Kafka 入门

什么是 Kafka

kafka 最初是 LinkedIn 的一个内部基础设施系统。最初开发的起因是，LinkedIn 虽然有了数据库和其他系统可以用来存储数据，但是缺乏一个可以帮助处理持续数据流的组件。所以在设计理念上，开发者不想只是开发一个能够存储数据的系统，如关系数据库、Nosql 数据库、搜索引擎等等，更希望把数据看成一个持续变化和不断增长的流，并基于这样的想法构建出一个数据系统，一个数据架构。

Kafka 外在表现很像消息系统，允许发布和订阅消息流，但是它和传统的消息系统有很大的差异，

首先，Kafka 是个现代分布式系统，以集群的方式运行，可以自由伸缩。

其次，Kafka 可以按照要求存储数据，保存多久都可以，

第三，流式处理将数据处理的层次提示到了新高度，消息系统只会传递数据，Kafka 的流式处理能力可以让我们用很少的代码就能动态地处理派生流和数据集。所以 Kafka 不仅仅是个消息中间件。

Kafka 不仅仅是一个消息中间件，同时它是一个流平台，这个平台上可以发布和订阅数据流（Kafka 的流，有一个单独的包 Stream 的处理），并把他们保存起来，进行处理，这个是 Kafka 作者的设计理念。

大数据领域，Kafka 还可以看成实时版的 Hadoop，但是还是有些区别，Hadoop 可以存储和定期处理大量的数据文件，往往以 TB 计数，而 Kafka 可以存储和持续处理大型的数据流。Hadoop 主要用在数据分析上，而 Kafka 因为低延迟，更适合于核心的业务应用上。所以国内的大公司一般会结合使用，比如京东在实时数据计算架构中就使用了到了 Kafka,具体见《张开涛-海量数据下的应用系统架构实践》

常见的大数据处理框架：storm、spark、Flink、(Blink 阿里)

Kafka 名字的由来：卡夫卡与法国作家马塞尔·普鲁斯特，爱尔兰作家詹姆斯·乔伊斯并称为西方现代主义文学的先驱和大师。《变形记》是卡夫卡的短篇代表作，是卡夫卡的艺术成就中的一座高峰，被认为是 20 世纪最伟大的小说作品之一（达到管理层的高度应该多看下人文相关的书籍，增长管理知识和人格魅力）。

本次课程，将会以 kafka_2.11-2.3.0 版本为主，其余版本不予考虑，并且 Kafka 是 scala 语言写的，小众语言,没有必要研究其源码，投入和产出比低，除非你的技术级别非常高或者需要去开发单独的消息中间件。

Kafka 中的基本概念

消息和批次

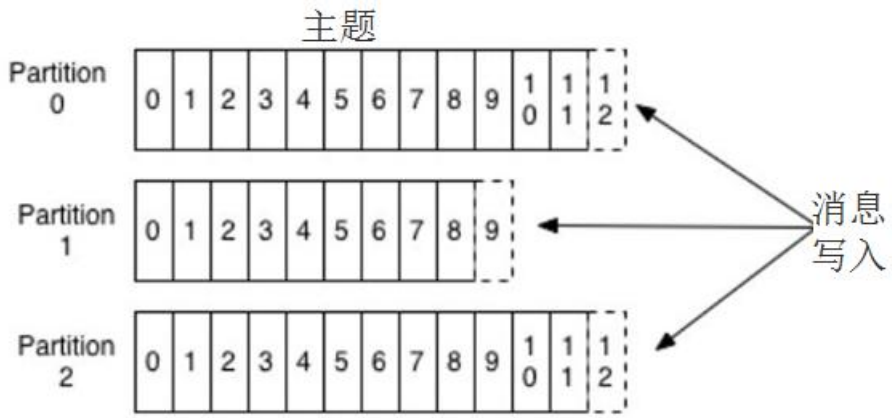
消息，Kafka 里的数据单元，也就是我们一般消息中间件里的消息的概念（可以比作数据库中一条记录）。消息由字节数组组成。消息还可以包含键（可选元数据，也是字节数组），主要用于对消息选取分区。

作为一个高效的消息系统，为了提高效率，消息可以被分批写入 Kafka。**批次**就是一组消息，这些消息属于同一个主题和分区。如果只传递单个消息，会导致大量的网络开销，把消息分成批次传输可以减少这开销。但是，这个需要权衡（时间延迟和吞吐量之间），批次里包含的消息越多，单位时间内处理的消息就越多，单个消息的传输时间就越长（吞吐量高延时也高）。如果进行压缩，可以提升数据的传输和存储能力，但需要更多的计算处理。

对于 Kafka 来说，消息是晦涩难懂的字节数组，一般我们使用序列化和反序列化技术，格式常用的有 JSON 和 XML，还有 Avro（Hadoop 开发的一款序列化框架），具体怎么使用依据自身的业务来定。

主题和分区

Kafka 里的消息用**主题**进行分类（主题好比数据库中的表），主题下有可以被分为若干个**分区（分表技术）**。分区本质上是个提交日志文件，有新消息，这个消息就会以追加的方式写入分区（写文件的形式），然后用先入先出的顺序读取。



但是因为主题会有多个分区，所以在整个主题的范围内，是无法保证消息的顺序的，单个分区则可以保证。

Kafka 通过分区来实现数据冗余和伸缩性，因为分区可以分布在不同的服务器上，那就是说一个主题可以跨越多个服务器（这是 Kafka 高性能的一个原因，多台服务器的磁盘读写性能比单台更高）。

前面我们说 Kafka 可以看成一个流平台，很多时候，我们会把一个主题的数据看成一个流，不管有多少个分区。

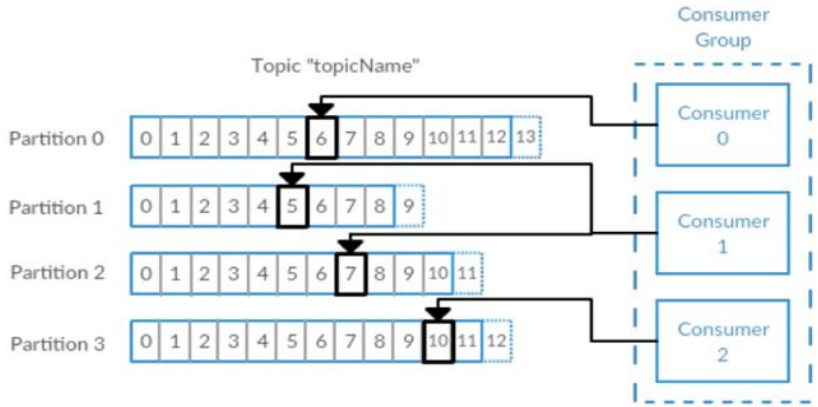
生产者和消费者、偏移量、消费者群组

就是一般消息中间件里生产者和消费者的概念。一些其他的高级客户端 API，像数据管道 API 和流式处理的 Kafka Stream，都是使用了最基本的生产者和消费者作为内部组件，然后提供了高级功能。

生产者默认情况下把消息均衡分布到主题的所有分区上，如果需要指定分区，则需要使用消息里的消息键和分区器。

消费者订阅主题，一个或者多个，并且按照消息的生成顺序读取。消费者通过检查所谓的偏移量来区分消息是否读取过。偏移量是一种元数据，一个不断递增的整数值，创建消息的时候，Kafka 会把他加入消息。在一个主题中一个分区里，每个消息的偏移量是唯一的。每个分区最后读取的消息偏移量会保存到 Zookeeper 或者 Kafka 上，这样分区的消费者关闭或者重启，读取状态都不会丢失。

多个消费者可以构成一个消费者群组。怎么构成？共同读取一个主题的消费者们，就形成了一个群组。群组可以保证每个分区只被一个消费者使用。



消费者和分区之间的这种映射关系叫做消费者对分区的所有权关系，很明显，一个分区只有一个消费者，而一个消费者可以有多个分区。

（吃饭的故事：一桌一个分区，多桌多个分区，生产者不断生产消息(消费)，消费者就是买单的人，消费者群组就是一群买单的人），一个分区只能被消费者群组中的一个消费者消费（不能重复消费），如果有一个消费者挂掉了<James 跑路了>，另外的消费者接上）

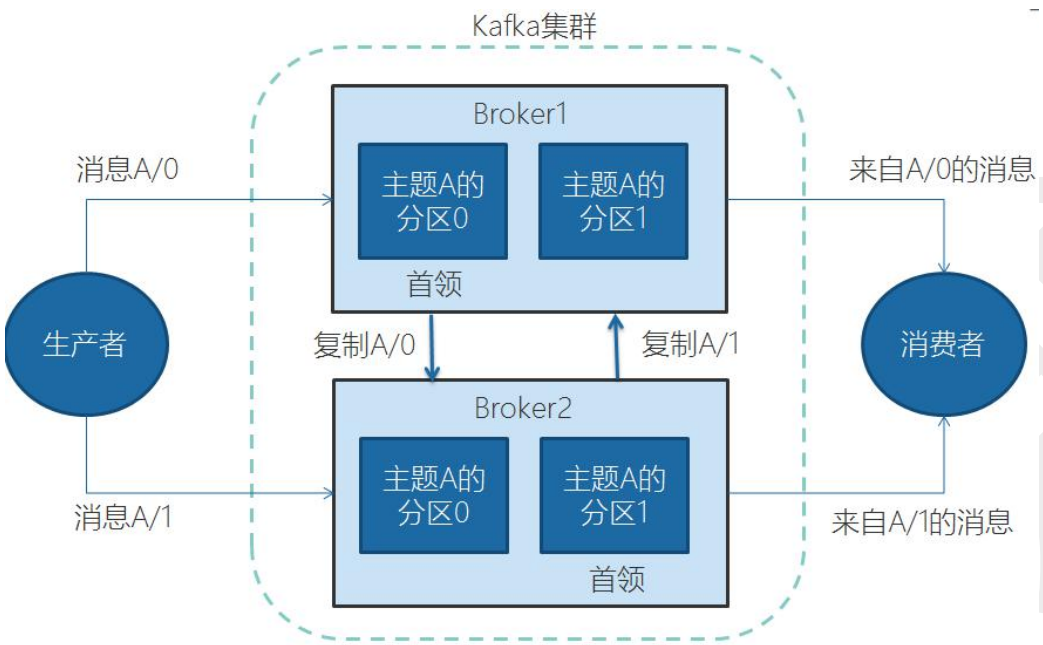
Broker 和集群

一个独立的 Kafka 服务器叫 Broker。broker 的主要工作是，接收生产者的消息，设置偏移量，提交消息到磁盘保存；为消费者提供服务，响应请求，返回消息。在合适的硬件上，单个 broker 可以处理上千个分区和每秒百万级的消息量。（要达到这个目的需要做操作系统调优和 JVM 调优）

多个 broker 可以组成一个集群。每个集群中 broker 会选举出一个集群控制器。控制器会进行管理，包括将分区分配给 broker 和监控 broker。

集群里，一个分区从属于一个 broker，这个 broker 被称为首领。但是分区可以被分配给多个 broker，这个时候会发生分区复制。

集群中 Kafka 内部一般使用管道技术进行高效的复制。



分区复制带来的好处是，提供了消息冗余。一旦首领 broker 失效，其他 broker 可以接管领导权。当然相关的消费者和生产者都要重新连接到新的首领上。

保留消息

在一定期限内保留消息是 Kafka 的一个重要特性，Kafka　broker 默认的保留策略是：要么保留一段时间（7 天），要么保留一定大小（比如 1 个 G）。到了限制，旧消息过期并删除。但是每个主题可以根据业务需求配置自己的保留策略（开发时要注意，Kafka 不像 Mysql 之类的永久存储）。

为什么选择 Kafka

优点

多生产者和多消费者

基于磁盘的数据存储，换句话说，Kafka 的数据天生就是持久化的。

高伸缩性，Kafka 一开始就被设计成一个具有灵活伸缩性的系统，对在线集群的伸缩丝毫不影响整体系统的可用性。

高性能，结合横向扩展生产者、消费者和 broker，Kafka 可以轻松处理巨大的信息流（LinkedIn 公司每天处理万亿级数据），同时保证亚秒级的消息延迟。

常见场景

活动跟踪

跟踪网站用户和前端应用发生的交互，比如页面访问次数和点击，将这些信息作为消息发布到一个或者多个主题上，这样就可以根据这些数据为机器学习提供数据，更新搜素结果等等（头条、淘宝等总会推送你感兴趣的内容，其实在数据分析之前就已经做了活动跟踪）。

传递消息

标准消息中间件的功能

收集指标和日志

收集应用程序和系统的度量监控指标，或者收集应用日志信息，通过 Kafka 路由到专门的日志搜索系统，比如 ES。（国内用得较多）

提交日志

收集其他系统的变动日志，比如数据库。可以把数据库的更新发布到 Kafka 上，应用通过监控事件流来接收数据库的实时更新，或者通过事件流将数据库的更新复制到远程系统。

还可以当其他系统发生了崩溃，通过重放日志来恢复系统的状态。（异地灾备）

流处理

操作实时数据流，进行统计、转换、复杂计算等等。随着大数据技术的不断发展和成熟，无论是传统企业还是互联网公司都已经不再满足于离线批处理，实时流处理的需求和重要性日益增长 。

近年来业界一直在探索实时流计算引擎和 API，比如这几年火爆的 Spark Streaming、Kafka Streaming、Beam 和 Flink，其中阿里双 11 会场展示的实时销售金额，就用的是流计算，是基于 Flink，然后阿里在其上定制化的 Blink。

Kafka 的安装、管理和配置

安装

预备环境

Kafka 是 Java 生态圈下的一员，用 Scala 编写，运行在 Java 虚拟机上，所以安装运行和普通的 Java 程序并没有什么区别。

安装 Kafka 官方说法，Java 环境推荐 Java8。

Kafka 需要 Zookeeper 保存集群的元数据信息和消费者信息。Kafka 一般会自带 Zookeeper，但是从稳定性考虑，应该使用单独的 Zookeeper，而且构建 Zookeeper 集群。

下载和安装 Kafka

在 http://kafka.apache.org/downloads 上寻找合适的版本下载，我们这里选用的是 kafka_2.11-2.3.0，下载完成后解压到本地目录。

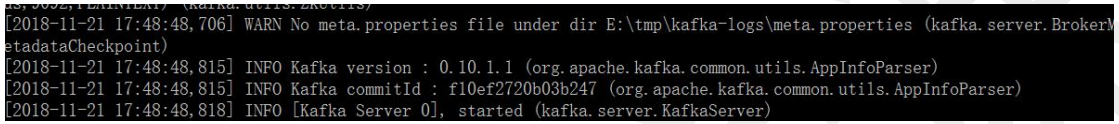
运行

启动 Zookeeper

进入 Kafka 目录下的 bin\windows

执行 kafka-server-start.bat ../../config/server.properties，出现以下画面表示成功

Linux 下与此类似，进入 bin 后，执行对应的 sh 文件即可



kafka 基本的操作和管理

##列出所有主题

kafka-topics.bat --zookeeper localhost:2181 --list

##列出所有主题的详细信息

kafka-topics.bat --zookeeper localhost:2181 --describe

##创建主题 主题名 my-topic，1 副本，8 分区

kafka-topics.bat --zookeeper localhost:2181 --create --topic my-topic --replication-factor 1 --partitions 8

##增加分区，注意：分区无法被删除

kafka-topics.bat --zookeeper localhost:2181 --alter --topic my-topic --partitions 16

##创建生产者（控制台）

kafka-console-producer.bat --broker-list localhost:9092 --topic my-topic

##创建消费者（控制台）

kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic my-topic --from-beginning

##列出消费者群组（仅 Linux）

kafka-topics.sh --new-consumer --bootstrap-server localhost:9092 --list

##列出消费者群组详细信息（仅 Linux）

kafka-topics.sh --new-consumer --bootstrap-server localhost:9092 --describe --group 群组名

Broker 配置

配置文件放在 Kafka 目录下的 config 目录中，主要是 server.properties 文件

常规配置

broker.id

在单机时无需修改，但在集群下部署时往往需要修改。它是个每一个 broker 在集群中的唯一表示，要求是正数。当该服务器的 IP 地址发生改变时，broker.id 没有变化，则不会影响 consumers 的消息情况

listeners

监听列表(以逗号分隔 不同的协议(如 plaintext,trace,ssl、不同的 IP 和端口)),hostname 如果设置为 0.0.0.0 则绑定所有的网卡地址；如果 hostname 为空则绑定默认的网卡。如果
没有配置则默认为 java.net.InetAddress.getCanonicalHostName()。

如：PLAINTEXT://myhost:9092,TRACE://:9091 或 PLAINTEXT://0.0.0.0:9092,

zookeeper.connect

zookeeper 集群的地址，可以是多个，多个之间用逗号分割。（一组 hostname:port/path 列表,hostname 是 zk 的机器名或 IP、port 是 zk 的端口、/path 是可选 zk 的路径，如果不指定，默认使用根路径）

log.dirs

Kafka 把所有的消息都保存在磁盘上，存放这些数据的目录通过 log.dirs 指定。可以使用多路径，使用逗号分隔。如果是多路径，Kafka 会根据“最少使用”原则，把同一个分区的日志片段保存到同一路径下。会往拥有最少数据分区的路径新增分区。

num.recovery.threads.per.data.dir

每数据目录用于日志恢复启动和关闭时的线程数量。因为这些线程只是服务器启动（正常启动和崩溃后重启）和关闭时会用到。所以完全可以设置大量的线程来达到并行操作的目的。注意，这个参数指的是每个日志目录的线程数，比如本参数设置为 8，而 log.dirs 设置为了三个路径，则总共会启动 24 个线程。

auto.create.topics.enable

是否允许自动创建主题。如果设为 true，那么 produce（生产者往主题写消息），consume（消费者从主题读消息）或者 fetch metadata（任意客户端向主题发送元数据请求时）一个不存在的主题时，就会自动创建。缺省为 true。

delete.topic.enable=true

删除主题配置，默认未开启

主题配置

新建主题的默认参数

num.partitions

每个新建主题的分区个数（分区个数只能增加，不能减少 ）。这个参数一般要评估，比如，每秒钟要写入和读取 1000M 数据，如果现在每个消费者每秒钟可以处理 50MB 的数据，那么需要 20 个分区，这样就可以让 20 个消费者同时读取这些分区，从而达到设计目标。（一般经验，把分区大小限制在 25G 之内比较理想）

log.retention.hours

日志保存时间，默认为 7 天（168 小时）。超过这个时间会清理数据。bytes 和 minutes 无论哪个先达到都会触发。与此类似还有 log.retention.minutes 和 log.retention.ms，都设置的话，优先使用具有最小值的那个。（提示：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。也就是最后修改时间是指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）

