1.入门

1.1 介绍

Kafka 是分布式、分区、可复制的提交日志服务。它采用独特的设计来实现消息服务系统。

主题：维护的一组消息分类；

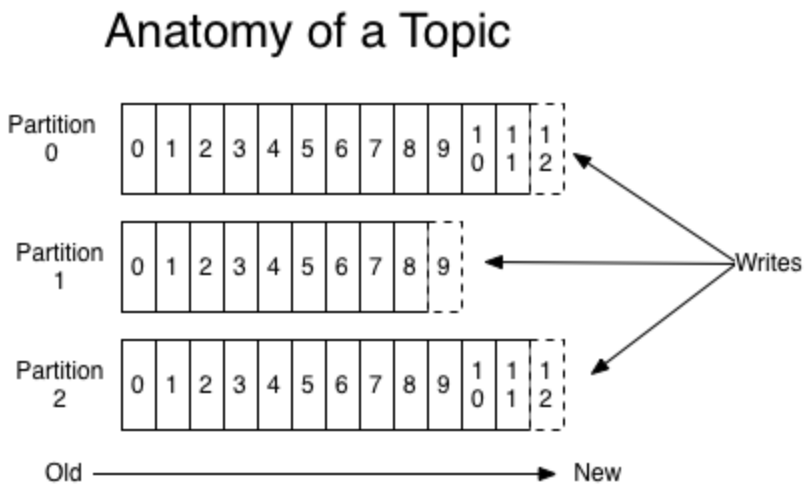
生产者：向 kafka 的主题中发布消息
消费者：订阅并消费主题中的消息
代理：kafka 集群由一个或多个服务端组成。每个服务端称为代理
因此，生产者通过网络发送消息给 kafka 集群，同时，kafka 集群把消息转发给消费者

结构图 : producer -> kafka cluster -> consumer

客户和服务器之间的通信是通过一个简单的、高性能的、语言无关的 TCP 协议。我们不仅提供 java 客户端，同时提供其它多种语言版本的客户端

1.1.1 主题和日志

一个主题就是一个用来发布消息的目录或订阅的名字，对于每个 topic，Kafka 维护一个分区日志，看起来如下
主题结构图



每个分区是一个有序的，可以不断追加消息的消息序列。分区中的每个消息都会分配一个在分区内是唯一的序列号，这个序列号叫做偏移量(offset)。

kafka 集群保留所有已经发布的消息（无论这些消息是否已经被消费）一段时间。我们可以配置这个时间来决定集群维护消息的时间长短。比如：如果消息被设置保存两天，那么两天内，消息都是可以被消费的。但是两天后为了节省磁盘空间就会删除消息。

kafka 性能对数据大小不敏感，因此保留大量数据毫无压力。

事实上，每个消费者仅仅需要保存的元数据是消费者在日志中的消费位置（偏移量），这个偏移量是由消费者控制：通常，消费者读取消息后会线性递增偏移量，但是，消费者可以按任意顺序消费消息。比如：消费者可以重置偏移量到老版本。

以上特性的组合使得 consumer 的代价很小。**consumer** 数量可以增加或减少而对整个集群影响很小。

日志分区的目的：第一：允许日志规模超出一台服务器的文件大小限制。每个单独的分区都必须受限于主机的文件限制，但一个主题可有多个分区，因此可以处理无限数量的数据。

1.1.2 分布式

日志的所有有分区被分发到集群中的服务器上，每个服务器处理全部分区中的部分分区数据和请求。为了容错，每个分区都被复制到一定数量（可配置）的不同服务器上。

每个分区（有多个副本）都有一台服务器作为**"leader"**，大于等于 **0** 台服务器做为**"followers"**。"leader"服务器处理分区的所有读写操作。**"followers"**服务器对当前分区做为旁观者，什么都不做。当"leader"服务器不可用时，那么"followers"中的一台将自动成为**"leader"**。每台服务器都即做为一些分区的"leader"，又做为其它分区的"followers"。

1.1.3 生产者

生产者向所选的主题发布数据。生产者负责选择哪些消息应该分配到主题内的哪个分区。这种选择分区方式，可以使用简单的循环方式负载均衡； 也可以通过一些语义分区函数实现（如：基于消息的 key 的 **hash** 等）

1.1.4 消费者

传统的消息处理有两种模型：队列和发布订阅。队列模式，消费者池中的消费者可以从一台服务器读数据，并且每个消息只被其中一个消费者消费。发布订阅模式，消息通过广播方式发送给所有消费者。kafka 提供了一个单一的抽象概念，可以满足这两种（队列、发布订阅）模式－－消费者组。

消费者通过分组名标识自己，每条消息被发布到主题，并只会分发给消费者组中的 唯一个 消费者实例（即只被组中的一个消费者消费）。这些消费者即可以是同一台服务器上不同的进程，也可以是位于不同服务器上进程。

如果所有的消费者实例属于同一分组（相同的分组名），那么这就是传统的队列模式（相同 topic，只有一个消费者能抢到消息）。

如果所有的消费者实例不属于同一分组，那么这就是发布订阅模式（每个消费者都能收到消息）

通常，主题有少量称为逻辑订阅者的分组。为了可扩展性和容错每个分组是由许多消费者实例组成。

kafka 也比传统消息系统拥有更强的顺序性。

传统队列维护消息顺序性。如果多个消费者从队列中消费消息，那么服务器以存储的顺序分发消息。虽然消息从服务器出队列是按顺序的，但是被分发给消费者时，是通过异步的方式，因此消息到达不同消费者时可能是乱序的。这意味者并发消费时，消费是乱序的。消息系统为了做到这点，会采用只有一个消费者消费的理念，但这也意味是无法并行操作。

kafka 这点做的更好，通过称为分区（主题内）的并行概念，kafka 即可以提供顺序又可以负载均衡。这是通过给主题内的相同分组下的消费者提供多个分区的架构，来实现每个分区只能被一个消费者消费。通过这种方式，可以确保同一分区只有一个消费者，因此一个分区消费消息是顺序的； 同时，由于有多个分区，因此可以负载均衡。注意：一个分组内，消费者数量不能多于分区数量。此处的：不能多于，应该不绝对。即：一个应用集群（有消费者）可

能远远多于分区数量，只能说超出的消费者永远都无分区消费，但并不影响其它消费者正常使用（我瞎猜的，需要确认-todo)

kafka 仅仅支持分区内的消息顺序消费，并不支持全局（同一主题的不同分区之间）的消息顺序。这处方式下，通过按消息 key 隔离数据的方式足够满足大部分应用； 但是，如果你需要一个全局顺序消费消息，你可以通过一个主题只有一个分区的方法实现，但是这也意味着一个分组只有一消费者；

保证（此处翻译偏差较大，待重新翻译）

同一个生产者发送的不同消息在分区中的存储是顺序的。
同一个消费者看的消息顺序与这些消息的存储顺序是一致的。
对一个拥有Ｎ个复制因子（Ｎ个分区），最多允许 N-1 台 server 故障还能保证消息不丢失。（是最多，并不代表一定。因为每个分区可以位于不同机器，也可以同时位于一台机器）

1.2 使用案例

1.2.1 消息处理

kafka 是一个很好的传统消息代理替代产品。消息代理有几种原因：解耦生产者与消息处理、缓存消息等。与大多数消息系统相比，kafka 有更好的吞吐量，内置分区，复制和容错性，这使它成为大规模消息处理应用很好的解决方案。

1.2.2 网站活动跟踪

kafka 的原始用例（为此而生）是能重建一套可以实时发布，实时订阅消息，用于处理用户活动轨迹跟踪的管道。也就是说网站的活动（页面浏览、搜索、用户其它行为）可以按活动类型分别发布到各自的主题；这些订阅可以被用于后续各种用途：包括实时处理、实时监控、加载到 **hadoop**、离线数据仓库。

因为每个用户浏览页面都会产生活动消息，因此，活动跟踪数据量非常大。

1.2.3 度量

kafka 经常被用于处理监控数据。这涉及到从分布式应用收集统计数据，并且做为后续分析的一个统一的数据源。（即分布式统计数据查询入口或代理）

1.2.4 日志收集

很多人把 kafka 做为日志收集解决方案。日志收集是从服务器上采集日志文件并把它们放入一个集中位置（如：文件服务器或 hdfs）统一处理。kafka 抽象了文件细节，并给出一个日志或事件消息流。这允许更低的延时处理，更容易支持多数据源以及分布式消息处理。与 Scribe 和 Flume 相比，kafka 提供同样的良好性能，并提供更好的可用性（因为多个副本），和更低的延时。

1.2.5 流处理

很多 kafka 用户，通过把数据处理分成多个步骤，每个步骤处理数据的不同功能并放入此步骤的 topic 中，并通过 kafka topics 串联起所有步骤，形成一个数据处理通道。

如：一个处理新闻的流程：首先通过 RSS 收集新闻，并发布到"articles"主题中；第二步，从"articles"主题中取新闻并清洗重复内容，然后发布一个新的主题中；最后，从上步的主题中取数据并推荐给用户。
这样的处理管道是基于单个主题的实时数据流程图。从 0.10.0.0 版本开始，一个轻量但强大的，被称为 kafka stream 的功能用于处理这样的数据。除了 **Kafka stream** 还有另外相似的开源工具：Apache Storm / Apache Samza。

Event Sourcing（事件溯源）
提交日志
kafka 可以做为分布式系统的外部提交日志服务器。可以帮助分布式节点存储数据失败时，做为重新同步机制，在节点与操作之间复制日志，以恢复数据。

这种情况，kafka 与 Apache BookKeeper 非常相似。

1.3 快速开始

本教程，假设你没有任何 kafka 知识。并且没有现成的 kafka 和 zookeeper 数据。

步骤 1: 下载代码 http://kafka.apache.org/downloads.html

下载 0.10.0.0 版本代码，并且解压

```
tar -xzf kafka_2.11-0.10.0.0.tgz
cd kafka_2.11-0.10.0.0
```

步骤 2: 启动服务

kafka 依赖 zookeeper，因此首先要启动 zookeeper；如果没有安装独立的 zookeeper，可以使用 kafka 内嵌的 zookepper。虽然这种方式快速但不是很好。

启动 zookeeper
```
> bin/zookeeper-server-start.sh config/zookeeper.properties
```

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

步骤 3: 创建一个主题

手动创建一个名为"test"的主题

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

查看刚创建的主题
```
> bin/kafka-topics.sh --list --zookeeper localhost:2181
```

可以通过配置“自动创建主题”，这样如果没有提前创建主题，那么在发布消息时，如果此消息对应的主题不存在，会自动创建。

步骤 4: 发送消息
通过命令行客户端，可以通过文件或标准输入（命令行）向 kafka 集群发送消息。默认每行都是一条消息。

启动生产者（启动成功进入命令行阻塞状态，可以输入数据，回车发送）
./kafka-console-producer.sh --broker-list localhost:9092 --topic test

步骤 5: 启动消费者
启动消费者命令行(启动后命令行处于阻塞状态，生产者发布的消息会在此显示)
./kafka-console-consumer.sh --zookeeper 127.0.0.1:2181 --from-beginning --topic test

步骤 6: 设置服务器集群

单个服务器挺没劲的，现在扩展到 3 台服务器（伪 3 台，在同一台机器上模拟）。
现在配置服务器，让三台服务彼此联系

```
 > cp config/server.properties config/server-1.properties
 > cp config/server.properties config/server-2.properties
```

修改上面配置文件，两个都修改

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 是唯一的，为的是让服务器知道自己是谁，以及自己的配置是哪个。在不同机器上，会配置 zookeeper 的所有机器。通过这个 id,可以知道自己的配置。
配置了日志和端口，由于是在同一台机器模拟，所以日志路径和端口不能重了。

启动新增的两台 kafka（zookeeper 已经启动，同一台只需要启动一个即可）

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

创建一个新的的主题:

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

现在我们已经创建一个集群，但是我们怎么知道每个 broker 都做了什么？执行如下命令：″describe toics”：

```
> 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
```

第一行是所有分区的简介，其它行是每个具体分区的说明。因为我们只有一个分区，因此总共只有两行。

″leader”: 是负责处理一个指定分区的所有读写操作。每个 leader 都是随机选择的。
“replicas”：副本是一个用来复制分区日志的节点列表。无论这些分区对应的服务器是 leader 还是处于激活状态，都会被复制。
“isr”
注意：在我们的例子中，节点 1 是我们主题中唯一分区的 leader；

我们可以运行相同的命令，来查看我们最早创建的主题″test”的信息：

```
> 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
```

遗憾的是：这个主题没有副本；

让我们向新主题发布一些消息：

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

现在让我们消费这些消息：

```
> bin/kafka-console-consumer.sh --zookeeper localhost:2181 --from-beginning --topic
my-replicated-topic
...
my test message 1
my test message 2
```

现在让我们来测试容错。broker 1 正在充当 leader，我们先干掉它：

```
  > ps | grep server-1.properties
  7564 ttys002    0:15.91
/System/Library/Frameworks/JavaVM.framework/Versions/1.8/Home/bin/java...
  > kill -9 7564
```

领袖已经由奴隶节点中的一个充当。节点 1 已经不再是 xxxx

```
 > 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
```

现在，即使原先那个写操作的 leader 已经仙逝，但消息仍然能被正常消费。

```
> bin/kafka-console-consumer.sh --zookeeper localhost:2181 --from-beginning --topic
my-replicated-topic
...
my test message 1
my test message 2
```

步骤 7: 使用 kafka connect 导入/导出数据

通过命令行读写数据是一个方便简单的开始。但是你可能希望通过其它数据源或者从 kafka 导出数据到其它系统。对大多数系统，你不需要写定制化的代码，只需要使用 kafka connect 就可以导入或导出数据。kafka connect 是一个可以运行多个 **connectors** 的扩展工具集。这些 connectors 实现了与外部系统交互的逻辑。在这个示例中，我们将体验怎么运行 kafka connect ,并通过简单的 connectors 从文件把数据导入到主题，并从主题中把数据导出到文件。
首先，我们创建一些种子数据，如下：

```
echo -e “foo\nbar” > test.txt
```

接着，我们以独立模式启动两个 connectors。所谓独立模式，即这两个 connectors 运行在一个单一的、本地的、专有的进程中。我们提供三个配置文件做为后续命令的参数。第一个是用于普通的连接进程，包括 broker，以及数据序列化格式。剩于的两个配置文件，每个都指定了一个需要创建的 connector。这些文件包含一个唯一的 connector 名字，

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

kafka 使用这些简单的配置文件启动，并创建两个 connectors：第一个 connector 是一个从文件中按行读取数据，并把数据发布到主题的数据源 connecotr。第二个是一个从主题消费消息，并把消息存到文件里的接收 Connector。在启动过程中，你会看到许多日志信息，包括 connetors 实例化的一些指标信息。一旦 kafka connect 进程启动，数据源 connector 就会开始从 text.txt 文件中读取数据。并且把数据发到 connect-test 主题。然后接收者 connector 开始从 connect-test 主题中读取数据，并且把数据写到 test.sink.txt 文件中。
我们可以检查 test.sink.txt 文件，来验证数据是否已经被传递到事个管道中。

```
cat test.sink.txt
foo
bar
```

注意：数据正在被存储到 connect-test 主题中，因此我们可以运行一个命令行消费者来查看主题中的数据。

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”}
...

