细细品味Kafka

——Kafka 简介及安装

精

华

集

锦

csAxp

http://www.xiapistudio.com/

2015 年 9 月 15 日

目录

1、Kafka 简介

1.1 序言

我们为何要使用 Kafka，或者说 Kafka 存在的意思。

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 的**设计目标**：

- 数据在磁盘上存取代价为 O(1)。一般数据在磁盘上是使用 **BTree** 存储的，存取代

价为 O（logn）。

- 高吞吐率。即使在普通的节点上每秒钟也能处理成百上千的 Message。
- 显式分布式，即所有的 Producer、Broker 和 Consumer 都会有多个，均为分布式的。
- 支持数据并行加载到 Hadoop 中。

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：

- 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。
- 主要的设计约束是吞吐量而不是功能。
- 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。
- 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。

1.2 入门介绍

Kafka 是一个分布式（distributed）、分区（partitioned）、复制（replicated）的提交日志服务。它提供了类似于 JMS 的特性，但是在设计实现上完全不同，此外它并不是 JMS 规范的实现。Kafka 对消息保存时根据 Topic（主题）进行归类，发送消息者成为 Producer（生产者），消息接受者成为 Consumer（消费者），此外 Kafka 集群有多个 Kafka 实例组成，每个实例（server）成为 Broker（中间/代理人）。无论是 Kafka 集群，还是 Producer 和 Consumer 都依赖于 Zookeeper 来保证系统可用性集群保存一些 Meta 信息。

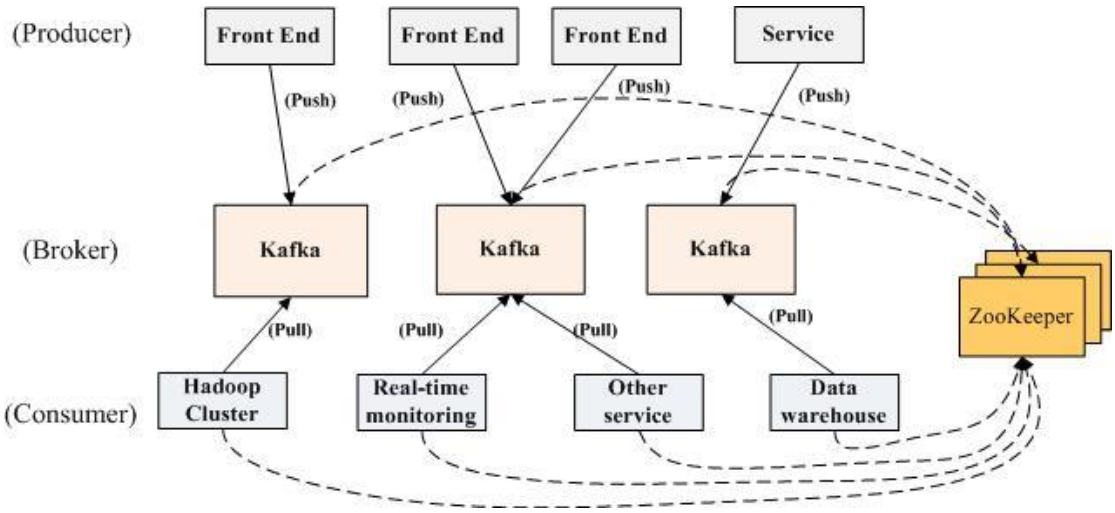


图 1.1 Kafka 部署结构

Kafka 是显式分布式架构，Producer、Broker（Kafka）和 Consumer 都可以有多个。Kafka 的作用类似于缓存，即活跃的数据和离线处理系统之间的缓存。

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。

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。

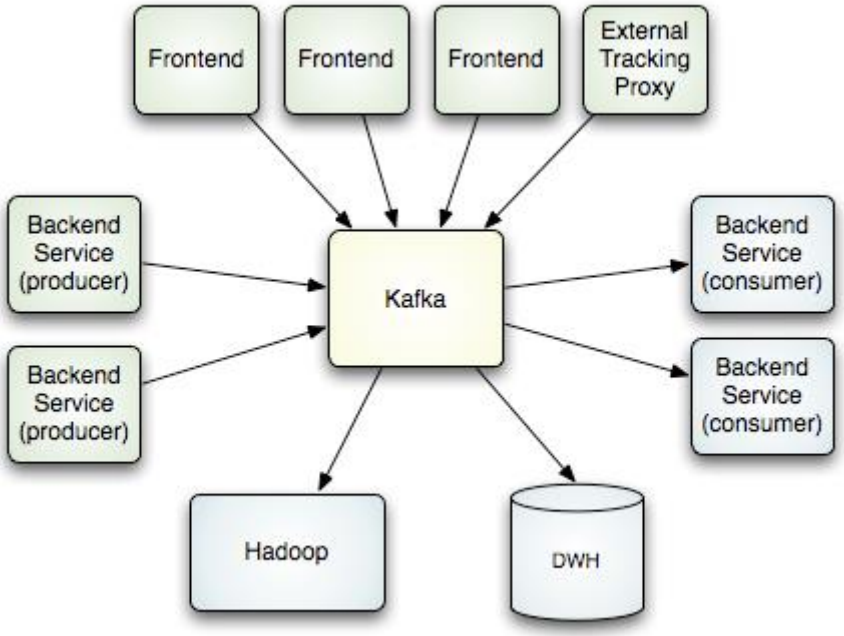


图 1.2 LinkedIn 的拓扑结构

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：

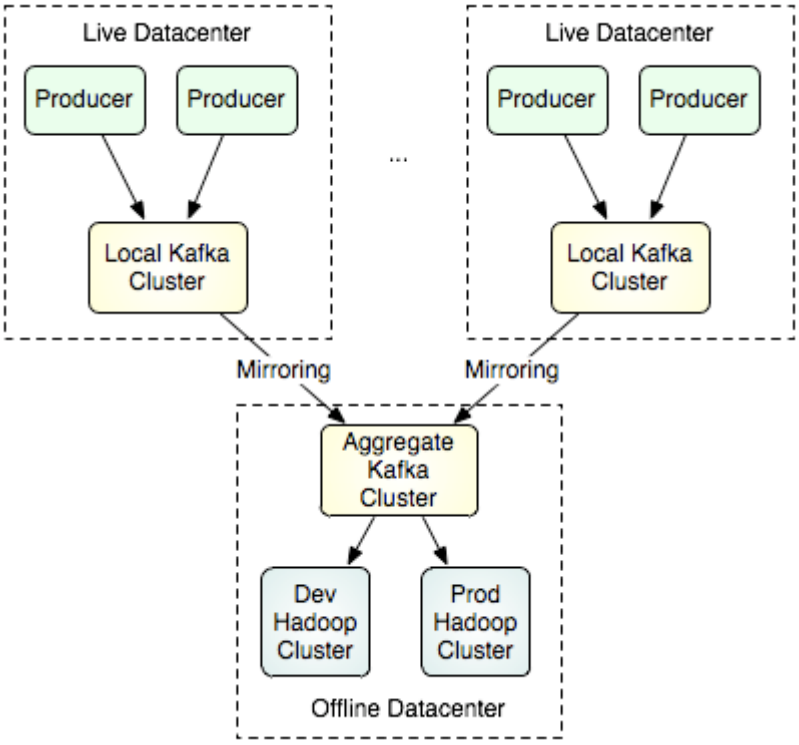


图 1.3 多数据中心拓扑结构

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。

1.3 基本概念

消息（message）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。

【主题和日志（Topics and Logs）】

一个 Topic 可以认为是一类消息，每个 topic 将被分成多个 partition（区），每个 partition 在存储层面是 append log 文件。任何发布到此 partition 的消息都会被直接追加到 log 文件的尾部，每条消息在文件中的位置称为 offset（偏移量），offset 为一个 long 型数字，它是唯一标记一条消息。它唯一的标记一条消息。Kafka 并没有提供其他额外的索引机制来存储 offset，因为在 Kafka 中几乎不允许对消息进行“随机读写”。

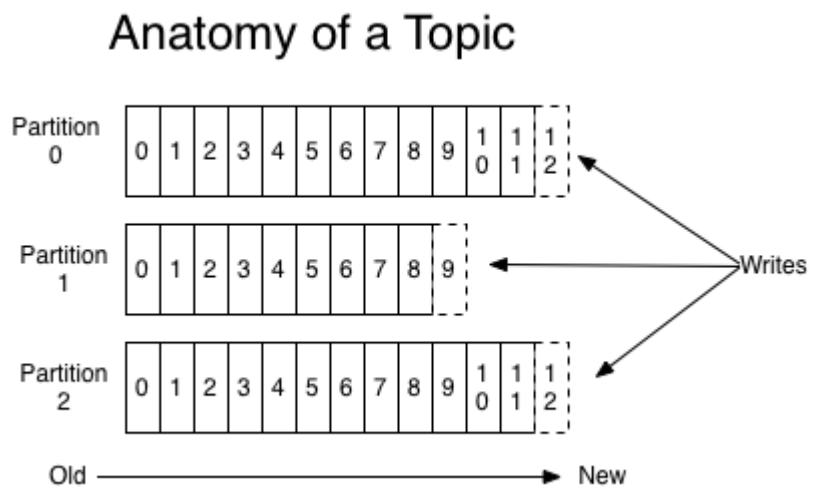


图 1.4 主题结构

Kafka 和 JMS 实现（activeMQ）不同的是：即使消息被消费，消息仍然不会被立即删除。日志文件将会根据 broker 中的配置要求，保留一定的时间之后才会删除；比如 log 文件保留 2 天，那么两天后，文件会被清除，无论其中的消息是否被消费。Kafka 通过这种简单的手段，来释放磁盘空间，以及减少消息消费之后对文件内容改动的磁盘 IO 开支。

对于 consumer 而言，它需要保存消费消息的 offset，对于 offset 的保存和使用，有 consumer 来控制；当 consumer 正常消费消息时，offset 将会"线性"的向前驱动，即消息将依次顺序被消费。事实上 consumer 可以使用任意顺序消费消息，它只需要将 offset 重置为任意值。（offset 将会保存在 zookeeper 中）

Kafka 集群几乎不需要维护任何 consumer 和 producer 状态信息，这些信息由 zookeeper 保存；因此 producer 和 consumer 的客户端实现非常轻量级，它们可以随意离开，而不会对集群造成额外的影响。

Partitions 的设计目的有多个。最根本原因是 Kafka 基于文件存储，通过分区，可以将日志内容分散到多个 server 上，来避免文件尺寸达到单机磁盘的上限，每个 partiton 都会被当前 server（Kafka 实例）保存；可以将一个 topic 切分多任意多个 partitions，来消息保存/消费的效率。此外越多的 partitions 意味着可以容纳更多的 consumer，有效提升并发消费的能力。

【分布式（Distribution）】

一个 Topic 的多个 partitions，被分布在 Kafka 集群中的多个 server 上；每个 server（Kafka 实例）负责 partitions 中消息的读写操作；此 Kafka 还可以配置 partitions 需要备份的个数（replicas），每个 partition 将会被备份到多台机器上,以提高可用性。

基于 replicated 方案，那么就意味着需要对多个备份进行调度；每个 partition 都有一个 server 为"**leader**"；leader 负责所有的读写操作，如果 leader 失效，那么将会有其他 follower 来接管（成为新的 leader）；follower 只是单调的和 leader 跟进，同步消息即可。由此可见作为 leader 的 server 承载了全部的请求压力，**因此从集群的整体考虑，有多少个 partitions 就意味着有多少个"leader"，kafka 会将"leader"均衡的分散在每个实例上，来确保整体的性能稳定。**

【生产者（Producers）】

Producer 将消息发布到指定的 Topic 中，同时 Producer 也能决定将此消息归属于哪个 partition；比如基于“round-robin”的简单地平衡负载或者根据一些语义分区函数（基于消息中的某些关键词），更多是将采用第二种分区方式。

