文档

Kafka 1.0 文档

Prior releases: 0.7.x, 0.8.0, 0.8.1.X, 0.8.2.X, 0.9.0.X, 0.10.0.X, 0.10.1.X, 0.10.2.X, 0.11.0.X.

1. 入门

1.1 介绍

Apache Kafka® 是 *一个分布式流处理平台*. 这到底意味着什么呢?

我们知道流处理平台有以下三种特性:

1. 可以让你发布和订阅流式的记录。这一方面与消息队列或者企业消息系统类似。
2. 可以储存流式的记录，并且有较好的容错性。

3. 可以在流式记录产生时就进行处理。

Kafka适合什么样的场景?

它可以用于两大类别的应用:

1. 构造实时流数据管道，它可以在系统或应用之间可靠地获取数据。 (相当于message queue)
2. 构建实时流式应用程序，对这些流数据进行转换或者影响。 (就是流处理，通过kafka stream topic和topic之间内部进行变化)

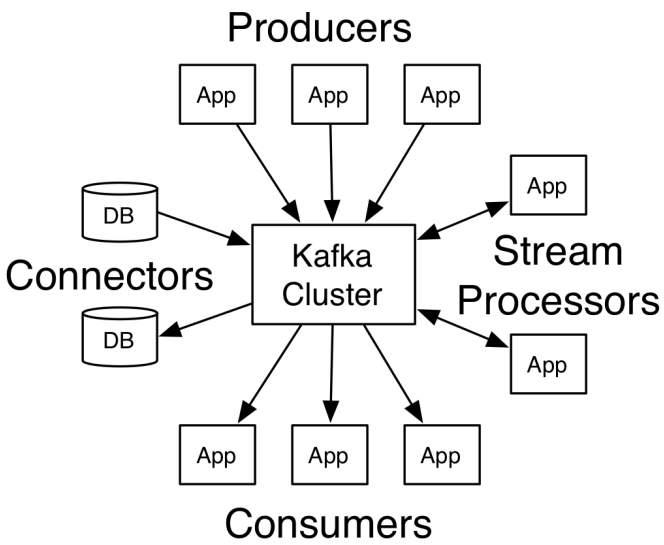
为了理解Kafka是如何做到以上所说的功能，从下面开始，我们将深入探索Kafka的特性。.

首先是一些概念:

- Kafka作为一个集群，运行在一台或者多台服务器上.
- Kafka 通过 *topic* 对存储的流数据进行分类。
- 每条记录中包含一个key，一个value和一个timestamp（时间戳）。

Kafka有四个核心的API:

- The Producer API 允许一个应用程序发布一串流式的数据到一个或者多个Kafka topic。
- The Consumer API 允许一个应用程序订阅一个或多个 topic ，并且对发布给他们的流式数据进行处理。
- The Streams API 允许一个应用程序作为一个*流处理器*，消费一个或者多个topic产生的输入流，然后生产一个输出流到一个或多个topic中去，在输入输出流中进行有效的转换。
- The Connector API 允许构建并运行可重用的生产者或者消费者，将Kafka topics连接到已存在的应用程序或者数据系统。比如，连接到一个关系型数据库，捕捉表（table）的所有变更内容。



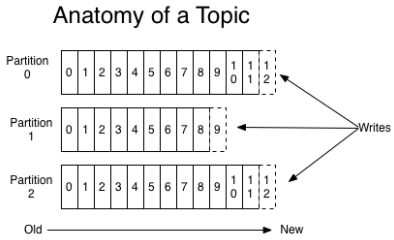
在Kafka中，客户端和服务器使用一个简单、高性能、支持多语言的 TCP 协议.此协议版本化并且向下兼容老版本， 我们为Kafka提供了Java客户端，也支持许多其他语言的客户端。

Topics和日志

让我们首先深入了解下Kafka的核心概念:提供一串流式的记录– topic 。

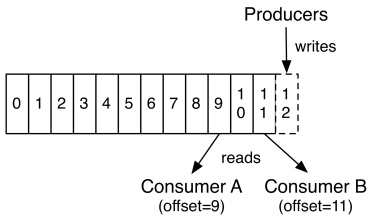
Topic 就是数据主题，是数据记录发布的地方,可以用来区分业务系统。Kafka中的Topics总是多订阅者模式，一个topic可以拥有一个或者多个消费者来订阅它的数据。

对于每一个topic， Kafka集群都会维持一个分区日志，如下所示：



每个分区都是有序且顺序不可变的记录集，并且不断地追加到结构化的commit log文件。分区中的每一个记录都会分配一个id号来表示顺序，我们称之为offset，*offset*用来唯一的标识分区中每一条记录。

Kafka 集群保留所有发布的记录—无论他们是否已被消费—并通过一个可配置的参数——保留期限来控制. 举个例子， 如果保留策略设置为2天，一条记录发布后两天内，可以随时被消费，两天过后这条记录会被抛弃并释放磁盘空间。Kafka的性能和数据大小无关，所以长时间存储数据没有什么问题.



事实上，在每一个消费者中唯一保存的元数据是offset（偏移量）即消费在log中的位置.偏移量由消费者所控制:通常在读取记录后，消费者会以线性的方式增加偏移量，但是实际上，由于这个位置由消费者控制，所以消费者可以采用任何顺序来消费记录。例如，一个消费者可以重置到一个旧的偏移量，从而重新处理过去的数据；也可以跳过最近的记录，从"现在"开始消费。

这些细节说明Kafka 消费者是非常廉价的—消费者的增加和减少，对集群或者其他消费者没有多大的影响。比如，你可以使用命令行工具，对一些topic内容执行 tail操作，并不会影响已存在的消费者消费数据。

日志中的 partition（分区）有以下几个用途。第一，当日志大小超过了单台服务器的限制，允许日志进行扩展。每个单独的分区都必须受限于主机的文件限制，不过一个主题可能有多个分区，因此可以处理无限量的数据。第二，可以作为并行的单元集—关于这一点，更多细节如下

分布式

日志的分区partition （分布）在Kafka集群的服务器上。每个服务器在处理数据和请求时，共享这些分区。每一个分区都会在已配置的服务器上进行备份，确保容错性.

每个分区都有一台 server 作为 “leader”，零台或者多台server作为 follwers 。leader server 处理一切对 partition （分区）的读写请求，而follwers只需被动的同步leader上的数据。当leader宕机了，followers 中的一台服务器会自动成为新的 leader。每台 server 都会成为某些分区的 leader 和某些分区的 follower，因此集群的负载是平衡的。

生产者

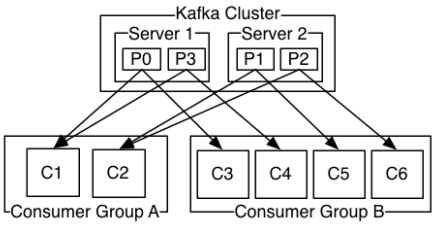
生产者可以将数据发布到所选择的topic（主题）中。生产者负责将记录分配到topic的哪一个 partition（分区）中。可以使用循环的方式来简单地实现负载均衡，也可以根据某些语义分区函数(例如：记录中的key)来完成。下面会介绍更多关于分区的使用。

消费者

消费者使用一个 *消费组* 名称来进行标识，发布到topic中的每条记录被分配给订阅消费组中的一个消费者实例. 消费者实例可以分布在多个进程中或者多个机器上。

如果所有的消费者实例在同一消费组中，消息记录会负载平衡到每一个消费者实例.

如果所有的消费者实例在不同的消费组中，每条消息记录会广播到所有的消费者进程.



如图，这个 Kafka 集群有两台 server 的，四个分区(p0-p3)和两个消费者组。消费组A有两个消费者，消费组B有四个消费者。

通常情况下，每个 topic 都会有一些消费组，一个消费组对应一个"逻辑订阅者"。一个消费组由许多消费者实例组成，便于扩展和容错。这就是发布和订阅的概念，只不过订阅者是一组消费者而不是单个的进程。

在Kafka中实现消费的方式是将日志中的分区划分到每一个消费者实例上，以便在任何时间，每个实例都是分区唯一的消费者。维护消费组中的消费关系由Kafka协议动态处理。如果新的实例加入组，他们将从组中其他成员处接管一些 partition 分区;如果一个实例消失，拥有的分区将被分发到剩余的实例。

Kafka 只保证分区内的记录是有序的，而不保证主题中不同分区的顺序。每个 partition 分区按照key值排序足以满足大多数应用程序的需求。但如果你需要总记录在所有记录的上面，可使用仅有一个分区的主题来实现，这意味着每个消费者组只有一个消费者进程。

保证

high-level Kafka给予以下保证:

- 生产者发送到特定topic partition 的消息将按照发送的顺序处理。 也就是说，如果记录M1和记录M2由相同的生产者发送，并先发送M1记录，那么M1的偏移比M2小，并在日志中较早出现
- 一个消费者实例按照日志中的顺序查看记录.
- 对于具有N个副本的主题，我们最多容忍N-1个服务器故障，从而保证不会丢失任何提交到日志中的记录.

关于保证的更多细节可以看文档的设计部分。

Kafka作为消息系统

Kafka streams的概念与传统的企业消息系统相比如何?

传统的消息系统有两个模块: 队列 和 发布-订阅。 在队列中，消费者池从server读取数据，每条记录被池子中的一个消费者消费; 在发布订阅中，记录被广播到所有的消费者。两者均有优缺点。 队列的优点在于它允许你将处理数据的过程分给多个消费者实例，使你可以扩展处理过程。 不好的是，队列不是多订阅者模式的——旦一个进程读取了数据，数据就会被丢弃。 而发布-订阅系统允许你广播数据到多个进程，但是无法进行扩展处理，因为每条消息都会发送给所有的订阅者。

消费组在Kafka有两层概念。在队列中，消费组允许你将处理过程分发给一系列进程(消费组中的成员)。 在发布订阅中，Kafka允许你将消息广播给多个消费组。

Kafka的优势在于每个topic都有以下特性—可以扩展处理并且允许多订阅者模式—不需要只选择其中一个.

Kafka相比于传统消息队列还具有更严格的顺序保证

传统队列在服务器上保存有序的记录，如果多个消费者消费队列中的数据， 服务器将按照存储顺序输出记录。 虽然服务器按顺序输出记录，但是记录被异步传递给消费者， 因此记录可能会无序的到达不同的消费者。这意味着在并行消耗的情况下， 记录的顺序是丢失的。因此消息系统通常使用“唯一消费者”的概念，即只让一个进程从队列中消费， 但这就意味着不能够并行地处理数据。

Kafka 设计的更好。topic中的partition是一个并行的概念。 Kafka能够为一个消费者池提供顺序保证和负载平衡，是通过将topic中的partition分配给消费者组中的消费者来实现的， 以便每个分区由消费组中的一个消费者消耗。通过这样，我们能够确保消费者是该分区的唯一读者，并按顺序消费数据。 众多分区保证了多个消费者实例间的负载均衡。但请注意，消费者组中的消费者实例个数不能超过分区的数量。

Kafka 作为存储系统

许多消息队列可以发布消息，除了消费消息之外还可以充当中间数据的存储系统。那么Kafka作为一个优秀的存储系统有什么不同呢?

数据写入Kafka后被写到磁盘，并且进行备份以便容错。直到完全备份，Kafka才让生产者认为完成写入，即使写入失败Kafka也会确保继续写入

Kafka使用磁盘结构，具有很好的扩展性—50kb和50TB的数据在server上表现一致。

可以存储大量数据，并且可通过客户端控制它读取数据的位置，您可认为Kafka是一种高性能、低延迟、具备日志存储、备份和传播功能的分布式文件系统。

关于Kafka提交日志存储和备份设计的更多细节，可以阅读 这页 。

Kafka用做流处理

Kafka 流处理不仅仅用来读写和存储流式数据，它最终的目的是为了能够进行实时的流处理。

在Kafka中，流处理器不断地从输入的topic获取流数据，处理数据后，再不断生产流数据到输出的topic中去。

例如，零售应用程序可能会接收销售和出货的输入流，经过价格调整计算后，再输出一串流式数据。

简单的数据处理可以直接用生产者和消费者的API。对于复杂的数据变换，Kafka提供了Streams API。Stream API 允许应用做一些复杂的处理，比如将流数据聚合或者join。

这一功能有助于解决以下这种应用程序所面临的问题：处理无序数据，当消费端代码变更后重新处理输入，执行有状态计算等。

Streams API建立在Kafka的核心之上：它使用Producer和Consumer API作为输入，使用Kafka进行有状态的存储， 并在流处理器实例之间使用相同的消费组机制来实现容错。

批处理

将消息、存储和流处理结合起来，使得Kafka看上去不一般，但这是它作为流平台所备的.

像HDFS这样的分布式文件系统可以存储用于批处理的静态文件。 一个系统如果可以存储和处理历史数据是非常不错的。

传统的企业消息系统允许处理订阅后到达的数据。以这种方式来构建应用程序，并用它来处理即将到达的数据。

Kafka结合了上面所说的两种特性。作为一个流应用程序平台或者流数据管道，这两个特性，对于Kafka 来说是至关重要的。

通过组合存储和低延迟订阅，流式应用程序可以以同样的方式处理过去和未来的数据。 一个单一的应用程序可以处理历史记录的数据，并且可以持续不断地处理以后到达的数据，而不是在到达最后一条记录时结束进程。 这是一个广泛的流处理概念，其中包含批处理以及消息驱动应用程序

同样，作为流数据管道，能够订阅实时事件使得Kafk具有非常低的延迟; 同时Kafka还具有可靠存储数据的特性，可用来存储重要的支付数据， 或者与离线系统进行交互，系统可间歇性地加载数据，也可在停机维护后再次加载数据。流处理功能使得数据可以在到达时转换数据。

有关Kafka提供的保证、API和功能的更多信息，请看文档的剩余部分。

1.2 使用案例

以下描述了一些 ApacheKafka ®的流行用例。有关这些领域的概述，请参阅 此博客中的文章。

消息

Kafka 很好地替代了传统的message broker（消息代理）。 Message brokers 可用于各种场合（如将数据生成器与数据处理解耦，缓冲未处理的消息等）。 与大多数消息系统相比，Kafka拥有更好的吞吐量、内置分区、具有复制和容错的功能，这使它成为一个非常理想的大型消息处理应用。

根据我们的经验，通常消息传递使用较低的吞吐量，但可能要求较低的端到端延迟，Kafka提供强大的持久性来满足这一要求。

在这方面，Kafka 可以与传统的消息传递系统（ActiveMQ 和 RabbitMQ）相媲美。

跟踪网站活动

Kafka 的初始用例是将用户活动跟踪管道重建为一组实时发布-订阅源。 这意味着网站活动（浏览网页、搜索或其他的用户操作）将被发布到中心topic，其中每个活动类型有一个topic。 这些订阅源提供一系列用例，包括实时处理、实时监视、对加载到Hadoop或离线数据仓库系统的数据进行离线处理和报告等。

每个用户浏览网页时都生成了许多活动信息，因此活动跟踪的数据量通常非常大

度量

Kafka 通常用于监控数据。这涉及到从分布式应用程序中汇总数据，然后生成可操作的集中数据源。

日志聚合

许多人使用Kafka来替代日志聚合解决方案。 日志聚合系统通常从服务器收集物理日志文件，并将其置于一个中心系统（可能是文件服务器或HDFS）进行处理。 Kafka 从这些日志文件中提取信息，并将其抽象为一个更加清晰的消息流。 这样可以实现更低的延迟处理且易于支持多个数据源及分布式数据的消耗。 与Scribe或Flume等以日志为中心的系统相比，Kafka具备同样出色的性能、更强的耐用性（因为复制功能）和更低的端到端延迟。

流处理

许多Kafka用户通过管道来处理数据，有多个阶段： 从Kafka topic中消费原始输入数据，然后聚合，修饰或通过其他方式转化为新的topic， 以供进一步消费或处理。 例如，一个推荐新闻文章的处理管道可以从RSS订阅源抓取文章内容并将其发布到“文章”topic; 然后对这个内容进行标准化或者重复的内容， 并将处理完的文章内容发布到新的topic; 最终它会尝试将这些内容推荐给用户。 这种处理管道基于各个topic创建实时数据流图。 从0.10.0.0开始，在Apache Kafka中，Kafka Streams 可以用来执行上述的数据处理，它是一个轻量但功能强大的流处理库。除Kafka Streams外，可供替代的开源流处理工具还包括Apache Storm 和Apache Samza.

采集日志

Event sourcing是一种应用程序设计风格，按时间来记录状态的更改。 Kafka 可以存储非常多的日志数据，为基于 event sourcing 的应用程序提供强有力的支持。

提交日志

Kafka 可以从外部为分布式系统提供日志提交功能。 日志有助于记录节点和行为间的数据，采用重新同步机制可以从失败节点恢复数据。 Kafka的日志压缩 功能支持这一用法。 这一点与Apache BookKeeper 项目类

似。

1.3 快速开始

本教程假定您是一只小白，没有Kafka 或ZooKeeper 方面的经验。 Kafka控制脚本在Unix和Windows平台有所不同，在Windows平台，请使用 `bin\windows\` 而不是 `bin/`, 并将脚本扩展名改为 `.bat`.

Step 1: 下载代码

下载 1.0.0版本并解压缩。.

```
> tar -xzf kafka_2.11-1.0.0.tgz
> cd kafka_2.11-1.0.0
```

Step 2: 启动服务器

Kafka 使用 ZooKeeper 如果你还没有ZooKeeper服务器，你需要先启动一个ZooKeeper服务器。 您可以通过与kafka打包在一起的便捷脚本来快速简单地创建一个单节点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)
...
```

Step 3: 创建一个 topic

让我们创建一个名为“test”的topic，它有一个分区和一个副本：

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

现在我们可以运行list（列表）命令来查看这个topic：

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

或者，您也可将代理配置为：在发布的topic不存在时，自动创建topic，而不是手动创建。

Step 4: 发送一些消息

Kafka自带一个命令行客户端，它从文件或标准输入中获取输入，并将其作为message（消息）发送到Kafka集群。默认情况下，每行将作为单独的message发送。

运行 producer，然后在控制台输入一些消息以发送到服务器。

```
> bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test
This is a message
This is another message
```

Step 5: 启动一个 consumer

Kafka 还有一个命令行consumer（消费者），将消息转储到标准输出。

```
> bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic test --from-beginning
This is a message
This is another message
```

如果您将上述命令在不同的终端中运行，那么现在就可以将消息输入到生产者终端中，并将它们在消费终端中显示出来。

所有的命令行工具都有其他选项；运行不带任何参数的命令将显示更加详细的使用信息。

Step 6: 设置多代理集群

到目前为止，我们一直在使用单个代理，这并不好玩。对 Kafka来说，单个代理只是一个大小为一的集群，除了启动更多的代理实例外，没有什么变化。 为了深入了解它，让我们把集群扩展到三个节点（仍然在本地机器上）。

首先，为每个代理创建一个配置文件 (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并设置如下属性：

```
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
```

`broker.id` 属性是集群中每个节点的名称，这一名称是唯一且永久的。我们必须重写端口和日志目录，因为我们在同一台机器上运行这些，我们不希望所有的代理尝试在同一个端口注册，或者覆盖彼此的数据。

我们已经建立Zookeeper和一个单节点了，现在我们只需要启动两个新的节点：

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

现在创建一个副本为3的新topic：

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

Good，现在我们有一个集群，但是我们怎么才能知道那些代理在做什么呢？运行"describe topics"命令来查看：

```
> 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
```

以下是对输出信息的解释。第一行给出了所有分区的摘要，下面的每行都给出了一个分区的信息。因为我们只有一个分区，所以只有一行。

- “leader”是负责给定分区所有读写操作的节点。每个节点都是随机选择的部分分区的领导者。
- “replicas”是复制分区日志的节点列表，不管这些节点是leader还是仅仅活着。
- “isr”是一组“同步”replicas，是replicas列表的子集，它活着并被指到leader。

请注意，在示例中，节点1是该主题中唯一分区的领导者。

我们可以在已创建的原始主题上运行相同的命令来查看它的位置：

```
> 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
```

这没什么大不了，原来的主题没有副本且在服务器0上。我们创建集群时，这是唯一的服务器。

让我们发表一些信息给我们的新topic：

```
> bin/kafka-console-producer.sh --broker-list localhost:9092 --topic my-replicated-topic
...
my test message 1
my test message 2
^C
```

现在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
```

让我们来测试一下容错性。 Broker 1 现在是 leader，让我们来杀了它：

```
> 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 上用:

```
> wmic process where "caption = 'java.exe' and commandline like '%server-1.properties%'" get processid
ProcessId
6016
> taskkill /pid 6016 /f
```

领导权已经切换到一个从属节点，而且节点1也不在同步副本集中了：

```
> 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: 1,2,0 Isr: 2,0
```

不过，即便原先写入消息的leader已经不在，这些消息仍可用于消费：

```
> bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --from-beginning --topic my-replicated-topic
...
my test message 1
my test message 2
^C
```

Step 7: 使用Kafka Connect来导入/导出数据

从控制台读出数据并将其写回是十分方便操作的，但你可能需要使用其他来源的数据或将数据从Kafka导出到其他系统。针对这些系统， 你可以使用Kafka Connect来导入或导出数据，而不是写自定义的集成代码。

Kafka Connect是Kafka的一个工具，它可以将数据导入和导出到Kafka。它是一种可扩展工具，通过运行 *connectors（连接器）*， 使用自定义逻辑来实现与外部系统的交互。 在本文中，我们将看到如何使用简单的connectors来运行Kafka Connect，这些connectors 将文件中的数据导入到Kafka topic中，并从中导出数据到一个文件。

首先，我们将创建一些种子数据来进行测试：

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

在Windows系统使用:

```
> echo foo> test.txt
> echo bar>> test.txt
```

接下来，我们将启动两个*standalone（独立）*运行的连接器，这意味着它们各自运行在一个单独的本地专用进程上。 我们提供三个配置文件。首先是Kafka Connect的配置文件，包含常用的配置，如Kafka brokers连接方式和数据的序列化格式。 其余的配置文件均指定一个要创建的连接器。这些文件包括连接器的唯一名称，类的实例，以及其他连接器所需的配置。

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

这些包含在Kafka中的示例配置文件使用您之前启动的默认本地群集配置，并创建两个连接器： 第一个是源连接器，用于从输入文件读取行，并将其输入到 Kafka topic。 第二个是接收器连接器，它从Kafka topic中读取消息，并在输出文件中生成一行。

在启动过程中，你会看到一些日志消息，包括一些连接器正在实例化的指示。 一旦Kafka Connect进程启动，源连接器就开始从 `test.txt` 读取行并且 将它们生产到主题 `connect-test` 中，同时接收器连接器也开始从主题 `connect-test` 中读取消息， 并将它们写入文件 `test.sink.txt` 中。我们可以通过检查输出文件的内容来验证数据是否已通过整个pipeline进行交付：

```
> more test.sink.txt
foo
bar
```

请注意，数据存储在Kafka topic `connect-test` 中，因此我们也可以运行一个console consumer（控制台消费者）来查看 topic 中的数据（或使用custom consumer（自定义消费者）代码进行处理）：

```
> 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"}
...
```

连接器一直在处理数据，所以我们可以将数据添加到文件中，并看到它在pipeline 中移动：

```
> echo Another line>> test.txt
```

您应该可以看到这一行出现在控制台用户输出和接收器文件中。

Step 8:使用 Kafka Streams 来处理数据

Kafka Streams是用于构建实时关键应用程序和微服务的客户端库，输入与输出数据存储在Kafka集群中。Kafka Streams把客户端能够轻便地编写部署标准Java和Scala应用程序的优势与Kafka服务器端集群技术相结合，使这些应用程序具有高度伸缩性、弹性、容错性、分布式等特性。 本快速入门示例将演示如何运行一个基于该库编程的流式应用程序。

1.4 生态圈

在主发行版之外，有大量的工具与 Kafka 集成。在 生态圈 里列出了许多内容，有流处理系统、Hadoop集成、监视和部署工具。

1.5 升级版本

从 0.8.x, 0.9.x, 0.10.0.x, 0.10.1.x, 0.10.2.x, 0.11.0.x 升级到1.0.0

Kafka 1.0.0 介绍了通信协议方面的改变。 遵循下面的滚动升级计划，可以保证您在升级过程中不用停机。 在升级之前，请先查看1.0.0版本中显著的变化。

滚动升级计划：

1. 更新所有代理上的server.properties 并添加以下属性： CURRENT_KAFKA_VERSION代表指您要升级的版本。CURRENT_MESSAGE_FORMAT_VERSION代表当前正在使用的消息格式版本。 如果您以前重写过消息格式版本，则应保留当前值。或者如果您正从0.11.0.x之前的版本升级，则应将current_message_format_version设置为与current_kafka_version匹配的值。
   - inter.broker.protocol.version=CURRENT_KAFKA_VERSION (例如 0.8.2, 0.9.0, 0.10.0, 0.10.1, 0.10.2, 0.11.0)。
   - log.message.format.version=CURRENT_MESSAGE_FORMAT_VERSION (请参阅 升级后在性能方面潜在的影响 ， 了解有关此配置的详细信息。

   如果您从0.11.0.x升级，且没有重写消息格式，那么您只需要覆盖inter-broker协议格式。
   - inter.broker.protocol.version=CURRENT_KAFKA_VERSION (例如 0.8.2, 0.9.0, 0.10.0, 0.10.1, 0.10.2, 0.11.0)。
2. 一次升级一个代理：关闭代理，更新代码，重新启动代理。
3. 整个群集升级后，通过编辑修改协议版本 `inter.broker.protocol.version` 并将其设置为1.0。
4. 重新启动代理，以使新的协议版本生效。
5. 如果您按照上面的指示重写了消息格式版本，则需要再执行一次滚动重启才能将其升级到最新版本。一旦所有（或大部分的）consumer升级到0.11.0或更高版本，请将每个代理上的log.message.format.version更改为1.0，然后逐个重启它们。 请注意，以前的Scala consumer 不支持0.11中新的消息格式，因此为了避免转换中的性能成本（或者使用一次语义），必须使用较新的Java consumer 。

其他升级说明：

1. 如果你可以接受停机，那么你可以把所有的broker关闭，更新代码并重启。系统将默认启动新的协议。
2. 在升级broker后，可以随时更新协议版本并重启。这不需要在升级broker后立即进行。更新消息格式版本也是如此。

1.0.0中显著的变化

- 由于功能稳定，所以默认启动删除topic功能。希望保留以前操作的用户请将代理配置 `delete.topic.enable` 设置为false。请记住，在topic中删除数据的操作是不可逆的（即没有“撤销删除”操作）。
- 对于可以按时间戳搜索的topic，如果找不到分区的偏移量，则会将该分区显示在搜索结果中，并将偏移量值设置为空。在以前的版本中，这类分区不会显示。这种更改是为了使搜索行为与不支持时间戳搜索的topic相一致。

- 如果 `inter.broker.protocol.version` 是1.0或更高版本，即使存在脱机日志目录，代理也会一直保持联机，并在实时日志目录上提交副本。由硬件故障导致的IOException，日志目录可能会变为脱机状态。用户需要监控每个代理度量标准 `offlineLogDirectoryCount` 来检查是否存在离线日志目录。
- 增加了一个可回溯的异常 KafkaStorageException 。 如果客户端的FetchRequest或ProducerRequest的版本不支持KafkaStorageException，则KafkaStorageException将在响应中转换为NotLeaderForPartitionException。
- -XX：在默认的JVM设置中，+ DisableExplicitGC被-XX:+ ExplicitGCInvokesConcurrent替换。在某些情况下，这有助于避免通过直接缓冲区分配本机内存时出现的内存异常。
- 重写的 `handleError` 方法已经从以下过时类中除去 `kafka.api` ： 包 `FetchRequest` ， `GroupCoordinatorRequest` ， `OffsetCommitRequest` ， `OffsetFetchRequest` ， `OffsetRequest` ， `ProducerRequest` ，和 `TopicMetadataRequest` 。这只是为了在代理上使用，但是实际上它已经不再被使用了，实现也没有被维护。只是因为二进制兼容性，保留了一个存根。
- Java客户端和工具现在接受任何字符串作为客户端ID。
- `kafka-consumer-offset-checker.sh` 工具已被弃用。使用 `kafka-consumer-groups.sh` 来得到consumer group 的详细信息
- SimpleAclAuthorizer默认将拒绝访问日志记录到授权人日志中。
- `AuthenticationException` 中的一个子类向客户端报告身份验证失败日志。如果客户端连接失败，并不会重新进行验证 。
- 自定义 `SaslServer` 实现可能会向客户端抛出 `SaslAuthenticationException` 来提供有关身份验证失败的错误信息。同时应注意在异常信息中，不要向未授权的客户泄露任何安全方面的关键信息。
- 向JMX提供版本和提交ID 的 `app-info` 将被弃用，由提供这些属性的metrics（度量）来替换。
- Kafka metrics 现在可能包含非数字值。 `org.apache.kafka.common.Metric#value()` 已被弃用并返回 `0.0` 以最大限度地减少用户读取每个客户端值时系统断开的概率（用户调用 `MetricsReporter` 或 `metrics()` 来读取）。 `org.apache.kafka.common.Metric#metricValue()` 用来检索数字和非数字的度量值
- 每个 Kafka 速率指标都有相应的累计计数度量标准，带后缀 `-total` 方便后续处理。 例如， `records-consumed-rate` 对应的度量标准是 `records-consumed-total` 。
- 当系统属性 `kafka_mx4jenable` 设置为 `true` 时，Mx4j才会启用。以前它是默认启用的，如果 `kafka_mx4jenable` 设置为 `true` ，则禁用Mx4j。
- 客户端jar 包中的 `org.apache.kafka.common.security.auth` 包现在是公有的，已被添加到javadocs中。这个包中的内部类已经移到其他地方了。
- 当使用授权且用户对topic没有必备的权限时，broker 返回TOPIC_AUTHORIZATION_FAILED错误表示broker对于已存在的topic无权限。如果用户具有权限但topic不存在，则返回UNKNOWN_TOPIC_OR_PARTITION错误。
- 为新的consumer配置config/consumer.properties 文件中的属性。

新的版本协议

- KIP-112: LeaderAndIsrRequest v1 引入一个分区字段 `is_new` 。
- KIP-112: UpdateMetadataRequest v4 引入一个分区字段 `offline_replicas` 。

- KIP-112: MetadataResponse v5 引入一个分区字段 `offline_replicas` 。
- KIP-112: ProduceResponse v4 引入了错误代码KafkaStorageException。
- KIP-112: FetchResponse v6 引入了错误代码KafkaStorageException。
- KIP-152:添加SaslAuthenticate request来报告身份验证失败。当SaslHandshake request版本大于0，将使用此请求。

升级 1.0.0 Kafka Streams 应用程序

- 将Streams应用程序从0.11.0升级到1.0.0不需要使用代理。Kafka Streams 1.0.0应用程序可以连接到0.11.0，0.10.2和0.10.1的代理（却不能连接到0.10.0代理）。
- 如果您正在监控 streams 指标，则需要更改一下报告和代码中的指标名称，因为传递指标的层次结构已更改。
- 有些公共的API，如 `ProcessorContext#schedule()` 、 `Processor#punctuate()` 、 `KStreamBuilder` 和 `TopologyBuilder` 正在被新的API取代。我们建议进行相应的代码更改，在升级时这些改变是细微的，因为新的API看起来非常相似。
- 更多详细信息，请参阅 1.0.0版本中Streams API 的变化。 。

从 0.8.x, 0.9.x, 0.10.0.x, 0.10.1.x 或 0.10.2.x 升级到 0.11.0.0

Kafka 0.11.0.0 引入了一个新的消息格式版本，在有线协议方面也有变化。 遵循下面的滚动升级计划，可以保证您在升级过程中不用停机。在升级之前，请先查看0.11.0.0版本中显著的变化。

从0.10.2 版本开始，Java客户端（生产者和消费者）已经可以与旧代理进行通信，0.11.0版本客户可以与0.10.0及其以上的代理进行通信。但如果代理版本大于0.10.0，则须先升级Kafka集群中的所有代理，然后再升级客户端。0.11.0版本的代理支持0.8.x及其以上的客户端。

对于滚动升级：

1. 更新所有代理上的server.properties并添加以下属性：CURRENT_KAFKA_VERSION指将要升级的版本。CURRENT_MESSAGE_FORMAT_VERSION指当前正在使用的消息格式版本。如果您以前没有重写消息格式，那么应该将CURRENT_MESSAGE_FORMAT_VERSION设置为与CURRENT_KAFKA_VERSION匹配的版本。
   - inter.broker.protocol.version=CURRENT_KAFKA_VERSION (例如： 0.8.2，0.9.0，0.10.0，0.10.1 或 0.10.2).
   - log.message.format.version=CURRENT_MESSAGE_FORMAT_VERSION （想了解有关此配置的详细信息，请参阅 升级后潜在的性能影响。）
2. 一次升级一个代理：关闭代理，更新代码并重启。
3. 整个群集升级后，通过编辑修改协议版本 `inter.broker.protocol.version` 为0.11.0，但不要更改 `log.message.format.version` 。
4. 重启代理，以使新的协议版本生效。

5. 一旦所有（或大部分）消费者升级到0.11.0及以上版本，则将每个代理的log.message.format.version更改为0.11.0，然后逐一重启它们。请注意，较低版本的Scala消费者不支持新的消息格式，因此为了避免向下转换的性能成本（或者利用一次语义），必须使用新的Java消费者。

其他升级说明：

1.
2. 在升级broker后，可以随时更新协议版本并重启。这不需要在升级broker后立即进行。更新消息格式版本也是如此。
3. 在更新全局设置 `log.message.format.version` 之前，也可以使用主题管理工具（`bin/kafka-topics.sh`）在各个topic上启用0.11.0消息格式。
4. 如果要从0.10.0之前的版本升级，则在切换到0.11.0之前，不必先将消息格式更新为0.10.0。

升级 0.10.2 Kafka Streams 应用程序

- 将Streams应用程序从 0.10.2 升级到 0.11.0 不需要使用代理。Kafka Streams 0.11.0应用程序可以连接到0.11.0，0.10.2和0.10.1的代理（却不能连接到0.10.0代理）。
- 如果您自定义配置 `key.serde`， `value.serde` 和 `timestamp.extractor`，建议使用替换的配置参数，因为这些配置已被弃用。
- 更多详细信息，请参阅 0.11.0版本中Streams API 的变化。

0.11.0.0版本中显著的变化

- 现在默认禁用 Unclean leader选择。这一新的默认值有利于耐用性而非可用性。希望保留原有配置的用户可将代理配置 `unclean.leader.election.enable` 改为为 `true`。
- 生产者配置 `block.on.buffer.full`， `metadata.fetch.timeout.ms` 和 `timeout.ms` 已被删除。他们在 0.9.0.0版本中就被弃用。
- `offsets.topic.replication.factor` 配置现在被限制由 topic 自动生成。当群集大小不满足复制因子要求时，topic 内部自动生成将失败并返回 GROUP_COORDINATOR_NOT_AVAILABLE 错误。
- 快速压缩数据时，为提高压缩率，制造商和代理默认使用的压缩块大小为（2 x 32 KB）而不是1 KB。有报告表明：使用较小的块压缩后，数据占用的空间比使用大的块多50%对于这种情况来说，拥有5000个分区的生产者需要额外的315 MB的JVM堆。
- 同样，使用gzip压缩数据时，生产者和代理会将缓冲区大小设置为8 KB而不是1 KB。gzip的默认值过低（512字节）。
- 代理配置 `max.message.bytes` 现在指批量消息的大小。之前将其应用于批量压缩的消息，或单独应用于未压缩的消息。批量消息可能只包含单个消息，因此大多数情况下，单个消息的大小只能通过批量格式的上限来控制。不过，消息格式转换有一些微妙的含义（详见 below for more detail）。请注意，代理以前会确保在每个提取请求中至少返回一条消息（无论总分区级别和分区级别的提取大小），但这一行为现在适用于批量消息。
- 默认启用GC日志旋转，详情请参阅KAFKA-3754。
- RecordMetadata，MetricName和Cluster类的构造函数已被删除。

- 通过提供用户header读写访问，新Headers接口增加了对用户header的支持。
- ProducerRecord和ConsumerRecord通过 `Headers headers()` 方法调用新的Headers API。
- ExtendedSerializer和ExtendedDeserializer接口用来来支持头文件的序列化和反序列化。如果配置的串行器和解串器不是上述类，那么头文件将被忽略。
- 引入了一个新的配置 `group.initial.rebalance.delay.ms` ，该配置指定时间以毫秒为单位。 `GroupCoordinator` 将延迟初始消费者以实现再平衡。当有新成员加入group时，将根据 `group.initial.rebalance.delay.ms` 的值进行平衡，延迟的时间最高可达 `max.poll.interval.ms` （默认为3秒）。在开发和测试中，为了不延迟执行的时间，可能需要将其设置为0。
- 在主题请求的元数据不存在时， `org.apache.kafka.common.Cluster#partitionsForTopic` 、 `partitionsForNode` 和 `availablePartitionsForTopic` 方法会返回一个空列表，而不是 `null` （这被认为是不好的做法）。
- Streams API 的配置参数 `timestamp.extractor` 、 `key.serde` 和 `value.serde` 分别被 `default.timestamp.extractor` 、 `default.key.serde` 和 `default.value.serde` 替代。
- 当实例RetriableCommitFailedException通过提交回调时，如遇Java消费者 `commitAsync` API中的偏移提交失败，我们不再公布底层原因。更多详细信息，请参阅KAFKA-5052。

新的版本协议

- KIP-107： FetchRequest v5 引入了一个分区字段 `log_start_offset` 。
- KIP-107：FetchResponse v5 引入了一个分区字段 `log_start_offset` 。
- KIP-82：ProduceRequest v3 在消息协议中引入了一组包含 `key` 字段和 `value` 字段的 `header` 。
- KIP-82：FetchResponse v5 在消息协议中引入了一组包含 `key` 字段和 `value` 字段的 `header` 。

有关一次语义的注记

在生产者方面，Kafka 0.11.0 支持幂等和事务性能力。幂等传递确保消息在单个生产者的生命周期内仅给特定的主题分区传递一次。事务交付允许生产者给多个分区发送数据，这样所有的消息都会被传递成功或失败。这些功能使Kafka符合“恰好一次语义”。有关这些功能的更多详细信息，请参阅用户指南。下面我们将指出一些有关升级群集过程中的特定注意事项。请注意，启用EoS不是必需的，如未使用，不会影响broker的行为。

1. 只有新的Java生产者和消费者支持一次语义。
2. 这些功能主要取决于0.11.0的消息格式。在旧版本中使用将导致不被支持的版本错误。
3. 事务状态存储在一个新的内部主题 `__transaction_state` 中。在首次使用事务性请求API时才创建此主题。同样地，消费者偏移主题也有几个配置设置用来控制主题。如 `transaction.state.log.min.isr` 控制主题的最小ISR。请参阅用户指南中的配置部分以获取完整的选项列表。
4. 对于安全集群，事务性API需要新的ACL，它可以被 `bin/kafka-acls.sh` 工具打开。
5. Kafka的EoS引入了新的请求API，并修改了几个现有的API。更多详细信息，请参阅 KIP-98

有关0.11.0中新消息格式的说明

为了更好地支持生产者的交付语义（见KIP-98）以及提升复制容错能力（参见KIP-101），0.11.0消息格式增强了几个主要的功能。虽然新格式包含了更多信息以实现这些改进，但我们已经使批处理格式更有效率。只要每批消息的数量大于2，就可以降低整体开销。然而，对于单批次，可能会有一些轻微的性能影响。请参阅这里 以便了解我们对新消息格式初始性能的分析结果。您也可以在KIP-98方案中找到更多有关信息格式的细节。

新消息格式中，一个显著的差异是：未压缩的消息会被存储为一个批次。这会对代理配置 `max.message.bytes` （它限制单个批次的大小）有一些影响。首先，如果一个旧版的客户端使用旧格式生产消息到主题分区，且每个消息都比 `max.message.bytes` 小,那么通过上述转换,合并成单批次后，代理仍可能会拒绝它们。通常，这可能发生在单个消息的聚合大小大于 `max.message.bytes` 时。旧的消费者阅读从新格式转换来的消息时也有类似的影响：如果提取大小未被设置为 `max.message.bytes` ，即使单个未压缩的消息小于已配置的获取大小，消费者也可能无法取得进展。此行为不影响0.10.1.0及更高版本的Java客户端，因为它的获取协议是新的，该协议保证即使超过获取大小也能返回至少一条消息。为了解决这些问题，你应该确保：1）生产者的批量大小没有大于 `max.message.bytes` ，并且2）消费者的获取大小为 `max.message.bytes` 。

大多数关于升级到0.10.0消息格式对性能影响的讨论，仍然与0.11.0升级有关。这主要影响不使用TLS保护的群集，因为在这种情况下“零复制”传输是不可行的。为了避免下变换的成本，您应确保客户应用程序升级到最新的0.11.0版本。值得注意的是，由于旧消费者在0.11.0.0已经被弃用，它不支持新的消息格式。您必须升级才能使用新消费者及新的消息格式，这不需要下转换的成本。请注意，0.11.0消费者向后兼容0.10.0及更高版本的代理，所以可以先在升级代理之前升级客户端。

从0.8.x，0.9.x，0.10.0.x或0.10.1.x升级到0.10.2.0

0.10.2.0在线协议方面有变化。遵循下面的滚动升级计划，可以保证您在升级过程中不用停机。请在升级之前，请查看0.10.2.0中的显著更改。

从0.10.2版本开始，Java客户端（生产者和消费者）获得了与旧代理进行通信的能力。0.10.2版本客户端可以与0.10.0版本及其以上的代理进行通信。但是，如果您的代理低于0.10.0版本，则必须先升级Kafka集群中的所有代理，然后再升级您的客户端。0.10.2版代理支持0.8.x及以上的客户端。

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, 0.9.0, 0.10.0 或 0.10.1)。
   - log.message.format.version=CURRENT_KAFKA_VERSION (请参阅升级后潜在的性能影响 ， 了解有关此配置的详细信息。)
2. 一次升级一个代理：关闭代理，更新代码并重启。
3. 整个群集升级完后，设置inter.broker.protocol.version为0.10.2来改变协议版本。

4. 如果以前的消息格式是0.10.0，则将log.message.format.version更改为0.10.2（对于0.10.0，0.10.1和0.10.2而言消息格式是相同的）。如果以前的消息格式版本低于0.10.0，则不要更改log.message.format.version - 只有在所有使用者都升级到0.10.0.0或更高版本后，才应更改此参数。
5. 重启代理，以使新的协议版本生效。
6. 如果此时log.message.format.version仍然低于0.10.0，请等到所有使用者都升级到0.10.0及以上版本，然后将每个代理的log.message.format.version改为0.10.2，逐一重启。

注意：如果你可以接受停机，那么你可以把所有的broker关闭，更新代码并重启。系统将默认启动新的协议。

注意：在升级broker后，可以随时更新协议版本并重启。这不需要在升级broker后立即进行。

升级0.10.1 Kafka Streams应用程序

- 将Streams应用程序从0.10.1升级到0.10.2不需要升级代理。Kafka Streams 0.10.2应用程序可以连接到0.10.2和0.10.1代理（但是不能连接到0.10.0代理）。
- 你需要重新编译你的代码。只交换Kafka Streams库的jar文件将不起作用，并会破坏您的应用程序。
- 如果您使用自定义的（即用户实现的）时间戳提取器，则需要更新代码，因为`TimestampExtractor`界面已更改。
- 如果您使用自定义指标，则需要更新代码，因为`StreamsMetric`界面已更改。
- 请参阅0.10.2中Streams API的更改以获取更多详细信息。

0.10.2.1中的显著变化

- StreamsConfig类中两个配置的默认值已更改，以提高Kafka Streams应用程序的灵活性。Kafka Streams 消费者中`retries`的默认值从0更改为10，`max.poll.interval.ms`的默认值从300000改为`Integer.MAX_VALUE`。

0.10.2.0中的显著变化

- Java客户端（生产者和消费者）可以与旧版本代理通信。0.10.2版本客户端可以与0.10.0及其以上版本的代理进行通信。请注意，使用旧代理时某些功能可能不可用或受限。
- 如果调用线程中断，Java消费者的几个方法可能会抛出`InterruptException`。请参阅`KafkaConsumer`Javadoc以获得更深入的解释。
- Java消费者现在会优雅地关闭。默认情况下，消费者最多等待30秒以完成待处理的请求。已添加了一个新的超时关闭API `KafkaConsumer`来控制最大等待时间。
- 用逗号分隔的多个正则表达式可以通过新的Java消费者中--whitelist选项传递给MirrorMaker。当使用旧的Scala消费者时，这行为与MirrorMaker一致。
- 将Streams应用程序从0.10.1升级到0.10.2不需要升级代理。Kafka Streams 0.10.2应用程序可以连接到0.10.2和0.10.1代理（但不能连接到0.10.0代理）。
- 已从Streams API中删除Zookeeper依赖项。Streams API现在使用Kafka协议来管理内部主题，而不是通过直接修改Zookeeper。这消除了直接访问Zookeeper的权限，也不必在Streams应用中设置

“StreamsConfig.ZOOKEEPER_CONFIG”了。如果Kafka集群被保护，Streams应用程序必须具有所需的安全权限才能创建新主题。

- StreamsConfig类添加了几个新的字段，包括"security.protocol", "connections.max.idle.ms", "retry.backoff.ms", "reconnect.backoff.ms" 和 "request.timeout.ms"。用户应注意默认值，并根据需要进行设置。欲了解更多详情，请参阅3.5 Kafka Streams 配置。

新版本协议

- KIP-88：如果 `topics` 数组设置为 `null`， OffsetFetchRequest v2 将支持检索所有主题的偏移量，
- KIP-88： OffsetFetchResponse v2 引入了一个顶级字段 `error_code` 。
- KIP-103：UpdateMetadataRequest v3 引入一个 `listener_name` 字段作为 `end_points` 数组的元素。
- KIP-108： CreateTopicsRequest v1引入 `validate_only` 字段。
- KIP-108：CreateTopicsResponse v1 引入 `error_message` 字段作为 `topic_errors` 数组的元素。

从0.8.x，0.9.x或0.10.0.X升级到0.10.1.0

0.10.1.0 在线协议方面有变化。遵循下面的滚动升级计划，可以保证您在升级过程中不用停机。请在升级之前，请查看0.10.1.0中的显著更改。
注意：由于引入了新协议，在升级客户端之前先升级您的Kafka群集是非常重要的（即，0.10.1.x客户端仅支持0.10.1.x或更高版本的代理，而0.10.1.x代理支持较旧的客户端）。

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 (请参阅升级后潜在的性能影响，了解有关此配置的详细信息。)
2. 一次升级一个代理：关闭代理，更新代码并重启。
3. 整个群集升级完后，设置 inter.broker.protocol.version为0.10.1.0来改变协议版本。
4. 如果以前的消息格式是0.10.0，则将log.message.format.version更改为0.10.1（对于0.10.0和0.10.1而言消息格式是相同的）。如果以前的消息格式版本低于0.10.0，则不要更改log.message.format.version - 只有在所有使用者都升级到0.10.0.0或更高版本后，才应更改此参数。
5. 重启代理，以使新的协议版本生效。
6. 如果此时log.message.format.version仍然低于0.10.0，请等到所有使用者都升级到0.10.0及以上版本，然后将每个代理的log.message.format.version改为0.10.1，并逐一重启。

注意：如果你可以接受停机，那么你可以把所有的broker关闭，更新代码并重启。系统将默认启动新的协议。

注意：在升级broker后，可以随时更新协议版本并重启。这不需要在升级broker后立即进行。

0.10.1.0中潜在的突出变化

- 日志保留时间不再基于日志段的上次修改时间。相反，它将基于日志段消息的最大时间戳。
- 日志滚动时间不再取决于日志段创建时间。相反，它现在基于消息中的时间戳。进一步来说，如果段中第一条消息的时间戳是T，则当新消息的时间戳大于或等于T + log.roll.ms时，日志滚动。
- 由于每个段添加了时间索引文件，0.10.0的开放文件处理程序将增加33%。
- 时间索引和偏移索引共享相同的索引大小配置。由于每次索引条目是偏移索引大小的1.5倍,用户可能需要增加log.index.size.max.bytes以避免可能的日志频繁滚动。
- 由于索引文件数量的增加，在一些具有大量日志段（如大于15K）的代理中，代理启动期间的日志加载过程可能会更长。根据我们的实验，将num.recovery.threads.per.data.dir设置为1可能会减少日志加载时间。

升级0.10.0 Kafka Streams应用程序

- 将Streams应用程序从0.10.0升级到0.10.1需要代理升级，因为Kafka Streams 0.10.1应用程序只能连接到0.10.1代理。
- 有几个API有变化且不向后兼容(更多的细节请参考0.10.1中Streams API的变化)。因此，您需要更新并重新编译您的代码。只交换Kafka Streams库的jar文件将不起作用，并会破坏您的应用程序。

0.10.1.0中的显著变化

- 新的Java消费者已经通过测试阶段，我们推荐使用它来开发。以前的Scala消费者仍然支持，但是在下一个版本中它们将被弃用，并会在以后的版本中删除。
- `--new-consumer` / `--new.consumer` 转换已不再需要MirrorMaker和消费者控制台之类的工具。只需要通过Kafka broker来连接，而不必使用ZooKeeper。此外，旧消费者控制台已被弃用，并将在以后的版本中删除。
- Kafka群集现在可以由群集ID唯一标识。当代理升级到0.10.1.0后，它会自动生成。群集ID可通过设置kafka.server：type = KafkaServer，name = ClusterId来获得，它是元数据响应的一部分。通过ClusterResourceListener接口，串行器，客户端拦截器和度量记录器可以接收集群ID。
- BrokerState "RunningAsController" (value 4)已被删除。由于一个bug，代理在转换出来之前会一直处于这个状态，因此此时移除该BrokerState的影响是最小的。推荐用于检测指定代理是否为控制器的方法是使用kafka.controller：type = KafkaController，name = ActiveControllerCount指标。
- 新的Java消费者允许用户通过分区上的时间戳搜索偏移量。
- 新的Java消费者可以获得后台线程的状态。新的配置 `max.poll.interval.ms` 可以控制用户主动离开组前轮询调用的最大时间（默认为5分钟）（在用户主动离开组前）。配置 `request.timeout.ms` 的值必须总是大于 `max.poll.interval.ms` ，因为这是JoinGroup请求平衡服务器消费的最大时间，所以我们已经将其默认值更改为5分钟以上。 `session.timeout.ms` 的默认值已经调整到10秒，`max.poll.records` 的默认值已经改为500。
- 当使用授权并且用户没有相关主题的授权时，代理不再返回TOPIC_AUTHORIZATION_FAILED错误，因为这会泄漏主题名称。相反会返回UNKNOWN_TOPIC_OR_PARTITION错误。这可能会导致在使用生产者和消费者时出现意外超时或延迟，因为Kafka客户端在出现未知主题错误时会自动重试。如果您担心这种情况发生，可以查看客户端日志。

- 提取响应的默认大小是固定的（消费者为50 MB，复制为10 MB）。现有分区的限制也适用（消费者和复制均为1 MB）。请注意，这些限制都不是以后的绝对最大值。
- 如果发现大于响应/分区大小限制的消息，则消费者和副本仍然可以进行。更具体地说，如果提取的第一个非空分区中第一条消息大于一个或两个限制，则该消息仍会被返回。
- 重载构造函数被添加到 `kafka.api.FetchRequest` 和 `kafka.javaapi.FetchRequest` 且可以让调用者指定分区的顺序（因为在v3中顺序是重要的）。原先的构造函数已被弃用，在发送请求之前将对分区进行洗牌以避免饥饿问题。

新的版本协议

- ListOffsetRequest v1支持基于时间戳的精确偏移搜索。
- MetadataResponse v2 引入了一个新的字段： "cluster_id"。
- FetchRequest v3可以限制响应大小（除已存在的分区限制外），如果请求进行更改，则返回大于限制的消息，而且分区的顺序是很重要的。
- JoinGroup v1 引入了一个新的字段："rebalance_timeout"。

从0.8.x或0.9.x升级到0.10.0.0

0.10.0.0 版本有 潜在的重大变化（请在升级之前查看）并可能有 升级后的性能影响。请遵循以下建议的滚动升级计划，可确保在升级过程中和升级完成后不会出现停机和性能影响。
注意：由于引入了新的协议，在升级客户端之前，升级您的Kafka集群是非常重要的。

针对0.9.0.0版本客户的说明： 由于0.9.0.0版本中的一个错误，依赖于ZooKeeper的客户端（使用旧Scala高级Consumer和MirrorMaker的客户端）不能与0.10.0.x代理一起工作。因此，**在将**代理升级到0.10.0.x **之前**，应将0.9.0.0客户端升级到0.9.0.1。对于0.8.X或0.9.0.1版本，这一步不是必需的。

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 或0.9.0.0).
   - log.message.format.version=CURRENT_KAFKA_VERSION (请参阅升级后潜在的性能影响，以了解有关此配置的详细信息。)
2. 升级代理：关闭代理，更新代码，重启。
3. 整个群集升级完后，设置inter.broker.protocol.version为0.10.0.0来改变协议版本。注意：您不应该编辑log.message.format.version - 只有当所有使用者都升级到0.10.0.0后，该参数才能被更改。
4. 重启代理，以使新的协议版本生效。
5. 如果所有使用者都升级到0.10.0，更改每个代理的log.message.format.version为0.10.0，然后逐个重启。

注意： 如果你可以接受停机，那么你可以把所有的broker关闭，更新代码并重启。系统将默认启动新的协议。

注意： 在升级broker后，可以随时更新协议版本并重启。这不需要在升级broker后立即进行。

在升级到0.10.0.0后，可能会对性能造成的影响

0.10.0中的消息格式包含一个新的时间戳字段，并使用压缩消息的相对偏移量。磁盘上的消息格式可以通过server.properties文件中的log.message.format.version来配置。磁盘上默认的消息格式是0.10.0。如果消费者客户端使用的是0.10.0.0之前的版本，则只能适用0.10.0之前的消息格式。在这种情况下，代理可以将消息从0.10.0格式转换为较早的格式，然后将响应发送给旧版本的消费者。但在这种情况下，代理不能使用零复制转移。Kafka社区中关于性能影响的报告显示，在升级之后，CPU使用率从20%上升到100%，这会迫使所有客户端立即升级，以使性能恢复正常。为避免这种消息转换，可以在代理升级到0.10.0.0时，将log.message.format.version设置为0.8.2或0.9.0。这样，代理仍可使用零拷贝将数据发送给以前的消费者。消费者升级之后，可以将代理上的消息格式更改为0.10.0，就可以使用新时间戳和改进压缩的新消息格式。Kafka支持这一转换以确保兼容性也有利于一些尚未更新到较新客户端的应用程序。但即使在过度配置的群集上也不支持所有消费者间的通信。因此，在代理升级后，尽量避免信息转换是非常重要的，但大多数客户端还没有。

对于升级到0.10.0.0的客户端，不会影响性能。

注意：通过设置消息格式版本，可以让所有现有消息都在该版本及其以下。且10.0.0之前的消费者可能会中断。尤其是，把消息格式设置为0.10.0后，不应将其更改回以前的格式，因为它可能会中断0.10.0.0版本前的使用者。

注意：由于在每条消息中额外地引入了时间戳，由于增加开销，发送小消息的生产者可能会产生消息吞吐量下降。同样，复制里现在每个消息额外传输8个字节。如果您运行的集群接近网络容量，则可能会压垮网卡，并由过载而导致故障和性能问题。

注意：如果您在生产者上启用了压缩，您可能会注意到某些情况下，生产者吞吐量或代理的压缩率会降低。在接收压缩消息时，0.10.0的代理会避免重新压缩消息，通常会减少延迟并提高吞吐量。然而，在某些情况下，这可能会减少生产者的批量大小，导致更差的吞吐量。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。另外，如果用snappy压缩消息的生产者缓冲区比代理缓冲区小，可能会不利于磁盘消息的压缩率。我们打算在更高版本的Kafka中进行配置。

0.10.0.0中潜在的重大更改

- 从Kafka 0.10.0.0开始，Kafka中的消息格式版本代表Kafka版本。例如，消息格式0.9.0指的是Kafka 0.9.0支持的最高消息版本。
- 已经引入消息格式0.10.0，并默认使用。它包含消息中的时间戳字段及用于压缩消息的相对偏移量。
- ProduceRequest / Response v2被引入，默认支持消息格式0.10.0
- FetchRequest / Response v2被引入，默认支持消息格式0.10.0
- MessageFormatter接口从`def writeTo(key: Array[Byte], value: Array[Byte], output: PrintStream)`改为`def writeTo(consumerRecord: ConsumerRecord[Array[Byte], Array[Byte]], output: PrintStream)`
- MessageReader接口从`def readMessage(): KeyedMessage[Array[Byte], Array[Byte]]`改为`def readMessage(): ProducerRecord[Array[Byte], Array[Byte]]`

- MessageFormatter的包从 `kafka.tools` 改为 `kafka.common`
- MessageReader的包从 `kafka.tools` 改为 `kafka.common`
- MirrorMakerMessageHandler不再公开 `handle(record: MessageAndMetadata[Array[Byte], Array[Byte]])` 方法，因为它从未被调用过。
- 0.7 KafkaMigrationTool不再与Kafka打包在一起。如果您需要从0.7迁移到0.10.0，请先迁移到0.8，然后按照升级过程从0.8升级到0.10.0。
- 新的消费者已经标准化它的API，接收 `java.util.Collection` 序列类型作为方法的参数。现有的代码可能需要更新才能使用0.10.0客户端库。
- LZ4压缩的消息使用可互操作的帧规范（LZ4f v1.5.1）进行处理。为了保持与旧客户端的兼容性，此更改仅适用于0.10.0及更高版本。使用v0 / v1（消息格式0.9.0）生成/获取LZ4压缩消息的客户端应该继续使用0.9.0成帧实现。使用Produce / Fetch协议v2或更高版本的客户端应使用可互操作的LZ4f成帧。可在http://www.lz4.org/上找到可互操作的LZ4库列表。

0.10.0.0中的显著变化

- 从Kafka 0.10.0.0开始，一个名为**Kafka Streams**的新客户端库被用于对存储在Kafka主题中的数据进行流处理。由于上面提到的消息格式更改，这个新的客户端库仅适用于0.10.x及其以上版本的代理。欲了解更多信息，请阅读Streams 文档。
- 新用户的配置参数receive.buffer.bytes默认为64K。
- 新的使用者公开配置参数 `exclude.internal.topics` 以防止内部主题（例如消费者偏移主题）意外地被符合正则表达式的订阅源订阅。默认情况下，该设置启用。
- 旧的Scala生产者已被弃用。用户应尽快将其代码迁移到kafka-clients JAR中的Java生产者。
- 新的消费者API已稳定。

从0.8.0，0.8.1.X或0.8.2.X升级到0.9.0.0

0.9.0.0有潜在的重大变化（请在升级之前查看），而且代理协议也有所改变。这意味着升级的代理和客户端可能与旧版本不兼容。在升级客户端之前升级您的Kafka集群是非常重要的。如果您正在使用MirrorMaker，则应先升级下游群集。

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. 升级代理。这可以通过将其关闭，更新代码并重启来完成。
3. 一旦整个群集升级完毕，将inter.broker.protocol.version设置为0.9.0.0来改变协议版本。
4. 重启代理，以使新的协议版本生效

注意：如果你可以接受停机，那么你可以把所有的broker关闭，更新代码并重启。系统将默认启动新的协议。

注意：在升级broker后，可以随时更新协议版本并重启。这不需要在升级broker后立即进行。

0.9.0.0中潜在的重大变化

- 不再支持Java 1.6。
- 不再支持Scala 2.9。
- 现在超过1000的Broker IDs默认自动分配broker IDs并保留。如果您的群集具有高于该阈值的现有代理ID，请确保添加代理配置属性reserved.broker.max.id。
- 删除配置参数replica.lag.max.messages。在决定哪些副本同步时，分区leaders不再考虑滞后消息的数量。
- 现在，配置参数replica.lag.time.max.ms不仅指自上次从副本获取请求后经过的时间，还指最后一次抓取副本的时间。副本仍从leaders 获取最新消息却没有赶上replica.lag.time.max.ms时，将被视为不同步。
- 被压缩的topics不接受没有密钥的消息，如果尝试这样做，生产者会抛出异常。在0.8.x中，没有密钥的消息会导致日志压缩线程出错并退出（并停止压缩所有被压缩的主题）。
- MirrorMaker不再支持多目标群集。因此它只接受一个--consumer.config参数。要镜像多个源群集，每个群集至少需要一个MirrorMaker实例并有自己的消费者配置。
- 在*org.apache.kafka.clients.tools.**中的打包工具已被移至*org.apache.kafka.tools.**。所有包含的脚本仍照常运行，只有直接导入这些类的自定义代码才会受到影响。
- kafka-run-class.sh中更改了默认的Kafka JVM性能选项（KAFKA_JVM_PERFORMANCE_OPTS）。
- kafka-topics.sh脚本（kafka.admin.TopicCommand）现在退出时返回非0。
- 当在主题名字中使用'.' 或 '_'而导致风险度量标准冲突及实际碰撞冲突时，kafka-topics.sh脚本（kafka.admin.TopicCommand）将显示警告。
- kafka-console-producer.sh脚本（kafka.tools.ConsoleProducer）默认使用Java生产者而不是旧的Scala生产者，用户须在“old-producer”中指定使用旧的生产者。
- 默认情况下，所有命令行工具将打印一切日志消息到stderr而不是stdout。

0.9.0.1中的显著变化

- 将broker.id.generation.enable设置为false可以禁用新的代理ID生成功能。
- 配置参数log.cleaner.enable默认为true。这意味着cleanup.policy = compact的主题默认被压缩，根据log.cleaner.dedupe.buffer.size，128 MB的堆将被分配给清理进程。您可以根据您使用的压缩主题来查看log.cleaner.dedupe.buffer.size和其他log.cleaner配置值。
- 新用户的配置参数fetch.min.bytes默认为1。

0.9.0.0中弃用的部分

- kafka-consumer-offset-checker.sh（kafka.tools.ConsumerOffsetChecker）已被弃用。今后，请使用kafka-consumer-groups.sh（kafka.admin.ConsumerGroupCommand）来实现此功能。
- kafka.tools.ProducerPerformance类已被弃用。今后，请使用org.apache.kafka.tools.ProducerPerformance来实现此功能（kafka-producer-perf-test.sh也使用新类）。
- 生产者配置block.on.buffer.full已被弃用，并会在未来的版本中删除。目前其默认值为false。KafkaProducer不再抛出BufferExhaustedException，而是使用max.block.ms值并抛出一个TimeoutException。如果block.on.buffer.full属性被设置为true，则会将max.block.ms设置为Long.MAX_VALUE，且不遵守metadata.fetch.timeout.ms

从0.8.1升级到0.8.2

0.8.2与0.8.1完全兼容。升级代理可以通过将其关闭，更新代码并重启来完成。

从0.8.0升级到0.8.1

0.8.1与0.8完全兼容。0.8.1与0.8完全兼容。

从0.7升级

0.7版本与新版本不兼容。新版本对API、ZooKeeper数据结构、协议以及配置进行了重大更改，以便添加副本（在0.7中没有）。从0.7升级到更高版本需要专门的迁移工具。可以在不停机的情况下完成迁移。

2. APIS

kafka包括五个核心apis：
1. Producer API允许应用程序将数据流发送到Kafka集群中的主题。
2. Consumer API允许应用程序从Kafka集群中的主题读取数据流。
3. Streams API允许将输入主题中的数据流转换为输出主题。
4. Connect API允许实现将数据不断从某些源系统或应用程序中输入Kafka的连接器，或从Kafka输入某些接收器系统或应用程序。
5. AdminClient API允许管理和检查主题，代理和其他Kafka对象。

Kafka通过独立于语言的协议公开了所有功能，客户端可以使用许多编程语言。 但是，只有Java客户端是作为主要Kafka项目的一部分进行维护的，其他的则作为独立的开源项目提供。

2.1 Producer API

Producer API允许应用程序将数据流发送到Kafka集群中的主题。

显示如何使用生产者的例子在javadoc中给出。

要使用生产者，你可以使用下面的maven依赖：

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>1.0.0</version>
</dependency>
```

