# 11 | Controller元数据：Controller都保存有哪些东西？有几种状态？

2020-05-14 胡夕

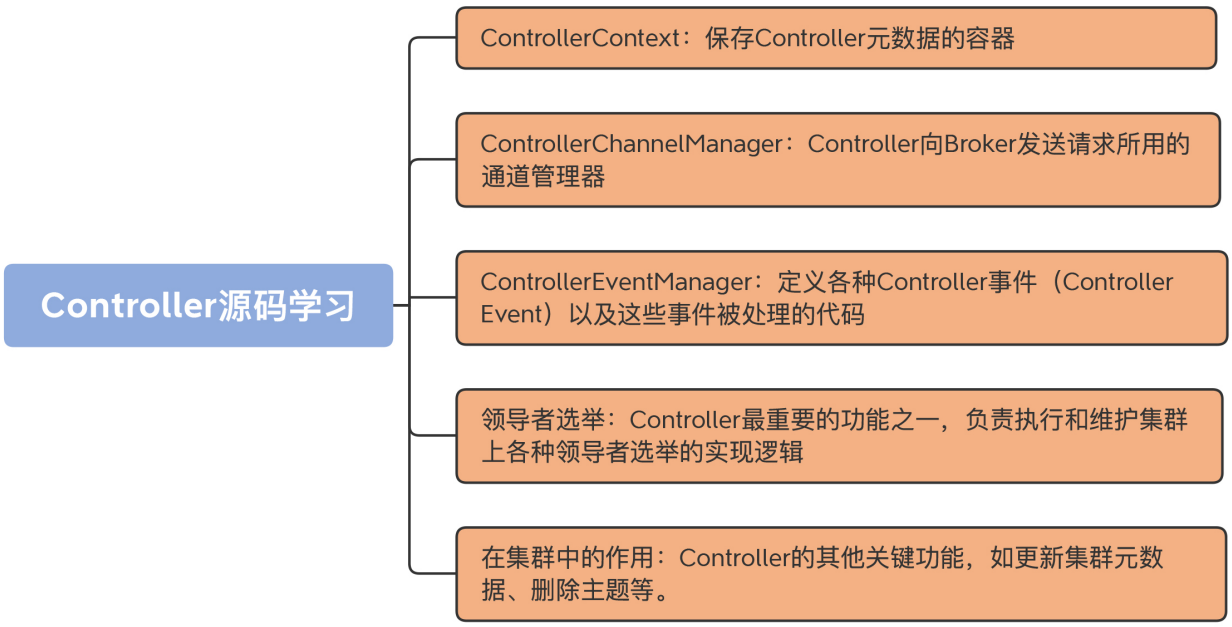



讲述：胡夕
时长 18:56 大小 17.35M

你好，我是胡夕。从今天开始，我们正式进入到第三大模块的学习：控制器（Controller）模块 。

提起 Kafka 中的 Controller 组件，我相信你一定不陌生。从某种意义上说，它是 Kafka 最核心的组件。一方面，它要为集群中的所有主题分区选举领导者副本；另一方面，它还承载着集群的全部元数据信息，并负责将这些元数据信息同步到其他 Broker 上。既然我们是 Kafka 源码解读课，那就绝对不能错过这么重量级的组件。

我画了一张图片，希望借助它帮你建立起对这个模块的整体认知。今天，我们先学习下 Controller 元数据。




极客时间

## 案例分享

在正式学习源码之前，我想向你分享一个真实的案例。

在我们公司的 Kafka 集群环境上，曾经出现了一个比较“诡异”的问题：某些核心业务的主题分区一直处于“不可用”状态。

通过使用“kafka-topics”命令查询，我们发现，这些分区的 Leader 显示是 -1。之前，这些 Leader 所在的 Broker 机器因为负载高宕机了，当 Broker 重启回来后，Controller 竟然无法成功地为这些分区选举 Leader，因此，它们一直处于“不可用”状态。

由于是生产环境，我们的当务之急是马上恢复受损分区，然后才能调研问题的原因。有人提出，重启这些分区旧 Leader 所在的所有 Broker 机器——这很容易想到，毕竟“重启大法”一直很好用。但是，这一次竟然没有任何作用。

