# 核心概念

* **生产者（Producer）：**消息发送方，将业务系统中产生的消息发送到brokers（brokers可以理解为消息代理，生产者和消费者之间是通过brokers进行消息的通信），rocketmq提供了以下消息发送方式：同步、异步、单向。
* **生产者组（Producer Group）：**相同角色的生产者被归为同一组，比如通常情况下一个服务会部署多个实例，这多个实例就是一个组，生产者分组的作用只体现在消息回查的时候，即如果一个生产者组中的一个生产者实例发送一个事务消息到broker后挂掉了，那么broker会回查此实例所在组的其他实例，从而进行消息的提交或回滚操作。
* **消费者（Consumer）：**消息消费方，从brokers拉取消息。站在用户的角度，有以下两种消费者。
* **主动消费者（PullConsumer）：**从brokers拉取消息并消费。
* **被动消费者（PushConsumer）：**内部也是通过pull方式获取消息，只是进行了扩展和封装，并给用户预留了一个回调接口去实现，当消息到底的时候会执行用户自定义的回调接口。
* **消费者组（Consumer Group）：**和生产者组类似。其作用体现在实现消费者的负载均衡和容错，有了消费者组变得异常容易。需要注意的是：同一个消费者组的每个消费者实例订阅的主题必须相同。
* **主题（Topic）：**主题就是消息传递的类型。一个生产者实例可以发送消息到多个主题，多个生产者实例也可以发送消息到同一个主题。同样的，对于消费者端来说，一个消费者组可以订阅多个主题的消息，一个主题的消息也可以被多个消费者组订阅。
* **消息（Message）：**消息就像是你传递信息的信封。每个消息必须指定一个主题，就好比每个信封上都必须写明收件人。
* **消息队列（Message Queues）：**在主题内部，逻辑划分了多个子主题，每个子主题被称为消息队列。这个概念在实现最大并发数、故障切换等功能上有巨大的作用。
* **标签（Tag）：**标签，可以被认为是子主题。通常用于区分同一个主题下的不同作用或者说不同业务的消息。同时也是避免主题定义过多引起性能问题，通常情况下一个生产者组只向一个主题发送消息，其中不同业务的消息通过标签或者说子主题来区分。
* **消息代理（Broker）：**消息代理是RockerMQ中很重要的角色。它接收生产者发送的消息，进行消息存储，为消费者拉取消息服务。它还存储消息消耗相关的元数据，包括消费群体，消费进度偏移和主题/队列信息。
* **命名服务（Name Server）：**命名服务作为路由信息提供程序。生产者/消费者进行主题查找、消息代理查找、读取/写入消息都需要通过命名服务获取路由信息。
* **消息顺序（Message Order）：**当我们使用DefaultMQPushConsumer时，我们可以选择使用“orderly”还是“concurrently”。
  + **orderly：**消费消息的有序化意味着消息被生产者按照每个消息队列发送的顺序消费。如果您正在处理全局顺序为强制的场景，请确保您使用的主题只有一个消息队列。注意：如果指定了消费顺序，则消息消费的最大并发性是消费组订阅的消息队列数。
  + **concurrently：**当同时消费时，消息消费的最大并发仅限于为每个消费客户端指定的线程池。注意：此模式不再保证消息顺序。

# 优势

1、rocketmq原生支持分布式

2、rocketmq可以保证严格的消息顺序

3、rocketmq可以提供亿级消息堆积能力，同时依然保持写入低延迟

4、丰富的消息拉取模式（push/pull）

5、消息持久化

6、定时消息

7、事务消息

# 消费模式

RocketMQ有两种消费模式：BROADCASTING广播模式，CLUSTERING集群模式，默认的是集群消费模式。

设置模式：

consumer.setMessageModel(MessageModel.BROADCASTING)

// 如果非第一次启动，那么按照上次消费的位置继续消费

consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME\_FROM\_FIRST\_OFFSET);

## 广播模式

一条消息被多个consumer消费，即使这些consumer属于同一个ConsumerGroup,消息也会被ConsumerGroup中的每个Consumer都消费一次，广播消费中ConsumerGroup概念可以认为在消息划分方面无意义。

## 集群消费

一个ConsumerGroup中的Consumer实例平均分摊消费消息。例如某个Topic有9条消息，其中一个ConsumerGroup有3个实例（可能是3个进程，或者3台机器），那么每个实例只消费其中部分，消费完的消息不能被其他实例消费。

# 重试机制

## producer端重试

1、网络不可达造成的重试：如果在生产者发送消息到broker或者broker向生产者发送结果时，因为网络原因，生产者没有获取成功响应。

2、broker存储过程中出现异常，会导致生产者重试。

生产者端可以在初始化producer对象时手动设置重试次数以及发送失败超时时间进行发送重试：

producer.setRetryTimesWhenSendFailed(3); //默认是2  
producer.setSendMsgTimeout(6000); //默认是3000

## consumer端重试

1、网络不可达，这种情况和生产者一样，不可达造成的重试（相当于浏览器访问一个请求没有返回200响应码，都算网络不可达）会是无限次的。

2、由于业务处理的原因导致异常的出现，导致给broker的响应不是成功的结果（相当于浏览器请求返回200，成功结果，但是业务code是非成功的code），这种失败时有次数限时的重试，每次重试会延迟发送消费信息，避免短时间内消费端没有解决bug或者服务没有启动，所以有个延迟重试。

对于第二种状态，在consumer端可以在发生异常的时候，获取消息发送次数，根据次数是否达到上限来决定返回消费状态。

消息消费的状态，有2种，一个是成功（CONSUME\_SUCCESS），一个是失败&稍后重试（RECONSUME\_LATER）

If(mssageExt.getReconsumeTimes()==3){

return ConsumeConcurrentlyStatus. CONSUME\_SUCCESS;

}else{

return ConsumeConcurrentlyStatus. RECONSUME\_LATER;

}

# 刷盘模式

## 同步刷盘

在消息到达MQ后，RocketMQ需要将数据持久化，同步刷盘是指数据到达内存之后，必须刷到commitlog日志之后才算成功，然后返回producer数据已经发送成功。

## 异步刷盘

同步刷盘是指数据到达内存之后,返回producer说数据已经发送成功。，然后再写入commitlog日志

# 事务消息

## 案例场景

A银行转账到B银行，保证最终结果一致性；B银行如果到帐失败，则A银行必须回滚。

## 错误方案

有人可能想到了，我可以把“发送消息”这个网络调用和update DB放在同1个事务里面，如果发送消息失败，update DB自动回滚。这样不就保证2个操作的原子性了吗？

