

18 | PartitionStateMachine: 揭秘分区状态机实现原理

2020-06-02 胡夕

Kafka核心源码解读

[进入课程 >](#)



讲述：胡夕

时长 22:47 大小 20.88M



你好，我是胡夕。今天我们进入到分区状态机（PartitionStateMachine）源码的学习。

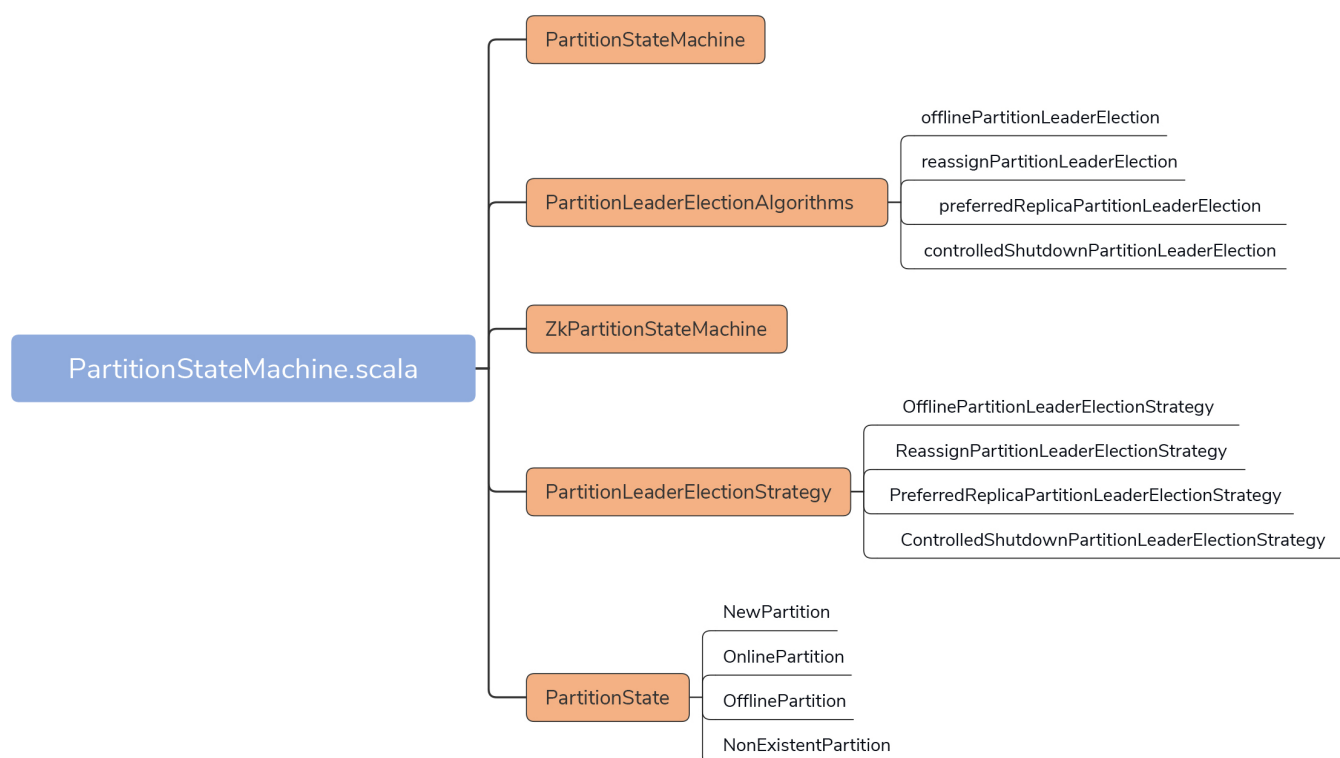
PartitionStateMachine 负责管理 Kafka 分区状态的转换，和 ReplicaStateMachine 是一脉相承的。从代码结构、实现功能和设计原理来看，二者都极为相似。上节课我们已经学过了 ReplicaStateMachine，相信你在学习这节课的 PartitionStateMachine 时，会轻松很多。

在面试的时候，很多面试官都非常喜欢问 Leader 选举的策略。学完了今天的课程之后，你不但能够说出 4 种 Leader 选举的场景，还能总结出它们的共性。对于面试来说，绝对是加分项！

话不多说，我们就正式开始吧。

PartitionStateMachine 简介

PartitionStateMachine.scala 文件位于 controller 包下，代码结构不复杂，可以看下这张思维导图：



代码总共有 5 大部分。

PartitionStateMachine：分区状态机抽象类。它定义了诸如 startup、shutdown 这样的公共方法，同时也给出了处理分区状态转换入口方法 handleStateChanges 的签名。

ZkPartitionStateMachine：PartitionStateMachine 唯一的继承子类。它实现了分区状态机的主体逻辑功能。和 ZkReplicaStateMachine 类似，ZkPartitionStateMachine 重写了父类的 handleStateChanges 方法，并配以私有的 doHandleStateChanges 方法，共同实现分区状态转换的操作。

PartitionState 接口及其实现对象：定义 4 类分区状态，分别是 NewPartition、OnlinePartition、OfflinePartition 和 NonExistentPartition。除此之外，还定义了它们之间的流转关系。

PartitionLeaderElectionStrategy 接口及其实现对象：定义 4 类分区 Leader 选举策略。你可以认为它们是发生 Leader 选举的 4 种场景。

PartitionLeaderElectionAlgorithms：分区 Leader 选举的算法实现。既然定义了 4 类选举策略，就一定有相应的实现代码，PartitionLeaderElectionAlgorithms 就提供了这 4 类选举策略的实现代码。

类定义与字段

PartitionStateMachine 和 ReplicaStateMachine 非常类似，我们先看下面这两段代码：

 复制代码

```
1 // PartitionStateMachine抽象类定义
2 abstract class PartitionStateMachine(
3     controllerContext: ControllerContext) extends Logging {
4     .....
5 }
6 // ZkPartitionStateMachine继承子类定义
7 class ZkPartitionStateMachine(config: KafkaConfig,
8     stateChangeLogger: StateChangeLogger,
9     controllerContext: ControllerContext,
10    zkClient: KafkaZkClient,
11    controllerBrokerRequestBatch: ControllerBrokerRequestBatch) extends Partition
12 // Controller所在Broker的Id
13 private val controllerId = config.brokerId
14 .....
15 }
```

从代码中，可以发现，它们的类定义一模一样，尤其是 ZkPartitionStateMachine 和 ZKReplicaStateMachine，它们接收的字段列表都是相同的。此刻，你应该可以体会到它们要做的处理逻辑，其实也是差不多的。


