Kafka 基础篇

1.Kafka 的用途有哪些？使用场景如何？

消息系统： Kafka 和传统的消息系统（也称作消息中间件）都具备系统解耦、冗余存储、流量削峰、缓冲、异步通信、扩展性、可恢复性等功能。与此同时，Kafka 还提供了大多数消息系统难以实现的消息顺序性保障及回溯消费的功能。

存储系统： Kafka 把消息持久化到磁盘，相比于其他基于内存存储的系统而言，有效地降低了数据丢失的风险。也正是得益于 Kafka 的消息持久化功能和多副本机制，我们可以把 Kafka 作为长期的数据存储系统来使用，只需要把对应的数据保留策略设置为“永久”或启用主题的日志压缩功能即可。

流式处理平台： Kafka 不仅为每个流行的流式处理框架提供了可靠的数据来源，还提供了一个完整的流式处理类库，比如窗口、连接、变换和聚合等各类操作。

2.Kafka 中的 ISR、AR 又代表什么？ISR 的伸缩又指什么

分区中的所有副本统称为 AR（Assigned Replicas）。所有与 leader 副本保持一定程度同步的副本（包括 leader 副本在内）组成 ISR（In-Sync Replicas），ISR 集合是 AR 集合中的一个子集。

ISR 的伸缩：
leader 副本负责维护和跟踪 ISR 集合中所有 follower 副本的滞后状态，当 follower 副本落后太多或失效时，leader 副本会把它从 ISR 集合中剔除。如果 OSR 集合中有 follower 副本“追上”了 leader 副本，那么 leader 副本会把它从 OSR 集合转移至 ISR 集合。默认情况下，当 leader 副本发生故障时，只有在 ISR 集合中的副本才有资格被选举为新的 leader，而在 OSR 集合中的副本则没有任何机会（不过这个原则也可以通过修改相应的参数配置来改变）。

replica.lag.time.max.ms ： 这个参数的含义是 Follower 副本能够落后 Leader 副本的最长时间间隔，当前默认值是 10 秒。

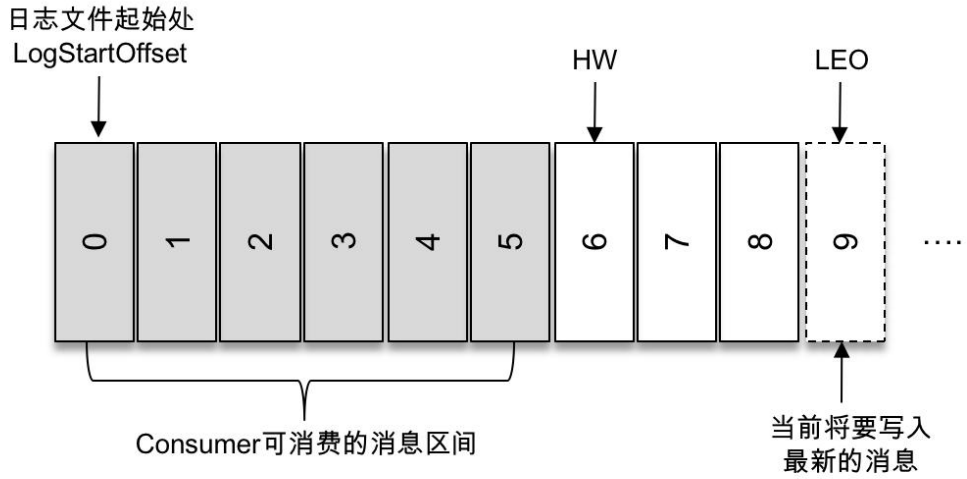
unclean.leader.election.enable：是否允许 Unclean 领导者选举。开启 Unclean 领导者选举可能会造成数据丢失，但好处是，它使得分区 Leader 副本一直存在，不至于停止对外提供服务，因此提升了高可用性。

3.Kafka 中的 HW、LEO、LSO、LW 等分别代表什么？

HW 是 High Watermark 的缩写，俗称高水位，它标识了一个特定的消息偏移量（offset），消费者只能拉取到这个 offset 之前的消息。

LSO 是 LogStartOffset，一般情况下，日志文件的起始偏移量 logStartOffset 等于第一个日志分段的 baseOffset，但这并不是绝对的，logStartOffset 的值可以通过

DeleteRecordsRequest 请求(比如使用 KafkaAdminClient 的 deleteRecords()方法、使用 kafka-delete-records.sh 脚本、日志的清理和截断等操作进行修改。



如上图所示，它代表一个日志文件，这个日志文件中有 9 条消息，第一条消息的 offset（LogStartOffset）为 0，最后一条消息的 offset 为 8，offset 为 9 的消息用虚线框表示，代表下一条待写入的消息。日志文件的 HW 为 6，表示消费者只能拉取到 offset 在 0 至 5 之间的消息，而 offset 为 6 的消息对消费者而言是不可见的。

LEO 是 Log End Offset 的缩写，它标识当前日志文件中下一条待写入消息的 offset，上图中 offset 为 9 的位置即为当前日志文件的 LEO，LEO 的大小相当于当前日志分区中最后一条消息的 offset 值加 1。分区 ISR 集合中的每个副本都会维护自身的 LEO，而 ISR 集合中最小的 LEO 即为分区的 HW，对消费者而言只能消费 HW 之前的消息。

LW 是 Low Watermark 的缩写，俗称“低水位”，代表 AR 集合中最小的 logStartOffset 值。副本的拉取请求(FetchRequest，它有可能触发新建日志分段而旧的被清理，进而导致 logStartOffset 的增加)和删除消息请求(DeleteRecordRequest)都有可能促使 LW 的增长。

4.Kafka 中是怎么体现消息顺序性的？

可以通过分区策略体现消息顺序性。
分区策略有轮询策略、随机策略、按消息键保序策略。

按消息键保序策略：一旦消息被定义了 Key，那么你就可以保证同一个 Key 的所有消息都进入到相同的分区里面，由于每个分区下的消息处理都是有顺序的，故这个策略被称为按消息键保序策略

```
List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);return Math.abs(key.hashCode()) % partitions.size();
```

5.Kafka 中的分区器、序列化器、拦截器是否了解？它们之间的处理顺序是什么？

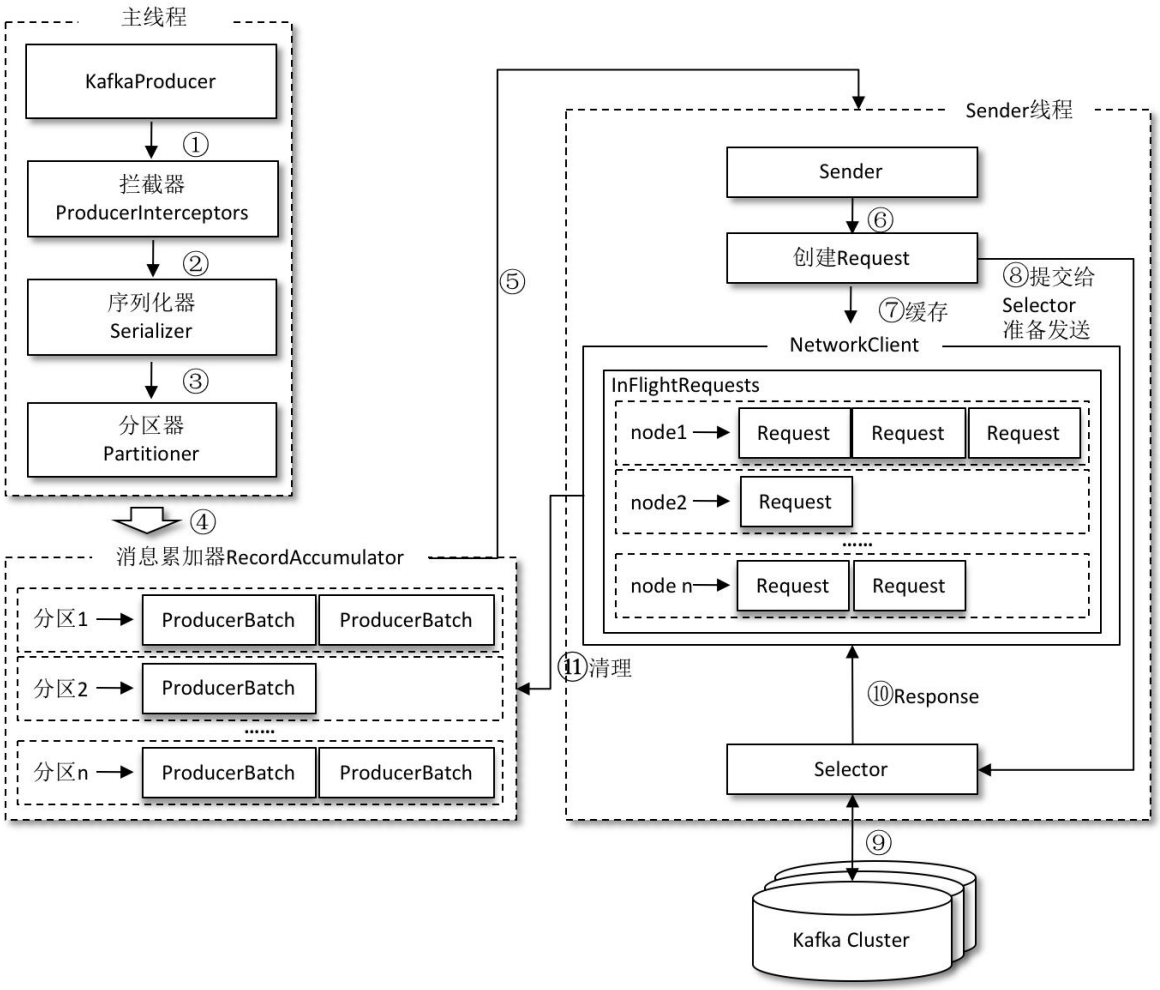
- 序列化器：生产者需要用序列化器（Serializer）把对象转换成字节数组才能通过网络发送给 Kafka。而在对侧，消费者需要用反序列化器（Deserializer）把从 Kafka 中收到的字节数组转换成相应的对象。
- 分区器：分区器的作用就是为消息分配分区。如果消息 ProducerRecord 中没有指定 partition 字段，那么就需要依赖分区器，根据 key 这个字段来计算 partition 的值。
- Kafka 一共有两种拦截器：生产者拦截器和消费者拦截器。
  - 生产者拦截器既可以用来在消息发送前做一些准备工作，比如按照某个规则过滤不符合要求的消息、修改消息的内容等，也可以用来在发送回调逻辑前做一些定制化的需求，比如统计类工作。
  - 消费者拦截器主要在消费到消息或在提交消费位移时进行一些定制化的操作。

