《Kafka官方文档》设计

[原文链接](http://kafka.apache.org/documentation/#design) 译者：BlackMamba（原文网址<http://ifeve.com/kafka-intro/>）

**Design**

**1. Motivation**

我们设计Kafka用来作为统一的平台来处理大公司可能拥有的所有实时数据源。为了做到这点，我们必须思考大量的使用场景。

它必须有高吞吐去支持大数据流，例如实时日志聚合。

它必须优雅的处理数据积压，以支持定期从离线系统加载数据。

这也以为这系统必须支持低延迟的分发来处理传统消息系统的场景。

我们想支持分区的、分布式的、实时的处理数据源并创建新的数据源，这推动了我们的分区和消费模型。

最后，将流反馈到其他系统进行服务的情况下，我们知道系统必须能够保证容错性，在部分机器故障的时候提供服务。

支持这些使用推动我们做了一些列特殊的元素设计，比起传统的消息系统更像是数据库日志。我们将在以下章节介绍一些设计要素。

**2. Persistence**

**Don’t fear the filesystem!**

Kafka强依赖文件系统来存储和缓存消息。“磁盘是缓慢的”是一个通常的认知，这是人们怀疑持久化的结构能否提供强大的性能。事实上，磁盘比人们想象的更慢也更快，这基于如何去使用它们；一个合适的设计可以使磁盘和网络一样的快速。

影响磁盘性能的核心因素是磁盘驱动的吞吐和过去十年磁盘的查找方式不同了。使用六个7200rpm的SATA RAID-5阵列的JBOD配置线性写入能力为600MB/sec，而随机写的性能仅仅是100k/sec，相差了6000倍。线性写入和读取是最可预测的，并且被操作系统大量的优化。现代操作系统提供read-ahead和write-behind技术，他们大块预读数据，并将较小的罗机械合并成较大的物理写入。在ACM Queue的文章中可以找到此问题相关的进一步讨论；他们实际上发现顺序访问磁盘在某些情况下比随机访问内存还快。

为了弥补性能的差异，现代操作系统在使用主内存来做磁盘缓存时变的越来越激进。当内存被回收时，现代操作系统将乐意将所有可用内存转移到磁盘缓存，而且性能会降低很多。所有的磁盘读写都需要通过这层缓存。这个功能不会被轻易关闭，除非使用Direct IO，因此尽管在进程内缓存了数据，这些数据也有可能在操作系统的pagecache中缓存，从而被缓存了两次。

此外，我们建立在JVM之上，任何在Java内存上花费过时间的人都知道两件事情：

对象的内存开销非常大，通常将存储的数据大小翻倍（或更多）。

Java的内存回收随着堆内数据的增多变的越来越缓慢。

由于这些因素，使用文件系统并依赖于pagecache要优于维护内存中缓存或其他结构——我们至少可以通过直接访问内存来是可用内存增加一倍，并通过存储字节码而不是对象的方式来节约更多的内存。这样做将可以在32G的机器上使用28-30GB的内存，而不需要承受GC的问题。此外，及时重启服务，内存会保持有效，而进程内缓存将需要重建（对于10G的数据可能需要10分钟），否则需要从冷数据加载（可怕的初始化性能）。这也大大简化了代码，因为保持缓存和文件之间的一致性是由操作系统负责的，这比进程中操作更不容易出错。

这是一个简单的设计：在进程内尽量缓冲数据，空间不足时将所有数据刷写到磁盘，我们采用了相反的方式。数据并尽快写入一个持久化的日志而不需要立即刷到磁盘。实际上这只是意味着数据被转移到了内核的pagecache。

（以pagecache为中心的设计风格）

**Constant Time Suffices**

在消息系统中使用持久化数据通常是具有关联的BTree或其他随机访问的数据结构，以维护消息的元数据。BTree是最通用的数据结构，可以在消息系统中支持各种各样的语义。BTree的操作时间复杂度是O(log N)。通常O(log N)被认为是固定时间的，但是在磁盘操作中却不是。每个磁盘一次只能执行一个seek，所以并行度受到限制。因此即使少量的磁盘搜索也会导致非常高的开销。由于操作系统将快速的缓存操作和非常慢的磁盘操作相结合，所以观察到树结构的操作通常是超线性的，因为数据随固定缓存增加。

直观的，持久化队列可以像日志的解决方案一样，简单的读取和追加数据到文件的结尾。这个结构的优势是所有的操作都是O(1)的，并且读取是可以并行不会阻塞的。这具有明显的性能优势，因为性能与数据大小完全分离，可以使用低速的TB级SATA驱动器。虽然这些驱动器的搜索性能不佳，但是对于大量读写而言，他们的性能是可以接受的，并且价格是三分之一容量是原来的三倍。

无需任何的性能代价就可以访问几乎无限的磁盘空间，这意味着我们可以提供一些在消息系统中非寻常的功能。例如，在Kafka中，我们可以将消息保留较长的时间（如一周），而不是在消费后就尽快删除。这位消费者带来了很大的灵活性。

**3. Efficiency**

我们在效率上付出了很大的努力。主要的用例是处理web的数据，这个数据量非常大：每个页面可能会生成十几个写入。此外我们假设每个发布的消息至少被一个Consumer消费，因此我们尽可能使消费的开销小一些。

