**Spring第四天**

* Spring中的JdbcTemplate
* Spring基于AOP的事务控制
* Spring基于XML的事务控制
* Spring基于注解的事务控制

## 1.1 JdbcTemplate的概述和环境搭建

### 1.1.1 JdbcTemplate的基本概述

它是spring框架中提供的一个对象。是对原生jdbc的简单封装，Spring框架为我们提供了很多操作的模板类。

操作关系型数据库的：

JdbcTemplate

HibernateTemplate

操作nosql数据库的:

RedisTemplate

操作消息队列的：

JmsTemplate

今天我们的主角在spring-jdbc-5.0.2.RELEASE.jar中。除了要导入这个jar包之外，我们还需要导入spring-tx-5.0.2.RELEASE.jar(它是和事务相关的)。

JdbcTemplate的基本作用：它是用于和数据库交互的，实现数据表的crud操作。

问题：如何创建JdbcTemplate对象?

JdbcTemplate又有哪些方法?

接下来，我们来搭建JdbcTemplate环境

### 1.1.2 搭建JdbcTemplate环境

* 创建maven工程，并引入依赖

|  |
| --- |
| <**dependencies**>  <**dependency**>  <**groupId**>org.springframework</**groupId**>  <**artifactId**>spring-context</**artifactId**>  <**version**>5.0.2.RELEASE</**version**>  </**dependency**>   <**dependency**>  <**groupId**>org.springframework</**groupId**>  <**artifactId**>spring-jdbc</**artifactId**>  <**version**>5.0.2.RELEASE</**version**>  </**dependency**>   <**dependency**>  <**groupId**>org.springframework</**groupId**>  <**artifactId**>spring-tx</**artifactId**>  <**version**>5.0.2.RELEASE</**version**>  </**dependency**>   <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  <**version**>5.1.6</**version**>  </**dependency**> </**dependencies**> |

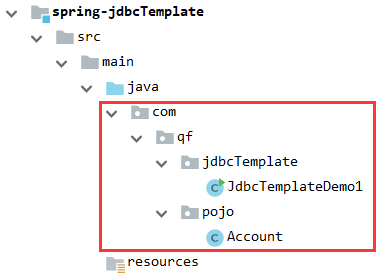
* 创建Account实体类

|  |
| --- |
| **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** +  **'}'**;  } } |

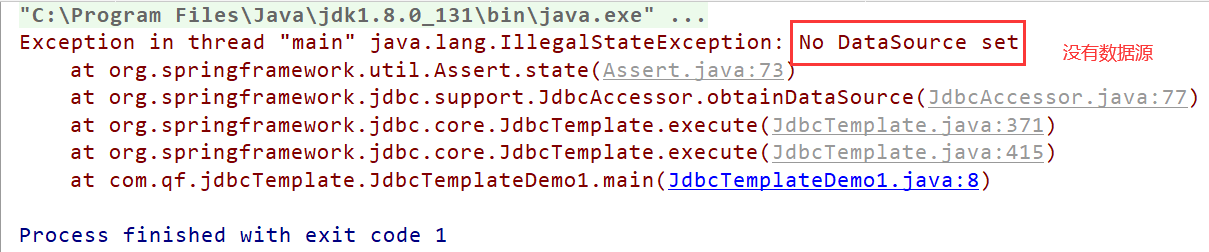
* 创建JdbcTemplateDemo1进行测试

|  |
| --- |
| **public class** JdbcTemplateDemo1 {  **public static void** main(String[] args) {  JdbcTemplate jt = **new** JdbcTemplate();  jt.execute(**"insert into account(name,money) values('Oneal',100.0)"**);  } } |

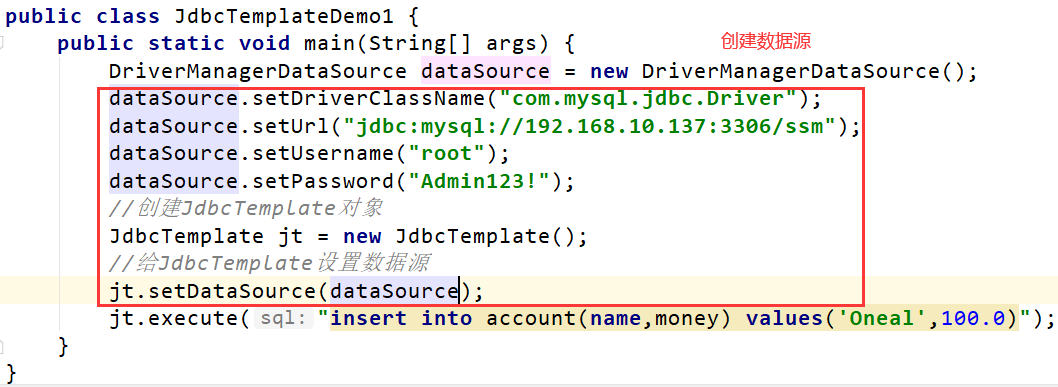
工程创建结构目录如下:

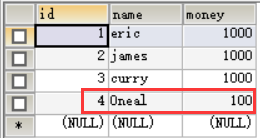


* 测试运行



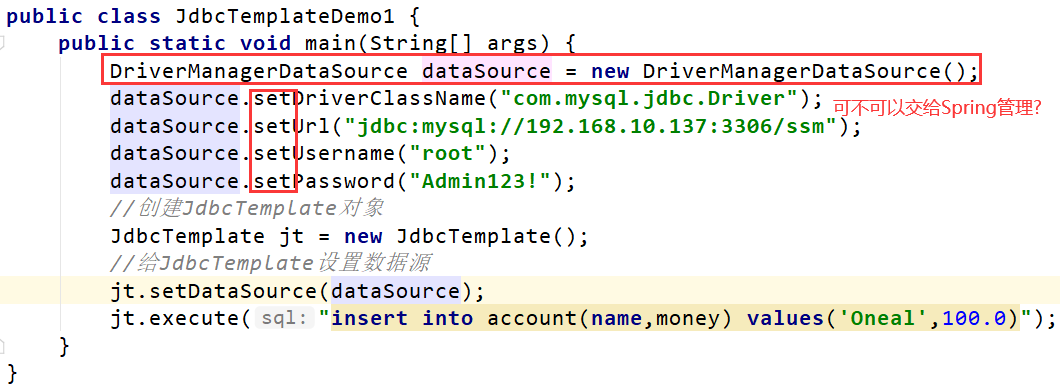
对于数据源（连接池）的选择有很多：C3P0、DPCP、Druid等等，但是这次我们使用spring集成的数据源。



