damn，又差一点点被干掉了
==========

哈喽，这里是最锋利的矛，在现如今的行情下，感觉每天都生活在要么被优化的恐惧中

最近搞了个线上bug，在火急火燎解决后，赶紧来和大家一起分享下

bug来啦

前两天线上突然出现告警有个订单撤销的接口报错无法撤销，在测试环境测试订单批量入库撤销场景后，发现单个订单撤销没有问题，多个撤销没有问题，多个订单内有明细指向相同的库存行时，撤销异常。

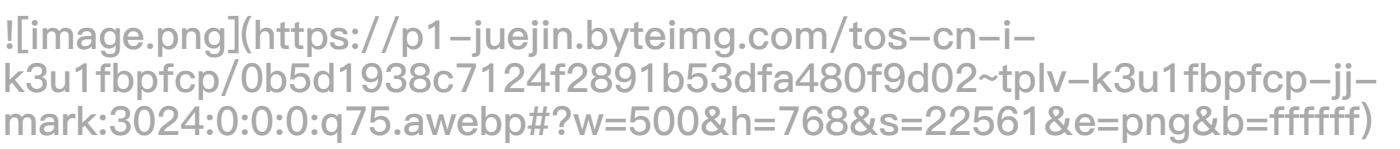
问题排查

代码大概如下：

```
public class OderService {

    private InventoryService inventoryService;
    private FinanceApi financeApi;
    private TransactionTemplate transactionTemplate;


    public List<String> orderReturn(List<Long> orderIds) {

        List<CompletableFuture<String>> futureReturn = new ArrayList<>();
        orderIds.forEach(orderId ->
futureReturn.add(CompletableFuture.supplyAsync(() -> {

            transactionTemplate.execute(status -> {

                // ....订单自身逻辑

                // 库存撤销
                inventoryService.inventoryOperation();
```

```
                // 财务撤销
                Boolean financeFlag = financeApi.financeReturn();
                // 财务撤销失败回滚订单事物
                if (!financeFlag){
                    throw new RuntimeException();
                }

                return Boolean.TRUE;
            });

            return orderId;
        })));

        List<String> failOrderNumber =
futureReturn.stream().map(CompletableFuture::join).collect(Collectors.toL
ist());

        return failOrderNumber;
    }
}
```

代码有点不好理解直接看流程图



1. 用户发起撤销
2. 订单方法开启事物

1. 执行校验方法
2. 查询订单明细中所有需要撤销的库存id
3. 调用库存撤销方法
3. 库存撤销方法执行

1. 库存撤销方法开启事物
2. 使用库存id加锁
3. 执行撤销操作
4. 释放库存id锁
5. 库存撤销方法提交事物
4. 调用财务服务撤销订单财务相关信息
5. 修改订单状态，提交事物

6. 撤销完成

正常单个订单流程非常的正常，没有问题，那么问题出现在哪呢?

答：批量撤销

聪明的同学可能已经想到了，问题就出现在批量撤销时候因为各个单据相互之间失败不能互相影响，所以是针对每个订单单独开一个线程来撤销的，线程全部执行完后统一返回执行失败的单号和原因。

而在多线程并发执行的情况下，由于调用库存方法由库存开启的事物因为spring事物传播机制的影响加入了由订单开启的事物中，但是调用库存方法的内部在finally中已经释放掉了锁，从而如果订单A和订单B操作存在同一个库存时，由于订单A还没有撤销完成，订单B就拿到了该库存锁开始操作库存了，但是查询出来的库存行数据是订单A操作完成之前的，故引起撤销异常。

问题解决

既然已经找到问题所在，那么处理起来就比较简单了

修改调用顺序

第一个想到投机取巧的方法就是把调用库存的方法放在最后面，这样释放库存锁不就在最后面了。

但是这个方法很快被否了，因为调用撤销财务信息的方法是通过feign去调用的一个外部服务，为了避免一部分的分布式事务的问题要才放在最后面，所以这样搞是不行滴。

事物延迟提交

那只有想办法把库存方法加的锁延迟到订单撤销事物提交之后这种方式了。我们所遇到的问题，在spring设计初期大佬们早就已经考虑到了。他就是事物同步管理器TransactionSynchronizationManager。

所谓TransactionSynchronizationManager功能之一就可以在事务的边界执行一些自定义的操作，说人话就是在事务提交后再执行一些自定义方法。其中有一个registerSynchronization，让我们来看看他的入参TransactionSynchronization是怎么定义的。

```
public interface TransactionSynchronization extends Flushable {

    / ** Completion status in case of proper commit.  */
    int STATUS_COMMITTED = 0;

     /**  Completion status in case of proper rollback. */
    int STATUS_ROLLED_BACK = 1;

     /**  Completion status in case of heuristic mixed completion or system errors. */
    int STATUS_UNKNOWN = 2;


     /**   * 事务挂起  *  Supposed to unbind resources from TransactionSynchronizationManager if managing any.   *   @see TransactionSynchronizationManager#unbindResource  */
    default void suspend() {
    }

     /**   * 事务恢复  *  Supposed to rebind resources to TransactionSynchronizationManager if managing any.   *   @see TransactionSynchronizationManager#bindResource  */
    default void resume() {
    }

     /**   * 将基础会话刷新到数据存储区(如果适用)，比如Hibernate/JPA的Session  *   @see org.springframework.transaction.TransactionStatus#flush()   */
    @Override
    default void flush() {
    }

     /**   * 在事务提交前触发，此处若发生异常，会导致回滚。   *   @see #beforeCompletion  */
    default void beforeCommit(boolean readOnly) {
```

```
    }

    /**  * 在beforeCommit之后，commit/rollback之前执行。即使异常，也不会回滚。  *  @see #beforeCommit *  @see #afterCompletion */
    default void beforeCompletion() {
    }

    /**  * 事务提交后执行。  */
    default void afterCommit() {
    }

    /**  * 事务提交/回滚执行 */
    default void afterCompletion(int status) {
    }
```

其中映入眼帘的afterCommit、afterCompletion就算不看注释我们也能知道要找的就是他两其中之一没错了，而根据我的业务需求，我的锁需要不管是在事物提交或者回滚后释放锁，所以选择afterCompletion准没错了。

要怎么用呢，很简单，把库存释放锁放在事物同步管理器里就可以了，上代码

```
@Transactional(rollbackFor = Exception.class)
public void inventoryOperation(List<Long> inventoryIds) {

    List<Rlock> rlockList = RedisUtil.getLocks(inventoryIds);

    try {
        RedisUtil.tryLocks(rlockList);
        // ...库存操作

    }finally {
        TransactionSynchronizationManager.registerSynchronization(new TransactionSynchronization() {
            @Override
            public void afterCompletion(int status) {
                RedisUtil.unLocks(rlockList);
            }
        });
    }
}
```

小结

ok，总结一下，此次线上事故就是因为锁在事物提交之前就提交而引起的，最终是通过TransactionSynchronizationManager事物同步管理器把锁延迟到事物提交之后来释放解决了这个问题。

解决这个问题是通过封装了一个工具类来实现的，大家思考下有什么更好的实现方法吗？比如注解等等（下期素材有了嘿嘿）

而日常我们在项目中会遇到很多需要确保事物之后场景还有很多，比如推送消息、缓存更新、发邮件或者短信通知等等。这次造成这个问题还是因为在写项目时思考不充分造成的，以后在涉及操作库存、财务这种比较关键的服务还是得多加思考啊。。 不然下次可能要去找财务结工资了

原文链接: https://juejin.cn/post/7381371976030027816