Table of Contents

README	1.1
概念(Concept)	1.2
特性(Features)	1.3
架构(Architecture)	1.4
设计(Design)	1.5
样例(Example)	1.6
最佳实践（Best Practice）	1.7
消息轨迹指南(Message Trace)	1.8
权限管理(Auth Management)	1.9
Dledger快速搭建(Quick Start)	1.10
集群部署(Cluster Deployment)	1.11
集群部署(Operation)	1.12
DefaultMQProducer API Reference	1.13

Apache RocketMQ开发者指南

> **TIPS**
>
> 本开发者指南基于 Apache RocketMQ 4.5.1
>
> 原始文档地址：https://github.com/apache/rocketmq/tree/rocketmq-all-4.5.1
>
> 考虑到官方原始地址访问慢、藏得深、难找、检索不便等问题，故而挪过来了。
>
> 此外，笔者还将官方文档编译成了PDF文件，**同学们可直接下载PDF文件，阅读、标记、做笔记啥的。PDF下载地址**：Apache RocketMQ开发者指南

这个开发者指南是帮忙您快速了解,并使用 Apache RocketMQ

1. 概念和特性

- 概念(Concept)：介绍RocketMQ的基本概念模型。
- 特性(Features)：介绍RocketMQ实现的功能特性。

2. 架构设计

- 架构(Architecture)：介绍RocketMQ部署架构和技术架构。
- 设计(Design)：介绍RocketMQ关键机制的设计原理，主要包括消息存储、通信机制、消息过滤、负载均衡、事物消息等。

3. 样例

- 样例(Example)：介绍RocketMQ的常见用法，包括基本样例、顺序消息样例、延时消息样例、批量消息样例、过滤消息样例、事物消息样例等。

4. 最佳实践

- 最佳实践（Best Practice）：介绍RocketMQ的最佳实践，包括生产者、消费者、Broker以及NameServer的最佳实践，客户端的配置方式以及JVM和linux的最佳参数配置。
- 消息轨迹指南(Message Trace)：介绍RocketMQ消息轨迹的使用方法。
- 权限管理(Auth Management)：介绍如何快速部署和使用支持权限控制特性的RocketMQ集群。
- Dledger快速搭建(Quick Start)：介绍Dledger的快速搭建方法。
- 集群部署(Cluster Deployment)：介绍Dledger的集群部署方式。

5. 运维管理

- 集群部署(Operation)：介绍单Master模式、多Master模式、多Master多slave模式等RocketMQ集群各种形式的部署方法以及运维工具mqadmin的使用方式。

6. API Reference（待补充）

- DefaultMQProducer API Reference

基本概念

1 消息模型（Message Model）

RocketMQ主要由 Producer、Broker、Consumer 三部分组成，其中Producer 负责生产消息，Consumer 负责消费消息，Broker 负责存储消息。Broker 在实际部署过程中对应一台服务器，每个 Broker 可以存储多个Topic的消息，每个Topic的消息也可以分片存储于不同的 Broker。Message Queue 用于存储消息的物理地址，每个Topic中的消息地址存储于多个 Message Queue 中。ConsumerGroup 由多个Consumer 实例构成。

2 消息生产者（Producer）

负责生产消息，一般由业务系统负责生产消息。一个消息生产者会把业务应用系统里产生的消息发送到broker服务器。RocketMQ提供多种发送方式，同步发送、异步发送、顺序发送、单向发送。同步和异步方式均需要Broker返回确认信息，单向发送不需要。

3 消息消费者（Consumer）

负责消费消息，一般是后台系统负责异步消费。一个消息消费者会从Broker服务器拉取消息、并将其提供给应用程序。从用户应用的角度而言提供了两种消费形式：拉取式消费、推动式消费。

4 主题（Topic）

表示一类消息的集合，每个主题包含若干条消息，每条消息只能属于一个主题，是RocketMQ进行消息订阅的基本单位。

5 代理服务器（Broker Server）

消息中转角色，负责存储消息、转发消息。代理服务器在RocketMQ系统中负责接收从生产者发送来的消息并存储、同时为消费者的拉取请求作准备。代理服务器也存储消息相关的元数据，包括消费者组、消费进度偏移和主题和队列消息等。

6 名字服务（Name Server）

名称服务充当路由消息的提供者。生产者或消费者能够通过名字服务查找各主题相应的Broker IP列表。多个Namesrv实例组成集群，但相互独立，没有信息交换。

7 拉取式消费（Pull Consumer）

Consumer消费的一种类型，应用通常主动调用Consumer的拉消息方法从Broker服务器拉消息、主动权由应用控制。一旦获取了批量消息，应用就会启动消费过程。

8 推动式消费（Push Consumer）

Consumer消费的一种类型，该模式下Broker收到数据后会主动推送给消费端，该消费模式一般实时性较高。

9 生产者组（Producer Group）

同一类Producer的集合，这类Producer发送同一类消息且发送逻辑一致。如果发送的是事物消息且原始生产者在发送之后崩溃，则Broker服务器会联系同一生产者组的其他生产者实例以提交或回溯消费。

10 消费者组（Consumer Group）

同一类Consumer的集合，这类Consumer通常消费同一类消息且消费逻辑一致。消费者组使得在消息消费方面，实现负载均衡和容错的目标变得非常容易。要注意的是，消费者组的消费者实例必须订阅完全相同的Topic。RocketMQ 支持两种消息模式：集群消费（Clustering）和广播消费（Broadcasting）。

11 集群消费（Clustering）

集群消费模式下,相同Consumer Group的每个Consumer实例平均分摊消息。

12 广播消费（Broadcasting）

广播消费模式下，相同Consumer Group的每个Consumer实例都接收全量的消息。

13 普通顺序消息（Normal Ordered Message）

普通顺序消费模式下，消费者通过同一个消费队列收到的消息是有顺序的，不同消息队列收到的消息则可能是无顺序的。

14 严格顺序消息（Strictly Ordered Message）

严格顺序消息模式下，消费者收到的所有消息均是有顺序的。

15 代理服务器（Broker Server）

消息中转角色，负责存储消息、转发消息。代理服务器在RocketMQ系统中负责接收从生产者发送来的消息并存储、同时为消费者的拉取请求作准备。代理服务器也存储消息相关的元数据，包括消费者组、消费进度偏移和主题和队列消息等。

16 消息（Message）

消息系统所传输信息的物理载体，生产和消费数据的最小单位，每条消息必须属于一个主题。RocketMQ中每个消息拥有唯一的Message ID，且可以携带具有业务标识的Key。系统提供了通过Message ID和Key查询消息的功能。

17 标签（Tag）

为消息设置的标志，用于同一主题下区分不同类型的消息。来自同一业务单元的消息，可以根据不同业务目的在同一主题下设置不同标签。标签能够有效地保持代码的清晰度和连贯性，并优化RocketMQ提供的查询系统。消费者可以根据Tag实现对不同子主题的不同消费逻辑，实现更好的扩展性。

特性(features)

1 订阅与发布

消息的发布是指某个生产者向某个topic发送消息；消息的订阅是指某个消费者关注了某个topic中带有某些tag的消息，进而从该topic消费数据。

2 消息顺序

消息有序指的是一类消息消费时，能按照发送的顺序来消费。例如：一个订单产生了三条消息分别是订单创建、订单付款、订单完成。消费时要按照这个顺序消费才能有意义，但是同时订单之间是可以并行消费的。RocketMQ可以严格的保证消息有序。

顺序消息分为全局顺序消息与分区顺序消息，全局顺序是指某个Topic下的所有消息都要保证顺序；部分顺序消息只要保证每一组消息被顺序消费即可。

- 全局顺序 对于指定的一个 Topic，所有消息按照严格的先入先出（FIFO）的顺序进行发布和消费。 适用场景：性能要求不高，所有的消息严格按照 FIFO 原则进行消息发布和消费的场景
- 分区顺序 对于指定的一个 Topic，所有消息根据 sharding key 进行区块分区。 同一个分区内的消息按照严格的 FIFO 顺序进行发布和消费。 Sharding key 是顺序消息中用来区分不同分区的关键字段，和普通消息的 Key 是完全不同的概念。 适用场景：性能要求高，以 sharding key 作为分区字段，在同一个区块中严格的按照 FIFO 原则进行消息发布和消费的场景。

3 消息过滤

RocketMQ的消费者可以根据Tag进行消息过滤，也支持自定义属性过滤。消息过滤目前是在Broker端实现的，优点是减少了对于Consumer无用消息的网络传输，缺点是增加了Broker的负担、而且实现相对复杂。

4 消息可靠性

RocketMQ支持消息的高可靠，影响消息可靠性的几种情况： 1) Broker正常关闭 2) Broker异常Crash 3) OS Crash 4) 机器掉电，但是能立即恢复供电情况 5) 机器无法开机（可能是cpu、主板、内存等关键设备损坏） 6) 磁盘设备损坏

1)、2)、3)、4) 四种情况都属于硬件资源可立即恢复情况，RocketMQ在这四种情况下能保证消息不丢，或者丢失少量数据（依赖刷盘方式是同步还是异步）。

5)、6)属于单点故障，且无法恢复，一旦发生，在此单点上的消息全部丢失。RocketMQ在这两种情况下，通过异步复制，可保证99%的消息不丢，但是仍然会有极少量的消息可能丢失。通过同步双写技术可以完全避免单点，同步双写势必会影响性能，适合对消息可靠性要求极高的场合，例如与Money相关的应用。注：RocketMQ从3.0版本开始支持同步双写。

5 至少一次

至少一次(At least Once)指每个消息必须投递一次。Consumer先Pull消息到本地，消费完成后，才向服务器返回ack，如果没有消费一定不会ack消息，所以RocketMQ可以很好的支持此特性。

6 回溯消费

回溯消费是指Consumer已经消费成功的消息，由于业务上需求需要重新消费，要支持此功能，Broker在向Consumer投递成功消息后，消息仍然需要保留。并且重新消费一般是按照时间维度，例如由于Consumer系统故障，恢复后需要重新消费1小时前的数据，那么Broker要提供一种机制，可以按照时间维度来回退消费进度。RocketMQ支持按照时间回溯消费，时间维度精确到毫秒。

7 事务消息

RocketMQ事务消息（Transactional Message）是指应用本地事务和发送消息操作可以被定义到全局事务中，要么同时成功，要么同时失败。RocketMQ的事务消息提供类似 X/Open XA 的分布事务功能，通过事务消息能达到分布式事务的最终一致。

8 定时消息

定时消息（延迟队列）是指消息发送到broker后，不会立即被消费，等待特定时间投递给真正的topic。 broker有配置项messageDelayLevel，默认值为“1s 5s 10s 30s 1m 2m 3m 4m 5m 6m 7m 8m 9m 10m 20m 30m 1h 2h”，18个level。可以配置自定义messageDelayLevel。注意，messageDelayLevel是broker的属性，不属于某个topic。发消息时，设置delayLevel等级即可：msg.setDelayLevel(level)。level有以下三种情况：

- level == 0，消息为非延迟消息
- 1<=level<=maxLevel，消息延迟特定时间，例如level==1，延迟1s
- level > maxLevel，则level== maxLevel，例如level==20，延迟2h

定时消息会暂存在名为SCHEDULE_TOPIC_XXXX的topic中，并根据delayTimeLevel存入特定的queue，queueId = delayTimeLevel – 1，即一个queue只存相同延迟的消息，保证具有相同发送延迟的消息能够顺序消费。broker会调度地消费SCHEDULE_TOPIC_XXXX，将消息写入真实的topic。

需要注意的是，定时消息会在第一次写入和调度写入真实topic时都会计数，因此发送数量、tps都会变高。

9 消息重试

Consumer消费消息失败后，要提供一种重试机制，令消息再消费一次。Consumer消费消息失败通常可以认为有以下几种情况：

- 由于消息本身的原因，例如反序列化失败，消息数据本身无法处理（例如话费充值，当前消息的手机号被注销，无法充值）等。这种错误通常需要跳过这条消息，再消费其它消息，而这条失败的消息即使立刻重试消费，99%也不成功，所以最好提供一种定时重试机制，即过10秒后再重试。
- 由于依赖的下游应用服务不可用，例如db连接不可用，外系统网络不可达等。遇到这种错误，即使跳过当前失败的消息，消费其他消息同样也会报错。这种情况建议应用sleep 30s，再消费下一条消息，这样可以减轻Broker重试消息的压力。

RocketMQ会为每个消费组都设置一个Topic名称为“%RETRY%+consumerGroup”的重试队列（这里需要注意的是，这个Topic的重试队列是针对消费组，而不是针对每个Topic设置的），用于暂时保存因为各种异常而导致Consumer端无法消费的消息。考虑到异常恢复起来需要一些时间，会为重试队列设置多个重试级别，每个重试级别都有与之对应的重新投递延时，重试次数越多投递延时就越大。RocketMQ对于重试消息的处理是先保存至Topic名称为“SCHEDULE_TOPIC_XXXX”的延迟队列中，后台定时任务按照对应的时间进行Delay后重新保存至“%RETRY%+consumerGroup”的重试队列中。

10 消息重投

生产者在发送消息时，同步消息失败会重投，异步消息有重试，oneway没有任何保证。消息重投保证消息尽可能发送成功、不丢失，但可能会造成消息重复，消息重复在RocketMQ中是无法避免的问题。消息重复在一般情况下不会发生，当出现消息量大、网络抖动，消息重复就会是大概率事件。另外，生产者主动重发、consumer负载变化也会导致重复消息。如下方法可以设置消息重试策略：

- retryTimesWhenSendFailed:同步发送失败重投次数，默认为2，因此生产者会最多尝试发送retryTimesWhenSendFailed + 1次。不会选择上次失败的broker，尝试向其他broker发送，最大程度保证消息不丢。超过重投次数，抛出异常，由客户端保证消息不丢。当出现RemotingException、MQClientException和部分MQBrokerException时会重投。
- retryTimesWhenSendAsyncFailed:异步发送失败重试次数，异步重试不会选择其他broker，仅在同一个broker上做重试，不保证消息不丢。
- retryAnotherBrokerWhenNotStoreOK:消息刷盘（主或备）超时或slave不可用（返回状态非SEND_OK），是否尝试发送到其他broker，默认false。十分重要消息可以开启。

11 流量控制

生产者流控，因为broker处理能力达到瓶颈；消费者流控，因为消费能力达到瓶颈。

生产者流控：

- commitLog文件被锁时间超过osPageCacheBusyTimeOutMills时，参数默认为1000ms，返回流控。
- 如果开启transientStorePoolEnable == true，且broker为异步刷盘的主机，且transientStorePool中资源不足，拒绝当前send请求，返回流控。
- broker每隔10ms检查send请求队列头部请求的等待时间，如果超过waitTimeMillsInSendQueue，默认200ms，拒绝当前send请求，返回流控。
- broker通过拒绝send 请求方式实现流量控制。

注意，生产者流控，不会尝试消息重投。

消费者流控：

- 消费者本地缓存消息数超过pullThresholdForQueue时，默认1000。
- 消费者本地缓存消息大小超过pullThresholdSizeForQueue时，默认100MB。
- 消费者本地缓存消息跨度超过consumeConcurrentlyMaxSpan时，默认2000。

消费者流控的结果是降低拉取频率。

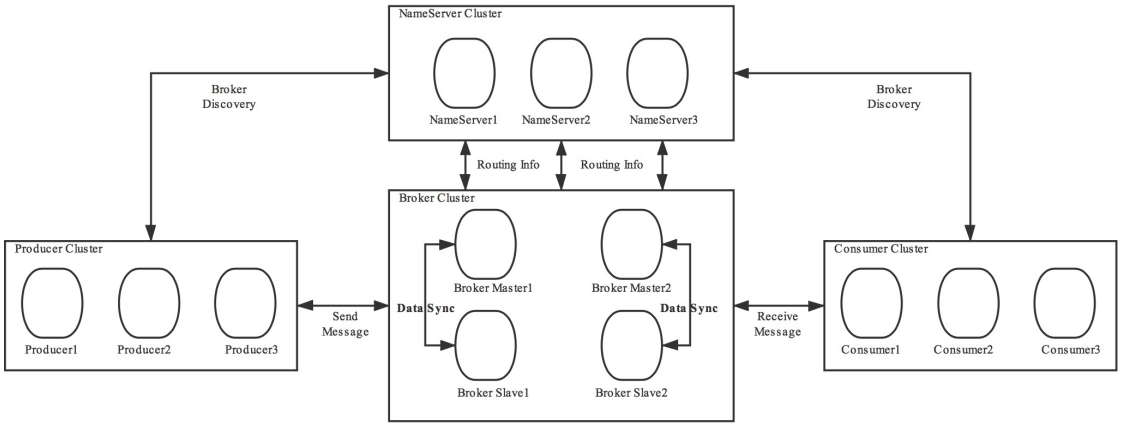
12 死信队列

死信队列用于处理无法被正常消费的消息。当一条消息初次消费失败，消息队列会自动进行消息重试；达到最大重试次数后，若消费依然失败，则表明消费者在正常情况下无法正确地消费该消息，此时，消息队列 不会立刻将消息丢弃，而是将其发送到该消费者对应的特殊队列中。

RocketMQ将这种正常情况下无法被消费的消息称为死信消息（Dead-Letter Message），将存储死信消息的特殊队列称为死信队列（Dead-Letter Queue）。在RocketMQ中，可以通过使用console控制台对死信队列中的消息进行重发来使得消费者实例再次进行消费。

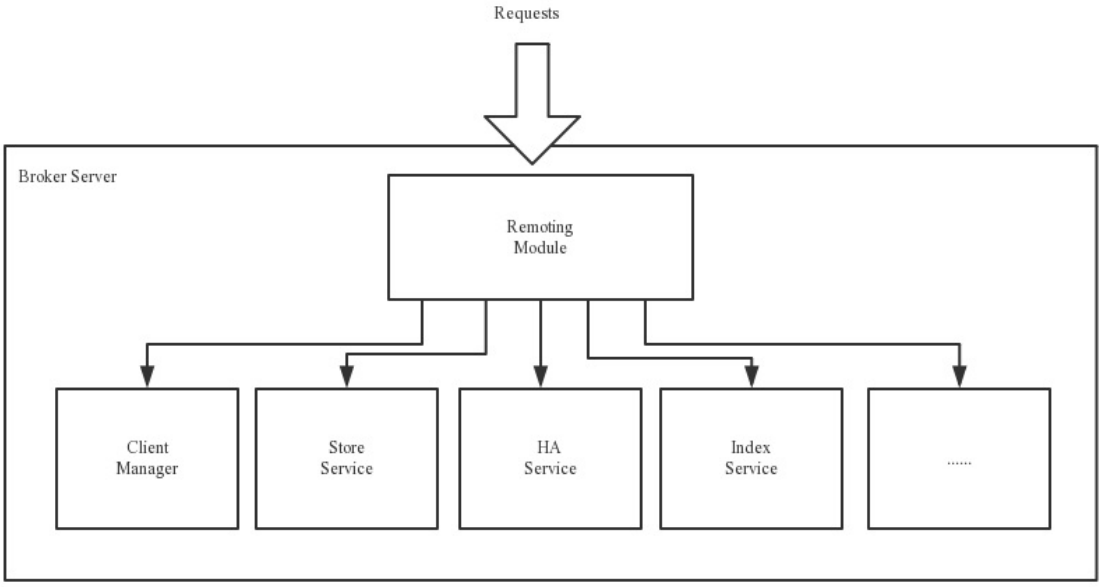
架构设计

1 技术架构

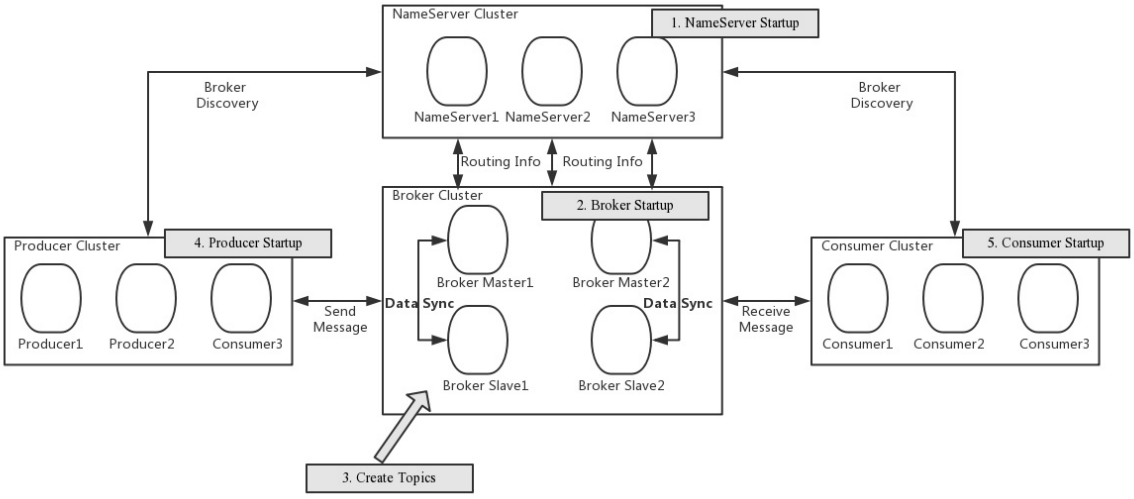


RocketMQ架构上主要分为四部分，如上图所示:

- Producer：消息发布的角色，支持分布式集群方式部署。Producer通过MQ的负载均衡模块选择相应的Broker集群队列进行消息投递，投递的过程支持快速失败并且低延迟。
- Consumer：消息消费的角色，支持分布式集群方式部署。支持以push推，pull拉两种模式对消息进行消费。同时也支持集群方式和广播方式的消费，它提供实时消息订阅机制，可以满足大多数用户的需求。
- NameServer：NameServer是一个非常简单的Topic路由注册中心，其角色类似Dubbo中的zookeeper，支持Broker的动态注册与发现。主要包括两个功能：Broker管理，NameServer接受Broker集群的注册信息并且保存下来作为路由信息的基本数据。然后提供心跳检测机制，检查Broker是否还存活；路由信息管理，每个NameServer将保存关于Broker集群的整个路由信息和用于客户端查询的队列信息。然后Producer和Conumser通过NameServer就可以知道整个Broker集群的路由信息，从而进行消息的投递和消费。NameServer通常也是集群的方式部署，各实例间相互不进行信息通讯。Broker是向每一台NameServer注册自己的路由信息，所以每一个NameServer实例上面都保存一份完整的路由信息。当某个NameServer因某种原因下线了，Broker仍然可以向其它NameServer同步其路由信息，Producer,Consumer仍然可以动态感知Broker的路由的信息。
- BrokerServer：Broker主要负责消息的存储、投递和查询以及服务高可用保证，为了实现这些功能，Broker包含了以下几个重要子模块。
- Remoting Module：整个Broker的实体，负责处理来自clients端的请求。
- Client Manager：负责管理客户端(Producer/Consumer)和维护Consumer的Topic订阅信息
- Store Service：提供方便简单的API接口处理消息存储到物理硬盘和查询功能。
- HA Service：高可用服务，提供Master Broker 和 Slave Broker之间的数据同步功能。
- Index Service：根据特定的Message key对投递到Broker的消息进行索引服务，以提供消息的快速查询。



2 部署架构



RocketMQ 网络部署特点

- NameServer是一个几乎无状态节点，可集群部署，节点之间无任何信息同步。
- Broker部署相对复杂，Broker分为Master与Slave，一个Master可以对应多个Slave，但是一个Slave只能对应一个Master，Master与Slave 的对应关系通过指定相同的BrokerName，不同的BrokerId 来定义，BrokerId为0表示Master，非0表示Slave。Master也可以部署多个。每个Broker与NameServer集群中的所有节点建立长连接，定时注册Topic信息到所有NameServer。 注意：当前RocketMQ版本在部署架构上支持一Master多Slave，但只有BrokerId=1的从服务器才会参与消息的读负载。
- Producer与NameServer集群中的其中一个节点（随机选择）建立长连接，定期从NameServer获取Topic路由信息，并向提供Topic 服务的Master建立长连接，且定时向Master发送心跳。Producer完全无状态，可集群部署。

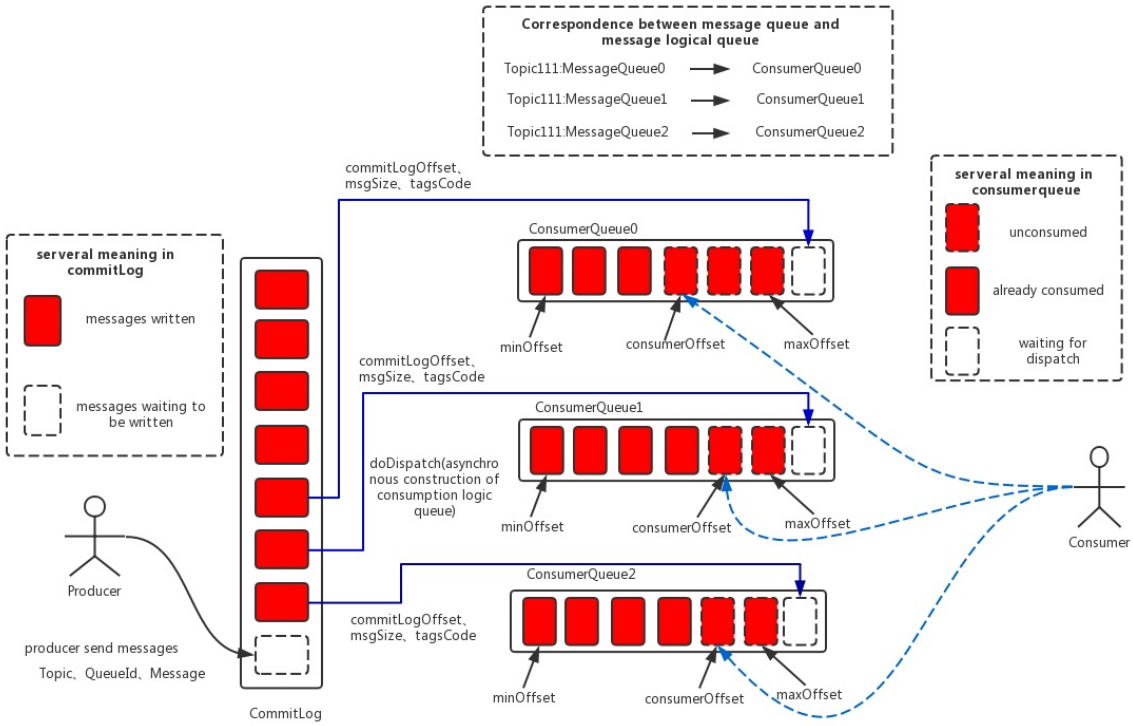
- Consumer与NameServer集群中的其中一个节点（随机选择）建立长连接，定期从NameServer获取Topic路由信息，并向提供Topic服务的Master、Slave建立长连接，且定时向Master、Slave发送心跳。Consumer既可以从Master订阅消息，也可以从Slave订阅消息，消费者在向Master拉取消息时，Master服务器会根据拉取偏移量与最大偏移量的距离（判断是否读老消息，产生读I/O），以及从服务器是否可读等因素建议下一次是从Master还是Slave拉取。

