# 目 录

[1. Kafka简介 1](#_Toc114497511)

[1.1 消息和批次 1](#_Toc114497512)

[1.2 主题和分区 1](#_Toc114497513)

[1.3 生产者和消费者 2](#_Toc114497514)

[1.4 broker和集群 3](#_Toc114497515)

[1.5 消息保留 4](#_Toc114497516)

[2. Kafka的安装部署 4](#_Toc114497517)

[2.1安装Java环境 4](#_Toc114497518)

[2.2安装Zookeeper 4](#_Toc114497519)

[2.3安装Kafka Broker 5](#_Toc114497520)

[2.4 Kafka Broker配置 6](#_Toc114497521)

[3. Kafka生产者 9](#_Toc114497524)

[3.1生产者概览 9](#_Toc114497525)

[3.2创建Kafka生产者 10](#_Toc114497526)

[3.3发送消息到Kafka 11](#_Toc114497527)

[3.4生产者配置 12](#_Toc114497530)

[4. Kafka消费者 14](#_Toc114497531)

[4.1消费者概览 14](#_Toc114497532)

[4.2创建Kafka消费者 17](#_Toc114497535)

[4.3订阅主题 17](#_Toc114497536)

[4.4轮询 18](#_Toc114497537)

[4.5消费者配置 19](#_Toc114497538)

[4.6提交和偏移量 20](#_Toc114497539)

[5. Kafka原理 24](#_Toc114497545)

[5.1集群成员关系 24](#_Toc114497546)

[5.2控制器 25](#_Toc114497547)

[5.3复制 25](#_Toc114497548)

[5.4处理请求 26](#_Toc114497549)

[5.5物理存储 28](#_Toc114497553)

[6. Kafka常用命令 30](#_Toc114497558)

[6.1启动Kafka 30](#_Toc114497559)

[6.2主题操作 30](#_Toc114497562)

[6.3消费者群组 31](#_Toc114497567)

[6.4生产和消费 31](#_Toc114497570)

# Kafka简介

Kafka是一款基于发布与订阅模式的消息系统（消息中间件），一般被称为“分布式提交日志”或“分布式流平台”。Kafka的数据按照一定顺序持久化保存，可以按需读取，并且数据分布在整个系统里，具备数据故障保护和性能伸缩能力。

## 1.1 消息和批次

消息是指Kafka的数据单元，类似数据库中的一个“数据行”或一条“记录”。消息由字节数组组成，对Kafka而言，消息里的数据没有特别的格式或含义。

批次就是一组消息，为了提高数据传输效率，减小网络开销，消息被分批次地写入Kafka，这些消息属于同一主题和分区。为了进一步提升数据传输和存储能力，批次数据会被进行数据压缩。

## 1.2 主题和分区

主题用于将Kafka的消息进行分类，类似于数据库中的“表”或文件系统中的“文件夹”。一个主题可以被分成若干个分区，分区是Kafka的基本存储单元。消息以追加的方式写入分区，然后以“先入先出”的顺序读取。由于一个主题可以包含若干个分区，因此无法保证在整个主题范围内消息的有序性，只能保证在单个分区中消息的有序性。如图 1所示，主题“test”包含4个分区，消息以追加的方式写入每个分区的尾部。

Kafka通过分区来实现数据的冗余性和伸缩性。分区可以分布在不同的服务器上，即一个主题可以横跨多个服务器，以此来实现比单个服务器更强大的性能。

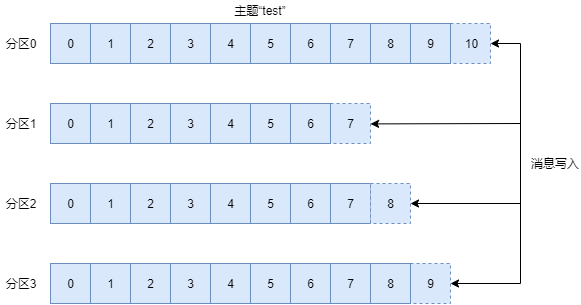


图 1 主题和分区

## 1.3 生产者和消费者

Kafka其实就是一个消息中间件，其本身作为服务端来使用，用来接收和存储消息，其客户端分为两种基本类型：生产者和消费者。

生产者，用于创建消息并把消息发布到特定的主题上，它可以把消息均衡地发布到主题的每个分区上（默认情况下），也可以把消息直接写到指定的分区（通过消息键和分区器来实现）。

消费者，用于读取消息，它通过订阅一个或多个主题并按照消息的生成顺序读取它们。消费者通过检查消息的偏移量来区分已经读取过的消息。偏移量是一个不断递增的整数值，用于记录最后读取消息的位置，在创建消息时，Kafka会把它添加到消息里，每个分区中的偏移量都是唯一的，消费者会把每个分区最后读取的消息偏移量保存在Zookeeper或Kafka上，如果消费者关闭或重启，它的读取状态不会丢失。

消费者群组由一个或多个消费者组成，也就是说，会有一个或多个消费者共同读取一个主题，消费者群组保证每个分区只能被一个消费者读取，一个消费者可以读取多个分区。如图 2所示，有3个消费者同时读取主题“test”，其中有两个消费者各自读取一个分区，另外一个消费者读取其他两个分区。通过这种方式，消费者可以读取包含大量消息的主题，如果一个消费者失效，群组里的其他消费者可以接管失效消费者的工作，保证消息不会漏读。

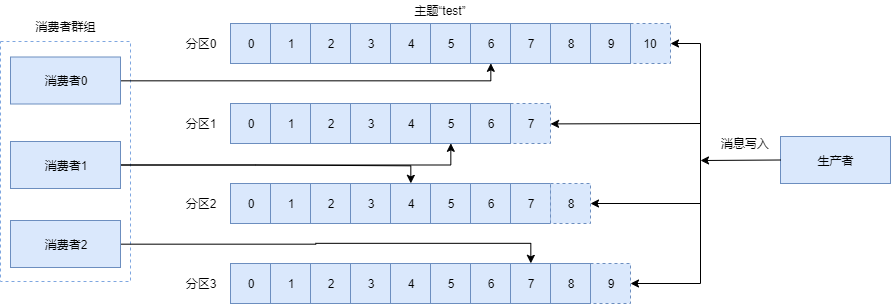


图 2 消费者群组从主题读取消息

## 1.4 broker和集群

一个独立的Kafka服务器被称为broker。broker接收来自生产者的消息，为消息设置偏移量，并提交消息到磁盘保存。broker对消费者提供服务，对读取分区的请求作出响应，返回已经提交到磁盘上的消息。根据官方描述，单个broker可以轻松处理千个分区和每秒百万级的消息量。

集群由多个broker组成，每个集群都有一个broker同时充当集群控制器的角色。控制器自动从集群的活跃成员中选举出来，负责管理工作，包括将分区分配给broker和监控broker。在集群中，一个分区从属于一个broker，该broker被称为分区的首领。一个分区可以包含多个分区副本，一个分区副本可以分配给多个broker，这时会发生分区复制，如图 3所示。分区复制机制为分区提供了消息冗余，如果有一个broker失效，其他broker可以接管领导权。

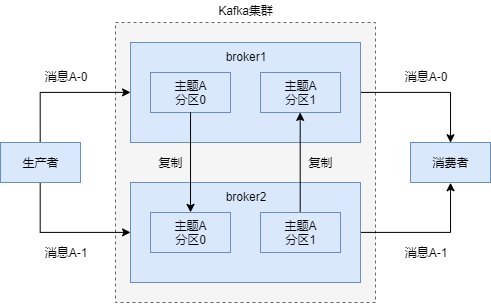


图 3 集群里的分区复制

## 1.5 消息保留

消息保留是Kafka的一个重要特性，保留策略为要么保留一段时间（可配置，默认为7天），要么保留消息达到一定大小的字节数（可配置，默认为1GB），当消息满足这些上限时，旧消息就会过期并被自动删除。

