RocketMQ封神之旅（二）-核心原理

原创

gonghaiyu 于 2021-06-09 23:44:46 发布 59 收藏

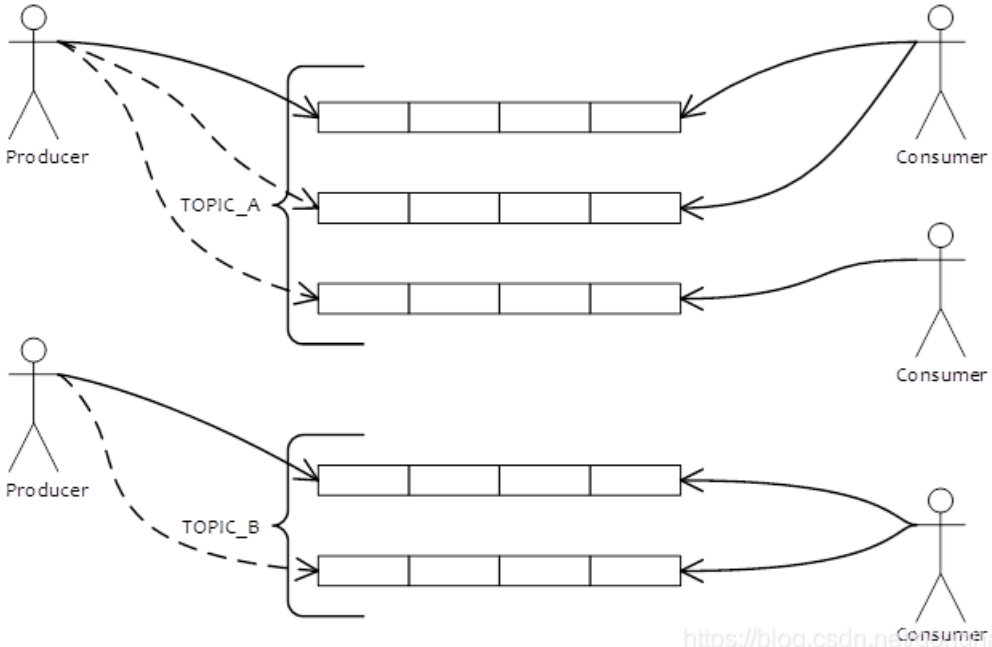
分类专栏： RocketMQ

本文链接：https://blog.csdn.net/gonghaiyu/article/details/117755653

> 本文从常见的物理部署结构开始，从启动、MQ接收、MQ消费的角度理解MQ的整个运行过程。本文很多资料来自于RocketMQ官网，感谢大佬们对开源的支持。

RocketMQ Overview

RocketMQ是什么？



1. □ 是一个队列模型的消息中间件，具有高性能、高可靠、高实时、分布式特点。
2. □ Producer、Consumer、队列都可以分布式。
3. □ Producer 向一些队列轮流发送消息，队列集合称为 Topic，Consumer如果做广播消费，则一个consumer实例消费这个Topic对应的所有队列，如果做集群消费，则多个Consumer实例平均消费这个topic对应的队列集合。
4. □ 能够保证严格的消息顺序
5. □ 提供丰富的消息拉取模式
6. □ 高效的订阅者水平扩展能力
7. □ 实时的消息订阅机制
8. □ 亿级消息堆积能力
9. □ 较少的依赖

从上面的9个特点中说明，用MQ准没错。

RocketMQ核心组件

保证消息至少被消费一次，但不承诺消息不会被消费者多次消费，其消费的幂等由消费者实现，从而极大地简化了其实现内核，提高了RocketMQ的整体性能。

NameServer

NameServer设计极其简单，摒弃了业界常用的使用Zookeeper充当信息管理的“注册中心”，而是自研NameServer来实现元数据的管理（Topic路由信息等）。从实际需求出发，因为Topic路由信息无须在集群之间保持强一致，追求最终一致性，并且能容忍分钟级的不一致。正是基于此种情况，RocketMQ的NameServer集群之间互不通信，极大地降低了NameServer实现的复杂程度，对网络的要求也降低了不少，但是性能相比较Zookeeper有了极大的提升。

Broker

其次是高效的IO存储机制。RocketMQ追求消息发送的高吞吐量，RocketMQ的消息存储文件设计成文件组的概念，组内单个文件大小固定，方便引入内存映射机制，所有主题的消息存储基于顺序写，极大地提供了消息写性能，同时为了兼顾消息消费与消息查找，引入了消息消费队列文件与索引文件。

