



下载APP



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

2020-07-09 胡夕

Kafka核心源码解读

[进入课程 >](#)**讲述：胡夕**

时长 15:31 大小 14.22M



你好，我是胡夕。

上节课，我们学习了位移主题中的两类消息：**消费者组注册消息**和**消费者组已提交位移消息**。今天，我们接着学习位移主题，重点是掌握写入位移主题和读取位移主题。

我们总说，位移主题是个神秘的主题，除了它并非我们亲自创建之外，它的神秘之处还体现在，它的读写也不由我们控制。默认情况下，我们没法向这个主题写入消息，而且直接读取该主题的消息时，看到的更是一堆乱码。因此，今天我们学习一下读写位移主题，正是去除它神秘感的重要一步。

### 写入位移主题

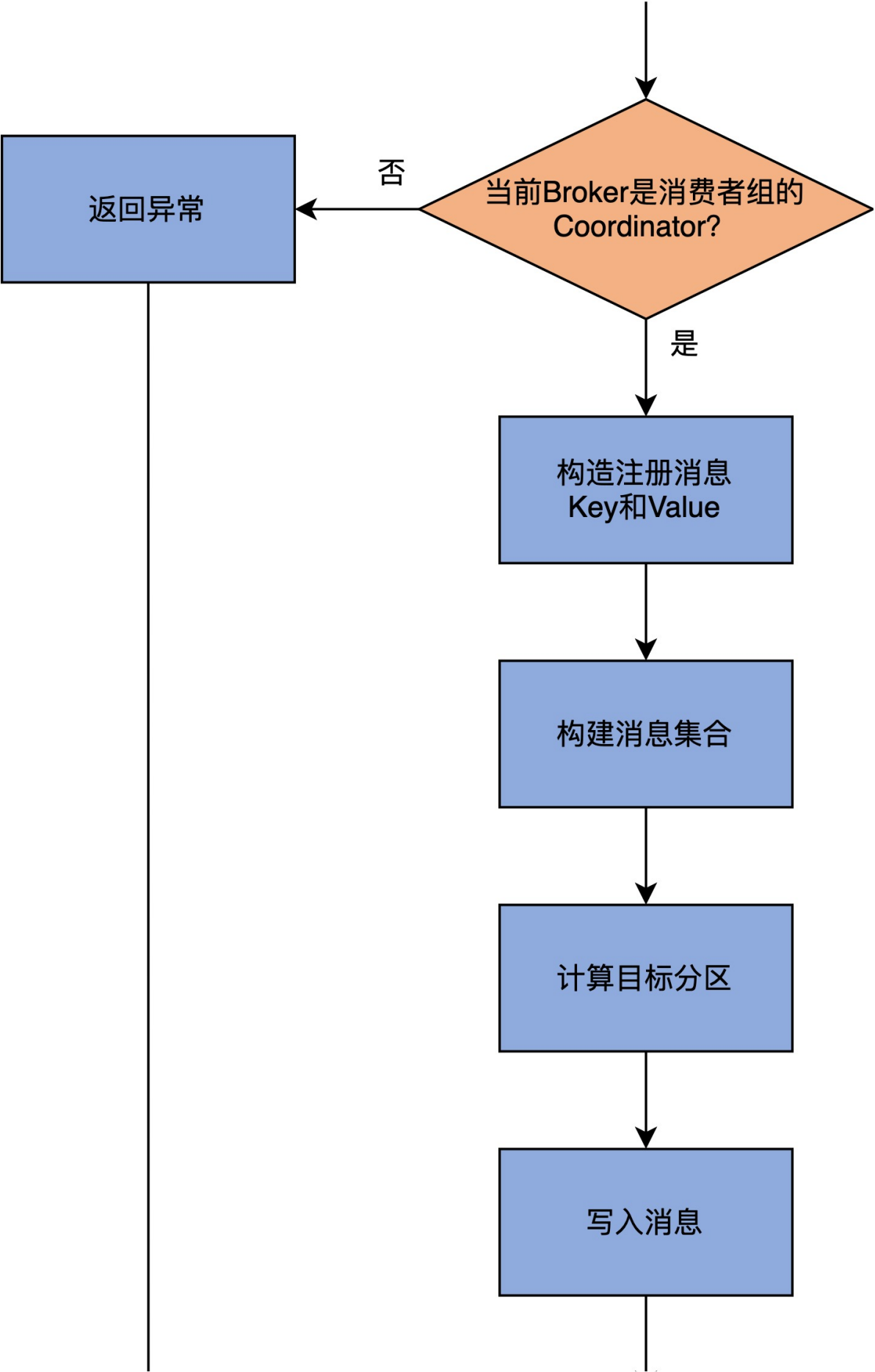
我们先来学习一下位移主题的写入。在 [第 29 讲](#) 学习 storeOffsets 方法时，我们已经学过了 appendForGroup 方法。Kafka 定义的两类消息类型都是由它写入的。在源码中，storeGroup 方法调用它写入消费者组注册消息，storeOffsets 方法调用它写入已提交位移消息。

