谷粒商城

版本：V 1.0

延迟队列与轮询



**当前的位置**

# 分布式事务的异步通信问题

使用分布式事务异步通信的结构，一个很大的问题就是**不确定性**。一个消息发送过去了，不管结果如何发送端都不会原地等待接收端。直到接收端再推送回来回执消息，发送端才直到结果。但是也有可能发送端消息发送后，石沉大海，杳无音信。这时候就需要一种机制能够**对这种不确定性进行补充**。

比如你给有很多笔友，平时写信一去一回，但是有时候会遇到迟迟没有回信的情况。那么针对这种偶尔出现的情况，你可以选择两种策略。一种方案是你发信的时候用定个闹钟，设定1天以后去问一下对方收没收到信。另一种方案就是每天夜里定个时间查看一下所有发过信但是已经一天没收到回复的信。然后挨个打个电话问一下。

第一种策略就是实现起来就是延迟队列，第二种策略就是定时轮询扫描。

二者的区别是**延迟队列**更加精准，但是如果周期太长，任务留在延迟队列中时间的就会非常长，会把队列变得冗长。

那么如果遇到这种长周期的事件，而且并不需要精确到分秒级的事件，可以利用**定时扫描**来实现，尤其是比较消耗性能的大范围扫描，可以安排到夜间执行。

# 二、延迟队列

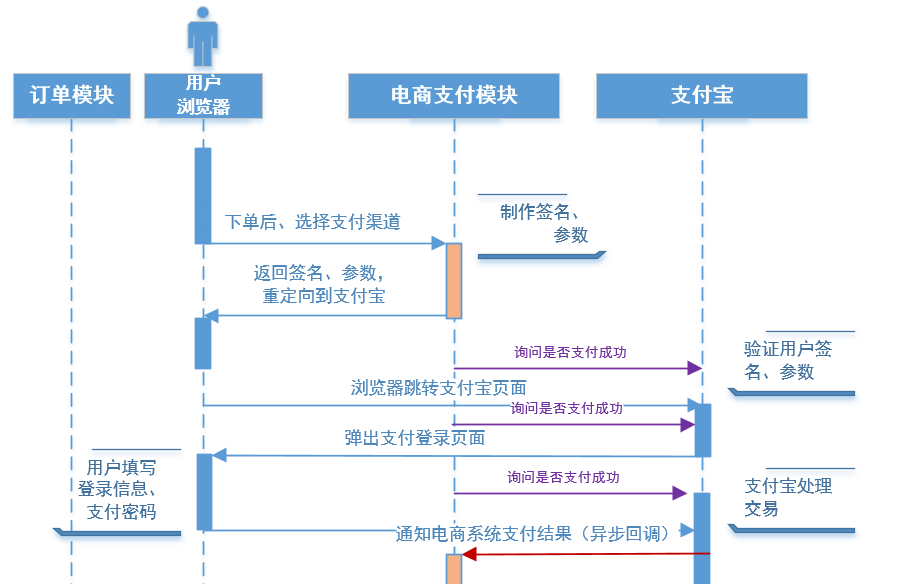
## 1、应用场景

当用户选择支付后，通常来说用户都会在支付宝正常支付，支付宝转账成功后，通过后台异步发送成功的请求到电商支付模块。

但是如果用户点击支付后，支付模块可能会长时间没有收到支付宝的支付成功通知。这种情况会‘有两种可能性，**一种是用户在弹出支付宝付款界面时没有继续支付**，另一种就是用户支付成功了，**但是因为网络等各种问题，支付模块没有收到通知**。

如果是上述第二种可能性，对于用户来说体验是非常糟糕的，甚至会怀疑平台的诚信。

所以为了尽可能避免第二种情况，在用户点击支付后一段时间后，不管用户是否付款，都要去**主动询问支付宝**，该笔单据是否付款。



图中紫线部分，就是支付模块一旦帮助用户重定向到支付宝后，就要每隔一段时间询问支付宝用户是否支付成功，直到收到支付宝的回复，或者超过了询问次数。

## 2、实现思路

首先，需要知道如何主动查询支付宝中某笔交易的状态。

支付宝查询接口文档：https://docs.open.alipay.com/api\_1/alipay.trade.query

其次，利用延迟队列反复调用。

## 3、实现支付宝订单状态查询

支付宝文档中的样例



out\_trade\_no ：第三方交易编号

Trade\_no ：支付宝回执的编号，由于没有外网我们得不到支付宝回执编号

所以在测试的时候只用out\_trade\_no.

