Autofac高级用法之动态代理

前言

Autofac的DynamicProxy来自老牌的Castle项目。DynamicProxy（以下称为动态代理）起作用主要是为我们的类生成一个代理类，这个代理类可以在我们调用原本类的方法之前，调用拦截器以实现AOP。那么动态代理是怎么实现的呢，这里简单一下提一下，这里主要是用了emit技术动态生成IL，相当于在内存中用IL给我们编写了一个Class。

通过静态代理实现AOP

我们新建一个类 `Cat` ，并实现 `ICat` 接口

ICat:

```
public interface ICat
{
    void Eat();
}
```

Cat:

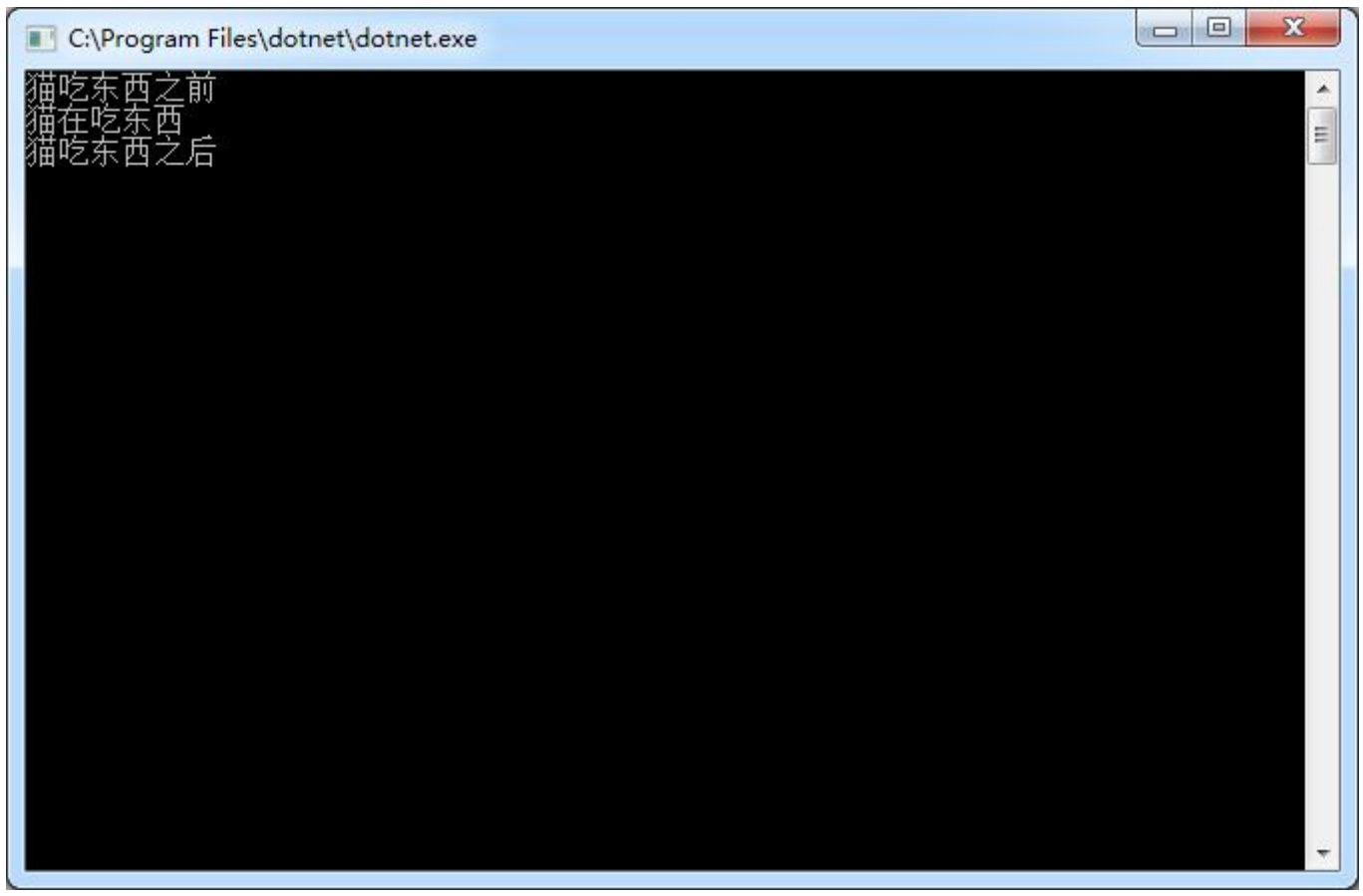
```
public class Cat:ICat
{
    public void Eat()
    {
        Console.WriteLine("猫在吃东西");
    }
}
```

