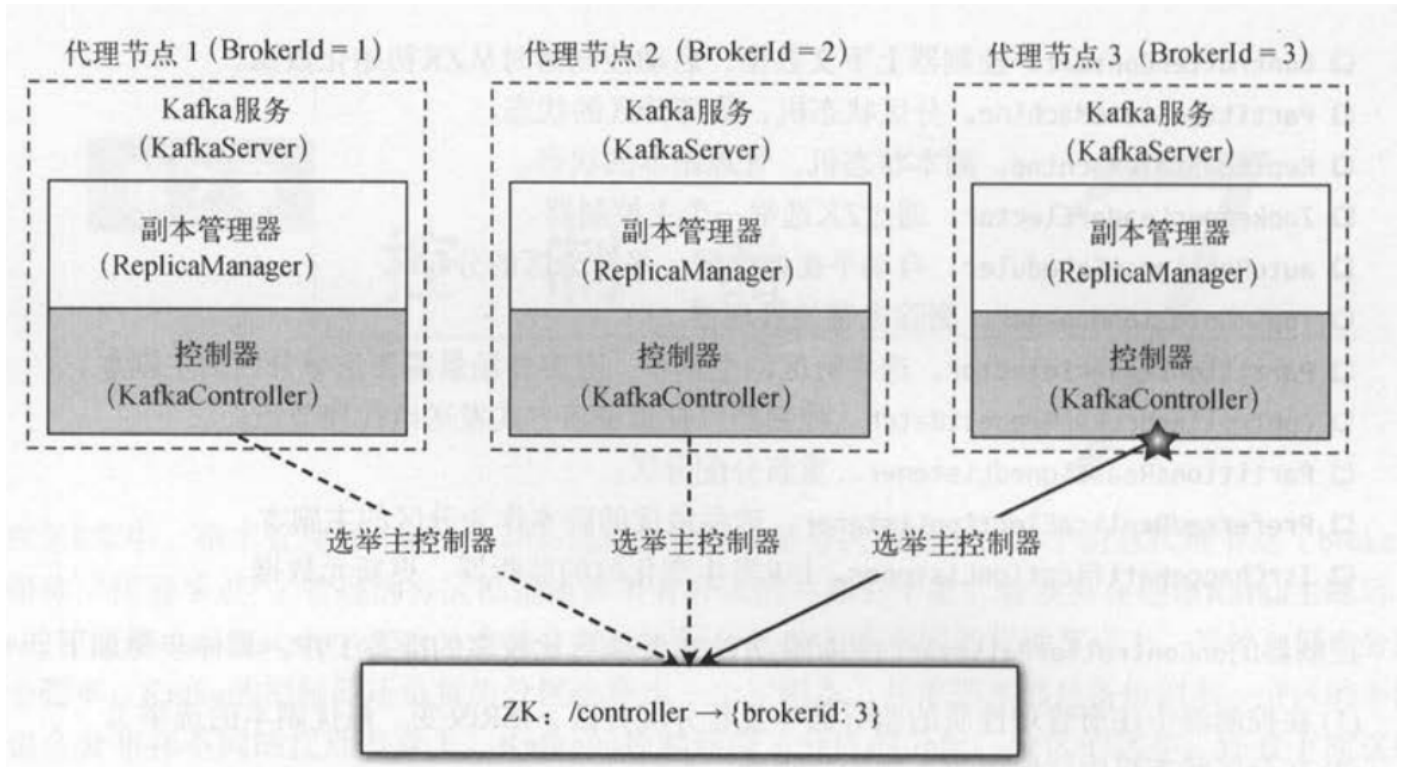# Kafka_Controller

---

Kafka_Controller作为controller模块中的核心组件，对整个Kafka控制器起着总管作用。kafka_controller由外界进行启动和关闭，主要功能有进行状态机和控制器上下文的初始化，并负责想不通的代理节点发送信息。kafka_controller通过调用其他类的方法来完成上述功能，并且注册监听器，将controllerEvent写进阻塞队列中供eventManager实例进行处理。下面通过对kafka_controller的重要函数来说明kafka_controller的主要功能

