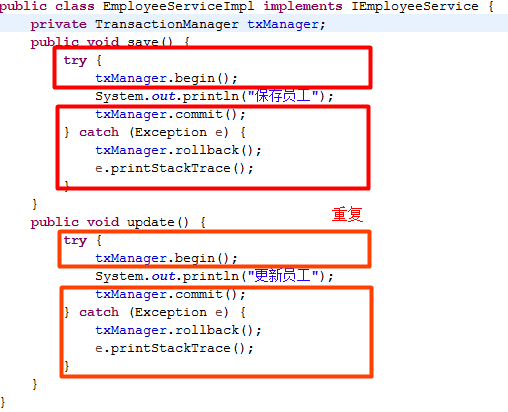
## Spring(三)

### 案例分析

案例:每一个业务方法都要**处理事务**.



**存在的问题**:

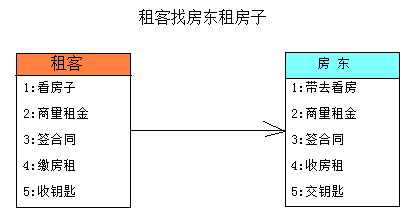
每一个Service实现类中的方法,都得需要处理事务,而这些处理事务的代码结构其实是相同的,存在着代码重复的.

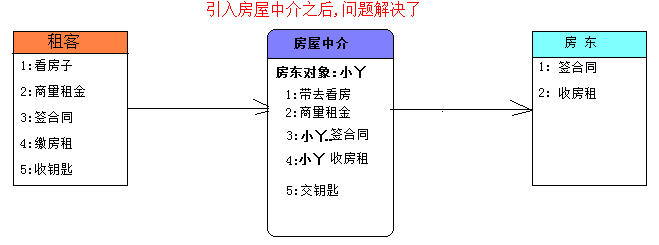
问题1: 处理事务的代码结构重复.

问题2: 在业务方法中只需要处理业务功能即可,不该处理事务(责任分离思想).

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

从生活中去寻找解决方案,所以第一步找一个相似的案例:房屋中介案例





房东做了很多重复的并且不应该自己去做的事情:带你看房,商量租金,交钥匙.

**她最感兴趣的是签合同和收房租**.

**房东就好比是Service实现类**.

### 代理模式

真实生活中有一种房屋中介是这样的，租客根本就不知道房东是谁，一切签合同、交租金、交钥匙等操作都直接和中介公司发生。我们把这种模式称之为代理模式。

代理模式：客户端直接使用的都是代理对象，不知道目标对象是谁，此时代理对象可以在客户端和目标对象之间起到中介的作用。

1、代理对象完全包含目标对象，客户端使用的都是代理对象的方法，和目标对象没有直接关系；

2、代理模式的职责：把不是目标对象该做的事情从目标对象上撇开——职责清晰；

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

静态代理：在程序运行前就已经存在代理类的字节码文件，代理对象和目标对象的关系在运行前就确定了。

动态代理：动态代理类是在程序运行期间由JVM通过反射等机制动态的生成的，所以不存在代理类的字节码文件。代理对象和真实对象的关系是在程序运行事情才确定的。

### 静态代理

静态代理：在程序运行前就已经存在代理类的字节码文件，代理对象和真实对象的关系在运行前就确定了。

静态代理优缺点:

