

二

25 消费者组重平衡全流程解析

你好，我是胡夕。今天我要和你分享的主题是：消费者组重平衡全流程解析。

之前我们聊到过消费者组的重平衡流程，它的作用是让组内所有的消费者实例就消费哪些主题分区达成一致。重平衡需要借助 Kafka Broker 端的 Coordinator 组件，在 Coordinator 的帮助下完成整个消费者组的分区重分配。今天我们就来详细说说这个流程。

先提示一下，我会以 Kafka 2.3 版本的源代码开启今天的讲述。在分享的过程中，对于旧版本的设计差异，我也会显式地说明。这样，即使你依然在使用比较旧的版本也不打紧，毕竟设计原理大体上是没有变化的。

触发与通知

我们先来简单回顾一下重平衡的 3 个触发条件：

1. 组成员数量发生变化。
2. 订阅主题数量发生变化。
3. 订阅主题的分区数发生变化。

就我个人的经验来看，在实际生产环境中，因命中第 1 个条件而引发的重平衡是最常见的。另外，消费者组中的消费者实例依次启动也属于第 1 种情况，也就是说，每次消费者组启动时，必然会触发重平衡过程。

这部分内容我在专栏[第 15 讲]中已经详细介绍过了，就不再赘述了。如果你不记得的话，可以先去复习一下。

今天，我真正想引出的是另一个话题：**重平衡过程是如何通知到其他消费者实例的？答案就是，靠消费者端的心跳线程（Heartbeat Thread）。**

Kafka Java 消费者需要定期地发送心跳请求（Heartbeat Request）到 Broker 端的协调者，以表明它还活着。在 Kafka 0.10.1.0 版本之前，发送心跳请求是在**消费者主线程**完成的，也就是你写代码调用 `KafkaConsumer.poll` 方法的那个线程。

这样做有诸多弊病，最大的问题在于，**消息处理逻辑也是在这个线程中完成的**。因此，一旦消息处理消耗了过长的时间，心跳请求将无法及时发到协调者那里，导致协调者“错误地”认为该消费者已“死”。自 0.10.1.0 版本开始，社区引入了一个单独的心跳线程来专门执行心跳请求发送，避免了这个问题。

但这和重平衡又有什么关系呢？其实，**重平衡的通知机制正是通过心跳线程来完成的**。当协调者决定开启新一轮重平衡后，它会将“**REBALANCE_IN_PROGRESS**”封装进心跳请求的响应中，发还给消费者实例。当消费者实例发现心跳响应中包含了“**REBALANCE_IN_PROGRESS**”，就能立马知道重平衡又开始了，这就是重平衡的通知机制。

对了，很多人还搞不清楚消费者端参数 `heartbeat.interval.ms` 的真实用途，我来解释一下。从字面上看，它就是设置了心跳的间隔时间，但这个参数的真正作用是控制重平衡通知的频率。如果你想要消费者实例更迅速地得到通知，那么就可以给这个参数设置一个非常小的值，这样消费者就能更快地感知到重平衡已经开启了。

