博主的更多文章>>

原创 （Dynamic Proxy）动态代理模式的Java实现

2010-06-15 20:23:07

标签：类加载器 反射 Dynamic Proxy 休闲 动态代理

动态代理（Dynamic Proxy）：相比前一篇文章所实现的静态代理，动态代理具有更强的灵活性，因为它不用在我们设计实现的时候就指定某一个代理类来代理哪一个被代理对象，我们可以把这种指定延迟到程序运行时由JVM来实现。

我们知道，所谓代理，就是需要代理类和被代理类有相同的对外接口或者说成服务，所以代理类一般都必须实现了所有被代理类已实现的接口，因为接口就是制定了一系列对外服务的标准。

正因为动态代理有这样灵活的特性，所以我们在设计动态代理类（DynamicProxy）时不用显式地让它实现与真实主题类（RealSubject）相同的接口（interface），而是把这种实现推迟到运行时。

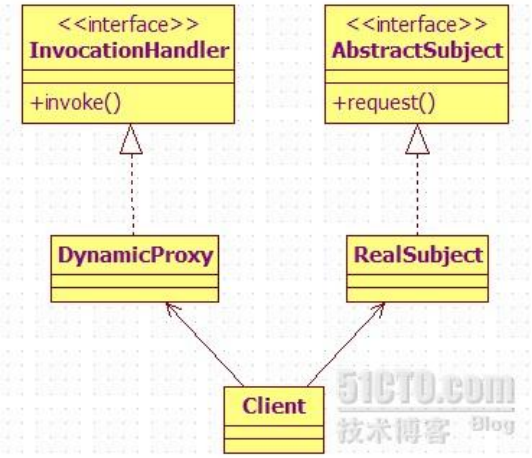
为了能让DynamicProxy类能够在运行时才去实现RealSubject类已实现的一系列接口并执行接口中相关的方法操作，需要让DynamicProxy类实现JDK自带的java.lang.reflect.InvocationHandler接口，该接口中的invoke()方法能够让DynamicProxy实例在运行时调用被代理类的“对外服务”，即调用被代理类需要对外实现的所有接口中的方法，也就是完成对真实方法的调用，Java帮助文档中称这些真实方法为处理程序。

按照上面所述，我们肯定必须先把被代理类RealSubject已实现的所有interface都加载到JVM中，不然JVM怎么能够找到这些方法呢？明白了这个道理，那么我们就可以创建一个被代理类的实例，获得该实例的类加载器ClassLoader。

所谓的类加载器ClassLoader，就是具有某个类的类定义，即类的内部相关结构（包括继承树、方法区等等）。

更重要的是，动态代理模式可以使得我们在不改变原来已有的代码结构的情况下，对原来的“真实方法”进行扩展、增强其功能，并且可以达到控制被代理对象的行为的目的。请详看下面代码中的DynamicProxy类，其中必须实现的invoke()方法在调用被代理类的真实方法的前后都可进行一定的特殊操作。这是动态代理最明显的优点。

虽然都是根据自己看了书之后的理解说了这么多，不知道能不能让人明白，这里先给出动态代理的类图吧，如下：



具体代码实现如下：

```
01. import java.lang.reflect.InvocationHandler;
02. import java.lang.reflect.Method;
03. import java.lang.reflect.Proxy;
04.
05. //抽象主题类,这里不能用abstract抽象类，一定要是interface
06. interface AbstractSubject {
07.     public abstract void request();
08. }
09.
10. // 真实主题类，即被代理类
11. class RealSubject implements AbstractSubject {
12.     public void request() {
13.         System.out.println("RealSubject's request() ...");
14.     }
15. }
16.
17. // 动态代理类，实现InvocationHandler接口
18. class DynamicProxy implements InvocationHandler {
19.
20.     // 被代理类的实例
21.     Object obj = null;
22.
23.     // 将被代理者的实例传进动态代理类的构造函数中
24.     public DynamicProxy(Object obj) {
25.         this.obj = obj;
26.     }
27.
28.     /**
29.      * 覆盖InvocationHandler接口中的invoke()方法
30.      *
31.      * 更重要的是，动态代理模式可以使得我们在不改变原来已有的代码结构
32.      * 的情况下，对原来的"真实方法"进行扩展、增强其功能，并且可以达到
33.      * 控制被代理对象的行为，下面的before、after就是我们可以进行特殊
34.      * 代码切入的扩展点了。
35.      */
36.     public Object invoke(Object proxy, Method method, Object[] args)
37.             throws Throwable {
38.         /*
39.          * before : doSomething();
40.          */
41.         Object result = method.invoke(this.obj, args);
42.
43.         /*
44.          * after : doSomething();
45.          */
46.         return result;
47.     }
48. }
49.
50. // 测试类
51. public class Client {
52.     public static void main(String[] args) {
53.
54.         // 被代理类的实例
55.         AbstractSubject realSubject = new RealSubject();
56.
57.         // 获得被代理类的类加载器，使得JVM能够加载并找到被代理类的内部结构，以及已实现的interface
58.         ClassLoader loader = realSubject.getClass().getClassLoader();
59.
60.         // 获得被代理类已实现的所有接口interface,使得动态代理类的实例
61.         Class<?>[] interfaces = realSubject.getClass().getInterfaces();
62.
63.         // 用被代理类的实例创建动态代理类的实例，用于真正调用处理程序
64.         InvocationHandler handler = new DynamicProxy(realSubject);
65.
66.         /*
67.          * loader : 被代理类的类加载器
68.          * interfaces : 被代理类已实现的所有接口，而这些是动态代理类要实现的接口列表
69.          * handler :  用被代理类的实例创建动态代理类的实例，用于真正调用处理程序
70.          *
```

```
71.          * return : 返回实现了被代理类所实现的所有接口的Object对象，即动态代理，需要强制转型
72.          */
73.         //获得代理的实例
74.         AbstractSubject proxy = (AbstractSubject) Proxy.newProxyInstance(
75.                 loader, interfaces, handler);
76.
77.         proxy.request();
78.         //打印出该代理实例的名称
79.         System.out.println(proxy.getClass().getName());
80.     }
81. }
```

