

## 15 | 如何理解Controller在Kafka集群中的作用？

2020-05-23 胡夕

Kafka核心源码解读

[进入课程 >](#)



讲述：胡夕

时长 16:06 大小 14.76M



你好，我是胡夕。

上节课，我们学习了 Controller 选举的源码，了解了 Controller 组件的选举触发场景，以及它是如何被选举出来的。Controller 就绪之后，就会行使它作为控制器的重要权利了，包括管理集群成员、维护主题、操作元数据，等等。

之前在学习 Kafka 的时候，我一直很好奇，新启动的 Broker 是如何加入到集群中的。官方文档里的解释是：“Adding servers to a Kafka cluster is easy, just assign them a unique broker id and start up Kafka on your new servers.” 显然，你只要启动 Broker 进程，就可以实现集群的扩展，甚至包括集群元数据信息的同步。



不过，你是否思考过，这一切是怎么做到的呢？其实，这就是 Controller 组件源码提供的一个重要功能：管理新集群成员。

当然，作为核心组件，Controller 提供的功能非常多。除了集群成员管理，主题管理也是一个极其重要的功能。今天，我就带你深入了解下它们的实现代码。可以说，这是 Controller 最核心的两个功能，它们几乎涉及到了集群元数据中的所有重要数据。掌握了这些，之后你在探索 Controller 的其他代码时，会更加游刃有余。

## 集群成员管理

首先，我们来看 Controller 管理集群成员部分的代码。这里的成员管理包含两个方面：

1. 成员数量的管理，主要体现在新增成员和移除现有成员；
2. 单个成员的管理，如变更单个 Broker 的数据等。

## 成员数量管理

每个 Broker 在启动的时候，会在 ZooKeeper 的 `/brokers/ids` 节点下创建一个名为 `broker.id` 参数值的临时节点。


举个例子，假设 Broker 的 `broker.id` 参数值设置为 1001，那么，当 Broker 启动后，你会在 ZooKeeper 的 `/brokers/ids` 下观测到一个名为 1001 的子节点。该节点的内容包括了 Broker 配置的主机名、端口号以及所用监听器的信息（注意：这里的监听器和上面说的 ZooKeeper 监听器不是一回事）。

当该 Broker 正常关闭或意外退出时，ZooKeeper 上对应的临时节点会自动消失。

基于这种临时节点的机制，Controller 定义了 `BrokerChangeHandler` 监听器，专门负责监听 `/brokers/ids` 下的子节点数量变化。

一旦发现新增或删除 Broker，`/brokers/ids` 下的子节点数目一定会发生变化。这会被 Controller 侦测到，进而触发 `BrokerChangeHandler` 的处理方法，即 `handleChildChange` 方法。

我给出 BrokerChangeHandler 的代码。可以看到，这里面定义了 handleChildChange 方法：

 复制代码

```
1 class BrokerChangeHandler(eventManager: ControllerEventManager) extends ZNodeCl
2   // Broker ZooKeeper ZNode: /brokers/ids
3   override val path: String = BrokerIdsZNode.path
4   override def handleChildChange(): Unit = {
5     eventManager.put(BrokerChange) // 仅仅是向事件队列写入BrokerChange事件
6   }
7 }
```

该方法的作用就是向 Controller 事件队列写入一个 BrokerChange 事件。**事实上，Controller 端定义的所有 Handler 的处理逻辑，都是向事件队列写入相应的 ControllerEvent，真正的事件处理逻辑位于 KafkaController 类的 process 方法中。**

