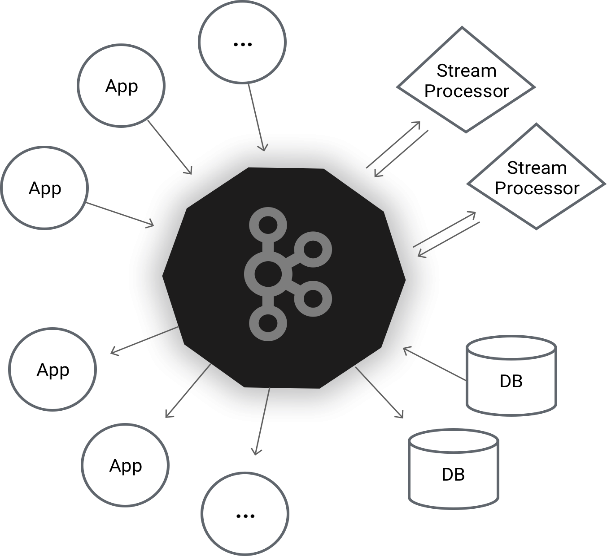
参考：<http://orchome.com/kafka/index>

**kafka中文教程**

本网翻译整理Apache kafka，提供整理Apache kafka的完整学习文档。

### 发布 & 订阅              处理                     存储

数据流，如消息传递系统 高效并实时数据流安全地在分布式集群中复制存储                              

kafka是用于构建实时数据管道和流应用程序。具有横向扩展，容错，wicked fast（变态快）等优点，并已在成千上万家公司运行。

## 简单说明什么是kafka

Apache kafka是消息中间件的一种，我发现很多人不知道消息中间件是什么，在开始学习之前，我这边就先简单的解释一下什么是消息中间件，只是粗略的讲解，目前kafka已经可以做更多的事情。

举个例子，生产者消费者，生产者生产鸡蛋，消费者消费鸡蛋，生产者生产一个鸡蛋，消费者就消费一个鸡蛋，假设消费者消费鸡蛋的时候噎住了（系统宕机了），生产者还在生产鸡蛋，那新生产的鸡蛋就丢失了。再比如生产者很强劲（大交易量的情况），生产者1秒钟生产100个鸡蛋，消费者1秒钟只能吃50个鸡蛋，那要不了一会，消费者就吃不消了（消息堵塞，最终导致系统超时），消费者拒绝再吃了，”鸡蛋“又丢失了，这个时候我们放个篮子在它们中间，生产出来的鸡蛋都放到篮子里，消费者去篮子里拿鸡蛋，这样鸡蛋就不会丢失了，都在篮子里，而这个篮子就是”kafka“。  
鸡蛋其实就是“数据流”，系统之间的交互都是通过“数据流”来传输的（就是tcp、http什么的），也称为报文，也叫“消息”。  
消息队列满了，其实就是篮子满了，”鸡蛋“ 放不下了，那赶紧多放几个篮子，其实就是kafka的扩容。  
各位现在知道kafka是干什么的了吧，它就是那个"篮子"。

## kafka名词解释

后面大家会看到一些关于kafka的名词，比如topic、producer、consumer、broker，我这边来简单说明一下。

* producer：生产者，就是它来生产“鸡蛋”的。
* consumer：消费者，生出的“鸡蛋”它来消费。
* topic：你把它理解为标签，生产者每生产出来一个鸡蛋就贴上一个标签（topic），消费者可不是谁生产的“鸡蛋”都吃的，这样不同的生产者生产出来的“鸡蛋”，消费者就可以选择性的“吃”了。
* broker：就是篮子了。

大家一定要学会抽象的去思考，上面只是属于业务的角度，如果从技术角度，topic标签实际就是队列，生产者把所有“鸡蛋（消息）”都放到对应的队列里了，消费者到指定的队列里取。

## 怎么样才算真正的学会kafka

最近面试发现，很多人用过kafka，但是没人了解原理，我们可是很注重原理的(PS:要不然怎么知道你真的会呢)。

* kafka节点之间如何复制备份的？
* kafka消息是否会丢失？为什么？
* kafka最合理的配置是什么？
* kafka的leader选举机制是什么？
* kafka对硬件的配置有什么要求？
* kafka的消息保证有几种方式？

。。。。。。 你是否都答得上来？（欢迎大家补充！）

这些问题在下面的文章中都可以找到答案，kafka之所以有这么火热，建议各位一定要看一下。

## 如何学习kafka

还是那句话，学习任何技术，跟学骑自行车一样，不要一开始只关注它的具体细节是什么。先学着怎么骑，骑着骑着就了解大致的原理，这个时候在去看它的原理，会很轻松。

如果你在学习的过程中遇到什么问题，直接评论或者在问题专区中提问，我们在线人员会回复你的问题。

#### 章节与kafka官网对应一致

第一章：开始  
第二章：API  
第三章：kafka的配置  
第四章：kafka如何设计的  
第五章：kafka的实现  
第六章：kafka的常用操作，如扩容，删除和增加topic。  
第七章：安全  
第八章：kafka Connect  
第九章：kafka 流  
第十章：笔记（[kafka命令大全](http://orchome.com/454)）

# Apache Kafka下载

作者：半兽人  
链接：http://orchome.com/5  
来源：OrcHome  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

你可以登录Apache kafka 官方下载。  
<http://kafka.apache.org/downloads.html>

### 注意:别下成源文件了！

带src的是源文件,如：

Source download: kafka-0.10.1.0-src.tgz (asc, md5)

你应该下的是：

Scala2.11 - kafka\_2.11-0.10.1.0.tgz (asc, md5)

推荐下载scala 2.11版本的。

# kafka入门介绍

## Kafka作为一个分布式的流平台，这到底意味着什么？

我们认为，一个流处理平台具有三个关键能力：

1. 发布和订阅消息（流），在这方面，它类似于一个消息队列或企业消息系统。
2. 以容错的方式存储消息（流）。
3. 在消息流发生时处理它们。

##### 什么是kakfa的优势？

它应用于2大类应用：

1. 构建实时的流数据管道，可靠地获取系统和应用程序之间的数据。
2. 构建实时流的应用程序，对数据流进行转换或反应。

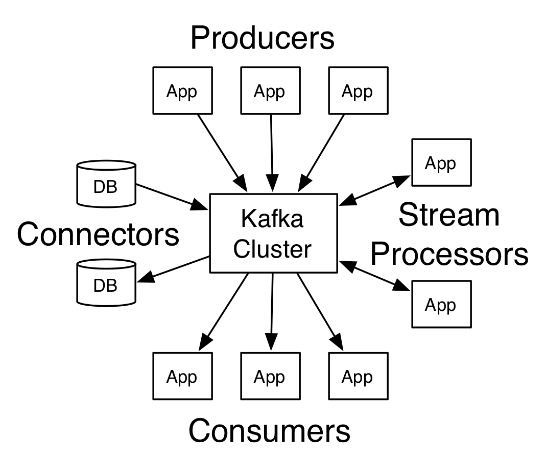
要了解kafka是如何做这些事情的，让我们从下到上深入探讨kafka的能力。

##### 首先几个概念：

1. kafka作为一个集群运行在一个或多个服务器上。
2. kafka集群存储的消息是以topic为类别记录的。
3. 每个消息（也叫记录record，我习惯叫消息）是由一个key，一个value和时间戳构成。

##### kafka有四个核心API：

* 应用程序使用 Producer API 发布消息到1个或多个topic（主题）。
* 应用程序使用 Consumer API 来订阅一个或多个topic，并处理产生的消息。
* 应用程序使用 Streams API 充当一个流处理器，从1个或多个topic消费输入流，并生产一个输出流到1个或多个输出topic，有效地将输入流转换到输出流。
* Connector API允许构建或运行可重复使用的生产者或消费者，将topic连接到现有的应用程序或数据系统。例如，一个关系数据库的连接器可捕获每一个变化。



Client和Server之间的通讯，是通过一条简单、高性能并且和开发语言无关的[TCP协议](https://kafka.apache.org/protocol.html)。并且该协议保持与老版本的兼容。Kafka提供了Java Client（客户端）。除了Java Client外，还有非常多的[其它编程语言的Client](https://cwiki.apache.org/confluence/display/KAFKA/Clients)。

## 首先来了解一下Kafka所使用的基本术语：

##### Topic

Kafka将消息种子(Feed)分门别类，每一类的消息称之为一个主题(Topic).

##### Producer

发布消息的对象称之为主题生产者(Kafka topic producer)

##### Consumer

订阅消息并处理发布的消息的种子的对象称之为主题消费者(consumers)

##### Broker

已发布的消息保存在一组服务器中，称之为Kafka集群。集群中的每一个服务器都是一个代理(Broker). 消费者可以订阅一个或多个主题（topic），并从Broker拉数据，从而消费这些已发布的消息。

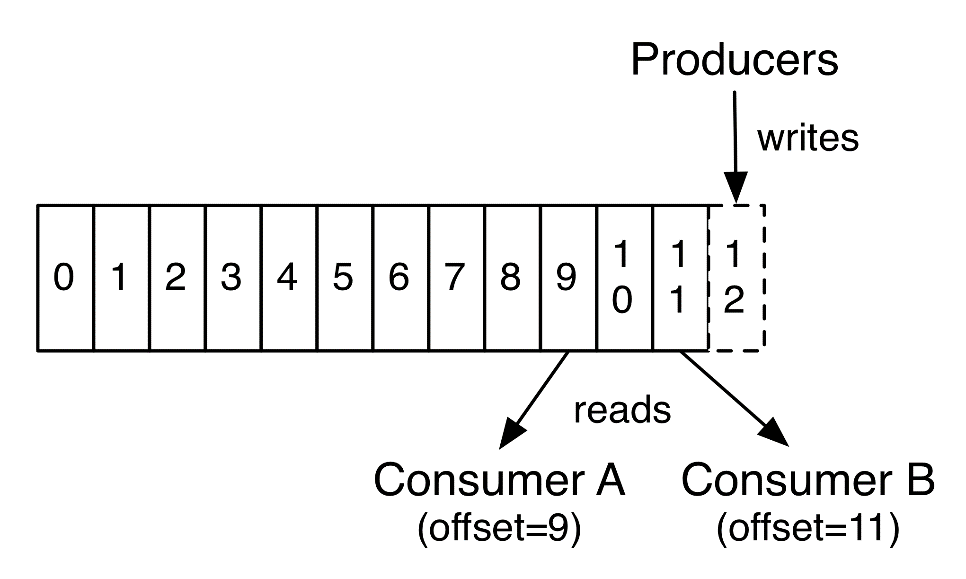
## 话题和日志 (Topic和Log)

让我们更深入的了解Kafka中的Topic。

Topic是发布的消息的类别或者种子Feed名。对于每一个Topic，Kafka集群维护这一个分区的log，就像下图中的示例：



每一个分区都是一个顺序的、不可变的消息队列， 并且可以持续的添加。分区中的消息都被分了一个序列号，称之为偏移量(offset)，在每个分区中此偏移量都是唯一的。

Kafka集群保持所有的消息，直到它们过期， 无论消息是否被消费了。 实际上消费者所持有的仅有的元数据就是这个偏移量，也就是消费者在这个log中的位置。 这个偏移量由消费者控制：正常情况当消费者消费消息的时候，偏移量也线性的的增加。但是实际偏移量由消费者控制，消费者可以将偏移量重置为更老的一个偏移量，重新读取消息。 可以看到这种设计对消费者来说操作自如， 一个消费者的操作不会影响其它消费者对此log的处理。 再说说分区。Kafka中采用分区的设计有几个目的。一是可以处理更多的消息，不受单台服务器的限制。Topic拥有多个分区意味着它可以不受限的处理更多的数据。第二，分区可以作为并行处理的单元，稍后会谈到这一点。  


## 分布式(Distribution)

Log的分区被分布到集群中的多个服务器上。每个服务器处理它分到的分区。 根据配置每个分区还可以复制到其它服务器作为备份容错。 每个分区有一个leader，零或多个follower。Leader处理此分区的所有的读写请求，而follower被动的复制数据。如果leader宕机，其它的一个follower会被推举为新的leader。 一台服务器可能同时是一个分区的leader，另一个分区的follower。 这样可以平衡负载，避免所有的请求都只让一台或者某几台服务器处理。

## 生产者(Producers)

生产者往某个Topic上发布消息。生产者也负责选择发布到Topic上的哪一个分区。最简单的方式从分区列表中轮流选择。也可以根据某种算法依照权重选择分区。开发者负责如何选择分区的算法。

## 消费者(Consumers)

通常来讲，消息模型可以分为两种， 队列和发布-订阅式。 队列的处理方式是 一组消费者从服务器读取消息，一条消息只有其中的一个消费者来处理。在发布-订阅模型中，消息被广播给所有的消费者，接收到消息的消费者都可以处理此消息。Kafka为这两种模型提供了单一的消费者抽象模型： 消费者组 （consumer group）。 消费者用一个消费者组名标记自己。 一个发布在Topic上消息被分发给此消费者组中的一个消费者。 假如所有的消费者都在一个组中，那么这就变成了queue模型。 假如所有的消费者都在不同的组中，那么就完全变成了发布-订阅模型。 更通用的， 我们可以创建一些消费者组作为逻辑上的订阅者。每个组包含数目不等的消费者， 一个组内多个消费者可以用来扩展性能和容错。正如下图所示：  


2个kafka集群托管4个分区（P0-P3），2个消费者组，消费组A有2个消费者实例，消费组B有4个。

正像传统的消息系统一样，Kafka保证消息的顺序不变。 再详细扯几句。传统的队列模型保持消息，并且保证它们的先后顺序不变。但是， 尽管服务器保证了消息的顺序，消息还是异步的发送给各个消费者，消费者收到消息的先后顺序不能保证了。这也意味着并行消费将不能保证消息的先后顺序。用过传统的消息系统的同学肯定清楚，消息的顺序处理很让人头痛。如果只让一个消费者处理消息，又违背了并行处理的初衷。 在这一点上Kafka做的更好，尽管并没有完全解决上述问题。 Kafka采用了一种分而治之的策略：分区。 因为Topic分区中消息只能由消费者组中的唯一一个消费者处理，所以消息肯定是按照先后顺序进行处理的。但是它也仅仅是保证Topic的一个分区顺序处理，不能保证跨分区的消息先后处理顺序。 所以，如果你想要顺序的处理Topic的所有消息，那就只提供一个分区。

## Kafka的保证(Guarantees)

* 生产者发送到一个特定的Topic的分区上，消息将会按照它们发送的顺序依次加入，也就是说，如果一个消息M1和M2使用相同的producer发送，M1先发送，那么M1将比M2的offset低，并且优先的出现在日志中。
* 消费者收到的消息也是此顺序。
* 如果一个Topic配置了复制因子（replication factor）为N， 那么可以允许N-1服务器宕机而不丢失任何已经提交（committed）的消息。

有关这些保证的更多详细信息，请参见文档的设计部分。

## kafka作为一个消息系统

##### Kafka的流与传统企业消息系统相比的概念如何？

传统的消息有两种模式：队列和发布订阅。 在队列模式中，消费者池从服务器读取消息（每个消息只被其中一个读取）; 发布订阅模式：消息广播给所有的消费者。这两种模式都有优缺点，队列的优点是允许多个消费者瓜分处理数据，这样可以扩展处理。但是，队列不像多个订阅者，一旦消息者进程读取后故障了，那么消息就丢了。而发布和订阅允许你广播数据到多个消费者，由于每个订阅者都订阅了消息，所以没办法缩放处理。

kafka中消费者组有两个概念：队列：消费者组（consumer group）允许同名的消费者组成员瓜分处理。发布订阅：允许你广播消息给多个消费者组（不同名）。

kafka的每个topic都具有这两种模式。

##### kafka有比传统的消息系统更强的顺序保证。

传统的消息系统按顺序保存数据，如果多个消费者从队列消费，则服务器按存储的顺序发送消息，但是，尽管服务器按顺序发送，消息异步传递到消费者，因此消息可能乱序到达消费者。这意味着消息存在并行消费的情况，顺序就无法保证。消息系统常常通过仅设1个消费者来解决这个问题，但是这意味着没用到并行处理。

kafka做的更好。通过并行topic的parition —— kafka提供了顺序保证和负载均衡。每个partition仅由同一个消费者组中的一个消费者消费到。并确保消费者是该partition的唯一消费者，并按顺序消费数据。每个topic有多个分区，则需要对多个消费者做负载均衡，但请注意，相同的消费者组中不能有比分区更多的消费者，否则多出的消费者一直处于空等待，不会收到消息。

## kafka作为一个存储系统

所有发布消息到消息队列和消费分离的系统，实际上都充当了一个存储系统（发布的消息先存储起来）。Kafka比别的系统的优势是它是一个非常高性能的存储系统。

写入到kafka的数据将写到磁盘并复制到集群中保证容错性。并允许生产者等待消息应答，直到消息完全写入。

kafka的磁盘结构 - 无论你服务器上有50KB或50TB，执行是相同的。

client来控制读取数据的位置。你还可以认为kafka是一种专用于高性能，低延迟，提交日志存储，复制，和传播特殊用途的分布式文件系统。

## kafka的流处理

仅仅读，写和存储是不够的，kafka的目标是实时的流处理。

在kafka中，流处理持续获取输入topic的数据，进行处理加工，然后写入输出topic。例如，一个零售APP，接收销售和出货的输入流，统计数量或调整价格后输出。

可以直接使用producer和consumer API进行简单的处理。对于复杂的转换，Kafka提供了更强大的Streams API。可构建聚合计算或连接流到一起的复杂应用程序。

助于解决此类应用面临的硬性问题：处理无序的数据，代码更改的再处理，执行状态计算等。

Sterams API在Kafka中的核心：使用producer和consumer API作为输入，利用Kafka做状态存储，使用相同的组机制在stream处理器实例之间进行容错保障。

## 拼在一起

消息传递，存储和流处理的组合看似反常，但对于Kafka作为流式处理平台的作用至关重要。

像HDFS这样的分布式文件系统允许存储静态文件来进行批处理。这样系统可以有效地存储和处理来自过去的历史数据。

传统企业的消息系统允许在你订阅之后处理未来的消息：在未来数据到达时处理它。

Kafka结合了这两种能力，这种组合对于kafka作为流处理应用和流数据管道平台是至关重要的。

批处理以及消息驱动应用程序的流处理的概念：通过组合存储和低延迟订阅，流处理应用可以用相同的方式对待过去和未来的数据。它是一个单一的应用程序，它可以处理历史的存储数据，当它处理到最后一个消息时，它进入等待未来的数据到达，而不是结束。

同样，对于流数据管道（pipeline），订阅实时事件的组合使得可以将Kafka用于非常低延迟的管道；但是，可靠地存储数据的能力使得它可以将其用于必须保证传递的关键数据，或与仅定期加载数据或长时间维护的离线系统集成在一起。流处理可以在数据到达时转换它。

有关Kafka提供的保证，api和功能的更多信息，可继续查阅本网

# Kafka的使用场景

下面是一些关于Apache kafka 流行的使用场景。这些领域的概述，可查看[博客文章](https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying)。

## 消息

kafka更好的替换传统的消息系统，消息系统被用于各种场景（解耦数据生产者，缓存未处理的消息，等），与大多数消息系统比较，kafka有更好的吞吐量，内置分区，副本和故障转移，这有利于处理大规模的消息。

根据我们的经验，消息往往用于较低的吞吐量，但需要低的端到端延迟，并需要提供强大的耐用性的保证。

在这一领域的kafka比得上传统的消息系统，如ActiveMQ或RabbitMQ。

## 网站活动追踪

kafka原本的使用场景：用户的活动追踪，网站的活动（网页游览，搜索或其他用户的操作信息）发布到不同的话题中心，这些消息可实时处理，实时监测，也可加载到Hadoop或离线处理数据仓库。

每个用户页面视图都会产生非常高的量。

## 指标

kafka也常常用于监测数据。分布式应用程序生成的统计数据集中聚合。

## 日志聚合

使用kafka代替一个日志聚合的解决方案。

## 流处理

kafka消息处理包含多个阶段。其中原始输入数据是从kafka主题消费的，然后汇总，丰富，或者以其他的方式处理转化为新主题，例如，一个推荐新闻文章，文章内容可能从“articles”主题获取；然后进一步处理内容，得到一个处理后的新内容，最后推荐给用户。这种处理是基于单个主题的实时数据流。从0.10.0.0开始，轻量，但功能强大的流处理，就进行这样的数据处理了。

除了Kafka Streams，还有Apache Storm和Apache Samza可选择。

## 事件采集

事件采集是一种应用程序的设计风格，其中状态的变化根据时间的顺序记录下来，kafka支持这种非常大的存储日志数据的场景。

## 提交日志

kafka可以作为一种分布式的外部提交日志，日志帮助节点之间复制数据，并作为失败的节点来恢复数据重新同步，kafka的日志压缩功能很好的支持这种用法，这种用法类似于ApachaBookKeeper项目。

**kafka安装和启动**

kafka的背景知识已经讲了很多了，让我们现在开始实践吧，假设你现在没有Kafka和ZooKeeper环境。

#### Step 1: 下载代码

下载0.10.0.0版本并且解压它。

> tar -xzf kafka\_2.11-0.10.0.0.tgz

> cd kafka\_2.11-0.10.0.0

#### Step 2: 启动服务

运行kafka需要使用Zookeeper，所以你需要先启动Zookeeper，如果你没有Zookeeper，你可以使用kafka自带打包和配置好的Zookeeper。

> 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

> 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: 创建一个主题(topic)

创建一个名为“test”的Topic，只有一个分区和一个备份：

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

创建好之后，可以通过运行以下命令，查看已创建的topic信息：

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

test

或者，除了手工创建topic外，你也可以配置你的broker，当发布一个不存在的topic时自动创建topic。

#### Step 4: 发送消息

Kafka提供了一个命令行的工具，可以从输入文件或者命令行中读取消息并发送给Kafka集群。每一行是一条消息。  
运行producer（生产者）,然后在控制台输入几条消息到服务器。

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

This is a message

This is another message

#### Step 5: 消费消息

Kafka也提供了一个消费消息的命令行工具，将存储的信息输出出来。

> bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic test --from-beginning

This is a message

This is another message

如果你有2台不同的终端上运行上述命令，那么当你在运行生产者时，消费者就能消费到生产者发送的消息。  
所有的命令行工具有很多的选项，你可以查看文档来了解更多的功能。

#### Step 6: 设置多个broker集群

到目前，我们只是单一的运行一个broker,，没什么意思。对于Kafka,一个broker仅仅只是一个集群的大小, 所有让我们多设几个broker.  
首先为每个broker创建一个配置文件:

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

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

现在编辑这些新建的文件，设置以下属性：

config/server-1.properties:

broker.id=1

listeners=PLAINTEXT://:9093

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

config/server-2.properties:

broker.id=2

listeners=PLAINTEXT://:9094

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

broker.id是集群中每个节点的唯一且永久的名称，我们修改端口和日志分区是因为我们现在在同一台机器上运行，我们要防止broker在同一端口上注册和覆盖对方的数据。

我们已经运行了zookeeper和刚才的一个kafka节点，所有我们只需要在启动2个新的kafka节点。

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

...

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

...

现在，我们创建一个新topic，把备份设置为：3

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

好了，现在我们已经有了一个集群了，我们怎么知道每个集群在做什么呢？运行命令“describe topics”

> 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

这是一个解释输出，第一行是所有分区的摘要，每一行提供一个分区信息，因为我们只有一个分区，所以只有一行。

* "leader"：该节点负责所有指定分区的读和写，每个节点的领导都是随机选择的。
* "replicas":备份的节点，无论该节点是否是leader或者目前是否还活着，只是显示。
* "isr"：备份节点的集合，也就是活着的节点集合。

我们运行这个命令，看看一开始我们创建的那个节点：

> 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

没有惊喜，刚才创建的topic（主题）没有Replicas，所以是0。

让我们来发布一些信息在新的topic上：

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

...

my test message 1

my test message 2

^C

现在，消费这些消息。

> bin/kafka-console-consumer.sh --zookeeper localhost:2181 --from-beginning --topic my-replicated-topic

...

my test message 1

my test message 2

^C

我们要测试集群的容错，kill掉leader，Broker1作为当前的leader，也就是kill掉Broker1。

> ps | grep server-1.properties

7564 ttys002 0:15.91 /System/Library/Frameworks/JavaVM.framework/Versions/1.6/Home/bin/java...

> kill -9 7564

备份节点之一成为新的leader，而broker1已经不在同步备份集合里了。

> 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

但是，消息仍然没丢：

> bin/kafka-console-consumer.sh --zookeeper localhost:2181 --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

从控制台写入和写回数据是一个方便的开始，但你可能想要从其他来源导入或导出数据到其他系统。对于大多数系统，可以使用kafka Connect，而不需要编写自定义集成代码。Kafka Connect是导入和导出数据的一个工具。它是一个可扩展的工具，运行连接器，实现与自定义的逻辑的外部系统交互。在这个快速入门里，我们将看到如何运行Kafka Connect用简单的连接器从文件导入数据到Kafka主题，再从Kafka主题导出数据到文件，首先，我们首先创建一些种子数据用来测试：

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

接下来，我们开始2个连接器运行在独立的模式，这意味着它们运行在一个单一的，本地的，专用的进程。我们提供3个配置文件作为参数。第一个始终是kafka Connect进程，如kafka broker连接和数据库序列化格式，剩下的配置文件每个指定的连接器来创建，这些文件包括一个独特的连接器名称，连接器类来实例化和任何其他配置要求的。

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

这是示例的配置文件，使用默认的本地集群配置并创建了2个连接器：第一个是导入连接器，从导入文件中读取并发布到Kafka主题，第二个是导出连接器，从kafka主题读取消息输出到外部文件，在启动过程中，你会看到一些日志消息，包括一些连接器实例化的说明。一旦kafka Connect进程已经开始，导入连接器应该读取从

test.txt

和写入到topic

connect-test

,导出连接器从主题

connect-test

读取消息写入到文件

test.sink.txt

. 我们可以通过验证输出文件的内容来验证数据数据已经全部导出：

cat test.sink.txt

foo

bar

注意，导入的数据也已经在Kafka主题

connect-test

里,所以我们可以使用该命令查看这个主题：

bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic connect-test --from-beginning

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

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

...

连接器继续处理数据，因此我们可以添加数据到文件并通过管道移动：

echo "Another line" >> test.txt

你应该会看到出现在消费者控台输出一行信息并导出到文件。

#### Step 8: 使用Kafka Stream来处理数据

Kafka Stream是kafka的客户端库，用于实时流处理和分析存储在kafka broker的数据，这个快速入门示例将演示如何运行一个流应用程序。一个WordCountDemo的例子（为了方便阅读，使用的是java8 lambda表达式）

KTable wordCounts = textLines

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

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

// Ensure the words are available as record keys for the next aggregate operation.

.map((key, value) -> new KeyValue<>(value, value))

// Count the occurrences of each word (record key) and store the results into a table named "Counts".

.countByKey("Counts")

它实现了wordcount算法，从输入的文本计算出一个词出现的次数。然而，不像其他的WordCount的例子，你可能会看到，在有限的数据之前，执行的演示应用程序的行为略有不同，因为它的目的是在一个无限的操作，数据流。类似的有界变量，它是一种动态算法，跟踪和更新的单词计数。然而，由于它必须假设潜在的无界输入数据，它会定期输出其当前状态和结果，同时继续处理更多的数据，因为它不知道什么时候它处理过的“所有”的输入数据。

现在准备输入数据到kafka的topic中，随后kafka Stream应用处理这个topic的数据。

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

接下来，使用控制台的producer 将输入的数据发送到指定的topic（streams-file-input）中，（在实践中，stream数据可能会持续流入，其中kafka的应用将启动并运行）

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

--zookeeper localhost:2181 \

--replication-factor 1 \

--partitions 1 \

--topic streams-file-input

> cat /tmp/file-input.txt | ./bin/kafka-console-producer --broker-list localhost:9092 --topic streams-file-input

现在，我们运行 WordCount 处理输入的数据：

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

不会有任何的STDOUT输出，除了日志，结果不断地写回另一个topic（streams-wordcount-output），demo运行几秒，然后，不像典型的流处理应用程序，自动终止。

现在我们检查WordCountDemo应用，从输出的topic读取。

> ./bin/kafka-console-consumer --zookeeper localhost:2181

--topic streams-wordcount-output

--from-beginning

--formatter kafka.tools.DefaultMessageFormatter

--property print.key=true

--property print.value=true

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

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

输出数据打印到控台（你可以使用Ctrl-C停止）：

all 1

streams 1

lead 1

to 1

kafka 1

hello 1

kafka 2

streams 2

join 1

kafka 3

summit 1

^C

第一列是message的key，第二列是message的value，要注意，输出的实际是一个连续的更新流，其中每条数据（即：原始输出的每行）是一个单词的最新的count，又叫记录键“kafka”。对于同一个key有多个记录，每个记录之后是前一个的更新。

**kafka的生态系统**

还有很多与kafka集成的外部的工具。更多信息点击[这里](https://cwiki.apache.org/confluence/display/KAFKA/Ecosystem)，包含了stream处理系统，hadoop的集成，监控和部署工具。

**从老版本升级kafka**

## 从老版本升级kafka

从0.8.x, 0.9.x 或 0.10.0.X 升级到 0.10.1.0

0.10.1.0有线协议更改，通过遵循以下建议的滚动升级，在升级期间不会停机。但是，需要注意升0.10.1.0中潜在的突发状况。

**注意**：由于引入了新的协议，要在升级客户端之前先升级kafka集群（即，0.10.1.x仅支持 0.10.1.x或更高版本的broker，但是0.10.1.x的broker向下支持旧版本的客户端）

##### 滚动升级:

1. 更新所有broker的server.properties文件，并添加以下属性：
   * inter.broker.protocol.version=CURRENT\_KAFKA\_VERSION (如：0.8.2.0, 0.9.0.0或0.10.0.0).
   * log.message.format.version=CURRENT\_KAFKA\_VERSION (有关此配置的详细信息，请查看升级后潜在的性能影响。)
2. 每次升级一个broker：关闭broker，替换新版本，然后重新启动它。
3. 一旦整个群集升级，通过编辑inter.broker.protocol.version并将其设置为0.10.1.0来转换所有协议。
4. 如果之前的消息格式是0.10.0，则将log.message.format.version更改为0.10.1（这无影响，因为0.10.0和0.10.1的消息格式是相同的）。如果之前的消息格式版本低于.10.0,还不能更改log.message.format.version - 一旦所有的消费者都已升级到 0.10.0.0 或更高版本时，才能更改此参数。
5. 逐个重新启动broker，使新协议版本生效。
6. 如果log.message.format.version低于0.10.0，请等待，知道所有消费者升级到0.10.0或更新的版本，然后将每个broker的log.message.format.version更改为0.10.1。然后逐个重启。

**注意**：如果你可接受停机，你可以简单地将所有broker关闭，更新版本并重启启动，它们将默认从新版本开始。

**注意**：变换协议版本和重启启动可以在broker升级完成后的任何时间去做，不必马上做。

### 在0.10.1.0中潜在的变化

* 日志保留时间不再基于日志段的最后修改时间。相反，它将基于日志段中消息的最大时间戳。
* 日志滚动时间不再取决于日志段的创建时间。而是基于消息中的时间戳。进一步来说。如果日志段中第一个消息的时间戳是T，则当新的消息的时间戳大于或等于T+log.roll.ms时，日志将推出。
* 0.10.0 的打开的文件处理将增加了约33%，因为为每个段增加时间索引文件。
* 时间索引和offset索引共享相同的索引大小配置。因为每个时间索引条目是offset索引条目的1.5备。用户可能需要增加log.index.size.max.bytes以避免频繁的日志滚动。
* 由于索引文件数量增加，对于一些有大量日志段的broker（即 >15k），在broker启动期间，日志加载处理可能更长。根据我们的实现，num.recovery.threads.per.data.dir设置为1可减少日志加载的时间。

### 0.10.1.0显著的变化

* 新的java消费者不再是测试阶段了，我们建议将其应用到所有的新开发当中。旧的Scala使用仍然支持，但将在下一个版本中弃用，并在未来的主要版本中移除。
* --new-consumer/--new.consumer转换不再需要使用MirrorMaker和类似于Console消费者工具。只需要通过一个Kafka broker连接，而不是ZooKeeper了。另外，控制台消费者和旧消费者已弃用，并且将在未来的主要版本中移除。
* Kafka集群现在可通过集群ID来标识唯一，broker升级到0.10.1.0时将自动的生成。集群ID可通过kafka.server:type=KafkaServer,name=ClusterId获取。它是元数据相应的一部分，序列化，客户端拦截器和度量记录器可通过实现ClusterResourceListener接口来接收集群ID。
* BrokerState "RunningAsController" (value 4) 已被移除。由于一个bug，brpker仅在转换出来之前处于这种状态，因此移除影响应该是最小的。推荐的方法是通过kafka检查给定的broker是否是控制器。controller:type=KafkaController,name=ActiveControllerCount
* 新的Java消费者现在允许用户通过分区上的时间戳来搜索offset。
* 新的Java消费者现在支持后台线程心跳检测，有一个新的配置max.poll.interval.ms控制消费者主动离开组之前poll调用之间的最大时间（默认是5分钟）。配置request.timeout.ms的值必须始终大于max.poll.interval.ms，因为JoinGroup请求在消费者重新平衡时候阻塞服务器的最大时间。因此我们更改了其默认值超过5分钟，最后，session.timeout.ms的默认值已调整为10秒，并max.poll.records的默认值更改为500。
* 当使用Authorizer并且用户对topic没有描述授权时，broker将不再向请求返回TOPIC\_AUTHORIZATION\_FAILED错误，因为这会泄漏topic名称。 相反，将返回UNKNOWN\_TOPIC\_OR\_PARTITION错误代码。 当使用生产者和消费者时，这可能导致意外的超时或延迟，因为Kafka客户端通常将在未知的topic错误时自动重试。 如果您怀疑这可能已经正在发生，你应该查阅客户端日志。
* 获取响应的默认的限制大小（消费者为50MB，副本为10MB）。现有的分区限制也适用（消费者和副本是1MB）。注意，这些限制不是绝对的最大值（下一节解释）。
* 如果一个消息大于响应/分区大小限制，消费者和副本可以继续使用。更具体的是，如果在第一个非空分区中的第一个消息大于限制，则消息将仍然返回。
* kafka.api.FetchRequest和kafka.javaapi.FetchRequest中增加了重载的构造函数。以允许调用者去指定分区的顺序（因为在v3中顺序很重要）。之前的构造函数已弃用。在请求发送之前，以避免资源匮乏问题引起的混洗。