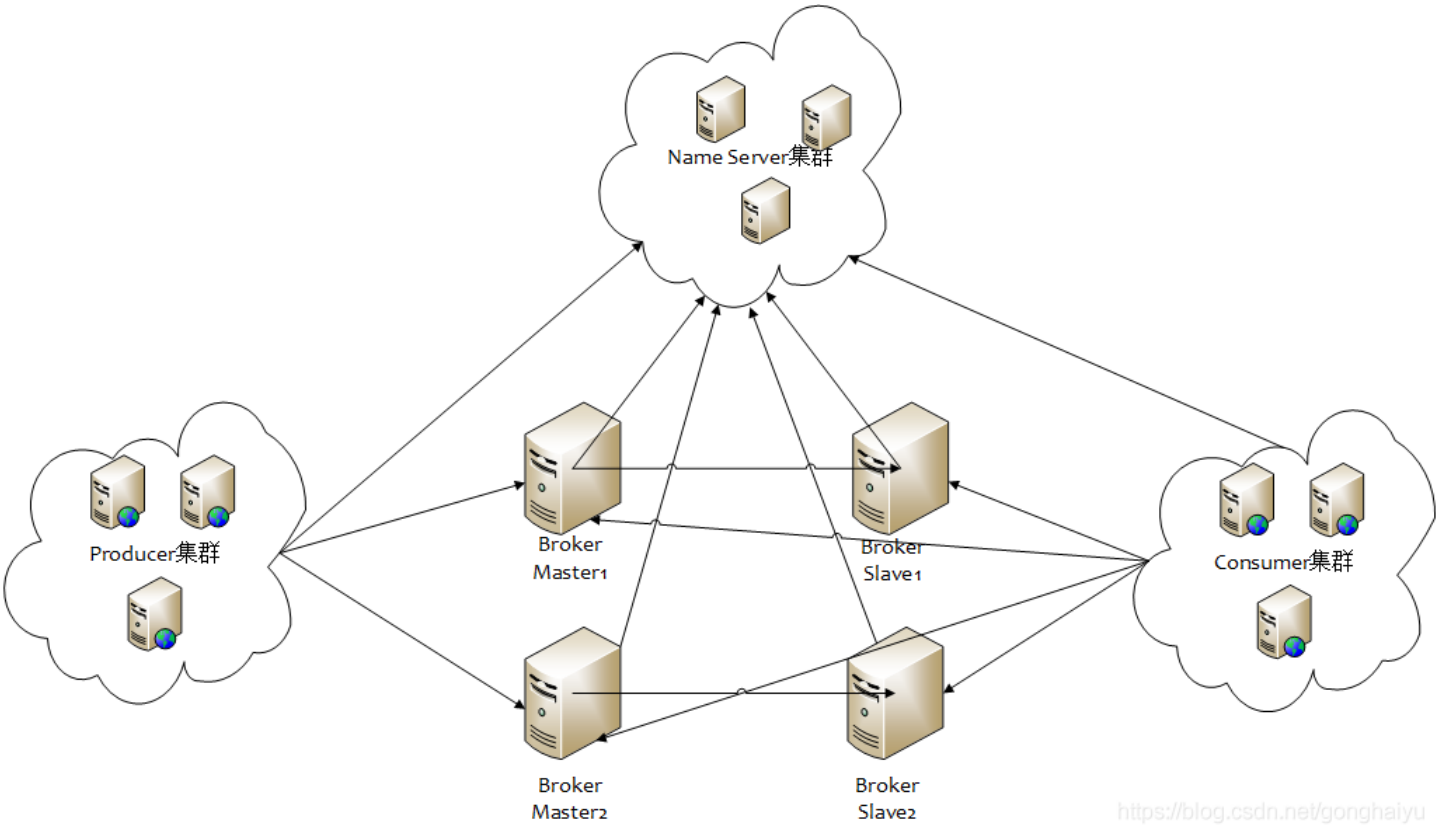
最后是容忍存在设计缺陷，适当将某些工作下放给RocketMQ使用者。消息中间件的实现者经常会遇到一个难题：如何保证消息一定能被消息消费者消费，并且保证只消费一次。RocketMQ的设计者给出的解决办法是不解决这个难题，而是退而求其次，只保证消息被消费者消费，但设计上允许消息被重复消费，这样极大地简化了消息中间件的内核，使得实现消息发送高可用变得非常简单与高效，消息重复问题由消费者在消息消费时实现幂等。

Producer

Consumer

RocketMQ物理部署结构

下面先开始部署一套集群玩玩，先说明下该集群的物理结构。



RocketMQ 网络部署特点

□ Name Server 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。

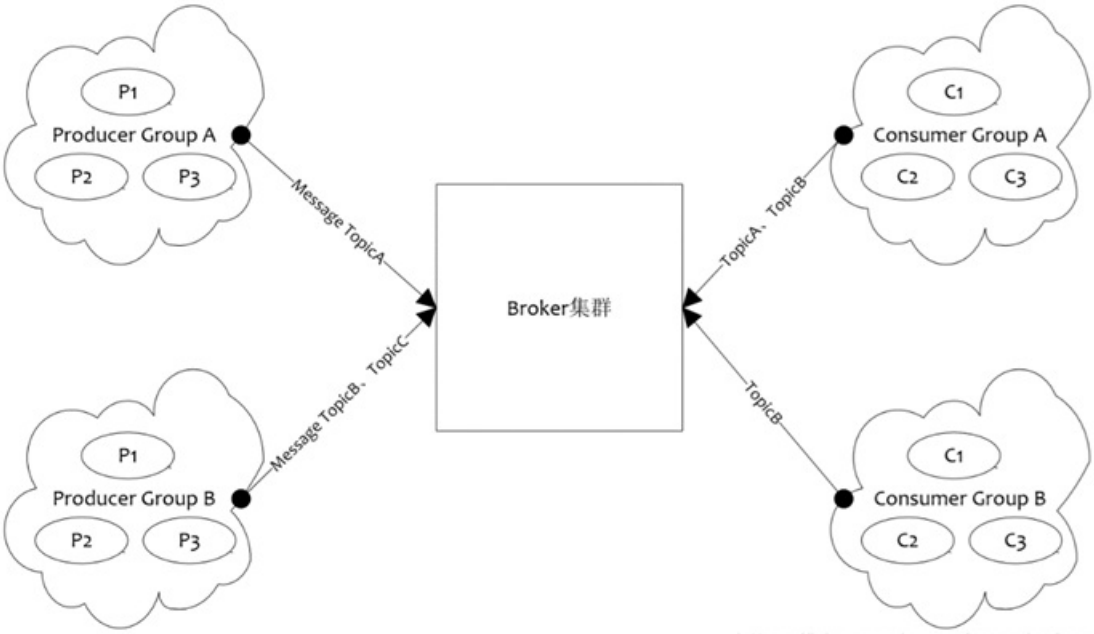
□ Broker 部署相对复杂，Broker 分为 Master 与 Slave，一个 Master 可以对应多个 Slave，但是一个 Slave 只能对应一个 Master，Master 与 Slave 的对应关系通过指定相同的 BrokerName，不同的 BrokerId 来定义，BrokerId为 0 表示 Master，非 0 表示 Slave。Master 也可以部署多个。每个 Broker 与 Name Server 集群中的所有节点建立长连接，定时注册 Topic 信息到所有 Name Server。

□ Producer 与 Name Server 集群中的其中一个节点（随机选择）建立长连接，定期从 Name Server 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 Name Server 集群中的其中一个节点（随机选择）建立长连接，定期从 Name Server 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，订阅规则由 Broker 配置决定。

> 以上说明了一个Master可对应一个或多个Slave。所有Brocker与NameServer建立长连接，定期向Name Server注册Topic。
> 运行过程：Producer和Consumer与NameServer建立长连接定期获取Topic。Producer与Master也建立长连接，用来发送心跳。Consumer与Master和Slave都建立长连接并定时发送心跳。Consumer的订阅规则由Broker配置决定。