那么，接下来，我们就来看 process 方法。你会发现，处理 BrokerChange 事件的方法实际上是 processBrokerChange，代码如下：

 复制代码

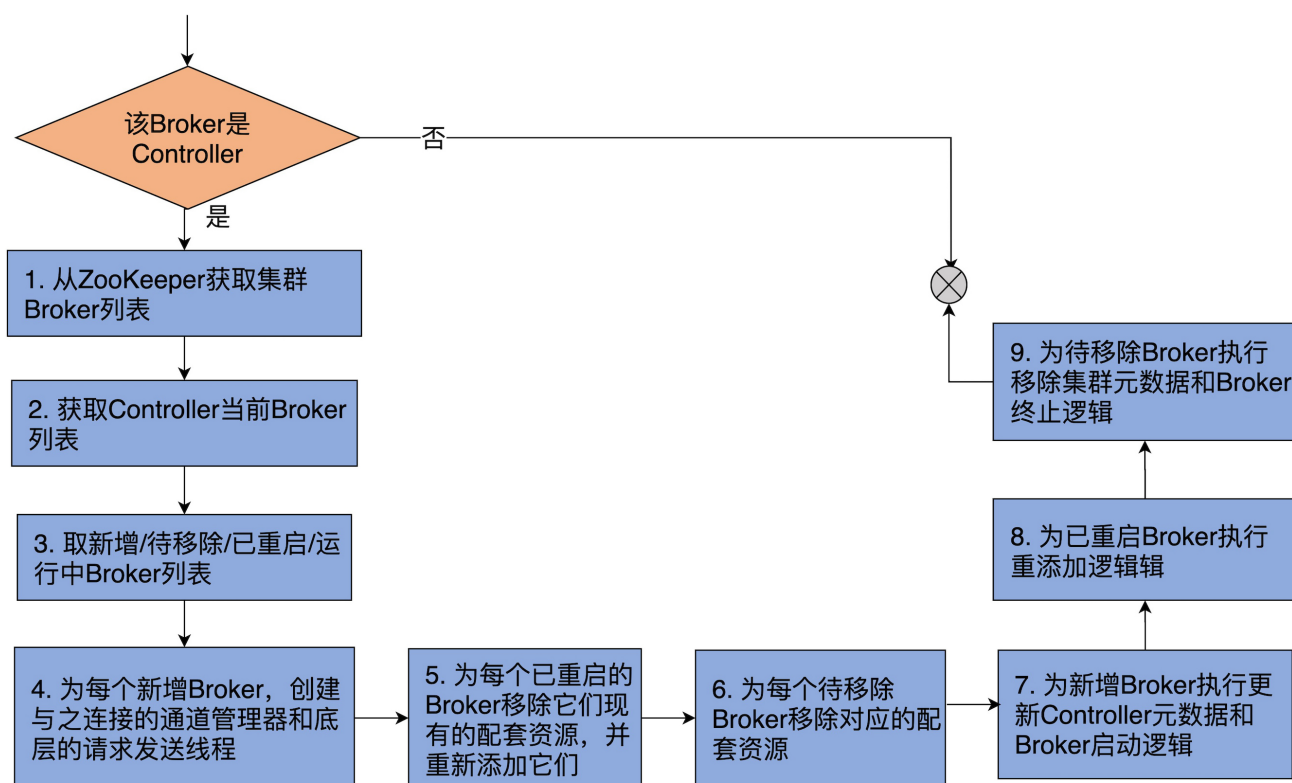
```
1 private def processBrokerChange(): Unit = {
2   // 如果该Broker不是Controller，自然无权处理，直接返回
3   if (!isActive) return
4   // 第1步：从ZooKeeper中获取集群Broker列表
5   val curBrokerAndEpochs = zkClient.getAllBrokerAndEpochsInCluster
6   val curBrokerIdAndEpochs = curBrokerAndEpochs map { case (broker, epoch) =>
7     val curBrokerIds = curBrokerIdAndEpochs.keySet
8     // 第2步：获取Controller当前保存的Broker列表
9     val liveOrShuttingDownBrokerIds = controllerContext.liveOrShuttingDownBroker
10    // 第3步：比较两个列表，获取新增Broker列表、待移除Broker列表、
11    // 已重启Broker列表和当前运行中的Broker列表
12    val newBrokerIds = curBrokerIds.diff(liveOrShuttingDownBrokerIds)
13    val deadBrokerIds = liveOrShuttingDownBrokerIds.diff(curBrokerIds)
14    val bouncedBrokerIds = (curBrokerIds & liveOrShuttingDownBrokerIds)
15      .filter(brokerId => curBrokerIdAndEpochs(brokerId) > controllerContext.livi
16    val newBrokerAndEpochs = curBrokerAndEpochs.filter { case (broker, _) => newl
17    val bouncedBrokerAndEpochs = curBrokerAndEpochs.filter { case (broker, _) =>
18    val newBrokerIdsSorted = newBrokerIds.toSeq.sorted
19    val deadBrokerIdsSorted = deadBrokerIds.toSeq.sorted
20    val liveBrokerIdsSorted = curBrokerIds.toSeq.sorted
21    val bouncedBrokerIdsSorted = bouncedBrokerIds.toSeq.sorted
22    info(s"Newly added brokers: ${newBrokerIdsSorted.mkString(",")}, " +
23      s"deleted brokers: ${deadBrokerIdsSorted.mkString(",")}, " +
24      s"bounced brokers: ${bouncedBrokerIdsSorted.mkString(",")}, " +
```

```

25     s"all live brokers: ${liveBrokerIdsSorted.mkString(",")}"
26 // 第4步: 为每个新增Broker创建与之连接的通道管理器和
27 // 底层请求发送线程 (RequestSendThread)
28 newBrokerAndEpochs.keySet.foreach(
29     controllerChannelManager.addBroker)
30 // 第5步: 为每个已重启的Broker移除它们现有的配套资源
31 // (通道管理器、RequestSendThread等), 并重新添加它们
32 bouncedBrokerIds.foreach(controllerChannelManager.removeBroker)
33 bouncedBrokerAndEpochs.keySet.foreach(
34     controllerChannelManager.addBroker)
35 // 第6步: 为每个待移除Broker移除对应的配套资源
36 deadBrokerIds.foreach(controllerChannelManager.removeBroker)
37 // 第7步: 为新增Broker执行更新Controller元数据和Broker启动逻辑
38 if (newBrokerIds.nonEmpty) {
39     controllerContext.addLiveBrokers(newBrokerAndEpochs)
40     onBrokerStartup(newBrokerIdsSorted)
41 }
42 // 第8步: 为已重启Broker执行重添加逻辑, 包含
43 // 更新ControllerContext、执行Broker重启动逻辑
44 if (bouncedBrokerIds.nonEmpty) {
45     controllerContext.removeLiveBrokers(bouncedBrokerIds)
46     onBrokerFailure(bouncedBrokerIdsSorted)
47     controllerContext.addLiveBrokers(bouncedBrokerAndEpochs)
48     onBrokerStartup(bouncedBrokerIdsSorted)
49 }
50 // 第9步: 为待移除Broker执行移除ControllerContext和Broker终止逻辑
51 if (deadBrokerIds.nonEmpty) {
52     controllerContext.removeLiveBrokers(deadBrokerIds)
53     onBrokerFailure(deadBrokerIdsSorted)
54 }
55 if (newBrokerIds.nonEmpty || deadBrokerIds.nonEmpty ||
56     bouncedBrokerIds.nonEmpty) {
57     info(s"Updated broker epochs cache: ${controllerContext.liveBrokerIdAndEpochs}")
58 }
59 }