【消费者（Consumers）】

本质上 Kafka 只支持 Topic。**每个 consumer 属于一个 consumer group；反过来说，每个 group 中可以有多个 consumer。发送到 Topic 的消息，只会被订阅此 Topic 的每个 group 中的一个 consumer 消费。**如果所有的 consumer 都具有相同的 group，这种情况和 queue 模式很像；消息将会在 consumers 之间负载均衡。如果所有的 consumer 都具有不同的 group，那这就是"发布-订阅"；消息将会广播给所有的消费者。

在 Kafka 中，一个 partition 中的消息只会被 group 中的一个 consumer 消费；每个 group 中 consumer 消息消费互相独立；我们可以认为一个 group 是一个"订阅"者，一个 Topic 中的

每个 partions，只会被一个"订阅者"中的一个 consumer 消费，不过一个 consumer 可以消费多个 partitions 中的消息。Kafka 只能保证一个 partition 中的消息被某个 consumer 消费时，消息是顺序的。事实上，从 Topic 角度来说，消息仍不是有序的。

Kafka 的设计原理决定，对于一个 topic，同一个 group 中不能有多于 partitions 个数的 consumer 同时消费，否则将意味着某些 consumer 将无法得到消息。

【保证（Guarantees）】

在一个高层次的 Kafka 中给予了以下几点保证：

- 发送到 Partitions 中的消息将会按照它接收的顺序追加到日志中。
- 对于消费者而言，它们消费消息的顺序和日志中消息顺序一致。
- 如果 Topic 的"replication factor"为 N，那么允许 N-1 个 Kafka 实例失效。

1.4 使用场景

【消息（Messaging）】

对于一些常规的消息系统，Kafka 是个不错的选择；partitons/replication 和容错，可以使 Kafka 良好的扩展性和性能优势。不过到目前为止，我们应该很清楚认识到，Kafka 并没有提供 JMS 中的"事务性"、"消息传输担保（消息确认机制）"、"消息分组"等企业级特性；Kafka 只能使用作为"常规"的消息系统，在一定程度上，尚未确保消息的发送与接收绝对可靠（比如消息重发、消息发送丢失等）。

在我们的经验里消息使用通常是较低的吞吐量，但可能需要较低的端到端的延迟，而这往往取决于 Kafka 提供强耐久性保证。在这一领域 Kafka 足以媲美传统的消息系统，如 ActiveMQ 或 RabbitMQ。

【网站活动跟踪（Website Activity Tracking）】

Kafka 可以作为"网站活性跟踪"的最佳工具；可以将网站的活动（页面访问量、搜索或其他用户可能需要的操作）信息发送到 Kafka 中，并实时处理（real-time processing）、实时监控（real-time monitoring）、或者离线统计分析（loading into Hadoop or offline data warehousing systems for offline processing and reporting）等。

【指标（Metrics）】

Kafka 经常被用于运行监控数据，这也包括分布式应用程序产生的操作数据信息。

【日志聚合（Log Aggregation）】

Kafka 的特性决定它非常适合作为"日志收集中心"；Application 可以将操作日志"批量""异步"的发送到 Kafka 集群中，而不是保存在本地或者 DB 中；Kafka 可以批量提交消息/压缩消息等，这对 producer 端而言，几乎感觉不到性能的开支。此时 consumer 端可以使 hadoop 等其他系统化的存储和分析系统。相比以日志为中心的系统（log-centric systems），如 Scribe

或Flume，Kafka提供同样良好的性能，更强的耐久性保证，这一切都归功于复制（replication）和较低的端到端延迟（much lower end-to-end latency）。

1.5 工作原理

Kafka 的**设计初衷**是希望作为一个统一的信息收集平台，能够实时的收集反馈信息，并需要能够支撑较大的数据量，且具备良好的容错能力。

ZooKeeper 是一个分布式的、分层级的文件系统，能促进客户端间的松耦合，并提供最终一致的，类似于传统文件系统中文件和目录的 Znode 视图。它提供了基本的操作，例如创建、删除和检查 Znode 是否存在。它提供了事件驱动模型，客户端能观察特定 Znode 的变化，例如现有 Znode 增加了一个新的子节点。ZooKeeper 运行多个 ZooKeeper 服务器，称为**Ensemble**，以获得高可用性。每个服务器都持有分布式文件系统的内存复本，为客户端的读取请求提供服务。

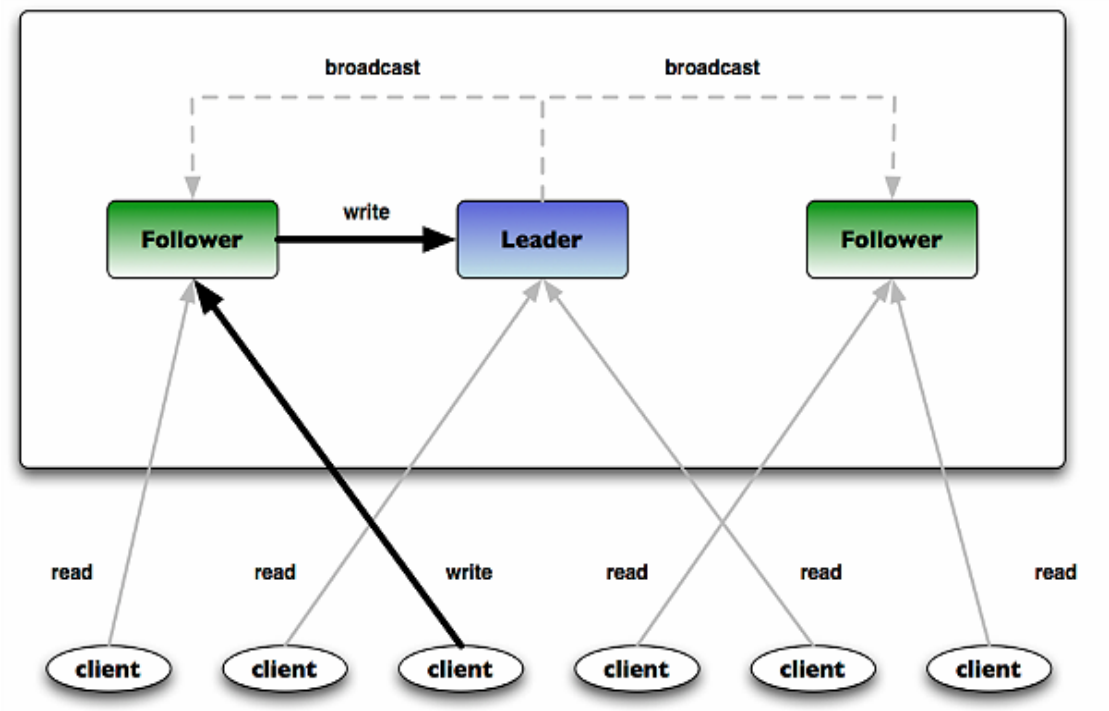


图 1.5 ZooKeeper Ensemble 架构

上图展示了典型的 ZooKeeper ensemble，一台服务器作为 Leader，其它作为 Follower。当 Ensemble 启动时，先选出 Leader，然后所有 Follower 复制 Leader 的状态。所有写请求都通过 Leader 路由，变更会广播给所有 Follower。变更广播被称为**原子广播**。

Kafka 中 ZooKeeper 的用途：正如 ZooKeeper 用于分布式系统的协调和促进，Kafka 使用 ZooKeeper 也是基于相同的原因。ZooKeeper 用于管理、协调 Kafka 代理。每个 Kafka 代理都通过 ZooKeeper 协调其它 Kafka 代理。当 Kafka 系统中新增了代理或者某个代理故障失效时，ZooKeeper 服务将通知生产者和消费者。生产者和消费者据此开始与其它代理协调工作。

【持久性】

Kafka 使用文件存储消息，这就直接决定 kafka 在性能上严重依赖文件系统的本身特性。且无论任何 OS 下，对文件系统本身的优化几乎没有可能。文件缓存/直接内存映射等是常用的手段。因为 Kafka 是对日志文件进行 append 操作，因此磁盘检索的开支是较小的；同时为了减少磁盘写入的次数，broker 会将消息暂时 buffer 起来，当消息的个数（或大小）达到一定阀值时，再 flush 到磁盘，这样减少了磁盘 IO 调用的次数。

【性能】

需要考虑的影响性能点很多，除磁盘 IO 之外，我们还需要考虑网络 IO，这直接关系到 Kafka 的吞吐量问题。Kafka 并没有提供太多高超的技巧；对于 producer 端，可以将消息 buffer 起来，当消息的条数达到一定阀值时，批量发送给 broker；对于 consumer 端也是一样，批量 fetch 多条消息。不过消息量的大小可以通过配置文件来指定。对于 Kafka broker 端，似乎有个 **sendfile 系统**调用可以潜在的提升网络 IO 的性能：将文件的数据映射到系统内存中，socket 直接读取相应的内存区域即可，而无需进程再次 copy 和交换。其实对于 producer/consumer/broker 三者而言，CPU 的开支应该都不大，因此启用消息压缩机制是一个良好的策略；压缩需要消耗少量的 CPU 资源，不过对于 Kafka 而言，网络 IO 更应该需要考虑。可以将任何在网络上传输的消息都经过压缩。Kafka 支持 gzip/snappy 等多种压缩方式。

【生产者】

负载均衡：producer 将会和 topic 下所有 partition leader 保持 socket 连接；消息由 producer 直接通过 socket 发送到 broker，中间不会经过任何"路由层"。事实上，消息被路由到哪个 partition 上，有 producer 客户端决定。比如可以采用"random"、"key-hash"、"轮询"等，如果一个 topic 中有多个 partitions，那么在 producer 端实现"消息均衡分发"是必要的。

其中 partition leader 的位置（host:port）注册在 zookeeper 中，producer 作为 zookeeper client，已经注册了 watch 用来监听 partition leader 的变更事件。

异步发送：将多条消息暂且在客户端 buffer 起来，并将他们批量的发送到 broker，小数据 IO 太多，会拖慢整体的网络延迟，批量延迟发送事实上提升了网络效率。不过这也有一定的隐患，比如说当 producer 失效时，那些尚未发送的消息将会丢失。

【消费者】

consumer 端向 broker 发送"fetch"请求，并告知其获取消息的 offset；此后 consumer 将会获得一定条数的消息；consumer 端也可以重置 offset 来重新消费消息。

在 JMS 实现中，Topic 模型基于 push 方式，即 broker 将消息推送给 consumer 端。不过在 Kafka 中，采用了 pull 方式，即 consumer 在和 broker 建立连接之后，主动去 pull（或者说 fetch）消息；这中模式有些优点，首先 consumer 端可以根据自己的消费能力适时的去 fetch 消息并处理，且可以控制消息消费的进度（offset）；此外，消费者可以良好的控制消息消费的数量，batch fetch。

其他 JMS 实现，消息消费的位置是有 prodiver 保留，以便避免重复发送消息或者将没有消费成功的消息重发等，同时还要控制消息的状态。这就要求 JMS broker 需要太多额外的工作。在 Kafka 中，partition 中的消息只有一个 consumer 在消费，且不存在消息状态的控制，也没有复杂的消息确认机制，可见 Kafka broker 端是相当轻量级的。当消息被 consumer 接收之后，consumer 可以在本地保存最后消息的 offset，并间歇性的向 zookeeper 注册 offset。由此可见，consumer 客户端也很轻量级。

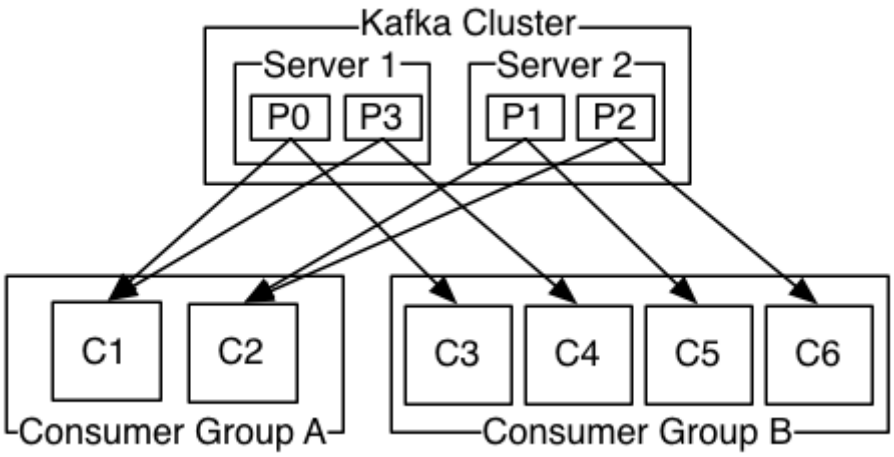


图 1.5 Kafka 集群

【消息传送机制】

对于 JMS 实现，消息传输担保非常直接：**有且只有一次（exactly once）**。在 kafka 中稍有不同：

1）**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。

2）**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。

3）**exactly once**：消息只会发送一次。

at most once：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：kafka 中并没有严格的去实现（基于 2 阶段提交，事务），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。（相比 at most once 而言，重复接收数据总比丢失数据要好）。

