

# RocketMQ实战（三）：分布式事务

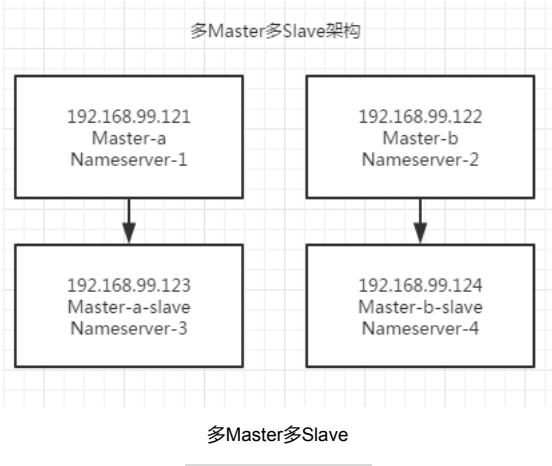


张丰哲 (/u/cb569cce501b) ✓ 已关注  
2017.04.25 22:40 字数 2902 阅读 3703 评论 21 喜欢 37 赞赏 1  
(/u/cb569cce501b)

接《RocketMQ实战（一）》  
(<http://www.jianshu.com/p/3afd610a8f7d>)，《RocketMQ实战（二）》(<http://www.jianshu.com/p/790d6bc4a1c1>)》，本篇博客主要讨论的话题是：顺序消费、RMQ在分布式事务中的应用等。

## 关于多Master多Slave的说明

由于在之前的博客中已经搭建了双Master，其实多Master多Slave大同小异，因此这里并不会一步步的演示搭建多Master多Slave，而是从思路，分析下重点应该注意的配置项。



- 第一，这四台机器，对外是一个统一的整体，是一个rocketmq cluster，因此需要brokerClusterName保持一致

第二，123机器是121的从，124机器是122的从，如何在配置中体现？主和从的brokerName需要保持一致，另外brokerId标示了谁是主，谁是从（brokerId=0的就是主，大于0的就是从）

第三，注意namesrvAddr的地址是4台NameServer

第四，配置项中brokerRole需要指明ASYNC\_MASTER（异步复制Master）or SYNC\_MASTER（同步双写Master）or SLAVE（从）

第五，和以前的多Master启动方式一致，先启动4台Namesrv，然后用指定配置文件的方式启动Master/Slave即可

第六，多Master多Slave的好处在于，即便集群中某个broker挂了，也可以继续消费，保证了实时性的高可用，但是并不是说某个master挂了，slave就可以升级master，开源版本的rocketmq是不可以的。也就是说，在这种情况下，slave只能提供读的功能，将失去消息负载的能力。