我们可以向源文件中添加数据来检查数据在管道中的流动:

echo “Another line” >> test.txt
“`
你是不是惊奇的发现，”Another line”这条数据已经保存到 test.sink.txt 文件中了。我的天哪，这太神奇了。

步骤 8: 使用 kafka Streams 处理数据
kafka stream 是一个处理实时的流式操作和数据分析的客户端库。 下面的快速入门示例将演示如何使用这个库去运行一个流式应用代码。下面是 WordCountDemo 示例的关键代码(使用的是 java8 lamda 表达式)。

KTable wordCounts = textLines
// 以空格为分隔符切分文本
.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（单词计数）算法，用于计算一个单词在文本中出现的次数。但是跟其它你以前见过的 WordCount 算法有些许不同。因为它是被设计用来处理无限的流式数据。它是一个跟踪和更新单词计数的动态算法。由于必须假定输入数据是无限的，并且无法确定什么时候处理完“所有”数据，因此它会定期输入出当前状态和处理结果。

现在，我们准备向 kafka 主题中输入一些数据，用于 kafka Streams 应用程序后续数据。

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

接着，我们把输入的数据通过命令行生产者发送到```streams-file-input```主题中（在实践中，流式数据会源源不断的发布到 kafka 中）。
```
  > bin/kafka-topics.sh --create \
```
```

```
        --zookeeper localhost:2181 \
        --replication-factor 1 \
        --partitions 1 \
        --topic streams-file-input
 > cat file-input.txt | bin/kafka-console-producer.sh --broker-list localhost:9092 --topic
streams-file-input

```

现在，我们可以启动 WordCount 示例应用来处理输入的数据。

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

```
因为结果被写回到另一个主题，因此，除了日志条件，没有任何内容输出到“标准输出”。不像典型的流处理程序，示例代码运行几秒后会自动停止。

我们现在可以通过读取输出主题的消息，来检查 WordCount 示例输出的数据：
```
> bin/kafka-console-consumer.sh --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

```
程序运行后，会在命令行输出以下数据：
```
  all     1
  streams 1
  lead    1
  to      1
  kafka   1
  hello   1
  kafka   2
  streams 2
  join    1
  kafka   3
  summit  1
```

第一列是消息的 key，第二列是消息的值。两列都使用了 java.lang.String 格式。注意，那个“输出”是一个持续的更新流。其中每条记录（原输出中的每一行）是一个单词的更新统计。对具有相同 key 的多条记录，以后的每条统计记录都是前一次的更新。

现在，你可以向 streams-file-input 主题中写入更多的数据，你会发现，添加的数据会追加到“streams-wordcount-output”主题，并且体现到单词计数记录中（上述操作，可以通过命令行生产者和消费者来观察）

1.4 生态

除了上面描述的工具外，还有很多其它工具，如： stream processing system，Hadoop 集成，monitoring，开发工具等。具体请看生态页;

2. API

apache kafka 包含一个新的 java 客户端（位于 org.apache.kafka.clients 包中）。新的 java 客户端是为了替换旧的 Scala 客户端。但是一段时间内两个客户端会共存。同时这些客户端可以以单独的 jar 存在，而旧的 Scala 客户端仍然打包在 server 中。

2.1 生产者 api

我们鼓励开发者使用新的 java 版客户端生产者 api。这个客户端是经过生产环境测试，并且包含比 Scala 客户端更快更全面的功能。你可以通过 maven 配置来加载客户端。

```
 <dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-clients</artifactId>
  <version>0.10.0.0</version>
 </dependency>
```

这里的 api 文档中包含了一些示例。http://kafka.apache.org/0100/javadoc/index.html?org/apache/kafka/clients/producer/KafkaProducer.html

2.2 消费者 api

在 0.90 版本，我们已经提供了新的 java 版本消息都客户端。目的是替换旧版基于 zookeeper 的上层（xxx???)consumer，以及低层的 consumer apis。当前版本是测试版。为了让用户平滑升级，我们仍然维护 0.8.0 版本的客户端。下面我们即讲解 0.8.0 版，也讲解最新版。
https://cwiki.apache.org/confluence/display/KAFKA/Consumer+Group+Example

```
public class ConsumerGroupExample {
    private final ConsumerConnector consumer;
    private final String topic;
    private  ExecutorService executor;

    public ConsumerGroupExample(String a_zookeeper, String
```

```
a_groupId, String a_topic) {
        consumer =
kafka.consumer.Consumer.createJavaConsumerConnector(
                createConsumerConfig(a_zookeeper, a_groupId));
        this.topic = a_topic;
    }

   private static ConsumerConfig createConsumerConfig(String
   a_zookeeper, String a_groupId) {
           Properties props = new Properties();
           props.put("zookeeper.connect", a_zookeeper);
           props.put("group.id", a_groupId);
           props.put("zookeeper.session.timeout.ms", "400");
           props.put("zookeeper.sync.time.ms", "200");
           props.put("auto.commit.interval.ms", "1000");
           return new ConsumerConfig(props);
    }


    public void run() {
        ConsumerIterator<byte[], byte[]> it =
m_stream.iterator();
        while (it.hasNext())
            System.out.println("Thread " + m_threadNumber
+ ": " + new String(it.next().message()));
        System.out.println("Shutting down Thread: " +
m_threadNumber);
    }
}
```

对大多数应用而言，上层 consumer api 就可以满足。但是有些应用可能需要一些没有暴露到上层 consumer 中的特性（如：当重启 consumer 时重置偏移量）。他们可以使用低层的 SimpleConsumer api。这个逻辑有点复杂，你可以参考这里的示例;

2.2.3 新版消费者 Api

新版 consumer api 删除了 0.8 版中，高层 api 与底层 api 的区别，直接使用统一的 api。你直接使用下面的 maven 配置加载新版客户端。

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>0.10.0.0</version>
</dependency>
```

用法示例

2.3 streams api

在 0.10.0 版本中我们新增了一个称为"Kafka Stream"的新客户端库。它主要用于处理流式操

作。特别注意,这个还是内测试版。有许多 api 在未来很可能有大的变动，因此使用需谨慎。你可以通过以下 maven 配置加载它。

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-streams</artifactId>
    <version>0.10.0.0</version>
</dependency>
```

这里的[javadocs]有一些示例（切记：标注了@InterfaceStability.Unstable 注解的 class，在未来很可能有大的变动，而且不会向后兼容。一句话：谁用谁填坑）。

3. 配置

kafka 使用基于 property 文件格式的键值对配置程序。这些键值对即可以来自 property 文件，也可以来自编程方式。

3.1 代理(broker)（服务器）配置

最核心最必须最基础的配置如下
broker.id
log.dirs
zookeeper.connect
主题级别的配置，以及具体的默认配置在下面有详情讨论：
名称 说明 数据类型 默认值 有效值 重要程度
zookeeper.connect zookeeper 集群地址(多个用英文逗号分隔主机 ip 及端口配置)
advertised.host.name DEPRECATED: only used when `advertised.listeners` or `listeners` are not set. Use
auto.create.topics.enable 发布消息，如果主题不存在，则自动创建主题。否则必须先创建主题。线上一般为 false
PLAINTEXT://myhost:9092,TRACE://:9091 PLAINTEXT://0.0.0.0:9092, TRACE://localhost:9093 string null high
log.dir 日志（即数据)目录，其实是 kafka 的持久化文件目录。
log.dirs 日志（即数据)目录，其实是 kafka 的持久化文件目录(可以配置多个用逗号隔开). 如果未设置，将使用 log.dir 配置的目录
log.flush.interval.messages The number of messages accumulated on a log partition before messages are flushed to disk long 9223372036854775807 [1,…]
offsets.commit.timeout.ms Offset commit will be delayed until all replicas for the offsets topic receive the commit or this timeout is reached. This is similar to the producer request timeout. int 5000 [1,…] high
offsets.load.buffer.size Batch size for reading from the offsets segments when loading offsets into the cache.int 5242880 [1,…] high
offsets.topic.num.partitions The number of partitions for the offset commit topic (should not change after deployment)int 50 [1,…] high
queued.max.requests The number of queued requests allowed before blocking the network threads int 500 [1,…] high

controller.socket.timeout.ms The socket timeout for controller-to-broker channels int 30000 medium
default.replication.factor 对自动创建的主题而言：默认副本数。默认值是 1,即不复制。建议 int 1 medium
更多关于 broker 的配置细节，可以到类 kafka.server.KafkaConfig 中查看。

有关主题的配置，即可以使用全局默认值，也可以每个主题单独配置。如果给定的主题没有配置，那么将使用全局默认配置。创建主题时，可以通过给定配置参数，覆盖全局默认配置。下面的示例就是创建一个命名为“my-topic”的主题，并且设置消息最大大小和刷新频率。

> 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

除了创建主题时设置配置，通过–alter 命令也可以随时修改主题配置。如下，设置主题 my-topic 的最大消息大小：

> bin/kafka-topics.sh --zookeeper localhost:2181 --**alter** --topic my-topic --config max.message.bytes=128000

也可以通过--delete-config 来删除主题定制的配置，恢复到全局默认配置。如下：

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

以下是主题级别的配置。服务器级的默认配置是全局配置。可以设置这个值用于配置没有自定义的主题。具体在“服务器配置属性列”
属性：主题属性名；
默认值：由服务器默认值指定。
服务器默认值：用于配置全局默认值。可修改

3.2 生产者(producer)配置

以下为生产者配置：
名称 说明 数据类型 默认值 Valid Values 重要程度
bootstrap.servers 一个主机/端口对的集合列表。目的是为了初始与 kafka 集群建立连接。注意：客户端不需要把的集群中所有的主机都配置在这里。此处的目的仅仅时初始的时候建立与集群的连接，一旦建立，客户端可以使用使用集群中的任何一台机器，即此处没有配置。配置的格式如下：host1:port1,host2:port2…。另外，此配置仅仅是为了与集群第一次建立连接，不需要全配置（会动态增加或删除机器，也没办法全配置），但至少配置两台，防止一台崩溃，就再也连不上集群了。 切记：是 kafka 集群主机，不是 zookeeper 主机。java 客户端不直接与 zookeeper 交互） l

key.serializer key:实现了 Serializer 接口的序列化类。用于告诉 kafka 如何序列化 key.
value.serializer value:实现了 Serializer 接口的序列化类。用于告诉 kafka 如何序列化 value.
acks leader 收到的来自所有 follower 复制成功的确认数量(比如：有 10 个

follower，只要 2 个确认成功就认为整个发送成功[还有一个是 leader 自己]，那么 acks=3)。只有达到这个数量，生产者才认为消息成功发送。
acks=0　如果值为 0，生产者不会等待服务器的任何确认。只要消息被立即发送到 socket buffer 中，就认为发送成功。 这种情况下，无法保证服务器已经收到了消息。同时 retries（失败重试） 配置也会失效（因为客户端收不到任何错误）。 返回的 offset（偏移量）也不是正确的值，永远都是－1. 注意：此时，通过 Feture.get()会是怎样的结果呢?
acks=1 这意味着，leader 把日志写入到本地日志（acks=1,代表它自己完成就可以），不等待任何其它 follower 的确认，就认为消息成功发送。 这种情况，如果在 follower 成功复制副本前，且 leader 已经告诉客户端已经成功后，leader 崩溃了，那么这条消息就会丢失。
acks=all 这表示 leader 会等待所有"in-sync"台 follower 的复制确认。只要一台"in-sync" follower 处于激活状态，数据就保证不会丢失。 这是最强的可用性保证。注：″in-sync"只是的有 follower 中的一部分，并不是所有 follower. 同时：acks 只能是：0,1,all，还是 0,1,2…99999…,all

buffer.memory生产者可以用来缓冲消息记录的总内存大小（单位：字节）。如果消息产生的速度超过发送的速度，那么生产会被阻塞″max.block.ms"时间，如果超过此时间会抛出异常。 这个设置大体上与生产使用的全部内存相当。但是这不是一个硬性标准，因为生产者用的所有内存不是都用来缓冲。例如：一些附加的内存用来压缩（如：压缩被启用）或维护请求。

compression.type　生产者发送的数据压缩格式。默认不压缩。合法的值有： none, gzip, snappy, 或 lz4。 压缩是针对数据的所有批处理。因此批处理会影响压缩比（越大的批处理，压缩比越高）。
retries　待：不知道此值是一个类似 boolean 弄的数据，还是重试次数。如果是 boolean 型，那么只重试一次? 文档并没说清。待查看源码 如果此值大于 0,那么客户端在接收到事务 error 时，会重新发送。注意：这种重试与异常重发没有区另（？？？） 允许重试可能会改变记录的次序。因为如果两条消息发送到同一分区，那么第一条失败了并重试，但是第二条成功了，那第二条会出现在第一的位置。
ssl.key.password　The password of the private key in the key store file. This is optional for client.
ssl.keystore.location　The location of the key store file. This is optional for client and can be used for two-way authentication for client.
ssl.keystore.password The store password for the key store file.This is optional for client and only needed if ssl.keystore.location is configured.