RocketMQ 逻辑部署结构



图表 5-3RocketMQ 逻辑部署结构

Producer Group

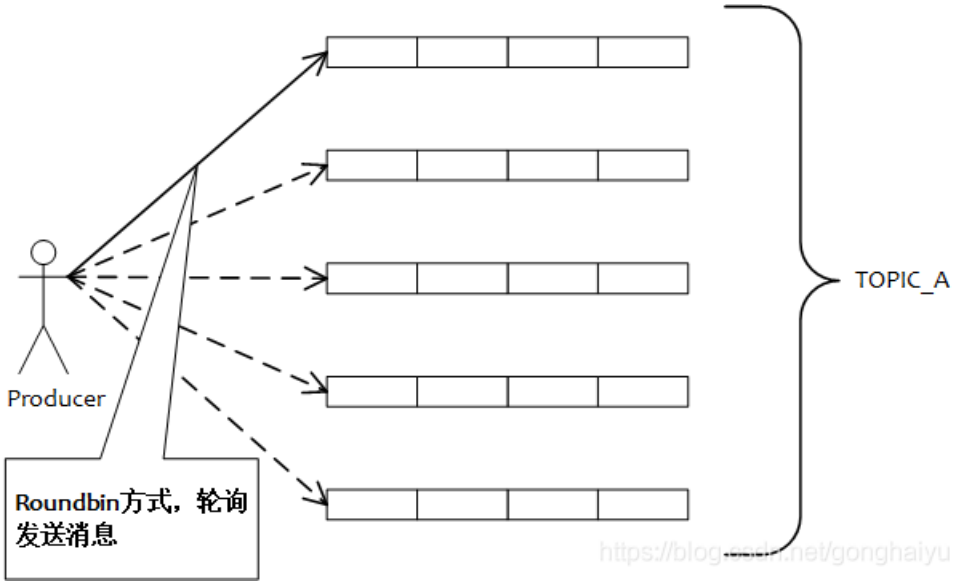
用来表示一个发送消息应用，一个 Producer Group 下包含多个 Producer 实例，可以是多台机器，也可以是一台机器的多个进程，或者一个进程的多个 Producer 对象。一个 Producer Group 可以发送多个 Topic消息，Producer Group 作用如下：

1. 标识一类 Producer
2. 可以通过运维工具查询这个发送消息应用下有多个 Producer 实例
3. 发送分布式事务消息时，如果 Producer 中途意外宕机，Broker 会主动回调 Producer Group 内的任意一台机器来确认事务状态。

Consumer Group

用来表示一个消费消息应用，一个 Consumer Group 下包含多个 Consumer 实例，可以是多台机器，也可以是多个进程，或者是一个进程的多个 Consumer 对象。一个 Consumer Group 下的多个 Consumer 以均摊方式消费消息，如果设置为广播方式，那么这个 Consumer Group 下的每个实例都消费全量数据。

发送消息负载均衡



如图所示，5 个队列可以部署在一台机器上，也可以分别部署在 5 台不同的机器上，发送消息通过轮询队列的方式发送，每个队列接收平均的消息量。通过增加机器，可以水平扩展队列容量。