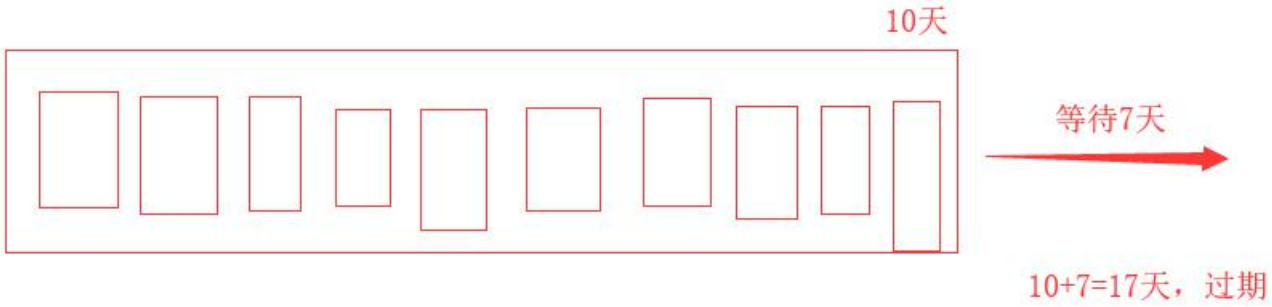
log.retention.bytes

topic 每个分区的最大文件大小，一个 topic 的大小限制 = 分区数*log.retention.bytes。-1 没有大小限制。log.retention.bytes 和 log.retention.minutes 任意一个达到要求，都会执行删除。(注意如果是 log.retention.bytes 先达到了，则是删除多出来的部分数据)，一般不推荐使用最大文件删除策略，而是推荐使用文件过期删除策略。

log.segment.bytes

分区的日志存放在某个目录下诸多文件中，这些文件将分区的日志切分成一段一段的，我们称为日志片段。这个属性就是每个文件的最大尺寸；当尺寸达到这个数值时，就会关闭当前文件，并创建新文件。被关闭的文件就开始等待过期。默认为 1G。

如果一个主题每天只接受 100MB 的消息，那么根据默认设置，需要 10 天才能填满一个文件。而且因为日志片段在关闭之前，消息是不会过期的，所以如果 log.retention.hours 保持默认值的话，那么这个日志片段需要 17 天才过期。因为关闭日志片段需要 10 天，等待过期又需要 7 天。



log.segment.ms

作用和 log.segment.bytes 类似，只不过判断依据是时间。同样的，两个参数，以先到的为准。这个参数默认是不开启的。

message.max.bytes

表示一个服务器能够接收处理的消息的最大字节数，注意这个值 producer 和 consumer 必须设置一致，且不要大于 fetch.message.max.bytes 属性的值(消费者能读取的最大消息,这个值应该大于或等于 message.max.bytes)。该值默认是 1000000 字节，大概 900KB~1MB。如果启动压缩，判断压缩后的值。这个值的大小对性能影响很大，值越大，网络和 IO 的时间越长，还会增加磁盘写入的大小。

Kafka 设计的初衷是迅速处理短小的消息，一般 10K 大小的消息吞吐性能最好（LinkedIn 的 kafka 性能测试）

硬件配置对 Kafka 性能的影响

为 Kafka 选择合适的硬件更像是一门艺术，就跟它的名字一样，我们分别从磁盘、内存、网络和 CPU 上来分析，确定了这些关注点，就可以在预算范围之内选择最优的硬件配置。

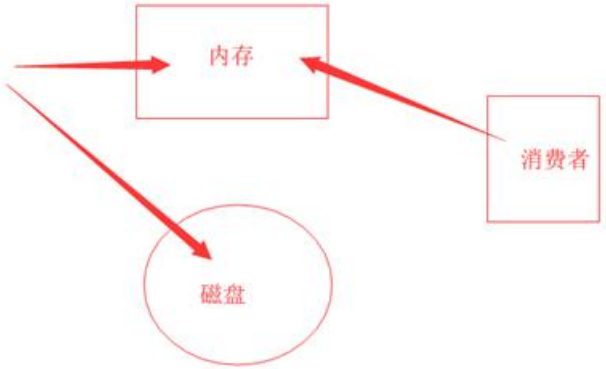
磁盘吞吐量/磁盘容量

磁盘吞吐量（IOPS 每秒的读写次数）会影响生产者的性能。因为生产者的消息必须被提交到服务器保存，大多数的客户端都会一直等待，直到至少有一个服务器确认消息已经成功提交为止。也就是说，磁盘写入速度越快，生成消息的延迟就越低。（SSD 固态贵单个速度快，HDD 机械偏移可以多买几个，设置多个目录加快速度，具体情况具体分析）

磁盘容量的大小，则主要看需要保存的消息数量。如果每天收到 1TB 的数据，并保留 7 天，那么磁盘就需要 7TB 的数据。

内存

Kafka 本身并不需要太大内存，内存则主要是影响消费者性能。在大多数业务情况下，消费者消费的数据一般会从内存（页面缓存，从系统内存中分）中获取，这比在磁盘上读取肯定要快的多。一般来说运行 Kafka 的 JVM 不需要太多的内存，剩余的系统内存可以作为页面缓存，或者用来缓存正在使用的日志片段，所以我们一般 Kafka 不会同其他的重要应用系统部署在一台服务器上，因为他们需要共享页面缓存，这个会降低 Kafka 消费者的性能。



网络

网络吞吐量决定了 Kafka 能够处理的最大数据流量。它和磁盘是制约 Kafka 拓展规模的主要因素。对于生产者、消费者写入数据和读取数据都要瓜分网络流量。同时做集群复制也非常消耗网络。

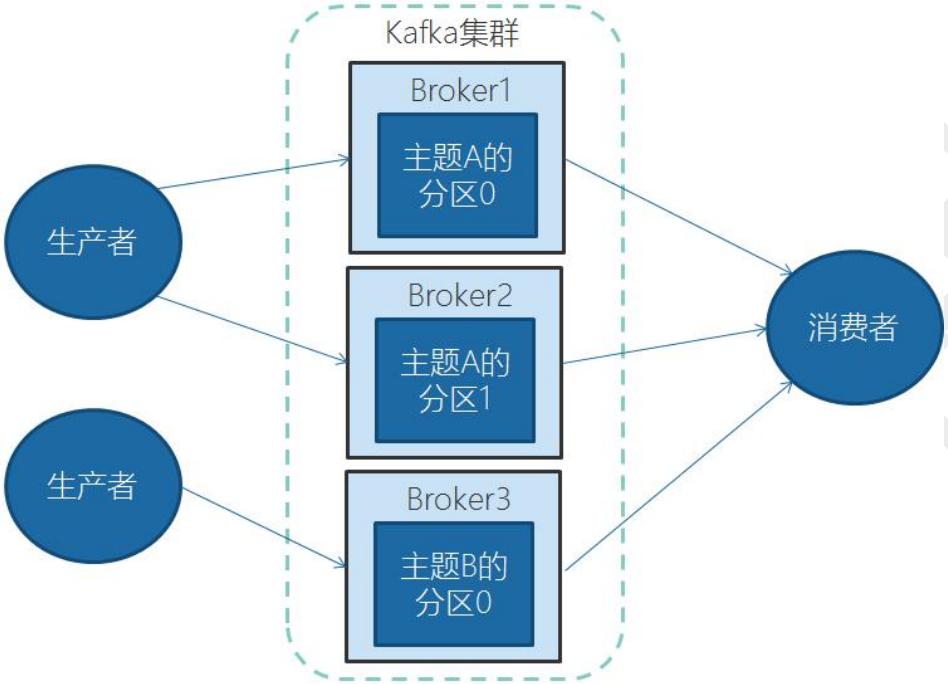
CPU

Kafka 对 cpu 的要求不高，主要是用在对消息解压和压缩上。所以 cpu 的性能不是在使用 Kafka 的首要考虑因素。

总结

我们要为 Kafka 选择合适的硬件时，优先考虑存储，包括存储的大小，然后考虑生产者的性能（也就是磁盘的吞吐量），选好存储以后，再来选择 CPU 和内存就容易得多。网络的选择要根据业务上的情况来定，也是非常重要的一环。

Kafka 的集群



为何需要 Kafka 集群

本地开发，一台 Kafka 足够使用。在实际生产中，集群可以跨服务器进行负载均衡，再则可以使用复制功能来避免单独故障造成的数据丢失。同时集群可以提供高可用性。

如何估算 Kafka 集群中 Broker 的数量

要估量以下几个因素：

需要多少磁盘空间保留数据，和每个 broker 上有多少空间可以用。比如，如果一个集群有 10TB 的数据需要保留，而每个 broker 可以存储 2TB，那么至少需要 5 个 broker。如果启用了数据复制，则还需要一倍的空间，那么这个集群需要 10 个 broker。

集群处理请求的能力。如果因为磁盘吞吐量和内存不足造成性能问题，可以通过扩展 broker 来解决。

Broker 如何加入 Kafka 集群

非常简单，只需要两个参数。第一，配置 zookeeper.connect，第二，为新增的 broker 设置一个集群内的唯一性 id。

Kafka 中的集群是可以动态扩容的。

第一个 Kafka 程序

创建我们的主题

kafka-topics.bat --zookeeper localhost:2181/kafka --create --topic hello-kafka --replication-factor 1 --partitions 4

生产者发送消息

我们这里使用 Kafka 内置的客户端 API 开发 kafka 应用程序。因为我们是 Java 程序员，所以这里我们使用 Maven，使用最新版本

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

```
    <version>2.3.0</version>
</dependency>
```

生产者代码示例如下

```
public static void main(String[] args) {
    //TODO 生产者三个属性必须指定(broker地址清单、key和value的序列化器)
    Properties properties = new Properties();
    properties.put("bootstrap.servers","127.0.0.1:9092");
    properties.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
    properties.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
    KafkaProducer<String,String> producer = new KafkaProducer<~>(properties);
    try {
        ProducerRecord<String,String> record;
        try {
            //TODO发送4条消息
            for(int i=0;i<4;i++){
                record = new ProducerRecord<String,String>(BusiConst.HELLO_TOPIC, String.valueOf(i), value: "lison");
                producer.send(record);
                System.out.println(i+", message is sent");
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    } finally {
        producer.close();
    }
```

必选属性

创建生产者对象时有三个属性必须指定。

bootstrap.servers

该属性指定 broker 的地址清单，地址的格式为 host:port。清单里不需要包含所有的 broker 地址，生产者会从给定的 broker 里查询其他 broker 的信息。不过最少提供 2 个 broker 的信息(用逗号分隔，比如: 127.0.0.1:9092,192.168.0.13:9092)，一旦其中一个宕机，生产者仍能连接到集群上。

key.serializer

生产者接口允许使用参数化类型，可以把 Java 对象作为键和值传 broker，但是 broker 希望收到的消息的键和值都是字节数组，所以，必须提供将对象序列化成字节数组的序列化器。key.serializer 必须设置为实现 org.apache.kafka.common.serialization.Serializer 的接口类，Kafka 的客户端默认提供了 ByteArraySerializer,IntegerSerializer, StringSerializer，也可以实现自定义的序列化器。

value.serializer

同 key.serializer。

参见代码，模块 kafka-no-spring 下包 hellokafka 中

消费者接受消息

消费者代码示例如下（Kafka 只提供拉取的方式）

```java
public static void main(String[] args) {
    //TODO 消费者三个属性必须指定(broker地址清单、key和value的反序列化器)
    Properties properties = new Properties();
    properties.put("bootstrap.servers","127.0.0.1:9092");
    properties.put("key.deserializer", StringDeserializer.class);
    properties.put("value.deserializer", StringDeserializer.class);
    //TODO 群组并非完全必须
    properties.put(ConsumerConfig.GROUP_ID_CONFIG,"test1");
    KafkaConsumer<String,String> consumer = new KafkaConsumer<~>(properties);
    try {
        //TODO 消费者订阅主题（可以多个）
        consumer.subscribe(Collections.singletonList(BusiConst.HELLO_TOPIC));
        while(true){
            //TODO 拉取（新版本）
            ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(500));
            for(ConsumerRecord<String, String> record:records){
                System.out.println(String.format("topic:%s,分区：%d,偏移量：%d," + "key:%s,value:%s",record.topic(),record.partition(),
                        record.offset(),record.key(),record.value()));
                //do my work
                //打包任务投入线程池
            }
        }
    } finally {
        consumer.close();
    }
```

必选参数

bootstrap.servers、key.serializer、value.serializer 含义同生产者

group.id

并非完全必需，它指定了消费者属于哪一个群组，但是创建不属于任何一个群组的消费者并没有问题。

参见代码，模块 kafka-no-spring 下包 hellokafka 中

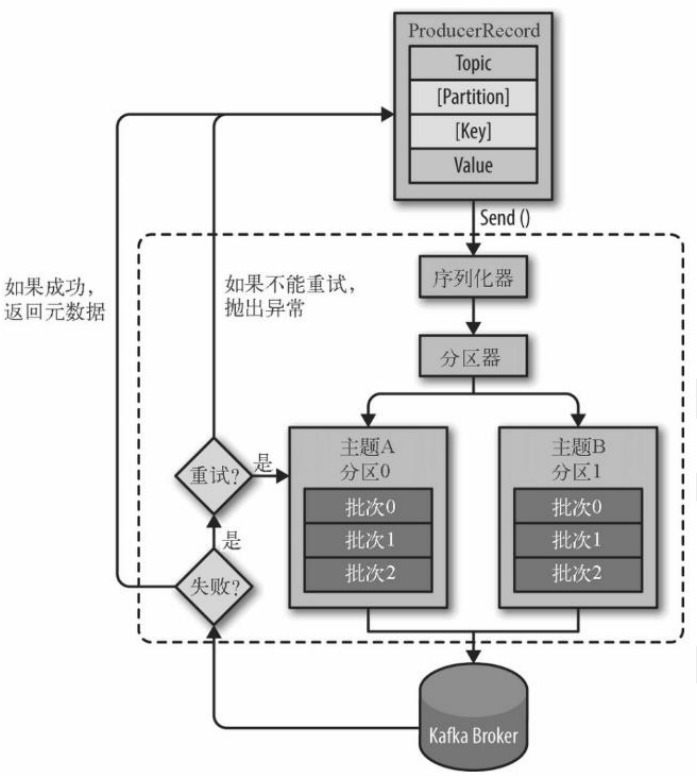
新版本特点：poll(Duration)这个版本修改了这样的设计，会把元数据获取也计入整个超时时间（更加的合理）

演示示例

1. 默认创建主题，只有一个分区时，演示生产者和消费者情况。
2. 修改主题分区为 2（使用管理命令），再重新演示生产者和消费者情况。

Kafka 的生产者

生产者发送消息的基本流程



从创建一个 ProducerRecord 对象开始， Producer Record 对象需要包含目标主题和要发送的内容。我们还可以指定键或分区。在发送 ProducerReco rd 对象时，生产者要先把键和值对象序列化成字节数组，这样它们才能够在网络上传输。

接下来，数据被传给分区器。如果之前在 Producer Record 对象里指定了分区，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 Producer Record 对象的键来选择一个分区。选好分区以后，生产者就知道该往哪个主题和分区发送这条记录了。紧接着，这条记录被添加到一个记录批次里（双端队列，尾部写入），这个批次里的所有消息会被发送到相同的主题和分区上。有一个独立的线程负责把这些记录批次发送到相应的 broker 上。

服务器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，以及记录在分区里的偏移量。如果写入失败， 则会返回一个错误。生产者在收到错误之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。

生产者发送消息一般会发生两类错误：

一类是可重试错误，比如连接错误（可通过再次建立连接解决）、无主 no leader（可通过分区重新选举首领解决）。

另一类是无法通过重试解决，比如“消息太大”异常，具体见 message.max.bytes，这类消息不会进行任何重试，直接抛出异常

使用 Kafka 生产者

三种发送方式

我们通过生成者的 send 方法进行发送。send 方法会返回一个包含 RecordMetadata 的 Future 对象。RecordMetadata 里包含了目标主题，分区信息和消息的偏移量。

发送并忘记

忽略 send 方法的返回值，不做任何处理。大多数情况下，消息会正常到达，而且生产者会自动重试，但有时会丢失消息。

同步发送

获得 send 方法返回的 Future 对象，在合适的时候调用 Future 的 get 方法。参见代码，模块 kafka-no-spring 下包 sendtype 中。

异步发送

实现接口 org.apache.kafka.clients.producer.Callback，然后将实现类的实例作为参数传递给 send 方法。参见代码，模块 kafka-no-spring 下包 sendtype 中。

多线程下的生产者

KafkaProducer 的实现是线程安全的，所以我们可以在多线程的环境下，安全的使用 KafkaProducer 的实例，如何节约资源的使用呢？参见代码，模块 kafka-no-spring 下包 concurrent 中