batch.size　无论何时消息记录发送到同一个分区时，生产者会试图批量把记录放在一起发送到服务器，以此来减少请求次数。这样会即帮助客户端也帮助服务器端的性能提升。此配置就是用来控制默认配置的（单位：字节）。 批处理的消息记录不会超过此配置。 小的批处理量会降低吞吐量（如果批处理量为 0,那么将完全禁止批处理）。大的批处理量又会非常浪费内存。

client.id　是一个字符串类型的值，生产者向服务器发送消息时创建。目的是：能在服务器端的日志中保存一个应用的逻辑名称，比通过 ip 和端口更方便的方式来追踪请求源。

connections.max.idle.ms　空闲链接存活时间。超过此配置的时间（毫秒），则关闭此空闲链接
linger.ms　生产者会把 linger.ms 指定的时间间隔内的多次请求打包成一个批处理请求。通常这只用于消息产生太快，以至于超过了向服务器传输速度的情况下。不过，在某些压力不是很大的负载下，仅仅为了降低请求次数也可以使用此配置。 通过此配置，消息不会立即发

送，而是延迟指定的时间后再这送。这样在延迟的时间间隔内，还允许其它消息与当前消息做为一次批处理。 这跟 TCP 中的 Nagle 算法类似。这个配置给定了一次批处理的范围：一旦消息达到了 **batch.size** 指定的字节数，会立即发送，此时会忽略当前配置。如果没有达到 batch.size 指定的值，那么消息会延迟 linger.ms 指定的时间后再发送。 说白了：batch.size 是通过消息大小来触发批处理。linger.ms 是通过时间来触发批处理。一方面可以防止消息延迟太久，另一方面可以防止消息积压太多。

max.block.ms 此配置是用来控制 KafkaProducer.send() 的时间，以及 KafkaProducer.partitionsFor() 的阻塞时间.缓冲慢了或者无数据不可用都会导致当前方法被阻塞。

max.request.size 一次请求的数据大小的上限（单位：字节）。这也影响最大的一条记录的大小上限。注意，服务器也有此配置，而且配置的值与当前值可能不相同。 此配置可以限止一次请求的批处理的消息条数，以防止发送过大的请求。

receive.buffer.bytes The size of the TCP receive buffer (SO_RCVBUF) to use when reading data.
request.timeout.ms 配置超时时间。如果在此配置的时间内还没有收到服务器的返回值，那么可能会失败重发或者如果重发次数过多时就直接失败。
sasl.kerberos.service.name The Kerberos principal name that Kafka runs as. This can be defined either in Kafka’s JAAS config or in Kafka’s config. string null mediu
sasl.mechanism SASL mechanism used for client connections. This may be any mechanism for which a security provider is available. GSSAPI is the default mechanism. string
security.protocol Protocol used to communicate with brokers. Valid values are: PLAINTEXT, SSL, SASL_PLAINTEXT, SASL_SSL. string PLAINTEXT medium
send.buffer.bytes ＴＣＰ缓冲大小 (SO_SNDBUF)。与批处理缓冲不同。此处仅指 TCP 使用的缓冲
ssl.enabled.protocols The list of protocols enabled for SSL connections. list [TLSv1.2, TLSv1.1, TLSv1]
ssl.keystore.type The file format of the key store file. This is optional for client. string
ssl.protocol The SSL protocol used to generate the SSLContext. Default setting is TLS, which is fine for most cases. Allowed values in recent JVMs are TLS, TLSv1.1 and TLSv1.2. SSL, SSLv2 and SSLv3 may be supported in older JVMs, but their usage is discouraged due to known security vulnerabilities. string TLS medium
ssl.provider The name of the security provider used for SSL connections. Default value is the default security provider of the JVM. string null medium
ssl.truststore.type The file format of the trust store file. string JKS medium
timeout.ms The configuration controls the maximum amount of time the server will wait for acknowledgments from followers to meet the acknowledgment requirements the producer has specified with the acks configuration. If the requested number of acknowledgments are not met when the timeout elapses an error will be returned. This timeout is measured on the server side and does not include the network latency of the

request. int 30000 [0,…] medium
block.on.buffer.full When our memory buffer is exhausted we must either stop accepting new records (block) or throw errors. By default this setting is false and the producer will no longer throw a BufferExhaustException but instead will use the max.block.ms value to block, after which it will throw a TimeoutException. Setting this property to true will set the max.block.ms to Long.MAX_VALUE. Also if this property is set to true, parameter metadata.fetch.timeout.ms is not longer honored.
This parameter is deprecated and will be removed in a future release. Parameter max.block.ms should be used instead.

3.3 消费者(consumer)配置

3.3.2 新版消费者配置

新版还是 beta 版

Name Description Type Default Valid Values Importance
bootstrap.servers 用于客户端向服务器建立初始连接的 kafka broker ip 及端口集。与 producer 中配置相同。 list high
key.deserializer 实现了接口 Deserializer 的类，用于反序列化 key.与 producer 中的 key.serialzer 对应。 class high
value.deserializer 实现了接口 Deserializer 的类，用于反序列化 value.与 producer 中的 value.serialzer 对应。 class high
fetch.min.bytes 批量读，数据量最小限止。 一次拉取请求返回给客户端的最小数据量。如果数据小于此值，服务器不会立即返回请求，而是等待更多数据，至到超过此值。 默认值为 1 字节，也就是说每次请求都会立即返回。此配置可以提高吞吐量，不过会增加延迟。 int 1 [0,…] high
group.id 一个标识多个 consumer 为一组，且为字符串类型的唯一值。对于集群环境此值非常有用，只有集群中一台机器可以消费，防止重复消费。 主题决定了哪些消费者可以消费。group.id 决定了同一个主题的多个消费者同一消费只能被一个消费（互斥）。 A unique string that identifies the consumer group this consumer belongs to. This property is required if the consumer uses either the group management functionality by using subscribe(topic) or the Kafka-based offset management strategy. string "" high
heartbeat.interval.ms The expected time between heartbeats to the consumer coordinator when using Kafka’s group management facilities. Heartbeats are used to ensure that the consumer’s session stays active and to facilitate rebalancing when new consumers join or leave the group. The value must be set lower than session.timeout.ms, but typically should be set no higher than 1/3 of that value. It can be adjusted even lower to control the expected time for normal rebalances. int 3000 high
max.partition.fetch.bytes The maximum amount of data per-partition the server will return. The maximum total memory used for a request will be #partitions * max.partition.fetch.bytes.This size must be at least as large as the maximum message size the server allows or else it is possible for the producer to send messages larger than the consumer can fetch. If that happens, the consumer can get stuck trying to fetch a large

message on a certain partition. int 1048576 [0,…] high
session.timeout.ms The timeout used to detect failures when using Kafka’s group management facilities. When a consumer’s heartbeat is not received within the session timeout, the broker will mark the consumer as failed and rebalance the group. Since heartbeats are sent only when poll() is invoked, a higher session timeout allows more time for message processing in the consumer’s poll loop at the cost of a longer time to detect hard failures.See also max.poll.records for another option to control the processing time in the poll loop. Note that the value must be in the allowable range as configured in the broker configuration by group.min.session.timeout.ms and group.max.session.timeout.ms. int 30000 high
ssl.key.password The password of the private key in the key store file. This is optional for client. password null high
ssl.keystore.location The location of the key store file. This is optional for client and can be used for two-way authentication for client. string null high
ssl.keystore.password The store password for the key store file.This is optional for client and only needed if ssl.keystore.location is configured. password null high
ssl.truststore.location The location of the trust store file. string null high
ssl.truststore.password The password for the trust store file. password null high
auto.offset.reset What to do when there is no initial offset in Kafka or if the current offset does not exist any more on the server (e.g. because that data has been deleted):
earliest: automatically reset the offset to the earliest offset
latest: automatically reset the offset to the latest offset
none: throw exception to the consumer if no previous offset is found for the consumer’s group
anything else: throw exception to the consumer.
string latest [latest, earliest, none] medium
connections.max.idle.ms Close idle connections after the number of milliseconds specified by this config. long 540000 medium
enable.auto.commit 偏移量是由 consumer 维护，并不代表是存储在本客户端（客户端无处可存，且对于多个 group 消费者无法实现互斥消费），实际上是存储在 zookeeper。 因此 offset 可以由服务器代为维护。 boolean true medium
exclude.internal.topics Whether records from internal topics (such as offsets) should be exposed to the consumer. If set to true the only way to receive records from an internal topic is subscribing to it. boolean true medium
max.poll.records 一次请求返回批量数据的上限。一次拉取，最多返回的消息条数。 此数据默认值过大，生产环境最好设置一个更小的值（我自己瞎猜的）。 int 2147483647 [1,…] medium
partition.assignment.strategy The class name of the partition assignment strategy that the client will use to distribute partition ownership amongst consumer instances when group management is used list [org.apache.kafka.clients.consumer.RangeAssignor] medium
receive.buffer.bytes 用于接收消费的 TCP 缓冲大小. int 65536 [0,…] medium
request.timeout.ms 请求超时时间。超过时间，客户端会失败重发（客户端重发在哪里配

置？），或者重发过多时失败。 int 40000 [0,…] medium
sasl.kerberos.service.name The Kerberos principal name that Kafka runs as. This can be defined either in Kafka’s JAAS config or in Kafka’s config. string null medium
sasl.mechanism SASL mechanism used for client connections. This may be any mechanism for which a security provider is available. GSSAPI is the default mechanism. string GSSAPI medium
security.protocol Protocol used to communicate with brokers. Valid values are: PLAINTEXT, SSL, SASL_PLAINTEXT, SASL_SSL. string PLAINTEXT medium
send.buffer.bytes 发送数据的 TCP 缓冲大小。 int 131072 [0,…] medium
ssl.enabled.protocols The list of protocols enabled for SSL connections. list [TLSv1.2, TLSv1.1, TLSv1] medium
ssl.keystore.type The file format of the key store file. This is optional for client. string JKS medium
ssl.protocol The SSL protocol used to generate the SSLContext. Default setting is TLS, which is fine for most cases. Allowed values in recent JVMs are TLS, TLSv1.1 and TLSv1.2. SSL, SSLv2 and SSLv3 may be supported in older JVMs, but their usage is discouraged due to known security vulnerabilities. string TLS medium
ssl.provider The name of the security provider used for SSL connections. Default value is the default security provider of the JVM. string null medium
ssl.truststore.type The file format of the trust store file. string JKS medium
auto.commit.interval.ms 如果 enable.auto.commit 设置为 true，那么此值为自动提交的频率（单位：毫秒）。即：每 auto.commit.interval.ms 自动提交一次 offsets。 long 5000 [0,…] low
check.crcs Automatically check the CRC32 of the records consumed. This ensures no on-the-wire or on-disk corruption to the messages occurred. This check adds some overhead, so it may be disabled in cases seeking extreme performance. boolean true low
client.id 一个发送到服务器的 id。目的是通过一个有意义的值用于跟踪请求源，而不仅仅通过 ip/port 来跟踪。如：使用应用名。 string "" low
fetch.max.wait.ms The maximum amount of time the server will block before answering the fetch request if there isn’t sufficient data to immediately satisfy the requirement given by fetch.min.bytes. int 500 [0,…] low
interceptor.classes 消费者拦截器列表。实现了 ConsumerInterceptor 接口的类允许拦截消费者收到的所有消息（可能对消息做统一修改）。默认情况下，没有配置任何拦截器。 list null low
metadata.max.age.ms The period of time in milliseconds after which we force a refresh of metadata even if we haven’t seen any partition leadership changes to proactively discover any new brokers or partitions. long 300000 [0,…] low
metric.reporters A list of classes to use as metrics reporters. Implementing the 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 The number of samples maintained to compute metrics. int

2 [1,…] low
metrics.sample.window.ms The window of time a metrics sample is computed over. long 30000 [0,…] low
reconnect.backoff.ms 链接失败重试，间隔时间。当链接失败，间隔一定时间后再重试，防止短时间内，死循环式快速重试。 此配置对从当前 consumer 发出的所有请示有效。 long 50 [0,…] low
retry.backoff.ms 失败重试，间隔时间。当请求失败时，间隔一定时间后再失败重试。防止短时间内，由于一些错误原因，导致死循环式快速失败重发。 long 100 [0,…] low
sasl.kerberos.kinit.cmd Kerberos kinit command path. string /usr/bin/kinit low
sasl.kerberos.min.time.before.relogin Login thread sleep time between refresh attempts. long 60000 low
sasl.kerberos.ticket.renew.jitter Percentage of random jitter added to the renewal time. double 0.05 low
sasl.kerberos.ticket.renew.window.factor Login thread will sleep until the specified window factor of time from last refresh to ticket’s expiry has been reached, at which time it will try to renew the ticket. double 0.8 low
ssl.cipher.suites A list of cipher suites. This is a named combination of authentication, encryption, MAC and key exchange algorithm used to negotiate the security settings for a network connection using TLS or SSL network protocol.By default all the available cipher suites are supported. list null low
ssl.endpoint.identification.algorithm The endpoint identification algorithm to validate server hostname using server certificate. string null low
ssl.keymanager.algorithm The algorithm used by key manager factory for SSL connections. Default value is the key manager factory algorithm configured for the Java Virtual Machine. string SunX509 low
ssl.trustmanager.algorithm The algorithm used by trust manager factory for SSL connections. Default value is the trust manager factory algorithm configured for the Java Virtual Machine. string PKIX low