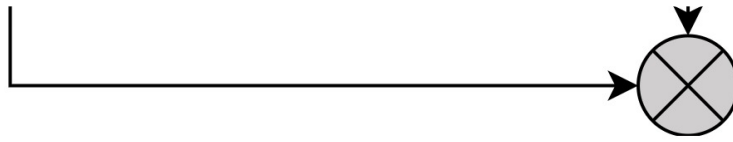
首先，我们需要知道 storeGroup 方法，它的作用是**向 Coordinator 注册消费者组**。我们看下它的代码实现：

[复制代码](#)

```
1 def storeGroup(group: GroupMetadata,
2               groupAssignment: Map[String, Array[Byte]],
3               responseCallback: Errors => Unit): Unit = {
4   // 判断当前Broker是否是该消费者组的Coordinator
5   getMagic(partitionFor(group.groupId)) match {
6     // 如果当前Broker不是Coordinator
7     case Some(magicValue) =>
8       val timestampType = TimestampType.CREATE_TIME
9       val timestamp = time.milliseconds()
10      // 构建注册消息的Key
11      val key = GroupMetadataManager.groupMetadataKey(group.groupId)
12      // 构建注册消息的Value
13      val value = GroupMetadataManager.groupMetadataValue(group, groupAssignme
14      // 使用Key和Value构建待写入消息集合
15      val records = {
16        val buffer = ByteBuffer.allocate(AbstractRecords.estimateSizeInBytes(m
17          Seq(new SimpleRecord(timestamp, key, value)).asJava))
18        val builder = MemoryRecords.builder(buffer, magicValue, compressionTyp
19        builder.append(timestamp, key, value)
20        builder.build()
21      }
22      // 计算要写入的目标分区
23      val groupMetadataPartition = new TopicPartition(Topic.GROUP_METADATA_TOP
24      val groupMetadataRecords = Map(groupMetadataPartition -> records)
25      val generationId = group.generationId
26      // putCacheCallback方法，填充Cache
27      .....
28      // 向位移主题写入消息
29      appendForGroup(group, groupMetadataRecords, putCacheCallback)
30      // 如果当前Broker不是Coordinator
31      case None =>
32        // 返回NOT_COORDINATOR异常
33        responseCallback(Errors.NOT_COORDINATOR)
34        None
35    }
36  }
```

为了方便你理解，我画一张图来展示一下 storeGroup 方法的逻辑。





storeGroup 方法的第 1 步是调用 getMagic 方法，来判断当前 Broker 是否是该消费者组的 Coordinator 组件。判断的依据，是尝试去获取位移主题目标分区的底层日志对象。如果能够获取到，就说明当前 Broker 是 Coordinator，程序进入到下一步；反之，则表明当前 Broker 不是 Coordinator，就构造一个 NOT\_COORDINATOR 异常返回。

第 2 步，调用我们上节课学习的 groupMetadataKey 和 groupMetadataValue 方法，去构造注册消息的 Key 和 Value 字段。

第 3 步，使用 Key 和 Value 构建待写入消息集合。这里的消息集合类是 MemoryRecords。

当前，建模 Kafka 消息集合的类有两个。

MemoryRecords：表示内存中的消息集合；

FileRecords：表示磁盘文件中的消息集合。

这两个类的源码不是我们学习的重点，你只需要知道它们的含义就行了。不过，我推荐你课下阅读一下它们的源码，它们在 clients 工程中，这可以进一步帮助你理解 Kafka 如何在内存和磁盘上保存消息。

第 4 步，调用 partitionFor 方法，计算要写入的位移主题目标分区。

第 5 步，调用 appendForGroup 方法，将待写入消息插入到位移主题的目标分区下。至此，方法返回。

需要提一下的是，在上面的代码中，我省略了 putCacheCallback 方法的源码，我们在第 29 讲已经详细地学习过它了。它的作用就是当消息被写入到位移主题后，填充 Cache。

