**AOP（Aspect-OrientedProgramming，面向方面编程），可以说是OOP（Object-Oriented Programing，面向对象编程）的补充和完善。**

****什么是AOP****

**AOP（Aspect-OrientedProgramming，面向方面编程），可以说是OOP（Object-Oriented Programing，面向对象编程）的补充和完善。OOP引入封装、继承和多态性等概念来建立一种对象层次结构，用以模拟公共行为的一个集合。当我们需要为分散的对象引入公共行为的时候，OOP则显得无能为力。也就是说，OOP允许你定义从上到下的关系，但并不适合定义从左到右的关系。例如日志功能。日志代码往往水平地散布在所有对象层次中，而与它所散布到的对象的核心功能毫无关系。对于其他类型的代码，如安全性、异常处理和透明的持续性也是如此。这种散布在各处的无关的代码被称为横切（cross-cutting）代码，在OOP设计中，它导致了大量代码的重复，而不利于各个模块的重用。**

****简介****

**前段时间写的java设计模式--代理模式，最近在看Spring Aop的时候，觉得于代理模式应该有密切的联系，于是决定了解下Spring Aop的实现原理。**

**说起AOP就不得不说下OOP了，OOP中引入封装、继承和多态性等概念来建立一种对象层次结构，用以模拟公共行为的一个集合。但是，如果我们需要为部分对象引入公共部分的时候，OOP就会引入大量重复的代码。例如：日志功能。**

**AOP技术利用一种称为“横切”的技术，解剖封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，这样就能减少系统的重复代码，降低模块间的耦合度，并有利于未来的可操作性和可维护性。AOP把软件系统分为两个部分：核心关注点和横切关注点。业务处理的主要流程是核心关注点，与之关系不大的部分是横切关注点。横切关注点的一个特点是，他们经常发生在核心关注点的多处，而各处都基本相似。比如权限认证、日志、事务处理。**

****实现原理****

**前面在学习代理模式的时候，了解到代理模式分为动态代理和静态代理。现在我们就以代理模式为基础先实现我们自己的AOP框架，再来研究Spring的AOP的实现原理。**

**先以静态代理实现，静态代理关键是在代理对象和目标对象实现共同的接口，并且代理对象持有目标对象的引用。**

****公共接口代码：****

**[?](http://www.jb51.net/article/81788.htm)**

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15** | **public interface IHello {**  **/\*\***  **\* 业务方法**  **\* @param str**  **\*/**  **void sayHello(String str);**  **}**  **目标类代码：**    **public class Hello implements IHello{**  **@Override**  **public void sayHello(String str) {**  **System.out.println("hello "+str);**  **}**  **}** |

**代理类代码，我们给它添加日志记录功能，在方法开始前后执行特定的方法，是不是和AOP特别像呢？**

**[?](http://www.jb51.net/article/81788.htm)**

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13** | **public class ProxyHello implements IHello{**  **private IHello hello;**  **public ProxyHello(IHello hello) {**  **super();**  **this.hello = hello;**  **}**  **@Override**  **public void sayHello(String str) {**  **Logger.start();//添加特定的方法**  **hello.sayHello(str);**  **Logger.end();**  **}**  **}** |

****日志类代码：****

**[?](http://www.jb51.net/article/81788.htm)**

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8** | **public class Logger {**  **public static void start(){**  **System.out.println(new Date()+ " say hello start...");**  **}**  **public static void end(){**  **System.out.println(new Date()+ " say hello end");**  **}**  **}** |

****测试代码：****

**[?](http://www.jb51.net/article/81788.htm)**

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7** | **public class Test {**  **public static void main(String[] args) {**  **IHello hello = new ProxyHello(new Hello());//如果我们需要日志功能，则使用代理类**  **//IHello hello = new Hello();//如果我们不需要日志功能则使用目标类**  **hello.sayHello("明天");**  **}**  **}** |

**这样我们就实现了一个最简单的AOP，但是这样会存在一个问题：如果我们像Hello这样的类很多，那么，我们是不是要去写很多个HelloProxy这样的类呢。其实也是一种很麻烦的事。在jdk1.3以后，jdk跟我们提供了一个API java.lang.reflect.InvocationHandler的类， 这个类可以让我们在JVM调用某个类的方法时动态的为些方法做些什么事。下面我们就来实现动态代理的实现。**

