**Spring 第三天**

* 完善我们的Account案例
* 分析案例中的问题
* 动态代理（难点）
* 动态代理的另外一种实现方式
* 解决案例中的问题
* AOP的概念
* Spring中AOP相关的术语
* Spring基于XML和注解的AOP配置

## 1.1 添加转账案例

我们在案例中，添加转账业务。

### 1.1.1 把上节内容的代码全部复制拷贝过来

* Pojo

|  |
| --- |
| **public class** Account {  **private** Integer **id**;  **private** String **name**;  **private** Double **money**;   **public** Integer getId() {  **return id**;  }   **public void** setId(Integer id) {  **this**.**id** = id;  }   **public** String getName() {  **return name**;  }   **public void** setName(String name) {  **this**.**name** = name;  }   **public** Double getMoney() {  **return money**;  }   **public void** setMoney(Double money) {  **this**.**money** = money;  }   @Override  **public** String toString() {  **return "Account{"** +  **"id="** + **id** +  **", name='"** + **name** + **'\''** +  **", money="** + **money** +  **'}'**;  } } |

* 配置文件

|  |
| --- |
| <**bean id="accountDao" class="com.qf.dao.impl.AccountDaoImpl"**>  <**property name="runner" ref="runner"**></**property**> </**bean**>  <**bean id="accountService" class="com.qf.service.impl.AccountServiceImpl"**>  <**property name="accountDao" ref="accountDao"**></**property**> </**bean**>  <**bean id="runner" class="org.apache.commons.dbutils.QueryRunner" scope="prototype"**>  *<!--注入数据源-->* <**constructor-arg name="ds" ref="dataSource"**></**constructor-arg**> </**bean**>  *<!-- 配置数据源 -->* <**bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"**>  *<!--连接数据库的必备信息-->* <**property name="driverClass" value="com.mysql.jdbc.Driver"**></**property**>  <**property name="jdbcUrl" value="jdbc:mysql://192.168.10.137:3306/ssm"**></**property**>  <**property name="user" value="root"**></**property**>  <**property name="password" value="Admin123!"**></**property**> </**bean**> |

### 1.1.2 在上面代码中添加转账业务

* AccountDao

|  |
| --- |
| **public interface** AccountDao {  *//根据账户名称查询对应的账户信息* **public** Account findAccountByName(String name);   *//修改账户信息* **public void** updateAccount(Account account); } |

* AccountDaoImpl

|  |
| --- |
| **public class** AccountDaoImpl **implements** AccountDao {  **private** QueryRunner **runner**;  //使用set方法注入  **public void** setRunner(QueryRunner runner) {  **this**.**runner** = runner;  }  */\*\*  \* 根据id查找账户信息  \** ***@param name*** *\** ***@return*** *\*/* **public** Account findAccountByName(String name) {  **try** {  List<Account> list = **runner**.query(**"select \* from account where name=?"**,**new** BeanListHandler<Account>(Account.**class**),name);  **if**(list == **null** || list.size() == 0){  **return null**;  }  **if**(list.size() > 1){  **throw new** RuntimeException();  }  **return** list.get(0);  } **catch** (SQLException e) {  e.printStackTrace();  **throw new** RuntimeException(e);  }  }   */\*\*  \* 修改账户信息  \** ***@param account*** *\*/* **public void** updateAccount(Account account) {  String sql = **"update account set money=? where name=?"**;  **try** {  **runner**.update(sql,account.getMoney(),account.getName());  } **catch** (SQLException e) {  e.printStackTrace();  }  } } |

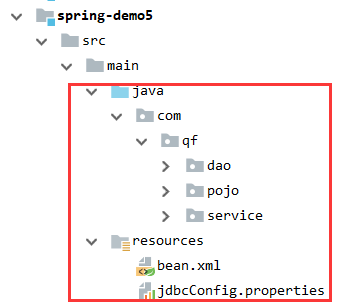
* AccountService

|  |
| --- |
| **public interface** AccountService {   */\*\*  \* 转账操作  \** ***@param sourceName*** *转出账户名称  \** ***@param targetName*** *转入账户名称  \** ***@param money*** *转账金额  \*/* **public void** transfer(String sourceName,String targetName,Double money); } |

* AccountServiceImpl

|  |
| --- |
| **public class** AccountServiceImpl **implements** AccountService {   **private** AccountDao **accountDao**;  //使用set方法注入  **public void** setAccountDao(AccountDao accountDao) {  **this**.**accountDao** = accountDao;  }  **public void** transfer(String sourceName, String targetName, Double money) {  *//根据名称查询出转账用户* Account sourceAccount = **accountDao**.findAccountByName(sourceName);  *//根据名称查询转入账用户* Account targetAccount = **accountDao**.findAccountByName(targetName);  *//转出账户扣钱* sourceAccount.setMoney(sourceAccount.getMoney() - money);  *//转入账户加钱* targetAccount.setMoney(targetAccount.getMoney() + money);  *//更新转出账户* **accountDao**.updateAccount(sourceAccount);  *//int i = 1/0;  //更新转入账户* **accountDao**.updateAccount(targetAccount);  } } |

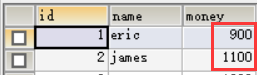
工程结构目录如下:



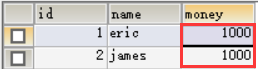
### 1.1.3 测试

|  |
| --- |
| @RunWith(SpringJUnit4ClassRunner.**class**) @ContextConfiguration(locations = {**"classpath:bean.xml"**}) **public class** TestAccount {   @Autowired  AccountService **accountService**;   @Test  **public void** testAccount(){  ApplicationContext context = **new** ClassPathXmlApplicationContext(**"bean.xml"**);  AccountService accountService = (AccountService) context.getBean(**"accountService"**);  accountService.transfer(**"eric"**,**"james"**,100.0);  } } |

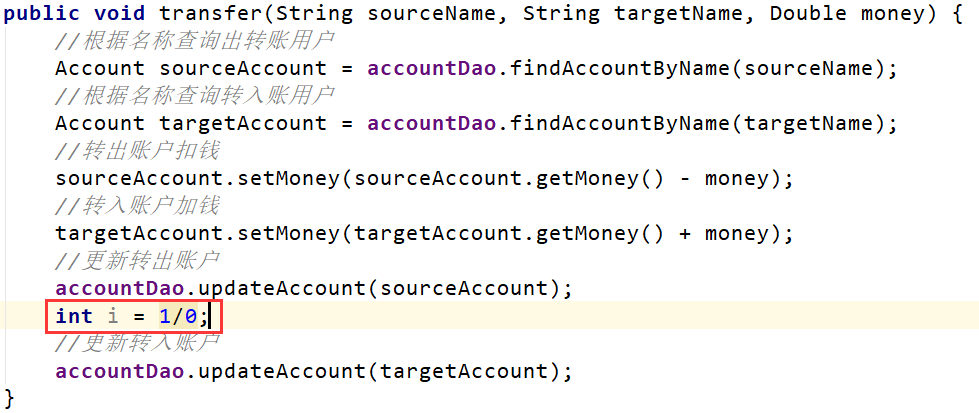
此时我们观察数据表里面的变化情况：



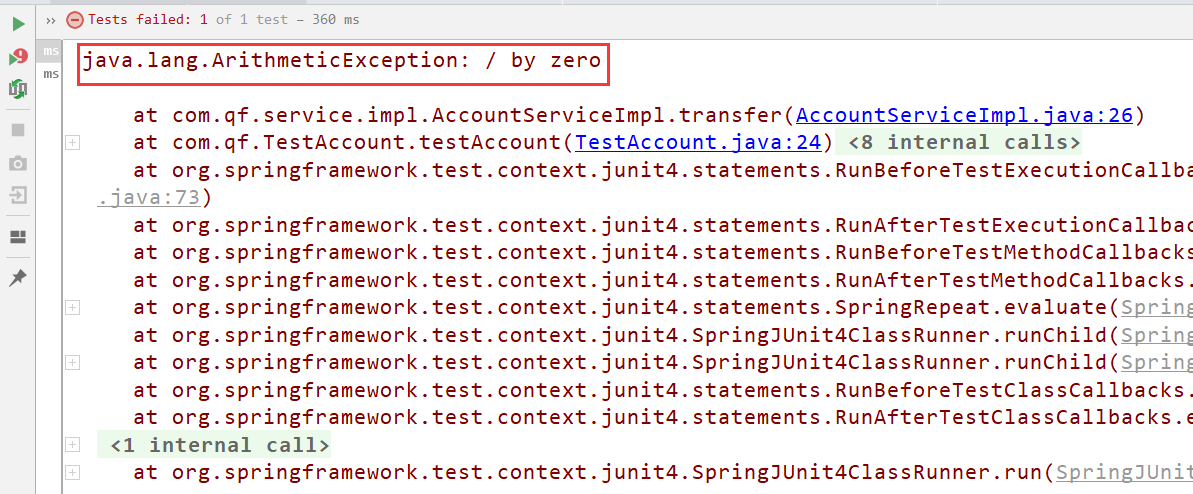
转账是成功的，但是涉及到业务的问题，如果业务层实现类有其中一个环节出问题，都会导致灾难。我们先把数据恢复到转账前。