优点：

1.业务类只需要关注业务逻辑本身，保证了业务类的重用性。

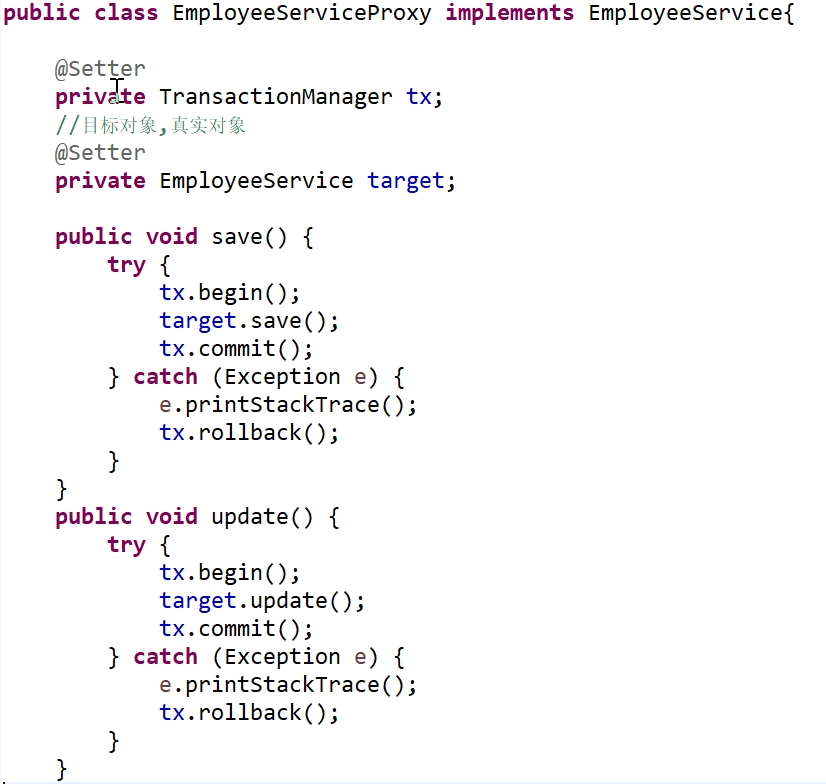
2.把真实对象隐藏起来了,保护真实对象

缺点：

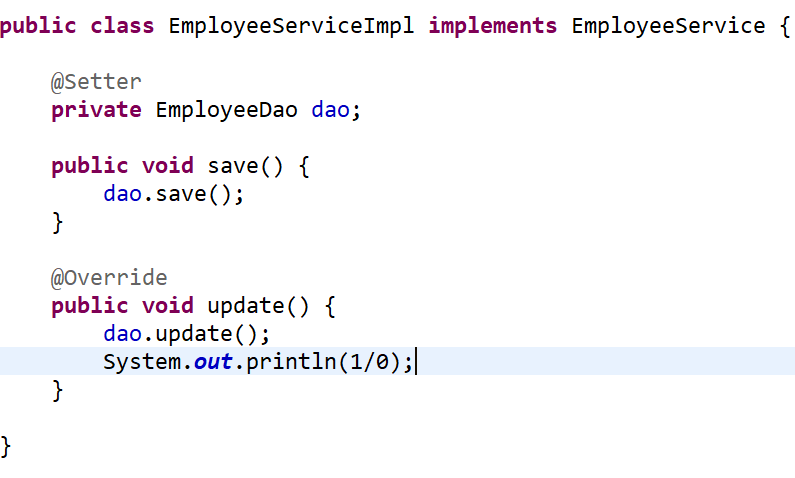
 1.代理对象的某个接口只服务于某一种类型的对象，也就是说每一个真实对象都得创建一个代理对象。

 2.如果需要代理的方法很多，则要为每一种方法都进行代理处理。

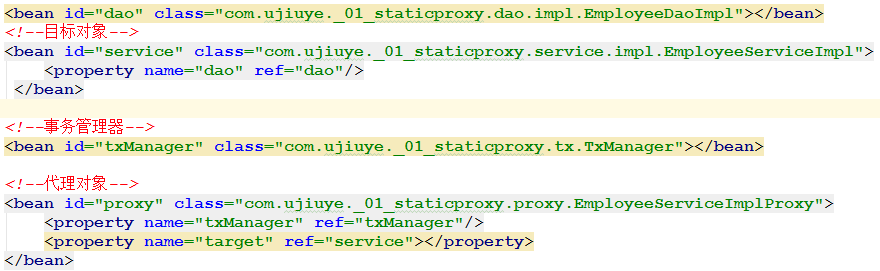
 3.如果接口增加一个方法，除了所有实现类需要实现这个方法外，所有代理类也需要实现此方法。增加了代码维护的复杂度。



EmployeeServiceProxy类



EmployeeServiceImpl类



小结:

1.静态代理是指在程序运行之前就存在代理对象的字节码(存在代理对象的类).

