1.引言

动态代理在 Java 中有着广泛的应用，比如 AOP 的实现原理、RPC远程调用、Java 注解对象获取、日志框架、全局性异常处理、事务处理等。

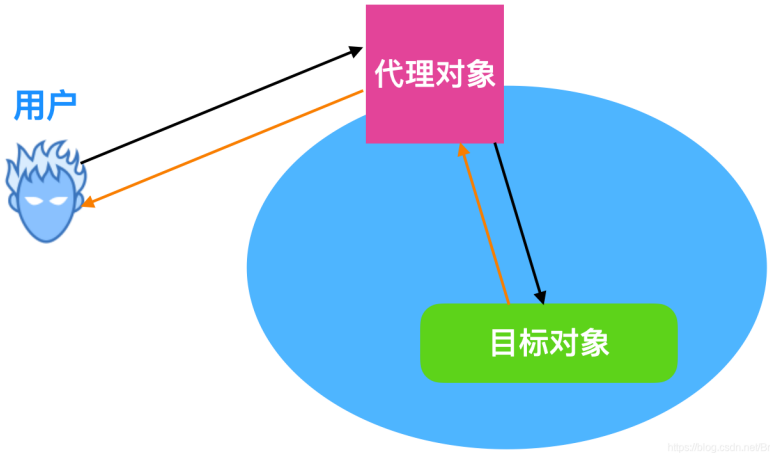
> *Spring框架的声明式事务管理，本质就是代理设计模式的体现*

在了解动态代理前，我们需要先了解一下什么是代理模式。

2.代理模式

`代理模式(Proxy Pattern)`是 23 种设计模式的一种，属于结构型模式。他指的是一个对象本身不做实际的操作，而是通过其他对象来得到自己想要的结果。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**。

> *这里能体现出一个非常重要的编程思想：不要随意去改源码，如果需要修改，可以通过代理的方式来扩展该方法。*

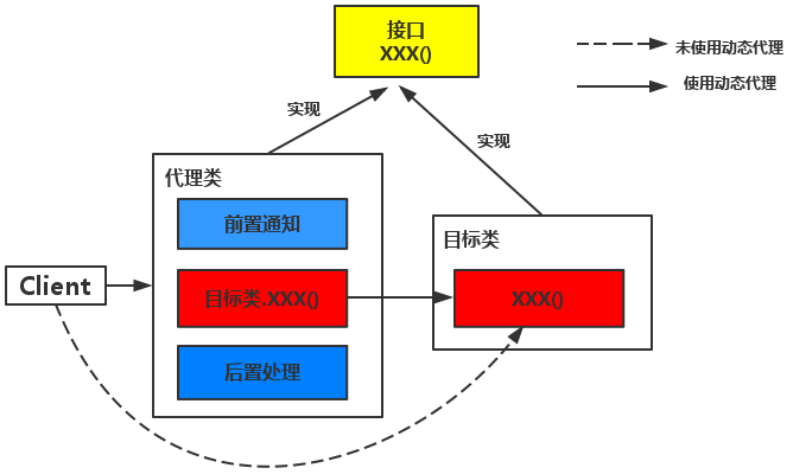


如上图所示，用户不能直接使用目标对象，而是构造出一个代理对象，由代理对象作为中转，代理对象负责调用目标对象真正的行为，从而把结果返回给用户。

也就是说，**代理的关键点就是代理对象和目标对象的关系**。

代理模式主要由三个元素共同构成：

1）一个接口，接口中的方法是要真正去实现的。
2）被代理类，实现上述接口，这是真正去执行接口中方法的类。
3）代理类，同样实现上述接口，同时封装被代理类对象，帮助被代理类去实现方法（动态代理和静态代理的区别仅仅是代理类是手写的代码还是JDK创建的）



使用代理模式必须要让代理类和目标类实现相同的接口，客户端通过代理类来调用目标方法，代理类会将所有的方法调用分派到目标对象上反射执行，还可以在分派过程中添加"前置通知"和后置处理!