```

代码有点长, 你可以看下我添加的重点注释。同时, 我再画一张图, 帮你梳理下这个方法做的事情。



极客时间

整个方法共有 9 步。

第 1~3 步：

前两步分别是 从 ZooKeeper 和 ControllerContext 中获取 Broker 列表；第 3 步是获取 4 个 Broker 列表：新增 Broker 列表、待移除 Broker 列表、已重启的 Broker 列表和当前运行中的 Broker 列表。

假设前两步中的 Broker 列表分别用 A 和 B 表示，由于 Kafka 以 ZooKeeper 上的数据为权威数据，因此，A 就是最新的运行中 Broker 列表，“A-B”就表示新增的 Broker，“B-A”就表示待移除的 Broker。

已重启的 Broker 的判断逻辑要复杂一些，它判断的是  $A \cap B$  集合中的那些 Epoch 值变更了的 Broker。你大体上可以把 Epoch 值理解为 Broker 的版本或重启的次数。显然，Epoch 值变更了，就说明 Broker 发生了重启行为。

第 4~9 步：

拿到这些集合之后，Controller 会分别为这 4 个 Broker 列表执行相应的操作，也就是这个方法中第 4~9 步要做的事情。总体上，这些相应的操作分为 3 类。


执行元数据更新操作：调用 ControllerContext 类的各个方法，更新不同的集群元数据信息。比如需要将新增 Broker 加入到集群元数据，将待移除 Broker 从元数据中移除等。

执行 Broker 终止操作：为待移除 Broker 和已重启 Broker 调用 onBrokerFailure 方法。

执行 Broker 启动操作：为已重启 Broker 和新增 Broker 调用 onBrokerStartup 方法。

下面我们深入了解下 onBrokerFailure 和 onBrokerStartup 方法的逻辑。相比于其他方法，这两个方法的代码逻辑有些复杂，要做的事情也很多，因此，我们重点研究下它们。

首先是处理 Broker 终止逻辑的 onBrokerFailure 方法，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   // 第1步：为每个待移除Broker，删除元数据对象中的相关项
4   deadBrokers.foreach(controllerContext.replicasOnOfflineDirs.remove
5   // 第2步：将待移除Broker从元数据对象中处于已关闭状态的Broker列表中去除
6   val deadBrokersThatWereShuttingDown =
7     deadBrokers.filter(id => controllerContext.shuttingDownBrokerIds.remove(id)
8   if (deadBrokersThatWereShuttingDown.nonEmpty)
9     info(s"Removed ${deadBrokersThatWereShuttingDown.mkString(",")} from list ")
10  // 第3步：找出待移除Broker上的所有副本对象，执行相应操作，
11  // 将其置为“不可用状态”（即Offline）
12  val allReplicasOnDeadBrokers = controllerContext.replicasOnBrokers(deadBrokers)
13  onReplicasBecomeOffline(allReplicasOnDeadBrokers)
14  // 第4步：注销注册的BrokerModificationsHandler监听器
15  unregisterBrokerModificationsHandler(deadBrokers)
16 }
```

Broker 终止，意味着我们必须删除 Controller 元数据缓存中与之相关的所有项，还要处理这些 Broker 上保存的副本。最后，我们还要注销之前为该 Broker 注册的 BrokerModificationsHandler 监听器。



其实，主体逻辑在于如何处理 Broker 上的副本对象，即 `onReplicasBecomeOffline` 方法。该方法大量调用了 Kafka 副本管理器和分区管理器的相关功能，后面我们会专门学习这两个管理器，因此这里我就不展开讲了。

现在，我们看 `onBrokerStartup` 方法。它是处理 Broker 启动的方法，也就是 Controller 端应对集群新增 Broker 启动的方法。同样，我先给出带注释的完整方法代码：

 复制代码

