二

17 消费者组重平衡能避免吗?

你好，我是胡夕。今天我要和你分享的内容是：消费者组重平衡能避免吗?

其实在专栏[第 15 期]中，我们讲过重平衡，也就是 Rebalance，现在先来回顾一下这个概念的原理和用途。Rebalance 就是让一个 Consumer Group 下所有的 Consumer 实例就如何消费订阅主题的所有分区达成共识的过程。在 Rebalance 过程中，所有 Consumer 实例共同参与，在协调者组件的帮助下，完成订阅主题分区的分配。但是，在整个过程中，所有实例都不能消费任何消息，因此它对 Consumer 的 TPS 影响很大。

你可能会对这里提到的“协调者”有些陌生，我来简单介绍下。所谓协调者，在 Kafka 中对应的术语是 Coordinator，它专门为 Consumer Group 服务，负责为 Group 执行 Rebalance 以及提供位移管理和组成员管理等。

具体来讲，Consumer 端应用程序在提交位移时，其实是向 Coordinator 所在的 Broker 提交位移。同样地，当 Consumer 应用启动时，也是向 Coordinator 所在的 Broker 发送各种请求，然后由 Coordinator 负责执行消费者组的注册、成员管理记录等元数据管理操作。

所有 Broker 在启动时，都会创建和开启相应的 Coordinator 组件。也就是说，**所有 Broker 都有各自的 Coordinator 组件**。那么，Consumer Group 如何确定为它服务的 Coordinator 在哪台 Broker 上呢？答案就在我们之前说过的 Kafka 内部位移主题 __consumer_offsets 身上。

目前，Kafka 为某个 Consumer Group 确定 Coordinator 所在的 Broker 的算法有 2 个步骤。

第 1 步：确定由位移主题的哪个分区来保存该 Group 数据：partitionId=Math.abs(groupId.hashCode() % offsetsTopicPartitionCount)。

第 2 步：找出该分区 Leader 副本所在的 Broker，该 Broker 即为对应的 Coordinator。

简单解释一下上面的算法。首先，Kafka 会计算该 Group 的 group.id 参数的哈希值。比如你有个 Group 的 group.id 设置成了“test-group”，那么它的 hashCode 值就应该是 627841412。其次，Kafka 会计算 __consumer_offsets 的分区数，通常是 50 个分区，之后将刚才那个哈希值对分区数进行取模加求绝对值计算，即 abs(627841412 % 50) = 12。

此时，我们就知道了位移主题的分区 12 负责保存这个 Group 的数据。有了分区号，算法的第 2 步就变得很简单了，我们只需要找出位移主题分区 12 的 Leader 副本在哪个 Broker 上就可以了。这个 Broker，就是我们要找的 Coordinator。

在实际使用过程中，Consumer 应用程序，特别是 Java Consumer API，能够自动发现并连接正确的 Coordinator，我们不用操心这个问题。知晓这个算法的最大意义在于，它能够帮助我们解决**定位问题**。当 Consumer Group 出现问题，需要快速排查 Broker 端日志时，我们能够根据这个算法准确定位 Coordinator 对应的 Broker，不必一台 Broker 一台 Broker 地盲查。

好了，我们说回 Rebalance。既然我们今天要讨论的是如何避免 Rebalance，那就说明 Rebalance 这个东西不好，或者说至少有一些弊端需要我们去规避。那么，Rebalance 的弊端是什么呢？总结起来有以下 3 点：

1. Rebalance 影响 Consumer 端 TPS。这个之前也反复提到了，这里就不再具体讲了。总之就是，在 Rebalance 期间，Consumer 会停下手头的事情，什么也干不了。
2. Rebalance 很慢。如果你的 Group 下成员很多，就一定会有这样的痛点。还记得我曾经举过的那个国外用户的例子吧？他的 Group 下有几百个 Consumer 实例，Rebalance 一次要几个小时。在那种场景下，Consumer Group 的 Rebalance 已经完全失控了。
3. Rebalance 效率不高。当前 Kafka 的设计机制决定了每次 Rebalance 时，Group 下的所有成员都要参与进来，而且通常不会考虑局部性原理，但局部性原理对提升系统性能是特别重要的。

关于第 3 点，我们来举个简单的例子。比如一个 Group 下有 10 个成员，每个成员平均消费 5 个分区。假设现在有一个成员退出了，此时就需要开启新一轮的 Rebalance，把这个成员之前负责的 5 个分区“转移”给其他成员。显然，比较好的做法是维持当前 9 个成员消费分区的方案不变，然后将 5 个分区随机分配给这 9 个成员，这样能最大限度地减少 Rebalance 对剩余 Consumer 成员的冲击。

