



下载APP



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

2020-07-04 胡夕

Kafka核心源码解读


[进入课程 >](#)**讲述：胡夕**

时长 20:13 大小 18.52M



你好，我是胡夕。今天，我们学习 GroupMetadataManager 类的源码。从名字上来看，它是组元数据管理器，但是，从它提供的功能来看，我更愿意将它称作消费者组管理器，因为它定义的方法，提供的都是添加消费者组、移除组、查询组这样组级别的基础功能。

不过，这个类的知名度不像 KafkaController、GroupCoordinator 那么高，你之前可能都没有听说过它。但是，它其实是非常重要的消费者组管理类。

GroupMetadataManager 类是在消费者组 Coordinator 组件被创建时被实例化的。是说，每个 Broker 在启动过程中，都会创建并维持一个 GroupMetadataManager 实例，以实现对该 Broker 负责的消费者组进行管理。更重要的是，生产环境输出日志中的与消费者组相关的大多数信息，都和它息息相关。

我举一个简单的例子。你应该见过这样的日志输出：

[复制代码](#)

```
1 Removed xxx expired offsets in xxx milliseconds.
```

这条日志每 10 分钟打印一次。你有没有想过，它为什么要这么操作呢？其实，这是由 GroupMetadataManager 类创建的定时任务引发的。如果你不清楚 GroupMetadataManager 的原理，虽然暂时不会影响你使用，但是，一旦你在实际环境中看到了有关消费者组的错误日志，仅凭日志输出，你是无法定位错误原因的。要解决这个问题，就只有一个办法：**通过阅读源码，彻底搞懂底层实现原理，做到以不变应万变。**

关于这个类，最重要的就是要掌握它是如何管理消费者组的，以及它对内部位移主题的操作方法。这两个都是重磅功能，我们必须要吃透它们的原理，这也是我们这三节课的学习重点。今天，我们先学习它的类定义和管理消费者组的方法。

类定义与字段

GroupMetadataManager 类定义在 coordinator.group 包下的同名 scala 文件中。这个类的代码将近 1000 行，逐行分析的话，显然效率不高，也没有必要。所以，我从类定义和字段、重要方法两个维度给出主要逻辑的代码分析。下面的代码是该类的定义，以及我选取的重要字段信息。

[复制代码](#)

```
1 // brokerId : 所在Broker的Id
2 // interBrokerProtocolVersion : Broker端参数inter.broker.protocol.version值
3 // config: 内部位移主题配置类
4 // replicaManager: 副本管理器类
5 // zkClient: ZooKeeper客户端
6 class GroupMetadataManager(
7   brokerId: Int,
8   interBrokerProtocolVersion: ApiVersion,
9   config: OffsetConfig,
10  replicaManager: ReplicaManager,
11  zkClient: KafkaZkClient,
12  time: Time,
13  metrics: Metrics) extends Logging with KafkaMetricsGroup {
14  // 压缩器类型。向位移主题写入消息时执行压缩操作
15  private val compressionType: CompressionType = CompressionType.forId(config.
16  // 消费者组元数据容器，保存Broker管理的所有消费者组的数据
17  private val groupMetadataCache = new Pool[String, GroupMetadata]
```

```
18 // 位移主题下正在执行加载操作的分区
19 private val loadingPartitions: mutable.Set[Int] = mutable.Set()
20 // 位移主题下完成加载操作的分区
21 private val ownedPartitions: mutable.Set[Int] = mutable.Set()
22 // 位移主题总分区数
23 private val groupMetadataTopicPartitionCount = getGroupMetadataTopicPartitio
24 .....
25 `
```

这个类的构造函数需要 7 个参数，后面的 time 和 metrics 只是起辅助作用，因此，我重点解释一下前 5 个参数的含义。

brokerId：这个参数我们已经无比熟悉了。它是所在 Broker 的 ID 值，也就是 broker.id 参数值。

interBrokerProtocolVersion：保存 Broker 间通讯使用的请求版本。它是 Broker 端参数 inter.broker.protocol.version 值。这个参数的主要用途是**确定位移主题消息格式的版本**。

config：这是一个 OffsetConfig 类型。该类型定义了与位移管理相关的重要参数，比如位移主题日志段大小设置、位移主题备份因子、位移主题分区数配置等。

replicaManager：副本管理器类。GroupMetadataManager 类使用该字段实现获取分区对象、日志对象以及写入分区消息的目的。

zkClient：ZooKeeper 客户端。该类中的此字段只有一个目的：从 ZooKeeper 中获取位移主题的分区数。

除了构造函数所需的字段，该类还定义了其他关键字段，我给你介绍几个非常重要的。

1.compressionType

压缩器类型。Kafka 向位移主题写入消息前，可以选择对消息执行压缩操作。是否压缩，取决于 Broker 端参数 offsets.topic.compression.codec 值，默认是不进行压缩。如果你的位移主题占用的磁盘空间比较多的话，可以考虑启用压缩，以节省资源。

2.groupMetadataCache

该字段是 GroupMetadataManager 类上最重要的属性，它保存这个 Broker 上 GroupCoordinator 组件管理的所有消费者组元数据。它的 Key 是消费者组名称，Value 是消费者组元数据，也就是 GroupMetadata。源码通过该字段实现对消费者组的添加、删除和遍历操作。

3.loadingPartitions

位移主题下正在执行加载操作的分区号集合。这里需要注意两点：首先，这些分区都是位移主题分区，也就是 `__consumer_offsets` 主题下的分区；其次，所谓的加载，是指读取位移主题消息数据，填充 GroupMetadataCache 字段的操作。

