



下载APP



32 | GroupCoordinator : 在Rebalance中 , Coordinator如何处理成员入组 ?

2020-07-11 胡夕

Kafka核心源码解读

[进入课程 >](#)**讲述 : 胡夕**

时长 23:26 大小 21.47M



你好,我是胡夕。不知不觉间,课程已经接近尾声了,最后这两节课,我们来学习一下消费者组的 Rebalance 流程是如何完成的。

提到 Rebalance,你的第一反应一定是“爱恨交加”。毕竟,如果使用得当,它能够自动帮我们实现消费者之间的负载均衡和故障转移;但如果配置失当,我们就可能触碰到它诟病已久的缺陷:耗时长,而且会出现消费中断。



在使用消费者组的实践中,你肯定想知道,应该如何避免 Rebalance。如果你不了解 Rebalance 的源码机制的话,就很容易掉进它无意中铺设的“陷阱”里。

举个小例子。有些人认为, Consumer 端参数 `session.timeout.ms` 决定了完成一次 Rebalance 流程的最大时间。这种认知是不对的, 实际上, 这个参数是用于检测消费者组成员存活性的, 即如果在这段超时时间内, 没有收到该成员发给 Coordinator 的心跳请求, 则把该成员标记为 Dead, 而且要显式地将其从消费者组中移除, 并触发新一轮的 Rebalance。而真正决定单次 Rebalance 所用最大时长的参数, 是 Consumer 端的 **`max.poll.interval.ms`**。显然, 如果没有搞懂这部分的源码, 你就没办法为这些参数设置合理的数值。

总体而言, Rebalance 的流程大致分为两大步: 加入组 (JoinGroup) 和组同步 (SyncGroup)。

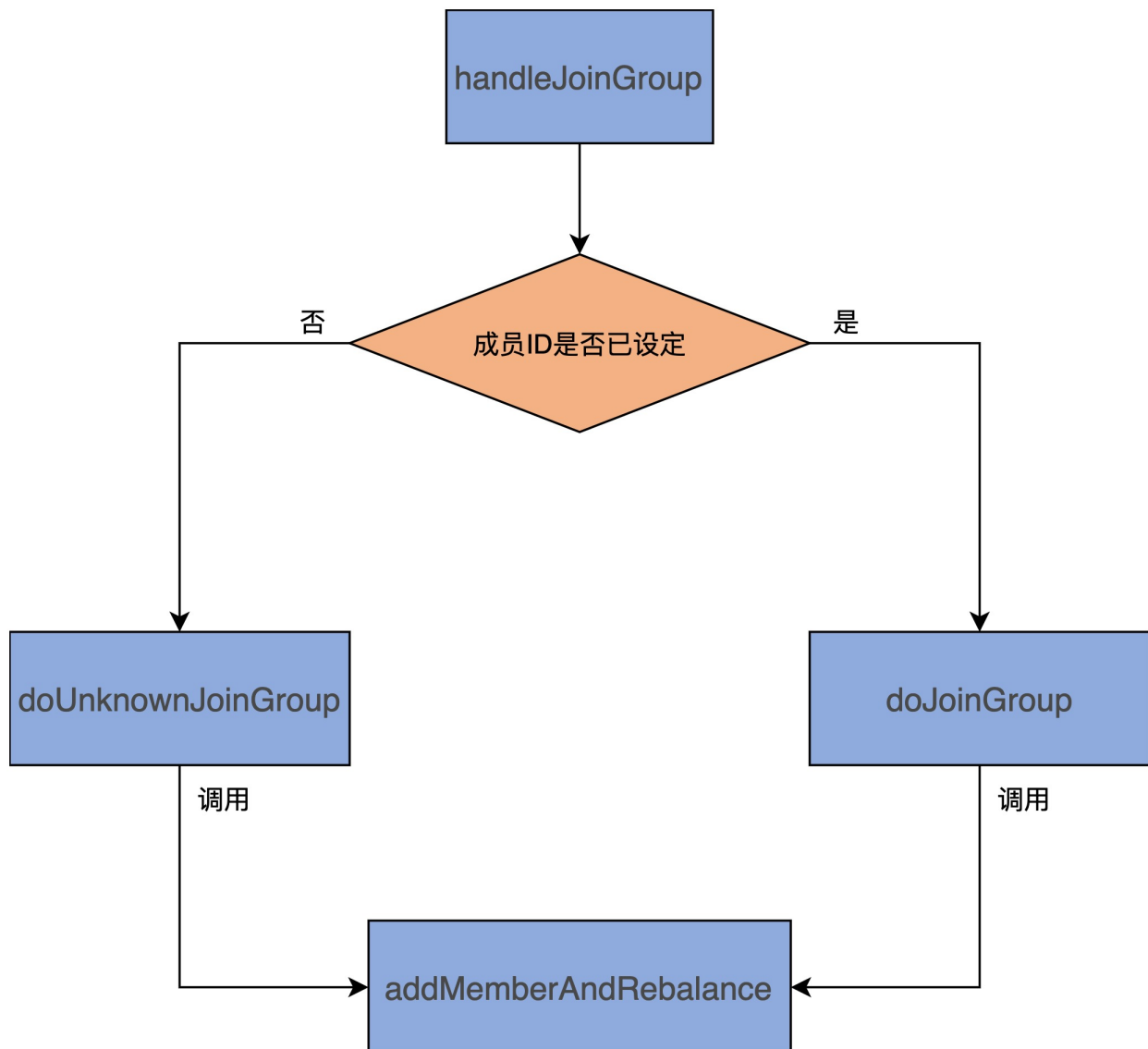
加入组, 是指消费者组下的各个成员向 Coordinator 发送 JoinGroupRequest 请求加入进组的过程。这个过程有一个超时时间, 如果有成员在超时时间之内, 无法完成加入组操作, 它就会被排除在这轮 Rebalance 之外。

组同步, 是指当所有成员都成功加入组之后, Coordinator 指定其中一个成员为 Leader, 然后将订阅分区信息发给 Leader 成员。接着, 所有成员 (包括 Leader 成员) 向 Coordinator 发送 SyncGroupRequest 请求。需要注意的是, **只有 Leader 成员发送的请求中包含了订阅分区消费分配方案, 在其他成员发送的请求中, 这部分的内容为空。**当 Coordinator 接收到分配方案后, 会通过向成员发送响应的方式, 通知各个成员要消费哪些分区。

当组同步完成后, Rebalance 宣告结束。此时, 消费者组处于正常工作状态。

今天, 我们就学习下第一大步, 也就是加入组的源码实现, 它们位于 GroupCoordinator.scala 文件中。下节课, 我们再深入地学习组同步的源码实现。