# Kafka的安装部署

Kafka是使用Java开发的应用程序，因此它可以运行在Windows、Linux、MacOS等多种操作系统上，可以以单机方式部署，也可以以集群方式部署，这里只介绍在Windows和Linux系统上的单机部署。

## 安装Java环境

Kafka的运行需要Java环境，推荐安装Java 8。如果系统已安装Java环境，可以跳过此步，具体安装过程请自行百度。

## 安装Zookeeper

Kafka使用Zookeeper保存集群的元数据信息和消费者信息。Kafka发行版中自带了Zookeeper，可以直接从脚本启动，也可以单独安装一个完整的Zookeeper，这里使用Kafka中自带的Zookeeper。

首先，修改配置文件config/zookeeper.properties，这里只修改配置dataDir，其他使用默认配置，如图 4所示：

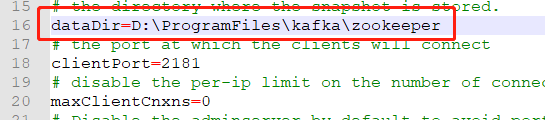


图 4 Zookeeper配置

然后，启动Zookeeper，执行命令：

Windows：D:/ProgramFiles/kafka/bin/windows/zookeeper-server-start.bat D:/ProgramFiles/kafka/config/zookeeper.properties

Linux：/data/kafka/bin/zookeeper-server-start.sh /data/kafka/config/zookeeper.properties

## 安装Kafka Broker

配置好Java和Zookeeper后，接下来就可以安装Kafka了。可以从官网<http://kafka.apache.org/downloads.html>下载最新版本的Kafka，也可以使用本文档附带的Kafka。

首先，修改配置文件config/server.properties，这里只修改listeners和log.dirs，其他使用默认配置，如图 5所示：

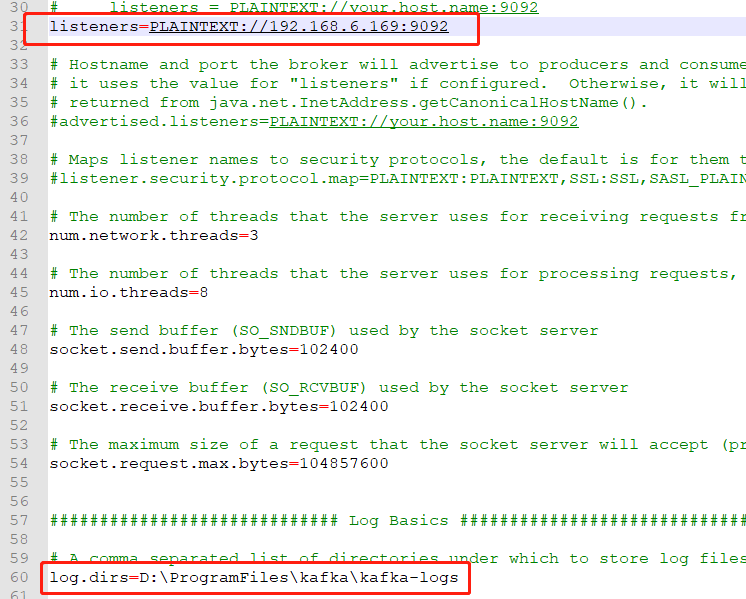


图 5 Kafka配置

然后，启动Kafka，执行命令：

Windows：D:/ProgramFiles/kafka/bin/windows/kafka-server-start.bat D:/ProgramFiles/kafka/config/server.properties

Linux：/data/kafka/bin/kafka-server-start.sh /data/kafka/config/server.properties

## Kafka Broker配置

Kafka有很多配置选项，涉及安装和调优等方面，大多数情况下可以使用默认配置。

### 常规配置

1. **broker.id**

用于指定broker的唯一标识符，默认值是0，可以设置为其他任意整数，但在整个Kafka集群中，它必须是唯一的。

1. **listeners**

用于指定Kafka Broker的监听器地址，是broker用来监听网络请求的，格式为listener\_name://host\_name:port，例如PLAINTEXT://192.168.6.169:9092，其中listener\_name表示监听器名称，这里为PLAINTEXT（注：它并不是安全协议，默认的4个安全协议已经做好了映射，只不过是监听器名称与安全协议名称恰好相同）。多个之间使用英文逗号隔开（listener\_name1://host\_name1:port1,listener\_name2://host\_name2:port2）。

1. **advertised.listeners**

用于将Kafka Broker的监听器信息发布到Zookeeper中，是暴露给外部的监听器，让其他broker或客户端可以与之通信，格式与listeners一样。如果没有设置，默认会自动使用listeners的配置。

1. **listener.security.protocol.map**

用于设置监听器名称和安全协议之间的映射关系集合，格式为listener\_name:protocol，多个之间用英文逗号隔开，默认值为PLAINTEXT:PLAINTEXT,SSL:SSL,SASL\_PLAINTEXT:SASL\_PLAINTEXT,SASL\_SSL:SASL\_SSL。

在示例listeners=PLAINTEXT://192.168.6.169:9092中，监听器名称为PLAINTEXT，从映射关系中可以看到，它对应的安全协议是PLAINTEXT。

1. **inter.broker.listener.name**

用于指定集群中各broker之间通信使用的监听器名称，如果未设置（默认情况下就没有设置），则使用security.inter.broker.protocol的配置（默认是PLAINTEXT，不能同时设置inter.broker.listener.name和security.inter.broker.protocol），如果使用了自定义的监听器名称，那么inter.broker.listener.name就是必须要设置的，例如以下配置：

*# 设置两个监听器，一个用于IPV4，一个用于IPV6*

listeners=IPV4://192.168.6.169:9092,IPV6://[2001::169]:9094

advertised.listeners=IPV4://192.168.6.169:9092,IPV6://[2001::169]:9094

*# 把IPV4的安全协议映射成PLAINTEXT，IPV6的安全协议也映射成PLAINTEXT*

listener.security.protocol.map=IPV4:PLAINTEXT,IPV6:PLAINTEXT

*# broker之间的连接使用IPV4监听器*

inter.broker.listener.name=IPV4

1. **log.dirs**

Kafka会把所有消息都持久化到磁盘上，log.dirs就用于指定存放这些日志片段的目录。如果指定多个路径，那么就用英文逗号进行分隔，broker就会根据“最少使用”原则，把同一分区的日志片段保存在同一路径下。

1. **zookeeper.connect**

用于指定Zookeeper服务的地址，格式为hostname:port，例如127.0.0.1:2181。

### 主题的默认配置

Kafka为新创建的主题提供了一些默认的配置参数，可以通过管理工具为每个主题单独配置一部分参数，比如分区个数和数据保留策略等。

1. **num.partitions**

用于指定新创建的主题包含的分区个数，默认值是1。分区的数量不宜过多，可以用主题吞吐量除以消费者吞吐量估算出分区的个数。例如，如果每秒钟要从主题上写入并读取1GB的数据，并且每个消费者每秒可以处理50MB的数据，那么至少需要20个分区，这样就可以让20个消费者同时读取这些分区，从而达到每秒1GB的吞吐量。

1. **log.retention.hours**

用于设置消息在磁盘上的保留时间，默认值为168小时，即一周。Kafka通常根据时间来决定数据可以保留多久，常用的包括log.retention.hours、log.retention.minutes和log.retention.hours，这3个参数的作用是一样的，都是决定消息多久以后会被删除，如果指定了多个参数，Kafka会优先使用具有最小值的那个参数。

