1.事务(Transaction)介绍

事务（Transaction），一般是指要做的或所做的事情。**在计算机术语中是指访问并可能更新数据库中各种数据项的一个程序执行单元(unit)**。

官方定义：事务是由一系列对系统中数据进行访问与更新的操作所组成的一个程序执行单元。

> *大象装进冰箱：*
>
> *1.开门*
>
> *2.装大象*
>
> *3.关门*

这里我们以取钱的例子来讲解：比如你去ATM机取1000块钱，大体有两个步骤：第一步输入密码金额，银行卡扣掉1000元钱；第二步从ATM出1000元钱。这两个步骤必须是要么都执行要么都不执行。如果银行卡扣除了1000块但是ATM出钱失败的话，你将会损失1000元；如果银行卡扣钱失败但是ATM却出了1000块，那么银行将损失1000元。

如何保证这两个步骤不会出现一个出现异常了，而另一个执行成功呢？事务就是用来解决这样的问题。事务是一系列的动作，它们综合在一起才是一个完整的工作单元，这些动作必须全部完成，如果有一个失败的话，那么事务就会回滚到最开始的状态，仿佛什么都没发生过一样。 在企业级应用程序开发中，事务管理是必不可少的技术，用来确保数据的完整性和一致性。

2.事务的四个特性（ACID）

①、原子性（Atomicity）：事务是一个原子操作，由一系列动作组成。事务的原子性确保动作要么全部完成，要么完全不起作用。

②、一致性（Consistency）：一旦事务完成（不管成功还是失败），系统必须确保它所建模的业务处于一致的状态，而不会是部分完成部分失败。在现实中的数据不应该被破坏。

③、隔离性（Isolation）：可能有许多事务会同时处理相同的数据，因此每个事务都应该与其他事务隔离开来，防止数据损坏。（类似于并发）。

隔离级别	脏读	不可重复读	幻读
读未提交（Read uncommitted）	V	V	V
读已提交（Read committed）	X	V	V
可重复读（Repeatable read）	X	X	V
可串行化（Serializable）	X	X	X

④、持久性（Durability）：一旦事务完成，无论发生什么系统错误，它的结果都不应该受到影响，这样就能从任何系统崩溃中恢复过来。通常情况下，事务的结果被写到持久化存储器中。

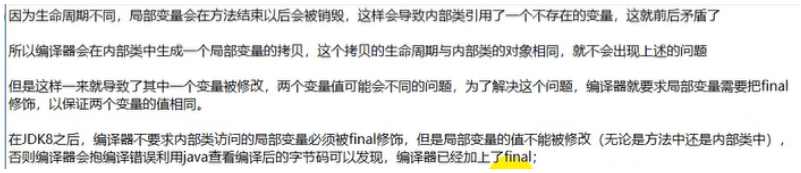
详情看《事务.md》中的内容

3.Spring 事务管理的核心接口

3.1 Spring事务管理的实现方式

1. 编程式事务管理（过时了，不详细介绍，只有代码没有笔记，代码主要看AccountServiceImpl类中的transferMoney方法以及配置文件）：通过编写代码实现的事务管理，包括定义事务的开始、正常执行后的事务提交和提交时的事务回滚。

