RocketMQ笔记（基于丁威老师ppt）

other

顺序性

mq的顺序消息是针对消费的，而不是针对生产者

mq场景：

1. 削峰填谷

   比如双十一的快递订单量

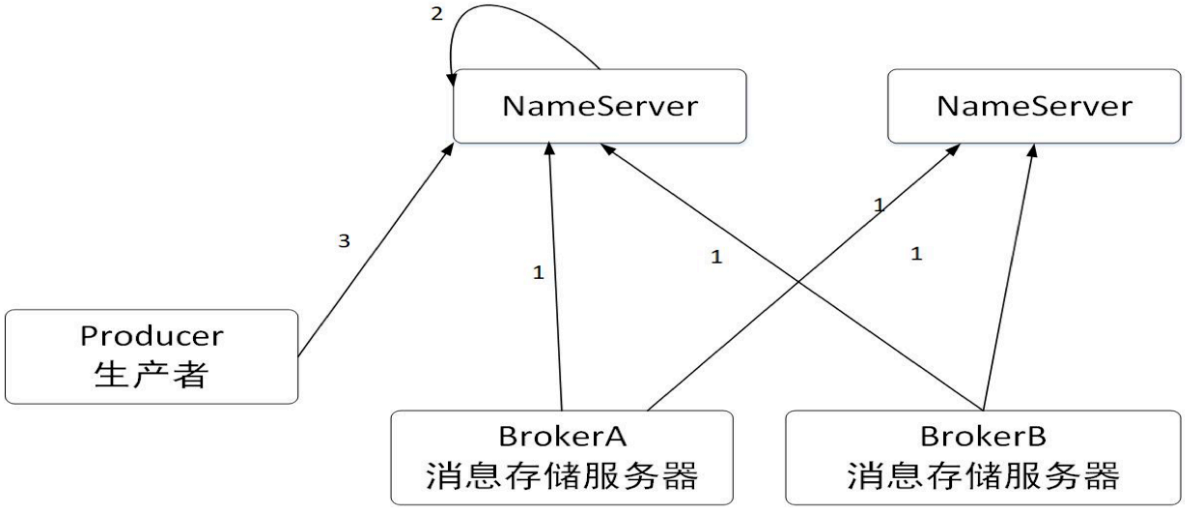
2. 解耦（最终一致性）

   比如，登录以后，要分配会员体系，新人礼物等。这样，登录系统与会员系统，就可以完全解耦

3. 异步

NameServer路由注册、发现、剔除设计模式

RocketMQ Topic 路由的注册、发现采用的就是拉取模式。



1. Broker 每30s向 NameServer 发送心跳包，心跳包中包含主题的路由信息（主题的读写队列数、操作权限等），NameServer 会通过 HashMap 更新 Topic 的路由信息，并记录最后一次收到 Broker 的时间戳。
2. NameServer 以每10s的频率清除已宕机的 Broker，NameServr 认为 Broker 宕机的依据是如果当前系统时间戳减去最后一次收到 Broker 心跳包的时间戳大于120s。
3. 消息生产者以每30s的频率去拉取主题的路由信息，即消息生产者并不会立即感知 Broker 服务器的新增与删除。