3.4 kafka Connect 配置

Name Description Type Default Valid Values Importance
config.storage.topic kafka topic to store configs string high
group.id A unique string that identifies the Connect cluster group this worker belongs to. string high
internal.key.converter Converter class for internal key Connect data that implements the Converter interface. Used for converting data like offsets and configs. class high
internal.value.converter Converter class for offset value Connect data that implements the Converter interface. Used for converting data like offsets and configs. class high
key.converter Converter class for key Connect data that implements the Converter interface. class high
offset.storage.topic kafka topic to store connector offsets in string high

status.storage.topic kafka topic to track connector and task status string high
value.converter Converter class for value Connect data that implements the Converter interface. class high
bootstrap.servers A list of host/port pairs to use for establishing the initial connection to the Kafka cluster. The client will make use of all servers irrespective of which servers are specified here for bootstrapping—this list only impacts the initial hosts used to discover the full set of servers. This list should be in the form host1:port1,host2:port2,.... Since these servers are just used for the initial connection to discover the full cluster membership (which may change dynamically), this list need not contain the full set of servers (you may want more than one, though, in case a server is down). list [localhost:9092] high
cluster ID for this cluster, which is used to provide a namespace so multiple Kafka Connect clusters or instances may co-exist while sharing a single Kafka cluster. string connect high
heartbeat.interval.ms The expected time between heartbeats to the group coordinator when using Kafka's group management facilities. Heartbeats are used to ensure that the worker's session stays active and to facilitate rebalancing when new members join or leave the group. The value must be set lower than session.timeout.ms, but typically should be set no higher than 1/3 of that value. It can be adjusted even lower to control the expected time for normal rebalances. int 3000 high
session.timeout.ms The timeout used to detect failures when using Kafka's group management facilities. int 30000 high
ssl.key.password The password of the private key in the key store file. This is optional for client. password null high
ssl.keystore.location The location of the key store file. This is optional for client and can be used for two-way authentication for client. string null high
ssl.keystore.password The store password for the key store file.This is optional for client and only needed if ssl.keystore.location is configured. password null high
ssl.truststore.location The location of the trust store file. string null high
ssl.truststore.password The password for the trust store file. password null high
connections.max.idle.ms Close idle connections after the number of milliseconds specified by this config. long 540000 medium
receive.buffer.bytes The size of the TCP receive buffer (SO_RCVBUF) to use when reading data. int 32768 [0,…] medium
request.timeout.ms The configuration controls the maximum amount of time the client will wait for the response of a request. If the response is not received before the timeout elapses the client will resend the request if necessary or fail the request if retries are exhausted. int 40000 [0,…] medium
sasl.kerberos.service.name The Kerberos principal name that Kafka runs as. This can be defined either in Kafka's JAAS config or in Kafka's config. string null medium
sasl.mechanism SASL mechanism used for client connections. This may be any mechanism for which a security provider is available. GSSAPI is the default mechanism.

string GSSAPI medium
security.protocol Protocol used to communicate with brokers. Valid values are: PLAINTEXT, SSL, SASL_PLAINTEXT, SASL_SSL. string PLAINTEXT medium
send.buffer.bytes The size of the TCP send buffer (SO_SNDBUF) to use when sending data. int 131072 [0,…] medium
ssl.enabled.protocols The list of protocols enabled for SSL connections. list [TLSv1.2, TLSv1.1, TLSv1] medium
ssl.keystore.type The file format of the key store file. This is optional for client. string JKS medium
ssl.protocol The SSL protocol used to generate the SSLContext. Default setting is TLS, which is fine for most cases. Allowed values in recent JVMs are TLS, TLSv1.1 and TLSv1.2. SSL, SSLv2 and SSLv3 may be supported in older JVMs, but their usage is discouraged due to known security vulnerabilities. string TLS medium
ssl.provider The name of the security provider used for SSL connections. Default value is the default security provider of the JVM. string null medium
ssl.truststore.type The file format of the trust store file. string JKS medium
worker.sync.timeout.ms When the worker is out of sync with other workers and needs to resynchronize configurations, wait up to this amount of time before giving up, leaving the group, and waiting a backoff period before rejoining. int 3000 medium
worker.unsync.backoff.ms When the worker is out of sync with other workers and fails to catch up within worker.sync.timeout.ms, leave the Connect cluster for this long before rejoining. int 300000 medium
access.control.allow.methods Sets the methods supported for cross origin requests by setting the Access-Control-Allow-Methods header. The default value of the Access-Control-Allow-Methods header allows cross origin requests for GET, POST and HEAD. string "" low
access.control.allow.origin Value to set the Access-Control-Allow-Origin header to for REST API requests.To enable cross origin access, set this to the domain of the application that should be permitted to access the API, or '*' to allow access from any domain. The default value only allows access from the domain of the REST API. string "" low
client.id An id string to pass to the server when making requests. The purpose of this is to be able to track the source of requests beyond just ip/port by allowing a logical application name to be included in server-side request logging. string "" low
metadata.max.age.ms The period of time in milliseconds after which we force a refresh of metadata even if we haven't seen any partition leadership changes to proactively discover any new brokers or partitions. long 300000 [0,…] low
metric.reporters A list of classes to use as metrics reporters. Implementing the 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 The number of samples maintained to compute metrics. 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 Interval at which to try committing offsets for tasks. long 60000 low
offset.flush.timeout.ms Maximum number of milliseconds to wait for records to flush and partition offset data to be committed to offset storage before cancelling the process and restoring the offset data to be committed in a future attempt. long 5000 low
reconnect.backoff.ms The amount of time to wait before attempting to reconnect to a given host. This avoids repeatedly connecting to a host in a tight loop. This backoff applies to all requests sent by the consumer to the broker. long 50 [0,…] low
rest.advertised.host.name If this is set, this is the hostname that will be given out to other workers to connect to. string null low
rest.advertised.port If this is set, this is the port that will be given out to other workers to connect to. int null low
rest.host.name Hostname for the REST API. If this is set, it will only bind to this interface. string null low
rest.port Port for the REST API to listen on. int 8083 low
retry.backoff.ms The amount of time to wait before attempting to retry a failed request to a given topic partition. This avoids repeatedly sending requests in a tight loop under some failure scenarios. long 100 [0,…] low
sasl.kerberos.kinit.cmd Kerberos kinit command path. string /usr/bin/kinit low
sasl.kerberos.min.time.before.relogin Login thread sleep time between refresh attempts. long 60000 low
sasl.kerberos.ticket.renew.jitter Percentage of random jitter added to the renewal time. double 0.05 low
sasl.kerberos.ticket.renew.window.factor Login thread will sleep until the specified window factor of time from last refresh to ticket’s expiry has been reached, at which time it will try to renew the ticket. double 0.8 low
ssl.cipher.suites A list of cipher suites. This is a named combination of authentication, encryption, MAC and key exchange algorithm used to negotiate the security settings for a network connection using TLS or SSL network protocol.By default all the available cipher suites are supported. list null low
ssl.endpoint.identification.algorithm The endpoint identification algorithm to validate server hostname using server certificate. string null low
ssl.keymanager.algorithm The algorithm used by key manager factory for SSL connections. Default value is the key manager factory algorithm configured for the Java Virtual Machine. string SunX509 low
ssl.trustmanager.algorithm The algorithm used by trust manager factory for SSL connections. Default value is the trust manager factory algorithm configured for the Java Virtual Machine. string PKIX low
task.shutdown.graceful.timeout.ms Amount of time to wait for tasks to shutdown gracefully. This is the total amount of time, not per task. All task have shutdown triggered, then they are waited on sequentially. long 5000 low

4. 设计

4.1 动机

我们设计 kafka 的目的是为了提供了一个处理所有实时数据汇总的统一平台。为了实现这个目标，我们不得不考虑各种各样的用例。

首先，它应该是一个可以以高吞吐量的方式来处理像实时日志收集一样的高容量事件流。

其次，能优雅处理离线系统周期性加载数据而导致的大批量数据积压。

另外，它能够处理低延迟的处理传统的点对点消息传递。

我们希望能支持分区，分布式，实时处理汇总的数据，传递。这促成了我们的分区和消费者模型。

在流信息被传递给其它数据系统时，我们知道系统必须能保证在机器出错故障时具有容错能力。

为了支持上面这些想法，我们设计了一些相对于传统消息系统更像数据库日志的独特组件。在下面的章节中，我们会概述其中一部分设计。

4.2 持久化

不要害怕文件系统
kafka 强依赖文件系统来存储和缓存消息。一个普遍的看法是"磁盘很慢"，这也使人们怀疑一个提供持久化功能的系统在性能上有没有竞争力。事实上，磁盘说慢非常慢，说快也很快。只要正确使用，正确设计磁盘结构，会让磁盘跟网络一样快。

磁盘性能的关键事实是：在过去十年中，磁盘的吞吐量已经受到磁盘寻址的严重影响（翻译不到位，可以看原文）。因此，一个配置了 6 块 six 7200 转每分钟的 SATA RAID-5 磁盘阵列，线性写可以达到 600MB/秒。但是相同配置随机写只有 100k/秒。整整相差了 6000 多倍。线性读写是所有使用方式中最容易想到的。并且也是被操作系统做了很大优化的方式。一个现代操作系统，会提供预读和延迟写技术来处理数据。预取即一次性取很多数据块。延迟写即把一组小的逻辑写整合成一次大的物理写。这也说明，顺序访问磁盘会比随机访问要快很多。

为了弥补性能损失，现代操作系统更激进的使用主存来充当磁盘内存。一个现代 os 会很乐意使用所有的空闲内存来充当磁盘缓存(pageCache)。只有在缓存回收（其它进程申请缓存，且缓存不够时，回收 pageCache）时损失一点性能(要写回磁盘)。磁盘的所有读写都将通过这个统一的缓存间接读取。如果不使用直接磁盘 io，这个特性很难关闭。因此，即使一个进程维护了一个进程内的数据缓存，那么数据很可能被复制到 os pageCache 中，所有数据都会被有效存储两次。

此外，我们是建立在 jvm 之上。研究过 java 内存的都了解以下两点：

(1) 对象的内存开销非常高，通常是数据存储的倍数，或者更糟。
(2) 随着堆内存数据增加，java 垃圾收集变的越来越困难。

因为这些特性，使用文件系统以及依赖 pageCache 比直接使用内存(虽然 pageCache 也在内存中，但不像 jvn 直接使用内存，而是通过 pageCache 间接使用：不知道理解是否正确)或其它结构。我们通过自动连接到空闲内存，至少让可用内存翻倍（有空闲内存就使用，其它程序申请就释放，不固定占有－－相同 jvm head）。 通过存储压缩过字节数据比存储对象，我们可能还能让可用内存翻倍。这就使在一个 32G 内存的机器上缓存 28-30G 也不会触发 GC。另外，即使服务重启了，缓存仍然存在。相反，进程内的缓存会全部丢失（重新加载需要花费很长时间）－－(注意：是服务重启，并不是机器重器。pageCache 是在内存中，由 os 管理，与进程无直接关系，因此不会丢失)。
这也简化了维护缓存与文件系统之间一致性的编程逻辑。这比使用一次性的进程内缓存更有效也更正确。

这意味着这是一个非常简单的设计：不用担心因为直接使用内存（进程内维护），刷新内存到文件系统时出现内存溢出导致数据丢失。所有数据立即写持久化日志到文件系统（os 层面，而磁盘，此处批的是 pageCache),而不是直接刷新到磁盘。这也意味着把数据转移到内核的 pageCache。

通过 pageCache 和顺序读写来提高性能
pageCache 是非进程内，所以不受大小限制，不会被 jvm GC。而且可以随时使用空闲内存，以及自由释放（别人不用我用，别人用我立马还）
而且，进程中内存也同时会有 pageCache，所以是重复。

顺序读写：是随机读写性能的 **6000** 倍。

恒定的时间复杂度

大部分持久化数据结构都会选择 BTree，这能提供丰富的存取，它的时间复杂度为 o(logN)。但是 kafka 使用更简单的方式，向文件中追加数据，或从文件中顺序读数据。缺点就是不提供丰富的读写方式（如：按某属性检索指定数据）。

4.3 效率

我们在效率上投入了大量努力。我们最主要的一个用例就是处理网站活动数据。网站活动数据容量非常大：每个页面浏览都可能产生几个写操作。另外，每条消息都至少有一个（通常很多）消费者消息。因此，我们努力让消息消费的代价变的尽可能小。

上一节我们讨论了磁盘效率，影响磁签署效率主要有两个原因：过多的小型 i/o 操作，和大量的字节复制。

小型 i/o 即发生在客户端与服务器之间，又发生在服务器自己的持久化操作中。

为了消除上述情况，我们创建了一个“消息集”的抽象概念，即消息按自然分组组合在一起。这种机制允许网络请示可以一次把多个消息分组打包在一起发送，这样可以分摊一次请求只发一条消息而带来的网络开销。服务器可以一次性把打包在一起的消息追加到日志中。消费者也可以整包消费消息。

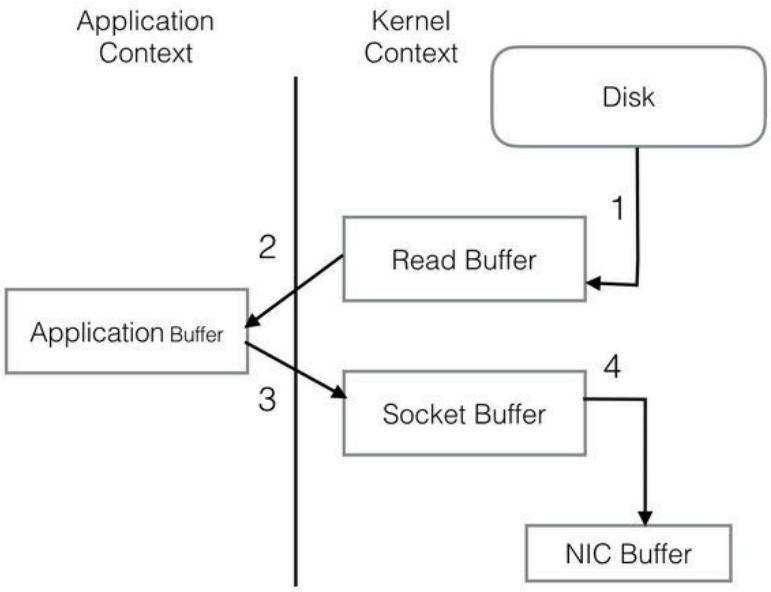
这个简单的优化让速度提升了一个数量级。批量操作导致大的网络数据包，大的磁盘线性操作，以及连序的内存块等等。所有这些，都让 kafka 能把随机的消息流以线性的方式写入，并发送给消费者。