【复制备份】

Kafka 将每个 partition 数据复制到多个 server 上，任何一个 partition 有一个 leader 和多个 follower（可以没有）；备份的个数可以通过 broker 配置文件来设定。leader 处理所有的 read-write 请求，follower 需要和 leader 保持同步。follower 和 consumer 一样，消费消息并保存在本地日志中；leader 负责跟踪所有的 follower 状态，如果 follower"落后"太多或者失效，leader 将会把它从 replicas 同步列表中删除。当所有的 follower 都将一条消息保存成功，此消息才被认为是"committed"，那么此时 consumer 才能消费它。即使只有一个 replicas 实例存活，仍然可以保证消息的正常发送和接收，只要 zookeeper 集群存活即可。（不同于其他分布式存储，比如 hbase 需要"多数派"存活才行）

当 leader 失效时，需在 followers 中选取出新的 leader，可能此时 follower 落后于 leader，因此需要选择一个"**up-to-date**"的 follower。选择 follower 时需要兼顾一个问题，就是新 leader server 上所已经承载的 partition leader 的个数，如果一个 server 上有过多的 partition leader，意味着此 server 将承受着更多的 IO 压力。在选举新 leader，需要考虑到"负载均衡"。

【日志】

如果一个 topic 的名称为"my_topic"，它有 2 个 partitions，那么日志将会保存在 my_topic_0 和 my_topic_1 两个目录中；日志文件中保存了一序列"**log entries**"（日志条目），每个 log entry 格式为"**4 个字节的数字 N 表示消息的长度**" + "**N 个字节的消息内容**"；每个日志都有一个 offset 来唯一的标记一条消息，offset 的值为 8 个字节的数字，表示此消息在此 partition 中所处的起始位置。每个 partition 在物理存储层面，有多个 log file 组成（称为 segment）。segment file 的命名为"最小 offset".kafka。例如"00000000000.kafka"；其中"最小 offset"表示此 segment 中起始消息的 offset。

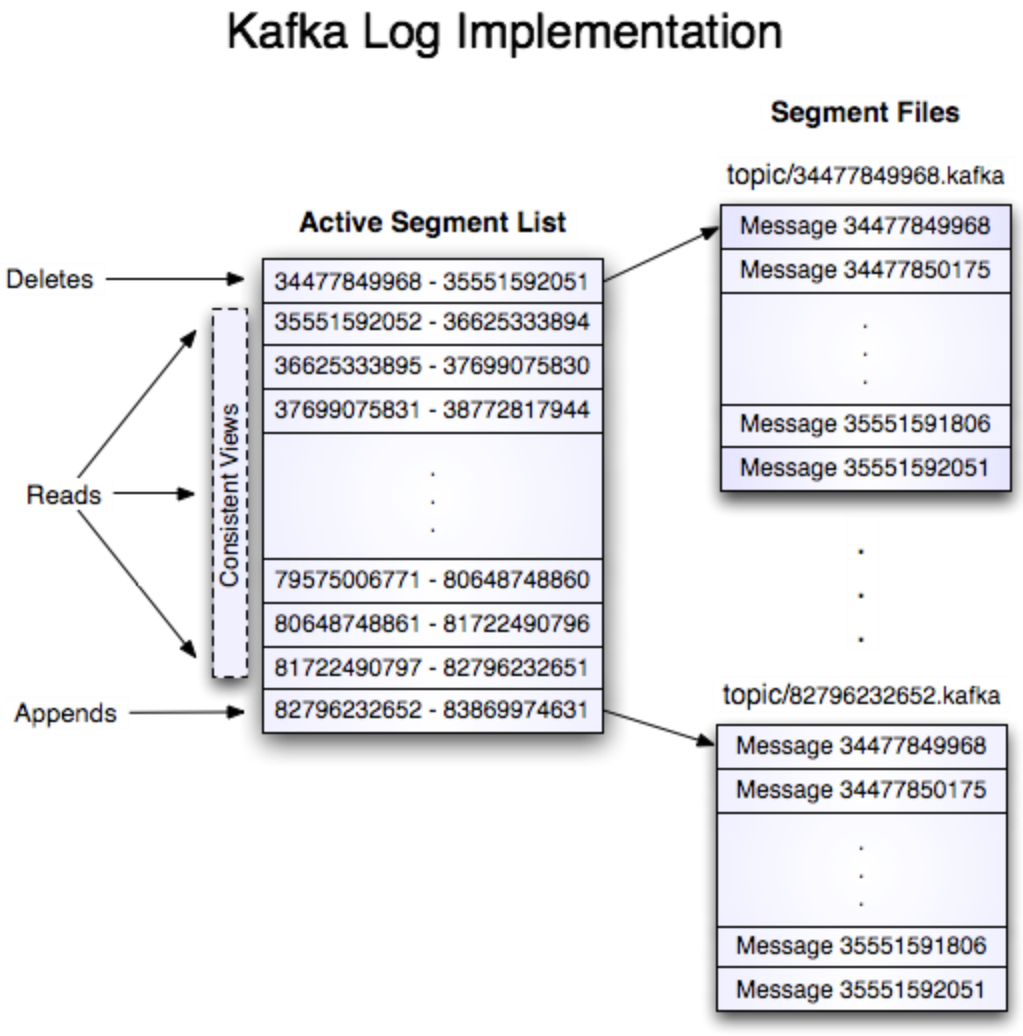


图 1.6 Kafka 日志实现

其中每个 partiton 中所持有的 segments 列表信息会存储在 zookeeper 中。

当 segment 文件尺寸达到一定阀值时（可以通过配置文件设定，默认 1G），将会创建一个新的文件；当 buffer 中消息的条数达到阀值时将会触发日志信息 flush 到日志文件中，同时如果"距离最近一次 flush 的时间差"达到阀值时，也会触发 flush 到日志文件。如果 broker

失效，极有可能会丢失那些尚未 flush 到文件的消息。因为 server 意外实现，仍然会导致 log 文件格式的破坏（文件尾部），那么就要求当 server 启动时需要检测最后一个 segment 的文件结构是否合法并进行必要的修复。

获取消息时，需要指定 offset 和最大 chunk 尺寸，offset 用来表示消息的起始位置，chunk size 用来表示最大获取消息的总长度（间接的表示消息的条数）。根据 offset，可以找到此消息所在 segment 文件，然后根据 segment 的最小 offset 取差值，得到它在 file 中的相对位置，直接读取输出即可。

日志文件的**删除策略**非常简单：启动一个后台线程定期扫描 log file 列表，把保存时间超过阀值的文件直接删除（根据文件的创建时间）。为了避免删除文件时仍然有 read 操作（consumer 消费），采取 **copy-on-write** 方式。

【分配】

Kafka 使用 zookeeper 来存储一些 meta 信息，并使用了 **zookeeper watch 机制**来发现 meta 信息的变更并作出相应的动作（比如 consumer 失效，触发负载均衡等）

1） **Broker Node Registry**：当一个 kafka broker 启动后，首先会向 zookeeper 注册自己的节点信息（临时 znode），同时当 broker 和 zookeeper 断开连接时，此 znode 也会被删除。

格式：/broker/ids/[0...N]　-->host:port；其中[0..N]表示 broker id，每个 broker 的配置文件中都需要指定**一个数字类型的 id（全局不可重复）**，znode 的值为此 broker 的 host:port 信息。

2）**Broker Topic Registry**：当一个 broker 启动时，会向 zookeeper 注册自己持有的 topic 和 partitions 信息，仍然是一个临时 znode。

格式：/broker/topics/[topic]/[0...N] ；其中[0..N]表示 partition 索引号。

3）**Consumer and Consumer Group**：每个 consumer 客户端被创建时，会向 zookeeper 注册自己的信息；此作用主要是为了"负载均衡"。

一个 group 中的多个 consumer 可以交错的消费一个 topic 的所有 partitions；简而言之，保证此 topic 的所有 partitions 都能被此 group 所消费，且消费时为了性能考虑，让 partition 相对均衡的分散到每个 consumer 上。

4） **Consumer Id Registry**：每个 consumer 都有一个唯一的 ID（host:uuid，可以通过配置文件指定，也可以由系统生成），此 ID 用来标记消费者信息。

格式：/consumers/[group_id]/ids/[consumer_id]

仍然是一个临时的 znode，此节点的值为{"topic_name":#streams...}，即表示此 consumer 目前所消费的 topic + partitions 列表。

5）**Consumer Offset Tracking**：用来跟踪每个 consumer 目前所消费的 partition 中最大的 offset。

格式：/consumers/[group_id]/offsets/[topic]/[broker_id-partition_id]　-->offset_value

此 znode 为持久节点，可以看出 offset 跟 group_id 有关，以表明当 group 中一个消费者失效，其他 consumer 可以继续消费。

6）**Partition Owner Registry**：用来标记 partition 正在被哪个 consumer 消费，临时 znode。

格式：/consumers/[group_id]/owners/[topic]/[broker_id-partition_id]　-->consumer_node_id

此节点表达了"一个 partition"只能被 group 下一个 consumer 消费，同时当 group 下某个 consumer 失效，那么将会触发负载均衡（即：让 partitions 在多个 consumer 间均衡消费，接管那些"游离"的 partitions）

当 consumer 启动时，所触发的操作：

A）首先进行"Consumer ID Registry"；

B）然后在"Consumer ID Registry"节点下注册一个 watch 用来监听当前 group 中其他 consumer 的"leave"和"join"；只要此 znode path 下节点列表变更，都会触发此 group 下 consumer 的负载均衡。（比如一个 consumer 失效，那么其他 consumer 接管 partitions）

C）在"Broker ID Registry"节点下，注册一个 watch 用来监听 broker 的存活情况；如果 broker 列表变更，将会触发所有的 groups 下的 consumer 重新 balance。

1.6 关键技术

在 Kafka 上，有两个原因可能导致低效：1）太多的网络请求 2）过多的字节拷贝。为了提高效率，Kafka 把 message 分成一组一组的，每次请求会把一组 message 发给相应的 consumer。 **此外，为了减少字节拷贝，使用了 sendfile 这个高级系统函数，也即 zero-copy 技术，极大地提高了数据传输的效率。**

程序读文件内容，调用 socket 发送内容，过程涉及以下几个步骤：

- 调用 syscall，如 read，陷入内核，读文件内容到内核缓存区 pagecache
- 程序将文件内容由内核缓存区拷贝到用户空间内存
- 调用 syscall，如 socket write 函数，陷入内核，将用户空间的内容拷贝的 socket 缓冲区内存，准备发送
- 将 socket 缓冲区内存拷贝到 NIC（Network Interface Controller）缓冲区，发送数据。

其中涉及了 2 次 syscall 和 4 次数据拷贝。相信读者一定发现这 4 次数据拷贝是显然没有必要的，直接把数据从 pagecache 的内核缓存区读到 NIC 缓冲区不就可以了吗？sendfile 系统函数做的就是这件事情。显然这会极大地提升数据传输的效率。在 java 中对应的函数调用是“**FileChannle.transferTo**”；另外，Kafka 还通过多条数据压缩传输、存取的办法来进一步提升吞吐率。