这个方案看似正确，其实是错误的，原因有2：

（1）网络的2将军问题：发送消息失败，发送方并不知道是消息中间件真的没有收到消息呢？还是消息已经收到了，只是返回response的时候失败了？

如果是已经收到消息了，而发送端认为没有收到，执行update db的回滚操作。则会导致A账号的钱没有扣，B账号的钱却加了。

（2）把网络调用放在DB事务里面，可能会因为网络的延时，导致DB长事务。严重的，会block整个DB。这个风险很大。

## 正确方案

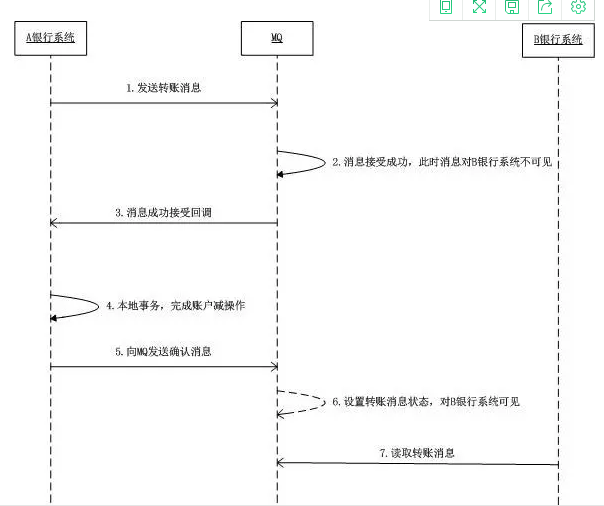
### RocketMQ自带事务消息

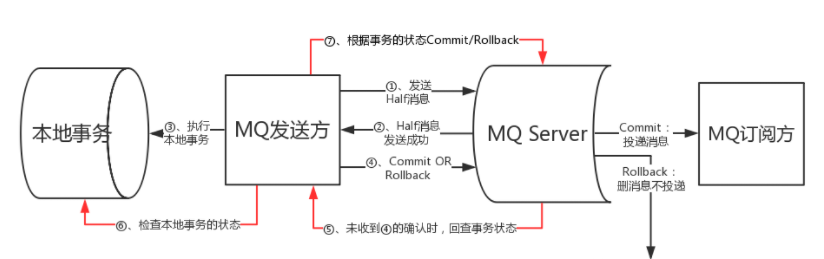
如果RocketMQ版本是在3.2.6以下，可以采用rocketmq自带事务消息。

第一阶段A系统发送Prepared消息到RocketMQ；第二阶段RocketMQ收到消息，此时消息对B系统不可见（即，B系统暂时无法收到该消息）；第三阶段RocketMQ返回该消息的地址给A系统，第四阶段执行本地事物，第五阶段通过第一阶段拿到的地址去访问消息，并修改消息状态。如果第五阶段的确认消息发送失败了，RocketMQ会通过事务回查机制定期扫描消息集群中的事物消息，发现Prepared消息，它会向消息发送者确认，RocketMQ会根据发送端设置的策略来决定是回滚还是继续发送确认消息；第六阶段如果发送确认消息成功且修改状态，则B系统可见；第七阶段B系统读取该消息。这样就保证了消息发送与本地事务同时成功或同时失败

具体逻辑如下：

1、自定义业务类，该类实现TransactionCheckListener接口的checkLocalTransactionState(final MessageExt msg)方法，该方法是对于本地业务执行完毕之后发送事务消息状态时失败导致Broker端的事务消息一直处于PREPARED状态的补救，Broker对于长期处于PREPARED状态的事务消息发起回查请求时，Producer在收到回查事务消息状态请求之后，调用该checkLocalTransactionState方法，该方法的请求参数是之前发送的事务消息，在该方法中根据此前发送的事务消息来检查该消息对应的本地业务逻辑处理的情况，根据处理情况告知Broker该事务消息的最终状态（commit或者rollback）；   
 2、自定义执行本地业务逻辑的类，该类实现LocalTransactionExecuter接口的executeLocalTransactionBranch(final Message msg, final Object arg)方法，在该方法中执行本地业务逻辑，根据业务逻辑执行情况反馈事务消息的状态（commit或者rollback）；   
 3、初始化TransactionMQProducer类，将第1步中的类赋值给TransactionMQProducer.transactionCheckListener变量；设置检查事务状态的线程池中线程的最大值、最小值、队列数等参数值；   
 4、调用TransactionMQProducer.start方法启动Producer，该启动过程与普通的Producer启动区别不大，首先调用DefaultMQProducerImpl.initTransactionEnv()方法初始化检查事务状态的线程池，将第三阶段的事务消息确认的逻辑放入该线程池中执行；然后调用DefaultMQProducer.start()启动Producer；   
 5、构建事务消息Message对象，然后与第2步创建的执行本地业务逻辑类为参数调用TransactionMQProducer.sendMessageInTransaction(Message msg,LocalTransactionExecuter tranExecuter, Object arg)方法。在方法中首先检查TransactionMQProducer.transactionCheckListener变量是否为空，若没有定义（即为空）则直接抛出异常，表示该回查事务消息状态类必须定义；然后调用DefaultMQProducerImpl.sendMessageInTransaction(Message msg, LocalTransactionExecuter tranExecuter, Object arg)方法。在该方法中分为三阶段，大致逻辑如下：   
 5.1）将事务消息Message对象的properties属性中的“TRAN\_MSG”字段设为true，“PGROUP”字段设为该Producer的producerGroup值；然后调用DefaultMQProducerImpl.send(Message msg)方法将该事务消息发送到Broker；如果发送失败时抛出异常，则不继续下面的处理逻辑，直接跑异常到应用层；否则继续下面的处理；   
 5.2）若发送消息的返回结果（SendResult对象）状态为SEND\_OK，则调用业务层实现的executeLocalTransactionBranch方法，执行本地业务逻辑并返回本地事务状态LocalTransactionState；   
 5.3）若发送消息的返回结果（SendResult对象）状态不是SEND\_OK，则不执行本地业务逻辑，直接将本地事务状态LocalTransactionState置为ROLLBACK\_MESSAGE；   
 5.5）调用DefaultMQProducerImpl.endTransaction(SendResult sendResult, LocalTransactionState localTransactionState, Throwable localException)方法，在该方法中：   
 A）创建EndTransactionRequestHeader对象，该对象的commitOrRollback变量是根据上一步的本地业务逻辑处理结果来设置（可能为MessageSysFlag.TransactionCommitType/MessageSysFlag.TransactionRollbackTypeMessageSysFlag.TransactionNotType）、tranStateTableOffset变量值等于发送消息之后返回的该消息的queueoffset值、msgId变量值等于发送消息之后由Broker返回的MessageId；   
 B）调用MQClientAPIImpl.endTransactionOneway(String addr, EndTransactionRequestHeader requestHeader, String remark, long timeoutMillis)方法向Broker发送END\_TRANSACTION请求码；





