DSF-685问题回溯报告

**回溯人：**王兵

**工号：**008197

**回溯时间：**2015-06-11

**JIRA：**http://jira.tuniu.org/browse/DSF-685

# 问题描述

订单系统在集群情况下一台主机会在一段时间后产生宕机现象，影响部分订单付款。

# 需求来源

此问题根源需求是PTICKET-1142 易到租车百元礼包券，JIRA地址：<http://jira.tuniu.org/browse/PTICKET-1142>。

需求概括一下，就是10万张优惠劵在出票成功时，通过短信发送给订单联系人。10万张优惠劵信息预先刷入数据库系统中，通过抢占方式获取，发完为止。

# 设计思路&开发

## 并发方案选择

资源抢占式任务，在单点多线程并发和集群并发情况下，会出现数据覆盖问题，所以在开发此需求时，决定单机多线程并发通过使用ReentrantLock解决并发问题。集群并发则在开发前一共提出如下几种解决方案：

1. 乐观锁
2. 悲观锁
3. 消息队列
4. 异步锁
5. redis

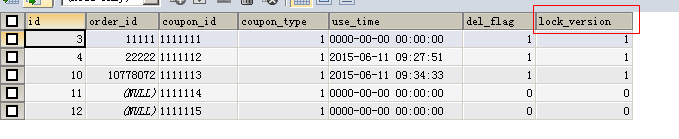
其中订单系统没有引入redis，不考虑。3，4两点工作量偏大，且实现较为复杂，开发量大过需求本身，不考虑。

剩下悲观锁和乐观锁，考虑到7，8，9旺季，如果采用悲观锁，采用锁行，肯定会引起系统并发处理量降低，系统性能降低，同时有可能影响数据库，所以最终决定采用乐观锁解决并发方式。

## 代码实现

首先在表中增加了lock\_version版本号字段，记录乐观锁版本号。然后每次并发产生时，获得更新数据库时的受影响行数（effectRows），利用受影响行数来做判断，如果更新后返回0行，则肯定是出现并发问题，此时只需要再重新查一条数据进行更新即可，如果再次发生并发返回0行，则再查一条新数据即可，直到查不到数据，或者更新成功退出（递归方法）。

### SQL



|  |
| --- |
| <select id=*"queryNotUsedThirdpartCouponByType"* resultMap=*"flightThirdpartCoupon"*>  SELECT  <include refid=*"all\_property"* />  FROM ord\_ru\_thirdpart\_coupon  WHERE coupon\_type = #{couponType,jdbcType=INTEGER}  AND del\_flag = 0  AND lock\_version = 0  ORDER BY id  LIMIT 1  </select>  <update id=*"updateThirdpartCouponStateAsUsed"*>  UPDATE  ord\_ru\_thirdpart\_coupon  SET  order\_id = #{orderId,jdbcType=INTEGER},  del\_flag = 1,  lock\_version=lock\_version + 1,  use\_time = CURRENT\_TIMESTAMP  WHERE coupon\_id = #{couponId,jdbcType=VARCHAR}  AND coupon\_type = #{couponType,jdbcType=INTEGER}  AND lock\_version = #{lockVersion,jdbcType=INTEGER}  </update> |