4.ownedPartitions

位移主题下完成加载操作的分区号集合。与 loadingPartitions 类似的是，该字段保存的分区也是位移主题下的分区。和 loadingPartitions 不同的是，它保存的分区都是**已经完成加载操作**的分区。

5.groupMetadataTopicPartitionCount

位移主题的分区数。它是 Broker 端参数 `offsets.topic.num.partitions` 的值，默认是 50 个分区。若要修改分区数，除了变更该参数值之外，你也可以手动创建位移主题，并指定不同的分区数。

在这些字段中，groupMetadataCache 是最重要的，GroupMetadataManager 类大量使用该字段实现对消费者组的管理。接下来，我们就重点学习一下该类是如何管理消费者组的。

重要方法


管理消费者组包含两个方面，对消费者组元数据的管理以及对消费者组位移的管理。组元数据和组位移都是 Coordinator 端重要的消费者组管理对象。

消费者组元数据管理

消费者组元数据管理分为查询获取组信息、添加组、移除组和加载组信息。从代码复杂度来讲，查询获取、移除和添加的逻辑相对简单，加载的过程稍微费事些。我们先说说查询获取。

查询获取消费者组元数据

GroupMetadataManager 类中查询及获取组数据的方法有很多。大多逻辑简单，你一看就能明白，比如下面的 getGroup 方法和 getOrCreateGroup 方法：

 复制代码

```
1 // getGroup方法：返回给定消费者组的元数据信息。
2 // 若该组信息不存在，返回None
3 def getGroup(groupId: String): Option[GroupMetadata] = {
4     Option(groupMetadataCache.get(groupId))
5 }
6 // getOrCreateGroup方法：返回给定消费者组的元数据信息。
7 // 若不存在，则视createIfNotExist参数值决定是否需要添加该消费者组
8 def getOrCreateGroup(groupId: String, createIfNotExist: Boolean): Option[
9     if (createIfNotExist)
10         // 若不存在且允许添加，则添加一个状态是Empty的消费者组元数据对象
11         Option(groupMetadataCache.getAndMaybePut(groupId, new GroupMetadata(groupId
12     else
13         Option(groupMetadataCache.get(groupId))
14 }
```

GroupMetadataManager 类的上层组件 GroupCoordinator 会大量使用这两个方法来获取给定消费者组的数据。这两个方法都会返回给定消费者组的元数据信息，但是它们之间是有区别的。

对于 getGroup 方法而言，如果该组信息不存在，就返回 None，而这通常表明，消费者组确实不存在，或者是，该组对应的 Coordinator 组件变更到其他 Broker 上了。

而对于 getOrCreateGroup 方法而言，若组信息不存在，就根据 createIfNotExist 参数值决定是否需要添加该消费者组。而且，getOrCreateGroup 方法是在消费者组第一个成员加入组时被调用的，用于把组创建出来。

在 GroupMetadataManager 类中，还有一些地方也散落着组查询获取的逻辑。不过它们与这两个方法中的代码大同小异，很容易理解，课下你可以自己阅读下。

移除消费者组元数据

接下来，我们看下如何移除消费者组信息。当 Broker 卸任某些消费者组的 Coordinator 角色时，它需要将这些消费者组从 groupMetadataCache 中全部移除掉，这就是 removeGroupsForPartition 方法要做的事情。我们看下它的源码：

[复制代码](#)

```
1 def removeGroupsForPartition(offsetsPartition: Int,
2                               onGroupUnloaded: GroupMetadata => Unit): Unit = {
3   // 位移主题分区
4   val topicPartition = new TopicPartition(Topic.GROUP_METADATA_TOPIC_NAME, off
5   info(s"Scheduling unloading of offsets and group metadata from $topicPartiti
6   // 创建异步任务，移除组信息和位移信息
7   scheduler.schedule(topicPartition.toString, () => removeGroupsAndOffsets)
8   // 内部方法，用于移除组信息和位移信息
9   def removeGroupsAndOffsets(): Unit = {
10    var numOffsetsRemoved = 0
11    var numGroupsRemoved = 0
12    inLock(partitionLock) {
13      // 移除ownedPartitions中特定位移主题分区记录
14      ownedPartitions.remove(offsetsPartition)
15      // 遍历所有消费者组信息
16      for (group <- groupMetadataCache.values) {
17        // 如果该组信息保存在特定位移主题分区中
18        if (partitionFor(group.groupId) == offsetsPartition) {
19          // 执行组卸载逻辑
20          onGroupUnloaded(group)
21          // 关键步骤！将组信息从groupMetadataCache中移除
22          groupMetadataCache.remove(group.groupId, group)
23          // 把消费者组从producer对应的组集合中移除
24          removeGroupFromAllProducers(group.groupId)
25          // 更新已移除组计数器
26          numGroupsRemoved += 1
27          // 更新已移除位移值计数器
28          numOffsetsRemoved += group.numOffsets
29        }
30      }
31    }
32    info(s"Finished unloading $topicPartition. Removed $numOffsetsRemoved cach
33    s"and $numGroupsRemoved cached groups.")
34  }
35 }
```

该方法的主要逻辑是，先定义一个内部方法 removeGroupsAndOffsets，然后创建一个异步任务，调用该方法来执行移除消费者组信息和位移信息。

那么，怎么判断要移除哪些消费者组呢？这里的依据就是**传入的位移主题分区**。每个消费者组及其位移的数据，都只会保存在位移主题的一个分区下。一旦给定了位移主题分区，那么，元数据保存在这个位移主题分区下的消费者组就要被移除掉。

