谷粒商城

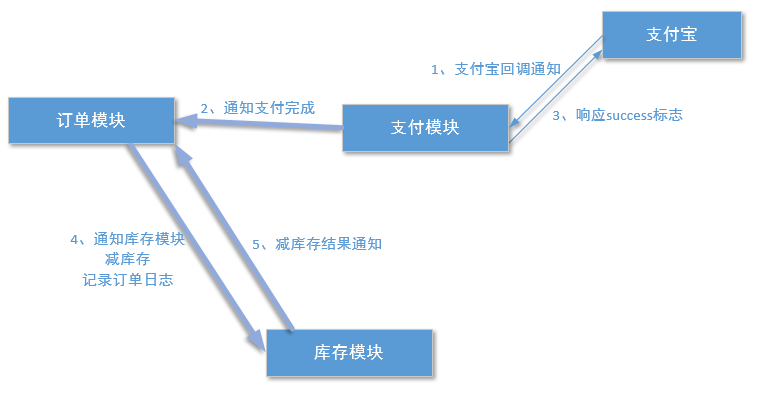
版本：V 1.0

异步通信



**当前的位置**

# 一、分布式的业务场景



1 、如何高效完成各个分布式系统的协作

通过消息队列来达到异步解耦的效果，减少了程序之间的阻塞等待时间，资源浪费。

2、消息的弊端？如何解决？

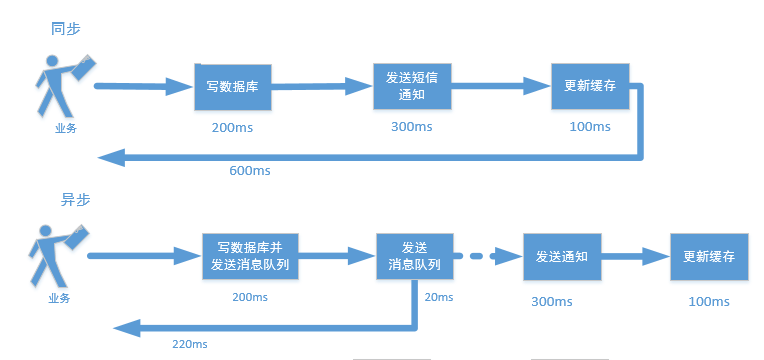
消息队列的问题在于**不确定性**，不能绝对保证消息的准确到达，所以要引入延迟队列、周期性的主动轮询，来发现未到达的消息，从而进行补偿。