结合部署架构图，描述集群工作流程：

- 启动NameServer，NameServer起来后监听端口，等待Broker、Producer、Consumer连上来，相当于一个路由控制中心。
- Broker启动，跟所有的NameServer保持长连接，定时发送心跳包。心跳包中包含当前Broker信息(IP+端口等)以及存储所有Topic信息。注册成功后，NameServer集群中就有Topic跟Broker的映射关系。
- 收发消息前，先创建Topic，创建Topic时需要指定该Topic要存储在哪些Broker上，也可以在发送消息时自动创建Topic。
- Producer发送消息，启动时先跟NameServer集群中的其中一台建立长连接，并从NameServer中获取当前发送的Topic存在哪些Broker上，轮询从队列列表中选择一个队列，然后与队列所在的Broker建立长连接从而向Broker发消息。
- Consumer跟Producer类似，跟其中一台NameServer建立长连接，获取当前订阅Topic存在哪些Broker上，然后直接跟Broker建立连接通道，开始消费消息。

设计(design)

1 消息存储



消息存储是RocketMQ中最为复杂和最为重要的一部分，本节将分别从RocketMQ的消息存储整体架构、PageCache与Mmap内存映射以及RocketMQ中两种不同的刷盘方式三方面来分别展开叙述。

1.1 消息存储整体架构

消息存储架构图中主要有下面三个跟消息存储相关的文件构成。

(1) CommitLog：消息主体以及元数据的存储主体，存储Producer端写入的消息主体内容,消息内容不是定长的。单个文件大小默认1G ，文件名长度为20位，左边补零，剩余为起始偏移量，比如00000000000000000000代表了第一个文件，起始偏移量为0，文件大小为1G=1073741824；当第一个文件写满了，第二个文件为00000000001073741824，起始偏移量为1073741824，以此类推。消息主要是顺序写入日志文件，当文件满了，写入下一个文件；

(2) ConsumeQueue：消息消费队列，引入的目的主要是提高消息消费的性能，由于RocketMQ是基于主题topic的订阅模式，消息消费是针对主题进行的，如果要遍历commitlog文件中根据topic检索消息是非常低效的。Consumer即可根据ConsumeQueue来查找待消费的消息。其中，ConsumeQueue（逻辑消费队列）作为消费消息的索引，保存了指定Topic下的队列消息在CommitLog中的起始物理偏移量offset，消息大小size和消息Tag的HashCode值。consumequeue文件可以看成是基于topic的commitlog索引文件，故consumequeue文件夹的组织方式如下：topic/queue/file三层组织结

构，具体存储路径为：$HOME/store/consumequeue/{topic}/{queueId}/{fileName}。同样consumequeue文件采取定长设计，每一个条目共20个字节，分别为8字节的commitlog物理偏移量、4字节的消息长度、8字节tag hashcode，单个文件由30W个条目组成，可以像数组一样随机访问每一个条目，每个ConsumeQueue文件大小约5.72M；

(3) IndexFile：IndexFile（索引文件）提供了一种可以通过key或时间区间来查询消息的方法。Index文件的存储位置是：$HOME \store\index${fileName}，文件名fileName是以创建时的时间戳命名的，固定的单个IndexFile文件大小约为400M，一个IndexFile可以保存 2000W个索引，IndexFile的底层存储设计为在文件系统中实现HashMap结构，故rocketmq的索引文件其底层实现为hash索引。

在上面的RocketMQ的消息存储整体架构图中可以看出，RocketMQ采用的是混合型的存储结构，即为Broker单个实例下所有的队列共用一个日志数据文件（即为CommitLog）来存储。RocketMQ的混合型存储结构(多个Topic的消息实体内容都存储于一个CommitLog中)针对Producer和Consumer分别采用了数据和索引部分相分离的存储结构，Producer发送消息至Broker端，然后Broker端使用同步或者异步的方式对消息刷盘持久化，保存至CommitLog中。只要消息被刷盘持久化至磁盘文件CommitLog中，那么Producer发送的消息就不会丢失。正因为如此，Consumer也就肯定有机会去消费这条消息。当无法拉取到消息后，可以等下一次消息拉取，同时服务端也支持长轮询模式，如果一个消息拉取请求未拉取到消息，Broker允许等待30s的时间，只要这段时间内有新消息到达，将直接返回给消费端。这里，RocketMQ的具体做法是，使用Broker端的后台服务线程—ReputMessageService不停地分发请求并异步构建ConsumeQueue（逻辑消费队列）和IndexFile（索引文件）数据。

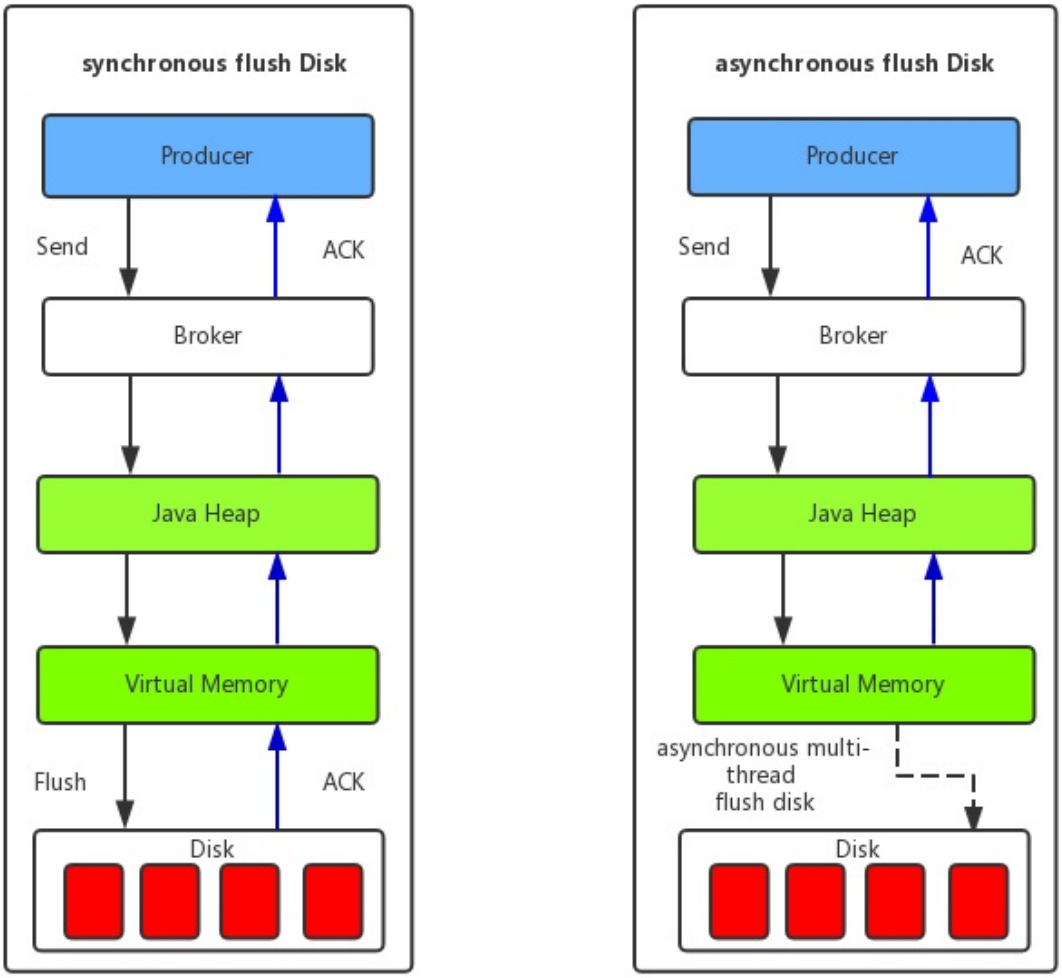
1.2 页缓存与内存映射

页缓存（PageCache)是OS对文件的缓存，用于加速对文件的读写。一般来说，程序对文件进行顺序读写的速度几乎接近于内存的读写速度，主要原因就是由于OS使用PageCache机制对读写访问操作进行了性能优化，将一部分的内存用作PageCache。对于数据的写入，OS会先写入至Cache内，随后通过异步的方式由pdflush内核线程将Cache内的数据刷盘至物理磁盘上。对于数据的读取，如果一次读取文件时出现未命中PageCache的情况，OS从物理磁盘上访问读取文件的同时，会顺序对其他相邻块的数据文件进行预读取。

在RocketMQ中，ConsumeQueue逻辑消费队列存储的数据较少，并且是顺序读取，在page cache机制的预读取作用下，Consume Queue文件的读性能几乎接近读内存，即使在有消息堆积情况下也不会影响性能。而对于CommitLog消息存储的日志数据文件来说，读取消息内容时候会产生较多的随机访问读取，严重影响性能。如果选择合适的系统IO调度算法，比如设置调度算法为“Deadline”（此时块存储采用SSD的话），随机读的性能也会有所提升。

另外，RocketMQ主要通过MappedByteBuffer对文件进行读写操作。其中，利用了NIO中的FileChannel模型将磁盘上的物理文件直接映射到用户态的内存地址中（这种Mmap的方式减少了传统IO将磁盘文件数据在操作系统内核地址空间的缓冲区和用户应用程序地址空间的缓冲区之间来回进行拷贝的性能开销），将对文件的操作转化为直接对内存地址进行操作，从而极大地提高了文件的读写效率（正因为需要使用内存映射机制，故RocketMQ的文件存储都使用定长结构来存储，方便一次将整个文件映射至内存）。

1.3 消息刷盘



(1) 同步刷盘：如上图所示，只有在消息真正持久化至磁盘后RocketMQ的Broker端才会真正返回给Producer端一个成功的ACK响应。同步刷盘对MQ消息可靠性来说是一种不错的保障，但是性能上会有较大影响，一般适用于金融业务应用该模式较多。

(2) 异步刷盘：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，降低了读写延迟，提高了MQ的性能和吞吐量。

2 通信机制

RocketMQ消息队列集群主要包括NameServe、Broker(Master/Slave)、Producer、Consumer4个角色，基本通讯流程如下：

(1) Broker启动后需要完成一次将自己注册至NameServer的操作；随后每隔30s时间定时向NameServer上报Topic路由信息。

(2) 消息生产者Producer作为客户端发送消息时候，需要根据消息的Topic从本地缓存的TopicPublishInfoTable获取路由信息。如果没有则更新路由信息会从NameServer上重新拉取，同时Producer会默认每隔30s向NameServer拉取一次路由信息。

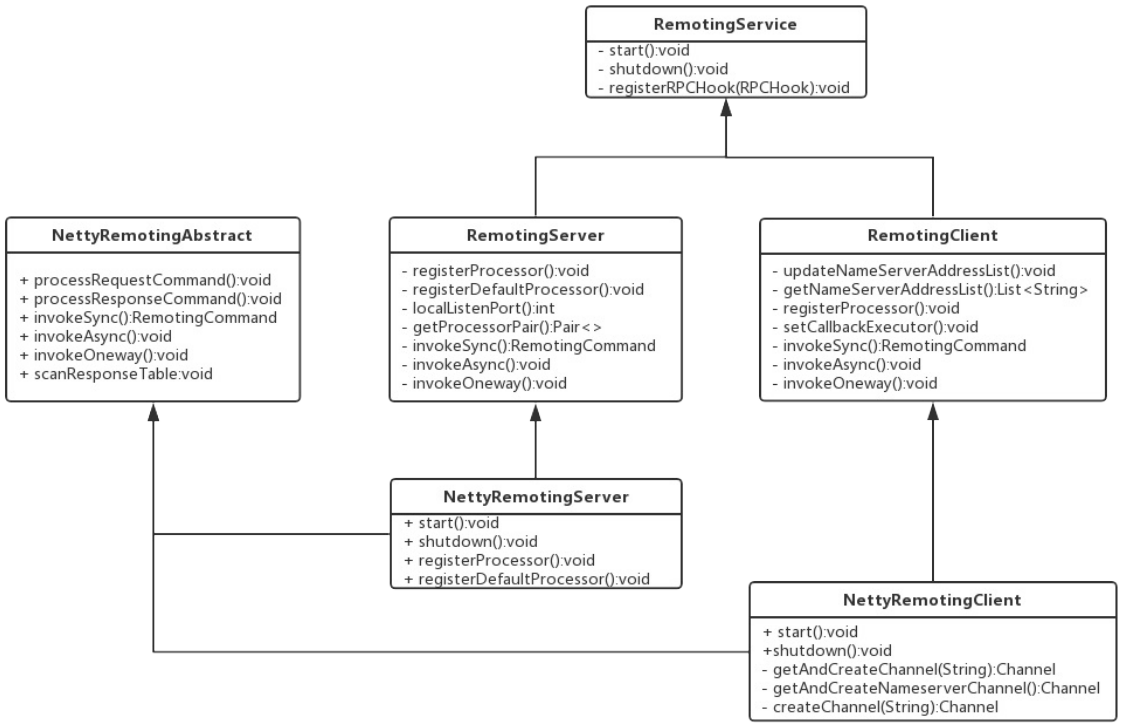
(3) 消息生产者Producer根据2）中获取的路由信息选择一个队列（MessageQueue）进行消息发送；Broker作为消息的接收者接收消息并落盘存储。

(4) 消息消费者Consumer根据2）中获取的路由信息，并再完成客户端的负载均衡后，选择其中的某一个或者某几个消息队列来拉取消息并进行消费。

从上面1）~3）中可以看出在消息生产者, Broker和NameServer之间都会发生通信（这里只说了MQ的部分通信），因此如何设计一个良好的网络通信模块在MQ中至关重要，它将决定RocketMQ集群整体的消息传输能力与最终的性能。

rocketmq-remoting 模块是 RocketMQ消息队列中负责网络通信的模块，它几乎被其他所有需要网络通信的模块（诸如rocketmq-client、rocketmq-broker、rocketmq-namesrv）所依赖和引用。为了实现客户端与服务器之间高效的数据请求与接收，RocketMQ消息队列自定义了通信协议并在Netty的基础之上扩展了通信模块。

2.1 Remoting通信类结构

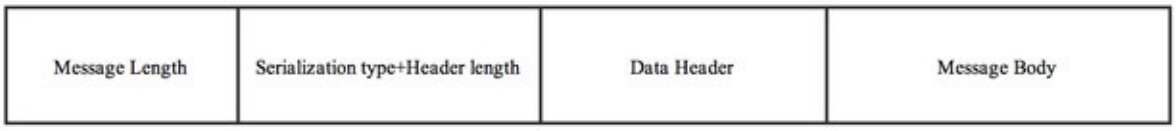


2.2 协议设计与编解码

在Client和Server之间完成一次消息发送时，需要对发送的消息进行一个协议约定，因此就有必要自定义RocketMQ的消息协议。同时，为了高效地在网络中传输消息和对收到的消息读取，就需要对消息进行编解码。在RocketMQ中，RemotingCommand这个类在消息传输过程中对所有数据内容的封装，不但包含了所有的数据结构，还包含了编码解码操作。

Header字段	类型	Request说明	Response说明
code	int	请求操作码，应答方根据不同的请求码进行不同的业务处理	应答响应码。0表示成功，非0则表示各种错误
language	LanguageCode	请求方实现的语言	应答方实现的语言
version	int	请求方程序的版本	应答方程序的版本
opaque	int	相当于reqeustId，在同一个连接上的不同请求标识码，与响应消息中的相对应	应答不做修改直接返回
flag	int	区分是普通RPC还是onewayRPC得标志	区分是普通RPC还是

flag	int	区分是普通RPC还是onewayRPC得标志	onewayRPC得标志
remark	String	传输自定义文本信息	传输自定义文本信息
extFields	HashMap	请求自定义扩展信息	响应自定义扩展信息



可见传输内容主要可以分为以下4部分：

(1) 消息长度：总长度，四个字节存储，占用一个int类型；

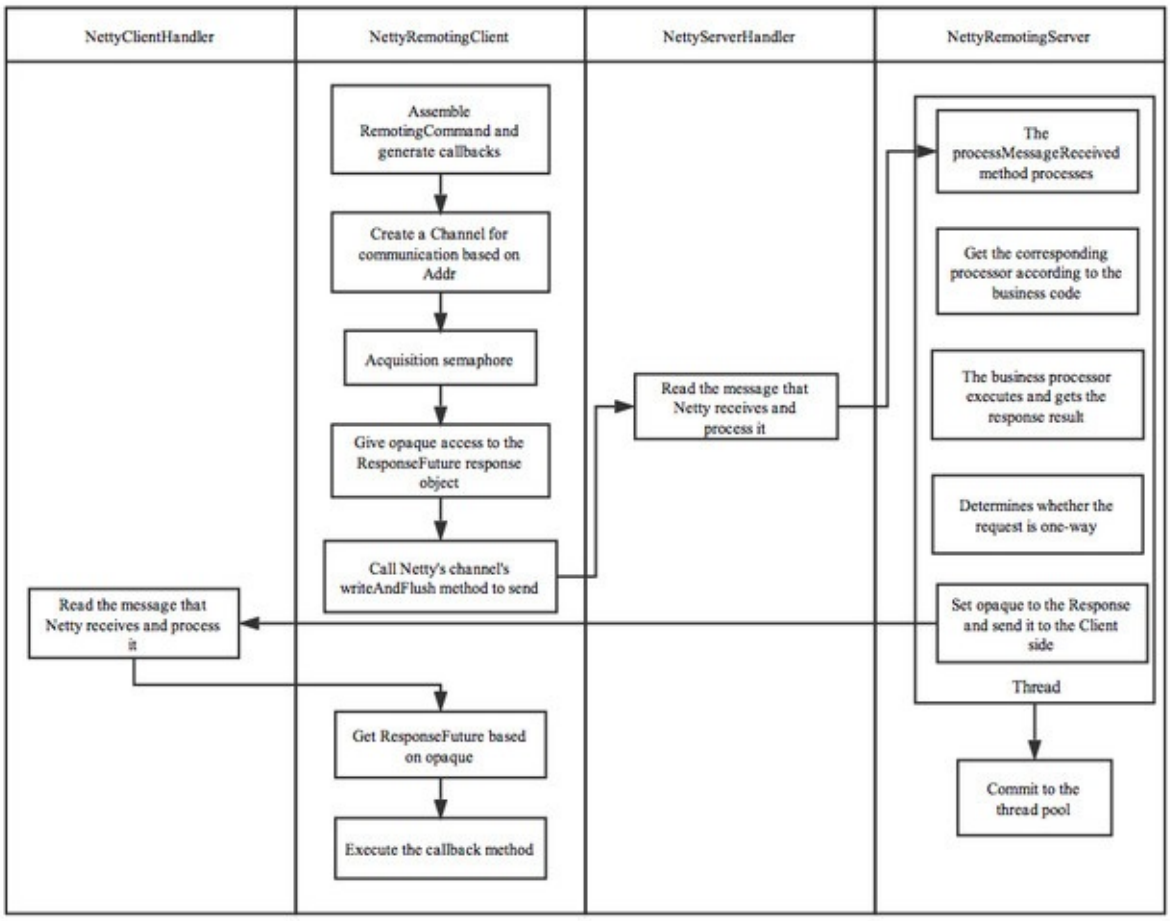
(2) 序列化类型&消息头长度：同样占用一个int类型，第一个字节表示序列化类型，后面三个字节表示消息头长度；

(3) 消息头数据：经过序列化后的消息头数据；

(4) 消息主体数据：消息主体的二进制字节数据内容；

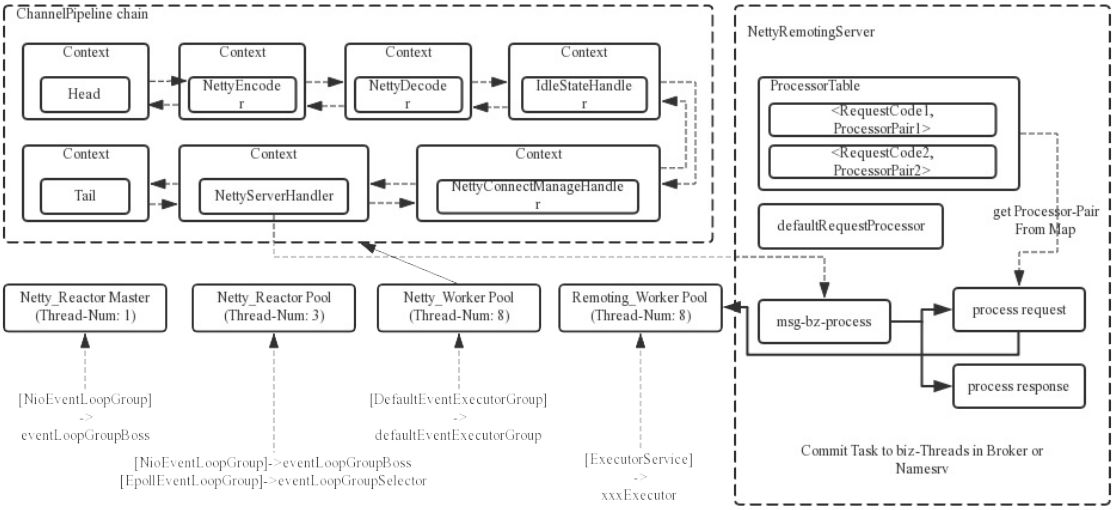
2.3 消息的通信方式和流程

在RocketMQ消息队列中支持通信的方式主要有同步(sync)、异步(async)、单向(oneway) 三种。其中“单向”通信模式相对简单，一般用在发送心跳包场景下，无需关注其Response。这里，主要介绍RocketMQ的异步通信流程。



2.4 Reactor多线程设计

RocketMQ的RPC通信采用Netty组件作为底层通信库，同样也遵循了Reactor多线程模型，同时又在这之上做了一些扩展和优化。



上面的框图中可以大致了解RocketMQ中NettyRemotingServer的Reactor 多线程模型。一个 Reactor 主线程（eventLoopGroupBoss，即为上面的1）负责监听 TCP网络连接请求，建立好连接，创建SocketChannel，并注册到selector上。RocketMQ的源码中会自动根据OS的类型选择NIO和Epoll，也可以通过参数配置）,然后监听真正的网络数据。拿到网络数据后，再丢给Worker线程池（eventLoopGroupSelector，即为上面的“N”，源码中默认设置为3），在真正执行业务逻辑之前需要进行SSL验证、编解码、空闲检查、网络连接管理，这些工作交给defaultEventExecutorGroup（即为上面的“M1”，源码中默认设置为8）去做。而处理业务操作放在业务线程池中执行，根据 RomotingCommand 的业务请求码code去processorTable这个本地缓存变量中找到对应的 processor，然后封装成task任务后，提交给对应的业务processor处理线程池来执行（sendMessageExecutor，以发送消息为例，即为上面的“M2”）。从入口到业务逻辑的几个步骤中线程池一直再增加，这跟每一步逻辑复杂性相关，越复杂，需要的并发通道越宽。

线程数	线程名	线程具体说明
1	NettyBoss_%d	Reactor 主线程
N	NettyServerEPOLLSelector%d%d	Reactor 线程池
M1	NettyServerCodecThread_%d	Worker线程池
M2	RemotingExecutorThread_%d	业务processor处理线程池

3 消息过滤

RocketMQ分布式消息队列的消息过滤方式有别于其它MQ中间件，是在Consumer端订阅消息时再做消息过滤的。RocketMQ这么做是还是在于其Producer端写入消息和Consomer端订阅消息采用分离存储的机制来实现的，Consumer端订阅消息是需要通过ConsumeQueue这个消息消费的逻辑队列拿到一个索引，然后再从CommitLog里面读取真正的消息实体内容，所以说到底也是还绕不开其存储结构。其ConsumeQueue的存储结构如下，可以看到其中有8个字节存储的Message Tag的哈希值，基于Tag的消息过滤正式基于这个字段值的。

主要支持如下2种的过滤方式 (1) Tag过滤方式：Consumer端在订阅消息时除了指定Topic还可以指定TAG，如果一个消息有多个TAG，可以用||分隔。其中，Consumer端会将这个订阅请求构建成一个 SubscriptionData，发送一个Pull消息的请求给Broker端。Broker端从RocketMQ的文件存储层—Store读取数据之前，会用这些数据先构建一个MessageFilter，然后传给Store。Store从 ConsumeQueue读取到一条记录后，会用它记录的消息tag hash值去做过滤，由于在服务端只是根据hashcode进行判断，无法精确对tag原始字符串进行过滤，故在消息消费端拉取到消息后，还需要对消息的原始tag字符串进行比对，如果不同，则丢弃该消息，不进行消息消费。

(2) SQL92的过滤方式：这种方式的大致做法和上面的Tag过滤方式一样，只是在Store层的具体过滤过程不太一样，真正的 SQL expression 的构建和执行由rocketmq-filter模块负责的。每次过滤都去执行SQL表达式会影响效率，所以RocketMQ使用了BloomFilter避免了每次都去执行。SQL92的表达式上下文为消息的属性。

4 负载均衡

RocketMQ中的负载均衡都在Client端完成，具体来说的话，主要可以分为Producer端发送消息时候的负载均衡和Consumer端订阅消息的负载均衡。

4.1 Producer的负载均衡

Producer端在发送消息的时候，会先根据Topic找到指定的TopicPublishInfo，在获取了TopicPublishInfo路由信息后，RocketMQ的客户端在默认方式下selectOneMessageQueue()方法会从TopicPublishInfo中的messageQueueList中选择一个队列（MessageQueue）进行发送消息。具体的容错策略均在MQFaultStrategy这个类中定义。这里有一个sendLatencyFaultEnable开关变量，如果开启，在随机递增取模的基础上，再过滤掉not available的Broker代理。所谓的"latencyFaultTolerance"，是指对之前失败的，按一定的时间做退避。例如，如果上次请求的latency超过550Lms，就退避3000Lms；超过1000L，就退避60000L；如果关闭，采用随机递增取模的方式选择一个队列（MessageQueue）来发送消息，latencyFaultTolerance机制是实现消息发送高可用的核心关键所在。

4.2 Consumer的负载均衡

在RocketMQ中，Consumer端的两种消费模式（Push/Pull）都是基于拉模式来获取消息的，而在Push模式只是对pull模式的一种封装，其本质实现为消息拉取线程在从服务器拉取到一批消息后，然后提交到消息消费线程池后，又“马不停蹄”的继续向服务器再次尝试拉取消息。如果未拉取到消息，则延迟一下又继续拉取。在两种基于拉模式的消费方式（Push/Pull）中，均需要Consumer端在知道从Broker端的哪一个消息队列—队列中去获取消息。因此，有必要在Consumer端来做负载均衡，即Broker端中多个MessageQueue分配给同一个ConsumerGroup中的哪些Consumer消费。