可以看到，写入位移主题和写入其它的普通主题并无差别。Coordinator 会构造符合规定格式的消息数据，并把它们传给 storeOffsets 和 storeGroup 方法，由它们执行写入操

作。因此，我们可以认为，Coordinator 相当于位移主题的消息生产者。

## 读取位移主题

其实，除了生产者这个角色以外，Coordinator 还扮演了消费者的角色，也就是读取位移主题。跟写入相比，读取操作的逻辑更加复杂一些，不光体现在代码长度上，更体现在消息读取之后的处理上。

首先，我们要知道，什么时候需要读取位移主题。


你可能会觉得，当消费者组查询位移时，会读取该主题下的数据。其实不然。查询位移时，Coordinator 只会从 GroupMetadata 元数据缓存中查找对应的位移值，而不会读取位移主题。真正需要读取位移主题的时机，**是在当前 Broker 当选 Coordinator**，也就是 Broker 成为了位移主题某分区的 Leader 副本时。

一旦当前 Broker 当选为位移主题某分区的 Leader 副本，它就需要将它内存中的元数据缓存填充起来，因此需要读取位移主题。在代码中，这是由 **scheduleLoadGroupAndOffsets** 方法完成的。该方法会创建一个异步任务，来读取位移主题消息，并填充缓存。这个异步任务要执行的逻辑，就是 loadGroupsAndOffsets 方法。

如果你翻开 loadGroupsAndOffsets 方法的源码，就可以看到，它本质上是调用 doLoadGroupsAndOffsets 方法实现的位移主题读取。下面，我们就重点学习下这个方法。

这个方法的代码很长，为了让你能够更加清晰地理解它，我先带你了解下它的方法签名，然后再给你介绍具体的实现逻辑。

首先，我们来看它的方法签名以及内置的一个子方法 logEndOffset。

 复制代码

```
1 private def doLoadGroupsAndOffsets(topicPartition: TopicPartition, onGroupLoad
2     // 获取位移主题指定分区的LEO值
3     // 如果当前Broker不是该分区的Leader副本，则返回-1
4     def logEndOffset: Long = replicaManager.getLogEndOffset(topicPartition).getO
5     .....
6 }
```



doLoadGroupsAndOffsets 方法，顾名思义，它要做两件事：加载消费者组；加载消费者组的位移。再强调一遍，所谓的加载，就是指读取位移主题下的消息，并将这些信息填充到缓存中。

该方法接收两个参数，第一个参数 topicPartition 是位移主题目标分区；第二个参数 onGroupLoaded 是加载完成后要执行的逻辑，这个逻辑是在上层组件中指定的，我们不需要掌握它的实现，这不会影响我们学习位移主题的读取。

doLoadGroupsAndOffsets 还定义了一个内置子方法 logEndOffset。它的目的很简单，就是**获取位移主题指定分区的 LEO 值，如果当前 Broker 不是该分区的 Leader 副本，就返回 -1。**

这是一个特别重要的事实，因为 Kafka 依靠它来判断分区的 Leader 副本是否发生变更。一旦发生变更，那么，在当前 Broker 执行 logEndOffset 方法的返回值，就是 -1，此时，Broker 就不再是 Leader 副本了。

doLoadGroupsAndOffsets 方法会**读取位移主题目标分区的日志对象**，并执行核心的逻辑动作，代码如下：

[复制代码](#)

```
1 .....
2 replicaManager.getLog(topicPartition) match {
3   // 如果无法获取到日志对象
4   case None =>
5     warn(s"Attempted to load offsets and group metadata from $topicPartition,
6   case Some(log) =>
7     // 核心逻辑.....
```

