



下载APP



28 | 消费者组元数据（下）：Kafka 如何管理这些元数据？

2020-07-02 胡夕

Kafka 核心源码解读

[进入课程 >](#)**讲述：胡夕**

时长 21:15 大小 19.47M



你好，我是胡夕。今天我们继续学习消费者组元数据。

学完上节课之后，我们知道，Kafka 定义了非常多的元数据，那么，这就必然涉及到对元数据的管理问题了。

这些元数据的类型不同，管理策略也就不一样。这节课，我将从消费者组状态、成员、位移和分区分配策略四个维度，对这些元数据进行拆解，带你一起了解下 Kafka 管理这些元数据的方法。



这些方法定义在 MemberMetadata 和 GroupMetadata 这两个类中，其中，GroupMetadata 类中的方法最为重要，是我们要重点学习的对象。在后面的课程中，你

会看到，这些方法会被上层组件 GroupCoordinator 频繁调用，因此，它们是我们学习 Coordinator 组件代码的前提条件，你一定要多花些精力搞懂它们。

消费者组状态管理方法

消费者组状态是很重要的一类元数据。管理状态的方法，要做的事情也就是设置和查询。这些方法大多比较简单，所以我把它们汇总在一起，直接介绍给你。

[复制代码](#)

```
1 // GroupMetadata.scala
2 // 设置/更新状态
3 def transitionTo(groupState: GroupState): Unit = {
4   assertValidTransition(groupState) // 确保是合法的状态转换
5   state = groupState // 设置状态到给定状态
6   currentStateTimestamp = Some(time.milliseconds()) // 更新状态变更时间戳
7 // 查询状态
8 def currentState = state
9 // 判断消费者组状态是指定状态
10 def is(groupState: GroupState) = state == groupState
11 // 判断消费者组状态不是指定状态
12 def not(groupState: GroupState) = state != groupState
13 // 消费者组能否Rebalance的条件是当前状态是PreparingRebalance状态的合法前置状态
14 def canRebalance = PreparingRebalance.validPreviousStates.contains(state)
```

1.transitionTo 方法

transitionTo 方法的作用是**将消费者组状态变更成给定状态**。在变更前，代码需要确保这次变更必须是合法的状态转换。这是依靠每个 GroupState 实现类定义的 **validPreviousStates 集合** 来完成的。只有在这个集合中的状态，才是合法的前置状态。简单来说，只有集合中的这些状态，才能转换到当前状态。

同时，该方法还会**更新状态变更的时间戳字段**。Kafka 有个定时任务，会定期清除过期的消费者组位移数据，它就是依靠这个时间戳字段，来判断过期与否的。

2.canRebalance 方法

它用于判断消费者组是否能够开启 Rebalance 操作。判断依据是，**当前状态是否是 PreparingRebalance 状态的合法前置状态**。只有 **Stable**、**CompletingRebalance** 和 **Empty** 这 3 类状态的消费者组，才有资格开启 Rebalance。

3.is 和 not 方法

至于 is 和 not 方法，它们分别判断消费者组的状态与给定状态吻合还是不吻合，主要被用于**执行状态校验**。特别是 is 方法，被大量用于上层调用代码中，执行各类消费者组管理任务的前置状态校验工作。

总体来说，管理消费者组状态数据，依靠的就是这些方法，还是很简单的吧？

成员管理方法

在介绍管理消费者组成员的方法之前，我先帮你回忆下 GroupMetadata 中保存成员的字 段。GroupMetadata 中使用 members 字段保存所有的成员信息。该字段是一个 HashMap，Key 是成员的 member ID 字段，Value 是 MemberMetadata 类型，该类型 保存了成员的元数据信息。