1、Consumer端的心跳包发送

在Consumer启动后，它就会通过定时任务不断地向RocketMQ集群中的所有Broker实例发送心跳包（其中包含了，消息消费分组名称、订阅关系集合、消息通信模式和客户端id的值等信息）。Broker端在收到Consumer的心跳消息后，会将它维护在ConsumerManager的本地缓存变量—consumerTable，同时并将封装后的客户端网络通道信息保存在本地缓存变量—channelInfoTable中，为之后做Consumer端的负载均衡提供可以依据的元数据信息。

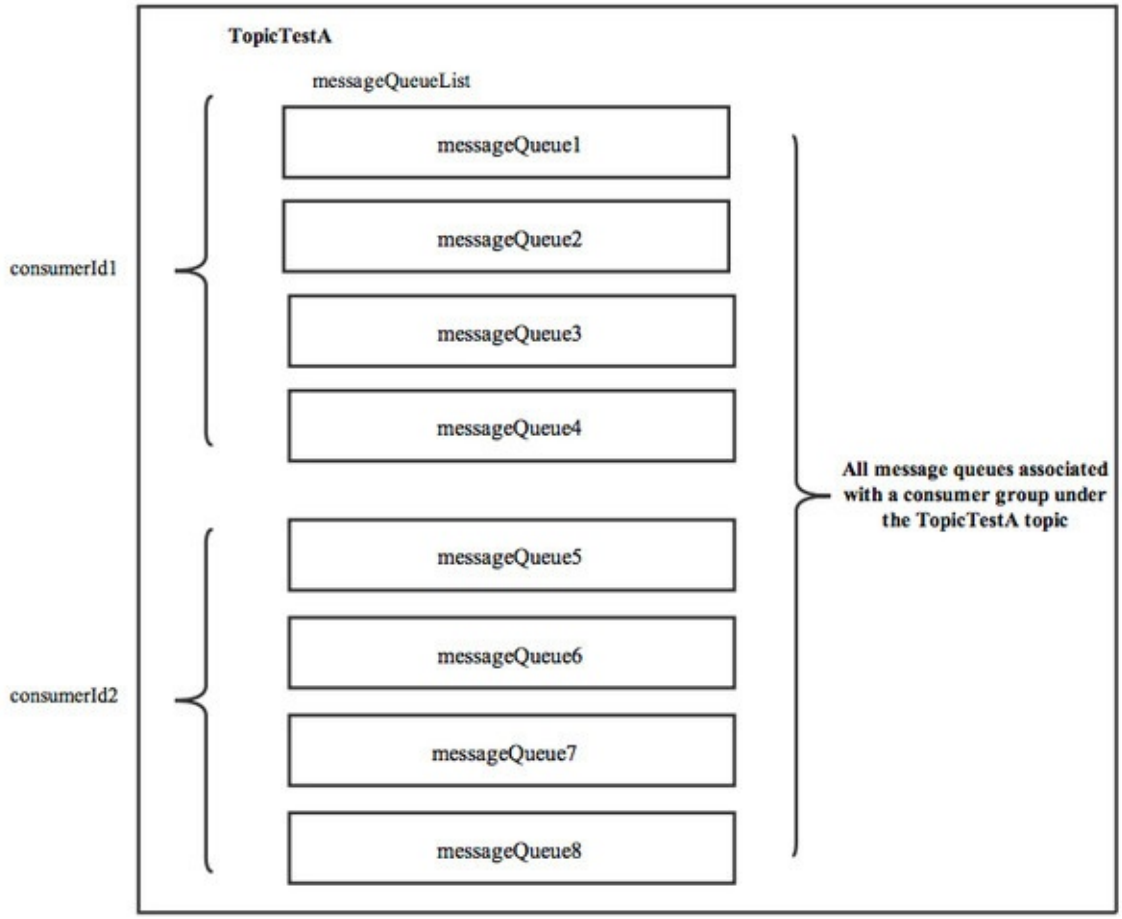
2、Consumer端实现负载均衡的核心类—RebalanceImpl

在Consumer实例的启动流程中的启动MQClientInstance实例部分，会完成负载均衡服务线程—RebalanceService的启动（每隔20s执行一次）。通过查看源码可以发现，RebalanceService线程的run()方法最终调用的是RebalanceImpl类的rebalanceByTopic()方法，该方法是实现Consumer端负载均衡的核心。这里，rebalanceByTopic()方法会根据消费者通信类型为“广播模式”还是“集群模式”做不同的逻辑处理。这里主要来看下集群模式下的主要处理流程：

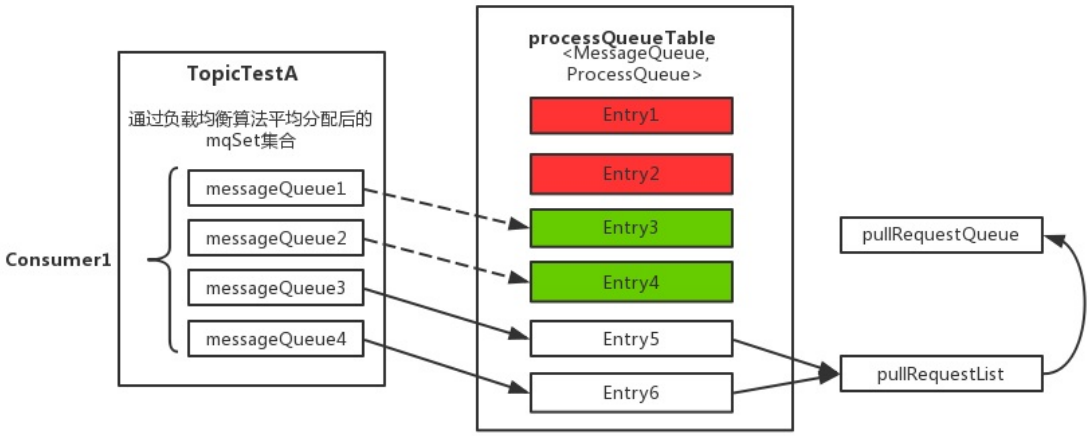
(1) 从rebalanceImpl实例的本地缓存变量—topicSubscribeInfoTable中，获取该Topic主题下的消息消费队列集合（mqSet）；

(2) 根据topic和consumerGroup为参数调用mQClientFactory.findConsumerIdList()方法向Broker端发送获取该消费组下消费者Id列表的RPC通信请求（Broker端基于前面Consumer端上报的心跳包数据而构建的consumerTable做出响应返回，业务请求码：GET_CONSUMER_LIST_BY_GROUP）；

(3) 先对Topic下的消息消费队列、消费者Id排序，然后用消息队列分配策略算法（默认为：消息队列的平均分配算法），计算出待拉取的消息队列。这里的平均分配算法，类似于分页的算法，将所有MessageQueue排好序类似于记录，将所有消费端Consumer排好序类似页数，并求出每一页需要包含的平均size和每个页面记录的范围range，最后遍历整个range而计算出当前Consumer端应该分配到的记录（这里即为：MessageQueue）。



(4) 然后，调用updateProcessQueueTableInRebalance()方法，具体的做法是，先将分配到的消息队列集合（mqSet）与processQueueTable做一个过滤比对。



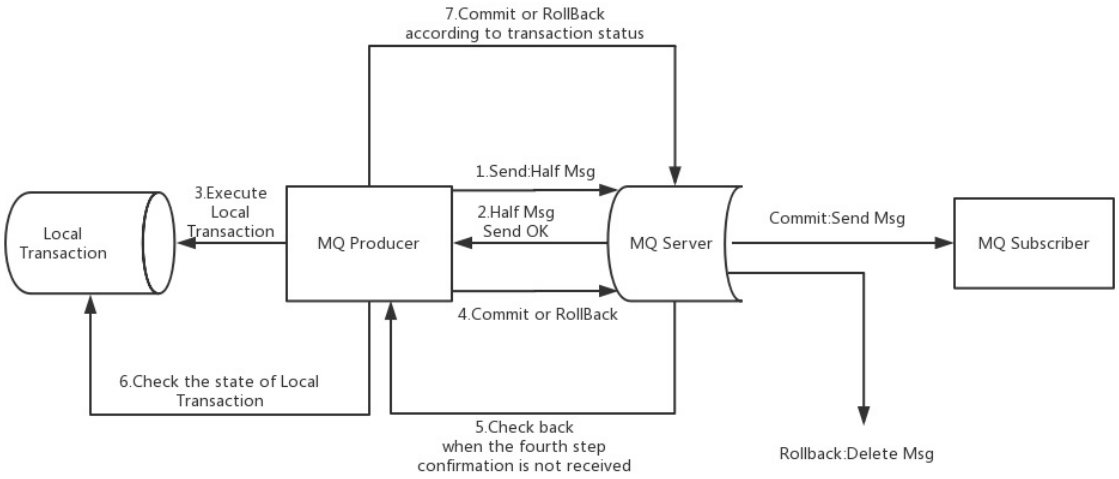
- 上图中processQueueTable标注的红色部分，表示与分配到的消息队列集合mqSet互不包含。将这些队列设置Dropped属性为true，然后查看这些队列是否可以移除出processQueueTable缓存变量，这里具体执行removeUnnecessaryMessageQueue()方法，即每隔1s 查看是否可以获取当前消费处理队列的锁，拿到的话返回true。如果等待1s后，仍然拿不到当前消费处理队列的锁则返回false。如果返回true，则从processQueueTable缓存变量中移除对应的Entry；
- 上图中processQueueTable的绿色部分，表示与分配到的消息队列集合mqSet的交集。判断该ProcessQueue是否已经过期了，在Pull模式的不用管，如果是Push模式的，设置Dropped属性为true，并且调用removeUnnecessaryMessageQueue()方法，像上面一样尝试移除Entry；

最后，为过滤后的消息队列集合（mqSet）中的每个MessageQueue创建一个ProcessQueue对象并存入RebalanceImpl的processQueueTable队列中（其中调用RebalanceImpl实例的computePullFromWhere(MessageQueue mq)方法获取该MessageQueue对象的下一个进度消费值offset，随后填充至接下来要创建的pullRequest对象属性中），并创建拉取请求对象—pullRequest添加到拉取列表—pullRequestList中，最后执行dispatchPullRequest()方法，将Pull消息的请求对象PullRequest依次放入PullMessageService服务线程的阻塞队列pullRequestQueue中，待该服务线程取出后向Broker端发起Pull消息的请求。其中，可以重点对比下，RebalancePushImpl和RebalancePullImpl两个实现类的dispatchPullRequest()方法不同，RebalancePullImpl类里面的该方法为空，这样子也就回答了上一篇中最后的那道思考题了。

消息消费队列在同一消费组不同消费者之间的负载均衡，其核心设计理念是在一个消息消费队列在同一时间只允许被同一消费组内的一个消费者消费，一个消息消费者能同时消费多个消息队列。

5 事务消息

Apache RocketMQ在4.3.0版中已经支持分布式事务消息，这里RocketMQ采用了2PC的思想来实现了提交事务消息，同时增加一个补偿逻辑来处理二阶段超时或者失败的消息，如下图所示。



5.1 RocketMQ事务消息流程概要

上图说明了事务消息的大致方案，其中分为两个流程：正常事务消息的发送及提交、事务消息的补偿流程。

1.事务消息发送及提交：

(1) 发送消息（half消息）。

(2) 服务端响应消息写入结果。

(3) 根据发送结果执行本地事务（如果写入失败，此时half消息对业务不可见，本地逻辑不执行）。

(4) 根据本地事务状态执行Commit或者Rollback（Commit操作生成消息索引，消息对消费者可见）

2.补偿流程：

(1) 对没有Commit/Rollback的事务消息（pending状态的消息），从服务端发起一次“回查”

(2) Producer收到回查消息，检查回查消息对应的本地事务的状态

(3) 根据本地事务状态，重新Commit或者Rollback

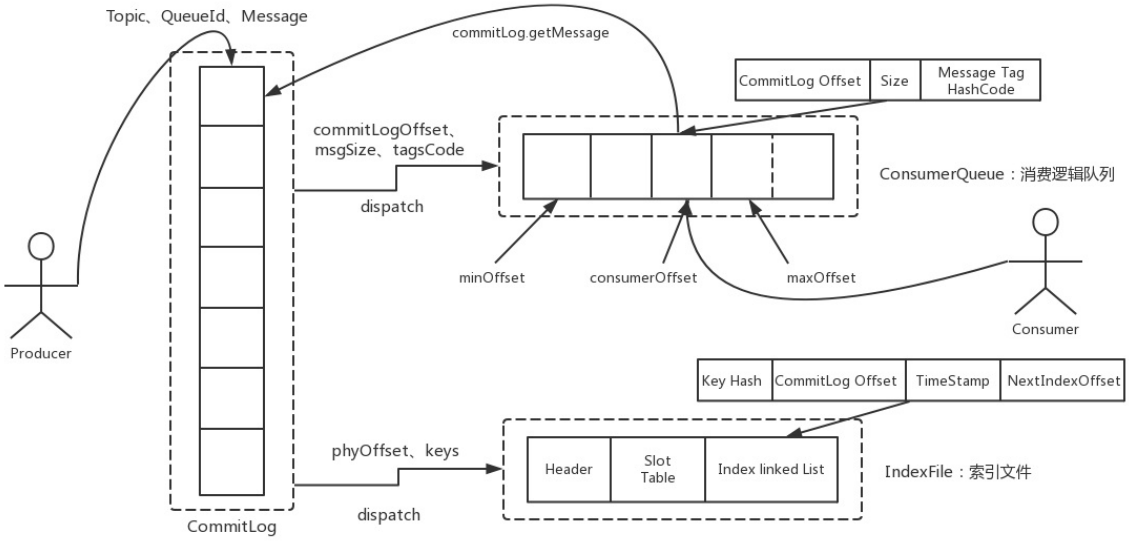
其中，补偿阶段用于解决消息Commit或者Rollback发生超时或者失败的情况。

5.2 RocketMQ事务消息设计

1.事务消息在一阶段对用户不可见

在RocketMQ事务消息的主要流程中，一阶段的消息如何对用户不可见。其中，事务消息相对普通消息最大的特点就是一阶段发送的消息对用户是不可见的。那么，如何做到写入消息但是对用户不可见呢？RocketMQ事务消息的做法是：如果消息是half消息，将备份原消息的主题与消息消费队列，然后改变主题为RMQ_SYS_TRANS_HALF_TOPIC。由于消费组未订阅该主题，故消费端无法消费half类型的消息，然后RocketMQ会开启一个定时任务，从Topic为RMQ_SYS_TRANS_HALF_TOPIC中拉取消息进行消费，根据生产者组获取一个服务提供者发送回查事务状态请求，根据事务状态来决定是提交或回滚消息。

在RocketMQ中，消息在服务端的存储结构如下，每条消息都会有对应的索引信息，Consumer通过ConsumeQueue这个二级索引来读取消息实体内容，其流程如下：



RocketMQ的具体实现策略是：写入的如果事务消息，对消息的Topic和Queue等属性进行替换，同时将原来的Topic和Queue信息存储到消息的属性中，正因为消息主题被替换，故消息并不会转发到该原主题的消息消费队列，消费者无法感知消息的存在，不会消费。其实改变消息主题是RocketMQ的常用“套路”，回想一下延时消息的实现机制。

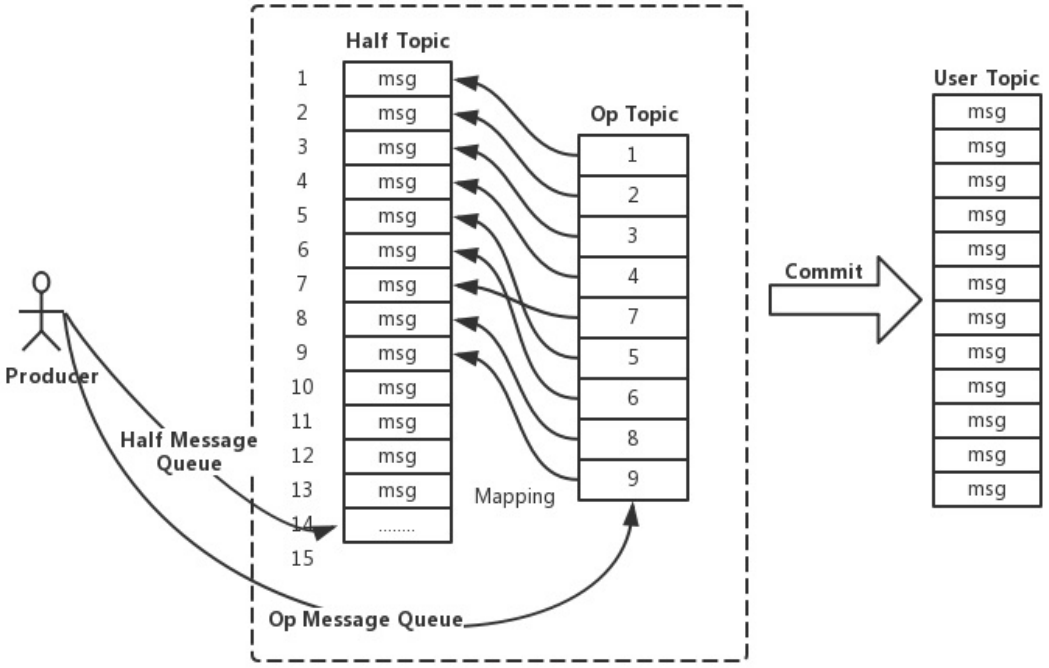
2.Commit和Rollback操作以及Op消息的引入

在完成一阶段写入一条对用户不可见的消息后，二阶段如果是Commit操作，则需要让消息对用户可见；如果是Rollback则需要撤销一阶段的消息。先说Rollback的情况。对于Rollback，本身一阶段的消息对用户是不可见的，其实不需要真正撤销消息（实际上RocketMQ也无法去真正的删除一条消息，因为是顺序写文件的）。但是区别于这条消息没有确定状态（Pending状态，事务悬而未决），需要一个操作来标识这条消息的最终状态。RocketMQ事务消息方案

中引入了Op消息的概念，用Op消息标识事务消息已经确定的状态（Commit或者Rollback）。如果一条事务消息没有对应的Op消息，说明这个事务的状态还无法确定（可能是二阶段失败了）。引入Op消息后，事务消息无论是Commit或者Rollback都会记录一个Op操作。Commit相对于Rollback只是在写入Op消息前创建Half消息的索引。

3.Op消息的存储和对应关系

RocketMQ将Op消息写入到全局一个特定的Topic中通过源码中的方法—TransactionalMessageUtil.buildOpTopic()；这个Topic是一个内部的Topic（像Half消息的Topic一样），不会被用户消费。Op消息的内容为对应的Half消息的存储的Offset，这样通过Op消息能索引到Half消息进行后续的回查操作。



4.Half消息的索引构建

在执行二阶段Commit操作时，需要构建出Half消息的索引。一阶段的Half消息由于是写到一个特殊的Topic，所以二阶段构建索引时需要读取出Half消息，并将Topic和Queue替换成真正的目标的Topic和Queue，之后通过一次普通消息的写入操作来生成一条对用户可见的消息。所以RocketMQ事务消息二阶段其实是利用了一阶段存储的消息的内容，在二阶段时恢复出一条完整的普通消息，然后走一遍消息写入流程。

5.如何处理二阶段失败的消息?

如果在RocketMQ事务消息的二阶段过程中失败了，例如在做Commit操作时，出现网络问题导致Commit失败，那么需要通过一定的策略使这条消息最终被Commit。RocketMQ采用了一种补偿机制，称为“回查”。Broker端对未确定状态的消息发起回查，将消息发送到对应的Producer端（同一个Group的Producer），由Producer根据消息来检查本地事务的状态，进而执行Commit或者Rollback。Broker端通过对比Half消息和Op消息进行事务消息的回查并且推进CheckPoint（记录那些事务消息的状态是确定的）。

值得注意的是，rocketmq并不会无休止的的信息事务状态回查，默认回查15次，如果15次回查还是无法得知事务状态，rocketmq默认回滚该消息。

6 消息查询

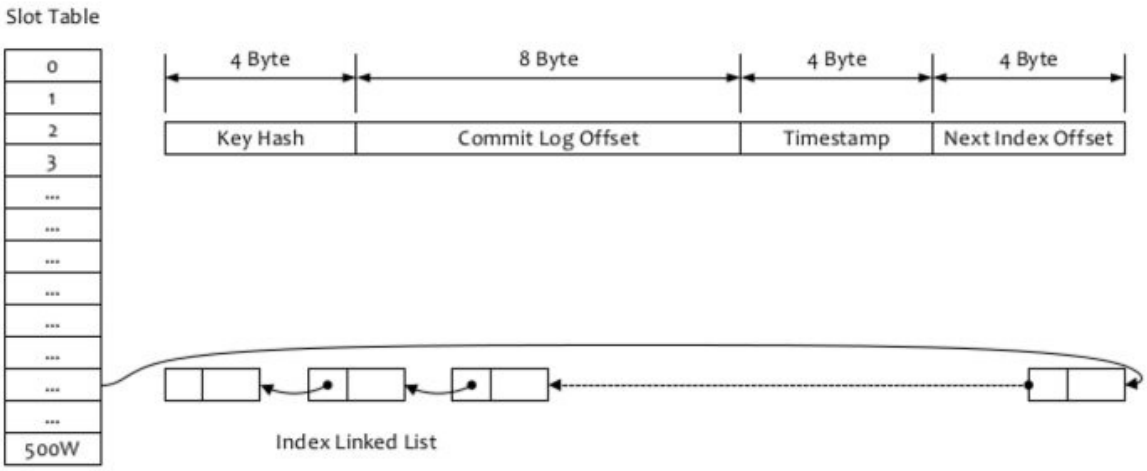
RocketMQ支持按照下面两种维度（“按照Message Id查询消息”、“按照Message Key查询消息”）进行消息查询。

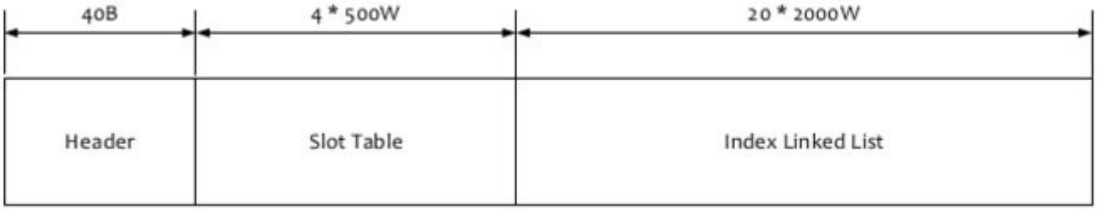
6.1 按照MessageId查询消息

RocketMQ中的MessageId的长度总共有16字节，其中包含了消息存储主机地址（IP地址和端口），消息Commit Log offset。“按照MessageId查询消息”在RocketMQ中具体做法是：Client端从MessageId中解析出Broker的地址（IP地址和端口）和Commit Log的偏移地址后封装成一个RPC请求后通过Remoting通信层发送（业务请求码：VIEW_MESSAGE_BY_ID）。Broker端走的是QueryMessageProcessor，读取消息的过程用其中的 commitLog offset 和 size 去 commitLog 中找到真正的记录并解析成一个完整的消息返回。

6.2 按照Message Key查询消息

“按照Message Key查询消息”，主要是基于RocketMQ的IndexFile索引文件来实现的。RocketMQ的索引文件逻辑结构，类似JDK中HashMap的实现。索引文件的具体结构如下：





IndexFile索引文件为用户提供通过“按照Message Key查询消息”的消息索引查询服务，IndexFile文件的存储位置是：$HOME\store\index${fileName}，文件名fileName是以创建时的时间戳命名的，文件大小是固定的，等于40+500W*4+2000W*20= 420000040个字节大小。如果消息的properties中设置了UNIQ_KEY这个属性，就用 topic + “#” + UNIQ_KEY的value作为 key 来做写入操作。如果消息设置了KEYS属性（多个KEY以空格分隔），也会用 topic + “#” + KEY 来做索引。

其中的索引数据包含了Key Hash/CommitLog Offset/Timestamp/NextIndex offset 这四个字段，一共20 Byte。NextIndex offset 即前面读出来的 slotValue，如果有 hash冲突，就可以用这个字段将所有冲突的索引用链表的方式串起来了。Timestamp记录的是消息storeTimestamp之间的差，并不是一个绝对的时间。整个Index File的结构如图，40 Byte 的Header用于保存一些总的统计信息，4*500W的 Slot Table并不保存真正的索引数据，而是保存每个槽位对应的单向链表的头。20*2000W 是真正的索引数据，即一个 Index File 可以保存 2000W个索引。

“按照Message Key查询消息”的方式，RocketMQ的具体做法是，主要通过Broker端的QueryMessageProcessor业务处理器来查询，读取消息的过程就是用topic和key找到IndexFile索引文件中的一条记录，根据其中的commitLog offset从CommitLog文件中读取消息的实体内容。

样例

1 基本样例

在基本样例中我们提供如下的功能场景：

- 使用RocketMQ发送三种类型的消息：同步消息、异步消息和单向消息。其中前两种消息是可靠的，因为会有发送是否成功的应答。
- 使用RocketMQ来消费接收到的消息。

1.1 加入依赖：

`maven:`

```
<dependency>
    <groupId>org.apache.rocketmq</groupId>
    <artifactId>rocketmq-client</artifactId>
    <version>4.3.0</version>
</dependency>
```

`gradle`

```
compile 'org.apache.rocketmq:rocketmq-client:4.3.0'
```

1.2 消息发送

1、Producer端发送同步消息

这种可靠性同步地发送方式使用的比较广泛，比如：重要的消息通知，短信通知。

```
public class SyncProducer {
    public static void main(String[] args) throws Exception {
        // 实例化消息生产者Producer
        DefaultMQProducer producer = new DefaultMQProducer("please_rename_unique_group_name");
        // 设置NameServer的地址
        producer.setNamesrvAddr("localhost:9876");
        // 启动Producer实例
        producer.start();
        for (int i = 0; i < 100; i++) {
            // 创建消息，并指定Topic，Tag和消息体
            Message msg = new Message("TopicTest" /* Topic */,
            "TagA" /* Tag */,
            ("Hello RocketMQ " + i).getBytes(RemotingHelper.DEFAULT_CHARSET) /* Message body */
            );
            // 发送消息到一个Broker
            SendResult sendResult = producer.send(msg);
            // 通过sendResult返回消息是否成功送达
            System.out.printf("%s%n", sendResult);
        }
        // 如果不再发送消息，关闭Producer实例。
        producer.shutdown();
    }
}
```

