用户中心1.0技术经验总结

# 分布式锁和事务

## 问题背景

采用第三方支付订单失败时，正常逻辑为：充值、消费、退款；但出现了充值、消费、退款、充值、消费这五条记录，多出了两条记录（充值，消费）。

## 问题原因

原因是由于事务的范围大于锁的范围，在一个大事务中获取锁，由于在事务提交之前释放了锁，导致第二个线程在第一个线程的事务提交前获得了锁，使第二个线程获取到了脏数据，导致最后的数据错误。

相关代码：

com.eayun.common.zk.DistributedLockService



## 解决方案

使用zk分布式锁时，在事务结束（commit或rollback）后才释放锁。对于分布式的工具类，在方法上提供选择，默认业务结束就释放锁，也可以在事务结束才释放锁。

修改后代码：



# Redis多线程和MQ

## 问题背景

在obs扣费consumer中，每小时每个obs用户会向该consumer发送三次消息（统计存储容量job，统计下载流量请求数job，统计回源流量job），这三条消息都统计成功了，才会开始计费。但是在一次扣费中，发现这个客户的扣费执行了两次，多扣除了一次费用。

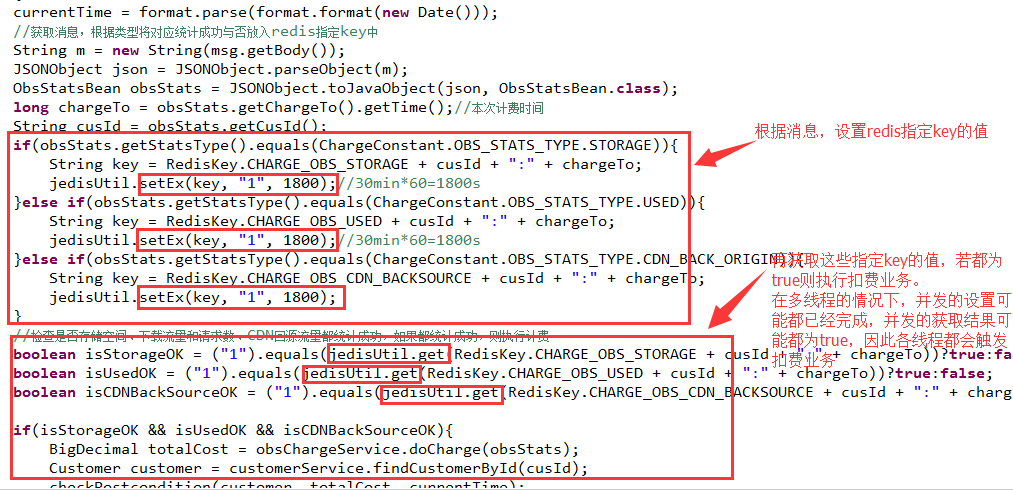
## 问题原因

这3个job发的消息，格式相同，类型不同，都由同一个consumer处理。收到消息后，判断当前的类型，给redis中对应的key置状态，之后再判断这三个状态，若都为统计成功（true），则开始执行扣费。

出现问题的地方为：在第一个统计成功后，第二个和第三个同时到来，第二个消息由机器1消费，第三个消息由机器2消费。都在置完状态之后判断三个状态值，因此都得出三个状态都是true的结论，所以出现了两次扣费。

相关代码：

com.eayun.charge.mq.ChargeObsConsumer



## 解决方案

在该consumer中，从向redis中写入该job统计成功的标识，到从redis中获取三个job统计成功的标识处，加入zk分布式锁，保证该读写操作是在锁中进行，即可避免由redis多线程导致的问题。

修改后代码：



类似场景，即需要通过多个redis读写来判断的业务，由于redis的每一个操作都是原子的，因此没有整体的事务，需要考虑多线程并发的情况下是否有问题，是否能用加锁解决需要根据当时实际业务决定。