所谓的管理成员，也就是添加成员（add 方法）、移除成员（remove 方法）和查询成员（has、get、size 方法等）。接下来，我们就逐一来学习。

添加成员

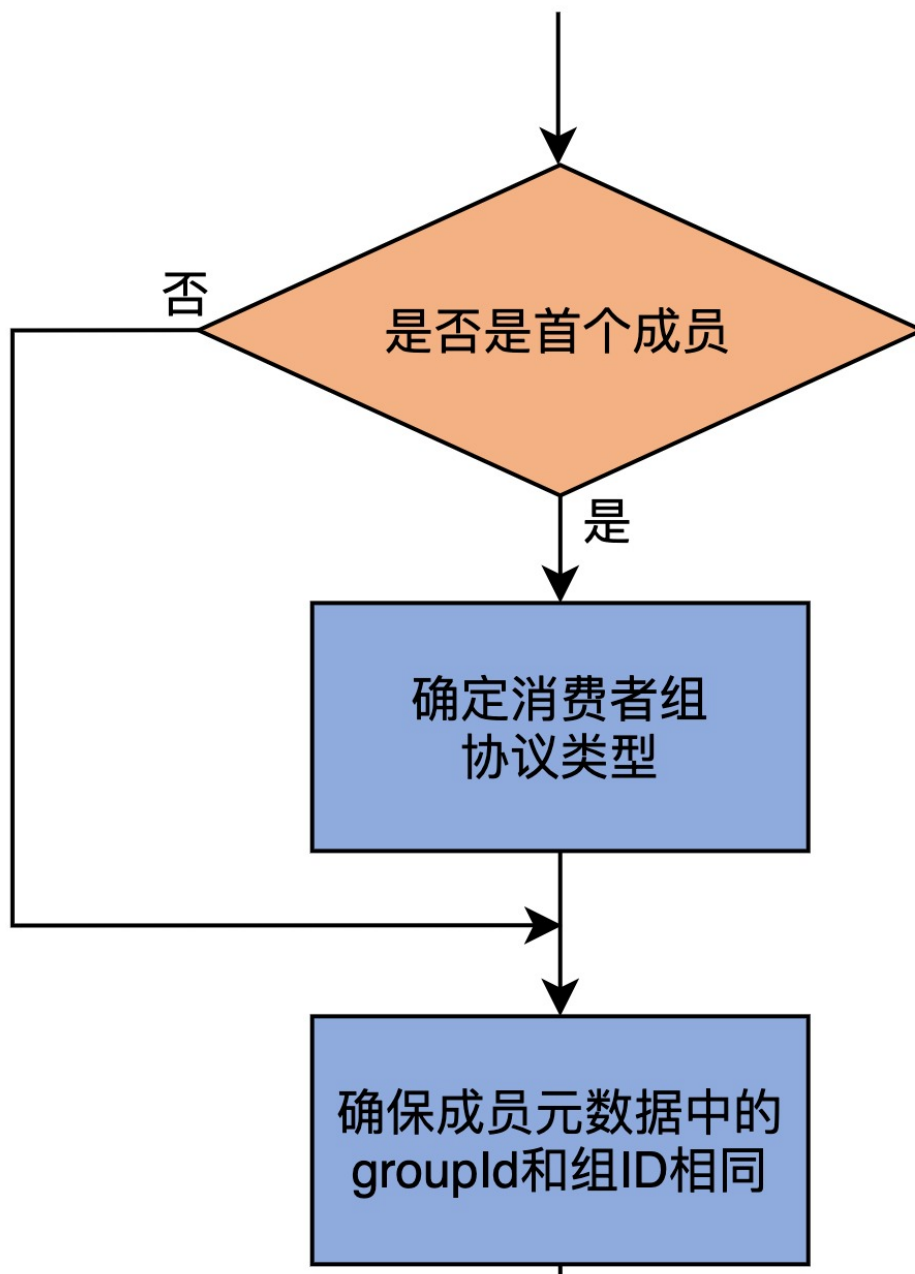
先说添加成员的方法：add。add 方法的主要逻辑，是将成员对象添加到 members 字段，同时更新其他一些必要的元数据，比如 Leader 成员字段、分区分配策略支持票数等。下面是 add 方法的源码：