# 二、消息队列简介

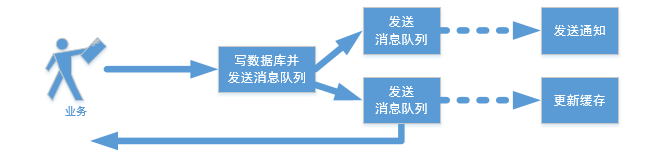
消息队列，也叫消息中间件。消息的传输过程中，消息保存在消息容器中。

消息队列都解决了什么问题？

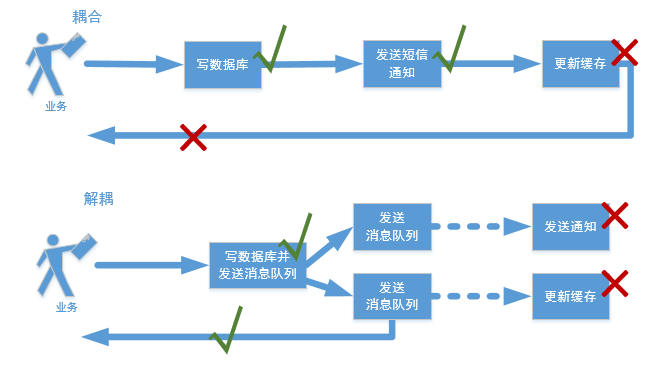
1. **异步**



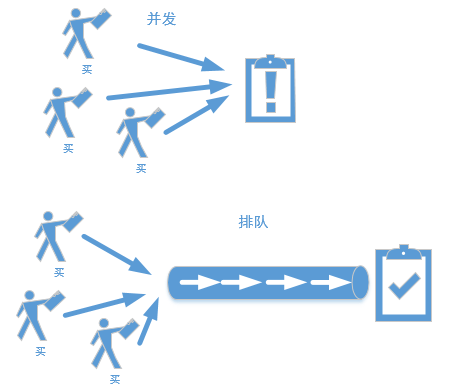
2、并行



3、解耦

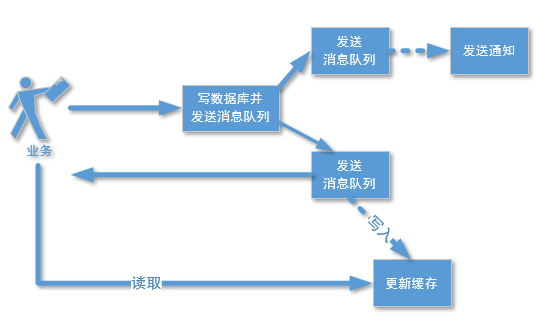


4、排队



5 弊端：不确定性和延迟

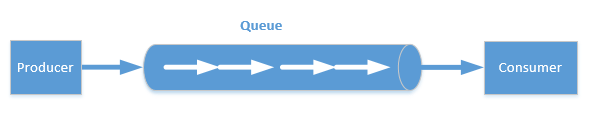
轮询，延迟队列！



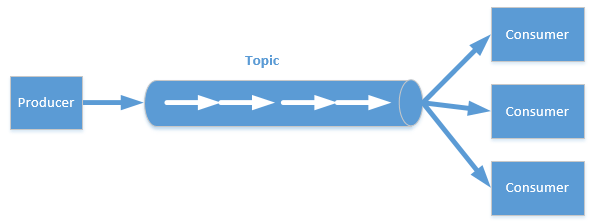
解决方案：最终一致性，换会时间，提高效率！

## 消息模式

点对点：给好友发送消息



订阅 ： 微信公共号



# 三 消息队列工具 ActiveMQ

## 1 、简介



同类产品： RabbitMQ 、 Kafka、Redis（List）….

### 1.1 对比RabbitMQ

最接近的同类型产品，经常拿来比较，性能伯仲之间，基本上可以互相替代。最主要区别是二者的协议不同**RabbitMQ**的协议是**AMQP**(Advanced Message Queueing Protoco)，而**ActiveMQ**使用的是**JMS**(Java Messaging Service )协议。顾名思义JMS是针对Java体系的传输协议，队列两端必须有JVM，所以如果开发环境都是**java**的话推荐使用ActiveMQ，可以用Java的一些对象进行传递比如Map、BLob、Stream等。而AMQP通用行较强，非java环境经常使用，传输内容就是标准字符串。

另外一点就是**RabbitMQ用Erlang开发**，安装前要装Erlang环境，比较麻烦。ActiveMQ解压即可用不用任何安装。

### 1.2 对比KafKa

Kafka性能超过ActiveMQ等传统MQ工具，集群扩展性好。

弊端是：

在传输过程中可能会出现消息重复的情况，

不保证发送顺序

一些传统MQ的功能没有，比如消息的事务功能。

所以通常用Kafka处理大数据日志。

### 1.3 对比Redis

其实Redis本身利用List可以实现消息队列的功能，但是功能很少，而且队列体积较大时性能会急剧下降。对于数据量不大、业务简单的场景可以使用。

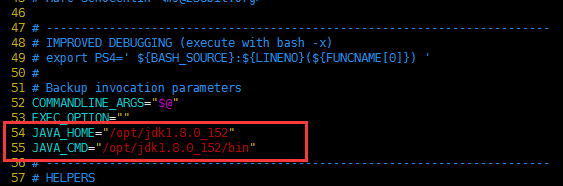
## 2 安装 ActiveMQ

拷贝apache-activemq-5.14.4-bin.tar.gz到Linux服务器的**/opt**下

解压缩 tar -zxvf apache-activemq-5.14.4-bin.tar.gz

重命名 mv apache-activemq-5.14.4 activemq

vim /opt/activemq/bin/activemq



增加两行 vim /etc/profile

|  |
| --- |
| JAVA\_HOME="/opt/jdk1.8.0\_152"  JAVA\_CMD="/opt/jdk1.8.0\_152/bin" |

/etc/init.d

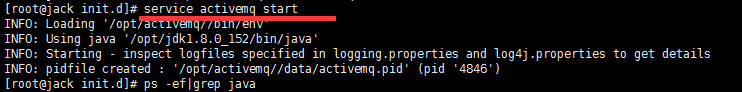
注册服务 /etc/init.d 使用软连接 ln -s

软连接：一个快捷方式

|  |
| --- |
| ln -s /opt/activemq/bin/activemq /etc/init.d/activemq  添加到服务  chkconfig --add activemq  # **禁止使用**  # cp /opt/activemq/bin/activemq /etc/init.d/activemq |