# 金额操作

## 问题背景

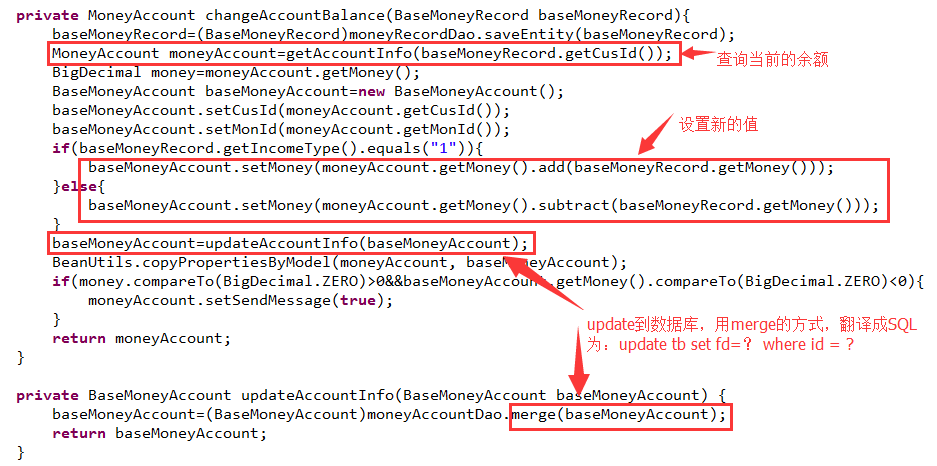
采用第三方支付订单失败时，正常逻辑为：充值、消费、退款；但出现了充值、消费、退款、充值、消费这五条记录，多出了两条记录（充值，消费），同时在多出的充值记录中，出现了金额不准的现象。

## 问题原因

在修改账户金额的接口中，是从子表（交易记录表）中insert记录，在主表（账户余额表）中采用的是先find余额，再update余额+变量值，由于调用修改账户金额接口的地方都采用的释放锁的方式为直接释放，所以如果释放锁后事务还未提交，下次修改账户金额时就会find出老数据，导致数据错误。

相关代码：

com.eayun.costcenter.service.impl.AccountOverviewServiceImpl



## 解决方案

修改主表的操作由之前的先find后update，改为直接update balance+money，即不再手动查询余额，直接修改。

修改后代码：



# TransactionHookUtil应用场景

## 应用场景

当创建云主机时，云主机调用订单，生成订单后，订单写入消息队列，云主机读取消息队列实际创建云主机。

## 产生问题

当订单写入消息队列，但未提交订单时，订单已将消息放入消息队列，当云主机读取消息后，查询订单表中的记录，订单表中并没有此记录，则会产生错误

## 代码应用

|  |
| --- |
| TransactionHookUtil.*registAfterCommitHook*(**new** Hook() {  @Override  **publicvoid** execute() {  **if** (order.getOrderType().equals(OrderType.*NEW*)  || order.getOrderType().equals(OrderType.*UPGRADE*)) {  *log*.info("发送新购或升级资源的消息队列。");  orderToResourceProducer.sendToNewOrUpgrade(order.getOrderNo(), order.getOrderType(),  order.getResourceType());  } **elseif** (order.getOrderType().equals(OrderType.*RENEW*)) {  *log*.info("发送续费资源的消息队列。");  SessionUserInfo sessionUser = (SessionUserInfo) SessionUtil.*getSession*()  .getAttribute(ConstantClazz.*SYS\_SESSION\_USERINFO*);  orderToResourceProducer.sendToRenewal(order.getOrderNo(), order.getCusId(),  sessionUser.getUserName(), order.getParams());  }  }  }); |

|  |
| --- |
| **publicstaticvoid** registAfterCommitHook(**final** Hook hook) {  TransactionSynchronizationManager.*registerSynchronization*(**new** TransactionSynchronizationAdapter(){  @Override  **publicvoid** afterCommit() {  hook.execute();  }  });  }  **publicinterface** Hook {  **void** execute();  } |

# 特殊事务处理逻辑，不回滚机制

## 应用场景

购买云主机，当订单传递消息给云主机后，云主机创建时调用eayunstack包中底层信息，完成订单信息。

## 产生问题

当云主机调用底层返回错误时，云主机会修改订单状态为失败已取消，系统将返回的错误抛出AppException，这个exception继承了RuntimeException，导致事物回滚，则云主机修改的订单状态也变更回原来的状态。需要将订单状态不回滚。

## 代码应用

|  |
| --- |
| @Transactional(noRollbackFor=AppException.**class**)  **publicvoid** onMessage(Message msg,Channel channel) **throws** Exception{  // to do  } |

# findone等实体泛型操作

## 应用场景

用dao执行findone操作时，获取实体，直接操作实体，即可更新数据库，无需后面saveorupdate

## 产生问题

当中间有报错时，最后虽然没有saveorupdate，也会执行入库，所以一定要注意。

## 代码应用

|  |
| --- |
| BaseCloudRoute baseRoute = routeDao.findOne(routeId);  **if**(rate==baseRoute.getRate()){  **return**;  }  **if**(!StringUtils.*isEmpty*(baseRoute.getQosId())){  **try**{  eayunQosService.changeQos(baseRoute.getDcId(), baseRoute.getQosId(), rate);  }**catch**(Exception e){  **throw** e;  }  }  baseRoute.setRate(rate);  routeDao.saveOrUpdate(baseRoute); |