```
1 private def onBrokerStartup(newBrokers: Seq[Int]): Unit = {
2   info(s"New broker startup callback for ${newBrokers.mkString(",")}")
3   // 第1步：移除元数据中新增Broker对应的副本集合
4   newBrokers.foreach(controllerContext.replicasOnOfflineDirs.remove)
5   val newBrokersSet = newBrokers.toSet
6   val existingBrokers = controllerContext.
7     liveOrShuttingDownBrokerIds.diff(newBrokersSet)
8   // 第2步：给集群现有Broker发送元数据更新请求，令它们感知到新增Broker的到来
9   sendUpdateMetadataRequest(existingBrokers.toSeq, Set.empty)
10  // 第3步：给新增Broker发送元数据更新请求，令它们同步集群当前的所有分区数据
11  sendUpdateMetadataRequest(newBrokers, controllerContext.partitionLeadershipI
12  val allReplicasOnNewBrokers = controllerContext.replicasOnBrokers(newBrokers:
13  // 第4步：将新增Broker上的所有副本设置为Online状态，即可用状态
14  replicaStateMachine.handleStateChanges(
15    allReplicasOnNewBrokers.toSeq, OnlineReplica)
16  partitionStateMachine.triggerOnlinePartitionStateChange()
17  // 第5步：重启之前暂停的副本迁移操作
18  maybeResumeReassignments { (_, assignment) =>
19    assignment.targetReplicas.exists(newBrokersSet.contains)
20  }
21  val replicasForTopicsToBeDeleted = allReplicasOnNewBrokers.filter(p => topicI
22  // 第6步：重启之前暂停的主题删除操作
23  if (replicasForTopicsToBeDeleted.nonEmpty) {
24    info(s"Some replicas ${replicasForTopicsToBeDeleted.mkString(",")} for top
25      s"${controllerContext.topicsToBeDeleted.mkString(",")} are on the newly
26      s"${newBrokers.mkString(",")}. Signaling restart of topic deletion for tl
27    topicDeletionManager.resumeDeletionForTopics(
28      replicasForTopicsToBeDeleted.map(_.topic))
29  }
30  // 第7步：为新增Broker注册BrokerModificationsHandler监听器
31  registerBrokerModificationsHandler(newBrokers)
32 }
```

如代码所示，第 1 步是移除新增 Broker 在元数据缓存中的信息。你可能会问：“这些 Broker 不都是新增的吗？元数据缓存中有它们的数据吗？”实际上，这里的 `newBrokers`

仅仅表示新启动的 Broker，它们不一定是全新的 Broker。因此，这里的删除元数据缓存是非常安全的做法。

第 2、3 步：分别给集群的已有 Broker 和新增 Broker 发送更新元数据请求。这样一来，整个集群上的 Broker 就可以互相感知到彼此，而且最终所有的 Broker 都能保存相同的分区数据。

第 4 步：将新增 Broker 上的副本状态置为 Online 状态。Online 状态表示这些副本正常提供服务，即 Leader 副本对外提供读写服务，Follower 副本自动向 Leader 副本同步消息。

第 5、6 步：分别重启可能因为新增 Broker 启动、而能够重新被执行的副本迁移和主题删除操作。


第 7 步：为所有新增 Broker 注册 BrokerModificationsHandler 监听器，允许 Controller 监控它们在 ZooKeeper 上的节点的数据变更。

## 成员信息管理

了解了 Controller 管理集群成员数量的机制之后，接下来，我们要重点学习下 Controller 如何监听 Broker 端信息的变更，以及具体的操作。

和管理集群成员类似，Controller 也是通过 ZooKeeper 监听器的方式来应对 Broker 的变化。这个监听器就是 BrokerModificationsHandler。一旦 Broker 的信息发生变更，该监听器的 handleDataChange 方法就会被调用，向事件队列写入 BrokerModifications 事件。

KafkaController 类的 processBrokerModification 方法负责处理这类事件，代码如下：

 复制代码

```
1 private def processBrokerModification(brokerId: Int): Unit = {
2     if (!isActive) return
3     // 第1步：获取目标Broker的详细数据，
4     // 包括每套监听器配置的主机名、端口号以及所使用的安全协议等
5     val newMetadataOpt = zkClient.getBroker(brokerId)
6     // 第2步：从元数据缓存中获得目标Broker的详细数据
7     val oldMetadataOpt = controllerContext.liveOrShuttingDownBroker(brokerId)
8 }
```



