**Java的三种代理模式**

**1.代理模式**

代理(Proxy)是一种设计模式,提供了对目标对象另外的访问方式;即通过代理对象访问目标对象.这样做的好处是:可以在目标对象实现的基础上,增强额外的功能操作,即扩展目标对象的功能.  
这里使用到编程中的一个思想:不要随意去修改别人已经写好的代码或者方法,如果需改修改,可以通过代理的方式来扩展该方法

代理模式的关键点是:代理对象与目标对象.代理对象是对目标对象的扩展,并会调用目标对象



### 1.1.静态代理

静态代理在使用时,需要定义接口或者父类,被代理对象与代理对象一起实现相同的接口或者是继承相同父类.

下面举个案例来解释:  
模拟保存动作,定义一个保存动作的接口:IUserDao.java,然后目标对象实现这个接口的方法UserDao.java,此时如果使用静态代理方式,就需要在代理对象(UserDaoProxy.java)中也实现IUserDao接口.调用的时候通过调用代理对象的方法来调用目标对象.  
需要注意的是,代理对象与目标对象要实现相同的接口,然后通过调用相同的方法来调用目标对象的方法

代码示例:  
接口:IUserDao.java

/\* 接口 \*/

public interface IUserDao {

void save();

}

目标对象:UserDao.java

/\* 接口实现 \* 目标对象 \*/

public class UserDao implements IUserDao {

public void save() {

System.out.println("----已经保存数据!----");

}

}

代理对象:UserDaoProxy.java

/\*代理对象,静态代理 \*/

public class UserDaoProxy implements IUserDao{

//接收保存目标对象

private IUserDao target;

public UserDaoProxy(IUserDao target){

this.target=target;

}

public void save() {

System.out.println("开始事务...");

target.save();//执行目标对象的方法

System.out.println("提交事务...");

}

}

测试类:App.java

/\* 测试类 \*/

public class App {

public static void main(String[] args) {

//目标对象

UserDao target = new UserDao();

//代理对象,把目标对象传给代理对象,建立代理关系

UserDaoProxy proxy = new UserDaoProxy(target);

proxy.save();//执行的是代理的方法

}

}

**静态代理总结:**  
1.可以做到在不修改目标对象的功能前提下,对目标功能扩展.  
2.缺点:

* 因为代理对象需要与目标对象实现一样的接口,所以会有很多代理类,类太多.同时,一旦接口增加方法,目标对象与代理对象都要维护.

如何解决静态代理中的缺点呢?答案是可以使用动态代理方式

### 1.2.动态代理

**动态代理有以下特点:**  
1.代理对象,不需要实现接口  
2.代理对象的生成,是利用JDK的API,动态的在内存中构建代理对象(需要我们指定创建代理对象/目标对象实现的接口的类型)  
3.动态代理也叫做:JDK代理,接口代理

4. 代理对象不需要实现接口,但是目标对象一定要实现接口,否则不能用动态代理

**JDK中生成代理对象的API**  
代理类所在包:java.lang.reflect.Proxy  
JDK实现代理只需要使用newProxyInstance方法,但是该方法需要接收三个参数,完整的写法是:

static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces,InvocationHandler h )

注意该方法是在Proxy类中是静态方法,且接收的三个参数依次为:

* ClassLoader loader,:指定当前目标对象使用类加载器,获取加载器的方法是固定的
* Class<?>[] interfaces,:目标对象实现的接口的类型,使用泛型方式确认类型
* InvocationHandler h:事件处理,执行目标对象的方法时,会触发事件处理器的方法,会把当前执行目标对象的方法作为参数传入

代码示例:  
接口类IUserDao.java以及接口实现类,目标对象UserDao是一样的,没有做修改.在这个基础上,增加一个代理工厂类(ProxyFactory.java),将代理类写在这个地方,然后在测试类(需要使用到代理的代码)中先建立目标对象和代理对象的联系,然后代用代理对象的中同名方法

代理工厂类:ProxyFactory.java

#### 首先我们要声明一个动态代理类，实现InvocationHandler接口

/\*\*

\* 创建动态代理对象

\* 动态代理不需要实现接口,但是需要继承InvocationHandler

\*/

**public** **class** ProxyFactory **implements** InvocationHandler{

//维护一个目标对象 被代理类的实例

**private** Object target;

//将被代理者的实例传进动态代理类的构造函数中

**public** ProxyFactory(Object target){

**this**.target=target;

}

/\*\*

\* 覆盖InvocationHandler接口中的invoke()方法

\* 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，下面的before、after就是我们可以进行特殊