要搞懂加入组的源码机制, 我们必须学习 4 个方法, 分别是 `handleJoinGroup`、`doUnknownJoinGroup`、`doJoinGroup` 和 `addMemberAndRebalance`。`handleJoinGroup` 是执行加入组的顶层方法, 被 `KafkaApis` 类调用, 该方法依据给定消费者组成员是否设置了成员 ID, 来决定是调用 `doUnknownJoinGroup` 还是 `doJoinGroup`, 前者对应于未设定成员 ID 的情形, 后者对应于已设定成员 ID 的情形。而这两个方法, 都会调用 `addMemberAndRebalance`, 执行真正的加入组逻辑。为了帮助你理解它们之间的交互关系, 我画了一张图, 借用它展示了这 4 个方法的调用顺序。



handleJoinGroup 方法

如果你翻开 `KafkaApis.scala` 这个 API 入口文件, 就可以看到, 处理 `JoinGroupRequest` 请求的方法是 `handleJoinGroupRequest`。而它的主要逻辑, 就是调用 **GroupCoordinator 的 `handleJoinGroup` 方法**, 来处理消费者组成员发送过来的加入组请求, 所以, 我们要具体学习一下 `handleJoinGroup` 方法。先看它的方法签名:

复制代码

```
1 def handleJoinGroup(  
2   groupId: String, // 消费者组名  
3   memberId: String, // 消费者组成员ID  
4   groupInstanceId: Option[String], // 组实例ID, 用于标识静态成员  
5   requireKnownMemberId: Boolean, // 是否需要成员ID不为空  
6   clientId: String, // client.id值  
7   clientHost: String, // 消费者程序主机名  
8   rebalanceTimeoutMs: Int, // Rebalance超时时间, 默认是max.poll.interval.ms值
```

```
9    sessionTimeoutMs: Int, // 会话超时时间
10   protocolType: String, // 协议类型
11   protocols: List[(String, Array[Byte])], // 按照分配策略分组的订阅分区
12   responseCallback: JoinCallback // 回调函数
13   ): Unit = {
14       .....
15   }
```

这个方法的参数有很多, 我介绍几个比较关键的。接下来在阅读其他方法的源码时, 你还会看到这些参数, 所以, 这里你一定要提前掌握它们的含义。

groupId : 消费者组名。

memberId : 消费者组成员 ID。如果成员是新加入的, 那么该字段是空字符串。

groupId : 这是社区于 2.4 版本引入的静态成员字段。静态成员的引入, 可以有效避免因系统升级或程序更新而导致的 Rebalance 场景。它属于比较高阶的用法, 而且目前还没有被大规模使用, 因此, 这里你只需要简单了解一下它的作用。另外, 后面在讲其他方法时, 我会直接省略静态成员的代码, 我们只关注核心逻辑就行了。

requireKnownMemberId : 是否要求成员 ID 不为空, 即是否要求成员必须设置 ID 的布尔字段。这个字段如果为 True 的话, 那么, Kafka 要求消费者组成员必须设置 ID。未设置 ID 的成员, 会被拒绝加入组。直到它设置了 ID 之后, 才能重新加入组。

clientId : 消费者端参数 client.id 值。Coordinator 使用它来生成 memberId。
memberId 的格式是 clientId 值 -UUID。

clientHost : 消费者程序的主机名。

rebalanceTimeoutMs : Rebalance 超时时间。如果在这个时间段内, 消费者组成员没有完成加入组的操作, 就会被禁止入组。

sessionTimeoutMs : 会话超时时间。如果消费者组成员无法在这段时间内向 Coordinator 汇报心跳, 那么将被视为“已过期”, 从而引发新一轮 Rebalance。

responseCallback : 完成加入组之后的回调逻辑方法。当消费者组成员成功加入组之后, 需要执行该方法。

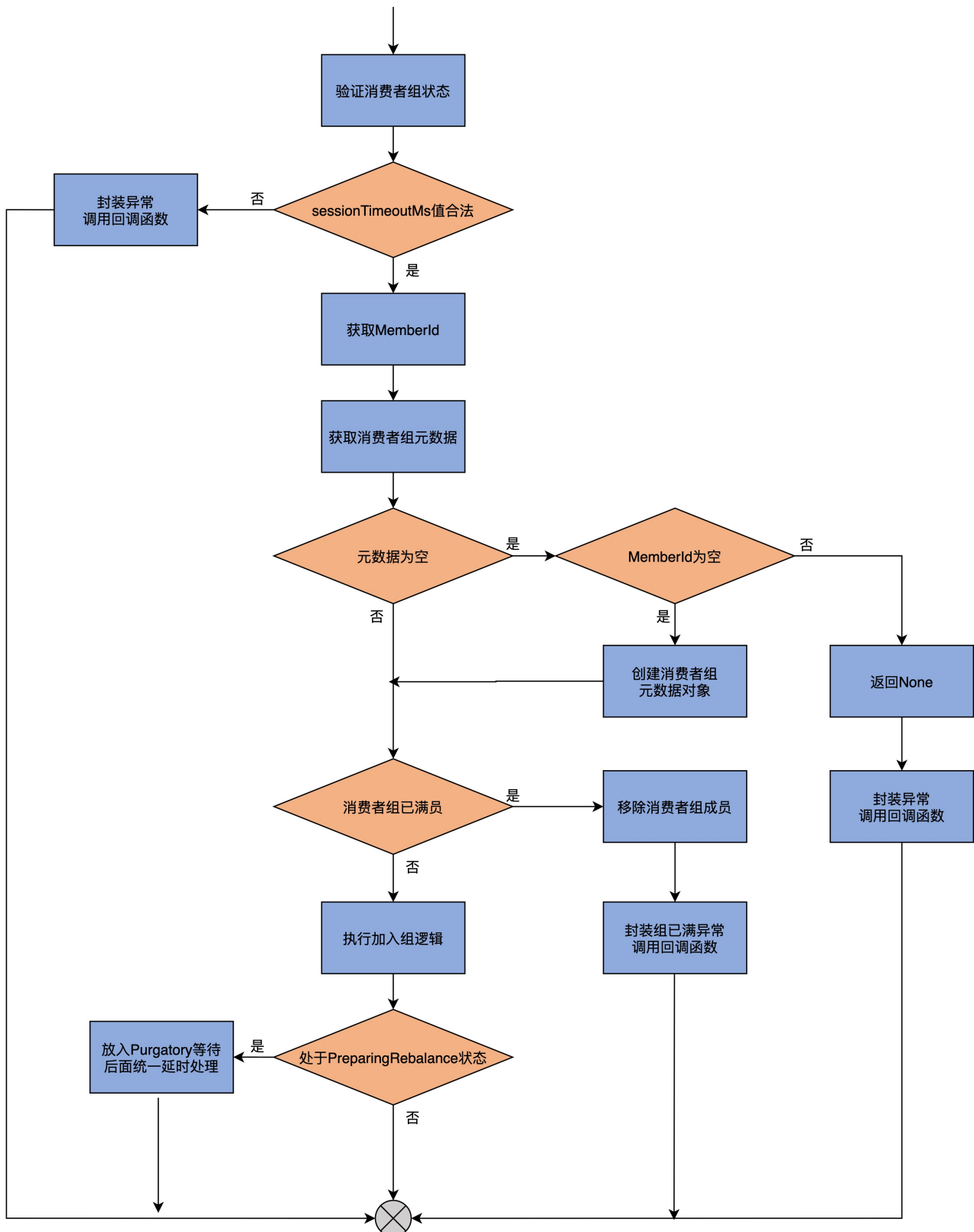
说完了方法签名, 我们看下它的主体代码 :

```
1 // 验证消费者组状态的合法性
```