之后，有人建议升级重启大法，即重启集群的所有 Broker——这在当时是不能接受的。且不说有很多业务依然在运行着，单是重启 Kafka 集群本身，就是一件非常缺乏计划性的事情。毕竟，生产环境怎么能随意重启呢？！

后来，我突然想到了 Controller 组件中重新选举 Controller 的代码。一旦 Controller 被选举出来，它就会向所有 Broker 更新集群元数据，也就是说，会“重刷”这些分区的状

态。

那么问题来了，我们如何在避免重启集群的情况下，干掉已有 Controller 并执行新的 Controller 选举呢？答案就在源码中的 **ControllerZNode.path** 上，也就是 ZooKeeper 的 /controller 节点。倘若我们手动删除了 /controller 节点，Kafka 集群就会触发 Controller 选举。于是，我们马上实施了这个方案，效果出奇得好：之前的受损分区全部恢复正常，业务数据得以正常生产和消费。

当然，给你分享这个案例的目的，并不是让你记住可以随意干掉 /controller 节点——这个操作其实是有一点危险的。事实上，我只是想通过这个真实的例子，向你说明，很多打开“精通 Kafka 之门”的钥匙是隐藏在源码中的。那么，接下来，我们就开始找“钥匙”吧。

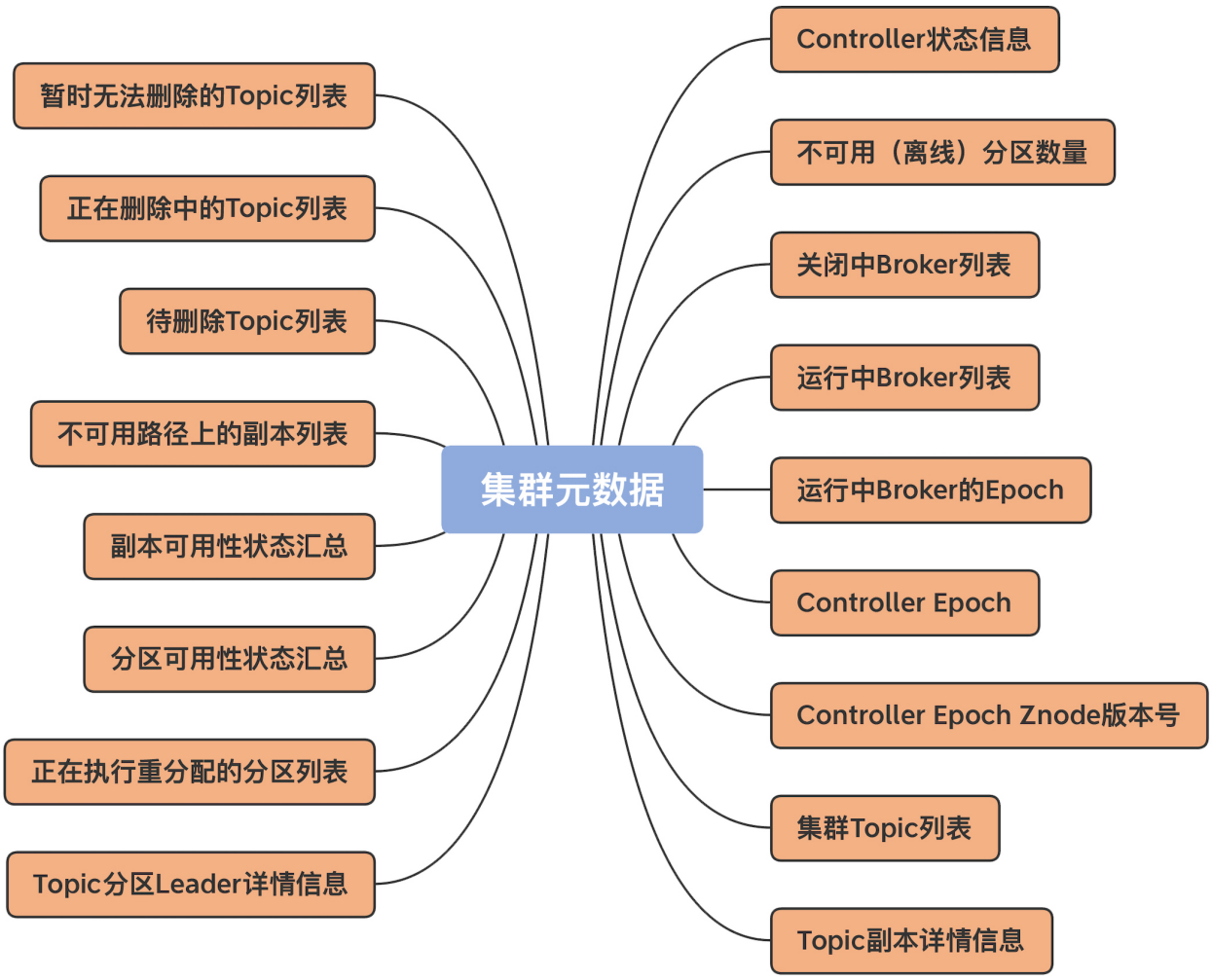
## 集群元数据

想要完整地了解 Controller 的工作原理，我们首先就要学习它管理了哪些数据。毕竟，Controller 的很多代码仅仅是做数据的管理操作而已。今天，我们就来重点学习 Kafka 集群元数据都有哪些。

如果说 ZooKeeper 是整个 Kafka 集群元数据的“真理之源（Source of Truth）”，那么 Controller 可以说是集群元数据的“真理之源副本（Backup Source of Truth）”。好吧，后面这个词是我自己发明的。你只需要理解，Controller 承载了 ZooKeeper 上的所有元数据即可。

