**Proxy InvocationHandler**

**AOP概述**

软件的编程语言最终的目的就是用更自然更灵活的方式模拟世界，从原始机器语言到过程语言再到面向对象的语言，我们看到编程语言在一步步用更自然、更 强大的方式描述软件。AOP是软件开发思想的一个飞跃，AOP的引入将有效弥补OOP的不足，OOP和AOP分别从纵向和横向对软件进行抽象，有效地消除 重复性的代码，使代码以更优雅的更有效的方式进行逻辑表达。

AOP有三种植入切面的方法：

**1.是编译期织入，这要求使用特殊的Java编译器，AspectJ是其中的代表者；2.是类装载期织入，而这要求使 用特殊的类装载器，AspectJ和AspectWerkz是其中的代表者；**

**3.为动态代理织入，在运行期为目标类添加增强生成子类的方式，Spring AOP采用动态代理织入切面。**

Spring AOP使用了两种代理机制，

**一种是基于JDK的动态代理，**

**另一种是基于CGLib的动态代理，**

之所以需要两种代理机制，很大程度上是因为JDK本身**只提供基于接口的代理，不支持类的代理。**

一、原理区别：

java动态代理是利用反射机制生成一个实现代理接口的匿名类，**在调用具体方法前调用InvokeHandler来处理（生成动态代理对象）。**

而cglib动态代理是利用asm开源包，对代理对象类的class文件加载进来，**通过修改其字节码生成子类来处理（子类也是动态代理对象）。**

1、如果**目标对象实现了接口**，默认情况下会采用JDK的动态代理实现AOP   
2、如果**目标对象实现了接口**，可以强制使用CGLIB实现AOP   
3、如果**目标对象没有实现了接口**（只有目标类），必须采用CGLIB库，spring会自动在JDK动态代理和CGLIB之间转换

如何强制使用CGLIB实现AOP？  
 （1）添加CGLIB库，SPRING\_HOME/cglib/\*.jar  
 （2）在spring配置文件中加入<aop:aspectj-autoproxy proxy-target-class="true"/>  
  
JDK动态代理和CGLIB字节码生成的区别？  
 （1）JDK动态代理只能**对实现了接口的类生成代理**，而不能针对类  
 （2）CGLIB是**针对类实现代理**，主要是**对指定的类生成一个子类，覆盖其中的方法**   因为是继承，所以该类或方法最好不要声明成final

**基于JDK的代理和基于CGLib的代理**是Spring AOP的核心实现技术，认识这两代理技术，有助于探究Spring AOP的实现机理。

**前置通知[Before advice]**：在连接点前面执行，前置通知不会影响连接点的执行，除非此处抛出异常。   
**正常返回通知[After returning advice]**：在连接点正常执行完成后执行，如果连接点抛出异常，则不会执行。   
**异常返回通知[After throwing advice]：**在连接点抛出异常后执行。   
**返回通知[After (finally) advice]**：在连接点执行完成后执行，不管是正常执行完成，还是抛出异常，都会执行返回通知中的内容。   
**环绕通知[Around advice]：**环绕通知围绕在连接点前后，比如一个方法调用的前后。这是最强大的通知类型，能在方法调用前后自定义一些操作。环绕通知还需要负责决定是继续处理join point(调用ProceedingJoinPoint的proceed方法)还是中断执行。

***前置通知→环绕通知→正常返回通知/异常返回通知→返回通知***

上面就是AOP的核心概念了。总结一下：

1、横切关注点

对哪些方法进行拦截，拦截后怎么处理，这些关注点称之为横切关注点

2、切面（aspect）

类是对物体特征的抽象，切面就是对横切关注点的抽象

3、连接点（joinpoint）

被拦截到的点，因为Spring只支持方法类型的连接点，所以在Spring中连接点指的就是被拦截到的方法，实际上连接点还可以是字段或者构造器

4、切入点（pointcut）

对连接点进行拦截的定义

5、通知（advice）

所谓通知指的就是指拦截到连接点之后要执行的代码，通知分为前置、后置、异常、最终、环绕通知五类

6、目标对象

代理的目标对象

7、织入（weave）

将切面应用到目标对象并导致代理对象创建的过程

8、引入（introduction）

在不修改代码的前提下，引入可以在**运行期**为类动态地添加一些方法或字段

**AOP要做的事情就是：生成代理对象，然后织入。**

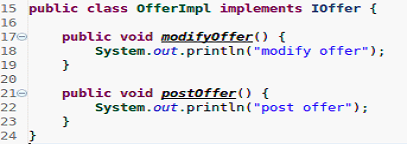
**现在，跟着我的脚步，先举个例子，由浅入深来看看SpringAOP到底是怎么实现的吧：**

