《KAFKA官方文档》入门指南

[**1.入门指南**](http://kafka.apache.org/documentation/#gettingStarted)

[**1.1简介**](http://kafka.apache.org/documentation/#introduction)

**Apache的Kafka™是一个分布式流平台(*a distributed streaming platform*)。这到底意味着什么？**

我们认为，一个流处理平台应该具有三个关键能力：

1. 它可以让你发布和订阅记录流。在这方面，它类似于一个消息队列或企业消息系统。
2. 它可以让你持久化收到的记录流，从而具有容错能力。
3. 它可以让你处理收到的记录流。

Kafka擅长哪些方面？

它被用于两大类应用：

1. 建立实时流数据管道从而能够可靠地在系统或应用程序之间的共享数据
2. 构建实时流应用程序，能够变换或者对数据
3. 进行相应的处理。

想要了解Kafka如何具有这些能力，让我们从下往上深入探索Kafka的能力。

首先，明确几个概念：

* Kafka是运行在一个或多个服务器的集群(Cluster)上的。
* Kafka集群分类存储的记录流被称为主题(Topics)。
* 每个消息记录包含一个键，一个值和时间戳。

Kafka有四个核心API：

* [生产者 API](http://kafka.apache.org/documentation.html#producerapi) 允许应用程序发布记录流至一个或多个Kafka的话题(Topics)。
* [消费者API](http://kafka.apache.org/documentation.html#consumerapi)允许应用程序订阅一个或多个主题，并处理这些主题接收到的记录流。
* [Streams API](http://kafka.apache.org/documentation/streams)允许应用程序充当*流处理器（stream processor）*，从一个或多个主题获取输入流，并生产一个输出流至一个或多个的主题，能够有效地变换输入流为输出流。
* [Connector API](http://kafka.apache.org/documentation.html#connect)允许构建和运行可重用的生产者或消费者，能够把 Kafka主题连接到现有的应用程序或数据系统。例如，一个连接到关系数据库的连接器(connector)可能会获取每个表的变化。

Kafka的客户端和服务器之间的通信是靠一个简单的，高性能的，与语言无关的[TCP协议](https://kafka.apache.org/protocol.html)完成的。这个协议有不同的版本，并保持向后兼容旧版本（向前兼容旧版本？）。Kafka不光提供了一个Java客户端，还有[许多语言](https://cwiki.apache.org/confluence/display/KAFKA/Clients)版本的客户端。

[**主题和日志**](http://kafka.apache.org/documentation/#intro_topics)

让我们先来了解Kafka的核心抽象概念记录流 – 主题。

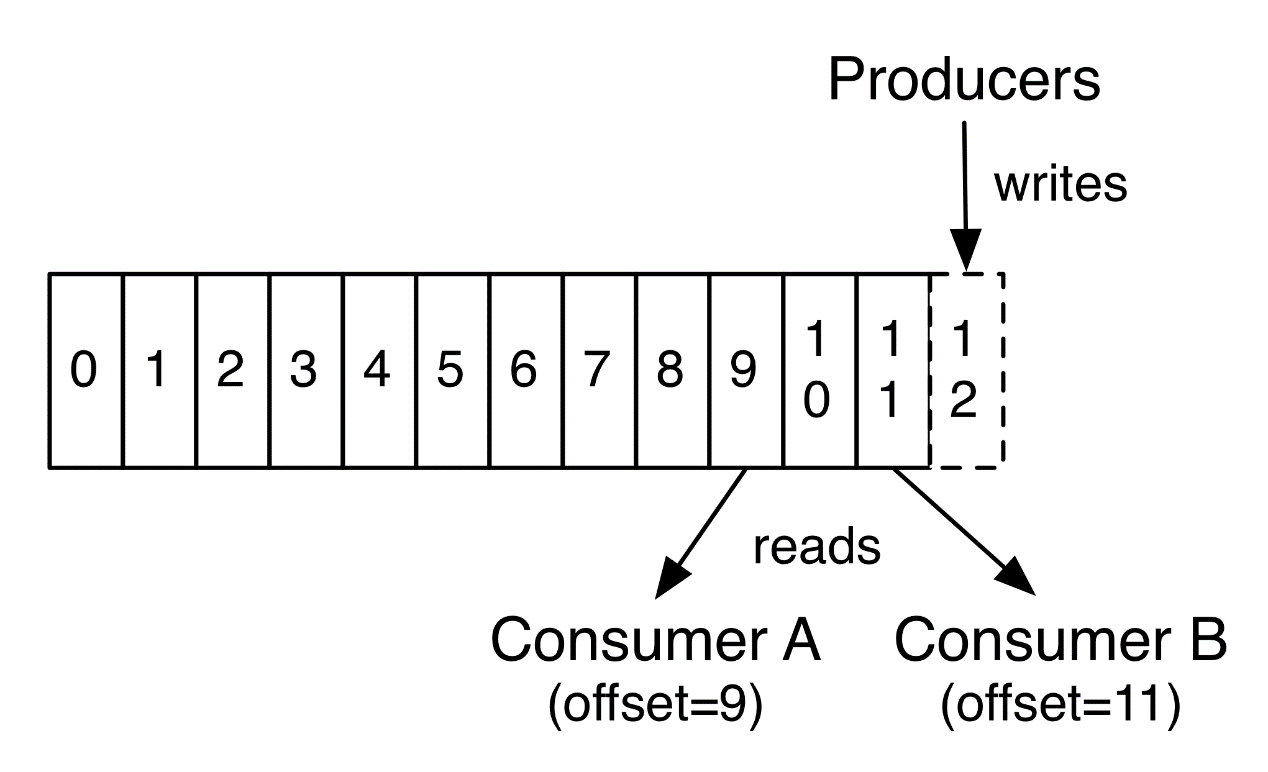
主题是一种分类或发布的一系列记录的名义上的名字。Kafka的主题始终是支持多用户订阅的; 也就是说，一个主题可以有零个，一个或多个消费者订阅写入的数据。

对于每一个主题，Kafka集群保持一个分区日志文件，看下图：



每个分区是一个有序的，不可变的消息序列，新的消息不断追加到这个有组织的有保证的日志上。分区会给每个消息记录分配一个顺序ID号 – *偏移量，* *能够*唯一地标识该分区中的每个记录。

Kafka集群保留所有发布的记录，不管这个记录有没有被消费过，Kafka提供可配置的保留策略去删除旧数据(还有一种策略根据分区大小删除数据)。例如，如果将保留策略设置为两天，在记录公布后两天，它可用于消费，之后它将被丢弃以腾出空间。Kafka的性能跟存储的数据量的大小无关， 所以将数据存储很长一段时间是没有问题的。



事实上，保留在每个消费者元数据中的最基础的数据就是消费者正在处理的当前记录的偏移量(offset)或位置(position)。这种偏移是由消费者控制：通常偏移会随着消费者读取记录线性前进，但事实上，因为其位置是由消费者进行控制，消费者可以在任何它喜欢的位置读取记录。例如，消费者可以恢复到旧的偏移量对过去的数据再加工或者直接跳到最新的记录，并消费从“现在”开始的新的记录。

这些功能的结合意味着，实现Kafka的消费者的代价都是很小的，他们可以增加或者减少而不会对集群或其他消费者有太大影响。例如，你可以使用我们的命令行工具去追随任何主题，而且不会改变任何现有的消费者消费的记录。

数据日志的分区，一举数得。首先，它们允许数据能够扩展到更多的服务器上去。每个单独的分区的大小受到承载它的服务器的限制，但一个话题可能有很多分区，以便它能够支持海量的的数据。其次，更重要的意义是分区是进行并行处理的基础单元。

**分布式**

日志的分区会跨服务器的分布在Kafka集群中，每个服务器会共享分区进行数据请求的处理。每个分区可以配置一定数量的副本分区提供容错能力。

每个分区都有一个服务器充当“leader”和零个或多个服务器充当“followers”。 leader处理所有的读取和写入分区的请求，而followers被动的从领导者拷贝数据。如果leader失败了，followers之一将自动成为新的领导者。每个服务器可能充当一些分区的leader和其他分区的follower，这样的负载就会在集群内很好的均衡分配。

[**生产者**](http://kafka.apache.org/documentation/#intro_producers)

生产者发布数据到他们所选择的主题。生产者负责选择把记录分配到主题中的哪个分区。这可以使用轮询算法( round-robin)进行简单地平衡负载，也可以根据一些更复杂的语义分区算法（比如基于记录一些键值）来完成。

[**消费者**](http://kafka.apache.org/documentation/#intro_consumers)