消息在通过 send() 方法发往 broker 的过程中，有可能需要经过拦截器（Interceptor）、序列化器（Serializer）和分区器（Partitioner）的一系列作用之后才能被真正地发往 broker。拦截器（下一章会详细介绍）一般不是必需的，而序列化器是必需的。消息经过序列化之后就需要确定它发往的分区，如果消息 ProducerRecord 中指定了 partition 字段，那么就不需要分区器的作用，因为 partition 代表的就是所要发往的分区号。

处理顺序 ：拦截器->序列化器->分区器

KafkaProducer 在将消息序列化和计算分区之前会调用生产者拦截器的 onSend() 方法来对消息进行相应的定制化操作。
然后生产者需要用序列化器（Serializer）把对象转换成字节数组才能通过网络发送给 Kafka。
最后可能会被发往分区器为消息分配分区。

6.Kafka 生产者客户端的整体结构是什么样子的？



整个生产者客户端由两个线程协调运行，这两个线程分别为主线程和 Sender 线程（发送线程）。

在主线程中由 KafkaProducer 创建消息，然后通过可能的拦截器、序列化器和分区器的作用之后缓存到消息累加器（RecordAccumulator，也称为消息收集器）中。

Sender 线程负责从 RecordAccumulator 中获取消息并将其发送到 Kafka 中。

RecordAccumulator 主要用来缓存消息以便 Sender 线程可以批量发送，进而减少网络传输的资源消耗以提升性能。

7.Kafka 生产者客户端中使用了几个线程来处理？分别是什么？

整个生产者客户端由两个线程协调运行，这两个线程分别为主线程和 Sender 线程（发送线程）。在主线程中由 KafkaProducer 创建消息，然后通过可能的拦截器、序列化器和分区

器的作用之后缓存到消息累加器（RecordAccumulator，也称为消息收集器）中。Sender 线程负责从 RecordAccumulator 中获取消息并将其发送到 Kafka 中。

8.Kafka 的旧版 Scala 的消费者客户端的设计有什么缺陷？

老版本的 Consumer Group 把位移保存在 ZooKeeper 中。Apache ZooKeeper 是一个分布式的协调服务框架，Kafka 重度依赖它实现各种各样的协调管理。将位移保存在 ZooKeeper 外部系统的做法，最显而易见的好处就是减少了 Kafka Broker 端的状态保存开销。

ZooKeeper 这类元框架其实并不适合进行频繁的写更新，而 Consumer Group 的位移更新却是一个非常频繁的操作。这种大吞吐量的写操作会极大地拖慢 ZooKeeper 集群的性能

9.“消费组中的消费者个数如果超过 topic 的分区，那么就会有消费者消费不到数据”这句话是否正确？如果正确，那么有没有什么 hack 的手段？

一般来说如果消费者过多，出现了消费者的个数大于分区个数的情况，就会有消费者分配不到任何分区。

开发者可以继承 AbstractPartitionAssignor 实现自定义消费策略，从而实现同一消费组内的任意消费者都可以消费订阅主题的所有分区：

```
public class BroadcastAssignor extends AbstractPartitionAssignor{
    @Override
    public String name() {
        return "broadcast";
    }

    private Map<String, List<String>> consumersPerTopic(
            Map<String, Subscription> consumerMetadata) {
        （具体实现请参考 RandomAssignor 中的 consumersPerTopic()方法）
    }

    @Override
    public Map<String, List<TopicPartition>> assign(
            Map<String, Integer> partitionsPerTopic,
            Map<String, Subscription> subscriptions) {
```

```
        Map<String, List<String>> consumersPerTopic =
                consumersPerTopic(subscriptions);
        Map<String, List<TopicPartition>> assignment = new HashMap<>();
                    //Java8
        subscriptions.keySet().forEach(memberId ->
                assignment.put(memberId, new ArrayList<>()));
                    //针对每一个主题，为每一个订阅的消费者分配所有的分区
        consumersPerTopic.entrySet().forEach(topicEntry->{
            String topic = topicEntry.getKey();
            List<String> members = topicEntry.getValue();

            Integer numPartitionsForTopic = partitionsPerTopic.get(topic);
            if (numPartitionsForTopic == null || members.isEmpty())
                return;
            List<TopicPartition> partitions = AbstractPartitionAssignor
                    .partitions(topic, numPartitionsForTopic);
            if (!partitions.isEmpty()) {
                members.forEach(memberId ->
                        assignment.get(memberId).addAll(partitions));
            }
        });
        return assignment;
    }
}
```

注意组内广播的这种实现方式会有一个严重的问题—默认的消费位移的提交会失效。

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?#

在旧消费者客户端中，消费位移是存储在 ZooKeeper 中的。而在新消费者客户端中，消费位移存储在 Kafka 内部的主题__consumer_offsets 中。
当前消费者需要提交的消费位移是 offset+1

10.有哪些情形会造成重复消费？

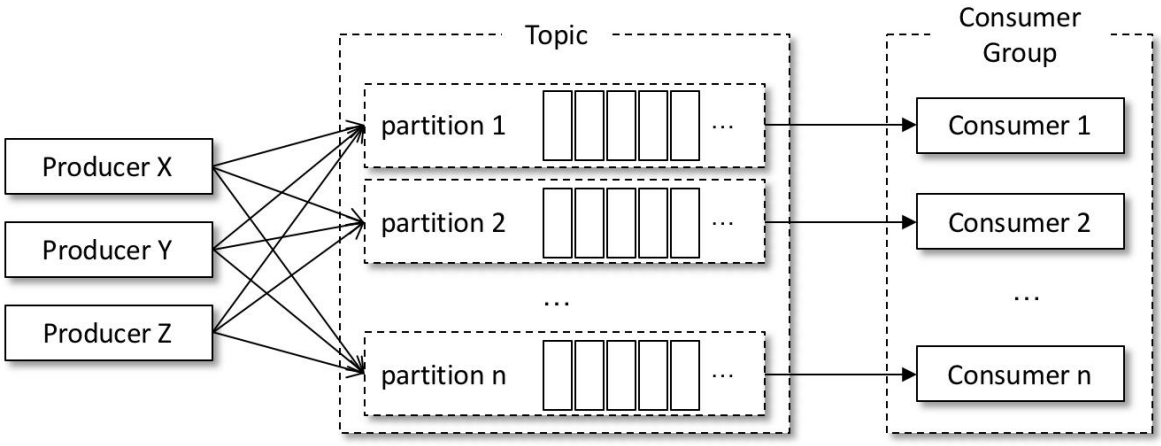
1. Rebalance
一个 consumer 正在消费一个分区的一条消息，还没有消费完，发生了 rebalance(加入了一个 consumer)，从而导致这条消息没有消费成功，rebalance 后，另一个 consumer 又把这条消息消费一遍。
2. 消费者端手动提交
如果先消费消息，再更新 offset 位置，导致消息重复消费。
3. 消费者端自动提交
设置 offset 为自动提交,关闭 kafka 时,如果在 close 之前,调用 consumer.unsubscribe()则有可能部分 offset 没提交，下次重启会重复消费。
4. 生产者端
生产者因为业务问题导致的宕机，在重启之后可能数据会重发

