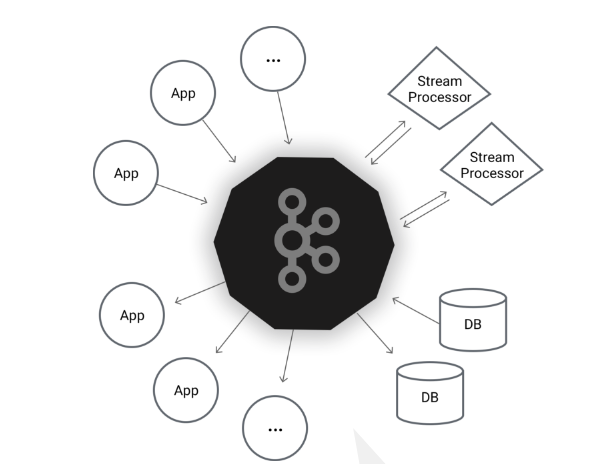
**kafka**

# 概述



kafka是一个分布式流平台。

1. 允许发布和订阅流式数据。在这一点上类似于MQ系统
2. 能够以容错式的方式存储流式数据
3. 能够实时处理流式数据。

使用场景：

构建实时流数据管道可以在系统和应用件可靠获取数据。

构建实时流数据处理应用来处理和响应实时流数据。

* apache kafak是一个开源消息系统，由scala写成。是由Apache软件基金会开发的一个开源消息系统项目。
* kafka最初是由LinkedIn开发，并于2011年初开源。2012年10月从Apache incubator毕业。该项目的目标是为处理实时数据提供一个统一、高通量、低等待的平台。
* kafka是一个分布式消息队列(生产者、消费者的功能)。它提供了类似于JMS的特性、但是在设计实现上完全不同，此外它并不是JMS规范的实现。
* kafka对消息保存时根据Topic进行归类，发送消息者称为Producer，消息接收者称为Consumer，此外kafka集群有多个kafka实例组成，每个实例(server)成为broker。
* 无论是kafka集群，还是producer和consumer都依赖于zookeeper集群保存一些meta信息，来保证系统可用性。

# kafka和其它MQ的区别

1. kafka没有消息确认机制
2. kafka 只有发布订阅模式
3. 吞吐量：kafka具有高吞吐量，内部采用消息的批处理，zero-copy机制，数据的存储和获取是本地磁盘顺序批量操作，具有O(1)的复杂度，消息处理的效率很高。
4. 可靠性：MQ支持对消息的可靠的传递，支持事物，不支持批量的操作；基于存储的可靠性的要求存储可以采用内存或者硬盘。
5. 抗压性：kafka在大数据量下依然保持O(1)的方式提供消息持久化能力，即使对TB级以上数据也能保证常数时间复杂度的访问性能。而MQ则会出现性能下降。
6. 是否支持离线数据处理和实时数据处理：

kafka同时支持离线数据处理和实时数据处理。

1. 扩展性比较：kafka支持在线水平扩展。
2. 消息安全性：kafka可以保证在绝大多数情况下消息的安全性。当消息算作commit的时候才会暴露给consumer，并保证at-least-once的投递原则。
3. 可用性，稳定容错性：kafka天然支持HA，leader挂了，follower自动选举出新的leader
4. 适用场景：kafka是分布式消息系统。mq是基础的消息系统。kafka适用于流式计算。如果需要消息可靠性传输，使用MQ，如果处理海量日志数据等，使用kafka。如果需要同时处理在线应用(消息)和离线应用(数据文件，日志)

# 使用场景

## 消息队列的使用

相比传统消息系统，kafka具有更高的吞吐量，内建分组，复制和容错机制。可以将kafka当做传统消息系统一样来使用。

## 网站活动跟踪

Kafka的初始目标是能够将用户活动流管道重建为一组实时发布 - 订阅的源。也就是说网站活动例如浏览页面，搜索或者其它用户可能的操作会被publish到中央的topics(每个活动类型对应一个topic)，这些源可以被订阅取到活动数据然后做一系列事情,比如实时计算或者实时监控或者存储到hadoop等离线数据仓库系统做离线处理(比如经过离线计算后生成报表之类)。

活动跟踪通常量比较大，因为为每个用户页面视图生成许多活动消息。

## 度量

卡夫卡通常用于运行监控数据。这涉及从分布式应用程序汇总统计数据以生成操作数据的集中式源。

## 日志聚合

许多人使用Kafka作为日志聚合解决方案的替代品。日志聚合通常从服务器收集物理日志文件，并将其置于中央位置（可能是文件服务器或HDFS）进行处理。Kafka抽象了提取文件的细节，并将日志或事件数据作为消息流进行更清晰的抽象。这样可以实现更低延迟的处理，并且更容易支持多个数据源和分布式数据消耗.。与Scribe或Flume等以日志为中心的系统相比，Kafka提供同样出色的性能，由复制产生的更强大的持久性保证以及更低的端到端延迟。

## 流式处理

Kafka的许多用户在处理管道中处理数据，这些数据由多个阶段组成，其中原始输入数据从Kafka主题中消耗，然后聚合，丰富或以其他方式转化为新的主题，以供进一步消费或后续处理。例如，用于推荐新闻文章的处理流水线可以从RSS提要抓取文章内容并将其发布到“文章”主题; 进一步的处理可以对这个内容进行归一化或者重复删除，并且将已清理的文章内容发布到新的主题; 最终处理阶段可能会尝试将这些内容推荐给用户。这种处理流水线基于各个主题创建实时数据流的图表。从0.10.0.0开始，这是一个轻量但功能强大的流处理库，称为Kafka Streams 在Apache Kafka中可用于执行如上所述的数据处理。除了Kafka Streams之外，替代性的开源流处理工具还包括Apache Storm和 Apache Samza。

## 事件源(Event sourcing)

Event sourcing 是一种应用程序设计的风格，即状态改变以时间排序的序列进行记录。Kafka支持超大日志数据，使它可以成为基于Evnet sourcing设计的应用程序的优秀后端。

## 提交日志(Commit Log)

Kafka可以作为分布式系统的一种外部提交日志

日志有助于复制节点之间的数据，并作为失败节点恢复数据的重新同步机制。Kafka中的日志压缩功能有助于支持这种用法。在这个用法中，Kafka与Apache BookKeeper项目类似。

# API

Producer api：允许应用程序作为生产者发送流式数据到kafka集群中的topics

Consumer api: 允许应用程序作为消费者从kafka集群中的topics读取数据

Streams api：允许应用程序将输入的流式数据转换然后再次发送到指定的topics

Connect api：允许连接器从某个源系统或应用程序持续拉数据到kafka或者从kafka推送数据到某个消费系统或应用程序。

AdminClient api：允许管理和监控topics,brokers以及其它kafka对象。

## Producer api

允许应用程序作为生产者发送流式数据到kafka集群中的topics

定义：public class KafkaProducer<K,V> extends java.lang.Object implements Producer<K,V>

Producer是线程安全的，多线程环境中，推荐使用一个单一的producer，将会比每个线程一个producer的速度要快。

示例如下：

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();

1. send方法是异步的。调用的时候会将一条记录添加到发送缓冲区，然后立即返回。这种方式允许生产者批量发送数据以提高效率。
2. acks

0：这意味着生产者producer不等待来自broker同步完成的确认就继续发送下一条（批）消息。此选项提供最低的延迟但最弱的耐久性保证，因为其没有任何确认机制。

1:意味着producer在leader已成功收到的数据并得到确认后发送下一条消息。等待leader的确认后就返回，而不管partion的follower是否已经完成。

2: 这意味着producer在follower副本确认接收到数据后才算一次发送完成。此选项提供最好的耐久性，我们保证没有信息将丢失，只要至少一个同步副本保持存活

