Kafka

第一章 简介

Kafka是什么

Apache Kafka 是一个分布式的流处理平台。它具有以下特点：

- 支持消息的发布和订阅，类似于 RabbtMQ、ActiveMQ 等消息队列；
- 支持数据实时处理；
- 能保证消息的可靠性投递；
- 支持消息的持久化存储，并通过多副本分布式的存储方案来保证消息的容错；
- 高吞吐率，单Broker可以轻松处理数千个分区以及每秒百万级的消息量。

基本概念

2.1 Messages And Batches

Kafka 的基本数据单元被称为 message(消息)，为减少网络开销，提高效率，多个消息会被放入 `同一批次` (Batch) 中后再写入。

2.2 Topics And Partitions

Kafka 的消息通过 `Topics(主题) 进行分类`，一个主题可以被分为 `若干个 Partitions(分区)`，一个分区就是一个 `提交日志 (commit log)`。消息以 `追加` 的方式写入分区，然后以先入先出的 `顺序读取`。Kafka 通过分区来实现数据的冗余和伸缩性，`分区可以分布在不同的服务器上`，这意味着一个 Topic 可以横跨多个服务器，以提供比单个服务器更强大的性能。

由于一个 Topic 包含多个分区，因此无法在整个 Topic 范围内保证消息的顺序性，但可以保证消息在单个分区内的顺序性。

2.3 Producers And Consumers

1. 生产者

`生产者负责创建消息`。一般情况下，生产者把消息均衡地分布到主题的所有分区上，而并不关心消息会被写到哪个分区。如果我们想要把消息写到指定的分区，可以通过 `自定义分区器` 来实现。

2. 消费者

`消费者是消费者群组的一部分，消费者负责消费消息`。`消费者可以订阅一个或者多个主题，并按照消息生成的顺序来读取它们`。消费者通过检查消息的 `偏移量 (offset)` 来区分读取过的消息。偏移量是一个不断递增的数值，在创建消息时，Kafka 会把它添加到其中，在给定的分区里，每个消息的偏移量都是唯一的。`消费者把每个分区最后读取的偏移量保存在 Zookeeper 或 Kafka 上`，如果消费者关闭或者重启，它还可以重新获取该偏移量，以保证读取状态不会丢失。

`一个分区只能被同一个消费者群组里面的一个消费者读取，但可以被不同消费者群组中所组成的多个消费者共同读取。多个消费者群组中消费者共同读取同一个主题时，彼此之间互不影响。`

2.4 Brokers And Clusters

一个独立的 Kafka 服务器被称为 Broker。Broker 接收来自生产者的消息，为消息设置偏移量，并提交消息到磁盘保存。Broker 为消费者提供服务，对读取分区的请求做出响应，返回已经提交到磁盘的消息。

Broker 是集群 (Cluster) 的组成部分。每一个集群都会选举出一个 Broker 作为集群控制器 (Controller)，集群控制器负责管理工作，包括将分区分配给 Broker 和监控 Broker。

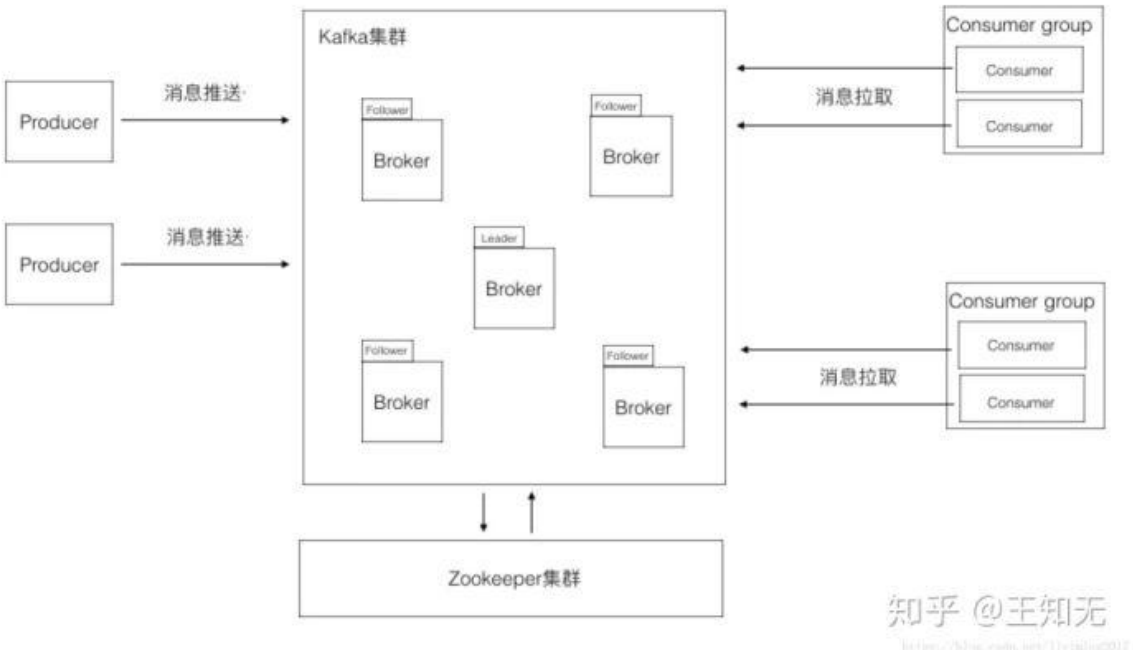
在集群中，一个分区 (Partition) 从属一个 Broker，该 Broker 被称为 分区的首领 (Leader)。一个分区可以分配给多个 Brokers，这个时候会发生分区复制。这种复制机制为分区提供了消息冗余，如果有一个 Broker 失效，其他 Broker 可以接管领导权。

第二章 核心概念

集群结构

实际上kafka的结构图是有些区别的，现在我们看下面的图：

Kafka集群结构图



producer和consumer想必大家都很熟悉，一个生产消息，一个消费掉消息。这里就不再做太多解释。

此图和第一张图可以看到有几个区别：

1、多了zookeeper集群，通过前几章的学习我们已经知道kafka是配合zookeeper进行工作的。

2、kafka集群中可以看到有若干个Broker，其中一个broker是leader，其他的broker是follower

3、consumer外面包裹了一层Consumer group。

我们先讲解一下Broker和consumer group的概念，以及Topic。

Broker

一个Borker就是Kafka集群中的一个实例，或者说是一个服务单元。连接到同一个zookeeper的多个broker实例组成kafka的集群。在若干个broker中会有一个broker是leader，其余的broker为follower。leader在集群启动时候选举出来，负责和外部的通讯。当leader死掉的时候，follower们会再次通过选举，选择出新的leader，确保集群的正常工作。

Consumer Group

Kafka和其它消息系统有一个不一样的设计，在consumer之上加了一层group。同一个group的consumer可以并行消费同一个topic的消息，但是同group的consumer，不会重复消费。这就好比多个consumer组成了一个团队，一起干活，当然干活的速度就上来了。group中的consumer是如何配合协调的，其实和topic的分区相关联，后面我们会详细论述。

如果同一个topic需要被多次消费，可以通过设立多个consumer group来实现。每个group分别消费，互不影响。

通过本节学习，我们从全局的层面了解了kafka的结构，接下来我们会深入到kafka内部，来看看它是怎么工作的。

Topic

kafka中消息订阅和发送都是基于某个topic。比如有个topic叫做NBA赛事信息，那么producer会把NBA赛事信息的消息发送到此topic下面。所有订阅此topic的consumer将会拉取到此topic下的消息。Topic就像一个特定主题的收件箱，producer往里丢，consumer取走。

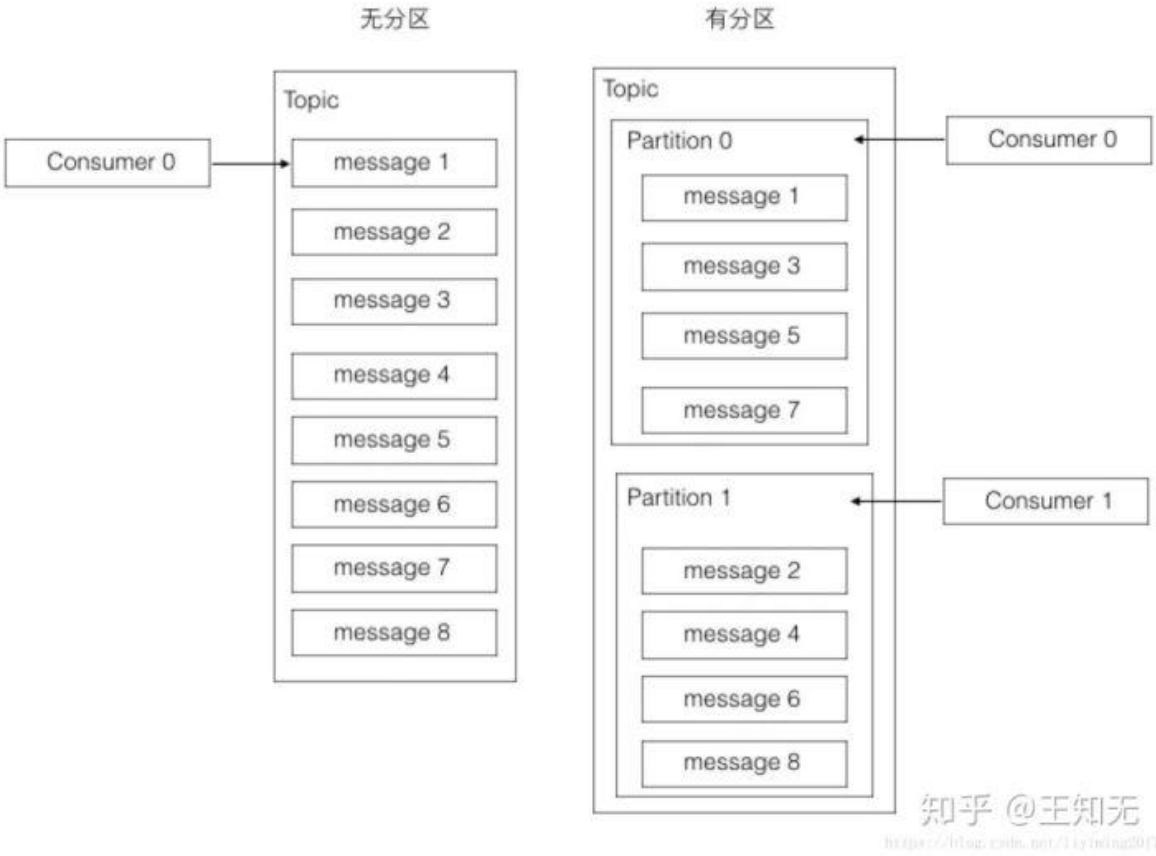
Kafka核心概念简介

kafka采用 分区（Partition） 的方式，使得消费者能够做到 并行消费 ，从而大大提高了自己的吞吐能力。同时为了实现高可用，每个分区又有若干份副本（Replica），这样在某个broker挂掉的情况下，数据不会丢失。