2.一个代理类只能代理一种类型的被代理类.

3.静态代理可以实现责任分离的目标.

4.静态代理增加了代码维护的成本.

### JDK代理

动态代理是在程序运行期间由JVM通过反射等机制动态的创建出代理对象的字节码。代理对象和真实对象的关系是在程序运行时才确定的。

JDK动态代理API分析:

1、java.lang.reflect.Proxy 类:

Java 动态代理机制生成的所有动态代理类的父类，它提供了一组静态方法来为一组接口动态地生成代理类及其对象。

主要方法：

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

方法职责：为指定类加载器、一组接口及调用处理器生成动态代理类实例

参数:

loader :类加载器

interfaces :目标对象实现的接口

hanlder :代理执行处理器

返回:动态生成的代理对象

2、java.lang.reflect.InvocationHandler接口:

public Object invoke(Object proxy, Method method, Object[] args)

方法职责：负责集中处理动态代理类上的所有方法调用

参数:

proxy :生成的代理对象

method :当前调用的真实方法对象

args :当前调用方法的实参

返回: 真实方法的返回结果

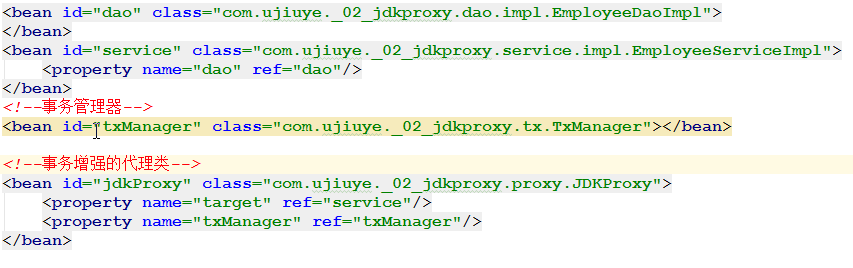
------------------------------------------------------------------------------------

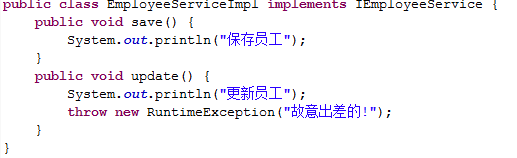
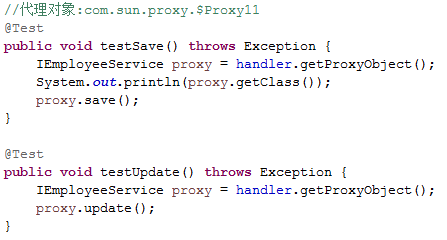
jdk动态代理操作步骤

