注：本文档来自
http://blog.csdn.net/derekjiang/article/details/9053863/

pdf 制作：elancom

版本：1.0

Kafka 中文文档

转自：http://www.oschina.net/translate/kafka-design

参与翻译(4 人)：fbm, 飞翔的猴子, Khiyuan, nesteaa

感谢这些同志们的辛勤工作，翻译的真不错，目前见到的最好的 Kafka 中文文章

我们为什么要搭建该系统

Kafka 是一个消息系统，原本开发自 LinkedIn，用作 LinkedIn 的活动流（activity stream）和运营数据处理管道（pipeline）的基础。现在它已为多家不同类型的公司 作为多种类型的数据管道（data pipeline）和消息系统使用。

活动流数据是所有站点在对其网站使用情况做报表时要用到的数据中最常规的部分。活动数据包括页面访问量（page view）、被查看内容方面的信息以及搜索情况等内容。这种数据通常的处理方式是先把各种活动以日志的形式写入某种文件，然后周期性地对这些文件进行统计 分析。运营数据指的是服务器的性能数据（CPU、IO 使用率、请求时间、服务日志等等数据）。运营数据的统计方法种类繁多。

近年来，活动和运营数据处理已经成为了网站软件产品特性中一个至关重要的组成部分，这就需要一套稍微更加复杂的基础设施对其提供支持。

活动流和运营数据的若干用例

- "动态汇总（News feed）"功能。将你朋友的各种活动信息广播给你
- 相关性以及排序。通过使用计数评级（count rating）、投票（votes）或者点击率（ click-through）判定一组给定的条目中那一项是最相关的.
- 安全：网站需要屏蔽行为不端的网络爬虫（crawler），对 API 的使用进行速率限制，探测出扩散垃圾信息的企图，并支撑其它的行为探测和预防体系，以切断网站的某些不正常活动。
- 运营监控：大多数网站都需要某种形式的实时且随机应变的方式，对网站运行效率进行监控并在有问题出现的情况下能触发警告。
- 报表和批处理: 将数据装载到数据仓库或者 Hadoop 系统中进行离线分析，然后针对业务行为做出相应的报表，这种做法很普遍。

活动流数据的特点

这种由不可变（immutable）的活动数据组成的高吞吐量数据流代表了对计算能力的一种真正的挑战，因其数据量很容易就可能会比网站中位于第二位的数据源的数据量大 10 到 100 倍。

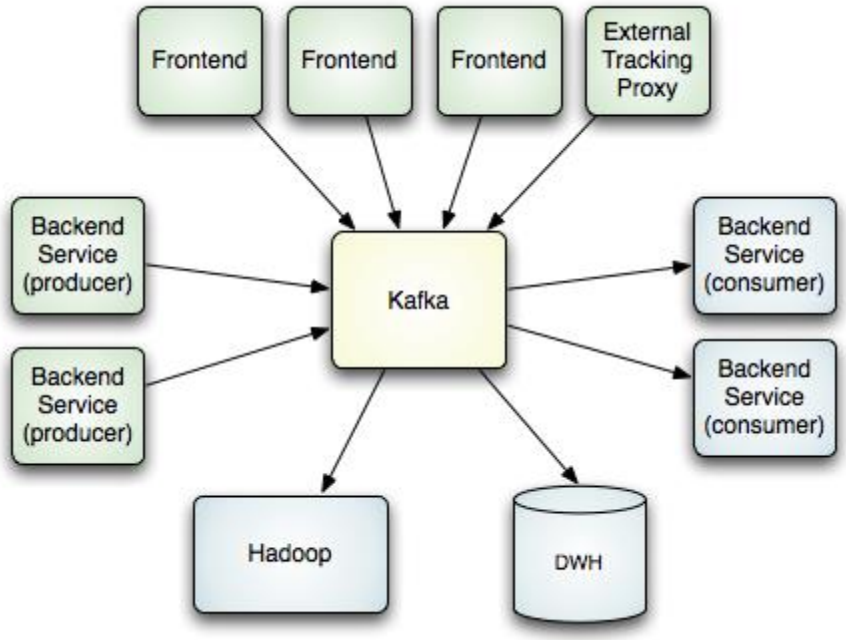
传统的日志文件统计分析对报表和批处理这种离线处理的情况来说，是一种很不错且很有伸缩性的方法，但是这种方法对于实时处理来说其时延太大，而且还具有较 高的运营复杂度。另一方面，现有的消息队列系统（messaging and queuing system）却很适合于在实时或近实时（near-real-time）的情况下使用，但它们对很长的未被处理的消息队列的处理很

不给力，往往并不将数 据持久化作为首要的事情考虑。这样就会造成一种情况，就是当把大量数据传送给 Hadoop 这样的离线系统后， 这些离线系统每个小时或每天仅能处理掉部分源数据。Kafka 的目的就是要成为一个队列平台，仅仅使用它就能够既支持离线又支持在线使用这两种情况。

Kafka 支持非常通用的消息语义（messaging semantics）。尽管我们这篇文章主要是想把它用于活动处理，但并没有任何限制性条件使得它仅仅适用于此目的。

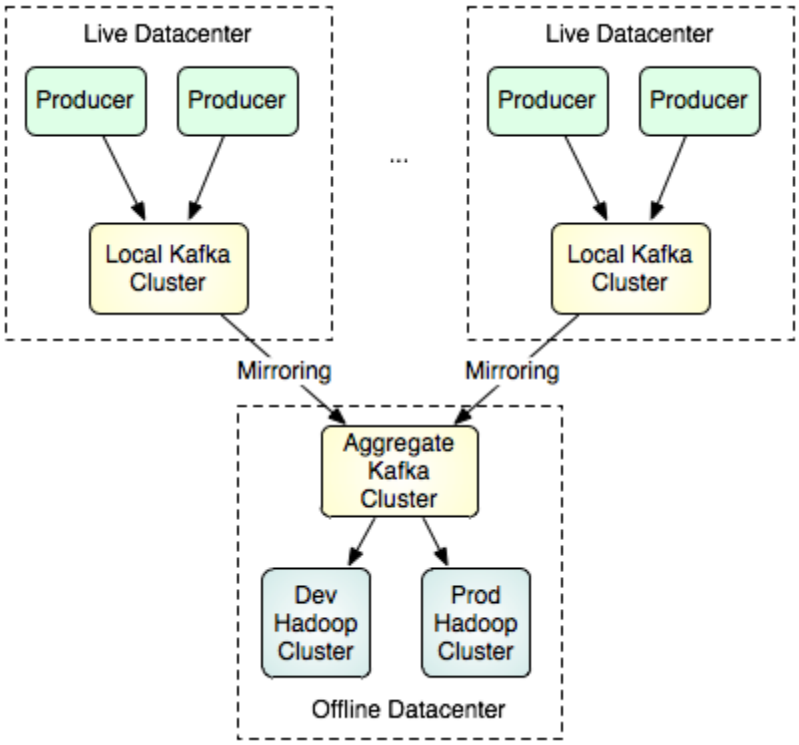
部署

下面的示意图所示是在 LinkedIn 中部署后各系统形成的拓扑结构。



要注意的是，一个单个的 Kafka 集群系统用于处理来自各种不同来源的所有活动数据。它同时为在线和离线的数据使用者提供了一个单个的数据管道，在线活动 和异步处理之间形成了一个缓冲区层。我们还使用 kafka，把所有数据复制（replicate）到另外一个不同的数据中心去做离线处理。

我们并不想让一个单个的 Kafka 集群系统跨越多个数据中心，而是想让 Kafka 支持多数据中心的数据流拓扑结构。这是通过在集群之间进行镜像或“同步” 实现的。这个功能非常简单，镜像集群只是作为源集群的数据使用者的角色运行。这意味着，一个单个的集群就能够将来自多个数据中心的数据集中到一个位置。下 面所示是可用于支持批量装载（batch loads）的多数据中心拓扑结构的一个例子：



请注意，在图中上面部分的两个集群之间不存在通信连接，两者可能大小不同，具有不同数量的节点。下面部分中的这个单个的集群可以镜像任意数量的源集群。要了解镜像功能使用方面的更多细节，请访问这里.

