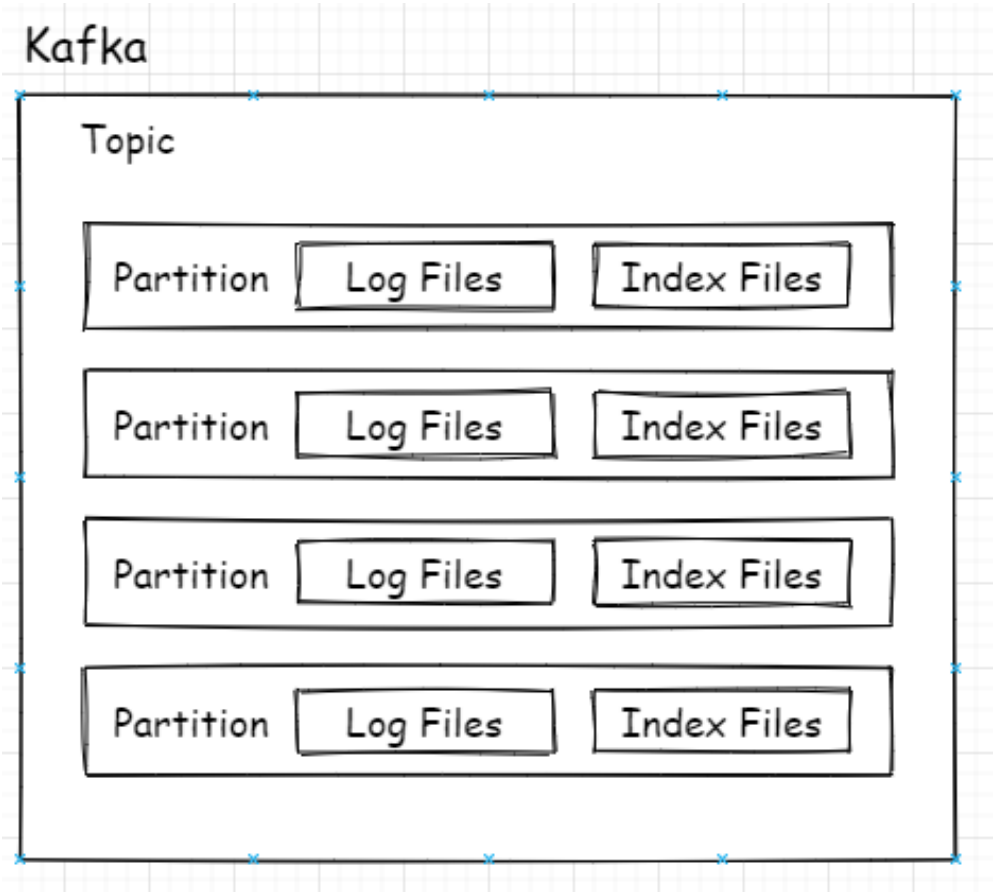
Kafka 知识点总结

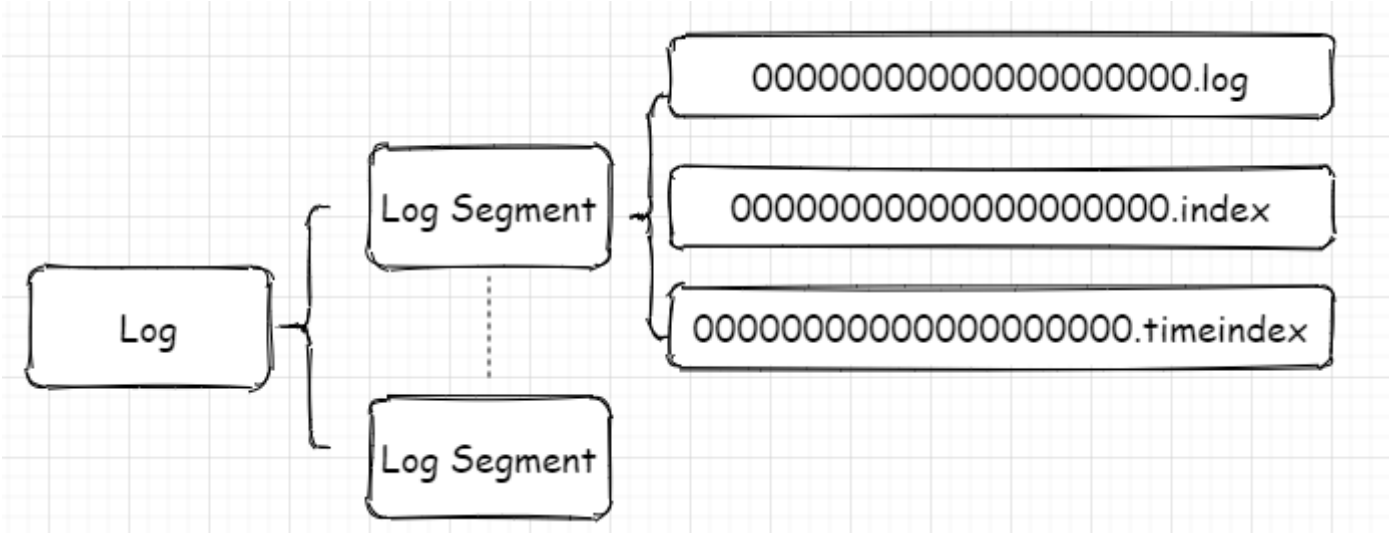
Topic

Topic 相当于RabbitMQ的队列名，也可以理解为数据库表的概念



Kafka的Topic可以有多个分区，分区其实就是最小的读取和存储结构，即Consumer看似订阅的是Topic，实则是从Topic下的某个分区获得消息，Producer也是发送消息也是如此。

上图是总体逻辑上的关系，映射到实际代码中在磁盘上的关系则是如下图所示：



每个分区对应一个Log对象，在磁盘中就是一个子目录，子目录下面会有多组日志段即多Log Segment，每组日志段包含：消息日志文件(以log结尾)、位移索引文件(以index结尾)、时间戳索引文件(以timeindex结尾)。其实还有其它后缀的文件，例如.txnindex、.deleted等等。篇幅有限，暂不提起。