事实上，集群 Broker 是不会与 ZooKeeper 直接交互去获取元数据的。相反地，它们总是与 Controller 进行通信，获取和更新最新的集群数据。而且社区已经打算把 ZooKeeper“干掉”了（我会在之后的“特别放送”里具体给你解释社区干掉 ZooKeeper 的操作），以后 Controller 将成为新的“真理之源”。

我们总说元数据，那么，到底什么是集群的元数据，或者说，Kafka 集群的元数据都定义了哪些内容呢？我用一张图给你完整地展示一下，当前 Kafka 定义的所有集群元数据信息。




极客时间

可以看到，目前，Controller 定义的元数据有 17 项之多。不过，并非所有的元数据都同等重要，你也不用完整地记住它们，我们只需要重点关注那些最重要的元数据，并结合源代码来了解下这些元数据都是用来做什么的。

在了解具体的元数据之前，我要先介绍下 ControllerContext 类。刚刚我们提到的这些元数据信息全部封装在这个类里。应该这么说，**这个类是 Controller 组件的数据容器类**。

## ControllerContext

Controller 组件的源代码位于 core 包的 src/main/scala/kafka/controller 路径下，这里面有很多 Scala 源文件，**ControllerContext 类就位于这个路径下的 ControllerContext.scala 文件中。**

该文件只有几百行代码，其中，最重要的数据结构就是 ControllerContext 类。前面说过，**它定义了前面提到的所有元数据信息，以及许多实用的工具方法**。比如，获取集群上所有主题分区对象的 allPartitions 方法、获取某主题分区副本列表的 partitionReplicaAssignment 方法，等等。