①　实现InvocationHandler接口，创建自己增强代码的处理器。

②　给Proxy类提供ClassLoader对象和代理接口类型数组，创建动态代理对象。

③　在处理器中实现增强操作。

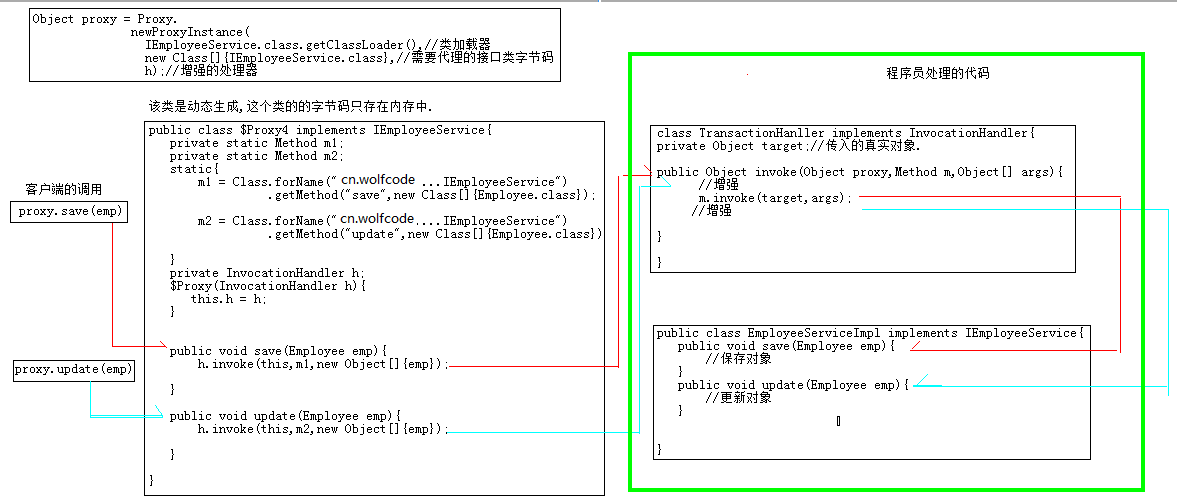




通过代理对象调用save方法

调用真实对象中的save方法

创建代理对象



JDK动态代理:

1,代理的对象必须要实现接口

2,需要为每个对象创建代理对象;

3,动态代理的最小单位是类(类中所有的方法都会被代理);

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

JDK动态代理总结：

1,JAVA动态代理是使用java.lang.reflect包中的Proxy类与InvocationHandler接口这两个来完成的。

2,要使用JDK动态代理，必须要定义接口。

3,JDK动态代理将会拦截所有pubic的方法（因为只能调用接口中定义的方法），这样即使在接口中增加了新的方法，不用修改代码也会被拦截。

4,如果只想拦截一部分方法，可以在invoke方法中对要执行的方法名进行判断。

### CGLib代理

针对于没有接口的类,如何做代理:cglib都是针对没有接口,做动态代理的.

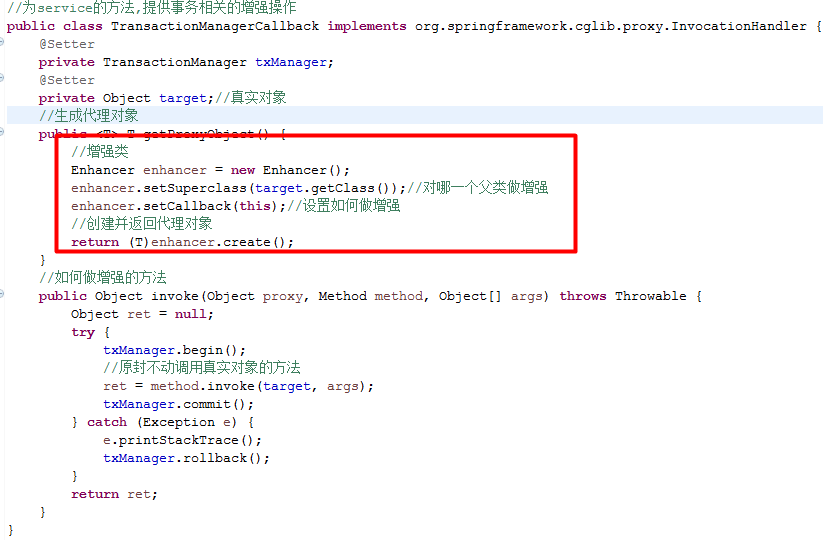
原理是对指定的目标类生成一个子类，并覆盖其中方法实现增强，但因为采用的是继承，所以不能对final修饰的类进行代理。

CGLIB代理总结：

1,CGLIB可以生成目标类的子类，并重写父类非final修饰符的方法。

2,要求类不能是final的，要拦截的方法要是非final、非static、非private的。

3,动态代理的最小单位是类(所有类中的方法都会被处理);



在Spring中：

若目标对象实现了若干接口，Spring就会使用JDK动态代理。

若目标对象没有实现任何接口，Spring就使用CGLIB库生成目标对象的子类。

对接口创建代理优于对类创建代理，因为会产生更加松耦合的系统，也更符合面向接口编程规范。

### Spring的AOP思想

**Spring的AOP**:

**什么叫做AOP:Aspect oritention programming(面向切面编程)**

把一个个的横切关注点（这些零散存在于业务方法中的功能代码，我们称之为横切面关注点）放到某个模块中去，称之为切面。那么每一个的切面都能影响业务的某一种功能，切面的目的就是功能增强，如日志切面就是一个横切关注点，应用中许多方法需要做日志记录的只需要插入日志的切面即可。这种面向切面编程的思想就是AOP思想了。