主要的设计元素

Kafka 之所以和其它绝大多数信息系统不同，是因为下面这几个为数不多的比较重要的设计决策：

1. Kafka 在设计之时为就将持久化消息作为通常的使用情况进行了考虑。
2. 主要的设计约束是吞吐量而不是功能。
3. 有关哪些数据已经被使用了的状态信息保存为数据使用者（consumer）的一部分，而不是保存在服务器之上。
4. Kafka 是一种显式的分布式系统。它假设，数据生产者（producer）、代理（brokers）和数据使用者（consumer）分散于多台机器之上。

以上这些设计决策将在下文中进行逐条详述。

基础知识

首先来看一些基本的术语和概念。

消息指的是通信的基本单位。由消息生产者（producer）发布关于某话题（topic）的消息，这句话的意思是，消息以一种物理方式被发送给了作为代理（broker）的服务器（可能是另外一台机器）。若干的消息使用者（consumer）订阅（subscribe）某个话题，然后生产者所发布的每条消息都会被发送给所有的使用者。

Kafka 是一个显式的分布式系统 —— 生产者、使用者和代理都可以运行在作为一个逻辑单位的、进行相互协作的集群中不同的机器上。对于代理和生产者，这么做非常自然，但使用者却需要一些特殊的支持。每个使用者进程都属于一个使用者小组（consumer group）。准确地讲，每条消息都只会发送给每个使用者小组中的一个进程。因此，使用者小组使得许多进程或多台机器在逻辑上作为一个单个的使用者出现。使用者小组这个概念非常强大，可以用来支持 JMS 中队列（queue）或者话题（topic）这两种语义。为了支持队列 语义，我们可以将所有的使用者组成一个单个的使用者小组，在这种情况下，每条消息都会发送给一个单个的使用者。为了支持话题语 义，可以将每个使用者分到它自己的使用者小组中，

随后所有的使用者将接收到每一条消息。在我们的使用当中，一种更常见的情况是，我们按照逻辑划分出多个使 用者小组，每个小组都是有作为一个逻辑整体的多台使用者计算机组成的集群。在大数据的情况下，Kafka 有个额外的优点，对于一个话题而言，无论有多少使用者订阅了它，一条条消息都只会存储一次。

消息持久化（Message Persistence）及其缓存

不要害怕文件系统！

在对消息进行存储和缓存时，Kafka 严重地依赖于文件系统。 大家普遍认为“磁盘很慢”，因而人们都对持久化结(persistent structure)构能够提供说得过去的性能抱有怀疑态度。实际上，同人们的期望值相比，磁盘可以说是既很慢又很快，这取决于磁盘的使用方式。设计的很 好的磁盘结构往往可以和网络一样快。

磁盘性能方面最关键的一个事实是，在过去的十几年中，硬盘的吞吐量正在变得和磁盘寻道时间严重不一致了。结果，在一个由 6 个 7200rpm 的 SATA 硬盘 组成的 RAID-5 磁盘阵列上，线性写入（linear write）的速度大约是 300MB/秒，但随即写入却只有 50k/秒，其中的差别接近 10000 倍。线性读取和写入是所有使用模式中最具可预计性的一种 方式，因而操作系统采用预读（read-ahead）和后写（write-behind）技术对磁盘读写进行探测并优化后效果也不错。预读就是提前将一个 比较大的磁盘块中内容读入内存，后写是将一些较小的逻辑写入操作合并起来组成比较大的物理写入操作。关于这个问题更深入的讨论请参考这篇文章 ACM Queue article；实际上他们发现，在某些情况下，顺序磁盘访问能够比随即内存访问还要快！

为了抵消这种性能上的波动，现代操作系变得越来越积极地将主内存用作磁盘缓存。所有现代的操作系统都会乐于将所有空 闲内存转做磁盘缓存，即时在需要回收这些内存的情况下会付出一些性能方面的代价。所有的磁盘读写操作都需要经过这个统一的缓存。想要舍弃这个特性都不太容 易，除非使用直接 I/O。因此，对于一个进程而言，即使它在进程内的缓存中保存了一份数据，这份数据也可能在 OS 的页面缓存(pagecache)中有重 复的一份，结构就成了一份数据保存了两次。

更进一步讲，我们是在 JVM 的基础之上开发的系统，只要是了解过一些 Java 中内存使用方法的人都知道这两点：

1. Java 对象的内存开销（overhead）非常大，往往是对象中存储的数据所占内存的两倍（或更糟）。
2. Java 中的内存垃圾回收会随着堆内数据不断增长而变得越来越不明确，回收所花费的代价也会越来越大。
3.

由 于这些因素，使用文件系统并依赖于页面缓存要优于自己在内存中维护一个缓存或者什么别的结构 —— 通过对所有空闲内存自动拥有访问权，我们至少将可用的缓存大小翻了一倍，然后通过保存压缩后的字节结构而非单个对象，缓存可用大小接着可能又翻了一倍。这 么做下来，在 GC 性能不受损失的情况下，我们可在一台拥有 32G 内存的机器上获得高达 28 到 30G 的缓存。而且，这种缓存即使在服务重启之后会仍然保持有 效，而不象进程内缓存，进程重启后还需要在内存中进行缓存重建(10G 的缓存重建时间可能需要 10 分钟），否则就需要以一个全空的缓存开始运行（这么做它 的初始性能会非常糟糕）。这还大大简化了代码，因为对缓存和文件系统之间的一致性进行维护的所有逻辑现在都是在 OS 中实现

的，这事 OS 做起来要比我们在进 程中做那种一次性的缓存更加高效，准确性也更高。如果你使用磁盘的方式更倾向于线性读取操作，那么随着每次磁盘读取操作，预读就能非常高效使用随后准能用得着的数据填充缓存。

这就让人联想到一个非常简单的设计方案 :不是要在内存中保存尽可能多的数据并在需要时将这些数据刷新（flush）到文件系统，而是我们要做完全相反的事 情。所有数据都要立即写入文件系统中持久化的日志中但不进行刷新数据的任何调用。实际中这么做意味着，数据被传输到 OS 内核的页面缓存中了，OS 随后会将 这些数据刷新到磁盘的。此外我们添加了一条基于配置的刷新策略，允许用户对把数据刷新到物理磁盘的频率进行控制（每当接收到 N 条消息或者每过 M 秒），从而 可以为系统硬件崩溃时“处于危险之中”的数据在量上加个上限。