代码：



### RocketMQ3.2.6版本后分布式事务实现

由于RocketMQ在3.2.6版本以后阉割了事务回查机制，在上述流程中的第五步向MQ发送消息如果失败的话，broker无法主动询问生产者端业务逻辑执行结果，也就无法将事务消息发送给B银行，就会造成A银行扣款成功而B银行收款未成功的数据不一致的情况。

1、borker收到确认结果后的的事务消息

下面是按key值查询事务成功提交COMMIT\_MESSAGE消息后的返回信息：



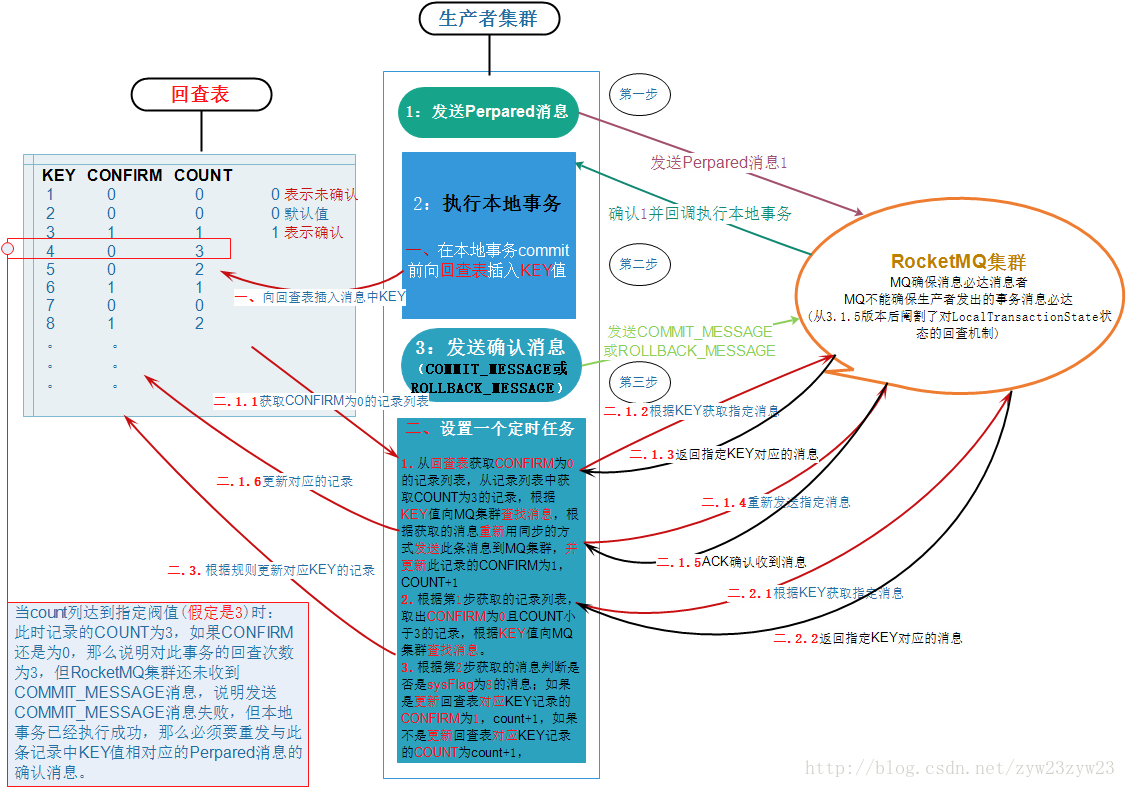
共返回两条消息：两条消息中大部分数据是一样的，但sysFlag、storeTimestamp、msgId、commitLogOffset、preparedTransactionOffset字段是不一样的：其中第1条为prepared发送的消息，第2条只有在提交COMMIT\_MESSGE消息成功后产生。

注意sysFlag、preparedTransactionOffset字段与prepared消息的区别，当提交COMMIT\_MESSGE消息成功后，推测MQ集群做了如下动作：1. 读取prepared消息，修改sysFlag、preparedTransactionOffset值，2. 在存入commitlog日志文件，设置consumerqueue序列；因为当作一条新的消息处理，所以toreTimestamp、msgId、commitLogOffset字段自然也就变了。所以按照发送的prepared消息的返回结果显示的msgId查看sysFlag状态只是prepared消息的sysFlag状态，RocketMQ4.2版本的话要用key值去查询，才能查看事务提交成功的消息标志sysFlag=8。   
下面是按key值查询事务失败提交ROLLBACK\_MESSAGE消息后的返回信息：



由此我们可以根据syyFlag判断我们在提交事务以后, 消息发送是否成功。

2、逻辑分析图



3、流程概述

1）在执行本地事务commit前向回查表插入消息的KEY值。

2）在生产者集群上设置一个定时任务（根据自身分布式事务流程执行的时间设定）。

a. 从回查表获取CONFIRM为0的记录列表，从记录列表中获取COUNT为3的记录，当count列达到指定阀值(假定是3)时：   
此时记录的COUNT为3，如果CONFIRM还是为0，那么说明对此事务的回查次数为3，但RocketMQ集群还未收到COMMIT\_MESSAGE消息，说明发送COMMIT\_MESSAGE消息失败，但本地事务已经执行成功，那么必须要重发与此条记录中KEY值相对应的Perpared消息的确认消息。根据KEY值向MQ集群查找消息，根据获取的消息重新用同步的方式发送此条消息到MQ集群，并更新此记录的CONFIRM为1，COUNT+1

b．根据第1步获取的记录列表，取出CONFIRM为0且COUNT小于3的记录，根据KEY值向MQ集群查找消息。

C．根据第2步获取的消息判断是否是sysFlag为8的消息；如果是，更新回查表对应KEY记录的CONFIRM为1，COUNT为count+1，如果不是，更新回查表对应KEY记录的COUNT为count+1。