2、发送异步消息

异步消息通常用在对响应时间敏感的业务场景，即发送端不能容忍长时间地等待Broker的响应。

```
public class AsyncProducer {
    public static void main(String[] args) throws Exception {
        // 实例化消息生产者Producer
        DefaultMQProducer producer = new DefaultMQProducer("please_rename_unique_group_name");
        // 设置NameServer的地址
        producer.setNamesrvAddr("localhost:9876");
        // 启动Producer实例
        producer.start();
        producer.setRetryTimesWhenSendAsyncFailed(0);
        for (int i = 0; i < 100; i++) {
                final int index = i;
                // 创建消息，并指定Topic，Tag和消息体
                Message msg = new Message("TopicTest",
                    "TagA",
                    "OrderID188",
                    "Hello world".getBytes(RemotingHelper.DEFAULT_CHARSET));
                // SendCallback接收异步返回结果的回调
                producer.send(msg, new SendCallback() {
                    @Override
                    public void onSuccess(SendResult sendResult) {
                        System.out.printf("%-10d OK %s %n", index,
                            sendResult.getMsgId());
                    }
                    @Override
                    public void onException(Throwable e) {
                        System.out.printf("%-10d Exception %s %n", index, e);
                        e.printStackTrace();
                    }
                });
        }
        // 如果不再发送消息，关闭Producer实例。
        producer.shutdown();
    }
}
```

3、单向发送消息

这种方式主要用在不特别关心发送结果的场景，例如日志发送。

```
public class OnewayProducer {
    public static void main(String[] args) throws Exception{
        // 实例化消息生产者Producer
        DefaultMQProducer producer = new DefaultMQProducer("please_rename_unique_group_name");
        // 设置NameServer的地址
        producer.setNamesrvAddr("localhost:9876");
        // 启动Producer实例
        producer.start();
        for (int i = 0; i < 100; i++) {
            // 创建消息，并指定Topic，Tag和消息体
            Message msg = new Message("TopicTest" /* Topic */,
                "TagA" /* Tag */,
                ("Hello RocketMQ " + i).getBytes(RemotingHelper.DEFAULT_CHARSET) /* Message body */
            );
            // 发送单向消息，没有任何返回结果
            producer.sendOneway(msg);

        }
        // 如果不再发送消息，关闭Producer实例。
        producer.shutdown();
    }
}
```

1.3 消费消息

```
public class Consumer {

    public static void main(String[] args) throws InterruptedException, MQClientException {

        // 实例化消费者
        DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("please_rename_unique_group_name");

        // 设置NameServer的地址
        consumer.setNamesrvAddr("localhost:9876");

        // 订阅一个或者多个Topic，以及Tag来过滤需要消费的消息
        consumer.subscribe("TopicTest", "*");
        // 注册回调实现类来处理从broker拉取回来的消息
        consumer.registerMessageListener(new MessageListenerConcurrently() {
            @Override
            public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs, ConsumeConcurrentlyContext c
ontext) {
                System.out.printf("%s Receive New Messages: %s %n", Thread.currentThread().getName(), msgs);
                // 标记该消息已经被成功消费
                return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
            }
        });
        // 启动消费者实例
        consumer.start();
        System.out.printf("Consumer Started.%n");
    }
}
```

2 顺序消息样例

消息有序指的是可以按照消息的发送顺序来消费(FIFO)。RocketMQ可以严格的保证消息有序，可以分为分区有序或者全局有序。

顺序消费的原理解析，在默认的情况下消息发送会采取Round Robin轮询方式把消息发送到不同的queue(分区队列)；而消费消息的时候从多个queue上拉取消息，这种情况发送和消费是不能保证顺序。但是如果控制发送的顺序消息只依次发送到同一个queue中，消费的时候只从这个queue上依次拉取，则就保证了顺序。当发送和消费参与的queue只有一个，则是全局有序；如果多个queue参与，则为分区有序，即相对每个queue，消息都是有序的。

下面用订单进行分区有序的示例。一个订单的顺序流程是：创建、付款、推送、完成。订单号相同的消息会被先后发送到同一个队列中，消费时，同一个OrderId获取到的肯定是同一个队列。

2.1 顺序消息生产

```
package org.apache.rocketmq.example.order2;

import org.apache.rocketmq.client.producer.DefaultMQProducer;
import org.apache.rocketmq.client.producer.MessageQueueSelector;
import org.apache.rocketmq.client.producer.SendResult;
import org.apache.rocketmq.common.message.Message;
import org.apache.rocketmq.common.message.MessageQueue;

import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

/**
* Producer，发送顺序消息
*/
```

```
public class Producer {

    public static void main(String[] args) throws Exception {
        DefaultMQProducer producer = new DefaultMQProducer("please_rename_unique_group_name");

        producer.setNamesrvAddr("127.0.0.1:9876");

        producer.start();

        String[] tags = new String[]{"TagA", "TagC", "TagD"};

        // 订单列表
        List<OrderStep> orderList = new Producer().buildOrders();

        Date date = new Date();
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        String dateStr = sdf.format(date);
        for (int i = 0; i < 10; i++) {
            // 加个时间前缀
            String body = dateStr + " Hello RocketMQ " + orderList.get(i);
            Message msg = new Message("TopicTest", tags[i % tags.length], "KEY" + i, body.getBytes());

            SendResult sendResult = producer.send(msg, new MessageQueueSelector() {
                @Override
                public MessageQueue select(List<MessageQueue> mqs, Message msg, Object arg) {
                    Long id = (Long) arg;  //根据订单id选择发送queue
                    long index = id % mqs.size();
                    return mqs.get((int) index);
                }
            }, orderList.get(i).getOrderId());//订单id

            System.out.println(String.format("SendResult status:%s, queueId:%d, body:%s",
                sendResult.getSendStatus(),
                sendResult.getMessageQueue().getQueueId(),
                body));
        }

        producer.shutdown();
    }

    /**
     * 订单的步骤
     */
    private static class OrderStep {
        private long orderId;
        private String desc;

        public long getOrderId() {
            return orderId;
        }

        public void setOrderId(long orderId) {
            this.orderId = orderId;
        }

        public String getDesc() {
            return desc;
        }

        public void setDesc(String desc) {
            this.desc = desc;
        }

        @Override
        public String toString() {
            return "OrderStep{" +
                "orderId=" + orderId +
                ", desc='" + desc + '\'' +
                '}';
```

```
        }
    }

    /**
     * 生成模拟订单数据
     */
    private List<OrderStep> buildOrders() {
        List<OrderStep> orderList = new ArrayList<OrderStep>();

        OrderStep orderDemo = new OrderStep();
        orderDemo.setOrderId(15103111039L);
        orderDemo.setDesc("创建");
        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(15103111065L);
        orderDemo.setDesc("创建");
        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(15103111039L);
        orderDemo.setDesc("付款");
        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(15103117235L);
        orderDemo.setDesc("创建");
        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(15103111065L);
        orderDemo.setDesc("付款");
        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(15103117235L);
        orderDemo.setDesc("付款");
        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(15103111065L);
        orderDemo.setDesc("完成");
        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(15103111039L);
        orderDemo.setDesc("推送");
        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(15103117235L);
        orderDemo.setDesc("完成");
        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(15103111039L);
        orderDemo.setDesc("完成");
        orderList.add(orderDemo);

        return orderList;
    }
}
```

2.2 顺序消费消息

```
import org.apache.rocketmq.client.consumer.DefaultMQPushConsumer;
import org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlyContext;
```

```
import org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlyStatus;
import org.apache.rocketmq.client.consumer.listener.MessageListenerConcurrently;
import org.apache.rocketmq.common.message.MessageExt;
import java.util.List;

package org.apache.rocketmq.example.order2;

import org.apache.rocketmq.client.consumer.DefaultMQPushConsumer;
import org.apache.rocketmq.client.consumer.listener.ConsumeOrderlyContext;
import org.apache.rocketmq.client.consumer.listener.ConsumeOrderlyStatus;
import org.apache.rocketmq.client.consumer.listener.MessageListenerOrderly;
import org.apache.rocketmq.common.consumer.ConsumeFromWhere;
import org.apache.rocketmq.common.message.MessageExt;

import java.util.List;
import java.util.Random;
import java.util.concurrent.TimeUnit;

/**
* 顺序消息消费，带事务方式（应用可控制Offset什么时候提交）
*/
public class ConsumerInOrder {

   public static void main(String[] args) throws Exception {
       DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("please_rename_unique_group_name_3");
       consumer.setNamesrvAddr("127.0.0.1:9876");
       /**
        * 设置Consumer第一次启动是从队列头部开始消费还是队列尾部开始消费<br>
        * 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);

       consumer.subscribe("TopicTest", "TagA || TagC || TagD");

       consumer.registerMessageListener(new MessageListenerOrderly() {

           Random random = new Random();

           @Override
           public ConsumeOrderlyStatus consumeMessage(List<MessageExt> msgs, ConsumeOrderlyContext context) {
               context.setAutoCommit(true);
               for (MessageExt msg : msgs) {
                   // 可以看到每个queue有唯一的consume线程来消费，订单对每个queue(分区)有序
                   System.out.println("consumeThread=" + Thread.currentThread().getName() + "queueId=" + msg.ge
tQueueId() + ", content:" + new String(msg.getBody()));
               }

               try {
                   //模拟业务逻辑处理中...
                   TimeUnit.SECONDS.sleep(random.nextInt(10));
               } catch (Exception e) {
                   e.printStackTrace();
               }
               return ConsumeOrderlyStatus.SUCCESS;
           }
       });

       consumer.start();

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

3 延时消息样例

3.1 启动消费者等待传入订阅消息

```
import org.apache.rocketmq.client.consumer.DefaultMQPushConsumer;
import org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlyContext;
import org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlyStatus;
import org.apache.rocketmq.client.consumer.listener.MessageListenerConcurrently;
import org.apache.rocketmq.common.message.MessageExt;
import java.util.List;

public class ScheduledMessageConsumer {
   public static void main(String[] args) throws Exception {
      // 实例化消费者
      DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("ExampleConsumer");
      // 订阅Topics
      consumer.subscribe("TestTopic", "*");
      // 注册消息监听者
      consumer.registerMessageListener(new MessageListenerConcurrently() {
          @Override
          public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> messages, ConsumeConcurrentlyContext
 context) {
              for (MessageExt message : messages) {
                  // Print approximate delay time period
                  System.out.println("Receive message[msgId=" + message.getMsgId() + "] " + (System.currentTime
Millis() - message.getStoreTimestamp()) + "ms later");
              }
              return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
          }
      });
      // 启动消费者
      consumer.start();
  }
}
```

3.2 发送延时消息

```
import org.apache.rocketmq.client.producer.DefaultMQProducer;
import org.apache.rocketmq.common.message.Message;

public class ScheduledMessageProducer {
   public static void main(String[] args) throws Exception {
      // 实例化一个生产者来产生延时消息
      DefaultMQProducer producer = new DefaultMQProducer("ExampleProducerGroup");
      // 启动生产者
      producer.start();
      int totalMessagesToSend = 100;
      for (int i = 0; i < totalMessagesToSend; i++) {
          Message message = new Message("TestTopic", ("Hello scheduled message " + i).getBytes());
          // 设置延时等级3,这个消息将在10s之后发送(现在只支持固定的几个时间,详看delayTimeLevel)
          message.setDelayTimeLevel(3);
          // 发送消息
          producer.send(message);
      }
       // 关闭生产者
      producer.shutdown();
  }
}
```

3.3 验证

您将会看到消息的消费比存储时间晚10秒。

3.4 延时消息的使用场景

比如电商里，提交了一个订单就可以发送一个延时消息，1h后去检查这个订单的状态，如果还是未付款就取消订单释放库存。

3.5 延时消息的使用限制

```
// org/apache/rocketmq/store/config/MessageStoreConfig.java

private String messageDelayLevel = "1s 5s 10s 30s 1m 2m 3m 4m 5m 6m 7m 8m 9m 10m 20m 30m 1h 2h";
```

现在RocketMq并不支持任意时间的延时，需要设置几个固定的延时等级，从1s到2h分别对应着等级1到18 消息消费失败会进入延时消息队列，消息发送时间与设置的延时等级和重试次数有关，详见代码 `SendMessageProcessor.java`

4 批量消息样例

批量发送消息能显著提高传递小消息的性能。限制是这些批量消息应该有相同的topic，相同的waitStoreMsgOK，而且不能是延时消息。此外，这一批消息的总大小不应超过4MB。

4.1 发送批量消息

如果您每次只发送不超过4MB的消息，则很容易使用批处理，样例如下：

```
String topic = "BatchTest";
List<Message> messages = new ArrayList<>();
messages.add(new Message(topic, "TagA", "OrderID001", "Hello world 0".getBytes()));
messages.add(new Message(topic, "TagA", "OrderID002", "Hello world 1".getBytes()));
messages.add(new Message(topic, "TagA", "OrderID003", "Hello world 2".getBytes()));
try {
   producer.send(messages);
} catch (Exception e) {
   e.printStackTrace();
   //处理error
}
```

4.2 消息列表分割

复杂度只有当你发送大批量时才会增长，你可能不确定它是否超过了大小限制（4MB）。这时候你最好把你的消息列表分割一下：

```


public class ListSplitter implements Iterator<List<Message>> {
   private final int SIZE_LIMIT = 1024 * 1024 * 4;
   private final List<Message> messages;
   private int currIndex;
   public ListSplitter(List<Message> messages) {
           this.messages = messages;
   }
   @Override public boolean hasNext() {
       return currIndex < messages.size();
   }
   @Override public List<Message> next() {
       int nextIndex = currIndex;
       int totalSize = 0;
       for (; nextIndex < messages.size(); nextIndex++) {
           Message message = messages.get(nextIndex);
           int tmpSize = message.getTopic().length() + message.getBody().length;
           Map<String, String> properties = message.getProperties();
           for (Map.Entry<String, String> entry : properties.entrySet()) {
               tmpSize += entry.getKey().length() + entry.getValue().length();
```

```
            }
            tmpSize = tmpSize + 20; // 增加日志的开销20字节
            if (tmpSize > SIZE_LIMIT) {
                //单个消息超过了最大的限制
                //忽略,否则会阻塞分裂的进程
                if (nextIndex - currIndex == 0) {
                   //假如下一个子列表没有元素,则添加这个子列表然后退出循环,否则只是退出循环
                   nextIndex++;
                }
                break;
            }
            if (tmpSize + totalSize > SIZE_LIMIT) {
                break;
            } else {
                totalSize += tmpSize;
            }

        }
        List<Message> subList = messages.subList(currIndex, nextIndex);
        currIndex = nextIndex;
        return subList;
    }
}
//把大的消息分裂成若干个小的消息
ListSplitter splitter = new ListSplitter(messages);
while (splitter.hasNext()) {
  try {
      List<Message>  listItem = splitter.next();
      producer.send(listItem);
  } catch (Exception e) {
      e.printStackTrace();
      //处理error
  }
}
```

5 过滤消息样例

在大多数情况下，TAG是一个简单而有用的设计，其可以来选择您想要的消息。例如：

```
DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("CID_EXAMPLE");
consumer.subscribe("TOPIC", "TAGA || TAGB || TAGC");
```

消费者将接收包含TAGA或TAGB或TAGC的消息。但是限制是一个消息只能有一个标签，这对于复杂的场景可能不起作用。在这种情况下，可以使用SQL表达式筛选消息。SQL特性可以通过发送消息时的属性来进行计算。在RocketMQ定义的语法下，可以实现一些简单的逻辑。下面是一个例子：

```
------------
| message  |
|----------|  a > 5 AND b = 'abc'
| a = 10   |  --------------------> Gotten
| b = 'abc'|
| c = true |
------------
------------
| message  |
|----------|   a > 5 AND b = 'abc'
| a = 1    |  --------------------> Missed
| b = 'abc'|
| c = true |
------------
```

5.1 基本语法

RocketMQ只定义了一些基本语法来支持这个特性。你也可以很容易地扩展它。

- 数值比较，比如：**>，>=，<，<=，BETWEEN，=；**
- 字符比较，比如：**=，<>，IN；**
- **IS NULL** 或者 **IS NOT NULL；**
- 逻辑符号 **AND，OR，NOT；**

常量支持类型为：

- 数值，比如：**123，3.1415；**
- 字符，比如：**'abc'，必须用单引号包裹起来；**
- **NULL**，特殊的常量
- 布尔值，**TRUE** 或 **FALSE**

只有使用push模式的消费者才能用使用SQL92标准的sql语句，接口如下：

```
public void subscribe(finalString topic, final MessageSelector messageSelector)
```

5.2 使用样例

1、生产者样例

发送消息时，你能通过 `putUserProperty` 来设置消息的属性

```
DefaultMQProducer producer = new DefaultMQProducer("please_rename_unique_group_name");
producer.start();
Message msg = new Message("TopicTest",
   tag,
   ("Hello RocketMQ " + i).getBytes(RemotingHelper.DEFAULT_CHARSET)
);
// 设置一些属性
msg.putUserProperty("a", String.valueOf(i));
SendResult sendResult = producer.send(msg);

producer.shutdown();
```

2、消费者样例

用MessageSelector.bySql来使用sql筛选消息

```
DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("please_rename_unique_group_name_4");
// 只有订阅的消息有这个属性a, a >=0 and a <= 3
consumer.subscribe("TopicTest", MessageSelector.bySql("a between 0 and 3");
consumer.registerMessageListener(new MessageListenerConcurrently() {
   @Override
   public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs, ConsumeConcurrentlyContext context) {
       return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
   }
});
consumer.start();
```

6 消息事务样例

事务消息共有三种状态，提交状态、回滚状态、中间状态：

- TransactionStatus.CommitTransaction: 提交事务，它允许消费者消费此消息。
- TransactionStatus.RollbackTransaction: 回滚事务，它代表该消息将被删除，不允许被消费。

- TransactionStatus.Unknown: 中间状态，它代表需要检查消息队列来确定状态。

6.1 发送事务消息样例

1、创建事务性生产者

使用 `TransactionMQProducer` 类创建生产者，并指定唯一的 `ProducerGroup`，就可以设置自定义线程池来处理这些检查请求。执行本地事务后、需要根据执行结果对消息队列进行回复。回传的事务状态在请参考前一节。

```
import org.apache.rocketmq.client.consumer.DefaultMQPushConsumer;
import org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlyContext;
import org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlyStatus;
import org.apache.rocketmq.client.consumer.listener.MessageListenerConcurrently;
import org.apache.rocketmq.common.message.MessageExt;
import java.util.List;
public class TransactionProducer {
   public static void main(String[] args) throws MQClientException, InterruptedException {
       TransactionListener transactionListener = new TransactionListenerImpl();
       TransactionMQProducer producer = new TransactionMQProducer("please_rename_unique_group_name");
       ExecutorService executorService = new ThreadPoolExecutor(2, 5, 100, TimeUnit.SECONDS, new ArrayBlockingQ
ueue<Runnable>(2000), new ThreadFactory() {
           @Override
           public Thread newThread(Runnable r) {
               Thread thread = new Thread(r);
               thread.setName("client-transaction-msg-check-thread");
               return thread;
           }
       });
       producer.setExecutorService(executorService);
       producer.setTransactionListener(transactionListener);
       producer.start();
       String[] tags = new String[] {"TagA", "TagB", "TagC", "TagD", "TagE"};
       for (int i = 0; i < 10; i++) {
           try {
               Message msg =
                   new Message("TopicTest1234", tags[i % tags.length], "KEY" + i,
                       ("Hello RocketMQ " + i).getBytes(RemotingHelper.DEFAULT_CHARSET));
               SendResult sendResult = producer.sendMessageInTransaction(msg, null);
               System.out.printf("%s%n", sendResult);
               Thread.sleep(10);
           } catch (MQClientException | UnsupportedEncodingException e) {
               e.printStackTrace();
           }
       }
       for (int i = 0; i < 100000; i++) {
           Thread.sleep(1000);
       }
       producer.shutdown();
   }
}
```

2、实现事务的监听接口

当发送半消息成功时，我们使用 `executeLocalTransaction` 方法来执行本地事务。它返回前一节中提到的三个事务状态之一。`checkLocalTranscation` 方法用于检查本地事务状态，并回应消息队列的检查请求。它也是返回前一节中提到的三个事务状态之一。

```
public class TransactionListenerImpl implements TransactionListener {
  private AtomicInteger transactionIndex = new AtomicInteger(0);
  private ConcurrentHashMap<String, Integer> localTrans = new ConcurrentHashMap<>();
  @Override
  public LocalTransactionState executeLocalTransaction(Message msg, Object arg) {
      int value = transactionIndex.getAndIncrement();
```

```
        int status = value % 3;
        localTrans.put(msg.getTransactionId(), status);
        return LocalTransactionState.UNKNOW;
    }
    @Override
    public LocalTransactionState checkLocalTransaction(MessageExt msg) {
        Integer status = localTrans.get(msg.getTransactionId());
        if (null != status) {
            switch (status) {
                case 0:
                    return LocalTransactionState.UNKNOW;
                case 1:
                    return LocalTransactionState.COMMIT_MESSAGE;
                case 2:
                    return LocalTransactionState.ROLLBACK_MESSAGE;
            }
        }
        return LocalTransactionState.COMMIT_MESSAGE;
    }
}
```

6.2 事务消息使用上的限制

1. 事务消息不支持延时消息和批量消息。
2. 为了避免单个消息被检查太多次而导致半队列消息累积，我们默认将单个消息的检查次数限制为 15 次，但是用户可以通过 Broker 配置文件的 `transactionCheckMax` 参数来修改此限制。如果已经检查某条消息超过 N 次的话（ N = `transactionCheckMax` ） 则 Broker 将丢弃此消息，并在默认情况下同时打印错误日志。用户可以通过重写 `AbstractTransactionCheckListener` 类来修改这个行为。
3. 事务消息将在 Broker 配置文件中的参数 transactionMsgTimeout 这样的特定时间长度之后被检查。当发送事务消息时，用户还可以通过设置用户属性 CHECK_IMMUNITY_TIME_IN_SECONDS 来改变这个限制，该参数优先于 `transactionMsgTimeout` 参数。
4. 事务性消息可能不止一次被检查或消费。
5. 提交给用户的目标主题消息可能会失败，目前这依日志的记录而定。它的高可用性通过 RocketMQ 本身的高可用性机制来保证，如果希望确保事务消息不丢失、并且事务完整性得到保证，建议使用同步的双重写入机制。
6. 事务消息的生产者 ID 不能与其他类型消息的生产者 ID 共享。与其他类型的消息不同，事务消息允许反向查询、MQ服务器能通过它们的生产者 ID 查询到消费者。

7 Logappender样例

RocketMQ日志提供log4j、log4j2和logback日志框架作为业务应用，下面是配置样例

7.1 log4j样例

按下面样例使用log4j属性配置

```
log4j.appender.mq=org.apache.rocketmq.logappender.log4j.RocketmqLog4jAppender
log4j.appender.mq.Tag=yourTag
log4j.appender.mq.Topic=yourLogTopic
log4j.appender.mq.ProducerGroup=yourLogGroup
log4j.appender.mq.NameServerAddress=yourRocketmqNameserverAddress
log4j.appender.mq.layout=org.apache.log4j.PatternLayout
log4j.appender.mq.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-4r [%t] (%F:%L) %-5p - %m%n
```

按下面样例使用log4j xml配置来使用异步添加日志

```
<appender name="mqAppender1"class="org.apache.rocketmq.logappender.log4j.RocketmqLog4jAppender">
  <param name="Tag" value="yourTag" />
```

```
  <param name="Topic" value="yourLogTopic" />
  <param name="ProducerGroup" value="yourLogGroup" />
  <param name="NameServerAddress" value="yourRocketmqNameserverAddress"/>
  <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern" value="%d{yyyy-MM-dd HH:mm:ss}-%p %t %c - %m%n" />
  </layout>
</appender>
<appender name="mqAsyncAppender1"class="org.apache.log4j.AsyncAppender">
  <param name="BufferSize" value="1024" />
  <param name="Blocking" value="false" />
  <appender-ref ref="mqAppender1"/>
</appender>
```

7.2 log4j2样例

用log4j2时，配置如下，如果想要非阻塞，只需要使用异步添加引用即可

```
<RocketMQ name="rocketmqAppender" producerGroup="yourLogGroup" nameServerAddress="yourRocketmqNameserverAddress
"
   topic="yourLogTopic" tag="yourTag">
  <PatternLayout pattern="%d [%p] hahahah %c %m%n"/>
</RocketMQ>
```

7.3 logback样例

```
<appender name="mqAppender1"class="org.apache.rocketmq.logappender.logback.RocketmqLogbackAppender">
  <tag>yourTag</tag>
  <topic>yourLogTopic</topic>
  <producerGroup>yourLogGroup</producerGroup>
  <nameServerAddress>yourRocketmqNameserverAddress</nameServerAddress>
  <layout>
      <pattern>%date %p %t - %m%n</pattern>
  </layout>
</appender>
<appender name="mqAsyncAppender1"class="ch.qos.logback.classic.AsyncAppender">
  <queueSize>1024</queueSize>
  <discardingThreshold>80</discardingThreshold>
  <maxFlushTime>2000</maxFlushTime>
  <neverBlock>true</neverBlock>
  <appender-ref ref="mqAppender1"/>
</appender>
```

8 OpenMessaging样例

OpenMessaging旨在建立消息和流处理规范，以为金融、电子商务、物联网和大数据领域提供通用框架及工业级指导方案。在分布式异构环境中，设计原则是面向云、简单、灵活和独立于语言。符合这些规范将帮助企业方便的开发跨平台和操作系统的异构消息传递应用程序。提供了openmessaging-api 0.3.0-alpha的部分实现，下面的示例演示如何基于OpenMessaging访问RocketMQ。

8.1 OMSProducer样例

下面的示例演示如何在同步、异步或单向传输中向RocketMQ代理发送消息。

```
import io.openmessaging.Future;
import io.openmessaging.FutureListener;
import io.openmessaging.Message;
import io.openmessaging.MessagingAccessPoint;
import io.openmessaging.OMS;
```

