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

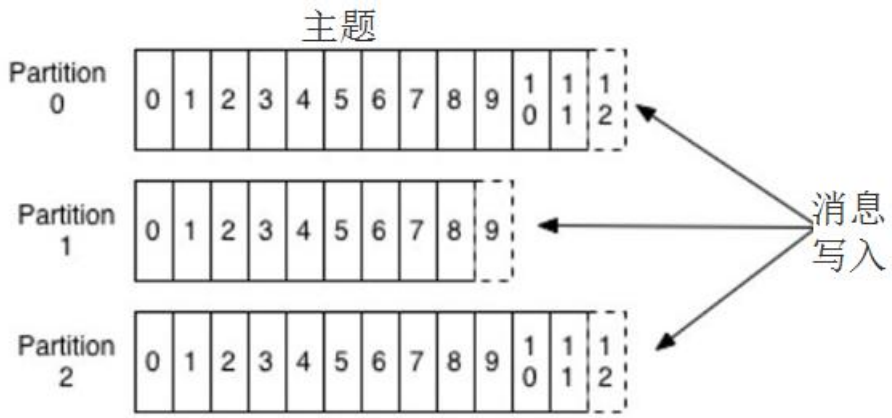
消息，Kafka 里的数据单元，也就是我们一般消息中间件里的消息的概念（可以比作数据库中一条记录）。消息由字节数组组成。消息还可以包含键（可选元数据，也是字节数组），主要用于对消息选取分区。