# 重名校验问题总结，sql中用正则表达式

## 应用场景

对于重名校验的问题，需要实时校验是否重名，如果有批量存在时，会有大量的sql执行，而且目前执行的sql判断语句是拼写一长串的名称，然后用in，将数据查出来判断。

## 产生问题

这种sql会比较低效，而且很耗内存，所以可以使用正则表达式，并且只查count即可，甚至只需要一个就可以证明已存在，无需全部查出。

## 代码应用

|  |
| --- |
| /\*\*  \* 验证重名  \*/  @Override  **publicboolean** getVolumeByName(CloudVolume vol)**throws** Exception {  **boolean** flag = **false**;  **int** num=1;  String[] nameList = **null**;  **if**(vol.getVolNumber()>0){  num = vol.getVolNumber();  **if**(vol.getVolNumber()>20){  num=20;  }  }    **if** (num != 0) {  nameList = **new** String[num];  }  **if** (num == 1) {  nameList[0] = vol.getVolName();  } **elseif** (num > 1) {  **for** (**int** i = 1; i <= num; i++) {  **if**(i>=10){  nameList[i-1] = vol.getVolName() + "\_" + i;  }**else**{  nameList[i-1]=vol.getVolName() +"\_0"+i;  }    }  }    **try**{  StringBuffer hql = **new** StringBuffer();  hql.append(" select count(\*) from ");  hql.append(" BaseCloudVolume v where 1=1 ");  hql.append(" and v.prjId = :prjId ");  hql.append(" and v.isDeleted in ('0','2') ");  hql.append(" and binary(v.volName) in (:names)");  **if** (!StringUtils.*isEmpty*(vol.getVolId())) {  hql.append(" and v.volId <> :volId");  }  org.hibernate.Query query = volumeDao.getHibernateSession().createQuery(hql.toString());  query.setParameter("prjId", vol.getPrjId());  query.setParameterList("names", nameList);  **if** (!StringUtils.*isEmpty*(vol.getVolId())) {  query.setParameter("volId", vol.getVolId());  }    **int** count = Integer.*parseInt*(String.*valueOf*(query.uniqueResult()));  flag = count == 0;    **if**(flag){  StringBuffer orderVolHql = **new** StringBuffer();  orderVolHql.append(" SELECT ");  orderVolHql.append(" cov.vol\_number, ");  orderVolHql.append(" cov.vol\_name ");  orderVolHql.append(" FROM ");  orderVolHql.append(" cloudorder\_volume cov ");  orderVolHql.append(" LEFT JOIN order\_info oi ON oi.order\_no = cov.order\_no ");  orderVolHql.append(" WHERE ");  orderVolHql.append(" binary(cov.vol\_name) = ? ");  orderVolHql.append(" AND cov.order\_type = '0' ");  orderVolHql.append(" AND oi.order\_state in ('1','2') ");  orderVolHql.append(" AND cov.prj\_id=? ");  javax.persistence.Query orderVolQuery = volumeDao.createSQLNativeQuery(orderVolHql.toString(),**new** Object[]{vol.getVolName(),vol.getPrjId()});  @SuppressWarnings("rawtypes")  List orderVolList = orderVolQuery.getResultList();  **for**(**int** i = 0;i<orderVolList.size();i++){  Object [] obj = (Object [])orderVolList.get(i);  **int** volNumber = Integer.*parseInt*(String.*valueOf*(obj[0]));  String volName = String.*valueOf*(obj[1]);  **if**(volNumber==1&&vol.getVolNumber()==1&&vol.getVolName().equals(volName)){  flag = **false**;  **break**;  }  **if**(volNumber>1&&vol.getVolNumber()>1&&vol.getVolName().equals(volName)){  flag = **false**;  **break**;  }  }  }  }**catch**(Exception e){  *log*.error(e.getMessage(),e);  **throw** e;  }    **return** flag;  } |

# BigDecimal泛型应用以及修改赋值问题

## 应用场景

现在系统中的钱相关的数值都用BigDecimal

## 产生问题

BigDecimal在设置数值或取位舍位时，都必须重新赋值为新的变量，设置后并不会直接改变原对象的内容。

## 代码应用

|  |
| --- |
| **private** BigDecimal calcVolumePrice(CloudOrderVolume orderVolume){  BigDecimal money=**new** BigDecimal(0);  **if**(OrderType.*NEW*.equals(orderVolume.getOrderType())){  ParamBean paramBean = **new** ParamBean();  paramBean.setDcId(orderVolume.getDcId());  paramBean.setPayType(orderVolume.getPayType());  paramBean.setNumber(orderVolume.getVolNumber());  paramBean.setCycleCount(orderVolume.getBuyCycle());  paramBean.setDataDiskCapacity(orderVolume.getVolSize());  money= billingFactorService.getPriceByFactor(paramBean);  }**elseif**(OrderType.*UPGRADE*.equals(orderVolume.getOrderType())){  UpgradeBean upgradeBean=**new** UpgradeBean();  upgradeBean.setDcId(orderVolume.getDcId());  upgradeBean.setDataDiskCapacity(orderVolume.getVolSize()-orderVolume.getVolOldSize());  upgradeBean.setCycleCount(DateUtil.*getUgradeRemainDays*(**new** Date(), orderVolume.getEndTime()));  money=billingFactorService.updateConfigPrice(upgradeBean);  }    **return** money.setScale(2, RoundingMode.*FLOOR*);    } |