那些情景下会造成消息漏消费？#

1. 自动提交
设置 offset 为自动定时提交，当 offset 被自动定时提交时，数据还在内存中未处理，此时刚好把线程 kill 掉，那么 offset 已经提交，但是数据未处理，导致这部分内存中的数据丢失。
2. 生产者发送消息
发送消息设置的是 fire-and-forget（发后即忘），它只管往 Kafka 中发送消息而并不关心消息是否正确到达。不过在某些时候（比如发生不可重试异常时）会造成消息的丢失。这种发送方式的性能最高，可靠性也最差。
3. 消费者端
先提交位移，但是消息还没消费完就宕机了，造成了消息没有被消费。自动位移提交同理
4. acks 没有设置为 all
如果在 broker 还没把消息同步到其他 broker 的时候宕机了，那么消息将会丢失

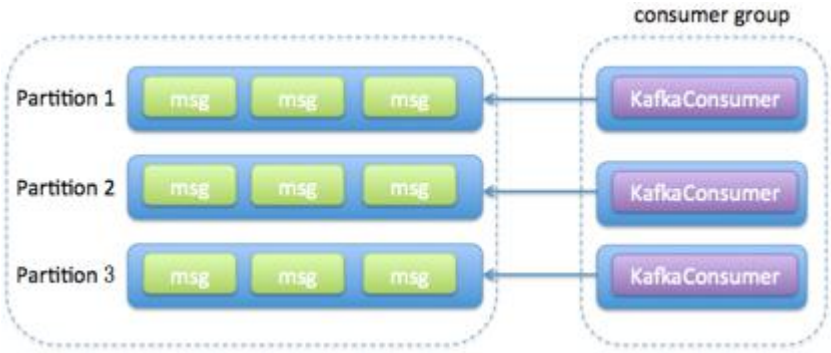
12.KafkaConsumer 是非线程安全的,那么怎么样实现多线程消费？

1. 线程封闭，即为每个线程实例化一个 KafkaConsumer 对象



一个线程对应一个 KafkaConsumer 实例，我们可以称之为消费线程。一个消费线程可以消费一个或多个分区中的消息，所有的消费线程都隶属于同一个消费组。

1. 消费者程序使用单或多线程获取消息，同时创建多个消费线程执行消息处理逻辑。获取消息的线程可以是一个，也可以是多个，每个线程维护专属的 KafkaConsumer 实例，处理消息则交由特定的线程池来做，从而实现消息获取与消息处理的真正解耦。具体架构如下图所示：



两个方案对比：

方案	优点	缺点
方案1： **多线程+多KafkaConsumer实例**	方便实现	占用更多系统资源
	速度快，无线程间交互开销	线程数受限于主题分区数，扩展性差
	易于维护分区内的消费顺序	线程自己处理消息容易超时，从而引发Rebalance
方案2： **单线程 + 单KafkaConsumer实例 + 消息处理Worker线程池**	可独立扩展消费获取线程数和Worker线程数	实现难度高
	伸缩性好	难以维护分区内的消息消费顺序
		处理链路拉长，不易于位移提交管理

13. 简述消费者与消费组之间的关系

1. Consumer Group 下可以有一个或多个 Consumer 实例。这里的实例可以是一个单独的进程，也可以是同一进程下的线程。在实际场景中，使用进程更为常见一些。
2. Group ID 是一个字符串，在一个 Kafka 集群中，它标识唯一的一个 Consumer Group。
3. Consumer Group 下所有实例订阅的主题的单个分区，只能分配给组内的某个 Consumer 实例消费。这个分区当然也可以被其他的 Group 消费。

14.当你使用 kafka-topics.sh 创建（删除）了一个 topic 之后，Kafka 背后会执行什么逻辑？

在执行完脚本之后，Kafka 会在 log.dir 或 log.dirs 参数所配置的目录下创建相应的主题分区，默认情况下这个目录为/tmp/kafka-logs/。

在 ZooKeeper 的/brokers/topics/目录下创建一个同名的实节点，该节点中记录了该主题的分区副本分配方案。示例如下：

```
[zk: localhost:2181/kafka(CONNECTED) 2] get /brokers/topics/topic-create
{"version":1,"partitions":{"2":[1,2],"1":[0,1],"3":[2,1],"0":[2,0]}}
```

15.topic 的分区数可不可以增加？如果可以怎么增加？如果不可以，那又是为什么？

可以增加，使用 kafka-topics 脚本，结合 --alter 参数来增加某个主题的分区数，命令如下：

```
bin/kafka-topics.sh --bootstrap-server broker_host:port --alter --topic <topic_name> --partitions <新分区数>
```

当分区数增加时，就会触发订阅该主题的所有 Group 开启 Rebalance。
首先，Rebalance 过程对 Consumer Group 消费过程有极大的影响。在 Rebalance 过程中，所有 Consumer 实例都会停止消费，等待 Rebalance 完成。这是 Rebalance 为人诟病的一个方面。
其次，目前 Rebalance 的设计是所有 Consumer 实例共同参与，全部重新分配所有分区。其实更高效的做法是尽量减少分配方案的变动。
最后，Rebalance 实在是太慢了。

16.topic 的分区数可不可以减少？如果可以怎么减少？如果不可以，那又是为什么？

不支持，因为删除的分区中的消息不好处理。如果直接存储到现有分区的尾部，消息的时间戳就不会递增，如此对于 Spark、Flink 这类需要消息时间戳（事件时间）的组件将会受到影响；如果分散插入现有的分区，那么在消息量很大的时候，内部的数据复制会占用很大的资源，而且在复制期间，此主题的可用性又如何得到保障？与此同时，顺序性问题、事务性问题，以及分区和副本的状态机切换问题都是不得不面对的。

17.创建 topic 时如何选择合适的分区数？

在 Kafka 中，性能与分区数有着必然的关系，在设定分区数时一般也需要考虑性能的因素。对不同的硬件而言，其对应的性能也会不太一样。
可以使用 Kafka 本身提供的用于生产者性能测试的 kafka-producer- perf-test.sh 和用于消费者性能测试的 kafka-consumer-perf-test.sh 来进行测试。
增加合适的分区数可以在一定程度上提升整体吞吐量，但超过对应的阈值之后吞吐量不升反降。如果应用对吞吐量有一定程度上的要求，则建议在投入生产环境之前对同款硬件资源做一个完备的吞吐量相关的测试，以找到合适的分区数阈值区间。
分区数的多少还会影响系统的可用性。如果分区数非常多，如果集群中的某个 broker 节点宕机，那么就会有大量的分区需要同时进行 leader 角色切换，这个切换的过程会耗费一笔可观的时间，并且在这个时间窗口内这些分区也会变得不可用。
分区数越多也会让 Kafka 的正常启动和关闭的耗时变得越长，与此同时，主题的分区数越多不仅会增加日志清理的耗时，而且在被删除时也会耗费更多的时间。