```
import io.openmessaging.producer.Producer;
import io.openmessaging.producer.SendResult;
import java.nio.charset.Charset;
import java.util.concurrent.CountDownLatch;

public class SimpleProducer {
    public static void main(String[] args) {
       final MessagingAccessPoint messagingAccessPoint =
           OMS.getMessagingAccessPoint("oms:rocketmq://localhost:9876/default:default");
       final Producer producer = messagingAccessPoint.createProducer();
       messagingAccessPoint.startup();
       System.out.printf("MessagingAccessPoint startup OK%n");
       producer.startup();
       System.out.printf("Producer startup OK%n");
       {
           Message message = producer.createBytesMessage("OMS_HELLO_TOPIC", "OMS_HELLO_BODY".getBytes(Charset.f
orName("UTF-8")));
           SendResult sendResult = producer.send(message);
           //final Void aVoid = result.get(3000L);
           System.out.printf("Send async message OK, msgId: %s%n", sendResult.messageId());
       }
       final CountDownLatch countDownLatch = new CountDownLatch(1);
       {
           final Future<SendResult> result = producer.sendAsync(producer.createBytesMessage("OMS_HELLO_TOPIC",
"OMS_HELLO_BODY".getBytes(Charset.forName("UTF-8"))));
           result.addListener(new FutureListener<SendResult>() {
               @Override
               public void operationComplete(Future<SendResult> future) {
                   if (future.getThrowable() != null) {
                       System.out.printf("Send async message Failed, error: %s%n", future.getThrowable().getMes
sage());
                   } else {
                       System.out.printf("Send async message OK, msgId: %s%n", future.get().messageId());
                   }
                   countDownLatch.countDown();
               }
           });
       }
       {
           producer.sendOneway(producer.createBytesMessage("OMS_HELLO_TOPIC", "OMS_HELLO_BODY".getBytes(Charset
.forName("UTF-8"))));
           System.out.printf("Send oneway message OK%n");
       }
       try {
           countDownLatch.await();
           Thread.sleep(500); // 等一些时间来发送消息
       } catch (InterruptedException ignore) {
       }
       producer.shutdown();
   }
}
```

8.2 OMSPullConsumer

用OMS PullConsumer 来从指定的队列中拉取消息

```
import io.openmessaging.Message;
import io.openmessaging.MessagingAccessPoint;
import io.openmessaging.OMS;
import io.openmessaging.OMSBuiltinKeys;
import io.openmessaging.consumer.PullConsumer;
import io.openmessaging.producer.Producer;
import io.openmessaging.producer.SendResult;

public class SimplePullConsumer {
    public static void main(String[] args) {
       final MessagingAccessPoint messagingAccessPoint =
```

```
            OMS.getMessagingAccessPoint("oms:rocketmq://localhost:9876/default:default");
        messagingAccessPoint.startup();
        final Producer producer = messagingAccessPoint.createProducer();
        final PullConsumer consumer = messagingAccessPoint.createPullConsumer(
            OMS.newKeyValue().put(OMSBuiltinKeys.CONSUMER_ID, "OMS_CONSUMER"));
        messagingAccessPoint.startup();
        System.out.printf("MessagingAccessPoint startup OK%n");
        final String queueName = "TopicTest";
        producer.startup();
        Message msg = producer.createBytesMessage(queueName, "Hello Open Messaging".getBytes());
        SendResult sendResult = producer.send(msg);
        System.out.printf("Send Message OK. MsgId: %s%n", sendResult.messageId());
        producer.shutdown();
        consumer.attachQueue(queueName);
        consumer.startup();
        System.out.printf("Consumer startup OK%n");
        // 运行直到发现一个消息被发送了
        boolean stop = false;
        while (!stop) {
            Message message = consumer.receive();
            if (message != null) {
                String msgId = message.sysHeaders().getString(Message.BuiltinKeys.MESSAGE_ID);
                System.out.printf("Received one message: %s%n", msgId);
                consumer.ack(msgId);
                if (!stop) {
                    stop = msgId.equalsIgnoreCase(sendResult.messageId());
                }
            } else {
                System.out.printf("Return without any message%n");
            }
        }
        consumer.shutdown();
        messagingAccessPoint.shutdown();
    }
}
```

8.3 OMSPushConsumer

以下示范如何将 OMS PushConsumer 添加到指定的队列，并通过 MessageListener 消费这些消息。

```
import io.openmessaging.Message;
import io.openmessaging.MessagingAccessPoint;
import io.openmessaging.OMS;
import io.openmessaging.OMSBuiltinKeys;
import io.openmessaging.consumer.MessageListener;
import io.openmessaging.consumer.PushConsumer;

public class SimplePushConsumer {
    public static void main(String[] args) {
        final MessagingAccessPoint messagingAccessPoint = OMS
            .getMessagingAccessPoint("oms:rocketmq://localhost:9876/default:default");
        final PushConsumer consumer = messagingAccessPoint.
            createPushConsumer(OMS.newKeyValue().put(OMSBuiltinKeys.CONSUMER_ID, "OMS_CONSUMER"));
        messagingAccessPoint.startup();
        System.out.printf("MessagingAccessPoint startup OK%n");
        Runtime.getRuntime().addShutdownHook(new Thread(new Runnable() {
            @Override
            public void run() {
                consumer.shutdown();
                messagingAccessPoint.shutdown();
            }
        }));
        consumer.attachQueue("OMS_HELLO_TOPIC", new MessageListener() {
            @Override
            public void onReceived(Message message, Context context) {
                System.out.printf("Received one message: %s%n", message.sysHeaders().getString(Message.BuiltinKe
ys.MESSAGE_ID));
```

```
                context.ack();
            }
        });
        consumer.startup();
        System.out.printf("Consumer startup OK%n");
    }
}
```

最佳实践

1 生产者

1.1 发送消息注意事项

1 Tags的使用

一个应用尽可能用一个Topic，而消息子类型则可以用tags来标识。tags可以由应用自由设置，只有生产者在发送消息设置了tags，消费方在订阅消息时才可以利用tags通过broker做消息过滤：message.setTags("TagA")。

2 Keys的使用

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。服务器会为每个消息创建索引（哈希索引），应用可以通过topic、key来查询这条消息内容，以及消息被谁消费。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。

```
// 订单Id
String orderId = "20034568923546";
message.setKeys(orderId);
```

3 日志的打印

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。send消息方法只要不抛异常，就代表发送成功。发送成功会有多个状态，在sendResult里定义。以下对每个状态进行说明：

- **SEND_OK**

消息发送成功。要注意的是消息发送成功也不意味着它是可靠的。要确保不会丢失任何消息，还应启用同步Master服务器或同步刷盘，即SYNC_MASTER或SYNC_FLUSH。

- **FLUSH_DISK_TIMEOUT**

消息发送成功但是服务器刷盘超时。此时消息已经进入服务器队列（内存），只有服务器宕机，消息才会丢失。消息存储配置参数中可以设置刷盘方式和同步刷盘时间长度，如果Broker服务器设置了刷盘方式为同步刷盘，即FlushDiskType=SYNC_FLUSH（默认为异步刷盘方式），当Broker服务器未在同步刷盘时间内（默认为5s）完成刷盘，则将返回该状态——刷盘超时。

- **FLUSH_SLAVE_TIMEOUT**

消息发送成功，但是服务器同步到Slave时超时。此时消息已经进入服务器队列，只有服务器宕机，消息才会丢失。如果Broker服务器的角色是同步Master，即SYNC_MASTER（默认是异步Master即ASYNC_MASTER），并且从Broker服务器未在同步刷盘时间（默认为5秒）内完成与主服务器的同步，则将返回该状态——数据同步到Slave服务器超时。

- **SLAVE_NOT_AVAILABLE**

消息发送成功，但是此时Slave不可用。如果Broker服务器的角色是同步Master，即SYNC_MASTER（默认是异步Master服务器即ASYNC_MASTER），但没有配置slave Broker服务器，则将返回该状态——无Slave服务器可用。

1.2 消息发送失败处理方式

Producer的send方法本身支持内部重试，重试逻辑如下：

- 至多重试2次（同步发送为2次，异步发送为0次）。
- 如果发送失败，则轮转到下一个Broker。这个方法的总耗时时间不超过sendMsgTimeout设置的值，默认10s。
- 如果本身向broker发送消息产生超时异常，就不会再重试。

以上策略也是在一定程度上保证了消息可以发送成功。如果业务对消息可靠性要求比较高，建议应用增加相应的重试逻辑：比如调用send同步方法发送失败时，则尝试将消息存储到db，然后由后台线程定时重试，确保消息一定到达Broker。

上述db重试方式为什么没有集成到MQ客户端内部做，而是要求应用自己去完成，主要基于以下几点考虑：首先，MQ的客户端设计为无状态模式，方便任意的水平扩展，且对机器资源的消耗仅仅是cpu、内存、网络。其次，如果MQ客户端内部集成一个KV存储模块，那么数据只有同步落盘才能较可靠，而同步落盘本身性能开销较大，所以通常会采用异步落盘，又由于应用关闭过程不受MQ运维人员控制，可能经常会发生 kill -9 这样暴力方式关闭，造成数据没有及时落盘而丢失。第三，Producer所在机器的可靠性较低，一般为虚拟机，不适合存储重要数据。综上，建议重试过程交由应用来控制。

1.3选择oneway形式发送

通常消息的发送是这样一个过程：

- 客户端发送请求到服务器
- 服务器处理请求
- 服务器向客户端返回应答

所以，一次消息发送的耗时时间是上述三个步骤的总和，而某些场景要求耗时非常短，但是对可靠性要求并不高，例如日志收集类应用，此类应用可以采用oneway形式调用，oneway形式只发送请求不等待应答，而发送请求在客户端实现层面仅仅是一个操作系统系统调用的开销，即将数据写入客户端的socket缓冲区，此过程耗时通常在微秒级。

2 消费者

2.1 消费过程幂等

RocketMQ无法避免消息重复（Exactly-Once），所以如果业务对消费重复非常敏感，务必要在业务层面进行去重处理。可以借助关系数据库进行去重。首先需要确定消息的唯一键，可以是msgId，也可以是消息内容中的唯一标识字段，例如订单Id等。在消费之前判断唯一键是否在关系数据库中存在。如果不存在则插入，并消费，否则跳过。（实际过程要考虑原子性问题，判断是否存在可以尝试插入，如果报主键冲突，则插入失败，直接跳过）

msgId一定是全局唯一标识符，但是实际使用中，可能会存在相同的消息有两个不同msgId的情况（消费者主动重发、因客户端重投机制导致的重复等），这种情况就需要使业务字段进行重复消费。

2.2 消费速度慢的处理方式

1 提高消费并行度

绝大部分消息消费行为都属于 IO 密集型，即可能是操作数据库，或者调用 RPC，这类消费行为的消费速度在于后端数据库或者外系统的吞吐量，通过增加消费并行度，可以提高总的消费吞吐量，但是并行度增加到一定程度，反而会下降。所以，应用必须要设置合理的并行度。 如下有几种修改消费并行度的方法：

- 同一个 ConsumerGroup 下，通过增加 Consumer 实例数量来提高并行度（需要注意的是超过订阅队列数的 Consumer 实例无效）。可以通过加机器，或者在已有机器启动多个进程的方式。
- 提高单个 Consumer 的消费并行线程，通过修改参数 consumeThreadMin、consumeThreadMax实现。

2 批量方式消费

某些业务流程如果支持批量方式消费，则可以很大程度上提高消费吞吐量，例如订单扣款类应用，一次处理一个订单耗时 1 s，一次处理 10 个订单可能也只耗时 2 s，这样即可大幅度提高消费的吞吐量，通过设置 consumer的 consumeMessageBatchMaxSize 返个参数，默认是 1，即一次只消费一条消息，例如设置为 N，那么每次消费的消息数小于等于 N。

3 跳过非重要消息

发生消息堆积时，如果消费速度一直追不上发送速度，如果业务对数据要求不高的话，可以选择丢弃不重要的消息。例如，当某个队列的消息数堆积到100000条以上，则尝试丢弃部分或全部消息，这样就可以快速追上发送消息的速度。示例代码如下：

```
public ConsumeConcurrentlyStatus consumeMessage(
        List<MessageExt> msgs,
        ConsumeConcurrentlyContext context) {
    long offset = msgs.get(0).getQueueOffset();
    String maxOffset =
            msgs.get(0).getProperty(Message.PROPERTY_MAX_OFFSET);
    long diff = Long.parseLong(maxOffset) - offset;
    if (diff > 100000) {
        // TODO 消息堆积情况的特殊处理
        return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
    }
    // TODO 正常消费过程
    return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
}
```

4 优化每条消息消费过程

举例如下，某条消息的消费过程如下：

- 根据消息从 DB 查询【数据 1】
- 根据消息从 DB 查询【数据 2】
- 复杂的业务计算
- 向 DB 插入【数据 3】
- 向 DB 插入【数据 4】

这条消息的消费过程中有4次与 DB的 交互，如果按照每次 5ms 计算，那么总共耗时 20ms，假设业务计算耗时 5ms，那么总过耗时 25ms，所以如果能把 4 次 DB 交互优化为 2 次，那么总耗时就可以优化到 15ms，即总体性能提高了 40%。所以应用如果对时延敏感的话，可以把DB部署在SSD硬盘，相比于SCSI磁盘，前者的RT会小很多。

2.3 消费打印日志

如果消息量较少，建议在消费入口方法打印消息，消费耗时等，方便后续排查问题。

```
public ConsumeConcurrentlyStatus consumeMessage(
        List<MessageExt> msgs,
        ConsumeConcurrentlyContext context) {
    log.info("RECEIVE_MSG_BEGIN: " + msgs.toString());
    // TODO 正常消费过程
    return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
}
```

如果能打印每条消息消费耗时，那么在排查消费慢等线上问题时，会更方便。

2.4 其他消费建议

1 关于消费者和订阅

第一件需要注意的事情是，不同的消费者组可以独立的消费一些 topic，并且每个消费者组都有自己的消费偏移量，请确保同一组内的每个消费者订阅信息保持一致。

2 关于有序消息

消费者将锁定每个消息队列，以确保他们被逐个消费，虽然这将会导致性能下降，但是当你关心消息顺序的时候会很有用。我们不建议抛出异常，你可以返回 ConsumeOrderlyStatus.SUSPEND_CURRENT_QUEUE_A_MOMENT 作为替代。

3 关于并发消费

顾名思义，消费者将并发消费这些消息，建议你使用它来获得良好性能，我们不建议抛出异常，你可以返回 ConsumeConcurrentlyStatus.RECONSUME_LATER 作为替代。

4 关于消费状态Consume Status

对于并发的消费监听器，你可以返回 RECONSUME_LATER 来通知消费者现在不能消费这条消息，并且希望可以稍后重新消费它。然后，你可以继续消费其他消息。对于有序的消息监听器，因为你关心它的顺序，所以不能跳过消息，但是你可以返回SUSPEND_CURRENT_QUEUE_A_MOMENT 告诉消费者等待片刻。

5 关于Blocking

不建议阻塞监听器，因为它会阻塞线程池，并最终可能会终止消费进程

6 关于线程数设置

消费者使用 ThreadPoolExecutor 在内部对消息进行消费，所以你可以通过设置 setConsumeThreadMin 或 setConsumeThreadMax 来改变它。

7 关于消费位点

当建立一个新的消费者组时，需要决定是否需要消费已经存在于 Broker 中的历史消息
CONSUME_FROM_LAST_OFFSET 将会忽略历史消息，并消费之后生成的任何消息。
CONSUME_FROM_FIRST_OFFSET 将会消费每个存在于 Broker 中的信息。你也可以使用 CONSUME_FROM_TIMESTAMP 来消费在指定时间戳后产生的消息。

3 Broker

3.1 Broker 角色

Broker 角色分为 ASYNC_MASTER（异步主机）、SYNC_MASTER（同步主机）以及SLAVE（从机）。如果对消息的可靠性要求比较严格，可以采用 SYNC_MASTER加SLAVE的部署方式。如果对消息可靠性要求不高，可以采用ASYNC_MASTER加SLAVE的部署方式。如果只是测试方便，则可以选择仅ASYNC_MASTER或仅SYNC_MASTER的部署方式。

3.2 FlushDiskType

SYNC_FLUSH（同步刷新）相比于ASYNC_FLUSH（异步处理）会损失很多性能，但是也更可靠，所以需要根据实际的业务场景做好权衡。

3.3 Broker 配置

参数名	默认值	说明
listenPort	10911	接受客户端连接的监听端口
namesrvAddr	null	nameServer 地址
brokerIP1	网卡的 InetAddress	当前 broker 监听的 IP
brokerIP2	跟 brokerIP1 一样	存在主从 broker 时，如果在 broker 主节点上配置了 brokerIP2 属性，broker 从节点会连接主节点配置的 brokerIP2 进行同步
brokerName	null	broker 的名称
brokerClusterName	DefaultCluster	本 broker 所属的 Cluser 名称
brokerId	0	broker id, 0 表示 master, 其他的正整数表示 slave
storePathCommitLog	$HOME/store/commitlog/	存储 commit log 的路径
storePathConsumerQueue	$HOME/store/consumequeue/	存储 consume queue 的路径
mapedFileSizeCommitLog	1024 *1024* 1024(1G)	commit log 的映射文件大小
deleteWhen	04	在每天的什么时间删除已经超过文件保留时间的 commit log
fileReserverdTime	72	以小时计算的文件保留时间
brokerRole	ASYNC_MASTER	SYNC_MASTER/ASYNC_MASTER/SLAVE
flushDiskType	ASYNC_FLUSH	SYNC_FLUSH/ASYNC_FLUSH SYNC_FLUSH 模式下的 broker 保证在收到确认生产者之前将消息刷盘。 ASYNC_FLUSH 模式下的 broker 则利用刷盘一组消息的模式，可以取得更好的性能。

4 NameServer

RocketMQ 中，Name Servers 被设计用来做简单的路由管理。其职责包括：

- Brokers 定期向每个名称服务器注册路由数据。
- 名称服务器为客户端，包括生产者，消费者和命令行客户端提供最新的路由信息。

5 客户端配置

相对于RocketMQ的Broker集群，生产者和消费者都是客户端。本小节主要描述生产者和消费者公共的行为配置。

5.1 客户端寻址方式

RocketMQ可以令客户端找到Name Server, 然后通过Name Server再找到Broker。如下所示有多种配置方式，优先级由高到低，高优先级会覆盖低优先级。

- 代码中指定Name Server地址，多个namesrv地址之间用分号分割

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

```
consumer.setNamesrvAddr("192.168.0.1:9876;192.168.0.2:9876");
```

- Java启动参数中指定Name Server地址

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

- 环境变量指定Name Server地址

```
export   NAMESRV_ADDR=192.168.0.1:9876;192.168.0.2:9876
```

- HTTP静态服务器寻址（默认）

客户端启动后，会定时访问一个静态HTTP服务器，地址如下：http://jmenv.tbsite.net:8080/rocketmq/nsaddr，这个URL的返回内容如下：

```
192.168.0.1:9876;192.168.0.2:9876
```

客户端默认每隔2分钟访问一次这个HTTP服务器，并更新本地的Name Server地址。URL已经在代码中硬编码，可通过修改/etc/hosts文件来改变要访问的服务器，例如在/etc/hosts增加如下配置：

```
10.232.22.67    jmenv.taobao.net
```

推荐使用HTTP静态服务器寻址方式，好处是客户端部署简单，且Name Server集群可以热升级。

5.2 客户端配置

DefaultMQProducer、TransactionMQProducer、DefaultMQPushConsumer、DefaultMQPullConsumer都继承于ClientConfig类，ClientConfig为客户端的公共配置类。客户端的配置都是get、set形式，每个参数都可以用spring来配置，也可以在代码中配置，例如namesrvAddr这个参数可以这样配置，producer.setNamesrvAddr("192.168.0.1:9876")，其他参数同理。

1 客户端的公共配置

参数名	默认值	说明
namesrvAddr		Name Server地址列表，多个NameServer地址用分号隔开
clientIP	本机IP	客户端本机IP地址，某些机器会发生无法识别客户端IP地址情况，需要应用在代码中强制指定
instanceName	DEFAULT	客户端实例名称，客户端创建的多个Producer、Consumer实际是共用一个内部实例（这个实例包含网络连接、线程资源等）
clientCallbackExecutorThreads	4	通信层异步回调线程数
pollNameServerInteval	30000	轮询Name Server间隔时间，单位毫秒
heartbeatBrokerInterval	30000	向Broker发送心跳间隔时间，单位毫秒
persistConsumerOffsetInterval	5000	持久化Consumer消费进度间隔时间，单位毫秒

2 Producer配置

参数名	默认值	说明

producerGroup	DEFAULT_PRODUCER	Producer组名，多个Producer如果属于一个应用，发送同样的消息，则应该将它们归为同一组
createTopicKey	TBW102	在发送消息时，自动创建服务器不存在的topic，需要指定Key，该Key可用于配置发送消息所在topic的默认路由。
defaultTopicQueueNums	4	在发送消息，自动创建服务器不存在的topic时，默认创建的队列数
sendMsgTimeout	10000	发送消息超时时间，单位毫秒
compressMsgBodyOverHowmuch	4096	消息Body超过多大开始压缩（Consumer收到消息会自动解压缩），单位字节
retryAnotherBrokerWhenNotStoreOK	FALSE	如果发送消息返回sendResult，但是sendStatus!=SEND_OK，是否重试发送
retryTimesWhenSendFailed	2	如果消息发送失败，最大重试次数，该参数只对同步发送模式起作用
maxMessageSize	4MB	客户端限制的消息大小，超过报错，同时服务端也会限制，所以需要跟服务端配合使用。
transactionCheckListener		事务消息回查监听器，如果发送事务消息，必须设置
checkThreadPoolMinSize	1	Broker回查Producer事务状态时，线程池最小线程数
checkThreadPoolMaxSize	1	Broker回查Producer事务状态时，线程池最大线程数
checkRequestHoldMax	2000	Broker回查Producer事务状态时，Producer本地缓冲请求队列大小
RPCHook	null	该参数是在Producer创建时传入的，包含消息发送前的预处理和消息响应后的处理两个接口，用户可以在第一个接口中做一些安全控制或者其他操作。

3 PushConsumer配置

参数名	默认值	说明
consumerGroup	DEFAULT_CONSUMER	Consumer组名，多个Consumer如果属于一个应用，订阅同样的消息，且消费逻辑一致，则应该将它们归为同一组
messageModel	CLUSTERING	消费模型支持集群消费和广播消费两种
consumeFromWhere	CONSUME_FROM_LAST_OFFSET	Consumer启动后，默认从上次消费的位置开始消费，这包含两种情况：一种是上次消费的位置未过期，则消费从上次中止的位置进行；一种是上次消费位置已经过期，则从当前队列第一条消息开始消费
consumeTimestamp	半个小时前	只有当consumeFromWhere值为CONSUME_FROM_TIMESTAMP时才起作用。
allocateMessageQueueStrategy	AllocateMessageQueueAveragely	Rebalance算法实现策略

subscription		订阅关系
messageListener		消息监听器
offsetStore		消费进度存储
consumeThreadMin	10	消费线程池最小线程数
consumeThreadMax	20	消费线程池最大线程数
consumeConcurrentlyMaxSpan	2000	单队列并行消费允许的最大跨度
pullThresholdForQueue	1000	拉消息本地队列缓存消息最大数
pullInterval	0	拉消息间隔，由于是长轮询，所以为0，但是如果应用为了流控，也可以设置大于0的值，单位毫秒
consumeMessageBatchMaxSize	1	批量消费，一次消费多少条消息
pullBatchSize	32	批量拉消息，一次最多拉多少条

4 PullConsumer配置

参数名	默认值	说明
consumerGroup	DEFAULT_CONSUMER	Consumer组名，多个Consumer如果属于一个应用，订阅同样的消息，且消费逻辑一致，则应该将它们归为同一组
brokerSuspendMaxTimeMillis	20000	长轮询，Consumer拉消息请求在Broker挂起最长时间，单位毫秒
consumerTimeoutMillisWhenSuspend	30000	长轮询，Consumer拉消息请求在Broker挂起超过指定时间，客户端认为超时，单位毫秒
consumerPullTimeoutMillis	10000	非长轮询，拉消息超时时间，单位毫秒
messageModel	BROADCASTING	消息支持两种模式：集群消费和广播消费
messageQueueListener		监听队列变化
offsetStore		消费进度存储
registerTopics		注册的topic集合
allocateMessageQueueStrategy	AllocateMessageQueueAveragely	Rebalance算法实现策略

5 Message数据结构

字段名	默认值	说明
Topic	null	必填，消息所属topic的名称
Body	null	必填，消息体
Tags	null	选填，消息标签，方便服务器过滤使用。目前只支持每个消息设置一个tag

Keys	null	选填，代表这条消息的业务关键词，服务器会根据keys创建哈希索引，设置后，可以在Console系统根据Topic、Keys来查询消息，由于是哈希索引，请尽可能保证key唯一，例如订单号，商品Id等。
Flag	0	选填，完全由应用来设置，RocketMQ不做干预
DelayTimeLevel	0	选填，消息延时级别，0表示不延时，大于0会延时特定的时间才会被消费
WaitStoreMsgOK	TRUE	选填，表示消息是否在服务器落盘后才返回应答。

6 系统配置

本小节主要介绍系统（JVM/OS）相关的配置。

6.1 JVM选项

推荐使用最新发布的JDK 1.8版本。通过设置相同的Xms和Xmx值来防止JVM调整堆大小以获得更好的性能。简单的JVM配置如下所示：

```
-server -Xms8g -Xmx8g -Xmn4g
```

如果您不关心RocketMQ Broker的启动时间，还有一种更好的选择，就是通过“预触摸”Java堆以确保在JVM初始化期间每个页面都将被分配。那些不关心启动时间的人可以启用它： -XX:+AlwaysPreTouch
禁用偏置锁定可能会减少JVM暂停， -XX:-UseBiasedLocking
至于垃圾回收，建议使用带JDK 1.8的G1收集器。

```
-XX:+UseG1GC -XX:G1HeapRegionSize=16m
-XX:G1ReservePercent=25
-XX:InitiatingHeapOccupancyPercent=30
```

这些GC选项看起来有点激进，但事实证明它在我们的生产环境中具有良好的性能。另外不要把-XX:MaxGCPauseMillis的值设置太小，否则JVM将使用一个小的年轻代来实现这个目标，这将导致非常频繁的minor GC，所以建议使用rolling GC日志文件：

```
-XX:+UseGCLogFileRotation
-XX:NumberOfGCLogFiles=5
-XX:GCLogFileSize=30m
```

如果写入GC文件会增加代理的延迟，可以考虑将GC日志文件重定向到内存文件系统：