# 死循环处理

## 应用场景

在循环判断时

## 产生问题

如果没有明确的退出条件时，或者退出条件可能受到阻塞制约时，则中间需要有停顿，否则会一直占用CPU。

## 代码应用

|  |
| --- |
| **while**(**true**){  **for**(Future f: **new** ArrayList<>(futureList)){  **if**(f.isDone()){  //isDone返回true的场景：正常结束、异常、线程被取消  futureList.remove(f);  }  }  **if**(futureList.isEmpty()){  **break**;  }  **try** {  Thread.*sleep*(50);//sleep a while，give cpu a break  } **catch** (InterruptedException e) {  *log*.error("线程休眠50ms异常",e);  }  } |

# http请求的远程调用

## 应用场景

当我们调用底层时，使用的http请求，并且synchronized整个方法。

## 产生问题

调用时，并没有设置超时以及其他断开连接的机制，则偶尔会产生底层没有返回现象，则会导致连接一直被占用，使得无法执行下一步操作，所有连接在等待。

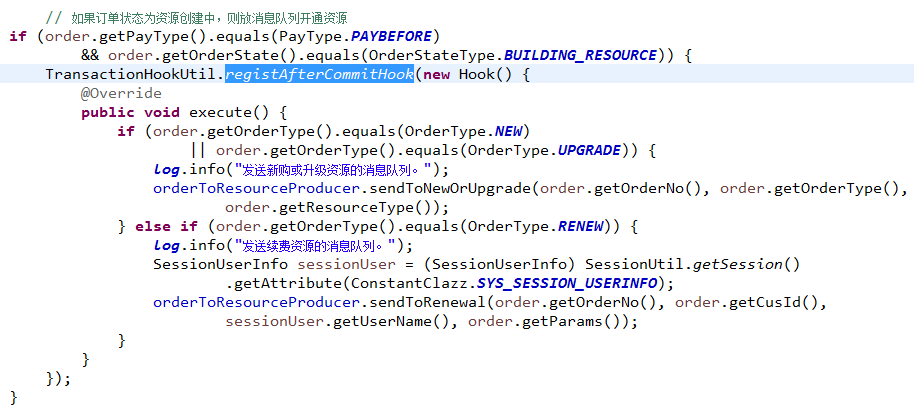
## 代码应用

|  |
| --- |
| **private** JSONObject getToken(JSONObject json, String url) **throws** AppException {  HttpClient httpclient = HttpClients.createDefault();  HttpPost post = **new** HttpPost(url + "/tokens");  //设置请求超时  RequestConfig requestConfig = RequestConfig.custom().setSocketTimeout(60000).setConnectTimeout(60000).build();  post.setConfig(requestConfig);  log.info("The current URL is:" + url);  log.info("The requestbody is:" + json);  JSONObject resJson = **null**;  String resData = **null**;  **int** response = 0;  log.info("url=" + url);  log.info("paras=" + json);  **return** resJson;  } |

# 事务Hook

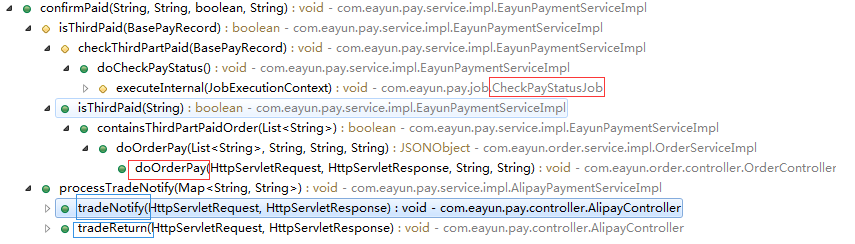
背景：订单模块在处理完成订单数据之后，发送mq消息，通知资源模块做后续处理。资源模块收到mq消息，读取订单数据，发现订单数据有可能还未提交到数据库。之前的解决方案是在资源模块延迟对订单数据的读取，但到底延迟多久去读才能读到提交后的数据也是不可知的。目前使用的方案为：在订单模块延迟发送mq消息，保存在订单数据提交到数据库之后才发mq。所以引入了事务Hook工具类。