Kakfa 进阶篇

1.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

2.优先副本是什么？它有什么特殊的作用？

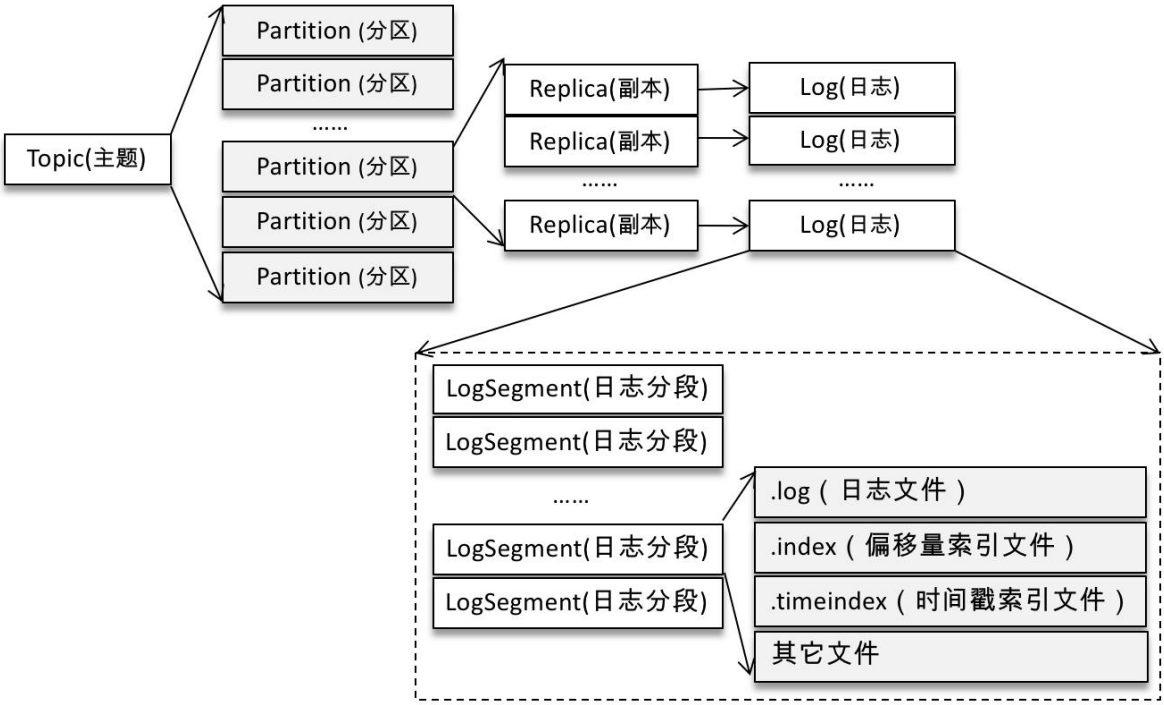
所谓的优先副本是指在 AR 集合列表中的第一个副本。
理想情况下，优先副本就是该分区的 leader 副本，所以也可以称之为 preferred leader。

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”。

3.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

1. 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。
2. 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。
3. 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。

4.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）

5.Kafka 中有那些索引文件？

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。

6.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。

7.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。

8.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。
我们可以通过 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。

1.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。

2.基于日志大小

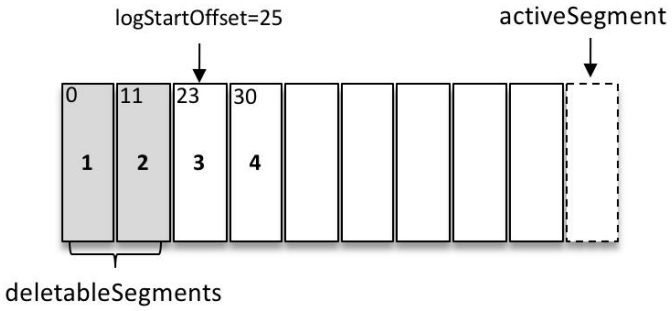
日志删除任务会检查当前日志的大小是否超过设定的阈值（retentionSize）来寻找可删除的日志分段的文件集合（deletableSegments）。

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。

3.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。

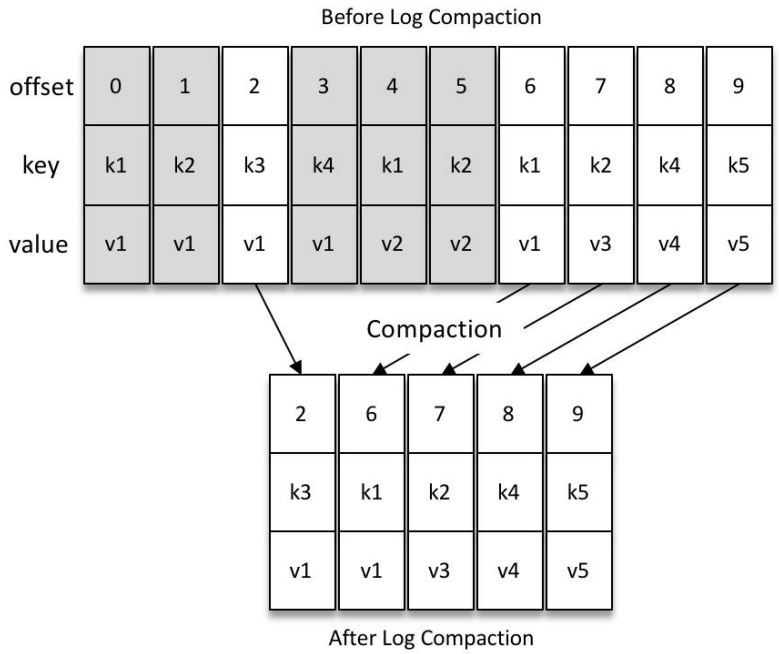
日志分段 2 的下一个日志偏移量的起始偏移量为 23，也小于 logStartOffset 的大小，将日志分段 2 加入 deletableSegments。

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

9.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。

10.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。

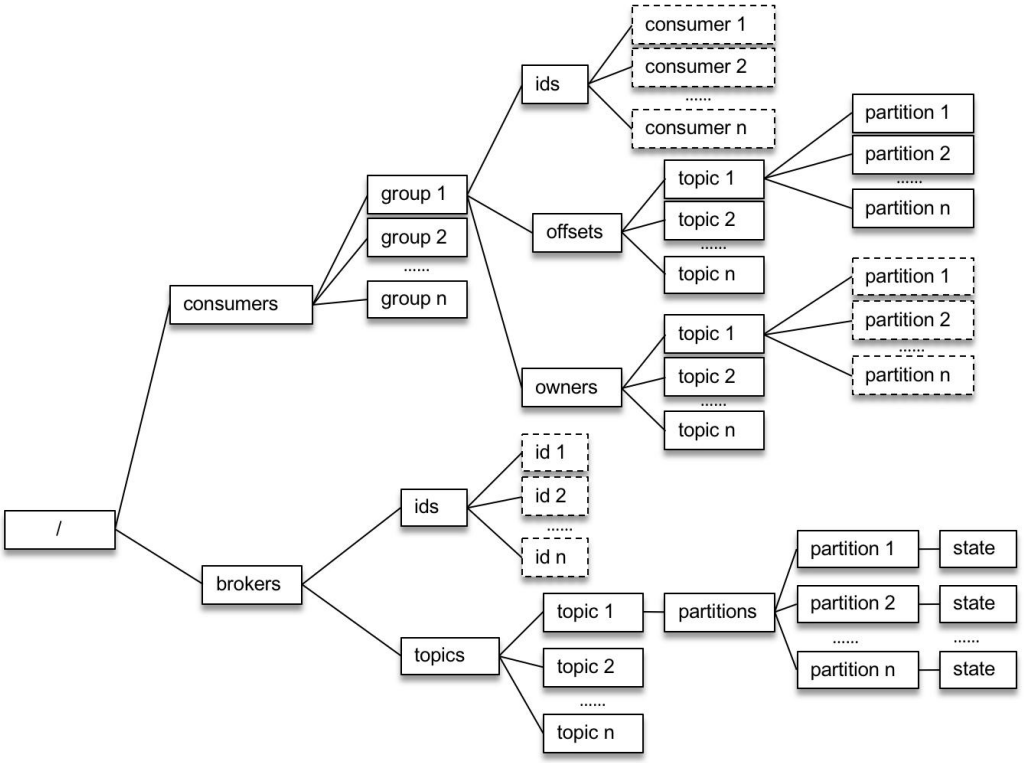
11.聊一聊 Kafka 的延时操作的原理

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。

12 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有 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。

13.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。

1. 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。
2. 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 broker 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。

14.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。

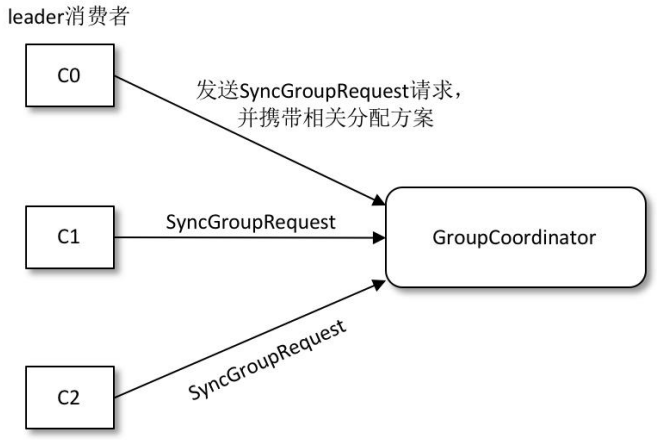
选举消费组的 leader
如果消费组内还没有 leader，那么第一个加入消费组的消费者即为消费组的 leader。如果某一时刻 leader 消费者由于某些原因退出了消费组，那么会重新选举一个新的 leader

选举分区分配策略

1. 收集各个消费者支持的所有分配策略，组成候选集 candidates。
2. 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。
3. 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。