首先，我们来看下 ControllerContext 类的定义，如下所示：

复制代码

```
class ControllerContext {
  val stats = new ControllerStats // Controller统计信息类
  var offlinePartitionCount = 0    // 离线分区计数器
  val shuttingDownBrokerIds = mutable.Set.empty[Int]  // 关闭中Broker的Id列表
  private val liveBrokers = mutable.Set.empty[Broker] // 当前运行中Broker对象列表
  private val liveBrokerEpochs = mutable.Map.empty[Int, Long]   // 运行中Broker
  var epoch: Int = KafkaController.InitialControllerEpoch   // Controller当前Ep
  var epochZkVersion: Int = KafkaController.InitialControllerEpochZkVersion  /
  val allTopics = mutable.Set.empty[String]  // 集群主题列表
  val partitionAssignments = mutable.Map.empty[String, mutable.Map[Int, Replic
  val partitionLeadershipInfo = mutable.Map.empty[TopicPartition, LeaderIsrAnd
  val partitionsBeingReassigned = mutable.Set.empty[TopicPartition]  // 正处于副
  val partitionStates = mutable.Map.empty[TopicPartition, PartitionState] // 主
  val replicaStates = mutable.Map.empty[PartitionAndReplica, ReplicaState]  //
  val replicasOnOfflineDirs = mutable.Map.empty[Int, Set[TopicPartition]]  //
  val topicsToBeDeleted = mutable.Set.empty[String]  // 待删除主题列表
  val topicsWithDeletionStarted = mutable.Set.empty[String]  // 已开启删除的主题列
  val topicsIneligibleForDeletion = mutable.Set.empty[String]  // 暂时无法执行删除
  ......
}
```

不多不少，这段代码中定义的字段正好 17 个，它们一一对应着上图中的那些元数据信息。下面，我选取一些重要的元数据，来详细解释下它们的含义。

这些元数据理解起来还是比较简单的，掌握了它们之后，你在理解 MetadataCache，也就是元数据缓存的时候，就容易得多了。比如，接下来我要讲到的 liveBrokers 信息，就是 Controller 通过 UpdateMetadataRequest 请求同步给其他 Broker 的 MetadataCache 的。

## ControllerStats

第一个是 ControllerStats 类的变量。它的完整代码如下：

```
复制代码
1  private[controller] class ControllerStats extends KafkaMetricsGroup {
2    // 统计每秒发生的Unclean Leader选举次数
3    val uncleanLeaderElectionRate = newMeter("UncleanLeaderElectionsPerSec", "el
4    // Controller事件通用的统计速率指标的方法
5    val rateAndTimeMetrics: Map[ControllerState, KafkaTimer] = ControllerState.v
6      state.rateAndTimeMetricName.map { metricName =>
7        state -> new KafkaTimer(newTimer(metricName, TimeUnit.MILLISECONDS, Time
8      }
9    }.toMap
10 }
```

顾名思义，它表征的是 Controller 的一些统计信息。目前，源码中定义了两大类统计指标：**UncleanLeaderElectionsPerSec 和所有 Controller 事件状态的执行速率与时间。**

其中，**前者是计算 Controller 每秒执行的 Unclean Leader 选举数量，通常情况下，执行 Unclean Leader 选举可能造成数据丢失，一般不建议开启它。**一旦开启，你就需要时刻关注这个监控指标的值，确保 Unclean Leader 选举的速率维持在一个很低的水平，否则会出现很多数据丢失的情况。

**后者是统计所有 Controller 状态的速率和时间信息**，单位是毫秒。当前，Controller 定义了很多事件，比如，TopicDeletion 是执行主题删除的 Controller 事件、ControllerChange 是执行 Controller 重选举的事件。ControllerStats 的这个指标通过在每个事件名后拼接字符串 RateAndTimeMs 的方式，为每类 Controller 事件都创建了对应的速率监控指标。

由于 Controller 事件有很多种，对应的速率监控指标也有很多，有一些 Controller 事件是需要你额外关注的。

举个例子，IsrChangeNotification 事件是标志 ISR 列表变更的事件，如果这个事件经常出现，说明副本的 ISR 列表经常发生变化，而这通常被认为是非正常情况，因此，你最好关注下这个事件的速率监控指标。