```

9     if (newMetadataOpt.nonEmpty && oldMetadataOpt.nonEmpty) {
10         val oldMetadata = oldMetadataOpt.get
11         val newMetadata = newMetadataOpt.get
12         // 第3步: 如果两者不相等, 说明Broker数据发生了变更
13         // 那么, 更新元数据缓存, 以及执行onBrokerUpdate方法处理Broker更新的逻辑
14         if (newMetadata.endPoints != oldMetadata.endPoints) {
15             info(s"Updated broker metadata: $oldMetadata -> $newMetadata")
16             controllerContext.updateBrokerMetadata(oldMetadata, newMetadata)
17             onBrokerUpdate(brokerId)
18         }
19     }
20 }

```


该方法首先获取 ZooKeeper 上最权威的 Broker 数据, 将其与元数据缓存上的数据进行比对。如果发现两者不一致, 就会更新元数据缓存, 同时调用 onBrokerUpdate 方法执行更新逻辑。

那么, onBrokerUpdate 方法又是如何实现的呢? 我们先看下代码:

```

1 private def onBrokerUpdate(updatedBrokerId: Int): Unit = {
2     info(s"Broker info update callback for $updatedBrokerId")
3     // 给集群所有Broker发送UpdateMetadataRequest, 让它们去更新元数据
4     sendUpdateMetadataRequest(
5         controllerContext.liveOrShuttingDownBrokerIds.toSeq, Set.empty)
6 }

```

 复制代码

可以看到, onBrokerUpdate 就是向集群所有 Broker 发送更新元数据信息请求, 把变更信息广播出去。

怎么样, 应对 Broker 信息变更的方法还是比较简单的吧?

## 主题管理

除了维护集群成员之外, Controller 还有一个重要的任务, 那就是对所有主题进行管理, 主要包括主题的创建、变更与删除。


掌握了前面集群成员管理的方法, 在学习下面的内容时会轻松很多。因为它们的实现机制是一脉相承的, 几乎没有任何差异。

## 主题创建 / 变更

我们重点学习下主题是如何被创建的。实际上，主题变更与创建是相同的逻辑，因此，源码使用了一套监听器统一处理这两种情况。

你一定使用过 Kafka 的 kafka-topics 脚本或 AdminClient 创建主题吧？实际上，这些工具仅仅是向 ZooKeeper 对应的目录下写入相应的数据而已，那么，Controller，或者说 Kafka 集群是如何感知到新创建的主题的呢？

这当然要归功于监听主题路径的 ZooKeeper 监听器：TopicChangeHandler。代码如下：

 复制代码

```
1 class TopicChangeHandler(eventManager: ControllerEventManager) extends ZNodeCh
2   // ZooKeeper节点: /brokers/topics
3   override val path: String = TopicsZNode.path
4   // 向事件队列写入TopicChange事件
5   override def handleChildChange(): Unit = eventManager.put(TopicChange)
6 }
```

代码中的 TopicsZNode.path 就是 ZooKeeper 下 /brokers/topics 节点。一旦该节点下新增了主题信息，该监听器的 handleChildChange 就会被触发，Controller 通过 ControllerEventManager 对象，向事件队列写入 TopicChange 事件。

KafkaController 的 process 方法接到该事件后，调用 processTopicChange 方法执行主题创建。代码如下：

 复制代码

```
1 private def processTopicChange(): Unit = {
2   if (!isActive) return
3   // 第1步: 从ZooKeeper中获取所有主题
4   val topics = zkClient.getAllTopicsInCluster(true)
5   // 第2步: 与元数据缓存比对, 找出新增主题列表与已删除主题列表
6   val newTopics = topics -- controllerContext.allTopics
7   val deletedTopics = controllerContext.allTopics.diff(topics)
8   // 第3步: 使用ZooKeeper中的主题列表更新元数据缓存
9   controllerContext.setAllTopics(topics)
10  // 第4步: 为新增主题注册分区变更监听器
11  // 分区变更监听器是监听主题分区变更的
12  registerPartitionModificationsHandlers(newTopics.toSeq)
13  // 第5步: 从ZooKeeper中获取新增主题的副本分配情况
14  val addedPartitionReplicaAssignment = zkClient.getFullReplicaAssignmentForToi
```

```

15 // 第6步：清除元数据缓存中属于已删除主题的缓存项
16 deletedTopics.foreach(controllerContext.removeTopic)
17 // 第7步：为新增主题更新元数据缓存中的副本分配条目
18 addedPartitionReplicaAssignment.foreach {
19     case (topicAndPartition, newReplicaAssignment) => controllerContext.update
20 }
21 info(s"New topics: [$newTopics], deleted topics: [$deletedTopics], new parti
22     s"[$addedPartitionReplicaAssignment]")
23 // 第8步：调整新增主题所有分区以及所属所有副本的运行状态为“上线”状态
24 if (addedPartitionReplicaAssignment.nonEmpty)
25     onNewPartitionCreation(addedPartitionReplicaAssignment.keySet)
26 }
27


```

虽然一共有 8 步，但大部分的逻辑都与更新元数据缓存项有关，因此，处理逻辑总体上还是比较简单的。需要注意的是，第 8 步涉及到了使用分区管理器和副本管理器来调整分区和副本状态。后面我们会详细介绍。这里你只需要知道，分区和副本处于“上线”状态，就表明它们能够正常工作，就足够了。

## 主题删除

和主题创建或变更类似，删除主题也依赖 ZooKeeper 监听器完成。

Controller 定义了 TopicDeletionHandler，用它来实现对删除主题的监听，代码如下：

 复制代码