启动服务

service activemq start



关闭服务

service activemq stop

通过netstat 查看端口

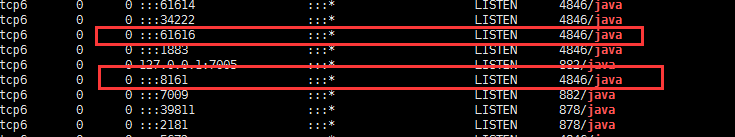
# netstat -tlnp

t:表示tcp

l:表示监听

n: 将ip和端口转换成域名和服务名

p:显示的程序名

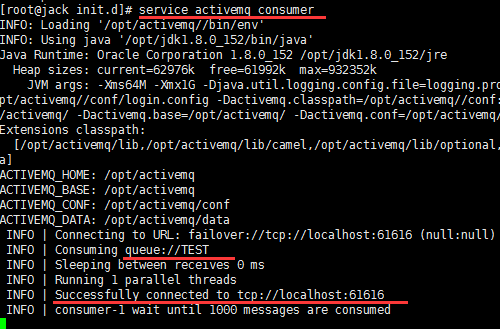


activemq两个重要的端口，一个是提供消息队列的默认端口：**61616**

另一个是控制台端口**8161**

通过控制台测试

启动消费端 service activemq consumer

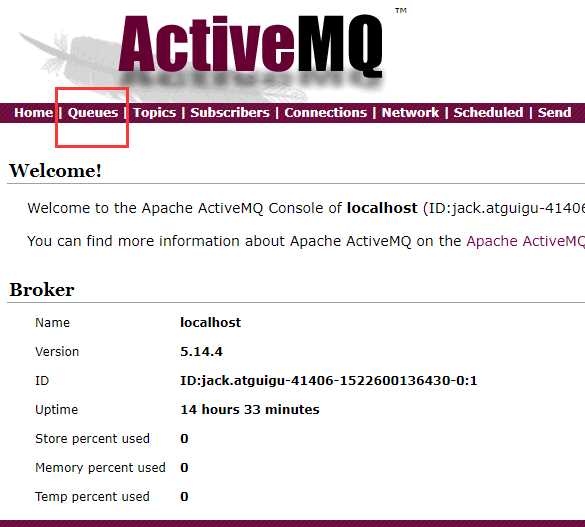


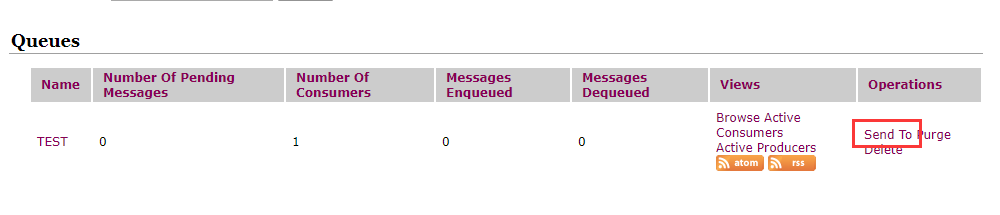
进入网页控制台

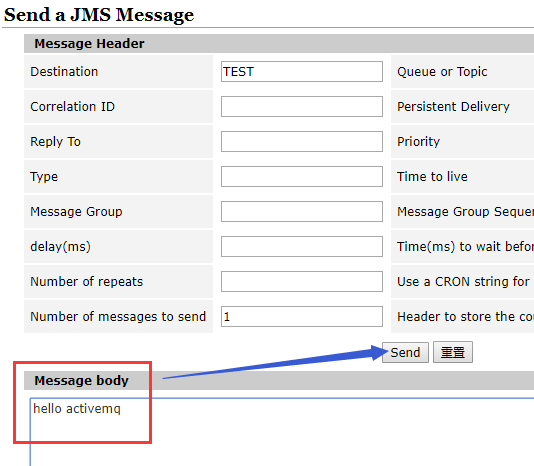


**账号/密码默认： admin/admin**

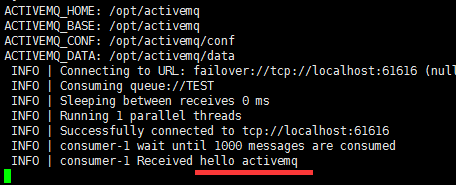
**点击Queues**







**观察客户端**



## 在Java中使用消息队列

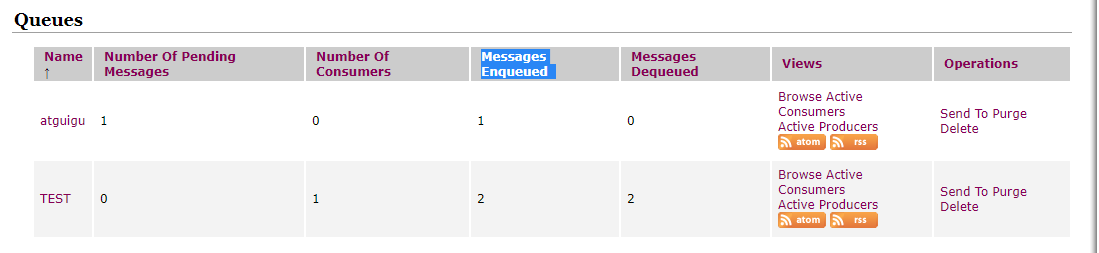
### 3.1 在gmall-service-util中导入依赖坐标

|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-activemq</**artifactId**>  <**exclusions**>  <**exclusion**>  <**groupId**>org.slf4j</**groupId**>  <**artifactId**>slf4j-log4j12</**artifactId**>  </**exclusion**>  </**exclusions**> </**dependency**>  <**dependency**>  <**groupId**>org.apache.activemq</**groupId**>  <**artifactId**>activemq-pool</**artifactId**>  <**version**>5.15.2</**version**>  <**exclusions**>  <**exclusion**>  <**groupId**>org.slf4j</**groupId**>  <**artifactId**>slf4j-log4j12</**artifactId**>  </**exclusion**>  </**exclusions**> </**dependency**> |

### 3.2 在payment项目中添加producer端

|  |
| --- |
| **public class** ProducerTest {  **public static void** main(String[] args) **throws** JMSException {  *// 创建连接工厂* ConnectionFactory connectionFactory = **new** ActiveMQConnectionFactory(**"tcp://192.168.67.201:61616"**);  Connection connection = connectionFactory.createConnection();  connection.start();  *// 创建session 第一个参数表示是否支持事务，false时，第二个参数Session.AUTO\_ACKNOWLEDGE，Session.CLIENT\_ACKNOWLEDGE，DUPS\_OK\_ACKNOWLEDGE其中一个  // 第一个参数设置为true时，第二个参数可以忽略 服务器设置为SESSION\_TRANSACTED* Session session = connection.createSession(**false**, Session.