## 1.什么是事务？

事务是逻辑上的一组操作，这组操作要么全部成功，要么全部失败

## 2.事物具有四大特性ACID

说到事务，就不得不说其4打特性，主要如下

原子性：（atomicity）  
原子性指的是事务是一个不可分割的工作单位，事务中的操作要么全部发生，要么都不发生  
（就像物理中，原子是最小不可分割的单位）  
一致性：（consistency）  
一致性指的是事务前后数据的完整性必须保持一致（比如说，转账：张三账户有2000元，李四账户有2000元，一共4000元  
张三项李四转账2000元后，一共还是4000元）  
事务必须是使[**数据库**](http://lib.csdn.net/base/mysql)从一个一致性状态变到另一个一致性状态。一致性与原子性是密切相关的  
隔离性：（isolation）  
隔离性指的是多个用户并发访问数据库是，一个用户的事务不能被其他用户的事务  
干扰，多个并发事务之间相互隔离  
持久性：（durability）。  
持久性指的是一个事务一旦被提交，他对数据库中的数据的改变是永久的，即使数据库故障  
也不应该对其有任何影响

## 3.Spring中事务的管理

Spring中主要提供了以下3个接口来对事务进行管理

PlatformTransactionManager 平台事务管理器  
主要用于事务的提交，回滚等说明  
TransactionDefintion  事务定义信息（隔离，传播，超时，只读）  
主要用于事务的隔离，传播，只读等说明  
TransactionStatus  事务具体运行状态  
是否是一个新的事务，是否有保存点，事务是否完成等说明

大致的运行过程：  
首先通过TransactionDefinition定义了事务（隔离，传播，超时，只读）等信息后，再交给PlatformTransactionManager平台事务管理器进行真正的事务管理，之后事务会产生一些相应的状态，之后就会保存到TransactionStatus中。

### 3.1平台事务管理器PlatformTransactionManager

主要定义了各个不同的数据库平台的一些接口，针对不同的数据库平台进行事务管理

org.springframework.jdbc.datasource.DataSourceTransactionManager  使用jdbc或Mybatis进行持久化时使用  
org.springframework.orm.hibernate3.HibernateTransactionManager使用Hibernate3.0版本进行持久化数据时使用  
org.springframework.orm.jdo.JdoTransactionManager  持久化机制为jdao  
org.springframework.orm.jpa.JpaTransactionManager  使用jpa进行持久化  
org.springframework.transaction.jta.JtaTransactionManager使用JTA来实现事务管理，在一个事务跨越多个资源时必须使用

我们在使用不同的数据库时候，可以选择不同事务管理器，还有这里列举了一些常用的，还有其他的可以参见SpringApi说明，也可以查看在线api:  http://tool.oschina.net/apidocs/#S

### 3.2事务定义信息接口 TransactionDefintion

主要用于声明事务的传播行为，和隔离级别等信息

假如一个事务不去考虑其隔离性，可能会引发如下问题（脏读，不可重复读，幻读）

##### 3.2.1事务的脏读，不可重复读，幻读

脏读：一个事务读取到了一个事务 改写但是未提交的数据，如果这些数据回滚，则读取到的数据是无效的  
不可重复读：在同一个事物中，多次读取同一数据，由于另外一个事务对该数据 修改提交，造成返回的结果不同。  
幻读（虚读）：一个事务读取几行记录后，另一个事务 插入或删除 了一些记录，导致再次读取的返回结果不同。

不可重复读和幻读，很相似，只是侧重点不同：

相同点都是一个事务读取了另一个提交了的数据  
不同点在于，不可重复读侧重点在于修改并提提交，幻读（虚度）在于删除或添加

可以参见博文：http://blog.csdn.net/v123411739/article/details/39298127

##### 3.2.2 事务的隔离级别

为了解决上诉问题，Spring在接口TransactionDefintion中定义了4中隔离级别，如下：

DEFAULT 使用数据默认的隔离级别（由spring根据使用的数据库决定）  
READ\_UNCOMMITED允许你读取还未提交的改变了的数据（可导致脏读，幻读，不可重复读）  
READ\_COMMINTED允许你读取事务已经提交后的数据（可防止脏读，但是幻读和不可重复读是有可能发生的）   
REPEATABLE\_READ对相同字段的多次读取是一致的，除非数据被事务本身改变(可防止脏读，不可重复读，当幻读任然可能发生)  
SERIALIZABLE 完全服从ACID的隔离级别，确保不发生脏读，幻读，不可重复读。这在所有的隔离级别中是最慢的，他是典型的  
通过完全锁定在事务中设计的数据表来完成的

