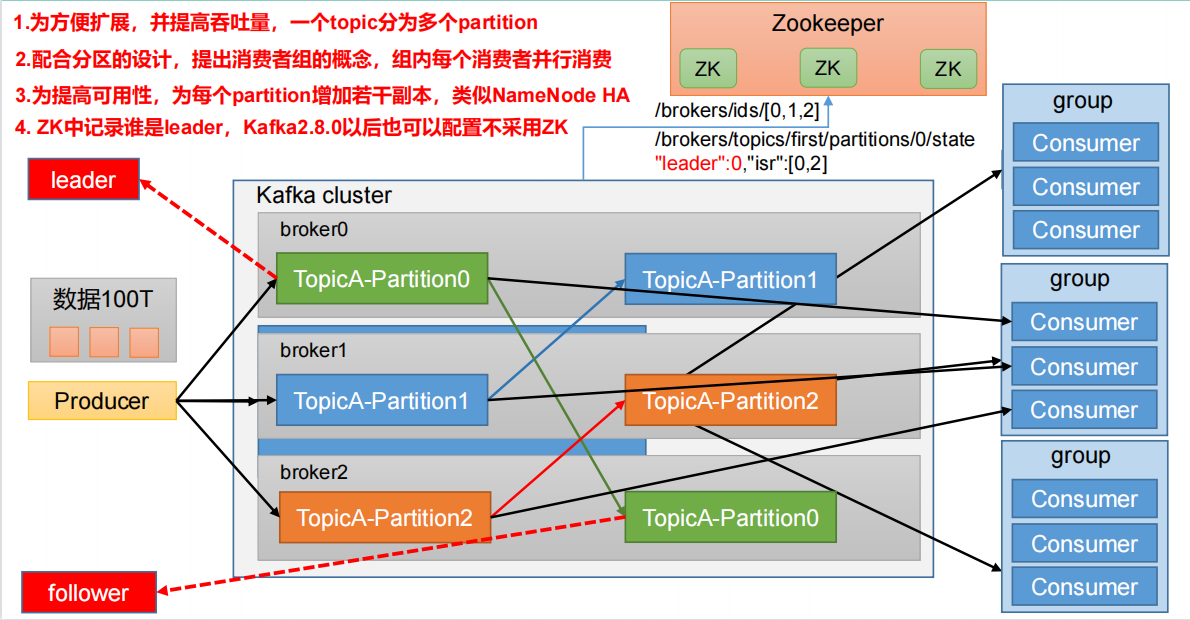
# Kafka总结

# 什么是kafka以及kafka的基础架构

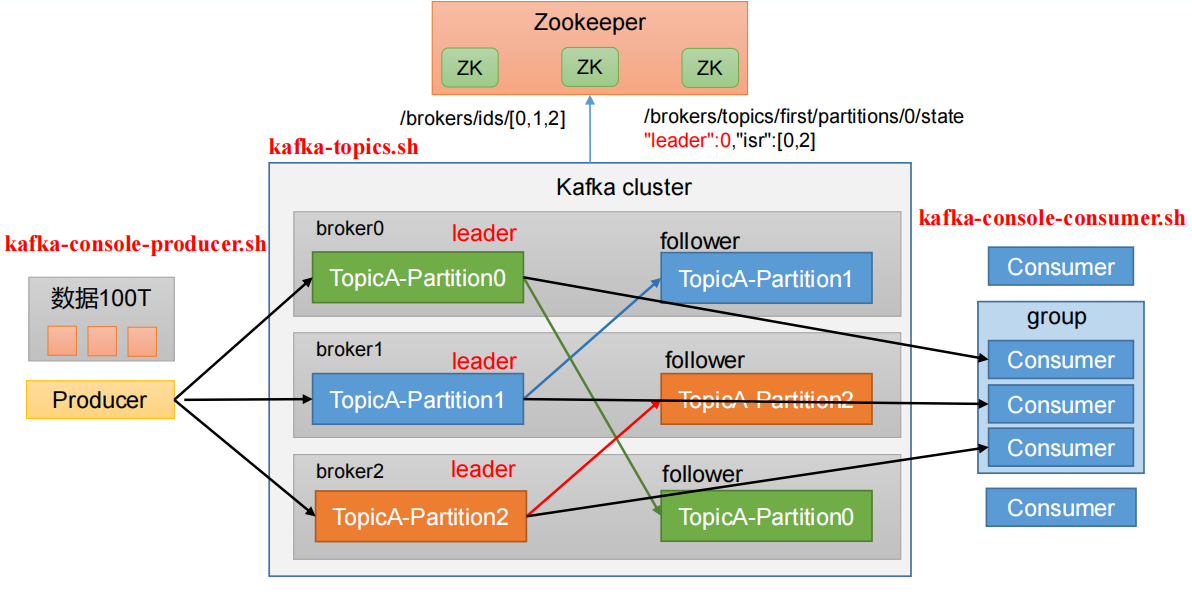
Kafka是一个高吞吐的分布式的消息系统，是基于发布/订阅模式的消息队列。



## 术语

* Producer：消息生产者，就是向 Kafka broker 发消息的客户端。
* Consumer：消息消费者，向 Kafka broker 拉取消息的客户端。
* 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。
* 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。
* 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。
* 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，每个broker上的topic叫做一个partition（一个 topic 可以分为多个 partition），每个 partition 是一个有序的队列。
* Replica：副本。一个 topic 的每个分区都有若干个副本，即一个 Leader 和若干个Follower。
* 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。
* 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。
* ISR: in-sync replica set（ISR）, 意为和Leader保持同步的Follower+Leader集合(leader：0，isr:0,1,2)。如果Follower长时间未向Leader发送通信请求或同步数据，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30s。例如2超时，(leader:0, isr:0,1)。
* AR: Kafka分区中的所有副本统称。
* OSR ：表示 Follower 与 Leader副本同步时，延迟过多的副本。AR=ISR+OSR
* LEO（Log End Offset）：每个副本的最后一个offset，LEO其实就是最新的offset + 1。
* HW（High Watermark）：所有副本中最小的LEO 。

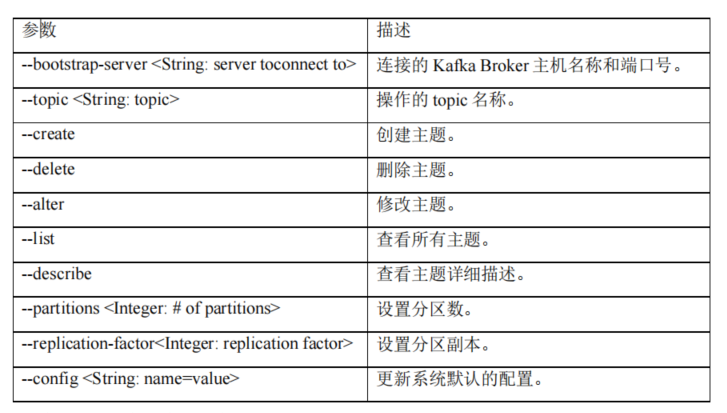
# Kafka 命令行操作



## 3.1、主题命令行操作

1）查看操作主题命令参数

[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh



2）查看当前服务器中的所有 topic

[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --bootstrap-server

hadoop102:9092 –list

3）创建 first topic

[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --bootstrap-server

hadoop102:9092 --create --partitions 1 --replication-factor 3 --topic first

选项说明：

--topic 定义 topic 名

--replication-factor 定义副本数

--partitions 定义分区数

4）查看 first 主题的详情

[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --bootstrap-server

hadoop102:9092 --describe --topic first

5）修改分区数（注意：分区数只能增加，不能减少）

[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --bootstrap-server

hadoop102:9092 --alter --topic first --partitions 3

6）再次查看 first 主题的详情

[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --bootstrap-server

hadoop102:9092 --describe --topic first

7）删除 topic（学生自己演示）

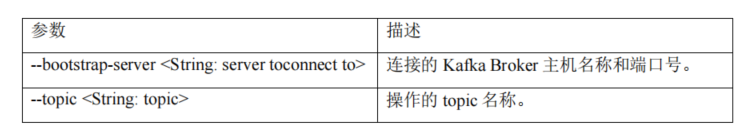
[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --bootstrap-server

