# 高级kafka

## 数据持久化

Kafka大量依赖文件系统去存储和缓存消息。对于硬盘有个传统的观念是硬盘总是很慢，这使很多人怀疑基于文件系统的架构能否提供优异的性能。实际上硬盘的快慢完全取决于使用它的方式。设计良好的硬盘架构可以和内存一样快。

在6块7200转的SATA RAID-5磁盘阵列的线性写速度差不多是600MB/s，但是随即写的速度却是100k/s，差了差不多6000倍。现代的操作系统都对次做了大量的优化，使用了 read-ahead 和 write-behind的技巧，读取的时候成块的预读取数据，写的时候将各种微小琐碎的逻辑写入组织合并成一次较大的物理写入。对此的深入讨论可以查看这里，它们发现线性的访问磁盘，很多时候比随机的内存访问快得多。

为了提高性能，现代操作系统往往使用内存作为磁盘的缓存，现代操作系统乐于把所有空闲内存用作磁盘缓存，虽然这可能在缓存回收和重新分配时牺牲一些性能。所有的磁盘读写操作都会经过这个缓存，这不太可能被绕开除非直接使用I/O。所以虽然每个程序都在自己的线程里只缓存了一份数据，但在操作系统的缓存里还有一份，这等于存了两份数据。

另外再来讨论一下JVM,以下两个事实是众所周知的：

Java对象占用空间是非常大的，差不多是要存储的数据的两倍甚至更高。

随着堆中数据量的增加，垃圾回收回变的越来越困难。

基于以上分析，如果把数据缓存在内存里，因为需要存储两份，不得不使用两倍的内存空间，Kafka基于JVM，又不得不将空间再次加倍,再加上要避免GC带来的性能影响，在一个32G内存的机器上，不得不使用到28-30G的内存空间。并且当系统重启的时候，又必须要将数据刷到内存中（ 10GB 内存差不多要用10分钟），就算使用冷刷新（不是一次性刷进内存，而是在使用数据的时候没有就刷到内存）也会导致最初的时候新能非常慢。但是使用文件系统，即使系统重启了，也不需要刷新数据。使用文件系统也简化了维护数据一致性的逻辑。

所以与传统的将数据缓存在内存中然后刷到硬盘的设计不同，Kafka直接将数据写到了文件系统的日志中。

常量时间的操作效率

在大多数的消息系统中，数据持久化的机制往往是为每个cosumer提供一个B树或者其他的随机读写的数据结构。B树当然是很棒的，但是也带了一些代价：比如B树的复杂度是O(log N)，O(log N)通常被认为就是常量复杂度了，但对于硬盘操作来说并非如此。磁盘进行一次搜索需要10ms，每个硬盘在同一时间只能进行一次搜索，这样并发处理就成了问题。虽然存储系统使用缓存进行了大量优化，但是对于树结构的性能的观察结果却表明，它的性能往往随着数据的增长而线性下降，数据增长一倍，速度就会降低一倍。

直观的讲，对于主要用于日志处理的消息系统，数据的持久化可以简单的通过将数据追加到文件中实现，读的时候从文件中读就好了。这样做的好处是读和写都是 O(1) 的，并且读操作不会阻塞写操作和其他操作。这样带来的性能优势是很明显的，因为性能和数据的大小没有关系了。

既然可以使用几乎没有容量限制（相对于内存来说）的硬盘空间建立消息系统，就可以在没有性能损失的情况下提供一些一般消息系统不具备的特性。比如，一般的消息系统都是在消息被消费后立即删除，Kafka却可以将消息保存一段时间（比如一星期），这给consumer提供了很好的机动性和灵活性。

Kafka允许用户为每个topic设置副本数量，副本数量决定了有几个broker来存放写入的数据。如果你的副本数量设置为3，那么一份数据就会被存放在3台不同的机器上，那么就允许有2个机器失败。一般推荐副本数量至少为2，这样就可以保证增减、重启机器时不会影响到数据消费。如果对数据持久化有更高的要求，可以把副本数量设置为3或者更多。

## 消息传输的事务定义

之前讨论了consumer和producer是怎么工作的，现在来讨论一下数据传输方面。数据传输的事务定义通常有以下三种级别：

最多一次: 消息不会被重复发送，最多被传输一次，但也有可能一次不传输。

最少一次: 消息不会被漏发送，最少被传输一次，但也有可能被重复传输.

