



下载APP



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

2020-07-14 胡夕

Kafka核心源码解读

[进入课程 >](#)**讲述：胡夕**

时长 18:08 大小 16.62M



你好，我是胡夕。今天，我们继续学习消费者组 Rebalance 流程，这节课我们重点学习这个流程的第 2 大步，也就是组同步。

组同步，也就是成员向 Coordinator 发送 SyncGroupRequest 请求，等待 Coordinator 发送分配方案。在 GroupCoordinator 类中，负责处理这个请求的入口方法就是 handleSyncGroup。它进一步调用 doSyncGroup 方法完成组同步的逻辑。后者除了给成员下发分配方案之外，还需要在元数据缓存中注册组消息，以及把组状态变更为 Stable。一旦完成了组同步操作，Rebalance 宣告结束，消费者组开始正常工作。



接下来，我们就来具体学习下组同步流程的实现逻辑。我们先从顶层的入口方法 handleSyncGroup 方法开始学习，**该方法被 KafkaApis 类的 handleSyncGroupRequest 方法调用，用于处理消费者组成员发送的**

**SyncGroupRequest 请求。**顺着这个入口方法，我们会不断深入，下沉到具体实现组同步逻辑的私有化方法 `doSyncGroup`。

## handleSyncGroup 方法

我们从 `handleSyncGroup` 的方法签名开始学习，代码如下：

 复制代码

```
1 def handleSyncGroup(  
2     groupId: String, // 消费者组名  
3     generation: Int, // 消费者组Generation号  
4     memberId: String, // 消费者组成员ID  
5     protocolType: Option[String], // 协议类型  
6     protocolName: Option[String], // 分区消费分配策略名称  
7     groupInstanceId: Option[String], // 静态成员Instance ID  
8     groupAssignment: Map[String, Array[Byte]], // 按照成员分组的分配方案  
9     responseCallback: SyncCallback // 回调函数  
10 ): Unit = {  
11     .....  
12 }
```

该方法总共定义了 8 个参数，你可以看下注释，了解它们的含义，我重点介绍 6 个比较关键的参数。

**groupId**：消费者组名，标识这个成员属于哪个消费者组。

**generation**：消费者组 Generation 号。Generation 类似于任期的概念，标识了 Coordinator 负责为该消费者组处理的 Rebalance 次数。每当有新的 Rebalance 开启时，Generation 都会自动加 1。

**memberId**：消费者组成员 ID。该字段由 Coordinator 根据一定的规则自动生成。具体的规则上节课我们已经学过了，我就不多说了。总体而言，成员 ID 的值不是由你直接指定的，但是你可以通过 `client.id` 参数，间接影响该字段的取值。

**protocolType**：标识协议类型的字段，这个字段可能的取值有两个：`consumer` 和 `connect`。对于普通的消费者组而言，这个字段的取值就是 `consumer`，该字段是 `Option` 类型，因此，实际的取值是 `Some("consumer")`；Kafka Connect 组件中也会用到消费者组机制，那里的消费者组的取值就是 `connect`。

**protocolName**：消费者组选定的分区消费分配策略名称。这里的选择方法，就是我们之前学到的 `GroupMetadata.selectProtocol` 方法。

**groupAssignment**：按照成员 ID 分组的分配方案。需要注意的是，**只有 Leader 成员发送的 SyncGroupRequest 请求，才包含这个方案**，因此，Coordinator 在处理 Leader 成员的请求时，该字段才有值。

你可能已经注意到了，protocolType 和 protocolName 都是 Option 类型，这说明，它们的取值可能是 None，即表示没有值。这是为什么呢？

目前，这两个字段的取值，其实都是 Coordinator 帮助消费者组确定的，也就是在 Rebalance 流程的上一步加入组中确定的。

如果成员成功加入组，那么，Coordinator 会给这两个字段赋上正确的值，并封装进 JoinGroupRequest 的 Response 里，发送给消费者程序。一旦消费者拿到了 Response 中的数据，就提取出这两个字段的值，封装进 SyncGroupRequest 请求中，再次发送给 Coordinator。

如果成员没有成功加入组，那么，Coordinator 会将这两个字段赋值成 None，加到 Response 中。因此，在这里的 handleSyncGroup 方法中，它们的类型就是 Option。

