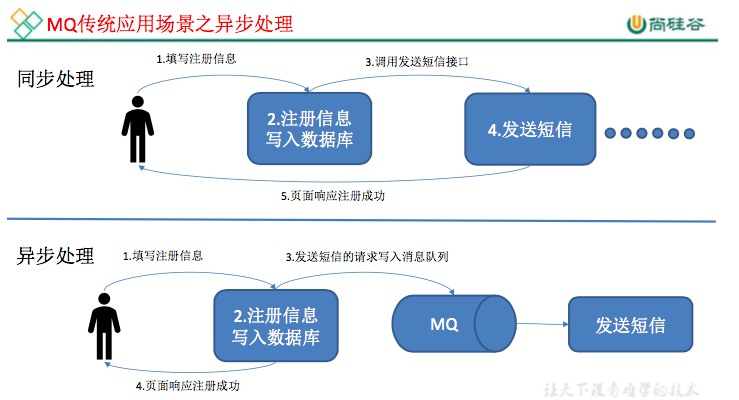
Kafka教程

1. 概述
2. **定义**

Kafka 是一个分布式的基于发布/订阅模式的消息队列(Message Queue)，主要应用于大数据实时处理领域。

1. **消息队列**

传统消息队列的应用场景



使用消息队列的好处

1)解耦 允许你独立的扩展或修改两边的处理过程，只要确保它们遵守同样的接口约束。

2)可恢复性

系统的一部分组件失效时，不会影响到整个系统。消息队列降低了进程间的耦合度，所以即使一个处理消息的进程挂掉，加入队列中的消息仍然可以在系统恢复后被处理。

3)缓冲

有助于控制和优化数据流经过系统的速度，解决生产消息和消费消息的处理速度不一致的情况。

4)灵活性 & 峰值处理能力（削峰）

在访问量剧增的情况下，应用仍然需要继续发挥作用，但是这样的突发流量并不常见。

如果为以能处理这类峰值访问为标准来投入资源随时待命无疑是巨大的浪费。使用消息队列能够使关键组件顶住突发的访问压力，而不会因为突发的超负荷的请求而完全崩溃。

5)异步通信

很多时候，用户不想也不需要立即处理消息。消息队列提供了异步处理机制，允许用户把一个消息放入队列，但并不立即处理它。想向队列中放入多少消息就放多少，然后在需要的时候再去处理它们。

注：主要作用是解耦和削峰

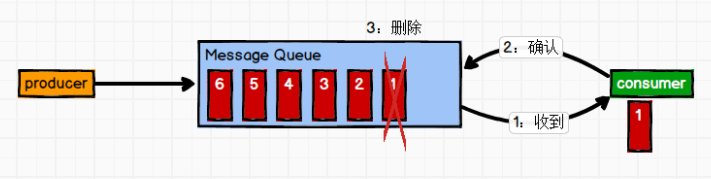
1. **消息队列两种模式**

1）、点对点模式

一对一，消费者主动拉取数据，消息收到后消息清除

消息生产者生产消息发送到Queue中，然后消息消费者从 Queue 中取出并且消费消息。

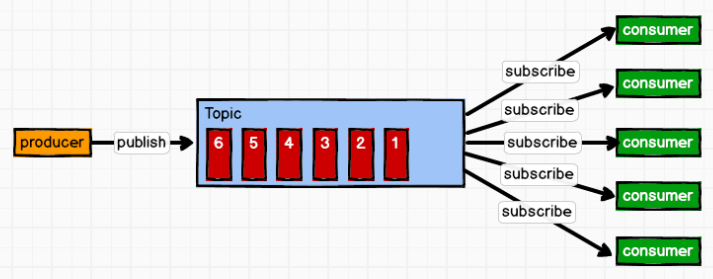
消息被消费以后，queue 中不再有存储，所以消息消费者不可能消费到已经被消费的消息。Queue支持存在多个消费者，但是对一个消息而言，只会有一个消费者可以消费。



2）、发布/订阅模式

一对多，消费者消费数据之后不会清除消息

消息生产者(发布)将消息发布到 topic 中，同时有多个消息消费者(订阅)消费该消息。和点对点方式不同，发布到 topic 的消息会被所有订阅者消费。

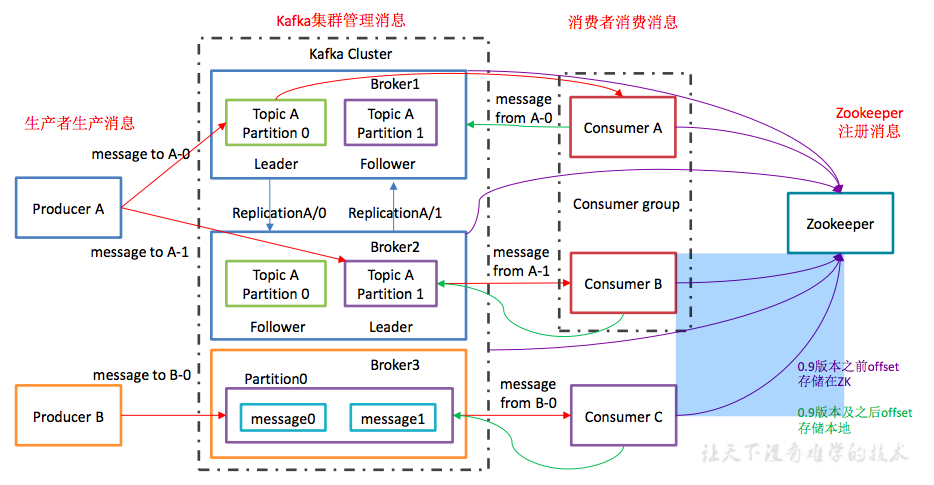


发布/订阅模式又有两种，一种是消费者主动拉取消息，另外一种是队列主动推消息给消费者。

队列主动推消息这种情况，队列不区别消费者消费能力的推送消息，可能会存在其中某个消费者消费信息太慢，导致某个消费者过载的问题，而有的消费者可能存在处理消息过快，长时间空闲，导致资源浪费。

消费者主动拉取消息，消费者就会维护一个长轮询，定时去询问队列是否有消息可以消费，这种情况会消耗浪费消费者部分资源。Kafka就采用的是这种方式。

1. Kafka架构
2. **基础架构**



1)Producer :消息生产者，就是向 kafka broker 发消息的客户端;

2)Consumer :消息消费者，向 kafka broker 取消息的客户端;

3)Consumer Group (CG):消费者组，由多个 consumer 组成。消费者组内每个消费者负责消费不同分区的数据，一个分区只能由一个组内消费者消费;消费者组之间互不影响。所有的消费者都属于某个消费者组，即消费者组是逻辑上的一个订阅者。

4)Broker :一台 kafka 服务器就是一个 broker。一个集群由多个 broker 组成。一个 broker 可以容纳多个 topic。

5)Topic :可以理解为一个队列，生产者和消费者面向的都是一个topic;

