{
    testServiceB1 = lifetimeScope.Resolve<ITestServiceB>();
    testServiceB2 = lifetimeScope.Resolve<ITestServiceB>();
    Console.WriteLine(Object.ReferenceEquals(testServiceB1, testServiceB2));//打印True
    using (ILifetimeScope lifetimeScope1 = lifetimeScope.BeginLifetimeScope())
    {
        testServiceB5 = lifetimeScope1.Resolve<ITestServiceB>();
    }
    Console.WriteLine(Object.ReferenceEquals(testServiceB1, testServiceB5));//打印True
}
ITestServiceB testServiceB3 = null;
ITestServiceB testServiceB4 = null;
using (ILifetimeScope lifetimeScope = container.BeginLifetimeScope("AutofacDemo"))
{
    testServiceB3 = lifetimeScope.Resolve<ITestServiceB>();
    testServiceB4 = lifetimeScope.Resolve<ITestServiceB>();
    Console.WriteLine(Object.ReferenceEquals(testServiceB3, testServiceB4));//打印True
}
Console.WriteLine(Object.ReferenceEquals(testServiceB1, testServiceB3));//打印False
```

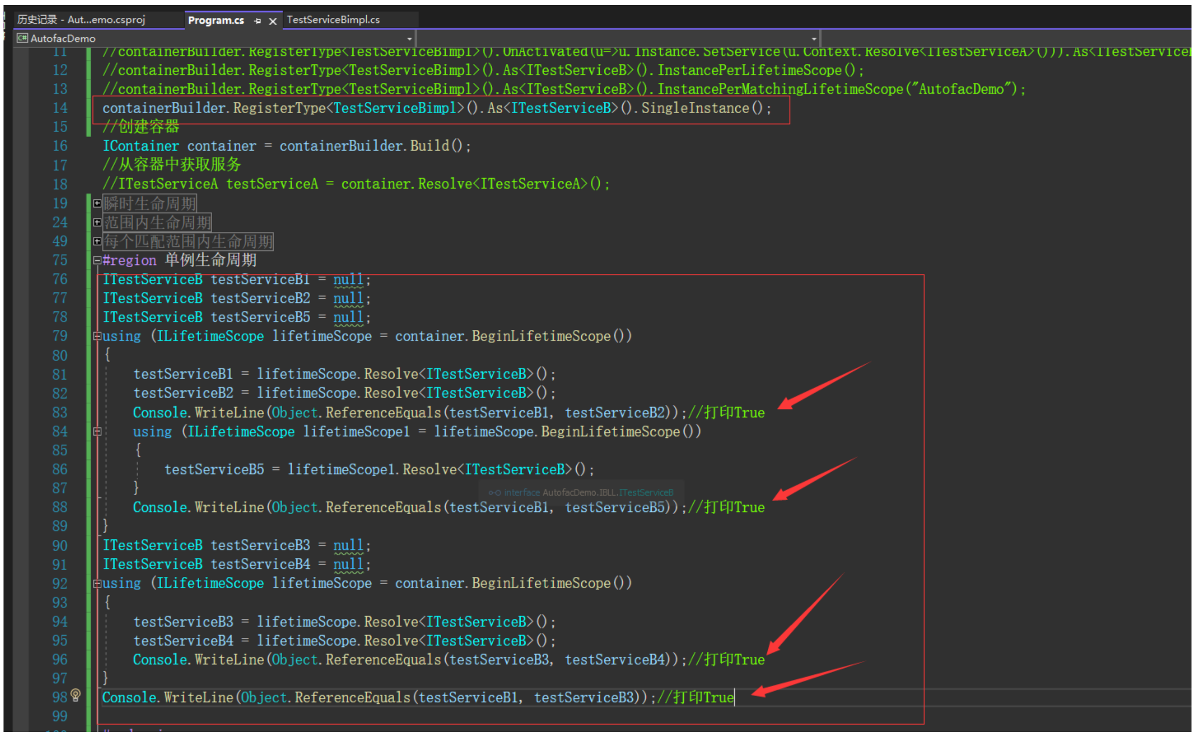
还有一点与InstancePerLifetimeScope中的不同点是，如果在InstancePerMatchingLifetimeScope范围内在用IContainer的对象取开启生命周期则会报错，而在InstancePerLifetimeScope中这种用法不会报错



```
//注册服务
containerBuilder.RegisterType<TestServiceAimpl>().As<ITestServiceA>();
//containerBuilder.RegisterType<TestServiceBimpl>().As<ITestServiceB>().PropertiesAutowired();//属性注入
//containerBuilder.RegisterType<TestServiceBimpl>().OnActivated(u=>u.Instance.SetService(u.Context.Resolve<ITestServiceA>())).
//containerBuilder.RegisterType<TestServiceBimpl>().As<ITestServiceB>().InstancePerLifetimeScope();
containerBuilder.RegisterType<TestServiceBimpl>().As<ITestServiceB>().InstancePerMatchingLifetimeScope("AutofacDemo");
//创建容器
IContainer container = containerBuilder.Build();
//从容器中获取服务
//ITestServiceA testServiceA = container.Resolve<ITestServiceA>();
瞬时生命周期
范围内生命周期
#region 每个匹配范围内生命周期
ITestServiceB testServiceB1 = null;
ITestServiceB testServiceB2 = null;
ITestServiceB testServiceB5 = null;
using (ILifetimeScope lifetimeScope = container.BeginLifetimeScope("A
{
    testServiceB1 = lifetimeScope.Resolve<ITestServiceB>();
    testServiceB2 = lifetimeScope.Resolve<ITestServiceB>();
    Console.WriteLine(Object.ReferenceEquals(testServiceB1, testServi
    using (ILifetimeScope lifetimeScope1 = container.BeginLifetimeSco
    {
        testServiceB5 = lifetimeScope1.Resolve<ITestServiceB>();
    }
    Console.WriteLine(Object.ReferenceEquals(testServiceB1, testServiceB5));//打印True
}
ITestServiceB testServiceB3 = null;
ITestServiceB testServiceB4 = null;
```