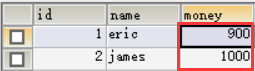
现在我们故意模拟转账业务出现问题。



再来测试：



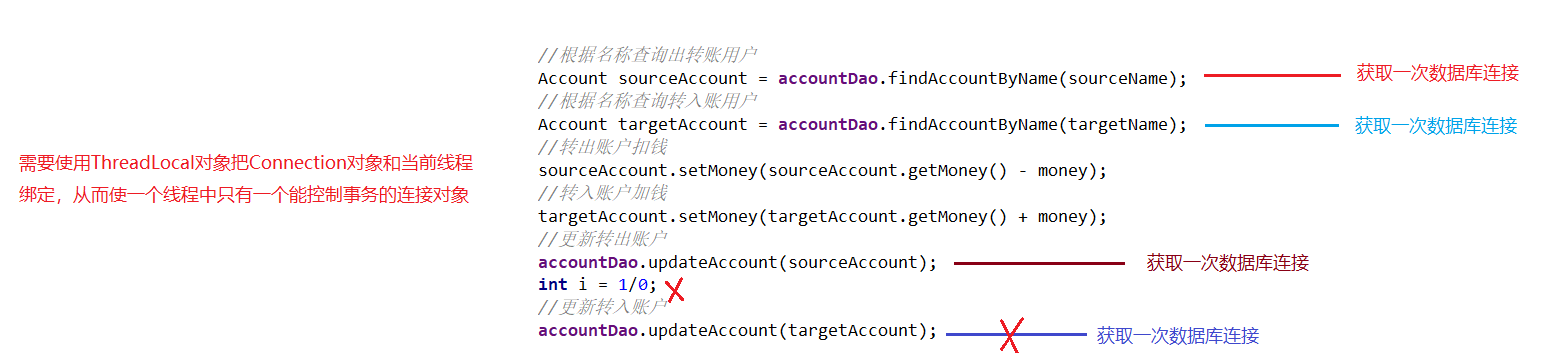
业务执行出错，但是！



这是因为：不满足事务的一致性（减钱的事务提交了，加钱的事务没有提交，甚至都没有执行到）。

## 1.2 改造案例

分析造成事务不一致的问题：



我们的事务控制应该写在业务层实现类，在那之前我们先写两个工具类。

### 1.2.1 工具类的编写

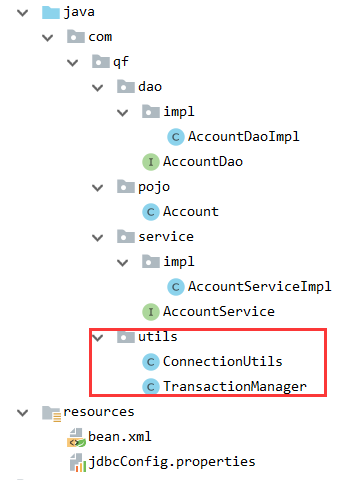
* 从数据源中获取一个连接对象，并和当前线程绑定

|  |
| --- |
| */\*\*  \* 连接的工具类，它用于从数据源中获取一个连接，并且实现和线程的绑定  \*/* **public class** ConnectionUtils {   **private** ThreadLocal<Connection> **tl** = **new** ThreadLocal<Connection>();   **private** DataSource **dataSource**;  //使用set方法进行注  **public void** setDataSource(DataSource dataSource) {  **this**.**dataSource** = dataSource;  }   */\*\*  \* 获取连接对象  \** ***@return*** *\*/* **public** Connection getThreadConnection(){  **try** {  Connection connection = **tl**.get();  **if**(connection == **null**){*//没有连接  // 需要从数据库中获取一个连接对象* connection = **dataSource**.getConnection();  **tl**.set(connection);  }  **return** connection;   } **catch** (SQLException e) {  **throw new** RuntimeException(e);  }  }   */\*\*  \* 将当前线程和连接对象解绑  \*/* **public void** removeConnection(){  **tl**.remove();  } } |

* 事务管理相关的工具类，它包含了开启事务、提交事务、回滚事务和释放连接等方法。

|  |
| --- |
| */\*\*  \* 和事务管理相关的工具类，它包含了开启事务、提交事务、回滚事务和释放连接等方法  \*/* **public class** TransactionManager {  **private** ConnectionUtils **connectionUtils**;   **public void** setConnectionUtils(ConnectionUtils connectionUtils) {  **this**.**connectionUtils** = connectionUtils;  }   */\*\*  \* 开启事务  \*/* **public void** beginTransaction() {  **try** {  *//获取绑定了当前线程的连接，并且关闭自动提交事务* **connectionUtils**.getThreadConnection().setAutoCommit(**false**);  } **catch** (Exception e) {  e.printStackTrace();  }  }   */\*\*  \* 提交事务  \*/* **public void** commit() {  **try** {  **connectionUtils**.getThreadConnection().commit();  } **catch** (Exception e) {  e.printStackTrace();  }  }   */\*\*  \* 回滚事务  \*/* **public void** rollback() {  **try** {  **connectionUtils**.getThreadConnection().rollback();  } **catch** (Exception e) {  e.printStackTrace();  }  }   */\*\*  \* 释放连接  \*/* **public void** release() {  **try** {  *//归还到服务器线程池中 此时当前线程和数据库连接对象并没有解绑* **connectionUtils**.getThreadConnection().close();  *//解绑* **connectionUtils**.removeConnection();  } **catch** (Exception e) {  e.printStackTrace();  }  }  } |

最后结构目录如下:



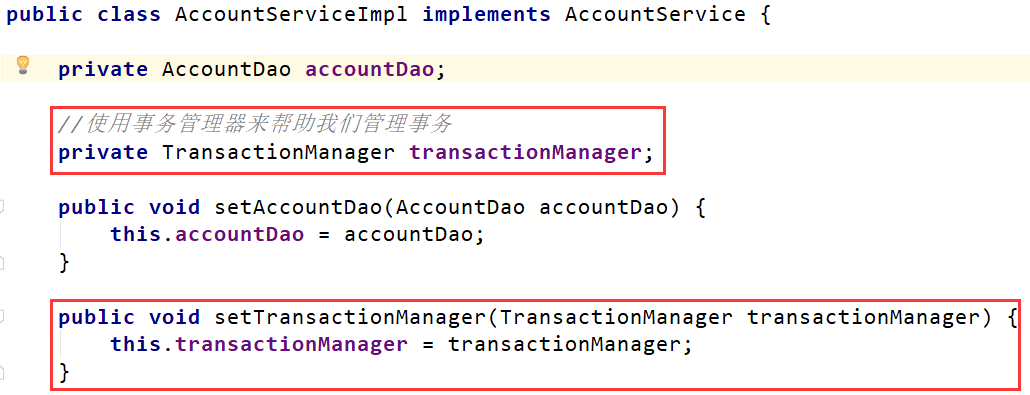
### 1.2.2 编写业务层和持久层事务控制代码

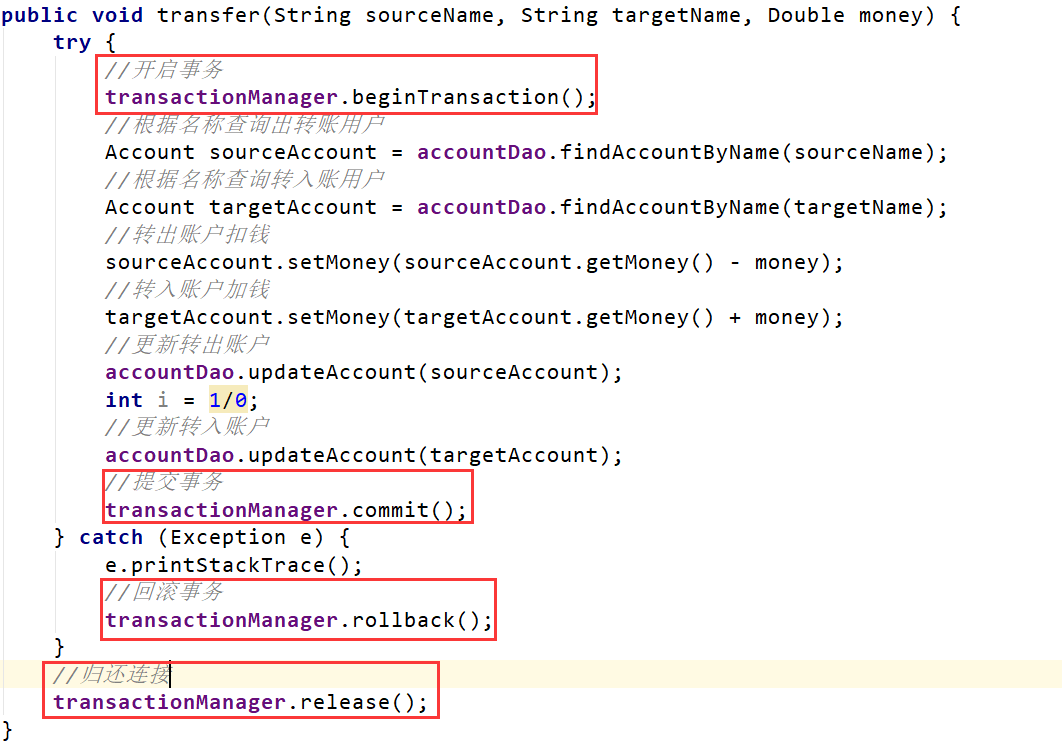
现在我们在业务层上需要做两个事情:

1、在业务实现类上面 注入事务管理器。

2、在业务代码上进行事务的操作。

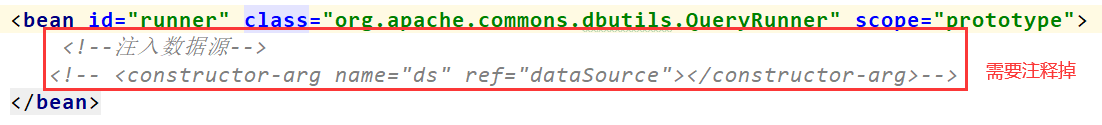
具体操作如下:





接下来解决持久层的事务问题，先看看下面的分析。

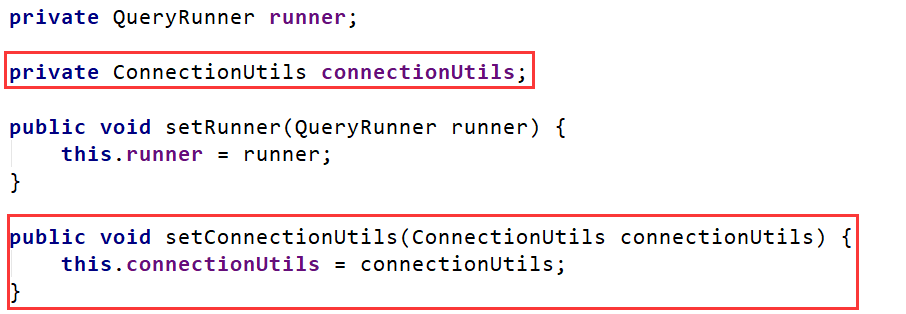
此时我们依然无法控制事务，因为我们的QueryRunner构造时传入了一个连接池对象，如下：



如果构造QueryRunner时传入了一个连接池对象，以后每条语句都会从构造时传入的连接池中获取链接对象，这样每条语句都有单独的连接对象就不能控制事务了。

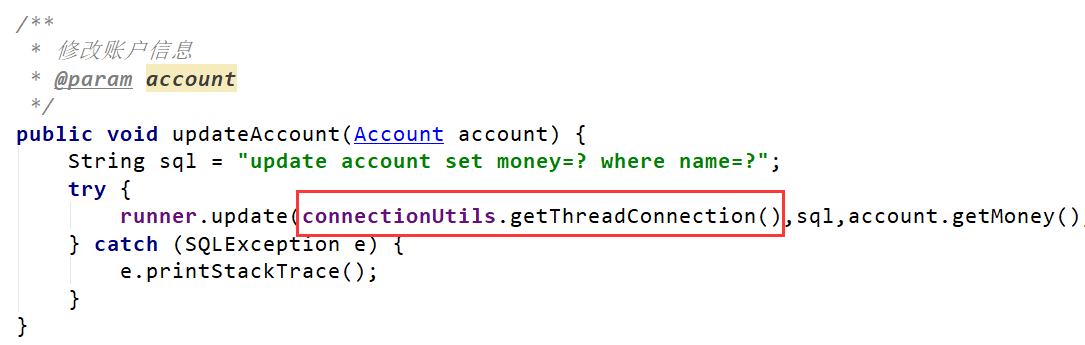
我们同样让spring用set方法注入ConnectionUtils对象，以获取Connection对象

AccountDaoImpl.java：



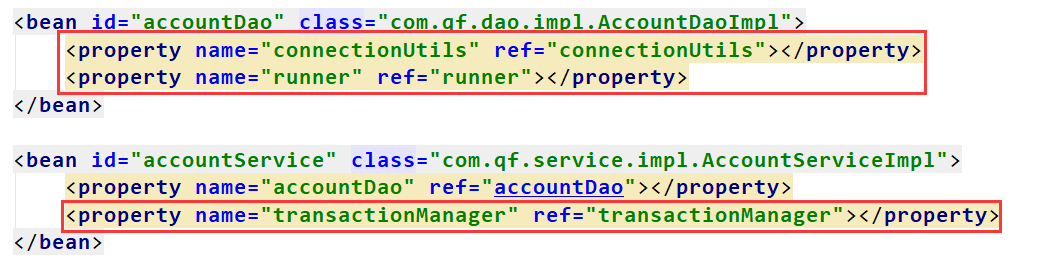
然后给每一个方法传递连接对象，如下：





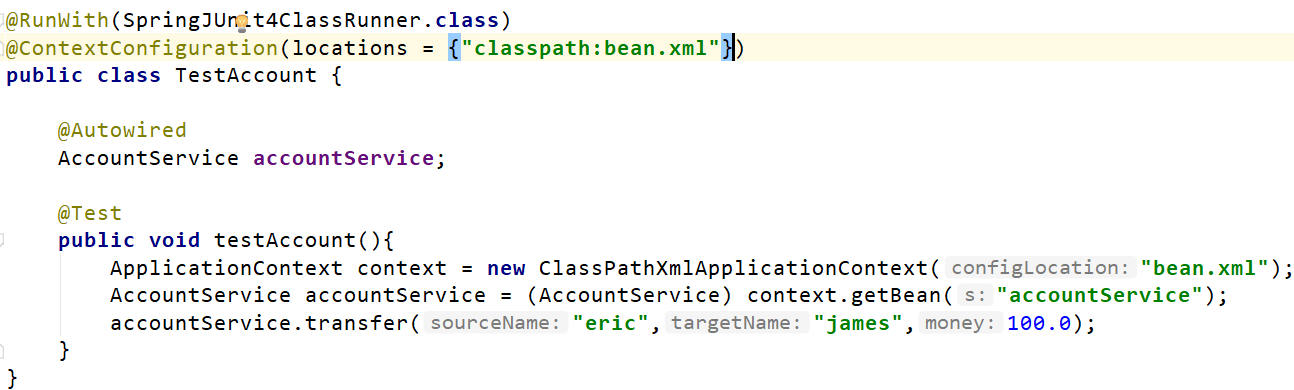
### 1.2.3 完善配置文件

将各个类的依赖关系配置出来：

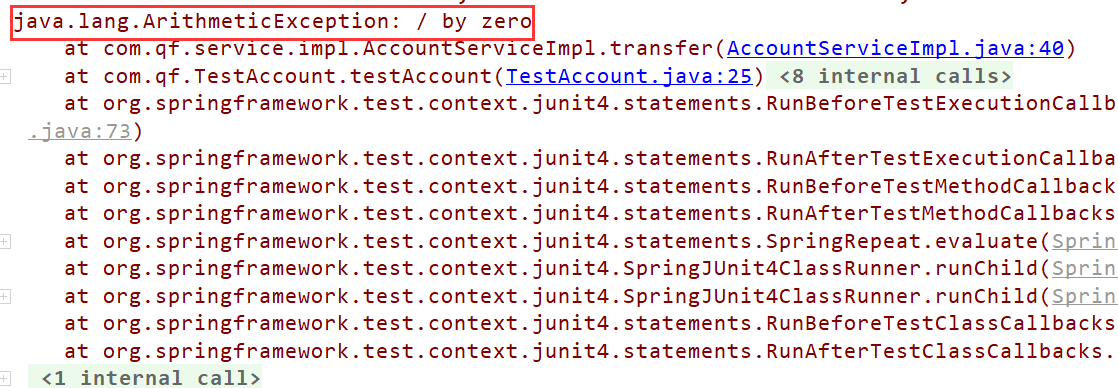


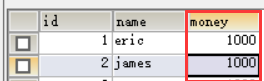


### 1.2.4 测试并查看效果



此时控制台报出异常信息：



然后查看数据库数据变化情况:  