1、首先通过基本参数初始化AlipayClient,此处和支付模块部分相同，不再详述。

2、业务参数



业务参数就两个，选哪个都可以，其中out\_trade\_no是电商系统生成的，trade\_no是支付宝回调后产生的。因为有可能一直就没收到支付宝的回调，也就没有trade\_no，所以咱们这里使用out\_trade\_no。

在支付模块的接口实现类中添加方法

接口中定义方法

|  |
| --- |
| PaymentService  **boolean** checkPayment(PaymentInfo paymentInfoQuery); |

实现类：

paymentServiceImpl

|  |
| --- |
| *// 查询支付是否成功* **public boolean** checkPayment(PaymentInfo paymentInfoQuery) {  *// 查询当前的支付信息* PaymentInfo paymentInfo = getpaymentInfo(paymentInfoQuery);  **if** (paymentInfo.getPaymentStatus()== PaymentStatus.***PAID*** || paymentInfo.getPaymentStatus()==PaymentStatus.***ClOSED***){  **return true**;  }  AlipayTradeQueryRequest request = **new** AlipayTradeQueryRequest();  request.setBizContent(**"{"** +  **"\"out\_trade\_no\":\""**+paymentInfo.getOutTradeNo()+**"\""** +  **" }"**);  AlipayTradeQueryResponse response = **null**;  **try** {  response = **alipayClient**.execute(request);  } **catch** (AlipayApiException e) {  e.printStackTrace();  }  **if**(response.isSuccess()){  **if** (**"TRADE\_SUCCESS"**.equals(response.getTradeStatus())||**"TRADE\_FINISHED"**.equals(response.getTradeStatus())){  *// IPAD* System.***out***.println(**"支付成功"**);  *// 改支付状态* PaymentInfo paymentInfoUpd = **new** PaymentInfo();  paymentInfoUpd.setPaymentStatus(PaymentStatus.***PAID***);  updatePaymentInfo(paymentInfo.getOutTradeNo(),paymentInfoUpd);  sendPaymentResult(paymentInfo,**"success"**);  **return true**;  }**else** {  System.***out***.println(**"支付失败"**);  **return false**;  }   } **else** {  System.***out***.println(**"支付失败"**);  **return false**;  } }  @Override **public void** updatePaymentInfo(String out\_trade\_no, PaymentInfo paymentInfoUpd) {  *// 创建Example对象* Example example = **new** Example(PaymentInfo.**class**);  *// 修改对象 -- 写实体类的属性名* example.createCriteria().andEqualTo(**"outTradeNo"**,out\_trade\_no);  *// 执行方法* **paymentInfoMapper**.updateByExampleSelective(paymentInfoUpd,example); } |

控制器

|  |
| --- |
| *// 查询订单信息* @RequestMapping(**"queryPaymentResult"**) @ResponseBody **public** String queryPaymentResult(HttpServletRequest request){  String orderId = request.getParameter(**"orderId"**);  PaymentInfo paymentInfoQuery = **new** PaymentInfo();  paymentInfoQuery.setOrderId(orderId);  **boolean** flag = **paymentService**.checkPayment(paymentInfoQuery);  **return ""**+flag; } |

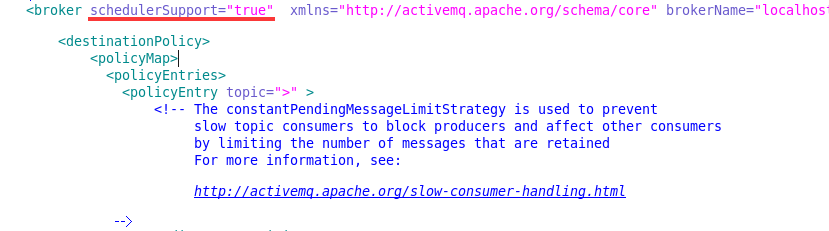
## 4、利用延迟队列反复调用查询接口。

执行策略：

选择支付渠道后，点击支付后提交到延迟队列，每隔一分钟执行一次查询操作，查询三次。

首先在消息队列中打开延迟队列配置：在activemq的conf目录下activemq.xml中

# vim /opt/activemq/conf/activemq.xml



开启 schedulerSupport="true"

