浅谈 Spring 的事务隔离级别与传播性

作者：jianzh5
原文链接：https://ld246.com/article/1571651587211
来源网站：链滴

这篇文章以一个问题开始，如果你知道答案的话就可以跳过不看啦@(o·ɪ·)@

Q:在一个批量任务执行的过程中，调用多个子任务时，如果有一些子任务发生异常，只是回滚那些出异常的任务，而不是整个批量任务，请问在Spring中事务需要如何配置才能实现这一功能呢?

隔离级别

隔离性（Isolation）作为事务特性的一个关键特性，它要求每个读写事务的对象对其他事务的操作对能相互分离，即该事务提交前对其他事务都不可见，在数据库层面都是使用锁来实现。

事务的隔离级别从低到高有以下四种：

● READ UNCOMMITTED（未提交读）：这是最低的隔离级别，其含义是允许一个事务读取另外一事务没有提交的数据。READ UNCOMMITTED是一种危险的隔离级别，在实际开发中基本不会使用主要是由于它会带来**脏读**问题。

时间	事务1	事务2	备注
1	读取库存为100	---	---
2	扣减库层50	---	剩余50
3		扣减库层50	---
4		提交事务	库存保存为0
5	回滚事务	---	事务回滚
0

脏读对于要求数据一致性的应用来说是致命的，目前主流的数据库的隔离级别都不会设置成READ UN OMMITTED。不过脏读虽然看起来毫无用处，但是它主要优点是并发能力高，适合那些对数据一致没有要求而追求高并发的场景。

● READ COMMITTED（读写提交）：它是指一个事务只能读取另外一个事务已经提交的数据，不能取未提交的数据。READ COMMITTED会带来**不可重复读**的问题：

时间	事务1	事务2	备注
1	读取库存为1		
2	扣减库存		事务未提交
3		读取库存为1	
4	提交事务		库存变成0
5		扣减库存	库存为0，
法扣减

不可重复读和脏读的区别是：脏读读取到的是未提交的数据，而不可重复读读到的确实已经提交的数，但是违反了数据库事务一致性的要求。

一般来说，不可重复读的问题是可以接受的，因为其读到的是已经提交的数据，本身并不会带来很大问题。因此，很多数据库如（ORACLE，SQL SERVER）将其默认隔离级别设置为READ COMMITTE，允许不可重复读的现象。

● REPEATABLE READ （可重复读）：可重复读的目标是为了克服READ COMMITED中出现的不可复读，它指在同一个事务内的查询都是与事务开始时刻一致，以上表为例，在REPEATABLE READ隔

级别下它会发生如下变化：

时间	事务1	事务2	备注
1	读取库存为1		
2	扣减库存		事务未提交
3		读取库存	**不允许读取等待事务1提交**
4	提交事务		库存变成0
5		读取库存	库存为0，法扣减

REPEATABLE READ虽然解决了不可重复读问题，但是他又会带来**幻读**问题，幻读是指，在一个事务，第一次查询某条记录，发现没有，但是，当试图更新这条不存在的记录时，竟然能成功，并且，再读取同一条记录，它就神奇地出现了。

时间	事务A	事务2	备注
1	begin	begin	
2		读取id为100的数据有数据	
3	插入id为100的数据		
4	提交事务		
5		读取id为100的数据有数据	
6		更新id为100的数据功	
7		读取id为100的数据取成功	
8		提交事务	

事务B在第2步第一次读取id=99的记录时，读到的记录为空，说明不存在id=99的记录。随后，事务在第3步插入了一条id=99的记录并提交。事务B在第5步再次读取id=99的记录时，读到的记录仍然空，但是，事务B在第6步试图更新这条不存在的记录时，竟然成功了，并且，事务B在第8步再次读取i=99的记录时，记录出现了。

● SERIALIZABLE（串行化）：数据库最高的隔离级别，它要求所有的SQL都会按照顺序执行，这样以克服上述所有隔离出现的各种问题，能够完全包住数据的一致性。

Spring中配置隔离级别

在Spring项目中配置隔离级别只需要做如下操作

```
@Transactional(isolation = Isolation.SERIALIZABLE)
public int insertUser(User user){
    return userDao.insertUser(user);
}
```