根据时间保留数据是通过检查磁盘上日志片段文件的最后修改时间来实现的。一般来说，最后修改时间指的就是日志片段的关闭时间，也就是文件里最后一个消息的时间戳。

1. **log.retention.bytes**

用于设置消息在磁盘上可以保留的字节大小，从而判断消息是否过期。它作用在每一个分区上，也就是说，如果一个包含8个分区的主题，该参数设置为1GB，那么这个主题最多可以保留8GB的数据，所以当主题分区数增加时，整个主题可以保留的数据也随之增加。

如果同时制定了log.retention.bytes和log.retention.hours（或另两个时间参数），只要满足其中的任何一个，消息就会被删除。

1. **log.segment.bytes**

用于设置当前日志片段被关闭的字节大小，默认为1GB。当日志片段大小达到该参数所指定的上限时，当前日志片段就会被关闭，一个新的日志片段会被打开。如果一个日志片段被关闭，就开始等待过期，没有关闭的日志片段永远不会过期。

1. **message.max.bytes**

用于限制单个消息的大小，即broker可以接收的单个消息的最大字节数，默认值为1MB。如果生产者尝试发送的消息超过这个大小，不仅消息不会被接收，还会收到broker返回的错误信息。跟其他与字节相关的配置参数一样，该参数指的是压缩后的消息大小，也就是说，只要压缩后的消息小于这个值，消息的实际大小可以远大于这个值。

# Kafka生产者

## 生产者概览

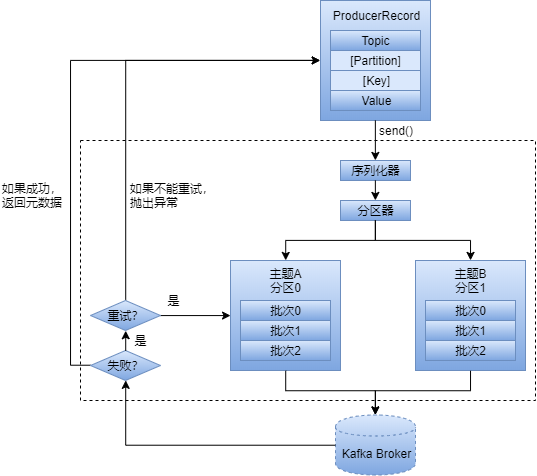


图 5 Kafka生产者过程

应用程序使用生产者创建消息并把消息发布到特定的主题上。如图 5所示，从创建一个ProducerRecord对象开始，ProducerRecord对象中需要包含目标主题和要发送的内容，还可以指定分区和键（可选）。在发送ProducerRecord对象时，生产者要先把键和值对象序列化城字节数组，这样它们才可以在网络上传输。

接下来，数据传给分区器。如果之前在ProducerRecord中指定了分区，那么分区器就不会再做任何事情，直接把指定的分区返回；如果没有指定分区，那么分区器会根据ProducerRecord中的键来选择分区（分区策略是，如果键为null，并且使用了默认分区器，那么消息将被随机且均衡地分配到主题内各个可用的分区上；如果键不为空，并且使用了默认分区器，那么Kafka会使用自己的散列算法对键进行散列，然后根据散列值把消息映射到特定分区上，此外还可以实现自定义的分区策略）。选好分区后，生产者就知道该往哪个主题和分区发送这条消息了。

然后，这条消息被添加到一个消息批次里，这个批次里的所有消息会被发送到相同主题和分区上。有一个独立的线程负责把这些批次发送到相应的broker上。

最后，broker在收到这些消息时会返回一个响应。如果消息成功写入Kafka，就返回一个RecordMetaData对象，它包含了主题和分区信息以及消息在分区里的偏移量；如果写入失败，就会返回一个错误，生产者在收到错误后会尝试重新发送消息，几次之后如果还是失败，就返回错误信息，不再重试。

## 创建Kafka生产者

要往Kafka里写入消息，首先要创建一个生产者对象，并设置一些属性。Kafka生产者有3个必选属性：

1. bootstrap.servers

用于指定broker的地址清单，格式为host:port。清单里不需要包含集群中的所有的broker地址，生产者会从给定的broker里查找其他broker信息。不过建议至少要提供两个broker信息，如果其中一个宕机，生产者仍然能够连接到集群上。

1. key.serializer

用于指定消息中键的序列化器。broker希望接收到的消息的键和值都是字节数组。生产者接口允许使用参数化类型，因此可以把Java对象作为键和值发送给broker。key.serializer必须被设置为一个实现了org.apache.kafka.common.serialization.Serializer接口的类，生产者会使用这个类把键对象序列化成字节数组。Kafka客户端默认提供了ByteArraySerializer、StringSerializer和IntegerSerializer，除此之外还可以定义自己的序列化器。

1. value.serializer

与key.serializer类似，用于指定消息中值的序列化器。

下面的代码片段演示了如何创建一个新的生产者，这里只指定了必要的属性，其他属性使用默认设置。

*// 创建Properties对象*Properties properties = **new** Properties();  
*// 设置broker地址以及键和值的序列化器*properties.put(**"bootstrap.servers"**, **"192.168.6.161:9092,192.168.6.169:9092"**);  
properties.put(**"key.serializer"**, **"org.apache.kafka.common.serialization.StringSerializer"**);  
properties.put(**"value.serializer"**, **"org.apache.kafka.common.serialization.StringSerializer"**);  
*// 创建生产者对象*KafkaProducer<String, String> producer = **new** KafkaProducer<>( properties);

## 发送消息到Kafka

首先要创建一个ProducerRecord对象，其有多个构造函数，这里使用其中一个，它需要目标主题的名字以及要发送的键和值对象。

然后调用生产者的send()方法发送ProducerRecord对象。从图 5可以看到，消息先是被放到缓冲区中，然后使用单独的线程发送到服务器端。send()方法会返回一个包含RecordMetadata的Future对象，这里忽略此返回值，所以无法知道消息是否发送成功，如果不关心发送结果，就可以使用这种方式。

我们可以忽略发送消息时可能发生的错误或在服务器端可能发生的错误，但在发送消息之前，生产者还有可能发生其他异常，如SerializationException（序列化消息失败）、BufferExhaustedException或TimeoutException（缓冲区已满）、InterruptException（发送线程被中断）等，因此需要使用try-catch捕获异常。

*// 创建ProducerRecord对象*ProducerRecord<String, String> record= **new** ProducerRecord<>(**"topic"**, **"key"**, **"value"**);  
**try** {  
 *// 发送ProducerRecord对象* producer.send(record);  
} **catch** (Exception e) {  
 e.printStackTrace();  
}

### 同步发送

**try** {  
 *// 发送ProducerRecord对象，返回RecordMetadata对象*  RecordMetadata metadata = producer.send(record).get();  
} **catch** (Exception e) {  
 e.printStackTrace();  
}

调用生产者的send()方法会先返回一个Future对象，然后调用Future对象的get()方法等待Kafka的响应。如果服务器返回错误，get()方法会抛出异常；如果没有发生错误，会得到一个RecordMetadata对象，可以用于获取消息的偏移量等信息。

KafkaProducer一般会发生两类错误。其中一类是可重试错误，这类错误可以通过重发消息来解决。比如对于连接错误，可以通过再次建立连接来解决，“无主（no leader）”错误可以通过重新为分区选举首领来解决。KafkaProducer可以被配置成自动重试，如果在多次重试后仍无法解决问题，应用程序会收到一个重试异常。另一类错误无法通过重试解决，比如“消息太大”异常，对于这类错误，KafkaProducer不会进行任何重试，直接抛出异常。

### 异步发送

假设消息在应用程序和Kafka集群之间一个来回需要10ms，如果在发送完每个消息后都等待回应，那么发送100个消息需要1秒，如果只发送消息而不等响应，那么发送100个消息所需的时间会少很多。大多数时候，我们并不需要等待响应，只需要在遇到消息发送失败时抛出异常、记录错误日志等以便排查问题和日后分析，这种消息发送方式成为异步发送。

