二

16 揭开神秘的“位移主题”面纱

你好，我是胡夕。今天我要和你分享的内容是：Kafka 中神秘的内部主题（Internal Topic）__consumer_offsets。

__consumer_offsets 在 Kafka 源码中有个更为正式的名字，叫**位移主题**，即 Offsets Topic。为了方便今天的讨论，我将统一使用位移主题来指代 __consumer_offsets。需要注意的是，它有两个下划线哦。

好了，我们开始今天的内容吧。首先，我们有必要探究一下位移主题被引入的背景及原因，即位移主题的前世今生。

在上一期中，我说过老版本 Consumer 的位移管理是依托于 Apache ZooKeeper 的，它会自动或手动地将位移数据提交到 ZooKeeper 中保存。当 Consumer 重启后，它能自动从 ZooKeeper 中读取位移数据，从而在上次消费截止的地方继续消费。这种设计使得 Kafka Broker 不需要保存位移数据，减少了 Broker 端需要持有的状态空间，因而有利于实现高伸缩性。

但是，ZooKeeper 其实并不适用于这种高频的写操作，因此，Kafka 社区自 0.8.2.x 版本开始，就在酝酿修改这种设计，并最终在新版本 Consumer 中正式推出了全新的位移管理机制，自然也包括这个新的位移主题。

新版本 Consumer 的位移管理机制其实也很简单，就是**将 Consumer 的位移数据作为一条条普通的 Kafka 消息，提交到 __consumer_offsets 中。可以这么说，__consumer_offsets 的主要作用是保存 Kafka 消费者的位移信息。**它要求这个提交过程不仅要实现高持久性，还要支持高频的写操作。显然，Kafka 的主题设计天然就满足这两个条件，因此，使用 Kafka 主题来保存位移这件事情，实际上就是一个水到渠成的想法了。

这里我想再次强调一下，和你创建的其他主题一样，位移主题就是普通的 Kafka 主题。你可以手动地创建它、修改它，甚至是删除它。只不过，它同时也是一个内部主题，大部分情况下，你其实并不需要“搭理”它，也不用花心思去管理它，把它丢给 Kafka 就完事了。

虽说位移主题是一个普通的 Kafka 主题，但**它的消息格式却是 Kafka 自己定义的**，用户不能修改，也就是说你不能随意地向这个主题写消息，因为一旦你写入的消息不满足 Kafka

规定的格式，那么 Kafka 内部无法成功解析，就会造成 Broker 的崩溃。事实上，Kafka Consumer 有 API 帮你提交位移，也就是向位移主题写消息。你千万不要自己写个 Producer 随意向该主题发送消息。

你可能会好奇，这个主题存的到底是什么格式的消息呢？所谓的消息格式，你可以简单地理解为是一个 KV 对。Key 和 Value 分别表示消息的键值和消息体，在 Kafka 中它们就是字节数组而已。想象一下，如果让你来设计这个主题，你觉得消息格式应该长什么样子呢？我先不说社区的设计方案，我们自己先来设计一下。

首先从 Key 说起。一个 Kafka 集群中的 Consumer 数量会有很多，既然这个主题保存的是 Consumer 的位移数据，那么消息格式中必须要有字段来标识这个位移数据是哪个 Consumer 的。这种数据放在哪个字段比较合适呢？显然放在 Key 中比较合适。

现在我们知道该主题消息的 Key 中应该保存标识 Consumer 的字段，那么，当前 Kafka 中什么字段能够标识 Consumer 呢？还记得之前我们说 Consumer Group 时提到的 Group ID 吗？没错，就是这个字段，它能够标识唯一的 Consumer Group。

说到这里，我再多说几句。除了 Consumer Group，Kafka 还支持独立 Consumer，也称 Standalone Consumer。它的运行机制与 Consumer Group 完全不同，但是位移管理的机制却是相同的。因此，即使是 Standalone Consumer，也有自己的 Group ID 来标识它自己，所以也适用于这套消息格式。

Okay，我们现在知道 Key 中保存了 Group ID，但是只保存 Group ID 就可以了吗？别忘了，Consumer 提交位移是在分区层面上进行的，即它提交的是某个或某些分区的位移，那么很显然，Key 中还应该保存 Consumer 要提交位移的分区。

好了，我们来总结一下我们的结论。**位移主题的 Key 中应该保存 3 部分内容：<Group ID，主题名，分区号 >**。如果你认同这样的结论，那么恭喜你，社区就是这么设计的！

接下来，我们再来看看消息体的设计。也许你会觉得消息体应该很简单，保存一个位移值就可以了。实际上，社区的方案要复杂得多，比如消息体还保存了位移提交的一些其他元数据，诸如时间戳和用户自定义的数据等。保存这些元数据是为了帮助 Kafka 执行各种各样后续的操作，比如删除过期位移消息等。但总体来说，我们还是可以简单地认为消息体就是保存了位移值。