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。

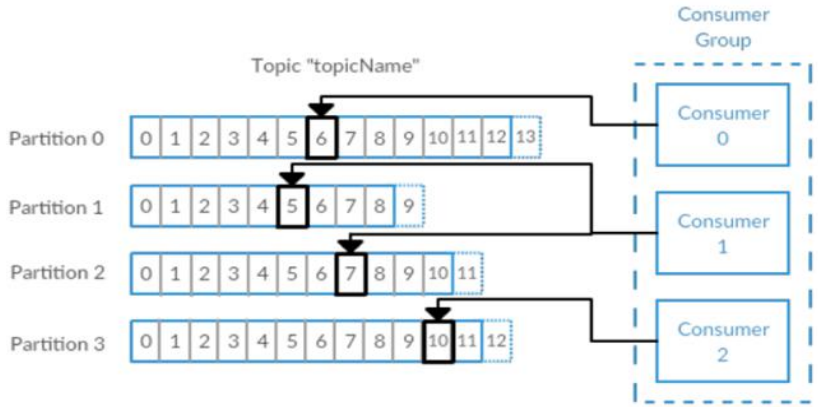
生产者和消费者、偏移量、消费者群组

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）

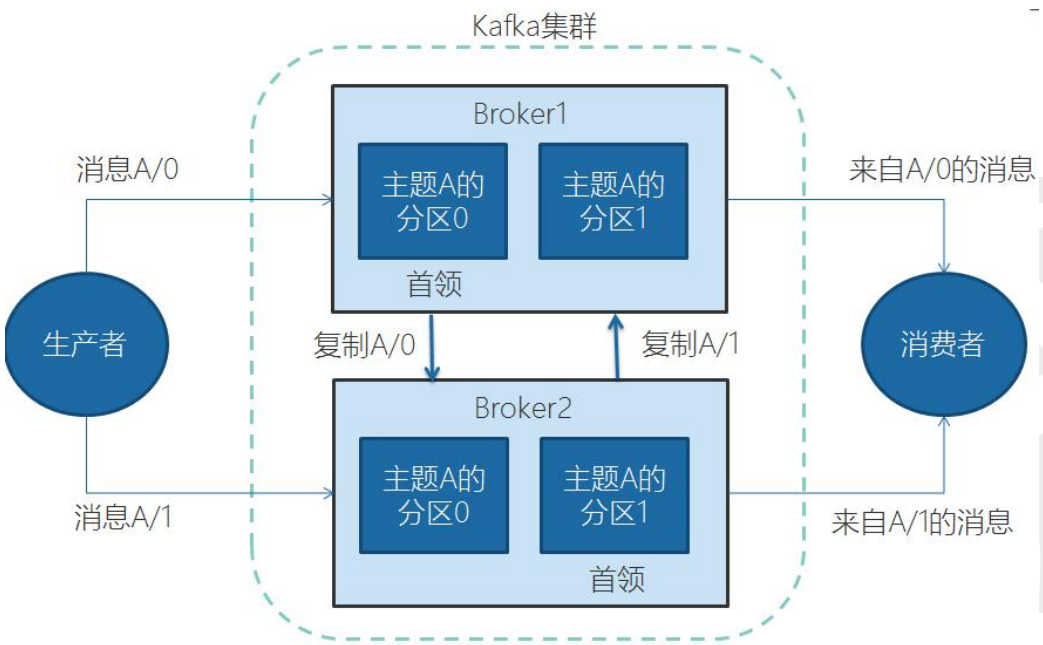
Broker 和集群

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。

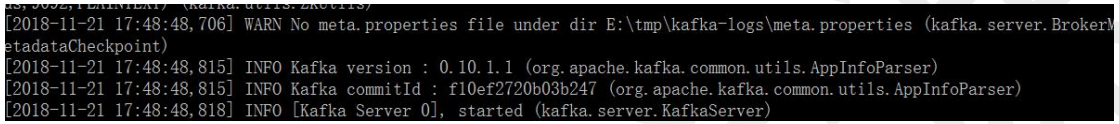
运行

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）

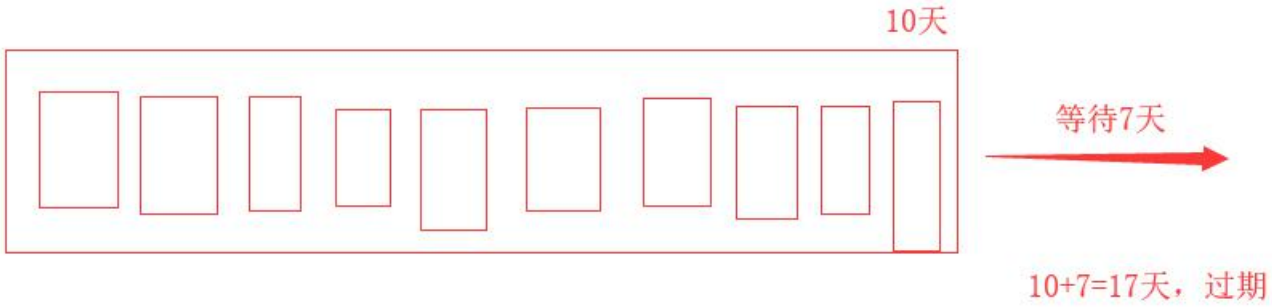