接下来我们详细分析kafka是如何基于Partition和Replica工作的。

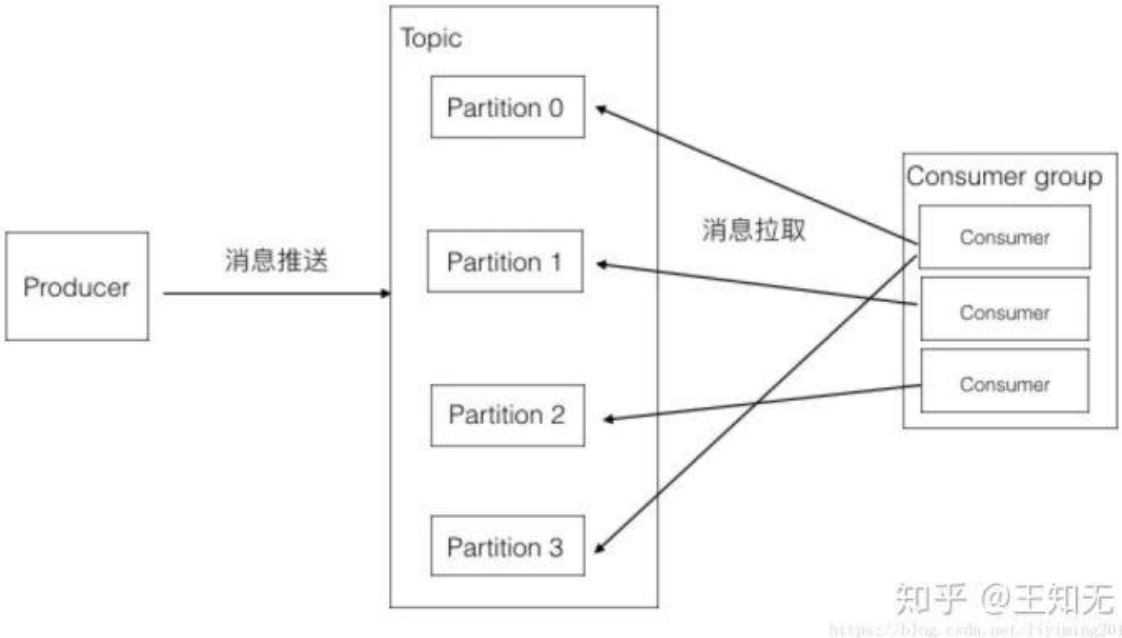
分区（Partition）

大多数消息系统，同一个topic下的消息，存储在一个队列。分区的概念就是把这个队列划分为若干个小队列，每一个小队列就是一个分区，如下图：



这样做的好处是什么呢？其实从上图已经可以看出来。无分区时，一个topic只有一个消费者在消费这个消息队列。采用分区后，如果有两个分区，最多两个消费者同时消费，消费的速度肯定会更快。如果觉得不够快，可以加到四个分区，让四个消费者并行消费。分区的设计大大的提升了kafka的吞吐量！！

我们再结合下图继续讲解Partition。



此图包含如下几个知识点：

1、一个partition只能被同组的一个consumer消费（图中只会有一个箭头指向一个partition）

2、同一个组里的一个consumer可以消费多个partition（图中第一个consumer消费Partition 0和3）

3、消费效率最高的情况是partition和consumer数量相同。这样确保每个consumer专职负责一个partition。

4、consumer数量不能大于partition数量。由于第一点的限制，当consumer多于partition时，就会有consumer闲置。

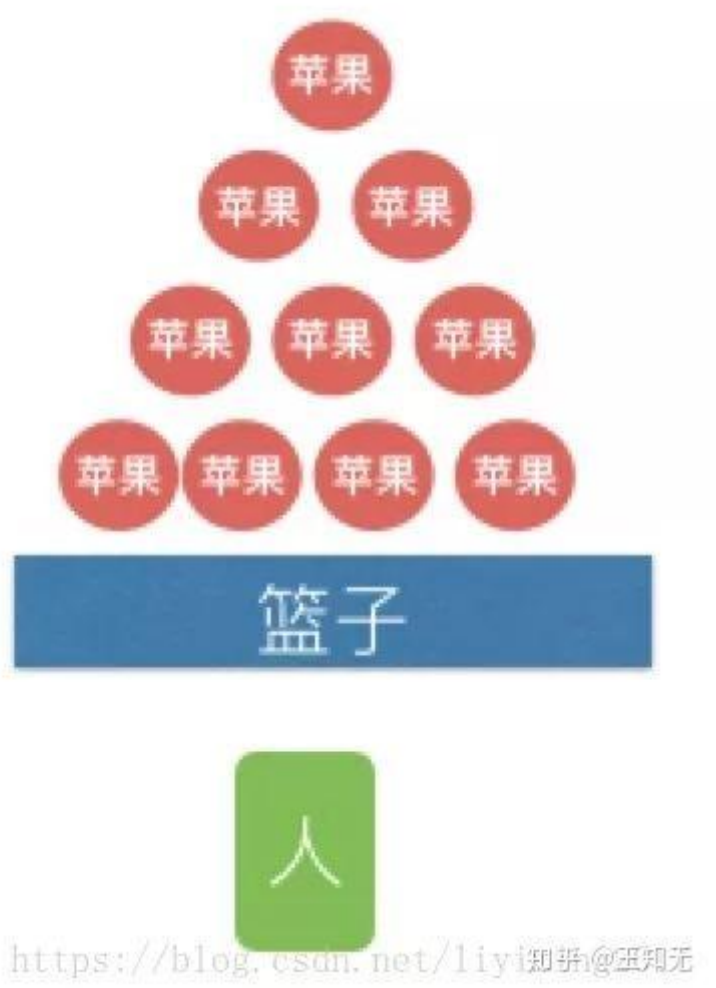
5、consumer group可以认为是一个订阅者的集群，其中的每个consumer负责自己所消费的分区

为了加深理解，我举个吃苹果的例子。

问题：有一篮子苹果，你如何把这一篮子苹果尽可能快的吃完?

办法一：

我一个人，一个一个苹果吃，如下图。这样显然很慢，我吃完一个才能拿下一个。

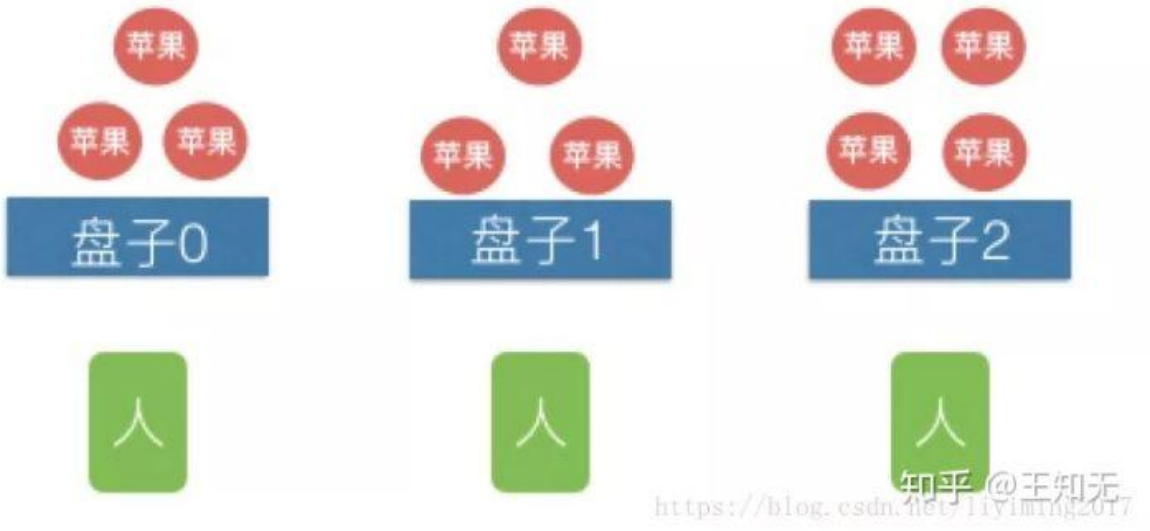


办法二：

我再找两个人来一块吃，第一个人拿走一个去吃，然后第二个人拿一个去吃，接着第三个人拿一个去吃，如此循环。速度肯定快了，但是三个人还是会排队等待。三个人排队时间可能很短，但是如果叫了100个人帮忙吃呢？会有大量时间消耗在排队上。

办法三：

我还是找两个人来一块吃，但我把苹果提前分到三个盘子里，每人分一个盘子，自己吃自己的，这样不但能三个人同时吃苹果，还无须排队。速度显然是最快的。



办法三正是kafka所采用的设计方式，盘子就是partition，每个人就是一个consumer，每个苹果就是一条message。办法三每个盘子中苹果的消费是有序的，而办法二的消费是完全无序的。