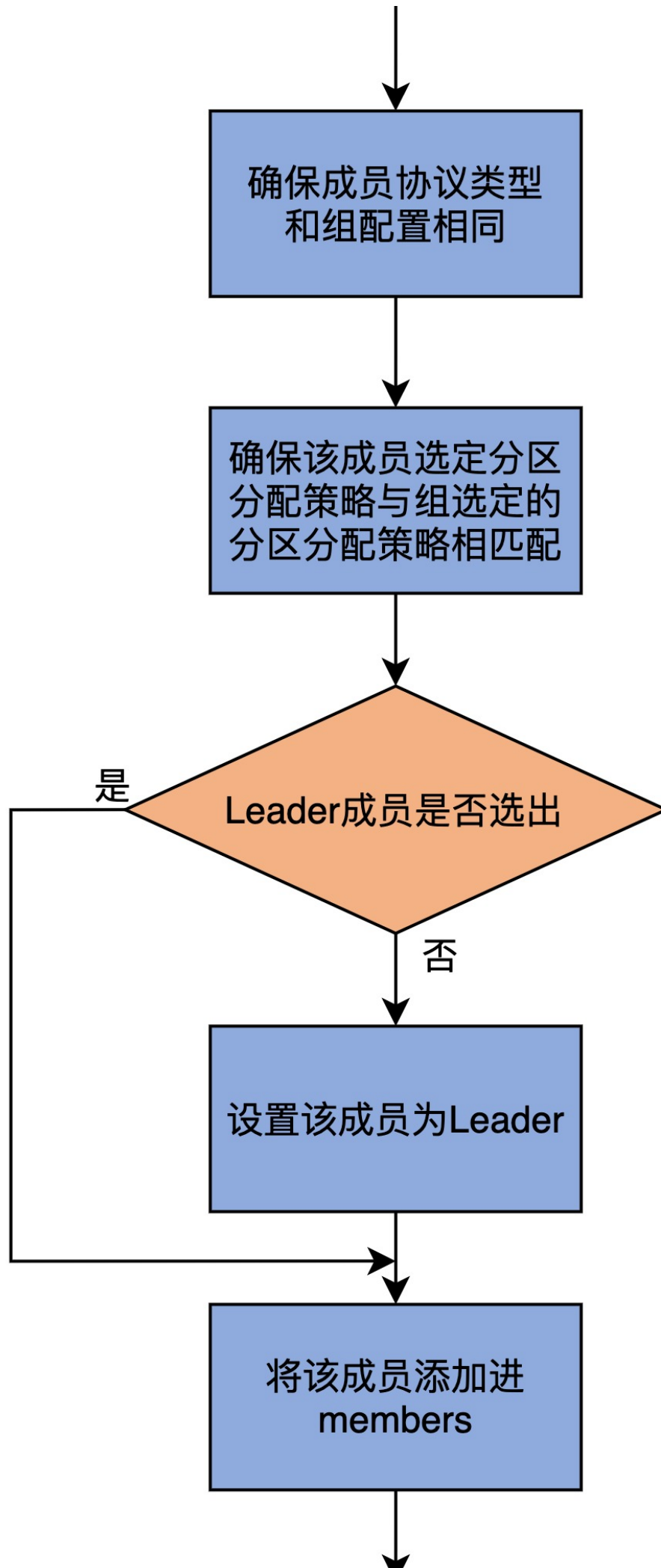
 复制代码

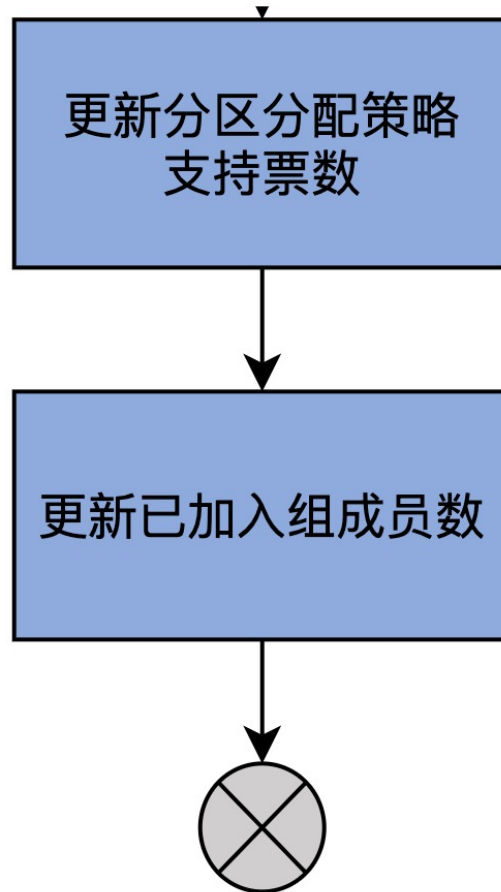
```
1 def add(member: MemberMetadata, callback: JoinCallback = null): Unit = {
2   // 如果是要添加的第一个消费者组成员
3   if (members.isEmpty)
4     // 就把该成员的protocolType设置为消费者组的protocolType
5     this.protocolType = Some(member.protocolType)
6   // 确保成员元数据中的groupId和组Id相同
7   assert(groupId == member.groupId)
8   // 确保成员元数据中的protocolType和组protocolType相同
9   assert(this.protocolType.orNull == member.protocolType)
10  // 确保该成员选定的分区分配策略与组选定的分区分配策略相匹配
11  assert(supportsProtocols(member.protocolType, MemberMetadata.plainProtocolSe
12  // 如果尚未选出Leader成员
13  if (leaderId.isEmpty)
14    // 将该成员设定为Leader成员
15    leaderId = Some(member.memberId)
16  // 将该成员添加进members
```

```
17 members.put(member.memberId, member)
18 // 更新分区分配策略支持票数
19 member.supportedProtocols.foreach{ case (protocol, _) => supportedProtocols(
20 // 设置成员加入组后的回调逻辑
21 member.awaitingJoinCallback = callback
22 // 更新已加入组的成员数
23 if (member.isAwaitingJoin)
24     numMembersAwaitingJoin += 1
25 }
```

我再画一张流程图，帮助你更直观地理解这个方法的作用。







我再具体解释一下这个方法的执行逻辑。

第一步，add 方法要判断 members 字段是否包含已有成员。如果没有，就说明要添加的成员是该消费者组的第一个成员，那么，就令该成员协议类型（protocolType）成为组的 protocolType。我在上节课中讲过，对于普通的消费者而言，protocolType 就是字符串"consumer"。如果不是首个成员，就进入到下一步。

第二步，add 方法会连续进行三次校验，分别确保**待添加成员的组 ID、protocolType** 和组配置一致，以及该成员选定的分区分配策略与组选定的分区分配策略相匹配。如果这些校验有任何一个未通过，就会立即抛出异常。