```
-Xloggc:/dev/shm/mq_gc_%p.log123
```

6.2 Linux内核参数

os.sh脚本在bin文件夹中列出了许多内核参数，可以进行微小的更改然后用于生产用途。下面的参数需要注意，更多细节请参考/proc/sys/vm/*的文档

- **vm.extra_free_kbytes**，告诉VM在后台回收（kswapd）启动的阈值与直接回收（通过分配进程）的阈值之间保留额外的可用内存。RocketMQ使用此参数来避免内存分配中的长延迟。（与具体内核版本相关）
- **vm.min_free_kbytes**，如果将其设置为低于1024KB，将会巧妙的将系统破坏，并且系统在高负载下容易出现死锁。
- **vm.max_map_count**，限制一个进程可能具有的最大内存映射区域数。RocketMQ将使用mmap加载CommitLog和ConsumeQueue，因此建议将为此参数设置较大的值。（agressiveness --> aggressiveness）
- **vm.swappiness**，定义内核交换内存页面的积极程度。较高的值会增加攻击性，较低的值会减少交换量。建议将值设置为10来避免交换延迟。
- **File descriptor limits**，RocketMQ需要为文件（CommitLog和ConsumeQueue）和网络连接打开文件描述符。我

们建议设置文件描述符的值为655350。

- Disk scheduler，RocketMQ建议使用I/O截止时间调度器，它试图为请求提供有保证的延迟。）

消息轨迹

1. 消息轨迹数据关键属性

Producer端	Consumer端	Broker端
生产实例信息	消费实例信息	消息的Topic
发送消息时间	投递时间,投递轮次	消息存储位置
消息是否发送成功	消息是否消费成功	消息的Key值
发送耗时	消费耗时	消息的Tag值

2. 支持消息轨迹集群部署

2.1 Broker端配置文件

这里贴出Broker端开启消息轨迹特性的properties配置文件内容：

```
brokerClusterName=DefaultCluster
brokerName=broker-a
brokerId=0
deleteWhen=04
fileReservedTime=48
brokerRole=ASYNC_MASTER
flushDiskType=ASYNC_FLUSH
storePathRootDir=/data/rocketmq/rootdir-a-m
storePathCommitLog=/data/rocketmq/commitlog-a-m
autoCreateSubscriptionGroup=true
## if msg tracing is open,the flag will be true
traceTopicEnable=true
listenPort=10911
brokerIP1=XX.XX.XX.XX1
namesrvAddr=XX.XX.XX.XX:9876
```

2.2 普通模式

RocketMQ集群中每一个Broker节点均用于存储Client端收集并发送过来的消息轨迹数据。因此，对于RocketMQ集群中的Broker节点数量并无要求和限制。

2.3 物理IO隔离模式

对于消息轨迹数据量较大的场景，可以在RocketMQ集群中选择其中一个Broker节点专用于存储消息轨迹，使得用户普通的消息数据与消息轨迹数据的物理IO完全隔离，互不影响。在该模式下，RockeMQ集群中至少有两个Broker节点，其中一个Broker节点定义为存储消息轨迹数据的服务端。

2.4 启动开启消息轨迹的Broker

```
nohup sh mqbroker -c ../conf/2m-noslave/broker-a.properties &
```

3. 保存消息轨迹的Topic定义

RocketMQ的消息轨迹特性支持两种存储轨迹数据的方式：

3.1 系统级的TraceTopic

在默认情况下，消息轨迹数据是存储于系统级的TraceTopic中(其名称为：**RMQ_SYS_TRACE_TOPIC**)。该Topic在Broker节点启动时，会自动创建出来（如上所叙，需要在Broker端的配置文件中将**traceTopicEnable**的开关变量设置为**true**）。

3.2 用户自定义的TraceTopic

如果用户不准备将消息轨迹的数据存储于系统级的默认TraceTopic，也可以自己定义并创建用户级的Topic来保存轨迹（即为创建普通的Topic用于保存消息轨迹数据）。下面一节会介绍Client客户端的接口如何支持用户自定义的TraceTopic。

4. 支持消息轨迹的Client客户端实践

为了尽可能地减少用户业务系统使用RocketMQ消息轨迹特性的改造工作量，作者在设计时候采用对原来接口增加一个开关参数(**enableMsgTrace**)来实现消息轨迹是否开启；并新增一个自定义参(**customizedTraceTopic**)数来实现用户存储消息轨迹数据至自己创建的用户级Topic。

4.1 发送消息时开启消息轨迹

```
        DefaultMQProducer producer = new DefaultMQProducer("ProducerGroupName",true);
        producer.setNamesrvAddr("XX.XX.XX.XX1");
        producer.start();
            try {
                {
                    Message msg = new Message("TopicTest",
                        "TagA",
                        "OrderID188",
                        "Hello world".getBytes(RemotingHelper.DEFAULT_CHARSET));
                    SendResult sendResult = producer.send(msg);
                    System.out.printf("%s%n", sendResult);
                }

            } catch (Exception e) {
                e.printStackTrace();
            }
```

4.2 订阅消息时开启消息轨迹

```
        DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("CID_JODIE_1",true);
        consumer.subscribe("TopicTest", "*");
        consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_FIRST_OFFSET);
        consumer.setConsumeTimestamp("20181109221800");
        consumer.registerMessageListener(new MessageListenerConcurrently() {
            @Override
            public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs, ConsumeConcurrentlyContext c
ontext) {
                System.out.printf("%s Receive New Messages: %s %n", Thread.currentThread().getName(), msgs);
                return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
            }
        });
        consumer.start();
        System.out.printf("Consumer Started.%n");
```

4.3 支持自定义存储消息轨迹Topic

在上面的发送和订阅消息时候分别将DefaultMQProducer和DefaultMQPushConsumer实例的初始化修改为如下即可支持自定义存储消息轨迹Topic。

```
        ##其中Topic_test11111需要用户自己预先创建，来保存消息轨迹；
        DefaultMQProducer producer = new DefaultMQProducer("ProducerGroupName",true,"Topic_test11111");
        ......

        DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("CID_JODIE_1",true,"Topic_test11111");
        ......
```

权限控制

1.权限控制特性介绍

权限控制（ACL）主要为RocketMQ提供Topic资源级别的用户访问控制。用户在使用RocketMQ权限控制时，可以在Client客户端通过 RPCHook注入AccessKey和SecretKey签名；同时，将对应的权限控制属性（包括Topic访问权限、IP白名单和AccessKey和SecretKey签名等）设置在distribution/conf/plain_acl.yml的配置文件中。Broker端对AccessKey所拥有的权限进行校验，校验不过，抛出异常； ACL客户端可以参考：**org.apache.rocketmq.example.simple**包下面的**AclClient**代码。

2. 权限控制的定义与属性值

2.1权限定义

对RocketMQ的Topic资源访问权限控制定义主要如下表所示，分为以下四种

权限	含义
DENY	拒绝
ANY	PUB 或者 SUB 权限
PUB	发送权限
SUB	订阅权限

2.2 权限定义的关键属性

字段	取值	含义	
globalWhiteRemoteAddresses	*;192.168.*.*;192.168.0.1	全局IP白名单	
accessKey	字符串	Access Key	
secretKey	字符串	Secret Key	
whiteRemoteAddress	*;192.168.*.*;192.168.0.1	用户IP白名单	
admin	true;false	是否管理员账户	
defaultTopicPerm	DENY;PUB;SUB;PUB\	SUB	默认的Topic权限
defaultGroupPerm	DENY;PUB;SUB;PUB\	SUB	默认的ConsumerGroup权限
topicPerms	topic=权限	各个Topic的权限	
groupPerms	group=权限	各个ConsumerGroup的权限	

具体可以参考**distribution/conf/plain_acl.yml**配置文件

3. 支持权限控制的集群部署

在**distribution/conf/plain_acl.yml**配置文件中按照上述说明定义好权限属性后，打开**aclEnable**开关变量即可开启RocketMQ集群的ACL特性。这里贴出Broker端开启ACL特性的properties配置文件内容：

```
brokerClusterName=DefaultCluster
brokerName=broker-a
brokerId=0
deleteWhen=04
fileReservedTime=48
brokerRole=ASYNC_MASTER
flushDiskType=ASYNC_FLUSH
storePathRootDir=/data/rocketmq/rootdir-a-m
storePathCommitLog=/data/rocketmq/commitlog-a-m
autoCreateSubscriptionGroup=true
## if acl is open,the flag will be true
aclEnable=true
listenPort=10911
brokerIP1=XX.XX.XX.XX1
namesrvAddr=XX.XX.XX.XX:9876
```

4. 权限控制主要流程

ACL主要流程分为两部分，主要包括权限解析和权限校验。

4.1 权限解析

Broker端对客户端的RequestCommand请求进行解析，拿到需要鉴权的属性字段。 主要包括： （1）AccessKey：类似于用户名，代指用户主体，权限数据与之对应； （2）Signature：客户根据 SecretKey 签名得到的串，服务端再用SecretKey进行签名验证；

4.2 权限校验

Broker端对权限的校验逻辑主要分为以下几步： （1）检查是否命中全局 IP 白名单；如果是，则认为校验通过；否则走 2； （2）检查是否命中用户 IP 白名单；如果是，则认为校验通过；否则走 3； （3）校验签名，校验不通过，抛出异常；校验通过，则走 4； （4）对用户请求所需的权限 和 用户所拥有的权限进行校验；不通过，抛出异常； 用户所需权限的校验需要注意已下内容： （1）特殊的请求例如 UPDATE_AND_CREATE_TOPIC 等，只能由 admin 账户进行操作； （2）对于某个资源，如果有显性配置权限，则采用配置的权限；如果没有显性配置权限，则采用默认的权限；

5. 热加载修改后权限控制定义

RocketMQ的权限控制存储的默认实现是基于yml配置文件。用户可以动态修改权限控制定义的属性，而不需重新启动Broker服务节点。

6. 权限控制的使用限制

(1)如果ACL与高可用部署(Master/Slave架构)同时启用，那么需要在Broker Master节点的distribution/conf/plain_acl.yml配置文件中 设置全局白名单信息，即为将Slave节点的ip地址设置至Master节点plain_acl.yml配置文件的全局白名单中。

(2)如果ACL与高可用部署(多副本Dledger架构)同时启用，由于出现节点宕机时，Dledger Group组内会自动选主，那么就需要将Dledger Group组 内所有Broker节点的plain_acl.yml配置文件的白名单设置所有Broker节点的ip地址。

特别注意在[4.5.0]版本中即使使用上面所述的白名单也无法解决开启ACL的问题，解决该问题的PR链接

Dledger快速搭建

前言

该文档主要介绍如何快速构建和部署基于 DLedger 的可以自动容灾切换的 RocketMQ 集群。

详细的新集群部署和旧集群升级指南请参考 部署指南。

1. 源码构建

构建分为两个部分，需要先构建 DLedger，然后 构建 RocketMQ

1.1 构建 DLedger

```
git clone https://github.com/openmessaging/openmessaging-storage-dledger.git
```

```
cd openmessaging-storage-dledger
```

```
mvn clean install -DskipTests
```

1.2 构建 RocketMQ

```
git clone https://github.com/apache/rocketmq.git
```

```
cd rocketmq
```

```
git checkout -b store_with_dledger origin/store_with_dledger
```

```
mvn -Prelease-all -DskipTests clean install -U
```

2. 快速部署

在构建成功后

```
cd distribution/target/apache-rocketmq
```

```
sh bin/dledger/fast-try.sh start
```

如果上面的步骤执行成功，可以通过 mqadmin 运维命令查看集群状态。

```
sh bin/mqadmin clusterList -n 127.0.0.1:9876
```

顺利的话，会看到如下内容：

```
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option PermSize=128m; support was removed in 8.0
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=128m; support was removed in 8.0
#Cluster Name     #Broker Name            #BID  #Addr                 #Version               #InTPS(LOAD)       #OutTPS(LOAD) #PCWait(ms) #Hour #SPACE
RaftCluster       RaftNode00              0     30.5.125.75:30931     V4_3_1                  0.00(0,0ms)         0.00(0,0ms)          0 429417.23 -1.0000
RaftCluster       RaftNode00              1     30.5.125.75:30911     V4_3_1                  0.00(0,0ms)         0.00(0,0ms)          0 429417.23 -1.0000
RaftCluster       RaftNode00              2     30.5.125.75:30921     V4_3_1                  0.00(0,0ms)         0.00(0,0ms)          0 429417.23 -1.0000
```

（BID 为 0 的表示 Master，其余都是 Follower）

启动成功，现在可以向集群收发消息，并进行容灾切换测试了。

关闭快速集群，可以执行：

```
sh bin/dledger/fast-try.sh stop
```

快速部署，默认配置在 conf/dledger 里面，默认的存储路径在 /tmp/rmqstore。

3. 容灾切换

部署成功，杀掉 Leader 之后（在上面的例子中，杀掉端口 30931 所在的进程），等待约 10s 左右，用 clusterList 命令查看集群，就会发现 Leader 切换到另一个节点了。

Dledger集群搭建

前言

该文档主要介绍如何部署自动容灾切换的 RocketMQ-on-DLedger Group。

RocketMQ-on-DLedger Group 是指一组**相同名称**的 Broker，至少需要 3 个节点，通过 Raft 自动选举出一个 Leader，其余节点 作为 Follower，并在 Leader 和 Follower 之间复制数据以保证高可用。
RocketMQ-on-DLedger Group 能自动容灾切换，并保证数据一致。
RocketMQ-on-DLedger Group 是可以水平扩展的，也即可以部署任意多个 RocketMQ-on-DLedger Group 同时对外提供服务。

1. 新集群部署

1.1 编写配置

每个 RocketMQ-on-DLedger Group 至少准备三台机器（本文假设为 3）。
编写 3 个配置文件，建议参考 conf/dledger 目录下的配置文件样例。
关键配置介绍：

name	含义	举例
enableDLegerCommitLog	是否启动 DLedger	true
dLegerGroup	DLedger Raft Group的名字，建议和 brokerName 保持一致	RaftNode00
dLegerPeers	DLedger Group 内各节点的端口信息，同一个 Group 内的各个节点配置必须要保证一致	n0-127.0.0.1:40911;n1-127.0.0.1:40912;n2-127.0.0.1:40913
dLegerSelfId	节点 id, 必须属于 dLegerPeers 中的一个；同 Group 内各个节点要唯一	n0
sendMessageThreadPoolNums	发送线程个数，建议配置成 Cpu 核数	16

这里贴出 conf/dledger/broker-n0.conf 的配置举例。

```
brokerClusterName = RaftCluster
brokerName=RaftNode00
listenPort=30911
namesrvAddr=127.0.0.1:9876
storePathRootDir=/tmp/rmqstore/node00
storePathCommitLog=/tmp/rmqstore/node00/commitlog
enableDLegerCommitLog=true
dLegerGroup=RaftNode00
dLegerPeers=n0-127.0.0.1:40911;n1-127.0.0.1:40912;n2-127.0.0.1:40913
## must be unique
dLegerSelfId=n0
sendMessageThreadPoolNums=16
```

1.2 启动 Broker

与老版本的启动方式一致。

```
nohup sh bin/mqbroker -c conf/dledger/xxx-n0.conf &
nohup sh bin/mqbroker -c conf/dledger/xxx-n1.conf &
nohup sh bin/mqbroker -c conf/dledger/xxx-n2.conf &
```

2. 旧集群升级

如果旧集群采用 Master 方式部署，则每个 Master 都需要转换成一个 RocketMQ-on-DLedger Group。
如果旧集群采用 Master-Slave 方式部署，则每个 Master-Slave 组都需要转换成一个 RocketMQ-on-DLedger Group。

2.1 杀掉旧的 Broker

可以通过 kill 命令来完成，也可以调用 `bin/mqshutdown broker` 。

2.2 检查旧的 Commitlog

RocketMQ-on-DLedger 组中的每个节点，可以兼容旧的 Commitlog ，但其 Raft 复制过程，只能针对新增加的消息。因此，为了避免出现异常，需要保证 旧的 Commitlog 是一致的。
如果旧的集群是采用 Master-Slave 方式部署，有可能在shutdown时，其数据并不是一致的，建议通过md5sum 的方式，检查最近的最少 2 个 Commmitlog 文件，如果发现不一致，则通过拷贝的方式进行对齐。

虽然 RocketMQ-on-DLedger Group 也可以以 2 节点方式部署，但其会丧失容灾切换能力（2n + 1 原则，至少需要3个节点才能容忍其中 1 个宕机）。
所以在对齐了 Master 和 Slave 的 Commitlog 之后，还需要准备第 3 台机器，并把旧的 Commitlog 从 Master 拷贝到第 3 台机器（记得同时拷贝一下 config 文件夹）。

在 3 台机器准备好了之后，旧 Commitlog 文件也保证一致之后，就可以开始走下一步修改配置了。

2.3 修改配置

参考新集群部署。

2.4 重新启动 Broker

参考新集群部署。

运维管理

1 集群搭建

1.1 单Master模式

这种方式风险较大，一旦Broker重启或者宕机时，会导致整个服务不可用。不建议线上环境使用,可以用于本地测试。

1）启动 NameServer

```
### 首先启动Name Server
$ nohup sh mqnamesrv &

### 验证Name Server 是否启动成功
$ tail -f ~/logs/rocketmqlogs/namesrv.log
The Name Server boot success...
```

2）启动 Broker

```
### 启动Broker
$ nohup sh bin/mqbroker -n localhost:9876 &

### 验证Name Server 是否启动成功，例如Broker的IP为：192.168.1.2，且名称为broker-a
$ tail -f ~/logs/rocketmqlogs/Broker.log
The broker[broker-a, 192.169.1.2:10911] boot success...
```

1.2 多Master模式

一个集群无Slave，全是Master，例如2个Master或者3个Master，这种模式的优缺点如下：

- 优点：配置简单，单个Master宕机或重启维护对应用无影响，在磁盘配置为RAID10时，即使机器宕机不可恢复情况下，由于RAID10磁盘非常可靠，消息也不会丢（异步刷盘丢失少量消息，同步刷盘一条不丢），性能最高；
- 缺点：单台机器宕机期间，这台机器上未被消费的消息在机器恢复之前不可订阅，消息实时性会受到影响。

1）启动NameServer

NameServer需要先于Broker启动，且如果在生产环境使用，为了保证高可用，建议一般规模的集群启动3个NameServer，各节点的启动命令相同，如下：

```
### 首先启动Name Server
$ nohup sh mqnamesrv &

### 验证Name Server 是否启动成功
$ tail -f ~/logs/rocketmqlogs/namesrv.log
The Name Server boot success...
```

2）启动Broker集群

```
### 在机器A，启动第一个Master，例如NameServer的IP为：192.168.1.1
$ nohup sh mqbroker -n 192.168.1.1:9876 -c $ROCKETMQ_HOME/conf/2m-noslave/broker-a.properties &

### 在机器B，启动第二个Master，例如NameServer的IP为：192.168.1.1
$ nohup sh mqbroker -n 192.168.1.1:9876 -c $ROCKETMQ_HOME/conf/2m-noslave/broker-b.properties &
```

```
...
```

如上启动命令是在单个NameServer情况下使用的。对于多个NameServer的集群，Broker启动命令中 `-n` 后面的地址列表用分号隔开即可，例如 `192.168.1.1:9876;192.161.2:9876` 。

1.3 多Master多Slave模式-异步复制

每个Master配置一个Slave，有多对Master-Slave，HA采用异步复制方式，主备有短暂消息延迟（毫秒级），这种模式的优缺点如下：

- 优点：即使磁盘损坏，消息丢失的非常少，且消息实时性不会受影响，同时Master宕机后，消费者仍然可以从Slave消费，而且此过程对应用透明，不需要人工干预，性能同多Master模式几乎一样；
- 缺点：Master宕机，磁盘损坏情况下会丢失少量消息。

1）启动NameServer

```
### 首先启动Name Server
$ nohup sh mqnamesrv &

### 验证Name Server 是否启动成功
$ tail -f ~/logs/rocketmqlogs/namesrv.log
The Name Server boot success...
```

2）启动Broker集群

```
### 在机器A，启动第一个Master，例如NameServer的IP为：192.168.1.1
$ nohup sh mqbroker -n 192.168.1.1:9876 -c $ROCKETMQ_HOME/conf/2m-2s-async/broker-a.properties &

### 在机器B，启动第二个Master，例如NameServer的IP为：192.168.1.1
$ nohup sh mqbroker -n 192.168.1.1:9876 -c $ROCKETMQ_HOME/conf/2m-2s-async/broker-b.properties &

### 在机器C，启动第一个Slave，例如NameServer的IP为：192.168.1.1
$ nohup sh mqbroker -n 192.168.1.1:9876 -c $ROCKETMQ_HOME/conf/2m-2s-async/broker-a-s.properties &

### 在机器D，启动第二个Slave，例如NameServer的IP为：192.168.1.1
$ nohup sh mqbroker -n 192.168.1.1:9876 -c $ROCKETMQ_HOME/conf/2m-2s-async/broker-b-s.properties &
```

1.4 多Master多Slave模式-同步双写

每个Master配置一个Slave，有多对Master-Slave，HA采用同步双写方式，即只有主备都写成功，才向应用返回成功，这种模式的优缺点如下：

- 优点：数据与服务都无单点故障，Master宕机情况下，消息无延迟，服务可用性与数据可用性都非常高；
- 缺点：性能比异步复制模式略低（大约低10%左右），发送单个消息的RT会略高，且目前版本在主节点宕机后，备机不能自动切换为主机。

1）启动NameServer

```
### 首先启动Name Server
$ nohup sh mqnamesrv &

### 验证Name Server 是否启动成功
$ tail -f ~/logs/rocketmqlogs/namesrv.log
The Name Server boot success...
```

2）启动Broker集群

```
### 在机器A，启动第一个Master，例如NameServer的IP为：192.168.1.1
$ nohup sh mqbroker -n 192.168.1.1:9876 -c $ROCKETMQ_HOME/conf/2m-2s-sync/broker-a.properties &

### 在机器B，启动第二个Master，例如NameServer的IP为：192.168.1.1
$ nohup sh mqbroker -n 192.168.1.1:9876 -c $ROCKETMQ_HOME/conf/2m-2s-sync/broker-b.properties &

### 在机器C，启动第一个Slave，例如NameServer的IP为：192.168.1.1
$ nohup sh mqbroker -n 192.168.1.1:9876 -c $ROCKETMQ_HOME/conf/2m-2s-sync/broker-a-s.properties &