以下为日志的定义

```
@threadsafe
class Log(@volatile private var _dir: File,  //分区所在目录
          @volatile var config: LogConfig,   //配置
          @volatile var logStartOffset: Long,//暴露给客户端的起始offset
          @volatile var recoveryPoint: Long, //恢复点
          scheduler: Scheduler,              //用于后台操作的线程池调度
          brokerTopicStats: BrokerTopicStats,
          val time: Time,
          val maxProducerIdExpirationMs: Int,
          val producerIdExpirationCheckIntervalMs: Int,
          val topicPartition: TopicPartition,
          val producerStateManager: ProducerStateManager,
          logDirFailureChannel: LogDirFailureChannel) extends Logging with KafkaMetricsGroup {
    ...
     /* the actual segments of the log ,日志段们, */
private val segments: ConcurrentNavigableMap[java.lang.Long, LogSegment] = new
ConcurrentSkipListMap[java.lang.Long, LogSegment]
    ...
                                                                                }
```

日志段定义

```
@nonthreadsafe
class LogSegment private[log] (val log: FileRecords, //实际保存消息的对象
                               val lazyOffsetIndex: LazyIndex[OffsetIndex], //位移索引
                               val lazyTimeIndex: LazyIndex[TimeIndex], //时间戳索引
                               val txnIndex: TransactionIndex, //已中止事务索引
                               val baseOffset: Long, // 起始位移
                               val indexIntervalBytes: Int, //多少字节插一个索引
                               val rollJitterMs: Long, // 日志段新增扰动值
                               val time: Time) extends Logging {
```

`indexIntervalBytes`可以理解为插了多少消息之后再建一个索引，由此可以看出**Kafka的索引其实是稀疏索引**，这样可以避免索引文件占用过多的内存，从而可以**在内存中保存更多的索引**。对应的就是Broker 端参数`log.index.interval.bytes` 值，默认4KB。

AAA 实际的**通过索引**查找消息过程是先通过offset找到索引所在的文件，然后**通过二分法**找到离目标最近的索引，再顺序遍历消息文件找到目标文件。这波操作时间复杂度为`O(log2n)+O(m)`,n是索引文件里索引的个数，m为稀疏程度。

再说下`rollJitterMs`,这其实是个扰动值，对应的参数是`log.roll.jitter.ms`,这其实就要说到日志段的切分了，`log.segment.bytes`,这个参数控制着日志段文件的大小，默认是1G，即当文件存储超过1G之后就新起一个文件写入。这是以大小为维度的，还有一个参数是`log.segment.ms`,以时间为维度切分。

那配置了这个参数之后如果有很多很多分区，然后因为这个参数是全局的，因此同一时刻需要做很多文件的切分，这磁盘IO就顶不住了啊，因此需要设置个`rollJitterMs`，来岔开它们。

日志段的写入

```
  @nonthreadsafe
  def append(largestOffset: Long, //这批消息里面最大的位移值
             largestTimestamp: Long, //这批消息里面最大的时间戳
             shallowOffsetOfMaxTimestamp: Long, //最大时间戳对应的位移值
             records: MemoryRecords): Unit = { //消息们
    if (records.sizeInBytes > 0) {
      trace(s"Inserting ${records.sizeInBytes} bytes at end offset $largestOffset at position
${log.sizeInBytes} " +
            s"with largest timestamp $largestTimestamp at shallow offset $shallowOffsetOfMaxTimestamp")
      val physicalPosition = log.sizeInBytes() //获取当前日志的位移
      if (physicalPosition == 0)               //说明当前日志为空，则记录时间戳作为切分的依据
        rollingBasedTimestamp = Some(largestTimestamp)

      ensureOffsetInRange(largestOffset)       //确保位移值合法

      // append the messages
      val appendedBytes = log.append(records)
      trace(s"Appended $appendedBytes to ${log.file} at end offset $largestOffset")
      // Update the in memory max timestamp and corresponding offset.
      if (largestTimestamp > maxTimestampSoFar) {  //更新最大时间戳和其对应的位移值
        maxTimestampSoFar = largestTimestamp
        offsetOfMaxTimestampSoFar = shallowOffsetOfMaxTimestamp
      }
      // append an entry to the index (if needed)
      if (bytesSinceLastIndexEntry > indexIntervalBytes) {
        offsetIndex.append(largestOffset, physicalPosition)
        timeIndex.maybeAppend(maxTimestampSoFar, offsetOfMaxTimestampSoFar)
        bytesSinceLastIndexEntry = 0
      }
      bytesSinceLastIndexEntry += records.sizeInBytes
    }
  }
```