相信通过这个例子你一定能充分理解partition的概念，以及为什么kafka会如此设计。

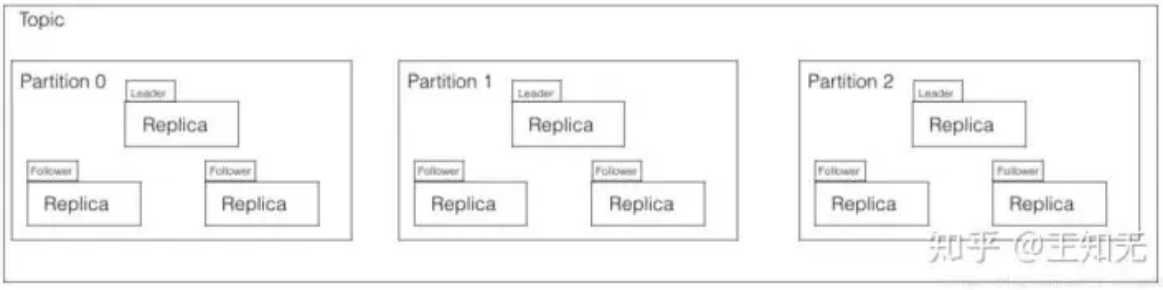
关于partition暂时说到这里，接下来介绍副本。

副本（Replica）

提到副本，肯定就会想到正本。`副本是正本的拷贝`。在kafka中，`正本和副本都称之为副本（Repalica）`，但存在leader和follower之分。活跃的称之为leader，其他的是follower。

每个分区的数据都会有多份副本，以此来保证Kafka的高可用。

Topic、partition、replica的关系如下图：

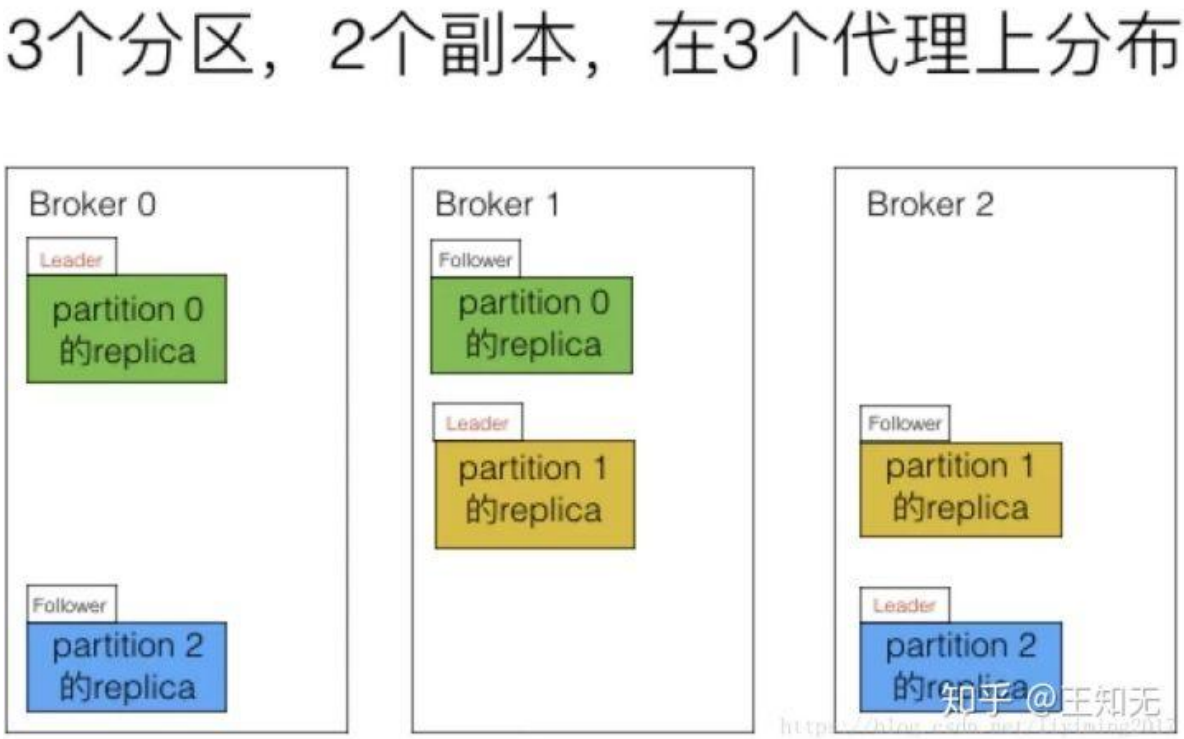


topic下会划分多个partition，每个partition都有自己的replica，其中只有一个是leader replica，其余的是follower replica。

消息进来的时候会先存入leader replica，然后从leader replica复制到follower replica。只有复制全部完成时，consumer才可以消费此条消息。这是为了确保意外发生时，数据可以恢复。consumer的消费也是从leader replica读取的。

由此可见，leader replica做了大量的工作。所以如果不同partition的leader replica在kafka集群的broker上分布不均匀，就会造成负载不均衡。

kafka通过轮询算法保证leader replica是均匀分布在多个broker上。如下图。



可以看到每个partition的leader replica均匀的分布在三个broker上，follower replica也是均匀分布的。关于Replica，有如下知识点：

1、Replica均匀分配在Broker上，同一个partition的replica不会在同一个borker上

2、同一个partition的Replica数量不能多于broker数量。多个replica为了数据安全，一台server存多个replica没有意义。server挂掉，上面的副本都要挂掉。

3、分区的leader replica均衡分布在broker上。此时集群的负载是均衡的。这就叫做分区平衡

分区平衡是个很重要的概念，接下来我们就来讲解分区平衡。

分区平衡

在讲分区平衡前，先讲几个概念：

1、AR： assigned replicas，已分配的副本。每个partition都有自己的AR列表，里面存储着这个partition最初分配的所有replica。注意AR列表不会变化，除非增加分区。

2、PR（优先replica）：AR列表中的第一个replica就是优先replica，而且永远是优先replica。最初，优先replica和leader replica是同一个replica。

3、`ISR：in sync replicas，同步副本`。每个partition都有自己的ISR列表。ISR是会根据同步情况动态变化的。

最初ISR列表和AR列表是一致的，但由于某个节点死掉，或者某个节点的follower replica落后leader replica太多，那么该节点就会被从ISR列表中移除。此时，ISR和AR就不再一致

接下来我们通过一个例子来理解分区平衡。

1、根据以上信息，一个拥有3个replica的partition，最初是下图的样子。

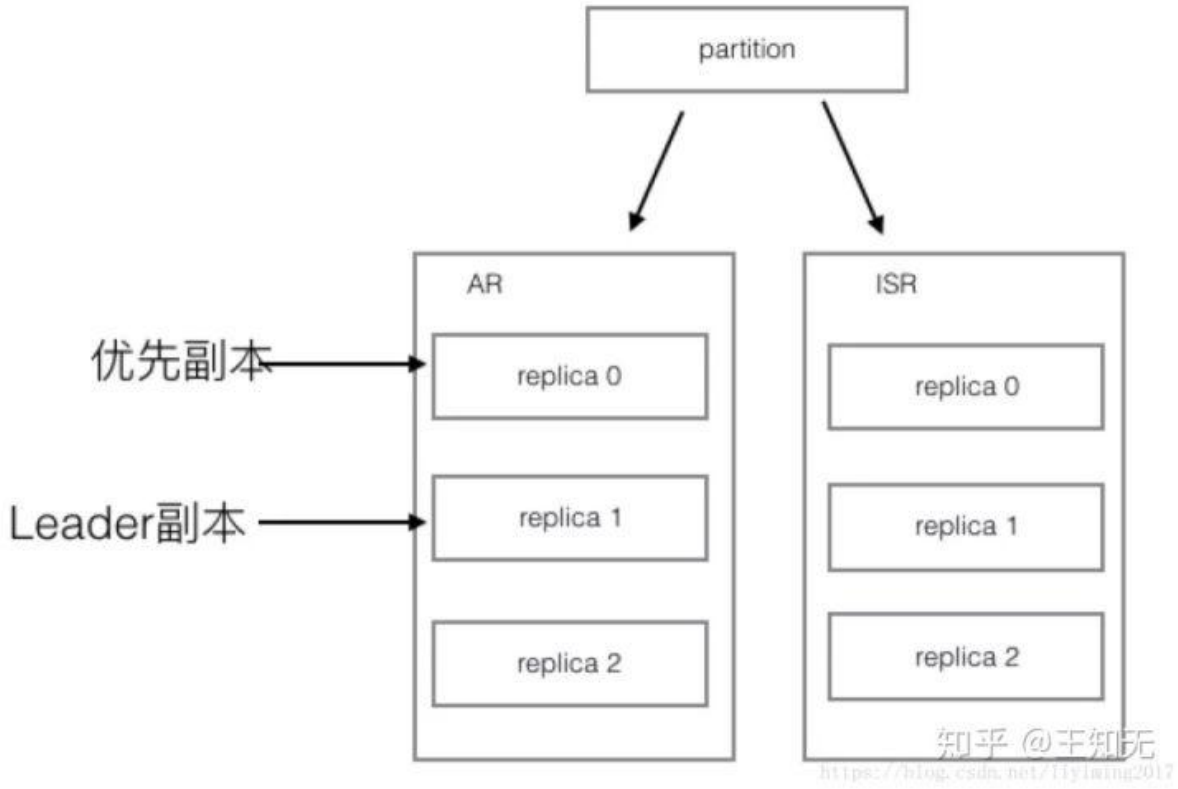
可以看到AR和ISR保持一致，并且初始时刻，优先副本和leader副本都指向replica 0.