（如在调用目标方法前**校验权限**，在调用完目标方法后打印日志等）等功能。

3.静态代理

□ **第一步**：创建 UserService 接口

```
public interface UserService {

    // 添加 user
    public void addUser(User user);

    // 删除 user
    public void deleteUser(int uid);
}
```

□ 第二步:创建 UserServiceImpl类（要被代理的类，目标类）

```
public class UserServiceImpl implements UserService {

    public void addUser(User user) {
        System.out.println("增加 User");
    }

    public void deleteUser(int uid) {
        System.out.println("删除 User");
    }
}
```

□ 第三步:创建事务类

```
public class MyTransaction {

    // 开启事务
    public void before() {
        System.out.println("开启事务");
    }

    // 提交事务
    public void after() {
        System.out.println("提交事务");
    }
}
```

□ ~~第四步：使用原始的方式给UserServiceImpl类增加事务功能~~

~~这个虚拟的第四步是没有使用代理之前给UserServiceImpl类增加事务的功能~~

```
/**
public class UserServiceImpl implements UserService {
    MyTransaction tx = new MyTransaction();
    public void addUser(User user) {
        tx.before();
        System.out.println("增加 User");
        tx.after();
    }
```

```
    public void deleteUser(int uid) {
        tx.before();
        System.out.println("删除 User");
        tx.after();
    }
}
*/
```

~~这种方式使代码的耦合度太高了，所以在不使用这种方式给UserServiceImpl类增加事务；~~

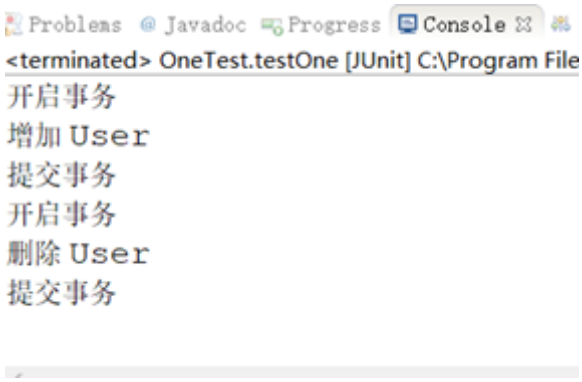
□ 第四步：创建代理类 ProxyUser.java给UserServiceImpl类增加事务功能

```
public class ProxyUser implements UserService {

    // 真实类,保存UserServiceImpl类的实例对象
    private UserService userService;
    // 事务类
    private MyTransaction transaction;

    // 使用构造函数实例化
    public ProxyUser(UserService userService,
MyTransaction transaction) {
        this.userService = userService;
        this.transaction = transaction;
    }

    public void addUser(User user) {
        transaction.before();
        userService.addUser(user);
        transaction.after();
    }

    public void deleteUser(int uid) {
        transaction.before();
        userService.deleteUser(uid);
        transaction.after();
    }
}
```

□ 测试：

```
public class TestUser {

    @Test
    public void testOne() {
        MyTransaction transaction = new MyTransaction();
        UserService userService = new UserServiceImpl();
        // 产生静态代理对象
        ProxyUser proxy = new ProxyUser(userService,
transaction);
        proxy.addUser(null);
        proxy.deleteUser(0);
    }
}
```

运行结果：



这是一个很基础的静态代理，业务类UserServiceImpl 只需要关注业务逻辑本身，保证了业务的重用性，这也是代理类的优点，没什么好说的。我们主要说说这样写的缺点：

①、代理对象的一个接口只服务于一种类型的对象，如果要代理的方法很多，势必要为每一种方法都进行代理，静态代理在程序规模稍大时就无法胜任了。

②、如果接口增加一个方法，比如 UserService 增加修改 updateUser()方法，则除了所有实现类需要实现这个方法外，所有代理类也需要实现此方法。增加了代码维护的复杂度。