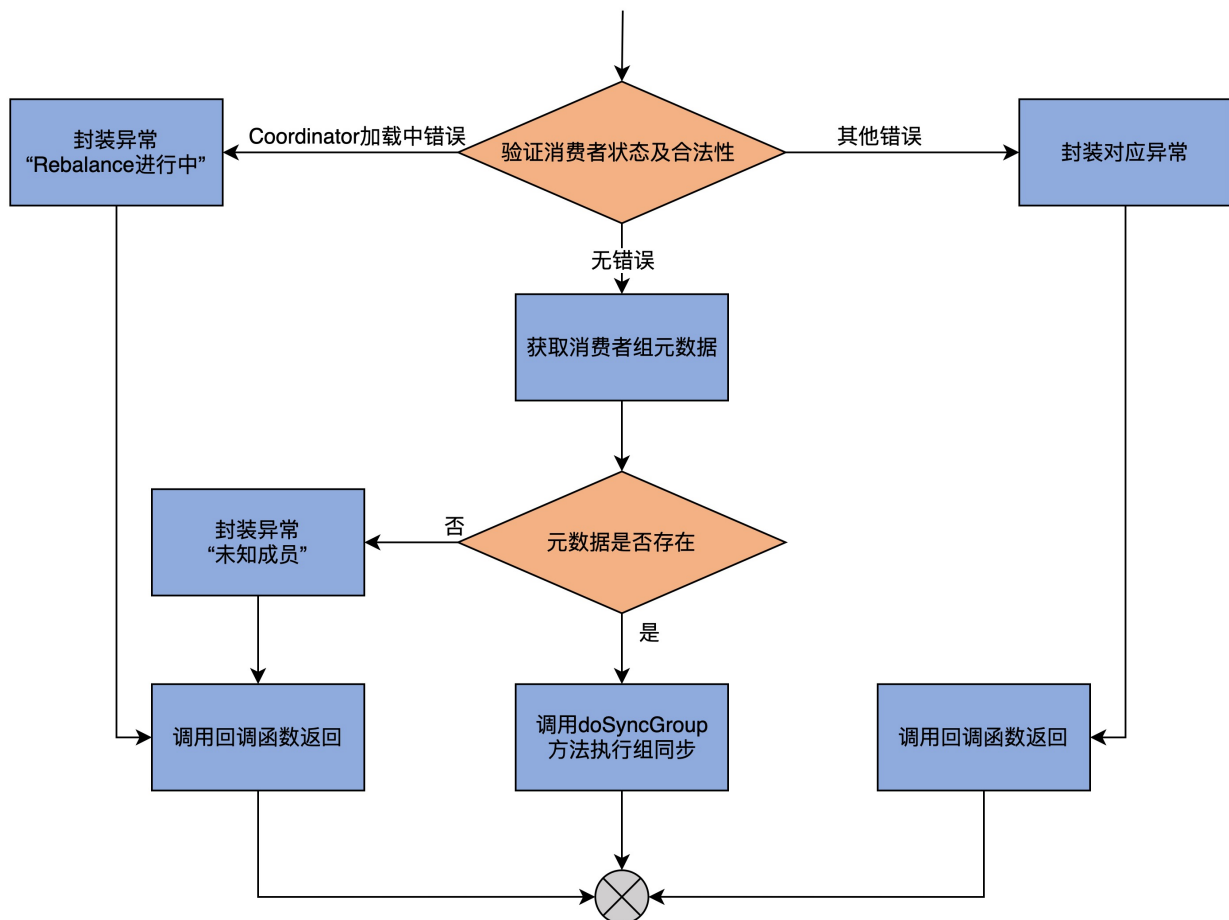
说完了 handleSyncGroup 的方法签名，我们看下它的代码：

[复制代码](#)

```
1 // 验证消费者状态及合法性
2 validateGroupStatus(groupId, ApiKeys.SYNC_GROUP) match {
3   // 如果未通过合法性检查，且错误原因是Coordinator正在加载
4   // 那么，封装REBALANCE_IN_PROGRESS异常，并调用回调函数返回
5   case Some(error) if error == Errors.COORDINATOR_LOAD_IN_PROGRESS =>
6     responseCallback(SyncGroupResult(Errors.REBALANCE_IN_PROGRESS))
7   // 如果是其它错误，则封装对应错误，并调用回调函数返回
8   case Some(error) => responseCallback(SyncGroupResult(error))
9   case None =>
10    // 获取消费者组元数据
11    groupManager.getGroup(groupId) match {
12      // 如果未找到，则封装UNKNOWN_MEMBER_ID异常，并调用回调函数返回
13      case None =>
14        responseCallback(SyncGroupResult(Errors.UNKNOWN_MEMBER_ID))
15      // 如果找到的话，则调用doSyncGroup方法执行组同步任务
16      case Some(group) => doSyncGroup(
17        group, generation, memberId, protocolType, protocolName,
18        groupInstanceId, groupAssignment, responseCallback)
19    }
20 }
```



