# Kafka面试题

## 一、Kafka目前有哪些内部topic，它们都有什么特征？各自的作用又是什么？

**\_\_consumer\_offsets**：作用是保存Kafka消费者的位移信息

**\_\_transaction\_state**：用来存储事务日志消息

## 二、优先副本是什么？它有什么特殊的作用？

所谓的优先副本是指在AR集合列表中的第一个副本。理想情况下，优先副本就是该分区的leader副本，所以也可以称之为preferred leader（首选领导者）。Kafka要确保所有主题的优先副本在Kafka集群中均匀分布，这样就保证了所有分区的 leader均衡分布。以此来促进集群的负载均衡，这一行为也可以称为“分区平衡”。

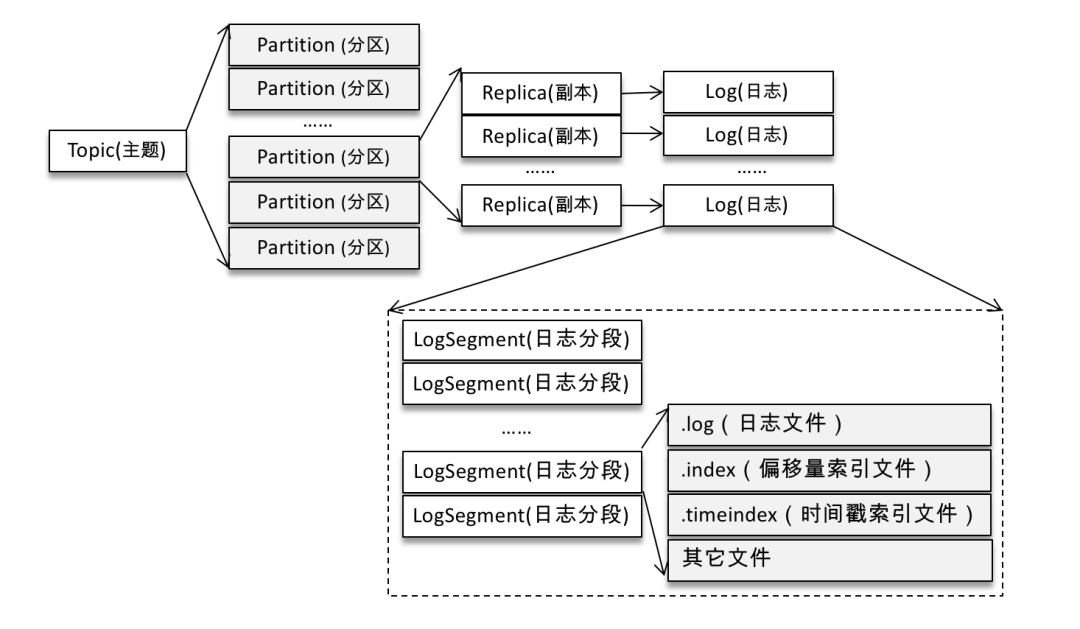
## 三、Kafka有哪几处地方有分区分配的概念？简述大致的过程及原理

生产者的分区分配是指为每条消息指定其所要发往的分区。可以编写一个具体的类实现org.apache.kafka.clients.producer.Partitioner接口。

消费者中的分区分配是指为消费者指定其可以消费消息的分区。Kafka 提供了消费者客户端参数 partition.assignment.strategy 来设置消费者与订阅主题之间的分区分配策略。

分区副本的分配是指为集群制定创建主题时的分区副本分配方案，即在哪个broker中创建哪些分区的副本。kafka-topics.sh 脚本中提供了一个replica-assignment参数来手动指定分区副本的分配方案。

## 四、简述Kafka的日志目录结构



Kafka中的消息是以主题为基本单位进行归类的，各个主题在逻辑上相互独立。每个主题又可以分为一个或多个分区。不考虑多副本的情况，一个分区对应一个日志（Log）。为了防止Log过大，Kafka又引入了日志分段（LogSegment）的概念，将Log切分为多个LogSegment，相当于一个巨型文件被平均分配为多个相对较小的文件。

Log和LogSegment也不是纯粹物理意义上的概念，Log在物理上只以文件夹的形式存储，而每个LogSegment对应于磁盘上的一个日志文件和两个索引文件，以及可能的其他文件（比如以“.txnindex”为后缀的事务索引文件）。

## 五、Kafka中有哪些索引文件？

每个日志分段文件对应了两个索引文件，主要用来提高查找消息的效率。

偏移量索引文件用来建立消息偏移量（offset）到物理地址之间的映射关系，方便快速定位消息所在的物理文件位置

时间戳索引文件则根据指定的时间戳（timestamp）来查找对应的偏移量信息。

## 六、如果我指定了一个offset，Kafka怎么查找到对应的消息？

Kafka是通过seek() 方法来指定消费的，在执行seek() 方法之前要去执行一次poll()方法，等到分配到分区之后会去对应的分区的指定位置开始消费，如果指定的位置发生了越界，那么会根据auto.offset.reset 参数设置的情况进行消费。

## 七、如果我指定了一个timestamp，Kafka怎么查找到对应的消息？

Kafka提供了一个 offsetsForTimes() 方法，通过 timestamp 来查询与此对应的分区位置。offsetsForTimes() 方法的参数 timestampsToSearch 是一个 Map 类型，key 为待查询的分区，而 value 为待查询的时间戳，该方法会返回时间戳大于等于待查询时间的第一条消息对应的位置和时间戳，对应于OffsetAndTimestamp中的offset和timestamp字段。

## 八、聊一聊你对Kafka的Log Retention的理解

日志删除（Log Retention）：按照一定的保留策略直接删除不符合条件的日志分段。

可以通过 broker 端参数log.cleanup.policy来设置日志清理策略，此参数的默认值为“delete”，即采用日志删除的清理策略。

### 基于时间

日志删除任务会检查当前日志文件中是否有保留时间超过设定的阈值（retentionMs）来寻找可删除的日志分段文件集合（deletableSegments）retentionMs 可以通过 broker 端参数 log.retention.hours、log.retention.minutes和log.retention.ms来配置，其中log.retention.ms的优先级最高，log.retention.minutes次之，log.retention.hours最低。默认情况下只配置了log.retention.hours参数，其值为168，故默认情况下日志分段文件的保留时间为7天。

删除日志分段时，首先会从Log对象中所维护日志分段的跳跃表中移除待删除的日志分段，以保证没有线程对这些日志分段进行读取操作。然后将日志分段所对应的所有文件添加上“.deleted”的后缀（当然也包括对应的索引文件）。最后交由一个以“delete-file”命名的延迟任务来删除这些以“.deleted”为后缀的文件，这个任务的延迟执行时间可以通过file.delete.delay.ms参数来调配，此参数的默认值为60000，即1分钟。

### 基于日志大小

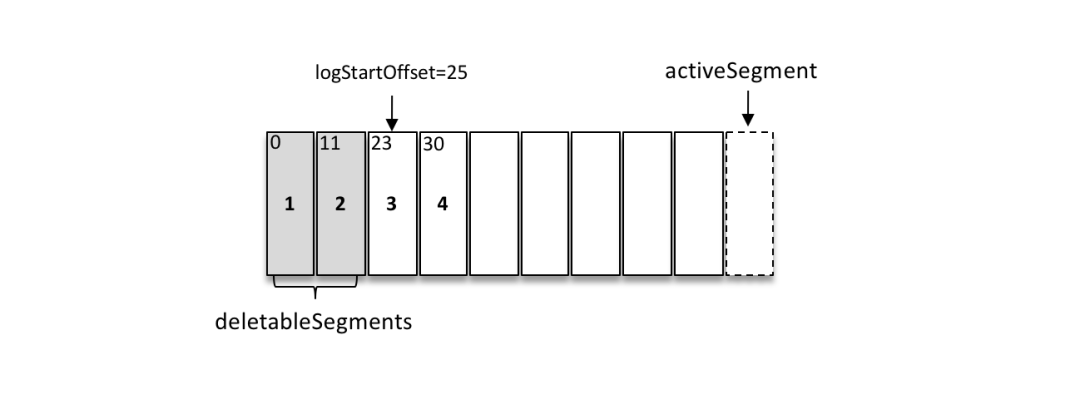
日志删除任务会检查当前日志的大小是否超过设定的阈值（retentionSize）来寻找可删除的日志分段的文件集合（deletableSegments）。

retentionSize可以通过broker端参数log.retention.bytes来配置，默认值为-1，表示无穷大。注意log.retention.bytes 配置的是Log 中所有日志文件的总大小，而不是单个日志分段（确切地说应该为 .log 日志文件）的大小。单个日志分段的大小由broker端参数log.segment.bytes来限制，默认值为1073741824，即1GB。