配置完成，要重启

#service activemq restart

发送延迟队列

|  |
| --- |
| **接口**  **public void** sendDelayPaymentResult(String outTradeNo,**int** delaySec ,**int** checkCount) |
| **实现类**  */\*\*  \* 延迟队列反复调用  \** ***@param outTradeNo*** *单号  \** ***@param delaySec*** *延迟秒  \** ***@param checkCount*** *几次  \*/* **public void** sendDelayPaymentResult(String outTradeNo,**int** delaySec ,**int** checkCount){  Connection connection = **activeMQUtil**.getConnection();  **try** {  connection.start();  Session session = connection.createSession(**true**, Session.***SESSION\_TRANSACTED***);  *// 创建队列* Queue paymentResultQueue = session.createQueue(**"PAYMENT\_RESULT\_CHECK\_QUEUE"**);  MessageProducer producer = session.createProducer(paymentResultQueue);  MapMessage mapMessage = **new** ActiveMQMapMessage();  mapMessage.setString(**"outTradeNo"**,outTradeNo);  mapMessage.setInt(**"delaySec"**,delaySec);  mapMessage.setInt(**"checkCount"**,checkCount);  *// 设置延迟多少时间* **mapMessage.setLongProperty(ScheduledMessage.*AMQ\_SCHEDULED\_DELAY*,delaySec\*1000);**  producer.send(mapMessage);  session.commit();  producer.close();  session.close();  connection.close();  } **catch** (JMSException e) {  e.printStackTrace();  } } |

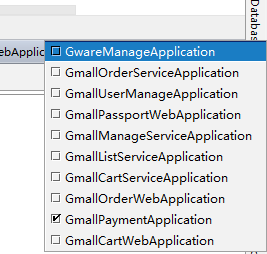
在payment项目中添加接收延迟队列的消费端

|  |
| --- |
| @Component **public class** PaymentConsumer {   @Reference  **private** PaymentService **paymentService**;   @JmsListener(destination = **"PAYMENT\_RESULT\_CHECK\_QUEUE"**,containerFactory = **"jmsQueueListener"**)  **public void** consumeSkuDeduct(MapMessage mapMessage) **throws** JMSException {   *// 获取消息队列中的数据* String outTradeNo = mapMessage.getString(**"outTradeNo"**);  **int** delaySec = mapMessage.getInt(**"delaySec"**);  **int** checkCount = mapMessage.getInt(**"checkCount"**);   *// 创建一个paymentInfo* PaymentInfo paymentInfo = **new** PaymentInfo();  paymentInfo.setOutTradeNo(outTradeNo);  PaymentInfo paymentInfoQuery = **paymentService**.getPaymentInfo(paymentInfo);  *// 调用 paymentService.checkPayment(paymentInfoQuery);* **boolean** flag = **paymentService**.checkPayment(paymentInfoQuery);  System.***out***.println(**"检查结果："**+flag);  **if** (!flag && checkCount!=0){  *// 还需要继续检查* System.***out***.println(**"检查的次数："**+checkCount);  **paymentService**.sendDelayPaymentResult(outTradeNo,delaySec,checkCount-1);  }  } } |

在生成支付二维码的方法下面添加检查代码

|  |
| --- |
| @RequestMapping(value = **"alipay/submit"**,method = RequestMethod.***POST***) @ResponseBody **public** String submitPayment(HttpServletRequest request, HttpServletResponse response){  **// 代码追后面** 15秒执行一次，总共需要执行3次。  **paymentService**.sendDelayPaymentResult(paymentInfo.getOutTradeNo(),15,3);  } |