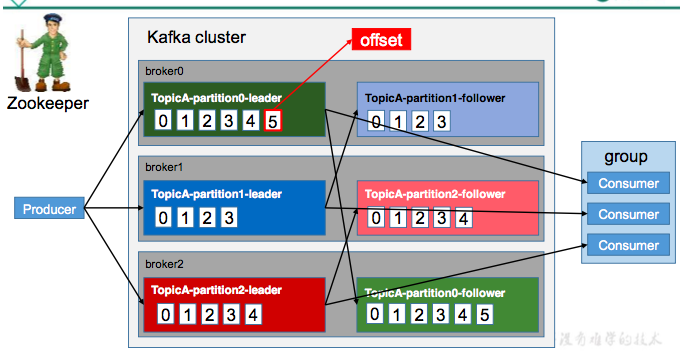
6)Partition:为了实现扩展性，一个非常大的 topic 可以分布到多个 broker(即服务器)上， 一个 topic 可以分为多个 partition，每个 partition 是一个有序的队列;

7)Replica:副本，为保证集群中的某个节点发生故障时，该节点上的 partition 数据不丢失，且 kafka 仍然能够继续工作，kafka 提供了副本机制，一个 topic 的每个分区都有若干个副本， 一个 leader 和若干个 follower

8)leader:每个分区多个副本的“主”，生产者发送数据的对象，以及消费者消费数据的对象都是 leader。

9)follower:每个分区多个副本中的“从”，实时从 leader 中同步数据，保持和 leader 数据的同步。leader 发生故障时，某个 follower 会成为新的 leader。

1. **工作流程**

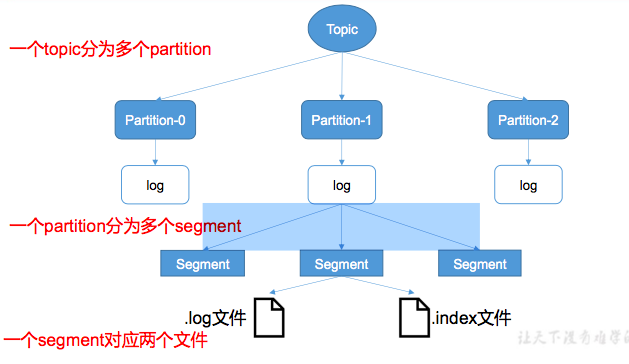


offset：偏移量，用于在消费者恢复服务的时候，记录消费者上次服务down掉时，消息消费到某处的一个值。0.9版本前该值是存于zookeeper，之后是存于kafka本地。

Kafka 中消息是以 topic 进行分类的，生产者生产消息，消费者消费消息，都是面向 topic 的。

topic 是逻辑上的概念，而 partition 是物理上的概念，每个 partition 对应于一个 log 文件，该 log 文件中存储的就是 producer 生产的数据。Producer 生产的数据会被不断追加到该 log 文件末端，且每条数据都有自己的 offset。消费者组中的每个消费者，都会实时记录自己消费到了哪个 offset，以便出错恢复时，从上次的位置继续消费。

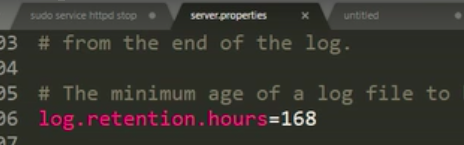
1. **文件存储**



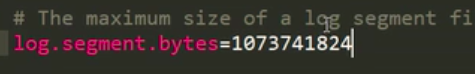
由于生产者生产的消息会不断追加到 log 文件末尾，为防止 log 文件过大导致数据定位效率低下，Kafka采取了分片和索引机制，将每个 partition 分为多个 segment。每个 segment 对应两个文件——“.index”文件和“.log”文件。这些文件位于一个文件夹下，该文件夹的命名 规则为:topic 名称+分区序号（也就是offset）。例如，first 这个 topic 有三个分区，则其对应的文件夹为 first- 0,first-1,first-2。



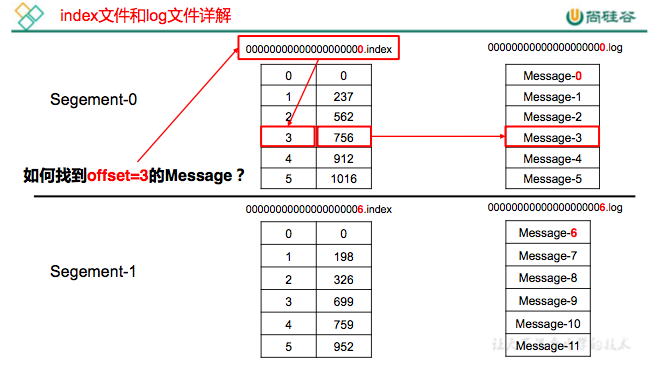
注：消息log最多只会保存7天，7天后会清除，该值可以在kafka的server.properties配置文件中进行修改。



log文件最大能到1G，配置在server.properties,一个segment对应一个log文件



index 和 log 文件以当前 segment 的第一条消息的 offset 命名。下图为 index 文件和 log 文件的结构示意图。



“.index”文件存储大量的索引信息，“.log”文件存储大量的数据，索引文件中的元数据指向对应数据文件中 message 的物理偏移地址。

生产消息，如何追加到文件？

一个消息过来，kafka的broker会将消息分配到对应的partition分区，然后再判断对应的segment片段log文件是否已满（默认1G），如果未满，则追加到log文件里面，如果已满，则创建新的segment片段log文件。

log文件名称生成规则？

比如有10个消息，一个segment存储5个消息，那个第一个segment名称则是00000000.log,第二个消息则从上个log的最大消息index+1,index默认是从0开始，上个segment结尾的index是4，则第二个segment的名称就是0000005.log

index索引算法？

内部是用的二分查找，快速定位到该数据在哪个segment的索引文件（有很多segment，每个segment对应一个index文件），比如上图，消息3会很快定位到索引文件0000000.index，消息3的偏移量offset也是3，所以快速找到index文件中3的位置，3位置存储的是消息在log文件存储的开始位置，以及消息的大小，所以得到这个两个值后，通过计算就能得到消息3在log文件中的起始位置和结束位置，这样就能取到整条消息。

1. **生产者**
2. 分区策略

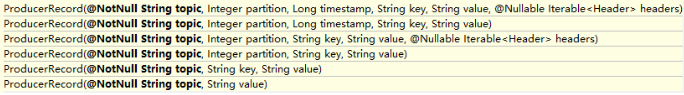
1)分区的原因

(1)方便在集群中扩展，每个 Partition 可以通过调整以适应它所在的机器，而一个 topic又可以有多个 Partition 组成，因此整个集群就可以适应任意大小的数据了;

(2)可以提高并发，因为可以以 Partition 为单位读写了。

2)分区的原则

我们需要将 producer 发送的数据封装成一个 ProducerRecord 对象。



如上ProducerRecord的构造函数包含3种情况

(1)指明 partition 的情况下，直接将指明的值直接作为 partiton 值;

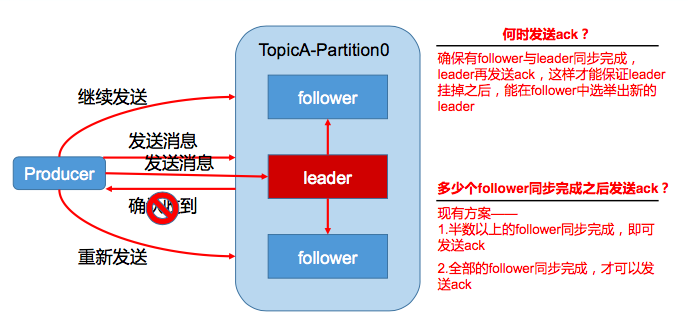
(2)没有指明 partition 值但有 key 的情况下，将 key 的 hash 值与 topic 的 partition 数进行取余得到 partition 值;