2、接下来，replica 0所在的机器下线了，那么情况会变成如下图所示：

可以看到replica 0已经从ISR中移除掉了。同时，由于重新选举，leader副本变成了replica 1，而优先副本还是replica 0。优先副本是不会改变的。

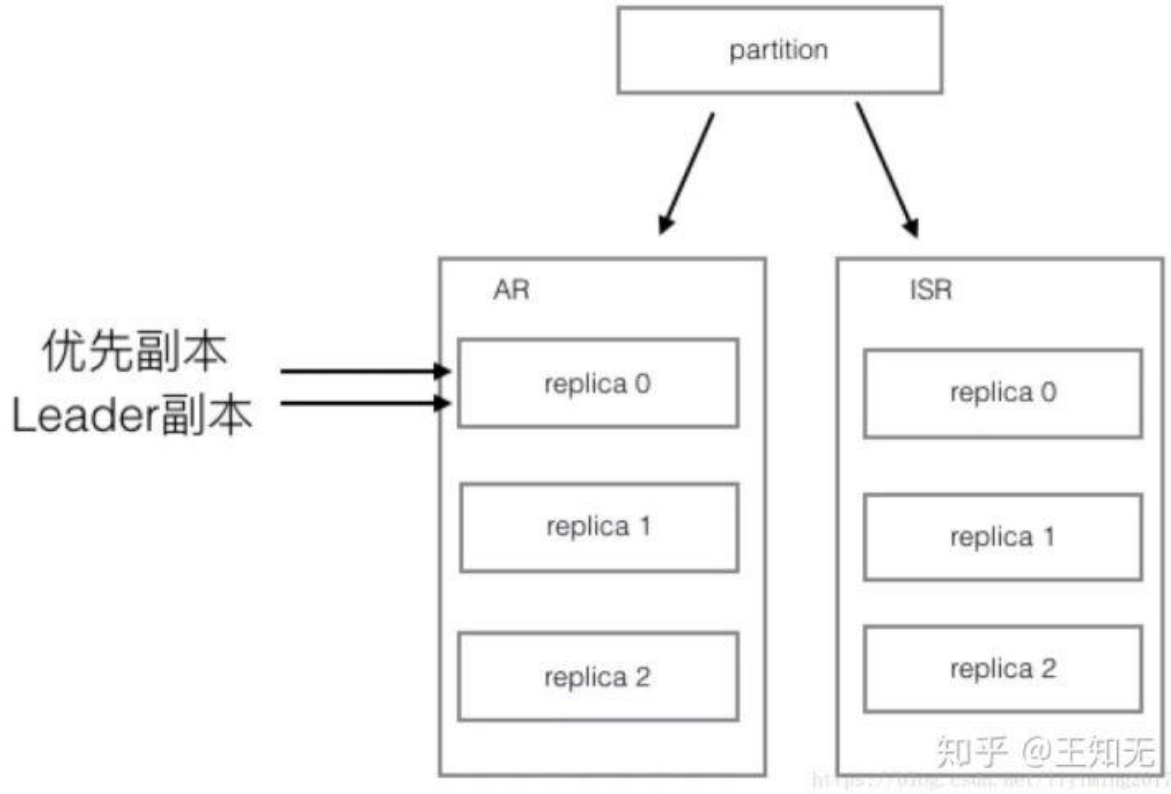
由于最初时，leader副本在broker均匀分布，分区是平衡的。但此时，由于此partition的leader副本换成了另外一个，所以此时分区平衡已经被破坏。

3、replica 0所在的机器修复了，又重新上线，情况如下图：



可以看到replica 0重新回到ISR列表中，不过此时他没能恢复leader的身份。只能作为follower当一名小弟。此时分区依旧是不平衡的。那是否意味着分区永远都会不平衡下去呢？不是的。

4、kafka会定时触发分区平衡操作，也可以主动触发分区平衡。这就是所谓的分区平衡操作，操作完后如下图。

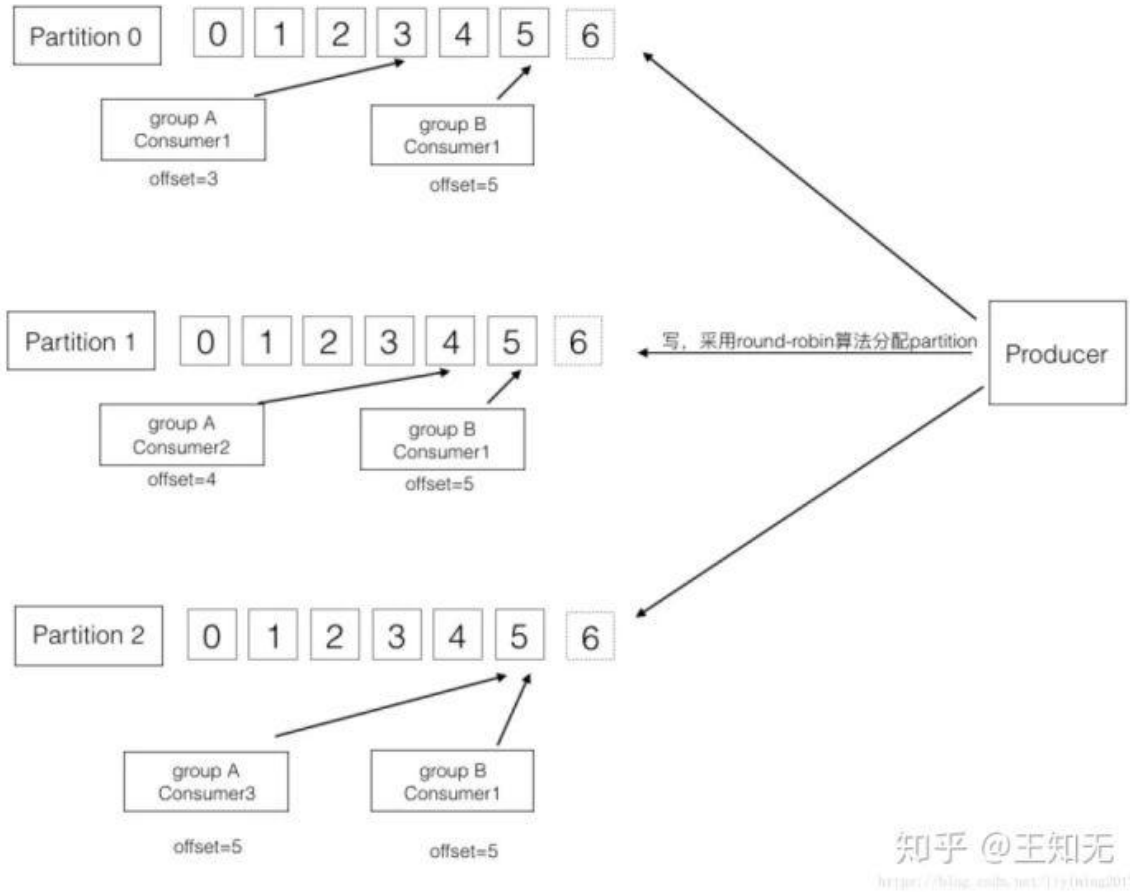


可以看到此时leader副本通过选举，会重新变回来replica 0，因为replica 0是优先副本，其实优先的含义就是选择leader时被优先选择。这样整个分区又回到了初始状态，而初始时，leader副本是均匀分布的。此时已经分区平衡了。

由此可见，分区平衡操作就是使leader副本和优先副本保持一致的操作。可以把优先副本理解为分区的平衡状态位，平衡操作就是让leader副本归位。

Partition的读和写

通过之前的学习，我们知道topic下划分了多个partition，消息的生产和消费最终都是发生在partition之上。下图是一个三个partition的topic的读写示意。



我们先看右边的producer，可以看到写的时候，采用round-robin算法，轮询往每个partition写入。

而在消费者端，每个consumer都维护一个offset值，指向的是它所消费到的消息坐标。

我们先看group A的三个consumer，他们分别独立消费不同的三个partition。每个consumer维护了自己的offset。

我们再看group B，可以看到两个group是并行消费整个topic，同一条消息会被不同group消费到。

此处有如下知识点：

1、每个partition都是有序的不可变的。

2、Kafka可以保证partition的消费顺序，但不能保证topic消费顺序。

3、无论消费与否，保留周期默认两天（可配置）。

4、每个consumer维护的唯一元数据是offset，代表消费的位置，一般线性向后移动。

5、consumer也可以重置offset到之前的位置，可以以任何顺序消费，不一定线性后移。

第三章 Kafka副本机制

Kafka集群

Kafka 使用 Zookeeper 来维护集群成员 (brokers) 的信息。每个 broker 都有一个唯一标识 `broker.id`，用于标识自己在集群中的身份，可以在配置文件 `server.properties` 中进行配置，或者由程序自动生成。下面是 Kafka brokers 集群自动创建的过程：

- 每一个 broker 启动的时候，它会在 Zookeeper 的 `/brokers/ids` 路径下创建一个 `临时节点`，并将自己的 `broker.id` 写入，从而将自身注册到集群；
- 当有多个 broker 时，所有 broker 会竞争性地在 Zookeeper 上创建 `/controller` 节点，由于 Zookeeper 上的节点不会重复，所以必然只会有一个 broker 创建成功，此时该 broker 称为 controller broker。它除了具备其他 broker 的功能外，`还负责管理主题分区及其副本的状态`。
- 当 broker 出现宕机或者主动退出从而导致其持有的 Zookeeper 会话超时时，`会触发注册在 Zookeeper 上的 watcher 事件`，此时 Kafka 会进行相应的容错处理；如果宕机的是 controller broker 时，还会触发新的 controller 选举。

副本机制

为了保证高可用，kafka 的分区`是多副本的`，如果一个副本丢失了，那么`还可以从其他副本中获取分区数据`。但是这`要求对应副本的数据必须是完整的`，这是 Kafka `数据一致性的基础`，所以才需要使用 `controller broker` 来进行专门的管理。下面将详解介绍 Kafka 的副本机制。

2.1 分区和副本

Kafka 的主题被分为多个分区，分区是 Kafka 最基本的存储单位。每个分区可以有多个副本 (可以在创建主题时使用 `replication-factor` 参数进行指定)。其中一个副本是首领副本 (Leader replica)，所有的事件都直接发送给首领副本；其他副本是跟随者副本 (Follower replica)，需要通过复制来保持与首领副本数据一致，当首领副本不可用时，其中一个跟随者副本将成为新首领。

2.2 ISR机制

每个分区都有一个 `ISR(in-sync Replica) 列表`，`用于维护所有同步的、可用的副本`。首领副本必然是同步副本，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副本：

- 与 Zookeeper 之间有一个活跃的会话，即必须定时向 Zookeeper 发送心跳；
- 在规定的时间内从首领副本那里低延迟地获取过消息。

如果副本不满足上面条件的话，就会被从 ISR 列表中移除，直到满足条件才会被再次加入。

这里给出一个主题创建的示例：使用 `--replication-factor` 指定副本系数为 3，创建成功后使用 `--describe` 命令可以看到分区 0 的有 0,1,2 三个副本，且三个副本都在 ISR 列表中，其中 1 为首领副本。