最后测试效果如下:  


此时数据添加成功。

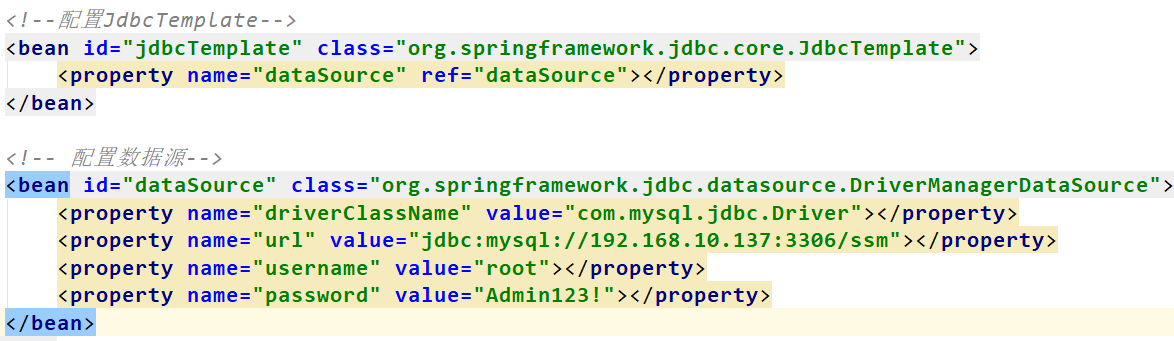
**但是我们发现了问题，如下：**

****

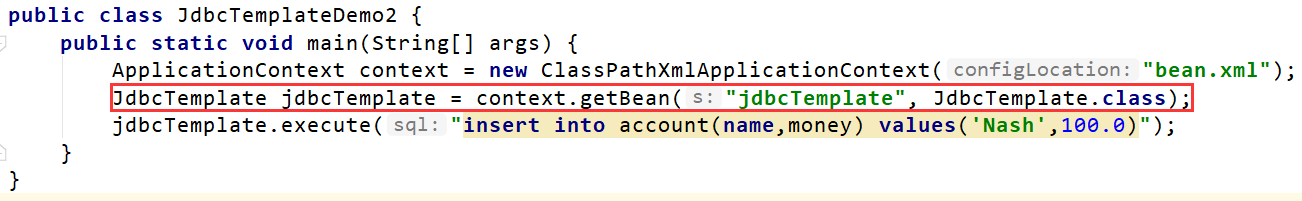
看到new和set就应该想到：可以用spring的ioc来配置。

### 1.1.3 JdbcTemplate在spring的ioc中使用

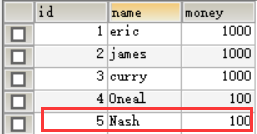
* 创建一个bean.xml。



* 定义一个测试类进行测试

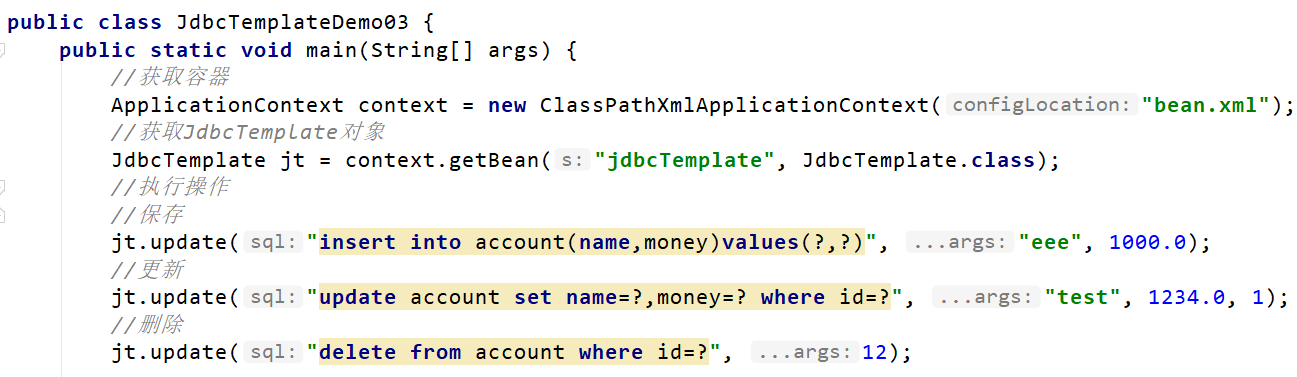


观察数据库效果:



### 1.1.4 JdbcTemplate的CRUD操作

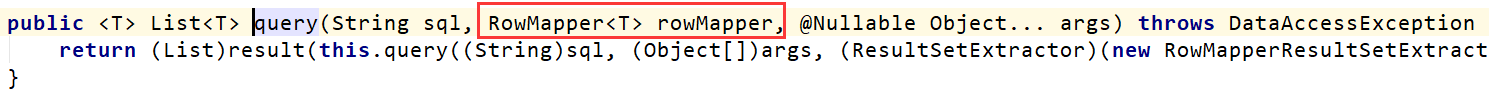
我们再创建一个Java类: JdbcTemplateDemo03.



