kafka 可以脱离 zookeeper 单独使用吗？为什么？

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 有几种数据保留的策略？

kafka 有两种数据保存策略：按照过期时间保留和按照存储的消息大小保留。

kafka 同时设置了 7 天和 10G 清除数据，到第五天的时候消息达到了 10G，这个时候 kafka 将如何处理？

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。

什么情况会导致 kafka 运行变慢？

cpu

性能瓶颈

磁盘读写瓶颈

网络瓶颈

使用 kafka 集群需要注意什么？

集群的数量不是越多越好，最好不要超过 7 个，因为节点越多，消息复制需要的时间就越长，整个群组的吞吐量就越低。 集群数量最好是单数，因为超过一半故障集群就不能用了，设置为单数容错率更高。

什么是kafka？

Kafka是分布式发布-订阅消息系统，它最初是由LinkedIn公司开发的，之后成为Apache项目的一部分，Kafka是一个分布式，可划分的，冗余备份的持久性的日志服务，它主要用于处理流式数据。

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。

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。

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。

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。

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 。

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。

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。

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。

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。

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。

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

kafka中consumer group 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）。

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。

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。

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.

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

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。

Kafka中的事务是怎么实现的?

事务，对于大家来说可能并不陌生，比如数据库事务、分布式事务，那么Kafka中的事务是什么样子的呢?

在说Kafka的事务之前，先要说一下Kafka中幂等的实现。幂等和事务是Kafka 0.11.0.0版本引入的两个特性，以此来实现EOS（exactly once semantics，精确一次处理语义）。

幂等，简单地说就是对接口的多次调用所产生的结果和调用一次是一致的。生产者在进行重试的时候有可能会重复写入消息，而使用Kafka的幂等性功能之后就可以避免这种情况。

开启幂等性功能的方式很简单，只需要显式地将生产者客户端参数enable.idempotence设置为true即可（这个参数的默认值为false）。

Kafka是如何具体实现幂等的呢？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对应的序列号的值加1。

broker端会在内存中为每一对维护一个序列号。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，这个异常是一个严重的异常。

引入序列号来实现幂等也只是针对每一对而言的，也就是说，Kafka的幂等只能保证单个生产者会话（session）中单分区的幂等。幂等性不能跨多个分区运作，而事务可以弥补这个缺陷。

事务可以保证对多个分区写入操作的原子性。操作的原子性是指多个操作要么全部成功，要么全部失败，不存在部分成功、部分失败的可能。

为了使用事务，应用程序必须提供唯一的transactionalId，这个transactionalId通过客户端参数transactional.id来显式设置。事务要求生产者开启幂等特性，因此通过将transactional.id参数设置为非空从而开启事务特性的同时需要将enable.idempotence设置为true（如果未显式设置，则KafkaProducer默认会将它的值设置为true），如果用户显式地将enable.idempotence设置为false，则会报出ConfigException的异常。

transactionalId与PID一一对应，两者之间所不同的是transactionalId由用户显式设置，而PID是由Kafka内部分配的。

另外，为了保证新的生产者启动后具有相同transactionalId的旧生产者能够立即失效，每个生产者通过transactionalId获取PID的同时，还会获取一个单调递增的producer epoch。如果使用同一个transactionalId开启两个生产者，那么前一个开启的生产者会报错。

从生产者的角度分析，通过事务，Kafka可以保证跨生产者会话的消息幂等发送，以及跨生产者会话的事务恢复。

前者表示具有相同transactionalId的新生产者实例被创建且工作的时候，旧的且拥有相同transactionalId的生产者实例将不再工作。

后者指当某个生产者实例宕机后，新的生产者实例可以保证任何未完成的旧事务要么被提交（Commit），要么被中止（Abort），如此可以使新的生产者实例从一个正常的状态开始工作。

KafkaProducer提供了5个与事务相关的方法，详细如下：

```
1  void initTransactions();
2  void beginTransaction() throws ProducerFencedException;
3  void sendOffsetsToTransaction(Map<TopicPartition, OffsetAndMetadata>
   offsets,String consumerGroupId)
4  throws ProducerFencedException;
5  void commitTransaction() throws ProducerFencedException;
6  void abortTransaction() throws ProducerFencedException;
```

initTransactions()方法用来初始化事务；beginTransaction()方法用来开启事务；sendOffsetsToTransaction()方法为消费者提供在事务内的位移提交的操作；commitTransaction()方法用来提交事务；abortTransaction()方法用来中止事务，类似于事务回滚。

initTransactions()方法用来初始化事务；beginTransaction()方法用来开启事务；sendOffsetsToTransaction()方法为消费者提供在事务内的位移提交的操作；commitTransaction()方法用来提交事务；abortTransaction()方法用来中止事务，类似于事务回滚。