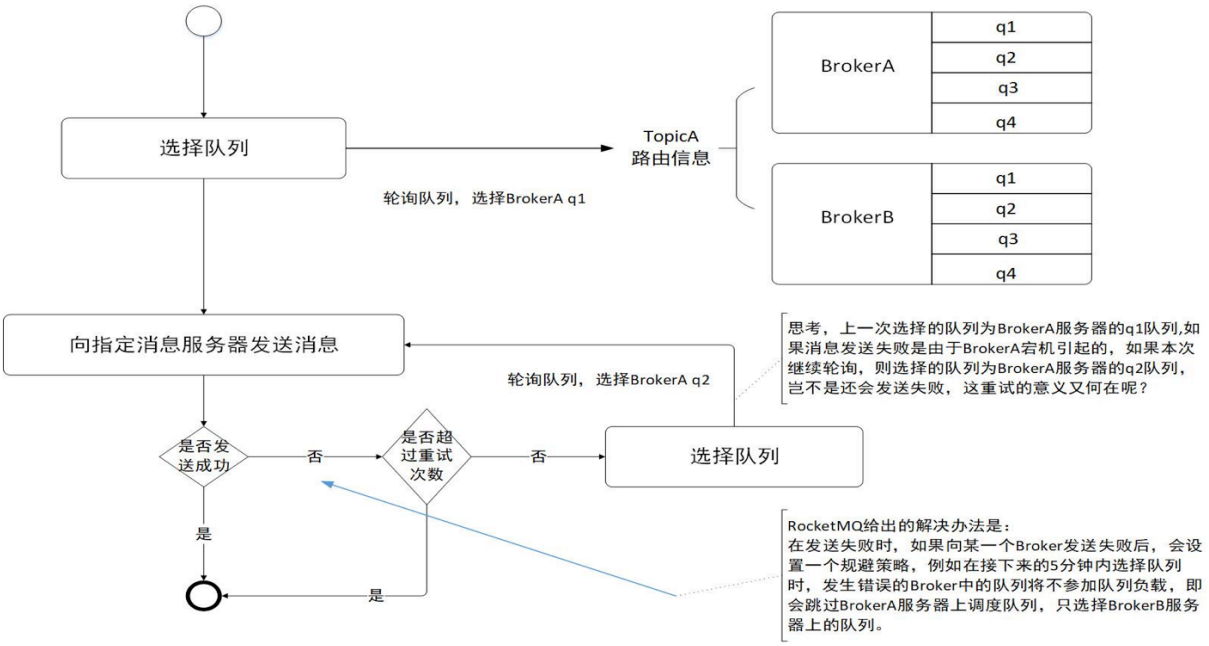
缺点

1. Topic 路由中心(Nameserver) topic 是基于最终一致性，极端情况下会出现数据不一致。
2. 客户端无法实时感知路由信息的变化，例如某台Broker 自身进程未关闭，但停止向 Nameserver 发送心跳包，但生产者无法立即感知该 Broker 服务器的异常，会对消息发送造成一定的可用性?

