Spring 事务管理

1 事务简介

事务管理是企业级应用程序开发中必不可少的技术, 主要用来确保数据的完整性和一致性. 事务就是一系列的动作, 它们被当做一个单独的工作单元. 这些动作要么全部完成, 要么全部不起作用

事务的四个关键属性(ACID) 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. **一致性(consistency)**: 一旦所有事务动作完成, 事务就被提交. 数据和资源就处于一种满足业务规则的一致性状态中. **隔离性(isolation)**: 可能有许多事务会同时处理相同的数据, 因此每个事物都应该与其他事务隔离开来, 防止数据损坏. **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. 通常情况下, 事务的结果被写到持久化存储器中.

2 Spring 中的事务管理

作为企业级应用程序框架, Spring 在不同的事务管理 API 之上定义了一个抽象层. 而应用程序开发人员不必了解底层的事务管理 API, 就可以使用 Spring 的事务管理机制.

Spring 既支持编程式事务管理, 也支持声明式的事务管理.

编程式事务管理:

将事务管理代码嵌入到业务方法中来控制事务的提交和回滚. 在编程式管理事务时, 必须在每个事务操作中包含额外的事务管理代码.

声明式事务管理:

大多数情况下比编程式事务管理更好用. 它将事务管理代码从业务方法中分离出来, 以声明的方式来实现事务管理. 事务管理作为一种横切关注点, 可以通过 AOP 方法模块化. Spring 通过 Spring AOP 框架支持声明式事务管理.

注解声明式事务管理

3 Spring 中的事务管理器

Spring 从不同的事务管理 API 中抽象了一整套的事务机制. 开发人员不必了解底层的事务 API, 就可以利用这些事务机制. 有了这些事务机制, 事务管理代码就能独立于特定的事务技术了.

JDBC操作:

在应用程序中只需要处理一个数据源, 而且通过 JDBC 存取

DataSourceTransactionManager

JTA操作

在 JavaEE 应用服务器上用 JTA(Java Transaction API) 进行事务管理

JtaTransactionManager

Hibernate操作

框架存取数据库

TransactionManager

.....

4 案例开发

1、转帐功能

帐户表：account

字段名	类型	说明	描述
id			主键ID
name			姓名
balance			余额
cardid			卡号

2、商品下单功能

商品表：product

字段名	类型	说明	描述
id			主键ID
name			商品名称
count			数量
price			价格

5 Spring声明式注解事务

spring_core.xml:

```
<!-- 配置 事务管理器  -->
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"></property>
</bean>
<!-- 启用声明式注解事务  -->
<tx:annotation-driven transaction-manager="transactionManager" />
```

@Transactional 注解声明式地管理事务

@Transactional 可以作用于接口、接口方法、类以及类方法上。当作用于类上时，该类的所有 public 方法将都具有该类型的事务属性，同时，我们也可以在方法级别使用该标注来覆盖类级别的定义。

建议：虽然 @Transactional 注解可以作用于接口、接口方法、类以及类方法上，但是 Spring 建议不要在接口或者接口方法上使用该注解，

@Transactional 注解应该只被应用到 public 方法上，这是由 Spring AOP 的本质决定的。如果你在 protected、private 或者默认可见性的方法上使用 @Transactional 注解，这将被忽略，也不会抛出任何异常。

6 Spring 事务回滚机制

spring事务管理器回滚一个事务的推荐方法是在当前事务的上下文内抛出异常。

spring事务管理器会捕捉任何未处理的异常，然后依据规则决定是否回滚抛出异常的事务。

默认配置下，spring只有在抛出的异常为运行时unchecked异常时才回滚该事务，也就是抛出的异常为RuntimeException的子类(Errors也会导致事务回滚)，而抛出checked异常则不会导致事务回滚。