```
未经处理的异常
Autofac.Core.DependencyResolutionException:"Unable to resolve the type 'AutofacDemo.BLL.TestServiceBimpl' because the lifetime scope it belongs in can't be located. The following services are exposed by this registration:
- AutofacDemo.IBLL.ITestServiceB
此异常最初是在此调用堆栈中引发的:
    [外部代码]
```

单例生命周期(SingleInstance)



```
//containerBuilder.RegisterType<TestServiceBimpl>().OnActivated(u=>u.Instance.SetService(u.Context.Resolve<ITestServiceA>())).As<ITestService
//containerBuilder.RegisterType<TestServiceBimpl>().As<ITestServiceB>().InstancePerLifetimeScope();
//containerBuilder.RegisterType<TestServiceBimpl>().As<ITestServiceB>().InstancePerMatchingLifetimeScope("AutofacDemo");
containerBuilder.RegisterType<TestServiceBimpl>().As<ITestServiceB>().SingleInstance();
//创建容器
IContainer container = containerBuilder.Build();
//从容器中获取服务
//ITestServiceA testServiceA = container.Resolve<ITestServiceA>();
瞬时生命周期
范围内生命周期
每个匹配范围内生命周期
#region 单例生命周期
ITestServiceB testServiceB1 = null;
ITestServiceB testServiceB2 = null;
ITestServiceB testServiceB5 = null;
using (ILifetimeScope lifetimeScope = container.BeginLifetimeScope())
{
    testServiceB1 = lifetimeScope.Resolve<ITestServiceB>();
    testServiceB2 = lifetimeScope.Resolve<ITestServiceB>();
    Console.WriteLine(Object.ReferenceEquals(testServiceB1, testServiceB2));//打印True
    using (ILifetimeScope lifetimeScope1 = lifetimeScope.BeginLifetimeScope())
    {
        testServiceB5 = lifetimeScope1.Resolve<ITestServiceB>();
    }
    Console.WriteLine(Object.ReferenceEquals(testServiceB1, testServiceB5));//打印True
}
ITestServiceB testServiceB3 = null;
ITestServiceB testServiceB4 = null;
using (ILifetimeScope lifetimeScope = container.BeginLifetimeScope())
{
    testServiceB3 = lifetimeScope.Resolve<ITestServiceB>();
    testServiceB4 = lifetimeScope.Resolve<ITestServiceB>();
    Console.WriteLine(Object.ReferenceEquals(testServiceB3, testServiceB4));//打印True
}
Console.WriteLine(Object.ReferenceEquals(testServiceB1, testServiceB3));//打印True
```

每个请求一个实例(InstancePerRequest)

这个不好演示,等到整合web项目时再演示

InstancePerOwned

这个由使用者自己控制

配置文件配置实例

nuget安装Autofac.Configuration配置扩展包(例子使用6.0.0版本)

```
Install-Package Autofac.Configuration -Version 6.0.0
```

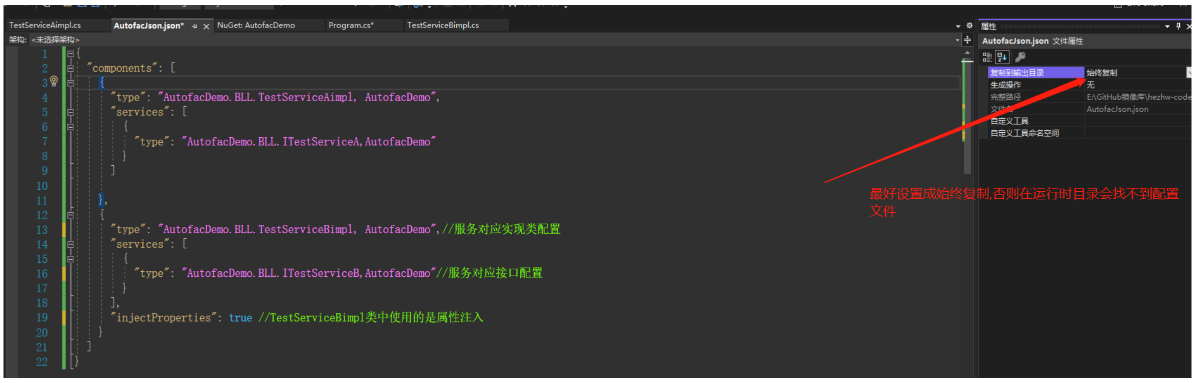
使用`json`文件配置

nuget安装json文件配置扩展包(例子使用6.0.0版本)