   这种种方式存在的问题：
   匿名内部类访问外部类属性需要给这个属性加final（JDK8之前）

2. 声明式事务管理：

   通过AOP技术实现的事务管理，其主要思想是将事务管理作为一个“切面”代码单独编写，然后通过AOP技术将事务管理的“切面”代码织入到业务目标类中。

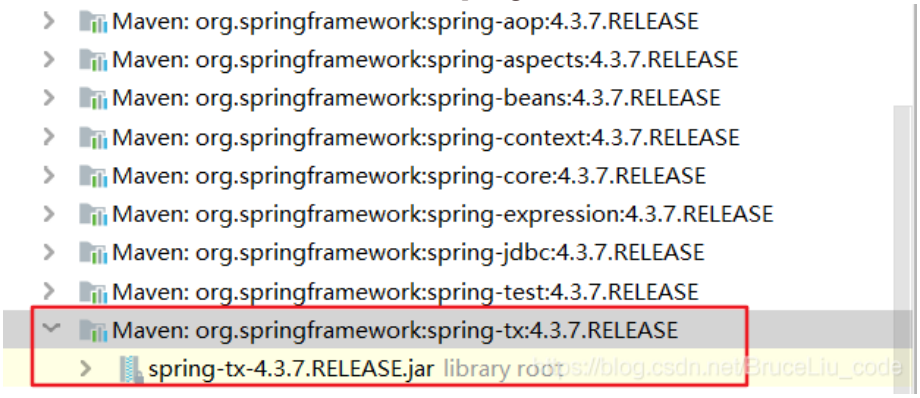
   两种配置方式

   a. XML配置
   b. 注解配置

声明式事务管理最大的优点在于：开发者无需通过编程的方式来管理事务，只需在配置文件中进行相关的事务规则声明，就可以将事务规则应用到业务逻辑中。

3.2 核心包、接口

首先我们创建一个Java工程，然后导入 Spring 核心事务包（专门做事务管理）



我们打开Spring的核心事务包，查看如下类：org.springframework.transaction，以下三个类是Spring中事务的顶级接口！

1. PlatformTransactionManager接口：可以根据属性管理事务。该接口中提供了三个管理事务的方法：

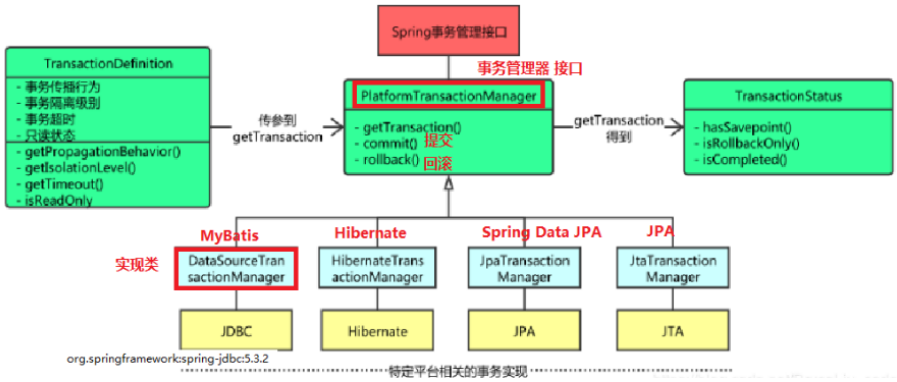
方法	说明
TransactionStatus getTransaction(TransactionDefinition definition)	用于获取事务状态信息
void commit(TransactionStatus status)	用于提交事务
void rollback(TransactionStatus status)	用于回滚事务

2. TransactionDefinition接口：用于定义事务的属性，其中包括了事务的隔离级别、事务的传播行为、事务的超时时间和是否为只读事务。（具体看3.2节）

3. TransactionStatus接口：用于界定事务的状态 。该接口中提供了6个方法：

方法	说明
boolean isNewTransaction()	判断当前事务是否为新事务
boolean hasSavepoint()	判断当前事务是否创建了一个保存点
boolean isRollbackOnly()	判断当前事务是否被标记为rollback-only
void setRollbackOnly()	将当前事务标记为rollback-only
boolean isCompleted()	判断当前事务是否已经完成（提交或回滚）
void flush()	刷新底层的修改到数据库

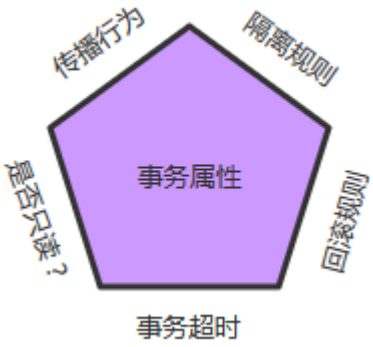
上面所示的三个类文件便是Spring的事务管理接口。如下图所示：下面我们分别对这三个接口进行简单的介绍



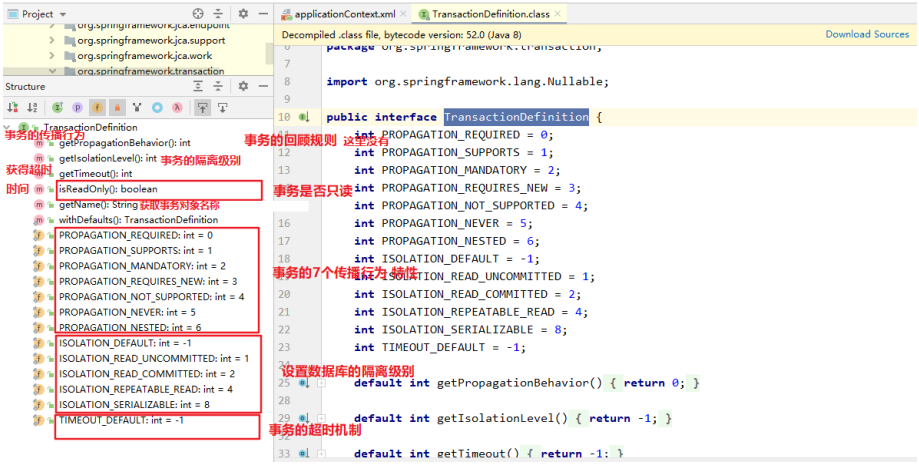
3.3 基本事务属性的定义

上面讲到的事务管理器接口`PlatformTransactionManager`通过`getTransaction(TransactionDefinition definition)`方法来得到事务，这个方法里面的参数是TransactionDefinition类，这个类就定义了一些基本的事务属性。