hadoop102:9092 --delete --topic first

## 3.2、生产者命令行操作

1）查看操作生产者命令参数

[atguigu@hadoop102 kafka]$ bin/kafka-console-producer.sh



2）发送消息

[atguigu@hadoop102 kafka]$ bin/kafka-console-producer.sh --

bootstrap-server hadoop102:9092 --topic first

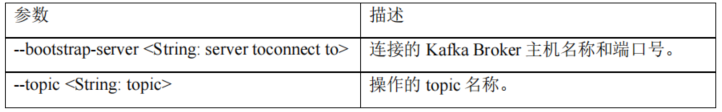
>hello world

>atguigu atguigu

## 3.3、消费者命令行操作

1）查看操作消费者命令参数

[atguigu@hadoop102 kafka]$ bin/kafka-console-consumer.sh



2）消费消息

（1）消费 first 主题中的数据。

[atguigu@hadoop102 kafka]$ bin/kafka-console-consumer.sh --

bootstrap-server hadoop102:9092 --topic first

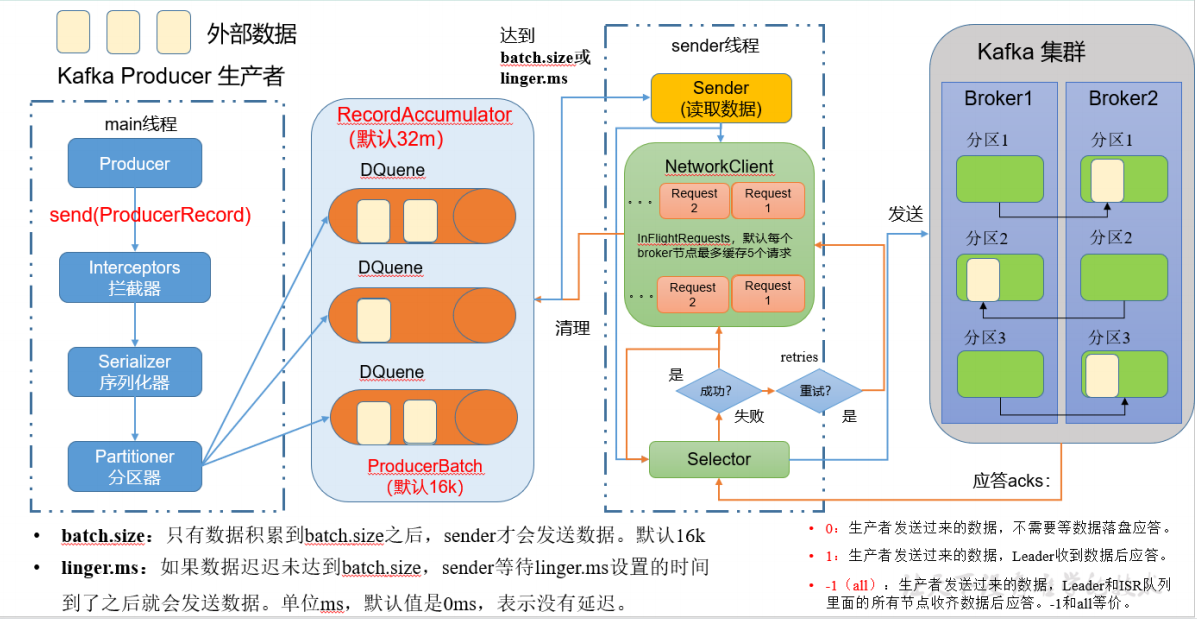
（2）把主题中所有的数据都读取出来（包括历史数据）。

[atguigu@hadoop102 kafka]$ bin/kafka-console-consumer.sh --

bootstrap-server hadoop102:9092 --from-beginning --topic first

# Kafka 生产者

## 4.1、生产者消息发送流程



**外部数据 ----> 存储到kafka集群中**

在数据发送的过程中会涉及到两个线程——main 线程和 Sender 线程，当接受外部传过来的数据的时候，会先创建一个main线程，在main线程中创建producer对象，然后调用send方法，将数据进行发送，然后会经过拦截器，对发送的数据进行处理、加工，再经过序列化器，对传输的数据进行序列化，在根据分区器的分区策略对传输的数据进行分区处理，到达缓冲区

（RecordAccumulator记录收集器，里面维护了一个一个双端的缓存队列和一个内存池，内存池的默认大小是32m，当向双端队列中发送数据的时候，会创建批次大小，当创建批次的时候，会从内存池中取出内存，当数据发送到kafka集中后，清理批次的时候，会释放内存，放回到内存池中）中指定的分区（一个分区会创建一个队列），在分区中按照批次的方式进行存储数据，每一个批次的默认大小（batch.size）为16k，当一个批次的（数据大小积累到batch.size（默认16k）大小时）或者（数据没有积累到batch.size，但是到达了延迟时间（linger.ms 默认是0ms，表示没有延迟）），唤醒sender线程，然后会发送请求从分区中拉取数据，拉取数据的方式是以brokerId为key,所有分区的请求为value放到队列中，如果发送数据的第一个请求到达集群中的某一个broker没有应答，允许继续发送请求，默认每个broker节点最多缓存5个请求，通过selector发送数据，数据发送成功之后，会有应答机制，返回acks，应答级别有3种，如果反馈回来的请求是成功，则会删除发送数据成功的请求以及清理分区中请求中拉取的数据，如果失败会进行重试，重试的次数（默认是Int的最大值，可以进行修改，一般是3-5次），

0：生产者发送过来的数据，不需要等数据落盘应答。1：生产者发送过来的数据，Leader收到数据后应答。 -1（all）：生产者发送过来的数据，Leader和ISR队列，里面的所有节点收齐数据后应答。-1和all等价。