消费者以*消费群（consumer group* *）*的名称来标识自己，每个发布到主题的消息都会发送给订阅了这个主题的消费群里面的一个消费者的一个实例。消费者的实例可以在单独的进程或单独的机器上。

如果所有的消费者实例都属于相同的消费群，那么记录将有效地被均衡到每个消费者实例。

如果所有的消费者实例有不同的消费群，那么每个消息将被广播到所有的消费者进程。



两个服务器的Kafka集群具有四个分区（P0-P3）和两个消费群。A消费群有两个消费者，B群有四个。

更常见的是，我们会发现主题有少量的消费群，每一个都是“逻辑上的订阅者”。每组都是由很多消费者实例组成，从而实现可扩展性和容错性。这只不过是发布 – 订阅模式的再现，区别是这里的订阅者是一组消费者而不是一个单一的进程的消费者。

Kafka消费群的实现方式是通过分割日志的分区，分给每个Consumer实例，使每个实例在任何时间点的都可以“公平分享”独占的分区。维持消费群中的成员关系的这个过程是通过Kafka动态协议处理。如果新的实例加入该组，他将接管该组的其他成员的一些分区; 如果一个实例死亡，其分区将被分配到剩余的实例。

Kafka只保证一个分区内的消息有序，不能保证一个主题的不同分区之间的消息有序。分区的消息有序与依靠主键进行数据分区的能力相结合足以满足大多数应用的要求。但是，如果你想要保证所有的消息都绝对有序可以只为一个主题分配一个分区，虽然这将意味着每个消费群同时只能有一个消费进程在消费。

**保证**

Kafka提供了以下一些高级别的保证：

* 由生产者发送到一个特定的主题分区的消息将被以他们被发送的顺序来追加。也就是说，如果一个消息M1和消息M2都来自同一个生产者，M1先发，那么M1将有一个低于M2的偏移，会更早在日志中出现。
* 消费者看到的记录排序就是记录被存储在日志中的顺序。
* 对于副本因子N的主题，我们将承受最多N-1次服务器故障切换而不会损失任何的已经保存的记录。

对这些保证的更多细节可以参考文档的设计部分。

[**Kafka作为消息系统**](http://kafka.apache.org/documentation/#kafka_mq)

如何将Kafka的流的概念和传统的企业信息系统作比较？

消息处理模型历来有两种：[队列](http://en.wikipedia.org/wiki/Message_queue)和[发布-订阅](http://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern)。在队列模型中，一组消费者可以从服务器读取记录，每个记录都会被其中一个消费者处理; 在发布-订阅模式里，记录被广播到所有的消费者。这两种模式都具有一定的优点和弱点。队列的优点是它可以让你把数据分配到多个消费者去处理，它可以让您扩展你的处理能力。不幸的是，队列不支持多个订阅者，一旦一个进程读取了数据，这个数据就会消失。发布-订阅模式可以让你广播数据到多个进程，但是因为每一个消息发送到每个订阅者，没办法对订阅者处理能力进行扩展。

Kafka的消费群的推广了这两个概念。消费群可以像队列一样让消息被一组进程处理（消费群的成员），与发布 – 订阅模式一样，Kafka可以让你发送广播消息到多个消费群。

Kafka的模型的优点是，每个主题都具有这两个属性，它可以扩展处理能力，也可以实现多个订阅者，没有必要二选一。

Kafka比传统的消息系统具有更强的消息顺序保证的能力。

传统的消息队列的消息在队列中是有序的，多个消费者从队列中消费消息，服务器按照存储的顺序派发消息。然而，尽管服务器是按照顺序派发消息，但是这些消息记录被异步传递给消费者，消费者接收到的消息也许已经是乱序的了。这实际上意味着消息的排序在并行消费中都将丢失。消息系统通常靠 “排他性消费”( exclusive consumer)来解决这个问题，只允许一个进程从队列中消费，当然，这意味着没有并行处理的能力。

Kafka做的更好。通过一个概念：并行性-分区-主题实现主题内的并行处理，Kafka是能够通过一组消费者的进程同时提供排序保证和负载均衡。每个主题的分区指定给每个消费群中的一个消费者，使每个分区只由该组中的一个消费者所消费。通过这样做，我们确保消费者是一个分区唯一的读者，从而顺序的消费数据。因为有许多的分区，所以负载还能够均衡的分配到很多的消费者实例上去。但是请注意，一个消费群的消费者实例不能比分区数量多。

**Kafka作为存储系统**

任何消息队列都能够解耦消息的生产和消费，还能够有效地存储正在传送的消息。Kafka与众不同的是，它是一个非常好的存储系统。

Kafka把消息数据写到磁盘和备份分区。Kafka允许生产者等待返回确认，直到副本复制和持久化全部完成才认为成功，否则则认为写入服务器失败。

Kafka使用的磁盘结构很好扩展，Kafka将执行相同的策略不管你是有50 KB或50TB的持久化数据。

由于存储的重要性，并允许客户控制自己的读取位置，你可以把Kafka认为是一种特殊用途的分布式文件系统，致力于高性能，低延迟的有保障的日志存储，能够备份和自我复制。

**Kafka流处理**

只是读，写，以及储存数据流是不够的，目的是能够实时处理数据流。

在Kafka中，流处理器是从输入的主题连续的获取数据流，然后对输入进行一系列的处理，并生产连续的数据流到输出主题。

例如，零售应用程序可能需要输入销售和出货量，根据输入数据计算出重新订购的数量和调整后的价格，然后输出到主题。

这些简单处理可以直接使用生产者和消费者的API做到。然而，对于更复杂的转换Kafka提供了一个完全集成的[流API](http://kafka.apache.org/documentation/streams)。这允许应用程序把一些重要的计算过程从流中剥离或者加入流一起。

这种设施可帮助解决这类应用面临的难题：处理杂乱的数据，改变代码去重新处理输入，执行有状态的计算等

流API建立在Kafka提供的核心基础单元之上：它使用生产者和消费者的API进行输入输出，使用Kafka存储有状态的数据，并使用群组机制在一组流处理实例中实现容错。

**把功能组合起来**

消息的传输，存储和流处理的组合看似不寻常却是Kafka作为流处理平台的关键。

像HDFS分布式文件系统，允许存储静态文件进行批量处理。像这样的系统允许存储和处理*过去的历史数据*。

传统的企业消息系统允许处理您订阅后才抵达的消息。这样的系统只能处理将来到达的数据。

Kafka结合了这些功能，这种结合对Kafka作为流应用平台以及数据流处理的管道至关重要。

通过整合存储和低延迟订阅，流处理应用可以把过去和未来的数据用相同的方式处理。这样一个单独的应用程序，不但可以处理历史的，保存的数据，当它到达最后一条记录不会停止，继续等待处理未来到达的数据。这是泛化了的的流处理的概念，包括了批处理应用以及消息驱动的应用。

同样，流数据处理的管道结合实时事件的订阅使人们能够用Kafka实现低延迟的管道; 可靠的存储数据的能力使人们有可能使用它传输一些重要的必须保证可达的数据。可以与一个定期加载数据的线下系统集成，或者与一个因为维护长时间下线的系统集成。流处理的组件能够保证转换（处理）到达的数据。

有关Kafka提供的保证，API和功能的更多信息，看其余[文件](http://kafka.apache.org/documentation.html)。

[**1.2使用案例**](http://kafka.apache.org/documentation/#uses)

下面描述了一些使用Apache Kafka™的流行用例。更多的关于这些领域实践的概述，参考这个博客。

[**消息**](http://kafka.apache.org/documentation/#uses_messaging)

Kafka能够很好的替代传统的消息中间件。消息中间件由于各种原因被使用（解耦数据的生产和消费，缓冲未处理的消息等）。相较于大多数消息处理系统，Kafka有更好的吞吐量，内置分区，副本复制和容错性，使其成为大规模消息处理应用的理想解决方案。

根据我们的经验消息的使用通常具有相对低的吞吐量，但可能需要端到端的低延迟，以及高可靠性的保证，这种低延迟和可靠性的保证恰恰是Kafka能够提供的。

在这一领域Kafka是能够和传统的消息系统相媲美的，例如[ActiveMQ](http://activemq.apache.org/)或 [RabbitMQ](https://www.rabbitmq.com/)。

[**网站活动跟踪**](http://kafka.apache.org/documentation/#uses_website)

最初的用例是用Kafka重建一个用户活动跟踪管道使之作为一组实时发布 – 订阅的数据源。这意味着网站活动（网页浏览，搜索，或其他可能的操作）被当作一组中心主题发布，每种活动被当作一个主题。这些数据源（feeds）可被一系列的应用订阅，包括实时处理，实时监测，加载到Hadoop系统或离线数据仓库系统进行离线处理和报告。

