常用的代理方式可以粗分为：静态代理和动态代理

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。

静态代理应用：

假设现在我们有一个类Calculator，代表一个计算器，它可以进行加减乘除操作

public class Calculator {

//加

public int add(int a, int b) {

int result = a + b;

return result;

}

//减

public int subtract(int a, int b) {

int result = a - b;

return result;

}

//乘法、除法...

}

现有一个需求：在每个方法执行前后打印日志。你有什么好的方案？

完成此需求，代码需满足：

1. 直接修改源程序，不符合开闭原则。应该对扩展开放，对修改关闭
2. 如果Calculator有几十个、上百个方法，修改量太大
3. 存在重复代码（都是在核心代码前后打印日志）
4. 日志打印硬编码在代理类中，不利于后期维护：比如你花了一上午终于写完了，组长告诉你这个功能取消，于是你又要打开Calculator花十分钟删除日志打印的代码！

按上面的描述，代理类和目标类需要实现同一个接口，所以我打算这样做：

* 将Calculator抽取为接口
* 创建目标类CalculatorImpl实现Calculator
* 创建代理类CalculatorProxy实现Calculator

代理类中的代码如下：

public class StaticProxy\_Proxy implements StaticProxy {  
  
 private StaticProxy tStaticProxy;  
  
 public StaticProxy\_Proxy(StaticProxy tStaticProxy){  
 this.tStaticProxy = tStaticProxy;  
 }  
  
 @Override  
 public int add(int a, int b) {  
 System.***out***.println("add方法开始");  
 int result = tStaticProxy.add(a, b);  
 System.***out***.println("add方法结束");  
 return result;  
 }  
}

静态代理的优点：可以在不修改目标对象的前提下，对目标对象进行功能的扩展和拦截。但是它也仅仅解决了上一种方案4大缺点中的第1点：

1. 直接修改源程序，不符合开闭原则。应该对扩展开放，对修改关闭 √
2. 如果Calculator有几十个、上百个方法，修改量太大 ×
3. 存在重复代码（都是在核心代码前后打印日志） ×
4. 日志打印硬编码在代理类中，不利于后期维护：比如你花了一上午终于写完了，组长告诉你这个功能取消，于是你又要打开Calculator花十分钟删除全部新增代码！×

总的来说，静态代理就是：

代理对象 = 增强代码 + 目标对象（原对象）

静态代理的缺陷：

程序员要手动为每一个目标类编写对应的代理类。如果当前系统已经有成百上千个类，工作量太大了。所以，现在我们的努力方向是：如何少写或者不写代理类，却能完成代理功能？

动态代理：

JDK动态代理、 CGLIB动态代理

本次主要讲解JDK的动态代理：

public class DynamicProxyTest {  
  
 public static void main(String[] args) throws Exception{  
 DynamicProxyImpl dynamicProxy = new DynamicProxyImpl();  
 DynamicProxy proxy = (DynamicProxy)getProxy(dynamicProxy);  
 int result = proxy.add(1, 2);  
 System.***out***.println("代理1后的结果：" + result);  
 DynamicProxy2 proxy2 = (DynamicProxy2) getProxy(dynamicProxy);  
 result = proxy2.subtract(1, 2);  
 System.***out***.println("代理2后的结果：" + result);  
 }  
  
 public static Object getProxy(final Object target) throws Exception{  
 Class<?>[] interfaces = target.getClass().getInterfaces();  
 Class proxyClass =  
 Proxy.getProxyClass(target.getClass().getClassLoader(), target.getClass().getInterfaces());  
  
 Constructor constructor = proxyClass.getConstructor(InvocationHandler.class);  
  