### 代码

|  |
| --- |
| **public** **int** yidaobaiyuanjuan4Order(Integer orderId) {//递归方法  **int** effectRow = 0;          yidaobaiyuanjuanlock.lock();//解决单节点多线程，获取锁  **try** {               FlightThirdpartCoupon queryCondition = **new** FlightThirdpartCoupon();              queryCondition.setCouponType(FlightThirdpartCouponType.YIDAOBAIYUANJUAN.getType());              queryCondition.setOrderId(orderId);               List<FlightThirdpartCoupon> alreayUsedList = queryThirdpartCoupon(queryCondition);  **if** (!alreayUsedList.isEmpty()) {                   LOGGER.info("订单" + orderId + "已经赠送过易到百元劵，劵号：" + alreayUsedList.get(0).getCouponId() + "，此订单不再赠送。");  **return** 0;              }  //此处代码当进行递归后始终不能查询到新值，一直是旧值，无法查询到新的数据               FlightThirdpartCoupon notUsedCoupon = queryNotUsedThirdpartCouponByType(queryCondition);  **if** (notUsedCoupon == **null**) {                   LOGGER.info("易到租车百元劵已经赠送完毕，订单" + orderId + "不再赠送");  **return** 0;              }               LOGGER.info("即将为订单" + orderId + "赠送易到租车百元劵，劵ID：" + notUsedCoupon.getCouponId());               notUsedCoupon.setOrderId(orderId);  //更新数据，通过受影响行数来进行是否更新成功。              effectRow = flightThirdpartCouponMapper.updateThirdpartCouponStateAsUsed(notUsedCoupon);  **if** (effectRow <= 0){                  LOGGER.info("订单" + orderId + "赠送易到租车百元劵产生多节点并发问题，劵ID：" + notUsedCoupon.getCouponId() + ",即将重新分配");                  effectRow = yidaobaiyuanjuan4Order(orderId);//更新返回0行，进行递归，重新操作，直到查询不到可用数据，或者更新成功。              }**else**{                  MemoAddVo memoAddVo = **new** MemoAddVo();                  memoAddVo.setOrderId(orderId);                  memoAddVo.setAddUid(39);                  memoAddVo.setAddTime(**new** Date());                  memoAddVo.setAddPerson("系统");                  memoAddVo.setContent("赠送易到租车百元劵：" + notUsedCoupon.getCouponId());                  memoService.add(memoAddVo);              }          } **catch** (Exception e) {              LOGGER.error("为订单" + orderId + "赠送易到用车百元时出现异常", e);          }**finally**{              yidaobaiyuanjuanlock.unlock();//释放锁          }  **return** effectRow;      } |

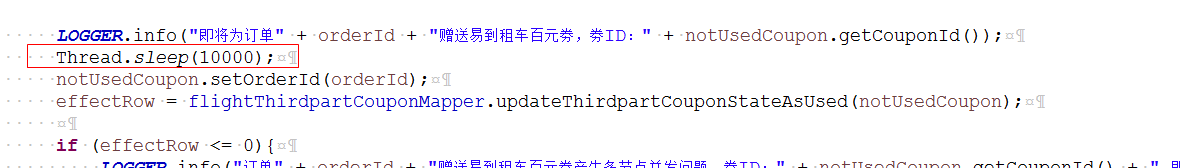
# 测试过程

## 本地自测

本地编写Junit单元测试，模拟集群并发，首先调用该方法，然后在睡眠的10秒钟内，手工去数据库更改同一条数据，观察代码运行结果。

测试代码如下：

首先在yidaobaiyuanjuan4Order(Integer orderId)方法中增加一行睡眠操作：



然后执行UT代码，并在10s内手工修改数据库：

try {

FlightTakeoutSuccessSmsDelegate aaa = (FlightTakeoutSuccessSmsDelegate)

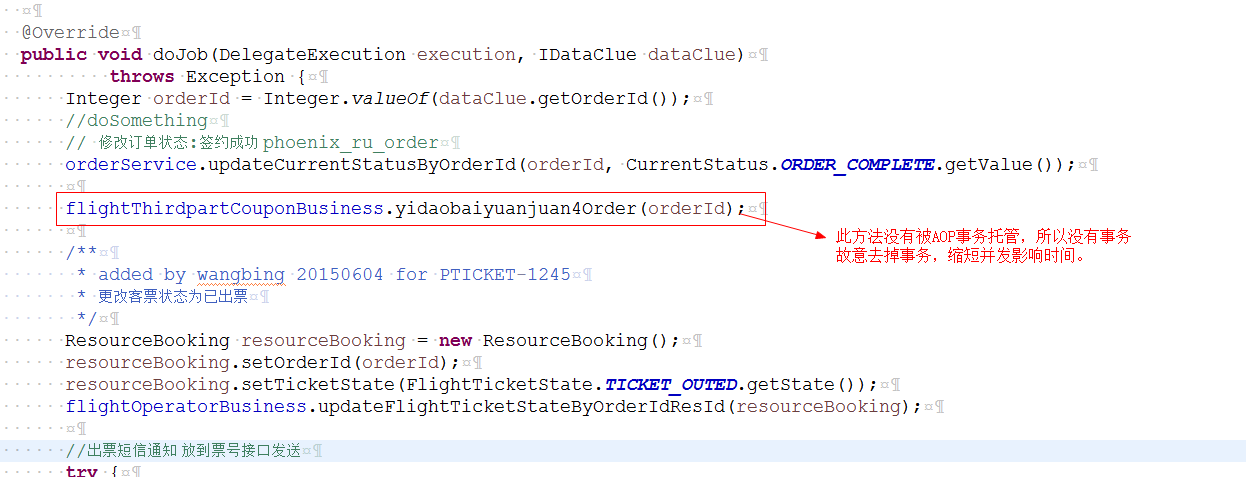
ContextLoader.getCurrentWebApplicationContext().getBean("flightTakeoutSuccessSmsDelegate");