提示：oracle默认的是：READ\_COMMINTED mysql默认是：REPEATABLE\_READ，前边的3个是我们经常需要使用的

##### 3.2.3事务的传播行为

一般我们的事务是在业务层中（Service）中使用的，假如遇到比较复杂的业务，需要在一个业务中调用另一个业务的方法：

如下：

业务逻辑ServiceA中的方法methoda()中的需要与业务逻辑ServiceB中的方法methodb()方法共同完成某个复杂的逻辑

假如都有事务，到底是使用谁的事务。

事务的传播行为，正是为了解决业务层中的方法相互调用的问题 的。

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52562950) [copy](http://blog.csdn.net/u014292162/article/details/52562950)

1. ServiceA{
2. methoda(){//需要调用serviceB中的业务方法methodb()共同完成
3. dao1.xxmethod();
4. serviceB.methodb();
6. }
7. }
9. ServiceB{
10. methodb(){
11. dao2.yymethod();
12. }
13. }
15. DAO1{
16. xxmethod(){
18. }
19. }
21. DAO2{
22. yymethod(){
24. }
26. }

Spring的事务定义信息接口TransactionDefintion还定义了如下7中事务传播行为常量，我们可以分为3类，（同一事物中，不同事务中，嵌套事务中）

**类一,两种在同一事物中**  
PROPAGATION\_REQUIRED--支持当前事务，如果当前没有事务，就新建一个事务。这是最常见的选择。(就比如上边的场景，methoda假如有事务  
则使用methoda的使用，假如methoda没有则新建一个事务)  
PROPAGATION\_SUPPORTS--支持当前事务，如果当前没有事务，就以非事务方式执行。(就比如上边的场景，methoda假如有事务  
则使用methoda的使用，假如methoda没有则不使用事务)  
PROPAGATION\_MANDATORY--支持当前事务，如果当前没有事务，就抛出异常。(就比如上边的场景，methoda假如有事务  
则使用methoda的使用，假如methoda没有则抛出异常)   
**类二，两者不再同一个事物中**  
PROPAGATION\_REQUIRES\_NEW--新建事务，如果当前存在事务，把当前事务挂起。(就比如上边的场景，methoda假如有事务挂起该事物  
不使用,而methodb新建一个事务)   
PROPAGATION\_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。(就比如上边的场景，methoda假如有事务挂起该事物  
不使用,而methodb不使用事务)    
PROPAGATION\_NEVER--以非事务方式执行，如果当前存在事务，则抛出异常。(就比如上边的场景，methoda假如有事务则抛出异常)    
  
**类三嵌套事务中**  
PROPAGATION\_NESTED 如果当前事务存在，则嵌套事务执行(如果在执行methoda完成的时候，就会使用事务设置一个保存点，再执行methodb，假如methodb没有异常，他们就一起提交了，如果  
发生了异常，你可根据自己的设定你可选择回滚到保存点位置，也可以回滚到最初的状态)

### 3.3  事务具体运行状态接口 TransactionStatus

里边主要定义了事务运行过程的一些具体状态（如是否是新的事务，是否只读，是否设置保存点）等等，这些都是可以获得的

## 4.Spring中的事务实现方式

Spring主要通过如下两种方式进行事务管理

-编程式事务管理  
主要通过TransactionTemplate手动管理事务，在实际开发中很少使用  
-使用XML配置声明  
主要通过Spring的AOP实现的，在开发中推荐使用（代码入侵性小）

#### 4.1Spring的编程式事务管理

所谓编程式事务指的是通过编码方式实现事务，即类似于JDBC编程实现事务管理。  
管理使用TransactionTemplate或者直接使用底层的PlatformTransactionManager。  
对于编程式事务管理，spring推荐使用TransactionTemplate。

这里我们通过转账的小案例来实现：

account账户表：

**[sql]** [view plain](http://blog.csdn.net/u014292162/article/details/52562950) [copy](http://blog.csdn.net/u014292162/article/details/52562950)

1. **create** **table** account(
2. id number(20) **primary** **key**,
3. **name** varchar2(20),
4. money number
5. );
7. **insert** **into** account **values**(1,'张三',1000);
8. **insert** **into** account **values**(2,'李四',1000);
9. **insert** **into** account **values**(3,'王五',1000);

外部文件db.properties

**[plain]** [view plain](http://blog.csdn.net/u014292162/article/details/52562950) [copy](http://blog.csdn.net/u014292162/article/details/52562950)