\* 代码切入的扩展点了。

\*/

@Override

**public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {

System.***out***.println("开始事务2");

//执行目标对象方法

Object returnValue = method.invoke(target, args);

System.***out***.println("提交事务2");

**return** returnValue;

}

}

测试类:App.java

/\* 测试类 \*/

//目标对象

//我们要代理的真实对象

IUserDao target = **new** UserDao();

//我们要代理哪个真实对象，就将该对象传进去，最后是通过该真实对象来调用其方法的

InvocationHandler handler = **new** ProxyFactory(target);

/\*

\* 通过Proxy的newProxyInstance方法来创建我们的代理对象，我们来看看其三个参数

\* 第一个参数 handler.getClass().getClassLoader() ，我们这里使用handler这个类的ClassLoader对 象来加载我们的代理对象

\* 第二个参数customer.getClass().getInterfaces()，我们这里为代理对象提供的接口是真实对象所实行的接 口，表示我要代理的是该真实对象，这样我就能调用这组接口中的方法了

\* 第三个参数handler， 我们这里将这个代理对象关联到了上方的 InvocationHandler 这个对象上

\*/

IUserDao proxy = (IUserDao) Proxy.*newProxyInstance*(handler.getClass().getClassLoader(), target.getClass().getInterfaces(), handler);

// 执行方法 【代理对象】

proxy.save();

**总结:**  
代理对象不需要实现接口,但是目标对象一定要实现接口,否则不能用动态代理

### 1.3.Cglib代理

上面的静态代理和动态代理模式都是要求目标对象是实现一个接口的目标对象,但是有时候目标对象只是一个单独的对象,并没有实现任何的接口,这个时候就可以使用以目标对象子类的方式类实现代理,这种方法就叫做:Cglib代理

Cglib代理,也叫作子类代理,它是在内存中构建一个子类对象从而实现对目标对象功能的扩展.

* JDK的动态代理有一个限制,就是使用动态代理的对象必须实现一个或多个接口,如果想代理没有实现接口的类,就可以使用Cglib实现.
* Cglib是一个强大的高性能的代码生成包,它可以在运行期扩展java类与实现java接口.它广泛的被许多AOP的框架使用,例如Spring AOP和synaop,为他们提供方法的interception(拦截)
* Cglib包的底层是通过使用一个小而块的字节码处理框架ASM来转换字节码并生成新的类.不鼓励直接使用ASM,因为它要求你必须对JVM内部结构包括class文件的格式和指令集都很熟悉.

Cglib子类代理实现方法:  
1.需要引入cglib的jar文件,但是Spring的核心包中已经包括了Cglib功能,所以直接引入spring-core-3.2.5.jar即可.  
2.引入功能包后,就可以在内存中动态构建子类  
3.代理的类不能为final,否则报错  
4.目标对象的方法如果为final/static,那么就不会被拦截,即不会执行目标对象额外的业务方法.

代码示例:  
Cglib代理工厂:ProxyFactory.java

/\*\*

\* Cglib子类代理工厂

\* 对UserDao在内存中动态构建一个子类对象

\*/

**public** **class** ProxyFactory **implements** MethodInterceptor{

//维护目标对象

**private** Object target;

**public** ProxyFactory(Object target) {

**this**.target = target;

}

//给目标对象创建一个代理对象

**public** Object getProxyInstance(){

//1.工具类

Enhancer en = **new** Enhancer();

//2.设置父类

en.setSuperclass(target.getClass());

//3.设置回调函数

en.setCallback(**this**);

//4.创建子类(代理对象)

**return** en.create();

}

@Override

**public** Object intercept(Object obj, Method method, Object[] args, MethodProxy proxy)

**throws** Throwable {

System.***out***.println("开始事务...");

//执行目标对象的方法

Object returnValue = method.invoke(target, args);

System.***out***.println("提交事务...");

**return** returnValue;

}

}

测试类:App.java

/\* 测试类 \*/

//目标对象

UserDao target = **new** UserDao();

//代理对象

UserDao proxy = (UserDao)**new** ProxyFactory(target).getProxyInstance();

//执行代理对象的方法

proxy.save();

在Spring的AOP编程中:  
如果加入容器的目标对象有实现接口,用JDK代理  
如果目标对象没有实现接口,用Cglib代理