常用运行时异常：空指针异常、除数不为0异常等

7 @Transactional 注解详解

属性	类型	描述
value	String	可选的限定描述符，指定使用的事务管理器
propagation	enum: Propagation	可选的事务传播行为设置
isolation	enum: Isolation	可选的事务隔离级别设置
readOnly	boolean	读写或只读事务，默认读写
timeout	int (in seconds granularity)	事务超时时间设置
rollbackFor	Class对象数组，必须继承自Throwable	导致事务回滚的异常类数组
rollbackForClassName	类名数组，必须继承自Throwable	导致事务回滚的异常类名字数组
noRollbackFor	Class对象数组，必须继承自Throwable	不会导致事务回滚的异常类数组
noRollbackForClassName	类名数组，必须继承自Throwable	不会导致事务回滚的异常类名字数组

事务传播属性

当事务方法被另一个事务方法调用时, 必须指定事务应该如何传播. 例如: 方法可能继续在现有事务中运行, 也可能开启一个新事务, 并在自己的事务中运行. 事务的传播行为可以由传播属性指定. Spring 定义了 7 种类传播行为.

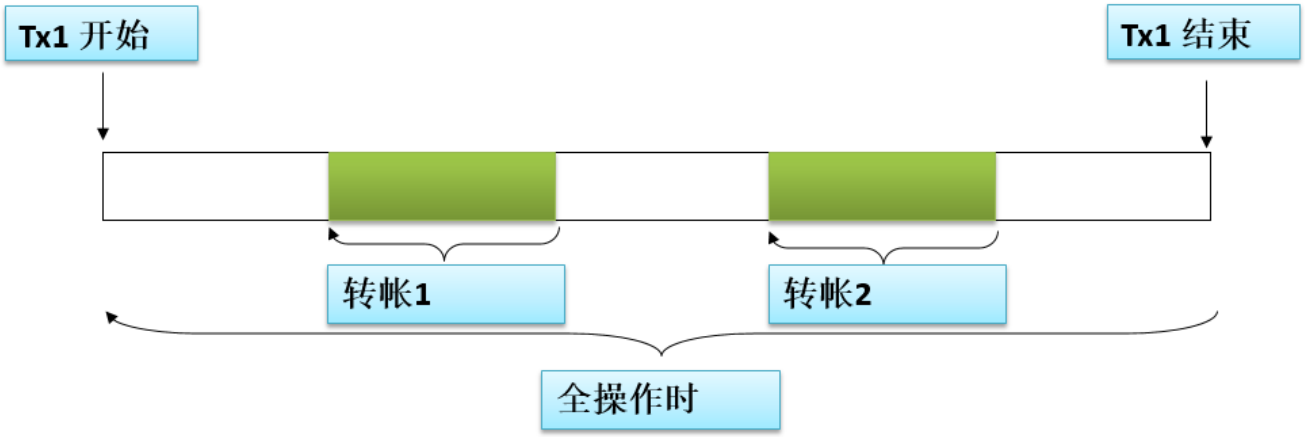
传播属性	描述
REQUIRED	如果有事务在运行，当前的方法就在这个事务内运行，否则，就启动一个新的事务，并在自己的事务内运行
REQUIRED_NEW	当前的方法必须启动新事务，并在它自己的事务内运行. 如果有事务正在运行，应该将它挂起
SUPPORTS	如果有事务在运行，当前的方法就在这个事务内运行. 否则它可以不运行在事务中.
NOT_SUPPORTE	当前的方法不应该运行在事务中. 如果有运行的事务，将它挂起
MANDATORY	当前的方法必须运行在事务内部，如果没有正在运行的事务，就抛出异常
NEVER	当前的方法不应该运行在事务中. 如果有运行的事务，就抛出异常
NESTED	如果有事务在运行，当前的方法就应该在这个事务的嵌套事务内运行. 否则，就启动一个新的事务，并在它自己的事务内运行.