log.retention.bytes

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。

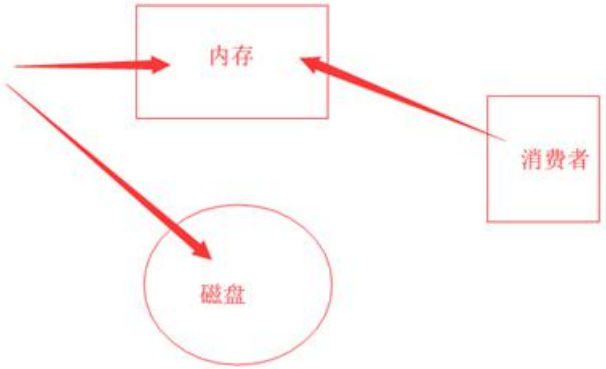
磁盘吞吐量/磁盘容量

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。

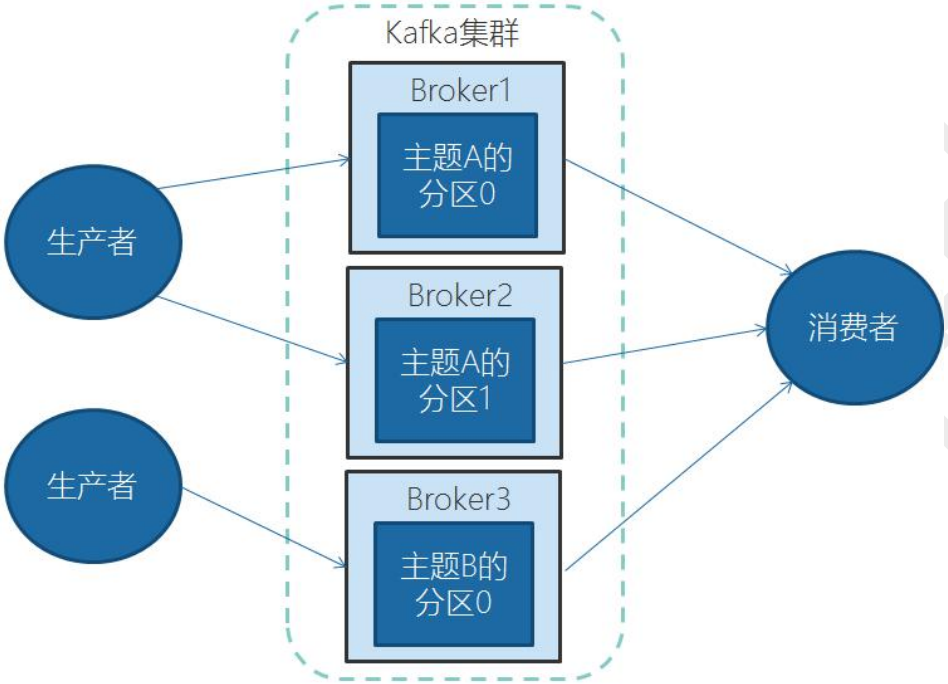
CPU

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

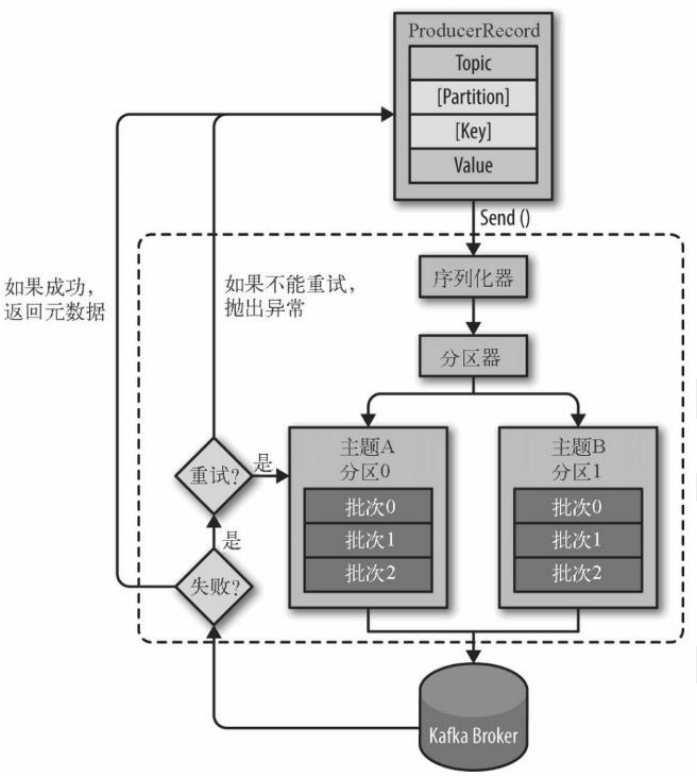
新版本特点：poll(Duration)这个版本修改了这样的设计，会把元数据获取也计入整个超时时间（更加的合理）

