## **Q1：Kafka是如何保障数据不丢失的？**

该问题已经成为了Kafka面试的惯例，如同Java的**HashMap**，属于高频出现的面试问题。那么，我们该怎么理解这个问题呢？问题是**Kafka如何保障数据不丢失**，即**Kafka的Broker提供了什么机制保证数据不丢失的。**

其实对于Kafka的Broker而言，Kafka 的**复制机制**和**分区的多副本**架构是Kafka 可靠性保证的核心。把消息写入多个副本可以使Kafka 在发生崩溃时仍能保证消息的持久性。

搞清楚了问题的核心，再来看一下该怎么回答这个问题：主要包括三个方面

1.Topic 副本因子个数：replication.factor >= 3  
2.同步副本列表(ISR)：min.insync.replicas = 2  
3.禁用unclean选举：unclean.leader.election.enable=false

下面将会逐步分析上面的三个配置：

* **副本因子**

Kafka的topic是可以分区的，并且可以为分区配置多个副本，该配置可以通过replication.factor参数实现。Kafka中的分区副本包括两种类型：领导者副本（Leader Replica）和追随者副本（Follower Replica)，每个分区在创建时都要选举一个副本作为领导者副本，其余的副本自动变为追随者副本。在 Kafka 中，追随者副本是不对外提供服务的，也就是说，任何一个追随者副本都不能响应消费者和生产者的读写请求。所有的请求都必须由领导者副本来处理。换句话说，所有的读写请求都必须发往领导者副本所在的 Broker，由该 Broker 负责处理。追随者副本不处理客户端请求，它唯一的任务就是从领导者副本**异步拉取**消息，并写入到自己的提交日志中，从而实现与领导者副本的同步。

一般来说，副本设为3可以满足大部分的使用场景，也有可能是5个副本(比如银行)。如果副本因子为N，那么在N-1个broker 失效的情况下，仍然能够从主题读取数据或向主题写入数据。所以，更高的副本因子会带来更高的可用性、可靠性和更少的故障。另一方面，副本因子N需要至少N个broker ，而且会有N个数据副本，也就是说它们会占用N倍的磁盘空间。实际生产环境中一般会在可用性和存储硬件之间作出权衡。

除此之外，副本的分布同样也会影响可用性。默认情况下，Kafka会确保分区的每个副本分布在不同的Broker上，但是如果这些Broker在同一个机架上，一旦机架的交换机发生故障，分区就会不可用。所以建议把Broker分布在不同的机架上，可以使用**broker.rack**参数配置Broker所在机架的名称。

* **同步副本列表**

In-sync replica(ISR)称之为同步副本，ISR中的副本都是与Leader进行同步的副本，所以不在该列表的follower会被认为与Leader是不同步的。那么，ISR中存在是什么副本呢？首先可以明确的是：Leader副本总是存在于ISR中。而follower副本是否在ISR中，取决于该follower副本是否与Leader副本保持了“同步”。

Kafka的broker端有一个参数**replica.lag.time.max.ms**, 该参数表示follower副本滞后与Leader副本的最长时间间隔，默认是10秒。这就意味着，只要follower副本落后于leader副本的时间间隔不超过10秒，就可以认为该follower副本与leader副本是同步的，所以哪怕当前follower副本落后于Leader副本几条消息，只要在10秒之内赶上Leader副本，就不会被踢出出局。

可以看出ISR是一个动态的，所以即便是为分区配置了3个副本，还是会出现同步副本列表中只有一个副本的情况(其他副本由于不能够与leader及时保持同步，被移出ISR列表)。如果这个同步副本变为不可用，我们必须在**可用性**和**一致性**之间作出选择(CAP理论)。

根据Kafka 对可靠性保证的定义，消息只有在被写入到所有同步副本之后才被认为是已提交的。但如果这里的“所有副本”只包含一个同步副本，那么在这个副本变为不可用时，数据就会丢失。如果要确保已提交的数据被写入不止一个副本，就需要把最小同步副本数量设置为大一点的值。对于一个包含3 个副本的主题分区，如果**min.insync.replicas=2** ，那么至少要存在两个同步副本才能向分区写入数据。

如果进行了上面的配置，此时必须要保证ISR中至少存在两个副本，如果ISR中的副本个数小于2，那么Broker就会停止接受生产者的请求。尝试发送数据的生产者会收到**NotEnoughReplicasException**异常，消费者仍然可以继续读取已有的数据。