2.1 Consumer AP

Consumer API允许应用程序从Kafka集群中的主题读取数据流。
展示如何使用消费者的例子在javadoc中给出。

要使用使用者，你可以使用下面的maven依赖：

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>1.0.0</version>
</dependency>
```

2.3 Streams API

Streams API允许将输入主题中的数据流转换为输出主题。
展示如何使用这个库的例子在javadoc中给出

有关使用Streams API的其他文档可以在这里找到。

要使用Kafka Streams，您可以使用以下maven依赖项：

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-streams</artifactId>
    <version>1.0.0</version>
</dependency>

```

2.4 Connect API

Connect API允许实现不断从一些源数据系统中拉入Kafka的连接器，或者从Kafka推入一些接收器数据系统
。
Connect的许多用户不需要直接使用这个API，但是他们可以使用预先建立的连接器而不需要编写任何代码。
有关使用Connect的更多信息，请点击 here.。

那些想要实现自定义连接器的人可以看到 javadoc.

2.5 AdminClient API

TAdminClient API支持管理和检查主题，代理，acl和其他Kafka对象。

要使用AdminClient API，请添加以下Maven依赖项：

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>1.0.0</version>
</dependency>

```

有关AdminClient API的更多信息，请参阅javadoc.

2.6 Legacy APIs

kafka还包括一个更有限的传统生产者和消费者API。 这些旧的Scala API已被弃用，只能用于兼容目的。
关于他们的信息可以在here找到。

3. 配置

Kafkas使用property文件格式的键值对来配置程序。这些键值对配置既可以来自property文件也可以来程序内
部。

3.1 Broker 配置

核心基础配置如下:

- `broker.id`
- `log.dirs`
- `zookeeper.connect`

Topic-level配置及其默认值在下面有更详尽的讨论。

名称	描述	类型	默认值	有效值	重要性
zookeeper.connect	Zookeeper主机地址	string			高
advertised.host.name	不建议:仅在未设置`advertised.listeners` 或 `listeners`时使用。用`advertised.listeners`替换。 主机名发布到zookeeper供客户端使用。在IaaS环境,这可能需要与broker绑定不通的端口。如果未设置,将使用`host.name`的值（如果已经配置）。否则，他将使用java.net.InetAddress.getCanonicalHostName()返回的值。	string	null		高
advertised.listeners	监听器发布到ZooKeeper供客户端使用，如果与`listeners`配置不同。在IaaS环境,这可能需要与broker绑定不通的接口。如果没有设置，将使用`listeners`的配置。与`listeners`不同的是，配置0.0.0.0元地址是无效的。	string	null		高
advertised.port	不建议:仅在未设置“advertised.listeners”或“listeners”时使用。使用`advertised.listeners`代替。这个端口发布到ZooKeeper供客户端使用。在IaaS环境，这可能需要与broker绑定不通的端口。如果没有设置，它将绑定和broker相同的端口。	int	null		高
auto.create.topics.enable	是否允许在服务器上自动创建topic	boolean	true		高
auto.leader.rebalance.enable	是否允许leader平衡。后台线程会定期检查并触发leader平衡。	boolean	true		高
background.threads	用于处理各种后台任务的线程数量	int	10	[1,...]	高
broker.id	用于服务的broker id。如果没设置，将生存一个唯一broker id。为了避免ZooKeeper生成的id和用户配置的broker id相冲突，生成的id将在reserved.broker.max.id的值基础上加1。	int	-1		高
compression.type	为特点的topic指定一个最终压缩类型。此配置接受的标准压缩编码方式有（'gzip', 'snappy', 'lz4'）。此外还有'uncompressed'相当于不压缩；'producer'意味着压缩类型由'producer'决定。	string	producer		高
delete.topic.enable	是否允许删除topic。如果关闭此配置，通过管理工具删除topic将不再生效。	boolean	true		高
host.name	不建议: 仅在未设置`listeners`时使用。使用`listeners`来代替。 如果设置了broker主机名，则他只会当定到这个地址。如果没设置，将绑定到所有接口。	string	""		高
leader.imbalance.check.interval.seconds	由控制器触发分区重新平衡检查的频率设置	long	300		高
leader.imbalance.per.broker.percentage	每个broker允许的不平衡的leader的百分比，如果高于这个比值将触发leader进行平衡。这个值用百分比来指定。	int	10		高
listeners	监听器列表 - 使用逗号分隔URI列表和监听器名称。如果侦听器名称不是安全协议，则还必须设置listener.security.protocol.map。指定主机名为0.0.0.0来绑定到所有接口。留空则绑定到默认接口上。合法监听器列表的示例：PLAINTEXT：// myhost：9092，SSL：//：9091 CLIENT：//0.0.0.0：9092，REPLICATION：// localhost：9093	string	null		高
log.dir	保存日志数据的目录（对log.dirs属性的补充）	string	/tmp/kafka-logs		高
log.dirs	保存日志数据的目录，如果未设置将使用log.dir的配置。	string	null		高
log.flush.interval.messages	在将消息刷新到磁盘之前，在日志分区上累积的消息数量。	long	9223372036854775807	[1,...]	高
log.flush.interval.ms	在刷新到磁盘之前，任何topic中的消息保留在内存中的最长时间（以毫秒为单位）。如果未设置，则使用log.flush.scheduler.interval.ms中的值。	long	null		高
log.flush.offset.checkpoint.interval.ms	日志恢复点的最后一次持久化刷新记录的频率	int	60000	[0,...]	高
log.flush.scheduler.interval.ms	日志刷新器检查是否需要将所有日志刷新到磁盘的频率（以毫秒为单位）	long	9223372036854775807		高
log.flush.start.offset.checkpoint.interval.ms	我们更新日志持久化记录开始offset的频率	int	60000	[0,...]	高
log.retention.bytes	日志删除的大小阈值	long	-1		高
log.retention.hours	日志删除的时间阈值（小时为单位）	int	168		高
log.retention.minutes	日志删除的时间阈值（分钟为单位），如果未设置，将使用log.retention.hours的值	int	null		高
log.retention.ms	日志删除的时间阈值（毫秒为单位），如果未设置，将使用log.retention.minutes的值	long	null		高
log.roll.hours	新日志段轮转时间间隔（小时为单位），次要配置为log.roll.ms	int	168	[1,...]	高
log.roll.jitter.hours	从logrolltimemillis（以小时计）中减去的最大抖动，次要配置log.roll.jitter.ms	int	0	[0,...]	高
log.roll.jitter.ms	从logrolltimemillis（以毫秒计）中减去的最大抖动，如果未设置，则使用log.roll.jitter.hours的配置	long	null		高
log.roll.ms	新日志段轮转时间间隔（毫秒为单位），如果未设置，则使用log.roll.hours配置	long	null		高
log.segment.bytes	单个日志段文件最大大小	int	1073741824	[14,...]	高
log.segment.delete.delay.ms	从文件系统中删除一个日志段文件前的保留时间	long	60000	[0,...]	高
message.max.bytes	kafka允许的最大的一个批次的消息大小。 如果这个数字增加，且有0.10.2版本以下的consumer，那么consumer的提取大小也必须增加，以便他们可以取得这么大的记录批次。 在最新的消息格式版本中，记录总是被组合到一个批次以提高效率。 在以前的消息格式版本中，未压缩的记录不会分组到批次中，并且此限制仅适用于该情况下的单个记录。	int	1000012	[0,...]	高

	可以使用topic设置`max.message.bytes`来设置每个topic。 `max.message.bytes` .				
min.insync.replicas	当producer将ack设置为“全部”（或“-1”）时，min.insync.replicas指定了被认为写入成功的最小副本数。如果这个最小值不能满足，那么producer将会引发一个异常（NotEnoughReplicas或NotEnoughReplicasAfterAppend）。当一起使用时，min.insync.replicas和acks允许您强制更大的耐久性保证。 一个经典的情况是创建一个复本数为3的topic，将min.insync.replicas设置为2，并且producer使用“all”选项。 这将确保如果大多数副本没有写入producer则抛出异常。	int	1	[1,...]	高
num.io.threads	服务器用于处理请求的线程数，可能包括磁盘I/O	int	8	[1,...]	高
num.network.threads	服务器用于从接收网络请求并发送网络响应的线程数	int	3	[1,...]	高
num.recovery.threads.per.data.dir	每个数据目录，用于启动时日志恢复和关闭时刷新的线程数	int	1	[1,...]	高
num.replica.fetchers	从源broker复制消息的拉取器的线程数。增加这个值可以增加follow broker的I/O并行度。	int	1		高
offset.metadata.max.bytes	与offset提交相关联的元数据条目的最大大小	int	4096		高
offsets.commit.required.acks	在offset提交可以接受之前，需要设置acks的数目，一般不需要更改，默认值为-1。	short	-1		高
offsets.commit.timeout.ms	offset提交将延迟到topic所有副本收到提交或超时。这与producer请求超时类似。	int	5000	[1,...]	高
offsets.load.buffer.size	每次从offset段文件往缓存加载时，批量读取的数据大小	int	5242880	[1,...]	高
offsets.retention.check.interval.ms	检查失效offset的频率	long	600000	[1,...]	高
offsets.retention.minutes	超过这个保留期限未提交的offset将被丢弃	int	1440	[1,...]	高
offsets.topic.compression.codec	用于offsets topic的压缩编解码器 - 压缩可用于实现“原子”提交	int	0		高
offsets.topic.num.partitions	Offsets topic的分区数量（部署后不应更改）	int	50	[1,...]	高
offsets.topic.replication.factor	offset topic的副本数（设置的越大，可用性越高）。内部topic创建将失败，直到集群大小满足此副本数要求。	short	3	[1,...]	高
offsets.topic.segment.bytes	为了便于更快的日志压缩和缓存加载，offset topic段字节应该保持相对较小	int	104857600	[1,...]	高
port	不建议: 仅在未设置“listener”时使用。使用`listeners`来代替。端口用来来监听和接受连接	int	9092		高
queued.max.requests	网络线程阻塞前队列允许的最大请求数	int	500	[1,...]	高
quota.consumer.default	不建议:仅在动态默认配额未配置或在zookeeper中使用。任何由clientid区分开来的consumer，如果它每秒产生的字节数多于这个值，就会受到限制	long	9223372036854775807	[1,...]	高
quota.producer.default	不建议:仅在动态默认配额未配置或在zookeeper中使用。任何由clientid区分开来的producer，如果它每秒产生的字节数多于这个值，就会受到限制	long	9223372036854775807	[1,...]	高
replica.fetch.min.bytes	复制数据过程中，replica收到的每个fetch响应，期望的最小的字节数，如果没有收到足够的字节数，就会等待更多的数据，直到达到replicaMaxWaitTimeMs（复制数据超时时间）	int	1		高
replica.fetch.wait.max.ms	副本follow同leader之间通信的最大等待时间，失败了会重试。 此值始终应始终小于replica.lag.time.max.ms，以防止针对低吞吐量topic频繁收缩ISR	int	500		高
replica.high.watermark.checkpoint.interval.ms	high watermark被保存到磁盘的频率，用来标记日后恢复点/td>	long	5000		高
replica.lag.time.max.ms	如果一个follower在这个时间内没有发送fetch请求或消费leader日志到结束的offset，leader将从ISR中移除这个follower，并认为这个follower已经挂了	long	10000		高
replica.socket.receive.buffer.bytes	socket接收网络请求的缓存大小	int	65536		高
replica.socket.timeout.ms	副本复制数据过程中，发送网络请求的socket超时时间。这个值应该大于replica.fetch.wait.max.ms的值	int	30000		高
request.timeout.ms	该配置控制客户端等待请求响应的最长时间。如果在超时之前未收到响应，则客户端将在必要时重新发送请求，如果重试仍然失败，则请求失败。	int	30000		高
socket.receive.buffer.bytes	服务端用来处理socket连接的SO_RCVBUFF缓冲大小。如果值为-1，则使用系统默认值。	int	102400		高
socket.request.max.bytes	socket请求的最大大小，这是为了防止server跑光内存，不能大于Java堆的大小。	int	104857600	[1,...]	高
socket.send.buffer.bytes	服务端用来处理socket连接的SO_SNDBUF缓冲大小。如果值为-1，则使用系统默认值。	int	102400		高
transaction.max.timeout.ms	事务允许的最大超时时间。如果客户请求的事务超时，那么broker将在InitProducerIdRequest中返回一错误。 这样可以防止客户超时时间过长，从而阻碍consumers读取事务中包含的topic。	int	900000	[1,...]	高
transaction.state.log.load.buffer.size	将producer ID和事务加载到高速缓存中时，从事务日志段（the transaction log segments）中批量读取的大小。	int	5242880	[1,...]	高
transaction.state.log.min.isr	覆盖事务topic的min.insync.replicas配置	int	2	[1,...]	高
transaction.state.log.num.partitions	事务topic的分区数（部署后不应该修改）	int	50	[1,...]	高
transaction.state.log.replication.factor	事务topic的副本数（设置的越大，可用性越高）。内部topic在集群数满足副本数之前，将会一直创建失败。	short	3	[1,...]	高
transaction.state.log.segment.bytes	事务topic段应保持相对较小，以便于更快的日志压缩和缓存负载。	int	104857600	[1,...]	高
transactional.id.expiration.ms	事务协调器在未收到任何事务状态更新之前，主动设置producer的事务标识为过期之前将等待的最长时间（以毫秒为单位）	int	604800000	[1,...]	高
unclean.leader.election.enable	指定副本是否能够不再ISR中被选举为leader，即使这样可能会丢数据	boolean	false		高

zookeeper.connection.timeout.ms	与ZK server建立连接的超时时间,没有配置就使用zookeeper.session.timeout.ms	int	null		高
zookeeper.session.timeout.ms	ZooKeeper的session的超时时间	int	6000		高
zookeeper.set.acl	ZooKeeper客户端连接是否设置ACL安全y安装	boolean	false		高
broker.id.generation.enable	是否允许服务器自动生成broker.id。如果允许则产生的值会交由reserved.broker.max.id审核	boolean	true		中
broker.rack	broker的机架位置。 这将在机架感知副本分配中用于容错。例如：RACK1，us-east-1	string	null		中
connections.max.idle.ms	连接空闲超时：服务器socket处理线程空闲超时关闭时间	long	600000		中
controlled.shutdown.enable	是否允许服务器关闭broker服务	boolean	true		中
controlled.shutdown.max.retries	当发生失败故障时，由于各种原因导致关闭服务的次数	int	3		中
controlled.shutdown.retry.backoff.ms	在每次重试关闭之前，系统需要时间从上次故障状态（控制器故障切换，副本延迟等）中恢复。这个配置决定了重试之前等待的时间。	long	5000		中
controller.socket.timeout.ms	控制器到broker通道的socket超时时间	int	30000		中
default.replication.factor	自动创建topic时的默认副本个数	int	1		中
delete.records.purgatory.purge.interval.requests	删除purgatory中请求的清理间隔时间（purgatory：broker对于无法立即处理的请求，将会放在purgatory中，当请求完成后，并不会立即清除，还会继续在purgatory中占用资源，直到下一次delete.records.purgatory.purge.interval.requests）	int	1		中
fetch.purgatory.purge.interval.requests	提取purgatory中请求的间隔时间	int	1000		中
group.initial.rebalance.delay.ms	在执行第一次重新平衡之前，group协调器将等待更多consumer加入group的时间。延迟时间越长意味着重新平衡的工作可能越小，但是等待处理开始的时间增加。	int	3000		中
group.max.session.timeout.ms	consumer注册允许的最大会话超时时间。超时时间越短，处理心跳越频繁从而使故障检测更快，但会导致broker被抢占更多的资源。	int	300000		medium
group.min.session.timeout.ms	consumer注册允许的最小会话超时时间。超时时间越短，处理心跳越频繁从而使故障检测更快，但会导致broker被抢占更多的资源。	int	6000		中
inter.broker.listener.name	broker间通讯的监听器名称。如果未设置，则侦听器名称由security.inter.broker.protocol定义。同时设置此项和security.inter.broker.protocol属性是错误的，只设置一个。	string	null		中
inter.broker.protocol.version	指定使用哪个版本的 inter-broker 协议。 在所有broker升级到新版本之后，这通常会有冲突。一些有效的例子是：0.8.0, 0.8.1, 0.8.1.1, 0.8.2, 0.8.2.0, 0.8.2.1, 0.9.0.0, 0.9.0.1，详情可以检查apiversion的完整列表	string	1.0-IV0		中
log.cleaner.backoff.ms	检查log是否需要清除的时间间隔。	long	15000	[0,...]	中
log.cleaner.dedupe.buffer.size	日志去重清理线程所需要的内存	long	134217728		中
log.cleaner.delete.retention.ms	日志记录保留时间	long	86400000		中
log.cleaner.enable	在服务器上启用日志清理器进程。如果任何topic都使用cleanup.policy = compact，包括内部topic offset，则建议开启。如果被禁用的话，这些topic将不会被压缩，而且会不断增长。	boolean	true		中
log.cleaner.io.buffer.load.factor	日志清理器去重的缓存负载数。完全重复数据的缓存比例可以改变。数值越高，清理的越多，但会导致更多的hash冲突	double	0.9		中
log.cleaner.io.buffer.size	所有清理线程的日志清理I/O缓存区所需要的内存	int	524288	[0,...]	中
log.cleaner.io.max.bytes.per.second	日志清理器受到的大小限制数，因此它的I/O读写总和将小于平均值	double	1.7976931348623157E308		中
log.cleaner.min.cleanable.ratio	日志中脏数据清理比例	double	0.5		中
log.cleaner.min.compaction.lag.ms	消息在日志中保持未压缩的最短时间。 仅适用于正在压缩的日志。	long	0		中
log.cleaner.threads	用于日志清理的后台线程的数量	int	1	[0,...]	中
log.cleanup.policy	超出保留窗口期的日志段的默认清理策略。用逗号隔开有效策略列表。有效策略：“delete”和“compact”	list	delete	[compact, delete]	中
log.index.interval.bytes	添加offset索引字段大小间隔（设置越大，代表扫描速度越快，但是也更耗内存）	int	4096	[0,...]	中
log.index.size.max.bytes	offset索引的最大字节数	int	10485760	[4,...]	中
log.message.format.version	指定broker用于将消息附加到日志的消息格式版本。应该是一个有效的apiversion值。例如：0.8.2，0.9.0.0，0.10.0，详情去看apiversion。通过设置特定的消息格式版本，用户得保证磁盘上的所有现有消息的版本小于或等于指定的版本。不正确地设置这个值会导致旧版本的用户出错，因为他们将接收到他们无法处理的格式消息。	string	1.0-IV0		中
log.message.timestamp.difference.max.ms	broker收到消息时的时间戳和消息中指定的时间戳之间允许的最大差异。当log.message.timestamp.type=CreateTime,如果时间差超过这个阈值，消息将被拒绝。如果log.message.timestamp.type = logappendtime，则该配置将被忽略。允许的最大时间戳差值，不应大于log.retention.ms，以避免不必要的频繁日志滚动。	long	9223372036854775807		中
log.message.timestamp.type	定义消息中的时间戳是消息创建时间还是日志追加时间。 该值应该是“createtime”或“logappendtime”。	string	CreateTime	[CreateTime, LogAppendTime]	中
log.preallocate	创建新的日志段前是否应该预先分配文件？如果你在windows上使用kafka，你可能需要打开个这个选项	boolean	false		中
log.retention.check.interval.ms	日志清理器检查是否有日志符合删除的频率（以毫秒为单位）	long	300000	[1,...]	中
max.connection		int	2147483647	[1,...]	

s.per.ip	每个IP允许的最大连接数				中
max.connections.per.ip.overrides	每个IP或主机名将覆盖默认的最大连接数	string	""		中
num.partitions	每个topic的默认日志分区数	int	1	[1,...]	中
principal.builder.class	实现kafkaprincipalbuilder接口类的全名，该接口用于构建授权期间使用的kafkaprincipal对象。此配置还支持以前已弃用的用于ssl客户端身份验证的principalbuilder接口。如果未定义主体构建器，则默认采用所使用的安全协议。对于ssl身份验证，如果提供了一个主体名称，主体名称将是客户端证书的专有名称;否则，如果不需要客户端身份验证，则主体名称将是匿名的。对于sasl身份验证，如果使用gssapi，则将使用由 `sasl.kerberos.principal.to.local.rules` 定义的规则来生成主体，而使用其他机制的sasl身份验证ID。若果用明文，委托人将是匿名的。	class	null		中
producer.purgatory.purge.interval.requests	producer请求purgatory的清除间隔（请求数量）	int	1000		中
queued.max.request.bytes	在不再读取请求之前队列的字节数	long	-1		中
replica.fetch.backoff.ms	当拉取分区发生错误时，睡眠的时间。	int	1000	[0,...]	中
replica.fetch.max.bytes	尝试提取每个分区的消息的字节数。这并不是绝对最大值，如果第一个非空分区的第一个批量记录大于这个值，那么批处理仍将被执行并返回，以确保进度可以正常进行下去。broker接受的最大批量记录大小通过 `message.max.bytes` （broker配置）或 `max.message.bytes` （topic配置）进行配置。	int	1048576	[0,...]	medium
replica.fetch.response.max.bytes	预计整个获取响应的最大字节数。记录被批量取回时，如果取第一个非空分区的第一个批量记录大于此值，记录的批处理仍将被执行并返回以确保可以进行下去。因此，这不是绝对的最大值。broker接受的最大批量记录大小通过 `message.max.bytes` （broker配置）或 `max.message.bytes` （topic配置）进行配置。	int	10485760	[0,...]	中
reserved.broker.max.id	可以用于broker.id的最大数量	int	1000	[0,...]	中
sasl.enabled.mechanisms	kafka服务器中启用的sasl机制的列表。 该列表可能包含安全提供程序可用的任何机制。默认情况下只有gssapi是启用的。	list	GSSAPI		中
sasl.kerberos.kinit.cmd	Kerberos kinit 命令路径。	string	/usr/bin/kinit		中
sasl.kerberos.min.time.before.relogin	登录线程在尝试刷新间隔内的休眠时间。	long	60000		中
sasl.kerberos.principal.to.local.rules	主体名称到简称映射的规则列表（通常是操作系统用户名）。按顺序，使用与principal名称匹配的第一个规则将其映射到简称。列表中的任何后续规则都将被忽略。 默认情况下，{username} / {hostname} @ {realm}形式的主体名称映射到{username}。 有关格式的更多细节，请参阅安全授权和acls。 请注意，如果由principal.builder.class配置提供了kafkaprincipalbuilder的扩展，则忽略此配置。	list	DEFAULT		中
sasl.kerberos.service.name	kafka运行的kerberos的主体名称。 这可以在kafka的JAAS配置或在kafka的配置中定义。	string	null		中
sasl.kerberos.ticket.renew.jitter	添加到更新时间的随机抖动的百分比	double	0.05		中
sasl.kerberos.ticket.renew.window.factor	登录线程将休眠，直到从上次刷新到ticket的到期的时间到达（指定窗口因子），在此期间它将尝试更新ticket。	double	0.8		中
sasl.mechanism.inter.broker.protocol	SASL机制，用于broker之间的通讯，默认是GSSAPI。	string	GSSAPI		中
security.inter.broker.protocol	broker之间的安全通讯协议，有效值有：PLAINTEXT, SSL, SASL_PLAINTEXT, SASL_SSL。同时设置此配置和inter.broker.listener.name属性会出错	string	PLAINTEXT		中
ssl.cipher.suites	密码套件列表。 这是一种用于使用tls或ssl网络协议来协商网络连接的安全设置的认证，加密，mac和密钥交换算法的命名组合。 默认情况下，所有可用的密码套件都受支持。	list	null		中
ssl.client.auth	配置请求客户端的broker认证。常见的设置： • `ssl.client.auth=required` 如果设置需要客户端认证。 • `ssl.client.auth=requested` 客户端认证可选，不同于requested，客户端可选择不提供自身的身份验证信息。 • `ssl.client.auth=none` 不需要客户端身份认证。	string	none	[required, requested, none]	中
ssl.enabled.protocols	已启用的SSL连接协议列表。	list	TLSv1.2,TLSv1.1,TLSv1		中
ssl.key.password	秘钥库文件中的私钥密码。对客户端是可选的。	password	null		中
ssl.keymanager.algorithm	用于SSL连接的密钥管理工厂算法。默认值是为Java虚拟机配置的密钥管理器工厂算法。	string	SunX509		中
ssl.keystore.location	密钥仓库文件的位置。客户端可选，并可用于客户端的双向认证。	string	null		中
ssl.keystore.password	密钥仓库文件的仓库密码。客户端可选，只有ssl.keystore.location配置了才需要。	password	null		中
ssl.keystore.type	密钥仓库文件的格式。客户端可选。	string	JKS		中
ssl.protocol	用于生成SSLContext，默认是TLS，适用于大多数情况。允许使用最新的JVM，LS, TLSv1.1 和TLSv1.2。 SSL，SSLv2和SSLv3 老的JVM也可能支持，但由于有已知的安全漏洞，不建议使用。	string	TLS		
ssl.provider	用于SSL连接的安全提供程序的名称。默认值由JVM的安全程序提供。	string	null		中
ssl.trustmanager.algorithm	信任管理工厂用于SSL连接的算法。默认为Java虚拟机配置的信任算法。	string	PKIX		中
ssl.truststore.location	信任文件的存储位置。	string	null		中
ssl.truststore.password	信任存储文件的密码。 如果密码未设置，则仍然可以访问信任库，但完整性检查将被禁用。	password	null		中
ssl.truststore.type	信任存储文件的文件格式。	string	JKS		中
alter.config.policy.class.name	应该用于验证的alter configs策略类。 该类应该实现org.apache.kafka.server.policy.alterconfigpolicy接口。	class	null		低

authorizer.class.name	用于认证授权的程序类	string	""		低
create.topic.policy.class.name	用于验证的创建topic策略类。 该类应该实现org.apache.kafka.server.policy.createtopicpolicy接口。	class	null		低
listener.security.protocol.map	侦听器名称和安全协议之间的映射。必须定义为相同的安全协议可用于多个端口或IP。例如，即使两者都需要ssl，内部和外部流量也可以分开。具体的说，用户可以定义名字为INTERNAL和EXTERNAL的侦听器，这个属性为：internal：ssl，external：ssl。 如图所示，键和值由冒号分隔，映射条目以逗号分隔。 每个监听者名字只能在映射表上出现一次。 通过向配置名称添加规范化前缀（侦听器名称小写），可以为每个侦听器配置不同的安全性（ssl和sasl）设置。 例如，为内部监听器设置不同的密钥仓库，将会设置名称为“listener.name.internal.ssl.keystore.location”的配置。 如果没有设置侦听器名称的配置，配置将回退到通用配置（即`ssl.keystore.location`）。	string	PLAINTEXT:PLAINTEXT,SSL:SSL,SASL_PLAINTEXT:SASL_PLAINTEXT,SASL_SSL:SASL_SSL		低
metric.reporters	度量报告的类列表，通过实现 `MetricReporter` 接口，允许插入新度量标准类。JmxReporter包含注册JVM统计。	list	""		低
metrics.num.samples	维持计算度量的样本数	int	2	[1,...]	低
metrics.recording.level	指标的最高记录级别	string	INFO		低
metrics.sample.window.ms	计算度量样本的时间窗口	long	30000	[1,...]	低
quota.window.num	在内存中保留客户端限额的样本数	int	11	[1,...]	低
quota.window.size.seconds	每个客户端限额的样本时间跨度	int	1	[1,...]	低
replication.quota.window.num	在内存中保留副本限额的样本数	int	11	[1,...]	低
replication.quota.window.size.seconds	每个副本限额样本数的时间跨度	int	1	[1,...]	低
ssl.endpoint.identification.algorithm	端点身份标识算法，使用服务器证书验证服务器主机名	string	null		低
ssl.secure.random.implementation	用于SSL加密操作的SecureRandom PRNG实现	string	null		低
transaction.abort.timed.out.transaction.cleanup.interval.ms	回滚已超时的事务的时间间隔	int	60000	[1,...]	低
transaction.remove.expired.transaction.cleanup.interval.ms	删除由于transactional.id.expiration.ms传递过程而过期的事务的时间间隔	int	3600000	[1,...]	low
zookeeper.sync.time.ms	ZK follower同步可落后leader多久</td>	int	2000		低

More details about broker configuration can be found in the scala class `kafka.server.KafkaConfig`.

3.2 Topic级别配置

与Topic相关的配置既包含服务器默认值，也包含可选的每个Topic覆盖值。 如果没有给出每个Topic的配置，那么服务器默认值就会被使用。 通过提供一个或多个 `--config` 选项，可以在创建Topic时设置覆盖值。 本示例使用自定义的最大消息大小和刷新率创建了一个名为 *my-topic* 的topic:

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

也可以在使用alter configs命令稍后更改或设置覆盖值. 本示例重置*my-topic*的最大消息的大小:

```
> bin/kafka-configs.sh --zookeeper localhost:2181 --entity-type topics --entity-name my-topic
    --alter --add-config max.message.bytes=128000
```

您可以执行如下操作来检查topic设置的覆盖值

```
> bin/kafka-configs.sh --zookeeper localhost:2181 --entity-type topics --entity-name my-topic --describe
```

您可以执行如下操作来删除一个覆盖值

