第三版：kafka 35 道

解释下 Kafka 中位移（offset）的作用

标准答案：在 Kafka 中，每个主题分区下的每条消息都被赋予了一个唯一的 ID 数值，用于标识它在分区中的位置。这个 ID 数值，就被称为位移，或者叫偏移量。一旦消息被写入到分区日志，它的位移值将不能被修改。

答完这些之后，你还可以把整个面试方向转移到你希望的地方：

1、 如果你深谙 Broker 底层日志写入的逻辑，可以强调下消息在日志中的存放格式

2、 如果你明白位移值一旦被确定不能修改，可以强调下“Log Cleaner 组件都不能影响位移值”这件事情

3、 如果你对消费者的概念还算熟悉，可以再详细说说位移值和消费者位移值之间的区别

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

推荐的答案：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。

加分点：

1、 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。

2、 强调 Leader 和 Follower 的消息序列在实际场景中不一致。通常情况下，很多因素可能造成 Leader 和 Follower 之间的不同步，比如程序问题，网络问题，broker 问题等，短暂的不同步我们可以关注（秒级别），但长时间的不同步可能就需要深入排查了，因为一旦 Leader 所在节点异常，可能直接影响可用性。

注意：之前确保一致性的主要手段是高水位机制（HW），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。

如何设置 Kafka 能接收的最大消息的大小?

对于 SRE 来讲，该题简直是送分题啊，但是，最大消息的设置通常情况下有生产者端，消费者端，broker 端和 topic 级别的参数，我们需要正确设置，以保证可以正常的生产和消费。

Broker 端参数：message.max.bytes，max.message.bytes（topic 级别），replica.fetch.max.bytes（否则 follow 会同步失败）

Consumer 端参数：fetch.message.max.bytes

监控 Kafka 的框架都有哪些?

对于 SRE 来讲，依然是送分题。但基础的我们要知道，Kafka 本身是提供了 JMX（Java Management Extensions）的，我们可以通过它来获取到 Kafka 内部的一些基本数据。

1、 Kafka Manager：更多是 Kafka 的管理，对于 SRE 非常友好，也提供了简单的瞬时指标监控。

2、 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。

3、 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，可以进行多集群管理。

4、 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。

5、 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。

6、 JMXTool：社区提供的命令行工具，能够实时监控 JMX 指标。可以使用 Kafka-run-class.sh Kafka.tools.JmxTool 来查看具体的用法。

Broker 的 Heap Size 如何设置?

1、 其实对于 SRE 还是送分题，因为目前来讲大部分公司的业务系统都是使用 Java 开发，因此 SRE 对于基本的 JVM 相关的参数应该至少都是非常了解的，核心就在于 JVM 的配置以及 GC 相关的知识。

2、 标准答案：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。

如何估算 Kafka 集群的机器数量?

1、 该题也算是 SRE 的送分题吧，对于 SRE 来讲，任何生产的系统第一步需要做的就是容量预估以及集群的架构规划，实际上也就是机器数量和所用资源之间的关联关系，资源通常来讲就是 CPU，内存，磁盘容量，带宽。但需要注意的是，Kafka 因为独有的设计，对于磁盘的要求并不是特别高，普通机械硬盘足够，而通常的瓶颈会出现在带宽上。

2、 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。同时，需要考虑到整个业务 Topic 数据保存的最大时间，以上几个因素，基本可以预估出来磁盘的容量需求。

3、 需要注意的是：对于磁盘来讲，一定要提前和业务沟通好场景，而不是等待真正有磁盘容量瓶颈了才去扩容磁盘或者找业务方沟通方案。

4、 对于带宽来说，常见的带宽有 1Gbps 和 10Gbps，通常我们需要知道，当带宽占用接近总带宽的 90%时，丢包情形就会发生。

Leader 总是-1，怎么破?

1、 对于有经验的 SRE 来讲，早期的 Kafka 版本应该多多少少都遇到过该种情况，通常情况下就是 Controller 不工作了，导致无法分配 leader，那既然知道问题后，解决方案也就很简单了。重启 Controller 节点上的 Kafka 进程，让其他节点重新注册Controller角色,但是如上面ZooKeeper的作用,你要知道为什么Controller可以自动注册。

2、 当然了，当你知道 controller 的注册机制后，你也可以说：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。但是，需要注意的是，直接操作 ZooKeeper 是一件风险很大的操作，就好比在 Linux 中执行了 rm -rf /xxx 一样，如果在/和 xxx 之间不小心多了几个空格，那”恭喜你”，今年白干了。

LEO、LSO、AR、ISR、HW 都表示什么含义?