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

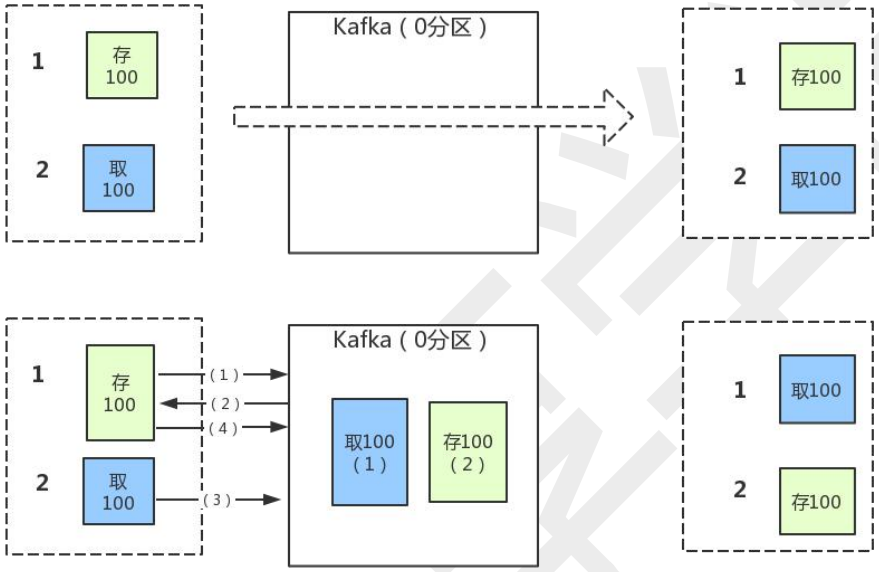
max.request.size

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。

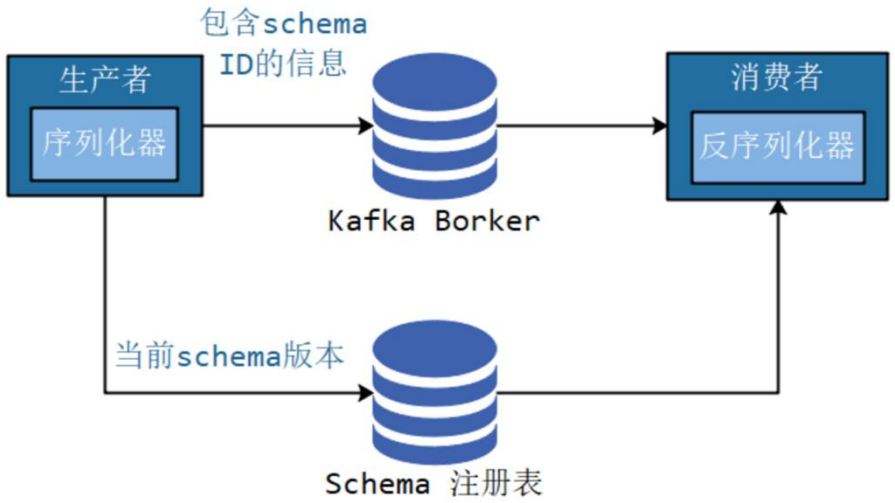
解决这个问题，可以考虑使用自带格式描述以及语言无关的序列化框架。比如 Protobuf，或者 Kafka 官方推荐的 Apache Avro。

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

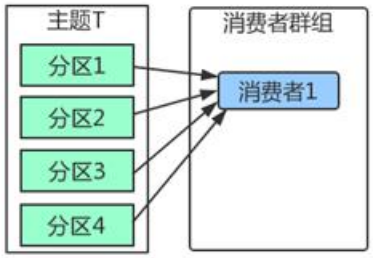
消费者的入门

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。

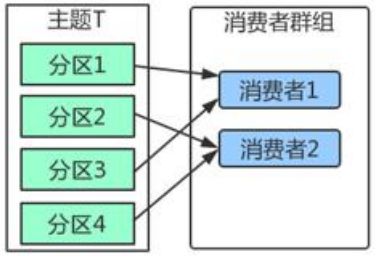
（买单的故事，群组，消费者的一群人， 消费者：买单的，分区：一笔单，一笔单能被买单一次，当然一个消费者可以买多个单，如果有一个消费者挂掉了<跑单了>，另外的消费者接上）

消费者群组

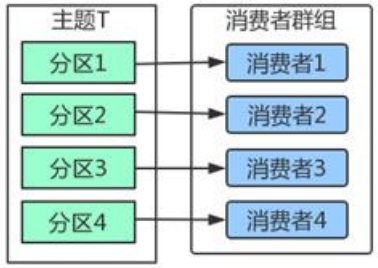
Kafka 里消费者从属于消费者群组，一个群组里的消费者订阅的都是同一个主题，每个消费者接收主题一部分分区的消息。



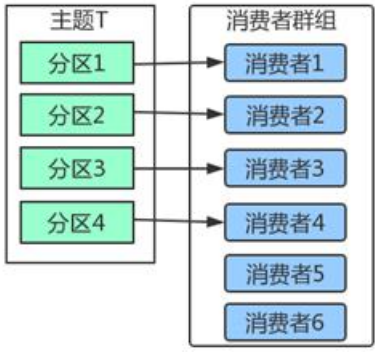
如上图，主题 T 有 4 个分区，群组中只有一个消费者，则该消费者将收到主题 T1 全部 4 个分区的消息。