举个例子来说明一下吧！现在系统中有很多的业务方法，如上传产品信息、修改产品信息、发布公司库等；现在需要对这些方法的执行做性能监控，看每个业务方法的执行时间；在不改变原业务代码的基础上，也许我们会这么做：

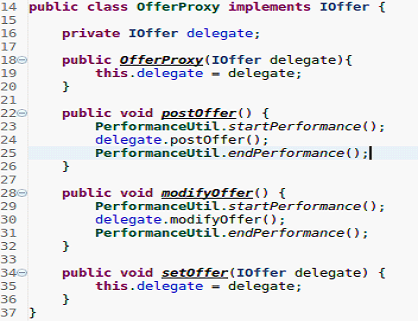
Offer接口：



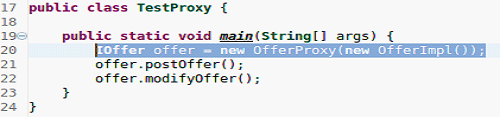
Offer实现：



Offer代理：



我们要通过下面的方式来使用：



上面的例子的输出为：

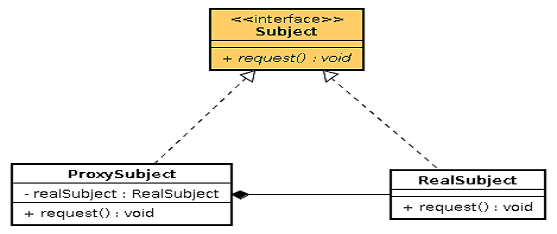
http://www.uml.org.cn/j2ee/images/2013011025.png

上面的例子中，OfferProxy实现了IOffer，而**所有的业务实现均委托给其成员offer**；可以想像，这应该就是最简单的AOP的实现了；但这种方式会存在一个问题：如果有非常多的这种业务对象需要性能监控，我们就需要写同样多的XyzProxy来满足需求，这也是非常巨大的工作量。

**上面说到了代理，我们先看看代理模式吧！**

**二、代理模式及实现**

下面是代理模式的类图：



代理模式类图

代理模式中，存在一个称为ProxyObject的**代理对象**和RealObject的**真实对象**，它们**都实现了相同的接口**；在调用的地方持有ProxyObject的实例，当调用**request()方法时**，**ProxyObject**可以在执行**RealObject.request()前后做一些特定的业务，甚至不调用RealObject.request()方法(但是这样就不叫代理了嘛~应该是调用真实对象的方法，然后在调用真实对象的前后生成一些特定的业务。)。**

代理类在程序运行时创建的代理方式被成为 动态代理。 也就是说，这种情况下，**代理类**（动态生成）并不是在Java代码中定义的，而是在运行时根据我们在Java代码中的“指示”动态生成的。相比于静态代理， **动态代理的优势在于可以很方便的对代理类的函数进行统一的处理，而不用修改每个代理类的函数。**

目前SpringAOP实现代理模式的方式有两种：**基于JDK的动态代理和基于CGLIB字节码的代理。**

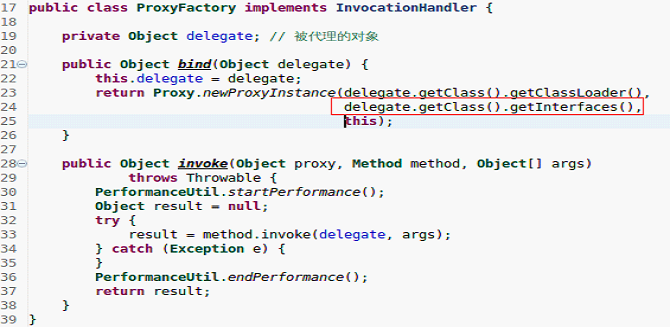
**2.1 JDK动态代理**

JDK动态代理，顾名思义，是**基于JDK的反射(reflect)机制**；在JDK中，**提供了InvocationHandler这个接口**，下面是JDK里面的注释：