15.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。

Kafka 高级篇

1.Kafka 中的事务是怎么实现的？

Kafka 中的事务可以使应用程序将消费消息、生产消息、提交消费位移当作原子操作来处理，同时成功或失败，即使该生产或消费会跨多个分区。

生产者必须提供唯一的 transactionalId，启动后请求事务协调器获取一个 PID，transactionalId 与 PID 一一对应。

每次发送数据给<Topic, Partition>前，需要先向事务协调器发送 AddPartitionsToTxnRequest，事务协调器会将该<Transaction, Topic, Partition>存于 __transaction_state 内，并将其状态置为 BEGIN。

在处理完 AddOffsetsToTxnRequest 之后，生产者还会发送 TxnOffsetCommitRequest 请求给 GroupCoordinator，从而将本次事务中包含的消费位移信息 offsets 存储到主题 __consumer_offsets 中

一旦上述数据写入操作完成，应用程序必须调用 KafkaProducer 的 commitTransaction 方法或者 abortTransaction 方法以结束当前事务。无论调用 commitTransaction() 方法还是 abortTransaction() 方法，生产者都会向 TransactionCoordinator 发送 EndTxnRequest 请求。
TransactionCoordinator 在收到 EndTxnRequest 请求后会执行如下操作：

1. 将 PREPARE_COMMIT 或 PREPARE_ABORT 消息写入主题 __transaction_state
2. 通过 WriteTxnMarkersRequest 请求将 COMMIT 或 ABORT 信息写入用户所使用的普通主题和 __consumer_offsets
3. 将 COMPLETE_COMMIT 或 COMPLETE_ABORT 信息写入内部主题 __transaction_state 标明该事务结束

在消费端有一个参数 isolation.level，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。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。

2.失效副本是指什么？有那些应对措施？

正常情况下，分区的所有副本都处于 ISR 集合中，但是难免会有异常情况发生，从而某些副本被剥离出 ISR 集合中。在 ISR 集合之外，也就是处于同步失效或功能失效（比如副本处于非存活状态）的副本统称为失效副本，失效副本对应的分区也就称为同步失效分区，即 under-replicated 分区。

Kafka 从 0.9.x 版本开始就通过唯一的 broker 端参数 replica.lag.time.max.ms 来抉择，当 ISR 集合中的一个 follower 副本滞后 leader 副本的时间超过此参数指定的值时则判定为同步失败，需要将此 follower 副本剔除出 ISR 集合。replica.lag.time.max.ms 参数的默认值为 10000。

在 0.9.x 版本之前，Kafka 中还有另一个参数 replica.lag.max.messages（默认值为 4000），它也是用来判定失效副本的，当一个 follower 副本滞后 leader 副本的消息数超过 replica.lag.max.messages 的大小时，则判定它处于同步失效的状态。它与 replica.lag.time.max.ms 参数判定出的失效副本取并集组成一个失效副本的集合，从而进一步剥离出分区的 ISR 集合。

Kafka 源码注释中说明了一般有这几种情况会导致副本失效：

follower 副本进程卡住，在一段时间内根本没有向 leader 副本发起同步请求，比如频繁的 Full GC。

follower 副本进程同步过慢，在一段时间内都无法追赶上 leader 副本，比如 I/O 开销过大。

如果通过工具增加了副本因子，那么新增加的副本在赶上 leader 副本之前也都是处于失效状态的。

如果一个 follower 副本由于某些原因（比如宕机）而下线，之后又上线，在追赶上 leader 副本之前也处于失效状态。

应对措施

我们用 UnderReplicatedPartitions 代表 leader 副本在当前 Broker 上且具有失效副本的分区的个数。

如果集群中有多个 Broker 的 UnderReplicatedPartitions 保持一个大于 0 的稳定值时，一般暗示着集群中有 Broker 已经处于下线状态。这种情况下，这个 Broker 中的分区个数与集群中的所有 UnderReplicatedPartitions（处于下线的 Broker 是不会上报任何指标值的）之和是相等的。通常这类问题是由于机器硬件原因引起的，但也有可能是由于操作系统或者 JVM 引起的 。

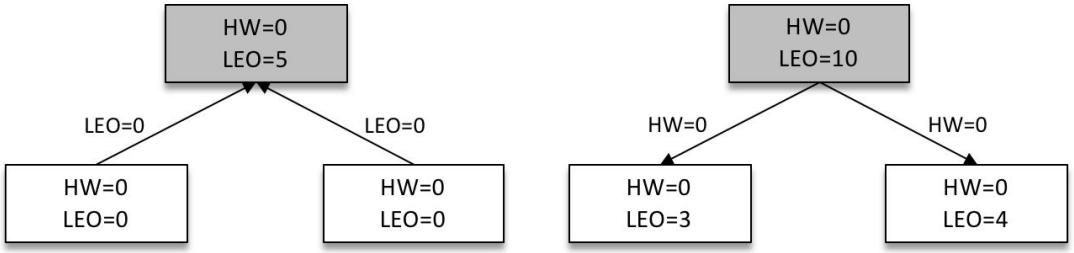
如果集群中存在 Broker 的 UnderReplicatedPartitions 频繁变动，或者处于一个稳定的大于 0 的值（这里特指没有 Broker 下线的情况）时，一般暗示着集群出现了性能问题，通常这类问题很难诊断，不过我们可以一步一步的将问题的范围缩小，比如先尝试确定这个性能问题是否只存在于集群的某个 Broker 中，还是整个集群之上。如果确定集群中所有的 under-replicated 分区都是在单个 Broker 上，那么可以看出这个 Broker 出现了问题，进而可以针对这单一的 Broker 做专项调查，比如：操作系统、GC、网络状态或者磁盘状态（比如：iowait、ioutil 等指标）。

3.多副本下，各个副本中的 HW 和 LEO 的演变过程

某个分区有 3 个副本分别位于 broker0、broker1 和 broker2 节点中，假设 broker0 上的副本 1 为当前分区的 leader 副本，那么副本 2 和副本 3 就是 follower 副本，整个消息追加的过程可以概括如下：

1. 生产者客户端发送消息至 leader 副本（副本 1）中。
2. 消息被追加到 leader 副本的本地日志，并且会更新日志的偏移量。
3. follower 副本（副本 2 和副本 3）向 leader 副本请求同步数据。
4. leader 副本所在的服务器读取本地日志，并更新对应拉取的 follower 副本的信息。
5. leader 副本所在的服务器将拉取结果返回给 follower 副本。
6. follower 副本收到 leader 副本返回的拉取结果，将消息追加到本地日志中，并更新日志的偏移量信息。

某一时刻，leader 副本的 LEO 增加至 5，并且所有副本的 HW 还都为 0。

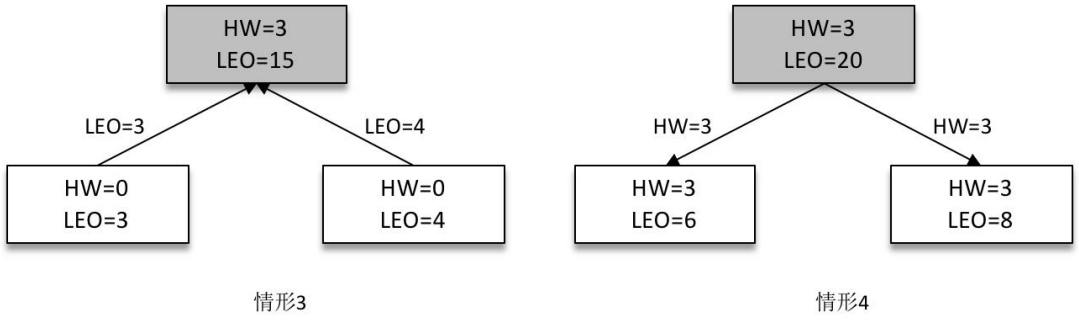


情形1（初识状态）　　情形2

之后 follower 副本（不带阴影的方框）向 leader 副本拉取消息，在拉取的请求中会带有自身的 LEO 信息,这个 LEO 信息对应的是 FetchRequest 请求中的 fetch_offset。leader 副本返回给 follower 副本相应的消息，并且还带有自身的 HW 信息，如上图（右）所示，这个 HW 信息对应的是 FetchResponse 中的 high_watermark。

此时两个 follower 副本各自拉取到了消息，并更新各自的 LEO 为 3 和 4。与此同时，follower 副本还会更新自己的 HW，更新 HW 的算法是比较当前 LEO 和 leader 副本中传送过来的 HW 的值，取较小值作为自己的 HW 值。当前两个 follower 副本的 HW 都等于 0（min(0,0) = 0）。

接下来 follower 副本再次请求拉取 leader 副本中的消息，如下图（左）所示。



此时 leader 副本收到来自 follower 副本的 FetchRequest 请求，其中带有 LEO 的相关信息，选取其中的最小值作为新的 HW，即 min(15,3,4)=3。然后连同消息和 HW 一起返回 FetchResponse 给 follower 副本，如上图（右）所示。注意 leader 副本的 HW 是一个很重要的东西，因为它直接影响了分区数据对消费者的可见性。