查询操作比较复杂，下面分析查询操作，如下：

(1)查询所有

查看源码，我们发现查询query的方法重载了很多，那么我们应该选择哪一个呢?



这个RowMapper是一个接口。类似于DbUtils里面的BeanListhandler的功能。我们应该如何使用?

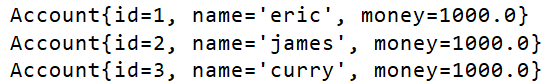
我们可以定义一个类，实现这个接口即可。

|  |
| --- |
| **public class** AccountRowMapper **implements** RowMapper<Account> {  */\*\*  \* 把结果集中的数据封装到Account中。再由spring把每个Account放入到集合中  \** ***@param rs*** *\** ***@param i*** *\** ***@return*** *\** ***@throws*** *SQLException  \*/* **public** Account mapRow(ResultSet rs, **int** i) **throws** SQLException {  Account account = **new** Account();  account.setId(rs.getInt(**"id"**));  account.setName(rs.getString(**"name"**));  account.setMoney(rs.getDouble(**"money"**));  **return** account;  } } |

所在在查询的时候，我们把这个类的对象传入到方法里面即可。

|  |
| --- |
| List<Account> accounts = jt.query(**"select** *\** **from account where money > ?"**, **new AccountRowMapper()**, 100.0); **for** (Account account : accounts) {  System.***out***.println(account); } |

查询结果如下：



(2)单个查询

我们使用跟查询所有同样的方法即可，做一些小改动，如下：

|  |
| --- |
| *//查询一个* List<Account> accounts = jt.query(**"select** *\** **from account where name = ?"**, **new** AccountRowMapper(), **"james"**); System.***out***.println(accounts.get(0)); |

(3)聚合函数查询

|  |
| --- |
| Long count = jt.queryForObject(**"select** *count***(***\****) from account where money > ?"**, Long.**class**, 100.0); System.***out***.println(count); |

### 1.1.5 JdbcTemplate在Dao中的使用

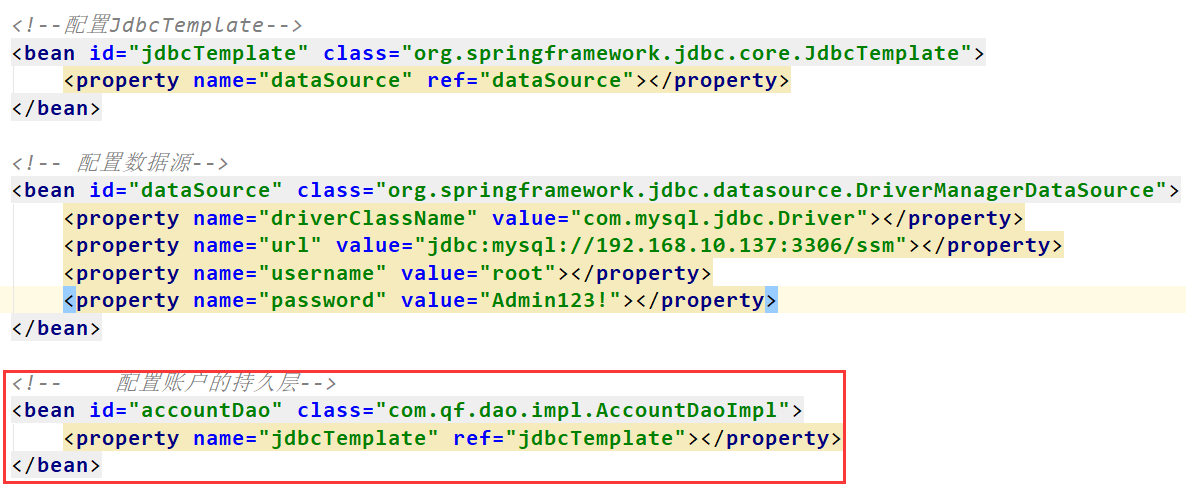
* 定义接口

|  |
| --- |
| **public interface** AccountDao {  */\*\*  \* 根据Id查询账户  \*  \** ***@param accountId*** *\** ***@return*** *\*/* Account findAccountById(Integer accountId);   */\*\*  \* 根据名称查询账户  \*  \** ***@param accountName*** *\** ***@return*** *\*/* Account findAccountByName(String accountName);   */\*\*  \* 更新账户  \*  \** ***@param account*** *\*/* **void** updateAccount(Account account);  } |

* 编写实现类

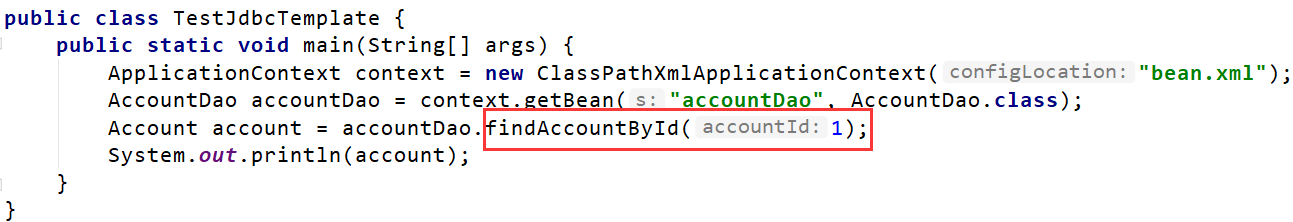
|  |
| --- |
| **public class** AccountDaoImpl **implements** AccountDao {   **private** JdbcTemplate **jdbcTemplate**;   **public void** setJdbcTemplate(JdbcTemplate jdbcTemplate) {  **this**.**jdbcTemplate** = jdbcTemplate;  }   **public** Account findAccountById(Integer accountId) {  List<Account> accounts = **jdbcTemplate**.query(**"select** *\** **from account where id=?"**, **new** AccountRowMapper(), accountId);  **return** accounts.get(0);  }   **public** Account findAccountByName(String accountName) {  List<Account> accounts = **jdbcTemplate**.query(**"select** *\** **from account where name=?"**, **new** AccountRowMapper(), accountName);  **return** accounts.get(0);  }   **public void** updateAccount(Account account) {  **jdbcTemplate**.update(**"update account set name=?,money=? where id=?"**, account.getName(), account.getMoney(), account.getId());  } } |