`removeGroupsForPartition` 方法传入的 `offsetsPartition` 参数，表示 Leader 发生变更的位移主题分区，因此，这些分区保存的消费者组都要从该 Broker 上移除掉。

具体的执行逻辑是什么呢？我来解释一下。

首先，异步任务从 `ownedPartitions` 中移除给定位移主题分区。

其次，遍历消费者组元数据缓存中的所有消费者组对象，如果消费者组正是在给定位移主题分区下保存的，就依次执行下面的步骤。

第 1 步，调用 `onGroupUnloaded` 方法执行组卸载逻辑。这个方法的逻辑是上层组件 `GroupCoordinator` 传过来的。它主要做两件事情：将消费者组状态变更到 `Dead` 状态；封装异常表示 `Coordinator` 已发生变更，然后调用回调函数返回。

第 2 步，把消费者组信息从 `groupMetadataCache` 中移除。这一步非常关键，目的是彻底清除掉该组的“痕迹”。

第 3 步，把消费者组从 `producer` 对应的组集合中移除。这里的 `producer`，是给 Kafka 事务用的。

第 4 步，增加已移除组计数器。

第 5 步，更新已移除位移值计数器。

到这里，方法结束。

添加消费者组元数据

下面，我们学习添加消费者组的管理方法，即 `addGroup`。它特别简单，仅仅是调用 `putIfNotExists` 将给定组添加进 `groupMetadataCache` 中而已。代码如下：

 复制代码

```
1 def addGroup(group: GroupMetadata): GroupMetadata = {  
2     val currentGroup = groupMetadataCache.putIfNotExists(group.groupId, group)  
3     if (currentGroup != null) {  
4         currentGroup
```

```
5    } else {  
6        group  
7    }  
8 }
```

加载消费者组元数据

现在轮到相对复杂的加载消费者组了。GroupMetadataManager 类中定义了一个 loadGroup 方法执行对应的加载过程。

[复制代码](#)

```
1 private def loadGroup(  
2     group: GroupMetadata, offsets: Map[TopicPartition, CommitRecordMetadataAndOf  
3     pendingTransactionalOffsets: Map[Long, mutable.Map[TopicPartition, CommitRec  
4     trace(s"Initialized offsets $offsets for group ${group.groupId}")  
5     // 初始化消费者组的位移信息  
6     group.initializeOffsets(offsets, pendingTransactionalOffsets.toMap)  
7     // 调用addGroup方法添加消费者组  
8     val currentGroup = addGroup(group)  
9     if (group != currentGroup)  
10         debug(s"Attempt to load group ${group.groupId} from log with generation ${  
11             s"because there is already a cached group with generation ${currentGroup  
12 }
```

该方法的逻辑有两步。

第 1 步，通过 initializeOffsets 方法，将位移值添加到 offsets 字段标识的消费者组提交位移元数据中，实现加载消费者组订阅分区提交位移的目的。

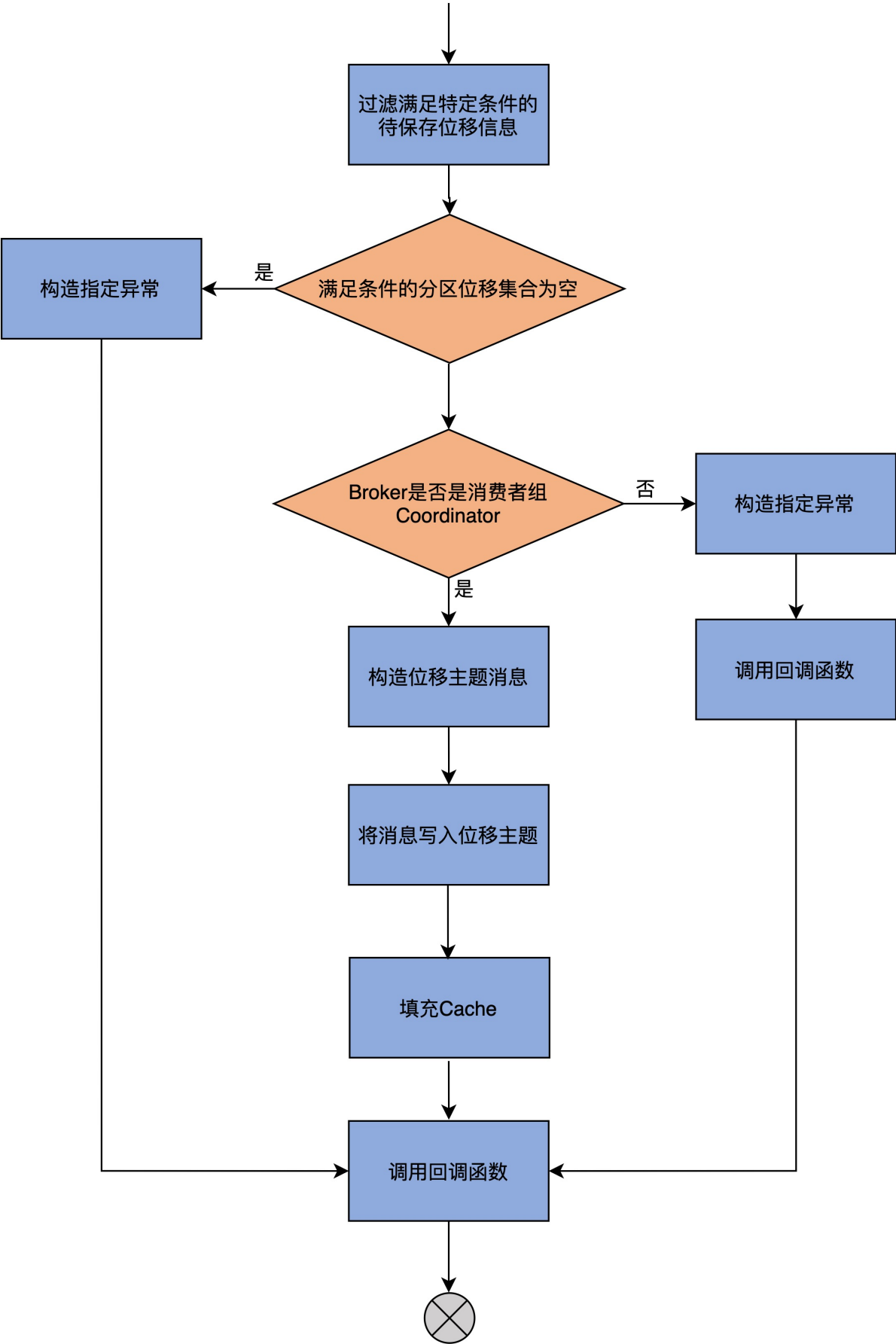
第 2 步，调用 addGroup 方法，将该消费者组元数据对象添加进消费者组元数据缓存，实现加载消费者组元数据的目的。