为了方便你理解，我画了一张流程图来说明此方法的主体逻辑。



`handleSyncGroup` 方法首先会调用上一节课我们学习过的 `validateGroupStatus` 方法，校验消费者组状态及合法性。这些检查项包括：

1. 消费者组名不能为空；
2. Coordinator 组件处于运行状态；
3. Coordinator 组件当前没有执行加载过程；
4. `SyncGroupRequest` 请求发送给了正确的 Coordinator 组件。

前两个检查项很容易理解，我重点解释一下最后两项的含义。

当 Coordinator 变更到其他 Broker 上时，需要从内部位移主题中读取消息数据，并填充到内存上的消费者组元数据缓存，这就是所谓的加载。

如果 Coordinator 变更了，那么，发送给老 Coordinator 所在 Broker 的请求就失效了，因为它没有通过第 4 个检查项，即发送给正确的 Coordinator；

如果发送给了正确的 Coordinator，但此时 Coordinator 正在执行加载过程，那么，它就没有通过第 3 个检查项，因为 Coordinator 尚不能对外提供服务，要等加载完成之后才可以。

代码对消费者组依次执行上面这 4 项校验，一旦发现有项目校验失败，validateGroupStatus 方法就会将检查失败的原因作为结果返回。如果是因为 Coordinator 正在执行加载，就意味着**本次 Rebalance 的所有状态都丢失了**。这里的状态，指的是消费者组下的成员信息。那么，此时最安全的做法，是**让消费者组重新从加入组开始**，因此，代码会封装 REBALANCE\_IN\_PROGRESS 异常，然后调用回调函数返回。一旦消费者组成员接收到此异常，就会知道，它至少找到了正确的 Coordinator，只需要重新开启 Rebalance，而不需要在开启 Rebalance 之前，再大费周章地去定位 Coordinator 组件了。但如果是其它错误，就封装该错误，然后调用回调函数返回。

倘若消费者组通过了以上校验，那么，代码就会获取该消费者组的元数据信息。如果找不到对应的元数据，就封装 UNKNOWN\_MEMBER\_ID 异常，之后调用回调函数返回；如果找到了元数据信息，就调用 doSyncGroup 方法执行真正的组同步逻辑。

显然，接下来我们应该学习 doSyncGroup 方法的源码了，这才是真正实现组同步功能的地方。

## doSyncGroup 方法

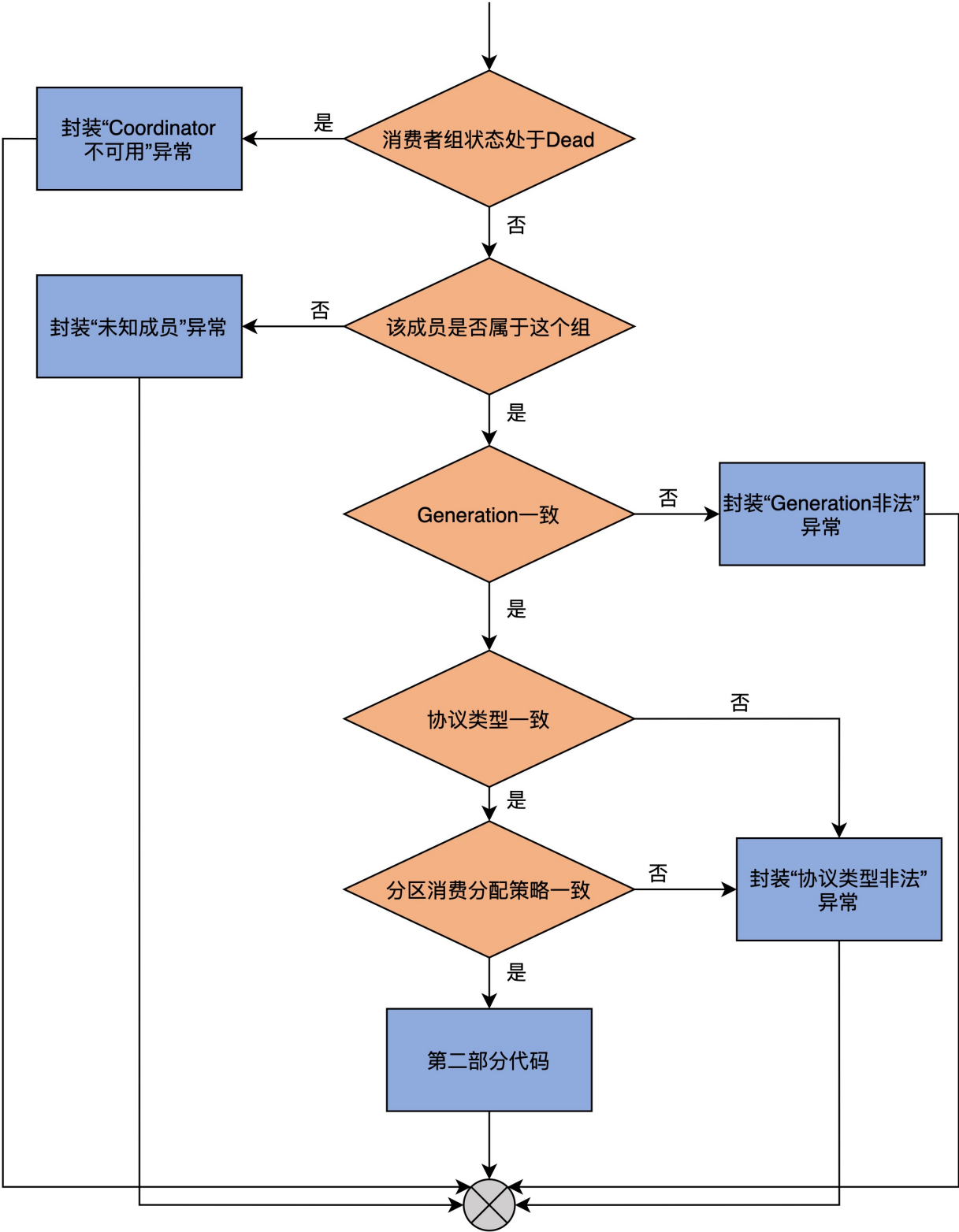
doSyncGroup 方法接收的输入参数，与它的调用方法 handleSyncGroup 如出一辙，所以这里我就不再展开讲了，我们重点关注一下它的源码实现。