为了在异步发送消息的同时能够对异常情况进行处理，生产者提供了回调支持，如下：

**try** {  
 *// 发送ProducerRecord对象，异步发送，支持回调* producer.send(record, **new** Callback() {  
 @Override  
 **public void** onCompletion(RecordMetadata metadata, Exception e) {  
 **if** (**null** != e) {  
 e.printStackTrace();  
 }  
 }  
 });  
} **catch** (Exception e) {  
 e.printStackTrace();  
}

## 生产者配置

除了bootstrap.servers、key.serializer、value.serializer以外，生产者还有很多其他可配参数，它们大部分都有合理的默认值，所以一般情况下不需要修改，这里介绍其中几种。

1. **acks**

用于指定必须要有多少个分区副本收到消息生产者才会认为消息写入成功，它对消息丢失的可能性有重要影响。该参数有如下选项：

（1）如果acks=0，生产者在成功写入消息之前不会等待任何来自服务器的响应，也就是说，如果当中出现了问题，导致服务器没有收到消息，那么生产者不会得知问题，消息也就丢失了。但是，由于生产者不需要等待服务器的响应，所以它可以以网络能够支持最大速度发送消息，从而达到很高的吞吐量。

（2）如果acks=1，只要集群的首领节点收到消息，生产者就会收到一个来自服务器的成功响应。如果消息无法到达首领节点（比如首领节点崩溃，新的首领还没有被选举出来），生产者会收到一个错误响应，为了避免数据丢失，生产者会重发消息。这个时候的吞吐量取决于使用的是同步发送还是异步发送，如果让客户端等待服务器的响应，显然会增加延迟。如果客户端使用回调，延迟问题就可以得到缓解，不过吞吐量还是会受发送中消息数量的限制（比如，生产者在收到服务器响应之前可以发送多少个消息）。

（3）如果acks=all，只有当所有参与复制的节点全部收到消息时，生产者才会收到一个来自服务器的成功响应。这种模式是最安全的，它可以保证不止一个服务器收到消息，就算有服务器发生崩溃，整个集群仍可以运行。不过，它的延迟也是最高的，因为我们需要等待不止一个服务器节点接收消息。

1. **buffer.memory**

用于设置生产者内存缓冲区大小，生产者用它缓存要发送到服务器的消息。如果应用程序发送消息的速度超过发送到服务器的速度，会导致生产者缓冲区空间不足，调用send()方法时要么被阻塞，要么抛出异常（取决于max.block.ms参数，表示在抛出异常之前可以阻塞多长时间）。

1. **compression.type**

用于指定消息被发送给broker之前使用的压缩算法，包括snappy、gzip、lz4。默认情况下，消息发送时不会被压缩。

1. **retries**

用于指定生产者可以重发消息的次数，如果达到这个次数，生产者会放弃重试并返回错误。默认情况下，生产者会在每次重试之间等待100ms（可以通过retry.backoff.ms参数配置）。生产者会自动进行重试，所有没必要在代码逻辑中处理那些可重试错误，只需处理那些不可重试错误或重试次数超出上限的情况。

1. **batch.size**

用于指定一个批次可以使用的内存大小（字节数），当有多个消息需要被发送同一个分区时，生产者会把它们放在同一批次里，当批次被填满（或超过一定时间，通过linger.ms参数设置），批次里的所有消息会被发送出去。

1. **linger.ms**

用于指定生产者在发送批次之前等待更多消息加入批次的时间，生产者会在批次被填满或linger.ms达到上限时把批次发送出去。

1. **client.id**

该参数可以是任意的字符串，服务器会用它来识别消息的来源，还可以用在日志和配额指标里。

1. **max.in.flight.requests.per.connection**

用于指定生产者在收到服务器响应之前可以发送多少个消息，值越大，就会占用越多的内存，不过也会提升吞吐量。设置为1可以保证消息是按照发送的顺序写入服务器的，即使发生了重试。

1. **timeout.ms、request.timeout.ms和metadata.fetch.timeout.ms**

request.timeout.ms用于指定生产者在发送数据时等待服务器返回响应的时间，metadata.fetch.timeout.ms用于指定生产者在获取元数据（如目标分区的首领是谁）时等待服务器返回响应的时间。如果等待响应超时，那么生产者要么重试发送数据，要么返回一个错误（抛出异常或执行回调）。timeout.ms用于指定broker等待同步副本返回消息确认的时间，与acks参数相匹配，如果在指定时间内没有收到同步副本的确认，那么broker就会返回一个错误。

1. **max.block.ms**

用于指定在调用send()方法或使用partitionsFor()方法获取元数据时生产者的阻塞时间，当生产者的发送缓冲区已满或没有可用的元数据时，这些方法就会阻塞，阻塞时间达到该参数值时，生产者会抛出超时异常。

1. **max.request.size**

用于指定生产者发送的请求大小，它可以指能发送的单个消息的最大值，也可以指单个请求里所有消息总的大小。例如，假设这个值为1MB，那么可以发送的单个最大消息为1MB，或生产者可以在单个请求里发送一个批次，该批次包含了1000个消息，每个消息大小为1KB。另外，broker对可接收的消息最大值也有限制（message.max.bytes），所以两边的配置最好可以匹配，避免生产者发送的消息被broker拒绝。

# Kafka消费者

## 消费者概览

应用程序使用消费者向Kafka订阅主题，并从订阅的主题上接收消息。

### 消费者和消费者群组

Kafka消费者从属于消费者群组，单个消费者的处理效率有限，而消费者群组就是对消费者的横向扩展，一个群组里的消费者订阅同一主题，每个消费者接收主题一部分分区的消息。

假设主题T1有4个分区，消费者群组G1里有且只有一个消费者C1，并且订阅了主题T1，那么消费者C1将接收到主题T1全部4个分区的消息，如图 6所示。

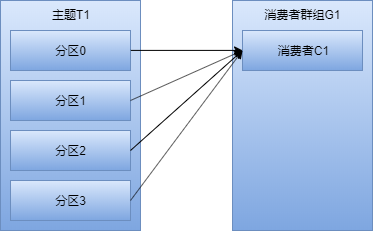


图 6 一个消费者和四个分区

如果在消费者群组G1里新增一个消费者C2，那么每个消费者将分别从两个分区获取消息，假设消费者C1接收来自分区0和分区2的消息，消费者C2接收来自分区1和分区3的消息，如图 7所示。