这个删除操作和基于时间的保留策略的删除操作相同。

### 基于日志起始偏移量

基于日志起始偏移量的保留策略的判断依据是某日志分段的下一个日志分段的起始偏移量baseOffset是否小于等于logStartOffset，若是，则可以删除此日志分段。



如上图所示，假设logStartOffset等于25，日志分段1的起始偏移量为0，日志分段2的起始偏移量为11，日志分段3的起始偏移量为23，通过如下动作收集可删除的日志分段的文件集合deletableSegments：

从头开始遍历每个日志分段，日志分段1的下一个日志分段的起始偏移量为11，小于logStartOffset的大小，将日志分段1加入deletableSegments。

日志分段2的下一个日志偏移量的起始偏移量为23，也小于 logStartOffset 的大小，将日志分段2加入deletableSegments。

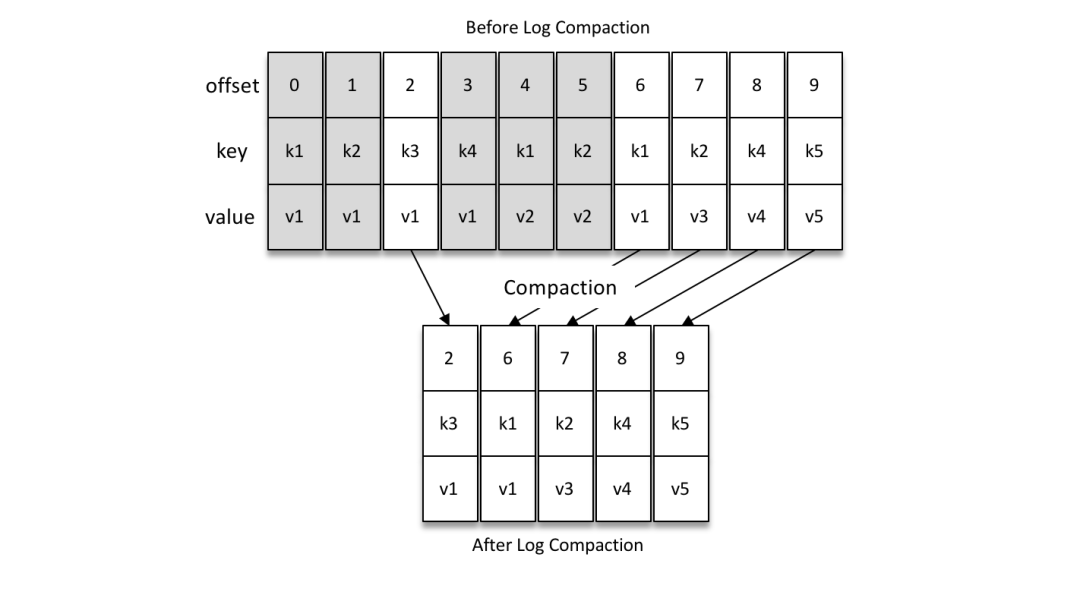
日志分段3的下一个日志偏移量在logStartOffset的右侧，故从日志分段3开始的所有日志分段都不会加入deletableSegments。

收集完可删除的日志分段的文件集合之后的删除操作同基于日志大小的保留策略和基于时间的保留策略相同。

## 九、聊一聊你对Kafka的Log Compaction的理解

日志压缩（Log Compaction）：针对每个消息的key进行整合，对于有相同key的不同value值，只保留最后一个版本。

如果要采用日志压缩的清理策略，就需要将log.cleanup.policy设置为“compact”，并且还需要将log.cleaner.enable （默认值为 true）设定为true。



如下图所示，Log Compaction对于有相同key的不同value值，只保留最后一个版本。如果应用只关心key对应的最新value值，则可以开启Kafka的日志清理功能，Kafka会定期将相同key的消息进行合并，只保留最新的value值。

## 十、聊一聊你对Kafka底层存储的理解

### 页缓存

页缓存是操作系统实现的一种主要的磁盘缓存，以此用来减少对磁盘I/O的操作。具体来说，就是把磁盘中的数据缓存到内存中，把对磁盘的访问变为对内存的访问。

当一个进程准备读取磁盘上的文件内容时，操作系统会先查看待读取的数据所在的页（page）是否在页缓存（pagecache）中，如果存在（命中）则直接返回数据，从而避免了对物理磁盘的I/O操作；如果没有命中，则操作系统会向磁盘发起读取请求并将读取的数据页存入页缓存，之后再将数据返回给进程。

同样，如果一个进程需要将数据写入磁盘，那么操作系统也会检测数据对应的页是否在页缓存中，如果不存在，则会先在页缓存中添加相应的页，最后将数据写入对应的页。被修改过后的页也就变成了脏页，操作系统会在合适的时间把脏页中的数据写入磁盘，以保持数据的一致性。

用过Java的人一般都知道两点事实：对象的内存开销非常大，通常会是真实数据大小的几倍甚至更多，空间使用率低下；Java的垃圾回收会随着堆内数据的增多而变得越来越慢。基于这些因素，使用文件系统并依赖于页缓存的做法明显要优于维护一个进程内缓存或其他结构，至少我们可以省去了一份进程内部的缓存消耗，同时还可以通过结构紧凑的字节码来替代使用对象的方式以节省更多的空间。

此外，即使Kafka服务重启，页缓存还是会保持有效，然而进程内的缓存却需要重建。这样也极大地简化了代码逻辑，因为维护页缓存和文件之间的一致性交由操作系统来负责，这样会比进程内维护更加安全有效。

### 零拷贝

除了消息顺序追加、页缓存等技术，Kafka还使用零拷贝（Zero-Copy）技术来进一步提升性能。所谓的零拷贝是指将数据直接从磁盘文件复制到网卡设备中，而不需要经由应用程序之手。零拷贝大大提高了应用程序的性能，减少了内核和用户模式之间的上下文切换。对Linux操作系统而言，零拷贝技术依赖于底层的 sendfile() 方法实现。对应于Java语言，FileChannal.transferTo() 方法的底层实现就是 sendfile() 方法。