在消费端有一个参数isolation.level，与事务有着莫大的关联，这个参数的默认值为“read_uncommitted”，意思是说消费端应用可以看到（消费到）未提交的事务，当然对于已提交的事务也是可见的。

这个参数还可以设置为“read_committed”，表示消费端应用不可以看到尚未提交的事务内的消息。

举个例子，如果生产者开启事务并向某个分区值发送3条消息msg1、msg2和msg3，在执行commitTransaction()或abortTransaction()方法前，设置为“read_committed”的消费端应用是消费不到这些消息的，不过在KafkaConsumer内部会缓存这些消息，直到生产者执行commitTransaction()方法之后它才能将这些消息推送给消费端应用。

反之，如果生产者执行了abortTransaction()方法，那么KafkaConsumer会将这些缓存的消息丢弃而不推送给消费端应用。

Kafka中有那些地方需要选举？这些地方的选举策略又有哪些?

kafka在所有broker中选出一个controller，所有Partition的Leader选举都由controller决定。controller会将Leader的改变直接通过RPC的方式（比Zookeeper Queue的方式更高效）通知需为此作出响应的Broker。同时controller也负责增删Topic以及Replica的重新分配。 当有broker fari over controller的处理过程如下：

1.Controller在Zookeeper注册Watch，一旦有Broker宕机（这是用宕机代表任何让系统认为其die的情景，包括但不限于机器断电，网络不可用，GC导致的Stop The World，进程crash等），其在Zookeeper对应的znode会自动被删除，Zookeeper会fire Controller注册的watch，Controller读取最新的幸存的Broker

2.Controller决定set_p，该集合包含了宕机的所有Broker上的所有Partition

3.对set_p中的每一个Partition

3.1 从/brokers/topics/[topic]/partitions/[partition]/state读取该Partition当前的ISR

3.2 决定该Partition的新Leader。如果当前ISR中有至少一个Replica还幸存，则选择其中一个作为新Leader，新的ISR则包含当前ISR中所有幸存的Replica（选举算法的实现类似于微软的PacificA）。否则选择该Partition中任意一个幸存的Replica作为新的Leader以及ISR（该场景下可能会有潜在的数据丢失）。如果该Partition的所有Replica都宕机了，则将新的Leader设置为-1。

3.3 将新的Leader，ISR和新的leader_epoch及controller_epoch写入/brokers/topics/[topic]/partitions/[partition]/state。注意，该操作只有其version在3.1至3.3的过程中无变化时才会执行，否则跳转到3.1

4.直接通过RPC向set_p相关的Broker发送LeaderAndISRRequest命令。Controller可以在一个RPC操作中发送多个命令从而提高效率。

如何处理所有Replica都不工作

上文提到，在ISR中至少有一个follower时，Kafka可以确保已经commit的数据不丢失，但如果某个Partition的所有Replica都宕机了，就无法保证数据不丢失了。这种情况下有两种可行的方案：

1.等待ISR中的任一个Replica“活”过来，并且选它作为Leader

2.选择第一个“活”过来的Replica（不一定是ISR中的）作为Leader

这就需要在可用性和一致性当中作出一个简单的折衷。如果一定要等待ISR中的Replica“活”过来，那不可用的时间就可能会相对较长。

而且如果ISR中的所有Replica都无法“活”过来了，或者数据都丢失了，这个Partition将永远不可用选择第一个“活”过来的Replica作为Leader，而这个Replica不是ISR中的Replica，那即使它并不保证已经包含了所有已commit的消息，它也会成为Leader而作为consumer的数据源（前文有说明，所有读写都由Leader完成）。

Kafka0.8.* 使用了第二种方式。根据Kafka的文档，在以后的版本中，Kafka支持用户通过配置选择这两种方式中的一种，从而根据不同的使用场景选择高可用性还是强一致性。

unclean.leader.election.enable 参数决定使用哪种方案，默认是true，采用第二种方案

Kafka 的设计是什么样的呢?

Kafka 将消息以 topic 为单位进行归纳

将向 Kafka topic 发布消息的程序成为 producers.

将预订 topics 并消费消息的程序成为 consumer.