从构建和运行一些类似的系统的经验发现，效率是多租户操作的关键。如果下游基础服务成为瓶颈，那么应用程序的抖动将会引起问题。我们确保应用程序不会引起基础服务的Load问题，这个非常重要的，当一个集群服务上百个应用程序的时候，因为应用的使用模式的变化时非常频繁的。

我们在之前的章节中讨论过磁盘的效率。一旦不良的磁盘访问模式被消除，这种类型的系统有两个低效的原因：太多太小的IO操作和过多的数据拷贝。

太小的IO操作问题存在于客户端和服务端之间，也存在于服务端自身的持久化当中。

为了避免这个问题，我们的协议围绕“message set”抽象，通常是将消息聚合到一起。这允许网络请求将消息聚合到一起，并分摊网络往返的开销，而不是一次发送单个消息。服务端依次将大块消息追加到日志中，消费者一次线性获取一批数据。

这种简单的优化产生了一个数量级的加速。分批带来了更大的网络包，连续的磁盘操作，连续的内存块等等，这些都使得Kafka将随机消息写入转化为线性的写入并流向Consumer。

其他低效的地方是字符复制。在消息少时不是问题，但是对负载的影响是显而易见的。为了避免这种情况，我们采用被producer、broker、Consumer共享的标准的二进制消息格式（所以数据可以在传输时不需要进行修改）。

由Broker维护的消息日志本身只是一批文件，每个文件由一系列以相同格式写入的消息构成。保持相同的格式保证了最重要的优化：网络传输和持久化日志块。现在UNIX操作系统提供了高度优化的代码路径用于将pagecache的数据传输到网络；在Linux中，这有sendfile实现。

要理解sendfile的影响，了解从文件到网络传输数据的data path非常重要：

1. 操作系统从磁盘读取文件数据到pagecache，在内核空间
2. 用户从内核空间将数据读到用户空间的buffer
3. 操作系统重新将用户buffer数据读取到内核空间写入到socket中
4. 操作系统拷贝socket buffer数据到NIC buffer并发送到网络

这显然是低效的，有四个副本和两个系统调用。使用sendfile，允许操作系统直接将数据从pagecache写入到网络，而避免不必要的拷贝。在这个过程中，只有最终将数据拷贝到NIC buffer是必要的。

我们期望一个共同的场景是多个Consumer消费一个Topic数据，使用zero-copy优化，数据被拷贝到pagecache并且被多次使用，而不是每次读取的时候拷贝到内存。这允许以接近网络连接的速度消费消息。

pagecache和sendfile的组合意味着在消费者追上写入的情况下，将看不到磁盘上的任何读取活动，因为他们都将从缓存读取数据。

sendfile和更多的zero-copy背景知识见[zero-copy](https://www.ibm.com/developerworks/linux/library/j-zerocopy/)

**End-to-end Batch Compression**

在一些场景下，CPU核磁盘并不是性能瓶颈，而是网络带宽。在数据中心和广域网上传输数据尤其如此。当然，用户可以压缩它的消息而不需要Kafka的支持，但是这可能导致非常差的压缩比，因为冗余的大部分是由于相同类型的消息之间的重复（例如JSON的字段名）。**多个消息进行压缩比单独压缩每条消息效率更高。**

Kafka通过允许递归消息来支持这一点。一批消息可以一起压缩并以此方式发送到服务端。这批消息将以压缩的形式被写入日志，只能在消费端解压缩。

Kafka支持GZIP，Snappy和LZ4压缩协议。更多的压缩相关的细节在[这里](https://cwiki.apache.org/confluence/display/KAFKA/Compression)。

**4. The Producer**

**Load balancing**

Producer直接向Leader Partition所在的Broker发送数据而不需要经过任何路由的干预。为了支持Producer直接向Leader Partition写数据，所有的Kafka服务节点都支持Topic Metadata的请求，返回哪些Server节点存活的、Partition的Leader节点的分布情况。

由客户端控制将数据写到哪个Partition。这可以通过随机或者一些负载均衡的策略来实现（即客户端去实现Partition的选择策略）。Kafka暴露了一个接口用于用户去指定一个Key，通过Key hash到一个具体的Partition。例如，如果Key是User id，那么同一个User的数据将被发送到同一个分区。这样就允许消费者在消费时能够对消费的数据做一些特定的处理。这样的设计被用于处理“局部敏感”的数据（结合上面的场景，Partition内的数据是可以保持顺序消费的，那么同一个用户的数据在一个分区，那么就可以保证对任何一个用户的处理都是顺序的）。

**Asynchronous send**

批处理是提升效率的主要方式一致，为了支持批处理，Kafka允许Producer在内存聚合数据并在一个请求中发出。批处理的大小可以是通过消息数量指定的，也可以是通过等待的时间决定的（例如64K或者10ms）。这样允许聚合更多的数据后发送，减少了IO操作。缓冲的数据大小是可以配置了，这样能适当增加延迟来提升吞吐。

更多的细节可以在Producer的配合和API文档中找到。

**5 The Consumer**

Kafka Consumer通过给Leader Partition所在的Broker发送“fetch”请求来进行消费。Consumer在请求中指定Offset，并获取从指定的Offset开始的一段数据。因此Consumer对消费的位置有绝对的控制权，通过重新设置Offset就可以重新消费数据。