消费者组位移管理

除了消费者组的管理，GroupMetadataManager 类的另一大类功能，是提供消费者组位移的管理，主要包括位移数据的保存和查询。我们总说，位移主题是保存消费者组位移信息的地方。实际上，**当消费者组程序在查询位移时，Kafka 总是从内存中的位移缓存数据查询，而不会直接读取底层的位移主题数据。**

保存消费者组位移

storeOffsets 方法负责保存消费者组位移。该方法的代码很长，我先画一张图来展示下它的完整流程，帮助你建立起对这个方法的整体认知。接下来，我们再从它的方法签名和具体代码两个维度，来具体了解一下它的执行逻辑。



我先给你解释一下保存消费者组位移的全部流程。

首先，storeOffsets 方法要过滤出满足特定条件的待保存位移信息。是否满足特定条件，要看 validateOffsetMetadataLength 方法的返回值。这里的特定条件，是指位移提交记录中的自定义数据大小，要小于 Broker 端参数 offset.metadata.max.bytes 的值，默认值是 4KB。


如果没有一个分区满足条件，就构造 OFFSET_METADATA_TOO_LARGE 异常，并调用回调函数。这里的回调函数执行发送位移提交 Response 的动作。

倘若有分区满足了条件，**接下来**，方法会判断当前 Broker 是不是该消费者组的 Coordinator，如果不是的话，就构造 NOT_COORDINATOR 异常，并提交给回调函数；如果是的话，就构造位移主题消息，并将消息写入进位移主题下。

然后，调用一个名为 putCacheCallback 的内置方法，填充 groupMetadataCache 中各个消费者组元数据中的位移值，**最后**，调用回调函数返回。

接下来，我们结合代码来查看下 storeOffsets 方法的实现逻辑。

首先我们看下它的方法签名。既然是保存消费者组提交位移的，那么，我们就要知道上层调用方都给这个方法传入了哪些参数。

 复制代码

```
1 // group: 消费者组元数据
2 // consumerId: 消费者组成员ID
3 // offsetMetadata: 待保存的位移值，按照分区分组
4 // responseCallback: 处理完成后的回调函数
5 // producerId: 事务型Producer ID
6 // producerEpoch: 事务型Producer Epoch值
7 def storeOffsets(
8     group: GroupMetadata,
9     consumerId: String,
10    offsetMetadata: immutable.Map[TopicPartition, OffsetAndMetadata],
11    responseCallback: immutable.Map[TopicPartition, Errors] => Unit,
12    producerId: Long = RecordBatch.NO_PRODUCER_ID,
13    producerEpoch: Short = RecordBatch.NO_PRODUCER_EPOCH): Unit = {
14    .....
15 }
16
```

这个方法接收 6 个参数，它们的含义我都用注释的方式标注出来了。producerId 和 producerEpoch 这两个参数是与 Kafka 事务相关的，你简单了解下就行。我们要重点掌握前面 4 个参数的含义。