我们发现数据并没有发生变化。说明我们的事务控制成功。

**总结：**

AccountDaoImpl需要用到ConnectionUtils，AccountServiceImpl需要用到TransactionManager，依赖关系非常复杂。

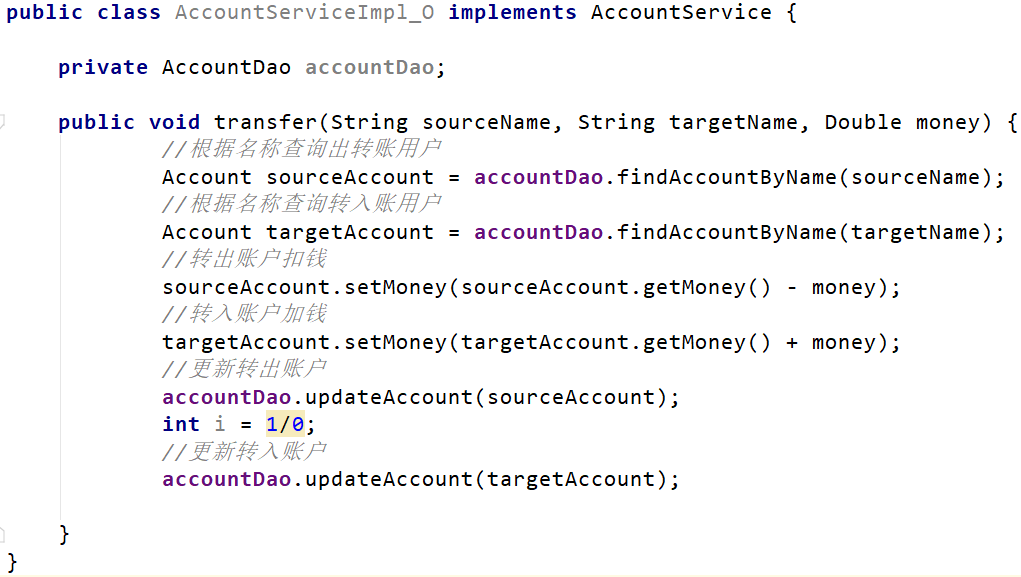
我们现在虽然实现了事务控制，但是代码非常的臃肿，我们要简化代码。

当然Spring是集成了事务控制的，但是我们先不用Spring的，先自己尝试简化代码。

我们理想中的业务层实现类代码应该是这个样子的，如下：

（下面这个类当作是一个新的类，名字为AccountServiceImpl\_0，而之前那个旧的类名还是为AccountServiceImpl）。

AccountServiceImpl\_0：

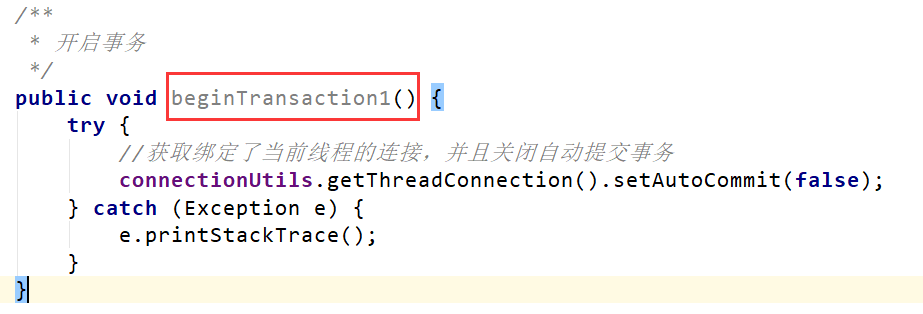


其实这就是我们最原始的业务层实现类代码，如果不涉及事务问题还好，但是后面的transfer()方法显然需要控制事务，这种写法的问题先然是很严重的，那我们有什么办法解决吗?

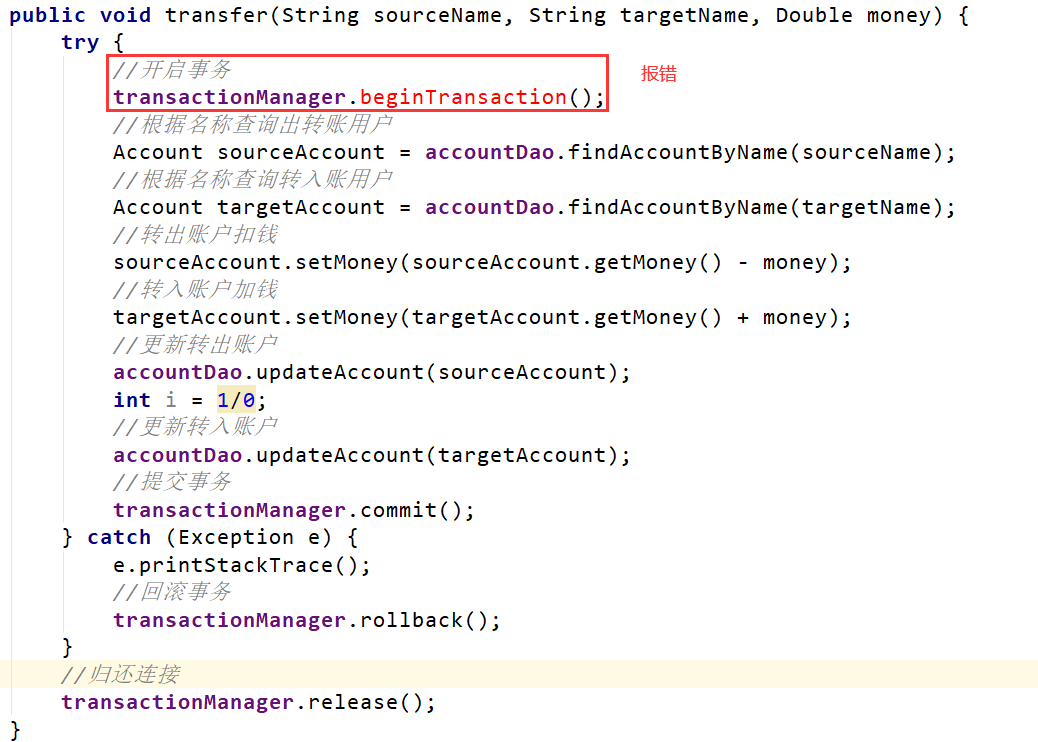
我们在Spring的第一天提到过，依赖分两种：类之间的依赖和方法之间的依赖。

关于类之间的依赖我们已经学会使用bean.xml或者注解去解决了，那么方法之间依赖怎么解决?

我们先看看方法之间的依赖是怎么一回事：我们把TransactionManager的beginTransaction()方法改成beginTransaction1()方法



发现AccountServiceImpl的transfer方法都报错，而AccountServiceImpl\_0没有任何报错。



如果想要不报错，就要对AccountServiceImpl的每一个方法都做出修改，这是相当麻烦的。

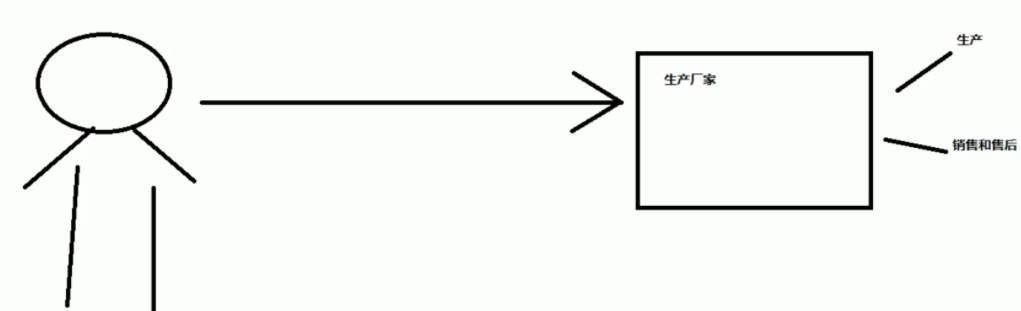
并且这只是一个Service，如果有上百个Service依赖于TransactionManager的beginTransaction()方法，那修改起来是相当麻烦的。