类图：

kafka控制器选举流程：

代理节点 1（BrokerId = 1）　　　　代理节点 2（BrokerId = 2）　　　　代理节点 3（BrokerId = 3）

Kafka服务 (KafkaServer) — 副本管理器 (ReplicaManager) — 控制器 (KafkaController)

选举主控制器　　　选举主控制器　　　选举主控制器

ZK：/controller → {brokerid: 3}
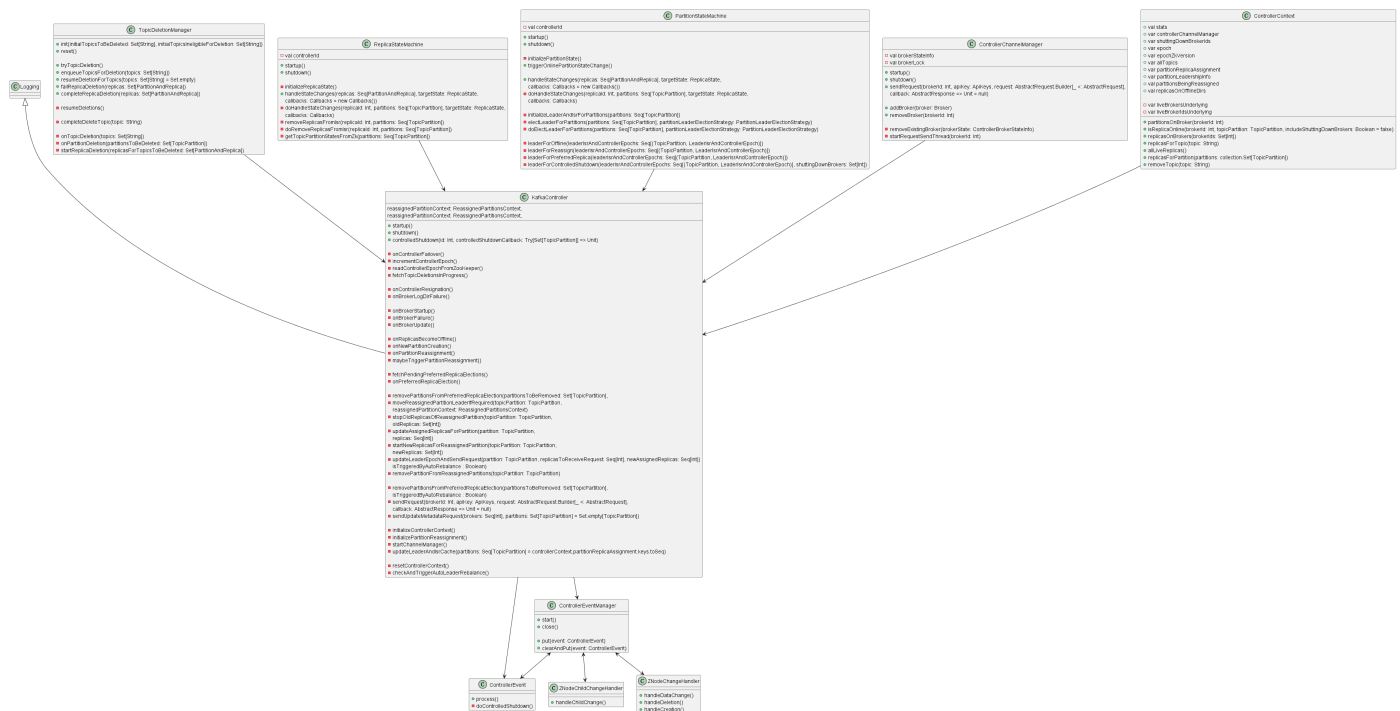
startUp()

```scala
def startup() = {
    zkClient.registerStateChangeHandler(new StateChangeHandler {
      override val name: String = StateChangeHandlers.ControllerHandler
      override def afterInitializingSession(): Unit = {
        eventManager.put(RegisterBrokerAndReelect)
      }
      override def beforeInitializingSession(): Unit = {
        val expireEvent = new Expire
        eventManager.clearAndPut(expireEvent)

        // Block initialization of the new session until the expiration event is
being handled,
        // which ensures that all pending events have been processed before
creating the new session
        expireEvent.waitUntilProcessingStarted()
      }
    })
    eventManager.put(Startup)
    eventManager.start()
  }
```

startUp函数的主要功能时注册registerStateChangeHandler监听器，并且对
eventManager类中的阻塞队列进行清空，方便之后controllerEventmanager事件发生时
成功写进阻塞队列并且由eventManager实例进行处理