更多发送配置

生产者有很多属性可以设置，大部分都有合理的默认值，无需调整。有些参数可能对内存使用，性能和可靠性方面有较大影响。可以参考 org.apache.kafka.clients.producer 包下的 ProducerConfig 类。代码见模块 kafka-no-spring 下包 ProducerConfig 中 ConfigKafkaProducer 类

acks：

Kafk 内部的复制机制是比较复杂的，这里不谈论内部机制（后续章节进行细讲），我们只讨论生产者发送消息时与副本的关系。

指定了必须要有多少个分区副本收到消息，生产者才会认为写入消息是成功的，这个参数对消息丢失的可能性有重大影响。

acks=0：生产者在写入消息之前不会等待任 何来自服务器的响应，容易丢消息，但是吞吐量高。

acks=1：只要集群的首领节点收到消息，生产者会收到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，新首领没有选举出来），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。不过，如果一个没有收到消息的节点成为新首领，消息还是会丢失。默认使用这个配置。

acks=all：只有当所有参与复制的节点都收到消息，生产者才会收到一个来自服务器的成功响应。延迟高。

金融业务，主备外加异地灾备。所以很多高可用场景一般不是设置 2 个副本，有可能达到 5 个副本，不同机架上部署不同的副本，异地上也部署一套副本。

buffer.memory

设置生产者内存缓冲区的大小（结合生产者发送消息的基本流程），生产者用它缓冲要发送到服务器的消息。如果数据产生速度大于向 broker 发送的速度，导致生产者空间不足，producer 会阻塞或者抛出异常。缺省 33554432 (32M)

max.block.ms

指定了在调用 send()方法或者使用 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，或者没有可用的元数据时，这些方法就会阻塞。在阻塞时间达到 max.block.ms 时，生产者会抛出超时异常。缺省 60000ms

retries

发送失败时，指定生产者可以重发消息的次数（缺省 Integer.MAX_VALUE）。默认情况下，生产者在每次重试之间等待 100ms，可以通过参数 retry.backoff.ms 参数来改变这个时间间隔。

receive.buffer.bytes 和 send.buffer.bytes

指定 TCP socket 接受和发送数据包的缓存区大小。如果它们被设置为-1，则使用操作系统的默认值。如果生产者或消费者处在不同的数据中心，那么可以适当增大这些值，因为跨数据中心的网络一般都有比较高的延迟和比较低的带宽。缺省 102400

batch.size

当多个消息被发送同一个分区时，生产者会把它们放在同一个批次里。该参数指定了一个批次可以使用的内存大小，按照字节数计算。当批次内存被填满后，批次里的所有消息会被发送出去。但是生产者不一定都会等到批次被填满才发送，半满甚至只包含一个消息的批次也有可能被发送。缺省 16384(16k) ，如果一条消息超过了批次的大小，会写不进去。

linger.ms

指定了生产者在发送批次前等待更多消息加入批次的时间。它和 batch.size 以先到者为先。也就是说，一旦我们获得消息的数量够 batch.size 的数量了，他将会立即发送而不顾这项设置，然而如果我们获得消息字节数比 batch.size 设置要小的多，我们需要“linger”特定的时间以获取更多的消息。这个设置默认为 0，即没有延迟。设定 linger.ms=5，例如，将会减少请求数目，但是同时会增加 5ms 的延迟，但也会提升消息的吞吐量。

compression.type

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。snappy 占用 cpu 少，提供较好的性能和可观的压缩比，如果比较关注性能和网络带宽，用这个。如果带宽紧张，用 gzip，会占用较多的 cpu，但提供更高的压缩比。

client.id

当向 server 发出请求时，这个字符串会发送给 server。目的是能够追踪请求源头，以此来允许 ip/port 许可列表之外的一些应用可以发送信息。这项应用可以设置任意字符串，因为没有任何功能性的目的，除了记录和跟踪。

max.in.flight.requests.per.connection

指定了生产者在接收到服务器响应之前可以发送多个消息，值越高，占用的内存越大，当然也可以提升吞吐量。发生错误时，可能会造成数据的发送顺序改变,默认是 5 (修改）。

如果需要保证消息在一个分区上的严格顺序，这个值应该设为 1。不过这样会严重影响生产者的吞吐量。

request.timeout.ms

客户端将等待请求的响应的最大时间,如果在这个时间内没有收到响应，客户端将重发请求;超过重试次数将抛异常，默认 30 秒。

metadata.fetch.timeout.ms

是指我们所获取的一些元数据的第一个时间数据。元数据包含：topic，host，partitions。此项配置是指当等待元数据 fetch 成功完成所需要的时间，否则会跑出异常给客户端

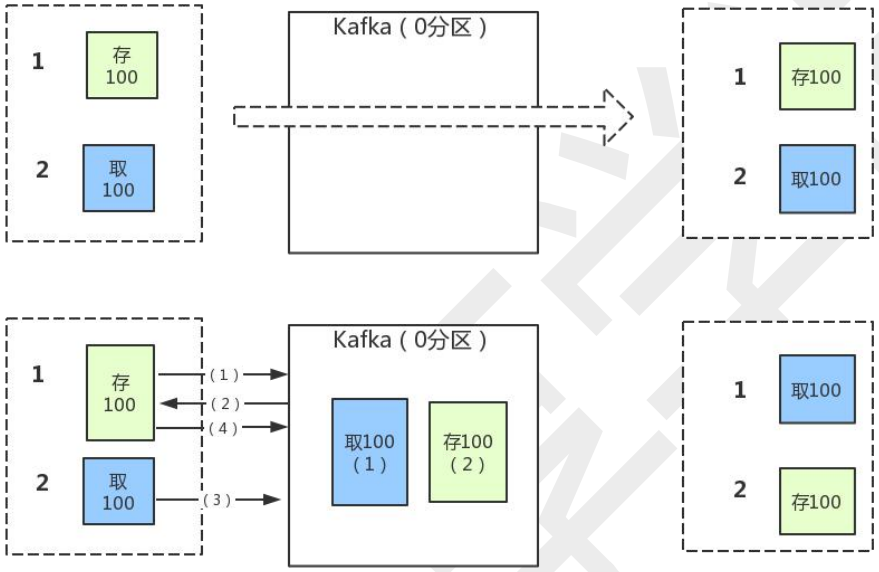
max.request.size

控制生产者发送请求最大大小。默认这个值为 1M，如果一个请求里只有一个消息，那这个消息不能大于 1M，如果一次请求是一个批次，该批次包含了 1000 条消息，那么每个消息不能大于 1KB。注意：broker 具有自己对消息记录尺寸的覆盖，如果这个尺寸小于生产者的这个设置，会导致消息被拒绝。这个参数和 Kafka 主机的 message.max.bytes 参数有关系。如果生产者发送的消息超过 message.max.bytes 设置的大小，就会被 Kafka 服务器拒绝。

以上参数不用去，一般来说，就记住 acks、batch.size、linger.ms、max.request.size 就行了，因为这 4 个参数重要些，其他参数一般没有太大必要调整。

顺序保证

Kafka 可以保证同一个分区里的消息是有序的。也就是说，发送消息时，主题只有且只有一个分区，同时生产者按照一定的顺序发送消息， broker 就会按照这个顺序把它们写入分区，消费者也会按照同样的顺序读取它们。在某些情况下， 顺序是非常重要的。例如，往一个账户存入 100 元再取出来，这个与先取钱再存钱是截然不同的！不过，有些场景对顺序不是很敏感。



如果把 retires 设为非零整数，同时把 max.in.flight.requests.per.connection 设为比 1 大的数，那么，如果第一个批次消息写入失败，而第二个批次写入成功， broker 会重试写入第一个批次。如果此时第一个批次也写入成功，那么两个批次的顺序就反过来了。

一般来说，如果某些场景要求消息是有序的，那么消息是否写入成功也是很关键的，所以不建议把 retires 设为 0(不重试的话消息可能会因为连接关闭等原因会丢) 。所以还是需要重试，同时把 max.in.flight.request.per.connection 设为 1，这样在生产者尝试发送第一批消息时，就不会有其他的消息发送给 broker 。不过这样会严重影响生产者的吞吐量，所以只有在对消息的顺序有严格要求的情况下才能这么做。

序列化

创建生产者对象必须指定序列化器，默认的序列化器并不能满足我们所有的场景。我们完全可以自定义序列化器。只要实现 org.apache.kafka.common.serialization.Serializer 接口即可。

如何实现，看模块 kafka-no-spring 下包 selfserial 中代码。


```
public class DemoUser {
    private int id;
    private String name;
```

自定义序列化需要考虑的问题

自定义序列化容易导致程序的脆弱性。举例，在我们上面的实现里，我们有多种类型的消费者，每个消费者对实体字段都有各自的需求，比如，有的将字段变更为 long 型，有的会增加字段，这样会出现新旧消息的兼容性问题。特别是在系统升级的时候，经常会出现一部分系统升级，其余系统被迫跟着升级的情况。

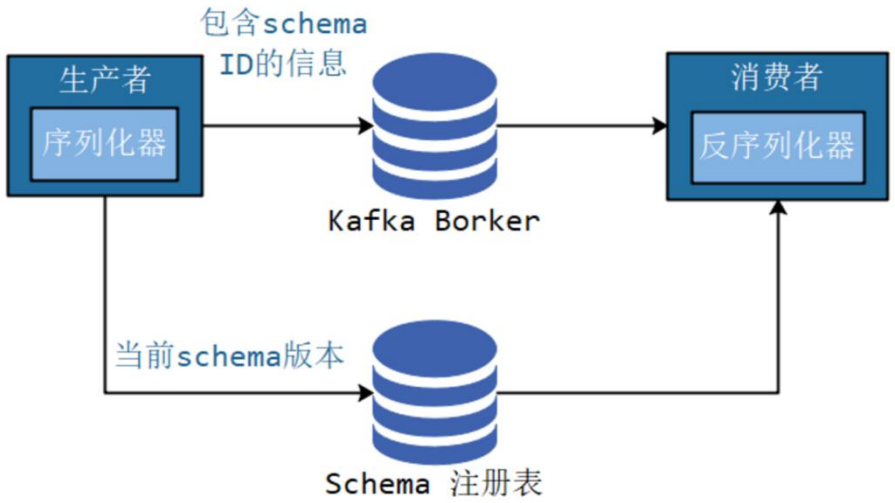
解决这个问题，可以考虑使用自带格式描述以及语言无关的序列化框架。比如 Protobuf，或者 Kafka 官方推荐的 Apache Avro。

Avro 会使用一个 JSON 文件作为 schema 来描述数据，Avro 在读写时会用到这个 schema，可以把这个 schema 内嵌在数据文件中。这样，不管数据格式如何变动，消费者都知道如何处理数据。

但是内嵌的消息，自带格式，会导致消息的大小不必要的增大，消耗了资源。我们可以使用 schema 注册表机制，将所有写入的数据用到的 schema 保存在注册表中，然后在消息中引用 schema 的标识符，而读取的数据的消费者程序使用这个标识符从注册表中拉取 schema 来反序列化记录。

注意：Kafka 本身并不提供 schema 注册表，需要借助第三方，现在已经有很多的开源实现，比如 Confluent Schema Registry，可以从 GitHub 上获取。如何使用参考如下网址：

https://cloud.tencent.com/developer/article/1336568



不过一般除非你使用 Kafka 需要关联的团队比较大，敏捷开发团队才会使用，一般的团队用不上。对于一般的情况使用 JSON 足够了。

分区

我们在新增 ProducerRecord 对象中可以看到，ProducerRecord 包含了目标主题，键和值，Kafka 的消息都是一个个的键值对。键可以设置为默认的 null。

键的主要用途有两个：一，用来决定消息被写往主题的哪个分区，拥有相同键的消息将被写往同一个分区，二，还可以作为消息的附加消息。

如果键值为 null，并且使用默认的分区器，分区器使用轮询算法将消息均衡地分布到各个分区上。

如果键不为空，并且使用默认的分区器，Kafka 对键进行散列（Kafka 自定义的散列算法，具体算法原理不知），然后根据散列值把消息映射到特定的分区上。很明显，同一个键总是被映射到同一个分区。但是只有不改变主题分区数量的情况下，键和分区之间的映射才能保持不变，一旦增加了新的分区，就无法保证了，所以如果要使用键来映射分区，那就要在创建主题的时候把分区规划好，而且永远不要增加新分区。

自定义分区器

某些情况下，数据特性决定了需要进行特殊分区，比如电商业务，北京的业务量明显比较大，占据了总业务量的 20%，我们需要对北京的订单进行单独分区处理，默认的散列分区算法不合适了， 我们就可以自定义分区算法，对北京的订单单独处理，其他地区沿用散列分区算法。或者某些情况下，我们用 value 来进行分区。

具体实现，先创建一个 4 分区的主题，然后观察模块 kafka-no-spring 下包 SelfPartitionProducer 中代码。

Kafka 的消费者

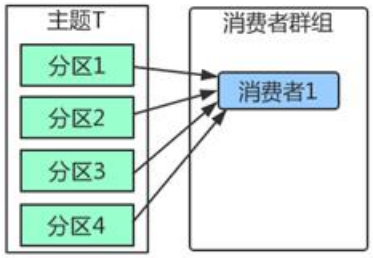
消费者的入门

消费者的含义，同一般消息中间件中消费者的概念。在高并发的情况下，生产者产生消息的速度是远大于消费者消费的速度，单个消费者很可能会负担不起，此时有必要对消费者进行横向伸缩，于是我们可以使用多个消费者从同一个主题读取消息，对消息进行分流。

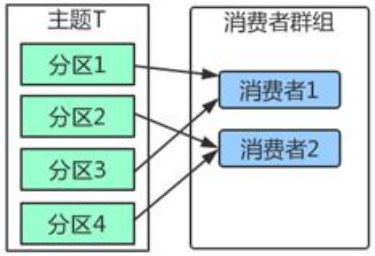
（买单的故事，群组，消费者的一群人， 消费者：买单的，分区：一笔单，一笔单能被买单一次，当然一个消费者可以买多个单，如果有一个消费者挂掉了<跑单了>，另外的消费者接上）

消费者群组

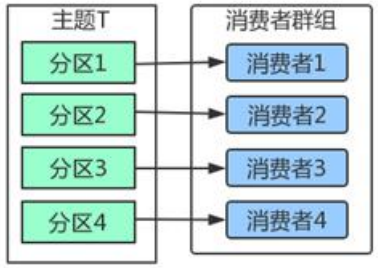
Kafka 里消费者从属于消费者群组，一个群组里的消费者订阅的都是同一个主题，每个消费者接收主题一部分分区的消息。



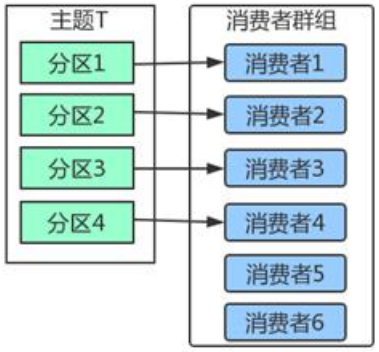
如上图，主题 T 有 4 个分区，群组中只有一个消费者，则该消费者将收到主题 T1 全部 4 个分区的消息。



如上图，在群组中增加一个消费者 2，那么每个消费者将分别从两个分区接收消息，上图中就表现为消费者 1 接收分区 1 和分区 3 的消息，消费者 2 接收分区 2 和分区 4 的消息。



如上图，在群组中有 4 个消费者，那么每个消费者将分别从 1 个分区接收消息。

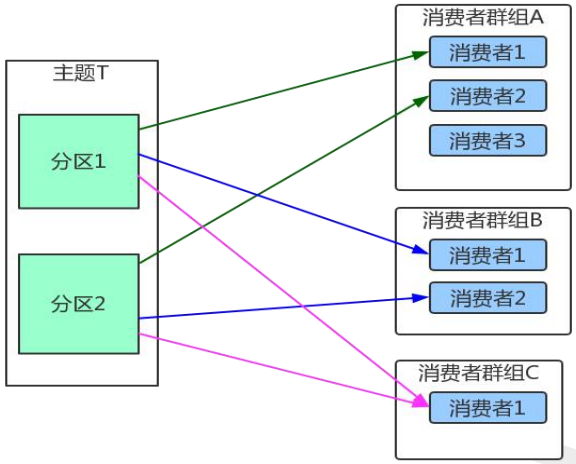


但是，当我们增加更多的消费者，超过了主题的分区数量，就会有一部分的消费者被闲置，不会接收到任何消息。

往消费者群组里增加消费者是进行横向伸缩能力的主要方式。所以我们有必要为主题设定合适规模的分区，在负载均衡的时候可以加入更多的消费者。但是要记住，一个群组里消费者数量超过了主题的分区数量，多出来的消费者是没有用处的。

如果是多个应用程序，需要从同一个主题中读取数据，只要保证每个应用程序有自己的消费者群组就行了。

具体实现，先建立一个 2 分区的主题，看模块 kafka-no-spring 下包 consumergroup 中代码。



消费者配置

消费者有很多属性可以设置，大部分都有合理的默认值，无需调整。有些参数可能对内存使用，性能和可靠性方面有较大影响。可以参考 org.apache.kafka.clients.consumer 包下 ConsumerConfig 类。

auto.offset.reset

消费者在读取一个没有偏移量的分区或者偏移量无效的情况下，如何处理。默认值是 latest，从最新的记录开始读取，另一个值是 earliest，表示消费者从起始位置读取分区的记录。