活动追踪通常会产生巨大的数据量，因为每个用户页面的浏览都会产生很多的活动消息。

[**测量**](http://kafka.apache.org/documentation/#uses_metrics)

Kafka通常用于监测数据的处理。这涉及从分布式应用程序聚集统计数据，生产出集中的运行数据源feeds（以便订阅）。

[**日志聚合**](http://kafka.apache.org/documentation/#uses_logs)

许多人用Kafka作为日志聚合解决方案的替代品。日志聚合通常从服务器收集物理日志文件，并把它们放在一个集中的地方（文件服务器或HDFS）进行处理。Kafka抽象了文件的详细信息，把日志或事件数据的简洁抽象作为消息流传输。这为低时延的处理提供支持，而且更容易支持多个数据源和分布式的数据消费。相比集中式的日志处理系统，Scribe or Flume，Kafka提供同样良好的性能，而且因为副本备份提供了更强的可靠性保证和更低的端到端延迟。

[**流处理**](http://kafka.apache.org/documentation/#uses_streamprocessing)

Kafka的流数据管道在处理数据的时候包含多个阶段，其中原始输入数据从Kafka主题被消费然后汇总，加工，或转化成新主题用于进一步的消费或后续处理。例如，用于推荐新闻文章的数据流处理管道可能从RSS源抓取文章内容，并将其发布到“文章”主题; 进一步的处理可能是标准化或删除重复数据，然后发布处理过的文章内容到一个新的话题; 最后的处理阶段可能会尝试推荐这个内容给用户。这样的数据流处理管道基于各个主题创建了实时数据数据流程图。从版本0.10.0.0开始，Apache Kafka加入了轻量级的但功能强大的流处理库[Kafka Streams](http://kafka.apache.org/%7B%7Bversion%7D%7D/documentation/streams) ，Kafka Streams支持如上所述的数据处理。除了Kafka Streams，可以选择的开源流处理工具包括 [Apache Storm](https://storm.apache.org/) and [Apache Samza](http://samza.apache.org/).

[**Event Sourcing**](http://kafka.apache.org/documentation/#uses_eventsourcing)

[Event sourcing](http://martinfowler.com/eaaDev/EventSourcing.html) 是一种应用程序设计风格，是按照时间顺序记录的状态变化的序列。Kafka的非常强大的存储日志数据的能力使它成为构建这种应用程序的极好的后端选择。

[**Commit Log**](http://kafka.apache.org/documentation/#uses_commitlog)

Kafka可以为分布式系统提供一种外部提交日志(commit-log)服务。日志有助于节点之间复制数据，并作为一种数据重新同步机制用来恢复故障节点的数据。Kafka的[log compaction](http://kafka.apache.org/documentation.html" \l "compaction) 功能有助于支持这种用法。Kafka在这种用法中类似于[Apache BookKeeper](http://zookeeper.apache.org/bookkeeper/) 项目。

[**1.3快速开始**](http://kafka.apache.org/documentation/#quickstart)

本教程假设你从零开始，没有现成的Kafka或ZooKeeper数据。由于Kafka控制台脚本在Unix基础的和Windows平台上的不同，在Windows平台上使用bin\windows\，而不是bin/，并修改脚本扩展为.bat。

[**第1步：下载代码**](http://kafka.apache.org/documentation/#quickstart_download)

[下载](https://www.apache.org/dyn/closer.cgi?path=/kafka/0.10.2.0/kafka_2.11-0.10.2.0.tgz)0.10.2.0释放和un-tar它。

> **tar -xzf kafka\_2.11-0.10.2.0.tgz**

> **cd kafka\_2.11-0.10.2.0**

[**第2步：启动服务器**](http://kafka.apache.org/documentation/#quickstart_startserver)

Kafka使用ZooKeeper的，所以你需要先启动ZooKeeper的服务器，如果你还没有，您可以使用Kafka包装里的方便脚本来得到一个快速和污染的单节点的ZooKeeper实例。

> **bin/zookeeper-server-start.sh config/zookeeper.properties**

[2013-04-22 15:01:37,495] INFO Reading configuration from: config/zookeeper.properties (org.apache.zookeeper.server.quorum.QuorumPeerConfig)

...

现在启动Kafka服务器：

> **bin/kafka-server-start.sh config/server.properties**

[2013-04-22 15:01:47,028] INFO Verifying properties (kafka.utils.VerifiableProperties)

[2013-04-22 15:01:47,051] INFO Property socket.send.buffer.bytes is overridden to 1048576 (kafka.utils.VerifiableProperties)

...

[**第3步：创建一个话题**](http://kafka.apache.org/documentation/#quickstart_createtopic)

让我们创建一个名为“test”主题，只有一个分区，只有一个副本：

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

现在我们可以看到，如果我们运行的列表主题命令话题：

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

test

除了手动创建主题，你还可以配置你的代理服务器(broker)，当一个不存在的主题被发布的时候它能自动创建相应的主题。

[**第4步：发送一些消息**](http://kafka.apache.org/documentation/#quickstart_send)

Kafka带有一个命令行客户端，获取从文件或来自标准输入的输入，并作为消息发送到Kafka集群。默认情况下，每一行将被作为单独的消息发送。

运行生产者脚本，然后输入一些信息到控制台发送到服务器。

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

**This is a message**

**This is another message**

[**第5步：启动消费者**](http://kafka.apache.org/documentation/#quickstart_consume)

Kafka也有一个命令行消费者，将收到的消息输出到标准输出。

> **bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic test --from-beginning**

This is a message

This is another message

如果你在不同的终端上运行上面的命令，那么你现在应该能看到从生产者终端输入的消息会出现在消费者终端。

所有的命令行工具都有其他选项; 不带参数运行命令将显示更加详细的使用信息。

[**第6步：设置多代理群集**](http://kafka.apache.org/documentation/#quickstart_multibroker)

到目前为止，我们已经运行了单个代理的服务器，但是这没有乐趣。对于Kafka，一个代理是只有一个单节点的集群，因此多代理集群只是比开始多了一些代理实例外，没有什么太大的变化。但只是为了感受一下，我们的集群扩展到三个节点（所有的节点还是在本地机器上）。

首先，我们为每个经纪人做一个配置文件（在Windows上使用copy命令来代替）：

> **cp config/server.properties config/server-1.properties**

> **cp config/server.properties config/server-2.properties**

现在，编辑这些新文件和设置以下属性：

config/server-1.properties:

broker.id=1

listeners=PLAINTEXT://:9093

log.dir=/tmp/kafka-logs-1

config/server-2.properties:

broker.id=2

listeners=PLAINTEXT://:9094

log.dir=/tmp/kafka-logs-2

该broker.id属性是集群中的每个节点的唯一和永久的名字。我们要重写端口和日志目录，因为我们都在同一台机器上运行这些代理，我们要防止经纪人在同一端口上注册或覆盖彼此的数据。

我们已经有Zookeeper服务和我们的单个节点服务，所以我们只需要启动两个新节点：

> **bin/kafka-server-start.sh config/server-1.properties &**

...

> **bin/kafka-server-start.sh config/server-2.properties &**

...

现在，创建一个新的具有三个的副本因子的主题：

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

好了，现在我们有一个集群，但是如何才能知道哪个代理节点在做什么？要查看运行“describe topics”命令：

> **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: 1 Replicas: 1,2,0 Isr: 1,2,0

下面是输出的解释。第一行给出了所有分区的摘要，每个附加的行提供了一个分区的信息。由于我们只有一个分区，所以这个主题只有一行。

* “Leader”，负责指定分区所有读取和写入的节点。每个节点将是一部分随机选择的分区中的领导者。
* “Replicas”是此分区日志的节点列表集合，不管这些节点是否是领导者或者只是还活着(不在in-sync状态)。
* “ISR”是一组”in-sync” 节点列表的集合。这个列表包括目前活着并跟leader保持同步的replicas，Isr 是Replicas的子集。

请注意，在我的例子节点1是该主题的唯一分区中的leader。

我们可以运行相同的命令看看我们创建原来的话题的状态：

> **bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic test**

Topic:test PartitionCount:1 ReplicationFactor:1 Configs:

Topic: test Partition: 0 Leader: 0 Replicas: 0 Isr: 0

所以毫不奇怪，原来的话题没有副本，只有我们创建它时的唯一的服务器0。

让我们发布一些消息到我们新的话题：

> **bin/kafka-console-producer.sh --broker-list localhost:9092 --topic my-replicated-topic**

...