```

1 class TopicDeletionHandler(eventManager: ControllerEventManager) extends ZNode
2 // ZooKeeper节点: /admin/delete_topics
3 override val path: String = DeleteTopicsZNode.path
4 // 向事件队列写入TopicDeletion事件
5 override def handleChildChange(): Unit = eventManager.put(TopicDeletion)
6 }


```

这里的 DeleteTopicsZNode.path 指的是 /admin/delete\_topics 节点。目前，无论是 kafka-topics 脚本，还是 AdminClient，删除主题都是在 /admin/delete\_topics 节点下创建名为待删除主题名的子节点。

比如，如果我要删除 test-topic 主题，那么，Kafka 的删除命令仅仅是在 ZooKeeper 上创建 /admin/delete\_topics/test-topic 节点。一旦监听到该节点被创建，

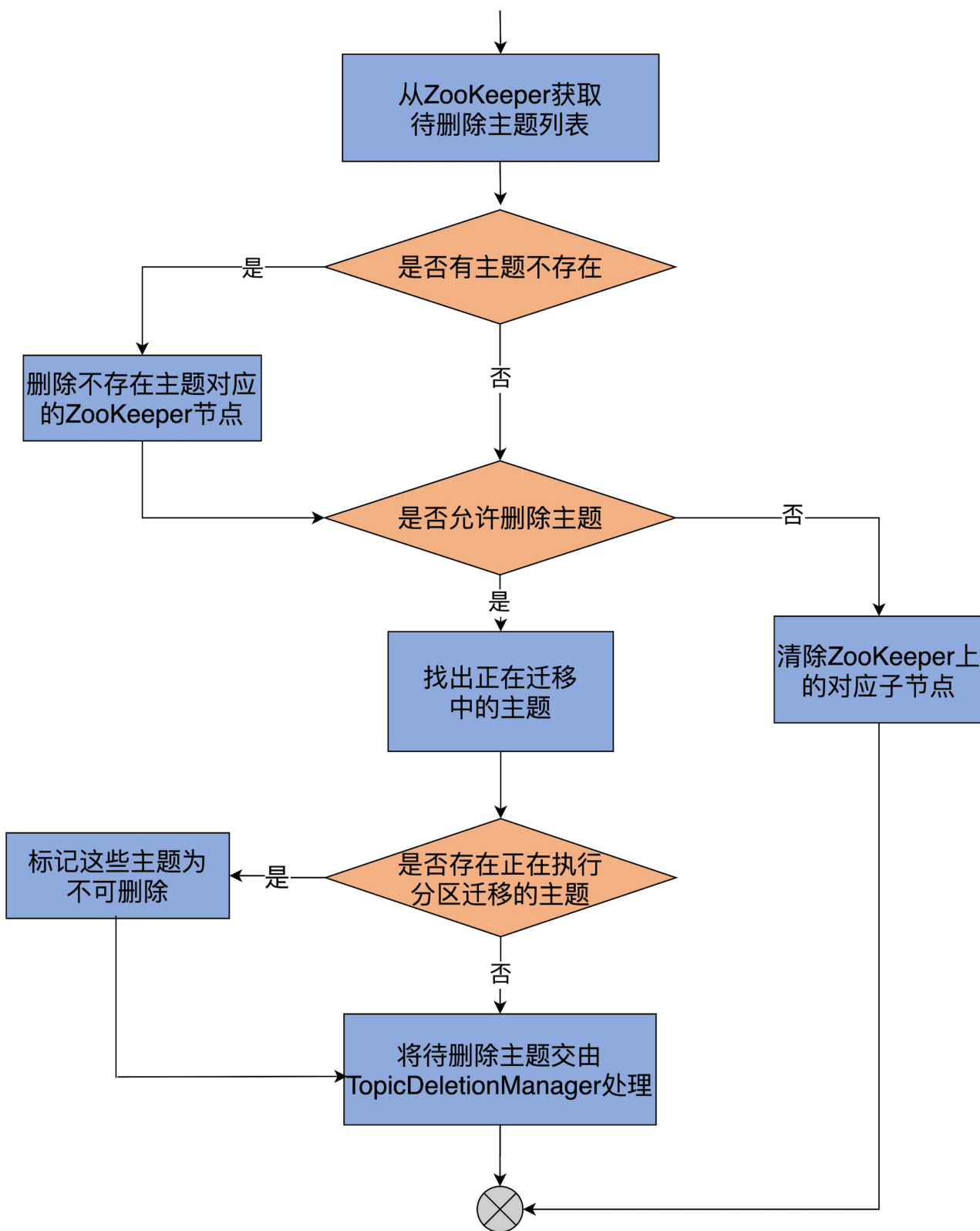
TopicDeletionHandler 的 handleChildChange 方法就会被触发，Controller 会向事件队列写入 TopicDeletion 事件。

处理 TopicDeletion 事件的方法是 processTopicDeletion，代码如下：

 复制代码