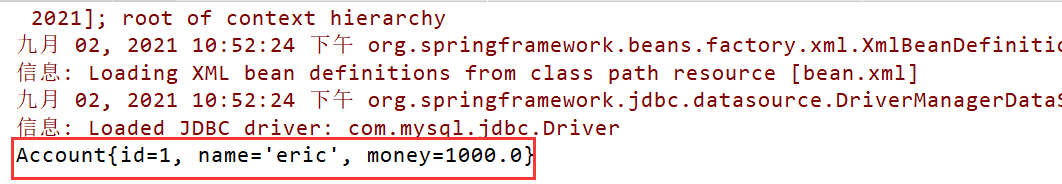
* 编写配置文件



* 测试

这里只测试一个接口。

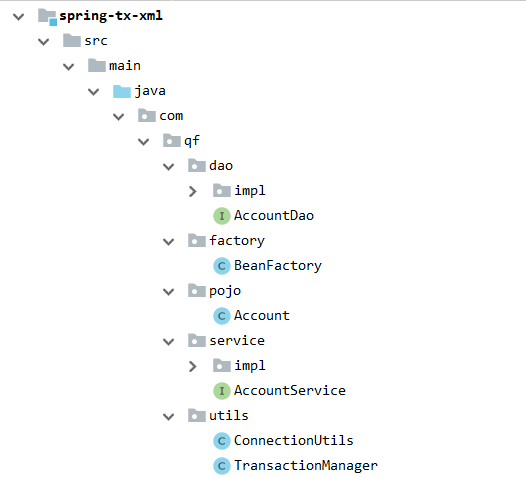




## 1.2 基于XML的AOP实现事务控制

* 创建工程

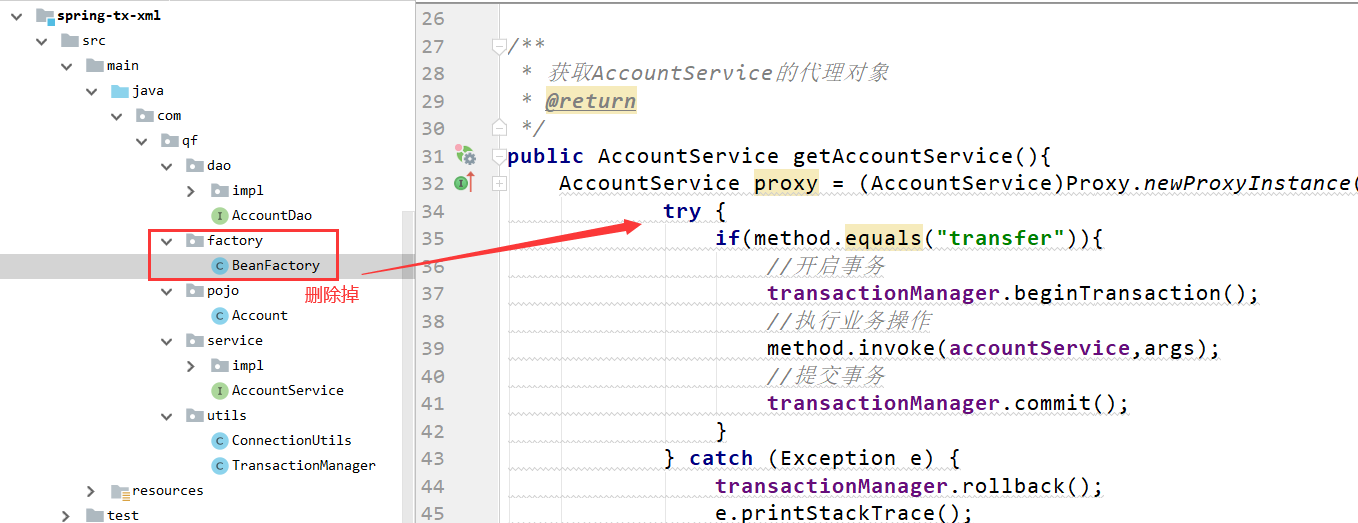
创建一个普通的maven工程，并且拷贝之前银行转账案例的代码，如下：



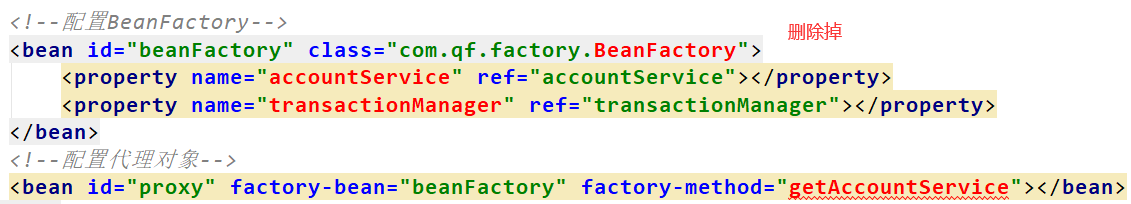
并且在pom.xml里导入AOP相关的坐标（Aspectj用于解析切入点表达式），如下：