**Push vs Pull**

我们考虑的一个初步问题是Consumer应该从Broker拉取数据还是Broker将数据推送给Consumer。在这方面，Kafka和大多数消息系统一样，采用传统的设计方式，由Producer向Broker推送数据，Consumer从Broker上拉取数据。一些日志中心系统，如Scribe和Apache Flume，遵循数据向下游推送的方式。两种方式各有利弊。基于推送的方式，由于是由Broker控制速率，不能很好对不同的Consumer做处理。Consumer的目标通常是以最大的速率消费消息，不幸的是，在一个基于推送的系统中，当Consumer消费速度跟不上生产速度 时，推送的方式将使Consumer“过载”。基于拉取的系统在这方面做的更好，Consumer只是消费落后并在允许时可以追上进度。消费者通过某种协议来缓解这种情况，消费者可以通过这种方式来表明它的负载，这让消费者获得充分的利用但不会“过载”。以上原因最终使我们使用更为传统的Pull的方式。

Pull模型的另一个优势是可以聚合数据批量发送给Consumer。Push模型必须考虑是立即推送数据给Consumer还是等待聚合一批数据之后发送。如果调整为低延迟，这将导致每次只发送一条消息（增加了网络交互）。基于Pull的模式，Consumer每次都会尽可能多的获取消息（受限于可消费的消息数和配置的每一批数据最大的消息数），所以可以优化批处理而不增加不必要的延迟。

基于Pull模式的一个缺陷是如果Broker没有数据，Consumer可能需要busy-waiting的轮训方式来保证高效的数据获取（在数据到达后快速的响应）。为了避免这种情况，我们在Pull请求中可以通过参数配置“long poll”的等待时间，可以在服务端等待数据的到达（可选的等待数据量的大小以保证每次传输的数据量，减少网络交互）。

你可以想象其他一些从端到端，采用Pull的可能的设计。Producer把数据写到本地日志，Broker拉取这些Consumer需要的数据。一个相似的被称为“store-and-forward”的Producer经常被提及。这是有趣的，但是我们觉得不太适合我们可能会有成千上万个Producer的目标场景。我们维护持久化数据系统的经验告诉我们，在系统中使多应用涉及到上千块磁盘将会使事情变得不可靠并且会使操作它们变成噩梦。最后再实践中，我们发现可以大规模的运行强大的SLAs通道，而不需要生产者持久化。

**Consumer Position**

记录哪些消息被消费过是消息系统的关键性能点。

大多数消息系统在Broker上保存哪些消息已经被消费的元数据。也就是说，Broker可以在消费传递给Consumer后立即记录或等待消费者确认之后记录。这是一个直观的选择，并且对于单个服务器而言并没有更好的方式可以存储这个状态。大多数消息系统中的存储设备并不能很好的伸缩，所以这也是务实的选择——当Broker确认消息被消费后就立即删除，以保证存储较少的数据。

让Broker和Consumer关于那些消息已经被消费了达成一致并不是一个简单的问题。如果Broker在将消息写到网络之后就立即认为消息已经被消费，那么如果Consumer消费失败（Consumer在消费消息之前Crash或者网络问题等）消息将丢失。为了解决这个问题，一些消息系统增加了ACK机制，消息被标记为只是发送出去而不是已经被消费，Broker需要等待Consumer发送的ACK请求之后标记具体哪些消息已经被消费了。这个策略修复了消息丢失的问题，但是引起了新的问题。第一，如果Consumer处理了消息，但是在发送ACK给Broker之前出现问题，那么消息会被重复消息。第二，Broker需要维护每一条消息的多个状态（是否被发送、是否被消费）。棘手的问题是要处理被发送出去但是没有被ACK的消息。

Kafka采用不同的方式处理。Topic被划分为多个内部有序的分区，每个分区任何时刻只会被一个group内的一个Consumer消费。这意味着一个Partition的Position信息只是一个数字，标识下一条要消费的消息的偏移量。这使得哪些消息已经被消费的状态变成了一个简单的数据。这个位置可以定期做CheckPoint。这使得消息的ACK的代价非常小。

这个方案还有其他的好处。消费者可以优雅的指定一个旧的偏移量并重新消费这些数据。这和通常的消息系统的观念相违背，但对很多消费者来说是一个很重要的特性。比如，如果Consumer程序存在BUG，在发现并修复后，可以通过重新消费来保证数据都正确的处理。

**Offline Data Load**

可扩展的持久化存储的能力，是消费者可以定期的将数据导入到像Hadoop这样的离线系统或关系型数据仓库中。

在Hadoop的场景中，我们通过把数据分发到独立的任务中进行并行处理，按照node/topic/partition组合，充分使用并行能力加载数据。Hadoop提供任务管理，失败的任务可以重新启动，而不需要担心重复数据的危险——任务会从原始位置重新启动。

**6. Message Delivery Semantics**

现在我们对Producer和Consumer已经有了一定的了解，接着我们来讨论Kafka在Producer和Consumer上提供的语义。显然的，在分发消息时是可以有多种语义的：

* At most once：消息可能丢失，但不会重复投递
* At least once：消息不会丢失，但可能会重复投递
* Exactly once：消息不丢失、不重复，会且只会被分发一次（真正想要的）

