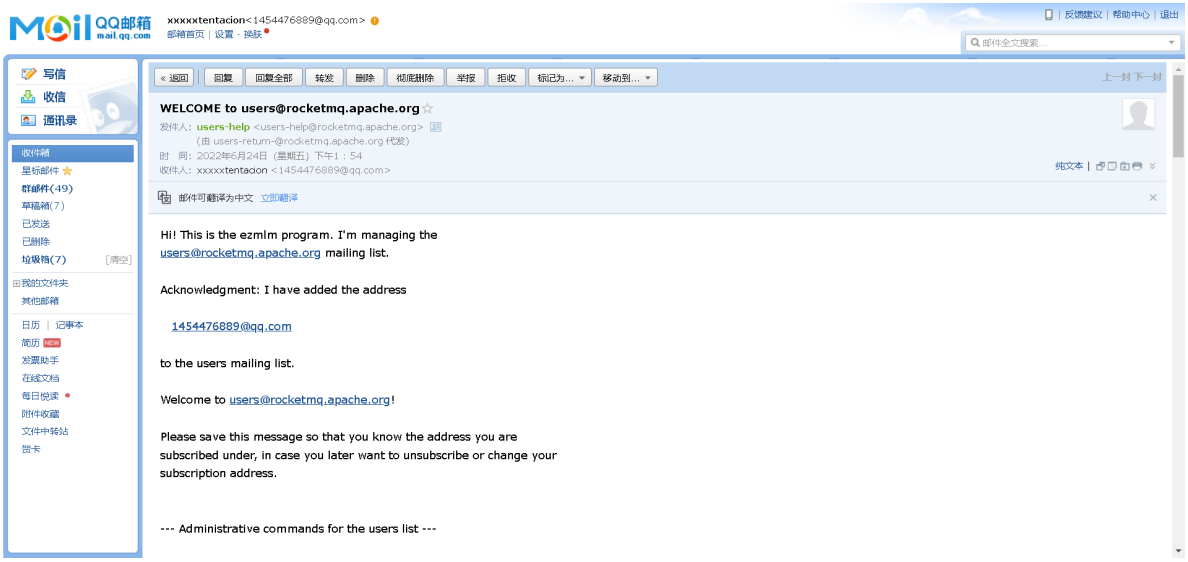

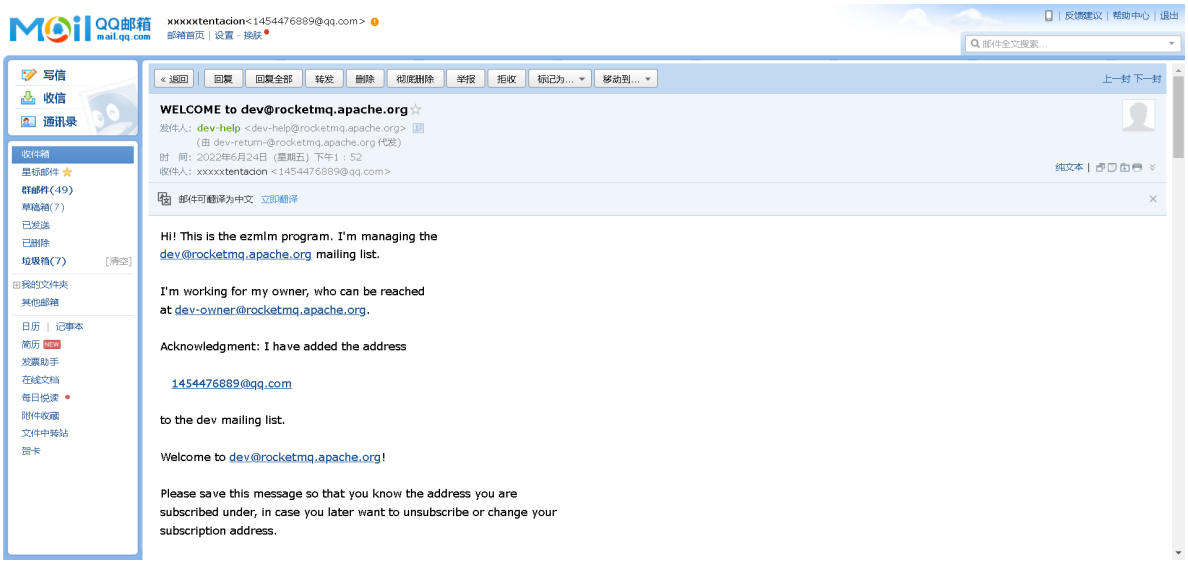

保证消息不丢失

2.1 同步发送

Produce有三种发消息的方式

- 同步发送
- 异步发送
- 单向发送
  由于同步和异步方式均需要Broker返回确认信息，单向发送只管发，不需要Broker返回确认信息，所以单向发送并不知道消息是不是发送成功，单向发送不能保证消息不丢失。

  produce要想发消息时保证消息不丢失，可以采用同步发送的方式去发消息，send消息方法只要不抛出异常，就代表发送成功。发送成功会有多个状态，以下对每个状态进行说明：