```
Install-Package Microsoft.Extensions.Configuration.Json -Version 6.0.0
```

创建配置文件`Conf\AutofacJson.json`

```
{
  "components": [
    {
      "type": "AutofacDemo.BLL.TestServiceAimpl, AutofacDemo",
      "services": [
        {
          "type": "AutofacDemo.IBLL.ITestServiceA,AutofacDemo"
        }
      ],
      "instanceScope": "Per-Lifetime-Scope"//生命周期
    },
    {
      "type": "AutofacDemo.BLL.TestServiceBimpl, AutofacDemo", //服务对应实现类配置
      "services": [
        {
          "type": "AutofacDemo.IBLL.ITestServiceB,AutofacDemo" //服务对应接口配置
        }
      ],
      "injectProperties": true //TestServiceBimpl类中使用的是属性注入
    }
  ]
}
```



```
{
  "components": [
    {
      "type": "AutofacDemo.BLL.TestServiceAimpl, AutofacDemo",
      "services": [
        {
          "type": "AutofacDemo.BLL.ITestServiceA,AutofacDemo"
        }
      ]


    },
    {
      "type": "AutofacDemo.BLL.TestServiceBimpl, AutofacDemo",//服务对应实现类配置
      "services": [
        {
          "type": "AutofacDemo.BLL.ITestServiceB,AutofacDemo"//服务对应接口配置
        }
      ],
      "injectProperties": true //TestServiceBimpl类中使用的是属性注入
    }
  ]
}
```

复制到输出目录	始终复制
生成操作	无
完整路径	E:\GitHub镜像库\hezhw-code
文件名	AutofacJson.json
自定义工具	
自定义工具命名空间	

最好设置成始终复制,否则在运行时目录会找不到配置文件

Json配置文件中生命周期instanceScope值的写法

- single-instance(单例)
- per-dependency(瞬时)
- per-lifetime-scope((每个生命周期范围的实例)
- per-request(每个请求一个实例)

示例代码

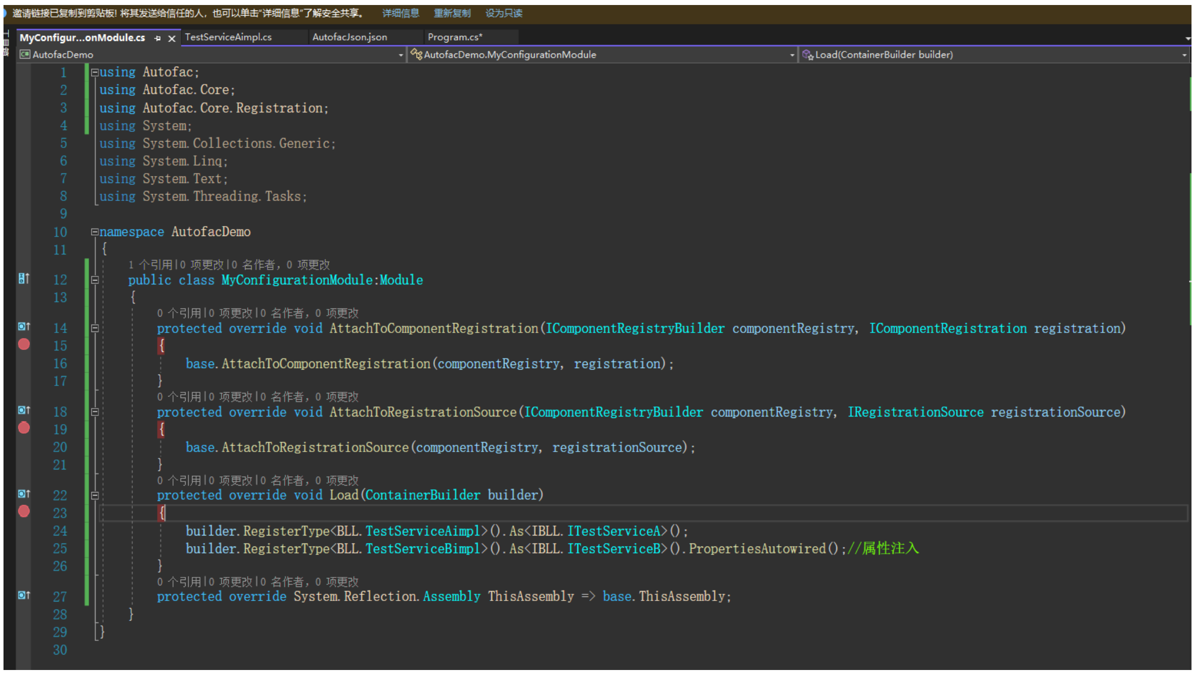
```
// 实例化ConfigurationBuilder.
var config = new Microsoft.Extensions.Configuration.ConfigurationBuilder();
//使用Microsoft.Extensions.Configuration.Json读取json配置文件
config.AddJsonFile("Conf/AutofacJson.json");