注意：如果是消费者在读取一个没有偏移量的分区或者偏移量无效的情况（因消费者长时间失效，包含的偏移量记录已经过时并被删除）下，默认值是 latest 的话，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成的记录**），可以先启动生产者，再启动消费者，观察到这种情况。观察代码，在模块 kafka-no-spring 下包 hellokafka 中。

enable .auto.commit

默认值 true，表明消费者是否自动提交偏移。为了尽量避免重复数据和数据丢失，可以改为 false，自行控制何时提交。

partition.assignment.strategy

分区分配给消费者的策略。系统提供两种策略。默认为 Range。允许自定义策略。

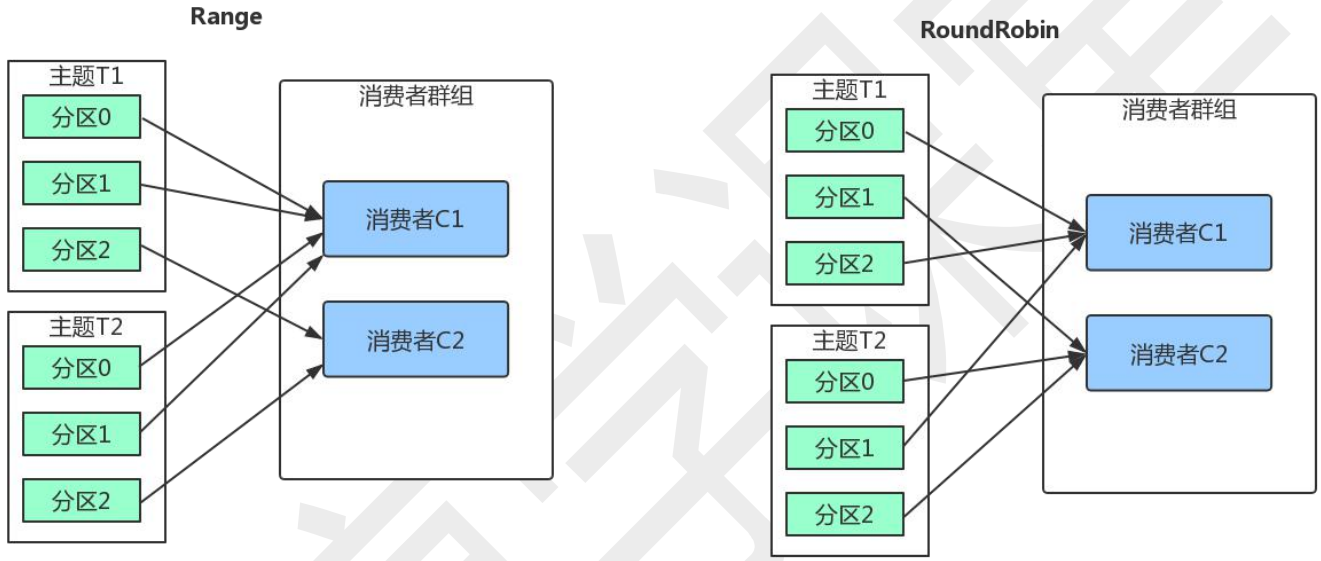
Range

把主题的连续分区分配给消费者。（如果分区数量无法被消费者整除、第一个消费者会分到更多分区）

RoundRobin

把主题的分区循环分配给消费者。

C1和C2都订阅了两个主题、C1是第一个消费者



自定义策略

extends 类 AbstractPartitionAssignor，然后在消费者端增加参数：

properties.put(ConsumerConfig.PARTITION_ASSIGNMENT_STRATEGY_CONFIG, 类.class.getName());即可。

max.poll.records

控制每次 poll 方法返回的的记录数量。

fetch.min.bytes

每次 fetch 请求时，server 应该返回的最小字节数。如果没有足够的数据返回，请求会等待，直到足够的数据才会返回。缺省为 1 个字节。多消费者下，可以设大这个值，以降低 broker 的工作负载

fetch.wait.max.ms

如果没有足够的数据能够满足 fetch.min.bytes，则此项配置是指在应答 fetch 请求之前，server 会阻塞的最大时间。缺省为 500 个毫秒。和上面的 fetch.min.bytes 结合起来，要么满足数据的大小，要么满足时间，就看哪个条件先满足。

max.partition.fetch.bytes

指定了服务器从每个分区里返回给消费者的最大字节数，默认 1MB。假设一个主题有 20 个分区和 5 个消费者，那么每个消费者至少要有 4MB 的可用内存来接收记录，而且一旦有消费者崩溃，这个内存还需更大。注意，这个参数要比服务器的 message.max.bytes 更大，否则消费者可能无法读取消息。

session.timeout.ms

如果 consumer 在这段时间内没有发送心跳信息，则它会被认为挂掉了。默认 3 秒。

client.id

当向 server 发出请求时，这个字符串会发送给 server。目的是能够追踪请求源头，以此来允许 ip/port 许可列表之外的一些应用可以发送信息。这项应用可以设置任意字符串，因为没有任何功能性的目的，除了记录和跟踪。

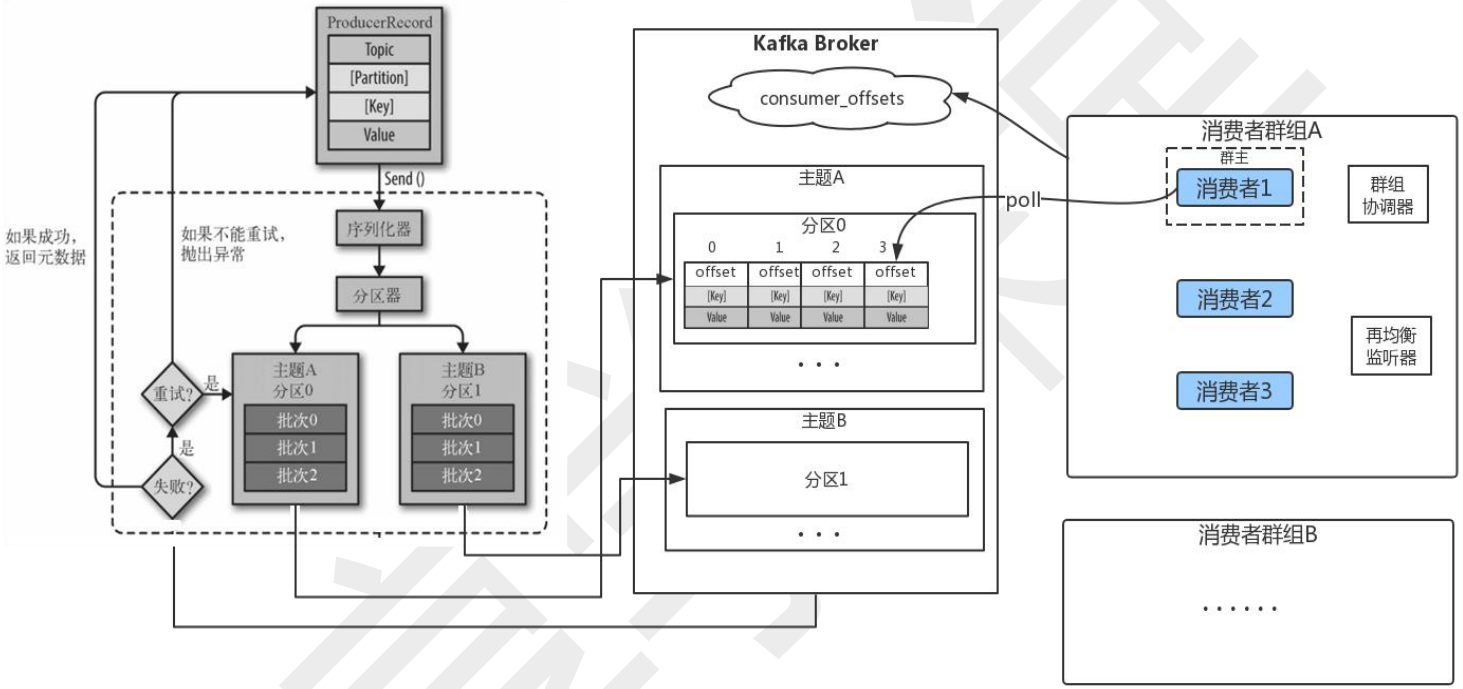
receive.buffer.bytes 和 send.buffer.bytes

指定 TCP socket 接受和发送数据包的缓存区大小。如果它们被设置为-1，则使用操作系统的默认值。如果生产者或消费者处在不同的数据中心，那么可以适当增大这些值，因为跨数据中心的网络一般都有比较高的延迟和比较低的带宽。

消费者中的基础概念

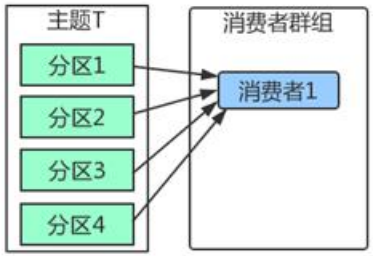
消费者的含义，同一般消息中间件中消费者的概念。在高并发的情况下，生产者产生消息的速度是远大于消费者消费的速度，单个消费者很可能会负担不起，此时有必要对消费者进行横向伸缩，于是我们可以使用多个消费者从同一个主题读取消息，对消息进行分流。

（买单的故事，群组，消费者的一群人，　消费者：买单的，分区：一笔单，一笔单能被买单一次，当然一个消费者可以买多个单，如果有一个消费者挂掉了<跑单了>，另外的消费者接上）

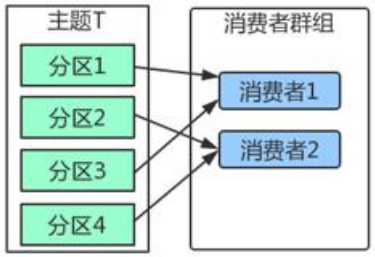


消费者群组

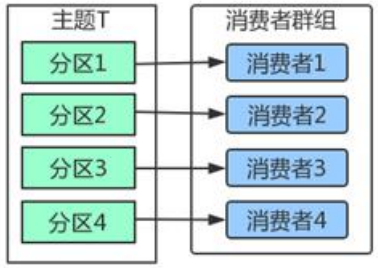
Kafka 里消费者从属于消费者群组，一个群组里的消费者订阅的都是同一个主题，每个消费者接收主题一部分分区的消息。



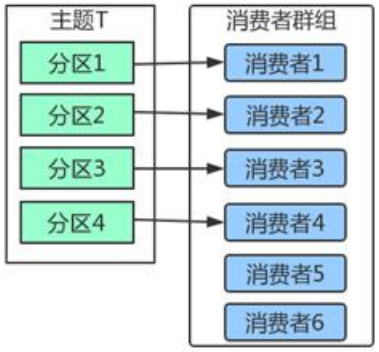
如上图，主题 T 有 4 个分区，群组中只有一个消费者，则该消费者将收到主题 T1 全部 4 个分区的消息。



如上图，在群组中增加一个消费者 2，那么每个消费者将分别从两个分区接收消息，上图中就表现为消费者 1 接收分区 1 和分区 3 的消息，消费者 2 接收分区 2 和分区 4 的消息。



如上图，在群组中有 4 个消费者，那么每个消费者将分别从 1 个分区接收消息。

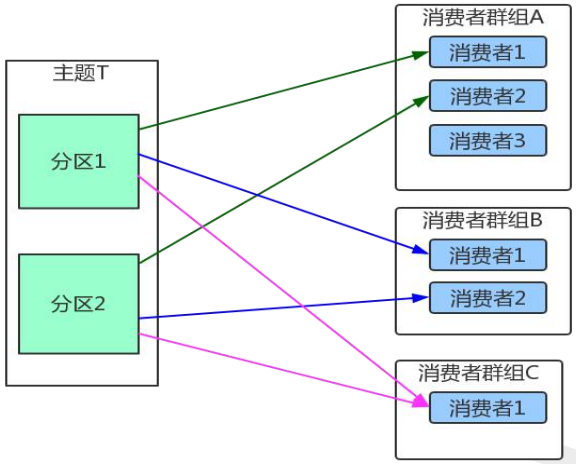


但是，当我们增加更多的消费者，超过了主题的分区数量，就会有一部分的消费者被闲置，不会接收到任何消息。

往消费者群组里增加消费者是进行横向伸缩能力的主要方式。所以我们有必要为主题设定合适规模的分区，在负载均衡的时候可以加入更多的消费者。但是要记住，一个群组里消费者数量超过了主题的分区数量，多出来的消费者是没有用处的。

如果是多个应用程序，需要从同一个主题中读取数据，只要保证每个应用程序有自己的消费者群组就行了。

具体实现，先建立一个 2 分区的主题，看模块 kafka-no-spring 下包 consumergroup 中代码。



订阅

创建消费者后，使用 subscribe()方法订阅主题，这个方法接受一个主题列表为参数，也可以接受一个正则表达式为参数；正则表达式同样也匹配多个主题。如果新创建了新主题，并且主题名字和正则表达式匹配，那么会立即触发一次再均衡，消费者就可以读取新添加的主题。比如，要订阅所有和 test 相关的主题，可以 subscribe(“tets.*”)

轮询

为了不断的获取消息，我们要在循环中不断的进行轮询，也就是不停调用 poll 方法。

poll 方法的参数为超时时间，控制 poll 方法的阻塞时间，它会让消费者在指定的毫秒数内一直等待 broker 返回数据。poll 方法将会返回一个记录（消息）列表，每一条记录都包含了记录所属的主题信息，记录所在分区信息，记录在分区里的偏移量，以及记录的键值对。

poll 方法不仅仅只是获取数据，在新消费者第一次调用时，它会负责查找群组，加入群组，接受分配的分区。如果发生了再均衡，整个过程也是在轮询期间进行的。

提交和偏移量

当我们调用 poll 方法的时候，broker 返回的是生产者写入 Kafka 但是还没有被消费者读取过的记录，消费者可以使用 Kafka 来追踪消息在分区里的位置，我们称之为**偏移量**。消费者更新自己读取到哪个消息的操作，我们称之为**提交**。

消费者是如何提交偏移量的呢？消费者会往一个叫做_consumer_offset 的特殊主题发送一个消息，里面会包括每个分区的偏移量。

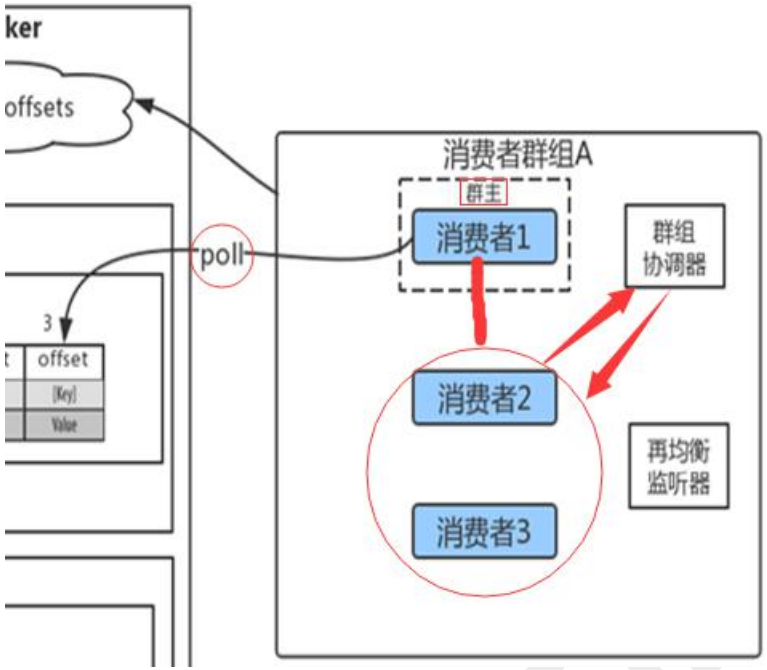
消费者中的核心概念

多线程安全问题

KafkaConsumer 的实现**不是**线程安全的，所以我们在多线程的环境下，使用 KafkaConsumer 的实例要小心，应该每个消费数据的线程拥有自己的 KafkaConsumer 实例，如何使用？参见代码，模块 kafka-no-spring 下包 concurrent 中

群组协调

消费者要加入群组时，会向群组协调器发送一个 JoinGroup 请求，第一个加入群主的消费者成为群主，群主会获得群组的成员列表，并负责给每一个消费者分配分区。分配完毕后，群主把分配情况发送给群组协调器，协调器再把这些信息发送给所有的消费者，每个消费者只能看到自己的分配信息，只有群主知道群组里所有消费者的分配信息。群组协调的工作会在消费者发生变化(新加入或者掉线)，主题中分区发生了变化（增加）时发生。



分区再均衡

当消费者群组里的消费者发生变化，或者主题里的分区发生了变化，都会导致再均衡现象的发生。从前面的知识中，我们知道，Kafka 中，存在着消费者对分区所有权的关系，

这样无论是消费者变化，比如增加了消费者，新消费者会读取原本由其他消费者读取的分区，消费者减少，原本由它负责的分区要由其他消费者来读取，增加了分区，哪个消费者来读取这个新增的分区，这些行为，都会导致分区所有权的变化，这种变化就被称为**再均衡**。

再均衡对 Kafka 很重要，这是消费者群组带来高可用性和伸缩性的关键所在。不过一般情况下，尽量减少再均衡，因为再均衡期间，消费者是无法读取消息的，会造成整个群组一小段时间的不可用。

消费者通过向称为群组协调器的 broker（不同的群组有不同的协调器）发送心跳来维持它和群组的从属关系以及对分区的所有权关系。如果消费者长时间不发送心跳，群组协调器认为它已经死亡，就会触发一次再均衡。

在 0.10.1 及以后的版本中，心跳由单独的线程负责，相关的控制参数为 max.poll.interval.ms。

Kafka 中的消费安全

一般情况下，我们调用 poll 方法的时候，broker 返回的是生产者写入 Kafka 同时 kafka 的消费者提交偏移量，这样可以确保消费者消息消费不丢失也不重复，所以一般情况下 Kafka 提供的原生的消费者是安全的，但是事情会这么完美吗？

