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。比如用户几天后待办提醒，生日提醒。

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

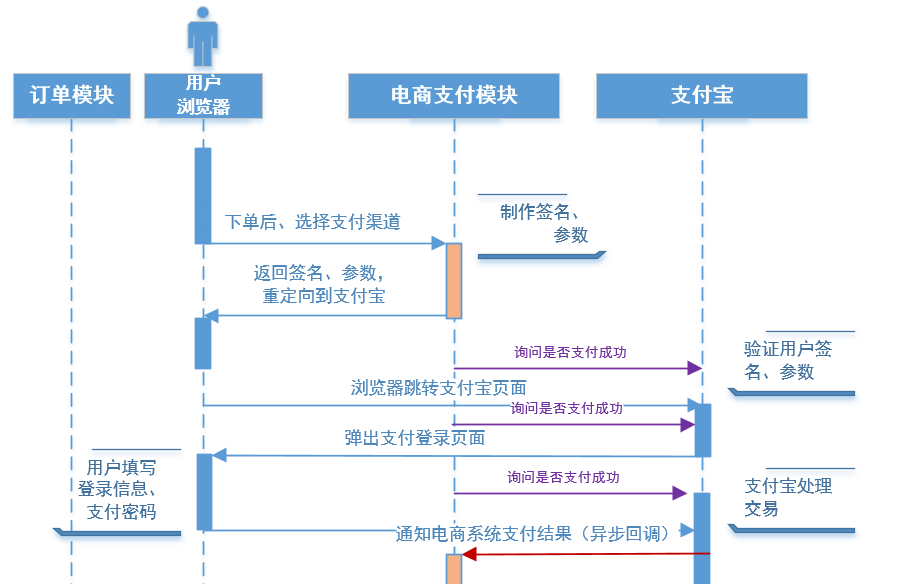
## 1 应用场景

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



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。

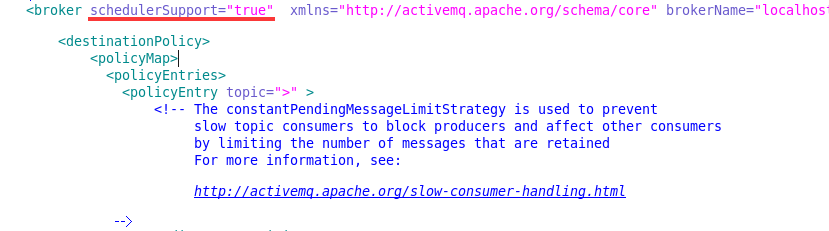
|  |
| --- |
| @Autowired AlipayClient **alipayClient**;  **public** PaymentStatus checkAlipayPayment(PaymentInfo paymentInfo){   System.***out***.println(**"开始主动检查支付状态，paymentInfo.toString() = "** + paymentInfo.toString());  *//先检查当前数据库是否已经变为“已支付状态”* **if**(paymentInfo.getId()==**null**){  System.***out***.println(**"outTradeNo:"**+paymentInfo.getOutTradeNo() );  paymentInfo = getPaymentInfo(paymentInfo);  }  **if** (paymentInfo.getPaymentStatus()== PaymentStatus.***PAID***){  System.***out***.println(**"该单据已支付:"**+paymentInfo.getOutTradeNo());  **return** PaymentStatus.***PAID***;  }   *//如果不是已支付，继续去查询alipay的接口* System.***out***.println(**"%% % % 查询alipay的接口"** );  AlipayTradeQueryRequest alipayTradeQueryRequest=**new** AlipayTradeQueryRequest();  alipayTradeQueryRequest.setBizContent(**"{\"out\_trade\_no\":\""**+paymentInfo.getOutTradeNo()+**"\"}"**);  AlipayTradeQueryResponse response=**null**;  **try** {  response = **alipayClient**.execute(alipayTradeQueryRequest);  } **catch** (AlipayApiException e) {  e.printStackTrace();  }     **if**(response.isSuccess()){  String tradeStatus = response.getTradeStatus();   **if** (**"TRADE\_SUCCESS"**.equals(tradeStatus)){  System.***out***.println(**"支付完成 ====================== "** );  *//如果结果是支付成功 ,则更新支付状态* PaymentInfo paymentInfo4Upt=**new** PaymentInfo();  paymentInfo4Upt.setPaymentStatus(PaymentStatus.***PAID***);  paymentInfo4Upt.setCallbackTime(**new** Date());  paymentInfo4Upt.setCallbackContent(response.getBody());  paymentInfo4Upt.setId(paymentInfo.getId());  **paymentInfoMapper**.updateByPrimaryKeySelective(paymentInfo4Upt);   *// 然后发送通知给订单* sendPaymentResult(paymentInfo,**"success"**);  **return** PaymentStatus.***PAID***;  }**else**{  System.***out***.println(**"支付尚未完成 ？？？？？？？？？？ "** );  **return** PaymentStatus.***UNPAID***;  }  }**else**{  System.***out***.println(**"支付尚未完成 ？？？？？？？？？？ "** );  **return** PaymentStatus.***UNPAID***;  }   } |

## 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



开启 schedulerSupport="true"