值得注意的是这分为两个问题：发布消息的可用性和消费消息的可用性。

许多系统都声称提供“exactly once”语义，仔细阅读会发现，这些声明是误导的（他们没有考虑Producer和Consumer可能Crash的场景，或是数据写入磁盘后丢失的情况）。

Kafka提供的语义是直接了当的。发送消息的时候我们有一个消息被Commit到Log的概念。一旦消息已经被Commit，它将不会丢失，只要还有一个复制了消息所在Partition的Broker存活着。“存活”的定义以及我们覆盖的失败的情况将在下一节描述。现在假设一个完美的Broker，并且不会丢失，来理解对Producer和Consumer提供的语义保证。如果Producer发送一条消息，并且发生了网络错误，我们是不能确认错误发生在消息Commit之前还是消息Commit之后的。类似于使用自增主键插入数据库，是不能确认写入之后的主键值的。

Producer没有使用的强制可能的语义。我们无法确认网络是否会发生异常，可以使Producer创建有序的主键使重试发送成为幂等的行为。这个特性对一个复制系统来说不是无价值的，因为服务器在发生故障的情况下依旧需要提供服务。使用这个功能，Producer可以重试，直到收到消息成功commit的响应，在这个点上保证消息发送的exactly once。我们希望把这个特性加到后续的Kafka版本中。

不是所有的场景都需要这样的保证。对应延迟敏感的场景，我们允许Producer指定其期望的可用性级别。如果Producer期望等待消息Commit，那么这可能消耗10ms。Producer也可以指定以异步的方式发送消息或只等Leader节点写入消息（不能Follower）。

接着我们从消费者的视角来描述语义。所有的副本(Partition)都拥有偏移量相同的日志。Consumer控制它在日志中的偏移量。如果Consumer一直正常运行，它可以只把偏移量存储在内存中，但是如果Consumer crash且我们期望另一个新的Consumer接管消费，那么需要选择一个位置来开始消费。假设Consumer读取了一些消息——它有集中处理消息和位置的方式。

它可以读取消息，然后保存位置信息，然后处理消息。在这个场景中，Consumer可能在保存位置信息后消费消息失败，那么下一次消费可能从保存的位点开始，尽管之前部分消息被处理失败。这是消费处理过程中失败的at-most-once（只被处理了一次，但是可能处理失败）。

它可以读取消息，之后处理消息，最后保存位置信息。这个场景中，Consumer可能在处理完消息，但是保存位点之前Crash，那么下一次会重新消费这些消息，尽管已经被消费过。这是Consumer Crash引起的at-least-once（消息可能会被处理多次）。

在很多场景冲，消息可以有一个逐渐，这样可以保证处理的幂等性（多次处理不会有影响）。

那么什么是exactly once语义？这里的限制实际上不是消息系统的特性，而是消息处理和位置信息的保存。经典的解决方案是采用两阶段提交的方式来处理。但是这也可以用一个更简单的方式来处理：通过将消息处理结果和位置信息保存在同一位置上。这是更好的，因为很多Consumer期望写入的系统并不支持两阶段提交。例如, 我们的Hadoop ETL工具从保存数据到hdfs上的同时也把位移位置也保存到hdfs中了, 这样可以保证数据和位移位置同时被更新或者都没更新.我们在很多系统上使用类似的模式, 用于解决那些需要这种强语义但是却没有主键用于区分重复的储存系统中.

默认Kafka提供at-least-once语义的消息分发，允许用户通过在处理消息之前保存位置信息的方式来提供at-most-once语义。exactly-once语义需要和输出系统像结合，Kafka提供的offset可以使这个实现变的“直接了当的”（变得比较简单）。

**7. Replication**

Kafka为Topic的每个Partition日志进行备份，备份数量可以由用户进行配置。这保证了系统的自动容错，如果有服务器宕机，消息可以从剩余的服务器中读取。

其他消息系统提供了备份相关的功能，但在我们看来，这是一个附加的功能，不能被大量使用，并且伴随着大量的缺点：Slave是不活跃的（这里应该是指Slave只提供了备份，并不可以被消费等等）、吞吐受到很大的影响、需要手动配置等等。在Kafka中，我们默认就提供备份，实际上我们认为没有备份的Topic是一种特殊的备份，只是备份数为1。

备份的单位是Topic的分区。在没有发生异常的情况下，Kafka中每个分区都会有一个Leader和0或多个Follower。备份包含Leader在内（也就是说如果备份数为3，那么有一个Leader Partition和两个Follower Partition）。所有的读写请求都落在Leader Partition上。通常情况下分区要比Broker多，Leader分区分布在Broker上。Follower上的日志和Leader上的日志相同，拥有相同的偏移量和消息顺序（当然，在特定时间内，Leader上日志会有一部分数据还没复制到Follower上）。

Follower作为普通的Consumer消费Leader上的日志，并应用到自己的日志中。Leader允许Follower自然的，成批的从服务端获取日志并应用到自己的日志中。

大部分分布式系统都需要自动处理故障，需要对节点“alive”进行精确的定义。例如在Kafka中，节点存活需要满足两个条件：

1. 节点需要保持它和ZooKeeper之间的Session（通过ZK的心跳机制）
2. 如果是Follower，需要复制Leader上的写事件，并且复制进度没有“落后太多”