消费者提交偏移量导致的问题

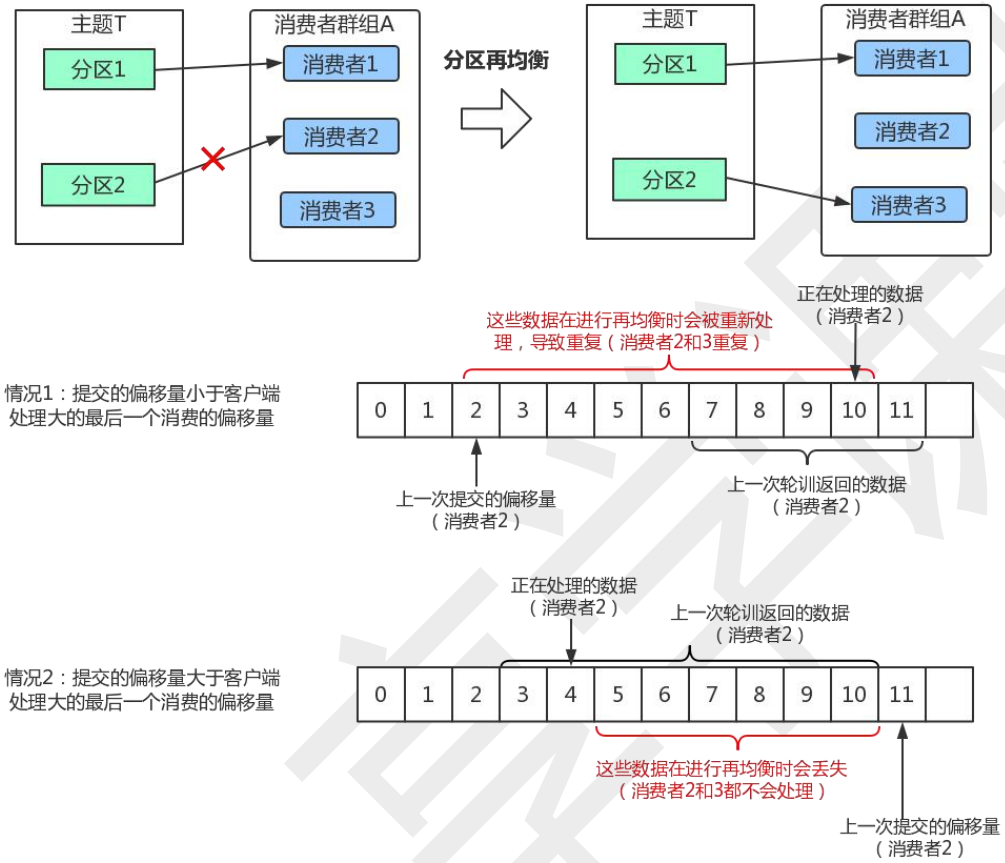
当我们调用 poll 方法的时候，broker 返回的是生产者写入 Kafka 但是还没有被消费者读取过的记录，消费者可以使用 Kafka 来追踪消息在分区里的位置，我们称之为**偏移量**。消费者更新自己读取到哪个消息的操作，我们称之为**提交**。

消费者是如何提交偏移量的呢？消费者会往一个叫做_consumer_offset 的特殊主题发送一个消息，里面会包括每个分区的偏移量。发生了再均衡之后，消费者可能会被分配新的分区，为了能够继续工作，消费者者需要读取每个分区最后一次提交的偏移量，然后从指定的地方，继续做处理。

分区再均衡的例子：某软件公司，有一个项目，有两块的工作，有两个码农，一个负责一块，干得好好的。突然一天，小王桌子一拍不干了，老子中了 5 百万了，不跟你们玩了，立马收拾完电脑就走了。然后你今天刚好入职，一个萝卜一个坑，你就入坑了。这个过程我们就好比我们的分区再均衡，分区就是一个项目中的不同块的工作，消费者就是码农，一个码农不玩了，另一个码农立马顶上，这个过程就发生了分区再均衡

1）如果提交的偏移量小于消费者实际处理的最后一个消息的偏移量，处于两个偏移量之间的消息会被重复处理，

2）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,那么处于两个偏移量之间的消息将会丢失



所以，处理偏移量的方式对客户端会有很大的影响 。KafkaConsumer API 提供了很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.comnit 被设为 true，消费者会自动把从 poll()方法接收到的**最大**偏移量提交上去。提交时间间隔由 auto.commit.interval.ms 控制，默认值是 5s。自动提交是在轮询里进行的，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。

不过,在使用这种简便的方式之前,需要知道它将会带来怎样的结果。

假设我们仍然使用默认的 5s 提交时间间隔，在最近一次提交之后的 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，不过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最大偏移量提交上去,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经处理完毕(enable.auto.comnit 被设为 true 时，在调用 close()方法之前也会进行自动提交)。一般情况下不会有什么问题,不过在处理异常或提前退出轮询时要格外小心。

自动提交虽然方便,但是很明显是一种基于时间提交的方式,不过并没有为我们留有余地来避免重复处理消息。

手动提交（同步）

我们通过控制偏移量提交时间来消除丢失消息的可能性，并在发生再均衡时减少重复消息的数量。消费者 API 提供了另一种提交偏移量的方式，开发者可以在必要的时候提交当前偏移量,而不是基于时间间隔。

把 auto.commit. offset 设为 false，自行决定何时提交偏移量。使用 commitsync()提交偏移量最简单也最可靠。这个方法会提交由 poll()方法返回的最新偏移量，提交成功后马上返回,如果提交失败就抛出异常。

注意： commitsync()将会提交由 poll()返回的最新偏移量,所以在处理完所有记录后要确保调用了 commitsync()，否则还是会有丢失消息的风险。如果发生了再均衡,从最近批消息到发生再均衡之间的所有消息都将被重复处理。

只要没有发生不可恢复的错误，commitSync（）方法会阻塞，会一直尝试直至提交成功，如果失败，也只能记录异常日志。

具体使用，参见模块 kafka-no-spring 下包 commit 包中代码 CommitSync。

异步提交

手动提交时，在 broker 对提交请求作出回应之前，应用程序会一直阻塞。这时我们可以使用异步提交 API，我们只管发送提交请求，无需等待 broker 的响应。

具体使用，参见模块 kafka-no-spring 下包 commit 包中代码。

在成功提交或碰到无法恢复的错误之前, commitsync()会一直重试,但是 commitAsync 不会。它之所以不进行重试,是因为在它收到服务器响应的时候,可能有一个更大的偏移量已经提交成功。

假设我们发出一个请求用于提交偏移量 2000,,这个时候发生了短暂的通信问题,服务器收不到请求,自然也不会作出任何响应。与此同时,我们处理了另外一批消息,并成功提交了偏移量 3000。如果 commitAsync()重新尝试提交偏移量 2000,它有可能在偏移量 3000 之后提交成功。这个时候如果发生再均衡,就会出现重复消息。

commitAsync()也支持回调,在 broker 作出响应时会执行回调。回调经常被用于记录提交错误或生成度量指标。

回调具体使用，参见模块 kafka-no-spring 下包 commit 包中代码 CommitAsync。

同步和异步组合

因为同步提交一定会成功、异步可能会失败，所以一般的场景是同步和异步一起来做。

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 再均衡前的最后一次提交,就要确保能够提交成功。

因此,在消费者关闭前一般会组合使用 commitAsync()和 commitsync()。具体使用，参见模块 kafka-no-spring 下包 commit 包中代码 SyncAndAsync。

特定提交

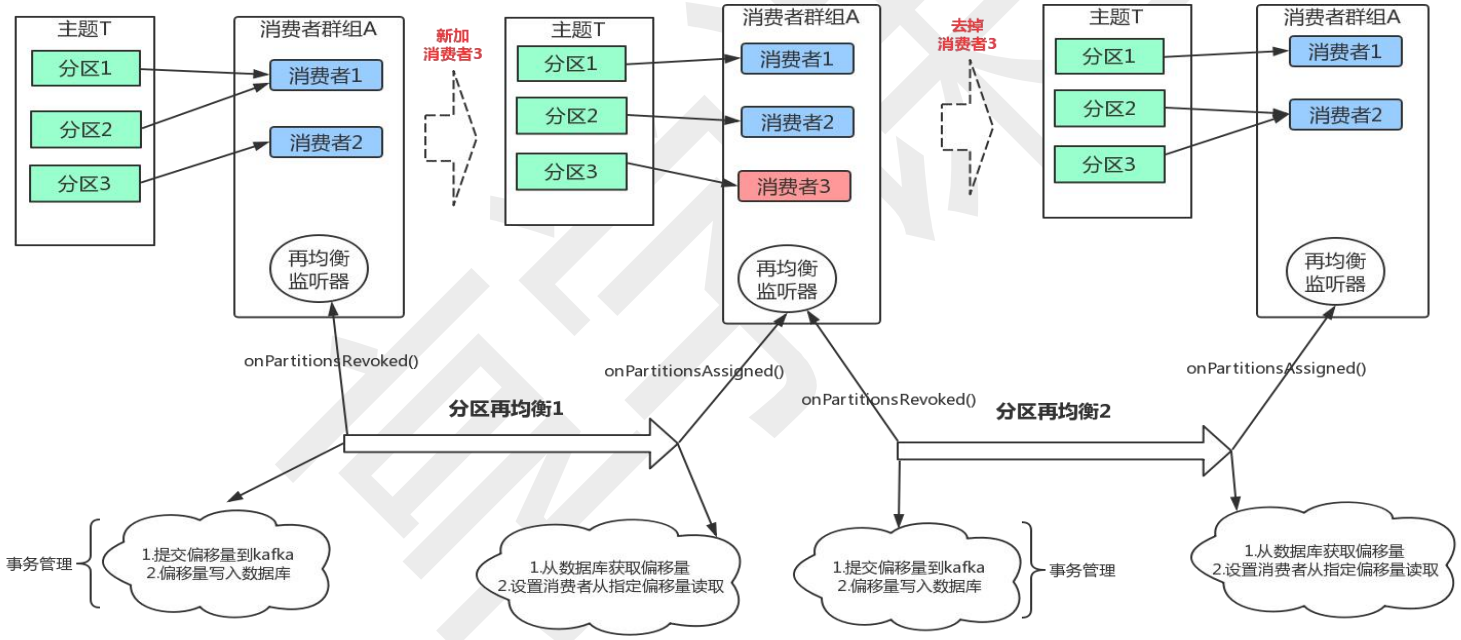
在我们前面的提交中，提交偏移量的频率与处理消息批次的频率是一样的。但如果想要更频繁地提交该怎么办?

如果 poll()方法返回一大批数据,为了避免因再均衡引起的重复处理整批消息,想要在批次中间提交偏移量该怎么办?这种情况无法通过调用 commitSync()或 commitAsync()来实现，因为它们只会提交最后一个偏移量,而此时该批次里的消息还没有处理完。

消费者 API 允许在调用 commitsync()和 commitAsync()方法时传进去希望提交的分区和偏移量的 map。假设我们处理了半个批次的消息,最后一个来自主题“customers”，分区 3 的消息的偏移量是 5000，你可以调用 commitsync()方法来提交它。不过，因为消费者可能不只读取一个分区,因为我们需要跟踪所有分区的偏移量,所以在这个层面上控制偏移量的提交会让代码变复杂。

具体使用，参见模块 kafka-no-spring 下包 commit 包中代码 CommitSpecial。

分区再均衡



再均衡监听器

在提交偏移量一节中提到过,消费者在退出和进行分区再均衡之前,会做一些清理工作比如，提交偏移量、关闭文件句柄、数据库连接等。

在为消费者分配新分区或移除旧分区时,可以通过消费者 API 执行一些应用程序代码，在调用 subscribe()方法时传进去一个 ConsumerRebalancelistener 实例就可以了。

ConsumerRebalancelistener 有两个需要实现的方法。

1) public void onPartitionsRevoked(Collection< TopicPartition> partitions)方法会在

再均衡开始之前和消费者停止读取消息之后被调用。如果在这里提交偏移量，下一个接管分区的消费者就知道该从哪里开始读取了

2) public void onPartitionsAssigned(Collection< TopicPartition> partitions)方法会在重新分配分区之后和消费者开始读取消息之前被调用。

具体使用，我们先创建一个 3 分区的主题，然后实验一下，参见模块 kafka-no-spring 下 rebalance 包中代码。

从特定偏移量处开始记录

到目前为止,我们知道了如何使用 poll()方法从各个分区的最新偏移量处开始处理消息。

不过,有时候我们也需要从特定的偏移量处开始读取消息。

如果想从分区的起始位置开始读取消息,或者直接跳到分区的末尾开始读取消息,可以使 seekToBeginning(Collection<TopicPartition> tp)和 seekToEnd(Collection<TopicPartition>tp)这两个方法。

不过,Kaka 也为我们提供了用于查找特定偏移量的 API。它有很多用途,比如向后回退几个消息或者向前跳过几个消息(对时间比较敏感的应用程序在处理滞后的情况下希望能够向前跳过若干个消息)。在使用 Kafka 以外的系统来存储偏移量时,它将给我们带来更大的惊喜--让消息的业务处理和偏移量的提交变得一致。

试想一下这样的场景:应用程序从 Kaka 读取事件(可能是网站的用户点击事件流),对它们进行处理(可能是使用自动程序清理点击操作并添加会话信息),然后把结果保存到数据库。假设我们真的不想丢失任何数据,也不想在数据库里多次保存相同的结果。

我们可能会，每处理一条记录就提交一次偏移量。尽管如此,在记录被保存到数据库之后以及偏移量被提交之前,应用程序仍然有可能发生崩溃,导致重复处理数据,数据库里就会出现重复记录。

如果保存记录和偏移量可以在一个原子操作里完成,就可以避免出现上述情况。记录和偏移量要么都被成功提交,要么都不提交。如果记录是保存在数据库里而偏移量是提交到Kafka上,那么就无法实现原子操作不过,如果在同一个事务里把记录和偏移量都写到数据库里会怎样呢?那么我们就会知道记录和偏移量要么都成功提交,要么都没有,然后重新处理记录。

现在的问题是:如果偏移量是保存在数据库里而不是 Kafka 里,那么消费者在得到新分区时怎么知道该从哪里开始读取?这个时候可以使用 seek()方法。在消费者启动或分配到新分区时,可以使用 seck()方法查找保存在数据库里的偏移量。我们可以使用使用 Consumer Rebalancelistener 和 seek()方法确保我们是从数据库里保存的偏移量所指定的位置开始处理消息的。

具体使用，参见模块 kafka-no-spring 下包 rebalance 包中代码。

优雅退出

如果确定要退出循环,需要通过另一个线程调用 consumer. wakeup()方法。如果循环运行在主线程里,可以在 ShutdownHook 里调用该方法。要记住,consumer. wakeup()是消费者唯一一个可以从其他线程里安全调用的方法。调用 consumer. wakeup()可以退出 poll(),并抛出 WakeupException 异常。我们不需要处理 Wakeup Exception,因为它只是用于跳出循环的一种方式。不过,在退出线程之前调用 consumer.close()是很有必要的,它会提交任何还没有提交的东西,并向群组协调器发送消息,告知自己要离开群组,接下来就会触发再均衡,而不需要等待会话超时。

反序列化

不过就是序列化过程的一个反向，原理和实现可以参考生产者端的实现，同样也可以自定义反序列化器。

独立消费者

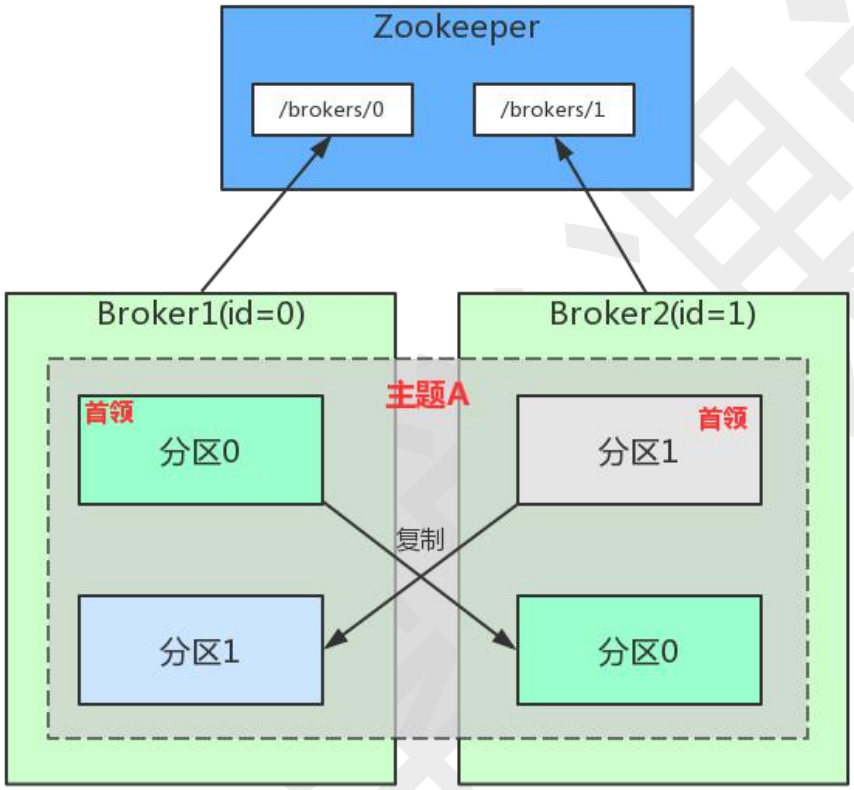
到目前为止,我们讨论了消费者群组,分区被自动分配给群组里的消费者,在群组里新增或移除消费者时自动触发再均衡。不过有时候可能只需要一个消费者从一个主题的所有分区或者某个特定的分区读取数据。这个时候就不需要消费者群组和再均衡了,只需要把主题或者分区分配给消费者,然后开始读取消息并提交偏移量。

如果是这样的话,就不需要订阅主题,取而代之的是为自己分配分区。一个消费者可以订阅主题(并加入消费者群组),或者为自己分配分区,但不能同时做这两件事情。

独立消费者相当于自己来分配分区，但是这样做的好处是自己控制，但是就没有动态的支持了，包括加入消费者（分区再均衡之类的），新增分区，这些都需要代码中去解决，所以一般情况下不推荐使用。

具体使用，参见模块 kafka-no-spring 下包 independconsumer 中代码。

深入理解 Kafka


Zookeeper
/brokers/0
/brokers/1
Broker1(id=0)
Broker2(id=1)
主题A
首领
分区0
首领
分区1
复制
分区1
分区0

集群的成员关系

Kafka 使用 zookeeper 来维护集群成员的信息。每个 broker 都有个唯一标识符，这个标识符可以在配置文件里指定，也可以自动生成。 在 broker 启动的时候，它通过创建临时节点把自己的 ID 注册到 zoo-keeper。 Kafka 组件订阅 Zookeeper 的/brokers/ids 路径(broker 在 zookeeper 上的注册路径)，当有 broker 加入集群或退出集群时，这些组件就可以获得通知 。

如果你要启动另一个具有相同 ID 的 broker，会得到一个错误。新 broker 会试着进行注册,但不会成功，因为 zookeeper 里已经有一个具有相同 ID 的 broker。