#### 新协议版本

* ListOffsetRequest v1支持基于时间戳的精确offset搜索。
* MetadataResponse v2引入了一个新字段：“cluster\_id”。
* FetchRequest v3支持限制响应大小（除了现有的分区限制）。
* JoinGroup v1引入了一个新字段：“rebalance\_timeout”。

## 从0.8.x 或 0.9.x 升级到 0.10.0.0

0.10.0.0具有潜在的突变更改（请在升级之前查看），以及升级后可能的性能影响。 通过遵循以下建议的滚动升级计划，可保障在升级期间和之后不会出现停机时间和性能影响。  
注意：由于引入了新协议，因此在升级客户端之前先升级Kafka集群。

注意，对于版本0.9.0.0：由于0.9.0.0中有一个bug，依赖于Zookeeper（旧的Scala高级消费者和MirrorMaker如果一起使用）的客户端将无法在0.10.0.x中使用。因此，broker升级到0.10.0.x之前，先升级0.9.0.0客户端到0.9.0.1。对于0.8.X或0.9.0.1客户端，此步骤不是必需的。

##### 滚动升级:

1. 更新所有broker的server.properties文件，并添加以下配置：
   * inter.broker.protocol.version=CURRENT\_KAFKA\_VERSION (例如：0.8.2 或 0.9.0.0).
   * log.message.format.version=CURRENT\_KAFKA\_VERSION (有关此配置的详细信息，请查看升级后潜在的性能影响。)
2. 升级broker，关闭它，然后升级到新版本，最后重启它。
3. 一旦整个集群升级完成，通过编辑inter.broker.protocol.version设置为0.10.0.0转换所有协议。注意：你现在应该还不需要设置message.format.version - 此配置应该当所有的消费者升级为0.10.0.0时才需要设置。
4. 依次重新启动broker，使新协议版本生效。
5. 一旦所有的消费者已经升级为.10.0，设置每个broker的log.message.format.version为0.10.0，然后逐个重启。

**注意** ：如果你接受停机目，你可以简单粗暴的关闭所有broker，更新版本并重新启动。它们默认从新协议开始。

**注意** ：变换协议版本和重启启动可以在broker升级完成后的任何时间去做，不必马上做。

#### 升级到0.10.0.0后潜在的性能影响

0.10.0中的消息格式包括新的时间戳字段，并使用压缩消息的相关联的offset。磁盘默认的消息格式是0.10.0，消息格式可以通过server.properties中的log.message.format.version配置。如果消费者客户端版本低于0.10.0.0。它只能“理解”0.10.0之前的消息格式。在这种情况下，broker在发送响应到旧版本消费者之前转换0.10.0格式到之前的格式。然而，这样的话，broker不是零复制传输。在Kafka社区关于性能影响的报告显示，在升级后，CPU利用率从20%提高100%。这迫使所有客户端马升级，促使性能恢复正常。为了避免消费者升级到0.10.0.0之前的消息转换，可以设置log.message.format.version为0.8.2或0.9.0。这样，broker仍然零复制传输将数据发送给旧的消费者。一旦消费者升级，就可以把消息格式更为0.10.0，就可以享受含新时间戳和优化后的压缩新消息格式。转换只是为了确保兼容性，尽可能避免消息转换才是至关重要的。

客户端升级到0.10.0.0，不会对性能产生影响。

**注意** ：通过设置消息格式版本，可以证明所有现有消息处于或低于该消息格式版本。否则消费者在0.10.0.0之前可能会中断。特别是，在消息格式设置为0.10.0之后，不应将其更改回较早的格式，因为它可能会在0.10.0.0之前的版本上中断消费者。

**注意** ：由于在每个消息中引入了额外的时间戳，生产者在发送少量消息可能会看到消息吞吐量下降（因为增加了开销）。 同样，复制每个消息传输也增加了8个字节。 如果你集群的能力与网络接近，可能会超过网卡，并看到由于过载的故障和性能问题。

**注意** ：如果生产者已经启用了压缩，则在某些情况下，可能注意到生产者吞吐量减少或broker的压缩率降低。当接收压缩消息时，0.10.0的broker避免再次压缩消息，这样减少延迟并提高吞吐量。然而，在某些情况下，这可能减少生产者的批次大小，导致较差的吞吐量。如果出现这种情况，可调整生产者的linger.ms 和 batch.size以提高吞吐量。另外，生产者用于压缩消息的缓存小于broker生产者使用的缓存，这可能对磁盘上的消息的压缩比有负面影响。 我们打算在未来的Kafka版本中进行配置。

0.10.0.0潜在的中断

* 从Kafka 0.10.0.0开始，Kafka中的消息格式版本表示为Kafka的版本。例如，消息格式0.9.0指的是支持的最高消息版本就是0.9.0。
* 消息格式0.10.0已经介绍过了，并且默认是使用的。消息包含了一个时间戳字段和压缩后消息的关系offset。
* 已经引入了ProduceRequest/Response v2，并默认使用支持消息格式0.10.0。
* 已经引入了FetchRequest/Response v2已经被引入，它默认使用支持消息格式0.10.0。
* MessageFormatter 接口从def writeTo(key: Array[Byte], value: Array[Byte], output: PrintStream) 更改为 def writeTo(consumerRecord: ConsumerRecord[Array[Byte], Array[Byte]], output: PrintStream)
* MessageReader 接口从 def readMessage(): KeyedMessage[Array[Byte], Array[Byte]] 更改为 def readMessage(): ProducerRecord[Array[Byte], Array[Byte]]
* MessageFormatter的包从kafka.tools到kafka.common
* MessageReader的包从kafka.tools到kafka.common
* MirrorMakerMessageHandler不再处理（记录：MessageAndMetadata [Array [Byte]，Array [Byte]]）方法从未被调用用。
* 0.7版本的KafkaMigrationTool不再和kafka一起打包。如果你需要从0.7迁移到0.10.0，请先迁移到0.8，然后按照的升级步骤从0.8升级到0.10.0。
* 新消费者API已标准化，接收java.util.Collection作为方法参数的序列化类型。升级现有的版本才能使用0.10.0客户端库
* LZ4压缩消息处理已更改为使用可互操作的规范框架(LZ4f v1.5.1)。为了保留与旧客户端的兼容性，此改变仅适用于消息格式为0.10.0和更高版本。使用v0/v1（消息格式0.9.0）Produce/Fetch LZ4压缩消息的客户端应继续使用0.9.0实现框架。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框架。可互操作的LZ4库的列表可在<http://www.lz4.org/查看>

在0.10.0.0的显著变化

* 从0.10.0.0开始，增加一个新的客户端Kafka Streams客户端，用于流式处理存储在kafka topic的数据。这个新客户端仅支持0.10.x或更高的版本。
* 新消费者默认receive.buffer.bytes是64K。
* 新的消费者现在公开了exclude.internal.topics配置，以防止内部topic（例如消费者offset topic）被其他的正则匹配订阅。默认是启用。
* 旧的的Scala的生产者已经弃用。使用者尽快使用最新的Java客户端。

新的消费者API已标记为稳定。

## 从0.8.0, 0.8.1.X或0.8.2.X升级到0.9.0.0

9.0.0有潜在的中断更改风险（在升级之前需要知道），并且与之前版本的broker之间的协议改变。这意味着此次升级可能和客户端旧版本不兼容。因此在升级客户端之前，先升级kafka集群。如果你使用MirrorMaker下游集群，则同样应首先升级。

##### 滚动升级

1. 升级所有broker的server.properties,并在其中添加inter.broker.protocol.version = 0.8.2.X
2. 每次升级一个broker：关闭broker，替换新版本，然后重新启动。
3. 一旦整个群集升级，通过编辑inter.broker.protocol.version并将其设置为0.9.0.0来转换所有协议。
4. 逐个重新启动broker，使新协议版本生效。

**注意** ：如果你可接受停机，你可以简单地将所有broker关闭，更新版本并重启启动，协议将默认从新版本开始。

**注意** ：变换协议版本和重启启动可以在broker升级完成后的任何时间去做，不必马上做。

### 0.9.0.0潜在的中断变化

* Java 1.6不再支持。
* Scala 2.9不再支持。
* 默认情况下，1000以上的Broker ID为自动分配。如果你的集群高于该阈值，需相应地增加reserved.broker.max.id配置。
* replica.lag.max.messages配置已经移除。分区leader在决定哪些副本处于同步时将不再考虑落后的消息的数。
* 配置参数replica.lag.time.max.ms现在不仅指自上次从副本获取请求后经过的时间，还指自副本上次被捕获以来的时间。 副本仍然从leader获取消息，但超过replica.lag.time.max.ms配置的最新消息将被认为不同步的。
* 压缩的topic不再接受没有key的消息，如果出现，生产者将抛出异常。 在0.8.x中，没有key的消息将导致日志压缩线程退出（并停止所有压缩的topic）。
* MirrorMaker不再支持多个目标集群。 它只接受一个--consumer.config。 要镜像多个源集群，每个源集群至少需要一个MirrorMaker实例，每个源集群都有自己的消费者配置。
* 在org.apache.kafka.clients.tools。包下的Tools已移至org.apache.kafka.tools。。 所有包含的脚本仍将照常工作，只有直接导入这些类的自定义代码将受到影响。
* 在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，失败时出现非零退出代码。
* kafka-topics.sh脚本（kafka.admin.TopicCommand）现在将在topic名称由于使用“.”或“\_”而导致风险度量标准冲突时打印警告。以及冲突的情况下的错误。
* kafka-console-producer.sh脚本（kafka.tools.ConsoleProducer）将默认使用新的Java Producer，用户必须指定“old-producer”才能使用旧生产者。
* 默认情况下，所有命令行工具都会将所有日志消息打印到stderr而不是stdout。

### 0.9.0.1中的显著变化

* 可以通过将broker.id.generation.enable设置为false来禁用新的broker ID生成功能。
* 默认情况下，配置参数log.cleaner.enable为true。 这意味着topic会清理。
* policy = compact现在将被默认压缩，并且128MB的堆（通过log.cleaner.dedupe.buffer.size）分配给清洗进程。你可能需要根据你对压缩topic的使用情况，查看log.cleaner.dedupe.buffer.size和其他log.cleaner配置值。
* 默认情况下，新消费者的配置参数fetch.min.bytes的默认值为1。

### 0.9.0.0弃用的

* kafka-topics.sh脚本的变更topic配置已弃用（kafka.admin.ConfigCommand），以后将使用kafka-configs.sh(kafka.admin.ConfigCommand) 。
* kafka-consumer-offset-checker.sh(kafka.tools.ConsumerOffsetChecker)已弃用，以后将使用kafka-consumer-groups.sh (kafka.admin.ConsumerGroupCommand)
* kafka.tools.ProducerPerformance已弃用。以后将使用org.apache.kafka.tools.ProducerPerformance（kafka-producer-perf-test.sh也将使用新类）
* 生产者的block.on.buffer.full已弃用，并将在以后的版本中移除。目前其默认已经更为false。KafkaProducer将不再抛出BufferExhaustedException，而是使用max.block.ms来中止，之后将抛出TimeoutException。如果block.on.buffer.full属性明确地设置为true，它将设置max.block.ms为Long.MAX\_VALUE和metadata.fetch.timeout.ms将不执行。

## 从0.8.1升级到0.8.2

0.8.2与0.8.1完全兼容。 关闭，更新代码并重新启动，逐个升级broker。

## 从0.8.0升级到0.8.1

0.8.1与0.8完全兼容。 关闭，更新代码并重新启动，逐个升级broker。

## 从0.7升级

版本0.7与较新版本不兼容。 对API，ZooKeeper数据结构，协议和配置进行了主要更改，以便添加复制（在0.7中缺失）。 从0.7版升级到更高版本需要一个特殊的迁移工具（通过下一章的API）。 此迁移可以在不停机的情况下完成。

**kafka接口API**

Apache Kafka引入一个新的java客户端（在org.apache.kafka.clients 包中），替代老的Scala客户端，但是为了兼容，将会共存一段时间。为了减少依赖，这些客户端都有一个独立的jar，而旧的Scala客户端继续与服务端保留在同个包下。

## Kafka有4个核心API：

* [Producer API](http://orchome.com/190) 允许应用程序发送数据流到kafka集群中的topic。
* [Consumer API](http://orchome.com/200) 允许应用程序从kafka集群的topic中读取数据流。
* [Streams API](http://orchome.com/304) 允许从输入topic转换数据流到输出topic。
* [Connect API](http://orchome.com/455) 通过实现连接器（connector），不断地从一些源系统或应用程序中拉取数据到kafka，或从kafka提交数据到宿系统（sink system）或应用程序。

kafka公开了其所有的功能协议，与语言无关。只有java客户端作为kafka项目的一部分进行维护，其他的作为开源的项目提供，这里提供了非java客户端的列表。  
<https://cwiki.apache.org/confluence/display/KAFKA/Clients>

**生产者API**

我们鼓励所有新开发的程序使用新的Java生产者，新的java生产者客户端比以前的Scala的客户端更快、功能更全面。通过下面的例子，引入Maven（可以更改新的版本号）。

<dependency>

<groupId>org.apache.kafka</groupId>

<artifactId>kafka-clients</artifactId>

<version>0.10.1.0</version>

</dependency>

如何使用生产者，请点击[这里](/303)。  
对以前传统Scala生产者API感兴趣的，可以在点击[这里](/8)。

**kafka生产者客户端（0.10.1.1API）**

### kafka客户端发布record(消息)到kafka集群。

新的生产者是线程安全的，在线程之间共享**单个生产者**实例，通常单例比多个实例要快。

一个简单的例子，使用producer发送一个有序的key/value(键值对)，放到java的main方法里就能直接运行，

Properties props = new Properties();

props.put("bootstrap.servers", "localhost:9092");

props.put("acks", "all");

props.put("retries", 0);

props.put("batch.size", 16384);

props.put("linger.ms", 1);

props.put("buffer.memory", 33554432);

props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");

props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

Producer<String, String> producer = new KafkaProducer<>(props);

for(int i = 0; i < 100; i++)

producer.send(new ProducerRecord<String, String>("my-topic", Integer.toString(i), Integer.toString(i)));

producer.close();

生产者的缓冲空间池保留尚未发送到服务器的消息，后台I/O线程负责将这些消息转换成请求发送到集群。如果使用后不关闭生产者，则会泄露这些资源。

send()方法是异步的，添加消息到缓冲区等待发送，并立即返回。生产者将单个的消息批量在一起发送来提高效率。

ack是判别请求是否为完整的条件（就是是判断是不是成功发送了）。我们指定了“all”将会阻塞消息，这种设置性能最低，但是是最可靠的。

retries，如果请求失败，生产者会自动重试，我们指定是0次，如果启用重试，则会有重复消息的可能性。

producer(生产者)缓存每个分区未发送消息。缓存的大小是通过 batch.size 配置指定的。值较大的话将会产生更大的批。并需要更多的内存（因为每个“活跃”的分区都有1个缓冲区）。

默认缓冲可立即发送，即遍缓冲空间还没有满，但是，如果你想减少请求的数量，可以设置linger.ms大于0。这将指示生产者发送请求之前等待一段时间，希望更多的消息填补到未满的批中。这类似于TCP的算法，例如上面的代码段，可能100条消息在一个请求发送，因为我们设置了linger(逗留)时间为1毫秒，然后，如果我们没有填满缓冲区，这个设置将增加1毫秒的延迟请求以等待更多的消息。需要注意的是，在高负载下，相近的时间一般也会组成批，即使是 linger.ms=0。在不处于高负载的情况下，如果设置比0大，以少量的延迟代价换取更少的，更有效的请求。

buffer.memory 控制生产者可用的缓存总量，如果消息发送速度比其传输到服务器的快，将会耗尽这个缓存空间。当缓存空间耗尽，其他发送调用将被阻塞，阻塞时间的阈值通过max.block.ms设定，之后它将抛出一个TimeoutException。

key.serializer和value.serializer示例，将用户提供的key和value对象ProducerRecord转换成字节，你可以使用附带的**ByteArraySerializaer**或**StringSerializer**处理简单的string或byte类型。

## send()

public Future<RecordMetadata> send(ProducerRecord<K,V> record,Callback callback)

异步发送一条消息到topic，并调用callback（当发送已确认）。

send是异步的，并且一旦消息被保存在等待发送的消息缓存中，此方法就立即返回。这样并行发送多条消息而不阻塞去等待每一条消息的响应。

发送的结果是一个[RecordMetadata](http://kafka.apache.org/0101/javadoc/org/apache/kafka/clients/producer/RecordMetadata.html)，它指定了消息发送的分区，分配的offset和消息的时间戳。如果topic使用的是CreateTime，则使用用户提供的时间戳或发送的时间（如果用户没有指定指定消息的时间戳）如果topic使用的是LogAppendTime，则追加消息时，时间戳是broker的本地时间。

由于send调用是异步的，它将为分配消息的此消息的RecordMetadata返回一个[Future](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html?is-external=true)。如果future调用[get()](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html?is-external=true#get)，则将阻塞，直到相关请求完成并返回该消息的metadata，或抛出发送异常。

如果要模拟一个简单的阻塞调用，你可以调用get()方法。

byte[] key = "key".getBytes();

byte[] value = "value".getBytes();

ProducerRecord<byte[],byte[]> record = new ProducerRecord<byte[],byte[]>("my-topic", key, value)

producer.send(record).get();

完全无阻塞的话,可以利用回调参数提供的请求完成时将调用的回调通知。

ProducerRecord<byte[],byte[]> record = new ProducerRecord<byte[],byte[]>("the-topic", key, value);

producer.send(myRecord,

new Callback() {

public void onCompletion(RecordMetadata metadata, Exception e) {

if(e != null)

e.printStackTrace();

System.out.println("The offset of the record we just sent is: " + metadata.offset());

}

});

发送到同一个分区的消息回调保证按一定的顺序执行，也就是说，在下面的例子中 callback1 保证执行 callback2 之前：

producer.send(new ProducerRecord<byte[],byte[]>(topic, partition, key1, value1), callback1);

producer.send(new ProducerRecord<byte[],byte[]>(topic, partition, key2, value2), callback2);

注意：callback一般在生产者的I/O线程中执行，所以是相当的快的，否则将延迟其他的线程的消息发送。如果你需要执行阻塞或计算昂贵（消耗）的回调，建议在callback主体中使用自己的[Executor](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executor.html?is-external=true)来并行处理。

##### pecified by:

send in interface Producer<K,V>

##### Parameters:

record - 发送的记录（消息）  
callback - 用户提供的callback，服务器来调用这个callback来应答结果（null表示没有callback）。

##### Throws:

InterruptException - 如果线程在阻塞中断。  
SerializationException - 如果key或value不是给定有效配置的serializers。  
TimeoutException - 如果获取元数据或消息分配内存话费的时间超过max.block.ms。  
KafkaException - Kafka有关的错误（不属于公共API的异常）。

**kafka消费者API**

随着0.9.0版本，我们已经增加了一个新的Java消费者替换我们现有的基于zookeeper的高级和低级消费者。这个客户端还是测试版的质量。为了确保用户平滑升级，我们仍然维护旧的0.8版本的消费者客户端继续在0.9集群上工作，两个老的0.8 API的消费者（ [高级消费者](http://orchome.com/10) 和 [低级消费者](http://orchome.com/11)）。

这个新的消费API，清除了0.8版本的高版本和低版本消费者之间的区别，你可以通过下面的maven，引入依赖到你的客户端。

<dependency>

<groupId>org.apache.kafka</groupId>

<artifactId>kafka-clients</artifactId>

<version>0.10.1.0</version>

</dependency>

如何使用新的消费者，请点击[这里](http://orchome.com/451)。

**kafka消费者客户端（0.10.0.1API）**

public class KafkaConsumer<K,V>

extends Object

implements Consumer<K,V>

Kafka客户端从集群中消费消息，并透明地处理kafka集群中的故障服务器，透明地调节适应集群中变化的数据分区。也和服务器交互，平衡消费者。

消费者TCP长连接到broker来拉取消息。故障导致的消费者关闭失败，将会泄露这些连接，消费者不是线程安全的，可以查看更多关于[Multi-threaded（多线程）](http://kafka.apache.org/0100/javadoc/org/apache/kafka/clients/consumer/KafkaConsumer.html#multithreaded)处理的细节。

#### 偏移量和消费者的位置

kafka为分区中的每条消息保存一个偏移量（offset），这个偏移量是该分区中一条消息的唯一标示符。也表示消费者在分区的位置。例如，一个位置是5的消费者(说明已经消费了0到4的消息)，下一个接收消息的偏移量为5的消息。实际上有两个与消费者相关的“位置”概念：

消费者的位置给出了下一条记录的偏移量。它比消费者在该分区中看到的最大偏移量要大一个。 它在每次消费者在调用poll(long)中接收消息时自动增长。

“已提交”的位置是已安全保存的最后偏移量，如果进程失败或重新启动时，消费者将恢复到这个偏移量。消费者可以选择定期自动提交偏移量，也可以选择通过调用commit API来手动的控制(如：commitSync 和 commitAsync)。

这个区别是消费者来控制一条消息什么时候才被认为是已被消费的，控制权在消费者，下面我们进一步更详细地讨论。

#### 消费者组和主题订阅

Kafka的消费者组概念，通过进程池瓜分消费和处理消息的工作。这些进程可以在同一台机器运行，也可分布到多台机器上，增加可扩展性和容错性,相同group.id的消费者将视为同一个消费者组。

分组中的每个消费者通过subscribe API动态的订阅一个topic列表。kafka将已订阅topic的消息发送到每个消费者组中。并通过平衡分区在消费者分组中所有成员之间来达到平均。因此每个分区恰好地分配1个消费者（一个消费者组中）。所有如果一个topic有4个分区，并且一个消费者分组有2个消费者。那么每个消费者消费2个分区。

消费者组的成员是动态维护的：如果一个消费者故障。分配给它的分区将重新分配给同一个分组中其他的消费者。同样的，如果一个新的消费者加入到分组，将从现有消费者中移一个给它。这被称为重新平衡分组，并在下面更详细地讨论。 当新分区添加到订阅的topic时，或者当创建与订阅的正则表达式匹配的新topic时，也将重新平衡。将通过定时刷新自动发现新的分区，并将其分配给分组的成员。

从概念上讲，你可以将消费者分组看作是由多个进程组成的单一逻辑订阅者。作为一个多订阅系统，Kafka支持对于给定topic任何数量的消费者组，而不重复。

这是在消息系统中常见的功能的略微概括。所有进程都将是单个消费者分组的一部分（类似传统消息传递系统中的队列的语义），因此消息传递就像队列一样，在组中平衡。与传统的消息系统不同的是，虽然，你可以有多个这样的组。但每个进程都有自己的消费者组（类似于传统消息系统中pub-sub的语义），因此每个进程都会订阅到该主题的所有消息。

此外，当分组重新分配自动发生时，可以通过ConsumerRebalanceListener通知消费者，这允许他们完成必要的应用程序级逻辑，例如状态清除，手动偏移提交等。有关更多详细信息，请参阅[Kafka存储的偏移](http://kafka.apache.org/0101/javadoc/org/apache/kafka/clients/consumer/KafkaConsumer.html#rebalancecallback)。

它也允许消费者通过使用assign(Collection)手动分配指定分区，如果使用手动指定分配分区，那么动态分区分配和协调消费者组将失效。

#### 发现消费者故障

订阅一组topic后，当调用poll(long）时，消费者将自动加入到组中。只要持续的调用poll，消费者将一直保持可用，并继续从分配的分区中接收消息。此外，消费者向服务器定时发送心跳。 如果消费者崩溃或无法在session.timeout.ms配置的时间内发送心跳，则消费者将被视为死亡，并且其分区将被重新分配。

还有一种可能，消费可能遇到“活锁”的情况，它持续的发送心跳，但是没有处理。为了预防消费者在这种情况下一直持有分区，我们使用max.poll.interval.ms活跃检测机制。 在此基础上，如果你调用的poll的频率大于最大间隔，则客户端将主动地离开组，以便其他消费者接管该分区。 发生这种情况时，你会看到offset提交失败（调用commitSync（）引发的CommitFailedException）。这是一种安全机制，保障只有活动成员能够提交offset。所以要留在组中，你必须持续调用poll。

消费者提供两个配置设置来控制poll循环：

1. max.poll.interval.ms：增大poll的间隔，可以为消费者提供更多的时间去处理返回的消息（调用poll(long)返回的消息，通常返回的消息都是一批）。缺点是此值越大将会延迟组重新平衡。
2. max.poll.records：此设置限制每次调用poll返回的消息数，这样可以更容易的预测每次poll间隔要处理的最大值。通过调整此值，可以减少poll间隔，减少重新平衡分组的

对于消息处理时间不可预测地的情况，这些选项是不够的。 处理这种情况的推荐方法是将消息处理移到另一个线程中，让消费者继续调用poll。 但是必须注意确保已提交的offset不超过实际位置。另外，你必须禁用自动提交，并只有在线程完成处理后才为记录手动提交偏移量（取决于你）。 还要注意，你需要pause暂停分区，不会从poll接收到新消息，让线程处理完之前返回的消息（如果你的处理能力比拉取消息的慢，那创建新线程将导致你机器内存溢出）。

## 示例

这个消费者API提供了灵活性，以涵盖各种消费场景，下面是一些例子来演示如何使用它们。

#### 自动提交偏移量

这是个【自动提交偏移量】的简单的kafka消费者API。

Properties props = new Properties();

props.put("bootstrap.servers", "localhost:9092");

props.put("group.id", "test");

props.put("enable.auto.commit", "true");

props.put("auto.commit.interval.ms", "1000");

props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);

consumer.subscribe(Arrays.asList("foo", "bar"));

while (true) {

ConsumerRecords<String, String> records = consumer.poll(100);

for (ConsumerRecord<String, String> record : records)

System.out.printf("offset = %d, key = %s, value = %s%n", record.offset(), record.key(), record.value());

}

设置enable.auto.commit,偏移量由auto.commit.interval.ms控制自动提交的频率。

集群是通过配置bootstrap.servers指定一个或多个broker。不用指定全部的broker，它将自动发现集群中的其余的borker（最好指定多个，万一有服务器故障）。

在这个例子中，客户端订阅了主题foo和bar。消费者组叫test。

broker通过心跳机器自动检测test组中失败的进程，消费者会自动ping集群，告诉进群它还活着。只要消费者能够做到这一点，它就被认为是活着的，并保留分配给它分区的权利，如果它停止心跳的时间超过session.timeout.ms,那么就会认为是故障的，它的分区将被分配到别的进程。

这个deserializer设置如何把byte转成object类型，例子中，通过指定string解析器，我们告诉获取到的消息的key和value只是简单个string类型。

#### 手动控制偏移量

不需要定时的提交offset，可以自己控制offset，当消息认为已消费过了，这个时候再去提交它们的偏移量。这个很有用的，当消费的消息结合了一些处理逻辑，这个消息就不应该认为是已经消费的，直到它完成了整个处理。

Properties props = new Properties();

props.put("bootstrap.servers", "localhost:9092");

props.put("group.id", "test");

props.put("enable.auto.commit", "false");

props.put("auto.commit.interval.ms", "1000");

props.put("session.timeout.ms", "30000");

props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);

consumer.subscribe(Arrays.asList("foo", "bar"));

final int minBatchSize = 200;

List<ConsumerRecord<String, String>> buffer = new ArrayList<>();

while (true) {

ConsumerRecords<String, String> records = consumer.poll(100);

for (ConsumerRecord<String, String> record : records) {

buffer.add(record);

}

if (buffer.size() >= minBatchSize) {

insertIntoDb(buffer);

consumer.commitSync();

buffer.clear();

}

}

在这个例子中，我们将消费一批消息并将它们存储在内存中。当我们积累足够多的消息后，我们再将它们批量插入到数据库中。如果我们设置offset自动提交（之前说的例子），消费将被认为是已消费的。这样会出现问题，我们的进程可能在批处理记录之后，但在它们被插入到数据库之前失败了。

为了避免这种情况，我们将在相应的记录插入数据库之后再手动提交偏移量。这样我们可以准确控制消息是成功消费的。提出一个相反的可能性：在插入数据库之后，但是在提交之前，这个过程可能会失败（即使这可能只是几毫秒，这是一种可能性）。在这种情况下，进程将获取到已提交的偏移量，并会重复插入的最后一批数据。这种方式就是所谓的“至少一次”保证，在故障情况下，可以重复。

如果您无法执行这些操作，可能会使已提交的偏移超过消耗的位置，从而导致缺少记录。 使用手动偏移控制的优点是，您可以直接控制记录何时被视为“已消耗”。

注意：使用自动提交也可以“至少一次”。但是要求你必须下次调用poll（long）之前或关闭消费者之前，处理完所有返回的数据。如果操作失败，这将会导致已提交的offset超过消费的位置，从而导致丢失消息。使用手动控制offset的有点是，你可以直接控制消息何时提交。、

上面的例子使用commitSync表示所有收到的消息为”已提交"，在某些情况下，你可以希望更精细的控制，通过指定一个明确消息的偏移量为“已提交”。在下面，我们的例子中，我们处理完每个分区中的消息后，提交偏移量。

try {

while(running) {

ConsumerRecords<String, String> records = consumer.poll(Long.MAX\_VALUE);

for (TopicPartition partition : records.partitions()) {

List<ConsumerRecord<String, String>> partitionRecords = records.records(partition);

for (ConsumerRecord<String, String> record : partitionRecords) {

System.out.println(record.offset() + ": " + record.value());

}

long lastOffset = partitionRecords.get(partitionRecords.size() - 1).offset();

consumer.commitSync(Collections.singletonMap(partition, new OffsetAndMetadata(lastOffset + 1)));

}

}

} finally {

consumer.close();

}

注意：已提交的offset应始终是你的程序将读取的下一条消息的offset。因此，调用commitSync（offsets）时，你应该加1个到最后处理的消息的offset。

#### 订阅指定的分区

在前面的例子中，我们订阅我们感兴趣的topic，让kafka提供给我们平分后的topic分区。但是，在有些情况下，你可能需要自己来控制分配指定分区，例如：

* 如果这个消费者进程与该分区保存了某种本地状态（如本地磁盘的键值存储），则它应该只能获取这个分区的消息。
* 如果消费者进程本身具有高可用性，并且如果它失败，会自动重新启动（可能使用集群管理框架如YARN，Mesos，或者AWS设施，或作为一个流处理框架的一部分）。 在这种情况下，不需要Kafka检测故障，重新分配分区，因为消费者进程将在另一台机器上重新启动。

要使用此模式，，你只需调用assign（Collection）消费指定的分区即可：

String topic = "foo";

TopicPartition partition0 = new TopicPartition(topic, 0);

TopicPartition partition1 = new TopicPartition(topic, 1);

consumer.assign(Arrays.asList(partition0, partition1));

一旦手动分配分区，你可以在循环中调用poll（跟前面的例子一样）。消费者分组仍需要提交offset，只是现在分区的设置只能通过调用assign修改，因为手动分配不会进行分组协调，因此消费者故障不会引发分区重新平衡。每一个消费者是独立工作的（即使和其他的消费者共享GroupId）。为了避免offset提交冲突，通常你需要确认每一个consumer实例的gorupId都是唯一的。

注意，手动分配分区（即，assgin）和动态分区分配的订阅topic模式（即，subcribe）不能混合使用。

#### offset存储在其他地方

消费者可以不使用kafka内置的offset仓库。可以选择自己来存储offset。要注意的是，将消费的offset和结果存储在同一个的系统中，用原子的方式存储结果和offset，但这不能保证原子，要想消费是完全原子的，并提供的“正好一次”的消费保证比kafka默认的“至少一次”的语义要更高。你需要使用kafka的offset提交功能。

这有结合的例子。

* 如果消费的结果存储在关系数据库中，存储在数据库的offset，让提交结果和offset在单个事务中。这样，事物成功，则offset存储和更新。如果offset没有存储，那么偏移量也不会被更新。
* 如果offset和消费结果存储在本地仓库。例如，可以通过订阅一个指定的分区并将offset和索引数据一起存储来构建一个搜索索引。如果这是以原子的方式做的，常见的可能是，即使崩溃引起未同步的数据丢失。索引程序从它确保没有更新丢失的地方恢复，而仅仅丢失最近更新的消息。

每个消息都有自己的offset，所以要管理自己的偏移，你只需要做到以下几点：

* 配置 enable.auto.commit=false
* 使用提供的 ConsumerRecord 来保存你的位置。
* 在重启时用 seek(TopicPartition, long) 恢复消费者的位置。

当分区分配也是手动完成的（像上文搜索索引的情况），这种类型的使用是最简单的。 如果分区分配是自动完成的，需要特别小心处理分区分配变更的情况。可以通过调用subscribe（Collection，ConsumerRebalanceListener）和subscribe（Pattern，ConsumerRebalanceListener）中提供的ConsumerRebalanceListener实例来完成的。例如，当分区向消费者获取时，消费者将通过实现ConsumerRebalanceListener.onPartitionsRevoked（Collection）来给这些分区提交它们offset。当分区分配给消费者时，消费者通过ConsumerRebalanceListener.onPartitionsAssigned(Collection)为新的分区正确地将消费者初始化到该位置。

ConsumerRebalanceListener的另一个常见用法是清除应用已移动到其他位置的分区的缓存。

#### 控制消费的位置

大多数情况下，消费者只是简单的从头到尾的消费消息，周期性的提交位置（自动或手动）。kafka也支持消费者去手动的控制消费的位置，可以消费之前的消息也可以跳过最近的消息。

有几种情况，手动控制消费者的位置可能是有用的。

一种场景是对于时间敏感的消费者处理程序，对足够落后的消费者，直接跳过，从最近的消费开始消费。

另一个使用场景是本地状态存储系统（上一节说的）。在这样的系统中，消费者将要在启动时初始化它的位置（无论本地存储是否包含）。同样，如果本地状态已被破坏（假设因为磁盘丢失），则可以通过重新消费所有数据并重新创建状态（假设kafka保留了足够的历史）在新的机器上重新创建。

kafka使用seek(TopicPartition, long)指定新的消费位置。用于查找服务器保留的最早和最新的offset的特殊的方法也可用（seekToBeginning(Collection) 和 seekToEnd(Collection)）。