那么什么是事务属性呢？事务属性可以理解成事务的一些基本配置，描述了事务策略如何应用到方法上。事务属性包含了5个方面，如图所示：



TransactionDefinition 接口方法如下：



3.3.1.传播行为（面试题）

传播行为：**当事务方法（被增强的目标类中的方法）被另一个事务方法（被增强的目标类中的方法）调用时，必须指定事务应该如何传播。**

Spring 定义了如下七中传播行为，这里以A业务和B业务之间如何传播事务为例说明：

B包容A

①、**PROPAGATION_REQUIRED** ： required , 必须。默认值，A如果有事务，B将使用该事务；如果A没有事务，B将创建一个新的事务。

②、PROPAGATION_SUPPORTS：supports ， 支持。A如果有事务，B将使用该事务；如果A没有事务，B将以非事务执行。

③、PROPAGATION_MANDATORY：mandatory ， 强制。A如果有事务，B将使用该事务；如果A没有事务，B将抛异常。

④、**PROPAGATION_REQUIRES_NEW** ： requires_new，必须新的。如果A有事务，将A的事务挂起，B创建一个新的事务；如果A没有事务，B创建一个新的事务。

⑤、PROPAGATION_NOT_SUPPORTED ： not_supported ,不支持。如果A有事务，将A的事务挂起，B将以非事务执行；如果A没有事务，B将以非事务执行。

⑥、PROPAGATION_NEVER ： never，从不。如果A有事务，B将抛异常；如果A没有事务，B将以非事务执行。

⑦、PROPAGATION_NESTED ： nested ， 嵌套。如果A当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行与PROPAGATION_REQUIRED类似的操作。

3.3.2.隔离级别

隔离级别：定义了一个事务可能受其他并发事务影响的程度。

并发事务引起的问题：
在典型的应用程序中，多个事务并发运行，经常会操作相同的数据来完成各自的任务。并发虽然是必须的，但可能会导致以下的问题。
①、**脏读（Dirty reads）**——脏读发生在一个事务读取了另一个事务改写但尚未提交的数据时。如果改写在稍后被回滚了，那么第一个事务获取的数据就是无效的。
②、**不可重复读（Nonrepeatable read）**——不可重复读发生在一个事务执行相同的查询两次或两次以上，但是每次都得到不同的数据时。这通常是因为另一个并发事务在两次查询期间进行了更新。
③、**幻读（Phantom read）**——幻读与不可重复读类似。它发生在一个事务（T1）读取了几行数据，接着另一个并发事务（T2）插入了一些数据时。在随后的查询中，第一个事务（T1）就会发现多了一些原本不存在的记录。
注意：不可重复读重点是修改，而幻读重点是新增或删除。

在 Spring 事务管理中，为我们定义了如下的隔离级别：
①、**ISOLATION_DEFAULT**：使用后端数据库默认的隔离级别(不同的数据隔离级别不同)
②、**ISOLATION_READ_UNCOMMITTED**：最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读
③、**ISOLATION_READ_COMMITTED**（Oracle）：允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生
④、**ISOLATION_REPEATABLE_READ**（mysql）：对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生
⑤、**ISOLATION_SERIALIZABLE**：最高的隔离级别，完全服从ACID的隔离级别，确保阻止脏读、不可重复读以及幻读，也是最慢的事务隔离级别，因为它通常是通过完全锁定事务相关的数据库表来实现的。

上面定义的隔离级别，在 Spring 的 TransactionDefinition.class 中也分别用常量 -1,0,1,2,4,8表示。比如 ISOLATION_DEFAULT 的定义：

```
/**
 * Use the default isolation level of the underlying
datastore.
 * All other levels correspond to the JDBC isolation
levels.
 * @see java.sql.Connection
 */
int ISOLATION_DEFAULT = -1;
```

3.3.3.只读

这是事务的第三个特性，是否为只读事务。如果事务只对后端的数据库进行该操作，数据库可以利用事务的只读特性来进行一些特定的优化。通过将事务设置为只读，你就可以给数据库一个机会，让它应用它认为合适的优化措施。
Spring会管理事务，但是查询一般都设置成只读事务，性能会高！