同理，ZkPartitionStateMachine 实例的创建和启动时机也跟 ZkReplicaStateMachine 的完全相同，即：每个 Broker 进程启动时，会在创建 KafkaController 对象的过程中，生成 ZkPartitionStateMachine 实例，而只有 Controller 组件所在的 Broker，才会启动分区状态机。

下面这段代码展示了 ZkPartitionStateMachine 实例创建和启动的位置：

```

1  class KafkaController(.....) {
2      .....
3      // 在KafkaController对象创建过程中，生成ZkPartitionStateMachine实例
4      val partitionStateMachine: PartitionStateMachine =
5          new ZkPartitionStateMachine(config, stateChangeLogger,
6              controllerContext, zkClient,
7              new ControllerBrokerRequestBatch(config,
8                  controllerChannelManager, eventManager, controllerContext,
9                      stateChangeLogger))
10     .....
11     private def onControllerFailover(): Unit = {
12         .....
13         replicaStateMachine.startup() // 启动副本状态机
14         partitionStateMachine.startup() // 启动分区状态机
15         .....
16     }
17 }

```

 复制代码

有句话我要再强调一遍：**每个 Broker 启动时，都会创建对应的分区状态机和副本状态机实例，但只有 Controller 所在的 Broker 才会启动它们。**如果 Controller 变更到其他 Broker，老 Controller 所在的 Broker 要调用这些状态机的 shutdown 方法关闭它们，新 Controller 所在的 Broker 调用状态机的 startup 方法启动它们。

分区状态

既然 ZkPartitionStateMachine 是管理分区状态转换的，那么，我们至少要知道分区都有哪些状态，以及 Kafka 规定的转换规则是什么。这就是 PartitionState 接口及其实现对象做的事情。和 ReplicaState 类似，PartitionState 定义了分区的状态空间以及流转规则。

我以 OnlinePartition 状态为例，说明下代码是如何实现流转的：

```

1  sealed trait PartitionState {
2      def state: Byte // 状态序号，无实际用途
3      def validPreviousStates: Set[PartitionState] // 合法前置状态集合
4  }
5
6  case object OnlinePartition extends PartitionState {
7      val state: Byte = 1
8      val validPreviousStates: Set[PartitionState] = Set(NewPartition, OnlineParti
9  }

```

 复制代码

如代码所示，每个 PartitionState 都定义了名为 validPreviousStates 的集合，也就是每个状态对应的合法前置状态集。

对于 OnlinePartition 而言，它的合法前置状态集包括 NewPartition、OnlinePartition 和 OfflinePartition。在 Kafka 中，从合法状态集以外的状态向目标状态进行转换，将被视为非法操作。

目前，Kafka 为分区定义了 4 类状态。

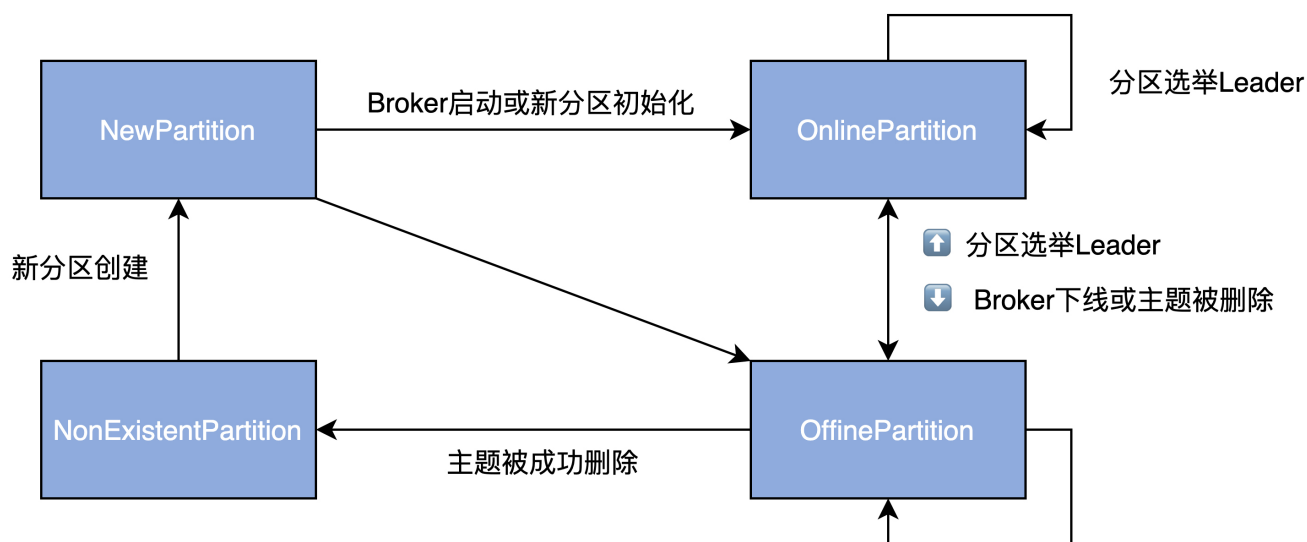
NewPartition：分区被创建后被设置成这个状态，表明它是一个全新的分区对象。处于这个状态的分区，被 Kafka 认为是“未初始化”，因此，不能选举 Leader。

OnlinePartition：分区正式提供服务时所处的状态。

OfflinePartition：分区下线后所处的状态。

NonExistentPartition：分区被删除，并且从分区状态机移除后所处的状态。

下图展示了完整的分区状态转换规则：



图中的双向箭头连接的两个状态可以彼此进行转换，如 OnlinePartition 和 OfflinePartition。Kafka 允许一个分区从 OnlinePartition 切换到 OfflinePartition，反之亦然。

另外，OnlinePartition 和 OfflinePartition 都有一根箭头指向自己，这表明 OnlinePartition 切换到 OnlinePartition 的操作是允许的。**当分区 Leader 选举发生的时候，就可能出现这种情况。接下来，我们就聊聊分区 Leader 选举那些事儿。**

分区 Leader 选举的场景及方法

刚刚我们说了两个状态机的相同点，接下来，我们要学习的分区 Leader 选举，可以说是 PartitionStateMachine 特有的功能了。

每个分区都必须选举出 Leader 才能正常提供服务，因此，对于分区而言，Leader 副本是非常重要的角色。既然这样，我们就必须要了解 Leader 选举什么流程，以及在代码中是如何实现的。我们重点学习下选举策略以及具体的实现方法代码。