#### 消费者流量控制

如果消费者分配了多个分区，并同时消费所有的分区，这些分区具有相同的优先级。在一些情况下，消费者需要首先消费一些指定的分区，当指定的分区有少量或者已经没有可消费的数据时，则开始消费其他分区。

例如流处理，当处理器从2个topic获取消息并把这两个topic的消息合并，当其中一个topic长时间落后另一个，则暂停消费，以便落后的赶上来。

kafka支持动态控制消费流量，分别在future的poll(long)中使用pause(Collection) 和 resume(Collection) 来暂停消费指定分配的分区，重新开始消费指定暂停的分区。

#### 多线程处理

Kafka消费者不是线程安全的。所有网络I/O都发生在进行调用应用程序的线程中。用户的责任是确保多线程访问正确同步的。非同步访问将导致ConcurrentModificationException。

此规则唯一的例外是wakeup()，它可以安全地从外部线程来中断活动操作。在这种情况下，将从操作的线程阻塞并抛出一个WakeupException。这可用于从其他线程来关闭消费者。 以下代码段显示了典型模式：

public class KafkaConsumerRunner implements Runnable {

private final AtomicBoolean closed = new AtomicBoolean(false);

private final KafkaConsumer consumer;

public void run() {

try {

consumer.subscribe(Arrays.asList("topic"));

while (!closed.get()) {

ConsumerRecords records = consumer.poll(10000);

// Handle new records

}

} catch (WakeupException e) {

// Ignore exception if closing

if (!closed.get()) throw e;

} finally {

consumer.close();

}

}

// Shutdown hook which can be called from a separate thread

public void shutdown() {

closed.set(true);

consumer.wakeup();

}

}

在单独的线程中，可以通过设置关闭标志和唤醒消费者来关闭消费者。

closed.set(true);

consumer.wakeup();

我们没有多线程模型的例子。但留下几个操作可用来实现多线程处理消息。

1. 每个线程一个消费者

每个线程自己的消费者实例。这里是这种方法的优点和缺点：

* + PRO: 这是最容易实现的
  + PRO: 因为它不需要在线程之间协调，所以通常它是最快的。
  + PRO: 它按顺序处理每个分区（每个线程只处理它接受的消息）。
  + CON: 更多的消费者意味着更多的TCP连接到集群（每个线程一个）。一般kafka处理连接非常的快，所以这是一个小成本。
  + CON: 更多的消费者意味着更多的请求被发送到服务器，但稍微较少的数据批次可能导致I/O吞吐量的一些下降。
  + CON: 所有进程中的线程总数受到分区总数的限制。

1. 解耦消费和处理

另一个替代方式是一个或多个消费者线程，它来消费所有数据，其消费所有数据并将ConsumerRecords实例切换到由实际处理记录处理的处理器线程池来消费的阻塞队列。这个选项同样有利弊：

* + PRO: 可扩展消费者和处理进程的数量。这样单个消费者的数据可分给多个处理器线程来执行，避免对分区的任何限制。
  + CON: 跨多个处理器的顺序保证需要特别注意，因为线程是独立的执行，后来的消息可能比遭到的消息先处理，这仅仅是因为线程执行的运气。如果对排序没有问题，这就不是个问题。
  + CON: 手动提交变得更困难，因为它需要协调所有的线程以确保处理对该分区的处理完成。

这种方法有多种玩法，例如，每个处理线程可以有自己的队列，消费者线程可以使用TopicPartitionhash到这些队列中，以确保按顺序消费，并且提交也将简化。

**Kafka Streams API**

### 2.3 Streams API

在0.10.0增加了一个新的客户端库，Kafka Stream，Kafka Stream具有Alpha的优点，你可以使用maven引入到你的项目：

<dependency>

<groupId>org.apache.kafka</groupId>

<artifactId>kafka-streams</artifactId>

<version>0.10.0.0</version>

</dependency>

如何使用，请点击[这里](http://orchome.com/305)。（注意，**@InterfaceStability.Unstable**注解的类，是公共API，在未来可能改变，不保证向后兼容）

**KafkaStreams客户端（0.10.1.1 API）**

Kafka Streams从一个或多个输入topic进行连续的计算并输出到0或多个外部topic中。

可以通过TopologyBuilder类定义一个计算逻辑处理器DAG拓扑。或者也可以通过提供的高级别KStream DSL来定义转换的KStreamBuilder。（PS：计算逻辑其实就是自己的代码逻辑）

KafkaStreams类管理Kafka Streams实例的生命周期。一个stream实例可以在配置文件中为处理器指定一个或多个Thread。

KafkaStreams实例可以作为单个streams处理客户端（也可能是分布式的），与其他的相同应用ID的实例进行协调（无论是否在同一个进程中，在同一台机器的其他进程中，或远程机器上）。这些实例将根据输入topic分区的基础上来划分工作，以便所有的分区都被消费掉。如果实例添加或失败，所有实例将重新平衡它们之间的分区分配，以保证负载平衡。

在内部，KafkaStreams实例包含一个正常的KafkaProducer和KafkaConsumer实例，用于读取和写入，

一个简单的例子：

Map<String, Object> props = new HashMap<>();

props.put(StreamsConfig.APPLICATION\_ID\_CONFIG, "my-stream-processing-application");

props.put(StreamsConfig.BOOTSTRAP\_SERVERS\_CONFIG, "localhost:9092");

props.put(StreamsConfig.KEY\_SERDE\_CLASS\_CONFIG, Serdes.String().getClass().getName());

props.put(StreamsConfig.VALUE\_SERDE\_CLASS\_CONFIG, Serdes.String().getClass().getName());

StreamsConfig config = new StreamsConfig(props);

KStreamBuilder builder = new KStreamBuilder();

builder.stream("my-input-topic").mapValues(value -> value.length().toString()).to("my-output-topic");

KafkaStreams streams = new KafkaStreams(builder, config);

streams.start();

**Kafka Connect API**

Connect API实现一个连接器（connector），不断地从一些数据源系统拉取数据到kafka，或从kafka推送到宿系统（sink system）。

大多数Connect使用者不需要直接操作这个API，可以使用之前构建的连接器，不需要编写任何代码。有关Connect的其他信息，点击[这里](http://orchome.com/343)。

想实现自定义连接器的，可以看[javadoc](http://kafka.apache.org/0101/javadoc/index.html?org/apache/kafka/connect)。

**Kafka Broker配置（0.10版）**

## 3.1 Broker配置

基本配置如下:

* broker.id
* log.dirs
* zookeeper.connect

下文将详细论述了主题级别配置和默认值。

| **名称** | **描述** | **类型** | **默认** | **有效值** | **重要程度** |
| --- | --- | --- | --- | --- | --- |
| zookeeper.connect | zookeeper host string | string |  |  | 高 |
| advertised.host.name | 过时的：当advertised.listeners或listeners没设置时候才使用。请改用advertised.listeners。Hostname发布到Zookeeper供客户端使用。在IaaS环境中，Broker可能需要绑定不同的接口。如果没有设置，将会使用host.name（如果配置了）。否则将从java.net.InetAddress.getCanonicalHostName()获取。 | string | null |  | 高 |
| advertised.listeners | 发布到Zookeeper供客户端使用监听（如果不同）。在IaaS环境中，broker可能需要绑定不同的接口。如果没设置，则使用listeners。 | string | null |  | 高 |
| advertised.port | 过时的：当advertised.listeners或listeners没有设置才使用。请改用advertised.listeners。端口发布到Zookeeper供客户端使用，在IaaS环境中，broker可能需要绑定到不同的端口。如果没有设置，将和broker绑定的同一个端口。 | int | null |  | 高 |
| auto.create.topics.enable | 启用自动创建topic | boolean | true |  | 高 |
| auto.leader.rebalance.enable | 启用自动平衡leader。如果需要，后台线程会定期检查并触发leader平衡。 | boolean | true |  | 高 |
| background.threads | 用于各种后台处理任务的线程数 | int | 10 | [1,...] | 高 |
| broker.id | 服务器的broker id。如果未设置，将生成一个独一无二的broker id。要避免zookeeper生成的broker id和用户配置的broker id冲突，从reserved.broker.max.id + 1开始生成。 | int | -1 |  | 高 |
| compression.type | 为给定topic指定最终的压缩类型。支持标准的压缩编码器（'gzip', 'snappy', 'lz4'）。也接受'未压缩'，就是没有压缩。保留由producer设置的原始的压缩编码。 | string | producer |  | 高 |
| delete.topic.enable | 启用删除topic。如果此配置已关闭，通过管理工具删除topic将没有任何效果 | boolean | false |  | 高 |
| host.name | 不赞成：当listeners没有设置才会使用。请改用listeners。如果设置，它将只绑定到此地址。如果没有设置，它将绑定到所有接口 | string | "" |  | 高 |
| leader.imbalance.check.interval.seconds | 由控制器触发分区再平衡检查的频率 | long | 300 |  | 高 |
| leader.imbalance.per.broker.percentage | 允许每个broker的leader比例不平衡。如果每个broker的值高于此值，控制器将触发leader平衡，该值以百分比的形式指定。 | int | 10 |  | 高 |
| listeners | 监听列表 - 监听逗号分隔的URL列表和协议。指定hostname为0.0.0.0绑定到所有接口，将hostname留空则绑定到默认接口。合法的listener列表是：PLAINTEXT://myhost:9092,TRACE://:9091 PLAINTEXT://0.0.0.0:9092, TRACE://localhost:9093 | string | null |  | 高 |
| log.dir | 保存日志数据的目录 (补充log.dirs属性) | string | /tmp/kafka-logs |  | 高 |
| log.dirs | 保存日志数据的目录。如果未设置，则使用log.dir中的值 | string | null |  | 高 |
| log.flush.interval.messages | 消息刷新到磁盘之前，累计在日志分区的消息数 | long | 9223372036854775807 | [1,...] | 高 |
| log.flush.interval.ms | topic中的消息在刷新到磁盘之前保存在内存中的最大时间（以毫秒为单位），如果未设置，则使用log.flush.scheduler.interval.ms中的值 | null |  | 高 |  |
| log.flush.offset.checkpoint.interval.ms | 我们更新的持续记录的最后一次刷新的频率。作为日志的恢复点。 | int | 60000 | [0,...] | 高 |
| log.flush.scheduler.interval.ms | 日志刷新的频率（以毫秒为单位）检查是否有任何日志需要刷新到磁盘 | long | 9223372036854775807 |  | 高 |
| log.retention.bytes | 删除日志之前的最大大小 | long | -1 |  | 高 |
| log.retention.hours | 删除日志文件保留的小时数（以小时为单位）。第三级是log.retention.ms属性 | int | 168 |  | 高 |
| log.retention.minutes | 删除日志文件之前保留的分钟数（以分钟为单位）。次于log.retention.ms属性。如果没设置，则使用log.retention.hours的值。 | int | null |  | 高 |
| log.retention.ms | 删除日志文件之前保留的毫秒数（以毫秒为单位），如果未设置，则使用log.retention.minutes的值。 | long | null |  | 高 |
| log.roll.hours | 新建一个日志段的最大时间（以小时为单位），次于log.roll.ms属性。 | int | 168 | [1,...] | 高 |
| log.roll.jitter.hours | 从logRollTimeMillis（以小时为单位）减去最大抖动，次于log.roll.jitter.ms属性。 | int | 0 | [0,...] | 高 |
| log.roll.ms | 新建一个日志段之前的最大事时间（以毫秒为单位）。如果未设置，则使用log.roll.hours的值。 | long | null |  | 高 |
| og.segment.bytes | 单个日志文件的最大大小 | int | 1073741824 | [14,...] | 高 |
| og.segment.delete.delay.ms | 从文件系统中删除文件之前的等待的时间 | long | 60000 | [0,...] | 高 |
| message.max.bytes | 服务器可以接收的消息的最大大小 | int | 1000012 | [0,...] | 高 |
| min.insync.replicas | 当producer设置acks为"all"（或"-1"）时。min.insync.replicas指定必须应答成功写入的replicas最小数。如果不能满足最小值，那么producer抛出一个异常（NotEnoughReplicas或NotEnoughReplicasAfterAppend）。 当一起使用时，min.insync.replicas和acks提供最大的耐用性保证。一个典型的场景是创建一个复制因子3的topic，设置min.insync.replicas为2，并且ack是“all”。如果多数副本没有接到写入时，将会抛出一个异常。 | int | 1 | [1,...] | 高 |
| num.io.threads | 服务器用于执行网络请求的io线程数 | int | 8 | [1,...] | 高 |
| num.network.threads | 服务器用于处理网络请求的线程数。 | int | 3 | [1,...] | 高 |
| num.recovery.threads.per.data.dir | 每个数据的目录线程数，用于启动时日志恢复和关闭时flush。 | int | 1 | [1,...] | 高 |
| num.replica.fetchers | 从源broker复制消息的提取线程数。递增该值可提高follower broker的I/O的并发。 | int | 1 |  | 高 |
| offset.metadata.max.bytes | offset提交关联元数据条目的最大大小 | int | 4096 |  | 高 |
| offsets.commit.required.acks | commit之前需要的应答数，通常，不应覆盖默认的（-1） | short | -1 |  | 高 |
| offsets.commit.timeout.ms | Offset提交延迟，直到所有副本都收到提交或超时。 这类似于生产者请求超时。 | int | 5000 | [1,...] | 高 |
| offsets.load.buffer.size | 当加载offset到缓存时，从offset段读取的批量大小。 | int | 5242880 | [1,...] | 高 |
| offsets.retention.check.interval.ms | 检查过期的offset的频率。 | long | 600000 | [1,...] | 高 |
| offsets.retention.minutes | offset topic的日志保留时间（分钟） | int | 1440 | [1,...] | 高 |
| offsets.topic.compression.codec | 压缩编码器的offset topic - 压缩可以用于实现“原子”提交 | int | 0 |  | 高 |
| offsets.topic.num.partitions | offset commit topic的分区数（部署之后不应更改） | int | 50 | [1,...] | 高 |
| offsets.topic.replication.factor | offset topic复制因子（ps：就是备份数，设置的越高来确保可用性）。为了确保offset topic有效的复制因子，第一次请求offset topic时，活的broker的数量必须最少最少是配置的复制因子数。 如果不是，offset topic将创建失败或获取最小的复制因子（活着的broker，复制因子的配置） | short | 3 | [1,...] | 高 |
| offsets.topic.segment.bytes | offset topic段字节应该相对较小一点，以便于加快日志压缩和缓存加载 | int | 104857600 | [1,...] | 高 |
| port | 不赞成：当listener没有设置才使用。请改用listeners。该port监听和接收连接。 | int | 9092 |  | 高 |
| queued.max.requests | 在阻塞网络线程之前允许的排队请求数 | int | 500 | [1,...] | 高 |
| quota.consumer.default | 过时的：当默认动态的quotas没有配置或在Zookeeper时。如果每秒获取的字节比此值高，所有消费者将通过clientId/consumer区分限流。 | long | 9223372036854775807 | [1,...] | 高 |
| quota.producer.default | 过时的：当默认动态的quotas没有配置，或在zookeeper时。如果生产者每秒比此值高，所有生产者将通过clientId区分限流。 | long | 9223372036854775807 | [1,...] | 高 |
| replica.fetch.min.bytes Minimum | 每个获取响应的字节数。如果没有满足字节数，等待replicaMaxWaitTimeMs。 | int | 1 |  | 高 |
| replica.fetch.wait.max.ms | 跟随者副本发出每个获取请求的最大等待时间，此值应始终小于replica.lag.time.max.ms，以防止低吞吐的topic的ISR频繁的收缩。 | int | 500 |  | 高 |
| replica.high.watermark. checkpoint.interval.ms | 达到高“水位”保存到磁盘的频率。 | long | 5000 |  | 高 |
| replica.lag.time.max.ms | 如果一个追随者没有发送任何获取请求或至少在这个时间的这个leader的没有消费完。该leader将从isr中移除这个追随者。 | long | 10000 |  | 高 |
| replica.socket.receive.buffer.bytes | 用于网络请求的socket接收缓存区 | int | 65536 |  | 高 |
| replica.socket.timeout.ms | 网络请求的socket超时，该值最少是replica.fetch.wait.max.ms | int | 30000 |  | 高 |
| request.timeout.ms | 该配置控制客户端等待请求的响应的最大时间，。如果超过时间还没收到消费。客户端将重新发送请求，如果重试次数耗尽，则请求失败。 | int | 30000 |  | 高 |
| socket.receive.buffer.bytes | socket服务的SO\_RCVBUF缓冲区。如果是-1，则默认使用OS的。 | int | 102400 |  | 高 |
| socket.request.max.bytes | socket请求的最大字节数 | int | 104857600 | [1,...] | 高 |
| socket.send.buffer.bytes | socket服务的SO\_SNDBUF缓冲区。如果是-1，则默认使用OS的。 | int | 102400 |  | 高 |
| unclean.leader.election.enable | 是否启用不在ISR中的副本参与选举leader的最后的手段。这样做有可能丢失数据。 | boolean | true |  | 高 |
| zookeeper.connection.timeout.ms | 连接zookeeper的最大等待时间，如果未设置，则使用zookeeper.session.timeout.ms。 | int | null |  | 高 |
| zookeeper.session.timeout.ms | Zookeeper会话的超时时间 | int | 6000 |  | 高 |
| zookeeper.set.acl | 设置客户端使用安全的ACL | boolean | false |  | 高 |
| broker.id.generation.enable | 启用自动生成broker id。启用该配置时应检查reserved.broker.max.id。 | boolean | true |  | 中等 |
| broker.rack | broker机架，用于机架感知副本分配的失败容错。例如：RACK1, us-east-1d | string | null |  | 中等 |
| connections.max.idle.ms Idle | 连接超时：闲置时间超过该设置，则服务器socket处理线程关闭这个连接。 | long | 600000 |  | 中等 |
| controlled.shutdown.enable | 启用服务器的关闭控制。 | boolean | true |  | 中等 |
| controlled.shutdown.max.retries | 控制因多种原因导致的shutdown失败，当这样失败发生，尝试重试的次数 | int | 3 |  | 中等 |
| controlled.shutdown.retry.backoff.ms | 在每次重试之前，系统需要时间从导致先前故障的状态（控制器故障转移，复制延迟等）恢复。 此配置是重试之前等待的时间数。 | long | 5000 |  | 中等 |
| controller.socket.timeout.ms | 控制器到broker通道的sockt超时时间 | int | 30000 |  | 中 |
| default.replication.factor | 自动创建topic的默认的副本数 | int | 1 |  | 中 |
| fetch.purgatory.purge.interval.requests | 拉取请求清洗间隔（请求数） | int | 1000 |  | 中 |
| group.max.session.timeout.ms | 已注册的消费者允许的最大会话超时时间，设置的时候越长使消费者有更多时间去处理心跳之间的消息。但察觉故障的时间也拉长了。 | int | 300000 |  | 中 |
| group.min.session.timeout.ms | 已经注册的消费者允许最小的会话超时时间，更短的时间去快速的察觉到故障，代价是频繁的心跳，这可能会占用大量的broker资源。 | int | 6000 |  | 中 |
| inter.broker.protocol.version | 指定broker内部通讯使用的版本。通常在更新broker时使用。有效的值为：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。查看ApiVersion找到的全部列表。 | string | 0.10.1-IV2 |  | 中 |
| log.cleaner.backoff.ms | 当没有日志要清理时，休眠的时间 | long | 15000 | [0,...] | 中 |
| log.cleaner.dedupe.buffer.size | 用于日志去重的内存总量（所有cleaner线程） | long | 134217728 |  | 中 |
| log.cleaner.delete.retention.ms | 删除记录保留多长时间？ | long | 86400000 |  | 中 |
| log.cleaner.enable | 在服务器上启用日志清洗处理？如果使用的任何topic的cleanup.policy=compact包含内部的offset topic，应启动。如果禁用，那些topic将不会被压缩并且会不断的增大。 | boolean | true |  | 中 |
| log.cleaner.io.buffer.load.factor | 日志cleaner去重缓冲负载因子。去重缓冲区的百分比，较高的值将允许同时清除更多的日志，但将会导致更多的hash冲突。 | double | 0.9 |  | 中 |
| log.cleaner.io.buffer.size | 所有日志清洁器线程I/O缓存的总内存 | int | 524288 | [0,...] | 中 |
| log.cleaner.io.max.bytes.per.second | 日志清理器限制，以便其读写i/o平均小与此值。 | double | 1.7976931348623157E308 |  | 中 |
| log.cleaner.min.cleanable.ratio | 脏日志与日志的总量的最小比率，以符合清理条件 | double | 0.5 |  | 中 |
| log.cleaner.min.compaction.lag.ms | 一条消息在日志保留不压缩的最小时间，仅适用于正在压缩的日志。 | long | 0 |  | 中 |
| log.cleaner.threads | 用于日志清除的后台线程数 | int | 1 | [0,...] | 中 |
| log.cleanup.policy | 超过保留时间段的默认清除策略。逗号分隔的有效的策略列表。有效的策略有：“delete”和“compact” | list | [delete] | [compact, delete] | 中 |
| log.index.interval.bytes | 添加一个条目到offset的间隔 | index | int | 4096 | [0,...] | 中 |
| log.index.size.max.bytes | offset index的最大大小（字节） | int | 10485760 | [4,...] | 中 |  |
| log.message.format.version | 指定追加到日志中的消息格式版本。例如： 0.8.2, 0.9.0.0, 0.10.0。通过设置一个特定消息格式版本，用户需要保证磁盘上所有现有的消息小于或等于指定的版本。错误的设置将导致旧版本的消费者中断，因为消费者接收一个不理解的消息格式。 | string | 0.10.1-IV2 |  | 中 |  |
| log.message.timestamp.difference.max.ms | 如果log.message.timestamp.type=CreateTime，broker接收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。如果时间戳超过此阈值，则消息将被拒绝。如果log.message.timestamp.type=LogAppendTime，则此配置忽略。 | long | 9223372036854775807 | [0,...] | 中 |  |
| log.message.timestamp.type | 定义消息中的时间戳是消息创建时间或日志追加时间。该值可设置为CreateTime 或 LogAppendTime | string | CreateTime | [CreateTime, LogAppendTime] | 中 |  |
| log.preallocate | 在创建新段时预分配文件？如果你在Windowns上使用kafka，你可能需要设置它为true。 | boolean | false |  | 中 |  |
| log.retention.check.interval.ms | 日志清除程序检查日志是否满足被删除的频率（以毫秒为单位） | long | 300000 | [1,...] | 中 |  |
| max.connections.per.ip | 允许每个ip地址的最大连接数。 | int | 2147483647 | [1,...] | 中 |  |
| max.connections.per.ip.overrides | per-ip或hostname覆盖默认最大连接数 | string | "" |  | 中 |  |
| num.partitions | topic的默认分区数 | int | 1 | [1,...] | 中 |  |
| principal.builder.class | 实现PrincipalBuilder接口类的完全限定名，该接口目前用于构建与SSL SecurityProtocol连接的Principal。 | class | class org.apache.kafka. common.security.auth .DefaultPrincipalBuilder |  | 中 |  |
| producer.purgatory.purge.interval.requests | 生产者请求purgatory的清洗间隔（请求数） | int | 1000 |  | 中 |  |
| replica.fetch.backoff.ms | 当拉取分区发生错误时休眠的时间 | 1000 | [0,...] |  | 中 |  |
| replica.fetch.max.bytes | 拉取每个分区的消息的字节数。这不是绝对的最大值，如果提取的第一个非空分区中的第一个消息大于这个值，则消息仍然返回，以确保进展。通过message.max.bytes (broker配置)或max.message.bytes (topic配置)定义broker接收的最大消息大小。 | int | 1048576 | [0,...] | 中 |  |
| replica.fetch.response.max.bytes | 预计整个获取响应的最大字节数，这不是绝对的最大值，如果提取的第一个非空分区中的第一个消息大于这个值，则消息仍然返回，以确保进展。通过message.max.bytes (broker配置)或max.message.bytes (topic配置)定义broker接收的最大消息大小。 | int | 10485760 | [0,...] | 中 |  |
| reserved.broker.max.id | broker.id的最大数 | int | 1000 | [0,...] | 中 |  |
| sasl.enabled.mechanisms | 可用的SASL机制列表，包含任何可用的安全提供程序的机制。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 | 登录线程在刷新尝试的休眠时间。 | long | 60000 |  | 中 |  |
| sasl.kerberos.principal.to.local.rules | principal名称映射到一个短名称（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其到短名称。忽略后面的规则。默认情况下，{username}/{hostname}@{REALM} 映射到 {username}。 | list | [DEFAULT] |  | 中 |  |
| sasl.kerberos.service.name | Kafka运行的Kerberos principal名称。 可以在JAAS或Kafka的配置文件中定义。 | string | null |  | 中 |  |
| sasl.kerberos.ticket.renew.jitter | 添加到更新时间的随机抖动的百分比 | time. double | 0.05 |  | 中 |  |
| sasl.kerberos.ticket.renew.window.factor | 登录线程休眠，直到从上次刷新到ticket的到期的时间已到达（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.protocolSecurity | broker之间的通讯协议，有效值有：PLAINTEXT, SSL, SASL\_PLAINTEXT, SASL\_SSL。 | string | PLAINTEXT |  | 中 |  |
| ssl.cipher.suites | 密码套件列表。认证，加密，MAC和秘钥交换算法的组合，用于使用TLS或SSL的网络协议交涉网络连接的安全设置，默认情况下，支持所有可用的密码套件。 | 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* 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Java虚拟机的密钥管理工厂算法。 | 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，由于有已知的安全漏洞，不建议使用。 | string | TLS |  | 中 |  |
| ssl.provider | 用于SSL连接的安全提供程序的名称。默认值是JVM的安全程序。 | 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 | 信任仓库文件的位置 | string | null |  | 中 |  |
| ssl.truststore.password | 信任仓库文件的密码 | password | null |  | 中 |  |
| ssl.truststore.type | 信任仓库文件的文件格式 | string | JKS |  | 中 |  |
| authorizer.class.name | 用于认证的授权程序类 | string | "" |  | 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.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 |  |
| zookeeper.sync.time.ms | ZK follower可落后与leader多久。 | int | 2000 |  | 低 |  |

更多关于broker配置的详情，可以在scala类中的kafka.server.KafkaConfig找到。

可查看之前的版本：[Kafka Broker配置（0.8.2）](http://orchome.com/12)

**Kafka Topic配置**

## 3.2 Topic配置

与topic相关的配置，服务器的默认值，也可可选择的覆盖指定的topic。如果没有给出指定topic的配置，则将使用服务器默认值。 可以通过-config选项在topic创建时设置。此示例使用自定义最大消息的大小和刷新率，创建一个名为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

也可以使用alter configs命令修改或设置。 此示例修改更新my-topic的最大的消息大小：

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

--alter --add-config max.message.bytes=128000

你可以执行以下命令验证结果

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

移除：

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

以下是topic级配置。是服务器的默认配置。服务器默认配置值仅适用于topic。

| **NAME** | **DESCRIPTION** | **TYPE** | **DEFAULT** | **VALID VALUES** | **SERVER DEFAULT PROPERTY** | **IMPORTANCE** |
| --- | --- | --- | --- | --- | --- | --- |
| cleanup.policy | “delete”或“compact”。指定在旧的日志段的保留策略。默认策略（“delete”），将达到保留时间或大小限制的日志废弃。 “compact”则压缩日志。 | list | delete | [compact, delete] | log.cleanup.policy | medium |
| compression.type | 针对指定的topic设置最终的压缩方式。标准的压缩格式有'gzip', 'snappy', lz4。还可以设置'uncompressed',就是不压缩；设置为'producer'这意味着保留生产者设置的原始压缩编解码。 | string | producer | [uncompressed, snappy, lz4, gzip, producer] | compression.type | medium |
| delete.retention.ms | 保留删除消息压缩topic的删除标记的时间。此设置还给出消费者如果从offset 0开始读取并确保获得最终阶段的有效快照的时间范围（否则，在完成扫描之前可能已经回收了）。 | long | 86400000 | [0,...] | log.cleaner.delete.retention.ms | medium |
| file.delete.delay.ms | 从文件系统中删除文件之前等待的时间 | long | 60000 | [0,...] | log.segment.delete.delay.ms | medium |
| flush.messages | 此设置允许指定我们强制fsync写入日志的数据的间隔。例如，如果这被设置为1，我们将在每个消息之后fsync; 如果是5，我们将在每五个消息之后fsync。一般，我们建议不要设置它，使用复制特性来保持持久性，并允许操作系统的后台刷新功能更高效。可以在每个topic的基础上覆盖此设置（请参阅每个主题的配置部分）。 | medium | long | 9223372036854775807 | [0,...] | log.flush.interval.messages |
| flush.ms | 此设置允许我们强制fsync写入日志的数据的时间间隔。例如，如果这设置为1000，那么在1000ms过去之后，我们将fsync。 一般，我们建议不要设置它，并使用复制来保持持久性，并允许操作系统的后台刷新功能，因为它更有效率 | long | 9223372036854775807 | [0,...] | log.flush.interval.ms | medium |
| follower.replication.throttled.replicas | follower复制限流列表。该列表应以[PartitionId]的形式描述一组副本：[BrokerId]，[PartitionId]：[BrokerId]：...或者通配符'\*'可用于限制此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offset索引添加索引条目的频率。默认设置确保我们大致每4096个字节索引消息。 更多的索引允许读取更接近日志中的确切位置，但使索引更大。你不需要改变这个值。 | int | 4096 | [0,...] | log.index.interval.bytes | medium |
| leader.replication.throttled.replicas | 在leader方面进行限制的副本列表。该列表设置以[PartitionId]的形式描述限制副本：[PartitionId]:[BrokerId],[PartitionId]:[BrokerId]:...或使用通配符‘\*’限制该topic的所有副本。 | list | "" | [partitionId],[brokerId]:[partitionId],[brokerId]:... | leader.replication.throttled.replicas | medium |
| max.message.bytes | kafka允许的最大的消息批次大小。如果增加此值，则消费者的版本比0.10.2老，那么消费者的提取的大小也必须增加，以便他们可以获取大的消息批次。 在最新的消息格式版本中，消息总是分组批量来提高效率。在之前的消息格式版本中，未压缩的记录不会分组批量，并且此限制仅适用于该情况下的单个消息。 | int | 1000012 | [0,...] | message.max.bytes | medium |
| message.format.version | 指定消息附加到日志的消息格式版本。该值应该是一个有效的ApiVersion。例如：0.8.2, 0.9.0.0, 0.10.0，更多细节检查ApiVersion。通过设置特定的消息格式版本，并且磁盘上的所有现有消息都小于或等于指定版本。不正确地设置此值将导致消费者使用旧版本，因为他们将接收到“不认识”的格式的消息。 | string | 0.11.0-IV2 |  | log.message.format.version | medium |
| min.cleanable.dirty.ratio | 此配置控制日志压缩程序将尝试清除日志的频率(假设启用了日志压缩)。默认情况下，我们将避免清理超过50％日志被压缩的日志。 该比率限制日志中浪费的最大空间重复(在最多50％的日志中可以是重复的50％)。更高的比率意味着更少，更有效的清洁，但意味着日志中的浪费更多。 | double | 0.5 | [0,...,1] | log.cleaner.min.cleanable.ratio | medium |
| min.compaction.lag.ms | 消息在日志中保持不压缩的最短时间。仅适用于正在压缩的日志。 | long | 0 | [0,...] | log.cleaner.min.compaction.lag.ms | medium |
| min.insync.replicas | 当生产者设置应答为"all"(或“-1”)时，此配置指定了成功写入的副本应答的最小数。如果没满足此最小数，则生产者将引发异常(NotEnoughReplicas或NotEnoughReplicasAfterAppend) 当min.insync.replicas和acks强制更大的耐用性时。典型的情况是创建一个副本为3的topic，将min.insync.replicas设置为2，并设置acks为“all”。如果多数副本没有收到写入，这将确保生产者引发异常。 | int | 1 | [1,...] | min.insync.replicas | medium |
| preallocate | 如果我们在创建新的日志段时在磁盘上预分配该文件，那么设为True。 | boolean | false |  | log.preallocate | medium |
| retention.bytes | 如果我们使用“删除”保留策略，则此配置将控制日志可以增长的最大大小，之后我们将丢弃旧的日志段以释放空间。默认情况下，没有设置大小限制则仅限于时间限制。 | long | -1 |  | log.retention.bytes | medium |
| retention.ms | 如果我们使用“删除”保留策略，则此配置控制我们将保留日志的最长时间，然后我们将丢弃旧的日志段以释放空间。这代表SLA消费者必须读取数据的时间长度。 | long | 604800000 |  | log.retention.ms | medium |
| segment.bytes | 此配置控制日志的段文件大小。一次保留和清理一个文件，因此较大的段大小意味着较少的文件，但对保留率的粒度控制较少。 | int | 1073741824 | [14,...] | log.segment.bytes | medium |
| segment.index.bytes | 此配置控制offset映射到文件位置的索引的大小。我们预先分配此索引文件，并在日志滚动后收缩它。通常不需要更改此设置。 | int | 10485760 | [0,...] | log.index.size.max.bytes | medium |
| segment.jitter.ms | 从计划的段滚动时间减去最大随机抖动，以避免异常的段滚动 | long | 0 | [0,...] | log.roll.jitter.ms | medium |
| segment.ms | 此配置控制Kafka强制日志滚动的时间段，以确保保留可以删除或压缩旧数据，即使段文件未满。 | long | 604800000 | [0,...] | log.roll.ms | medium |
| unclean.leader.election.enable | 是否将不在ISR中的副本作为最后的手段选举为leader，即使这样做可能会导致数据丢失。 | boolean | false |  |  |  |