如果发送请求失败，生产者会自动重试，尽管已经设置restries为0？(不确定)。允许重试将会导致有可能对同一条消息多次传输。

producer为每一个partition维护一个未发送记录的缓冲区。这些bufers各自的大小可以通过batch.size指定。将该值设大了经会导致批量提交的效率但也会更多的占用内存。

默认情况下，一个buffer是可以立即发送的，尽管还有未使用的空间。如果你想降低请求的次数，可以设置linger.ms为比0更大的值。linger.ms表示生产者在前一次发送之后等待多久后再次发送，这样做可以使更多的消息批量提交以提高缓存利用率。

在上面的示例代码中，如果有100条消息产生，在设置linger.ms为1ms的情况下，所有100条数据将会在一起请求中被发送。注意：及时到达的批量消息也会批量被提交尽管linger.ms被设置为0。因此在负载比较重的情况下可以忽略linger.ms的设置。但不管怎样，设置linger.ms大于0的某个值可以负载比较小的情况起到提高发送效率的作用。

buffer.memory控制了producer分配所有buffers时的总内存大小。如果记录的发送速度比发送到服务器的速度快，那么这个缓冲区空间将被耗尽。当缓冲区空间耗尽时，另外的发送调用将被阻塞。 阻塞时间的阈值由max.block.ms决定，在此之后它将引发TimeoutException。

key.serializer和value.serializer指示如何将用户提供的ProducerRecord键和值对象转换为字节。 您可以使用包含的ByteArraySerializer或StringSerializer作为简单的字符串或字节类型。

从Kafka 0.11开始，KafkaProducer支持另外两种模式：幂等生产者和事务生产者

幂等生产者增强了kafka的发送机制，从至少发送一次精确到只发送一次。也就是说生产者重试将不会导致发送多份副本。

事务性生产者允许应用程序原子性的发送消息到多个分区（或主题）。

如何开启幂等性支持：

设置enable.idempotence为true，如果设置enable.idempotence为true，retries配置将会默认设为Integer.MAX\_VALUE并且acks 配置将会默认设置为all。

幂等生产者没有API改变，所以现有的应用程序不需要被修改来利用这个特征。

为了利用幂等性的生产者。应用程序级别的代码应该避免重复发送。也就是说，如果一个应用程序想要使用幂等性生产者，就不要去配置retries属性。另外，如果send(ProducerRecord)甚至在无限重试的情况下返回一个错误（例如，如果消息在发送之前在缓冲区中到期），那么建议关闭生产者并检查最后生成的消息的内容以确保 它不重复。最后，幂等生产者只能保证在单个会话中发送的消息具有同等性。

要使用事务性生产者和伴随的API，您必须设置transactional.id配置属性。 如果设置了transactional.id，则idempotence将自动启用，同时生产者配置将依赖于哪个幂等。

此外，交易中包含的主题应该配置为耐用性(replication.factor应该至少为3，而这些主题的min.insync.replicas应该设置为2.)。最后，为了使事务保证从端到端实现，消费者必须配置为只读取提交的消息。

transactional.id的目的是启用跨单个生产者实例的多个会话的事务恢复。 通常来自分区的有状态应用程序中的碎片标识符。 因此，对于在分区应用程序中运行的每个生产者实例，它应该是唯一的。

所有新的事务相关的API都是锁住并且在错误的时候会抛出异常。下面的例子演示了事务API的使用。同上一个例子相似，不同的是所有的100个消息在同一个事务中。

Properties props = new Properties();

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

props.put("transactional.id", "my-transactional-id");

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

producer.initTransactions();

try {

producer.beginTransaction();

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

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

producer.commitTransaction();

} catch (ProducerFencedException | OutOfOrderSequenceException | AuthorizationException e) {

// We can't recover from these exceptions, so our only option is to close the producer and exit.

producer.close();

} catch (KafkaException e) {

// For all other exceptions, just abort the transaction and try again.

producer.abortTransaction();

}

producer.close();

在上述例子中，每一个producer只能有一个打开的事务。所有在beginTransaction和commitTransaction的消息发送都是同一个事务的一部分。当transactional.id被指定后，所有这个生产者的消息发送都属于这个事务的一部分。

事务性生产者使用异常来传达错误状态。 特别是，在返回的Future中不需要为producer.send（）指定回调或调用.get（）：如果任何producer.send（）或事务性调用在执行期间遇到不可恢复的错误，将抛出KafkaException 异常。 有关检测事务性发送错误的更多详细信息，请参阅send（ProducerRecord）文档。

通过在收到KafkaException异常时调用producer.abortTransaction（），我们可以确保任何成功的写入被标记为中止，从而保持事务性的保证。

### producer的同步异步机制：

通过配置producer.type的值来确定是异步还是同步，默认为同步。async/sync 默认是sync。

如果设置为异步，那么提供了批量发送的功能,当满足以下其中一个条件的时候就触发发送。

1. batch.num.messages 异步发送 每次批量发送的条目 ；
2. queue.buffering.max.ms 异步发送的时候 发送时间间隔 单位是毫秒。

其次，异步发送消息的实现很简单，客户端消息发送过来以后，先放入到一个队列中然后就返回了。Producer再开启一个线程（ProducerSendThread）不断从队列中取出消息，然后调用同步发送消息的接口将消息发送给Broker。

### 分区partion发送

为了负载均衡一个topic可能会有多个partition，不同的partition存在在不同的broker里面，因此可以设定一定的partition的规则来确定什么样的消息发送到那个partition当中，代码如下：

public class CustomizePartitioner implements Partitioner{

/\*\*

\* 返回分区索引编号

\* @param key sendMessage时，输出的partKey

\* @param numPartitions topic中的分区总数

\* @return

\*/

@Override

public int partition(Object key, int numPartitions) {

System.out.println("key:" + key + " numPartitions:" + numPartitions);

String partKey = (String)key;

if ("part2".equals(partKey))

return 2;

return 0;

}

}

partition当中的metadata.broker.list：

该选项用于存放broker的元信息，官方翻译如下：

这个选项是用于一个producer启动的时候，在启动的时候producer会通过这个选项配置的broker的地址去获取元信息（topics, partitions and replicas）。它的格式如下，host1:port1,host2:port2。这个list可以是一个broker集合的子集。

需要注意的是producer是如何动态获取集群中的broker信息的变化呢，它又没有和zookeeper进行交互?

1,producer没有直接和zookeeper进行通信，但是broker集群会和zookeeper进行进行通信，然后broker集群会把元信息返回给producer；

2,producer在调用send方法的时候会去定时的刷新metadata信息（这自己又些以为，不太明白producer的定时刷新的机制）

3，由于在调用send之前可能会刷新metadata信息，因此可能会有一些延迟。如果不想要该延迟，把topic.metadata.refresh.interval.ms值改为-1，这样只有在发送失败时，才会重新刷新。Kafka的集群中如果某个partition所在的broker挂了，可以检查错误后重启重新加入集群，手动做rebalance，producer的连接会再次断掉，直到rebalance完成，那么刷新后取到的连接着中就会有这个新加入的broker。

除了上面说的之外，kafka还提供了新的producer写法，见http://kafka.apache.org/documentation.html#producerapi

## Consumer api

允许应用程序作为消费者从kafka集群中的topics读取数据

定义：public class KafkaConsumer<K,V>extends java.lang.Object implements Consumer<K,V>