PartitionLeaderElectionStrategy

先明确下分区 Leader 选举的含义，其实很简单，就是为 Kafka 主题的某个分区推选 Leader 副本。

那么，Kafka 定义了哪几种推选策略，或者说，在什么情况下需要执行 Leader 选举呢？

这就是 PartitionLeaderElectionStrategy 接口要做的事情，请看下面的代码：

 复制代码

```
1 // 分区Leader选举策略接口
2 sealed trait PartitionLeaderElectionStrategy
3 // 离线分区Leader选举策略
4 final case class OfflinePartitionLeaderElectionStrategy(
5     allowUnclean: Boolean) extends PartitionLeaderElectionStrategy
6 // 分区副本重分配Leader选举策略
7 final case object ReassignPartitionLeaderElectionStrategy
8     extends PartitionLeaderElectionStrategy
9 // 分区Preferred副本Leader选举策略
10 final case object PreferredReplicaPartitionLeaderElectionStrategy
11     extends PartitionLeaderElectionStrategy
12 // Broker Controlled关闭时Leader选举策略
13 final case object ControlledShutdownPartitionLeaderElectionStrategy
14     extends PartitionLeaderElectionStrategy
```

当前，分区 Leader 选举有 4 类场景。

OfflinePartitionLeaderElectionStrategy: 因为 Leader 副本下线而引发的分区 Leader 选举。

ReassignPartitionLeaderElectionStrategy: 因为执行分区副本重分配操作而引发的分区 Leader 选举。

PreferredReplicaPartitionLeaderElectionStrategy: 因为执行 Preferred 副本 Leader 选举而引发的分区 Leader 选举。


ControlledShutdownPartitionLeaderElectionStrategy: 因为正常关闭 Broker 而引发的分区 Leader 选举。

PartitionLeaderElectionAlgorithms

针对这 4 类场景，分区状态机的 PartitionLeaderElectionAlgorithms 对象定义了 4 个方法，分别负责为每种场景选举 Leader 副本，这 4 种方法是：

```
offlinePartitionLeaderElection;  
reassignPartitionLeaderElection;  
preferredReplicaPartitionLeaderElection;  
controlledShutdownPartitionLeaderElection.
```

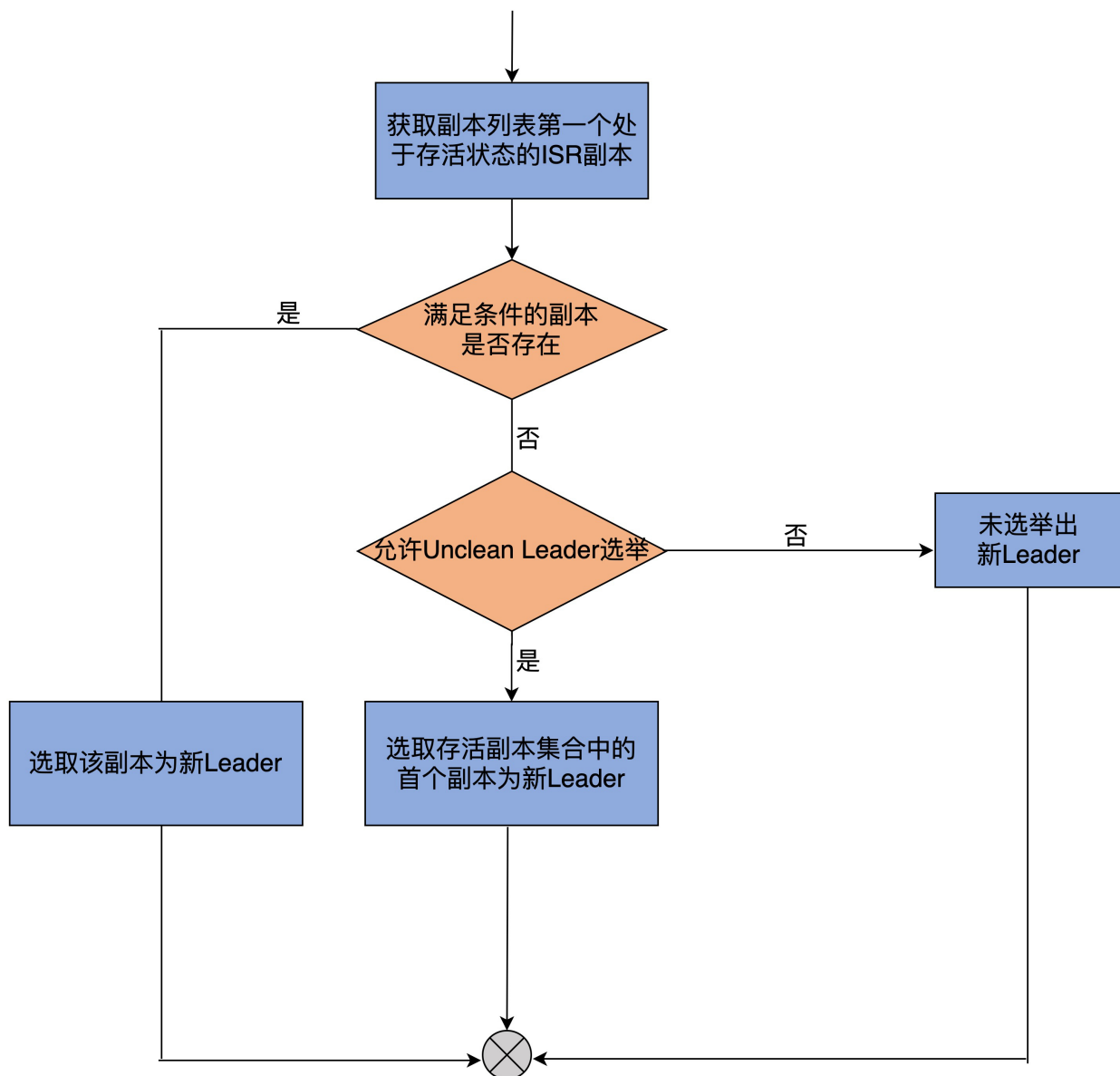
offlinePartitionLeaderElection 方法的逻辑是这 4 个方法中最复杂的，我们就先从它开始学起。

 复制代码

```
1 def offlinePartitionLeaderElection(assignment: Seq[Int],  
2   isr: Seq[Int], liveReplicas: Set[Int],  
3   uncleanLeaderElectionEnabled: Boolean, controllerContext: ControllerContext)  
4   // 从当前分区副本列表中寻找首个处于存活状态的ISR副本  
5   assignment.find(id => liveReplicas.contains(id) && isr.contains(id)).orElse  
6     // 如果找不到满足条件的副本，查看是否允许Unclean Leader选举  
7     // 即Broker端参数unclean.leader.election.enable是否等于true  
8     if (uncleanLeaderElectionEnabled) {  
9       // 选择当前副本列表中的第一个存活副本作为Leader  
10      val leaderOpt = assignment.find(liveReplicas.contains)  
11      if (leaderOpt.isDefined)  
12        controllerContext.stats.uncleanLeaderElectionRate.mark()  
13      leaderOpt  
14    } else {  
15      None // 如果不允许Unclean Leader选举，则返回None表示无法选举Leader
```

```
16     }  
17 }  
18 }
```

