代理是一种模式，提供了对目标对象的间接访问方式，即通过代理访问目标对象。如此便于在目标实现的基础上增加额外的功能操作，前拦截，后拦截等，以满足自身的业务需求，同时代理模式便于扩展目标对象功能的特点也为多人所用。

（1）静态代理

静态代理的实现模式：首先创建一个接口（JDK代理都是面向接口的），然后创建具体实现类来实现这个接口，在创建一个代理类同样实现这个接口，不同之处在于，具体实现类的方法中需要将接口中定义的方法的业务逻辑功能实现，而代理类中的方法只要调用具体类中的对应方法即可，这样我们在需要使用接口中的某个方法的功能时直接调用代理类的方法即可，将具体的实现类隐藏在底层。

|  |
| --- |
| 第一步：定义总接口Iuser.java  public interface Iuser {  public void eat(String s);  }  第二步：创建具体实现类UserImpl.java  public class UserImpl implements Iuser {  　　@Override  　　public void eat(String s) {  　　　　System.out.println("我要吃"+s);  　　}  }  第三步：创建代理类UserProxy.java  public class UserProxy implements Iuser {  　　private Iuser user = new UserImpl();  　　@Override  　　public void eat(String s) {  　　　　System.out.println("静态代理前置内容");  　　　　user.eat(s);  　　　　System.out.println("静态代理后置内容");  　　}  }  第四步：创建测试类ProxyTest.java;  public class ProxyTest {  　　public static void main(String[] args) {  　　　　UserProxy proxy = new UserProxy();  　　　　proxy.eat("苹果");  　　}  }  运行结果：  静态代理前置内容  我要吃苹果  静态代理后置内容 |

上面的程序中，先创建了一个接口，然后定义一个类实现这个接口，同时定好以一个代理类同时实现这个接口，但是可以看出具体实现类主要是完成方法的定义和实现，而代理类就直接调用类中的方法，之后定义一个测试类，编写主函数定义代理类对象完成代理类中的方法

**静态代理的总结**

　　优点：可以做到不对目标对象进行修改的前提下，对目标对象进行功能的扩展和拦截。

缺点：因为代理对象，需要实现与目标对象一样的接口，会导致代理类十分繁多，不易维护，同时一旦接口增加方法，则目标对象和代理类都需要维护。

其工作原理就类似于现实生活中一些技术公司和代理厂家，他们都有一个共同的目标，就是商品，但技术公司主要是完成技术研发，即研制生产方法、用何种技术手段去生产，代理厂家主要是完成生产，在这里，技术公司就相当于具体实现类，代理厂家就相当于代理类。

（2）动态代理

动态代理的思维模式与之前的一般模式是一样的，也是面向接口进行编码，创建代理类将具体类隐藏解耦，不同之处在于代理类的创建时机不同，动态代理需要在运行时因需实时创建。

动态代理的实现步骤：

　　第一步：创建接口，JDK动态代理基于接口实现，所以接口必不可少（准备工作）

　　第二步：实现InvocationHandler接口，重写invoke方法（准备工作）

　　第三步：调用Proxy的静态方法newProxyInstance方法生成代理实例（生成实例时需要提供类加载器，我们可以使用接口类的加载器即可）

　　第四步：使用新生成的代理实例调用某个方法实现功能

其实现过程前两步相同，后两步有差异

|  |
| --- |
| 第三步：创建实现InvocationHandler接口的代理类  import java.lang.reflect.InvocationHandler;  import java.lang.reflect.Method;  public class DynamicProxy implements InvocationHandler {  　　private Object object;//用于接收具体实现类的实例对象  　　//使用带参数的构造器来传递具体实现类的对象  　　public DynamicProxy(Object obj){  　　　　this.object = obj;  　　}  　　@Override  　　public Object invoke(Object proxy, Method method, Object[] args)throws Throwable {  　　　　System.out.println("前置内容");  　　　　method.invoke(object, args);  　　　　System.out.println("后置内容");  　　　　return null;  　　}  }  第四步：创建测试类ProxyTest.java  import java.lang.reflect.InvocationHandler;  import java.lang.reflect.Proxy;  public class ProxyTest {  　　public static void main(String[] args) {  　　　　Iuser user = new UserImpl();  　　　　InvocationHandler h = new DynamicProxy(user);  　　　　Iuser proxy = (Iuser) Proxy.newProxyInstance(Iuser.class.getClassLoader(), new Class[]{Iuser.class}, h);  　　　　proxy.eat("苹果");  　　}  }  运行结果：  前置内容  我要吃苹果  后置内容 |