第三步，判断消费者组的 Leader 成员是否已经选出了。如果还没有选出，就将该成员设置成 Leader 成员。当然了，如果 Leader 已经选出了，自然就不需要做这一步了。需要注意的是，这里的 Leader 和我们在学习副本管理器时学到的 Leader 副本是不同的概念。这里

的 Leader 成员，是指**消费者组下的一个成员**。该成员负责为所有成员制定分区分配方案，制定方法的依据，就是消费者组选定的分区分配策略。


第四步，更新消费者组分区分配策略支持票数。关于 supportedProtocols 字段的含义，我在上节课的末尾用一个例子形象地进行了说明，这里就不再重复说了。如果你没有印象了，可以再复习一下。

最后一步，设置成员加入组后的回调逻辑，同时更新已加入组的成员数。至此，方法结束。

作为关键的成员管理方法之一，add 方法是实现消费者组 Rebalance 流程至关重要的一环。每当 Rebalance 开启第一大步——加入组的操作时，本质上就是在利用这个 add 方法实现新成员入组的逻辑。

移除成员

有 add 方法，自然也就有 remove 方法，下面是 remove 方法的完整源码：

 复制代码

```
1 def remove(memberId: String): Unit = {
2   // 从members中移除给定成员
3   members.remove(memberId).foreach { member =>
4     // 更新分区分配策略支持票数
5     member.supportedProtocols.foreach{ case (protocol, _) => supportedProtocol
6     // 更新已加入组的成员数
7     if (member.isAwaitingJoin)
8       numMembersAwaitingJoin -= 1
9   }
10  // 如果该成员是Leader，选择剩下成员列表中的第一个作为新的Leader成员
11  if (isLeader(memberId))
12    leaderId = members.keys.headOption
13 }
```

remove 方法比 add 要简单一些。**首先**，代码从 members 中移除给定成员。**之后**，更新分区分配策略支持票数，以及更新已加入组的成员数。**最后**，代码判断该成员是否是 Leader 成员，如果是的话，就选择成员列表中尚存的第一个成员作为新的 Leader 成员。

查询成员

查询 members 的方法有很多，大多都是很简单的场景。我给你介绍 3 个比较常见的。

 复制代码

```
1 def has(memberId: String) = members.contains(memberId)
2 def get(memberId: String) = members(memberId)
3 def size = members.size
```

has 方法，判断消费者组是否包含指定成员；

get 方法，获取指定成员对象；

size 方法，统计总成员数。

其它的查询方法逻辑也都很简单，比如 allMemberMetadata、rebalanceTimeoutMs，等等，我就不多讲了。课后你可以自行阅读下，重点是体会这些方法利用 members 都做了什么事情。

位移管理方法

除了组状态和成员管理之外，GroupMetadata 还有一大类管理功能，就是**管理消费者组的提交位移**（Committed Offsets），主要包括添加和移除位移值。

不过，在学习管理位移的功能之前，我再带你回顾一下保存位移的 offsets 字段的定义。毕竟，接下来我们要学习的方法，主要操作的就是这个字段。

 复制代码

```
1 private val offsets = new mutable.HashMap[TopicPartition, CommitRecordMetadata]
```

它是 HashMap 类型，Key 是 TopicPartition 类型，表示一个主题分区，而 Value 是 CommitRecordMetadataAndOffset 类型。该类封装了位移提交消息的位移值。

在详细阅读位移管理方法之前，我先解释下这里的“位移”和“位移提交消息”。

消费者组需要向 Coordinator 提交已消费消息的进度，在 Kafka 中，这个进度有个专门的术语，叫作提交位移。Kafka 使用它来定位消费者组要消费的下一条消息。那么，提交位移在 Coordinator 端是如何保存的呢？它实际上是保存在内部位移主题中。提交的方式

是，消费者组成员向内部主题写入符合特定格式的事件消息，这类消息就是所谓的位移提交消息（Commit Record）。关于位移提交消息的事件格式，我会在第 30 讲具体介绍，这里你可以暂时不用理会。而这里所说的 CommitRecordMetadataAndOffset 类，就是标识位移提交消息的地方。我们看下它的代码：

[复制代码](#)

```
1 case class CommitRecordMetadataAndOffset(appendedBatchOffset: Option[Long], of
2     def olderThan(that: CommitRecordMetadataAndOffset): Boolean = appendedBatchO
3 }
```

这个类的构造函数有两个参数。

appendedBatchOffset：保存的是位移主题消息自己的位移值；

offsetAndMetadata：保存的是位移提交消息中保存的消费者组的位移值。

添加位移值

在 GroupMetadata 中，有 3 个向 offsets 中添加订阅分区的已消费位移值的方法，分别是 initializeOffsets、onOffsetCommitAppend 和 completePendingTxnOffsetCommit。

initializeOffsets 方法的代码非常简单，如下所示：

[复制代码](#)

```
1 def initializeOffsets(
2     offsets: collection.Map[TopicPartition, CommitRecordMetadataAndOffset],
3     pendingTxnOffsets: Map[Long, mutable.Map[TopicPartition, CommitRecordMetadat
4     this.offsets += offsets
5     this.pendingTransactionalOffsetCommits += pendingTxnOffsets
6 }
```