```
> bin/kafka-configs.sh --zookeeper localhost:2181  --entity-type topics --entity-name my-topic --alter --delete-config max.message.bytes
```

以下是Topic级别配置。 “服务器默认属性”列是该属性的默认配置。 一个Topic如果没有给出一个明确的覆盖值，相应的服务器默认配置将会生效。

名称	描述	类型	默认值	有效值	服务器默认属性	重要性
cleanup.policy	该配置项可以是 "delete" 或 "compact"。 它指定在旧日志段上使用的保留策略。 默认策略 ("delete") 将在达到保留时间或大小限制时丢弃旧段。	list	delete	[compact, delete]	log.cleanup.policy	medium

	"compact" 设置将启用该topic的日志压缩 。					
compression.type	为给定的topic指定最终压缩类型。这个配置接受标准的压缩编解码器 ('gzip', 'snappy', lz4) 。它为'uncompressed'时意味着不压缩，当为'producer'时，这意味着保留producer设置的原始压缩编解码器。	string	producer	[uncompressed, snappy, lz4, gzip, producer]	compression.type	medium
delete.retention.ms	保留 日志压缩 topics的删除墓碑标记的时间。此设置还对consumer从偏移量0开始时必须完成读取的时间进行限制，以确保它们获得最后阶段的有效快照(否则，在完成扫描之前可能会收集到删除墓碑)。	long	86400000	[0,...]	log.cleaner.delete.retention.ms	medium
file.delete.delay.ms	删除文件系统上的一个文件之前所需等待的时间。	long	60000	[0,...]	log.segment.delete.delay.ms	medium
flush.messages	这个设置允许指定一个时间间隔n，每隔n个消息我们会强制把数据fsync到log。例如，如果设置为1，我们会在每条消息之后同步。如果是5，我们会在每五个消息之后进行fsync。一般来说，我们建议您不要设置它，而是通过使用replication机制来持久化数据，和允许更高效的操作系统后台刷新功能。这个设置可以针对每个topic的情况自定义 (请参阅 topic的配置部分).	long	9223372036854775807	[0,...]	log.flush.interval.messages	medium
flush.ms	这个设置允许指定一个时间间隔，每隔一段时间我们将强制把数据fsync到log。例如，如果这个设置为1000，我们将在1000 ms后执行fsync。一般来说，我们建议您不要设置它，而是通过使用replication机制来持久化数据，和允许更高效的操作系统后台刷新功能。	long	9223372036854775807	[0,...]	log.flush.interval.ms	medium
follower.replication.throttled.replicas	应该在follower侧限制日志复制的副本列表。该列表应以[PartitionId]：[BrokerId]，[PartitionId]：[BrokerId]：...的形式描述一组副本，或者也可以使用通配符“*”来限制该topic的所有副本。	list	""	[partitionId], [brokerId]: [partitionId], [brokerId]:...	follower.replication.throttled.replicas	medium
index.interval.bytes	此设置控制Kafka向其偏移索引添加索引条目的频率。默认设置确保我们大约每4096个字节索引一条消息。更多的索引允许读取更接近日志中的确切位置，但这会使索引更大。您可能不需要改变该值。	int	4096	[0,...]	log.index.interval.bytes	medium
leader.replication.throttled.replicas	应该在leader侧限制日志复制的副本列表。该列表应以[PartitionId]：[BrokerId]，[PartitionId]：[BrokerId]：...的形式描述一组副本，或者也可以使用通配符“*”来限制该topic的所有副本。	list	""	[partitionId], [brokerId]: [partitionId], [brokerId]:...	leader.replication.throttled.replicas	medium
max.message.bytes	Kafka允许的最大记录批次大小。如果这个参数被增加了且consumers是早于0.10.2版本，那么consumers的fetch size必须增加到该值，以便他们可以取得这么大的记录批次。 在最新的消息格式版本中，记录总是分组成多个批次以提高效率。在以前的消息格式版本中，未压缩的记录不会分组到多个批次，并且限制在该情况下只能应用单条记录。	int	1000012	[0,...]	message.max.bytes	medium
message.format.version	指定broker将用于将消息附加到日志的消息格式版本。该值应该是有效的ApiVersion。如：0.8.2，0.9.0.0，0.10.0，查看ApiVersion获取更多细节。通过设置特定的消息格式版本，用户将发现磁盘上的所有现有消息都小于或等于指定的版本。不正确地设置此值将导致旧版本的使用者中断，因为他们将收到他们不理解的格式的消息。	string	1.0-IV0		log.message.format.version	medium
message.timestamp.difference.max.ms	broker接收消息时所允许的时间戳与消息中指定的时间戳之间的最大差异。如果message.timestamp.type=CreateTime，则如果时间戳的差异超过此阈值，则将拒绝消息。如果message.timestamp.type=LogAppendTime，则忽略此配置。	long	9223372036854775807	[0,...]	log.message.timestamp.difference.max.ms	medium
message.timestamp.type	定义消息中的时间戳是消息创建时间还是日志附加时间。值应该是“CreateTime”或“LogAppendTime”	string	CreateTime		log.message.timestamp.type	medium
min.cleanable.dirty.ratio	此配置控制日志compaction程序尝试清理日志的频率(假设启用了log compaction)。默认情况下，我们将避免清除超过50%的日志已经合并的日志。这个比率限制了重复在日志中浪费的最大空间(最多为50%，日志中最多有50%可能是重复的)。一个更高的比率将意味着更少，更高效的清理，但将意味着在日志中浪费更多的空间。	double	0.5	[0,...,1]	log.cleaner.min.cleanable.ratio	medium
min.compaction.lag.ms	消息在日志中保持未压缩的最短时间。仅适用于被合并的日志。	long	0	[0,...]	log.cleaner.min.compaction.lag.ms	medium
min.insync.replicas	当producer将ack设置为“all”(或“-1”)时，此配置指定必须确认写入才能被认为成功的副本的最小数量。如果这个最小值无法满足，那么producer将引发一个异常(NotEnough Replicas或NotEnough ReplicasAfterAppend)。 当使用时，min.insync.Copicas和ack允许您执行更好的持久化保证。一个典型的场景是创建一个复制因子为3的topic，将min.insync.Copicas设置为2，并生成带有“All”的ack。这将确保如果大多数副本没有接收到写，则producer将引发异常。	int	1	[1,...]	min.insync.replicas	medium
preallocate	如果在创建新的日志段时应该预先分配磁盘上的文件，则为True。	boolean	false		log.preallocate	medium
retention.bytes	如果使用“delete”保留策略，此配置控制分区(由日志段组成)在放弃旧日志段以释放空间之前的最大大小。默认情况下，没有大小限制，只有时间限制。由于此限制是在分区级别强制执行的，因此，将其乘以分区数，计算出topic保留值，以字节为单位。	long	-1		log.retention.bytes	medium
retention.ms	如果使用“delete”保留策略，此配置控制保留日志的最长时间，然后将旧日志段丢弃以释放空间。这代表了用户读取数据的速度的SLA。	long	604800000		log.retention.ms	medium
segment.bytes	此配置控制日志的段文件大小。保留和清理总是一次完成一个文件，所以更大的段大小意味着更少的文件，但对保留的粒度控制更少。	int	1073741824	[14,...]	log.segment.bytes	medium
segment.index.bytes	此配置控制将偏移量映射到文件位置的索引大小。我们预先分配这个索引文件并且只在日志滚动后收缩它。您通常不需要更改此设置。	int	10485760	[0,...]	log.index.size.max.bytes	medium
segment.jitter.ms	从预定的分段滚动时间减去最大随机抖动，以避免段滚动产生惊群效应。	long	0	[0,...]	log.roll.jitter.ms	medium
segment.ms	这个配置控制在一段时间后，Kafka将强制日志滚动，即使段文件没有满，以确保保留空间可以删除或合并旧数据。	long	604800000	[0,...]	log.roll.ms	medium
unclean.leader.election.enable	指示是否启用不在ISR集合中的副本选为领导者作为最后的手段，即使这样做可能导致数据丢失。	boolean	false		unclean.leader.election.enable	medium

3.3 Producer 配置

以下是JAVA生产者的配置：

NAME	DESCRIPTION	TYPE	DEFAULT	VALID VALUES	IMPORTANCE
bootstrap.servers	这是一个用于建立初始连接到kafka集群的"主机/端口对"配置列表。不论这个参数配置了哪些服务器来初始化连接，客户端都是会均衡地与集群中的所有服务器建立连接。一配置的服务器清单仅用于初始化连接，以便找到集群中的所有服务器。配置格式：`host1:port1,host2:port2,...`。由于这些主机是用于初始化连接，以获得整个集群（集群是会动态变化的），因此这个配置清单不需要包含整个集群的服务器。（当然，为了避免单节点风险，这个清单最好配置多台主机）。	list			high
key.serializer	关键字的序列化类，实现以下接口：`org.apache.kafka.common.serialization.Serializer` 接口。	class			high
value.serializer	值的序列化类，实现以下接口：`org.apache.kafka.common.serialization.Serializer` 接口。	class			high
acks	此配置是 Producer 在确认一个请求发送完成之前需要收到的反馈信息的数量。 这个参数是为了保证发送请求的可靠性。以下配置方式是允许的： • `acks=0` 如果设置为0，则 producer 不会等待服务器的反馈。该消息会被立刻添加到 socket buffer 中并认为已经发送完成。在这种情况下，服务器是否收到请求是没法保证的，并且参数 `retries` 也不会生效（因为客户端无法获得失败信息）。每个记录返回的 offset 总是被设置为-1。 • `acks=1` 如果设置为1，leader节点会将记录写入本地日志，并且在所有 follower 节点反馈之前就先确认成功。在这种情况下，如果 leader 节点在接收记录之后，并且在 follower 节点复制数据完成之前产生错误，则这条记录会丢失。 • `acks=all` 如果设置为all，这就意味着 leader 节点会等待所有同步中的副本确认之后再确认这条记录是否发送完成。只要至少有一个同步副本存在，记录就不会丢失。这种方式是对请求传递的最有效保证。acks=-1与acks=all是等效的。	string	1	[all, -1, 0, 1]	high
buffer.memory	Producer 用来缓冲等待被发送到服务器的记录的总字节数。如果记录发送的速度比发送到服务器的速度快， Producer 就会阻塞，如果阻塞的时间超过 `max.block.ms` 配置的时长，则会抛出一个异常。 这个配置与 Producer 的可用总内存有一定的对应关系，但并不是完全等价的关系，因为 Producer 的可用内存并不是全部都用来缓存。一些额外的内存可能会用于压缩(如果启用了压缩)，以及维护正在运行的请求。	long	33554432	[0,...]	high
compression.type	Producer 生成数据时可使用的压缩类型。默认值是none(即不压缩)。可配置的压缩类型包括：`none`， `gzip`， `snappy`，或者 `lz4` 。压缩是针对批处理的所有数据，所以批处理的效果也会影响压缩比(更多的批处理意味着更好的压缩)。	string	none		high
retries	若设置大于0的值，则客户端会将发送失败的记录重新发送，尽管这些记录有可能是暂时性的错误。请注意，这种 retry 与客户端收到错误信息之后重新发送记录并无区别。允许 retries 并且没有设置 `max.in.flight.requests.per.connection` 为1时，记录的顺序可能会被改变。比如：当两个批次都被发送到同一个 partition ，第一个批次发生错误并发生 retries 而第二个批次已经成功，则第二个批次的记录就会先于第一个批次出现。	int	0	[0,...,2147483647]	high
ssl.key.password	key store 文件中私钥的密码。这对于客户端来说是可选的。	password	null		high
ssl.keystore.location	key store 文件的位置。这对于客户端来说是可选的，可用于客户端的双向身份验证。	string	null		high
ssl.keystore.password	key store 文件的密码。这对于客户端是可选的，只有配置了 ssl.keystore.location 才需要配置该选项。	password	null		high
ssl.truststore.location	trust store 文件的位置。	string	null		high
ssl.truststore.password	trust store 文件的密码。如果一个密码没有设置到 trust store ， 这个密码仍然是可用的，但是完整性检查是禁用的。	password	null		high
batch.size	当将多个记录被发送到同一个分区时， Producer 将尝试将记录组合到更少的请求中。这有助于提升客户端和服务器端的性能。这个配置控制一个批次的默认大小（以字节为单位）。 当记录的大小超过了配置的字节数， Producer 将不再尝试往批次增加记录。 发送到 broker 的请求会包含多个批次的数据，每个批次对应一个 partition 的可用数据 小的 batch.size 将减少批处理，并且可能会降低吞吐量(如果 batch.size = 0的话将完全禁用批处理)。 很大的 batch.size 可能造成内存浪费，因为我们一般会在 batch.size 的基础上分配一部分缓存以应付额外的记录。	int	16384	[0,...]	medium
client.id	发出请求时传递给服务器的 ID 字符串。这样做的目的是为了在服务端的请求日志中能够通过逻辑应用名称来跟踪请求的来源，而不是只能通过IP和端口号跟进。	string	""		medium
connections.max.idle.ms	在此配置指定的毫秒数之后，关闭空闲连接。	long	540000		medium
linger.ms	producer 会将两个请求发送时间间隔内到达的记录合并到一个单独的批处理请求中。通常只有当记录到达的速度超过了发送的速度时才会出现这种情况。然而，在某些场景下，即使处于可接受的负载下，客户端也希望能减少请求的数量。这个设置是通过添加少量的人为延迟来实现的 — 即，与其立即发送记录， producer 将等待给定的延迟时间，以便将在等待过程中到达的其他记录能合并到本批次的处理中。这可以认为是与 TCP 中的 Nagle 算法类似。这个设置为批处理的延迟提供了上限:一旦我们接受到记录超过了分区的 batch.size ， Producer 会忽略这个参数，立刻发送数据。但是如果累积的字节数少于 batch.size ， 那么我们将在指定的时间内"逗留"(linger)，以等待更多的记录出现。这个设置默认为0(即没有延迟)。例如：如果设置 `linger.ms=5` ，则发送的请求会减少并降低部分负载，但同时会增加5毫秒的延迟。	long	0	[0,...]	medium
max.block.ms	该配置控制 `KafkaProducer.send()` 和 `KafkaProducer.partitionsFor()` 允许被阻塞的时长。这些方法可能因为缓冲区满了或者元数据不可用而被阻塞。用户提供的序列化程序或分区程序的阻塞将不会被计算到这个超时。	long	60000	[0,...]	medium
max.request.size	请求的最大字节数。这个设置将限制 Producer 在单个请求中发送的记录批量的数量，以避免发送巨大的请求。这实际上也等同于批次的最大记录数的限制。请注意，服务器对批次的大小有自己的限制，这可能与此不同。	int	1048576	[0,...]	medium
partitioner.class	指定计算分区的类，实现 `org.apache.kafka.clients.producer.Partitioner` 接口。	class	org.apache.kafka.clients.producer.internals.DefaultPartitioner		medium
receive.buffer.bytes	定义读取数据时 TCP 接收缓冲区（SO_RCVBUF）的大小，如果设置为-1，则使用系统默认值。	int	32768	[-1,...]	medium

request.timeout.ms	客户端等待请求响应的最大时长。如果超时未收到响应，则客户端将在必要时重新发送请求，如果重试的次数达到允许的最大重试次数，则请求失败。这个参数应该比 replica.lag.time.max.ms（Broker 的一个参数）更大，以降低由于不必要的重试而导致的消息重复的可能性。	int	30000	[0,...]	medium
sasl.jaas.config	SASL 连接使用的 JAAS 登陆上下文参数，以 JAAS 配置文件的格式进行配置。 JAAS 配置文件格式可参考这里。值的格式: ' (=)*;'	password	null		medium
sasl.kerberos.service.name	Kafka 运行时的 Kerberos 主体名称。可以在 Kafka 的 JAAS 配置文件或者 Kafka 的配置文件中配置。	string	null		medium
sasl.mechanism	用于客户端连接的 SASL 机制。可以是任意安全可靠的机制。默认是 GSSAPI 机制。	string	GSSAPI		medium
security.protocol	与 brokers 通讯的协议。可配置的值有: PLAINTEXT, SSL, SASL_PLAINTEXT, SASL_SSL.	string	PLAINTEXT		medium
send.buffer.bytes	定义发送数据时的 TCP 发送缓冲区（SO_SNDBUF）的大小。如果设置为-1，则使用系统默认值。	int	131072	[-1,...]	medium
ssl.enabled.protocols	可用于 SSL 连接的协议列表。	list	TLSv1.2,TLSv1.1,TLSv1		medium
ssl.keystore.type	key store 文件的文件格类型。这对于客户端来说是可选的。	string	JKS		medium
ssl.protocol	用于生成SSLContext的SSL协议。默认设置是TLS，大多数情况下不会有问题。在最近的jvm版本中，允许的值是TLS、tlsv1.1和TLSv1.2。在旧的jvm中可能会支持SSL、SSLv2和SSLv3，但是由于存在已知的安全漏洞，因此不建议使用。	string	TLS		medium
ssl.provider	用于 SSL 连接security provider 。默认值是当前 JVM 版本的默认 security provider 。	string	null		medium
ssl.truststore.type	trust store 的文件类型。	string	JKS		medium
enable.idempotence	当设置为true时， Producer 将确保每个消息在 Stream 中只写入一个副本。如果为false，由于 Broker 故障导致 Producer 进行重试之类的情况可能会导致消息重复写入到 Stream 中。请注意，启用幂等性需要确保 `max.in.flight.requests.per.connection` 小于或等于5，`retries` 大于等于0，并且 `ack` 必须设置为all 。如果这些值不是由用户明确设置的，那么将自动选择合适的值。如果设置了不兼容的值，则将抛出一个ConfigException的异常。	boolean	false		low
interceptor.classes	配置 interceptor 类的列表。实现 `org.apache.kafka.clients.producer.ProducerInterceptor` 接口之后可以拦截(并可能改变)那些 Producer 还没有发送到 kafka 集群的记录。默认情况下，没有 interceptor 。	list	null		low
max.in.flight.requests.per.connection	在发生阻塞之前，客户端的一个连接上允许出现未确认请求的最大数量。注意，如果这个设置大于1，并且有失败的发送，则消息可能会由于重试而导致重新排序(如果重试是启用的话)。	int	5	[1,...]	low
metadata.max.age.ms	刷新元数据的时间间隔，单位毫秒。即使没有发现任何分区的 leadership 发生变更也会强制刷新以便能主动发现新的 Broker 或者新的分区。	long	300000	[0,...]	low
metric.reporters	用于指标监控报表的类清单。实现 `org.apache.kafka.common.metrics.MetricsReporter` 接口之后允许插入能够通知新的创建度量的类。JmxReporter 总是包含在注册的 JMX 统计信息中。	list	""		low
metrics.num.samples	计算 metrics 所需要维持的样本数量。	int	2	[1,...]	low
metrics.recording.level	metrics 的最高纪录级别。	string	INFO	[INFO, DEBUG]	low
metrics.sample.window.ms	计算 metrics 样本的时间窗口。	long	30000	[0,...]	low
reconnect.backoff.max.ms	当重新连接到一台多次连接失败的 Broker 时允许等待的最大毫秒数。如果配置该参数，则每台主机的 backoff 将呈指数级增长直到达到配置的最大值。当统计到 backoff 在增长，系统会增加20%的随机波动以避免大量的连接失败。	long	1000	[0,...]	low
reconnect.backoff.ms	在尝试重新连接到给定的主机之前，需要等待的基本时间。这避免了在一个紧凑的循环中反复连接到同一个主机。这个 backoff 机制应用于所有客户端尝试连接到 Broker 的请求。	long	50	[0,...]	low
retry.backoff.ms	在尝试将一个失败的请求重试到给定的 topic 分区之前需要等待的时间。这避免在某些失败场景下在紧凑的循环中重复发送请求。	long	100	[0,...]	low
sasl.kerberos.kinit.cmd	Kerberos kinit 命令的路径。	string	/usr/bin/kinit		low
sasl.kerberos.min.time.before.relogin	重新尝试登陆之前,登录线程的休眠时间。	long	60000		low
sasl.kerberos.ticket.renew.jitter	随机抖动增加到更新时间的百分比。	double	0.05		low
sasl.kerberos.ticket.renew.window.factor	登录线程将持续休眠直到上一次刷新到 ticket 的过期时间窗口，在此时间窗口它将尝试更新 ticket 。	double	0.8		low
ssl.cipher.suites	密码套件列表。密码套件是利用 TLS 或 SSL 网络协议来实现网络连接的安全设置，是一个涵盖认证，加密，MAC和密钥交换算法的组合。默认情况下，支持所有可用的密码套件。	list	null		low
ssl.endpoint.identification.algorithm	使用服务器证书验证服务器主机名的 endpoint 识别算法。	string	null		low
ssl.keymanager.algorithm	key manager factory 用于 SSL 连接的算法。默认值是Java虚拟机配置的 key manager factory 算法。	string	SunX509		low
ssl.secure.random.implementation	用于 SSL 加密操作的 SecureRandom PRNG 实现。	string	null		low
ssl.trustmanager.algorithm	trust manager factory 用于SSL连接的算法。默认值是Java虚拟机配置的 trust manager factory 算法。	string	PKIX		low
transaction.timeout.ms	主动中止进行中的事务之前，事务协调器等待 Producer 更新事务状态的最长时间（以毫秒为单位）。如果此值大于 Broker 中的 max.transaction.timeout.ms 设置的时长，则请求将失败并提示"InvalidTransactionTimeout"错误。	int	60000		low
transactional.id	用于事务交付的 TransactionalId。 这使跨越多个生产者会话的可靠性语义成为可能，因为它可以保证客户在开始任何新的事务之前，使用相同的 TransactionalId 的事务都已经完成。 如果没有提供 TransactionalId ，则 Producer 被限制为幂等递送。 请注意，如果配置了 TransactionalId，则必须启用 enable.idempotence 。 缺省值为空，这意味着无法使用事务。	string	null	non-empty string	low

如果对老的Scala版本的 Producer 配置感兴趣，请点击 这里.

3.4 Consumer Configs

In 0.9.0.0 we introduced the new Java consumer as a replacement for the older Scala-based simple and high-level consumers. The configs for both new and old consumers are described below.

3.4.1 New Consumer Configs

Below is the configuration for the new consumer:

NAME	DESCRIPTION	TYPE	DEFAULT	VALID VALUES	IMPORTANCE
bootstrap.servers	A list of host/port pairs to use for establishing the initial connection to the Kafka cluster. The client will make use of all servers irrespective of which servers are specified here for bootstrapping—this list only impacts the initial hosts used to discover the full set of servers. This list should be in the form `host1:port1,host2:port2,...`. Since these servers are just used for the initial connection to discover the full cluster membership (which may change dynamically), this list need not contain the full set of servers (you may want more than one, though, in case a server is down).	list			high
key.deserializer	Deserializer class for key that implements the `org.apache.kafka.common.serialization.Deserializer` interface.	class			high
value.deserializer	Deserializer class for value that implements the `org.apache.kafka.common.serialization.Deserializer` interface.	class			high
fetch.min.bytes	The minimum amount of data the server should return for a fetch request. If insufficient data is available the request will wait for that much data to accumulate before answering the request. The default setting of 1 byte means that fetch requests are answered as soon as a single byte of data is available or the fetch request times out waiting for data to arrive. Setting this to something greater than 1 will cause the server to wait for larger amounts of data to accumulate which can improve server throughput a bit at the cost of some additional latency.	int	1	[0,...]	high
group.id	A unique string that identifies the consumer group this consumer belongs to. This property is required if the consumer uses either the group management functionality by using `subscribe(topic)` or the Kafka-based offset management strategy.	string	""		high
heartbeat.interval.ms	The expected time between heartbeats to the consumer coordinator when using Kafka's group management facilities. Heartbeats are used to ensure that the consumer's session stays active and to facilitate rebalancing when new consumers join or leave the group. The value must be set lower than `session.timeout.ms`, but typically should be set no higher than 1/3 of that value. It can be adjusted even lower to control the expected time for normal rebalances.	int	3000		high
max.partition.fetch.bytes	The maximum amount of data per-partition the server will return. Records are fetched in batches by the consumer. If the first record batch in the first non-empty partition of the fetch is larger than this limit, the batch will still be returned to ensure that the consumer can make progress. The maximum record batch size accepted by the broker is defined via `message.max.bytes` (broker config) or `max.message.bytes` (topic config). See fetch.max.bytes for limiting the consumer request size.	int	1048576	[0,...]	high
session.timeout.ms	The timeout used to detect consumer failures when using Kafka's group management facility. The consumer sends periodic heartbeats to indicate its liveness to the broker. If no heartbeats are received by the broker before the expiration of this session timeout, then the broker will remove this consumer from the group and initiate a rebalance. Note that the value must be in the allowable range as configured in the broker configuration by `group.min.session.timeout.ms` and `group.max.session.timeout.ms`.	int	10000		high
ssl.key.password	The password of the private key in the key store file. This is optional for client.	password	null		high
ssl.keystore.location	The location of the key store file. This is optional for client and can be used for two-way authentication for client.	string	null		high
ssl.keystore.password	The store password for the key store file. This is optional for client and only needed if ssl.keystore.location is configured.	password	null		high
ssl.truststore.location	The location of the trust store file.	string	null		high
ssl.truststore.password	The password for the trust store file. If a password is not set access to the truststore is still available, but integrity checking is disabled.	password	null		high
auto.offset.reset	What to do when there is no initial offset in Kafka or if the current offset does not exist any more on the server (e.g. because that data has been deleted): • earliest: automatically reset the offset to the earliest offset • latest: automatically reset the offset to the latest offset • none: throw exception to the consumer if no previous offset is found for the consumer's group • anything else: throw exception to the consumer.	string	latest	[latest, earliest, none]	medium
connections.max.idle.ms	Close idle connections after the number of milliseconds specified by this config.	long	540000		medium
enable.auto.commit	If true the consumer's offset will be periodically committed in the background.	boolean	true		medium
exclude.internal.topics	Whether records from internal topics (such as offsets) should be exposed to the consumer. If set to `true` the only way to receive records from an internal topic is subscribing to it.	boolean	true		medium
fetch.max.bytes	The maximum amount of data the server should return for a fetch request. Records are fetched in batches by the consumer, and if the first record batch in the first non-empty partition of the fetch is larger than this value, the record batch will still be returned to ensure that the consumer can make progress. As such, this is not a absolute maximum. The maximum record batch size accepted by the broker is defined via `message.max.bytes` (broker config) or `max.message.bytes` (topic config). Note that the consumer performs multiple fetches in parallel.	int	52428800	[0,...]	medium
isolation.level	Controls how to read messages written transactionally. If set to `read_committed`, consumer.poll() will only return transactional messages which have been committed. If set to `read_uncommitted` ' (the default), consumer.poll() will return all messages, even transactional messages which have been aborted. Non-transactional messages will be returned unconditionally in either mode. Messages will always be returned in offset order. Hence, in `read_committed` mode, consumer.poll() will only return messages up to the last stable offset (LSO), which is the one less than the offset of the first open transaction. In particular any messages appearing after	string	read_uncommitted	[read_committed, read_uncommitted]	medium

	messages belonging to ongoing transactions will be withheld until the relevant transaction has been completed. As a result, `read_committed` consumers will not be able to read up to the high watermark when there are in flight transactions. Further, when in `read_committed the seekToEnd method will return the LSO`				
max.poll.interval.ms	The maximum delay between invocations of poll() when using consumer group management. This places an upper bound on the amount of time that the consumer can be idle before fetching more records. If poll() is not called before expiration of this timeout, then the consumer is considered failed and the group will rebalance in order to reassign the partitions to another member.	int	300000	[1,...]	medium
max.poll.records	The maximum number of records returned in a single call to poll().	int	500	[1,...]	medium
partition.assignment.strategy	The class name of the partition assignment strategy that the client will use to distribute partition ownership amongst consumer instances when group management is used	list	class org.apache.kafka.clients.consumer.RangeAssignor		medium
receive.buffer.bytes	The size of the TCP receive buffer (SO_RCVBUF) to use when reading data. If the value is -1, the OS default will be used.	int	65536	[-1,...]	medium
request.timeout.ms	The configuration controls the maximum amount of time the client will wait for the response of a request. If the response is not received before the timeout elapses the client will resend the request if necessary or fail the request if retries are exhausted.	int	305000	[0,...]	medium
sasl.jaas.config	JAAS login context parameters for SASL connections in the format used by JAAS configuration files. JAAS configuration file format is described here. The format for the value is: ' (=)*;'	password	null		medium
sasl.kerberos.service.name	The Kerberos principal name that Kafka runs as. This can be defined either in Kafka's JAAS config or in Kafka's config.	string	null		medium
sasl.mechanism	SASL mechanism used for client connections. This may be any mechanism for which a security provider is available. GSSAPI is the default mechanism.	string	GSSAPI		medium
security.protocol	Protocol used to communicate with brokers. Valid values are: PLAINTEXT, SSL, SASL_PLAINTEXT, SASL_SSL.	string	PLAINTEXT		medium
send.buffer.bytes	The size of the TCP send buffer (SO_SNDBUF) to use when sending data. If the value is -1, the OS default will be used.	int	131072	[-1,...]	medium
ssl.enabled.protocols	The list of protocols enabled for SSL connections.	list	TLSv1.2,TLSv1.1,TLSv1		medium
ssl.keystore.type	The file format of the key store file. This is optional for client.	string	JKS		medium
ssl.protocol	The SSL protocol used to generate the SSLContext. Default setting is TLS, which is fine for most cases. Allowed values in recent JVMs are TLS, TLSv1.1 and TLSv1.2. SSL, SSLv2 and SSLv3 may be supported in older JVMs, but their usage is discouraged due to known security vulnerabilities.	string	TLS		medium
ssl.provider	The name of the security provider used for SSL connections. Default value is the default security provider of the JVM.	string	null		medium
ssl.truststore.type	The file format of the trust store file.	string	JKS		medium
auto.commit.interval.ms	The frequency in milliseconds that the consumer offsets are auto-committed to Kafka if `enable.auto.commit` is set to `true`.	int	5000	[0,...]	low
check.crcs	Automatically check the CRC32 of the records consumed. This ensures no on-the-wire or on-disk corruption to the messages occurred. This check adds some overhead, so it may be disabled in cases seeking extreme performance.	boolean	true		low
client.id	An id string to pass to the server when making requests. The purpose of this is to be able to track the source of requests beyond just ip/port by allowing a logical application name to be included in server-side request logging.	string	""		low
fetch.max.wait.ms	The maximum amount of time the server will block before answering the fetch request if there isn't sufficient data to immediately satisfy the requirement given by fetch.min.bytes.	int	500	[0,...]	low
interceptor.classes	A list of classes to use as interceptors. Implementing the `org.apache.kafka.clients.consumer.ConsumerInterceptor` interface allows you to intercept (and possibly mutate) records received by the consumer. By default, there are no interceptors.	list	null		low
metadata.max.age.ms	The period of time in milliseconds after which we force a refresh of metadata even if we haven't seen any partition leadership changes to proactively discover any new brokers or partitions.	long	300000	[0,...]	low
metric.reporters	A list of classes to use as metrics reporters. Implementing the `org.apache.kafka.common.metrics.MetricsReporter` interface allows plugging in classes that will be notified of new metric creation. The JmxReporter is always included to register JMX statistics.	list	""		low
metrics.num.samples	The number of samples maintained to compute metrics.	int	2	[1,...]	low
metrics.recording.level	The highest recording level for metrics.	string	INFO	[INFO, DEBUG]	low
metrics.sample.window.ms	The window of time a metrics sample is computed over.	long	30000	[0,...]	low
reconnect.backoff.max.ms	The maximum amount of time in milliseconds to wait when reconnecting to a broker that has repeatedly failed to connect. If provided, the backoff per host will increase exponentially for each consecutive connection failure, up to this maximum. After calculating the backoff increase, 20% random jitter is added to avoid connection storms.	long	1000	[0,...]	low
reconnect.backoff.ms	The base amount of time to wait before attempting to reconnect to a given host. This avoids repeatedly connecting to a host in a tight loop. This backoff applies to all connection attempts by the client to a broker.	long	50	[0,...]	low
retry.backoff.ms	The amount of time to wait before attempting to retry a failed request to a given topic partition. This avoids repeatedly sending requests in a tight loop under some failure scenarios.	long	100	[0,...]	low
sasl.kerberos.kinit.cmd	Kerberos kinit command path.	string	/usr/bin/kinit		low
sasl.kerberos.min.time.before.relogin	Login thread sleep time between refresh attempts.	long	60000		low
sasl.kerberos.ticket.renew.jitter	Percentage of random jitter added to the renewal time.	double	0.05		low
sasl.kerberos.ticket.renew.window.factor	Login thread will sleep until the specified window factor of time from last refresh to ticket's expiry has been reached, at which time it will try to renew the ticket.	double	0.8		low
ssl.cipher.suites	A list of cipher suites. This is a named combination of authentication, encryption, MAC and key exchange algorithm used to negotiate the security settings for a network connection using TLS or SSL network protocol. By default all the available cipher suites are supported.	list	null		low

ssl.endpoint.identification.algorithm	The endpoint identification algorithm to validate server hostname using server certificate.	string	null		low
ssl.keymanager.algorithm	The algorithm used by key manager factory for SSL connections. Default value is the key manager factory algorithm configured for the Java Virtual Machine.	string	SunX509		low
ssl.secure.random.implementation	The SecureRandom PRNG implementation to use for SSL cryptography operations.	string	null		low
ssl.trustmanager.algorithm	The algorithm used by trust manager factory for SSL connections. Default value is the trust manager factory algorithm configured for the Java Virtual Machine.	string	PKIX		low

3.4.2 Old Consumer Configs

The essential old consumer configurations are the following:

- `group.id`
- `zookeeper.connect`

PROPERTY	DEFAULT	DESCRIPTION
group.id		A string that uniquely identifies the group of consumer processes to which this consumer belongs. By setting the same group id multiple processes indicate that they are all part of the same consumer group.
zookeeper.connect		Specifies the ZooKeeper connection string in the form `hostname:port` where host and port are the host and port of a ZooKeeper server. To allow connecting through other ZooKeeper nodes when that ZooKeeper machine is down you can also specify multiple hosts in the form `hostname1:port1,hostname2:port2,hostname3:port3` . The server may also have a ZooKeeper chroot path as part of its ZooKeeper connection string which puts its data under some path in the global ZooKeeper namespace. If so the consumer should use the same chroot path in its connection string. For example to give a chroot path of `/chroot/path` you would give the connection string as `hostname1:port1,hostname2:port2,hostname3:port3/chroot/path` .
consumer.id	null	Generated automatically if not set.
socket.timeout.ms	30 * 1000	The socket timeout for network requests. The actual timeout set will be max.fetch.wait + socket.timeout.ms.
socket.receive.buffer.bytes	64 * 1024	The socket receive buffer for network requests
fetch.message.max.bytes	1024 * 1024	The number of bytes of messages to attempt to fetch for each topic-partition in each fetch request. These bytes will be read into memory for each partition, so this helps control the memory used by the consumer. The fetch request size must be at least as large as the maximum message size the server allows or else it is possible for the producer to send messages larger than the consumer can fetch.
num.consumer.fetchers	1	The number fetcher threads used to fetch data.
auto.commit.enable	true	If true, periodically commit to ZooKeeper the offset of messages already fetched by the consumer. This committed offset will be used when the process fails as the position from which the new consumer will begin.
auto.commit.interval.ms	60 * 1000	The frequency in ms that the consumer offsets are committed to zookeeper.
queued.max.message.chunks	2	Max number of message chunks buffered for consumption. Each chunk can be up to fetch.message.max.bytes.
rebalance.max.retries	4	When a new consumer joins a consumer group the set of consumers attempt to "rebalance" the load to assign partitions to each consumer. If the set of consumers changes while this assignment is taking place the rebalance will fail and retry. This setting controls the maximum number of attempts before giving up.
fetch.min.bytes	1	The minimum amount of data the server should return for a fetch request. If insufficient data is available the request will wait for that much data to accumulate before answering the request.
fetch.wait.max.ms	100	The maximum amount of time the server will block before answering the fetch request if there isn't sufficient data to immediately satisfy fetch.min.bytes
rebalance.backoff.ms	2000	Backoff time between retries during rebalance. If not set explicitly, the value in zookeeper.sync.time.ms is used.
refresh.leader.backoff.ms	200	Backoff time to wait before trying to determine the leader of a partition that has just lost its leader.
auto.offset.reset	largest	What to do when there is no initial offset in ZooKeeper or if an offset is out of range: * smallest : automatically reset the offset to the smallest offset * largest : automatically reset the offset to the largest offset * anything else: throw exception to the consumer
consumer.timeout.ms	-1	Throw a timeout exception to the consumer if no message is available for consumption after the specified interval
exclude.internal.topics	true	Whether messages from internal topics (such as offsets) should be exposed to the consumer.
client.id	group id value	The client id is a user-specified string sent in each request to help trace calls. It should logically identify the application making the request.
zookeeper.session.timeout.ms	6000	ZooKeeper session timeout. If the consumer fails to heartbeat to ZooKeeper for this period of time it is considered dead and a rebalance will occur.
zookeeper.connection.timeout.ms	6000	The max time that the client waits while establishing a connection to zookeeper.
zookeeper.sync.time.ms	2000	How far a ZK follower can be behind a ZK leader
offsets.storage	zookeeper	Select where offsets should be stored (zookeeper or kafka).
offsets.channel.backoff.ms	1000	The backoff period when reconnecting the offsets channel or retrying failed offset fetch/commit requests.
offsets.channel.socket.timeout.ms	10000	Socket timeout when reading responses for offset fetch/commit requests. This timeout is also used for ConsumerMetadata requests that are used to query for the offset manager.
offsets.commit.max.retries	5	Retry the offset commit up to this many times on failure. This retry count only applies to offset commits during shut-down. It does not apply to commits originating from the auto-commit thread. It also does not apply to attempts to query for the offset coordinator before committing offsets. i.e., if a consumer metadata request fails for any reason, it will be retried and that retry does not count toward this limit.
dual.commit.enabled	true	If you are using "kafka" as offsets.storage, you can dual commit offsets to ZooKeeper (in addition to Kafka). This is required during migration from zookeeper-based offset storage to kafka-based offset storage. With respect to any given consumer group, it is safe to turn this off after all instances within that group

		have been migrated to the new version that commits offsets to the broker (instead of directly to ZooKeeper).
partition.assignment.strategy	range	Select between the "range" or "roundrobin" strategy for assigning partitions to consumer streams. The round-robin partition assignor lays out all the available partitions and all the available consumer threads. It then proceeds to do a round-robin assignment from partition to consumer thread. If the subscriptions of all consumer instances are identical, then the partitions will be uniformly distributed. (i.e., the partition ownership counts will be within a delta of exactly one across all consumer threads.) Round-robin assignment is permitted only if: (a) Every topic has the same number of streams within a consumer instance (b) The set of subscribed topics is identical for every consumer instance within the group. Range partitioning works on a per-topic basis. For each topic, we lay out the available partitions in numeric order and the consumer threads in lexicographic order. We then divide the number of partitions by the total number of consumer streams (threads) to determine the number of partitions to assign to each consumer. If it does not evenly divide, then the first few consumers will have one extra partition.

More details about consumer configuration can be found in the scala class `kafka.consumer.ConsumerConfig`.

3.5 Kafka Connect Configs

Below is the configuration of the Kafka Connect framework.

NAME	DESCRIPTION	TYPE	DEFAULT	VALID VALUES	IMPORTANCE
config.storage.topic	The name of the Kafka topic where connector configurations are stored	string			high
group.id	A unique string that identifies the Connect cluster group this worker belongs to.	string			high
key.converter	Converter class used to convert between Kafka Connect format and the serialized form that is written to Kafka. This controls the format of the keys in messages written to or read from Kafka, and since this is independent of connectors it allows any connector to work with any serialization format. Examples of common formats include JSON and Avro.	class			high
offset.storage.topic	The name of the Kafka topic where connector offsets are stored	string			high
status.storage.topic	The name of the Kafka topic where connector and task status are stored	string			high
value.converter	Converter class used to convert between Kafka Connect format and the serialized form that is written to Kafka. This controls the format of the values in messages written to or read from Kafka, and since this is independent of connectors it allows any connector to work with any serialization format. Examples of common formats include JSON and Avro.	class			high
internal.key.converter	Converter class used to convert between Kafka Connect format and the serialized form that is written to Kafka. This controls the format of the keys in messages written to or read from Kafka, and since this is independent of connectors it allows any connector to work with any serialization format. Examples of common formats include JSON and Avro. This setting controls the format used for internal bookkeeping data used by the framework, such as configs and offsets, so users can typically use any functioning Converter implementation.	class			low
internal.value.converter	Converter class used to convert between Kafka Connect format and the serialized form that is written to Kafka. This controls the format of the values in messages written to or read from Kafka, and since this is independent of connectors it allows any connector to work with any serialization format. Examples of common formats include JSON and Avro. This setting controls the format used for internal bookkeeping data used by the framework, such as configs and offsets, so users can typically use any functioning Converter implementation.	class			low
bootstrap.servers	A list of host/port pairs to use for establishing the initial connection to the Kafka cluster. The client will make use of all servers irrespective of which servers are specified here for bootstrapping—this list only impacts the initial hosts used to discover the full set of servers. This list should be in the form `host1:port1,host2:port2,...`. Since these servers are just used for the initial connection to discover the full cluster membership (which may change dynamically), this list need not contain the full set of servers (you may want more than one, though, in case a server is down).	list	localhost:9092		high
heartbeat.interval.ms	The expected time between heartbeats to the group coordinator when using Kafka's group management facilities. Heartbeats are used to ensure that the worker's session stays active and to facilitate rebalancing when new members join or leave the group. The value must be set lower than `session.timeout.ms`, but typically should be set no higher than 1/3 of that value. It can be adjusted even lower to control the expected time for normal rebalances.	int	3000		high
rebalance.timeout.ms	The maximum allowed time for each worker to join the group once a rebalance has begun. This is basically a limit on the amount of time needed for all tasks to flush any pending data and commit offsets. If the timeout is exceeded, then the worker will be removed from the group, which will cause offset commit failures.	int	60000		high
session.timeout.ms	The timeout used to detect worker failures. The worker sends periodic heartbeats to indicate its liveness to the broker. If no heartbeats are received by the broker before the expiration of this session timeout, then the broker will remove the worker from the group and initiate a rebalance. Note that the value must be in the allowable range as configured in the broker configuration by `group.min.session.timeout.ms` and `group.max.session.timeout.ms`.	int	10000		high
ssl.key.password	The password of the private key in the key store file. This is optional for client.	password	null		high
ssl.keystore.location	The location of the key store file. This is optional for client and can be used for two-way authentication for client.	string	null		high
ssl.keystore.password	The store password for the key store file. This is optional for client and only needed if ssl.keystore.location is configured.	password	null		high
ssl.truststore.location	The location of the trust store file.	string	null		high
ssl.truststore.password	The password for the trust store file. If a password is not set access to the truststore is still available, but integrity checking is disabled.	password	null		high
connections.max.idle.ms	Close idle connections after the number of milliseconds specified by this config.	long	540000		medium
receive.buffer.bytes	The size of the TCP receive buffer (SO_RCVBUF) to use when reading data. If the value is -1, the OS default will be used.	int	32768	[0,...]	medium
request.timeout.ms	The configuration controls the maximum amount of time the client will wait for the response of a request. If the response is not received before the timeout elapses the client will resend the request if necessary or fail the request if retries are exhausted.	int	40000	[0,...]	medium
sasl.jaas.config	JAAS login context parameters for SASL connections in the format used by JAAS configuration files. JAAS configuration file format is described here. The format for the value is: ' (=)*;'	password	null		medium

sasl.kerberos.service.name	The Kerberos principal name that Kafka runs as. This can be defined either in Kafka's JAAS config or in Kafka's config.	string	null		medium
sasl.mechanism	SASL mechanism used for client connections. This may be any mechanism for which a security provider is available. GSSAPI is the default mechanism.	string	GSSAPI		medium
security.protocol	Protocol used to communicate with brokers. Valid values are: PLAINTEXT, SSL, SASL_PLAINTEXT, SASL_SSL.	string	PLAINTEXT		medium
send.buffer.bytes	The size of the TCP send buffer (SO_SNDBUF) to use when sending data. If the value is -1, the OS default will be used.	int	131072	[0,...]	medium
ssl.enabled.protocols	The list of protocols enabled for SSL connections.	list	TLSv1.2,TLSv1.1,TLSv1		medium
ssl.keystore.type	The file format of the key store file. This is optional for client.	string	JKS		medium
ssl.protocol	The SSL protocol used to generate the SSLContext. Default setting is TLS, which is fine for most cases. Allowed values in recent JVMs are TLS, TLSv1.1 and TLSv1.2. SSL, SSLv2 and SSLv3 may be supported in older JVMs, but their usage is discouraged due to known security vulnerabilities.	string	TLS		medium
ssl.provider	The name of the security provider used for SSL connections. Default value is the default security provider of the JVM.	string	null		medium
ssl.truststore.type	The file format of the trust store file.	string	JKS		medium
worker.sync.timeout.ms	When the worker is out of sync with other workers and needs to resynchronize configurations, wait up to this amount of time before giving up, leaving the group, and waiting a backoff period before rejoining.	int	3000		medium
worker.unsync.backoff.ms	When the worker is out of sync with other workers and fails to catch up within worker.sync.timeout.ms, leave the Connect cluster for this long before rejoining.	int	300000		medium
access.control.allow.methods	Sets the methods supported for cross origin requests by setting the Access-Control-Allow-Methods header. The default value of the Access-Control-Allow-Methods header allows cross origin requests for GET, POST and HEAD.	string	""		low
access.control.allow.origin	Value to set the Access-Control-Allow-Origin header to for REST API requests.To enable cross origin access, set this to the domain of the application that should be permitted to access the API, or '*' to allow access from any domain. The default value only allows access from the domain of the REST API.	string	""		low
client.id	An id string to pass to the server when making requests. The purpose of this is to be able to track the source of requests beyond just ip/port by allowing a logical application name to be included in server-side request logging.	string	""		low
config.storage.replication.factor	Replication factor used when creating the configuration storage topic	short	3	[1,...]	low
metadata.max.age.ms	The period of time in milliseconds after which we force a refresh of metadata even if we haven't seen any partition leadership changes to proactively discover any new brokers or partitions.	long	300000	[0,...]	low
metric.reporters	A list of classes to use as metrics reporters. Implementing the `org.apache.kafka.common.metrics.MetricsReporter` interface allows plugging in classes that will be notified of new metric creation. The JmxReporter is always included to register JMX statistics.	list	""		low
metrics.num.samples	The number of samples maintained to compute metrics.	int	2	[1,...]	low
metrics.recording.level	The highest recording level for metrics.	string	INFO	[INFO, DEBUG]	low
metrics.sample.window.ms	The window of time a metrics sample is computed over.	long	30000	[0,...]	low
offset.flush.interval.ms	Interval at which to try committing offsets for tasks.	long	60000		low
offset.flush.timeout.ms	Maximum number of milliseconds to wait for records to flush and partition offset data to be committed to offset storage before cancelling the process and restoring the offset data to be committed in a future attempt.	long	5000		low
offset.storage.partitions	The number of partitions used when creating the offset storage topic	int	25	[1,...]	low
offset.storage.replication.factor	Replication factor used when creating the offset storage topic	short	3	[1,...]	low
plugin.path	List of paths separated by commas (,) that contain plugins (connectors, converters, transformations). The list should consist of top level directories that include any combination of: a) directories immediately containing jars with plugins and their dependencies b) uber-jars with plugins and their dependencies c) directories immediately containing the package directory structure of classes of plugins and their dependencies Note: symlinks will be followed to discover dependencies or plugins. Examples: plugin.path=/usr/local/share/java,/usr/local/share/kafka/plugins,/opt/connectors	list	null		low
reconnect.backoff.max.ms	The maximum amount of time in milliseconds to wait when reconnecting to a broker that has repeatedly failed to connect. If provided, the backoff per host will increase exponentially for each consecutive connection failure, up to this maximum. After calculating the backoff increase, 20% random jitter is added to avoid connection storms.	long	1000	[0,...]	low
reconnect.backoff.ms	The base amount of time to wait before attempting to reconnect to a given host. This avoids repeatedly connecting to a host in a tight loop. This backoff applies to all connection attempts by the client to a broker.	long	50	[0,...]	low
rest.advertised.host.name	If this is set, this is the hostname that will be given out to other workers to connect to.	string	null		low
rest.advertised.port	If this is set, this is the port that will be given out to other workers to connect to.	int	null		low
rest.host.name	Hostname for the REST API. If this is set, it will only bind to this interface.	string	null		low
rest.port	Port for the REST API to listen on.	int	8083		low
retry.backoff.ms	The amount of time to wait before attempting to retry a failed request to a given topic partition. This avoids repeatedly sending requests in a tight loop under some failure scenarios.	long	100	[0,...]	low
sasl.kerberos.kinit.cmd	Kerberos kinit command path.	string	/usr/bin/kinit		low
sasl.kerberos.min.time.before.relogin	Login thread sleep time between refresh attempts.	long	60000		low
sasl.kerberos.ticket.renew.jitter	Percentage of random jitter added to the renewal time.	double	0.05		low
sasl.kerberos.ticket.renew.window.factor	Login thread will sleep until the specified window factor of time from last refresh to ticket's expiry has been reached, at which time it will try to renew the ticket.	double	0.8		low

ssl.cipher.suites	A list of cipher suites. This is a named combination of authentication, encryption, MAC and key exchange algorithm used to negotiate the security settings for a network connection using TLS or SSL network protocol. By default all the available cipher suites are supported.	list	null		low
ssl.endpoint.identification.algorithm	The endpoint identification algorithm to validate server hostname using server certificate.	string	null		low
ssl.keymanager.algorithm	The algorithm used by key manager factory for SSL connections. Default value is the key manager factory algorithm configured for the Java Virtual Machine.	string	SunX509		low
ssl.secure.random.implementation	The SecureRandom PRNG implementation to use for SSL cryptography operations.	string	null		low
ssl.trustmanager.algorithm	The algorithm used by trust manager factory for SSL connections. Default value is the trust manager factory algorithm configured for the Java Virtual Machine.	string	PKIX		low
status.storage.partitions	The number of partitions used when creating the status storage topic	int	5	[1,...]	low
status.storage.replication.factor	Replication factor used when creating the status storage topic	short	3	[1,...]	low
task.shutdown.graceful.timeout.ms	Amount of time to wait for tasks to shutdown gracefully. This is the total amount of time, not per task. All task have shutdown triggered, then they are waited on sequentially.	long	5000		low

3.6 Kafka Streams Configs

Below is the configuration of the Kafka Streams client library.

3.7 AdminClient Configs

Below is the configuration of the Kafka Admin client library.

NAME	DESCRIPTION	TYPE	DEFAULT	VALID VALUES	IMPORTANCE
bootstrap.servers	A list of host/port pairs to use for establishing the initial connection to the Kafka cluster. The client will make use of all servers irrespective of which servers are specified here for bootstrapping—this list only impacts the initial hosts used to discover the full set of servers. This list should be in the form `host1:port1,host2:port2,...`. Since these servers are just used for the initial connection to discover the full cluster membership (which may change dynamically), this list need not contain the full set of servers (you may want more than one, though, in case a server is down).	list			high
ssl.key.password	The password of the private key in the key store file. This is optional for client.	password	null		high
ssl.keystore.location	The location of the key store file. This is optional for client and can be used for two-way authentication for client.	string	null		high
ssl.keystore.password	The store password for the key store file. This is optional for client and only needed if ssl.keystore.location is configured.	password	null		high
ssl.truststore.location	The location of the trust store file.	string	null		high
ssl.truststore.password	The password for the trust store file. If a password is not set access to the truststore is still available, but integrity checking is disabled.	password	null		high
client.id	An id string to pass to the server when making requests. The purpose of this is to be able to track the source of requests beyond just ip/port by allowing a logical application name to be included in server-side request logging.	string	""		medium
connections.max.idle.ms	Close idle connections after the number of milliseconds specified by this config.	long	300000		medium
receive.buffer.bytes	The size of the TCP receive buffer (SO_RCVBUF) to use when reading data. If the value is -1, the OS default will be used.	int	65536	[-1,...]	medium
request.timeout.ms	The configuration controls the maximum amount of time the client will wait for the response of a request. If the response is not received before the timeout elapses the client will resend the request if necessary or fail the request if retries are exhausted.	int	120000	[0,...]	medium
sasl.jaas.config	JAAS login context parameters for SASL connections in the format used by JAAS configuration files. JAAS configuration file format is described here. The format for the value is: ' (=)*;'	password	null		medium
sasl.kerberos.service.name	The Kerberos principal name that Kafka runs as. This can be defined either in Kafka's JAAS config or in Kafka's config.	string	null		medium
sasl.mechanism	SASL mechanism used for client connections. This may be any mechanism for which a security provider is available. GSSAPI is the default mechanism.	string	GSSAPI		medium
security.protocol	Protocol used to communicate with brokers. Valid values are: PLAINTEXT, SSL, SASL_PLAINTEXT, SASL_SSL.	string	PLAINTEXT		medium
send.buffer.bytes	The size of the TCP send buffer (SO_SNDBUF) to use when sending data. If the value is -1, the OS default will be used.	int	131072	[-1,...]	medium
ssl.enabled.protocols	The list of protocols enabled for SSL connections.	list	TLSv1.2,TLSv1.1,TLSv1		medium
ssl.keystore.type	The file format of the key store file. This is optional for client.	string	JKS		medium
ssl.protocol	The SSL protocol used to generate the SSLContext. Default setting is TLS, which is fine for most cases. Allowed values in recent JVMs are TLS, TLSv1.1 and TLSv1.2. SSL, SSLv2 and SSLv3 may be supported in older JVMs, but their usage is discouraged due to known security vulnerabilities.	string	TLS		medium
ssl.provider	The name of the security provider used for SSL connections. Default value is the default security provider of the JVM.	string	null		medium
ssl.truststore.type	The file format of the trust store file.	string	JKS		medium
metadata.max.age.ms	The period of time in milliseconds after which we force a refresh of metadata even if we haven't seen any partition leadership changes to proactively discover any new brokers or partitions.	long	300000	[0,...]	low
metric.reporters	A list of classes to use as metrics reporters. Implementing the `org.apache.kafka.common.metrics.MetricsReporter` interface allows plugging in	list	""		low

	classes that will be notified of new metric creation. The JmxReporter is always included to register JMX statistics.				
metrics.num.samples	The number of samples maintained to compute metrics.	int	2	[1,...]	low
metrics.recording.level	The highest recording level for metrics.	string	INFO	[INFO, DEBUG]	low
metrics.sample.window.ms	The window of time a metrics sample is computed over.	long	30000	[0,...]	low
reconnect.backoff.max.ms	The maximum amount of time in milliseconds to wait when reconnecting to a broker that has repeatedly failed to connect. If provided, the backoff per host will increase exponentially for each consecutive connection failure, up to this maximum. After calculating the backoff increase, 20% random jitter is added to avoid connection storms.	long	1000	[0,...]	low
reconnect.backoff.ms	The base amount of time to wait before attempting to reconnect to a given host. This avoids repeatedly connecting to a host in a tight loop. This backoff applies to all connection attempts by the client to a broker.	long	50	[0,...]	low
retries	The maximum number of times to retry a call before failing it.	int	5	[0,...]	low
retry.backoff.ms	The amount of time to wait before attempting to retry a failed request. This avoids repeatedly sending requests in a tight loop under some failure scenarios.	long	100	[0,...]	low
sasl.kerberos.kinit.cmd	Kerberos kinit command path.	string	/usr/bin/kinit		low
sasl.kerberos.min.time.before.relogin	Login thread sleep time between refresh attempts.	long	60000		low
sasl.kerberos.ticket.renew.jitter	Percentage of random jitter added to the renewal time.	double	0.05		low
sasl.kerberos.ticket.renew.window.factor	Login thread will sleep until the specified window factor of time from last refresh to ticket's expiry has been reached, at which time it will try to renew the ticket.	double	0.8		low
ssl.cipher.suites	A list of cipher suites. This is a named combination of authentication, encryption, MAC and key exchange algorithm used to negotiate the security settings for a network connection using TLS or SSL network protocol. By default all the available cipher suites are supported.	list	null		low
ssl.endpoint.identification.algorithm	The endpoint identification algorithm to validate server hostname using server certificate.	string	null		low
ssl.keymanager.algorithm	The algorithm used by key manager factory for SSL connections. Default value is the key manager factory algorithm configured for the Java Virtual Machine.	string	SunX509		low
ssl.secure.random.implementation	The SecureRandom PRNG implementation to use for SSL cryptography operations.	string	null		low
ssl.trustmanager.algorithm	The algorithm used by trust manager factory for SSL connections. Default value is the trust manager factory algorithm configured for the Java Virtual Machine.	string	PKIX		low

4. 设计思想

4.1 动机

我们设计的 Kafka 能够作为一个统一的平台来处理大公司可能拥有的所有实时数据馈送。 要做到这点，我们必须考虑相当广泛的用例。

Kafka 必须具有高吞吐量来支持高容量事件流，例如实时日志聚合。

Kafka 需要能够正常处理大量的数据积压，以便能够支持来自离线系统的周期性数据加载。

这也意味着系统必须处理低延迟分发，来处理更传统的消息传递用例。

我们希望支持对这些馈送进行分区，分布式，以及实时处理来创建新的分发馈送等特性。由此产生了我们的分区模式和消费者模式。

最后，在数据流被推送到其他数据系统进行服务的情况下，我们要求系统在出现机器故障时必须能够保证容错。

为支持这些使用场景导致我们设计了一些独特的元素，使得 Kafka 相比传统的消息系统更像是数据库日志。我们将在后面的章节中概述设计中的部分要素。

4.2 持久化

不要害怕文件系统！

Kafka 对消息的存储和缓存严重依赖于文件系统。人们对于“磁盘速度慢”的普遍印象，使得人们对于持久化的架构能够提供强有力的性能产生怀疑。事实上，磁盘的速度比人们预期的要慢的多，也快得多，这取决于人们使用磁盘的方式。而且设计合理的磁盘结构通常可以和网络一样快。

关于磁盘性能的关键事实是，磁盘的吞吐量和过去十年里磁盘的寻址延迟不同。因此，使用6个7200rpm、SATA接口、RAID-5的磁盘阵列在JBOD配置下的顺序写入的性能约为600MB/秒，但随机写入的性能仅约为100k/秒，相差6000倍以上。因为线性的读取和写入是磁盘使用模式中最有规律的，并且由操作系统进行了大量的优化。现代操作系统提供了 read-ahead 和 write-behind 技术，read-ahead 是以大的 data block 为单位预先读取数据，而 write-behind 是将多个小型的逻辑写合并成一次大型的物理磁盘写入。关于该问题的进一步讨论可以参考 ACM Queue article，他们发现实际上顺序磁盘访问在某些情况下比随机内存访问还要快！

为了弥补这种性能差异，现代操作系统在越来越注重使用内存对磁盘进行 cache。现代操作系统主动将所有空闲内存用作 disk caching，代价是在内存回收时性能会有所降低。所有对磁盘的读写操作都会通过这个统一的 cache。如果不使用直接I/O，该功能不能轻易关闭。因此即使进程维护了 in-process cache，该数据也可能会被复制到操作系统的 pagecache 中，事实上所有内容都被存储了两份。

此外，Kafka 建立在 JVM 之上，任何了解 Java 内存使用的人都知道两点：

1. 对象的内存开销非常高，通常是所存储的数据的两倍(甚至更多)。
2. 随着堆中数据的增加，Java 的垃圾回收变得越来越复杂和缓慢。

受这些因素影响，相比于维护 in-memory cache 或者其他结构，使用文件系统和 pagecache 显得更有优势--我们可以通过自动访问所有空闲内存将可用缓存的容量至少翻倍，并且通过存储紧凑的字节结构而不是独立的对象，有望将缓存容量再翻一番。 这样使得32GB的机器缓存容量可以达到28-30GB,并且不会产生额外的 GC 负担。此外，即使服务重新启动，缓存依旧可用，而 in-process cache 则需要在内存中重建(重建一个10GB的缓存可能需要10分钟)，否则进程就要从 cold cache 的状态开始(这意味着进程最初的性能表现十分糟糕)。 这同时也极大的简化了代码，因为所有保持 cache 和文件系统之间一致性的逻辑现在都被放到了 OS 中，这样做比一次性的进程内缓存更准确、更高效。如果你的磁盘使用更倾向于顺序读取，那么 read-ahead 可以有效的使用每次从磁盘中读取到的有用数据预先填充 cache。

这里给出了一个非常简单的设计：相比于维护尽可能多的 in-memory cache，并且在空间不足的时候匆忙将数据 flush 到文件系统，我们把这个过程倒过来。所有数据一开始就被写入到文件系统的持久化日志中，而不用在 cache 空间不足的时候 flush 到磁盘。实际上，这表明数据被转移到了内核的 pagecache 中。

这种 pagecache-centric 的设计风格出现在一篇关于 Varnish 设计的文章中。

常量时间就足够了

消息系统使用的持久化数据结构通常是和 BTree 相关联的消费者队列或者其他用于存储消息源数据的通用随机访问数据结构。BTree 是最通用的数据结构，可以在消息系统能够支持各种事务性和非事务性语义。 虽然 BTree 的操作复杂度是 O(log N)，但成本也相当高。通常我们认为 O(log N) 基本等同于常数时间，但这条在磁盘操作中不成立。磁盘寻址是每10ms一跳，并且每个磁盘同时只能执行一次寻址，因此并行性受到了限制。因此即使是少量的磁盘寻址也会很高的开销。由于存储系统将非常快的cache操作和非常慢的物理磁盘操作混

合在一起，当数据随着 fixed cache 增加时，可以看到树的性能通常是非线性的——比如数据翻倍时性能下降不只两倍。

所以直观来看，持久化队列可以建立在简单的读取和向文件后追加两种操作之上，这和日志解决方案相同。这种架构的优点在于所有的操作复杂度都是O(1)，而且读操作不会阻塞写操作，读操作之间也不会互相影响。这有着明显的性能优势，由于性能和数据大小完全分离开来——服务器现在可以充分利用大量廉价、低转速的1+TB SATA硬盘。 虽然这些硬盘的寻址性能很差，但他们在大规模读写方面的性能是可以接受的，而且价格是原来的三分之一、容量是原来的三倍。

在不产生任何性能损失的情况下能够访问几乎无限的硬盘空间，这意味着我们可以提供一些其它消息系统不常见的特性。例如：在 Kafka 中，我们可以让消息保留相对较长的一段时间(比如一周)，而不是试图在被消费后立即删除。正如我们后面将要提到的，这给消费者带来了很大的灵活性。

4.3 Efficiency

我们在性能上已经做了很大的努力。 我们主要的使用场景是处理WEB活动数据，这个数据量非常大，因为每个页面都有可能大量的写入。此外我们假设每个发布 message 至少被一个consumer (通常很多个consumer)消费， 因此我们尽可能的去降低消费的代价。

我们还发现，从构建和运行许多相似系统的经验上来看，性能是多租户运营的关键。如果下游的基础设施服务很轻易被应用层冲击形成瓶颈，那么一些小的改变也会造成问题。通过非常快的(缓存)技术，我们能确保应用层冲击基础设施之前，将负载稳定下来。 当尝试去运行支持集中式集群上成百上千个应用程序的集中式服务时，这一点很重要，因为应用层使用方式几乎每天都会发生变化。

我们在上一节讨论了磁盘性能。 一旦消除了磁盘访问模式不佳的情况，该类系统性能低下的主要原因就剩下了两个：大量的小型 I/O 操作，以及过多的字节拷贝。

小型的 I/O 操作发生在客户端和服务端之间以及服务端自身的持久化操作中。

为了避免这种情况，我们的协议是建立在一个 “消息块” 的抽象基础上，合理将消息分组。 这使得网络请求将多个消息打包成一组，而不是每次发送一条消息，从而使整组消息分担网络中往返的开销。Consumer 每次获取多个大型有序的消息块，并由服务端 依次将消息块一次加载到它的日志中。

这个简单的优化对速度有着数量级的提升。批处理允许更大的网络数据包，更大的顺序读写磁盘操作，连续的内存块等等，所有这些都使 KafKa 将随机流消息顺序写入到磁盘， 再由 consumers 进行消费。

另一个低效率的操作是字节拷贝，在消息量少时，这不是什么问题。但是在高负载的情况下，影响就不容忽视。为了避免这种情况，我们使用 producer ， broker 和 consumer 都共享的标准化的二进制消息格式，这样数据块不用修改就能在他们之间传递。

broker 维护的消息日志本身就是一个文件目录，每个文件都由一系列以相同格式写入到磁盘的消息集合组成，这种写入格式被 producer 和 consumer 共用。保持这种通用格式可以对一些很重要的操作进行优化: 持久化日志块的网络传输。 现代的unix 操作系统提供了一个高度优化的编码方式，用于将数据从 pagecache 转移到 socket 网络连接中；在 Linux 中系统调用 sendfile 做到这一点。

为了理解 sendfile 的意义，了解数据从文件到套接字的常见数据传输路径就非常重要：

1. 操作系统从磁盘读取数据到内核空间的 pagecache
2. 应用程序读取内核空间的数据到用户空间的缓冲区
3. 应用程序将数据(用户空间的缓冲区)写回内核空间到套接字缓冲区(内核空间)
4. 操作系统将数据从套接字缓冲区(内核空间)复制到通过网络发送的 NIC 缓冲区

这显然是低效的，有四次 copy 操作和两次系统调用。使用 sendfile 方法，可以允许操作系统将数据从 pagecache 直接发送到网络，这样避免重新复制数据。所以这种优化方式，只需要最后一步的copy操作，将数据复制到 NIC 缓冲区。

我们期望一个普遍的应用场景，一个 topic 被多消费者消费。使用上面提交的 zero-copy（零拷贝）优化，数据在使用时只会被复制到 pagecache 中一次，节省了每次拷贝到用户空间内存中，再从用户空间进行读取的消耗。这使得消息能够以接近网络连接速度的 上限进行消费。

pagecache 和 sendfile 的组合使用意味着，在一个kafka集群中，大多数 consumer 消费时，您将看不到磁盘上的读取活动，因为数据将完全由缓存提供。

JAVA 中更多有关 sendfile 方法和 zero-copy （零拷贝）相关的资料，可以参考这里的 文章。

端到端的批量压缩

在某些情况下，数据传输的瓶颈不是 CPU ，也不是磁盘，而是网络带宽。对于需要通过广域网在数据中心之间发送消息的数据管道尤其如此。当然，用户可以在不需要 Kakfa 支持下一次一个的压缩消息。但是这样会造成非常差的压缩比和消息重复类型的冗余，比如 JSON 中的字段名称或者是或 Web 日志中的用户代理或公共字符串值。高性能的压缩是一次压缩多个消息，而不是压缩单个消息。

Kafka 以高效的批处理格式支持一批消息可以压缩在一起发送到服务器。这批消息将以压缩格式写入，并且在日志中保持压缩，只会在 consumer 消费时解压缩。

Kafka 支持 GZIP，Snappy 和 LZ4 压缩协议，更多有关压缩的资料参看 这里。

4.4 The Producer

Load balancing

生产者直接发送数据到主分区的服务器上，不需要经过任何中间路由。 为了让生产者实现这个功能，所有的 kafka 服务器节点都能响应这样的元数据请求： 哪些服务器是活着的，主题的哪些分区是主分区，分配在哪个服务器上，这样生产者就能适当地直接发送它的请求到服务器上。

客户端控制消息发送数据到哪个分区，这个可以实现随机的负载均衡方式,或者使用一些特定语义的分区函数。 我们有提供特定分区的接口让用于根据指定的键值进行hash分区(当然也有选项可以重写分区函数)，例如，如果使用用户ID作为key，则用户相关的所有数据都会被分发到同一个分区上。 这允许消费者在消费数据时做一些特定的本地化处理。这样的分区风格经常被设计用于一些本地处理比较敏感的消费者。

Asynchronous send

批处理是提升性能的一个主要驱动，为了允许批量处理，kafka 生产者会尝试在内存中汇总数据，并用一次请求批次提交信息。 批处理，不仅仅可以配置指定的消息数量，也可以指定等待特定的延迟时间(如64k 或10ms)，这允许汇总更多的数据后再发送，在服务器端也会减少更多的IO操作。 该缓冲是可配置的，并给出了一个机制，通过权衡少量额外的延迟时间获取更好的吞吐量。

更多的细节可以在 producer 的 configuration 和 api文档中进行详细的了解。

4.5 消费者

Kafka consumer通过向 broker 发出一个“fetch”请求来获取它想要消费的 partition。consumer 的每个请求都在 log 中指定了对应的 offset，并接收从该位置开始的一大块数据。因此，consumer 对于该位置的控制就显得极为重要，并且可以在需要的时候通过回退到该位置再次消费对应的数据。

Push vs. pull

最初我们考虑的问题是：究竟是由 consumer 从 broker 那里 pull 数据，还是由 broker 将数据 push 到 consumer。Kafka 在这方面采取了一种较为传统的设计方式，也是大多数的消息系统所共享的方式：即 producer 把数据 push 到 broker，然后 consumer 从 broker 中 pull 数据。 也有一些 logging-centric 的系统，比如 Scribe 和 Apache Flume，沿着一条完全不同的 push-based 的路径，将数据 push 到下游节点。这两种方法都有优缺点。然而，由于 broker 控制着数据传输速率， 所以 push-based 系统很难处理不同的 consumer。让 broker 控制数据传输速率主要是为了让 consumer 能够以可能的最大速率消费；不幸的是，这导致着在 push-based 的系统中，当消费速率低于生产速率时，consumer 往往会不堪重负（本质上类似于拒绝服务攻击）。pull-based 系统有一个很好的特性， 那就是当 consumer 速率落后于 producer 时，可以在适当的时间赶上来。还可以通过使用某种 backoff 协议来减少这种现象：即 consumer 可以通过 backoff 表示它已经不堪重负了，然而通过获得负载情况来充分使用 consumer（但永远不超载）这一方式实现起来比它看起来更棘手。前面以这种方式构建系统的尝试，引导着 Kafka 走向了更传统的 pull 模型。

另一个 pull-based 系统的优点在于：它可以大批量生产要发送给 consumer 的数据。而 push-based 系统必须选择立即发送请求或者积累更多的数据，然后在不知道下游的 consumer 能否立即处理它的情况下发送这些数据。如果系统调整为低延迟状态，这就会导致一次只发送一条消息，以至于传输的数据不再被缓冲，这种方式是极度浪费的。 而 pull-based 的设计修复了该问题，因为 consumer 总是将所有可用的（或者达到配置的最大长度）消息 pull 到 log 当前位置的后面，从而使得数据能够得到最佳的处理而不会引入不必要的延迟。

简单的 pull-based 系统的不足之处在于：如果 broker 中没有数据，consumer 可能会在一个紧密的循环中结束轮询，实际上 busy-waiting 直到数据到来。为了避免 busy-waiting，我们在 pull 请求中加入参数，使得 consumer 在一个“long pull”中阻塞等待，直到数据到来（还可以选择等待给定字节长度的数据来确保传输长度）。

你可以想象其它可能的只基于 pull 的， end-to-end 的设计。例如producer 直接将数据写入一个本地的 log，然后 broker 从 producer 那里 pull 数据，最后 consumer 从 broker 中 pull 数据。通常提到的还有“store-and-forward”式 producer， 这是一种很有趣的设计，但我们觉得它跟我们设定的有数以千计的生产者的应用场景

不太相符。我们在运行大规模持久化数据系统方面的经验使我们感觉到，横跨多个应用、涉及数千磁盘的系统事实上并不会让事情更可靠，反而会成为操作时的噩梦。在实践中， 我们发现可以通过大规模运行的带有强大的 SLAs 的 pipeline，而省略 producer 的持久化过程。

消费者的位置

令人惊讶的是，持续追踪*已经被消费的内容*是消息系统的关键性能点之一。

大多数消息系统都在 broker 上保存被消费消息的元数据。也就是说，当消息被传递给 consumer，broker 要么立即在本地记录该事件，要么等待 consumer 的确认后再记录。这是一种相当直接的选择，而且事实上对于单机服务器来说，也没与其它地方能够存储这些状态信息。 由于大多数消息系统用于存储的数据结构规模都很小，所以这也是一个很实用的选择——因为只要 broker 知道哪些消息被消费了，就可以在本地立即进行删除，一直保持较小的数据量。

也许不太明显，但要让 broker 和 consumer 就被消费的数据保持一致性也不是一个小问题。如果 broker 在每条消息被发送到网络的时候，立即将其标记为 **consumed**，那么一旦 consumer 无法处理该消息（可能由 consumer 崩溃或者请求超时或者其他原因导致），该消息就会丢失。 为了解决消息丢失的问题，许多消息系统增加了确认机制：即当消息被发送出去的时候，消息仅被标记为**sent** 而不是 **consumed**；然后 broker 会等待一个来自 consumer 的特定确认，再将消息标记为**consumed**。这个策略修复了消息丢失的问题，但也产生了新问题。 首先，如果 consumer 处理了消息但在发送确认之前出错了，那么该消息就会被消费两次。第二个是关于性能的，现在 broker 必须为每条消息保存多个状态（首先对其加锁，确保该消息只被发送一次，然后将其永久的标记为 consumed，以便将其移除）。 还有更棘手的问题要处理，比如如何处理已经发送但一直得不到确认的消息。

Kafka 使用完全不同的方式解决消息丢失问题。Kafka的 topic 被分割成了一组完全有序的 partition，其中每一个 partition 在任意给定的时间内只能被每个订阅了这个 topic 的 consumer 组中的一个 consumer 消费。这意味着 partition 中 每一个 consumer 的位置仅仅是一个数字，即下一条要消费的消息的offset。这使得被消费的消息的状态信息相当少，每个 partition 只需要一个数字。这个状态信息还可以作为周期性的 checkpoint。这以非常低的代价实现了和消息确认机制等同的效果。

这种方式还有一个附加的好处。consumer 可以*回退*到之前的 offset 来再次消费之前的数据，这个操作违反了队列的基本原则，但事实证明对大多数 consumer 来说这是一个必不可少的特性。 例如，如果 consumer 的代码有 bug，并且在 bug 被发现前已经有一部分数据被消费了， 那么 consumer 可以在 bug 修复后通过回退到之前的 offset 来再次消费这些数据。

离线数据加载

可伸缩的持久化特性允许 consumer 只进行周期性的消费，例如批量数据加载，周期性将数据加载到诸如 Hadoop 和关系型数据库之类的离线系统中。

在 Hadoop 的应用场景中，我们通过将数据加载分配到多个独立的 map 任务来实现并行化，每一个 map 任务负责一个 node/topic/partition，从而达到充分并行化。Hadoop 提供了任务管理机制，失败的任务可以重新启动而不会有重复数据的风险，只需要简单的从原来的位置重启即可。

4.6 消息交付语义

现在我们对于 producer 和 consumer 的工作原理已将有了一点了解，让我们接着讨论 Kafka 在 producer 和 consumer 之间提供的语义保证。显然，Kafka可以提供的消息交付语义保证有多种：

- *At most once*——消息可能会丢失但绝不重传。
- *At least once*——消息可以重传但绝不丢失。
- *Exactly once*——这正是人们想要的, 每一条消息只被传递一次.

值得注意的是，这个问题被分成了两部分：发布消息的持久性保证和消费消息的保证。

很多系统声称提供了“Exactly once”的消息交付语义, 然而阅读它们的细则很重要, 因为这些声称大多数都是误导性的 (即它们没有考虑 consumer 或 producer 可能失败的情况，以及存在多个 consumer 进行处理的情况，或者写入磁盘的数据可能丢失的情况。).

Kafka 的语义是直截了当的。 发布消息时，我们会有一个消息的概念被“committed”到 log 中。 一旦消息被提交，只要有一个 broker 备份了该消息写入的 partition，并且保持“alive”状态，该消息就不会丢失。 有关 committed message 和 alive partition 的定义，以及我们试图解决的故障类型都将在下一节进行细致描述。 现在让我们假设存在完美无缺的 broker，然后来试着理解 Kafka 对 producer 和 consumer 的语义保证。如果一个 producer 在试图发送消息的时候发生了网络故障， 则不确定网络错误发生在消息提交之前还是之后。这与使用自动生成的键插入到数据库表中的语义场景很相似。

在 0.11.0.0 之前的版本中, 如果 producer 没有收到表明消息已经被提交的响应, 那么 producer 除了将消息重传之外别无选择。 这里提供的是 at-least-once 的消息交付语义，因为如果最初的请求事实上执行成功了，那么重传过程中该消息就会被再次写入到 log 当中。 从 0.11.0.0 版本开始，Kafka producer新增了幂等性的传递选项，该选项保证重传不会在 log 中产生重复条目。 为实现这个目的, broker 给每个 producer 都分配了一个 ID，并且 producer 给每条被发送的消息分配了一个序列号来避免产生重复的消息。 同样也是从 0.11.0.0 版本开始, producer 新增了使用类似事务性的语义将消息发送到多个 topic partition 的功能： 也就是说，要么所有的消息都被成功的写入到了 log，要么一个都没写进去。这种语义的主要应用场景就是 Kafka topic 之间的 exactly-once 的数据传递(如下所述)。

并非所有使用场景都需要这么强的保证。对于延迟敏感的应用场景，我们允许生产者指定它需要的持久性级别。如果 producer 指定了它想要等待消息被提交，则可以使用10ms的量级。然而， producer 也可以指定它想要完全异步地执行发送，或者它只想等待直到 leader 节点拥有该消息（follower 节点有没有无所谓）。

现在让我们从 consumer 的视角来描述语义。 所有的副本都有相同的 log 和相同的 offset。consumer 负责控制它在 log 中的位置。如果 consumer 永远不崩溃，那么它可以将这个位置信息只存储在内存中。但如果 consumer 发生了故障，我们希望这个 topic partition 被另一个进程接管， 那么新进程需要选择一个合适的位置开始进行处理。假设 consumer 要读取一些消息--它有几个处理消息和更新位置的选项。

1. Consumer 可以先读取消息，然后将它的位置保存到 log 中，最后再对消息进行处理。在这种情况下，消费者进程可能会在保存其位置之后，带还没有保存消息处理的输出之前发生崩溃。而在这种情况下，即使在此位置之前的一些消息没有被处理，接管处理的进程将从保存的位置开始。在 consumer 发生故障的情况下，这对应于“at-most-once”的语义，可能会有消息得不到处理。

2. Consumer 可以先读取消息，然后处理消息，最后再保存它的位置。在这种情况下，消费者进程可能会在处理了消息之后，但还没有保存位置之前发生崩溃。而在这种情况下，当新的进程接管后，它最初收到的一部分消息都已经被处理过了。在 consumer 发生故障的情况下，这对应于“at-least-once”的语义。 在许多应用场景中，消息都设有一个主键，所以更新操作是幂等的（相同的消息接收两次时，第二次写入会覆盖掉第一次写入的记录）。

那么 exactly once 语义（即你真正想要的东西）呢？当从一个 kafka topic 中消费并输出到另一个 topic 时 (正如在一个Kafka Streams 应用中所做的那样)，我们可以使用我们上文提到的 0.11.0.0 版本中的新事务型 producer，并将 consumer 的位置存储为一个 topic 中的消息，所以我们可以在输出 topic 接收已经被处理的数据的时候，在同一个事务中向 Kafka 写入 offset。如果事务被中断，则消费者的位置将恢复到原来的值，而输出 topic 上产生的数据对其他消费者是否可见，取决于事务的“隔离级别”。 在默认的“read_uncommitted”隔离级别中，所有消息对 consumer 都是可见的，即使它们是中止的事务的一部分，但是在“read_committed”的隔离级别中，消费者只能访问已提交的事务中的消息（以及任何不属于事务的消息）。

在写入外部系统的应用场景中，限制在于需要在 consumer 的 offset 与实际存储为输出的内容间进行协调。解决这一问题的经典方法是在 consumer offset 的存储和 consumer 的输出结果的存储之间引入 two-phase commit。但这可以用更简单的方法处理，而且通常的做法是让 consumer 将其 offset 存储在与其输出相同的位置。 这也是一种更好的方式，因为大多数 consumer 想写入的输出系统都不支持 two-phase commit。举个例子，Kafka Connect连接器，它将所读取的数据和数据的 offset 一起写入到 HDFS，以保证数据和 offset 都被更新，或者两者都不被更新。 对于其它很多需要这些较强语义，并且没有主键来避免消息重复的数据系统，我们也遵循类似的模式。

因此，事实上 Kafka 在Kafka Streams中支持了exactly-once 的消息交付功能，并且在 topic 之间进行数据传递和处理时，通常使用事务型 producer/consumer 提供 exactly-once 的消息交付功能。 到其它目标系统的 exactly-once 的消息交付通常需要与该类系统协作，但 Kafka 提供了 offset，使得这种应用场景的实现变得可行。(详见 Kafka Connect)。否则，Kafka 默认保证 at-least-once 的消息交付， 并且 Kafka 允许用户通过禁用 producer 的重传功能和让 consumer 在处理一批消息之前提交 offset，来实现 at-most-once 的消息交付。

4.7 Replication

Kafka 允许 topic 的 partition 拥有若干副本，你可以在server端配置partition 的副本数量。当集群中的节点出现故障时，能自动进行故障转移，保证数据的可用性。

其他的消息系统也提供了副本相关的特性，但是在我们（带有偏见）看来，他们的副本功能不常用，而且有很大缺点：slaves 处于非活动状态，导致吞吐量受到严重影响，并且还要手动配置副本机制。Kafka 默认使用备份机制，事实上，我们将没有设置副本数 的 topic 实现为副本数为1的 topic 。

创建副本的单位是 topic 的 partition ，正常情况下， 每个分区都有一个 leader 和零或多个 followers 。 总的副本数是包含 leader 的总和。 所有的读写操作都由 leader 处理，一般 partition 的数量都比 broker 的数量多的多，各分区的 leader 均 匀的分布在brokers 中。所有的 followers 节点都同步 leader 节点的日志，日志中的消息和偏移量都和 leader 中的一致。（当然, 在任何给定时间, leader 节点的日志末尾时可能有几个消息尚未被备份完成）。

Followers 节点就像普通的 consumer 那样从 leader 节点那里拉取消息并保存在自己的日志文件中。Followers 节点可以从 leader 节点那里批量拉取消息日志到自己的日志文件中。

与大多数分布式系统一样，自动处理故障需要精确定义节点 “alive” 的概念。Kafka 判断节点是否存活有两种方式。

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

我们认为满足这两个条件的节点处于 “in sync” 状态，区别于 “alive” 和 “failed” 。 Leader会追踪所有 “in sync” 的节点。如果有节点挂掉了, 或是写超时, 或是心跳超时, leader 就会把它从同步副本列表中移除。 同步超时和写超时的时间由 replica.lag.time.max.ms 配置确定。

分布式系统中，我们只尝试处理 “fail/recover” 模式的故障，即节点突然停止工作，然后又恢复（节点可能不知道自己曾经挂掉）的状况。Kafka 没有处理所谓的 “Byzantine” 故障，即一个节点出现了随意响应和恶意响应（可能由于 bug 或 非法操作导致）。

现在, 我们可以更精确地定义, 只有当消息被所有的副本节点加入到日志中时, 才算是提交, 只有提交的消息才会被 consumer 消费, 这样就不用担心一旦 leader 挂掉了消息会丢失。另一方面， producer 也 可以选择是否等待消息被提交，这取决他们的设置在延迟时间和持久性之间的权衡，这个选项是由 producer 使用的 acks 设置控制。 请注意，Topic 可以设置同步备份的最小数量， producer 请求确认消息是否被写入到所有的备份时, 可以用最小同步数量判断。如果 producer 对同步的备份数没有严格的要求，即使同步的备份数量低于 最小同步数量（例如，仅仅只有 leader 同步了数据），消息也会被提交，然后被消费。

在所有时间里，Kafka 保证只要有至少一个同步中的节点存活，提交的消息就不会丢失。

节点挂掉后，经过短暂的故障转移后，Kafka将仍然保持可用性，但在网络分区（ network partitions ）的情况下可能不能保持可用性。

备份日志：Quorums, ISRs, 和状态机

Kafka的核心是备份日志文件。备份日志文件是分布式数据系统最基础的要素之一，实现方法也有很多种。其他系统也可以用 kafka 的备份日志模块来实现状态机风格的分布式系统

备份日志按照一系列有序的值(通常是编号为0、1、2、…)进行建模。有很多方法可以实现这一点，但最简单和最快的方法是由 leader 节点选择需要提供的有序的值，只要 leader 节点还存活，所有的 follower 只需要拷贝数据并按照 leader 节点的顺序排序。

当然，如果 leader 永远都不会挂掉，那我们就不需要 follower 了。 但是如果 leader crash，我们就需要从 follower 中选举出一个新的 leader。 但是 followers 自身也有可能落后或者 crash，所以 我们必须确保我们 leader的候选者们 是一个数据同步 最新的 follower 节点。

如果选择写入时候需要保证一定数量的副本写入成功，读取时需要保证读取一定数量的副本，读取和写入之间有重叠。这样的读写机制称为 Quorum。

这种权衡的一种常见方法是对提交决策和 leader 选举使用多数投票机制。Kafka 没有采取这种方式，但是我们还是要研究一下这种投票机制，来理解其中蕴含的权衡。假设我们有2*f*+ 1个副本，如果在 leader 宣布消息提交之前必须有*f*+1个副本收到 该消息，并且如果我们从这至少*f*+1个副本之中，有着最完整的日志记录的 follower 里来选择一个新的 leader，那么在故障次数少于*f*的情况下，选举出的 leader 保证具有所有提交的消息。这是因为在任意*f*+1个副本中，至少有一个副本一定包含 了所有提交的消息。该副本的日志将是最完整的，因此将被选为新的 leader。这个算法都必须处理许多其他细节（例如精确定义怎样使日志更加完整，确保在 leader down 掉期间, 保证日志一致性或者副本服务器的副本集的改变），但是现在我们将忽略这些细节。

这种大多数投票方法有一个非常好的优点：延迟是取决于最快的服务器。也就是说，如果副本数是3，则备份完成的等待时间取决于最快的 Follwer 。

这里有很多分布式算法，包含 ZooKeeper 的 Zab, Raft, 和 Viewstamped Replication. 我们所知道的与 Kafka 实际执行情况最相似的学术刊物是来自微软的 PacificA

大多数投票的缺点是，多数的节点挂掉让你不能选择 leader。要冗余单点故障需要三份数据，并且要冗余两个故障需要五份的数据。根据我们的经验，在一个系统中，仅仅靠冗余来避免单点故障是不够的，但是每写5次，对磁盘空间需求是5倍， 吞吐量下降到 1/5，这对于处理海量数据问题是不切实际的。这可能是为什么 quorum 算法更常用于共享集群配置（如 ZooKeeper ）， 而不适用于原始数据存储的原因，例如 HDFS 中 namenode 的高可用是建立在 基于投票的元数据 ， 这种代价高昂的存储方式不适用数据本身。

Kafka 采取了一种稍微不同的方法来选择它的投票集。 Kafka 不是用大多数投票选择 leader 。Kafka 动态维护了一个同步状态的备份的集合 （a set of in-sync replicas）， 简称 ISR ，在这个集合中的节点都是和 leader 保持高度一致的，只有这个集合的成员才 有资格被选举为 leader，一条消息必须被这个集合 *所有* 节点读取并追加到日志中了，这条消息才能视为提交。这个 ISR 集合发生变化会在 ZooKeeper 持久化，正因为如此，这个集合中的任何一个节点都有资格被选为 leader 。这对于 Kafka 使用模型中， 有很多分区和并确保主从关系是很重要的。因为 ISR 模型和 *f+1* 副本，一个 Kafka topic 冗余 *f* 个节点故障而不会丢失任何已经提交的消息。

我们认为对于希望处理的大多数场景这种策略是合理的。在实际中，为了冗余 *f* 节点故障，大多数投票和 ISR 都会在提交消息前确认相同数量的备份被收到（例如在一次故障生存之后，大多数的 quorum 需要三个备份节点和一次确认，ISR 只需要两个备份节点和一次确认），多数投票方法的一个优点是提交时能避免最慢的服务器。但是，我们认为通过允许客户端选择是否阻塞消息提交来改善，和所需的备份数较低而产生的额外的吞吐量和磁盘空间是值得的。

另一个重要的设计区别是，Kafka 不要求崩溃的节点恢复所有的数据，在这种空间中的复制算法经常依赖于存在 “稳定存储”，在没有违反潜在的一致性的情况下，出现任何故障再恢复情况下都不会丢失。 这个假设有两个主要的问题。首先，我们在持久性数据系统的实际操作中观察到的最常见的问题是磁盘错误，并且它们通常不能保证数据的完整性。其次，即使磁盘错误不是问题，我们也不希望在每次写入时都要求使用 fsync 来保证一致性， 因为这会使性能降低两到三个数量级。我们的协议能确保备份节点重新加入ISR 之前，即使它挂时没有新的数据, 它也必须完整再一次同步数据。

Unclean leader 选举: 如果节点全挂了?

请注意，Kafka 对于数据不会丢失的保证，是基于至少一个节点在保持同步状态，一旦分区上的所有备份节点都挂了，就无法保证了。

但是，实际在运行的系统需要去考虑假设一旦所有的备份都挂了，怎么去保证数据不会丢失，这里有两种实现的方法

1. 等待一个 ISR 的副本重新恢复正常服务，并选择这个副本作为领 leader （它有极大可能拥有全部数据）。
2. 选择第一个重新恢复正常服务的副本（不一定是 ISR 中的）作为leader。

这是可用性和一致性之间的简单妥协，如果我只等待 ISR 的备份节点，那么只要 ISR 备份节点都挂了，我们的服务将一直会不可用，如果它们的数据损坏了或者丢失了，那就会是长久的宕机。另一方面，如果不是 ISR 中的节点恢复服务并且我们允许它成为 leader ， 那么它的数据就是可信的来源，即使它不能保证记录了每一个已经提交的消息。 kafka 默认选择第二种策略，当所有的 ISR 副本都挂掉时，会选择一个可能不同步的备份作为 leader ，可以配置属性 unclean.leader.election.enable 禁用此策略，那么就会使用第 一种策略即停机时间优于不同步。

这种困境不只有 Kafka 遇到，它存在于任何 quorum-based 规则中。例如，在大多数投票算法当中，如果大多数服务器永久性的挂了，那么您要么选择丢失100%的数据，要么违背数据的一致性选择一个存活的服务器作为数据可信的来源。

可用性和持久性保证

向 Kafka 写数据时，producers 设置 ack 是否提交完成， 0：不等待broker返回确认消息,1: leader保存成功返回或, -1(all): 所有备份都保存成功返回.请注意. 设置 “ack = all” 并不能保证所有的副本都写入了消息。默认情况下，当 acks = all 时，只要 ISR 副本同步完成，就会返回消息已经写入。例如，一个 topic 仅仅设置了两个副本，那么只有一个 ISR 副本，那么当设置acks = all时返回写入成功时，剩下了的那个副本数据也可能数据没有写入。 尽管这确保了分区的最大可用性，但是对于偏好数据持久性而不是可用性的一些用户，可能不想用这种策略，因此，我们提供了两个topic 配置，可用于优先配置消息数据持久性：

1. 禁用 unclean leader 选举机制 - 如果所有的备份节点都挂了,分区数据就会不可用，直到最近的 leader 恢复正常。这种策略优先于数据丢失的风险， 参看上一节的 unclean leader 选举机制。
2. 指定最小的 ISR 集合大小，只有当 ISR 的大小大于最小值，分区才能接受写入操作，以防止仅写入单个备份的消息丢失造成消息不可用的情况，这个设置只有在生产者使用 acks = all 的情况下才会生效，这至少保证消息被 ISR 副本写入。此设置是一致性和可用性 之间的折衷，对于设置更大的最小ISR大小保证了更好的一致性，因为它保证将消息被写入了更多的备份，减少了消息丢失的可能性。但是，这会降低可用性，因为如果 ISR 副本的数量低于最小阈值，那么分区将无法写入。

备份管理

以上关于备份日志的讨论只涉及单个日志文件，即一个 topic 分区，事实上，一个Kafka集群管理着成百上千个这样的 partitions。我们尝试以轮询调度的方式将集群内的 partition 负载均衡，避免大量topic拥有的分区集中在 少数几个节点上。同样，我们也试图平衡leadership,以至于每个节点都是部分 partition 的leader节点。

优化主从关系的选举过程也是重要的，这是数据不可用的关键窗口。原始的实现是当有节点挂了后，进行主从关系选举时，会对挂掉节点的所有partition 的领导权重新选举。相反，我们会选择一个 broker 作为“controller”节点。controller 节点负责 检测 brokers 级别故障,并负责在 broker 故障的情况下更改这个故障 Broker 中的 partition 的 leadership 。这种方式可以批量的通知主从关系的变化，使得对于拥有大量partition 的broker ,选举过程的代价更低并且速度更快。如果 controller 节点挂了，其他 存活的 broker 都可能成为新的 controller 节点。

4.8 日志压缩

日志压缩可确保 Kafka 始终至少为单个 topic partition 的数据日志中的每个 message key 保留最新的已知值。 这样的设计解决了应用程序崩溃、系统故障后恢复或者应用在运行维护过程中重启后重新加载缓存的场景。 接下来让我们深入讨论这些在使用过程中的更多细节，阐述在这个过程中它是如何进行日志压缩的。

迄今为止，我们只介绍了简单的日志保留方法（当旧的数据保留时间超过指定时间、日志大达到规定大小后就丢弃）。 这样的策略非常适用于处理那些暂存的数据，例如记录每条消息之间相互独立的日志。 然而在实际使用过程中还有一种非常重要的场景——根据key进行数据变更（例如更改数据库表内容），使用以上的方式显然不行。

让我们来讨论一个关于处理这样的流式数据的具体的例子。 假设我们有一个topic，里面的内容包含用户的email地址；每次用户更新他们的email地址时，我们发送一条消息到这个topic，这里使用用户Id作为消息的key值。 现在，我们在一段时间内为id为123的用户发送一些消息，每个消息对应email地址的改变（其他ID消息省略）：

