# 简介

**[Java](http://lib.csdn.net/base/javase" \o "Java SE知识库" \t "http://blog.csdn.net/goskalrie/article/details/_blank)**编程的目标是实现现实不能完成的，优化现实能够完成的，是一种虚拟技术。生活中的方方面面都可以虚拟到代码中。代理模式所讲的就是现实生活中的这么一个概念：**中介**。

**代理模式的定义**：给某一个对象提供一个代理，并由 **代理对象控制对原对象的引用**。

代理模式包含如下角色：

**ISubject**：抽象主题角色，是一个接口。该接口是对象和它的代理共用的接口。

**RealSubject**：真实主题角色，是实现抽象主题接口的类。

**Proxy**：代理角色，内部含有对真实对象RealSubject的引用，从而可以操作真实对象。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。

**实现动态代理的关键技术是反射。**

# 静态代理

代理模式有几种，虚拟代理，计数代理，远程代理，动态代理。主要分为两类，静态代理和动态代理。静态代理比较简单，是由程序员编写的代理类，并在程序运行前就编译好的，而不是由程序动态产生代理类，这就是所谓的静态。

考虑这样的场景，管理员在网站上执行操作，在生成操作结果的同时需要记录操作日志，这是很常见的。此时就可以使用代理模式，代理模式可以通过聚合和继承两种方式实现：

**[java]** [view plain](http://blog.csdn.net/goskalrie/article/details/52458773" \o "view plain) [copy](http://blog.csdn.net/goskalrie/article/details/52458773" \o "copy) [print](http://blog.csdn.net/goskalrie/article/details/52458773" \o "print)[?](http://blog.csdn.net/goskalrie/article/details/52458773" \o "?)

1. /\*\*方式一：聚合式静态代理
2. \* @author Goser    (mailto:goskalrie@163.com)
3. \* @Since 2016年9月7日
4. \*/
5. //1.抽象主题接口
6. **public** **interface** Manager {
7. **void** doSomething();
8. }
9. //2.真实主题类
10. **public** **class** Admin **implements** Manager {
11. **public** **void** doSomething() {
12. System.out.println("Admin do something.");
13. }
14. }
15. //3.以聚合方式实现的代理主题,实现日志操作
16. **public** **class** AdminPoly **implements** Manager{
17. **private** Admin admin;
19. **public** AdminPoly(Admin admin) {
20. **super**();
21. **this**.admin = admin;
22. }
24. **public** **void** doSomething() {
25. System.out.println("Log:admin操作开始");
26. admin.doSomething();
27. System.out.println("Log:admin操作结束");
28. }
29. }
30. //4.测试代码
32. Manager m = **new** AdminPoly(**new** Admin());
33. m.doSomething();
34. //方式二：继承式静态代理
35. //与上面的方式仅代理类和测试代码不同
36. //1.代理类 ，实现日志操作
37. **public** **class** AdminProxy **extends** Admin {
38. @Override
39. **public** **void** doSomething() {
40. System.out.println("Log:admin操作开始");
41. **super**.doSomething();
42. System.out.println("Log:admin操作开始");
43. }
44. }
45. //2.测试代码
46. AdminProxy proxy = **new** AdminProxy();
47. proxy.doSomething();

聚合实现方式中代理类聚合了被代理类，且代理类及被代理类都实现了同一个接口，可实现灵活多变。继承式的实现方式则不够灵活。

比如，在管理员操作的同时需要进行权限的处理，操作内容的日志记录，操作后数据的变化三个功能。三个功能的排列组合有6种，也就是说使用继承要编写6个继承了Admin的代理类，而使用聚合，仅需要针对权限的处理、日志记录和数据变化三个功能编写代理类，在业务逻辑中根据具体需求改变代码顺序即可。

# ****动态代理****

一般来说，对代理模式而言，一个主题类与一个代理类一一对应，这也是静态代理模式的特点。

但是，也存在这样的情况，有n各主题类，但是代理类中的“前处理、后处理”都是一样的，仅调用主题不同。也就是说，多个主题类对应一个代理类，共享“前处理，后处理”功能，动态调用所需主题，大大减小了程序规模，这就是动态代理模式的特点。

## ****JDK动态代理****

### ****实现****

**[java]** [view plain](http://blog.csdn.net/goskalrie/article/details/52458773" \o "view plain) [copy](http://blog.csdn.net/goskalrie/article/details/52458773" \o "copy) [print](http://blog.csdn.net/goskalrie/article/details/52458773" \o "print)[?](http://blog.csdn.net/goskalrie/article/details/52458773" \o "?)