1. jdbc.driver=oracle.jdbc.driver.OracleDriver
2. jdbc.url=jdbc:oracle:thin:@localhost:1521:XE
3. jdbc.username=xxx
4. jdbc.password=xxx

配置文件applicationContext.xml

**[html]** [view plain](http://blog.csdn.net/u014292162/article/details/52562950) [copy](http://blog.csdn.net/u014292162/article/details/52562950)

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:context="http://www.springframework.org/schema/context"
4. xmlns:aop="http://www.springframework.org/schema/aop"
5. xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
7. http://www.springframework.org/schema/aop
8. http://www.springframework.org/schema/aop/spring-aop-3.2.xsd
9. http://www.springframework.org/schema/context
10. http://www.springframework.org/schema/context/spring-context.xsd"**>**
12. <!-- 引入外部文件 -->
13. **<context:property-placeholder** location="classpath:db.properties"**/>**
15. <!-- 配置c3p0的连接池 -->
16. **<bean** id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"**>**
17. **<property** name="driverClass"**>**
18. **<value>**${jdbc.driver}**</value>**
19. **</property>**
20. **<property** name="jdbcUrl"**>**
21. **<value>**${jdbc.url}**</value>**
22. **</property>**
23. **<property** name="user"**>**
24. **<value>**${jdbc.username}**</value>**
25. **</property>**
26. **<property** name="password"**>**
27. **<value>**${jdbc.password}**</value>**
28. **</property>**
29. **</bean>**
31. <!-- 配置业务层的类 -->
32. **<bean** id="accountService" class="com.xxx.spring.chap5.service.impl.AccountServiceImpl"**>**
33. **<property** name="accountDao" ref="accountDao"**></property>**
34. <!-- 注入事务管理模板 -->
35. **<property** name="transactionTemplate" ref="transactionTemplate"**></property>**
36. **</bean>**
37. <!-- 配置dao,Dao继承了JdbcDaoSupport后，只要注入了连接池就会有模板，就可以通过模板对数据库进行相应的操作，可以参见源码-->
38. **<bean** id="accountDao" class="com.xx.spring.chap5.dao.impl.AccountDaoImpl"**>**
39. **<property** name="dataSource" ref="dataSource"**></property>**
40. **</bean>**
42. <!-- 配置事务管理 -->
43. **<bean** id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager"**>**
44. **<property** name="dataSource" ref="dataSource"**></property>**
45. **</bean>**
47. <!-- 配置事务管理模板，Spring为了简化事务管理的代码，提供了事务管理模板 -->
48. **<bean** id="transactionTemplate" class="org.springframework.transaction.support.TransactionTemplate"**>**
49. **<property** name="transactionManager" ref="transactionManager"**></property>**
50. **</bean>**
51. **</beans>**

Service:业务逻辑接口

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52562950) [copy](http://blog.csdn.net/u014292162/article/details/52562950)

1. /\*\*
2. \* 转账的业务接口
3. \* \*/
4. **public** **interface** AccountService {
6. /\*\*
7. \* @param out 转出的账户
8. \* @param in 转入的账户
9. \* @param money 转账金额
10. \* \*/
11. **public** **void** transfer(String out,String in,Double money);
13. }

Service业务逻辑实现

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52562950) [copy](http://blog.csdn.net/u014292162/article/details/52562950)

1. **import** org.springframework.transaction.TransactionStatus;
2. **import** org.springframework.transaction.support.TransactionCallbackWithoutResult;
3. **import** org.springframework.transaction.support.TransactionTemplate;
4. **import** com.xxx.spring.chap5.dao.AccountDao;
5. **import** com.xxx.spring.chap5.service.AccountService;
6. /\*\*
7. \* 转账业务接口的具体实现
8. \* \*/
9. **public** **class** AccountServiceImpl **implements** AccountService {
11. **private** AccountDao accountDao;
13. **private** TransactionTemplate transactionTemplate;  //事务管理模板
15. **public** TransactionTemplate getTransactionTemplate() {
16. **return** transactionTemplate;
17. }
19. **public** **void** setTransactionTemplate(TransactionTemplate transactionTemplate) {
20. **this**.transactionTemplate = transactionTemplate;
21. }
23. **public** AccountDao getAccountDao() {
24. **return** accountDao;
25. }
27. **public** **void** setAccountDao(AccountDao accountDao) {
28. **this**.accountDao = accountDao;
29. }
31. @Override
32. **public** **void** transfer(**final** String out, **final** String in, **final** Double money) {
34. //使用事务模板execute中需要传入TransactionCallback的实现类对象
35. transactionTemplate.execute(**new** TransactionCallbackWithoutResult() {
36. @Override
37. **protected** **void** doInTransactionWithoutResult(TransactionStatus arg0) {
38. accountDao.outMoney(out, money);
39. accountDao.inMoney(in, money);
40. }
41. });
42. }
44. }