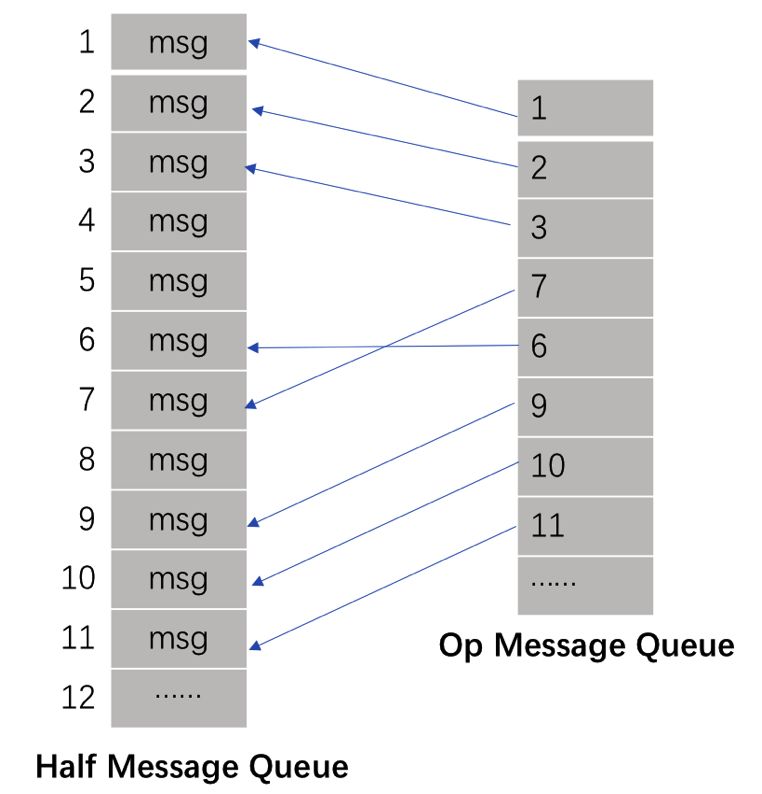
4、总结

这里是用COUNT来避免刚刚发送消息就开始判断sysFlag的情况,相当于预留了一点rabbitmq的消息发送延迟时间,这个也可以使用CREATE\_TIME来代替,比如判断记录列表中CREATE\_TIME在当前时间前3s以前,未成功发送的消息

### RocketMQ4.3版本实现分布式事务

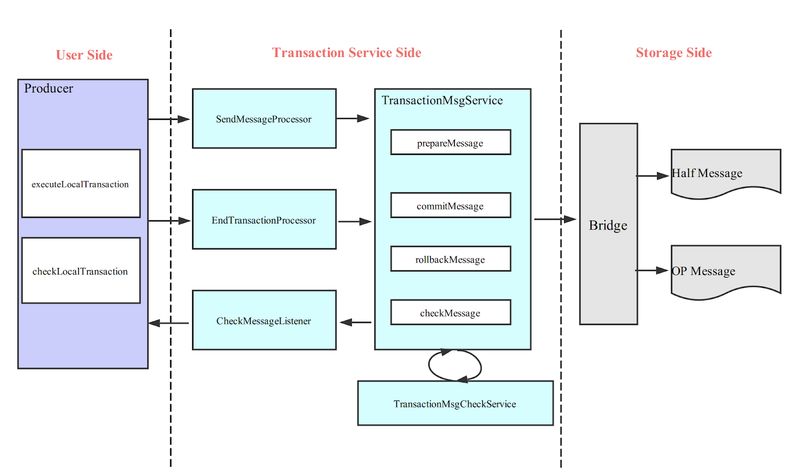
4.3版本RocketMQ事务消息在实现上充分利用了 RocketMQ 本身机制，在实现零依赖的基础上，同样实现了高性能、可扩展、全异步等一系列特性。

在具体实现上，RocketMQ 通过使用 Half Topic 以及 Operation Topic 两个内部队列来存储事务消息推进状态，如下图所示：



其中，Half Topic 对应队列中存放着 prepare 消息，Operation Topic 对应的队列则存放了 prepare message 对应的 commit/rollback 消息，消息体中则是 prepare message 对应的 offset，服务端通过比对两个队列的差值来找到尚未提交的超时事务，进行回查。

在具体实现上，事务消息作为普通消息的一个应用场景，在实现过程中进行了分层抽象，从而避免了对 RocketMQ 原有存储机制的修改，如下图所示：

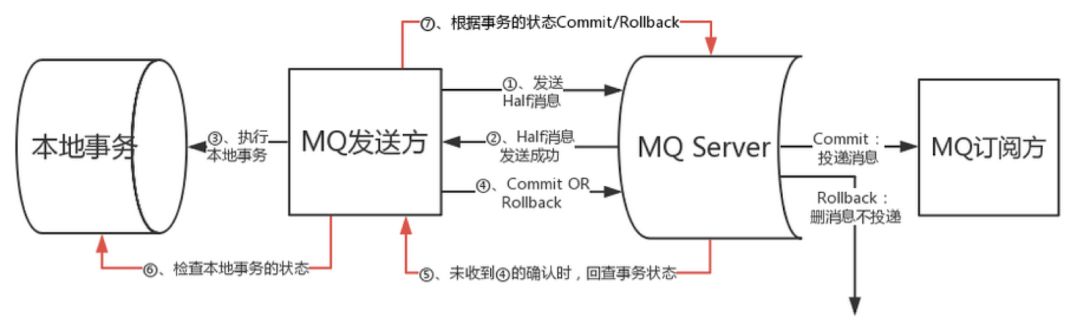


从用户侧来说，用户需要分别实现本地事务执行以及本地事务回查方法，因此只需关注本地事务的执行状态即可；而在 service 层，则对事务消息的两阶段提交进行了抽象，同时针对超时事务实现了回查逻辑，通过不断扫描当前事务推进状态，来不断反向请求 Producer 端获取超时事务的执行状态，在避免事务挂起的同时，也避免了 Producer 端的单点故障。而在存储层，RocketMQ 通过 Bridge 封装了与底层队列存储的相关操作，用以操作两个对应的内部队列，用户也可以依赖其他存储介质实现自己的 service，RocketMQ 会通过 ServiceProvider 加载进来。

从上述事务消息设计中可以看到，RocketMQ 事务消息较好的解决了事务的最终一致性问题，事务发起方仅需要关注本地事务执行以及实现回查接口给出事务状态判定等实现，而且在上游事务峰值高时，可以通过消息队列，避免对下游服务产生过大压力。

事务消息不仅适用于上游事务对下游事务无依赖的场景，还可以与一些传统分布式事务架构相结合，而 MQ 的服务端作为天生的具有高可用能力的协调者，使得我们未来可以基于 RocketMQ 提供一站式轻量级分布式事务解决方案，用以满足各种场景下的分布式事务需求。

**事务消息作为一种异步确保型事务，  将两个事务分支通过 MQ 进行异步解耦，RocketMQ 事务消息的设计流程同样借鉴了两阶段提交理论，整体交互流程如下图所示：**



1、事务发起方首先发送 prepare(即发送Half消息) 消息到 MQ。

2、在发送 prepare 消息成功后(即收到Half消息发送成功的回执消息) ,执行本地事务(业务系统自己的本地事物逻辑代码)。

3、根据本地事务执行结果,返回给MQ发送方发送: commit 或者是 rollback。

4、如果MQ发送方接收到的消息是: rollback，MQ 将删除该 prepare 消息不进行下发，如果是 commit 消息，MQ 将会把这个消息发送给 consumer 端。