消费者组状态机

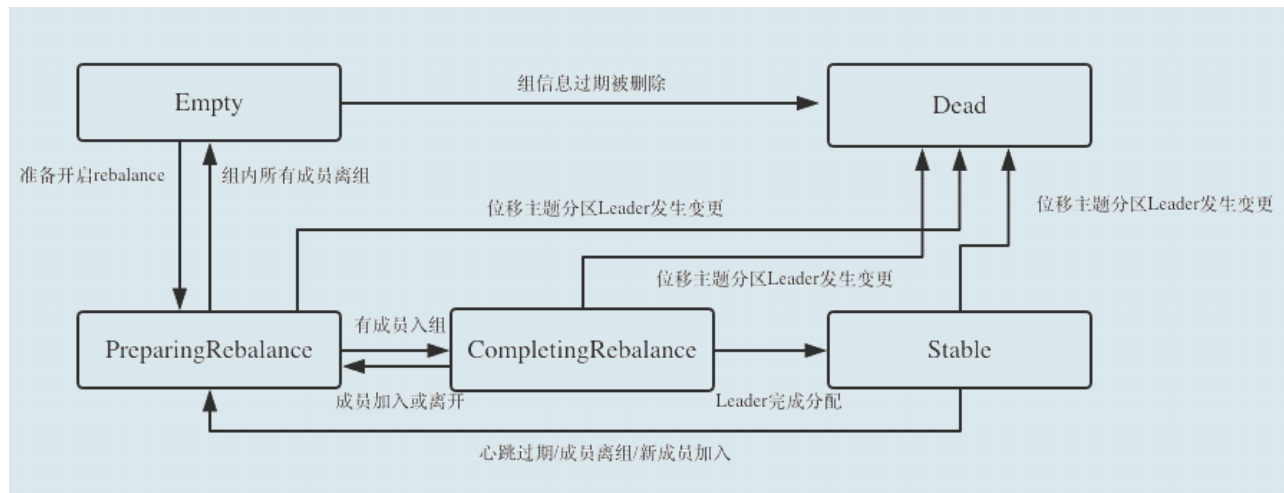
重平衡一旦开启，Broker 端的协调者组件就要开始忙了，主要涉及到控制消费者组的状态流转。当前，Kafka 设计了一套消费者组状态机（State Machine），来帮助协调者完成整个重平衡流程。严格来说，这套状态机属于非常底层的设计，Kafka 官网上压根就没有提到过，但你最好还是了解一下，因为它能够帮助你搞懂消费者组的设计原理，比如消费者组的**过期位移（Expired Offsets）删除等**。

目前，Kafka 为消费者组定义了 5 种状态，它们分别是：Empty、Dead、PreparingRebalance、CompletingRebalance 和 Stable。那么，这 5 种状态的含义是什么呢？我们一起来看看下面这张表格。

消费者组的5种状态	
状态	含义
Empty	组内没有任何成员，但消费者组可能存在已提交的位移数据，而且这些位移尚未过期。
Dead	同样是组内没有任何成员，但组的元数据信息已经在协调者端被移除。协调者组件保存着当前向它注册过的所有组信息，所谓的元数据信息就类似于这个注册信息。
PreparingRebalance	消费者组准备开启重平衡，此时所有成员都要重新请求加入消费者组。
	消费者组下所有成员已经加入，各个成员正在等

CompletingRebalance	待分配方案。该状态在老一点的版本中被称为AwaitingSync，它和CompletingRebalance是等价的。
Stable	消费者组的稳定状态。该状态表明重平衡已经完成，组内各成员能够正常消费数据了。

了解了这些状态的含义之后，我们来看一张图片，它展示了状态机的各个状态流转。



我来解释一下消费者组启动时的状态流转过程。一个消费者组最开始是 Empty 状态，当重平衡过程开启后，它会被置于 PreparingRebalance 状态等待成员加入，之后变更到 CompletingRebalance 状态等待分配方案，最后流转到 Stable 状态完成重平衡。

当有新成员加入或已有成员退出时，消费者组的状态从 Stable 直接跳到 PreparingRebalance 状态，此时，所有现存成员就必须重新申请加入组。当所有成员都退出组后，消费者组状态变更为 Empty。Kafka 定期自动删除过期位移的条件就是，组要处于 Empty 状态。因此，如果你的消费者组停掉了很长时间（超过 7 天），那么 Kafka 很可能就把该组的位移数据删除了。我相信，你在 Kafka 的日志中一定经常看到下面这个输出：

```
Removed XXX expired offsets in XXX milliseconds.
```

这就是 Kafka 在尝试定期删除过期位移。现在你知道了，只有 Empty 状态下的组，才会执行过期位移删除的操作。

消费者端重平衡流程

有了上面的内容作铺垫，我们就可以开始介绍重平衡流程了。重平衡的完整流程需要消费者端和协调者组件共同参与才能完成。我们先从消费者的视角来审视一下重平衡的流程。

在消费者端，重平衡分为两个步骤：分别是加入组和等待领导者消费者（Leader Consumer）分配方案。这两个步骤分别对应两类特定的请求：**JoinGroup 请求**和**SyncGroup 请求**。