1. //1. 抽象主题
2. **public** **interface** Moveable {
3. **void** move()  **throws** Exception;
4. }
5. //2. 真实主题
6. **public** **class** Car **implements** Moveable {
7. **public** **void** move() **throws** Exception {
8. Thread.sleep(**new** Random().nextInt(1000));
9. System.out.println("汽车在马路行驶中…");
10. }
11. }
12. //3.**代理处理器类**
13. **public** **class** TimeHandler **implements** InvocationHandler {
14. **private** Object target;  //真实主题
16. **public** TimeHandler(Object target) {
17. **super**();
18. **this**.target = target;
19. }
20. /\*\*
21. \* 参数：
22. \*proxy 代理的对象
23. \*target 被代理的对象
24. \*method 被代理对象的方法
25. \*args 方法的参数
26. \* 返回：
27. \*Object 方法返回值
28. \*/
29. **public** Object invoke(Object proxy, Method method, Object[] args)**throws** Throwable {
30. **long** startTime = System.currentTimeMillis();
31. System.out.println("汽车开始行驶…");
32. method.invoke(target, args);
33. **long** stopTime = System.currentTimeMillis();
34. System.out.println("汽车结束行驶…汽车行驶时间：" + (stopTime - startTime) + "毫秒！");
35. **return** **null**;
36. }
38. }
39. //测试类
40. **public** **class** Test {
41. **public** **static** **void** main(String[] args) **throws** Exception{
42. Car car = **new** Car();
43. InvocationHandler h = **new** TimeHandler(car);
44. Class<?> cls = car.getClass();
45. /\*\*
46. \*loader 类加载器
47. \*interfaces 实现接口
48. \*h InvocationHandler
49. \*/
50. Moveable m = (Moveable)Proxy.newProxyInstance(cls.getClassLoader(),cls.getInterfaces(), h);
51. m.move();
52. }
53. }

代码讲解：

在**[测试](http://lib.csdn.net/base/softwaretest" \o "软件测试知识库" \t "http://blog.csdn.net/goskalrie/article/details/_blank)**代码中，Proxy.newProxyInstance()方法需要3个参数：类加载器（要进行代理的类）、被代理类实现的接口，事务处理器。所以先实例化Car，实例化InvocationHandler的子类TimeHandler，将各参数传入Proxy的静态方法newProxyInstance()即可获得Car的代理类，前面的静态代理，代理类是我们编写好的，而动态代理则不需要我们去编写代理类，是在程序中动态生成的。

### ****JDK动态代理步骤****

**1.** 创建一个实现InvocationHandler接口的类，它必须实现invoke()方法

**2.** 创建被代理的类及接口

**3.** 调用Proxy的静态方法，创建一个代理类

**4.** 通过代理调用方法

使用Java动态代理机制的好处：

1、减少编程的工作量：假如需要实现多种代理处理逻辑，只要写多个代理处理器就可以了，无需每种方式都写一个代理类。

2、系统扩展性和维护性增强，程序修改起来也方便多了(一般只要改代理处理器类就行了)。

使用Java动态代理机制的限制：

目前根据GOF的代理模式，代理类和委托类需要都实现同一个接口。也就是说只有实现了某个接口的类可以使用Java动态代理机制。但是，事实上使用中并不是遇到的所有类都会给你实现一个接口。因此，对于没有实现接口的类，目前无法使用该机制。有人说这不是废话吗，本来Proxy模式定义的就是委托类要实现接口的啊！但是没有实现接口的类，该如何实现动态代理呢？

当然不是没有办法，这也是我后面抽时间要继续整理和总结原先使用过的一件神器，相关Blog会不定期发上来。那就是大名鼎鼎的CGLib...