```
1 private def processTopicDeletion(): Unit = {
2     if (!isActive) return
3     // 从ZooKeeper中获取待删除主题列表
4     var topicsToBeDeleted = zkClient.getTopicDeletions.toSet
5     debug(s"Delete topics listener fired for topics ${topicsToBeDeleted.mkString"}
6     // 找出不存在的主题列表
7     val nonExistentTopics = topicsToBeDeleted -- controllerContext.allTopics
8     if (nonExistentTopics.nonEmpty) {
9         warn(s"Ignoring request to delete non-existing topics ${nonExistentTopics.mkString"}
10        zkClient.deleteTopicDeletions(nonExistentTopics.toSeq, controllerContext.executor)
11    }
12    topicsToBeDeleted --= nonExistentTopics
13    // 如果delete.topic.enable参数设置成true
14    if (config.deleteTopicEnable) {
15        if (topicsToBeDeleted.nonEmpty) {
16            info(s"Starting topic deletion for topics ${topicsToBeDeleted.mkString"}
17            topicsToBeDeleted.foreach { topic =>
18                val partitionReassignmentInProgress = controllerContext.partitionsBeingReassigned.contains(topic)
19                if (partitionReassignmentInProgress)
20                    topicDeletionManager.markTopicIneligibleForDeletion(
21                        Set(topic), reason = "topic reassignment in progress")
22            }
23            // 将待删除主题插入到删除等待集合交由TopicDeletionManager处理
24            topicDeletionManager.enqueueTopicsForDeletion(topicsToBeDeleted)
25        }
26    } else { // 不允许删除主题
27        info(s"Removing $topicsToBeDeleted since delete topic is disabled")
28        // 清除ZooKeeper下/admin/delete_topics下的子节点
29        zkClient.deleteTopicDeletions(topicsToBeDeleted.toSeq, controllerContext.executor)
30    }
31 }
```

为了帮助你更直观地理解，我再画一张图来说明下：



首先，代码从 ZooKeeper 的 `/admin/delete_topics` 下获取子节点列表，即待删除主题列表。

之后，比对元数据缓存中的主题列表，获知压根不存在的主题列表。如果确实有不存在的主题，删除 `/admin/delete_topics` 下对应的子节点就行了。同时，代码会更新待删除主题列表。

表，将这些不存在的主题剔除掉。

接着，代码会检查 Broker 端参数 `delete.topic.enable` 的值。如果该参数为 `false`，即不允许删除主题，代码就会清除 ZooKeeper 下的对应子节点，不会做其他操作。反之，代码会遍历待删除主题列表，将那些正在执行分区迁移的主题暂时设置成“不可删除”状态。