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。

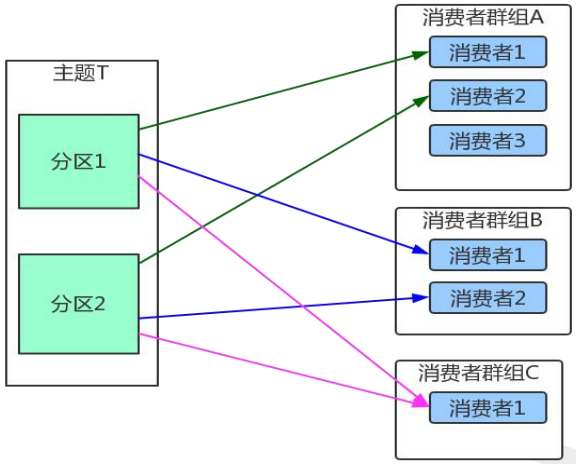


但是，当我们增加更多的消费者，超过了主题的分区数量，就会有一部分的消费者被闲置，不会接收到任何消息。

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。

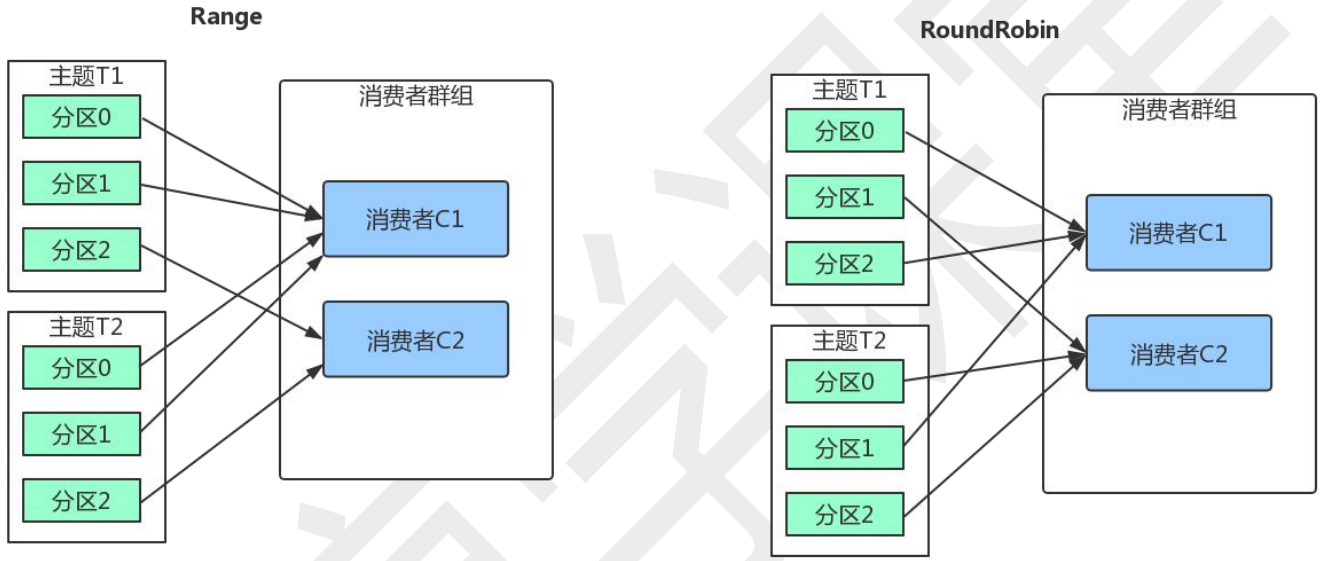
Range

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。

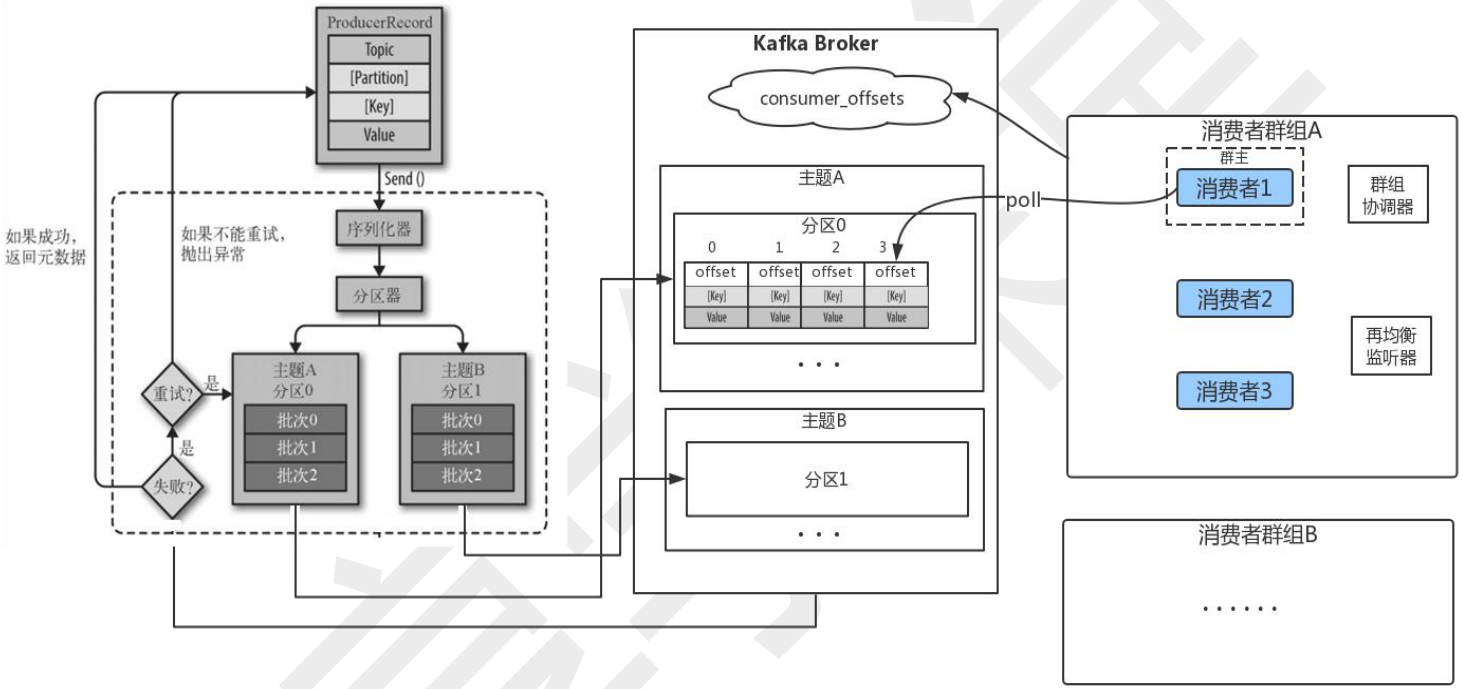
receive.buffer.bytes 和 send.buffer.bytes

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