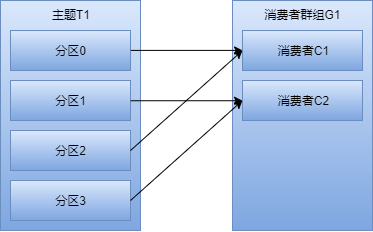


图 7 两个消费者和四个分区

如果在消费者群组G1里有4个消费者，那么每个消费者可以分配到一个分区，如图 8所示。

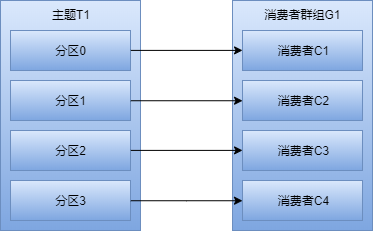


图 8 四个消费者和四个分区

如果群组里的消费者超过主题的分区数量，那么有一部分消费者就会被闲置，不会接收到任何消息，如图 9所示。

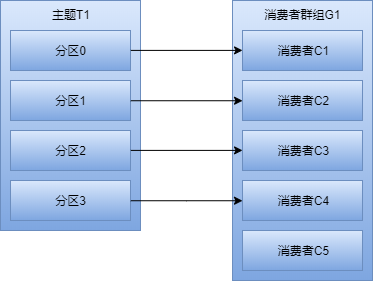


图 9 五个消费者和四个分区

在上面例子的基础上，如果新增一个群组G2，那么这个群组中的所有消费者将从主题T1上接收所有的消息，与群组G1之间互不影响，如图 10所示。

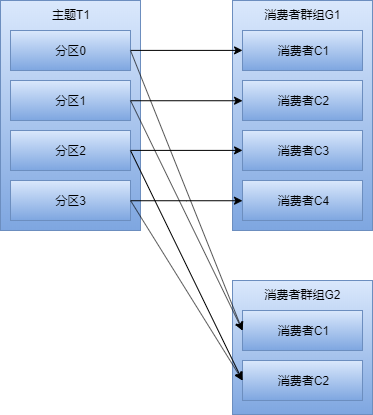


图 10 两个消费者群组对应一个主题

### 消费者群组和分区再均衡

群组里的消费者共同读取主题的分区，一个新的消费者加入群组时，它读取的是原本由其他消费者读取的消息；当一个消费者被关闭或发生崩溃时，它会离开群组，原本由他读取的分区将由群组里的其他消费者读取。在主题发生变化时，比如新增分区，会发生分区重分配。

分区的所有权从一个消费者转移到另一个消费者，这种行为被称为再均衡。再均衡为消费者群组带来了高可用性和伸缩性（可以放心地添加或移除消费者），但是在正常情况下，我们并不希望发生这样的行为，因为在再均衡期间，消费者无法读取消息，造成整个群组在一小段时间内不可用，另外，当分区被重新分配给另一个消费者时，消费者当前的读取状态会丢失，它有可能还需要去刷新缓存，在它重新恢复状态之前会拖慢应用程序。

消费者通过向被指派为群组协调器的broker（不同群组可以有不同的协调器）发送心跳来维持它们和群组的从属关系以及它们对分区的所有权关系。只要消费者以正常的时间间隔（通过heartbeat.interval.ms配置）发送心跳，就被认定为是活跃的，说明它还在读取分区里的消息。消费者会在轮询消息或提交偏移量时发送心跳。如果消费者没有在规定的时间内（通过session.timeout.ms配置）发送心跳，会话就会过期，群组协调器认为它已经死亡，就会触发一次再均衡。

## 创建Kafka消费者

在读取消息之前，需要先创建一个KafkaConsumer对象。与创建KafkaProducer类似，KafkaConsumer有3个必需的属性：bootstrap.servers、key.deserializer和value.deserializer。

其中bootstrap.servers用于指定Kafka broker的地址清单，格式为host:port（多个地址之间以英文逗号分隔）。另外两个属性key.deserializer和value.deserializer分别用于指定键和值的反序列化器，即把字节数组转换成Java对象，用法与生产者的serializer类似。

此外，消费者还有一个属性group.id，它不是必需的，不过我们现在姑且认为它是必需的。它指定了KafkaConsumer属于哪一个消费者群组，也可以创建一个不属于任何群组的消费者，但是这种做法不常见。

下面的代码片段演示了如何创建一个KafkaConsumer对象，这里只设置了必需的属性，其他属性使用默认值。

*// 创建Properties对象*Properties properties = **new** Properties();  
*// 设置broker地址、消费者群组ID、键和值的反序列化器*properties.put(**"bootstrap.servers"**, **"192.168.6.161:9092,192.168.6.169:9092"**);  
properties.put(**"group.id"**, **"group1"**);  
properties.put(**"key.deserializer"**, **"org.apache.kafka.common.serialization.StringDeserializer"**);  
properties.put(**"value.deserializer"**, **"org.apache.kafka.common.serialization.StringDeserializer"**);  
*// 创建消费者对象*KafkaConsumer<String, String> consumer = **new** KafkaConsumer<>( properties);

## 订阅主题

创建完消费者后，就要订阅主题，消费者就会从这个主题上读取消息。调用消费者的subscribe()方法，可以传入一个主题列表或正则表达式，代码如下：

*// 订阅一个主题*consumer.subscribe(Collections.*singleton*(**"topicTest"**));  
*// 订阅多个主题*consumer.subscribe(Arrays.*asList*(**"topicTest1"**, **"topicTest2"**));  
*// 使用正则表达式订阅主题*consumer.subscribe(Pattern.*compile*(**"topic.\*"**));

## 轮询

消息轮询是消费者API的核心，通过一个简单的轮询向服务器请求数据。一旦消费者订阅了主题，轮询就会处理所有细节，包括群组协调、分区再均衡、发送心跳和获取数据，开发人员只需要使用一组简单的API来处理从分区返回的数据即可，代码如下：

**try** {  
 *// 无限循环，消费者实际上是一个长期运行的应用程序，消费者必须持续地对Kafka进行轮询，否在会被认为已经死亡，它的分区会被移交给群组里的其他消费者* **while** (**true**) {  
 *// 调用poll()方法进行轮询来请求数据，参数为超时时间，用于控制poll()方法的阻塞时间（在消费者缓冲区里没有可用数据时会阻塞）  
 // 设置为0时，poll()方法会立即返回，否则它会在指定的毫秒数内一直等待broker返回数据，不管有没有可用的数据  
 // poll()方法返回一个记录列表，每天记录都包含了记录的所属主题、所在分区、在分区里的偏移量以及键和值* ConsumerRecords<String, String> records = consumer.poll(100);  
 *// 遍历返回的记录列表，逐条处理* **for** (ConsumerRecord<String, String> record : records) {  
 System.***out***.println(**"主题："** + record.topic());  
 System.***out***.println(**"分区："** + record.partition());  
 System.***out***.println(**"偏移量："** + record.offset());  
 System.***out***.println(**"键："** + record.key());  
 System.***out***.println(**"值："** + record.value());  
 }  
 }  
} **catch** (Exception e) {  
 e.printStackTrace();  
} **finally** {  
 *// 在退出应用程序之前使用close()方法关闭消费者，网络连接和socket也会随之关闭，并立即触发一次再均衡* consumer.close();  
}

## 消费者配置

除了bootstrap.servers、group.id、key.deserializer、value.deserializer以外，消费者还有很多其他可配参数，它们大部分都有合理的默认值，所以一般情况下不需要修改，这里介绍其中几种。

1. **fetch.min.bytes**

用于指定消费者从服务器获取记录的最小字节数，一般与fetch.max.wait.ms配合使用。broker在收到消费者的数据请求时，如果可用的数据量小于fetch.min.bytes指定的大小，那么它会等到有足够多的可用数据时才会把它返回给消费者。

1. **fetch.max.wait.ms**

用于指定消费者在获取数据时等待broker返回数据的时间，默认为500ms，一般与fetch.min.bytes配合使用，意思是当有足够多的数据（fetch.min.bytes）或时间超过fetch.max.wait.ms所指定的上限时，broker才会返回数据。

1. **max.partition.fetch.bytes**

用于指定服务器从每个分区里返回给消费者的最大字节数，默认值为1MB。当调用消费者的poll()方法时从每个分区里返回的记录最多不超过max.partition.fetch.bytes指定的大小。max.partition.fetch.bytes的值必须比broker能够接收的最大字节数（通过max.message.size配置）大，否则消费者可能无法读取这些消息，导致消费者一直被挂起重试。

1. **session.timeout.ms和heatbeat.interval.ms**

session.time.out用于指定消费者在被认为死亡之前可以与服务器断开连接的时间，默认为3s。如果消费者没有在session.timeout.ms指定的时间内发送心跳给群组协调器，那么就被认为已经死亡，协调器就会触发再均衡，把属于这个消费者的分区分配各群组里的其他消费者。

heartbeat.interval.ms与session.time.out紧密相关。heartbeat.interval.ms指定了poll()方法向群组协调器发送心跳的频率，而session.timeout.ms则指定了消费者可以多久不发送心跳，所以一般需要同时修改这两个属性，heartbeat.interval.ms必须比session.time.out小，一般是session.time.out的三分之一。