REQUIRED_NEW嵌套的事务里的方法都在一个类里的话，里面的即时是新建的事务，执行完方法也不会提交（如果想实现，事务的方法需要放在不同类中）

REQUIRED 传播行为

当前方法被另一个事务方法 调用时，将会合并成一个事务。

当 前转帐（1人对1人）方法被另一个事务方法 调用时, 它默认会在现有的事务内运行. 这个默认的传播行为就是 REQUIRED. 因此在 全操作方法的开始和终止边界内只有一个事务. 这个事务只在 全操作 方法结束的时候被提交,事务传播属性可以在 @Transactional 注解的 propagation 属性中定义

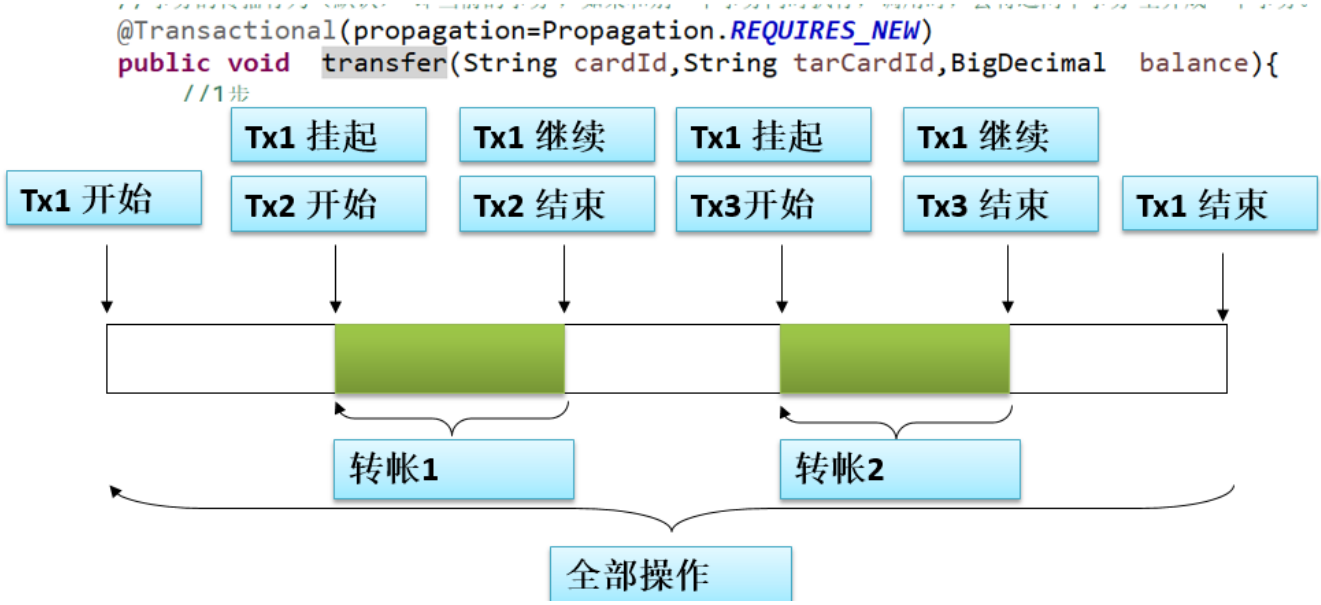



```
@Transactional(propagation=Propagation.REQUIRED)
@Override
public int transferAll(int cardId, int[] targetCardIds, double balance) {
    for(int i=0;i<targetCardIds.length;i++){
        this.transfer(cardId, targetCardIds[i], balance);
    }
    return 1;
}
```

REQUIRES_NEW 传播行为

另一种常见的传播行为是 REQUIRES_NEW.

它表示该方法必须启动一个新事务, 并在自己的事务内运行. 如果有事务在运行, 就应该先挂起它.



注意：

REQUIRES_NEW嵌套的事务里的方法都在一个类里的话，里面的即时是新建的事务，执行完方法也不会提交。

原因是A方法（有事务）调用B方法（要独立新事务），如果两个方法写在同一个类里，spring的事务会只处理同一个事务。

1：需要将两个方法分别写在不同的类里。 2：调用B方法的时候，将service自己注入自己，用这个注入对象来调用B方法。

```
@Service
public class AccountPartService {//部分转帐成功
    @Autowired
    private AccountPartService pService;

    @Transactional
    public void transferPart(String cardId,String[] tarCardIds,BigDecimal  balance){
        for(String tarCardId : tarCardIds){
            pService.transfer(cardId, tarCardId, balance);
        }
    }
```