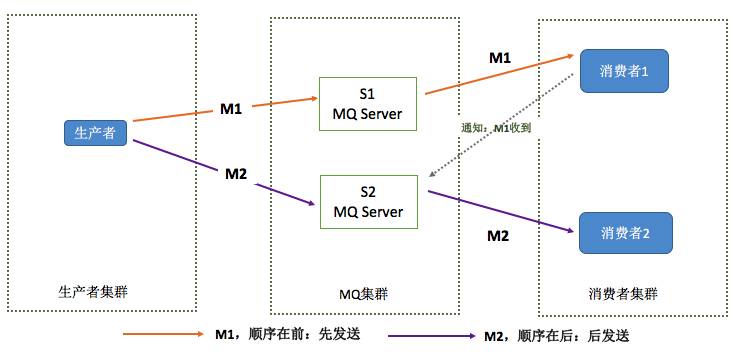
5、如果执行本地事务过程中，执行端挂掉，或者超时，MQ 将会不停的询问其同组的其他 producer 来获取状态。

6、Consumer 端的消费成功机制有 MQ 保证。

# 顺序消息

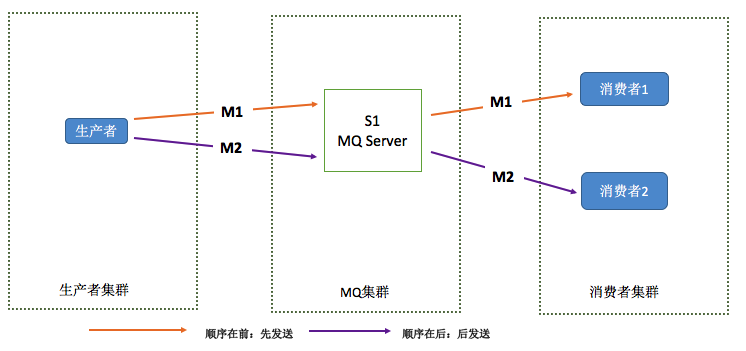
消息有序指的是可以按照消息的发送顺序来消费。例如：一笔订单产生了 3 条消息，分别是订单创建、订单付款、订单完成。消费时，要按照顺序依次消费才有意义。与此同时多笔订单之间又是可以并行消费的。首先来看如下示例：

假如生产者产生了2条消息：M1、M2，要保证这两条消息的顺序，应该怎样做？你脑中想到的可能是这样：



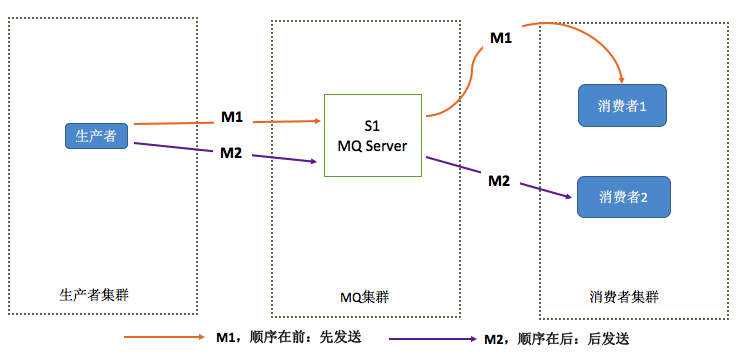
假定M1发送到S1，M2发送到S2，如果要保证M1先于M2被消费，那么需要M1到达消费端被消费后，通知S2，然后S2再将M2发送到消费端。

这个模型存在的问题是，如果M1和M2分别发送到两台Server上，就不能保证M1先达到MQ集群，也不能保证M1被先消费。换个角度看，如果M2先于M1达到MQ集群，甚至M2被消费后，M1才达到消费端，这时消息也就乱序了，说明以上模型是不能保证消息的顺序的。如何才能在MQ集群保证消息的顺序？一种简单的方式就是将M1、M2发送到同一个Server上：



这样可以保证M1先于M2到达MQServer（生产者等待M1发送成功后再发送M2），根据先达到先被消费的原则，M1会先于M2被消费，这样就保证了消息的顺序。

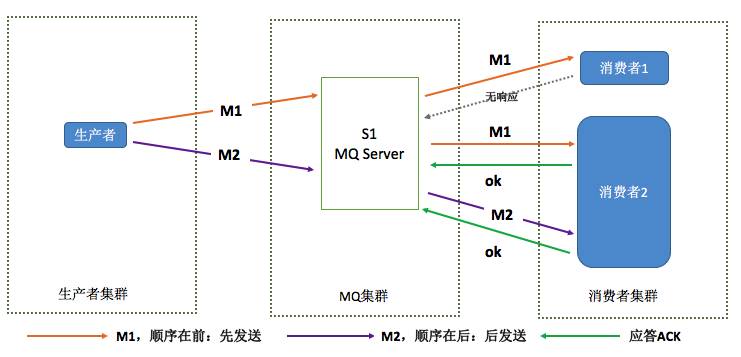
这个模型也仅仅是理论上可以保证消息的顺序，在实际场景中可能会遇到下面的问题：



只要将消息从一台服务器发往另一台服务器，就会存在网络延迟问题。如上图所示，如果发送M1耗时大于发送M2的耗时，那么M2就仍将被先消费，仍然不能保证消息的顺序。即使M1和M2同时到达消费端，由于不清楚消费端1和消费端2的负载情况，仍然有可能出现M2先于M1被消费的情况。

那如何解决这个问题？将M1和M2发往同一个消费者，且发送M1后，需要消费端响应成功后才能发送M2。

聪明的你可能已经想到另外的问题：如果M1被发送到消费端后，消费端1没有响应，那是继续发送M2呢，还是重新发送M1？一般为了保证消息一定被消费，肯定会选择重发M1到另外一个消费端2，就如下图所示。



这样的模型就严格保证消息的顺序，细心的你仍然会发现问题，消费端1没有响应Server时有两种情况，一种是M1确实没有到达(数据在网络传送中丢失)，另外一种消费端已经消费M1且已经发送响应消息，只是MQ Server端没有收到。如果是第二种情况，重发M1，就会造成M1被重复消费。也就引入了我们要说的第二个问题，消息重复问题，这个后文会详细讲解。

回过头来看消息顺序问题，严格的顺序消息非常容易理解，也可以通过文中所描述的方式来简单处理。总结起来，要实现严格的顺序消息，简单且可行的办法就是：

**保证生产者 - MQServer - 消费者是一对一对一的关系**

这样的设计虽然简单易行，但也会存在一些很严重的问题，比如：

1、并行度就会成为消息系统的瓶颈（吞吐量不够）

2、更多的异常处理，比如：只要消费端出现问题，就会导致整个处理流程阻塞，我们不得不花费更多的精力来解决阻塞的问题。

但我们的最终目标是要集群的高容错性和高吞吐量。这似乎是一对不可调和的矛盾，那么阿里是如何解决的？

**世界上解决一个计算机问题最简单的方法：“恰好”不需要解决它！——沈询**

有些问题，看起来很重要，但实际上我们可以通过合理的设计或者将问题分解来规避。如果硬要把时间花在解决问题本身，实际上不仅效率低下，而且也是一种浪费。从这个角度来看消息的顺序问题，我们可以得出两个结论：