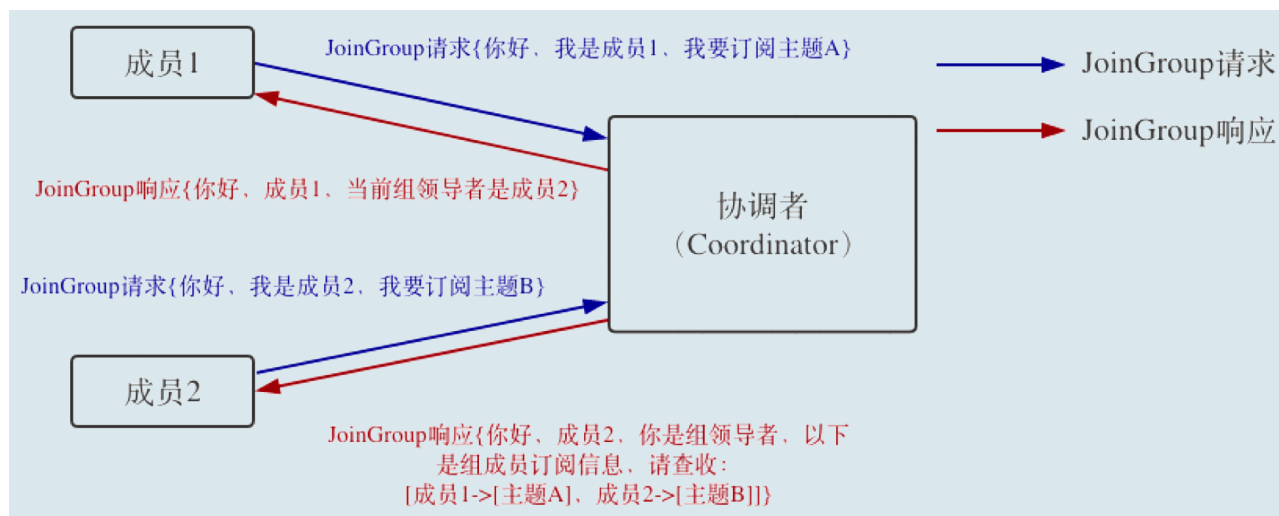
当组内成员加入组时，它会向协调者发送 JoinGroup 请求。在该请求中，每个成员都要将自己订阅的主题上报，这样协调者就能收集到所有成员的订阅信息。一旦收集了全部成员的 JoinGroup 请求后，协调者会从这些成员中选择一个担任这个消费者组的领导者。

通常情况下，第一个发送 JoinGroup 请求的成员自动成为领导者。你一定要注意区分这里的领导者和之前我们介绍的领导者副本，它们不是一个概念。这里的领导者是具体的消费者实例，它既不是副本，也不是协调者。**领导者消费者的任务是收集所有成员的订阅信息，然后根据这些信息，制定具体的分区消费分配方案。**

选出领导者之后，协调者会把消费者组订阅信息封装进 JoinGroup 请求的响应体中，然后发给领导者，由领导者统一做出分配方案后，进入到下一步：发送 SyncGroup 请求。

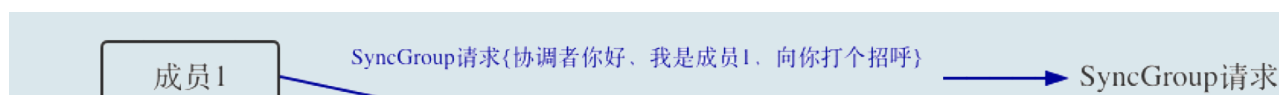
在这一步中，领导者向协调者发送 SyncGroup 请求，将刚刚做出的分配方案发给协调者。值得注意的是，其他成员也会向协调者发送 SyncGroup 请求，只不过请求体中并没有实际的内容。这一步的主要目的是让协调者接收分配方案，然后统一以 SyncGroup 响应的方式分发给所有成员，这样组内所有成员就都知道自己该消费哪些分区了。