## 十一、聊一聊Kafka的延时操作的原理

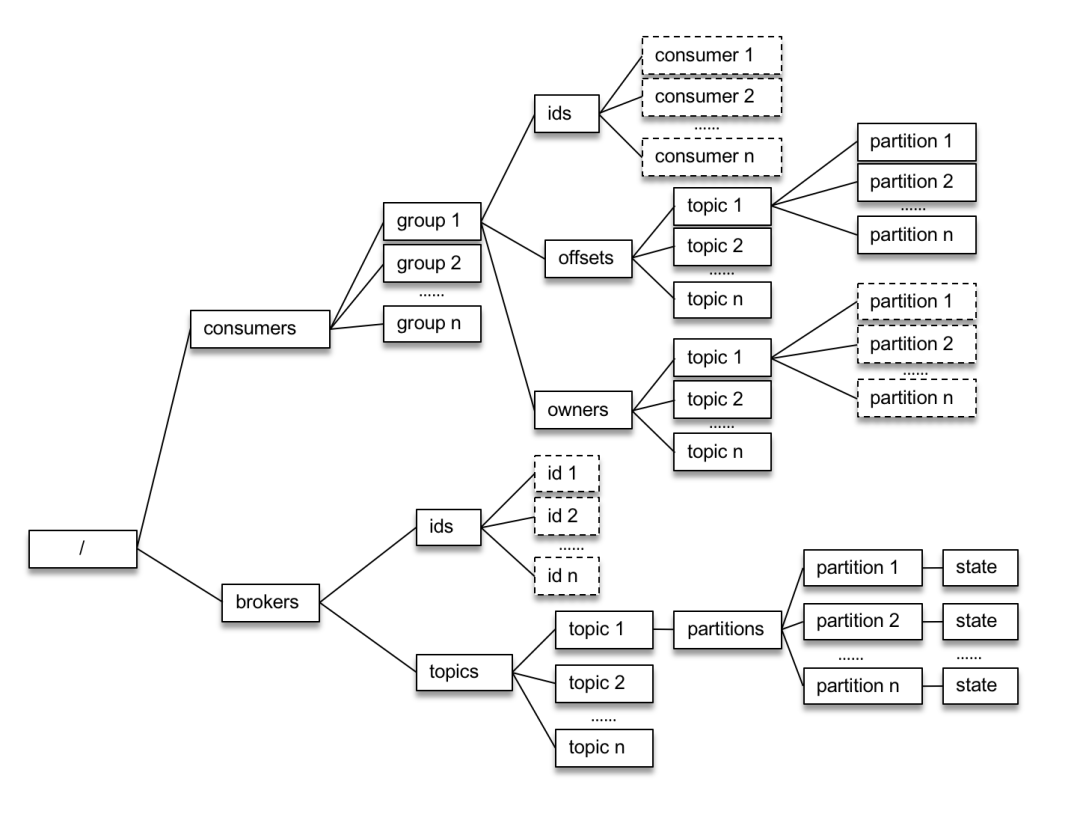
Kafka中有多种延时操作，比如延时生产，还有延时拉取（DelayedFetch）、延时数据删除（DelayedDeleteRecords）等。

延时操作创建之后会被加入延时操作管理器（DelayedOperationPurgatory）来做专门的处理。延时操作有可能会超时，每个延时操作管理器都会配备一个定时器（SystemTimer）来做超时管理，定时器的底层就是采用时间轮（TimingWheel）实现的。

## 十二、聊一聊Kafka控制器的作用

在Kafka集群中会有一个或多个broker，其中有一个broker会被选举为控制器（Kafka Controller），它负责管理整个集群中所有分区和副本的状态。当某个分区的leader副本出现故障时，由控制器负责为该分区选举新的leader副本。当检测到某个分区的ISR集合发生变化时，由控制器负责通知所broker更新其元数据信息。当使用kafka-topics.sh脚本为某个topic增加分区数量时，同样还是由控制器负责分区的重新分配。

## 十三、Kafka的旧版Scala的消费者客户端的设计有什么缺陷？



如上图，旧版消费者客户端每个消费组（）在ZooKeeper中都维护了一个 /consumers//ids 路径，在此路径下使用临时节点记录隶属于此消费组的消费者的唯一标识（consumerIdString），/consumers//owner路径下记录了分区和消费者的对应关系，/consumers//offsets 路径下记录了此消费组在分区中对应的消费位移。

每个消费者在启动时都会在 /consumers//ids 和 /brokers/ids 路径上注册一个监听器。当 /consumers//ids 路径下的子节点发生变化时，表示消费组中的消费者发生了变化；当 /brokers/ids 路径下的子节点发生变化时，表示 broker出现了增减。这样通过ZooKeeper所提供的Watcher，每个消费者就可以监听消费组和Kafka集群的状态了。

这种方式下每个消费者对ZooKeeper的相关路径分别进行监听，当触发再均衡操作时，一个消费组下的所有消费者会同时进行再均衡操作，而消费者之间并不知道彼此操作的结果，这样可能导致Kafka工作在一个不正确的状态。与此同时，这种严重依赖于ZooKeeper集群的做法还有两个比较严重的问题。

* **羊群效应**（Herd Effect）：所谓的羊群效应是指ZooKeeper中一个被监听的节点变化，大量的Watcher通知被发送到客户端，导致在通知期间的其他操作延迟，也有可能发生类似死锁的情况。
* **脑裂问题**（Split Brain）：消费者进行再均衡操作时每个消费者都与ZooKeeper进行通信以判断消费者或broke变化的情况，由于ZooKeeper本身的特性，可能导致在同一时刻各个消费者获取的状态不一致，这样会导致异常问题发生。

## 十四、消费再均衡的原理是什么？（提示：消费者协调器和消费组协调器）

就目前而言，一共有如下几种情形会触发再均衡的操作：

* 有新的消费者加入消费组。
* 有消费者宕机下线。消费者并不一定需要真正下线，例如遇到长时间的GC、网络延迟导致消费者长时间未向GroupCoordinator发送心跳等情况时，GroupCoordinator会认为消费者已经下线。
* 有消费者主动退出消费组（发送LeaveGroupRequest请求）。比如客户端调用了unsubscrible() 方法取消对某些主题的订阅。
* 消费组所对应的GroupCoorinator节点发生了变更。
* 消费组内所订阅的任一主题或者主题的分区数量发生变化。

GroupCoordinator是Kafka服务端中用于管理消费组的组件。而消费者客户端中的ConsumerCoordinator组件负责与GroupCoordinator进行交互。

### 第一阶段（FIND\_COORDINATOR）

消费者需要确定它所属的消费组对应的GroupCoordinator所在的broker，并创建与该broker相互通信的网络连接。如果消费者已经保存了与消费组对应的GroupCoordinator节点的信息，并且与它之间的网络连接是正常的，那么就可以进入第二阶段。否则，就需要向集群中的某个节点发送FindCoordinatorRequest请求来查找对应的 GroupCoordinator，这里的“某个节点”并非是集群中的任意节点，而是负载最小的节点。

### 第二阶段（JOIN\_GROUP）

在成功找到消费组所对应的GroupCoordinator之后就进入加入消费组的阶段，在此阶段的消费者会向 GroupCoordinator发送JoinGroupRequest请求，并处理响应。

选举消费组的leader

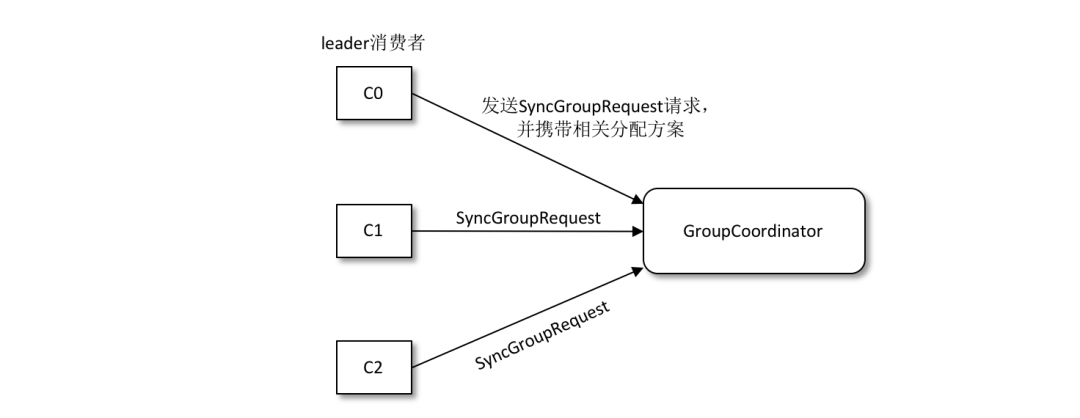
如果消费组内还没有leader，那么第一个加入消费组的消费者即为消费组的leader。如果某一时刻leader消费者由于某些原因退出了消费组，那么会重新选举一个新的leader。