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

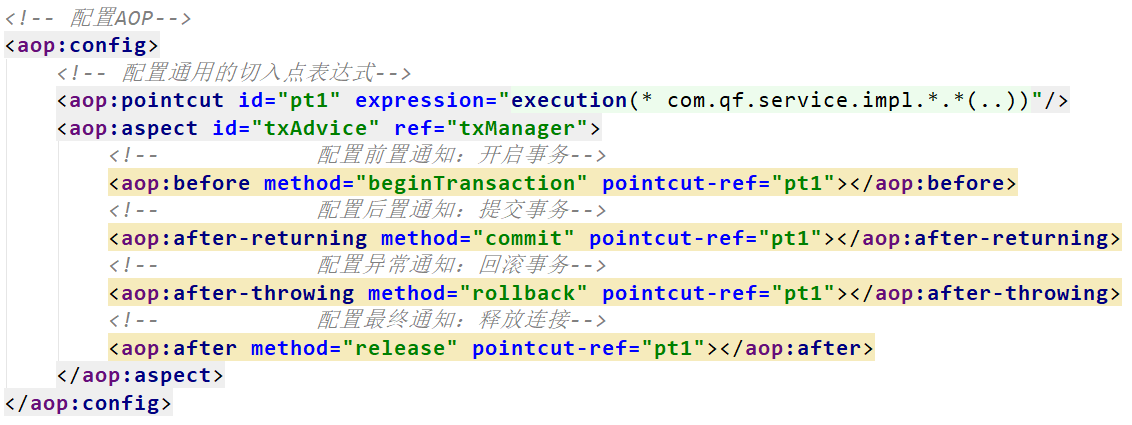
* 同时把我们自己写的跟动态代理相关的代码删掉：



同时在bean.xml文件中删掉相关的bean标签配置，如下：



* 在bean.xml定义AOP相关的配置



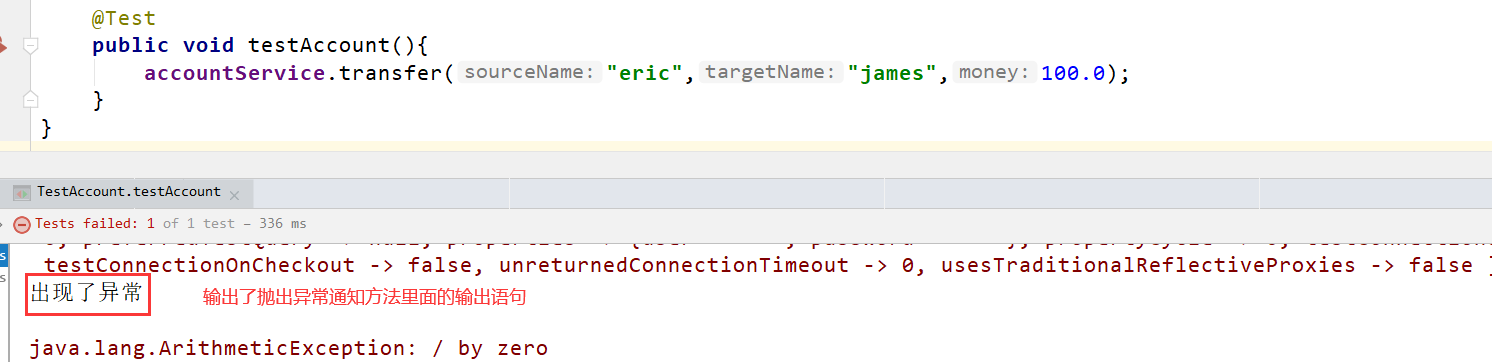
最后整个bean.xml的配置信息如下:

|  |
| --- |
| <**bean id="accountDao" class="com.qf.dao.impl.AccountDaoImpl"**>  <**property name="connectionUtils" ref="connectionUtils"**></**property**>  <**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**>  <**bean id="txManager" class="com.qf.utils.TransactionManager"**>  <**property name="connectionUtils" ref="connectionUtils"**></**property**> </**bean**>  <**bean id="connectionUtils" class="com.qf.utils.ConnectionUtils"**>  <**property name="dataSource" ref="dataSource"**></**property**> </**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**>  *<!-- 配置AOP-->* <**aop:config**>  *<!-- 配置通用的切入点表达式-->* <**aop:pointcut id="pt1" expression="execution(\* com.qf.service.impl.\*.\*(..))"**/>  <**aop:aspect id="txAdvice" ref="txManager"**>  *<!-- 配置前置通知：开启事务-->* <**aop:before method="beginTransaction" pointcut-ref="pt1"**></**aop:before**>  *<!-- 配置后置通知：提交事务-->* <**aop:after-returning method="commit" pointcut-ref="pt1"**></**aop:after-returning**>  *<!-- 配置异常通知：回滚事务-->* <**aop:after-throwing method="rollback" pointcut-ref="pt1"**></**aop:after-throwing**>  *<!-- 配置最终通知：释放连接-->* <**aop:after method="release" pointcut-ref="pt1"**></**aop:after**>  </**aop:aspect**> </**aop:config**> |

* 业务类还原到最开始的状态

|  |
| --- |
| **public class** AccountServiceImpl **implements** AccountService {  **private** AccountDao **accountDao**;   **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);  } } |

* 测试



我们发现数据库里面的数据也没有发生变化，说明事务得到了控制。

## 1.3 Spring的事务管理的API

前面我们进行了事务的控制，我们发现，我们需要自己去编写事务管理的代码。这是非常繁琐的。其实spring也给我们提供了事务管理的API。

(1) PlatformTransactionManager：平台事务管理器

* 平台事务管理器：接口，是Spring用于管理事务的真正的对象。
  + **DataSourceTransactionManager ：底层使用JDBC管理事务。**
  + HibernateTransactionManager ：底层使用Hibernate管理事务。

(2)TransactionDefinition ：事务定义信息

事务定义：用于定义事务的相关的信息，隔离级别、超时信息、**传播行为**、是否只读。

* 获取事务对象名称：

String getName()