我们称满足这两个条件的节点为“同步的”来避免使用“alive”或“failed”这样模糊的概念。Leader节点保存同步中的Follower节点。如果一个Follower宕机或复制落后太多，Leader将从同步的Follower List中将其移除。通过replica.lag.time.max.ms配置来定义“落后太多”。

在分布式系统的术语中，我们只尝试处理“失败/恢复”模型——节点突然停止工作之后恢复的情况。Kafka不处理“拜占庭”问题。

一条消息在被应用到所有的备份上之后被认为是“已经提交的”。只有提交了的消息会被Consumer消费。这意味着Consumer不需要担心Leader节点宕机后消息会丢失。另一方面，Producer可以配置是否等待消息被提交，这取决于他们在延迟和可用性上的权衡。这个可以通过Producer的配置项中设置。

Kafka提供了一条消息被提交之后，只要还有一个备份可用，消息就不会丢失的保证。

Kafka保证在节点故障后依旧可用，但是无法再网络分区的情况下保持可用。

**Replicated Logs: Quorums, ISRs, and State Machines (Oh my!)**

Kafka分区机制的核心是日志复制。日志复制是分布式系统中最基础的东西，有很多方式可以实现。日志复制可以作为基于状态机的分布式系统的基础设置。

日志复制模型用于处理连续、有序的输入（例如给log entry添加0、1、2这样的编号）。有很多方式实现日志复制，最简单的方式是Leader选择和提供这个顺序之。只要Leader节点存活，Follower只需要按照Leader选择的值和顺序来复制即可。

当然，如果Leader不会宕机，那我们也不需要Follower了！在Leader宕机之后，我们需要在Follower中选择一个节点成为新的Leader。Follower可能会宕机或者日志落后较多，所以我们必须确保选择一个“及时同步”（复制进度和Leader最近的节点）成为新的Leader。复制算法必须提供这样的保证：如果Client收到一条消息已经被Commit了，如果Leader宕机，新Leader必须包含这条已经被Commit的消息。这是一个权衡：Leader在确认消息Commit之前需要等待更多的Follower来确认复制了消息来保证在Leader宕机后有更多可以成为Leader的Follower节点。

如果你选择了所需要的ACK的数量以及选择Leader时需要比较的日志数以确保能重合，这个叫做Quorum。

一个通用的来权衡的方式是提交日志和选择Leader时都采用大多数投票的原则。这不是Kafka使用的方式，但是无所谓，让我们去理解这种方式来了解实现原理。假设一共有2f+1个备份，那么f+1的副本必须在Leader提交commit之前接收到消息，这样就可以从f+1个节点中选择出新节点作为Leader。因为任何f+1个节点，必然有一个节点包含最全的日志。还有很多关于这个算法的细节需要处理（如何定义日志更全、在Leader节点宕机时保持日志一致性等）在这里先忽略。

大多数选票的方法有非常好的特性：延迟取决于同步最快的Server节点。这说明，如果备份数为3，那么延迟取决于两个备份节点中较快的节点。

有很多类似的算法变体，例如ZooKeeper的Zab，Raft,Viewstamped Replication等。和Kafka最相似的学术刊物是微软的PacificA。

大多数选票方式的取消是它不能容忍很多的故障，导致你没有可以被选为新Leader的节点。为了容忍一个节点故障，需要3分数据备份，容忍两个节点故障则需要5个节点。在我们的经验中，只有足够的冗余才能容忍单一的故障在实际系统中是不够的，每次写5次副本，使用5倍的存储空间，和1/5的带宽，在大体量的数据存储上不是很可行。这就是为什么quorum算法多多应用在像ZK这样存储配置的集群中，而不是数据存储系统中。例如HDFS的namenode的高可用建立在大多数选票的机制上，但是数据存储却不是。

Kafka使用一个明显不同的方式来选择quorum集合。代替大多数选票，Kafka动态的维护一个“同步的备份（in-sync replicas ISR）”的集合。只有这个集合中的成员能被选举为Leader。一个写入请求需要同步到所有的同步中的备份才能认为是提交的。ISR集合在变更时会被持久化到ZK。因此，任何ISR中的备份都可以被选举为新的Leader。这对于Kafka这种拥有多分区并且需要保证这节点负载均衡的模型来说非常重要。使用ISR模型和f+1个副本，Kafka可以容忍f个备份不可用的情况。

对于大多数的场景，我们认为这样的妥协是合理的。在实践中，为了容忍f个节点故障，大多数选票原则和ISR方式都需要等待相同的备份在提交消息前进行确认（如需要容忍一个节点故障，大多数选票的选择需要3个节点，并且提交消息需要至少一个备份的确认；ISR只需要两个节点，需要确认的副本数一样是一个）。相对于大多数选票的原则，ISR方式不需要等待最慢的服务器确认消息是一个优势。尽管如此，我们进行改善，让客户端决定是否等待消息提交，使用较小的副本数，这样带来的吞吐和更小的磁盘空间要求是有价值的。