^

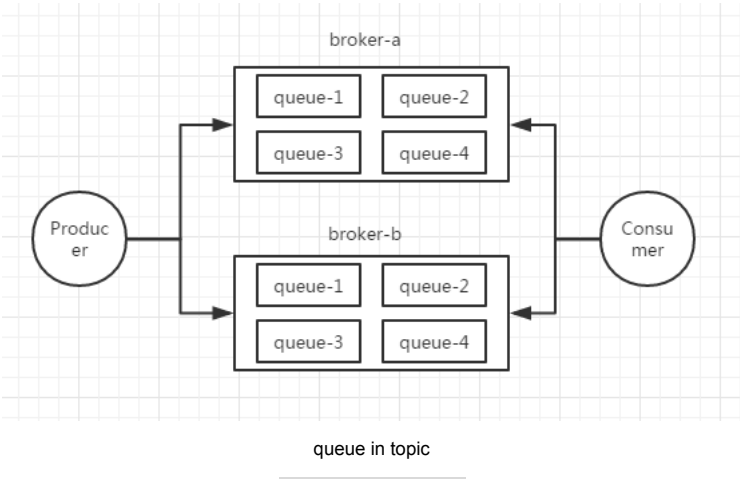
+

🔖

🔗

## Queue in Topic

对于RocketMQ而言，Topic只是一个逻辑上的概念，真正的消息存储其实是在Topic中的Queue中。想一想，为什么RocketMQ要这么设计呢？其实是为了消息的顺序消费，后文中将为大家介绍。



```
public DefaultMQProducer() { this("DEFAULT_PRODUCER", (RPCHook)null); }

public DefaultMQProducer(String producerGroup) {
    this(producerGroup, (RPCHook)null);
}

public DefaultMQProducer(RPCHook rpcHook) { this("DEFAULT_PRODUCER", rpcHook); }

public DefaultMQProducer(String producerGroup, RPCHook rpcHook) {
    this.createTopicKey = "TBW102";
    this.defaultTopicQueueNums = 4;
    this.sendMessageTimeout = 3000;
    this.compressMsgBodyOverHowmuch = 4096;
    this.retryTimesWhenSendFailed = 2;
    this.retryAnotherBrokerWhenNotStoreOK = false;
    this.maxMessageSize = 131072;
    this.unitMode = false;
    this.producerGroup = producerGroup;
    this.defaultMQProducerImpl = new DefaultMQProducerImpl(this, rpcHook);
}
```

默认一个Topic中4个队列

```
#在发送消息时，自动创建服务器不存在的topic，默认创建的队列数
defaultTopicQueueNums=4
```

配置文件中指定

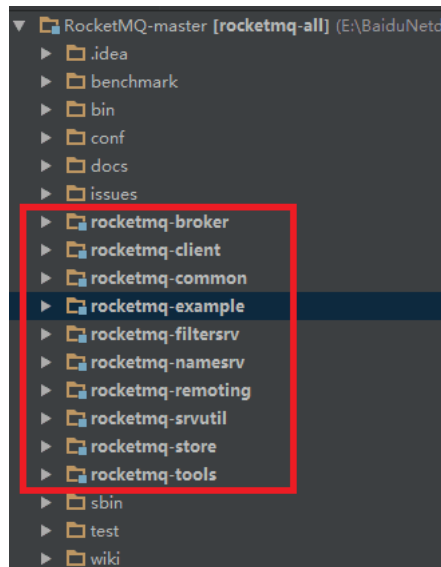
## 初步认识RocketMQ的核心模块

^

+

🔖

🔗



rocketmq模块

**rocketmq-broker**：接受生产者发来的消息并存储（通过调用rocketmq-store），消费者从这里取得消息。

**rocketmq-client**：提供发送、接受消息的客户端API。

**rocketmq-namesrv**：NameServer，类似于Zookeeper，这里保存着消息的TopicName，队列等运行时的元信息。（有点NameNode的味道）

**rocketmq-common**：通用的一些类，方法，数据结构等

**rocketmq-remoting**：基于Netty4的client/server + fastjson序列化 + 自定义二进制协议

**rocketmq-store**：消息、索引存储等

**rocketmq-filter-srv**：消息过滤器Server，需要注意的是，要实现这种过滤，需要上传代码到MQ！【一般而言，我们利用Tag足以满足大部分的过滤需求，如果更灵活更复杂的过滤需求，可以考虑filter-srv组件】

**rocketmq-tools**：命令行工具

## Order Message

RocketMQ提供了3种模式的Producer：

NormalProducer（普通）、OrderProducer（顺序）、TransactionProducer（事务）

在前面的博客当中，涉及的都是NormalProducer，调用传统的send方法，消息是无序的。接下来，我们来看看顺序消费。模拟这样一个场景，如果一个用户完成一个订单需要3条消息，比如订单的创建、订单的支付、订单的发货，很显然，同一个用户的订单消息必须要顺序消费，但是不同用户之间的订单可以并行消费。

生产者端代码示例：

```
DefaultMQProducer producer = new DefaultMQProducer("OrderProducer");
producer.setNamesrvAddr("192.168.99.121:9876");
producer.start();

String[] tags = new String[] { "createTag", "payTag", "sendTag" };

for(int orderId = 1 ; orderId <= 10 ; orderId++){ //订单消息

    for(int type = 0; type < 3 ; type++){ //每种订单分为 创建订单消息/支付点单/发货订单

        Message msg =
            new Message("OrderTopic", tags[type % tags.length], orderId + ":" + type,
                (orderId + ":" + type).getBytes());

        SendResult sendResult = producer.send(msg, new MessageQueueSelector() {
            @Override
            public MessageQueue select(List<MessageQueue> mqs, Message msg, Object arg) {
                Integer id = (Integer) arg;
                int index = id % mqs.size();
                return mqs.get(index);
            }
        }, orderId);

        System.out.println(sendResult);
    }
}

producer.shutdown();
```