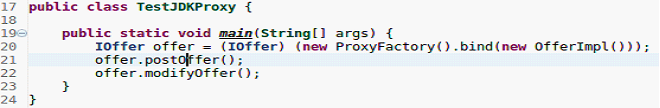
编写切面逻辑类，必须要实现InvocationHandler 接口。然后重写invoke方法，可以写入before和after等特定业务操作。

是不是有一种豁然开朗的感觉呢？没错，答案就在你心中。

这样，上面的代码就可以改成下面的实现方式：



调用端：

http://www.uml.org.cn/j2ee/images/2013011025.png

**一样的结果，这里已经通过实现了invocationHandler的接口，直接生成了代理的对象，并且跑的每个方法，都是调用了重写的invoke方法，自然而然方法里面的before和after的特定业务也会被实现。（当然这儿生成代理对象，需要接口有实现类。如果IOffer没有实现类，仅仅只有一个接口，就要用CGLIB实现方式。）**

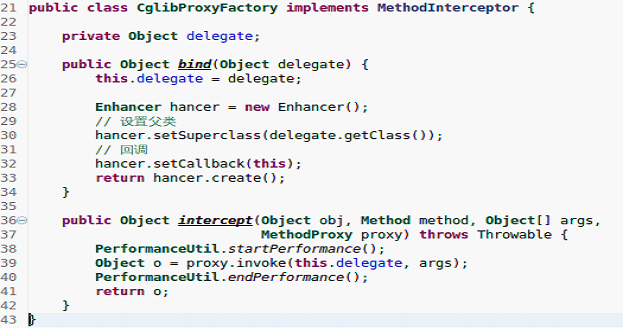
通过这种方式，你不需要为针对每一个业务写一个代理对象，就可以很轻松地完成你的需求；但也许你已经注意到了，**JDK的动态代理，在创建代理对象(上面红色代码部分)时，被代理的对象需要实现接口(即面向接口编程)；**

这就是JDK的动态代理，简单吧！下面看看CGLIB代理方式。

**2.2 CGLIB代理**

**如果目标对象没有实现任何接口，那怎么办呢？**不用担心，你可以用CGLIB来实现代理：**实现动态代理类CglibProxy，需要实现MethodInterceptor接口，实现intercept方法。**该代理中在add方法前后加入了自定义的切面逻辑，目标类add方法执行语句为proxy.invokeSuper(object, args);

获取增强的目标类的工厂Factory，其中增强的方法类对象是有Enhancer来实现的，

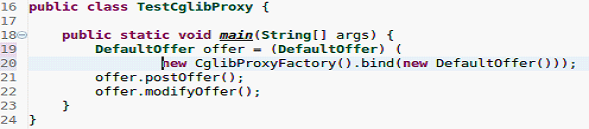


上面的Enhancer返回一个delegate的子类实例，就是使用动态代理的对象**必须实现一个或多个接口 。如果想代理没有实现接口的类可以使用CGLIB包。**也就是说无实现接口的类想被代理的话，直接帮你通过这个Enhancer产生一个实例对象。继承了delegate所有的方法。

  //回调方法的参数为代理类对象CglibProxy，最后增强目标类调用的是代理类对象CglibProxy中的intercept方法

// 下一句的create不是单纯的目标类，而是增强过的目标类

调用端：



其中DefaultOffer为增强过的目标类。它是为没有实现接口的类，此时可以通过CGlib动态代理生成其子类，继承其所有的方法，然后再生成动态代理的类。

使用CGLIB创建的代理对象，其实比JDK动态代理多了一个步骤：就是继承了要代理的目标类，然后对目标类中所有非final方法进行覆盖，但在覆盖方法时会添加一些拦截代码(上面CglibProxyFactory类中的intercept方法，**也就是增强，实现特定业务**)。

当然，以上的例子都是自己实现的动态代理对象的代码，比源码省去了很多的东西，下一篇我们将开始学习一下源码里的一些东西。