两个 follower 副本在收到新的消息之后更新 LEO 并且更新自己的 HW 为 3（min(LEO,3)=3）。

4.Kafka 在可靠性方面做了哪些改进？（HW, LeaderEpoch）

HW

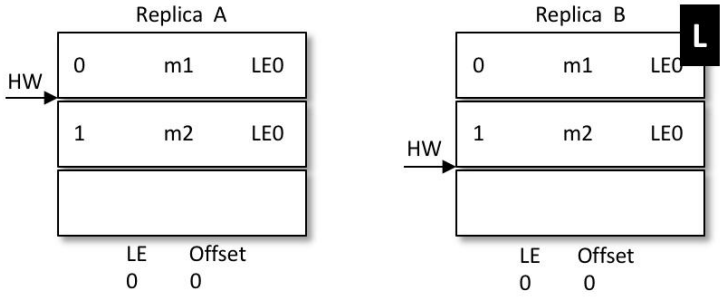
HW 是 High Watermark 的缩写，俗称高水位，它标识了一个特定的消息偏移量（offset），消费者只能拉取到这个 offset 之前的消息。

分区 ISR 集合中的每个副本都会维护自身的 LEO，而 ISR 集合中最小的 LEO 即为分区的 HW，对消费者而言只能消费 HW 之前的消息。

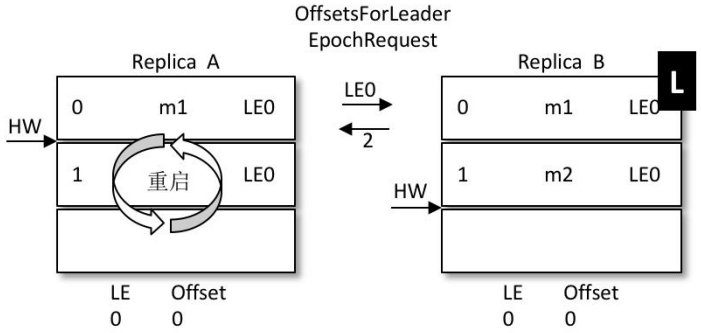
leader epoch

leader epoch 代表 leader 的纪元信息（epoch），初始值为 0。每当 leader 变更一次，leader epoch 的值就会加 1，相当于为 leader 增设了一个版本号。
每个副本中还会增设一个矢量 <LeaderEpoch => StartOffset>，其中 StartOffset 表示当前 LeaderEpoch 下写入的第一条消息的偏移量。

假设有两个节点 A 和 B，B 是 leader 节点，里面的数据如图：

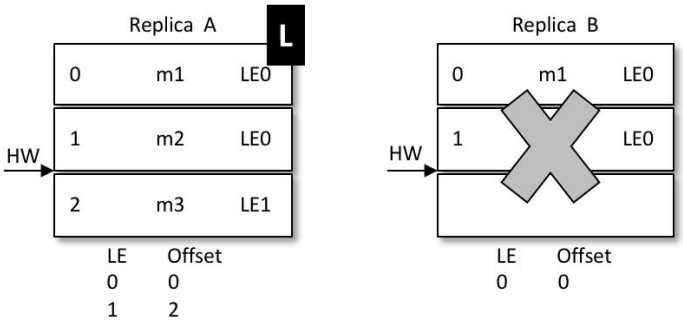


A 发生重启，之后 A 不是先忙着截断日志而是先发送 OffsetsForLeaderEpochRequest 请求给 B，B 作为目前的 leader 在收到请求之后会返回当前的 LEO（LogEndOffset，注意图中 LEO 和 LEO 的不同），与请求对应的响应为 OffsetsForLeaderEpochResponse。如果 A 中的 LeaderEpoch（假设为 LE_A）和 B 中的不相同，那么 B 此时会查找 LeaderEpoch 为 LE_A+1 对应的 StartOffset 并返回给 A

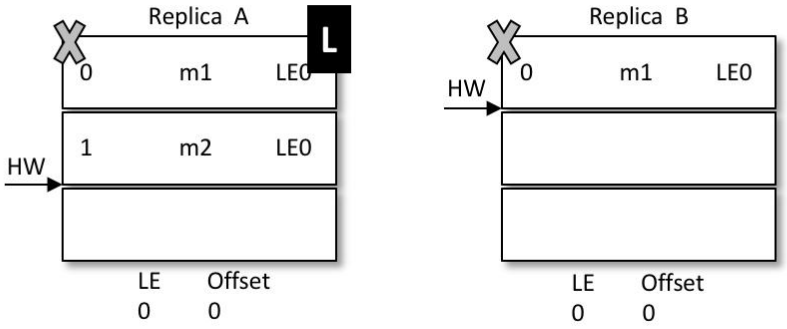


如上图所示，A 在收到 2 之后发现和目前的 LEO 相同，也就不需要截断日志了，以此来保护数据的完整性。

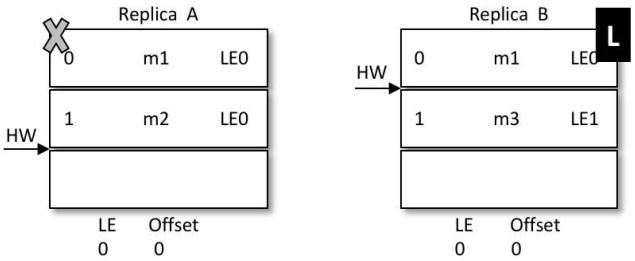
再如，之后 B 发生了宕机，A 成为新的 leader，那么对应的 LE=0 也变成了 LE=1，对应的消息 m2 此时就得到了保留。后续的消息都可以以 LE1 为 LeaderEpoch 陆续追加到 A 中。这个时候 A 就会有两个 LE，第二 LE 所记录的 Offset 从 2 开始。如果 B 恢复了，那么就会从 A 中获取到 LE+1 的 Offset 为 2 的值返回给 B。



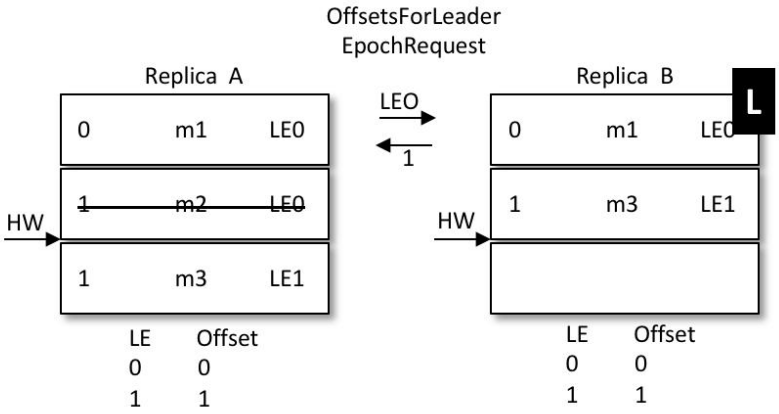
再来看看 LE 如何解决数据不一致的问题：
当前 A 为 leader，B 为 follower，A 中有 2 条消息 m1 和 m2，而 B 中有 1 条消息 m1。假设 A 和 B 同时“挂掉”，然后 B 第一个恢复过来并成为新的 leader。



之后 B 写入消息 m3，并将 LEO 和 HW 更新至 2，如下图所示。注意此时的 LeaderEpoch 已经从 LEO 增至 LE1 了。



紧接着 A 也恢复过来成为 follower 并向 B 发送 OffsetsForLeaderEpochRequest 请求，此时 A 的 LeaderEpoch 为 LEO。B 根据 LEO 查询到对应的 offset 为 1 并返回给 A，A 就截断日志并删除了消息 m2，如下图所示。之后 A 发送 FetchRequest 至 B 请求来同步数据，最终 A 和 B 中都有两条消息 m1 和 m3，HW 和 LEO 都为 2，并且 LeaderEpoch 都为 LE1，如此便解决了数据不一致的问题。