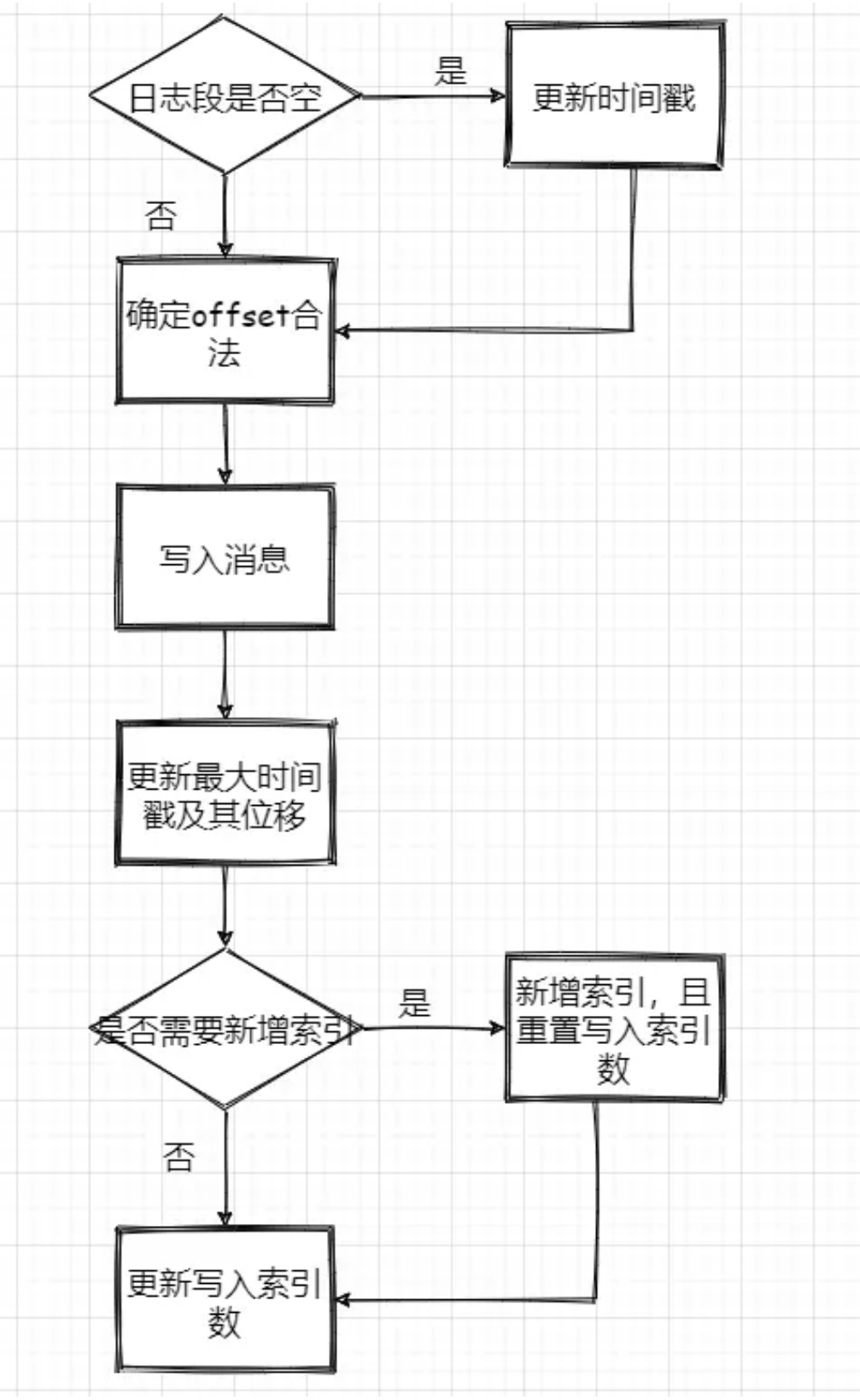
1、判断下当前日志段是否为空，空的话记录下时间，来作为之后日志段的切分依据

2、确保位移值合法，最终调用的是`AbstractIndex.toRelative(..)`方法，即使判断offset是否小于0，是否大于int最大值。

3、append消息，实际上就是通过`FileChannel`将消息写入，当然只是写入内存中及页缓存，是否刷盘看配置。

4、更新日志段最大时间戳和最大时间戳对应的位移值。这个时间戳其实用来作为定期删除日志的依据

5、更新索引项，如果需要的话`(bytesSinceLastIndexEntry > indexIntervalBytes)`



日志段的读取

```
  @threadsafe
  def read(startOffset: Long,  //读取的第一条消息的位移
           maxSize: Int,       //能读取最大字节数
           maxPosition: Long = size,  //最大能读到的文件位置
           minOneMessage: Boolean = false): FetchDataInfo = { //是否至少返回一条
    if (maxSize < 0)
      throw new IllegalArgumentException(s"Invalid max size $maxSize for log read from segment $log")

    val startOffsetAndSize = translateOffset(startOffset)  //根据位移找到消息物理位置和大小

    // if the start position is already off the end of the log, return null
    if (startOffsetAndSize == null)
      return null

    val startPosition = startOffsetAndSize.position
    val offsetMetadata = LogOffsetMetadata(startOffset, this.baseOffset, startPosition)

    val adjustedMaxSize = //如果minOneMessage为true则调整下maxSize为一条消息的size
      if (minOneMessage) math.max(maxSize, startOffsetAndSize.size)
      else maxSize

    // return a log segment but with zero size in the case below
    if (adjustedMaxSize == 0)
      return FetchDataInfo(offsetMetadata, MemoryRecords.EMPTY)

    // calculate the length of the message set to read based on whether or not they gave us a maxOffset
再算下最小能获取的size
    val fetchSize: Int = min((maxPosition - startPosition).toInt, adjustedMaxSize)

      //从指定位置获取指定大小的集合
    FetchDataInfo(offsetMetadata, log.slice(startPosition, fetchSize),
      firstEntryIncomplete = adjustedMaxSize < startOffsetAndSize.size)
  }
```

1、根据第一条消息的offset，通过`OffsetIndex`找到对应的消息所在的物理位置和大小。

2、获取`LogOffsetMetadata`,元数据包含消息的offset、消息所在segment的起始offset和物理位置

3、判断`minOneMessage`是否为`true`,若是则调整为必定返回一条消息大小，其实就是在单条消息大于`maxSize`的情况下得以返回，防止消费者饿死

4、再计算最大的`fetchSize`,即（最大物理位移-此消息起始物理位移）和`adjustedMaxSize`的最小值(这波我不是很懂，因为以上一波操作`adjustedMaxSize`已经最小为一条消息的大小了)