```
123 => bill@microsoft.com
        .
        .
        .
123 => bill@gatesfoundation.org
        .
        .
        .
123 => bill@gmail.com
```

日志压缩为我提供了更精细的保留机制，所以我们至少保留每个key的最后一次更新 （例如：bill@gmail.com）。 这样我们保证日志包含每一个key的最终值而不只是最近变更的完整快照。这意味着下游的消费者可以获得最终的状态而无需拿到所有的变化的消息信息。

让我们先看几个有用的使用场景，然后再看看如何使用它。

1. *数据库更改订阅*。 通常需要在多个数据系统设置拥有一个数据集，这些系统中通常有一个是某种类型的数据库（无论是RDBMS或者新流行的key-value数据库）。 例如，你可能有一个数据库，缓存，搜索引擎集群或者Hadoop集群。每次变更数据库，也同时需要变更缓存、搜索引擎以及hadoop集群。 在只需处理最新日志的实时更新的情况下，你只需要最近的日志。但是，如果你希望能够重新加载缓存或恢复搜索失败的节点，你可能需要一个完整的数据集。
2. *事件源*。 这是一种应用程序设计风格，它将查询处理与应用程序设计相结合，并使用变更的日志作为应用程序的主要存储。
3. *日志高可用*。 执行本地计算的进程可以通过注销对其本地状态所做的更改来实现容错，以便另一个进程可以重新加载这些更改并在出现故障时继续进行。 一个具体的例子就是在流查询系统中进行计数，聚合和其他类似“group by”的操作。实时流处理框架Samza， 使用这个特性正是出于这一原因。

在这些场景中，主要需要处理变化的实时feed，但是偶尔当机器崩溃或需要重新加载或重新处理数据时，需要处理所有数据。 日志压缩允许在同一topic下同时使用这两个用例。这种日志使用方式更详细的描述请看这篇博客。

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

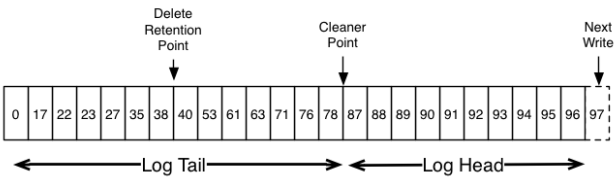
日志压缩机制是更细粒度的、每个记录都保留的机制，而不是基于时间的粗粒度。 这个理念是选择性的删除那些有更新的变更的记录的日志。 这样最终日志至少包含每个key的记录的最后一个状态。

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

这个功能的灵感来自于LinkedIn的最古老且最成功的基础设置———一个称为Databus的数据库变更日志缓存系统。 不像大多数的日志存储系统，Kafka是专门为订阅和快速线性的读和写的组织数据。 和Databus不同，Kafka作为真实的存储，压缩日志是非常有用的，这非常有利于上游数据源不能重放的情况。

日志压缩基础

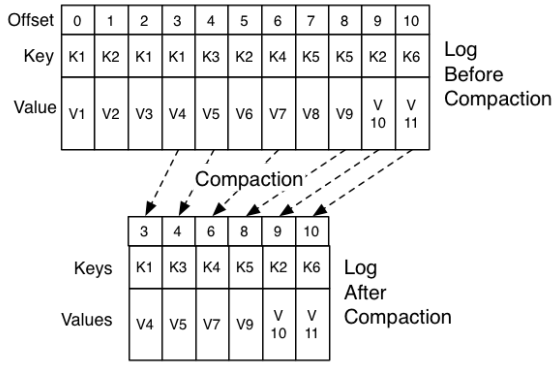
这是一个高级别的日志逻辑图，展示了kafka日志的每条消息的offset逻辑结构。



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

压缩也允许删除。通过消息的key和空负载（null payload）来标识该消息可从日志中删除。 这个删除标记将会引起所有之前拥有相同key的消息被移除（包括拥有key相同的新消息）。 但是删除标记比较特殊，它将在一定周期后被从日志中删除来释放空间。这个时间点被称为“delete retention point”，如上图。

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



What guarantees does log compaction provide?

日志压缩的保障措施如下：

1. 任何滞留在日志head中的所有消费者能看到写入的所有消息；这些消息都是有序的offset。 topic使用min.compaction.lag.ms来保障消息写入之前必须经过的最小时间长度，才能被压缩。 这限制了一条消息在Log Head中的最短存在时间。
2. 始终保持消息的有序性。压缩永远不会重新排序消息，只是删除了一些。
3. 消息的Offset不会变更。这是消息在日志中的永久标志。
4. 任何从头开始处理日志的Consumer至少会拿到每个key的最终状态。 另外，只要Consumer在小于Topic的delete.retention.ms设置（默认24小时）的时间段内到达Log head，将会看到所有删除记录的所有删除标记。 换句话说，因为移除删除标记和读取是同时发生的，Consumer可能会因为落后超过delete.retention.ms而导致错过删除标记。

日志压缩的细节

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

1. 选择log head与log tail比率最高的日志。
2. 在head log中为每个key的最后offset创建一个的简单概要。
3. 它从日志的开始到结束，删除那些在日志中最新出现的key的旧的值。新的、干净的日志将会立即被交到到日志中，所以只需要一个额外的日志段空间（不是日志的完整副本）
4. 日志head的概要本质上是一个空间密集型的哈希表，每个条目使用24个字节。所以如果有8G的整理缓冲区， 则能迭代处理大约366G的日志头部(假设消息大小为1k)。

配置Log Cleaner

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