然然后我们为其创建一个代理类， `CatProxy`

```
public class CatProxy:ICat
{
    private readonly ICat _cat;
    public CatProxy(ICat cat)
    {
        _cat = cat;
    }
    public void Eat()
    {
        Console.WriteLine("猫吃东西之前");
        _cat.Eat();
        Console.WriteLine("猫吃东西之后");
    }
}
```

现在我们调用一下试试效果：

```
public class Progarm
{
    static void Main(string[] args)
    {
        ICat icat=new Cat();

        var catProxy=new CatProxy(icat);

        catProxy.Eat();

        Console.Read();
    }
}
```



可以看见，我们已经成功的通过代理实现在猫吃东西之前和之后执行我们定义的代码，这就是一个简单的AOP，这个称之为静态代理，需要我们手动编写代理类，这个是十分耗费时间的，那么有什么方法帮我们自动生成代理呢，当然有了，接下来介绍我们的动态代理。

动态代理（DynamicProxy）实现AOP

我在前言中已经简单提了下动态代理的实现原理，我们这里就只说说怎么用，而不去讨论怎么实现了（烧脑阔）。我们这里使用Autofac的DynamicProxy。

我们依然使用前一章节所用的控制台项目，通过nuget安装两个Package： `Autofac` 、 `Autofac.Extras.DynamicProxy`

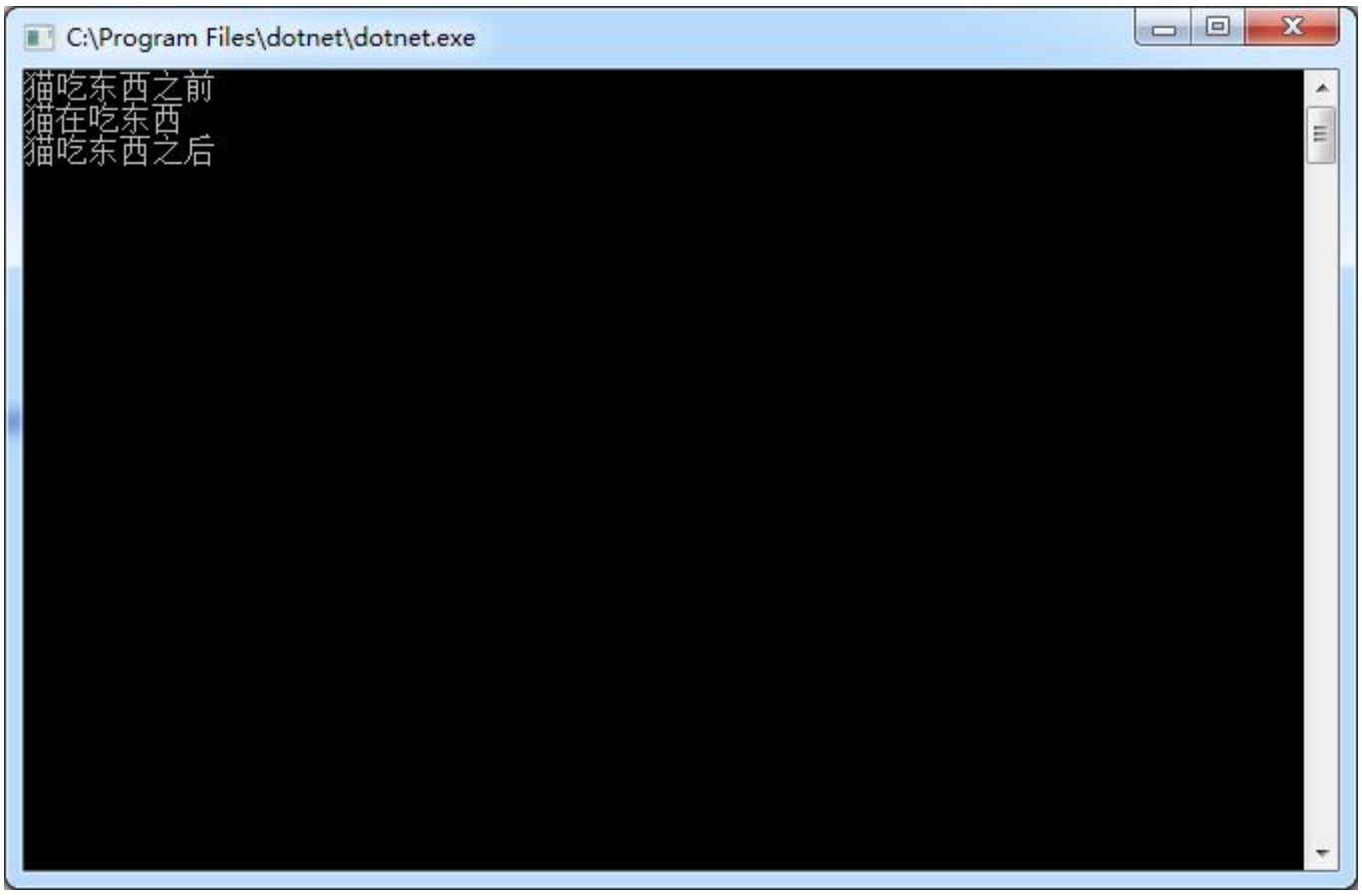
首先我们需要定义一个拦截器：

```
public class CatInterceptor:IInterceptor
{
    public void Intercept(IInvocation invocation)
    {
        Console.WriteLine("猫吃东西之前");
        invocation.Proceed();
        Console.WriteLine("猫吃东西之后");
    }
}
```