5、调用 `FileRecords` 的 `slice` 方法从指定位置读取指定大小的消息集合，并且构造`FetchDataInfo`返回

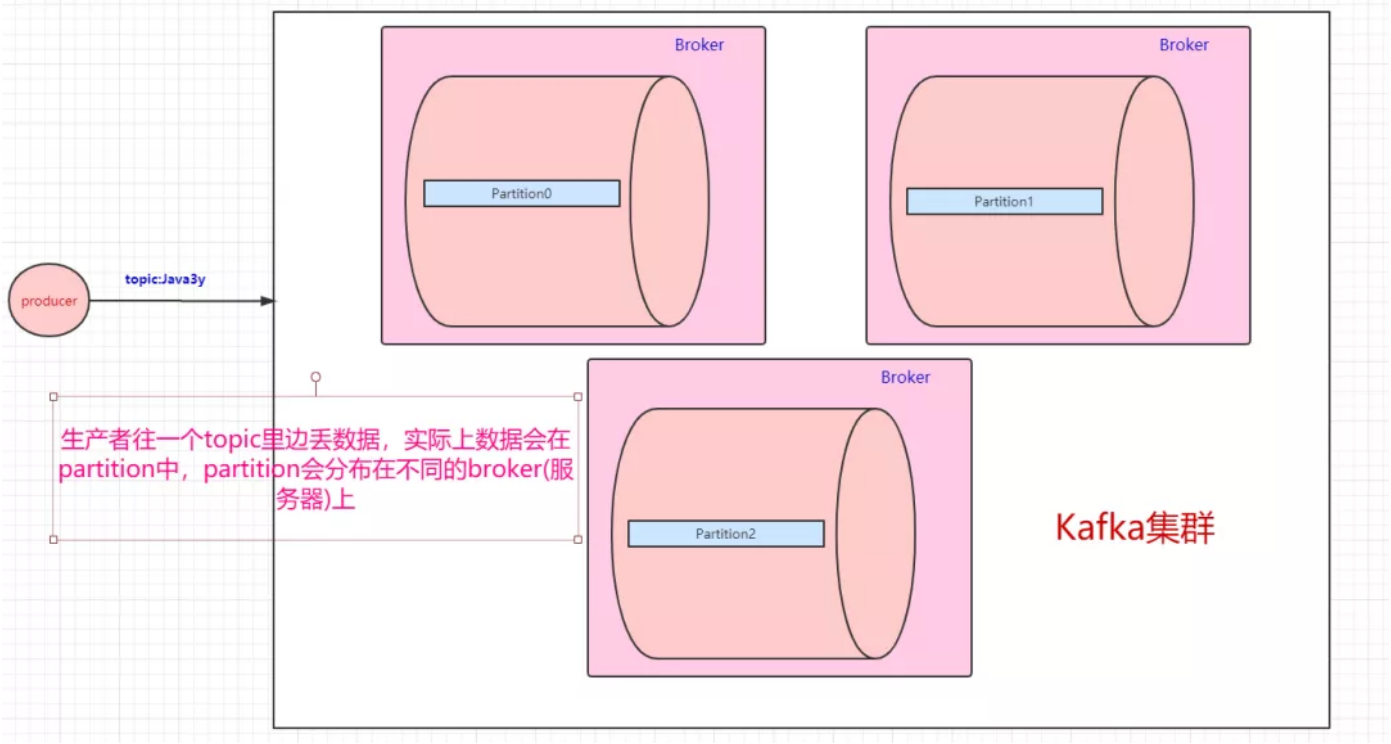
Partition

为了提高一个topic的吞吐量，Kafka会把tipic进行分区（Partition）

Broker

一台Kafka服务器叫做Broker，Kafka集群就是多台Kafka服务器。

一个topic会分为多个partition，实际上partition会**分布**在不同的broker中。



由此得知：**Kafka是天然分布式的。**

Kafka如何实现高可用

将数据存到不同的partition上，kafka会把这个partition做备份。比如现在我们有三个partition，分别存在3台broker上，每个partioion都会备份，这些备份散落在不同的broker上。生产者往topic丢数据，是与**主**分区交互，消费者消费topic的数据，也是与主分区交互。

备份分区仅仅用作于备份，不做读写。如果某个Broker挂了，那就会选举出其他Broker的partition来作为主分区，这就实现了**高可用**。

当生产者把数据丢进topic时，partition是怎么将其持久化的呢?

（不持久化如果Broker中途挂了，那肯定会丢数据嘛)。

Kafka是将partition的数据写在磁盘的（消息日志），不过Kafka只允许追加写入（顺序访问），避免了缓慢的随机I/O操作。

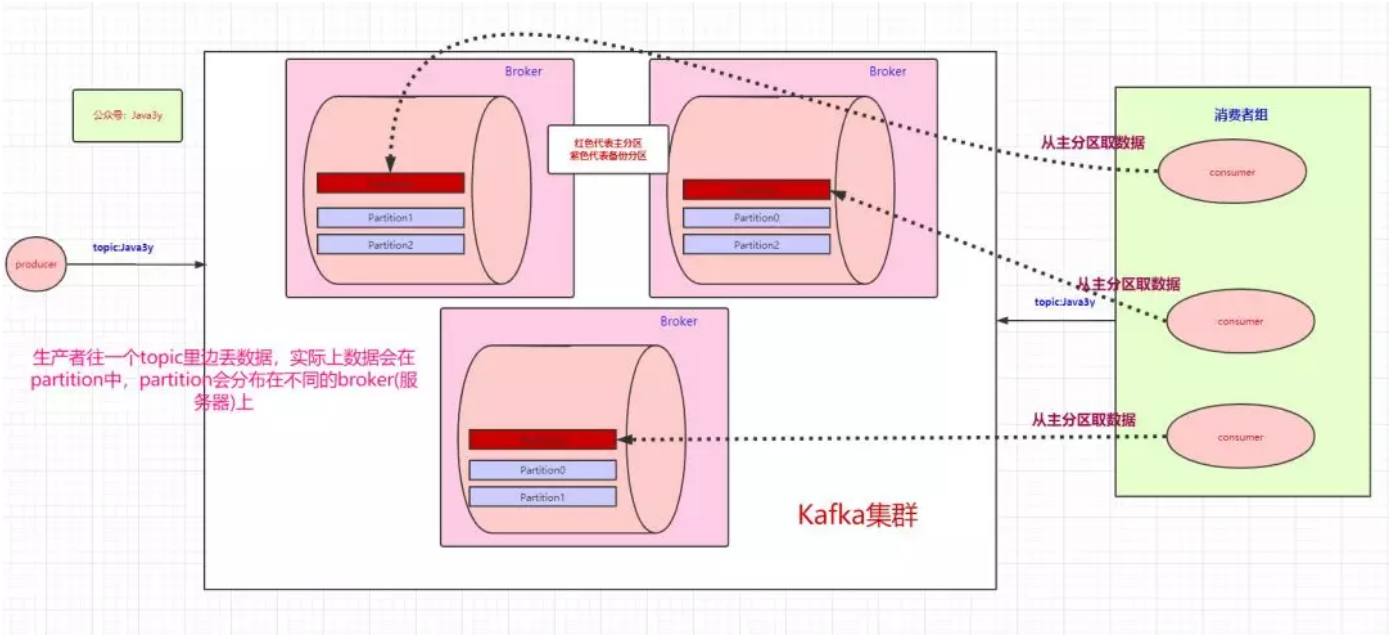
- Kafka也不是partition一有数据就立马将数据写到磁盘上，它会先**缓存**一部分，等到足够多数据量或等待一定的时间再批量写入(flush)。

生产者消费者--0

生产者可以有多个，消费者也可以有多个。像上面图的情况，是一个消费者消费三个分区的数据。多个消费者可以组成一个**消费者组**。本来是一个消费者消费三个分区的，现在我们有消费者组，就可以**每个消费者去消费一个分区**（也是为了提高吞吐量）