**my test message 1**

**my test message 2**

**^C**

现在让我们来消费这些消息：

> **bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --from-beginning --topic my-replicated-topic**

...

my test message 1

my test message 2

**^C**

现在，让我们测试容错性。代理1是领导者，让我们杀死它：

> **ps aux | grep server-1.properties**

*7564* ttys002 0:15.91 /System/Library/Frameworks/JavaVM.framework/Versions/1.8/Home/bin/java...

> **kill -9 7564**

在Windows上使用：

> **wmic process get processid,caption,commandline | find "java.exe" | find "server-1.properties"**

java.exe java -Xmx1G -Xms1G -server -XX:+UseG1GC ... build\libs\kafka\_2.10-0.10.2.0.jar" kafka.Kafka config\server-1.properties *644*

> **taskkill /pid 644 /f**

领导权已经切换到备机中的一个节点上去了，节点1不再在同步中的副本集(in-sync replica set)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: 2 Replicas: 1,2,0 Isr: 2,0

但消息仍然是可用于消费，即使是原来负责写任务的领导者已经不在了：

> **bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --from-beginning --topic my-replicated-topic**

...

my test message 1

my test message 2

**^C**

[**第7步：使用Kafka连接导入/导出数据**](http://kafka.apache.org/documentation/#quickstart_kafkaconnect)

从控制台写入数据和写回控制台是一个很方便入门的例子，但你可能想用Kafka使用其他来源的数据或导出Kafka的数据到其他系统。相对于许多系统需要编写定制集成的代码，您可以使用Kafka连接到系统去导入或导出数据。

Kafka Connect是包括在Kafka中一个工具，用来导入导出数据到Kafka。它是*connectors*的一个可扩展工具，其执行定制逻辑，用于与外部系统交互。在这个快速入门，我们将看到如何使用Kafka Connect做一些简单的连接器从一个文件导入数据到Kafka的主题，和将主题数据导出到一个文件。

首先，我们需要创建一些原始数据来开始测试：

> **echo -e "foo\nbar" > test.txt**

接下来，我们将启动两个运行在*独立*模式的连接器，这意味着他们在一个单一的，局部的，专用的进程中运行。我们提供三个配置文件作为参数。第一始终是Kafka连接过程中的公共配置，如要连接到的Kafka的代理服务器的配置和数据的序列化格式的配置。剩余的每个配置文件用来创建指定的连接器。这些文件包括一个唯一的连接器名称，需要实例化的连接器类，还有创建该连接器所需的其他配置。

> **bin/connect-standalone.sh config/connect-standalone.properties config/connect-file-source.properties config/connect-file-sink.properties**

用这些Kafka的示例配置文件，使用前面已经启动的本地群集的默认配置，建立两个连接器：第一是一个源连接器，其从输入文件中读取每行的内容，发布到的Kafka主题和第二个是一个sink连接器负责从Kafka主题读取消息，生产出的消息按行输出到文件。

在启动过程中，你会看到一些日志信息，包括一些表明该连接器被实例化的信息。一旦Kafka Connect进程已经开始，源连接器应该开始从test.txt读取每行的消息，并将其生产发布到主题connect-test，而sink连接器应该从主题connect-test读取消息，并将其写入文件test.sink.txt。我们可以通过检查输出文件的内容来验证数据都已通过整个管道输送：

> **cat test.sink.txt**

foo

bar

请注意，数据被存储在Kafka主题的connect-test中，所以我们也可以运行控制台消费者消费主题中的数据（或使用定制的消费者代码来处理它）：

> **bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic connect-test --from-beginning**

{"schema":{"type":"string","optional":false},"payload":"foo"}

{"schema":{"type":"string","optional":false},"payload":"bar"}

...

连接器不停的处理数据，因此我们可以将数据添加到该文件，并能看到数据通过管道移动：

> **echo "Another line" >> test.txt**

您应该看到一行消息出现在控制台消费者的控制台和sink文件中。

[**第8步：使用Kafka Streams处理数据**](http://kafka.apache.org/documentation/#quickstart_kafkastreams)

Kafka Streams 是Kafka的客户端库， 用来做实时流处理和分析存储在Kafka代理服务器的数据。该快速入门例子将演示如何运行这个流应用库。这里是要点[WordCountDemo](https://github.com/apache/kafka/blob/0.10.2/streams/examples/src/main/java/org/apache/kafka/streams/examples/wordcount/WordCountDemo.java)的示例代码（转换为方便阅读的Java 8 lambda表达式）。

// Serializers/deserializers (serde) for String and Long types

final Serde<String> stringSerde = Serdes.String();

final Serde<Long> longSerde = Serdes.Long();

// Construct a `KStream` from the input topic ""streams-file-input", where message values

// represent lines of text (for the sake of this example, we ignore whatever may be stored

// in the message keys).

KStream<String, String> textLines = builder.stream(stringSerde, stringSerde, "streams-file-input");

KTable<String, Long> wordCounts = textLines

// Split each text line, by whitespace, into words.

.flatMapValues(value -> Arrays.asList(value.toLowerCase().split("\\W+")))

// Group the text words as message keys

.groupBy((key, value) -> value)

// Count the occurrences of each word (message key).

.count("Counts")

// Store the running counts as a changelog stream to the output topic.

wordCounts.to(stringSerde, longSerde, "streams-wordcount-output");

它实现了单词计数算法，计算输入文本中一个单词的出现次数。然而，与其他单词计数的算法不同，其他的算法一般都是对有界数据进行操作，该算法演示应用程序的表现略有不同，因为他可以被设计去操作**无限的，无界的流**数据。和操作有界数据的算法相似，它是一个有状态的算法，可以跟踪和更新单词的计数。然而，因为它必须承担潜在的无界输入数据的处理，它会周期性地输出其当前状态和结果，同时继续处理更多的数据，因为它无法知道他有没有处理完“所有”的输入数据。

作为第一步骤，我们将准备好输入到Kafka主题的数据，随后由Kafka Streams应用程序进行处理。

> **echo -e "all streams lead to kafka\nhello kafka streams\njoin kafka summit" > file-input.txt**

或在Windows上：

> **echo all streams lead to kafka> file-input.txt**

> **echo hello kafka streams>> file-input.txt**

> **echo|set /p=join kafka summit>> file-input.txt**

接下来，我们使用控制台生产者把输入的数据发送到主题名**streams-file-input** 的主题上，其内容从STDIN一行一行的读取，并一行一行的发布到主题，每一行的消息都有一个空键和编码后的字符串（在实践中，当应用程序将启动并运行后，流数据很可能会持续流入Kafka）：

> **bin/kafka-topics.sh --create \**

**--zookeeper localhost:2181 \**

**--replication-factor 1 \**

**--partitions 1 \**

**--topic streams-file-input**

> **bin/kafka-console-producer.sh --broker-list localhost:9092 --topic streams-file-input < file-input.txt**

现在，我们可以运行单词计数应用程序来处理输入数据：

> **bin/kafka-run-class.sh org.apache.kafka.streams.examples.wordcount.WordCountDemo**

演示应用程序将从输入主题**streams-file-input**读取数据，对读取的消息的执行单词计数算法，并且持续写入其当前结果到输出主题**streams-wordcount-output**。因此，除了写回Kafka的日志条目，不会有任何的STDOUT输出。该演示将运行几秒钟，与典型的流处理应用不同，演示程序会自动终止。