// Register the ConfigurationModule with Autofac.
var module = new Autofac.Configuration.ConfigurationModule(config.Build());//将配置文件加载至module
var builder = new ContainerBuilder();//创建ContainerBuilder
builder.RegisterModule(module);//注册服务
IContainer container = builder.Build();//创建容器
ITestServiceB testServiceB = container.Resolve<ITestServiceB>();//获取实例
testServiceB.Hello("Hello, World!");
```

其他常用属性配置

image-20211128105041985

Module的使用

自定义MyConfigurationModule继承抽象类Autofac.Module


邀请链接已复制到剪贴板! 将其发送给信任的人，也可以单击"详细信息"了解安全共享。 详细信息 重新复制 设为只读
MyConfigur...onModule.cs
TestServiceAimpl.cs
AutofacJson.json
Program.cs*
AutofacDemo
AutofacDemo.MyConfigurationModule
Load(ContainerBuilder builder)

```
using Autofac;
using Autofac.Core;
using Autofac.Core.Registration;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace AutofacDemo
{
    1 个引用|0 项更改|0 名作者，0 项更改
    public class MyConfigurationModule:Module
    {
        0 个引用|0 项更改|0 名作者，0 项更改
        protected override void AttachToComponentRegistration(IComponentRegistryBuilder componentRegistry, IComponentRegistration registration)
        {
            base.AttachToComponentRegistration(componentRegistry, registration);
        }
        0 个引用|0 项更改|0 名作者，0 项更改
        protected override void AttachToRegistrationSource(IComponentRegistryBuilder componentRegistry, IRegistrationSource registrationSource)
        {
            base.AttachToRegistrationSource(componentRegistry, registrationSource);
        }
        0 个引用|0 项更改|0 名作者，0 项更改
        protected override void Load(ContainerBuilder builder)
        {
            builder.RegisterType<BLL.TestServiceAimpl>().As<IBLL.ITestServiceA>();
            builder.RegisterType<BLL.TestServiceBimpl>().As<IBLL.ITestServiceB>().PropertiesAutowired();//属性注入
        }
        0 个引用|0 项更改|0 名作者，0 项更改
        protected override System.Reflection.Assembly ThisAssembly => base.ThisAssembly;
    }
}
```

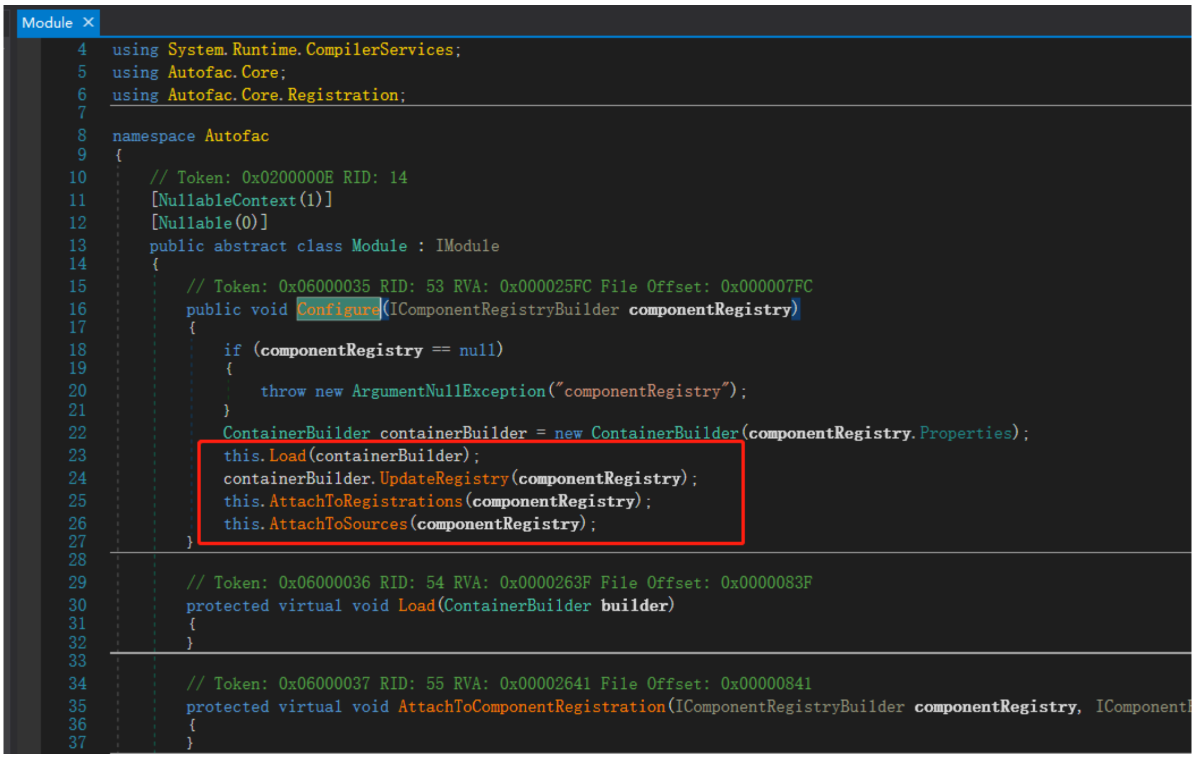
Module基本使用