测试：



# 三、轮询扫描

## 1 、应用场景

长期没有付款的订单，要定期关闭掉。

如果时限比较小，比如30分钟未付款的订单就关闭（一般是锁了库存的订单），也可以用**延时队列**解决。

如果时限比较长比如1-2天，可以选择用轮询扫描。

## 2 、实现方式 spring task

但是springboot整合了自家的spring task ，配置更简单，全程只用注解就可以，不用额外的xml。

在订单项目gmall-order-service中做测试Task

|  |
| --- |
| *在类上添加注解*  @EnableScheduling @Component  **public class** OrderTask {  *// 5 每分钟的第五秒 // 0/5 没隔五秒执行一次* @Scheduled(cron = **"5 \* \* \* \* ?"**) **public void** work(){  System.***out***.println(**"Thread ====== "**+ Thread.*currentThread*()); } @Scheduled(cron = **"0/5 \* \* \* \* ?"**) **public void** work1(){  System.***out***.println(**"Thread1 ====== "**+ Thread.*currentThread*()); }  } |
| *调日志输出模式为：error* |

关于@Scheduled

|  |  |
| --- | --- |
| 秒 | 0-59 |
| 分 | 0-59 |
| 小时 | 0-23 |
| 日期 | 1-31 |
| 月份 | 1-12 |
| 星期 | 1-7 |
| 年（可选） | 1970-2099 |

Linux 系统中有定时任务！命令

## 3、业务扫描

在OrderTask类中添加添加定时器 关闭过期订单

|  |
| --- |
| @Scheduled(cron = **"0/20 \* \* \* \* ?"**) **public void** checkOrder(){  System.out.println("开始处理过期订单");  long starttime = System.currentTimeMillis();  List<OrderInfo> expiredOrderList = **orderService**.getExpiredOrderList();  **for** (OrderInfo orderInfo : expiredOrderList) {  *// 处理未完成订单* **orderService**.execExpiredOrder(orderInfo);  }  long costtime = System.currentTimeMillis() - starttime;  System.out.println("一共处理"+expiredOrderList.size()+"个订单 共消耗"+costtime+"毫秒"); } |

在gmall-order-service开始业务完成，每隔一段时间进行查询未完成的业务

|  |
| --- |
| 接口  **public** List<OrderInfo> getExpiredOrderList(); |
| 实现类  *// 扫描过期订单方法* **public** List<OrderInfo> getExpiredOrderList(){  Example example = **new** Example(OrderInfo.**class**);  example.createCriteria().andLessThan(**"expireTime"**,**new** Date()).andEqualTo(**"processStatus"**,ProcessStatus.***UNPAID***);  **return orderInfoMapper**.selectByExample(example); } |

处理未完成订单

|  |
| --- |
| 接口  **public void** execExpiredOrder(OrderInfo orderInfo); |
| 实现类  *// 处理未完成订单* **public void** execExpiredOrder(OrderInfo orderInfo){  *// 订单信息* updateOrderStatus(orderInfo.getId(),ProcessStatus.***CLOSED***);  *// 付款信息* **paymentService**.closePayment(orderInfo.getId()); } |

在支付项目中gmall-payment 中做支付信息修改

|  |
| --- |
| **接口**  **public void** closePayment(String orderId); |
| **实现类**  *// 关闭支付信息* **public void** closePayment(String orderId){  Example example = **new** Example(PaymentInfo.**class**);  example.createCriteria().andEqualTo(**"orderId"**,orderId);  PaymentInfo paymentInfo = **new** PaymentInfo();  paymentInfo.setPaymentStatus(PaymentStatus.***ClOSED***);  **paymentInfoMapper**.updateByExampleSelective(paymentInfo,example); } |

轮询：延迟队列电商中一般都会出现！具体用谁？看精准度，看时间间隔，时间间隔短的用延迟队列，周期性比较长，用轮询！

SQL 语句：

|  |
| --- |
| UPDATE order\_info SET process\_status = 'UNPAID' , order\_status='UNPAID' WHERE id IN (148,149,150,151,152,153,154,155,156) |