```
1  log.cleanup.policy=compact
```

这个可以通过创建Topic时配置或者之后使用Topic命令实现。

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

```
1  log.cleaner.min.compaction.lag.ms
```

这可以保证消息在配置的时长内不被压缩。 如果没有设置，除了最后一个日志外，所有的日志都会被压缩。活动的 segment 是不会被压缩的，即使它保存的消息的滞留时长已经超过了配置的最小压缩时间长。

关于cleaner更详细的配置在 这里。

4.9 Quotas 配额

Kafka 集群可以对客户端请求进行配额，控制集群资源的使用。Kafka broker 可以对客户端做两种类型资源的配额限制，同一个group的client 共享配额。

1. 定义字节率的阈值来限定网络带宽的配额。 (从 0.9 版本开始)
2. request 请求率的配额，网络和 I/O线程 cpu利用率的百分比。 (从 0.11 版本开始)

为什么要对资源进行配额?

producers 和 consumers 可能会生产或者消费大量的数据或者产生大量的请求，导致对 broker 资源的垄断，引起网络的饱和，对其他clients和brokers本身造成DOS攻击。 资源的配额保护可以有效防止这些问题，在大型多租户集群中，因为一小部分表现不佳的客户端降低了良好的用户体验，这种情况下非常需要资源的配额保护。 实际情况中，当把Kafka当做一种服务提供的时候，可以根据客户端和服务端的契约对 API 调用做限制。

Client groups

Kafka client 是一个用户的概念， 是在一个安全的集群中经过身份验证的用户。在一个支持非授权客户端的集群中，用户是一组非授权的 users，broker使用一个可配置的 `PrincipalBuilder` 类来配置 group 规则。Client-id 是客户端的逻辑分组，客户端应用使用一个有意义的名称进行标识。(user, client-id)元组定义了一个安全的客户端逻辑分组，使用相同的user 和 client-id 标识。

资源配额可以针对 （user,client-id），users 或者client-id groups 三种规则进行配置。对于一个请求连接，连接会匹配最细化的配额规则的限制。同一个 group 的所有连接共享这个 group 的资源配额。 举个例子，如果 (user="test-user", client-id="test-client") 客户端producer 有10MB/sec 的生产资源配置，这10MB/sec 的资源在所有 "test-user" 用户，client-id是 "test-client" 的producer实例中是共享的。

Quota Configuration（资源配额的配置）

资源配额的配置可以根据 (user, client-id)，user 和 client-id 三种规则进行定义。在配额级别需要更高（或者更低）的配额的时候，是可以覆盖默认的配额配置。 这种机制和每个 topic 可以自定义日志配置属性类似。 覆盖 User 和 (user, client-id) 规则的配额配置会写到zookeeper的 ***/config/users***路径下，client-id 配额的配置会写到 ***/config/clients*** 路径下。 这些配置的覆盖会被所有的 brokers 实时的监听到并生效。所以这使得我们修改配额配置不需要重启整个集群。更多细节参考 here。 每个 group 的默认配额可以使用相同的机制进行动态更新。

配额配置的优先级顺序是:

1. /config/users/<user>/clients/<client-id>
2. /config/users/<user>/clients/<default>
3. /config/users/<user>
4. /config/users/<default>/clients/<client-id>
5. /config/users/<default>/clients/<default>
6. /config/users/<default>
7. /config/clients/<client-id>
8. /config/clients/<default>

Broker 的配置属性 (quota.producer.default, quota.consumer.default) 也可以用来设置 client-id groups 默认的网络带宽配置。这些配置属性在未来的 release 版本会被 deprecated。 client-id 的默认配额也是用zookeeper配置，和其他配额配置的覆盖和默认方式是相似的。

Network Bandwidth Quotas（网络带宽配额配置）

网络带宽配额使用字节速率阈值来定义每个 group 的客户端的共享配额。 默认情况下，每个不同的客户端 group 是集群配置的固定配额，单位是 bytes/sec。 这个配额会以broker 为基础进行定义。在 clients 被限制之前，每个 group 的clients可以发布和拉取单个broker 的最大速率，单位是 bytes/sec。

Request Rate Quotas 请求速率配额

请求速率的配额定义了一个客户端可以使用 broker request handler I/O 线程和网络线程在一个配额窗口时间内使用的百分比。 `n%` 的配置代表一个线程的 `n%`的使用率，所以这种配额是建立在总容量 `((num.io.threads + num.network.threads) * 100)%`之上的. 每个 group 的client 的资源在被限制之前可以使用单位配额时间窗口内I/O线程和网络线程利用率的 `n%`。 由于分配给 I/O和网络线程的数量是基于 broker 的核数，所以请求量的配额代表每个group 的client 使用cpu的百分比。

Enforcement（限制）

默认情况下，集群给每个不同的客户端group 配置固定的配额。 这个配额是以broker为基础定义的。每个 client 在受到限制之前可以利用每个broker配置的配额资源。 我们觉得给每个broker配置资源配额比为每个客户端配置一个固定的集群带宽资源要好，为每个客户端配置一个固定的集群带宽资源需要一个机制来共享 client 在brokers上的配额使用情况。这可能比配额本身实现更难。

broker在检测到有配额资源使用违反规则会怎么办？在我们计划中，broker不会返回error，而是会尝试减速 client 超出的配额设置。 broker 会计算出将客户端限制到配额之下的延迟时间，并且延迟response响应。这种方法对于客户端来说也是透明的（客户端指标除外）。这也使得client不需要执行任何特殊的 backoff 和 retry 行为。而且不友好的客户端行为（没有 backoff 的重试）会加剧正在解决的资源配额问题。

网络字节速率和线程利用率可以用多个小窗口来衡量（例如 1秒30个窗口），以便快速的检测和修正配额规则的违反行为。实际情况中 较大的测量窗口（例如，30秒10个窗口）会导致大量的突发流量，随后长时间的延迟，会使得用户体验不是很好。

5. 实现思路

5.1 网络层

网络层相当于一个 NIO 服务,在此不在详细描述. sendfile(零拷贝) 的实现是通过 `MessageSet` 接口的 `writeTo` 方法完成的.这样的机制允许 file-backed 集使用更高效的 `transferTo` 实现,而不在使用进程内的写缓存.线程模型是一个单独的接受线程和 N 个处理线程,每个线程处理固定数量的连接.这种设计方式在其他地方经过大量的测试,发现它是实现简单而且快速的.协议保持简单以允许未来实现其他语言的客户端.

5.2 消息

消息包含一个可变长度的 header ,一个可变长度不透明的字节数组 key ,一个可变长度不透明的字节数组 value , 消息中 header 的格式会在下一节描述. 保持消息中的 key 和 value 不透明(二进制格式)是正确的决定: 目前构建序列化库取得很大的进展,而且任何单一的序列化方式都不能满足所有的用途.毋庸置疑,使用kafka的特定应用程序可能会要求特定的序列化类型作为自己使用的一部分. `RecordBatch` 接口就是一种简单的消息迭代器,它可以使用特定的方法批量读写消息到 NIO 的 `Channel` 中.

5.3 消息格式

消息通常按照批量的方式写入.record batch 是批量消息的技术术语,它包含一条或多条 records.不良情况下, record batch 只包含一条 record. Record batches 和 records 都有他们自己的 headers.在 kafka 0.11.0及后续版本中(消息格式的版本为 v2 或者 magic=2)解释了每种消息格式.点击查看消息格式详情.

5.3.1 Record Batch

以下为 RecordBatch 在硬盘上的格式.

```
baseOffset: int64
batchLength: int32
partitionLeaderEpoch: int32
magic: int8 (current magic value is 2)
crc: int32
attributes: int16
    bit 0~2:
        0: no compression
        1: gzip
        2: snappy
        3: lz4
    bit 3: timestampType
    bit 4: isTransactional (0 means not transactional)
    bit 5: isControlBatch (0 means not a control batch)
    bit 6~15: unused
lastOffsetDelta: int32
firstTimestamp: int64
maxTimestamp: int64
producerId: int64
producerEpoch: int16
baseSequence: int32
records: [Record]

```

请注意,启用压缩时，压缩的记录数据将直接按照记录数进行序列化。

CRC(一种数据校验码) 会覆盖从属性到批处理结束的数据, (即 CRC 后的所有字节数据). CRC 位于 magic 之后,这意味着,在决定如何解释批次的长度和 magic 类型之前,客户端需要解析 magic 类型.CRC 计算不包括分区 leader epoch 字段,是为了避免 broker 收到每个批次的数据时 需要重新分配计算 CRC . CRC-32C (Castagnoli) 多项式用于计算.

压缩: 不同于旧的消息格式, magic v2 及以上版本在清理日志时保留原始日志中首次及最后一次 offset/sequence .这是为了能够在日志重新加载时恢复生产者的状态.例如,如果我们不保留最后一次序列号,当分区 leader 失败以后,生产者会报 OutOfSequence 的错误.必须保留基础序列号来做重复检查(broker 通过检查生产者该批次请求中第一次及最后一次序列号是否与上一次的序列号相匹配来判断是否重复).因此,当批次中所有的记录被清理但批次数据依然保留是为了保存生产者最后一次的序列号,日志中可能有空的数据.不解的是在压缩中时间戳可能不会被保留,所以如果批次中的第一条记录被压缩,时间戳也会改变

5.3.1.1 批次控制

批次控制包含成为控制记录的单条记录. 控制记录不应该传送给应用程序,相反,他们是消费者用来过滤中断的事务消息.

控制记录的 key 符合以下模式:

```
version: int16 (current version is 0)
type: int16 (0 indicates an abort marker, 1 indicates a commit)
```

批次记录值的模式依赖于类型. 对客户端来说它是透明的.

5.3.2 Record(记录)

Record level headers were introduced in Kafka 0.11.0. The on-disk format of a record with Headers is delineated below.

Record 级别的头部信息在0.11.0 版本引入. 拥有 headers 的 Record 的磁盘格式如下.

```
length: varint
attributes: int8
    bit 0~7: unused
timestampDelta: varint
offsetDelta: varint
keyLength: varint
key: byte[]
valueLen: varint
value: byte[]
Headers => [Header]

```

5.4.2.1 Record Header

```
headerKeyLength: varint
headerKey: String
headerValueLength: varint
Value: byte[]

```

我们使用了和 Protobuf 编码格式相同的 varint 编码. 更多后者相关的信息 在这里. Record 中 headers 的数量也被编码为 varint .

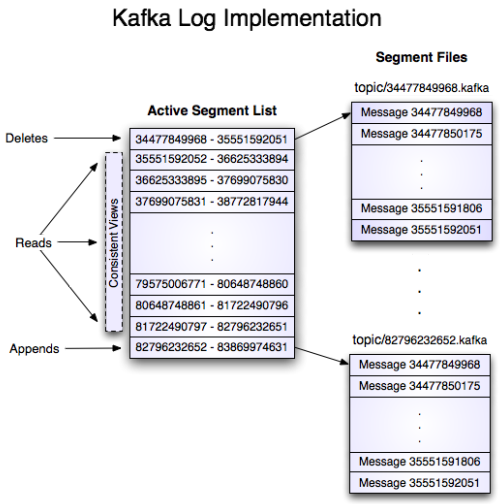
5.4 日志

命名为 "my_topic" 的主题日志有两个分区,包含两个目录 (命名为 `my_topic_0` 和 `my_topic_1`),目录中分布着包含该 topic 消息的日志文件.日志文件的格式是 "log entries" 的序列; 每个日志对象是由4位的数字*N*存储日志长度,后跟 *N* 字节的消息.每个消息使用64位的整数作为 *offset* 唯一标记, offset 即为发送到该 topic partition 中所有流数据的起始位置.每个消息的磁盘格式如下. 每个日志文件使用它包含的第一个日志的 offset

来命名.所以创建的第一个文件是 00000000000.kafka, 并且每个附件文件会有大概 *S* 字节前一个文件的整数名称,其中 *S* 是配置给出的最大文件大小.

记录的精确二进制格式是版本化的,并且按照标准接口进行维护,所以批量的记录可以在 producer, broker 和客户端之间传输,而不需要在使用时进行重新复制或转化.前一章包含了记录的磁盘格式的详情.

消息的偏移量用作消息 id 是不常见的.我们最开始的想法是使用 producer 自增的 GUID ,并维护从 GUID 到每个 broker 的 offset 的映射.这样的话每个消费者需要为每个服务端维护一个 ID,提供全球唯一的 GUID 没有意义.而且,维护一个从随机 ID 到偏移量映射的复杂度需要一个重度的索引结构,它需要与磁盘进行同步,本质上需要一个完整的持久随机访问数据结构.因此为了简化查找结构,我们决定针对每个分区使用一个原子计数器,它可以利用分区id和节点id唯一标识一条消息.虽然这使得查找结构足够简单,但每个消费者的多个查询请求依然是相似的.一旦我们决定使用使用计数器,直接跳转到对应的偏移量显得更加自然-毕竟对于每个分区来说它们都是一个单调递增的整数.由于消费者API隐藏了偏移量，所以这个决定最终是一个实现细节，我们采用了更高效的方法。



Writes

日志允许序列化的追加到最后一个文件中.当文件大小达到配置的大小(默认 1G)时,会生成一个新的文件.日志中有两个配置参数: *M* 是在 OS 强制写文件到磁盘之前的消息条数, *S* 是强制写盘的秒数.这提供了一个在系统崩溃时最多丢失 *M* 条或者 *S* 秒消息的保证.

Reads

通过提供消息的64位逻辑偏移量和 *S* 位的 max chunk size 完成读请求.这会返回一个包含 *S* 位的消息缓存迭代器. *S* 必须大于任何单条的数据,但是在异常的大消息情况下,读取操作可以重试多次,每次会加倍缓冲的大小,直到消息被读取成功.可以指定最大消息大小和缓存大小使服务器拒绝接收超过其大小的消息,并为客户端设置消息的最大限度,它需要尝试读取多次获得完整的消息.读取缓冲区可能以部分消息结束,这很容易通过大小分界来检测.

按照偏移量读取的实际操作需要在数据存储目录中找到第一个日志分片的位置,在全局的偏移量中计算指定文件的偏移量,然后读取文件偏移量.搜索是使用二分查找法查找在内存中保存的每个文件的偏移量来完成的.

日志提供了将消息写入到当前的能力,以允许客户端从'当前开始订阅.在消费者未能在其SLA指定的天数内消费其数据的情况下,这也是有用的.在这种情况下,客户端会尝试消费不存在的偏移量的数据,这会抛出 OutOfRangeException 异常,并且也会重置 offset 或者失败.

以下是发送给消费者的结果格式.

```
MessageSetSend (fetch result)

total length     : 4 bytes
error code       : 2 bytes
message 1        : x bytes
...
message n        : x bytes
```

```
MultiMessageSetSend (multiFetch result)

total length       : 4 bytes
error code         : 2 bytes
messageSetSend 1
...
messageSetSend n
```

Deletes

在一个时点下只有一个 log segment 的数据能被删除。日志管理器允许使用可插拔的删除策略来选择哪些文件符合删除条件.当前的删除策略会删除 *N* 天之前改动的日志,尽管保留最后的 *N* GB 数据可能有用.为了避免锁定读，同时允许删除修改 segment 列表，我们使用 copy-on-write 形式的 segment 列表实现，在删除的同时它提供了一致的视图允许在多个 segment 列表视图上执行二进制的搜索。

Guarantees

日志提供了配置项 *M*，它控制了在强制刷盘之前的最大消息数。启动时，日志恢复线程会运行，对最新的日志片段进行迭代，验证每条消息是否合法。如果消息对象的总数和偏移量小于文件的长度并且 消息数据包的 CRC32 校验值与存储在消息中的 CRC 校验值相匹配的话，说明这个消息对象是合法的。如果检测到损坏，日志会在最后一个合法 offset 处截断。

请注意，有两种损坏必须处理：由于崩溃导致的未写入的数据块的丢失和将无意义已损坏的数据块添加到文件。原因是：通常系统不能保证文件索引节点和实际数据快之间的写入顺序，除此之外，如果在块数据被写入之前，文件索引已更新为新的大小，若此时系统崩溃，文件不会的到有意义的数据，则会导致数据丢失。

5.5 分布式

Consumer Offset Tracking（消费者offset跟踪）

高级别的consumer跟踪每个分区已消费的offset，并定期提交，以便在重启的情况下可以从这些offset中恢复。Kafka提供了一个选项在指定的broker中来存储所有给定的consumer组的offset，称为offset manager。例如，该consumer组的所有consumer实例向offset manager（broker）发送提交和获取offset请求。高级别的consumer将会自动处理这些过程。如果你使用低级别的consumer，你将需要手动管理offset。目前在低级别的java consumer中不支持，只能在Zookeeper中提交或获取offset。如果你使用简单的Scala consumer，将可拿到offset manager，并显式的提交或获取offset。对于包含offset manager的consumer可以通过发送GroupCoordinatorRequest到任意kafka broker，并接受GroupCoordinatorResponse响应，consumer可以继续向`offset manager broker`提交或获取offset。如果offset manager位置变动，consumer需要重新发现offset

manager。如果你想手动管理你的offset，你可以看看OffsetCommitRequest 和 OffsetFetchRequest的源码是如何实现的。

当offset manager接收到一个OffsetCommitRequest，它将追加请求到一个特定的压缩名为__consumer_offsets的kafka topic中，当offset topic的所有副本接收offset之后，offset manager将发送一个提交offset成功的响应给consumer。万一offset无法在规定的时间内复制，offset将提交失败，consumer在回退之后可重试该提交（高级别consumer自动进行）。broker会定期压缩offset topic，因为只需要保存每个分区最近的offset。offset manager会缓存offset在内存表中，以便offset快速获取。

当offset manager接收一个offset的获取请求，将从offset缓存中返回最新的的offset。如果offset manager刚启动或新的consumer组刚成为offset manager（成为offset topic分区的leader），则需要加载offset topic的分区到缓存中，在这种情况下，offset将获取失败，并报出OffsetsLoadInProgress异常，consumer回滚后，重试OffsetFetchRequest（高级别consumer自动进行这些操作）。

从ZooKeeper迁移offset到kafka

Kafka consumers在早先的版本中offset默认存储在ZooKeeper中。可以通过下面的步骤迁移这些consumer到Kafka：

1. 在consumer配置中设置`offsets.storage=kafka`和`dual.commit.enabled=true`。
2. consumer做滚动消费，验证你的consumer是健康正常的。
3. 在你的consumer配置中设置`dual.commit.enabled=false`。
4. consumer做滚动消费，验证你的consumer是健康正常的。

回滚（就是从kafka回到Zookeeper）也可以使用上面的步骤，通过设置`offsets.storage=zookeeper`。

ZooKeeper 目录

下面给出了Zookeeper的结构和算法，用于协调consumer和broker。

Notation

当一个path中的元素表示为[XYZ]，这意味着xyz的值不是固定的，实际上每个xyz的值可能是Zookeeper的znode，例如`/topic/[topic]`是一个目录，/topic包含一个子目录(每个topic名称)。数字的范围如[0...5]来表示子目录0，1，2，3，4。箭头`->`用于表示znode的内容，例如:/hello->world表示znode /hello包含值”world”。

Broker节点注册

```
/brokers/ids/[0...N] --> {"jmx_port":...,"timestamp":...,"endpoints":[...],"host":...,"version":...,"port":...} (ephemeral node)
```

这是当前所有broker的节点列表，其中每个提供了一个唯一的逻辑broker的id标识它的consumer（必须作为配置的一部分）。在启动时，broker节点通过在/brokers/ids/下用逻辑broker id创建一个znode来注册它自己。

逻辑broker id的目的是当broker移动到不同的物理机器时，而不会影响消费者。尝试注册一个已存在的broker id时将返回错误（因为2个server配置了相同的broker id）。

由于broker在Zookeeper中用的是临时znode来注册，因此这个注册是动态的，如果broker关闭或宕机，节点将消失（通知consumer不再可用）。

Broker Topic 注册

```
1  /brokers/topics/[topic]/partitions/[0...N]/state --> {"controller_epoch":...,"leader":...,"version":...,"leader_epoch":...,"isr":[...]} (ephemeral node)
```

每个broker在它自己的topic下注册，维护和存储该topic分区的数据。

Consumers and Consumer Groups

topic的consumer也在zookeeper中注册自己，以便相互协调和平衡数据的消耗。consumer也可以通过设置offsets.storage = zookeeper将他们的偏移量存储在zookeeper中。但是，这个偏移存储机制将在未来的版本中被弃用。因此，建议将数据迁移到kafka中。

多个consumer可组成一组，共同消费一个topic，在同一组中的每个consumer共享一个group_id。例如，如果一个consumer是foobar，在三台机器上运行，你可能分配这个这个consumer的ID是“foobar”。这个组id是在consumer的配置文件中配置的。

每个分区正好被一个consumer组的consumer所消费，一组中的consumer尽可能公平地分配分区。

Consumer Id 注册

除了由所有consumer共享的group_id，每个consumer都有一个临时且唯一的consumer_id（主机名的形式:uuid）用于识别。consumer的id在以下目录中注册。

```
1  /consumers/[group_id]/ids/[consumer_id] --> {"version":...,"subscription":{...:...},"pattern":...,"timestamp":...} (ephemeral node)
```

组中的每个consumer用consumer_id注册znode。znode的值包含一个map。这个id只是用来识别在组里目前活跃的consumer，这是个临时节点，如果consumer在处理中挂掉，它就会消失。

Consumer Offsets

消费者跟踪他们在每个分区中消耗的最大偏移量。 如果offsets.storage = zookeeper，则此值存储在ZooKeeper目录中。

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

Partition Owner registry

每个代理分区由给定消费组中的单个消费者使用。 在消费开始之前，消费者必须确定分区的所有权。为了建立对它的所有权，消费者将自己的ID写在它声明所有权的特定经纪人分区下的临时节点中。

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

Cluster Id

集群ID是分配给Kafka集群的唯一且不可变的标识符。 集群ID最多可以有22个字符，允许的字符是由正则表达式[a-zA-Z0-9 _ \ -] +定义的，对应于不带填充的URL安全的Base64变体使用的字符。 从概念上讲，它是在第一次启动集群时自动生成的。

当第一次成功启动版本为0.10.1或更高版本的broker时生成。 broker尝试在启动期间从/cluster/id znode获取集群标识。如果znode不存在，那么代理会生成一个新的集群标识并使用此集群标识并创建znode。

Broker node registration

Broker节点基本上是独立的，所以他们只发布他们有什么信息。 Broker加入时，会在Broker节点注册表目录下进行注册，并写入有关其主机名和端口的信息。 Broker还在broker topic registry中注册现有主题及其逻辑分区的列表。 在代理上创建新主题时会动态注册。

Consumer registration algorithm

当消费者启动时，它执行以下操作：

在其消费组的consumer ID注册中注册自己。

在 consumer ID注册表下注册关于更改（新消费者加入或任何现有的消费者离开）的监听。 （每次

更改都会触发更改的消费者所属组中的所有消费者之间的再平衡。）

在经纪人ID注册表下注册一个监视变更（新的经纪人加入或任何现有的经纪人离开）。 （每个变化

都会触发所有消费群体中的所有消费者重新平衡。）

如果消费者使用主题过滤器创建消息流，则还会在 broker topic registry下注册关于更改（新增主

题）的监听。 （每次更改都会触发对可用主题的重新评估，以确定主题过滤器允许哪些主题。新的

允许主题将触发消费者组中所有消费者之间的重新平衡）。

强迫自己在消费组内重新平衡。

消费者重新平衡算法

消费者重新平衡算法允许组中的所有消费者就哪个消费者正在消费哪个分区达成共识。 消费者重新平衡是在每次添加或删除代理节点或者同一组内的其他消费者时被触发。 对于给定的主题和给定的消费者组，broker分区在组内的消费者之间平均分配。 分区总是由单个消费者使用。 这个设计简化了实现。 如果我们允许一个分区被多个消费者同时使用，那么分区上就会出现竞争，并且需要某种锁定。 如果消费者比分区多，一些消费者就根本得不到任何数据。 在重新平衡期间，我们尝试以这种方式将分区分配给消费者，以减少每个消费者必须连接的代理节点的数量。

每位消费者在重新平衡期间执行以下操作

```
1. For each topic T that C<sub>i</sub> subscribes to
2.   let P<sub>T</sub> be all partitions producing topic T
3.   let C<sub>G</sub> be all consumers in the same group as C<sub>i</sub> that consume topic T
4.   sort P<sub>T</sub> (so partitions on the same broker are clustered together)
5.   sort C<sub>G</sub>
6.   let i be the index position of C<sub>i</sub> in C<sub>G</sub> and let N = size(P<sub>T</sub>)/size(C<sub>G</sub>)
7.   assign partitions from i*N to (i+1)*N - 1 to consumer C<sub>i</sub>
8.   remove current entries owned by C<sub>i</sub> from the partition owner registry
9.   add newly assigned partitions to the partition owner registry
10          (we may need to re-try this until the original partition owner releases its ownership)
```

.当一个消费者触发再平衡时，同一时间内同一群体内的其他消费者也重新平衡。

6. 基本操作

以下是基于LinkedIn使用Kafka作为生产系统的一些使用经验。如果您有其他好的技巧请告诉我们。

6.1 基础的 Kafka 操作

本节将回顾在Kafka集群上执行的最常见操作。所有在本节中看到的工具都可以在Kafka发行版的 `bin /` 目录下找到，如果没有参数运行，每个工具都会打印所有可能的命令行选项的细节。

添加和删除 `topics`

您可以选择手动添加 `topic` ，或者在数据首次发布到不存在的 `topic` 时自动创建 `topic` 。如果 `topic` 是自动创建的，那么您可能需要调整用于自动创建 `topic` 的默认 `topic 配置`。

使用 `topic` 工具来添加和修改 `topic` ：

```
1  > bin/kafka-topics.sh --zookeeper zk_host:port/chroot --create --topic my_topic_name
2        --partitions 20 --replication-factor 3 --config x=y
```

`replication-factor` 控制有多少服务器将复制每个写入的消息。如果您设置了3个复制因子，那么只能最多2个相关的服务器能出问题，否则您将无法访问数据。我们建议您使用2或3个复制因子，以便在不中断数据消费的情况下透明的调整集群。

`partitions` 参数控制 `topic` 将被分片到多少个日志里。`partitions` 会产生几个影响。首先，每个分区只属于一个台服务器，所以如果有20个分区，那么全部数据(包含读写负载)将由不超过20个服务器（不包含副本）处理。最后 `partitions` 还会影响 `consumer` 的最大并行度。这在概念部分中有更详细的讨论。

每个分区日志都放在自己的Kafka日志目录下的文件夹中。这些文件夹的名称由主题名称，破折号（ - ）和分区ID组成。由于典型的文件夹名称长度不能超过255个字符，所以主题名称的长度会受到限制。我们假设分区的数量不会超过10万。因此，主题名称不能超过249个字符。这在文件夹名称中留下了足够的空间以显示短划线和可能的5位长的分区ID。

在命令行上添加的配置会覆盖服务器的默认设置，例如数据应该保留的时间长度。此处记录了完整的每个 topic 配置。

修改 topics

使用相同的 topic 工具，您可以修改 topic 的配置或分区。

要添加分区，你可以做如下操作

```
> bin/kafka-topics.sh --zookeeper zk_host:port/chroot --alter --topic my_topic_name
      --partitions 40
```

请注意，分区的一个用处是对数据进行语义分区，并且添加分区不会更改现有数据的分区，因此如果依赖该分区，则可能会影响消费者。也就是说，如果数据是通过 hash（key）%number_of_partitions 进行分区的，那么这个分区可能会通过添加分区进行混洗，但Kafka不会尝试以任何方式自动重新分配数据。

增加一个配置项：

```
> bin/kafka-configs.sh --zookeeper zk_host:port/chroot --entity-type topics --entity-name my_topic_name --alter --add-config x=y
```

删除一个配置项：

```
> bin/kafka-configs.sh --zookeeper zk_host:port/chroot --entity-type topics --entity-name my_topic_name --alter --delete-config x
```

删除一个 topic ：

```
> bin/kafka-topics.sh --zookeeper zk_host:port/chroot --delete --topic my_topic_name
```

当前，Kafka 不支持减少一个 topic 的分区数。

有关更改 一个 topic 复制因子的说明，请参见此处.

优雅的关机

Kafka集群将自动检测到任何 broker 关机或故障，并为该机器上的分区选择新的 leader。无论服务器出现故障还是因为维护或配置更改而故意停机，都会发生这种情况。 对于后一种情况，Kafka支持更优雅的停止服务器的机制，而不仅仅是杀死它。 当一个服务器正常停止时，它将采取两种优化措施：

1. 它将所有日志同步到磁盘，以避免在重新启动时需要进行任何日志恢复活动（即验证日志尾部的所有消息的校验和）。由于日志恢复需要时间，所以从侧面加速了重新启动操作。
2. 它将在关闭之前将以该服务器为 leader 的任何分区迁移到其他副本。这将使 leader 角色传递更快，并将每个分区不可用的时间缩短到几毫秒。

只要服务器的停止不是通过直接杀死，同步日志就会自动发生，但控制 leader 迁移需要使用特殊的设置：

```
controlled.shutdown.enable=true
```

请注意，只有当 broker 托管的分区具有副本（即，复制因子大于1 *且至少其中一个副本处于活动状态）时，对关闭的控制才会成功。 这通常是你想要的，因为关闭最后一个副本会使 topic 分区不可用。*

Balancing leadership

每当一个 borker 停止或崩溃时，该 borker 上的分区的leader 会转移到其他副本。这意味着，在 broker 重新启动时，默认情况下，它将只是所有分区的跟随者，这意味着它不会用于客户端的读取和写入。

为了避免这种不平衡，Kafka有一个首选副本的概念。如果分区的副本列表为1,5,9，则节点1首选为节点5或9的 leader ，因为它在副本列表中较早。您可以通过运行以下命令让Kafka集群尝试恢复已恢复副本的领导地位：

```
1  > bin/kafka-preferred-replica-election.sh --zookeeper zk_host:port/chroot
```

由于运行此命令可能很乏味，您也可以通过以下配置来自动配置Kafka：

```
1  auto.leader.rebalance.enable=true
```

垮机架均衡副本

机架感知功能可以跨不同机架传播相同分区的副本。这扩展了 Kafka 为 broker 故障提供的容错担保，弥补了机架故障，如果机架上的所有 broker 都失败，则可以限制数据丢失的风险。该功能也可以应用于其他 broker 分组，例如EC2中的可用区域。

您可以通过向 broker 配置添加属性来指定 broker 属于的特定机架：

```
1  broker.rack=my-rack-id
```

当 topic 创建，修改或副本重新分配时， 机架约束将得到保证，确保副本跨越尽可能多的机架（一个分区将跨越 min(#racks, replication-factor) 个不同的机架）。

用于向 broker 分配副本的算法可确保每个 broker 的 leader 数量将保持不变，而不管 broker 在机架之间如何分布。这确保了均衡的吞吐量。

但是，如果 broker 在机架间分布不均 ，副本的分配将不均匀。具有较少 broker 的机架将获得更多复制副本，这意味着他们将使用更多存储并将更多资源投入复制。因此，每个机架配置相同数量的 broker 是明智的。

集群之间镜像数据

我们指的是通过“镜像”复制Kafka集群之间的数据的过程，以避免与在单个集群中的节点之间发生的复制混淆。Kafka附带了一个在Kafka集群之间镜像数据的工具 mirror-maker。该工具从源集群中消费数据并产生数据到目标集群。 这种镜像的常见用例是在另一个数据中心提供副本。这个场景将在下一节中详细的讨论。

您可以运行许多这样的镜像进程来提高吞吐量和容错能力（如果一个进程死亡，其他进程将承担额外的负载）。

从源群集中的 topic 中读取数据，并将其写入目标群集中具有相同名称的 topic。事实上，镜像只不过是把一个 Kafka 的 consumer 和 producer 联合在一起了。

源和目标集群是完全独立的实体：它们可以有不同数量的分区，偏移量也不会相同。由于这个原因，镜像集群并不是真正意义上的容错机制（因为 consumer 的偏移量将会不同）。为此，我们建议使用正常的群集内复制。然而，镜像制作进程将保留并使用消息 key 进行分区，所以在每个 key 的基础上保存顺序。

以下示例显示如何从输入群集中镜像单个 topic（名为 my-topic ）：

```
> bin/kafka-mirror-maker.sh
      --consumer.config consumer.properties
      --producer.config producer.properties --whitelist my-topic
```

请注意，我们使用 --whitelist 选项指定 topic 列表。此选项允许使用任何 Java风格的正则表达式 因此，您可以使用 --whitelist 'A|B' 来镜像名为A 和 B 的两个 topic 。 或者您可以使用 --whitelist '*' 来镜像全部 topic。确保引用的任何正则表达式不会被 shell 尝试将其展开为文件路径。为了方便起见，我们允许使用 ',' 而不是 '|' 指定 topic 列表。

有时，说出你不想要的东西比较容易。与使用 --whitelist 来表示你想要的相反，通过镜像您可以使用 --blacklist 来表示要排除的内容。 这也需要一个正则表达式的参数。但是，当启用新的 consumer 时，不支持 --blacklist （即 bootstrap.servers ）已在 consumer 配置中定义）。

将镜像与配置项 auto.create.topics.enable = true 结合使用，可以创建一个副本群集，即使添加了新的 topic，也可以自动创建和复制源群集中的所有数据。

检查 consumer 位置

有时观察到消费者的位置是有用的。我们有一个工具，可以显示 consumer 群体中所有 consumer 的位置，以及他们所在日志的结尾。要在名为my-group的 consumer 组上运行此工具，消费一个名为my-topic的 topic 将如下所示：

```
> bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --describe --group my-group

注意：这将仅显示使用Java consumer API（基于非ZooKeeper的 consumer）的 consumer 的信息。

TOPIC                          PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG        CONSUMER-ID                                       HOST                           C
my-topic                       0          2               4               2          consumer-1-029af89c-873c-4751-a720-cefd41a669d6   /127.0.0.1                     c
my-topic                       1          2               3               1          consumer-1-029af89c-873c-4751-a720-cefd41a669d6   /127.0.0.1                     c
my-topic                       2          2               3               1          consumer-2-42c1abd4-e3b2-425d-a8bb-e1ea49b29bb2   /127.0.0.1                     c
```

这个工具也适用于基于ZooKeeper的 consumer：

```
> bin/kafka-consumer-groups.sh --zookeeper localhost:2181 --describe --group my-group

注意：这只会显示关于使用ZooKeeper的 consumer 的信息（不是那些使用Java consumer API的消费者）。

TOPIC                          PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG        CONSUMER-ID
my-topic                       0          2               4               2          my-group_consumer-1
my-topic                       1          2               3               1          my-group_consumer-1
my-topic                       2          2               3               1          my-group_consumer-2
```

管理 Consumer 组

通过 ConsumerGroupCommand 工具，我们可以列出，描述或删除 consumer 组。请注意，删除仅在组元数据存储在ZooKeeper中时可用。当使用新的 consumer API （ broker 协调分区处理和重新平衡）时， 当该组的最后一个提交偏移量过期时，该组将被删除。 例如，要列出所有 topic 中的所有 consumer 组：

```
> bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --list

test-consumer-group
```

如前所述,为了查看偏移量，我们这样“describe”consumer 组：

```
> bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --describe --group test-consumer-group

TOPIC                          PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG        CONSUMER-ID                                       HOST                           C
test-foo                       0          1               3               2          consumer-1-a5d61779-4d04-4c50-a6d6-fb35d942642d   /127.0.0.1                     c
```

如果您正在使用老的高级 consumer 并在ZooKeeper中存储组元数据（即 offsets.storage = zookeeper ），则传递 --zookeeper 而不是bootstrap-server：

```
> bin/kafka-consumer-groups.sh --zookeeper localhost:2181 --list
```

扩展您的群集

将服务器添加到Kafka集群非常简单，只需为其分配唯一的 broker ID并在您的新服务器上启动Kafka即可。但是，这些新的服务器不会自动分配到任何数据分区，除非将分区移动到这些分区，否则直到创建新 topic 时才会提供服务。所以通常当您将机器添加到群集中时，您会希望将一些现有数据迁移到这些机器上。

迁移数据的过程是手动启动的，但是完全自动化。在迁移数据时，Kafka会将新服务器添加为正在迁移的分区的 follower，并允许它完全复制该分区中的现有数据。当新服务器完全复制了此分区的内容并加入了同步副本时，其中一个现有副本将删除其分区的数据。

分区重新分配工具可用于跨 broker 移动分区。理想的分区分布将确保所有 broker 的数据负载和分区大小比较均衡。分区重新分配工具不具备自动分析Kafka集群中的数据分布并移动分区以获得均匀负载的功能。因此，管理员必须找出哪些 topic 或分区应该移动。

分区重新分配工具可以以3种互斥方式运行：

- --generate: 在此模式下，给定一个 topic 列表和一个 broker 列表，该工具会生成一个候选重新分配，以将指定的 topic 的所有分区移动到新的broker。此选项仅提供了一种便捷的方式，可以根据 tpoc 和目标 broker 列表生成分区重新分配计划。
- --execute: 在此模式下，该工具基于用户提供的重新分配计划启动分区重新分配。（使用--reassignment-json-file选项）。这可以是由管理员制作的自定义重新分配计划，也可以是使用--generate选项提供的自定义重新分配计划。
- --verify: 在此模式下，该工具将验证最近用 --execute 模式执行间的所有分区的重新分配状态。状态可以是成功完成，失败或正在进行

自动将数据迁移到新机器

分区重新分配工具可用于将当前一组 broker 的一些 topic 移至新增的topic。这在扩展现有群集时通常很有用，因为将整个 topic 移动到新 broker 集比移动一个分区更容易。当这样做的时候，用户应该提供需要移动到新的 broker 集合的 topic 列表和新的目标broker列表。该工具然后会均匀分配新 broker 集中 topic 的所有分区。在此过程中，topic 的复制因子保持不变。实际上，所有输入 topic 的所有分区副本都将从旧的broker 组转移到新 broker中。

例如，以下示例将把名叫foo1，foo2的 topic 的所有分区移动到新的 broker 集5,6。最后，foo1和foo2的所有分区将只在<5,6> broker 上存在。

由于该工具接受由 topic 组成的输入列表作为json文件，因此首先需要确定要移动的 topic 并创建 json 文件，如下所示：

```
> cat topics-to-move.json
{"topics": [{"topic": "foo1"},
            {"topic": "foo2"}],
 "version":1
 }
```

一旦json文件准备就绪，就可以使用分区重新分配工具来生成候选分配：

```
> bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --topics-to-move-json-file topics-to-move.json --broker-list "5,6" --generate
当前分区副本分配

{"version":1,
"partitions":[{"topic":"foo1","partition":2,"replicas":[1,2]},
              {"topic":"foo1","partition":0,"replicas":[3,4]},
              {"topic":"foo2","partition":2,"replicas":[1,2]},
              {"topic":"foo2","partition":0,"replicas":[3,4]},
              {"topic":"foo1","partition":1,"replicas":[2,3]},
              {"topic":"foo2","partition":1,"replicas":[2,3]}]
}

建议的分区重新分配配置

{"version":1,
"partitions":[{"topic":"foo1","partition":2,"replicas":[5,6]},
              {"topic":"foo1","partition":0,"replicas":[5,6]},
              {"topic":"foo2","partition":2,"replicas":[5,6]},
              {"topic":"foo2","partition":0,"replicas":[5,6]},
              {"topic":"foo1","partition":1,"replicas":[5,6]},
              {"topic":"foo2","partition":1,"replicas":[5,6]}]
}
```

该工具会生成一个候选分配，将所有分区从topic foo1，foo2移动到brokers 5,6。但是，请注意，这个时候，分区操作还没有开始，它只是告诉你当前的任务和建议的新任务。应该保存当前的分配，以防您想要回滚到它。新的任务应该保存在一个json文件（例如expand-cluster-reassignment.json）中，并用--execute选项输入到工具中，如下所示：

```
> bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --reassignment-json-file expand-cluster-reassignment.json --execute
当前分区副本分配

{"version":1,
"partitions":[{"topic":"foo1","partition":2,"replicas":[1,2]},
              {"topic":"foo1","partition":0,"replicas":[3,4]},
              {"topic":"foo2","partition":2,"replicas":[1,2]},
              {"topic":"foo2","partition":0,"replicas":[3,4]},
              {"topic":"foo1","partition":1,"replicas":[2,3]},
              {"topic":"foo2","partition":1,"replicas":[2,3]}]
}

保存这个以在回滚期间用作--reassignment-json-file选项
成功开始重新分配分区
{"version":1,
"partitions":[{"topic":"foo1","partition":2,"replicas":[5,6]},
              {"topic":"foo1","partition":0,"replicas":[5,6]},
              {"topic":"foo2","partition":2,"replicas":[5,6]},
              {"topic":"foo2","partition":0,"replicas":[5,6]},
              {"topic":"foo1","partition":1,"replicas":[5,6]},
              {"topic":"foo2","partition":1,"replicas":[5,6]}]
}
```

最后，可以使用--verify选项来检查分区重新分配的状态。请注意，相同的expand-cluster-reassignment.json（与--execute选项一起使用）应与--verify选项一起使用：

```
> bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --reassignment-json-file expand-cluster-reassignment.json --verify
Status of partition reassignment:
Reassignment of partition [foo1,0] completed successfully
Reassignment of partition [foo1,1] is in progress
Reassignment of partition [foo1,2] is in progress
Reassignment of partition [foo2,0] completed successfully
Reassignment of partition [foo2,1] completed successfully
Reassignment of partition [foo2,2] completed successfully
```

自定义分区分配和迁移

分区重新分配工具也可用于选择性地将分区的副本移动到特定的一组 broker。当以这种方式使用时，假定用户知道重新分配计划并且不需要该工具产生候选的重新分配，有效地跳过 --generate 步骤并直接到--execute步骤

例如，以下示例将 topic foo1的分区0 移到 broker 5,6中和将 topic foo2的分区1移到 broker 2,3中：

第一步是在json文件中定义重新分配计划：