(3)既没有 partition 值又没有 key 值的情况下，第一次调用时随机生成一个整数(后面每次调用在这个整数上自增)，将这个值与 topic 可用的 partition 总数取余得到 partition 值，也就是常说的 round-robin 算法。

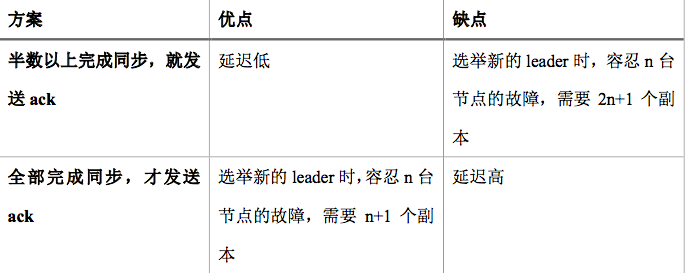
比如我有3个分区，编号是0、1、2，第一次通过随机计算是在1分区，那么第二次就是2partition，第三次就放0partition，第四次又重新循环放2partition。

1. 数据可靠性保证

为保证 producer 发送的数据，能可靠的发送到指定的 topic，topic 的每个 partition 收到 producer 发送的数据后，都需要向 producer 发送 ack(acknowledgement 确认收到)，如果 producer 收到 ack，就会进行下一轮的发送，否则重新发送数据。



1)副本数据同步策略



Kafka 选择了第二种方案，原因如下:

1.同样为了容忍 n 台节点的故障，第一种方案需要 2n+1 个副本，而第二种方案只需要 n+1 个副本，而 Kafka 的每个分区都有大量的数据，第一种方案会造成大量数据的冗余。

2.虽然第二种方案的网络延迟会比较高，但网络延迟对 Kafka 的影响较小。

2)ISR

采用第二种方案之后，设想以下情景:leader 收到数据，所有 follower 都开始同步数据， 但有一个 follower，因为某种故障，迟迟不能与 leader 进行同步，那 leader 就要一直等下去， 直到它完成同步，才能发送 ack。这个问题怎么解决呢?

Leader 维护了一个动态的 in-sync replica set (ISR)，意为和 leader 保持同步的 follower 集 合。当 ISR 中的 follower 完成数据的同步之后，leader 就会给 follower 发送 ack。如果 follower 长时间未向 leader 同步数据，则该 follower 将被踢出 ISR，该时间阈值由replica.lag.time.max.ms 参数设定（默认10秒）。Leader 发生故障之后，就会从 ISR 中选举新的 leader。

ISR其实是个同步队列

什么样的follower会加入到ISR？

1）、同步时间，拉取的速度，在某个时间内同步的，则放入ISR，超过该时间就剔除

2）、follower与leader之间数据数目的差值，一般follower都会比leader的数据少或者等于，只要它与leader之间的数据条目差值低于一个数字就会被加入ISR。比如差值是10，10条内就加入ISR，大于10条就剔除ISR。

注：在0.9版本之前以上两者都存在，0.9及以后，去掉了第二种。

第二种移除的原因：比如这个差值设置为10，消费生产者一般是batch提交消息，比如一次提交了12条，这时，leader和所有follower之间的差值都大于了10，此时，所有的follower都被移除ISR。此时follower继续同步数据，当与leader的差值又小于10时，则又将该follower加入到ISR。如果消费者此时频繁进行该操作，则会对ISR（ISR信息是从zookeeper获取，并加载到内存中）进行频繁的移除和加入操作，增加系统的负担。

ack 应答机制

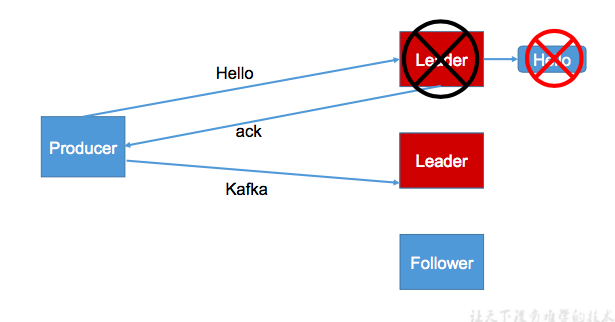
对于某些不太重要的数据，对数据的可靠性要求不是很高，能够容忍数据的少量丢失， 所以没必要等 ISR 中的 follower 全部接收成功。

所以 Kafka 为用户提供了三种可靠性级别，用户根据对可靠性和延迟的要求进行权衡，选择以下的配置。

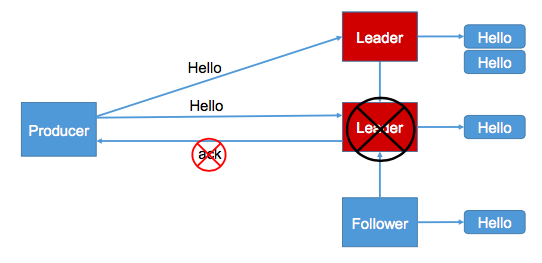
acks 参数配置:

0:producer不等待 broker 的 ack，这一操作提供了一个最低的延迟，broker 一接收到还没有写入磁盘就已经返回，当 broker 故障时有可能丢失数据;

1:producer 等待 broker 的 ack，partition 的 leader 落盘成功后返回 ack，如果在 follower 同步成功之前 leader 故障，那么将会丢失数据;



-1(all):producer 等待 broker 的 ack，partition 的 leader 和 follower 全部落盘成功后才 返回 ack。但是如果在 follower 同步完成后，broker 发送 ack 之前，leader 发生故障，那么会造成数据重复。

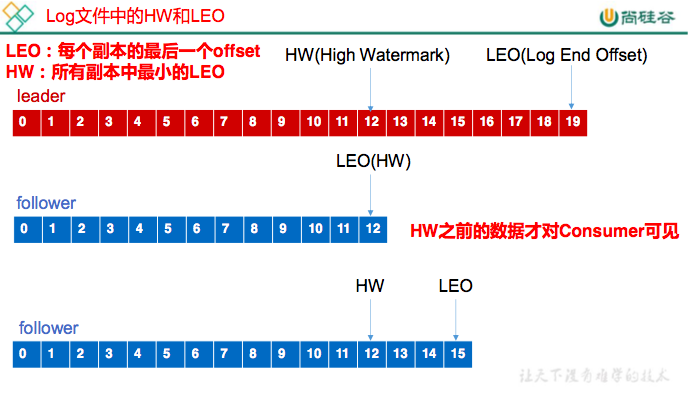


注：-1时其实也可能会存在数据丢失，比如leader落盘后，如果此时所有的follower同步比较慢，还未完全同步完数据，follower就不在ISR里，此时ISR只有leader，那么leader会马上返回ack，此时要是leader挂掉，数据就会丢失。

1. **数据存储一致性**

如果leader的数据是12条，follower1的数据是8条，follower2的数据是10条，此时如果leader挂了，选中follower2作为leader，follower1开始从新leader中同步数据，此时，旧leader突然复活了，旧leader也开始从新leader同步数据，但是其实此时新leader的数据反而比旧leader的数据少。造成存储数据不一致的问题。