4. JDK动态代理

4.1 JDK动态代理概念以及使用

> *关于Proxy类（来自JDK 6 API，只有部分，详细内容看JDK API）*
>
> *Proxy提供用于创建动态代理类和实例的静态方法，它还是由于这些方法创建的所有动态代理类的超类。*
>
> *创建一个代理：*

```
//参数handler的是InvocationHandler 接口的实现类的实力对象
Foo f = (Foo)
Proxy.newProxyInstance(Foo.class.getClassLoader(),n
ew Class[] { Foo.class },handler);
```

动态代理类（以下简称为代理类）是一个实现 了 在创建类时在运行时指定的接口列表的类，该类具有下面描述的行为：

1. *代理接口：是动态代理类实现的一个接口。*
2. *代理实例：是动态代理类的一个实例。*
3. *每个代理实例都有一个关联的调用处理程序对象，它可以实现接口InvocationHandler。*
4. *调用某个代理实例上的某个方法（记作Q）时，将被指派到实例的调用处理程序的invoke方法，并给这个invoke方法传递参数：动态代理类的实例、Q方法的Method对象、调用Q方法时传递的参数的Object类型的数组。*
5. *调用处理程序以适当的方式处理编码的方法调用，并且他返回的结果作为代理实例上方法调用的结果返回。*

动态代理类具有以下属性：

1. *动态代理类是公共的、最终的、不是抽象的*
2. *未指定代理类的非限定名称。但是，以字符串“$Proxy”开头的类名空间应该为代理类保留*
3. *动态代理类继承自java.lang.reflect.Proxy*
4. *代理类会按同意顺序准确的实现其创建时指定的接口*
5. *如果代理类实现了非公共的接口，那么他将在该接口相同的报中定义；否则代理类的包也是未指定的。注意，包密封时将不阻止代理类在运行时在特定保重的成功定义，也不会阻止相同类加载器和带有特定签名的包所定义的类*

代理实例具有以下属性：

1. *现有代理实例proxy、java.lang.reflect.Proxy类、一个由proxy的动态代理类实现的Foo接口，以及一个目标类（被代理的类，实现了Foo接口）fooImpl，它们之间的关系：*

   proxy 实现 Foo

   proxy 继承 Proxy

   fooImpl 实现 Foo

2. *代理实例上的接口方法调用将按照该方法的文档描述进行编码，并被指派到调用处理程序的invoke方法中。*

```
public static Object newProxyInstance(ClassLoader
loader,Class<?>[] interfaces,InvocationHandler h)
    方法介绍：返回一个实现了指定接口的动态代理类实例，该接口
可以将方法调用指派到指定的调用处理程序
    参数：
        loader - 定义代理类的类加载器
        interfaces - 代理类要实现的接口列表
        h - 指派方法调用的调用处理程序
    返回：一个带有动态代理类的指定调用处理程序的代理实例，它
由指定的类加载器定义，并实现指定的接口。
```

> 关于接口InvocationHandler接口（来自JDK 6 API）
>
> 这个接口中的一些名词是源自于Proxy中，例如代理实例，代理类.....都和动态代理类相关，具体看上面我对Proxy的部分介绍。
>
> 这个接口是由代理实例（动态代理类的实例）的调用处理程序实现的接口。
>
> 每个代理实例（动态代理类的实例）都有一个关联的调用处理程序。对代理实例（动态代理类的实例）调用方法时，将对方法调用进行编码，并将其指派到它的调用处理程序的invoke方法。
>
> ```
> Object invoke(Object proxy, Method method, Object[]
> args):
>     方法介绍：在代理实例上处理方法调用并返回结果，在与方法关
> 联的代理实例上调用方法时，将在调用处理程序上调用这个invoke方
> 法。
>  参数：
>     procy - 在其上调用方法的代理实例（动态代理类）
>     method - 对应于在代理实例上调用接口方法的Method实例。
> Method对象的声明类将是在其中声明方法的接口，该接口可以是代理
> 类赖以继承方法方法的代理接口的超接口
>     args - 包含传入代理实例上方法调用的参数之的对象数组，如
> 果接口方法不适用参数，则这个之为null。基本类型的参数被包装在
> 适当基本包装器类的实例中（如：Integer、Boolean）
> ```