它仅仅是将给定的一组订阅分区提交位移值加到 offsets 中。当然，同时它还会更新 pendingTransactionalOffsetCommits 字段。

不过，由于这个字段是给 Kafka 事务机制使用的，因此，你只需要关注这个方法的第一行语句就行了。当消费者组的协调者组件启动时，它会创建一个异步任务，定期地读取位移

主题中相应消费者组的提交位移数据，并把它们加载到 `offsets` 字段中。

我们再来看第二个方法，`onOffsetCommitAppend` 的代码。

[复制代码](#)

```
1 def onOffsetCommitAppend(topicPartition: TopicPartition, offsetWithCommitRecordMetadata: OffsetWithCommitRecordMetadata) {
2   if (pendingOffsetCommits.contains(topicPartition)) {
3     if (offsetWithCommitRecordMetadata.appendedBatchOffset.isEmpty)
4       throw new IllegalStateException("Cannot complete offset commit write with no data" in the log.)
5     // offsets 字段中没有该分区位移提交数据，或者
6     // offsets 字段中该分区对应的提交位移消息在位移主题中的位移值小于待写入的位移值
7     if (!offsets.contains(topicPartition) || offsets(topicPartition).olderThan(offsetWithCommitRecordMetadata.offset))
8       // 将该分区对应的提交位移消息添加到 offsets 中
9       offsets.put(topicPartition, offsetWithCommitRecordMetadata)
10  }
11  pendingOffsetCommits.get(topicPartition) match {
12    case Some(stagedOffset) if offsetWithCommitRecordMetadata.offsetAndMetadata.compareTo(stagedOffset) > 0 =>
13      pendingOffsetCommits.remove(topicPartition)
14    case _ =>
15  }
16 }
17 }
```

该方法在提交位移消息被成功写入后调用。主要判断的依据，是 `offsets` 中是否已包含该主题分区对应的消息值，或者说，`offsets` 字段中该分区对应的提交位移消息在位移主题中的位移值是否小于待写入的位移值。如果是的话，就把该主题已提交的位移值添加到 `offsets` 中。

第三个方法 `completePendingTxnOffsetCommit` 的作用是完成一个待决事务（Pending Transaction）的位移提交。所谓的待决事务，就是指正在进行中、还没有完成的事务。在处理待决事务的过程中，可能会出现将待决事务中涉及到的分区的位移值添加到 `offsets` 中的情况。不过，由于该方法是与 Kafka 事务息息相关的，你不需要重点掌握，这里我就不展开说了。

移除位移值

`offsets` 中订阅分区的已消费位移值也是能够被移除的。你还记得，Kafka 主题中的消息有默认的留存时间设置吗？位移主题是普通的 Kafka 主题，所以也要遵守相应的规定。如果当前时间与已提交位移消息时间戳的差值，超过了 Broker 端参数

offsets.retention.minutes 值，Kafka 就会将这条记录从 offsets 字段中移除。这就是方法 removeExpiredOffsets 要做的事情。

这个方法的代码有点长，为了方便你掌握，我分块给你介绍下。我先带你了解下它的内部嵌套类方法 getExpireOffsets，然后再深入了解它的实现逻辑，这样你就能很轻松地掌握 Kafka 移除位移值的代码原理了。

首先，该方法定义了一个内部嵌套方法 **getExpiredOffsets**，专门用于获取订阅分区过期的位移值。我们来阅读下源码，看看它是如何做到的。

[复制代码](#)

```
1 def getExpiredOffsets(  
2   baseTimestamp: CommitRecordMetadataAndOffset => Long,  
3   subscribedTopics: Set[String] = Set.empty): Map[TopicPartition, OffsetAndMet  
4   // 遍历offsets中的所有分区，过滤出同时满足以下3个条件的所有分区  
5   // 条件1：分区所属主题不在订阅主题列表之内  
6   // 条件2：该主题分区已经完成位移提交  
7   // 条件3：该主题分区在位移主题中对应消息的存在时间超过了阈值  
8   offsets.filter {  
9     case (topicPartition, commitRecordMetadataAndOffset) =>  
10      !subscribedTopics.contains(topicPartition.topic()) &&  
11      !pendingOffsetCommits.contains(topicPartition) && {  
12        commitRecordMetadataAndOffset  
13          .offsetAndMetadata.expireTimestamp match {  
14            case None =>  
15              currentTimestamp - baseTimestamp(commitRecordMetadataAndOffset) >=  
16            case Some(expireTimestamp) =>  
17              currentTimestamp >= expireTimestamp  
18          }  
19        }  
20    }.map {  
21      // 为满足以上3个条件的分区提取出commitRecordMetadataAndOffset中的位移值  
22      case (topicPartition, commitRecordOffsetAndMetadata) =>  
23        (topicPartition, commitRecordOffsetAndMetadata.offsetAndMetadata)  
24    }.toMap  
25  }
```