1、不关注乱序的应用实际大量存在

2、队列无序并不意味着消息无序

要保证消息的顺序消费，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MessageListenerOrderly回调

# 重复消息

## 幂等性

提到重复消息，要想提一提幂等性这个概念。

### 概述

**幂等性**原本是数学上的概念，即使公式：f(x)=f(f(x)) 能够成立的数学性质。用在编程领域，则意为对同一个系统，使用同样的条件，一次请求和重复的多次请求对系统资源的影响是一致的。

幂等性是分布式系统设计中十分重要的概念，具有这一性质的接口在设计时总是秉持这样的一种理念：调用接口发生异常并且重复尝试时，总是会造成系统所无法承受的损失，所以必须阻止这种现象的发生。

幂等有两个维度：一是空间维度上的幂等，即幂等对象的范围，是个人还是机构，是某一次交易还是某种类型的交易...二是时间维度上的幂等，即幂等的保证时间，是几秒、几分钟还是永久性的...

不同的需求，会有不一样的解决方案，难度和成本也不一样。

### 适用领域

试想这样的一种场景：在电商平台上支付后，因为网络原因导致系统提示你支付失败，于是你又重新付款了一次，等完成后检查网银发现被系统扣了两次款，这是一种什么样的体验？

造成上述问题的原因可能有很多，比如第一次付款时实际支付成功，但是信息返回时网络中断导致系统误判；又比如第一次付款的确失败了，但第二次付款时发生意外，导致支付请求被重复发送等等。在一次支付的过程中，每个环节都有可能会发生问题，我们要如何规避这类问题引发的分险？

幂等性是解决这类问题的方案之一，所以在电商，银行，互联网金融等对数据准确性要求很高的领域中，这一特性具有十分重要的地位。

### 常用思路

#### 多版本并发控制

多版本并发控制，乐观锁的一种实现，在数据更新时需要去比较持有数据的版本号，版本号不一致的操作无法成功。例如博客点赞次数自动+1的接口：

public boolean addCount(Long id, Long version);

update blogTable set count= count+1,version=version+1 where id=321 and version=123；

每一个version只有一次执行成功的机会，一旦失败必须重新获取。

#### 去重表

利用数据库表单的特性来实现幂等，常用的一个思路是在表上构建唯一性索引，保证某一类数据一旦执行完毕，后续同样的请求再也无法成功写入。

例子还是上述的博客点赞问题，要想防止一个人重复点赞，可以设计一张表，将博客id与用户id绑定建立唯一索引，每当用户点赞时就往表中写入一条数据，这样重复点赞的数据就无法写入。

#### TOKEN机制

这种机制就比较重要了，适用范围较广，有多种不同的实现方式。其核心思想是为每一次操作生成一个唯一性的凭证，也就是token。一个token在操作的每一个阶段只有一次执行权，一旦执行成功则保存执行结果。对重复的请求，返回同一个结果。

以电商平台为例子，电商平台上的订单id就是最适合的token。当用户下单时，会经历多个环节，比如生成订单，减库存，减优惠券等等。

每一个环节执行时都先检测一下该订单id是否已经执行过这一步骤，对未执行的请求，执行操作并缓存结果，而对已经执行过的id，则直接返回之前的执行结果，不做任何操作。这样可以在最大程度上避免操作的重复执行问题，缓存起来的执行结果也能用于事务的控制等。

## 重复消息解决办法

造成消息重复的根本原因是：网络不可达。只要通过网络交换数据，就无法避免这个问题。所以解决这个问题的办法就是绕过这个问题。那么问题就变成了：如果消费端收到两条一样的消息，应该怎样处理？

1、消费端处理消息的业务逻辑保持幂等性

2、保证每条消息都有唯一编号且保证消息处理成功与去重表的日志同时出现

第1条很好理解，只要保持幂等性，不管来多少条重复消息，最后处理的结果都一样。第2条原理就是利用一张日志表来记录已经处理成功的消息的ID，如果新到的消息ID已经在日志表中，那么就不再处理这条消息。

第1条解决方案，很明显应该在消费端实现，不属于消息系统要实现的功能。第2条可以消息系统实现，也可以业务端实现。正常情况下出现重复消息的概率其实很小，如果由消息系统来实现的话，肯定会对消息系统的吞吐量和高可用有影响，所以最好还是由业务端自己处理消息重复的问题，这也是RocketMQ不解决消息重复的问题的原因。

**RocketMQ不保证消息不重复，如果你的业务需要保证严格的不重复消息，需要你自己在业务端去重。**

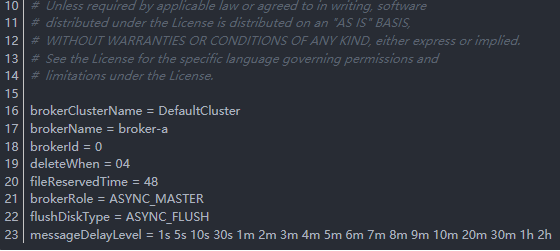
# 延时/定时消息

RocketMQ 支持定时消息，但是不支持任意时间精度，仅支持特定的 level，例如定时 5s， 10s， 1m 等。其中，level=0 级表示不延时，level=1 表示 1 级延时，level=2 表示 2 级延时，以此类推。

## 配置

打开安装目录的./conf/broker.conf  文件，并添加如下延迟级别的时长设置:

messageDelayLevel = 1s 5s 10s 30s 1m 2m 3m 4m 5m 6m 7m 8m 9m 10m 20m 30m 1h 2h



**延迟配置说明：**

1、配置项配置了从**1**级开始，各级延时的时间，可以修改这个指定级别的延时时间；

2、时间单位支持：s、m、h、d，分别表示秒、分、时、天；

3、默认值就是上面声明的，可手工调整；

4、默认值已够用，不建议修改这个值。

## 代码实现

1、新建一个发送消息的生产者

# 生产者

## 核心类

1、核心发布消息的类DefaultMQProducer，继承自MQProducer接口，此接口定义了一系列发送消息的方法，如普通消息，顺序消息，延时消息等，最终进行网络通信会交给MQClientAPIImpl处理。