**Kafka Producer配置**

## 3.3 生产者配置

java生产者配置：

| **NAME** | **DESCRIPTION** | **TYPE** | **DEFAULT** | **VALID VALUES** | **IMPORTANCE** |
| --- | --- | --- | --- | --- | --- |
| bootstrap.servers | host/port列表，用于初始化建立和Kafka集群的连接。列表格式为host1:port1,host2:port2,....，无需添加所有的集群地址，kafka会根据提供的地址发现其他的地址（你可以多提供几个，以防提供的服务器关闭） | list |  |  | high |
| key.serializer | key的序列化类（实现序列化接口） | class |  |  | high |
| value.serializer | value的序列化类（实现序列化接口） | class |  |  | high |
| acks | 生产者需要leader确认请求完成之前接收的应答数。此配置控制了发送消息的耐用性，支持以下配置： acks=0 如果设置为0，那么生产者将不等待任何消息确认。消息将立刻天际到socket缓冲区并考虑发送。在这种情况下不能保障消息被服务器接收到。并且重试机制不会生效（因为客户端不知道故障了没有）。每个消息返回的offset始终设置为-1。 acks=1，这意味着leader写入消息到本地日志就立即响应，而不等待所有follower应答。在这种情况下，如果响应消息之后但follower还未复制之前leader立即故障，那么消息将会丢失。 acks=all 这意味着leader将等待所有副本同步后应答消息。此配置保障消息不会丢失（只要至少有一个同步的副本或者）。这是最强壮的可用性保障。等价于acks=-1。 | string | 1 | [all, -1, 0, 1] | high |
| buffer.memory | 生产者用来缓存等待发送到服务器的消息的内存总字节数。如果消息发送比可传递到服务器的快，生产者将阻塞max.block.ms之后，抛出异常。 此设置应该大致的对应生产者将要使用的总内存，但不是硬约束，因为生产者所使用的所有内存都用于缓冲。一些额外的内存将用于压缩（如果启动压缩），以及用于保持发送中的请求。 | long | 33554432 | [0,...] | high |
| compression.type | 数据压缩的类型。默认为空（就是不压缩）。有效的值有 none，gzip，snappy, 或 lz4。压缩全部的数据批，因此批的效果也将影响压缩的比率（更多的批次意味着更好的压缩）。 | string | none |  | high |
| retries | 设置一个比零大的值，客户端如果发送失败则会重新发送。注意，这个重试功能和客户端在接到错误之后重新发送没什么不同。如果max.in.flight.requests.per.connection没有设置为1，有可能改变消息发送的顺序，因为如果2个批次发送到一个分区中，并第一个失败了并重试，但是第二个成功了，那么第二个批次将超过第一个。 | int | 0 | [0,...,2147483647] | high |
| ssl.key.password | 密钥仓库文件中的私钥的密码。 | password | null |  | high |
| ssl.keystore.location | 密钥仓库文件的位置。可用于客户端的双向认证。 | string | null |  | high |
| ssl.keystore.password | 密钥仓库文件的仓库密码。只有配置了ssl.keystore.location时才需要。 |  |  |  |  |
|  | password | null |  | high |  |
| ssl.truststore.location | 信任仓库的位置 | string | null |  | high |
| ssl.truststore.password | 信任仓库文件的密码 | password | null |  | high |
| batch.size | 当多个消息要发送到相同分区的时，生产者尝试将消息批量打包在一起，以减少请求交互。这样有助于客户端和服务端的性能提升。该配置的默认批次大小（以字节为单位）： 不会打包大于此配置大小的消息。 发送到broker的请求将包含多个批次，每个分区一个，用于发送数据。 较小的批次大小有可能降低吞吐量（批次大小为0则完全禁用批处理）。一个非常大的批次大小可能更浪费内存。因为我们会预先分配这个资源。 | int | 16384 | [0,...] | medium |
| client.id | 当发出请求时传递给服务器的id字符串。这样做的目的是允许服务器请求记录记录这个【逻辑应用名】，这样能够追踪请求的源，而不仅仅只是ip/prot。 | string | "" |  | medium |
| connections.max.idle.ms | 多少毫秒之后关闭闲置的连接。 | long | 540000 |  | medium |
| linger.ms | 生产者组将发送的消息组合成单个批量请求。正常情况下，只有消息到达的速度比发送速度快的情况下才会出现。但是，在某些情况下，即使在适度的负载下，客户端也可能希望减少请求数量。此设置通过添加少量人为延迟来实现。- 也就是说，不是立即发出一个消息，生产者将等待一个给定的延迟，以便和其他的消息可以组合成一个批次。这类似于Nagle在TCP中的算法。此设置给出批量延迟的上限：一旦我们达到分区的batch.size值的记录，将立即发送，不管这个设置如何，但是，如果比这个小，我们将在指定的“linger”时间内等待更多的消息加入。此设置默认为0（即无延迟）。假设，设置 linger.ms=5，将达到减少发送的请求数量的效果，但对于在没有负载情况，将增加5ms的延迟。 | long | 0 | [0,...] | medium |
| max.block.ms | 该配置控制 KafkaProducer.send() 和 KafkaProducer.partitionsFor() 将阻塞多长时间。此外这些方法被阻止，也可能是因为缓冲区已满或元数据不可用。在用户提供的序列化程序或分区器中的锁定不会计入此超时。 | long | 60000 | [0,...] | medium |
| max.request.size | 请求的最大大小（以字节为单位）。此设置将限制生产者的单个请求中发送的消息批次数，以避免发送过大的请求。这也是最大消息批量大小的上限。请注意，服务器拥有自己的批量大小，可能与此不同。 | int | 1048576 | [0,...] | medium |
| partitioner.class | 实现Partitioner接口的的Partitioner类。 | class | org.apache.kafka.clients.producer.internals.DefaultPartitioner |  | medium |
| receive.buffer.bytes | 读取数据时使用的TCP接收缓冲区(SO\_RCVBUF)的大小。如果值为-1，则将使用OS默认值。 | int | 32768 | [-1,...] | medium |
| 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，客户端将在必要时重新发送请求，如果重试耗尽，则该请求将失败。 这应该大于replica.lag.time.max.ms，以减少由于不必要的生产者重试引起的消息重复的可能性。 | int | 30000 | [0,...] | medium |
| sasl.jaas.config | JAAS配置文件使用的格式的SASL连接的JAAS登录上下文参数。[这里](http://docs.oracle.com/javase/8/docs/technotes/guides/security/jgss/tutorials/LoginConfigFile.html)描述JAAS配置文件格式。该值的格式为：'（=）\*;' | password | null |  | medium |
| sasl.kerberos.service.name | Kafka运行的Kerberos主体名称。可以在Kafka的JAAS配置或Kafka的配置中定义。 | string | null |  | medium |
| sasl.mechanism | SASL机制用于客户端连接。这是安全提供者可用与任何机制。GSSAPI是默认机制。 | string | GSSAPI |  | medium |
| security.protocol | 用于与broker通讯的协议。 有效值为：PLAINTEXT，SSL，SASL\_PLAINTEXT，SASL\_SSL。 | string | PLAINTEXT |  | medium |
| send.buffer.bytes | 发送数据时，用于TCP发送缓存（SO\_SNDBUF）的大小。如果值为 -1，将默认使用系统的。 | int | 131072 | [-1,...] | medium |
| ssl.enabled.protocols | 启用SSL连接的协议列表。 | list | TLSv1.2,TLSv1.1,TLSv1 |  | medium |
| ssl.keystore.type | 密钥存储文件的文件格式。对于客户端是可选的。 | string | JKS |  | medium |
| ssl.protocol | 最近的JVM中允许的值是TLS，TLSv1.1和TLSv1.2。 较旧的JVM可能支持SSL，SSLv2和SSLv3，但由于已知的安全漏洞，不建议使用SSL。 | string | TLS |  | medium |
| ssl.provider | 用于SSL连接的安全提供程序的名称。默认值是JVM的默认安全提供程序。 | string | null |  | medium |
| ssl.truststore.type | 信任仓库文件的文件格式。 | string | JKS |  | medium |
| enable.idempotence | 当设置为‘true’，生产者将确保每个消息正好一次复制写入到stream。如果‘false’，由于broker故障，生产者重试。即，可以在流中写入重试的消息。此设置默认是‘false’。请注意，启用幂等式需要将max.in.flight.requests.per.connection设置为1，重试次数不能为零。另外acks必须设置为“全部”。如果这些值保持默认值，我们将覆盖默认值。 如果这些值设置为与幂等生成器不兼容的值，则将抛出一个ConfigException异常。如果这些值设置为与幂等生成器不兼容的值，则将抛出一个ConfigException异常。 | boolean | false |  | low |
| interceptor.classes | 实现ProducerInterceptor接口，你可以在生产者发布到Kafka群集之前拦截（也可变更）生产者收到的消息。默认情况下没有拦截器。 | list | null |  | low |
| max.in.flight.requests.per.connection | 阻塞之前，客户端单个连接上发送的未应答请求的最大数量。注意，如果此设置设置大于1且发送失败，则会由于重试（如果启用了重试）会导致消息重新排序的风险。 | int | 5 | [1,...] | low |
| metadata.max.age.ms | 在一段时间段之后（以毫秒为单位），强制更新元数据，即使我们没有看到任何分区leader的变化，也会主动去发现新的broker或分区。 | long | 300000 | [0,...] | low |
| metric.reporters | 用作metrics reporters（指标记录员）的类的列表。实现MetricReporter接口，将受到新增加的度量标准创建类插入的通知。 JmxReporter始终包含在注册JMX统计信息中。 | list | "" |  | low |
| metrics.num.samples | 维护用于计算度量的样例数量。 | int | 2 | [1,...] | low |
| metrics.recording.level | 指标的最高记录级别。 | string | INFO | [INFO, DEBUG] | low |
| metrics.sample.window.ms | 度量样例计算上 | long | 30000 | [0,...] | low |
| reconnect.backoff.max.ms | 重新连接到重复无法连接的代理程序时等待的最大时间（毫秒）。 如果提供，每个主机的回退将会连续增加，直到达到最大值。 计算后退增加后，增加20％的随机抖动以避免连接风暴。 | long | 1000 | [0,...] | low |
| reconnect.backoff.ms | 尝试重新连接到给定主机之前等待的基本时间量。这避免了在循环中高频率的重复连接到主机。这种回退适应于客户端对broker的所有连接尝试。 | long | 50 | [0,...] | low |
| retry.backoff.ms | 尝试重试指定topic分区的失败请求之前等待的时间。这样可以避免在某些故障情况下高频次的重复发送请求。 | long | 100 | [0,...] | low |
| sasl.kerberos.kinit.cmd | Kerberos kinit 命令路径。 | string | /usr/bin/kinit |  | low |
| sasl.kerberos.min.time.before.relogin | Login线程刷新尝试之间的休眠时间。 | long | 60000 |  | low |
| sasl.kerberos.ticket.renew.jitter | 添加更新时间的随机抖动百分比。 | double | 0.05 |  | low |
| sasl.kerberos.ticket.renew.window.factor | 登录线程将睡眠，直到从上次刷新ticket到期时间的指定窗口因子为止，此时将尝试续订ticket。 | double | 0.8 |  | low |
| ssl.cipher.suites | 密码套件列表。这是使用TLS或SSL网络协议来协商用于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，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端点识别算法。 | string | null |  | low |
| ssl.keymanager.algorithm | 用于SSL连接的密钥管理因子算法。默认值是为Java虚拟机配置的密钥管理器工厂算法。 |  |  |  |  |
|  | string | SunX509 |  | low |  |
| ssl.secure.random.implementation | 用于SSL加密操作的SecureRandom PRNG实现。 | string | null |  | low |
| ssl.trustmanager.algorithm | 用于SSL连接的信任管理因子算法。默认值是JAVA虚拟机配置的信任管理工厂算法。 | string | PKIX |  | low |
| transaction.timeout.ms | 生产者在主动中止正在进行的交易之前，交易协调器等待事务状态更新的最大时间（以ms为单位）。如果此值大于broker中的max.transaction.timeout.ms设置，则请求将失败，并报“InvalidTransactionTimeout”错误。 | int | 60000 |  | low |
| transactional.id | 用于事务传递的TransactionalId。这样可以跨多个生产者会话的可靠性语义，因为它允许客户端保证在开始任何新事务之前使用相同的TransactionalId的事务已经完成。如果没有提供TransactionalId，则生产者被限制为幂等传递。请注意，如果配置了TransactionalId，则必须启用enable.idempotence。 默认值为空，这意味着无法使用事务。 | string | null | non-empty string | low |

对于那些对传统Scala生产者配置感兴趣的用户，可以在[这里](http://orchome.com/14)找到。

**Kafka Consumer配置**

# 3.4 kafka消费者配置

在0.9.0.0中，我们引入了新的Java消费者来替代早期基于Scala的简单和高级消费者。新老客户端的配置如下。

## 3.4.1 新消费者配置

新消费者配置：（注意，右面是可拖动的）

| **NAME** | **DESCRIPTION** | **TYPE** | **DEFAULT** | **VALID VALUES** | **IMPORTANCE** |
| --- | --- | --- | --- | --- | --- |
| bootstrap.servers | host/port,用于和kafka集群建立初始化连接。因为这些服务器地址仅用于初始化连接，并通过现有配置的来发现全部的kafka集群成员（集群随时会变化），所以此列表不需要包含完整的集群地址（但尽量多配置几个，以防止配置的服务器宕机）。 | list |  |  | high |
| key.deserializer | key的解析序列化接口实现类（Deserializer）。 | class |  |  | high |
| value.deserializer | value的解析序列化接口实现类（Deserializer） | class |  |  | high |
| fetch.min.bytes | 服务器哦拉取请求返回的最小数据量，如果数据不足，请求将等待数据积累。默认设置为1字节，表示只要单个字节的数据可用或者读取等待请求超时，就会应答读取请求。将此值设置的越大将导致服务器等待数据累积的越长，这可能以一些额外延迟为代价提高服务器吞吐量。 | int | 1 | [0,...] | high |
| group.id | 此消费者所属消费者组的唯一标识。如果消费者用于订阅或offset管理策略的组管理功能，则此属性是必须的。 | string | "" |  | high |
| heartbeat.interval.ms | 当使用Kafka的分组管理功能时，心跳到消费者协调器之间的预计时间。心跳用于确保消费者的会话保持活动状态，并当有新消费者加入或离开组时方便重新平衡。该值必须必比session.timeout.ms小，通常不高于1/3。它可以调整的更低，以控制正常重新平衡的预期时间。 | int | 3000 |  | high |
| max.partition.fetch.bytes | 服务器将返回每个分区的最大数据量。如果拉取的第一个非空分区中第一个消息大于此限制，则仍然会返回消息，以确保消费者可以正常的工作。broker接受的最大消息大小通过message.max.bytes（broker config）或max.message.bytes (topic config)定义。参阅fetch.max.bytes以限制消费者请求大小。 | int | 1048576 | [0,...] | high |
| session.timeout.ms | 用于发现消费者故障的超时时间。消费者周期性的发送心跳到broker，表示其还活着。如果会话超时期满之前没有收到心跳，那么broker将从分组中移除消费者，并启动重新平衡。请注意，该值必须在broker配置的group.min.session.timeout.ms和group.max.session.timeout.ms允许的范围内。 | int | 10000 |  | high |
| ssl.key.password | 密钥存储文件中的私钥的密码。 客户端可选 | password | null |  | high |
| ssl.keystore.location | 密钥存储文件的位置， 这对于客户端是可选的，并且可以用于客户端的双向认证。 | string | null |  | high |
| 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 |  | high |
| ssl.truststore.location | 信任仓库文件的位置 | string | null |  | high |
| ssl.truststore.password | 信任仓库文件的密码 | password | null |  | high |
| auto.offset.reset | 当Kafka中没有初始offset或如果当前的offset不存在时（例如，该数据被删除了），该怎么办。 最早：自动将偏移重置为最早的偏移最新：自动将偏移重置为最新偏移 none：如果消费者组找到之前的offset，则向消费者抛出异常其他：抛出异常给消费者。 | string | latest | [latest, earliest, none] | medium |
| connections.max.idle.ms | 指定在多少毫秒之后关闭闲置的连接 | long | 540000 |  | medium |
| enable.auto.commit | 如果为true，消费者的offset将在后台周期性的提交 | boolean | true |  | medium |
| exclude.internal.topics | 内部topic的记录（如偏移量）是否应向消费者公开。如果设置为true，则从内部topic接受记录的唯一方法是订阅它。 | boolean | true |  | medium |
| fetch.max.bytes | 服务器为拉取请求返回的最大数据值。这不是绝对的最大值，如果在第一次非空分区拉取的第一条消息大于该值，该消息将仍然返回，以确保消费者继续工作。接收的最大消息大小通过message.max.bytes (broker config) 或 max.message.bytes (topic config)定义。注意，消费者是并行执行多个提取的。 | int | 52428800 | [0,...] | medium |
| max.poll.interval.ms | 使用消费者组管理时poll()调用之间的最大延迟。消费者在获取更多记录之前可以空闲的时间量的上限。如果此超时时间期满之前poll()没有调用，则消费者被视为失败，并且分组将重新平衡，以便将分区重新分配给别的成员。 | int | 300000 | [1,...] | medium |
| max.poll.records | 在单次调用poll()中返回的最大记录数。 | int | 500 | [1,...] | medium |
| partition.assignment.strategy | 当使用组管理时，客户端将使用分区分配策略的类名来分配消费者实例之间的分区所有权 | list | class org.apache.kafka .clients.consumer .RangeAssignor |  | medium |
| receive.buffer.bytes | 读取数据时使用的TCP接收缓冲区（SO\_RCVBUF）的大小。 如果值为-1，则将使用OS默认值。 | int | 65536 | [-1,...] | medium |
| request.timeout.ms | 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，客户端将在必要时重新发送请求，如果重试耗尽则客户端将重新发送请求。 | int | 305000 | [0,...] | medium |
| sasl.jaas.config | JAAS配置文件中SASL连接登录上下文参数。 这里描述JAAS配置文件格式。 该值的格式为： '(=)\*;' | password | null |  | medium |
| sasl.kerberos.service.name | Kafka运行Kerberos principal名。可以在Kafka的JAAS配置文件或在Kafka的配置文件中定义。 | string | null |  | medium |
| sasl.mechanism | 用于客户端连接的SASL机制。安全提供者可用的机制。GSSAPI是默认机制。 | string | GSSAPI |  | medium |
| security.protocol | 用于与broker通讯的协议。 有效值为：PLAINTEXT，SSL，SASL\_PLAINTEXT，SASL\_SSL。 | string | PLAINTEXT |  | medium |
| send.buffer.bytes | 发送数据时要使用的TCP发送缓冲区（SO\_SNDBUF）的大小。 如果值为-1，则将使用OS默认值。 | int | 131072 | [-1,...] | medium |
| ssl.enabled.protocols | 启用SSL连接的协议列表。 | list | TLSv1.2,TLSv1.1,TLSv1 |  | medium |
| ssl.keystore.type | key仓库文件的文件格式，客户端可选。 | string | JKS |  | medium |
| ssl.protocol | 用于生成SSLContext的SSL协议。 默认设置是TLS，这对大多数情况都是适用的。 最新的JVM中的允许值为TLS，TLSv1.1和TLSv1.2。 较旧的JVM可能支持SSL，SSLv2和SSLv3，但由于已知的安全漏洞，不建议使用SSL。 | string | TLS |  | medium |
| ssl.provider | 用于SSL连接的安全提供程序的名称。 默认值是JVM的默认安全提供程序。 | string | null |  | medium |
| ssl.truststore.type | 信任存储文件的文件格式。 | string | JKS |  | medium |
| auto.commit.interval.ms | 如果enable.auto.commit设置为true，则消费者偏移量自动提交给Kafka的频率（以毫秒为单位）。 | int | 5000 | [0,...] | low |
| check.crcs | 自动检查CRC32记录的消耗。 这样可以确保消息发生时不会在线或磁盘损坏。 此检查增加了一些开销，因此在寻求极致性能的情况下可能会被禁用。 | boolean | true |  | low |
| client.id | 在发出请求时传递给服务器的id字符串。 这样做的目的是通过允许将逻辑应用程序名称包含在服务器端请求日志记录中，来跟踪ip/port的请求源。 | string | "" |  | low |
| fetch.max.wait.ms | 如果没有足够的数据满足fetch.min.bytes，服务器将在接收到提取请求之前阻止的最大时间。 | int | 500 | [0,...] | low |
| interceptor.classes | 用作拦截器的类的列表。 你可实现ConsumerInterceptor接口以允许拦截（也可能变化）消费者接收的记录。 默认情况下，没有拦截器。 | list | null |  | low |
| metadata.max.age.ms | 在一定时间段之后（以毫秒为单位的），强制更新元数据，即使没有任何分区领导变化，任何新的broker或分区。 | long | 300000 | [0,...] | low |
| metric.reporters | 用作度量记录员类的列表。实现MetricReporter接口以允许插入通知新的度量创建的类。JmxReporter始终包含在注册JMX统计信息中。 | list | "" |  | low |
| metrics.num.samples | 保持的样本数以计算度量。 | int | 2 | [1,...] | low |
| metrics.recording.level | 最高的记录级别。 | 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.ms | 尝试重新连接指定主机之前等待的时间，避免频繁的连接主机，这种机制适用于消费者向broker发送的所有请求。 | long | 50 | [0,...] | low |
| retry.backoff.ms | 尝试重新发送失败的请求到指定topic分区之前的等待时间。避免在某些故障情况下，频繁的重复发送。 | long | 100 | [0,...] | low |
| sasl.kerberos.kinit.cmd Kerberos | kinit命令路径。 | string | /usr/bin/kinit |  | low |
| sasl.kerberos.min.time.before.relogin | 尝试/恢复之间的登录线程的休眠时间。 | long | 60000 |  | low |
| sasl.kerberos.ticket.renew.jitter | 添加到更新时间的随机抖动百分比。 | double | 0.05 |  | low |
| sasl.kerberos.ticket.renew.window.factor | 登录线程将休眠，直到从上次刷新到ticket的指定的时间窗口因子到期，此时将尝试续订ticket。 | double | 0.8 |  | low |
| ssl.cipher.suites | 密码套件列表，用于TLS或SSL网络协议的安全设置，认证，加密，MAC和密钥交换算法的明明组合。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端点识别算法。 | string | null |  | low |
| ssl.keymanager.algorithm | 密钥管理器工厂用于SSL连接的算法。 默认值是为Java虚拟机配置的密钥管理器工厂算法。 | string | SunX509 |  | low |
| ssl.secure.random.implementation | 用于SSL加密操作的SecureRandom PRNG实现。 | string | null |  | low |
| ssl.trustmanager.algorithm | 信任管理器工厂用于SSL连接的算法。 默认值是为Java虚拟机配置的信任管理器工厂算法。 | string | PKIX |  | low |

## 3.4.2 旧消费者配置

旧消费者配置如下：

* group.id
* zookeeper.connect

| **PROPERTY** | **DEFAULT** | **DESCRIPTION** |
| --- | --- | --- |
| group.id |  | 标识消费者所属消费者组(独一的)。通过设置相同的组ID，多个消费者表明属于该消费者组的一部分。 |
| zookeeper.connect |  | 指定ZooKeeper连接字符串，格式为hostname：port，其中host和port是ZooKeeper服务器的主机和端口。 为了使ZooKeeper宕机时连接到其他ZooKeeper节点，你还可以以hostname1:host1，hostname2:port2，hostname3:port3的形式指定多个主机。 还可以设置ZooKeeper chroot路径，作为其ZooKeeper连接字符串的一部分，将其数据放置在全局ZooKeeper命名空间中的某个路径下。 如果是这样，消费者应该在其连接字符串中使用相同的chroot路径。 例如，要给出/chroot/path的chroot路径，你需要将该值设置为：hostname1:port1,hostname2:port2,hostname3:port3/chroot/path。 |
| consumer.id | null | 如果未设置将自动生成。 |
| socket.timeout.ms | 30 \* 1000 | 网络请求socker的超时时间。实际的超时是 max.fetch.wait+socket.timeout.ms的时间。 |
| socket.receive.buffer.bytes | 64 \* 1024 | 网络请求socker的接收缓存大小 |
| fetch.message.max.bytes | 1024 \* 1024 | 每个拉取请求的每个topic分区尝试获取的消息的字节大小。这些字节将被读入每个分区的内存，因此这有助于控制消费者使用的内存。 拉取请求的大小至少与服务器允许的最大消息的大小一样大，否则生产者可能发送大于消费者可以拉取的消息。 |
| num.consumer.fetchers | 1 | 用于拉取数据的拉取线程数。 |
| auto.commit.enable | true | 如果为true，请定期向ZooKeeper提交消费者已经获取的消息的偏移量。 当进程失败时，将使用这种承诺偏移量作为新消费者开始的位置。 |
| auto.commit.interval.ms | 60 \* 1000 | 消费者offset提交到zookeeper的频率（以毫秒为单位） |
| queued.max.message.chunks | 2 | 消费缓存消息块的最大大小。每个块可以达到fetch.message.max.bytes。 |
| rebalance.max.retries | 4 | 当新的消费者加入消费者组时，消费者集合尝试“重新平衡”负载，并为每个消费者分配分区。如果消费者集合在分配时发生时发生变化，则重新平衡将失败并重试。此设置控制尝试之前的最大尝试次数。 |
| fetch.min.bytes | 1 | 拉取请求返回最小的数据量。如果没有足够的数据，请求将等待数据积累，然后应答请求。 |
| fetch.wait.max.ms | 100 | 如果没有足够的数据（fetch.min.bytes），服务器将在返回请求数据之前阻塞的最长时间。 |
| rebalance.backoff.ms | 2000 | 重新平衡时重试之间的回退时间。如果未设置，则使用zookeeper.sync.time.ms中的值。 |
| refresh.leader.backoff.ms | 200 | 回退时间等待，然后再尝试选举一个刚刚失去leader的分区。 |
| auto.offset.reset | largest | 如果ZooKeeper中没有初始偏移量，或偏移值超出范围，该怎么办？ 最小：自动将偏移重置为最小偏移最大：自动将偏移重置为最大偏移 \* 其他任何事情：抛出异常消费者 |
| consumer.timeout.ms | -1 | 如果在指定的时间间隔后没有消息可用，则向用户发出超时异常 |
| exclude.internal.topics | true | 来自内部topic的消息（如偏移量）是否应该暴露给消费者。 |
| client.id | group id value | 客户端ID是每个请求中发送的用户指定的字符串，用于帮助跟踪调用。 它应该逻辑地标识发出请求的应用程序。 |
| zookeeper.session.timeout.ms | 6000 | ZooKeeper会话超时。如果消费者在这段时间内没有对ZooKeeper心跳，那么它被认为是死亡的，并且会发生重新平衡。 |
| zookeeper.connection.timeout.ms | 6000 | 与zookeeper建立连接时客户端等待的最长时间。 |
| zookeeper.sync.time.ms | 2000 | ZK follower可以罗ZK leader多久 |
| offsets.storage | zookeeper | 选择存储偏移量的位置（zookeeper或kafka）。 |
| offsets.channel.backoff.ms | 1000 | 重新连接offset通道或重试失败的偏移提取/提交请求时的回退周期。 |
| offsets.channel.socket.timeout.ms | 10000 | 读取offset拉取/提交响应的Socker的超时时间。此超时也用于查询offset manager的ConsumerMetadata请求。 |
| offsets.commit.max.retries | 5 | 失败时重试偏移提交的最大次数。此重试计数仅适用于停机期间的offset提交，它不适用于自动提交线程的提交。它也不适用于在提交offset之前查询偏移协调器的尝试。即如果消费者元数据请求由于任何原因而失败，则将重试它，并且重试不计入该限制。 |
| dual.commit.enabled | true | 如果使用“kafka”作为offsets.storage，则可以向ZooKeeper（除Kafka之外）进行双重提交offset。在从基于zookeeper的offset存储迁移到kafka存储的时候可以这么做。对于任何给定的消费者组，在该组中的所有实例已迁移到提交到broker（而不是直接到ZooKeeper）的新的版本之后，可以关闭这个。 |
| partition.assignment.strategy | range | 在“range”或“roundrobin”策略之间选择将分区分配给消费者流。  循环分区分配器分配所有可用的分区和所有可用的消费者线程。然后，继续从分区到消费者线程进行循环任务。如果所有消费者实例的订阅是相同的，则分区将被均匀分布。（即，分区所有权计数将在所有消费者线程之间的差异仅在一个delta之内。）循环分配仅在以下情况下被允许：（a）每个主题在消费者实例中具有相同数量的流（b）订阅的topic的对于组内的每个消费者实例都是相同的。  范围(Range)分区基于每个topic。对于每个主题，我们按数字顺序排列可用的分区，并以字典顺序排列消费者线程。然后，我们将分区数除以消费者流（线程）的总数来确定分配给每个消费者的分区数。如果不均匀分割，那么前几个消费者将会有多的分区。 |

有关消费者配置的更多详细信息，请参见scaf类kafka.consumer.ConsumerConfig。

**Kafka Streams配置**

## 3.6 Kafka Streams配置

Kafka Stream客户端库配置（注意，窗口可拖动）。

| **NAME** | **DESCRIPTION** | **TYPE** | **DEFAULT** | **VALID VALUES** | **IMPORTANCE** |
| --- | --- | --- | --- | --- | --- |
| application.id | 流处理应用程序标识。必须在Kafka集群中是独一无二的。 1）默认客户端ID前缀，2）成员资格管理的group-id，3）changgelog的topic前缀 | string |  |  | high |
| bootstrap.servers | 用于建立与Kafka集群的初始连接的主机/端口列表。 客户端将会连接所有服务器，跟指定哪些服务器无关 - 通过指定的服务器列表会自动发现全部的服务器。此列表格式host1：port1，host2：port2，...由于这些服务器仅用于初始连接以发现完整的集群成员（可能会动态更改），所以此列表不需要包含完整集 的服务器（您可能需要多个服务器，以防指定的服务器关闭）。 | list |  |  | high |
| replication.factor | 流处理程序创建更改日志topic和重新分配topic的副本数 | int | 1 |  | high |
| state.dir | 状态存储的目录地址。 | string | /tmp/kafka-streams |  | high |
| cache.max.bytes.buffering | 用于缓冲所有线程的最大内存字节数 | long | 10485760 | [0,...] | low |
| client.id | 发出请求时传递给服务器的id字符串。 这样做的目的是通过允许将逻辑应用程序名称包含在服务器端请求日志记录中，来追踪请求源的ip/port。 | string | "" |  | high |
| default.key.serde | 用于实现Serde接口的key的默认序列化器/解串器类。 | class | org.apache.kafka.common.serialization.Serdes$ByteArraySerde |  | medium |
| default.timestamp.extractor | 实现TimestampExtractor接口的默认时间戳提取器类。 | class | org.apache.kafka.streams.processor.FailOnInvalidTimestamp |  | medium |
| default.value.serde | 用于实现Serde接口的值的默认serializer / deserializer类。 | class | org.apache.kafka.common.serialization.Serdes$ByteArraySerde |  | medium |
| num.standby.replicas | 每个任务的备用副本数。 | int | 0 |  | low |
| num.stream.threads | 执行流处理的线程数。 | int | 1 |  | low |
| processing.guarantee | 应使用的加工保证。可能的值为at\_least\_once（默认）和exact\_once。 | string | at\_least\_once | [at\_least\_once, exactly\_once] | medium |
| security.protocol | 用于与broker沟通的协议。 有效值为：PLAINTEXT，SSL，SASL\_PLAINTEXT，SASL\_SSL。 | string | PLAINTEXT |  | medium |
| application.server | host:port指向用户嵌入定义的末端，可用于发现单个KafkaStreams应用程序中状态存储的位置 | string | "" |  | low |
| buffered.records.per.partition | 每个分区缓存的最大记录数。 | int | 1000 |  | low |
| commit.interval.ms | 用于保存process位置的频率。 注意，如果'processing.guarantee'设置为'exact\_once'，默认值为100，否则默认值为30000。 | long | 30000 |  | low |
| connections.max.idle.ms | 关闭闲置的连接时间（以毫秒为单位）。 | long | 540000 |  | medium |
| key.serde | 用于实现Serde接口的key的Serializer/deserializer类.此配置已被弃用，请改用default.key.serde | class | null |  | low |
| metadata.max.age.ms | 即使我们没有看到任何分区leader发生变化，主动发现新的broker或分区，强制更新元数据时间（以毫秒为单位）。 | long | 300000 | [0,...] | low |
| metric.reporters | metric reporter的类列表。实现MetricReporter接口，JmxReporter始终包含在注册JMX统计信息中。 | list | "" |  | low |
| metrics.num.samples | 保持的样本数以计算度量。 | int | 2 | [1,...] | low |
| metrics.recording.level | 日志级别。 | string | INFO | [INFO, DEBUG] | low |
| metrics.sample.window.ms | 时间窗口计算度量标准。 | long | 30000 | [0,...] | low |
| partition.grouper | 实现PartitionGrouper接口的Partition grouper类。 | class | org.apache .kafka.streams .processor .DefaultPartitionGrouper |  | medium |
| poll.ms | 阻塞输入等待的时间（以毫秒为单位）。 | long | 100 |  | low |
| receive.buffer.bytes | 读取数据时使用的TCP接收缓冲区（SO\_RCVBUF）的大小。 如果值为-1，则将使用OS默认值。 | int | 32768 | [0,...] | medium |
| reconnect.backoff.max.ms | 因故障无法重新连接broker，重新连接的等待的最大时间（毫秒）。如果提供，每个主机会连续增加，直到达到最大值。随机递增20％的随机抖动以避免连接风暴。 | long | 1000 | [0,...] | low |
| reconnect.backoff.ms | 尝试重新连接之前等待的时间。避免在高频繁的重复连接服务器。 这种backoff适用于消费者向broker发送的所有请求。 | long | 50 | [0,...] | low |
| request.timeout.ms | 控制客户端等待请求响应的最长时间。如果在配置时间内未收到响应，客户端将在需要时重新发送请求，如果重试耗尽，则请求失败。 | int | 40000 | [0,...] | low |
| retry.backoff.ms | 尝试重试失败请求之前等待的时间。以避免了在某些故障情况下，在频繁重复发送请求。 | long | 100 | [0,...] | low |
| rocksdb.config.setter | 一个Rocks DB配置setter类，或实现RocksDBConfigSetter接口的类名 | null |  |  | low |
| send.buffer.bytes | 发送数据时要使用的TCP发送缓冲区（SO\_SNDBUF）的大小。 如果值为-1，则将使用OS默认值。 | int | 131072 | [0,...] | low |
| state.cleanup.delay.ms | 在分区迁移删除状态之前等待的时间（毫秒）。 | long | 60000 |  | low |
| timestamp.extractor | 实现TimestampExtractor接口的Timestamp抽取器类。此配置已弃用，请改用default.timestamp.extractor | class | null |  | low |
| windowstore.changelog.additional.retention.ms | 添加到Windows维护管理器以确保数据不会从日志中过早删除。默认为1天 | long | 86400000 |  | low |
| zookeeper.connect | Zookeeper连接字符串，用于Kafka主题管理。此配置已被弃用，将被忽略，因为Streams API不再使用Zookeeper。 | string | "" |  |  |