动态代理就不要自己手动生成代理类了，我们去掉 ProxyUser.java 类，增加一个 ObjectInterceptor.java 类

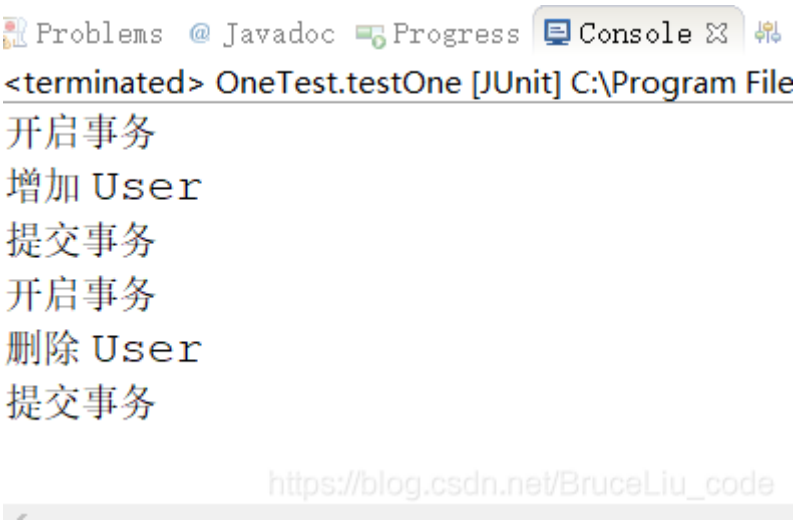
```
import java.lang.reflect.InvocationHandler
/**
设计模式那本书中叫它调用处理程序，
这个处理程序的作用是拦截被代理的类（目标类）UserServiceImpl中的方法
调用
*/
public class ObjectInterceptor implements
InvocationHandler {
```

```
    // 目标类（被代理的类：UserServiceImpl）
    private Object target;
    // 切面类（这里指事务类）
    private MyTransaction transaction;

    // 通过构造器赋值
    public ObjectInterceptor(Object target, MyTransaction
transaction) {
        this.target = target;
        this.transaction = transaction;
    }

    public Object invoke(Object proxy, Method method,
Object[] args) throws Throwable {
        // 开启事务
        this.transaction.before();
        // 调用目标类方法，调用Method中的invoke方法
        method.invoke(this.target, args);
        // 提交事务
        this.transaction.after();
        return null;
    }
}
```