现在，我们通过读取输出主题的输出得到单词计数演示程序的结果：

> **bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 \**

**--topic streams-wordcount-output \**

**--from-beginning \**

**--formatter kafka.tools.DefaultMessageFormatter \**

**--property print.key=true \**

**--property print.value=true \**

**--property key.deserializer=org.apache.kafka.common.serialization.StringDeserializer \**

**--property value.deserializer=org.apache.kafka.common.serialization.LongDeserializer**

下面的数据会被输出到控制台：

all 1

lead 1

to 1

hello 1

streams 2

join 1

kafka 3

summit 1

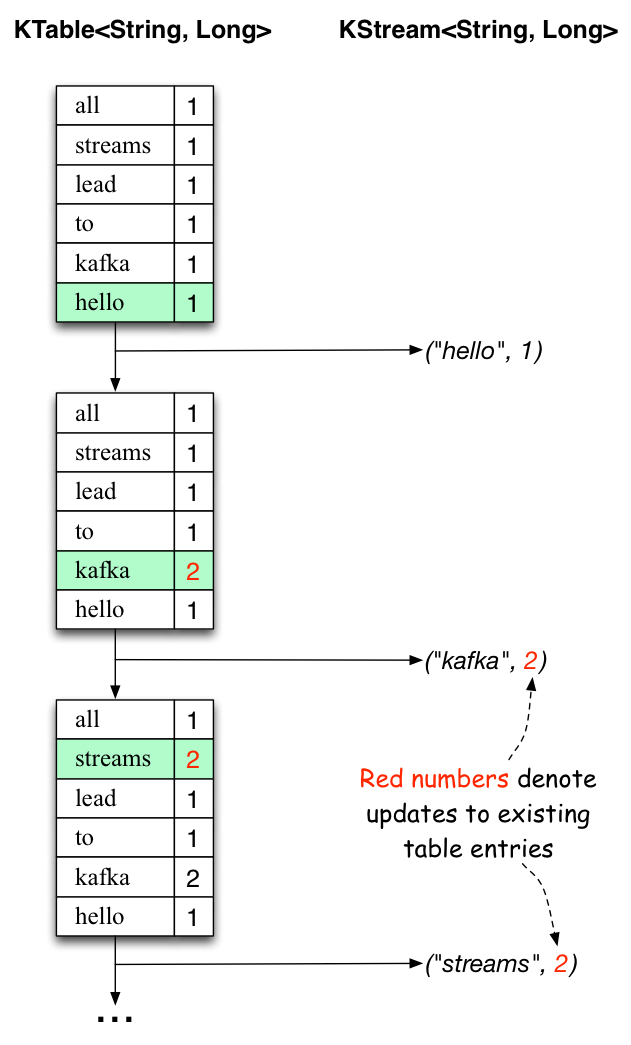
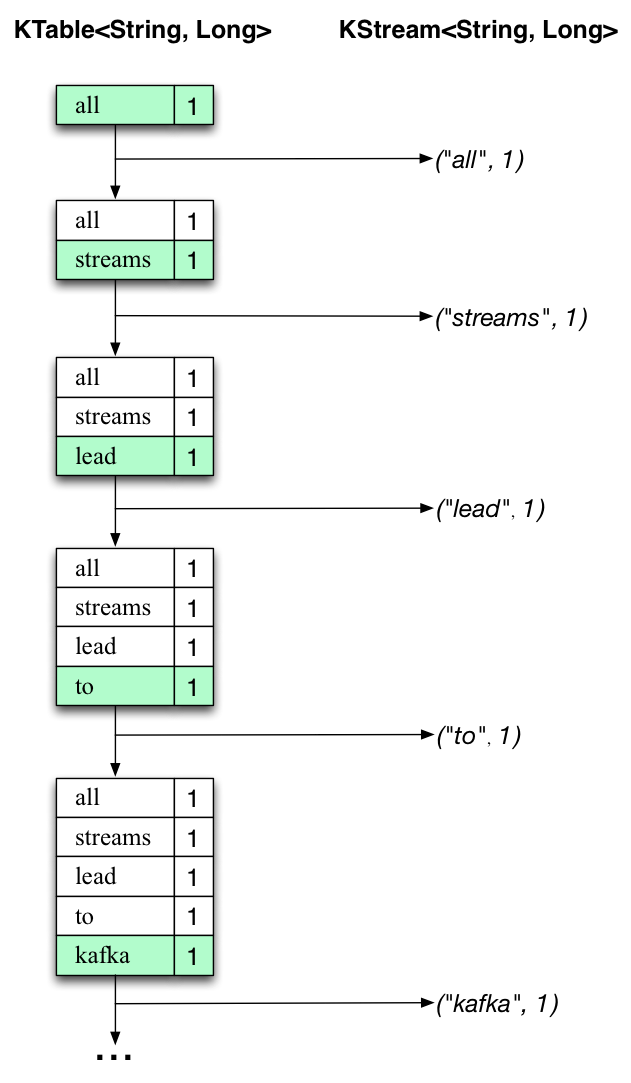
这里，第一列是java.lang.String类型的消息健，而第二列是java.lang.Long型消息值。注意，这里的输出其实是数据更新的连续流，每个数据记录（上面的例子里的每行的输出）都有一个单词更新后的数目值，例如“Kafka”作为键的记录。对于具有相同键的多个记录，每个后面的记录都是前一个记录的更新。

下面的两个图说明什么发生在幕后的过程。第一列显示的当前状态的变化，用KTable<String, Long>来统计单词出现的数目。第二列显示KTable状态更新导致的发生变化的记录，这个变化的记录被发送到输出Kafka主题**streams-wordcount-output**。

首先， “all streams lead to kafka”这样一行文本正在被处理。当新的单词被处理的时候，KTable会增加一个新的表项（以绿色背景高亮显示），并有相应的变化记录发送到下游KStream。

当第二行“hello kafka streams”被处理的时候，我们观察到，现有的KTable中的表项第一次被更新（这里: 单词 “kafka” 和 “streams”）。再次，改变的记录被发送到输出话题。

以此类推（我们跳过的第三行是如何被处理的插图）。这就解释了为什么输出主题有我们上面例子显示的内容，因为它包含了完整的更改记录。



跳出这个具体的例子我们从整体去看， Kafka流利用表和日志变化(changelog)流之间的二元性（here: 表= the KTable, 日志变化流 = the downstream KStream）：你可以发布的每一个表的变化去一个流，如果你从开始到结束消费了整个的日志变化(changelog)流，你可以重建表的内容。

现在，你可以写更多的输入信息到**streams-file-input**主题，并观察更多的信息加入到了 **streams-wordcount-output**主题，反映了更新后的单词数目（例如，使用上述的控制台生产者和控制台消费者）。

您可以通过**Ctrl-C**停止控制台消费者。

[**1.4生态系统**](http://kafka.apache.org/documentation/#ecosystem)

除了Kafka的主要版本之外，还有很多应用集成了Kafka工具。该[生态系统页面](https://cwiki.apache.org/confluence/display/KAFKA/Ecosystem)中列出的许多工具，包括流处理系统，Hadoop的集成，监控和部署工具。

[**1.5从以前版本升级**](http://kafka.apache.org/documentation/#upgrade)

[**从0.8.4，0.9.x，0.10.0.x或0.10.1.x升级到0.10.2.0**](http://kafka.apache.org/documentation/#upgrade_10_2_0)

0.10.2.0的有线协议有变化。通过下面的推荐滚动升级计划，你能保证在升级过程中无需停机。但是，请在升级之前查看[0.10.2.0版本显著的变化](http://kafka.apache.org/documentation/" \l "upgrade_1020_notable)。

从0.10.2版本开始，Java客户端（生产者和消费者）已获得与旧版本代理服务器沟通的能力。版本0.10.2客户可以跟0.10.0版或更新版本的代理沟通。但是，如果你的代理比0.10.0老，你必须在升级客户端之前升级Kafka集群中的所有代理服务器(Broker)。版本0.10.2代理支持0.8.x和更新的客户端。

**对于滚动升级：**

1. 更新所有代理服务器上的server.properties文件，添加以下属性：
   * inter.broker.protocol.version = CURRENT\_KAFKA\_VERSION（例如0.8.2，0.9.0，0.10.0或0.10.1）。
   * log.message.format.version = CURRENT\_KAFKA\_VERSION（参见[升级后的潜在性能的影响](http://kafka.apache.org/documentation/" \l "upgrade_10_performance_impact)了解此配置做什么的详细信息。）
2. 逐一升级代理：关闭代理，更新代码，并重新启动。
3. 一旦整个群集升级成功，通过编辑inter.broker.protocol.version将其设置为0.10.2的协议版本。
4. 如果您以前的消息格式为0.10.0，改变log.message.format.version至0.10.2（这是一个无效操作，因为0.10.0，0.10.1和0.10.2的消息格式相同）。如果您以前的消息格式版本低于0.10.0，不要改变log.message.format.version – 这个参数只能在所有的消费者都已经升级到0.10.0.0或更高版本之后改动。
5. 逐一重新启动代理服务器使新协议版本生效。
6. 如果这时log.message.format.version仍比0.10.0低，等到所有的消费者都已经升级到0.10.0或更高版本，然后更改每个代理服务器的log.message.format.version到0.10.2，然后逐一重新启动。

**注意：**如果你愿意接受宕机，你可以简单地把所有的代理服务器关闭，更新代码，然后重新启动他们。他们将默认使用新的协议。

**注：改变**协议版本并重新启动可以在代理服务器升级之后的任何时间做，没有必要必须立刻就做。

[**升级0.10.1版本的Kafka流应用**](http://kafka.apache.org/documentation/#upgrade_1020_streams)