3.3.4.事务超时

为了使应用程序很好地运行，事务不能运行太长的时间。因为事务可能涉及对后端数据库的锁定，所以长时间的事务会不必要的占用数据库资源。事务超时就是事务的一个定时器，在特定时间内事务如果没有执行完毕，那么就会自动回滚，而不是一直等待其结束。

3.3.5.回滚规则

事务五边形的最后一个方面是一组规则，这些规则定义了哪些异常会导致事务回滚而哪些不会。默认情况下，事务只有遇到运行期异常时才会回滚，而在遇到检查型异常时不会回滚（这一行为与EJB的回滚行为是一致的）。但是你可以声明事务在遇到特定的检查型异常时像遇到运行期异常那样回滚。同样，你还可以声明事务遇到特定的异常不回滚，即使这些异常是运行期异常。
可以指定何种类型的异常西是否需要回滚撤销！！

4. 转账案例（不用事务实现转账）

我们还是以转账为实例。不用事务看如何实现转账。在数据库中有如下表account ,内容如下：

```
CREATE TABLE `ar_account` (
  `id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
  `username` VARCHAR(20) NOT NULL,
  `money` DECIMAL(10,2) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=INNODB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8;

/*Data for the table `ar_account` */

INSERT  INTO `ar_account`(`id`,`username`,`money`) VALUES
(1,'cat','1000.00');
INSERT  INTO `ar_account`(`id`,`username`,`money`) VALUES
(2,'Tom','1000.00');
```

创建Mapper接口：

```
/**
 * @author bruceliu
 * @create 2019-09-22 23:49
 * @description
 */
public interface AccountMapper {

    /**
     * 加钱方法
     *
     * @param id
     * @param money
     */
```

```
    void increaseMoney(@Param("id") Integer id,
@Param("money") Double money);

    /**
     * 减钱方法
     *
     * @param id
     * @param money
     */
    void decreaseMoney(@Param("id") Integer id,
@Param("money") Double money);
}
```

创建**XML**文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.ssm.mapper.AccountMapper">

    <select id="increaseMoney">
        update ar_account set money=money+#{money} where
id=#{id}
    </select>

    <select id="decreaseMoney">
        update ar_account set money=money-#{money} where
id=#{id}
    </select>

</mapper>
```

创建**Service**接口：

```
/**
 * @author bruceliu
 * @create 2019-09-22 23:53
 * @description
 */
public interface AccountService {

    // 转账业务
    void transfer(Integer from, Integer to, Double money);
}
```

创建**Service**接口实现层：

```java
/**
 * @author bruceliu
 * @create 2019-09-22 23:54
 * @description
 */
@Service
public class AccountServiceImpl implements AccountService
{

    @Autowired
    AccountMapper accountMapper;

    @Override
    public void transfer(Integer from, Integer to, Double
money) {
        accountMapper.decreaseMoney(from, money);
        accountMapper.increaseMoney(to, money);
    }
}
```

测试类：

```java
/**
 * @author bruceliu
 * @create 2019-09-17 11:58
 * @description 启动Spring框架测试
 */
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext.xml")
public class TestSpring {

    @Resource
    AccountService accountService;

    @Test
    public void test1(){
        accountService.transfer(1, 2, 100.0);
    }
}
```

查看数据库表 **account**

id	username	money
1	cat	900
2	Tom	1100

上面的结果和我们想的一样，Tom 账户 money 减少了1000块。而 Marry 账户金额增加了1000块。

这时候问题来了，比如在 Tom 账户 money 减少了1000块正常。而 Marry 账户金额增加时发生了异常，实际应用中比如断电（这里我们人为构造除数不能为0的异常），如下：

```
 @Override
    public void transfer(Integer from, Integer to, Double
money) {
        accountMapper.decreaseMoney(from, money);
        System.out.println(100/0);
        accountMapper.increaseMoney(to, money);
    }
```

那么这时候我们执行测试程序，很显然会报错，那么数据库是什么情况呢？

```
java.lang.ArithmeticException: / by zero

    at
com.ssm.service.impl.AccountServiceImpl.transfer(AccountSe
rviceImpl.java:22)
    at com.ssm.test.TestSpring.test1(TestSpring.java:25)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native
Method)
    at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAc
cessorImpl.java:62)
    at
sun.reflect.DelegatingMethodAccessorImpl.invoke(Delegating
MethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:497)
    at
org.junit.runners.model.FrameworkMethod$1.runReflectiveCal
l(FrameworkMethod.java:50)
    at
org.junit.internal.runners.model.ReflectiveCallable.run(Re
flectiveCallable.java:12)
    at
org.junit.runners.model.FrameworkMethod.invokeExplosively(
FrameworkMethod.java:47)
    at
org.junit.internal.runners.statements.InvokeMethod.evaluat
e(InvokeMethod.java:17)
    at
org.springframework.test.context.junit4.statements.RunBefo
reTestMethodCallbacks.evaluate(RunBeforeTestMethodCallback
s.java:75)
    at
org.springframework.test.context.junit4.statements.RunAfte
rTestMethodCallbacks.evaluate(RunAfterTestMethodCallbacks.
java:86)
    at
org.springframework.test.context.junit4.statements.SpringR
epeat.evaluate(SpringRepeat.java:84)
    at
org.junit.runners.ParentRunner.runLeaf(ParentRunner.java:3
25)
```