group：消费者组元数据信息。该字段的类型就是我们之前学到的 GroupMetadata 类。

consumerId：消费者组成员 ID，仅用于 DEBUG 调试。

offsetMetadata：待保存的位移值，按照分区分组。

responseCallback：位移保存完成后需要执行的回调函数。

接下来，我们看下 storeOffsets 的代码。为了便于你理解，我删除了与 Kafka 事务操作相关的部分。

[复制代码](#)

```
1 // 过滤出满足特定条件的待保存位移数据
2 val filteredOffsetMetadata = offsetMetadata.filter { case (_, offsetAndMetadat
3     validateOffsetMetadataLength(offsetAndMetadata.metadata)
4 }
5 .....
6 val isTxnOffsetCommit = producerId != RecordBatch.NO_PRODUCER_ID
7 // 如果没有任何分区的待保存位移满足特定条件
8 if (filteredOffsetMetadata.isEmpty) {
9     // 构造OFFSET_METADATA_TOO_LARGE异常并调用responseCallback返回
10    val commitStatus = offsetMetadata.map { case (k, _) => k -> Errors.OFFSET_ME
11    responseCallback(commitStatus)
12    None
13 } else {
14     // 查看当前Broker是否为给定消费者组的Coordinator
15     getMagic(partitionFor(group.groupId)) match {
16         // 如果是Coordinator
17         case Some(magicValue) =>
18             val timestampType = TimestampType.CREATE_TIME
19             val timestamp = time.milliseconds()
20             // 构造位移主题的位移提交消息
21             val records = filteredOffsetMetadata.map { case (topicPartition, offsetA
22                 val key = GroupMetadataManager.offsetCommitKey(group.groupId, topicPar
23                 val value = GroupMetadataManager.offsetCommitValue(offsetAndMetadata,
24                     new SimpleRecord(timestamp, key, value)
25             }
26             val offsetTopicPartition = new TopicPartition(Topic.GROUP_METADATA_TOPIC
27             // 为写入消息创建内存Buffer
28             val buffer = ByteBuffer.allocate(AbstractRecords.estimateSizeInBytes(mag
29             if (isTxnOffsetCommit && magicValue < RecordBatch.MAGIC_VALUE_V2)
```

```

30         throw Errors.UNSUPPORTED_FOR_MESSAGE_FORMAT.exception("Attempting to m
31         val builder = MemoryRecords.builder(buffer, magicValue, compressionType,
32         producerId, producerEpoch, 0, isTxnOffsetCommit, RecordBatch.NO_PARTIT
33         records.foreach(builder.append)
34         val entries = Map(offsetTopicPartition -> builder.build())
35         // putCacheCallback函数定义.....
36         if (isTxnOffsetCommit) {
37             .....
38         } else {
39             group.inLock {
40                 group.prepareOffsetCommit(offsetMetadata)
41             }
42         }
43         // 写入消息到位移主题,同时调用putCacheCallback方法更新消费者元数据
44         appendForGroup(group, entries, putCacheCallback)
45         // 如果是Coordinator
46         case None =>
47             // 构造NOT_COORDINATOR异常并提交给responseCallback方法
48             val commitStatus = offsetMetadata.map {
49                 case (topicPartition, _) =>
50                     (topicPartition, Errors.NOT_COORDINATOR)
51             }
52             responseCallback(commitStatus)
53             None
54         }
55

```

我为方法的关键步骤都标注了注释，具体流程前面我也介绍过了，应该很容易理解。不过，这里还需要注意两点，也就是 `appendForGroup` 和 `putCacheCallback` 方法。前者是向位移主题写入消息；后者是填充元数据缓存的。我们结合代码来学习下。

`appendForGroup` 方法负责写入消息到位移主题，同时传入 `putCacheCallback` 方法，更新消费者元数据。以下是它的代码：

[复制代码](#)

```

1 private def appendForGroup(
2     group: GroupMetadata,
3     records: Map[TopicPartition, MemoryRecords],
4     callback: Map[TopicPartition, PartitionResponse] => Unit): Unit = {
5     replicaManager.appendRecords(
6         timeout = config.offsetCommitTimeoutMs.toLong,
7         requiredAcks = config.offsetCommitRequiredAcks,
8         internalTopicsAllowed = true,
9         origin = AppendOrigin.Coordinator,
10        entriesPerPartition = records,
11        delayedProduceLock = Some(group.lock),
12        responseCallback = callback)

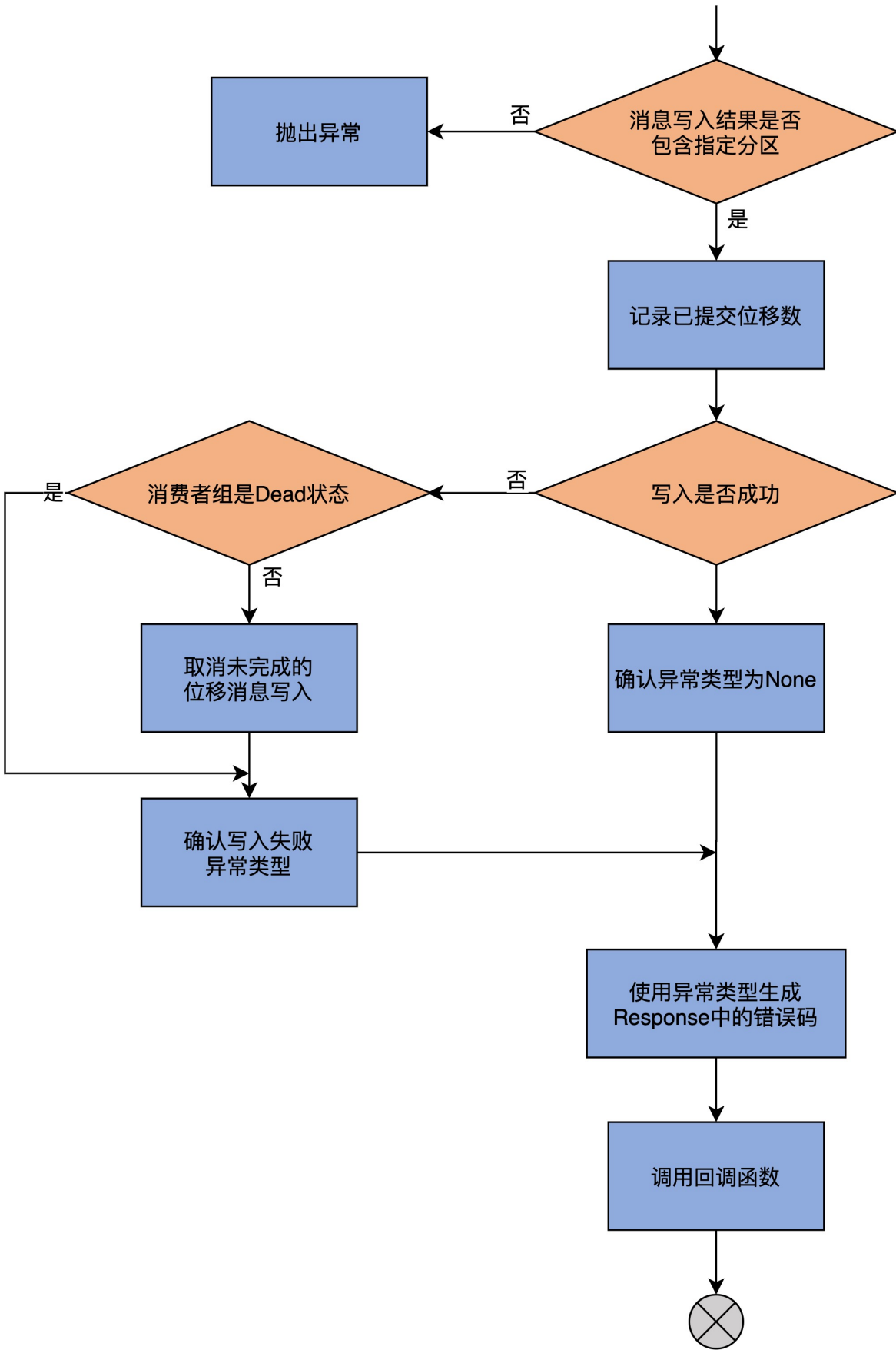
```




```
13 }
```