当然了，位移主题的消息格式可不是只有这一种。事实上，它有 3 种消息格式。除了刚刚我们说的这种格式，还有 2 种格式：

1. 用于保存 Consumer Group 信息的消息。
2. 用于删除 Group 过期位移甚至是删除 Group 的消息。

第 1 种格式非常神秘，以至于你几乎无法在搜索引擎中搜到它的身影。不过，你只需要记住它是用来注册 Consumer Group 的就可以了。

第 2 种格式相对更加有名一些。它有个专属的名字：tombstone 消息，即墓碑消息，也称 delete mark。下次你在 Google 或百度中见到这些词，不用感到惊讶，它们指的是一个东西。这些消息只出现在源码中而不暴露给你。它的主要特点是它的消息体是 null，即空消息体。

那么，何时会写入这类消息呢？一旦某个 Consumer Group 下的所有 Consumer 实例都停止了，而且它们的位移数据都已被删除时，Kafka 会向位移主题的对应分区写入 tombstone 消息，表明要彻底删除这个 Group 的信息。

好了，消息格式就说这么多，下面我们来说说位移主题是怎么被创建的。通常来说，**当 Kafka 集群中的第一个 Consumer 程序启动时，Kafka 会自动创建位移主题。**我们说过，位移主题就是普通的 Kafka 主题，那么它自然也有对应的分区数。但如果是 Kafka 自动创建的，分区数是怎么设置的呢？这就要看 Broker 端参数 offsets.topic.num.partitions 的取值了。它的默认值是 50，因此 Kafka 会自动创建一个 50 分区的位移主题。如果你曾经惊讶于 Kafka 日志路径下冒出很多 __consumer_offsets-xxx 这样的目录，那么现在应该明白了吧，这就是 Kafka 自动帮你创建的位移主题啊。

你可能会问，除了分区数，副本数或备份因子是怎么控制的呢？答案也很简单，这就是 Broker 端另一个参数 offsets.topic.replication.factor 要做的事情了。它的默认值是 3。

总结一下，**如果位移主题是 Kafka 自动创建的，那么该主题的分区数是 50，副本数是 3。**

当然，你也可以选择手动创建位移主题，具体方法就是，在 Kafka 集群尚未启动任何 Consumer 之前，使用 Kafka API 创建它。手动创建的好处在于，你可以创建满足你实际场景需要的位移主题。比如很多人说 50 个分区对我来讲太多了，我不想要这么多分区，那么你可以自己创建它，不用理会 offsets.topic.num.partitions 的值。

不过我给你的建议是，还是让 Kafka 自动创建比较好。目前 Kafka 源码中有一些地方硬编码了 50 分区数，因此如果你自行创建了一个不同于默认分区数的位移主题，可能会碰到各种各种奇怪的问题。这是社区的一个 bug，目前代码已经修复了，但依然在审核中。

创建位移主题当然是为了用的，那么什么地方会用到位移主题呢？我们前面一直在说 Kafka Consumer 提交位移时会写入该主题，那 Consumer 是怎么提交位移的呢？目前 Kafka Consumer 提交位移的方式有两种：**自动提交位移和手动提交位移。**

Consumer 端有个参数叫 enable.auto.commit，如果值是 true，则 Consumer 在后台默默地为你定期提交位移，提交间隔由一个专属的参数 auto.commit.interval.ms 来控制。自动提交位移有一个显著的优点，就是省事，你不用操心位移提交的事情，就能保证消息消费不