 复制代码

```
2 validateGroupStatus(groupId, ApiKeys.JOIN_GROUP).foreach { error =>
3   responseCallback(JoinGroupResult(memberId, error))
4   return
5 }
6 // 确保sessionTimeoutMs介于
7 // [group.min.session.timeout.ms值, group.max.session.timeout.ms值]之间
8 // 否则抛出异常, 表示超时时间设置无效
9 if (sessionTimeoutMs < groupConfig.groupMinSessionTimeoutMs ||
10    sessionTimeoutMs > groupConfig.groupMaxSessionTimeoutMs) {
11   responseCallback(JoinGroupResult(memberId, Errors.INVALID_SESSION_TIMEOUT))
12 } else {
13   // 消费者组成员ID是否为空
14   val isUnknownMember = memberId == JoinGroupRequest.UNKNOWN_MEMBER_ID
15   // 获取消费者组信息, 如果组不存在, 就创建一个新的消费者组
16   groupManager.getOrCreateGroup(groupId, isUnknownMember) match {
17     case None =>
18       responseCallback(JoinGroupResult(memberId, Errors.UNKNOWN_MEMBER_ID))
19     case Some(group) =>
20       group.inLock {
21         // 如果该消费者组已满员
22         if (!acceptJoiningMember(group, memberId)) {
23           // 移除该消费者组成员
24           group.remove(memberId)
25           group.removeStaticMember(groupInstanceId)
26           // 封装异常表明组已满员
27           responseCallback(JoinGroupResult(
28             JoinGroupRequest.UNKNOWN_MEMBER_ID,
29             Errors.GROUP_MAX_SIZE_REACHED))
30         // 如果消费者组成员ID为空
31         } else if (isUnknownMember) {
32           // 为空ID成员执行加入组操作
33           doUnknownJoinGroup(group, groupInstanceId, requireKnownMemberId, cli
34         } else {
35           // 为非空ID成员执行加入组操作
36           doJoinGroup(group, memberId, groupInstanceId, clientId, clientHost,
37         }
38         // 如果消费者组正处于PreparingRebalance状态
39         if (group.is(PreparingRebalance)) {
40           // 放入Purgatory, 等待后面统一延时处理
41           joinPurgatory.checkAndComplete(GroupKey(group.groupId))
42         }
43       }
44     }
45 }
```

为了方便你更直观地理解, 我画了一张图来说明它的完整流程。



第 1 步，调用 `validateGroupStatus` 方法验证消费者组状态的合法性。所谓的合法性，也就是消费者组名 `groupId` 不能为空，以及 `JoinGroupRequest` 请求发送给了正确的 `Coordinator`，这两者必须同时满足。如果没有通过这些检查，那么，`handleJoinGroup` 方法会封装相应的错误，并调用回调函数返回。否则，就进入到下一步。

第 2 步, 代码检验 `sessionTimeoutMs` 的值是否介于`[group.min.session.timeout.ms, group.max.session.timeout.ms]`之间, 如果不是, 就认定该值是非法值, 从而封装一个对应的异常调用回调函数返回, 这两个参数分别表示消费者组允许配置的最小和最大会话超时时间; 如果是的话, 就进入下一步。

第 3 步, 代码获取当前成员的 ID 信息, 并查看它是否为空。之后, 通过 `GroupMetadataManager` 获取消费者组的元数据信息, 如果该组的元数据信息存在, 则进入到下一步; 如果不存在, 代码会看当前成员 ID 是否为空, 如果为空, 就创建一个空的元数据对象, 然后进入到下一步, 如果不为空, 则返回 `None`。一旦返回了 `None`, `handleJoinGroup` 方法会封装“未知成员 ID”的异常, 调用回调函数返回。

第 4 步, 检查当前消费者组是否已满员。该逻辑是通过 **`acceptJoiningMember` 方法**实现的。这个方法根据**消费者组状态**确定是否满员。这里的消费者组状态有三种。

状态一: 如果是 `Empty` 或 `Dead` 状态, 肯定不会是满员, 直接返回 `True`, 表示可以接纳申请入组的成员;

状态二: 如果是 `PreparingRebalance` 状态, 那么, 批准成员入组的条件是必须满足一下两个条件之一。

该成员是之前已有的成员, 且当前正在等待加入组;

当前等待加入组的成员数小于 Broker 端参数 `group.max.size` 值。

只要满足这两个条件中的任意一个, 当前消费者组成员都会被批准入组。

状态三: 如果是其他状态, 那么, 入组的条件是**该成员是已有成员, 或者是当前组总成员数小于 Broker 端参数 `group.max.size` 值**。需要注意的是, 这里比较的是**组当前的总成员数**, 而不是等待入组的成员数, 这是因为, 一旦 `Rebalance` 过渡到 `CompletingRebalance` 之后, 没有完成加入组的成员, 就会被移除。

倘若成员不被批准入组, 那么, 代码需要将该成员从元数据缓存中移除, 同时封装“组已满员”的异常, 并调用回调函数返回; 如果成员被批准入组, 则根据 `Member ID` 是否为空, 就执行 `doUnknownJoinGroup` 或 `doJoinGroup` 方法执行加入组的逻辑。

第 5 步是尝试完成 JoinGroupRequest 请求的处理。如果消费者组处于 PreparingRebalance 状态, 那么, 就将该请求放入 Purgatory, 尝试立即完成; 如果是其它状态, 则无需将请求放入 Purgatory。毕竟, 我们处理的是加入组的逻辑, 而此时消费者组的状态应该要变更到 PreparingRebalance 后, Rebalance 才能完成加入组操作。当然, 如果延时请求不能立即完成, 则交由 Purgatory 统一进行延时处理。