另外一影响性能的是字节复制。在低消息率（消息量小）的情况下，这不是问题。但是在大量负载的情况下，这影响是显著的。为了消除这个，我们采用了一个标准的二进制消息格式。生产者、消费者、代理都遵守这个标准（因此数据包在传递过程中不需要修改）。

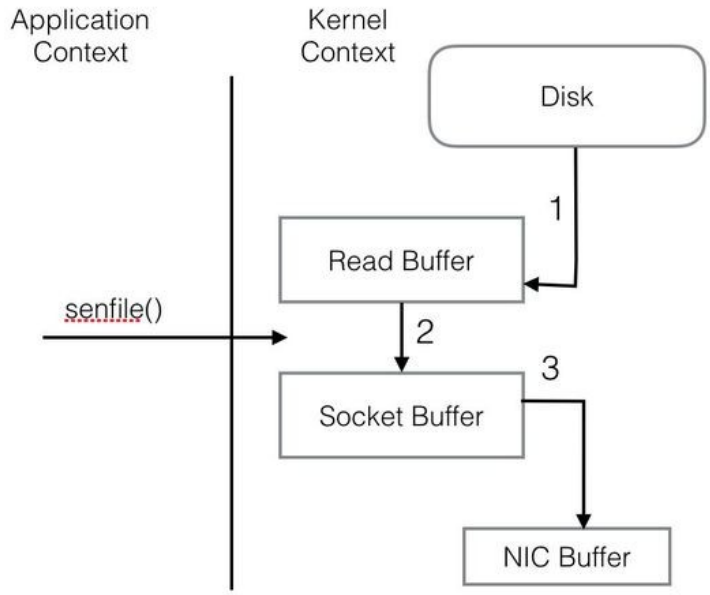
消息日志本身就是一个由 broker 维护的文件集目录。这些文件顺序存储着消息集。这些消息存储格式与生产者和消费者使用了相同的格式（传统的 mq 可能使用不的格式：如生产者使用字符串，代理使用二进制，消费者又使用字符串）。保持通用的格式可以优化一个最重要的操作：持久化日志块的网络传输。现代 unix 操作系统，为从 pagecache 向 socker 传输数据提供了一个高度优化的编程方式。
在 linux 下，这个是通过“ sendfile system call.”实现的

为了理解 sendFile 的影响，需要了解数据从文件到 socket 的传输路径。

(1). 操作系统把数据从磁盘上读到内核空间中的 pagecache 中。
(2). 应用程序把数据从内核空间读到用户空间的缓冲中。
(3). 应用程序把数据写回到内核空间的 socket 缓冲中。
(4). 操作系统把数据从 socket 缓冲中复制到 NIC 缓冲中。然后发送到网络上。
消息传输过程 1



四个副本，两次系统调用，这显然是低效。通过使用 sendfile 可以消除重复复制。它是允许 os 直接把数据从 pagecache 复制到 network。



消息传输过程 1

我们希望针对一个主题多个消费者的情况能有一个通用的用例。使用上面零复制优化方案，数据从磁盘到 pagecache，只复制一次。多次消费共用同一个 pagecache 的数据。而不像那种把数据放到内存的做法，每次读都会把数据从内核中复制出去。这让消息的消费速率基本接近网络传输的最高限制。

把 pagecache 和 sendfile 结合起使用，让 kafka 集群，你会发现基本没有读取磁盘的操作，因此数据完全从缓存中读取（基本没有读操作，并不代表没有写操作。此处有一个前提：生产者和消费者同时在线，生产者生产的数据放入 **pagecache** 中，还没有被回收时，就被消费者消费，因此基本不需要再从磁盘读到 pagecache）。

点到点的批量压缩

在一些情况下，cpu 和磁盘并不是瓶颈，网络带宽才瓶颈。对于霜从数据中心向广域网发消息的数据管道，确实是这样的。当然，用户可以一次压缩一条消息（完全不需要 kafka 的支持即可）。但是这可能导致很低的压缩率。因为一条消息的冗余数据并不多。如果想提高压缩率，应该多条相同类型的消息一起批量压缩（相同类型的消息它们结构相同，那么属性名就高度冗余）。

kafka 支持循环递归消息集（具体参考：MessageSet）。一批消息可以一起压缩并以压缩的格式一起发送到服务器。这批消息会以压缩格式存储在日志中。只有在被消息时被消费者解压。

kafka 支持 GZIP， Snappy，LZ4 压缩协议。更多细节看这里

4.4 生成者

负载均衡

生产者直接发送数据到代理。代理是没有任何中间路由层的。为了帮助代理实现路由，每个 kafka 节点能提供在作何时间获取服务器元数据的请求。这些元数据包括哪些服务器牌激活状态，同一个主题分区的领导者位置信息（如：ip/端品）。

客端控制消息发布到哪个分区。这种控制算法可是随机的（随机负载均衡），或者通过一些语议分区涵数。我们公开了语议分区接口，允许用户提供一个消息键，通过这个键 hash 到不同的分区（当然如果需要，可以重写分区函数）。例如：如果消息键是一个用户 id,那么同一个用户的所有消息都会被发送到相同的分区上，反过来消费者消息时就可以（？？？）。这种设计方式，是为了对位置敏感的消费者（？？：估计类似于分布式 session）

异步发送

批量发送是效率的主要因素。为了批量操作，kafka 生产者通过在内存中缓存数据，并一次请求发送较大的批次。批量操作可以通过配置一个消息数量上限和一个延迟时间（64k 或 10ms）来控制批量操作。这样就允许一次请求可以发送更消息，让服务器使用更少量但是更大的 i/o 操作。缓存是可配置的，为牺牲少量延迟但是提供更高吞吐量的追求提供了一个可选机制。

4.5 消费者

kafka 消费者通过拉的方式从代理消费消息。消费者在每次请求中指定它在消息日志中的偏移量（从哪个位置开始消费），并从该位置接收一个日志块。因此，消费者可以控制偏移量位置，并在需要时重新消费已消费过的数据。

推送 vs 拉取

一个我们考虑的问题是，消费者是应该从代理拉取数据，还是代理把数据推给消费者。 在这方面，kafka 选择了一个大多数消息系统者会使用，但更传统的方式：生产者主动把数据推送到代理，消费者主动从代理拉取数据。一些以日志为中心的系统，例如： Scribe 和 Apache Flume，是基于推送的方式把数据推到下流系统。这两种方式各有利弊，但是基于推的系统，因为代理控制了数据传输速率，所以很难处理各种不同类型的消费者。一般来说，我们的目标是为了让消费者尽可能以最大的效率消息消息。但是，基于推的系统，如果消费端消费速率跟不上生产速率，就会导致过载。基于拉的系统有更好的特性，消费者可以落后生产者，只是在消费者有能力的情况下追赶生产者即可。这种方式可以通过某种补偿协议实现。消费者可以表示自己过载，但是仍然可以充分利用传输速率。因此，我们选择使用拉模型。

基于拉的系统还有另外一个优点：消费者可以主动批量操作数据。一个基于推的系统，在不知道消费者是否会立即处理的情况下，必须选择是立即推送还是缓存更多数据后批量推送。如果需要低延迟，就必须一次一条消息的方式发送，这太浪费了。基于拉的设计修复了这个缺陷，因为消费者可以从日志的当前位置拉取后面的所有消息（可以配置最大拉取数量）。因此在不会引起不必要的延迟的情况下，这是一个最佳的批处理方式。

一个简单的基于拉的系统有一个不足之处就是：如果代理没有数据了，消费者可能仍然循环拉取数据。为了消除这个，我们在请求的提供了参数，让以通过长连接的方式来阴塞一个请求，至到代理有数据时再返回。

消费者位置

让人惊讶的是，跟踪哪些消息被消费了，是影响一个消息系统性能的一个关键点。大多消息系

统都会在代理上保存哪些消息被消费的元数据。也就是说，消息一旦分发给消费者，代理就要立即在本地记录下，或者等消费者确认。这是一个直观的选择。事实上，对一个单一人服务器，这种状态除此之外还能存到哪儿。由于许多消息系统存储的消息规模并不大，因此这也是一个务实的选择。代理知道哪些消息已经消费了，并可以立即删除，因此可以保持一个很小量的数据存储。

让代理和消费者对消息是否已经消费达成协议到底是不是一个微不足道的问题也许很不明显。如果消息每次通过网络分发出去，代理就必须马上记录下来，如果消费者处理消息失败了（如：超时了，或崩溃了），那么消息可能就丢失了。为了解决这个问题，许多消息系统增加了一个回执确认的特性。也就是说，消息一是发出，消息会被标记为"发送"状态而不是"已消费"状态。代理等待消费者回执确认后，再把消息标记为已消息。但是，这又带来了新的问题，首先，如果消费者处理完消息，在发送回执的时候失败了，那么消息可能会被消费两次。现在为了解决这个问题，代理必须对每条消息保存多个状态（一个是加锁，以至不会发送第二次。第二可删除状态），这个会影响性能。

kafka 使用不同的处理方式。主题被切分成小的顺序的分区集合。每一个分区同一时间只能被一个消费者消费。这就意味着一个消费者的位置在每个分区上只用一个 integer 字段就可标记下一条消息的偏移量。这使得消息被消费的状态维护非常小。状态要以定期状态，这使消息确认代价非常小。

kafka 采用的这种方式有个副作用。一个消费者可以故意倒回去消息已经被消费的数据。这违反了队列的通用协议。但是这原来就应该是大多数消费者的重要特性。例如，如果消费者代码有一个 bug，在消息被消费以后才发与，那么在 bug 修复以后可以重新消费消息。
业务处理失败的话，如果没有提交 offset，会发起重新分配，被分配到的消费者会重新消费此条消息。

离线数据加载

可扩展的持久化特性使这些消费者成为可能。像：定期批量加载数据到离线系统的消费者（hadooop 或关系型数据仓库）
在 hadoop 并行加载数据的情况下我们通过拆分 map-task，每个节点/主题/分区都允许并行加载。hadoop 提供 task 管理，任务失败时，可以重新从原来的位置加载（不用担心失败，因为有副本）。

4.6 消息传递语义

待...

4.7 复制

kafka 复制每个主题的分区日志，如果配置了多台服务。这允许集群中一台服务失效后，通过副本仍然能保持消息的可用性。

别的消息系统也提供了相关的复制特性（kafka 是完全基于副本），但是它们感觉就是一个附加的，没有完全使用的，有很大缺点：奴隶服务器是非激活状态，吞叶量严重影响，它需要手动配置等等。kafka 默认就是基于副本，事实上，我们(???)

副本的最小单位主题分区。在没有失败的情况下，每个 kafka 分区有一个 leader 和 0 个或更多 followers。副本总数（包括 leader）组成了副本因子。所有的读写都由 leader 的分区上处

理。通常，有比 leader 多很多的分区。leader 均匀的分布在所有 broker 上。followers 上的副本日志与 leader 保持完全一样：都有相同的偏移量，消息拥有相同的顺序（当然，在任何时间，leader 日志未尾有几条还没有复制的消息－因为延迟）。

followers 从 leader 消费消息就像一个普通的消费者，把消息追加他们日志（不是复制文件的方式创建副本，而是假装成消费者从 leader 消费）。**followers** 从 **leader** 批量处理日志。

像大多数分布式系统自动处理故障一样，需要对节点的"激活"状态有一个明确的定义。对 kafka 节点而言，需要满足两个我们的：
节点必须能与 zookeeper 保持会话（心跳机制）。如果节点是一个奴隶节点，它必须能复制 leader 上的所有写操作，并且不能落后太多。我们指的节点满足这两个条件叫做"同步"。为了避免"激活"和""失败"这两个概念的含糊不清。leader 跟踪"同步"的所有节点集合。如果 followers 挂了，或者卡住了，或者落后太多（复制的太慢），那么 leader 就从它的同步副本中删除。卡住或滞后的副本控制是由 replica.lag.time.max.ms 配置。

一条消息只有当所有同步副本把消息都追加到它们的分区日志中才称为"已提交"。只有"已提交"的消息才会发送给消费者。这意味消费者不需要担心它们看到消息会因为 leader 挂掉而消失。另一方面，生产者可以选择是等待消息处于"已提交"状态还是不需要等待。这取决于为了低延迟还是持久化之间的权衡。这是通过生产者的"**acks**"配置来控制。

做为 kafka 心脏的分区，kafka 分区是一自制的日志。副本是分布式系统最基础的原语之一。有许多实现方法。副本可以被一些使用，但是由其它分布式系统以状态模式的方式实现。

一个日志复制模型，就是处理一系统值时顺序保持一致。有很多方法可以实现，但是有一个最简单也是最快的方式：选一个 leader，由这个 leader 来选择提供给它的值的顺序。只要 leader 是激活的，所有 followers 只要复制 leader 所选的顺序就可以保证所有副本消息顺序的一致性。

当然，如果 leader 没有失效，我们是不是需要 followers 的。但是，当 leader 挂了时，我们需要从 followers 中选一个新的 leader。不过，followers 本身也可能落后太多，或者崩溃。因此我们必须保证我们选择一个最新的 follower。一个日志副本的基本保证的算法是：如果我们告诉客户端消息已经提交了，此时 leader 失效了，那么选出的新 leader 必须包含那条消息。这就需要一个折衷，如果一条消息在被标明"已提交"之前等待更多的 follower 确认，那么它在失效后，会有更多的 follower 可以做为新的 leader，这会造成吞吐量降。