Shutdown()

```scala
def shutdown() = {
    eventManager.close()
    onControllerResignation()
  }
```

Shutdown函数的主要功能是关闭eventManager实例

onControllerFailover()

```scala
private def onControllerFailover() {
    info("Reading controller epoch from ZooKeeper")
    readControllerEpochFromZooKeeper()
    info("Incrementing controller epoch in ZooKeeper")
    incrementControllerEpoch()
    info("Registering handlers")

    // before reading source of truth from zookeeper, register the listeners to get
broker/topic callbacks
    val childChangeHandlers = Seq(brokerChangeHandler, topicChangeHandler,
topicDeletionHandler, logDirEventNotificationHandler,
      isrChangeNotificationHandler)
    childChangeHandlers.foreach(zkClient.registerZNodeChildChangeHandler)
    val nodeChangeHandlers = Seq(preferredReplicaElectionHandler,
partitionReassignmentHandler)

nodeChangeHandlers.foreach(zkClient.registerZNodeChangeHandlerAndCheckExistence)

    info("Deleting log dir event notifications")
    zkClient.deleteLogDirEventNotifications()
    info("Deleting isr change notifications")
    zkClient.deleteIsrChangeNotifications()
    info("Initializing controller context")
    initializeControllerContext()
    info("Fetching topic deletions in progress")
    val (topicsToBeDeleted, topicsIneligibleForDeletion) =
fetchTopicDeletionsInProgress()
    info("Initializing topic deletion manager")
    topicDeletionManager.init(topicsToBeDeleted, topicsIneligibleForDeletion)

    // We need to send UpdateMetadataRequest after the controller context is
initialized and before the state machines
    // are started. The is because brokers need to receive the list of live brokers
from UpdateMetadataRequest before
    // they can process the LeaderAndIsrRequests that are generated by
replicaStateMachine.startup() and
    // partitionStateMachine.startup().
    info("Sending update metadata request")
    sendUpdateMetadataRequest(controllerContext.liveOrShuttingDownBrokerIds.toSeq)

    replicaStateMachine.startup()
    partitionStateMachine.startup()
```

```
        info(s"Ready to serve as the new controller with epoch $epoch")

  maybeTriggerPartitionReassignment(controllerContext.partitionsBeingReassigned.keySe
t)
    topicDeletionManager.tryTopicDeletion()
    val pendingPreferredReplicaElections = fetchPendingPreferredReplicaElections()
    onPreferredReplicaElection(pendingPreferredReplicaElections)
    info("Starting the controller scheduler")
    kafkaScheduler.startup()
    if (config.autoLeaderRebalanceEnable) {
      scheduleAutoLeaderRebalanceTask(delay = 5, unit = TimeUnit.SECONDS)
    }

    if (config.tokenAuthEnabled) {
      info("starting the token expiry check scheduler")
      tokenCleanScheduler.startup()
      tokenCleanScheduler.schedule(name = "delete-expired-tokens",
        fun = tokenManager.expireTokens,
        period = config.delegationTokenExpiryCheckIntervalMs,
        unit = TimeUnit.MILLISECONDS)
    }
  }
```

onControllerFailover()首先对zk节点的Epoch进行更新，然后initializeControllerContext()
更新上下文。然后分别对实例topicDeletionManager()，replicaStateMachine，
partitionStateMachine进行初始化和启动，并在中间发送元数据个各个broker。最终调用

- maybeTriggerPartitionReassignment(controllerContext.partitionsBeingReassigned
  .keySet)，topicDeletionManager.tryTopicDeletion()进行可能之前节点下线时出现
  问题而没有进行的关于主题的删除以及分区的重新分配
- onPreferredReplicaElection(pendingPreferredReplicaElections)进行分区领导的选
  举