## 4、利用多线程实现异步并发操作

默认扫描是单线程的即一次任务执行完，第二次的任务才能执行。如果第一次的任务被一些其他情况阻塞住了，那么第二次的扫描就没法开始了。

在订单项目中gmall-order-service项目中添加一个配置文件类AsyOrderConfig。

线程通信！

|  |
| --- |
| **package** com.atguigu.gmall.order.config;  **import** org.springframework.aop.interceptor.AsyncUncaughtExceptionHandler; **import** org.springframework.context.annotation.Bean; **import** org.springframework.context.annotation.Configuration; **import** org.springframework.scheduling.annotation.AsyncConfigurer; **import** org.springframework.scheduling.annotation.EnableAsync; **import** org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor;  **import** java.util.concurrent.Executor;  @Configuration @EnableAsync **public class** AsyOrderConfig **implements** AsyncConfigurer{  @Override  @Bean  **public** Executor getAsyncExecutor() {  *// 获取线程池 – 数据库的连接池* ThreadPoolTaskExecutor threadPoolTaskExecutor = **new** ThreadPoolTaskExecutor();  *// 设置线程数* threadPoolTaskExecutor.setCorePoolSize(10);  *// 设置最大连接数* threadPoolTaskExecutor.setMaxPoolSize(100);  *// 设置等待队列，如果10个不够，可以有100个线程等待 缓冲池* threadPoolTaskExecutor.setQueueCapacity(100);  *// 初始化操作* threadPoolTaskExecutor.initialize();  **return** threadPoolTaskExecutor;  }  @Override  **public** AsyncUncaughtExceptionHandler getAsyncUncaughtExceptionHandler() {  **return null**;  } } |

轮询：

在代码中的方法上可以标记@Async

|  |
| --- |
| *// 处理未完成订单* @Async **public void** execExpiredOrder(OrderInfo orderInfo){  *// 订单信息* updateOrderStatus(orderInfo.getId(),ProcessStatus.***CLOSED***);  *// 付款信息* **paymentService**.closePayment(orderInfo.getId()); } |