***AUTO\_ACKNOWLEDGE***);  *// 创建队列* Queue queue = session.createQueue(**"Atguigu"**);   MessageProducer producer = session.createProducer(queue);  *// 创建消息对象* ActiveMQTextMessage activeMQTextMessage = **new** ActiveMQTextMessage();  activeMQTextMessage.setText(**"hello ActiveMq!"**);  *// 发送消息* producer.send(activeMQTextMessage);  producer.close();  connection.close();  } } |

**注意：如果有事务需要先提交事务session.commit();**



Number Of Pending Messages 等待消费的消息 这个是当前未出队列的数量。可以理解为总接收数-总出队列数

Number Of Consumers  消费者 这个是消费者端的消费者数量

Messages Enque[ued](http://www.haogongju.net/tag/ued) 进入队列的消息  进入队列的总数量,包括出队列的。 这个数量只增不减

Messages Dequeued 出了队列的消息  可以理解为是消费者消费掉的数量

总结：

当有一个消息进入这个队列时，等待消费的消息是1，进入队列的消息是1。  
当消息消费后，等待消费的消息是0，进入队列的消息是1，消费者 1，出队列的消息是1.  
在来一条消息时，等待消费的消息是1，进入队列的消息就是2.

### 3.3 在payment项目中添加consumer端

|  |
| --- |
| **public class** ConsumerTest {  **public static void** main(String[] args) **throws** JMSException {  ActiveMQConnectionFactory activeMQConnectionFactory = **new** ActiveMQConnectionFactory(ActiveMQConnection.***DEFAULT\_USER***, ActiveMQConnection.***DEFAULT\_PASSWORD***, **"tcp://192.168.67.201:61616"**);  *// 创建连接* Connection connection = activeMQConnectionFactory.createConnection();  connection.start();  *// 创建会话* Session session = connection.createSession(**false**, Session.***AUTO\_ACKNOWLEDGE***);  *// 创建队列* Queue queue = session.createQueue(**"Atguigu"**);  *// 创建Consumer* MessageConsumer consumer = session.createConsumer(queue);  *// 接收消息* consumer.setMessageListener(**new** MessageListener() {  @Override  **public void** onMessage(Message message) {  *// 参数就是收到的消息* **if** (message **instanceof** TextMessage){  **try** {  String text = ((TextMessage) message).getText();  System.***out***.println(text+**"接收的消息！"**);  } **catch** (JMSException e) {  e.printStackTrace();  }  }  }  });  } } |