顺序消息模式

注意，一个Message除了Topic/Tag外，还有Key的概念。

上图的send方法不同于以往，有一个MessageQueueSelector，将用于指定特定的消息发往特定的队列当中！

```
consumer.registerMessageListener(new MessageListenerOrderly() {

    @Override
    public ConsumeOrderlyStatus consumeMessage(List<MessageExt> msgs, ConsumeOrderlyContext context) {

        try {
            //模拟业务处理消息的时间
            Thread.sleep(new Random().nextInt(1000));

            System.out.println(new String(msgs.get(0).getBody(), "UTF-8"));
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (UnsupportedEncodingException e) {
            e.printStackTrace();
        }

        return ConsumeOrderlyStatus.SUCCESS;
    }
});

consumer.start();

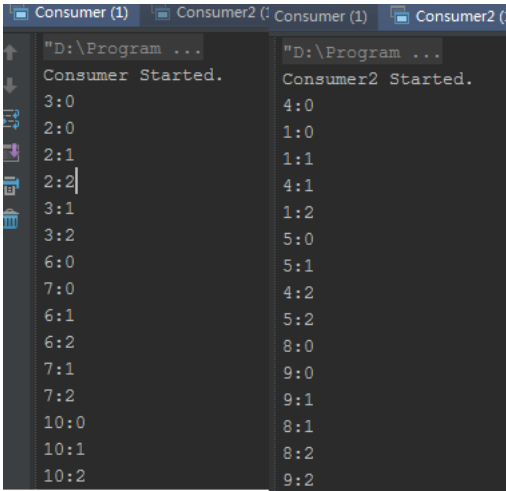
System.out.println("Consumer Started.");
```

顺序消费

注意在以前普通消费消息时设置的回调是MessageListenerConcurrently，而顺序消费的回调设置是MessageListenerOrderly。

当我们启动2个Consumer进行消费时，可以观察到：





多个消费者消费的结果

可以观察得到，虽然从全局上来看，消息的消费不是有序的，但是每一个订单下的3条消息是顺序消费的！

其实，如果需要保证消息的顺序消费，那么很简单，首先需要做到一组需要有序消费的消息发往同一个broker的同一个队列上！其次消费者端采用有序Listener即可。

这里，RocketMQ底层是如何做到消息顺序消费的，看一看源码你就能大概了解到，至少来说，在多线程消费场景下，一个线程只去消费一个队列上的消息，那么自然就保证了消息消费的顺序性，同时也保证了多个线程之间的并发性。也就是说其实broker并不能完全保证消息的顺序消费，它仅仅能保证的消息的顺序发送而已！

关于多线程消费这块，RocketMQ早就替我们想好了，这样设置即可：

```
//消费线程数最小数量 默认10
consumer.setConsumeThreadMin(10);
//消费线程数最大数量 默认20
consumer.setConsumeThreadMax(20);
```

消费多线程设置

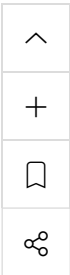
想一想，在ActiveMQ中，我们如果想实现并发消费的话，恐怕还得搞个线程池提交任务吧，RocketMQ让我们的工作变得简单！

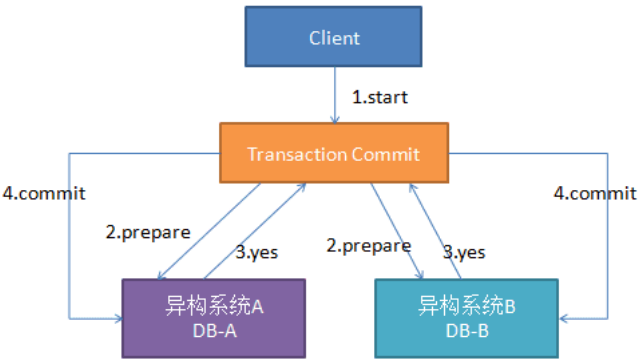
## Transaction Message

在说事务消息之前，我们先来说说分布式事务的那些事！

什么是分布式事务，我的理解是一半事务。怎么说，比如有2个异构系统，A异构系统要做T1，B异构系统要做T2，要么都成功，要么都失败。

要知道异构系统，很显然，不在一个数据库实例上，它们往往分布在不同物理节点上，本地事务已经失效。





2阶段提交

2阶段提交协议，Two-Phase Commit，是处理分布式事务的一种常见手段。  
2PC，存在2个重要角色：事务协调器（TC），事务执行者。

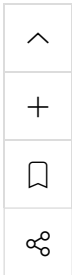
2PC，可以看到节点之间的通信次数太多了，时间很长！时间变长了，从而导致，事务锁定的资源时间也变长了，造成资源等待时间变长！在高并发场景下，存在严重的性能问题！