使用场景：主线程里有对数据库做变更，触发（不限于使用mq方式）另一个线程执行，另一个线程依赖于主线程数据。



# 第三方支付结果处理

背景：第三方支付由于需要与外部系统对接，支付结果需要由外部系统主动通知我司。不排除因为各种原因有未能通知到的情况。所以这块在设计时我们增加了主动查询触发场景。



如图：蓝色的接口，为支付宝主动发起的支付结果回调通知。

如果因为咱们服务停用，支付结果是接收不到的，如果没有其它处理方式，会产生很严重问题，比如：客户余额不能冲上，资源不能创建等等。

红色的是我们主动查询支付结果的场景：

CheckPayStatusJob，在定时查询状态为支付中的订单支付结果。

doOrderPay接口，在订单支付时，为防止重复支付，在本地数据库不能判断到底支付完成时，需要查询支付宝系统中的最新数据。

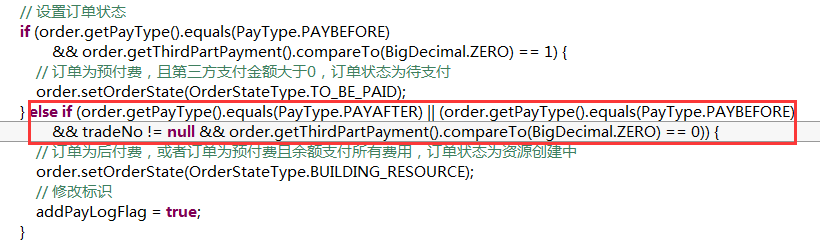
有了以上主动查询场景，在遗漏支付宝通知时，问题也不大，只是数据会稍微延后。

由于支付结果确认入口有多个，并且会在不同的进程中执行，所以使用了zk的分布式锁。并段时间因为分布式锁在事务提交之前释放，导致支付确认逻辑执行多次。引发了一个订单支付后费用中心多出交易记录、创建多个相同资源问题。

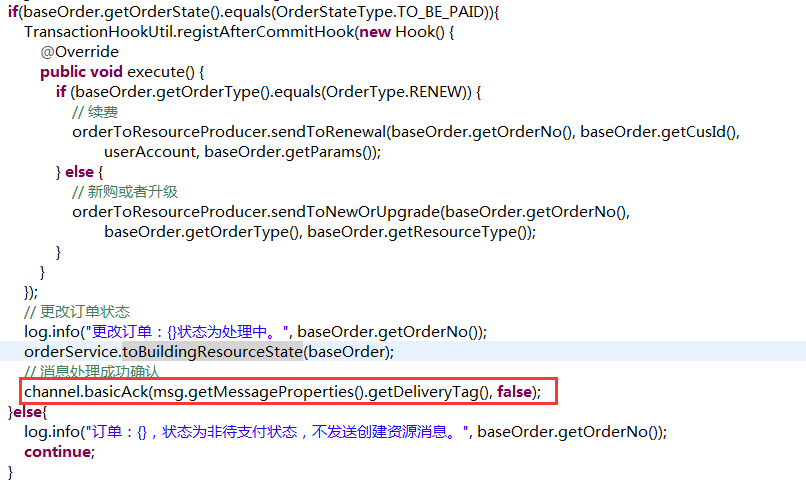
# 代码逻辑不够完善

在开发过程中，各种场景考虑不够完善，

* 当订单为预付费，且第三方支付金额为零的情况，不会给资源发送消息；



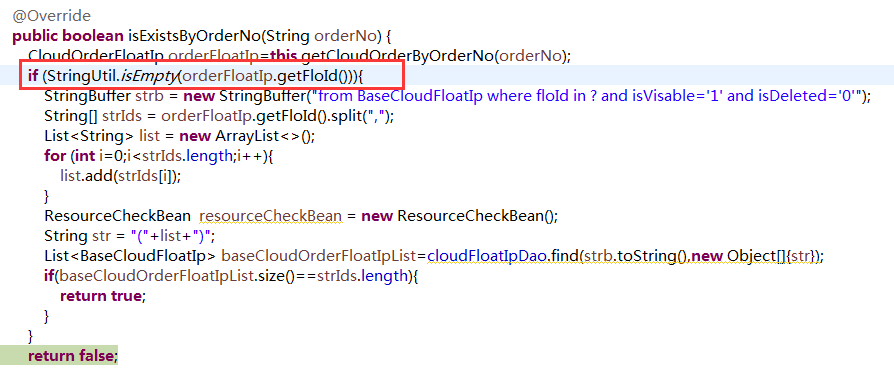
* 消费者消费消息时，逻辑判断if-else，只在if中反馈给消息中间件成功信息，造成消息被多次消费。