Kafka的解决方案如下



**LEO**:指的是每个副本最大的 offset;

HW:指的是消费者能见到的最大的 offset，ISR 队列中最小的 LEO。

(1)follower 故障

follower 发生故障后会被临时踢出 ISR，待该 follower 恢复后，follower 会读取本地磁盘记录的上次的 HW，并将 log 文件高于 HW 的部分截取掉，从 HW 开始向 leader 进行同步。 等该 follower 的 LEO 大于等于该 Partition 的 HW，即 follower 追上 leader 之后，就可以重新加入 ISR 了。

(2)leader 故障

leader 发生故障之后，会从 ISR 中选出一个新的 leader，之后，为保证多个副本之间的数据一致性，其余的 follower 会先将各自的 log 文件高于 HW 的部分截掉，然后从新的 leader 同步数据。

注意:这只能保证副本之间的数据一致性，并不能保证数据不丢失或者不重复。

1. **Exactly Once精准一次**

将服务器的 ACK 级别设置为-1，可以保证 Producer 到 Server 之间不会丢失数据，即 At Least Once 语义。相对的，将服务器 ACK 级别设置为 0，可以保证生产者每条消息只会被发送一次，即 At Most Once 语义。

At Least Once 可以保证数据不丢失，但是不能保证数据不重复;相对的，At Most Once 可以保证数据不重复，但是不能保证数据不丢失。但是，对于一些非常重要的信息，比如说交易数据，下游数据消费者要求数据既不重复也不丢失，即 Exactly Once 语义。在 0.11 版 本以前的 Kafka，对此是无能为力的，只能保证数据不丢失，再在下游消费者对数据做全局去重。对于多个下游应用的情况，每个都需要单独做全局去重，这就对性能造成了很大影响。

0.11 版本的 Kafka，引入了一项重大特性:幂等性。所谓的幂等性就是指 Producer 不论 向 Server 发送多少次重复数据，Server 端都只会持久化一条。幂等性结合 At Least Once 语 义，就构成了 Kafka 的 Exactly Once 语义。即:At Least Once + 幂等性 = Exactly Once

要启用幂等性，只需要将 Producer 的参数中 enable.idompotence 设置为 true 即可。Kafka的幂等性实现其实就是将原来下游需要做的去重放在了数据上游。开启幂等性的 Producer 在初始化的时候会被分配一个 PID，发往同一 Partition 的消息会附带 Sequence Number。而 Broker 端会对<PID, Partition, SeqNumber>做缓存，当具有相同主键的消息提交时，Broker 只 会持久化一条。

但是 PID 重启就会变化，同时不同的 Partition 也具有不同主键，所以幂等性无法保证跨分区跨会话的 Exactly Once。

当然，我们可以生产者生成一个全局唯一的id给broker，broker根据该ID来保证Exactly Once。

1. **消费者**
2. 消费方式

consumer 采用 pull(拉)模式从 broker 中读取数据。

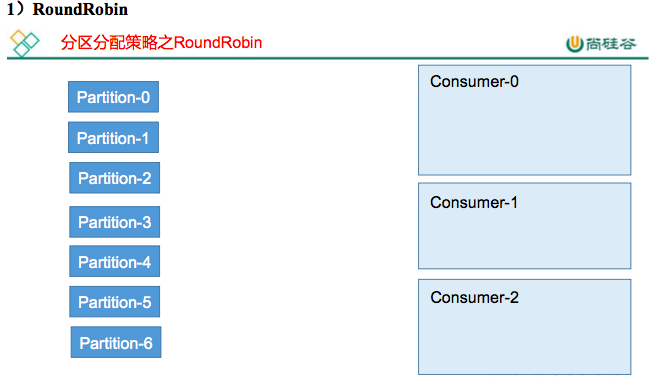
push(推)模式很难适应消费速率不同的消费者，因为消息发送速率是由 broker 决定的。 它的目标是尽可能以最快速度传递消息，但是这样很容易造成 consumer 来不及处理消息,典型的表现就是拒绝服务以及网络拥塞。而 pull 模式则可以根据 consumer 的消费能力以适当的速率消费消息。

pull 模式不足之处是，如果 kafka 没有数据，消费者可能会陷入循环中，一直返回空数据。针对这一点，Kafka 的消费者在消费数据时会传入一个时长参数 timeout，如果当前没有数据可供消费，consumer 会等待一段时间之后再返回，这段时长即为 timeout。

1. 分区分配策略

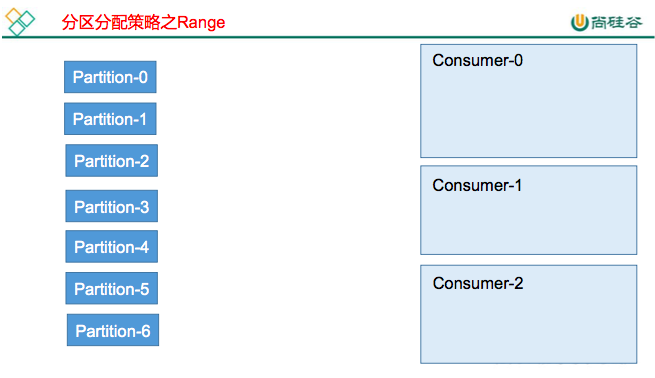
一个 consumer group 中有多个 consumer，一个 topic 有多个 partition，所以必然会涉及 到 partition 的分配问题，即确定那个 partition 由哪个 consumer 来消费。

Kafka 有两种分配策略，一是 RoundRobin，一是 Range。



Roundbin也就是轮询策略，kafka是把旗下属于同一个消费组所有的topic包含的partition整合到一起进行轮询，比如Topic1下面存在partition1、partition2；Topic2下面存在partition3、partition4；此时kafka是将partition1、partition2、partition3、partition4全整合到一起轮询给消费者消费。这种消费方式存在一个问题，比如消费者A只订阅了Topic1，消费者B只订阅了Topic2，但是这种轮询方式可能存在把Topic2的消费发给消费者A，Topic1消息发给消费者B。所以在选择这种方式的时候，必须保证消费者都是消费同一个Topic。

2)Range，系统默认采用这个

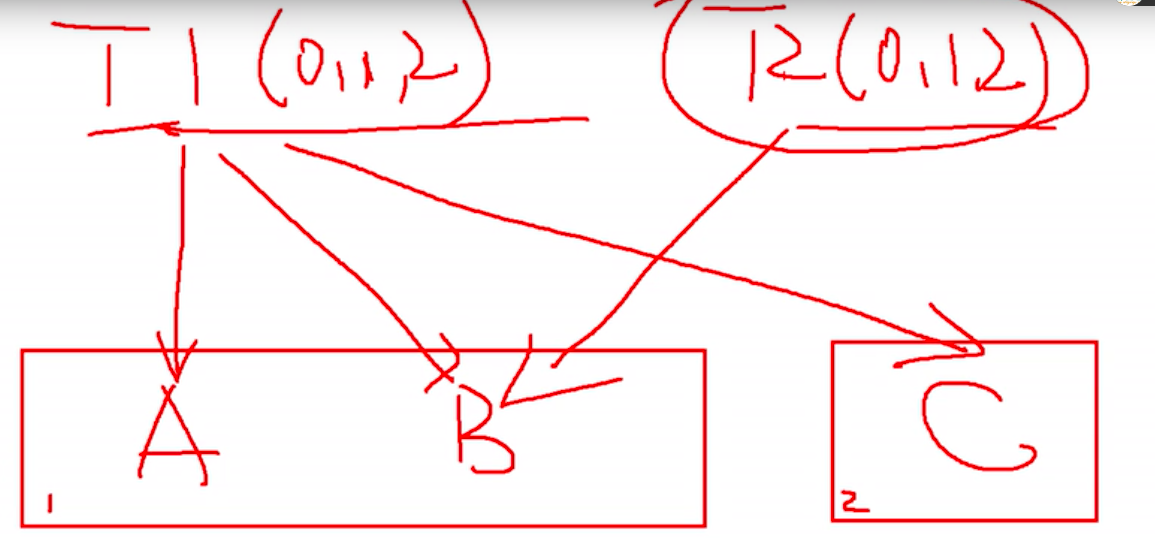


这种方式则是根据Topic进行分配，一个Topic对应一个消费者组进行分配，就不会存在轮询上面的问题，但是有可能不同Topic的消息量不一样，就造成消费者的负载不同，有的消费者负载过高，有的资源空闲的问题。