上面的代码中我们使用了串行化的隔离级别来包住数据的一致性，这使它将阻塞其他的事务进行并发所以它只能运用在那些低并发而又需要保证数据一致性的场景下。

隔离级别字典：

```
DEFAULT(-1), ## 数据库默认级别
READ_UNCOMMITTED(1),
READ_COMMITTED(2),
REPEATABLE_READ(4),
SERIALIZABLE(8);
```

传播行为

在Spring中，当一个方法调用另外一个方法时，可以让事务采取不同的策略工作，如新建事务或者挂当前事务等，这便是事务的传播行为。

定义

在Spring的事务机制中对数据库存在7种传播行为，通过枚举类Propagation定义。

```
public enum Propagation {
    /**
     * 需要事务，默认传播性行为。
     * 如果当前存在事务，就沿用当前事务，否则新建一个事务运行子方法
     */
    REQUIRED(0),
    /**
     * 支持事务，如果当前存在事务，就沿用当前事务，
     * 如果不存在，则继续采用无事务的方式运行子方法
     */
    SUPPORTS(1),
    /**
     * 必须使用事务，如果当前没有事务，抛出异常
     * 如果存在当前事务,就沿用当前事务
     */
    MANDATORY(2),
    /**
     * 无论当前事务是否存在，都会创建新事务允许方法
     * 这样新事务就可以拥有新的锁和隔离级别等特性，与当前事务相互独立
     */
    REQUIRES_NEW(3),
    /**
     * 不支持事务，当前存在事务时，将挂起事务，运行方法
     */
    NOT_SUPPORTED(4),
    /**
     * 不支持事务，如果当前方法存在事务，将抛出异常，否则继续使用无事务机制运行
     */
    NEVER(5),
    /**
     * 在当前方法调用子方法时，如果子方法发生异常
     * 只回滚子方法执行过的SQL，而不回滚当前方法的事务
     */
```

```
    NESTED(6);
    ......
}
```

日常开发中基本只会使用到REQUIRED(0),REQUIRES_NEW(3),NESTED(6)三种。

NESTED和REQUIRES_NEW是有区别的。NESTED传播行为会沿用当前事务的隔离级别和锁等特性，而EQUIRES_NEW则可以拥有自己独立的隔离级别和锁等特性。

NESTED的实现主要依赖于数据库的保存点（SAVEPOINT）技术，SAVEPOINT记录了一个保存点，以通过ROLLBACK TO SAVEPOINT来回滚到某个保存点。如果数据库支持保存点技术时就启用保存技术；如果不支持就会新建一个事务去执行代码，也就相当于REQUIRES_NEW。

Transactional自调用失效

如果一个类中自身方法的调用，我们称之为自调用。如一个订单业务实现类OrderServiceImpl中有methodA方法调用了自身类的methodB方法就是自调用，如：

```
@Transactional
public void methodA(){
    for (int i = 0; i < 10; i++) {
        methodB();
    }
}

@Transactional(isolation = Isolation.READ_COMMITTED,propagation = Propagation.REQUIRE
_NEW)
public int methodB(){
    ......
}
```

在上面方法中不管methodB如何设置隔离级别和传播行为都是不生效的。即自调用失效。

这主要是由于@Transactional的底层实现原理是基于AOP实现，而AOP的原理是动态代理，在自调的过程中是类自身的调用，而不是代理对象去调用，那么就不会产生AOP，于是就发生了自调用失败现象。

要克服这个问题，有2种方法：

- 编写两个Service,用一个Service的methodA去调用另外一个Service的methodB方法，这样就是代对象的调用，不会有问题；
- 在同一个Service中，methodA不直接调用methodB，而是先从Spring IOC容器中重新获取代理象`OrderServiceImpl·,获取到后再去调用methodB。说起来有点乱，还是show you the code。

```
public class OrderServiceImpl implements OrderService,ApplicationContextAware {
    private ApplicationContext applicationContext = null;

    @Override
    public void setApplicationContext(ApplicationContext applicationContext) {
        this.applicationContext = applicationContext;
    }

    @Transactional
```

```
    public void methodA(){
        OrderService orderService = applicationContext.getBean(OrderService.class);
        for (int i = 0; i < 10; i++) {
            orderService.methodB();
        }
    }

    @Transactional(isolation = Isolation.READ_COMMITTED,propagation = Propagation.REQUI
ES_NEW)
    public int methodB(){
        ......
    }

}
```

上面代码中我们先实现了ApplicationContextAware接口，然后通过applicationContext.getBean()取了OrderService的接口对象。这个时候获取到的是一个代理对象，也就能正常使用AOP的动态代理。

回到最开始的那个问题，看完这篇文章是不是有答案了呢？