### 在机器D，启动第二个Slave，例如NameServer的IP为：192.168.1.1
$ nohup sh mqbroker -n 192.168.1.1:9876 -c $ROCKETMQ_HOME/conf/2m-2s-sync/broker-b-s.properties &
```

以上Broker与Slave配对是通过指定相同的BrokerName参数来配对，Master的BrokerId必须是0，Slave的BrokerId必须是大于0的数。另外一个Master下面可以挂载多个Slave，同一Master下的多个Slave通过指定不同的BrokerId来区分。$ROCKETMQ_HOME指的RocketMQ安装目录，需要用户自己设置此环境变量。

2 mqadmin管理工具

> 注意：
>
> 1. 执行命令方法：`./mqadmin {command} {args}`
> 2. 几乎所有命令都需要配置-n表示NameServer地址，格式为ip:port
> 3. 几乎所有命令都可以通过-h获取帮助
> 4. 如果既有Broker地址（-b）配置项又有clusterName（-c）配置项，则优先以Broker地址执行命令，如果不配置Broker地址，则对集群中所有主机执行命令，只支持一个Broker地址。-b格式为ip:port，port默认是10911
> 5. 在tools下可以看到很多命令，但并不是所有命令都能使用，只有在MQAdminStartup中初始化的命令才能使用，你也可以修改这个类，增加或自定义命令
> 6. 由于版本更新问题，少部分命令可能未及时更新，遇到错误请直接阅读相关命令源码

2.1 Topic相关

名称	含义	命令选项	说明
updateTopic	创建更新Topic配置	-b	Broker 地址，表示 topic 所在 Broker，只支持单台 Broker，地址为ip:port
		-c	cluster 名称，表示 topic 所在集群（集群可通过 clusterList 查询）
		-h-	打印帮助
		-n	NameServer服务地址，格式 ip:port
		-p	指定新topic的读写权限(W=2|R=4|WR=6)

		-r	可读队列数（默认为 8）
		-w	可写队列数（默认为 8）
		-t	topic 名称（名称只能使用字符 ^[a-zA-Z0-9_-]+$ ）
deleteTopic	删除Topic	-c	cluster 名称，表示删除某集群下的某个 topic （集群可通过 clusterList 查询）
		-h	打印帮助
		-n	NameServer 服务地址，格式 ip:port
		-t	topic 名称（名称只能使用字符 ^[a-zA-Z0-9_-]+$ ）
topicList	查看 Topic 列表信息	-h	打印帮助
		-c	不配置-c只返回topic列表，增加-c返回clusterName, topic, consumerGroup信息，即topic的所属集群和订阅关系，没有参数
		-n	NameServer 服务地址，格式 ip:port
topicRoute	查看 Topic 路由信息	-t	topic 名称
		-h	打印帮助
		-n	NameServer 服务地址，格式 ip:port
topicStatus	查看 Topic 消息队列 offset	-t	topic 名称
		-h	打印帮助

		-n	NameServer 服务地址，格式 ip:port
topicClusterList	查看 Topic 所在集群列表	-t	topic 名称
		-h	打印帮助
		-n	NameServer 服务地址，格式 ip:port
updateTopicPerm	更新 Topic 读写权限	-t	topic 名称
		-h	打印帮助
		-n	NameServer 服务地址，格式 ip:port
		-b	Broker 地址，表示 topic 所在 Broker，只支持单台 Broker，地址为ip:port
		-p	指定新 topic 的读写权限(W=2|R=4|WR=6)
		-c	cluster 名称，表示 topic 所在集群（集群可通过 clusterList 查询），-b优先，如果没有-b，则对集群中所有Broker执行命令
updateOrderConf	从NameServer上创建、删除、获取特定命名空间的kv配置，目前还未启用	-h	打印帮助
		-n	NameServer 服务地址，格式 ip:port
		-t	topic，键
		-v	orderConf，值
		-m	method，可选get、put、delete
		-t	topic 名称
		-h	打印帮助
			NameServer 服务地址，

allocateMQ	以平均负载算法计算消费者列表负载消息队列的负载结果		格式 ip:port
		-i	ipList，用逗号分隔，计算这些ip去负载Topic的消息队列
statsAll	打印Topic订阅关系、TPS、积累量、24h读写总量等信息	-h	打印帮助
		-n	NameServer 服务地址，格式 ip:port
		-a	是否只打印活跃topic
		-t	指定topic

2.2 集群相关

名称	含义	命令选项	说明
clusterList	查看集群信息，集群、BrokerName、BrokerId、TPS等信息	-m	打印更多信息 (增加打印出如下信息 #InTotalYest, #OutTotalYest, #InTotalToday ,#OutTotalToday)
		-h	打印帮助
		-n	NameServer 服务地址，格式 ip:port
		-i	打印间隔，单位秒
clusterRT	发送消息检测集群各 Broker RT。消息发往 ${BrokerName} Topic。	-a	amount，每次探测的总数，RT = 总时间 / amount
		-s	消息大小，单位B
		-c	探测哪个集群
		-p	是否打印格式化日志，以|分割，默认不打印
		-h	打印帮助

	-m	所属机房，打印使用
	-i	发送间隔，单位秒
	-n	NameServer 服务地址，格式 ip:port

2.3 Broker相关

名称	含义	命令选项	说明
updateBrokerConfig	更新 Broker 配置文件，会修改Broker.conf	-b	Broker 地址，格式为 ip:port
		-c	cluster 名称
		-k	key 值
		-v	value 值
		-h	打印帮助
		-n	NameServer 服务地址，格式 ip:port
brokerStatus	查看 Broker 统计信息、运行状态（你想要的信息几乎都在里面）	-b	Broker 地址，地址为 ip:port
		-h	打印帮助
		-n	NameServer 服务地址，格式 ip:port
brokerConsumeStats	Broker中各个消费者的消费情况，按Message Queue维度返回Consume Offset，Broker Offset，Diff，TImestamp等信息	-b	Broker 地址，地址为 ip:port
		-t	请求超时时间
		-l	diff阈值，超过阈值才打印
		-o	是否为顺序topic，一般为false
		-h	打印帮助
		-n	NameServer 服务地址，格式 ip:port
getBrokerConfig	获取Broker配置	-b	Broker 地址，地址为 ip:port
		-n	NameServer 服务地址，格式 ip:port

wipeWritePerm	从NameServer上清除Broker写权限	-n	NameServer 服务地址，格式 ip:port
		-h	打印帮助
cleanExpiredCQ	清理Broker上过期的Consume Queue，如果手动减少对列数可能产生过期队列	-n	NameServer 服务地址，格式 ip:port
		-h	打印帮助
		-b	Broker 地址，地址为 ip:port
		-c	集群名称
cleanUnusedTopic	清理Broker上不使用的Topic，从内存中释放Topic的Consume Queue，如果手动删除Topic会产生不使用的Topic	-n	NameServer 服务地址，格式 ip:port
		-h	打印帮助
		-b	Broker 地址，地址为 ip:port
		-c	集群名称
sendMsgStatus	向Broker发消息，返回发送状态和RT	-n	NameServer 服务地址，格式 ip:port
		-h	打印帮助
		-b	BrokerName，注意不同于Broker地址
		-s	消息大小，单位B
		-c	发送次数

2.4 消息相关

名称	含义	命令选项	说明
queryMsgById	根据offsetMsgId查询msg，如果使用开源控制台，应使用offsetMsgId，此命令还有其他参数，具体作用请阅读QueryMsgByIdSubCommand。	-i	msgId
		-h	打印帮助
		-n	NameServer 服务地址，格式 ip:port

queryMsgByKey	根据消息 Key 查询消息	-k	msgKey
		-t	Topic 名称
		-h	打印帮助
		-n	NameServer 服务地址，格式 ip:port
queryMsgByOffset	根据 Offset 查询消息	-b	Broker 名称，（这里需要注意 填写的是 Broker 的名称，不是 Broker 的地址，Broker 名称可以在 clusterList 查到）
		-i	query 队列 id
		-o	offset 值
		-t	topic 名称
		-h	打印帮助
		-n	NameServer 服务地址，格式 ip:port
queryMsgByUniqueKey	根据msgId查询，msgId不同于offsetMsgId，区别详见常见运维问题。-g，-d配合使用，查到消息后尝试让特定的消费者消费消息并返回消费结果	-h	打印帮助
		-n	NameServer 服务地址，格式 ip:port
		-i	uniqe msg id
		-g	consumerGroup
		-d	clientId
		-t	topic名称
checkMsgSendRT	检测向topic发消息的RT，功能类似clusterRT	-h	打印帮助
		-n	NameServer 服务地址，格式 ip:port
		-t	topic名称
		-a	探测次数
		-s	消息大小
		-h	打印帮助

sendMessage	发送一条消息，可以根据配置发往特定Message Queue，或普通发送。	-h	打印帮助
		-n	NameServer 服务地址，格式 ip:port
		-t	topic名称
		-p	body，消息体
		-k	keys
		-c	tags
		-b	BrokerName
		-i	queueId
consumeMessage	消费消息。可以根据offset、开始&结束时间戳、消息队列消费消息，配置不同执行不同消费逻辑，详见ConsumeMessageCommand。	-h	打印帮助
		-n	NameServer 服务地址，格式 ip:port
		-t	topic名称
		-b	BrokerName
		-o	从offset开始消费
		-i	queueId
		-g	消费者分组
		-s	开始时间戳，格式详见-h
		-d	结束时间戳
		-c	消费多少条消息
printMsg	从Broker消费消息并打印，可选时间段	-h	打印帮助
		-n	NameServer 服务地址，格式 ip:port
		-t	topic名称
		-c	字符集，例如UTF-8
		-s	subExpress，过滤表达式
		-b	开始时间戳，格式参见-h
		-e	结束时间戳
		-d	是否打印消息体
		-h	打印帮助
		-n	NameServer 服务地址，格式 ip:port

printMsgByQueue	类似printMsg，但指定Message Queue	-i	queueId
		-a	BrokerName
		-c	字符集，例如UTF-8
		-s	subExpress，过滤表达式
		-b	开始时间戳，格式参见-h
		-e	结束时间戳
		-p	是否打印消息
		-d	是否打印消息体
		-f	是否统计tag数量并打印
resetOffsetByTime	按时间戳重置offset，Broker和consumer都会重置	-h	打印帮助
		-n	NameServer 服务地址，格式 ip:port
		-g	消费者分组
		-t	topic名称
		-s	重置为此时间戳对应的offset
		-f	是否强制重置，如果false，只支持回溯offset，如果true，不管时间戳对应offset与consumeOffset关系
		-c	是否重置c++客户端offset

2.5 消费者、消费组相关

名称	含义	命令选项	说明
consumerProgress	查看订阅组消费状态，可以查看具体的client IP的消息积累量	-g	消费者所属组名
		-s	是否打印client IP
		-h	打印帮助
		-n	NameServer 服务地址，格式 ip:port

consumerStatus	查看消费者状态，包括同一个分组中是否都是相同的订阅，分析Process Queue是否堆积，返回消费者jstack结果，内容较多，使用者参见ConsumerStatusSubCommand	-h	打印帮助
		-n	NameServer 服务地址，格式 ip:port
		-g	consumer group
		-i	clientId
		-s	是否执行jstack
getConsumerStatus	获取 Consumer 消费进度	-g	消费者所属组名
		-t	查询主题
		-i	Consumer 客户端 ip
		-n	NameServer 服务地址，格式 ip:port
		-h	打印帮助
updateSubGroup	更新或创建订阅关系	-n	NameServer 服务地址，格式 ip:port
		-h	打印帮助
		-b	Broker地址
		-c	集群名称
		-g	消费者分组名称
		-s	分组是否允许消费
		-m	是否从最小offset开始消费
		-d	是否是广播模式
		-q	重试队列数量
		-r	最大重试次数
		-i	当slaveReadEnable开启时有效，且还未达到从slave消费时建议从哪个BrokerId消费，可以配置备机id，主动从备机消费

		-w	如果Broker建议从slave消费，配置决定从哪个slave消费，配置BrokerId，例如1
		-a	当消费者数量变化时是否通知其他消费者负载均衡
deleteSubGroup	从Broker删除订阅关系	-n	NameServer 服务地址，格式 ip:port
		-h	打印帮助
		-b	Broker地址
		-c	集群名称
		-g	消费者分组名称
cloneGroupOffset	在目标群组中使用源群组的offset	-n	NameServer 服务地址，格式 ip:port
		-h	打印帮助
		-s	源消费者组
		-d	目标消费者组
		-t	topic名称
		-o	暂未使用

2.6 连接相关

名称	含义	命令选项	说明
consumerConnec tion	查询 Consumer 的网络连接	-g	消费者所属组名
		-n	NameServer 服务地址，格式 ip:port
		-h	打印帮助
producerConnec tion	查询 Producer 的网络连接	-g	生产者所属组名
		-t	主题名称
		-n	NameServer 服务地址，格式 ip:port

	-h	打印帮助

2.7 NameServer相关

名称	含义	命令选项	说明
updateKvConfig	更新NameServer的kv配置，目前还未使用	-s	命名空间
		-k	key
		-v	value
		-n	NameServer 服务地址，格式 ip:port
		-h	打印帮助
deleteKvConfig	删除NameServer的kv配置	-s	命名空间
		-k	key
		-n	NameServer 服务地址，格式 ip:port
		-h	打印帮助
getNamesrvConfig	获取NameServer配置	-n	NameServer 服务地址，格式 ip:port
		-h	打印帮助
updateNamesrvConfig	修改NameServer配置	-n	NameServer 服务地址，格式 ip:port
		-h	打印帮助
		-k	key
		-v	value

2.8 其他

名称	含义	命令选项	说明
startMonitoring	开启监控进程，监控消息误删、重试队列消息数等	-n	NameServer 服务地址，格式 ip:port
		-h	打印帮助

3 运维常见问题

3.1 RocketMQ的mqadmin命令报错问题

> 问题描述：有时候在部署完RocketMQ集群后，尝试执行“mqadmin”一些运维命令，会出现下面的异常信息：

```
org.apache.rocketmq.remoting.exception.RemotingConnectException: connect to <null> failed
```

解决方法：可以在部署RocketMQ集群的虚拟机上执行 `export NAMESRV_ADDR=ip:9876` （ip指的是集群中部署NameServer组件的机器ip地址）命令之后再使用“mqadmin”的相关命令进行查询，即可得到结果。

3.2 RocketMQ生产端和消费端版本不一致导致不能正常消费的问题

> 问题描述：同一个生产端发出消息，A消费端可消费，B消费端却无法消费，rocketMQ Console中出现：
>
> ```
> Not found the consumer group consume stats, because return offset table is empty, maybe the consumer not consume any message的异常消息。
> ```

解决方案：RocketMQ 的jar包：rocketmq-client等包应该保持生产端，消费端使用相同的version。

3.3 新增一个topic的消费组时，无法消费历史消息的问题

> 问题描述：当同一个topic的新增消费组启动时，消费的消息是当前的offset的消息，并未获取历史消息。

解决方案：rocketmq默认策略是从消息队列尾部，即跳过历史消息。如果想消费历史消息，则需要设置： `org.apache.rocketmq.client.consumer.DefaultMQPushConsumer#setConsumeFromWhere` 。常用的有以下三种配置：

- 默认配置,一个新的订阅组第一次启动从队列的最后位置开始消费，后续再启动接着上次消费的进度开始消费,即跳过历史消息；

```
consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_LAST_OFFSET);
```

- 一个新的订阅组第一次启动从队列的最前位置开始消费，后续再启动接着上次消费的进度开始消费,即消费Broker未过期的历史消息；

```
consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_FIRST_OFFSET);
```

- 一个新的订阅组第一次启动从指定时间点开始消费，后续再启动接着上次消费的进度开始消费，和consumer.setConsumeTimestamp()配合使用，默认是半个小时以前；

```
consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_TIMESTAMP);
```

3.4 如何开启从Slave读数据功能

在某些情况下，Consumer需要将消费位点重置到1-2天前，这时在内存有限的Master Broker上，CommitLog会承载比较重的IO压力，影响到该Broker的其它消息的读与写。可以开启 `slaveReadEnable=true` ，当Master Broker发现Consumer的消费位点与CommitLog的最新值的差值的容量超过该机器内存的百分比（ `accessMessageInMemoryMaxRatio=40%` ），会推荐Consumer从Slave Broker中去读取数据，降低Master Broker的IO。

3.5 性能调优问题

异步刷盘建议使用自旋锁，同步刷盘建议使用重入锁，调整Broker配置项 `useReentrantLockWhenPutMessage` ，默认为false；异步刷盘建议开启 `TransientStorePoolEnable` ；建议关闭transferMsgByHeap，提高拉消息效率；同步刷盘建议适当增大 `sendMessageThreadPoolNums` ，具体配置需要经过压测。

3.6 在RocketMQ中msgId和offsetMsgId的含义与区别

使用RocketMQ完成生产者客户端消息发送后，通常会看到如下日志打印信息：

```
SendResult [sendStatus=SEND_OK, msgId=0A42333A0DC818B4AAC246C290FD0000, offsetMsgId=0A42333A00002A9F00000000013
```

- msgId，对于客户端来说msgId是由客户端producer实例端生成的，具体来说，调用方法 `MessageClientIDSetter.createUniqIDBuffer()` 生成唯一的Id；
- offsetMsgId，offsetMsgId是由Broker服务端在写入消息时生成的（采用”IP地址+Port端口”与“CommitLog的物理偏移量地址”做了一个字符串拼接），其中offsetMsgId就是在RocketMQ控制台直接输入查询的那个messageId。

DefaultMQProducer

类简介

```
public class DefaultMQProducer extends ClientConfig implements MQProducer
```

> `DefaultMQProducer` 类是应用用来投递消息的入口，开箱即用，可通过无参构造方法快速创建一个生产者。主要负责消息的发送，支持同步/异步/oneway的发送方式，这些发送方式均支持批量发送。可以通过该类提供的getter/setter方法，调整发送者的参数。`DefaultMQProducer` 提供了多个send方法，每个send方法略有不同，在使用前务必详细了解其意图。下面给出一个生产者示例代码，点击查看更多示例代码。

```
public class Producer {
    public static void main(String[] args) throws MQClientException {
        // 创建指定分组名的生产者
        DefaultMQProducer producer = new DefaultMQProducer("ProducerGroupName");

        // 启动生产者
        producer.start();

        for (int i = 0; i < 128; i++)
            try {
                // 构建消息
                Message msg = new Message("TopicTest",
                        "TagA",
                        "OrderID188",
                        "Hello world".getBytes(RemotingHelper.DEFAULT_CHARSET));

                // 同步发送
                SendResult sendResult = producer.send(msg);

                // 打印发送结果
                System.out.printf("%s%n", sendResult);
            } catch (Exception e) {
                e.printStackTrace();
            }

        producer.shutdown();
    }
}
```

注意：该类是线程安全的。在配置并启动完成后可在多个线程间安全共享。

字段摘要

类型	字段名称	描述
DefaultMQProducerImpl	defaultMQProducerImpl	生产者的内部默认实现
String	producerGroup	生产者分组
String	createTopicKey	在发送消息时，自动创建服务器不存在的topic
int	defaultTopicQueueNums	创建topic时默认的队列数量
int	sendMsgTimeout	发送消息的超时时间
int	compressMsgBodyOverHowmuch	压缩消息体的阈值
int	retryTimesWhenSendFailed	同步模式下内部尝试发送消息的最大次数

int	retryTimesWhenSendAsyncFailed	异步模式下内部尝试发送消息的最大次数
boolean	retryAnotherBrokerWhenNotStoreOK	是否在内部发送失败时重试另一个broker
int	maxMessageSize	消息的最大长度
TraceDispatcher	traceDispatcher	消息追踪器。使用rcpHook来追踪消息

构造方法摘要

方法名称	方法描述
DefaultMQProducer()	由默认参数值创建一个生产者
DefaultMQProducer(final String producerGroup)	使用指定的分组名创建一个生产者
DefaultMQProducer(final String producerGroup, boolean enableMsgTrace)	使用指定的分组名创建一个生产者，并设置是否开启消息追踪
DefaultMQProducer(final String producerGroup, boolean enableMsgTrace, final String customizedTraceTopic)	使用指定的分组名创建一个生产者，并设置是否开启消息追踪及追踪topic的名称
DefaultMQProducer(RPCHook rpcHook)	使用指定的hook创建一个生产者
DefaultMQProducer(final String producerGroup, RPCHook rpcHook)	使用指定的分组名及自定义hook创建一个生产者
DefaultMQProducer(final String producerGroup, RPCHook rpcHook, boolean enableMsgTrace,final String customizedTraceTopic)	使用指定的分组名及自定义hook创建一个生产者，并设置是否开启消息追踪及追踪topic的名称

使用方法摘要

返回值	方法名称	方法描述
void	createTopic(String key, String newTopic, int queueNum)	在broker上创建指定的topic
void	createTopic(String key, String newTopic, int queueNum, int topicSysFlag)	在broker上创建指定的topic
long	earliestMsgStoreTime(MessageQueue mq)	查询最早的消息存储时间
List	fetchPublishMessageQueues(String topic)	获取topic的消息队列
long	maxOffset(MessageQueue mq)	查询给定消息队列的最大offset
long	minOffset(MessageQueue mq)	查询给定消息队列的最小offset
QueryResult	queryMessage(String topic, String key, int maxNum, long begin, long end)	按关键字查询消息
long	searchOffset(MessageQueue mq, long timestamp)	查找指定时间的消息队列的物理offset
SendResult	send(Collection msgs)	同步批量发送消息
SendResult	send(Collection msgs, long timeout)	同步批量发送消息
SendResult	send(Collection msgs, MessageQueue messageQueue)	向指定的消息队列同步批量发送消息
SendResult	send(Collection msgs, MessageQueue messageQueue, long timeout)	向指定的消息队列同步批量发送消息，并指定超时时间

SendResult	send(Message msg)	同步单条发送消息
SendResult	send(Message msg, long timeout)	同步发送单条消息，并指定超时时间
SendResult	send(Message msg, MessageQueue mq)	向指定的消息队列同步发送单条消息
SendResult	send(Message msg, MessageQueue mq, long timeout)	向指定的消息队列同步单条发送消息，并指定超时时间
void	send(Message msg, MessageQueue mq, SendCallback sendCallback)	向指定的消息队列异步单条发送消息，并指定回调方法
void	send(Message msg, MessageQueue mq, SendCallback sendCallback, long timeout)	向指定的消息队列异步单条发送消息，并指定回调方法和超时时间
SendResult	send(Message msg, MessageQueueSelector selector, Object arg)	向消息队列同步单条发送消息，并指定发送队列选择器
SendResult	send(Message msg, MessageQueueSelector selector, Object arg, long timeout)	向消息队列同步单条发送消息，并指定发送队列选择器与超时时间
void	send(Message msg, MessageQueueSelector selector, Object arg, SendCallback sendCallback)	向指定的消息队列异步单条发送消息
void	send(Message msg, MessageQueueSelector selector, Object arg, SendCallback sendCallback, long timeout)	向指定的消息队列异步单条发送消息，并指定超时时间
void	send(Message msg, SendCallback sendCallback)	异步发送消息
void	send(Message msg, SendCallback sendCallback, long timeout)	异步发送消息，并指定回调方法和超时时间
TransactionSendResult	sendMessageInTransaction(Message msg, LocalTransactionExecuter tranExecuter, final Object arg)	发送事务消息，并指定本地执行事务实例
TransactionSendResult	sendMessageInTransaction(Message msg, Object arg)	发送事务消息
void	sendOneway(Message msg)	单向发送消息，不等待broker响应
void	sendOneway(Message msg, MessageQueue mq)	单向发送消息到指定队列，不等待broker响应
void	sendOneway(Message msg, MessageQueueSelector selector, Object arg)	单向发送消息到队列选择器的选中的队列，不等待broker响应
void	shutdown()	关闭当前生产者实例并释放相关资源
void	start()	启动生产者
MessageExt	viewMessage(String offsetMsgId)	根据给定的msgId查询消息
MessageExt	public MessageExt viewMessage(String topic, String msgId)	根据给定的msgId查询消息，并指定topic

字段详细信息

- producerGroup

```
private String producerGroup
```

生产者的分组名称。相同的分组名称表明生产者实例在概念上归属于同一分组。这对事务消息十分重要，如果原始生产者在事务之后崩溃，那么broker可以联系同一生产者分组的不同生产者实例来提交或回滚事务。

默认值：DEFAULT_PRODUCER

注意： 由数字、字母、下划线、横杠（-）、竖线（|）或百分号组成；不能为空；长度不能超过255。

- defaultMQProducerImpl

```
protected final transient DefaultMQProducerImpl defaultMQProducerImpl
```

生产者的内部默认实现，在构造生产者时内部自动初始化，提供了大部分方法的内部实现。

- createTopicKey

```
private String createTopicKey = MixAll.AUTO_CREATE_TOPIC_KEY_TOPIC
```

在发送消息时，自动创建服务器不存在的topic，需要指定Key，该Key可用于配置发送消息所在topic的默认路由。

默认值：TBW102

建议：测试或者demo使用，生产环境下不建议打开自动创建配置。

- defaultTopicQueueNums

```
private volatile int defaultTopicQueueNums = 4
```

创建topic时默认的队列数量。

默认值：4

- sendMsgTimeout

```
private int sendMsgTimeout = 3000
```

发送消息时的超时时间。

默认值：3000，单位：毫秒

建议：不建议修改该值，该值应该与broker配置中的sendTimeout一致，发送超时，可临时修改该值，建议解决超时问题，提高broker集群的Tps。

- compressMsgBodyOverHowmuch

```
private int compressMsgBodyOverHowmuch = 1024 * 4
```

压缩消息体阈值。大于4K的消息体将默认进行压缩。

默认值：1024 * 4，单位：字节

建议：可通过DefaultMQProducerImpl.setZipCompressLevel方法设置压缩率（默认为5，可选范围[0,9]）；可通过DefaultMQProducerImpl.tryToCompressMessage方法测试出compressLevel与compressMsgBodyOverHowmuch最佳值。

- retryTimesWhenSendFailed

```
private int retryTimesWhenSendFailed = 2
```

同步模式下，在返回发送失败之前，内部尝试重新发送消息的最大次数。

默认值：2，即：默认情况下一条消息最多会被投递3次。

注意：在极端情况下，这可能会导致消息的重复。

- retryTimesWhenSendAsyncFailed

```
private int retryTimesWhenSendAsyncFailed = 2
```

异步模式下，在发送失败之前，内部尝试重新发送消息的最大次数。

默认值：2，即：默认情况下一条消息最多会被投递3次。

注意：在极端情况下，这可能会导致消息的重复。

- retryAnotherBrokerWhenNotStoreOK

  `private boolean retryAnotherBrokerWhenNotStoreOK = false`

  同步模式下，消息保存失败时是否重试其他broker。

  默认值：false

  注意：此配置关闭时，非投递时产生异常情况下，会忽略retryTimesWhenSendFailed配置。

- maxMessageSize

  `private int maxMessageSize = 1024 * 1024 * 4`

  消息的最大大小。当消息题的字节数超过maxMessageSize就发送失败。

  默认值：1024 *1024* 4，单位：字节

- traceDispatcher

  `private TraceDispatcher traceDispatcher = null`

  在开启消息追踪后，该类通过hook的方式把消息生产者，消息存储的broker和消费者消费消息的信息像链路一样记录下来。在构造生产者时根据构造入参enableMsgTrace来决定是否创建该对象。

构造方法详细信息

1. DefaultMQProducer

   `public DefaultMQProducer()`

   创建一个新的生产者。

2. DefaultMQProducer

   `DefaultMQProducer(final String producerGroup)`

   使用指定的分组名创建一个生产者。

   - 入参描述：

     参数名 | 类型 | 是否必须 | 缺省值 |描述 ---|---|---|---|--- producerGroup | String | 是 | DEFAULT_PRODUCER | 生产者的分组名称

3. DefaultMQProducer

   `DefaultMQProducer(final String producerGroup, boolean enableMsgTrace)`

   使用指定的分组名创建一个生产者，并设置是否开启消息追踪。

   - 入参描述：

     参数名 | 类型 | 是否必须 | 缺省值 |描述 ---|---|---|---|--- producerGroup | String | 是 | DEFAULT_PRODUCER | 生产者的分组名称 enableMsgTrace | boolean | 是 | false |是否开启消息追踪

4. DefaultMQProducer

   `DefaultMQProducer(final String producerGroup, boolean enableMsgTrace, final String customizedTraceTopic)`

   使用指定的分组名创建一个生产者，并设置是否开启消息追踪及追踪topic的名称。

- 入参描述：

  参数名 | 类型 | 是否必须 | 缺省值 |描述 ---|---|---|---|--- producerGroup | String | 是 | DEFAULT_PRODUCER | 生产者的分组名称 rpcHook | RPCHook | 否 | null |每个远程命令执行后会回调rpcHook enableMsgTrace | boolean | 是 | false |是否开启消息追踪 customizedTraceTopic | String | 否 | RMQ_SYS_TRACE_TOPIC | 消息跟踪topic的名称

5. DefaultMQProducer

   `DefaultMQProducer(RPCHook rpcHook)`

   使用指定的hook创建一个生产者。

   - 入参描述：

     参数名 | 类型 | 是否必须 | 缺省值 |描述 ---|---|---|---|--- rpcHook | RPCHook | 否 | null |每个远程命令执行后会回调rpcHook

6. DefaultMQProducer

   `DefaultMQProducer(final String producerGroup, RPCHook rpcHook)`

   使用指定的分组名及自定义hook创建一个生产者。

   - 入参描述：

     参数名 | 类型 | 是否必须 | 缺省值 |描述 ---|---|---|---|--- producerGroup | String | 是 | DEFAULT_PRODUCER | 生产者的分组名称 rpcHook | RPCHook | 否 | null |每个远程命令执行后会回调rpcHook

7. DefaultMQProducer

   `DefaultMQProducer(final String producerGroup, RPCHook rpcHook, boolean enableMsgTrace,final String customizedTraceTopic)`

   使用指定的分组名及自定义hook创建一个生产者，并设置是否开启消息追踪及追踪topic的名称。

   - 入参描述：

     参数名 | 类型 | 是否必须 | 缺省值 |描述 ---|---|---|---|--- producerGroup | String | 是 | DEFAULT_PRODUCER | 生产者的分组名称 rpcHook | RPCHook | 否 | null |每个远程命令执行后会回调rpcHook enableMsgTrace | boolean | 是 | false |是否开启消息追踪 customizedTraceTopic | String | 否 | RMQ_SYS_TRACE_TOPIC | 消息跟踪topic的名称

使用方法详细信息

1. createTopic

   `public void createTopic(String key, String newTopic, int queueNum)`

   在broker上创建一个topic。

   - 入参描述：

     参数名 | 类型 | 是否必须 | 默认值 |值范围 | 说明 ---|---|---|---|---|--- key | String | 是 | | | 访问密钥。 newTopic | String | 是 | | | 新建topic的名称。由数字、字母、下划线（_）、横杠（-）、竖线（|）或百分号（%）组成；长度小于255；不能为TBW102或空。 queueNum | int | 是 | 0 | (0, maxIntValue] | topic的队列数量。

   - 返回值描述：

     void

   - 异常描述：

     MQClientException - 生产者状态非Running；未找到broker等客户端异常。

2. createTopic