这说明AccountServiceImpl具有很强的方法之间的耦合，这是不可取的，我们可以用代理解决。

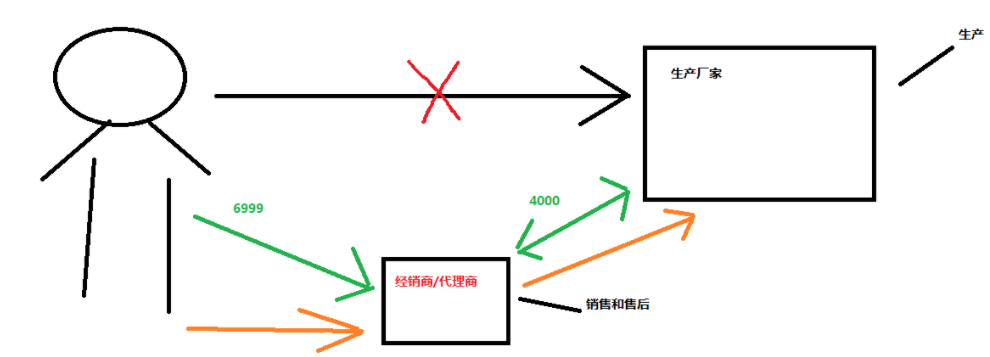
## 1.3 代理的分析

我们接下来看一个例子来解释什么是动态代理。

早期，电脑生产厂家同时负责生产和销售和售后三个服务。



后期，电脑生产厂家生意越做越大，发现无法同时三个服务，于是找了一些代理商。



此时代理商负责销售和售后两个服务，而生产厂家只负责生产，这样子生产厂家的生意也越做越大了

这相当于：使用动态代理增强了生产厂家对象的功能。

### 1.3.1 jdk动态代理

Jdk动态代理的概念：

* 特点：代理对象在程序的运行过程中创建。
* 作用：**不修改源码的基础上对方法增强。**
* 分类：

基于接口的动态代理（现在讲）jdk

基于子类的动态代理（待会儿讲）cglib

* 基于接口的动态代理：

涉及的类：Proxy

提供者：JDK官方

* 如何创建代理对象：

使用Proxy类中的newProxyInstance方法

* 创建代理对象的要求：

被代理类最少实现一个接口，如果没有则不能使用

* newProxyInstance方法的参数：
* ClassLoader：类加载器：

它是用于加载代理对象字节码的，和被代理对象使用相同的类加载器。是固定写法（写被代理对象的类加载器）

* Class[]：字节码数组

它是用于让代理对象和被代理对象有相同的方法（只要两者都实现了同一个接口，那么两者的方法必然相同，所以我们传接口的字节码文件即可）。固定写法（写接口字节码文件）

* InvocationHandler：处理器

用于提供增强的代码，它是让我们写如何代理，**我们一般都是写一个该接口的实现类。**通常情况下都是匿名内部类，但不是必须的，此接口的实现类都是谁用谁写。

搞清楚动态代理的概念之后，接下来我们开始代码编写。

需求: 我们可以自己去厂家买电脑，同时也可以让代理帮我们买电脑。

**让代理买电脑其实就是对我们自己买电脑的增强。**

我们可以自己买电脑，也可以让代理帮我们买电脑，那我们索性就把买电脑的方法抽取出来，形成一个接口。

|  |
| --- |
| **public interface** ProductDao {  **public void** buyProduct(String name); } |

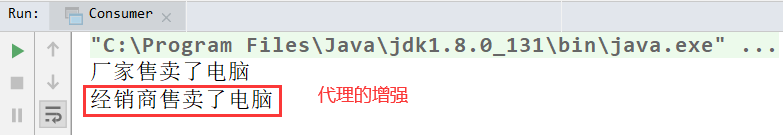
我们模拟自己买电脑

|  |
| --- |
| */\*\*  \* 模拟我们从厂家直接买电脑  \*/* **public class** Producer **implements** ProductDao {  **public void** buyProduct(String name) {  System.***out***.println(name + **"售卖了电脑"**);  } } |

然后我们模拟代理帮我们买电脑

|  |
| --- |
| */\*\*  \* 消费者  \* 分别直接买  \* 让代理买  \*/* **public class** Consumer {  **public static void** main(String[] args) {  *//从厂家直接买电脑* **final** Producer producer = **new** Producer();  producer.buyProduct(**"厂家"**);   ***//模拟让代理买电脑* ProductDao proxy = (ProductDao) Proxy.*newProxyInstance*(Producer.class.getClassLoader(), producer.getClass().getInterfaces(), new InvocationHandler() {  */\*\*  \* 执行被代理接口的任何方法 都会执行该方法  \* @param proxy 当前代理对象的引用 一般很少用  \* @param method 当前执行的方法  \* @param args 当前执行方法 需要的参数  \* @return 和被代理对象一样的返回值  \* @throws Throwable  \*/* public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  if(method.getName().equals("buyProduct")){  method.invoke(producer,args);  }  return null;  }  });  proxy.buyProduct("经销商");**  } } |

最后控制台效果:



### 1.3.2 基于子类的动态代理

那我们要如何代理一个普通的Java类呢？我们可以使用基于子类的动态代理。

刚才使用的基于接口的动态代理是JDK官方支持的，但是现在要讲的基于子类的动态代理需要第三方的支持，所以我们先导入maven的坐标，如下：

|  |
| --- |
| <dependencies>  <dependency>  <groupId>cglib</groupId>  <artifactId>cglib</artifactId>  <version>2.1\_3</version>  </dependency>  </dependencies> |

* 特点：

字节码随用随创建，随用随加载。

* 作用：

不修改源码的基础上对方法增强。

* 涉及的类：

Enhancer。

* 提供者：

第三方cglib库。

* 如何创建代理对象：

使用Enhancer类中的create方法。

* 创建代理对象的要求：

被代理类不能是最终类（也就是这个类不能用final修饰，因为最终类不能创建子类）。

* create方法的参数：
* class：字节码

它是用于指定被代理对象的字节码。

* callback：用于提供增强的代码

我们一般写的都是该接口的子接口实现类：MethodInterceptor（为方法拦截的意思）。

我们现在还是创建一个Producer类，只不过不要再实现接口了。

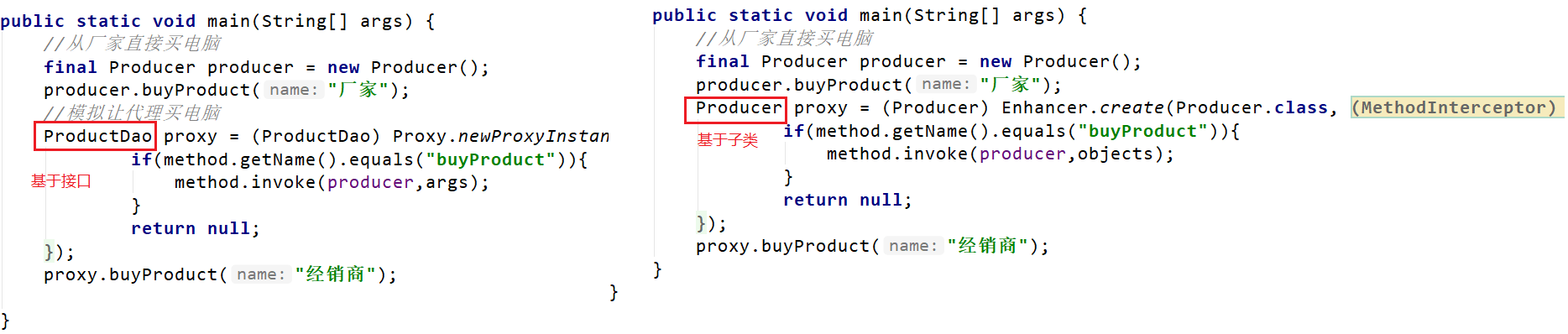
|  |
| --- |
| */\*\*  \* 模拟我们从厂家直接买电脑  \*/* **public class** Producer{   **public void** buyProduct(String name) {  System.***out***.println(name + **"售卖了电脑"**);  } } |

然后我们再模拟自己买电脑和让代理帮我们买电脑。

|  |
| --- |
| */\*\*  \* 消费者  \* 分别直接买  \* 让代理买  \*/* **public class** Consumer {  **public static void** main(String[] args) {  *//从厂家直接买电脑* **final** Producer producer = **new** Producer();  producer.buyProduct(**"厂家"**);   **Producer proxy = (Producer) Enhancer.*create*(Producer.class, new MethodInterceptor() {  */\*\*  \* @param o 当前代理对象的引用  \* @param method 代理对象需要执行的方法  \* @param objects 代理对象执行该方法需要用到的参数  \* @param methodProxy 当前执行方法的代理对象（不常用）  \* @return 返回值也是跟被代理对象执行方法的返回值一样  \* @throws Throwable  \*/* public Object intercept(Object o, Method method, Object[] objects, MethodProxy methodProxy) throws Throwable {  if(method.getName().equals("buyProduct")){  method.invoke(producer,objects);  }  return null;  }  });  proxy.buyProduct("经销商");**  } } |

运行结果跟之前一样，不再重复展示。

这里对比一下两种代理的返回值，如下：



那么我们再这里讲动态代理有什么用呢?

