32、JAVA动态代理

# 1、作用

主要用来做方法的增强，让你可以在不修改源码的情况下，增强一些方法，在方法执行前后做任何你想做的事情（甚至根本不去执行这个方法），因为在***InvocationHandler***的invoke方法中，你可以直接获取正在调用方法对应的***Method***对象，具体应用的话，比如可以添加调用日志，做事务控制等。包括我们Spring的AOP其就是通过动态代理的机制实现的，所以我们必须要好好的理解动态代理的机制。

 AOP的拦截功能是由java中的动态代理来实现的。说白了，就是在目标类的基础上增加切面逻辑，生成增强的目标类（该切面逻辑或者在目标类函数执行之前，或者目标类函数执行之后，或者在目标类函数抛出异常时候执行。不同的切入时机对应不同的Interceptor的种类，如BeforeAdviseInterceptor，AfterAdviseInterceptor以及ThrowsAdviseInterceptor等）。

那么动态代理是如何实现将切面逻辑（advise）织入到目标类方法中去的呢？下面我们就来详细介绍并实现AOP中用到的两种动态代理。

AOP的源码中用到了两种动态代理来实现拦截切入功能：jdk动态代理和cglib动态代理。两种方法同时存在，各有优劣。

jdk动态代理是由java内部的反射机制来实现的，java动态代理只能对接口进行代理，Java的继承机制注定了这些动态代理类们无法实现对class的动态代理。（个人理解：可以理解为继承之后父类中也可以写方法，最终不知道指向）

cglib动态代理底层则是借助asm来实现的。

总的来说，反射机制在生成类的过程中比较高效，而asm在生成类之后的相关执行过程中比较高效（可以通过将asm生成的类进行缓存，这样解决asm生成类过程低效问题）。

还有一点必须注意：jdk动态代理的应用前提，必须是目标类基于统一的接口。如果没有上述前提，jdk动态代理不能应用。由此可以看出，jdk动态代理有一定的局限性，cglib这种第三方类库实现的动态代理应用更加广泛，且在效率上更有优势。。

多继承在Java中本质上就行不通。有很多条理由，人们可以否定对 class代理的必要性，但是同样有一些理由，相信支持class动态代理会更美好。接口和类的划分，本就不是很明显，只是到了Java中才变得如此的细化。如果只从方法的声明及是否被定义来考量，有一种两者的混合体，它的名字叫抽象类。实现对抽象类的动态代理，相信也有其内在的价值。此外，还有一些历史遗留的类，它们将因为没有实现任何接口而从此与动态代理永世无缘。如此种种，不得不说是一个小小的遗憾。但是，不完美并不等于不伟大，伟大是一种本质，Java动态代理就是佐例

## 2、编辑接口

|  |
| --- |
| **public** **interface** Subject  {  **public** **void** one();    **public** **void** two();  } |

# 3、编辑代理对象，也就是上面的接口的实现类

|  |
| --- |
| **public** **class** RealSubject **implements** Subject  {  @Override  **public** **void** one() {  System.*out*.println("方法1 one");  }  @Override  **public** **void** two() {  System.*out*.println("方法2 two");    }    } |

# 4、代理的调用处理器

## 1、实现InvocationHandler 接口

## 2、构造器中添加真实的代理对象

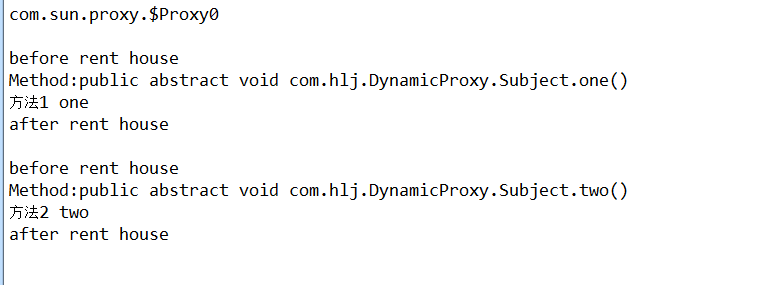
## 3、method.invoke 前后可以添加自定义的方法

|  |
| --- |
| **public** **class** ProxyHandler **implements** InvocationHandler  {  //　这个就是我们要代理的真实对象  **private** Object proxied;    // 构造方法，给我们要代理的真实对象赋初值  **public** ProxyHandler( Object proxied )  {  **this**.proxied = proxied;  }    @Override  **public** Object invoke(Object object, Method method, Object[] args)  **throws** Throwable  {  //　　在代理真实对象前我们可以添加一些自己的操作  System.*out*.println("before rent house");    System.*out*.println("Method:" + method);    // 当代理对象调用真实对象的方法时，其会自动的跳转到代理对象关联的handler对象的invoke方法来进行调用  method.invoke(proxied, args);    //　　在代理真实对象后我们也可以添加一些自己的操作  System.*out*.println("after rent house");    **return** **null**;  }    } |

# 5、动态代理测试调用

|  |
| --- |
| **public** **class** DynamicProxyMain {  **public** **static** **void** main( String args[] )  {  // 我们要代理的真实对象  Subject realSubject = **new** RealSubject();  // 我们要代理哪个真实对象，就将该对象传进去，最后是通过该真实对象来调用其方法的,只能对接口进行代理  ProxyHandler handler = **new** ProxyHandler(realSubject);  /\*  \* 通过Proxy的newProxyInstance方法来创建我们的代理对象，我们来看看其三个参数  \* 第一个参数 handler.getClass().getClassLoader() ，  我们这里使用handler这个类的ClassLoader对象来加载我们的代理对象  \* 第二个参数realSubject.getClass().getInterfaces()，  我们这里为代理对象提供的接口是真实对象所实行的接口，表示我要代理的是该真实对象，这样我就能调用这组接口中的方法了  \* 第三个参数handler， 我们这里将这个代理对象关联到了上方的 InvocationHandler 这个对象上  \*/  Subject subject = (Subject)Proxy.*newProxyInstance*(  handler.getClass().getClassLoader(),  realSubject .getClass().getInterfaces(),  handler);  /\*\*  这里才开始执行  \*/  System.*out*.println(subject.getClass().getName());  System.*out*.println();  subject.one();  System.*out*.println();  subject.two();  }    } |

## 1、控制台



## 6、解释控制台原因

### 1、代码System.*out*.println(subject.getClass().getName()); 控制台结果为：com.sun.proxy.$Proxy0

### 1、为什么我们这里可以将其转化为Subject类型的对象？

原因就是在newProxyInstance这个方法的第二个参数上，我们给这个代理对象提供了一组什么接口，那么我这个代理对象就会实现了这组接口，这个时候我们当然可以将这个代理对象强制类型转化为这组接口（数组）中的任意一个，因为这里的接口是Subject类型，所以就可以将其转化为Subject类型了。

**同时我们一定要记住，通过 Proxy.newProxyInstance 创建的代理对象是在jvm运行时动态生成的一个对象，它并不是我们的InvocationHandler类型，也不是我们定义的那组接口的类型，而是在运行是动态生成的一个对象，**

**并且命名方式都是这样的形式，以$开头，proxy为中，最后一个数字表示对象的标号**。