至此, handleJoinGroup 逻辑结束。

实际上, 我们可以看到, 真正执行加入组逻辑的是 doUnknownJoinGroup 和 doJoinGroup 这两个方法。那么, 接下来, 我们就来学习下这两个方法。

doUnknownJoinGroup 方法

如果是全新的消费者组成员加入组, 那么, 就需要为它们执行 doUnknownJoinGroup 方法, 因为此时, 它们的 Member ID 尚未生成。

除了 memberId 之外, 该方法的输入参数与 handleJoinGroup 方法几乎一模一样, 我就不一一地详细介绍了, 我们直接看它的源码。为了便于你理解, 我省略了关于静态成员以及 DEBUG/INFO 调试的部分代码。

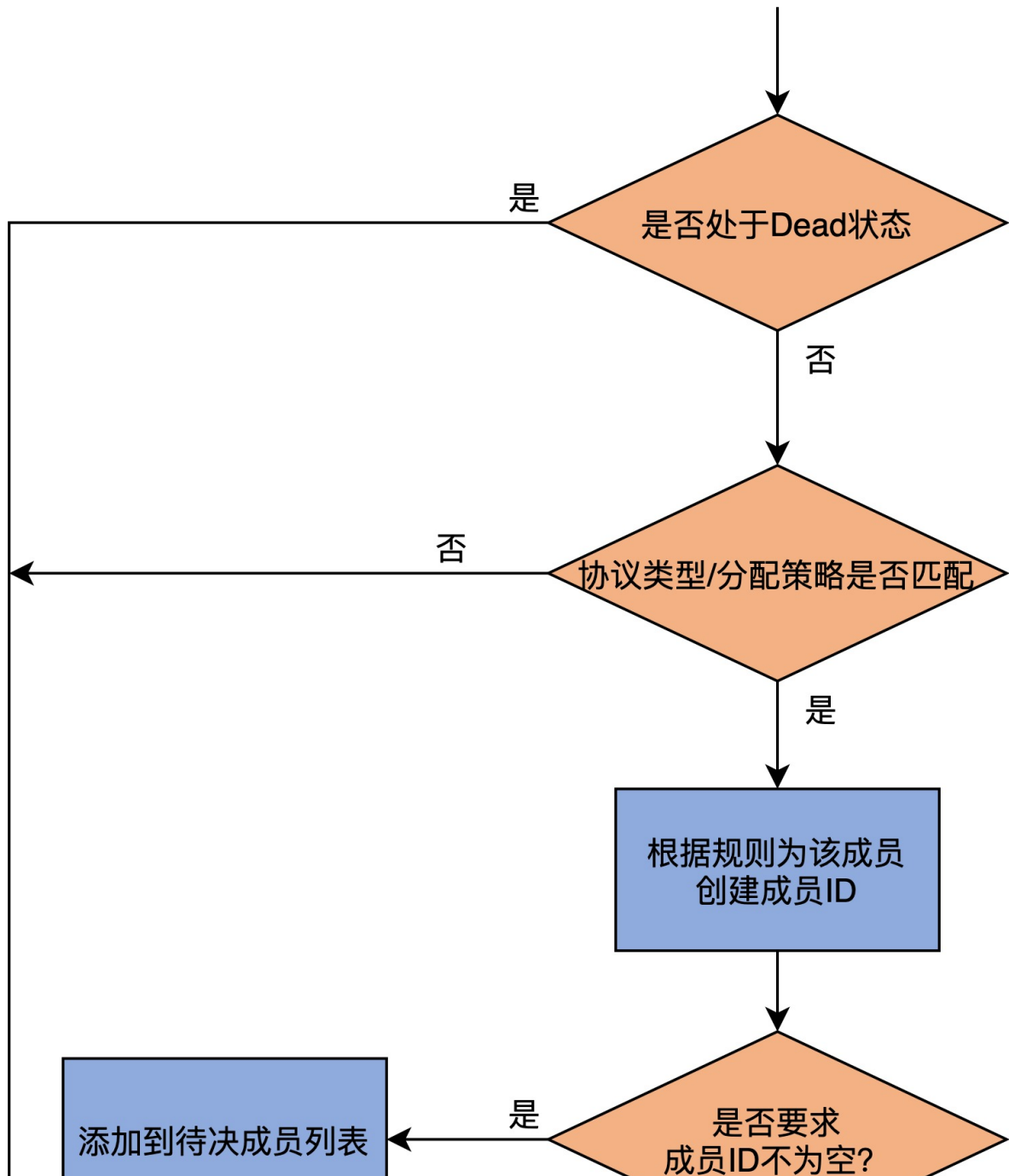
 复制代码

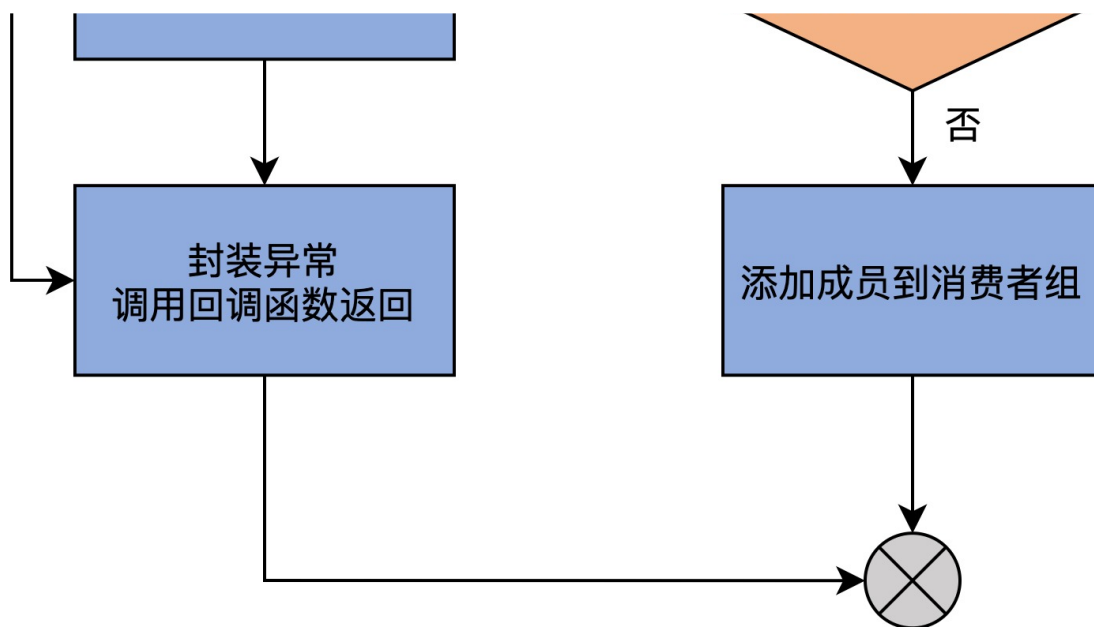
```
1 group.inLock {
2     // Dead状态
3     if (group.is(Dead)) {
4         // 封装异常调用回调函数返回
5         responseCallback(JoinGroupResult(
6             JoinGroupRequest.UNKNOWN_MEMBER_ID,
7             Errors.COORDINATOR_NOT_AVAILABLE))
8         // 成员配置的协议类型/分区消费分配策略与消费者组的不匹配
9     } else if (!group.supportsProtocols(protocolType, MemberMetadata.plainProtoc
10 responseCallback(JoinGroupResult(JoinGroupRequest.UNKNOWN_MEMBER_ID, Errors.
11 } else {
12     // 根据规则为该成员创建成员ID
13     val newMemberId = group.generateMemberId(clientId, groupInstanceId)
14     // 如果配置了静态成员
15     if (group.hasStaticMember(groupInstanceId)) {
16         .....
17     // 如果要求成员ID不为空
18     } else if (requireKnownMemberId) {
19         .....
20     group.addPendingMember(newMemberId)
21     addPendingMemberExpiration(group, newMemberId, sessionTimeoutMs)
```



```
22     responseCallback(JoinGroupResult(newMemberId, Errors.MEMBER_ID_REQUIRED)
23 } else {
24     .....
25     // 添加成员
26     addMemberAndRebalance(rebalanceTimeoutMs, sessionTimeoutMs, newMemberId,
27         clientId, clientHost, protocolType, protocols, group, responseCallback
28 }
29 }
30 }
31
```