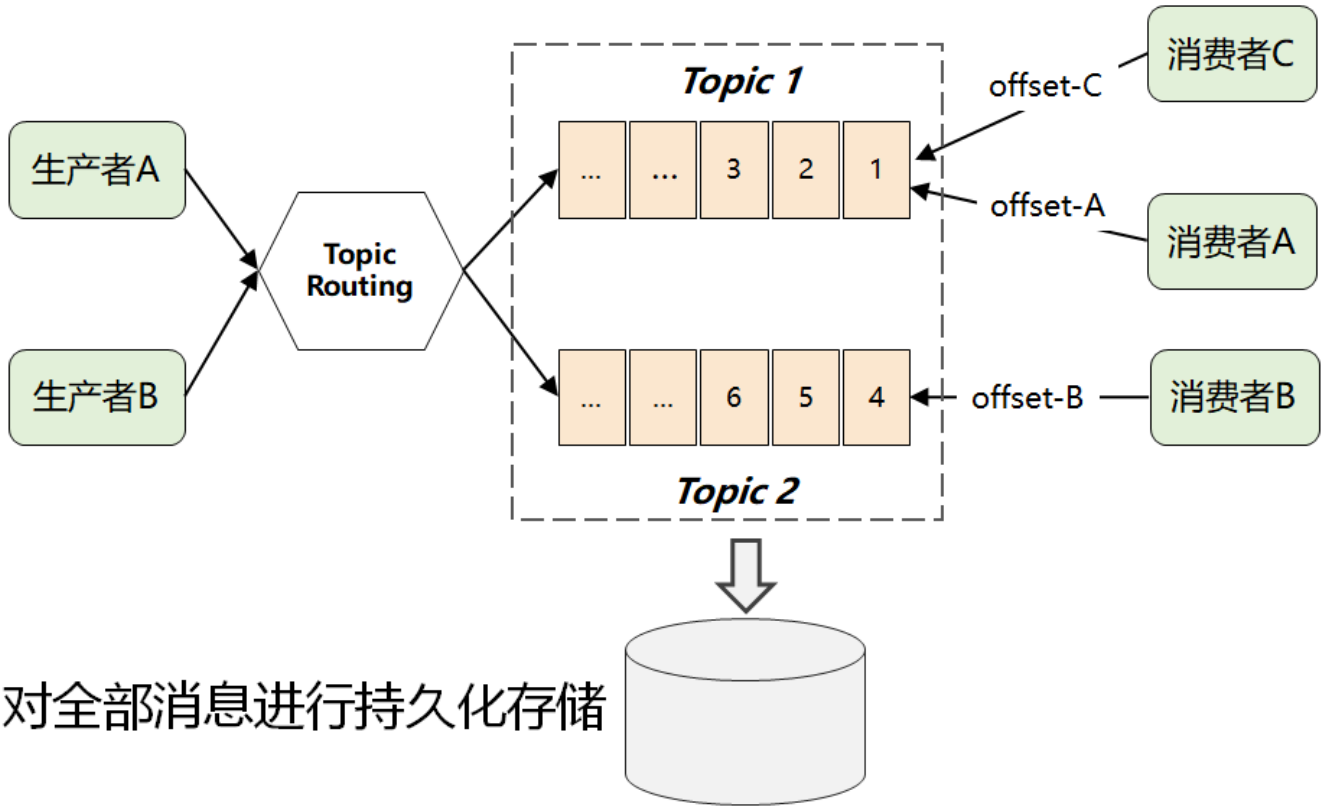
- 如果消费者组中的某个消费者挂了，那么其中一个消费者可能就要消费两个partition了

- 如果只有三个partition，而消费者组有4个消费者，那么一个消费者会空闲
- 如果多加入一个**消费者组**，无论是新增的消费者组还是原本的消费者组，都能消费topic的全部数据。（消费者组之间从逻辑上它们是**独立**的）