持久层Dao层接口

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52562950) [copy](http://blog.csdn.net/u014292162/article/details/52562950)

1. /\*\*
2. \* 转账的dao层接口
3. \* \*/
4. **public** **interface** AccountDao {
6. /\*\*
7. \* @param out 转出账户
8. \* @param money 转账金额
9. \* \*/
10. **public** **void** outMoney(String out,Double money);
12. /\*\*
13. \* @param in 转入账户
14. \* @param money 转账金额
15. \* \*/
16. **public** **void** inMoney(String in,Double money);
18. }

持久层Dao接口实现

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52562950) [copy](http://blog.csdn.net/u014292162/article/details/52562950)

1. **import** org.springframework.jdbc.core.support.JdbcDaoSupport;
2. **import** com.briup.spring.chap5.dao.AccountDao;
3. /\*\*
4. \* 转账的dao层接口实现
5. \* Dao继承了JdbcDaoSupport后，只要注入了连接池就会有模板，就可以通过模板对数据库进行相应的操作，
6. \* \*/
7. **public** **class** AccountDaoImpl **extends** JdbcDaoSupport **implements** AccountDao{
9. @Override
10. **public** **void** outMoney(String out, Double money) {
11. String sql = "update account set money = money - ?  where name = ?";
12. **this**.getJdbcTemplate().update(sql, money,out);
13. }
15. @Override
16. **public** **void** inMoney(String in, Double money) {
17. String sql = "update account set money = money + ?  where name = ?";
18. **this**.getJdbcTemplate().update(sql, money,in);
19. }
21. }

测试：

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52562950) [copy](http://blog.csdn.net/u014292162/article/details/52562950)

1. **import** org.junit.Test;
2. **import** org.springframework.context.ApplicationContext;
3. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
4. **import** com.xxx.spring.chap5.service.AccountService;
5. /\*\*
6. \* 转账测试类
7. \* \*/
8. **public** **class** SpringTransactionTest {
10. @Test
11. **public** **void** test1() **throws** Exception {
12. ApplicationContext ac = **new** ClassPathXmlApplicationContext("com/xxx/spring/chap5/applicationContext.xml");
13. AccountService accountService = ac.getBean("accountService",AccountService.**class**);
14. accountService.transfer("张三", "李四", 200.0);
15. }
16. }

数据库查询结果

**[plain]** [view plain](http://blog.csdn.net/u014292162/article/details/52562950) [copy](http://blog.csdn.net/u014292162/article/details/52562950)

1. 1   张三  800
2. 2   李四  1200
3. 3   王五  1000

假如Service层转账接口出现异常

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52562950) [copy](http://blog.csdn.net/u014292162/article/details/52562950)

1. @Override
2. **public** **void** transfer(**final** String out, **final** String in, **final** Double money) {
4. //使用事务模板TransactionCallback
5. transactionTemplate.execute(**new** TransactionCallbackWithoutResult() {
6. @Override
7. **protected** **void** doInTransactionWithoutResult(TransactionStatus arg0) {
8. accountDao.outMoney(out, money);
9. **int** i = 1/0;
10. accountDao.inMoney(in, money);
11. }
12. });
13. }

会发生异常java.lang.ArithmeticException: / by zero...

钱也不会转过去

#### 4.2Spring中声明式事务管理

管理建立在AOP之上的。其本质是对方法前后进行拦截，然后在目标方法开始之前创建或者加入一个事务，  
在执行完目标方法之后根据执行情况提交或者回滚事务。声明式事务最大的优点就是不需要通过编程的方式管理事务，  
这样就不需要在业务逻辑代码中掺杂事务管理的代码，只需在配置文件中做相关的事务规则声明  
(或通过基于@Transactional注解的方式)，便可以将事务规则应用到业务逻辑中。

这里主要说了解3中实现（基于代理实现，基于AspectJ实现，基于注解实现）

##### 4.2.1声明式事务管理-基于代理实现

这种方式，主要是使用TransactionProxyFactoryBean代理拦截器实现，通过配置事务代理器从而实现事务管理

配置文件：applicationContext2.xml

**[html]** [view plain](http://blog.csdn.net/u014292162/article/details/52562950) [copy](http://blog.csdn.net/u014292162/article/details/52562950)