□ 测试类

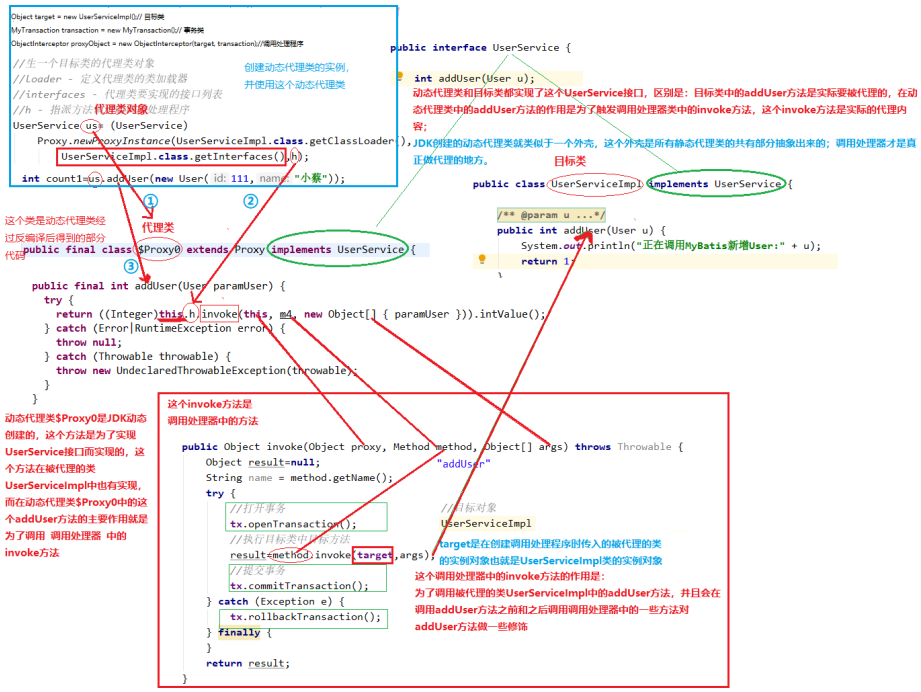
```
public class TestUser2 {

    @Test
    public void testOne() {
        // 目标类
        Object target = new UserServiceImpl();
        // 事务类
        MyTransaction transaction = new MyTransaction();
        //调用处理程序
        ObjectInterceptor proxyObject = new
ObjectInterceptor(target, transaction);
        /**
         * 三个参数的含义： 1、目标类的类加载器 2、目标类所有实现的
接口 3、拦截器
         */
        UserService userService = (UserService)
Proxy.newProxyInstance(target.getClass().getClassLoader(),
target.getClass().getInterfaces(), proxyObject);
        userService.addUser(null);
        userService.deleteUser(11);
    }
}
```

运行结果：



那么使用动态代理来完成这个需求就很好了，后期在 UserService 中增加业务方法，都不用更改代码就能自动给我们生成代理对象。而且将 UserService 换成别的类也是可以的。也就是做到了代理对象能够代理多个目标类，多个目标方法。

4.2 关于动态代理类的原理图



从这张图中可以看出来，JDK动态代理类仅仅是把静态代理类中 静态的、不变的、共有的、可统一定制的 部分抽象出来，这部分抽象出来的内容可以使用JDK中的Proxy API创建出来，而变化的部分通过调用处理器来实现，所以实现代理的地方就是调用处理器；

使用JDK动态代理步骤：

1. 创建调用处理器的时候把被代理类的实例对象、切面类（这里是封装事务操作的类）的实例对象传到调用处理器对象中，之后动态代理对象发出信号时使用调用处理器做代理操作（之所以说是发出信号是因为调用动态代理类中的方法时不会做别的任何操作，仅仅会使调用处理器中的invoke方法被调用）。

2. 创建调用处理器，把 被代理类的实例对象、切面类（这里是封装事务操作的类）的实例对象 传到调用处理器对象中；
3. 调用JDK的 Proxy API 根据 类加载器、被代理的类实现的接口、调用处理器对象 创建JDK动态代理类的实例对象；
4. 调用动态代理类中的方法，发出信号，使调用处理器器中的invoke方法做代理操作（之所以说是发出信号是因为调用动态代理类中的方法时不会做别的任何操作，仅仅会使调用处理器中的invoke方法被调用）。