为了更加了解 zero-copy 技术，我将 IBM 的“**通过零拷贝实现有效数据传输**”文章附加在下面，可以详细的知道该技术的原理。

很多 Web 应用程序都会提供大量的静态内容，其数量多到相当于读完整个磁盘的数据再将同样的数据写回响应套接字（socket）。此动作看似只需较少的 CPU 活动，但它的效率非常低：首先内核读出全盘数据，然后将数据跨越内核用户推到应用程序，然后应用程序再次跨越内核用户将数据推回，写出到套接字。应用程序实际上在这里担当了一个不怎么高效的中介角色，将磁盘文件的数据转入套接字。

数据每遍历用户内核一次，就要被拷贝一次，这会消耗 CPU 周期和内存带宽。幸运的是，您可以通过一个叫 零拷贝— 很贴切 — 的技巧来消除这些拷贝。使用零拷贝的应用程序要求内核直接将数据从磁盘文件拷贝到套接字，而无需通过应用程序。零拷贝不仅大大地提高了应用程序的性能，而且还减少了内核与用户模式间的上下文切换。

Java 类库通过 java.nio.channels.FileChannel 中的 transferTo() 方法来在 Linux 和 UNIX 系统上支持零拷贝。可以使用 transferTo() 方法直接将字节从它被调用的通道上传输到另外一个可写字节通道上，数据无需流经应用程序。本文首先展示了通过传统拷贝语义进行的简单文件传输引发的开销，然后展示了使用 transferTo() 零拷贝技巧如何提高性能。

数据传输：传统方法

考虑一下从一个文件中读出数据并将数据传输到网络上另一程序的场景（这个场景表述出了很多服务器应用程序的行为，包括提供静态内容的 Web 应用程序、FTP 服务器、邮件服务器等）。操作的核心在清单 1 的两个调用中（参见 下载，查找完整示例代码的链接）：

清单 1. 把字节从文件拷贝到套接字

```
File.read(fileDesc, buf, len);
Socket.send(socket, buf, len);
```

清单 1 的概念很简单，但实际上，拷贝的操作需要四次用户模式和内核模式间的上下文切换，而且在操作完成前数据被复制了四次。图 1 展示了数据是如何在内部从文件移动到套接字的：

图 1. 传统的数据拷贝方法

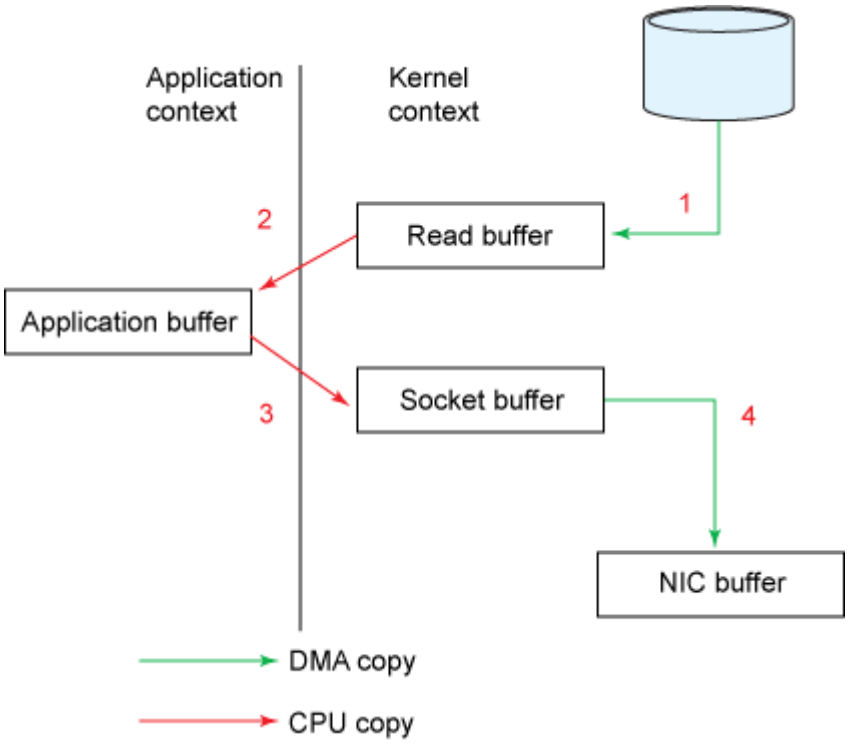
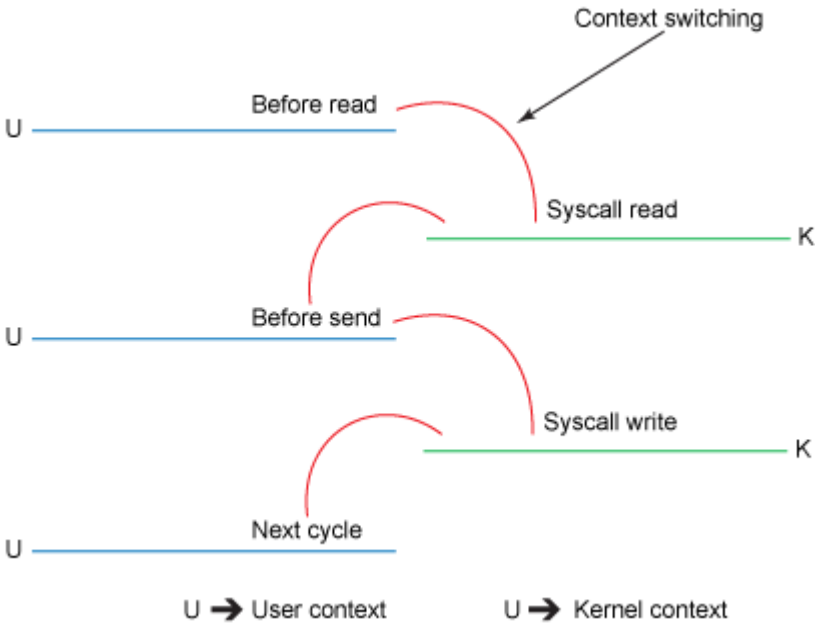


图 2 展示了上下文切换：



这里涉及的步骤有：

1. read() 调用（参见 图 2）引发了一次从用户模式到内核模式的上下文切换。在内部，发出 sys_read()（或等效内容）以从文件中读取数据。直接内存存取（direct memory access，DMA）引擎执行了第一次拷贝（参见 图 1），它从磁盘中读取文件内容，然后将它们存储到一个内核地址空间缓存区中。
2. 所需的数据被从读取缓冲区拷贝到用户缓冲区，read() 调用返回。该调用的返回引发了内核模式到用户模式的上下文切换（又一次上下文切换）。现在数据被储存在用户地址空间缓冲区。
3. send() 套接字调用引发了从用户模式到内核模式的上下文切换。数据被第三次拷贝，并被再次放置在内核地址空间缓冲区。但是这一次放置的缓冲区不同，该缓冲区与目标套接字相关联。
4. send() 系统调用返回，结果导致了第四次的上下文切换。DMA 引擎将数据从内核缓冲区传到协议引擎，第四次拷贝独立地、异步地发生 。

使用中间内核缓冲区（而不是直接将数据传输到用户缓冲区）看起来可能有点效率低下。但是之所以引入中间内核缓冲区的目的是想提高性能。在读取方面使用中间内核缓冲区，可以允许内核缓冲区在应用程序不需要内核缓冲区内的全部数据时，充当 “预读高速缓存（readahead cache）” 的角色。这在所需数据量小于内核缓冲区大小时极大地提高了性能。在写入方面的中间缓冲区则可以让写入过程异步完成。

不幸的是，如果所需数据量远大于内核缓冲区大小的话，这个方法本身可能成为一个性能瓶颈。数据在被最终传入到应用程序前，在磁盘、内核缓冲区和用户缓冲区中被拷贝了多次。**零拷贝通过消除这些冗余的数据拷贝而提高了性能。**

数据传输：零拷贝方法

再次检查传统场景，您就会注意到第二次和第三次拷贝根本就是多余的。应用程序只是起到缓存数据并将其传回到套接字的作用而以，别无他用。数据可以直接从读取缓冲区传输到套接字缓冲区。transferTo() 方法就能够让您实现这个操作。清单 2 展示了 transferTo() 的方法签名：

清单 2. transferTo() 方法

```
public void transferTo(long position, long count, WritableByteChannel target);
```

transferTo() 方法将数据从文件通道传输到了给定的可写字节通道。在内部，它依赖底层操作系统对零拷贝的支持；在 UNIX 和各种 Linux 系统中，此调用被传递到 sendfile() 系统调用中，如清单 3 所示，清单 3 将数据从一个文件描述符传输到了另一个文件描述符：

清单 3. sendfile() 系统调用

```
#include <sys/socket.h>
ssize_t sendfile(int out_fd, int in_fd, off_t *offset, size_t count);
```

清单 1 中的 file.read() 和 socket.send() 调用动作可以替换为一个单一的 transferTo() 调用，如清单 4 所示：

清单 4. 使用 transferTo() 将数据从磁盘文件拷贝到套接字

```
transferTo(position, count, writableChannel);
```

图 3 展示了使用 transferTo() 方法时的数据路径：

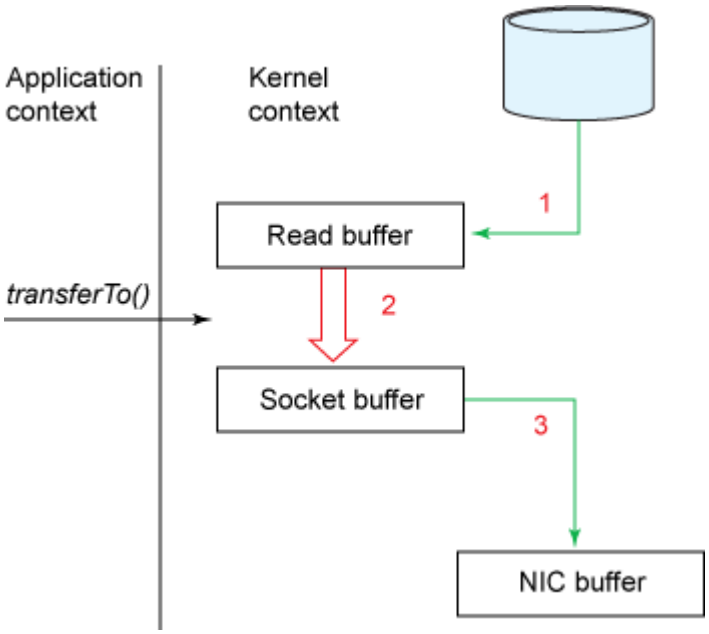
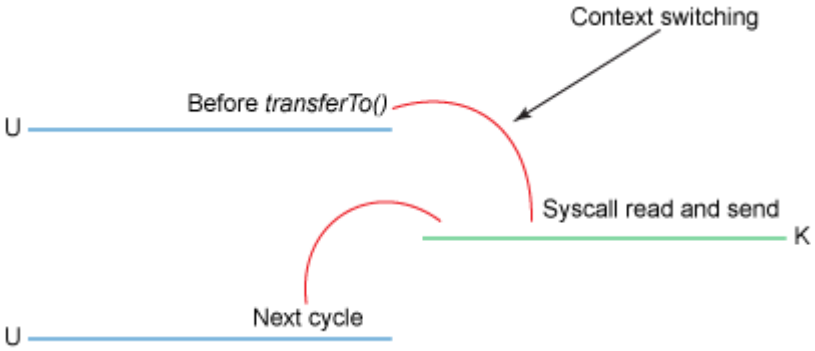