**Kafka Connect配置**

## 3.5 Kafka Connect Configs

Kafka Connect框架的相关配置（注意，窗口可向右拉动）。

| **NAME** | **DESCRIPTION** | **TYPE** | **DEFAULT** | **VALID VALUES** | **IMPORTANCE** |
| --- | --- | --- | --- | --- | --- |
| config.storage.topic | kafka topic仓库配置 | string |  |  | high |
| group.id | 唯一的字符串，用于标识此worker所属的Connect集群组。 | string |  |  | high |
| key.converter | 用于Kafka Connect和写入到Kafka的序列化消息的之间格式转换的转换器类。 这可以控制写入或从kafka读取的消息中的键的格式，并且由于这与连接器无关，因此它允许任何连接器使用任何序列化格式。 常见格式的示例包括JSON和Avro。 | class |  |  | high |
| offset.storage.topic | 连接器的offset存储到哪个topic中 | string |  |  | high |
| status.storage.topic | 追踪连接器和任务状态存储到哪个topic中 | string |  |  | high |
| value.converter | 用于Kafka Connect格式和写入Kafka的序列化格式之间转换的转换器类。 控制了写入或从Kafka读取的消息中的值的格式，并且由于这与连接器无关，因此它允许任何连接器使用任何序列化格式。 常见格式的示例包括JSON和Avro。 | class |  |  | high |
| internal.key.converter | 用于在Kafka Connect格式和写入Kafka的序列化格式之间转换的转换器类。 这可以控制写入或从Kafka读取的消息中的key的格式，并且由于这与连接器无关，因此它允许任何连接器使用任何序列化格式。 常见格式的示例包括JSON和Avro。 此设置用于控制框架内部使用的记账数据的格式，例如配置和偏移量，因此用户可以使用运行各种Converter实现。 | class |  |  | low |
| internal.value.converter | 用于在Kafka Connect格式和写入Kafka的序列化格式之间转换的转换器类。 这控制了写入或从Kafka读取的消息中的值的格式，并且由于这与连接器无关，因此它允许任何连接器使用任何序列化格式。 常见格式的示例包括JSON和Avro。 此设置用于控制框架内部使用的记账数据的格式，例如配置和偏移量，因此用户可以使用运行各种Converter实现。 | class |  |  | low |
| bootstrap.servers | 用于建立与Kafka集群的初始连接的主机/端口列表。此列表用来发现完整服务器集的初始主机。 该列表的格式应为host1：port1，host2：port2，....由于这些服务器仅用于初始连接以发现完整的集群成员资格（可能会动态更改），因此,不需要包含完整的服务器（尽管如此，你需要多配置几个，以防止配置的宕机）。 | list | localhost:9092 |  | high |
| heartbeat.interval.ms | 心跳间隔时间。心跳用于确保会话保持活动，并在新成员加入或离开组时进行重新平衡。 该值必须设置为低于session.timeout.ms，但通常应设置为不高于该值的1/3。 | int | 3000 |  | high |
| rebalance.timeout.ms | 限制所有组中消费者的任务处理数据和提交offset所需的时间。如果超时，那么woker将从组中删除，这也将导致offset提交失败。 | int | 60000 |  | high |
| session.timeout.ms | 用于察觉worker故障的超时时间。worker定时发送心跳以表明自己是活着的。如果broker在会话超时时间到期之前没有接收到心跳，那么broker将从分组中移除该worker，并启动重新平衡。注意，该值必须在group.min.session.timeout.ms和group.max.session.timeout.ms范围内。 | int | 10000 |  | high |
| ssl.key.password | 密钥存储文件中私钥的密码。 这对于客户端是可选的。 | password | null |  | high |
| ssl.keystore.location | 密钥存储文件的位置。 这对于客户端是可选的，可以用于客户端的双向身份验证。 | string | null |  | high |
| ssl.keystore.password | 密钥存储文件的存储密码。 客户端是可选的，只有配置了ssl.keystore.location才需要。 | password | null |  | high |
| ssl.truststore.location | 信任存储文件的位置。 | string | null |  | high |
| ssl.truststore.password | 信任存储文件的密码。 | password | null |  | high |
| connections.max.idle.ms | 多少毫秒之后关闭空闲的连接。 | long | 540000 |  | medium |
| receive.buffer.bytes | 读取数据时使用的TCP接收缓冲区（SO\_RCVBUF）的大小。 如果值为-1，则将使用OS默认值。 | int | 32768 | [0,...] | medium |
| request.timeout.ms | 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，客户端将在必要时重新发送请求，如果重试耗尽，则该请求将失败。 | int | 40000 | [0,...] | medium |
| sasl.jaas.config | 用于JAAS配置文件的SASL连接的JAAS登录上下文参数格式。[这里](#http://docs.oracle.com/javase/8/docs/te)描述了JAAS配置文件的格式。该值的格式为：' (=)\*;' | password | null |  | medium |
| sasl.kerberos.service.name | Kafka运行的Kerberos principal名称。 可以在Kafka的JAAS配置或Kafka的配置中定义。 | string | null |  | medium |
| sasl.mechanism | 用户客户端连接的SASL机制。可以提供者任何安全机制。 GSSAPI是默认机制。 | string | GSSAPI |  | medium |
| security.protocol | 用于和broker通讯的策略。有效的值有：PLAINTEXT, SSL, SASL\_PLAINTEXT, SASL\_SSL。 |  | string | PLAINTEXT | medium |
| send.buffer.bytes | 发送数据时使用TCP发送缓冲区（SO\_SNDBUF）的大小。如果值为-1，则将使用OS默认。 | int | 131072 | [-1,...] | medium |
| ssl.enabled.protocols | 启用SSL连接的协议列表。 | list | TLSv1.2,TLSv1 .1,TLSv1 |  | medium |
| ssl.keystore.type | 密钥存储文件的文件格式。 对于客户端是可选的。 | string | JKS |  | medium |
| ssl.protocol | 用于生成SSLContext的SSL协议。 默认设置是TLS，这对大多数情况都是适用的。 最新的JVM中的允许值为TLS，TLSv1.1和TLSv1.2。 旧的JVM可能支持SSL，SSLv2和SSLv3，但由于已知的安全漏洞，不建议使用SSL。 | string | TLS |  | medium |
| ssl.provider | 用于SSL连接的安全提供程序的名称。 默认值是JVM的默认安全提供程序。 | string | null |  | medium |
| ssl.truststore.type | 信任存储文件的文件格式。 | string | JKS |  | medium |
| worker.sync.timeout.ms | 当worker与其他worker不同步并需要重新同步配置时，需等待一段时间才能离开组，然后才能重新加入。 | int | 3000 |  | medium |
| worker.unsync.backoff.ms | 当worker与其他worker不同步，并且无法在worker.sync.timeout.ms 期间追赶上，在重新连接之前，退出Connect集群的时间。 | int | 300000 |  | medium |
| access.control.allow.methods | 通过设置Access-Control-Allow-Methods标头来设置跨源请求支持的方法。 Access-Control-Allow-Methods标头的默认值允许GET，POST和HEAD的跨源请求。 | string | "" |  | low |
| access.control.allow.origin | 将Access-Control-Allow-Origin标头设置为REST API请求。要启用跨源访问，请将其设置为应该允许访问API的应用程序的域，或者 \*" 以允许从任何的域。 默认值只允许从REST API的域访问。 | string | "" |  | low |
| client.id | 在发出请求时传递给服务器的id字符串。这样做的目的是通过允许逻辑应用程序名称包含在请求消息中，来跟踪请求来源。而不仅仅是ip/port | string | "" |  | low |
| config.storage.replication.factor | 当创建配置仓库topic时的副本数 | short | 3 | [1,...] | low |
| metadata.max.age.ms | 在没有任何分区leader改变，主动地发现新的broker或分区的时间。 | long | 300000 | [0,...] | low |
| metric.reporters | A list of classes to use as metrics reporters. Implementing the MetricReporter 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 | 保留计算metrics的样本数（译者不清楚是做什么的） | int | 2 | [1,...] | low |
| metrics.sample.window.ms | The window of time a metrics sample is computed over. | long | 30000 | [0,...] | low |
| offset.flush.interval.ms | 尝试提交任务偏移量的间隔。 | long | 60000 |  | low |
| offset.flush.timeout.ms | 在取消进程并恢复要在之后尝试提交的offset数据之前，等待消息刷新并分配要提交到offset仓库的offset数据的最大毫秒数。 | long | 5000 |  | low |
| offset.storage.partitions | 创建offset仓库topic的分区数 | int | 25 | [1,...] | low |
| offset.storage.replication.factor | 创建offset仓库topic的副本数 | short | 3 | [1,...] | low |
| plugin.path | 包含插件(连接器,转换器,转换)逗号(,)分隔的路径列表。该列表应包含顶级目录，其中包括以下任何组合：a）包含jars与插件及其依赖关系的目录 b）具有插件及其依赖项的uber-jars c）包含插件类的包目录结构的目录及其依赖关系,注意：将遵循符号链接来发现依赖关系或插件。 示例：plugin.path=/usr/local/share/java,/usr/local/share/kafka/plugins,/opt/connectors | list | null |  | low |
| reconnect.backoff.max.ms | 无法连接broker时等待的最大时间（毫秒）。如果设置，则每个host的将会持续的增加，直到达到最大值。计算增加后，再增加20％的随机抖动，以避免高频的反复连接。 | long | 1000 | [0,...] | low |
| reconnect.backoff.ms | 尝试重新连接到主机之前等待的时间。 避免了高频率反复的连接主机。 这种机制适用于消费者向broker发送的所有请求。 | long | 50 | [0,...] | low |
| rest.advertised.host.name | 如果设置，其他wokers将通过这个hostname进行连接。 | string | null |  | low |
| rest.advertised.port | 如果设置，其他的worker将通过这个端口进行连接。 | int | null |  | low |
| rest.host.name | REST API的主机名。 如果设置，它将只绑定到这个接口。 | string | null |  | low |
| rest.port | 用于监听REST API的端口 | int | 8083 | low |  |
| retry.backoff.ms | 失败请求重新尝试之前的等待时间，避免了在某些故障的情况下，频繁的重复发送请求。 | long | 100 | [0,...] | low |
| sasl.kerberos.kinit.cmd | Kerberos kinit命令路径. | string | /usr/bin/kinit |  | low |
| sasl.kerberos.min.time.before.relogin | 尝试refresh之间登录线程的休眠时间. | long | 60000 |  | low |
| sasl.kerberos.ticket.renew.jitter | 添加到更新时间的随机抖动百分比。 | double | 0.05 |  | low |
| sasl.kerberos.ticket.renew.window.factor | 登录线程将休眠，直到从上次刷新ticket到期，此时将尝试续订ticket。 | double | 0.8 |  | low |
| ssl.cipher.suites | 密码套件列表。用于TLS或SSL网络协议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，支持所有可用的密码套件。 | list | null |  | low |
| ssl.endpoint.identification.algorithm | 末端识别算法使用服务器证书验证服务器主机名。 | string | null |  | low |
| ssl.keymanager.algorithm | 用于SSL连接的key管理工厂的算法，默认值是Java虚拟机配置的密钥管理工厂算法。 | 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 | 用于SSL连接的信任管理仓库算法。默认值是Java虚拟机配置的信任管理器工厂算法。 | string | PKIX |  | low |
| status.storage.partitions | 用于创建状态仓库topic的分区数 | int | 5 | [1,...] | low |
| status.storage.replication.factor | 用于创建状态仓库topic的副本数 | short | 3 | [1,...] | low |
| task.shutdown.graceful.timeout.ms | 等待任务正常关闭的时间，这是总时间，不是每个任务，所有任务触发关闭，然后依次等待。 |  |  |  |  |

**Kafka AdminClient配置**

## 3.7 AdminClient Configs

以下是Kafka Admin客户端库的配置。

| **NAME** | **DESCRIPTION** | **TYPE** | **DEFAULT** | **VALID VALUES** | **IMPORTANCE** |
| --- | --- | --- | --- | --- | --- |
| bootstrap.servers | host/port,用于和kafka集群建立初始化连接。因为这些服务器地址仅用于初始化连接，并通过现有配置的来发现全部的kafka集群成员（集群随时会变化），所以此列表不需要包含完整的集群地址（但尽量多配置几个，以防止配置的服务器宕机）。 | list |  |  | high |
| ssl.key.password | 密钥仓库文件中的私钥密码。对于客户端是可选的。 | password | null |  | high |
| ssl.keystore.location | 密钥仓库文件的位置。这对于客户端是可选的，可以用于客户端的双向认证。 | string | null |  | high |
| ssl.keystore.password | 密钥仓库文件的仓库密钥。这对于客户端是可选的，只有配置了ssl.keystore.location才需要。 | password | null |  | high |
| ssl.truststore.location | 信任存储文件的位置。 | string | null |  | high |
| ssl.truststore.password | 信任存储文件的密码。如果未设置密码，对信任库的访问仍然可用，但是完整性检查将被禁用。 | password | null |  | high |
| client.id | 在发出请求时传递给服务器的id字符串。这样做的目的是通过允许在服务器端请求日志记录中包含逻辑应用程序名称来跟踪请求源的ip/port。 | string | "" |  | medium |
| connections.max.idle.ms | 关闭闲置连接的时间。 | long | 300000 |  | medium |
| receive.buffer.bytes | 读取数据时使用的TCP接收缓冲区（SO\_RCVBUF）的大小。如果值为-1，则将使用OS默认值。 | int | 65536 | [-1,...] | medium |
| request.timeout.ms | 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，客户端将在必要时重新发送请求，如果重试耗尽，则该请求将失败。 | int | 120000 | [0,...] | medium |
| sasl.jaas.config | JAAS配置文件使用的格式的SASL连接的JAAS登录上下文参数。这里描述JAAS配置文件格式。该值的格式为：' (=)\*;' | password | null |  | medium |
| sasl.kerberos.service.name | Kafka运行的Kerberos principal名。可以在Kafka的JAAS配置或Kafka的配置中定义。 | string | null |  | medium |
| sasl.mechanism | 用于客户端连接的SASL机制。安全提供者可用的任何机制。GSSAPI是默认机制。 | string | GSSAPI |  | medium |
| security.protocol | 与broker通讯的协议。有效的值有: PLAINTEXT, SSL, SASL\_PLAINTEXT,SASL\_SSL. | string | PLAINTEXT |  | medium |
| send.buffer.bytes | 发送数据时时使用TCP发送缓冲区（SO\_SNDBUF）的大小。如果值为-1，则使用OS默认值。 | int | 131072 | [-1,...] | medium |
| ssl.enabled.protocols | 启用SSL连接的协议列表。 | list | TLSv1.2,TLSv1.1,TLSv1 |  | medium |
| ssl.keystore.type | 密钥仓库文件的文件格式。对于客户端是可选的。 | string | JKS |  | medium |
| ssl.protocol | 用于生成SSLContext的SSL协议。默认设置是TLS，这对大多数情况都是适用的。最新的JVM中允许的值是TLS,TLSv1.1和TLSv1.2。较旧的JVM可能支持SSL,SSLv2和SSLv3,但由于已知的安全漏洞,不建议使用。 | string | TLS |  | medium |
| ssl.provider | 用于SSL连接的安全提供程序的名称。 默认值是JVM的默认安全提供程序。 | string | null |  | medium |
| ssl.truststore.type | 信任仓库文件的文件格式 | string | JKS |  | medium |
| metadata.max.age.ms | 我们强制更新元数据的时间段（以毫秒为单位），即使我们没有任何分区leader发生变化，主动发现任何新的broker或分区。 | long | 300000 | [0,...] | low |
| metric.reporters | 用作指标记录的类的列表。实现MetricReporter接口，以允许插入将被通知新的度量创建的类。JmxReporter始终包含在注册JMX统计信息中。 | list | "" |  | low |
| metrics.num.samples | 用于计算度量维护的样例数。 | int | 2 | [1,...] | low |
| metrics.recording.level | The highest recording level for metrics. | string | INFO | [INFO, DEBUG] | low |
| metrics.sample.window.ms | 时间窗口计算度量标准。 | long | 30000 | [0,...] | low |
| reconnect.backoff.max.ms | 重新连接到重复无法连接的broker程序时等待的最大时间（毫秒）。如果提供，每个主机的回退将会连续增加，直到达到最大值。 计算后退增加后，增加20％的随机抖动以避免连接风暴。 | long | 1000 | [0,...] | low |
| reconnect.backoff.ms | 尝试重新连接到给定主机之前等待的基本时间量。这避免了在频繁的重复连接主机。此配置适用于client对broker的所有连接尝试。 | long | 50 | [0,...] | low |
| retries | 在失败之前重试调用的最大次数 | int | 5 | [0,...] | low |
| retry.backoff.ms | 尝试重试失败的请求之前等待的时间。这样可以避免在某些故障情况下以频繁的重复发送请求。 | long | 100 | [0,...] | low |
| sasl.kerberos.kinit.cmd | Kerberos kinit命令路径。 | string | /usr/bin/kinit |  | low |
| sasl.kerberos.min.time.before.relogin | 刷新尝试之间的登录线程睡眠时间。 | long | 60000 |  | low |
| sasl.kerberos.ticket.renew.jitter | 添加到更新时间的随机抖动百分比。 | double | 0.05 |  | low |
| sasl.kerberos.ticket.renew.window.factor | 登录线程将休眠，直到从上次刷新到“票”到期时间的指定窗口为止，此时将尝试续订“票”。 |  |  |  |  |
|  | double | 0.8 |  | low |  |
| ssl.cipher.suites | 密码套件列表。是TLS或SSL网络协议来协商用于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，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端点识别算法。 | string | null |  | low |
| ssl.keymanager.algorithm | 用于SSL连接的密钥管理工厂算法。默认值是Java虚拟机配置的密钥管理器工厂算法。 | 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 | 用于SSL连接的信任管理工厂算法，默认是Java虚拟机机制。 | string | PKIX |  |  |

**kafka设计动机**

We designed Kafka to be able to act as a unified platform for handling all the real-time data feeds a large company might have. To do this we had to think through a fairly broad set of use cases.

我们设计kafka是必须能够作为一个统一的平台，来处理一家大公司可能有的所有实时数据。要做到这一点，我们不得不思考一个相当广泛的用例。

It would have to have high-throughput to support high volume event streams such as real-time log aggregation.

它必须具有高吞吐量，以支持大容量的事件流，如实时日志聚集。

It would need to deal gracefully with large data backlogs to be able to support periodic data loads from offline systems.

它将需要优雅地处理大型数据积压，要能够支持从脱机系统的定期数据装载。

It also meant the system would have to handle low-latency delivery to handle more traditional messaging use-cases.

这也意味着该系统将不得不处理低延迟交付，以处理更传统的消息传递的用例。

We wanted to support partitioned, distributed, real-time processing of these feeds to create new, derived feeds. This motivated our partitioning and consumer model.

我们想要支持以分区，分布式的，实时处理来这些创建新派生的feeds，促使我们的分区和消费模式。

Finally in cases where the stream is fed into other data systems for serving we new the system would have to be able to guarantee fault-tolerance in the presence of machine failures.

最后，在其中该流被送入其他数据系统用于服务的情况下，我们的新的系统必须能够保证在机器故障存在的容错。

Supporting these uses led use to a design with a number of unique elements, more akin to a database log then a traditional messaging system. We will outline some elements of the design in the following sections.

为了支持这些用途，需要设计一些独特的元素。比起传统的消息系统，更像数据库日志，我们将在下一节介绍。

**kafka持久化**

### 不要害怕文件系统!

Kafka relies heavily on the filesystem for storing and caching messages. There is a general perception that "disks are slow" which makes people skeptical that a persistent structure can offer competitive performance. In fact disks are both much slower and much faster than people expect depending on how they are used; and a properly designed disk structure can often be as fast as the network.

Kafka高度依赖文件系统来存储和缓存消息。一般的人都认为“磁盘是缓慢的”，这使得人们对“持久化结构”持有怀疑态度。实际上，磁盘比人们预想的快很多也慢很多，这取决于它们如何被使用；一个好的磁盘结构设计可以使之跟网络速度一样快。

The key fact about disk performance is that the throughput of hard drives has been diverging from the latency of a disk seek for the last decade. As a result the performance of linear writes on a [JBOD](http://en.wikipedia.org/wiki/Non-RAID_drive_architectures) configuration with six 7200rpm SATA RAID-5 array is about 600MB/sec but the performance of random writes is only about 100k/sec—a difference of over 6000X. These linear reads and writes are the most predictable of all usage patterns, and are heavily optimized by the operating system. A modern operating system provides read-ahead and write-behind techniques that prefetch data in large block multiples and group smaller logical writes into large physical writes. A further discussion of this issue can be found in this ACM Queue article; they actually find that sequential disk access can in some cases be faster than random memory access!

一个有关磁盘性能的关键事实是：在过去的十年，磁盘驱动器的吞吐量跟磁盘寻道的延迟是相背离的。结果就是：JBOD配置的6个7200rpm SATA RAID-5 的磁盘阵列上线性写的速度大概是600M/秒，但是随机写的速度只有100K/秒，两者相差将近6000倍。线性读写在大多数应用场景下是可以预测的，因此，现代的操作系统提供了预读和写技术，将多个大块预取数据，并将较小的写入组合成一个大的物理写。更多的讨论可以在ACM Queue Artical中找到，他们发现，对磁盘的线性读在有些情况下可以比内存的随机访问要更快。

To compensate for this performance divergence modern operating systems have become increasingly aggressive in their use of main memory for disk caching. A modern OS will happily divert *all* free memory to disk caching with little performance penalty when the memory is reclaimed. All disk reads and writes will go through this unified cache. This feature cannot easily be turned off without using direct I/O, so even if a process maintains an in-process cache of the data, this data will likely be duplicated in OS pagecache, effectively storing everything twice.

为了补偿这个性能上的差异，现代操作系统用内存做磁盘缓存时变得越来越重。现在操作系统很乐意将所有空闲的内存用于磁盘缓冲，尽管在内存回收的时候会有一点性能上的代价。所有的磁盘读写会通过这个统一的缓存。没有使用直接I/O的情况下，不能轻易关闭此功能。所以即使 一个进程维护着一个进程内的数据缓存，这些数据还是会在OS的页缓存中被复制，从而有效地存储两次。

Furthermore we are building on top of the JVM, and anyone who has spent any time with Java memory usage knows two things:

此外，我们建立在JVM的顶部，熟悉java内存应用管理的人应该清楚以下两件事情：

1. The memory overhead of objects is very high, often doubling the size of the data stored (or worse).  
   一个对象的内存消耗是非常高的，经常是所存数据的两倍（或者更多）。
2. Java garbage collection becomes increasingly fiddly and slow as the in-heap data increases.  
   随着堆内数据的增多，Java的垃圾回收变得越来越繁琐而缓慢。

As a result of these factors using the filesystem and relying on pagecache is superior to maintaining an in-memory cache or other structure—we at least double the available cache by having automatic access to all free memory, and likely double again by storing a compact byte structure rather than individual objects. Doing so will result in a cache of up to 28-30GB on a 32GB machine without GC penalties. Furthermore this cache will stay warm even if the service is restarted, whereas the in-process cache will need to be rebuilt in memory (which for a 10GB cache may take 10 minutes) or else it will need to start with a completely cold cache (which likely means terrible initial performance). This also greatly simplifies the code as all logic for maintaining coherency between the cache and filesystem is now in the OS, which tends to do so more efficiently and more correctly than one-off in-process attempts. If your disk usage favors linear reads then read-ahead is effectively pre-populating this cache with useful data on each disk read.

由于这些因素，使用文件系统并依赖pagecache（页缓存）将优于缓存在内存中或其他的结构 - 我们通过自动访问所有可用的内存将使得可用的内存至少提高一倍。并可能通过存储紧凑型字节结构再次提高一倍。这将使得32G机器上高达28-32GB的缓存，并无需GC。此外，即使服务重新启动，该缓存保持可用，而进程内的缓存则需要在内存中重建（10GB缓存需要10分钟），否则将需要启动完全冷却的缓存（这意味着可怕的初始化性能）。这也大大简化了代码，因为在缓存和文件系统之间维持的一致性的所有逻辑现在都在OS中，这比一次性进程更加有效和更正确。如果你的磁盘支持线性的读取，那么预读取将有效地将每个磁盘中有用的数据预填充此缓存。

This suggests a design which is very simple: rather than maintain as much as possible in-memory and flush it all out to the filesystem in a panic when we run out of space, we invert that. All data is immediately written to a persistent log on the filesystem without necessarily flushing to disk. In effect this just means that it is transferred into the kernel's pagecache.

这说明了一个非常简单的设计：当空间耗尽时，将它全部刷出文件系统。我们反过来看，不要在内存中尽可能多地维护， 所有数据立即写入文件系统上的持久性日志上，而不必刷新到磁盘上。 实际上这只是意味着它被转移到内核的页缓存中。

This style of pagecache-centric design is described in an [article](http://varnish.projects.linpro.no/wiki/ArchitectNotes) on the design of Varnish here (along with a healthy dose of arrogance).

这种以页缓存为中心的设计风格在这里描述。

#### ****常数时间就足够了**** （Constant Time Suffices）

The persistent data structure used in messaging systems are often a per-consumer queue with an associated BTree or other general-purpose random access data structures to maintain metadata about messages. BTrees are the most versatile data structure available, and make it possible to support a wide variety of transactional and non-transactional semantics in the messaging system. They do come with a fairly high cost, though: Btree operations are O(log N). Normally O(log N) is considered essentially equivalent to constant time, but this is not true for disk operations. Disk seeks come at 10 ms a pop, and each disk can do only one seek at a time so parallelism is limited. Hence even a handful of disk seeks leads to very high overhead. Since storage systems mix very fast cached operations with very slow physical disk operations, the observed performance of tree structures is often superlinear as data increases with fixed cache--i.e. doubling your data makes things much worse then twice as slow.

在消息系统中使用的持久数据结构常常具有相关联的BTree或其他通过随机访问数据结构的每个消费者队列，以维护关于消息的元数据。BTrees是可用的最通用的数据结构，可以在消息系统中支持各种各样的事务和非事务性语义。尽管，Btree的操作是O(log N)，但它们的成本相当高。通常O(log N)O(log N)基本上等同于恒定时间，但是磁盘操作不是这样，磁盘寻找在10ms的pop，每个磁盘一次只能做一次寻找，所以并行性受限制。因此，即使是少量的磁盘搜索导致非常高的开销。由于存储系统将非常快速的缓存操作与非常慢的物理磁盘操作相结合，因为数据随固定缓存而增加，所以观察到的树结构的性能通常是超线性的。- 即，你的数据翻倍则使得事情慢两倍还多。

Intuitively a persistent queue could be built on simple reads and appends to files as is commonly the case with logging solutions. This structure has the advantage that all operations are O(1) and reads do not block writes or each other. This has obvious performance advantages since the performance is completely decoupled from the data size—one server can now take full advantage of a number of cheap, low-rotational speed 1+TB SATA drives. Though they have poor seek performance, these drives have acceptable performance for large reads and writes and come at 1/3 the price and 3x the capacity.

直观上，持久队列可以建立在简单的读取和附加到文件上，就像日志解决方案的情况一样。 这种结构的优点是所有操作都是O(1)，并且读取不会阻塞写入或彼此。 这具有明显的性能优势，因为性能与数据大小完全分离 - 服务器现在可以充分利用这点，低转速 1+TB SATA驱动器。虽然这些驱动器的搜索性能不佳，但是对于大量的读写而言，这些驱动器具有可接受的性能，并且价格是1/3，能力为3倍。

Having access to virtually unlimited disk space without any performance penalty means that we can provide some features not usually found in a messaging system. For example, in Kafka, instead of attempting to deleting messages as soon as they are consumed, we can retain messages for a relative long period (say a week). This leads to a great deal of flexibility for consumers, as we will describe.

事实上，无需任何性能损失就可以访问几乎无限制的磁盘空间，这意味着我们可以提供一般消息传递系统无法提供的特性。 例如，在Kafka中，消息被消费后不是立马被删除，我们可以保留消息相对较长的时间（例如一个星期）。 这将为消费者带来很大的灵活性。

**kafka效率**

We have put significant effort into efficiency. One of our primary use cases is handling web activity data, which is very high volume: each page view may generate dozens of writes. Furthermore we assume each message published is read by at least one consumer (often many), hence we strive to make consumption as cheap as possible.

我们已经投入了很多精力到效率中。我们主要使用案例之一处理 web 活动数据，这是非常高的容量： 每个页面视图模式下可以产生几十个写入操作。此外，我们假设每个发布的消息至少一名消费者 （通常很多），因此我们努力使消费尽可能的廉价。

We have also found, from experience building and running a number of similar systems, that efficiency is a key to effective multi-tenant operations. If the downstream infrastructure service can easily become a bottleneck due to a small bump in usage by the application, such small changes will often create problems. By being very fast we help ensure that the application will tip-over under load before the infrastructure. This is particularly important when trying to run a centralized service that supports dozens or hundreds of applications on a centralized cluster as changes in usage patterns are a near-daily occurrence.  
我们还从建设和运行多个类似的系统中发现，有效率的多租户操作是效率的关键。 如果底层的基础架构服务由于应用程序少量的效率慢的代码而容易成为瓶颈，这种通常会产生问题。 我们非常快速度的处理以确保应用程序在基础设施之前tip-over，从而使得低于负载。 当在集中式集群中运行支持数十个或数百个应用程序的集中式服务时，这一点就会特别重要，因为使用模式的变化几乎每天都会发生。

We discussed disk efficiency in the previous section. Once poor disk access patterns have been eliminated, there are two common causes of inefficiency in this type of system: too many small I/O operations, and excessive byte copying.  
我们在上一节中讨论的磁盘的效率。一旦穷磁盘访问模式被淘汰，这种类型的系统有两个低效率的常见原因：大量的小I/O操作，和过度的字节复制。

The small I/O problem happens both between the client and the server and in the server's own persistent operations.  
client和server之间，服务器自己的持久化操作中，就会发生小I/O问题。

To avoid this, our protocol is built around a "message set" abstraction that naturally groups messages together. This allows network requests to group messages together and amortize the overhead of the network roundtrip rather than sending a single message at a time. The server in turn appends chunks of messages to its log in one go, and the consumer fetches large linear chunks at a time.  
为了避免这种情况，我们把这些消息集合在一起，这样减少网络请求的往返，而不是一次发送单个消息。

This simple optimization produces orders of magnitude speed up. Batching leads to larger network packets, larger sequential disk operations, contiguous memory blocks, and so on, all of which allows Kafka to turn a bursty stream of random message writes into linear writes that flow to the consumers.  
这种简单的优化产生了数量级的加速。 批处理导致更大的网络数据包，更大的顺序磁盘操作，连续的内存块等，这些都允许Kafka将随机消息写入的突发流转换成流向消费者的线性写入。

The other inefficiency is in byte copying. At low message rates this is not an issue, but under load the impact is significant. To avoid this we employ a standardized binary message format that is shared by the producer, the broker, and the consumer (so data chunks can be transferred without modification between them).  
另一个是无效率是字节复制。 在低速率下，这不是一个问题，但负载的情况下影响是显着的。 为了避免这种情况，我们采用由生产者，经纪人和消费者共享的标准化二进制消息格式（样数据块就可以在它们之间自由传输，无需转换）。

The message log maintained by the broker is itself just a directory of files, each populated by a sequence of message sets that have been written to disk in the same format used by the producer and consumer. Maintaining this common format allows optimization of the most important operation: network transfer of persistent log chunks. Modern unix operating systems offer a highly optimized code path for transferring data out of pagecache to a socket; in Linux this is done with the sendfile system call.   
broker维护的消息日志本身就是文件的目录，每个文件都是由生产者和消费者使用相同的格式写入磁盘的。维护这个公共的格式并允许优化最重要的操作：网络传输持久性日志块。 现代的unix操作系统提供一个优化的代码路径，用于将数据从页缓存传输到socket；在Linux中，是通过sendfile系统调用来完成的。Java提供了访问这个系统调用的方法：FileChannel.transferTo api。

To understand the impact of sendfile, it is important to understand the common data path for transfer of data from file to socket:  
要理解senfile的影响，重要的是要了解将数据从文件传输到socket的公共数据路径：

1. The operating system reads data from the disk into pagecache in kernel space  
   操作系统将数据从磁盘读入到内核空间的页缓存
2. The application reads the data from kernel space into a user-space buffer  
   应用程序将数据从内核空间读入到用户空间缓存中
3. The application writes the data back into kernel space into a socket buffer  
   应用程序将数据写回到内核空间到socket缓存中
4. The operating system copies the data from the socket buffer to the NIC buffer where it is sent over the network  
   操作系统将数据从socket缓冲区复制到网卡缓冲区，以便将数据经网络发出

This is clearly inefficient, there are four copies and two system calls. Using sendfile, this re-copying is avoided by allowing the OS to send the data from pagecache to the network directly. So in this optimized path, only the final copy to the NIC buffer is needed.  
这样做明显是低效的，这里有四次拷贝，两次系统调用。如果使用sendfile，再次拷贝可以被避免：允许操作系统将数据直接从页缓存发送到网络上。所以在这个优化的路径中，只有最后一步将数据拷贝到网卡缓存中是需要的。

We expect a common use case to be multiple consumers on a topic. Using the zero-copy optimization above, data is copied into pagecache exactly once and reused on each consumption instead of being stored in memory and copied out to kernel space every time it is read. This allows messages to be consumed at a rate that approaches the limit of the network connection.  
我们假设一个topic有多个消费者的情况。 并使用上面的零拷贝优化，数据被复制到页缓存中一次，并在每个消费上重复使用，而不是存储在存储器中，也不在每次读取时复制到用户空间。 这使得以接近网络连接限制的速度消费消息。

This combination of pagecache and sendfile means that on a Kafka cluster where the consumers are mostly caught up you will see no read activity on the disks whatsoever as they will be serving data entirely from cache.  
这种页缓存和sendfile组合，意味着Kafka集群的消费者大多数都完全从缓存消费消息，而磁盘没有任何读取活动。

For more background on the sendfile and zero-copy support in Java, see this [article](http://www.ibm.com/developerworks/linux/library/j-zerocopy).  
有关在Java中Sendfile和zero-copy（零拷贝）的支持更多的背景，请参阅本文。

#### 端到端的批量压缩

In some cases the bottleneck is actually not CPU or disk but network bandwidth. This is particularly true for a data pipeline that needs to send messages between data centers over a wide-area network. Of course the user can always compress its messages one at a time without any support needed from Kafka, but this can lead to very poor compression ratios as much of the redundancy is due to repetition between messages of the same type (e.g. field names in JSON or user agents in web logs or common string values). Efficient compression requires compressing multiple messages together rather than compressing each message individually.  
在某些情况下，瓶颈实际上不是CPU或磁盘，而是网络带宽。 对于需要在广域网上的数据中心之间发送消息的数据流水线尤其如此。 当然，用户可以一次压缩其消息，而无需Kafka所需的任何支持，但这可能导致非常差的压缩比，因为冗余的很多是由于相同类型的消息之间的重复（例如， Web日志中的JSON或用户代理或通用字符串值）。 有效的压缩需要压缩多个消息，而不是单独压缩每个消息。

Kafka supports this by allowing recursive message sets. A batch of messages can be clumped together compressed and sent to the server in this form. This batch of messages will be written in compressed form and will remain compressed in the log and will only be decompressed by the consumer.  
Kafka通过递归消息集来支持这一点。 一批消息可以一起压缩并以此形式发送到服务器。 这批消息将以压缩形式写入，并将在日志中保持压缩，并且只能由消费者解压缩。

Kafka supports GZIP and Snappy compression protocols. More details on compression can be found [here](https://cwiki.apache.org/confluence/display/KAFKA/Compression).  
Kafka支持GZIP和Snappy压缩协议，更多的细节可以在这里找到：https://cwiki.apache.org/confluence/display/KAFKA/Compression

**kafka生产者**

### 负载平衡

The producer sends data directly to the broker that is the leader for the partition without any intervening routing tier. To help the producer do this all Kafka nodes can answer a request for metadata about which servers are alive and where the leaders for the partitions of a topic are at any given time to allow the producer to appropriate direct its requests.