### 3.4 关于事务控制

|  |  |  |
| --- | --- | --- |
| **producer提交时的事务** | **事务开启 true** | **只执行send并不会提交到队列中，只有当执行session.commit()时，消息才被真正的提交到队列中。** |
| **事务不开启 false** | **只要执行send，就进入到队列中。** |
| **consumer 接收时的事务** | **事务开启，签收必须写**  Session.***SESSION\_TRANSACTED*** | **收到消息后，消息并没有真正的被消费。消息只是被锁住。一旦出现该线程死掉、抛异常，或者程序执行了session.rollback()那么消息会释放，重新回到队列中被别的消费端再次消费。** |
| **事务不开启，签收方式选择**  Session.AUTO\_ACKNOWLEDGE | 只要调用comsumer.receive方法 ，自动确认。 |
| **事务不开启，签收方式选择**  Session.CLIENT\_ACKNOWLEDGE | 需要客户端执行 message.acknowledge(),否则视为未提交状态，线程结束后，其他线程还可以接收到。  这种方式跟事务模式很像，区别是不能手动回滚,而且可以单独确认某个消息。  **手动签收** |
| **事务不开启，签收方式选择**  Session.DUPS\_OK\_ACKNOWLEDGE | 在Topic模式下做批量签收时用的，可以提高性能。但是某些情况消息可能会被重复提交，使用这种模式的consumer要可以处理重复提交的问题。 |

### 3.5 持久化与非持久化

**通过**producer.setDeliveryMode(DeliveryMode.***PERSISTENT***) 进行设置

持久化的好处就是当activemq宕机的话，消息队列中的消息不会丢失。非持久化会丢失。但是会消耗一定的性能。

持久化：当服务器宕机，消息依然存在。

非持久化：当服务器宕机，消息不存在。

在zookeeper中节点，有持久化-非持久化。

默认持久化！

将activemq 关闭！

# 四 Activemq与springboot整合

配置在gmall-service-util项目中

## 1 工具类ActiveMQUtil

|  |
| --- |
| **public class** ActiveMQUtil {  PooledConnectionFactory **pooledConnectionFactory** = **null**;  **public void** init(String brokerUrl){  ActiveMQConnectionFactory activeMQConnectionFactory = **new** ActiveMQConnectionFactory(brokerUrl);  **pooledConnectionFactory** = **new** PooledConnectionFactory(activeMQConnectionFactory);  *//设置超时时间* **pooledConnectionFactory**.setExpiryTimeout(2000);  *// 设置出现异常的时候，继续重试连接* **pooledConnectionFactory**.setReconnectOnException(**true**);  *// 设置最大连接数* **pooledConnectionFactory**.setMaxConnections(5);  }  *// 获取连接* **public** Connection getConnection(){  Connection connection = **null**;  **try** {  connection = **pooledConnectionFactory**.createConnection();  } **catch** (JMSException e) {  e.printStackTrace();  }  **return** connection;  } } |

## 2 配置类ActiveMQConfig

|  |
| --- |
| @Configuration **public class** ActiveMQConfig {   @Value(**"${spring.activemq.broker-url:disabled}"**)  String **brokerURL** ;   @Value(**"${activemq.listener.enable:disabled}"**)  String **listenerEnable**;   *// 获取*activeMQUtil@Bean  **public** ActiveMQUtil getActiveMQUtil(){  **if** (**"disabled"**.equals(**brokerURL**)){  **return null**;  }  ActiveMQUtil activeMQUtil = **new** ActiveMQUtil();  activeMQUtil.init(**brokerURL**);  **return** activeMQUtil;  }   @Bean(name = **"jmsQueueListener"**)  **public** DefaultJmsListenerContainerFactory jmsQueueListenerContainerFactory(ActiveMQConnectionFactory activeMQConnectionFactory) {   **if**(**"disabled"**.equals(**listenerEnable**)){  **return null**;  }  DefaultJmsListenerContainerFactory factory = **new** DefaultJmsListenerContainerFactory();  factory.setConnectionFactory(activeMQConnectionFactory);  *// 设置事务* factory.setSessionTransacted(**false**);  *// 手动签收* factory.setSessionAcknowledgeMode(Session. AUTO\_ACKNOWLEDGE);  *// 设置并发数* factory.setConcurrency(**"5"**);  *// 重连间隔时间* factory.setRecoveryInterval(5000L);  **return** factory;  }  *// 接收消息* @Bean  **public** ActiveMQConnectionFactory activeMQConnectionFactory (){  ActiveMQConnectionFactory activeMQConnectionFactory =  **new** ActiveMQConnectionFactory(**brokerURL**);  **return** activeMQConnectionFactory;  } } |