什么时候触发分配策略呢？

当消费者数量发生变化的时候，比如新增，减少时。

示例：



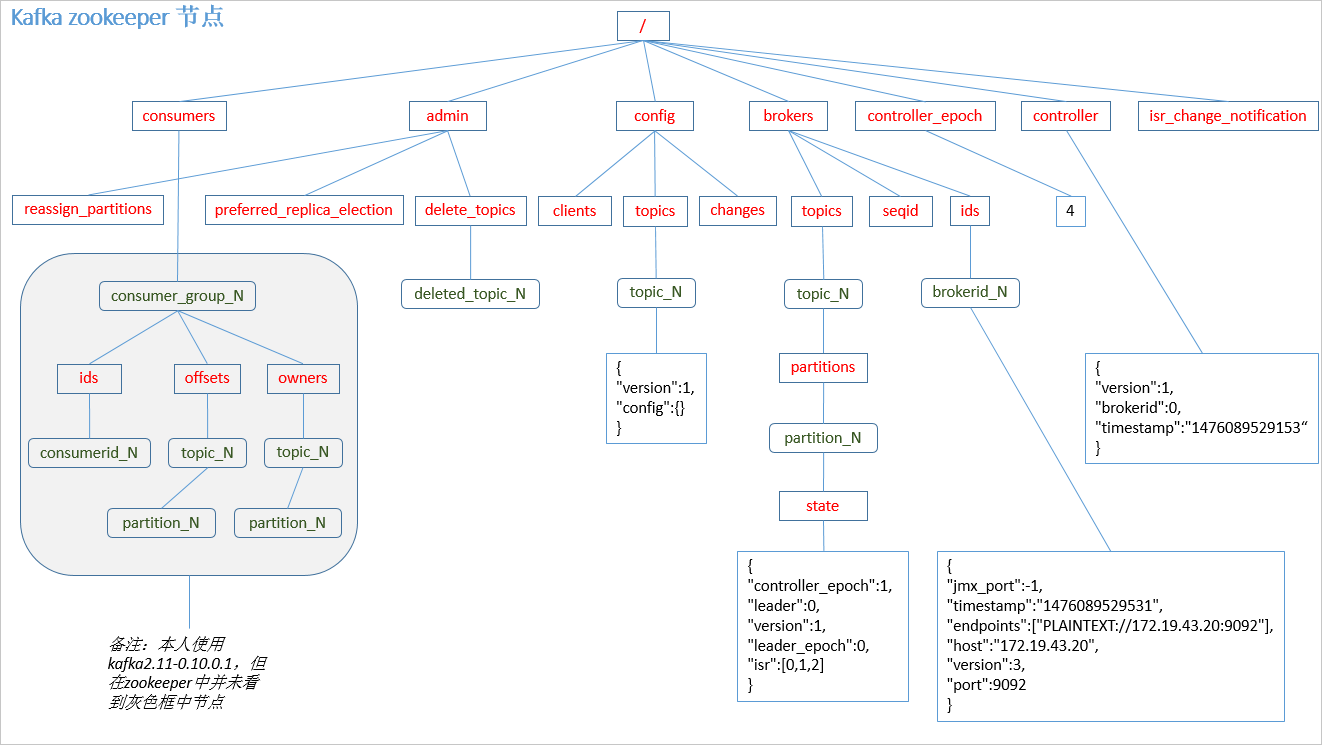
主题topic T1,T2;分别存在0、1、2三个分区，此时消费者A、B属于同一个消费组，C属于另一个消费组。

Roundbin分配方式的结果是：T1、T2的所有分区进行整合，6个轮询分配给A、B（有可能T2的消息会分给A）,T1（0，1，2）只分配给C。此方式只看消费组。

Range结果：T1（0、1、2）分给C，T1（0、1）给A，T1（2）给B，T2（0、1、2）分给B，这种方式首先看谁订阅这个主题，然后再看消费组

1. Offset的维护

由于 consumer 在消费过程中可能会出现断电宕机等故障，consumer 恢复后，需要从故障前的位置的继续消费，所以 consumer 需要实时记录自己消费到了哪个 offset，以便故障恢复后继续消费。



Kafka 0.9 版本之前，consumer 默认将 offset 保存在 Zookeeper 中，从 0.9 版本开始， consumer 默认将 offset 保存在 Kafka 一个内置的 topic 中，该 topic 为\_\_consumer\_offsets。

如何去确定offset？

Offset是根据消费者组、Topic、partition去确定的，首先同一个组的消费者，消费者增加时从offset开始消费消息，而不是从头开始。

为什么offset不是根据消费者去确定？

因为消费者挂了，这个offset就失效了，所以不能跟消费者绑定。

1）、修改配置文件 consumer.properties（kafka的配置文件）

2)、读取 offset

0.11.0.0 之前版本:

|  |
| --- |
| bin/kafka-console-consumer.sh --topic \_\_consumer\_offsets -- zookeeper hadoop102:2181 --formatter "kafka.coordinator.GroupMetadataManager\$OffsetsMessageFormatter" --consumer.config config/consumer.properties --from-beginning |

0.11.0.0 之后版本(含):

|  |
| --- |
| bin/kafka-console-consumer.sh --topic \_\_consumer\_offsets -- zookeeper hadoop102:2181 --formatter "kafka.coordinator.group.GroupMetadataManager\$OffsetsMessageForm atter" --consumer.config config/consumer.properties --from- beginning |

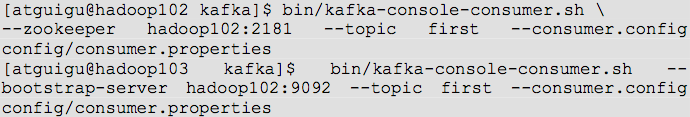
1. 消费者组案例

需求:测试同一个消费者组中的消费者，同一时刻只能有一个消费者消费。

(1)、在 hadoop102、hadoop103 上修改/opt/module/kafka/config/consumer.properties 配置 文件中的 group.id 属性为任意组名。



(2)在 hadoop102、hadoop103 上分别启动消费者



(3)在 hadoop104 上启动生产者



(4)查看 hadoop102 和 hadoop103 的接收者。

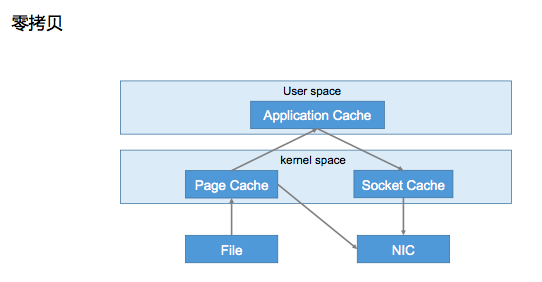
同一时刻只有一个消费者接收到消息。

1. **Kafka高效读写数据**

1)顺序写磁盘

Kafka 的 producer 生产数据，要写入到 log 文件中，写的过程是一直追加到文件末端， 为顺序写。官网有数据表明，同样的磁盘，顺序写能到 600M/s，而随机写只有 100K/s。这与磁盘的机械机构有关，顺序写之所以快，是因为其省去了大量磁头寻址的时间。

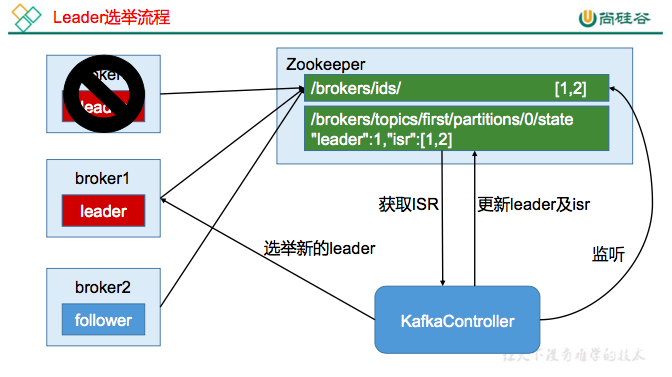
2)零复制技术



1. **Zookeeper在Kafka中的作用**

Kafka 集群中有一个 broker 会被选举为 Controller，负责管理集群 broker 的上下线，所有 topic 的分区副本分配和 leader 选举等工作。

Controller 的管理工作都是依赖于 Zookeeper 的。 以下为 partition 的 leader 选举过程:



1. 安装配置

其他见pdf文件，在这里我只说明修改kafka的配置文件

在conf下找到server.properties，红色是重点修改的地方，其他的可以默认。

|  |
| --- |
| #broker 的全局唯一编号，不能重复  broker.id=0  #删除 topic 功能使能  delete.topic.enable=true  #处理网络请求的线程数量  num.network.threads=3  #用来处理磁盘 IO 的现成数量  num.io.threads=8  #发送套接字的缓冲区大小  socket.send.buffer.bytes=102400  #接收套接字的缓冲区大小  socket.receive.buffer.bytes=102400  #请求套接字的缓冲区大小  socket.request.max.bytes=104857600  #kafka 运行日志存放的路径  log.dirs=/opt/module/kafka/logs  #topic 在当前 broker 上的分区个数  num.partitions=1  #用来恢复和清理 data 下数据的线程数量  num.recovery.threads.per.data.dir=1  #segment 文件保留的最长时间，超时将被删除  log.retention.hours=168  #配置连接 Zookeeper 集群地址  zookeeper.connect=hadoop102:2181,hadoop103:2181,hadoop104:2181 |

1. 命令行操作
2. 查看当前服务器中的所有 topic

bin/kafka-topics.sh --zookeeper hadoop102:2181 --list

--zookeeper：zookeeper的地址，hadoop102是zk的主机名或者IP地址也行，2181是端口号

2)创建 topic

bin/kafka-topics.sh --zookeeper hadoop102:2181 --create --replication-factor 3 --partitions 1 --topic first

--replication-factor：副本数

--topic:主题名称 first

--partitions：分区数

3)删除 topic

bin/kafka-topics.sh --zookeeper hadoop102:2181 --delete --topic first

注：需要 server.properties 中设置 delete.topic.enable=true 否则只是标记删除

4)发送消息

bin/kafka-console-producer.sh --broker-list hadoop102:9092 --topic first

5)消费消息

bin/kafka-console-consumer.sh --zookeeper hadoop102:2181 --topic first

bin/kafka-console-consumer.sh --bootstrap-server hadoop102:9092 --topic first

bin/kafka-console-consumer.sh --bootstrap-server hadoop102:9092 --from-beginning --topic first

--from-beginning:会把主题中以往所有的数据都读取出来。而过往数据最大缓存时间是7天，超过7天就会消失

6)查看某个 Topic 的详情

bin/kafka-topics.sh --zookeeper hadoop102:2181 --describe --topic first

7)修改分区数

bin/kafka-topics.sh --zookeeper hadoop102:2181 --alter --topic first --partitions 6

1. 事务

Kafka 从 0.11 版本开始引入了事务支持。事务可以保证 Kafka 在 Exactly Once 语义的基础上，生产和消费可以跨分区和会话，要么全部成功，要么全部失败。

1. **Producer事务**

为了实现跨分区跨会话的事务，需要引入一个全局唯一的 Transaction ID(kafka客户端给的，不是kafka生成的)，并将 Producer 获得的 PID 和 Transaction ID 绑定。这样当 Producer 重启后就可以通过正在进行的 Transaction ID 获得原来的 PID。

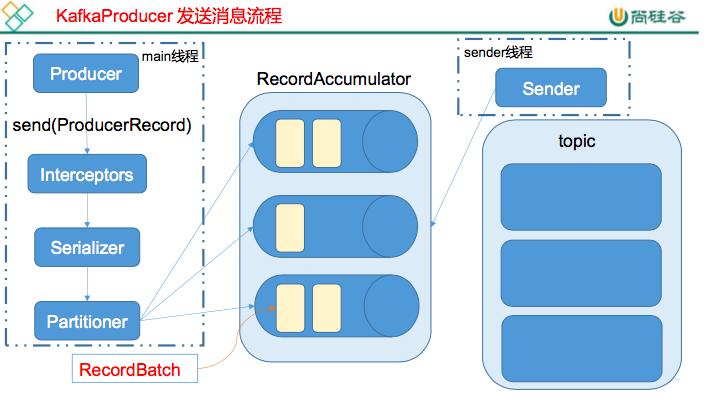
为了管理 Transaction，Kafka 引入了一个新的组件 Transaction Coordinator [koʊˈɔːrdɪneɪtər]。Producer 就是通过和 Transaction Coordinator 交互获得 Transaction ID 对应的任务状态。Transaction Coordinator 还负责将事务所有写入 Kafka 的一个内部 Topic，这样即使整个服务重启，由于事务状态得到保存，进行中的事务状态可以得到恢复，从而继续进行

1. **Consumer事务**

上述事务机制主要是从 Producer 方面考虑，对于 Consumer 而言，事务的保证就会相对较弱，尤其是无法保证 Commit 的信息被精确消费。这是由于 Consumer 可以通过 offset 访问任意信息，而且不同的 Segment File 生命周期不同，同一事务的消息可能会出现重启后被删除的情况。

1. API
2. **Producer API**
3. 消息发送流程

Kafka 的 Producer 发送消息采用的是异步发送的方式。在消息发送的过程中，涉及到了 两个线程——main 线程和 Sender 线程，以及一个线程共享变量——RecordAccumulator [əˈkjuːmjəleɪtər] （累加器）。main 线程将消息发送给 RecordAccumulator，Sender 线程不断从 RecordAccumulator 中拉取消息发送到 Kafka broker。



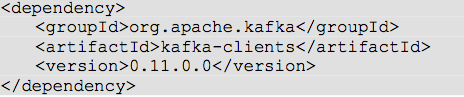
相关参数:

batch.size:只有数据积累到 batch.size 之后，sender 才会发送数据。

linger.ms:如果数据迟迟未达到 batch.size，sender 等待 linger.time 之后就会发送数据。

1. 异步发送API

1)导入依赖



2)编写代码

需要用到的类:

KafkaProducer:需要创建一个生产者对象，用来发送数据

ProducerConfig:获取所需的一系列配置参数

ProducerRecord:每条数据都要封装成一个 ProducerRecord 对象

1.不带回调函数的 API，如下参数也可以使用ProducerConfig类

|  |
| --- |
| import org.apache.kafka.clients.producer.\*; import java.util.Properties;  import java.util.concurrent.ExecutionException; public class CustomProducer {   public static void main(String[] args) throws ExecutionException, InterruptedException {   Properties props = new Properties();  //kafka 集群，broker-list  props.put("bootstrap.servers", "hadoop102:9092");  props.put("acks", "all");  //重试次数  props.put("retries", 1);  //批次大小  props.put("batch.size", 16384);  //等待时间  props.put("linger.ms", 1);  //RecordAccumulator 缓冲区大小  props.put("buffer.memory", 33554432);  props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");  props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");  //创建生产者对象  Producer<String, String> producer = new KafkaProducer<>(props);  //发送数据  for (int i = 0; i < 100; i++) { producer.send(new ProducerRecord<String, String>("first",  Integer.*toString*(i), Integer.*toString*(i)));  }  //关闭  producer.close();  } } |

2.带回调函数的 API

回调函数会在 producer 收到 ack 时调用，为异步调用，该方法有两个参数，分别是RecordMetadata 和 Exception，如果 Exception 为 null，说明消息发送成功，如果 Exception 不为 null，说明消息发送失败。

注意:消息发送失败会自动重试，不需要我们在回调函数中手动重试。

|  |
| --- |
| import org.apache.kafka.clients.producer.\*; import java.util.Properties; import java.util.concurrent.ExecutionException; public class CustomProducer {  public static void main(String[] args) throws ExecutionException, InterruptedException {  Properties props = new Properties();  props.put("bootstrap.servers", "hadoop102:9092");//kafka 集  群，broker - list props.put("acks", "all");  props.put("retries", 1);  //重试次数  props.put("batch.size", 16384);  //批次大小  props.put("linger.ms", 1);  //等待时间  props.put("buffer.memory", 33554432);  //RecordAccumulator 缓 冲区大小  props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");  props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");  Producer<String, String> producer = new KafkaProducer<>(props);  for (int i = 0; i < 100; i++) {  producer.send(new ProducerRecord<String, String>("first",  Integer.*toString*(i), Integer.*toString*(i)), new Callback() {  //回调函数，该方法会在 Producer 收到 ack 时调用，为异步调用 @Override  public void onCompletion(RecordMetadata metadata,  Exception exception) {  if (exception == null) {  System.*out*.println("success->" +  metadata.offset());  } else {  exception.printStackTrace();  }  }  });  }  producer.close();  } } |

1. 自定义分区生产者

自定义一个类MyPartitioner继承Partitioner，实现partition方法，该方法就是进行分区的方法



然后在Properties类中进行配置

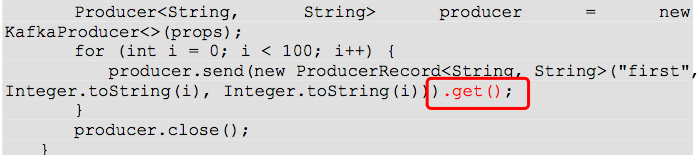


1. 同步发送API

同步发送的意思就是，一条消息发送之后，会阻塞当前线程，直至返回 ack。常用到需要保证消息发送顺序的时候。

由于 send 方法返回的是一个 Future 对象，根据 Futrue 对象的特点，我们也可以实现同步发送的效果，只需在调用 Future 对象的 get 方发即可。

如下与异步发送的相同代码则不贴了，只截图不同的地方



1. **Consumer API**

Consumer 消费数据时的可靠性是很容易保证的，因为数据在 Kafka 中是持久化的，故不用担心数据丢失问题。

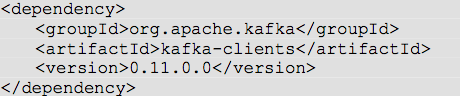
由于 consumer 在消费过程中可能会出现断电宕机等故障，consumer 恢复后，需要从故障前的位置的继续消费，所以 consumer 需要实时记录自己消费到了哪个 offset，以便故障恢 复后继续消费。

所以 offset 的维护是 Consumer 消费数据必须考虑的问题。

消费者在第一次启动的时候会从zookeeper或者broker里面读取Offset值，此后消费者会自己在本地维护一个offset，直到消费者进程挂掉，而不会再从zookeeper中获取该值。

1. **自动提交 offset**

1)导入依赖



2)编写代码

需要用到的类:

KafkaConsumer:需要创建一个消费者对象，用来消费数据

ConsumerConfig:获取所需的一系列配置参数

ConsuemrRecord:每条数据都要封装成一个 ConsumerRecord 对象

为了使我们能够专注于自己的业务逻辑，Kafka 提供了自动提交 offset 的功能。 自动提交 offset 的相关参数:

enable.auto.commit:是否开启自动提交 offset 功能

auto.commit.interval.ms:自动提交 offset 的时间间隔

以下为自动提交 offset 的代码:

|  |
| --- |
| import org.apache.kafka.clients.consumer.ConsumerRecord; import org.apache.kafka.clients.consumer.ConsumerRecords; import org.apache.kafka.clients.consumer.KafkaConsumer; import java.util.Arrays; import java.util.Properties;  public class CustomConsumer {  public static void main(String[] args) {  //创建消费者配置信息  Properties props = new Properties();  //连接的集群  props.put("bootstrap.servers", "hadoop102:9092");  //消费者组  props.put("group.id", "test");  //开启自动提交  props.put(ConsumerConfig.ENABLE\_AUTO\_COMMIT\_CONFIG, true);  //自动提交的延时  props.put("auto.commit.interval.ms", "1000");  //设置key的反序列化类  props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");  //设置value的反序列化类  props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");  KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);  //配置主题Topic  consumer.subscribe(Arrays.*asList*("first"));  //消费者循环取消息  while (true) {  //阻塞方法，有消息就会获取，无则等待，可以设置超时时间  ConsumerRecords<String, String> records = consumer.poll(100);  for (ConsumerRecord<String, String> record : records)  System.*out*.printf("offset = %d, key = %s, value = %s%n", record.offset(), record.key(), record.value());  }  } } |

1. **手动提交offset**

虽然自动提交 offset 十分简介便利，但由于其是基于时间提交的，开发人员难以把握 offset 提交的时机（比如提交太早，消息消费出错，导致消息丢失）。因此 Kafka 还提供了手动提交 offset 的 API。

手动提交 offset 的方法有两种:分别是 commitSync(同步提交)和 commitAsync(异步提交)。两者的相同点是，都会将本次 poll 的一批数据最高的偏移量提交;不同点是， commitSync 阻塞当前线程，一直到提交成功，并且会自动失败重试(由不可控因素导致，也会出现提交失败);而 commitAsync 则没有失败重试机制，故有可能提交失败。