该客户端透明地处理Kafka代理的故障，并透明地适应它在集群中获取迁移的主题分区。 该客户还与broker进行交互，消费者组允许消费者group消费时负载均衡。

客户保持TCP连接到必要的代理来获取数据。 使用后不关闭使用者将会泄漏这些连接。 消费者不是线程安全的。 有关更多详细信息，请参阅多线程处理。

### 偏移量和消费者位置

Kafka为分区中的每条记录保留一个数字偏移量。 该偏移量用作该分区内记录的唯一标识符，也表示消费者在分区中的位置。 例如，处于位置5的消费者已经消耗了具有偏移量0至4的记录，并且将接下来以偏移量5接收该记录。实际上有两个与消费者的用户有关的位置概念：

position:消费者给出下一个记录的偏移量。 这将比消费者在该分区中看到的最高偏移大一点。 每当消费者收到broker中的消息时，消费者就会自动前进（长）。

committed position:上次安全存储的偏移量。如果流程失败并重新启动，则这是消费者需要回复的偏移量。用户可以定期自动提交偏移量，或者可以选择通过调用其中一个提交API(例如commitSync和commitAsync)来手动控制该提交位置。

### Consumer Groups 和Topic Subscriptions

Kafka）使用消费者组的概念来允许一系列进程划分消费和处理记录的工作。这些进程可以运行在同一台机器上也可以分布在不同的机器上以达到负载均衡和高可用的目地。所有实例共享同一个group.id的消费者都属于同一个消费者组。

每一个组内的消费者可以通过subscribe apis动态的设置要消费的topics。卡夫卡将把订阅主题中的每条消息传递给每个消费者组中的一个进程。通过这种方式可以实现组内消费者的负载均衡，每个消息被分配给组内的一个消费者。因此，如果一个主题有4个分区，一个消费者组有2个消费者，每个消费者将会从两个分区中消费消息。

消费者组的成员资格是动态维护的，如果一个消费者失败，分配给它的分区会被重新分配给组内的另一个消费者。类似地，如果一个新的消费者加入到这个组，部分分区会从已经存在的消费者中移动到这个新的消费者。这就是对组的重负载均衡。group重负载均衡对新加入的分区或者一个新的topic生成时。group会通过定时的自动刷新元数据自动检测新的分区，并分配它们到组内的成员消费者。

从主观上你可以想象一个消费者组为一个由多个物理消费者组成的逻辑的消费者。作为一个多订购的系统，kafka支持任意多个消费者组对一个指定的topic进行消费，不复制数据(多余的消费者创建是很便宜的，所以不妨多创建几个消费者)

这是消息传递系统中常见功能的轻微泛化。为了获得类似于传统消息传递系统中的队列的语义，所有进程将成为单个消费者组的一部分，因此记录传递将在队列中平衡，如队列。

与传统的消息传递系统不同，您可以拥有多个这样的组。为了在传统的消息传递系统中获得与pub-sub类似的语义，每个进程都有自己的消费者组，所以每个进程都会订阅发布到该主题的所有记录。

此外，当组重新分配自动发生时，消费者可以通过一个ConsumerRebalanceListener通知消费者，这允许他们完成必要的应用程序级逻辑，例如状态清理，手工偏移提交等。请参阅在Kafka外部存储偏移以获取更多详细信息。

消费者也可以使用assign（Collection）手动分配特定分区（类似于较旧的“简单”消费者）。 在这种情况下，动态分区分配和用户组协调将被禁用。

### 检测消费者异常

在订阅了一组主题之后，消费者将在调用poll（long）时自动加入组。poll api旨在确保消费者的生存。 只要您继续调用poll，消费者就会留在群组中，并继续从分配的分区接收消息。 底层实现中，消费者定期发送心跳到服务器。 如果消费者在session.timeout.ms期间内崩溃或无法发送心跳信号，那么消费者将被视为死亡，并且其分区将被重新分配。

消费者也有可能会遇到“活锁”的情况，即继续发送心跳，但没有任何进展。 为了防止消费者在这种情况下无限期地保持其分区，我们使用max.poll.interval.ms设置提供活跃检测机制。 基本上，如果您至少不像配置的最大时间间隔那么频繁地调用轮询，则客户端将主动离开组，以便另一个消费者可以接管其分区。 发生这种情况时，您可能会看到一个偏移提交失败（如调用commitSync（）所抛出的CommitFailedException所示）。 这是一个安全机制，保证只有组的活跃成员能够提交偏移量。 所以留在这个小组里，你必须继续poll。

用户提供了两个配置设置来控制轮询循环的行为：

1. max.poll.interval.ms：通过增加期望的poll之间的时间间隔，您可以给消费者更多的时间来处理从pol（long）返回的一批记录。 缺点是增加此值可能会延迟群组重新平衡，因为消费者只会在调用内部加入时重新平衡。 您可以使用此设置限制完成重新平衡的时间，但是如果消费者实际上不能经常调用轮询，则风险是处理会更慢。
2. max.poll.records：使用此设置可将单次调用返回的记录总数限制为轮询。 这可以更容易地预测每个轮询间隔内必须处理的最大值。 通过调整这个值，您可能会减少轮询间隔，这将减少组重新平衡的影响。

对于消息处理时间不可预知地变化的用例，这些选项都不足以满足要求。 处理这些情况的推荐方法是将消息处理移动到另一个线程，这使得消费者可以在处理器仍在工作时继续调用轮询。 必须小心确保提交的偏移量不超过实际位置。 通常情况下，只有在线程完成处理之后（取决于所需的传递语义），您必须禁用自动提交并手动提交记录的已处理偏移量。 还要注意，你需要暂停分区，以便在线程完成处理之前返回的内容之前，不会收到来自轮询的新记录。

### 有用的示例

消费者API提供灵活性以涵盖各种消费用例。 这里有一些示例来演示如何使用它们。

#### 自动偏移量提交

这个例子演示了Kafka的依赖于自动偏移提交的消费者API的简单用法。

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意味着偏移是通过config auto.commit.interval.ms控制的频率自动提交的。

在这个例子中，消费者正在订购foo和bar这样的主题，作为一个名为test的消费者组的一部分，如group.id所配置的。

反序列化设置指定如何将字节转换为对象。 例如，通过指定字符串反序列化器，我们说我们的记录的键和值只是简单的字符串。

#### 手动控制偏移量

用户不需要依靠消费者定期提交消耗的偏移量，而是可以控制何时将消耗记录为消耗，从而提交偏移量。 当消息的消耗与一些处理逻辑相结合时，这是非常有用的，因此消息在完成处理之前不应被视为消耗。

Properties props = new Properties();

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

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

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

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，我们将消耗一批记录并在内存中进行批处理。当我们有足够的记录批处理时，我们将它们插入到数据库中。如果我们像前面的例子那样允许偏移量自动提交，那么记录在被轮询返回给用户之后将被视为消耗。那么在批量处理记录之后，我们的流程就有可能失败，但在它们被插入数据库之前就会失败。

为了避免这种情况，只有在相应的记录被插入数据库之后，我们才会手动提交偏移量。这使我们能够精确地控制记录被考虑消耗的时间。这提出了相反的可能性：在插入到数据库之后但是在提交之前（尽管这可能仅仅是几毫秒，这是可能的），该过程可能在间隔期间失败。在这种情况下，接管消费的消费者将消耗上次提交的偏移，并重复插入最后一批数据。以这种方式使用Kafka提供了通常所说的“至少一次”交付保证，因为每个记录可能会被交付一次，但在失败的情况下可能被复制。

注意：使用自动偏移提交也可以给你“至少一次”投递，但是要求你必须消费在下一个poll之前消费poll方法返回的每一条记录。 如果您未能完成其中任何一项操作，则提交的偏移量可能会超过消耗的位置，从而导致记录丢失。 使用手动偏移控制的优点是，您可以直接控制记录被视为“消耗”的时间。

以上示例使用commitSync将所有收到的记录标记为已提交。 在某些情况下，您可能希望通过显式指定偏移量来更好地控制哪些记录已被提交。 在下面的例子中，我们完成处理每个分区中的记录后提交偏移量。

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：提交的偏移量应该始终是应用程序将读取的下一条消息的偏移量。 因此，在调用commitSync（偏移量）时，应该在最后处理的消息的偏移量上加1。