1. **auto.offset.reset**

用于指定消费者在读取一个没有偏移量的分区或偏移量无效的情况下（因消费者长时间失效，包含偏移量的记录已经过期并被删除）该如何处理。默认值是latest，意思是，在偏移量无效的情况下，消费者将从最新的记录开始读取数据（在消费者启动之后生产的记录）。另一个值是earliest，意思是，在偏移量无效的情况下，消费者从起始位置读取分区的记录。

1. **enable.auto.commit**

用于指定消费者是否自动提交偏移量，默认值为true。为了尽量避免出现重复数据和数据丢失，可以把它设为false，由应用程序自己控制何时提交偏移量。如果把它设置为true，可以通过配置auto.commit.interval.ms属性来控制提交的频率。

1. **partition.assignment.strategy**

分区会被分配给群组里的消费者，PartitionAssignor根据给定的消费者和主题，决定哪些分区应被分配给哪个消费者。Kafka有两个默认的分配策略。

（1）Range。该策略会把主题的若干个连续的分区分配给消费者。假设消费者C1和消费者C2同时订阅了主题T1和主题T2，并且每个主题上有3个分区，那么消费者C1有可能分配到这两个主题的分区0和分区1，而消费者C2分配到这两个主题的分区2。因为每个主题拥有奇数个分区，而分配是在主题内独立完成的，第一个消费者最后分配到比第二个消费者更多的分区。只要使用了Range策略，并且分区数量无法被消费者数量整除，就会出现这种情况。

（2）RoundRobin。该策略把主题的所有分区逐个分配给消费者。如果使用该策略给消费者C1和C2分配分区，那么消费者C1将分到主题T1的分区0和分区2以及主题2的分区1，消费者C2将分到主题T1的分区1以及主题2的分区0和分区2。一般来说，如果所有消费者都订阅相同的主题，该策略会给所有消费者分配相同数量的分区，或最多就差一个分区。

partition.assignment.strategy就用于指定分区策略，默认使用的是org.apache.kafka.clients.consumer.RangeAssignor，这个类实现了Range策略。此外，还可以使用自定义策略，这时partition.assignment.strategy的值就是自定义类的名字。

1. **client.id**

用于标识从客户端发送过来的消息，可以是任意字符串，通常被用在日志、度量指标和配额里。

1. **max.poll.records**

用于指定单次调用消费者的poll()方法时返回的最大数据量。

## 提交和偏移量

每次调用消费者的poll()方法总是返回由生产者写入Kafka但还没有被消费者读取过的记录。我们把消息在分区里的位置叫做偏移量，把更新分区当前位置的操作叫做提交。

消费者往一个叫做\_consumer\_offset的特殊主题发送消息，消息里包含每个分区的偏移量，如果消费者一直处于运行状态，那么偏移量就没有什么用处，不过如果消费者发生崩溃或有新的消费者加入群组，就会触发再均衡，完成再均衡之后，每个消费者可能分配到新的分区，而不是之前处理那个，为了能够继续之前的工作，消费者需要读取每个分区最后一次提交的偏移量，然后从偏移量指定的地方继续处理。

如果提交的偏移量小于客户端处理的最后一个消息的偏移量，那么处于两个偏移量之间的消息就会被重复处理，如图 11所示。

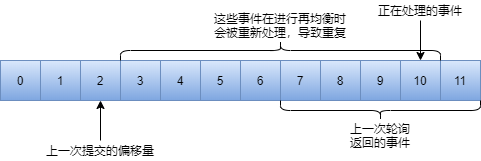


图 11 提交的偏移量小于客户端处理的最后一个消息的偏移量

如果提交的偏移量大于客户端处理的最后一个消息的偏移量，那么处于来那个偏移量之间的消息将会丢失，如图 12所示。

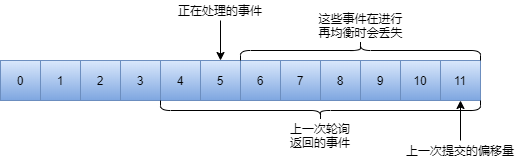


图 12 提交的偏移量大于客户端处理的最后一个消息的偏移量

所以，处理偏移量的方式对客户端会有很大的影响，KafkaConsumer API提供了多种方式来提交偏移量。

### 自动提交

最简单的提交方式是让消费者自动提交偏移量。如果enable.auto.commit被设置为true，那么每隔5s（通过auto.commit.interval.ms配置，默认5s），消费者会自动把从poll()方法接收到的最大偏移量提交上去，这个过程是在轮询里进行的，消费者每次在进行轮询时会检查是否该提交偏移量了，如果是，那么就会提交从上一次轮询返回的偏移量。

假设我们使用默认的5s提交时间间隔，在最近一次提交后的3s发生了再均衡，再均衡之后，消费者从最后一次提交的偏移量位置开始读取消息，这个时候偏移量已经落后了3s，所以在这3s内到达的消息会被重复处理，可以通过修改提交时间间隔来更频繁地提交偏移量，减小可能出现重复消息的时间窗，但这种情况是无法完全避免的。

在使用自动提交时，每次调用轮询方法都会把上一次调用返回的偏移量提交上去，它并不知道具体哪些消息已经被处理了，所以在再次调用之前最好确保所有当前调用返回的消息都已处理完成（在调用close()方法之前也会进行自动提交），一般情况下不会出现什么问题，但在处理异常或提前退出轮询时要格外小心。

### 手动提交

消费者API提供了手动提交偏移量的方式，把auto.commit.offset设为false，让应用程序决定何时提交偏移量。可以使用commitSync()方法进行提交，这个API会提交由poll()方法返回的最新偏移量，需要在处理完所有记录后调用该方法，提交成功后马上返回，如果发生可恢复性的错误，commitSync()方法会一直重试直到提交成功，如果发生不可恢复性的错误或者重试次数超出上限，则直接抛出异常，代码如下：

**while** (**true**) {  
 ConsumerRecords<String, String> records = consumer.poll(100);  
 **for** (ConsumerRecord<String, String> record : records) {  
 System.***out***.println(**"值："** + record.value());  
 }  
 **try** {  
 *// 提交当前批次最新的偏移量* consumer.commitSync();  
 } **catch** (CommitFailedException e) {  
 e.printStackTrace();  
 }  
}

### 异步提交

手动提交有一个不足之处，在broker对提交请求作出回应之前，应用程序会一直阻塞，这样会限制应用程序的吞吐量。为了弥补这个不足，Kafka提供了异步提交API，我们只管发送提交请求，不需要等待broker的响应。使用起来很简单，调用消费者的commitAsync()方法即可，该方法没有重试机制，支持回调，在broker作出响应时执行回调，常用于日志记录，代码如下：

**while** (**true**) {  
 ConsumerRecords<String, String> records = consumer.poll(100);  
 **for** (ConsumerRecord<String, String> record : records) {  
 System.***out***.println(**"值："** + record.value());  
 }  
 *// 异步提交，没有回调* consumer.commitAsync();  
 *// 异步提交，支持回调* consumer.commitAsync(**new** OffsetCommitCallback() {  
 @Override  
 **public void** onComplete(Map<TopicPartition, OffsetAndMetadata> map, Exception e) {  
 **if** (e != **null**) {  
 System.***out***.println(e.getMessage());  
 }  
 }  
 });  
}