选举分区分配策略

* 收集各个消费者支持的所有分配策略，组成候选集candidates。
* 每个消费者从候选集candidates中找出第一个自身支持的策略，为这个策略投上一票。
* 计算候选集中各个策略的选票数，选票数最多的策略即为当前消费组的分配策略。

### 第三阶段（SYNC\_GROUP）

leader消费者根据在第二阶段中选举出来的分区分配策略来实施具体的分区分配，在此之后需要将分配的方案同步给各个消费者，通过GroupCoordinator这个“中间人”来负责转发同步分配方案的。



### 第四阶段（HEARTBEAT）

进入这个阶段之后，消费组中的所有消费者就会处于正常工作状态。在正式消费之前，消费者还需要确定拉取消息的起始位置。假设之前已经将最后的消费位移提交到了GroupCoordinator，并且GroupCoordinator将其保存到了 Kafka内部的 \_\_consumer\_offsets 主题中，此时消费者可以通过OffsetFetchRequest请求获取上次提交的消费位移并从此处继续消费。

消费者通过向GroupCoordinator发送心跳来维持它们与消费组的从属关系，以及它们对分区的所有权关系。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中的消息。心跳线程是一个独立的线程，可以在轮询消息的空档发送心跳。如果消费者停止发送心跳的时间足够长，则整个会话就被判定为过期，GroupCoordinator也会认为这个消费者已经死亡，就会触发一次再均衡行为。

## 十五、Kafka中的幂等是怎么实现的？

为了实现生产者的幂等性，Kafka为此引入了producer id（以下简称 PID）和序列号（sequence number）这两个概念。

每个新的生产者实例在初始化的时候都会被分配一个PID，这个PID对用户而言是完全透明的。对于每个PID，消息发送到的每一个分区都有对应的序列号，这些序列号从0开始单调递增。生产者每发送一条消息就会将<PID，分区> 对应的序列号的值加1。

broker 端会在内存中为每一对 <PID，分区> 维护一个序列号。对于收到的每一条消息，只有当它的序列号的值（SN\_new）比 broker 端中维护的对应的序列号的值（SN\_old）大1（即 SN\_new = SN\_old + 1）时，broker 才会接收它。如果 SN\_new< SN\_old + 1，那么说明消息被重复写入，broker 可以直接将其丢弃。如果 SN\_new> SN\_old + 1，那么说明中间有数据尚未写入，出现了乱序，暗示可能有消息丢失，对应的生产者会抛出 OutOfOrderSequenceException，这个异常是一个严重的异常，后续的诸如send()、beginTransaction()、commitTransaction() 等方法的调用都会抛出IllegalStateException的异常。

## 十六、为什么要使用 kafka，为什么要使用消息队列

* **缓冲和削峰**：上游数据时有突发流量，下游可能扛不住，或者下游没有足够多的机器来保证冗余，kafka在中间可以起到一个缓冲的作用，把消息暂存在kafka中，下游服务就可以按照自己的节奏进行慢慢处理。
* **解耦和扩展性**：项目开始的时候，并不能确定具体需求。消息队列可以作为一个接口层，解耦重要的业务流程。只需要遵守约定，针对数据编程即可获取扩展能力。
* **冗余**：可以采用一对多的方式，一个生产者发布消息，可以被多个订阅topic的服务消费到，供多个毫无关联的业务使用。
* **健壮性**：消息队列可以堆积请求，所以消费端业务即使短时间死掉，也不会影响主要业务的正常进行。
* **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。

十七、Kafka中的ISR、AR又代表什么？ISR的伸缩又指什么

ISR:In-Sync Replicas 副本同步队列

AR:Assigned Replicas 所有副本

ISR是由leader维护，follower从leader同步数据有一些延迟（包括延迟时间replica.lag.time.max.ms和延迟条数replica.lag.max.messages两个维度, 当前最新的版本0.10.x中只支持replica.lag.time.max.ms这个维度），任意一个超过阈值都会把follower剔除出ISR, 存入OSR（Outof-Sync Replicas）列表，新加入的follower也会先存放在OSR中。AR=ISR+OSR。

## 十八、kafka中的broker 是干什么的

broker 是消息的代理，Producers往Brokers里面的指定Topic中写消息，Consumers从Brokers里面拉取指定Topic的消息，然后进行业务处理，broker在中间起到一个代理保存消息的中转站。

kafka中的 zookeeper 起到什么作用，可以不用zookeeper么

zookeeper 是一个分布式的协调组件，早期版本的kafka用zk做meta信息存储，consumer的消费状态，group的管理以及 offset的值。考虑到zk本身的一些因素以及整个架构较大概率存在单点问题，新版本中逐渐弱化了zookeeper的作用。新的consumer使用了kafka内部的group coordination协议，也减少了对zookeeper的依赖，

但是broker依然依赖于ZK，zookeeper 在kafka中还用来选举controller 和 检测broker是否存活等等。

## 十九、kafka follower如何与leader同步数据

Kafka的复制机制既不是完全的同步复制，也不是单纯的异步复制。完全同步复制要求All Alive Follower都复制完，这条消息才会被认为commit，这种复制方式极大的影响了吞吐率。而异步复制方式下，Follower异步的从Leader复制数据，数据只要被Leader写入log就被认为已经commit，这种情况下，如果leader挂掉，会丢失数据，kafka使用ISR的方式很好的均衡了确保数据不丢失以及吞吐率。Follower可以批量的从Leader复制数据，而且Leader充分利用磁盘顺序读以及send file(zero copy)机制，这样极大的提高复制性能，内部批量写磁盘，大幅减少了Follower与Leader的消息量差。

## 二十、什么情况下一个 broker 会从 isr中踢出去

leader会维护一个与其基本保持同步的Replica列表，该列表称为ISR(in-sync Replica)，每个Partition都会有一个ISR，而且是由leader动态维护 ，如果一个follower比一个leader落后太多，或者超过一定时间未发起数据复制请求，则leader将其重ISR中移除 。

## 二十一、kafka 为什么那么快

* Cache Filesystem Cache PageCache缓存
* 顺序写 由于现代的操作系统提供了预读和写技术，磁盘的顺序写大多数情况下比随机写内存还要快。
* Zero-copy 零拷技术减少拷贝次数
* Batching of Messages 批量量处理。合并小的请求，然后以流的方式进行交互，直顶网络上限。
* Pull 拉模式 使用拉模式进行消息的获取消费，与消费端处理能力相符。

## 二十二、kafka producer如何优化打入速度

增加线程

提高 batch.size

增加更多 producer 实例

增加 partition 数

设置 acks=-1 时，如果延迟增大：可以增大 num.replica.fetchers（follower 同步数据的线程数）来调解；

跨数据中心的传输：增加 socket 缓冲区设置以及 OS tcp 缓冲区设置。

## 二十三、kafka producer 打数据，ack 为 0， 1， -1 的时候代表啥， 设置 -1 的时候，什么情况下，leader 会认为一条消息 commit了

1（默认） 数据发送到Kafka后，经过leader成功接收消息的的确认，就算是发送成功了。在这种情况下，如果leader宕机了，则会丢失数据。

0 生产者将数据发送出去就不管了，不去等待任何返回。这种情况下数据传输效率最高，但是数据可靠性确是最低的。

-1 producer需要等待ISR中的所有follower都确认接收到数据后才算一次发送完成，可靠性最高。当ISR中所有Replica都向Leader发送ACK时，leader才commit，这时候producer才能认为一个请求中的消息都commit了。