与onControllerFailover()相对的函数为onControllerResignation()，负责再当前节点没有
成功选举为控制器的情况下，对之前注册的监听器，启动或初始化的
topicDeletionManager，partitionStateMachine，replicaStateMachine进行关闭，并重置
上下文

onBrokerStartup(newBrokers: Seq[Int])

```
  private def onBrokerStartup(newBrokers: Seq[Int]) {
    info(s"New broker startup callback for ${newBrokers.mkString(",")}")
    newBrokers.foreach(controllerContext.replicasOnOfflineDirs.remove)
    val newBrokersSet = newBrokers.toSet
    // send update metadata request to all live and shutting down brokers. Old
brokers will get to know of the new
    // broker via this update.
    // In cases of controlled shutdown leaders will not be elected when a new
```

```
broker comes up. So at least in the
    // common controlled shutdown case, the metadata will reach the new brokers
faster
    sendUpdateMetadataRequest(controllerContext.liveOrShuttingDownBrokerIds.toSeq)
    // the very first thing to do when a new broker comes up is send it the entire
list of partitions that it is
    // supposed to host. Based on that the broker starts the high watermark threads
for the input list of partitions
    val allReplicasOnNewBrokers =
controllerContext.replicasOnBrokers(newBrokersSet)
    replicaStateMachine.handleStateChanges(allReplicasOnNewBrokers.toSeq,
OnlineReplica)
    // when a new broker comes up, the controller needs to trigger leader election
for all new and offline partitions
    // to see if these brokers can become leaders for some/all of those
    partitionStateMachine.triggerOnlinePartitionStateChange()
    // check if reassignment of some partitions need to be restarted
    val partitionsWithReplicasOnNewBrokers =
controllerContext.partitionsBeingReassigned.filter {
      case (_, reassignmentContext) =>
reassignmentContext.newReplicas.exists(newBrokersSet.contains)
    }
    partitionsWithReplicasOnNewBrokers.foreach { case (tp, context) =>
onPartitionReassignment(tp, context) }
    // check if topic deletion needs to be resumed. If at least one replica that
belongs to the topic being deleted exists
    // on the newly restarted brokers, there is a chance that topic deletion can
resume
    val replicasForTopicsToBeDeleted = allReplicasOnNewBrokers.filter(p =>
topicDeletionManager.isTopicQueuedUpForDeletion(p.topic))
    if (replicasForTopicsToBeDeleted.nonEmpty) {
      info(s"Some replicas ${replicasForTopicsToBeDeleted.mkString(",")} for topics
scheduled for deletion " +
        s"${topicDeletionManager.topicsToBeDeleted.mkString(",")} are on the newly
restarted brokers " +
        s"${newBrokers.mkString(",")}. Signaling restart of topic deletion for
these topics")

topicDeletionManager.resumeDeletionForTopics(replicasForTopicsToBeDeleted.map(_.top
ic))
    }
    registerBrokerModificationsHandler(newBrokers)
  }
```

- sendUpdateMetadataRequest()：上线新节点以后向新节点发送元数据

- replicaStateMachine.handleStateChanges(allReplicasOnNewBrokers.toSeq,
  OnlineReplica) partitionStateMachine.triggerOnlinePartitionStateChange()，对于
  新上线的节点而言可能会涉及到分区状态机和副本状态机的修改，因此调用两个状
  态机的函数进行此动作

- 对于新上线的节点上的副本，可能会涉及到删除，因此调用topicDeletionManager
  里的resumeDeletionForTopics()对需要进行删除的副本进行删除。删除之后在zk节
  点上注册registerBrokerModificationsHandler监听器

与onBrokerStartup(newBrokers: Seq[Int])相对应的，对代理节点进行操作的函数有onBrokerFailure(deadBrokers: Seq[Int])与onBrokerUpdate(updatedBrokerId: Int)。Failure函数负责将下线节点上的所有副本转为offline状态，并注销registerBrokerModificationsHandler监听器；onBrokerUpdate()函数负责给代理节点发送元数据并进行更新

以上说的onReplicasBecomeOffline()负责调用分区状态机和副本状态机的handlestatechanges()函数将分区和副本均改为下线状态

onPartitionReassignment(topicPartition: TopicPartition, reassignedPartitionContext: ReassignedPartitionsContext)

