https://www.cnblogs.com/exolution/archive/2012/10/24/2736597.html

[不得不说的AOP——基于Javassist的动态代理实现](https://www.cnblogs.com/exolution/archive/2012/10/24/2736597.html)

【以前的文章，从我的其他博客搬来】

关键词：AOP、SOC、代理模式、动态代理。

上一话，我提到了一个重要的概念就是SOC，即关注点分离。为什么要分离关注点，因为这样可以让我们更加简单而富有条理的去处理这个繁杂的世界。AOP就是SOC的一种体现。

什么是AOP？

AOP全程是Aspect Oriented Programming意即面向切面编程。他并不是什么OOP的替代技术，只是OOP的一种延续。使用SOC的思想，去解耦，去降低问题的复杂度。那么，为什么要叫做面向切面编程呢？想想一下，当你在软件开发设计类的时候，会不会发现，有些逻辑或者功能，是每个类或者说大多数类都需要完成的呢？比如异常/错误处理，比如系统日志记录，比如持久化中的事务处理。与其一个一个的实现，为什么不能把他们单独提取出来加以处理呢。如果说把这些类平行的放在一起。这些公共的部分刚好并排在一起，能够一刀切下。可能这就是切面概念的由来，呵呵。AOP就是为了解决上述问题而出现的。

代理模式

为什么要提代理模式。因为AOP的广泛实现都是通过动态代理，而动态代理又不得不说代理模式。

代理模式，顾名思义，就是对一个类的访问，变为访问这个类的代理人。经由代理再访问这个类。（代理与被代理的类实现了相同的接口，因此客户感觉不到通过代理访问这个类和直接访问这个类的区别）

为什么需要代理呢，因为一个良好的设计不应该轻易的修改。这正是开闭原则的体现：一个良好的设计应该对修改关闭，对扩展开放。而代理正是为了扩展类而存在的。他可以控制对现有类（就是需要被代理的类）服务的访问，通俗的解释就是 可以拦截对于现有类方法的调用并做些处理。

而动态代理，是指在运行期动态的为指定的类生成其代理类。（需要相关的运行时编译技术，）

AOP如Spring的AOP实现就是以这种方式实现的。他使用动态生成的代理类拦截了现有类的“切点”。并进行控制，使得这些切面的逻辑完全与该类脱离，实现了关注点分离。

下面附上我用Javassist实现的简单动态代理。（Javassist是一个运行时编译库，他能动态的生成或修改类的字节码，类似的有ASM和CGLIB，大多数框架就是基于后者实现的）