这种以页面缓存为中心的设计风格在一篇讲解 Varnish 的设计思想的文章中有详细的描述（文风略带有助于身心健康的傲气）。

常量时长足矣

消息系统元数据的持久化数据结构往往采用 BTree。 BTree 是目前最通用的数据结构，在消息系统中它可以用来广泛支持多种不同的事务性或非事务性语义。 它的确也带来了一个非常高的处理开销，Btree 运算的时间复杂度为 O(log N)。一般 O(log N)被认为基本上等于常量时长，但对于磁盘操作来讲，情况就不同了。磁盘寻道时间一次要花 10ms 的时间，而且每个磁盘同时只能进行一个寻道操作，因而其 并行程度很有限。因此，即使少量的磁盘寻道操作也会造成非常大的时间开销。因为存储系统混合了高速缓存操作和真正的物理磁盘操作，所以树型结构（tree structure）可观察到的性能往往是超线性的（superlinear）。

更进一步讲，BTrees 需要一种非常复杂的页面级或行级锁定机制才能避 免在每次操作时锁定一整颗树。实现这种机制就要为行级锁定付出非常高昂的代价，否则就必须对所有的读取操作进行串行化（serialize）。因为对磁盘 寻道操作的高度依赖，就不太可能高效地从驱动器密度（drive density）的提高中获得改善，因而就不得不使用容量较小(< 100GB)转速较高的 SAS 驱动去，以维持一种比较合理的数据与寻道容量之比。

直 觉上讲，持久化队列可以按照通常的日志解决方案的样子构建，只是简单的文件读取和简单地向文件中添加内容。虽然这种结果必然无法支持 BTree 实现中的丰 富语义，但有个优势之处在于其所有的操作的复杂度都是 O(1)，读取操作并不需要阻止写入操作，而且反之亦然。这样做显然有性能优势，因为性能完全同数据 大小之间脱离了关系 —— 一个服务器现在就能利用大量的廉价、低转速、容量超过 1TB 的 SATA 驱动器。虽然这些驱动器寻道操作的性能很低，但这些驱动器在大量数据读写的情况下性 能还凑和，而只需 1/3 的价格就能获得 3 倍的容量。能够存取到几乎无限大的磁盘空间而无须付出性能代价意味着，我们可以提供一些消息系统中并不常见的功能。例如，在 Kafka 中，消息在使用完后并没有立即 删除，而是会将这些消息保存相当长的一段时间（比方说一周）。

效率最大化

我们的假设是，系统里消息的量非常之大，实际消息量是网站页面浏览总数的数倍之多（因为每个页面浏览就是我们要处理的其中一个活动）。而且我们假设发布的每条消息都会被至少读取一次（往往是多次），因而我们要为消息使用而不是消息的产生进行系统优化，

导致低效率的原因常见的有两个：过多的网络请求和大量的字节拷贝操作。

为了提高效率，API 是围绕这“消息集”（message set）抽象机制进行设计的，消息集将消息进行自然分组。这么做能让网络请求把消息合成一个小组，分摊网络往返（roundtrip）所带来的开销，而不是每次仅仅发送一个单个消息。

MessageSet 实现（implementation）本身是对字节数组或文件进行一次包装后形成的一薄层 API。因而，里面并不存在消息处理所需的 单独的序列化（serialization）或逆序列化（deserialization）的步骤。消息中的字段（field）是按需进行逆序列化的（或 者说，在不需要时就不进行逆序列化）。

由代理维护的消息日志本身不过是那些已写入磁盘的消息集的目录。按此进行抽象处理后，就可以让代理和消息使用者共用一个单个字节的格式（从某种程度上说，消息生产者也可以用它，消息生产者的消息要求其校验和（checksum）并在验证后才会添加到日志中）

使用共通的格式后就能对最重要的操作进行优化了 :持久化后日志块（chuck）的网络传输。为了将数据从页面缓存直接传送给 socket，现代的 Unix 操作系统提供了一个高度优化的代码路径（code path）。在 Linux 中这是通过 sendfile 这个系统调用实现的。通过 Java 中的 API，`FileChannel.transferTo`，由它来简洁的调用上述的系统调用。

为了理解 sendfile 所带来的效果，重要的是要理解将数据从文件传输到 socket 的数据路径：

1. 操作系统将数据从磁盘中读取到内核空间里的页面缓存
2. 应用程序将数据从内核空间读入到用户空间的缓冲区
3. 应用程序将读到的数据写回内核空间并放入 socke 的缓冲区
4. 操作系统将数据从 socket 的缓冲区拷贝到 NIC（网络借口卡，即网卡）的缓冲区，自此数据才能通过网络发送出去

这样效率显然很低，因为里面涉及 4 次拷贝，2 次系统调用。使用 sendfile 就可以避免这些重复的拷贝操作，让 OS 直接将数据从页面缓存发送到网络中，其中只需最后一步中的将数据拷贝到 NIC 的缓冲区。

我们预期的一种常见的用例是一个话题拥有多个消息使用者。采用前文所述的零拷贝优化方案，数据只需拷贝到页面缓存中一次，然后每次发送给使用者时都对它进行重复使用即可，而无须先保存到内存中，然后在阅读该消息时每次都需要将其拷贝到内核空间中。如此一来，消息使用的速度就能接近网络连接的极限。

要得到 Java 中对 send'file 和零拷贝的支持方面的更多背景知识，请参考 IBM developerworks 上的这篇文章。

端到端的批量压缩

多数情况下系统的瓶颈是网络而不是 CPU。 这一点对于需要将消息在个数据中心间进行传输的数据管道来说，尤其如此。当然，无需来自 Kafka 的支持，用户总是可以自行将消息压缩后进行传输，但这么 做的压缩率会非常低，因为不同的消息里都有很多重复性的内容（比如 JSON 里的字段名、web 日志中的用户代理或者常用的字符串）。高效压缩需要将多条消 息一起进行压缩而不是分别压缩每条消息。理想情况下，以端到端的方式这么做是行得通的 —— 也即，数据在消息生产者发送之前先压缩一下，然后在服务器上一直保存压缩状态，只有到最终的消息使用者那里才需要将其解压缩。

通过运行递归消息集，Kafka 对这种压缩方式提供了支持。 一批消息可以打包到一起进行压缩，然后以这种形式发送给服务器。这批消息都会被发送给同一个消息使用者，并会在到达使用者那里之前一直保持为被压缩的形式。

Kafka 支持 GZIP 和 Snappy 压缩协议。关于压缩的更多更详细的信息，请参见这里。

客户状态

追踪（客户）消费了什么是一个消息系统必须提供的一个关键功能之一。它并不直观，但是记录这个状态是该系统的关键性能之一。状态追踪要求（不断）更新一个 有持久性的实体的和一些潜在会发生的随机访问。因此它更可能受到存储系统的查询时间的制约而不是带宽（正如上面所描述的）。

大部分消息系统保留着关于代理者使用(消费)的消息的元数据。也就是说，当消息被交到客户手上时，代理者自己记录了整个过程。这是一个相当直观的选择,而 且确实对于一个单机服务器来说，它(数据)能去(放在)哪里是不清晰的。又由于许多消息系统存储使用的数据结构规模小，所以这也是个实用的选择--因为代 理者知道什么被消费了使得它可以立刻删除它(数据)，保持数据大小不过大。