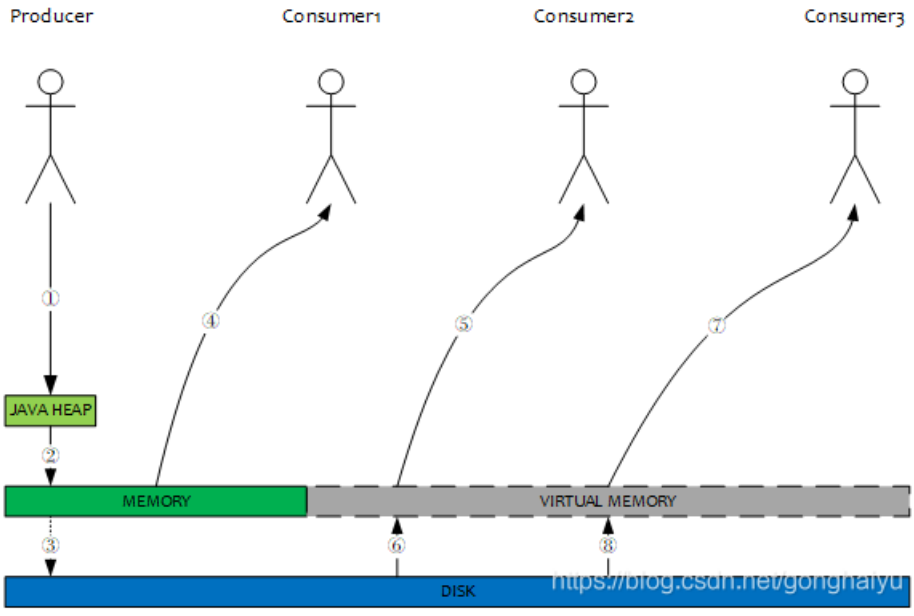
另外也可以自定义方式选择发往哪个队列。

RocketMQ数据存储结构

producer发过来的消息，该怎么存储呢？存储之前，我们有必要说下数据流。

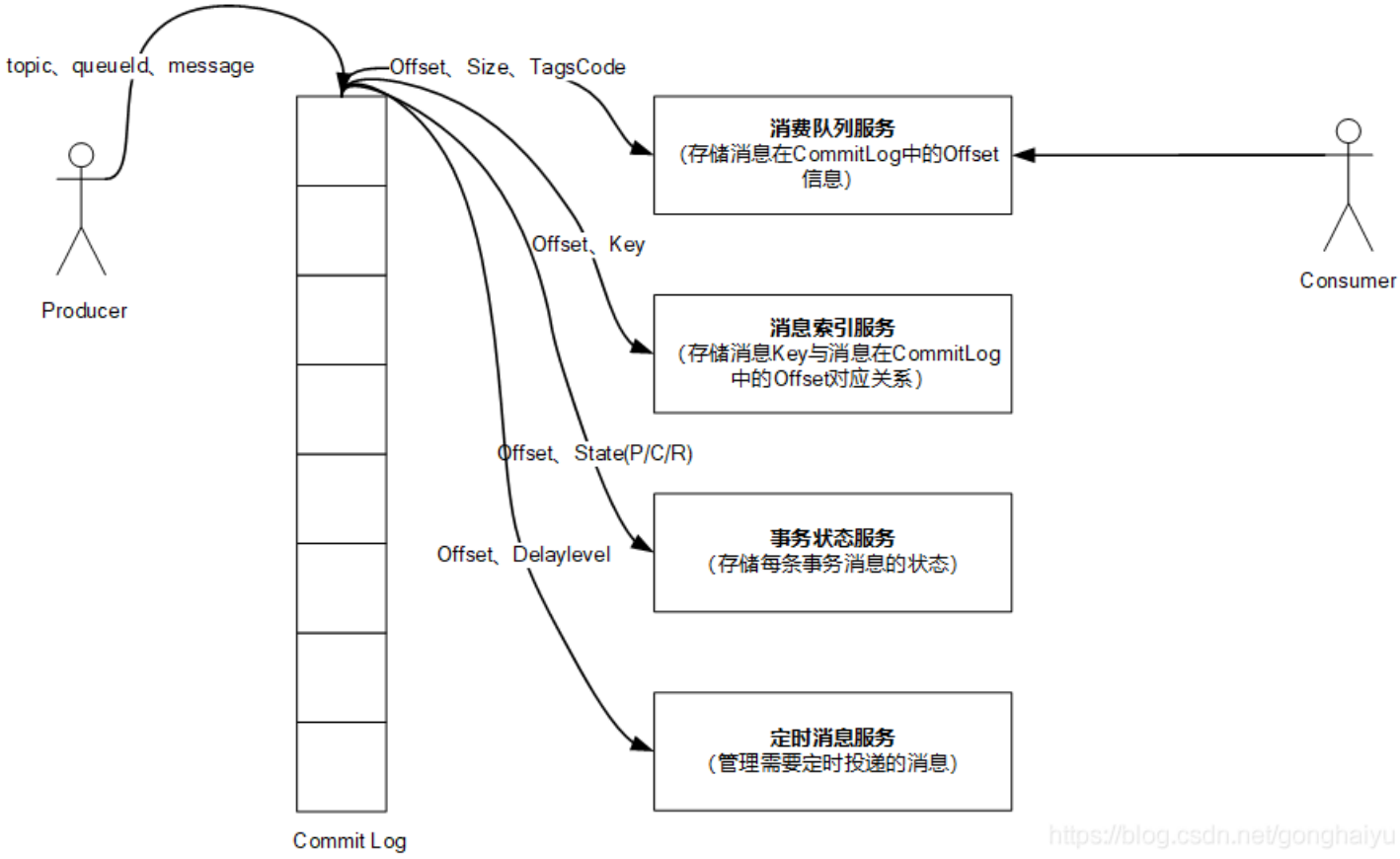
RocketMQ的数据流

下面是单个JVM进程将数据可能存储的地方。通过下面的讲解，可以看到RocketMQ能支持的并发量是很大的。

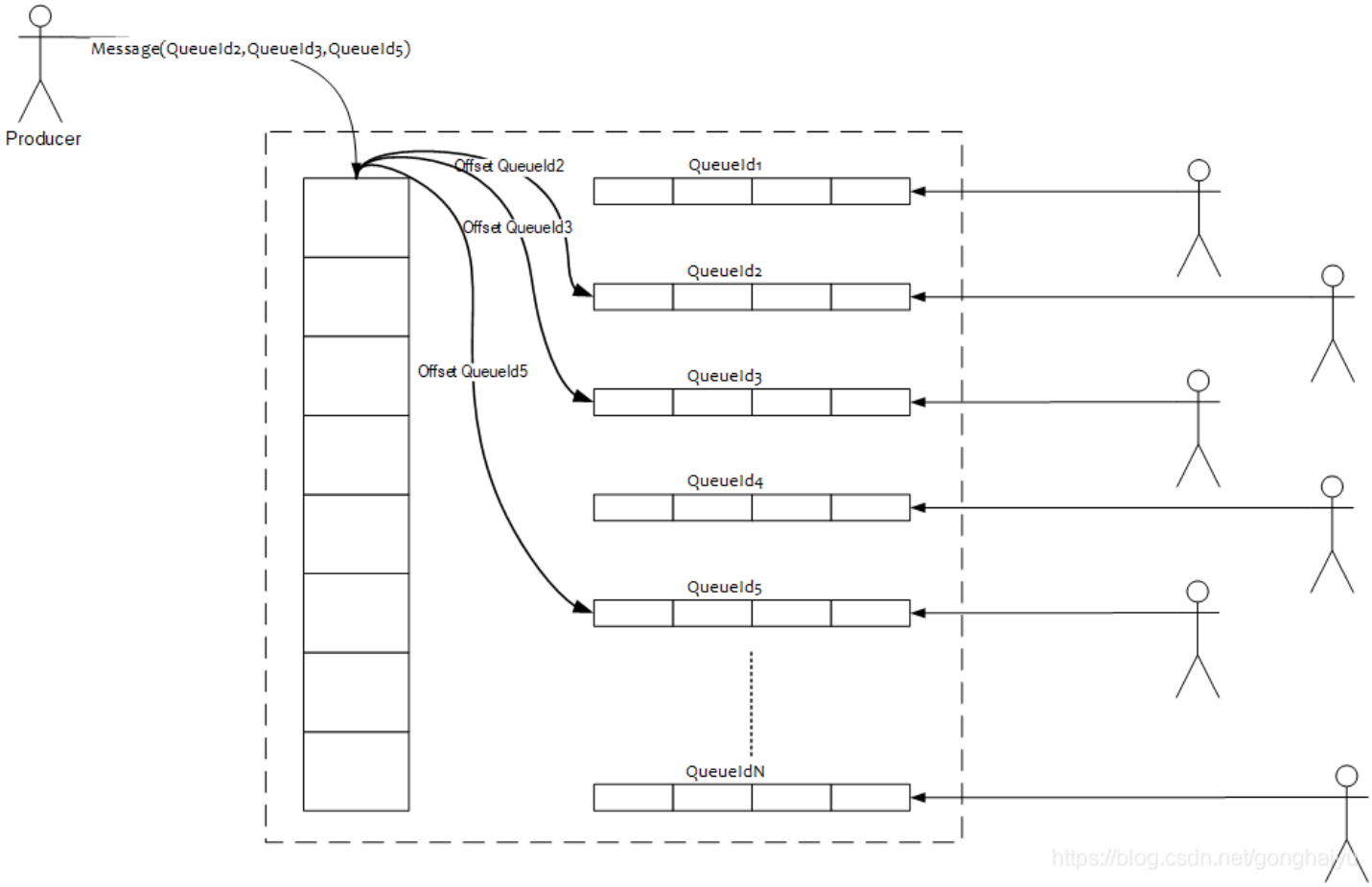


(1). Producer 发送消息，消息从 socket 进入 java 堆。
(2). Producer 发送消息，消息从 java 堆转入 PAGACACHE，物理内存。
(3). Producer 发送消息，由异步线程刷盘，消息从 PAGECACHE 刷入磁盘。
(4). Consumer 拉消息（正常消费），消息直接从 PAGECACHE（数据在物理内存）转入 socket，到达 consumer，不经过 java 堆。这种消费场景最多，线上 96G 物理内存，按照 1K 消息算，可以在物理内存缓存 1 亿条消息。
(5). Consumer 拉消息（异常消费），消息直接从 PAGECACHE（数据在虚拟内存）转入 socket。
(6). Consumer 拉消息（异常消费），由于 Socket 访问了虚拟内存，产生缺页中断，此时会产生磁盘 IO，从磁盘 Load 消息到 PAGECACHE，然后直接从 socket 发出去。
(7). 同 5 一致。
(8). 同 6 一致。