针对缺点的解决方案

不再NameServer的设计中解决，而是在消息发送时解决（快速失败，选择其他生效的broker）

消息发送高可用设计



消息发送流程主要包括如下三个方面：

1. Topic路由寻址
2. 选择消息队列
3. 消息发送、重试、Broker规避

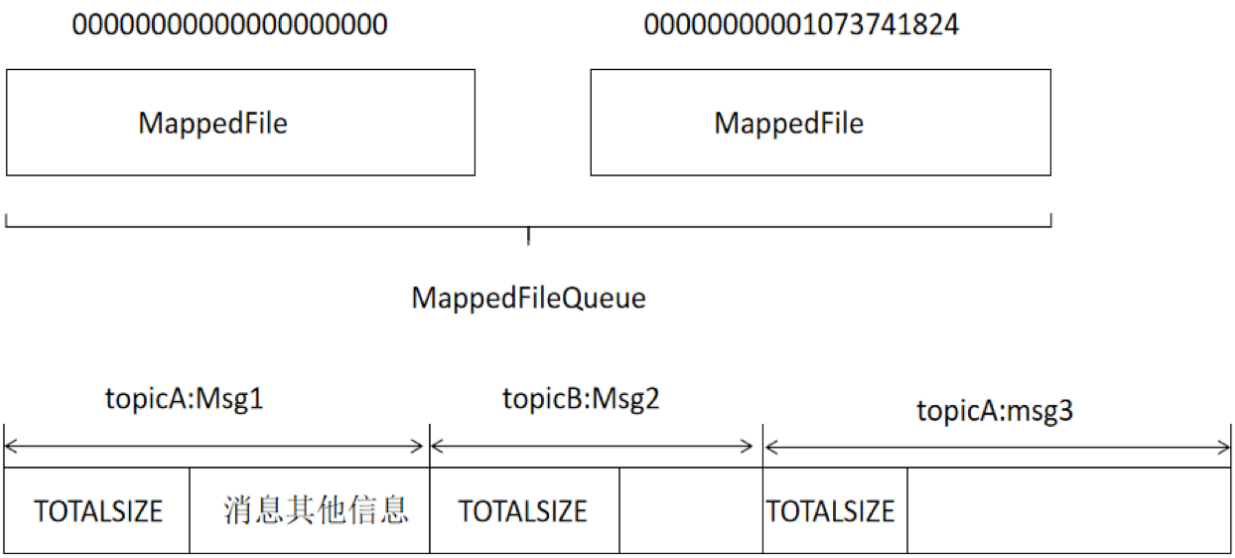
Broker规避

在上一节的问题中，极端情况下，nameserver数据不一致，或nameserver无法正确的收到broker的健康信息。此时，消费者从nameserver中获取到的broker实际上可能已经下线，在nameserver中不保证该broker的健康性。故在消费者在向broker发送消息时，若失败，则会采取快速失败的策略，并规避当前的broker，向其他的broker发送消息。

如果想某一个Broker发送消息失败后，会设置一个规避策略。例如在接下来的5分钟内选择队列时，发送错误的Broker中的队列将不参加队列负载，即会跳过发生错误的Broker服务器上的调度队列，只会选择其他Broker服务上的队列。

RockerMQ的存储设计

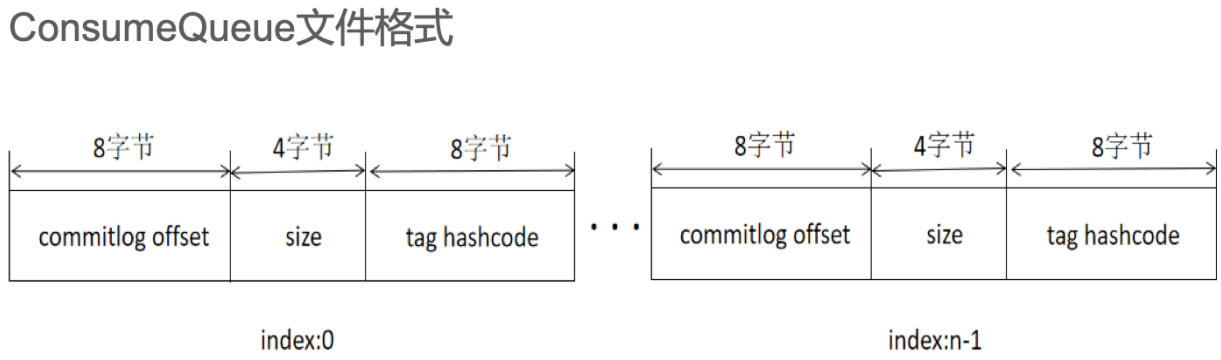
Commitlog



消息存储文件：所有主题的消息随着到达Broker的顺序写入commitlog文件，每个文件默认为1G,文件的命名也及其巧妙，使用该存储在消息文件中的第一个全局偏移量来命名文件，这样的设计主要是方便根据消息的物理偏移量，快速定位到消息所在的物理文件。RocketMQ commitlog文件使用顺序写，极大提高了文件的写性能。