另一个重要的设计是Kafka不需要故障的节点恢复所有的数据。这是不常见的，复制算法依赖于存储介质在任何故障的情况下都不丢失数据并且不违反一致性原则。这个假设有两个主要的问题。第一，磁盘故障是持久化数据系统中最常见的问题，并且它通常导致数据不完整。第二，即使这不是一个问题，我们也不希望在每一次写入之后都使用fsync来保证一致性，这会使性能下降两三个数量级。我们的协议中允许一个副本重新加入到ISR集合中，在重新加入之前，它需要重新同步在故障时丢失的数据。

**Unclean leader election: What if they all die?**

Kafka保证的数据不丢失，在至少有一个备份保持同步的情况下。如果一个分区所有的备份的节点都故障，那么就不能提供这个保障了。

但是实践系统中需要一些合理的事情，在所有备份故障时。如果不巧遇上这个问题，去考虑哪些情况会发生是非常重要的。有两种方式去做：

1. 等待一个ISR中的副本恢复并将其选举为新的Leader（期望它拥有所有的数据）。
2. 选择第一个副本（无需在ISR中）作为Leader。

这是在可用性和一致性之间的权衡。如果我们等待ISR中的备份恢复，那么会在这个期间一直不可用。如果这样的副本被损坏，那么我们将永久性的失效。另一方便，如果使用不在ISR中的备份成为Leader，尽管它可能不包含所有的日志。默认情况下，Kafka使用第二种策略，当所有ISR中的备份不可用时，倾向于选择可能不一致的备份。这个方式可以通过unclean.leader.election.enable配置禁用，在哪些停机时间优于不一致的场景。

这种困境不是kafka特有的, 这存在于任何基于quorum方式的结构中. 例如, 多数投票算法, 如果大多数的服务器都永久性失效了, 你必须选择丢失全部的数据或者接受某一台可能数据不一致的服务器上的数据.

**Availability and Durability Guarantees**

在向Kafka写入数据时，Producer可以选择是否等待0，1或（-1）个备份响应。注意，这里说的“被所有备份响应”不是说被所有分配的备份响应，默认情况下只的时所有ISR集合中的备份响应。例如，如果一个Topic配置成只需要两个备份，并且一个备份故障了，那么写入一个备份即认为收到了所有响应。但是，如果这个备份也故障了，那么数据会丢失。这样保证了分区的最大可用，但是可能不是那些相对于可用性更需要可靠性的用户的需求。因此，我们提供两种Topic级别的配置，相对于可用性，优先保证可靠性：

1. 禁用unclean leader election；如果所有备份不可用，那么分区保持不可用，直到最近的Leader重新恢复可用。这可能导致不可用，但是不会丢失数据。
2. 配置一个最小的ISR大小；分区只会在满足最小ISR的情况下接受请求，这样可以避免数据只写入一个备份，而这个备份后续故障导致数据丢失。这个配置只在Producer使用acks=all的配置时有效。这个配置在一致性和可用性上做了权衡。更大的ISR提供了更好的一致性，但是降低了可用性，如果同步备份数小于最小ISR配置时将不可用。

**Replica Management**

以上的讨论都是基于一个日志，即一个Topic的分区考虑的。但是Kafka集群拥有成百上千这样的分区。我们尝试使用轮训的方式来平衡分区，避免高数量的Topic的分区集中在一部分少量的节点上。同样我们要平衡所有Leader分区，这样每个节点上承载的主分区都有一定的比例。

优化Leader的选举过程也是非常重要的，因为这是系统不可用的窗口期。一个直观的实现是，如果一个节点故障了，为这个节点上所有的分区都独立的执行一次选举。代替这种方式，我们选择一个Broker作为Controller，Controller负责一个故障节点影响的所有分区的Leader变更。这样的好处是我们可以批量处理，减少独立选举时大量的通知，这使得大量分区需要选举时变得更快，代价更小。如果Controller故障了，剩余的Broker中会有一个节点成为新的Controller。

**8 Log Compaction**

日志压缩确保Kafka会为一个Topic分区数据日志中保留至少message key的最后一个值。它解决了应用crash或系统故障或应用在操作期间重启来重新加载缓存的场景。让我们深入到细节中解释日志压缩是如何工作的。

到屋面位置，我们只说明了在一段时间或达到特定大小的时候丢弃旧日志的简单方法。这适用于像日志这样每一条数据都是独立数据的情况。但是重要类别的数据是根据key处理的数据（例如DB中表的变更数据）。

让我们来讨论这样一个具体的流的例子。一个Topic包含了用户email address信息；每一次用户变更邮箱地址，我们都向这个topic发送一条消息，使用用户ID作为primay key。现在我们已经为用户ID为123的用户发送了一些消息，每条消息包含了email address的变更：

123 => bill@microsoft.com

123 => bill@gatesfoundation.org

123 => bill@gmail.com

日志压缩为我们提供了更精细的保留机制，至少保存每个key最后一个变更（如123 => bill@gmail.com）。这样做我们确保了这个日志包含了所有key最后一个值的快照。这样Consumer可以重建状态而不需要保留完成的变更日志。