图 4. 使用 transferTo() 方法的上下文切换



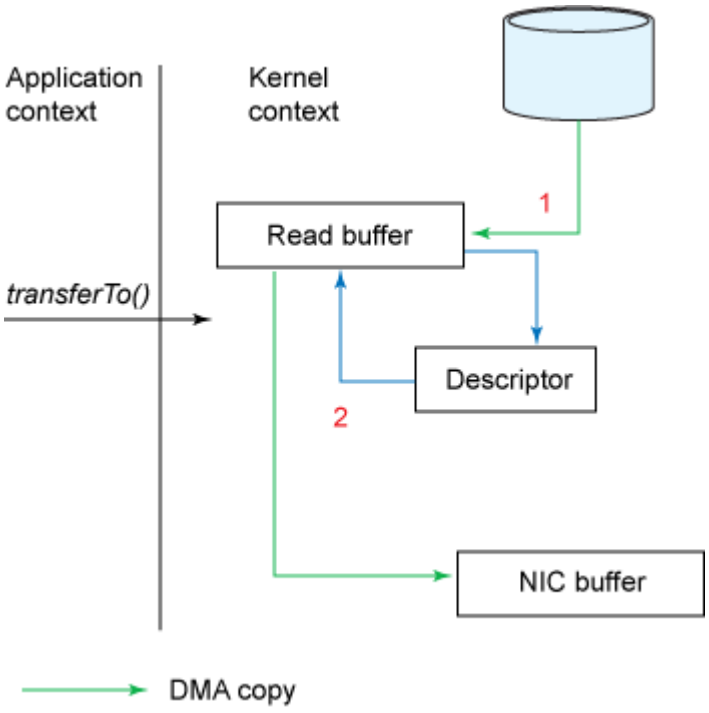
使用 清单 4 所示的 transferTo() 方法时的步骤有：

1. transferTo() 方法引发 DMA 引擎将文件内容拷贝到一个读取缓冲区。然后由内核将数据拷贝到与输出套接字相关联的内核缓冲区。
2. 数据的第三次复制发生在 DMA 引擎将数据从内核套接字缓冲区传到协议引擎时。

改进的地方：我们将上下文切换的次数从四次减少到了两次，将数据复制的次数从四次减少到了三次（其中只有一次涉及到了 CPU）。但是这个代码尚未达到我们的零拷贝要求。如果底层网络接口卡支持收集操作 的话，那么我们就可以进一步减少内核的数据复制。在 Linux 内核 2.4 及后期版本中，套接字缓冲区描述符就做了相应调整，以满足该需求。这种方法不仅可以减少多个上下文切换，还可以消除需要涉及 CPU 的重复的数据拷贝。对于用户方面，用法还是一样的，但是内部操作已经发生了改变：

1. transferTo() 方法引发 DMA 引擎将文件内容拷贝到内核缓冲区。
2. 数据未被拷贝到套接字缓冲区。取而代之的是，只有包含关于数据的位置和长度的信息的描述符被追加到了套接字缓冲区。DMA 引擎直接把数据从内核缓冲区传输到协议引擎，从而消除了剩下的最后一次 CPU 拷贝。

图 5 展示了结合使用 transferTo() 方法和收集操作的数据拷贝：



性能比较：构建一个文件服务器

接下来就让我们实际应用一下零拷贝，在客户机和服务器间传输文件（参见 下载，查找示例代码）。TraditionalClient.java 和 TraditionalServer.java 是基于传统的复制语义的，它们使用了 File.read() 和 Socket.send()。TraditionalServer.java 是一个服务器程序，它在一个特定的端口上监听要连接的客户机，然后以每次 4K 字节的速度从套接字读取数据。TraditionalClient.java 连接到服务器，从文件读取 4K 字节的数据（使用 File.read()），并将内容通过套接字发送到服务器（使用 socket.send()）。

TransferToServer.java 和 TransferToClient.java 执行的功能与此相同，但使用 transferTo() 方法（sendfile() 系统调用）来将文件从服务器传输到客户机。

我们在一个运行 2.6 内核的 Linux 系统上执行了示例程序，并以毫秒为单位分别度量了使用传统方法和 transferTo() 方法传输不同大小的文件的运行时间。表 1 展示了度量的结果：

表 1. 性能对比：传统方法与零拷贝

文件大小	正常文件传输（ms）	transferTo（ms）
7MB	156	45
21MB	337	128
63MB	843	387
98MB	1320	617
200MB	2124	1150
350MB	3631	1762
700MB	13498	4422
1GB	18399	8537

如您所见，与传统方法相比，transferTo() API 大约减少了 65% 的时间。这就极有可能提高了需要在 I/O 通道间大量拷贝数据的应用程序的性能，如 Web 服务器。

结束语

我们已经展示了使用 transferTo() 方法较使用传统方法 — 从一个通道读出数据并将其写入到另外一个通道 — 的性能优势。中间缓冲区拷贝 — 甚至于那些隐藏在内核内的拷贝 — 都会产生一定的开销。在需要在通道间大量拷贝数据的应用程序中，零拷贝技巧能够显著地提高性能。

2、Kafka 设计

2.1 消息持久化

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构（persistent structure）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。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，即使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：

- 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）。
- 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。

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。

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）。

2.2 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**，其间的详细过程如下所述：

- 操作系统将数据从磁盘中读取到内核空间里的页面缓存
- 应用程序将数据从内核空间读入到用户空间的缓冲区
- 应用程序将读到的数据写回内核空间并放入 socke 的缓冲区
- 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。

2.3 端到端压缩

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。

2.4 客户端状态

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。

2.5 使用者状态

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。

相关问题还有一个，就是到底是应该让使用者从代理那里把数据 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。

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。

2.6 生产者状态

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。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。

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。

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。

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。

3、Kafka 安装

2.1 运行环境

Kafka 官网下载地址：http://kafka.apache.org/downloads.html

0.8.2.1 is the latest release. The current stable version is 0.8.2.1.
You can verify your download by following these procedures and using these KEYS.

Source download: kafka-0.8.2.1-src.tgz (asc, md5)
Binary downloads:

- Scala 2.9.1 - kafka_2.9.1-0.8.2.1.tgz (asc, md5)
- Scala 2.9.2 - kafka_2.9.2-0.8.2.1.tgz (asc, md5)
- Scala 2.10 - kafka_2.10-0.8.2.1.tgz (asc, md5)
- Scala 2.11 - kafka_2.11-0.8.2.1.tgz (asc, md5)

We build for multiple versions of Scala. This only matters if you are using Scala and you want a version built for the same Scala version you use. Otherwise any version should work (2.10 is recommended).

备注：Scala 是一种针对 JVM 将函数和面向对象技术组合在一起的编程语言。Scala 编程语言近来抓住了很多开发者的眼球。它看起来像是一种纯粹的面向对象编程语言，而又无缝地结合了命令式和函数式的编程风格。Scala 的名称表明，它还是一种高度可伸缩的语言。Scala 的设计始终贯穿着一个理念：创造一种更好地支持组件的语言。Scala 融汇了许多前所未有的特性，而同时又运行于 JVM 之上。随着开发者对 Scala 的兴趣日增，以及越来越多的工具支持，无疑 Scala 语言将成为你手上一件必不可少的工具。

另外国内也有类似的产品——MetaQ，MetaQ（全称 Metamorphosis）是一个高性能、高可用、可扩展的分布式消息中间件，思路起源于 LinkedIn 的 Kafka，采用 Java 语言编写，但并不是 Kafka 的一个 Copy。MetaQ 具有消息存储顺序写、吞吐量大和支持本地和 XA 事务等特性，适用于大吞吐量、顺序消息、广播和日志数据传输等场景，目前在淘宝和支付宝有着广泛的应用。

2.2 主要配置

- Producer 主要配置：

```
##对于开发者而言,需要通过broker.list指定当前producer需要关注的broker列表
##producer通过和每个broker链接,并获取partitions,
##如果某个broker链接失败,将导致此上的partitons无法继续发布消息
##格式:host1:port,host2:port2,其中host:port需要参考broker配置文件.
##对于producer而言没有使用zookeeper自动发现broker列表，非常奇怪。(0.8V和0.7有区别)
metadata.broker.list=
##producer接收消息ack的时机.默认为0.
##0: producer不会等待broker发送ack
##1: 当leader接收到消息之后发送ack
##2: 当所有的follower都同步消息成功后发送ack.
request.required.acks=0
##producer消息发送的模式,同步或异步.
##异步意味着消息将会在本地buffer,并适时批量发送
##默认为sync,建议async
producer.type=sync
##消息序列化类,将消息实体转换成byte[]
serializer.class=kafka.serializer.DefaultEncoder
key.serializer.class=${serializer.class}
##partitions路由类,消息在发送时将根据此实例的方法获得partition索引号.
partitioner.class=kafka.producer.DefaultPartitioner

##消息压缩算法,none,gzip,snappy
compression.codec=none
##消息在producer端buffer的条数.仅在producer.type=async下有效
##batch.num.messages=200
```

- Consumer 主要配置：

```
##当前消费者的group名称,需要指定
group.id=
##consumer作为zookeeper client,需要通过zk保存一些meta信息,此处为zk connectString
zookeeper.connect=hostname1:port,hostname2:port2
##当前consumer的标识,可以设定,也可以有系统生成.
conusmer.id=
##获取消息的最大尺寸,broker不会像consumer输出大于此值的消息chunk
##每次feth将得到多条消息,此值为总大小
fetch.messages.max.bytes=1024*1024
##当consumer消费一定里的消息之后,将会自动向zookeeper提交offset信息
##注意offset信息并不是每消费一次消息,就像zk提交一次,而是现在本地保存,并定期提交
auto.commit.enable=true
##自动提交的时间间隔,默认为1分钟.
auto.commit.interval.ms=60*1000
```

- Broker 配置：

```
##broker标识,id为正数,且全局不得重复.
broker.id=1
##日志文件保存的目录
log.dirs=~/kafka/logs
##broker需要使用zookeeper保存meata信息,因此broker为zk client;
##此处为zookeeper集群的connectString,后面可以跟上path,比如
##hostname:port/chroot/kafka
##不过需要注意,path的全路径需要有自己来创建(使用zookeeper脚本工具)
zookeeper.connect=hostname1:port1,hostname2:port2
##用来侦听链接的端口,prudcer或consumer将在此端口建立链接
port=6667
##指定broker实例绑定的网络接口地址
host.name=
##每个partition的备份个数,默认为1,建议根据实际条件选择
##此致值大意味着消息各个server上同步时需要的延迟较高
num.partitions=2
##日志文件中每个segment文件的尺寸,默认为1G
##log.segment.bytes=1024*1024*1024
##滚动生成新的segment文件的最大时间
##log.roll.hours=24*7
##segment文件保留的最长时间,超时将被删除
##log.retention.hours=24*7
##partiton中buffer中,消息的条数,达到阀值,将触发flush到磁盘.
log.flush.interval.messages=10000
#消息buffer的时间,达到阀值,将触发flush到磁盘.
log.flush.interval.ms=3000
##partition leader等待follower同步消息的最大时间,
##如果超时,leader将follower移除同步列表
replica.lag.time.max.ms=10000
##允许follower落后的最大消息条数,如果达到阀值,将follower移除同步列表
##replica.lag.max.message=4000
##消息的备份的个数,默认为1
num.replica.fetchers=1
```

想要了解更多可以查看官网：http://kafka.apache.org/documentation.html#configuration。server.properties 中所有配置参数说明(解释)如下列表：