为了方便你理解，我画了一张图来展示下这个方法的流程。





首先，代码会检查消费者组的状态。

如果是 Dead 状态，则封装异常，然后调用回调函数返回。你可能会觉得奇怪，既然是向该组添加成员，为什么组状态还能是 Dead 呢？实际上，这种情况是可能的。因为，在成员加入组的同时，可能存在另一个线程，已经把组的元数据信息从 Coordinator 中移除了。比如，组对应的 Coordinator 发生了变更，移动到了其他的 Broker 上，此时，代码封装一个异常返回给消费者程序，后者会去寻找最新的 Coordinator，然后重新发起加入组操作。

如果状态不是 Dead，就检查该成员的协议类型以及分区消费分配策略，是否与消费者组当前支持的方案匹配，如果不匹配，依然是封装异常，然后调用回调函数返回。这里的匹配与否，是指成员的协议类型与消费者组的是否一致，以及成员设定的分区消费分配策略是否被消费者组下的其它成员支持。

如果这些检查都顺利通过，接着，代码就会为该成员生成成员 ID，生成规则是 clientId-UUID。这便是 generateMemberId 方法做的事情。然后，handleJoinGroup 方法会根据 requireKnownMemberId 的取值，来决定下面的逻辑路径：

如果该值为 True，则将该成员加入到待决成员列表（Pending Member List）中，然后封装一个异常以及生成好的成员 ID，将该成员的入组申请“打回去”，令其分配好了成员 ID 之后再重新申请；

如果为 False , 则无需这么苛刻 , 直接调用 `addMemberAndRebalance` 方法将其加入到组中。至此 , `handleJoinGroup` 方法结束。

通常来说 , 如果你没有启用静态成员机制的话 , `requireKnownMemberId` 的值是 True , 这是由 `KafkaApis` 中 `handleJoinGroupRequest` 方法的这行语句决定的 :

[复制代码](#)

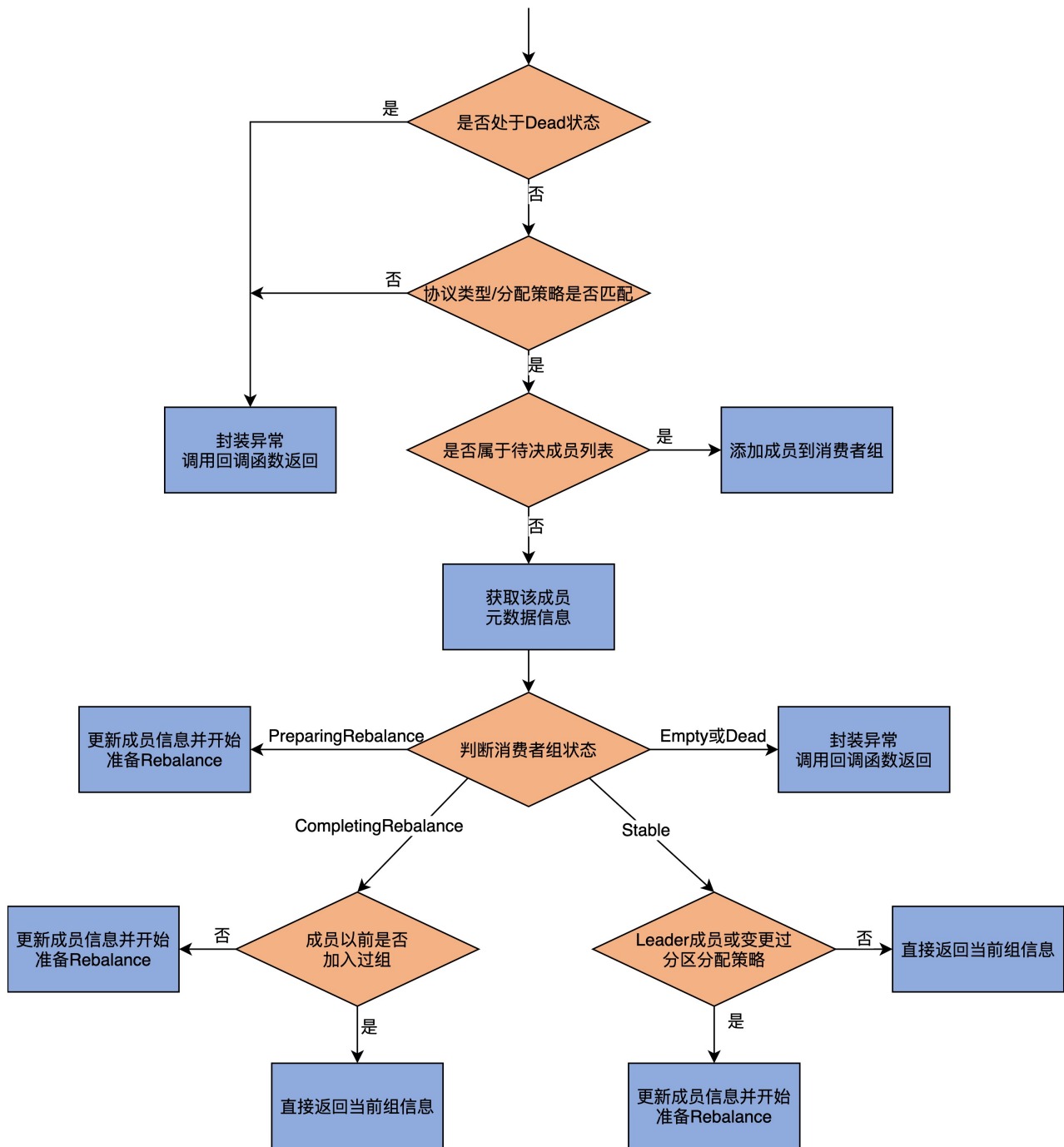
```
1 val requireKnownMemberId = joinGroupRequest.version >= 4 && groupInstanceId.is
```