在 broker 停机、 出现网络分区或长时间垃圾回收停顿时，broker 会从 Zookeeper 上断开连接，此时 broker 在启动时创建的临时节点会自动从 Zookeeper 上移除。 监听 broker 列表的 Kafka 组件会被告知该 broker 已移除。

在关闭 broker 时，它对应的节点也会消失，不过它的 ID 会继续存在于其他数据结构中。 例如,主题的副本列表里就可能包含这些 ID。在完全关闭一个 broker 之后，如果使用相同的 ID 启动另一个全新的 broker，它会立刻加入集群，并拥有与旧 broker 相同的分区和主题。

什么是控制器

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 里创建一个临时节点/controuer 让自己成为控制器。 其他 broker 在启动时也会尝试创建这个节点,不过它们会收到一个“节点已存在”的异常,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 。 其他 broker 在控制器节点上创建 Zookeeperwatch 对象,这样它们就可以收到这个节点的变更通知。这种方式可以确保集群里一次只有一个控制器存在。

如果控制器被关闭或者与 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 watch 对象得到控制器节点消失的通知，它们会尝试让自己成为新的控制器。 第一个在 Zookeeper 里成功创建控制器节点的 broker 就会成为新的控制器，其他节点会收到“节点已存在”的异常,然后在新的控制器节点上再次创建 watch 对象。

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。该请求消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情求,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。

简而言之，Kafka 使用 Zookeeper 的临时节点来选举控制器,并在节点加入集群或退出集群时通知控制器。 控制器负责在节点加入或离开集群时进行分区首领选举。

复制-Kafka 的核心

复制功能是 Kafka 架构的核心。在 Kafka 的文档里，Kafka 把自己描述成“一个分布式的、可分区的、可复制的提交日志服务”。复制之所以这么关键，是因为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，每个主题被分为若干个分区,每个分区有多个副本。那些副本被保存在 broker 上，每个 broker 可以保存成百上千个属于不同主题和分区的副本。

replication-factor

用来设置主题的副本数。每个主题可以有多个副本，副本位于集群中不同的 broker 上，也就是说副本的数量不能超过 broker 的数量，否则创建主题时会失败。

比如 partitions 设置为 2，replicationFactor 设置为 1. Broker 为 2 个的话，分区会均匀在 broker

```
kafka-topics.bat --zookeeper localhost:2181/kafka --create --topic topicB --replication-factor 1 --partitions 2
```

比如 partitions 设置为 2，replicationFactor 设置为 2. Broker 为 2 个的话，每个 broker 都有副本存在

```
kafka-topics.bat --zookeeper localhost:2181/kafka --create --topic topicA --replication-factor 2 --partitions 2
```

副本类型。

优先副本和优先副本的关系：

古代皇帝选太子，皇帝有三个儿子，大儿子、二儿子、三儿子，优先副本是大儿子（优先副本这个关系定下来了就不会变），首领副本就是太子，皇帝选太子时，一般也会将优先副本选为首领副本<变成太子>，但是太子是有可能变动变动的，如果大儿子犯事了，那么二儿子就会成为太子）

首领副本

每个分区都有一个首领副本。为了保证一致性,所有生产者请求和消费者请求都会经过这个副本 。

跟随者副本

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,它们唯一一的任务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，其中的一个跟随者会被提升为新首领 。

优先副本

除了当前首领之外，每个分区都有一个优先副本（首选首领），创建主题时选定的首领分区就是分区的优先副本。 之所以把它叫作优先副本，是因为在创建分区时，需要在 broker 之间均衡首领副本。 因此，我们希望首选首领在成为真正的首领时，broker 间的负载最终会得到均衡。 默认情况下，Kafka 的 auto.leader.rebalance.enable 被设为 true,它会检查优先副本是不是当前首领,如果不是,并且该副本是同步的，那么就会触发首领选举，让优先副本成为当前首领。

工作机制

首领的另一个任务是搞清楚哪个跟随者的状态与自己是一致的。跟随者为了保持与首领的状态一致，在有新消息到达时尝试从首领那里复制消息，不过有各种原因会导致同步失败。例如,网络拥塞导致复制变慢，broker 发生崩溃导致复制滞后,直到重启 broker 后复制才会继续。

为了与首领保持同步，跟随者向首领发送获取数据的请求，这种请求与消费者为了读取消息而发送的请求是一样的。首领将响应消息发给跟随者。请求消息里包含了跟随者想要获取消息的偏移量，而且这些偏移量总是有序的 。

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,它是不会发送第 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复制的进度。如果跟随

者在 10s 内没有请求任何消息,或者虽然在请求消息,但在 10s 内没有请求最新的数据,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,在首领发生失效时,它就不可能成为新首领，因为它没有包含全部的消息。

相反,持续请求得到的最新消息副本被称为**同步副本**。在首领发生失效时,只有同步副本才有可能被选为新首领。

处理请求的内部机制

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 提供了一个二进制协议(基于 TCP),指定了请求消息的格式以及 broker 如何对请求作出响应——包括成功处理请求或在处理请求过程中遇到错误。

客户端发起连接并发送请求,broker 处理请求并作出响应。 broker 按照请求到达的顺序来处理它们这种顺序保证让 Kaka 具有了消息队列的特性,同时保证保存的消息也是有序的。

所有的请求消息都包含一个标准消息头:

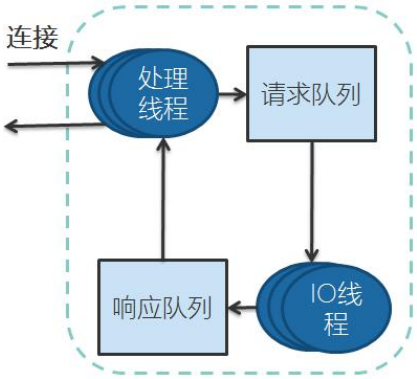
Request type(也就是 API key)

Request version(broker 可以处理不同版本的客户端请求,并根据客户端版本作出不同的响应)

Correlation id-一个具有唯一性的数字,用于标识请求消息,同时也会出现在响应消息和错误日志里(用于诊断问题)

Client Id 用于标识发送请求的客户端

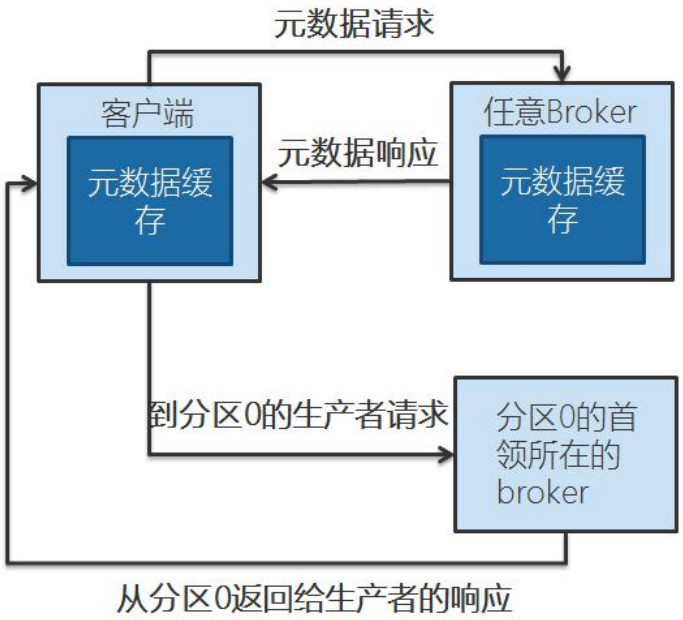
broker 会在它所监听的每一个端口上运行一个 Acceptor 线程,这个线程会创建一个连接并把它交给 Processor 线程去处理。 Processor 线程(也被叫作“网络线程”)的数量是可配置的。网络线程负责从客户端获取请求消息,把它们放进请求队列,然后从响应队列获取响应消息,把它们发送给客户端。



请求消息被放到请求队列后,IO 线程会负责处理它们。比较常见的请求类型有：

生产请求：生产者发送的请求,它包含客户端要写入 broker 的消息。

获取请求：在消费者和跟随者副本需要从 broker 读取消息时发送的请求。



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。当针对特定分区的获取请求被发送到一个不含有该分区首领的 broker 上，也会出现同样的错误。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。

那么客户端怎么知道该往哪里发送请求呢？客户端使用了另一种请求类型，也就是元数据请求。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。元数据请求可以发送给任意一个 broker ，因为所有 broker 都缓存了这些信息。

一般情况下，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。它们需要时不时地通过发送元数据请求来刷新这些信息（刷新的时间间隔通过 metadata.max.age.ms 参数来配置,2.1.3 的客户端默认参数 30S），从而知道元数据是否发生了变更，比如，在新 broker 加入

集群时，部分副本会被移动到新的 broker 上。另外，如果客户端收到“非首领”错误，它会在尝试重发请求之前先刷新元数据，因为这个错误说明了客户端正在使用过期的元数据信息，之前的请求被发到了错误的 broker 上。

生产请求

我们曾经说过， acks 这个配置参数，该参数指定了需要多少个 broker 确认才可以认为一个消息写入是成功的。不同的配置对“写入成功”的界定是不一样的,如果 acks=1,那么只要首领收到消息就认为写入成功;如果 acks=all,那么需要所有**同步副本**收到消息才算写入成功; 如果 acks=0, 那么生产者在把消息发出去之后, 完全不需要等待 broker 的响应。

包含首领副本的 broker 在收到生产请求时, 会对请求做一些验证。

- 发送数据的用户是否有主题写入权限?

  请求里包含的 acks 值是否有效(只允许出现 0、1 或 all) ?(ack=-1 等同于 ack=all)

  如果 acks=all, 是否有足够多的同步副本保证消息已经被安全写入? z

之后,消息被写入本地磁盘。在 Linux 系统上,消息会被写到文件系统缓存里,并不保证它们何时会被刷新到磁盘上。Kafka 不会一直等待数据被写到磁盘上，它依赖复制功能来保证消息的持久性。

在消息被写入分区的首领之后, broker 开始检查 acks 配置参数一如果 acks 被设为 0 或 1, 那么 broker 立即返回响应;如果 acks 被设为 all,那么请求会被保存在一个叫作炼狱的缓冲区里, 直到首领发现所有跟随者副本都复制了消息, 响应才会被返回给客户端。

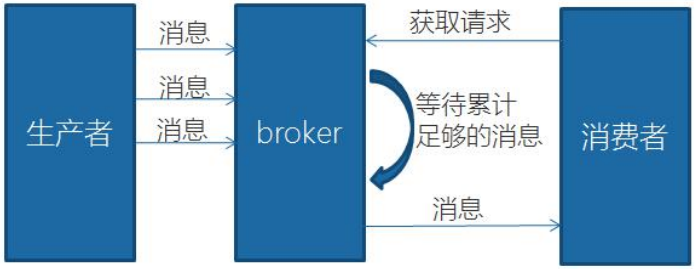
获取请求

broker 处理获取请求的方式与处理生产请求的方式很相似。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, 好像在说: “请把主题 Test 分区 0 偏移量从 53 开始的消息以及主题 Test 分区 3 偏移量从 64 开始的消息发给我。”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。 这个限制是非常重要的, 因为客户端需要为 broker 返回的数据分配足够的内存。 如果没有这个限制, broker 返回的大量数据有可能耗尽客户端的内存。

我们之前讨论过,请求需要先到达指定的分区首领上,然后客户端通过查询元数据来确保请求的路由是正确的。首领在收到请求时,它会先检查请求是否有效，比如,指定的偏移量在分区上是否存在?如果客户端请求的是已经被删除的数据,或者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 使用零复制技术向客户端发送消息——也就是说，Kafka 直接把消息从文件(或者更确切地说是 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。 这是 Kafka 与其他大部分数据库系统不一样的地方，其他数据库在将数据发送给客户端之前会先把它们保存在本地缓存里。 这项技术避免了字节复制，也不需要管理内存缓冲区，从而获得更好的性能。

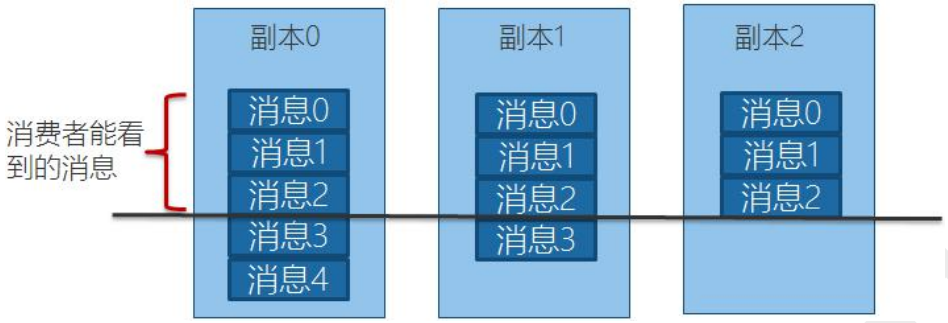
客户端除了可以设置 broker 返回数据的上限，也可以设置下限。 例如，如果把下限设置为 10KB,就好像是在告诉 broker:“等到有 10KB 数据的时候再把它们发送给我。”在主题消息流量不是很大的情况下,这样可以减少 CPU 和网络开销。 客户端发送一个请求，broker 等到有足够的数据时才把它们返回给客户端，然后客户端再发出情求，而不是让客户端每隔几毫秒就发送一次请求,每次只能得到很少的数据甚至没有数据。对比这两种情况，它们最终读取的数据总量是一样的，但前者的来回传送次数更少，因此开销也更小。



当然,我们不会让客户端一直等待 broker 累积数据。在等待了一段时间之后,就可以把可用的数据拿回处理,而不是一直等待下去。所以,客户端可以定义一个超时时间,告诉 broker: “如果你无法在 K 毫秒内累积满足要求的数据量，那么就把当前这些数据返回给我。”

ISR

并不是所有保存在分区首领上的数据都可以被客户端读取。大部分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个副本上，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而不是错误。



因为还没有被足够多副本复制的消息被认为是“不安全”的，如果首领发生崩横,另一 个副本成为新首领,那么这些消息就丢失了。如果我们允许消费者读取这些消息,可能就会破坏一致性。试想，一个消费者读取并处理了这样的一个消息,而另一个消费者发现这个消息其实并不存在。所以,我们会等到所有同步副本复制了这些消息,才允许消费者读取它们。这也意味着,如果 broker 间的消息复制因为某些原因变慢,那么消息到达消费者的时间也会随之变长(因为我们会先等待消息复制完毕) 。延迟时间可以通过参数 replica. lag. time. max. ms 来配置，它指定了副本在复制消息时可被允许的最大延迟时间。

Kafka 的数据复制是以 Partition 为单位的。而多个备份间的数据复制，通过 Follower 向 Leader 拉取数据完成。从一这点来讲，有点像 Master-Slave 方案。不同的是，Kafka 既不是完全的同步复制，也不是完全的异步复制，而是基于 ISR 的动态复制方案。

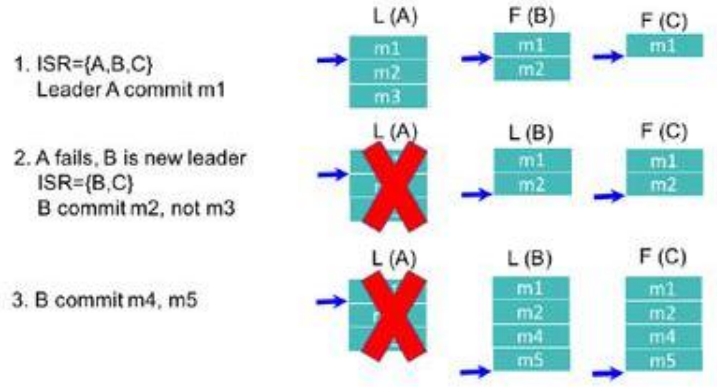
ISR，也即 In-Sync Replica。每个 Partition 的 Leader 都会维护这样一个列表，该列表中，包含了所有与之同步的 Replica（包含 Leader 自己）。每次数据写入时，只有 ISR 中的所有 Replica 都复制完，Leader 才会将其置为 Commit，它才能被 Consumer 所消费。

这种方案，与同步复制非常接近。但不同的是，这个 ISR 是由 Leader 动态维护的。如果 Follower 不能紧“跟上”Leader，它将被 Leader 从 ISR 中移除，待它又重新“跟上”Leader 后，会被 Leader 再次加加 ISR 中。每次改变 ISR 后，Leader 都会将最新的 ISR 持久化到 Zookeeper 中。

至于如何判断某个 Follower 是否“跟上”Leader，不同版本的 Kafka 的策略稍微有些区别。

从 0.9.0.0 版本开始，replica.lag.max.messages 被移除，故 Leader 不再考虑 Follower 落后的消息条数。另外，Leader 不仅会判断 Follower 是否在 replica.lag.time.max.ms 时间内向其发送 Fetch 请求，同时还会考虑 Follower 是否在该时间内与之保持同步。

示例



在第一步中，Leader A 总共收到 3 条消息，但由于 ISR 中的 Follower 只同步了第 1 条消息（m1），故只有 m1 被 Commit，也即只有 m1 可被 Consumer 消费。此时 Follower B 与 Leader A 的差距是 1，而 Follower C 与 Leader A 的差距是 2，虽然有消息的差距，但是满足同步副本的要求保留在 ISR 中。同步副本概念参见《复制》

在第二步中，由于旧的 Leader A 宕机，新的 Leader B 在 replica.lag.time.max.ms 时间内未收到来自 A 的 Fetch 请求，故将 A 从 ISR 中移除，此时 ISR={B，C}。同时，由于此时新的 Leader B 中只有 2 条消息，并未包含 m3（m3 从未被任何 Leader 所 Commit），所以 m3 无法被 Consumer 消费。

(上图中就是因为 acks 不为 all 或者-1,不全部复制，就会导致单台服务器宕机时的数据丢失 m3 丢失了)