Kafka 以集群的方式运行，可以由一个或多个服务组成，每个服务叫做一个 broker.

producers 通过网络将消息发送到 Kafka 集群，集群向消费者提供消息

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

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

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

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

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送)

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

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。

kafka 文件存储基本结构

在 Kafka 文件存储中，同一个 Topic 下有多个不同 partition，每个 partition 为一个目录。partition命名规则为Topic名称 + 有序序号。如果 partition 数量为 num，则第一个 partition 序号从 0 开始，序号最大值为 num - 1。

例如，自己创建一个名为 orderMq 的 Topic：

```
[root@mini1 bin]# ./kafka-topics.sh --create --zookeeper mini1:2181 --replication-factor 2 --partitions 3 --topic orderMq
Created topic "orderMq".
```

orderMq 这个 topic 对应的 partitions 在三台机器上名称分别为：

```
drwxr-xr-x. 2 root root 4096 11月 21 22:25 orderMq-0
drwxr-xr-x. 2 root root 4096 11月 21 22:25 orderMq-2
```

```
drwxr-xr-x. 2 root root 4096 11月 14 18:45 orderMq-1
drwxr-xr-x. 2 root root 4096 11月 14 18:45 orderMq-2
```

```
drwxr-xr-x. 2 root root 4096 11月 21 22:25 orderMq-0
drwxr-xr-x. 2 root root 4096 11月 21 22:25 orderMq-1
```

注：重复的是副本，partition 名分别为 orderMq-0, orderMq-1, orderMq-2；

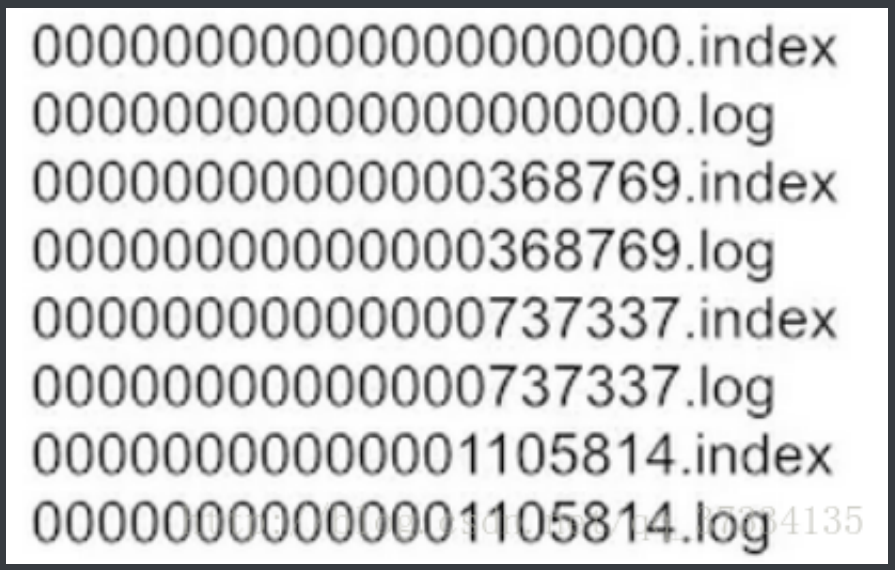
每个 partition（即每个目录）相当于一个巨型文件被平均分配到多个大小相等的 **segment（段）** 数据文件中，但每个 segment 消息数量不一定相等。这种特性方便旧 segment 文件快速被删除，默认保留7天的数据。例如在 orderMq-0 目录下：

```
[root@mini3 orderMq-0]# ll
-rw-r--r--. 1 root root 10485760 11月 21 22:31 00000000000000000000.index
-rw-r--r--. 1 root root      219 11月 22 05:22 00000000000000000000.log
```

由上可知，index 和 log 为后缀名的文件的合称，就是 segment 文件。每个 partition 只需要支持顺序读写就行了，segment 文件生命周期（什么时候创建，什么时候删除）由服务端配置参数决定。