生产者将数据直接发送到分区leader的broker上（没有任何干预的路由层）。为了帮助producer做到这一点，Kafka所有节点都可应答给producer哪些服务器是正常的，哪些topic分区的leader允许producer在给定的时间内可以直接请求。

The client controls which partition it publishes messages to. This can be done at random, implementing a kind of random load balancing, or it can be done by some semantic partitioning function. We expose the interface for semantic partitioning by allowing the user to specify a key to partition by and using this to hash to a partition (there is also an option to override the partition function if need be). For example if the key chosen was a user id then all data for a given user would be sent to the same partition. This in turn will allow consumers to make locality assumptions about their consumption. This style of partitioning is explicitly designed to allow locality-sensitive processing in consumers.

客户端控制消息发布到哪个parition，可以随机，实现一种的随机负载平衡，或者也可以通过语义分区函数，我们暴露接口，以允许用户通过key去指定分区和使用使用hash来指向分区（如果需要，可重写分区函数）。例如：如果选择的key是用户ID，那么对给定的用户ID的所有数据将被发送到相同分区。反过来，消费者有能指定消费那个分区，这种设计风格，让消费者可以对敏感性的消息进行局部处理。

### 异步发送 asynchronous send

Batching is one of the big drivers of efficiency, and to enable batching the Kafka producer will attempt to accumulate data in memory and to send out larger batches in a single request. The batching can be configured to accumulate no more than a fixed number of messages and to wait no longer than some fixed latency bound (say 64k or 10 ms). This allows the accumulation of more bytes to send, and few larger I/O operations on the servers. This buffering is configurable and gives a mechanism to trade off a small amount of additional latency for better throughput.  
批处理是效率的一大驱动力，kafka生产者使用批处理试图在内存中积累数据，在单个请求发送累积的大批量数据，可以配置批处理积累的不大于一定的消息数，并等待时间不超过配置的延迟（64k 或 10毫秒）。这将累积更多消息 用于少数较大的I/O操作上，为了更好的吞吐量，这种缓存是可配置，并给出一种来权衡极少量的额外的延迟的机制。

Details on [configuration](http://kafka.apache.org/documentation.html#newproducerconfigs) and [api](http://kafka.apache.org/082/javadoc/index.html?org/apache/kafka/clients/producer/KafkaProducer.html) for the producer can be found elsewhere in the documentation.  
生产者的配置和api的详细信息可以在其他文档中找到。

**kafka消费者**

The Kafka consumer works by issuing "fetch" requests to the brokers leading the partitions it wants to consume. The consumer specifies its offset in the log with each request and receives back a chunk of log beginning from that position. The consumer thus has significant control over this position and can rewind it to re-consume data if need be.

kafka消费者通过向broker的leader分区发起“提取”请求。消费者指定每次请求日志的偏移量并收到那一块日志的起始位置。因此，消费者可以重新指定位置，重新消费。

### 推送 vs 拉取

An initial question we considered is whether consumers should pull data from brokers or brokers should push data to the consumer. In this respect Kafka follows a more traditional design, shared by most messaging systems, where data is pushed to the broker from the producer and pulled from the broker by the consumer. Some logging-centric systems, such as [Scribe](http://github.com/facebook/scribe) and [Apache Flume](http://flume.apache.org/) follow a very different push based path where data is pushed downstream. There are pros and cons to both approaches. However a push-based system has difficulty dealing with diverse consumers as the broker controls the rate at which data is transferred. The goal is generally for the consumer to be able to consume at the maximum possible rate; unfortunately in a push system this means the consumer tends to be overwhelmed when its rate of consumption falls below the rate of production (a denial of service attack, in essence). A pull-based system has the nicer property that the consumer simply falls behind and catches up when it can. This can be mitigated with some kind of backoff protocol by which the consumer can indicate it is overwhelmed, but getting the rate of transfer to fully utilize (but never over-utilize) the consumer is trickier than it seems. Previous attempts at building systems in this fashion led us to go with a more traditional pull model.

我们考虑的第一个问题是消费者应该从broker中pull数据还是broker向消费者push数据，在这方面，kafka遵循比较传统的设计，大多数消息系统，生产者推消息到broker，消费者从broker拉取消息，一些日志中心的系统，比如 Scribe 和Apache Flume ，采用非常不同的push模式（push数据到下游）。事实上，push模式和pull模式各有优劣。push模式很难适应消费速率不同的消费者，因为消息发送速率是由broker决定的。push模式的目标是尽可能以最快速度传递消息，但是这样很容易造成消费者来不及处理消息，典型的表现就是拒绝服务以及网络拥塞。而pull模式则可以根据consumer的消费能力以适当的速率消费消息。

Another advantage of a pull-based system is that it lends itself to aggressive batching of data sent to the consumer. A push-based system must choose to either send a request immediately or accumulate more data and then send it later without knowledge of whether the downstream consumer will be able to immediately process it. If tuned for low latency this will result in sending a single message at a time only for the transfer to end up being buffered anyway, which is wasteful. A pull-based design fixes this as the consumer always pulls all available messages after its current position in the log (or up to some configurable max size). So one gets optimal batching without introducing unnecessary latency.

基于pull模式的另一个优点是，它有助于积极的批处理的数据发送到消费者。基于push模式必须选择要么立即发送请求或者积累更多的数据，然后在不知道下游消费者是否能够立即处理它的情况下发送，如果是低延迟，这将导致一次只发送一条消息，以便传输缓存，这是实在是一种浪费，基于pull的设计解决这个问题，消费者总是pull在日志的当前位置之后pull所有可用的消息（或配置一些大size），所以消费者可设置消费多大的量，也不会引起不必要的等待时间。

The deficiency of a naive pull-based system is that if the broker has no data the consumer may end up polling in a tight loop, effectively busy-waiting for data to arrive. To avoid this we have parameters in our pull request that allow the consumer request to block in a "long poll" waiting until data arrives (and optionally waiting until a given number of bytes is available to ensure large transfer sizes).

基于pull模式不足之处在于，如果broker没有数据，消费者会轮询，忙等待数据直到数据到达，为了避免这种情况，我们允许消费者在pull请求时候使用“long poll”进行阻塞，直到数据到达（并且设置等待时间的好处是可以积累消息，组成大数据块一并发送）。

You could imagine other possible designs which would be only pull, end-to-end. The producer would locally write to a local log, and brokers would pull from that with consumers pulling from them. A similar type of "store-and-forward" producer is often proposed. This is intriguing but we felt not very suitable for our target use cases which have thousands of producers. Our experience running persistent data systems at scale led us to feel that involving thousands of disks in the system across many applications would not actually make things more reliable and would be a nightmare to operate. And in practice we have found that we can run a pipeline with strong SLAs at large scale without a need for producer persistence.  
你可以想一些其他的可能性的设计，不仅仅是pull，端对端。生产者在本地写入本地日志，broker从那里pull，消费者再pull，类似一种“存储-转发”的生产者，这种方式很有意思，但是我们觉得不是很适合我们这种有成千上万的生产者的情况，我们的大规模运行的持久化数据系统的经验使我们觉得，在许多应用领域涉及数以千计的系统磁盘不会真的使事情变得更加可靠，将是操作的噩梦。而在实践中我们发现，我们可以运行与大型强大的SLA管道，而不需要生产的持久性。

### 消费者定位

Keeping track of what has been consumed, is, surprisingly, one of the key performance points of a messaging system.

追踪已经消费的消息是令人惊讶的，在消息系统中，这是关键的性能之一。

 Most messaging systems keep metadata about what messages have been consumed on the broker. That is, as a message is handed out to a consumer, the broker either records that fact locally immediately or it may wait for acknowledgement from the consumer. This is a fairly intuitive choice, and indeed for a single machine server it is not clear where else this state could go. Since the data structure used for storage in many messaging systems scale poorly, this is also a pragmatic choice--since the broker knows what is consumed it can immediately delete it, keeping the data size small.  
大多数消息系统保留在broker上消费消息的元数据。 也就是说，当消息发送给消费者时，broker本地立即记录该事实，或者可以等待消费者的应答确认。 这是一个相当直观的选择，实际上对于单个机器服务器来说，尚不清楚这个状态是什么。 由于许多消息系统中用于存储的数据结构规模不大，这也是务实的选择 - 因为broker知道哪些已经消费，可以立即删除它，从而保持数据大小不变。

What is perhaps not obvious, is that getting the broker and consumer to come into agreement about what has been consumed is not a trivial problem. If the broker records a message as consumed immediately every time it is handed out over the network, then if the consumer fails to process the message (say because it crashes or the request times out or whatever) that message will be lost. To solve this problem, many messaging systems add an acknowledgement feature which means that messages are only marked as sent not consumed when they are sent; the broker waits for a specific acknowledgement from the consumer to record the message as consumed. This strategy fixes the problem of losing messages, but creates new problems. First of all, if the consumer processes the message but fails before it can send an acknowledgement then the message will be consumed twice. The second problem is around performance, now the broker must keep multiple states about every single message (first to lock it so it is not given out a second time, and then to mark it as permanently consumed so that it can be removed). Tricky problems must be dealt with, like what to do with messages that are sent but never acknowledged.

并不明显的是，让broker和消费者所消费的消息达成一致并不是一个微不足道的问题。如果broker每次通过网络发出消息立即记录的话，那么如果消费者无法处理该消息（比如崩溃或请求超时），则该消息将丢失。为了解决这个问题，许多消息系统添加了一个“应答”功能，这意味着当消息发送时，消息仅仅标记为“发送”而不是“已消费”。broker等待消费者应答该消息，消息才被标记为“已消费”。这确认解决了丢失消息的问题，但是产生了一个新的问题。首先，如果消费者处理了消息，但是在发送应答时失败了，那么该消息将会被处理两次。第二个问题是关于性能，现在broker必须保持关于每个单个消息的多个状态（首先锁定它，所以它不会被发送两次，然后将其标记为永久已消耗，以便可以被删除）。必须处理这些棘手的问题，就像发送但未应答的消息一样。

Kafka handles this differently. Our topic is divided into a set of totally ordered partitions, each of which is consumed by one consumer at any given time. This means that the position of consumer in each partition is just a single integer, the offset of the next message to consume. This makes the state about what has been consumed very small, just one number for each partition. This state can be periodically checkpointed. This makes the equivalent of message acknowledgements very cheap.

kafka处理方式不同。 我们的topic被分为一组完全有序的分区，每个分区在任何给定的时间都由每个订阅消费者组中的一个消费者消费。 这意味着消费者在每个分区中的位置只是一个整数，下一个消息消费的偏移量。 这使得关于已消费到哪里的状态变得非常的小，每个分区只有一个数字。 可以定期检查此状态。 这使得等同于消息应答并更轻量。

There is a side benefit of this decision. A consumer can deliberately rewind back to an old offset and re-consume data. This violates the common contract of a queue, but turns out to be an essential feature for many consumers. For example, if the consumer code has a bug and is discovered after some messages are consumed, the consumer can re-consume those messages once the bug is fixed.

这么做有一个好处。 消费者可以故意地回到旧的偏移量并重新消费数据。 这违反了一个队列的共同契约，但这被证明是许多消费者的基本特征。 例如，如果消费者代码有bug，并且在消费一些消息之后被发现，消费者可以在修复错误后重新消费这些消息。

### 离线数据加载

Scalable persistence allows for the possibility of consumers that only periodically consume such as batch data loads that periodically bulk-load data into an offline system such as Hadoop or a relational data warehouse.

可扩展持久性，以允许消费者只需周期性地消费如批量数据加载的可能性，以便将数据定期批量加载到如Hadoop或关系数据仓库之类的离线系统中。

In the case of Hadoop we parallelize the data load by splitting the load over individual map tasks, one for each node/topic/partition combination, allowing full parallelism in the loading. Hadoop provides the task management, and tasks which fail can restart without danger of duplicate data—they simply restart from their original position.

在Hadoop的情况下，我们通过将负载分解为单独的map任务来并行化数据负载，每个node/topic/partition组合一个负载，允许在加载中完全并行。 Hadoop提供任务管理，无法重新启动的任务可以重新启动，而不会有重复数据的危险 - 他们只需从原始位置重新启动。

**kafka消息传递保障**

Now that we understand a little about how producers and consumers work, let's discuss the semantic guarantees Kafka provides between producer and consumer. Clearly there are multiple possible message delivery guarantees that could be provided:  
现在我们了解一些关于生产者和消费者是如何工作的，接下来我们来讨论kafka提供了生产者和消费者之间的担保语义。有多种可能的消息传递保证可以提供：

* *At most once*—Messages may be lost but are never redelivered.  
  最多一次 --- 消息可能丢失，但绝不会重发。
* *At least once*—Messages are never lost but may be redelivered.  
  至少一次 --- 消息绝不会丢失，但有可能重新发送。
* *Exactly once*—this is what people actually want, each message is delivered once and only once.  
  正好一次 --- 这是人们真正想要的，每个消息传递一次且仅一次。

It's worth noting that this breaks down into two problems: the durability guarantees for publishing a message and the guarantees when consuming a message.

可分解成两个问题：发送消息时的耐久性保障和消费消息的保障。

Many systems claim to provide "exactly once" delivery semantics, but it is important to read the fine print, most of these claims are misleading (i.e. they don't translate to the case where consumers or producers can fail, or cases where there are multiple consumer processes, or cases where data written to disk can be lost).

很多消息系统声称提供“正好一次”的传递语义，但是在阅读相关文章时，更多是误导（例如，它们没有解释消费者或生产者可能失败的情况，有多个消费者进程的情况，或写入磁盘的数据可能丢失的情况）

Kafka's semantics are straight-forward. When publishing a message we have a notion of the message being "committed" to the log. Once a published message is committed it will not be lost as long as one broker that replicates the partition to which this message was written remains "alive". The definition of alive as well as a description of which types of failures we attempt to handle will be described in more detail in the next section. For now let's assume a perfect, lossless broker and try to understand the guarantees to the producer and consumer. If a producer attempts to publish a message and experiences a network error it cannot be sure if this error happened before or after the message was committed. This is similar to the semantics of inserting into a database table with an autogenerated key.

kafka的语义是很直接的，我们有一个概念，当发布一条消息时，该消息 “committed（承诺）” 到了日志，一旦发布的消息是”承诺“的，只要副本分区写入了此消息的一个broker仍然"活着”，它就不会丢失。“活着”的定义以及描述的类型，我们处理失败的情况将在下一节中详细描述。现在让我们假设一个完美的不会丢消息的broker，并去了解如何保障生产者和消费者的，如果一个生产者发布消息并且正好遇到网络错误，就不能确定已提交的消息是否是在这个错误发生之前或之后。这类似于用自动生成key插入到一个数据库表。

Prior to 0.11.0.0, if a producer failed to receive a response indicating that a message was committed, it had little choice but to resend the message. This provides at-least-once delivery semantics since the message may be written to the log again during resending if the original request had in fact succeeded. Since 0.11.0.0, the Kafka producer also supports an idempotent delivery option which guarantees that resending will not result in duplicate entries in the log. To achieve this, the broker assigns each producer an ID and deduplicates messages using a sequence number that is sent by the producer along with every message. Also beginning with 0.11.0.0, the producer supports the ability to send messages to multiple topic partitions using transaction-like semantics: i.e. either all messages are successfully written or none of them are. The main use case for this is exactly-once processing between Kafka topics (described below).  
在0.11.0.0之前，如果一个生产者没有收到消息提交的响应，那么只能重新发送消息。 这提供了至少一次传递语义，因为如果原始请求实际上已成功，则在重新发送期间再次将消息写入到日志中。自0.11.0.0起，Kafka生产者支持幂等传递选项，保证重新发送不会导致日志中重复。 broker为每个生产者分配一个ID，并通过生产者发送的序列号为每个消息进行去重。从0.11.0.0开始，生产者支持使用类似事务的语义将消息发送到多个topic分区的能力：即所有消息都被成功写入，或者没有。这个主要用于Kafka topic之间“正好一次“处理（如下所述）。

Not all use cases require such strong guarantees. For uses which are latency sensitive we allow the producer to specify the durability level it desires. If the producer specifies that it wants to wait on the message being committed this can take on the order of 10 ms. However the producer can also specify that it wants to perform the send completely asynchronously or that it wants to wait only until the leader (but not necessarily the followers) have the message.

并不是所有的情况都需要这么强力的保障，对于延迟敏感的，我们允许生产者指定它想要的耐用性水平。如生产者可以指定它获取需等待10毫秒量级上的响应。生产者也可以指定异步发送，或只等待leader（不需要副本的响应）有响应，

Now let's describe the semantics from the point-of-view of the consumer. All replicas have the exact same log with the same offsets. The consumer controls its position in this log. If the consumer never crashed it could just store this position in memory, but if the consumer fails and we want this topic partition to be taken over by another process the new process will need to choose an appropriate position from which to start processing. Let's say the consumer reads some messages -- it has several options for processing the messages and updating its position.

现在让我们从消费者的角度描述语义。所有的副本都有相同的日志相同的偏移量。消费者控制offset在日志中的位置。如果消费者永不宕机它可能只是在内存中存储这个位置，但是如果消费者故障，我们希望这个topic分区被另一个进程接管，新进程需要选择一个合适的位置开始处理。我们假设消费者读取了一些消息，几种选项用于处理消息和更新它的位置。

1. It can read the messages, then save its position in the log, and finally process the messages. In this case there is a possibility that the consumer process crashes after saving its position but before saving the output of its message processing. In this case the process that took over processing would start at the saved position even though a few messages prior to that position had not been processed. This corresponds to "at-most-once" semantics as in the case of a consumer failure messages may not be processed.  
   读取消息，然后在日志中保存它的位置，最后处理消息。在这种情况下，有可能消费者保存了位置之后，但是处理消息输出之前崩溃了。在这种情况下，接管处理的进程会在已保存的位置开始，即使该位置之前有几个消息尚未处理。这对应于“最多一次” ，在消费者处理失败消息的情况下，不进行处理。
2. It can read the messages, process the messages, and finally save its position. In this case there is a possibility that the consumer process crashes after processing messages but before saving its position. In this case when the new process takes over the first few messages it receives will already have been processed. This corresponds to the "at-least-once" semantics in the case of consumer failure. In many cases messages have a primary key and so the updates are idempotent (receiving the same message twice just overwrites a record with another copy of itself).  
   读取消息，处理消息，最后保存消息的位置。在这种情况下，可能消费进程处理消息之后，但保存它的位置之前崩溃了。在这种情况下，当新的进程接管了它，这将接收已经被处理的前几个消息。这就符合了“至少一次”的语义。在多数情况下消息有一个主键，以便更新幂等（其任意多次执行所产生的影响均与一次执行的影响相同）。
3. So what about exactly once semantics (i.e. the thing you actually want)? When consuming from a Kafka topic and producing to another topic (as in a Kafka Streams application), we can leverage the new transactional producer capabilities in 0.11.0.0 that were mentioned above. The consumer's position is stored as a message in a topic, so we can write the offset to Kafka in the same transaction as the output topics receiving the processed data. If the transaction is aborted, the consumer's position will revert to its old value and the produced data on the output topics will not be visible to other consumers, depending on their "isolation level." In the default "read\_uncommitted" isolation level, all messages are visible to consumers even if they were part of an aborted transaction, but in "read\_committed," the consumer will only return messages from transactions which were committed (and any messages which were not part of a transaction).  
   那么什么是“正好一次”语义（也就是你真正想要的东西）? 当从Kafka主题消费并生产到另一个topic时（例如Kafka Stream），我们可以利用之前提到0.11.0.0中的生产者新事物功能。消费者的位置作为消息存储到topic中，因此我们可以与接收处理后的数据的输出topic使用相同的事务写入offset到Kafka。如果事物中断，则消费者的位置将恢复到老的值，根据其”隔离级别“，其他消费者将不会看到输出topic的生成数据，在默认的”读取未提交“隔离级别中，所有消息对消费者都是可见的，即使是被中断的事务的消息。但是在”读取提交“中，消费者将只从已提交的事物中返回消息。

When writing to an external system, the limitation is in the need to coordinate the consumer's position with what is actually stored as output. The classic way of achieving this would be to introduce a two-phase commit between the storage of the consumer position and the storage of the consumers output. But this can be handled more simply and generally by letting the consumer store its offset in the same place as its output. This is better because many of the output systems a consumer might want to write to will not support a two-phase commit. As an example of this, consider a Kafka Connect connector which populates data in HDFS along with the offsets of the data it reads so that it is guaranteed that either data and offsets are both updated or neither is. We follow similar patterns for many other data systems which require these stronger semantics and for which the messages do not have a primary key to allow for deduplication.  
当写入到外部系统时，需要将消费者的位置与实际存储为输出的位置进行协调。实现这一目标的典型方法是在消费者位置的存储和消费者输出的存储之间引入两阶段的”提交“。但是，这可以更简单，通过让消费者将其offset存储在与其输出相同的位置。这样最好，因为大多数的输出系统不支持两阶段”提交“。作为一个例子，考虑一个Kafka Connect连接器，它填充HDFS中的数据以及它读取的数据的offset，以保证数据和offset都被更新，或者都不更新。 对于需要这些更强大语义的许多其他数据系统，我们遵循类似的模式，并且消息不具有允许重复数据删除的主键。

So effectively Kafka guarantees at-least-once delivery by default and allows the user to implement at most once delivery by disabling retries on the producer and committing its offset prior to processing a batch of messages. Exactly-once delivery requires co-operation with the destination storage system but Kafka provides the offset which makes implementing this straight-forward.

kafka默认是保证“至少一次”传递，并允许用户通过禁止生产者重试和处理一批消息前提交它的偏移量来实现 “最多一次”传递。而“正好一次”传递需要与目标存储系统合作，但kafka提供了偏移量，所以实现这个很简单。

**kafka副本和leader选举**

# 4.7 副本

Kafka replicates the log for each topic's partitions across a configurable number of servers (you can set this replication factor on a topic-by-topic basis). This allows automatic failover to these replicas when a server in the cluster fails so messages remain available in the presence of failures.

kafka集群在各个服务器上备份topic分区中日志（ps：就是备份我们的消息，称为副本，你可以设置每个topic的副本数）。当集群中某个服务器发生故障时，自动切换到这些副本，从而保障在故障时消息仍然可用。

Other messaging systems provide some replication-related features, but, in our (totally biased) opinion, this appears to be a tacked-on thing, not heavily used, and with large downsides: slaves are inactive, throughput is heavily impacted, it requires fiddly manual configuration, etc. Kafka is meant to be used with replication by default—in fact we implement un-replicated topics as replicated topics where the replication factor is one.  
其他消息系统提供一些副本相关的功能，但是，在我们看来（有偏见），这似乎是一个附加的东西，没有大量的使用，这有很大的缺点：slave不活跃，吞吐量受到严重影响，它需要的精确的手动配置等。kafka使用的是默认副本 — 就是不需要副本的topic的复制因子就是1。

The unit of replication is the topic partition. Under non-failure conditions, each partition in Kafka has a single leader and zero or more followers. The total number of replicas including the leader constitute the replication factor. All reads and writes go to the leader of the partition. Typically, there are many more partitions than brokers and the leaders are evenly distributed among brokers. The logs on the followers are identical to the leader's log—all have the same offsets and messages in the same order (though, of course, at any given time the leader may have a few as-yet unreplicated messages at the end of its log).

副本以topic的分区为单位。在正常情况下，kafka每个分区都有一个单独的leader，0个或多个follower。副本的总数包括leader。所有的读取和写入到该分区的leader。通常，分区数比broker多，leader均匀分布在broker。follower的日志完全等同于leader的日志 — 相同的顺序相同的偏移量和消息（当然，在任何一个时间点上，leader比follower多几条消息，尚未同步到follower）

Followers consume messages from the leader just as a normal Kafka consumer would and apply them to their own log. Having the followers pull from the leader has the nice property of allowing the follower to naturally batch together log entries they are applying to their log.

Followers作为普通的消费者从leader中消费消息并应用到自己的日志中。并允许follower从leader拉取批量日志应用到自己的日志，这样具有良好的性能。

As with most distributed systems automatically handling failures requires having a precise definition of what it means for a node to be "alive". For Kafka node liveness has two conditions

和大多数分布式系统一样，自动处理失败的节点。需要精确的定义什么样的节点是“活着”的，对于kafka的节点活着有2个条件：

1. A node must be able to maintain its session with ZooKeeper (via ZooKeeper's heartbeat mechanism)  
   一个节点必须能维持与zookeeper的会话（通过zookeeper的心跳机制）
2. If it is a slave it must replicate the writes happening on the leader and not fall "too far" behind  
   如果它是一个slave，它必须复制leader并且不能落后"太多"

We refer to nodes satisfying these two conditions as being "in sync" to avoid the vagueness of "alive" or "failed". The leader keeps track of the set of "in sync" nodes. If a follower dies, gets stuck, or falls behind, the leader will remove it from the list of in sync replicas. The definition of, how far behind is too far, is controlled by the replica.lag.max.messages configuration and the definition of a stuck replica is controlled by the replica.lag.time.max.ms configuration.

我们让节点满足这2个“同步”条件，以区分“活着”还是“故障”。leader跟踪“同步”节点。如果一个follower死掉，卡住，或落后，leader将从同步副本列表中移除它。落后是通过replica.lag.max.messages配置控制，卡住是通过replica.lag.time.max.ms配置控制的。

In distributed systems terminology we only attempt to handle a "fail/recover" model of failures where nodes suddenly cease working and then later recover (perhaps without knowing that they have died). Kafka does not handle so-called "Byzantine" failures in which nodes produce arbitrary or malicious responses (perhaps due to bugs or foul play).

在分布式系统，我们只是尝试处理故障节点突然停止工作和然后稍后恢复的“故障/恢复”模式（也许不知道它们已经故障了）。kafka不处理节点产生任意或恶意的响应（也许是因为bug或犯规），所谓的“Byzantine”故障。

We can now more precisely define that a message is considered committed when all in sync replicas for that partition have applied it to their log. Only committed messages are ever given out to the consumer. This means that the consumer need not worry about potentially seeing a message that could be lost if the leader fails. Producers, on the other hand, have the option of either waiting for the message to be committed or not, depending on their preference for tradeoff between latency and durability. This preference is controlled by the acks setting that the producer uses. Note that topics have a setting for the "minimum number" of in-sync replicas that is checked when the producer requests acknowledgment that a message has been written to the full set of in-sync replicas. If a less stringent acknowledgement is requested by the producer, then the message can be committed, and consumed, even if the number of in-sync replicas is lower than the minimum (e.g. it can be as low as just the leader).

我们现在可以更精确的定义当该分区的所有同步副本已经写入到其日志中时漫该消息视为“已提交”。只有“已提交”的消息才会给到消费者。所有消费者无需担心如果leader故障，会消费到丢失的消息。另一方面，生产者可以选择等待消费提交，这取决于你更偏向延迟或耐用性（通过acks控制）。当生产者请求确保消息已经写入到全部的同步副本中（可以通过设置topic同步副本的“最小数”）。如果生产者要求不严格，则即使同步副本的数量低于最小值，也可以提交和消费该消息。

The guarantee that Kafka offers is that a committed message will not be lost, as long as there is at least one in sync replica alive, at all times.

kafka提供担保，在任何时候，只要至少有一个同步副本活着，承诺的消息就不会丢失。

Kafka will remain available in the presence of node failures after a short fail-over period, but may not remain available in the presence of network partitions.

kafka短暂的故障转移期间，失败的节点仍可用。但可能无法在网络分区仍然可用。

#### 复制日志：Quorums，ISR，和状态机制

**Quorum**：原指为了处理事务、拥有做出决定的权力而必须出席的众议员或参议员的数量（一般指半数以上）。

At its heart a Kafka partition is a replicated log. The replicated log is one of the most basic primitives in distributed data systems, and there are many approaches for implementing one. A replicated log can be used by other systems as a primitive for implementing other distributed systems in the [state-machine style](http://en.wikipedia.org/wiki/State_machine_replication).

kafka分区的核心是一个副本日志，副本是在分布式数据系统的最基础原始功能之一。并有许多方法实现，副本日志可以被其他系统用作状态机类型实现其他分布式系统的原始功能。

A replicated log models the process of coming into consensus on the order of a series of values (generally numbering the log entries 0, 1, 2, ...). There are many ways to implement this, but the simplest and fastest is with a leader who chooses the ordering of values provided to it. As long as the leader remains alive, all followers need to only copy the values and ordering, the leader chooses.

副本日志模拟了对一系列值顺序进入的过程（通常日志编号是 0，1，2，……）。有很多方法可以实现这一点，但最简单和最快的是leader提供选择的排序值，只要leader活着，所有的followers只需要复制和排序。

Of course if leaders didn't fail we wouldn't need followers! When the leader does die we need to choose a new leader from among the followers. But followers themselves may fall behind or crash so we must ensure we choose an up-to-date follower. The fundamental guarantee a log replication algorithm must provide is that if we tell the client a message is committed, and the leader fails, the new leader we elect must also have that message. This yields a tradeoff: if the leader waits for more followers to acknowledge a message before declaring it committed then there will be more potentially electable leaders.

当然，如果leader没有故障，我们就不需要follower！当leader确实故障了，我们需要从follower中选出新的leader，但是follower自己可能落后或崩溃，所以我们必须选择一个最新的follower。日志复制算法必须提供保证，如果我们告诉客户端消息是已发送，leader故障了，我们选举的新的leader必须要有这条消息，这就产生一个权衡：如果leader等待更多的follwer声明已提交之前，应答消息的话，将会有更多有资格的leader。

If you choose the number of acknowledgements required and the number of logs that must be compared to elect a leader such that there is guaranteed to be an overlap, then this is called a Quorum.

如果你选择需要应答数量必须和日志的数量进行比较，选出一个leader。这样保证有重叠，那么这就是所谓的Quorum（法定人数）。

A common approach to this tradeoff is to use a majority vote for both the commit decision and the leader election. This is not what Kafka does, but let's explore it anyway to understand the tradeoffs. Let's say we have 2*f*+1 replicas. If *f*+1 replicas must receive a message prior to a commit being declared by the leader, and if we elect a new leader by electing the follower with the most complete log from at least *f*+1 replicas, then, with no more than *f* failures, the leader is guaranteed to have all committed messages. This is because among any *f*+1 replicas, there must be at least one replica that contains all committed messages. That replica's log will be the most complete and therefore will be selected as the new leader. There are many remaining details that each algorithm must handle (such as precisely defined what makes a log more complete, ensuring log consistency during leader failure or changing the set of servers in the replica set) but we will ignore these for now.

一种常见的方法，用多数投票决定leader选举。kafka不是这样做的，但先让我们了解这个权衡，假如，我们有2f+1副本，如果f+1副本在leader提交之前必须收到消息，并且如果我们选举新的leader，至少从f+1副本选出最完整日志的follwer，并且不大于f的失败，leader担保所有已提交的信息。这是因为任何f+1副本中，必须至少有一个副本，其中包含所有已提交的消息。该副本的日志是最完整的，因此选定为新的leader。有许多其余细节，每个算法必须处理（如 精确的定义是什么让一个日志更加完整，确保日志一致性，leader故障期间或更改服务器的副本集），但我们现在不讲这些。

This majority vote approach has a very nice property: the latency is dependent on only the fastest servers. That is, if the replication factor is three, the latency is determined by the faster slave not the slower one.

这种投票表决的方式有一个非常好的特性：仅依赖速度最快的服务器，也就是说，如果复制因子为三个，由最快的一个来确定。

There are a rich variety of algorithms in this family including ZooKeeper's [Zab](http://www.stanford.edu/class/cs347/reading/zab.pdf), [Raft](https://ramcloud.stanford.edu/wiki/download/attachments/11370504/raft.pdf), and [Viewstamped Replication](http://pmg.csail.mit.edu/papers/vr-revisited.pdf). The most similar academic publication we are aware of to Kafka's actual implementation is [PacificA](http://research.microsoft.com/apps/pubs/default.aspx?id=66814) from Microsoft.

有各种丰富的算法，包括zookeeper的Zab、 Raft和 Viewstamped Replication。kafka实现的最相似的学术理论是微软的PacificA。

The downside of majority vote is that it doesn't take many failures to leave you with no electable leaders. To tolerate one failure requires three copies of the data, and to tolerate two failures requires five copies of the data. In our experience having only enough redundancy to tolerate a single failure is not enough for a practical system, but doing every write five times, with 5x the disk space requirements and 1/5th the throughput, is not very practical for large volume data problems. This is likely why quorum algorithms more commonly appear for shared cluster configuration such as ZooKeeper but are less common for primary data storage. For example in HDFS the namenode's high-availability feature is built on a [majority-vote-based journal](http://blog.cloudera.com/blog/2012/10/quorum-based-journaling-in-cdh4-1), but this more expensive approach is not used for the data itself.