**brokerURL: tcp://192.168.67.200:61616**

# 五 在支付业务模块中应用

## 1 支付成功通知

支付模块利用消息队列通知订单系统，支付成功

在gmall-payment支付模块中配置application.properties

|  |
| --- |
| **spring.activemq.broker-url**=**tcp://192.168.67.204:61616 spring.activemq.pool.enabled**=**true activemq.listener.enable**=**true** |

在PaymentServiceImpl中增加发送方法：

接口：发送消息，给activemq 支付结果！success,fail

发送一个orderId，result 【success,fail】

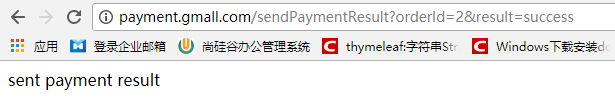
|  |
| --- |
| **public void** sendPaymentResult(PaymentInfo paymentInfo,String result); |

|  |
| --- |
| *// 添加发送方法* **public void** sendPaymentResult(PaymentInfo paymentInfo,String result){  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\_QUEUE"**);  MessageProducer producer = session.createProducer(paymentResultQueue);  MapMessage mapMessage = **new** ActiveMQMapMessage();  mapMessage.setString(**"orderId"**,paymentInfo.getOrderId());  mapMessage.setString(**"result"**,result);  producer.send(mapMessage);  **session.commit();**  producer.close();  session.close();  connection.close();  } **catch** (JMSException e) {  e.printStackTrace();  } } |

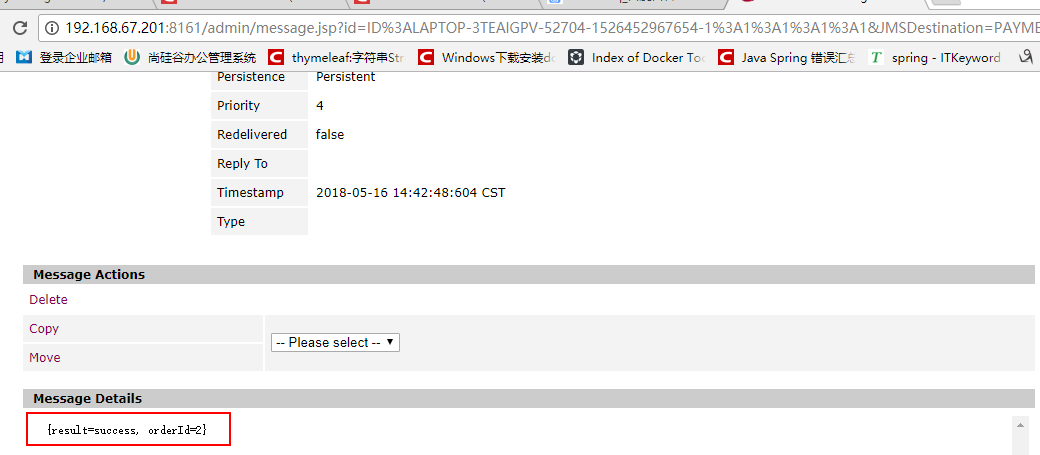
在PaymentController中增加一个方法用来测试

|  |
| --- |
| *// 发送验证* @RequestMapping(**"sendPaymentResult"**) @ResponseBody **public** String sendPaymentResult(PaymentInfo paymentInfo,@RequestParam(**"result"**) String result){  **paymentService**.sendPaymentResult(paymentInfo,result);  **return "sent payment result"**; } |
|  |

在浏览器中访问：



查看队列内容：有一个在队列中没有被消费的消息。



## 2 订单模块消费消息

在订单模块中【gmall-order-service】新添加一个消费类

配置文件添加

|  |
| --- |
| **spring.activemq.broker-url**=**tcp://192.168.67.204:61616 spring.activemq.pool.enabled**=**true activemq.listener.enable**=**true** |