4.3 查看 JDK 动态代理生成的动态代理类的源代码的步骤

1. 保存 JDK 动态代理的生成的class文件：

```
import sun.misc.ProxyGenerator;
/**
 * 保存 JDK 动态代理生产的类的字节码文件
 * @param filePath 保存路径，默认在项目路径下生成$Proxy0.class
文件
 */
private static void saveProxyFile(String... filePath) {
    FileOutputStream out = null;
    try {
        //第二个参数是目标类的实现的接口
        byte[] classFile =
ProxyGenerator.generateProxyClass("$Proxy0",
IronManVIPMovie.class.getInterfaces());
        //class文件的保存路径
        String path=filePath[0] + "$Proxy0.class";
        System.out.println(path);
        out = new FileOutputStream(path);
        out.write(classFile);
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        try {
            if (out != null) {
                out.flush();
                out.close();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

2. 使用反编译器反编译class文件，得到对应的源代码，视频中使用的反编译器：D:\c_51CTO资料\Java\石头老师Java工程师养成之路（部分）\JAVAEE主流框架之Spring框架实战开发教程(源码+讲义)\3《Spring之代理设计模式》课程资料-01\反编译器\jd-gui.exe

5.使用CGLIB动态代理

5.1 CGLib动态代理概念以及使用

使用JDK创建代理有一个限制,它只能为接口创建代理实例.这一点可以从Proxy的接口方法 `newProxyInstance(ClassLoader loader,Class [] interfaces,InvocarionHandler h)` 中看的很清楚

第二个入参 interfaces就是需要代理实例实现的接口列表.

对于没有通过接口定义业务方法的类,如何动态创建代理实例呢? JDK动态代理技术显然已经黔驴技穷,CGLib作为一个替代者,填补了这一空缺.

CGLib是第三方的框架，不是JDK自带的。

CGLib采用底层的字节码技术,可以为一个类创建子类, 在子类中采用方法拦截的技术拦截所有父类方法的调用并顺势植入横切逻辑.

向Maven项目中导入CGLib依赖：（Spring的和辛堡中已经集成了CGLib所需要的包，所以并不需要另外导入）

```
<dependency>
        <groupId>cglib</groupId>
        <artifactId>cglib</artifactId>
        <version>2.2</version>
</dependency>
```

□ 创建创建CGLib代理器

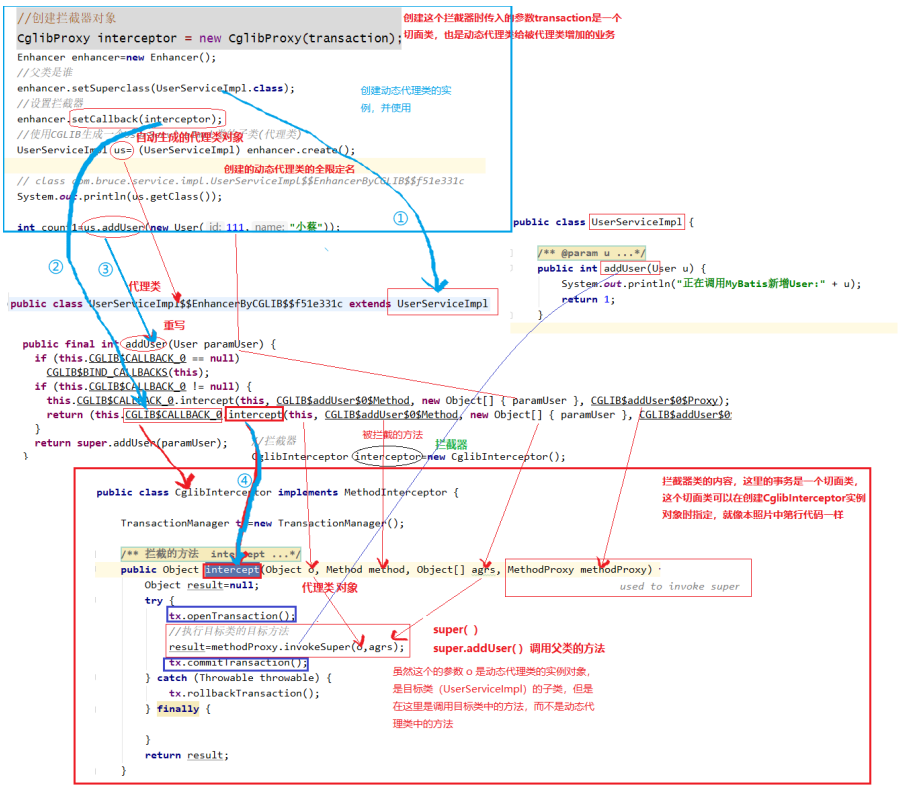
```
import net.sf.cglib.proxy.MethodInterceptor;