# 订单通知资源两次问题处理

用户中心版本中的订单开发，和上层资源、第三方支付交互较多，暂时没有出现重大BUG，在开发过程中，与其他同事的沟通还是比较到位的，出现问题，能第一时间配合处理，核实原因，比如：

第三方支付时校验，页面报错，提示资源不存在，核实代码后发现，云主机判断资源是否存在有逻辑问题；



订单存在两次交易记录，创建两次资源问题。代码同1.2的第二点。

# 防火墙代码增加注解，增加代码可读性



在代码注释方面感觉要好一点。涉及以前代码时没有注释是一件很头疼的事

# 防火墙删除的坑



在操作底层时，不要想当然根据底层逻辑走，有可能网络延迟，底层的上个逻辑还没走完会造成不易查出的问题。在这次防火墙删除时刚开始直接删除了防火墙就删除相应的策略和规则，可是底层删除防火墙需要时间，由于成都网络原因，在本地开始测试基本成功。后来测试连接外网环境才发现这一问题。

# 尽量减少脏数据（防火墙创建）



这是一个教训，虽然底层不能回滚但是代码还是要尽可能回滚，尽量减少底层脏数据

# JavaScript打开新的标签页

　　在调试支付宝网页支付的时候，开始的设计异步获取支付宝的支付表单，用js传递表单文本到新的标签页发起支》付。调试过程中记录了如下两种方式   
方式一：

1. var rechargeWindow = window.open('about:blank','blank');
2. $http.post('pay/recharge/rechargeBalance.do', $scope.model).then(function(response){
3. //在新标签页输出表单
4. rechargeWindow.document.open();
5. rechargeWindow.document.write(response.data);
6. rechargeWindow.document.close();
7. rechargeWindow.focus();
8. });

方式二：

1. $http.post('pay/recharge/rechargeBalance.do', $scope.model).then(function(response){
2. encodeURIComponent(response.data);
3. var payWindow = window.open("data:text/html,"+ response.data,"\_blank");
4. payWindow.focus();
5. });

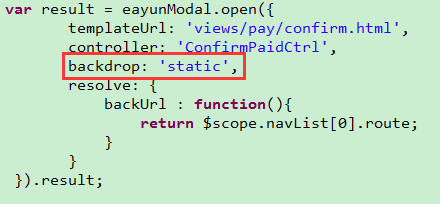
虽然最终因为种种原因，采用了POST提交数据获取支付表单，以form的target属性直接打开新标签。