2.3 不完全的首领选举

对于副本机制，在 broker 级别有一个可选的配置参数 `unclean.leader.election.enable`，默认值为 fasle，代表禁止不完全的首领选举。这是针对当首领副本挂掉且 ISR 中没有其他可用副本时，是否允许某个不完全同步的副本成为首领副本，这可能会导致数据丢失或者数据不一致，在某些对数据一致性要求较高的场景 (如金融领域)，这可能无法容忍的，所以其默认值为 false，如果你能够允许部分数据不一致的话，可以配置为 true。

2.4 最少同步副本

ISR 机制的另外一个相关参数是 `min.insync.replicas`，可以在 broker 或者主题级别进行配置，代表 ISR 列表中至少要有几个可用副本。这里假设设置为 2，那么当可用副本数量小于该值时，就认为整个分区处于不可用状态。此时客户端再向分区写入数据时候就会抛出异常 `org.apache.kafka.common.errors.NotEnoughReplicasExceptoin: Messages are rejected since there are fewer in-sync replicas than required。`

2.5 发送确认

Kafka 在生产者上有一个可选的参数 ack，该参数指定了必须要有多少个分区副本收到消息，生产者才会认为消息写入成功：

- **acks=0** ： `消息发送出去就认为已经成功了，不会等待任何来自服务器的响应；`
- **acks=1** ： `只要集群的首领节点收到消息，生产者就会收到一个来自服务器成功响应；`
- **acks=all** ： `只有当所有参与复制的节点全部收到消息时，生产者才会收到一个来自服务器的成功响应。`

数据请求

3.1 元数据请求机制

在所有副本中，只有领导副本才能进行消息的读写处理。由于不同分区的领导副本可能在不同的 broker 上，如果某个 broker 收到了一个分区请求，但是该分区的领导副本并不在该 broker 上，那么它就会向客户端返回一个 `Not a Leader for Partition` 的错误响应。 为了解决这个问题，Kafka 提供了元数据请求机制。

首先集群中的每个 broker 都会缓存所有主题的分区副本信息，客户端会定期发送发送元数据请求，然后将获取的元数据进行缓存。定时刷新元数据的时间间隔可以通过为客户端配置 `metadata.max.age.ms` 来进行指定。有了元数据信息后，客户端就知道了领导副本所在的 broker，之后直接将读写请求发送给对应的 broker 即可。

如果在定时请求的时间间隔内发生的分区副本的选举，则意味着原来缓存的信息可能已经过时了，此时还有可能会收到 `Not a Leader for Partition` 的错误响应，这种情况下客户端会再次求发出元数据请求，然后刷新本地缓存，之后再去正确的 broker 上执行对应的操作，过程如下图：

3.2 数据可见性

需要注意的是，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到，为了保证数据一致性，只有被所有同步副本 (ISR 中所有副本) 都保存了的数据才能被客户端读取到。

3.3 零拷贝

Kafka 所有数据的写入和读取都是通过零拷贝来实现的。传统拷贝与零拷贝的区别如下：

传统模式下的四次拷贝与四次上下文切换

以将磁盘文件通过网络发送为例。传统模式下，一般使用如下伪代码所示的方法先将文件数据读入内存，然后通过 Socket 将内存中的数据发送出去。

```
buffer = File.read
Socket.send(buffer)
```

这一过程实际上发生了四次数据拷贝。首先通过系统调用将文件数据读入到内核态 Buffer（DMA 拷贝），然后应用程序将内存态 Buffer 数据读入到用户态 Buffer（CPU 拷贝），接着用户程序通过 Socket 发送数据时将用户态 Buffer 数据拷贝到内核态 Buffer（CPU 拷贝），最后通过 DMA 拷贝将数据拷贝到 NIC Buffer。同时，还伴随着四次上下文切换，如下图所示：

sendfile和transferTo实现零拷贝

Linux 2.4+ 内核通过 `sendfile` 系统调用，提供了零拷贝。数据通过 DMA 拷贝到内核态 Buffer 后，直接通过 DMA 拷贝到 NIC Buffer，无需 CPU 拷贝。这也是零拷贝这一说法的来源。除了减少数据拷贝外，因为整个读文件到网络发送由一个 `sendfile` 调用完成，整个过程只有两次上下文切换，因此大大提高了性能。零拷贝过程如下图所示：

从具体实现来看，Kafka 的数据传输通过 TransportLayer 来完成，其子类 `PlaintextTransportLayer` 的 `transferFrom` 方法通过调用 Java NIO 中 FileChannel 的 `transferTo` 方法实现零拷贝，如下所示：

```
@Override
public long transferFrom(FileChannel fileChannel, long position, long count) throws IOException {
      return fileChannel.transferTo(position, count, socketChannel);
}
```

注： `transferTo` 和 `transferFrom` 并不保证一定能使用零拷贝。实际上是否能使用零拷贝与操作系统相关，如果操作系统提供 `sendfile` 这样的零拷贝系统调用，则这两个方法会通过这样的系统调用充分利用零拷贝的优势，否则并不能通过这两个方法本身实现零拷贝。

物理存储

4.1 分区分配

在创建主题时，Kafka 会首先决定如何在 broker 间分配分区副本，它遵循以下原则：

- 在所有 broker 上均匀地分配分区副本；
- 确保分区的每个副本分布在不同的 broker 上；
- 如果使用了 `broker.rack` 参数为 broker 指定了机架信息，那么会尽可能的把每个分区的副本分配到不同机架的 broker 上，以避免一个机架不可用而导致整个分区不可用。

基于以上原因，如果你在一个单节点上创建一个 3 副本的主题，通常会抛出下面的异常：

```
Error while executing topic command : org.apache.kafka.common.errors.InvalidReplicationFactor
Exception: Replication factor: 3 larger than available brokers: 1.
```

4.2 分区数据保留规则

保留数据是 Kafka 的一个基本特性， 但是 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 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。分别对应以下四个参数：

- `log.retention.bytes` ： 删除数据前允许的最大数据量；默认值-1，代表没有限制；
- `log.retention.ms`： 保存数据文件的毫秒数，如果未设置，则使用 `log.retention.minutes` 中的值，默认为 null；
- `log.retention.minutes`： 保留数据文件的分钟数，如果未设置，则使用 `log.retention.hours` 中的值，默认为 null；
- `log.retention.hours`： 保留数据文件的小时数，默认值为 168，也就是一周。

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 Kafka 把分区分成若干个片段，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。如果按照默认值保留数据一周，而且每天使用一个新片段，那么你就会看到，在每天使用一个新片段的同时会删除一个最老的片段，所以大部分时间该分区会有 7 个片段存在。

4.3 文件格式

通常保存在磁盘上的数据格式与生产者发送过来消息格式是一样的。 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 (格式如下所示)，然后保存到磁盘上。之后消费者读取后再自己解压这个包装消息，获取每条消息的具体信息。

第四章 生产者

生产者发送消息的过程

首先介绍一下 Kafka 生产者发送消息的过程：

- Kafka 会将发送消息包装为 ProducerRecord 对象， ProducerRecord 对象包含了 `目标主题和要发送的内容` ， `同时还可以指定键和分区` 。在发送 ProducerRecord 对象前， `生产者会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` 。如果之前已经在 ProducerRecord 对象里指定了分区，那么分区器就不会再做任何事情。如果没有指定分区，那么分区器会根据 ProducerRecord 对象的键来选择一个分区，紧接着，这条记录被添加到一个记录批次里，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，如果达到指定的重试次数后还没有成功，则直接抛出异常，不再重试。

创建生产者

2.1 项目依赖

本项目采用 Maven 构建，想要调用 Kafka 生产者 API，需要导入 `kafka-clients` 依赖，如下：

```
<dependency>
      <groupId>org.apache.kafka</groupId>
      <artifactId>kafka-clients</artifactId>
      <version>2.2.0</version>
</dependency>
```

2.2 创建生产者

创建 Kafka 生产者时，以下三个属性是必须指定的：

- **bootstrap.servers**： 指定 broker 的地址清单，清单里不需要包含所有的 broker 地址，生产者会从给定的 broker 里查找 broker 的信息。不过建议至少要提供两个 broker 的信息作为容错；
- **key.serializer**： 指定键的序列化器；
- **value.serializer**： 指定值的序列化器。

```
package producer;

import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.common.serialization.StringSerializer;

import java.util.Properties;
```

```
9
10  /**
11   * Project:  BigDataCode
12   * Create date:  2023/5/17
13   * Created by fujiahao
14   */
15
16  /**
17   * Kafka普通的异步发送
18   */
19
20  public class CustomProducer {
21        public static void main(String[] args) {
22
23              // 配置
24              Properties properties = new Properties();
25              // 连接集群
26              properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "master:9092");
27              // 指定对应的key和value的序列化类型
28              properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
    StringSerializer.class.getName());
29              properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
    StringSerializer.class.getName());
30
31              // 1.创建kafka生产者对象
32              // "" hello
33              KafkaProducer<String, String> kafkaProducer = new KafkaProducer<>(properties);
34
35              // 2.发送数据
36              for (int i = 0; i < 5; i++) {
37                    kafkaProducer.send(new ProducerRecord<>("first", "lwPigKing" + i));
38              }
39
40              // 3.关闭资源
41              kafkaProducer.close();
42        }
43  }
```

1. 启动Kakfa

Kafka 的运行依赖于 zookeeper，需要预先启动，可以启动 Kafka 内置的 zookeeper，也可以启动自己安装的：

```
1  # zookeeper启动命令
2  zkServer.sh start
```

启动单节点 kafka 用于测试：

```
1  /opt/module/kafka/bin/kafka-server-start.sh -daemon /opt/module/kafka/config/server.properties
```

2. 创建topic

```
# 创建用于测试主题
bin/kafka-topics.sh --create \
                            --bootstrap-server master:9092 \
                             --replication-factor 1 --partitions 1 \
                             --topic first

# 查看所有主题
 bin/kafka-topics.sh --list --bootstrap-server master:9092
```

3. 启动消费者

启动一个控制台消费者用于观察写入情况，启动命令如下：

```
bin/kafka-console-consumer.sh --bootstrap-server master:9092 --topic first --from-beginning
```