/**
目标类中目标方法的拦截器
*/
public class CglibProxy implements MethodInterceptor {

    // 切面类（这里指事务类）
    private MyTransaction transaction;

    public CglibProxy(MyTransaction transaction) {
        this.transaction=transaction;
    }

    /**
    拦截方法
    */
    public Object intercept(Object obj, Method method,
Object[] objects, MethodProxy methodProxy) throws
Throwable {
        transaction.before();
        //执行目标方法
```

```
        Object invoke = methodProxy.invokeSuper(obj,
objects);
        transaction.after();
        return invoke;
    }
}
```

□ 测试类

```
   @Test
    public void testOne() {
        // 事务类
        MyTransaction transaction = new MyTransaction();

        //创建拦截器对象
        CglibProxy interceptor = new
CglibProxy(transaction);
        //字节码加强器，这个类中封装了一个方法可以生成代理对象
        Enhancer enhancer = new Enhancer();
        //指定代理类的父类是谁
        enhancer.setSuperclass(UserServiceImpl.class);
        //给代理对象设置拦截器
        enhancer.setCallback(interceptor);
        //使用CGLib生成目标类的子类（代理类）
        UserServiceImpl userService =
(UserServiceImpl)enhancer.create();

        // class
com.bruce.service.impl.UserServiceImpl$$EnhancerByCGLIB$$f
51e331c
        System.out.println(userService.getClass());

        userService.addUser(null);
        userService.deleteUser(11);
    }
```

5.2 关于CGLib动态代理原理图



从这张图中可以看出来，CGLib动态代理类仅仅是把静态代理类中 静态的、不变的、共有的、可统一定制的 部分抽象出来，这部分抽象出来的内容可以使用CGLib中的API创建出来，而变化的部分通过拦截器来实现，所以真正实现代理的地方就是拦截器；

使用CGLib动态代理步骤：

1. 创建拦截器，把 切面类（这里是封装事务操作的类）的实例对象 传到拦截器对象中；
2. 调用CGLib的API根据 拦截器、被代理的类的Class实例对象 创建CGLib动态代理类的实例对象；
3. 调用动态代理类中的方法，发出信号，使拦截器中的拦截方法做代理操作（之所以说是发出信号是因为调用动态代理类中的方法时不会做别的任何操作，仅仅会使拦截器器中的intercept方法被调用）。

5.3 查看CGLib动态代理的生成的class文件：

1. 保存 CGLib 动态代理的生成的class文件：

```
/**
只要把下面这行代码写到使用CGLib生成动态代理类的代码之前就可以保存代理类的字节码文件；
在指定目录下生成动态代理类，设置环境变量，把CGLib生成的代理类的字节码文件放在指定目录下，这种方式生成的文件不止代理类的字节码文件；
*/
System.setProperty(DebuggingClassWriter.DEBUG_LOCATION_PROPERTY, "D:\\classcglib");