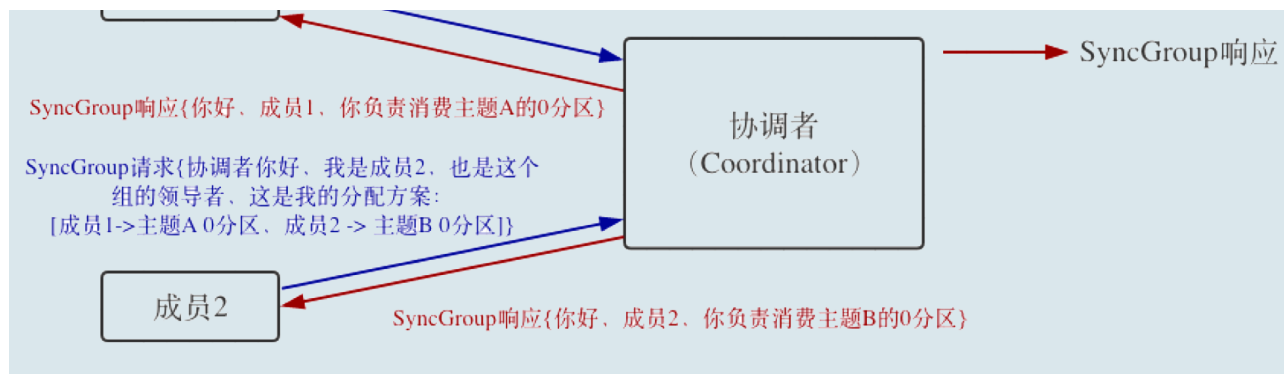
接下来，我用一张图来形象地说明一下 JoinGroup 请求的处理过程。



就像前面说的，JoinGroup 请求的主要作用是将组成员订阅信息发送给领导者消费者，待领导者制定好分配方案后，重平衡流程进入到 SyncGroup 请求阶段。

下面这张图描述的是 SyncGroup 请求的处理流程。





SyncGroup 请求的主要目的, 就是让协调者把领导者制定的分配方案下发给各个组内成员。当所有成员都成功接收到分配方案后, 消费者组进入到 Stable 状态, 即开始正常的消费工作。

讲完这里, **消费者端**的重平衡流程我已经介绍完了。接下来, 我们从**协调者端**来看一下重平衡是怎么执行的。

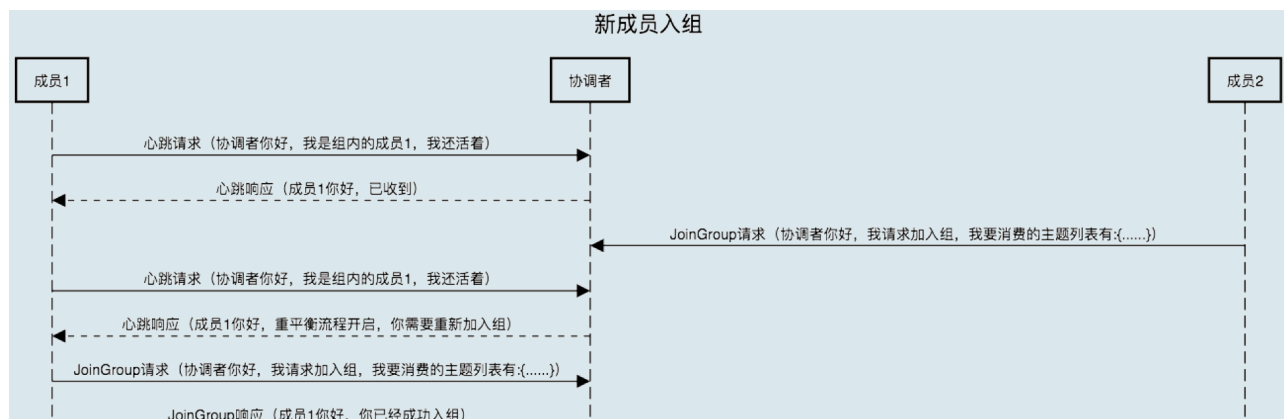
Broker 端重平衡场景剖析

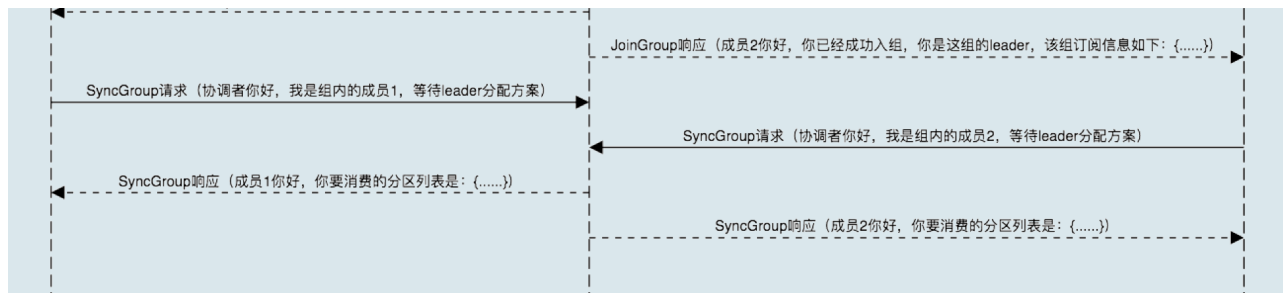
要剖析协调者端处理重平衡的全流程, 我们必须分几个场景来讨论。这几个场景分别是新成员加入组、组成员主动离组、组成员崩溃离组、组成员提交位移。接下来, 我们一个一个来讨论。