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. **<span** style="white-space:pre"**>**    **</span>**xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:context="http://www.springframework.org/schema/context"
4. xmlns:aop="http://www.springframework.org/schema/aop"
5. xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
7. http://www.springframework.org/schema/aop
8. http://www.springframework.org/schema/aop/spring-aop-3.2.xsd
9. http://www.springframework.org/schema/context
10. http://www.springframework.org/schema/context/spring-context.xsd"**>**
11. <!-- 声明式事务管理-代理实现 -->
12. <!-- 引入外部文件 -->
13. **<context:property-placeholder** location="classpath:db.properties"**/>**
15. <!-- 配置c3p0的连接池 -->
16. **<bean** id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"**>**
17. **<property** name="driverClass"**>**
18. **<value>**${jdbc.driver}**</value>**
19. **</property>**
20. **<property** name="jdbcUrl"**>**
21. **<value>**${jdbc.url}**</value>**
22. **</property>**
23. **<property** name="user"**>**
24. **<value>**${jdbc.username}**</value>**
25. **</property>**
26. **<property** name="password"**>**
27. **<value>**${jdbc.password}**</value>**
28. **</property>**
29. **</bean>**
31. <!-- 配置业务层的类 -->
32. **<bean** id="accountService2" class="com.xxx.spring.chap5.service.impl.AccountServiceImpl2"**>**
33. **<property** name="accountDao" ref="accountDao2"**></property>**
34. **</bean>**
35. <!-- 配置dao,Dao继承了JdbcDaoSupport后，只要注入了连接池就会有模板，就可以通过模板对数据库进行相应的操作，可以参见源码-->
36. **<bean** id="accountDao2" class="com.xxx.spring.chap5.dao.impl.AccountDaoImpl2"**>**
37. **<property** name="dataSource" ref="dataSource"**></property>**
38. **</bean>**
39. <!-- 配置事务管理器 -->
40. **<bean** name="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager"**>**
41. **<property** name="dataSource" ref="dataSource"**></property>**
42. **</bean>**
44. <!-- 配置业务层的代理 -->
45. **<bean** id="accountServiceProxy" class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean"**>**
46. <!-- 配置目标类  -->
47. **<property** name="target" ref="accountService2"**></property>**
48. <!-- 注入事物管理器 -->
49. **<property** name="transactionManager" ref="transactionManager"**></property>**
50. <!-- 注入事物的相关属性,事物的隔离级别，传播行为等
51. key值为方法名可以使用通配符\*
52. value值为
53. --**>**
54. **<property** name="transactionAttributes"**>**
55. **<props>**
56. <!-- prop的格式
57. key为方法名，可以使用\*通配符号
58. value值的格式：
59. 1. PROPAGATION  :事务的传播行为
60. 2. ISOLATION    :事务隔离级别
61. 3. readOnly     :只读
62. 4. -Exception   :发生那些异常回滚
63. 5. +Exception   :发生那些事务不回滚
65. 之间使用,号隔开
66. --**>**
67. **<prop** key="trans\*"**>**PROPAGATION\_REQUIRED**</prop>**
68. **</props>**
69. **</property>**
70. **</bean>**
71. **</beans>**

Service实现改为如下：

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52562950) [copy](http://blog.csdn.net/u014292162/article/details/52562950)

1. **import** org.springframework.transaction.TransactionStatus;
2. **import** org.springframework.transaction.support.TransactionCallbackWithoutResult;
3. **import** org.springframework.transaction.support.TransactionTemplate;
5. **import** com.xxx.spring.chap5.dao.AccountDao;
6. **import** com.xxx.spring.chap5.service.AccountService;
8. /\*\*
9. \* 转账业务接口的具体实现
10. \* \*/
11. **public** **class** AccountServiceImpl2 **implements** AccountService {
13. **private** AccountDao accountDao;
15. **private** TransactionTemplate transactionTemplate;  //事务管理模板
17. **public** TransactionTemplate getTransactionTemplate() {
18. **return** transactionTemplate;
19. }
21. **public** **void** setTransactionTemplate(TransactionTemplate transactionTemplate) {
22. **this**.transactionTemplate = transactionTemplate;
23. }
25. **public** AccountDao getAccountDao() {
26. **return** accountDao;
27. }
29. **public** **void** setAccountDao(AccountDao accountDao) {
30. **this**.accountDao = accountDao;
31. }
33. @Override
34. **public** **void** transfer(**final** String out, **final** String in, **final** Double money) {
35. accountDao.outMoney(out, money);
36. accountDao.inMoney(in, money);
37. }
38. }