该方法接收两个参数。

baseTimestamp：它是一个函数类型，接收 CommitRecordMetadataAndOffset 类型的字段，然后计算时间戳，并返回；

subscribedTopics：即订阅主题集合，默认是空。

方法开始时，代码从 offsets 字段中过滤出同时满足 3 个条件的所有分区。


条件 1：分区所属主题不在订阅主题列表之内。当方法传入了不为空的主题集合时，就说明该消费者组此时正在消费中，正在消费的主题是不能执行过期位移移除的。

条件 2：主题分区已经完成位移提交，那种处于提交中状态，也就是保存在 pendingOffsetCommits 字段中的分区，不予考虑。

条件 3：该主题分区在位移主题中对应消息的存在时间超过了阈值。老版本的 Kafka 消息直接指定了过期时间戳，因此，只需要判断当前时间是否越过了这个过期时间。但是，目前，新版 Kafka 判断过期与否，主要是**基于消费者组状态**。如果是 Empty 状态，过期的判断依据就是当前时间与组变为 Empty 状态时间的差值，是否超过 Broker 端参数 offsets.retention.minutes 值；如果不是 Empty 状态，就看当前时间与提交位移消息中的时间戳差值是否超过了 offsets.retention.minutes 值。如果超过了，就视为已过期，对应的位移值需要被移除；如果没有超过，就不需要移除了。

当过滤出同时满足这 3 个条件的分区后，提取出它们对应的位移值对象并返回。

学过了 getExpiredOffsets 方法代码的实现之后，removeExpiredOffsets 方法剩下的代码就很容易理解了。

 复制代码

```
1 def removeExpiredOffsets(  
2   currentTimestamp: Long, offsetRetentionMs: Long): Map[TopicPartition, Offset  
3   // getExpiredOffsets方法代码.....  
4   // 调用getExpiredOffsets方法获取主题分区的过期位移  
5   val expiredOffsets: Map[TopicPartition, OffsetAndMetadata] = protocolType ma  
6     case Some(_) if is(Empty) =>  
7       getExpiredOffsets(  
8         commitRecordMetadataAndOffset => currentStateTimestamp .getOrElse(c  
9       )  
10    case Some(ConsumerProtocol.PROTOCOL_TYPE) if subscribedTopics.isDefined =>  
11      getExpiredOffsets(  
12        _.offsetAndMetadata.commitTimestamp,  
13        subscribedTopics.get  
14      )  
15    case None =>  
16      getExpiredOffsets(_.offsetAndMetadata.commitTimestamp)
```

```
17     case _ =>
18         Map()
19     }
20     if (expiredOffsets.nonEmpty)
21         debug(s"Expired offsets from group '$groupId': ${expiredOffsets.keySet}")
22     // 将过期位移对应的主题分区从offsets中移除
23     offsets --= expiredOffsets.keySet
24     // 返回主题分区对应的过期位移
25     expiredOffsets
26 }
```

代码根据消费者组的 `protocolType` 类型和组状态调用 `getExpiredOffsets` 方法，同时决定传入什么样的参数：

如果消费者组状态是 `Empty`，就传入组变更为 `Empty` 状态的时间，若该时间没有被记录，则使用提交位移消息本身的写入时间戳，来获取过期位移；

如果是普通的消费者组类型，且订阅主题信息已知，就传入提交位移消息本身的写入时间戳和订阅主题集合共同确定过期位移值；

如果 `protocolType` 为 `None`，就表示，这个消费者组其实是一个 `Standalone` 消费者，依然是传入提交位移消息本身的写入时间戳，来决定过期位移值；

如果消费者组的状态不符合刚刚说的这些情况，那就说明，没有过期位移值需要被移除。

当确定了要被移除的位移值集合后，代码会将它们从 `offsets` 中移除，然后返回这些被移除的位移值信息。至此，方法结束。

分区分配策略管理方法

最后，我们讨论下消费者组分区分配策略的管理，也就是字段 `supportedProtocols` 的管理。`supportedProtocols` 是分区分配策略的支持票数，这个票数在添加成员、移除成员时，会进行相应的更新。

消费者组每次 `Rebalance` 的时候，都要重新确认本次 `Rebalance` 结束之后，要使用哪个分区分配策略，因此，就需要特定的方法来对这些票数进行统计，把票数最多的那个策略作为新的策略。