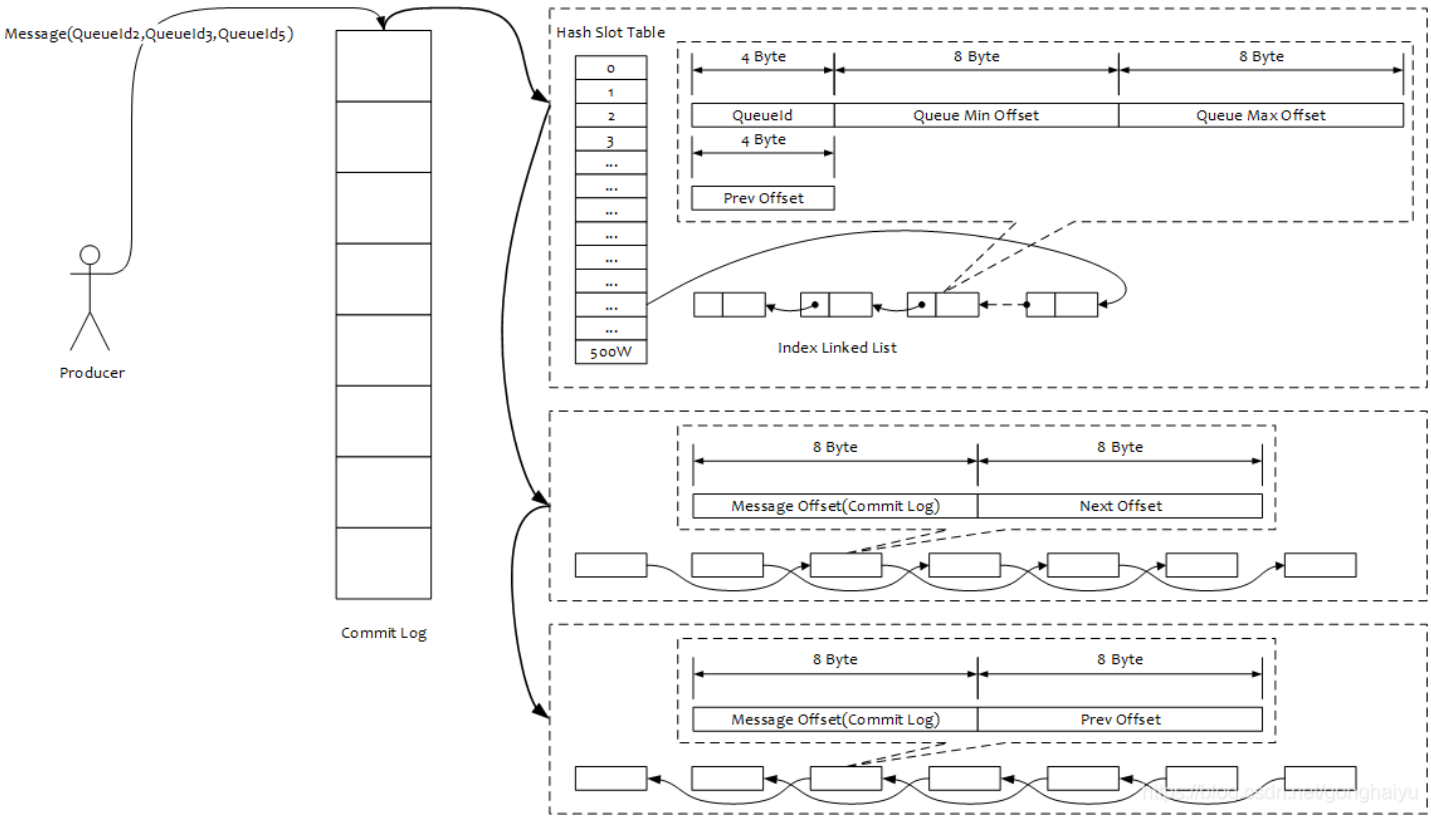
RocketMQ在内存或磁盘中的数据结构



消息队列服务（外部视图）

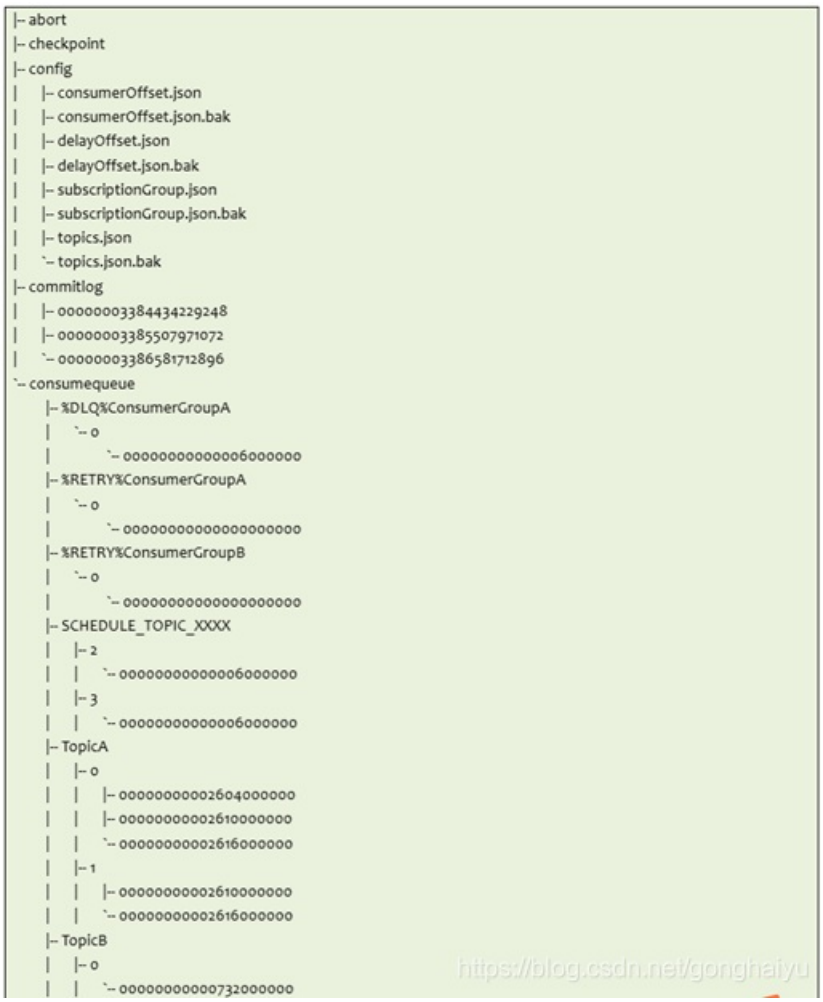


消息队列服务（内部视图）



存储目录结构

从下图可以看到有几类文件。



commitLog文件

commitLog文件存储消息的几乎所有信息，**这里要重点关注下消息体也存在此处**。存储位置：

ROCKET OME/store/commitlog/{fileName}。其中filename是文件的物理偏移量。例如，第一个文件第一条消息的物理偏移量是0，文件名是00000000000000000000，由于每个文件大小固定是1GB，所以第二个文件是00000000001073741824。