Dao层不变

测试类为：

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52562950) [copy](http://blog.csdn.net/u014292162/article/details/52562950)

1. /\*\*
2. \* 声明式事务管理--代理实现
3. \* \*/
4. @Test
5. **public** **void** test2() **throws** Exception {
6. ApplicationContext ac = **new** ClassPathXmlApplicationContext("com/xxx/spring/chap5/applicationContext2.xml");
7. AccountService accountService = ac.getBean("accountServiceProxy",AccountService.**class**);//代理对象
8. accountService.transfer("张三", "李四", 200.0);
9. }

查询结果：

**[plain]** [view plain](http://blog.csdn.net/u014292162/article/details/52562950) [copy](http://blog.csdn.net/u014292162/article/details/52562950)

1. <span style="white-space:pre">    </span>1  张三  800
2. 2   李四  1200
3. 3   王五  1000

假如Service中出现异常：

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52562950) [copy](http://blog.csdn.net/u014292162/article/details/52562950)

1. @Override
2. **public** **void** transfer(**final** String out, **final** String in, **final** Double money) {
3. accountDao.outMoney(out, money);
4. **int** i = 1/0;
5. accountDao.inMoney(in, money);
6. }

会抛出异常  
java.lang.ArithmeticException: / by zero  
  
但是钱没有多，也没有少，意思是本次操作失败会，保证账户的安全

##### 4.2.2声明式事务管理-基于AspectJ实现

SpringAOP中可以通过在xml文件中配置事务通知来对事务进行管理，这是一种更常用的方式

配置文件：applicationContext3.xml

**[html]** [view plain](http://blog.csdn.net/u014292162/article/details/52562950) [copy](http://blog.csdn.net/u014292162/article/details/52562950)

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:context="http://www.springframework.org/schema/context"
5. xmlns:tx="http://www.springframework.org/schema/tx"
6. xmlns:aop="http://www.springframework.org/schema/aop"
7. xsi:schemaLocation="http://www.springframework.org/schema/beans
8. http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
9. http://www.springframework.org/schema/tx
10. http://www.springframework.org/schema/tx/spring-tx.xsd
11. http://www.springframework.org/schema/aop
12. http://www.springframework.org/schema/aop/spring-aop-3.2.xsd
13. http://www.springframework.org/schema/context
14. http://www.springframework.org/schema/context/spring-context.xsd"**>**
15. <!-- 基于AspectJ的声明式事务管理 -->
16. <!-- 引入外部文件 -->
17. **<context:property-placeholder** location="classpath:db.properties" **/>**
19. <!-- 配置c3p0的连接池 -->
20. **<bean** id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"**>**
21. **<property** name="driverClass"**>**
22. **<value>**${jdbc.driver}**</value>**
23. **</property>**
24. **<property** name="jdbcUrl"**>**
25. **<value>**${jdbc.url}**</value>**
26. **</property>**
27. **<property** name="user"**>**
28. **<value>**${jdbc.username}**</value>**
29. **</property>**
30. **<property** name="password"**>**
31. **<value>**${jdbc.password}**</value>**
32. **</property>**
33. **</bean>**
35. <!-- 配置业务层的类 -->
36. **<bean** id="accountService2" class="com.xxx.spring.chap5.service.impl.AccountServiceImpl2"**>**
37. **<property** name="accountDao" ref="accountDao2"**></property>**
38. **</bean>**
39. <!-- 配置dao,Dao继承了JdbcDaoSupport后，只要注入了连接池就会有模板，就可以通过模板对数据库进行相应的操作，可以参见源码 -->
40. **<bean** id="accountDao2" class="com.xxx.spring.chap5.dao.impl.AccountDaoImpl2"**>**
41. **<property** name="dataSource" ref="dataSource"**></property>**
42. **</bean>**
43. <!-- 配置事务管理器 -->
44. **<bean** name="transactionManager"
45. class="org.springframework.jdbc.datasource.DataSourceTransactionManager"**>**
46. **<property** name="dataSource" ref="dataSource"**></property>**
47. **</bean>**
49. <!-- 配置事务通知 -->
50. **<tx:advice** id="txAdvice" transaction-manager="transactionManager"**>**
51. <!-- 设置事物属性 -->
52. **<tx:attributes>**
53. <!-- 设置各种方法的
54. propagation 为传播行为
55. isolation 事务的隔离级别
56. read-only 设置之都属性
57. rollback-for 发生生么异常回滚
58. no-rollback-for 发生那些异常不回滚
59. --**>**
60. **<tx:method** name="tran\*" propagation="REQUIRED" read-only="true" **/>**
61. **<tx:method** name="get\*" propagation="REQUIRED" read-only="true" **/>**
62. **<tx:method** name="find\*" propagation="REQUIRED" read-only="true" **/>**
63. **<tx:method** name="save\*" propagation="REQUIRED" read-only="false"
64. rollback-for="java.lang.Exception" **/>**
65. **<tx:method** name="insert\*" propagation="REQUIRED" read-only="false"
66. rollback-for="java.lang.Exception" **/>**
67. **<tx:method** name="update\*" propagation="REQUIRED" read-only="false"
68. rollback-for="java.lang.Exception" **/>**
69. **<tx:method** name="delete\*" propagation="REQUIRED" read-only="false"
70. rollback-for="java.lang.Exception" **/>**
71. **<tx:method** name="\*" propagation="REQUIRED" rollback-for="java.lang.Exception" **/>**
72. **</tx:attributes>**
73. **</tx:advice>**
75. <!-- 配置AOP -->
76. **<aop:config>**
77. <!-- 配置切入点,表示切入点为类AccountServiceImpl2下的所有方法 -->
78. **<aop:pointcut**
79. expression="execution(\* com.xxx.spring.chap5.service.impl.AccountServiceImpl2.\*(..))"
80. id="pointcut" **/>**
81. <!-- 配置切面,表示AccountServiceImpl2下的所有方法都使用txAdivice增强 -->
82. **<aop:advisor** advice-ref="txAdvice" pointcut-ref="pointcut" **/>**
83. **</aop:config>**
84. **</beans>**