* **禁用unclean选举**

选择一个同步副本列表中的分区作为leader 分区的过程称为**clean leader election**。注意，这里要与在非同步副本中选一个分区作为leader分区的过程区分开，在非同步副本中选一个分区作为leader的过程称之为**unclean leader election**。由于ISR是动态调整的，所以会存在ISR列表为空的情况，通常来说，非同步副本落后 Leader 太多，因此，如果选择这些副本作为新 Leader，就可能出现数据的丢失。毕竟，这些副本中保存的消息远远落后于老 Leader 中的消息。在 Kafka 中，选举这种副本的过程可以通过Broker 端参数 **unclean.leader.election.enable**控制是否允许 Unclean 领导者选举。开启 Unclean 领导者选举可能会造成数据丢失，但好处是，它使得分区 Leader 副本一直存在，不至于停止对外提供服务，因此提升了高可用性。反之，禁止 Unclean Leader 选举的好处在于维护了数据的一致性，避免了消息丢失，但牺牲了高可用性。分布式系统的CAP理论说的就是这种情况。

不幸的是，**unclean leader election**的选举过程仍可能会造成数据的不一致，因为同步副本并不是**完全**同步的。由于复制是**异步**完成的，因此无法保证follower可以获取最新消息。比如Leader分区的最后一条消息的offset是100，此时副本的offset可能不是100，这受到两个参数的影响：

**replica.lag.time.max.ms**：同步副本滞后与leader副本的时间  
**zookeeper.session.timeout.ms**：与zookeeper会话超时时间

简而言之，如果我们允许不同步的副本成为leader，那么就要承担丢失数据和出现数据不一致的风险。如果不允许它们成为leader，那么就要接受较低的可用性，因为我们必须等待原先的首领恢复到可用状态。

关于unclean选举，不同的场景有不同的配置方式。对**数据质量和数据一致性**要求较高的系统会禁用这种unclean的leader选举(比如银行)。如果在**可用性**要求较高的系统里，比如实时点击流分析系统， 一般不会禁用unclean的leader选举。

## **Q2：如何解决Kafka数据丢失问题？**

你可能会问：这个问题跟Q1有什么区别呢？其实一般在面试问题中可以理解成一个问题。之所以在这里做出区分，是因为两者的解决方式不一样。**Q1问题是从Kafka的Broker侧来看待数据丢失的问题，而Q2是从Kafka的生产者与消费者的角度来看待数据丢失的问题**。

先来看一下如何回答这个问题：主要包括两个方面：

**Producer**  
retries=Long.MAX\_VALUE  
设置 retries 为一个较大的值。这里的 retries 同样是 Producer 的参数，对应前面提到的 Producer 自动重试。当出现网络的瞬时抖动时，消息发送可能会失败，此时配置了 retries > 0 的 Producer 能够自动重试消息发送，避免消息丢失。  
acks=all  
设置 acks = all。acks 是 Producer 的一个参数，代表了你对“已提交”消息的定义。如果设置成 all，则表明所有副本 Broker 都要接收到消息，该消息才算是“已提交”。这是最高等级的“已提交”定义。  
max.in.flight.requests.per.connections=1  
该参数指定了生产者在收到服务器晌应之前可以发送多少个消息。它的值越高，就会占用越多的内存，不过也会提升吞吐量。把它设为1 可以保证消息是按照发送的顺序写入服务器的，即使发生了重试。  
Producer要使用带有回调通知的API，也就是说不要使用 producer.send(msg)，而要使用 producer.send(msg, callback)。  
其他错误处理  
使用生产者内置的重试机制，可以在不造成消息丢失的情况下轻松地处理大部分错误，不过  
仍然需要处理其他类型的错误，例如消息大小错误、序列化错误等等。  
**Consumer**  
禁用自动提交：enable.auto.commit=false  
消费者处理完消息之后再提交offset  
配置auto.offset.reset  
这个参数指定了在没有偏移量可提交时(比如消费者第l次启动时)或者请求的偏移量在broker上不存在时(比如数据被删了)，消费者会做些什么。  
这个参数有两种配置。一种是**earliest**：消费者会从分区的开始位置读取数据，不管偏移量是否有效，这样会导致消费者读取大量的重复数据，但可以保证最少的数据丢失。一种是**latest(默认)**，如果选择了这种配置， 消费者会从分区的末尾开始读取数据，这样可以减少重复处理消息，但很有可能会错过一些消息。

## **Q3：Kafka可以保障永久不丢失数据吗？**

上面分析了一些保障数据不丢失的措施，在一定程度上可以避免数据的丢失。但是请注意：**Kafka 只对“已提交”的消息（committed message）做有限度的持久化保证**。所以说，Kafka不能够完全保证数据不丢失，需要做出一些权衡。

首先，要理解什么是**已提交的消息**，当 Kafka 的若干个 Broker 成功地接收到一条消息并写入到日志文件后，它们会告诉生产者程序这条消息已成功提交。此时，这条消息在 Kafka 看来就正式变为**已提交**消息了。所以说无论是ack=all，还是ack=1,不论哪种情况，Kafka 只对已提交的消息做持久化保证这件事情是不变的。