也许不显然的是 ,让代理和使用者这两者对消息的使用情况做到一致表述绝不是一件轻而易举的事情。如果代理每次都是在将消息发送到网络中后就将该消息记录为已使用的话，一旦使用者没能真正处理到该消息（比方说，因为它宕机或这请求超时了抑或别的什么原因），就会出现消息丢失的情况。为了解决此问题，许多消息系新加了一个确认功能，当消息发出后仅把它标示为已发送而不是已使用 ,然后代理需要等到来自使用者的特定的确认信息后才将消息记录为已使用。 这种策略的确解决了丢失消息的问题 ,但由此产生了新问题。首先，如果使用者已经处理了该消息但却未能发送出确认信息 ,那么就会让这一条消息被处理两次。第 二个问题是关于性能的，这种策略中的代理必须为每条单个的消息维护多个状态（首先为了防止重复发送就要将消息锁定，然后，然后还要将消息标示为已使用后才 能删除该消

息）。另外还有一些棘手的问题需要处理，比如，对于那些以发出却未得到确认的消息该如何处理？

消息传递语义（Message delivery semantics）

系统可以提供的几种可能的消息传递保障如下所示:

- 最多一次—这种用于处理前段文字所述的第一种情况。消息在发出后立即标示为已使用，因此消息不会被发出去两次，但这在许多故障中都会导致消息丢失。
- 至少一次—这种用于处理前文所述的第二种情况，系统保证每条消息至少会发送一次，但在有故障的情况下可能会导致重复发送。
- 仅仅一次—这种是人们实际想要的，每条消息只会而且仅会发送一次。

这个问题已得到广泛的研究，属于“事务提交”问题的一个变种。提供仅仅一次语义的算法已经有了，两阶段或者三阶段提交法以及 Paxos 算法的一些变种就是 其中的一些例子，但它们都有与生俱来的的缺陷。这些算法往往需要多个网络往返（round trip），可能也无法很好的保证其活性（liveness）（它们可能会导致无限期停机）。FLP 结果给出了这些算法的一些基本的局限。

Kafka 对元数据做了两件很不寻常的事情。一件是，代理将数据流划分为一组互相独立的分区。这些分区的语义由生产者定义，由生产者来指定每条消息属于哪个分区。一 个分区内的消息以到达代理的时间为准进行排序，将来按此顺序将消息发送给使用者。这么一来，就用不着为每一天消息保存一条元数据（比如说，将消息标示为已 使用）了，我们只需为使用者、话题和分区的每种组合记录一个“最高水位标记”（high water mark）即可。因

此，标示使用者状态所需的元数据总量实际上特别小。在 Kafka 中，我们将该最高水位标记称为“偏移量”（offset），这么叫的原 因将在实现细节部分讲解。

使用者的状态

在 Kafka 中，由使用者负责维护反映哪些消息已被使用的状态信息（偏移量）。典型情况下，Kafka 使用者的 library 会把状态数据保存到 Zookeeper 之中。然而，让使用者将状态信息保存到保存它们的消息处理结果的那个数据存储(datastore)中也许会更佳。例如，使用者也许就 是要把一些统计值存储到集中式事物 OLTP 数据库中，在这种情况下，使用者可以在进行那个数据库数据更改的同一个事务中将消息使用状态信息存储起来。这样 就消除了分布式的部分，从而解决了分布式中的一致性问题！这在非事务性系统中也有类似的技巧可用。搜索系统可用将使用者状态信息同它的索引段（index segment）存储到一起。尽管这么做可能无法保证数据的持久性（durability），但却可用让索引同使用者状态信息保存同步：如果由于宕机造成 有一些没有刷新到磁盘的索引段信息丢了，我们总是可用从上次建立检查点(checkpoint)的偏移量处继续对索引进行处理。与此类似，Hadoop 的 加载作业（load job）从 Kafka 中并行加载，也有相同的技巧可用。每个 Mapper 在 map 任务结束前，将它使用的最后一个消息的偏移量存入 HDFS。

这个决策还带来一个额外的好处。使用者可用故意回退（rewind）到 以前的偏移量处，再次使用一遍以前使用过的数据。虽然这么做违背了队列的一般协约（contract），但对很多使用者来讲却是个很基本的功能。举个例 子，如果使用者的代码里有个 Bug，而且是在它处理完一些消息之后才被发现的，那么当把 Bug 改正后，使用者还有机会重新处理一遍那些消息。

Push 和 Pull

相关问题还有一个，就是到底是应该让使用者从代理那里吧数据 Pull（拉）回来还是应该让代理把数据 Push（推）给使用者。和大部分消息系统一 样，Kafka 在这方面遵循了一种更加传统的设计思路：由生产者将数据 Push 给代理，然后由使用者将数据代理那里 Pull 回来。近来有些系统，比如 scribe 和 flume，更着重于日志统计功能，遵循了一种非常不同的基于 Push 的设计思路，其中每个节点都可以作为代理，数据一直都是向下游 Push 的。上述两种方法都各有优缺点。然而，因为基于 Push 的系统中代理控制着数据的传输速率，因此它难以应付大量不同种类的使用者。我们的设计目标是，让使用者能以它最大的速率使用数据。不幸的是，在 Push 系统中当数据的使用速率低于产生的速率时，使用者往往会处于超载状态（这实际上就是一种拒绝服务攻击）。基于 Pull 的系统在使用者的处理速度稍 稍落后的情况下会表现更佳，而且还可以让使用者在有能力的时候往往前赶赶。让使用者采用某种退避协议（backoff protocol）向代理表明自己处于超载状态，可以解决部分问题，但是，将传输速率调整到正好可以完全利用（但从不能过度利用）使用者的处理能力可比初 看上去难多了。以前我们尝试过多次，想按这种方式构建系统，得到的经验教训使得我们选择了更加常规的 Pull 模型。

分发

Kafka 通常情况下是运行在集群中的服务器上。没有中央的“主”节点。代理彼此之间是对等的，不需要任何手动配置即可可随时添加和删除。同样，生产者和 消费者可以在任何时候开启。 每个代理都可以在 Zookeeper(分布式协调系统)中注册的一些元数据（例如，可

用的主题）。生产者和消费者可以使用 Zookeeper 发现主题和相 互协调。关于生产者和消费者的细节将在下面描述。

生产者

生产者自动负载均衡

对于生产者，Kafka 支持客户端负载均衡，也可以使用一个专用的负载均衡器对 TCP 连接进行负载均衡调整。专用的第四层负载均衡器在 Kafka 代理之上 对 TCP 连接进行负载均衡。在这种配置的情况，一个给定的生产者所发送的消息都会发送给一个单个的代理。使用第四层负载均衡器的好处是，每个生产者仅需一 个单个的 TCP 连接而无须同 Zookeeper 建立任何连接。不好的地方在于所有均衡工作都是在 TCP 连接的层次完成的，因而均衡效果可能并不佳（如果有 些生产者产生的消息远多于其它生产者，按每个代理对 TCP 连接进行平均分配可能会导致每个代理接收到的消息总数并不平均）。