存储的数据结构如下图：

大小	4	4	8	8	8	4		1	
属性名	totlesize	queueId	queueoffset	physicaloffset	bornhost	bodyLengt	body	topiclength	topic
含义	消息大小	消息队列Id	消息队列偏移量	物理偏移量	发送者地址	消息体长度	消息体	topic长度	topic

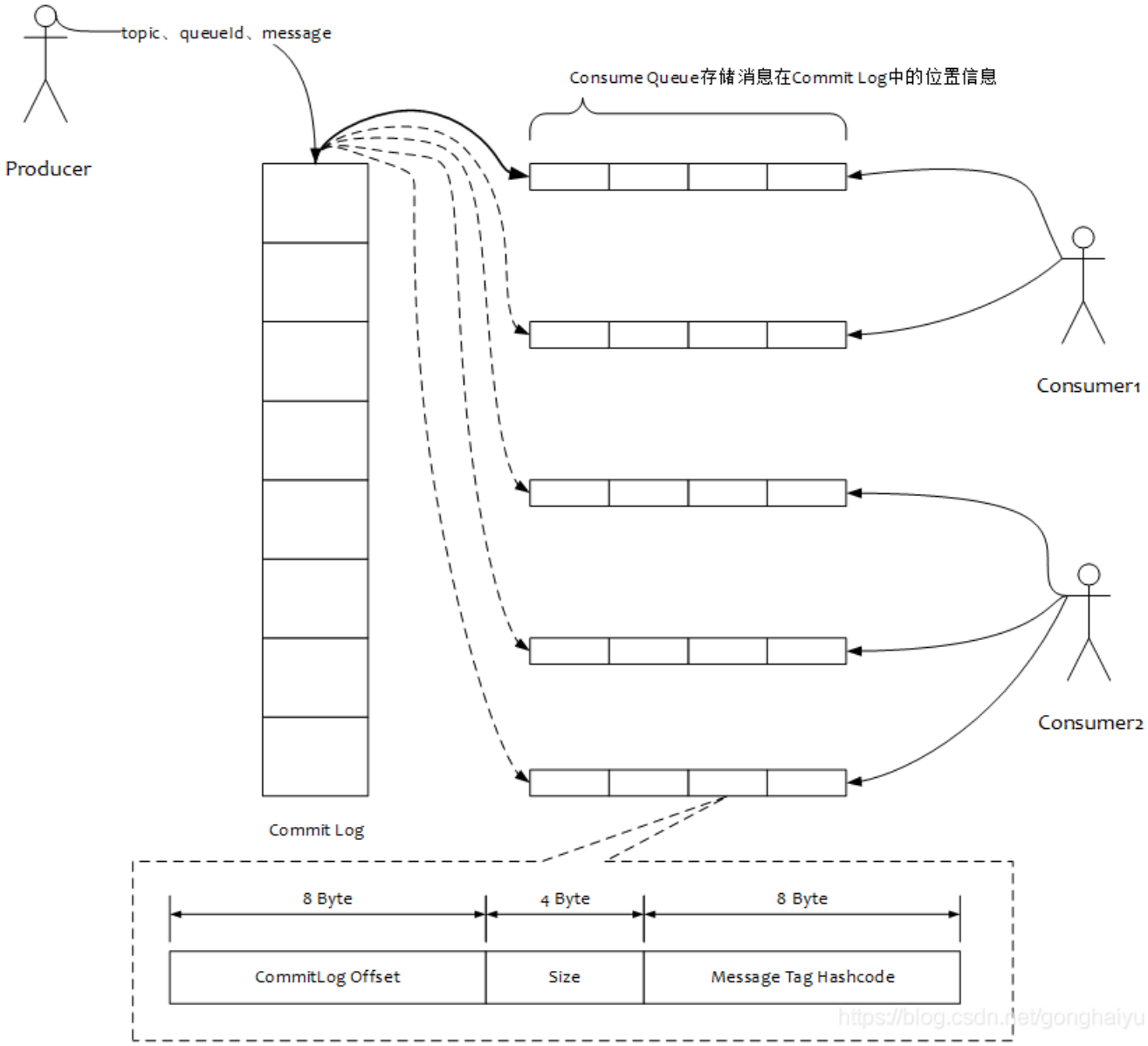
图中列出了主要的存储信息，省去了部分不那么重要的信息，例如固定值魔数。可以看出，每一条消息数据的大小并不是固定的，消息元数据占据的大小存储在totlesize中。

consumerQueue文件

ConsumeQueue中并不需要存储消息的内容，而存储的是消息在CommitLog中的offset。也就是说，ConsumeQueue其实是CommitLog的一个索引文件。存储的目录如图所示：${ROCKET_HOME }/store/consumequeue/{topic}/。文件大小是固定的30w * 20字节，其中每条消息20字节，每个文件存储30w条。

ConsumeQueue 是消息消费的逻辑队列，消息达到 CommitLog 文件后将被异步转发到消息消费队列ConsumeQueue，供消息消费者消费，这里面包含 MessageQueue 在 CommitLog 中的物理位置偏移量 Offset，消息实体内容的大小和 Message Tag 的 hash 值。每个文件默认大小约为 600W 个字节，如果文件满了后会也会生成一个新的文件。

单条数据结构如下图：



其中，commitLog offset表示本条消息在commitLog中的偏移量，size表示消息大小，message tag hashcode表示tag参数的哈希值，主要用于过滤。可以看出，如果已知topic和队列，可以根据消费者消费进度，使用consumerQueue文件顺序读取消息，然后使用获取到的偏移量从commitLog中快速查找到消息的消息体等重要信息，然后返回到消费者。