消费类

|  |
| --- |
| **package com.atguigu.gmall.order.mq**;  **import** com.alibaba.dubbo.config.annotation.Reference; **import** com.atguigu.gmall.bean.enums.ProcessStatus; **import** com.atguigu.gmall.service.OrderService; **import** org.springframework.beans.factory.annotation.Autowired; **import** org.springframework.jms.annotation.JmsListener; **import** org.springframework.stereotype.Component;  **import** javax.jms.JMSException; **import** javax.jms.MapMessage;  @Component **public class** OrderConsumer {   @Autowired  OrderService **orderService**;   @JmsListener(destination = **"PAYMENT\_RESULT\_QUEUE"**,containerFactory = **"jmsQueueListener"**)  **public void** consumerPaymentResult(MapMessage mapMessage) **throws** JMSException {  String orderId = mapMessage.getString(**"orderId"**);  String result = mapMessage.getString(**"result"**);  System.***out***.println(**"result = "** + result);  System.***out***.println(**"orderId = "** + orderId);  **if** (**"success"**.equals(result)){  **orderService**.updateOrderStatus(orderId, ProcessStatus.***PAID***);  }  } } |

实现类

|  |
| --- |
| **public void** updateOrderStatus(String orderId,ProcessStatus processStatus){  OrderInfo orderInfo = **new** OrderInfo();  orderInfo.setId(orderId);  orderInfo.setProcessStatus(processStatus);  orderInfo.setOrderStatus(processStatus.getOrderStatus());  **orderInfoMapper**.updateByPrimaryKeySelective(orderInfo); } |

此处：记得在配置文件中添加 因为：修改的状态为enum 类型，所以在配置文件中添加如下配置

# application.properties

|  |
| --- |
| **mapper.enum-as-simple-type**=**true** |

## 3 订单模块发送减库存通知

订单模块除了接收到请求改变单据状态，还要发送库存系统

查看看《库存管理系统接口手册》中【**减库存的消息队列消费端接口**】中的描述，组织相应的消息数据进行传递。

创建数据库表：

|  |
| --- |
| create table ware\_order\_task  (  id bigint /\*auto\_increment\*/ not null,  order\_id bigint(20),  consignee VARCHAR(100),  consignee\_tel VARCHAR(20),  delivery\_address VARCHAR(1000),  order\_comment VARCHAR(200),  payment\_way VARCHAR(2),  task\_status VARCHAR(20),  order\_body VARCHAR(200),  tracking\_no VARCHAR(200),  create\_time DATETIME,  ware\_id bigint,  task\_comment VARCHAR(500)  );  alter table ware\_order\_task  add constraint PK\_ware\_order\_task\_id primary key (id); |
| create table ware\_order\_task\_detail  (  id bigint /\*auto\_increment\*/ not null,  sku\_id bigint,  sku\_name VARCHAR(200),  sku\_num INTEGER,  task\_id bigint  );  alter table ware\_order\_task\_detail  add constraint PK\_ware\_orail\_id6CBB primary key (id); |