可见 , 如果你使用的是比较新的 Kafka 客户端版本 , 而且没有配置过 Consumer 端参数 `group.instance.id` 的话 , 那么 , 这个字段的值就是 True , 这说明 , Kafka 要求消费者成员加入组时 , 必须要分配好成员 ID。

关于 `addMemberAndRebalance` 方法的源码 , 一会儿在学习 `doJoinGroup` 方法时 , 我再给你具体解释。

doJoinGroup 方法

接下来 , 我们看下 `doJoinGroup` 方法。这是为那些设置了成员 ID 的成员 , 执行加入组逻辑的方法。它的输入参数全部承袭自 `handleJoinGroup` 方法输入参数 , 你应该已经很熟悉了 , 因此 , 我们直接看它的源码实现。由于代码比较长 , 我分成两个部分来介绍。同时 , 我再画一张图 , 帮助你理解整个方法的逻辑。



第 1 部分

这部分主要做一些校验和条件检查。

复制代码

```

1 // 如果是Dead状态, 封装COORDINATOR_NOT_AVAILABLE异常调用回调函数返回
2 if (group.is(Dead)) {
3     responseCallback(JoinGroupResult(memberId, Errors.COORDINATOR_NOT_AVAILABLE)
4 // 如果协议类型或分区消费分配策略与消费者组的不匹配
5 // 封装INCONSISTENT_GROUP_PROTOCOL异常调用回调函数返回
6 } else if (!group.supportsProtocols(protocolType, MemberMetadata.plainProtocol
7     responseCallback(JoinGroupResult(memberId, Errors.INCONSISTENT_GROUP_PROTOCO

```

```

8 // 如果是待决成员, 由于这次分配了成员ID, 故允许加入组
9 } else if (group.isPendingMember(memberId)) {
10     if (groupId.isDefined) {
11         .....
12     } else {
13         .....
14         // 令其加入组
15         addMemberAndRebalance(rebalanceTimeoutMs, sessionTimeoutMs, memberId, group,
16             clientId, clientHost, protocolType, protocols, group, responseCallback)
17     }
18 } else {
19     // 第二部分代码.....
20 }

```

doJoinGroup 方法开头和 doUnkownJoinGroup 非常类似, 也是判断是否处于 Dead 状态, 并且检查协议类型和分区消费分配策略是否与消费者组的相匹配。

不同的是, doJoinGroup 要判断当前申请入组的成员是否是待决成员。如果是的话, 那么, 这次成员已经分配好了成员 ID, 因此, 就直接调用 addMemberAndRebalance 方法令其入组; 如果不是的话, 那么, 方法进入到第 2 部分, 即处理一个非待决成员的入组申请。


第 2 部分

代码如下:

```

1 // 获取该成员的元数据信息
2 val member = group.get(memberId)
3 group.currentState match {
4     // 如果是PreparingRebalance状态
5     case PreparingRebalance =>
6         // 更新成员信息并开始准备Rebalance
7         updateMemberAndRebalance(group, member, protocols, responseCallback)
8     // 如果是CompletingRebalance状态
9     case CompletingRebalance =>
10         // 如果成员以前申请过加入组
11         if (member.matches(protocols)) {
12             // 直接返回当前组信息
13             responseCallback(JoinGroupResult(
14                 members = if (group.isLeader(memberId)) {
15                     group.currentMemberMetadata
16                 } else {
17                     List.empty
18                 }
19             ))
20         }
21     }

```

 复制代码

```
19         },
20         memberId = memberId,
21         generationId = group.generationId,
22         protocolType = group.protocolType,
23         protocolName = group.protocolName,
24         leaderId = group.leaderOrNull,
25         error = Errors.NONE))
26     // 否则, 更新成员信息并开始准备Rebalance
27     } else {
28         updateMemberAndRebalance(group, member, protocols, responseCallback)
29     }
30     // 如果是Stable状态
31     case Stable =>
32         val member = group.get(memberId)
33         // 如果成员是Leader成员, 或者成员变更了分区分配策略
34         if (group.isLeader(memberId) || !member.matches(protocols)) {
35             // 更新成员信息并开始准备Rebalance
36             updateMemberAndRebalance(group, member, protocols, responseCallback)
37         } else {
38             responseCallback(JoinGroupResult(
39                 members = List.empty,
40                 memberId = memberId,
41                 generationId = group.generationId,
42                 protocolType = group.protocolType,
43                 protocolName = group.protocolName,
44                 leaderId = group.leaderOrNull,
45                 error = Errors.NONE))
46         }
47     // 如果是其它状态, 封装异常调用回调函数返回
48     case Empty | Dead =>
49         warn(s"Attempt to add rejoining member $memberId of group ${group.groupId}
50             s"unexpected group state ${group.currentState}")
51         responseCallback(JoinGroupResult(memberId, Errors.UNKNOWN_MEMBER_ID))
```

这部分代码的**第 1 步**, 是获取要加入组成员的元数据信息。

第 2 步, 是查询消费者组的当前状态。这里有 4 种情况。

1. 如果是 PreparingRebalance 状态, 就说明消费者组正要开启 Rebalance 流程, 那么, 调用 updateMemberAndRebalance 方法更新成员信息, 并开始准备 Rebalance 即可。
2. 如果是 CompletingRebalance 状态, 那么, 就判断一下, 该成员的分区消费分配策略与订阅分区列表是否和已保存记录中的一致, 如果相同, 就说明该成员已经应该发起过加入组的操作, 并且 Coordinator 已经批准了, 只是该成员没有收到, 因此, 针对这种

情况, 代码构造一个 `JoinGroupResult` 对象, 直接返回当前的组信息给成员。但是, 如果 `protocols` 不相同, 那么, 就说明成员变更了订阅信息或分配策略, 就要调用 `updateMemberAndRebalance` 方法, 更新成员信息, 并开始准备新一轮 `Rebalance`。