RocketMQ还可以根据消息时间戳查询消息，借助consumerQueue完成。因为consumerQueue文件是一个有序的文件，时间戳小的消息一定在时间戳大的消息的前面，所以RocketMQ借助二分查找来完成按照时间戳查找消息的操作。

很显然，Consumer消费消息的时候，要读2次：先读ConsumeQueue得到offset，再通过offset找到CommitLog对应的消息内容。

ConsumerQueue的作用：

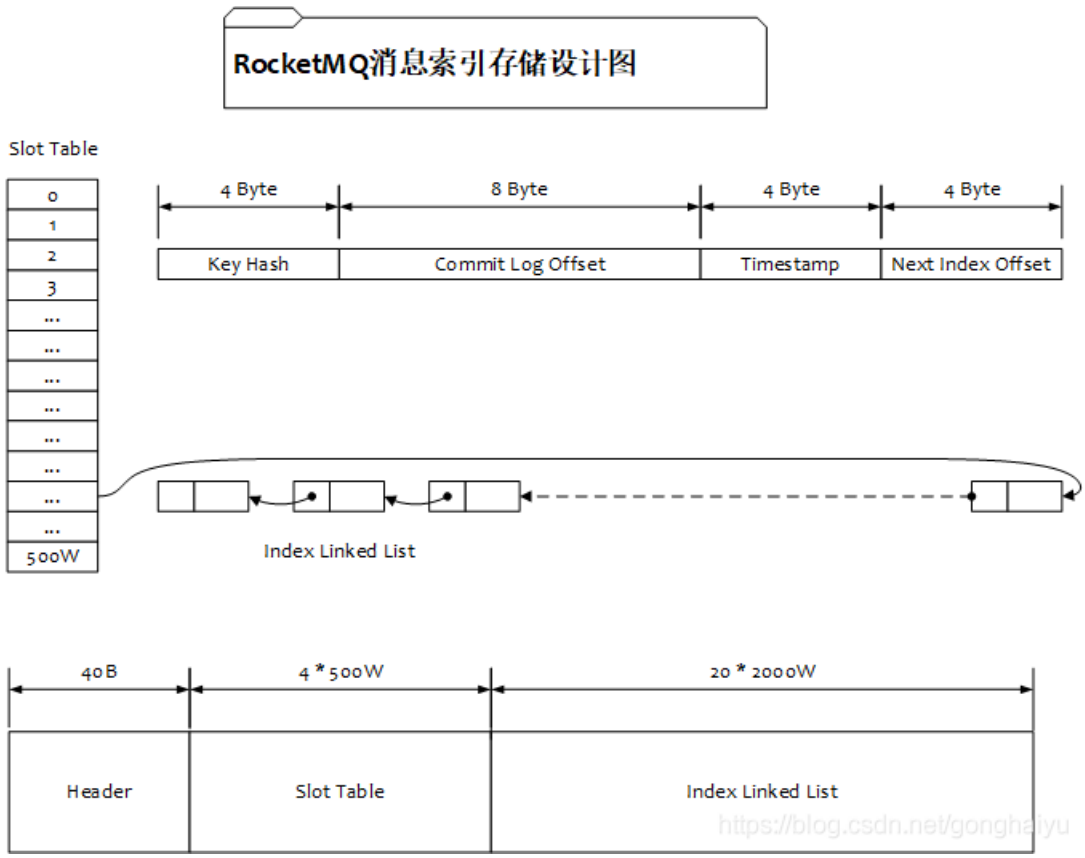
（1）通过broker保存的offset（offsetTable.offset json文件中保存的ConsumerQueue的下标）可以在ConsumeQueue中获取消息，从而快速的定位到commitLog的消息位置。

（2）过滤tag是也是通过遍历ConsumeQueue来实现的（先比较hash(tag)符合条件的再到consumer比较tag原文）。

（3）并且ConsumeQueue还能保存于操作系统的PageCache进行缓存提升检索性能。

3. IndexFile文件

RocketMQ除了可以顺序消费消息，还可以消息的key来查询，key查询借助indexFile来完成，indexFile的存储结构如下图：



indexFile文件是一个典型的hashtable实现，用链表来解决hash冲突。indexFile文件分为3部分。

第一部分header，其中包含了第一条索引的时间戳（beginTimestamp），最后一条消息的时间戳（endTimeStamp）起始物理偏移量（beginPhyoffset）结束物理偏移量（endPhyoffset）哈希条目已用个数（indexCount）

第二部分slot table，一共有500w个条目，每条大小4字节。表示Index Listed List中数据的偏移量。

第三部分是 Index Listed List，一共2000W条，存储的结构如上图。keyHash表示本条数据所对应Key的hash值，不保存key而保存hash值是为了保证每条数据固定大小；commitLogOffset表示需要查询的消息在commitLog中的偏移量，根据偏移量可以快速查找到commitLog中的消息；timestap表示本条消息与第一条消息时间戳的差值；next index offset是链表索引，表示链表前一个节点的偏移量。

添加索引时，计算key的哈希值，并找到对应的slot，并根据header中的indexCount在LinkedList中在添加一条索引数据。如果原slot中已经存在一条数据，将原slot中的数据写入新增数据的next index offset字段，并更新slot中的数据为新数据的偏移量。

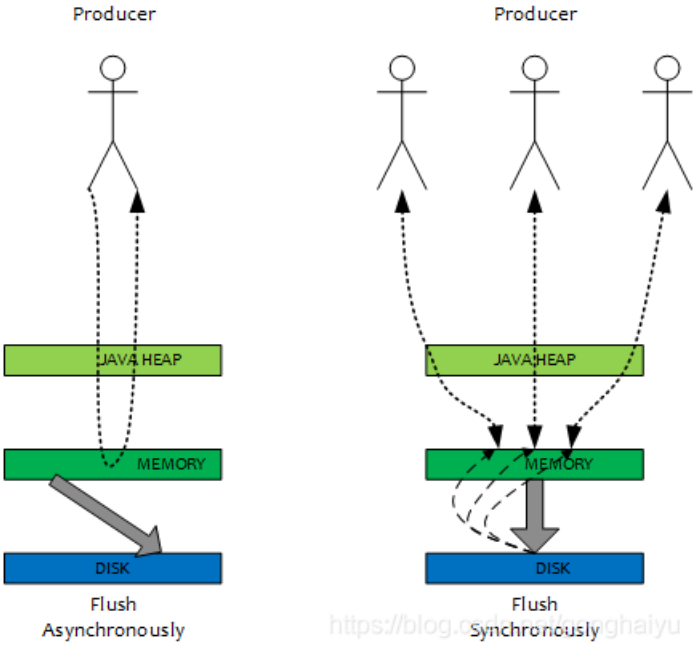
根据key查询时，先计算key的hash值，然后找到slot中存储的偏移量，在根据偏移量依次轮询链表中的消息数据，找到对应的key之后返回数据即可。