```
    at
org.springframework.test.context.junit4.SpringJUnit4ClassR
unner.runChild(SpringJUnit4ClassRunner.java:252)
    at
org.springframework.test.context.junit4.SpringJUnit4ClassR
unner.runChild(SpringJUnit4ClassRunner.java:94)
    at
org.junit.runners.ParentRunner$3.run(ParentRunner.java:290
)
    at
org.junit.runners.ParentRunner$1.schedule(ParentRunner.jav
a:71)
    at
org.junit.runners.ParentRunner.runChildren(ParentRunner.ja
va:288)
    at
org.junit.runners.ParentRunner.access$000(ParentRunner.jav
a:58)
    at
org.junit.runners.ParentRunner$2.evaluate(ParentRunner.jav
a:268)
```

数据库**account**：

id	username	money
1	cat	800
2	Tom	1100

我们发现，程序执行报错了，但是数据库 Tom 账户金额依然减少了 1000 块，但是 Marry 账户的金额却没有增加。这在实际应用中肯定是不允许的，那么如何解决呢？

5. 声明式事务处理实现转账（基于AOP的xml 配置）

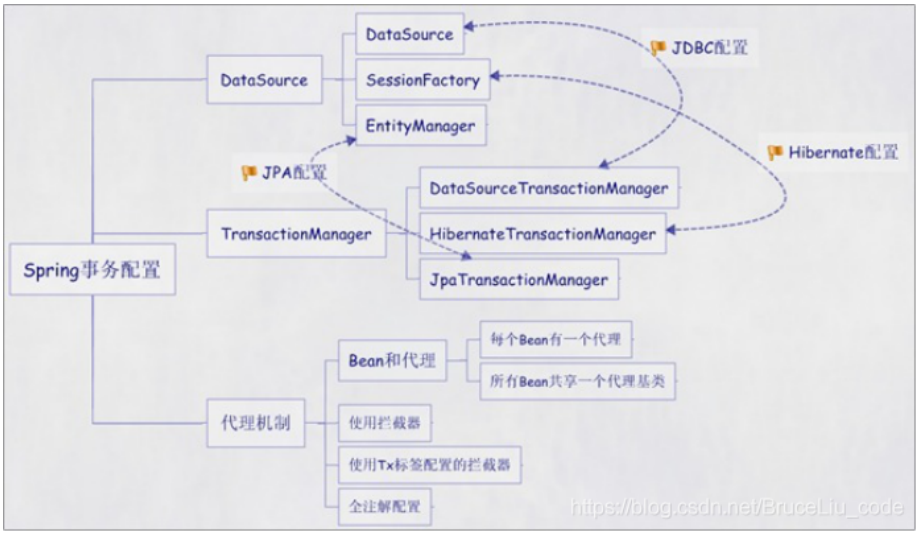
5.1.TransactionManager

在不同平台，操作事务的代码各不相同，因此spring提供了一个 TransactionManager 接口：
DateSourceTransactionManager 用于 JDBC 的事务管理

在《Spring框架专题(七)-Spring整合MyBatis.md》中的XML的基础上添加以下bean标签：

```
<!--6. 配置平台事务管理器，Spring中只要用到事务，就需要配置，无论是
编程式事务还是声明式事务都要配置这个标签
-->
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTrans
actionManager">
     <property name="dataSource" ref="ds"/>
</bean>
```

HibernateTransactionManager 用于 Hibernate 的事务管理
JpaTransactionManager 用于 Jpa 的事务管理



5.2.添加tx命名空间

事务基础组件，对 DAO 的支持
修改 applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
    xmlns="http://www.springframework.org/schema/beans"
xmlns:context="http://www.springframework.org/schema/conte
xt"
    xmlns:aop="http://www.springframework.org/schema/aop"
xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:p="http://www.springframework.org/schema/p"

xsi:schemaLocation="http://www.springframework.org/schema/
beans
http://www.springframework.org/schema/beans/spring-beans-
4.2.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-
context-4.2.xsd
http://www.springframework.org/schema/aop
```

```
http://www.springframework.org/schema/aop/spring-aop-
4.2.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-
4.2.xsd">



</beans>
```

在ts命名空间下提供了>元素来配置事务的通知（增强处理）。当使用>元素配置了事务的增强处理后，就可以通过编写的AOP配置让Spring自动对目标生成代理.