GroupMetadata 类中定义了两个方法来做这件事情，分别是 candidateProtocols 和 selectProtocol 方法。

确认消费者组支持的分区分配策略集

首先来看 candidateProtocols 方法。它的作用是**找出组内所有成员都支持的分区分配策略集**。代码如下：

[复制代码](#)

```
1 private def candidateProtocols: Set[String] = {  
2     val numMembers = members.size // 获取组内成员数  
3     // 找出支持票数=总成员数的策略，返回它们的名称  
4     supportedProtocols.filter(_._2 == numMembers).map(_._1).toSet  
5 }
```

该方法首先会获取组内的总成员数，然后，找出 supportedProtocols 中那些支持票数等于总成员数的分配策略，并返回它们的名称。**支持票数等于总成员数的意思，等同于所有成员都支持该策略。**

选出消费者组的分区消费分配策略

接下来，我们看下 selectProtocol 方法，它的作用是**选出消费者组的分区消费分配策略**。

[复制代码](#)

```
1 def selectProtocol: String = {  
2     // 如果没有任何成员，自然无法确定选用哪个策略  
3     if (members.isEmpty)  
4         throw new IllegalStateException("Cannot select protocol for empty group")  
5     // 获取所有成员都支持的策略集合  
6     val candidates = candidateProtocols  
7     // 让每个成员投票，票数最多的那个策略当选  
8     val (protocol, _) = allMemberMetadata  
9         .map(_._1.vote(candidates))  
10        .groupBy(identity)  
11        .maxBy { case (_, votes) => votes.size }  
12    protocol  
13 }
```

这个方法首先会判断组内是否有成员。如果没有任何成员，自然就无法确定选用哪个策略了，方法就会抛出异常，并退出。否则的话，代码会调用刚才的 candidateProtocols 方

法，获取所有成员都支持的策略集合，然后让每个成员投票，票数最多的那个策略当选。

你可能会好奇，这里的 vote 方法是怎么实现的呢？其实，它就是简单地查找而已。我举一个简单的例子，来帮助你理解。

比如，candidates 字段的值是[“策略 A” ， “策略 B”]，成员 1 支持[“策略 B” ， “策略 A”]，成员 2 支持[“策略 A” ， “策略 B” ， “策略 C”]，成员 3 支持[“策略 D” ， “策略 B” ， “策略 A”]，那么，vote 方法会将 candidates 与每个成员的支持列表进行比对，找出成员支持列表中第一个包含在 candidates 中的策略。因此，对于这个例子来说，成员 1 投票策略 B，成员 2 投票策略 A，成员 3 投票策略 B。可以看到，投票的结果是，策略 B 是两票，策略 A 是 1 票。所以，selectProtocol 方法返回策略 B 作为新的策略。

有一点你需要注意，**成员支持列表中的策略是有顺序的**。这就是说，[“策略 B” ， “策略 A”]和[“策略 A” ， “策略 B”]是不同的，成员会倾向于选择靠前的策略。

总结

今天，我们结合 GroupMetadata 源码，学习了 Kafka 对消费者组元数据的管理，主要包括组状态、成员、位移和分区分配策略四个维度。我建议你在课下再仔细地阅读一下这些管理数据的方法，对照着源码和注释走一遍完整的操作流程。

另外，在这两节课中，我没有谈及待决成员列表（Pending Members）和待决位移（Pending Offsets）的管理，因为这两个元数据项属于中间临时状态，因此我没有展开讲，不理解这部分代码的话，也不会影响我们理解消费者组元数据以及 Coordinator 是如何使用它们的。不过，我建议你可以阅读下与它们相关的代码部分。要知道，Kafka 是非常喜欢引用中间状态变量来管理各类元数据或状态的。