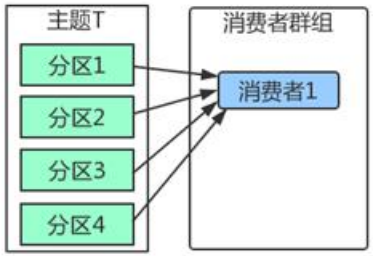
消费者的含义，同一般消息中间件中消费者的概念。在高并发的情况下，生产者产生消息的速度是远大于消费者消费的速度，单个消费者很可能会负担不起，此时有必要对消费者进行横向伸缩，于是我们可以使用多个消费者从同一个主题读取消息，对消息进行分流。

（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）

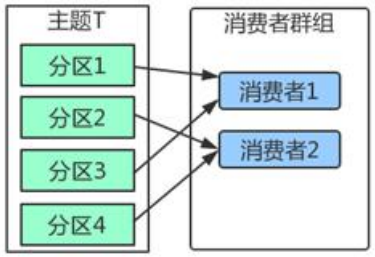


消费者群组

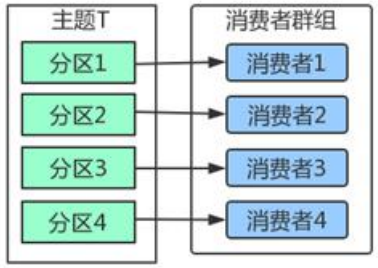
Kafka 里消费者从属于消费者群组，一个群组里的消费者订阅的都是同一个主题，每个消费者接收主题一部分分区的消息。



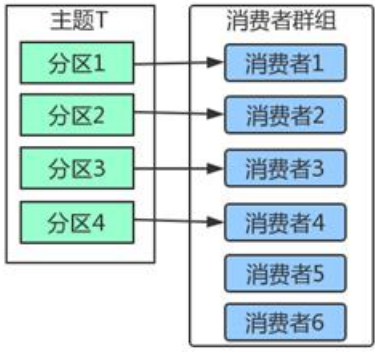
如上图，主题 T 有 4 个分区，群组中只有一个消费者，则该消费者将收到主题 T1 全部 4 个分区的消息。