精确的一次（Exactly once）: 不会漏传输也不会重复传输,每个消息都传输被一次而且仅仅被传输一次，这是大家所期望的。

大多数消息系统声称可以做到“精确的一次”，但是仔细阅读它们的的文档可以看到里面存在误导，比如没有说明当consumer或producer失败时怎么样，或者当有多个consumer并行时怎么样，或写入硬盘的数据丢失时又会怎么样。kafka的做法要更先进一些。当发布消息时，Kafka有一个“committed”的概念，一旦消息被提交了，只要消息被写入的分区的所在的副本broker是活动的，数据就不会丢失。关于副本的活动的概念，下节文档会讨论。现在假设broker是不会down的。

如果producer发布消息时发生了网络错误，但又不确定实在提交之前发生的还是提交之后发生的，这种情况虽然不常见，但是必须考虑进去，现在Kafka版本还没有解决这个问题，将来的版本正在努力尝试解决。

并不是所有的情况都需要“精确的一次”这样高的级别，Kafka允许producer灵活的指定级别。比如producer可以指定必须等待消息被提交的通知，或者完全的异步发送消息而不等待任何通知，或者仅仅等待leader声明它拿到了消息（followers没有必要）。

现在从consumer的方面考虑这个问题，所有的副本都有相同的日志文件和相同的offset，consumer维护自己消费的消息的offset，如果consumer不会崩溃当然可以在内存中保存这个值，当然谁也不能保证这点。如果consumer崩溃了，会有另外一个consumer接着消费消息，它需要从一个合适的offset继续处理。这种情况下可以有以下选择：

consumer可以先读取消息，然后将offset写入日志文件中，然后再处理消息。这存在一种可能就是在存储offset后还没处理消息就crash了，新的consumer继续从这个offset处理，那么就会有些消息永远不会被处理，这就是上面说的“最多一次”。

consumer可以先读取消息，处理消息，最后记录offset，当然如果在记录offset之前就crash了，新的consumer会重复的消费一些消息，这就是上面说的“最少一次”。

“精确一次”可以通过将提交分为两个阶段来解决：保存了offset后提交一次，消息处理成功之后再提交一次。但是还有个更简单的做法：将消息的offset和消息被处理后的结果保存在一起。比如用Hadoop ETL处理消息时，将处理后的结果和offset同时保存在HDFS中，这样就能保证消息和offser同时被处理了。

## 性能提升

Kafka在提高效率方面做了很大努力。Kafka的一个主要使用场景是处理网站活动日志，吞吐量是非常大的，每个页面都会产生好多次写操作。读方面，假设每个消息只被消费一次，读的量的也是很大的，Kafka也尽量使读的操作更轻量化。

我们之前讨论了磁盘的性能问题，线性读写的情况下影响磁盘性能问题大约有两个方面：太多的琐碎的I/O操作和太多的字节拷贝。I/O问题发生在客户端和服务端之间，也发生在服务端内部的持久化的操作中。

### 消息集（message set）

为了避免这些问题，Kafka建立了“消息集（message set）”的概念，将消息组织到一起，作为处理的单位。以消息集为单位处理消息，比以单个的消息为单位处理，会提升不少性能。Producer把消息集一块发送给服务端，而不是一条条的发送；服务端把消息集一次性的追加到日志文件中，这样减少了琐碎的I/O操作。consumer也可以一次性的请求一个消息集。

另外一个性能优化是在字节拷贝方面。在低负载的情况下这不是问题，但是在高负载的情况下它的影响还是很大的。为了避免这个问题，Kafka使用了标准的二进制消息格式，这个格式可以在producer,broker和producer之间共享而无需做任何改动。

### zero copy

Broker维护的消息日志仅仅是一些目录文件，消息集以固定队的格式写入到日志文件中，这个格式producer和consumer是共享的，这使得Kafka可以一个很重要的点进行优化：消息在网络上的传递。现代的unix操作系统提供了高性能的将数据从页面缓存发送到socket的系统函数，在linux中，这个函数是sendfile.

为了更好的理解sendfile的好处，我们先来看下一般将数据从文件发送到socket的数据流向：

操作系统把数据从文件拷贝内核中的页缓存中

应用程序从页缓存从把数据拷贝自己的内存缓存中

应用程序将数据写入到内核中socket缓存中

操作系统把数据从socket缓存中拷贝到网卡接口缓存，从这里发送到网络上。

这显然是低效率的，有4次拷贝和2次系统调用。Sendfile通过直接将数据从页面缓存发送网卡接口缓存，避免了重复拷贝，大大的优化了性能。

在一个多consumers的场景里，数据仅仅被拷贝到页面缓存一次而不是每次消费消息的时候都重复的进行拷贝。这使得消息以近乎网络带宽的速率发送出去。这样在磁盘层面你几乎看不到任何的读操作，因为数据都是从页面缓存中直接发送到网络上去了。