## offlinePartitionCount

**该字段统计集群中所有离线或处于不可用状态的主题分区数量。**所谓的不可用状态，就是我最开始举的例子中“Leader=-1”的情况。

ControllerContext 中的 updatePartitionStateMetrics 方法根据**给定主题分区的当前状态和目标状态**，来判断该分区是否是离线状态的分区。如果是，则累加 offlinePartitionCount 字段的值，否则递减该值。方法代码如下：

```
// 更新offlinePartitionCount元数据
private def updatePartitionStateMetrics(
  partition: TopicPartition,
  currentState: PartitionState,
  targetState: PartitionState): Unit = {
  // 如果该主题当前并未处于删除中状态
  if (!isTopicDeletionInProgress(partition.topic)) {
    // targetState表示该分区要变更到的状态
    // 如果当前状态不是OfflinePartition，即离线状态并且目标状态是离线状态
    // 这个if语句判断是否要将该主题分区状态转换到离线状态
    if (currentState != OfflinePartition && targetState == OfflinePartition) {
      offlinePartitionCount = offlinePartitionCount + 1
    // 如果当前状态已经是离线状态，但targetState不是
    // 这个else if语句判断是否要将该主题分区状态转换到非离线状态
    } else if (currentState == OfflinePartition && targetState != OfflineParti
      offlinePartitionCount = offlinePartitionCount - 1
    }
  }
}
```

该方法首先要判断，此分区所属的主题当前是否处于删除操作的过程中。如果是的话，Kafka 就不能修改这个分区的状态，那么代码什么都不做，直接返回。否则，代码会判断该分区是否要转换到离线状态。如果 targetState 是 OfflinePartition，那么就将 offlinePartitionCount 值加 1，毕竟多了一个离线状态的分区。相反地，如果 currentState 是 offlinePartition，而 targetState 反而不是，那么就将 offlinePartitionCount 值减 1。

## shuttingDownBrokerIds

顾名思义，**该字段保存所有正在关闭中的 Broker ID 列表**。当 Controller 在管理集群 Broker 时，它要依靠这个字段来甄别 Broker 当前是否已关闭，因为处于关闭状态的 Broker 是不适合执行某些操作的，如分区重分配（Reassignment）以及主题删除等。

另外，Kafka 必须要为这些关闭中的 Broker 执行很多清扫工作，Controller 定义了一个 onBrokerFailure 方法，它就是用来做这个的。代码如下：

```
复制代码
1  private def onBrokerFailure(deadBrokers: Seq[Int]): Unit = {
2    info(s"Broker failure callback for ${deadBrokers.mkString(",")}")
3    // deadBrokers: 给定的一组已终止运行的Broker Id列表
4    // 更新Controller元数据信息，将给定Broker从元数据的replicasOnOfflineDirs中移除
5    deadBrokers.foreach(controllerContext.replicasOnOfflineDirs.remove)
6    // 找出这些Broker上的所有副本对象
7    val deadBrokersThatWereShuttingDown =
8      deadBrokers.filter(id => controllerContext.shuttingDownBrokerIds.remove(id
9    if (deadBrokersThatWereShuttingDown.nonEmpty)
10     info(s"Removed ${deadBrokersThatWereShuttingDown.mkString(",")} from list o
11   // 执行副本清扫工作
12   val allReplicasOnDeadBrokers = controllerContext.replicasOnBrokers(deadBroke
13   onReplicasBecomeOffline(allReplicasOnDeadBrokers)
14   // 取消这些Broker上注册的ZooKeeper监听器
15   unregisterBrokerModificationsHandler(deadBrokers)
16 }
```

该方法接收一组已终止运行的 Broker ID 列表，首先是更新 Controller 元数据信息，将给定 Broker 从元数据的 replicasOnOfflineDirs 和 shuttingDownBrokerIds 中移除，然后为这组 Broker 执行必要的副本清扫工作，也就是 onReplicasBecomeOffline 方法做的事情。

该方法主要依赖于分区状态机和副本状态机来完成对应的工作。在后面的课程中，我们会专门讨论副本状态机和分区状态机，这里你只要简单了解下它要做的事情就行了。后面等我们学完了这两个状态机之后，你可以再看下这个方法的具体实现原理。

这个方法的主要目的是把给定的副本标记成 Offline 状态，即不可用状态。具体分为以下这几个步骤：

1. 利用分区状态机将给定副本所在的分区标记为 Offline 状态；
2. 将集群上所有新分区和 Offline 分区状态变更为 Online 状态；
3. 将相应的副本对象状态变更为 Offline。