这是kafka_controller类中最为重要的函数之一，它负责在有新的副本上线时，进行分区的重新分配。并且它会被多次调用。每次被调用时，由于要考虑到当前的ISR与AR是否都包含了OAR与RAR，因此只有当上述条件满足时，才会进行第二步操作，否则条件判断将一直为false

```scala
private def onPartitionReassignment(topicPartition: TopicPartition,
reassignedPartitionContext: ReassignedPartitionsContext) {
    val reassignedReplicas = reassignedPartitionContext.newReplicas
    if (!areReplicasInIsr(topicPartition, reassignedReplicas)) {
      info(s"New replicas ${reassignedReplicas.mkString(",")} for partition
$topicPartition being reassigned not yet " +
        "caught up with the leader")
      val newReplicasNotInOldReplicaList = reassignedReplicas.toSet --
controllerContext.partitionReplicaAssignment(topicPartition).toSet
      val newAndOldReplicas = (reassignedPartitionContext.newReplicas ++
controllerContext.partitionReplicaAssignment(topicPartition)).toSet
      //1. Update AR in ZK with OAR + RAR.
      updateAssignedReplicasForPartition(topicPartition, newAndOldReplicas.toSeq)
      //2. Send LeaderAndIsr request to every replica in OAR + RAR (with AR as OAR
+ RAR).
      updateLeaderEpochAndSendRequest(topicPartition,
controllerContext.partitionReplicaAssignment(topicPartition),
        newAndOldReplicas.toSeq)
      //3. replicas in RAR - OAR -> NewReplica
      startNewReplicasForReassignedPartition(topicPartition,
reassignedPartitionContext, newReplicasNotInOldReplicaList)
      info(s"Waiting for new replicas ${reassignedReplicas.mkString(",")} for
partition ${topicPartition} being " +
        "reassigned to catch up with the leader")
    } else {
      //4. Wait until all replicas in RAR are in sync with the leader.
      val oldReplicas =
controllerContext.partitionReplicaAssignment(topicPartition).toSet --
reassignedReplicas.toSet
      //5. replicas in RAR -> OnlineReplica
      reassignedReplicas.foreach { replica =>
        replicaStateMachine.handleStateChanges(Seq(new
```

```
PartitionAndReplica(topicPartition, replica)), OnlineReplica)
      }
      //6. Set AR to RAR in memory.
      //7. Send LeaderAndIsr request with a potential new leader (if current leader
not in RAR) and
      //   a new AR (using RAR) and same isr to every broker in RAR
      moveReassignedPartitionLeaderIfRequired(topicPartition,
reassignedPartitionContext)
      //8. replicas in OAR - RAR -> Offline (force those replicas out of isr)
      //9. replicas in OAR - RAR -> NonExistentReplica (force those replicas to be
deleted)
      stopOldReplicasOfReassignedPartition(topicPartition,
reassignedPartitionContext, oldReplicas)
      //10. Update AR in ZK with RAR.
      updateAssignedReplicasForPartition(topicPartition, reassignedReplicas)
      //11. Update the /admin/reassign_partitions path in ZK to remove this
partition.
      removePartitionFromReassignedPartitions(topicPartition)
      //12. After electing leader, the replicas and isr information changes, so
resend the update metadata request to every broker

sendUpdateMetadataRequest(controllerContext.liveOrShuttingDownBrokerIds.toSeq,
Set(topicPartition))
      // signal delete topic thread if reassignment for some partitions belonging
to topics being deleted just completed
      topicDeletionManager.resumeDeletionForTopics(Set(topicPartition.topic))
    }
```

在上述函数关于分区的重新分配中，可以大致分为以下五个步骤：

1. 将RAR加入AR，此时AR=OAR+RAR
2. 将RAR加入ISR，此时ISR=OAR+RAR
3. ISR不变，但分区的主副本从RAR中选举
4. 将OAR从ISR中移除，此时ISR=RAR
5. 将OAR从AR中移除，此时AR=RAR

关于onPartitionReassignment()为什么可能要执行两次的原因，因为
onPartitionReassignment()每次执行时关于allNewReplicas的判断，如果为假，则说明
还并没有将新建的所有副本加入进ISR中，因此需要在执行一次；当该条件判断为真
时，则说明第一步操作已经完成，此时再将之前的旧的副本分别从ISR和AR中移除就可
以了