让我们列一些日志压缩有用的场景，然后看他是如果被使用的。

1. DB变更订阅。这是很常见的，一个数据在多个数据系统中，而且其中一个系统是数据库类型的（如RDBMS或KV系统）。例如可能有一个数据库，一个户缓存系统，一个搜索集群，一个Hadoop集群。DB的任何一个变更需要反映到缓存、搜索集群，最终保存到Hadoop中。在这个场景中，你只需要实时系统最新的更新日志。但是如果需要重新加载缓存或恢复宕机的检索节点，就需要完整的数据。
2. 事件源。这是一种应用设计风格，它将查询处理和应用程序设计结合到一起，并使用日志作为程序的主要存储。
3. 高可用日志。一个本地集成程序可以通过变更日志来做到容错，这样另一个程序能够在当前程序故障时继续处理。例如, 像流数据查询例子, 如计数, 汇总或其他的分组操作. 实时系统框架如Samza, 就是为了达到这个目的使用这个特性的。

在这些场景中，主要处理实时的变更，但有时需要重新加载或重新处理时，需要加载所有数据。日志压缩允许使用相同的Topic来支持这些场景，这种日志使用风格在后续的内容中会更详细的描述。

想法很简单，我们有有无限的日志，以上每种情况记录变更日志，我们从一开始就捕获每一次变更。使用这个完整的日志，我们可以通过回放日志来恢复到任何一个时间点的状态。这种假设的情况下，完整的日志是不实际的，对于那些每一行记录会变更多次的系统，即使数据集很小，日志也会无限的增长下去。丢弃旧日志的简单操作可以限制空间的增长，但是无法重建状态——因为旧的日志被丢弃，可能一部分记录的状态会无法重建（这写记录所有的状态变更都在就日志中）。

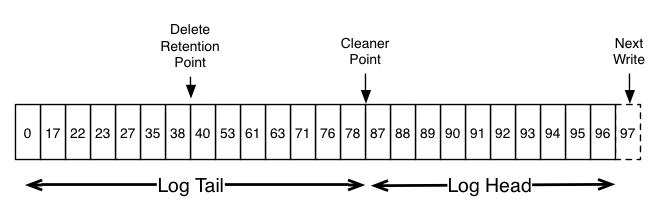
日志压缩机制是更细粒度的，每个记录都保留的机制，而不是基于时间的粗粒度。这个想法是选择性的删除哪些有更新的变更的记录的日志。这样最终日志至少包含每个key的记录的最后一个状态。

这个策略可以为每个Topic设置，这样一个集群中，可以一部分Topic通过时间和大小保留日志，另外一些可以通过压缩保留。

这个功能的灵感来自于LinkedIn的最古老且最成功的基础设置——一个称为Databus的数据库变更日志缓存系统。不像大多数的日志存储系统，Kafka为了订阅而量身打造，用于线性的快速读写。和Databus不同，Kafka作为真实的存储，压缩日志是非常有用的，在上有数据源不能重放的情况下。

**Log Compaction Basics**

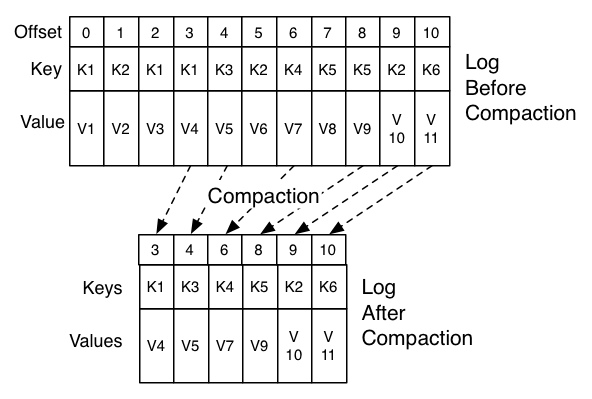
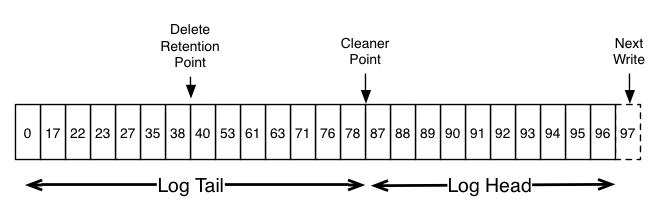
这里是一个展示Kafka日志的逻辑结构的图（每条消息包含了一个offset）：



Log head中包含传统的Kafka日志。它包含了连续的连续的offset和所有的消息。日志压缩增加了处理tail Log的选项。上图展示了日志压缩的的Log tail的情况。tail中的消息保存了初次写入时的offset。即使该offset的消息被压缩，所有offset仍然在日志中是有效的。在这个场景中，无法区分和下一个出现的更高offset的位置。如上面的例子中，36、37、38是属于相同位置的，从他们开始读取日志都将从38开始。

压缩允许删除。一条消息伴随着空的值被认为从日志中删除。这个删除标记将会引起所有之前拥有相同key的消息被移除（包括拥有key相同的新消息），但是删除标记比较特殊，它将在一定周期后被从日志中删除来示范空间。这个时间点被称为“delete retention point”。

压缩操作通过在后台周期性的拷贝日志段来完成。清除操作不会阻塞读取，并且可以被配置不超过一定IO吞吐来避免影响Producer和Consumer。实际的日志段压缩过程有点像如下：