使用 ISR 方案的原因

由于 Leader 可移除不能及时与之同步的 Follower，故与同步复制相比可避免最慢的 Follower 拖慢整体速度，也即 ISR 提高了系统可用性。

ISR 中的所有 Follower 都包含了所有 Commit 过的消息，而只有 Commit 过的消息才会被 Consumer 消费，故从 Consumer 的角度而言，ISR 中的所有 Replica 都始终处于同步状态，从而与异步复制方案相比提高了数据一致性。

ISR 相关配置说明

Broker 的 min.insync.replicas 参数指定了 Broker 所要求的 ISR 最小长度，默认值为 1。也即极限情况下 ISR 可以只包含 Leader。但此时如果 Leader 宕机，则该 Partition 不可用，可用性得不到保证。

只有被 ISR 中所有 Replica 同步的消息才被 Commit，但 Producer 发布数据时，Leader 并不需要 ISR 中的所有 Replica 同步该数据才确认收到数据。Producer 可以通过 acks 参数指定最少需要多少个 Replica 确认收到该消息才视为该消息发送成功。acks 的默认值是 1，即 Leader 收到该消息后立即告诉 Producer 收到该消息，此时如果在 ISR 中的消息复制完该消息前 Leader 宕机，那该条消息会丢失。而如果将该值设置为 0，则 Producer 发送完数据后，立即认为该数据发送成功，不作任何等待，而实际上该数据可能发送失败，并且 Producer 的 Retry 机制将不生效。更推荐的做法是，将 acks 设置为 all 或者-1，此时只有 ISR 中的所有 Replica 都收到该数据（也即该消息被 Commit），Leader 才会告诉 Producer 该消息发送成功，从而保证不会有未知的数据丢失。

物理存储机制

Kafka 的基本存储单元是分区。分区无法在多个 broker 间进行再细分,也无法在同一个 broker 的多个磁盘上进行再细分。

在配置 Kafka 的时候，管理员指定了一个用于存储分区的目录清单——也就是 log.dirs 参数的值（不要把它与存放错误日志的目录混淆了，日志目录是配置在 1og4j.properties 文件里的)。 该参数一般会包含每个挂载点的目录。

分区分配

在创建主题时，Kafka 首先会决定如何在 broker 间分配分区。假设你有 6 个 broker，打算创建一个包含 10 个分区的主题,并且复制系数为 3（确保至少有 3 台 broker）。那么 Kafka 就会有 30 个分区副本，它们可以被分配给 6 个 broker。 在进行分区分配时，我们要达到如下的目标。

- 在 broker 间平均地分布分区副本。对于我们的例子来说，就是要保证每个 broker 可以分到 5 个副本。
- 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。
- 如果为 broker 指定了机架信息,那么尽可能把每个分区的副本分配到不同机架的 broker 上 。 这样做是为了保证一个机架的不可用不会导致整体的分区不可用 。

为了实现这个目标，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 l 会在 broker5 上，首领分区 2 会在 broker 0 上(只有 6 个 broker)，并以此类推。然后，我们从分区首领开始,依次分配跟随者副本。如果分区 0 的首领在 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 broker 0 上。分区 1 的首领在 broker5 上,那么它的第一个跟随者副本在 broker0 上,第二个跟随者副本在 broker1 上。

为分区和副本选好合适的 broker 之后，接下来要决定这些分区应该使用哪个目录。 我们单独为每个分区分配目录，规则很简单：计算每个目录里的分区数量，新的分区总是被添加到数量最小的那个目录里。 也就是说，如果添加了一个新磁量，所有新的分区都会被创建到这个磁盘上。因为在完成分配工作之前,新磁盘的分区数量总是最少的。 (最少使用原则)

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。 相反，Kafka 管理员为每个主题配置了数据保留期限，规定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 。

因为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，所以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,以较小的那个为准。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。

当前正在写入数据的片段叫作活跃片段。 活动片段永远不会被删除，所以如果你要保留数据 1 天,但片段里包含了 5 天的数据,那么这些数据会被保留 5 天,因为在片段被关闭之前这些数据无法被删除。如果你要保留数据一周,而且每天使用一个新片段,那么你就会看到,每天在使用一个新片段的同时会删除一个最老的片段一所以大部分时间该分区会有 7 个片段存在。

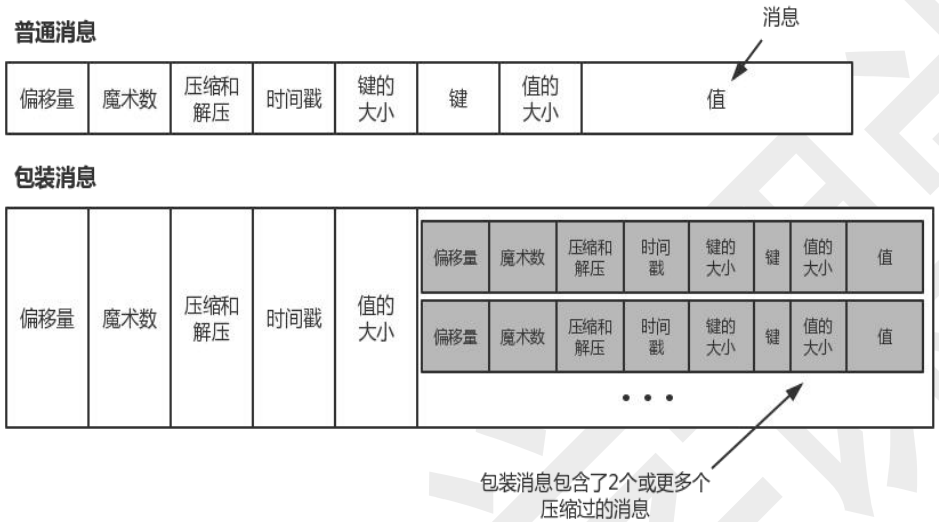
文件格式

Kafka 的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者发送给消费者的消息格式是一样的 。 因为使用了相同的消息格式进行磁盘存储和网络传输，Kafka 可以使用零复制技术给消费者发送消息，同时避免了对生产者已经压缩过的消息进行解压和再压缩。

除了键、值和偏移量外,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 或 Lz4)和时间戳(在 0.10.0 版本里引入的)。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，这个是可配置的。

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，被当作 “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之后，会看到整个批次的消息，它们都有自己的时间戳和偏移量。

如果在生产者端使用了压缩功能(极力推荐),那么发送的批次越大,就意味着在网络传输和磁盘存储方面会获得越好的压缩性能，同时意味着如果修改了消费者使用的消息格式 (例如，在消息里增加了时间戳)，那么网络传输和磁盘存储的格式也要随之修改，而且 broker 要知道如何处理包含了两种消息格式的文件。一种是普通消息，一种是包装消息



Kafka 附带了一个叫 DumpLogSegment 的工具，可以用它查看片段的内容。它可以显示每个消息的偏移量、校验和、魔术数字节、消息大小和压缩算法。

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 100 开始的 1MB 消息，那么 broker 必须立即定位到偏移量 100(可能是在分区的任意一个片段里)，然后开始从这个位置读取消息。为了帮助 broker 更快地定位到指定的偏移量，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和偏移量在文件里的位置。

索引也被分成片段，所以在删除消息时，也可以删除相应的索引 。 Kafka 不维护索引的校验和。 如果索引出现损坏，Kafka 会通过重新读取消息并录制偏移量和位置来重新生成索引。 如果有必要，管理员可以删除索引，这样做是绝对安全的，Kafka 会自动重新生成这些索引 。

超时数据的清理机制

一般情况下，Kafka 会根据设置的时间保留数据,把超过时效的旧数据删除掉。不过,试想一下这样的场景,如果你使用 Kafka 保存客户的收货地址,那么保存客户的最新地址比保存客户上周甚至去年的地址要有意义得多,这样你就不用担心会用错旧地址,而且短时间内客户也不会修改新地址。另外一个场景,一个应用程序使用 Kafka 保存它的状态,每次状态发生变化,它就把状态写入 Kafka。在应用程序从崩演中恢复时,它从 Kafka 读取消息来恢复最近的状态。在这种情况下,应用程序只关心它在崩粉前的那个状态,而不关心运行过程中的那些状态。

Kafka 通过改变主题的保留策略来满足这些使用场景 。 早于保留时间的事件会被删除，为每个键保留最新的值，从而达到清理的效果

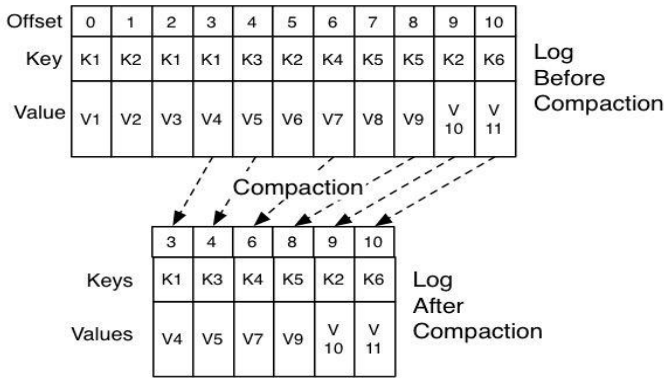
每个日志片段可以分为以下两个部分 。

干净的部分，这些消息之前被清理过，每个键只有一个对应的值，这个值是上一次清理时保留下来的。 **污浊的部分**，这些消息是在上一次清理之后写入的。

为了清理分区，清理线程会读取分区的污独部分，并在内存里创建一个 map。 map 里的每个元素包含了消息键的散列值和消息的偏移量,键的散列值是 16B,加上偏移量总共是 24B。如果要清理一个 1GB 的日志片段,并假设每个消息大小为 1KB,那么这个片段就包含_一百万个消息,而我们只需要用 24MB 的 map 就可以清理这个片段。 (如果有重复的键，可以重用散列项，从而使用更少的内存。)

清理线程在创建好偏移量 map 后,开始从干净的片段处读取消息,从最旧的消息开始,把它们的内容与 map 里的内容进行比对。它会检查消息的键是否存在于 map 中，如果不存在，那么说明消息的值是最新的,就把消息复制到替换片段上。如果键已存在,消息会被忽略，因为在分区的后部已经有一个具有相同键的消息存在。在复制完所有的消息之后,我们就将替换片段与原始片段进行交换,然后开始清理下一个片段。完成整个清理过程之后,每个键对应一个不同的消息一这些消息的值都是最新的。 清理前后的分区片段如图所示。

清理的思想就是根据 Key 的重复来进行整理，注意，它不是数据删除策略，而是类似于压缩策略，如果 key 送入了值，对于业务来说，key 的值应该是最新的 value 才有意义，所以进行清理后只会保存一个 key 的最新的 value,这个适用于一些业务场景，比如说 key 代表用户 ID,Value 用户名称，如果使用清理功能就能够达到最新的用户的名称的消息（这个功能有限，请参考使用）



可靠的数据传递

Kafka 提供的可靠性保证和架构上的权衡

可靠性时，我们一般会使用保证这个词，它是指确保系统在各种不同的环境下能够发生一致的行为。

ACID 大概是大家最熟悉的一个例子，它是关系型数据库普遍支持的标准可靠性保证。ACID 指的是原子性、一致性、隔离性和持久性。如果一个供应商说他们的数据库遵循 ACID 规范，其实就是在说他们的数据库支持与事务相关的行为。

有了这些保证，我们才能相信关系型数据库的事务特性可以确保应用程序的安全。我们知道系统承诺可以做到些什么，也知道在不同条件下它们会发生怎样的行为。我们了解这些保证机制，并基于这些保证机制开发安全的应用程序。

所以，了解系统的保证机制对于构建可靠的应用程序来说至关重要，这也是能够在不同条件下解释系统行为的前提。那么 Kafka 可以在哪些方面作出保证呢？

- Kafka 可以保证分区消息的顺序。如果使用同一个生产者往同一个分区写入消息，而且消息 B 在消息 A 之后写入，那么 Kafka 可以保证消息 B 的偏移量比消息 A 的偏移量大，而且消费者会先读取消息 A 再读取消息 B 。

- 只有当消息被写入分区的所有同步副本时（但不一定要写入磁盘），它才被认为是“ 已提交”的。生产者可以选择接收不同类型的确认，比如在消息被完全提交时的确认，或者在消息被写入首领副本时的确认，或者在消息被发送到网络时的确认。
- 只要还有一个副本是活跃的，那么已经提交的消息就不会丢失。消费者只能读取已经提交的消息。

这些基本的保证机制可以用来构建可靠的系统，但仅仅依赖它们是无法保证系统完全可靠的。构建一个可靠的系统需要作出一些权衡， Kafka 管理员和开发者可以在配置参数上作出权衡，从而得到他们想要达到的可靠性。这种权衡一般是指消息存储的可靠性和一致性的重要程度与可用性、高吞吐量、低延迟和硬件成本的重要程度之间的权衡。

复制

Kafka 的复制机制和分区的多副本架构是 Kafka 可靠性保证的核心。把消息写入多个副本可以使 Kafka 在发生崩愤时仍能保证消息的持久性。

回顾一下主要内容：

Kafka 的主题被分为多个分区，分区是基本的数据块。分区存储在单个磁盘上， Kafka 可以保证分区里的事件是有序的，分区可以在线（可用），也可以离线（不可用） 。每个分区可以有多个副本，其中一个副本是首领。所有的事件都直接发送给首领副本，或者直接从首领副本读取事件。其他副本只需要与首领保持同步，并及时复制最新的事件。当首领副本不可用时，其中一个同步副本将成为新首领。

分区首领是同步副本，而对于跟随者副本来说，它需要满足以下条件才能被认为是同步的。

- 与 Zoo keeper 之间有一个活跃的会话，也就是说，它在过去的 6 秒（可配置）内向 Zoo keeper 发送过心跳。
- 在过去的 10s 内（可配置）从首领那里获取过消息。
- 在过去的 10s 内从首领那里获取过最新的消息。光从首领那里获取消息是不够的，它还必须是儿乎零延迟的。

如果跟随者副本不能满足以上任何一点，比如与 Zookeeper 断开连接，或者不再获取新消息，或者获取消息滞后了 10s 以上，那么它就被认为是不同步的。一个不同步的副本通过与 Zookeeper 重新建立连接，井从首领那里获取最新消息，可以重新变成同步的。这个过程在网络出现临时问题井很快得到修复的情况下会很快完成，但如果 broker 发生崩愤就需要较长的时间。

注意：如果一个或多个副本在同步和非同步状态之间快速切换，说明集群内部出现了问题，通常是 Java 不恰当的垃圾回收配置导致的。不恰当的垃圾回收配置会造成几秒钟的停顿，从而让 broker 与 Zoo keeper 之间断开连接，最后变成不同步的，进而发生状态切换。

Broker 配置对可靠性的影响

复制系数

主题级别的配置参数是 replication.factor，而在 broker 级别则可以通过 default. replication.factor 来配置自动创建的主题。

Kafka 的默认复制系数就是 3，不过用户可以修改它。

如果复制系数为 N，那么在凡 l 个 broker 失效的情况下，仍然能够从主题读取数据或向主题写入数据。所以，更高的复制系数会带来更高的可用性、可靠性和更少的故障。另一方面，复制系数 N 需要至少 N 个 broker ，而且会有 N 个数据副本，也就是说它们会占用 N 倍的磁盘空间。我们一般会在可用性和存储硬件之间作出权衡。

那么该如何确定一个主题需要几个副本呢？这要看主题的重要程度，以及你愿意付出多少成本来换取可用性。。

如果因 broker 重启导致的主题不可用是可接受的（这在集群里是很正常的行为），那么把复制系数设为 1 就可以了。在作出这个权衡的时候，要确保这样不会对你的组织和用户造成影响，因为你在节省了硬件成本的同时也降低了可用性。复制系数为 2 意味着可以容忍 1 个 broker 发生失效，看起来已经足够了。不过要记住，有时候 1 个 broker 发生失效会导致集群不稳定（通常是旧版的 Kafka ），迫使你重启另一个 broker——集群控制器。也就是说，如果将复制系数设为 2 ，就有可能因为重启等问题导致集群不可用。

基于以上几点原因，在要求可用性的场景里把复制系数设为 3 。在大多数情况下，这已经足够安全了，不过要求更可靠时，可以设为更高，比如我 5 个副本，以防不测。

副本的分布也很重要。默认情况下， Kafka 会确保分区的每个副本被放在不同的 broker 上。不过，有时候这样仍然不够安全。如果这些 broker 处于同一个机架上， 一旦机架的交换机发生故障，分区就会不可用，这时候把复制系数设为多少都不管用。为了避免机架级别的故障，我们建议把 broker 分布在多个不同的机架上。

不完全的首领选举

unclean.leader.election 只能在 broker 级别（实际上是在集群范围内）进行配置， 它的默认值是 true。

当分区首领不可用时， 一个同步副本会被选为新首领。如果在选举过程中没有丢失数据，也就是说提交的数据同时存在于所有的同步副本上，那么这个选举就是“完全”的。

但如果在首领不可用时其他副本都是不同步的，我们该怎么办呢？

这种情况会在以下两种场景里出现。

• 分区有 3 个副本，其中的两个跟随者副本不可用（比如有两个 broker 发生崩愦）。这个时候，如果生产者继续往首领写入数据，所有消息都会得到确认井被提交（因为此时首领是唯一的同步副本）。现在我们假设首领也不可用了（又一个 broker 发生崩愦），这个时候，如果之前的一个跟随者重新启动，它就成为了分区的唯一不同步副本。

• 分区有 3 个副本，因为网络问题导致两个跟随者副本复制消息滞后，所以尽管它们还在复制消息，但已经不同步了。首领作为唯一的同步副本继续接收消息。这个时候，如果首领变为不可用，另外两个副本就再也无法变成同步的了。