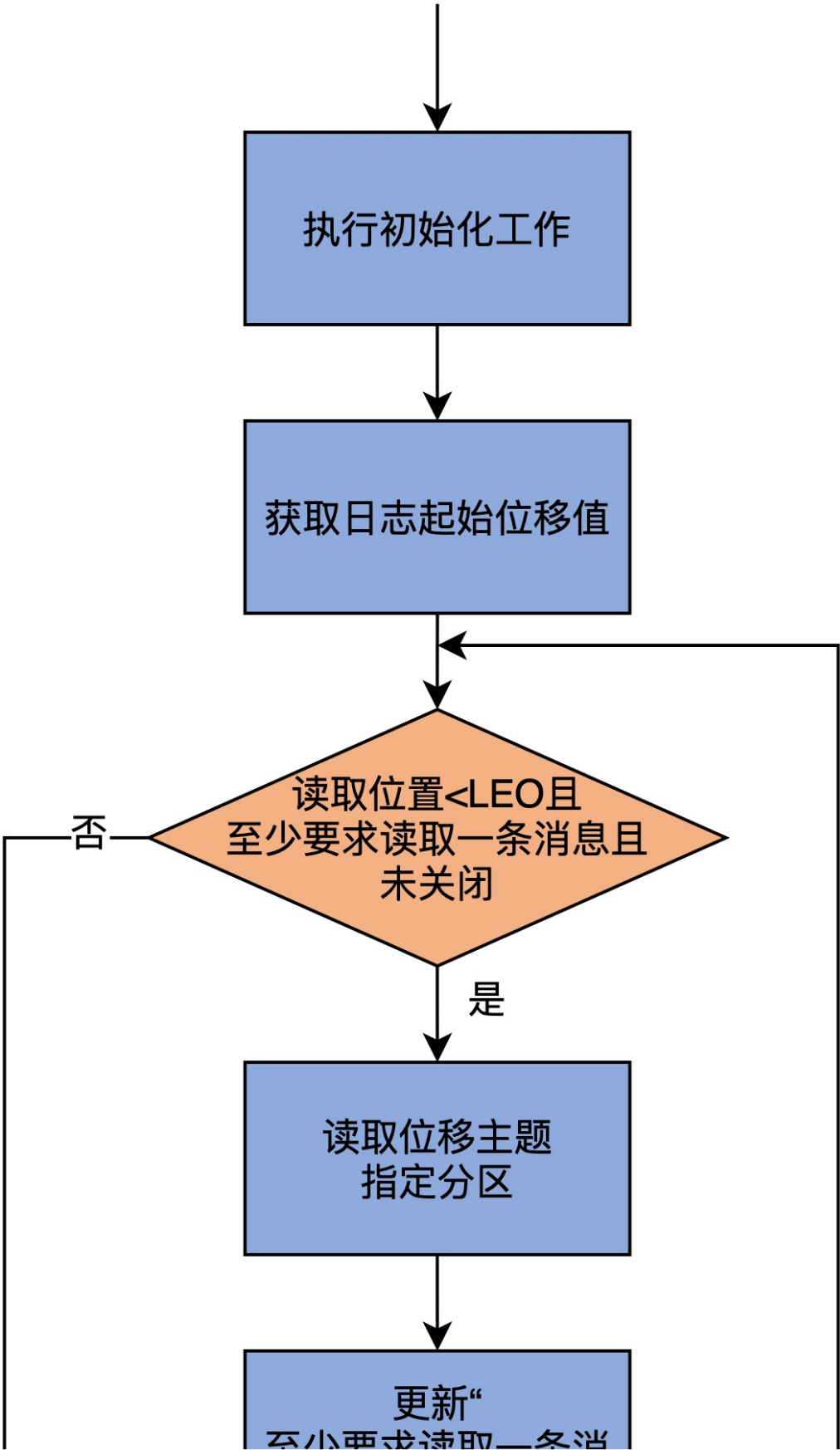
我把核心的逻辑分成 3 个部分来介绍。

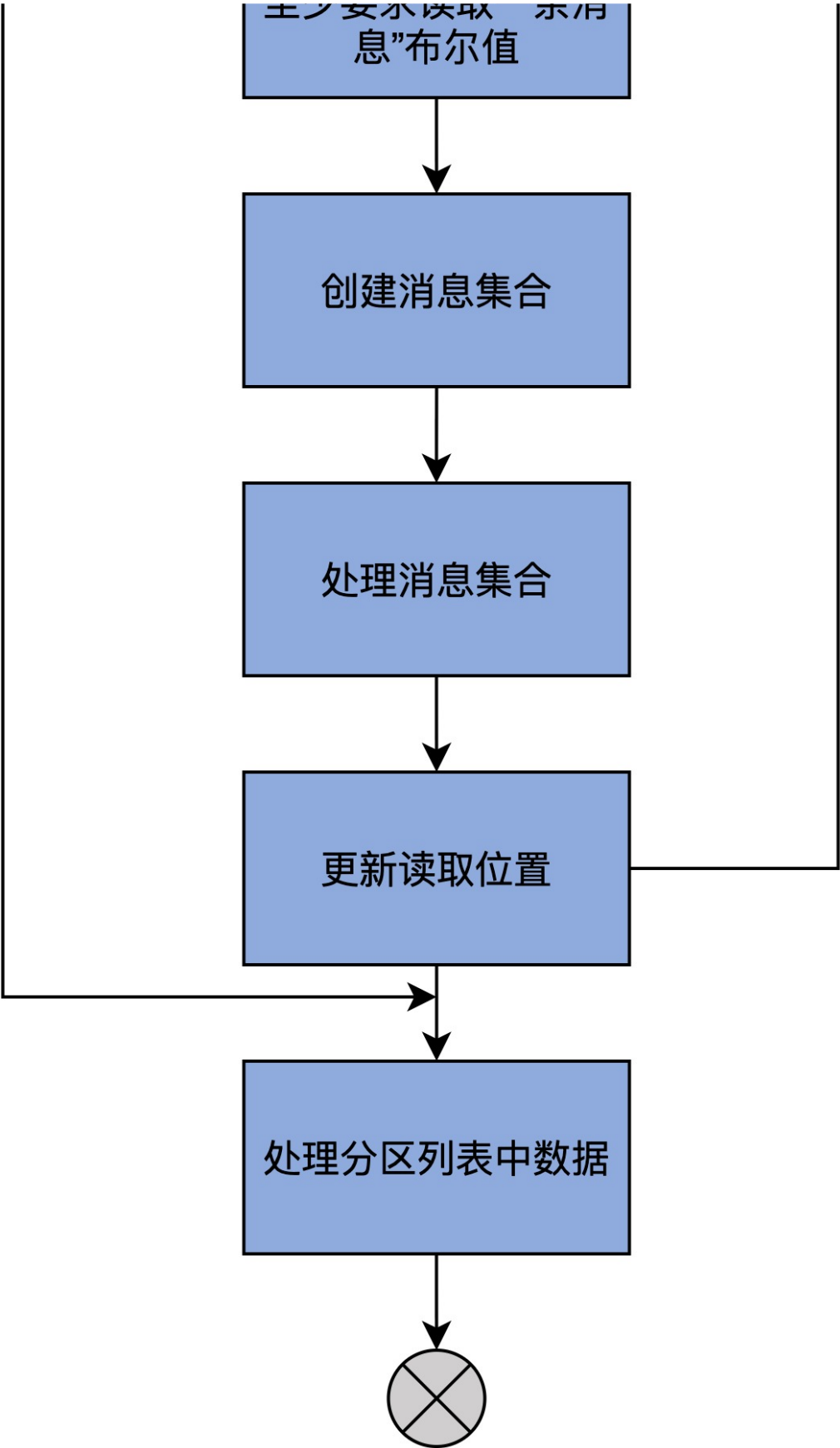
第 1 部分：初始化 4 个列表 + 读取位移主题；

第 2 部分：处理读到的数据，并填充 4 个列表；

第 3 部分：分别处理这 4 个列表。

在具体讲解这个方法所做的事情之前，我先画一张流程图，从宏观层面展示一下这个流程。






第 1 部分

首先，我们来学习一下第一部分的代码，完成了对位移主题的读取操作。



 复制代码

```

1 // 已完成位移值加载的分区列表
2 val loadedOffsets = mutable.Map[GroupTopicPartition, CommitRecordMetadataAndOf
3 // 处于位移加载中的分区列表，只用于Kafka事务
4 val pendingOffsets = mutable.Map[Long, mutable.Map[GroupTopicPartition, Commit
5 // 已完成组信息加载的消费者组列表
6 val loadedGroups = mutable.Map[String, GroupMetadata]()
7 // 待移除的消费者组列表
8 val removedGroups = mutable.Set[String]()
9 // 保存消息集合的ByteBuffer缓冲区
10 var buffer = ByteBuffer.allocate(0)
11 // 位移主题目标分区日志起始位移值
12 var currOffset = log.logStartOffset
13 // 至少要求读取一条消息
14 var readAtLeastOneRecord = true
15 // 当前读取位移<LEO，且至少要求读取一条消息，且GroupMetadataManager未关闭
16 while (currOffset < log.endOffset && readAtLeastOneRecord && !shuttingDown.get(
17 // 读取位移主题指定分区
18 val fetchDataInfo = log.read(currOffset,
19     maxLength = config.loadBufferSize,
20     isolation = FetchLogEnd,
21     minOneMessage = true)
22 // 若无消息可读，则不再要求至少读取一条消息
23 readAtLeastOneRecord = fetchDataInfo.records.sizeInBytes > 0
24 // 创建消息集合
25 val memRecords = fetchDataInfo.records match {
26     case records: MemoryRecords => records
27     case fileRecords: FileRecords =>
28         val sizeInBytes = fileRecords.sizeInBytes
29         val bytesNeeded = Math.max(config.loadBufferSize, sizeInBytes)
30         if (buffer.capacity < bytesNeeded) {
31             if (config.loadBufferSize < bytesNeeded)
32                 warn(s"Loaded offsets and group metadata from $topicPartition with b
33                     s"configured offsets.load.buffer.size (${config.loadBufferSize} by
34                 buffer = ByteBuffer.allocate(bytesNeeded)
35             } else {
36                 buffer.clear()
37             }
38             fileRecords.readInto(buffer, 0)
39             MemoryRecords.readableRecords(buffer)
40         }
41         .....
42 }