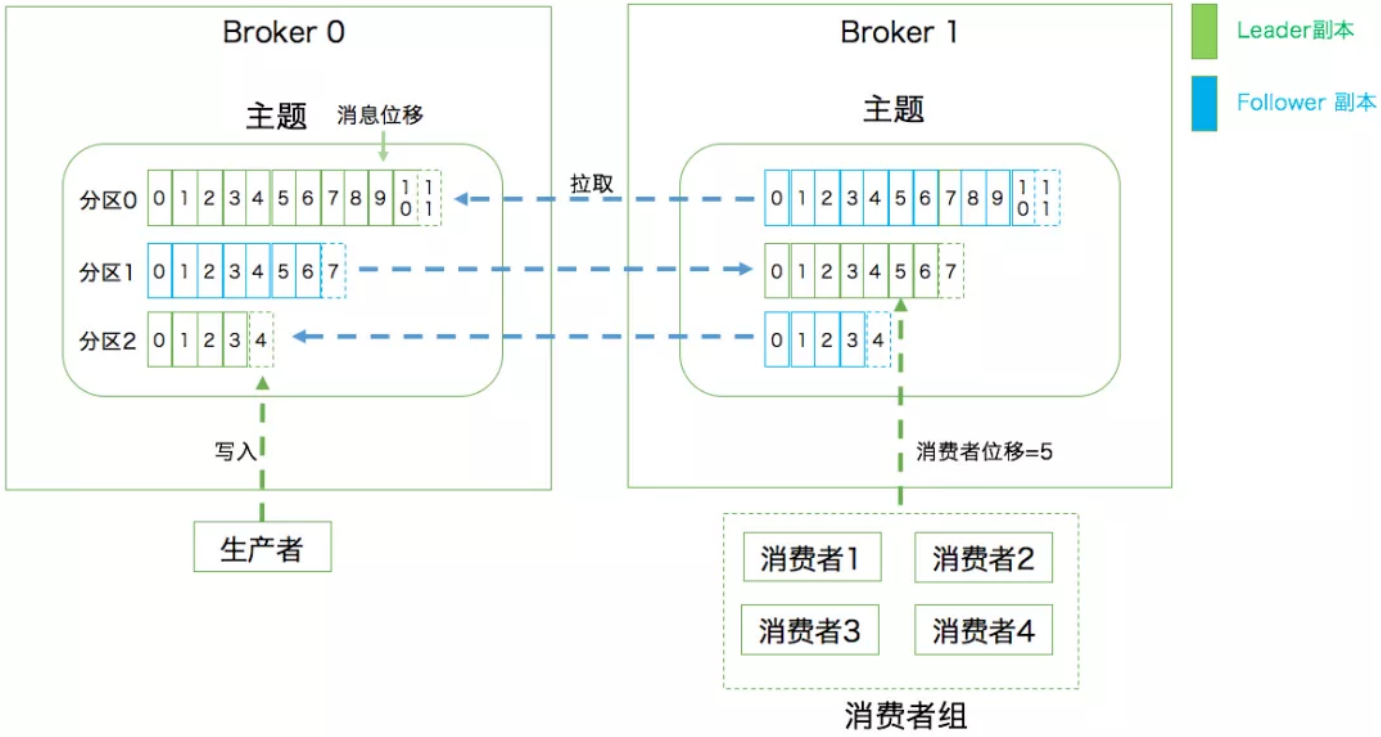
Offset

offset 表示消费者的消费进度。每次消费者消费的时候，都会提交这个`offset`，Kafka可以让你选择是自动提交还是手动提交。



Zookeeper在Kafka的作用

1. 探测broker和consumer的添加和移除
2. 维护partition的Master/Slaver 的关系，如果主分区挂了，需要选出备份分区作为主分区
3. 维护topic、partition等元配置信息



数据写到消息队列，可能会存在数据丢失问题，数据在消息队列需要持久化

Kafka会将partition以消息日志的方式(落磁盘)存储起来，通过 顺序访问IO和缓存(等到一定的量或时间)才真正把数据写到磁盘上，来提高速度。

为什么会出现重复消费?

凡是分布式就无法避免网络抖动/机器宕机等问题的发生，很有可能消费者A读取了数据，还没来得及消费，就挂掉了。Zookeeper发现消费者A挂了，让消费者B去消费原本消费者A的分区，等消费者A重连的时候，发现已经重复消费同一条数据了。(各种各样的情况，消费者超时等等都有可能...) 如果业务上不允许重复消费的问题，最好消费者那端做业务上的校验（如果已经消费过了，就不消费了）

重复消费如何避免

Kafka如何保证生产可靠性和消费可靠性

Kafka为什么比其他消息中间件快

MS

KafKa架构

在我看来，KafKa的架构可以为生产者，消费者，broker以及zookeeper这四大块

生产者负责推送消息到broker，消费者负责从broker拉取消息，broker存储消息，zooker管理注册消息管理消息状态

kafka 与其他消息组件对比

特性	ActiveMQ	RabbitMQ	RocketMQ	Kafka
单机吞吐量	万级，比 RocketMQ、Kafka 低一个数量级	同 ActiveMQ	10 万级，支撑高吞吐	10 万级，高吞吐，一般配合大数据类的系统来进行实时数据计算、日志采集等场景
topic 数量对吞吐量的影响			topic 可以达到几百/几千的级别，吞吐量会有较小幅度的下降，这是 RocketMQ 的一大优势，在同等机器下，可以支撑大量的 topic	topic 从几十到几百个时候，吞吐量会大幅度下降，在同等机器下，Kafka 尽量保证 topic 数量不要过多，如果要支撑大规模的 topic，需要增加更多的机器资源
时效性	ms 级	微秒级，这是 RabbitMQ 的一大特点，延迟最低	ms 级	延迟在 ms 级以内
可用性	高，基于主从架构实现高可用	同 ActiveMQ	非常高，分布式架构	非常高，分布式，一个数据多个副本，少数机器宕机，不会丢失数据，不会导致不可用
消息可靠性	有较低的概率丢失数据	基本不丢	经过参数优化配置，可以做到 0 丢失	同 RocketMQ
功能支持	MQ 领域的功能极其完备	基于 erlang 开发，并发能力很强，性能极好，延时很低	MQ 功能较为完善，还是分布式的，扩展性好	功能较为简单，主要支持简单的 MQ 功能，在大数据领域的实时计算以及日志采集被大规模使用

kafka 实现高吞吐的原理

- 读写文件依赖OS文件系统的页缓存，而不是在JVM内部缓存数据，利用OS来缓存，内存利用率高
- sendfile技术（零拷贝），避免了传统网络IO四步流程
- 支持End-to-End的压缩
- 顺序IO以及常量时间get、put消息
- Partition 可以很好的横向扩展和提供高并发处理

kafka怎样保证不重复消费

此问题其实等价于保证消息队列消费的幂等性

主要需要结合实际业务来操作:

- 比如你拿个数据要写库，你先根据主键查一下，如果这数据都有了，你就别插入了，update 一下好吧。
- 比如你是写 Redis，那没问题了，反正每次都是 set，天然幂等性。
- 比如你不是上面两个场景，那做的稍微复杂一点，你需要让生产者发送每条数据的时候，里面加一个全局唯一的 id，类似订单 id 之类的东西，然后你这里消费到了之后，先根据这个 id 去比如 Redis 里查一下，之前消费过吗？如果没有消费过，你就处理，然后这个 id 写 Redis。如果消费过了，那你就别处理了，保证别重复处理相同的消息即可。
- 比如基于数据库的唯一键来保证重复数据不会重复插入多条。因为有唯一键约束了，重复数据插入只会报错，不会导致数据库中出现脏数据。

kafka怎样保证不丢失消息

消费端弄丢了数据

唯一可能导致消费者弄丢数据的情况，就是说，你消费到了这个消息，然后消费者那边**自动提交了 offset**，让 Kafka 以为你已经消费好了这个消息，但其实你才刚准备处理这个消息，你还没处理，你自己就挂了，此时这条消息就丢咯。

这不是跟 RabbitMQ 差不多吗，大家都知道 Kafka 会自动提交 offset，那么只要**关闭自动提交** offset，在处理完之后自己手动提交 offset，就可以保证数据不会丢。但是此时确实还是**可能会有重复消费**，比如你刚处理完，还没提交 offset，结果自己挂了，此时肯定会重复消费一次，自己保证幂等性就好了。

生产环境碰到的一个问题，就是说我们的 Kafka 消费者消费到了数据之后是写到一个内存的 queue 里先缓冲一下，结果有的时候，你刚把消息写入内存 queue，然后消费者会自动提交 offset。然后此时我们重启了系统，就会导致内存 queue 里还没来得及处理的数据就丢失了。

Kafka 弄丢了数据

这块比较常见的一个场景，就是 Kafka 某个 broker 宕机，然后重新选举 partition 的 leader。大家想想，要是此时其他的 follower 刚好还有些数据没有同步，结果此时 leader 挂了，然后选举某个 follower 成 leader 之后，不就少了一些数据？这就丢了一些数据啊。