* 从0.10.1升级您的流应用程序到0.10.2不需要升级代理。0.10.2 Kafka流应用程序可以连接到0.10.2和0.10.1代理（但无法连接到 0.10.0的代理）。
* 你需要重新编译代码。只是替换Kafka流的jar文件将无法正常工作，这破坏你的应用程序。
* 如果您使用自定义（即用户实现的）的时间戳提取，则需要更新此代码，因为TimestampExtractor接口改变了。
* 如果您注册了自定义指标，您将需要更新此代码，因为StreamsMetric接口被改变了。
* [0.10.2 流 API的变化](http://kafka.apache.org/0102/documentation/streams#streams_api_changes_0102)有更多的细节。

[**0.10.2.1显著的变化**](http://kafka.apache.org/documentation/#upgrade_10201_notable)

* 对于StreamsConfig类的两个配置的默认值的修改提高了Kafka流应用的弹性。内部Kafka流生产者retries默认值从0变化到10，内部Kafka流消费者max.poll.interval.ms 缺省值从300000到改变Integer.MAX\_VALUE。

[**0.10.2.0显著的变化**](http://kafka.apache.org/documentation/#upgrade_1020_notable)

* 在Java客户端（生产者和消费者）已获得与旧版本代理沟通的能力。版本0.10.2客户端可以跟0.10.0版或更新版本的代理沟通。请注意，某些功能在跟就代理沟通的时候不可用或被限制了。
* 在Java消费者中有几种方法现在可能抛出InterruptException如果调用线程被中断。请参阅KafkaConsumer的Javadoc，对这种变化有一个更深入的解释。
* Java的消费者现在被恰当关闭。默认情况下，消费者会等待30秒才能完成挂起的请求。一个带有timeout参数的新的API已添加到KafkaConsumer去控制最大等待时间。
* 用逗号分隔的多个正则表达式可以传递多个Java消费者给MirrorMaker–whitelist选择。这使得与MirrorMaker使用老Scala消费者时的行为一致。
* 从0.10.1升级您的流应用程序0.10.2不需要代理服务器升级。Kafka 0.10.2流应用程序可以连接到0.10.2和0.10.1代理（但无法连接到0.10.0代理）。
* Zookeeper的依赖从流API中删除。流API现在使用Kafka协议来管理内部主题，而不是直接修改动物园管理员的主题。这消除了需要直接访问Zookeeper的特权，而“StreamsConfig.ZOOKEEPER\_CONFIG”也不需要在流应用被设置。如果Kafka集群是安全认证的，流应用程序必须具备必要的安全权限才可以创建新的主题。
* 一些新的参数，包括“security.protocol”， “connections.max.idle.ms”， “retry.backoff.ms”， “reconnect.backoff.ms”和“request.timeout.ms”添加到StreamsConfig类。如果用户需要设置这些，要注意这些默认值。欲了解更多详情，请参阅[3.5Kafka流CONFIGS](http://kafka.apache.org/0102/documentation/" \l "streamsconfigs)。
* 该offsets.topic.replication.factor代理的配置现在在主题生产中强制使用。直到集群的大小符合这个复制因子要求，否则，主题的生产将失败，返回GROUP\_COORDINATOR\_NOT\_AVAILABLE错误。

[**新的协议版本**](http://kafka.apache.org/documentation/#upgrade_1020_new_protocols)

* [KIP-88](https://cwiki.apache.org/confluence/display/KAFKA/KIP-88%3A+OffsetFetch+Protocol+Update)：OffsetFetchRequest v2支持偏移检索所有的主题，如果topics数组设置为null。
* [KIP-88](https://cwiki.apache.org/confluence/display/KAFKA/KIP-88%3A+OffsetFetch+Protocol+Update)：OffsetFetchResponse V2引入了顶级error\_code域。
* [KIP-103](https://cwiki.apache.org/confluence/display/KAFKA/KIP-103%3A+Separation+of+Internal+and+External+traffic)：UpdateMetadataRequest v3引入一个listener\_name字段到end\_points数组中的元素。
* [KIP-108](https://cwiki.apache.org/confluence/display/KAFKA/KIP-108%3A+Create+Topic+Policy)：CreateTopicsRequest V1引入了一个validate\_only参数。
* [KIP-108](https://cwiki.apache.org/confluence/display/KAFKA/KIP-108%3A+Create+Topic+Policy)：CreateTopicsResponse V1引入了error\_message到数组topic\_errors的元素。

[**从0.8.4，0.9.x版本或0.10.0.X升级到0.10.1.0**](http://kafka.apache.org/documentation/#upgrade_10_1)

0.10.1.0有线协议发生了变化。通过下面的推荐滚动升级计划，能保证在升级过程中无需停机。但是，请注意在升级之前仔细阅读[0.10.1.0潜在的重大更改](http://kafka.apache.org/documentation/" \l "upgrade_10_1_breaking)。  
注意：由于新协议的引入，它是升级你的客户端之前请先完成Kafka集群的升级（即0.10.1.x客户端仅支持0.10.1.x或更高版本的代理，但0.10.1.x的代理可以支持旧版本客户端）。

**对于滚动升级：**

1. 更新所有代理上的server.properties文件，并添加以下属性：
   * inter.broker.protocol.version = CURRENT\_KAFKA\_VERSION（例如0.8.2.0，0.9.0.0或0.10.0.0）。
   * log.message.format.version = CURRENT\_KAFKA\_VERSION（参见[升级后的潜在性能的影响](http://kafka.apache.org/documentation/" \l "upgrade_10_performance_impact)对于此配置做什么的详细信息。）
2. 升级代理服务器一次一个：关闭代理，更新代码，并重新启动。
3. 一旦整个群集升级完成，通过编辑inter.broker.protocol.version并将其设置为0.10.1.0的协议版本。
4. 如果您以前的消息格式为0.10.0，改变log.message.format.version至0.10.1（这是一个无效操作，如果0.10.0和0.10.1两个协议的消息格式相同）。如果您以前的消息格式版本低于0.10.0，不要改变log.message.format.version — 这个参数只能在所有的消费者都已经升级到0.10.0.0或更高版本之后修改。
5. 逐一重新启动代理，新版本协议生效。
6. 如果log.message.format.version仍比0.10.0低，等到所有的消费者都已经升级到0.10.0或更高版本，然后更改log.message.format.version到0.10.1，逐一重新启动代理服务器。

**注意：**如果你愿意接受宕机，你可以简单地把所有的代理服务器关闭，更新代码，然后重新启动他们。他们将默认使用新的协议。

**注：改变**协议版本并重新启动可以在代理服务器升级之后的任何时间做，没有必要必须立刻就做。

[**0.10.1.0的重大更改**](http://kafka.apache.org/documentation/#upgrade_10_1_breaking)

* 日志保留时间不再基于日志段的最后修改时间。相反，它会基于日志段里拥有最大的时间戳的消息。
* 日志滚动时间不再取决于日志段创建时间。相反，它现在是基于消息的时间戳。进一步来说，如果日志段中第一个消息的时间戳是T，当一个新的消息具有的时间戳大于或等于T + log.roll.m，该日志将被覆盖。
* 0.10.0的打开文件的处理程序将增加〜33％，因为每个日志段增加的时间索引文件。
* 时间索引和偏移索引共享相同的索引大小的配置。因为时间索引条目大小是1.5倍偏移索引条目的大小。用户可能需要增加log.index.size.max.bytes以避免潜在的频繁的日志滚动。
* 由于增加的索引文件，在某些代理服务器上具有大量的日志段（例如> 15K），代理启动期间日志加载过程可能很长。根据我们的实验，num.recovery.threads.per.data.dir设置为1可减少日志装载时间。

[**升级0.10.0Kafka流应用**](http://kafka.apache.org/documentation/#upgrade_1010_streams)

* 从0.10.0升级您的流应用程序到0.10.1确实需要一个[代理的升级](http://kafka.apache.org/documentation/" \l "upgrade_10_1)，因为Kafka 0.10.1的流应用程序只能连接到0.10.1代理。
* 有几个API的变化不向后兼容（参见[流API在0.10.1的变化](http://kafka.apache.org/0102/documentation/streams" \l "streams_api_changes_0101)有详细介绍）。因此，你需要更新和重新编译代码。只是交换了Kafka流库的jar文件将无法正常工作，并会破坏你的应用程序。

[**0.10.1.0显著的变化**](http://kafka.apache.org/documentation/#upgrade_1010_notable)