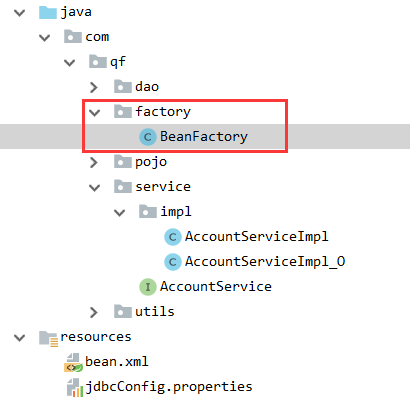
**事务控制也可以动态代理.**

## 1.4使用动态代理实现事务控制

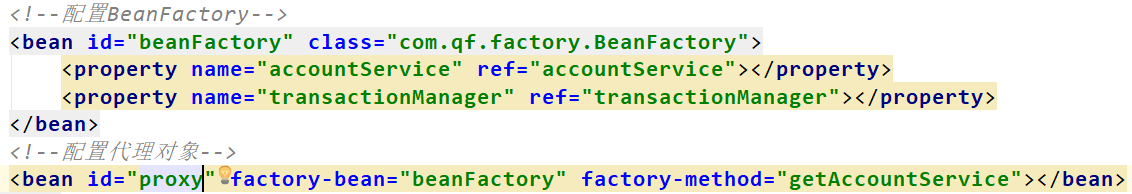
我们回到刚才银行转账的案例中，如下：

* 我们创建一个工厂，专门用来给Service创建代理对象，如下：

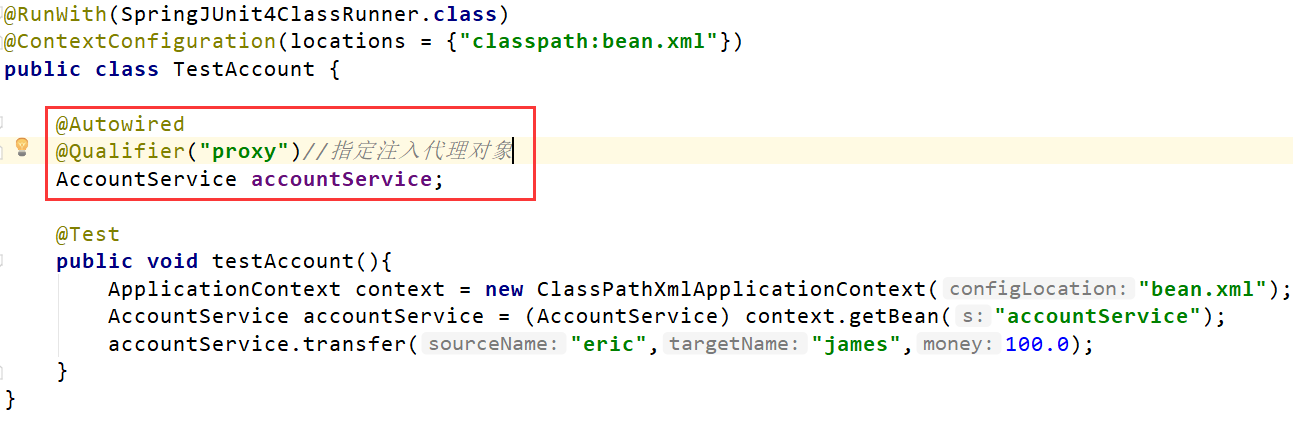
|  |
| --- |
| */\*\*  \* 用于创建service对象的代理工厂  \*/* **public class** BeanFactory {   **private** AccountService **accountService**;   **private** TransactionManager **transactionManager**;   **public void** setAccountService(AccountService accountService) {  **this**.**accountService** = accountService;  }   **public void** setTransactionManager(TransactionManager transactionManager) {  **this**.**transactionManager** = transactionManager;  }   */\*\*  \* 获取AccountService的代理对象  \** ***@return*** *\*/* **public** AccountService getAccountService(){  AccountService proxy = (AccountService)Proxy.*newProxyInstance*(AccountService.**class**.getClassLoader(), **accountService**.getClass().getInterfaces(), **new** InvocationHandler() {  **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {  **try** {  **if**(method.equals(**"transfer"**)){  *//开启事务* **transactionManager**.beginTransaction();  *//执行业务操作* method.invoke(**accountService**,args);  *//提交事务* **transactionManager**.commit();  }  } **catch** (Exception e) {  **transactionManager**.rollback();  e.printStackTrace();  **throw new** RuntimeException(e);  }**finally** {  **transactionManager**.release();  }  **return null**;  }  });  **return** proxy;  }  } |



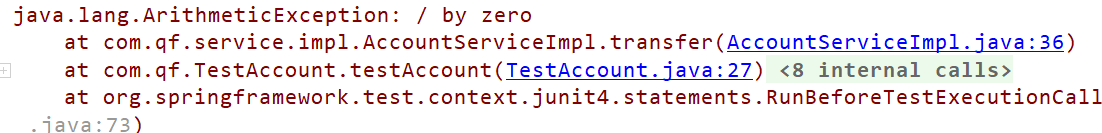
* 然后把我们所欠缺的依赖在bean.xml补全

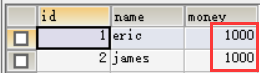


* 测试



* 运行结果

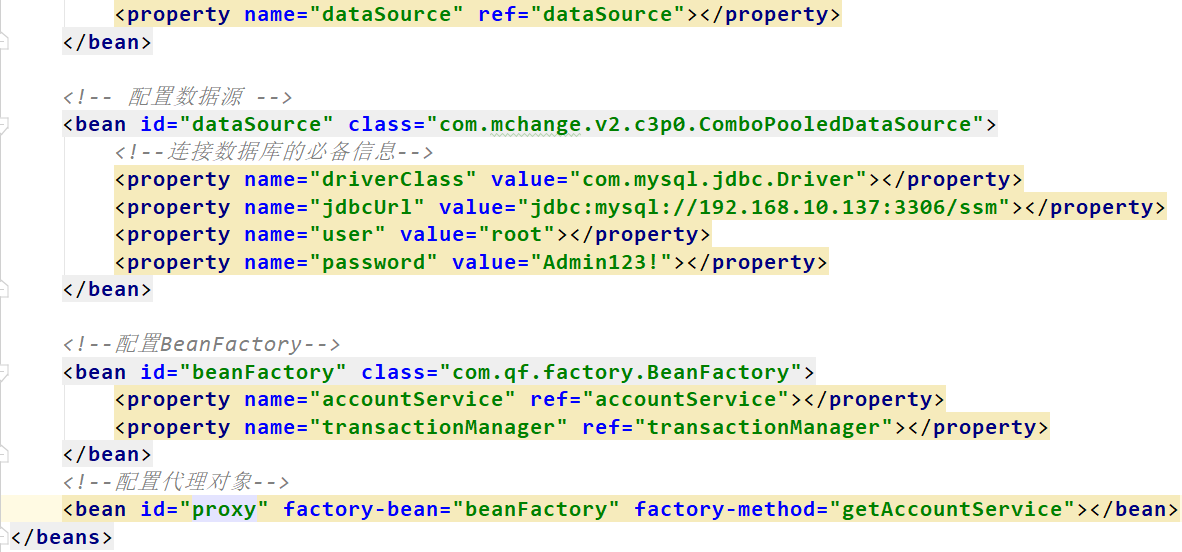




发现我们事务控制成功。

我们的业务层实现类代码变得简单了，但是bean.xml代码就很复杂了，如下：

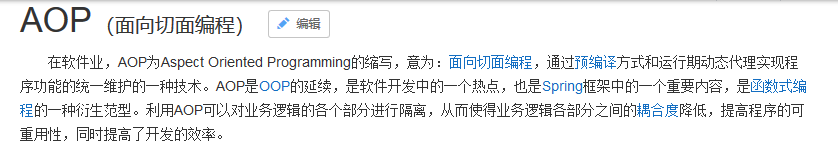




我们有没有更好的解决办法？我们可以使用AOP去代替动态代理实现事务控制。

## 1.5 AOP

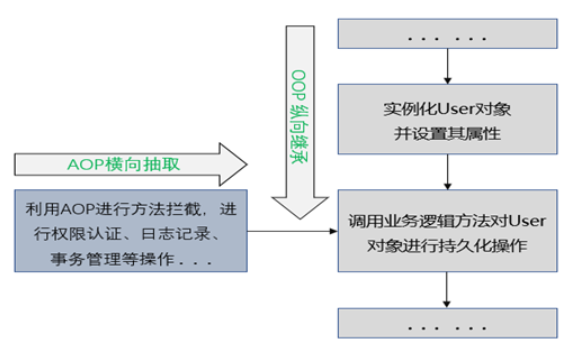
### 1.5.1 AOP的基本概念



AOP：面向切面编程，AOP是OOP的扩展和延伸，用来解决OOP开发中遇到的问题。

AOP利用的是一种横切技术，解剖开封装的对象内部，并将哪些影响多个类的公共行为封装到一个可重用模块，这就是所谓的Aspect方面/切面。所谓的切面，简单点所说，就是将哪些与业务无关，却为业务模块所共同调用的行为（方法）提取封装，减少系统的重复代码，以达到逻辑处理过程中各部分之间低耦合的隔离效果。