5.为什么 Kafka 不支持读写分离？

因为这样有两个明显的缺点：

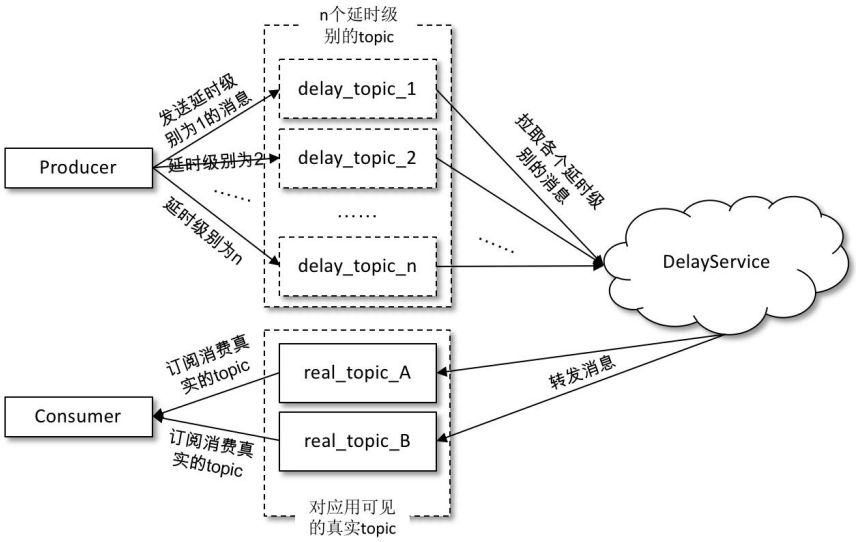
1. 数据一致性问题。数据从主节点转到从节点必然会有一个延时的时间窗口，这个时间窗口会导致主从节点之间的数据不一致。
2. 延时问题。数据从写入主节点到同步至从节点中的过程需要经历网络→主节点内存→主节点磁盘→网络→从节点内存→从节点磁盘这几个阶段。对延时敏感的应用而言，主写从读的功能并不太适用。

对于 Kafka 来说，必要性不是很高，因为在 Kafka 集群中，如果存在多个副本，经过合理的配置，可以让 leader 副本均匀的分布在各个 broker 上面，使每个 broker 上的读写负载都是一样的。

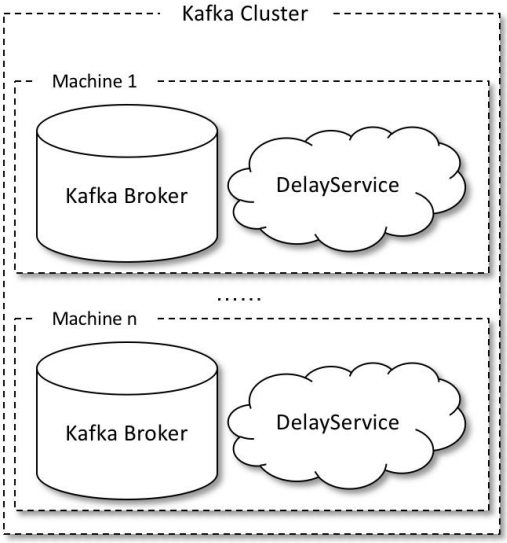
6.Kafka 中的延迟队列怎么实现

在发送延时消息的时候并不是先投递到要发送的真实主题（real_topic）中，而是先投递到一些 Kafka 内部的主题（delay_topic）中，这些内部主题对用户不可见，然后通过一个自定义的服务拉取这些内部主题中的消息，并将满足条件的消息再投递到要发送的真实的主题中，消费者所订阅的还是真实的主题。

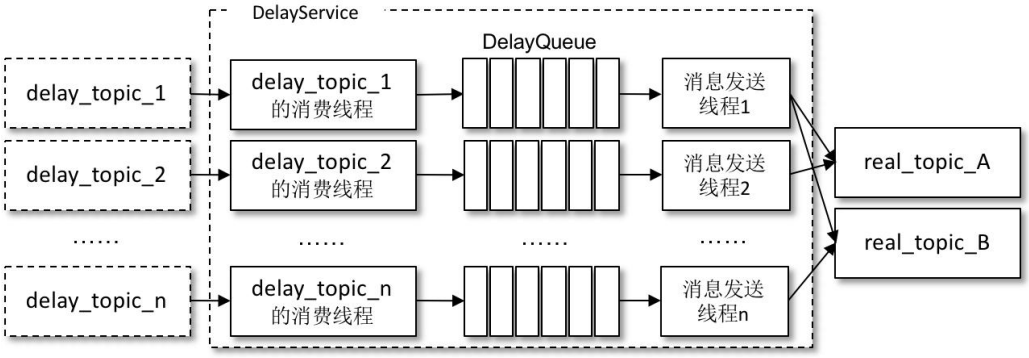
如果采用这种方案，那么一般是按照不同的延时等级来划分的，比如设定 5s、10s、30s、1min、2min、5min、10min、20min、30min、45min、1hour、2hour 这些按延时时间递增的延时等级，延时的消息按照延时时间投递到不同等级的主题中，投递到同一主题中的消息的延时时间会被强转为与此主题延时等级一致的延时时间，这样延时误差控制在两个延时等级的时间差范围之内（比如延时时间为 17s 的消息投递到 30s 的延时主题中，之后按照延时时间为 30s 进行计算，延时误差为 13s）。虽然有一定的延时误差，但是误差可控，并且这样只需增加少许的主题就能实现延时队列的功能。



发送到内部主题（delay_topic_*）中的消息会被一个独立的 DelayService 进程消费，这个 DelayService 进程和 Kafka broker 进程以一对一的配比进行同机部署（参考下图），以保证服务的可用性。



针对不同延时级别的主题，在 DelayService 的内部都会有单独的线程来进行消息的拉取，以及单独的 DelayQueue（这里用的是 JUC 中 DelayQueue）进行消息的暂存。与此同时，在 DelayService 内部还会有专门的消息发送线程来获取 DelayQueue 的消息并转发到真实的主题中。从消费、暂存再到转发，线程之间都是一一对应的关系。如下图所示，DelayService 的设计应当尽量保持简单，避免锁机制产生的隐患。



为了保障内部 DelayQueue 不会因为未处理的消息过多而导致内存的占用过大，DelayService 会对主题中的每个分区进行计数，当达到一定的阈值之后，就会暂停拉取该分区中的消息。

因为一个主题中一般不止一个分区，分区之间的消息并不会按照投递时间进行排序，DelayQueue 的作用是将消息按照再次投递时间进行有序排序，这样下游的消息发送线程就能够按照先后顺序获取最先满足投递条件的消息。

7.Kafka 中怎么实现死信队列和重试队列？

死信可以看作消费者不能处理收到的消息，也可以看作消费者不想处理收到的消息，还可以看作不符合处理要求的消息。比如消息内包含的消息内容无法被消费者解析，为了确保消息的可靠性而不被随意丢弃，故将其投递到死信队列中，这里的死信就可以看作消费者不能处理的消息。再比如超过既定的重试次数之后将消息投入死信队列，这里就可以将死信看作不符合处理要求的消息。

重试队列其实可以看作一种回退队列，具体指消费端消费消息失败时，为了防止消息无故丢失而重新将消息回滚到 broker 中。与回退队列不同的是，重试队列一般分成多个重试等级，每个重试等级一般也会设置重新投递延时，重试次数越多投递延时就越大。

理解了他们的概念之后我们就可以为每个主题设置重试队列，消息第一次消费失败入重试队列 Q1，Q1 的重新投递延时为 5s，5s 过后重新投递该消息；如果消息再次消费失败则入重试队列 Q2，Q2 的重新投递延时为 10s，10s 过后再次投递该消息。

然后再设置一个主题作为死信队列，重试越多次重新投递的时间就越久，并且需要设置一个上限，超过投递次数就进入死信队列。重试队列与延时队列有相同的地方，都需要设置延时级别。

8.Kafka 中怎么做消息审计？

消息审计是指在消息生产、存储和消费的整个过程之间对消息个数及延迟的审计，以此来检测是否有数据丢失、是否有数据重复、端到端的延迟又是多少等内容。

目前与消息审计有关的产品也有多个，比如 Chaperone（Uber）、Confluent Control Center、Kafka Monitor（LinkedIn），它们主要通过在消息体（value 字段）或在消息头（headers 字段）中内嵌消息对应的时间戳 timestamp 或全局的唯一标识 ID（或者是两者兼备）来实现消息的审计功能。

内嵌 timestamp 的方式主要是设置一个审计的时间间隔 time_bucket_interval（可以自定义设置几秒或几分钟），根据这个 time_bucket_interval 和消息所属的 timestamp 来计算相应的时间桶（time_bucket）。

内嵌 ID 的方式就更加容易理解了，对于每一条消息都会被分配一个全局唯一标识 ID。如果主题和相应的分区固定，则可以为每个分区设置一个全局的 ID。当有消息发送时，首先获取对应的 ID，然后内嵌到消息中，最后才将它发送到 broker 中。消费者进行消费审计时，可以判断出哪条消息丢失、哪条消息重复。