#### 手动分配分区

在前面的示例中，我们订阅了我们感兴趣的主题，并让Kafka根据组中的活动消费者动态分配这些主题的分区的公平份额。 但是，在某些情况下，您可能需要更好地控制分配的特定分区。 例如：

1. 如果进程正在维护与该分区相关联的某种本地状态（如本地磁盘上的键值存储），那么它应该只获取它在磁盘上维护的分区的记录。
2. 如果进程本身是高度可用的，并且如果失败（可能使用诸如YARN，Mesos或AWS工具的集群管理框架，或者作为流处理框架的一部分）将被重新启动。 在这种情况下，不需要Kafka检测到故障并重新分配分区，因为消费过程将在另一台计算机上重新启动。

要使用这种模式，您不必使用订阅来订阅主题，只需使用您要使用的完整分区列表来调用assign（Collection）。

String topic = "foo";

TopicPartition partition0 = new TopicPartition(topic, 0);

TopicPartition partition1 = new TopicPartition(topic, 1);

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

一旦分配完毕，就可以在循环中调用轮询，就像在前面的例子中消耗记录一样。 用户指定的组仍然用于提交偏移量，但是现在该分区组只会随着另一个调用分配而改变。 手动分区分配不使用组协调，因此消费者故障不会导致分配的分区重新平衡。 即使每个消费者与另一个消费者共享一个组，也是独立行事的。 为避免冲突提交冲突，通常应确保groupId对于每个消费者实例都是唯一的。

请注意，无法通过主题订阅（即使用订阅）将手动分区分配（即使用分配）与动态分区分配相混合。

#### 外部存储偏移量

消费者应用程序不需要使用Kafka的内置偏移量存储，它可以将偏移量存储在自己选择的存储中。 这样做的主要场景是允许应用程序将偏移量和消耗结果存储在同一个系统中，使得结果和偏移量都以原子方式存储。 这并不总是可能的，但是如果是这样的话，它将使得消费完全原子化，并给予“恰好一次”的语义，这比Kafka的偏移提交功能获得的默认“至少一次”语义更强。

这里有几个这种用法的例子：

1. 如果消费的结果存储在关系数据库中，那么在数据库中存储偏移量也可以允许在单个事务中提交结果和偏移量。 因此，事务将成功，偏移将根据所消耗的内容或结果将不会被更新，并且偏移将不会被更新。
2. 如果结果存储在本地存储中，那么也可以在那里存储偏移量。 例如，搜索索引可以通过订阅特定分区并将偏移量和索引数据一起存储来构建。 如果这是以原子的方式完成的，那么即使发生崩溃，导致不同步的数据丢失，也会经常出现这种情况，不管是否存储了相应的偏移量。 这意味着在这种情况下，返回丢失最近更新的索引过程只是从它所具有的索引中恢复索引，以确保不会丢失更新。

每个记录都带有自己的偏移量，所以要管理自己的偏移量，只需执行以下操作：

1. 配置 enable.auto.commit=false
2. 使用每个ConsumerRecord提供的偏移量来保存您的位置。
3. 在重新启动时使用seek（TopicPartition，long）恢复消费者的位置。

当分区分配也是手动完成时，这种用法是最简单的（这在上面描述的搜索索引用例中很可能）。 如果分区分配自动完成，需要特别注意处理分区分配改变的情况。 这可以通过在调用subscribe（Collection，ConsumerRebalanceListener）和订阅（Pattern，ConsumerRebalanceListener）中提供一个ConsumerRebalanceListener实例来完成。 例如，当从消费者获取分区时，消费者将通过实现ConsumerRebalanceListener.onPartitionsRevoked（Collection）来提交这些分区的偏移量。 将分区分配给使用者时，使用者需要查找这些新分区的偏移量，并通过实现ConsumerRebalanceListener.onPartitionsAssigned（Collection）正确地将使用者初始化为该位置。

ConsumerRebalanceListener的另一个常见用途是刷新应用程序为其他位置移动的分区而保留的任何缓存。

#### 控制消费者的位置

在大多数使用情况下，消费者会简单地从头到尾地使用记录，定期（自动或手动）提交其位置。 不过kafka允许消费者手动控制其位置，随意在分区中向前或向后移动。 这意味着消费者可以重新使用旧的记录，或跳过最近的记录，而不需要实际使用中间记录。

有几种情况可以手动控制消费者的位置。

1. 一种情况是对时间敏感的记录处理，对于落后的消费者来说可能是有意义的，而不是试图赶上处理所有记录，而只是跳到最近的记录。
2. 另一个用例是用于维护本地状态的系统，如前一节所述。 在这样的系统中，消费者希望在启动时将其位置初始化为本地存储中包含的任何位置。 同样，如果本地状态被破坏（比如说因为磁盘丢失），通过重新使用所有数据并重新创建状态（假设Kafka保留足够的历史记录），可以在新机器上重新创建状态。

Kafka允许使用seek（TopicPartition，long）指定位置来指定新的位置。 寻求最早和最近的服务器维护偏移的特殊方法也可用（分别为seekToBeginning（Collection）和seekToEnd（Collection））。

#### 消费流控制

如果消费者被分配了多个分区来获取数据，它将尝试同时消耗所有的分区，从而有效地赋予这些分区相同的消耗优先级。 但是，在某些情况下，消费者可能首先要关注从全部分配的分区的某个子集中提取数据，并且只有在这些分区中的数据很少或不需要消耗时才开始提取其他分区。

其中一种情况是流处理，处理器从两个主题获取并在这两个流上执行连接。 当其中一个主题滞后于另一个主题时，处理器想暂停从前面的主题中获取，以便赶上滞后的流。 另外一个例子是，当消费者开始有很多历史数据赶上的时候，应用程序通常希望获得关于某些主题的最新数据，然后再考虑其他主题。

Kafka支持通过使用暂停（Collection）和恢复（Collection）来暂停指定分配的分区上的消耗，并在未来的poll（long）调用中分别恢复指定暂停分区上的消耗，从而动态控制消耗流量。

#### 读取事务型消息

应用程序可以以原子方式写入多个主题和分区。 为了使这个工作，消费者从这些分区读取应该被配置为只读取提交的数据。 这可以通过在消费者的配置中设置isolation.level = read\_committed来实现。

在read\_committed模式下，使用者将只读取已成功提交的事务性消息。 它将继续像以前一样读取非事务性消息。 在read\_committed模式下没有客户端缓冲。 相反，read\_committed使用者的分区的结束偏移量将是属于打开的事务的分区中的第一个消息的偏移量。 这个偏移被称为“最后稳定偏移”（LSO）。

一个read\_committed的消费者只能读取LSO并过滤掉所有已经中止的事务消息。 LSO还影响read\_committed使用者的seekToEnd（Collection）和endOffsets（Collection）的行为，详细信息在每个方法的文档中。 最后，读取滞后度量也被调整为相对于读取委托消费者的LSO。

具有事务消息的分区将包括指示事务结果的提交或中止标记。 有标记不返回到应用程序，但日志中有偏移量。 因此，使用事务性消息从主题读取的应用程序将会看到消耗的偏移量中的差距。 这些丢失的消息将是事务标记，并在两个隔离级别中被消费者过滤掉。 此外，使用read\_committed的使用者的应用程序也可能会由于中止事务而看到间隙，因为这些消息不会被消费者返回，而是具有有效的偏移量。

#### 多线程处理

卡夫卡消费者不是线程安全的。 所有网络I / O都发生在call()调用时的应用程序的线程中。 用户有责任确保多线程访问的正确同步。 未同步的访问将导致ConcurrentModificationException。

这个规则唯一的例外是wakeup（），它可以安全地从外部线程使用来中断一个活动的操作。 在这种情况下，一个WakeupException将从该线程阻塞的操作中抛出。 这可以用来关闭另一个线程的消费者。 以下片段显示了典型模式：