采用客户端基于 zookeeper 的负载均衡可以解决部分问题。如果这么做就能让生产者动态地发现新的代理，并按请求数量进行负载均衡。类似的，它还能让 生产者按照某些键值（key）对数据进行分区（partition）而不是随机乱分，因而可以保存同使用者的关联关系（例如，按照用户 id 对数据使用进行 分区）。这种分法叫做“语义分区”（semantic partitioning），下文再讨论其细节。

下面讲解基于 zookeeper 的负载均衡的工作原理。在发生下列事件时要对 zookeeper 的监视器（watcher）进行注册：

- 加入了新的代理
- 有一个代理下线了

- 注册了新的话题
- 代理注册了已有话题。

生产者在其内部为每一个代理维护了一个弹性的连接（同代理 建立的连接）池。通过使用 zookeeper 监视器的回调函数（callback），该连接池在建立/保持同所有在线代理的连接时都要进行更新。当生产者 要求进入某特定话题时，由分区者（partitioner）选择一个代理分区（参加语义分区小结）。从连接池中找出可用的生产者连接，并通过它将数据发送 到刚才所选的代理分区。

异步发送

对于可伸缩的消息系统而言，异步非阻塞式操作是不可或缺的。在 Kafka 中，生产者有个选项（producer.type=async）可用指定使用异步 分发出产请求（produce request）。这样就允许用一个内存队列（in-memory queue）把生产请求放入缓冲区，然后再以某个时间间隔或者事先配置好的批量大小将数据批量发送出去。因为一般来说数据会从一组以不同的数据速度生产数 据的异构的机器中发布出，所以对于代理而言，这种异步缓冲的方式有助于产生均匀一致的流量，因而会有更佳的网络利用率和更高的吞吐量。

语义分区

下面看看一个想要为每个成员统计一个个人空间访客总数的程序该怎么做。应该把一个成员的所有个人空间访问事件发送给某特定分区，因此就可以把对一个成员的 所有更新都放在同一个使用者线程中的同一个事件流中。生产者具有从语义上将消息映射到有效的 Kafka 节点和分区之上的能力。这样就可以用一个语义分区函 数将消息流按照消息中的某个键值进行分区，并将不同分区发送给各自相应的代理。通过实现 kafak.producer.Partitioner

接口，可以 对分区函数进行定制。在缺省情况下使用的是随即分区函数。上例中，那个键值应该是 member_id，分区函数可以是 hash(member_id)%num_partitions。

对 Hadoop 以及其它批量数据装载的支持

具有伸缩性的持久化方案使得 Kafka 可支持批量数据装载，能够周期性将快照数据载入进行批量处理的离线系统。我们利用这个功能将数据载入我们的数据仓库（data warehouse）和 Hadoop 集群。

批量处理始于数据载入阶段，然后进入非循环图（acyclic graph）处理过程以及输出阶段（支持情况在这里）。支持这种处理模型的一个重要特性是，要有重新装载从某个时间点开始的数据的能力（以防处理中有任何错误发生）。

对于 Hadoop，我们通过在单个的 map 任务之上分割装载任务对数据的装载进行了并行化处理，分割时，所有节点/话题/分区的每种组合都要分出一个来。Hadoop 提供了任务管理，失败的任务可以重头再来，不存在数据被重复的危险。

实施细则

下面给出了一些在上一节所描述的低层相关的实现系统的某些部分的细节的简要说明。

API 设计

生产者 APIs

生产者 API 是给两个底层生产者的再封装

-kafka.producer.SyncProducerandkafka.producer.async.AsyncProducer.

[java] view plaincopy

```
1. class Producer {
2.
3.     /* Sends the data, partitioned by key to the topic
     using either the */
4.     /* synchronous or the asynchronous producer */
5.     public void send(kafka.javaapi.producer.ProducerData pro
   ducerData);
6.
7.     /* Sends a list of data, partitioned by key to th
   e topic using either */
8.     /* the synchronous or the asynchronous producer */

9.     public void send(java.util.List< kafka.javaapi.producer.
   ProducerData> producerData);
10.
11.    /* Closes the producer and cleans up */
12.    public void close();
13.
14.}
```

该 API 的目的是将生产者的所有功能通过一个单个的 API 公开给其使用者（client）。新建的生产者可以：

- 对多个生产者请求进行排队/缓冲并异步发送批量数据 —— kafka.producer.Producer 提供了在将多个生产请求序列化并发送给适当的 Kafka 代理分区之前，对这些生产请求进行批量处理的能力 （producer.type=async）。批量的大小可以通过一些配置参数进行控制。当事件进入队列时会先放入队列进行缓冲，直到时间到了 queue.time 或者批量大小到达 batch.size 为止，后台线程（kafka.producer.async.ProducerSendThread）会将这批数据从队列中取出，交给 kafka.producer.EventHandler 进行序列化并发送给适当的 kafka 代理分区。通过 event.handler 这个配置参数，可 以在系统中插入一个自定义的事件处理器。在该生产者队列管道中的各个不同阶段，为了插入自定义的日志/跟踪代码或者自定义的监视逻

辑，如能注入回调函数会 非常有用。通过实现 kafka.producer.asyn.CallbackHandler 接口并将配置参数 callback.handler 设置为实 现类就能够实现注入。

- 使用用户指定的 Encoder 处理数据的序列化（serialization）

```
1 interface Encoder<T> {
2   public Message toMessage(T data);
3 }
```

- Encoder 的缺省值是一个什么活都不干的 kafka.serializer.DefaultEncoder。
- 提 供基于 zookeeper 的代理自动发现功能 —— 通过使用 zk.connect 配置参数指定 zookeeper 的连接 url，就能够使用基于 zookeeper 的代理发现和负载均衡功能。在有些应用场 合，可能不太适合于依赖 zookeeper。在这种情况下，生产者可以从 broker.list 这个配置参数中获得一个代理的静态列表，每个生产请求会被 随即的分配给各代理分区。如果相应的代理宕机，那么生产请求就会失败。
- 通过使用一个可选性的、由用户指定的 Partitioner 提供由软件实现的负载均衡功能 —— 数据发送路径选择决策受 kafka.producer.Partitioner 的影响。

```
1 interface Partitioner<T> {
2    int partition(T key, int numPartitions);
3 }
```

- 分区 API 根据相关的键值以及系统中具有的代理分区的数量返回一个分区 id。将该 id 用作索引，在 broker_id 和 partition 组成的经过排序 的列表中为相应的生产者请求找出一个代理分区。缺省的分区策略是 hash(key)%numPartitions。如果 key 为 null，那就进行随机选 择。使用 partitioner.class 这个配置参数也可以插入自定义的分区策略。

使用者 API

我们有两个层次的使用者 API。底层比较简单的 API 维护了一个同单个代理建立的连接，完全同发送给服务器的网络请求相吻合。该 API 完全是无状态的，每个请求都带有一个偏移量作为参数，从而允许用户以自己选择的任意方式维护该元数据。

高层 API 对使用者隐藏了代理的具体细节，让使用者可运行于集群中的机器之上而无需关心底层的拓扑结构。它还维护着数据使用的状态。高层 API 还提供了订阅同一个过滤表达式（例如，白名单或黑名单的正则表达式）相匹配的多个话题的能力。