# 一个完美的验证金额的正则表达式

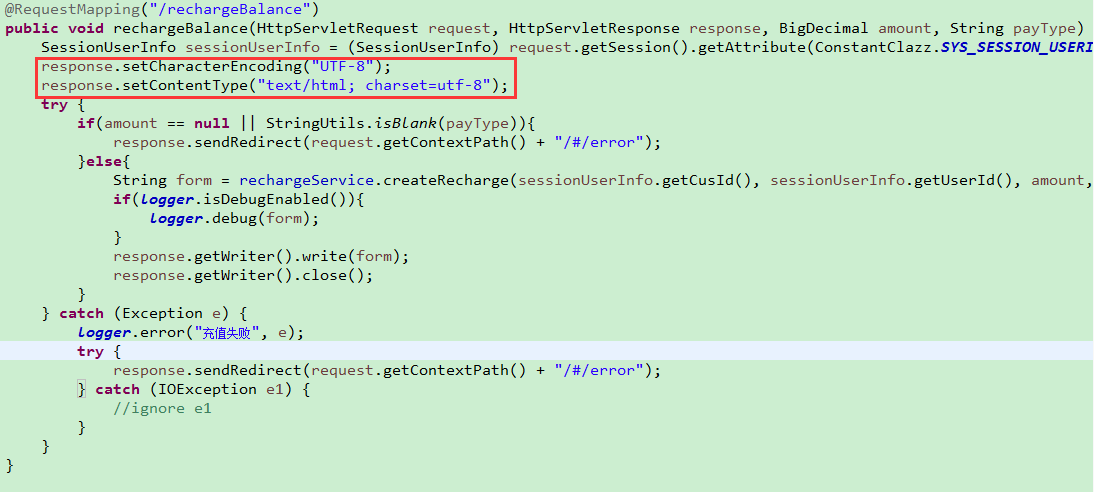
保留2位小数，金额不超过1亿

1. /^((0.0[1-9])|(0.[1-9][0-9]?)|([1-9][0-9]\*(.\d{1,2})?))$/

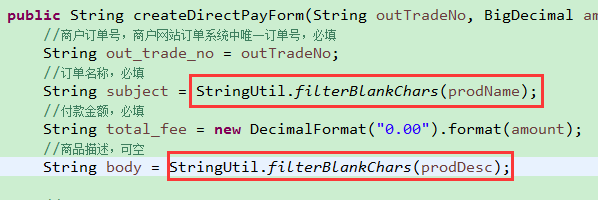
# AngularJS打开新窗口，点击背景不允许关闭窗口

设置模板参数backdrop为static   


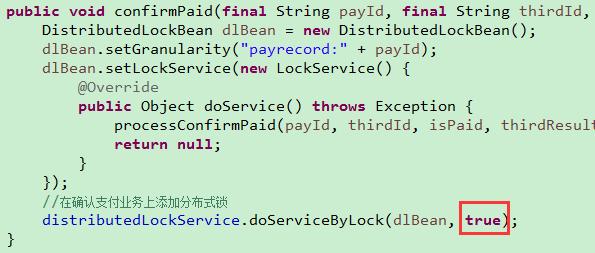
# 调用充值接口，response编码

在获取订单支付表单文本的时候有给response设置UTF-8编码，充值接口忘记设置编码，由于文本中包含中文，导致发起支付失败

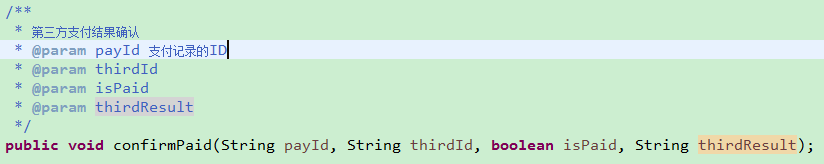
# 订单名称、商品描述包含特殊字符导致签名失败

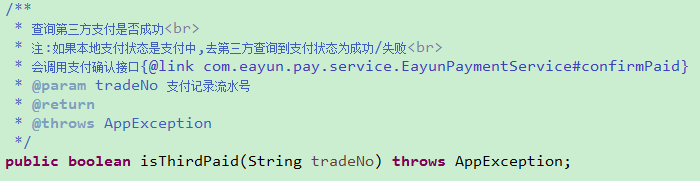
根据以往的开发经验，支付宝在服务端校验签名之前会过滤掉卖家、买家名称中的制表符，而此次开发中不涉及手动输入卖家和买家信息，所以未作相关过滤处理。然而实际调试过程中发现，订单名称和商品描述中包含制表符也会导致签名失败:   


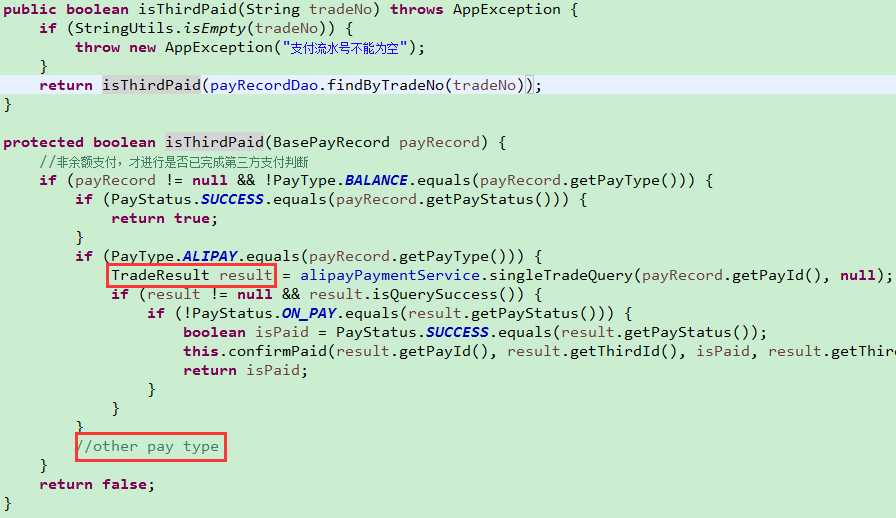
# 支付确认，并发问题

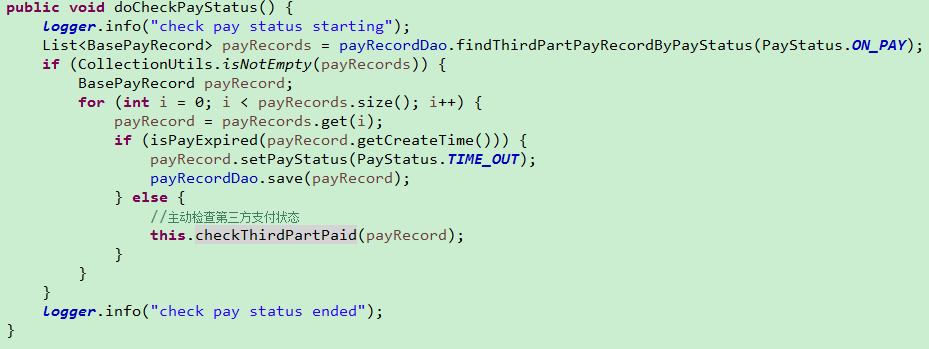
未考虑到分布式锁与事务同步问题，使得锁的释放先于事务提交，从而导致存在重复确认支付，进而产生重复的交易记录及重复通知订单，导致创建资源。   
解决方案：   
将锁的释放强制调整到事务完成后(无论事务提交还是回滚)   