## liveBrokers

**该字段保存当前所有运行中的 Broker 对象**。每个 Broker 对象就是一个 <Id，EndPoint，机架信息 > 的三元组。ControllerContext 中定义了很多方法来管理该字段，如

addLiveBrokersAndEpochs、removeLiveBrokers 和 updateBrokerMetadata 等。我拿 updateBrokerMetadata 方法进行说明，以下是源码：

```
复制代码
1 def updateBrokerMetadata(oldMetadata: Broker, newMetadata: Broker): Unit = {
2     liveBrokers -= oldMetadata
3     liveBrokers += newMetadata
4   }
```

每当新增或移除已有 Broker 时，ZooKeeper 就会更新其保存的 Broker 数据，从而引发 Controller 修改元数据，也就是会调用 updateBrokerMetadata 方法来增减 Broker 列表中的对象。怎么样，超简单吧？！

## liveBrokerEpochs

**该字段保存所有运行中 Broker 的 Epoch 信息**。Kafka 使用 Epoch 数据防止 Zombie Broker，即一个非常老的 Broker 被选举成为 Controller。

另外，源码大多使用这个字段来获取所有运行中 Broker 的 ID 序号，如下面这个方法定义的那样：

```
复制代码
1 def liveBrokerIds: Set[Int] = liveBrokerEpochs.keySet -- shuttingDownBrokerIds
```

liveBrokerEpochs 的 keySet 方法返回 Broker 序号列表，然后从中移除关闭中的 Broker 序号，剩下的自然就是处于运行中的 Broker 序号列表了。

## epoch & epochZkVersion

这两个字段一起说，因为它们都有“epoch”字眼，放在一起说，可以帮助你更好地理解两者的区别。epoch 实际上就是 ZooKeeper 中 /controller_epoch 节点的值，你可以认为它就是 Controller 在整个 Kafka 集群的版本号，而 epochZkVersion 实际上是 /controller_epoch 节点的 dataVersion 值。

Kafka 使用 epochZkVersion 来判断和防止 Zombie Controller。这也就是说，原先在老 Controller 任期内的 Controller 操作在新 Controller 不能成功执行，因为新 Controller 的 epochZkVersion 要比老 Controller 的大。

另外，你可能会问：“这里的两个 Epoch 和上面的 liveBrokerEpochs 有啥区别呢？”实际上，这里的两个 Epoch 值都是属于 Controller 侧的数据，而 liveBrokerEpochs 是每个 Broker 自己的 Epoch 值。

## allTopics

**该字段保存集群上所有的主题名称。**每当有主题的增减，Controller 就要更新该字段的值。

比如 Controller 有个 processTopicChange 方法，从名字上来看，它就是处理主题变更的。我们来看下它的代码实现，我把主要逻辑以注释的方式标注了出来：

复制代码

```
private def processTopicChange(): Unit = {
    if (!isActive) return // 如果Contorller已经关闭，直接返回
    val topics = zkClient.getAllTopicsInCluster(true) // 从ZooKeeper中获取当前所有
    val newTopics = topics -- controllerContext.allTopics // 找出当前元数据中不存在
    val deletedTopics = controllerContext.allTopics -- topics // 找出当前元数据中
    controllerContext.allTopics = topics // 更新Controller元数据
    // 为新增主题和已删除主题执行后续处理操作
    registerPartitionModificationsHandlers(newTopics.toSeq)
    val addedPartitionReplicaAssignment = zkClient.getFullReplicaAssignmentFor
    deletedTopics.foreach(controllerContext.removeTopic)
    addedPartitionReplicaAssignment.foreach {
      case (topicAndPartition, newReplicaAssignment) => controllerContext.upda
    }
    info(s"New topics: [$newTopics], deleted topics: [$deletedTopics], new par
      s"[$addedPartitionReplicaAssignment]")
    if (addedPartitionReplicaAssignment.nonEmpty)
      onNewPartitionCreation(addedPartitionReplicaAssignment.keySet)
  }

```

## partitionAssignments

**该字段保存所有主题分区的副本分配情况。**在我看来，**这是 Controller 最重要的元数据了。**事实上，你可以从这个字段衍生、定义很多实用的方法，来帮助 Kafka 从各种维度获

取数据。

比如，如果 Kafka 要获取某个 Broker 上的所有分区，那么，它可以这样定义：

```
复制代码
1 partitionAssignments.flatMap {
2       case (topic, topicReplicaAssignment) => topicReplicaAssignment.filter {
3         case (_, partitionAssignment) => partitionAssignment.replicas.contains
4       }.map {
5         case (partition, _) => new TopicPartition(topic, partition)
6       }
7     }.toSet
```