一个常用的的算法是：majority vote（"少数服从多数"），但 Kafka 并未采用这种方式。这种模式下，如果我们有 2f+1 个 Replica（包含 Leader 和 Follower），那在 commit 之前必须保证有 f+1 个 Replica 复制完消息，为了保证正确选出新的 Leader，fail 的 Replica 不能超过 f 个。因为在剩下的任意 f+1 个 Replica 里，至少有一个 Replica 包含有最新的所有消息。这种方式有个很大的优势，系统的 latency 只取决于最快的几个 Broker，而非最慢那个。Majority Vote 也有一些劣势，为了保证 Leader Election 的正常进行，它所能容忍的 fail 的 follower 个数比较少。如果要容忍 1 个 follower 挂掉，必须要有 3 个以上的 Replica，如果要容忍 2 个 Follower 挂掉，必须要有 5 个以上的 Replica。也就是说，在生产环境下为了保证较高的容错程度，必须要有大量的 Replica，而大量的 Replica 又会在大数据量下导致性能的急剧下降。这就是这种算法更多用在 ZooKeeper 这种共享集群配置的系统中而很少在需要存储大量数据的系统中使用的原因。例如 HDFS 的 HA Feature 是基于 majority-vote-based journal，但是

它的数据存储并没有使用这种方式。

实际上，Leader Election 算法非常多，比如 ZooKeeper 的 Zab, Raft 和 Viewstamped Replication。而 Kafka 所使用的 Leader Election 算法更像微软的 PacificA 算法。

Kafka 在 ZooKeeper 中动态维护了一个 **ISR**（in-sync replicas），这个 ISR 里的所有 Replica 都跟上了 leader，只有 ISR 里的成员才有被选为 Leader 的可能。在这种模式下，对于 **f+1** 个 Replica，一个 Partition 能在保证不丢失已经 commit 的消息的前提下容忍 **f** 个 Replica 的失败。在大多数使用场景中，这种模式是非常有利的。事实上，为了容忍 f 个 Replica 的失败，Majority Vote 和 ISR 在 commit 前需要等待的 Replica 数量是一样的，但是 ISR 需要的总的 Replica 的个数几乎是 Majority Vote 的一半。

虽然 Majority Vote 与 ISR 相比有不需等待最慢的 Broker 这一优势，但是 Kafka 作者认为 Kafka 可以通过 Producer 选择是否被 commit 阻塞来改善这一问题，并且节省下来的 Replica 和磁盘使得 ISR 模式仍然值得。

不纯洁的 leader 选举：如果所有副本都失效了怎么办?

kafka 保证数据不丢失的前提是：至少有一个副本是同步的。如果所有节点的分区副本都失效了，就无法保证数据不丢失。

如果真发生了这种事情，那么有两个方案：
等待 ISR 中的任一个 Replica“活”过来，并且选它作为 Leader
选择第一个“活”过来的 Replica（不一定是 ISR 中的）作为 Leader
这就需要在可用性和一致性当中作出一个简单的折衷。如果一定要等待 ISR 中的 Replica“活”过来，那不可用的时间就可能会相对较长。而且如果 ISR 中的所有 Replica 都无法“活”过来了，或者数据都丢失了，这个 Partition 将永远不可用。选择第一个“活”过来的 Replica 作为 Leader，而这个 Replica 不是 ISR 中的 Replica，那即使它并不保证已经包含了所有已 commit 的消息，它也会成为 Leader 而作为 consumer 的数据源（前文有说明，所有读写都由 Leader 完成）。Kafka0.8.*使用了第二种方式。根据 Kafka 的文档，在以后的版本中，Kafka 支持用户通过配置选择这两种方式中的一种，从而根据不同的使用场景选择高可用性还是强一致性。

可用性和持久性

当向 kafka 写入消息时，生产者可以选择是否等待消息被 commited 的确认（0,1,all(-1)）。注意，all(-1)，不保证所有的副本都收到消息了。默认情况下，acks=all，只保证所有“同步”（为了性能，一般只选指定数量的副本做为同步副本，只要这些副本确认了就可以保证消息不丢失）的副本已经确认。例如：如果一个主题配置了两个副本，一个失效（那么就只有一个同步副本），当 acks=all 成功时，然后，写入的数据仍然可能会丢失。如果余下的副本也失败了？？？。虽然这是为了保证最大的可用性，但有时这种对有些更喜欢持久性的用户是不可用取的方案。因此我们提供了两个主题级别的配置来满足持久性：

禁用 unclean 的领导人选举-如果所有的副本变得不可用，那么分区将保持不可用，直到最近的领导者再次可用。这有效地提高数据丢失的风险。
指定一个最小 ISR 数量，只有当 ISR 数量超过最小 ISR 数量，分区才接收写入。

副本管理

我们上面的讨论都是围绕单一的日志，即一个主题分区。然后 kafka 集群中有成百上千个分区。我们试图用一个循环的试负载分区。避免对高容量的主题所有分区分区都集中在少数的几台节点上。同样，我们也让每个节点都成为某个分区的 leader(防止某个节点是很多主题的 leader，而其它节点仅仅是 follower)

优化领导人选举进程是非常重要的，因为它是不可用性的关键窗口。一个最简单的选举实现是：当一个节点失效时，将一个一个终止这个节点上每个分区的选举。相反，我们选举出所有节点上的一个做为"controller"，这个"controller"检测 borker 级别的失败，并且当一个 broker 失败时，负责更改所有影响的分区的 leader。
这使得我们可以批量发起变更通知，这让选举程序面对大量分区时，变得代价更小也更快。如果 controller 失效了，余下的 broker 中的一个将成为新的 controller.

4.8 日志压缩

即：日志在指定时间内或大小内可以保存一段时间，一但超出，将被删除。但日志压缩可以按消息 key 合并压缩长时间保留（如：相同 key－如：userId，可能有多条数据，如：如每次修改就是一条消息，那么会压缩成一条数据，而且是最后一条数据）
为了更好的实现负载均衡和消息的顺序性，kafka 的 producer 在分发消息时可以通过分发策略发送给指定的 partition。实现分发的程序是需要制定消息的 key 值，而 kafka 通过 key 进行策略分发。

4.9 配额（限额）

从 0.9 版本，kafka 集群提供了强制生产者和消费者的配额。限额是对每个 client_id 的，基于字节的截阀值。一个 clientId 逻辑上限定一个应用。因为一个 clientId 可跨越多个生产者和消费者。所以配额可以做为一个整体限制一个 clientId 对应的所有生产者和消费者。

为什么需要配额?
对于处理非常大容量数据的生产者或消费者，很可能会独占 broker 资源，并导致网络饱和以及对别的客户端和 borkers 产生 DOS 攻击。有了配额，就可解决这个问题。在大的多租户集群中，一系列小的恶意操作都可能降低用户的体验。

强制
默认情况，每个唯一的 clientId 会收来自集群配置的配额（bytes/sec，通过 quota.producer.default, quota.consumer.default 配置）。配置是基于 broker，每个 client 只能发布或拉取以最大速率范围内。我们决定基于 broker 定义配额，要比配置到 client 上要好。因为那样实现起来太难了。

当检测到一个超额行为，broker 会怎么做?
在我们的解决方案中，broker 不会向客端报错，而是直接降低客端速度。它计算违规的 client 需要延迟的时间，并延迟 response.这种方法对客户端是透明的。

覆盖默认配额
有时可能需要一个更高的配额，clientId 的配额是在 zookeeper 下的/config/clients 中配置。这个配置会影响所有 brokers，并立即生效。这让我们不无可厚非重启整个集群。

5. 实现

5.1 api 设计

生产者 APIs

生产者 ＡＰＩ包含两个低层 producers： kafka.producer.SyncProducer 和 kafka.producer.async.AsyncProducer.

```
class Producer {

  /* 发送数据，通过 key 分区。即可以使用同步也可以使用异常 producer 发送 */
  public void send(kafka.javaapi.producer.ProducerData<K,V> producerData);

  /* 发送数据集，通过 key 分区。即可以使用同步也可以使用异常 producer 发送 */
  public void send(java.util.List<kafka.javaapi.producer.ProducerData<K,V>>
producerData);

  /* 关闭 producer，并清理*/
  public void close();

}
```

目标是通过暴露一个包含所有功能的 api 接口给客户端。
新的 producer：

要以处理多生产者请求的队列/缓冲 和 批量数据异步调度。
kafka.producer.Producer 提供批量处理多个生产者请示的能力(**producer.type=async**).批量处理大小可以通过参数配置。数据被先缓冲在队列中，当达到 queue.time 或 batch.size 时，一个后台线程（kafka.producer.async.ProducerSendThread）从队列中取出批量数据，然后让 kafka.producer.EventHandler 序列化，并发送到 broker 分区。
一个自定义 event Handler 可以通过 event.handler 配置参数进行配置。对于各种各样的生产者队列管线，注册回调方法非常有用。即可以注册自定义日志记录/代码跟踪，也可以是自定义监控逻辑。可以通过实现 kafka.producer.async.CallbackHandler 接口，并设置 class 的 callback.handler 配置。

可以自己指定一个 Encoder，用于序列化数据

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

默认的是： kafka.serializer.DefaultEncoder

通过用户指定一个 Partitioner，用于负载均衡
路由决策是由 kafka.producer.Partitioner.决定。

```
interface Partitioner<T> {
```

```
  int partition(T key, int numPartitions);
}
```
分区 api 使用 key 和可用 broker 分区的数量生成一个分区 id。这个 id 做为从一个存储了 brokerIds 和分区的列中检索一个 broker 分区的索引。
默认的策略是 hash(key)%numPartitions，如果 key 为空，会随机分配一个分区。可以通过 partitioner.class 配置一个自定义的分区策略。

Consumer APIs

我们有两个层次的 consumer api。 low-level “simple” API 维护一个连接到单个 broker 的链接(conection)。并有一个发送到服务器的网络请示的对应关系。这个 api 是完全无状态的。随着偏移量的增加，允许用户自己维护这些元数据。

high-level api 隐藏了 brokers 与 consumer 之间的细节。允许消费集群上的消费，而不需要关注低层机器拓扑。higth-level api 也提供了用于匹配发布订阅的过滤器表达式（即：白名单或黑名单正则）

Low-level API

```
class SimpleConsumer {

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

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

  /**
   * Get a list of valid offsets (up to maxSize) before the given time.
   * The result is a list of offsets, in descending order.
   * @param time: time in millisecs,
   *              if set to OffsetRequest$.MODULE$.LATEST_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);
}
```
low-level api 是为了实现一些高级 api 提供的低层接口。因为一些离线消费者特殊需求。

High-level API

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

  interface ConsumerConnector {
```

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

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

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

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

5.2 网络层

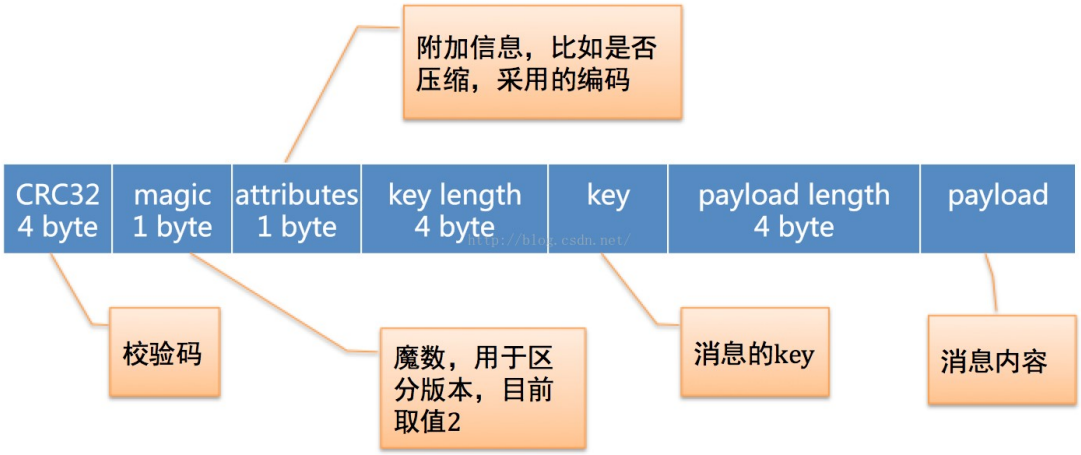
网络层是一个 NIO 服务器,这里不做过多描术。sendfile 是通过给 MessageSet 接口地址的 writeTo 方法实现。这使得支持文件备份的信息集合，可以使用更有效的 transferTo 方法，而不是使用内置的缓冲写数据。该线程模型是一个单一的 **acceptor** 线程和 **n** 个 **processor** 线程组成。每个都处理一个固定数量的连接。

5.3 消息

消息是由一个固定长度的头，一个可变长的模糊含义的 key 的字节数组，一个可变长的模糊含义的值的字节数组组成。
头包含以下字段:

A CRC32 checksum to detect corruption or truncation.
A format version. 版本号
An attributes identifier
A timestamp
让 key 和 value 含义模糊是正确的决定：现在正在做序列化库已经有了很大的进步，任何特写的选择对使用都是不正确的。不用说，一个特写的应用程序使用 kafka 很可能会指定一个特定的序列化类型。MessageSet 接口是一个简单的迭代器，用于从 NIO 通道中读写大量的消息。

5.4 消息格式

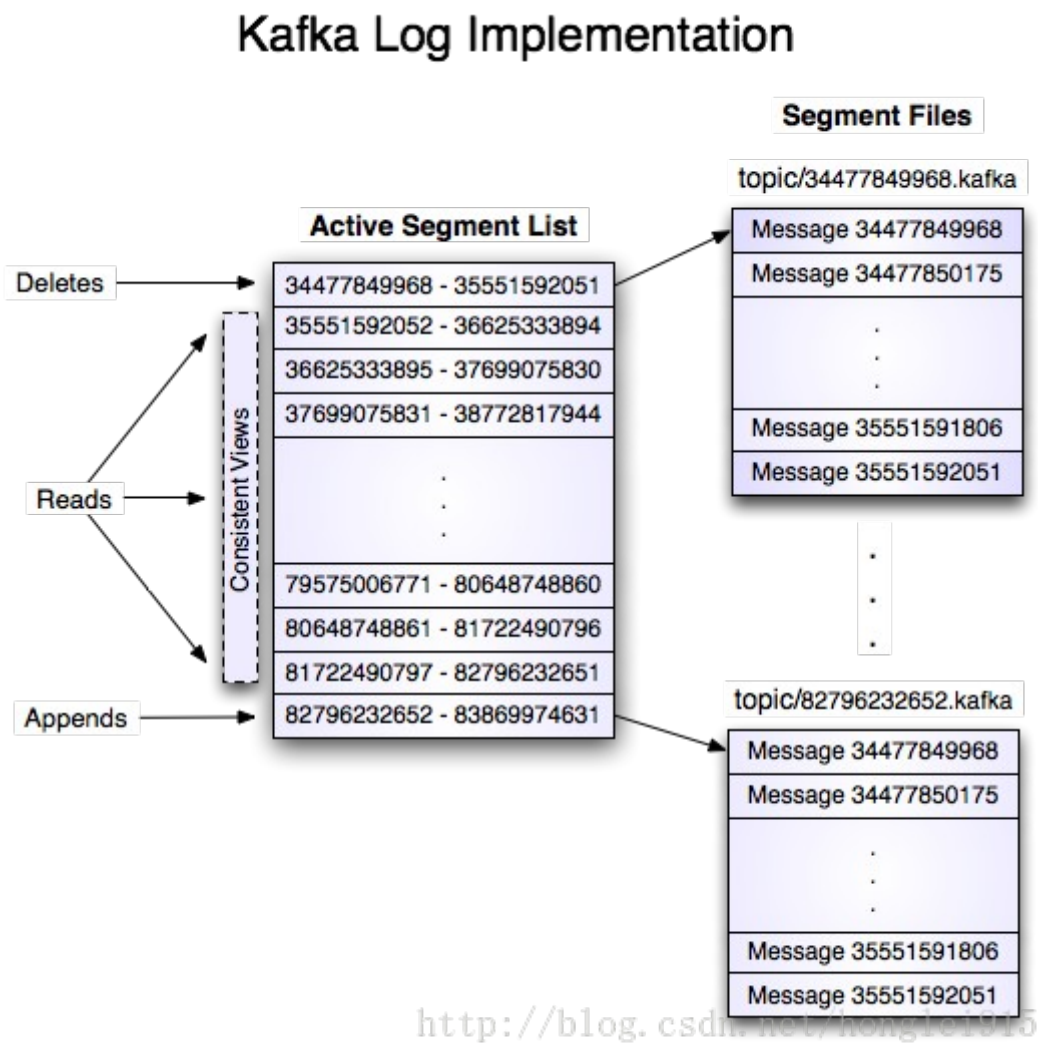


5.5 日志

一个叫做“my_topic”且有两个分区的的 topic,它的日志有两个文件夹组成，`my_topic_0` 和 `my_topic_1`,每个文件夹里放着具体的数据文件，每个数据文件都是一系列的日志实体，每个日志实体有一个 4 个字节的整数 N 标注消息的长度，后边跟着 N 个字节的消息。每个消息都可以由一个 64 位的整数 offset 标注，offset 标注了这条消息在发送到这个分区的消息流中的起始位置。每个日志文件的名称都是这个文件第一条日志的 offset.所以第一个日志文件的名字就是 00000000000.kafka.所以每相邻的两个文件名字的差就是一个数字 S,S 差不多就是配置文件中指定的日志文件的最大容量。

消息的格式都由一个统一的接口维护，所以消息可以在 producer,broker 和 consumer 之间无缝的传递。存储在硬盘上的消息格式如下所示：