```
// Register the ConfigurationModule with Autofac.
var module = new AutofacDemo.MyConfigurationModule();//实例化自定义的module实例
var builder = new ContainerBuilder();//创建容器ContainerBuilder
builder.RegisterModule(module);//注册module
IContainer container = builder.Build();//创建容器
ITestServiceB testServiceB = container.Resolve<ITestServiceB>();//获取实例
testServiceB.Hello("Hello, World!");
```

原理:ContainerBuilder调用Build()方法时,会调用到基类Autofac.Module的Configure方法，该方法会依次调用自定义类MyConfigurationModule中的以下方法

- void Load(ContainerBuilder builder)
- void AttachToComponentRegistration(IComponentRegistryBuilder componentRegistry, IComponentRegistration registration)

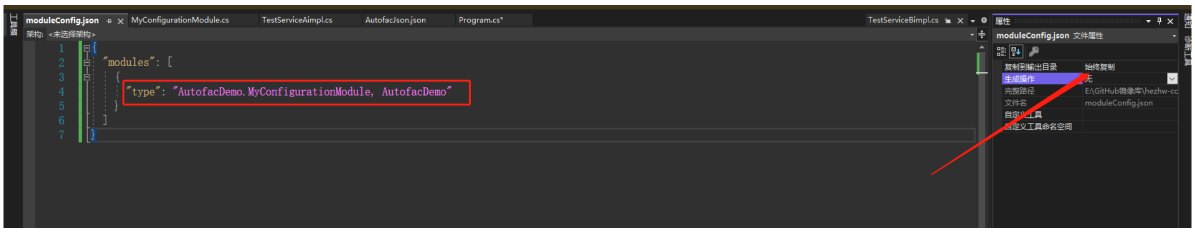
- AttachToRegistrationSource(IComponentRegistryBuilder componentRegistry, IRegistrationSource registrationSource)



```
using System.Runtime.CompilerServices;
using Autofac.Core;
using Autofac.Core.Registration;

namespace Autofac
{
    // Token: 0x0200000E RID: 14
    [NullableContext(1)]
    [Nullable(0)]
    public abstract class Module : IModule
    {
        // Token: 0x06000035 RID: 53 RVA: 0x000025FC File Offset: 0x000007FC
        public void Configure(IComponentRegistryBuilder componentRegistry)
        {
            if (componentRegistry == null)
            {
                throw new ArgumentNullException("componentRegistry");
            }
            ContainerBuilder containerBuilder = new ContainerBuilder(componentRegistry.Properties);
            this.Load(containerBuilder);
            containerBuilder.UpdateRegistry(componentRegistry);
            this.AttachToRegistrations(componentRegistry);
            this.AttachToSources(componentRegistry);
        }

        // Token: 0x06000036 RID: 54 RVA: 0x0000263F File Offset: 0x0000083F
        protected virtual void Load(ContainerBuilder builder)
        {
        }

        // Token: 0x06000037 RID: 55 RVA: 0x00002641 File Offset: 0x00000841
        protected virtual void AttachToComponentRegistration(IComponentRegistryBuilder componentRegistry, IComponent
        {
        }
```

使用配置文件配置module

Conf文件夹下新建moduleConfig.json文件



```
{
  "modules": [
    {
      "type": "AutofacDemo.MyConfigurationModule, AutofacDemo"
    }
  ]
}
```

moduleConfig.json 文件属性

复制到输出目录	始终复制
生成操作	无
完整路径	E:\GitHub镜像库\hezhw-cc
文件名	moduleConfig.json
自定义工具	
自定义工具命名空间	

使用示例

```
// 实例化ConfigurationBuilder.
var config = new Microsoft.Extensions.Configuration.ConfigurationBuilder();
//使用Microsoft.Extensions.Configuration.Json读取json配置文件
config.AddJsonFile("Conf/moduleConfig.json");

// Register the ConfigurationModule with Autofac.
var module = new Autofac.Configuration.ConfigurationModule(config.Build());//将配置文件加载至module
var builder = new ContainerBuilder();//创建ContainerBuilder
builder.RegisterModule(module);//注册服务
IContainer container = builder.Build();//创建容器
ITestServiceB testServiceB = container.Resolve<ITestServiceB>();//获取实例
testServiceB.Hello("Hello, World!");
```

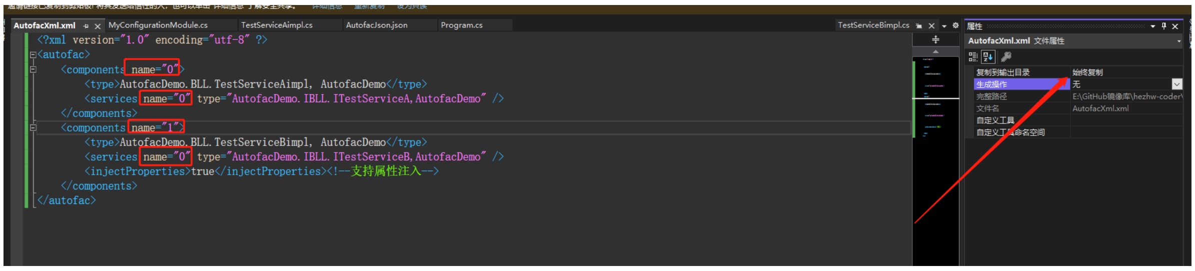
使用`xml`文件配置

nuget安装xml文件配置扩展包(例子使用6.0.0版本)

```
Install-Package Microsoft.Extensions.Configuration.Xml -Version 6.0.0
```

创建配置文件`Conf\AutofacXml.xml`