## 4.2、异步发送 API

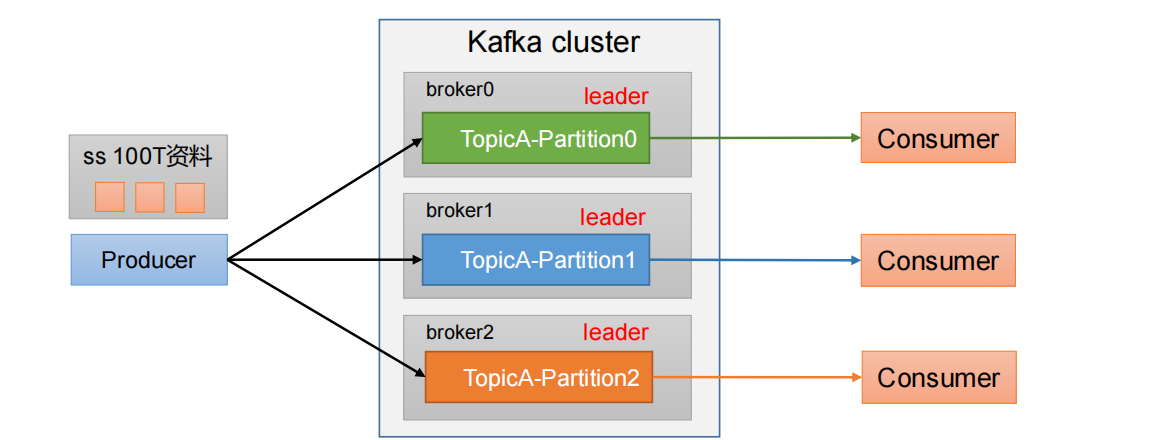
## 4.3、同步发送 API

## 4.4、生产者分区

### 4.4.1、分区的好处

（1）便于合理使用存储资源，每个Partition在一个Broker上存储，可以把海量的数据按照分区切割成一块一块的数据存储在多台Broker上。合理控制分区的任务，可以实现负载均衡的效果。

（2）提高并行度，生产者可以以分区为单位发送数据；消费者可以以分区为单位进行消费数据。



### 4.4.2、生产者发送消息的分区策略

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

例如partition=0，所有数据写入分区0

（2）没有指明partition值，但是有key（一般是数据库表的表名字）的情况下，将key的hash值与topic的partition数进行取余得到partition值；

例如：key1的hash值=5， key2的hash值=6 ，topic的partition数=2，那 么key1 对应的value1写入1号分区，key2对应的value2写入0号分区。

（3）既没有partition值又没有key值的情况下，Kafka采用Sticky Partition（黏性分区器），会随机选择一个分区，并尽可能一直使用该分区，待该分区的当前batch已满或者当前批次的等待时间到了，Kafka再随机一个分区进行使用（和上一次的分区不同）。

例如：第一次随机选择0号分区，等0号分区当前批次满了（默认16k）或者linger.ms设置的时间到，Kafka再随机一个分区进行使用（如果还是0会继续随机）。

### 4.4.3、自定义分区

#### 4.4.3.1、实现步骤

（1）自定义类实现 Partitioner 接口。

（2）重写 partition()方法。

（3）使用分区器中的方法，在生产者的配置中添加分区器参数。

|  |
| --- |
| package com.song.kafka\_demo.producer;  import org.apache.kafka.clients.producer.Partitioner;  import org.apache.kafka.common.Cluster;  import java.util.Map;  public class MyPartitioner implements Partitioner {  @Override  public int partition(String topic, Object key, byte[] keyBytes, Object value, byte[] valueBytes, Cluster cluster) {  // 获取数据 song  String msgValues = value.toString();  int partition;  if (msgValues.contains("song")) {  partition = 0;  } else {  partition = 1;  }  return partition;  }  @Override  public void close() {  }  @Override  public void configure(Map<String, ?> configs) {  }  }  ===============================================  // 关联自定义分区器  properties.put(ProducerConfig.PARTITIONER\_CLASS\_CONFIG,"com.song.kafka\_demo.producer.MyPartitioner"); |

## 4.5、生产经验——生产者如何提高吞吐量

• RecordAccumulator：修改缓冲区大小，默认32m，修改为64m

• batch.size： 修改批次大小，默认16k

• linger.ms：修改等待时间，修改为5-100ms一次拉一个，来了就走

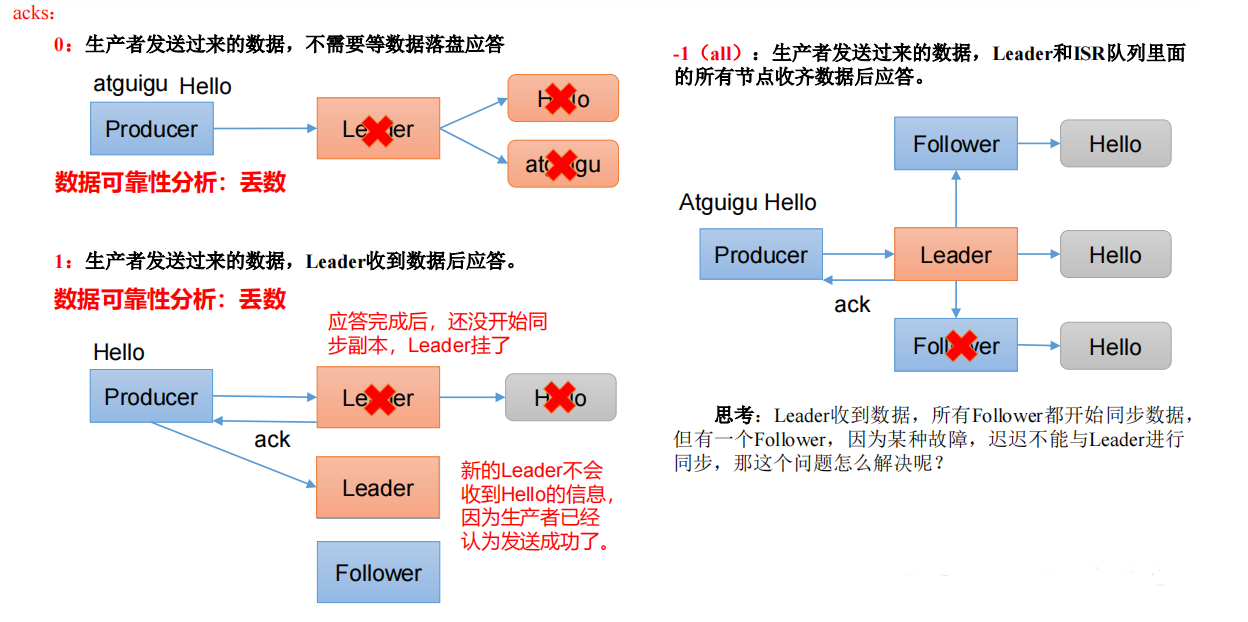
• compression.type：压缩snappy

## 4.6、生产经验——数据可靠性

### 4.6.1、ack 应答原理

#### 4.6.1.1、应答级别

* 0：生产者发送过来的数据，不需要等数据落盘应答。
* 1：生产者发送过来的数据，Leader收到数据后应答。
* -1（all）：生产者发送过来的数据，Leader和ISR队列，里面的所有节点收齐数据后应答。-1和all等价。