### 压缩

很多时候，性能的瓶颈并非CPU或者硬盘而是网络带宽，对于需要在数据中心之间传送大量数据的应用更是如此。当然用户可以在没有Kafka支持的情况下各自压缩自己的消息，但是这将导致较低的压缩率，因为相比于将消息单独压缩，将大量文件压缩在一起才能起到最好的压缩效果。

Kafka采用了端到端的压缩：因为有“消息集”的概念，客户端的消息可以一起被压缩后送到服务端，并以压缩后的格式写入日志文件，以压缩的格式发送到consumer，消息从producer发出到consumer拿到都被是压缩的，只有在consumer使用的时候才被解压缩，所以叫做“端到端的压缩”。

Kafka支持GZIP和Snappy压缩协议。那么如何区分消息是压缩的还是未压缩的呢，Kafka在消息头部添加了一个描述压缩属性字节，这个字节的后两位表示消息的压缩采用的编码，如果后两位为0，则表示消息未被压缩。

## Producer和Consumer

### Kafka Producer消息发送

producer直接将数据发送到broker的leader(主节点)，不需要在多个节点进行分发。为了帮助producer做到这点，所有的Kafka节点都可以及时的告知:哪些节点是活动的，目标topic目标分区的leader在哪。这样producer就可以直接将消息发送到目的地了。  
 客户端控制消息将被分发到哪个分区。可以通过负载均衡随机的选择，或者使用分区函数。Kafka允许用户实现分区函数，指定分区的key，将消息hash到不同的分区上(当然有需要的话，也可以覆盖这个分区函数自己实现逻辑).比如如果你指定的key是user id，那么同一个用户发送的消息都被发送到同一个分区上。经过分区之后，consumer就可以有目的的消费某个分区的消息。  
**异步发送**  
 批量发送可以很有效的提高发送效率。Kafka producer的异步发送模式允许进行批量发送，先将消息缓存在内存中，然后一次请求批量发送出去。这个策略可以配置的，比如可以指定缓存的消息达到某个量的时候就发出去，或者缓存了固定的时间后就发送出去（比如100条消息就发送，或者每5秒发送一次）。这种策略将大大减少服务端的I/O次数。缓存是在producer端进行的，那么当producer崩溃时，这些消息就会丢失。

### Kafka Consumer

**推还是拉？**

Kafka最初考虑的问题是，customer应该从brokes拉取消息还是brokers将消息推送到consumer，也就是pull还push。在这方面，Kafka遵循了一种大部分消息系统共同的传统的设计：producer 将消息推送到broker，consumer从broker拉取消息。  
 一些消息系统比如Scribe和Apache Flume采用了push模式，将消息推送到下游的consumer。这样做有好处也有坏处：由broker决定消息推送的速率，对于不同消费速率的consumer就不太好处理了。消息系统都致力于让consumer以最大的速率最快速的消费消息，但不幸的是，push模式下，当broker推送的速率远大于consumer消费的速率时，consumer恐怕就要崩溃了。最终Kafka还是选取了传统的pull模式。  
 Pull模式的另外一个好处是consumer可以自主决定是否批量的从broker拉取数据。Push模式必须在不知道下游consumer消费能力和消费策略的情况下决定是立即推送每条消息还是缓存之后批量推送。如果为了避免consumer崩溃而采用较低的推送速率，将可能导致一次只推送较少的消息而造成浪费。Pull模式下，consumer就可以根据自己的消费能力去决定这些策略。  
 Pull有个缺点是，如果broker没有可供消费的消息，将导致consumer不断在循环中轮询，直到新消息到达。为了避免这点，Kafka有个参数可以让consumer阻塞直到新消息到达(当然也可以阻塞知道消息的数量达到某个特定的量这样就可以批量发送)。

**消费状态跟踪**

对消费消息状态的记录也是很重要的。

大部分消息系统在broker端的维护消息被消费的记录：一个消息被分发到consumer后broker就马上进行标记或者等待customer的通知后进行标记。这样也可以在消息在消费后立马就删除以减少空间占用。  
  
 如果一条消息发送出去之后就立即被标记为消费过的，一旦consumer处理消息时失败了（比如程序崩溃）消息就丢失了。为了解决这个问题，很多消息系统提供了另外一个个功能：当消息被发送出去之后仅仅被标记为已发送状态，当接到consumer已经消费成功的通知后才标记为已被消费的状态。这虽然解决了消息丢失的问题，但产生了新问题，首先如果consumer处理消息成功了但是向broker发送响应时失败了，这条消息将被消费两次。