我再画一张流程图，帮助你理解它的代码逻辑：



这个方法总共接收 5 个参数。除了你已经很熟悉的 ControllerContext 类，其他 4 个非常值得我们花一些时间去探究下。

1.assignments

这是分区的副本列表。该列表有个专属的名称，叫 Assigned Replicas，简称 AR。当我们创建主题之后，使用 kafka-topics 脚本查看主题时，应该可以看到名为 Replicas 的一系列数据。这列数据显示的，就是主题下每个分区的 AR。assignments 参数类型是 Seq[Int]。这揭示了一个重要的事实：**AR 是有顺序的，而且不一定和 ISR 的顺序相同！**

2.isr

ISR 在 Kafka 中很有名气，它保存了分区所有与 Leader 副本保持同步的副本列表。注意，Leader 副本自己也在 ISR 中。另外，作为 Seq[Int]类型的变量，isr 自身也是有顺序的。

3.liveReplicas

从名字可以推断出，它保存了该分区下所有处于存活状态的副本。怎么判断副本是否存活呢？可以根据 Controller 元数据缓存中的数据来判定。简单来说，所有在运行中的 Broker 上的副本，都被认为是存活的。

4.uncleanLeaderElectionEnabled

在默认配置下，只要不是由 AdminClient 发起的 Leader 选举，这个参数的值一般是 false，即 Kafka 不允许执行 Unclean Leader 选举。所谓的 Unclean Leader 选举，是指在 ISR 列表为空的情况下，Kafka 选择一个非 ISR 副本作为新的 Leader。由于存在丢失数据的风险，目前，社区已经通过把 Broker 端参数 unclean.leader.election.enable 的默认值设置为 false 的方式，禁止 Unclean Leader 选举了。

值得一提的是，社区于 2.4.0.0 版本正式支持在 AdminClient 端为给定分区选举 Leader。目前的设计是，如果 Leader 选举是由 AdminClient 端触发的，那就默认开启 Unclean Leader 选举。不过，在学习 offlinePartitionLeaderElection 方法时，你可以认为 uncleanLeaderElectionEnabled=false，这并不会影响你对该方法的理解。

了解了这几个参数的含义，我们就可以研究具体的流程了。

代码首先会顺序搜索 AR 列表，并把第一个同时满足以下两个条件的副本作为新的 Leader 返回：

该副本是存活状态，即副本所在的 Broker 依然在运行中；

该副本在 ISR 列表中。

倘若无法找到这样的副本，代码会检查是否开启了 Unclean Leader 选举：如果开启了，则降低标准，只要满足上面第一个条件即可；如果未开启，则本次 Leader 选举失败，没有新 Leader 被选出。

其他 3 个方法要简单得多，我们直接看代码：

 复制代码

```
1 def reassignPartitionLeaderElection(reassignment: Seq[Int], isr: Seq[Int], live
2   reassignment.find(id => liveReplicas.contains(id) && isr.contains(id))
3 }
4
5 def preferredReplicaPartitionLeaderElection(assignment: Seq[Int], isr: Seq[Int]
6   assignment.headOption.filter(id => liveReplicas.contains(id) && isr.contains
7 }
8
9 def controlledShutdownPartitionLeaderElection(assignment: Seq[Int], isr: Seq[Int]
10   assignment.find(id => liveReplicas.contains(id) && isr.contains(id) && !shut
11 }
```

可以看到，它们的逻辑几乎是相同的，大概的原理都是从 AR，或给定的副本列表中寻找存活状态的 ISR 副本。

讲到这里，你应该已经知道 Kafka 为分区选举 Leader 的大体思路了。**基本上就是，找出 AR 列表（或给定副本列表）中首个处于存活状态，且在 ISR 列表的副本，将其作为新 Leader。**

处理分区状态转换的方法

掌握了刚刚的这些知识之后，现在，我们正式来看 PartitionStateMachine 的工作原理。

handleStateChanges

前面我提到过，handleStateChanges 是入口方法，所以我们先看它的方法签名：

```

1 def handleStateChanges(
2     partitions: Seq[TopicPartition],
3     targetState: PartitionState,
4     leaderElectionStrategy: Option[PartitionLeaderElectionStrategy]):
5     Map[TopicPartition, Either[Throwable, LeaderAndIsr]]

```

[复制代码](#)

如果用一句话概括 `handleStateChanges` 的作用，应该这样说：**`handleStateChanges` 把 `partitions` 的状态设置为 `targetState`，同时，还可能需要用 `leaderElectionStrategy` 策略为 `partitions` 选举新的 Leader，最终将 `partitions` 的 Leader 信息返回。**

其中，`partitions` 是待执行状态变更的目标分区列表，`targetState` 是目标状态，`leaderElectionStrategy` 是一个可选项，如果传入了，就表示要执行 Leader 选举。

下面是 `handleStateChanges` 方法的完整代码，我以注释的方式给出了主要的功能说明：

```

1 override def handleStateChanges(
2     partitions: Seq[TopicPartition],
3     targetState: PartitionState,
4     partitionLeaderElectionStrategyOpt: Option[PartitionLeaderElectionStrategy]
5 ): Map[TopicPartition, Either[Throwable, LeaderAndIsr]] = {
6     if (partitions.nonEmpty) {
7         try {
8             // 清空Controller待发送请求集合，准备本次请求发送
9             controllerBrokerRequestBatch.newBatch()
10            // 调用doHandleStateChanges方法执行真正的状态变更逻辑
11            val result = doHandleStateChanges(
12                partitions,
13                targetState,
14                partitionLeaderElectionStrategyOpt
15            )
16            // Controller给相关Broker发送请求通知状态变化
17            controllerBrokerRequestBatch.sendRequestsToBrokers(
18                controllerContext.epoch)
19            // 返回状态变更处理结果
20            result
21        } catch {
22            // 如果Controller易主，则记录错误日志，然后重新抛出异常
23            // 上层代码会捕获该异常并执行maybeResign方法执行卸任逻辑
24            case e: ControllerMovedException =>
25                error(s"Controller moved to another broker when moving some partition")
26                throw e
27            // 如果是其他异常，记录错误日志，封装错误返回
28            case e: Throwable =>
29                error(s"Error while moving some partitions to $targetState state", e)