DataClue dataClue = new DataClue();

dataClue.setOrderId("10777485");

aaa.doJob(null, dataClue);

} catch (Exception e) {}



|  |
| --- |
| <tx:advice id=*"txAdvice"* transaction-manager=*"ordTransactionManager"*>  <tx:attributes>  <tx:method name=*"get\*"* read-only=*"true"* />  <tx:method name=*"set\*"* read-only=*"true"* />  <tx:method name=*"query\*"* read-only=*"true"* />  <tx:method name=*"find\*"* read-only=*"true"* />  <tx:method name=*"load\*"* read-only=*"true"* />  <tx:method name=*"count\*"* read-only=*"true"* />  <tx:method name=*"save\*"* rollback-for=*"Exception"* />  <tx:method name=*"add\*"* rollback-for=*"Exception"* />  <tx:method name=*"update\*"* rollback-for=*"Exception"* />  <tx:method name=*"delete\*"* rollback-for=*"Exception"* />  <tx:method name=*"merage\*"* rollback-for=*"Exception"* />  <tx:method name=*"create\*"* rollback-for=*"Exception"* />  <tx:method name=*"cancel\*"* rollback-for=*"Exception"* />  <tx:method name=*"remove\*"* rollback-for=*"Exception"* />  <tx:method name=*"change\*"* rollback-for=*"Exception"* />  </tx:attributes>  </tx:advice>  <aop:config>  <aop:pointcut id=*"busiOperation"*  expression=*"execution(\* com.tuniu..business..\*Business.\*(..)) and !within(com.tuniu.css.order.manage.train..\*Business) and !within(com.tuniu.css.order.manage.nrhelper..\*Business)"* />  <aop:advisor advice-ref=*"txAdvice"* pointcut-ref=*"busiOperation"* />  </aop:config> |

## 测试环境测试

由于测试环境不存在集群，单机通过加锁已经避免了多线程并发问题，所以集群情况下只能采用模拟方式测试，测试方法同本地相同，通过Rest接口，编写测试代码进行测试。

## 测试结果

本地和测试环境通过测试代码运行正常，测试结果符合预期的结果。

# 根因分析

通过排查后发现，yidaobaiyuanjuan4Order方法本身没有问题，自测时通过Junit或者Rest接口，使用ContextLoader获取bean对象来进行代码调用和预期是完全一样。且在PC和95测试环境均测试通过。

但是，根本问题就出现在调用方式上，订单系统中使用了Activiti流程引擎，而此方法正是被Activiti拉起的流程代码所调用，当被Activiti调用时，即使是多次查询，下面的代码永远无法查询到新的数据（即使数据库数据已经被更新）。

FlightThirdpartCoupon notUsedCoupon = queryNotUsedThirdpartCouponByType(queryCondition);

目前，已经确认是由Activiti流程引擎引起，但目前还不清楚为什么Activiti会导致此问题。

# 解决方案

1. 去掉代码中的递归操作，如果发现出现并发问题，则此订单不再发送。
2. 后续涉及Activiti流程代码时，需要结合Activiti流程引擎测试。

# 改进措施

此次问题导致服务器宕机，影响较为严重，产生该问题主要因素是没有有效的模拟集群并进行端到端测试。后续将在单元测试完成后，再进行E2E的测试方案评审，然后完整进行E2E模拟和测试。