讲真，我不认为这是炫技的题目，特别是作为 SRE 来讲，对于一个开源软件的原理以及概念的理解，是非常重要的。

1、 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。

2、 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。

3、 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。

4、 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。

5、 HW（High watermark）：高水位值，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。

需要注意的是，通常在 ISR 中，可能会有人问到为什么有时候副本不在 ISR 中，这其实也就是上面说的 Leader 和 Follower 不同步的情况，为什么我们前面说，短暂的不同步我们可以关注，但是长时间的不同步，我们需要介入排查了，因为 ISR 里的副本后面都是通过 replica.lag.time.max.ms，即 Follower 副本的 LEO 落后 Leader LEO 的时间是否超过阈值来决定副本是否在 ISR 内部的。

Kafka 能手动删除消息吗?

1、 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。

2、 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 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。

3、 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。

consumer_offsets 是做什么用的?

这是一个内部主题，主要用于存储消费者的偏移量，以及消费者的元数据信息（消费者实例，消费者 id 等等）

需要注意的是：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。

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。

1、 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。

2、 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。

3、 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。

4、 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。

Kafka 的哪些场景中使用了零拷贝（Zero Copy）？

1、 其实这道题对于 SRE 来讲，有点超纲了，不过既然 Zero Copy 是 Kafka 高性能的保证，我们需要了解它。

2、 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。

3、 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。

4、 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。

Kafka 为什么不支持读写分离?

1、 这其实是分布式场景下的通用问题，因为我们知道 CAP 理论下，我们只能保证 C（一致性）和 A（可用性）取其一，如果支持读写分离，那其实对于一致性的要求可能就会有一定折扣，因为通常的场景下，副本之间都是通过同步来实现副

本数据一致的，那同步过程中肯定会有时间的消耗，如果支持了读写分离，就意味着可能的数据不一致，或数据滞后。

2、 Leader/Follower 模型并没有规定 Follower 副本不可以对外提供读服务。很多框架都是允许这么做的，只是 Kafka 最初为了避免不一致性的问题，而采用了让 Leader 统一提供服务的方式。

3、 不过，自 Kafka 2.4 之后，Kafka 提供了有限度的读写分离，也就是说，Follower 副本能够对外提供读服务。

如何调优 Kafka?

1、 作为 SRE 来讲，任何生产环境的调优，首先需要识别问题和瓶颈点，而不是随意的进行臆想调优。随后，需要确定优化目标，并且定量给出目标。

2、 对于 Kafka 来讲，常见的调优方向基本为：吞吐量、延时、持久性和可用性，每种目标之前都是由冲突点，这也就要求了，我们在对业务接入使用时，要进行业务场景的了解，以对业务进行相对的集群隔离，因为每一个方向的优化思路都是不同的，甚至是相反的。

3、 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：

1、 Producer 端：增加 batch.size 和 linger.ms，启用压缩，关闭重试

2、 Broker 端：增加 num.replica.fetchers 提升 Follower 同步 TPS，避免 Broker Full GC 等。

3、 Consumer：增加 fetch.min.bytes

Controller 发生网络分区 (Network Partitioning) 时，Kafka 会怎么样?

1、 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。

2、 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。

3、 不过，通常而言，我们在设计整个部署架构时，为了避免这种网络分区的发生，一般会将 broker 节点尽可能的防止在一个机房或者可用区。

4、 由于 Controller 会给 Broker 发送 3 类请求，LeaderAndIsrRequest，StopReplicaRequest，UpdateMetadataRequest，因此，一旦出现网络分区，这些请求将不能顺利到达 Broker 端。

5、 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。

Java Consumer 为什么采用单线程来获取消息?