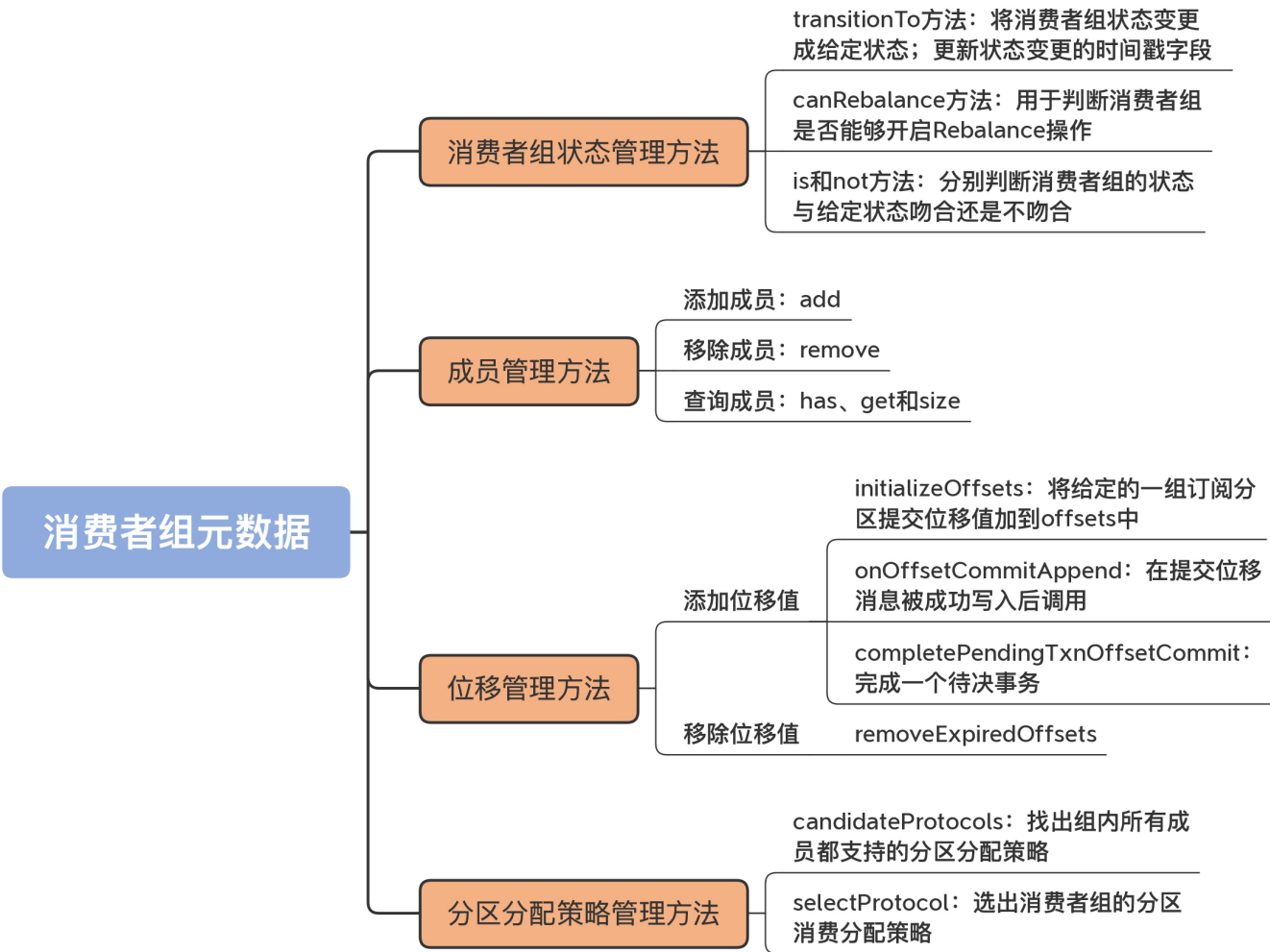
现在，我们再来简单回顾下这节课的重点。

消费者组元数据管理：主要包括对组状态、成员、位移和分区分配策略的管理。

组状态管理：transitionTo 方法负责设置状态，is、not 和 get 方法用于查询状态。

成员管理：add、remove 方法用于增减成员，has 和 get 方法用于查询特定成员。

分区分配策略管理：定义了专属方法 `selectProtocols`，用于在每轮 Rebalance 时选举分区分配策略。



至此，我们花了两节课的时间，详细地学习了消费者组元数据及其管理方法的源码。这些操作元数据的方法被上层调用方 `GroupCoordinator` 大量使用，就像我在开头提到的，如果现在我们不彻底掌握这些元数据被操作的手法，等我们学到 `GroupCoordinator` 代码时，就会感到有些吃力，所以，你一定要好好地学习这两节课。有了这些基础，等到学习 `GroupCoordinator` 源码时，你就能更加深刻地理解它的底层实现原理了。

课后讨论

在讲到 `MemberMetadata` 时，我说过，每个成员都有自己的 Rebalance 超时时间设置，那么，Kafka 是怎么确认消费者组使用哪个成员的超时时间作为整个组的超时时间呢？

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 27 | 消费者组元数据（上）：消费者组都有哪些元数据？

下一篇 29 | GroupMetadataManager：组元数据管理器是个什么东西？

精选留言 (2)

💬 写留言



胡夕 置顶

2020-07-07

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，我们重点学习了消费者组元数据都有哪些。课后我请你思考这样一个问题：kafka-consumer-groups脚本输出中的ASSIGNMENT-STRATEGY项对应于哪一项元数据。实

际上，它对应于GroupMetadata类的protocolName字段，即消费者组分区消费分配策...
展开



伯安知心
2020-07-02

每个消费者都有一个自己的rebalance时间，这是为了每个消费者按照自己的需要设置，但是在服务端消费者组需要管理这些组内所有消费者，消费者组也需要一个rebalance时间来平衡消费者，在服务端prepareRebalance方法有个参数delayedRebalance 要么初始化InitialDelayedJoin得到rebalance时间，要么非初始化方法DelayedJoin，而DelayedJoin中参数rebalanceTimeoutMs 从方法timeout.max(member.rebalanceTimeoutMs...
展开

作者回复:

