

Kafka原理与技术

Kafka的基本介绍

Kafka最初由Linkedin公司开发，是一个分布式、分区、多副本、多订阅者，基于zookeeper协调的分布式日志系统（也可以当做MQ系统），常用于web/nginx日志、访问日志，消息服务等等场景。Linkedin于2010年贡献给了Apache基金会并成为顶级开源项目。

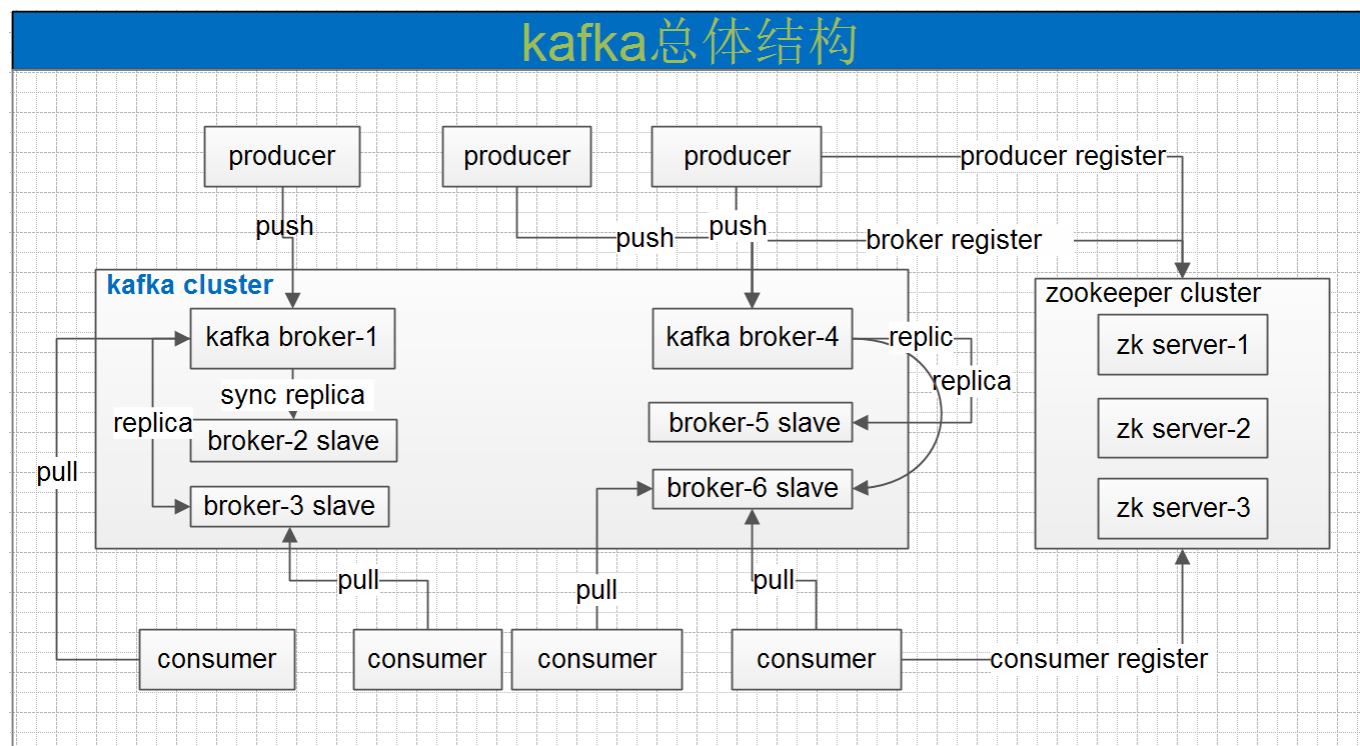
主要应用场景是：日志收集系统和消息系统。

Kafka主要设计目标如下：

- 以时间复杂度为 $O(1)$ 的方式提供消息持久化能力，即使对TB级以上数据也能保证常数时间的访问性能。
- 高吞吐率，即使在非常廉价的商用机器上也能做到单机支持每秒100K条消息的传输。
- 支持Kafka Server间的消息分区，及分布式消费，同时保证每个partition内的消息顺序传输。
- 同时支持离线数据处理和实时数据处理。

Kafka的设计原理

Kafka整体系统架构设计如下：



如果想及时了

解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

一个典型的 Kafka 集群包含若干 Producer，若干 Broker，若干 Consumer，以及一个 Zookeeper 集群。Kafka 通过 Zookeeper 管理集群配置，选举 Leader，以及在 Consumer Group发生变化时进行 Rebalance。Producer 使用 push 模式将消息发布到 Broker；Consumer 使用 pull 模式从 Broker 订阅并消费消息。

Kafka专用术语：

- Broker：消息中间件处理结点，一个Kafka节点就是一个Broker，多个Broker可以组成一个Kafka集群。
- Topic：一类消息，Kafka集群能够同时负责多个topic的分发。
- Partition：topic物理上的分组，一个topic可以分为多个partition，每个partition是一个有序的队列。
- Segment：partition物理上由多个segment组成。
- offset：每个partition都由一系列有序的、不可变的消息组成，这些消息被连续的追加到partition中。partition中的每个消息都有一个连续的序列号叫做offset，用于partition唯一标识一条消息。
- Producer：负责发布消息到Kafka broker。
- Consumer：消息消费者，向Kafka broker读取消息的客户端。
- Consumer Group：每个Consumer属于一个特定的Consumer Group。

Kafka数据传输的事务特点

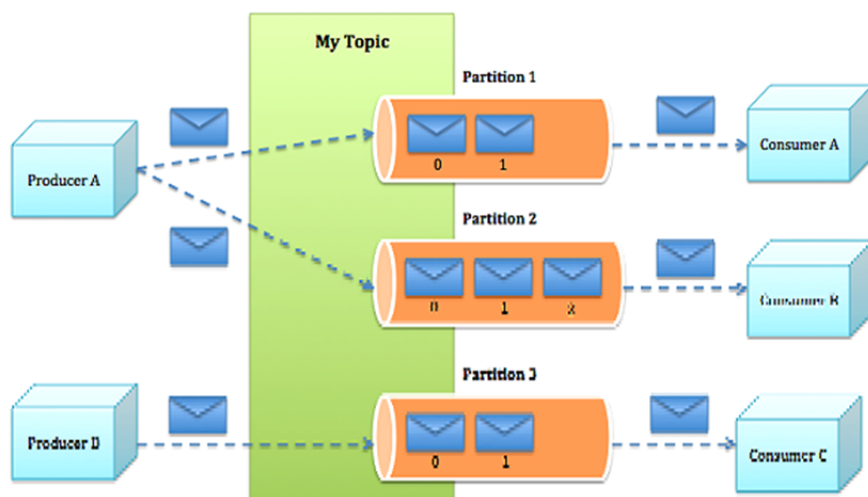
- at most once：最多一次，这个和JMS中"非持久化"消息类似，发送一次，无论成败，将不会重发。消费者fetch消息，然后保存offset，然后处理消息；当client保存offset之后，但是在消息处理过程中出现了异常，导致部分消息未能继续处理。那么此后"未处理"的消息将不能被fetch到，这就是"at most once"。
- at least once：消息至少发送一次，如果消息未能接受成功，可能会重发，直到接收成功。消费者fetch消息，然后处理消息，然后保存offset。如果消息处理成功之后，但是在保存offset阶段zookeeper异常导致保存操作未能执行成功，这就导致接下来再次fetch时可能获得上次已经处理过的消息，这就是"at least once"，原因offset没有及时的提交给zookeeper，zookeeper恢复正常还是之前offset状态。
- exactly once：消息只会发送一次。kafka中并没有严格的去实现（基于2阶段提交），我们认为这种策略在kafka中是没有必要的。

通常情况下 at-least-once 是我们首选。

Kafka消息存储

Topic & Partition

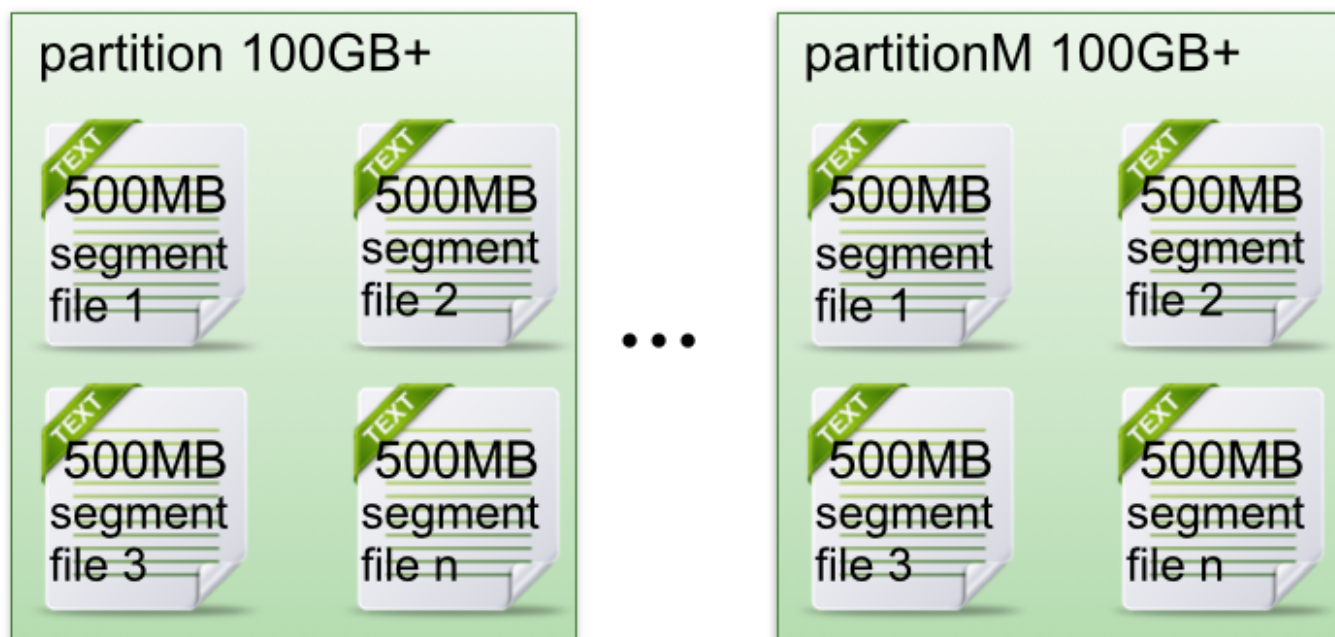
同一个Topic 通常存储的是一类消息，每个topic内部实现又被分成多个partition，每个partition在存储层面是append log文件。



如果想及时了

解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

在Kafka文件存储中，同一个topic下有多个不同partition，每个partition为一个目录，partition命名规则为topic名称+有序序号，第一个partition序号从0开始，序号最大值为partitions数量减1。



如果想及时了

解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

- 每个partition（目录）相当于一个巨型文件被平均分配到多个大小相等segment（段）数据文件中。但每个段segment file消息数量不一定相等，这种特性方便old segment file快速被删除。
- 每个partition只需要支持顺序读写就行了，segment文件生命周期由服务端配置参数决定

。

这样做的好处就是能快速删除无用文件，有效提高磁盘利用率。

- segment file组成：由2大部分组成，分别为index file和data file，此2个文件一一对应，成对出现，后缀“.index”和“.log”分别表示为segment索引文件、数据文件。
- segment文件命名规则：partition全局的第一个segment从0开始，后续每个segment文件名为上一个segment文件最后一条消息的offset值。数值最大为64位long大小，19位数字字符长度，没有数字用0填充。

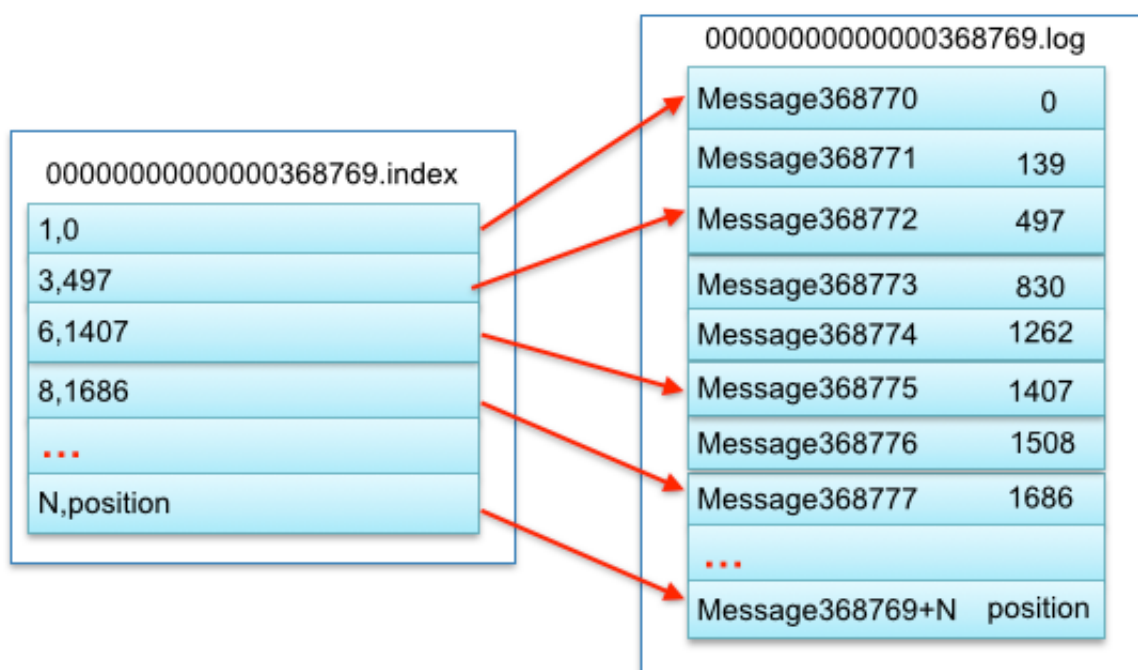
```
00000000000000000000.index
00000000000000000000.log
000000000000000368769.index
000000000000000368769.log
000000000000000737337.index
000000000000000737337.log
000000000000001105814.index
000000000000001105814.log
```

.....

如果想及时了

解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

segment中index与data file对应关系物理结构如下：



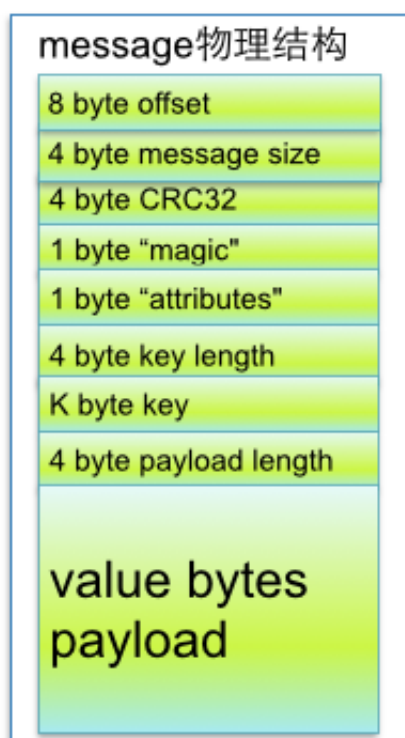
如果想及时了

解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

上图中索引文件存储大量元数据，数据文件存储大量消息，索引文件中元数据指向对应数据文件中message的物理偏移地址。

其中以索引文件中元数据3,497为例，依次在数据文件中表示第3个message（在全局partiton表示第368772个message），以及该消息的物理偏移地址为497。

了解到segment data file由许多message组成，下面详细说明message物理结构如下：



如果想及时了

解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

参数说明：

关键字	解释说明
8 byte offset	在partition(分区)内的每条消息都有一个有序的id号，这个id号被称为偏移(offset)，它可以唯一确定每条消息在partition(分区)内的位置。即offset表示partiion的第多少message
4 byte message size	message大小
4 byte CRC32	用crc32校验message

关键字	解释说明
1 byte "magic"	表示本次发布Kafka服务程序协议版本号
1 byte "attributes"	表示为独立版本、或标识压缩类型、或编码类型。
4 byte key length	表示key的长度,当key为-1时, K byte key字段不填
K byte key	可选
value bytes payload	表示实际消息数据。

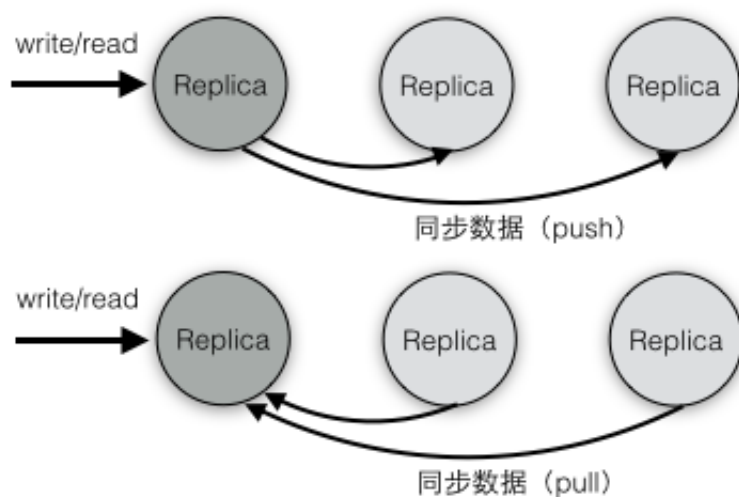
副本 (replication) 策略

Kafka的高可靠性的保障来源于其健壮的副本 (replication) 策略。

1) 数据同步

kafka在0.8版本前没有提供Partition的Replication机制，一旦Broker宕机，其上的所有Partition就都无法提供服务，而Partition又没有备份数据，数据的可用性就大大降低了。所以0.8后提供了Replication机制来保证Broker的failover。

引入Replication之后，同一个Partition可能会有多个Replica，而这时需要在这些Replication之间选出一个Leader，Producer和Consumer只与这个Leader交互，其它Replica作为Follower从Leader中复制数据。



选举一个Leader副本负责读写，leader同步数据给follower，用于故障时做failover

如果想及时了

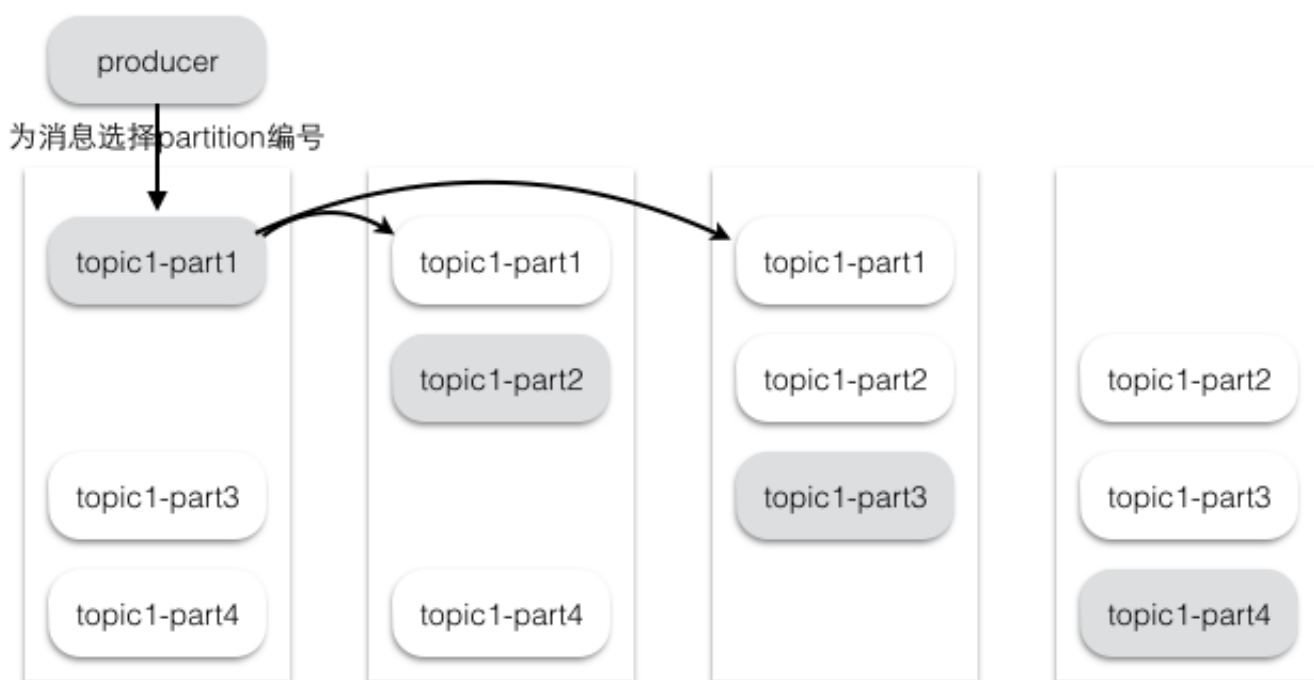
解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

2) 副本放置策略

为了更好的做负载均衡，Kafka尽量将所有的Partition均匀分配到整个集群上。Kafka分配Replica的算法如下：

- 将所有存活的N个Brokers和待分配的Partition排序
- 将第i个Partition分配到第 $(i \bmod n)$ 个Broker上，这个Partition的第一个Replica存在于这个分配的Broker上，并且会作为partition的优先副本
- 将第i个Partition的第j个Replica分配到第 $((i + j) \bmod n)$ 个Broker上

假设集群一共有4个brokers，一个topic有4个partition，每个Partition有3个副本。下图是每个Broker上的副本分配情况。



如果想及时了

解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

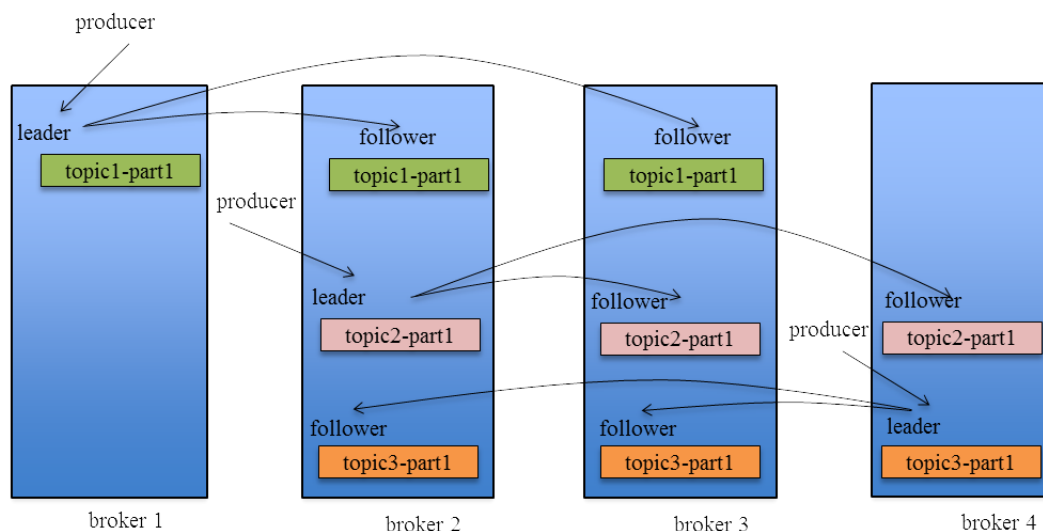
3) 同步策略

Producer在发布消息到某个Partition时，先通过ZooKeeper找到该Partition的Leader，然后无论该Topic的Replication Factor为多少，Producer只将该消息发送到该Partition的Leader。Leader会将该消息写入其本地Log。每个Follower都从Leader pull数据。这种方式上，Follower存储的数据顺序与Leader保持一致。Follower在收到该消息并写入其Log后，向Leader发送ACK。一旦Leader收到了ISR中的所有Replica的ACK，该消息就被认为已经commit了，Leader将增加HW并且向Producer发送ACK。

为了提高性能，每个Follower在接收到数据后就立马向Leader发送ACK，而非等到数据写入Log中。因此，对于已经commit的消息，Kafka只能保证它被存于多个Replica的内存中，而不能保证它们被持久化到磁盘中，也就不能完全保证异常发生后该条消息一定能被Consumer消费。

Consumer读消息也是从Leader读取，只有被commit过的消息才会暴露给Consumer。

Kafka Replication的数据流如下图所示：



如果想及时了

解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

对于Kafka而言，定义一个Broker是否“活着”包含两个条件：

- 一是它必须维护与ZooKeeper的session（这个通过ZooKeeper的Heartbeat机制来实现）。
- 二是Follower必须能够及时将Leader的消息复制过来，不能“落后太多”。

Leader会跟踪与其保持同步的Replica列表，该列表称为ISR（即in-sync Replica）。如果一个Follower宕机，或者落后太多，Leader将把它从ISR中移除。这里所描述的“落后太多”指Follower复制的消息落后于Leader后的条数超过预定值或者Follower超过一定时间未向Leader发送fetch请求。

Kafka只解决fail/recover，一条消息只有被ISR里的所有Follower都从Leader复制过去才会被认为已提交。这样就避免了部分数据被写进了Leader，还没来得及被任何Follower复制就宕机了，而造成数据丢失（Consumer无法消费这些数据）。而对于Producer而言，它可以选择是否等待消息commit。这种机制确保了只要ISR有一个或以上的Follower，一条被commit的消息就不会丢失。

4) leader选举

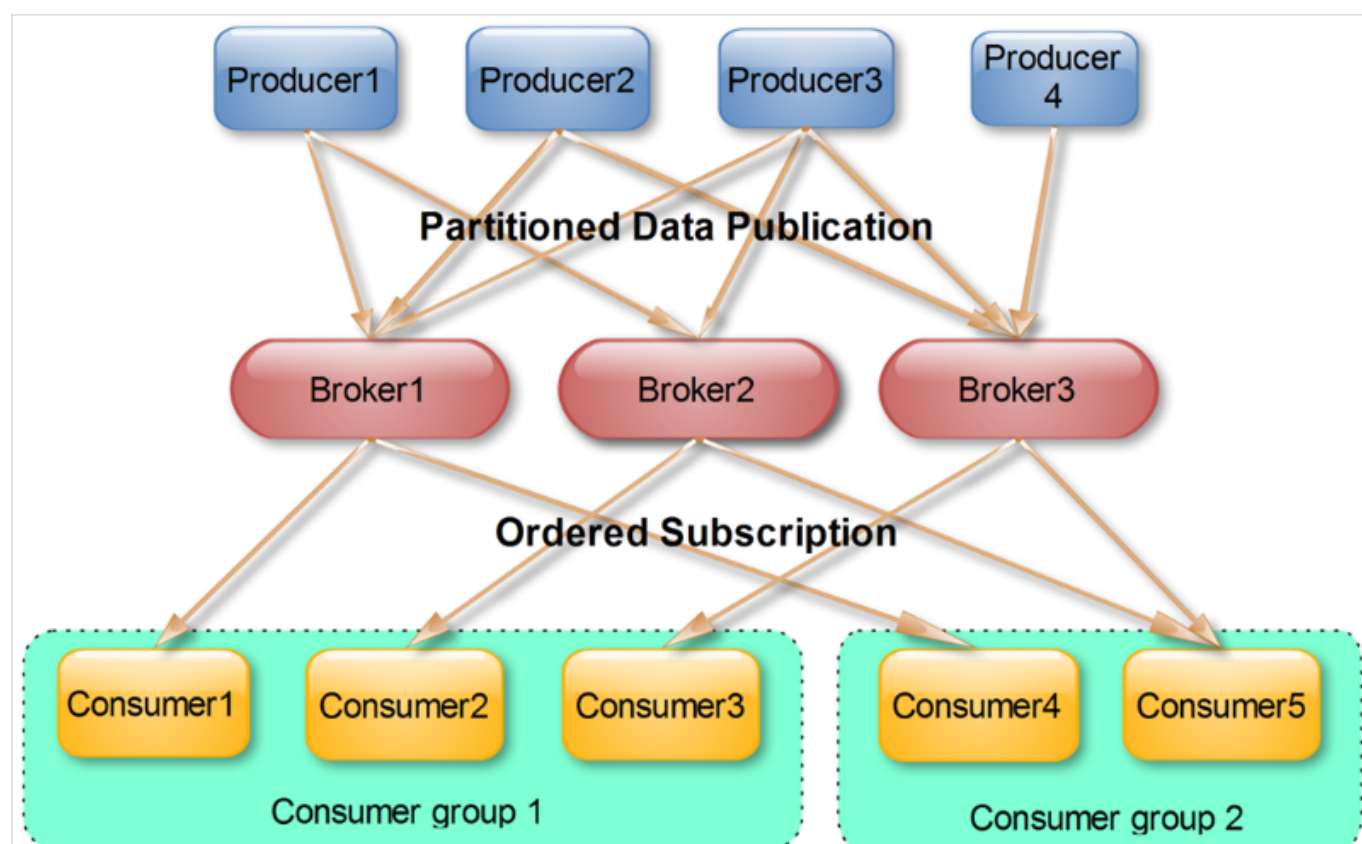
Leader选举本质上是一个分布式锁，有两种方式实现基于ZooKeeper的分布式锁：

- 节点名称唯一性：多个客户端创建一个节点，只有成功创建节点的客户端才能获得锁
- 临时顺序节点：所有客户端在某个目录下创建自己的临时顺序节点，只有序号最小的才获得锁

Majority Vote的选举策略和ZooKeeper中的Zab选举是类似的，实际上ZooKeeper内部本身就实现了少数服从多数的选举策略。kafka中对于Partition的leader副本的选举采用了第一种方法：为Partition分配副本，指定一个ZNode临时节点，第一个成功创建节点的副本就是Leader节点，其他副本会在这个ZNode节点上注册Watcher监听器，一旦Leader宕机，对应的临时节点就会被自动删除，这时注册在该节点上的所有Follower都会收到监听器事件，它们都会尝试创建该节点，只有创建成功的那个follower才会成为Leader（ZooKeeper保证对于一个节点只有一个客户端能创建成功），其他follower继续重新注册监听事件。

Kafka消息分组，消息消费原理

同一Topic的一条消息只能被同一个Consumer Group内的一个Consumer消费，但多个Consumer Group可同时消费这一消息。



如果想及时了

解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

这是Kafka用来实现一个Topic消息的广播（发给所有的Consumer）和单播（发给某一个Consumer）的手段。一个Topic可以对应多个Consumer Group。如果需实现广播，只要每个Consumer有一个独立的Group就可以了。要实现单播只要所有的Consumer在同一个Group里。用Consumer Group还可以将Consumer进行自由的分组而不需要多次发送消息到不同的Topic。

Push vs. Pull

作为一个消息系统，Kafka遵循了传统的方式，选择由Producer向broker push消息并由Consumer从broker pull消息。

push模式很难适应消费速率不同的消费者，因为消息发送速率是由broker决定的。push模式的目标是尽可能以最快速度传递消息，但是这样很容易造成Consumer来不及处理消息，典型的表现就是拒绝服务以及网络拥塞。而pull模式则可以根据Consumer的消费能力以适当的速率消费消息。

对于Kafka而言，pull模式更合适。pull模式可简化broker的设计，Consumer可自主控制消费消息的速率，同时Consumer可以自己控制消费方式——即可批量消费也可逐条消费，同时还能选择不同的提交方式从而实现不同的传输语义。

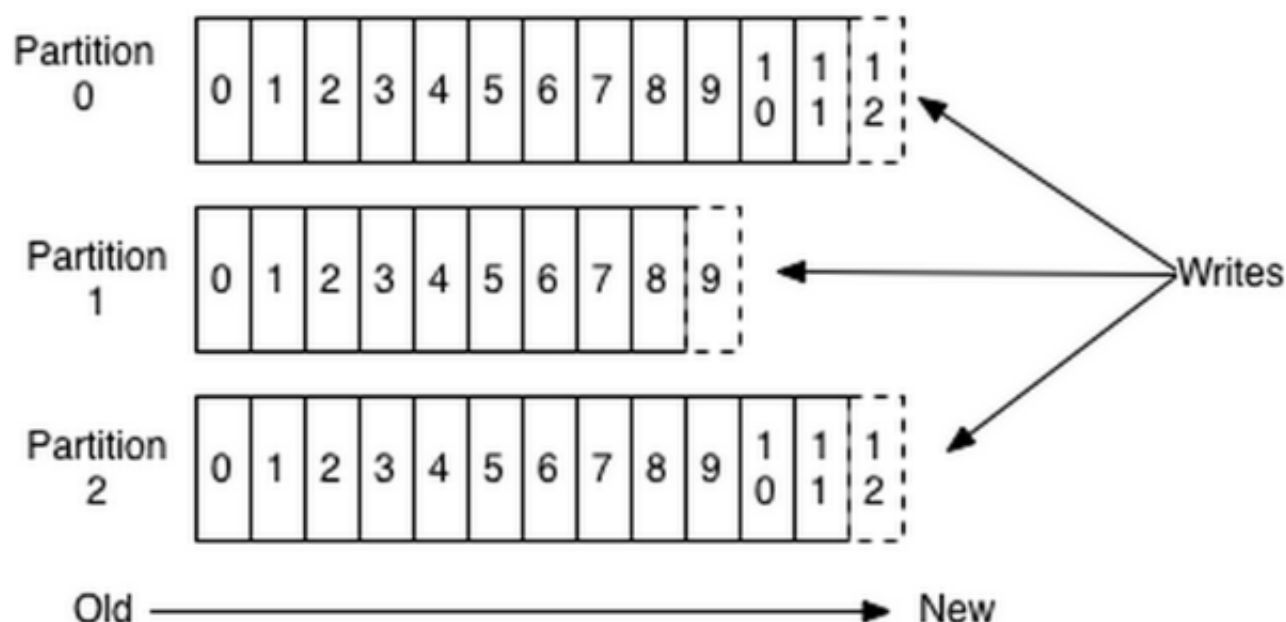
Kafak顺序写入与数据读取

生产者（producer）是负责向Kafka提交数据的，Kafka会把收到的消息都写入到硬盘中，它绝对不会丢失数据。为了优化写入速度Kafak采用了两个技术，顺序写入和MMFile。

顺序写入

因为硬盘是机械结构，每次读写都会寻址，写入，其中寻址是一个“机械动作”，它是最耗时的。所以硬盘最“讨厌”随机I/O，最喜欢顺序I/O。为了提高读写硬盘的速度，Kafka就是使用顺序I/O。

每条消息都被append到该Partition中，属于顺序写磁盘，因此效率非常高。



如果想及时了

解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

对于传统的message queue而言，一般会删除已经被消费的消息，而Kafka是不会删除数据的，它

会把所有的数据都保留下来，每个消费者（Consumer）对每个Topic都有一个offset用来表示读取到了第几条数据。

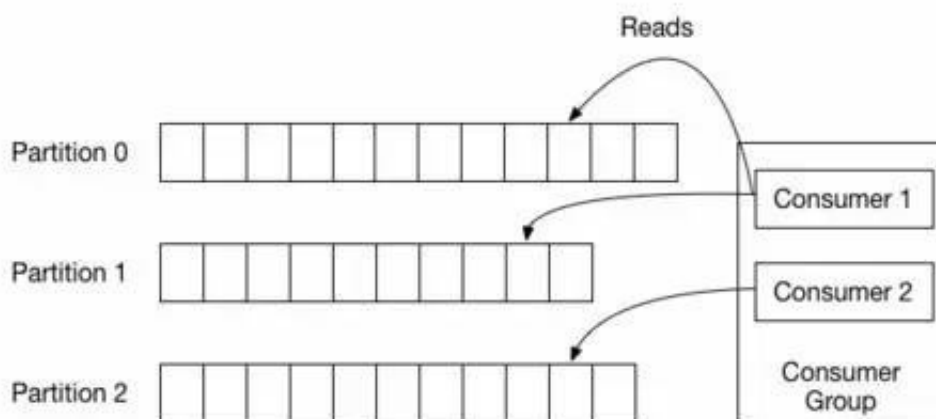


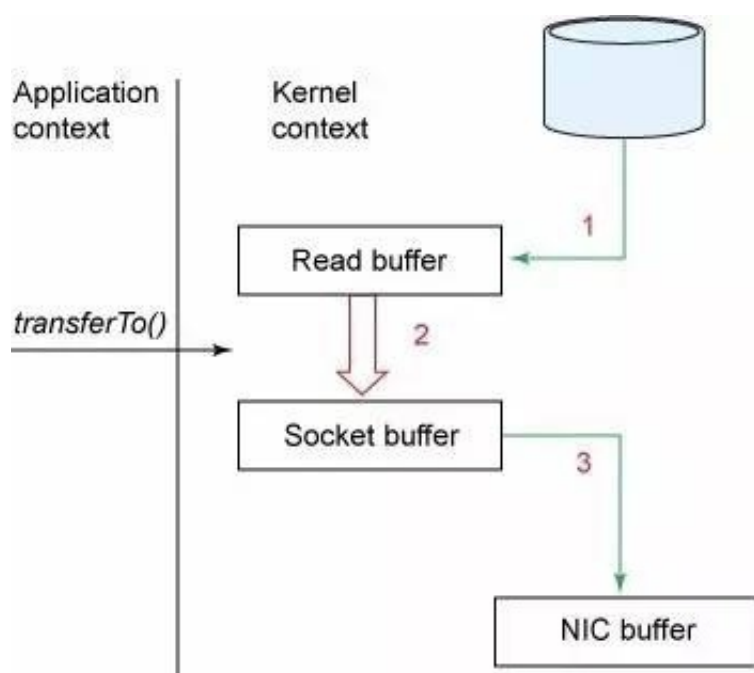
Figure 1: Consumer Group

如果想及时了

解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

即便是顺序写入硬盘，硬盘的访问速度还是不可能追上内存。所以Kafka的数据并不是实时的写入硬盘，它充分利用了现代操作系统分页存储来利用内存提高I/O效率。

在Linux Kernal 2.2之后出现了一种叫做“零拷贝(zero-copy)”系统调用机制，就是跳过“用户缓冲区”的拷贝，建立一个磁盘空间和内存空间的直接映射，数据不再复制到“用户态缓冲区”系统上下文切换减少2次，可以提升一倍性能。



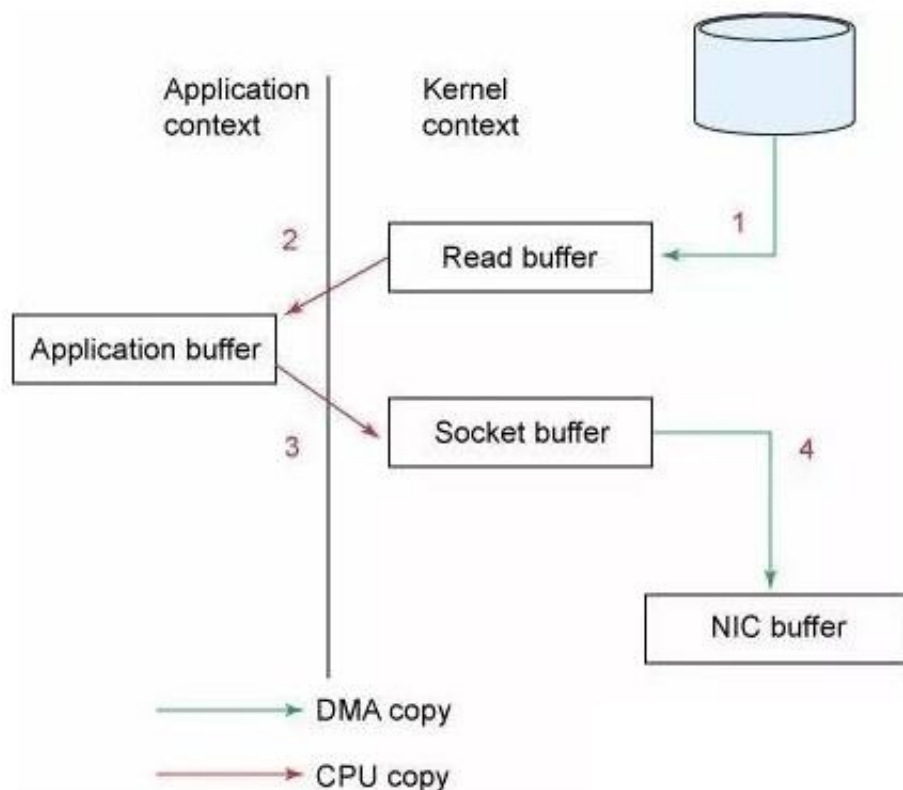
如果想及时了

解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

通过mmap，进程像读写硬盘一样读写内存（当然是虚拟机内存）。使用这种方式可以获取很大的I/O提升，省去了用户空间到内核空间复制的开销（调用文件的read会把数据先放到内核空间的内存中，然后再复制到用户空间的内存中。）

消费者（读取数据）

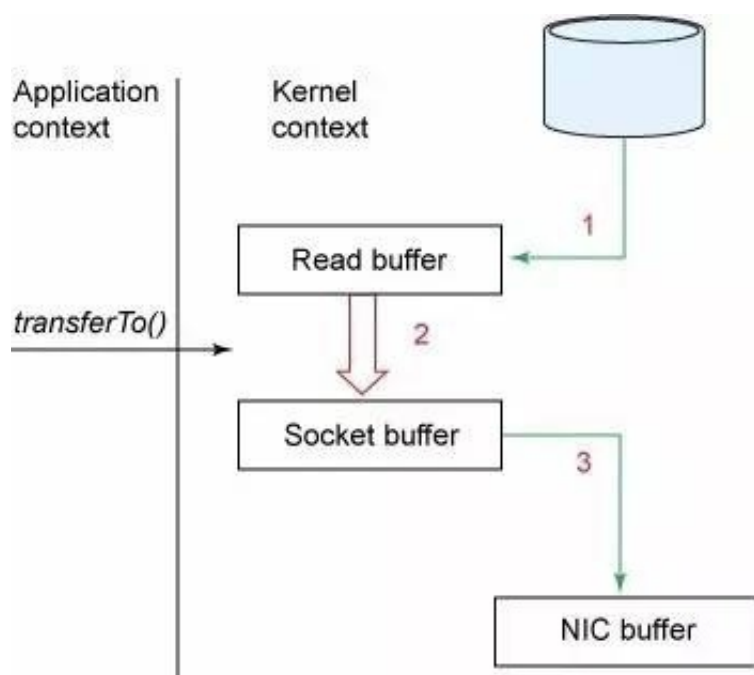
试想一下，一个Web Server传送一个静态文件，如何优化？答案是zero copy。传统模式下我们从硬盘读取一个文件是这样的。



如果想及时了

解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

先复制到内核空间（read是系统调用，放到了DMA，所以用内核空间），然后复制到用户空间（1、2）；从用户空间重新复制到内核空间（你用的socket是系统调用，所以它也有自己的内核空间），最后发送给网卡（3、4）。



如果想及时了

解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

Zero Copy中直接从内核空间（DMA的）到内核空间（Socket的），然后发送网卡。这个技术非常普遍，Nginx也是用的这种技术。

实际上，Kafka把所有的消息都存放在一个一个的文件中，当消费者需要数据的时候Kafka直接把“文件”发送给消费者。当不需要把整个文件发出去的时候，Kafka通过调用Zero Copy的sendfile这个函数，这个函数包括：

- out_fd作为输出（一般及时socket的句柄）
- in_fd作为输入文件句柄
- off_t表示in_fd的偏移（从哪里开始读取）
- size_t表示读取多少个

参考

- <http://www.cnblogs.com/liuming1992/p/6423007.html>
- <http://blog.csdn.net/lifuxiangcaohui/article/details/51374862>
- <http://www.jasongj.com/2015/01/02/Kafka深度解析>
- <http://www.infoq.com/cn/articles/kafka-analysis-part-2>
- <http://zqhxyuan.github.io/2016/02/23/2016-02-23-Kafka-Controller>
- <https://tech.meituan.com/kafka-fs-design-theory.html>
- <https://my.oschina.net/silence88/blog/856195>
- https://toutiao.io/posts/508935/app_preview

本文转载自 linkedkeeper.com（文 / 张松然）

本博客文章除特别声明，全部都是原创！

原创文章版权归过往记忆大数据（[过往记忆](#)）所有，未经许可不得转载。

本文链接: **【】** ()