会丢失。但这一点同时也是缺点。因为它太省事了，以至于丧失了很大的灵活性和可控性，你完全没法把控 Consumer 端的位移管理。

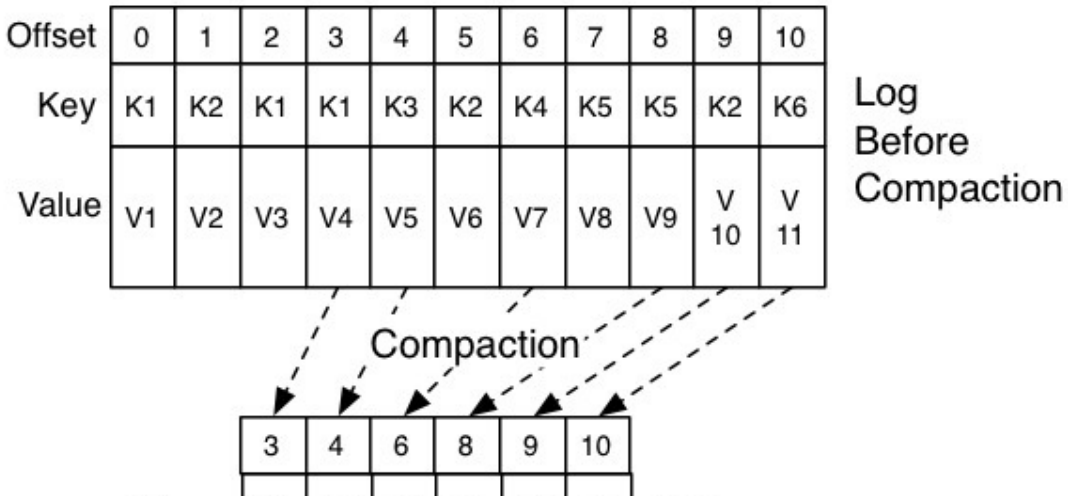
事实上，很多与 Kafka 集成的大数据框架都是禁用自动提交位移的，如 Spark、Flink 等。这就引出了另一种位移提交方式：**手动提交位移**，即设置 enable.auto.commit = false。一旦设置了 false，作为 Consumer 应用开发的你就要承担起位移提交的责任。Kafka Consumer API 为你提供了位移提交的方法，如 consumer.commitSync 等。当调用这些方法时，Kafka 会向位移主题写入相应的消息。

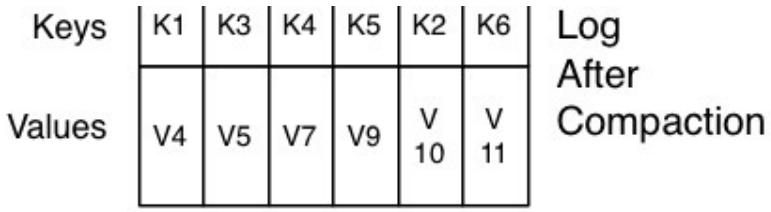
如果你选择的是自动提交位移，那么就可能存在一个问题：只要 Consumer 一直启动着，它就会无限期地向位移主题写入消息。

我们来举个极端一点的例子。假设 Consumer 当前消费到了某个主题的最新一条消息，位移是 100，之后该主题没有任何新消息产生，故 Consumer 无消息可消费了，所以位移永远保持在 100。由于是自动提交位移，位移主题中会不停地写入位移 =100 的消息。显然 Kafka 只需要保留这类消息中的最新一条就可以了，之前的消息都是可以删除的。这就要求 Kafka 必须要有针对位移主题消息特点的消息删除策略，否则这种消息会越来越多，最终撑爆整个磁盘。

Kafka 是怎么删除位移主题中的过期消息的呢？答案就是 Compaction。国内很多文献都将其翻译成压缩，我个人是有一点保留意见的。在英语中，压缩的专有术语是 Compression，它的原理和 Compaction 很不相同，我更倾向于翻译成压实，或干脆采用 JVM 垃圾回收中的术语：整理。

不管怎么翻译，Kafka 使用**Compact 策略**来删除位移主题中的过期消息，避免该主题无限期膨胀。那么应该如何定义 Compact 策略中的过期呢？对于同一个 Key 的两条消息 M1 和 M2，如果 M1 的发送时间早于 M2，那么 M1 就是过期消息。Compact 的过程就是扫描日志的所有消息，剔除那些过期的消息，然后把剩下的消息整理在一起。我在这里贴一张来自官网的图片，来说明 Compact 过程。





图中位移为 0、2 和 3 的消息的 Key 都是 K1。Compact 之后，分区只需要保存位移为 3 的消息，因为它是最新发送的。

Kafka 提供了专门的后台线程定期地巡检待 Compact 的主题，看看是否存在满足条件的可删除数据。这个后台线程叫 Log Cleaner。很多实际生产环境中都出现过位移主题无限膨胀占用过多磁盘空间的问题，如果你的环境中也有这个问题，我建议你去检查一下 Log Cleaner 线程的状态，通常都是这个线程挂掉了导致的。

小结

总结一下，今天我跟你分享了 Kafka 神秘的位移主题 __consumer_offsets，包括引入它的契机与原因、它的作用、消息格式、写入的时机以及管理策略等，这对我们了解 Kafka 特别是 Kafka Consumer 的位移管理是大有帮助的。实际上，将很多元数据以消息的方式存入 Kafka 内部主题的做法越来越流行。除了 Consumer 位移管理，Kafka 事务也是利用了这个方法，当然那是另外的一个内部主题了。

社区的想法很简单：既然 Kafka 天然实现了高持久性和高吞吐量，那么任何有这两个需求的子服务自然也就不必求助于外部系统，用 Kafka 自己实现就好了。

上一页 下一页