System.out.println("==========CGLIB$CALLBACK_0==========");
Field h = userService.getClass().getDeclaredField("CGLIB$CALLBACK_0");
h.setAccessible(true);
Object obj = h.get(userService);
System.out.println(obj.getClass());
```

2. 使用反编译器反编译class文件，得到对应的源代码，视频中使用的反编译器：D:\c_51CTO资料\Java\石头老师Java工程师养成之路（部分）\JAVAEE主流框架之Spring框架实战开发教程(源码+讲义)\3《Spring之代理设计模式》课程资料-01\反编译器\jd-gui.exe

5.4 补充

1. 在使用CGLib动态代理时记得在intercept方法中确定哪些方法会被代理那些方法不会被代理，否则会出现异常：

   ```
   No operations allowed after connection closed.
   ```
2. 在使用CGLib创建的动态代理类（这个代理类是UserServiceImpl类的子类）的时候，要确保UserServiceDao能成功注入到CGLib创建的动态代理类中；
   因为这涉及到一个顺序：如果Spring先创建UserDao的对象，之后才创建CGLib动态代理类，那么UserServiceDao对象也会传递到CGLib动态代理类中；如果先创建CGLib代理类的实例对象，之后才创建UserServiceDao实例，那么会导致CGLib创建的代理类中UserServiceDao类型的字段为null。

6.JDK和CGLib动态代理总结

6.1.原理区别

java动态代理是利用反射机制生成一个实现代理接口的匿名类，在调用具体方法前调用InvokeHandler来处理。核心是实现InvocationHandler接口，使用invoke()方法进行面向切面的处理，调用相应的通知。

而cglib动态代理是利用asm开源包，将被代理的类（目标类）的class文件加载进来，通过修改其字节码生成目标类的子类（动态代理类）来处理。核心是实现MethodInterceptor接口，使用intercept()方法进行面向切面的处理，调用相应的通知。

6.2.CGlib比JDK快？

1、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，在jdk6之前比使用Java反射效率要高。唯一需要注意的是，**CGLib不能对声明为final的类进行代理，因为CGLib原理是动态生成被代理类的子类。**

2、在jdk6、jdk7、jdk8逐步对JDK动态代理优化之后，在调用次数较少的情况下，JDK代理效率高于CGLIB代理效率，只有当进行大量调用的时候，jdk6和jdk7比CGLIB代理效率低一点，但是到jdk8的时候，jdk代理效率高于CGLIB代理。

3、在对JDK动态代理与CGlib动态代理的代码实验中看，1W次执行下，JDK7及8的动态代理性能比CGlib要好20%左右。

6.3.各自局限

1、JDK的动态代理机制只能代理实现了接口的类，而不能实现接口的类就不能实现JDK的动态代理。

2、cglib是针对类来实现代理的，他的原理是对指定的目标类生成一个子类，并覆盖其中方法实现增强，但因为采用的是继承，所以不能对final修饰的类进行代理。

6.4. Spring中代理的使用

Spring框架中的AOP中的代理实例就是由AOP框架动态生成的一个对象；Spring中的AOP代理，可以是JDK动态代理，也可以使用CGLib动态代理；

1. 如果目标对象实现了接口，默认情况下会采用JDK的动态代理实现AOP
2. 如果目标对象实现了接口，可以强制使用CGLIB实现AOP
3. 如果目标对象没有实现了接口，必须采用CGLIB库，spring会自动在JDK动态代理和CGLIB之间转换

Spring中AOP代理默认使用的时JDK动态代理的方式来实现；可以强制使用CGlib（如果目标对象实现了接口的情况下）：
在spring配置中加入`<aop:aspectj-autoproxy proxy-target-class=“true”/>`）
在springboot项目配置： `spring.aop.proxy-target-class=false`

8. 总结

动态代理 VS 静态代理：

类型	优点	缺点
静态代理		需要手动创建代理类，代码冗余

类型	优点	缺点
动态代理	减少代码冗余，代理类由JDK动态创建，可以在代理类中指定那些方法可以被代理	

JDK动态代理 VS CGLIB动态代理：

类型	机制	回调方式	适用场景	效率
JDK动态代理	委托机制，代理类和目标类都实现了同样的接口，InvocationHandler持有目标类，代理类委托InvocationHandler去调用目标类原始方法	反射	目标类实现了接口	效率瓶颈在反射调用稍慢
CGLIB动态代理	继承机制，代理类继承了目标类并重写了目标方法，通过回调函数MethodInterceptor调用父类方法执行原始逻辑	通过FastClass方法索引调用	非接口类，非final类，非final方法	第一次调用因为要生成多个Class对象较JDK慢，多次调用因为方法索引较反射方式快，如果方法多swtich case过多其效率还需测试