```
public void createTopic(String key, String newTopic, int queueNum, int topicSysFlag)
```

在broker上创建一个topic。

- 入参描述：

  参数名 | 类型 | 是否必须 | 默认值 |值范围 | 说明 ---|---|---|---|---|--- key | String | 是 | | | 访问密钥。 newTopic | String | 是 | | | 新建topic的名称。 queueNum | int | 是 | 0 | (0, maxIntValue] | topic的队列数量。 topicSysFlag | int | 是 | 0 | | 保留字段，暂未使用。

- 返回值描述：

  void

- 异常描述：

  MQClientException - 生产者状态非Running；未找到broker等客户端异常。

3. earliestMsgStoreTime

```
public long earliestMsgStoreTime(MessageQueue mq)
```

查询最早的消息存储时间。

- 入参描述：

  参数名 | 类型 | 是否必须 | 默认值 | 值范围 | 说明 ---|---|---|---|---|--- mq | MessageQueue | 是 | | | 要查询的消息队列

- 返回值描述：

  指定队列最早的消息存储时间。单位：毫秒。

- 异常描述：

  MQClientException - 生产者状态非Running；没有找到broker；broker返回失败；网络异常；线程中断等客户端异常。

4. fetchPublishMessageQueues

```
public List<MessageQueue> fetchPublishMessageQueues(String topic)
```

获取topic的消息队列。

- 入参描述：

  参数名 | 类型 | 是否必须 | 默认值 | 值范围 | 说明 ---|---|---|---|---|--- topic | String | 是 | | | topic名称

- 返回值描述：

  传入topic下的消息队列。

- 异常描述：

  MQClientException - 生产者状态非Running；没有找到broker；broker返回失败；网络异常；线程中断等客户端异常。

5. maxOffset

```
public long maxOffset(MessageQueue mq)
```

查询消息队列的最大物理偏移量。

- 入参描述：

参数名 | 类型 | 是否必须 | 默认值 | 值范围 | 说明 ---|---|---|---|---|--- mq | MessageQueue | 是 | | | 要查询的消息队列

- 返回值描述：

  给定消息队列的最大物理偏移量。

- 异常描述：

  MQClientException - 生产者状态非Running；没有找到broker；broker返回失败；网络异常；线程中断等客户端异常。

6. minOffset

```
public long minOffset(MessageQueue mq)
```

查询给定消息队列的最小物理偏移量。

- 入参描述：

  参数名 | 类型 | 是否必须 | 默认值 | 值范围 | 说明 ---|---|---|---|---|--- mq | MessageQueue | 是 | | | 要查询的消息队列

- 返回值描述：

  给定消息队列的最小物理偏移量。

- 异常描述：

  MQClientException - 生产者状态非Running；没有找到broker；broker返回失败；网络异常；线程中断等客户端异常。

7. queryMessage

```
public QueryResult queryMessage(String topic, String key, int maxNum, long begin, long end)
```

按关键字查询消息。

- 入参描述：

  参数名 | 类型 | 是否必须 | 默认值 | 值范围 | 说明 ---|---|---|---|---|--- topic | String | 是 | | | topic名称 key | String | 否 | null | | 查找的关键字 maxNum | int | 是 | | | 返回消息的最大数量 begin | long | 是 | | | 开始时间戳，单位：毫秒 end | long | 是 | | | 结束时间戳，单位：毫秒

- 返回值描述：

  查询到的消息集合。

- 异常描述：

  MQClientException - 生产者状态非Running；没有找到broker；broker返回失败；网络异常等客户端异常客户端异常。
  InterruptedException - 线程中断。

8. searchOffset

```
public long searchOffset(MessageQueue mq, long timestamp)
```

查找指定时间的消息队列的物理偏移量。

- 入参描述：

  参数名 | 类型 | 是否必须 | 默认值 | 值范围 | 说明 ---|---|---|---|---|--- mq | MessageQueue | 是 | | | 要查询的消息队列。 timestamp | long | 是 | | | 指定要查询时间的时间戳。单位：毫秒。

- 返回值描述：

  指定时间的消息队列的物理偏移量。

- 异常描述：

  MQClientException - 生产者状态非Running；没有找到broker；broker返回失败；网络异常；线程中断等客户端异常。

9. send

```
public SendResult send(Collection<Message> msgs)
```

同步批量发送消息。在返回发送失败之前，内部尝试重新发送消息的最大次数（参见*retryTimesWhenSendFailed*属性）。未明确指定发送队列，默认采取轮询策略发送。

- 入参描述：

  参数名 | 类型 | 是否必须 | 默认值 | 值范围 | 说明 ---|---|---|---|---|--- msgs | Collection | 是 | | | 待发送的消息集合。集合内的消息必须属同一个topic。

- 返回值描述：

  批量消息的发送结果，包含msgId，发送状态等信息。

- 异常描述：

  MQClientException - broker不存在或未找到；namesrv地址为空；未找到topic的路由信息等客户端异常。
  RemotingException - 网络异常。
  MQBrokerException - broker发生错误。
  InterruptedException - 发送线程中断。
  RemotingTooMuchRequestException - 发送超时。

10. send

```
public SendResult send(Collection<Message> msgs, long timeout)
```

同步批量发送消息，如果在指定的超时时间内未完成消息投递，会抛出*RemotingTooMuchRequestException*。 在返回发送失败之前，内部尝试重新发送消息的最大次数（参见*retryTimesWhenSendFailed*属性）。未明确指定发送队列，默认采取轮询策略发送。

- 入参描述：

  参数名 | 类型 | 是否必须 | 默认值 | 值范围 | 说明 ---|---|---|---|---|--- msgs | Collection | 是 | | | 待发送的消息集合。集合内的消息必须属同一个topic。 timeout | long | 是 | 参见*sendMsgTimeout*属性 | | 发送超时时间，单位：毫秒。

- 返回值描述：

  批量消息的发送结果，包含msgId，发送状态等信息。

- 异常描述：

  MQClientException - broker不存在或未找到；namesrv地址为空；未找到topic的路由信息等客户端异常。
  RemotingException - 网络异常。
  MQBrokerException - broker发生错误。
  InterruptedException - 发送线程中断。
  RemotingTooMuchRequestException - 发送超时。

11. send

```
public SendResult send(Collection<Message> msgs, MessageQueue messageQueue)
```

向给定队列同步批量发送消息。

注意：指定队列意味着所有消息均为同一个topic。

- 入参描述：

  参数名 | 类型 | 是否必须 | 默认值 | 值范围 | 说明 ---|---|---|---|---|--- msgs | Collection | 是 | | | 待发送的消息集合。集合内的消息必须属同一个topic。 messageQueue | MessageQueue | 是 | | | 待投递的消息队列。指定队列意味着待投递消息均为同一个topic。

- 返回值描述：

  批量消息的发送结果，包含msgId，发送状态等信息。

- 异常描述：

  MQClientException - broker不存在或未找到；namesrv地址为空；未找到topic的路由信息等客户端异常。
  RemotingException - 网络异常。
  MQBrokerException - broker发生错误。
  InterruptedException - 发送线程中断。
  RemotingTooMuchRequestException - 发送超时。

12. send

```
public SendResult send(Collection<Message> msgs, MessageQueue messageQueue, long timeout)
```

向给定队列同步批量发送消息，如果在指定的超时时间内未完成消息投递，会抛出*RemotingTooMuchRequestException*。

注意：指定队列意味着所有消息均为同一个topic。

- 入参描述：

  参数名 | 类型 | 是否必须 | 默认值 | 值范围 | 说明 ---|---|---|---|---|--- msgs | Collection | 是 | | | 待发送的消息集合。集合内的消息必须属同一个topic。 timeout | long | 是 | 参见*sendMsgTimeout*属性 | | 发送超时时间，单位：毫秒。 messageQueue | MessageQueue | 是 | | | 待投递的消息队列。指定队列意味着待投递消息均为同一个topic。

- 返回值描述：

  批量消息的发送结果，包含msgId，发送状态等信息。

- 异常描述：

  MQClientException - broker不存在或未找到；namesrv地址为空；未找到topic的路由信息等客户端异常。
  RemotingException - 网络异常。
  MQBrokerException - broker发生错误。
  InterruptedException - 发送线程中断。
  RemotingTooMuchRequestException - 发送超时。

13. send

```
public SendResult send(Message msg)
```

以同步模式发送消息，仅当发送过程完全完成时，此方法才会返回。 在返回发送失败之前，内部尝试重新发送消息的最大次数（参见*retryTimesWhenSendFailed*属性）。未明确指定发送队列，默认采取轮询策略发送。

- 入参描述：

  参数名 | 类型 | 是否必须 | 默认值 | 值范围 | 说明 ---|---|---|---|---|--- msg | Message | 是 | | | 待发送的消息。

- 返回值描述：

消息的发送结果，包含msgId，发送状态等信息。

- 异常描述：

  MQClientException - broker不存在或未找到；namesrv地址为空；未找到topic的路由信息等客户端异常。
  RemotingException - 网络异常。
  MQBrokerException - broker发生错误。
  InterruptedException - 发送线程中断。
  RemotingTooMuchRequestException - 发送超时。

14. send

```
public SendResult send(Message msg, long timeout)
```

以同步模式发送消息，如果在指定的超时时间内未完成消息投递，会抛出*RemotingTooMuchRequestException*。仅当发送过程完全完成时，此方法才会返回。 在返回发送失败之前，内部尝试重新发送消息的最大次数（参见*retryTimesWhenSendFailed*属性）。未明确指定发送队列，默认采取轮询策略发送。

- 入参描述：

  参数名 | 类型 | 是否必须 | 默认值 | 值范围 | 说明 ---|---|---|---|---|--- msg | Message | 是 | | | 待发送的消息。 timeout | long | 是 | 参见*sendMsgTimeout*属性 | | 发送超时时间，单位：毫秒。

- 返回值描述：

  消息的发送结果，包含msgId，发送状态等信息。

- 异常描述：

  MQClientException - broker不存在或未找到；namesrv地址为空；未找到topic的路由信息等客户端异常。
  RemotingException - 网络异常。
  MQBrokerException - broker发生错误。
  InterruptedException - 发送线程中断。
  RemotingTooMuchRequestException - 发送超时。

15. send

```
public SendResult send(Message msg, MessageQueue mq)
```

向指定的消息队列同步发送单条消息。仅当发送过程完全完成时，此方法才会返回。

- 入参描述：

  参数名 | 类型 | 是否必须 | 默认值 | 值范围 | 说明 ---|---|---|---|---|--- msg | Message | 是 | | | 待发送的消息。 mq | MessageQueue | 是 | | | 待投递的消息队列。

- 返回值描述：

  消息的发送结果，包含msgId，发送状态等信息。

- 异常描述：

  MQClientException - broker不存在或未找到；namesrv地址为空；未找到topic的路由信息等客户端异常。
  RemotingException - 网络异常。
  MQBrokerException - broker发生错误。
  InterruptedException - 发送线程中断。
  RemotingTooMuchRequestException - 发送超时。

16. send

```
public SendResult send(Message msg, MessageQueue mq, long timeout)
```

向指定的消息队列同步发送单条消息，如果在指定的超时时间内未完成消息投递，会抛出*RemotingTooMuchRequestException*。仅当发送过程完全完成时，此方法才会返回。

- 入参描述：

  参数名 | 类型 | 是否必须 | 默认值 | 值范围 | 说明 ---|---|---|---|---|--- msg | Message | 是 | | | 待发送的消息。 timeout | long | 是 | 参见*sendMsgTimeout*属性 | | 发送超时时间，单位：毫秒。 mq | MessageQueue | 是 | | | 待投递的消息队列。指定队列意味着待投递消息均为同一个topic。

- 返回值描述：

  消息的发送结果，包含msgId，发送状态等信息。

- 异常描述：

  MQClientException - broker不存在或未找到；namesrv地址为空；未找到topic的路由信息等客户端异常。
  RemotingException - 网络异常。
  MQBrokerException - broker发生错误。
  InterruptedException - 发送线程中断。
  RemotingTooMuchRequestException - 发送超时。

17. send

    ```
    public void send(Message msg, MessageQueue mq, SendCallback sendCallback)
    ```

    向指定的消息队列异步发送单条消息，异步发送调用后直接返回，并在在发送成功或者异常时回调`sendCallback`，所以异步发送时`sendCallback`参数不能为null，否则在回调时会抛出`NullPointerException`。异步发送时，在成功发送前，其内部会尝试重新发送消息的最大次数（参见*retryTimesWhenSendAsyncFailed*属性）。

    - 入参描述：

      参数名 | 类型 | 是否必须 | 默认值 | 值范围 | 说明 ---|---|---|---|---|--- msg | Message | 是 | | | 待发送的消息。 mq | MessageQueue | 是 | | | 待投递的消息队列。指定队列意味着待投递消息均为同一个topic。 sendCallback | SendCallback | 是 | | | 回调接口的实现。

    - 返回值描述：

      void

    - 异常描述：

      MQClientException - broker不存在或未找到；namesrv地址为空；未找到topic的路由信息等客户端异常。
      RemotingException - 网络异常。
      InterruptedException - 发送线程中断。

18. send

    ```
    public void send(Message msg, MessageQueue mq, SendCallback sendCallback, long timeout)
    ```

    向指定的消息队列异步发送单条消息，异步发送调用后直接返回，并在在发送成功或者异常时回调`sendCallback`，所以异步发送时`sendCallback`参数不能为null，否则在回调时会抛出`NullPointerException`。若在指定时间内消息未发送成功，回调方法会收到*RemotingTooMuchRequestException*异常。 异步发送时，在成功发送前，其内部会尝试重新发送消息的最大次数（参见*retryTimesWhenSendAsyncFailed*属性）。

    - 入参描述：

      参数名 | 类型 | 是否必须 | 默认值 | 值范围 | 说明 ---|---|---|---|---|--- msg | Message | 是 | | | 待发送的消息。 mq | MessageQueue | 是 | | | 待投递的消息队列。 sendCallback | SendCallback | 是 | | | 回调接口的实现。 timeout | long | 是 | 参见*sendMsgTimeout*属性 | | 发送超时时间，单位：毫秒。

    - 返回值描述： void

- 异常描述：

  MQClientException - broker不存在或未找到；namesrv地址为空；未找到topic的路由信息等客户端异常。
  RemotingException - 网络异常。
  InterruptedException - 发送线程中断。

19. send

    ```
    public SendResult send(Message msg, MessageQueueSelector selector, Object arg)
    ```

    向通过 `MessageQueueSelector` 计算出的队列同步发送消息。

    可以通过自实现 `MessageQueueSelector` 接口，将某一类消息发送至固定的队列。比如：将同一个订单的状态变更消息投递至固定的队列。

    注意：此消息发送失败内部不会重试。

    - 入参描述：

      参数名 | 类型 | 是否必须 | 默认值 | 值范围 | 说明 ---|---|---|---|---|--- msg | Message | 是 | | | 待发送的消息。 selector | MessageQueueSelector | 是 | | | 队列选择器。 arg | Object | 否 | | | 供队列选择器使用的参数对象。

    - 返回值描述：

      消息的发送结果，包含msgId，发送状态等信息。

    - 异常描述：

      MQClientException - broker不存在或未找到；namesrv地址为空；未找到topic的路由信息等客户端异常。
      RemotingException - 网络异常。
      MQBrokerException - broker发生错误。
      InterruptedException - 发送线程中断。
      RemotingTooMuchRequestException - 发送超时。

20. send

    ```
    public SendResult send(Message msg, MessageQueueSelector selector, Object arg, long timeout)
    ```

    向通过 `MessageQueueSelector` 计算出的队列同步发送消息，并指定发送超时时间。

    可以通过自实现 `MessageQueueSelector` 接口，将某一类消息发送至固定的队列。比如：将同一个订单的状态变更消息投递至固定的队列。

    注意：此消息发送失败内部不会重试。

    - 入参描述：

      参数名 | 类型 | 是否必须 | 默认值 | 值范围 | 说明 ---|---|---|---|---|--- msg | Message | 是 | | | 待发送的消息。 selector | MessageQueueSelector | 是 | | | 队列选择器。 arg | Object | 否 | | | 供队列选择器使用的参数对象。 timeout | long | 是 | 参见*sendMsgTimeout*属性 | | 发送超时时间，单位：毫秒。

    - 返回值描述：

      消息的发送结果，包含msgId，发送状态等信息。

    - 异常描述： MQClientException - broker不存在或未找到；namesrv地址为空；未找到topic的路由信息等客户端异常。
      RemotingException - 网络异常。
      MQBrokerException - broker发生错误。
      InterruptedException - 发送线程中断。
      RemotingTooMuchRequestException - 发送超时。

21. send

`public void send(Message msg, MessageQueueSelector selector, Object arg, SendCallback sendCallback)`

向通过 `MessageQueueSelector` 计算出的队列异步发送单条消息，异步发送调用后直接返回，并在在发送成功或者异常时回调 `sendCallback` ，所以异步发送时 `sendCallback` 参数不能为null，否则在回调时会抛出 `NullPointerException` 。 异步发送时，在成功发送前，其内部会尝试重新发送消息的最大次数（参见*retryTimesWhenSendAsyncFailed*属性）。

可以通过自实现 `MessageQueueSelector` 接口，将某一类消息发送至固定的队列。比如：将同一个订单的状态变更消息投递至固定的队列。

- 入参描述：

  参数名 | 类型 | 是否必须 | 默认值 | 值范围 | 说明 ---|---|---|---|---|--- msg | Message | 是 | | | 待发送的消息。 selector | MessageQueueSelector | 是 | | | 队列选择器。 arg | Object | 否 | | | 供队列选择器使用的参数对象。 sendCallback | SendCallback | 是 | | | 回调接口的实现。

- 返回值描述：

  void

- 异常描述：

  MQClientException - broker不存在或未找到；namesrv地址为空；未找到topic的路由信息等客户端异常。
  RemotingException - 网络异常。
  InterruptedException - 发送线程中断。

22. send

`public void send(Message msg, MessageQueueSelector selector, Object arg, SendCallback sendCallback, long timeout)`

向通过 `MessageQueueSelector` 计算出的队列异步发送单条消息，异步发送调用后直接返回，并在在发送成功或者异常时回调 `sendCallback` ，所以异步发送时 `sendCallback` 参数不能为null，否则在回调时会抛出 `NullPointerException` 。 异步发送时，在成功发送前，其内部会尝试重新发送消息的最大次数（参见*retryTimesWhenSendAsyncFailed*属性）。

可以通过自实现 `MessageQueueSelector` 接口，将某一类消息发送至固定的队列。比如：将同一个订单的状态变更消息投递至固定的队列。

- 入参描述：

  参数名 | 类型 | 是否必须 | 默认值 | 值范围 | 说明 ---|---|---|---|---|--- msg | Message | 是 | | | 待发送的消息。 selector | MessageQueueSelector | 是 | | | 队列选择器。 arg | Object | 否 | | | 供队列选择器使用的参数对象。 sendCallback | SendCallback | 是 | | | 回调接口的实现。 timeout | long | 是 | 参见*sendMsgTimeout*属性 | | 发送超时时间，单位：毫秒。

- 返回值描述：

  void

- 异常描述：

  MQClientException - broker不存在或未找到；namesrv地址为空；未找到topic的路由信息等客户端异常。
  RemotingException - 网络异常。
  InterruptedException - 发送线程中断。

23. send

`public void send(Message msg, SendCallback sendCallback)`

异步发送单条消息，异步发送调用后直接返回，并在在发送成功或者异常时回调 `sendCallback` ，所以异步发送时 `sendCallback` 参数不能为null，否则在回调时会抛出 `NullPointerException` 。 异步发送时，在成功发送前，其内部会尝试重新发送消息的最大次数（参见*retryTimesWhenSendAsyncFailed*属性）。

- 入参描述：

  参数名 | 类型 | 是否必须 | 默认值 | 值范围 | 说明 ---|---|---|---|---|--- msg | Message | 是 | | | 待发送的消息。 sendCallback | SendCallback | 是 | | | 回调接口的实现。

- 返回值描述：

  void

- 异常描述：

  MQClientException - broker不存在或未找到；namesrv地址为空；未找到topic的路由信息等客户端异常。
  RemotingException - 网络异常。
  InterruptedException - 发送线程中断。

24. send

```
public void send(Message msg, SendCallback sendCallback, long timeout)
```

异步发送单条消息，异步发送调用后直接返回，并在在发送成功或者异常时回调 `sendCallback` ，所以异步发送时 `sendCallback` 参数不能为null，否则在回调时会抛出 `NullPointerException` 。 异步发送时，在成功发送前，其内部会尝试重新发送消息的最大次数（参见*retryTimesWhenSendAsyncFailed*属性）。

- 入参描述：

  参数名 | 类型 | 是否必须 | 默认值 | 值范围 | 说明 ---|---|---|---|---|--- msg | Message | 是 | | | 待发送的消息。 sendCallback | SendCallback | 是 | | | 回调接口的实现。 timeout | long | 是 | 参见*sendMsgTimeout*属性 | | 发送超时时间，单位：毫秒。

- 返回值描述：

  void

- 异常描述：

  MQClientException - broker不存在或未找到；namesrv地址为空；未找到topic的路由信息等客户端异常。
  RemotingException - 网络异常。
  InterruptedException - 发送线程中断。

25. sendMessageInTransaction

```
public TransactionSendResult sendMessageInTransaction(Message msg, LocalTransactionExecuter tranExecuter,
final Object arg)
```

发送事务消息。该类不做默认实现，抛出 `RuntimeException` 异常。参见： `TransactionMQProducer` 类。

- 入参描述：

  参数名 | 类型 | 是否必须 | 默认值 | 值范围 | 说明 ---|---|---|---|---|--- msg | Message | 是 | | | 待投递的事务消息 tranExecuter | `LocalTransactionExecuter` | 是 | | | 本地事务执行器。该类已过期，将在5.0.0版本中移除。请勿使用该方法。 arg | Object | 是 | | | 供本地事务执行程序使用的参数对象

- 返回值描述：

  事务结果，参见： `LocalTransactionState` 类。

- 异常描述：

  RuntimeException - 永远抛出该异常。

26. sendMessageInTransaction

    `public TransactionSendResult sendMessageInTransaction(Message msg, final Object arg)`

    发送事务消息。该类不做默认实现，抛出 `RuntimeException` 异常。参见： `TransactionMQProducer` 类。

    - 入参描述：

      参数名 | 类型 | 是否必须 | 默认值 | 值范围 | 说明 ---|---|---|---|---|--- msg | Message | 是 | | | 待投递的事务消息 arg | Object | 是 | | | 供本地事务执行程序使用的参数对象

    - 返回值描述：

      事务结果，参见： `LocalTransactionState` 类。

    - 异常描述：

      RuntimeException - 永远抛出该异常。

27. sendOneway

    `public void sendOneway(Message msg)`

    以oneway形式发送消息，broker不会响应任何执行结果，和UDP类似。它具有最大的吞吐量但消息可能会丢失。

    可在消息量大，追求高吞吐量并允许消息丢失的情况下使用该方式。

    - 入参描述：

      参数名 | 类型 | 是否必须 | 默认值 | 值范围 | 说明 ---|---|---|---|---|--- msg | Message | 是 | | | 待投递的消息

    - 返回值描述：

      void

    - 异常描述：

      MQClientException - broker不存在或未找到；namesrv地址为空；未找到topic的路由信息等客户端异常。
      RemotingException - 网络异常。
      InterruptedException - 发送线程中断。

28. sendOneway

    `public void sendOneway(Message msg, MessageQueue mq)`

    向指定队列以oneway形式发送消息，broker不会响应任何执行结果，和UDP类似。它具有最大的吞吐量但消息可能会丢失。

    可在消息量大，追求高吞吐量并允许消息丢失的情况下使用该方式。

    - 入参描述：

      参数名 | 类型 | 是否必须 | 默认值 | 值范围 | 说明 ---|---|---|---|---|--- msg | Message | 是 | | | 待投递的消息 mq | MessageQueue | 是 | | | 待投递的消息队列

    - 返回值描述： void

    - 异常描述： MQClientException - broker不存在或未找到；namesrv地址为空；未找到topic的路由信息等客户端异常。
      RemotingException - 网络异常。
      InterruptedException - 发送线程中断。

29. sendOneway

    `public void sendOneway(Message msg, MessageQueueSelector selector, Object arg)`

向通过 `MessageQueueSelector` 计算出的队列以oneway形式发送消息，broker不会响应任何执行结果，和UDP类似。它具有最大的吞吐量但消息可能会丢失。

可在消息量大，追求高吞吐量并允许消息丢失的情况下使用该方式。

- 入参描述：

  参数名 | 类型 | 是否必须 | 默认值 | 值范围 | 说明 ---|---|---|---|---|--- msg | Message | 是 | | | 待发送的消息。 selector | MessageQueueSelector | 是 | | | 队列选择器。 arg | Object | 否 | | | 供队列选择器使用的参数对象。

- 返回值描述：

  void

- 异常描述：

  MQClientException - broker不存在或未找到；namesrv地址为空；未找到topic的路由信息等客户端异常。
  RemotingException - 网络异常。
  InterruptedException - 发送线程中断。

30. shutdown

    ```
    public void shutdown()
    ```

    关闭当前生产者实例并释放相关资源。

    - 入参描述：

      无。

    - 返回值描述：

      void

    - 异常描述：

31. start

    ```
    public void start()
    ```

    启动生产者实例。在发送或查询消息之前必须调用此方法。它执行了许多内部初始化，比如：检查配置、与namesrv建立连接、启动一系列心跳等定时任务等。

    - 入参描述：

      无。

    - 返回值描述：

      void

    - 异常描述：

      MQClientException - 初始化过程中出现失败。

32. viewMessage

    ```
    public MessageExt viewMessage(String offsetMsgId)
    ```

    根据给定的msgId查询消息。

    - 入参描述：

      参数名 | 类型 | 是否必须 | 默认值 | 值范围 | 说明 ---|---|---|---|---|--- offsetMsgId | String | 是 | | | offsetMsgId

    - 返回值描述：

返回 `MessageExt`，包含：topic名称，消息题，消息ID，消费次数，生产者host等信息。

- 异常描述：

  RemotingException - 网络层发生错误。
  MQBrokerException - broker发生错误。
  InterruptedException - 线程被中断。
  MQClientException - 生产者状态非Running；msgId非法等。

33. viewMessage

```
public MessageExt viewMessage(String topic, String msgId)
```

根据给定的msgId查询消息，并指定topic。

- 入参描述：

  参数名 | 类型 | 是否必须 | 默认值 | 值范围 | 说明 ---|---|---|---|---|--- msgId | String | 是 | | | msgId topic | String | 是 | | | topic名称

- 返回值描述：

  返回 `MessageExt`，包含：topic名称，消息题，消息ID，消费次数，生产者host等信息。

- 异常描述：

  RemotingException - 网络层发生错误。
  MQBrokerException - broker发生错误。
  InterruptedException - 线程被中断。
  MQClientException - 生产者状态非Running；msgId非法等。