鉴于它的代码很长，我把它拆解成两个部分，并配以流程图进行介绍。

第 1 部分：主要**对消费者组做各种校验**，如果没有通过校验，就封装对应的异常给回调函数；

第 2 部分：**根据不同的消费者组状态选择不同的执行逻辑。** 你需要特别关注一下，在 CompletingRebalance 状态下，代码是如何实现组同步的。

我先给出第 1 部分的流程图，你可以先看一下，对这个流程有个整体的感知。



下面，我们来看这部分的代码：

[复制代码](#)

```
1  if (group.is(Dead)) {
2      responseCallback(
3          SyncGroupResult(Errors.COORDINATOR_NOT_AVAILABLE))
4  } else if (group.isStaticMemberFenced(memberId, groupInstanceId, "sync-group"))
5      responseCallback(SyncGroupResult(Errors.FENCED_INSTANCE_ID))
6  } else if (!group.has(memberId)) {
7      responseCallback(SyncGroupResult(Errors.UNKNOWN_MEMBER_ID))
8  } else if (generationId != group.generationId) {
9      responseCallback(SyncGroupResult(Errors.ILLEGAL_GENERATION))
10 } else if (protocolType.isDefined && !group.protocolType.contains(protocolType)
11 responseCallback(SyncGroupResult(Errors.INCONSISTENT_GROUP_PROTOCOL))
12 } else if (protocolName.isDefined && !group.protocolName.contains(protocolName)
13 responseCallback(SyncGroupResult(Errors.INCONSISTENT_GROUP_PROTOCOL))
14 } else {
15     // 第2部分源码.....
16 }
```

可以看到，代码非常工整，全是 if-else 类型的判断。

**首先**，这部分代码会判断消费者组的状态是否是 Dead。如果是的话，就说明该组的元数据信息已经被其他线程从 Coordinator 中移除了，这很可能是因为 Coordinator 发生了变更。此时，最佳的做法是**拒绝该成员的组同步操作**，封装 COORDINATOR\_NOT\_AVAILABLE 异常，显式告知它去寻找最新 Coordinator 所在的 Broker 节点，然后再尝试重新加入组。