然后在Autofac容器中注册我们的拦截器和类型：

```
static void Main(string[] args)
{

    var builder = new ContainerBuilder();

    builder.RegisterType<CatInterceptor>();//注册拦截器
    builder.RegisterType<Cat>().As<ICat>().InterceptedBy(typeof(CatInterceptor)).EnableInterfaceInterceptors();//注册Cat并为其添加拦截器

    var container = builder.Build();

    var cat = container.Resolve<ICat>();

    cat.Eat();

    Console.Read();
}
```

我们运行一下看看效果：



通过运行我们可以看出，和上一章节的效果一样，但是我们并不需要取手动定义我们的代理类，而是通过组件动态生成了。

关于这个拦截器，我们还可以通过Attribute的方式绑定到我们的具体类型，而不需要在注册到容器的时候动态指定。

```
[Intercept(typeof(CatInterceptor))]
public class Cat:ICat
{
    public void Eat()
    {
        Console.WriteLine("猫在吃东西");
    }
}
```

注册的代码可改为:

```
builder.RegisterType<Cat>().As<ICat>().EnableInterfaceInterceptors();
```

动态代理的高级用法

我们前面说了，动态代理是动态生成一个代理类，那么我们可以**动态**的为这个代理类添加一个接口吗，答案当然是可以。

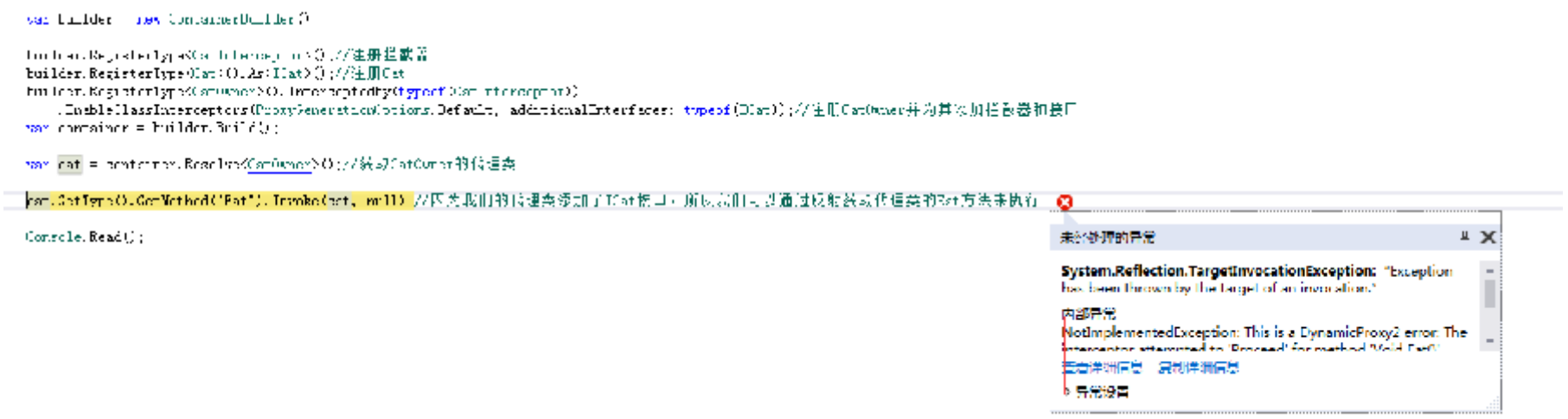
现在我们定义一个 `铲屎官` 类:

```
public class CatOwner
{

}
```

可以看出我们的 `铲屎官` 类什么都没有，如果我们的铲屎官想喂猫吃东西怎么办，按照我们传统的思维当然是实例化一个cat传入我们的 `CatOwner` ，但是我们可以用我们的DynamicProxy动态生成。

```
var builder = new ContainerBuilder();

builder.RegisterType<CatInterceptor>();//注册拦截器
builder.RegisterType<Cat>().As<ICat>();//注册Cat
builder.RegisterType<CatOwner>().InterceptedBy(typeof(CatInterceptor))
    .EnableClassInterceptors(ProxyGenerationOptions.Default, additionalInterfaces: typeof(ICat));//注册CatOwner并为其添加拦截器和接口
var container = builder.Build();

var cat = container.Resolve<CatOwner>();//获取CatOwner的代理类

cat.GetType().GetMethod("Eat").Invoke(cat, null);//因为我们的代理类添加了ICat接口，所以我们可以通过反射获取代理类的Eat方法来执行

Console.Read();
```