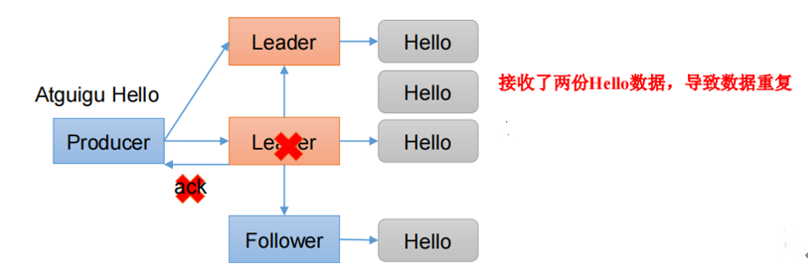


1. Acks为0，生产者发送过来的数据，不需要等数据落盘应答，但是有可能在落盘的过程中，leader服务器挂了，但是已经返回成功的信息，所以则会导致丢失数据。
2. Acks为1，生产者发送过来的数据，Leader收到数据后再去应答，但是有可能在应答完成之后，还没有开始同步副本数据，Leader挂了，然后会在剩余的follower中重新选择出一个Leader，但是新的Leader不会接受到之前发送的数据，因为之前的Leader已经返回发送成功的消息了，所以则会导致数据丢失。
3. Acks为-1，生产者发送过来的数据，Leader和ISR队列中的所有节点都收齐数据之后才去应答，但是如果其中的一个Follower出现故障迟迟不能应答怎么办？在Leader中维护了一个动态的ISR（in-sync replica set），意为和Leader保持同步的Follower+Leader集合(leader：0，isr:0,1,2（brokerId）)，如果在30s（该时间阈值由replica.lag.time.max.ms参数设定，默认30s）内，Follower未向Leader发送通信请求或同步数据，则该Follower将被踢出ISR。

#### 4.6.1.2、最终解决方式

数据完全可靠条件 = ACK级别设置为-1 + 分区副本大于等于2 + ISR里应答的最小副本数量大于等于2，

**但是使用此方式会造成数据重复性问题，解决方式看下面的数据去重**



acks： -1（all）：生产者发送过来的数据，Leader和ISR队列里面的所有节点收齐数据后应答，接受完数据之后，在未进行应答反馈之前，leader挂掉，会在follower中重新选举leader，producer未接受到成功的反馈，所以回向leader再次发送数据，就造成了消息重复的问题。

### 4.6.2、可靠性总结

* acks=0，生产者发送过来数据就不管了，可靠性差，效率高；
* acks=1，生产者发送过来数据Leader应答，可靠性中等，效率中等；
* acks=-1，生产者发送过来数据Leader和ISR队列里面所有Follwer应答，可靠性高，效率低；

**在生产环境中：**

* acks=0很少使用；
* acks=1，一般用于传输普通日志，允许丢个别数据；
* acks=-1，一般用于传输和钱相关的数据，对可靠性要求比较高的场景。

## 4.7、生产经验——数据去重

### 数据传递语义

* 至少一次（At Least Once）= ACK级别设置为-1 + 分区副本大于等于2 + ISR里应答的最小副本数量大于等于2
* 最多一次（At Most Once）= ACK级别设置为0
* 精确一次（Exactly Once）：对于一些非常重要的信息，比如和钱相关的数据，要求数据既不能重复也不丢失。

**总结：**

* At Least Once可以保证数据不丢失，但是不能保证数据不重复；
* At Most Once可以保证数据不重复，但是不能保证数据不丢失。

### 精确一次

 精确一次（Exactly Once）：对于一些非常重要的信息，比如和钱相关的数据，要求数据既不能重复也不丢失。依赖于幂等性和事务

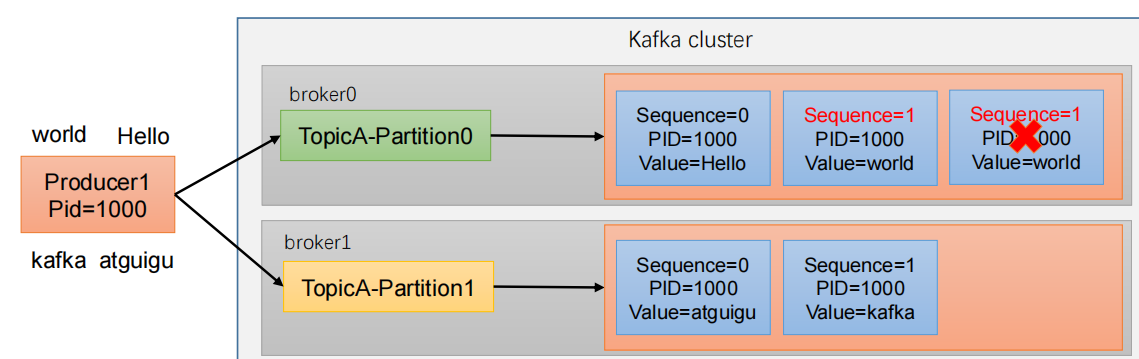
#### 4.7.2.1、幂等性

幂等性就是指Producer不论向Broker发送多少次重复数据，Broker端都只会持久化一条，保证了不重复。

精确一次（Exactly Once） = 幂等性 + 至少一次（ ack=-1 + 分区副本数>=2 + ISR最小副本数量>=2）。

重复数据的判断标准：具有<PID, Partition, SeqNumber>相同主键的消息提交时，Broker只会持久化一条。其中PID是Kafka每次重启都会分配一个新的；Partition 表示分区号；Sequence Number是单调自增的。

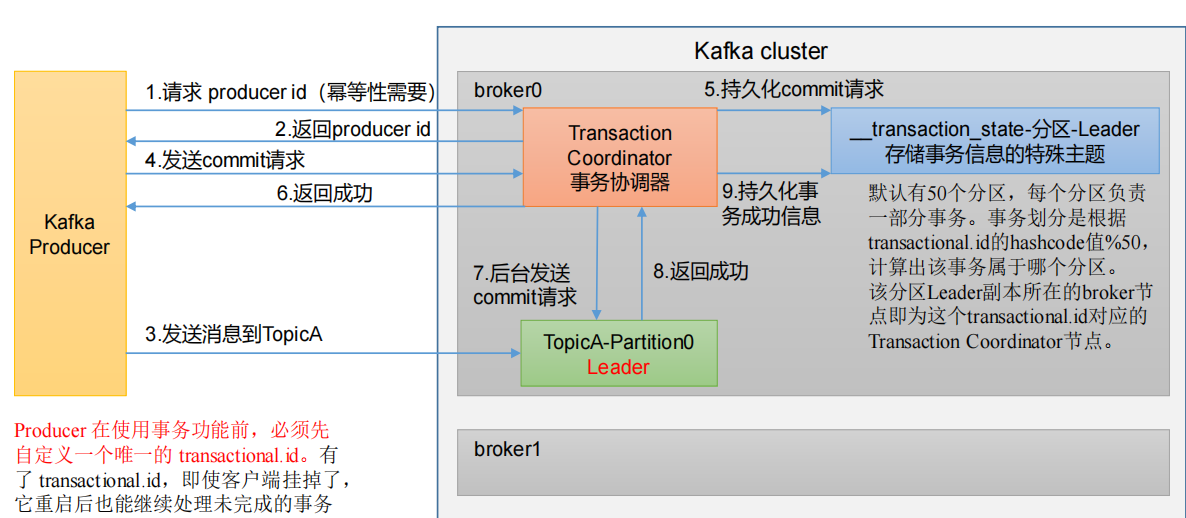
所以幂等性只能保证的是在单分区单会话内不重复。



开启幂等性参数 enable.idempotence 默认为 true，false 关闭幂等性。如果想要在整个集群中保证数据不重复，则还需要依赖于事务。

#### 4.7.2.2、事务

开启事务的前提是必须开启幂等性。



事务的实现主要是依赖于事务协调器（Transaction Coordinator），但是在kafka中每一个broker都有事务协调器，所以应该使用哪一个事务协调器？