接下来的 isStaticMemberFenced 方法判断是有关静态成员的，我们可以不用理会。

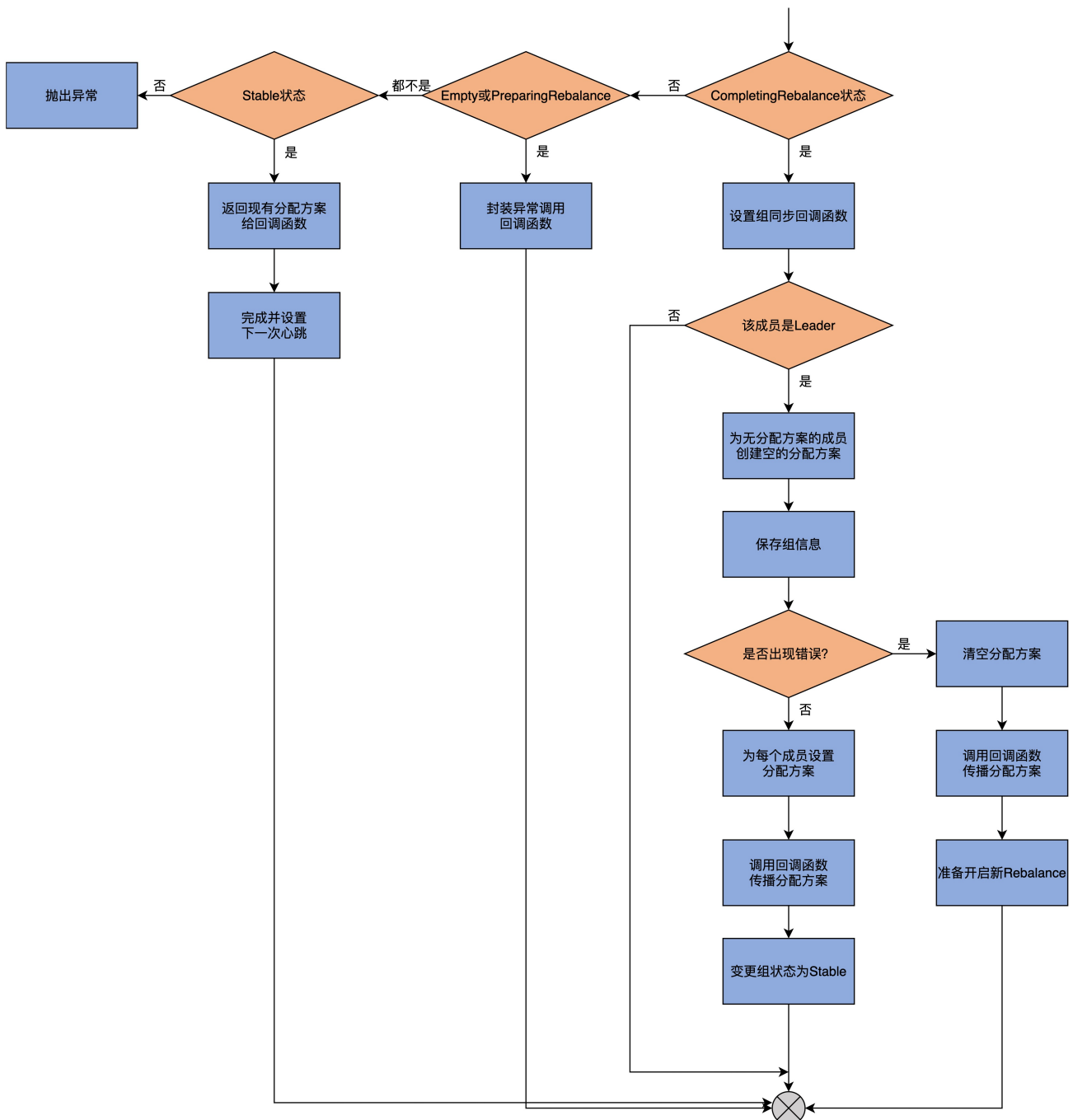
**之后**，代码判断 memberId 字段标识的成员是否属于这个消费者组。如果不属于的话，就封装 UNKNOWN\_MEMBER\_ID 异常，并调用回调函数返回；如果属于的话，则继续下面的判断。

**再之后**，代码**判断成员的 Generation 是否和消费者组的相同**。如果不同的话，则封装 ILLEGAL\_GENERATION 异常给回调函数；如果相同的话，则继续下面的判断。

接下来，代码**判断成员和消费者组的协议类型是否一致**。如果不一致，则封装 INCONSISTENT\_GROUP\_PROTOCOL 异常给回调函数；如果一致，就进行下一步。


最后，判断**成员和消费者组的分区消费分配策略是否一致**。如果不一致，同样封装 INCONSISTENT\_GROUP\_PROTOCOL 异常给回调函数。

如果这些都一致，则顺利进入到第 2 部分。在开始之前，我依然用一张图来展示一下这里的实现逻辑。



进入到这部分之后，代码要做什么事情，完全**取决于消费者组的当前状态**。如果消费者组处于 CompletingRebalance 状态，这部分代码要做的事情就比较复杂，我们一会儿再说，现在先看除了这个状态之外的逻辑代码。