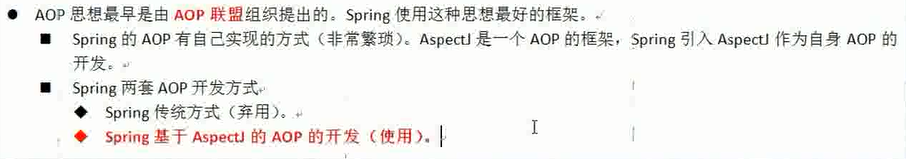
AOP采取横向抽取机制，取代了传统的纵向继承体系重复性代码。



### 1.5.2 springAop --基于AspectJ技术

#### 1.5.2.1 Aspect简介

SpringAop的底层采用的是动态代理技术，但是动态代理(jdk动态代理 cglib字节码代理)过于繁琐。于是spring引入了第三方的AspectJ框架。



#### 1.5.2.2 AOP的相关术语



### 1.5.3 SpringAop入门(AspectJ XML)

#### 1.5.3.1 引入aspectj的依赖

|  |
| --- |
| <**dependency**>  <**groupId**>org.aspectj</**groupId**>  <**artifactId**>aspectjweaver</**artifactId**>  <**version**>1.8.7</**version**> </**dependency**> |

#### 1.5.3.2 编写目标类

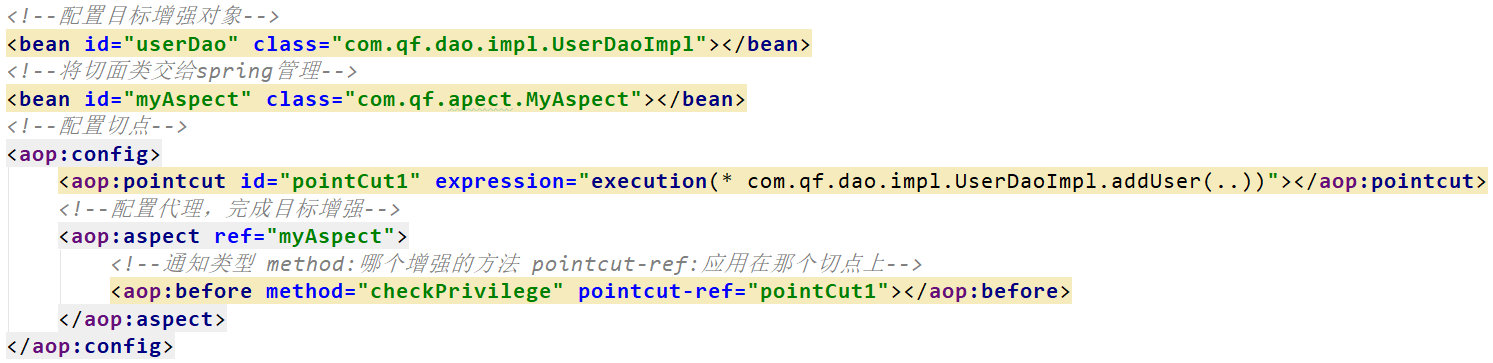
|  |
| --- |
| **public interface** UserDao {  **public void** findAll();  **public void** addUser();  **public void** updateUser();  **public void** deleteUser(); } |

|  |
| --- |
| **public class** UserDaoImpl **implements** UserDao {  *//查询的方法* **public void** findAll() {  System.***out***.println(**"findAll方法执行了....."**);  }   *//新增的方法* **public void** addUser() {  System.***out***.println(**"add方法执行了....."**);  }   *//修改的方法* **public void** updateUser() {  System.***out***.println(**"update方法执行了....."**);  }   *//删除的方法* **public void** deleteUser() {  System.***out***.println(**"delete方法执行了....."**);  } } |

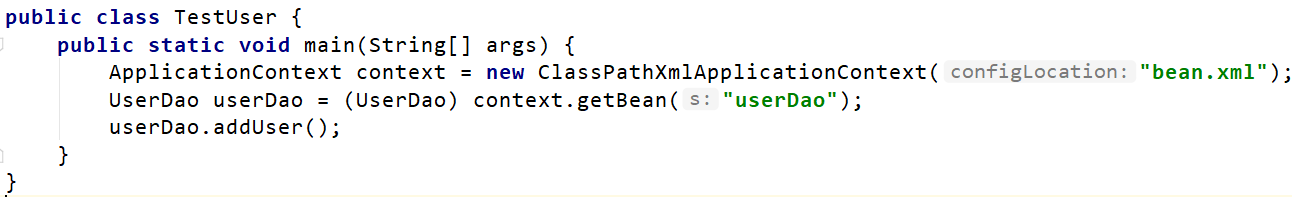
#### 1.5.3.3 编写切面类

|  |
| --- |
| **public class** MyAspect {  *//前置通知，开启权限校验* **public void** checkPrivilege(){  System.***out***.println(**"开启权限校验...."**);  }   *//后置通知* **public void** printLog(Object result){  System.***out***.println(**"后置通知:打印日志记录"** + result);  }   *//环绕通知* **public void** around(ProceedingJoinPoint joinPoint){  System.***out***.println(**"开启环绕通知前"**);  **try** {  joinPoint.proceed();  } **catch** (Throwable throwable) {  throwable.printStackTrace();  }  System.***out***.println(**"开启环绕通知后"**);  }   *//抛出异常通知* **public void** afterThrowing(Throwable tx){  System.***out***.println(**"抛出异常通知"** + tx.getMessage());  }   *//最终通知* **public void** after(){  System.***out***.println(**"执行最终通知"**);  }  } |