首先Producer 在使用事务功能前，必须先自定义一个唯一的 transactional.id，事务id不但可以判断出要使用哪一个事务协调器，还能够保证在在客户端挂掉，重新启动之后也能继续处理未完成的事务。

在每一个broker中都存在一个存储事务信息的特殊主题(\_\_transaction\_state-分区-Leader)，在这个Topic中默认有50个分区，每个分区负责一部分事务，事务划分是根据设置的transactional-id的哈希值计算对\_transaction\_state分区数（默认值是50）取余运算，找到分区编号，该分区对应的leader副本所对应的broker的即为Tranaction Coordinator所在节点。

（1）Producer向任意一个brokers发送获取事务协调器的请求（FindCoordinatorRequest）来获取Transaction Coordinator的地址。

（2）producer向事务协调器发起获取PID（幂等性需要）的请求，返回PID

（3）然后producer发送消息到topic分区中,并且同时会向事务协调器发起持久化commit请求，事务协调器接收到持久化commit请求后，会将这个请求保存到存放事务的topic中去。

（4）接着事务协调器会向生产者返回成功的信息，并且会向topic分区中发送commit请求，判断topic分区中的消息是否持久化成功，如果处理完毕，则向事务协调器返回成功信息，最后事务协调器会向存储事务的topic主题中进行事务成功的记录。