3. 如果是 `Stable` 状态, 那么, 就判断该成员是否是 `Leader` 成员, 或者是它的订阅信息或分配策略发生了变更。如果是这种情况, 就调用 `updateMemberAndRebalance` 方法强迫一次新的 `Rebalance`。否则的话, 返回当前组信息给该成员即可, 通知它们可以发起 `Rebalance` 的下一步操作。
4. 如果这些状态都不是, 而是 `Empty` 或 `Dead` 状态, 那么, 就封装 `UNKNOWN_MEMBER_ID` 异常, 并调用回调函数返回。

可以看到, 这部分代码频繁地调用 `updateMemberAndRebalance` 方法。如果你翻开它的代码, 会发现, 它仅仅做两件事情。

更新组成员信息; 调用 `GroupMetadata` 的 `updateMember` 方法来更新消费者组成员;

准备 `Rebalance`: 这一步的核心思想, 是将消费者组状态变更到 `PreparingRebalance`, 然后创建 `DelayedJoin` 对象, 并交由 `Purgatory`, 等待延时处理加入组操作。


这个方法的代码行数不多, 而且逻辑很简单, 就是变更消费者组状态, 以及处理延时请求并放入 `Purgatory`, 因此, 我不展开说了, 你可以自行阅读下这部分代码。

addMemberAndRebalance 方法

现在, 我们学习下 `doUnknownJoinGroup` 和 `doJoinGroup` 方法都会用到的 `addMemberAndRebalance` 方法。从名字上来看, 它的作用有两个:

向消费者组添加成员;

准备 `Rebalance`。

 复制代码

```
1 private def addMemberAndRebalance(  
2     rebalanceTimeoutMs: Int,
```



```
3    sessionTimeoutMs: Int,
4    memberId: String,
5    groupInstanceId: Option[String],
6    clientId: String,
7    clientHost: String,
8    protocolType: String,
9    protocols: List[(String, Array[Byte])],
10   group: GroupMetadata,
11   callback: JoinCallback): Unit = {
12   // 创建MemberMetadata对象实例
13   val member = new MemberMetadata(
14     memberId, group.groupId, groupInstanceId,
15     clientId, clientHost, rebalanceTimeoutMs,
16     sessionTimeoutMs, protocolType, protocols)
17   // 标识该成员是新成员
18   member.isNew = true
19   // 如果消费者组准备开启首次Rebalance, 设置newMemberAdded为True
20   if (group.is(PreparingRebalance) && group.generationId == 0)
21     group.newMemberAdded = true
22   // 将该成员添加到消费者组
23   group.add(member, callback)
24   // 设置下次心跳超期时间
25   completeAndScheduleNextExpiration(group, member, NewMemberJoinTimeoutMs)
26   if (member.isStaticMember) {
27     info(s"Adding new static member $groupInstanceId to group ${group.groupId}")
28     group.addStaticMember(groupInstanceId, memberId)
29   } else {
30     // 从待决成员列表中移除
31     group.removePendingMember(memberId)
32   }
33   // 准备开启Rebalance
34   maybePrepareRebalance(group, s"Adding new member $memberId with group instan
35
```

这个方法的参数列表虽然很长, 但我相信, 你对它们已经非常熟悉了, 它们都是承袭自其上层调用方法的参数。

我来介绍一下这个方法的执行逻辑。

第 1 步, 该方法会根据传入参数创建一个 MemberMetadata 对象实例, 并设置 isNew 字段为 True, 标识其是一个新成员。isNew 字段与心跳设置相关联, 你可以阅读下 MemberMetadata 的 hasSatisfiedHeartbeat 方法的代码, 搞明白该字段是如何帮助 Coordinator 确认消费者组成员心跳的。

第 2 步，代码会判断消费者组是否是首次开启 Rebalance。如果是的话，就把 `newMemberAdded` 字段设置为 `True`；如果不是，则无需执行这个赋值操作。这个字段的作用，是 Kafka 为消费者组 Rebalance 流程做的一个性能优化。大致的思想，是在消费者组首次进行 Rebalance 时，让 Coordinator 多等待一段时间，从而让更多的消费者组成员加入到组中，以免后来者申请入组而反复进行 Rebalance。这段多等待的时间，就是 Broker 端参数 **`group.initial.rebalance.delay.ms`** 的值。这里的 `newMemberAdded` 字段，就是用于判断是否需要多等待这段时间的一个变量。

我们接着说回 `addMemberAndRebalance` 方法。该方法的**第 3 步**是调用 `GroupMetadata` 的 `add` 方法，将新成员信息加入到消费者组元数据中，同时设置该成员的下次心跳超期时间。

第 4 步，代码将该成员从待决成员列表中移除。毕竟，它已经正式加入到组中了，就不需要待在待决列表中。

第 5 步，调用 `maybePrepareRebalance` 方法，准备开启 Rebalance。

总结

至此，我们学完了 Rebalance 流程的第一大步，也就是加入组的源码学习。在这一步中，你要格外注意，**加入组时是区分有无消费者组成员 ID**。对于未设定成员 ID 的分支，代码调用 `doUnknownJoinGroup` 为成员生成 ID 信息；对于已设定成员 ID 的分支，则调用 `doJoinGroup` 方法。而这两个方法，底层都是调用 `addMemberAndRebalance` 方法，实现将消费者组成员添加进组的逻辑。