5.3.添加事务相关配置

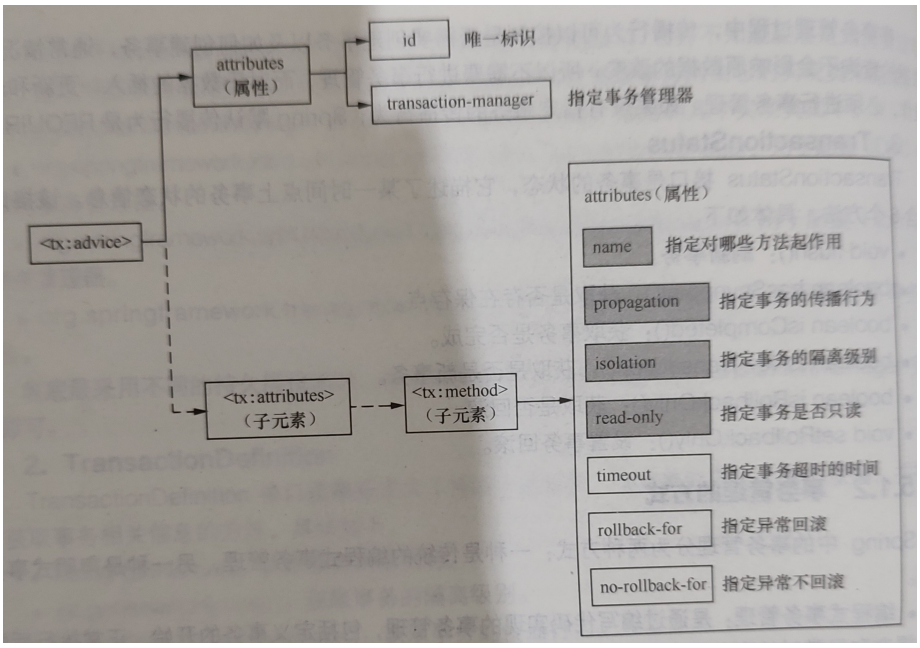
修改applicationContext.xml

```
<!-- 配置事务:
       Spring事务配置有2种方案:
          1.使用XML来配置声明式事务!
          2.使用注解来配置声明式事务!(推荐)
    -->

    <!--事务平台管理器 Spring框架管理事务，有一个核心的接口:
PlatformTransactionManager 这个接口有很多实现类，其中有一个专门
是JDBC事务管理的类：DataSourceTransactionManager-->
    <!--如果是MyBatis框架，那么事务管理器:
DataSourceTransactionManager-->
    <!--6. 配置平台事务管理器，Spring中只要用到事务，就需要配置，
无论是编程式事务还是
        声明式事务都要配置这个标签
    -->
    <bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTrans
actionManager">
        <property name="dataSource" ref="ds"/>
    </bean>


<!--下面的tx命名空间下的、aop命名空间下的内容定位了哪个目标方法需要
被织入事务-->
    <!--将事务织入到目标方法之上-->
    <tx:advice id="advice"  transaction-
manager="transactionManager">
        <!--指定事务管理器-->
        <tx:attributes>

            <!--转账方法需要事务  propagation="REQUIRED" 默认
值! 必须在事务中进行，有事务那么就是使用，没有就创建
```

```
                isolation="DEFAULT" 数据库的四大隔离级别，如果不配置就是默认
                Mysql默认隔离级别：REPEATABLE_READ

                timeout="-1" 事务超时机制，没有超时 timeout="3" 目标方法3秒之后超时，自动回滚！

                no-rollback-for="java.lang.ArithmeticException" 配置具体的某一个异常 不回滚  默认是所有的异常都回滚！
            -->
            <!--目标类中的transferMoney方法-->
            <tx:method name="transferMoney" propagation="REQUIRED" isolation="DEFAULT" timeout="30"/>
            <!--目标类中以add开头的方法-->
            <tx:method name="add*" propagation="REQUIRED" isolation="REPEATABLE_READ"/>
            <!--目标类中以insert开头的方法-->
            <tx:method name="insert*" propagation="REQUIRED" isolation="REPEATABLE_READ"/>
            <!--目标类中以update开头的方法-->
            <tx:method name="update*" propagation="REQUIRED" isolation="REPEATABLE_READ"/>
            <!--目标类中以del开头的方法-->
            <tx:method name="del*" propagation="REQUIRED" isolation="REPEATABLE_READ"/>

            <!--查询的话，不需要在事务中进行 查询配置只读事务 read-only="true"-->
            <!--目标类中以find开头的方法-->
            <tx:method name="find*" propagation="NOT_SUPPORTED" read-only="true" isolation="REPEATABLE_READ"/>
            <!--目标类中以aerach开头的方法-->
            <tx:method name="search*" propagation="NOT_SUPPORTED" read-only="true" isolation="REPEATABLE_READ"/>

            <!--propagation="SUPPORTS" 可有可无-->
            <tx:method name="*" propagation="SUPPORTS" read-only="true" isolation="REPEATABLE_READ"/>

        </tx:attributes>
    </tx:advice>

    <!--配置切面-->
    <aop:config>
        <aop:pointcut id="pointcut" expression="execution(* com.ssm.service.impl.*.*(..))"/>
        <aop:advisor advice-ref="advice" pointcut-ref="pointcut"/>
    </aop:config>
```