2、rocketmq从4.1.3版本开始又支持了事务消息，由TransactionMQProducer类提供（之后会有专门的文章进行详细解读事务消息）

## 生产者属性配置

| 参数名 | 默认值 | 说明 |
| --- | --- | --- |
| producerGroup | DEFAULT\_PRODUCER | Producer组名，相同分组的producer应该有相同的发送消息逻辑 |
| createTopicKey | AUTO\_CREATE\_TOPIC\_KEY | 自动创建topic时，以此默认topic为模板创建指定topic |
| defaultTopicQueueNums | 4 | 自动创建topic队列数量 |
| sendMsgTimeout | 3000 | 发送消息的超时时间，单位ms |
| compressMsgBodyOverHowmuch | 4098 | 消息体超过多大会进行压缩，单位字节 |
| retryTimesWhenSendFailed | 2 | 同步发送消息，发送失败重试次数 |
| retryTimesWhenSendAsyncFailed | 2 | 异步发送消息，发送失败的重试次数 |
| retryAnotherBrokerWhenNotStoreOK | false | 同步发送消息，消息存储失败是否重试其他broker |
| maxMessageSize | 4194304 | 客户端限制消息的大小，默认4M |
| TransactionListener |  | 事务消息时，必须设置的回查监听器 |

## 生产者之group

我们在创建producer时必须要指定一个group，这里有两个作用：

生产者一般会是集群部署的，group用来标识一类生产者，相同group的生产者一般要有相同的发送逻辑。

在发送事务消息时，当事务消息异常，broker端来回查事务状态时，需要知道是由哪类生产者发送的事务消息，生产端会根据group名称来查找对应的producer来执行相应的回查逻辑。

## 生产者启动流程

简单说明下整个启动流程：

1、首先在DefaultMQProducerImpl中会做一些参数校验，如group是否合法；然后会创建MQClientInstance实例，此实例包含网络连接、线程资源等，相同的clientId会共享此实例，所以通过MQClientManager来管理。

2、核心的启动流程在MQClientInstance类中，如果nameserver地址没有配置的话，会先通过静态的http服务器地址去抓取nameserver的地址；再则启动netty客户端。

3、启动一些定时任务，跟producer有关的如下几个：

如果producer没有配置nameserver地址，启动定时抓取nameserver的地址的定时任务，任务延时10s开始，每隔2分支执行一次。

轮询nameserver定时任务，主要是定时更新topic的路由信息，任务延时10ms开始，每隔30s执行一次。

清除下线的broker和向broker发送心跳，任务延时1s执行，每隔30s执行一次

## 生产者寻址

RocketMQ 有多种配置方式可以令客户端找到 NameServer, 然后通过 NameServer 再找到 Broker，分别如下，  
优先级由高到低，高优优先级会覆盖低优先级

1、代码中指定 Name Server 地址

producer.setNamesrvAddr("192.168.0.1:9876;192.168.0.2:9876");

2、启动参数指定

-Drocketmq.namesrv.addr=192.168.0.1:9876;192.168.0.2:9876

3、环境变量指定 Name Server 地址

export NAMESRV\_ADDR=192.168.0.1:9876;192.168.0.2:9876

4、HTTP 静态服务器寻址（默认）

如果以上三种都没有设置name server的地址，客户端启动后先会访问一个静态http服务器获取name server的地址，然后会启动一个定时任务访问这个静态 HTTP 服务器，地址如下：

<http://jmenv.tbsite.net:8080/rocketmq/nsaddr>

## 发送消息如何获取路由信息

1、broker在启动的时候通过参数autoCreateTopicEnable设置是否自动创建topic，默认为true，此时会创建一个名为TBW102(4.3版本已经改名为AUTO\_CREATE\_TOPIC\_KEY)的topic（参见类TopicConfigManager），broker在向namesrv注册时会把默认的topic注册上去。如果设置false，则不会注册。

2、producer在发送消息时会在本地获取路由信息，第一次发送的话本地肯定没有，就会去namesrv获取，如果此时namesrv也没有，则会获取TBW102的topic信息（参见DefaultMQProducerImpl.tryToFindTopicPublishInfo），以此为模板创建topic，然后选择topic下的一台broker发，broker创建后，会通过心跳注册到namesrv上。

3、如果autoCreateTopicEnable设置false的话，producer发送消息会报找不到路由的异常，此时必须手动创建topic。

建议autoCreateTopicEnable设置false，基于以上第二步，自动创建topic后，以后所有该TOPIC的消息，都将发送到刚才选择的这台broke上，达不到负载均衡的目的。所以基于目前RocketMQ的设计，建议关闭自动创建TOPIC的功能，然后根据消息量的大小，手动创建TOPIC。

可以通过管理工具mqadmin来手动创建topic

sh mqadmin updateTopic -c [集群名称] -n [nameserver地址] -t [topic名称] -w [写队列数] -r [读队列数]

手动创建了Topic后，producer就可以轮询的发送到不同的broker了。

# 消费者

## DefaultMQPushConsumer

使用DefaultMQPushConsumer主要是设置好各种参数和传入处理消息的函数。系统收到消息后自动调用处理函数来处理消息，自动保存Offset，而且加入新的DefaultMQPushConsumer后会自动做负载均衡。下面结合org.apache.rocketmq.example.quickstart包中的源码来介绍。

DefaultMQPushConsumer需要设置三个参数：一是这个Consumer的GroupName，二是NameServer的地址和端口号，三是Topic的名称，下面详细介绍。

Consumer的GroupName用于把多个Consumer组织到一起，提高并发处理能力，GroupName需要和消息模式(MessageModel）配合使用。

RocketMQ支持两种消息模式：Clustering 和 Broadcasting。

在Clustering 模式下，同一个ConsumerGroup(GroupName相同)里的每个Consumer只消费所订阅消息的一部分内容，同一个ConsumerGroup里所有的Consumer消费的内容合起来才是所订阅Topic内容的整体，从而达到负载均衡的目的。

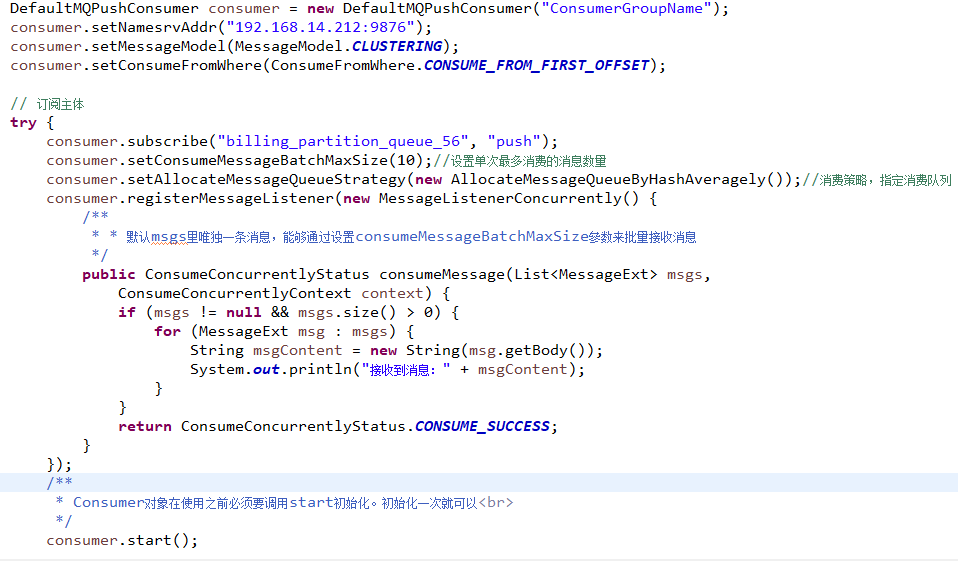
在Broadcasting模式下，同一个ConsumerGroup里的每个Consumer都能消费到所订阅Topic的全部消息，也就是一个消息会被多次分发，被多个Consumer消费。