PS：CGLib中的动态代理已经新鲜出炉，欢迎访问：<http://shensy.iteye.com/blog/1873155>

**一个动态代理的例子**

**interface IHello{**

**/\*\***

**\* 业务方法**

**\***

**\* @param str**

**\*/**

**void sayHello(String str);**

**}**

**/\***

**\* 目标类代码：**

**\*/**

**class Hello implements IHello{**

**@Override**

**public void sayHello(String str) {**

**System.out.println("hello " + str);**

**}**

**}**

**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");**

**}**

**}**

**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();**

**System.out.println("\nproxy:"+proxy.getClass().getSimpleName());**

**System.out.println("target:"+target.getClass().getSimpleName());**

**System.out.println("method: "+method.getName());**

**System.out.println("args[0]: "+args[0].getClass().getName());**

**System.out.println("result: "+result+"\n");**

**return result;**

**}**

**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");**

**}**

**}**

**输出结果：**

**Fri Jan 06 20:45:50 CST 2017 say hello start...**

**hello 明天**

**Fri Jan 06 20:45:50 CST 2017 say hello end**

**proxy: $Proxy0**

**target: Hello**

**method: sayHello**

**args[0]: java.lang.String**

**result: null**

**JDK中所要进行动态代理的类必须要实现一个接口，也就是说只能对该类所实现接口中定义的方法进行代理，这在实际编程中具有一定的局限性，而且使用反射的效率也并不是很高。**

**二、使用CGLib实现：**

**使用CGLib实现动态代理，完全不受代理类必须实现接口的限制，而且CGLib底层采用ASM字节码生成框架，使用字节码技术生成代理类，比使用Java反射效率要高。唯一需要注意的是，CGLib不能对声明为final的方法进行代理，因为CGLib原理是动态生成被代理类的子类。**

**下面，将通过一个实例介绍使用CGLib实现动态代理。**

这是一个需要被代理的类，也就是父类，通过字节码技术创建这个类的子类，实现动态代理。

1. **public** **class** SayHello {
2. **public** **void** say(){
3. System.out.println("hello everyone");
4. }
5. }

该类实现了创建子类的方法与代理的方法。 **getProxy(SuperClass.class)方法通过入参即父类的字节码，通过扩展父类的class来创建代理对象。**intercept()方法拦截所有目标类方法的调用，obj表示目标类的实例，method为目标类方法的反射对象，args为方法的动态入参，proxy为代理类实例。proxy.invokeSuper(obj, args)通过代理类调用父类中的方法。

1. **public** **class** CglibProxy **implements** MethodInterceptor{
2. **private** Enhancer enhancer = **new** Enhancer();
3. //返回子类实例
4. **public** Object getProxy(Class clazz){
5. //设置需要创建子类的类
6. enhancer.setSuperclass(clazz);
7. enhancer.setCallback(**this**);
8. //通过字节码技术动态创建子类实例
9. **return** enhancer.create();
10. }
11. //实现MethodInterceptor接口方法
12. **public** Object intercept(Object obj, Method method, Object[] args,  MethodProxy proxy)
13. **throws** Throwable {
15. System.out.println("前置代理");
16. //通过代理类调用父类中的方法
17. Object result = proxy.invokeSuper(obj, args);
18. System.out.println("后置代理");
19. **return** result;
20. }
21. }

具体实现类：

1. **public** **class** DoCGLib {
2. **public** **static** **void** main(String[] args) {
3. CglibProxy proxy = **new** CglibProxy();
4. //通过生成子类的方式创建代理类，返回子类实例， SayHello为父类
5. SayHello proxyImp = (SayHello)proxy.getProxy(SayHello.**class**);
6. proxyImp.say();
7. }
8. }
9. getProxy方法：
10. //返回子类实例
11. **public** Object getProxy(Class clazz){
12. //设置需要创建子类的类
13. enhancer.setSuperclass(clazz);
14. enhancer.setCallback(**this**);
15. //通过字节码技术动态创建子类实例
16. **return** enhancer.create();
17. }

输出结果：

**[plain]** [view plain](http://blog.csdn.net/yakoo5/article/details/9099133/" \o "view plain) [copy](http://blog.csdn.net/yakoo5/article/details/9099133/" \o "copy)

1. 前置代理
2. hello everyone
3. 后置代理