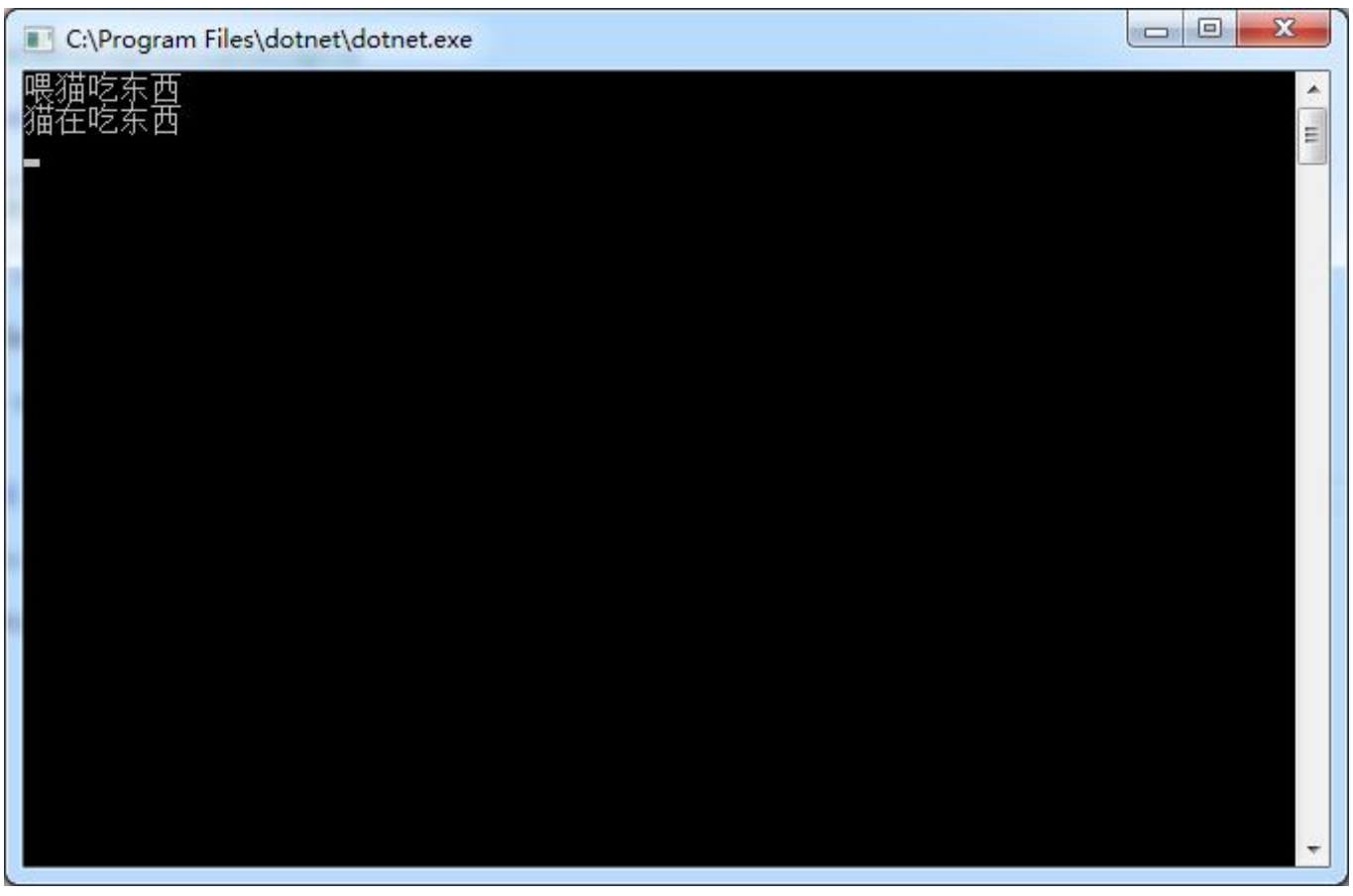
我们上面的代码是肯定不能运行的，因为我们的代理类虽然添加了ICat接口，但是却没有具体实现它，所以抛出为卫视现异常：



我们可以使用AOP在我们执行代理类的Eat方法之前去调用我们的具体实现 `Cat` 的Eat方法，我们修改一下拦截器。

```
public class CatInterceptor:IInterceptor
{
    private readonly ICat _cat;

    /// <summary>
    /// 通过依赖注入 注入ICat的具体实现
    /// </summary>
    /// <param name="cat"></param>
    public CatInterceptor(ICat cat)
    {
        _cat = cat;
    }
    public void Intercept(IInvocation invocation)
    {
        Console.WriteLine("喂猫吃东西");

        invocation.Method.Invoke(_cat, invocation.Arguments);//调用Cat的指定方法
    }
}
```

我们看一下运行效果：



可以看见我们从一个什么都没有的 `CatOwner` 类，来为其调用了一个具体的猫吃东西的行为，是不是感觉很神奇!

> 有人可能会说，一个铲屎官为什么要去实现一个ICat接口。我想说纯属胡编乱造，只是想阐明这个用法，这个意思。

应用场景

用过ABP框架的人都应该知道其有个技术名为DynamicWebapi，非常方便可以动态帮我们的应用逻辑层生成webapi，而不需要我们手动去编写webapi来发布。这里据用到了上面所说的技术，动态生成Wabpi Controller，然后为其添加应用逻辑接口，在调用具体的应用逻辑方法时（Action）通过AOP拦截调用具体应用逻辑实现来完成。

Demo:https://github.com/stulzq/BlogDemos/tree/master/AutofacDynamicProxyTest