1、 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。

2、 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。

简述 Follower 副本消息同步的完整流程

1、 首先，Follower 发送 FETCH 请求给 Leader。

2、 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。

3、 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。

4、 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。

5、 因此，对于消费者而言，消费到的消息永远是所有副本中最小的那个 HW。

消息队列的作用

解耦

快递小哥手上有很多快递需要送，他每次都需要先电话一一确认收货人是否有空、哪个时间段有空，然后再确定好送货的方案。这样完全依赖收货人了！如果快递一多，快递小哥估计的忙疯了……

如果有了便利店，快递小哥只需要将同一个小区的快递放在同一个便利店，然后通知收货人来取货就可以了，这时候快递小哥和收货人就实现了解耦！

异步

快递小哥打电话给我后需要一直在你楼下等着，直到我拿走你的快递他才能去送其他人的。快递小哥将快递放在小芳便利店后，又可以干其他的活儿去了，不需要等待你到来而一直处于等待状态。提高了工作的效率。

削峰

假设双十一我买了不同店里的各种商品，而恰巧这些店发货的快递都不一样，有中通、圆通、申通、各种通等……

更巧的是他们都同时到货了！中通的小哥打来电话叫我去北门取快递、圆通小哥叫我去南门、申通小哥叫我去东门。我一时手忙脚乱……

我们能看到在系统需要交互的场景中，使用消息队列中间件真的是好处多多，基于这种思路，就有了丰巢、菜鸟驿站等比小芳便利店更专业的“中间件”了。

Kafka 中有哪几个组件?

主题：Kafka 主题是一堆或一组消息。

生产者：在 Kafka，生产者发布通信以及向 Kafka 主题发布消息。

消费者：Kafka 消费者订阅了一个主题，并且还从主题中读取和处理消息。

经纪人：在管理主题中的消息存储时，我们使用 Kafka Brokers。

简单说一下 ack 机制

ack：producer 收到多少 broker 的答复才算真的发送成功

0 表示 producer 无需等待 leader 的确认(吞吐最高、数据可靠性最差)

1 代表需要 leader 确认写入它的本地 log 并立即确认

-1/all 代表所有的 ISR 都完成后确认(吞吐最低、数据可靠性最高)

Kafka 如何判断节点是否存活

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

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

问题 5：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）

能说一下 leader 选举过程吗

我们知道 ZooKeeper 集群中也有选举机制，是通过 Paxos 算法，通过不同节点向其他节点发送信息来投票选举出 leader，但是 Kafka 的 leader 的选举就没有这么复杂了。

Kafka 的 Leader 选举是通过在 ZooKeeper 上创建/controller 临时节点来实现 leader 选举，并在该节点中写入当前 broker 的信息 {“version”:1,”brokerid”:1,” timestamp”:” 1512018424988” } 利用 ZooKeeper 的强一致性特性，一个节点只能被一个客户端创建成功，创建成功的 broker 即为 leader，即先到先得原则，leader 也就是集群中的 controller，负责集群中所有大小事务。当 leader 和

ZooKeeper 失去连接时，临时节点会删除，而其他 broker 会监听该节点的变化，当节点删除时，其他 broker 会收到事件通知，重新发起 leader 选举。

Kafka 什么情况下会 rebalance

rebalance 的触发条件有五个。

条件 1：有新的 consumer 加入

条件 2：旧的 consumer 挂了

条件 3：coordinator 挂了，集群选举出新的 coordinator

条件 4：topic 的 partition 新加

条件 5：consumer 调用 unsubscrible()，取消 topic 的订阅

rebalance 发生时，Group 下所有 consumer 实例都会协调在一起共同参与，Kafka 能够保证尽量达到最公平的分配。但是 Rebalance 过程对 consumer group 会造成比较严重的影响。在 Rebalance 的过程中 consumer group 下的所有消费者实例都会停止工作，等待 Rebalance 过程完成。

能简单说一下 rebalance 过程吗?

主要的流程如下：

发送 GCR 请求寻找 Coordinator：这个过程主要会向集群中负载最小的 broker 发起请求，等待成功返回后，那么该 Broker 将作为 Coordinator，尝试连接该 Coordinator

发送 JGR 请求加入该组：当成功找到 Coordinator 后，那么就要发起加入 group 的请求，表示该 consumer 是该组的成员，Coordinator 会接收到该请求，会给集群分配一个 Leader（通常是第一个），让其负责 partition 的分配

发送 SGR 请求：JGR 请求成功后，如果发现当前 Consumer 是 leader，那么会进行 partition 的分配，并发起 SGR 请求将分配结果发送给 Coordinator;如果不是 leader，那么也会发起 SGR 请求，不过分配结果为空