```

[复制代码](#)

```

30         partitions.iterator.map(_ -> Left(e)).toMap
31     }
32 } else { // 如果partitions为空, 什么都不用做
33     Map.empty
34 }
35

```

整个方法就两步：第 1 步是，调用 `doHandleStateChanges` 方法执行分区状态转换；第 2 步是，Controller 给相关 Broker 发送请求，告知它们这些分区的状态变更。至于哪些 Broker 属于相关 Broker，以及给 Broker 发送哪些请求，实际上是在第 1 步中被确认的。

当然，这个方法的重点，就是第 1 步中调用的 `doHandleStateChanges` 方法。

doHandleStateChanges

先来看这个方法的实现：

[复制代码](#)

```

1 private def doHandleStateChanges(
2     partitions: Seq[TopicPartition],
3     targetState: PartitionState,
4     partitionLeaderElectionStrategyOpt: Option[PartitionLeaderElectionStrategy]
5 ): Map[TopicPartition, Either[Throwable, LeaderAndIsr]] = {
6     val stateChangeLog = stateChangeLogger.withControllerEpoch(controllerContext.controllerEpoch)
7     val traceEnabled = stateChangeLog.isTraceEnabled
8     // 初始化新分区的状态为NonExistentPartition
9     partitions.foreach(partition => controllerContext.putPartitionStateIfNotExists(partition, targetState))
10    // 找出要执行非法状态转换的分区，记录错误日志
11    val (validPartitions, invalidPartitions) = controllerContext.checkValidPartitions(partitions, targetState)
12    invalidPartitions.foreach(partition => logInvalidTransition(partition, targetState))
13    // 根据targetState进入到不同的case分支
14    targetState match {
15        .....
16    }
17 }
18


```

这个方法首先会做状态初始化的工作，具体来说就是，在方法调用时，不在元数据缓存中的所有分区的状态，会被初始化为 `NonExistentPartition`。

接着，检查哪些分区执行的状态转换不合法，并为这些分区记录相应的错误日志。

之后，代码携合法状态转换的分区列表进入到 case 分支。由于分区状态只有 4 个，因此，它的 case 分支代码远比 ReplicaStateMachine 中的简单，而且，只有 OnlinePartition 这一路的分支逻辑相对复杂，其他 3 路仅仅是将分区状态设置成目标状态而已，

所以，我们来深入研究下目标状态是 OnlinePartition 的分支：

 复制代码

```
1 case OnlinePartition =>
2   // 获取未初始化分区列表，也就是NewPartition状态下的所有分区
3   val uninitializedPartitions = validPartitions.filter(
4     partition => partitionState(partition) == NewPartition)
5   // 获取具备Leader选举资格的分区列表
6   // 只能为OnlinePartition和OfflinePartition状态的分区选举Leader
7   val partitionsToElectLeader = validPartitions.filter(
8     partition => partitionState(partition) == OfflinePartition ||
9     partitionState(partition) == OnlinePartition)
10  // 初始化NewPartition状态分区，在ZooKeeper中写入Leader和ISR数据
11  if (uninitializedPartitions.nonEmpty) {
12    val successfulInitializations = initializeLeaderAndIsrForPartitions(uninit
13    successfulInitializations.foreach { partition =>
14      stateChangeLog.info(s"Changed partition $partition from ${partitionState
15      s"${controllerContext.partitionLeadershipInfo(partition).leaderAndIsr}
16      controllerContext.putPartitionState(partition, OnlinePartition)
17    }
18  }
19  // 为具备Leader选举资格的分区推选Leader
20  if (partitionsToElectLeader.nonEmpty) {
21    val electionResults = electLeaderForPartitions(
22      partitionsToElectLeader,
23      partitionLeaderElectionStrategyOpt.getOrElse(
24        throw new IllegalArgumentException("Election strategy is a required fi
25      )
26    )
27    electionResults.foreach {
28      case (partition, Right(leaderAndIsr)) =>
29        stateChangeLog.info(
30          s"Changed partition $partition from ${partitionState(partition)} to :
31        )
32        // 将成功选举Leader后的分区设置成OnlinePartition状态
33        controllerContext.putPartitionState(
34          partition, OnlinePartition)
35      case (_, Left(_)) => // 如果选举失败，忽略之
36    }
37    // 返回Leader选举结果
38    electionResults
39  } else {
40    Map.empty
41  }
```

虽然代码有点长，但总的步骤就两步。

第 1 步是为 NewPartition 状态的分区做初始化操作，也就是在 ZooKeeper 中，创建并写入分区节点数据。节点的位置

是/brokers/topics/<topic>/partitions/<partition>，每个节点都要包含分区的 Leader 和 ISR 等数据。而 **Leader 和 ISR 的确定规则是：选择存活副本列表的第一个副本作为 Leader；选择存活副本列表作为 ISR。**至于具体的代码，可以看下