```
> cat custom-reassignment.json
{"version":1,"partitions":[{"topic":"foo1","partition":0,"replicas":[5,6]},{"topic":"foo2","partition":1,"replicas":[2,3]}]}
```

然后，使用带有 --execute 选项的 json 文件来启动重新分配过程：

```
 > bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --reassignment-json-file custom-reassignment.json --execute
当前分区副本分配情况

 {"version":1,
 "partitions":[{"topic":"foo1","partition":0,"replicas":[1,2]},
               {"topic":"foo2","partition":1,"replicas":[3,4]}]
 }

 保存这个以在回滚期间用作 --reassignment-json-file 选项
 成功开始重新分配分区
 {"version":1,
 "partitions":[{"topic":"foo1","partition":0,"replicas":[5,6]},
               {"topic":"foo2","partition":1,"replicas":[2,3]}]
 }
```

可以使用--verify选项来检查分区重新分配的状态。 请注意，相同的expand-cluster-reassignment.json（与--execute选项一起使用）应与--verify选项一起使用：

```
> bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --reassignment-json-file custom-reassignment.json --verify
Status of partition reassignment:
Reassignment of partition [foo1,0] completed successfully
Reassignment of partition [foo2,1] completed successfully
```

下线 brokers

分区重新分配工具不具备为下线 broker 自动生成重新分配计划的功能。因此，管理员必须自己整理重新分配计划，将托管在即将下线的 broker 上的所有分区的副本移动到其他 broker。这可能比较单调，因为重新分配需要确保所有副本不会从将下线的 broker 只转移到唯一的 broker。 为了使这一过程毫不费力，我们计划在未来为下线 broker 添加工具支持。

增加复制因子

增加现有分区的复制因子很容易。只需在自定义重新分配json文件中指定额外的副本，并将其与--execute选项一起使用，以增加指定分区的复制因子。

例如，以下示例将foo的分区0的复制因子从1增加到3。在增加复制因子之前，该分区的唯一副本存在于 broker 5上。作为增加复制因子的一部分，我们将添加更多副本到 broker 6和7。

第一步是在json文件中自定义重新分配计划：

```
> cat increase-replication-factor.json
{"version":1,
"partitions":[{"topic":"foo","partition":0,"replicas":[5,6,7]}]}
```

然后，使用带有--execute选项的json文件来启动重新分配过程：

```
> bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --reassignment-json-file increase-replication-factor.json --execute
当前分区副本分配
```

```
3
4  {"version":1,
5  "partitions":[{"topic":"foo","partition":0,"replicas":[5]}]}
6
7  保存这个以在回滚期间用作--reassignment-json-file选项
8  成功开始重新分配分区
9  {"version":1,
10 "partitions":[{"topic":"foo","partition":0,"replicas":[5,6,7]}]}
```

可以使用--verify选项来检查分区重新分配的状态。请注意，与--verify选项使用的increase-replication-factor.json要与--execute选项一起使用的相同：

```
1  > bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --reassignment-json-file increase-replication-factor.json --verify
2  Status of partition reassignment:
3  Reassignment of partition [foo,0] completed successfully
```

您还可以使用kafka-topics工具验证复制因子的增加情况：

```
1  > bin/kafka-topics.sh --zookeeper localhost:2181 --topic foo --describe
2  Topic:foo   PartitionCount:1    ReplicationFactor:3 Configs:
3    Topic: foo    Partition: 0    Leader: 5   Replicas: 5,6,7 Isr: 5,6,7
```

限制数据迁移过程中的带宽使用

Kafka允许您设置复制流量的阈值，设置用于将副本从机器移动到另一台机器上的带宽上限。在重新平衡群集，引导新 broker 或添加或删除 broker 时，这非常有用，因为它限制了这些数据密集型操作对用户的影响。

有两个接口可以用来调节阈值。最简单也是最安全的是在调用kafka-reassign-partitions.sh时调节，但也可以使用kafka-configs.sh直接查看和更改流量阀值。

例如，如果要使用下面的命令执行重新平衡，它将以不超过 50MB/s 的速度移动分区。

```
1  $ bin/kafka-reassign-partitions.sh --zookeeper myhost:2181--execute --reassignment-json-file bigger-cluster.json —throttle 50000000
```

当你执行这个脚本时，你会看到：

```
1  The throttle limit was set to 50000000 B/s
2  Successfully started reassignment of partitions.
```

如果你想改变阈值，在重新平衡期间，比如增加吞吐量以便更快地完成，你可以通过重新运行execute命令来传递同样的reassignment-json-file：

```
1  $ bin/kafka-reassign-partitions.sh --zookeeper localhost:2181  --execute --reassignment-json-file bigger-cluster.json --throttle 700000000
2    There is an existing assignment running.
3    The throttle limit was set to 700000000 B/s
```

当重新平衡完成后，管理员可以使用--verify选项检查重新平衡的状态。如果重新平衡完成，流量阈值将通过--verify命令删除。 一旦重新平衡完成后，管理员必须及时通过 --verify 选项删除节流阀，如果不这样做可能会导致正常的复制流量受到限制。

当执行--verify选项并且重新分配完成时，脚本将确认节流阀已被移除：

```
1  > bin/kafka-reassign-partitions.sh --zookeeper localhost:2181  --verify --reassignment-json-file bigger-cluster.json
2  Status of partition reassignment:
3  Reassignment of partition [my-topic,1] completed successfully
4  Reassignment of partition [mytopic,0] completed successfully
5  Throttle was removed.
```

管理员还可以使用kafka-configs.sh验证分配的配置。有两对节流阀配置用于管理节流过程。节流阈值本身在 broker 级别使用动态属性进行配置的：

```
1  leader.replication.throttled.rate
2    follower.replication.throttled.rate
```

还有一组枚举类型的复制节流配置：

```
leader.replication.throttled.replicas
  follower.replication.throttled.replicas
```

每个 topic 配置了哪些。所有四个配置值都由kafka-reassign-partitions.sh自动分配(下面讨论).

查看流量限制配置：

```
> bin/kafka-configs.sh --describe --zookeeper localhost:2181 --entity-type brokers
Configs for brokers '2' are leader.replication.throttled.rate=700000000,follower.replication.throttled.rate=700000000
Configs for brokers '1' are leader.replication.throttled.rate=700000000,follower.replication.throttled.rate=700000000
```

这显示应用于复制协议的 leader 和 follower 的节流阀。默认情况下，双方都被分配相同的限制吞吐量值。

要查看节流副本的列表，请执行以下操作：

```
> bin/kafka-configs.sh --describe --zookeeper localhost:2181 --entity-type topics
Configs for topic 'my-topic' are leader.replication.throttled.replicas=1:102,0:101,
    follower.replication.throttled.replicas=1:101,0:102
```

这里我们看到 leader 节流阀被应用于 broker 102上的分区1和 broker 101上的分区0。follower 节流阀同样被应用于 broker 101上的分区1和 broker 102上的分区0。

默认情况下，kafka-reassign-partitions.sh将把 leader 的节流阀应用于重新平衡之前存在的所有副本，其中任何一个都可能是 leader。它会将 follower 应用到所有的目的地。 因此，如果 broker 101,102上有副本的分区，被重新分配到102,103，那么该分区的 leader 节流阀将被应用于101,102，并且 follower 节流阀将仅被应用于103。

如果需要，还可以使用kafka-configs.sh上的--alter开关手动更改节流阀配置。

安全使用节流复制

在使用节流复制时应该小心。 尤其是：

(1) 移除节流阀

一旦重新分配完成，应及时移除节流阀（通过运行kafka-reassign-partitions -verify）。

(2) 确保进展：

如果与传入写入速率相比阈值设置得太低，复制可能无法取得进展。这发生在：

```
max(BytesInPerSec) > throttle
```

BytesInPerSec是监控 producer 写入每个 broker 的写入吞吐量的指标。

管理员可以在重新平衡期间使用以下指标监控复制是否正在取得进展：

```
kafka.server:type=FetcherLagMetrics,name=ConsumerLag,clientId=([-.\w]+),topic=
```

复制期间滞后数据应该不断减少。如果指标不降低，管理员应按上述方法增大节流阈值。

设置配额

配额覆盖值和默认值可以在（user=user1, client-id=clientA），用户或客户端级别配置，如此处所述。 默认情况下，客户端是无限制的配额。可以为每个（user, client-id），用户或客户端组设置自定义配额。

为（user = user1, client-id = clientA）配置自定义配额：

```
> bin/kafka-configs.sh  --zookeeper localhost:2181 --alter --add-config 'producer_byte_rate=1024,consumer_byte_rate=2048,request_percentage=200' --entity-type users --
Updated config for entity: user-principal 'user1', client-id 'clientA'.
```

为user = user1配置自定义配额：

```
> bin/kafka-configs.sh  --zookeeper localhost:2181 --alter --add-config 'producer_byte_rate=1024,consumer_byte_rate=2048,request_percentage=200' --entity-type users --
Updated config for entity: user-principal 'user1'.
```

为client-id=clientA配置自定义配额：

```
> bin/kafka-configs.sh  --zookeeper localhost:2181 --alter --add-config 'producer_byte_rate=1024,consumer_byte_rate=2048,request_percentage=200' --entity-type clients
Updated config for entity: client-id 'clientA'.
```

可以通过指定 --entity-default 选项而不是--entity-name来为每个（user, client-id），用户或客户端ID组设置默认配额。

为user = userA配置默认客户端配额：

```
> bin/kafka-configs.sh  --zookeeper localhost:2181 --alter --add-config 'producer_byte_rate=1024,consumer_byte_rate=2048,request_percentage=200' --entity-type users --
Updated config for entity: user-principal 'user1', default client-id.
```

为用户配置默认配额：

```
> bin/kafka-configs.sh  --zookeeper localhost:2181 --alter --add-config 'producer_byte_rate=1024,consumer_byte_rate=2048,request_percentage=200' --entity-type users --
Updated config for entity: default user-principal.
```

配置client-id的默认配额：

```
> bin/kafka-configs.sh  --zookeeper localhost:2181 --alter --add-config 'producer_byte_rate=1024,consumer_byte_rate=2048,request_percentage=200' --entity-type clients
Updated config for entity: default client-id.
```

以下是如何描述给定（user, client-id）的配额：

```
> bin/kafka-configs.sh  --zookeeper localhost:2181 --describe --entity-type users --entity-name user1 --entity-type clients --entity-name clientA
Configs for user-principal 'user1', client-id 'clientA' are producer_byte_rate=1024,consumer_byte_rate=2048,request_percentage=200
```

描述给定用户的配额：

```
> bin/kafka-configs.sh  --zookeeper localhost:2181 --describe --entity-type users --entity-name user1
Configs for user-principal 'user1' are producer_byte_rate=1024,consumer_byte_rate=2048,request_percentage=200
```

描述给定 client-id的配额：

```
> bin/kafka-configs.sh  --zookeeper localhost:2181 --describe --entity-type clients --entity-name clientA
Configs for client-id 'clientA' are producer_byte_rate=1024,consumer_byte_rate=2048,request_percentage=200
```

如果未指定实体名称，则描述指定类型的所有实体。 例如，描述所有用户：

```
> bin/kafka-configs.sh  --zookeeper localhost:2181 --describe --entity-type users
Configs for user-principal 'user1' are producer_byte_rate=1024,consumer_byte_rate=2048,request_percentage=200
Configs for default user-principal are producer_byte_rate=1024,consumer_byte_rate=2048,request_percentage=200
```

对于(user, client)也是同样的：

```
> bin/kafka-configs.sh  --zookeeper localhost:2181 --describe --entity-type users --entity-type clients
Configs for user-principal 'user1', default client-id are producer_byte_rate=1024,consumer_byte_rate=2048,request_percentage=200
Configs for user-principal 'user1', client-id 'clientA' are producer_byte_rate=1024,consumer_byte_rate=2048,request_percentage=200
```

通过在 broker 上设置这些配置，可以适用于所有client-id的默认配额。只有在Zookeeper中未配置配额覆盖或默认配置时才应用这些属性。默认情况下，每个 client-id 都会收到一个无限制的配额。以下设置每个 producer 和 consumer 客户端的默认配额为10MB/sec。

```
1  quota.producer.default=10485760
2  quota.consumer.default=10485760
```

请注意，这些属性已被弃用，并可能在未来版本中删除。使用kafka-configs.sh配置的默认值优先于这些属性。

6.2 数据中心

有一些部署需要维护一个跨越多个数据中心的数据管道。对此，我们推荐的方法是在每个拥有众多应用实例的数据中心内部署一个本地Kafka集群，在每个数据中心内只与本地的kafka集群进行交互，然后各集群之间通过镜像进行同步，（请参阅镜像制作工具< / a>了解怎么做到这一点）。.

这种部署模式允许数据中心充当一个独立的实体，并允许我们能够集中的管理和调节数据中心之间的复制。在这种部署模式下，即使数据中心间的链路不可用，每个设施也可以独立运行：当发生这种情况时，镜像会落后，直到链路恢复正常并追上时为止。

如果应用程序需要所有数据的全局视图，你可以提供一个聚合数据的集群，使用镜像将所有数据中心的本地集群镜像聚合起来。聚合集群用于需要全部数据集的应用程序读取。

这并是不唯一的部署模式，可以通过广域网读取或者写入到远程的Kafka集群，但是这显然会增加获取集群的延时。

Kafka能在生产端和消费端很轻易的批处理数据，所以即使在高延时的连接中也可以实现高吞吐量。为此.虽然我们可能需要在生产端，消费端还有broker端增加TCP 套接字缓冲区大小，修改如下参数配置 `socket.send.buffer.bytes` 和 `socket.receive.buffer.bytes` 。具体请参见这里。

通常不建议在高延时链路的情况下部署一个跨越多个数据中心的Kafka集群。这将对Kafka写入和ZooKeeper写入产生非常高的复制延时，当各位置节点之间的网络不可用时，Kafka和ZooKeeper也将不保证可用

6.3 Kafka 配置

重要的客户端配置

最重要的老的 scala 版本的 producer 配置

- acks
- compression
- sync vs async production
- batch size (for async producers)

最重要的新的 Java 版本的 producer 配置

- *acks*
- *compression*
- *batch size*

最重要的 consumer 配置是 fetch size。

所有的配置请查阅 configuration 章节。

一个生产服务器配置

以下是生产服务器配置示例：

```
# ZooKeeper
zookeeper.connect=[list of ZooKeeper servers]

# Log configuration
num.partitions=8
default.replication.factor=3
log.dir=[List of directories. Kafka should have its own dedicated disk(s) or SSD(s).]

# Other configurations
broker.id=[An integer. Start with 0 and increment by 1 for each new broker.]
listeners=[list of listeners]
auto.create.topics.enable=false
min.insync.replicas=2
queued.max.requests=[number of concurrent requests]
```

我们的客户端配置在不同的使用场景下需要相应的变化。

6.4 Java 版本

从安全角度来看，我们建议您使用JDK 1.8的最新发布版本，因为较早的免费版本已经披露了安全漏洞。LinkedIn目前正在使用G1垃圾收集器运行JDK1.8 u5（希望升级到更新的版本）。如果您决定使用G1（当前默认值），并且您仍然使用JDK1.7，请确保您使用的是u51或者以上版本。LinkedIn已经在测试中试用了u21，但是在该版本中，GC方面存在着一些问题。 LinkedIn的调整如下所示：

```
-Xmx6g -Xms6g -XX:MetaspaceSize=96m -XX:+UseG1GC
-XX:MaxGCPauseMillis=20 -XX:InitiatingHeapOccupancyPercent=35 -XX:G1HeapRegionSize=16M
-XX:MinMetaspaceFreeRatio=50 -XX:MaxMetaspaceFreeRatio=80
```

作为参考，下面是LinkedIn最繁忙的群集（峰值）之一的统计数据：

- *60 brokers*
- *50k partitions (replication factor 2)*
- *800k messages/sec in*
- *300 MB/sec inbound, 1 GB/sec+ outbound*

该调整看起来相当激进，但是在集群中的所有broker的GC暂停时间90%都在大约21ms，并且每秒钟的yong GC少于一次。

6.5 硬件和操作系统

我们正在使用双四核24GB的Intel Xeon 机器。

*您需要足够的内存来缓存活动的readers和writers。您可以通过假设您希望缓存30秒，将您的内存需求计算为write_throughput * 30来进行内存需求的后期估计。*

磁盘的吞吐量很重要。我们有8x7200转的SATA硬盘。通常磁盘的吞吐量是瓶颈，磁盘是越多越好。您能不能从更昂贵的磁盘中受益取决于你的刷新配置（如果您经常强制刷新，那么更高转速的SAS硬盘可能更好）。

OS

Kafka应该在任何Unix系统运行良好，并且已经在Linux和Solaris上进行了测试。

我们已经发现了在windows运行的一些问题，目前windows还不是一个理想的支持平台，虽然我们很乐意改变这个问题。

Kafka不需要太多的操作系统层面的调优，但是有两个潜在重要的操作系统级别的配置：

- 文件描述符限制: Kafka把文件描述符用于日志段和打开连接。如果一个broker上有许多分区，则考虑broker至少(number_of_partitions)*(partition_size/segment_size) 个文件描述符来跟踪所有的日志段和broker所创建的连接。我们推荐每一个broker一开始至少配置100000个文件描述符。
- 最大套接字缓冲区大小：可以增加以实现数据中心之间的高性能数据传输，如此处所述。

磁盘和文件系统

我们建议使用多个驱动器以获得良好的吞吐量，并且为了确保良好的延迟，不应该与应用程序日志或其他的操作系统文件系统活动共享用于Kafka数据的相同驱动器。您可以将这些驱动器RAID成单个卷或格式，并且把每个驱动器挂载到它自己的目录。Kafka的副本冗余功能可以由RAID或者应用程序级别提供，可以折衷选择实现。

如果您配置了多个数据目录，那么分区将被循环分配到数据目录。每个分区只属于一个数据目录，如果分区间的数据不均衡，则可能导致磁盘间的负载不均衡。

RAID可以更好地平衡磁盘之间的负载（尽管似乎并不总是如此），因为它在较低的级别上进行平衡负载。RAID的主要缺点是通常会大幅度影响写入性能并且降低可用磁盘空间。

RAID的另一个潜在好处是能够容忍磁盘故障。 然而，我们的经验是，重建RAID阵列是I/O密集型操作以至于服务器不可用，所以这不提供太多的实际可用性改进。

应用程序 vs. OS 刷新管理

Kafka总是立即将所有数据写入文件系统，并支持配置刷新策略的功能，该策略控制何时将数据从OS缓存中强制刷新到磁盘上。该刷新策略可以控制在一段时间之后或者在写入一定数量的消息之后把数据持久化到磁盘。这里有几个可选配置项。

Kafka最总必须调用fsync指令才能知道数据已经被刷新。当从任何不被fsync所知的日志段崩溃中恢复时，Kafka将通过每个消息的CRC来检查其完整性，并且在启动时执行的恢复过程中会重建相应的偏移量索引文件。

请注意，Kafka中的持久性并不需要将数据同步到磁盘，因为失败的节点将始终从其副本中恢复。

我们建议使用完全禁用应用程序fsync的默认刷新设置。这意味着依靠操作系统和Kafka自己的后台完成的刷新操作。这种设置对大多数用途是最好的选择：无需调整，巨大的吞吐量和延时，以及完全的恢复保证。我们认为通过复制提供的保证比同步到本地磁盘更好，但是一些偏执的人仍然可能喜欢让OS，Kafka和应用程序级别的fsync策略都得到支持。

使用应用程序级别刷新设置的缺点是它的磁盘使用模式效率低下（它是操作系统在重新排序时没有什么回旋余地），并且在大多数Linux文件系统block中fsync写入文件时会引入延时，而后台刷新则会做更多粒度的页面级锁定。

通常你不需要对文件系统进行任何低级别的调整，但是，在接下来的几节中，我们也会介绍其中的一些内容，以防万一。

理解Linux操作系统刷新行为

在Linux中，写入文件系统的数据在pagecache中保存，直到必须写入磁盘（由于应用程序级别的fsync或操作系统自己的刷新策略）。数据的刷新是通过一组叫做pdflush的后台线程来完成的（或者在2.6.32内核中的“flusher threads”）。

pdflush有一个可配置的策略，可以控制在缓存中可以维护多少脏数据，以及多久时间之前必须将数据写回到磁盘。 该策略详情请参阅这里。 当Pdflush无法跟上数据写入的速率时，最终会导致写入过程阻塞发生写入延时来减慢数据的堆积。

您可以通过执行下面命令来查看当前OS内存使用状态

```
1  > cat /proc/meminfo
```

这些值的含义在上面的链接中有描述。

相对于进程内缓存，使用 pagecache 来存储将被写入到磁盘的数据有几个优势：

- *I/O 调度器将一批连续的小写入转换成为更大的物理写入，从而提高吞吐量。*
- *I/O调度器将尝试重新排序写入操作，以尽量减少磁盘磁头的移动，从而提高吞吐量I/O*
- *它会自动使用机器上的所有可用内存*

文件系统的选择

Kafka在磁盘上使用常规的文件，不依赖于特定的文件系统。 然而，使用最多的两个文件系统是EXT4和XFS。 从历史上看，EXT4使用更多，但最近对XFS文件系统的改进已经表明它对Kafka的负载具有更好的性能，而且不会影响稳定性。

通过尝试各种文件系统的创建和挂载选项，在具有重要消息负载的集群上执行对比测试。Kafka监测的主要指标是“Request Local Time”，它表示追加操作的时间。XFS的本地时间更短（160ms比250ms +最好的EXT4配置），以及更低的平均等待时间。 随着磁盘性能变化，XFS的性能也表现出较小的波动。

一般文件系统注意事项

对于用于数据目录的任何文件系统，在Linux系统上，建议在挂载时使用以下选项：

- noatime：此选项禁止在读取文件时更新文件的atime（上次访问时间）属性。 这可以消除大量的文件系统写入，特别是在引导consumer的情况下。 Kafka根本不依赖atime属性，因此禁用这个属性是安全的。

XFS 注意事项

XFS文件系统具有大量的自动调整功能，因此无需在文件系统创建时或挂载时对默认设置进行任何更改。唯一值得考虑的调整参数是：

- largeio: 这会影响统计调用报告的首选I/O大小。 虽然这可以允许在更大的磁盘写入时获得更高的性能，但是实际上它对性能的影响很小或者没有影响。
- nobarrier: 对于具有 battery-backed cache 的底层设备，此选项可以通过禁用定期写入刷新来提供更多的性能。 但是，如果底层设备运行良好，则会向文件系统报告不需要刷新，此选项不起作用。

EXT4 注意事项

EXT4是Kafka数据目录的文件系统的一个可选择的选项，但是为了获得最高的性能需要调整几个挂载选项。 另外，这些选项在故障情况下通常是不安全的，并且会导致更多的数据丢失和损坏。 对于单个 broker 失败，这不是一个问题，因为可以擦除磁盘，并从集群重建副本。 但在诸如停电等多故障情况下，这可能意味着不容易恢复损坏的底层文件系统（数据）。 以下选项可以调整：

- data=writeback:Ext4默认为data = ordered，这会导致某些写入操作上有很强的顺序性。Kafka不需要这样的顺序，因为它在所有未刷新的日志上进行非常偏执的数据恢复。此设置消除了排序约束，似乎显著减少了延迟。
- Disabling journaling: Journaling 是一个折衷：在服务器崩溃之后，它会使重新启动更快，但会引入大量额外的锁定，从而增加写入性能的差异。那些不关心重启时间，想要减少写入延迟尖峰的主要来源，可以完全关闭Journaling。
- commit=num_secs: 这调整了ext4向其元数据日志提交的频率。将其设置为较低的值可以减少崩溃期间未刷新数据的丢失。将其设置为更高的值则将提高吞吐量。
- nobh: 当使用 data=writeback 模式时，此设置控制额外的排序保证。 Kafka应该是安全的，因为我们不依赖写入顺序并提高吞吐量和延迟。
- delalloc: 延迟分配意味着文件系统避免分配任何 block 直到物理写入发生。这使得ext4可以在很大程度上分配连续区域而不是较小的页面，并有助于确保数据连续写入。这个功能非常适合吞吐量。它似乎涉及到文件系统中的一些锁定，这增加了一些延迟差异。

6.6 监控

Kafka在服务器和Scala客户端使用Yammer Metrics输出指标报告。 Java客户端通过使用Kafka Metrics，一个内置的指标注册表，最大限度地减少了传递到客户端应用程序的依赖关系。两者都通过JMX公开指标，并且可以配置为使用可插拔的统计记录器，连接到您的监控系统来生成统计报告。

查看可用指标的最简单方法是启动jconsole并将其指向正在运行的kafka客户端或服务器;这样你就能浏览JMX的所有指标。

我们对以下指标进行图形化和告警:

DESCRIPTION	MBEAN NAME	NORMAL VALUE
Message in rate	kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec	
Byte in rate	kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec	
Request rate	kafka.network:type=RequestMetrics,name=RequestsPerSec,request={Produce|FetchConsumer|FetchFollower}	
Byte out rate	kafka.server:type=BrokerTopicMetrics,name=BytesOutPerSec	
Log flush rate and time	kafka.log:type=LogFlushStats,name=LogFlushRateAndTimeMs	
# of under replicated partitions (|ISR| < |all replicas|)	kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions	0
# of under minIsr partitions (|ISR| < min.insync.replicas)	kafka.server:type=ReplicaManager,name=UnderMinIsrPartitionCount	0
# of offline log directories	kafka.log:type=LogManager,name=OfflineLogDirectoryCount	0
Is controller active on broker	kafka.controller:type=KafkaController,name=ActiveControllerCount	only one broker in the cluster should have 1
Leader election rate	kafka.controller:type=ControllerStats,name=LeaderElectionRateAndTimeMs	non-zero when there are broker failures
Unclean leader election rate	kafka.controller:type=ControllerStats,name=UncleanLeaderElectionsPerSec	0
Partition counts	kafka.server:type=ReplicaManager,name=PartitionCount	mostly even across brokers
Leader replica counts	kafka.server:type=ReplicaManager,name=LeaderCount	mostly even across brokers
ISR shrink rate	kafka.server:type=ReplicaManager,name=IsrShrinksPerSec	If a broker goes down, ISR for some of the partitions will shrink. When that broker is up again, ISR will be expanded once the replicas are fully caught up. Other than that, the expected value for both ISR shrink rate and expansion rate is 0.
ISR expansion rate	kafka.server:type=ReplicaManager,name=IsrExpandsPerSec	See above
Max lag in messages btw follower and leader replicas	kafka.server:type=ReplicaFetcherManager,name=MaxLag,clientId=Replica	lag should be proportional to the maximum batch size of a produce request.
Lag in messages per follower replica	kafka.server:type=FetcherLagMetrics,name=ConsumerLag,clientId=([-.\w]+),topic=([-.\w]+),partition=([0-9]+)	lag should be proportional to the maximum batch size of a produce request.
Requests waiting in the producer purgatory	kafka.server:type=DelayedOperationPurgatory,name=PurgatorySize,delayedOperation=Produce	non-zero if ack=-1 is used
Requests waiting in the fetch purgatory	kafka.server:type=DelayedOperationPurgatory,name=PurgatorySize,delayedOperation=Fetch	size depends on fetch.wait.max.ms in the consumer
Request total time	kafka.network:type=RequestMetrics,name=TotalTimeMs,request={Produce|FetchConsumer|FetchFollower}	broken into queue, local, remote and response send time
Time the request waits in the request queue	kafka.network:type=RequestMetrics,name=RequestQueueTimeMs,request={Produce|FetchConsumer|FetchFollower}	
Time the request is processed at the leader	kafka.network:type=RequestMetrics,name=LocalTimeMs,request={Produce|FetchConsumer|FetchFollower}	
Time the request waits for the	kafka.network:type=RequestMetrics,name=RemoteT	non-zero for produce requests when ack=-1

follower	imeMs,request={Produce|FetchConsumer|FetchFollower}	
Time the request waits in the response queue	kafka.network:type=RequestMetrics,name=ResponseQueueTimeMs,request={Produce|FetchConsumer|FetchFollower}	
Time to send the response	kafka.network:type=RequestMetrics,name=ResponseSendTimeMs,request={Produce|FetchConsumer|FetchFollower}	
Number of messages the consumer lags behind the producer by. Published by the consumer, not broker.	*Old consumer:* kafka.consumer:type=ConsumerFetcherManager,name=MaxLag,clientId=([-.\w]+) *New consumer:* kafka.consumer:type=consumer-fetch-manager-metrics,client-id={client-id} Attribute: records-lag-max	
The average fraction of time the network processors are idle	kafka.network:type=SocketServer,name=NetworkProcessorAvgIdlePercent	between 0 and 1, ideally > 0.3
The average fraction of time the request handler threads are idle	kafka.server:type=KafkaRequestHandlerPool,name=RequestHandlerAvgIdlePercent	between 0 and 1, ideally > 0.3
Bandwidth quota metrics per (user, client-id), user or client-id	kafka.server:type={Produce|Fetch},user=([-.\w]+),client-id=([-.\w]+)	Two attributes. throttle-time indicates the amount of time in ms the client was throttled. Ideally = 0. byte-rate indicates the data produce/consume rate of the client in bytes/sec. For (user, client-id) quotas, both user and client-id are specified. If per-client-id quota is applied to the client, user is not specified. If per-user quota is applied, client-id is not specified.
Request quota metrics per (user, client-id), user or client-id	kafka.server:type=Request,user=([-.\w]+),client-id=([-.\w]+)	Two attributes. throttle-time indicates the amount of time in ms the client was throttled. Ideally = 0. request-time indicates the percentage of time spent in broker network and I/O threads to process requests from client group. For (user, client-id) quotas, both user and client-id are specified. If per-client-id quota is applied to the client, user is not specified. If per-user quota is applied, client-id is not specified.
Requests exempt from throttling	kafka.server:type=Request	exempt-throttle-time indicates the percentage of time spent in broker network and I/O threads to process requests that are exempt from throttling.

producer/consumer/connect/streams的通用监控指标

以下指标可用于 producer/consumer/connector/streams 实例。有关具体指标，请参阅以下部分。

METRIC/ATTRIBUTE NAME	DESCRIPTION	MBEAN NAME
connection-close-rate	Connections closed per second in the window.	kafka.[producer|consumer|connect]:type=[producer|consumer|connect]-metrics,client-id=([-.\w]+)
connection-creation-rate	New connections established per second in the window.	kafka.[producer|consumer|connect]:type=[producer|consumer|connect]-metrics,client-id=([-.\w]+)
network-io-rate	The average number of network operations (reads or writes) on all connections per second.	kafka.[producer|consumer|connect]:type=[producer|consumer|connect]-metrics,client-id=([-.\w]+)
outgoing-byte-rate	The average number of outgoing bytes sent per second to all servers.	kafka.[producer|consumer|connect]:type=[producer|consumer|connect]-metrics,client-id=([-.\w]+)
request-rate	The average number of requests sent per second.	kafka.[producer|consumer|connect]:type=[producer|consumer|connect]-metrics,client-id=([-.\w]+)
request-size-avg	The average size of all requests in the window.	kafka.[producer|consumer|connect]:type=[producer|consumer|connect]-metrics,client-id=([-.\w]+)
request-size-max	The maximum size of any request sent in the window.	kafka.[producer|consumer|connect]:type=[producer|consumer|connect]-metrics,client-id=([-.\w]+)
incoming-byte-rate	Bytes/second read off all sockets.	kafka.[producer|consumer|connect]:type=[producer|consumer|connect]-metrics,client-id=([-.\w]+)
response-rate	Responses received sent per second.	kafka.[producer|consumer|connect]:type=[producer|consumer|connect]-metrics,client-id=([-.\w]+)
select-rate	Number of times the I/O layer checked for new I/O to perform per second.	kafka.[producer|consumer|connect]:type=[producer|consumer|connect]-metrics,client-id=([-.\w]+)
io-wait-time-ns-avg	The average length of time the I/O thread spent waiting for a socket ready for reads or writes in nanoseconds.	kafka.[producer|consumer|connect]:type=[producer|consumer|connect]-metrics,client-id=([-.\w]+)
io-wait-ratio	The fraction of time the I/O thread spent waiting.	kafka.[producer|consumer|connect]:type=[producer|consumer|connect]-metrics,client-id=([-.\w]+)
io-time-ns-avg	The average length of time for I/O per select call in nanoseconds.	kafka.[producer|consumer|connect]:type=[producer|consumer|connect]-metrics,client-id=([-.\w]+)
io-ratio	The fraction of time the I/O thread spent doing I/O.	kafka.[producer|consumer|connect]:type=[producer|consumer|connect]-metrics,client-id=([-.\w]+)
connection-count	The current number of active connections.	kafka.[producer|consumer|connect]:type=[producer|consumer|connect]-metrics,client-id=([-.\w]+)

producer/consumer/connect/streams对应的每个broker的通用指标

以下指标可用于 producer/consumer/connector/streams 实例。有关具体指标，请参阅以下部分。

METRIC/ATTRIBUTE NAME	DESCRIPTION	MBEAN NAME
outgoing-byte-rate	The average number of outgoing bytes sent per second for a node.	kafka.producer:type=[consumer|producer|connect]-node-metrics,client-id=([-.\w]+),node-id=([0-9]+)
request-rate	The average number of requests sent per second for a node.	kafka.producer:type=[consumer|producer|connect]-node-metrics,client-id=([-.\w]+),node-id=([0-9]+)
request-size-avg	The average size of all requests in the window for a node.	kafka.producer:type=[consumer|producer|connect]-node-metrics,client-id=([-.\w]+),node-id=([0-9]+)
request-size-max	The maximum size of any request sent in the window for a node.	kafka.producer:type=[consumer|producer|connect]-node-metrics,client-id=([-.\w]+),node-id=([0-9]+)
incoming-byte-rate	The average number of responses received per second for a node.	kafka.producer:type=[consumer|producer|connect]-node-metrics,client-id=([-.\w]+),node-id=([0-9]+)
request-latency-avg	The average request latency in ms for a node.	kafka.producer:type=[consumer|producer|connect]-node-metrics,client-id=([-.\w]+),node-id=([0-9]+)
request-latency-max	The maximum request latency in ms for a node.	kafka.producer:type=[consumer|producer|connect]-node-metrics,client-id=([-.\w]+),node-id=([0-9]+)
response-rate	Responses received sent per second for a node.	kafka.producer:type=[consumer|producer|connect]-node-metrics,client-id=([-.\w]+),node-id=([0-9]+)

Producer 监控

以下指标可用于 producer 实例。

METRIC/ATTRIBUTE NAME	DESCRIPTION	MBEAN NAME
waiting-threads	The number of user threads blocked waiting for buffer memory to enqueue their records.	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
buffer-total-bytes	The maximum amount of buffer memory the client can use (whether or not it is currently used).	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
buffer-available-bytes	The total amount of buffer memory that is not being used (either unallocated or in the free list).	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
bufferpool-wait-time	The fraction of time an appender waits for space allocation.	kafka.producer:type=producer-metrics,client-id=([-.\w]+)

Producer 发送人指标

新的 consumer 监控

以下指标可用于新的 consumer 实例。

Consumer Group Metrics

METRIC/ATTRIBUTE NAME	DESCRIPTION	MBEAN NAME
commit-latency-avg	The average time taken for a commit request	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
commit-latency-max	The max time taken for a commit request	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
commit-rate	The number of commit calls per second	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
assigned-partitions	The number of partitions currently assigned to this consumer	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
heartbeat-response-time-max	The max time taken to receive a response to a heartbeat request	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
heartbeat-rate	The average number of heartbeats per second	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
join-time-avg	The average time taken for a group rejoin	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
join-time-max	The max time taken for a group rejoin	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
join-rate	The number of group joins per second	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
sync-time-avg	The average time taken for a group sync	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
sync-time-max	The max time taken for a group sync	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)

sync-rate	The number of group syncs per second	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
last-heartbeat-seconds-ago	The number of seconds since the last controller heartbeat	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)

Consumer Fetch Metrics

Connect监控

一个Connect工作进程包含所有的producer和consumer指标以及特定于Connect的指标。这个工作进程本身有许多指标，而每个连接器和任务都还有额外的指标。

Streams监控

一个Kafka Streams实例包含所有的producer和consumer指标以及特定于streams的额外指标。 默认情况下，Kafka Streams有两个记录级别的指标：debug和info。debug级别记录所有的度量，而info级别只记录线程级别的度量。

注意，这些指标有一个3层的层次结构。在顶层，有每个线程的指标。每个线程都有任务，并有自己的指标。每个任务都有许多处理器节点，并有自己的指标。每个任务都有许多状态存储和记录缓存，所有这些都有自己的指标。

使用以下配置选项指定要收集哪些指标：

```
metrics.recording.level="info"
```

Thread指标

以下所有指标的记录级别都是``info``:

METRIC/ATTRIBUTE NAME	DESCRIPTION	MBEAN NAME
commit-latency-avg	The average execution time in ms for committing, across all running tasks of this thread.	kafka.streams:type=stream-metrics,client-id=([-.\w]+)
commit-latency-max	The maximum execution time in ms for committing across all running tasks of this thread.	kafka.streams:type=stream-metrics,client-id=([-.\w]+)
poll-latency-avg	The average execution time in ms for polling, across all running tasks of this thread.	kafka.streams:type=stream-metrics,client-id=([-.\w]+)
poll-latency-max	The maximum execution time in ms for polling across all running tasks of this thread.	kafka.streams:type=stream-metrics,client-id=([-.\w]+)
process-latency-avg	The average execution time in ms for processing, across all running tasks of this thread.	kafka.streams:type=stream-metrics,client-id=([-.\w]+)
process-latency-max	The maximum execution time in ms for processing across all running tasks of this thread.	kafka.streams:type=stream-metrics,client-id=([-.\w]+)
punctuate-latency-avg	The average execution time in ms for punctuating, across all running tasks of this thread.	kafka.streams:type=stream-metrics,client-id=([-.\w]+)
punctuate-latency-max	The maximum execution time in ms for punctuating across all running tasks of this thread.	kafka.streams:type=stream-metrics,client-id=([-.\w]+)
commit-rate	The average number of commits per second across all tasks.	kafka.streams:type=stream-metrics,client-id=([-.\w]+)
poll-rate	The average number of polls per second across all tasks.	kafka.streams:type=stream-metrics,client-id=([-.\w]+)
process-rate	The average number of process calls per second across all tasks.	kafka.streams:type=stream-metrics,client-id=([-.\w]+)
punctuate-rate	The average number of punctuates per second across all tasks.	kafka.streams:type=stream-metrics,client-id=([-.\w]+)
task-created-rate	The average number of newly created tasks per second.	kafka.streams:type=stream-metrics,client-id=([-.\w]+)
task-closed-rate	The average number of tasks closed per second.	kafka.streams:type=stream-metrics,client-id=([-.\w]+)

skipped-records-rate	The average number of skipped records per second.	kafka.streams:type=stream-metrics,client-id=([-.\w]+)

Task指标

以下所有指标的记录级别都是``debug``:

METRIC/ATTRIBUTE NAME	DESCRIPTION	MBEAN NAME
commit-latency-avg	The average commit time in ns for this task.	kafka.streams:type=stream-task-metrics,client-id=([-.\w]+),task-id=([-.\w]+)
commit-latency-max	The maximum commit time in ns for this task.	kafka.streams:type=stream-task-metrics,client-id=([-.\w]+),task-id=([-.\w]+)
commit-rate	The average number of commit calls per second.	kafka.streams:type=stream-task-metrics,client-id=([-.\w]+),task-id=([-.\w]+)

Processor Node指标

以下所有指标的记录级别都是``debug``:

METRIC/ATTRIBUTE NAME	DESCRIPTION	MBEAN NAME
process-latency-avg	The average process execution time in ns.	kafka.streams:type=stream-processor-node-metrics,client-id=([-.\w]+),task-id=([-.\w]+),processor-node-id=([-.\w]+)
process-latency-max	The maximum process execution time in ns.	kafka.streams:type=stream-processor-node-metrics,client-id=([-.\w]+),task-id=([-.\w]+),processor-node-id=([-.\w]+)
punctuate-latency-avg	The average punctuate execution time in ns.	kafka.streams:type=stream-processor-node-metrics,client-id=([-.\w]+),task-id=([-.\w]+),processor-node-id=([-.\w]+)
punctuate-latency-max	The maximum punctuate execution time in ns.	kafka.streams:type=stream-processor-node-metrics,client-id=([-.\w]+),task-id=([-.\w]+),processor-node-id=([-.\w]+)
create-latency-avg	The average create execution time in ns.	kafka.streams:type=stream-processor-node-metrics,client-id=([-.\w]+),task-id=([-.\w]+),processor-node-id=([-.\w]+)
create-latency-max	The maximum create execution time in ns.	kafka.streams:type=stream-processor-node-metrics,client-id=([-.\w]+),task-id=([-.\w]+),processor-node-id=([-.\w]+)
destroy-latency-avg	The average destroy execution time in ns.	kafka.streams:type=stream-processor-node-metrics,client-id=([-.\w]+),task-id=([-.\w]+),processor-node-id=([-.\w]+)
destroy-latency-max	The maximum destroy execution time in ns.	kafka.streams:type=stream-processor-node-metrics,client-id=([-.\w]+),task-id=([-.\w]+),processor-node-id=([-.\w]+)
process-rate	The average number of process operations per second.	kafka.streams:type=stream-processor-node-metrics,client-id=([-.\w]+),task-id=([-.\w]+),processor-node-id=([-.\w]+)
punctuate-rate	The average number of punctuate operations per second.	kafka.streams:type=stream-processor-node-metrics,client-id=([-.\w]+),task-id=([-.\w]+),processor-node-id=([-.\w]+)
create-rate	The average number of create operations per second.	kafka.streams:type=stream-processor-node-metrics,client-id=([-.\w]+),task-id=([-.\w]+),processor-node-id=([-.\w]+)
destroy-rate	The average number of destroy operations per second.	kafka.streams:type=stream-processor-node-metrics,client-id=([-.\w]+),task-id=([-.\w]+),processor-node-id=([-.\w]+)
forward-rate	The average rate of records being forwarded downstream, from source nodes only, per second.	kafka.streams:type=stream-processor-node-metrics,client-id=([-.\w]+),task-id=([-.\w]+),processor-node-id=([-.\w]+)

State Store指标

以下所有指标的记录级别都是``debug``:

METRIC/ATTRIBUTE NAME	DESCRIPTION	MBEAN NAME
put-latency-avg	The average put execution time in ns.	kafka.streams:type=stream-[store-type]-state-metrics,client-id=([-.\w]+),task-id=([-.\w]+),[store-type]-state-id=([-.\w]+)
put-latency-max	The maximum put execution time in ns.	kafka.streams:type=stream-[store-type]-state-metrics,client-id=([-.\w]+),task-id=([-.\w]+),[store-type]-state-id=([-.\w]+)
put-if-absent-latency-avg	The average put-if-absent execution time in ns.	kafka.streams:type=stream-[store-type]-state-metrics,client-id=([-.\w]+),task-id=([-.\w]+),[store-type]-state-id=([-.\w]+)
put-if-absent-latency-max	The maximum put-if-absent execution time in ns.	kafka.streams:type=stream-[store-type]-state-metrics,client-id=([-.\w]+),task-id=([-.\w]+),[store-type]-state-id=([-.\w]+)
get-latency-avg	The average get execution time in ns.	kafka.streams:type=stream-[store-type]-state-metrics,client-id=([-.\w]+),task-id=([-.\w]+),[store-type]-state-id=([-.\w]+)
get-latency-max	The maximum get execution time in ns.	kafka.streams:type=stream-[store-type]-state-metrics,client-id=([-.\w]+),task-id=([-.\w]+),[store-type]-state-id=([-.\w]+)
delete-latency-avg	The average delete execution time in ns.	kafka.streams:type=stream-[store-type]-state-metrics,client-id=([-.\w]+),task-id=([-.\w]+),[store-type]-state-id=([-.\w]+)
delete-latency-max	The maximum delete execution time in ns.	kafka.streams:type=stream-[store-type]-state-metrics,client-id=([-.\w]+),task-id=([-.\w]+),[store-type]-state-id=([-.\w]+)

put-all-latency-avg	The average put-all execution time in ns.	kafka.streams:type=stream-[store-type]-state-metrics,client-id=([-.\w]+),task-id=([-.\w]+),[store-type]-state-id=([-.\w]+)
put-all-latency-max	The maximum put-all execution time in ns.	kafka.streams:type=stream-[store-type]-state-metrics,client-id=([-.\w]+),task-id=([-.\w]+),[store-type]-state-id=([-.\w]+)
all-latency-avg	The average all operation execution time in ns.	kafka.streams:type=stream-[store-type]-state-metrics,client-id=([-.\w]+),task-id=([-.\w]+),[store-type]-state-id=([-.\w]+)
all-latency-max	The maximum all operation execution time in ns.	kafka.streams:type=stream-[store-type]-state-metrics,client-id=([-.\w]+),task-id=([-.\w]+),[store-type]-state-id=([-.\w]+)
range-latency-avg	The average range execution time in ns.	kafka.streams:type=stream-[store-type]-state-metrics,client-id=([-.\w]+),task-id=([-.\w]+),[store-type]-state-id=([-.\w]+)
range-latency-max	The maximum range execution time in ns.	kafka.streams:type=stream-[store-type]-state-metrics,client-id=([-.\w]+),task-id=([-.\w]+),[store-type]-state-id=([-.\w]+)
flush-latency-avg	The average flush execution time in ns.	kafka.streams:type=stream-[store-type]-state-metrics,client-id=([-.\w]+),task-id=([-.\w]+),[store-type]-state-id=([-.\w]+)
flush-latency-max	The maximum flush execution time in ns.	kafka.streams:type=stream-[store-type]-state-metrics,client-id=([-.\w]+),task-id=([-.\w]+),[store-type]-state-id=([-.\w]+)
restore-latency-avg	The average restore execution time in ns.	kafka.streams:type=stream-[store-type]-state-metrics,client-id=([-.\w]+),task-id=([-.\w]+),[store-type]-state-id=([-.\w]+)
restore-latency-max	The maximum restore execution time in ns.	kafka.streams:type=stream-[store-type]-state-metrics,client-id=([-.\w]+),task-id=([-.\w]+),[store-type]-state-id=([-.\w]+)
put-rate	The average put rate for this store.	kafka.streams:type=stream-[store-type]-state-metrics,client-id=([-.\w]+),task-id=([-.\w]+),[store-type]-state-id=([-.\w]+)
put-if-absent-rate	The average put-if-absent rate for this store.	kafka.streams:type=stream-[store-type]-state-metrics,client-id=([-.\w]+),task-id=([-.\w]+),[store-type]-state-id=([-.\w]+)
get-rate	The average get rate for this store.	kafka.streams:type=stream-[store-type]-state-metrics,client-id=([-.\w]+),task-id=([-.\w]+),[store-type]-state-id=([-.\w]+)
delete-rate	The average delete rate for this store.	kafka.streams:type=stream-[store-type]-state-metrics,client-id=([-.\w]+),task-id=([-.\w]+),[store-type]-state-id=([-.\w]+)
put-all-rate	The average put-all rate for this store.	kafka.streams:type=stream-[store-type]-state-metrics,client-id=([-.\w]+),task-id=([-.\w]+),[store-type]-state-id=([-.\w]+)
all-rate	The average all operation rate for this store.	kafka.streams:type=stream-[store-type]-state-metrics,client-id=([-.\w]+),task-id=([-.\w]+),[store-type]-state-id=([-.\w]+)
range-rate	The average range rate for this store.	kafka.streams:type=stream-[store-type]-state-metrics,client-id=([-.\w]+),task-id=([-.\w]+),[store-type]-state-id=([-.\w]+)
flush-rate	The average flush rate for this store.	kafka.streams:type=stream-[store-type]-state-metrics,client-id=([-.\w]+),task-id=([-.\w]+),[store-type]-state-id=([-.\w]+)
restore-rate	The average restore rate for this store.	kafka.streams:type=stream-[store-type]-state-metrics,client-id=([-.\w]+),task-id=([-.\w]+),[store-type]-state-id=([-.\w]+)

Record Cache指标

以下所有指标的记录级别都是``debug``:

METRIC/ATTRIBUTE NAME	DESCRIPTION	MBEAN NAME
hitRatio-avg	The average cache hit ratio defined as the ratio of cache read hits over the total cache read requests.	kafka.streams:type=stream-record-cache-metrics,client-id=([-.\w]+),task-id=([-.\w]+),record-cache-id=([-.\w]+)
hitRatio-min	The mininum cache hit ratio.	kafka.streams:type=stream-record-cache-metrics,client-id=([-.\w]+),task-id=([-.\w]+),record-cache-id=([-.\w]+)
hitRatio-max	The maximum cache hit ratio.	kafka.streams:type=stream-record-cache-metrics,client-id=([-.\w]+),task-id=([-.\w]+),record-cache-id=([-.\w]+)

其它

我们建议监视GC时间和其它状态，以及各种服务器状态，如CPU利用率、I/O服务时间等。 在客户端，我们建议监视消息/字节率(全局和每个topic)、请求速率/大小/时间以及消费者端上所有分区之间的最大延迟消息和最小获取请求速率。为了使消费者跟上生产的速率，最大延迟需要小于阈值，最小提取率需要大于0。

审计

我们所做的最后一个提示是关于数据交付的正确性。我们审计发送的每条消息都被所有消费者使用，并测量发生的延迟。对于重要的topics，如果在某个时间段内没有达到一定的完整性，我们会提醒。这个细节在KAFKA-260中讨论。

6.7 ZooKeeper

稳定版本

当前稳定分支是3.4，最新版本是3.4.9。

Operationalizing ZooKeeper

在操作上，我们有一下符合规范的ZooKeeper安装方式：

- *在物理/硬件/网络上的冗余：尽量不要把他们放在同一个机架上，合适的硬件配置（但不要过分），尽量保持电源，网络等。一个典型的ZooKeeper集群有5或7台服务器，分别允许宕机2台和3台服务器。如果你想部署一个小型集群，3台服务器也可以部署，但是要记住，在这种情况下你只能宕机1台服务器。*
- *I/O隔离：如果你有大量的写入操作流入，你几乎肯定会把事务日志放在一组特定的磁盘上。写入事物日志是同步的（但为了性能会分批写入），因此并发写入会明显影响性能。数据快照是异步落盘，因此通常可以与操作系统和消息日志文件共享磁盘性能。你可以配置dataLogDir参数单独为服务器配置磁盘组。*
- *应用隔离：除非你真的了解其他应用的运行模式，否则不要和ZooKeeper安装在一起，最好是单独部署运行ZooKeeper（尽管ZooKeeper可以均衡的利用硬件资源）。*
- *谨慎使用虚拟化：他的运行状况取决于你的集群架构，读写模式和SLA，即便是由虚拟化层引入的微小开销也可能造成ZooKeeper的中断，毕竟ZooKeeper对此十分敏感。*
- *ZooKeeper配置：他是java运行的，首先确保你给他分配足够的堆空间（我们通常配置3–5G，但这是根据我们现有数据实际情况来定的）。不幸的是，我们没有一个好的固定公式来确定他的值，但是要记住分配个ZooKeeper的堆空间越大，快照也就越大，从而会影响快照的恢复时间。实际上，如果快照变得太大（几个G），那你能需要增加initlimit参数的值，以便为服务器提供足够的时间来恢复并加入集群。*
- *监控:JMX和4个字母的命令（ZooKeeper提供的一系列命令，如：conf,cons,dump等）非常有用，他们在某些功能上重复了（这种情况下我们更喜欢4lw命令，他们似乎更容易预测情况，至少，他们和基础设施监控兼容性更好）*
- *不要过度构建集群：大型集群，尤其是大量写入的情况下，意味着大量的集群内部通信（集群成员节点的写入和后续的仲裁更新），但是过小的将集群将承担不必要的风险。添加更多的服务器可以增加集群的读取能力。*

总体来看，我们应尽量保持zookeeper尽可能小的处理负载（标准增长容量规划） 并尽可能的简单。与官方版本相比，我们尽量对配置和应用布局不做什么更改，尽可能保持官方原版。基于这些原因，我们倾向于跳过操作系统打包的版本。因为为了有更好的表现，它倾向于把关注点放在可能“混乱”的标准系统层上。

7. 安全

7.1 安全总览

在0.9.0.0版中，Kafka社区添加了一些特性，通过单独使用或者一起使用这些特性，提高了Kafka集群的安全性。目前支持下列安全措施：

1. 使用SSL或SASL验证来自客户端(producers和consumers)、其他brokers和工具的连接。Kafka支持以下SASL机制：
   - SASL/GSSAPI (Kerberos) - 从版本0.9.0.0开始
   - SASL/PLAIN - 从版本0.10.0.0开始

- SASL/SCRAM-SHA-256 和 SASL/SCRAM-SHA-512 - 从版本0.10.2.0开始

2. 验证从brokers 到 ZooKeeper的连接
3. 对brokers与clients之间、brokers之间或brokers与工具之间使用SSL传输对数据加密(注意，启用SSL时性能会下降，其大小取决于CPU类型和JVM实现)。
4. 授权客户端的读写操作
5. 授权是可插拔的，并且支持与外部授权服务的集成

值得注意的是，安全是可选的 - 支持非安全集群，也支持需要认证，不需要认证，加密和未加密clients的混合集群。 以下指南介绍了如何在clients和brokers中配置和使用安全功能。

7.2 使用SSL加密和授权

Apache Kafka允许客户端通过SSL进行连接。默认情况下，SSL被禁用，但可以根据需要启用。

1. 为每个Kafka broker生成SSL密钥和证书

部署一个或多个支持SSL的brokers的第一步是为集群中的每台计算机生成密钥和证书。你可以使用Java的keytool实用程序来完成此任务。最初我们将生成一个临时密钥库的密钥，以便我们稍后可以导出并用CA签名。