Rebalance 有什么影响

Rebalance 本身是 Kafka 集群的一个保护设定，用于剔除掉无法消费或者过慢的消费者，然后由于我们的数据量较大，同时后续消费后的数据写入需要走网络 IO，很有可能存在依赖的第三方服务存在慢的情况而导致我们超时。Rebalance 对我们数据的影响主要有以下几点：

数据重复消费：消费过的数据由于提交 offset 任务也会失败，在 partition 被分配给其他消费者的时候，会造成重复消费，数据重复且增加集群压力

Rebalance 扩散到整个 ConsumerGroup 的所有消费者，因为一个消费者的退出，导致整个 Group 进行了 Rebalance，并在一个比较慢的时间内达到稳定状态，影响面较大

频繁的 Rebalance 反而降低了消息的消费速度，大部分时间都在重复消费和 Rebalance

数据不能及时消费，会累积 lag，在 Kafka 的 TTL 之后会丢弃数据 上面的影响对于我们系统来说，都是致命的。

怎么解决 rebalance 中遇到的问题呢?

要避免 Rebalance，还是要从 Rebalance 发生的时机入手。我们在前面说过，Rebalance 主要发生的时机有三个：

组成员数量发生变化

订阅主题数量发生变化

订阅主题的分区数发生变化

后两个我们大可以人为的避免，发生 rebalance 最常见的原因是消费组成员的变化。

消费者成员正常的添加和停掉导致 rebalance，这种情况无法避免，但是时在某些情况下，Consumer 实例会被 Coordinator 错误地认为 “已停止” 从而被“踢出”Group。从而导致 rebalance。

当 Consumer Group 完成 Rebalance 之后，每个 Consumer 实例都会定期地向 Coordinator 发送心跳请求，表明它还存活着。如果某个 Consumer 实例不能及时地发送这些心跳请求，Coordinator 就会认为该 Consumer 已经 “死”了，从而将其从 Group 中移除，然后开启新一轮 Rebalance。这个时间可以通过 Consumer 端的参数 session.timeout.ms 进行配置。默认值是 10 秒。

除了这个参数，Consumer 还提供了一个控制发送心跳请求频率的参数，就是 heartbeat.interval.ms。这个值设置得越小，Consumer 实例发送心跳请求的频率就越高。频繁地发送心跳请求会额外消耗带宽资源，但好处是能够更加快速地知晓当前是否开启 Rebalance，因为，目前 Coordinator 通知各个 Consumer 实例开启 Rebalance 的方法，就是将 REBALANCE_NEEDED 标志封装进心跳请求的响应体中。

除了以上两个参数，Consumer 端还有一个参数，用于控制 Consumer 实际消费能力对 Rebalance 的影响，即 max.poll.interval.ms 参数。它限定了 Consumer 端应用程序两次调用 poll 方法的最大时间间隔。它的默认值是 5 分钟，表示你的 Consumer 程序如果在 5 分钟之内无法消费完 poll 方法返回的消息，那么 Consumer 会主动发起 “离开组” 的请求，Coordinator 也会开启新一轮 Rebalance。

通过上面的分析，我们可以看一下那些 rebalance 是可以避免的：

第一类非必要 Rebalance 是因为未能及时发送心跳，导致 Consumer 被 “踢出” Group 而引发的。这种情况下我们可以设置 session.timeout.ms 和 heartbeat.interval.ms 的值，来尽量避免 rebalance 的出现。（以下的配置是在网上找到的最佳实践，暂时还没测试过）

设置 session.timeout.ms = 6s。设置 heartbeat.interval.ms = 2s。要保证 Consumer 实例在被判定为 “dead” 之前，能够发送至少 3 轮的心跳请求，即 session.timeout.ms >= 3 * heartbeat.interval.ms。将 session.timeout.ms 设置成 6s 主要是为了让 Coordinator 能够更快地定位已经挂掉的 Consumer，早日把它们踢出 Group。

第二类非必要 Rebalance 是 Consumer 消费时间过长导致的。此时，max.poll.interval.ms 参数值的设置显得尤为关键。如果要避免非预期的 Rebalance，你最好将该参数值设置得大一点，比你的下游最大处理时间稍长一点。