下面，我们来看看MQ在高并发场景下，是如何解决分布式事务的。

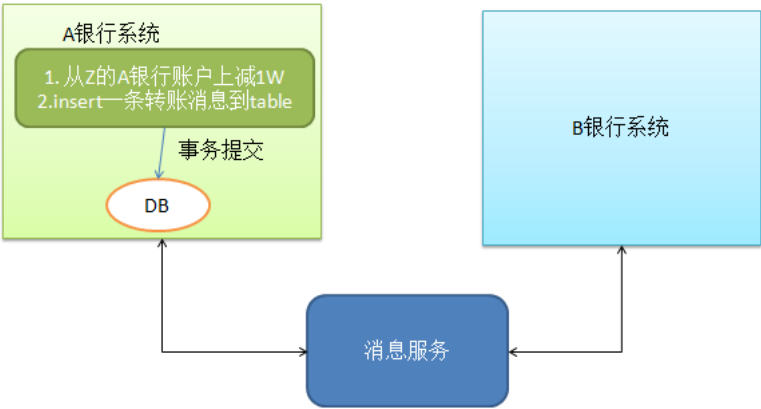
考虑生活中的场景：

我们去北京庆丰包子铺吃炒肝，先去营业员那里付款（Action1），拿到小票（Ticket），然后去取餐窗口排队拿炒肝（Action2）。思考2个问题：第一，为什么不在付款的同时，给顾客炒肝？如果这样的话，会增加处理时间，使得后面的顾客等待时间变长，相当于降低了接待顾客的能力（降低了系统的QPS）。第二，付了款，拿到的是Ticket，顾客为什么会接受？从心理上说，顾客相信Ticket会兑现炒肝。事实上也是如此，就算在最后炒肝没了，或者断电断水（系统出现异常），顾客依然可以通过Ticket进行退款操作，这样都不会有什么损失！（虽然这么说，但是实际上包子铺最大化了它的利益，如果炒肝真的没了，浪费了顾客的时间，不过顾客顶多发发牢骚，最后接受）

生活已经告诉我们处理分布式事务，保证数据最终一致性的思路！这个Ticket（凭证）其实就是消息！



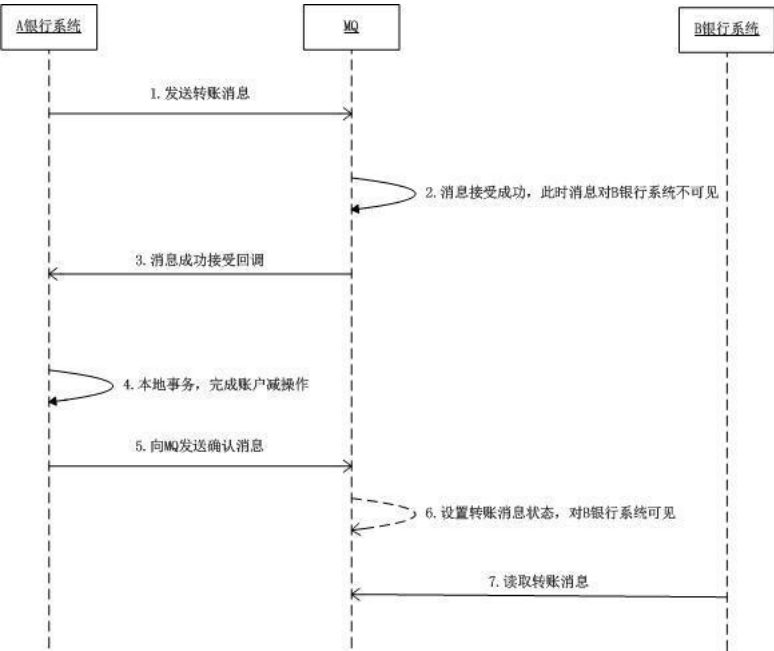
模拟账号z从A银行向B银行转账1W



业务和消息生成耦合在一起

业务操作和消息的生成耦合在一起，保证了只要A银行的账户发生扣款，那么一定会生成一条转账消息。只要A银行系统的事务成功提交，我们可以通过实时消息服务，将转账消息通知B银行系统，如果B银行系统回复成功，那么A银行系统可以在table中设置这条转账消息的状态。