 复制代码


```
1 group.currentState match {
2   case Empty =>
3     // 封装UNKNOWN_MEMBER_ID异常，调用回调函数返回
4     responseCallback(SyncGroupResult(Errors.UNKNOWN_MEMBER_ID))
5   case PreparingRebalance =>
6     // 封装REBALANCE_IN_PROGRESS异常，调用回调函数返回
7     responseCallback(SyncGroupResult(Errors.REBALANCE_IN_PROGRESS))
8   case CompletingRebalance =>
9     // 下面详细展开.....
10  case Stable =>
11    // 获取消费者组成员元数据
12    val memberMetadata = group.get(memberId)
13    // 封装组协议类型、分配策略、成员分配方案，调用回调函数返回
14    responseCallback(SyncGroupResult(group.protocolType, group.protocolName, m
15    // 设定成员下次心跳时间
16    completeAndScheduleNextHeartbeatExpiration(group, group.get(memberId))
17  case Dead =>
18    // 抛出异常
19    throw new IllegalStateException(s"Reached unexpected condition for Dead gr
20 }
```

如果消费者组的当前状态是 Empty 或 PreparingRebalance，那么，代码会封装对应的异常给回调函数，供其调用。

如果是 Stable 状态，则说明，此时消费者组已处于正常工作状态，无需进行组同步的操作。因此，在这种情况下，简单返回消费者组当前的分配方案给回调函数，供它后面发送给消费者组成员即可。

如果是 Dead 状态，那就说明，这是一个异常的情况了，因为理论上，不应该为处于 Dead 状态的组执行组同步，因此，代码只能选择抛出 IllegalStateException 异常，让上层方法处理。

如果这些状态都不是，那么，消费者组就只能处于 CompletingRebalance 状态，这也是执行组同步操作时消费者组最有可能处于的状态。因此，这部分的逻辑要复杂一些，我们看下代码：

 复制代码

```
1 // 为该消费者组成员设置组同步回调函数
2 group.get(memberId).awaitingSyncCallback = responseCallback
3 // 组Leader成员发送的SyncGroupRequest请求需要特殊处理
```

```
4  if (group.isLeader(memberId)) {
5      info(s"Assignment received from leader for group ${group.groupId} for genera
6      // 如果有成员没有被分配任何消费方案, 则创建一个空的方案赋给它
7      val missing = group.allMembers.diff(groupAssignment.keySet)
8      val assignment = groupAssignment ++ missing.map(_ -> Array.empty[Byte]).toMa
9
10     if (missing.nonEmpty) {
11         warn(s"Setting empty assignments for members $missing of ${group.groupId}
12     }
13     // 把消费者组信息保存在消费者组元数据中, 并且将其写入到内部位移主题
14     groupManager.storeGroup(group, assignment, (error: Errors) => {
15         group.inLock {
16             // 如果组状态是CompletingRebalance以及成员和组的generationId相同
17             if (group.is(CompletingRebalance) && generationId == group.generationId)
18                 // 如果有错误
19                 if (error != Errors.NONE) {
20                     // 清空分配方案并发送给所有成员
21                     resetAndPropagateAssignmentError(group, error)
22                     // 准备开启新一轮的Rebalance
23                     maybePrepareRebalance(group, s"error when storing group assignment d
24                 // 如果没错误
25             } else {
26                 // 在消费者组元数据中保存分配方案并发送给所有成员
27                 setAndPropagateAssignment(group, assignment)
28                 // 变更消费者组状态到Stable
29                 group.transitionTo(Stable)
30             }
31         }
32     }
33 })
34 groupCompletedRebalanceSensor.record()
35
```

第 1 步, 为该消费者组成员设置组同步回调函数。我们总说回调函数, 其实它的含义很简单, 也就是将传递给回调函数的数据, 通过 Response 的方式发送给消费者组成员。

第 2 步, 判断当前成员是否是消费者组的 Leader 成员。如果不是 Leader 成员, 方法直接结束, 因为, 只有 Leader 成员的 groupAssignment 字段才携带了分配方案, 其他成员是没有分配方案的; 如果是 Leader 成员, 则进入到下一步。

第 3 步, 为没有分配到任何分区的成员创建一个空的分配方案, 并赋值给这些成员。这一步的主要目的, 是构造一个统一格式的分配方案字段 assignment。

第 4 步, 调用 storeGroup 方法, 保存消费者组信息到消费者组元数据, 同时写入到内部位移主题中。一旦完成这些动作, 则进入到下一步。

第 5 步，在组状态是 CompletingRebalance，而且成员和组的 Generation ID 相同的情况下，就判断一下刚刚的 storeGroup 操作过程中是否出现过错误：

如果有错误，则清空分配方案并发送给所有成员，同时准备开启新一轮的 Rebalance；

如果没有错误，则在消费者组元数据中保存分配方案，然后发送给所有成员，并将消费者组状态变更到 Stable。

倘若组状态不是 CompletingRebalance，或者是成员和组的 Generation ID 不相同，这就说明，消费者组可能开启了新一轮的 Rebalance，那么，此时就不能继续给成员发送分配方案。