IndexFile 是消息索引文件，Index 索引文件提供了对 CommitLog 进行数据检索，提供了一种通过 key 或者时间区间来查找 CommitLog 中的消息的方法。在物理存储中，文件名是以创建的时间戳命名，固定的单个 IndexFile 大小大概为 400M，一个 IndexFile 可以保存 2000W 个索引。

刷盘策略

RocketMQ 的所有消息都是持久化的，先写入系统 PAGECACHE，然后刷盘，可以保证内存与磁盘都有一份数据，访问时，直接从内存读取。

异步刷盘



在有 RAID 卡，SAS 15000 转磁盘测试顺序写文件，速度可以达到 300M 每秒左右，而线上的网卡一般都为千兆网卡，写磁盘速度明显快于数据网络入口速度，那么是否可以做到写完内存就向用户返回，由后台线程刷盘呢？

1. 由于磁盘速度大于网卡速度，那么刷盘的进度肯定可以跟上消息的写入速度。
2. 万一由于此时系统压力过大，可能堆积消息，除了写入 IO，还有读取 IO，万一出现磁盘读取落后情况，会不会导致系统内存溢出，答案是否定的，原因如下：
   （1）写入消息到 PAGECACHE 时，如果内存不足，则尝试丢弃干净的 PAGE，腾出内存供新消息使用，策略是 LRU 方式。
   （2）如果干净页不足，此时写入 PAGECACHE 会被阻塞，系统尝试刷盘部分数据，大约每次尝试 32 个 PAGE，来找出更多干净 PAGE。
   综上，内存溢出的情况不会出现。

同步刷盘

同步刷盘与异步刷盘的唯一区别是异步刷盘写完 PAGECACHE 直接返回，而同步刷盘需要等待刷盘完成才返回，
同步刷盘流程如下：
(1). 写入 PAGECACHE 后，线程等待，通知刷盘线程刷盘。
(2). 刷盘线程刷盘后，唤醒前端等待线程，可能是一批线程。
(3). 前端等待线程向用户返回成功。

消息可靠性

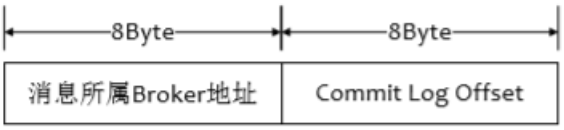
RocketMQ从3.0版本开始支持同步双写，来解决不同情况下的消息丢失问题。

消息查询

RocketMQ支持两种查询方式。

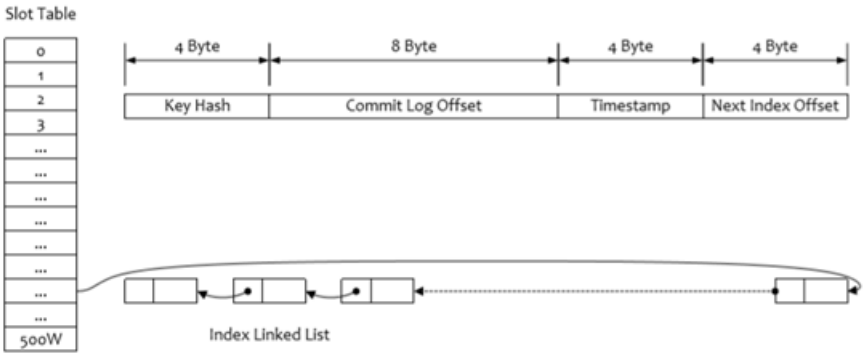
按照 Message Id 查询消息

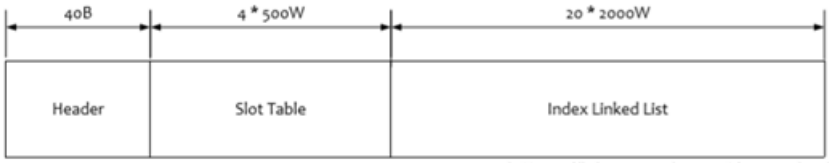
MsgId 总共 16 字节，包含消息存储主机地址，消息 Commit Log offset。从 MsgId 中解析出 Broker 的地址和Commit Log 的偏移地址，然后按照存储格式所在位置消息 buffer 解析成一个完整的消息。



图表 7-2 Message Id 组成

按照 Message Key 查询消息





图表 7-3 索引的逻辑结构，类似 HashMap 实现

1. 根据查询的 key 的 hashcode%slotNum 得到具体的槽的位置（slotNum 是一个索引文件里面包含的最大槽的数目，例如图中所示 slotNum=5000000）。
2. 根据 slotValue（slot 位置对应的值）查找到索引项列表的最后一项（倒序排列，slotValue 总是指向最新的一个索引项）。
3. 遍历索引项列表返回查询时间范围内的结果集（默认一次最大返回的 32 条记录）
4. Hash 冲突；寻找 key 的 slot 位置时相当于执行了两次散列函数，一次 key 的 hash，一次 key 的 hash 值取模，因此这里存在两次冲突的情况；第一种，key 的 hash 值不同但模数相同，此时查询的时候会在比较一次 key 的 hash 值（每个索引项保存了 key 的 hash 值），过滤掉 hash 值不相等的项。第二种，hash 值相等但 key 不等，出于性能的考虑冲突的检测放到客户端处理（key 的原始值是存储在消息文件中的，避免对数据文件的解析），客户端比较一次消息体的 key 是否相同。
5. 存储；为了节省空间索引项中存储的时间是时间差值（存储时间-开始时间，开始时间存储在索引文件头中），整个索引文件是定长的，结构也是固定的。索引文件存储结构参见前面的图 。

消息推拉模式选择

RocketMQ采用长轮询 Pull的方式进行消费。
RocketMQ 的 Consumer 都是从 Broker 拉消息来消费，但是为了能做到实时收消息，RocketMQ 使用长轮询方式，可以保证消息实时性同Push方式一致。这种长轮询方式类似于 Web QQ 收发消息机制。请参考以下信息了解。
更多http://www.ibm.com/developerworks/cn/web/wa-lo-comet/

参考

IndexFile文件结构：https://blog.csdn.net/rodbate/article/details/78763379

RocketMQ持久化策略：https://zhuanlan.zhihu.com/p/103784624