```

**首先**，这部分代码创建了 4 个列表。

`loadedOffsets`：已完成位移值加载的分区列表；

pendingOffsets：位移值加载中的分区列表；

loadedGroups：已完成组信息加载的消费者组列表；

removedGroups：待移除的消费者组列表。

**之后**，代码又创建了一个 ByteBuffer 缓冲区，用于保存消息集合。**接下来**，计算位移主题目标分区的日志起始位移值，这是要读取的起始位置。**再之后**，代码定义了一个布尔类型的变量，该变量表示本次至少要读取一条消息。

这些初始化工作都做完之后，代码进入到 while 循环中。循环的条件有 3 个，而且需要同时满足：

读取位移值小于日志 LEO 值；

布尔变量值是 True；

GroupMetadataManager 未关闭。

只要满足这 3 个条件，代码就会一直执行 while 循环下的语句逻辑。整个 while 下的逻辑被分成了 3 个步骤，我们现在学习的第 1 部分代码，包含了前两步。最后一步在第 3 部分中实现，即处理上面的这 4 个列表。我们先看前两步。

第 1 步是**读取位移主题目标分区的日志对象**，从日志中取出真实的消息数据。读取日志这个操作，是使用我们在 [第 3 讲](#)中学过的 Log.read 方法完成的。当读取到完整的日志之后，doLoadGroupsAndOffsets 方法会查看返回的消息集合，如果一条消息都没有返回，则取消“至少要求读取一条消息”的限制，即把刚才的布尔变量值设置为 False。

第 2 步是根据上一步获取到的消息数据，创建保存在内存中的消息集合对象，也就是 MemoryRecords 对象。

由于 doLoadGroupsAndOffsets 方法要将读取的消息填充到缓存中，因此，这里必须做出 MemoryRecords 类型的消息集合。这就是第二路 case 分支要将 FileRecords 转换成 MemoryRecords 类型的原因。

至此，第 1 部分逻辑完成。这一部分的产物就是成功地从位移主题目标分区读取到消息，然后转换成 MemoryRecords 对象，等待后续处理。

## 第 2 部分

现在, 代码进入到第 2 部分: **处理消息集合**。

值得注意的是, 这部分代码依然在 while 循环下, 我们看下它是如何实现的:

[复制代码](#)

```
1 // 遍历消息集合的每个消息批次(RecordBatch)
2 memRecords.batches.forEach { batch =>
3     val isTxnOffsetCommit = batch.isTransactional
4     // 如果是控制类消息批次
5     // 控制类消息批次属于Kafka事务范畴, 这里不展开讲
6     if (batch.isControlBatch) {
7         .....
8     } else {
9         // 保存消息批次第一条消息的位移值
10        var batchBaseOffset: Option[Long] = None
11        // 遍历消息批次下的所有消息
12        for (record <- batch.asScala) {
13            // 确保消息必须有Key, 否则抛出异常
14            require(record.hasKey, "Group metadata/offset entry key should not be nu
15            // 记录消息批次第一条消息的位移值
16            if (batchBaseOffset.isEmpty)
17                batchBaseOffset = Some(record.offset)
18            // 读取消息Key
19            GroupMetadataManager.readMessageKey(record.key) match {
20                // 如果是OffsetKey, 说明是提交位移消息
21                case offsetKey: OffsetKey =>
22                    .....
23                    val groupTopicPartition = offsetKey.key
24                    // 如果该消息没有Value
25                    if (!record.hasValue) {
26                        if (isTxnOffsetCommit)
27                            pendingOffsets(batch.producerId)
28                                .remove(groupTopicPartition)
29                        else
30                            // 将目标分区从已完成位移值加载的分区列表中移除
31                            loadedOffsets.remove(groupTopicPartition)
32                    } else {
33                        val offsetAndMetadata = GroupMetadataManager.readOffsetMessageValu
34                        if (isTxnOffsetCommit)
35                            pendingOffsets(batch.producerId).put(groupTopicPartition, CommitR
36                        else
37                            // 将目标分区加入到已完成位移值加载的分区列表
38                            loadedOffsets.put(groupTopicPartition, CommitRecordMetadataAndOf
39                    }
40                // 如果是GroupMetadataKey, 说明是注册消息
41                case groupMetadataKey: GroupMetadataKey =>
42                    val groupId = groupMetadataKey.key
```

```
43     val groupMetadata = GroupMetadataManager.readGroupMessageValue(group
44     // 如果消息Value不为空
45     if (groupMetadata != null) {
46         // 将该消费者组从待移除消费者组列表中移除
47         removedGroups.remove(groupId)
48         // 将消费者组加入到已完成加载的消费组列表
49         loadedGroups.put(groupId, groupMetadata)
50     // 如果消息Value为空，说明是Tombstone消息
51     } else {
52         // 将该消费者组从已完成加载的组列表中移除
53         loadedGroups.remove(groupId)
54         // 将消费者组加入到待移除消费组列表
55         removedGroups.add(groupId)
56     }
57     // 如果是未知类型的Key，抛出异常
58     case unknownKey =>
59         throw new IllegalStateException(s"Unexpected message key $unknownKey")
60     }
61 }
62 }
63 // 更新读取位置到消息批次最后一条消息的位移值+1，等待下次while循环
64 currOffset = batch.nextOffset
65 }
```

这一部分的主要目的，是处理上一步获取到的消息集合，然后把相应数据添加到刚刚说到的 4 个列表中，具体逻辑是代码遍历消息集合的每个消息批次（Record Batch）。我来解释一下这个流程。

**首先**，判断该批次是否是控制类消息批次，如果是，就执行 Kafka 事务专属的一些逻辑。由于我们不讨论 Kafka 事务，因此，这里我就不详细展开了。如果不是，就进入到下一步。

**其次**，遍历该消息批次下的所有消息，并依次执行下面的步骤。

第 1 步，记录消息批次中第一条消息的位移值。

第 2 步，读取消息 Key，并判断 Key 的类型，判断的依据如下：

如果是提交位移消息，就判断消息有无 Value。如果没有，那么，方法将目标分区从已完成位移值加载的分区列表中移除；如果有，则将目标分区加入到已完成位移值加载的分区列表中。

如果是注册消息，依然是判断消息有无 Value。如果存在 Value，就把该消费者组从待移除消费者组列表中移除，并加入到已完成加载的消费组列表；如果不存在 Value，就说明，这是一条 Tombstone 消息，那么，代码把该消费者组从已完成加载的组列表中移除，并加入到待移除消费组列表。


如果是未知类型的 Key，就直接抛出异常。

最后，更新读取位置，等待下次 while 循环，这个位置就是整个消息批次中最后一条消息的位移值 +1。

至此，这部分代码宣告结束，它的主要产物就是被填充了的 4 个列表。那么，第 3 部分，就要开始处理这 4 个列表了。

### 第 3 部分

最后一部分的完整代码如下：

 复制代码

```
1 // 处理loadedOffsets
2 val (groupOffsets, emptyGroupOffsets) = loadedOffsets
3   .groupBy(_.group)
4   .map { case (k, v) =>
5       // 提取出<组名, 主题名, 分区号>与位移值对
6       k -> v.map { case (groupTopicPartition, offset) => (groupTopicPartition.to
7   }.partition { case (group, _) => loadedGroups.contains(group) }
8   .....
9 // 处理loadedGroups
10 loadedGroups.values.foreach { group =>
11     // 提取消费者组的已提交位移
12     val offsets = groupOffsets.getOrElse(group.groupId, Map.empty[TopicPartition
13     val pendingOffsets = pendingGroupOffsets.getOrElse(group.groupId, Map.empty[
14     debug(s"Loaded group metadata $group with offsets $offsets and pending offse
15     // 为已完成加载的组执行加载组操作
16     loadGroup(group, offsets, pendingOffsets)
17     // 为已完成加载的组执行加载组操作之后的逻辑
18     onGroupLoaded(group)
19 }
20 (emptyGroupOffsets.keySet ++ pendingEmptyGroupOffsets.keySet).foreach { groupI
21     val group = new GroupMetadata(groupId, Empty, time)
22     val offsets = emptyGroupOffsets.getOrElse(groupId, Map.empty[TopicPartition,
23     val pendingOffsets = pendingEmptyGroupOffsets.getOrElse(groupId, Map.empty[L
24     debug(s"Loaded group metadata $group with offsets $offsets and pending offse
25     // 为空的消费者组执行加载组操作
26     loadGroup(group, offsets, pendingOffsets)
27
```

```
28 // 为空的消费者执行加载组操作之后的逻辑
29 onGroupLoaded(group)
30 }
31 // 处理removedGroups
32 removedGroups.foreach { groupId =>
33     if (groupMetadataCache.contains(groupId) && !emptyGroupOffsets.contains(group
34         throw new IllegalStateException(s"Unexpected unload of active group $group
35         s"loading partition $topicPartition")