至此，CompletingRebalance 状态下的组同步操作完成。总结一下，组同步操作完成了以下 3 件事情：

1. 将包含组成员分配方案的消费者组元数据，添加到消费者组元数据缓存以及内部位移主题中；
2. 将分配方案通过 SyncGroupRequest 响应的方式，下发给组下所有成员。
3. 将消费者组状态变更到 Stable。

我建议你对照着代码，自行寻找并阅读一下完成这 3 件事情的源码，这不仅有助于你复习下今天所学的内容，还可以帮你加深对源码的理解。阅读的时候，你思考一下，这些代码的含义是否真的如我所说。如果你有不一样的理解，欢迎写在留言区，我们可以开放式讨论。

## 总结

今天，我们重点学习了 Rebalance 流程的第 2 步，也就是组同步。至此，关于 Rebalance 的完整流程，我们就全部学完了。

Rebalance 流程是 Kafka 提供的一个非常关键的消费者组功能。由于它非常重要，所以，社区在持续地对它进行着改进，包括引入增量式的 Rebalance 以及静态成员等。我们在这两节课学的 Rebalance 流程，是理解这些高级功能的基础。如果你不清楚 Rebalance 过程中的这些步骤都是做什么的，你就无法深入地掌握增量式 Rebalance 或静态成员机制所做的事情。

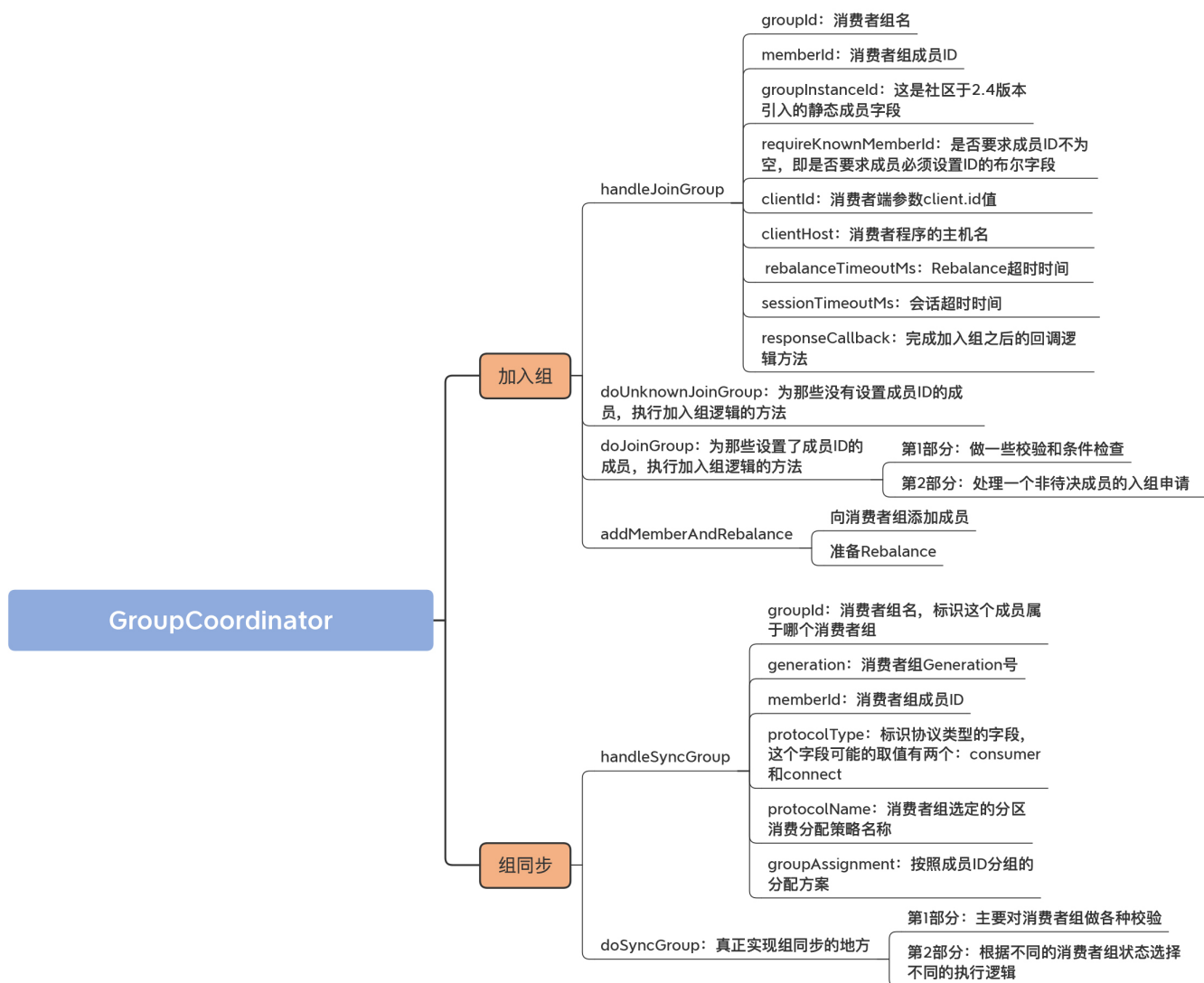
因此，我建议你结合上节课的内容，好好学习一下消费者组的 Rebalance，彻底弄明白一个消费者组成员是如何参与其中并最终完成 Rebalance 过程的。