静态代理与动态代理不同之处体现在静态代理我们知道要对哪个接口、哪个实现类来创建代理类，所以我们在编译前就直接实现与实现类相同的接口，直接在实现的方法中调用实现类中的相应（同名）方法即可；而动态代理不同，我们不知道它什么时候创建，也不知道要创建针对哪个接口、实现类的代理类（因为它是在运行时因需实时创建的）。

虽然二者创建时机不同，创建方式也不相同，但是原理是相同的，不同之处仅仅是：静态代理可以直接编码创建，而动态代理是利用反射机制来抽象出代理类的创建过程。

第一点：静态代理需要实现与实现类相同的接口，而动态代理需要实现的是固定的Java提供的内置接口（一种专门提供来创建动态代理的接口）InvocationHandler接口，因为java在接口中提供了一个可以被自动调用的方法invoke。  
第二点：private Object object;  
　　　 public UserProxy(Object obj){this.object = obj;}

这几行代码与静态代理之中在代理类中定义的接口指向具体实现类的实例的代码异曲同工，通过这个构造器可以创建代理类的实例，创建的同时还能将具体实现类的实例与之绑定（object指的就是实现类的实例，这个实例需要在测试类中创建并作为参数来创建代理类的实例），实现了静态代理类中private Iuser user = new UserImpl();一行代码的作用相近，这里为什么不是相同，而是相近呢，主要就是因为静态代理的那句代码中包含的实现类的实例的创建，而动态代理中实现类的创建需要在测试类中完成，所以此处是相近。  
　 第三点：invoke(Object proxy, Method method, Object[] args)方法，该方法是InvocationHandler接口中定义的唯一方法，该方法在调用指定的具体方法时会自动调用。其参数为：代理实例、调用的方法、方法的参数列表  
　　在这个方法中我们定义了几乎和静态代理相同的内容，仅仅是在方法的调用上不同，不同的原因与之前分析的一样（创建时机的不同，创建的方式的不同，即反射），Method类是反射机制中一个重要的类，用于封装方法，该类中有一个方法那就是invoke(Object object,Object...args)方法，其参数分别表示：所调用方法所属的类的对象和方法的参数列表，这里的参数列表正是从测试类中传递到代理类中的invoke方法三个参数中最后一个参数（调用方法的参数列表）中，在传递到method的invoke方法中的第二个参数中的（此处有点啰嗦）。  
　　第四点：测试类中的异同  
　　静态代理中我们测试类中直接创建代理类的对象，使用代理类的对象来调用其方法即可，若是别的接口（这里指的是别的调用方）要调用Iuser的方法，也可以使用此法  
动态代理中要复杂的多，首先我们要将之前提到的实现类的实例创建（补充完整），然后利用这个实例作为参数，调用代理来的带参构造器来创建“代理类实例对象”，这里加引号的原因是因为它并不是真正的代理类的实例对象，而是创建真正代理类实例的一个参数，这个实现了InvocationHandler接口的类严格意义上来说并不是代理类，我们可以将其看作是创建代理类的必备中间环节，这是一个调用处理器，也就是处理方法调用的一个类，不是真正意义上的代理类，可以这么说：创建一个方法调用处理器实例。  
　　下面才是真正的代理类实例的创建，之前创建的”代理类实例对象“仅仅是一个参数

Iuser proxy = (Iuser) Proxy.newProxyInstance(Iuser.class.getClassLoader(), new Class[]{Iuser.class}, h);

　　这里使用了动态代理所依赖的第二个重要类Proxy，此处使用了其静态方法来创建一个代理实例，其参数分别是：类加载器（可为父类的类加载器）、接口数组、方法调用处理器实例  
　　这里同样使用了多态，使用接口指向代理类的实例，最后会用该实例来进行具体方法的调用即可。

关于InvocationHandler

InvocationHandler是JDK中提供的专门用于实现基于接口的动态代理的接口，主要用于进行方法调用模块，而代理类和实例的生成需要借助Proxy类完成。

　　每个代理类的实例的调用处理器都是实现该接口实现的，而且是必备的，即每个动态代理实例的实现都必须拥有实现该接口的调用处理器，也可以这么说，每个动态代理实例都对应一个调用处理器。

　　这里要区分两个概念，代理类和代理实例，调用处理器是在创建代理实例的时候才与其关联起来的，所以它与代理实例是一一对应的，而不是代理类。

动态代理的总结

　　 优点：代理对象无需实现接口，免去了编写很多代理类的烦恼，同时接口增加方法也无需再维护目标对象和代理对象，只需在事件处理器中添加对方法的判断即可。

　　 缺点：代理对象不需要实现接口，但是目标对象一定要实现接口，否则无法使用JDK动态代理。