## 4.8、生产经验——数据有序

## 4.9、生产经验——数据乱序

# Kafka Broker

## 5.1、Kafka Broker 工作流程

### 5.1.1、Zookeeper 存储的 Kafka 信息

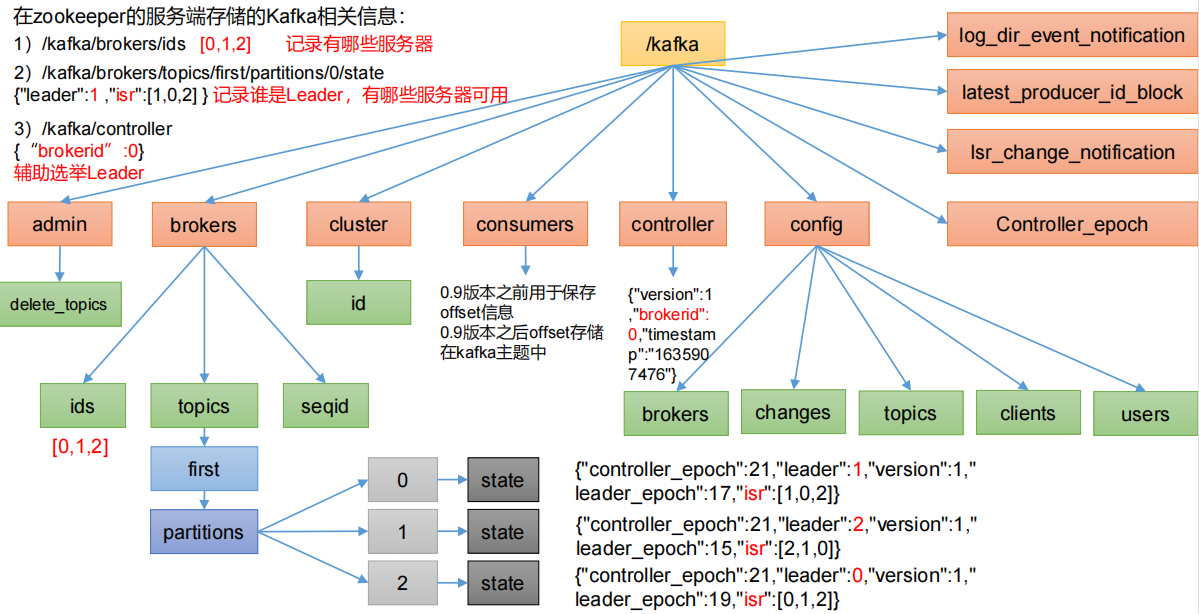
（1）启动 Zookeeper 客户端。

[atguigu@hadoop102 zookeeper-3.5.7]$ bin/zkCli.sh

（2）通过 ls 命令可以查看 kafka 相关信息

[zk: localhost:2181(CONNECTED) 2] ls /kafka

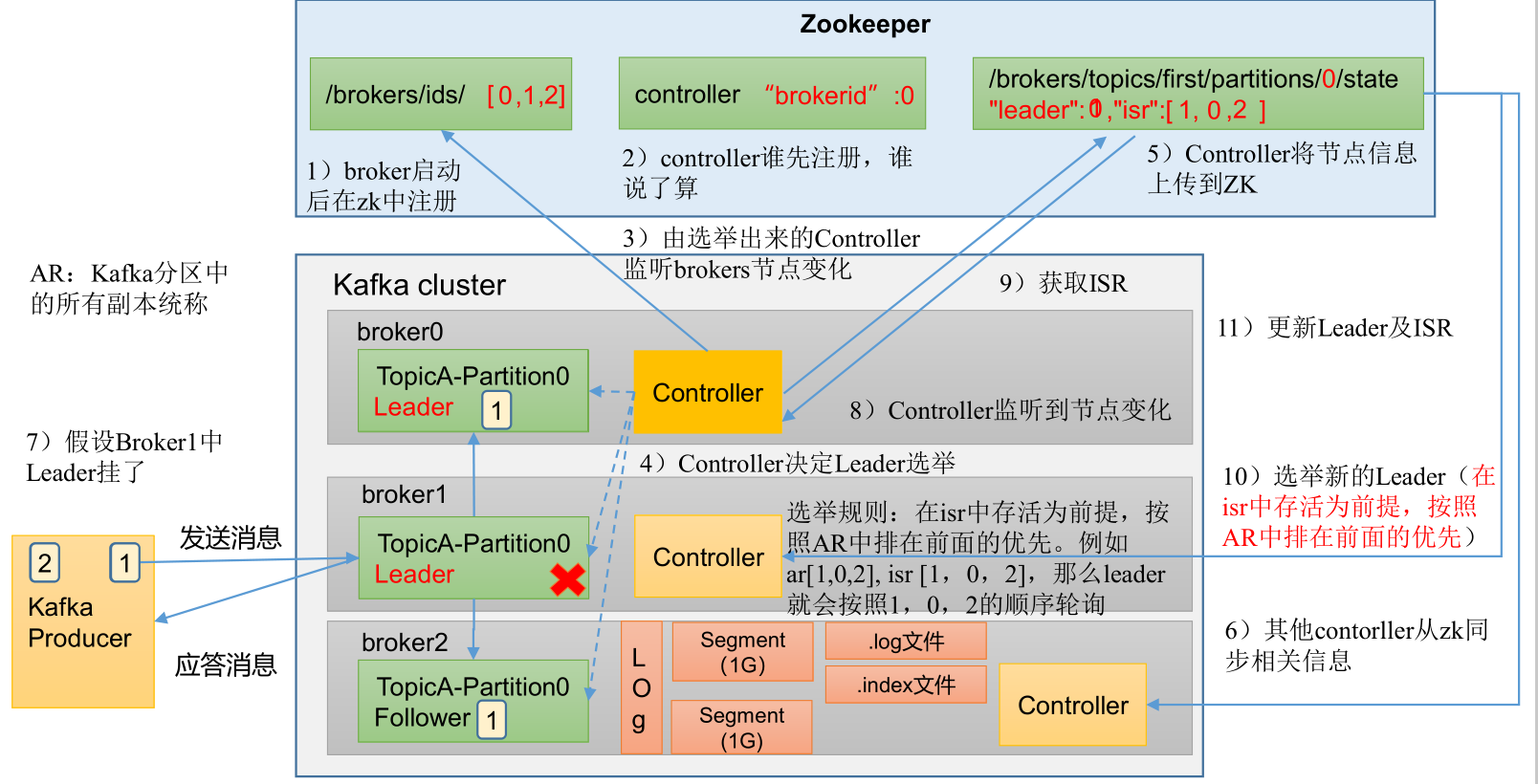
### 5.1.2、服务端存储的相关信息



主要有以下几个比较主要的

* Brokers
* Ids:主要存储的是brokerId
* Topics:记录的是所有的主题中的Leader信息以及ISR信息， topic(主题) --- partitions（分区）--- 分区号 --- state ---> 具体的Leader信息和ISR信息。
* Consumers
* 0.9版本之前用于保存offset信息，0.9版本之后offset存放的位置在kafka的主题中（topic），放到topic的原因，是因为每一个消费者消费信息都会更新offset，造成了kafka与zk进行频繁通信，效率较慢。
* Controller
* 辅助选举leader，在broker节点上都有一个controller模块，broker节点上的controller，才是真正决定谁是Leader。每个节点上都有Controller，那个节点说了算？这时候zookeeper上的Controller模块的用途就展现出来了，所有broker节点上的controller都会去抢先去注册zookeeper上的Controller，谁先抢占到注册权，谁才是负责Leader的选举的controller。

### 5.1.3、broker的工作流程



Broker启动之后，都会在zookeeper中进行注册，然后每一个broker节点上的controller就会去抢占zookeeper中的controller节点，谁抢先注册到controller节点，谁才是负责分区副本Leader的选举的controller，当某一个controller抢占到注册权后，会先监听zookeeper节点下的brokers(下面的ids)里面的节点变化（当节点发生变化的时候，会第一时间知道），然后再进行Leader的真正选举，Leader的选举策略首先是以在ISR中存活为前提，按照AR（所有的副本分区的统称）中排列在前面的顺序进行优先选举。例如ar[1,0,2], isr [1，0，2]，那么leader就会按照1，0，2的顺序轮询，AR的顺序是固定的，应该是在注册的时候确定的，选举出来topic分区副本的leader之后，会将Leader信息保存到zookeeper中topics节点下具体的topic节点中，其他broker节点中的controller会从zookeeper中同步相关的leader信息，其他的controller同步leader数据的目的是当已经抢占到注册权的controller宕机之后，能够更快的替代原先的controller，当选举出来的副本leader宕机之后，controller监控到brokers的节点的变化，会重新从zookeeper中拉取回leader信息和ISR信息，根据选举策略重新进行选举，选举完新的leader之后，会将新的Leader信息以及ISR信息更新到zookeeper中的topics节点下的具体topic节点中，其他的controller也会再次重新同步相关的数据。

### 5.1.4、服役新节点/退役旧节点

* 创建一个要均衡的主题。
* 生成一个负载均衡的计划。
* 创建副本存储计划。
* 执行副本存储计划。
* 验证副本存储计划。

## 5.3、Kafka 副本

### 5.3.1、副本基本信息

（1）Kafka 副本作用：提高数据可靠性。

（2）Kafka 默认副本 1 个，生产环境一般配置为 2 个，保证数据可靠性；太多副本会增加磁盘存储空间，增加网络上数据传输，降低效率。

（3）Kafka 中副本分为：Leader 和 Follower。Kafka 生产者只会把数据发往 Leader，然后 Follower 找 Leader 进行同步数据。

（4）Kafka 分区中的所有副本统称为 AR（Assigned Repllicas）。AR = ISR + OSR

ISR，表示和 Leader 保持同步的 Follower 集合。如果 Follower 长时间未向 Leader 发送通信请求或同步数据，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 30s。Leader 发生故障之后，就会从 ISR中选举新的 Leader。OSR ，表示 Follower 与 Leader副本同步时，延迟过多的副本。

### 5.3.2、Leader及Follower数据同步问题

#### 5.3.2.1、Follower出现故障



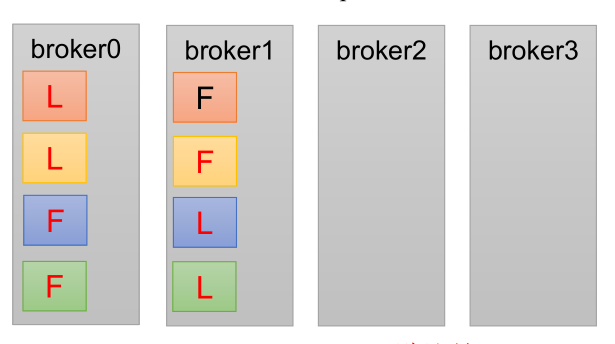
当Follower发生故障后会被临时踢出ISR，这个期间没有出现故障的Leader和Follower继续接收数据，等待该故障的Follower恢复正常之后，Follower会读取本地磁盘记录的上次的HW（高水位线），并且将Log文件中高于HW的数据给截掉，从HW的位置开始向Leader进行数据同步，等到该Follower 的LEO （副本的最后一个offset+1）大于等于该Partition 的HW的时候，即Follower追上Leader之后，就可以重新加入ISR了。

#### 5.3.2.2、Leader出现故障



当Leader发生故障之后，会从ISR中选出一个新的Leader，选出新的Leader后，会与其他分区进行比较，获取当前分区的HW，如果Follower中的数据高于Leader中的数据，为保证多个副本之间的数据一致性，所有的Follower会将各自的log文件中高于HW的部分数据进行截掉，然后从新的Leader中进行数据同步。

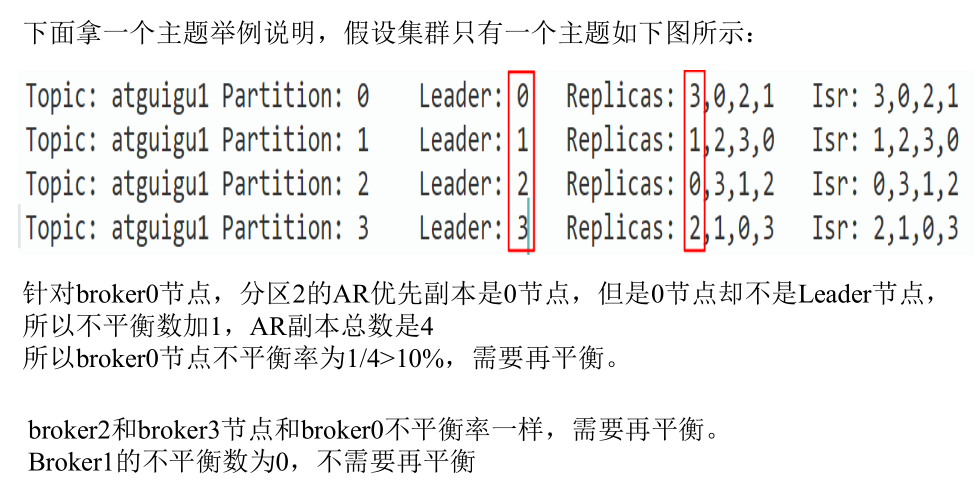
### 5.5.3、Leader Partition自动平衡



正常情况下，Kafka本身会自动把Leader Partition均匀分散在各个机器上，来保证每台机器的读写吞吐量都是均匀的。但是如果某些broker宕机，会导致Leader Partition过于集中在其他少部分几台broker上，这会导致少数几台broker的读写请求压力过高，其他宕机的broker重启之后都是follower partition，读写请求很低，造成集群负载不均衡。