多数投票的缺点是，故障数还不太多的情况下会让你没有候选人可选，要容忍1个故障需要3个数据副本，容忍2个故障需要5个数据副本。实际的系统以我们的经验只能容忍单个故障的冗余是不够的，但是如果5个数据副本，每个写5次，5倍的磁盘空间要求，1/5的吞吐量，这对于大数据量系统是不实用的，这可能是quorum算法更通常在共享集群配置。如zookeeper，主要用于数据存储的系统是不太常见的。例如，在HDFS namenode的高可用性特性是建立在majority-vote-based journal，但这更昂贵的方法不能用于数据本身。

Kafka takes a slightly different approach to choosing its quorum set. Instead of majority vote, Kafka dynamically maintains a set of in-sync replicas (ISR) that are caught-up to the leader. Only members of this set are eligible for election as leader. A write to a Kafka partition is not considered committed until *all* in-sync replicas have received the write. This ISR set is persisted to ZooKeeper whenever it changes. Because of this, any replica in the ISR is eligible to be elected leader. This is an important factor for Kafka's usage model where there are many partitions and ensuring leadership balance is important. With this ISR model and *f+1* replicas, a Kafka topic can tolerate *f* failures without losing committed messages.

kafka采用了一种稍微不同的方法选择quorum，而不是多数投票，kafka动态维护一组同步leader数据的副本（ISR），只有这个组的成员才有资格当选leader，kafka副本写入不被认为是已提交，直到所有的同步副本已经接收才认为。这组ISR保存在zookeeper，正因为如此，在ISR中的任何副本都有资格当选leader，这是kafka的使用模型，有多个分区和确保leader平衡是很重要的一个重要因素。有了这个模型，ISR和f+1副本，kafka的主题可以容忍f失败而不会丢失已提交的消息。

For most use cases we hope to handle, we think this tradeoff is a reasonable one. In practice, to tolerate *f* failures, both the majority vote and the ISR approach will wait for the same number of replicas to acknowledge before committing a message (e.g. to survive one failure a majority quorum needs three replicas and one acknowledgement and the ISR approach requires two replicas and one acknowledgement). The ability to commit without the slowest servers is an advantage of the majority vote approach. However, we think it is ameliorated by allowing the client to choose whether they block on the message commit or not, and the additional throughput and disk space due to the lower required replication factor is worth it.

对于大多数情况下，我们希望这么处理，我们认为这个代价是合理的，在实践中，容忍f故障，多数投票和ISR方法将等待相同数量的副本提交消息之前进行确认（例如：活着1个，故障多数的quorum，需要3个副本和1个应答，ISR方法需要2个副本和1个应答）。排除最慢的服务器是多数投票的优点，但是，我们认为允许客户选择是否阻塞消息的提交可以改善这个问题，并通过降低复制因子获得额外的吞吐量和磁盘空间也是值得的。

Another important design distinction is that Kafka does not require that crashed nodes recover with all their data intact. It is not uncommon for replication algorithms in this space to depend on the existence of "stable storage" that cannot be lost in any failure-recovery scenario without potential consistency violations. There are two primary problems with this assumption. First, disk errors are the most common problem we observe in real operation of persistent data systems and they often do not leave data intact. Secondly, even if this were not a problem, we do not want to require the use of fsync on every write for our consistency guarantees as this can reduce performance by two to three orders of magnitude. Our protocol for allowing a replica to rejoin the ISR ensures that before rejoining, it must fully re-sync again even if it lost unflushed data in its crash.

另一个重要的区别是，kafka不要求节点崩溃后所有的数据保持原样恢复。不违反一致性，在任何故障恢复场景不丢失的“稳定存储”复制算法是极少的。这种假设有两个主要的问题，首先，根据我们的观察，磁盘错误在持久化数据系统是最常见的问题，通常数据不会完好无损。其次，即使这不是一个问题，我们不希望在每次写入都用fsync做一致性的保障。因为这导致2个至3个数量级的性能下降，我们允许一个副本重新加入ISR协议确保在加入之前，必须再次完全重新同步，即使丢失崩溃未刷新的数据。

### unclean leader选举：如果他们都死了怎么办？

Note that Kafka's guarantee with respect to data loss is predicated on at least on replica remaining in sync. If all the nodes replicating a partition die, this guarantee no longer holds.

请注意，kafka对数据丢失的保障是基于至少有一个副本在保持同步。如果分区的所有复制节点都死了，这保证就不再成立。

However a practical system needs to do something reasonable when all the replicas die. If you are unlucky enough to have this occur, it is important to consider what will happen. There are two behaviors that could be implemented:

如果你人品超差，遇到所有的副本都死了，这时候，你要考虑将会发生问题，并做重要的2件事：

1. Wait for a replica in the ISR to come back to life and choose this replica as the leader (hopefully it still has all its data).  
   等待在ISR中的副本起死回生并选择该副本作为leader（希望它仍有所有数据）。
2. Choose the first replica (not necessarily in the ISR) that comes back to life as the leader.  
   选择第一个副本 （不一定在 ISR)，作为leader。

This is a simple tradeoff between availability and consistency. If we wait for replicas in the ISR, then we will remain unavailable as long as those replicas are down. If such replicas were destroyed or their data was lost, then we are permanently down. If, on the other hand, a non-in-sync replica comes back to life and we allow it to become leader, then its log becomes the source of truth even though it is not guaranteed to have every committed message. In our current release we choose the second strategy and favor choosing a potentially inconsistent replica when all replicas in the ISR are dead.This behavior can be disabled using configuration property unclean.leader.election.enable, to support use cases where downtime is preferable to inconsistency.

这是在可用性和一致性的简单权衡。如果我们等待ISR中的副本，那么只要副本不可用，我们将保持不可用，如果这些副本摧毁或数据已经丢失，那么就是永久的不可用。另一方面，如果non-in-sync（非同步）的副本，我们让它成为leader，让它的日志成为`源`，即使它不能保证承诺的消息不丢失。在我们当前的版本中我们选择第2种方式，支持选择在ISR中所有副本死了时候可选择不能保证一致的副本。可以通过配置unclean.leader.election.enable禁用此行为，以支持停机优先于不一致。

This dilemma is not specific to Kafka. It exists in any quorum-based scheme. For example in a majority voting scheme, if a majority of servers suffer a permanent failure, then you must either choose to lose 100% of your data or violate consistency by taking what remains on an existing server as your new source of truth.

这个难题不是只有kafka有，任何基于quorum的都有。例如在多数投票中，如果多数服务器都遭受永久性的故障，那么你必须选择丢失100%的数据，或违反一致性，用剩下现有服务器作为新的源。

### 可用性和耐久性保证  （Availability and Durability Guarantees）

When writing to Kafka, producers can choose whether they wait for the message to be acknowledged by 0,1 or all (-1) replicas. Note that "acknowledgement by all replicas" does not guarantee that the full set of assigned replicas have received the message. By default, when acks=all, acknowledgement happens as soon as all the current in-sync replicas have received the message. For example, if a topic is configured with only two replicas and one fails (i.e., only one in sync replica remains), then writes that specify acks=all will succeed. However, these writes could be lost if the remaining replica also fails. Although this ensures maximum availability of the partition, this behavior may be undesirable to some users who prefer durability over availability. Therefore, we provide two topic-level configurations that can be used to prefer message durability over availability:

当写入到kakfa时，生产者可以选择是否等待0,1 或 全部副本（-1）的消息确认。需要注意的是“所有副本确认”并不能保证全部分配副本已收到消息。默认情况下，当acks=all时，只要当前所有在同步中的副本收到消息，就会进行确认。例如：如果一个topic有2个副本，有一个故障（即，只剩下一个同步副本），即使写入是 acks=all 也将会成功。如果剩下的副本也故障了那么这些写入就会丢失。虽然这可以确保分区的最大可用性，这种方式可能不受欢迎，一些用户喜欢耐久性超过可用性。因此，我们提供两种配置。

1. Disable unclean leader election - if all replicas become unavailable, then the partition will remain unavailable until the most recent leader becomes available again. This effectively prefers unavailability over the risk of message loss. See the previous section on Unclean Leader Election for clarification.  
   禁用unclean leader选举 - 如果所有副本不可用，那份分区将一直不可用，直到最近的leader再次变得可用，这种宁愿不可用，而不是冒着丢失消息的风险。
2. Specify a minimum ISR size - the partition will only accept writes if the size of the ISR is above a certain minimum, in order to prevent the loss of messages that were written to just a single replica, which subsequently becomes unavailable. This setting only takes effect if the producer uses required.acks=-1 and guarantees that the message will be acknowledged by at least this many in-sync replicas. This setting offers a trade-off between consistency and availability. A higher setting for minimum ISR size guarantees better consistency since the message is guaranteed to be written to more replicas which reduces the probability that it will be lost. However, it reduces availability since the partition will be unavailable for writes if the number of in-sync replicas drops below the minimum threshold.  
   指定一个最小的ISR大小 — 如果ISR的大小高于最小值，则该分区才接受写入，以预防消息丢失，防止消息写到单个副本上，则让其变为不可用。如果生产者使用的是acks=all并保证最少这些同步分本已确认，则设置才生效。该设置提供一致性和可用性之间的权衡。ISR的大小设置的越高更好的保证一致性，因为消息写到更多的副本以减少消息丢失的风险。但是，这样降低了可用性，因为如果同步副本数低于最小的阈值，则该分区将不可写入。

### 副本管理

The above discussion on replicated logs really covers only a single log, i.e. one topic partition. However a Kafka cluster will manage hundreds or thousands of these partitions. We attempt to balance partitions within a cluster in a round-robin fashion to avoid clustering all partitions for high-volume topics on a small number of nodes. Likewise we try to balance leadership so that each node is the leader for a proportional share of its partitions.

上面讨论的复制日志只说了单个日志，即一个topic的分区，然而，kafka集群需要管理成百上千的分区，我们试图用循环的方式在集群内平衡分区，以避免高容量高热度的主题的所有分区仅在少数几个节点上。同样，我们尽量使每个节点都是其分区按比例分担平衡的leader。

It is also important to optimize the leadership election process as that is the critical window of unavailability. A naive implementation of leader election would end up running an election per partition for all partitions a node hosted when that node failed. Instead, we elect one of the brokers as the "controller". This controller detects failures at the broker level and is responsible for changing the leader of all affected partitions in a failed broker. The result is that we are able to batch together many of the required leadership change notifications which makes the election process far cheaper and faster for a large number of partitions. If the controller fails, one of the surviving brokers will become the new controller.

同样重要的是优化leader选举的过程，一个傻的实现是当节点故障，leader将在运行中的所有分区中选举一个节点来托管。相反的，我们选出一个broker作为“控制器”。这个控制器检查broker级别故障和负责改变所有故障的broker中的受影响的leader的分区，这样的好处是，我们能够批量处理多个需要leader变更的分区，这使得选举更廉价、更快。如果控制器发生故障，在幸存的broker之中，将选举一个成为新的控制器。

**kafka日志压缩**

# 日志压缩

Log compaction ensures that Kafka will always retain at least the last known value for each message key within the log of data for a single topic partition. It addresses use cases and scenarios such as restoring state after application crashes or system failure, or reloading caches after application restarts during operational maintenance. Let's dive into these use cases in more detail and then describe how compaction works.

日志压缩确保kafka始终保留至少单个topic分区数据中每条消息key的最后的值。它解决了一些用例和场景，如应用程序崩溃或系统故障后还原状态，或应用程序在运行维护过程中重新启动后重新加载缓存。让我们深入讨论这些使用中的更多细节，描述它是如何压缩的。

So far we have described only the simpler approach to data retention where old log data is discarded after a fixed period of time or when the log reaches some predetermined size. This works well for temporal event data such as logging where each record stands alone. However an important class of data streams are the log of changes to keyed, mutable data (for example, the changes to a database table).

目前为止，我们只简单说了方法，旧的数据保留一段固定的时间，或当日志达到规定的大小后丢弃。这非常适用于暂时性事件数据，如日志记录，每个记录是独立的。而是一类重要的数据流，日志是变化的，可变的数据（例如：更改数据库表）。

Let's discuss a concrete example of such a stream. Say we have a topic containing user email addresses; every time a user updates their email address we send a message to this topic using their user id as the primary key. Now say we send the following messages over some time period for a user with id 123, each message corresponding to a change in email address (messages for other ids are omitted):

让我们来讨论一个关于流的具体的例子，假设我们有一个topic里包含用户的email地址，每次用户更新他们的email地址，我们发送一条消息到这个topic，使用用户Id作为主键。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

Log compaction gives us a more granular retention mechanism so that we are guaranteed to retain at least the last update for each primary key (e.g.bill@gmail.com). By doing this we guarantee that the log contains a full snapshot of the final value for every key not just keys that changed recently. This means downstream consumers can restore their own state off this topic without us having to retain a complete log of all changes.

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主键的最后一次更新 （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。

Let's start by looking at a few use cases where this is useful, then we'll see how it can be used.

让我们先看几个有用的用例，然后我们再看到怎么使用它。

1. *Database change subscription*. It is often necessary to have a data set in multiple data systems, and often one of these systems is a database of some kind (either a RDBMS or perhaps a new-fangled key-value store). For example you might have a database, a cache, a search cluster, and a Hadoop cluster. Each change to the database will need to be reflected in the cache, the search cluster, and eventually in Hadoop. In the case that one is only handling the real-time updates you only need recent log. But if you want to be able to reload the cache or restore a failed search node you may need a complete data set.  
   数据库更改订阅，通常需要在多个数据库系统，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仓库）。例如，你可能有一个数据库，缓存，搜索集群，以及Hadoop集群。每次变更数据库，也同时需要变更缓存，搜索集群，和Hadoop。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. *Event sourcing*. This is a style of application design which co-locates query processing with application design and uses a log of changes as the primary store for the application.  
   事件源。查询处理与应用设计共存，这是一种应用程序的设计风格，并使用一个变更日志作为应用程序的主仓库。
3. *Journaling for high-availability*. A process that does local computation can be made fault-tolerant by logging out changes that it makes to it's local state so another process can reload these changes and carry on if it should fail. A concrete example of this is handling counts, aggregations, and other "group by"-like processing in a stream query system. Samza, a real-time stream-processing framework, [uses this feature](http://samza.apache.org/learn/documentation/0.7.0/container/state-management.html) for exactly this purpose.  
   高可用的日志：本地的计算进程可以通过注销它自己的本地状态的变更进行容错。使另一个进程可以重载加载这些更改并继续执行（如果它故障）。例如：处理计数、聚合和其他的“group by”，- 像流查询系统。Samza，实时流处理框架，使用这个特性正是出于这一原因。

In each of these cases one needs primarily to handle the real-time feed of changes, but occasionally, when a machine crashes or data needs to be re-loaded or re-processed, one needs to do a full load. Log compaction allows feeding both of these use cases off the same backing topic. This style of usage of a log is described in more detail in [this blog post](http://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying).

在每一种情况下，首先需要去处理实时变更的feed（ps：新请求来的消息），但是偶尔，当机器崩溃或数据需要重新加载或重新处理时，需要做完整的加载。数据压缩允许feed这2种用例，这种风格的更详细的请看博客帖子。

The general idea is quite simple. If we had infinite log retention, and we logged each change in the above cases, then we would have captured the state of the system at each time from when it first began. Using this complete log we could restore to any point in time by replaying the first N records in the log. This hypothetical complete log is not very practical for systems that update a single record many times as the log will grow without bound even for a stable dataset. The simple log retention mechanism which throws away old updates will bound space but the log is no longer a way to restore the current state—now restoring from the beginning of the log no longer recreates the current state as old updates may not be captured at all.

通常想法是很简单，如果我们有无限的日志保留，我们记录每个变更，在上述的情况下，那么我们就从当它第一次开始每次捕获系统状态。保留完整日志，我们旧可以通过在日志重放第一个N记录来恢复到任意的时间点。这个假设的完整的日志对单条记录更新多次的系统是很不实用，即使是一个稳定的数据集，但日志将无线增长。一个简单的机制是扔掉旧日志，但是日志不能在恢复了。

Log compaction is a mechanism to give finer-grained per-record retention, rather than the coarser-grained time-based retention. The idea is to selectively remove records where we have a more recent update with the same primary key. This way the log is guaranteed to have at least the last state for each key.

日志压缩是一种机制给每条细粒度的保留，而不是基于时间的粗粒度的保留，是有选择地删除记录，我们保留相同的主键的最新记录。这种方式的日志保证至少有每个key的最后状态。

This retention policy can be set per-topic, so a single cluster can have some topics where retention is enforced by size or time and other topics where retention is enforced by compaction.

可以为每个topic设置保存策略，可以通过大小或时间，以及通过其他的压缩方式保存。

This functionality is inspired by one of LinkedIn's oldest and most successful pieces of infrastructure—a database changelog caching service called [Databus](https://github.com/linkedin/databus). Unlike most log-structured storage systems Kafka is built for subscription and organizes data for fast linear reads and writes. Unlike Databus, Kafka acts a source-of-truth store so it is useful even in situations where the upstream data source would not otherwise be replayable.

此功能的灵感来自LinkedIn最古老和最成功的基础设施 — 一个叫做Databus的 “数据库更新日志缓存服务”。不像大多数日志结构的存储系统，Kafka是专门为订阅和快速线性的读和写的组织数据，不同与Databus，kafka作为source-of-truth（真源：这里简单解释一下，消息发送到kafka这里，那么kafka里的消息就是最真的源了，因为如果kafka宕机了，从kafka的角度来讲，那kafka能自己恢复消息吗？不能，因为它不知道找谁，因此，kafka里面的消息就是真的源头数据），因此非常利于那些上游数据无法回放的情形。

### 日志压缩基础

Here is a high-level picture that shows the logical structure of a Kafka log with the offset for each message.

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。

The head of the log is identical to a traditional Kafka log. It has dense, sequential offsets and retains all messages. Log compaction adds an option for handling the tail of the log. The picture above shows a log with a compacted tail. Note that the messages in the tail of the log retain the original offset assigned when they were first written—that never changes. Note also that all offsets remain valid positions in the log, even if the message with that offset has been compacted away; in this case this position is indistinguishable from the next highest offset that does appear in the log. For example, in the picture above the offsets 36, 37, and 38 are all equivalent positions and a read beginning at any of these offsets would return a message set beginning with 38.

的kafka日志。它是密集的，连续offset，并保存所有的消息。日志压缩增加了一个选项来处理尾部(tail)的日志，上图显示了一个尾部压缩日志。另外，日志尾部已分配的消息将保留原来的偏移量 —— 永远不会改变，还要注意，在日志中所有的偏移量仍然保持有效的位置，即使消息已经压缩，在这种情况下，在日志的下一个最高的offset的位置是无法区分的。例如，上图的偏移量36，37和38都是等效的位置，读这些offset都将返回消息集的开始位置38。

Compaction also allows for deletes. A message with a key and a null payload will be treated as a delete from the log. This delete marker will cause any prior message with that key to be removed (as would any new message with that key), but delete markers are special in that they will themselves be cleaned out of the log after a period of time to free up space. The point in time at which deletes are no longer retained is marked as the "delete retention point" in the above diagram.

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。这个删除标记将导致删除在这个key之前的任何消息（以及该key的所有新消息）。但是删除标记是特殊的，他们自己去清理日志，在一段时间之后释放空间。在删除的时候不再保留标记作为“删除保存点”，如上图。

The compaction is done in the background by periodically recopying log segments. Cleaning does not block reads and can be throttled to use no more than a configurable amount of I/O throughput to avoid impacting producers and consumers. The actual process of compacting a log segment looks something like this:

压缩是在后台通过定期重新复制日志段来完成的。清洗不会阻塞读，可以限流I/O吞吐量（是可配置），以避免影响生产者和消费者。实际压缩处理日志看起来像这样：

### 日志压缩提供什么保障？

Log compaction guarantees the following:  
日志压缩保障如下：

1. Any consumer that stays caught-up to within the head of the log will see every message that is written; these messages will have sequential offsets. The topic's min.compaction.lag.ms can be used to guarantee the minimum length of time must pass after a message is written before it could be compacted. I.e. it provides a lower bound on how long each message will remain in the (uncompacted) head.

 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的使用min.compaction.lag.ms用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。也就是说，它提供了消息保留在head（未压缩）的最少时间。

1. Ordering of messages is always maintained. Compaction will never re-order messages, just remove some.  
   始终保持消息的排序。压缩永远不会重新排序消息，只是删除了一些。
2. The offset for a message never changes. It is the permanent identifier for a position in the log.  
   消息的偏移量永远不会改变。消息在日志中的位置将永久保存。
3. Any consumer progressing from the start of the log, will see at least the final state of all records in the order they were written. All delete markers for deleted records will be seen provided the consumer reaches the head of the log in a time period less than the topic'sdelete.retention.mssetting (the default is 24 hours). This is important as delete marker removal happens concurrently with read, and thus it is important that we do not remove any delete marker prior to the consumer seeing it.  
   从日志开始消费的所有消费者将至少看到其按顺序写入的最终状态的消息。此外，假如消费者在小于topic的delete.retention.ms setting设置的时间段（默认24小时）到达日志的head。将会看到所有已删除消息的删除标记。换句话说：由于删除日志与读取同时发生，消费者将优于删除。

### 日志压缩的细节

Log compaction is handled by the log cleaner, a pool of background threads that recopy log segment files, removing records whose key appears in the head of the log. Each compactor thread works as follows:  
日志cleaner处理日志压缩，后台线程池重新复制日志段文件，移除在日志head中出现的消息。每个压缩线程工作方式如下：

1. It chooses the log that has the highest ratio of log head to log tail  
   选择log head（日志头）到log tail（日志尾）比率最高的日志。
2. It creates a succinct summary of the last offset for each key in the head of the log  
   在head日志中为每个key的最后offset创建一个的简单概要。
3. It recopies the log from beginning to end removing keys which have a later occurrence in the log. New, clean segments are swapped into the log immediately so the additional disk space required is just one additional log segment (not a fully copy of the log).  
   它从日志的开始到结束，删除那些在日志中最新出现的key，新的，干净的段将立刻交换到日志中。因此，所需的额外磁盘空间只是一个额外的日志段（不是日志的完整副本）。
4. The summary of the log head is essentially just a space-compact hash table. It uses exactly 24 bytes per entry. As a result with 8GB of cleaner buffer one cleaner iteration can clean around 366GB of log head (assuming 1k messages).  
   日志head的概要本质上是一个空间密集型的哈希表，每个entry使用固定的24byte。这样8GB的cleaner buffer一次迭代可清理大约366GB的日志（假设消息1K）。

### 配置Log Cleaner

The log cleaner is enabled by default. This will start the pool of cleaner threads. To enable log cleaning on a particular topic you can add the log-specific property  
Log cleaner默认是启动的。也将启动cleaner线程池。你也可以针对特定topic启用log清洁，通过

log.cleanup.policy=compact

This can be done either at topic creation time or using the alter topic command.

可以在创建topic时或使用alter topic命令指定。

The log cleaner can be configured to retain a minimum amount of the uncompacted "head" of the log. This is enabled by setting the compaction time lag.  
log cleaner可以配置保留日志“head”不压缩的最小数。通过设置压缩延迟时间。

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

This can be used to prevent messages newer than a minimum message age from being subject to compaction. If not set, all log segments are eligible for compaction except for the last segment, i.e. the one currently being written to. The active segment will not be compacted even if all of its messages are older than the minimum compaction time lag.  
这可以预防消息在一个最小消息时间绝不会被压缩。如果不设置，除了最新的段，其他所有的段都是可以压缩的，即，当前正在写入的那个。即使其所有消息都比最小压缩时间滞后更长，正在写入的段也不会被压缩。

Further cleaner configurations are described [here](http://kafka.apache.org/documentation.html#brokerconfigs).  
关于cleaner更详细的配置在这里。

**kafka配额**

# 4.9 配额

从0.9开始，Kafka集群能够对生产和消费设置配额。为每个客户端分组设置配额阈值（基于字节比率）。

Kafka集群有能力对请求进行配额来控制客户端使用的broker资源。可以为共享配额的每个客户组执行两种类型的客户配额：

1. 通过配额定义网络带宽的字节率阈值（从0.9版本开始）
2. 请求率配额将CPU的利用率阈值定义为网络和I/O线程的百分比（自0.11版本起）

## 为什么需要配额？

生产者和消费者的可能生产/消费非常大量的数据，从而垄断了broker的资源，引起网络饱和，配额可防止这个问题。在大型多节点集群中更加重要。其中有一小部分不良行为的用户将被降权。事实上，当kafka作为服务运行时，可以根据约定好的协议执行API限制。

## 客户端集群

在安全的集群中，Kafka客户端通过已认证的用户的principal来标识的。在一个无需认证客户端的集群中，用户principal是一个未认证的分组。用户的principal是通过broker（使用可配置的PrincipalBuilder）。Client-id是客户端应用程序使用具有意义名称的客户端的逻辑分组。元组（user，client-id）定义了共享用户principal和client-id的客户端安全逻辑分组。

配额可以应用到（user，client-id），用户或clinet-id组中。为了一个给定的连接，应用与连接匹配的最具体的配额。所有的配额分组的连接  
共享配置的分组配额。例如，如果（user=“test=user”,client-id="test-client"）生产配额是10MB/sec，那么将会应用到所有生产者用户是“test-user”和clinet-id“是test-client”的实例上。

## 配额配置

为user和client-id分组定义配额配置。可根据自身需要去覆盖默认的配置，这个机制类似于topic日志配置覆盖。用户和（user, client-id）配额覆盖写在ZooKeeper的/config/users下,client-id配额覆盖写在/config/clients下。这些配置被所有broker读取，并立即生效。并且我们更改配置而无需重启整个集群。点击这里查看更多细节。每个分组默认的配额也可使用相同的机制来动态地更新。

配额配置的优先级顺序为：

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>

可以通过Broker配置（quota.producer.default, quota.consumer.default）为client-id分组设置默认的网络带宽配额。但已不赞成使用，并将在后面的版本移除。client-id的默认的配额可以在zookeeper设置（类似其他的配额覆盖和默认）。

## 网络带宽配额

网络带宽配额被定义为的字节速率阈值（客户端的每个分组共享的配额）。默认情况下，每个独立的客户端分组按照集群的配置接收固定的配额（字节/秒）。这个配额是基于每个broker上的定义。每个客户端分组在客户端被限制之前发布/获取每个broker的最大X字节/秒。

## 请求的比率配额

请求比率配额定义为一个客户端可以请求利用配额窗口中的每个broker的请求处理I/O线程和网络线程的时间百分比。n%的配额代表一个线程的n%，因此配额超出 ((num.io.threads + num.network.threads) \* 100)%的总容量。在限制之前，每个客户端分组可能在配额窗口的所有I/O和网络线程使用高达n%的总百分比。由于分配给I/O和网络线程的线程数通常基于broker主机上可用的核心数量（CPU核心数），所以请求率配额代表共享配额的每组客户端可能使用的CPU的总百分比。

## 强制执行

默认情况下，每个独立的客户端分组接收一个固定的配额（集群配置的）。该配额基于每个broker定义的。每个客户端在被限制之前都使用该配额。我们决定，为每个broker定义的这些配额比固定的集群带宽要好得多，因为需要一个机制在所有broker中共享客户端配额。这可能比配额实现本身更难！

当检测到配额违规时，broker该做出什么样的反应？在我们的解决方案中，broker不返回错误，而是尝试减慢超出其配额的客户端。它计算延迟量，使违规的客户端根据其配额并延迟该时段的响应时间。这个方法保持对客户端违法配额透明（除client度量）。这也使他们不必实现任何特殊的回退或重试（否则可能会变得很麻烦）。 事实上，坏的客户端行为（不重试回退）可能加速尝试解决的配额问题。

字节率和线程利用率是通过多个小窗口（例如每个1秒的30个窗口）来测量，以便快速检测和纠正配额违规。 通常，具有大的测量窗口（例如，每个30秒的10个窗口）导致大量的流量突发，随后引起长时间的延迟，这在用户体验方面不好。

**kafka接口设计**

# 5.1 API设计

### 生产者API

The Producer API that wraps the 2 low-level producers - kafka.producer.SyncProducer and kafka.producer.async.AsyncProducer.

生产者API，它封装了2个低级别的生产者 - **kafka.producer.SyncProducer**  和  **kafka.producer.async.AsyncProducer**。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | class Producer {    /\* Sends the data, partitioned by key to the topic using either the \*/    /\* synchronous or the asynchronous producer \*/    public void send(kafka.javaapi.producer.ProducerData<K,V> producerData);      /\* Sends a list of data, partitioned by key to the topic using either \*/    /\* the synchronous or the asynchronous producer \*/    public void send(java.util.List<kafka.javaapi.producer.ProducerData<K,V>> producerData);      /\* Closes the producer and cleans up \*/    public void close();    } |

The goal is to expose all the producer functionality through a single API to the client. The new producer -  
通过API提供给客户端，来暴露生产者所有的功能。新的生产者 -

* can handle queueing/buffering of multiple producer requests and asynchronous dispatch of the batched data -  
  可以处理多个生产者请求和异步批量数据派发的队列/缓冲 -

**kafka.producer.Producer** provides the ability to batch multiple produce requests **(producer.type=async)**, before serializing and dispatching them to the appropriate kafka broker partition. The size of the batch can be controlled by a few config parameters. As events enter a queue, they are buffered in a queue, until either **queue.time** or **batch.sizeis** reached. A background thread (**kafka.producer.async.ProducerSendThread**) dequeues the batch of data and lets the **kafka.producer.EventHandler** serialize and send the data to the appropriate kafka broker partition. A custom event handler can be plugged in through the **event.handler** config parameter. At various stages of this producer queue pipeline, it is helpful to be able to inject callbacks, either for plugging in custom logging/tracing code or custom monitoring logic. This is possible by implementing the **kafka.producer.async.CallbackHandler** interface and setting **callback.handler** config parameter to that class.  
**kafka.producer.Producer** 提供批处理多个生产请求的能力（**producer.type=async**），序列化和派发到broker分区之前，可以配置参数控制批量的大小。随着事件进入队列，缓存在队列，直到满足 queue.time 或 batch.size。后台线程（**kafka.producer.async.ProducerSendThread**）发送一批数据用**kafka.producer.EventHandler**序列化并发送到broker分区。还可以通过**event.handler**配置参数可以插入自定义的事件处理。生产者队列管道在不同的阶段，无论是插入自动的日志记录/跟踪代码或自定义的监控逻辑，能够注入回调，通过实现**kafka.producer.async.CallbackHandler**接口和设置**callback.handler**配置参数的类。

* handles the serialization of data through a user-specified Encoder-   
  处理数据序列化，通过用户指定的 **Encoder**-

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

* The default is the no-op **kafka.serializer.DefaultEncoder**  
  默认是**空操作kafka.serializer.DefaultEncoder**
* provides software load balancing through an optionally user-specified Partitioner-  
  提供平衡负载，通过用户指定的**Partitioner**-

The routing decision is influenced by the kafka.producer.Partitioner.  
路由决定由kafka.producer.Partitioner影响。

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

The partition API uses the key and the number of available broker partitions to return a partition id. This id is used as an index into a sorted list of broker\_ids and partitions to pick a broker partition for the producer request. The default partitioning strategy ishash(key)%numPartitions. If the key is null, then a random broker partition is picked. A custom partitioning strategy can also be plugged in using thepartitioner.classconfig parameter.  
该分区API，使用key和可用broker分区数，返回一个分区ID。这个id用作索引broker\_ids和分区排序列表来为生产者请求挑选一个broker分区。默认的分区策略是hash（key）% numPartitions。如果key是空，则随机broker分区，还可以插入自定义分区策略使用partitioner.class配置参数。

### 消费者API

We have 2 levels of consumer APIs. The low-level "simple" API maintains a connection to a single broker and has a close correspondence to the network requests sent to the server. This API is completely stateless, with the offset being passed in on every request, allowing the user to maintain this metadata however they choose.  
我们有两个层次的消费者API，低级别的“简单”API保持一个broker的连接，并有密切通讯网络发送到服务器的请求。这个API完全无状态的，偏移量被传递到每个请求，允许用户保持这个元数据。

The high-level API hides the details of brokers from the consumer and allows consuming off the cluster of machines without concern for the underlying topology. It also maintains the state of what has been consumed. The high-level API also provides the ability to subscribe to topics that match a filter expression (i.e., either a whitelist or a blacklist regular expression).  
高级别API封装broker消费具体细节。它维持已消费的状态，高级别API还提供了通过匹配表达式（或过滤）的方式获取指定topic（即，无论是白名单或黑名单的正则表达式）。

##### 低级别API

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | class SimpleConsumer {      /\* Send fetch request to a broker and get back a set of messages. \*/    public ByteBufferMessageSet fetch(FetchRequest request);      /\* Send a list of fetch requests to a broker and get back a response set. \*/    public MultiFetchResponse multifetch(List<FetchRequest> fetches);      /\*\*     \* Get a list of valid offsets (up to maxSize) before the given time.     \* The result is a list of offsets, in descending order.     \* @param time: time in millisecs,     \*              if set to OffsetRequest$.MODULE$.LATIEST\_TIME(), get from the latest offset available.     \*              if set to OffsetRequest$.MODULE$.EARLIEST\_TIME(), get from the earliest offset available.     \*/    public long[] getOffsetsBefore(String topic, int partition, long time, int maxNumOffsets);  } |

The low-level API is used to implement the high-level API as well as being used directly for some of our offline consumers (such as the hadoop consumer) which have particular requirements around maintaining state.  
低级别的API用于实现高级别的API，以及直接用于一些有特定需求的离线的消费者。

##### 高级别API

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | /\* create a connection to the cluster \*/  ConsumerConnector connector = Consumer.create(consumerConfig);    interface ConsumerConnector {      /\*\*     \* This method is used to get a list of KafkaStreams, which are iterators over     \* MessageAndMetadata objects from which you can obtain messages and their     \* associated metadata (currently only topic).     \*  Input: a map of <topic, #streams>     \*  Output: a map of <topic, list of message streams>     \*/    public Map<String,List<KafkaStream>> createMessageStreams(Map<String,Int> topicCountMap);      /\*\*     \* You can also obtain a list of KafkaStreams, that iterate over messages     \* from topics that match a TopicFilter. (A TopicFilter encapsulates a     \* whitelist or a blacklist which is a standard Java regex.)     \*/    public List<KafkaStream> createMessageStreamsByFilter(        TopicFilter topicFilter, int numStreams);      /\* Commit the offsets of all messages consumed so far. \*/    public commitOffsets()      /\* Shut down the connector \*/    public shutdown()  } |

This API is centered around iterators, implemented by the KafkaStream class. Each KafkaStream represents the stream of messages from one or more partitions on one or more servers. Each stream is used for single threaded processing, so the client can provide the number of desired streams in the create call. Thus a stream may represent the merging of multiple server partitions (to correspond to the number of processing threads), but each partition only goes to one stream.