参数	说明(解释)
broker.id =0	每一个 broker 在集群中的唯一表示，要求是正数。当该服务器的 IP 地址发生改变时，broker.id 没有变化，则不会影响 consumers 的消息情况
log.dirs=/data/kafka-logs	kafka 数据的存放地址，多个地址的话用逗号分割,多个目录分布在不同磁盘上可以提高读写性能 /data/kafka-logs-1，/data/kafka-logs-2
port =9092	broker server 服务端口
message.max.bytes =6525000	表示消息体的最大大小，单位是字节
num.network.threads =4	broker 处理消息的最大线程数，一般情况下数量为 cpu 核数
num.io.threads =8	broker 处理磁盘 IO 的线程数，数值为 cpu 核数 2 倍
background.threads =4	一些后台任务处理的线程数，例如过期消息文件的删除等，一般情况下不需要去做修改
queued.max.requests =500	等待 IO 线程处理的请求队列最大数，若是等待 IO 的请求超过这个数值，那么会停止接受外部消息，应该是一种自我保护机制。
host.name	broker 的主机地址，若是设置了，那么会绑定到这个地址上，若是没有，会绑定到所有的接口上，并将其中之一发送到 ZK，一般不设置
socket.send.buffer.bytes=100*1024	socket 的发送缓冲区，socket 的调优参数 SO_SNDBUFF
socket.receive.buffer.bytes =100*1024	socket 的接受缓冲区，socket 的调优参数 SO_RCVBUFF
socket.request.max.bytes =100*1024*1024	socket 请求的最大数值，防止 serverOOM，message.max.bytes 必然要小于 socket.request.max.bytes，会被 topic 创建时的指定参数覆盖
log.segment.bytes =1024*1024*1024	topic 的分区是以一堆 segment 文件存储的，这个控制每个 segment 的大小，会被 topic 创建时的指定参数覆盖
log.roll.hours =24*7	这个参数会在日志 segment 没有达到 log.segment.bytes 设置的大小，也会强制新建一个 segment 会被 topic 创建时的指定参数覆盖
log.cleanup.policy = delete	日志清理策略选择有：delete 和 compact 主要针对过期数据的处理，或是日志文件达到限制的额度，会被 topic 创建时的指定参数覆盖
log.retention.minutes=300 或 log.retention.hours=24	数据文件保留多长时间，存储的最大时间超过这个时间会根据 log.cleanup.policy 设置数据清除策略 log.retention.bytes 和 log.retention.minutes 或 log.retention.hours 任意一个达到要求，都会执行删除有 2 删除数据文件方式： 按照文件大小删除：log.retention.bytes 按照 2 中不同时间粒度删除：分别为分钟，小时

log.retention.bytes=-1	topic 每个分区的最大文件大小，一个 topic 的大小限制 = 分区数 *log.retention.bytes 。 -1 没有大小限 log.retention.bytes 和 log.retention.minutes 任意一个达到要求，都会执行删除，会被 topic 创建时的指定参数覆盖
log.retention.check.interval.ms=5minutes	文件大小检查的周期时间，是否处罚 log.cleanup.policy 中设置的策略
log.cleaner.enable=false	是否开启日志清理
log.cleaner.threads = 2	日志清理运行的线程数
log.cleaner.io.max.bytes.per.second=None	日志清理时候处理的最大大小
log.cleaner.dedupe.buffer.size=500*1024*1024	日志清理去重时候的缓存空间，在空间允许的情况下，越大越好
log.cleaner.io.buffer.size=512*1024	日志清理时候用到的 IO 块大小一般不需要修改
log.cleaner.io.buffer.load.factor=0.9	日志清理中 hash 表的扩大因子一般不需要修改
log.cleaner.backoff.ms =15000	检查是否处罚日志清理的间隔
log.cleaner.min.cleanable.ratio=0.5	日志清理的频率控制，越大意味着更高效的清理，同时会存在一些空间上的浪费，会被 topic 创建时的指定参数覆盖
log.cleaner.delete.retention.ms=1day	对于压缩的日志保留的最长时间，也是客户端消费消息的最长时间，同 log.retention.minutes 的区别在于一个控制未压缩数据，一个控制压缩后的数据。会被 topic 创建时的指定参数覆盖
log.index.size.max.bytes=10*1024*1024	对于 segment 日志的索引文件大小限制，会被 topic 创建时的指定参数覆盖
log.index.interval.bytes =4096	当执行一个 fetch 操作后，需要一定的空间来扫描最近的 offset 大小，设置越大，代表扫描速度越快，但是也更好内存，一般情况下不需要搭理这个参数
log.flush.interval.messages=None 例如：log.flush.interval.messages=1000 表示每当消息记录数达到 1000 时 flush 一次数据到磁盘	log 文件”sync”到磁盘之前累积的消息条数,因为磁盘 IO 操作是一个慢操作,但又是一个”数据可靠性"的必要手段,所以此参数的设置,需要在"数据可靠性"与"性能"之间做必要的权衡.如果此值过大,将会导致每次"fsync"的时间较长(IO 阻塞),如果此值过小,将会导致"fsync"的次数较多,这也意味着整体的 client 请求有一定的延迟.物理 server 故障,将会导致没有 fsync 的消息丢失.
log.flush.scheduler.interval.ms =3000	检查是否需要固化到硬盘的时间间隔
log.flush.interval.ms = None 例如：log.flush.interval.ms=1000 表示每间隔 1000 毫秒 flush 一次数据到磁盘	仅仅通过 interval 来控制消息的磁盘写入时机,是不足的.此参数用于控制"fsync"的时间间隔,如果消息量始终没有达到阀值,但是离上一次磁盘同步的时间间隔达到阀值,也将触发.

log.delete.delay.ms =60000	文件在索引中清除后保留的时间一般不需要去修改
log.flush.offset.checkpoint.interval.ms =60000	控制上次固化硬盘的时间点，以便于数据恢复一般不需要去修改
auto.create.topics.enable =true	是否允许自动创建 topic，若是 false，就需要通过命令创建 topic
default.replication.factor =1	是否允许自动创建 topic，若是 false，就需要通过命令创建 topic
num.partitions =1	每个 topic 的分区个数，若是在 topic 创建时候没有指定的话会被 topic 创建时的指定参数覆盖
以下是 kafka 中 Leader,replicas 配置参数	
controller.socket.timeout.ms =30000	partition leader 与 replicas 之间通讯时,socket 的超时时间
controller.message.queue.size=10	partition leader 与 replicas 数据同步时,消息的队列尺寸
replica.lag.time.max.ms =10000	replicas 响应 partition leader 的最长等待时间，若是超过这个时间，就将 replicas 列入 ISR(in-sync replicas)，并认为它是死的，不会再加入管理中
replica.lag.max.messages =4000	如果 follower 落后与 leader 太多,将会认为此 follower[或者说 partition relicas]已经失效 ##通常,在 follower 与 leader 通讯时,因为网络延迟或者链接断开,总会导致 replicas 中消息同步滞后 ##如果消息之后太多,leader 将认为此 follower 网络延迟较大或者消息吞吐能力有限,将会把此 replicas 迁移 ##到其他 follower 中. ##在 broker 数量较少,或者网络不足的环境中,建议提高此值.
replica.socket.timeout.ms=30*1000	follower 与 leader 之间的 socket 超时时间
replica.socket.receive.buffer.bytes=64*1024	leader 复制时候的 socket 缓存大小
replica.fetch.max.bytes =1024*1024	replicas 每次获取数据的最大大小
replica.fetch.wait.max.ms =500	replicas 同 leader 之间通信的最大等待时间，失败了会重试
replica.fetch.min.bytes =1	fetch 的最小数据尺寸,如果 leader 中尚未同步的数据不足此值,将会阻塞,直到满足条件
num.replica.fetchers=1	leader 进行复制的线程数，增大这个数值会增加 follower 的 IO
replica.high.watermark.checkpoint.interval.ms =5000	每个 replica 检查是否将最高水位进行固化的频率
controlled.shutdown.enable =false	是否允许控制器关闭 broker ,若是设置为 true,会关闭所有在这个 broker 上的 leader，并转移到其他 broker
controlled.shutdown.max.retries =3	控制器关闭的尝试次数

controlled.shutdown.retry.backoff.ms =5000	每次关闭尝试的时间间隔
leader.imbalance.per.broker.percentage =10	leader 的不平衡比例，若是超过这个数值，会对分区进行重新的平衡
leader.imbalance.check.interval.seconds =300	检查 leader 是否不平衡的时间间隔
offset.metadata.max.bytes	客户端保留 offset 信息的最大空间大小
kafka 中 zookeeper 参数配置	
zookeeper.connect = localhost:2181	zookeeper 集群的地址，可以是多个，多个之间用逗号分割 hostname1:port1,hostname2:port2,hostname3:port3
zookeeper.session.timeout.ms=6000	ZooKeeper 的最大超时时间，就是心跳的间隔，若是没有反映，那么认为已经死了，不易过大
zookeeper.connection.timeout.ms =6000	ZooKeeper 的连接超时时间
zookeeper.sync.time.ms =2000	ZooKeeper 集群中 leader 和 follower 之间的同步实际那

2.3 安装启动

学习 kafka 的基础是先把 kafka 系统部署起来，然后简单的使用它，从直观上感觉它，然后逐步的深入了解它。

1）下载软件

我们这里下载“kafka_2.10-0.8.2.1.tgz”，从下载页面中有很多版本，有一个推荐版本。所以我们这里也选择推荐的版本。

```
wget http://mirrors.hust.edu.cn/apache/kafka/0.8.2.1/kafka_2.10-0.8.2.1.tgz
```

2）配置环境

Java 环境配置略，请安装 Java 1.6 以上版本，我们这里选择 Java 1.8 版本。

```
#Set Java Enviroment
export JAVA_HOME=$HOME/jdk1.8.0_45
export CLASSPATH=.:$JAVA_HOME/lib/tools/jar:$JAVA_HOME/lib/dt.jar

#Set Kafka Enviroment
export KAFKA_HOME=$HOME/kafka_2.10-0.8.2.1
export PATH=$PATH:KAFKA_HOME/bin
```

3）ZooKeeper 环境

如果是仅仅是使用，仅仅启动 Kafka 自带的 ZooKeeper 即可，如果是搭建 Kafka 分布

式，则需要搭建 ZooKeeper 分布式集群了，具体如何搭建可以参考我整理的《细细品味Hadoop_第 16 期_ZooKeeper 简介及安装_V1.2.pdf》。本期仅仅是 Kafka 简介和安装，那么相应的 Kafka 是我们的关注的重点，所以我们启动 Kafka 自带的即可。

```
cd /home/xiapistudio/kafka_2.10-0.8.2.1
bin/zookeeper-server-start.sh config/zookeeper.properties &
```

等启动完之后，可以通过下面命令进去 ZooKeeper 服务。

```
bin/zookeeper-shell.sh 127.0.0.1:2181
```

4）Kafka 配置

Kafka 的安装也分为三种方式：单机模式、集群模式，伪分布模式。单机模式安装则比较简单，直接解压安装文件，然后启动即可。而集群模式就需要分布式的 ZooKeeper 服务支持了，这样更加稳定，但是启动方式和单机模式一样，只是配置上面会有差异。另外伪分布式仅仅是在一台机器上面启动多个实例（broker），其配置文件为“server.properties”。

（1）单机模式

我们先来说一下单机模式，其实单机模式配置文件不需要任何更改，则可以启动。

```
bin/kafka-server-start.sh config/server.properties &
```

启动完后，则使用 jps 看一下相关的进程，相关信息如下：

```
[xiapi@xiapistudio kafka_2.10-0.8.2.1]$ jps
12673 Jps
13400 QuorumPeerMain
21821 Kafka
```

上面看到“QuorumPeerMain”是 ZooKeeper 服务的进程，而“Kafka”是 Kafka 服务的进程。下面我们说一下 Kafka 的集群搭建方式。

（2）集群模式

Kafka 最为重要三个配置依次为：broker.id、log.dir、port 和 zookeeper.connect。集群模式的 Kafka 在每一台机器启动一个 broker 实例，并使用分布式的 ZooKeeper 服务。

下面是我们集群的服务配置，假设你拥有 ZooKeeper 分布式集群。如果对配置参数不了解，可以看看上一小节的主要配置说明。

```
# The id of the broker. This must be set to a unique integer for each broker.
broker.id=1    （其他机器则为：2/3）

# The port the socket server listens on
port=9092
```

```
# A comma seperated list of directories under which to store log files
log.dirs=/tmp/kafka-logs

# Zookeeper connection string (see zookeeper docs for details).
# This is a comma separated host:port pairs, each corresponding to a zk
# server. e.g. "127.0.0.1:3000,127.0.0.1:3001,127.0.0.1:3002".
# You can also append an optional chroot string to the urls to specify the
# root directory for all kafka znodes.
zookeeper.connect=192.1.0.10:2181,192.1.0.11:2181,192.1.0.12:2181
```

在集群模式下，其配置除了“broker.id”每台机器不一样之外，别的都一样。切记在拷贝之后要更改一下相应的“broker.id”的值，其值为整数。上面的“log.dirs”可以根据自己的实际情况进行选择合适的目录，比如“/home/xiapistudio/logs/kafka”。启动方式和单机模式一致，这里不再叙说。