遗憾的是，目前 Kafka 并不是这样设计的。在默认情况下，每次 Rebalance 时，之前的分配方案都不会被保留。就拿刚刚这个例子来说，当 Rebalance 开始时，Group 会打散这 50 个分区（10 个成员 * 5 个分区），由当前存活的 9 个成员重新分配它们。显然这不是效率很高的做法。基于这个原因，社区于 0.11.0.0 版本推出了 StickyAssignor，即有粘性的分区分配策略。所谓的有粘性，是指每次 Rebalance 时，该策略会尽可能地保留之前的分配方案，尽量实现分区分配的最小变动。不过有些遗憾的是，这个策略目前还有一些 bug，而且需要升级到 0.11.0.0 才能使用，因此在实际生产环境中用得还不是很多。

总而言之，Rebalance 有以上这三个方面的弊端。你可能会问，这些问题有解吗？特别是针对 Rebalance 慢和影响 TPS 这两个弊端，社区有解决办法吗？针对这两点，我可以很负

责任地告诉你：“无解！”特别是 Rebalance 慢这个问题，Kafka 社区对此无能为力。“本事大不如不摊上”，既然我们没办法解决 Rebalance 过程中的各种问题，干脆就避免 Rebalance 吧，特别是那些不必要的 Rebalance。

就我个人经验而言，**在真实的业务场景中，很多 Rebalance 都是计划外的或者说是不必要的**。我们应用的 TPS 大多是被这类 Rebalance 拖慢的，因此避免这类 Rebalance 就显得很有必要了。下面我们就来说说如何避免 Rebalance。

要避免 Rebalance，还是要从 Rebalance 发生的时机入手。我们在前面说过，Rebalance 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

后面两个通常都是运维的主动操作，所以它们引发的 Rebalance 大都是不可避免的。接下来，我们主要说说因为组成员数量变化而引发的 Rebalance 该如何避免。

如果 Consumer Group 下的 Consumer 实例数量发生变化，就一定会引发 Rebalance。这是 Rebalance 发生的最常见的原因。我碰到的 99% 的 Rebalance，都是这个原因导致的。

Consumer 实例增加的情况很好理解，当我们启动一个配置有相同 group.id 值的 Consumer 程序时，实际上就向这个 Group 添加了一个新的 Consumer 实例。此时，Coordinator 会接纳这个新实例，将其加入到组中，并重新分配分区。通常来说，增加 Consumer 实例的操作都是计划内的，可能是出于增加 TPS 或提高伸缩性的需要。总之，它不属于我们要规避的那类“不必要 Rebalance”。

我们更在意的是 Group 下实例数减少这件事。如果你就是要停掉某些 Consumer 实例，那自不必说，关键是在某些情况下，Consumer 实例会被 Coordinator 错误地认为“已停止”从而被“踢出”Group。如果是这个原因导致的 Rebalance，我们就不能不管了。

Coordinator 会在什么情况下认为某个 Consumer 实例已挂从而要退组呢？这个绝对是需要好好讨论的话题，我们来详细说说。

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。Consumer 端有个参数，叫 session.timeout.ms，就是被用来表征此事的。该参数的默认值是 10 秒，即如果 Coordinator 在 10 秒之内没有收到 Group 下某 Consumer 实例的心跳，它就会认为这个 Consumer 实例已经挂了。可以这么

说，session.timout.ms 决定了 Consumer 存活性的时间间隔。

除了这个参数，Consumer 还提供了一个允许你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。

搞清楚了这些参数的含义，接下来我们来明确一下到底哪些 Rebalance 是“不必要的”。

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。因此，你需要仔细地设置**session.timeout.ms 和 heartbeat.interval.ms**的值。我在这里给出一些推荐数值，你可以“无脑”地应用在你的生产环境中。

- 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。毕竟，我们还是希望能尽快揪出那些“尸位素餐”的 Consumer，早日把它们踢出 Group。希望这份配置能够较好地帮助你规避第一类“不必要”的 Rebalance。

第二类非必要 Rebalance 是 Consumer 消费时间过长导致的。我之前有一个客户，在他们的场景中，Consumer 消费数据时需要将消息处理之后写入到 MongoDB。显然，这是一个很重的消费逻辑。MongoDB 的一丁点不稳定都会导致 Consumer 程序消费时长的增加。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。就拿 MongoDB 这个例子来说，如果写 MongoDB 的最长时间是 7 分钟，那么你可以将该参数设置为 8 分钟左右。

总之，你要为你的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 了。

如果你按照上面的推荐数值恰当地设置了这几个参数，却发现还是出现了 Rebalance，那么我建议你去排查一下**Consumer 端的 GC 表现**，比如是否出现了频繁的 Full GC 导致的长时间停顿，从而引发了 Rebalance。为什么特意说 GC？那是因为在实际场景中，我见过太多因为 GC 设置不合理导致程序频发 Full GC 而引发的非预期 Rebalance 了。

小结

总而言之，我们一定要避免因为各种参数或逻辑不合理而导致的组成员意外离组或退出的情形，与之相关的主要参数有：

- session.timeout.ms
- heartbeat.interval.ms
- max.poll.interval.ms
- GC 参数

按照我们今天所说的内容，恰当地设置这些参数，你一定能够大幅度地降低生产环境中的 Rebalance 数量，从而整体提升 Consumer 端 TPS。

上一页 下一页