解决方案，使用leader的再平衡，默认情况下再平衡机制是打开的（auto.leader.rebalance.enable，默认是true），每个broker允许的不平衡的leader的比率默认是10%，如果某个broker超过了这个值，控制器会触发leader的平衡。Kafka中检查leader负载是否平衡的间隔时间为5分钟（默认值300秒）。

计算不平衡的比例方式：

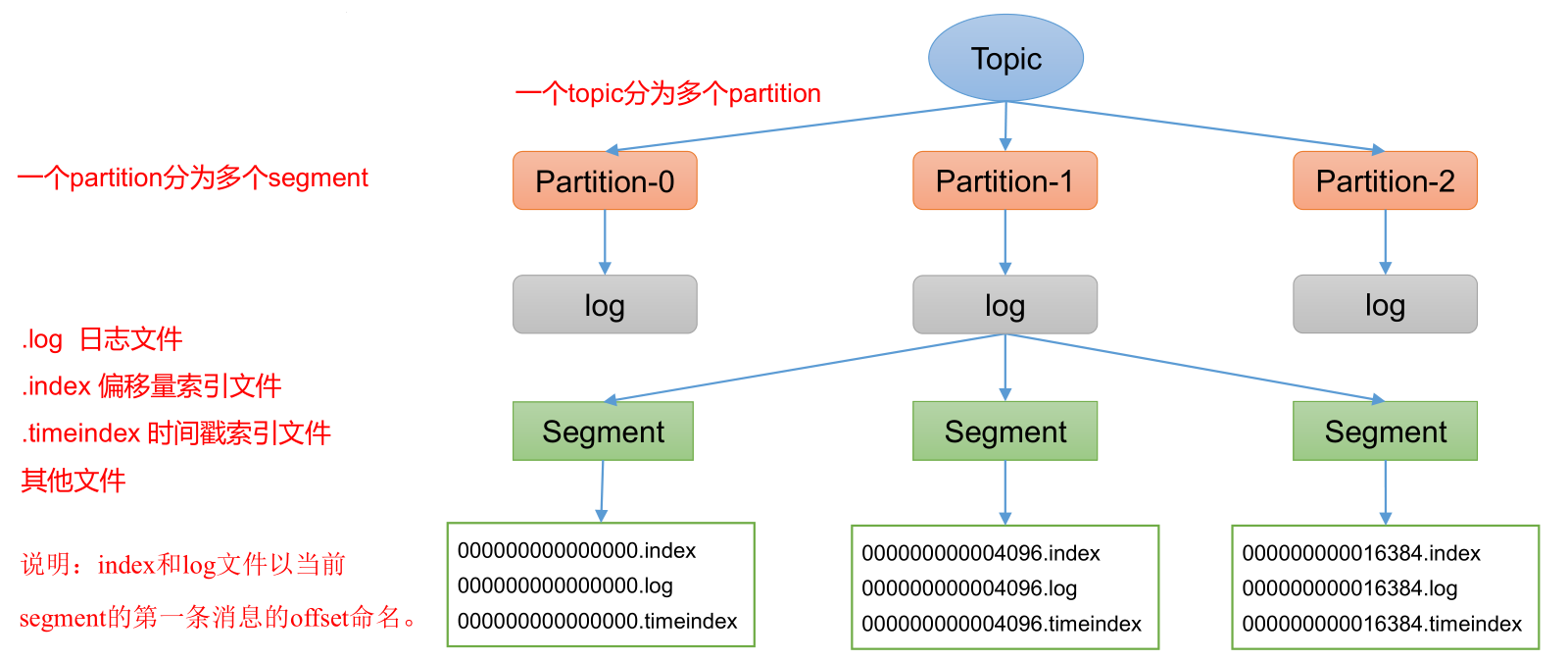


分区2的AR优先副本是0节点，但是0节点却不是Leader节点，说明发生了宕机。

但是一般情况下再平衡不建议开启，或者选择开启的话，再平衡比例可以选择大一些，因为触发再平衡机制后，在整个平衡的过程中是不能够处理数据的，所以再平衡会影响效率。

## 5.4、文件存储

### 5.4.1、Topic 数据的存储机制

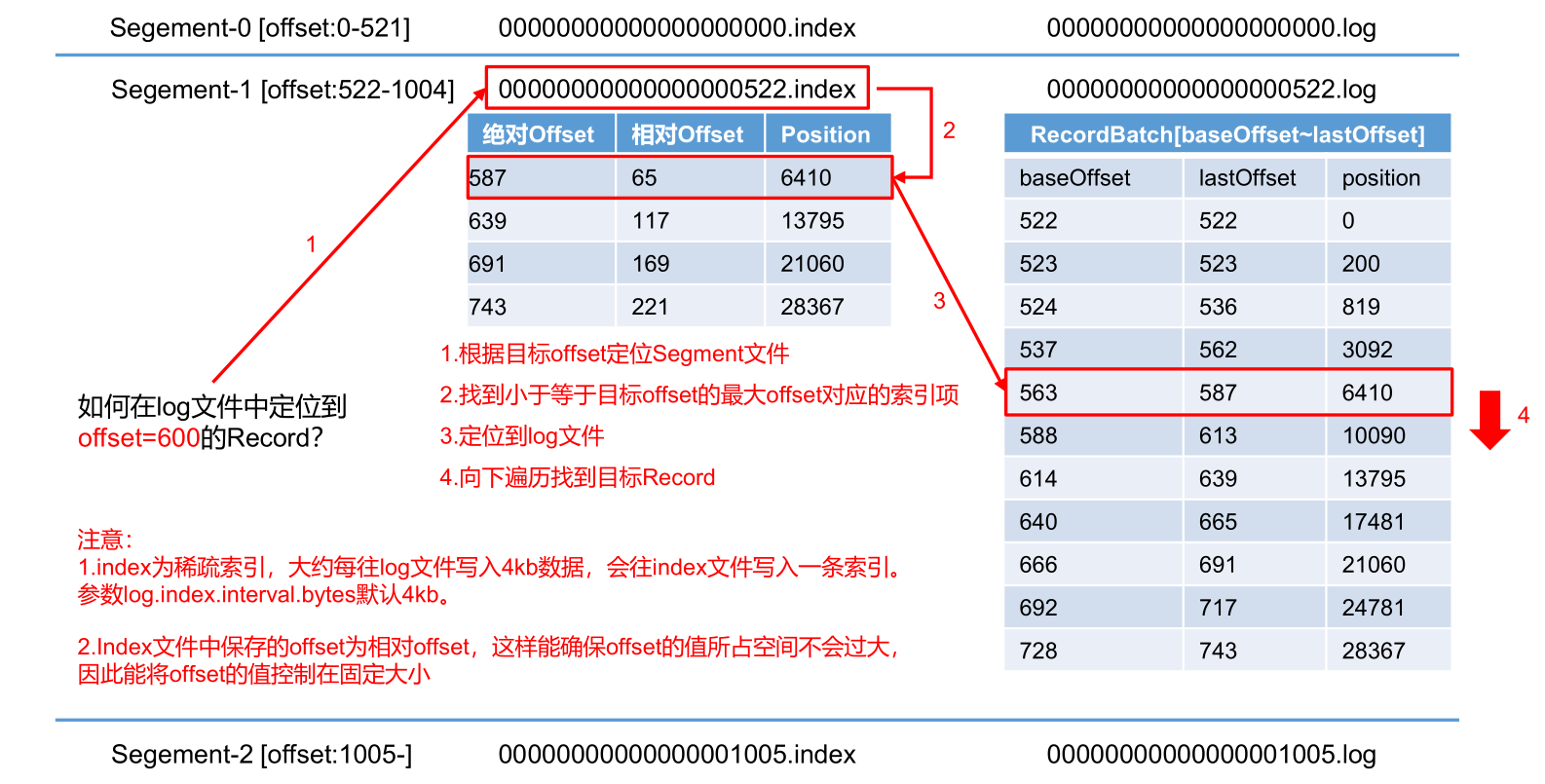


Topic是逻辑上的概念，而partition是物理上的概念，每个partition对应于一个log文件，该log文件中存储的就是Producer生产的数据。

Producer生产的数据会被不断追加到该log文件末端，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包括：“.index”文件、“.log”文件和.timeindex等文件。

“.log”文件存放的是数据，“.index”存放的时候偏移量索引文件，“.timeindex”文件存放的是时间戳索引文件，而且“.log”和“.index”文件的命名规则是以当前segment的第一条消息的offset索引进行命名的，每个segment的默认大小是1G，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，例如：/opt/mode/kafka/data/first-0。

### 5.4.2、Log文件和Index文件详解



* .index为稀疏索引，大约每往log文件写入4kb数据，会往index文件写入一条索引。参数log.index.interval.bytes默认4kb。
* .Index文件中保存的offset为相对offset，这样能确保offset的值所占空间不会过大，因此能将offset的值控制在固定大小

**如何在log文件中定位到offset=600的Record？**

首先会根据offset去比较索引文件以及“.log”文件的文件名，定位到是哪一个segment文件，然后根据给定的offset，与（segment的起始offset+相对offset）然后进行比较，找到小于等于目标offset的最大offset对应的索引项的position的值，根据找到的position的值，去该segment下的“.log”文件中找对应的position位置，然后向下遍历寻找，就能找到目标offset=600的记录数据。

### 5.4.3、文件清理策略

Kafka 中默认的日志保存时间为 7 天，可以通过调整如下参数修改保存时间。

* log.retention.hours，最低优先级小时，默认 7天。
* log.retention.minutes，分钟。
* log.retention.ms，最高优先级毫秒。
* log.retention.check.interval.ms，负责设置检查周期，默认 5 分钟。

那么日志一旦超过了设置的时间，怎么处理呢？

在Kafka 中提供的日志清理策略有 delete 和 compact两种。