最后，把剩下可以删除的主题交由 `TopicDeletionManager`，由它执行真正的删除逻辑。

这里的 `TopicDeletionManager` 是 Kafka 专门负责删除主题的管理器，下节课我会详细讲解它的代码实现。

## 总结

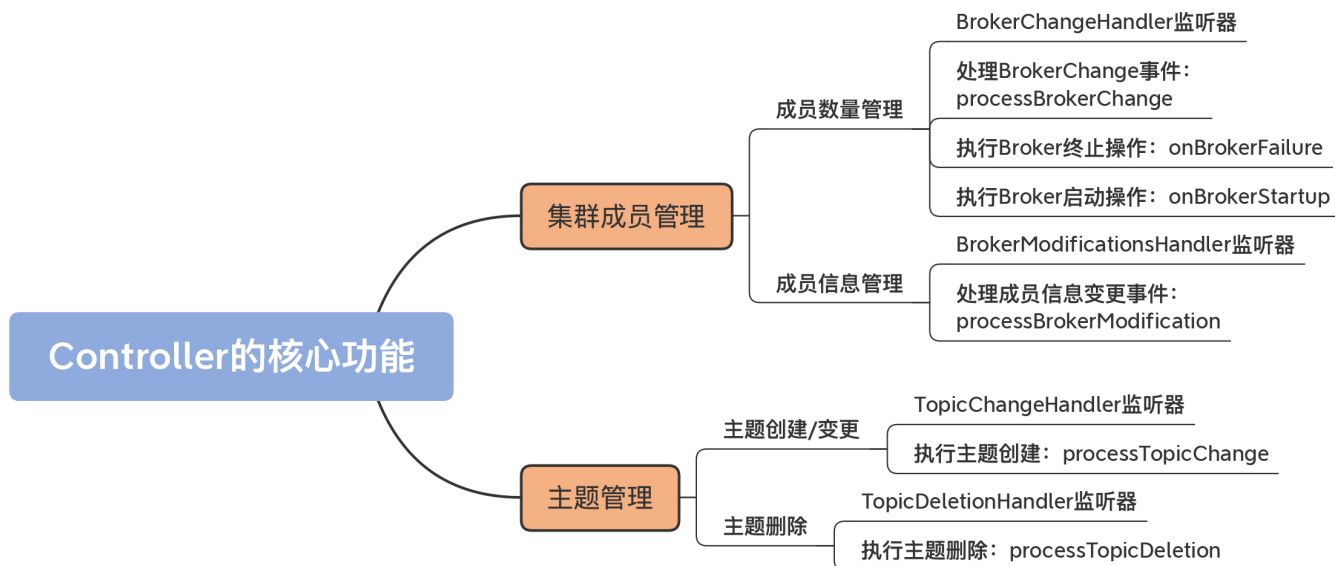
今天，我们学习了 Controller 的两个主要功能：管理集群 Broker 成员和主题。这两个功能是 Controller 端提供的重要服务。我建议你仔细地查看这两部分的源码，弄明白 Controller 是如何管理集群中的重要资源的。

针对这些内容，我总结了几个重点，希望可以帮助你更好地理解和记忆。

**集群成员管理：**Controller 负责对集群所有成员进行有效管理，包括自动发现新增 Broker、自动处理下线 Broker，以及及时响应 Broker 数据的变更。

**主题管理：**Controller 负责对集群上的所有主题进行高效管理，包括创建主题、变更主题以及删除主题，等等。对于删除主题而言，实际的删除操作由底层的 `TopicDeletionManager` 完成。





接下来，我们将进入到下一个模块：状态机模块。在该模块中，我们将系统学习 Kafka 提供的三大状态机或管理器。Controller 非常依赖这些状态机对下辖的所有 Kafka 对象进行管理。在下一个模块中，我将带你深入了解分区或副本在底层的状态流转是怎么样的，你一定不要错过。

## 课后讨论

如果我们想要使用脚本命令增加一个主题的分区，你知道应该用 KafkaController 类中的哪个方法吗？

欢迎你在留言区畅所欲言，跟我交流讨论，也欢迎你把今天的内容分享给你的朋友。

# 6月-7月课表抢先看

## 充 ¥500 得 ¥580

赠「¥ 118 月球主题 AR 笔记本」



【点击】图片, 立即查看>>>

© 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 14 | Controller选举是怎么实现的?

下一篇 期中测试 | 这些源码知识, 你都掌握了吗?

### 精选留言 (4)

写留言



胡夕 置顶

2020-05-29

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Controller选举部分的代码。课后我请你思考这样一个问题: 源码中当Controller选举之后哪里更新的元数据请求? 其实这是在onControllerFailover方法中完成的。该方法中调用: ...

展开



RonnieXie

2020-05-27

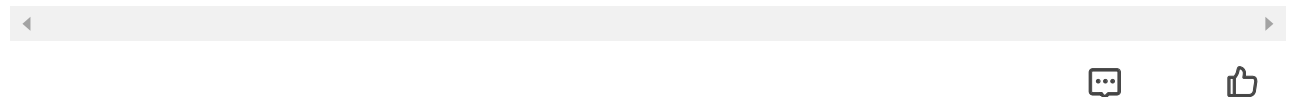
胡夕老师，我们项目最近在预研使用图数据库，刚看碰巧在一篇文章 (<https://zhuanlan.zhihu.com/p/99381529>)

看到你们公司使用JanusGraph，想问下这数据库稳定性如何，同时也看到另外一款百度开源的HugeGraph，

你们选型的JanusGraph看中的优势是什么？

展开 ∨

作者回复: 比较稳定，API也算好用。没怎么用过百度的图数据库。另外我们公司用图数据库很长时间了，那时HugeGraph还没有开源。



**RonnieXie**

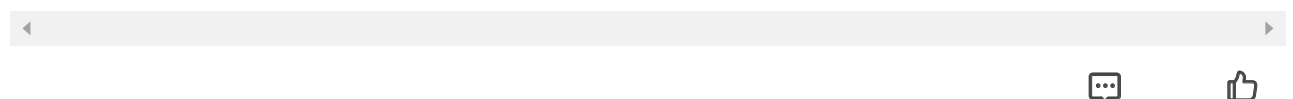
2020-05-27

老师，学习了关于Controller源码课，我再深入看相关源码，梳理了流程图，您看这流程图是否正确

<http://note.youdao.com/noteshare?id=1f999786b77ada75fbb58b2831770e47&sub=554F637E4F984C7B8036365C51E0758B>

展开 ∨

作者回复: 很棒的总结啊！我自己也学到了一些：)



**空知**

2020-05-24

调用脚本kafka-topics.sh

```
exec $(dirname $0)/kafka-run-class.sh kafka.admin.TopicCommand "$@"
```

对应的类object TopicCommand 的topicService.createTopic(opts)根据是否定义ZK信息走实现类ZookeeperTopicService或者AdminClientTopicService然后 ZookeeperTopicService里面...

展开 ∨

作者回复: 是的