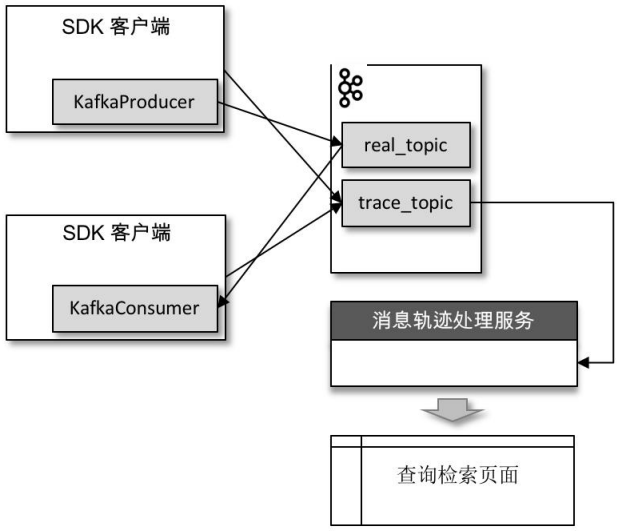
9.Kafka 中怎么做消息轨迹？

消息轨迹指的是一条消息从生产者发出，经由 broker 存储，再到消费者消费的整个过程中，各个相关节点的状态、时间、地点等数据汇聚而成的完整链路信息。生产者、broker、消费

者这 3 个角色在处理消息的过程中都会在链路中增加相应的信息，将这些信息汇聚、处理之后就可以查询任意消息的状态，进而为生产环境中的故障排除提供强有力的数据支持。

对消息轨迹而言，最常见的实现方式是封装客户端，在保证正常生产消费的同时添加相应的轨迹信息埋点逻辑。无论生产，还是消费，在执行之后都会有相应的轨迹信息，我们需要将这些信息保存起来。

我们同样可以将轨迹信息保存到 Kafka 的某个主题中，比如下图中的主题 trace_topic。



生产者在将消息正常发送到用户主题 real_topic 之后（或者消费者在拉取到消息消费之后）会将轨迹信息发送到主题 trace_topic 中。

10. 怎么计算 Lag？（注意 read_uncommitted 和 read_committed 状态下的不同)#

如果消费者客户端的 isolation.level 参数配置为“read_uncommitted”（默认）,它对应的 Lag 等于 HW - ConsumerOffset 的值，其中 ConsumerOffset 表示当前的消费位移。

如果这个参数配置为“read_committed”，那么就要引入 LSO 来进行计算了。LSO 是 LastStableOffset 的缩写,它对应的 Lag 等于 LSO - ConsumerOffset 的值。

- 首先通过 DescribeGroupsRequest 请求获取当前消费组的元数据信息，当然在这之前还会通过 FindCoordinatorRequest 请求查找消费组对应的 GroupCoordinator。
- 接着通过 OffsetFetchRequest 请求获取消费位移 ConsumerOffset。
- 然后通过 KafkaConsumer 的 endOffsets(Collection partitions)方法（对应于 ListOffsetRequest 请求）获取 HW（LSO）的值。
- 最后通过 HW 与 ConsumerOffset 相减得到分区的 Lag，要获得主题的总体 Lag 只需对旗下的各个分区累加即可。

11.Kafka 有哪些指标需要着重关注？

比较重要的 Broker 端 JMX 指标：

- BytesIn/BytesOut：即 Broker 端每秒入站和出站字节数。你要确保这组值不要接近你的网络带宽，否则这通常都表示网卡已被“打满”，很容易出现网络丢包的情形。
- NetworkProcessorAvgIdlePercent：即网络线程池线程平均的空闲比例。通常来说，你应该确保这个 JMX 值长期大于 30%。如果小于这个值，就表明你的网络线程池非常繁忙，你需要通过增加网络线程数或将负载转移给其他服务器的方式，来给该 Broker 减负。
- RequestHandlerAvgIdlePercent：即 I/O 线程池线程平均的空闲比例。同样地，如果该值长期小于 30%，你需要调整 I/O 线程池的数量，或者减少 Broker 端的负载。
- UnderReplicatedPartitions：即未充分备份的分区数。所谓未充分备份，是指并非所有的 Follower 副本都和 Leader 副本保持同步。一旦出现了这种情况，通常都表明该分区有可能会出现数据丢失。因此，这是一个非常重要的 JMX 指标。
- ISRShrink/ISRExpand：即 ISR 收缩和扩容的频次指标。如果你的环境中出现 ISR 中副本频繁进出的情形，那么这组值一定是很高的。这时，你要诊断下副本频繁进出 ISR 的原因，并采取适当的措施。
- ActiveControllerCount：即当前处于激活状态的控制器的数量。正常情况下，Controller 所在 Broker 上的这个 JMX 指标值应该是 1，其他 Broker 上的这个值是 0。如果你发现存在多台 Broker 上该值都是 1 的情况，一定要赶快处理，处理方式主要是查看网络连通性。这种情况通常表明集群出现了脑裂。脑裂问题是非常严重的分布式故障，Kafka 目前依托 ZooKeeper 来防止脑裂。但一旦出现脑裂，Kafka 是无法保证正常工作的。

12.Kafka 的那些设计让它有如此高的性能？

1.分区

kafka 是个分布式集群的系统，整个系统可以包含多个 broker，也就是多个服务器实例。每个主题 topic 会有多个分区，kafka 将分区均匀地分配到整个集群中，当生产者向对应主题传递消息，消息通过负载均衡机制传递到不同的分区以减轻单个服务器实例的压力。

一个 Consumer Group 中可以有多个 consumer，多个 consumer 可以同时消费不同分区的消息，大大的提高了消费者的并行消费能力。但是一个分区中的消息只能被一个 Consumer Group 中的一个 consumer 消费。

2.网络传输上减少开销

批量发送：
在发送消息的时候，kafka 不会直接将少量数据发送出去，否则每次发送少量的数据会增加网络传输频率，降低网络传输效率。kafka 会先将消息缓存在内存中，当超过一个的大小或者超过一定的时间，那么会将这些消息进行批量发送。
端到端压缩：
当然网络传输时数据量小也可以减小网络负载，kafaka 会将这些批量的数据进行压缩，将一批消息打包后进行压缩，发送 broker 服务器后，最终这些数据还是提供给消费者用，所以数据在服务器上还是保持压缩状态，不会进行解压，而且频繁的压缩和解压也会降低性能，最终还是以压缩的方式传递到消费者的手上。

3.顺序读写

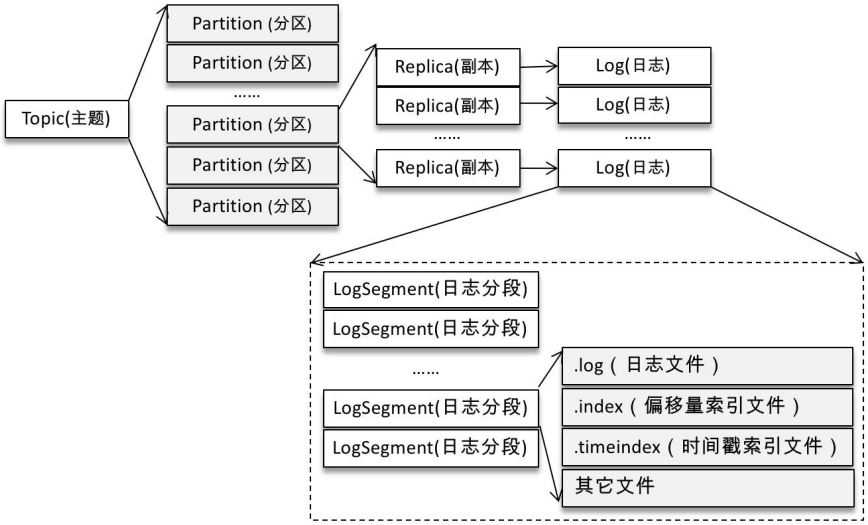
kafka 将消息追加到日志文件中，利用了磁盘的顺序读写，来提高读写效率。

4.零拷贝技术

零拷贝将文件内容从磁盘通过 DMA 引擎复制到内核缓冲区，而且没有把数据复制到 socket 缓冲区，只是将数据位置和长度信息的描述符复制到了 socket 缓存区，然后直接将数据传输到网络接口，最后发送。这样大大减小了拷贝的次数，提高了效率。kafka 正是调用 linux 系统给出的 sendfile 系统调用来使用零拷贝。Java 中的系统调用给出的是 FileChannel.transferTo 接口。

5. 优秀的文件存储机制

如果分区规则设置得合理，那么所有的消息可以均匀地分布到不同的分区中，这样就可以实现水平扩展。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，这样也便于消息的维护和清理。



Kafka 中的索引文件以稀疏索引（sparse index）的方式构造消息的索引，它并不保证每个消息在索引文件中都有对应的索引项。每当写入一定量（由 broker 端参数 log.index.interval.bytes 指定，默认值为 4096，即 4KB）的消息时，偏移量索引文件和时间戳索引文件分别增加一个偏移量索引项和时间戳索引项，增大或减小 log.index.interval.bytes 的值，对应地可以增加或缩小索引项的密度。