#### 5.4.3.1、delete日志删除：将过期数据删除

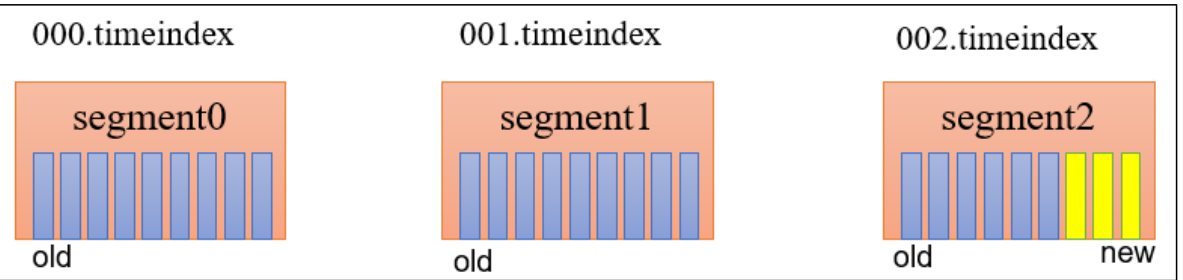
配置文件清理策略为日志删除，log.cleanup.policy = delete 所有数据启用删除策略。有删除的时候有两种方式，基于时间以及基于大小。

（1）基于时间：默认打开。以 segment 中所有记录中的最大时间戳作为该文件时间戳。

（2）基于大小：默认关闭。超过设置的所有日志总大小，删除最早的 segment。

log.retention.bytes，默认等于-1，表示无穷大。

如果一个 segment中有一部分数据过期，一部分没有过期，怎么处理？



当segment文件中有一部分数据过期，一部分没有过期的时候，会以该文件中的最大时间戳作为过期时间，如果最大过期时间没有达到要清理的时间的话则不用清理，等待下次检查过期文件的来进行清理。

#### 5.4.3.2、compact日志压缩

## 5.5、高效读写数据

## 5.6、Kafka Broker 工作流程

# 6、Kafka 消费者

## 6.1、Kafka 消费方式

## 6.2、Kafka 消费者工作流程

## 6.3、消费者 API

## 6.4、生产经验——分区的分配以及再平衡

## 6.5、offset 位移

## 6.6、生产经验——消费者事务

## 6.7、生产经验——数据积压（消费者如何提高吞吐量）

## 6.8、生产经验——数据积压（消费者如何提高吞吐量）

# 7、Kafka-Eagle 监控

# 8、Kafka-Kraft 模式

## 8.1、Kafka-Kraft 架构

## 8.2、Kafka-Kraft 集群部署

## 8.3、Kafka-Kraft 集群启动停止脚本

# 9、集成 SpringBoot

# 10、调优-Kafka 硬件配置选择

## 10.1、场景说明

## 10.2、服务器台数选择

## 10.3、磁盘选择

## 10.4、内存选择

## 10.5、CPU 选择

## 10.6、网络选择

# 11、调优- Kafka 生产者

## 11.1、Kafka 生产者核心参数配置

## 11.2、生产者如何提高吞吐量

## 11.3、数据可靠性

## 11.4、数据去重

## 11.5、数据有序

## 11.6、数据乱序

# 12、调优-Kafka Broker

## 12.1、Broker 核心参数配置

## 12.2、服役新节点/退役旧节点

## 12.3、增加分区

## 12.4、增加副本因子

## 12.5、手动调整分区副本存储

## 12.6、Leader Partition 负载平衡

## 12.7、自动创建主题

## 12.8、Leader Partition 负载平衡

# 13、调优-Kafka 消费者

## 13.1、Kafka 消费者核心参数配置

## 13.2、消费者再平衡

## 13.3、指定 Offset 消费

## 13.4、指定时间消费

## 13.5、消费者事务

## 13.6、消费者如何提高吞吐量

# 14、调优-Kafka 总体

## 14.1、如何提升吞吐量

## 14.2、数据精准一次

## 14.3、合理设置分区数

## 14.4、单条日志大于 1m

## 14.5、服务器挂了

## 14.6、集群压力测试

# 15、源码-Kafka 生产者源码

# 16、源码-Kafka 消费者源码

# 17、源码-Kafka 服务器源码

# 18、面试题