底层 API

```
class SimpleConsumer {


  /* Send fetch request to a broker and get back a set of messages. */
  public ByteBufferMessageSet fetch(FetchRequest request);


  /* Send a list of fetch requests to a broker and get back a response set. */
  public MultiFetchResponse multifetch(List<FetchRequest> fetches);


  /**
   * Get a list of valid offsets (up to maxSize) before the given time.
   * The result is a list of offsets, in descending order.
   * @param time: time in millisecs,
   *            if set to OffsetRequest$.MODULE$.LATIEST_TIME(), get from the
latest offset available.
   *            if set to OffsetRequest$.MODULE$.EARLIEST_TIME(), get from the
earliest offset available.
   */
  public long[] getOffsetsBefore(String topic, int partition, long time, int
maxNumOffsets);
}
```

底层 API 不但用于实现高层 API，而且还直接用于我们的离线使用者（比如 Hadoop 这个使用者），这些使用者还对状态的维护有比较特定的需求。

高层 API

```
/* create a connection to the cluster */
ConsumerConnector connector = Consumer.create(consumerConfig);


interface ConsumerConnector {


  /**
   * This method is used to get a list of KafkaStreams, which are iterators over
   * MessageAndMetadata objects from which you can obtain messages and their
   * associated metadata (currently only topic).
   *  Input: a map of <topic, #streams>
   *  Output: a map of <topic, list of message streams>
   */
  public Map<String,List<KafkaStream>> createMessageStreams(Map<String,Int>
topicCountMap);


  /**
   * You can also obtain a list of KafkaStreams, that iterate over messages
   * from topics that match a TopicFilter. (A TopicFilter encapsulates a
   * whitelist or a blacklist which is a standard Java regex.)
   */
  public List<KafkaStream> createMessageStreamsByFilter(
      TopicFilter topicFilter, int numStreams);


  /* Commit the offsets of all messages consumed so far. */
  public commitOffsets()
```

```
  /* Shut down the connector */
  public shutdown()
}
```

该 API 的中心是一个由 KafkaStream 这个类实现的迭代器(iterator)。每个 KafkaStream 都代表着一个从一个或多个分区到一个 或多个服务器的消息流。每个流都是使用单个线程进行处理的 ,所以 ,该 API 的使用者在该 API 的创建调用中可以提供所需的任意个数的流。这样，一个流可能 会代表多个服务器分区的合并（同处理线程的数目相同），但每个分区只会把数据发送给一个流中。

createMessageStreams 方法为使用者注册到相应的话题之上，这将导致需要对使用者/代理的分配情况进行重新平衡。为了将重新平衡操作减 少到最小。该 API 鼓励在一次调用中就创建多个话题流。createMessageStreamsByFilter 方法为发现同其过滤条件想匹配的话题 （额外地)注册了多个监视器(watchers)。应该注意 ,createMessageStreamsByFilter 方法所返回的每个流都可能会对多 个话题进行迭代（比如，在满足过滤条件的话题有多个的情况下）。

网络层

网络层就是一个特别直截了当的 NIO 服务器，在此就不进行过于细致的讨论了。sendfile 是通过给 MessageSet 接口添加了一个 writeTo 方法实现的。这样就可以让基于文件的消息更加高效地利用 transferTo 实现，而不是使用线程内缓冲区读写方式。线程模型用的是一个单个的接收器（acceptor 线程和每个可以处理固定数量网络连接的 N 个处理器线程。这种设计方案在别处已经经过了非常彻底的检验，发现其实现起来简单、运行起来很快。其中使用的协议一直都非常简单，将来还可以用其它语言实现其客户端。

消息

消息由一个固定大小的消息头和一个变长不透明字节数字的有效载荷构成（opaque byte array payload）。消息头包含格式的版本信息和一个用于探测出坏数据和不完整数据的 CRC32 校验。让有效载荷保持不透明是个非常正确的决策：在用于序列 化的代码库方面现在正在取得非常大的进展，任何特定的选择都不可能适用于所有的使用情况。都不用说，在 Kafka 的某特定应用中很有可能在它的使用中需要 采用某种特殊的序列化类型。MessageSet 接口就是一个使用特殊的方法对 NIOChannel 进行大宗数据读写（bulk reading and writing to an NIOChannel）的消息迭代器。

消息的格式

```
        /**
         * A message. The format of an N byte message is the following:
         *
         * If magic byte is 0
         *
         * 1. 1 byte "magic" identifier to allow format changes
         *
         * 2. 4 byte CRC32 of the payload
         *
         * 3. N - 5 byte payload
         *
         * If magic byte is 1
         *
         * 1. 1 byte "magic" identifier to allow format changes
         *
```

```
         * 2. 1 byte "attributes" identifier to allow annotations on the message
independent of the version (e.g. compression enabled, type of codec used)
         *
         * 3. 4 byte CRC32 of the payload
         *
         * 4. N - 6 byte payload
         *
         */
```

日志

具有两个分区的、名称为"my_topic"的话题的日志由两个目录组成（即：my_topic_0 和 my_topic_1)，目录中存储的是内容为该话 题的消息的数据文件。日志的文件格式是一系列的“日志项”；每条日志项包含一个表示消息长度的 4 字节整数 N，其后接着保存的是 N 字节的消息。每条消息用一 个 64 位的整数偏移量进行唯一性标示，该偏移量表示了该消息在那个分区中的那个话题下发送的所有消息组成的消息流中所处的字节位置。每条消息在磁盘上的格 式如下文所示。每个日志文件的以它所包含的第一条消息的偏移量来命名。因此，第一个创建出来的文件的名字将为 00000000000.kafka，随后每 个后加的文件的名字将是前一个文件的文件名大约再加 S 个字节所得的整数，其中，S 是配置文件中指定的最大日志文件的大小。