**动态代理实现主要是实现InvocationHandler，并且将目标对象注入到代理对象中，利用反射机制来执行目标对象的方法。**

**接口实现与静态代理相同，代理类代码：**

**[?](http://www.jb51.net/article/81788.htm)**

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18** | **public class DynaProxyHello implements InvocationHandler{**  **private Object target;//目标对象**  **/\*\***  **\* 通过反射来实例化目标对象**  **\* @param object**  **\* @return**  **\*/**  **public Object bind(Object object){**  **this.target = object;**  **return Proxy.newProxyInstance(this.target.getClass().getClassLoader(), this.target.getClass().getInterfaces(), this);**  **}**  **@Override**  **public Object invoke(Object proxy, Method method, Object[] args)**  **throws Throwable {**  **Object result = null;**  **Logger.start();//添加额外的方法**  **//通过反射机制来运行目标对象的方法**  **result = method.invoke(this.target, args);**  **Logger.end();**  **return result;**  **}**  **}** |

****测试类代码：****

**[?](http://www.jb51.net/article/81788.htm)**

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7** | **public class DynaTest {**  **public static void main(String[] args) {**  **IHello hello = (IHello) new DynaProxyHello().bind(new Hello());//如果我们需要日志功能，则使用代理类**  **//IHello hello = new Hello();//如果我们不需要日志功能则使用目标类**  **hello.sayHello("明天");**  **}**  **}** |

**看完上面的代码可能和Spring AOP相比有一个问题，日志类只能在方法前后打印，但是AOP应该是可以在满足条件就可以执行，所有是否可以将DynaPoxyHello对象和日志操作对象(Logger)解耦呢？**

**看下面代码实现，将将DynaPoxyHello对象和日志操作对象(Logger)解耦：**

**我们要在被代理对象的方法前面或者后面去加上日志操作代码(或者是其它操作的代码)，那么，我们可以抽象出一个接口，这个接口里就只有两个方法：一个是在被代理对象要执行方法之前执行的方法,我们取名为start，第二个方法就是在被代理对象执行方法之后执行的方法,我们取名为end。**

****Logger的接口：****

**[?](http://www.jb51.net/article/81788.htm)**

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4** | **public interface ILogger {**  **void start(Method method);**  **void end(Method method);**  **}** |

****Logger的接口实现：****

**[?](http://www.jb51.net/article/81788.htm)**

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10** | **public class DLogger implements ILogger{**  **@Override**  **public void start(Method method) {**  **System.out.println(new Date()+ method.getName() + " say hello start...");**  **}**  **@Override**  **public void end(Method method) {**  **System.out.println(new Date()+ method.getName() + " say hello end");**  **}**  **}** |

****动态代理类：****

**[?](http://www.jb51.net/article/81788.htm)**

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23** | **public class DynaProxyHello implements InvocationHandler{**  **//调用对象**  **private Object proxy;**  **//目标对象**  **private Object target;**  **public Object bind(Object target,Object proxy){**  **this.target=target;**  **this.proxy=proxy;**  **return Proxy.newProxyInstance(this.target.getClass().getClassLoader(), this.target.getClass().getInterfaces(), this);**  **}**  **@Override**  **public Object invoke(Object proxy, Method method, Object[] args)**  **throws Throwable {**  **Object result = null;**  **//反射得到操作者的实例**  **Class clazz = this.proxy.getClass();**  **//反射得到操作者的Start方法**  **Method start = clazz.getDeclaredMethod("start", new Class[]{Method.class});**  **//反射执行start方法**  **start.invoke(this.proxy, new Object[]{this.proxy.getClass()});**  **//执行要处理对象的原本方法**  **method.invoke(this.target, args);**  **//反射得到操作者的end方法**  **Method end = clazz.getDeclaredMethod("end", new Class[]{Method.class});**  **//反射执行end方法**  **end.invoke(this.proxy, new Object[]{method});**  **return result;**  **}**  **}** |

****测试代码：****

**[?](http://www.jb51.net/article/81788.htm)**

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7** | **public class DynaTest {**  **public static void main(String[] args) {**  **IHello hello = (IHello) new DynaProxyHello().bind(new Hello(),new DLogger());//如果我们需要日志功能，则使用代理类**  **//IHello hello = new Hello();//如果我们不需要日志功能则使用目标类**  **hello.sayHello("明天");**  **}**  **}** |

**通过上面例子，可以发现通过动态代理和发射技术，已经基本实现了AOP的功能，如果我们只需要在方法执行前打印日志，则可以不实现end()方法，这样就可以控制打印的时机了。如果我们想让指定的方法打印日志，我们只需要在invoke（）方法中加一个对method名字的判断，method的名字可以写在xml文件中，这样我们就可以实现以配置文件进行解耦了，这样我们就实现了一个简单的spring aop框架。**