Service实现

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52562950) [copy](http://blog.csdn.net/u014292162/article/details/52562950)

1. **import** org.springframework.transaction.annotation.Propagation;
2. **import** org.springframework.transaction.annotation.Transactional;
3. **import** org.springframework.transaction.support.TransactionTemplate;
4. **import** com.xxx.spring.chap5.dao.AccountDao;
5. **import** com.xxx.spring.chap5.service.AccountService;
7. /\*\*
8. \* 转账业务接口的具体实现
9. \*
10. \* \*/
11. **public** **class** AccountServiceImpl2 **implements** AccountService {
13. **private** AccountDao accountDao;
15. **private** TransactionTemplate transactionTemplate;  //事务管理模板
17. **public** TransactionTemplate getTransactionTemplate() {
18. **return** transactionTemplate;
19. }
21. **public** **void** setTransactionTemplate(TransactionTemplate transactionTemplate) {
22. **this**.transactionTemplate = transactionTemplate;
23. }
25. **public** AccountDao getAccountDao() {
26. **return** accountDao;
27. }
28. **public** **void** setAccountDao(AccountDao accountDao) {
29. **this**.accountDao = accountDao;
30. }
31. @Override
32. **public** **void** transfer(**final** String out, **final** String in, **final** Double money) {
33. accountDao.outMoney(out, money);
34. accountDao.inMoney(in, money);
35. }
37. }

测试：

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52562950) [copy](http://blog.csdn.net/u014292162/article/details/52562950)

1. /\*\*
2. \* 声明式事务管理-基于AspectJ实现
3. \* \*/
4. @Test
5. **public** **void** test3() **throws** Exception {
6. ApplicationContext ac = **new** ClassPathXmlApplicationContext("com/xxx/spring/chap5/applicationContext3.xml");
7. AccountService accountService = ac.getBean("accountService2",AccountService.**class**);
8. accountService.transfer("张三", "李四", 200.0);
9. }

查询结果：

**[plain]** [view plain](http://blog.csdn.net/u014292162/article/details/52562950) [copy](http://blog.csdn.net/u014292162/article/details/52562950)

1. 1   张三  800
2. 2   李四  1200
3. 3   王五  1000

##### 4.2.3声明式事务管理-基于注解实现

这种也是一种比较常见的使用，主要在xml文件中开始事务注解后，在需要使用事务的业务中添加@Transactionl注解

可以参见：http://www.cnblogs.com/younggun/archive/2013/07/16/3193800.html

配置文件：applicationContext4.xml

**[html]** [view plain](http://blog.csdn.net/u014292162/article/details/52562950) [copy](http://blog.csdn.net/u014292162/article/details/52562950)