消息的确切的二进制格式都有版本，它保持为一个标准的接口，让消息集可以根据需要在生产者、代理、和使用者直接进行自由传输而无须重新拷贝或转换。其格式如下所示：

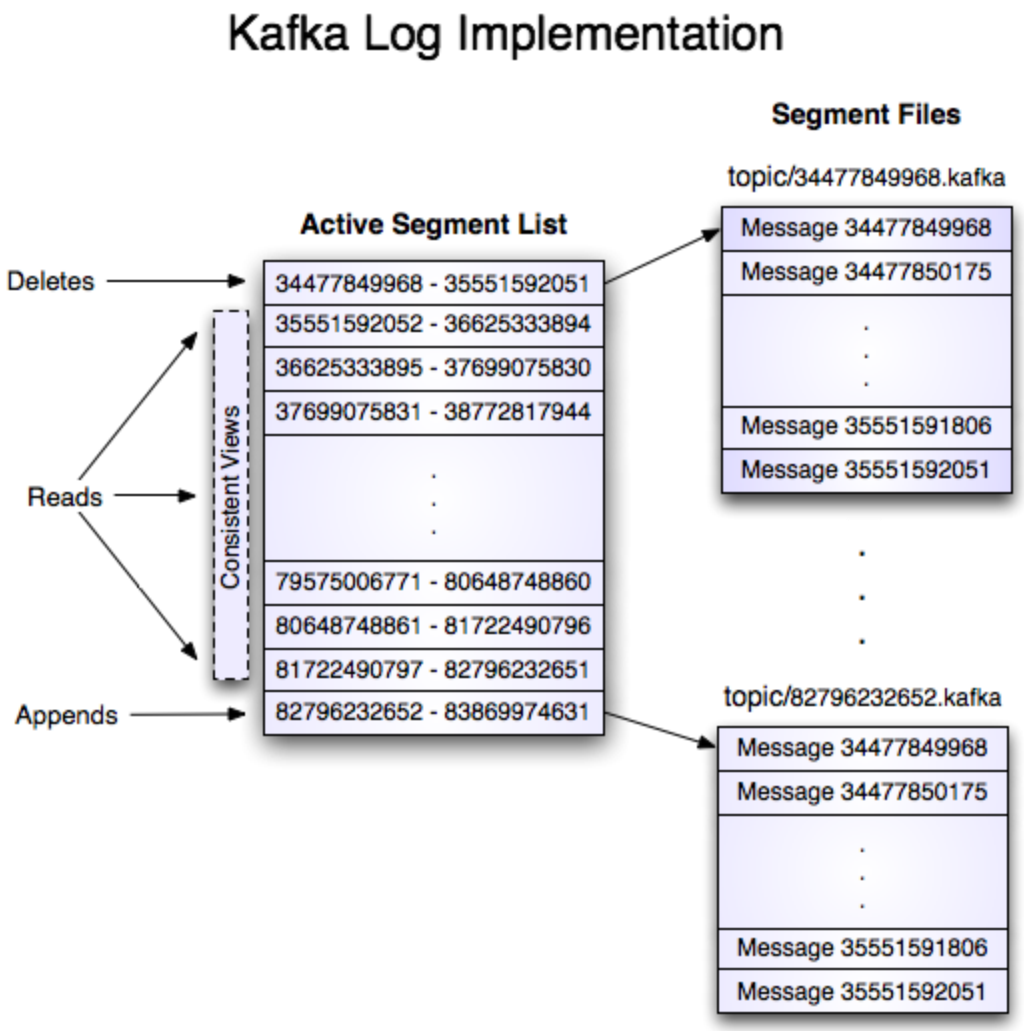
```
On-disk format of a message


message length : 4 bytes (value: 1+4+n)
"magic" value  : 1 byte
```

```
crc          : 4 bytes

payload      : n bytes
```

将消息的偏移量作为消息的可不常见。我们原先的想法是使用由生产者产生的 GUID 作为消息 id，然后在每个代理上作一个从 GUID 到偏移量的映射。但是， 既然使用者必须为每个服务器维护一个 ID，那么 GUID 所具有的全局唯一性就失去了价值。更有甚者，维护将从一个随机数到偏移量的映射关系带来的复杂性，使得我们必须使用一种重量级的索引结构，而且这种结构还必须与磁盘保持同步，这样我们还就必须使用一种完全持久化的、需随机访问的数据结构。如此一来，为 了简化查询结构，我们就决定使用一个简单的依分区的原子计数器（atomic counter），这个计数器可以同分区 id 以及节点 id 结合起来唯一的指定一条消息；这种方法使得查询结构简化不少，尽管每次在处理使用者请求时仍有可 能会涉及多次磁盘寻道操作。然而，一旦我们决定使用计数器，跳向直接使用偏移量作为 id 就非常自然了，毕竟两者都是分区内具有唯一性的、单调增加的整数。 既然偏移量是在使用者 API 中并不会体现出来，所以这个决策最终还是属于一个实现细节，进而我们就选择了这种更加高效的方式。



写操作

日志可以顺序添加 ,添加的内容总是保存到最后一个文件。当大小超过配置中指定的大小(比如说 1G)后 , 该文件就会换成另外一个新文件。有关日志的配置参数 有两个 , 一个是 M , 用于指出写入多少条消息之后就要强制 OS 将文件刷新到磁盘 ; 另一个是 S , 用来指定过多少秒就要强制进行一次刷新。这样就可以保证一旦发 生系统崩溃 ,最多会有 M 条消息丢失 , 或者最长会有 S 秒的数据丢失 ,

读操作

可以通过给出消息的 64 位逻辑偏移量和 S 字节的数据块最大的字节数对日志文件进行读取。读取操作返回的是这 S 个字节中包含的消息的迭代器。S 应该要比最长 的单条消息的字节数大，但在出现特别长的消息情况下，可以重复进行多次读取，每次的缓冲区大小都加倍，直到能成功读取出这样长的一条消息。也可以指定一个 最大的消息和缓冲区大小并让服务器拒绝接收比这个大小大一些的消息 ,这样也能给客户端一个能够读取一条完整消息所需缓冲区的大小的上限。很有可能会出现读 取缓冲区以一个不完整的消息结尾的情况，这个情况用大小界定（size delimiting）很容易就能探知。

从某偏移量开始进行日志读取的实际过程需要先找出存储所需数据的日志段文件 ,从全局偏移量计算出文件内偏移量 ,然后再从该文件偏移量处开始读取。搜索过程通过对每个文件保存在内存中的范围值进行一种变化后的二分查找完成。

日志提供了获取最新写入的消息的功能，从而允许从“当下”开始消息订阅。这个功能在使用者在 SLA 规定的天数内没能正常使用数据的情况下也很有用。当使用 者企图从一个并不存在的偏移量开始使用数据时就会出现这种情况，此时使用者会得到一个 OutOfRangeException 异常，它可以根据具体的使用 情况对自己进行重启或者仅仅失败而退出。

以下是发送给数据使用者（consumer）的结果的格式。

```
MessageSetSend (fetch result)



total length    : 4 bytes

error code      : 2 bytes
```

```
message 1       : x bytes

...

message n       : x bytes

MultiMessageSetSend (multiFetch result)


total length      : 4 bytes

error code        : 2 bytes

messageSetSend 1

...

messageSetSend n
```

删除

一次只能删除一个日志段的数据。日志管理器允许通过可加载的删除策略设定删除的文件。当前策略删除修改事件超过 N 天以上的文件，也可以选择保留最后 N GB 的数据。 为了避免删除时的读取锁定冲突，我们可以使用副本写入模式，以便在进行删除的同时对日志段的一个不变的静态快照进行二进制搜索。

数据正确性保证

日志功能里有一个配置参数 M，可对在强制进行磁盘刷新之前可写入的消息的最大条目数进行控制。在系统启动时会运行一个日志恢复过程，对最新的日志段内所有消息进行迭代，以对每条消息项的有效性进行验证。一条消息项是合法的，仅当其大小加偏移量小于文件的大小并且该消息中有效载荷的 CRC32 值同该消息中存储的 CRC 值相等。在探测出有数据损坏的情况下，就要将文件按照最后一个有效的偏移量进行截断。

要注意，这里有两种必需处理的数据损坏情况：由于系统崩溃造成的未被正常写入的数据块（block）因而需要截断的情况以及由于文件中被加入了毫无意义的 数据块而造成的数据

损坏情况。造成数据损坏的原因是，一般来说 OS 并不能保证文件索引节点（inode）和实际数据块这两者的写入顺序，因此，除了可能会 丢失未刷新的已写入数据之外，在索引节点已经用新的文件大小更新了但在将数据块写入磁盘块之前发生了系统崩溃的情况下 ,文件就可能会获得一些毫无意义的数 据。CRC 值就是用于这种极端情况，避免由此造成整个日志文件的损坏（尽管未得到保存的消息当然是真的找不回来了）。