这样耦合的方式，从架构上来看，就有点不太优雅，而且存在一些问题。比如说，消息的存储实质上是在A银行系统中的，如果A银行系统出了问题，将导致无法转账。如果解耦，将消息独立出来呢？



业务和消息解耦

如上图所示，消息数据独立存储，业务和消息解耦，实质上消息的发送有2次，一条是转账消息，另一条是确认消息。

到这里，我们先来看看基于RocketMQ的代码：

^

+

🔖

🔗

```

TransactionCheckListener transactionCheckListener = new TransactionCheckListenerImpl();
TransactionMQProducer producer = new TransactionMQProducer("transactionProduceGroup");
producer.setNamesrvAddr("192.168.99.121:9876");
producer.setTransactionCheckListener(transactionCheckListener);
producer.start();

TransactionExecutorImpl tranExecutor = new TransactionExecutorImpl();

try {
    Message msg =
        new Message("TransactionTopic", "Tag", "KEY1",
            ("Hello RocketMQ 1").getBytes());

    Message msg2 =
        new Message("TransactionTopic", "Tag", "KEY2",
            ("Hello RocketMQ 2").getBytes());

    SendResult sendResult = producer.sendMessageInTransaction(msg, tranExecutor, null);
    System.out.println(new Date() + "msg1:" + sendResult);

    sendResult = producer.sendMessageInTransaction(msg2, tranExecutor, null);
    System.out.println(new Date() + "msg2:" + sendResult);
}
catch (MQClientException e) {
    e.printStackTrace();
}
producer.shutdown();

```

生产者示例代码

生产者这里用到是：TransactionMQProducer。

这里涉及到2个角色：本地事务执行器（代码中的TransactionExecutorImpl）、服务器回查客户端Listener（代码中的TransactionCheckListener）。

如果事务消息发送到MQ上后，会回调 本地事务执行器；但是此时事务消息是prepare状态，对消费者还不可见，需要 本地事务执行器 返回RMQ一个确认消息。

```

/**
 * 执行本地事务
 */
public class TransactionExecutorImpl implements LocalTransactionExecutor {

    @Override
    public LocalTransactionState executeLocalTransactionBranch(final Message msg, final Object arg) {

        try{
            //DB操作 应该带上事务 service -> dao
            //如果数据库操作失败 需要回滚 同时 返回RMQ一个失败消息 意味着消费者将无法消费到这条失败的消息
            //如果成功 需要告诉RMQ一个成功的消息,意味着消费者将读取到消息
            //arg就是attachment
            if(new Random().nextInt(3) == 2){
                int a = 1 / 0;
            }

            System.out.println(new Date() + "本地事务执行成功,发送确认消息");

        }catch(Exception e){

            System.out.println(new Date() + "本地事务执行失败");
            return LocalTransactionState.ROLLBACK_MESSAGE;

        }

        //这种消息意味着事务消息将不会被消费者读取到:
        //LocalTransactionState.ROLLBACK_MESSAGE LocalTransactionState.UNKNOWN
        return LocalTransactionState.COMMIT_MESSAGE;
    }
}

```

本地事务执行器



事务消息是否对消费者可见，完全由事务返回给RMQ的状态码决定（状态码的本质也是一条消息）。

```
/**
 * 未决事务，服务器回查客户端
 */
public class TransactionCheckListenerImpl implements TransactionCheckListener {

    @Override
    public LocalTransactionState checkLocalTransactionState(MessageExt msg) {

        System.out.println("server checking TrMsg " + msg.toString());

        //由于RMQ迟迟没有收到消息的确认消息,因此主动询问这条prepare消息,是否正常?
        //可以查询数据库看这条消息是否已经处理

        return LocalTransactionState.COMMIT_MESSAGE;
    }
}
```

回查Listener

Consumer (2) TransactionProducer

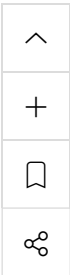
"D:\Program ...  
Consumer Started.  
Hello RocketMQ 1