initializeLeaderAndIsrForPartitions 方法代码片段的倒数第 5 行：

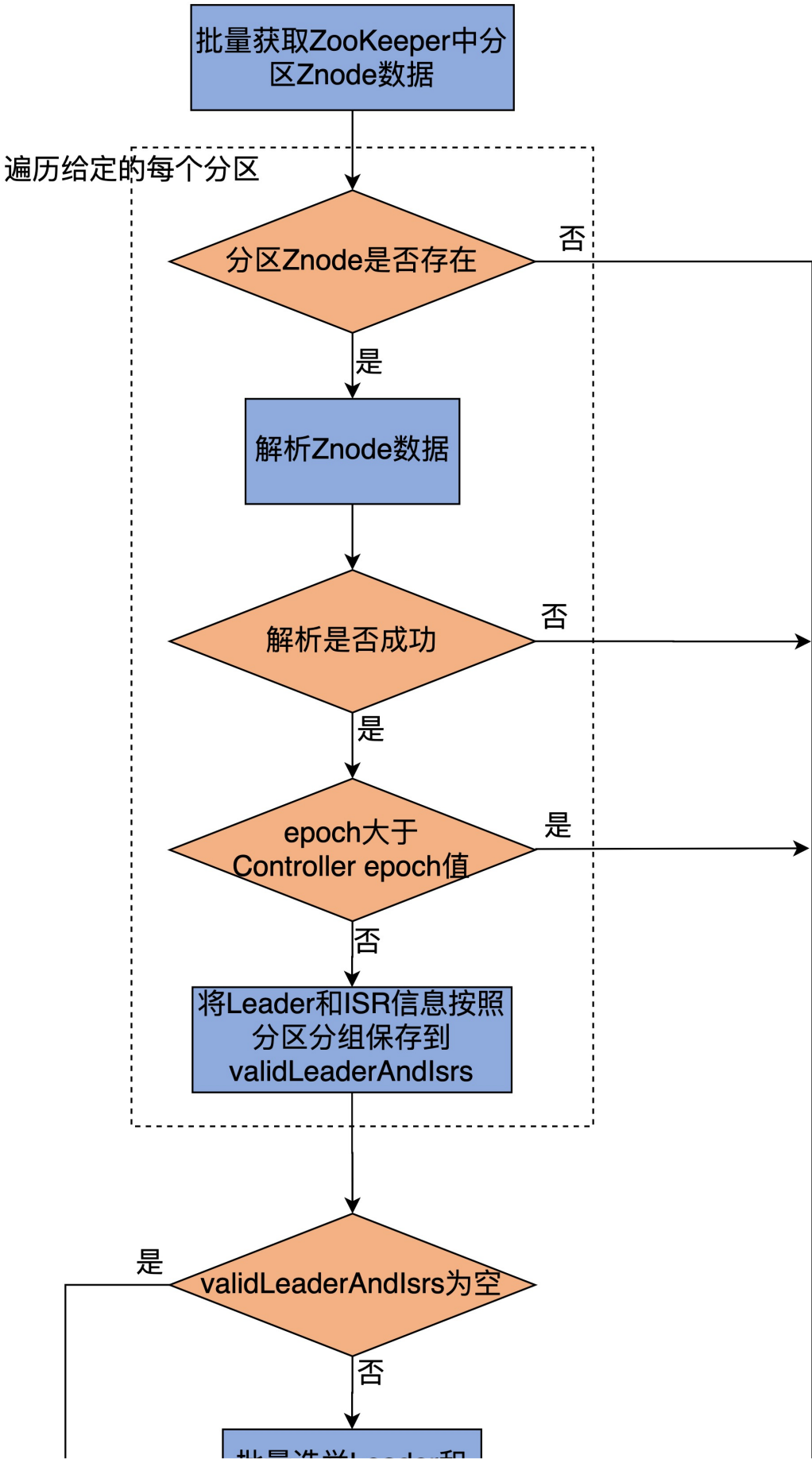
 复制代码

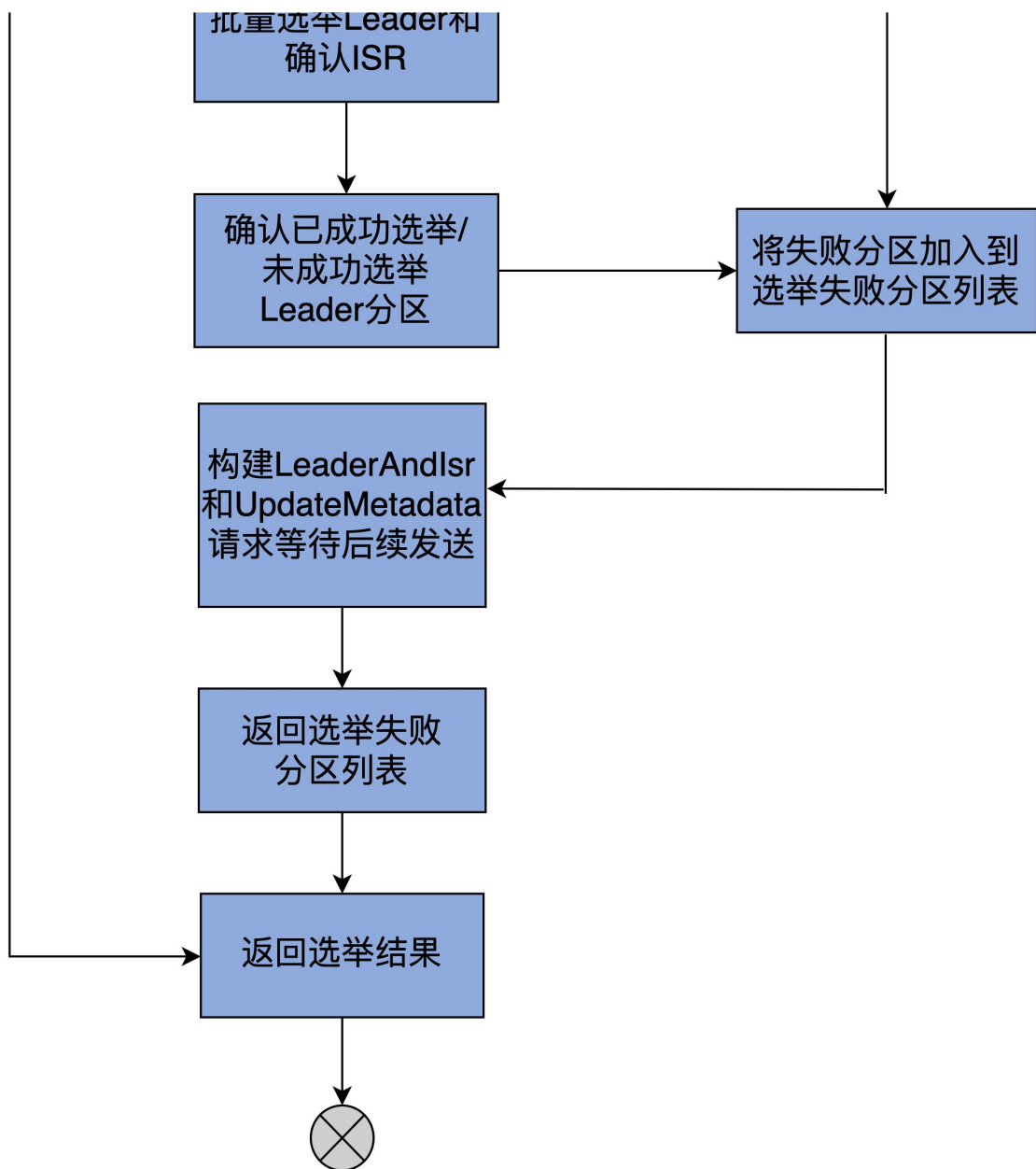
```
1 private def initializeLeaderAndIsrForPartitions(partitions: Seq[TopicPartition]
2     .....
3     // 获取每个分区的副本列表
4     val replicasPerPartition = partitions.map(partition => partition -> contro
5     // 获取每个分区的所有存活副本
6     val liveReplicasPerPartition = replicasPerPartition.map { case (partition,
7         val liveReplicasForPartition = replicas.filter(replica => controllerCo
8         partition -> liveReplicasForPartition
9     }
10    // 按照有无存活副本对分区进行分组
11    // 分为两组：有存活副本的分区；无任何存活副本的分区
12    val (partitionsWithoutLiveReplicas, partitionsWithLiveReplicas) = liveRepl
13    .....
14    // 为"有存活副本的分区"确定Leader和ISR
15    // Leader确认依据：存活副本列表的首个副本被认定为Leader
16    // ISR确认依据：存活副本列表被认定为ISR
17    val leaderIsrAndControllerEpochs = partitionsWithLiveReplicas.map { case (
18        val leaderAndIsr = LeaderAndIsr(liveReplicas.head, liveReplicas.toList)
19        .....
20    }.toMap
21    .....
22 }
```