（3）伪分布模式

一般的伪分布模式，基本就是一台机器上面模拟分布式性能，Kafka 的伪分布式就是单机上面启动多个 broker 实例，每个实例对外的端口自然就不能一样了。这一点和 ZooKeeper 伪分布式有点类似。自然这些都要反应到配置上面了，那我们下面看一下配置如何配置，伪分布模式，我们需要拷贝多个 “server.properties”为“server-1.properties”、“server-2.properties”和“server-3.properties”

```
|--/home/xiapistudio/kafka_2.10-0.8.2.1/conf
    |---- server-1.properties
        |-- broker.id=1
        |-- port=9092
        |-- log.dirs=/tmp/kafka-logs-1
        |-- zookeeper.connect=127.0.0.1:2181
    |---- server-2.properties
        |-- broker.id=2
        |-- port=9093
        |-- log.dirs=/tmp/kafka-logs-2
        |-- zookeeper.connect=127.0.0.1:2181
    |---- server-3.properties
        |-- broker.id=3
        |-- port=9094
        |-- log.dirs=/tmp/kafka-logs-3
        |-- zookeeper.connect=127.0.0.1:2181
```

启动命令则为如下所示：

```
bin/kafka-server-start.sh config/server-1.properties &
bin/kafka-server-start.sh config/server-2.properties &
```

```
bin/kafka-server-start.sh config/server-3.properties &
```

5）关闭服务

如果要停止 Kafka 服务，可以使用下面命令，另外如果是单机模式，可以使用自带的命令停止 ZooKeeper 服务。

停止 Kafka 服务

```
bin/kafka-server-stop.sh
```

停止 ZooKeeper 服务

```
zookeeper-server-stop.sh
```

2.4 快速入门

1）简单练习

（1）创建 topic

创建一个叫做“test”的 topic，它只有一个分区，一个副本。

```
bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic test
```

可以通过 list 命令查看创建的 topic：

```
bin/kafka-topics.sh --list --zookeeper localhost:2181
>
    test
```

除了手动创建 topic，还可以配置 broker 让它自动创建 topic。

（2）发送消息

Kafka 使用一个简单的命令行 producer，从文件中或者从标准输入中读取消息并发送到服务端。默认的每条命令将发送一条消息。

运行 producer 并在控制台中输一些消息，这些消息将被发送到服务端：

```
bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test
>
    hello xiapistudio.
```

ctrl+c 可以退出发送。

（3）消费消息

Kafka 也有一个命令行 consumer 可以读取消息并输出到标准输出：

```
bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test
>
    hello xiapistudio.
```

你在一个终端中运行 consumer 命令行，另一个终端中运行 producer 命令行，就可以在一个终端输入消息，另一个终端读取消息。

这两个命令都有自己的可选参数，可以在运行的时候不加任何参数可以看到帮助信息。

（4）多个实例

刚才只是启动了单个 broker（单机模式），现在启动有 3 个 broker 组成的集群（伪分布式），这时候停掉单机模式的 Kafka，按照伪分布式模式启动 Kafka，然后执行下面的操作。

前面提到 broker.id 在集群中唯一的标注一个节点，因为在同一个机器上，所以必须制定不同的端口和日志文件，避免数据被覆盖。

创建一个拥有 3 个副本的 topic：

```
bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 3 --partitions 1 --topic my-replicated-topic
```

现在我们搭建了一个集群，怎么知道每个节点的信息呢？运行“"describe topics”命令就可以了：

```
bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic my-replicated-topic
>
Topic:my-replicated-topic       PartitionCount:1        ReplicationFactor:3     Configs:
    Topic: my-replicated-topic      Partition: 0    Leader: 2       Replicas: 2,3,1 Isr: 2,3,1
```

下面解释一下这些输出。第一行是对所有分区的一个描述，然后每个分区都会对应一行，因为我们只有一个分区所以下面就只加了一行。

- leader：负责处理消息的读和写，leader 是从所有节点中随机选择的.
- replicas：列出了所有的副本节点，不管节点是否在服务中.
- isr：工作中的复制节点的集合，也就是正在服务中的节点.

在我们的例子中，节点 1 是作为 leader 运行。

向 topic 发送消息：

```
bin/kafka-console-producer.sh --broker-list localhost:9092 --topic my-replicated-topic
>
    hello www.xiapistudio.com !
```

消费这些消息：

```
bin/kafka-console-consumer.sh --zookeeper localhost:2181 --from-beginning --topic my-replicated-topic
>
    hello www.xiapistudio.com !
```

从上面看 broker2 是 leader，我们现在把 broker2 干掉，测试一下 Kafka 的容错能力。

```
ps x | grep server-2.properties | grep -v grep | awk '{print $1}' | xargs kill -9
```

另外一个节点被选做了 leader，node 2 不再出现在 in-sync 副本列表中：

```
bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic my-replicated-topic
>
Topic:my-replicated-topic PartitionCount:1 ReplicationFactor:3 Configs:
    Topic: my-replicated-topic Partition: 0 Leader: 3 Replicas: 2,3,1 Isr: 3,1
```

虽然最初负责续写消息的 leader down 掉了，但之前的消息还是可以消费的：

```
bin/kafka-console-consumer.sh --zookeeper localhost:2181 --from-beginning --topic my-replicated-topic
>
    hello www.xiapistudio.com !
```

看来 Kafka 的容错机制还是不错的。

2）简单运维

（1）如何在 Kafka 上创建一个 Topic

- 脚本手工创建：

```
bin/kafka-topics.sh --zookeeper 10.214.9.14:2181 --replication-factor 2 --partitions 12 --create --topic my-topic
```

  - topic 指定 topic name
  - partitions 指定分区数，这个参数需要根据 broker 数和数据量决定，正常情况下，每个 broker 上两个 partition 最好；
  - replication-factor 指定 partition 的 replicas 数，建议设置为 2；

- 程序自动创建：

开启自动创建配置：auto.create.topics.enable=true

使用程序直接往 kafka 中相应的 topic 发送数据，如果 topic 不存在就会按默认配置进行创建。

（2）如何在 Kafka 上对一个 Topic 增加 partition

操作步骤如下：

通过 kafka-topics.sh 工具的 alter 命令，将 topic_test 的 partitions 从 12 增加到 20：

```
bin/kafka-topics.sh --zookeeper 10.214.9.14:2181 --partitions 20 --alter --topic my-topic
```

（3）如何在 Kafka 上对一个 Topic 增加 replicas

操作步骤如下：

通过指手动写扩充 replicas 的配置文件，然后使用工具进行操作。

第一步：查看 topic 的详细信息

```
bin/kafka-topics.sh --zookeeper 10.214.9.14:2181 --describe --topic my-topic
>
Topic:my-topic PartitionCount:12 ReplicationFactor:1 Configs:
Topic: my-topic Partition: 0 Leader: 0 Replicas: 0 Isr: 0
Topic: my-topic Partition: 1 Leader: 1 Replicas: 1 Isr: 1
Topic: my-topic Partition: 2 Leader: 2 Replicas: 2 Isr: 2
Topic: my-topic Partition: 3 Leader: 0 Replicas: 0 Isr: 0
Topic: my-topic Partition: 4 Leader: 1 Replicas: 1 Isr: 1
Topic: my-topic Partition: 5 Leader: 2 Replicas: 2 Isr: 2
Topic: my-topic Partition: 6 Leader: 0 Replicas: 0 Isr: 0
Topic: my-topic Partition: 7 Leader: 1 Replicas: 1 Isr: 1
Topic: my-topic Partition: 8 Leader: 2 Replicas: 2 Isr: 2
Topic: my-topic Partition: 9 Leader: 0 Replicas: 0 Isr: 0
Topic: my-topic Partition: 10 Leader: 1 Replicas: 1 Isr: 1
Topic: my-topic Partition: 11 Leader: 2 Replicas: 2 Isr: 2
```

第二步：修改配置文件

将原有 replicas 为[0]扩充为[0,4], [1]扩充为[1,5],[2]扩充为[2,3]

```
[xiapi@xiapistudio kafka]$ cat partitions-to-move.json
{
 “partitions”:
[
{
 “topic”: “my-topic”,
 “partition”: 0,
 “replicas”: [0,4]
},
{
 “topic”: “my-topic”,
 “partition”: 1,
 “replicas”: [1,5]
},
{
```

```
 “topic”: “my-topic”,
 “partition”: 2,
 “replicas”: [2,3]
},
{
 “topic”: “my-topic”,
 “partition”: 3,
 “replicas”: [0,4]
},
{
 “topic”: “my-topic”,
 “partition”: 4,
 “replicas”: [1,5]
},
{
 “topic”: “my-topic”,
 “partition”: 5,
 “replicas”: [2,3]
},
{
 “topic”: “my-topic”,
 “partition”: 6,
 “replicas”: [0,4]
},
{
 “topic”: “my-topic”,
 “partition”: 7,
 “replicas”: [1,5]
},
{
 “topic”: “my-topic”,
 “partition”: 8,
 “replicas”: [2,3]
},
{
 “topic”: “my-topic”,
 “partition”: 9,
 “replicas”: [0,4]
},
{
 “topic”: “my-topic”,
 “partition”: 10,
 “replicas”: [1,5]
},
```

```
{
 “topic”: “my-topic”,
 “partition”: 11,
 “replicas”: [2,3]
}
],
 “version”:1
}
```

第三步：执行操作

```
bin/kafka-reassign-partitions.sh --zookeeper 10.214.9.14:2181 --execute --reassignment-json-file partitions-to-move.json
```

第四步：检查修改情况

```
[xiapi@xiapistudio kafka]$ bin/kafka-topics.sh --zookeeper 10.214.9.14:2181 --describe --topic my-topic
Topic:my-topic PartitionCount:12 ReplicationFactor:2 Configs:
Topic: my-topic Partition: 0 Leader: 0 Replicas: 0,4 Isr: 0,4
Topic: my-topic Partition: 1 Leader: 1 Replicas: 1,5 Isr: 1,5
Topic: my-topic Partition: 2 Leader: 2 Replicas: 2,3 Isr: 2,3
Topic: my-topic Partition: 3 Leader: 0 Replicas: 0,4 Isr: 0,4
Topic: my-topic Partition: 4 Leader: 1 Replicas: 1,5 Isr: 1,5
Topic: my-topic Partition: 5 Leader: 2 Replicas: 2,3 Isr: 2,3
Topic: my-topic Partition: 6 Leader: 0 Replicas: 0,4 Isr: 0,4
Topic: my-topic Partition: 7 Leader: 1 Replicas: 1,5 Isr: 1,5
Topic: my-topic Partition: 8 Leader: 2 Replicas: 2,3 Isr: 2,3
Topic: my-topic Partition: 9 Leader: 0 Replicas: 0,4 Isr: 0,4
Topic: my-topic Partition: 10 Leader: 1 Replicas: 1,5 Isr: 1,5
Topic: my-topic Partition: 11 Leader: 2 Replicas: 2,3 Isr: 2,3
```

（4）如何在 Kafka 中修改 Topic 的 preferred replica

操作背景：

假如 topic my-topic 中 partition 0 的 replicas 为[0,4]，则 0 为 preferred replica，应该成为 leader。这时我们期望 4 为 preferred replica，并变成 leader。

执行步骤如下：

第一步：查看当前的 topic 详细信息

```
xiapi@xiapistudio:~$ bin/kafka-topics.sh --zookeeper 10.214.9.14:2181--describe --topic my-topic
Topic:my-topic PartitionCount:12 ReplicationFactor:2 Configs:
Topic: my-topic Partition: 0 Leader: 0 Replicas: 0,4 Isr: 0,4
```

```
Topic: my-topic Partition: 1 Leader: 1 Replicas: 1,5 Isr: 1,5
Topic: my-topic Partition: 2 Leader: 2 Replicas: 2,3 Isr: 2,3
Topic: my-topic Partition: 3 Leader: 0 Replicas: 0,4 Isr: 0,4
Topic: my-topic Partition: 4 Leader: 1 Replicas: 1,5 Isr: 1,5
Topic: my-topic Partition: 5 Leader: 2 Replicas: 2,3 Isr: 2,3
Topic: my-topic Partition: 6 Leader: 0 Replicas: 0,4 Isr: 0,4
Topic: my-topic Partition: 7 Leader: 1 Replicas: 1,5 Isr: 1,5
Topic: my-topic Partition: 8 Leader: 2 Replicas: 2,3 Isr: 2,3
Topic: my-topic Partition: 9 Leader: 0 Replicas: 0,4 Isr: 0,4
Topic: my-topic Partition: 10 Leader: 1 Replicas: 1,5 Isr: 1,5
Topic: my-topic Partition: 11 Leader: 2 Replicas: 2,3 Isr: 2,3
```