```xml
<?xml version="1.0" encoding="utf-8" ?>
<autofac>
    <components name="0">
        <type>AutofacDemo.BLL.TestServiceAimpl, AutofacDemo</type>
        <services name="0" type="AutofacDemo.IBLL.ITestServiceA,AutofacDemo" />
    </components>
    <components name="1">
        <type>AutofacDemo.BLL.TestServiceBimpl, AutofacDemo</type>
        <services name="0" type="AutofacDemo.IBLL.ITestServiceB,AutofacDemo" />
        <injectProperties>true</injectProperties><!--支持属性注入-->
    </components>
</autofac>
```



```xml
<?xml version="1.0" encoding="utf-8" ?>
<autofac>
    <components name="0">
        <type>AutofacDemo.BLL.TestServiceAimpl, AutofacDemo</type>
        <services name="0" type="AutofacDemo.IBLL.ITestServiceA,AutofacDemo" />
    </components>
    <components name="1">
        <type>AutofacDemo.BLL.TestServiceBimpl, AutofacDemo</type>
        <services name="0" type="AutofacDemo.IBLL.ITestServiceB,AutofacDemo" />
        <injectProperties>true</injectProperties><!--支持属性注入-->
    </components>
</autofac>
```

复制到输出目录	始终复制
生成操作	无
完整路径	E:\GitHub镜像库\hezhw-coder\
文件名	AutofacXml.xml
自定义工具	
自定义工具命名空间	

请注意 *XML* 中*components*和*services*的序号"命名" - 这是由于 *Microsoft.Extensions.Configuration* 处理序号集合（数组）的方式

使用示例

```csharp
// 实例化ConfigurationBuilder.
var config = new Microsoft.Extensions.Configuration.ConfigurationBuilder();
//使用Microsoft.Extensions.Configuration.Xml读取xml配置文件
config.AddXmlFile("Conf/AutofacXml.xml");

// Register the ConfigurationModule with Autofac.
var module = new Autofac.Configuration.ConfigurationModule(config.Build());//将配置文件加载至module
var builder = new ContainerBuilder();//创建ContainerBuilder
builder.RegisterModule(module);//注册服务
IContainer container = builder.Build();//创建容器
ITestServiceB testServiceB = container.Resolve<ITestServiceB>();//获取实例
testServiceB.Hello("Hello, World!");
```

AOP的实现

nuget安装Castle.Core与Autofac.Extras.DynamicProxy包

在示例中的Autofac.Extras.DynamicProxy包6.0.0版本引用的是Castle.Core包4.4.0版本，所以需引用想对应的版本

```
Install-Package Castle.Core -Version 4.4.0
```

```
Install-Package Autofac.Extras.DynamicProxy -Version 6.0.0
```

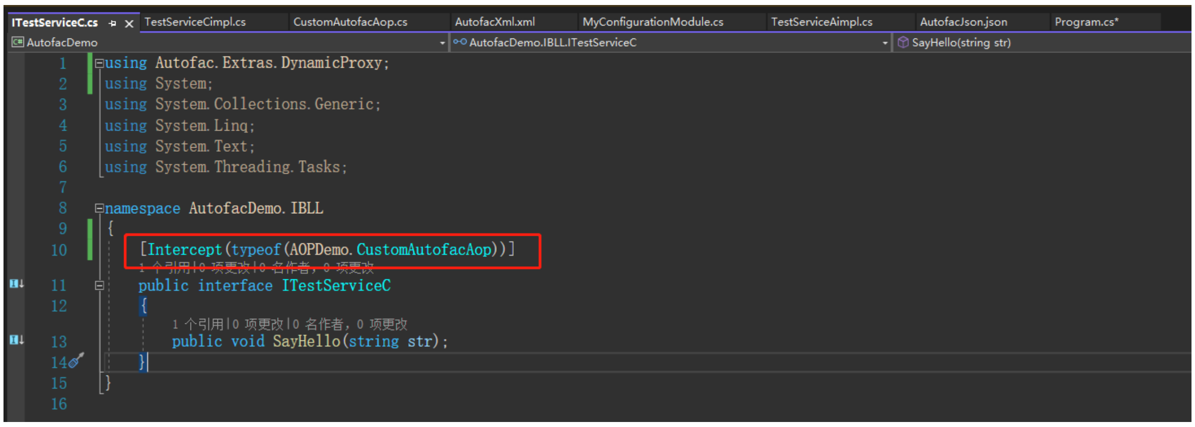
自定义切面类CustomAutofacAop实现Castle.DynamicProxy.IInterceptor接口

```
using Castle.DynamicProxy;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace AutofacDemo.AOPDemo
{
    public class CustomAutofacAop : Castle.DynamicProxy.IInterceptor
    {
        public void Intercept(IInvocation invocation)
        {
            Console.WriteLine($"{invocation.Method.Name}执行前.....");
            invocation.Proceed();
            Console.WriteLine($"{invocation.Method.Name}执行后.....");