可以看到，该方法就是调用 `ReplicaManager` 的 `appendRecords` 方法，将消息写入到位移主题中。

下面，我们再关注一下 `putCacheCallback` 方法的实现，也就是将写入的位移值填充到缓存中。我先画一张图来展示下 `putCacheCallback` 的逻辑。



现在，我们结合代码，学习下它的逻辑实现。

 复制代码

```
1 def putCacheCallback(responseStatus: Map[TopicPartition, PartitionResponse]):
2   // 确保消息写入到指定位移主题分区, 否则抛出异常
3   if (responseStatus.size != 1 || !responseStatus.contains(offsetTopicPartitio
4     throw new IllegalStateException("Append status %s should only have one par
5     .format(responseStatus, offsetTopicPartition))
6   // 更新已提交位移数指标
7   offsetCommitsSensor.record(records.size)
8   val status = responseStatus(offsetTopicPartition)
9   val responseError = group.inLock {
10    // 写入结果没有错误
11    if (status.error == Errors.NONE) {
12      // 如果不是Dead状态
13      if (!group.is(Dead)) {
14        filteredOffsetMetadata.foreach { case (topicPartition, offsetAndMetada
15          if (isTxnOffsetCommit)
16            .....
17          else
18            // 调用GroupMetadata的onOffsetCommitAppend方法填充元数据
19            group.onOffsetCommitAppend(topicPartition, CommitRecordMetadataAnd
20        }
21      }
22      Errors.NONE
23    // 写入结果有错误
24    } else {
25      if (!group.is(Dead)) {
26        .....
27        filteredOffsetMetadata.foreach { case (topicPartition, offsetAndMetada
28          if (isTxnOffsetCommit)
29            group.failPendingTxnOffsetCommit(producerId, topicPartition)
30          else
31            // 取消未完成的位移消息写入
32            group.failPendingOffsetWrite(topicPartition, offsetAndMetadata)
33        }
34      }
35      .....
36    // 确认异常类型
37    status.error match {
38      case Errors.UNKNOWN_TOPIC_OR_PARTITION
39        | Errors.NOT_ENOUGH_REPLICAS
40        | Errors.NOT_ENOUGH_REPLICAS_AFTER_APPEND =>
41        Errors.COORDINATOR_NOT_AVAILABLE
42
43      case Errors.NOT_LEADER_FOR_PARTITION
44        | Errors.KAFKA_STORAGE_ERROR =>
45        Errors.NOT_COORDINATOR
46
47      case Errors.MESSAGE_TOO_LARGE
48        | Errors.RECORD_LIST_TOO_LARGE
49        | Errors.INVALID_FETCH_SIZE =>
50        Errors.INVALID_COMMIT_OFFSET_SIZE
```

```
51         case other => other
52     }
53 }
54 }
55 // 利用异常类型构建提交返回状态
56 val commitStatus = offsetMetadata.map { case (topicPartition, offsetAndMetad
57     if (validateOffsetMetadataLength(offsetAndMetadata.metadata))
58         (topicPartition, responseError)
59     else
60         (topicPartition, Errors.OFFSET_METADATA_TOO_LARGE)
61 }
62 // 调用回调函数
63 responseCallback(commitStatus)
64
```

putCacheCallback 方法的主要目的，是将多个消费者组位移值填充到 GroupMetadata 的 offsets 元数据缓存中。

首先，该方法要确保位移消息写入到指定位移主题分区，否则就抛出异常。

之后，更新已提交位移数指标，然后判断写入结果是否有错误。

如果没有错误，只要组状态不是 Dead 状态，就调用 GroupMetadata 的 onOffsetCommitAppend 方法填充元数据。onOffsetCommitAppend 方法的主体逻辑，是将消费者组订阅分区的位移值写入到 offsets 字段保存的集合中。当然，如果状态是 Dead，则什么都不做。