```
消息长度:      4 bytes (value: 1+4+n)
版本号:        1 byte
CRC 校验码:    4 bytes
具体的消息:    n bytes
```



使用消息偏移量做为消息 id 是不寻常的。我们最初是通过生产者创建一伴 GUID，然后维护一个 GUID 与偏移量的映射来做。但是即然客户端要维护每个服务器 id,那么全局 GUID 就没什么意义了。另外，维护一个随机 id 与偏移量的映射需要每重的索引数据结构，必须同步到硬盘上。因此简化查询结构，我们决定使用一个简单的原子计数器（每个分区一个），可以与分区 id 和节点 id 结合起来来唯一定位一条消息。这让消息查询结构非常简单。一旦我们钉了计数器，直接使用偏移量就很自然了，毕竟计数器对一个分区是单调递增了。由于偏移量的具体实现是对客户 api 隐藏的，因此我们可以做的更有效。

写操作

消息被不断的追加到最后一个日志的末尾，当日志的大小达到一个指定的值时就会产生一个新的文件。对于写操作有两个参数，一个规定了消息的数量达到这个值时必须将数据刷新到硬盘上，另外一个规定了刷新到硬盘的时间间隔，这对数据的持久性是个保证，在系统崩溃的时候只会丢失一定数量的消息或者一个时间段的消息。

读操作

读操作需要两个参数：一个 64 位的 offset 和一个 S 字节的最大读取量。S 通常比单个消息的大小要大，但在一些个别消息比较大的情况下，S 会小于单个消息的大小。

这种情况下读操作会不断重试，每次重试都会将读取量加倍，直到读取到一个完整的消息。可以配置单个消息的最大值，这样服务器就会拒绝大小超过这个值的消息。也可以给客户端指定一个尝试读取的最大上限，避免为了读到一个完整的消息而无限次的重试。
在实际执行读取操纵时，首先需要定位数据所在的日志文件，然后根据 offset 计算出在这个日志中的 offset(前面的的 offset 是整个分区的 offset),然后在这个 offset 的位置进行读取。定位操作是由二分查找法完成的，Kafka 在内存中为每个文件维护了 offset 的范围。
下面是发送给 consumer 的结果的格式:

```
MessageSetSend (fetch result)

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

MultiMessageSetSend (multiFetch result)

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

删除

日志管理器允许定制删除策略。目前的策略是删除修改时间在 N 天之前的日志（按时间删除），也可以使用另外一个策略：保留最后的 N GB 数据的策略(按大小删除)。为了避免在删除时阻塞读操作，采用了 copy-on-write 形式的实现，删除操作进行时，读取操作的二分查找功能实际是在一个静态的快照副本上进行的，这类似于 Java 的 CopyOnWriteArrayList。

日志提供获取最新写入的消息的能力，让客户可以开始定阅"现在"的数据。这对于那些因为消息过期数据而失败的消息都很有用。这种情况，如果客户端指定了一个不存在偏移量，将返回 OutOfRangeException，可以重置它自己。

6. 操作

这里有一些 LinkedIn 公司在生产系统的使用经验。

6.1 kafka 基本操作

这里有一些 kafka 集群是非常通用的操作。在 kafka 分布式的 bin/目录下，本章介绍的所有工具都是有效的。每个工具如果以无参方式运行，都将打印出工具的命令行选项细节。

增加和删除主题(topics)
你可以手动添加一个主题，也可以自动创建主题。当数据第一次发布，并且没有指定主题时，主题会被自动创建。如果主题是被自动创建的，你可能希望通过默认配置进行调整。

主题通过以下工具添加和修改：

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

复制因子用来控制写入的消息会被多少服务器复制。如果你的复制因子是 3,那么如果 2 个服务失效，也不会丢失数据。我们建议你复制因子设置为 2 或者 3，这样在不中断数据消费的情况下，机器可以透明的弹性变化。

分区数量用来控制多数据日志会被共享。分区数量有几个影响，首先，每个分区必须安装在单独的机器上。因此如果你有 20 个分区，那么将不超过 20 台服务器来处理（不计算副本）。最后，分区数量决定了消费者的最大并发数。
详细请看这里

每个共享的分区日志都放在 kafka 日志目录下属于分区自己的文件夹下。这些文件夹名字是由主题名，"-"和分区 id 组成。因为文件夹名不能超过 255 个字节，所以主题名也是有限止的。我们假定分区数量不能超过 100,000 个，那么主题名就不能超过 249 个，这是为了给"-"(一个字符)和分区 id(5 个字符 999999)留下足够空间。

命令行中的添加的配置会覆盖服务器中的默认配置。每个主题的完整配置文档请看[]这里](http://kafka.apache.org/documentation.html#topic-config)

修改主题
你可以使用相同的主题工具来修改配置或者主题的分区

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

要注意，通过语义分区数据的用例中，添加分区是不会改变现数据的分区，因此这可能扰乱消费者。也就是说，如果数据是通过 hash(key)%number_of_partitions（分区数），那么添加分区会导致整个分区重新调整，但是 kafka 不会以任何方式自动重新分布数据。

增加配置：

```
> bin/kafka-topics.sh --zookeeper zk_host:port/chroot --alter --topic my_topic_name
--config x=y
```

删除配置

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

删除选项默认是关闭的，需要配置来启动删除功能：

```
delete.topic.enable=true
```

kafka 目前不支持减少主题的分区数量。

平滑关闭

kafka 集群会自动检测 broker 是否关闭或失效，并且选举新的 leader。无论是服务器失效了，还是为了维护或配置故意关闭，都会导致 kafk 集群自动检测。对于后者这种情况（维护或配

置），kafka 支持一种更平滑的机制来停止一个服务器而不是直接干掉它(kill)。
平滑有两个优化：
(1). 在平滑关闭时，kafka 会把所有日志同步到磁盘上，以避免重启后重新恢复数据（即：检查所有日志尾部的消息的完整性）。日志恢复需要时间，因此这加快了服务重启的速度。
(2). 在关闭之前将优先迁移 leader 服务器上的分区数据，这样会让领导转移的更快，最大限度的减少每个分区不可用的时间到毫秒内。
如果服务是被平滑关闭而不是硬生生的 kill，那么同步日志将是自动完成，不过 leadership 的转移还需要配合一个特殊的设置：

controlled.shutdown.enable=true

注意：只有 broker 上的所有分区都有副本（即：副本因子大于 1,到少一个副本是激活的）关闭都会成功。这可能就是你想要的，因为关闭最后那个副本会导致主题分区不可用（说白了要关闭一个，至少还有一个热备的，否则，只有自己一人，那么自己关闭了，必然导致服务不可用）。

平衡 **leadership**
只要一个服务器停止或崩溃了，leadership 就会转移到分区的其它副本上。即默认情况下，只要 broker 重启了，它只能当做分区的 follower。也就是说它不能被客户端访问用来读写了。
为了避免这种不平衡，kafka 有一个首选副本的概念。如果一个分区的副本列表是 1,5,9 三个节点。那么节点 1 是 leader 的首选副本。因为它是最早位于列表中的。你可以通过以下命令让集群重新恢复原有副本为 leader.

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

运行这个命令太烦锁，我们可以使用以下配置来让 kafka 自动平衡：

auto.leader.rebalance.enable=true

通过机架平衡副本（机架：就是机房中的机架－－0.10.0.0 版本新增加了机架感知功能）

机架感知特性通过不同的机架，扩展了同一分区的不同副本。这把 kafka 从保证 broker 失效扩大覆盖到机架故障，限制了一个机架上所有 broker 一次性全部失效导致数据丢失的风险。（小道消息：此功能是由 netflix 提供）

你可以通过以下配置，来指定一个 broker 属于某个特写的机架。

broker.rack=my-rack-id

当一个主题被创建，修改或副本被重新分布，机架约束会被兑现，以确保副本可以跨越多个机架（一个副本到少跨越 m 个不同的机架，m 由机架数，副本因子确定）。
分配 broker 副本的算法保证每个 broker 的 leader 数量是恒定的，无论 broker 在机架上如何分布。这保证了吞吐量的均衡。
如果机架上分配不同数量的 broker，那么副本分配将不均衡。 具有少量 broker 的机架将具有更多的副本，意味着他们将使用更多的存储并且向副本中存储更的资源。因此，明智的做法是每个机架配置相同的 brokers.

集群间的数据镜像
略... todo

检查消费者位置
有时，查看消费者位置非常有用。我们有一个工具可以展示一个消费者中的所有消费者位置。查看一个消费者组名为"my-group"，主题为"my-topic"的消费者位置，运行工具如下：

> bin/kafka-run-class.sh kafka.tools.ConsumerOffsetChecker --zookeeper localhost:2181 --group test
Group Topic Pid Offset logSize Lag Owner
my-group my-topic 0 0 0 0 test_jkreps-mn-1394154511599-60744496-0
my-group my-topic 1 0 0 0 test_jkreps-mn-1394154521217-1a0be913-0

从 0.9.0 版本，kafka.tools.ConsumerOffsetChecker 工具已经过时了，请使用 kafka.admin.ConsumerGroupCommand(或 bin/kafka-consumer-groups.sh 脚本)

管理消费者组

使用 ConsumerGroupCommand 工具，可以查询（列表、详情），删除 消费者组。比如：列出所有消费者组

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

test-consumer-group
1
2
3
4
为了查看上个例子中的偏移量，我们展示消费者组的详情如下：

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

GROUP TOPIC PARTITION CURRENT-OFFSET LOG-END-OFFSET LAG OWNER
test-consumer-group test-foo 0 1 3 2 test-consumer-group_postamac.local-1456198719410-29ccd54f-0
1
2
3
4
5
当你使用新的 consumer api 时，你可以使用”–new-consumer”来管理组：

> bin/kafka-consumer-groups.sh --new-consumer --bootstrap-server broker1:9092 --list
1
集群扩展
向 kafka 集群中添加服务器是容易的，仅仅给他们分配一个唯一的 brokerId，并且把这台机器上的 kafka 启动想来即可。但是这些新服务器不会自动分配任何数据分区。除非新创建主题，或者数据分区移动到它们上。因此，当你向集群添加一个新的机器时，你想把现存的数据迁移到这些机器。

迁移数据的过程需要手动启动，但是完全自动化执行。迁移过程的低层实现是这样的：kafka 增加一个新的服务器，这个新的服务器做为一个分区的 follower。允许它完全复制现存的分区数据。当新的服务器完全复制了此分区中的数据后，并且加了同步副本（in-sync）时，现存副本中的一个就可以删除它们的分区数据了。
（解释：新增加了服务器，不影响原有数据，这就导致大家都不使用它。除非两种情况：把原有分区数据重新分给它，另外，新创建主题。通知情况，新创建主题并不频繁。那么就需要重新分配数据。重新分配数据需要手动启动，而且新的机器也只能做 follower）