* 获取事务隔离级别：

int getIsolationLevel() spring提供的事务隔离级别默认跟数据库的隔离级别一致。

ISOLATION\_DEFAULT

默认级别，归属下列的某一种（spring的默认级别是数据库的默认级别）。

ISOLATION\_READ\_UNCOMMITTED

可以读取未提交数据。

ISOLATION\_READ\_COMMITTED

只能读取已提交数据，解决脏读问题（Oracle的默认级别）。

ISOLATION\_REPEATABLE\_READ

是否读取其它事务提交修改后的数据，解决不可重复读问题（MySQL的默认级别）。

ISOLATION\_SERIALIZABLE

是否读取其他事务提交添加后的数据，解决幻读问题。

* 获取事务传播行为：

int getPropagationBehavior()

传播行为是指：什么情况下必须要开启事务（增删改），什么情况下可以不开启事务（查）。

REQUIRED

如果当前没有事务，就新建一个事务，如果已经存在一个事务，就加入到该事务中（默认值）。

SUPPORTS

支持当前事务，如果当前没有事务，就以非事务方式执行（没有事务）。

MANDATORY

使用当前事务，如果当前没有事务，就抛出异常。

REQUERS\_NEW

新建事务，如果当前在事务中，把当前事务挂起。

NOT\_SUPPORTED

以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。

NEVER

以非事务方式运行，如果当前存在事务，抛出异常。

NESTED

如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行类似REQUIRED的操作。

* 获取事务超时时间：

int getTimeout()

当提交或者回滚达到多长时间就过期。默认值是-1，没有超时限制。如果有，以秒为单位进行设置。

* 获取事务是否只读：

boolean isReadOnly()

读写型事务：增加、删除和修改会开启事务。

只读型事务：执行查询时，也会开启事务。

**建议查询时设置为只读。**

* 事务隔离级别反映事务提交并发访问时的处理态度

### 1.3.1 配置编程式事务

#### 1.3.1.1 搭建案例环境

我们现在使用JdbcTemplate进行数据持久层操作

* 引入pom依赖

|  |
| --- |
| <**dependencies**>  <**dependency**>  <**groupId**>org.springframework</**groupId**>  <**artifactId**>spring-context</**artifactId**>  <**version**>5.0.2.RELEASE</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.springframework</**groupId**>  <**artifactId**>spring-jdbc</**artifactId**>  <**version**>5.0.2.RELEASE</**version**>  </**dependency**>  <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  <**version**>5.1.6</**version**>  </**dependency**>  <**dependency**>  <**groupId**>c3p0</**groupId**>  <**artifactId**>c3p0</**artifactId**>  <**version**>0.9.1.2</**version**>  </**dependency**>  <**dependency**>  <**groupId**>junit</**groupId**>  <**artifactId**>junit</**artifactId**>  <**version**>4.12</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.springframework</**groupId**>  <**artifactId**>spring-test</**artifactId**>  <**version**>5.0.2.RELEASE</**version**>  </**dependency**>   **<dependency>  <groupId>org.aspectj</groupId>  <artifactId>aspectjweaver</artifactId>  <version>1.8.7</version>  </dependency>** </**dependencies**> |

* dao

|  |
| --- |
| **public class** AccountDaoImpl **implements** AccountDao {   **private** JdbcTemplate **jdbcTemplate**;   **public void** setJdbcTemplate(JdbcTemplate jdbcTemplate) {  **this**.**jdbcTemplate** = jdbcTemplate;  }   */\*\*  \* 根据id查找账户信息  \** ***@param name*** *\** ***@return*** *\*/* **public** Account findAccountByName(String name) {  List<Account> accounts = **jdbcTemplate**.query(**"select** *\** **from account where name = ?"**, **new** AccountRowMapper(), name);  **return** accounts.get(0);  }   */\*\*  \* 修改账户信息  \** ***@param account*** *\*/* **public void** updateAccount(Account account) {  String sql = **"update account set money=? where name=?"**;  **try** {  **jdbcTemplate**.update(sql,account.getMoney(),account.getName());  } **catch** (Exception e) {  e.printStackTrace();  }  } } |

* rowMapper

|  |
| --- |
| **public class** AccountRowMapper **implements** RowMapper<Account> {   **public** Account mapRow(ResultSet resultSet, **int** i) **throws** SQLException {  Account account = **new** Account();  account.setId(resultSet.getInt(**"id"**));  account.setName(resultSet.getString(**"name"**));  account.setMoney(resultSet.getDouble(**"money"**));  **return** account;  } } |

* service

|  |
| --- |
| **public class** AccountServiceImpl **implements** AccountService {   **private** AccountDao **accountDao**;   **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);  } } |

* 配置文件

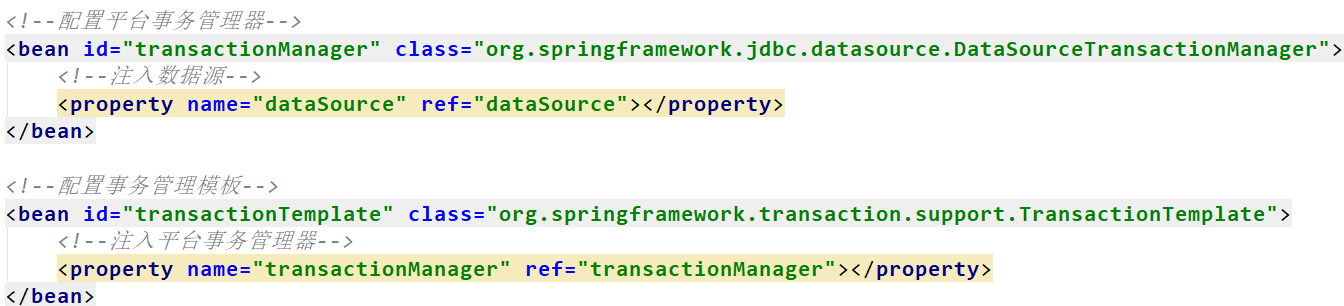
|  |
| --- |
| <**bean id="accountDao" class="com.qf.dao.impl.AccountDaoImpl"**>  <**property name="jdbcTemplate" ref="jdbcTemplate"**></**property**> </**bean**>  <**bean id="accountService" class="com.qf.service.impl.AccountServiceImpl"**>  <**property name="accountDao" ref="accountDao"**></**property**> </**bean**>  <**bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate"**>  <**property name="dataSource" ref="dataSource"**></**property**> </**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**> |