同步提交offset

首先要关闭自动提交



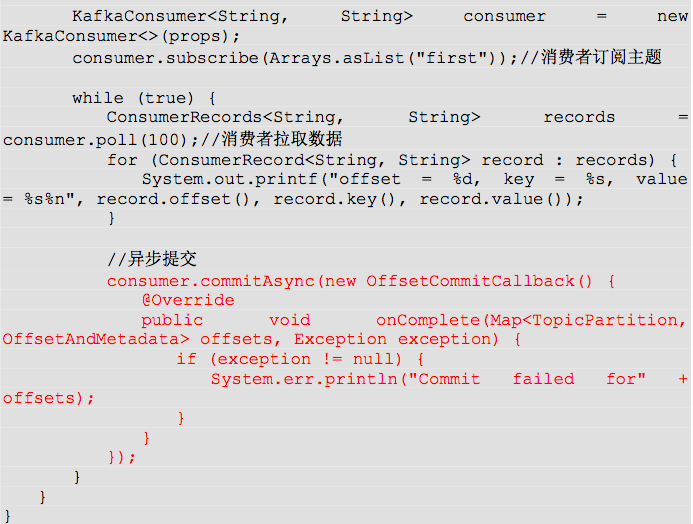


异步提交offset

虽然同步提交 offset 更可靠一些，但是由于其会阻塞当前线程，直到提交成功。因此吞吐量会收到很大的影响。因此更多的情况下，会选用异步提交 offset 的方式。

以下为异步提交 offset 的示例:





数据漏消费和重复消费分析

无论是同步提交还是异步提交 offset，都有可能会造成数据的漏消费或者重复消费。先提交 offset 后消费，有可能造成数据的漏消费;而先消费后提交 offset，有可能会造成数据的重复消费。

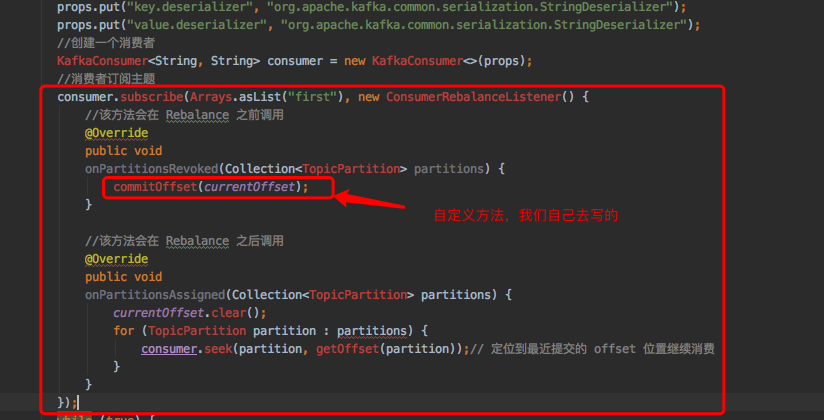
1. **自定义存储 offset**

Kafka 0.9 版本之前，offset 存储在 zookeeper，0.9 版本及之后，默认将 offset 存储在 Kafka 的一个内置的 topic 中。除此之外，Kafka 还可以选择自定义存储 offset。

offset 的维护是相当繁琐的，因为需要考虑到消费者的 Rebalace。

当有新的消费者加入消费者组、已有的消费者退出消费者组或者所订阅的主题的分区发生变化，就会触发到分区的重新分配，重新分配的过程叫做 Rebalance。

消费者发生 Rebalance 之后，每个消费者消费的分区就会发生变化。因此消费者要首先获取到自己被重新分配到的分区，并且定位到每个分区最近提交的 offset 位置继续消费。要实现自定义存储offset，需要借助 ConsumerRebalanceListener，以下为示例代码，其中提交和获取 offset 的方法，需要根据所选的 offset 存储系统自行实现。



* 1. **自定义拦截器**

Producer 拦截器(interceptor)是在 Kafka 0.10 版本被引入的，主要用于实现 clients 端的定制化控制逻辑。

对于 producer 而言，interceptor 使得用户在消息发送前以及 producer 回调逻辑前有机会对消息做一些定制化需求，比如修改消息等。同时，producer 允许用户指定多个 interceptor 按序作用于同一条消息从而形成一个拦截链(interceptor chain)。Intercetpor 的实现接口是org.apache.kafka.clients.producer.ProducerInterceptor，其定义的方法包括:

(1)configure(configs)

获取配置信息和初始化数据时调用。

(2)onSend(ProducerRecord):

该方法封装进 KafkaProducer.send 方法中，即它运行在用户主线程中。Producer 确保在消息被序列化以及计算分区前调用该方法。用户可以在该方法中对消息做任何操作，但最好保证不要修改消息所属的 topic 和分区，否则会影响目标分区的计算。

(3)onAcknowledgement(RecordMetadata, Exception):

该方法会在消息从 RecordAccumulator 成功发送到 Kafka Broker 之后，或者在发送过程中失败时调用。并且通常都是在 producer 回调逻辑触发之前。onAcknowledgement 运行在 producer 的 IO 线程中，因此不要在该方法中放入很重的逻辑，否则会拖慢 producer 的消息 发送效率。

(4)close:

关闭 interceptor，主要用于执行一些资源清理工作

如前所述，interceptor 可能被运行在多个线程中，因此在具体实现时用户需要自行确保线程安全。另外倘若指定了多个 interceptor，则 producer 将按照指定顺序调用它们，并仅仅是捕获每个 interceptor 可能抛出的异常记录到错误日志中而非在向上传递。这在使用过程中要特别留意。

案例

1)需求:

实现一个简单的双 interceptor 组成的拦截链。第一个 interceptor 会在消息发送前将时间 戳信息加到消息 value 的最前部;第二个 interceptor 会在消息发送后更新成功发送消息数或 失败发送消息数。



增加时间戳拦截器

|  |
| --- |
| public class TimeInterceptor implements ProducerInterceptor<String, String> {  @Override  public void configure(Map<String, ?> configs) {  }  @Override  public ProducerRecord<String, String> onSend(ProducerRecord<String, String> record) { // 创建一个新的 record，把时间戳写入消息体的最前部  return new ProducerRecord(record.topic(), record.partition(), record.timestamp(), record.key(),  System.*currentTimeMillis*() + "," + record.value().toString());  }  @Override  public void onAcknowledgement(RecordMetadata Exception exception) {  }  @Override  public void close() {  } } |

(2)统计发送消息成功和发送失败消息数，并在 producer 关闭时打印这两个计数器

|  |
| --- |
| public class CounterInterceptor implements ProducerInterceptor<String, String> {  private int errorCounter = 0;  private int successCounter = 0;  @Override   public void configure(Map<String, ?> configs) {  }  @Override  public ProducerRecord<String, onSend(ProducerRecord<String, String> record){  return record;  }  @Override  public void onAcknowledgement(RecordMetadata Exception exception) {  // 统计成功和失败的次数  if (exception == null) {  successCounter++;  } else {  errorCounter++;  }  }  @Override  public void close() {  // 保存结果   System.*out*.println("Successful sent: " + successCounter);  System.*out*.println("Failed sent: " + errorCounter);  } } |

(3)producer 主程序，省略其他代码

|  |
| --- |
| // 2 构建拦截链  List<String> interceptors = new ArrayList<>(); interceptors.add("com.atguigu.kafka.interceptor.TimeInterce ptor"); interceptors.add("com.atguigu.kafka.interceptor.CounterInterceptor"); props.put(ProducerConfig.INTERCEPTOR\_CLASSES\_CONFIG, interceptors); |

测试效果



1. kafka监控

Eagle

1. aa