4. 运行项目

此时可以看到消费者控制台，输出如下，这里 `kafka-console-consumer` 只会打印出值信息，不会打印出键信息。

2.3 可能出现的问题

在这里可能出现的一个问题是：生产者程序在启动后，一直处于等待状态。这通常出现在你使用默认配置启动 Kafka 的情况下，此时需要对 `server.properties` 文件中的 `listeners` 配置进行更改：

```
# master 为我启动kafka服务的主机名，你可以换成自己的主机名或者ip地址
listeners=PLAINTEXT://master:9092
```

发送消息

上面的示例程序调用了 `send` 方法发送消息后没有做任何操作，在这种情况下，我们没有办法知道消息发送的结果。想要知道消息发送的结果，可以使用同步发送或者异步发送来实现。

2.1 同步发送

在调用 `send` 方法后可以接着调用 `get()` 方法，`send` 方法的返回值是一个 Future<RecordMetadata>对象，RecordMetadata 里面包含了发送消息的主题、分区、偏移量等信息。改写后的代码如下：

```
package producer;

import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.common.serialization.StringSerializer;

import java.util.Properties;
import java.util.concurrent.ExecutionException;

/**
 * Project:  BigDataCode
 * Create date:  2023/5/17
 * Created by fujiahao
 */

```

```java
/**
 * Kafka同步发送
 */

public class CustomProducerSync {
      public static void main(String[] args) throws ExecutionException, InterruptedException {

            // 配置
            Properties properties = new Properties();
            // 连接集群
            properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "master:9092");
            // 指定对应的key和value的序列化类型
            properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
            properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());

            // 1.创建kafka生产者对象
            // "" hello
            KafkaProducer<String, String> kafkaProducer = new KafkaProducer<>(properties);

            // 2.发送数据
            for (int i = 0; i < 5; i++) {
                  // 在异步发送后加上get()方法就变成了同步发送
                  kafkaProducer.send(new ProducerRecord<>("first", "lwPigKing" + i)).get();
            }

            // 3.关闭资源
            kafkaProducer.close();
      }
}

```

此时得到的输出如下：偏移量和调用次数有关，所有记录都分配到了 0 分区，这是因为在创建 `Hello-Kafka` 主题时候，使用 `--partitions` 指定其分区数为 1，即只有一个分区。

```
topic=Hello-Kafka, partition=0, offset=40
topic=Hello-Kafka, partition=0, offset=41
topic=Hello-Kafka, partition=0, offset=42
topic=Hello-Kafka, partition=0, offset=43
topic=Hello-Kafka, partition=0, offset=44
topic=Hello-Kafka, partition=0, offset=45
topic=Hello-Kafka, partition=0, offset=46
topic=Hello-Kafka, partition=0, offset=47
topic=Hello-Kafka, partition=0, offset=48
topic=Hello-Kafka, partition=0, offset=49
```

2.2 异步发送

通常我们并不关心发送成功的情况，更多关注的是失败的情况，因此 Kafka 提供了异步发送和回调函数。 代码如下：

```java
package producer;

import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.common.serialization.StringSerializer;
```

```
7
8   import java.util.Properties;
9
10  /**
11   * Project:  BigDataCode
12   * Create date:  2023/5/17
13   * Created by fujiahao
14   */
15
16  /**
17   * Kafka普通的异步发送
18   */
19
20  public class CustomProducer {
21        public static void main(String[] args) {
22
23              // 配置
24              Properties properties = new Properties();
25              // 连接集群
26              properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "master:9092");
27              // 指定对应的key和value的序列化类型
28              properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
    StringSerializer.class.getName());
29              properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
    StringSerializer.class.getName());
30
31              // 1.创建kafka生产者对象
32              // "" hello
33              KafkaProducer<String, String> kafkaProducer = new KafkaProducer<>(properties);
34
35              // 2.发送数据
36              for (int i = 0; i < 5; i++) {
37                    kafkaProducer.send(new ProducerRecord<>("first", "lwPigKing" + i));
38              }
39
40              // 3.关闭资源
41              kafkaProducer.close();
42        }
43  }
```

自定义分区器

Kafka 有着 默认的分区机制 ：

- 如果键值为 null， 则使用轮询 (Round Robin) 算法将消息均衡地分布到各个分区上；
- 如果键值不为 null，那么 Kafka 会使用内置的散列算法对键进行散列，然后分布到各个分区上。

某些情况下，你可能有着自己的分区需求，这时候可以采用自定义分区器实现。这里给出一个自定义分区器的示例：

```
1   package producer;
2
3   import org.apache.kafka.clients.producer.Partitioner;
4   import org.apache.kafka.common.Cluster;
5
6   import java.util.Map;
```

```
7
8   /**
9    * Project:  BigDataCode
10   * Create date:  2023/5/17
11   * Created by fujiahao
12   */
13  public class MyPartitioner implements Partitioner {
14        @Override
15        // s为topic主题、o为key、bytes为序列化key、byte1为序列化value、o1为value数据
16        public int partition(String s, Object o, byte[] bytes, Object o1, byte[] bytes1, Cluster
    cluster) {
17
18              // 获取数据
19              String msgValues = o1.toString();
20
21              int partition;
22              if (msgValues.contains("lwPigKing")) {
23                    partition = 0;
24              } else {
25                    partition = 1;
26              }
27
28              return partition;
29        }
30
31        @Override
32        public void close() {
33
34        }
35
36        @Override
37        public void configure(Map<String, ?> map) {
38
39        }
40  }
```

需要在创建生产者时指定分区器，和分区器所需要的配置参数：

```
1   package producer;
2
3   import org.apache.kafka.clients.producer.*;
4   import org.apache.kafka.common.serialization.StringSerializer;
5
6   import java.util.Properties;
7
8   /**
9    * Project:  BigDataCode
10   * Create date:  2023/5/17
11   * Created by fujiahao
12   */
13
14  /**
15   * Kafka指定分区
16   */
17
18  public class CustomProducerCallBackPartitions {
19        public static void main(String[] args) {
20
```

```
21              // 配置
22              Properties properties = new Properties();
23              // 连接集群
24              properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "master:9092");
25              // 指定对应的key和value的序列化类型
26              properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
    StringSerializer.class.getName());
27              properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
    StringSerializer.class.getName());
28
29              // 关联自定义分区器
30              properties.put(ProducerConfig.PARTITIONER_CLASS_CONFIG, "producer.MyPartitioner");
31
32              // 1.创建kafka生产者对象
33              // "" hello
34              KafkaProducer<String, String> kafkaProducer = new KafkaProducer<>(properties);
35
36              // 2.发送数据
37              for (int i = 0; i < 5; i++) {
38                    // new Callback()回调
39                    kafkaProducer.send(new ProducerRecord<>("first", 1,"", "lwPigKing" + i), new
    Callback() {
40                          @Override
41                          public void onCompletion(RecordMetadata recordMetadata, Exception e) {
42                                if (e == null) {
43                                      System.out.println("主题: " + recordMetadata.topic() + "分
    区: " + recordMetadata.partition());
44                                }
45                          }
46                    });
47              }
48
49              // 3.关闭资源
50              kafkaProducer.close();
51        }
52  }
53
```

生产者其他属性

上面生产者的创建都仅指定了服务地址，键序列化器、值序列化器，实际上 Kafka 的生产者还有很多可配置属性，如下：

1. acks

`acks 参数` 指定了必须要有多少个分区副本收到消息，生产者才会认为消息写入是成功的：

- **acks=0** ： 消息发送出去就认为已经成功了，不会等待任何来自服务器的响应；
- **acks=1** ： 只要集群的首领节点收到消息，生产者就会收到一个来自服务器成功响应；
- **acks=all** ： 只有当所有参与复制的节点全部收到消息时，生产者才会收到一个来自服务器的成功响应。

2. buffer.memory

设置生产者内存`缓冲区的大小`。

3. compression.type

默认情况下，发送的消息不会被`压缩`。如果想要进行压缩，可以配置此参数，可选值有 snappy，gzip，lz4。

4. retries

发生错误后，`消息重发的次数`。如果达到设定值，生产者就会放弃重试并返回错误。

5. batch.size

当有多个消息需要被发送到同一个分区时，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`。

6. linger.ms

该参数制定了生产者在发送批次之前`等待更多消息加入批次的时间`。

7. clent.id

客户端 id,服务器用来识别消息的来源。

8. max.in.flight.requests.per.connection

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

9. timeout.ms, request.timeout.ms & metadata.fetch.timeout.ms

- timeout.ms 指定了 borker 等待同步副本返回消息的确认时间；
- request.timeout.ms 指定了生产者在发送数据时等待服务器返回响应的时间；
- metadata.fetch.timeout.ms 指定了生产者在获取元数据（比如分区首领是谁）时等待服务器返回响应的时间。

10. max.block.ms

指定了在调用`send()`方法或使用`partitionsFor()`方法获取元数据时生产者的阻塞时间。当生产者的发送缓冲区已满，或者没有可用的元数据时，这些方法会阻塞。在阻塞时间达到 max.block.ms 时，生产者会抛出超时异常。

11. max.request.size

该参数用于控制生产者发送的请求大小。它可以指发送的单个消息的最大值，也可以指单个请求里所有消息总的大小。例如，假设这个值为 1000K，那么可以发送的单个最大消息为 1000K，或者生产者可以在单个请求里发送一个批次，该批次包含了 1000 个消息，每个消息大小为 1K。

12. receive.buffer.bytes & send.buffer.byte

这两个参数分别指定 TCP socket 接收和发送数据包缓冲区的大小，-1 代表使用操作系统的默认值。

第五章 消费者

消费者和消费者群组

在 Kafka 中，`消费者通常是消费者群组的一部分`，`多个消费者群组共同读取同一个主题时，彼此之间互不影响`。Kafka 之所以要引入消费者群组这个概念是因为 Kafka 消费者经常会做一些高延迟的操作，比如把数据写到数据库或 HDFS，或者进行耗时的计算，在这些情况下，单个消费者无法跟上数据生成的速度。此时可以增加更多的消费者，让它们分担负载，分别处理部分分区的消息，这就是 Kafka 实现横向伸缩的主要手段。