场景一：新成员入组。

新成员入组是指组处于 Stable 状态后, 有新成员加入。如果是全新启动一个消费者组, Kafka 是有一些自己的小优化的, 流程上会有些许的不同。我们这里讨论的是, 组稳定了之后有新成员加入的情形。

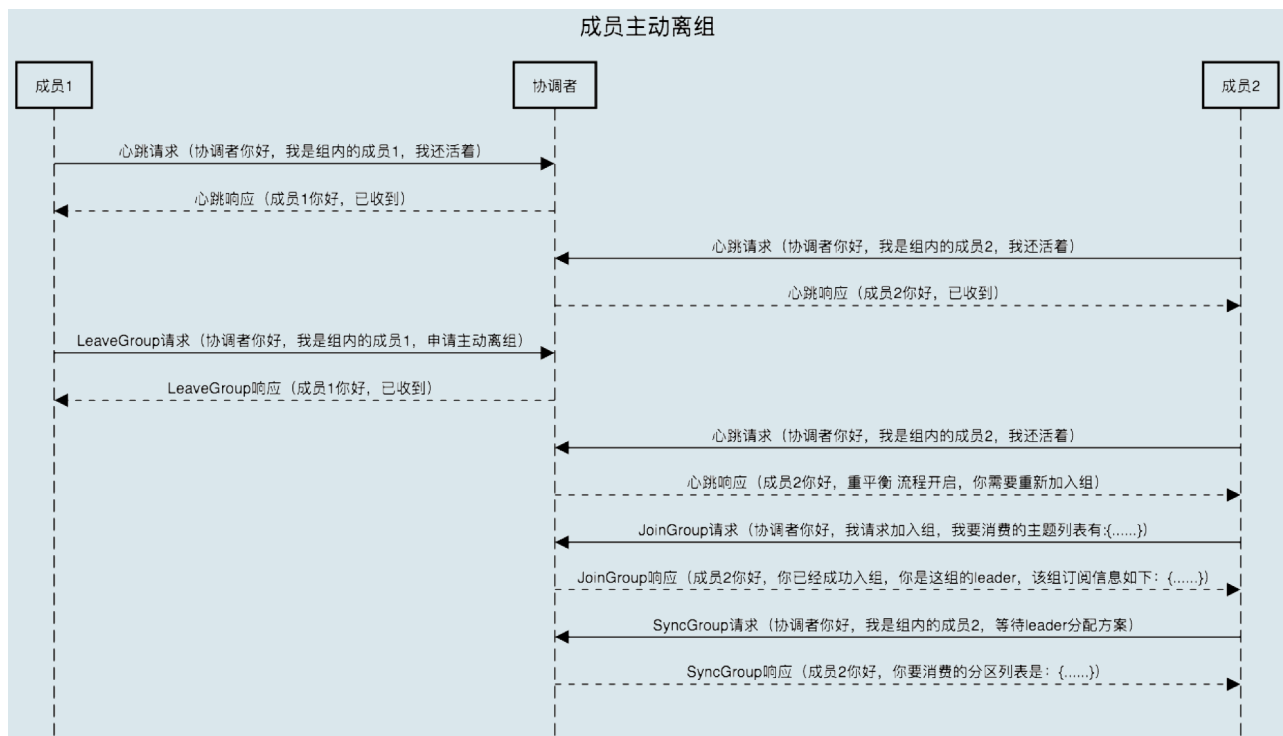
当协调者收到新的 JoinGroup 请求后, 它会通过心跳请求响应的方式通知组内现有的所有成员, 强制它们开启新一轮的重平衡。具体的过程和之前的客户端重平衡流程是一样的。现在, 我用一张时序图来说明协调者一端是如何处理新成员入组的。