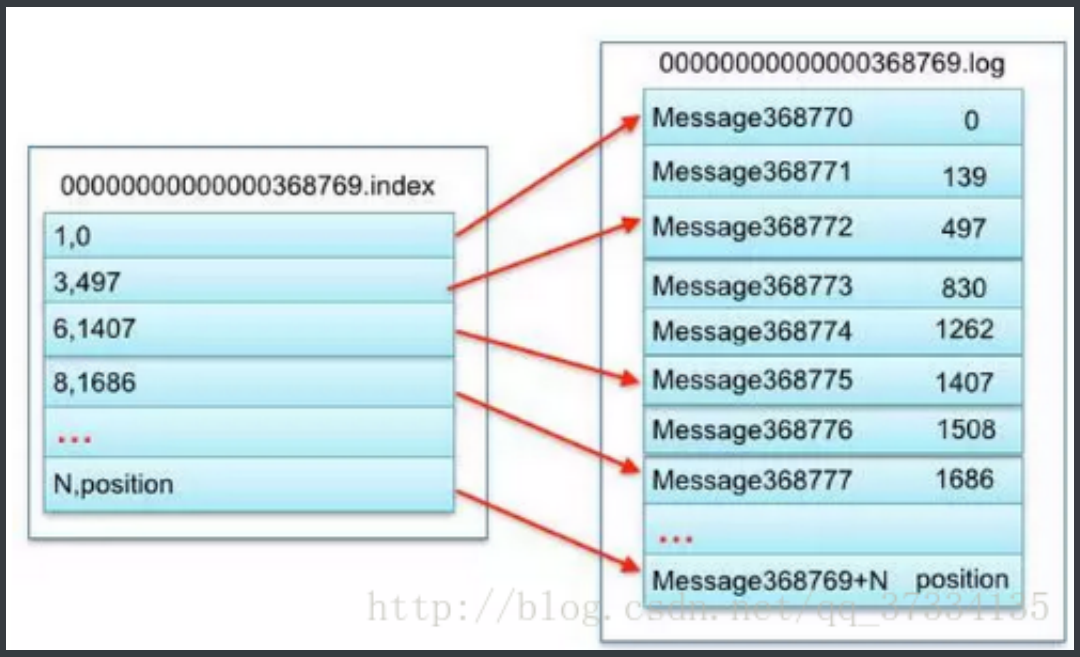
segment 文件组成

Segment 文件由两大部分组成，分别为索引文件 (index file) 和数据文件 (data file)，这两个文件一一对应，成对出现。如下图所示：



Segment 文件命名规则：partition 全局的第一个 segment 从 0 开始，后续每个 segment 文件名为上一个 segment 文件最后一条消息的 offset 值。数值最大为 64 位 long 大小，19 位数字字符长度，没有数字用 0 填充。

索引文件存储大量元数据，数据文件存储大量消息。**索引文件中元数据指向对应数据文件中message的物理偏移地址。** 如下图所示：



kafka 查找消息

读取 offset=368776 的消息，需要通过下面两个步骤查找。

第一步：查找segment file

以起始偏移量命名并排序这些文件，只要根据offset 二分查找文件列表，就可以快速定位到具体文件。

- 00000000000000000000.index 表示最开始的文件，起始偏移量 (offset) 为 0
- 00000000000000368769.index 的消息量起始偏移量为 368770 = 368769 + 1
- 00000000000000737337.index 的起始偏移量为 737338=737337 + 1

其他后续文件依次类推。最后 offset=368776 时定位到 00000000000000368769.index 和对应log文件。

第二步：通过 Segment 查找消息

- 当 offset=368776 时，依次定位到 00000000000000368769.index 的元数据物理位置和 00000000000000368769.log 的物理偏移地址，然后再通过 00000000000000368769.log 顺序查找直到 offset=368776 为止。