```

**首先**，代码对 loadedOffsets 进行分组，将那些已经完成组加载的消费者组位移值分到一组，保存在字段 groupOffsets 中；将那些有位移值，但没有对应组信息的分成另外一组，也就是字段 emptyGroupOffsets 保存的数据。

**其次**，代码为 loadedGroups 中的所有消费者组执行加载组操作，以及加载之后的操作 onGroupLoaded。还记得吧，loadedGroups 中保存的都是已完成组加载的消费者组。这里的 onGroupLoaded 是上层调用组件 Coordinator 传入的。它主要的作用是处理消费者组下所有成员的心跳超时设置，并指定下一次心跳的超时时间。

**再次**，代码为 emptyGroupOffsets 的所有消费者组，创建空的消费者组元数据，然后执行和上一步相同的组加载逻辑以及加载后的逻辑。

**最后**，代码检查 removedGroups 中的所有消费者组，确保它们不能出现在消费者组元数据缓存中，否则将抛出异常。

至此，doLoadGroupsAndOffsets 方法的逻辑全部完成。经过调用该方法后，Coordinator 成功地读取了位移主题目标分区下的数据，并把它们填充到了消费者组元数据缓存中。

## 总结

今天，我们重点学习了 GroupMetadataManager 类中读写位移主题的方法代码。Coordinator 会使用这些方法对位移主题进行操作，实现对消费者组的管理。写入操作比较简单，它和一般的消息写入并无太大区别，而读取操作相对复杂一些。更重要的是，和我们的直观理解可能相悖的是，Kafka 在查询消费者组已提交位移时，是不会读取位移主题的，而是直接从内存中的消费者组元数据缓存中查询。这一点你一定要重点关注。

我们来简单回顾一下这节课的重点。



读写方法：appendForGroup 方法负责写入位移主题，doLoadGroupsAndOffsets 负责读取位移主题，并加载组信息和位移值。

查询消费者组位移：查询位移时不读取位移主题，而是读取消费者组元数据缓存。



至此，GroupMetadataManager 类的重要源码，我们就学完了。作为一个有着将近 1000 行代码，而且集这么多功能于一身的大文件，这个类的代码绝对值得你多读几遍。

除了我们集中介绍的这些功能之外，GroupMetadataManager 类其实还是连接 GroupMetadata 和 Coordinator 的重要纽带，Coordinator 利用 GroupMetadataManager 类实现操作 GroupMetadata 的目的。

我刚开始学习这部分源码的时候，居然不清楚 GroupMetadata 和 GroupMetadataManager 的区别是什么。现在，经过这 3 节课的内容，相信你已经知道，GroupMetadata 建模的是元数据信息，而 GroupMetadataManager 类建模的是管理元数据的方法，也是管理内部位移主题的唯一组件。以后碰到任何有关位移主题的问题，你都可以直接到这个类中去寻找答案。

### 课后讨论

其实，除了读写位移主题之外，GroupMetadataManager 还提供了清除位移主题数据的方法。代码中的 cleanGroupMetadata 就是做这个事儿的。请你结合源码，分析一下 cleanGroupMetadata 方法的流程。

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 30 | GroupMetadataManager: 位移主题保存的只是位移吗?

下一篇 32 | GroupCoordinator: 在Rebalance中, Coordinator如何处理成员入组?

## 精选留言 (1)

💬 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, 我们重点学习了消费者组管理器GroupMetadataManager类中关于位移主题的管理。课后我请你思考下kafka-console-consumer脚本中输出字段的含义。对于已提交位移消息来说, 它的Key格式是offset\_commit::group=<groupId>,partition=<分区...

展开 ∨