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();

}

}

然后在一个单独的线程中，消费者可以通过设置关闭标志并唤醒消费者来关闭。

closed.set(true);

consumer.wakeup();

注意，尽管可以使用线程中断而不是wakeup（）来中止阻塞操作（在这种情况下会引发InterruptException），但我们不鼓励使用它们，因为它们可能导致消费者的中断关闭。 中断主要支持那些不能使用wakeup（）的情况。 当消费者线程由不知道Kafka客户端的代码管理时。

我们有意避免实现特定的线程模型进行处理。 这为实现记录的多线程处理留下了几个选项。

* 一个消费者一个线程

一个简单的选择是给每个线程自己的消费者实例。 以下是这种方法的优缺点：

优点:

1. 这是最容易实现的
2. 它通常是最快的，因为不需要线程间的协调
3. 它使得每个分区的有序处理非常容易实现（每个线程只是按照接收到的顺序处理消息）。

缺点：

1. 更多的消费者意味着更多的TCP连接到集群（每个线程一个）。 通常Kafka非常有效地处理连接，所以这通常是一个小的成本。
2. 多个消费者意味着更多的请求被发送到服务器，稍微少一点的数据可能会导致I / O吞吐量下降。
3. 所有进程中的总线程数将受到分区总数的限制。

* 将消费和处理解耦

另一种方法是让一个或多个消费者线程执行所有数据消费，并将ConsumerRecords实例移交给实际处理记录处理的处理器线程池所消耗的阻塞队列。 这个选项同样有利有弊：

优点：

1. 该选项允许独立扩展消费者和处理器的数量。 这样就可以让单个消费者提供多个处理器线程，避免了对分区的任何限制。

缺点：

1. 保证处理器之间的顺序需要特别小心，因为线程将独立执行，因为线程执行时机的运行，较早的数据块可能实际上在稍后的数据块之后被处理。 对于没有订购要求的处理来说，这不是问题。
2. 手动提交位置变得更加困难，因为它要求所有线程协调，以确保对该分区的处理已完成。

这种方法有许多可能的变化。 例如，每个处理器线程可以有自己的队列，消费者线程可以使用TopicPartition散列到这些队列中，以确保按顺序使用并简化提交。

## Streams API

允许对来自一个或多个输入主题的输入执行连续计算，并将输出发送到零个，一个或多个输出主题。

可以通过使用拓扑来定义处理器的DAG拓扑或者通过使用提供高级DSL来定义变换的StreamsBuilder来指定计算逻辑。

一个KafkaStreams实例可以包含配置中为处理工作指定的一个或多个线程。

KafkaStreams实例可以与具有相同应用程序标识的任何其他实例（无论是在同一进程中，在本机上的其他进程还是在远程计算机上）协调为单个（可能是分布式）流处理应用程序。 这些实例会根据输入主题分区的分配划分工作，以便所有分区都被占用。 如果实例添加或失败，则所有（剩余的）实例将重新平衡分区之间的分配，以平衡处理负载并确保处理所有输入主题分区。

内部KafkaStreams实例包含一个普通的KafkaProducer和KafkaConsumer实例，用于读取输入和写入输出。

一个简单的例子可能是这样的：

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.DEFAULT\_KEY\_SERDE\_CLASS\_CONFIG, Serdes.String().getClass());

props.put(StreamsConfig.DEFAULT\_VALUE\_SERDE\_CLASS\_CONFIG, Serdes.String().getClass());

StreamsConfig config = new StreamsConfig(props);

StreamsBuilder builder = new StreamsBuilder();

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

KafkaStreams streams = new KafkaStreams(builder.build(), config);

streams.start();

## Connect API

# 持久化

卡夫卡在很大程度上依赖文件系统来存储和缓存消息。 人们普遍认为，“磁盘速度慢”使人们怀疑一个持久的结构可以提供有竞争力的表现。 实际上，磁盘比人们所期望的要慢得多，速度也要快得多，这取决于它们的使用方式。 一个设计合理的磁盘结构通常可以和网络一样快。

关于磁盘性能的关键事实是，硬盘的吞吐量与过去十年的磁盘搜索的延迟不同。 因此，使用6个7200rpm SATA RAID-5阵列的JBOD配置上的线性写入的性能约为600MB /秒，但随机写入的性能仅约为100k / sec，相差6000X以上。 这些线性读取和写入是所有使用模式中最可预测的，并且由操作系统进行了大量优化。 现代操作系统提供预读和后写技术，以大块为单位预取数据，并将较小的逻辑写入大型物理写入。 关于这个问题的进一步讨论可以在这个ACM队列文章中找到。 他们实际上发现顺序磁盘访问在某些情况下可能比随机存储访问更快！

为了弥补这种性能差异，现代操作系统在使用主内存进行磁盘缓存方面变得越来越积极。 现代操作系统将愉快地将所有空闲内存转移到磁盘高速缓存，并在回收内存时性能降低。 所有的磁盘读写操作都将通过这个统一的缓存。 如果不使用直接I / O，此功能将无法轻松关闭，因此即使进程维护数据的进程内缓存，该数据也可能会在操作系统页面缓存中复制，从而有效地存储所有内容两次。

而且，我们正在建立在JVM的基础之上，任何花时间使用Java内存的人都知道两件事情：

1. 对象的内存开销很高，通常会使所存储数据的大小加倍（或更糟糕）。
2. 随着堆内数据的增加，Java垃圾收集变得越来越复杂和缓慢。

由于使用文件系统和依赖页面缓存的这些因素优于维护内存缓存或其他结构，所以我们至少将可用缓存翻了一番，方法是自动访问所有可用内存，并且可能再次通过存储一个压缩包字节结构而不是单个对象。这样做会导致在32GB的机器上高达28-30GB的缓存，而不受GC惩罚。此外，即使服务重新启动，该缓存也会保持温暖，而进程内缓存将需要在内存中重建（对于10GB的缓存可能需要10分钟），否则将需要以完全冷的缓存开始（这可能意味着糟糕的初始表现）。这也极大地简化了代码，因为所有保持高速缓存和文件系统之间的一致性的逻辑现在都在OS中，这比一次性的进程内尝试更高效和更正确地执行。如果您的磁盘使用情况支持线性读取，则预读使用每个磁盘读取的有用数据有效地预先填充此缓存。

这表明一个非常简单的设计：当我们在空间不足的时候，不是尽可能地保持内存中的空间，而是把所有内容全部冲洗到文件系统中，我们将其反转。 所有数据立即写入文件系统的持久日志，而不必冲刷到磁盘。 实际上，这只是意味着它被转移到内核的页面缓存中。

这种以页面缓存为中心的设计风格在这里的一个关于Varnish设计的文章中描述（伴随着一种健康的傲慢态度）。

# 配置

## Broker配置：

1. zookeeper.connect
2. advertised.listeners

当Kafka broker启动时，它会在ZK上注册自己的IP和端口号，客户端就通过这个IP和端口号来连接。在AWS这种IaaS环境下，由于java.net.InetAddress.getCanonicalHostName调用拿到的HostName是类似ip-172-31-10-199这样的只有内网才能访问到的主机名，所以默认注册到ZK上的IP是内网才能访问的内网IP。此时就需要显示指定 advertised.host.name, advertised.listeners参数，让注册到ZK上的IP是外网IP。例如对于 59.64.11.22 IP对应的broker需要在 server.properties 配置文件里增加如下如下配置：advertised.listeners=PLAINTEXT://59.64.11.22:9092

auto.create.topics.enable：默认是true。不存在topic的时候是否在server端自动创建。

auto.leader.rebalance.enable：默认是true。是否启动 partition-rebalance 线程。

background.threads：默认是10.用于各种后台处理任务（如文件删除）的线程数。

broker.id：默认-1。此服务器的代理ID。 如果未设置，则会生成唯一的代理标识。为避免zookeeper生成的代理标识与用户配置的代理标识之间的冲突，生成的代理标识从reserved.broker.max.id + 1开始。