#### 1.3.1.2 配置平台事务管理器和事务管理模板

|  |
| --- |
| *<!--配置平台事务管理器-->* <**bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager"**>  *<!--注入数据源-->* <**property name="dataSource" ref="dataSource"**></**property**> </**bean**> |

|  |
| --- |
| *<!--配置事务管理模板-->* <**bean id="transactionTemplate" class="org.springframework.transaction.support.TransactionTemplate"**>  *<!--注入平台事务管理器-->* <**property name="transactionManager" ref="transactionManager"**></**property**> </**bean**> |

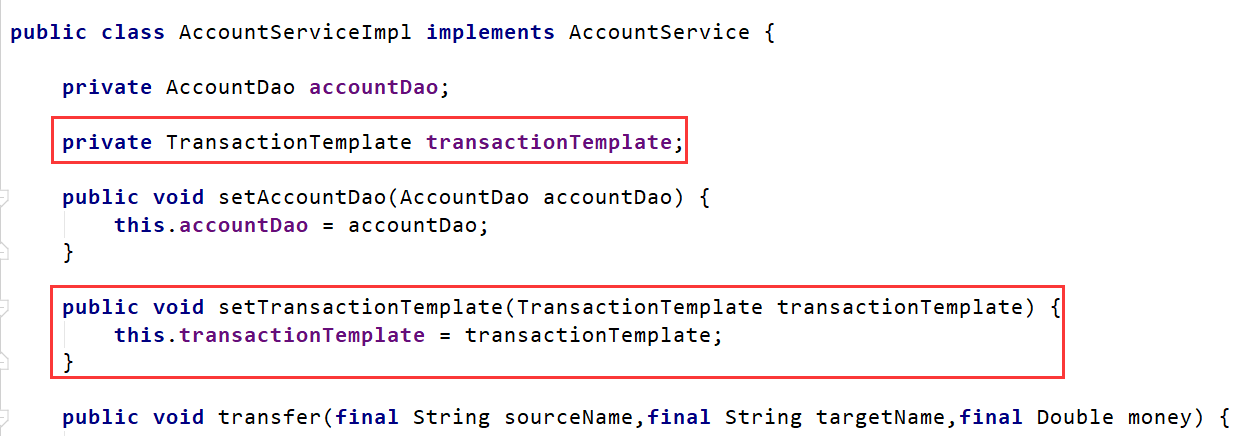
在配置文件中如下:



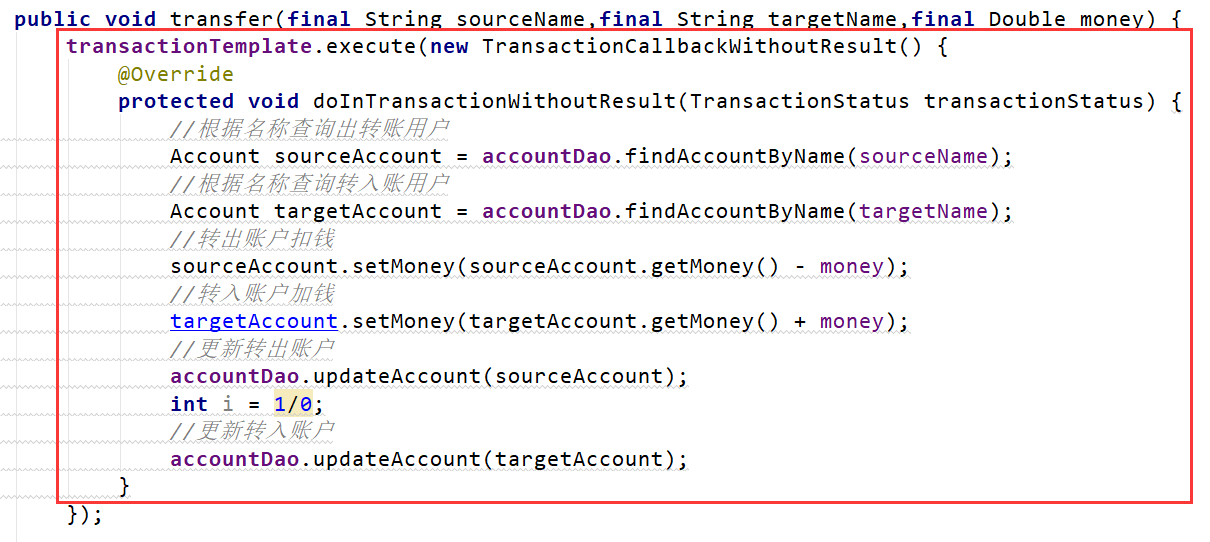
#### 1.3.1.3 在业务类中注入事务管理模板

因为事务管理模板才是真正干活对象，真正进行事务的控制。

|  |
| --- |
| <**bean id="accountService" class="com.qf.service.impl.AccountServiceImpl"**>  <**property name="accountDao" ref="accountDao"**></**property**>  <**property name="transactionTemplate" ref="transactionTemplate"**></**property**> </**bean**> |