配置介绍：

分布图：



> *tx:advice 是用于配置事务相关信息, transaction-manager属性是引入对应类型的事务管理;*
> *jdbc/mybatias : DataSourceTransactionManager*
>
> *hibernate: HibernateTransactionManager*
>
> *JPA:JPATransactionManager*
>
> ---
>
> *tx:attributes 此标签所配置的是 哪些方法可以作为事务方法(为后面切点进行补充)*
>
> ---
>
> *tx:method 标签设置具体要添加事务的方法和其他属性*
>
> 1. *name 是必须的,表示与事务属性关联的方法名(业务方法名),对切入点进行细化。通配符* 可以用来指定一批关联到相同的事务属性的方法。如：'get*'、'handle*'、'on*Event' 等等.*
> 2. *propagation 不是必须的， 默认值是REQUIRED 表示事务传播行为, 包括 REQUIRED,SUPPORTS,MANDATORY,REQUIRES_NEW,NOT_SUPPORTED,NEVER,NESTED*

PROPAGATION_REQUIRED	0	当前有事务就用当前的，没有就用新的
PROPAGATION_SUPPORTS	1	事务可有可无，不是必须的
PROPAGATION_MANDATORY	2	当前一定要有事务，不然就抛异常
PROPAGATION_REQUIRES_NEW	3	无论是否有事务，都起个新的事务
PROPAGATION_NOT_SUPPORTED	4	不支持事务，按非事务方式运行
PROPAGATION_NEVER	5	不支持事务，如果有事务则抛异常
PROPAGATION_NESTED	6	当前有事务就在当前事务里再起一个事务

3. *isolation　不是必须的 默认值DEFAULT*

隔离性	值	脏读	不可重复读	幻读
ISOLATION_READ_UNCOMMITTED	1	√	√	√
ISOLATION_READ_COMMITTED	2	×	√	√
ISOLATION_REPEATABLE_READ	3	×	×	√
ISOLATION_SERIALIZABLE	4	×	×	×

4. *timeout　不是必须的 默认值-1(永不超时)　表示事务超时的时间（以秒为单位）*
5. *read-only　不是必须的 默认值false 不是只读的　表示事务是否只读?*
6. *rollback-for 不是必须的　表示将被触发进行回滚的Exception(s)；以逗号分开。*
   如：'com.itqf.MyBusinessException,ServletException'
7. *no-rollback-for 不是必须的　表示不被触发进行回滚的Exception(s)；以逗号分开。*
   如：'com.foo.MyBusinessException,ServletException'

aop:config标签 设置事务的切点,配置参与事务的类和对应的方法.

注意:
aop:config和tx:advice 但是两者并不冲突, aop:config面向切面编程的切点,选择对应的方法进行切入,而tx:adivce是设置事务的相关的属性和描述,换句话说,aop:config选择了对应的切入点,tx:advice是在这些切入点上根据 method name属性再次进行筛选!!!

6.配置声明事务（基于AOP的 注解 配置）

除了基于XML的事务配置,Spring还提供了基于注解的事务配置,即通过@Transactional对需要事务增强的Bean接口,实现类或者方法进行标注,在容器中配置基于注解的事务增强驱动,即可启用注解的方式声明事务!