compression.type：默认为producer。producer用于压缩数据的压缩类型。默认是无压缩。正确的选项值是none、gzip、snappy。压缩最好用于批量处理，批量处理消息越多，压缩性能越好。

delete.topic.enable：默认是true。启用删除主题。 如果此配置已关闭，则通过管理工具删除主题将不起作用。

leader.imbalance.check.interval.seconds：默认300s。检测leader不平衡的时间间隔。

leader.imbalance.per.broker.percentage： 默认值：10。每个broker允许的不平衡的leader的百分比。如果每个broker超过了这个百分比，复制控制器会重新平衡leadership。

listeners

监听器列表 - 逗号分隔的我们将监听的URI列表和监听器名称。 如果侦听器名称不是安全协议，则还必须设置listener.security.protocol.map。 指定主机名为0.0.0.0以绑定到所有接口。 保留主机名为空以绑定到默认界面。 合法侦听器列表的示例：PLAINTEXT：// myhost：9092，SSL：//：9091 CLIENT：//0.0.0.0：9092，REPLICATION：// localhost：9093

如果不配置，默认取PLAINTEXT://your.host.name:9092。如果没有设置advertised.listeners，就使用 listeners

log.dir：保存日志数据的目录（对log.dirs属性的补充）

log.dirs：日志数据保存的目录。 如果未设置，则使用log.dir中的值

log.flush.interval.messages：在将消息刷新到磁盘之前，在日志分区上累积的消息数量。

log.flush.interval.ms：任何主题中的消息在刷新到磁盘之前都保存在内存中的最长时间（以毫秒为单位）。 如果未设置，则使用log.flush.scheduler.interval.ms中的值

log.flush.offset.checkpoint.interval.ms：默认值60000。更新作为日志恢复点的最后一次刷新的持久记录的频率

log.flush.scheduler.interval.ms：日志刷新器检查是否需要将任何日志刷新到磁盘的频率（以毫秒为单位）

log.flush.start.offset.checkpoint.interval.ms：默认值60000。更新记录起始偏移量的持续记录的频率

log.retention.bytes：默认值-1。删除之前日志的最大大小

log.retention.hours：默认值168。保留日志文件的小时数（以小时为单位），大写为log.retention.minutes属性：

默认null。在删除日志文件之前保留日志的分钟数（以分钟为单位），次要log.retention.ms属性。 如果未设置，则使用log.retention.hours中的值。

log.retention.ms：默认null。保留日志文件的毫秒数（以毫秒为单位），如果未设置，则使用log.retention.minutes中的值

log.roll.hours：默认值：24 \* 7

这个设置会强制Kafka去roll一个新的log segment文件，即使当前使用的segment文件的大小还没有超过。次于log.roll.ms属性

log.roll.jitter.hours:从logRollTimeMillis减去的最大抖动（以小时为单位），次要log.roll.jitter.ms属性。

log.roll.jitter.ms：从logRollTimeMillis中减去的最大抖动（以毫秒为单位）。 如果未设置，则使用log.roll.jitter.hours中的值。

log.roll.ms：默认为null。这个设置会强制Kafka去roll一个新的log segment文件，即使当前使用的segment文件的大小还没有超过。

log.segment.bytes：单个日志文件的最大大小

log.segment.delete.delay.ms：从文件系统中删除文件之前等待的时间。

message.max.bytes：

在最新的消息格式版本中，记录总是分组分批以提高效率。 在以前的消息格式版本中，未压缩的记录不会分组到批次中，并且此限制仅适用于该情况下的单个记录。

可以使用主题级别max.message.bytes config来设置每个主题。

min.insync.replicas：默认为1。

当生产者将ack设置为“全部”（或“-1”）时，min.insync.replicas指定必须确认写入被认为成功的最小副本数。 如果这个最小值不能满足，那么生产者将会引发一个异常（NotEnoughReplicas或NotEnoughReplicasAfterAppend）。

当一起使用时，min.insync.replicas和acks允许您执行更大的耐久性保证。 一个典型的情况是创建一个复制因子为3的主题，将min.insync.replicas设置为2，并使用“全部”选项来产生。 这将确保生产者如果大多数副本没有收到写入引发异常。

num.io.threads：默认值为8

服务器用于处理请求的线程数，可能包括磁盘I / O

num.network.threads：默认值3

服务器用于接收来自网络的请求并向网络发送响应的线程数

num.recovery.threads.per.data.dir：默认值1

每个数据目录的线程数，用于启动时的日志恢复和关闭时的刷新

num.replica.fetchers：默认值1

用于从源broker复制消息的读取器线程数。 增加此值可以提高跟随者代理中的I / O并行度。

offset.metadata.max.bytes：默认值4096

与偏移量提交相关联的元数据条目的最大大小

offsets.commit.required.acks：默认值-1

提交之前所需的确认可以被接受。 一般来说，默认值（-1）不应该被覆盖

offsets.commit.timeout.ms：默认值5000.

偏移提交将被延迟，直到偏移量主题的所有副本都收到提交或达到此超时。 这与生产者请求超时类似。

offsets.load.buffer.size：默认值5242880

将偏移量加载到缓存中时从批量偏移段读取的批量大小。

offsets.retention.check.interval.ms：默认600秒

检查失效偏移的频率

offsets.retention.minutes：默认值1440

比这个保留期更早的偏置将被丢弃

offsets.topic.compression.codec：默认值0

用于偏移主题的压缩编解码器 - 压缩可用于实现“原子”提交

offsets.topic.num.partitions：默认值50

偏移提交主题的分区数量（部署后不应更改）

offsets.topic.replication.factor：默认值3

偏移量主题的复制因子（设置得更高以确保可用性）。 内部主题创建将失败，直到群集大小满足此复制因子要求。

offsets.topic.segment.bytes：默认值104857600

为了便于更快的日志压缩和缓存加载，偏移量的主题段字节应保持相对较小

queued.max.requests：默认值500

阻塞网络线程之前允许排队的请求数量

replica.fetch.min.bytes：默认值1

每个提取响应的预期最小字节数。 如果没有足够的字节，请等待replicaMaxWaitTimeMs

replica.fetch.wait.max.ms：默认值50

跟随者副本发出的每个获取请求的最长等待时间。 此值始终应始终小于replica.lag.time.max.ms，以防止针对低吞吐量主题频繁收缩ISR

replica.high.watermark.checkpoint.interval.ms：默认值5000

高水印被保存到磁盘的频率

replica.lag.time.max.ms：默认值10000

如果一个追随者至少没有发送任何提取请求，或者至少在这段时间内没有消耗掉领导者的日志偏移量，那么领导者会从isr中移除它

replica.socket.receive.buffer.bytes：默认值65536

套接字接收缓冲区的网络请求

replica.socket.timeout.ms：默认值30000

网络请求的套接字超时。 它的值应该至少是replica.fetch.wait.max.ms

request.timeout.ms：默认值30000

配置控制客户端等待请求响应的最长时间。 如果在超时过去之前未收到响应，则客户端将在必要时重新发送请求，或者如果重试耗尽，则请求失败。

socket.receive.buffer.bytes：默认值102400

套接字服务器套接字的SO\_RCVBUF缓冲区。 如果值为-1，则将使用操作系统默认值。

socket.request.max.bytes：默认值102400

套接字请求中的最大字节数

socket.send.buffer.bytes：默认值102400

套接字服务器套接字的SO\_SNDBUF缓冲区。 如果值为-1，则将使用操作系统默认值。

transaction.max.timeout.ms：默认值900000

交易的最大允许超时时间。 如果客户请求的交易时间超过这个时间，那么代理将在InitProducerIdRequest中返回一个错误。 这样可以防止客户过度超时，从而阻碍消费者阅读交易中包含的主题。