## 二十四、kafka unclean 配置代表啥，会对 spark streaming 消费有什么影响

unclean.leader.election.enable 为true的话，意味着非ISR集合的broker 也可以参与选举，这样有可能就会丢数据，spark streaming在消费过程中拿到的 end offset 会突然变小，导致 spark streaming job挂掉。如果unclean.leader.election.enable参数设置为true，就有可能发生数据丢失和数据不一致的情况，Kafka的可靠性就会降低；而如果unclean.leader.election.enable参数设置为false，Kafka的可用性就会降低。

## 二十五、如果leader crash时，ISR为空怎么办

kafka在Broker端提供了一个配置参数：unclean.leader.election,这个参数有两个值：

true（默认）：允许不同步副本成为leader，由于不同步副本的消息较为滞后，此时成为leader，可能会出现消息不一致的情况。

false：不允许不同步副本成为leader，此时如果发生ISR列表为空，会一直等待旧leader恢复，降低了可用性。

## 二十六、kafka的message格式是什么样的

一个Kafka的Message由一个固定长度的header和一个变长的消息体body组成

header部分由一个字节的magic(文件格式)和四个字节的CRC32(用于判断body消息体是否正常)构成。

当magic的值为1的时候，会在magic和crc32之间多一个字节的数据：attributes(保存一些相关属性，

比如是否压缩、压缩格式等等);如果magic的值为0，那么不存在attributes属性

body是由N个字节构成的一个消息体，包含了具体的key/value消息

## 二十七、kafka中consumer group（消费者组）是什么概念

同样是逻辑上的概念，是Kafka实现单播和广播两种消息模型的手段。同一个topic的数据，会广播给不同的group；同一个group中的worker，只有一个worker能拿到这个数据。换句话说，对于同一个topic，每个group都可以拿到同样的所有数据，但是数据进入group后只能被其中的一个worker消费。group内的worker可以使用多线程或多进程来实现，也可以将进程分散在多台机器上，worker的数量通常不超过partition的数量，且二者最好保持整数倍关系，因为Kafka在设计时假定了一个partition只能被一个worker消费（同一group内）。

消费者组是 Kafka 独有的概念，标准答案：

1、定义：即消费者组是 Kafka 提供的可扩展且具有容错性的消费者机制。

2、原理：在 Kafka 中，消费者组是一个由多个消费者实例 构成的组。多个实例共同订阅若干个主题，实现共同消费。同一个组下的每个实例都配置有 相同的组 ID，被分配不同的订阅分区。当某个实例挂掉的时候，其他实例会自动地承担起 它负责消费的分区。

## 二十八、Kafka中的消息是否会丢失和重复消费？

要确定Kafka的消息是否丢失或重复，从两个方面分析入手：消息发送和消息消费。

**1、消息发送**

Kafka消息发送有两种方式：同步（sync）和异步（async），默认是同步方式，可通过producer.type属性进行配置。Kafka通过配置request.required.acks属性来确认消息的生产：

0---表示不进行消息接收是否成功的确认；

1---表示当Leader接收成功时确认；

-1---表示Leader和Follower都接收成功时确认；

综上所述，有6种消息生产的情况，下面分情况来分析消息丢失的场景：

（1）acks=0，不和Kafka集群进行消息接收确认，则当网络异常、缓冲区满了等情况时，消息可能丢失；

（2）acks=1、同步模式下，只有Leader确认接收成功后但挂掉了，副本没有同步，数据可能丢失；

**2、消息消费**

Kafka消息消费有两个consumer接口，Low-level API和High-level API：

Low-level API：消费者自己维护offset等值，可以实现对Kafka的完全控制；

High-level API：封装了对parition和offset的管理，使用简单；

如果使用高级接口High-level API，可能存在一个问题就是当消息消费者从集群中把消息取出来、并提交了新的消息offset值后，还没来得及消费就挂掉了，那么下次再消费时之前没消费成功的消息就“诡异”的消失了；

解决办法：

**针对消息丢失**：同步模式下，确认机制设置为-1，即让消息写入Leader和Follower之后再确认消息发送成功；异步模式下，为防止缓冲区满，可以在配置文件设置不限制阻塞超时时间，当缓冲区满时让生产者一直处于阻塞状态；

**针对消息重复**：将消息的唯一标识保存到外部介质中，每次消费时判断是否处理过即可。

消息重复消费及解决参考：<https://www.javazhiyin.com/22910.html>

## 二十九、为什么Kafka不支持读写分离？

在Kafka中，生产者写入消息、消费者读取消息的操作都是与leader副本进行交互的，从而实现的是一种主写主读的生产消费模型。

Kafka并不支持主写从读，因为主写从读有2个很明显的缺点:

(1)数据一致性问题。数据从主节点转到从节点必然会有一个延时的时间窗口，这个时间窗口会导致主从节点之间的数据不一致。某一时刻，在主节点和从节点中A数据的值都为X， 之后将主节点中A的值修改为Y，那么在这个变更通知到从节点之前，应用读取从节点中的A数据的值并不为最新的Y，由此便产生了数据不一致的问题。

(2)延时问题。类似Redis这种组件，数据从写入主节点到同步至从节点中的过程需要经历网络→主节点内存→网络→从节点内存这几个阶段，整个过程会耗费一定的时间。而在Kafka中，主从同步会比Redis更加耗时，它需要经历网络→主节点内存→主节点磁盘→网络→从节 点内存→从节点磁盘这几个阶段。对延时敏感的应用而言，主写从读的功能并不太适用。

## 三十、Kafka中是怎么体现消息顺序性的？

kafka每个partition中的消息在写入时都是有序的，消费时，每个partition只能被每一个group中的一个消费者消费，保证了消费时也是有序的。

整个topic不保证有序。如果为了保证topic整个有序，那么将partition调整为1.

## 三十一、消费者提交消费位移时提交的是当前消费到的最新消息的offset还是offset+1?

offset+1

## 三十二、kafka如何实现延迟队列？

Kafka并没有使用JDK自带的Timer或者DelayQueue来实现延迟的功能，而是基于时间轮自定义了一个用于实现延迟功能的定时器（SystemTimer）。JDK的Timer和DelayQueue插入和删除操作的平均时间复杂度为O(nlog(n))，并不能满足Kafka的高性能要求，而基于时间轮可以将插入和删除操作的时间复杂度都降为O(1)。时间轮的应用并非Kafka独有，其应用场景还有很多，在Netty、Akka、Quartz、Zookeeper等组件中都存在时间轮的踪影。

底层使用数组实现，数组中的每个元素可以存放一个TimerTaskList对象。TimerTaskList是一个环形双向链表，在其中的链表项TimerTaskEntry中封装了真正的定时任务TimerTask.

Kafka中到底是怎么推进时间的呢？Kafka中的定时器借助了JDK中的DelayQueue来协助推进时间轮。具体做法是对于每个使用到的TimerTaskList都会加入到DelayQueue中。Kafka中的TimingWheel专门用来执行插入和删除TimerTaskEntry的操作，而DelayQueue专门负责时间推进的任务。再试想一下，DelayQueue中的第一个超时任务列表的expiration为200ms，第二个超时任务为840ms，这里获取DelayQueue的队头只需要O(1)的时间复杂度。如果采用每秒定时推进，那么获取到第一个超时的任务列表时执行的200次推进中有199次属于“空推进”，而获取到第二个超时任务时有需要执行639次“空推进”，这样会无故空耗机器的性能资源，这里采用DelayQueue来辅助以少量空间换时间，从而做到了“精准推进”。Kafka中的定时器真可谓是“知人善用”，用TimingWheel做最擅长的任务添加和删除操作，而用DelayQueue做最擅长的时间推进工作，相辅相成。

## 三十三、在 Kafka 中，ZooKeeper 的作用是什么?

目前，Kafka使用ZooKeeper存放集群元数据、成员管理、Controller选举，以及其他一些管理类任务。之后，等KIP-500提案完成后，Kafka将完全不再依赖于ZooKeeper。