NameServer的地址和端口号，可以填写多个，用分号隔开，达到消除单点故障的目的，比如 “ip1:port;ip2:port;ip3:port”。

Topic名称用来标识消息类型，需要提前创建。如果不需要消费某个Topic下的所有消息，可以通过指定消息的Tag进行消息过滤，比如：Consumer.subscribe("TopicTest", "tag1 || tag2 || tag3")，表示这个Consumer要消费“TopicTest”下带有tag1或tag2或tag3的消息(Tag是在发送消息时设置的标签)。在填写Tag参数的位置，用null或者“\*”表示要消费这个Topic的所有消息。

### 示例代码





### 处理流程

从Broker的源码中可以看出，服务端接到新消息请求后，如果队列里没有新消息，并不急于返回，通过一个循环不断查看状态，每次 waitForRunning一段时候(默认是5秒)，然后后再Check。默认情况下当Broker一直没有新消息，第三次Check的时候，等待时间超过Request里面的 BrokerSuspendMaxTimeMillis，就返回空结果。在等待的过程中，Broker收到了新的消息后会直接调用notifyMessageArriving函数返回请求结果。“长轮询”的核心是，Broker端HOLD住客户端过来的请求一小段时间，在这个时间内有新消息到达，就利用现有的连接立刻返回消息给Consumer。“长轮询”的主动权还是掌握在Consumer手中，Broker即使有大量消息积压，也不会主动推送给Consumer。

## DefaultMQPullConsumer

### 示例代码





示例代码的处理逻辑是遍历指定Topic下的所有MessageQueue，然后从中pull消息，并记录消费的offset。主要包括下面三件事：

获取MessageQueue并遍历

一个Topic包含多个MessageQueue，如果这个Consumer需要获取Topic下的所有消息，就要遍历所有的MessageQueue。如果有特殊情况，也可以选择某些指定的MessageQueue来消费。

维护Offset

从一个MessageQueue中拉取消息时，要传入Offset参数，随着不断读取消息，offset不断增加，这个时候需要用户把offset存储下来，根据具体情况可以保存在内存中、写到磁盘或数据库等。

根据不同的拉取状态做不同的处理

拉取消息的请求发出后，会返回FOUND、NO\_NEW\_MSG、NO\_MATCHED\_MSG和OFFSET\_ILLEGAL四种状态。需要根据每个状态做不同的处理。比较重要的两个状态是FOUND和NO\_NEW\_MSG，分别表示拉取到新的消息和没有新的消息。

### 处理逻辑

首先根据mq获得相应的brokerId，再根据brokerName跟brokerId从客户端本地获得相应的brokerAddr。如果在本地没找到相应的broker地址，那么调用updateTopicRouteInfoFromNameServer方法，从名称服务器中获取并更新本地路由信息，然后再获取。再配置sysFlag、判断下是否是集群中的slaver、根据配置信息封装本次请求的requestHeader。

之后调用MQClientAPIImpl的pullMessage方法，完成底层通信。

## 如何选择消费方式

对于任何一款消息中间件而言，消费者客户端一般有两种方式从消息中间件获取消息并消费：  
（1）Push方式：由消息中间件（MQ消息服务器代理）主动地将消息推送给消费者；采用Push方式，可以尽可能实时地将消息发送给消费者进行消费。但是，在消费者的处理消息的能力较弱的时候(比如，消费者端的业务系统处理一条消息的流程比较复杂，其中的调用链路比较多导致消费时间比较久。概括起来地说就是“慢消费问题”)，而MQ不断地向消费者Push消息，消费者端的缓冲区可能会溢出，导致异常；  
（2）Pull方式：由消费者客户端主动向消息中间件（MQ消息服务器代理）拉取消息；采用Pull方式，如何设置Pull消息的频率需要重点去考虑，举个例子来说，可能1分钟内连续来了1000条消息，然后2小时内没有新消息产生（概括起来说就是“消息延迟与忙等待”）。如果每次Pull的时间间隔比较久，会增加消息的延迟，即消息到达消费者的时间加长，MQ中消息的堆积量变大；若每次Pull的时间间隔较短，但是在一段时间内MQ中并没有任何消息可以消费，那么会产生很多无效的Pull请求的RPC开销，影响MQ整体的网络性能；

# 总结

1、RocketMQ同一台机，启动多个生产者实例或多个消费者实例，需要设置不同的实例名称

一、Producer最佳实践

1、一个应用尽可能用一个 Topic，消息子类型用 tags 来标识，tags 可以由应用自由设置。只有发送消息设置了tags，消费方在订阅消息时，才可以利用 tags 在 broker 做消息过滤。

2、每个消息在业务层面的唯一标识码，要设置到 keys 字段，方便将来定位消息丢失问题。由于是哈希索引，请务必保证 key 尽可能唯一，这样可以避免潜在的哈希冲突。

3、消息发送成功或者失败，要打印消息日志，务必要打印 sendresult 和 key 字段。

4、对于消息不可丢失应用，务必要有消息重发机制。例如：消息发送失败，存储到数据库，能有定时程序尝试重发或者人工触发重发。

5、某些应用如果不关注消息是否发送成功，请直接使用sendOneWay方法发送消息。

二、Consumer最佳实践

1、消费过程要做到幂等（即消费端去重）

2、尽量使用批量方式消费方式，可以很大程度上提高消费吞吐量。

3、优化每条消息消费过程

三、其他配置

线上应该关闭autoCreateTopicEnable，即在配置文件中将其设置为false。

RocketMQ在发送消息时，会首先获取路由信息。如果是新的消息，由于MQServer上面还没有创建对应的Topic，这个时候，如果上面的配置打开的话，会返回默认TOPIC的（RocketMQ会在每台broker上面创建名为TBW102的TOPIC）路由信息，然后Producer会选择一台Broker发送消息，选中的broker在存储消息时，发现消息的topic还没有创建，就会自动创建topic。后果就是：以后所有该TOPIC的消息，都将发送到这台broker上，达不到负载均衡的目的。

所以基于目前RocketMQ的设计，建议关闭自动创建TOPIC的功能，然后根据消息量的大小，手动创建TOPIC。