# RocketMQ详解

# RocketMQ 消息路由机制

Broker 服务器在启动时向所有 Namesrv 注册，Producer 在发送消息之前先从 Namesrv 获取 Broker 服务器地址列表，然后根据负载算法从列表中选择一台消息服务器进行消息发送。Namesrv 与每台 Broker 服务器保持长连接，并间隔 30s 检测 Broker 是否存活，如果检测到 Broker 宕机则从路由注册表中将其移除。

Namesrv 是RocketMQ的寻址服务。用于把 Broker的路由信息做聚合。客户端依靠 Namesrv 决定去获取对应topic的路由信息，从而决定对哪些 Broker 做连接。

1. Namesrv 是一个几乎无状态的结点，Namesrv 之间采取share-nothing的设计，互不通信。
2. 对于一个 Namesrv 集群列表，客户端连接 Namesrv 的时候，只会选择随机连接一个结点，以做到负载均衡。
3. Namesrv所有状态都从Broker上报而来，本身不存储任何状态，所有数据均在内存。
4. 如果中途所有Namesrv全部挂了，影响到路由信息的更新，不会影响Producer和Broker的通信。

## 1.1 路由注册

通过Broker与Namesrv的心跳功能实现。

Broker启动时向集群中所有的Namesrv发送心跳，每隔30s向集群中所有 Namesrv 发送心跳包，Namesrv 收到 Broker 心跳包时会更新 brokerLiveTable 缓存中 BrokerLivelnfo 的 lastUpdateTimestamp，然后 NameServer 每隔10s扫描 brokerLiveTable 一次，如果连续120s没有收到心跳包，Namesrv 将移除该 Broker 的路由信息同时关闭Socket连接。

## 1.2 路由删除

删除 topicQueueTable、 brokerAddrTable、 brokerLiveTable、 filterServerTable删除与该 Broker 相关的信息。

1. Namesrv定时扫描bokerLiveTable检测上次心跳包与当前系统时间的时间差，如果时间差大于2分钟，则需要删除该broker信息。
2. Broker在正常被关闭的情况下，会执行unregisterBroker方法。

## 1.3 路由发现

RocketMq路由发现不是实时的，当Topic路由出现变化后，Namesrv不主动推送给客户端，而是由客户端定时拉去Topic最新的路由信息。

当RequestCode等于GET\_ROUTEINTO\_BY\_TOPIC时会去获取路由信息，需要注意就是顺序消息的区分就是这个时候填充 orderTopicConf 来进行区别。

/\*\*

\* 顺序消息配置。

\* 格式为--> brokerName1:num1;brokerName2:num2

\*/

private String orderTopicConf;

为了保证queue的顺序，这个顺序是固定的，生产端会使用这个顺序来进行选择发送，如果是顺序消息，通过这个顺序可以保证都落到同一个queue；如果不配置，如果broker宕机后，queue就会发生变化，无法保证顺序。

## 1.4 定时任务

1. Broker每隔30s给Namesrv发一次心跳。
2. Namesrv每隔10s扫描一次Broker，移除处于不激活状态的Broker。
3. Namesrv每隔10分钟打印一次KV配置。

## 1.5 问题

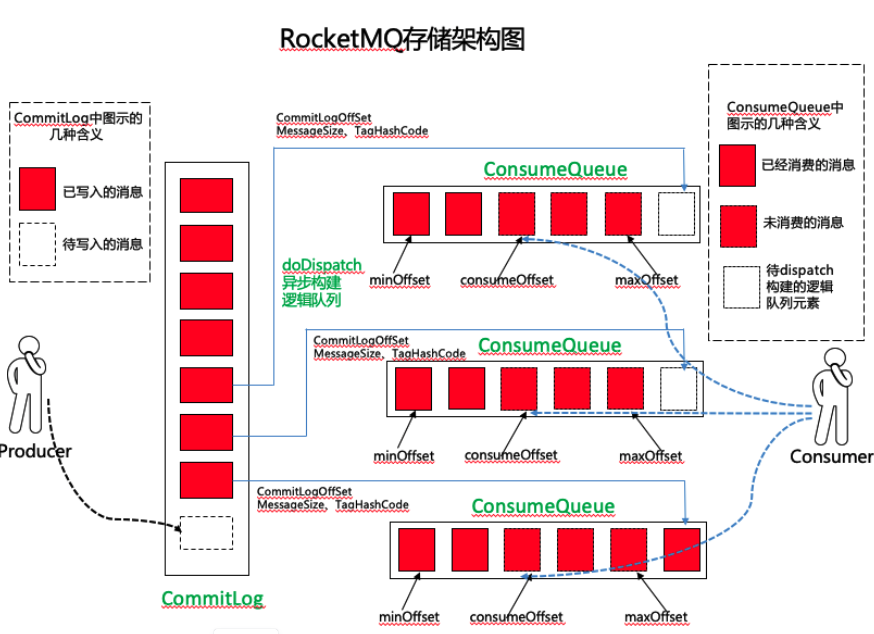
1. 因为 Producer 定时拉取 Namesrv 上 Broker 信息，就会存在 Namesrv 根据主题获取路由信息包含宕机的Broker，会导致消息发送失败，那么这种情况怎么处理？

解答：消息发送队列负载默认采用轮询机制，消息发送时默认选择重试机制来保证消息发送的高可用。当 Broker 宕机后，虽然消息发送者无法第一时间感知 Broker 宕机，但是当消息发送者向 Broker 发送消息返回异常后，生产者会在接下来一定时间内，例如5分钟内不会再次选择该 Broker上的队列，这样就规避了发生故障的 Broker，结合重试机制，巧妙实现消息发送的高可用。

1. RocketMQ为什么不实用zookeeper？

解答：这个问题首先我们来看 ZooKeeper 能实现什么，选举、Node节点、数据强一致性，但是 Namesrv 采取share-nothing的设计，都是无状态节点，节点之间消息不需要同步数据，再者都是非持久化方式存储，综合考虑。

# 消息存储文件设计

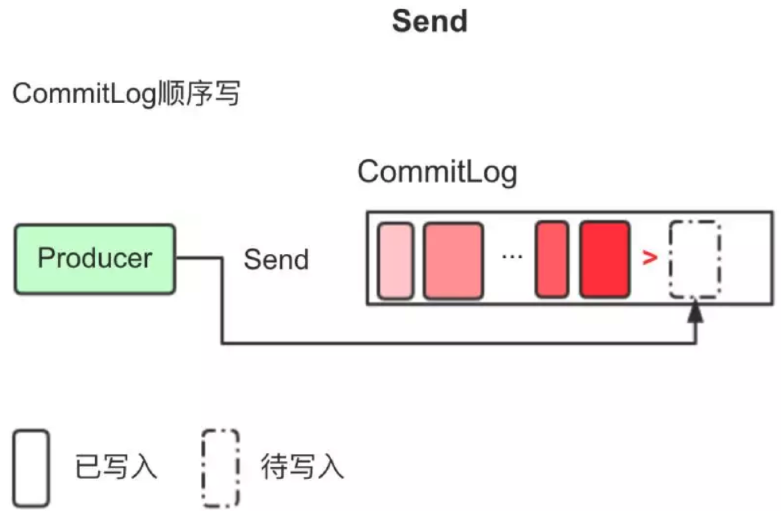


## 2.1 整体架构设计

1. 消息生产与消息消费相互分离，Producer端发送消息最终写入的是CommitLog（消息存储的日志数据文件），Consumer端先从ConsumeQueue（消息逻辑队列）读取持久化消息的起始物理位置偏移量offset、大小size和消息Tag的HashCode值，随后再从CommitLog中进行读取待拉取消费消息的真正实体内容部分；
2. RocketMQ的CommitLog文件采用混合型存储（所有的Topic下的消息队列共用同一个CommitLog的日志数据文件），并通过建立类似索引文件 ConsumeQueue 的方式来区分不同Topic下面的不同 MessageQueue 的消息，同时为消费消息起到一定的缓冲作用（只有ReputMessageService异步服务线程通过doDispatch异步生成了ConsumeQueue队列的元素后，Consumer端才能进行消费）。这样，只要消息写入并刷盘至CommitLog文件后，消息就不会丢失，即使ConsumeQueue中的数据丢失，也可以通过CommitLog来恢复。
3. RocketMQ每次读写文件的时候真的是完全顺序读写么？这里，发送消息时，生产者端的消息确实是顺序写入CommitLog；订阅消息时，消费者端也是顺序读取ConsumeQueue，然而根据其中的起始物理位置偏移量offset读取消息真实内容却是随机读取CommitLog。也就是说顺序写，随机读。

## 2.2 消息存储

### 2.2.1 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。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。

消息顺序写入CommitLog，Producer不直接与ConsumeQueue打交道，为了保证消息存储不发生混乱，会对CommitLog写加锁。

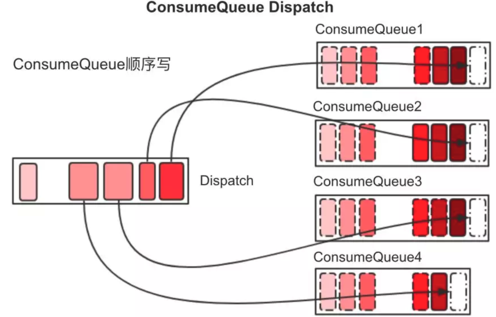
### 2.2.2 ConsumeQueue

消息消费的逻辑队列，其中包含了这个MessageQueue在CommitLog中的起始物理位置偏移量offset，消息实体内容的大小和Message Tag的哈希值。从实际物理存储来说，ConsumeQueue对应每个Topic和QueuId下面的文件。单个文件大小约5.72M，每个文件由30W条数据组成，每个文件默认大小为600万个字节，当一个ConsumeQueue类型的文件写满了，则写入下一个文件。

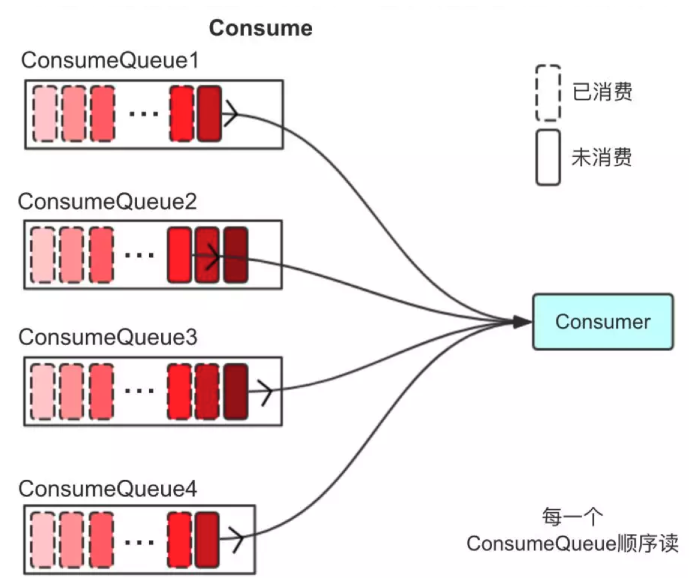
消费时，Consumer不直接与CommitLog打交道，而是从ConsumeQueue中去拉取数据。

**异步写入ConsumeQueue**

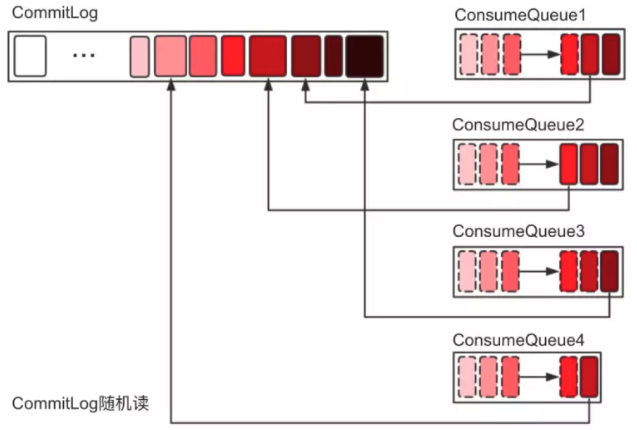
CommitLog持久后，Broker会将消息逐个异步Dispatch到对应的ConsumeQueue文件中。



消费者消费ConsumeQueue

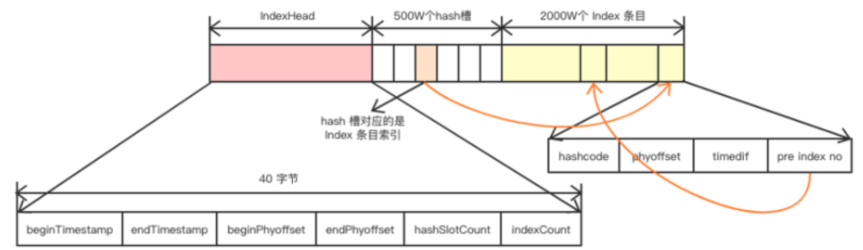


消费ConsumeQueue是没有消息真实内容的，但是里面有对CommitLog的偏移值引用，所以再次映射到CommitLog获取真实消息数据。 问题出现了，CommitLog会进行随机读，但是整体有序。



### 2.2.3 Index文件

用于为生成的索引文件提供访问服务，通过消息Key值查询消息真正的实体内容。在实际的物理存储上，文件名则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。



**IndexHeader头部**

beginTimestamp：第一个索引消息落在Broker的时间戳；

endTimestamp：最后一个索引消息落在Broker的时间戳；

beginPhyOffset：第一个索引消息在commitlog的偏移量（commitlog 文件偏移量）；

endPhyOffset：最后一个索引消息在commitlog的偏移量（commitlog 文件偏移量）；

hashSlotCount：构建索引占用的槽位数（不是hash槽使用个数，所以意义不大）；

indexCount：构建的索引个数（Index 条目列表当前已使用的个数）；

**Hash槽**

Slot Table里面的每一项保存的是这个topic-key是第几个索引；根据topic-key的Hash值除以500W取余得到这个Slot Table的序列号，然后将此索引的顺序个数存入此Table中。

**Index条目列表**

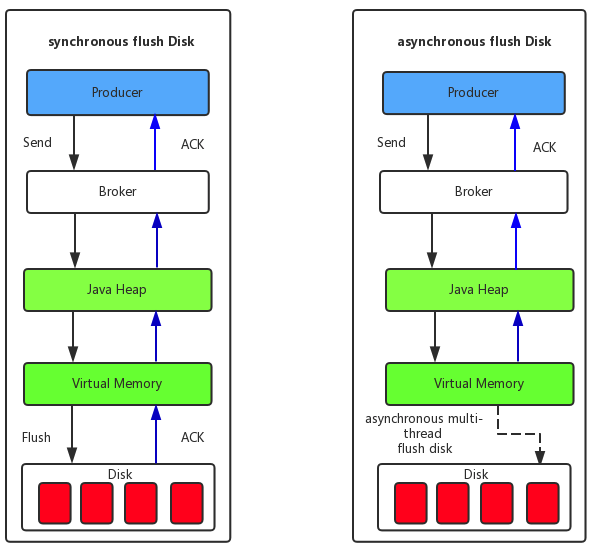
keyHash:topic-key(key是消息的key)的Hash值；

phyOffset:commitLog真实的物理位移；

timeOffset：时间位移，消息的存储时间与Index Header中beginTimestamp的时间差；

slotValue：当topic-key(key是消息的key)的Hash值取500W的余之后得到的Slot Table的slot位置中已经有值了（即Hash值取余后在Slot Table中有冲突时），则会用最新的Index值覆盖，并且将上一个值写入最新Index的slotValue中，从而形成了一个链表的结构。

### 2.2.4 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。

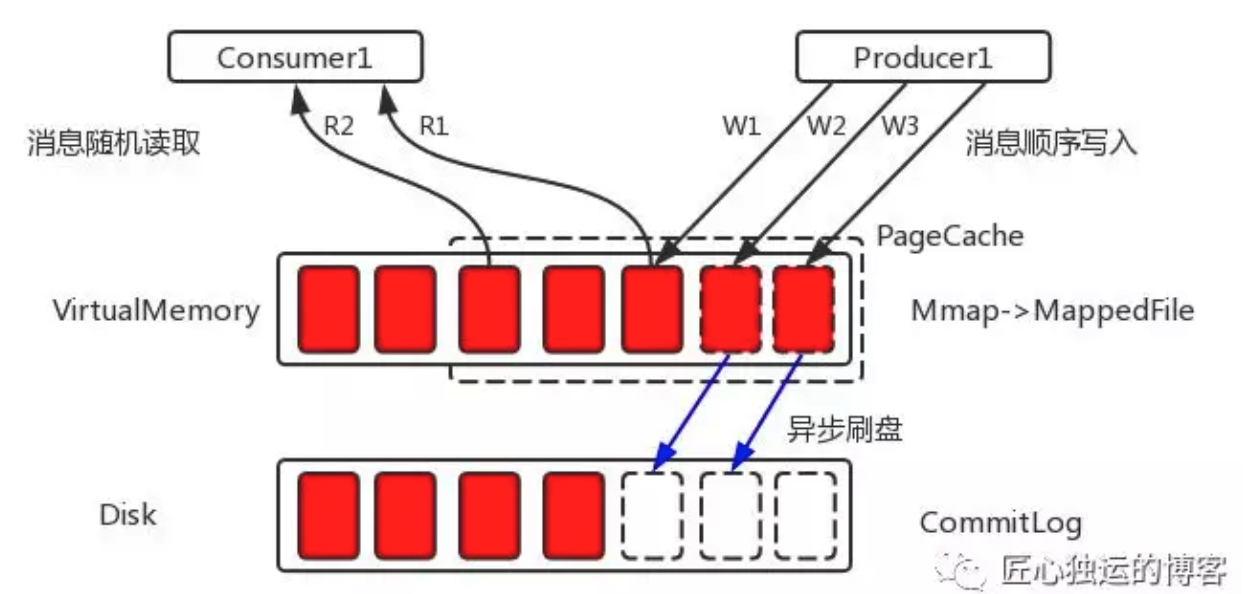
(2) 异步刷盘：能够充分利用OS的PageCache的优势，只要消息写入PageCache即可将成功的ACK返回给Producer端。消息刷盘采用后台异步线程提交的方式进行，降低了读写延迟，提高了MQ的性能和吞吐量。

## 2.3 MQ消息存储关键技术

### 2.3.1 OS的pageCache机制

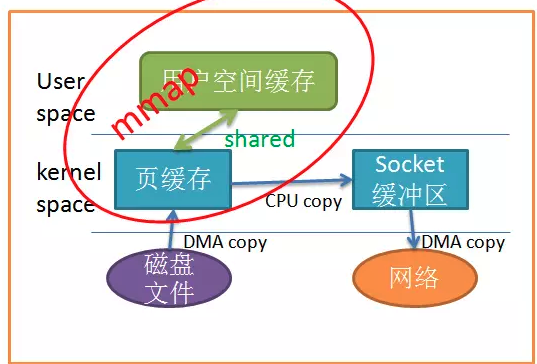
page cache 是 Linux内核实现的磁盘缓存，主要用来减少对磁盘的I/O操作，把磁盘访问变成物理内存访问。 无论通过标准I/O还是mmap，首先都是读入page cache，page cache 内核地址映射到page的物理地址。如果是标准I/O就拷贝到用户地址，如果是mmap，则将用户地址映射到相同的page物理地址。 不过mmap只建立了地址映射，所以当进程发起对映射空间访问时会引发缺页异常，从而引发磁盘文件到物理内存的拷贝。

1. 对于数据文件的读取，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（ps：顺序读入紧随其后的少数几个页面）。这样，只要下次访问的文件已经被加载至PageCache时，读取操作的速度基本等于访问内存。
2. 对于数据文件的写入，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。 对于文件的顺序读写操作来说，读和写的区域都在OS的PageCache内，此时读写性能接近于内存。RocketMQ的大致做法是，将数据文件映射到OS的虚拟内存中（通过JDK NIO的MappedByteBuffer），写消息的时候首先写入PageCache，并通过异步刷盘的方式将消息批量的做持久化（同时也支持同步刷盘）；订阅消费消息时（对CommitLog操作是随机读取），由于PageCache的局部性热点原理且整体情况下还是从旧到新的有序读，因此大部分情况下消息还是可以直接从Page Cache中读取，不会产生太多的缺页（Page Fault）中断而从磁盘读取。



### 2.3.2 Mmap

Mmap内存映射和普通标准IO操作的本质区别在于它并不需要将文件中的数据先拷贝至OS的内核IO缓冲区，而是可以直接将用户进程私有地址空间中的一块区域与文件对象建立映射关系，这样程序就好像可以直接从内存中完成对文件读/写操作一样。只有当缺页中断发生时，直接将文件从磁盘拷贝至用户态的进程空间内，只进行了一次数据拷贝。对于容量较大的文件来说（文件大小一般需要限制在1.5~2G以下），采用Mmap的方式其读/写的效率和性能都非常高。



## 2.4 问题

1. 当Topic数量增多到上百时，kafka的单个Broker的TPS大概降低了1个数量级，而RocketMQ在海量Topic的场景下，依然保持较高的TPS？

解答：在启动 Broker 和 Clients 之前，我们需要提供的一个 Namesrv 地址列表来访问 Namesrv。在RocketMQ中，可以用四种方式完成。

1. CommitLog 多个Topic 写一个文件 顺序写
2. Kafka topic较多时候,每个patition 写多个文件 这么多文件的顺序读写在并发时变成了随机读写
3. RocketMQ在队列非常多的情况下Consume Queue不也是和Kafka类似，虽然每一个文件是顺序IO，但整体是随机IO。
4. RocketMQ的ConsumeQueue是不会存储消息的内容，每个消息也就占用20 Byte，所以文件可以控制得非常小，绝大部分的访问还是Page Cache的访问，而不是磁盘访问。
5. CommitLog的”随机读”对性能的影响？

解答：只有一个CommitLog，虽然是随机读，但整体还是有序地读，只要那整块区域还在PageCache的范围内，还是可以充分利用PageCache。