我们来回顾一下这节课的重点。

组同步：成员向 Coordinator 发送 SyncGroupRequest 请求以获取分配方案。

handleSyncGroup 方法：接收 KafkaApis 发来的 SyncGroupRequest 请求体数据，执行组同步逻辑。

doSyncGroup 方法：真正执行组同步逻辑的方法，执行组元数据保存、分配方案下发以及状态变更操作。



讲到这里，Coordinator 组件的源码，我就介绍完了。在这个模块中，我们基本上还是践行“自上而下 + 自下而上”的学习方式。我们先从最低层次的消费者组元数据类开始学

习，逐渐上浮到它的管理器类 GroupMetadataManager 类以及顶层类 GroupCoordinator 类。接着，在学习 Rebalance 流程时，我们反其道而行之，先从 GroupCoordinator 类的入口方法进行拆解，又逐渐下沉到 GroupMetadataManager 和更底层的 GroupMetadata 以及 MemberMetadata。

如果你追随着课程的脚步一路走来，你就会发现，我经常采用这种方式讲解源码。我希望，你在日后的源码学习中，也可以多尝试运用这种方法。所谓择日不如撞日，我今天就给你推荐一个课后践行此道的绝佳例子。

我建议你去阅读下 clients 工程中的实现消息、消息批次以及消息集合部分的源码，也就是 Record、RecordBatch 和 Records 这些接口和类的代码，去反复实践“自上而下”和“自下而上”这两种阅读方法。

其实，这种方式不只适用于 Kafka 源码，在阅读其他框架的源码时，也可以采用这种方式。希望你不断总结经验，最终提炼出一套适合自己的学习模式。

## 课后讨论

Coordinator 不会将所有消费者组的所有成员的分配方案下发给单个成员，这就是说，成员 A 看不到成员 B 的分区消费分配方案。那么，你能找出来，源码中的哪行语句做了这件事情吗？

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

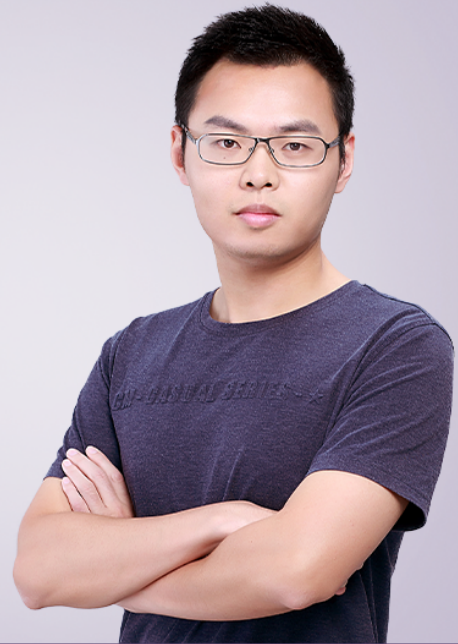
# 设计模式之美

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

下一篇 特别放送 (一) | 经典的Kafka学习资料有哪些?

## 精选留言 (3)

💬 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GroupCoordinator执行Rebalance第一步加入组的代码。课后我请你思考一下maybePrepareRebalance方法满足什么条件才会开启Rebalance? 如果你翻开maybePrepareRebalance的代码, 可以看到它会调用canRebalance方法执行是...  
展开 ∨



常想一二

2020-07-27

老师, 您好, 请教个问题, 下面的报错是什么原因导致的, 要怎么查找问题的原因并解决

java.lang.IllegalStateException: Correlation id for response (62213091) does not match request (62213090),

展开 ✓



**伯安知心**

2020-07-14

handleSyncGroup方法验证组状态的时候validateGroupStatus通过模式匹配如果发现没有错误，执行查找自己broker保存的组groupManager.getGroup(groupId) 然后每个组依次执行自己的doSyncGroup方法。