### 同步和异步组合提交

一般情况下，针对偶尔出现的提交失败，不进行重试不会有太大问题，因为如果提交失败是因为临时问题导致的，那么后续的提交总会有成功的。但如果这是发生在关闭消费者或再均衡的最后一次提交，就要确保能够提交成功。因此，在消费者关闭之前一般会组合使用commitAsync()和commitSync()，代码如下：

**try** {  
 **while** (**true**) {  
 ConsumerRecords<String, String> records = consumer.poll(100);  
 **for** (ConsumerRecord<String, String> record : records) {  
 System.***out***.println(**"值："** + record.value());  
 }  
 *// 异步提交* consumer.commitAsync();  
 }  
} **catch** (Exception e) {  
 e.printStackTrace();  
} **finally** {  
 **try** {  
 *// 同步提交* consumer.commitSync();  
 } **catch** (CommitFailedException e) {  
 e.printStackTrace();  
 } **finally** {  
 consumer.close();  
 }  
}

### 提交特定的偏移量

提交偏移量的频率与处理消息批次的频率是一样的，但如果想要更频繁的提交或想要在批次中间提交该怎么办？消费者API允许在调用commitSync()和commitAsync()时传入一个map（key为TopicPartition，value为OffsetAndMetadata），这样就可以实现提交特定的偏移量，代码如下：

Map<TopicPartition, OffsetAndMetadata> offsetMap = **new** HashMap<>();  
**int** count = 0;  
**while** (**true**) {  
 ConsumerRecords<String, String> records = consumer.poll(100);  
 **for** (ConsumerRecord<String, String> record : records) {  
 System.***out***.println(**"值："** + record.value());  
 *// 每处理1000条记录提交一次* TopicPartition topicPartition = **new** TopicPartition(record.topic(), record.partition());  
 OffsetAndMetadata offsetAndMetadata = **new** OffsetAndMetadata(record.offset() + 1, **""**);  
 offsetMap.put(topicPartition, offsetAndMetadata);  
 **if** (count % 1000 == 0) {  
 consumer.commitAsync(offsetMap, **null**);  
 }  
 count++;  
 }  
}

# Kafka原理

## 集群成员关系

Kafka使用Zookeeper维护集群成员的信息。每个broker都有一个唯一标识符（broker.id，可配置），在broker启动时，它通过创建临时节点把自己的ID注册到Zookeeper上（注册路径为/brokers/ids），Kafka组件监听broker在Zookeeper中的注册路径，当有broker加入集群或退出集群时，这些组件就可以获得通知。

## 控制器

控制器其实就是一个broker，只不过它除了具有一般broker的功能之外，还负责分区首领的选举。集群里第一个启动的broker通过在Zookeeper中创建一个临时节点/controller来让自己成为控制器，其他broker在启动时也会尝试创建该节点，但会收到一个“节点已存在”异常，然后意识到控制器节点已存在，也就是说集群里已经有一个控制器了。

为了让其他broker可以收到控制器节点的变更通知，确保集群里只有一个控制器存在，它们会在控制器节点上创建一个监听器，如果控制器被关闭或与Zookeeper断开连接，Zookeeper上的临时节点就会消失，集群里的其他broker通过监听器得到控制器节点消失的通知，它们就会尝试让自己成为新的控制器，第一个在Zookeeper上成功创建控制器节点的broker就会成为新的控制器，其他broker会收到“节点已存在”异常，然后在新的控制器节点上再次创建监听器。

当控制器发现一个broker已经离开集群，它就知道，那些失去首领的分区需要一个新首领（这些分区的首领刚好是在这个broker上）。控制器遍历这些分区，并确定谁应该成为新首领（简单来说就是分区副本列表里的下一个副本），然后向所有包含新首领或现有跟随者的broker发送请求，请求信息包含了谁是新首领以及谁是分区跟随者的信息。随后，新首领开始处理来自生产者和消费者的请求，而跟随者开始从新首领那里复制消息。

当控制器发现一个broker加入集群时，它会使用broker id来检查新加入的broker是否包含现有分区的副本，如果有，控制器就把变更通知发送给新加入的broker和其他broker，新broker上的副本开始从首领那里复制消息。

## 复制

复制功能是Kafka分布式架构的核心，它可以在个别节点失效时仍能保证Kafka的可用性和持久性。

Kafka使用主题来组织数据，每个主题分为若干个分区，每个分区有多个副本，副本保存在broker上。有两种类型的副本，一种是首领副本，一种是跟随者副本。每个分区都有一个首领副本，所有生产者请求和消费者请求都会经过首领副本；首领以外的副本都是跟随者副本，跟随者副本不处理来自客户端的请求，它们唯一的认为就是从首领那里复制消息，保持与首领一致的状态，如果首领发生崩溃，其中的一个跟随者就会成为新的首领。

首领的另一个任务是确认哪个跟随者的状态与自己是一致的。跟随者为了与首领保持同步，在有新消息到达时尝试从首领那里复制消息，即向首领发送获取数据的请求，请求消息里包含了跟随者想要获取消息的偏移量，并且这些偏移量总是有序的，然后首领将响应消息发送给跟随者。

假设一个跟随者副本先请求消息1，接着请求消息2，然后请求消息3，在收到这3个请求的响应之前是不会发送第4个请求消息的，如果跟随者发送了请求消息4，那么首领就知道它已经收到了前面3个请求的响应。通过查看每个跟随者请求的最新偏移量，首领就会知道每个跟随者的复制进度。如果跟随者在10s内没有请求任何消息或者虽然请求消息但在10s内没有请求最新的数据（时间可以通过replica.lag.time.max.ms参数配置），那么它就会被认为是不同步的。如果一个副本无法与首领保持一致，那么在首领发生崩溃时，它是不可能成为新的首领的。相反，持续请求得到最新消息的副本被称为同步的副本，在首领发生崩溃时，只有同步副本才有可能成为新的首领。

## 处理请求

broker的大部分工作是处理客户端、分区副本和控制器发送给分区首领的请求。Kafka提供了一个基于TCP的二进制协议，指定了请求消息的格式以及broker如何对请求作出响应。客户端发起连接并发送请求，broker按照请求到达的顺序处理它们并作出响应。

常见的请求类型包括生产请求、获取请求、元数据请求。

### 生产请求

生产请求就是生产者发送的请求，包含客户端要写入broker的消息。生产请求必须发送给分区的首领副本，包含首领副本的broker在收到生产请求时，会对请求进行一些验证，然后消息被写入本地磁盘（Linux系统中，消息会被写到文件系统缓存里，并不保证何时会被刷新到磁盘）。在消息被写入分区的首领后，broker开始检查acks配置参数——如果acks为0或1，那么broker立即返回响应；如果acks为all，那么请求会被保存在一个缓冲区里，直到首领发现所有跟随者副本都复制了消息，才会给客户端返回响应。

### 获取请求

生产请求就是在消费者和跟随者副本需要从broker读取消息时发送的请求，broker处理获取请求的方式与处理生产请求的方式类似，客户端发送请求，向broker请求主题分区里具有特定偏移量的消息，客户端还可以指定broker最多可以从一个分区里返回多少数据（通过max.partition.fetch.bytes参数配置）。

与生产请求一样，获取请求必须先到达指定的分区首领上，首领在收到请求时，先会检查请求是否有效——比如，指定的偏移量在分区上是否存在？如果客户端请求的数据被删除或请求的偏移量不存在，那么broker将返回一个错误；如果请求的偏移量存在，broker将按照客户端指定的数量上限从分区里读取消息，再把消息返回给客户端。Kafka直接把消息从文件（准确来说是Linux文件系统缓存）里发送到网络通道，而不需要经过任何中间缓冲区。

客户端除了可以设置broker返回数据的上限外，还可以设置下限和超时时间。客户端发送一个请求，broker等到有足够多的数据或等待一段时间后才把它们返回给客户端，就好像在告诉broker：“如果你无法n毫秒内累计满足要求的数据量，那么就把当前的这些数据返回给我”，如图 14所示。