第二步：修改 replicas 的顺序

```
xiapi@xiapistudio:~$ cat partitions-to-move.json
{
 “partitions”:
[
{
 “topic”: “my-topic”,
 “partition”: 0,
 “replicas”: [4,0]
}
],
 “version”:1
}

xiapi@xiapistudio:~$ bin/kafka-reassign-partitions.sh --zookeeper 10.214.9.14:2181 --execute --reassignment-json-file partitions-to-move.json
```

第三步：更改 leader

```
xiapi@xiapistudio:~$ cat topic-partition-list.json
{
 “partitions”:
[
{“topic”: “my-topic”, “partition”: “0”}
]
}

xiapi@xiapistudio:~$ bin/kafka-preferred-replica-election.sh --zookeeper 10.214.9.14:2181 --path-to-json-file topic-partition-list.json
```

第四步：检查 replicas leader 切换情况

```
xiapi@xiapistudio:~$ bin/kafka-topics.sh --zookeeper 10.214.9.14:2181 --describe --topic my-topic
Topic:my-topic PartitionCount:12 ReplicationFactor:2 Configs:
Topic: my-topic Partition: 0 Leader: 4 Replicas: 4,0 Isr: 0,4
Topic: my-topic Partition: 1 Leader: 1 Replicas: 1,5 Isr: 1,5
Topic: my-topic Partition: 2 Leader: 2 Replicas: 2,3 Isr: 2,3
Topic: my-topic Partition: 3 Leader: 0 Replicas: 0,4 Isr: 0,4
Topic: my-topic Partition: 4 Leader: 1 Replicas: 1,5 Isr: 1,5
Topic: my-topic Partition: 5 Leader: 2 Replicas: 2,3 Isr: 2,3
Topic: my-topic Partition: 6 Leader: 0 Replicas: 0,4 Isr: 0,4
Topic: my-topic Partition: 7 Leader: 1 Replicas: 1,5 Isr: 1,5
Topic: my-topic Partition: 8 Leader: 2 Replicas: 2,3 Isr: 2,3
Topic: my-topic Partition: 9 Leader: 0 Replicas: 0,4 Isr: 0,4
Topic: my-topic Partition: 10 Leader: 1 Replicas: 1,5 Isr: 1,5
Topic: my-topic Partition: 11 Leader: 2 Replicas: 2,3 Isr: 2,3
```

（5）如何在 Kafka 中对 Topic 的 leader 进行均衡

操作背景：

在创建一个 topic 时，kafka 尽量将 partition 均分在所有的 brokers 上，并且将 replicas 也 j 均分在不同的 broker 上。

每个 partitiion 的所有 replicas 叫做"assigned replicas"，"assigned replicas"中的第一个 replicas 叫"preferred replica"，刚创建的 topic 一般"preferred replica"是 leader。leader replica 负责所有的读写。

但随着时间推移，broker 可能会停机，会导致 leader 迁移，导致机群的负载不均衡。我们期望对 topic 的 leader 进行重新负载均衡，让 partition 选择"preferred replica"做为 leader。

- 对所有 Topics 进行操作：

```
bin/kafka-preferred-replica-election.sh --zookeeper 10.214.9.14:2181
```

- 对某个 Topic 进行操作：

```
[xiapi@xiapistudio kafka]$ cat topic-partition-list.json
{
 "partitions":
  [
    {"topic":"test.example", "partition": "0"}
  ]
}

bin/kafka-preferred-replica-election.sh --zookeeper 10.214.9.14:2181 --path-to-json-file topic-partition-list.json
```

（6）Apache Kafka 中 topic 级别配置

- topic 级别配置用法

配置 topic 级别参数时，相同(参数)属性 topic 级别会覆盖全局的，否则默认为全局配置属性值。

创建 topic 参数可以设置一个或多个--config "Property(属性)",下面是创建一个 topic 名称为"my-topic"例子，它设置了 2 个参数 max message size 和 flush rate:

✧ 创建 topic 时配置参数

```
bin/kafka-topics.sh --zookeeper 10.214.9.14:2181 --create --topic my-topic --partitions 1 --replication-factor 1 --config max.message.bytes=64000 --config flush.messages=1
```

✧ 修改 topic 时配置参数

覆盖已经有 topic 参数，下面例子修改"my-topic"的 max message 属性

```
bin/kafka-topics.sh --zookeeper10.214.9.14:2181 --alter --topic my-topic --config max.message.bytes=128000
```

✧ 删除 topic 级别配置参数

```
bin/kafka-topics.sh --zookeeper10.214.9.14:2181 --alter --topic my-topic --delete-config max.message.bytes
```

✧ topic 级别在 zookeeper 存储结构

笔者配置的 zk kafka 集群的根目录为“/”，因此所有节点信息都在此目录下。

"my-topic"在 zk 上路径为“/config/topics/my-topic”,存储内容如下：

```
{
    "version": 1,
    "config": {
        "max.message.bytes": "12800000",
        "flush.messages": "1000"
    }
}
```

- topic 级别配置属性表

以下是 topic 级别配置， kafak server 中默认配置为下表“Server Default Property”列，当需要设置 topic 级别配置时，属性设置为“Property(属性)”列。

Property (属性)	Default (默认值)	Server Default Property	说明解释
cleanup.policy	delete	log.cleanup.policy	日志清理策略选择有：delete 和 compact 主要针对过期数据的处理，或

			是日志文件达到限制的额度，会被 topic 创建时的指定参数覆盖
delete.retention.ms	86400000 (24 hours)	log.cleaner.delete.retention.ms	对于压缩的日志保留的最长时间，也是客户端消费消息的最长时间，同 log.retention.minutes 的区别在于一个控制未压缩数据，一个控制压缩后的数据。会被 topic 创建时的指定参数覆盖
flush.messages	None	log.flush.interval.messages	log 文件”sync”到磁盘之前累积的消息条数,因为磁盘 IO 操作是一个慢操作,但又是一个”数据可靠性"的必要手段,所以此参数的设置,需要在"数据可靠性"与"性能"之间做必要的权衡.如果此值过大,将会导致每次"fsync"的时间较长(IO 阻塞),如果此值过小,将会导致"fsync"的次数较多,这也意味着整体的 client 请求有一定的延迟.物理 server 故障,将会导致没有 fsync 的消息丢失.
flush.ms	None	log.flush.interval.ms	仅仅通过 interval 来控制消息的磁盘写入时机,是不足的.此参数用于控制"fsync"的时间间隔,如果消息量始终没有达到阀值,但是离上一次磁盘同步的时间间隔达到阀值,也将触发.
index.interval.bytes	4096	log.index.interval.bytes	当执行一个 fetch 操作后，需要一定的空间来扫描最近的 offset 大小，设置越大，代表扫描速度越快，但是也更好内存，一般情况下不需要搭理这个参数
message.max.bytes	1,000,000	message.max.bytes	表示消息的最大大小，单位是字节
min.cleanable.dirty.ratio	0.5	log.cleaner.min.cleanable.ratio	日志清理的频率控制，越大意味着更高效的清理，同时会存在一些空间上的浪费，会被 topic 创建时的指定参数覆盖
retention.bytes	None	log.retention.bytes	topic 每个分区的最大文件大小，一个 topic 的大小限制 = 分区数 *log.retention.bytes。-1 没有大小限 log.retention.bytes 和 log.retention.minutes 任意一个达到要求，都会执行删除，会被 topic 创建时的指定参数覆盖

retention.ms	None	log.retention.minutes	数据存储的最大时间超过这个时间会根据 log.cleanup.policy 设置的策略处理数据，也就是消费端能够多久去消费数据 log.retention.bytes 和 log.retention.minutes 达到要求，都会执行删除，会被 topic 创建时的指定参数覆盖
segment.bytes	1 GB	log.segment.bytes	topic 的分区是以一堆 segment 文件存储的，这个控制每个 segment 的大小，会被 topic 创建时的指定参数覆盖
segment.index.bytes	10 MB	log.index.size.max.bytes	对于 segment 日志的索引文件大小限制，会被 topic 创建时的指定参数覆盖
log.roll.hours	7 days	log.roll.hours	这个参数会在日志 segment 没有达到 log.segment.bytes 设置的大小，也会强制新建一个 segment 会被 topic 创建时的指定参数覆盖

（6）Apache Kafka 下线 broker 的操作

操作背景：

主动下线是指 broker 运行正常，因为机器需要运维（升级操作系统，添加磁盘等）而主动停止 broker

分两种情况处理：

✧　所有的 topic 的 replica >= 2

此时，直接停止一个 broker，会自动触发 leader election 操作，不过目前 leader election 是逐个 partition 进行，等待所有 partition 完成 leader election 耗时较长，这样不可服务的时间就比较长。为了缩短不可服务时间窗口，可以主动触发停止 broker 操作，这样可以逐个 partition 转移，直到所有 partition 完成转移，再停止 broker。

```
bin/kafka-run-class.sh kafka.admin.ShutdownBroker --zookeeper 10.214.9.14:2181 --broker #brokerId# --num.retries 3 --retry.interval.ms 60
```

然后 shutdown broker

```
bin/kafka-server-stop.sh
```

✧　存在 topic 的 replica=1

当存在 topic 的副本数小于 2，只能手工把当前 broker 上这些 topic 对应的 partition 转移到其他 broker 上。当此 broker 上剩余的 topic 的 replica > 2 时，参照上面的方法继续处理。

4、参考文献

感谢以下文章的编写作者，没有你们的铺路，我或许会走得很艰难，参考不分先后，贡献同等珍贵。

1） http://www.oschina.net/translate/kafka-design
2） http://www.open-open.com/lib/view/open1354277579741.html
3） http://kafka.apache.org/documentation.html
4） http://shift-alt-ctrl.iteye.com/blog/1930345
5） http://www.ibm.com/developerworks/cn/java/j-zerocopy/
6） http://blog.csdn.net/lizhitao/article/details/24991799
7） http://www.cnblogs.com/likehua/p/3999538.html
8） http://blog.csdn.net/lizhitao/article/details/24991799
9） http://blog.csdn.net/desilting/article/details/22872839
10）http://liyonghui160com.iteye.com/blog/2105824
11）http://czj4451.iteye.com/blog/2041096
12）http://blog.csdn.net/lizhitao/article/details/39499283

5、打赏小编

	编辑简介： 高级软件工程师（T5），河北工业大学硕士研究生，现在就职于百度在线网络技术（北京）有限公司。专注于大数据以及其相关研究，在离线计算和实时计算方面有较为深入的研究，积累了丰富的实战经验。热衷于知识分享，其细细品味系列教程深受网友喜爱。
姓名：解耀伟	网站：www.xiapistudio.com
笔名：虾皮	博客：http://www.cnblogs.com/xia520pi/
扣扣：461052034	邮箱：xieyaowei1986@163.com

从高考复习开始养成了总结的习惯，习惯于在学习的过程中，把相关的文章融会贯通，并加以实践，结合自己的实际情况把相关的内容整理成册，便于学习和总结。在这几年里陆陆续续分享了很多细细品味系列文章。

每一期文章都耗费了不少的心血，很多时候都是在星期天业余的时间完成，现在也建立了自己独立的网站：www.xiapistudio.com，需要一些资金来维持，同时也可以鼓励我写更多的好东西来分享。如果你看了本文章对自己有用，可以通过支付宝的形式来进行打赏，1 元、2 元、10 元皆可，多少并不重要，只要你感觉文章使你受益即可。

温馨提示：在转账时，可以写明“打赏虾皮”或者“打赏虾皮工作室”。我的支付宝已经进行实名认证，支付宝是的个人头像，请认准后再支付。