AccountServiceImpl:

```
//调用B方法的时候，将service自己注入自己，用这个注入对象来调用B方法。
@Autowired
private AccountService accountService;

//全部转帐
@Transactional(propagation=Propagation.REQUIRED)
@Override
public int transferAll(int cardId, int[] targetCardIds, double balance) {
    for(int i=0;i<targetCardIds.length;i++){
        accountService.transfer(cardId, targetCardIds[i], balance);
    }
    return 1;
```

```
}

//单个转帐功能
@Transactional(propagation=Propagation.REQUIRES_NEW)
@Override
public int transfer(int cardId, int targetCardId, double balance) {
    //1 查询当前余额是否满足转出金额
    Account ac=accountDao.selectByCardId(cardId);
    if(ac.getBalance()<balance){
        throw new AccountException(cardId+"余额不足");
    }
    //2 cardId进行更新 减
    int num=accountDao.updateSubByCardId(cardId, balance);
    if(num<1){
        //操作失败
        throw new AccountException(cardId+"余额更新失败");
    }
    //3 targetCardId进行更新加
    int num1=accountDao.updateAddByCardId(targetCardId, balance);
    if(num1<1){
        //操作失败
        throw new AccountException(targetCardId+"余额更新失败3");
    }
    System.out.println(cardId+"像"+targetCardId+"转帐成功: "+balance);
    return 1;
}
```

8 并发事务所导致的问题

当同一个应用程序或者不同应用程序中的多个事务在同一个数据集上并发执行时, 可能会出现许多意外的问题

并发事务所导致的问题可以分为下面三种类型: 脏读:一个事务正在对数据进行更新操作，但是更新还未提交，另一个事务这时也来操作这组数据，并且读取了前一个事务还未提交的数据，而前一个事务如果操作失败进行了回滚，后一个事务读取的就是错误数据，这样就造成了脏读。 **不可重复读:**一个事务多次读取同一数据，在该事务还未结束时，另一个事务也对该数据进行了操作，而且在第一个事务两次读取之间，第二个事务对数据进行了更新，那么第一个事务前后两次读取到的数据是不同的，这样就造成了不可重复读。 **幻读:**第一个数据正在查询符合某一条件的数据，这时，另一个事务又插入了一条符合条件的数据，第一个事务在第二次查询符合同一条件的数据时，发现多了一条前一次查询时没有的数据，仿佛幻觉一样，这就是幻像读

Spring 支持的事务隔离级别

隔离级别	描述
DEFAULT	使用底层数据库的默认隔离级别. 对于大多数数据库来说, 默认隔离级别都是READ_COMMITED
READ_UNCOMMITTED	允许事务读取未被其他事物提交的变更. 脏读, 不可重复读和幻读的问题都会出现
READ_COMMITED	只允许事务读取已经被其它事务提交的变更. 可以避免脏读, 但不可重复读和幻读问题仍然可能出现
REPEATABLE_READ	确保事务可以多次从一个字段中读取相同的值. 在这个事务持续期间, 禁止其他事物对这个字段进行更新. 可以避免脏读和不可重复读, 但幻读的问题仍然存在.
SERIALIZABLE	确保事务可以从一个表中读取相同的行. 在这个事务持续期间, 禁止其他事务对该表执行插入, 更新和删除操作. 所有并发问题都可以避免, 但性能十分低下.

事务的隔离级别要得到底层数据库引擎的支持, 而不是应用程序或者框架的支持. Oracle 支持的 2 种事务隔离级别: READ_COMMITED , SERIALIZABLE Mysql 支持 4 中事务隔离级别.

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

总结:

从理论上来说, 事务应该彼此完全隔离, 以避免并发事务所导致的问题. 然而, 那样会对性能产生极大的影响, 因为事务必须按顺序运行. 在实际开发中, 为了提升性能, 事务会以较低的隔离级别运行.

作业

商品表：product

字段名	类型	说明	描述
id			主键ID
name			商品名称
count			数量
price			价格

帐户表：account

字段名	类型	说明	描述
id			主键ID
name			姓名
balance			余额
cardid			卡号