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。

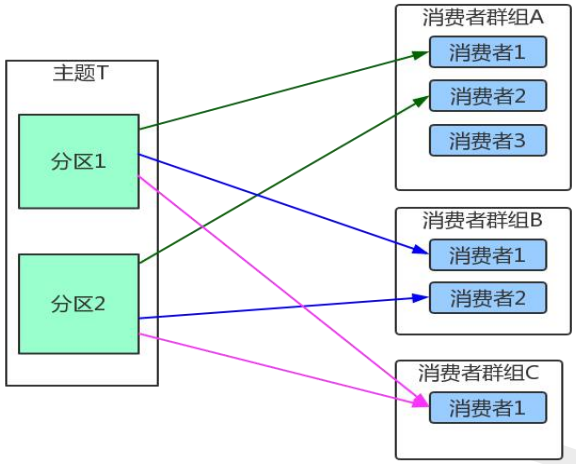


但是，当我们增加更多的消费者，超过了主题的分区数量，就会有一部分的消费者被闲置，不会接收到任何消息。

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。

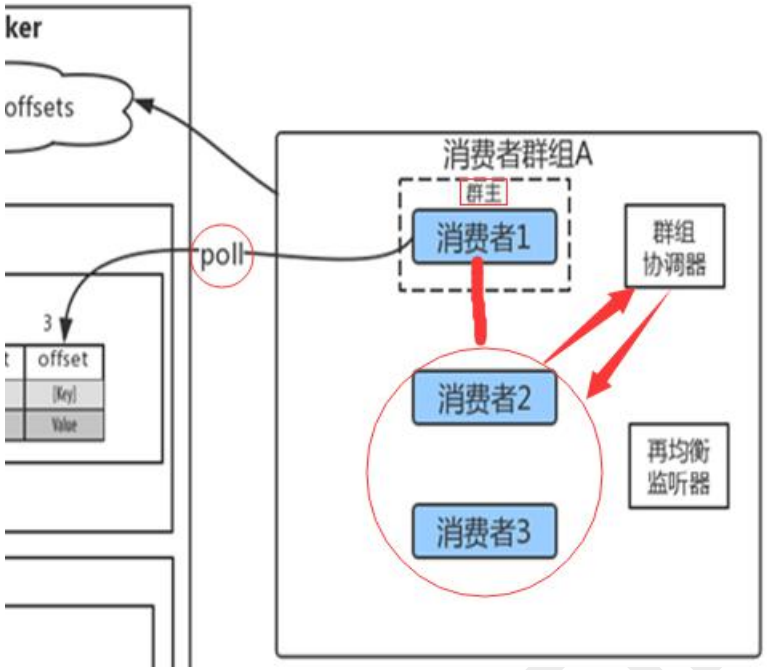
消费者中的核心概念

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

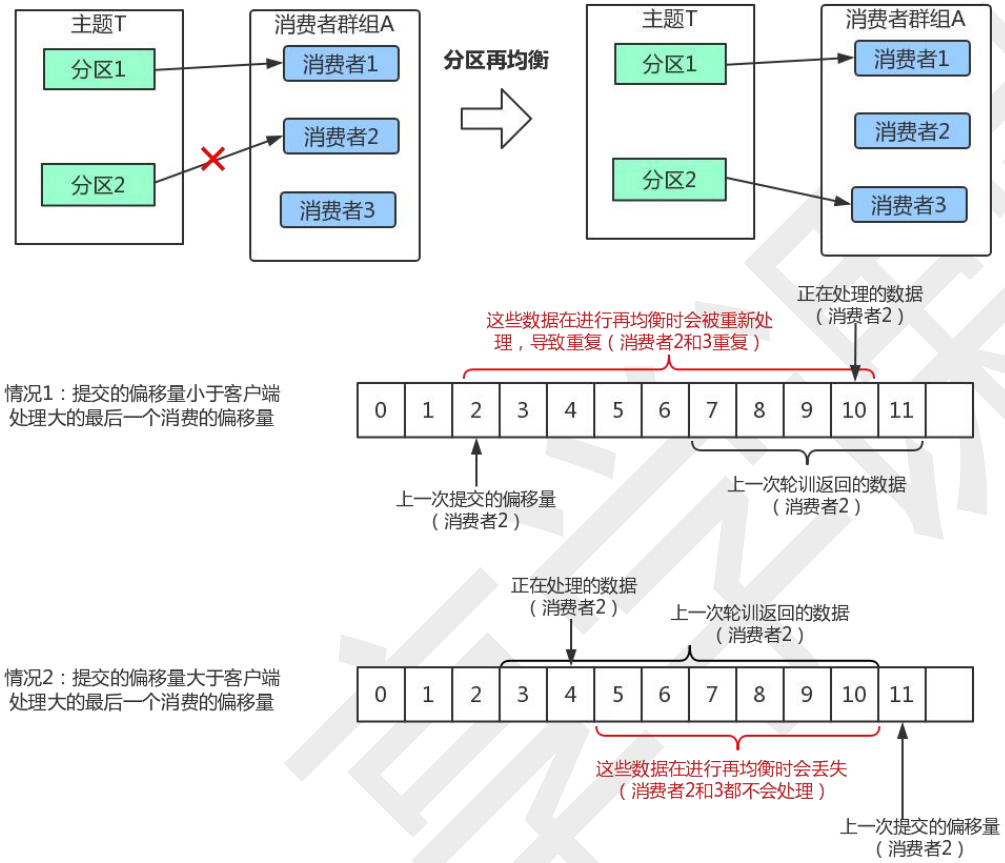
当我们调用 poll 方法的时候，broker 返回的是生产者写入 Kafka 但是还没有被消费者读取过的记录，消费者可以使用 Kafka 来追踪消息在分区里的位置，我们称之为**偏移量**。消费者更新自己读取到哪个消息的操作，我们称之为**提交**。

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

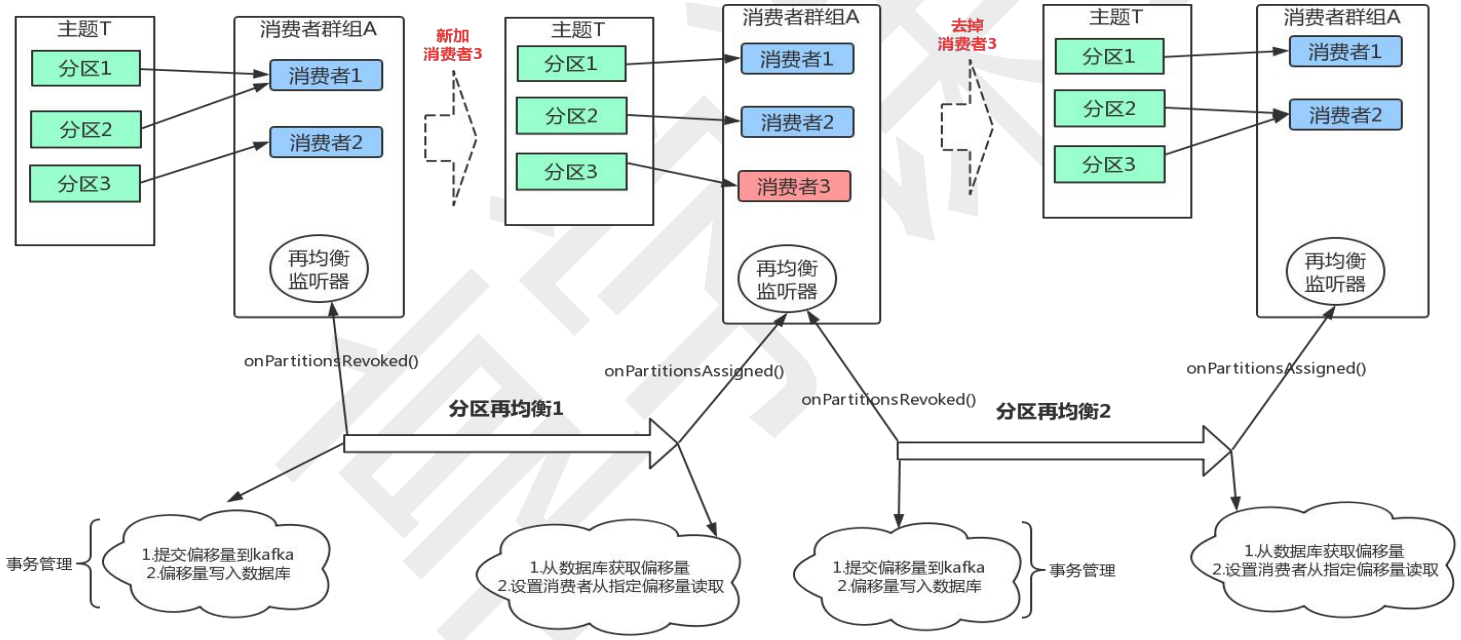
在我们前面的提交中，提交偏移量的频率与处理消息批次的频率是一样的。但如果想要更频繁地提交该怎么办?

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。