SEND_OK：消息发送成功，Broker刷盘、主从同步成功
FLUSH_DISK_TIMEOUT：消息发送成功，但是服务器同步刷盘（默认为异步刷盘）超时（默认超时时间5秒）
FlUSH_SLAVE_TIMEOUT：消息发送成功，但是服务器同步复制（默认为异步复制）到Slave时超时（默

认超时时间5秒)
SLAVE_NOT_AVAILABLE：Broker从节点不存在
注意：同步发送只要返回以上四种状态，就代表该消息在生产阶段消息正确的投递到了RocketMq，没有丢失。但也只是消息投递过程没有丢失，并不代表存储阶段消息不丢失，当出现超时或失败状态时，则会触发默认的2次重试。从生产者角度看，如果消息未能正确的存储在MQ中，或者消费者未能正确的消费到这条消息，都是消息丢失。如果业务要求严格，为保证消息的成功投递和保存，我们可以只取SEND_OK标识消息发送成功，把失败的消息记录到数据库，并启动一个定时任务，扫描发送失败的消息，重新发送5次，直到成功或者发送5次后发送邮件或短信通知人工介入处理。如下代码所示

```
    log.info(发送消息mq  key  发送中........)          //用数据库记录日志：  插入本地消息表：消息：key，消息状态：发送中
    try{

        //发消息
         SendResult   sendResult = send(msg)

       // 状态为SEND_OK标识发送成功
       if(sendResult .getSendStatus() == SEND_OK){
             log.info(发送消息mq  key  发送sucess........)  //用数据库记录日志：更新本地消息表：消息：key，消息状态：发送成功
       }else{
           log.info(发送消息mq  key  发送fail........)  //用数据库记录日志：  更新本地消息表：消息：key，消息状态：发送失败
      }

```

2.2 采用事务消息

这个结论比较容易理解，因为RocketMQ的事务消息机制就是为了保证零丢失来设计的，并且经过阿里的验证，肯定是非常靠谱的。 点击查看事务消息详情！！！ 我们以最常见的电商订单场景为例，来简单分析下事务消息机制如何保证消息不丢失。我们看下下面这个 流程图：

1、为什么要发送个half消息？有什么用?

我们通常是会在订单系统中先完成下单，再发送消息给MQ，如果发送到MQ时，发现MQ宕机了，就会非常尴尬了。而这个half消息是 在订单系统进行下单操作前发送，并且对下游服务的消费者是不可见的。那这个消息的作用更多的体现在确认RocketMQ的服务是否正 常。相当于嗅探下RocketMQ服务是否正常，并且通知RocketMQ，我马上就要发一个很重要的消息了，你做好准备。

2.half消息如果写入失败了怎么办?

如果half消息如果写入失败，我们就可以认为MQ的服务是有问题的，这时，就不能通知下游服务了。我们可以在下单时给订单一个 状态标记入库，然后等待MQ服务正常后再重新下单通知下游服务。可以使用定时任务实现!

3.本地事务中订单系统写数据库失败了怎么办?

如果没有使用事务消息，我们只能判断下单失败，抛出了异常，那就不往MQ发消息了，这样至少保证不会对下游服务进行错误的通 知。但是这样的话，如果过一段时间可以正常下单了，那么这个异常单就丢失了，需要用户人为的重新点击。当然，也可以设计另外 的 补偿机制，例如：将这次异常下单数据缓存起来，再启动一个线程定时尝试往数据库写，省去用户重复点击的操作。

可以有一种更优雅的方案，就是使用事务消息机制：

如果下单时，写数据库失败(可能是数据库崩了，需要等一段时间才能恢复)。那我们可以另外找个地方把订单消息先缓存起来(Redis、 文本或者其他方式)，然后给RocketMQ返回一个UNKNOWN状态。

这样RocketMQ就会过一段时间来回查事务状态。我们就可以在回查事务状态时尝试把已保存至redis缓存的订单数据再次写入数据库， 如果数据库这时候已经恢复了，那就能完整正常的下单，再继续后面的业务，如果没有恢复，会进行多次回查，直到数据库恢复，这样 这个订单的消息就不会因为数据库临时崩了而丢失。如果多次回查数据库依旧没有恢复，再采用其他补偿机制。

如果不使用事务消息，可以有以下方案：

最简单的方式是启动一个定时任务，每隔一段时间扫描订单表，比对未支付的订单的下单时间，将超过时间的订单回收。这种方式显然 是有很大问题的，需要定时扫描很庞大的一个订单信息，这对系统是个不小的压力。

使用RocketMQ的延迟消息机制，在订单生成时，往MQ发一个延迟1分钟的消息，消息内容为订单号，消费到这个消息后去检查订单的 支付状态，如果订单已经支付，就往下游发送通知。如果没有支付，就再发一个延迟1分钟的消息。最终在第十个消息时把订单回收。 这个方案就不用对全部的订单表进行扫描，而只需要每次处理一个单独的延时消息。