测试执行时间！

线程池的好处：提高数据处理能力，能够在多并发的情况下，减轻服务器压力，提高性能！

扩展：电商中用到线程池技术都有哪些？

# 四、拆单

## 1 为什么要拆单

因为同一个订单中不同的商品可能会**因为商家不同、仓库不同、物流渠道不同**。分成多个子订单分别进行跟踪。

## 2 何时拆单

可以选择下单时拆分，也可以选择支付时拆单。谷粒商城的项目以京东为参考选择支付时拆单。

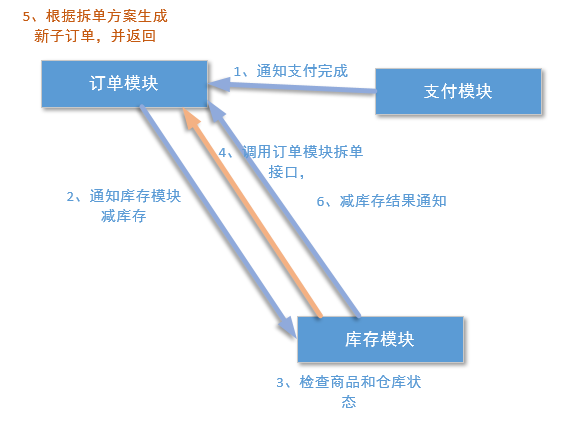
拆单的依据：是根据商品是否是同一个库存！

## 3 拆单对用户支付是否有影响？

用户支付时不影响拆单，如果是因为多商家进行的拆单，用户付款的金额会进行分账到不同的商家。

## 4 开发分析

### 示意图



### 库存系统调用拆单的接口

**拆单接口说明**

由库存系统发起申请

库存的配置文件中：

|  |  |  |  |
| --- | --- | --- | --- |
| 调用接口 | http://order.gmall.com/orderSplit | | |
| 请求参数 | orderId | | 订单系统的订单ID |
|  | wareSkuMap | | 仓库编号与商品的对照关系  例如  [{"wareId":"1","skuIds":["2","10"]},{"wareId":"2","skuIds":["3"]}]  表示：sku为2号，10号的商品在1号仓库  sku为3号的商品在2号仓库 |
| 请求方式 | post | | |
| 返回值格式 | json集合 | 子订单的集合json【拆单之后的集合】 | |
|  | orderId | 订单系统的订单ID | |
|  | consignee | 收货人 | |
|  | consigneeTel | 收货电话 | |
|  | orderComment | 订单备注 | |
|  | orderBody | 订单概要 | |
|  | deliveryAddress | 发货地址 | |
|  | paymentWay | 支付方式： ‘1’ 为货到付款，‘2’为在线支付。 | |
|  | details:  skuId  skuNum  skuName | 购买商品明细  例如：  details:[{skuId:101,skuNum:1,skuName:  ’小米手64G’},  {skuId:201,skuNum:1,skuName:’索尼耳机’}] | |
|  | wareId | 传入时的仓库编号 | |
| 返回值例： | [  {  "orderBody": "小米 红米5 移动联通电信4G手机 双卡双待等1件商品",  "consignee": "admin",  "orderComment": "123123",  "wareId": "1",  "orderId": "16",  "deliveryAddress": "宏福科技综合楼",  "details": [  {  "skuName": "小米 红米5 动联通电信4G手机 双卡双待",  "skuId": "2",  "skuNum": 7  }  ],  "paymentWay": "2"  },  {  "orderBody": "小米 红米5 Plus 全面屏手 版 4GB+64GB 黑色 等1件商品",  "consignee": "admin",  "orderComment": "123123",  "wareId": "2",  "orderId": "17",  "deliveryAddress": "宏福科技综合楼",  "details": [  {  "skuName": "小米 红米5 Plus 全 +64GB 黑色 ",  "skuId": "3",  "skuNum": 3  }  ],  "paymentWay": "2"  }  ] | | |
|  |
|  |

### 代码实现

#### OrderController

|  |
| --- |
| @RequestMapping(**"orderSplit"**) @ResponseBody **public** String orderSplit(HttpServletRequest request){  String orderId = request.getParameter(**"orderId"**);  String wareSkuMap = request.getParameter(**"wareSkuMap"**);  *// 定义订单集合* List<OrderInfo> subOrderInfoList = **orderService**.splitOrder(orderId,wareSkuMap);  List<Map> wareMapList=**new** ArrayList<>();  **for** (OrderInfo orderInfo : subOrderInfoList) {  Map map = **orderService**.initWareOrder(orderInfo);  wareMapList.add(map);  }  **return** JSON.*toJSONString*(wareMapList); } |

注意： initWareOrder 方法中一定有仓库Id。

map.put(**"wareId"**,orderInfo.getWareId());

#### OrderServiceImpl

|  |
| --- |
| **public** List<OrderInfo> splitOrder(String orderId,String wareSkuMap){  List<OrderInfo> subOrderInfoList = **new** ArrayList<>();  *// 1 先查询原始订单* OrderInfo orderInfoOrigin = getOrderInfo(orderId);  *// 2 wareSkuMap 反序列化* List<Map> maps = JSON.*parseArray*(wareSkuMap, Map.**class**);  *// 3 遍历拆单方案* **for** (Map map : maps) {  String wareId = (String) map.get(**"wareId"**);  List<String> skuIds = (List<String>) map.get(**"skuIds"**);  *// 4 生成订单主表，从原始订单复制，新的订单号，父订单* OrderInfo subOrderInfo = **new** OrderInfo();  BeanUtils.*copyProperties*(subOrderInfo,orderInfoOrigin);  subOrderInfo.setId(**null**);  *// 5 原来主订单，订单主表中的订单状态标志为拆单* subOrderInfo.setParentOrderId(orderInfoOrigin.getId());  subOrderInfo.setWareId(wareId);   *// 6 明细表 根据拆单方案中的skuids进行匹配，得到那个的子订单* List<OrderDetail> orderDetailList = orderInfoOrigin.getOrderDetailList();  *// 创建一个新的订单集合* List<OrderDetail> subOrderDetailList = **new** ArrayList<>();  **for** (OrderDetail orderDetail : orderDetailList) {  **for** (String skuId : skuIds) {  **if** (skuId.equals(orderDetail.getSkuId())){  orderDetail.setId(**null**);  subOrderDetailList.add(orderDetail);  }  }  }  subOrderInfo.setOrderDetailList(subOrderDetailList);  subOrderInfo.sumTotalAmount();  *// 7 保存到数据库中* saveOrder(subOrderInfo);  subOrderInfoList.add(subOrderInfo);  }  updateOrderStatus(orderId,ProcessStatus.***SPLIT***);  *// 8 返回一个新生成的子订单列表* **return** subOrderInfoList; } |

注意：必须将商品放在不同的仓库中！



#### 测试

测试：payment.gmall.com/sendPaymentResult?orderId=53&result=success ,如果该订单已经进行拆单了。那么可以从新付款再试！