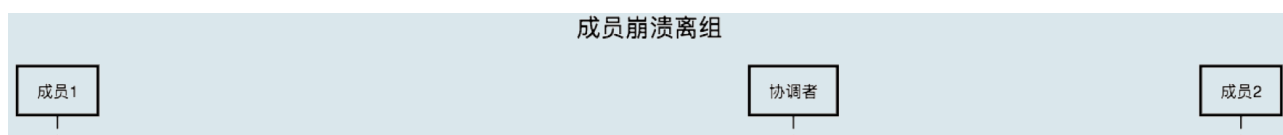
场景二：组成员主动离组。

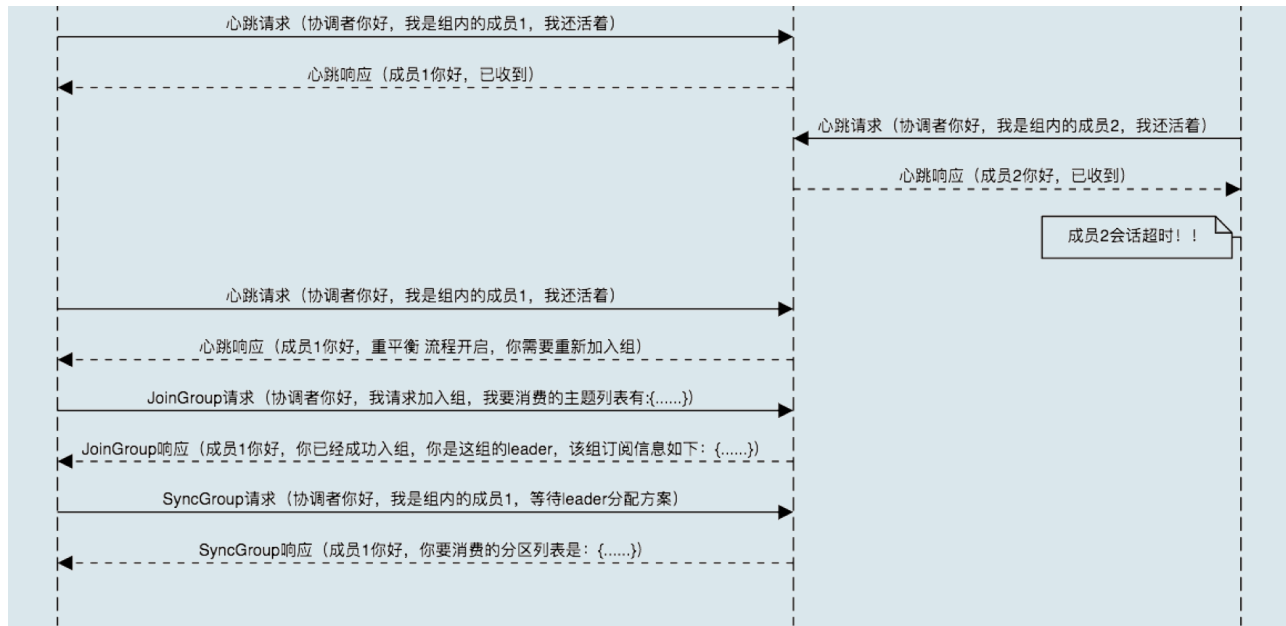
何谓主动离组？就是指消费者实例所在线程或进程调用 `close()` 方法主动通知协调者它要退出。这个场景就涉及到了第三类请求：**LeaveGroup 请求**。协调者收到 `LeaveGroup` 请求后，依然会以心跳响应的方式通知其他成员，因此我就不再赘述了，还是直接用一张图来说明。