如果使用事务消息，可以更优雅的处理订单超时问题，可以用事务消息的状态回查机制来替代定时的任务。在下单时，给Broker返回一 个UNKNOWN的未知状态。而在状态回查的方法中去查询订单的支付状态。这样整个业务逻辑就会简单很多。我们只需要配RocketMQ 中的事务消息回查次数(默认15次)和事务回查间隔时间(messageDelayLevel)，就可以更优雅的完成这个支付状态检查的需求。

5. 使用事务消息可以解决 生产者 >> Broker >> 消费者 之间的分布式事务吗？

   整体来说，在订单这个场景下，消息不丢失的问题实际上就还是转化成了下单这个业务与下游服务的业务的分布式事务一致性问题。而事务一致性问题一直以来都是一个非常复杂的问题。

   RocketMQ的事务消息机制，实际上只保证了整个事务消息的一半，他保证的是订单系统下单和发消息这两个事件的事务一致性，而对下游服务的事务并没有保证。但是即便如此，也是分布式事务的一个很好的降级方案。目前来看，也是业内最好的降级方案。

3. Broker如何保证接收到的消息不会丢失

   使用同步刷盘机制：

   同步刷盘机制，只有在消息真正持久化至磁盘后，RocketMQ的Broker端才会真正地返回给Producer端一个成功的ACK响应，保证了消息可靠性，但影响了性能。异步刷盘则能够充分利用OS的PageCache的优势，只要消息写入PageCache即可将成功的ACK返回给Producer端，消息刷盘采用后台异步线程提交的方式进行，提高了MQ的性能和吞吐量，但是可能会丢消息。点击查看配置方式

   使用同步复制机制：

   同步复制是等Master和Slave都写入消息成功后才反馈给客户端写入成功的状态。在同步复制下，如果Master节点故障，Slave上 有全部的数据备份，这样容易恢复数据。但是同步复制会增大数据写入的延迟，降低系统的吞吐量。异步复制是只要master写入消 息成功，就反馈给客户端写入成功的状态。速度快，同样可能丢消息！

4. 消费者如何确保拉取到的消息被成功消费？

   正常情况下，rocketMq拉取消息后，执行业务逻辑。一旦执行成功，将会返回一个ACK响应给Broker，这时MQ就会修改offset，将该消息标记为已消费，不再往其他消费者推送消息。如果出现消费超时(默认15分钟)、拉取消息后消费者服务宕机等消费失败的情况，此时的Broker由于没有等到消费者返回的ACK，会向同一个消费者组中的其他消费者间隔性的重发消息，直到消息返回成功（默认是重复发送16次，若16次还是没有消费成功，那么该消息会转移到死信队列,人工处理或

是单独写服务处理这些死信消息)
在Broker的这种重新推送机制下，正常同步消费是不会丢消息的，但是异步消费就不一定，比如下面这种情况：

```
 DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("please_rename_unique_group_name_4");
        consumer.registerMessageListener(new MessageListenerConcurrently() {
            @Override
            public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs,

 ConsumeConcurrentlyContext context) {
                new Thread(){
                    public void run(){
                        //处理业务逻辑
                        System.out.printf("%s Receive New Messages: %s %n", Thread.currentThread().getName(), msgs);
                    }
                };
                return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
            }
        });

```

> 在监听器中，新启动了一个线程，异步处理业务逻辑。监听器返回了消费成功 `ConsumeConcurrentlyStatus.CONSUME_SUCCESS` ，但是异步处理线程业务却失败了，那么这个消息并没有被消费，但是在 Broker中已经标记为消费过了，导致消息丢失，只不过这里的丢失是业务代码处理不合理造成的。

所以，为了保证消费的可靠性，消费端尽量不要使用异步消费机制，如果你一定要使用异步，那么就记录消费失败的日志入库，开启其他线程重新消费！

消费幂等性如何保证?

由于存在重试机制（消息重新发送），在消费时需要做幂等性校验，防止重复消费。解决方案如下：

第一次消费时，可以把消息的唯一ID，或者业务id（orderId）存在redis中（key = orderId, value = 1）
如果发生了重试，先去redis中查一下该唯一ID是否存在，如果存在，则为重复消费！不再处理

5. Broker宕机后如何防止消息丢失?
   当NameServer全部挂了(注意是全部，只要有一个存活就是正常)，或者Broker宕机了。在这种情况下，RocketMQ相当于整个服务都不可用了，那他本身肯定无法给我们保证消息不丢失了。只能自己设计一个降级方案来处理这个问题了。

总结

综上所述：RocketMQ处理消息零丢失的方案要考虑以下几点：

1. 生产者使用同步发送，或者发送事务消息
2. Broker配置同步刷盘 + 同步复制

3. 消费者不要使用异步消费
4. 整个MQ挂了之后准备降级方案