第二个问题时，broker必须维护每条消息的状态，并且每次都要先锁住消息然后更改状态然后释放锁。这样麻烦又来了，且不说要维护大量的状态数据，比如如果消息发送出去但没有收到消费成功的通知，这条消息将一直处于被锁定的状态。  
 Kafka采用了不同的策略。Topic被分成了若干分区，每个分区在同一时间只被一个consumer消费。这意味着每个分区被消费的消息在日志中的位置仅仅是一个简单的整数：offset。这样就很容易标记每个分区消费状态就很容易了，仅仅需要一个整数而已。这样消费状态的跟踪就很简单了。这带来了另外一个好处：consumer可以把offset调成一个较老的值，去重新消费老的消息。

## 主从同步

Kafka允许topic的分区拥有若干副本，这个数量是可以配置的，你可以为每个topci配置副本的数量。Kafka会自动在每个个副本上备份数据，所以当一个节点down掉时数据依然是可用的。  
  
 Kafka的副本功能不是必须的，你可以配置只有一个副本，这样其实就相当于只有一份数据。创建副本的单位是topic的分区，每个分区都有一个leader和零或多个followers.所有的读写操作都由leader处理，一般分区的数量都比broker的数量多的多，各分区的leader均匀的分布在brokers中。所有的followers都复制leader的日志，日志中的消息和顺序都和leader中的一致。flowers向普通的consumer那样从leader那里拉取消息并保存在自己的日志文件中。

Kafka判断一个节点是否活着有两个条件：

1. 节点必须可以维护和ZooKeeper的连接，Zookeeper通过心跳机制检查每个节点的连接。
2. 如果节点是个follower,他必须能及时的同步leader的写操作，延时不能太久。

符合以上条件的节点准确的说应该是“同步中的（in sync）”，而不是模糊的说是“活着的”或是“失败的”。Leader会追踪所有“同步中”的节点，一旦一个down掉了，或是卡住了，或是延时太久，leader就会把它移除。

只有当消息被所有的副本加入到日志中时，才算是“committed”，只有committed的消息才会发送给consumer，这样就不用担心一旦leader down掉了消息会丢失。Producer也可以选择是否等待消息被提交的通知，这个是由参数request.required.acks决定的。  
Kafka保证只要有一个“同步中”的节点，“committed”的消息就不会丢失。

### Leader的选择

**Kafka的核心是日志文件，日志文件在集群中的同步是分布式数据系统最基础的要素。**  
 如果leaders永远不会down的话我们就不需要followers了！一旦leader down掉了，需要在followers中选择一个新的leader.但是followers本身有可能延时太久或者crash，所以必须选择高质量的follower作为leader.必须保证，一旦一个消息被提交了，但是leader down掉了，新选出的leader必须可以提供这条消息。大部分的分布式系统采用了多数投票法则选择新的leader,对于多数投票法则，就是根据所有副本节点的状况动态的选择最适合的作为leader.Kafka并不是使用这种方法。

Kafaka动态维护了一个同步状态的副本的集合（a set of in-sync replicas），简称ISR，在这个集合中的节点都是和leader保持高度一致的，任何一条消息必须被这个集合中的每个节点读取并追加到日志中了，才回通知外部这个消息已经被提交了。因此这个集合中的任何一个节点随时都可以被选为leader.ISR在ZooKeeper中维护。ISR中有f+1个节点，就可以允许在f个节点down掉的情况下不会丢失消息并正常提供服。ISR的成员是动态的，如果一个节点被淘汰了，当它重新达到“同步中”的状态时，他可以重新加入ISR.这种leader的选择方式是非常快速的，适合kafka的应用场景。

如果所有节点都down掉了怎么办？Kafka对于数据不会丢失的保证，是基于至少一个节点是存活的，一旦所有节点都down了，这个就不能保证了。  
实际应用中，当所有的副本都down掉时，必须及时作出反应。可以有以下两种选择:

1.等待ISR中的任何一个节点恢复并担任leader。

2.选择所有节点中（不只是ISR）第一个恢复的节点作为leader.

这是一个在可用性和连续性之间的权衡。如果等待ISR中的节点恢复，一旦ISR中的节点起不起来或者数据丢失了，那集群就永远恢复不了了。如果等待ISR意外的节点恢复，这个节点的数据就会被作为线上数据，有可能和真实的数据有所出入，因为有些数据它可能还没同步到。Kafka目前选择了第二种策略，在未来的版本中将使这个策略的选择可配置，可以根据场景灵活的选择。