#InTotalToday	#OutTotalToday
4	1

Consumer (2) TransactionProducer

"D:\Program ...  
Tue Apr 25 21:57:09 CST 2017本地事务执行成功,发送确认消息  
Tue Apr 25 21:57:09 CST 2017msg1:SendResult [sendStatus=SEND\_OK, msgId=C0A8637900002A9F00000000000000  
Tue Apr 25 21:57:09 CST 2017本地事务执行失败  
Tue Apr 25 21:57:09 CST 2017msg2:SendResult [sendStatus=SEND\_OK, msgId=C0A8637900002A9F000000000000001  
Process finished with exit code 0

运行结果



生产者发送了2条消息给RMQ，有一条本地事务执行成功，有一条本地事务执行失败。

2条业务消息 + 2条确认消息 因此是4条；

注意到消费者只消费了一条数据，就是只有告诉RMQ本地事务执行成功的那条消息才会被消费！因此是1条！

但是，注意到本地事务执行失败的消息，RMQ并没有check listener？这是为什么呢？因为RMQ在3.0.8的时候还是支持check listener回查机制的，但是到了3.2.6的时候将事务回查机制“阉割”了！

那么3.0.8的时候，RMQ是怎么做事务回查的呢？看一看源码，你会知道，其实事务消息开始是prepare状态，然后RMQ会将其持久化到MySQL当中，然后如果收到确认消息，就删除掉这条prepare消息，如果迟迟收不到确认消息，那么RMQ会定时的扫描prepare消息，发送给produce group进行回查确认！

到这里，问题来了，要知道3.2.6版本，没有回查机制了，会存在问题么？

当然会存在问题！假设，我们发送一条转账事务消息给RMQ，成功后回调本地事务，DB减操作成功，刚准备给RMQ一个确认消息，此时突然断电，或者网络抖动，使得这条确认消息没有发送出去。此时RMQ中的那条转账事务消息，始终处于prepare状态，消费者读取不到，但是却已经完成一方的账户资金变动！！！

既然，RMQ3.2.6版本不为我们进行回查，那么只能由我们自己完成了。具体怎么做呢，咱们下期再来分析~

see u , good night~

📖 日记本 (/nb/10261827) 举报文章 © 著作权归作者所有



张丰哲 (/u/cb569cce501b) ♂

写了 63893 字，被 2144 人关注，获得了 1674 个喜欢 (/u/cb569cce501b)

✓ 已关注

资深Java工程师 51CTO博客【2014-2016】：http://zhangfengzhe.blog.51cto.com/

好好学习，天天赞赏~

赞赏支持




♡ 喜欢 | 37



更多分享

(http://cwb.assets.jianshu.io/notes/images/1149029)



写下你的评论...

21条评论 只看作者 按喜欢排序 按时间正序 按时间倒序



李懷繩 (/u/ec65ceb8d9f8)  
9楼 · 2017.10.07 10:34

(/u/ec65ceb8d9f8)  
RocketMQ实战(三):分布式事务 - , 写的不错不错, 收藏了。

推荐下, 分布式队列中间件 RocketMQ 源码解析 14 篇 : <http://c7.gg/rSTs>  
(<http://c7.gg/rSTs>)

炫

👍 14人赞    💬 回复

景轩顶 (/u/fe1421e713cc) : 写的蛮用心的, 希望多多坚持那  
2017.10.07 23:51    💬 回复

张丰哲 (/u/cb569cce501b) : @景轩顶 (/users/fe1421e713cc) 😊  
2017.10.12 11:22    💬 回复

✍️ 添加新评论



landy8530 (/u/36a7d3a994ac)  
3楼 · 2017.06.18 00:04

(/u/36a7d3a994ac)  
你好, 我现在搭建的3.2.6版本中, 根本进不去TransactionExecuterImpl中  
executelocalTransactionBranch的方法体了。是不是阿里已经又升级了这个版本的分布  
式事务处理? 阉割得更彻底了? 本地事务执行器(代码中的  
TransactionExecuterImpl)、服务器回查客户端Listener(代码中的  
TransactionCheckListener)都进不去了。

👍 1人赞    💬 回复

张丰哲 (/u/cb569cce501b) : RocketMQ3.2.6已经阉割了事务处理了, 😊  
2017.06.18 17:32    💬 回复

landy8530 (/u/36a7d3a994ac) : @张丰哲 (/users/cb569cce501b) 因为16年的时候, 还能执行部  
分代码的  
2017.06.19 00:01    💬 回复