生产环境也遇到过，我们也是，之前 Kafka 的 leader 机器宕机了，将 follower 切换为 leader 之后，就会发现说这个数据就丢了。

所以此时一般是要求起码设置如下 4 个参数：

- 给 topic 设置 `replication.factor` 参数：这个值必须大于 1，要求每个 partition 必须有至少 2 个副本。
- 在 Kafka 服务端设置 `min.insync.replicas` 参数：这个值必须大于 1，这个是要求一个 leader 至少感知到有至少一个 follower 还跟自己保持联系，没掉队，这样才能确保 leader 挂了还有一个 follower 吧。
- 在 producer 端设置 `acks=all`：这个是要求每条数据，必须是**写入所有 replica 之后，才能认为是写成功了**。
- 在 producer 端设置 `retries=MAX`（很大很大很大的一个值，无限次重试的意思）：这个是**要求一旦写入失败，就无限重试**，卡在这里了。

我们生产环境就是按照上述要求配置的，这样配置之后，至少在 Kafka broker 端就可以保证在 leader 所在 broker 发生故障，进行 leader 切换时，数据不会丢失。

生产者会不会弄丢数据?

如果按照上述的思路设置了 `acks=all`，一定不会丢，要求是，你的 leader 接收到消息，所有的 follower 都同步到了消息之后，才认为本次写成功了。如果没满足这个条件，生产者会自动不断的重试，重试无限次。

Ack 有哪几种, 生产中怎样选择?

ack=0/1/-1的不同情况：

- Ack = 0

  producer不等待broker的ack，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；

- Ack = 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；

- Ack = -1

  producer等待broker的ack，partition的leader和follower全部落盘成功后才返回ack，数据一般不会丢失，延迟时间长但是可靠性高。

生产中主要以 Ack=-1为主,如果压力过大,可切换为Ack=1. Ack=0的情况只能在测试中使用

如何通过offset寻找数据

如果consumer要找offset是1008的消息，那么，

1，按照二分法找到小于1008的segment，也就是00000000000000001000.log和00000000000000001000.index

2，用目标offset减去文件名中的offset得到消息在这个segment中的偏移量。也就是1008-1000=8，偏移量是8。

3，再次用二分法在index文件中找到对应的索引，也就是第三行6,45。

4，到log文件中，从偏移量45的位置开始（实际上这里的消息offset是1006），顺序查找，直到找到offset为1008的消息。查找期间kafka是按照log的存储格式来判断一条消息是否结束的。

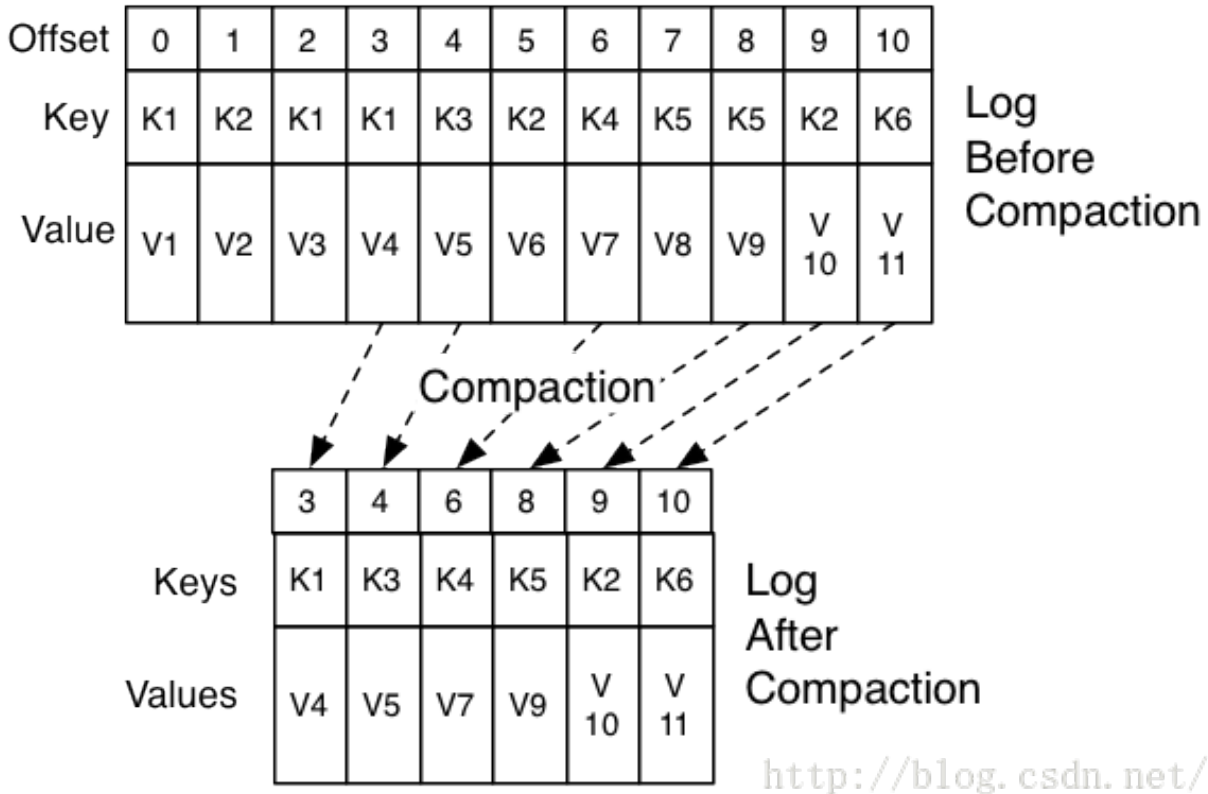
如何清理过期数据

- **删除**

  log.cleanup.policy=delete启用删除策略

  - 直接删除，删除后的消息不可恢复。可配置以下两个策略： 清理超过指定时间清理：log.retention.hours=16
  - 超过指定大小后，删除旧的消息： log.retention.bytes=1073741824 为了避免在删除时阻塞读操作，采用了copy-on-write形式的实现，删除操作进行时，读取操作的二分查找功能实际是在一个静态的快照副本上进行的，这类似于Java的CopyOnWriteArrayList。
- **压缩**