测试结果：

```
RealSubject's  request()  ...
DesignPattern.proxy.dynamicProxy.$Proxy0
```

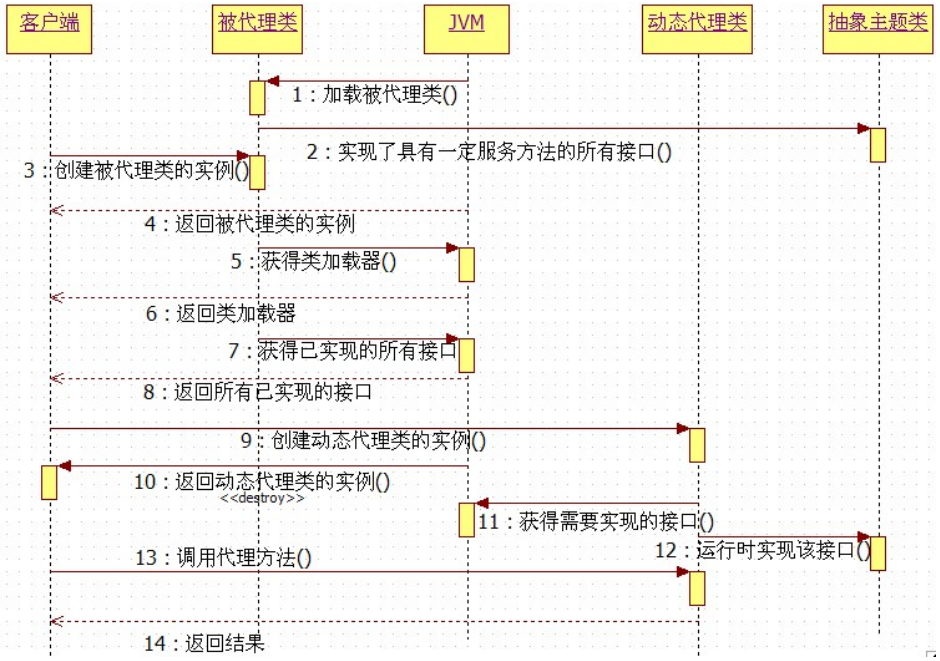
运行结果与前一篇文章中静态代理的一样，我们也发现这个动态代理的实例的名称为“$Proxy0”，前面的都是我所用的包名，记得以前学习内部类时，内部类编译之后生成的.class文件的默认命名方式是带有“$”。但是现在这个肯定不是内部类，因为“$”之前并没有任何一个外部类的名称。是不是以后遇到“$Proxy0”这样的名字就可以推断出该实例一定是个动态代理类的实例呢？有待证明。

其实，在上面整个代码当中，注释最少的就是DynamicProxy类中的invoke()方法了，因为看了Java帮助文档也不是特别明白，所以不敢乱写，这一点还有待探究。

话又说回来，动态代理机制确实很灵活，或者说很智能，但是这是运用到了Java中的反射机制，而反射机制又与JVM中栈区、堆区、方法区等底层细节以及类的加载、生命周期等知识相关，要完全理解相当不容易，看来还有很长的路要走呢！

最后，可能大家都觉得测试类即Client类中代码很繁杂，人家一看就不想要使用动态代理了，认为一使用就要有相当的消耗。那么此时我们可以再继续扩展一下，设计一个类能够根据传进的相关参数而返回最终客户需要的代理，这样的类设计是不是很像前面文章中工厂方法模式的应用呢？

花了快一个钟之后加上这个时序图，因为对此不甚了解，应该有许多错误，望各位指正，谢谢！



附件下载：
源代码

相关文章

Java反射经典实例 Java Reflection Cookbook
深入浅出话反射——明明白白我的心
白话C#反射
反射和属性
MSR路由器做BGP 反射

职位推荐

中级Java工程师
招聘Siebel开发兼职讲师
java开发工程师
Java
JAVA开发工程师

文章评论

[1楼]楼主 haolloyin 回复
2010-06-24 19:48:51
呵呵...又把上面的时序图中关于返回的线给改成正确的了，应该是虚线，而不是实线。

[2楼] meetcomet 回复
2013-11-15 10:35:50
写的太好了。谢谢。

[3楼] 工藤兰夏 回复
2015-03-07 12:37:19
写得很不错，谢谢楼主！我看的资料书上说的java动态代理机制搞得我很莫名其妙，原来只是生成代理的方法有所不一样，思想还是一样的。不过动态代理除了试用于任何方法前后都需要logging这种情况，就可以在invoke中一次性全部实现。如果是调用每种方法的前后权限设置都不一样，这种动态代理又有什么益处呢，或者说动态代理适用于什么背景下？不怎么理解动态的定义，是指的由jvm生成代理，而不是人为编写代理类吗？

[4楼] 1219957063 回复
2015-04-05 11:31:21
博主表达的清晰易懂，终于有点明白动态代理了，赞！

发表评论

昵 称: 登录 快速注册

验证码: 请点击后输入验证码 博客过2级，无需填写验证码

内 容: