深度解读 RocketMQ 存储机制

来源 | 阿里开发者公众号

简介：RocketMQ 实现了灵活的多分区和多副本机制，有效的避免了集群内单点故障对于整体服务可用性的影响。存储机制和高可用策略是 RocketMQ 稳定性的核心，社区上关于 RocketMQ 目前存储实现的分析与讨论一直是一个热议的话题。本文想从一个不一样的视角，着重于作者眼中的这种存储实现是在解决哪些复杂的问题，因此我从本文最初的版本中删去了冗杂的代码细节分析，由浅入深的分析存储机制的缺陷与优化方向。

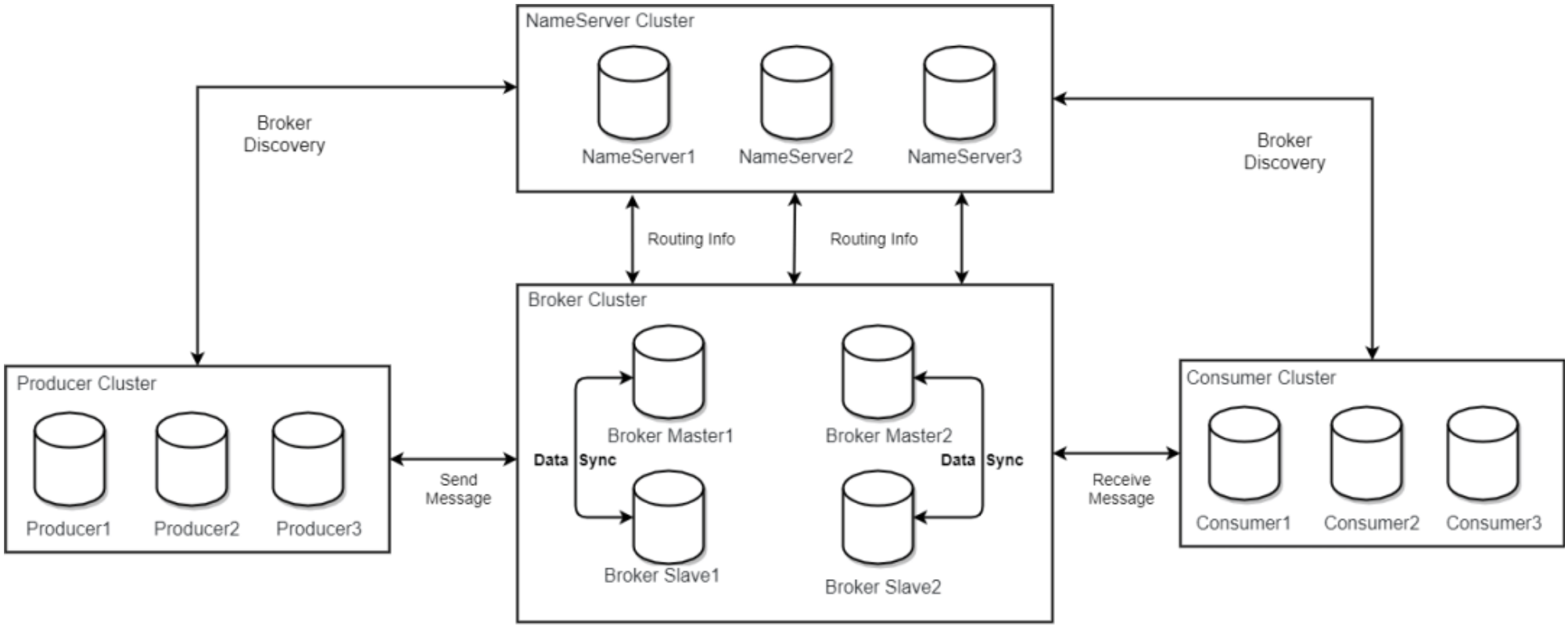
RocketMQ 实现了灵活的多分区和多副本机制，有效的避免了集群内单点故障对于整体服务可用性的影响。存储机制和高可用策略是 RocketMQ 稳定性的核心，社区上关于 RocketMQ 目前存储实现的分析与讨论一直是一个热议的话题。本文想从一个不一样的视角，着重于谈谈我眼中的这种存储实现是在解决哪些复杂的问题，因此我从本文最初的版本中删去了冗杂的代码细节分析，由浅入深的分析存储机制的缺陷与优化方向。

RocketMQ 的架构模型与存储分类

先来简单介绍下 RocketMQ 的架构模型。RocketMQ 是一个典型的发布订阅系统，通过 Broker 节点中转和持久化数据，解耦上下游。Broker 是真实存储数据的节点，由多个水平部署但不一定完全对等的副本组构成，单个副本组的不同节点的数据会达到最终一致。对于单个副本组来说同一时间最多只会有一个可读写的 Master 和若干个只读的 Slave，主故障时需要选举来进行单点故障的容错，此时这个副本组是可读不可写的。NameServer 是独立的一个无状态组件，接受 Broker 的元数据注册并动态维护着一些映射关系，同时为客户端提供服务发现的能力。在这个模型中，我们使用不同主题 (Topic) 来区分不同类别信息流，为消费者设置订阅组 (Group) 进行更好的管理与负载均衡。

如下图中间部分所示：

1. 服务端 Broker Master1 和 Slave1 构成其中的一个副本组。
2. 服务端 Broker 1 和 Broker 2 两个副本组以负载均衡的形式共同为客户端提供读写。



RocketMQ 目前的存储实现可以分为几个部分：

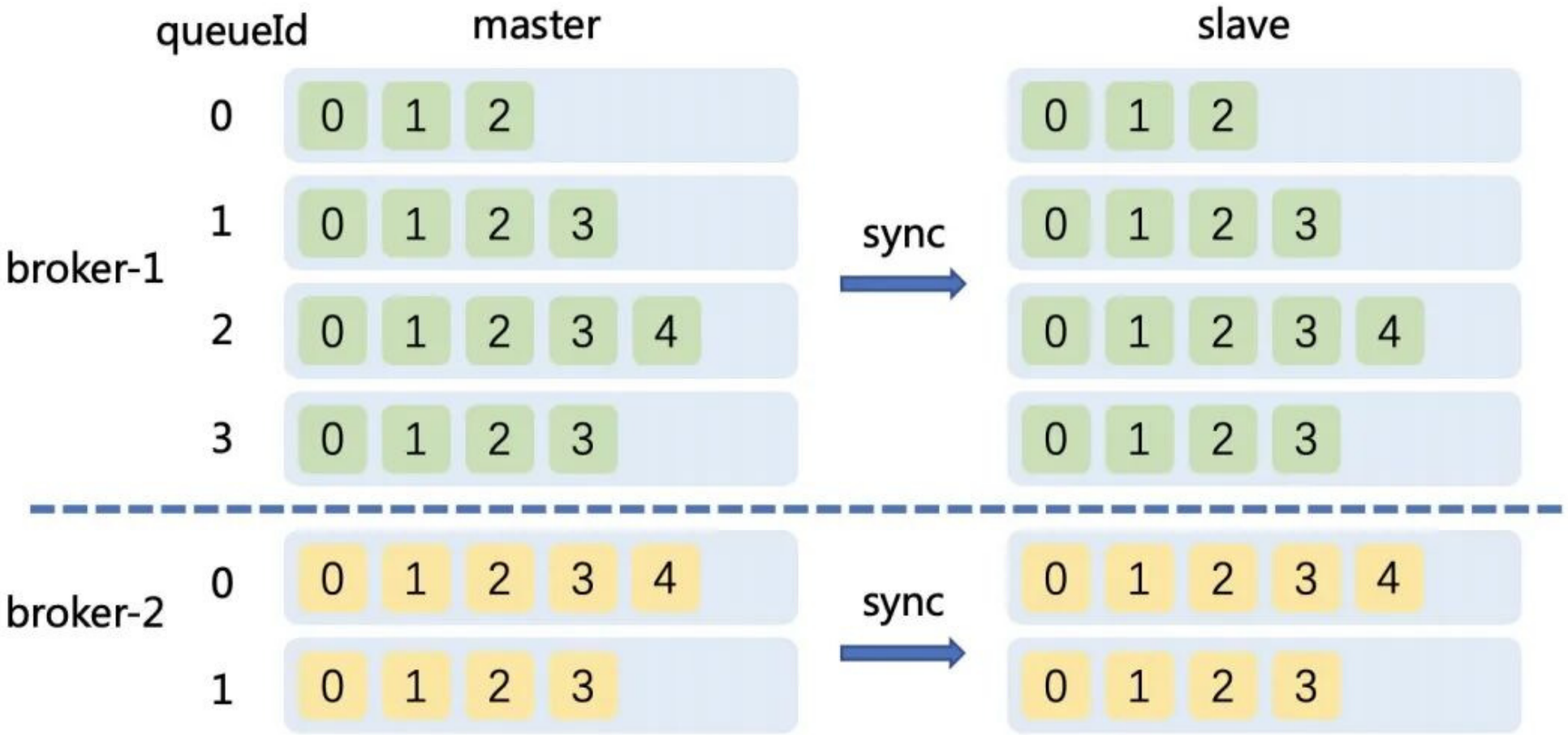
1. 元数据管理

- 具体指当前存储节点的主题 Topic，订阅组 Group，消费进度 ConsumerOffset。
- 多个配置文件 Config，以及为了故障恢复的存储 Checkpoint 和 FileLock。
- 用来记录副本主备身份的 Epoch / SN (sequence number) 文件等（5.0-beta 引入，也可以看作 term）

2. 消息数据管理，包括消息存储的文件 CommitLog，文件版定时消息的 TimerLog。
3. 索引数据管理，包括按队列的顺序索引 ConsumeQueue 和随机索引 IndexFile。

元数据管理与优化

为了提升整体的吞吐量与提供跨副本组的高可用能力，RocketMQ 服务端一般会为单个 Topic 创建多个逻辑分区，即在多个副本组上各自维护部分分区 (Partition)，我们把它称为队列 (MessageQueue)。同一个副本组上同一个 Topic 的队列数相同并从 0 开始连续编号，不同副本组上的 MessageQueue 数量可以不同。



例如 topic-a 可以在 broker-1 主副本上有 4 个队列，编号 (queueId) 是 0-3，在 broker-1 备副本上完全相同，但是 broker-2 上可能就只有 2 个队列，编号 0-1。在 Broker 上元数据的组织管理方式是与上述模型匹配的，每一个 Topic 的 TopicConfig，包含了几个核心的属性，名称，读写队列数，权限与许多元数据标识，这个模型类似于 K8s 的 StatefulSet，队列从 0 开始编号，扩缩队列都在尾部操作（例如 24 个队列缩分区到 16，是留下了编号为 0-15 的分区）。这使得我们无需像 Kafka 一样对每个分区单独维护状态机，同时大幅度的简化了关于分区的实现。

我们会在存储节点的内存中简单的维护 Map<String, TopicConfig> 的结构来将 TopicName 直接映射到它的具体参数。这个设计足够的简单，也隐含了一些

缺陷，例如它没有实现一个原生 Namespace 机制来实现存储层面上多租户环境下的元数据的隔离，这也是 RocketMQ 5.0 向云原生时代迈进过程中一个重要的演进方向。

当 Broker 接收到外部管控命令，例如创建或删除一些 Topic，这个内存 Map 中就会对应的更新或者删除一个 KV 对，需要立刻序列化一次并向磁盘覆盖，否则就会造成丢失更新。对于单租户的场景下，Topic (Key) 的数量不会超过几千个，文件大小也只有数百 KB，速度是非常快。但是在云上大多租的场景下，一个存储节点的 Topic 可以达到十几 MB。每次变更一个 KV 就全量向磁盘覆盖写这个大文件，这个操作的开销非常高，尤其是在数据需要跨集群，跨节点迁移，或者应急情况下扩容逃生场景下，同步写文件严重延长了外围管控命令的响应时间，也成为云上大共享模式下严峻的挑战之一。在这个背景下，两个解决方案很自然的就产生了，即批量更新接口和增量更新机制。

1. 批量更新指每次服务端可以接受一批 TopicConfig 的更新，这样 Broker 刷写文件的频率就显著的降低。
2. 增量更新指将这个 Map 的持久化换成逻辑替换成 KV 型的数据库或实现元数据的 Append 写，以 Compaction 的形式维护一致性。

除了最重要的 Topic 信息，Broker 还管理着 Group 信息，消费组的消费进度 ConsumerOffset 和多个配置文件。Group 的变更和 Topic 类似，都是只有新建或者删除时才需要持久化。而 ConsumeOffset 是用来维护每个订阅组的消费进度的，结构如 Map<String/_ topicName@groupId _/, Map>。这里我们从文件本身的作用和数据结构的角度进行分析下，Topic Group 虽然数量多，但是变化的频率还是比较低的，而提交与持久化位点时时刻刻都在进行，进而导致这个 Map 几乎在实时更新，但是上一次更新后的数据 (last commit offset) 对当前来说又没有什么用，并且允许丢少量更新。所以这里 RocketMQ 没有像 Topic Group 那样采取数据变化时刷写文件，而是使用一个定时任务对这个 Map 做 CheckPoint。这个周期默认是 5 秒，所以当服务端主备切换或者正常发布时，都会有秒级的消息重复。

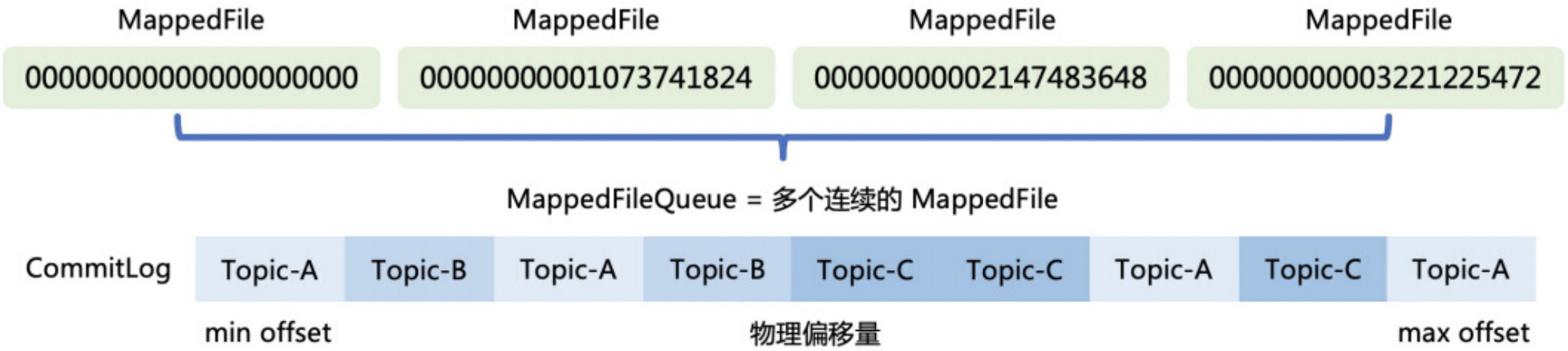
那么这里还有没有优化的空间呢？事实上大部分的订阅组都是不在线的，每次我们也只需要更新位点有变化的这部分订阅组。所以这里我们可以采取一个差分优化的策略（参加过 ACM 的选手应该更熟悉，搜索差分数据传输），在主备同步 Offset 或者持久化的时候只更新变化的内容。假如此时我们除了知道当前的 Offset，还需要一个历史 Offset 的提交记录怎么办，这种情况下，也使用一个内置的系统 Topic 来保存每次提交（某种意义上的自举实现，Kafka 就是使用一个内部 Topic 来保存位点），通过回放或查找消息来追溯消费进度。由于 RocketMQ 支持海量 Topic，元数据的规模会更加大，采用目前的实现开销更小。所以选用哪种实现完全是由我们所面对的需求决定的，实现也可以是灵活多变的。当然，在 RocketMQ 元数据管理上，如何在上层保证分布式环境下多个副本组上的数据一致又是另外一个令人头疼的难题，后续文章会更加详细的讨论这点。

消息数据管理

很多文章都提到 RocketMQ 存储的核心是一个极致优化的顺序写盘，以 append only 的形式不断的将新的消息追加到文件末尾。

RocketMQ 使用了一种称为 MappedByteBuffer 的内存映射文件的办法，将一个文件映射到进程的地址空间，实现文件的磁盘地址和进程的一段虚拟地址关联，实际上是利用了NIO 中的 FileChannel 模型。在进行这种绑定后，用户进程就可以用指针（偏移量）的形式写入磁盘而不用进行 read / write 的系统调用，减少了数据在缓冲区之间来回拷贝的开销。当然这种内核实现的机制有一些限制，单个 mmap 的文件不能太大 (RocketMQ 选择了 1G)，此时再把多个

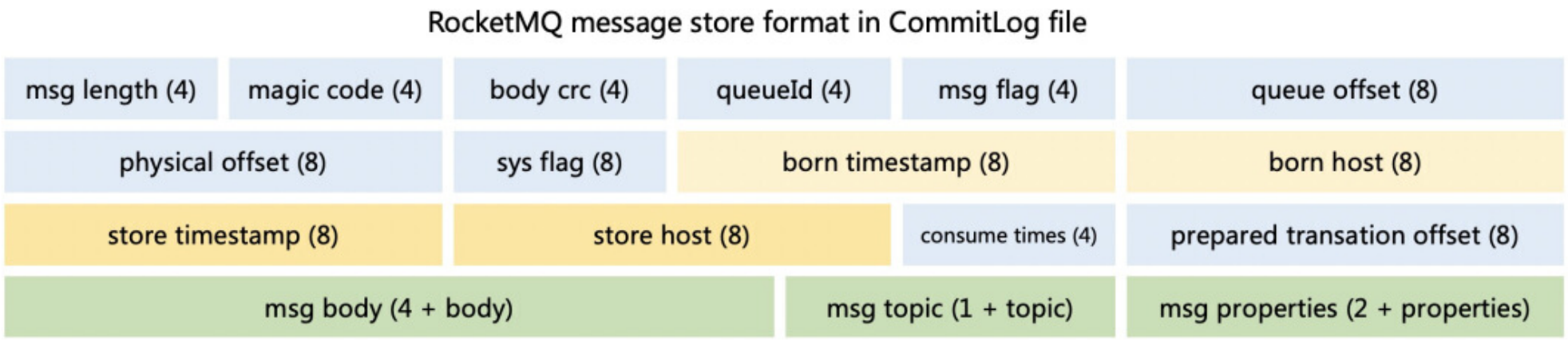
mmap 的文件用一个链表串起来构成一个逻辑队列 (称为 MappedFileQueue)，就可以在逻辑上实现一个无需考虑长度的存储空间来保存全部的消息。



这里不同 Topic 的消息直接进行混合的 append only 写，相比于随机写来说性能的提升非常显著的。还有一个重要的细节，这里的混合写的写放大非常低。当我们回头去看 Google 实现的 BigTable 的理论模型，各种 LSM 树及其变种，都是将原来的直接维护树转为增量写的方式来保证写性能，再叠加周期性的异步合并来减少文件的个数，这个动作也称为 Compaction。RocksDB 和 LevelDB 在写放大，读放大，空间放大都有几倍到几十倍的开销。得益于消息本身的不可变性，和非堆积的场景下，数据一旦写入中间代理 Broker 很快就会被下游消费掉的特性，此时我们不需要在写入时就维护 memTable，避免了数据的分发与重建。相比于各种数据库的存储引擎，消息这样近似 FIFO 的实现可以节省大量的资源，同时减少了 CheckPoint 的复杂度。对于同一个副本组上的多个副本之间的数据复制都是全部由存储层自行管理，这个设计类似于 bigtable 和 GFS，azure 的 Partation layer，也被称为 Layered Replication 分层架构。

单条消息的存储格式

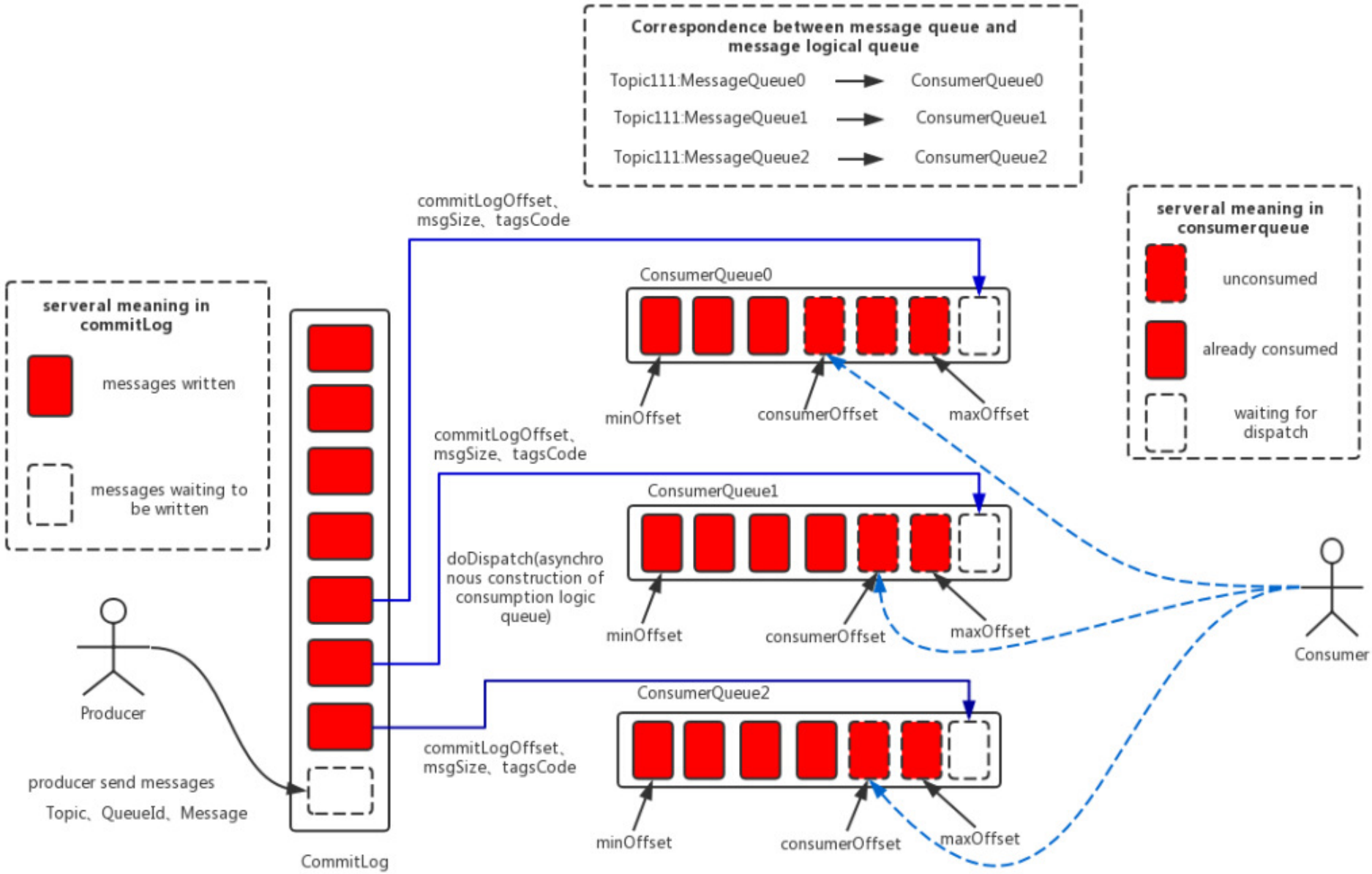
RocketMQ 有一套相对复杂的消息存储编码用来将消息对象序列化，随后再将一个非定长的数据落到上述的真实的写入到文件中，值得注意的存储格式中包括了索引队列的编号和位置。



存储时单条消息本身元数据占用的存储空间为固定的 91B + 部分属性，而消息的 payload 通常大于 2K，也就是说元数据带来的额外存储开销只增加了 5%-10% 左右。很明显，单条消息越大，存储本身额外的开销（比例）就相对的越少。但如果有大消息的诉求，例如想在 body 中保存一张序列化后的图片（二进制大对象），从目前的实现上说，在消息中保存引用，将真实数据保存到到其他组件，消费时读取引用（比如文件名或者 uk）其实是一个更合适的设计。

多条消息的连续写

上文提到，不同 Topic 的消息数据是直接混合追加数据到 CommitLog 中 (也就是上文提到的 MappedFileQueue)，再交由其他后端线程做分发。其实我觉得 RocketMQ 这种 CommitLog 与元数据的分开管理的机制也有一些 PacificaA (微软提出的复制框架) 的影子，从而以一种更简单的方式实现强一致。这里的强一致指的是在 Master Broker (对应于 PacificA 的 Primary) 对所有消息的持久化进行定序，再通过全序广播 (total order broadcast) 实现线性一致 (Linearizability)。这几种实现都会需要解决两个类似的问题，一是如何实现单机下的顺序写，二是如何加快写入的速度。


Correspondence between message queue and message logical queue
Topic111:MessageQueue0
ConsumerQueue0
Topic111:MessageQueue1
ConsumerQueue1
Topic111:MessageQueue2
ConsumerQueue2
serveral meaning in consumerqueue
unconsumed
already consumed
waiting for dispatch
serveral meaning in commitLog
messages written
messages waiting to be written
commitLogOffset、msgSize、tagsCode
commitLogOffset、msgSize、tagsCode
commitLogOffset、msgSize、tagsCode
doDispatch(asynchronous construction of consumption logic queue)
ConsumerQueue0
ConsumerQueue1
ConsumerQueue2
minOffset
consumerOffset
maxOffset
Producer
Consumer
producer send messages
Topic、QueueId、Message
CommitLog

如果是副本组是异步多写的（高性能中可靠性），将日志非最新（水位最高）的备选为主，主备的数据日志可能会产生分叉。在 RocketMQ 5.0 中，主备会通过基于版本的协商机制，使用落后补齐，截断未提交数据等方式来保证数据的一致性。顺便一提，RocketMQ 5.0 中实现了 logic queue 方案解决全局分区数变化的问题，这和 PacificaA 中通过 new-seal 新增副本组和分片 merge 给计算层读的一些优化策略有一些异曲同工之妙，具体可以参考这个设计方案 [10]。

独占锁实现顺序写

如何保证单机存储写 CommitLog 的顺序性，直观的想法就是对写入动作加独占锁保护，即同一时刻只允许一个线程加锁成功，那么该选什么样的锁实现才合适呢？RocketMQ 目前实现了两种方式。
1. 基于 AQS 的 ReentrantLock
2. 基于 CAS 的 SpinLock

那么什么时候选取 spinlock，什么时候选取 reentranlock？回忆下两种锁的实现，对于 ReentrantLock，底层 AQS 抢不到锁的话会休眠，但是 SpinLock 会一直抢锁，造成明显的 CPU 占用。SpinLock 在 trylock 失败时，可以预期持有锁的线程会很快退出临界区，死循环的忙等待很可能要比进程挂起等待更高效。这也是为什么在高并发下为了保持 CPU 平稳占用而采用方式一，单次请求响应时间短的场景下采用方式二能够减少 CPU 开销。两种实现适用锁内操作时间不同的场景，那线程拿到锁之后需要进行哪些动作呢？

1. 预计算索引的位置，即 ConsumeQueueOffset，这个值也需要保证严格递增。
2. 计算在 CommitLog 存储的位置，physicalOffset 物理偏移量，也就是全局文件的位置。
3. 记录存储时间戳 storeTimestamp，主要是为了保证消息投递的时间严格保序。

因此不少文章也会建议在同步持久化的时候采用 ReentrantLock，异步持久化的时候采用 SpinLock。那么这个地方还有没有优化的空间？目前可以考虑使用较新的 futex 取代 spinlock 机制。futex 维护了一个内核层的等待队列和许多个 SpinLock 链表。当获得锁时，尝试 cas 修改，如果成功则获得锁，否则就将当前线程 uaddr hash 放入到等待队列 (wait queue)，分散对等待队列的竞争，减小单个队列的长度。这听起来是不是也有一点点 concurrentHashMap 和 LongAddr 的味道，其实核心思想都是类似的，即分散竞争。

成组提交与可见性

受限于磁盘IO，块存储的响应通常非常慢。要求所有请求立即持久化是不可能的，为了提升性能，大部分的系统总是将操作日志缓存到内存中，比如在满

足”日志缓冲区中数据量超过一定大小 / 距离上次刷入磁盘超过一定时间” 的任一条件时，通过后台线程定期持久化操作日志。但这种成组提交的做法有一个很大的问题，存储系统意外故障时，会丢失最后一部分更新操作。例如数据库引擎总是要求先将操作日志刷入磁盘 (优先写入 redo log) 才能更新内存中的数据，这样断电重启则可以通过 undo log 进行事务回滚与丢弃。

在消息系统的实现上有一些微妙的不同，不同场景下对消息的可靠性要求不同，在金融云场景下可能要求主备都同步持久化完成消息才对下游可见，但日志场景希望尽可能低的延迟，同时允许故障场景少量丢失。此时可以将 RocketMQ 配置为单主异步持久化来提高性能，降低成本。此时宕机，存储层会损失最后一小段没保存的消息，而下游的消费者实际上已经收到了。当下游的消费者重置位点到一个更早的时间，回放至同样位点的时候，只能读取到了新写入的消息，但读取不到之前消费过的消息（相同位点的消息不是同一条），这是一种 read uncommitted。

这样会有什么问题呢？对于普通消息来说，由于这条消息已经被下游处理，最坏的影响是重置位点时无法消费到。但是对于 Flink 这样的流计算框架，以 RocketMQ 作为 Source 的时候，通过回放最近一次 CheckPoint 到当前的数据的 offset 来实现高可用，不可重复读会造成计算系统没法做到精确的 excatly once 消费，计算的结果也就不正确了。相应的解决的方案之一是在副本组多数派确认的时候才构建被消费者可见的索引，这么做宏观上的影响就是写入的延迟增加了，这也可以从另一个角度解读为隔离级别的提升带来的代价。

对于权衡延迟和吞吐量这个问题，可以通过加快主备复制速度，改变复制的协议等手段来优化，这里大家可以看下 SIGMOD 2022 关于 Kafka 运行在 RDMA 网络上显著降低延迟的论文《KafkaDirect: Zero-copy Data Access for Apache Kafka over RDMA Networks》链接[9]。

持久化机制

关于这一块的讨论在社区里讨论是最多的，不少文章都把持久化机制称为刷盘。我不喜欢这个词，因为它不准确。在 RocketMQ 中提供了三种方式来持久化，对应了三个不同的线程实现，实际使用中只会选择一个。

- 同步持久化，使用 GroupCommitService。
- 异步持久化且未开启 TransientStorePool 缓存，使用 FlushRealTimeService。
- 异步持久化且开启 TransientStorePool 缓存，使用 CommitRealService。

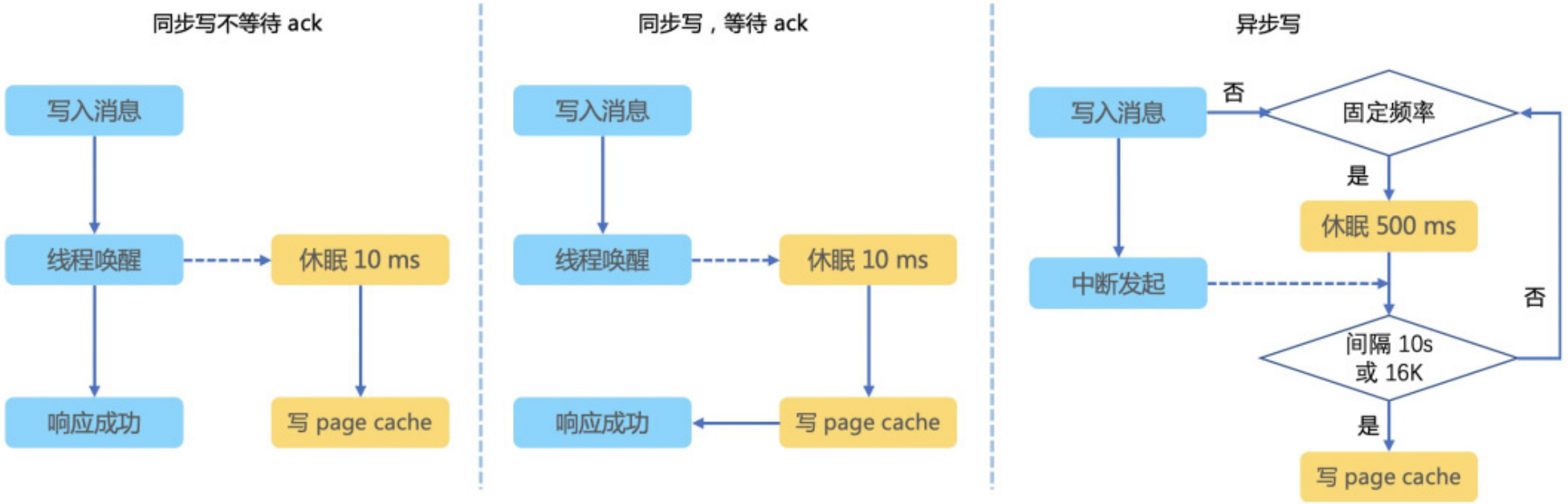
持久化

同步刷盘的落盘线程统一都是 GroupCommitService。写入线程仅仅负责唤醒落盘线程，将消息转交给存储线程，而不会等待消息存储完成之后就立刻返回了。我个人对这个设计的理解是，消息写入线程相对与存储线程来说也可以看作 IO 线程，而真实存储的线程需要攒批持久化会陷入中断，所以才要大费周章的做转交。

从同步刷盘的实现看，落盘线程每隔 10 ms 会检查一次，如果有数据未持久化，便将 page cache 中的数据刷入磁盘。此时操作系统 crash 或者断电，那未落盘的数据丢失会不会对生产者有影响呢？此时生产者只要使用了可靠发送 (指非 oneway 的 rpc 调用)，这时对于发送者来说还没有收到成功的响应，此时客

户端会进行重试，将消息写入其他可用的节点。

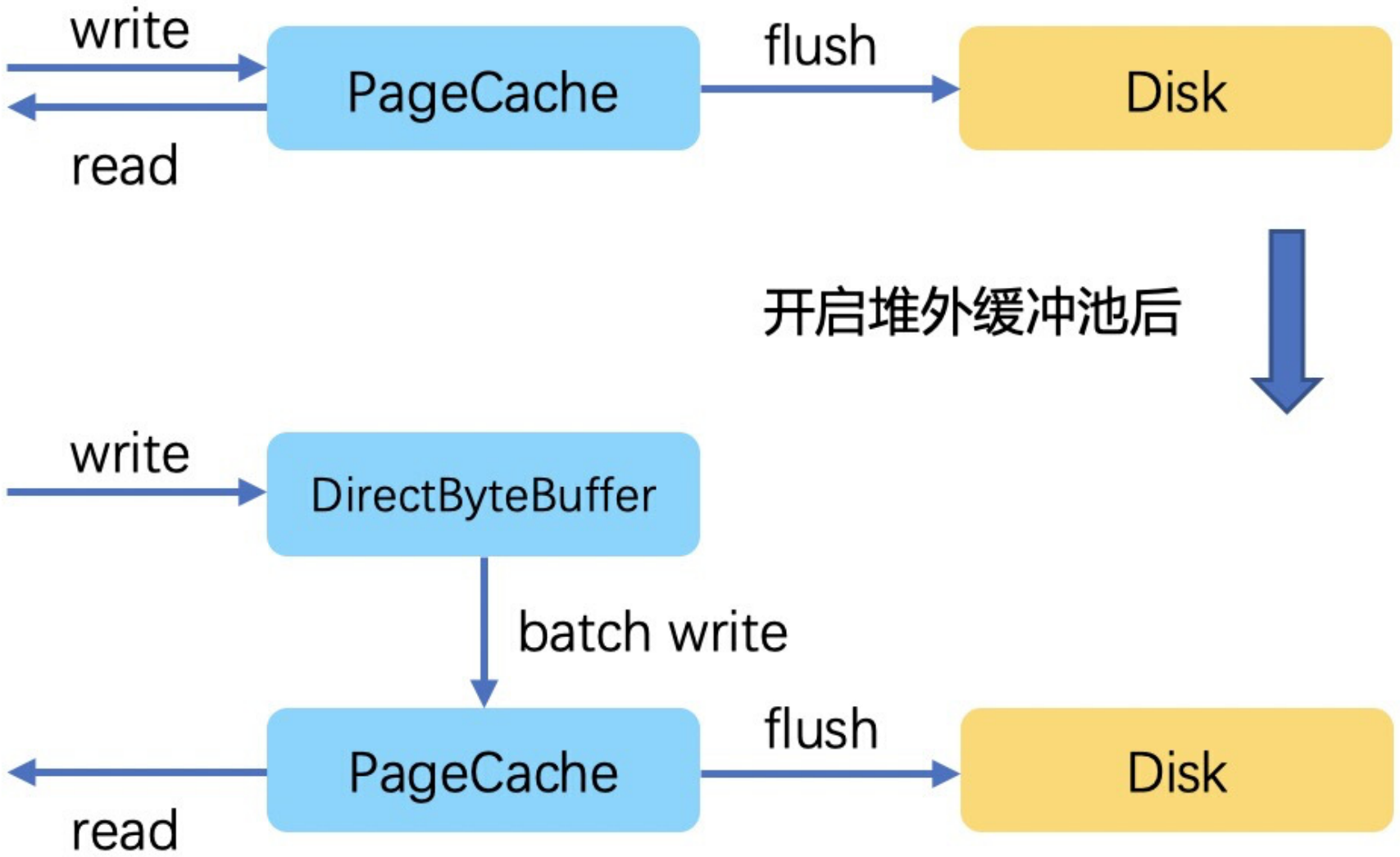
异步持久化对应的线程是 FlushRealTimeService，实现上又分为固定频率和非固定频率，核心区别是线程是否响应中断。所谓的固定频率是指每次有新的消息到来的时候不管，不响应中断，每隔 500ms（可配置）flush 一次，如果发现未落盘数据不足（默认 16K），直接进入下一个循环，如果数据写入量很少，一直没有填充满16K，就不会落盘了吗？这里还有一个基于时间的兜底方案，即线程发现距离上次写入已经很久了（默认 10 秒），也会执行一次 flush。但事实上 FileChannel 还是 MappedByteBuffer 的 force() 方法都不能精确控制写入的数据量，这里的写行为也只是对内核的一种建议。对于非固定频率实现，即每次有新的消息到来的时候，都会发送唤醒信号，当唤醒动作在数据量较大时，存在性能损耗，但消息量较少且情况下实时性好，更省资源。在生产中，具体选择哪种持久化实现由具体的场景决定。是同步写还是多副本异步写来保证数据存储的可靠性，本质上是读写延迟和和成本之间的权衡。



读写分离

广义上来说，读写分离这个名词有两个不同的含义：

- 像数据库一样主写从读，分摊读压力，牺牲延迟可靠性更高，适用于消息读写比非常高的场景。
- 存储写入将消息暂存至 DirectByteBuffer，当数据成功写入后，再归还给缓冲池，将写入 page cache 的动作异步化。



这里主要来讨论第二点，当 Broker 配置异步持久化且开启缓冲池，启用的异步刷盘线程是 CommitRealTimeService。我们知道操作系统本身一般是当 page cache 上积累了大量脏页后才会触发一次 flush 动作（由一些 vm 参数控制，比如 dirty_background_ratio 和 dirty_ratio）。这里有一个很有意思的说法是

CPU 的 cache 是由硬件维护一致性，而 page cache 需要由软件来维护，也被称为 syncable。高并发下写入 page cache 可能会造成刷脏页时磁盘压力较高，导致写入时出现毛刺现象。为了解决这个问题，出现了读写分离的实现，使用堆外内存将消息 hold 住，然后进行异步批量写入。

RocketMQ 启动时会默认初始化 5 块（参数 transientStorePoolSize 决定）堆外内存（DirectByteBuffer）循环利用，由于复用堆外内存，这个小方案也被成为池化，池化的好处及弊端如下：

- 好处：数据写堆外后便很快返回，减少了用户态与内核态的切换开销。
- 弊端：数据可靠性降为最低级别，进程重启就会丢数据（当然这里一般配合多副本机制进行保障），也会增加一些端到端的延迟。

宕机与故障恢复

宕机一般是由于底层的硬件问题导致，RocketMQ 宕机后如果磁盘没有永久故障，一般只需要原地重启，Broker 首先会进行存储状态的恢复，加载 CommitLog，ConsumeQueue 到内存，完成 HA 协商，最后初始化 Netty Server 提供服务。目前的实现是最后初始化对用户可见的网络层服务，实际上这里也可以先初始化网络库，分批将 Topic 注册到 NameServer，这样正常升级时可以对用户的影响更小。

在 recover 的过程中还有很多软件工程实现上的细节，比如从块设备加载的时候需要校验消息的 crc 看是否产生错误，对最后一小段未确认的消息进行 dispatch 等操作。默认从倒数第三个文件 recover CommitLog 加载消息到 page cache (假设未持久化的数据 < 3G)，防止一上线由于客户端请求的消息不在内存，导致疯狂的缺页中断阻塞线程。分布式场景下还需要对存储的数据维护一致性，这也就涉及到日志的截断，同步和回发等问题，后续我将在高可用篇再具体讨论这一点。

文件的生命周期

聊完了消息的生产保存，再来讨论下消息的生命周期，只要磁盘没有满，消息可以长期保存。前面提到 RocketMQ 将消息混合保存在 CommitLog，对于消息和流这样近似 FIFO 的系统来说，越近期的消息价值越高，所以默认以滚动的形式从前向后删除最久远的消息，而不会关注文件上的消息是否全部被消费。触发文件清除操作的是一个定时任务，默认每 10s 执行一次。在一次定时任务触发时，可能会有多个物理文件超过过期时间可被删除，因此删除一个文件不但要判断这个文件是否还被使用，还需要间隔一定时间（参数 deletePhysicFilesInterval）再删除另外一个文件，由于删除文件是一个非常耗费 IO 的操作，可能会引起存储抖动，导致新消息写入和消费的延迟。所以又新增了一个定时删除的能力，使用 deleteWhen 配置操作时间（默认是凌晨4点）。我们把由于磁盘空间不足导致的删除称为被动行为，由于高速介质通常比较贵（傲腾 ESSD等），出于成本考虑，我们还会异步的主动的将热数据转移到二级介质上。在一些特殊的场景下，删除的同时可能还需要对磁盘做安全擦除来防止数据恢复。

避免存储抖动

快速失败

消息被服务端 Netty 的 IO 线程读取后就会进入到阻塞队列中排队，而单个 Broker 节点有时会因为 GC，IO 抖动等因素造成短时存储写失败。如果请求来不及处理，排队的请求就会越积越多导致 OOM，客户端视角看从发送到收到服务端响应的时间大大延长，最终发送超时。RocketMQ 为了缓解这种抖动问题，引入了快速失败机制，即开启一个扫描线程，不断的去检查队列中的第一个排队节点，如果该节点的排队时间已经超过了 200ms，就会拿出这个请求，立即向客户端返回失败，客户端会重试到其他副本组（客户端还有一些熔断与隔离机制），实现整体服务的高可用。

存储系统不止是被动的感知一些下层原因导致的失败，RocketMQ 还设计了很多简单有效的算法来进行主动估算。例如消息写入时 RocketMQ 想要判断操作系统的 page cache 是否繁忙，但是 JVM 本身没有提供这样的 Monitor 工具来评估 page cache 繁忙程度，于是利用系统的处理时间来判断写入是否超过1秒，如果超时的话，让新请求会快速失败。再比如客户端消费时会判断当前主的内存使用率比较高，大于物理内存的 40%时，就会建议客户端从备机拉取消息。

预分配与文件预热

为了在 CommitLog 写满之后快速的切换物理文件，后台使用一个后台线程异步创建新的文件并进行对进行内存锁定，还大费周章的设计了一个额外文件预热开关（配置 warmMapedFileEnable），这么做主要有两个原因：
请求分配内存并进行 mlock 系统调用后并不一定会为进程完全锁定这些物理内存，此时的内存分页可能是写时复制的。此时需要向每个内存页中写入一些假的值，有些固态的主控可能会对数据压缩，所以这里不会写入 0。

调用 mmap 进行映射后，OS 只是建立虚拟内存地址至物理地址的映射表，而实际并没有加载任何文件至内存中。这里可能会有大量缺页中断。RocketMQ 在做 mmap 内存映射的同时进行 madvise 调用，同时向 OS 表明 WILLNEED 的意愿。使 OS 做一次内存映射后对应的文件数据尽可能多的预加载至内存中，从而达到内存预热的效果。

当然，这么做也是有弊端的。预热后，写文件的耗时缩短了很多，但预热本身就会带来一些写放大。整体来看，这么做能在一定程度上提高响应时间的稳定性，减少毛刺现象，但在 IO 本身压力很高的情况下则不建议开启。RocketMQ 是适用于 Topic 数量较多的业务消息场景。所以 RocketMQ 采用了和 Kafka 不一样的零拷贝方案，Kafka 采用的是阻塞式 IO 进行 sendfile，适用于系统日志消息这种高吞吐量的大块文件。而 RocketMQ 选择了 mmap + write 非阻塞式 IO (基于多路复用) 作为零拷贝方式，这是因为 RocketMQ 定位于业务级消息这种小数据块/高频率的 IO 传输，当想要更低的延迟的时候选择 mmap 更合适。

当 kernal 把可用的内存分配后 free 的内存就不够了，如果进程一下产生大量的新分配需求或者缺页中断，还需要将通过淘汰算法进行内存回收，此时可能会产生抖动，写入会有短时的毛刺现象。

冷数据读取

对于 RocketMQ 来说，读取冷数据可能有两种情况。

- 请求来自于这个副本组的其他节点，进行副本组内的数据复制，也可能是离线转储到其他系统。
- 请求来自于客户端，是消费者来消费几个小时以前的数据，属于正常的业务诉求。

对于第一种情况，在 RocketMQ 低版本源码中，对于需要大量复制 CommitLog 的情况（例如备磁盘故障，或新上线一个备机），主默认使用 DMA 拷贝的形式将数据直接通过网络复制给备机，此时由于大量的缺页中断阻塞了 io 线程，此时会影响 Netty 处理新的请求，在实现上让一些组件之间的内部通信使用 fastRemoting 提供的第二个端口，解决这个问题的临时方案还包括先用业务线程将数据 load 回内存而不使用零拷贝，但这个做法没有从本质上解决阻塞的问题。对于冷拷贝的情况，可以使用 madvice 建议 os 读取避免影响主的消息写入，也可以从其他备复制数据。

对于第二种情况，对各个存储产品来说都是一个挑战，客户端消费一条消息时，热数据全部存储在 page cache，对于冷数据会退化为随机读（系统会有一个对 page cache 连续读的预测机制）。需要消费超过几个小时之前的数据的场景下，消费者一般都是做数据分析或者离线任务，此时下游的目标都是吞吐量优先而非延迟。对于 RocketMQ 来说有两个比较好的解决方案，第一是同 redirect 的方式将读取请求转发给备进行分摊读压力，或者是从转储后的二级介质读取。在数据转储后，RocketMQ 本身的数据存储格式会发生变化，详见后文。

索引数据管理

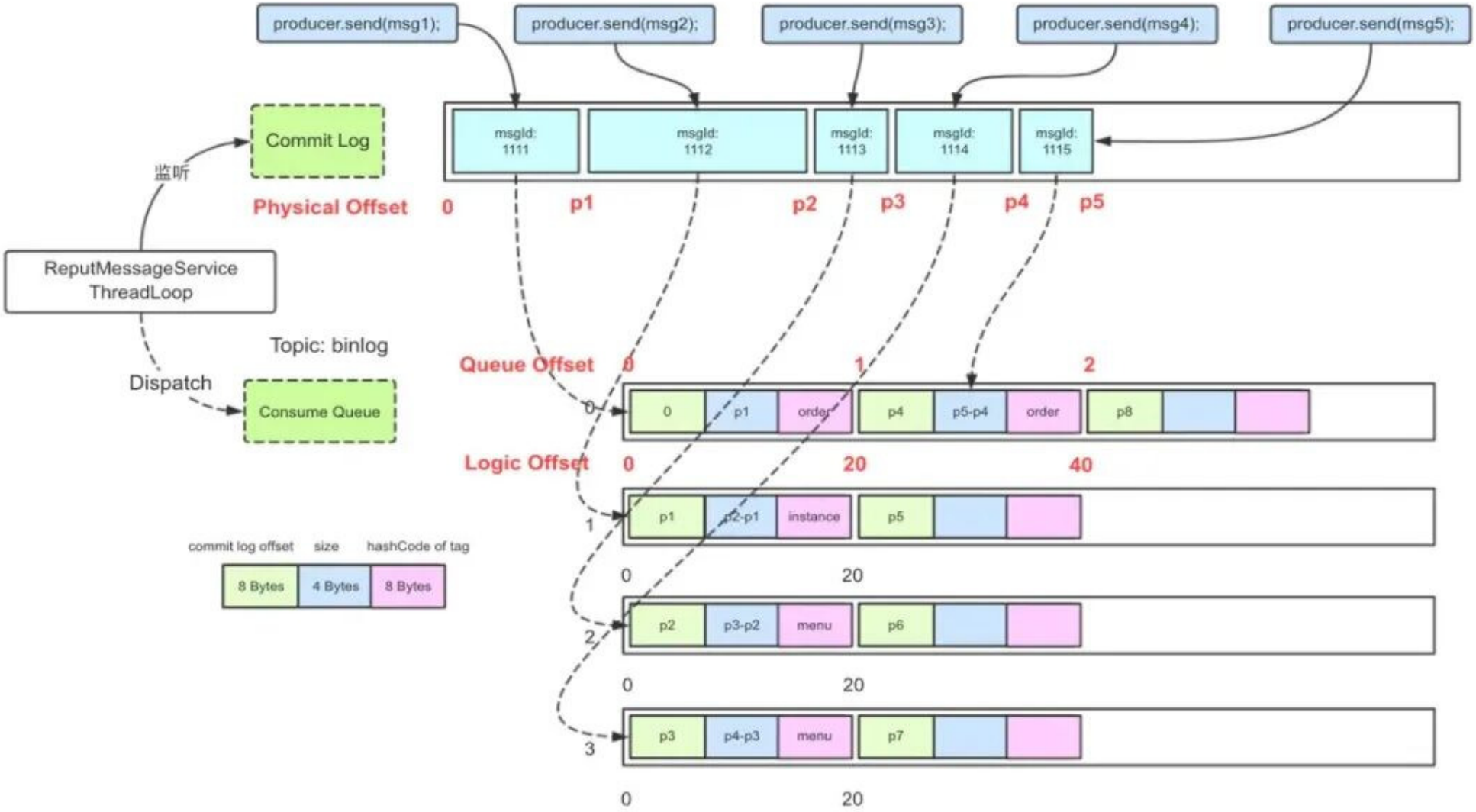
在数据写入 CommitLog 后，在服务端当 MessageStore 向 CommitLog 写入一些消息后，有一个后端的 ReputMessageService 服务 (dispatch 线程) 会异步的构建多种索引，满足不同形式的读取诉求。

队列维度的有序索引 ConsumeQueue

在 RocketMQ 的模型下，消息本身存在的逻辑队列称为 MessageQueue，而对应的物理索引文件称为 ConsumeQueue。从某种意义上说 MessageQueue = 多个连续 ConsumeQueue 索引 + CommitLog 文件。

ConsumeQueue 相对与 CommitLog 来说是一个更加轻量。dispatch 线程会源源不断的将消息从 CommitLog 取出，再拿出消息在 CommitLog 中的物理偏移量 (相对于文件存储的 Index)，消息长度以及Tag Hash 作为单条消息的索引，分发到对应的消费队列。偏移 + 长度构成了对 CommitLog 的引用 (Ref)。这种 Ref 机制对于单挑消息只有 20B，显著降低了索引存储开销。ConsumeQueue 实际写入的实现与 CommitLog 不同，CommitLog 有很多存储策略可以选择

且混合存储，一个 ConsumeQueue 只会保存一个 Topic 的一个分区的索引，持久化默认使用 FileChannel，实际上这里使用 mmap 的话对小数据量的请求更加友好，不用陷入中断。



客户端的 pull 请求到服务端执行了如下流程来查询消息：

1. 根据 Tag 的 Hash 值查询 ConsumeQueue 文件（由 physicOffset + size + Tag HashCode 组成）

2. 根据 ConsumeQueue 拿到 physicOffset + size
3. 根据 physicOffset 查询 CommitLog 文件（上文的MappedFileQueue）获得消息

RocketMQ 中默认指定每个消费队列的文件存储 30 万条索引，而一个索引占用 20 个字节，这样每个文件的大小是 300 _1000 _20 / 1024 / 1024 ≈ 5.72M。为什么消费队列文件存储消息的个数要设置成 30 万呢？这个经验值适合消息量比较大的场景，事实上这个值对于大部分场景来说是偏大的，有效数据的真实占用率很低，导致ConsumeQueue 空载率高。

先来看看如果过大或者过小会带来什么问题。因为消息总是有失效期的，例如 3 天失效，如果消费队列的文件设置过大的话，有可能一个文件中包含了过去一个月的消息索引，但这个时候原始的数据已经滚动没了，白白浪费了很多空间。但也不宜太小，导致 ConsumeQueue 也有大量小文件，降低读写性能。

下面给出一个非严谨的空载率推导过程：假设此时单机的 Topic = 5000，单节点单个 Topic 的队列数一般是 8，分区数量 = 4万。以 1T 消息数据为例，每条消息大小是 4KB，索引数量 = 消息数量 = 1024 1024 * 1024 / 4 = 2.68 亿。最少需要的 ConsumeQueue = 索引数量 / 30万 = 895 个，实际使用率 (有效数据量) 约等于 2.4%。随着 ConsumeQueue Offset 的原子自增滚动，cq 头部是无效数据导致占用的磁盘空间会变大。根据公有云线上的情况来看，非 0 数据约占 5%，实际有效数据只占 1%。对于 ConsumeQueue 这样的索引文件，我们可以使用 RocksDB 或者傲腾这样的持久化内存来存储，或者对 ConsumeQueue 单独实现一个用户态文件系统，几个方案都可以减少整体索引文件大小，提高访问性能。这一点在后文关于存储机制的优化中，我们再详聊。

由于 CommitLog - ConsumerQueue - Offset 的关系从消息写入的那一刻开始就确定了，在 Topic 跨副本组迁移，副本组要下线等需要切流的场景下，如果需要消息可读，需要采用复制数据的方案来实现 Topic 跨副本组迁移，只能采用消息级别的拷贝，而不能简单的把一个分区从副本组 A 移动到副本组 B。有一些消息产品在面对这个场景时，采用了数据按分区复制的方案，这种方案可能会立刻产生大量的数据传输（分区 rebalance），而 RocketMQ 的切流一般可以做到秒级生效。

消息维度的随机索引 IndexFile

RocketMQ 作为业务消息的首选，上文中 ReputMessageService 线程除了构建消费队列的索引外，还同时为每条消息根据 id, key 构建了索引到 IndexFile。这是方便快速快速定位目标消息而产生的，当然这个构建随机索引的能力是可以降级的，IndexFile文件结构如下：

IndexFile 也是定长的，从单个文件的数据结构来说，这是实现了一种简单原生的哈希拉链机制。当一条新的消息索引进来时，首先使用 hash 算法命中黄色部分 500w 个 slot 中的一个，如果存在冲突就使用拉链解决，将最新索引数据的 next 指向上一条索引位置。同时将消息的索引数据 append 至文件尾部（绿色部分），这样便形成了一条当前 slot 按照时间存入的倒序的链表。这里其实也是一种 LSM compaction 在消息模型下的改进，降低了写放大。

存储机制的演进方向

RocketMQ 的存储设计是以简单可靠队列模型作为核心来抽象的，也因此产生了一些缺陷和对应的优化方案。

KV 模型与 Queue 模型结合

RocketMQ 实现了单条业务消息的退避重试，在生产实践中，我们发现部分用户在客户端消费限流时直接将消息返回失败，在重试消息量比较大的时候，由于原有实现下重试队列数有限，导致重试消息无法很好的负载均衡到所有客户端。同时，消息来回的在服务端和客户端之间传输，使得两侧的开销都增加了，用户侧正确的做法应该是消费限流时，让消费的线程等待一会儿。从存储服务的角度上来说，这其实是一种队列模型的不足，让一条队列只能被一个消费者持有。RocketMQ 提出了 pop 消费这种全新的概念，让单条队列的消息能够被多个客户端消费到，这涉及到服务端对单条消息的加解锁，KV 模型就非常契合这个场景。从长远来看，像定时消息事务消息可以有一些基于 KV 的更原生的实现，这也是 RocketMQ 未来努力的方向之一。

消息的压缩与归档存储

压缩就是用时间去换空间的经典 trade-off，希望以较小的 CPU 开销带来更少的磁盘占用或更少的网络 I/O 传输。目前 RocketMQ 客户端从延迟考虑仅单条大于 4K 的消息进行单条压缩存储的。服务端对于收到的消息没有立刻进行压缩存储有多个原因，例如为了保证数据能够及时的写入磁盘，消息稀疏的时候攒批效果比较差等，所以 Body 没有压缩存储。而对于大部分的业务 Topic 来说，其实 Body 一般都有很大程度上是相似的，可以压缩到原来的几分之一到几十分之一。

存储一般有高速（高频）介质与低速介质，热数据存放在高频介质上（如傲腾，ESSD，SSD），冷数据存放在低频介质上（NAS，OSS），以此来满足低成本保存更久的数据。从高频介质转到更低频的 NAS 或者 OSS 时，不可避免的产生了一次数据拷贝。我们可以在这个过程中异步的对数据进行规整（闲时资源富余）。那么我们为什么要做规整呢，直接零拷贝复制不香吗？答案就是低频介质虽然便宜大碗，但通常 iops 和吞吐量更低。对于 RocketMQ 来说需要规整的数据就是索引和 CommitLog 中的消息，也就是说在高频介质与低频介质上消息的存储格式可以是完全不同的。当热消息降级到二级存储的时候，数据密集且异步，这里就是一个非常合适的机会进行压缩和规整。业界也有一些基于 FPGA 来加速存储压缩的案例，将来我们也会持续的做这方面的尝试。

存储层资源共享与争抢

磁盘 IO 的抢占

没错，这里想谈谈的其实是硬盘的调度算法。在一个考虑性价比的场景下，由于 RocketMQ 的存储机制，我们可以把索引文件存储在 SSD，消息本身放在 HDD 里，因为热消息总是在 PageCache 中的，所以在 IO 调度上优先满足写而饿死读。对于没有堆积的消费者来说，消费到的数据是从 page cache 拷贝到 socket 再传输给用户，实时性已经很高了。而对于消费冷数据（几个小时，几天以前的数据）用户的诉求一般是尽快获取到消息即可，此时服务端可以选择尽快满足用户的 Pull 请求，由于大量的随机 IO，这样磁盘会产生严重的 rt 抖动。仔细考虑，这里其实用户想要的是尽可能大的吞吐量，假设访问冷数据需要 200 毫秒，假设在服务端把冷读的行为滞后，再加上延迟 500 毫秒再返回给用户数据，并没有显著的区别。而这里的 500 毫秒，服务端内部就可以合并大量的 IO 操作，我们也可以使用 madvice 系统调用去建议内核读取。这里的合并带来的收益很高，可以显著的减少对热数据的写入的影响，大幅度提升性能。

用户态文件系统

还是为了解决随机读效率低的问题，我们可以设计一个用户态文件系统，让 IO 调用全部 kernel-bypass。

主要有几个方向：

1. 多点挂载。常用的 Ext4 等文件系统不支持多点挂载，让存储能够支持多个实例的对同一份数据的共享访问。
2. 调整对于 IO 的合并策略，IO优先级，polling 模式，队列深度等。
3. 使用文件系统类似 O_DIRECT 的非缓存方式读写数据。

RocketMQ 的未来

RocketMQ 存储系统经过多年的发展，基本功能特性已经比较完善，通过一系列的创新技术解决了分布式存储系统中的难题，稳定的服务于阿里集团和海量的云上用户。RocketMQ 在云原生时代的演进中遇到了更多的有趣的场景和挑战，这是一个需要全链路调优的复杂工程。我们会持续在规模，稳定性，多活容灾等企业级特性，成本与弹性等方面发力，将 RocketMQ 打造为“消息，事件，流”一体化的融合平台。同时，我们也会将开源行动更加可持续的发展下去，为社会创造价值。

参考文献 [1]. 深入理解 Linux 中的page cache https://www.jianshu.com/p/ae741edd682c [2]. PacificA: Replication in Log-Based Distributed Storage Systems. https://www.microsoft.com/en-us/research/wp-content/uploads/2008/02/tr-2008-25.pdf [3]. J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. B. Zdonik. Anti-caching: A new approach to database management system architecture. PVLDB, 6(14):1942–1953, 2013. [4]. 《RocketMQ 技术内幕》 [5]. 一致性协议中的“幽灵复现”. [https://zhuanlan.zhihu.com/p/47025699.](https://zhuanlan.zhihu.com/p/47025699. [6]. Calder B, Wang J, Ogus A, et al. Windows Azure Storage: a highly available cloud storage service with strong consistency[C]//Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. ACM, 2011: 143-157. [7]. Chen Z, Cong G, Aref W G. STAR: A distributed stream warehouse system for spatial data[C] 2020: 2761-2764. [8]. design data-intensive application 《构建数据密集型应用》 [9]https://github.com/apache/rocketmq/blob/5.0.0-alpha/docs/cn/statictopic/RocketMQ_Static_Topic_Logic_Queue_设计.md