# 支付模块较高程度的抽象

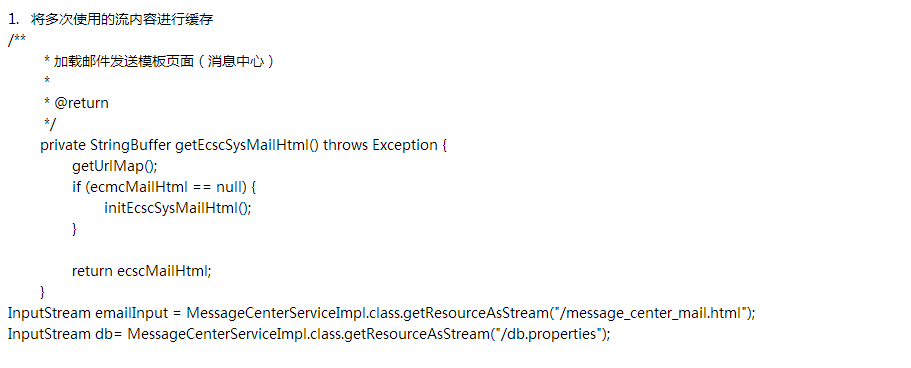
在编码设计时，针对支付模块进行了较大程度的抽象，比如第三方支付结果查询、第三方支付确认等方法做的比较通用，后期如果需要扩展其他的第三方支付，只需要加入相应的对接第三方支付服务商的接口，不需要对当前代码做太多的修改。   
第三方支付确认接口，抽象出了thirdId、isPaid、thirdResult等字段，如引入微信支付，则只需要在TenpayService中的支付回调接口直接调用EayunPaymentService.confirmPaid(…)即可   


需要查询第三方支付状态的地方任只需要调用原接口，不需作任何修改，   


只需要接入其他第三方支付的人员在isThirdPaid(BasePayRecord payRecord)方法中添加相应的第三方交易查询代码。单笔交易查询结果定义了统一的返回结果实体类TradeResult，所以甚至可以再抽象一个类出来AbstractThirdPartPayService，由AbstractThirdPartPayService统一提供singleTradeQuery(…)方法   


而对于第三方支付交易结果查询的定时任务，则不需要做任何的修改   


# 缓存使用

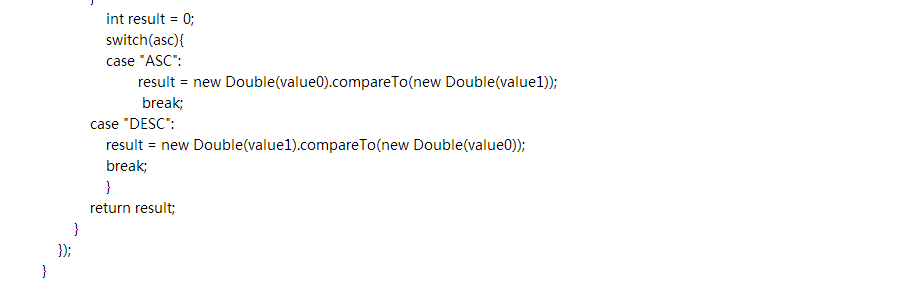


总结：该内容数据不变，使用缓存进行保存，避免多次使用IO流，影响系统性能

# 尽量减少数据库请求次数



# 内存排序



总结：在数据库未能进行排序的情况下，可以对数据里的内容进行排序， 可以根据不同的条件，然后在进行其他业务

# Hibernate缓存

## 问题背景

在多线程中,使用了分布式锁进行修改账户金额,在线程一执行完后,释放了锁(事务已提交),线程二获取锁成功,获取的余额应为新的余额,但出现了获取到的余额仍为老的数据的情况.

## 问题原因

在线程二获取账户余额时,应该获取到的是新的值,但是可能会出现获取到旧的值(事务已提交成功),在查找hibernate方法中,发现有refresh的方法,即对缓存进行刷新,保证获取到的是提交后最新的值.

## 解决方案

在获取到余额时,执行一下hibernate的refresh方法,保证得到的是提交后的最新值.