transaction.state.log.load.buffer.size：默认值5242880

将生产者ID和事务加载到缓存中时，从事务日志段读取的批量大小。

transaction.state.log.min.isr：默认值2

覆盖事务主题的min.insync.replicas配置。

transaction.state.log.num.partitions：默认值50

事务主题的分区数量（部署后不应更改）。

transaction.state.log.replication.factor：默认值3

交易主题的复制因子（设置更高以确保可用性）。 内部主题创建将失败，直到群集大小满足此复制因子要求。

transaction.state.log.segment.bytes：默认值104857600

事务主题段字节应保持相对较小，以便于更快的日志压缩和缓存负载

transactional.id.expiration.ms：默认值 604800000

事务协调器在未收到任何事务状态更新之前主动过期生产者的事务标识之前将等待的最长时间（以毫秒为单位）。

unclean.leader.election.enable：默认值false

指示是否启用不在ISR集合中的副本作为最后的选择，尽管这样做可能导致数据丢失

zookeeper.connection.timeout.ms：默认值null

客户端等待与zookeeper建立连接的最长时间。 如果未设置，则使用zookeeper.session.timeout.ms中的值

zookeeper.session.timeout.ms:默认值6000

Zookeeper会话超时时间

zookeeper.set.acl：默认值false

将客户端设置为使用安全ACL

broker.id.generation.enable:默认值true

在服务器上启用自动代理ID生成。 启用时，应该检查为reserved.broker.max.id配置的值。

broker.rack：默认值null

经纪人的机架。 这将在机架感知复制分配中用于容错。 例如：`RACK1`，`us-east-1d`

connections.max.idle.ms：默认值600000

空闲连接超时：服务器套接字处理器线程关闭闲置超过此的连接

controlled.shutdown.enable：默认值true

启用服务器的受控关闭

controlled.shutdown.max.retries：默认值3

受控关机可能由于多种原因而失败。 这决定了发生这种故障时的重试次数

controlled.shutdown.retry.backoff.ms：默认值5000

在每次重试之前，系统需要时间从导致先前故障的状态（控制器故障切换，副本延迟等）中恢复。 此配置确定重试之前等待的时间量。

controller.socket.timeout.ms：默认30000

控制器到代理通道的套接字超时

default.replication.factor：默认值1

自动创建主题的默认复制因子

delete.records.purgatory.purge.interval.requests：默认值1

删除记录请求purge的清除间隔（请求数）

fetch.purgatory.purge.interval.requests：默认值1000

提取请求purge的清除间隔（请求数）

group.initial.rebalance.delay.ms：默认值3000

在执行第一次再平衡之前，团队协调员将等待更多消费者加入新团队的时间。 延迟时间越长意味着重新平衡的可能性越小，但是直到处理开始的时间增加。

group.max.session.timeout.ms：默认值300000

注册用户的最大允许会话超时。 较长的超时使消费者有更多的时间来处理心跳之间的消息，代价是较长的时间来检测失败。

group.min.session.timeout.ms：默认值6000

注册用户允许的最小会话超时。 更短的超时导致以更频繁的消费者心跳为代价的更快速的故障检测，这可能压倒经纪人资源。

inter.broker.listener.name：默认值null

用于经纪人之间沟通的听众名称。 如果未设置，则侦听器名称由security.inter.broker.protocol定义。 同时设置此和security.inter.broker.protocol属性是错误的。

inter.broker.protocol.version：默认值1.0-IV0

指定将使用哪个版本的代理间协议。 在所有经纪人升级到新版本之后，这通常会受到冲击。 一些有效值的示例如下：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检查ApiVersion的完整列表。

log.cleaner.backoff.ms：默认值15000

没有日志清理的睡眠时间

log.cleaner.dedupe.buffer.size：默认值 134217728

用于所有清理程序线程的日志重复数据消除的总内存

log.cleaner.delete.retention.ms：默认值86400000

删除记录保留多久？

log.cleaner.enable：默认值true

启用日志清理器进程在服务器上运行。 应该启用，如果使用cleanup.policy = compact任何主题，包括内部偏移主题。 如果被禁用的话，这些话题将不会被压缩，并且会不断增长。

log.cleaner.io.buffer.load.factor：默认值0.9

日志清理器重复数据删除缓存负载因 完全重复数据删除缓冲区的百分比可以变成。 更高的值将允许一次清理更多的日志，但会导致更多的散列冲突

log.cleaner.io.buffer.size：默认值524288

用于日志清理程序I / O缓冲区的整个内存跨越所有清理程序线程

log.cleaner.io.max.bytes.per.second：默认值1.7976931348623157E308

日志清理器将受到限制，所以其读取和写入I / O的总和将平均小于此值

log.cleaner.min.cleanable.ratio：默认值0.5

日志中的脏日志与总日志的最小比率，以符合清理的条件

log.cleaner.min.compaction.lag.ms：默认值0

消息在日志中保持未压缩的最短时间。 仅适用于正在压缩的日志。

log.cleaner.threads：默认值1

用于日志清理的后台线程的数量

log.cleanup.policy：默认值delete

超出保留窗口的段的默认清理策略。 以逗号分隔的有效策略列表。 有效的政策是：“delete”和“compact”

log.index.interval.bytes：默认值4096

我们添加一个条目到偏移索引的时间间隔

log.index.size.max.bytes：默认值10485760

偏移索引的最大大小（以字节为单位）

log.message.format.version：默认值1.0-IV0

指定代理将用于将消息附加到日志的消息格式版本。 该值应该是有效的ApiVersion。 一些例子是：0.8.2，0.9.0.0，0.10.0，查看ApiVersion了解更多细节。 通过设置特定的消息格式版本，用户正在证明磁盘上的所有现有消息小于或等于指定的版本。 不正确地设置这个值将导致使用旧版本的用户中断，因为他们将接收到他们不理解的格式的消息。

log.message.timestamp.difference.max.ms：默认值9223372036854775807

代理收到消息时的时间戳和消息中指定的时间戳之间允许的最大差异。 如果log.message.timestamp.type = CreateTime，如果时间戳的差值超过此阈值，则会拒绝一条消息。 如果log.message.timestamp.type = LogAppendTime，则忽略此配置。允许的最大时间戳差异不应大于log.retention.ms，以避免不必要地频繁进行日志滚动。

log.message.timestamp.type：默认值CreateTime

定义消息中的时间戳是消息创建时间还是日志追加时间。 该值应该是“CreateTime”或“LogAppendTime”

log.preallocate：默认值false

创建新分段时应该预先分配文件吗？ 如果您在Windows上使用Kafka，则可能需要将其设置为true。

log.retention.check.interval.ms：默认值300000

日志清理程序检查是否有日志符合删除的频率（以毫秒为单位）

max.connections.per.ip：默认值2147483647

我们从每个IP地址允许的最大连接数

max.connections.per.ip.overrides：默认值””