* 新的Java消费者不是beta版了，我们推荐它做新的应用开发。老Scala消费者仍然支持，但他们会在未来的版本中将会弃用，并将在未来的主版本中删除。
* 在使用像MirrorMaker和控制台消费者新建消费者的过程中–new-consumer/ –new.consumer开关不再被需要; 一个简单地使用是通过一个Kafka代理去连接，而不是Zookeeper的合集。此外，控制台消费者去连接旧版本的消费者已被弃用，并将在未来的主版本中删除。
* Kafka集群现在可以通过一个集群ID被唯一标识。其会在一个代理升级到0.10.1.0时自动生成。集群ID经由kafka.server可用：type= KafkaServer，name= ClusterId metric ，它是所述元数据响应的一部分。串行器，客户端拦截器和度量报告可以通过实现ClusterResourceListener接口接收集群ID。
* BrokerState “RunningAsController”（值4）已被删除。由于一个bug，代理在转换状态之前只会简单的这种状态下，因此去除的影响应该很小。一种推荐的检测方法是一个给定的代理的控制器是由kafka.controller实现：type=KafkaController,name=ActiveControllerCount metric。
* 新的Java消费者现在可以允许用户通过时间戳在分区上搜索偏移量(offset)。
* 新的Java消费者现在可以从后台线程支持心跳检查。有一个新的配置 max.poll.interval.ms，它控制消费者会主动离开组（5分钟默认情况下）之前轮询调用的最大时间。配置的值 request.timeout.ms必须始终大于max.poll.interval.ms因为这是一个JoinGroup请求可以在服务器上被阻止到消费者被负载均衡之前的最长时间.所以我们可以改变默认值为刚好超过5分钟。最后，默认值session.timeout.ms已调整到10秒，默认值max.poll.records已更改为500。
* 当授权者和用户没有**说明**某个主题的授权，代理将不再返回TOPIC\_AUTHORIZATION\_FAILED给请求，因为这会泄漏主题名称。相反，UNKNOWN\_TOPIC\_OR\_PARTITION错误代码将被返回。使用Kafka生产者和消费者通常会在收到未知的主题错误时自动重试，这可能会导致意外的超时或延迟。如果你怀疑这种情况发生了，你可以查看客户端的log去检查。
* 获取返回有默认的大小限制（消费者50 MB和副本的复制10 MB）。现有的每个分区的限制也适用（消费者和副本复制为1 MB）。请注意，这些限制都不是绝对最大值，在下一个要点有解释。
* 消费者和副本可以继续进行，如果发现一个消息大于返回/分区大小的限制。更具体地，如果在非空的分区上提取的第一个消息比任一个或两个限值大，仍然会被返回。
* 重载的构造函数加入到kafka.api.FetchRequest和kafka.javaapi.FetchRequest允许调用者指定分区顺序（因为顺序在V3是很重要的）。先前存在的构造函数被弃用，在发送请求以避免饥饿问题之前，分区会被洗牌。

[**新的协议版本**](http://kafka.apache.org/documentation/#upgrade_1010_new_protocols)

* ListOffsetRequest V1支持精确的基于时间戳的偏移搜索。
* MetadataResponse V2引入了一个新的参数： “CLUSTER\_ID”。
* FetchRequest v3支持限制请求返回的大小（除了现有的每个分区的限制），它能够返回比限制更大的消息和在请求中加入分区的顺序具有重要意义。
* JoinGroup V1引入了一个新的字段： “rebalance\_timeout”。

[**升级0.8.4或0.9.x版本到0.10.0.0**](http://kafka.apache.org/documentation/#upgrade_10)

0.10.0.0具有的[潜在的重大更改](http://kafka.apache.org/documentation/" \l "upgrade_10_breaking)（请在升级前仔细检查更改）和[在升级后的性能影响](http://kafka.apache.org/documentation/" \l "upgrade_10_performance_impact)。通过下面的推荐滚动升级计划，能保证不宕机，不影响性能和随后的升级。  
注意：由于新协议的引入，升级客户端之前升级您的Kafka集群是很重要的。

**注意0.9.0.0版本的客户端：**由于0.9.0.0引入了一个错误，即依赖于ZooKeeper的客户（老Scala高层次消费者和与老消费者一起使用的MirrorMaker）不能和0.10.0.x代理一起工作。因此，**在**代理都升级到0.10.0.x之前, 0.9.0.0客户端应升级到0.9.0.1 . 这一步对0.8.4或0.9.0.1客户端没有必要。

**对于滚动升级：**

1. 更新所有代理服务器的server.properties文件，并添加以下属性：
   * inter.broker.protocol.version = CURRENT\_KAFKA\_VERSION（例如0.8.2或0.9.0.0）。
   * log.message.format.version = CURRENT\_KAFKA\_VERSION（参见[升级后的潜在性能的影响](http://kafka.apache.org/documentation/" \l "upgrade_10_performance_impact)对于此配置做什么的详细信息。）
2. 升级代理。这可以通过简单地将其关机，更新代码，并重新启动实现。
3. 一旦整个群集升级结束，通过编辑inter.broker.protocol.version并将其设置为0.10.0.0的协议版本。注意：您不应该修改log.message.format.version — 这个参数只能在所有的消费者都已经升级到0.10.0.0之后再修改。
4. 逐一重新启动代理，新协议版本生效。
5. 一旦所有的消费者都已经升级到0.10.0，逐一修改log.message.format.version至0.10.0和重启代理服务器。

**注意：**如果你愿意接受宕机，你可以简单地把所有的代理服务器关闭，更新代码，然后重新启动他们。他们将默认使用新的协议。

**注：改变**协议版本并重新启动可以在代理服务器升级之后的任何时间做，没有必要必须立刻就做。

[**升级到0.10.0.0带来的潜在的性能影响**](http://kafka.apache.org/documentation/#upgrade_10_performance_impact)

0.10.0消息格式包括一个新的时间戳字段，并对压缩的消息使用相对偏移。磁盘上的消息格式可以通过在server.properties文件的log.message.format.version进行配置。默认的磁盘上的消息格式为0.10.0。如果消费者客户端的版本是0.10.0.0之前的版本，那它只能明白0.10.0之前的消息格式。在这种情况下，代理能够把消息从0.10.0格式转换到一个较早的格式再发送旧版本的响应给消费者。然而，代理不能在这种情况下使用零拷贝转移。Kafka社区报告显示性能的影响为CPU利用率从20％增加至将近100％，这迫使所有客户端的必须即时升级使性能恢复正常。为了避免这样的消息转换带来的性能问题，消费者升级到0.10.0.0之前，在升级代理到0.10.0.0的过程中设置log.message.format.version到0.8.2或0.9.0。这样一来，代理仍然可以使用零拷贝传输，将数据发送到老消费者。一旦消费者升级完成，消息格式更改为0.10.0，这样代理就可以享受新的消息格式包括新的时间戳和改进的压缩算法。这种转换可以支持兼容性，对只有几个还没有更新到最新客户端的应用程序非常有用，但不切实际的是使用一个过度使用的集群中去支持所有消费者的流量。因此，当代理已经升级，但大多数客户端还没有完成升级的情况，要尽可能避免使用这种信息转换。

对于升级到0.10.0.0客户，没有性能影响。

**注：**设置消息格式版本是一个证明，现有的所有支持的消息都在这个版本或低于该消息格式的版本。否则， 0.10.0.0之前的消费者可能不能正常工作。特别是消息格式设置为0.10.0之后，不应该再改回先前的格式，因为它可能使得0.10.0.0之前的消费者工作异常。

**注：**由于每个消息中引入了另外的时间戳，生产者发送的消息大小比较小的时候因为额外的负载开销也许会看到吞吐量的下降。同样，副本的复制会让每个消息额外传输8个字节。如果你正在运行接近集群承载能力的网络容量，你可能会压垮网卡，由于超载而发生故障和性能问题。

**注：**如果您已对生产者启用压缩算法，您可能会注意到降低的生产者吞吐量和/或在某些情况下代理降低的压缩比。当接收到压缩的消息，0.10.0代理避免再次压缩消息，其通常降低了等待时间，并提高了吞吐量。在某些情况下，这可能会减少生产者批量消息包的大小，这可能导致更糟糕的吞吐量。如果发生这种情况，用户可以调整生产者的linger.ms和batch.size以获得更好的吞吐量。此外，用于高效压缩消息的生产者缓冲区比代理使用的缓冲区小，这可能对磁盘的压缩消息比率有负面的影响。我们打算在未来的Kafka版本中能够配置这些参数。

[**0.10.0.0潜在的重大更改**](http://kafka.apache.org/documentation/#upgrade_10_breaking)