第 2 步是为具备 Leader 选举资格的分区推选 Leader，代码调用 electLeaderForPartitions 方法实现。这个方法会不断尝试为多个分区选举 Leader，直到所有分区都成功选出 Leader。

选举 Leader 的核心代码位于 doElectLeaderForPartitions 方法中，该方法主要有 3 步。

代码很长，我先画一张图来展示它的主要步骤，然后再分步骤给你解释每一步的代码，以免你直接陷入冗长的源码行里面去。





看着好像图也很长，别着急，我们来一步步拆解下。

就像前面说的，这个方法大体分为 3 步。第 1 步是从 ZooKeeper 中获取给定分区的 Leader、ISR 信息，并将结果封装进名为 `validLeaderAndIsrs` 的容器中，代码如下：

```
1 // doElectLeaderForPartitions方法的第1部分
2 val getDataResponses = try {
3     // 批量获取ZooKeeper中给定分区的znode节点数据
4     zkClient.getTopicPartitionStatesRaw(partitions)
5 } catch {
6     case e: Exception =>
7         return (partitions.iterator.map(_ -> Left(e)).toMap, Seq.empty)
8 }
```

 复制代码

```

9 // 构建两个容器，分别保存可选举Leader分区列表和选举失败分区列表
10 val failedElections = mutable.Map.empty[TopicPartition, Either[Exception, LeaderAndIsr]]
11 val validLeaderAndIsrs = mutable.Buffer.empty[(TopicPartition, LeaderAndIsr)]
12 // 遍历每个分区的znode节点数据
13 getDataResponses.foreach { getDataResponse =>
14     val partition = getDataResponse.ctx.get.asInstanceOf[TopicPartition]
15     val currState = partitionState(partition)
16     // 如果成功拿到znode节点数据
17     if (getDataResponse.resultCode == Code.OK) {
18         TopicPartitionStateZNode.decode(getDataResponse.data, getDataResponse.stat
19         // 节点数据中含Leader和ISR信息
20         case Some(leaderIsrAndControllerEpoch) =>
21             // 如果节点数据的Controller Epoch值大于当前Controller Epoch值
22             if (leaderIsrAndControllerEpoch.controllerEpoch > controllerContext.ep
23                 val failMsg = s"Aborted leader election for partition $partition sin
24                 s"already written by another controller. This probably means that
25                 s"a soft failure and another controller was elected with epoch ${l
26             // 将该分区加入到选举失败分区列表
27             failedElections.put(partition, Left(new StateChangeFailedException(f
28         } else {
29             // 将该分区加入到可选举Leader分区列表
30             validLeaderAndIsrs += partition -> leaderIsrAndControllerEpoch.leade
31         }
32         // 如果节点数据不含Leader和ISR信息
33         case None =>
34             val exception = new StateChangeFailedException(s"LeaderAndIsr informat
35             // 将该分区加入到选举失败分区列表
36             failedElections.put(partition, Left(exception))
37     }
38     // 如果没有拿到znode节点数据，则将该分区加入到选举失败分区列表
39 } else if (getDataResponse.resultCode == Code.NONODE) {
40     val exception = new StateChangeFailedException(s"LeaderAndIsr information i
41     failedElections.put(partition, Left(exception))
42 } else {
43     failedElections.put(partition, Left(getDataResponse.resultException.get))
44 }
45 }
46
47 if (validLeaderAndIsrs.isEmpty) {
48     return (failedElections.toMap, Seq.empty)
49 }
50

```

首先，代码会批量读取 ZooKeeper 中给定分区的所有 Znode 数据。之后，会构建两个容器，分别保存可选举 Leader 分区列表和选举失败分区列表。接着，开始遍历每个分区的 Znode 节点数据，如果成功拿到 Znode 节点数据，节点数据包含 Leader 和 ISR 信息且节点数据的 Controller Epoch 值小于当前 Controller Epoch 值，那么，就将该分区加入到可选举 Leader 分区列表。倘若发现 Zookeeper 中保存的 Controller Epoch 值大于当前

Epoch 值，说明该分区已经被一个更新的 Controller 选举过 Leader 了，此时必须终止本次 Leader 选举，并将该分区放置到选举失败分区列表中。

遍历完这些分区之后，代码要看下 validLeaderAndIsrs 容器中是否包含可选举 Leader 的分区。如果一个满足选举 Leader 的分区都没有，方法直接返回。至此，doElectLeaderForPartitions 方法的第一大步完成。

下面，我们看下该方法的第 2 部分代码：


 复制代码

```
1 // doElectLeaderForPartitions方法的第2部分
2 // 开始选举Leader，并根据有无Leader将分区进行分区
3 val (partitionsWithoutLeaders, partitionsWithLeaders) =
4     partitionLeaderElectionStrategy match {
5         case OfflinePartitionLeaderElectionStrategy(allowUnclean) =>
6             val partitionsWithUncleanLeaderElectionState = collectUncleanLeaderElectionState(
7                 validLeaderAndIsrs,
8                 allowUnclean
9             )
10            // 为OfflinePartition分区选举Leader
11            leaderForOffline(controllerContext, partitionsWithUncleanLeaderElectionState)
12        case ReassignPartitionLeaderElectionStrategy =>
13            // 为副本重分配的分区分区选举Leader
14            leaderForReassign(controllerContext, validLeaderAndIsrs).partition(_.leaderIsReassigned)
15        case PreferredReplicaPartitionLeaderElectionStrategy =>
16            // 为分区执行Preferred副本Leader选举
17            leaderForPreferredReplica(controllerContext, validLeaderAndIsrs).partition(_.leaderIsPreferred)
18        case ControlledShutdownPartitionLeaderElectionStrategy =>
19            // 为因Broker正常关闭而受影响的分区分区选举Leader
20            leaderForControlledShutdown(controllerContext, validLeaderAndIsrs).partition(_.leaderIsControlledShutdown)
21    }
22
```

这一步是根据给定的 PartitionLeaderElectionStrategy，调用 PartitionLeaderElectionAlgorithms 的不同方法执行 Leader 选举，同时，区分出成功选举 Leader 和未选出 Leader 的分区。

