代理模式   
  
代理模式的作用是：为其他对象提供一种代理以控制对这个对象的访问。在某些情况下，一个客户不想或者不能直接引用另一个对象，而代理对象可以在客户端和目标对象之间起到中介的作用。   
  
代理模式一般涉及到的角色有：   
  
抽象角色：声明真实对象和代理对象的共同接口；   
  
代理角色：代理对象角色内部含有对真实对象的引用，从而可以操作真实对象，同时代理对象提供与真实对象相同的接口以便在任何时刻都能代替真实对象。同时，代理对象可以在执行真实对象操作时，附加其他的操作，相当于对真实对象进行封装。   
  
真实角色：代理角色所代表的真实对象，是我们最终要引用的对象。(参见文献1)   
  
以下以《Java与模式》中的示例为例：   
  
抽象角色：

**Java代码  [收藏代码](javascript:void())**

1. **abstract** **public** **class** Subject
3. {
5. **abstract** **public** **void** request();
7. }

真实角色：实现了Subject的request()方法。 

**Java代码  [收藏代码](javascript:void())**

1. **public** **class** RealSubject **extends** Subject
3. {
5. **public** RealSubject()
7. {
9. }


13. **public** **void** request()
15. {
17. System.out.println("From real subject.");
19. }
21. }

代理角色： 

**Java代码  [收藏代码](javascript:void())**

1. **public** **class** ProxySubject **extends** Subject
3. {
5. **private** RealSubject realSubject;  //以真实角色作为代理角色的属性


9. **public** ProxySubject()
11. {
13. }


17. **public** **void** request()  //该方法封装了真实对象的request方法
19. {
21. preRequest();
23. **if**( realSubject == **null** )
25. {
27. realSubject = **new** RealSubject();
29. }
31. realSubject.request();  //此处执行真实对象的request方法
33. postRequest();
35. }


39. **private** **void** preRequest()
41. {
43. //something you want to do before requesting
45. }


49. **private** **void** postRequest()
51. {
53. //something you want to do after requesting
55. }
57. }

客户端调用： 

**Java代码  [收藏代码](javascript:void())**

1. Subject sub=**new** ProxySubject();
3. Sub.request();

       由以上代码可以看出，客户实际需要调用的是RealSubject类的request()方法，现在用ProxySubject来代理RealSubject类，同样达到目的，同时还封装了其他方法(preRequest(),postRequest())，可以处理一些其他问题。   
  
       另外，如果要按照上述的方法使用代理模式，那么真实角色必须是事先已经存在的，并将其作为代理对象的内部属性。但是实际使用时，一个真实角色必须对应一个代理角色，如果大量使用会导致类的急剧膨胀；此外，如果事先并不知道真实角色，该如何使用代理呢？这个问题可以通过Java的动态代理类来解决。   
  
  
  
2.动态代理类   
  
       Java动态代理类位于Java.lang.reflect包下，一般主要涉及到以下两个类：   
  
(1). Interface InvocationHandler：该接口中仅定义了一个方法Object：invoke(Object obj,Method method, J2EEjava语言JDK1.4APIjavalangObject.html">Object[] args)。在实际使用时，第一个参数obj一般是指代理类，method是被代理的方法，如上例中的request()，args为该方法的参数数组。这个抽象方法在代理类中动态实现。   
  
  
(2).Proxy：该类即为动态代理类，作用类似于上例中的ProxySubject，其中主要包含以下内容：   
  
Protected Proxy(InvocationHandler h)：构造函数，估计用于给内部的h赋值。   
  
Static Class getProxyClass (ClassLoader loader, Class[] interfaces)：获得一个代理类，其中loader是类装载器，interfaces是真实类所拥有的全部接口的数组。   
  
Static Object newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler h)：返回代理类的一个实例，返回后的代理类可以当作被代理类使用(可使用被代理类的在Subject接口中声明过的方法)。   
  
  
  
       所谓Dynamic Proxy是这样一种class：它是在运行时生成的class，在生成它时你必须提供一组interface给它，然后该class就宣称它实现了这些interface。你当然可以把该class的实例当作这些interface中的任何一个来用。当然啦，这个Dynamic Proxy其实就是一个Proxy，它不会替你作实质性的工作，在生成它的实例时你必须提供一个handler，由它接管实际的工作。(参见文献3)   
  
    在使用动态代理类时，我们必须实现InvocationHandler接口，以第一节中的示例为例：   
  
抽象角色(之前是抽象类，此处应改为接口)： 

**Java代码  [收藏代码](javascript:void())**

1. **public** **interface** Subject
3. {
5. **abstract** **public** **void** request();
7. }

具体角色RealSubject：同上；   
  
  
  
代理角色： 

**Java代码  [收藏代码](javascript:void())**

1. **import** java.lang.reflect.Method;
3. **import** java.lang.reflect.InvocationHandler;


7. **public** **class** DynamicSubject **implements** InvocationHandler {
9. **private** Object sub;


13. **public** DynamicSubject() {
15. }


19. **public** DynamicSubject(Object obj) {
21. sub = obj;
23. }




29. **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {
31. System.out.println("before calling " + method);


35. method.invoke(sub,args);


39. System.out.println("after calling " + method);
41. **return** **null**;
43. }


47. }

       该代理类的内部属性为Object类，实际使用时通过该类的构造函数DynamicSubject(Object obj)对其赋值；此外，在该类还实现了invoke方法，该方法中的   
  
method.invoke(sub,args);   
  
其实就是调用被代理对象的将要被执行的方法，方法参数sub是实际的被代理对象，args为执行被代理对象相应操作所需的参数。通过动态代理类，我们可以在调用之前或之后执行一些相关操作。   
  
客户端： 

**Java代码  [收藏代码](javascript:void())**

1. **import** java.lang.reflect.InvocationHandler;
3. **import** java.lang.reflect.Proxy;
5. **import** java.lang.reflect.Constructor;
7. **import** java.lang.reflect.Method;


11. **public** **class** Client
13. {


17. **static** **public** **void** main(String[] args) **throws** Throwable
19. {
21. RealSubject rs = **new** RealSubject();  //在这里指定被代理类
23. InvocationHandler ds = **new** DynamicSubject(rs);  //初始化代理类
25. Class cls = rs.getClass();
27. //以下是分解步骤
29. /\*
31. Class c = Proxy.getProxyClass(cls.getClassLoader(),cls.getInterfaces()) ;
33. Constructor ct=c.getConstructor(new Class[]{InvocationHandler.class});
35. Subject subject =(Subject) ct.newInstance(new Object[]{ds});
37. \*/
39. //以下是一次性生成
41. Subject subject = (Subject) Proxy.newProxyInstance(cls.getClassLoader(),
43. cls.getInterfaces(),ds );


47. subject.request();
49. }

       通过这种方式，被代理的对象(RealSubject)可以在运行时动态改变，需要控制的接口(Subject接口)可以在运行时改变，控制的方式(DynamicSubject类)也可以动态改变，从而实现了非常灵活的动态代理关系