**What guarantees does log compaction provide?**

日志压缩提供了如下的保证：

1. 所有跟上消费的Consumer能消费到所有写入的消息；这些消息有连续的序列号。Topic的min.compaction.lag.ms可以用于保证消息写入多久后才会被压缩。这限制了一条消息在Log Head中的最短存在时间。
2. 消息的顺序会被保留。压缩不会重排序消息，只是移除其中一部分。
3. 消息的Offset不会变更。这是消息在日志中的永久标志。
4. 任何从头开始处理日志的Consumer至少会拿到每个key的最终状态。另外，只要Consumer在小于Topic的delete.retention.ms设置（默认24小时）的时间段内到达Log head，将会看到所有删除记录的所有删除标记。换句话说，因为移除删除标记和读取是同事发生的，Consumer可能会因为落后超过delete.retention.ms而导致错过删除标记。

**Log Compaction Details**

日志压缩由Log Cleaner执行，后台线程池重新拷贝日志段，移除那些key存在于Log Head中的记录。每个压缩线程如下工作：

1. 选择Log Head相对于Log Head在日志中占更高比例的日志
2. 创建Log Head中每个Key最后一个offset的摘要
3. 从头到尾的拷贝日志，并删除之后日志终于到相同key的记录。新的、干净的日志将会立即被交到到日志中，所以只需要一个额外的日志段空间
4. Log Head的摘要实际上是一个空间紧凑的哈希表。每个条目使用24个字节。所以如果有8G的整理缓冲区, 则能迭代处理大约366G的日志头部(假设消息大小为1k)。

**Configuring The Log Cleaner**

Log Cleaner默认启用。这会启动清理的线程池。如果要开始特定Topic的清理功能，可以开启特定的属性：

log.cleanup.policy=compact

这个可以通过创建Topic时配置或者之后使用Topic命令实现。

Log Cleaner可以配置保留最小的不压缩的日志头。可以通过配置压缩的延迟时间：

log.cleaner.min.compaction.lag.ms

这可以用于保证消息比在被压缩的消息大一段时间。如果没有设置，除了最后一个日志外，所有的日志都会被压缩。当前写入的自如端不会被压缩，即使所有的消息都落后于比配置的最小压缩时间。

更多的配置在[这里](http://kafka.apache.org/documentation.html" \l "brokerconfigs)

**9 Quotas**

从0.9版本开始，Kafka可以对生产和消费请求进行限额配置。基于字节速率来限制，每个group中所有的客户端共享一个限额。

**Why are quotas necessary?**

Producer和Consumer可能生产或消费大量的数据而耗尽Broker的资源，导致网络饱和。进行限额可以避免这些问题，特别是在多租户的集群中，一小部分低质量的客户端会降低整个集群的体验。实际上，当运行Kafka作为服务时，这还可以对API的使用进行限制。

**Client groups**

Kafka客户端的身份代表了用于鉴权。 在无鉴权机制的集群中, 用户身份是由服务器使用可配置的PrincipalBuilder进行选择的, Client-id作为客户端逻辑分组, 是由客户端应用选择的一个有意义的名称. 标量(user, client-id)定义共享这个用户身份和客户端ID的逻辑客户端分组.

配额可以用于(user, client-id)组合, 或user, client-id分组。

对一个给定的连接, 最符合这个连接的配额被使用到, 一个限额组的所有连接共享这个限额配置, 例如: 如果(user=”test-user”, client-id=”test-client”) 10MB/s的配额, 这个配置会被所有的具有”test-user”用户和客户端ID是 “test-client”的所有生产者所共享.

**Quota Configuration**

配额可以按照(user, client-id)或者, user或client-id进行分组, 如果需要更高或更低的配额, 可以覆盖默配额, 这个机制类似于对日志主题配置的覆盖, user 或者 (user, client-id)配额可以覆盖写入到zookeeper下的 /config/users ,client-id配置, 可以写入到 /config/clients。这些覆盖写入会被服务器很快的读取到, 这让我们修改配置不需要重新启动服务器. 每个分组的默认配置也可以同样的方式动态修改。

限额的配置顺序如下:

1. /config/users//clients/
2. /config/users//clients/
3. /config/users/
4. /config/users//clients/
5. /config/users//clients/
6. /config/users/
7. /config/clients/
8. /config/clients/

Broker的quota.producer.default，quota.consumer.default也可以用来配置默认的client-id分组的默认值。这可属性已经不鼓励使用，后续将会删除。默认client-id限额配置可以和其它默认配置一样, 在Zookeeper直接设置。

**Enforcement**

默认情况下，每个唯一的客户端group会收到一个集群配置的固定的限额。这个限额是基于每个Broker的。每个客户端能发布或获取在每台服务器都的最大速率, 我们按服务器定义配置, 而不是按整个集群定义,是因为如果是集群范围的需要额外的机制来共享配额的使用情况, 这会导致配额机制的实现比较难。

Broker检测到限额违规时时如何处理的？在我们的解决方案中，Broker不会返回错误给客户端，而是降低客户端的速率。Broker计算使客户端回到合理限额的需要的响应延迟。这种方法的处理对客户端是透明，使他们不必执行任何棘手的，特殊的操作。实际上，错误的客户端还可能加剧正在解决的限额问题。

客户端字节率在多个小窗口（例如每个1秒的30个窗口）上进行测量，以便快速检测和纠正配额违规。 通常，具有大的测量窗口（例如，每个30秒的10个窗口）导致大量的流量脉冲，随后是长时间的延迟，这在用户体验方面不是很好。