其次，要理解**有限度的持久化保证**，也就是说 Kafka 不可能保证在任何情况下都做到不丢失消息。必须保证Kafka的Broker是可用的，换句话说，假如消息保存在 N 个 Kafka Broker 上，那么这个前提条件就是这 N 个 Broker 中至少有 1 个存活。只要这个条件成立，Kafka 就能保证你的这条消息永远不会丢失。

总结一下，Kafka 是能做到不丢失消息的，**只不过这些消息必须是已提交的消息**，而且还要满足一定的条件。

## **Q4：如何保障Kafka中的消息是有序的？**

首先需要明确的是：**Kafka的主题是分区有序的**，如果一个主题有多个分区，那么Kafka会按照key将其发送到对应的分区中，所以，对于给定的key，与其对应的record在分区内是有序的。

Kafka可以保证同一个分区里的消息是有序的，即生产者按照一定的顺序发送消息，Broker就会按照这个顺序将他们写入对应的分区中，同理，消费者也会按照这个顺序来消费他们。

在一些场景下，消息的顺序是非常重要的。比如，**先存钱再取钱**与**先取钱再存钱**是截然不同的两种结果。

上面的问题中提到一个参数**max.in.flight.requests.per.connections=1**,该参数的作用是在重试次数大于等于1时，保证数据写入的顺序。如果该参数不为1，那么当第一个批次写入失败时，第二个批次写入成功，Broker会重试写入第一个批次，如果此时第一个批次重试写入成功，那么这两个批次消息的顺序就反过来了。

一般来说，如果对消息的顺序有要求，那么在为了保障数据不丢失，需要先设置发送重试次数retries>0,同时需要把**max.in.flight.requests.per.connections**参数设为1，这样在生产者尝试发送第一批消息时，就不会有其他的消息发送给broker，虽然会影响吞吐量，但是可以保证消息的顺序。

除此之外，还可以使用单分区的Topic，但是会严重影响吞吐量。

## **Q5：如何确定合适的Kafka主题的分区数量？**

选择合适的分区数量可以达到高度并行读写和负载均衡的目的，在分区上达到均衡负载是实现吞吐量的关键。需要根据每个分区的生产者和消费者的期望吞吐量进行估计。

举个栗子：假设期望读取数据的速率(吞吐量)为**1GB/Sec**，而一个消费者的读取速率为**50MB/Sec**，此时至少需要20个分区以及20个消费者(一个消费者组)。同理，如果期望生产数据的速率为**1GB/Sec**，而每个生产者的生产速率为**100MB/Sec**，此时就需要有10个分区。在这种情况下，如果设置20个分区，既可以保障**1GB/Sec**的生产速率，也可以保障消费者的吞吐量。通常需要将分区的数量调整为消费者或者生产者的数量，只有这样才可以同时实现生产者和消费者的吞吐量。

一个简单的计算公式为：**分区数 = max(生产者数量，消费者数量)**

* 生产者数量=整体生产吞吐量/每个生产者对单个分区的最大生产吞吐量
* 消费者数量=整体消费吞吐量/每个消费者从单个分区消费的最大吞吐量

## **Q6：如何调整生产环境中Kafka主题的分区数量？**

需要注意的是：当我们增加主题的分区数量时，会违背**同一个key进行同一个分区**的事实。我们可以创建一个新的主题，使得该主题有更多的分区数，然后暂停生产者，将旧的主题中的数据复制到新的主题中，然后将消费者和生产者切换到新的主题，操作起来会非常棘手。

## **Q7:如何重平衡Kafka集群？**

在下面情况发生时，需要重平衡集群：

* 主题分区在整个集群里的不均衡分布造成了集群负载的不均衡。
* broker离线造成分区不同步。
* 新加入的broker 需要从集群里获得负载。

使用**kafka-reassign-partitions.sh**命令进行重平衡

## **Q8:如何查看消费者组是否存在滞后消费？**

我们可以使用**kafka-consumer-groups.sh**命令进行查看，比如：

$ bin/kafka-consumer-groups.sh --bootstrap-server cdh02:9092 --describe --group my-group

## 会显示下面的一些指标信息

TOPIC PARTITION CURRENT-OFFSET LOG-END-OFFSET LAG CONSUMER-ID HOST CLIENT-ID

主题 分区 当前offset LEO 滞后消息数 消费者id 主机 客户端id

一般情况下，如果运行良好，**CURRENT-OFFSET**的值会与**LOG-END-OFFSET**的值非常接近。通过这个命令可以查看哪个分区的消费出现了滞后。

## **总结**

本文主要分享了8个常见的Kafka面试题，对于每个题目都给出了相应的答案。对照这些问题，相信会对Kafka会有更深刻的认识。