这个API是围绕中心的迭代器，KafkaStream类实现的，每个KafkaStream代表一个或多个服务器上的多个分区的消息流，每个流都是单个线程处理，因此客户可以提供所有要的数量创建调用流，因此，流可能代表多个服务器分区的合并（对应处理线程的数量），但每个分区只能进入一个流。

The createMessageStreams call registers the consumer for the topic, which results in rebalancing the consumer/broker assignment. The API encourages creating many topic streams in a single call in order to minimize this rebalancing. The createMessageStreamsByFilter call (additionally) registers watchers to discover new topics that match its filter. Note that each stream that createMessageStreamsByFilter returns may iterate over messages from multiple topics (i.e., if multiple topics are allowed by the filter).

createMessageStreams调用注册消费者的topic，这将导致重新平衡消费者/broker分配。API鼓励创建单个调用多个的topic流，以尽量减少这种再平衡。createMessageStreamsByFilter调用（另外）登记观察者发现符合过滤器的新topic。需要注意的是，每个流的createMessageStreamsByFilter返回的消息可能遍历了多个topic（既，如果是过滤器允许多个topic）。

**kafka网络层**

# 5.2 网络层

The network layer is a fairly straight-forward NIO server, and will not be described in great detail. The sendfile implementation is done by giving the MessageSet interface a writeTo method. This allows the file-backed message set to use the more efficient transferTo implementation instead of an in-process buffered write. The threading model is a single acceptor thread and *N* processor threads which handle a fixed number of connections each. This design has been pretty thoroughly tested [elsewhere](http://sna-projects.com/blog/2009/08/introducing-the-nio-socketserver-implementation) and found to be simple to implement and fast. The protocol is kept quite simple to allow for future implementation of clients in other languages.

网络层是一个很直接的NIO服务。这里不详细说明。sendfile实现是MessageSet接口的writeTo方法。 允许文件备份消息集使用更有效的transferTo实现而不是在过程中使用缓存写入。线程模型是一个单接收线程和处理每个连接的N个处理器线程。这种设计已经在别的地方得到充分验证，发现实现简单、快速。该协议很简单，方便以后客户端用其他语言来实现。

**kafka消息格式**

## 5.3 kafka消息格式

消息（又名记录）始终是按批次写入。一批消息用技术术语表达就是记录批次，记录批次包含一个或多个记录。 在低性能的情况下，一个批次只有单条消息。记录批次和记录都有自己的头文件。下面介绍了Kafka版本0.11.0及更高版本（消息格式版本v2或magic = 2）的格式。点击[此处](#https://cwiki.apache.org/confluence/dis)查看有关邮件格式0和1的详细信息。

#### 5.3.1 消息批次

以下是RecordBatch的磁盘格式。

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，当启用压缩时，压缩记录数据将按照记录数的计数直接序列化的。

CRC涵盖从属性到批次结束的数据（即。CRC之后的所有的字节）。它位于magic字节之后，也就是说，在决定如何解析批次长度和magic字节之前，客户端必须解析magic字节。分区leader epoch字段不包括在CRC计算中，以避免broker接收的每个批次分配该字段时，重新计算CRC。CRC-32C (Castagnoli)多项式被用于计算。

压缩：与老的消息格式不同，magic v2及以上版本在日志清理时，保留原始批次中的第一个和最后一个offset/sequence号。为了当日志重新加载时能恢复生产者的状态。这个功能是必须的，假设，如果我们不保留最后的序列数。如果分区leader故障，则生产者将看到OutOfSequence错误。必须保留基本序列号以进行重复检查（broker通过验证传入批次的第一个和最后一个序列号与该生产者的最后一个序列号相匹配）。因此，当清除批次中的所有消息，以保留生产者的最后序列号时，可以在日志中具有空批次。这里有一个奇怪的是，在压缩过程中，baseTimestamp字段不保留。所以如果批次中第一条消息被压缩，那么它将改变。

#### 5.3.1.1 控制批次

控制批次包含一个称为控制记录的记录。控制记录不传递给应用程序。相反，它们被消费者用来过滤掉中止的事务消息。

控制记录的关键是符合以下模式：

version: int16 (current version is 0)

type: int16 (0 indicates an abort marker, 1 indicates a commit)

控制记录的值的模式取决于类型。 该值对客户端是不透明的。

#### 5.3.2 记录

记录级header在Kafka 0.11.0中引入。具有Header的记录的磁盘格式如下所示。

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 记录的header

headerKeyLength: varint

headerKey: String

headerValueLength: varint

Value: byte[]

我们使用与Protobuf相同的varint编码。有关后者的更多信息，请参见[这里](https://developers.google.com/protocol-buffers/docs/encoding#varints)。记录中的header计数也被编码为一个varint。

**kafka日志**

# 5.5 日志

A log for a topic named "my\_topic" with two partitions consists of two directories (namely my\_topic\_0 and my\_topic\_1) populated with data files containing the messages for that topic. The format of the log files is a sequence of "log entries""; each log entry is a 4 byte integer *N* storing the message length which is followed by the *N* message bytes. Each message is uniquely identified by a 64-bit integer *offset* giving the byte position of the start of this message in the stream of all messages ever sent to that topic on that partition. The on-disk format of each message is given below. Each log file is named with the offset of the first message it contains. So the first file created will be 00000000000.kafka, and each additional file will have an integer name roughly *S* bytes from the previous file where *S* is the max log file size given in the configuration.

假设有2个分区的主题“my\_topic”，它将由2个目录构成（my\_topic\_0和my\_topic\_1），用于存放该主题消息的数据文件。日志文件的格式是一个“日志条目”序列。每条日志条目都由一个存储消息长度的4字节整型N和紧跟着的N字节消息组成。其中每条消息都有一个64位整型的唯一标识offset，offset（偏移量）代表了topic分区中所有消息流中该消息的起始字节位置。每条消息在磁盘上的格式如下：每个日志文件用第一条消息的offset来命名的，因此，创建的第一个文件将是00000000000.kafka，并且每个附加文件都将是上一个文件S字节的整数命名，其中S是配置中设置的最大日志文件大小。

The exact binary format for messages is versioned and maintained as a standard interface so message sets can be transfered between producer, broker, and client without recopying or conversion when desirable. This format is as follows:  
消息是二进制格式并作为一个标准接口，所以消息可以在producer，broker，client之间传输，无需再copy或转换。格式如下:

On-disk format of a message

message length : 4 bytes (value: 1+4+n)

"magic" value : 1 byte

crc : 4 bytes

payload : n bytes

The use of the message offset as the message id is unusual. Our original idea was to use a GUID generated by the producer, and maintain a mapping from GUID to offset on each broker. But since a consumer must maintain an ID for each server, the global uniqueness of the GUID provides no value. Furthermore the complexity of maintaining the mapping from a random id to an offset requires a heavy weight index structure which must be synchronized with disk, essentially requiring a full persistent random-access data structure. Thus to simplify the lookup structure we decided to use a simple per-partition atomic counter which could be coupled with the partition id and node id to uniquely identify a message; this makes the lookup structure simpler, though multiple seeks per consumer request are still likely. However once we settled on a counter, the jump to directly using the offset seemed natural—both after all are monotonically increasing integers unique to a partition. Since the offset is hidden from the consumer API this decision is ultimately an implementation detail and we went with the more efficient approach.

使用消息offset作为消息id是不常见的，我们最初的想法是使用由生产者生成的GUID，并维护GUID到每个broker的offset映射。但是消费者必须维护每个服务ID，独一无二的GUID，另外，维护来自随机id的映射到一个offset的复杂度，需要一个非常复杂的索引结构，还必须与磁盘同步，基本上需要一个完整的持久性随机存储数据结构。因此，为了简化查找结构，我们决定使用一个简单的每个分区的原子计数器，它可以加上分区id和节点id来唯一标识一个消息；这使得查询结构更简单，虽然每个消费者仍然可能需要查找多个。然而，我们一旦选定了一个counter（计数器），直接跳到使用offset — 两者毕竟都是单纯递增到唯一的整数分区。由于offset在consumer API是隐藏的，这个最终的实现细节和我们用更有效的方法。

### 写

The log allows serial appends which always go to the last file. This file is rolled over to a fresh file when it reaches a configurable size (say 1GB). The log takes two configuration parameter *M* which gives the number of messages to write before forcing the OS to flush the file to disk, and *S* which gives a number of seconds after which a flush is forced. This gives a durability guarantee of losing at most *M* messages or *S* seconds of data in the event of a system crash.  
日志允许串行的追加消息到最后的一个文件。当它达到配置文件中设置的大小（1GB），就会滚动新的文件上。日志采用了2个配置参数：M，它定义了强制OS刷新文件到磁盘之前主动写入的消息数量。S，它定义了几秒后强制刷新。这样提供了耐久性的保障。当系统崩溃时候，丢失最多M消息，或S秒的数据。

### 读取

Reads are done by giving the 64-bit logical offset of a message and an *S*-byte max chunk size. This will return an iterator over the messages contained in the *S*-byte buffer. *S* is intended to be larger than any single message, but in the event of an abnormally large message, the read can be retried multiple times, each time doubling the buffer size, until the message is read successfully. A maximum message and buffer size can be specified to make the server reject messages larger than some size, and to give a bound to the client on the maximum it need ever read to get a complete message. It is likely that the read buffer ends with a partial message, this is easily detected by the size delimiting.

读取是通过定义的64位逻辑的消息和S-byte块大小的offset来完成。返回一个迭代器，它包含在S-byte缓冲区的消息。S比单个消息大，但是在消息很大的情况下，读取可重试多次，每次的缓冲区大小加倍，直到消息被成功的读取。可以指定最大消息和缓冲区的大小，使服务器拒绝一些超过这个大小的消息。

The actual process of reading from an offset requires first locating the log segment file in which the data is stored, calculating the file-specific offset from the global offset value, and then reading from that file offset. The search is done as a simple binary search variation against an in-memory range maintained for each file.  
从一个offset读取的实际过程中，首先需要在存储的数据中找出日志段文件，然后通过全局offset计算找到的日志段内的offset。然后从该文件的offset读取数据。搜索是通过二进制搜索每个文件在内存中的变化来完成的。

The log provides the capability of getting the most recently written message to allow clients to start subscribing as of "right now". This is also useful in the case the consumer fails to consume its data within its SLA-specified number of days. In this case when the client attempts to consume a non-existant offset it is given an OutOfRangeException and can either reset itself or fail as appropriate to the use case.  
日志提供获取最新写的消息来允许客户端开始在“right now”订阅的能力，这是在其SLA指定的天数内未消费的情况下是很有用的。在这种情况下，当客户端尝试去消费一个不存在的offset，将报OutOfRangeException，并重置它自己，或在适当的情况下直接失败。

The following is the format of the results sent to the consumer.  
下面是发送给消费者的结果的格式。

MessageSetSend (fetch result)

total length : 4 bytes

error code : 2 bytes

message 1 : x bytes

...

message n : x bytes

MultiMessageSetSend (multiFetch result)

total length : 4 bytes

error code : 2 bytes

messageSetSend 1

...

messageSetSend n

#### 删除

Data is deleted one log segment at a time. The log manager allows pluggable delete policies to choose which files are eligible for deletion. The current policy deletes any log with a modification time of more than *N* days ago, though a policy which retained the last *N* GB could also be useful. To avoid locking reads while still allowing deletes that modify the segment list we use a copy-on-write style segment list implementation that provides consistent views to allow a binary search to proceed on an immutable static snapshot view of the log segments while deletes are progressing.  
数据删除在一个时间的日志段。日志管理器允许插入删除策略来选择删除哪些文件，目前的策略删除N天以前日志（修改时间），虽然它保留了最后的N，GB也可能是有用策略。为了避免锁定读取，同时仍然允许删除和修改段列表，我们使用一个copy-on-write风格的段列表实现， 提供一致的视图来允许一个二叉搜索进行一个不变的日志段的静态快照视图同时进行删除。

### 保障（Guarantees）

The log provides a configuration parameter *M* which controls the maximum number of messages that are written before forcing a flush to disk. On startup a log recovery process is run that iterates over all messages in the newest log segment and verifies that each message entry is valid. A message entry is valid if the sum of its size and offset are less than the length of the file AND the CRC32 of the message payload matches the CRC stored with the message. In the event corruption is detected the log is truncated to the last valid offset.

日志提供了一个配置参数M，它来控制消息在强制刷新到磁盘之前，就写入磁盘的最大数。在启动日志恢复进程运行，超过了最新的日志段的所有消息迭代并验证每条消息是有效的。如果消息的总大小并且offset小于文件的长度和消息有效负载CRC32匹配消息存储CRC的消息条目，则是有效的。如果检索到脏日志则截取最后有效的offset。

Note that two kinds of corruption must be handled: truncation in which an unwritten block is lost due to a crash, and corruption in which a nonsense block is ADDED to the file. The reason for this is that in general the OS makes no guarantee of the write order between the file inode and the actual block data so in addition to losing written data the file can gain nonsense data if the inode is updated with a new size but a crash occurs before the block containing that data is not written. The CRC detects this corner case, and prevents it from corrupting the log (though the unwritten messages are, of course, lost).  
需要注意的是两种腐败必须处理：截断由崩溃导致未写入的块丢失。无意义的块被添加到文件的脏数据，这么做的原因是，在一般的操作系统是没有文件节点和实际数据块之间写入顺序的保障，所以除了丢失写入的数据，如果该节点新size更新，但块包含写入前崩溃产生的无用数据。CRC发现这种问题，并阻止脏数据（虽然未写入消息，当然，丢失）

**kafka分布**

### ****消费者offset跟踪 （Consumer Offset Tracking）****

The high-level consumer tracks the maximum offset it has consumed in each partition and periodically commits its offset vector so that it can resume from those offsets in the event of a restart. Kafka provides the option to store all the offsets for a given consumer group in a designated broker (for that group) called the *offset manager*. i.e., any consumer instance in that consumer group should send its offset commits and fetches to that offset manager (broker). The high-level consumer handles this automatically. If you use the simple consumer you will need to manage offsets manually. This is currently unsupported in the Java simple consumer which can only commit or fetch offsets in ZooKeeper. If you use the Scala simple consumer you can discover the offset manager and explicitly commit or fetch offsets to the offset manager. A consumer can look up its offset manager by issuing a ConsumerMetadataRequest to any Kafka broker and reading the ConsumerMetadataResponse which will contain the offset manager. The consumer can then proceed to commit or fetch offsets from the offsets manager broker. In case the offset manager moves, the consumer will need to rediscover the offset manager. If you wish to manage your offsets manually, you can take a look at these [code samples that explain how to issue OffsetCommitRequest and OffsetFetchRequest](https://cwiki.apache.org/confluence/display/KAFKA/Committing+and+fetching+consumer+offsets+in+Kafka).

高级别消费者跟踪每个分区已消费的最大的offset，并定期提交offset，在重新启动的情况下，可从这些offset恢复。Kafka提供了一个选项在指定的broker中来存储所有给定的消费者组的offset，称为offset manager。例如，该消费者组的所有消费者实例发送其offset，提交并获取该offset manager（broker）。高级别消费者都将会自动处理这些。如果你使用低级别的消费者，你将需要去手动管理offset。目前在低级别的java消费者不支持，只能在Zookeeper提交或获取offset。如果你使用简单的Scala消费者，你可拿到offset manager，并显式的提交或获取offset。消费者可以通过发送GroupCoordinatorRequest到任何的broker，并接受GroupCoordinatorResponse响应对象，对象包含offset manager，那么消费者可以继续从`offset manager broker`提交或获取offset。如果offset manager位置变动，消费者需要重新发现offset manager。如果你想手动管理你的offset，你可以看看OffsetCommitRequest 和 OffsetFetchRequest如何做的。

When the offset manager receives an OffsetCommitRequest, it appends the request to a special [compacted](http://kafka.apache.org/documentation.html#compaction) Kafka topic named *\_\_consumer\_offsets*. The offset manager sends a successful offset commit response to the consumer only after all the replicas of the offsets topic receive the offsets. In case the offsets fail to replicate within a configurable timeout, the offset commit will fail and the consumer may retry the commit after backing off. (This is done automatically by the high-level consumer.) The brokers periodically compact the offsets topic since it only needs to maintain the most recent offset commit per partition. The offset manager also caches the offsets in an in-memory table in order to serve offset fetches quickly.

当offset manager接收一个OffsetCommitRequest，它追加请求到一个特定的压缩的名为\_\_consumer\_offsets的topic中，当offset topic的所有副本接收offset之后，offset manager发送一个成功的offset提交响应给消费者。万一offset无法在规定的时间内复制，offset将提交失败，消费者在回退之后可重试该提交（高级别消费者自动做的）。broker定期压缩offset topic，只需要保存每个分区最近的offset。offset manager也缓存offset在内存表中，以便offset快速获取。

When the offset manager receives an offset fetch request, it simply returns the last committed offset vector from the offsets cache. In case the offset manager was just started or if it just became the offset manager for a new set of consumer groups (by becoming a leader for a partition of the offsets topic), it may need to load the offsets topic partition into the cache. In this case, the offset fetch will fail with an OffsetsLoadInProgress exception and the consumer may retry the OffsetFetchRequest after backing off. (This is done automatically by the high-level consumer.)

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刚启动或新的消费者组集刚成为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，消费者可后退后，重试OffsetFetchRequest（高级别消费者自动做这些）。

### 迁移offset从zookeeper到kafka

Kafka consumers in earlier releases store their offsets by default in ZooKeeper. It is possible to migrate these consumers to commit offsets into Kafka by following these steps:

Kafka消费者在早先的版本中offset默认存储在ZooKeeper。可以通过下面的步骤迁移这些消费者到Kafka。

1. Set offsets.storage=kafka and dual.commit.enabled=true in your consumer config.  
   在消费者配置设置offsets.storage=kafka和dual.commit.enabled=true。
2. Do a rolling bounce of your consumers and then verify that your consumers are healthy.  
   消费者做滚动消费，验证你的消费者是健康的。
3. Set dual.commit.enabled=false in your consumer config.  
   在你的消费者配置设置dual.commit.enabled=false
4. Do a rolling bounce of your consumers and then verify that your consumers are healthy.  
   消费者做滚动消费，验证你的消费者是健康的。

A roll-back (i.e., migrating from Kafka back to ZooKeeper) can also be performed using the above steps if you setoffsets.storage=zookeeper.

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

The following gives the ZooKeeper structures and algorithms used for co-ordination between consumers and brokers.

下面给出了Zookeeper的结构和算法，用于协调消费者和经纪人之间。

### 符号（Notation）

When an element in a path is denoted [xyz], that means that the value of xyz is not fixed and there is in fact a ZooKeeper znode for each possible value of xyz. For example /topics/[topic] would be a directory named /topics containing a sub-directory for each topic name. Numerical ranges are also given such as [0...5] to indicate the subdirectories 0, 1, 2, 3, 4. An arrow -> is used to indicate the contents of a znode. For example /hello -> world would indicate a znode /hello containing the value "world".

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 Node Registry

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

This is a list of all present broker nodes, each of which provides a unique logical broker id which identifies it to consumers (which must be given as part of its configuration). On startup, a broker node registers itself by creating a znode with the logical broker id under /brokers/ids. The purpose of the logical broker id is to allow a broker to be moved to a different physical machine without affecting consumers. An attempt to register a broker id that is already in use (say because two servers are configured with the same broker id) is an error.

这是所有当前broker的节点列表，其中每个提供了一个唯一的逻辑broker的id标识它的消费者（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，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）。

Since the broker registers itself in ZooKeeper using ephemeral znodes, this registration is dynamic and will disappear if the broker is shutdown or dies (thus notifying consumers it is no longer available).

由于broker在Zookeeper中用的是临时znode，这个注册是动态的，如果broker关闭或宕机，节点将消失（通知消费者不在可用）。

### Broker Topic Registry

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

Each broker registers itself under the topics it maintains and stores the number of partitions for that topic.

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

Consumers of topics also register themselves in ZooKeeper, in order to coordinate with each other and balance the consumption of data. Consumers can also store their offsets in ZooKeeper by settingoffsets.storage=zookeeper. However, this offset storage mechanism will be deprecated in a future release. Therefore, it is recommended to [migrate offsets storage to Kafka](http://kafka.apache.org/documentation.html#offsetmigration).

topic的消费者也在zookeeper注册他们自己，以便相互协调和平衡数据的消耗，消费者还可以通过设置offsets.storage=zookeeper来存储offset，但是，这个机制在未来的版本将会弃用。因此，建议迁移数据到kafka。

Multiple consumers can form a group and jointly consume a single topic. Each consumer in the same group is given a shared group\_id. For example if one consumer is your foobar process, which is run across three machines, then you might assign this group of consumers the id "foobar". This group id is provided in the configuration of the consumer, and is your way to tell the consumer which group it belongs to.  
多个消费者可组成一组，共同消费一个topic，在同一组中的每个消费者共享一个group\_id。例如，如果一个消费者是foobar，在三个机器上运行，你可能分配这个这个消费组的ID是“foobar”。这个组id是在消费者的配置文件中配置。

The consumers in a group divide up the partitions as fairly as possible, each partition is consumed by exactly one consumer in a consumer group.  
每个分区正好被一个消费者组的消费者所消费，一组中的消费者尽可能公平地分配分区。

### Consumer Id Registry

In addition to the group\_id which is shared by all consumers in a group, each consumer is given a transient, unique consumer\_id (of the form hostname:uuid) for identification purposes. Consumer ids are registered in the following directory.  
除了由所有消费者共享的group\_id，每个消费者都有一个临时，唯一的consumer\_id（主机名的形式:uuid）用于识别。消费者的id在注册到在以下目录中。

/consumers/[group\_id]/ids/[consumer\_id] --> {"topic1": #streams, ..., "topicN": #streams} (ephemeral node)

Each of the consumers in the group registers under its group and creates a znode with its consumer\_id. The value of the znode contains a map of <topic, #streams>. This id is simply used to identify each of the consumers which is currently active within a group. This is an ephemeral node so it will disappear if the consumer process dies.  
组中的每个消费者用consumer\_id注册znode。znode的值包含一个map<topic,#streams>。这个id只是用来识别在组里目前活跃的消费者，这是个临时节点，如果消费者在处理中挂掉，它就会消失。

### Consumer Offsets

Consumers track the maximum offset they have consumed in each partition. This value is stored in a ZooKeeper directory ifoffsets.storage=zookeeper. This valued is stored in a ZooKeeper directory.  
消费者追踪它们在每个分区消费的最大offset，如果offsets.storage=zookeeper，那此值就存在zookeeper的目录中。

/consumers/[group\_id]/offsets/[topic]/[broker\_id-partition\_id] --> offset\_counter\_value ((persistent node)

#### Partition Owner registry

Each broker partition is consumed by a single consumer within a given consumer group. The consumer must establish its ownership of a given partition before any consumption can begin. To establish its ownership, a consumer writes its own id in an ephemeral node under the particular broker partition it is claiming.  
每个broker分区由一个消费者组里的单个消费者消费，任何消费者开始消费之前，消费者必须建立其所有权，为了建立所有权，消费者写入自己的ID到临时节点 。

/consumers/[group\_id]/owners/[topic]/[broker\_id-partition\_id] --> consumer\_node\_id (ephemeral node)

### Cluster Id（集群ID）

The cluster id is a unique and immutable identifier assigned to a Kafka cluster. The cluster id can have a maximum of 22 characters and the allowed characters are defined by the regular expression [a-zA-Z0-9\_\-]+, which corresponds to the characters used by the URL-safe Base64 variant with no padding. Conceptually, it is auto-generated when a cluster is started for the first time.

集群ID是Kafka集群中唯一且不可变的标识符。集群ID最多可以包含22个字符，对应于没有填充的URL安全Base64变量所使用的字符，允许通过正则表达式 [a-zA-Z0-9 \_ \]]定义。从概念上讲，当第一次启动集群时，集群ID就会自动生成。

Implementation-wise, it is generated when a broker with version 0.10.1 or later is successfully started for the first time. The broker tries to get the cluster id from the /cluster/id znode during startup. If the znode does not exist, the broker generates a new cluster id and creates the znode with this cluster id.

实际上，当第一次成功启动时生成（0.10.1或更高版本）。broker尝试在启动期间从 /cluster/id znode获取集群ID。 如果znode不存在，broker将生成一个新的集群ID，并使用此集群ID创建znode。

### Broker node registration

The broker nodes are basically independent, so they only publish information about what they have. When a broker joins, it registers itself under the broker node registry directory and writes information about its host name and port. The broker also register the list of existing topics and their logical partitions in the broker topic registry. New topics are registered dynamically when they are created on the broker.  
broker节点基本上都是独立的， 所以它们只发布有关它们的信息，当broker连接，注册broker节点到其注册目录下，并写入它的host name和prot信息。broker还注册了其注册的现有topic和逻辑分区的列表。当创建一个新topic，就在broker上动态的注册新的topic。

### Consumer registration algorithm

When a consumer starts, it does the following:  
当一个消费者启动，它会做以下几步：

1. Register itself in the consumer id registry under its group.  
   在消费者id登记处组下注册它自己
2. Register a watch on changes (new consumers joining or any existing consumers leaving) under the consumer id registry. (Each change triggers rebalancing among all consumers within the group to which the changed consumer belongs.)  
   在消费ID登记处注册一个观察者（新加入的消费者或任何现有的消费者离开）。（每次变化就会触发，改变所属的组内的所有消费者重新平衡。）
3. Register a watch on changes (new brokers joining or any existing brokers leaving) under the broker id registry. (Each change triggers rebalancing among all consumers in all consumer groups.)  
   在broker的id登陆处注册一个观察者（新加入的broker和现有的离开）。（每次改变触发，在所有消费者组的所有消费者之间重新平衡。）
4. If the consumer creates a message stream using a topic filter, it also registers a watch on changes (new topics being added) under the broker topic registry. (Each change will trigger re-evaluation of the available topics to determine which topics are allowed by the topic filter. A new allowed topic will trigger rebalancing among all consumers within the consumer group.)  
   如果消费者创建一个消费流用于topic过滤器，并在broker topic登记处注册一个观察者（监听新添加的topic）。（每次变化将触发可用的topic进行重新评估，以确定哪些topic是过滤器允许的。新的topic将触发消费者组中所有消费者之间进行重新平衡）
5. Force itself to rebalance within in its consumer group.  
   在其消费者组中进行强制重新平衡。

### 消费者再平衡算法（Consumer rebalancing algorithm）

The consumer rebalancing algorithms allows all the consumers in a group to come into consensus on which consumer is consuming which partitions. Consumer rebalancing is triggered on each addition or removal of both broker nodes and other consumers within the same group. For a given topic and a given consumer group, broker partitions are divided evenly among consumers within the group. A partition is always consumed by a single consumer. This design simplifies the implementation. Had we allowed a partition to be concurrently consumed by multiple consumers, there would be contention on the partition and some kind of locking would be required. If there are more consumers than partitions, some consumers won't get any data at all. During rebalancing, we try to assign partitions to consumers in such a way that reduces the number of broker nodes each consumer has to connect to.  
消费者再平衡算法允许组中所有的消费者消费哪一个分区达成共识，同组中的broker和其他的消费者的每一次增加或移除触发消费者再平衡。对于一个给定的topic和给定的消费者组，组内的消费者之间均匀的分配broker分区。如果我们允许一个分区被多个消费者共同消费，这需要锁了，所有我们设计一个分区永远只有一个消费者进行消费。这样设计简化了很多。如果消费者比分区多，那么一些消费者将不会获得任何数据。在再平衡期间，我们试图分配分区给消费者。以这样的方式来减少每个消费者连接到broker的节点数。

Each consumer does the following during rebalancing:  
以下是每个消费者再平衡的过程：

1. For each topic T that Ci subscribes to

2. let PT be all partitions producing topic T

3. let CG be all consumers in the same group as Ci that consume topic T

4. sort PT (so partitions on the same broker are clustered together)

5. sort CG 6. let i be the index position of Ci in CG and let N = size(PT)/size(CG)

7. assign partitions from i\*N to (i+1)\*N - 1 to consumer Ci 8. remove current entries owned by Ci from the partition owner registry

9. add newly assigned partitions to the partition owner registry

(we may need to re-try this until the original partition owner releases its ownership)

When rebalancing is triggered at one consumer, rebalancing should be triggered in other consumers within the same group about the same time.

当一个消费者的再平衡被触发时，在同一时间内，相同的组内的其他消费者也会被触发。

**kafka添加和修改topic**

You have the option of either adding topics manually or having them be created automatically when data is first published to a non-existent topic. If topics are auto-created then you may want to tune the default [topic configurations](http://kafka.apache.org/documentation.html#topic-config) used for auto-created topics.  
如果你第一次发布一个不存在的topic时，它会自动创建。你也可以手动添加topic。

Topics are added and modified using the topic tool:  
topic的添加和修改使用下面的工具。

> bin/kafka-topics.sh --zookeeper zk\_host:port/chroot --create --topic my\_topic\_name

--partitions 20 --replication-factor 3 --config x=y

The replication factor controls how many servers will replicate each message that is written. If you have a replication factor of 3 then up to 2 servers can fail before you will lose access to your data. We recommend you use a replication factor of 2 or 3 so that you can transparently bounce machines without interrupting data consumption.  
副本控制每个消息在服务器中的备份，如果有3个副本，那么最多允许有2个节点宕掉才能不丢数据,集群中推荐设置2或3个副本，才不会中断数据消费。

The partition count controls how many logs the topic will be sharded into. There are several impacts of the partition count. First each partition must fit entirely on a single server. So if you have 20 partitions the full data set (and read and write load) will be handled by no more than 20 servers (no counting replicas). Finally the partition count impacts the maximum parallelism of your consumers. This is discussed in greater detail in the [concepts section](http://kafka.apache.org/documentation.html#intro_consumers).

分区数控制topic将分片成多少log。关于分区数的影响，首先每个分区必须完全安装在独立的服务器上。因此，如果你有20个分区的话(读和写的负载)，那么处理完整的数据集不要超过20个服务器（不计算备份）。最后的分区数影响你的消费者的最大并行。这个在概念章节里进行更详细的讨论。

The configurations added on the command line override the default settings the server has for things like the length of time data should be retained. The complete set of per-topic configurations is documented [here](http://kafka.apache.org/documentation.html#topic-config).

命令行上添加的配置覆盖了服务器的默认设置，服务器有关于时间长度的数据，应该保留。这里记录了每个主题的完整配置。

**kafka修改删除topic**

You can change the configuration or partitioning of a topic using the same topic tool.

你可以使用同样的topic工具更改topic的配置和分区。

To add partitions you can do  
你可以添加分区

> bin/kafka-topics.sh --zookeeper zk\_host:port/chroot --alter --topic my\_topic\_name

--partitions 40

Be aware that one use case for partitions is to semantically partition data, and adding partitions doesn't change the partitioning of existing data so this may disturb consumers if they rely on that partition. That is if data is partitioned byhash(key) % number\_of\_partitionsthen this partitioning will potentially be shuffled by adding partitions but Kafka will not attempt to automatically redistribute data in any way.

要知道，一个用例的分区是语义上的分区数据，添加分区不能改变现有的数据，如果分区被使用中，这就可能会扰乱消费者。也就是说如果数据通过哈希(key)number\_of\_partitions划分，那么该分区将通过添加分区进行洗牌，但kafka不以任何方式自动分配数据。

To add configs:

添加配置：

> bin/kafka-topics.sh --zookeeper zk\_host:port/chroot --alter --topic my\_topic\_name --config x=y

To remove a config:

移除配置：

> bin/kafka-topics.sh --zookeeper zk\_host:port/chroot --alter --topic my\_topic\_name --deleteConfig x

And finally deleting a topic:

最后删除主题：

> bin/kafka-topics.sh --zookeeper zk\_host:port/chroot --delete --topic my\_topic\_name

Topic deletion option is disabled by default. To enable it set the server config

主题删除选项默认是关闭的，设置服务器配置开启它。

delete.topic.enable=true

Kafka does not currently support reducing the number of partitions for a topic or changing the replication factor.

kafka目前不支持减少分区数和改变备份数。

如果想彻底删除数据节点，可以看这篇文章 http://orchome.com/140

**优雅的关闭kafka**

The Kafka cluster will automatically detect any broker shutdown or failure and elect new leaders for the partitions on that machine. This will occur whether a server fails or it is brought down intentionally for maintenance or configuration changes. For the later cases Kafka supports a more graceful mechanism for stoping a server then just killing it. When a server is stopped gracefully it has two optimizations it will take advantage of:

Kafka集群自动检测broker关闭或者失败，并且在该机器上的分区选举新的leaders，当服务器出现故障或故意进行维护或配置更改时，为这种情况，Kafka支持一个更优雅机制关闭服务然后kill它，当服务器正常停止，它有2个最佳：

1. It will sync all its logs to disk to avoid needing to do any log recovery when it restarts (i.e. validating the checksum for all messages in the tail of the log). Log recovery takes time so this speeds up intentional restarts.

它把所有日志同步到磁盘里，当重启时，以避免需要做任何的日志恢复。日志恢复需要时间，所以这样可以加快有意启动。

1. It will migrate any partitions the server is the leader for to other replicas prior to shutting down. This will make the leadership transfer faster and minimize the time each partition is unavailable to a few milliseconds.  
   在关闭之间，它将所有leader分区服务器移动到其他的副本，并且把每个分区不可用的几毫秒的时间降至更低。

Syncing the logs will happen automatically happen whenever the server is stopped other than by a hard kill, but the controlled leadership migration requires using a special setting:

当发生服务器停止不是通过直接kill，就会自动同步日志，但是leader迁移需要使用特殊的设置：

controlled.shutdown.enable=true

Note that controlled shutdown will only succeed if *all* the partitions hosted on the broker have replicas (i.e. the replication factor is greater than 1 *and* at least one of these replicas is alive). This is generally what you want since shutting down the last replica would make that topic partition unavailable.

注意，控制关闭broker和副本上的所有分区才行（即，副本大于1并且这些副本至少一个活着）。这通常因为你关闭最后一个副本将使这个主题分区不可用。