Commitlog 对于一个broker而言，仅有一个文件，该broker的所有topic均顺序写入该Commitlog

ConsumerQueue

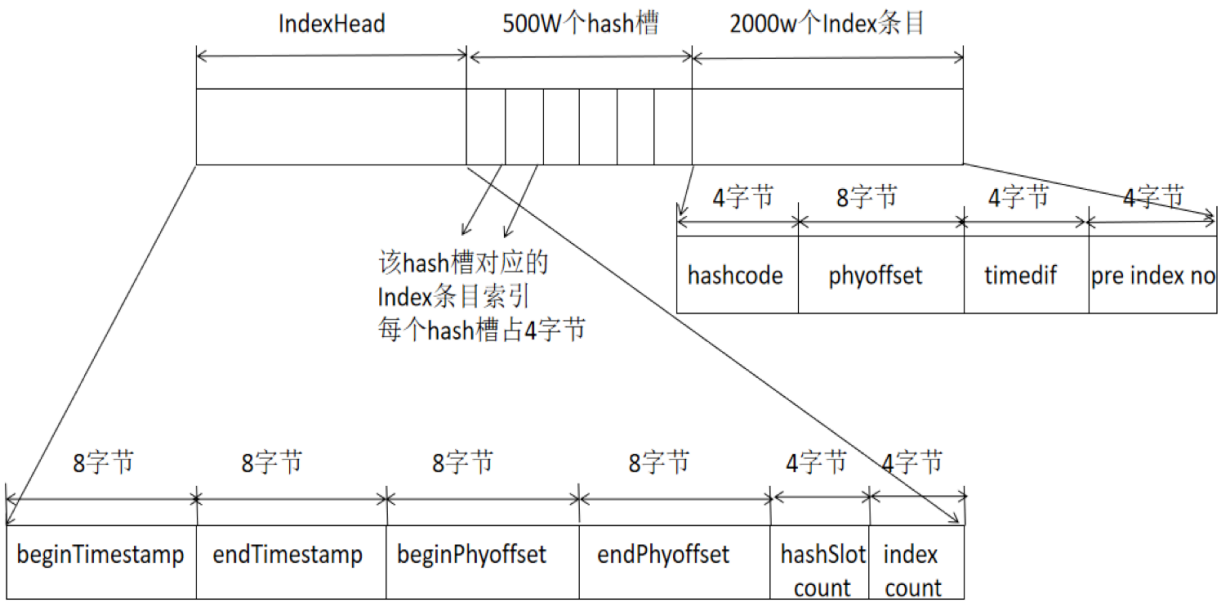


ConsumeQueue文件：消息消费队列文件，是Commitlog文件的基于Topic的索引文件，主要用于消费者根据Topic消费消息，其组织方式为/topic/queue，同一个队列中存在多个文件，consumequeue设计极具技巧性，其每个条目使用固定长度（8字节commitlog物理偏移量、4字节消息长度、8字节tag hashcode），这里不是存储tag的原始字符串，而是存储hashcode，目的就是确保每个条目的长度固定，可以使用访问类似数组下标的方式来快速定位条目，极大的提高了ConsumeQueue文件的读取性能，试想一下，消息消费者根据topic、消息消费进度(consumeuqe逻辑偏移量)，即第几个Consumeque条目，这样根据消费进度去访问消息的方法为使用逻辑偏移量logicOffset * 20即可找到该条目的起始偏移量(consumequeue文件中的偏移量)，然后读取该偏移量后20个字节即得到了一个条目，无需遍历consumequeue文件。

ConsumerQueue 是基于Topic 的索引文件，每个topic，都会有其独自的topic文件

之所以存储tag hashcode 是为了固定每一块数据的大小，所以访问时可以用近乎数组的形式访问

IndexFile



IndexFile文件基于物理磁盘文件实现Hash索引。其文件由40字节的文件头、500W个hash槽，每个hash槽为4个字节，最后由2000万个Index条目，每个条目由20个字节构成，分别为4字节的索引key的hashcode、8字节消息物理偏移量、4字节时间戳、4字节的前一个Index条目(hash冲突的链表结构)。

IndexHead

名词解释

1. beginTimestamp : 该索引文件的第一个消息(Message)的存储时间(落盘时间) 物理位置(pos: 0-7) 8bytes
2. endTimestamp : 该索引文件的最后一个消息(Message)的存储时间(落盘时间) 物理位置(pos: 8-15) 8bytes
3. beginPhyoffset : 该索引文件第一个消息(Message)的在CommitLog(消息存储文件)的物理位置偏移量(可以通过该物理偏移直接获取到该消息) 物理位置(pos: 16-23) 8bytes
4. beginPhyoffset : 该索引文件最后一个消息(Message)的在CommitLog(消息存储文件)的物理位置偏移量 (pos: 24-31) 8bytes
5. hashSlotCount : 该索引文件目前的hash slot的个数 (pos: 32-35) 4bytes
6. indexCount : 该索引文件目前的索引个数 (pos: 36-39) 4bytes

500W个hash槽

每个槽存放的是什么东西?

是后续Index条目的物理偏移量?

还是后续Index条目的 类似于 下标的东西?

RocketMQ消息消费

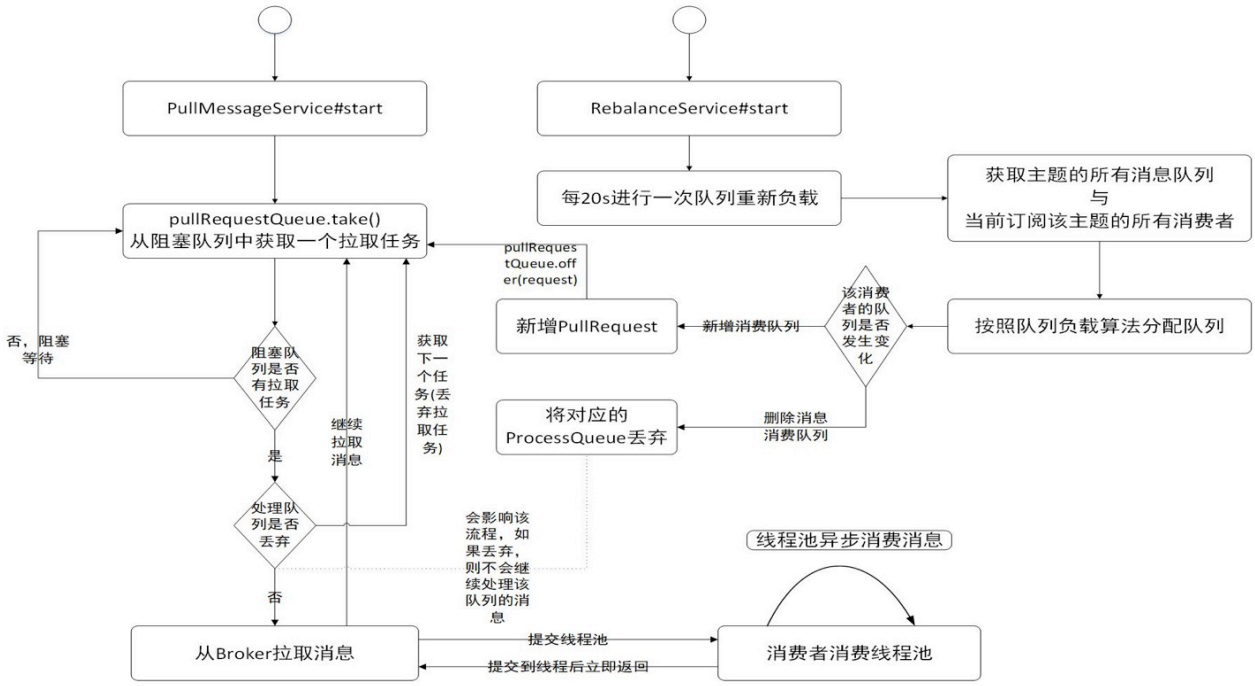
消息消费通常需要考虑消息队列负载、消费模式、拉取机制、消息过滤、消息消费（处理消息）、消费进度反馈、消息消费限流等方面。

1. 消息队列负载：集群内(同一消费组)内的消费者共同承担主题下所有消息的消费，即一条消息只能被集群中一个消费者消费。RocketMQ的队列负载原则是一个消费者可以承担同一主题下的多个消息消费队列，但同一个消息消费队列同一时间只允许被分配给一个消费者。
2. 消息消费模式：RocketMQ执行集群消费和广播消费两种模式。
3. 消息拉取模式：RocketMQ消息拉取支持推、拉两种模式，其本质为拉模式。
4. 消息消费：RocketMQ支持顺序消息、并发消费两种模式，每个消费组使用独立的线程池来处理拉取到的消息。
5. RocketMQ消息消费端的限流主要包含两个维度：
    - 消息堆积数量

      如果消息消费处理队列中的消息条数超过1000条会触发消费端的流控，其具体做法是放弃本次拉取动作，并且延迟50ms后将放入该拉取任务放入到pullRequestQueue中，每1000次流控会打印一次消费端流控日志。
    - 消息堆积大小

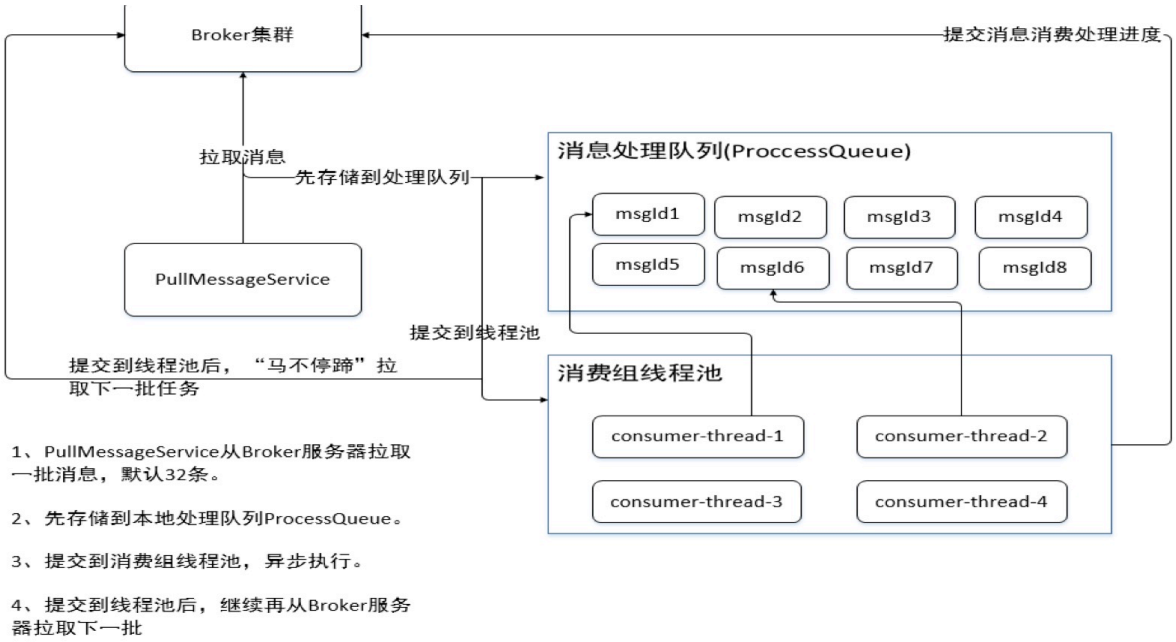
      如果处理队列中堆积的消息总内存大小超过100M,同样触发一次流控。

并发消息拉取与消费处理流程



并发消息拉取与消费的几个核心要点：

- PullMessageService线程与RebalanceService线程的交互
- 每消费组一个线程池，用来异步处理消息。
- 消费进度反馈



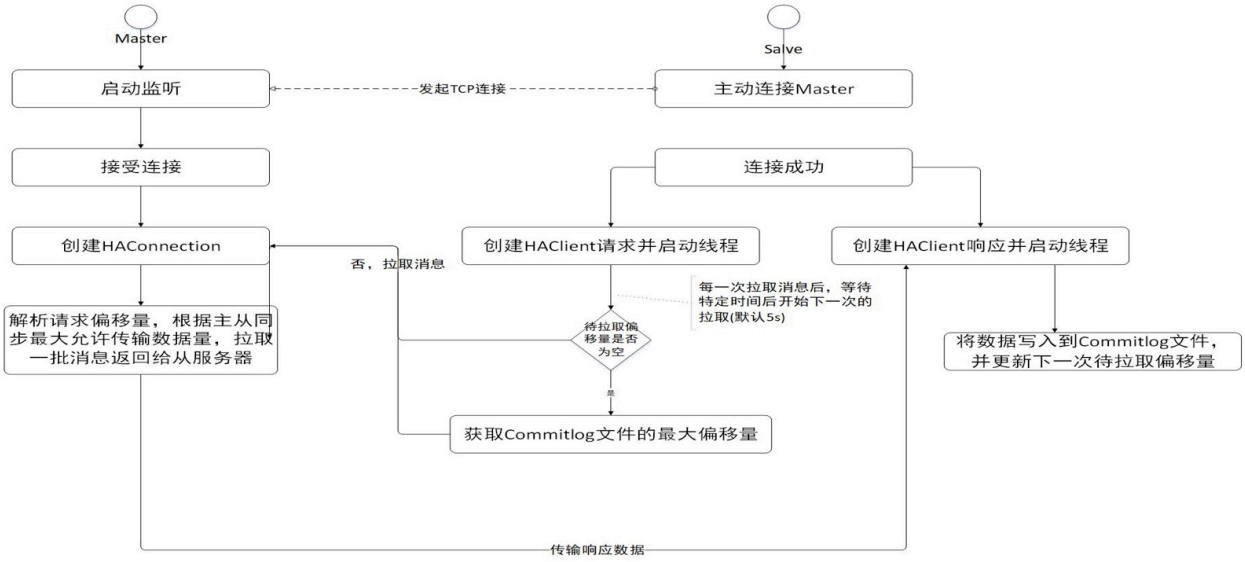
前提：一般来说集群内(同一消费组内)的消费者共同承担主题下所有消息的消费，即一条消息只能被集群中一个消费者消费。RocketMQ的队列负载原则是一个消费者可以承担同一主题下的多个消息消费队列，但同一个消息消费队列同一时间只允许被分配给一个消费者。

一般来说，处理消息队列（ProcessQueue）中消息的，会有一个线程池。所以可能存在先来的消息未被消费，而后来的消息被消费了，此时如何提交消费处理进度呢？提交那个连续的最后一个消息。

若此时，服务宕机需要重启，存在可能性，部分消息被重新消费，这就要求消费者接口设计成幂等，或者消费者通过key+redis 的方案解决（不在RocketMQ解决，减轻mq的复杂性）

自己思考，这种设计，也与TCP协议中，滑动窗口算法有异曲同工之处。

主从同步



1. 首先启动Master并在指定端口监听。
2. 客户端启动，主动连接Master，建立TCP连接。
3. 客户端向服务端拉取消息，如果是第一次拉取的话，先获取本地commitlog文件中最大的偏移量，

以该偏移量向服务端拉取消息。如果没有消息可读时会阻塞5s。