场景三：组成员崩溃离组。

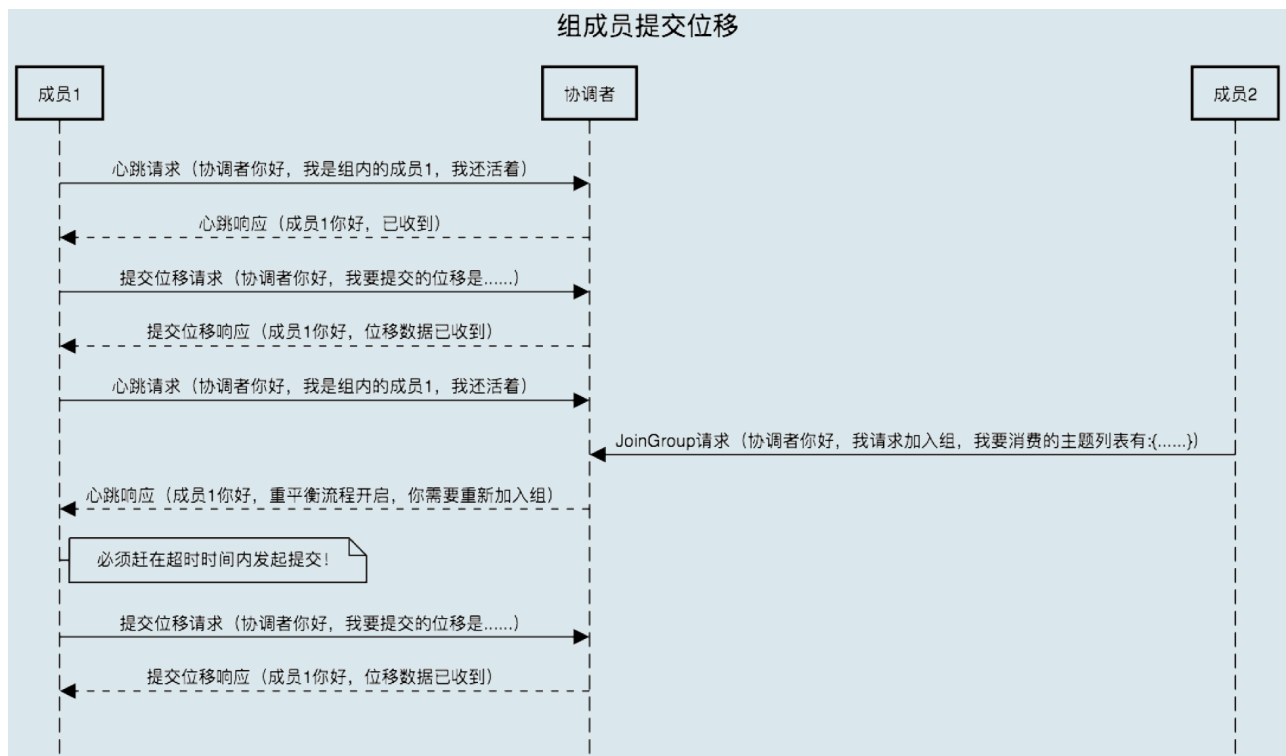
崩溃离组是指消费者实例出现严重故障，突然宕机导致的离组。它和主动离组是有区别的，因为后者是主动发起的离组，协调者能马上感知并处理。但崩溃离组是被动的，协调者通常需要等待一段时间才能感知到，这段时间一般是由消费者端参数 `session.timeout.ms` 控制的。也就是说，Kafka 一般不会超过 `session.timeout.ms` 就能感知到这个崩溃。当然，后面处理崩溃离组的流程与之前是一样的，我们来看看下面这张图。





场景四：重平衡时协调者对组内成员提交位移的处理。

正常情况下，每个组内成员都会定期汇报位移给协调者。当重平衡开启时，协调者会给予成员一段缓冲时间，要求每个成员必须在这段时间内快速地上报自己的位移信息，然后再开启正常的 JoinGroup/SyncGroup 请求发送。还是老办法，我们使用一张图来说明。



小结

好了，消费者重平衡流程我已经全部讲完了。虽然全程我都是拿两个成员来举例子，但你可

以很容易地扩展到多个成员的消费组，毕竟它们的原理是相同的。我希望你能多看几遍今天的内容，彻底掌握 Kafka 的消费者重平衡流程。社区正在对目前的重平衡流程做较大程度的改动，如果你不了解这些基础的设计原理，后面想深入学习这部分内容的话，会十分困难。

重点知识梳理

- 重平衡的3个触发条件：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。
- Kafka为消费者组定义的5种状态：Empty、Dead、PreparingRebalance、CompletingRebalance和Stable。
- 消费者端的重平衡的2个步骤：加入组和等待领导者消费者分配方案。这2个步骤分别对应JoinGroup请求和SyncGroup请求。
- 协调者端处理重平衡的4个场景：新成员入组；组成员主动离组；组成员崩溃离组；重平衡时协调者对组内成员提交位移的处理。



[上一页](#)

[下一页](#)