需要注意的是：同一个分区只能被同一个消费者群组里面的一个消费者读取，不可能存在同一个分区被同一个消费者群里多个消费者共同读取的情况，如图：

可以看到即便消费者 Consumer5 空闲了，但是也不会去读取任何一个分区的数据，这同时也提醒我们在使用时应该合理设置消费者的数量，以免造成闲置和额外开销。

分区再均衡

因为群组里的消费者共同读取主题的分区，所以当一个消费者被关闭或发生崩溃时，它就离开了群组，原本由它读取的分区将由群组里的其他消费者来读取。同时在主题发生变化时，比如添加了新的分区，也会发生分区与消费者的重新分配，分区的所有权从一个消费者转移到另一个消费者，这样的行为被称为`再均衡`。正是因为再均衡，所以`消费费者群组才能保证高可用性和伸缩性`。

消费者通过向`群组协调器`所在的 broker `发送心跳来维持它们和群组的从属关系以及它们对分区的所有权`。只要消费者以正常的时间间隔发送心跳，就被认为是活跃的，说明它还在读取分区里的消息。消费者会在轮询消息或提交偏移量时发送心跳。如果消费者停止发送心跳的时间足够长，会话就会过期，群组协调器认为它已经死亡，就会触发再均衡。

创建Kafka消费者

在创建消费者的时候以下以下三个选项是必选的：

- **bootstrap.servers**： 指定 broker 的地址清单，清单里不需要包含所有的 broker 地址，生产者会从给定的 broker 里查找 broker 的信息。不过建议至少要提供两个 broker 的信息作为容错；
- **key.deserializer**： 指定键的反序列化器；
- **value.deserializer**： 指定值的反序列化器。

除此之外你还需要指明你需要想订阅的主题，可以使用如下两个 API：

- **consumer.subscribe(Collection<String> topics)** ： 指明需要订阅的主题的集合；
- **consumer.subscribe(Pattern pattern)** ： 使用正则来匹配需要订阅的集合。

最后只需要通过轮询 API(`poll`) 向服务器定时请求数据。一旦消费者订阅了主题，轮询就会处理所有的细节，包括群组协调、分区再均衡、发送心跳和获取数据，这使得开发者只需要关注从分区返回的数据，然后进行业务处理。

自动提交偏移量

3.1 偏移量的重要性

Kafka 的每一条消息都有一个偏移量属性，记录了其在分区中的位置，偏移量是一个单调递增的整数。消费者通过往一个叫作`__consumer_offset`的特殊主题发送消息，消息里包含每个分区的偏移量。 如果消费者一直处于运行状态，那么偏移量就没有什么用处。不过，如果有消费者退出或者新分区加入，此时就会触发再均衡。完成再均衡之后，每个消费者可能分配到新的分区，而不是之前处理的那个。为了能够继续之前的工作，消费者需要读取每个分区最后一次提交的偏移量，然后从偏移量指定的地方继续处理。 因为这个原因，所以如果不能正确提交偏移量，就可能会导致数据丢失或者重复出现消费，比如下面情况：

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

3.2 自动提交偏移量

Kafka `支持自动提交和手动提交偏移量` 两种方式。这里先介绍比较简单的自动提交：

只需要将消费者的 `enable.auto.commit` 属性配置为 `true` 即可完成自动提交的配置。 此时每隔固定的时间，消费者就会把 `poll()` 方法接收到的最大偏移量进行提交，提交间隔由 `auto.commit.interval.ms` 属性进行配置， `默认值是 5s` 。

使用自动提交是存在隐患的，假设我们使用默认的 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。基于这个原因，Kafka 也提供了手动提交偏移量的 API，使得用户可以更为灵活的提交偏移量。

```
package consumer;

/**
 * Project:  BigDataCode
 * Create date:  2023/5/23
 * Created by fujiahao
 */

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.serialization.StringDeserializer;

import java.time.Duration;
import java.util.ArrayList;
import java.util.Properties;

/**
 * 自动提交offset
 */

public class CustomConsumerAutoOffset {
      public static void main(String[] args) {

            // 0.配置
            Properties properties = new Properties();
            properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,"master:9092");
            properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
            properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
            properties.put(ConsumerConfig.GROUP_ID_CONFIG, "test");

            // 自动提交
            properties.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, true);
            // 提交时间间隔
            properties.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG, 1000);

```

```
            // 1.创建一个消费者 "","hello"
            KafkaConsumer<String, String> kafkaConsumer = new KafkaConsumer<>(properties);

            // 2.订阅主题:lwPigKing
            ArrayList<String> topics = new ArrayList<>();
            topics.add("lwPigKing");
            kafkaConsumer.subscribe(topics);

            // 3.消费数据
            while (true) {

                  ConsumerRecords<String, String> consumerRecords =
kafkaConsumer.poll(Duration.ofSeconds(1));
                  for (ConsumerRecord<String, String> consumerRecord : consumerRecords) {
                        System.out.println(consumerRecord);
                  }

            }

      }
}
```

手动提交偏移量

用户可以通过将 `enable.auto.commit` 设为 `false`，然后手动提交偏移量。基于用户需求手动提交偏移量可以分为两大类：

- 手动提交当前偏移量：即手动提交当前轮询的最大偏移量；
- 手动提交固定偏移量：即按照业务需求，提交某一个固定的偏移量。

而按照 Kafka API，手动提交偏移量又可以分为同步提交和异步提交。

4.1 同步提交

通过调用 `consumer.commitSync()` 来进行同步提交，不传递任何参数时提交的是当前轮询的最大偏移量。

```
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.serialization.StringDeserializer;

import java.time.Duration;
import java.util.ArrayList;
import java.util.Properties;

/**
 * 同步和移动提交、手动设置offset
 */

public class CustomConsumerByHandSync {
      public static void main(String[] args) {

            // 0.配置
            Properties properties = new Properties();
            properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,"master:9092");
```

```
            properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
            properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
            properties.put(ConsumerConfig.GROUP_ID_CONFIG, "test");

            // 手动提交
            properties.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);

            // 1.创建一个消费者 "","hello"
            KafkaConsumer<String, String> kafkaConsumer = new KafkaConsumer<>(properties);

            // 2.订阅主题:lwPigKing
            ArrayList<String> topics = new ArrayList<>();
            topics.add("lwPigKing");
            kafkaConsumer.subscribe(topics);

            // 3.消费数据
            while (true) {

                  ConsumerRecords<String, String> consumerRecords =
kafkaConsumer.poll(Duration.ofSeconds(1));
                  for (ConsumerRecord<String, String> consumerRecord : consumerRecords) {
                        System.out.println(consumerRecord);
                  }

                  // 手动同步提交offset
                  // kafkaConsumer.commitSync();

                  // 手动异步提交offset
                  kafkaConsumer.commitAsync();
            }

      }
}
```

如果某个提交失败，同步提交还会进行重试，这可以保证数据能够最大限度提交成功，但是同时也会降低程序的吞吐量。基于这个原因，Kafka 还提供了异步提交的 API。

4.2 异步提交

异步提交可以提高程序的吞吐量，因为此时你可以尽管请求数据，而不用等待 Broker 的响应。代码如下：

```
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.serialization.StringDeserializer;

import java.time.Duration;
import java.util.ArrayList;
import java.util.Properties;

/**
 * 同步和移动提交、手动设置offset
 */

public class CustomConsumerByHandSync {
```

```
15          public static void main(String[] args) {
16
17                  // 0.配置
18                  Properties properties = new Properties();
19                  properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,"master:9092");
20                  properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
    StringDeserializer.class.getName());
21                  properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
    StringDeserializer.class.getName());
22                  properties.put(ConsumerConfig.GROUP_ID_CONFIG, "test");
23
24                  // 手动提交
25                  properties.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);
26
27                  // 1.创建一个消费者 "","hello"
28                  KafkaConsumer<String, String> kafkaConsumer = new KafkaConsumer<>(properties);
29
30                  // 2.订阅主题:lwPigKing
31                  ArrayList<String> topics = new ArrayList<>();
32                  topics.add("lwPigKing");
33                  kafkaConsumer.subscribe(topics);
34
35                  // 3.消费数据
36                  while (true) {
37
38                          ConsumerRecords<String, String> consumerRecords =
    kafkaConsumer.poll(Duration.ofSeconds(1));
39                          for (ConsumerRecord<String, String> consumerRecord : consumerRecords) {
40                                  System.out.println(consumerRecord);
41                          }
42
43                          // 手动同步提交offset
44                          // kafkaConsumer.commitSync();
45
46                          // 手动异步提交offset
47                          kafkaConsumer.commitAsync();
48                  }
49
50          }
51  }
```

异步提交存在的问题是，在提交失败的时候不会进行自动重试，实际上也不能进行自动重试。假设程序同时提交了 200 和 300 的偏移量，此时 200 的偏移量失败的，但是紧随其后的 300 的偏移量成功了，此时如果重试就会存在 200 覆盖 300 偏移量的可能。同步提交就不存在这个问题，因为在同步提交的情况下，300 的提交请求必须等待服务器返回 200 提交请求的成功反馈后才会发出。基于这个原因，某些情况下，需要同时组合同步和异步两种提交方式。

> 注：虽然程序不能在失败时候进行自动重试，但是我们是可以手动进行重试的，你可以通过一个 Map<TopicPartition, Integer> offsets 来维护你提交的每个分区的偏移量，然后当失败时候，你可以判断失败的偏移量是否小于你维护的同主题同分区的最后提交的偏移量，如果小于则代表你已经提交了更大的偏移量请求，此时不需要重试，否则就可以进行手动重试。

4.3 同步加异步提交

下面这种情况，在正常的轮询中使用异步提交来保证吞吐量，但是因为在最后即将要关闭消费者了，所以此时需要用同步提交来保证最大限度的提交成功。

```
try {
      while (true) {
            ConsumerRecords<String, String> records = consumer.poll(Duration.of(100,
ChronoUnit.MILLIS));
            for (ConsumerRecord<String, String> record : records) {
                  System.out.println(record);
            }
            // 异步提交
            consumer.commitAsync();
      }
} catch (Exception e) {
      e.printStackTrace();
} finally {
      try {
            // 因为即将要关闭消费者，所以要用同步提交保证提交成功
            consumer.commitSync();
      } finally {
            consumer.close();
      }
}
```