4. 服务端解析请求，并返回一批数据给客户端。
5. 客户端收到一批消息后，将消息写入本地commitlog文件中，然后向Master汇报拉取进度，并更新下一次待拉取偏移量。
6. 然后重复第3步。

Q&A

1、主，从服务器都在运行过程中，消息消费者是从主拉取消息还是从从拉取?

答：默认情况下，RocketMQ消息消费者从主服务器拉取，当主服务器积压的消息超过了物理内存的40%，则建议从从服务器拉取。但如果slaveReadEnable为false，表示从服务器不可读，从服务器也不会接管消息拉取。

2、当消息消费者向从服务器拉取消息后，会一直从从服务器拉取?

答：不是的。分如下情况：

1）如果从服务器的slaveReadEnable设置为false，则下次拉取，从主服务器拉取。

2）如果从服务器允许读取并且从服务器积压的消息未超过其物理内存的30%，下次拉取使用的Broker为订阅组的brokerId指定的Broker服务器，该值默认为0，代表主服务器。

3）如果从服务器允许读取并且从服务器积压的消息超过了其物理内存的30%，下次拉取使用的Broker为订阅组的whichBrokerWhenConsumeSlowly指定的Broker服务器，该值默认为1，代表从服务器。

3、主从服务消息消费进是如何同步的?

答：消息消费进度的同步时单向的，从服务器开启一个定时任务，定时从主服务器同步消息消费进度；无论消息消费者是从主服务器拉的消息还是从从服务器拉取的消息，在向Broker反馈消息消费进度时，优先向主服务器汇报；消息消费者向主服务器拉取消息时，如果消息消费者内存中存在消息消费进度时，主会尝试跟新消息消费进度。