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:context="http://www.springframework.org/schema/context"
5. xmlns:tx="http://www.springframework.org/schema/tx"
6. xmlns:aop="http://www.springframework.org/schema/aop"
7. xsi:schemaLocation="http://www.springframework.org/schema/beans
8. http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
9. http://www.springframework.org/schema/aop
10. http://www.springframework.org/schema/aop/spring-aop-3.2.xsd
11. http://www.springframework.org/schema/tx
12. http://www.springframework.org/schema/tx/spring-tx.xsd
13. http://www.springframework.org/schema/context
14. http://www.springframework.org/schema/context/spring-context.xsd"**>**
15. <!-- 基于注解式事务管理 -->
16. <!-- 引入外部文件 -->
17. **<context:property-placeholder** location="classpath:db.properties" **/>**
19. <!-- 配置c3p0的连接池 -->
20. **<bean** id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"**>**
21. **<property** name="driverClass"**>**
22. **<value>**${jdbc.driver}**</value>**
23. **</property>**
24. **<property** name="jdbcUrl"**>**
25. **<value>**${jdbc.url}**</value>**
26. **</property>**
27. **<property** name="user"**>**
28. **<value>**${jdbc.username}**</value>**
29. **</property>**
30. **<property** name="password"**>**
31. **<value>**${jdbc.password}**</value>**
32. **</property>**
33. **</bean>**
35. <!-- 配置业务层的类 -->
36. **<bean** id="accountService2" class="com.xxx.spring.chap5.service.impl.AccountServiceImpl2"**>**
37. **<property** name="accountDao" ref="accountDao2"**></property>**
38. **</bean>**
39. <!-- 配置dao,Dao继承了JdbcDaoSupport后，只要注入了连接池就会有模板，就可以通过模板对数据库进行相应的操作，可以参见源码 -->
40. **<bean** id="accountDao2" class="com.xxx.spring.chap5.dao.impl.AccountDaoImpl2"**>**
41. **<property** name="dataSource" ref="dataSource"**></property>**
42. **</bean>**
43. <!-- 配置事务管理器 -->
44. **<bean** name="transactionManager"
45. class="org.springframework.jdbc.datasource.DataSourceTransactionManager"**>**
46. **<property** name="dataSource" ref="dataSource"**></property>**
47. **</bean>**
49. <!-- 注解事务管理器配置 -->
50. **<tx:annotation-driven** transaction-manager="transactionManager"**/>**
51. **</beans>**

Sservice业务层实现：

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52562950) [copy](http://blog.csdn.net/u014292162/article/details/52562950)

1. **import** org.springframework.transaction.annotation.Propagation;
2. **import** org.springframework.transaction.annotation.Transactional;
3. **import** org.springframework.transaction.support.TransactionTemplate;
4. **import** com.xxx.spring.chap5.dao.AccountDao;
5. **import** com.xxx.spring.chap5.service.AccountService;
6. /\*\*
7. \* 转账业务接口的具体实现
8. \*
9. \* @Transactional中注解的属性：
10. \* propagation:事务的传播行为
11. \* isolation:事务的隔离级别
12. \* readOnly:是否只读
13. \* rollbackFor：发生那些异常回滚
14. \* noRollbackFor:发生那些异常不回滚,这些默认可以不写使用@Transactional就行
15. \* \*/
16. @Transactional(propagation=Propagation.REQUIRED,readOnly=**true**,rollbackFor={RuntimeException.**class**, Exception.**class**})
17. **public** **class** AccountServiceImpl2 **implements** AccountService {
19. **private** AccountDao accountDao;
20. **public** AccountDao getAccountDao() {
21. **return** accountDao;
22. }
24. **public** **void** setAccountDao(AccountDao accountDao) {
25. **this**.accountDao = accountDao;
26. }
27. @Override
28. **public** **void** transfer(**final** String out, **final** String in, **final** Double money) {
29. accountDao.outMoney(out, money);
30. accountDao.inMoney(in, money);
31. }
33. }

测试：

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52562950) [copy](http://blog.csdn.net/u014292162/article/details/52562950)

1. /\*\*
2. \* 4声明式事务管理-基于注解的实现
3. \* \*/
4. @Test
5. **public** **void** test4() **throws** Exception {
6. ApplicationContext ac = **new** ClassPathXmlApplicationContext("com/xxx/spring/chap5/applicationContext4.xml");
7. AccountService accountService = ac.getBean("accountService2",AccountService.**class**);
8. accountService.transfer("张三", "李四", 200.0);
9. }

查询结果：

**[plain]** [view plain](http://blog.csdn.net/u014292162/article/details/52562950) [copy](http://blog.csdn.net/u014292162/article/details/52562950)

1. 1   张三  800
2. 2   李四  1200
3. 3   王五  1000