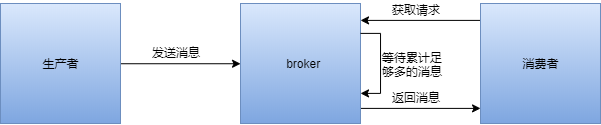


图 14 broker延迟作出响应以便累计足够多的数据

此外，客户端只能读取已经被写入所有同步副本的消息。分区首领知道每个消息会被复制到哪些副本上，在消息还没有被写入所有同步副本之前，是不会发送给消费者的（尝试获取这些消息的请求会得到空的响应），如图 15所示。

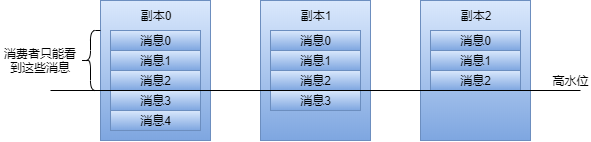


图 15 消费者只能看到已经同步到所有副本的消息

### 元数据请求

生产请求和获取请求都必须发送给分区的首领副本，如果broker收到一个针对特定分区的请求，而该分区的首领在另一个broker上，那么发送请求的客户端会收到一个“非分区首领”的错误响应。

Kafka客户端要自己负责把生产请求和获取请求发送到正确的broker上，这就要使用另一种请求类型，叫做元数据请求。这种请求包含了客户端感兴趣的主题列表，服务器端的响应消息里指明了这些主题所包含的分区、每个分区都有哪些副本、以及哪个副本是首领，所有的broker都缓存了这些信息，因此元数据请求可以发送给任意一个broker。

客户端会把这些信息缓存起来，并直接往目标broker上发送生产请求和获取请求。它们需要定时发送元数据请求来刷新这些信息（刷新时间间隔通过metadata.max.age.ms参数配置），从而知道元数据是否发生了变化。此外，如果客户端收到“非首领”错误，它会在重发请求之前先刷新元数据，如图 16所示。

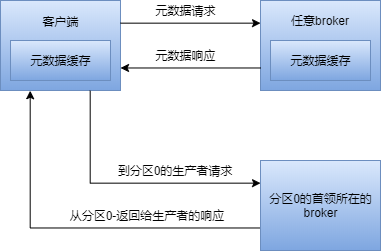


图 16 客户端请求和元数据请求

## 物理存储

### 分区分配

Kafka的基本存储单元是分区。在创建主题时，Kafka首先会决定如何在broker间分配分区。假设有6个broker，打算创建一个包含10个分区的主题，复制系数是3，即每个分区有3个副本，那么一共有30个分区副本，它们被分配到6个broker上，分配的原则是：

（1）在broker之间平均地分配分区副本。在我们的例子中，就是要保证每个broker可以分到5个副本；

（2）确保每个分区的每个副本分布在不同的broker上。假设分区0的首领副本在broker2上，那么可以把跟随者副本放在broker3和broker4上，但不能放在broker2上，也不能两个都放在broker3上；

（3）如果为broker指定了机架信息，那么尽可能把每个分区副本分配到不同机架的broker上。这样可以保证一个机架的不可用不会导致整体的分区不可用。

遵循以上原则，我们先随机选择一个broker（假设是4），然后使用轮询的方式给每个broker分配分区来确定首领分区的位置。于是，首领分区0会在broker4上，首领分区1会在broker5上，首领分区2会在broker0上，首领分区3会在broker1上，以此类推。然后，我们从分区首领开始，依次分配跟随者副本。分区0的首领副本在broker4上，那么它的第一个跟随者副本会在broker5上，第二个跟随者副本会在broker0上；分区1的首领副本在broker5上，那么它的第一个跟随者副本会在broker0上，第二个跟随者副本会在broker1上，以此类推。

如果配置了机架信息，那么就不是按照数字顺序来选择broker了，而是按照交替机架的方式。假设broker0、broker1、broker2放在同一机架上，broker3、broker4、broker5放在其他不同机架上。我们不是按照0到5的顺序来选择broker，而是按照0，3，1，4，2，5的顺序来选择。于是，如果分区0的首领在broker4上，那么第一个跟随者副本会在broker2上，第二个跟随者副本会在broker5上，以此类推。

### 文件管理

保留数据是Kafka的一个基本特性，Kafka不会一直保留数据，也不会等到所有消费者都读取了消息之后才删除消息。Kafka为每个主题配置了数据保留期限，规定了数据被删除之前可以保留多长时间，或者清理数据之前可以保留的数据量大小。

在一个大文件里查找和删除消息是很费时的，也很容易出错，因此Kafka把分区分成若干个片段。默认情况下，每个片段包含1GB或一周的数据，以较小地哪个为准。在broker往分区写入数据时，如果达到片段上限，就关闭当前文件，并打开一个新文件。当前正在写入的数据片段叫做活跃片段，活跃片段永远不会被删除。

### 文件格式

Kafka把消息和偏移量保存在文件里，保存在磁盘上的数据格式与从生产者发送过来或发送给消费者的消息格式是一样的。

除了键、值和偏移量之外，消息里还包含了消息大小、校验和、消息格式版本号、压缩算法（Snappy、Gzip、LZ4）和时间戳。时间戳可以是生产者发送消息的时间，也可以是消息到达broker的时间，可配。

如果生产者发送的是压缩过的消息，那么同一个批次的消息会被压缩在一起，被当做“包装消息”进行发送。于是，broker就会收到一个这样的消息，然后再把它发送给消费者，消费者在解压这个消息后，会看到整个批次的消息，它们都有自己的时间戳和偏移量，如图 17所示。



图 17 普通消息和包装消息

### 索引

消费者可以从Kafka的任意可用偏移量位置开始读取消息。假设消费者要读取从偏移量50开始的1MB消息，那么broker必须立即定位到偏移量50（可能在分区的任意一个片段中），然后开始从这个位置读取消息。为了使得broker能够快速地定位到指定偏移量，Kafka为每个分区维护了一个索引，索引把偏移量映射到片段文件盒偏移量在文件里的位置，与分区类似，索引也被分成若干个片段。

# Kafka常用命令

## 启动Kafka

### 启动Zookeeper

(/data/kafka/bin/zookeeper-server-start.sh /data/kafka/config/zookeeper.properties &)

### 启动Kafka Server

(/data/kafka/bin/kafka-server-start.sh /data/kafka/config/server.properties &)

## 主题操作

### 创建主题

/data/kafka/bin/kafka-topic.sh --bootstrap-server 192.168.6.169:9092 --create --topic testTopic --replication-factor 2 --partitions 3

### 删除主题

/data/kafka/bin/kafka-topic.sh --bootstrap-server 192.168.6.169:9092 --delete --topic testTopic

### 列出集群里的所有主题

/data/kafka/bin/kafka-topics.sh --bootstrap-server 192.168.6.169:9092 --list

### 列出主题详细信息

/data/kafka/bin/kafka-topics.sh --bootstrap-server 192.168.6.169:9092 --describe --topic testTopic

## 消费者群组

### 列出所有消费者群组

/data/kafka/bin/kafka-consumer-group.sh --bootstrap-server 192.168.6.169:9092 --list

### 列出群组的详细信息

/data/kafka/bin/kafka-consumer-group.sh --bootstrap-server 192.168.6.169:9092 --describe --group testGroup

## 生产和消费

### 控制台生产者

/data/kafka/bin/kafka-console-producer.sh --broker-list 192.168.6.169:9092 --topic testTopic

### 控制台消费者

/data/kafka/bin/kafka-console-consumer.sh --bootstrap-server 192.168.6.169:9092 --topic testTopic