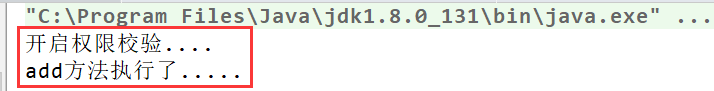
#### 1.5.3.4 配置切面



#### 1.5.3.5 测试



最后效果如下：



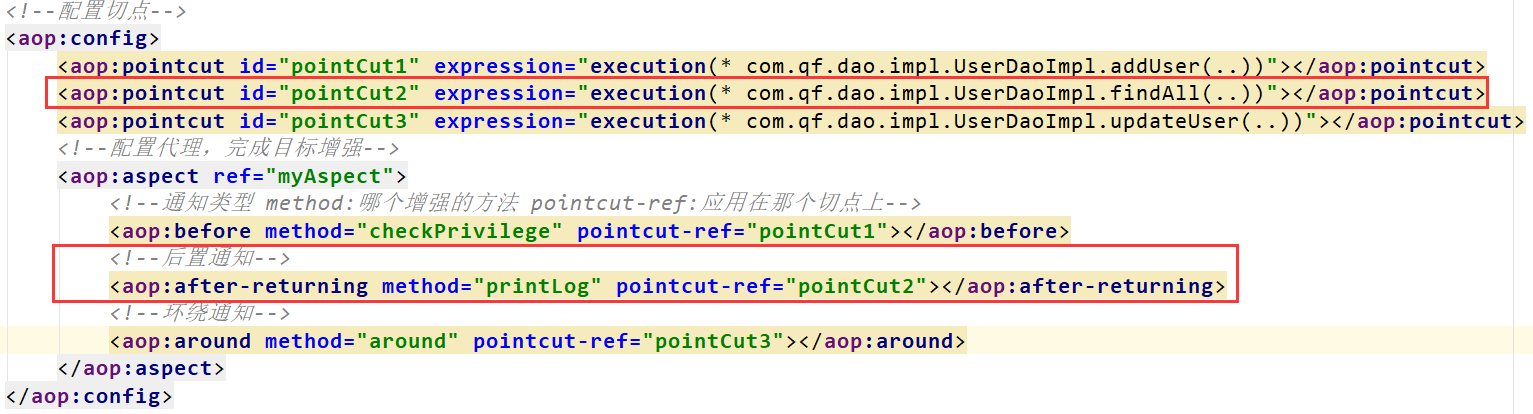
### 1.5.4 通知类型

#### 1.5.4.1 前置通知

在目标方法执行之前进行增强，刚刚做了演示，这里不再重复。

#### 1.5.4.2 后置通知

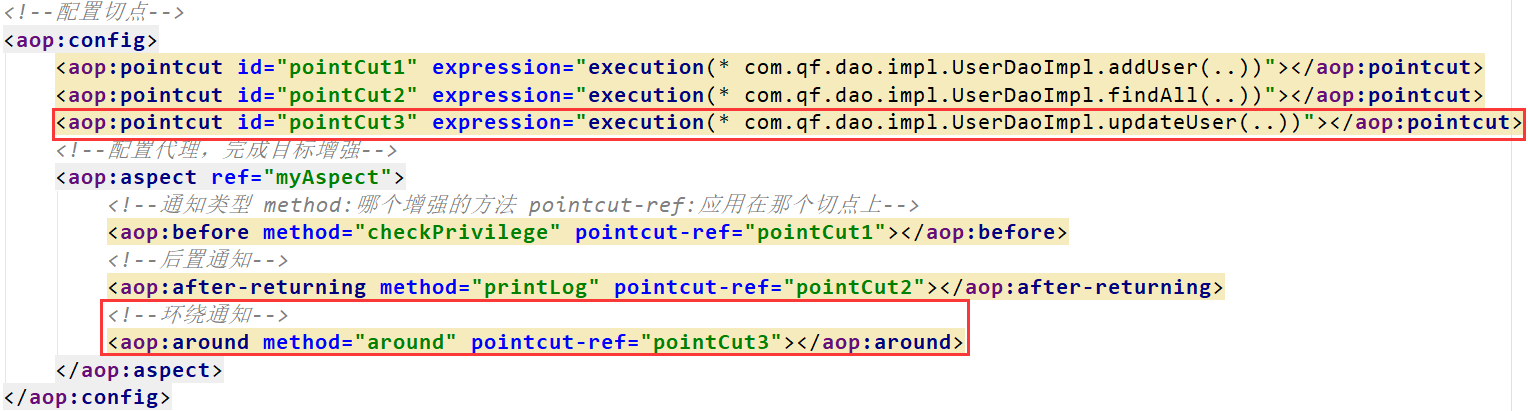
在目标方法执行之后进行的操作。

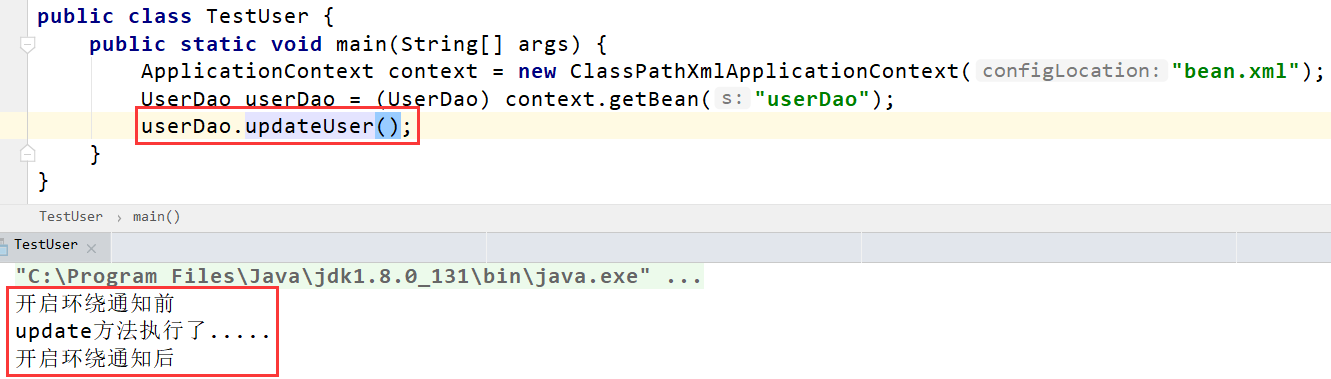




#### 1.5.4.3 环绕通知

在目标方法执行前和执行后都要执行

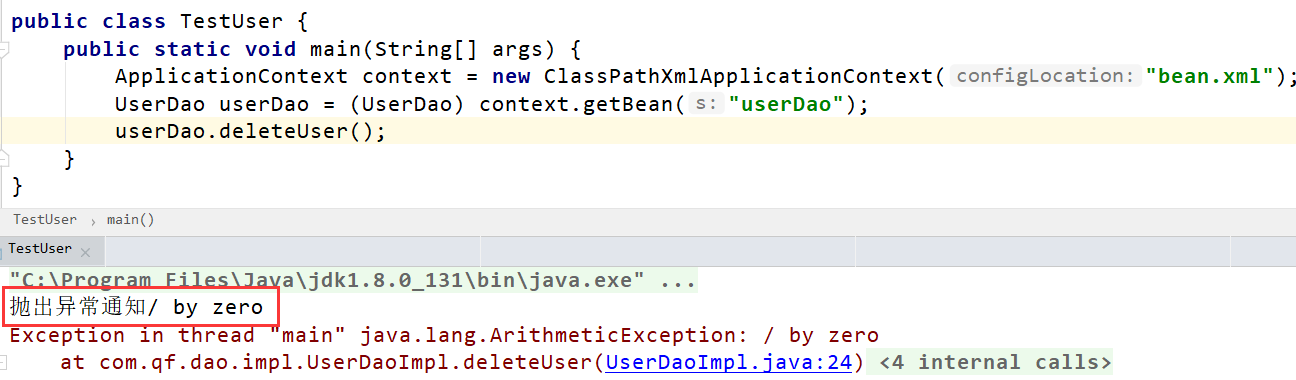




#### 1.5.4.4抛出异常通知

在目标方法执行出现异常的时候，进行增强





#### 1.5.4.5最终通知

无论代码是否有异常，总会执行(了解)。



### 1.5.5 SpringAop入门(AspectJ 注解开发)

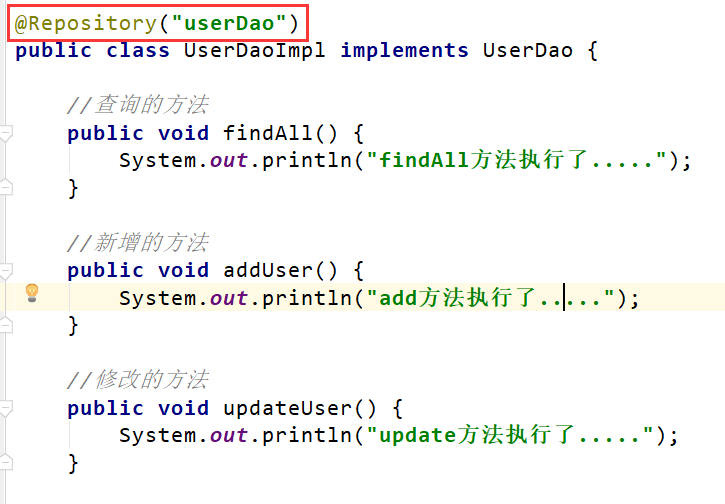
* 在配置文件引入spring对aop注解的支持

将我们上面XML开发的所有配置全部注释掉，然后引入下面的配置文件

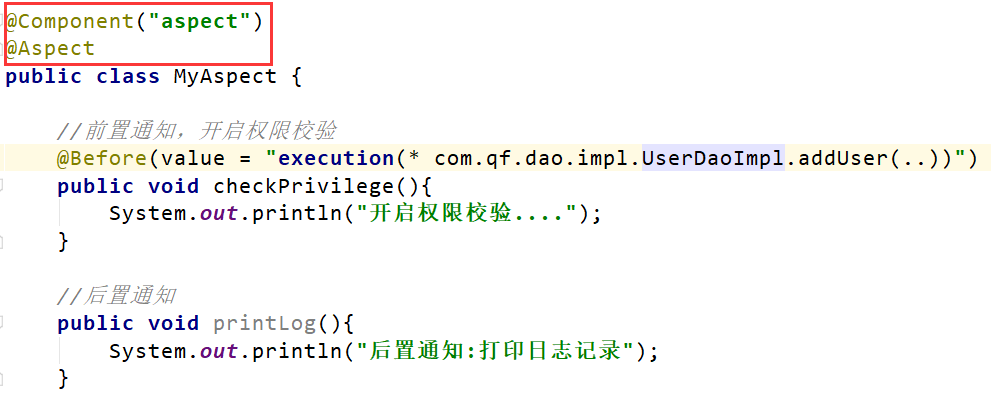
|  |
| --- |
| *<!--开启包扫描-->* <**context:component-scan base-package="com.qf"**></**context:component-scan**> *<!--开启spring对aop注解的支持-->* <**aop:aspectj-autoproxy**></**aop:aspectj-autoproxy**> |

* 编写目标类 UserDaoImpl

跟上面XML开发的一样，不再重复



* 编写切面类



* 测试