记住，**一定要突出“目前”**，以彰显你非常了解社区的演进计划。“存放元数据”是指主题 分区的所有数据都保存在 ZooKeeper 中，且以它保存的数据为权威，其他“人”都要与它保持对齐。“成员管理”是指 Broker 节点的注册、注销以及属性变更等等。“Controller 选举”是指选举集群 Controller，而其他管理类任务包括但不限于主题删除、参数配置等。

不过，抛出 KIP-500 也可能是个双刃剑。碰到非常资深的面试官，他可能会进一步追问你 KIP-500 是做的。一言以蔽之:**KIP-500思想，是使用社区自研的基于 Raft 的共识算法， 替代ZooKeeper，实现Controller自选举**。

## 三十四、阐述下Kafka中的领导者副本(Leader Replica)和追随者副本 (Follower Replica)的区别

这道题表面上是考核你对Leader和Follower区别的理解，但很容易引申到Kafka的同步机制上。因此，我建议你主动出击，一次性地把隐含的考点也答出来，也许能够暂时把面试官“唬住”，并体现你的专业性。

你可以这么回答：**Kafka 副本当前分为领导者副本和追随者副本。只有 Leader 副本才能对外提供读写服务，响应Clients端的请求。Follower副本只是采用拉(PULL)的方式，被动地同步 Leader 副本中的数据，并且在 Leader 副本所在的 Broker 宕机后，随时 准备应聘 Leader 副本。**

通常来说，回答到这个程度，其实才只说了60%，因此，我建议你再回答两个额外的加分项。

* **强调Follower副本也能对外提供读服务**。自Kafka 2.4版本开始，社区通过引入新的Broker端参数，允许Follower副本有限度地提供读服务。
* **强调Leader和Follower的消息序列在实际场景中不一致**。很多原因都可能造成Leader和Follower保存的消息序列不一致，比如程序Bug、网络问题等。这是很严重的错误，必须要完全规避。你可以补充下，之前确保一致性的主要手段是高水位机制， 但高水位值无法保证Leader连续变更场景下的数据一致性，因此，社区引入了Leader Epoch机制，来修复高水位值的弊端。关于“Leader Epoch机制”，国内的资料不是很多，它的普及度远不如高水位，不妨大胆地把这个概念秀出来，力求惊艳一把。

## 三十五、如何设置Kafka能接收的最大消息的大小?

**这道题除了要回答消费者端的参数设置之外，一定要加上Broker端的设置，这样才算完整**。毕竟，如果Producer 都不能向Broker端发送数据很大的消息，又何来消费一说呢? 因此，你需要同时设置Broker端参数和Consumer端参数。

Broker端参数：message.max.bytes、max.message.bytes（主题级别）和replica.fetch.max.bytes。

Consumer端参数：fetch.message.max.bytes。

Broker端的最后一个参数比较容易遗漏。我们必须调整Follower副本能够接收的最大消息的大小，否则，副本同步就会失败。因此，把这个答出来的话，就是一个加分项。

## 三十六、监控Kafka的框架都有哪些?

面试官其实是在考察你对监控框架的了解广度，或者说，你是否知道很多能监控Kafka的框架或方法。下面这些就是Kafka发展历程上比较有名气的监控系统。

**Kafka Manager**：应该算是最有名的专属Kafka监控框架了，是独立的监控系统。

**Kafka Monitor**：LinkedIn开源的免费框架，支持对集群进行系统测试，并实时监控测试结果。

**CruiseControl**：也是LinkedIn公司开源的监控框架，用于实时监测资源使用率，以及提供常用运维操作等。无UI界面，只提供REST API。

**JMX 监控**：由于Kafka提供的监控指标都是基于JMX的，因此，市面上任何能够集成JMX的框架都可以使用，比如Zabbix和Prometheus。

**已有大数据平台自己的监控体系**：像Cloudera提供的CDH这类大数据平台，天然就提供Kafka监控方案。

**JMXTool**：社区提供的命令行工具，能够实时监控JMX指标。答上这一条，属于绝对 的加分项，因为知道的人很少，而且会给人一种你对Kafka工具非常熟悉的感觉。如果你暂时不了解它的用法，可以在命令行以无参数方式执行一下kafka-run-class.sh kafka.tools.JmxTool，学习下它的用法。

## 三十七、Broker的Heap Size如何设置?

如何设置Heap Size的问题，其实和Kafka关系不大，它是一类非常通用的面试题目。一旦你应对不当，面试方向很有可能被引到JVM和GC上去，那样的话，你被问住的几率就会增大。因此，我建议你简单地介绍一下Heap Size 的设置方法，并把重点放在Kafka Broker堆大小设置的最佳实践上。

比如，你可以这样回复：**任何Java进程JVM堆大小的设置都需要仔细地进行考量和测试。一个常见的做法是，以默认的初始JVM堆大小运行程序，当系统达到稳定状态后，手动触发一次Full GC，然后通过JVM工具查看GC后的存活对象大小。之后，将堆大小设置成存活对象总大小的1.5~2倍。对于Kafka而言，这个方法也是适用的。不过，业界有个最佳实践，那就是将Broker的Heap Size固定为 6GB。经过很多公司的验证，这个大小是足够且良好的。**

## 三十八、如何估算Kafka集群的机器数量?

这道题目考查的是**机器数量和所用资源之间的关联关系**。所谓资源，也就是CPU、内存、磁盘和带宽。

通常来说，CPU 和内存资源的充足是比较容易保证的，因此，你需要从磁盘空间和带宽占用两个维度去评估机器数量。

在预估磁盘的占用时，你一定不要忘记计算副本同步的开销。如果一条消息占用 1KB 的磁盘空间，那么，在有3 个副本的主题中，你就需要3KB的总空间来保存这条消息。显式地将这些考虑因素答出来，能够彰显你考虑问题的全面性，是一个难得的加分项。

对于评估带宽来说，常见的带宽有1Gbps和10Gbps，但你要切记，**这两个数字仅仅是最大值**。因此，你最好和面试官确认一下给定的带宽是多少。然后，明确阐述出当带宽占用接近总带宽的90%时，丢包情形就会发生。这样能显示出你的网络基本功。

## 三十九、Leader总是 -1，怎么破?

在生产环境中，你一定碰到过“某个主题分区不能工作了”的情形。使用命令行查看状态的话，会发现Leader是-1，于是，你使用各种命令都无济于事，最后只能用“重启大法”。

但是，有没有什么办法，可以不重启集群，就能解决此事呢？这就是此题的由来。

我直接给答案：**删除ZooKeeper节点 /controller，触发Controller重选举。 Controller重选举能够为所有主题分区重刷分区状态，可以有效解决因不一致导致的Leader不可用问题**。我几乎可以断定，当面试官问出此题时，要么就是他真的不知道怎么解决在向你寻求答案，要么他就是在等你说出这个答案。所以，千万别一上来就说“来个重 启”之类的话。

## 四十、LEO、LSO、AR、ISR、HW 都表示什么含义?

* **LEO**：Log End Offset。日志末端位移值或末端偏移量，表示日志下一条待插入消息的位移值。举个例子，如果日志有10条消息，位移值从0开始，那么，第10条消息的位移值就是9。此时，LEO = 10。
* **LSO**：Log Stable Offset。这是Kafka事务的概念。如果你没有使用到事务，那么这个值不存在（其实也不是不存在，只是设置成一个无意义的值）。该值控制了事务型消费者能够看到的消息范围。它经常与Log Start Offset，即日志起始位移值相混淆，因为有些人将后者缩写成LSO，这是不对的。在Kafka中，LSO 就是指代 Log Stable Offset。
* **AR**：Assigned Replicas。AR是主题被创建后，分区创建时被分配的副本集合，副本个数由副本因子决定。
* **ISR**：In-Sync Replicas。Kafka 中特别重要的概念，指代的是 AR 中那些与 Leader 保 持同步的副本集合。在AR中的副本可能不在ISR中，但Leader副本天然就包含在ISR中。关于ISR，**还有一个常见的面试题目是如何判断副本是否应该属于ISR**。目前的判断依据是:**Follower副本的LEO落后Leader LEO的时间，是否超过了Broker端参数 replica.lag.time.max.ms 值**。如果超过了，副本就会被从ISR中移除。
* **HW**：高水位值（High watermark）。这是控制消费者可读取消息范围的重要字段。一个普通消费者只能“看到”Leader副本上介于Log Start Offset和HW(不含)之间的所有消息。水位以上的消息是对消费者不可见的。关于HW，问法有很多，我能想到的最高级的问法，就是让你完整地梳理下Follower副本拉取Leader 副本、执行同步机制的详细步骤（详细步骤见四十八题）。