Java代码

1. **package** com.lifewool;
3. **import** javassist.CtMethod;
4. **import** javassist.ClassPool;
5. **import** javassist.CtClass;
6. **import** javassist.CtConstructor;
7. **import** javassist.CtField;
8. **import** javassist.CannotCompileException;
9. **import** javassist.NotFoundException;
11. /\*\*
12. \* 基于Javassist动态生成字节码实现简单的动态代理
13. \* Dynamic Proxy based on Javassist
14. \* @author:godsong
15. \* @version:1.01 2012/3/16
16. \* \*/
17. **public** **class** DProxy {
18. /\*\*
19. \* 动态生成的代理类名前缀
20. \* prefix name for Proxy
21. \* \*/
22. **private** **static** **final** String PROXY\_CLASS\_NAME=".Gproxy$";
23. /\*\*
24. \* 代理类名索引 用于标示一个唯一的代理类（具体的代理类名为Gproxy$n）
25. \* index for generate a unique proxy class
26. \* \*/
27. **private** **static** **int** proxyIndex=1;
28. /\*\*
29. \* 代理拦截器(利用继承减少动态构造的字节码)
30. \* Proxy interceptor(desingn for inherit)
31. \* \*/
32. **protected** Interceptor interceptor;
33. /\*\*
34. \* Prohibit instantiation
35. \* 利用私有构造函数阻止该类实例化
36. \* \*/
37. **private** DProxy(){}
38. **protected** DProxy(Interceptor interceptor){
39. **this**.interceptor=interceptor;
40. }
41. /\*\*
42. \* 创建动态代理的工厂方法
43. \* static factory method for create proxy
44. \* @param targetClass :被代理的类型
45. \* @param interceptor 拦截器实例
46. \* @return 返回动态代理实例
47. \*   它实现了targerClass的所有接口。
48. \*   因此可以向上转型为这些之中的任意接口
49. \* \*/
50. **public** **static** Object createProxy(Class<?> targetClass,Interceptor interceptor){
51. **int** index=0;
52. /\*获得运行时类的上下文\*/
53. ClassPool pool=ClassPool.getDefault();
54. /\*动态创建代理类\*/
55. CtClass proxy=pool.makeClass(targetClass.getPackage().getName()+PROXY\_CLASS\_NAME+proxyIndex++);
57. **try**{
58. /\*获得DProxy类作为代理类的父类\*/
59. CtClass superclass=pool.get("com.lifewool.DProxy");
60. proxy.setSuperclass(superclass);
61. /\*获得被代理类的所有接口\*/
62. CtClass[] interfaces=pool.get(targetClass.getName()).getInterfaces();
63. **for**(CtClass i:interfaces){
64. /\*动态代理实现这些接口\*/
65. proxy.addInterface(i);
66. /\*获得结构中的所有方法\*/
67. CtMethod[] methods=i.getDeclaredMethods();
68. **for**(**int** n=0;n<methods.length;n++){
69. CtMethod m=methods[n];
70. /\*构造这些Method参数 以便传递给拦截器的interceptor方法\*/
71. StringBuilder fields=**new** StringBuilder();
72. fields.append("private static java.lang.reflect.Method method"+index);
73. fields.append("=Class.forName(\"");
74. fields.append(i.getName());
75. fields.append("\").getDeclaredMethods()[");
76. fields.append(n);
77. fields.append("];");
78. /\*动态编译之\*/
79. CtField cf=CtField.make(fields.toString(), proxy);
80. proxy.addField(cf);
81. GenerateMethods(pool,proxy,m,index);
82. index++;
83. }
84. }
85. /\*创建构造方法以便注入拦截器\*/
86. CtConstructor cc=**new** CtConstructor(**new** CtClass[]{pool.get("com.lifewool.Interceptor")}, proxy);
87. cc.setBody("{super($1);}");
88. proxy.addConstructor(cc);
89. //proxy.writeFile();
90. **return** proxy.toClass().getConstructor(Interceptor.**class**).newInstance(interceptor);
91. }**catch**(Exception e){
92. e.printStackTrace();
93. **return** **null**;
94. }
95. }
96. /\*\*
97. \* 动态生成生成方法实现（内部调用）
98. \* \*/
99. **private** **static** **void** GenerateMethods(ClassPool pool,CtClass proxy,CtMethod method,**int** index){
101. **try** {
102. CtMethod cm=**new** CtMethod(method.getReturnType(), method.getName(), method.getParameterTypes(), proxy);
103. /\*构造方法体\*/
104. StringBuilder mbody=**new** StringBuilder();
105. mbody.append("{super.interceptor.intercept(this,method");
106. mbody.append(index);
107. mbody.append(",$args);}");
108. cm.setBody(mbody.toString());
109. proxy.addMethod(cm);
110. } **catch** (CannotCompileException e) {
111. e.printStackTrace();
112. }
113. **catch** (NotFoundException e){
114. e.printStackTrace();
115. }
116. }
117. }
118. 客户端代码
119. **public** **interface** Interface {
120. **void** Action(**int** a);
121. }
122. **class** clazz **implements** Interface{
124. @Override
125. **public** **void** Action(**int** a) {
126. System.out.println("do Action"+a);
127. }
128. }
129. **class** MyInterceptor **implements** Interceptor{
131. Object proxyed;
132. **public** MyInterceptor(Object i){
133. proxyed=i;
134. }
135. @Override
136. **public** **int** intercept(Object instance, Method method, Object[] Args) {
137. **try** {
138. System.out.println("before action");
139. method.invoke(**this**.proxyed, Args);
140. System.out.println("after action");
141. } **catch** (IllegalArgumentException e) {
142. e.printStackTrace();
143. } **catch** (IllegalAccessException e) {
144. e.printStackTrace();
145. } **catch** (InvocationTargetException e) {
146. e.printStackTrace();
147. }
148. **return** 0;
149. }
150. }
151. **public** **class** test {
152. /\*\*
153. \* @param args
154. \*/
155. **public** **static** **void** main(String[] args) {
156. clazz c=**new** clazz();
157. Interface i=(Interface)DProxy.createProxy(clazz.**class**, **new** MyInterceptor(c));
158. i.Action(123);
159. }
161. }

结果：

before action

do Action123

after action