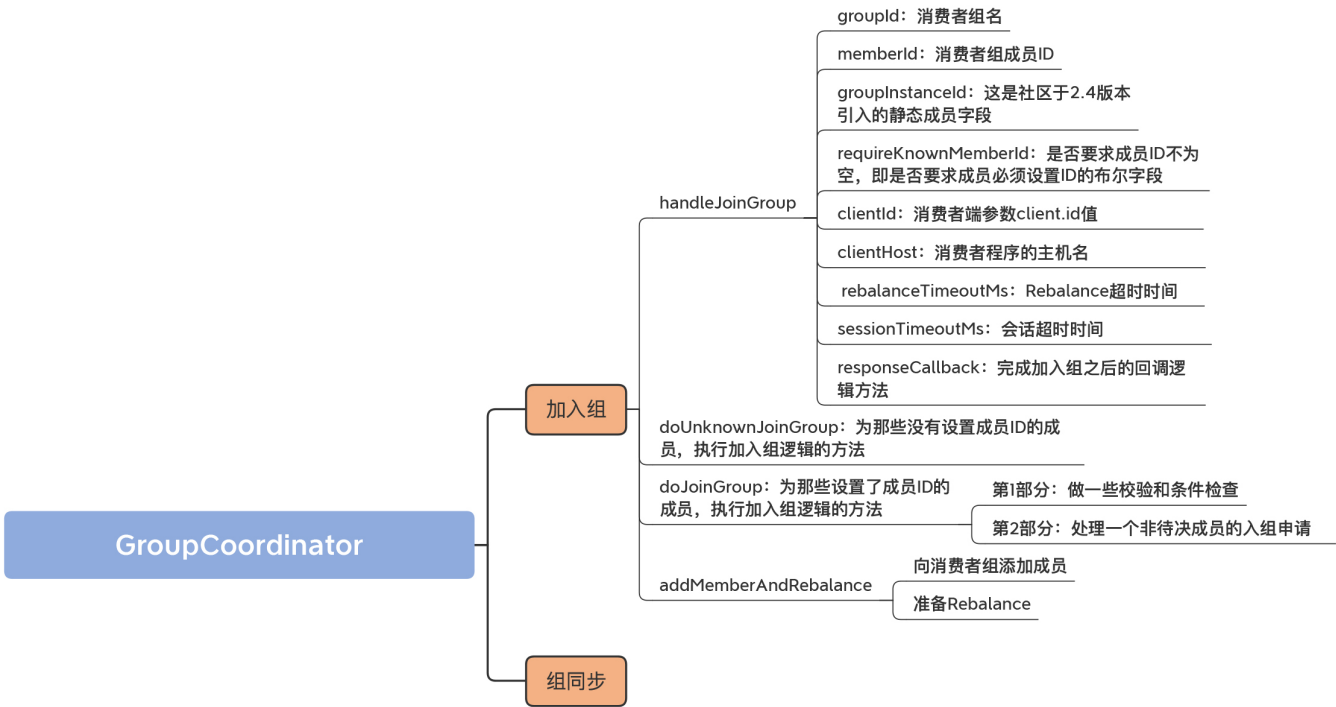
我们来简单回顾一下这节课的重点。

Rebalance 流程：包括 `JoinGroup` 和 `SyncGroup` 两大步。

`handleJoinGroup` 方法：Coordinator 端处理成员加入组申请的方法。

Member Id：成员 ID。Kafka 源码根据成员 ID 的有无，决定调用哪种加入组逻辑方法，比如 `doUnknownJoinGroup` 或 `doJoinGroup` 方法。

`addMemberAndRebalance` 方法：实现加入组功能的实际方法，用于完成“加入组 + 开启 Rebalance”这两个操作。



当所有成员都成功加入到组之后，所有成员会开启 Rebalance 的第二大步：组同步。在这一步中，成员会发送 SyncGroupRequest 请求给 Coordinator。那么，Coordinator 又是如何应对的呢？咱们下节课见分晓。

课后讨论

今天，我们曾多次提到 maybePrepareRebalance 方法，从名字上看，它并不一定会开启 Rebalance。那么，你能否结合源码说说看，到底什么情况下才能开启 Rebalance？

欢迎在留言区写下你的思考和答案，跟我交流讨论，也欢迎你把今天的内容分享给你的朋友。

提建议

更多课程推荐

设计模式之美

前 Google 工程师手把手教你写高质量代码

王争

前 Google 工程师

《数据结构与算法之美》专栏作者



涨价倒计时 🕒

限时秒杀 **¥149**, 7月31日涨价至 **¥299**

© 版权归极客邦科技所有, 未经许可不得传播售卖。页面已增加防盗追踪, 如有侵权极客邦将依法追究其法律责任。

上一篇 31 | GroupMetadataManager: 查询位移时, 不用读取位移主题?

下一篇 33 | GroupCoordinator: 在Rebalance中, 如何进行组同步?

精选留言 (6)

💬 写留言



胡夕 置顶

2020-07-21

你好, 我是胡夕。我来公布上节课的“课后讨论”题答案啦~

上节课, 我们重点学习了GroupMetadataManager类读写位移主题的源码。课后我请你分析cleanGroupMetadata方法的流程。实际上, 这个方法调用带参数的同名方法cleanUpGroupMetadata来执行组位移的清除。后者会遍历给定的所有消费者组, 之后调用re...
展开 ∨



双椒叔叔

2020-07-13

胡老师, 您好

分区迁移遇到的问题怎么解决呢

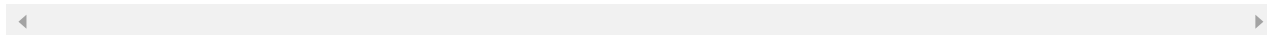
1038, 1037, 1029-----reassign----->1038,1037,1048

其实就是1029机子先宕掉了, 然后我想要把死掉的1029机子上的副本迁移到1048上

...

展开 ✓

作者回复: 删除zk中/admin/reassign_partitions下对应的节点然后重启broker试试呢

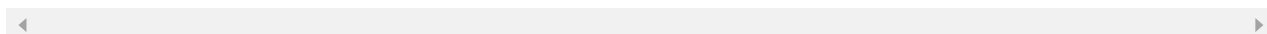


云端漫步

2020-07-25

GroupState是Stable, CompletingRebalance, Empty这三种的情况下才可以

作者回复: 💎💎💎💎💎



Jeff.Smile

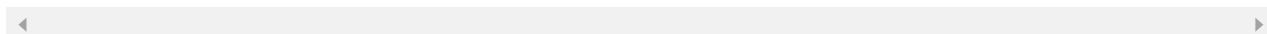
2020-07-22

老师您好, 想问下如何能明确观察到rebalance的过程, 因为有时候知道发生了rebalance, 但是不能确认是什么原因引起的rebalance, 所以想通过一定手段定位下, 比如tcpdump抓包, 但是用wireshark打开并转为kafka协议好像也看不出rebalance的过程, 也可能是我用的不熟练, 想请老师可以补充下这方面的, 毕竟理论结合实践才是解决问题的途径。

补充说一下, 好像server.log也只能看到说Rebalance开始了, Member被移除。。之类...

展开 ✓

作者回复: 打开DEBUG日志, 会有非常详细的日志输出, 重点看看Coordinator部分的DEBUG日志



懂码哥(GerryWen)

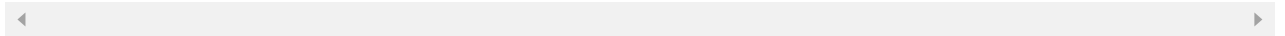
2020-07-13

胡大大, 您的社区名字有点意思。 https://issues.apache.org/jira/secure/ViewProfile.jspa?name=huxi_2b



展开 ✓

作者回复: 额。。。不是你想的那样：)



伯安知心

2020-07-11

从代码上看，进入maybePrepareRebalance的时候，首先把group加入锁中，因为这里要访问消费组的元数据（线程不安全），然后只有一个判断if (group.canRebalance)，这个判断主要是判断消费组元数据中的validPreviousStates的map集合中是否存在PreparingRebalance状态的数据，事实上这个状态就是一个过渡的中间状态比如某些成员加入超时的状态，所有成员离开了组，通过分区迁移删除组。

展开 ∨

作者回复: 🙏