## 四十一、Kafka 能手动删除消息吗?

其实，Kafka不需要用户手动删除消息。它本身提供了留存策略，能够自动删除过期消息。 当然，它是支持手动删除消息的。因此，你最好从这两个维度去回答。

* 对于设置了Key且参数 cleanup.policy=compact 的主题而言，我们可以构造一条<Key，null>的消息发送给Broker，依靠Log Cleaner组件提供的功能删除掉该Key的消息。
* 对于普通主题而言，我们可以使用kafka-delete-records命令，或编写程序调用Admin.deleteRecords方法来删除消息。这两种方法殊途同归，底层都是调用Admin的deleteRecords方法，通过将分区Log Start Offset值抬高的方式间接删除消息。

## 四十二、\_\_consumer\_offsets是做什么用的?

这是一个内部主题，公开的官网资料很少涉及到。因此，我认为，此题属于面试官炫技一类的题目。你要小心这里的考点：该主题有3个重要的知识点，你一定要全部答出来，才会显得对这块知识非常熟悉。

它是一个内部主题，无需手动干预，由Kafka自行管理。当然，我们可以创建该主题。

它的主要作用是负责注册消费者以及保存位移值。可能你对保存位移值的功能很熟悉，但其实**该主题也是保存消费者元数据的地方。千万记得把这一点也回答上**。另外，这里的消费者泛指消费者组和独立消费者，而不仅仅是消费者组。

Kafka的GroupCoordinator组件提供对该主题完整的管理功能，包括该主题的创建、写入、读取和Leader维护等。

## 四十三、分区Leader选举策略有几种?

分区的Leader副本选举对用户是完全透明的，它是由Controller独立完成的。你需要回答的是，在哪些场景下，需要执行分区Leader选举。每一种场景对应于一种选举策略。当前，Kafka有4种分区Leader选举策略。

* **OfflinePartition Leader选举**：每当有分区上线时，就需要执行Leader选举。所谓的分区上线，可能是创建了新分区，也可能是之前的下线分区重新上线。这是最常见的分区Leader选举场景。
* **ReassignPartition Leader选举**：当你手动运行kafka-reassign-partitions命令，或者是调用Admin的 alterPartitionReassignments方法执行分区副本重分配时，可能触发此类选举。假设原来的AR是[1，2，3]，Leader是 1，当执行副本重分配后，副本集合AR被设置成[4，5，6]，显然，Leader必须要变更，此时会发生Reassign Partition Leader选举。
* **PreferredReplicaPartition Leader选举**：当你手动运行kafka-preferred-replica- election命令，或自动触发了Preferred Leader选举时，该类策略被激活。所谓的Preferred Leader，指的是AR中的第一个副本。比如AR是[3，2，1]，那么， Preferred Leader就是3。
* **ControlledShutdownPartition Leader选举**：当Broker正常关闭时，该Broker上的所有Leader副本都会下线，因此，需要为受影响的分区执行相应的Leader选举。

这4类选举策略的大致思想是类似的，即从AR中挑选首个在ISR中的副本，作为新Leader。当然，个别策略有些微小差异。不过，回答到这种程度，应该足以应付面试官了。毕竟，微小差别对选举Leader这件事的影响很小。

## 四十四、Kafka的哪些场景中使用了零拷贝(Zero Copy)?

Zero Copy 是特别容易被问到的高阶题目。在Kafka中，体现Zero Copy使用场景的地方有两处：**基于mmap的索引和日志文件读写所用的TransportLayer**。

先说第一个。索引都是基于MappedByteBuffer的，也就是让用户态和内核态共享内核态的数据缓冲区，此时，数据不需要复制到用户态空间。不过，mmap虽然避免了不必要的拷贝，但不一定就能保证很高的性能。在不同的操作系统下，mmap的创建和销毁成本可能是不一样的。很高的创建和销毁开销会抵消Zero Copy带来的性能优势。由于这种不确定性，在Kafka中，只有索引应用了mmap，最核心的日志并未使用mmap机制。

再说第二个。TransportLayer是Kafka传输层的接口。它的某个实现类使用了FileChannel的transferTo方法。该方法底层使用sendfile实现了Zero Copy。对Kafka而言，如果I/O通道使用普通的PLAINTEXT，那么，Kafka就可以利用Zero Copy特性，直接将页缓存中的数据发送到网卡的Buffer中，避免中间的多次拷贝。相反，如果I/O通道启用了SSL，那么，Kafka便无法利用Zero Copy特性了。

## 四十五、如何调优Kafka?

回答任何调优问题的第一步，就是**确定优化目标，并且定量给出目标！**这点特别重要。对于Kafka而言，常见的优化目标是吞吐量、延时、持久性和可用性。每一个方向的优化思路都是不同的，甚至是相反的。

确定了目标之后，还要明确优化的维度。有些调优属于通用的优化思路，比如对操作系统、JVM等的优化；有些则是有针对性的，比如要优化Kafka的TPS。我们需要从3个方向去考虑

* **Producer 端**：增加 batch.size、linger.ms，启用压缩，关闭重试等。
* **Broker 端**：增加 num.replica.fetchers，提升Follower同步TPS，避免Broker Full GC等。
* **Consumer**：增加 fetch.min.bytes等

## 四十六、Controller发生网络分区(Network Partitioning)时，Kafka会怎 样?

这道题目能够诱发我们对分布式系统设计、CAP理论、一致性等多方面的思考。不过，针对故障定位和分析的这类问题，我建议你首先言明“实用至上”的观点，即不论怎么进行理论分析，永远都要以实际结果为准。一旦发生 Controller网络分区，那么，第一要务就是查看集群是否出现“脑裂”，即同时出现两个甚至是多个Controller组件。这可以根据Broker端监控指标ActiveControllerCount来判断。

现在，我们分析下，一旦出现这种情况，Kafka会怎么样。

由于Controller会给Broker发送3类请求，LeaderAndIsrRequest、 StopReplicaRequest和UpdateMetadataRequest，因此，一旦出现网络分区，这些请求将不能顺利到达Broker端。这将影响主题的创建、修改、删除操作的信息同步，表现为集群仿佛僵住了一样，无法感知到后面的所有操作。因此，网络分区通常都是非常严重的问题，要赶快修复。

## 四十七、Java Consumer 为什么采用单线程来获取消息?

在回答之前，如果先把这句话说出来，一定会加分：**Java Consumer是双线程的设计。一个线程是用户主线程，负责获取消息；另一个线程是心跳线程，负责向Kafka汇报消费者存活情况。将心跳单独放入专属的线程，能够有效地规避因消息处理速度慢而被视为下线的“假死”情况。**

单线程获取消息的设计能够避免阻塞式的消息获取方式。单线程轮询方式容易实现异步非阻塞式，这样便于将消费者扩展成支持实时流处理的操作算子。因为很多实时流处理操作算子都不能是阻塞式的。另外一个可能的好处是，可以简化代码的开发。多线程交互的代码是非常容易出错的。