  将数据压缩，只保留每个key最后一个版本的数据。 首先在broker的配置中设置log.cleaner.enable=true启用cleaner，这个默认是关闭的。 在topic的配置中设置log.cleanup.policy=compact启用压缩策略。

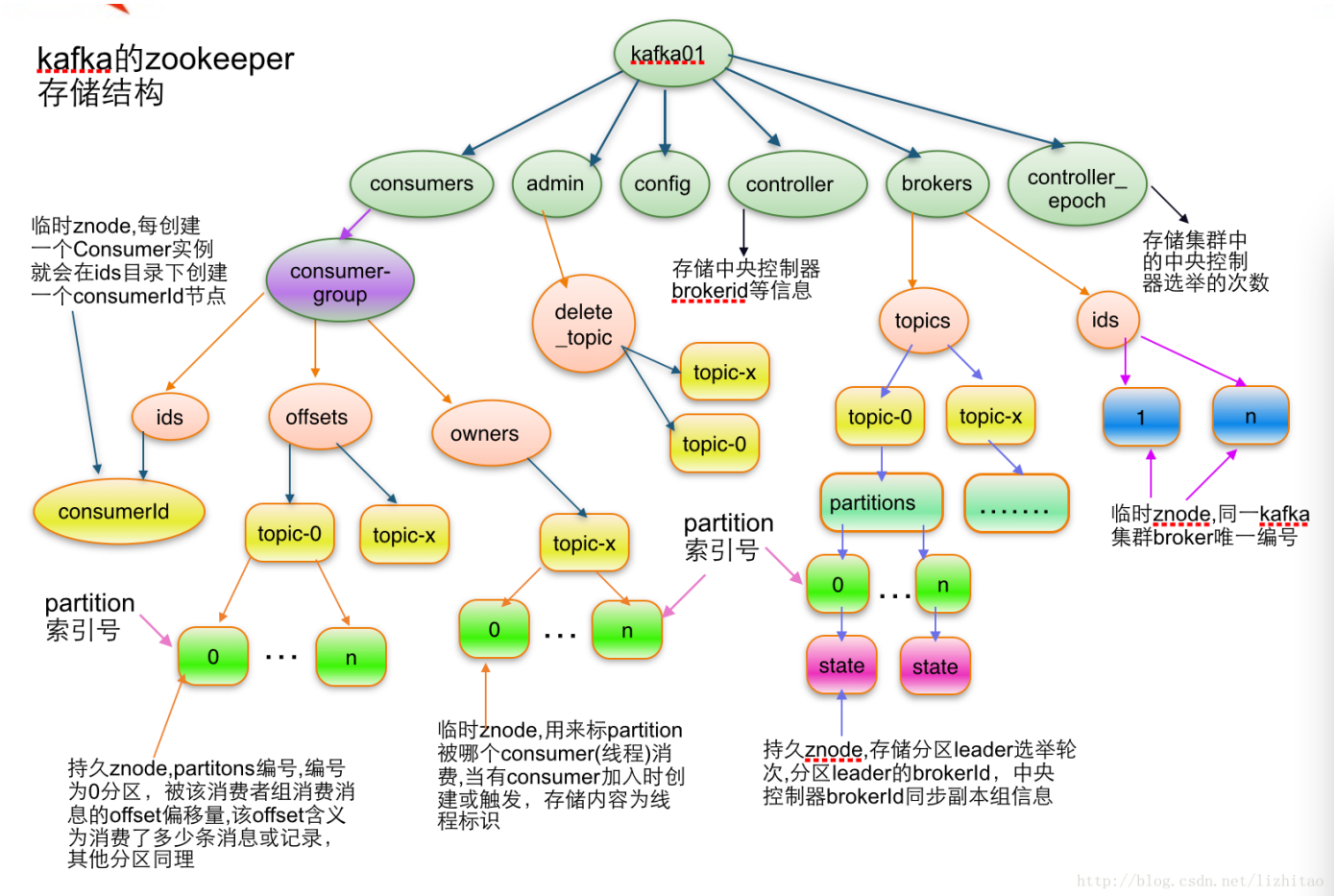


  如上图，在整个数据流中，每个Key都有可能出现多次，压缩时将根据Key将消息聚合，只保留最后一次出现时的数据。这样，无论什么时候消费消息，都能拿到每个Key的最新版本的数据。 压缩后的offset可能是不连续的，比如上图中没有5和7，因为这些offset的消息被merge了，当从这些offset消费消息时，将会拿到比这个offset大的offset对应的消息，比如，当试图获取offset为5的消息时，实际上会拿到offset为6的消息，并从这个位置开始消费。 这种策略只适合特殊场景，比如消息的key是用户ID，消息体是用户的资料，通过这种压缩策略，整个消息集里就保存了所有用户最新的资料。 压缩策略支持删除，当某个Key的最新版本的消息没有内容时，这个Key将被删除，这也符合以上逻辑。

1条message中包含哪些信息

Field	Description
Attributes	该字节包含有关消息的元数据属性。 最低的2位包含用于消息的压缩编解码器。 其他位应设置为0。
Crc	CRC是消息字节的其余部分的CRC32。 这用于检查代理和使用者上的消息的完整性。
	key是用于分区分配的可选参数。 key可以为null。
MagicByte	这是用于允许向后兼容的消息二进制格式演变的版本ID。 当前值为0。
Offset	这是kafka中用作日志序列号的偏移量。 当producer发送消息时，它实际上并不知道偏移量，并且可以填写它喜欢的任何值。
Value	该值是实际的消息内容，作为不透明的字节数组。 Kafka支持递归消息，在这种情况下，它本身可能包含消息集。 消息可以为null。

zookeeper在kafka中的作用



zk的作用主要有如下几点:

1. kafka的元数据都存放在zk上面,由zk来管理
2. 0.8之前版本的kafka, consumer的消费状态，group的管理以及 offset的值都是由zk管理的,现在offset会保存在本地topic文件里
3. 负责borker的leader选举和管理

kafka 可以脱离 zookeeper 单独使用吗

kafka 不能脱离 zookeeper 单独使用，因为 kafka 使用 zookeeper 管理和协调 kafka 的节点服务器。

kafka同时设置了7天和10G清除数据,到第5天的时候消息到达了10G,这个时候kafka如何处理?

这个时候 kafka 会执行数据清除工作，时间和大小不论那个满足条件，都会清空数据。