前面说过了，这 4 种不同的策略定义了 4 个专属的方法来进行 Leader 选举。其实，如果你打开这些方法的源码，就会发现它们大同小异。基本上，选择 Leader 的规则，就是选择副本集合中首个存活且处于 ISR 中的副本作为 Leader。

现在，我们再来看这个方法的最后一部分代码，这一步主要是更新 ZooKeeper 节点数据，以及 Controller 端元数据缓存信息。

 复制代码

```
1 // doElectLeaderForPartitions方法的第3部分
2 // 将所有选举失败的分区全部加入到Leader选举失败分区列表
3 partitionsWithoutLeaders.foreach { electionResult =>
4     val partition = electionResult.topicPartition
5     val failMsg = s"Failed to elect leader for partition $partition under strate
6     failedElections.put(partition, Left(new StateChangeFailedException(failMsg))
7 }
8 val recipientsPerPartition = partitionsWithLeaders.map(result => result.topicPa
9 val adjustedLeaderAndIsrs = partitionsWithLeaders.map(result => result.topicPa
10 // 使用新选举的Leader和ISR信息更新ZooKeeper上分区的znode节点数据
11 val UpdateLeaderAndIsrResult(finishedUpdates, updatesToRetry) = zkClient.updat
12     adjustedLeaderAndIsrs, controllerContext.epoch, controllerContext.epochZkVer
13 // 对于ZooKeeper znode节点数据更新成功的分区，封装对应的Leader和ISR信息
14 // 构建LeaderAndIsr请求，并将该请求加入到Controller待发送请求集合
15 // 等待后续统一发送
16 finishedUpdates.foreach { case (partition, result) =>
17     result.foreach { leaderAndIsr =>
18         val replicaAssignment = controllerContext.partitionFullReplicaAssignment(p
19         val leaderIsrAndControllerEpoch = LeaderIsrAndControllerEpoch(leaderAndIsr
20         controllerContext.partitionLeadershipInfo.put(partition, leaderIsrAndContri
21         controllerBrokerRequestBatch.addLeaderAndIsrRequestForBrokers(recipientsPe
22             leaderIsrAndControllerEpoch, replicaAssignment, isNew = false)
23     }
24 }
25 // 返回选举结果，包括成功选举并更新ZooKeeper节点的分区、选举失败分区以及
26 // ZooKeeper节点更新失败的分区
27 (finishedUpdates ++ failedElections, updatesToRetry)
```

首先，将上一步中所有选举失败的分区，全部加入到 Leader 选举失败分区列表。

然后，使用新选举的 Leader 和 ISR 信息，更新 ZooKeeper 上分区的 Znode 节点数据。对于 ZooKeeper Znode 节点数据更新成功的那些分区，源码会封装对应的 Leader 和 ISR 信息，构建 LeaderAndIsr 请求，并将该请求加入到 Controller 待发送请求集合，等待后续统一发送。

最后，方法返回选举结果，包括成功选举并更新 ZooKeeper 节点的分区列表、选举失败分区列表，以及 ZooKeeper 节点更新失败的分区列表。

这会儿，你还记得 `handleStateChanges` 方法的第 2 步是 Controller 给相关的 Broker 发送请求吗？那么，到底要给哪些 Broker 发送哪些请求呢？其实就是在上面这步完成的，即这行语句：

 复制代码

```
1 controllerBrokerRequestBatch.addLeaderAndIsrRequestForBrokers(  
2     recipientsPerPartition(partition), partition,  
3     leaderIsrAndControllerEpoch, replicaAssignment, isNew = false)
```

总结

今天，我们重点学习了 `PartitionStateMachine.scala` 文件的源码，主要是研究了 Kafka 分区状态机的构造原理和工作机制。

学到这里，我们再来回答开篇面试官的问题，应该就不是什么难事了。现在我们知道了，Kafka 目前提供 4 种 Leader 选举策略，分别是分区下线后的 Leader 选举、分区执行副本重分配时的 Leader 选举、分区执行 Preferred 副本 Leader 选举，以及 Broker 下线时的分区 Leader 选举。

这 4 类选举策略在选择 Leader 这件事情上有着类似的逻辑，那就是，它们几乎都是选择当前副本有序集合中的、首个处于 ISR 集合中的存活副本作为新的 Leader。当然，个别选举策略可能会有细小的差别，你可以结合我们今天学到的源码，课下再深入地研究一下每一类策略的源码。

我们来回顾下这节课的重点。

`PartitionStateMachine` 是 Kafka Controller 端定义的分区状态机，负责定义、维护和管理合法的分区状态转换。

每个 Broker 启动时都会实例化一个分区状态机对象，但只有 Controller 所在的 Broker 才会启动它。

Kafka 分区有 4 类状态，分别是 `NewPartition`、`OnlinePartition`、`OfflinePartition` 和 `NonExistentPartition`。其中 `OnlinePartition` 是分区正常工作时的状态。`NewPartition` 是未初始化状态，处于该状态下的分区尚不具备选举 Leader 的资格。

Leader 选举有 4 类场景，分别是 Offline、Reassign、Prefer Leader Election 和 ControlledShutdown。每类场景都对应于一种特定的 Leader 选举策略。

handleStateChanges 方法是主要的入口方法，下面调用 doHandleStateChanges 私有方法实现实际的 Leader 选举功能。

下个模块，我们将来到 Kafka 延迟操作代码的世界。在那里，你能了解 Kafka 是如何实现一个延迟请求的处理的。另外，一个 $O(N)$ 时间复杂度的时间轮算法也等候在那里，到时候我们一起研究下它！

课后讨论

源码中有个 triggerOnlineStateChangeForPartitions 方法，请你分析下，它是做什么用的，以及它何时被调用？

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。

更多课程推荐

MySQL 实战 45 讲

从原理到实战，丁奇带你搞懂 MySQL

林晓斌

网名丁奇
前阿里资深技术专家



涨价倒计时 🕒

今日秒杀 **¥79**，6月13日涨价至 **¥129**

© 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 (2)

写留言



胡夕 置顶

2020-06-05

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，咱们结合源码重点了解了ReplicaStateMachine的源码。课后我请你自行分析doHandleStateChanges方法中最后一路分支的代码。下面我给出我的答案：最后一路case分支是NonExistentReplica。当进入到这路分支后，代码会遍历所有能够执行状态转换...
展开



Geek_f406a1

2020-06-06

kafka集群间分区级数据复制如何做？

展开

作者回复: 通过MirrorMaker工具，虽然不太好用。社区2.4推出了MirrorMaker2，只是不知道目前的效果如何。其他公司也有开源产品，如Uber的uReplicator



1