        }
    }
}
```

接口上配置AOP

注:如果在接口上配置AOP,则实现类中的所有实现方法都会起作用

- 引用using Autofac.Extras.DynamicProxy;命名空间
- 在接口上打上InterceptAttribute特性



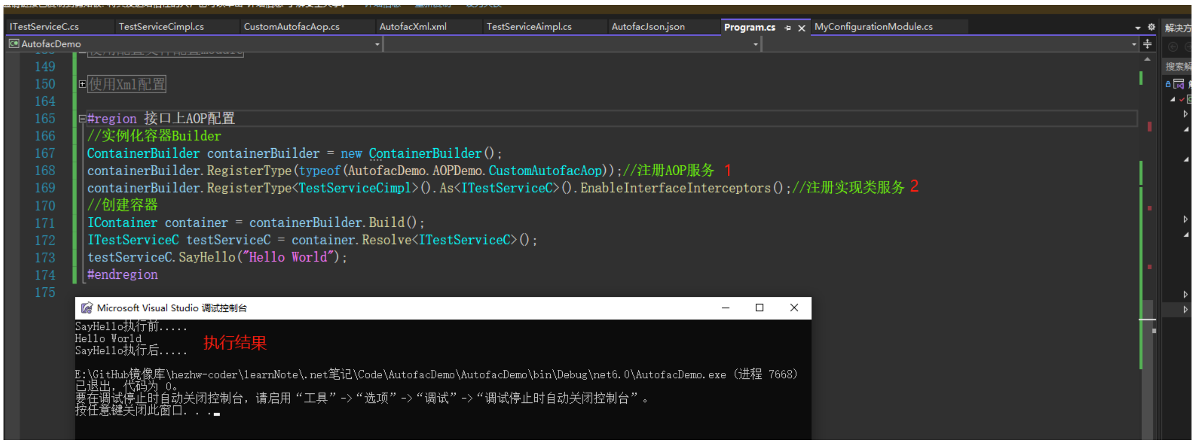
```
using Autofac.Extras.DynamicProxy;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace AutofacDemo.IBLL
{
    [Intercept(typeof(AOPDemo.CustomAutofacAop))]
    1 个引用|0 项更改|0 名作者，0 项更改
    public interface ITestServiceC
    {
        1 个引用|0 项更改|0 名作者，0 项更改
        public void SayHello(string str);
    }
}
```

注册服务时使AOP生效

- 将自定义的切面类CustomAutofacAop注入到容器中
- 注册实现类服务时使用EnableInterfaceInterceptors方法使AOP生效

代码示例

```
//实例化容器Builder
ContainerBuilder containerBuilder = new ContainerBuilder();
containerBuilder.RegisterType(typeof(AutofacDemo.AOPDemo.CustomAutofacAop));//注册AOP服务
containerBuilder.RegisterType<TestServiceCimpl>().As<ITestServiceC>().EnableInterfaceInterceptors();//注册实现类服务
//创建容器
IContainer container = containerBuilder.Build();
ITestServiceC testServiceC = container.Resolve<ITestServiceC>();
testServiceC.SayHello("Hello World");
```



```
使用Xml配置
#region 接口上AOP配置
//实例化容器Builder
ContainerBuilder containerBuilder = new ContainerBuilder();
containerBuilder.RegisterType(typeof(AutofacDemo.AOPDemo.CustomAutofacAop));//注册AOP服务 1
containerBuilder.RegisterType<TestServiceCimpl>().As<ITestServiceC>().EnableInterfaceInterceptors();//注册实现类服务 2
//创建容器
IContainer container = containerBuilder.Build();
ITestServiceC testServiceC = container.Resolve<ITestServiceC>();
testServiceC.SayHello("Hello World");
#endregion
```

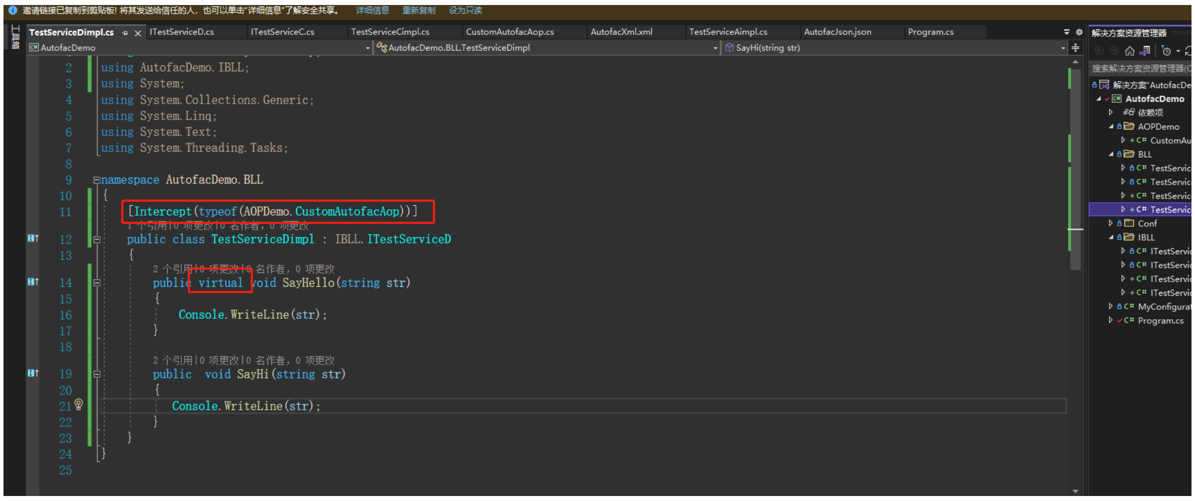
```
SayHello执行前.....
Hello World
SayHello执行后.....
```
执行结果