* 从Kafka0.10.0.0开始，Kafka消息格式的版本被表示为Kafka版本。例如，消息格式0.9.0指通过Kafka0.9.0支持的最高消息版本。
* 消息格式0.10.0已经推出，它是默认使用的版本。它引入了一个时间戳字段和相对偏移被用于压缩消息。
* ProduceRequest /Response V2已经被引入，它在默认情况下支持消息格式0.10.0
* FetchRequest /Response V2已经被引入，它在默认情况下支持消息格式0.10.0
* MessageFormatter接口从def writeTo(key: Array[Byte], value: Array[Byte], output: PrintStream)变为 def writeTo(consumerRecord: ConsumerRecord[Array[Byte], Array[Byte]], output: PrintStream)
* MessageReader接口从def readMessage(): KeyedMessage[Array[Byte], Array[Byte]]变为 def readMessage(): ProducerRecord[Array[Byte], Array[Byte]]
* MessageFormatter的包从kafka.tools改变为kafka.common
* MessageReader的包从kafka.tools改变我kafka.common
* MirrorMakerMessageHandler不再公开方法handle(record: MessageAndMetadata[Array[Byte], Array[Byte]])，因为它从来没有被调用。
* 0.7 KafkaMigrationTool不再被打包进Kafka包。如果您需要从0.7迁移到0.10.0，请先迁移到0.8，然后再按照文档的升级过程升级0.8到0.10.0。
* 新的消费者拥有标准化的API，接受java.util.Collection作为方法参数序列类型。现有的代码可能需要更新才能与0.10.0客户端库一起工作。
* LZ4-compressed的消息处理被改变为使用可互操作的帧规范（LZ4f V1.5.1）。为了保持与旧客户端的兼容性，这一变化仅适用于消息格式0.10.0及更高版本。使用V0 / V1（消息格式0.9.0）的客户端应该继续使用0.9.0帧规范实现执行产生/抓取LZ4压缩消息。使用生产/获取协议v2或更高版本客户端应该使用互操作LZ4f帧规范。可互操作的LZ4库的列表，请参考http://www.lz4.org/

[**0.10.0.0显著的变化**](http://kafka.apache.org/documentation/#upgrade_10_notable)

* 从Kafka0.10.0.0开始，新的客户端库**Kafka流**可用于流处理存储在Kafka主题的数据。这个新的客户端库只适用于0.10.x及后面版本的代理。欲了解更多信息，请阅读[流文件](http://kafka.apache.org/0102/documentation/streams)。
* 对新的消费者，配置参数receive.buffer.bytes的默认值现在是64k。
* 新的消费者现在公开暴露配置参数exclude.internal.topics去限制内部主题（诸如消费者偏移主题），不让这些主题被偶然的包括在正则表达式的主题订阅中。默认情况下，它处于启用状态。
* 老Scala生产者已被弃用。用户要尽快迁移他们的代码到Kafka客户端JAR里的Java生产者。
* 新的消费者API已经被标记为稳定。

[**升级0.8.0，0.8.1.X或0.8.2.X到0.9.0.0**](http://kafka.apache.org/documentation/#upgrade_9)

0.9.0.0具有的[潜在的重大更改](http://kafka.apache.org/documentation/" \l "upgrade_9_breaking)（请在升级前检查），还有以前的版本到现在的代理间协议的变化。这意味着升级的代理和客户端可能不兼容旧版本。您在升级您的客户端之前升级Kafka集群是很重要的。如果您正在使用MirrorMaker下游集群应该先升级为好。

**对于滚动升级：**

1. 更新所有代理上的server.properties文件，并添加以下属性：inter.broker.protocol.version = 0.8.2.X
2. 逐一升级的代理。可以通过简单地将其关闭，更新代码，并重新启动它实现。
3. 一旦整个群集升级成功，通过编辑inter.broker.protocol.version并将其设置为0.9.0.0的协议版本。
4. 逐一重新启动代理使新协议版本生效

**注意：**如果你愿意接受宕机，你可以简单地把所有的代理服务器关闭，更新代码，然后重新启动他们。他们将默认使用新的协议。

**注：改变**协议版本并重新启动可以在代理服务器升级之后的任何时间做，没有必要必须立刻就做。

[**0.9.0.0潜在的重大更改**](http://kafka.apache.org/documentation/#upgrade_9_breaking)

* Java 1.6不再支持。
* Scala 2.9不再支持。
* 1000以上的代理ID现在默认保留，用来做自动分配的代理ID。如果您的集群已存在高于阈值的经纪人的ID确保相应地增加reserved.broker.max.id代理配置属性。
* 配置参数replica.lag.max.messages被删除。分区Leader将不再考虑滞后的消息数量来决定哪些副本是同步的，。
* 配置参数replica.lag.time.max.ms现在不仅指从副本提取请求所花费的时间，也标识副本最后一次同步到现在经过的时间。那些副本仍然从领导者获取信息，但在replica.lag.time.max.ms时间内没有从leader最新消息的副本将被认为是不同步的。
* 压缩主题不再接受没有主键消息和遇到这种情况生产者会抛出一个异常。在0.8.4，没有主键的消息会导致日志压缩线程退出（并停止所有压缩主题的处理）。
* MirrorMaker不再支持多种目标集群。因此，它只能接受一个–consumer.config参数。要镜像多个源集群，则需要每个源集群至少一个MirrorMaker实例，每个都有自己的消费者配置。
* 在包*org.apache.kafka.clients.tools.\*里的工具*已移至*org.apache.kafka.tools.\*包*。所有的其中的脚本将仍然像往常一样起作用，只是直接导入这些类的自定义代码将受到影响。
* 默认的KafkaJVM性能选项（KAFKA\_JVM\_PERFORMANCE\_OPTS）已经在kafka-run-class.sh被改变。
* 该kafka-topics.sh脚本（kafka.admin.TopicCommand）现在失败会返回非零退出代码。
* 该kafka-topics.sh脚本（kafka.admin.TopicCommand）现在碰到由于使用“.” 或“\_”的主题的名称将打印警告信息，以及在实际发生冲突的情况下打印错误信息。
* 该kafka-console-producer.sh脚本（kafka.tools.ConsoleProducer）默认将使用Java生产者而不是旧的Scala生产者，并且用户必须指定“老生产者”使用旧版本的生产者。
* 默认情况下，所有命令行工具将打印所有消息记录到stderr而不是stdout。

[**0.9.0.1的显着变化**](http://kafka.apache.org/documentation/#upgrade_901_notable)

* 新的代理ID自动生成功能可以通过设置broker.id.generation.enable为false禁用。
* 配置参数log.cleaner.enable现在默认为true。这意味主题在配置 cleanup.policy=compact下将缺省压缩，清洁器进程通过log.cleaner.dedupe.buffer.size缺省被分配128MB堆。您可以检查你的配置log.cleaner.dedupe.buffer.size，并根据您的压缩主题使用其他log.cleaner配置值。
* 对于新的消费者，配置参数fetch.min.bytes的默认值现在是1。

**0.9.0.0弃用的功能**

* 从kafka-topics.sh脚本（kafka.admin.TopicCommand）改变主题配置已被弃用。今后，请使用kafka-configs.sh脚本（kafka.admin.ConfigCommand）。
* 该kafka-consumer-offset-checker.sh（kafka.tools.ConsumerOffsetChecker）已被弃用。今后，请使用kafka-consumer-groups.sh（kafka.admin.ConsumerGroupCommand）。
* 该kafka.tools.ProducerPerformance类已弃用。今后，请使用org.apache.kafka.tools.ProducerPerformance（kafka-producer-perf-test.sh也将改为使用新的类）。
* 生产者配置block.on.buffer.full已被弃用，并将在未来的版本中删除。目前，它的默认值已更改为false。该KafkaProducer将不再抛出BufferExhaustedException而是将使用max.block.ms值来阻止，之后它会抛出一个TimeoutException。如果block.on.buffer.full属性显式的设置为true，它将设置max.block.ms到Long.MAX\_VALUE，而metadata.fetch.timeout.ms将不被认可。

[**0.8.1 升级到0.8.2**](http://kafka.apache.org/documentation/#upgrade_82)

0.8.2与0.8.1完全兼容。可以通过简单地将其关闭，更新代码，并重新启动逐一升级代理。

[**0.8.0升级到 0.8.1**](http://kafka.apache.org/documentation/#upgrade_81)

0.8.1与0.8完全兼容。可以通过简单地将其关闭，更新代码，并重新启动逐一升级代理。[**从0.7升级**](http://kafka.apache.org/documentation/#upgrade_7)

0.7版本与新版本不兼容。API，Zookeeper的数据结构和协议，可以配置的增加副本（这是在0.7没有的），都发生了重大变化。从0.7到更高版本的升级需要[特殊的工具](https://cwiki.apache.org/confluence/display/KAFKA/Migrating+from+0.7+to+0.8)进行迁移。这种迁移可以无需宕机就可以完成。

注：在此文章中这两个单词被翻译为右边的对应词语。

Records – 记录(持久化的消息)

Broker –代理服务器