总之，要为业务处理逻辑留下充足的时间。这样，Consumer 就不会因为处理这些消息的时间太长而引发 Rebalance 。

Kafka 一次 reblance 大概要多久

1 个 Topic，10 个 partition，3 个 consumer 测试结果 经过几轮测试发现每次 rebalance 所消耗的时间大概在 80ms~100ms 平均耗时在 87ms 左右。

如何保证 Kafka 顺序消费

这个在我看来是一个伪命题，如果要保证顺序消费为啥要用 Kafka 呢，只是需要做到异步或者解耦？如果一定要做到顺序消费，肯定是可以的，但是这个浪费资源，因为 Kafka 就是针对高并发大吞吐量而生，下面说一下顺序消费方案：

1、 一个 topic、一个 partition、一个线程

2、 一个 topic、n 个 partition、n 个线程，这里生产时需要根据需求将需要排序的数据发送到指定的 message key

Kafka 为何这么快

Kafka 实现了零拷贝原理来快速移动数据，避免了内核之间的切换。Kafka 可以将数据记录分批发送，从生产者到文件系统（Kafka 主题日志）到消费者，可以端到端的查看这些批次的数据。

批处理能够进行更有效的数据压缩并减少 I/O 延迟，Kafka 采取顺序写入磁盘的方式，避免了随机磁盘寻址的浪费，更多关于磁盘寻址的了解，请参阅 程序员需要了解的硬核知识之磁盘 。总结一下其实就是四个要点

顺序读写

零拷贝

消息压缩

分批发送

Apache Kafka 是什么?

能问这道题，主要是想看候选人对于 Kafka 的使用场景以及定位认知理解有多深，同时候可以知道候选人对于这项技术的关注度。

我们都知道，在开源软件中，大部分软件随着用户量的增加，整个软件的功能和定位也有了新的变化，而 Apache Kafka 一路发展到现在，已经由最初的分布式提交日志系统逐渐演变成了实时流处理框架。
因此，这道题你最好这么回答：Apach Kafka 是一款分布式流处理平台，用于实时构建流处理应用。它有一个核心的功能广为人知，即作为企业级的消息引擎被广泛使用（通常也会称之为消息总线 message bus）。
关于分布式流处理平台，其实从它官方的 Logo 以及 Slogan 我们就很容易看出来。

什么是消费者组?

1、 消费者组是 Kafka 独有的概念，如果面试官问这个，就说明他对此是有一定了解的。

2、 胡大给的标准答案是：官网上的介绍言简意赅，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。

3、 但实际上，消费者组（Consumer Group）其实包含两个概念，作为队列，消费者组允许你分割数据处理到一组进程集合上（即一个消费者组中可以包含多个消费者进程，他们共同消费该 topic 的数据），这有助于你的消费能力的动态调整；作为-订阅模型（publish-subscribe），Kafka 允许你将同一份消息广播到多个消费者组里，以此来丰富多种数据使用场景。

4、 需要注意的是：在消费者组中，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。因此，消费者组在一定程度上也保证了消费者程序的高可用性。

注意：消费者组的题目，能够帮你在某种程度上掌控下面的面试方向。

1、 如果你擅长位移值原理（Offset），就不妨再提一下消费者组的位移提交机制；

2、 如果你擅长 Kafka Broker，可以提一下消费者组与 Broker 之间的交互；

3、 如果你擅长与消费者组完全不相关的 Producer，那么就可以这么说：“消费者组要消费的数据完全来自于 Producer 端生产的消息，我对 Producer 还是比较熟悉的。”

总之，你总得对 consumer group 相关的方向有一定理解，然后才能像面试官表名你对某一块很理解。

在 Kafka 中，ZooKeeper 的作用是什么?

1、 这道题，也是我经常会问候选人的题，因为任何分布式系统中虽然都通过一些列的算法去除了传统的关系型数据存储，但是毕竟还是有些数据要存储的，同时分布式系统的特性往往是需要有一些中间人角色来统筹集群。比如我们在整个微服务框架中的 Dubbo，它也是需要依赖一些注册中心或配置中心类的中间件的，以及云原生的 Kubernetes 使用 etcd 作为整个集群的枢纽。

2、 标准答案：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。

1、 “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。

2、 “成员管理” 是指 Broker 节点的注册、注销以及属性变更，等等。

3、 “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。

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。