OrderConsumer.java

|  |
| --- |
| @Component **public class** OrderConsumer {   @Reference  OrderService **orderService**;   @JmsListener(destination = **"PAYMENT\_RESULT\_QUEUE"**,containerFactory = **"jmsQueueListener"**)  **public void** consumerPaymentResult(MapMessage mapMessage) **throws** JMSException {  String orderId = mapMessage.getString(**"orderId"**);  String result = mapMessage.getString(**"result"**);  System.***out***.println(**"result = "** + result);  System.***out***.println(**"orderId = "** + orderId);  **if** (**"success"**.equals(result)){  *// 更新支付状态* **orderService**.updateOrderStatus(orderId, ProcessStatus.***PAID***);  *// 通知减库存* **orderService**.sendOrderStatus(orderId);  **orderService**.updateOrderStatus(orderId, ProcessStatus.***NOTIFIED\_WARE***  );  }**else** {  **orderService**.updateOrderStatus(orderId,ProcessStatus.***UNPAID***);  }  } } |
| OrderService接口  **public void** sendOrderStatus(String orderId); |
| OrderServiceImpl实现类  **public void** sendOrderStatus(String orderId){  Connection connection = **activeMQUtil**.getConnection();  String orderJson = initWareOrder(orderId);  **try** {  connection.start();  Session session = connection.createSession(**true**, Session.***SESSION\_TRANSACTED***);  Queue order\_result\_queue = session.createQueue(**"ORDER\_RESULT\_QUEUE"**);  MessageProducer producer = session.createProducer(order\_result\_queue);   ActiveMQTextMessage textMessage = **new** ActiveMQTextMessage();  textMessage.setText(orderJson);  producer.send(textMessage);  session.commit();  session.close();  producer.close();  connection.close();  } **catch** (JMSException e) {  e.printStackTrace();  } }  **public** String initWareOrder(String orderId){  OrderInfo orderInfo = getOrderInfo(orderId);  Map map = initWareOrder(orderInfo);  **return** JSON.*toJSONString*(map); } *// 设置初始化仓库信息方法* **public** Map initWareOrder (OrderInfo orderInfo){  Map<String,Object> map = **new** HashMap<>();  map.put(**"orderId"**,orderInfo.getId());  map.put(**"consignee"**, orderInfo.getConsignee());  map.put(**"consigneeTel"**,orderInfo.getConsigneeTel());  map.put(**"orderComment"**,orderInfo.getOrderComment());  map.put(**"orderBody"**,orderInfo.getTradeBody());  map.put(**"deliveryAddress"**,orderInfo.getDeliveryAddress());  map.put(**"paymentWay"**,**"2"**);   **map.put("wareId",orderInfo.getWareId());**   *// 组合json* List detailList = **new** ArrayList();  List<OrderDetail> orderDetailList = orderInfo.getOrderDetailList();  **for** (OrderDetail orderDetail : orderDetailList) {  Map detailMap = **new** HashMap();  detailMap.put(**"skuId"**,orderDetail.getSkuId());  detailMap.put(**"skuName"**,orderDetail.getSkuName());  detailMap.put(**"skuNum"**,orderDetail.getSkuNum());  detailList.add(detailMap);  }  map.put(**"details"**,detailList);  **return** map; } |

注意：getOrderInfo(orderId); 方法中一定要根据orderId取得到orderDetail。

|  |
| --- |
| @Override **public** OrderInfo getOrderInfo(String orderId) {  OrderInfo orderInfo = **orderInfoMapper**.selectByPrimaryKey(orderId);   OrderDetail orderDetail = **new** OrderDetail();  orderDetail.setOrderId(orderId);  List<OrderDetail> orderDetailList = **orderDetailMapper**.select(orderDetail);   orderInfo.setOrderDetailList(orderDetailList);  **return** orderInfo; } |

注意：需要在仓库系统中

@Component  
**public class** WareConsumer

类上加个注解

在库存系统中将activemq的配置打开

|  |
| --- |
| **spring.activemq.broker-url**=**tcp://192.168.67.200:61616 spring.activemq.in-memory**=**true spring.activemq.pool.enabled**=**false** |

重启仓库系统

减库存：如果你的商品明细中的商品，在不同的仓库。则减库存会失败！

异常：需要拆单！因为我的数据，在两个仓库中



测试：<http://payment.gmall.com/sendPaymentResult?orderId=98&result=success>

将这个控制器重新访问一下就可以了！

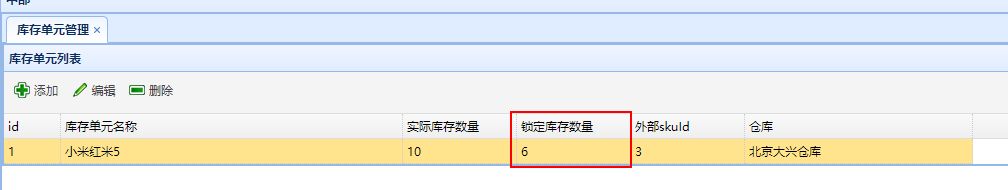
## 4 消费减库存结果

给仓库系统发送减库存消息后，还要接受减库存成功或者失败的消息。

同样根据《库存管理系统接口手册》中【**商品减库结果消息**】的说明完成。消费该消息的消息队列监听程序。

接受到消息后主要做的工作就是更新订单状态。

|  |
| --- |
| @JmsListener(destination = **"SKU\_DEDUCT\_QUEUE"**,containerFactory = **"jmsQueueListener"**) **public void** consumeSkuDeduct(MapMessage mapMessage) **throws** JMSException {  String orderId = mapMessage.getString(**"orderId"**);  String status = mapMessage.getString(**"status"**);  **if**(**"DEDUCTED"**.equals(status)){  **orderService**.updateOrderStatus( orderId , ProcessStatus.***WAITING\_DELEVER***);  }**else**{  **orderService**.updateOrderStatus( orderId , ProcessStatus.***STOCK\_EXCEPTION***);  } } |



分布式事务：activemq 消息流程图