对于这两种场景，我们要作出一个两难的选择。

如果不同步的副本不能被提升为新首领，那么分区在旧首领（最后一个同步副本）恢复之前是不可用的。有时候这种状态会持续数小时（比如更换内存芯片）。

• 如果不同步的副本可以被提升为新首领，那么在这个副本变为不同步之后写入旧首领的消息、会全部丢失，导致数据不一致。为什么会这样呢？假设在副本 0 和副本 1 不可用时，偏移量 100-200 的消息被写入副本 2 （首领）。现在副本 2 变为不可用的，而副本 0 变为可用的。副本 0 只包含偏移量 0～ 100 的消息，不包含偏移量 100～ 200 的悄息。如果我们允许副本 0 成为新首领，生产者就可以继续写人数据，消费者可以继续读取数据。于是，新首领就有了偏移量 100 ～200 的新消息。这样，部分消费者会读取到偏移量 100 ～200 的旧消息，部分消费者会读取到偏移量 100～200 的新消息，还有部分消费者读取的是二者的混合。这样会导致非常不好的结果，比如生成不准确的报表。另外，副本 2 可能会重新变为可用，并成为新首领的跟随者。这个时候，它会把比当前首领旧的消息全部删除，而这些消息对于所有消费者来说都是不可用的。

简而言之，如果我们允许不同步的副本成为首领，那么就要承担丢失数据和出现数据不一致的风险。如果不允许它们成为首领，那么就要接受较低的可用性，因为我们必须等待原先的首领恢复到可用状态。

如果把 unclean.leader.election 设为 true ，就是允许不同步的副本成为首领（也就是“ 不完全的选举”），那么我们将面临丢失消息的风险。如果把这个参数设为 false ,就要等待原先的首领重新上线，从而降低了可用性。

我们经常看到一些对数据质量和数据一致性要求较高的系统会禁用这种不完全的首领选举（ 把这个参数设为 false ） 。比如银行系统，大部分银行系统宁愿选择在几分钟甚至几个小时内不处理信用卡支付事务，也不会冒险处理错误的消息。不过在对可用性要求较高的系统里，比如实时点击流分析系统， 一般会启用不完全的首领选举。

最少同步副本

在主题级别和 broker 级别上，这个参数都叫 min.insync.replicas。

我们知道，尽管为一个主题配置了 3 个副本，还是会出现只有一个同步副本的情况（ acks=0 或 1）。如果这个同步副本变为不可用，我们必须在可用性和一致性之间作出选择---这又是一个两难的选择。根据 Kafka 对可靠性保证的定义，消息只有在被写入到所有同步副本之后才被认为是已提交的。但如果这里的“所有副本”只包含一个同步副本，那么在这个副本变为不可用时，数据就会丢失。

如果要确保已提交的数据被写入不止一个副本，就需要把最少同步副本数量设置为大一点的值。对于一个包含 3 个副本的主题，如果 min.insync.replicas 被设为 2 ，那么至少要存在两个同步副本才能向分区写入数据。

如果 3 个副本都是同步的，或者其中一个副本变为不可用，都不会有什么问题。不过，如果有两个副本变为不可用，那么 broker 就会停止接受生产者的请求。尝试发送数据的生产者会收到 NotEnoughReplicasException 异常。消费者仍然可以继续读取已有的数据。实际上，如果使用这样的配置，那么当只剩下一个同步副本时，它就变成只读了，这是为了避免在发生不完全选举时数据的写入和读取出现非预期的行为。为了从只读状态中恢复，必须让两个不可用分区中的一个重新变为可用的（比如重启 broker ），并等待它变为同步的。

可靠系统里的生产者

即使我们尽可能把 broker 配置得很可靠，但如果没有对生产者进行可靠性方面的配置， 整个系统仍然有可能出现突发性的数据丢失。

请看以下两个例子。

为 broker 配置了 3 个副本，并且禁用了不完全首领选举，这样应该可以保证万无一失。我们把生产者发送消息的 acks 设为 1 （只要首领接收到消息就可以认为消息写入成功）。生产者发送一个消息给首领，首领成功写入，但跟随者副本还没有接收到这个消息。首领向生产者发送了一个响应，告诉它“消息写入成功”，然后它崩溃了，而此时消息还没有被其他副本复制过去。另外两个副本此时仍然被认为是同步的（毕竟判断一个副本不同步需要一小段时间），而且其中的一个副本成了新的首领。因为消息还没有被写入这个副本，所以就丢失了，但发送消息的客户端却认为消息已成功写入。

因为消费者看不到丢失的消息，所以此时的系统仍然是一致的（因为副本没有收到这个消息，所以消息不算已提交），但从生产者角度来看，它丢失了一个消息。

- 为 broker 配置了 3 个副本，并且禁用了不完全首领选举。我们接受了之前的教训， 把生产者的 acks 设为 all 。假设现在往 Kafka 发送消息，分区的首领刚好崩愤，新的首领正在选举当中， Kafka 会向生产者返回“首领不可用”的响应。在这个时候，如果生产者没能正确处理这个错误，也没有重试发送消息直到发送成功，那么消息也有可能丢失。这算不上是 broker 的可靠性问题，因为 broker 并没有收到这个消息。这也不是一致性问题，因为消费者并没有读到这个消息。问题在于如果生产者没能正确处理这些错误，弄丢消息的是它们自己。

那么，我们该如何避免这些悲剧性的后果呢？从上面两个例子可以看出，每个使用 Kafk a 的开发人员都要注意两件事情。

1. 根据可靠性需求配置恰当的 acks 值。
2. 在参数配置和代码里正确处理错误。

发送确认

复习一下，生产者可以选择以下 3 种不同的确认模式。

acks=0 意味着如果生产者能够通过网络把消息发送出去，那么就认为消息已成功写入 Kafka 。在这种情况下还是有可能发生错误，比如发送的对象无法被序列化或者网卡发生故障，但如果是分区离线或整个集群长时间不可用，那就不会收到任何错误。即使是在发生完全首领选举的情况下，这种模式仍然会丢失消息，因为在新首领选举过程中它并不知道首领已经不可用了。在 acks=0 模式下的运行速度是非常快的（这就是为什么很多基准测试都是基于这个模式），你可以得到惊人的吞吐量和带宽利用率，不过如果选择了这种模式， 一定会丢失一些消息。

acks=1 意味若首领在收到消息并把它写入到分区数据文件（不一定同步到磁盘上）时会返回确认或错误响应。在这个模式下，如果发生正常的首领选举，生产者会在选举时收到一个 LeadeNotAvailableExcepti.on 异常，如果生产者能恰当地处理这个错误，它会重试发送消息，最终消息会安全到达新的首领那里。不过在这个模式下仍然有可能丢失数据，比如消息已经成功写入首领，但在消息被复制到跟随者副本之前首领发生崩溃。

acks=all 意味着首领在返回确认或错误响应之前，会等待所有同步副本都收到悄息。如果和 min.insync.replicas 参数结合起来，就可以决定在返回确认前至少有多少个副本能够收到消息。这是最保险的做法——生产者会一直重试直到消息被成功提交。不过这也是最慢的做法，生产者在继续发送其他消息之前需要等待所有副本都收到当前的消息。可以通过使用异步模式和更大的批次来加快速度，但这样做通常会降低吞吐量。

配置生产者的重试参数

生产者需要处理的错误包括两部分： 一部分是生产者可以自动处理的错误，还有一部分是需要开发者手动处理的错民。

如果 broker 返回的错误可以通过重试来解决，那么生产者会自动处理这些错误。生产者向 broker 发送消息时， broker 可以返回一个成功响应码或者一个错误响应码。错民响应码可以分为两种， 一种是在重试之后可以解决的，还有一种是无法通过重试解决的。例如，如果 broker 返回的是 LEADER_NOT_AVAILABLE 错误，生产者可以尝试重新发送消息。也许在这个时候一个新的首领被选举出来了，那么这次发送就会成功。也就是说，LEADER_NOT_AVAILABLE 是一个可重试错误。

另一方面，如果 broker 返回的是 INVALID_CONFIG 错误，即使通过重试也无能改变配置选项，所以这样的重试是没有意义的。这种错误是不可重试错误。

一般情况下，如果你的目标是不丢失任何消息，那么最好让生产者在遇到可重试错误时能够保持重试。为什么要这样？因为像首领选举或网络连接这类问题都可以在几秒钟之内得到解决，如果让生产者保持重试，就不需要额外去处理这些问题了。

那么为生产者配置多少重试次数比较好？这个要看在生产者放弃重试并抛出异常之后想做些什么。如果你想抓住异常并再多重试几次，那么就可以把重试次数设置得多一点， 让生产者继续重试；如果你想直接丢弃消息，多次重试造成的延迟已经失去发送消息的意义；如果你想把消息保存到某个地方然后回过头来再继续处理，那就可以停止重试。

Kafka 的跨数据中心复制工具（ MirrorMaker）默认会进行无限制的重试。作为一个具有高可靠性的复制工具，它决不会丢失消息。

要注意，重试发送一个已经失败的消息会带来一些风险，如果两个消息都写入成功，会导致消息重复。例如，生产者因为网络问题没有收到 broker 的确认，但实际上消息已经写入成功，生产者会认为网络出现了临时故障，就重试发送该消息（因为它不知道消息已经写入成功）。在这种情况下，broker 会收到两个相同的消息。重试和恰当的错误处理可以保证每个消息“至少被保存一次”，但无法保证每个消息“只被保存一次”。现实中的很多应用程序在消息里加入唯一标识符，用于检测重复消息，消费者在读取消息时可以对它们进行清理。还要一些应用程序可以做到消息的“幂等”，也就是说，即使出现了重复消息，也不会对处理结果的正确性造成负面影响。

额外的错误处理

使用生产者内置的重试机制可以在不造成消息丢失的情况下轻松地处理大部分错误，不过对于开发人员来说，仍然需要处理其他类型的错误，包括：

- 不可重试的 broker 错误，例如消息大小错误、认证错误等 . 在消息发送之前发生的错误，例如序列化错误：

- 在生产者达到重试次数上限时或者在消息占用的内存达到上限时发生的错误。

错误处理器的代码逻辑与具体的应用程序及其目标有关。丢弃“不合理的消息”？把错误记录下来？把这些消息保存在本地磁盘上？具体使用哪一种逻辑要根据具体的架构来决定。如果错误处理只是为了重试发送消息，那么最好还是使用生产者内置的重试机制。

可靠系统里的消费者

可以看到，只有那些被提交到 Kafka 的数据，也就是那些已经被写入所有同步副本的数据，对消费者是可用的，这意味着消费者得到的消息已经具备了一致性。

消费者唯一要做的是跟踪哪些消息是已经读取过的，哪些是还没有读取过的。这是在读取消息时不丢失消息的关键。

在从分区读取数据时，消费者会获取一批事件，检查这批事件里最大的偏移量，然后从这个偏移量开始读取另外一批事件。这样可以保证消费者总能以正确的顺序获取新数据， 不会错过任何事件。

如果一个消费者退出，另一个消费者需要知道从什么地方开始继续处理，它需要知道前一个消费者在退出前处理的最后一个偏移量是多少。所谓的“另一个”消费者，也可能就是它自己重启之后重新回来工作。这也就是为什么消费者要“提交”它们的偏移量。它们把当前读取的偏移量保存起来，在退出之后，同一个群组里的其他消费者就可以接手它们的工作。如果消费者提交了偏移量却未能处理完消息，那么就有可能造成消息丢失，这也是消费者丢失消息的主要原因。在这种情况下，如果其他消费者接手了工作，那些没有被处理完的消息就会被忽略，永远得不到处理。所以我们要重视偏移量提交的时间点和提交的方式。

消费者的可靠性配置

为了保证消费者行为的可靠性，需要注意以下 4 个非常重要的配置参数。

第 1 个是 group.id 。如果两个消费者具有相同的 group.id ， 并且订阅了同一个主题，那么每个消费者会分到主题分区的一个子集， 也就是说它们只能读到所有消息的一个子集（不过群组会读取主题所有的消息）。如果你希望消费者可以看到主题的所有消息，那么需要为它们设置唯一的 group.id 。

第 2 个是 auto.offset.reset 。这个参数指定了在没有偏移量可提交时（比如消费者第 1 次启动时）或者请求的偏移量在 broker 上不存在时，消费者会做些什么。这个参数有两种配置。一种是 earliest ，如果选择了这种配置，消费者会从分区的开始位置读取数据，不管偏移量是否有效，这样会导致消费者读取大量的重复数据，但可以保证最少的数据丢失。一种是 latest，如果选择了这种配置， 消费者会从分区的末尾开始读取数据，这样可以减少重复处理消息，但很有可能会错过一些消息。

第 3 个是 enable.auto.commit 。这是一个非常重要的配置参数，你可以让消费者基于任务调度自动提交偏移量，也可以在代码里手动提交偏移量。自动提交的一个最大好处是，在实现消费者逻辑时可以少考虑一些问题。如果你在消费者轮询操作里处理所有的数据，那么自动提交可以保证只提交已经处理过的偏移量。自动提交的主要缺点是，无法控制重复处理消息（比如消费者在自动提交偏移量之前停止处理消息），而且如果把消息交给另外一个后台线程去处理，自动提交机制可能会在消息还没有处理完毕就提交偏移量。

第 4 个配置参数 auto.commit.interval.ms 与第 3 个参数有直接的联系。如果选择了自动提交偏移盘，可以通过该参数配置提交的频度， 默认值是每 5 秒钟提交一次。一般来说，频繁提交会增加额外的开销，但也会降低重复处理消息的概率。

显式提交偏移量

如果选择了自动提交偏移量，就不需要关心显式提交的问题。不过如果希望能够更多地控制偏移量提交的时间点，那么就要仔细想想该如何提交偏移量了一一要么是为了减少重复处理消息，要么是因为把消息处理逻辑放在了轮询之外。

在开发具有可靠性的消费者应用程序时需要注意的事项。我们先从简单的开始，再逐步深入。

1．总是在处理完事件后再提交偏移量

如果所有的处理都是在轮询里完成，而且消息处理总是幂等的，或者少量消息丢失无关紧要， 那么可以使用自动提交，或者在轮询结束时进行手动提交。

2. 提交频度是性能和重复消息数量之间的权衡

即使是在最简单的场景里，比如所有的处理都在轮询里完成，并且不需要在轮询之间维护状态，你仍然可以在一个循环里多次提交偏移量（甚至可以在每处理完一个事件之后），或者多个循环里只提交一次，这完全取决于你在性能和重复处理消息之间作出的权衡。

3. 确保对提交的偏移量心里有数

在轮询过程中提交偏移量有一个不好的地方，就是提交的偏移量有可能是读取到的最新偏移量，而不是处理过的最新偏移量。要记住，在处理完消息后再提交偏移量是非常关键的，否则会导致消费者错过消息。

4．再均衡

在设计应用程序时要注意处理消费者的再均衡问题。一般要在分区被撤销之前提交偏移量，并在分配到新分区时清理之前的状态。

5.消费者可能需要重试

有时候，在进行轮询之后，有些消息不会被完全处理，可能稍后再来处理。例如，假设要把 Kafka 的数据写到数据库里，不过那个时候数据库不可用，于是你想稍后重试。要注意，你提交的是偏移量，而不是对消息的“确认”，这个与传统的发布和订阅消息系统不太一样。如果记录的#30 处理失败，但记录的#31 处理成功，那么你不应该提交#31， 否则会导致的#31 以内的偏移量都被提交，包括的#30 在内。不过可以采用下面这种模式来解决这个问题。

在遇到可重试错误时，把错误写入一个独立的主题，然后继续。一个独立的消费者群组负责从该主题上读取错误消息，并进行重试，或者使用其中的一个消费者同时从该主题上读取错误消息并进行重试，不过在重试时需要暂停该主题。这种模式有点像其他消息系统里的死信队列 。

6.消费者可能需要维护状态

有时候你希望在多个轮询之间维护状态，例如，你想计算消息的移动平均数，希望在首次轮询之后计算平均数，然后在后续的轮询中更新这个结果。如果进程重启，你不仅需要从上一个偏移量开始处理数据，还要恢复移动平均数。有一种办法是在提交偏移量的同时把最近计算的平均数写到一个“结果”主题上。消费者线程在重新启动之后，它就可以拿到最近的平均数并接着计算。不过这并不能完全地解决问题，因为 Kafka 并没有提供事务支持。消费者有可能在写入平均数之后来不及提交偏移量就崩溃了，或者反过来也一样。这是一个很复杂的问题，你不应该尝试自己去解决这个问题，建议尝试一下 Kafka 流计算，它为聚合、连接、时间窗和其他复杂的分析提供了高级的 API 。

7. 长时间处理

有时候处理数据需要很长时间：你可能会从发生阻塞的外部系统获取信息，或者把数据写到外部系统，或者进行一个非常复杂的计算，但是我们要尽量保持轮询。在这种情况下， 一种常见的做法是使用一个线程地来处理数据，因为使用多个线程可以进行并行处理，从而加快处理速度。在把数据移交给线程地去处理之后，你就可以暂停消费者，然后保持轮询，但不获取新数据，直到工作线程处理完成。在工作线程处理完成之后，可以让消费者继续获取新数据。

8. 仅一次传递

有些应用程序不仅仅需要“至少一次”（意味着没有数据丢失），还需要“仅一次”语义。Kafka 现在还不能完全支持仅一次语义，消费者还是有一些办法可以保证 Kafka 里的每个消息只被写到外部系统一次（但不会处理向 Kafka 写入数据时可能出现的重复数据） 。

实现仅一次处理最简单且最常用的办能是把结果写到一个支持唯一键的系统里，比如键值存储引擎、关系型数据库、ElasticSearch 或其他数据存储引擎。在这种情况下，要么消息本身包含一个唯一键（通常都是这样），要么使用主题、分区和偏移量的组合来创建唯一键——它们的组合可以唯一标识一个 Kafka 记录。如果你把消息和一个唯一键写入系统，然后碰巧又读到一个相同的消息，只要把原先的键值覆盖掉即可。数据存储引擎会覆盖已经存在的键值对，就像没有出现过重复数据一样。这个模式被叫作幂等性写入，它是一种很常见也很有用的模式。

如果写入消息的系统支持事务， 那么就可以使用另一种方法。最简单的是使用关系型数据库。我们把消息和偏移量放在同一个事务里，这样它们就能保持同步。在消费者启动时，它会获取最近处理过的消息偏移量，然后调用 seek （）方也从该偏移量位置继续读取数据。