张丰哲 (/u/cb569cce501b) : @landy8530 (/users/36a7d3a994ac) 由于2017我司自主研发了  
MQ, 切换了, 没有再关注后续RocketMQ呢。至少来说, 2016的时候, 3.2.6版本对事务支持不  
完整, 如果事务消息没有确认, 服务器不会回查的, 需要自己保证事务一致性。  
2017.06.19 07:46    💬 回复

✍️ 添加新评论    还有2条评论, 展开查看



风车车 (/u/d53b94c2c297)  
2楼 · 2017.04.30 11:42

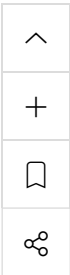
(/u/d53b94c2c297)  
3.0.8的源码在哪里可以找到呢

👍 赞    💬 回复

张丰哲 (/u/cb569cce501b) : 我这边手头倒是有一份源码分享给你 :  
<https://pan.baidu.com/s/1pKGPpV1> (<https://pan.baidu.com/s/1pKGPpV1>)  
2017.05.02 11:06    💬 回复

之\_7dff (/u/9409041379cc) : RocketMQ-console 监控台现在怎么去搭建呢  
2017.06.01 20:55    💬 回复

张丰哲 (/u/cb569cce501b) : @之\_7dff (/users/9409041379cc) 在第一篇ROCKETMQ中有涉及,  
😊  
2017.06.01 21:43    💬 回复



添加新评论



gl328518397 (/u/761eebae5903)

4楼 · 2017.08.13 17:40

(/u/761eebae5903)

请问 RMQ 是 定期 check listener 处于 prepare 状态的消息么？  
会对 “ROLLBACK\_MESSAGE” 的消息 做 check listener 么？

赞 回复

张丰哲 (/u/cb569cce501b)：和RMQ的版本有关系，在早期版本当中，RMQ是会定时的扫描 prepare消息，发送给produce group进行回查确认的。

2017.08.15 22:01 回复

gl328518397 (/u/761eebae5903)：@张丰哲 (/users/cb569cce501b) 好的，多谢~明白了。

2017.08.16 08:11 回复

添加新评论



夕阳的告别诗 (/u/Ode75fab58c3)

6楼 · 2017.09.28 11:07

(/u/Ode75fab58c3)

如果一个事务要分别调用两个银行的扣款呢。这种情况怎么处理

赞 回复

张丰哲 (/u/cb569cce501b)：这个时候，只能“拆”了，可以借助MQ来完成。

2017.10.07 20:30 回复

科比24号先生 (/u/b43014ddb684)：比如支付回调，需要调用订单service修改订单状态，优惠券service修改该用户优惠券状态，和发送支付成功短信。这种场景怎么使用rmq保证事务一致性

2017.10.20 10:26 回复

添加新评论



科比24号先生 (/u/b43014ddb684)

10楼 · 2017.10.20 10:26

(/u/b43014ddb684)

比如支付回调，需要调用订单service修改订单状态，优惠券service修改该用户优惠券状态，和发送支付成功短信。这种场景怎么使用rmq保证事务一致性

赞 回复

张丰哲 (/u/cb569cce501b)：首先来说，订单/优惠券 是在一个库么？

如果在一个库，那就放在一个事务中，成功后，在发送支付成功短信。

如果不在一个库，那么建议使用异步定时任务扫描表的状态，然后进行相应的动作。

当然，也可以利用RMQ的事务消息来做，不过涉及的生产、消费的步骤太多。

2017.10.22 14:32 回复

添加新评论

被以下专题收入，发现更多相似内容

+ 收入我的专题



Java学习笔记 (/c/04cb7410c597?)






utm\_source=desktop&utm\_medium=notes-included-collection)



程序员 (/c/NEt52a?utm\_source=desktop&utm\_medium=notes-included-

collection)



-  Java 杂谈 (/c/0b39448c4e08?utm\_source=desktop&utm\_medium=notes-included-collection)
-  java进阶干货 (/c/addfce4ca518?utm\_source=desktop&utm\_medium=notes-included-collection)
-  高并发，分布式事务 (/c/6fd7fda0a677?utm\_source=desktop&utm\_medium=notes-included-collection)
-  RocketMQ (/c/613c1ca3873c?utm\_source=desktop&utm\_medium=notes-included-collection)
-  MQ (/c/0c6b140b87f8?utm\_source=desktop&utm\_medium=notes-included-collection)
- 展开更多 ▾

^

+

🔖

🔗