如果刚才的写入结果有错误，那么，就通过 failPendingOffsetWrite 方法取消未完成的位移消息写入。

接下来，代码要将日志写入的异常类型转换成表征提交状态错误的异常类型。具体来说，就是将 UNKNOWN_TOPIC_OR_PARTITION、NOT_LEADER_FOR_PARTITION 和 MESSAGE_TOO_LARGE 这样的异常，转换到 COORDINATOR_NOT_AVAILABLE 和 NOT_COORDINATOR 这样的异常。之后，再将这些转换后的异常封装进 commitStatus 字段中传给回调函数。

最后，调用回调函数返回。至此，方法结束。

好了，保存消费者组位移信息的 storeOffsets 方法，我们就学完了，它的关键逻辑，是构造位移主题消息并写入到位移主题，然后将位移值填充到消费者组元数据中。

查询消费者组位移

现在，我再说说查询消费者组位移，也就是 getOffsets 方法的代码实现。比起 storeOffsets，这个方法要更容易理解。我们看下它的源码：

[复制代码](#)

```
1 def getOffsets(  
2   groupId: String,  
3   requireStable: Boolean,  
4   topicPartitionsOpt: Option[Seq[TopicPartition]]): Map[TopicPartition, Partit  
5   .....  
6   // 从groupMetadataCache字段中获取指定消费者组的元数据  
7   val group = groupMetadataCache.get(groupId)  
8   // 如果没有组数据，返回空数据  
9   if (group == null) {  
10    topicPartitionsOpt.getOrElse(Seq.empty[TopicPartition]).map { topicPartiti  
11      val partitionData = new PartitionData(OffsetFetchResponse.INVALID_OFFSET  
12        Optional.empty(), "", Errors.NONE)  
13      topicPartition -> partitionData  
14    }.toMap  
15    // 如果存在组数据  
16  } else {  
17    group.inLock {  
18      // 如果组处于Dead状态，则返回空数据  
19      if (group.is(Dead)) {  
20        topicPartitionsOpt.getOrElse(Seq.empty[TopicPartition]).map { topicPar  
21          val partitionData = new PartitionData(OffsetFetchResponse.INVALID_OF  
22            Optional.empty(), "", Errors.NONE)  
23          topicPartition -> partitionData  
24        }.toMap  
25      } else {  
26        val topicPartitions = topicPartitionsOpt.getOrElse(group.allOffsets.ke  
27        topicPartitions.map { topicPartition =>  
28          if (requireStable && group.hasPendingOffsetCommitsForTopicPartition(  
29            topicPartition -> new PartitionData(OffsetFetchResponse.INVALID_OF  
30              Optional.empty(), "", Errors.UNSTABLE_OFFSET_COMMIT)  
31          } else {  
32            val partitionData = group.offset(topicPartition) match {  
33              // 如果没有该分区位移数据，返回空数据  
34              case None =>  
35                new PartitionData(OffsetFetchResponse.INVALID_OFFSET,  
36                  Optional.empty(), "", Errors.NONE)  
37              // 从消费者组元数据中返回指定分区的位移数据  
38              case Some(offsetAndMetadata) =>  
39                new PartitionData(offsetAndMetadata.offset,
```



```
40         offsetAndMetadata.leaderEpoch, offsetAndMetadata.metadata, E
41     }
42     topicPartition -> partitionData
43 }
44 }.toMap
45 }
46 }
47 }
48 }
```