4.4 提交特定偏移量

在上面同步和异步提交的 API 中，实际上我们都没有对 commit 方法传递参数，此时默认提交的是当前轮询的最大偏移量，如果你需要提交特定的偏移量，可以调用它们的重载方法。

```
/*同步提交特定偏移量*/
commitSync(Map<TopicPartition, OffsetAndMetadata> offsets)
/*异步提交特定偏移量*/
commitAsync(Map<TopicPartition, OffsetAndMetadata> offsets, OffsetCommitCallback callback)
```

需要注意的是，因为你可以订阅多个主题，所以 `offsets` 中必须要包含所有主题的每个分区的偏移量，示例代码如下：

```
try {
      while (true) {
            ConsumerRecords<String, String> records = consumer.poll(Duration.of(100,
ChronoUnit.MILLIS));
            for (ConsumerRecord<String, String> record : records) {
                  System.out.println(record);
                  /*记录每个主题的每个分区的偏移量*/
                  TopicPartition topicPartition = new TopicPartition(record.topic(),
record.partition());
                  OffsetAndMetadata offsetAndMetadata = new OffsetAndMetadata(record.offset()+1,
"no metaData");
                  /*TopicPartition 重写过 hashCode 和 equals 方法，所以能够保证同一主题和分区的实
例不会被重复添加*/
                  offsets.put(topicPartition, offsetAndMetadata);
            }
            /*提交特定偏移量*/
            consumer.commitAsync(offsets, null);
      }
} finally {
```

```
        consumer.close();
    }
```

监听分区再均衡

因为分区再均衡会导致分区与消费者的重新划分，有时候你可能希望在再均衡前执行一些操作：比如提交已经处理但是尚未提交的偏移量，关闭数据库连接等。此时可以在订阅主题时候，调用 `subscribe` 的重载方法传入自定义的分区再均衡监听器。

```
 /*订阅指定集合内的所有主题*/
subscribe(Collection<String> topics, ConsumerRebalanceListener listener)
 /*使用正则匹配需要订阅的主题*/
subscribe(Pattern pattern, ConsumerRebalanceListener listener)
```

代码示例如下：

```
Map<TopicPartition, OffsetAndMetadata> offsets = new HashMap<>();

consumer.subscribe(Collections.singletonList(topic), new ConsumerRebalanceListener() {
      /*该方法会在消费者停止读取消息之后，再均衡开始之前就调用*/
      @Override
      public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
            System.out.println("再均衡即将触发");
            // 提交已经处理的偏移量
            consumer.commitSync(offsets);
      }

      /*该方法会在重新分配分区之后，消费者开始读取消息之前被调用*/
      @Override
      public void onPartitionsAssigned(Collection<TopicPartition> partitions) {

      }
});

try {
      while (true) {
            ConsumerRecords<String, String> records = consumer.poll(Duration.of(100,
ChronoUnit.MILLIS));
            for (ConsumerRecord<String, String> record : records) {
                  System.out.println(record);
                  TopicPartition topicPartition = new TopicPartition(record.topic(),
record.partition());
                  OffsetAndMetadata offsetAndMetadata = new OffsetAndMetadata(record.offset() + 1,
"no metaData");
                  /*TopicPartition 重写过 hashCode 和 equals 方法，所以能够保证同一主题和分区的实
例不会被重复添加*/
                  offsets.put(topicPartition, offsetAndMetadata);
            }
            consumer.commitAsync(offsets, null);
      }
} finally {
      consumer.close();
}
```

退出轮询

Kafka 提供了 `consumer.wakeup()` 方法用于退出轮询，它通过抛出 `WakeupException` 异常来跳出循环。需要注意的是，在退出线程时最好显示的调用 `consumer.close()` ,此时消费者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。

下面的示例代码为监听控制台输出，当输入 `exit` 时结束轮询，关闭消费者并退出程序：

```
/*调用 wakeup 优雅的退出*/
final Thread mainThread = Thread.currentThread();
new Thread(() -> {
      Scanner sc = new Scanner(System.in);
      while (sc.hasNext()) {
            if ("exit".equals(sc.next())) {
                  consumer.wakeup();
                  try {
                        /*等待主线程完成提交偏移量、关闭消费者等操作*/
                        mainThread.join();
                        break;
                  } catch (InterruptedException e) {
                        e.printStackTrace();
                  }
            }
      }
}).start();

try {
      while (true) {
            ConsumerRecords<String, String> records = consumer.poll(Duration.of(100,
ChronoUnit.MILLIS));
            for (ConsumerRecord<String, String> rd : records) {
                  System.out.printf("topic = %s,partition = %d, key = %s, value = %s, offset =
%d,\n",
                                    rd.topic(), rd.partition(), rd.key(), rd.value(),
rd.offset());
            }
      }
} catch (WakeupException e) {
      //对于 wakeup() 调用引起的 WakeupException 异常可以不必处理
} finally {
      consumer.close();
      System.out.println("consumer 关闭");
}
```

独立的消费者

因为 Kafka 的设计目标是高吞吐和低延迟，所以在 Kafka 中，消费者通常都是从属于某个群组的，这是因为单个消费者的处理能力是有限的。但是某些时候你的需求可能很简单，比如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即可。

在这种情况下，就不需要订阅主题， 取而代之的是消费者为自己分配分区。 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。 分配分区的示例代码如下：

```
List<TopicPartition> partitions = new ArrayList<>();
List<PartitionInfo> partitionInfos = consumer.partitionsFor(topic);

/*可以指定读取哪些分区 如这里假设只读取主题的 0 分区*/
for (PartitionInfo partition : partitionInfos) {
      if (partition.partition()==0){
            partitions.add(new TopicPartition(partition.topic(), partition.partition()));
      }
}

// 为消费者指定分区
consumer.assign(partitions);


while (true) {
      ConsumerRecords<Integer, String> records = consumer.poll(Duration.of(100,
ChronoUnit.MILLIS));
      for (ConsumerRecord<Integer, String> record : records) {
            System.out.printf("partition = %s, key = %d, value = %s\n",
                                    record.partition(), record.key(), record.value());
      }
      consumer.commitSync();
}
```

消费者可选属性

1. fetch.min.byte

消费者从服务器获取记录的最小字节数。如果可用的数据量小于设置值，broker 会等待有足够的可用数据时才会把它返回给消费者。

2. fetch.max.wait.ms

broker 返回给消费者数据的等待时间，默认是 500ms。

3. max.partition.fetch.bytes

该属性指定了服务器从每个分区返回给消费者的最大字节数，默认为 1MB。

4. session.timeout.ms

消费者在被认为死亡之前可以与服务器断开连接的时间，默认是 3s。

5. auto.offset.reset

该属性指定了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该作何处理：

- latest (默认值)： 在偏移量无效的情况下，消费者将从最新的记录开始读取数据（在消费者启动之后生成的最新记录）；
- earliest： 在偏移量无效的情况下，消费者将从起始位置读取分区的记录。

6. enable.auto.commit

是否自动提交偏移量，默认值是 true。为了避免出现重复消费和数据丢失，可以把它设置为 false。

7. client.id

客户端 id，服务器用来识别消息的来源。

8. max.poll.records

单次调用 `poll()` 方法能够返回的记录数量。

9. receive.buffer.bytes & send.buffer.byte

这两个参数分别指定 TCP socket 接收和发送数据包缓冲区的大小，-1 代表使用操作系统的默认值。

第六章 其他操作

1 增加副本因子

```
增加副本因子
vim increase-replication-factor.json

{
     "version":1,
     "partitions":[
          {"topic":"four",
           "partition":0,
           "replicas":[0,1,2]},
           {"topic":four,
             "partitions":1,
             "replicas":[0,1,2]},
             {"topic":four,
              "partition":2,
              "replicas":[0,1,2]}
              ]
}

kafka-reassign-partitions.sh --bootstrap-server
master:9092 --reassignment-json-file increase-replication-factor.json
--execute
```

2 手动调整分区

```
手动调整分区

创建副本存储计划
vim increase-replication-factor.json
{
     "version":1,
     "partitions":[{"topic":"three","partition":0,"replicas":[0,1]},
                          {"topic":"three","partition":1,"replicas":[0,1]},
                          {"topic":"three","partition":2,"replicas":[1,0]},
                          {"topic":"three","partition":3,"replicas":[1,0]}]
}
```

```
执行副本存储计划
kafka-reassign-partitions.sh --bootstrap-server master:9092
--reassignment-json-file increase-replication-factor.json --execute

验证副本存储计划
kafka-reassign-partitions.sh --bootstrap-server master:9092
--reassignment-json-file increase-replication-factor.json --verify
```

3 服役新节点

```
服役新节点

1.创建一个负载均衡的主题
vim topic-to-move.json
{
     "topics": [
          {"topic":"first"}
     ],

     "version":1
}

2.生成一个负载均衡的计划
kafka-reassign-partitions.sh --bootstrap-server master:9092
--topic-to-move-json-file topics-to-move.json
--broker-list "0, 1, 2, 3" --generate

3.创建副本存储计划
vim increase-replication-factor.json
在该json文件里写入Proposed partition reassignment configuration所生成的内容

4.执行副本存储计划
kafka-reassign-partitions.sh --bootstrap-server master:9092
--reassignment-json-file increase-replication-factor.json --execute

5.验证副本存储计划
kafka-reassign-partitions.sh --bootstrap-server master:9092
--reassignment-json-file increase-replication-factor.json --verify
```

4 退役旧节点

```
退役旧节点

1.创建一个负载均衡的主题
vim topic-to-move.json
{
     "topics": [
          {"topic":"first"}
     ],

     "version":1
}

2.生成一个负载均衡的计划
kafka-reassign-partitions.sh --bootstrap-server master:9092
```

```
--topic-to-move-json-file topics-to-move.json
--broker-list "0, 1, 2" --generate

3.创建副本存储计划
vim increase-replication-factor.json
在该json文件里写入Proposed partition reassignment configuration所生成的内容

4.执行副本存储计划
kafka-reassign-partitions.sh --bootstrap-server master:9092
--reassignment-json-file increase-replication-factor.json --execute

5.验证副本存储计划
kafka-reassign-partitions.sh --bootstrap-server master:9092
--reassignment-json-file increase-replication-factor.json --verify

6.停止kafka
kafka-server-stop.sh
```

5 查看文件存储index和log信息

```
kafka-run-class.sh kafka.tools.DumpLogSegments -files ./0000000.index
```