```
keytool -keystore server.keystore.jks -alias localhost -validity {validity} -genkey -keyalg RSA
```

你需要在上面的命令中指定两个参数：

1. keystore: 存储证书的密钥库文件。密钥库文件包含证书的私钥；因此，它需要安全保存。
2. validity: 证书有效期。

注意：默认情况下，属性ssl.endpoint.identification.algorithm未定义，所以不执行主机名验证。为了启用主机名验证，请设置以下属性：

```
ssl.endpoint.identification.algorithm=HTTPS
```

启用后，客户端将根据以下两个字段之一验证服务器的完全限定域名(FQDN)：

1. Common Name (CN)
2. Subject Alternative Name (SAN)

这两个字段都是有效的，但RFC-2818建议使用SAN。 SAN也更加灵活，允许声明多个DNS条目。另一个优点是可以将CN设置为更有意义的值用于授权目的。要添加SAN字段，请将以下参数 `-ext SAN=DNS:{FQDN}` 附加到keytool命令中：

```
keytool -keystore server.keystore.jks -alias localhost -validity {validity} -genkey -keyalg RSA -ext SAN=DNS:{FQDN}
```

之后可以运行以下命令来验证生成的证书内容：

```
keytool -list -v -keystore server.keystore.jks
```

2. 创建你自己的证书管理机构 (CA)

在完成第一步之后, 集群中的每一台机器都有一个公私密钥对, 和一个用于标识该机器的证书. 但是, 这个证书是未签名的, 也就是说攻击者也可以创造一个这样的证书, 假装是其中任何一台机器.

因此, 给集群中每个机器的证书签名, 来防止伪造的证书, 是非常重要的. 一个证书管理机构 (CA) 负责证书的签名. CA 的工作流程类似政府发放护照. 政府给每一个护照盖章(签名), 因此护照变得难以伪造. 其他政府通过验证这个盖章来确认护照是可信的. 同理, CA 给证书签名, 并且加密方法保证已经签名的证书是难以计算和伪造的. 所以, 只要 CA 是真实的可信的权威机构, 客户端能更好的保证他们连接到的是可信的机器.

```
openssl req -new -x509 -keyout ca-key -out ca-cert -days 365
```

生成的 CA 是一个简单的公私密钥对和证书, 然后, 它将对其他证书签名.
下一步是将生成的 CA 添加到 **客户端的信任存储区 (clients' truststore)**, 这样客户端才可以相信这个 CA:

```
keytool -keystore client.truststore.jks -alias CARoot -import -file ca-cert
```

Note: 如果你将 Kafka broker 配置成需要客户端授权, 可以在 Kafka broker 配置中设置 setting ssl.client.auth 属性为 "requested" 或者 "required". 然后你也一定要给 Kafka broker 提供一个信任存储区, 并且这个信任存储区应该拥有所有给客户端密钥签名的 CA 证书.

```
keytool -keystore server.truststore.jks -alias CARoot -import -file ca-cert
```

相对于第一步中的密钥库 (keystore) 存储每个机器自己的标识, 客户端的信任存储区 (truststore) 保存所有客户端需要信任的证书. 导入一个证书到一台机器的信任存储区, 意味着这台机器将信任所有被这个证书签名的证书. 与上面相似的, 相信政府 (CA) 即相信政府发放的所有护照(证书). 这一特征称为信任链, 当在大型的 Kafka 集群中部署 SSL 时, 这一特征特别有用. 你可以用一个单一的 CA 给集群中的所有证书签名, 而且可以让所有机器共享信任该 CA 的信任存储区. 这样, 每个机器都可以认证除自己之外的全部机器.

3. 给证书签名

下一步是用第二步生成的 CA 给第一步生成的所有证书签名. 首先, 你需要从密钥库中把证书导出来:

```
keytool -keystore server.keystore.jks -alias localhost -certreq -file cert-file
```

然后用 CA 给导出的证书签名:

```
openssl x509 -req -CA ca-cert -CAkey ca-key -in cert-file -out cert-signed -days {validity} -CAcreateserial -passin pass:{ca-password}
```

最后, 你要把 CA 的证书和被签名的证书一起导入密钥库中:

```
keytool -keystore server.keystore.jks -alias CARoot -import -file ca-cert
keytool -keystore server.keystore.jks -alias localhost -import -file cert-signed
```

参数的定义如下所示:

1. keystore: 密钥库的地址
2. ca-cert: CA 的证书
3. ca-key: CA 的私钥
4. ca-password: CA 的密码
5. cert-file: 从服务器导出的未被签名的证书
6. cert-signed: 已经签名的服务器的证书

这是一个集合上面所有步骤, 用 bash 写的例子. 注意其中一条命令假设密码是 `test1234`, 所以你要么使用这个密码, 要么在执行命令前修改该密码:

```
#!/bin/bash
#Step 1
keytool -keystore server.keystore.jks -alias localhost -validity
#Step 2
openssl req -new -x509 -keyout ca-key -out ca-cert -days 365
keytool -keystore server.truststore.jks -alias CARoot -import -f
keytool -keystore client.truststore.jks -alias CARoot -import -f
#Step 3
keytool -keystore server.keystore.jks -alias localhost -certreq
openssl x509 -req -CA ca-cert -CAkey ca-key -in cert-file -out c
keytool -keystore server.keystore.jks -alias CARoot -import -fil
keytool -keystore server.keystore.jks -alias localhost -import -
```

4. **配置 Kafka Broker**

Kafka Broker 支持在多个端口上监听连接. 我们需要在 server.properties 中配置以下属性, 必须有一个或多个用逗号隔开的值:

```
listeners
```

如果 broker 之间的通信没有启用 SSL (如何启用可参照下文). PLAINTEXT 和 SSL 两个协议都需要提供端口信息.

```
listeners=PLAINTEXT://host.name:port,SSL://host.name:port
```

下面是 broker 端需要的 SSL 的配置:

```
ssl.keystore.location=/var/private/ssl/server.keystore.jks
ssl.keystore.password=test1234
ssl.key.password=test1234
ssl.truststore.location=/var/private/ssl/server.truststore.jks
ssl.truststore.password=test1234
```

Note: 严格来讲 ssl.truststore.password 是可选配置的, 但是强烈建议配置. 如果没有配置 password, 依然可以访问信任存储区 (truststore), 但是不能进行真实性检查. 值得考虑的可选配置:

1. ssl.client.auth=none ("required" => 客户端授权是必须的, "requested" => 客户端授权是需要的, 但是没有证书依然可以连接. 因为 "requested" 会造成错误的安全感, 而且在客户端配置错误的情况下依然可以连接成功, 所以不鼓励使用.)
2. ssl.cipher.suites (可选的). 指定的密码套件, 由授权, 加密, MAC 和密钥交换算法组成, 用于给使用 TLS 或者 SSL 协议的网络协商安全设置. (默认是空的 list)
3. ssl.enabled.protocols=TLSv1.2,TLSv1.1,TLSv1 (列出你将要从客户端接收的 SSL 协议. 注意, SSL 已经废弃, 支持 TLS, 并且不建议在生产环境中使用 SSL.)
4. ssl.keystore.type=JKS
5. ssl.truststore.type=JKS
6. ssl.secure.random.implementation=SHA1PRNG

如果你想 broker 间的通信启用 SSL, 将下面内容添加到 server.properties 文件中(默认是 PLAINTEXT):

```
security.inter.broker.protocol=SSL
```

由于一些国家的进口规定, Oracle 的实现限制了默认加密算法的强度. 如果需要更强的算法 (例如, 256 位的 AES 密钥), 必须获得 JCE Unlimited Strength Jurisdiction Policy 文件, 并且在 JDK/JRE 中安装. 参考 JCA Providers 文档 获取更多新.

JRE/JDK 将有一个默认的伪随机数生成器 (PRNG), 用于加密操作, 所以没有要求使用

```
ssl.secure.random.implementation
```

配置 PRNG 的实现. 但是, 其中某些实现会造成性能问题 (尤其是, Linux 系统的默认选择,

```
NativePRNG
```

, 利用了一个全局锁.) 万一 SSL 连接的性能变成一个问题, 可以考虑, 明确的设置要使用的 PRNG 实现.

```
SHA1PRNG
```

实现是非阻塞的, 在高负载 (加上复制消息的流量, 每个 broker 生产消息的速度是 50 MB/sec) 下有非常好的性能表现.

一旦你启动了 broker, 你应该能在 server.log 中看到如下内容:

```
with addresses: PLAINTEXT -> EndPoint(192.168.64.1,9092,PLAINTE
```

用以下命令可以快速验证服务器的 keystore 和 truststore 是否设置正确:

```
openssl s_client -debug -connect localhost:9093 -tls1
```

(Note: TLSv1 需要在 ssl.enabled.protocols 中列出)
在命令的输出中, 你应该能看到服务器的证书:

```
-----BEGIN CERTIFICATE-----
{variable sized random bytes}
-----END CERTIFICATE-----
subject=/C=US/ST=CA/L=Santa Clara/O=org/OU=org/CN=Sriharsha Chir
issuer=/C=US/ST=CA/L=Santa Clara/O=org/OU=org/CN=kafka/emailAddr
```

如果没有显示证书信息, 或者有任何其他错误信息, 那么你的 keystore 设置不正确。

5. **配置 Kafka 客户端**

只有新的 Kafka Producer 和 Consumer 支持 SSL, 旧的 API 不支持 SSL. Producer 和 Consumer 的 SSL 的配置相同.

如果在 broker 中不需要客户端授权, 那么下面是最小的配置的例子:

```
security.protocol=SSL
ssl.truststore.location=/var/private/ssl/client.truststore.jks
ssl.truststore.password=test1234
```

Note: 严格来讲 ssl.truststore.password 是可选配置的, 但是强烈建议配置. 如果没有配置 password, 依然可以访问 truststore, 但是不能进行真实性检查. 如果要求客户端授权, 必须像第一步一样创建一个 keystore, 和配置下面这些信息:

```
ssl.keystore.location=/var/private/ssl/client.keystore.jks
ssl.keystore.password=test1234
ssl.key.password=test1234
```

根据我们的需求和 broker 的配置, 也需要设置其他的配置:

1. ssl.provider (可选的). 用于 SSL 连接的安全提供程序名称. 默认值是 JVM 的默认安全提供程序.
2. ssl.cipher.suites (可选的). 指定的密码套件, 由授权, 加密, MAC 和密钥交换算法组成, 用于给使用 TLS 或者 SSL 协议的网络协商安全设置.
3. ssl.enabled.protocols=TLSv1.2,TLSv1.1,TLSv1. 需要列出至少一个在 broker 端配置的协议
4. ssl.truststore.type=JKS
5. ssl.keystore.type=JKS

使用 console-producer 和 console-consumer 的例子:

```
kafka-console-producer.sh --broker-list localhost:9093 --topic test --producer.config client-ssl.properties
kafka-console-consumer.sh --bootstrap-server localhost:9093 --topic test --consumer.config client-ssl.properties
```

7.3 使用SASL实现身份验证

1. JAAS 的配置

Kafka 使用Java验证和授权API (JAAS) 来完成 SASL 配置.

1. Kafka brokers 的 JAAS 的配置

`KafkaServer` 是每一个 KafkaServer/Broker 的 JASS 文件里面的节点名称。 在这个节点中，提供了用于所有 brokers 之间通信的 SASL 客户端连接的 SASL 配置选项。

`Client` 节点是用于认证 SASL 与 zookeeper 之间的连接。它也允许 brokers 通过设定 zookeeper 节点中 SASL ACL 来锁定这些节点 从而确定这些节点不被其他 broker 修改。 在所有的 broker 中都必须使用相同的名称。 如果您想使用 Client 以外名称, 请设置系统属性 `zookeeper.sasl.clientconfig` 中填写合适的名称 (*e.g.*, `-Dzookeeper.sasl.clientconfig=ZkClient`).

ZooKeeper 使用 "zookeeper" 作为默认的名称. 如果您想改变它的话, 请设置系统属性 `zookeeper.sasl.client.username` 中填写合适的名称 (*e.g.*, `-Dzookeeper.sasl.client.username=zk`).

2. Kafka clients 的 JAAS 的配置

Client 通过配置客户端属性 sasl.jaas.config 或者通过与 brokers 相似的方法来配置 JAAS （static JAAS config file）

1. 通过客户端配置属性来配置JAAS

   客户无需创建物理文件来配置角色 只需要在 JASS 配置中指定 producer 或者 consumer。 这种模式通过为每个客户端指定不同的属性来使用不同的凭证，从而确保可以在在同一个 JVM 中使用不同的 producers 和 consumers。如果静态JAAS配置系统属性的方法 `java.security.auth.login.config` 和配置客户端属性 `sasl.jaas.config` 的方法 同时被使用, 客户端的配置将会被使用.

   请看 GSSAPI (Kerberos), PLAIN or SCRAM 中的示例配置。

2. 通过静态配置文件来配置 JAAS

   使用静态 JAAS 配置文件 来配置 SASL 的客户端认证服务：

   1. 添加一个名为 `KafkaClient` 客户端的登陆节点，同时为 `KafkaClient` 中所选机制来配置一个登陆模块 如设置GSSAPI (Kerberos), PLAIN 或者 SCRAM。 例如, GSSAPI 的凭证配置如下:

      ```
      KafkaClient {
      com.sun.security.auth.module.Krb5LoginModule required
      useKeyTab=true
      storeKey=true
      keyTab="/etc/security/keytabs/kafka_client.keytab"
      principal="kafka-client-1@EXAMPLE.COM";
      };
      ```

   2. 将 JASS 的配置文件位置作为 JVM 的参数传递给每个客户端的 JVM。 例如:

      ```
      -Djava.security.auth.login.config=/etc/kafka/kafka_client_jaas.conf
      ```

2. SASL 的配置

SASL 可以使用 PLAINTEXT 或者 SSL 协议作为传输层，相对应的就是使用 SASL_PLAINTEXT 或者 SASL_SSL 安全协议。 如果使用 SASL_SSL 安全协议，必须配置 SSL证书。

1. SASL 安全机制

Kafka 支持以下的四个 SASL 机制:

- GSSAPI (Kerberos)
- PLAIN
- SCRAM-SHA-256
- SCRAM-SHA-512

2. Kafka brokers 的SASL配置

1. 在 server.properties 文件中配置一个 SASL 端口， 要为 *listeners* 添加其中至少一个参数（SASL_PLAINTEXT 或者 SASL_SSL）：

```
listeners=SASL_PLAINTEXT://host.name:port
```

如果您只配置一个 SASL 端口 （或者您只想希望在 Kafka brokersz 之间使用 SASL 协议相互认证的话）您需要确保您在 borker 之间通信中 使用了了相同的 SASL 协议。

```
security.inter.broker.protocol=SASL_PLAINTEXT (or SASL_SSL)
```

2. 选择一个或者多个 支持的安全机制 然后按照所给的步骤来为所选的安全机制配置 SASL 协议。 如果您想在 broker 中启用多个安全机制， 请根据此处步骤操作。

3. **Kafka 客户端配置 SASL**

SASL 授权只在新的 Java Kafka producer and consumer API 中被支持。之前的API并不支持

要在客户端配置 SASL 授权， 需在 broker 中选择一个已启用的SASL 机制用于客户端授权 并且根据下面的步骤为所选的安全机制配置 SASL 协议。

3. 使用 SASL/Kerberos 认证协议

1. 准备条件

1. **Kerberos**
   如果您的项目已经使用 Kerberos 认证协议服务器（例如使用了Active Directory），那么您就无需为 Kafka 安装新的服务器，否则您需要从您 Linux 社区中的服务包中安装一个。 此处有一个简短的指南教您如何安装并且配置它(Ubuntu,Redhat). 请注意，若果您正在使用 Oracle Java， 您需要对应您的 Java 版本 下载 JCE 策略文件，并且将它拷贝到 $JAVA_HOME/jre/lib/security 路径下面。
2. **创建 Kerberos 证书**
   如果您在使用团队的 Kerberos 服务器或者 Active Directory 服务器, 请向您的 Kerberos 管理员咨询集群中每一个 Kafka broker 的证书 和 每个将通过 Kerberos 身份验证的用户的证书（通过客户端或者工具等方式通过身份验证）。
   如果您已经安装了自己的 Kerberos， 您可以根据下面的步骤为您自己创建证书：

```
sudo /usr/sbin/kadmin.local -q 'addprinc -randkey kafka/{hostname}@{REALM}'
sudo /usr/sbin/kadmin.local -q "ktadd -k /etc/security/keytabs/{keytabname}.keytab kafka/{hostname}@{REALM}"
```

3. **确保所有的主机名可以访问其对应的主机** - Kerberos 要求您的所有主机都可以被其 FQDN 进行解析。

2. Kafka Brokers 的配置

1. 为每一个 Kafka broker 的 config 目录下，创建一个类似下面的适当修改的 JAAS 文件，我们在这个例子中把它命名为 kafka_server_jaas.conf （请注意每一个 broker 都应该有他自己 keytab):

```
KafkaServer {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    storeKey=true
```

```
    keyTab="/etc/security/keytabs/kafka_server.keytab"
    principal="kafka/kafka1.hostname.com@EXAMPLE.COM";
};

// Zookeeper client authentication
Client {
com.sun.security.auth.module.Krb5LoginModule required
useKeyTab=true
storeKey=true
keyTab="/etc/security/keytabs/kafka_server.keytab"
principal="kafka/kafka1.hostname.com@EXAMPLE.COM";
};
```

JAAS 文件中 的 `KafkaServer` 节点 告诉 broker 使用哪一个认证以及对应的认证的 keytab 的位置。它允许 broker 根据节点中的 keytab 中的信息来认证登陆。 有关 Zookeeper SASL 配置的更多详细信息，请参阅 此处。

2. 将 JAAS 和 krb5（可选）的文件位置作为 JVM 的参数传给每一个 Kafka broker。获取更多资料请参照 此处。

```
    -Djava.security.krb5.conf=/etc/kafka/krb5.conf
        -Djava.security.auth.login.config=/etc/kafka/kafka_server_jaas.
```

3. 确保启动 Kafka broker 的操作系统用户可以读取 JAAS 文件中的 keytabs。
4. 在配置server.properties文件中，确保SASL 端口 和 SASL 安全机制配置正确. 例如:

```
    listeners=SASL_PLAINTEXT://host.name:port
        security.inter.broker.protocol=SASL_PLAINTEXT
        sasl.mechanism.inter.broker.protocol=GSSAPI
        sasl.enabled.mechanisms=GSSAPI
```

我们在 server.properties 文件中， 必须配置和 Kafka broker 证书中名称相匹配的服务器名称。 在下面的例子中， 证书的名称为"kafka/kafka1.hostname.com@EXAMPLE.com"，所以

```
    sasl.kerberos.service.name=kafka
```

3. **Kafka 客户端的配置**

在客户端配置 SASL 认证:

1. 客户端 (producers, consumers, connect workers, 等等) 将使用自己的证书对集群进行身份验证（证书的名字通常与运行客户端的用户同名） 然后根据需要来创建或者获取证书。从而为每一个客户端配置 JAAS 配置属性。 通过指定不同的证书， JVM的不同客户端可以作为不同的用户运行。 在 producer.properties 或者 consumer.properties 文件中 `sasl.jaas.config` 属性描述着客户端（producer 或 consumer）通过何种方式连接 Kafka broker。 下面的例子是使用 keytab 配置客户端（建议用于长时间运行的进程）：

```
    sasl.jaas.config=com.sun.security.auth.module.Krb5LoginModule requi
        useKeyTab=true \
        storeKey=true  \
```

```
        keyTab="/etc/security/keytabs/kafka_client.keytab" \
        principal="kafka-client-1@EXAMPLE.COM";
```

对于像 kafka-console-consumer 和 kafka-console-producer 这样的命令行应用程序， kinit 可以与 "useTicketCache=true" 一同使用。就像：

```
    sasl.jaas.config=com.sun.security.auth.module.Krb5LoginModule requi
        useTicketCache=true;
```

客户端的 JAAS 配置 也可以设定为与此处所述的 broker 类似的参数。 客户端使用登陆的节点名称 (`KafkaClient`)的这个选项只来确保 一个 JVM 的所有的客户端只对应一个用户。

2. 确保启动 Kafka broker 的操作系统用户可以读取 JAAS 文件中的 keytabs。
3. （可选）将 krb5 的文件位置作为 JVM 的参数传给每一个 Kafka broker。获取更多资料请参照此处：

```
    -Djava.security.krb5.conf=/etc/kafka/krb5.conf
```

4. 在 producer.properties 和 consumer.properties 文件中配置以下的属性：

```
    security.protocol=SASL_PLAINTEXT (or SASL_SSL)
    sasl.mechanism=GSSAPI
    sasl.kerberos.service.name=kafka
```

4. 使用 SASL/PLAIN 认证

SASL/PLAIN 是一种简单的 username/password 认证机制, 通常与 TLS 加密一起使用, 用于实现安全认证. Kafka 提供了一个默认的 SASL/PLAIN 实现, 可以做扩展后在生产环境使用, 如 这里 的描述.

username 在 ACL 等的配置中作为已认证的 `Principal`.

1. 配置 Kafka Broker

1. 在每一个 Kafka broker 的 config 目录中, 添加一个类似于下面的适当修改过的 JAAS 文件. 在这个例子中, 让我们将它命名为 kafka_server_jaas.conf:

```
KafkaServer {
    org.apache.kafka.common.security.plain.PlainLoginModule required
    username="admin"
    password="admin-secret"
    user_admin="admin-secret"
    user_alice="alice-secret";
};
```

这个配置定义了两个用户 (*admin* and *alice*). broker 使用在 `KafkaServer` 部分的 `username` 和 `password` 属性初始化与其他 broker 的连接. 在这个例子中, *admin* 是 broker 间通信的用户. `user_userName` 属性的值定义了所有连接到 broker 的用户的密码. 这个 broker 验证所有客户端的连接, 包括那些使用了这些配置的 broker 的连接.

2. 将 JAAS 配置文件的路径作为 JVM 的参数, 并传递到每一个 Kafka broker:

```
-Djava.security.auth.login.config=/etc/kafka/kafka_server_jaas.conf
```

3. 如 这里 描述的, 在 server.properties 中配置 SASL 端口和 SASL 机制. 例如:

```
listeners=SASL_SSL://host.name:port
    security.inter.broker.protocol=SASL_SSL
    sasl.mechanism.inter.broker.protocol=PLAIN
    sasl.enabled.mechanisms=PLAIN
```

2. **配置 Kafka Client**

在客户端上配置 SASL 验证:

1. 在每一个客户端的 producer.properties 或者 consumer.properties 中配置 JAAS 属性. 登录模块描述了像生产者和消费者这样的客户端如何连接到 Kafka Broker. 下面是一个 PLAIN 机制的客户端的配置示例:

```
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required \
    username="alice" \
    password="alice-secret";
```

`username` 和 `password` 选项被客户端用于配置客户端连接的用户. 在这个例子中, 客户端以 *alice* 用户连接到 broker. 通过在 `sasl.jaas.config` 中指定不同的用户名和密码, 在一个 JVM 内不同的客户端可以以不同的用户连接.

客户端的 JAAS 配置也可以像 这里 描述的 broker 一样, 指定为一个 JVM 参数. 客户端使用名为 `KafkaClient` 的登录部分. 此选项仅允许来自 JVM 的所有客户端连接中的一个用户.

2. 在 producer.properties 或者 consumer.properties 中配置下面这些属性:

```
security.protocol=SASL_SSL
sasl.mechanism=PLAIN
```

3. **在生产环境中使用 SASL/PLAIN**

- SASL/PLAIN 应该只用 SSL 作为传输层, 以保证在没有加密的情况下不会在线上传输明文密码.
- Kafka 中 SASL/PLAIN 的默认实现是在 JAAS 配置文件声明用户名和密码, 如 这里 展示的. 为了避免在磁盘上保存密码, 你可以实现你自己的 `javax.security.auth.spi.LoginModule`, 从一个外部源提供用户名和密码. 登录模块的实现应该提供 `Subject` 的用户名作为公共证书和密码作为私有的凭证. `org.apache.kafka.common.security.plain.PlainLoginModule` 的默认实现可以作为参考例子.
- 在生产系统, 外部的认证服务可以实现密码认证. 通过添加你自己的 `javax.security.sasl.SaslServer` 实现, Kafka brokers 可以和这些服务集成. Kafka 中的默认实现在 `org.apache.kafka.common.security.plain` 包中, 可以作为开始学习的例子.

- 新的提供者一定要在 JVM 中安装和注册. 可以通过添加提供类到平常的 CLASSPATH, 或者把提供类打包成 jar 文件并添加到*JAVA_HOME*/lib/ext中来安装提供者.
- 将提供者添加到安全属性文件 (security properties file) *JAVA_HOME*/lib/security/java.security 静态注册提供者.

```
    security.provider.n=providerClassName
```

其中 *providerClassName* 是新提供者的全称. *n* 是优先顺序, 比较小的数字表明更高的优先级.

- 此外, 你可以在运行时通过在客户端程序开始时或者在登录模块的一个静态初始化程序中唤起 `Security.addProvider` 来动态的注册提供者. 例如:

```
    Security.addProvider(new PlainSaslServerProvider());
```

- 更多细节, 请查看 JCA Reference.

5. 使用 SASL/SCRAM 认证

Salted Challenge Response Authentication Mechanism (SCRAM) 是 SASL 机制家族中的一员, 是用传统的机制解决安全问题, 执行像 PLAIN 和 DIGEST-MD5 一样的 username/password 认证. 这个机制在 RFC 5802 定义. Kafka 支持 SCRAM-SHA-256 和 SCRAM-SHA-512, 可以和 TLS 一起用户执行安全认证. username 在 ACL 等的配置中作为已认证的 `Principal`. Kafka 中默认的 SCRAM 实现是在 Zookeeper 中保存 SCRAM 证书, 适用于 Zookeeper 在私有网络的 Kafka 安装. 更多信息请查看 安全注意事项.

1. 创建 SCRAM 证书

Kafka 中的 SCRAM 实现使用 Zookeeper 作为证书存储. 可以使用 `kafka-configs.sh` 在 Zookeeper 中创建证书. 对每一个启用了 SCRAM 机制的, 必须通过添加机制名配置来创建证书. 用于 broker 间通信的证书必须在 Kafka broker 启动前创建. 客户端的证书可以动态创建和更新, 新的连接使用更新后的证书认证.

为用户 *alice* 创建密码为 *alice-secret* 的 SCRAM 凭证:

```
1  > bin/kafka-configs.sh --zookeeper localhost:2181 --alter --add-config 'SCRAM-SHA-256=[iterations=8192,password=alice-secret],SCRAM-SHA-512=[password=alice-s
```

如果未指定迭代次数, 将使用默认的迭代次数, 4096. 创建一个随机 salt, SCRAM 标识由 salt, 迭代和 StoredKey 组成. ServerKey 保存在 Zookeeper 中. 有关 SCRAM 的标识和各个字段的详细信息, 请参考 RFC 5802.

以下示例中, 关于 broker 间的通信, 需要一个 *admin* 用户. 使用如下命令创建:

```
1  > bin/kafka-configs.sh --zookeeper localhost:2181 --alter --add-config 'SCRAM-SHA-256=[password=admin-secret],SCRAM-SHA-512=[password=admin-secret]' --entity
```

使用 *--describe* 选项可以 列出所有存在的证书:

```
1  > bin/kafka-configs.sh --zookeeper localhost:2181 --describe --entity-type users --entity-name alice
```

使用 *--delete* 选项删除一个或多个 SCRAM 机制的证书:

```
> bin/kafka-configs.sh --zookeeper localhost:2181 --alter --delete-config 'SCRAM-SHA-512' --entity-type users --entity-name alice
```

2. **配置 Kafka Broker**

   1. 在每个 Kafka broker 的 config 目录, 添加一个类似下面的适当修改的 JAAS 文件. 在这个例子中, 我们将其命名为 kafka_server_jaas.conf:

      ```
      KafkaServer {
          org.apache.kafka.common.security.scram.ScramLoginModule require
          username="admin"
          password="admin-secret";
      };
      ```

      broker 使用 `KafkaServer` 部分中 `username` 和 `password` 属性来初始化与其他 broker 的连接. 在这个例子中, *admin* 是 broker 间通信的用户.
   2. JAAS 配置文件的路径作为 JVM 参数传递给每个 Kafka broker:

      ```
      -Djava.security.auth.login.config=/etc/kafka/kafka_server_jaas.conf
      ```

   3. 参考 这里 的描述, 在 server.properties 配置 SASL 端口和 SASL 机制. 例如:

      ```
      listeners=SASL_SSL://host.name:port
      security.inter.broker.protocol=SASL_SSL
      sasl.mechanism.inter.broker.protocol=SCRAM-SHA-256 (or SCRAM-SHA-51
      sasl.enabled.mechanisms=SCRAM-SHA-256 (or SCRAM-SHA-512)
      ```

3. **配置 Kafka Client**

   To configure SASL authentication on the clients: 在客户端配置 SASL 认证:

   1. 在每个客户端的 producer.properties 或者 consumer.properties 文件中配置 JAAS 配置项. 登录模块描述了像生产者和消费者这样的客户端如何连接到 Kafka Broker. 下面是一个 SCRAM 机制的客户端的配置示例:

      ```
      sasl.jaas.config=org.apache.kafka.common.security.scram.ScramLoginModule required \
          username="alice" \
          password="alice-secret";
      ```

      `username` 和 `password` 选项被客户端用于配置客户端连接的用户信息. 在这个例子中, 客户端作为 *alice* 用户连接到 broker. 通过在 `sasl.jaas.config` 中指定不同的用户名和密码, 在一个 JVM 内不同的客户端可以根据不同的用户连接.

      客户端的 JAAS 配置也可以像 broker 这里 描述的一样, 指定为一个 JVM 参数. 客户端使用命名为 `KafkaClient` 的登录部分. 此选项仅允许来自 JVM 的所有客户端连接中的一个用户.

2. 在 producer.properties 或者 consumer.properties 中配置下面这些属性:

```
security.protocol=SASL_SSL
sasl.mechanism=SCRAM-SHA-256 (or SCRAM-SHA-512)
```

4. **SASL/SCRAM 的安全注意事项**

- Kafka 中 SASL/SCRAM 的默认实现在 Zookeeper 中保存 SCRAM 证书. 这个适合当 Zookeeper 是安全并且是在私有网络时, 在生产环境安装.
- Kafka 只支持强散列函数 SHA-256 和 SHA-512, 和最小迭代数为4096. 如果 Zookeeper 安全性收到威胁, 强散列函数结合强密码, 高迭代次数可以防止暴力攻击.
- SCRAM 应只使用 TLS 加密, 防止 SCRAM 交换时的中途拦截. 如果 Zookeeper 受到威胁, 这可以防止字典或者暴力攻击, 和防止伪装模拟.
- 当 Zookeeper 不安全是, SASL/SCRAM 的默认实现可以在安装时使用自定义的登录模块重写. 更多细节查看 这里.
- 了解安全注意事项的更多细节, 参考 RFC 5802.

6. **Enabling multiple SASL mechanisms in a broker（broker 节点上启用多个 SASL 机制 ）**

1. 在JAAS配置文件的`KafkaServer` 部分中指定所有启用机制的登录模块的配置。例如：

```
        KafkaServer {
            com.sun.security.auth.module.Krb5LoginModule required
            useKeyTab=true
            storeKey=true
            keyTab="/etc/security/keytabs/kafka_server.keytab"
            principal="kafka/kafka1.hostname.com@EXAMPLE.COM";

            org.apache.kafka.common.security.plain.PlainLoginModule requi
            username="admin"
            password="admin-secret"
            user_admin="admin-secret"
            user_alice="alice-secret";
        };
```

2. 在 server.properties 配置文件中启用 SASL 机制：

```
    sasl.enabled.mechanisms=GSSAPI,PLAIN,SCRAM-SHA-256,SCRAM-SHA-512
```

3. 如果需要，可以在 server.properties 中指定 SASL 安全协议和 broker 间的通信机制：

```
    security.inter.broker.protocol=SASL_PLAINTEXT (or SASL_SSL)
    sasl.mechanism.inter.broker.protocol=GSSAPI (or one of the other enab
```

4. 遵循 GSSAPI (Kerberos)、 PLAIN 和 SCRAM 特定的步骤来配置启用 SASL 机制。

7. **Modifying SASL mechanism in a Running Cluster（在运行的集群里修改 SASL 机制）**

按照下面的步骤可以修改正在运行中的集群的 SASL 机制：

1. 为了给每个 broker 启用新的 SASL 机制，需要在 server.properties 中添加 `sasl.enabled.mechanisms` 配置。更新 这里描述的两种机制到JAAS配置文件。 依次更新每个集群节点
2. 使用新机制重新启动客户端。
3. 如果有必要，可以在 server.properties 文件中指定新的`sasl.mechanism.inter.broker.protocol`的配置，来修改 broker 间的通信机制。然后再次依次的更新集群。
4. 如果有必要，可以从 server.properties 文件中的`sasl.enabled.mechanisms` 配置中和 JAAS 配置文件中删除旧的机制条目，来删除旧的机制。 然后再次依次的更新集群。

7.4 Authorization and ACLs(授权和访问控制列表)

Kafka 带有一个可扩展的 Authorizer (授权器) 和一个用 zookeeper 实现的 Authorizer (授权器)，zookeeper 会存储所有的 acls 授权信息。 Kafka acls 授权信息定义的规则是 " P is [Allowed/Denied] Operation O From Host H On Resource R"。您可以在KIP-11上阅读更多关于 acl 结构信息。您可以通过Kafka认证器(authorizer)的CLI对acls进行添加、删除或查询。默认情况下，如果资源R没有关联acls 授权信息，那么除超级用户以外的任何人都不能访问资源R。如果要更改该行为，你可以在server.properties 文件中填写以下内容。

```
allow.everyone.if.no.acl.found=true
```

还可以在server.properties中添加超级用户，如下所示（请注意，由于SSL用户名可能包含逗号，因此分隔符是分号）

```
super.users=User:Bob;User:Alice
```

默认情况下，SSL用户名的格式为："CN=writeuser,OU=Unknown,O=Unknown,L=Unknown,ST=Unknown,C=Unknown"。可以通过在server.properties中设置一个自定义的PrincipalBuilder来改变这种情况，如下所示。

```
principal.builder.class=CustomizedPrincipalBuilderClass
```

默认情况下，SASL用户名将成为Kerberos主体的主要部分。有一种方式可以改变，通过设置`sasl.kerberos.principal.to.local.rules` 在server.properties中自定义规则。`sasl.kerberos.principal.to.local.rules`是一个列表，其中每个规则的工作方式与Kerberos configuration file (krb5.conf)中的auth_to_local相同。每一个规则以 RULE: 开始，并且包含一个格式为

[n:string](regexp)s/pattern/replacement/g的表达式。请查看kerberos文档获取更多细节。下面是添加一个规则的示例，该示例可以很正确地翻译user@MYDOMAIN.COM。并让user同时保持默认的规则:

```
sasl.kerberos.principal.to.local.rules=RULE:[1:$1@$0](.*@MYDOMAIN.COM)s/@.*//,
```

Command Line Interface(命令行界面)

Kafka的授权管理CLI(命令行界面)可以在bin目录下找到，并使用所有其他CLIs。命令行界面脚本的名字是**kafka-acls.sh**。以下列出了脚本支持的所有选项：

选项	说明	默认值	选项类型
--add	表示用户正在尝试添加acl的脚本		Action
--remove	表示用户正在尝试删除acl的脚本。		Action
--list	表示用户正在尝试列出acls的脚本。		Action
--authorizer	授权器的完全限定类名。	kafka.security.auth.SimpleAclAuthorizer	Configuration
--authorizer-properties	键值对(key=val)将被传递给授权器进行初始化。对于默认授权器，示例值为：zookeeper.connect = localhost：2181		Configuration
--cluster	指定一个集群作为资源。		Resource
--topic [topic-name]	指定一个topic(主题)作为资源。		Resource
--group [group-name]	指定consumer-group(消费组)作为资源。		Resource
--allow-principal	Principal(委托人)是PrincipalType:name的格式，将被添加到ACL中并有允许权限。 你可以在单个命令中指定多个 --allow-principal。		Principal
--deny-principal	Principal是PrincipalType:name格式，将被添加到ACL中并有拒绝权限。 你可以在单个命令中指定多个--deny-principal。		Principal
--allow-host	在--allow-principal中列出将允许principal从哪些IP地址访问。	如果指定 --allow-principal的默认值为 *，将允许从所有主机进行访问。	Host
--deny-host	在--deny-principal中列出将拒绝principal从哪些IP地址访问。	如果指定 --deny-principal的默认值为 *，将允许从所有主机进行访问。	Host
--operation	将会被允许和拒绝的操作 有效值: Read(读), Write(写), Create(创建), Delete(删除), Alter(修改), Describe(描述), ClusterAction(集群操作), All(所有)	All	Operation
--producer	该选项可为生产者添加和删除acls(访问控制列表)。它生成的acls允许在topic(主题)上进行WRITE, DESCRIBE操作，并且在集群上进行CREATE操作。		Convenience
--consumer	该选项可为消费者添加和删除acls(访问控制列表)。 它生成的acls允许在topic(主题)上进行READ, DESCRIBE操作，并且通过消费组进行READ操作。		Convenience
--force	该选项对所有的查询都假设为yes，并且不提示。		Convenience

Examples(例子)

- **Adding Acls**

  如果您想添加一个 acl 授权信息 "允许 User:Bob 和 User:Alice 从 IP 198.51.100.0 和 198.51.100.1 对 Topic 中的 Test-Topic 进行读写"。您可以按照下面的选项在终端执行 CLI :

  ```
  1  bin/kafka-acls.sh --authorizer-properties zookeeper.connect=localhost:2181 --add --allow-principal User:Bob --allow-principal User:Alice --allow-host 198.51.100.0 -
  ```

  默认情况下，所有的没有显示声明 acl 授权信息的用户，对某个资源操作时都将被拒绝。在极少情况下，我们需要在acl 授权信息定义一个少数用户不允许访问，其他用户都允许访问的规则，这个时候就需要使用 --deny-principal 和 --deny-host 选项。如果我们想允许所有用户对 Test-topic 进行读取，仅拒绝 User:BadBob 从 IP 198.51.100.3 进行读取。我们可以使用下面的命令：

  ```
  1  bin/kafka-acls.sh --authorizer-properties zookeeper.connect=localhost:2181 --add --allow-principal User:* --allow-host * --deny-principal User:BadBob --deny-host 19
  ```

  Note that ``--allow-host`` and ``deny-host`` only support IP addresses (hostnames are not supported). 注意：``--allow-host`` 和 ``deny-host`` 只支持 IP地址(不支持主机名) 以上示例通过--topic [topic-name]指定资

源选项来将acls添加到 topic 上。同样，用户可以通过指定 --cluster 将 acls 添加到 cluster；通过指定 --group [group-name] 将 acls 添加到一个 consumer group。

- **Removing Acls**

  移除 acl 授权信息几乎是相同的命令。唯一的不同就是必须使用 --remove 选项来代替 --add 选项。为了移除上面第一个示例添加的 acls 授权信息，我们可以用 CLI 终端执行下面的选项：

```
1  bin/kafka-acls.sh --authorizer-properties zookeeper.connect=localhost:2181 --remove --allow-principal User:Bob --allow-principal User:Alice --allow-host 198.51.100.
```

- **List Acls**

  我们可以通过指定 --list 选项和资源来列出任何资源的 acls 授权信息。如果我们想列出 Test-topic 的所有的acls授权信息,我们可以用 CLI 终端执行下面的选项：

```
1  bin/kafka-acls.sh --authorizer-properties zookeeper.connect=localhost:2181 --list --topic Test-topic
```

- **Adding or removing a principal as producer or consumer**

  ACL 管理最常见的情况就是添加或移除一个用户作为 producer 或 consumer。因此我们添加方便的选项来处理这些情景。为了添加 User:Bob 作为 Test-topic 的 producer 我们可以执行下面的命令：

```
1  bin/kafka-acls.sh --authorizer-properties zookeeper.connect=localhost:2181 --add --allow-principal User:Bob --producer --topic Test-topic
```

  类似的，如果要添加 Alice 到 consumer group Group-1 中作为 Test-topic 的一个 consumer，我们仅需要传递 --consumer 选项：

```
1  bin/kafka-acls.sh --authorizer-properties zookeeper.connect=localhost:2181 --add --allow-principal User:Bob --consumer --topic Test-topic --group Group-1
```

  注意：对于 consumer 选项我们必须指定 consumer group。 为了从 producer 和 consumer 的角色中移除一个用户我们仅需要传 --remove 选项

7.5 将安全功能应用在正在运行的集群中

通过前面讨论的安全协议，选择一个或者多个协议保护正在运行的集群。 您可以通过以下步骤来完善安全功能：

- 滚动重启集群节点开启额外的安全端口。
- 使用安全端口重启客户端而不是 PLAINTEXT 端口（假设你正在设置 client-broker 的安全连接 ）。
- 滚动重启集群节点开启 broker和broker之间的安全模式（如果需要的话）。
- 最后滚动关闭 PLAINTEXT 端口

关于配置 SSL 和 SASL 的详细步骤已经在 7.2 和 7.3 描述了。 遵循文档中的步骤配置你想要的安全协议，开启安全模式。

你可以为 broker-client 之间和 broker-broker 之间的通信配置不同的安全协议。 这两种协议必须分别启用。 PLAINTEXT 端口要同时保持开放，以便 broker 和 clients 能够继续通信

通过SIGTERM(终止进程) 滚动停止 brokers. 在停止下一个节点之前， 等待重新启动的节点的副本回到 ISR 列表是一种好的行为。

举个例子， 假设我们希望使用 SSL 加密 broker 与 client之间 和 broker和broker 之间的通信。 在第一次滚动重启的时候，每个节点上开启 SSL 端口：

```
listeners=PLAINTEXT://broker1:9091,SSL://broker1:9092
```

然后我们重启客户端， 将其配置更改为指向新打开的安全端口：

```
bootstrap.servers = [broker1:9092,...]
security.protocol = SSL
...etc
```

在第二次滚动重启集群节点时， 我们指定 Kafka 使用 SSL 作为 broker 之间的 安全协议（所有集群节点将使用相同的 SSL 端口）

```
listeners=PLAINTEXT://broker1:9091,SSL://broker1:9092
security.inter.broker.protocol=SSL
```

最后一次滚动重启节点的过程中， 我们通过关闭 PLAINTEXT 端口， 来确保集群的安全。

```
listeners=SSL://broker1:9092
security.inter.broker.protocol=SSL
```

另外， 我们可能使用多个端口，以便在 broker 与 client之间 和 broker与broker 之间使用不同的协议来通信。假设我们已经在 broker 与 client 和 broker 之间使用 SSL 加密， 而现在想在 broker 与 client 通信中添加 SASL 认证。在第一次滚动重启节点的时候，我们将开启两个额外的端口来实现两种不同的安全协议。

```
listeners=PLAINTEXT://broker1:9091,SSL://broker1:9092,SASL_SSL://b
```

然后我们重启客户端， 将其配置更改为新打开的安全端口（SASL 和 SSAL 端口）：

```
bootstrap.servers = [broker1:9093,...]
security.protocol = SASL_SSL
...etc
```

第二次滚动重启集群节点的时候， 通过使用我们之前打开的 SSL 端口（9092），将集群切换到 broker-broker 加密的通信。

```
listeners=PLAINTEXT://broker1:9091,SSL://broker1:9092,SASL_SSL://b
security.inter.broker.protocol=SSL
```

最后滚动重启集群节点， 我们通过关闭他们的 PLAINTEXT 端口， 来确保集群的安全。

```
        listeners=SSL://broker1:9092,SASL_SSL://broker1:9093
        security.inter.broker.protocol=SSL
```

ZooKeeper 可以独立于 Kafka 集群进行安全防护。 7.6.2 章节将介绍 ZooKeeper 的安全防护。

7.6 ZooKeeper 认证

7.6.1 新集群

在对broker进行Zookeeper认证前，有两个必要步骤：

1. 创建一个JAAS登录文件并在文件中设置适当的系统属性，如上文所述。
2. 在每个broker上设置配置项 `zookeeper.set.acl`为true。

The metadata stored in ZooKeeper for the Kafka cluster is world-readable,but can only be modified by the brokers. The rationale behind this decision is that the data stored in ZooKeeper is not sensitive,but inappropriate manipulation of that data can cause cluster disruption. We also recommend limiting the access to ZooKeeper via network segmentation (only brokers and some admin tools need access to ZooKeeper if the new Java consumer and producer clients are used). Kafka 存储在Zookeeper的元数据是全局可读的，但是只有 brokers 能够进行修改。 这一决定背后的理由是，存储在 zookeeper 上的数据虽然不重要，但是如果数据被肆意串改，会导致broker程序出错，程序中断。 我们还是建议通过ip 网段来限制访问 zookeeper 的权限（仅允许broker和一些管理工具访问zookeeper）。

7.6.2 集群迁移

如果你使用kafka的版本不支持安全的或简单的禁用安全，你还是想设置集群安全，则需要执行以下步骤启用ZooKeeper认证（最小的中断操作）：

1. 滚动重新启动设置的JAAS登录文件，这样可以使broker进行身份认证。在滚动重启结束后，broker就能够用ACL操作这些znode（节点）了（但不能创建）。
2. 进行第二次滚动重启，这次设置配置参数`zookeeper.set.acl`为true，这样就能使用安全的ACL创建znode。
3. 执行ZkSecurityMigrator工具。执行脚本：./bin/zookeeper-security-migration.sh，将zookeeper.acl 设置secure，这个工具通过相应的sub-trees修改的znodes ACL。

除了以上操作，还需要在安全集群中关闭认证，具体操作如下：

1. 滚动重新启动设置的JAAS登录文件，进行broker认证，这里需要将`zookeeper.set.acl`设置为false。 重启结束之后，broker停止用ACL创建znodes，但是仍然能认证和操作znodes。

2. 执行ZkSecurityMigrator工具. 运行脚本：`./bin/zookeeper-security-migration.sh`，将`zookeeper.acl`设置为unsecure，这个工具通过相应的sub-trees修改的znodes ACL。
3. 执行第二次滚动重启broker，这次忽略了JAAS登录文件设置系统属性。

这里提供了一个如何运行迁移工具的例子，：

```
./bin/zookeeper-security-migration.sh --zookeeper.acl=secure --zookeeper.connect=localhost:2181
```

运行以下命令查看完整列表：

```
./bin/zookeeper-security-migration.sh --help
```

7.6.3 ZooKeeper 整体迁移

在进行迁移时候有必要对全部zookeeper启用身份验证。要做到这一点，我们需要设置一些属性参数。请参阅更详细的zookeeper文档：:

1. Apache ZooKeeper documentation
2. Apache ZooKeeper wiki

8. KAFKA CONNECT

8.1 概述

Kafka Connect 是一款可扩展并且可靠地在 Apache Kafka 和其他系统之间进行数据传输的工具。 可以很简单的快速定义 *connectors* 将大量数据从 Kafka 移入和移出. Kafka Connect 可以摄取数据库数据或者收集应用程序的 metrics 存储到 Kafka topics，使得数据可以用于低延迟的流处理。 一个导出的 job 可以将来自 Kafka topic 的数据传输到二级存储，用于系统查询或者批量进行离线分析。

Kafka Connect 功能包括:

- **Kafka connectors 通用框架：** - Kafka Connect 将其他数据系统和Kafka集成标准化,简化了 connector 的开发,部署和管理
- **分布式和单机模式** - 可以扩展成一个集中式的管理服务，也可以单机方便的开发,测试和生产环境小型的部署。
- **REST 接口** - submit and manage connectors to your Kafka Connect cluster via an easy to use REST API
- **offset 自动管理** - 只需要connectors 的一些信息，Kafka Connect 可以自动管理offset 提交的过程，因此开发人员无需担心开发中offset提交出错的这部分。
- **分布式的并且可扩展** - Kafka Connect 构建在现有的 group 管理协议上。Kafka Connect 集群可以扩展添加更多的workers。
- **整合流处理/批处理** - 利用 Kafka 已有的功能，Kafka Connect 是一个桥接stream 和批处理系统理想的方式。

8.2 用户指南

quickstart 提供了一个简单的例子，演示如何运行一个单机版的Kafka Connect。 这一节描述如何配置，如何管理Kafka Connect 的更多细节。

运行 Kafka Connect

Kafka Connect 当前支持两种执行方式: 单机 (单个进程) 和 分布式.

在单机模式下所有的工作都是在一个进程中运行的。connect的配置项很容易配置和开始使用，当只有一台机器(worker)的时候也是可用的(例如，收集日志文件到kafka)，但是不利于Kafka Connect 的容错。你可以通过下面的命令启动一个单机进程:

```
1  > bin/connect-standalone.sh config/connect-standalone.properties connector1.properties [connector2.properties ...]
```

第一个参数是 worker 的配置文件. 其中包括 Kafka connection 参数，序列化格式，和如何频繁的提交 offsets。 所提供的示例可以在本地良好的运行，使用默认提供的配置 `config/server.properties` 。它需要调整以配合不同的配置或生产环境部署。所有的workers（独立和分布式）都需要一些配置 :

- `bootstrap.servers` - List of Kafka servers used to bootstrap connections to Kafka
- `key.converter` - Converter class used to convert between Kafka Connect format and the serialized form that is written to Kafka. This controls the format of the keys in messages written to or read from Kafka, and since this is independent of connectors it allows any connector to work with any serialization format. Examples of common formats include JSON and Avro.
- `value.converter` - Converter class used to convert between Kafka Connect format and the serialized form that is written to Kafka. This controls the format of the values in messages written to or read from Kafka, and since this is independent of connectors it allows any connector to work with any serialization format. Examples of common formats include JSON and Avro.

单机模式的重要配置如下:

- `offset.storage.file.filename` - 存储 offset 数据的文件

此处配置的参数适用于由Kafka Connect使用的 producer 和 consumer 访问配置，offset 和 status topic。对于 Kafka source和 sink 任务的配置，可以使用相同的参数，但必须以 `consumer.` 和 `producer.` 作为前缀。 此外，从 worker 配置中继承的参数只有一个，就是 `bootstrap.servers` 。大多数情况下，这是足够的，因为同一个集群通常用于所有的场景。但是需要注意的是一个安全集群，需要额外的参数才能允许连接。这些参数需要在 worker 配置中设置三次，一次用于管理访问，一次用于 Kafka sinks，还有一次用于 Kafka source。

其余参数用于 connector 的配置文件，你可以导入尽可能多的配置文件，但是所有的配置文件都将在同一个进程内(在不同的线程上)执行。

分布式模式下会自动进行负载均衡，允许动态的扩缩容，并提供对 active task，以及这个任务对应的配置和offset提交记录的容错。分布式执行方式和单机模式非常相似:

```
1  > bin/connect-distributed.sh config/connect-distributed.properties
```

和单机模式不同在于启动的实现类和决定 Kafka connect 进程如何工作的配置参数，如何分配 work,offsets 存储在哪里和任务状态。在分布式模式中，Kafka Connect 存储 offsets,配置和存储在 Kafka topic中的任务状

态。建议手动创建Kafka 的 offsets,配置和状态，以实现自己所期望的分区数和备份因子。如果启动Kafka Connect之前没有创建 topic，则会使用默认分区数和复制因子自动创建创建 topic，但这可能不是最适合的。

特别是，除了上面提到的常用设置之外，以下配置参数在启动集群之前至关重要:

- `group.id` (default `connect-cluster`) - unique name for the cluster, used in forming the Connect cluster group; note that this **must not conflict** with consumer group IDs
- `config.storage.topic` (default `connect-configs`) - topic to use for storing connector and task configurations; note that this should be a single partition, highly replicated, compacted topic. You may need to manually create the topic to ensure the correct configuration as auto created topics may have multiple partitions or be automatically configured for deletion rather than compaction
- `offset.storage.topic` (default `connect-offsets`) - topic to use for storing offsets; this topic should have many partitions, be replicated, and be configured for compaction
- `status.storage.topic` (default `connect-status`) - topic to use for storing statuses; this topic can have multiple partitions, and should be replicated and configured for compaction

注意在分布式模式下 connector 配置不会通过命令行传递。相反，会使用下面提到的 REST API来创建，修改和销毁 connectors。

Configuring Connectors

Connector 配置是简单的key-value 映射的格式。对于单机模式，这些配置会在 properties 文件中定义，并通过命令行传递给 Connect 进程。在分布式模式中，它们将被包含在创建（或修改）connector 的请求的JSON格式串中。

大多数配置都依赖于 connectors,所以在这里不能概述。但是，有几个常见选项可以看一下:

- `name` - Unique name for the connector. Attempting to register again with the same name will fail.
- `connector.class` - The Java class for the connector
- `tasks.max` - The maximum number of tasks that should be created for this connector. The connector may create fewer tasks if it cannot achieve this level of parallelism.
- `key.converter` - (optional) Override the default key converter set by the worker.
- `value.converter` - (optional) Override the default value converter set by the worker.

`connector.class` 配置支持多种名称格式：这个 connector class 的全名或者别名。如果 connector 是 org.apache.kafka.connect.file.FileStreamSinkConnector，则可以指定全名，也可以使用FileStreamSink 或 FileStreamSinkConnector 来简化配置。

Sink connectors 还有一个额外的选项来控制他的输出:

- `topics` - A list of topics to use as input for this connector

对于任何其他选项，你应该查阅 connector的文档.

Transformations

connectors可以配置 transformations 操作，实现轻量级的消息单次修改，他们可以方便地用于数据修改和事件路由。

A transformation chain 可以在connector 配置中指定。

- `transforms` - List of aliases for the transformation, specifying the order in which the transformations will be applied.
- `transforms.$alias.type` - Fully qualified class name for the transformation.
- `transforms.$alias.$transformationSpecificConfig` Configuration properties for the transformation

例如，让我们使用内置的 file soucre connector，并使用 transformation 来添加静态字段。

这个例子中，我们会使用 schemaless json 数据格式。为了使用 schemaless 格式，我们将 `connect-standalone.properties` 文件中下面两行从true改成false:

```
key.converter.schemas.enable
value.converter.schemas.enable
```

这个 file source connector 读取每行数据作为一个字符串。我们会将每行数据包装进一个 Map 数据结构,然后添加一个二级字段来标识事件的来源。做这样一个操作，我们使用两个 transformations:

- **HoistField** to place the input line inside a Map
- **InsertField** to add the static field. In this example we'll indicate that the record came from a file connector

添加完 transformations, `connect-file-source.properties` 文件像下面这样:

```
name=local-file-source
connector.class=FileStreamSource
tasks.max=1
file=test.txt
topic=connect-test
transforms=MakeMap, InsertSource
transforms.MakeMap.type=org.apache.kafka.connect.transforms.HoistField$Value
transforms.MakeMap.field=line
transforms.InsertSource.type=org.apache.kafka.connect.transforms.InsertField$Value
transforms.InsertSource.static.field=data_source
transforms.InsertSource.static.value=test-file-source
```

所有以 `transforms` 为开头的行都将被添加了静态字段用于 transformations 。 你可以看到我们创建的两个 transformations: "InsertSource" 和 "MakeMap" 是我们给的 transformations 的别称. transformation 类型基于下面给的一系列内嵌 transformations。每个 transformation 类型都有额外的配置: HoistField 需要一个配置叫做 "field”,这是 map中原始字符串的字段名称。InsertField transformation 让我们指定字段名称和我们要添加的内容。

当我们对一个 sample file 运行 file source connector 操作，不做transformations 操作，然后使用 `kafka-console-consumer.sh` 读取数据，结果如下:

```
"foo"
"bar"
"hello world"
```

然后我们创建一个新的file connector,然后将这个transformations 添加到配置文件中。这次结果如下:

```
{"line":"foo","data_source":"test-file-source"}
```

```
2  {"line":"bar","data_source":"test-file-source"}
3  {"line":"hello world","data_source":"test-file-source"}
```

你可以看到我们读取的行现在JSON map的一部分，并且还有一个静态值的额外字段。这只是用transformations 做的一个简单的例子。.

Kafka Connect 包含几个广泛适用的数据和 routing transformations

- InsertField - Add a field using either static data or record metadata
- ReplaceField - Filter or rename fields
- MaskField - Replace field with valid null value for the type (0, empty string, etc)
- ValueToKey
- HoistField - Wrap the entire event as a single field inside a Struct or a Map
- ExtractField - Extract a specific field from Struct and Map and include only this field in results
- SetSchemaMetadata - modify the schema name or version
- TimestampRouter - Modify the topic of a record based on original topic and timestamp. Useful when using a sink that needs to write to different tables or indexes based on timestamps
- RegexRouter - modify the topic of a record based on original topic, replacement string and a regular expression

如何配置每个transformation，参考下面:

org.apache.kafka.connect.transforms.InsertField

Insert field(s) using attributes from the record metadata or a configured static value.

Use the concrete transformation type designed for the record key (`org.apache.kafka.connect.transforms.InsertField$Key`) or value (`org.apache.kafka.connect.transforms.InsertField$Value`).

NAME	DESCRIPTION	TYPE	DEFAULT	VALID VALUES	IMPORTANCE
offset.field	Field name for Kafka offset - only applicable to sink connectors. Suffix with `!` to make this a required field, or `?` to keep it optional (the default).	string	null		medium
partition.field	Field name for Kafka partition. Suffix with `!` to make this a required field, or `?` to keep it optional (the default).	string	null		medium
static.field	Field name for static data field. Suffix with `!` to make this a required field, or `?` to keep it optional (the default).	string	null		medium
static.value	Static field value, if field name configured.	string	null		medium
timestamp.field	Field name for record timestamp. Suffix with `!` to make this a required field, or `?` to keep it optional (the default).	string	null		medium
topic.field	Field name for Kafka topic. Suffix with `!` to make this a required field, or `?` to keep it optional (the default).	string	null		medium

org.apache.kafka.connect.transforms.ReplaceField

Filter or rename fields.

Use the concrete transformation type designed for the record key (`org.apache.kafka.connect.transforms.ReplaceField$Key`) or value (`org.apache.kafka.connect.transforms.ReplaceField$Value`).

NAME	DESCRIPTION	TYPE	DEFAULT	VALID VALUES	IMPORTANCE
blacklist	Fields to exclude. This takes precedence over the whitelist.	list	""		medium
renames	Field rename mappings.	list	""	list of colon-delimited pairs, e.g. `foo:bar,abc:xyz`	medium
whitelist	Fields to include. If specified, only these fields will be used.	list	""		medium

org.apache.kafka.connect.transforms.MaskField

Mask specified fields with a valid null value for the field type (i.e. 0, false, empty string, and so on).

Use the concrete transformation type designed for the record key (`org.apache.kafka.connect.transforms.MaskField$Key`) or value (`org.apache.kafka.connect.transforms.MaskField$Value`).

NAME	DESCRIPTION	TYPE	DEFAULT	VALID VALUES	IMPORTANCE
fields	Names of fields to mask.	list		non-empty list	high

org.apache.kafka.connect.transforms.ValueToKey

Replace the record key with a new key formed from a subset of fields in the record value.

NAME	DESCRIPTION	TYPE	DEFAULT	VALID VALUES	IMPORTANCE
fields	Field names on the record value to extract as the record key.	list		non-empty list	high

org.apache.kafka.connect.transforms.HoistField

Wrap data using the specified field name in a Struct when schema present, or a Map in the case of schemaless data.

Use the concrete transformation type designed for the record key (`org.apache.kafka.connect.transforms.HoistField$Key`) or value (`org.apache.kafka.connect.transforms.HoistField$Value`).

NAME	DESCRIPTION	TYPE	DEFAULT	VALID VALUES	IMPORTANCE
field	Field name for the single field that will be created in the resulting Struct or Map.	string			medium

org.apache.kafka.connect.transforms.ExtractField

Extract the specified field from a Struct when schema present, or a Map in the case of schemaless data. Any null values are passed through unmodified.

Use the concrete transformation type designed for the record key (`org.apache.kafka.connect.transforms.ExtractField$Key`) or value (`org.apache.kafka.connect.transforms.ExtractField$Value`).

NAME	DESCRIPTION	TYPE	DEFAULT	VALID VALUES	IMPORTANCE
field	Field name to extract.	string			medium

org.apache.kafka.connect.transforms.SetSchemaMetadata

Set the schema name, version or both on the record's key (`org.apache.kafka.connect.transforms.SetSchemaMetadata$Key`) or value (`org.apache.kafka.connect.transforms.SetSchemaMetadata$Value`) schema.

NAME	DESCRIPTION	TYPE	DEFAULT	VALID VALUES	IMPORTANCE
schema.name	Schema name to set.	string	null		high
schema.version	Schema version to set.	int	null		high

org.apache.kafka.connect.transforms.TimestampRouter

Update the record's topic field as a function of the original topic value and the record timestamp.

This is mainly useful for sink connectors, since the topic field is often used to determine the equivalent entity name in the destination system(e.g. database table or search index name).

NAME	DESCRIPTION	TYPE	DEFAULT	VALID VALUES	IMPORTANCE
timestamp.format	Format string for the timestamp that is compatible with `java.text.SimpleDateFormat`.	string	yyyyMMdd		high
topic.format	Format string which can contain `${topic}` and `${timestamp}` as placeholders for the topic and timestamp, respectively.	string	${topic}-${timestamp}		high

org.apache.kafka.connect.transforms.RegexRouter

Update the record topic using the configured regular expression and replacement string.

Under the hood, the regex is compiled to a `java.util.regex.Pattern`. If the pattern matches the input topic, `java.util.regex.Matcher#replaceFirst()` is used with the replacement string to obtain the new topic.

NAME	DESCRIPTION	TYPE	DEFAULT	VALID VALUES	IMPORTANCE
regex	Regular expression to use for matching.	string		valid regex	high
replacement	Replacement string.	string			high

org.apache.kafka.connect.transforms.Flatten

Flatten a nested data structure, generating names for each field by concatenating the field names at each level with a configurable delimiter character. Applies to Struct when schema present, or a Map in the case of schemaless data. The default delimiter is '.'.

Use the concrete transformation type designed for the record key (`org.apache.kafka.connect.transforms.Flatten$Key`) or value (`org.apache.kafka.connect.transforms.Flatten$Value`).

NAME	DESCRIPTION	TYPE	DEFAULT	VALID VALUES	IMPORTANCE
delimiter	Delimiter to insert between field names from the input record when generating field names for the output record	string	.		medium

org.apache.kafka.connect.transforms.Cast

Cast fields or the entire key or value to a specific type, e.g. to force an integer field to a smaller width. Only simple primitive types are supported -- integers, floats, boolean, and string.

Use the concrete transformation type designed for the record key (`org.apache.kafka.connect.transforms.Cast$Key`) or value (`org.apache.kafka.connect.transforms.Cast$Value`).

NAME	DESCRIPTION	TYPE	DEFAULT	VALID VALUES	IMPORTANCE
spec	List of fields and the type to cast them to of the form field1:type,field2:type to cast fields of Maps or Structs. A single type to cast the entire value. Valid types are int8, int16, int32, int64, float32, float64, boolean, and string.	list		list of colon-delimited pairs, e.g. `foo:bar,abc:xyz`	high

org.apache.kafka.connect.transforms.TimestampConverter

Convert timestamps between different formats such as Unix epoch, strings, and Connect Date/Timestamp types.Applies to individual fields or to the entire value.

Use the concrete transformation type designed for the record key (`org.apache.kafka.connect.transforms.TimestampConverter$Key`) or value (`org.apache.kafka.connect.transforms.TimestampConverter$Value`).

NAME	DESCRIPTION	TYPE	DEFAULT	VALID VALUES	IMPORTANCE
target.type	The desired timestamp representation: string, unix, Date, Time, or Timestamp	string			high
field	The field containing the timestamp, or empty if the entire value is a timestamp	string	""		high
format	A SimpleDateFormat-compatible format for the timestamp. Used to generate the output when type=string or used to parse the input if the input is a string.	string	""		medium

REST API

由于Kafka Connect 旨在作为服务运行，它还提供了一个用于管理 connectors 的REST API。sssss默认情况下，此服务在端口8083上运行。以下是当前支持的功能:

- `GET /connectors` - return a list of active connectors
- `POST /connectors` - create a new connector; the request body should be a JSON object containing a string `name` field and an object `config` field with the connector configuration parameters
- `GET /connectors/{name}` - get information about a specific connector
- `GET /connectors/{name}/config` - get the configuration parameters for a specific connector
- `PUT /connectors/{name}/config` - update the configuration parameters for a specific connector
- `GET /connectors/{name}/status` - get current status of the connector, including if it is running, failed, paused, etc., which worker it is assigned to, error information if it has failed, and the state of all its tasks
- `GET /connectors/{name}/tasks` - get a list of tasks currently running for a connector
- `GET /connectors/{name}/tasks/{taskid}/status` - get current status of the task, including if it is running, failed, paused, etc., which worker it is assigned to, and error information if it has failed

- `PUT /connectors/{name}/pause` - pause the connector and its tasks, which stops message processing until the connector is resumed
- `PUT /connectors/{name}/resume` - resume a paused connector (or do nothing if the connector is not paused)
- `POST /connectors/{name}/restart` - restart a connector (typically because it has failed)
- `POST /connectors/{name}/tasks/{taskId}/restart` - restart an individual task (typically because it has failed)
- `DELETE /connectors/{name}` - delete a connector, halting all tasks and deleting its configuration

Kafka Connect还提供用于获取有关 connector plugin sss信息的REST API:

- `GET /connector-plugins` - return a list of connector plugins installed in the Kafka Connect cluster. Note that the API only checks for connectors on the worker that handles the request, which means you may see inconsistent results, especially during a rolling upgrade if you add new connector jars
- `PUT /connector-plugins/{connector-type}/config/validate` - validate the provided configuration values against the configuration definition. This API performs per config validation, returns suggested values and error messages during validation.

8.3 Connector 开发者指南

本指南介绍开发人员如何为 Kafka connector 编写新的 connectors，用于Kafka和其他系统之间移动数据。

Core Concepts 和 APIs

Connectors 和 Tasks

`HDFSSinkConnector`

要在Kafka和另一个系统之间复制数据，用户会为想要 pull 数据或者 push 数据的系统创建一个 `connector`。 connector 有两类: `SourceConnectors` 从其他系统导入数据 (e.g. `JDBCSourceConnector` 会将关系型数据库导入到Kafka中)和 `SinkConnectors` 导出数据(e.g. `HDFSSinkConnector` 会将Kafka topic 的内容导出到 HDFS 文件)

`Connectors` 自身不执行任何数据复制: `Connector` 的配置描述要复制的数据，并且 `Connector` 负责负责将 job 分解为可分发给 worker 的一组 `Tasks`。这些 `Tasks` 也分为两类: `SourceTask` 和 `SinkTask`。

通过分配，每个 `Task` 必须将数据的一部分子集复制到Kafka或者从Kafka复制。在 Kafka Connect中，应该始终可以将这些分配的数据框架化为一组输入和输出流，这些流由具有一致结构的记录组成。有的时候这些映射是显而易见的:一组日志文件中的每个文件都认为是一个流，每条解析的行数据使用相同的模式和偏移量作为字节偏移量存储在文件中。在其他情况下，可能需要花费更多功夫来映射模型: 一个 JDBC connector可以将表映射成stream，但是offset不能确定。可以使用时间戳字段进行增量查询返回新的数据，最后查询的时间戳可以用作偏移量。

Streams 和 Records

每个stream 都应该是一串 key-value的就。keys和values可以有复杂的数据结构-提供了很多基本类型，arrays，objects和嵌套的数据结构。每个stream 都应该是一串 key-value的就。keys和values可以有复杂的数据结构-提供了很多基本类型，也可以用来表示arrays，objects和嵌套的数据结构。运行时的数据格式不承担任何特定的序列化格式:此转换由框架内部处理。

除了 key 和 value, records(sources生成的记录和发送到sinks的记录) 关联 stream IDs和offsets。框架会定期的提交已经处理数据的offsets，以便在发生故障时，可以从最后一次提交的offsets恢复处理，避免不必要的重新处理和重复事件。.

Dynamic Connectors

不是所有的jobs都是静态，所以 `Connector` 的实现还要负责监控外部系统是否需要重新更改配置。例如，在 `JDBCSourceConnector` 的例子中，这个 `Connector` 可能分配一组 tables 给 `Task` 。当一个新的 table 创建了，必须发现这个时间，以便通过更新配置来将新表分配给其中一个 `Tasks` 。当发现需要重新配置的变更(或者 `Tasks` 数量)的时候，他会通知框架，并更新相应的 `Tasks` 。

开发一个简单的 Connector

开发一个 connector 只需要实现两个接口, `Connector` 和 `Task` 接口. 一个简单的例子的源码在 Kafka `file` package中。 connector 用于单机模式，并拥有 `SourceConnector` 和 `SourceTask` 实现来读取一个文件的每行记录，并将其作为记录发发送， `SinkConnector` 的 `SinkTask` 将记录写入到文件。

本节的其余部分将通过一些代码演示创建 connector 的关键步骤，但开发人员还应参考完整的示例源代码，因为为简洁起见，省略了许多细节。

Connector Example

首先我们用一个简单的例子介绍一下 `SourceConnector` ， `SinkConnector` 的实现和它非常相似。首先创建一个继承自 `SourceConnector` 的类，并添加一些字段来存储解析出的配置信息(要读取的文件名以及要发送数据的topic):

```
public class FileStreamSourceConnector extends SourceConnector {
    private String filename;
    private String topic;
```

最简单的方法是 `taskClass()` ，它定义了应该在 worker 进程中实例化以实际读取数据的类:

```
@Override
public Class<? extends Task> taskClass() {
    return FileStreamSourceTask.class;
}
```

我们会在后面定义 `FileStreamSourceTask` 类。接下来，我们要往 `FileStreamSourceConnector` 类中添加一些标准的生命周期方法。

:

```
@Override
public void start(Map<String, String> props) {
    // The complete version includes error handling as well.
    filename = props.get(FILE_CONFIG);
    topic = props.get(TOPIC_CONFIG);
}

@Override
public void stop() {
    // Nothing to do since no background monitoring is required.
}
```

最后，实现的真正核心是 `taskConfigs()` 。因此即使允许我们按照 `maxTasks` 参数创建更多的 task，我们也只返回包含一个 entry 的 list：

```
@Override
public List<Map<String, String>> taskConfigs(int maxTasks) {
    ArrayList<Map<String, String>> configs = new ArrayList<>();
    // Only one input stream makes sense.
    Map<String, String> config = new HashMap<>();
    if (filename != null)
        config.put(FILE_CONFIG, filename);
    config.put(TOPIC_CONFIG, topic);
    configs.add(config);
    return configs;
}
```

尽管这个示例中没有用到，但 `SourceTask` 还提供了两个用于提交源系统中的 offset 的 API： `commit` 和 `commitRecord` 。这些 API 是为具有消息确认机制的源系统提供的。覆盖这些方法允许 source connector 在源系统中的消息被写入 Kafka 后，批量或单独确认消息。 `commit` API 将偏移量存储在源系统中， `poll` 方法每次返回的偏移量都会被存储起来。这个API 的实现应该一直处于阻塞状态直到提交完成。在每个 `SourceRecord` 被写入 Kafka 之后， `commitRecord` API 会将其偏移量保存在源系统中。由于 Kafka Connect 会自动记录偏移量，因此不需要 `SourceTask` 来实现它们。在 connector 确实需要确认源系统中的消息的情况下，通常只需要其中一个API。

即使是多任务, 这个方法实现通常也很简单。它只需要确定 input task 的数量，这可能需要联系它从中拉取数据的远程服务，然后由这些 task 来分摊工作。由于这些多个 task 分摊工作的的模式非常普遍，因为在 `ConnectorUtils` 工具类中提供了一些实用程序来简化这些模式。

注意，这个简单的例子没有包括动态输入。有关如何触发 task 配置的更新，请参阅下一节的讨论。

Task Example - Source Task

接下来我们将描述相应的 `SourceTask` 的实现。 这里的实现很简短, 但如果要想完全涵盖本指南的内容就太长了。我们将用伪代码描述大多数实现，你也可以参考完整示例的源代码。

就像 Connector 一样，我们需要创建一个类并继承对应的基类 `Task` 。 它也有一些标准的生命周期方法：

```
public class FileStreamSourceTask extends SourceTask {
    String filename;
    InputStream stream;
    String topic;

    @Override
    public void start(Map<String, String> props) {
        filename = props.get(FileStreamSourceConnector.FILE_CONFIG);
        stream = openOrThrowError(filename);
        topic = props.get(FileStreamSourceConnector.TOPIC_CONFIG);
    }

    @Override
    public synchronized void stop() {
        stream.close();
    }
```

这是一个轻量级的简化版本，但表明这些方法应该相对简单，并且它们唯一要做的就是分配和释放资源。 关于这个实现有两点要注意： 第一， `start()` 方法没有处理从之前的 offset 恢复的情形，这一点将在后面的

章节中讨论。第二， `stop()` 方法是同步的。这是必须的，因为 `SourceTasks` 有一个专用的线程，可以无限期的阻塞下去，所以它们需要通过 Worker 中另一个线程的调用来停止。

接下来，我们要实现 task 的主要功能， `poll()` 方法负责从输入系统获取事件并返回一个 `List<SourceRecord>` ：

```
@Override
public List<SourceRecord> poll() throws InterruptedException {
    try {
        ArrayList<SourceRecord> records = new ArrayList<>();
        while (streamValid(stream) && records.isEmpty()) {
            LineAndOffset line = readToNextLine(stream);
            if (line != null) {
                Map<String, Object> sourcePartition = Collections.singletonMap("filename", filename);
                Map<String, Object> sourceOffset = Collections.singletonMap("position", streamOffset);
                records.add(new SourceRecord(sourcePartition, sourceOffset, topic, Schema.STRING_SCHEMA, line));
            } else {
                Thread.sleep(1);
            }
        }
        return records;
    } catch (IOException e) {
        // Underlying stream was killed, probably as a result of calling stop. Allow to return
        // null, and driving thread will handle any shutdown if necessary.
    }
    return null;
}
```

同样，我们也省略了一些细节，但是我们可以看到重要的步骤： `poll()` 方法会被反复调用，并且对于每次调用，它会循环尝试从文件中读取记录。对于它读取的每一行，它也会跟踪文件偏移量。它使用这些信息创建一个输出 `SourceRecord` ，带有四部分信息：source partition（本示例中只有一个，即正在读取的单个文件），source offset（文件中的字节偏移量），output topic name 和 output value（读取的行，并且我们使用一个模式指定该值始终是一个字符串）。 SourceRecord构造函数的其他变体还可以包含特定的输出分区和密钥。

.注意，这个实现使用了普通的Java `InputStream` 接口，没有可用数据时会休眠。这是可以接受的，因为 Kafka 为每个 task 提供了一个专用线程。虽然 task 的实现必须覆盖基本的 `poll()` 接口，但实现起来有很大的灵活性。在这种情况下，基于 NIO 的实现将更加高效，但这种简单的方法很有效，实现起来很快，并且与旧版本的 Java 兼容。

Sink Tasks

之前的部分描述了如何实现一个简单的 `SourceTask` 。与 `SourceConnector` 和 `SinkConnector` 不同的是， `SourceTask` 和 `SinkTask` 有着完全不同的接口： `SourceTask` 使用 pull 接口， `SinkTask` 使用 push 接口。两者共享相同的生命周期方法，但 `SinkTask` 接口有点特殊：

```
public abstract class SinkTask implements Task {
    public void initialize(SinkTaskContext context) {
        this.context = context;
    }

    public abstract void put(Collection<SinkRecord> records);

    public void flush(Map<TopicPartition, OffsetAndMetadata> currentOffsets) {
    }
```

`SinkTask` 的文档包含了所有的细节，但这个接口几乎和 `SourceTask` 一样简单。 `put()` 方法应包含了大部分实现，包括接受 `SinkRecords` 集合，以及执行任何必需的转换，并将它们存储在目标系统中。此方法无需确保数据在返回之前已完全写入目标系统。实际上，在许多情况下 internal buffering 会很有用，使得我们可以一次发送整批记录，从而减少将事件插入下游数据存储的开销。 `SinkRecords` 而本质上和 `SourceRecords` 有着相同的信息: Kafka topic, partition, offset and the event key and value。

`flush()` 方法用于提交 offset 数据，这使得 task 可以从故障中恢复并且从安全点恢复，因此不会有事件丢失。该方法应该将任何未完成的数据推送到目标系统，然后阻塞，直到写入被确认。 参数 `offsets` 通常会被忽略，但在某些情况下很有用，例如实现需要在目标存储区中存储 offset 信息以提供精确的 exactly-once 交付。例如，HDFS connector 可以做到这一点，它使用原子性移动操作来确保 `flush()` 操作以原子方式将数据和 offset 提交到 HDFS 中的最终位置。

Resuming from Previous Offsets

`SourceTask` 的实现包含每条记录的 stream ID (本示例中是输入的文件名) 和 offset (记录在文件中的位置)。该框架使用它定期提交 offset 数据，以便在发生故障的情况下，task 可以恢复并且最小化 reprocess 和可能重复的事件的数量（或者从最近 Kafka Connect 正常停止过的 offset 恢复，例如在独立模式下或作业重新配置导致的停止）。这个提交过程由框架完全自动化，但只有 connector 知道如何退回到 input stream 中的正确位置并从该位置恢复。

为了在启动时正确恢复，task 可以使用它的 `initialize()` 方法中传入的 `SourceContext` 来访问 offset 数据。在 `initialize()` 中，我们会添加更多的代码来读取 offset 数据（如果存在）并寻找对应的位置：

```
stream = new FileInputStream(filename);
Map<String, Object> offset = context.offsetStorageReader().offset(Collections.singletonMap(FILENAME_FIELD, filename));
if (offset != null) {
    Long lastRecordedOffset = (Long) offset.get("position");
    if (lastRecordedOffset != null)
        seekToOffset(stream, lastRecordedOffset);
}
```

当然，您可能需要为每个 input stream 读取多个 key。`OffsetStorageReader` 接口允许您发出批量读取以有效加载所有 offset，然后通过查找每个输入流来把它们放到对应的位置。

动态输入/输出流

Kafka Connect 旨在定义批量数据复制作业，例如复制整个数据库，而不是创建许多作业以分别复制每张表。这种设计的一个结果是，connector 的输入或输出流的集合可以随时间变化。

Source connector 需要监视源系统的变化，例如数据库中表的增加与删除。当数据库中的表发生改变时，他们应通过 `ConnectorContext` 对象通知框架需要重新配置。例如，在 `SourceConnector` 中：

```
if (inputsChanged())
    this.context.requestTaskReconfiguration();
```

该框架将及时请求新的配置信息并更新 task，使它们能够在再次配置之前正常地提交进度。请注意，在 `SourceConnector` 中，该监视功能当前由 connector 实现决定。如果需要额外的线程来执行监视，则连接器必须自行分配线程。

理想情况下，用于监视变动的的代码将独立于 `Connector`，不会对 task 有任何影响。然而，变动也可以影响 task，最常见的是其输入流之一在输入系统中被破坏，例如，如果一个表从数据库中删除。如果 `Task` 在 `Connector` 之前遇到问题，这在 `Connector` 轮询数据源变动的时候很常见，需要由 `Task` 处理后续的 error。谢天谢地，这通常可以通过捕捉和处理适当的 exception 来处理。

`SinkConnectors` 通常只需要处理 stream 中增加的内容，这可能会转化为其输出中的新条目（例如新的数据库表）。该框架负责管理 Kafka 输入发生的任何更改，例如，由于批量订阅而导致输入主题集发生更改时。 `SinkTasks` 应该期望新的输入流，这可能需要在下游系统中创建新资源，例如数据库中的新表。在这些情况下最棘手的情况可能是多个 `SinkTasks` 第一次看到新的输入流并同时尝试创建新的资源。另一方面， `SinkConnectors` 通常不需要特殊的代码来处理动态的流集。

连接配置验证

Kafka Connect 允许在提交要执行的 connector 之前验证 connector 配置，并且可以提供有关错误和建议值的反馈。为了利用这一点，连接器开发人员需要提供一个 `config()` 的实现来将配置定义公开给框架。

下面 `FileStreamSourceConnector` 中的代码定义了配置并将其公开给框架。

```
private static final ConfigDef CONFIG_DEF = new ConfigDef()
    .define(FILE_CONFIG, Type.STRING, Importance.HIGH, "Source filename.")
    .define(TOPIC_CONFIG, Type.STRING, Importance.HIGH, "The topic to publish data to");

public ConfigDef config() {
    return CONFIG_DEF;
}
```

`ConfigDef` 类用于指定一组预期的配置。对于每个配置，您可以指定名称，类型，默认值，文档，组信息，组中的顺序，配置值的宽度以及适合在UI中显示的名称。另外，您可以通过覆盖 `Validator` 类来提供用于单个配置验证的特殊验证逻辑。此外，由于配置之间可能存在依赖关系，例如，配置的有效值和可见性可能会根据配置中的其它值而改变。为了解决这个问题， `ConfigDef` 允许你指定配置的依赖关系，并提供了一个 `Recommender` 的实现来获取有效的值，并给定当前配置值的配置可见性。

另外， `Connector` 中的 `validate()` 方法提供了一个默认验证的实现，该实现返回允许配置的列表以及每个配置的配置错误和推荐值。但是，它不使用建议的值进行配置验证。您可以覆盖默认实现以提供自定义的配置验证，该自定义配置验证可能使用推荐值。

处理模式

FileStream connector 是个很好的例子，因为它们很简单，但它们也有着简单的结构化数据——每行只是一个字符串。几乎所有实用的 connector 都需要具有更复杂数据格式的模式。

要创建更复杂的数据，您需要使用 Kafka Connect 的 `data` API。大多数结构化记录除了原始类型外还需要与两个类进行交互： `Schema` 和 `Struct` 。

其 API 文档提供了一个完整的参考，但下面是一个创建 `Schema` 和 `Struct` 的简单示例：：

```
Schema schema = SchemaBuilder.struct().name(NAME)
    .field("name", Schema.STRING_SCHEMA)
    .field("age", Schema.INT_SCHEMA)
    .field("admin", new SchemaBuilder.boolean().defaultValue(false).build())
    .build();

Struct struct = new Struct(schema)
    .put("name", "Barbara Liskov")
    .put("age", 75);
```

如果您正在实现一个 source connector，则需要确定何时以及如何创建模式。在可能的情况下，应尽可能避免重新计算它们。例如，如果您的 connector 确定有固定模式，请静态创建并重用单个实例。

但是，许多 connector 都有动态模式。一个简单的例子就是 database connector。考虑到即使数据库只有一个表，也不会为整个 connector 预定义模式（因为它随表格而变化）。但是在 connector 的整个生命周期中，它也可能不会因为只有一张表而使用固定模式，因为用户可能会执行 `ALTER TABLE` 命令，connector必须能够检测到这些变化并做出适当的反应。

Sink connectors 通常更简单，因为它们在消耗数据，因此不需要创建模式。但是，他们应该同样仔细地验证他们收到的模式是否具有预期的格式。当模式不匹配时，通常表明上游 produce 正在生成无法正确转换到目标系统的无效数据，sink connector 应该抛出异常来向系统指示此错误。

Kafka Connect Administration

Kafka Connect 的 REST layer 层提供了一组 API 对群集进行管理。这包括查看 connector 配置和任务状态的 API，以及更改其当前行为的 API（例如更改配置和重新启动任务）。

当 connector 首次提交到群集时，worker 将重新平衡群集中的全部 connector 及其 task，以便每个 worker的工作量大致相同。当 connector 增加或减少它们需要的任务数量或连接器的配置发生变化时，也使用相同的重新平衡过程。您可以使用 REST API 来查看 connector 及其 task 的当前状态，包括每个 connector 分配的 worker 的 ID。例如，查询文件源的状态（使用 GET/connectors/file-source/status）可能会产生如下输出：

```
{
"name": "file-source",
"connector": {
    "state": "RUNNING",
    "worker_id": "192.168.1.208:8083"
},
"tasks": [
    {
    "id": 0,
    "state": "RUNNING",
    "worker_id": "192.168.1.209:8083"
    }
]
}
```

Connector及其 task 将状态更新发布到群集中所有 worker 监视的共享主题（使用status.storage.topic配置）。由于 worker 异步使用此主题，因此状态更改通过状态 API 可见之前通常会有（短）延迟。以下状态可用于 connector 或其 task 之一：

- **UNASSIGNED:** Connector/Task 还没有分配给一个worker。
- **RUNNING:** Connector/Task 正在运行。
- **PAUSED:** Connector/Task 已经被管理暂停。
- **FAILED:** Connector/Task 失败了(通常是通过引发一个异常，在状态输出中报告)。

在大多数情况下，connector 和 task 状态都会匹配，但在发生变动或 task 失败时，它们在短时间内可能会有所不同。例如，首次启动 connector 时，connector 及其 task 全部转换为 RUNNING 状态之前可能会有明显的延迟。由于 Connect 不会自动重启失败的 task，因此 task 失败时状态也会发生变化。要手动重新启动 connector/task，可以使用上面列出的重启 API。请注意，如果尝试在重新平衡过程中重启 task，Connect 将返回409（冲突）状态码。您可以在重新平衡完成后重试，但可能没有必要，因为重新平衡实际上会重启群集中的所有 connector 和 task。

有时候暂时停止 connector 的消息处理很有用。例如，如果远程系统正在进行维护，则 source connector 最好停止从该系统中轮询获取新数据，而不是使用异常垃圾消息填充日志。对于这个用例，Connect 提供了一个

pause/resume API。当 source connector 暂停时，Connect 将停止轮询它以获取其他记录。当 sink connector 暂停时，Connect 将停止向其发送新消息。暂停状态是持久化的，因此即使重新启动群集，connector 也不会再次开始消息处理，直到 task 恢复。请注意，在所有 connector 的 task 都转换到 PAUSED 状态之前可能会有一段延迟，因为它们可能需要一些时间才能完成暂停过程中的任何处理。另外，失败的任务在重启之前不会转换到 PAUSED 状态。

9. KAFKA STREAMS

Kafka Streams 是一个用于处理和分析存储在 Kafka 系统中的数据的客户端库。 它建立在重要的流处理概念上，如恰当地区分事件时间（event time）和处理时间（processing time），支持窗口操作（window），exactly-once 处理语义以及简单高效的应用程序状态管理。

Kafka Streams 的**入门门槛很低**。我们可以在单节点环境上快速实现一个小规模的验证性的程序，只要程序能在多节点的集群环境成功运行即可部署到高负载的生产环境。 Kafka Streams 通过利用 Kafka 的并行模型实现对相同应用程序的多个实例的负载平衡，这对于用户来说是透明的。

Learn More about Kafka Streams read this Section.