分发

Zookeeper 目录

接下来讨论 zookeeper 用于在使用者和代理直接进行协调的结构和算法。

记法

当一个路径中的元素是用[xyz]这种形式表示的时，其意思是, xyz 的值并不固定而且实际上 xyz 的每种可能的值都有一个 zookpeer z 节点（znode）。例如，/topics/[topic]表示了一个名为/topics 的目录，其中包含的子目录同话题对应，一个话题一个目录并且目 录名即为话题的名称。也可以给出数字范围，例如[0...5]，表示的是子目录 0、1、2、3、4。箭头->用于给出 z 节点的内容。例如 /hello -> world 表示的是一个名称为/hello 的 z 节点，包含的值为"world"。

代理节点的注册

```
/brokers/ids/[0...N] --> host:port (ephemeral node)
```

上面是所有出现的代理节点的列表 列表中每一项都提供了一个具有唯一性的逻辑代理 id，用于让使用者能够识别代理的身份（这个必须在配置中给出）。在启动 时，代理节点就要

用/brokers/ids 下列出的逻辑代理 id 创建一个 z 节点，并在自己注册到系统中。使用逻辑代理 id 的目的是，可以让我们在不影响 数据使用者的情况下就能把一个代理搬到另一台不同的物理机器上。试图用已在使用中的代理 id（比如说，两个服务器配置成了同一个代理 id）进行注册会导致 发生错误。

因为代理是以非长久性 z 节点的方式注册的，所以这个注册过程是动态的，当代理关闭或宕机后注册信息就会消失（至此要数据使用者，该代理不再有效）。

代理话题的注册

```
/brokers/topics/[topic]/[0...N] --> nPartions (ephemeral node)
```

每个代理会都要注册在某话题之下，注册后它会维护并保存该话题的分区总数。

使用者和使用者小组

为了对数据的使用进行负载均衡并记录使用者使用的每个代理上的每个分区上的偏移量，所有话题的使用者都要在 Zookeeper 中进行注册。

多个使用者可以组成一个小组共同使用一个单个的话题。同一小组内的每个使用者共享同一个给定的 group_id。比如说，如果某个使用者负责用三台机器进行某某处理过程，你就可以为这组使用者分配一个叫做 “某某” 的 id。这个小组 id 是在使用者的配置文件中指定的，并且这就是你告诉使用者它到底属于哪个组的方法。

小组内的使用者要尽量公正地划分出分区，每个分区仅为小组内的一个使用者所使用。

使用者 ID 的注册

除了小组内的所有使用者都要共享一个 group_id 之外，每个使用者为了要同其它使用者区别开来，还要有一个非永久性的、具有唯一性的 consumer_id(采用 hostname:uuid 的形式）。 consumer_id 要在以下的目录中进行注册。

```
/consumers/[group_id]/ids/[consumer_id] --> {"topic1": #streams, ..., "topicN": #streams} (ephemeral node)
```

小组内的每个使用者都要在它所属的小组中进行注册并采用 consumer_id 创建一个 z 节点。z 节点的值包含了一个<topic, #streams>的 map。 consumer_id 只是用来识别小组内活跃的每个使用者。使用者建立的 z 节点是个临时性的节点，因此如果这个使用者进程终止了，注册信息也将随之消失。

数据使用者偏移追踪

数据使用者跟踪他们在每个分区中耗用的最大偏移量。这个值被存储在一个 Zookeeper(分布式协调系统)目录中。

```
/consumers/[group_id]/offsets/[topic]/[broker_id-partition_id] --> offset_counter_value ((persistent node)
```

分区拥有者注册表

每个代理分区都被分配给了指定使用者小组中的单个数据使用者。数据使用者必须在耗用给定分区前确立对其的所有权。要确立其所有权，数据使用者需要将其 id 写入到特定代理分区中的一个临时节点(ephemeral node)中。

```
/consumers/[group_id]/owners/[topic]/[broker_id-partition_id] --> consumer_node_id (ephemeral node)
```

代理节点的注册

代理节点之间基本上都是相互独立的，因此它们只需要发布它们拥有的信息。当有新的代理加入进来时，它会将自己注册到代理节点注册目录中，写下它的主机名和 端口。代理还要将已有话题的列表和它们的逻辑分区注册到代理话题注册表中。在代理上生成新话题时，需要动态的对话题进行注册。

使用者注册算法

当使用者启动时，它要做以下这些事情：

1. 将自己注册到它属小组下的使用者 id 注册表。
2. 注册一个监视使用者 id 列的表变化情况（有新的使用者加入或者任何现有使用者的离开）的变化监视器。(每个变化都会触发一次对发生变化的使用者所属的小组内的所有使用者进行负载均衡。)
3. 主次一个监视代理 id 注册表的变化情况(有新的代理加入或者任何现有的代理的离开）的变化监视器。（每个变化都会触发一次对所有小组内的所有使用者负载均衡。）
4. 如果使用者使用某话题过滤器创建了一个消息流，它还要注册一个监视代理话题变化情况（添加了新话题）的变化监视器。（每个变化都会触发一次对所有可用话题的评估，以找出话题过滤器过滤出哪些话题。新过滤出来的话题将触发一次对该使用者所在的小组内所有的使用者负载均衡。）
5. 迫使自己在小组内进行重新负载均衡。

使用者重新负载均衡的算法

使用者重新复杂均衡的算法可用让小组内的所有使用者对哪个使用者使用哪些分区达成一致意见。使用者重新负载均衡的动作每次添加或移除代理以及同一小组内的 使用者时被触

发。对于一个给定的话题和一个给定的使用者小组，代理分区是在小组内的所有使用者中进行平均划分的。一个分区总是由一个单个的使用者使用。这 种设计方案简化了实施过程。假设我们运行多个使用者以并发的方式同时使用同一个分区 ,那么在该分区上就会形成争用（ contention ）的情况，这样一来就需要某种形式的锁定机制。如果使用者的个数比分区多，就会出现有写使用者根本得不到数据的情况。在重新进行负载均衡的过程中，我们按照尽量减少每个使用者需要连接的代理的个数的方式，尝尝试着将分区分配给使用者。

每个使用者在重新进行负载均衡时需要做下列的事情：

1. 针对 C_i 所订阅的每个话题 T
2. 将 P_T 设为生产话题 T 的所有分区
3. 将 C_G 设为小组内同 C_i 一样使用话题 T 的所有使用者
4. 对 P_T 进行排序（让同一个代理上的各分区挨在一起）
5. 对 C_G 进行排序
6. 将 i 设为 C_i 在 C_G 中的索引值并让 N = size(P_T)/size(C_G)
7. 将从 i*N 到(i+1)*N - 1 的分区分配给使用者 C_i
8. 将 C_i 当前所拥有的分区从分区拥有者注册表中删除
9. 将新分配的分区加入到分区拥有者注册表中
   （我们可能需要多次尝试才能让原先的分区拥有者释放其拥有权）

在触发了一个使用者要重新进行负载均衡时 ,同一小组内的其它使用者也会几乎在同时被触发重新进行负载均衡。