## 四十八、简述 Follower 副本消息同步的完整流程

首先，Follower发送FETCH请求给Leader。接着，Leader会读取底层日志文件中的消息数据，再更新它内存中的 Follower副本的LEO值，更新为FETCH请求中的fetchOffset值。最后，尝试更新分区高水位值。Follower接收到FETCH响应之后，会把消息写入到底层日志，接着更新LEO和HW值。

Leader和Follower的HW值更新时机是不同的，Follower的HW更新永远落后于Leader的HW。这种时间上的错配是造成各种不一致的原因。

## 四十九、数据传输的事物定义有哪三种？

数据传输的事务定义通常有以下三种级别：

（1）最多一次: 消息不会被重复发送，最多被传输一次，但也有可能一次不传输；

（2）最少一次: 消息不会被漏发送，最少被传输一次，但也有可能被重复传输；

（3）精确的一次（Exactly once）: 不会漏传输也不会重复传输,每个消息都传输被一次而且仅仅被传输一次，这是大家所期望的。

## 五十、Kafka判断一个节点是否还活着有那两个条件？

（1）节点必须可以维护和ZooKeeper的连接，Zookeeper通过心跳机制检查每个节点的连接；

（2）如果节点是个follower，他必须能及时的同步leader的写操作，延时不能太久。

## 五十一、producer是否直接将数据发送到broker的leader(主节点)？

producer直接将数据发送到broker的leader(主节点)，不需要在多个节点进行分发，为了帮助 producer 做到这点，所有的Kafka节点都可以及时的告知:哪些节点是活动的，目标topic目标分区的leader在哪。这样producer就可以直接将消息发送到目的地了。

## 五十二、Kafa consumer是否可以消费指定分区消息？

Kafa consumer消费消息时，向broker发出"fetch"请求去消费特定分区的消息，consumer指定消息在日志中的偏移量（offset），就可以消费从这个位置开始的消息，customer 拥有了offset的控制权，可以向后回滚去重新消费之前的消息，这是很有意义的。

## 五十三、Kafka消息是采用Pull模式，还是Push模式？

Kafka最初考虑的问题是，customer应该从brokes拉取消息还是brokers将消息推送到consumer，也就是pull还 push。在这方面，Kafka遵循了一种大部分消息系统共同的传统的设计：producer将消息推送到broker，consumer 从broker拉取消息一些消息系统比如Scribe和Apache Flume采用了push模式，将消息推送到下游的consumer。这样做有好处也有坏处：由broker决定消息推送的速率，对于不同消费速率的consumer就不太好处理了。消息系统都致力于让consumer以最大的速率最快速的消费消息，但不幸的是，push模式下，当broker推送的速率远大于 consumer消费的速率时，consumer恐怕就要崩溃了。最终Kafka还是选取了传统的pull模式，Pull模式的另外一个好处是consumer可以自主决定是否批量的从broker拉取数据。Push模式必须在不知道下游consumer消费能力和消费策略的情况下决定是立即推送每条消息还是缓存之后批量推送。如果为了避免consumer崩溃而采用较低的推送速率，将可能导致一次只推送较少的消息而造成浪费。Pull模式下，consumer就可以根据自己的消费能力去决定这些策略Pull有个缺点是，如果broker没有可供消费的消息，将导致consumer不断在循环中轮询，直到新消息到t达。为了避免这点，Kafka有个参数可以让consumer阻塞知道新消息到达（当然也可以阻塞知道消息的数量达到某个特定的量这样就可以批量发）。

## 五十四、Kafka存储在硬盘上的消息格式是什么？

消息由一个固定长度的头部和可变长度的字节数组组成。

头部包含了一个版本号和CRC32校验码。

消息长度: 4 bytes (value: 1+4+n)

版本号: 1 byte

CRC校验码: 4 bytes

具体的消息: n bytes

## 五十五、Kafka高效文件存储设计特点

1.Kafka把topic中一个parition大文件分成多个小文件段，通过多个小文件段，就容易定期清除或删除已经消费完文件，减少磁盘占用。

2.通过索引信息可以快速定位message和确定response的最大大小。

3.通过index元数据全部映射到memory，可以避免segment file的IO磁盘操作。

4.通过索引文件稀疏存储，可以大幅降低index文件元数据占用空间大小。

## 五十六、Kafka与传统消息系统之间有三个关键区别

1.Kafka持久化日志，这些日志可以被重复读取和无限期保留

2.Kafka是一个分布式系统：它以集群的方式运行，可以灵活伸缩，在内部通过复制数据提升容错能力和高可用性

3.Kafka支持实时的流式处理

## 五十七、Kafka创建Topic时如何将分区放置到不同的Broker中

副本因子不能大于Broker的个数；

第一个分区（编号为 0）的第一个副本放置位置是随机从brokerList选择的；

其他分区的第一个副本放置位置相对于第0个分区依次往后移。也就是如果我们有5个Broker，5个分区，假设第一个分区放在第四个Broker上，那么第二个分区将会放在第五个Broker上；第三个分区将会放在第一个Broker上；第四个分区将会放在第二个Broker上，依次类推；剩余的副本相对于第一个副本放置位置其实是由 nextReplicaShift 决定的，而这个数也是随机产生的。

## 五十八、Kafka新建的分区会在哪个目录下创建

在启动Kafka集群之前，我们需要配置好log.dirs参数，其值是Kafka数据的存放目录，这个参数可以配置多个目录，目录之间使用逗号分隔，通常这些目录是分布在不同的磁盘上用于提高读写性能。

当然我们也可以配置log.dir参数，含义一样。只需要设置其中一个即可。如果log.dirs参数只配置了一个目录，那么分配到各个Broker上的分区肯定只能在这个目录下创建文件夹用于存放数据。但是如果log.dirs参数配置了多个目录，那么 Kafka 会在哪个文件夹中创建分区目录呢？

答案是：Kafka会在含有分区目录最少的文件夹中创建新的分区目录，分区目录名为 Topic

名+分区ID。注意，是分区文件夹总数最少的目录，而不是磁盘使用量最少的目录！也就是说，如果你给log.dirs参数新增了一个新的磁盘，新的分区目录肯定是先在这个新的磁盘上创建直到这个新的磁盘目录拥有的分区目录不是最少为止。

## 五十九、partition的数据如何保存到硬盘

topic 中的多个partition以文件夹的形式保存到broker，每个分区序号从0递增，且消息有序Partition文件下有多个segment（xxx.index，xxx.log）segment文件里的大小和配置文件大小一致可以根据要求修改 默认为1g如果大小大于1g时，会滚动一个新的segment并且以上一个segment最后一条消息的偏移量命名。

## 六十、kafka的ack机制

request.required.acks 有三个值 0 1 -1

* 0：生产者不会等待broker的ack，这个延迟最低但是存储的保证最弱当server挂掉的时候就会丢数据；
* 1：服务端会等待ack值leader副本确认接收到消息后发送 ack 但是如果leader挂掉后他不确保是否复制完成新leader也会导致数据丢失；
* -1：同样在1的基础上 服务端会等所有的follower的副本受到数据后才会受到leader发出的ack，这样数据不会丢失。

## 六十一、Kafka的消费者如何消费数据

消费者每次消费数据的时候，消费者都会记录消费的物理偏移量（offset）的位置等到下次消费时，他会接着上次位置继续消费。

## 六十二、消费者负载均衡策略

一个消费者组中的一个分片对应一个消费者成员，他能保证每个消费者成员都能访问，如果组中成员太多会有空闲的成员。

## 六十三、数据有序

一个消费者组里它的内部是有序的；

消费者组与消费者组之间是无序的。

## 六十四、kafaka生产数据时数据的分组策略

生产者决定数据产生到集群的哪个partition中

每一条消息都是以（key，value）格式

Key是由生产者发送数据传入

所以生产者（key）决定了数据产生到集群的哪个partition