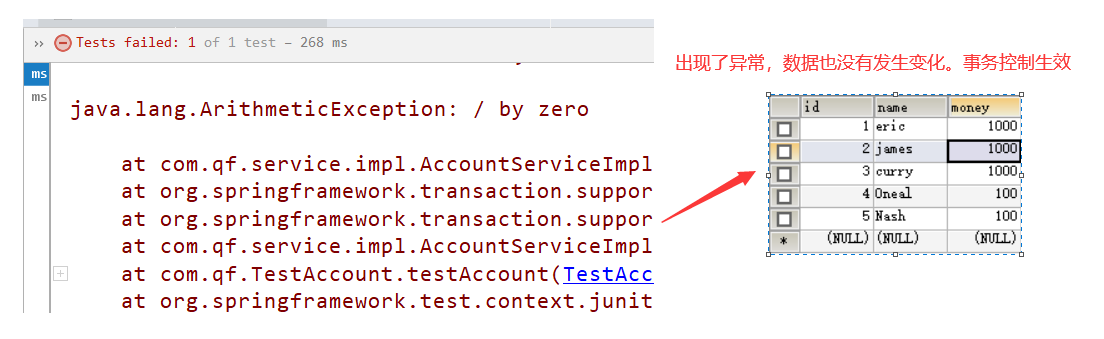


#### 1.3.1.4 在业务类里面进行事务控制

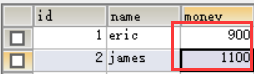


#### 1.3.1.5 测试效果

* 模拟异常情况 int i = 1/0;



* 正常情况 去掉int i = 1/0;



### 1.3.2 配置声明式事务(XML方式—重点)

spring的编程式事务控制是不常用的，了解即可。

* 去掉事务管理模板之后bean.xml如下

|  |
| --- |
| <**bean id="accountDao" class="com.qf.dao.impl.AccountDaoImpl"**>  <**property name="jdbcTemplate" ref="jdbcTemplate"**></**property**> </**bean**>  <**bean id="accountService" class="com.qf.service.impl.AccountServiceImpl"**>  <**property name="accountDao" ref="accountDao"**></**property**> </**bean**>  <**bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate"**>  <**property name="dataSource" ref="dataSource"**></**property**> </**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**>  *<!--配置平台事务管理器-->* <**bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager"**>  *<!--注入数据源-->* <**property name="dataSource" ref="dataSource"**></**property**> </**bean**> |

* 把业务类还原

|  |
| --- |
| **public class** AccountServiceImpl **implements** AccountService {   **private** AccountDao **accountDao**;   **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);  } } |

* 配置事务增强

|  |
| --- |
| *<!--配置事务增强-->* <**tx:advice id="txAdvice" transaction-manager="transactionManager"**>  *<!--配置事务的属性-->* <**tx:attributes**>  *<!--name:给哪个业务方法配置事务 propagation：事务的传播行为 REQUIRED默认值-->* <**tx:method name="transfer" propagation="REQUIRED"**/>  </**tx:attributes**> </**tx:advice**> |

* 配置aop，将事务增强和切点连接在一起

|  |
| --- |
| <**aop:config**>  <**aop:pointcut expression="execution(\* com.qf.service.impl.\*.\*(..))" id="p1"**></**aop:pointcut**>  <**aop:advisor advice-ref="txAdvice" pointcut-ref="p1"**></**aop:advisor**> </**aop:config**> |

完成之后，整个配置文件信息如下:

|  |
| --- |
| <**bean id="accountDao" class="com.qf.dao.impl.AccountDaoImpl"**>  <**property name="jdbcTemplate" ref="jdbcTemplate"**></**property**> </**bean**>  <**bean id="accountService" class="com.qf.service.impl.AccountServiceImpl"**>  <**property name="accountDao" ref="accountDao"**></**property**> </**bean**>  <**bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate"**>  <**property name="dataSource" ref="dataSource"**></**property**> </**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**>  *<!--配置平台事务管理器-->* <**bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager"**>  *<!--注入数据源-->* <**property name="dataSource" ref="dataSource"**></**property**> </**bean**>  ***<!--配置事务增强-->* <tx:advice id="txAdvice" transaction-manager="transactionManager">  *<!--配置事务的属性-->* <tx:attributes>  *<!--name:给哪个业务方法配置事务 propagation：事务的传播行为 REQUIRED默认值-->* <tx:method name="transfer" propagation="REQUIRED"/>  </tx:attributes> </tx:advice>  <aop:config>  <aop:pointcut expression="execution(\* com.qf.service.impl.\*.\*(..))" id="p1"></aop:pointcut>  <aop:advisor advice-ref="txAdvice" pointcut-ref="p1"></aop:advisor> </aop:config>** |

* 测试

效果略

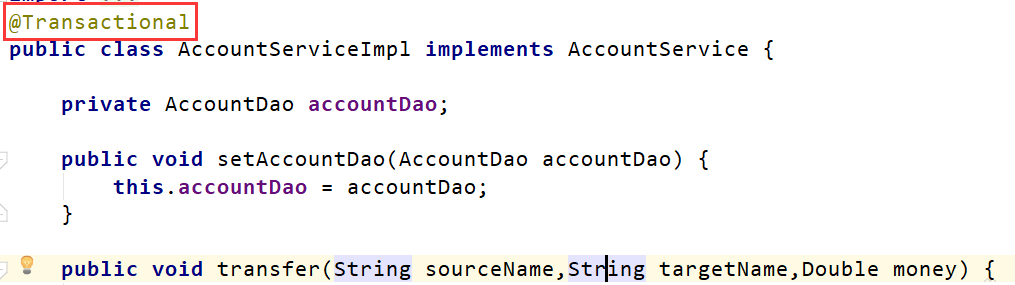
### 1.3.3 配置声明式事务(注解的方式—重点)

* 配置开启注解对事务的支持

|  |
| --- |
| <**tx:annotation-driven**></**tx:annotation-driven**> |



* 在业务类上添加注解



* 测试效果(略)