每个IP或主机名将覆盖默认的最大连接数

num.partitions：默认值1

每个主题的默认日志分区数量

principal.builder.class：默认值null

实现KafkaPrincipalBuilder接口的类的完全限定名，用于构建在授权过程中使用的KafkaPrincipal对象。 此配置还支持以前用于通过SSL进行客户端身份验证的弃用的PrincipalBuilder接口。 如果未定义主体构建器，则默认行为取决于正在使用的安全协议。 对于SSL身份验证，如果提供了一个主体名称，则主体名称将是客户端证书的可分辨名称; 否则，如果客户端身份验证不是必需的，主体名称将是匿名的。 对于SASL认证，如果使用GSSAPI，则使用由sasl.kerberos.principal.to.local.rules定义的规则导出主体，而使用其他机制的SASL认证ID。 对于PLAINTEXT，身份将是匿名的

producer.purgatory.purge.interval.requests：默认值1000

生产者请求炼狱的清除间隔（请求数）

queued.max.request.bytes：默认值-1

在不再读取请求之前，允许排队的字节数

replica.fetch.backoff.ms：默认值1000

提取分区错误时发生的睡眠时间。

replica.fetch.max.bytes：默认值1048576

尝试为每个分区获取的消息的字节数。 这不是绝对最大值，如果第一个非空分区中的第一个记录批次大于此值，那么记录批次仍将被返回以确保可以进行。 代理接受的最大记录批量大小通过message.max.bytes（broker config）或max.message.bytes（topic config）进行定义。

replica.fetch.response.max.bytes：默认值10485760

整个提取响应的预期最大字节数。 记录是批量提取的，如果第一个记录批次在第一个非空分区的分区大于这个值，那么记录批处理仍然会被返回，以确保可以进行。 因此，这不是一个绝对的最大值。 代理接受的最大记录批量大小通过message.max.bytes（broker config）或max.message.bytes（topic config）进行定义。

reserved.broker.max.id：默认值1000

可以用于broker.id的最大数量

sasl.enabled.mechanisms：默认值GSSAPI

kafka服务器中启用的SASL机制列表。 该列表可能包含安全提供程序可用的任何机制。 只有GSSAPI默认启用。

sasl.kerberos.kinit.cmd:默认值/usr/bin/kinit

Kerberos kinit命令路径。

sasl.kerberos.min.time.before.relogin：默认值60000

登录线程在刷新尝试之间的休眠时间

sasl.kerberos.principal.to.local.rules：默认值DEFAULT

从主体名称到简称（通常是操作系统用户名）的映射规则列表。 规则按顺序进行评估，并使用匹配主体名称的第一条规则将其映射到短名称。 列表中后面的任何规则都会被忽略。 默认情况下，{username} / {hostname} @ {REALM}形式的主体名称映射到{username}。 有关格式的更多细节，请参阅安全授权和acls。 请注意，如果由principal.builder.class配置提供了KafkaPrincipalBuilder的扩展，则会忽略此配置。

sasl.kerberos.service.name：默认值null

Kafka运行的Kerberos主体名称。 这可以在Kafka的JAAS配置或Kafka的配置中定义。

sasl.kerberos.ticket.renew.jitter：默认值0.05

随机抖动增加到更新时间的百分比。

sasl.kerberos.ticket.renew.window.factor：默认值0.8

登录线程将休眠，直到已经到达从上次刷新到票证到期的指定窗口时间因子，届时它将尝试更新票证。

sasl.mechanism.inter.broker.protocol：默认值GSSAPI

用于经纪人间通信的SASL机制。 默认是GSSAPI。

security.inter.broker.protocol：默认值PLAINTEXT

用于在代理之间进行通信的安全协议。 有效值为：PLAINTEXT，SSL，SASL\_PLAINTEXT，SASL\_SSL。 同时设置此属性和inter.broker.listener.name属性是错误的。

ssl.cipher.suites：默认值null

密码套件列表。 这是用于使用TLS或SSL网络协议来协商网络连接的安全设置的认证，加密，MAC和密钥交换算法的命名组合。 默认情况下，所有可用的密码套件都受支持。ssl.client.auth：默认值none

配置kafka代理来请求客户端认证。 以下设置是常见的：

ssl.client.auth =必需如果设置为必需的客户端身份验证是必需的。

ssl.client.auth = requested这意味着客户端身份验证是可选的。 不像请求，如果这个选项被设置，客户端可以选择不提供关于它自己的认证信息

ssl.client.auth = none这意味着不需要客户端身份验证。

ssl.enabled.protocols：默认值TLSv1.2,TLSv1.1,TLSv1

启用了SSL连接的协议列表。

ssl.key.password：默认值null

密钥存储文件中的私钥密码。 这对于客户端是可选的。

ssl.keymanager.algorithm：默认值SunX509

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。

ssl.keystore.location：默认值null

密钥存储文件的位置。 这对客户端是可选的，可以用于客户端的双向认证。

ssl.keystore.password：默认值null

密钥存储文件的存储密码。 这对客户端是可选的，只有在配置了ssl.keystore.location时才需要。

ssl.keystore.type:默认值JKS

密钥存储文件的文件格式。 这对于客户端是可选的。

ssl.protocol：默认值TLS

用于生成SSLContext的SSL协议。 默认设置是TLS，在大多数情况下这是很好的。 最近的JVM中允许的值是TLS，TLSv1.1和TLSv1.2。 较早的JVM中可能支持SSL，SSLv2和SSLv3，但由于已知的安全漏洞而不鼓励使用SSL

ssl.provider：默认值null

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。

ssl.trustmanager.algorithm：默认值PKIX

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。

ssl.truststore.location：默认值null

信任存储文件的位置。

ssl.truststore.password：默认值null

信任存储文件的密码。 如果未设置密码，则仍然可以访问信任库，但完整性检查被禁用。

ssl.truststore.type：默认值JKS

信任存储文件的文件格式。

alter.config.policy.class.name：默认值null

应该用于验证的alter configs策略类。 该类应该实现org.apache.kafka.server.policy.AlterConfigPolicy接口。

authorizer.class.name：默认值””

应该用于授权的授权者类

create.topic.policy.class.name:默认值null

应该用于验证的创建主题策略类。 该类应该实现org.apache.kafka.server.policy.CreateTopicPolicy接口。

listener.security.protocol.map：默认值PLAINTEXT:PLAINTEXT,SSL:SSL,SASL\_PLAINTEXT:SASL\_PLAINTEXT,SASL\_SSL:SASL\_SSL

侦听器名称和安全协议之间的映射。 必须定义相同的安全协议才能在多个端口或IP中使用。 例如，即使两个都需要SSL，内部和外部流量也可以分开。 具体来说，用户可以定义名字为INTERNAL和EXTERNAL的监听器，这个属性为：`INTERNAL：SSL，EXTERNAL：SSL`。 如图所示，键和值由冒号分隔，映射条目以逗号分隔。 每个侦听器名称只能在地图中出现一次。 通过向配置名称添加规范化前缀（侦听器名称小写），可以为每个侦听器配置不同的安全性（SSL和SASL）设置。 例如，要为INTERNAL侦听器设置不同的密钥库，将会设置名称为“listener.name.internal.ssl.keystore.location”的配置。 如果侦听器名称的配置没有设置，配置将回退到通用配置（即`ssl.keystore.location`）。

metric.reporters：默认值“”

用作度量记录的类的列表。 实现org.apache.kafka.common.metrics.MetricsReporter接口允许插入将被通知新的度量标准创建的类。 JmxReporter始终包含在注册JMX统计信息中。

metrics.num.samples：默认值2

维持用于计算度量的样本数量。

metrics.recording.level：默认值INFO

指标的最高记录级别。

metrics.sample.window.ms：默认值30000

计算指标样本的时间窗口。

quota.window.num:默认值11

为客户配额保留的内存样本数

quota.window.size.seconds：默认值1

客户配额的每个样本的时间跨度

replication.quota.window.num：默认值11

要保留在内存中用于复制配额的样本数

replication.quota.window.size.seconds：默认值1

每个样本复制配额的时间跨度

ssl.endpoint.identification.algorithm：默认值null

使用服务器证书验证服务器主机名的端点识别算法

ssl.secure.random.implementation：默认值null

用于SSL加密操作的SecureRandom PRNG实现。

transaction.abort.timed.out.transaction.cleanup.interval.ms：默认值60000

回滚已超时的事务的时间间隔

transaction.remove.expired.transaction.cleanup.interval.ms：默认值3600000

由于事务性.id.expiration.ms传递而移除已经过期的事务的时间间隔

zookeeper.sync.time.ms：默认值2000

ZK追随者可以在ZK领导者后面走多远