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

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

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

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

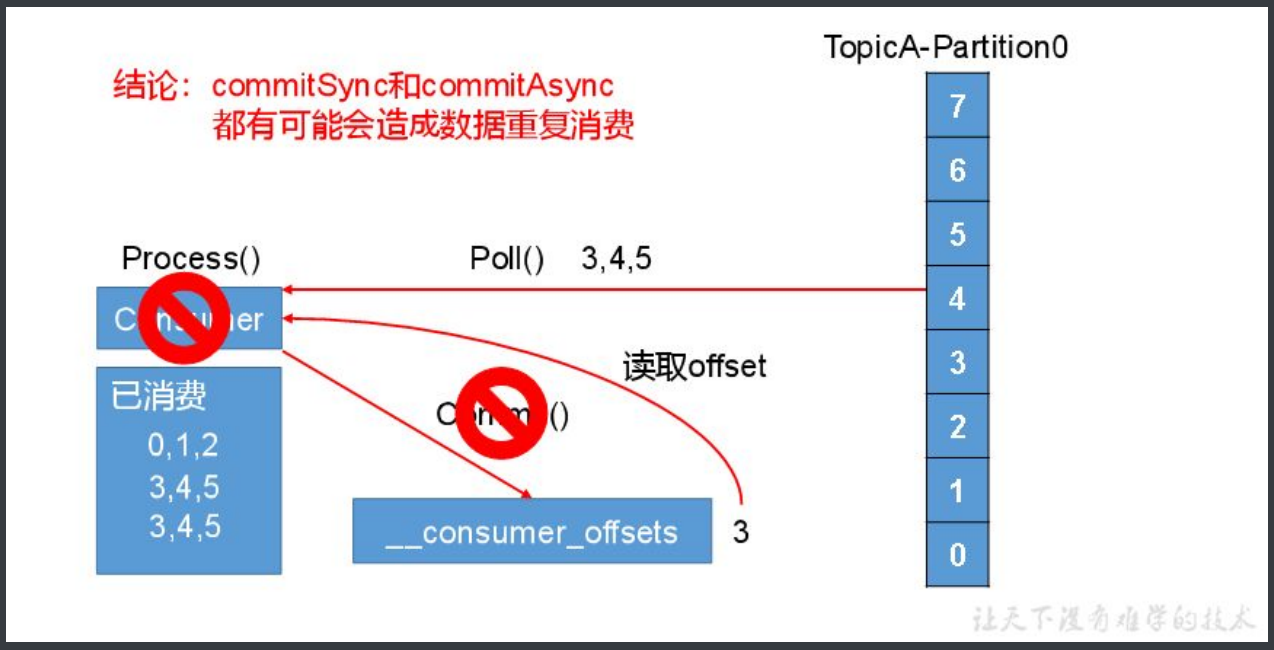
不确保是否复制完成新 leader 也会导致数据丢失

-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

Kafka 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

有哪些情形会造成重复消费?



commitSync 和 commitAsync 都有可能造成数据重复消费

消费者消费后没有commit offset(程序崩溃/强行kill/消费耗时/自动提交偏移情况下unscrible)

kafka高可用原理?

broker启动会尝试向zookeeper创建临时节点：/controller，第一个broker选举成功成为集群的controller，其余节点都会在/controller注册watcher监控controller状态；当controller挂掉，所有broker感知到，重新尝试选举controller

controller节点通过zookeeper监控各broker状态，如果由broker挂掉，controller负责从其负责的leader分区的isr列表中选举一个作为新的leader

kafka副本和leader选举

kafka高性能原因是什么?

零拷贝、利用操作系统页缓存、磁盘顺序写 kafka零拷贝原理 分区、分段、建立索引、生产者、消费者批处理

Kafka中的HW、LEO、LSO、LW等分别代表什么?

HW：High Watermark 高水位，取一个partition对应的ISR中最小的LEO作为HW，consumer最多只能消费到HW所在的位置上一条信息

LEO：LogEndOffset 当前日志文件中下一条待写信息的offset

HW/LEO：这两个都是指最后一条的下一条的位置而不是指最后一条的位置

LSO：Last Stable Offset 对未完成的事务而言，LSO 的值等于事务中第一条消息的位置(firstUnstableOffset)，对已完成的事务而言，它的值同 HW 相同

LW：Low Watermark 低水位, 代表 AR 集合中最小的 logStartOffset 值

kafka如何保证不丢失消息?

- 复制因子：创建topic的时候指定复制因子大于1时，一个分区被分配到一个broker上，同时会在其他broker上维护一个分区副本；
- isr列表：分区及其副本分别为leader和follower，leader对外提供读写服务，follower会向leader发送同步请求，拉取最新的数据，如果follower和leader的消息差距保持在一定范围之内，那么这个follower在isr列表内；当分区leader所在broker宕机，会从isr列表中选举一个follower作为新的leader提供服务
- 通过kafka的acks参数可以控制消息的发送行为，acks可选的值有0、1、all；当设置为0时，生产者消息发送成功即为成功，不关心是否写入到磁盘及后续操作；当设置为1时，消息发送到分区leader后写入磁盘即为成功；当设置为all时，消息发送到分区leader并写入磁盘后，同步给isr列表中的所有分区副本后即为成功

Kafka生产者客户端中使用了几个线程来处理？分别是什么?

2个，

主线程和Sender线程。主线程负责创建消息，然后通过分区器、序列化器、拦截器作用之后缓存到累加器RecordAccumulator中。

Sender线程负责将RecordAccumulator中消息发送到kafka中.

有哪些情形会造成重复消费?

消费者消费后没有commit offset(程序崩溃/强行kill/消费耗时/自动提交偏移情况下unscrible)

如果我指定了一个offset，Kafka怎么查找到对应的消息?

1.通过文件名前缀数字x找到该绝对offset 对应消息所在文件

2.offset-x为在文件中的相对偏移

3.通过index文件中记录的索引找到最近的消息的位置

4.从最近位置开始逐条寻找

Kafka 消费者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