AOP的目的：

AOP能够将那些与业务无关，却为业务模块所共同调用的逻辑或责任（例如事务处理、日志管理、权限控制等）封装起来，

便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可拓展性和可维护性。

AOP的优势：

降低模块的耦合度、使系统容易扩展、更好的代码复用性

说人话: 把多个方法前/后的共同代码抽离出来,使用动态代理机制来控制,先执行抽离出来的代码,再执行每一个真实方法.

AOP当中的概念:

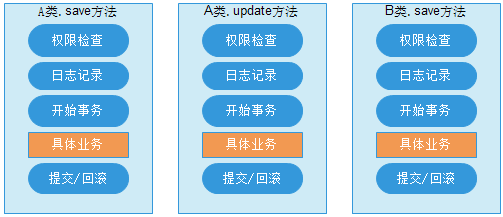
Joinpoint:连接点，被拦截到需要被增强的方法。where：去哪里做增强

Pointcut：切入点，哪些包中的哪些类中的哪些方法，可认为是连接点的集合。where:去哪些地方做增强

Advice：增强，当拦截到Joinpoint之后，在方法执行的什么时机（when）做什么样(what)的增强。

Aspect：切面，Pointcut+Advice，去哪些地方+在什么时候+做什么增强

Weaving：织入，把Advice加到Target上之后，创建出Proxy对象的过程。





**Pointcut语法:**

AOP思想本应该由SUN公司来制定规范,但是被AOP联盟捷足先登了.

AOP联盟制定AOP规范，首先就要解决一个问题，怎么表示在哪些方法上增强—— **AspectJ(语言)**。

**AspectJ切入点语法如下**：

execution(modifiers-pattern? ret-type-pattern declaring-type-pattern? name-pattern(param-pattern)throws-pattern?)

翻译成中文：

execution(<访问修饰符>? <返回类型> <声明类型>? <方法名>(<参数>) <异常>?)

举例：

public static Class java.lang.Class.forName(String className)throws ClassNotFoundException

?:表示可以出现一次,或者不出现.没有问号,表示必须出现一次.

通配符:

\*：匹配任何部分,只能表示一个单词

..: 可用于全限定名中和方法参数中，分别表示子包和0到N个参数





### XML方式实现AOP

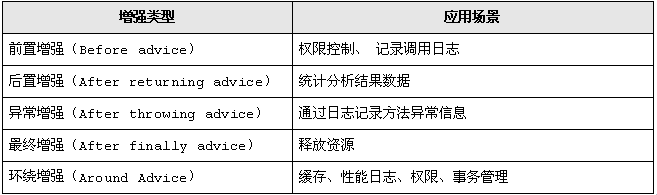
1. 导包

com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar

1. 新增aop命名空间
2. 配置aop



### AOP不同时机的增强



**各种不同的增强**：

aop:before（前置增强）：在方法执行之前执行增强；

aop:after-returning（后置增强）：在方法正常执行完成之后执行增强；

aop:after-throwing（异常增强）:在方法抛出异常退出时执行增强；

aop:after（最终增强）：在方法执行之后执行，相当于在finally里面执行；可以通过配置throwing来获得拦截到的异常信息

aop:around（环绕增强）：最强大的一种增强类型。 环绕增强可以在方法调用前后完成自定义的行为，环绕通知有两个要求，

1，方法必须要返回一个Object（返回的结果）

2，方法的第一个参数是ProceedingJoinPoint（可以继续向下传递的切入点）

### 注解方式

1. 配置文件中添加aop的注解解析器
2. 在切面类上,添加@Aspect注解.
3. 在横切关乎点所在的类,的方法上,添加什么时候执行额注解.