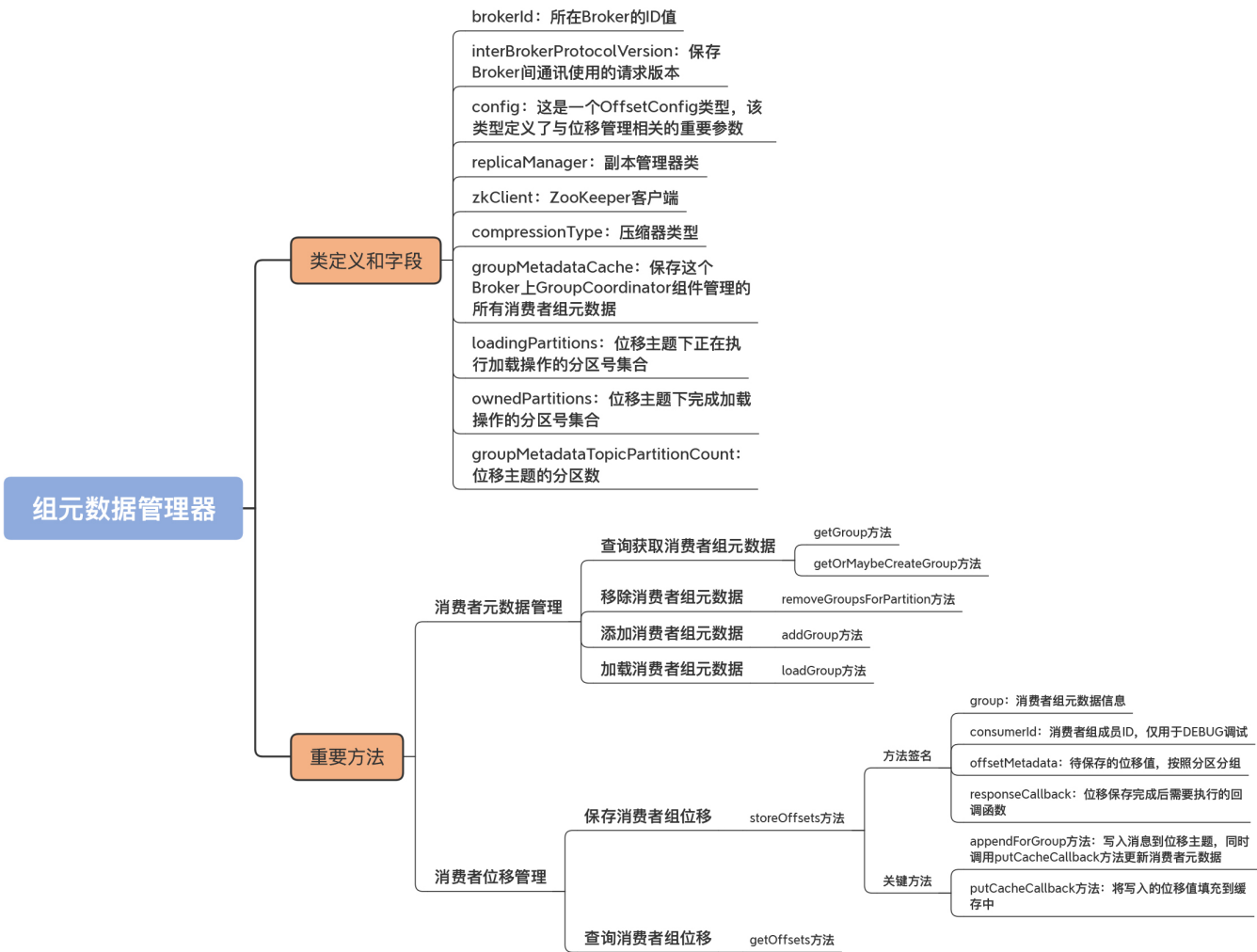
getOffsets 方法首先会读取 groupMetadataCache 中的组元数据，如果不存在对应的记录，则返回空数据集，如果存在，就接着判断组是否处于 Dead 状态。

如果是 Dead 状态，就说明消费者组已经被销毁了，位移数据也被视为不可用了，依然返回空数据集；若状态不是 Dead，就提取出消费者组订阅的分区信息，再依次为它们获取对应的位移数据并返回。至此，方法结束。

总结

今天，我们学习了 GroupMetadataManager 类的源码。作为消费者组管理器，它负责管理消费者组的方方面面。其中，非常重要的两个管理功能是消费者组元数据管理和消费者组位移管理，分别包括查询获取、移除、添加和加载消费者组元数据，以及保存和查询消费者组位移，这些方法是上层组件 GroupCoordinator 倚重的重量级功能载体，你一定要彻底掌握它们。

我画了一张思维导图，帮助你复习一下今天的重点内容。



下节课，我将带你继续研读 GroupMetadataManager 源码，去探寻有关位移主题的那些代码片段。

课后讨论

请思考这样一个问题：在什么场景下，需要移除 GroupMetadataManager 中保存的消费者组记录？

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

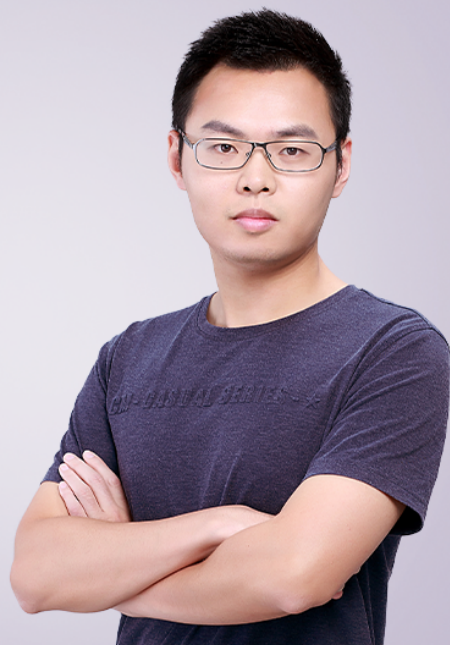
设计模式之美

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。

精选留言 (3)

[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Kafka如何管理消费者组元数据。课后我请你思考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。实际上，Kafka取消费者组下所有成员的最大超时时间作为整个组的超时时间。具体代码可以看下GroupMetadata...
[展开](#)

**伯安知心**

2020-07-06

从代码看，删除消费组的位移方法是removeGroupsForPartition，只有在处理身份或者位移变动调用handleGroupEmigration，请求这个方法的有2个调用。第一个方法handleLeaderAndIsrRequest判断就是新选择的broker重启太快导致leaderandisr请求变化，收到之前的请求也需要清理组的缓存记录，并且变动的topic是GROUP_METADATA_TOPIC_NAME的需要删除消费组位移记录，第二个方法handleStopReplicaRequest判断就...
[展开](#)

**刘飞**

2020-07-05

胡大您好！单个kafka集群broker数，topic数以及partition数是否有上限？broker数或者topic数以及partition太多会不会影响kafka的性能？这方面是否有最佳实践？

作者回复: 最佳实践是单台broker上分区数最好不超过2000