 Object proxy = constructor.newInstance(new InvocationHandler() {  
 @Override  
 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
 System.***out***.println(method.getName() + "开始执行");  
 Object result = method.invoke(target, args);  
 System.***out***.println(result);  
 System.***out***.println(method.getName() + "执行结束");  
 return result;  
 }  
 });  
 return proxy;  
 }  
  
 public static Object getProxy02(Object target){  
 Object proxy = Proxy.newProxyInstance(  
 target.getClass().getClassLoader(),  
 target.getClass().getInterfaces(),  
 new InvocationHandler(){  
 @Override  
 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
 System.***out***.println(method.getName() + "方法开始执行...");  
 Object result = method.invoke(target, args);  
 System.***out***.println(result);  
 System.***out***.println(method.getName() + "方法执行结束...");  
 return result;  
 }  
 }  
 );  
 return proxy;  
 }  
}

JDK中的动态代理是通过反射类Proxy以InvocationHandler回调接口实现的

有两种方式可以实现该功能：

方式一： Proxy.getProxyClass();

public static Class<?> getProxyClass(ClassLoader loader,  
 Class<?>... interfaces)

1、获取代理对象的Class对象

通过Proxy.getProxyClass(类加载器，目标类实现的接口)获取代理对象的Class文件。

Class<?>[] interfaces = target.getClass().getInterfaces();

// interfaces: target目标对象所实现的所有的接口类  
Class proxyClass =  
 Proxy.getProxyClass(target.getClass().getClassLoader(), target.getClass().getInterfaces());

//proxyClass: 获取代理对象的class

2、获取代理对象的构造函数对象

代理对象.getConstructor(InvocationHandler.class)

获取代理对象相应的构造函数，返回值是Constructor对象

Constructor constructor = proxyClass.getConstructor(InvocationHandler.class);

3、通过构造函数对象创建对象实例

Object proxy = constructor.newInstance(new InvocationHandler() {  
 @Override  
 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

//invoike: 方法只有代理对象在实际调用方法的时候才会进来  
 System.***out***.println(method.getName() + "开始执行");  
 Object result = method.invoke(target, args);  
 System.***out***.println(result);  
 System.***out***.println(method.getName() + "执行结束");  
 return result;  
 }  
});

在invoke(object proxy, Method method, Object[] args);

中，通过method.invoke(目标对象， 参数)，实际是：int result = proxy.add(1, 2);在调用时，通过反射，确定了此处的method方法为add，并再次通过反射进行调用，传入真实对象以及相应的参数。

代理对象在执行add方法时，实际是调用了：

constructor.newInstance(new InvocationHandler()

中的invoke方法，底层具体的业务方法还是target中的方法，只是代理对象里面增加了相应的前置或者后置处理，目标类中的处理方法代理对象是不涉及改动的。

方式二： public static Object newProxyInstance(ClassLoader loader,  
 Class<?>[] interfaces,  
 InvocationHandler h)

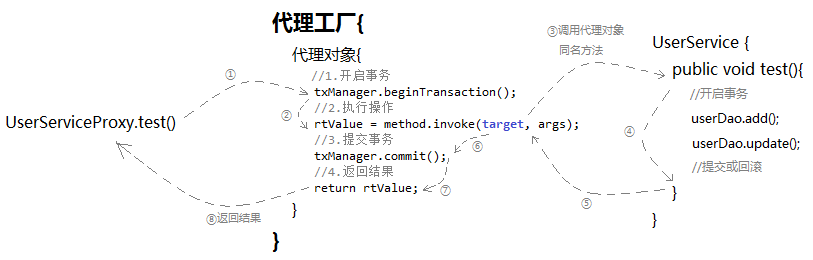
JDK动态代理的局限性：

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。

-------------------------------------------------

之所以实现相同接口，是为了尽可能保证代理对象的内部结构和目标对象一致，这样我们对代理对象的操作最终都可以转移到目标对象身上，代理对象只需专注于增强代码的编写。

Spring中AOP的使用：



@Autowired  
private TmPolicySupervisionService tmPolicySupervisionService;

自动注入的可能是TmPolicySupervisionService的实现类，也有可能是代理对象。

也就是说，代理对象方法 = 事务 + 目标对象方法。