再比如，如果 Kafka 要获取某个主题的所有分区对象，代码可以这样写：

```
复制代码
1 partitionAssignments.getOrElse(topic, mutable.Map.empty).map {
2       case (partition, _) => new TopicPartition(topic, partition)
3     }.toSet
```

实际上，这两段代码分别是 ControllerContext.scala 中 partitionsOnBroker 方法和 partitionsForTopic 两个方法的主体实现代码。

讲到这里，9 个重要的元数据字段我就介绍完了。前面说过，ControllerContext 中一共定义了 17 个元数据字段，你可以结合这 9 个字段，把其余 8 个的定义也过一遍，做到心中有数。**你对 Controller 元数据掌握得越好，就越能清晰地理解 Controller 在集群中发挥的作用。**

值得注意的是，在学习每个元数据字段时，除了它的定义之外，我建议你去搜索一下，与之相关的工具方法都是如何实现的。如果后面你想要新增获取或更新元数据的方法，你要对操作它们的代码有很强的把控力才行。

## 总结

今天，我们揭开了 Kafka 重要组件 Controller 的学习大幕。我给出了 Controller 模块的学习路线，还介绍了 Controller 的重要元数据。

Controller 元数据：Controller 当前定义了 17 种元数据，涵盖 Kafka 集群数据的方方面面。

ControllerContext：定义元数据以及操作它们的类。

关键元数据字段：最重要的元数据包括 offlinePartitionCount、liveBrokers、partitionAssignments 等。

ControllerContext 工具方法：ControllerContext 类定义了很多实用方法来管理这些元数据信息。

下节课，我们将学习 Controller 是如何给 Broker 发送请求的。Controller 与 Broker 进行交互与通信，是 Controller 奠定王者地位的重要一环，我会向你详细解释它是如何做到这一点的。

## 课后讨论

我今天并未给出所有的元数据说明，请你自行结合代码分析一下，partitionLeadershipInfo 里面保存的是什么数据？

欢迎你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。

上一篇　10 | KafkaApis：Kafka最重要的源码入口，没有之一

下一篇　12 | ControllerChannelManager：Controller如何管理请求发送?

## 精选留言 (7)

写留言

**胡夕** 置顶

2020-05-19

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，咱们重点了解了Kafka中重要的源码入口类KafkaApis。课后我请你思考如果一个Consumer要向Broker提交位移，它应该具备什么权限以及声明权限的具体代码位置哪段代码。下面我给出我的答案：消费者要具有GROUP的READ权限和TOPIC的READ权限才...

展开

**曾轼麟**

2020-05-31

我个人比较期待kafka摆脱zookeeper的时候，之前看过一篇文章，对比kafka和RocketMQ的性能对比，其中总结出，kafka的性能会受到topic数量的增加而下降，看了源码后才逐渐明白其实制约kafka的正是zookeeper

展开

**曾轼麟**
2020-05-31

在kafka中我发现经常使用epoch的方式来判断版本新旧，其实epoch这种设计思想类似于乐观锁的方式

**曾轼麟**
2020-05-31

partitionLeadershipInfo存储的主要是leader，leaderEpoch，isr集合，zkVersion，这些都定义在LeaderAndIsr这个类里面

展开

**伯安知心**
2020-05-15

您说删除的watcher，oncontrollerfailover重新初始化上下文，我感觉代价昂贵了些，就应该做这么多事吗?

作者回复: 可以具体说说哪一步在您看来是多余的，也许是个优化的方向：）

1

**伯安知心**
2020-05-15

还有请问partitionLeadershipInfo中controller的epoch是干吗的呢?

作者回复: 你可以认为是controller换了多少次。比如epoch = 5，说明controller前前后后更换过6次

1

**伯安知心**
2020-05-15

partitionLeadershipInfo总的来说每个分区对应的分区主副本，isr集合，还有controller的epoch数，

作者回复: 嗯，是的