分区调整工个可以用于移动分区到各个 broker。一个理想的分区分配应该能保证数据负载均衡和分区数量均衡。分区调整工具没有能力智能分调整负载。因此，管理员必须找出哪些主题或分区需要移动。

分区调整工个可以运行以下 3 种方式（互斥）：

–generate：这种模式，给定一个主题列表和一个 broker 列表，工具生成一个重新分配报告（把指定主题的所有分区都移动新有 brokers 上），这个选项提供了一个方便的生成重新分配的计划。

–execute : 这种模式，工具根据用户提供的重新调整计划（–generate 生成的， 使用 reassignment-json-file 选项指定）。这个可以是一个自定义的重新调整计划，也可以使用 –generate 生成的

–verify: 在这种模式下，工具验证由–execute 执行的重新调整。验证状态为：成功、失败、正在进行中。

自动迁移数据到新机器上
重新调整工具可以用来把现有 brokers 集上的部分主题迁移到新的机器上。当扩展一个存在的集群时，这非常有用。因为很容易把全部主题都移到新的 brokers 集中，而不是一次只移一个分区。
如果要这么做，我们必须提供一批需要移动的主题列表，和一批新加入的 broker 列表。工具将均匀的把给定的主题列表中的所有分区移到新的 broker 集上。迁移过程中，复制因子保持不变。
指定的主题集中的所有分区副本都从旧的 brokers 集中迁移到新的 brokers 中。

例如，下面的示例将把 foo1,foo2 两个主题的所有分区都迁移到新的 broker 机器 5,6 上。最后，foo1,foo2 两个主题的所有分区都厚在 brokers 5,6 上。

因为工个接受 json 文件格式的参数（包含：需要迁移的主题列表），所以首先创建一个 json 文件，要把需要迁移的主题配置到文件中

```
> cat topics-to-move.json
{"topics": [{"topic": "foo1"},
        {"topic": "foo2"}],
 "version":1
}
1
2
```

3
4
5
一旦 json 文件就绪，使用分区重新调整工具生成一个分配计划。

> bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --topics-to-move-json-file topics-to-move.json --broker-list "5,6" --generate
 Current partition replica assignment

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

 Proposed partition reassignment configuration

 {"version":1,
  "partitions":[{"topic":"foo1","partition":2,"replicas":[5,6]},
                {"topic":"foo1","partition":0,"replicas":[5,6]},
                {"topic":"foo2","partition":2,"replicas":[5,6]},
                {"topic":"foo2","partition":0,"replicas":[5,6]},
                {"topic":"foo1","partition":1,"replicas":[5,6]},
                {"topic":"foo2","partition":1,"replicas":[5,6]}]
 }
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18

19
20
21
22
工具生成了一个把主题 foo1,foo2 所有分区迁移到 brokers 5,6 上的计划。注意，分区迁移还没有开始。它只是告诉你当前分配计划和新计划的提议。为了防止万一需要回滚，新的计划应该保存起来。
新的调整计划应该保存成一个 json 文件（如：expand-cluster-reassignment.json)，并以–execute 选项的方式，如下：

> bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --reassignment-json-file expand-cluster-reassignment.json --execute
Current partition replica assignment

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

Save this to use as the --reassignment-json-file option during rollback
Successfully started reassignment of partitions

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

1
2
3
4
5
6
7
8
9
10
11
12
13
14

15
16
17
18
19
20
21
22
23
最后，–verify 可以肜来验证重新调整的状态。注意，那个调整计划文件 expand-cluster-reassignment.json 还要再次使用，如下：

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

1
2
3
4
5
6
7
8
9
10
自定义分区分配和迁移
分区重新调整工具还能用于有选择性把一个分区的所有副本迁移到指定的一系列 brokers 中。当使用这种方式，它假定用户已经知道重新分配的计划，因此不需要指定一个重新分配计划，也就是跳过–generate 步骤，直接执行–execute 步骤。

例如：以下例子把 foo1 主题中的分区 0 移到 brokers 5,6 上，并且把 foo2 主题中的分区 1 移到 brokers 2,3 上。
第一步是手动创建一个 json 格式的分配计划文件。

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

1
2
3

然后执行：–execute

> bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --reassignment-json-file custom-reassignment.json --execute
Current partition replica assignment

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

Save this to use as the --reassignment-json-file option during rollback
Successfully started reassignment of partitions
{"version":1,
 "partitions":[{"topic":"foo1","partition":0,"replicas":[5,6]},
               {"topic":"foo2","partition":1,"replicas":[2,3]}]
}
1
2
3
4
5
6
7
8
9
10
11
12
13
14
执行验证： –verify

bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --reassignment-json-file custom-reassignment.json --verify
Status of partition reassignment:
Reassignment of partition [foo1,0] completed successfully
Reassignment of partition [foo2,1] completed successfully
1
2
3
4
5
除役（删除）brokers
重新分配工具还没有能力为将除役的 brokers 自动生成一个重新调整的计划。因此，管理员必

须自己想一个重新调整计划来移动将要被除役的 brokers 上的分区到其它 brokers 上。这个过程会非常乏味，并且还要保证上分区数据不会全部移到同一台机器上。为了让这个过程轻松，我们在未来的版本上增加这样的工具支持。

增加复制因子
增加一个现存分区的复制因子是很容易的。只要在自定义的重新调整分区的 json 文件中指定一个扩展的副本，并且使用–execute 选项就可以增加一个指定分区的复制因子

例如，下面的示例是增加主题 foo 的分区 0 的复制因子，把复制因子 1 增加到 3。在增加前，那个分区的唯一副本在 broker 5 上。同时，我们将增加更多的副本到 broker 6 和 7 上（复制因子增加了，就必须有对应的 broker 来存储新增的副本）。

第一步：创建一个重新调整计划文件：

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

使用–execute 选项，启动重新分配进程：

```
> bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --reassignment-json-file
increase-replication-factor.json --execute
Current partition replica assignment

{"version":1,
"partitions":[{"topic":"foo","partition":0,"replicas":[5]}]}

Save this to use as the --reassignment-json-file option during rollback
Successfully started reassignment of partitions
{"version":1,
"partitions":[{"topic":"foo","partition":0,"replicas":[5,6,7]}]}
```

使用 –verify 选面验证：

```
bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --reassignment-json-file
```

increase-replication-factor.json --verify
Status of partition reassignment:
Reassignment of partition [foo,0] completed successfully
1
2
3
4
你也可以使用 kafka-topics 工具来验证：

> bin/kafka-topics.sh --zookeeper localhost:2181 --topic foo --describe
Topic:foo PartitionCount:1 ReplicationFactor:3 Configs:
 Topic: foo Partition: 0 Leader: 5 Replicas: 5,6,7 Isr: 5,6,7
1
2
3
4
设置配额
可以通过在 brokers 上配置默认的配额应用到所有 clientId 上。默认情况，每个 clientId 都会被限额。下面就是配置每个消费者和生者 clientId 的限额为：10MB/秒。
 quota.producer.default=10485760
 quota.consumer.default=10485760
1
2
也可以对每个 client 自定义配置

> bin/kafka-configs.sh --zookeeper localhost:2181 --alter --add-config 'producer_byte_rate=1024,consumer_byte_rate=2048' --entity-name clientA --entity-type clients
Updated config for clientId: "clientA".
1
2
3
下面是如果显示一个给定 client 的配额：

> ./kafka-configs.sh --zookeeper localhost:2181 --describe --entity-name clientA --entity-type clients
Configs for clients:clientA are producer_byte_rate=1024,consumer_byte_rate=2048
1
2
3

6.2 数据中心

一些部署可能需要管理一个跨越多个数据中心的数据管道。我们的建议是在每个数据中心本地部署一个 kafka 集群，每个数据中心的应该只与本地的 kafka 集群交互。不同数据中心的

kafka 集群，通过镜像实现数据同步。

这种部署械允许数据中心做为独立的实体。并允许我们管理协调数据中心之间的复制。这允许每个设置独立操作，即使数据中心的链接是失效的：镜像会在链接成功后自动同步。

对于需要所有数据全局视图的应用，你可使用镜像来提供从所有数据中心聚合好数据的集群。这些聚合的集群用于需要全部数据集合的应用。

这不是唯一的部署方式，因为通过网络，很容易对远程 kafka 集群进行读写，但是每明显，这会增加请求延迟。

kafka 是通过批处理数据才获得高吞吐量，即使在高延迟的 connection 上。因此，为了保证高吞吐量，我们需要设置 TCP socket buffer size。这个设置是通过：
socket.send.buffer.bytes and socket.receive.buffer.bytes 来配置的，具体看这里;

在一个高延迟的网络上，只创建一个跨越多个数据中心的单一 kafka 集群是不明智的，这会即会导致 kafka 写也会导致 zookeeper 写操作的高延迟复制。

6.3 重要配置

客户端重要配置
压缩
同步 vs 异步生产
批处理大小(对异步而言)
最重要的消费者配置是抓取数据的大小。

生产服务器配置

```
# Replication configurations
num.replica.fetchers=4
replica.fetch.max.bytes=1048576
replica.fetch.wait.max.ms=500
replica.high.watermark.checkpoint.interval.ms=5000
replica.socket.timeout.ms=30000
replica.socket.receive.buffer.bytes=65536
replica.lag.time.max.ms=10000

controller.socket.timeout.ms=30000
controller.message.queue.size=10

# Log configuration
num.partitions=8
message.max.bytes=1000000
auto.create.topics.enable=true
log.index.interval.bytes=4096
log.index.size.max.bytes=10485760
log.retention.hours=168
```

log.flush.interval.ms=10000
log.flush.interval.messages=20000
log.flush.scheduler.interval.ms=2000
log.roll.hours=168
log.retention.check.interval.ms=300000
log.segment.bytes=1073741824

# ZK configuration
zookeeper.connection.timeout.ms=6000
zookeeper.sync.time.ms=2000

# Socket server configuration
num.io.threads=8
num.network.threads=8
socket.request.max.bytes=104857600
socket.receive.buffer.bytes=1048576
socket.send.buffer.bytes=1048576
queued.max.requests=16
fetch.purgatory.purge.interval.requests=100
producer.purgatory.purge.interval.requests=100

6.4 java 版本

从安全角度考虑，我们推荐使用 jdk1.8 及以上版本。linkedIn 使用的是 jdk.8 u5，及Ｇ 1 垃圾收集器。如果你想使用 G1，并且使用 jdk1.7，要确保 u51 或更新版本。 linkedIn 测试了 u21,但是 GC 有很多问题。
linedIn 的配置如下：

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

作为参考，这里有一些 linkedIn 公司最繁忙的集群中的一些属性：

60 个 broker
50K 个分区（复制因子为 2)
800k 条消息/每秒
300MB/秒 入站（写），1GB/秒 出站（读）
The tuning looks fairly aggressive, but all of the brokers in that cluster have a 90% GC pause time of about 21ms, and they’re doing less than 1 young GC per second.

6.5 硬件和操作系统

操作系统
kafka 在 unix 系统上运行良好，在 linux 和 solaris 也测试过

windows 上运行会有一些问题，它支持的不是很好。

通常不需要太多 os 级别的调整，但是确实有两个重要 os 级别的配置：
文件描述符限制：kafka 中的日志段和打开 connection 需要使用文件描述符。如果一个 broker 有很多分区，那么它需要到少(number_of_partitions)*(partions_size/segment_size) 个段需要追踪。我们建议至少允许 100000 文件描述符。
最大 socket buffer size：可以提高数据中心之间的高性能数据传输；

应用 vs os 刷新管理
kafka 通常是立即把数据写入到文件。同时支持配置刷新策略来控制数据什么时间从 os cache 中刷新到磁盘。这个刷新策略可以控制，在一段时间后或一定量的消息后强制把数据从缓存中刷新到磁盘。

6.7 zookeeper

稳定版
目前最稳定版是 34，最新版本是 3.4.6。zkClient 是 kafka 用于与 zookeeper 交互的客户端层。

操作 zookeeper
操作上我们通过下面说明安装一个健壮的 zookeeper：

物理/硬件/网络冗余布局：不要把它们放在同一个机架上，合适即可（但也不要走极端）。电源和网络有冗余。一个典型的 zookeeper 应该有 5-7 个服务器，这样可以容忍 2-3 台挂掉。当然，如果你有一个小的部署，那么使用 3 台服务器也是可以接受的。但是要记住，你仅仅能容忍 1 台服务器挂掉。

i/o 分离：如果你有一个非常大量的写操作，你可能希望有一组专用的磁盘来记录事先日志。写事务日志是同步的（但是是批量的），因此并行操作可以大在提高性能。ZooKeeper 快速就是这样一个：同源并行写，理想情况下应该与事务日志分开写的功能。快照是异步写到磁盘的。你可以用 dataLogDir 参数配置一个服务器独立的磁盘组。

应用分离：除非你了解别的程序的应用模式，否则不要把别别的应用与 zookeeper 安装在一想。最好把 zookeeper 隔离安装。

zookeeper 配置：使用的是 java，因此确保有足够的堆内存（我们通常使用 3-5 G，不过这要看数据量）。不幸的是，我们没有一个合适的公式。但是要记信，对于更多的 zookeeper 状态，会让快照变大，大的快照会影响恢复时间。如果快照变的太大，那么你就需要增 initLimit 参数的设置，以给服务器足够的时间用来恢复和加入到集群。

监控：JMX 和 4lw 命令都是非常有用的，他们在某些情况下是重复的（在这种情况下，我们更喜欢 4lw，他们更有预见性，至少，他们在监控设施上与 LI 工作的更好）。

不要过度建设集群
总的来说，我们试图保持 zookeeper 系统尽可能的小，并且尽可能的简单。我们尽可能不做任何比官网花哨的配置或程序布局。出于这个原因，我们跳过操作系统封装的版本，因为操作系统封装的版本，为了更好的方式表达它，会把一些事情放在操作系统层面，这会导致混乱。