6.1.使用@Transactional注解

顺着原来的思路,使用`@Transactional`对基于aop/tx命名空间的事务配置进行改造!
修改service类添加@Transactional注解

```
//对业务类进行事务增强的标注
 /**
     * 转账方法
     * @param fromName
```

```
     * @param toName
     * @param money
     * @return
     */
    @Transactional(propagation=
Propagation.REQUIRED,isolation =
Isolation.DEFAULT,readOnly = false,timeout =
-1,rollbackFor = Exception.class)
    @Override
    public int transferMoney(String fromName, String
toName, Double money) {
        int count1 = accountMapper.OutMoney(fromName,
money);
        int i=100/0;
        int count2 = accountMapper.InMoney(toName, money);
        return count1+count2;
    }
```

因为注解本身具有一组默认的事务属性,所以往往只要在需要事务的业务类中添加一个@Transactional注解,就完成了业务类事务属性的配置!
当然,注解只能提供元数据,它本身并不能完成事务切面织入的功能.因此,还需要在Spring的配置中通过一行配置'通知'Spring容器对标注@Transactional注解的Bean进行加工处理!
配置:

```
<!-- 6.配置平台事务管理器 -->
<bean id="transactionManager"

class="org.springframework.jdbc.datasource.DataSourceTrans
actionManager">
    <property name="dataSource" ref="dataSource" />
</bean>

<!--对标注@Transactional注解的Bean进行加工处理,以织入事物管理切面
7.开启注解事务支持
-->
<tx:annotation-driven transaction-
manager="transactionManager" />
```

在默认情况, `<tx:annotation-driven />` 中transaction-manager属性会自动使用名为 "transactionManager" 的事务管理器.所以,如果用户将事务管理器的id定义为 transactionManager , 则可以进一步将①处的配置简化为 **<tx:annotation-driven />.**

使用以上测试用例即可使用以上测试用例即可

6.2.@Transactional其他方面介绍

- 关于@Transactional的属性
  基于@Transactional注解的配置和基于xml的配置一样,它拥有一组普

适性很强的默认事务属性,往往可以直接使用默认的属性.

```
- 事务传播行为: PROPAGATION_REQUIRED.
- 事务隔离级别: ISOLATION_DEFAULT.
- 读写事务属性:读/写事务.
- 超时时间:依赖于底层的事务系统默认值
- 回滚设置:任何运行期异常引发回滚,任何检查型异常不会引发回滚.
```

默认值可能适应大部分情况,但是我们依然可以可以自己设定属性,具体属性表如下:

属 性 名	说 明
propagation	事务传播行为，通过以下枚举类提供合法值： org.springframework.transaction.annotation.Propagation 例如：@Transactional(propagation=Propagation.REQUIRES_NEW)
isolation	事务隔离级别，通过以下枚举类提供合法值： org.springframework.transaction.annotation.Isolation 例如：@Transactional(isolation=Isolation.READ_COMMITTED)
readOnly	事务读写性，布尔型。例如：@Transactional(readOnly=true)
timeout	超时时间，int 型，以秒为单位。例如：@Transactional(timeout=10)
rollbackFor	一组异常类，遇到时进行回滚，类型为：Class<? extends Throwable>[]，默认值为{}。例如： @Transactional(rollbackFor={SQLException.class})。多个异常之间可用逗号分隔
rollbackForClassName	一组异常类名，遇到时进行回滚，类型为 String[]，默认值为{}。例如： @Transactional(noRollbackForClassName={"Exception"})
noRollbackFor	一组异常类，遇到时不回滚，类型为 Class<? extends Throwable>[]，默认值为{}
noRollbackForClassName	一组异常类名，遇到时不回滚，类型为 String[]，默认值为{}

- 在方法使用注解
  方法出添加注解会覆盖类定义处的注解,如果有写方法需要使用特殊的事务属性,则可以在类注解的基础上提供方法注解,如下:

```
@Transactional(propagation=
Propagation.NOT_SUPPORTED,readOnly = true)
public class AccountServiceImpl implements AccountService
{

    @Autowired
    AccountMapper accountMapper;

    @Transactional(propagation=
Propagation.REQUIRED,isolation =
Isolation.DEFAULT,readOnly = false,timeout =
-1,rollbackFor = Exception.class)
    @Override
    public int addAccount(Account account) {
        return accountMapper.addAccount(account);
    }

    /**
     * 转账方法
     * @param fromName
     * @param toName
     * @param money
     * @return
```

```
     */
    @Transactional(propagation=
Propagation.REQUIRED,isolation =
Isolation.DEFAULT,readOnly = false,timeout =
-1,rollbackFor = Exception.class)
    @Override
    public int transferMoney(String fromName, String
toName, Double money) {
        int count1 = accountMapper.OutMoney(fromName,
money);
        int i=100/0;
        /*try {
            Thread.sleep(6000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
        }*/
        int count2 = accountMapper.InMoney(toName, money);
        return count1+count2;
    }
}
```