发送延迟队列

|  |
| --- |
| **public void** sendDelayPaymentResult(String outTradeNo,**int** delaySec,**int** checkCount){  *//发送支付结果* Connection connection = **activeMQUtil**.getConnection();  **try** {  connection.start();  Session session = connection.createSession(**true**, Session.***SESSION\_TRANSACTED***);  Queue paymentResultQueue = session.createQueue(**"PAYMENT\_RESULT\_CHECK\_QUEUE"**);  MessageProducer producer = session.createProducer(paymentResultQueue);  producer.setDeliveryMode(DeliveryMode.***PERSISTENT***);  MapMessage mapMessage= **new** ActiveMQMapMessage();  mapMessage.setString(**"outTradeNo"**,outTradeNo);  mapMessage.setInt(**"delaySec"**,delaySec);  mapMessage.setInt(**"checkCount"**,checkCount);  mapMessage.setLongProperty(ScheduledMessage.***AMQ\_SCHEDULED\_DELAY***,delaySec\*1000);  producer.send(mapMessage);   session.commit();  producer.close();  session.close();  connection.close();   } **catch** (JMSException e) {  e.printStackTrace();  }  } |

接收延迟队列的消费端

|  |
| --- |
| @Component **public class** PaymentConsumer {   @Autowired  PaymentService **paymentService**;    @JmsListener(destination = **"PAYMENT\_RESULT\_CHECK\_QUEUE"**,containerFactory = **"jmsQueueListener"**)  **public void** consumeCheckResult(MapMessage mapMessage) **throws** JMSException {  **int** delaySec = mapMessage.getInt(**"delaySec"**);  String outTradeNo = mapMessage.getString(**"outTradeNo"**);  **int** checkCount = mapMessage.getInt(**"checkCount"**);   PaymentInfo paymentInfo=**new** PaymentInfo();  paymentInfo.setOutTradeNo(outTradeNo);  PaymentStatus paymentStatus = **paymentService**.checkAlipayPayment(paymentInfo);  **if**(paymentStatus==PaymentStatus.***UNPAID***&&checkCount>0){  System.***out***.println(**"checkCount = "** + checkCount);  **paymentService**.sendDelayPaymentResult(outTradeNo,delaySec,checkCount-1);  }   }   } |

# 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

轮询扫描有很多工具，比较经典的就是 quartz。

但是springboot整合了自家的spring task ，功能上基本和quartz差不多，但是配置更简单，全程只用注解就可以，不用额外的xml。

测试Demo

|  |
| --- |
| @Component @EnableScheduling **public class** OrderTask {   @Autowired  OrderService **orderService**;   @Scheduled(cron = **"0/5 \* \* \* \* ?"**)  **public void** work() **throws** InterruptedException {  System.***out***.println(**"thread = ==============="** + Thread.*currentThread*());  }  } |

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。

|  |
| --- |
| @Bean **public** TaskScheduler taskScheduler() {  ThreadPoolTaskScheduler taskScheduler = **new** ThreadPoolTaskScheduler();  taskScheduler.setPoolSize(5);  **return** taskScheduler; } |

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 |

## 3 业务扫描

|  |
| --- |
| @Scheduled(cron = **"0/30 \* \* \* \* ?"**) **public void** checkUnpaidOrder() {  System.***out***.println(**"开始检查未付款单据 = "**);  Long beginTime=System.*currentTimeMillis*();  List<OrderInfo> unpaidOrderList = **orderService**.getUnpaidOrderList();  **for** (OrderInfo orderInfo : unpaidOrderList) {  **orderService**.checkExpireOrder(orderInfo);  }  Long costtime=System.*currentTimeMillis*()-beginTime;  System.***out***.println(**"开始检查完毕未付款单据 = 共消耗"**+costtime); } |

|  |
| --- |
|  |

|  |
| --- |
| **public void** checkExpireOrder(OrderInfo orderInfo ) {    updateProcessStatus(orderInfo.getId(), ProcessStatus.***CLOSED***);  **paymentService**.closePayment(orderInfo.getId());     **return** ;  } |

## 4 利用多线程实现异步并发操作

|  |
| --- |
| @Configuration @EnableAsync **public class** AsyncTaskConfig **implements** AsyncConfigurer {  @Override  @Bean  **public** Executor getAsyncExecutor() {  ThreadPoolTaskExecutor threadPoolTaskExecutor=**new** ThreadPoolTaskExecutor();  threadPoolTaskExecutor.setCorePoolSize(10); *//线程数* threadPoolTaskExecutor.setQueueCapacity(100); *//等待队列容量 ，线程数不够任务会等待* threadPoolTaskExecutor.setMaxPoolSize(50); *// 最大线程数，等待数不够会增加线程数，直到达此上线 超过这个范围会抛异常* threadPoolTaskExecutor.initialize();  **return** threadPoolTaskExecutor;   }   @Override  @Bean  **public** AsyncUncaughtExceptionHandler getAsyncUncaughtExceptionHandler() {  **return null**;  } } |

在代码中的方法上可以标记@Async

|  |
| --- |
| @Async  **public void** checkExpireOrder(OrderInfo orderInfo ) {  Date expireDate= DateUtil.*addDays*(orderInfo.getCreateTime(),1) ;  **if** (**new** Date().after(expireDate)){  updateProcessStatus(orderInfo.getId(), ProcessStatus.***CLOSED***);  **paymentService**.closePayment(orderInfo.getId());  }  **return** ; } |

|  |
| --- |
| **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);  } |