```
E:\GitHub镜像库\hezhw-coder\learnNote\.net笔记\Code\AutofacDemo\AutofacDemo\bin\Debug\net6.0\AutofacDemo.exe (进程 7668)
已退出，代码为 0。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口. . .
```

类上配置AOP

注:如果在类上配置AOP,则实现类中的所有的虚方法(virtual)都会起作用

- 引用using Autofac.Extras.DynamicProxy;命名空间
- 在类上打上InterceptAttribute特性



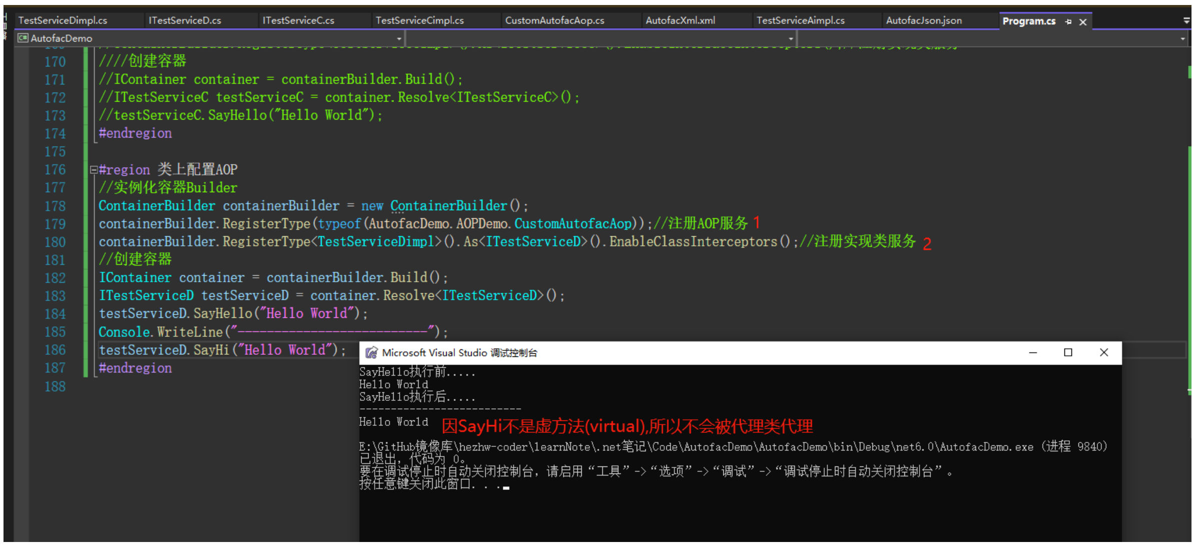
```
using AutofacDemo.IBLL;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace AutofacDemo.BLL
{
    [Intercept(typeof(AOPDemo.CustomAutofacAop))]
    public class TestServiceDimpl : IBLL.ITestServiceD
    {
        public virtual void SayHello(string str)
        {
            Console.WriteLine(str);
        }

        public  void SayHi(string str)
        {
           Console.WriteLine(str);
        }
    }
}
```

注册服务时使AOP生效

- 将自定义的切面类CustomAutofacAop注入到容器中
- 注册实现类服务时使用EnableClassInterceptors方法使AOP生效

代码示例

```csharp
//实例化容器Builder
ContainerBuilder containerBuilder = new ContainerBuilder();
containerBuilder.RegisterType(typeof(AutofacDemo.AOPDemo.CustomAutofacAop));//注册AOP服务
containerBuilder.RegisterType<TestServiceDimpl>().As<ITestServiceD>
().EnableClassInterceptors();//注册实现类服务
//创建容器
IContainer container = containerBuilder.Build();
ITestServiceD testServiceD = container.Resolve<ITestServiceD>();
testServiceD.SayHello("Hello World");
Console.WriteLine("--------------------------");
testServiceD.SayHi("Hello World");
```



```csharp
////创建容器
//IContainer container = containerBuilder.Build();
//ITestServiceC testServiceC = container.Resolve<ITestServiceC>();
//testServiceC.SayHello("Hello World");
#endregion

#region 类上配置AOP
//实例化容器Builder
ContainerBuilder containerBuilder = new ContainerBuilder();
containerBuilder.RegisterType(typeof(AutofacDemo.AOPDemo.CustomAutofacAop));//注册AOP服务 1
containerBuilder.RegisterType<TestServiceDimpl>().As<ITestServiceD>().EnableClassInterceptors();//注册实现类服务 2
//创建容器
IContainer container = containerBuilder.Build();
ITestServiceD testServiceD = container.Resolve<ITestServiceD>();
testServiceD.SayHello("Hello World");
Console.WriteLine("--------------------------");
testServiceD.SayHi("Hello World");
#endregion
```

```
SayHello执行前.....
Hello World
SayHello执行后.....
--------------------------
Hello World  因SayHi不是虚方法(virtual),所以不会被代理类代理

E:\GitHub镜像库\hezhw-coder\learnNote\.net笔记\Code\AutofacDemo\AutofacDemo\bin\Debug\net6.0\AutofacDemo.exe (进程 9840)
已退出，代码为 0。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口. . .
```