关于该函数的调用时序图如下。
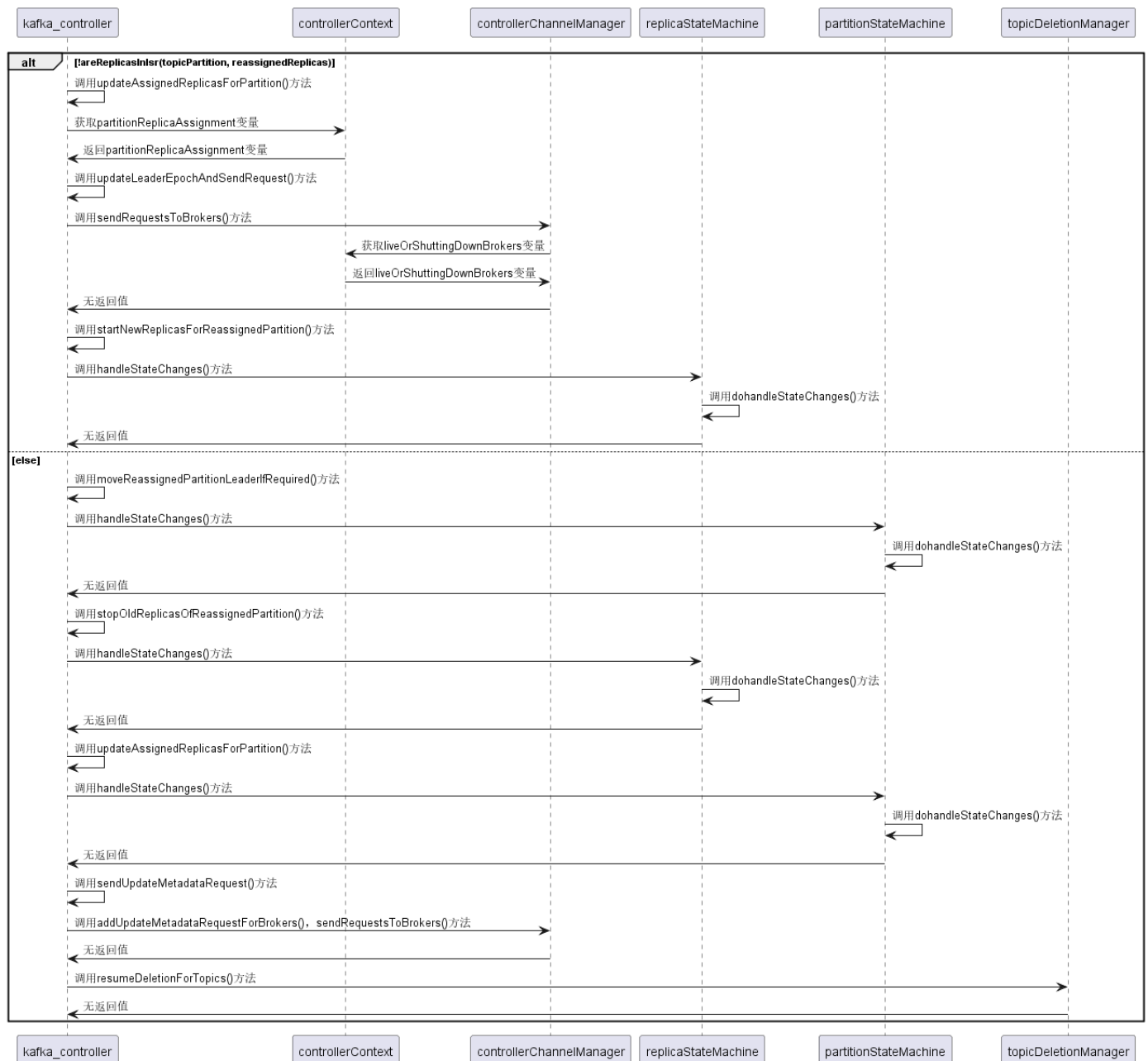
kafka_controller中其他的函数，比如resetControllerContext()更新上下文，fetchPendingPreferredReplicaElections()进行副本领导机制选举，在此不过多赘述。kafka_contoller中的方法，最终被contoller_event类的子类所用，即prossess()函数调用kafka_contoller中的函数(主要由由startUp类进行调用)

关于controller中的其他类，如TopicDeletionStopReplicaResponseReceived，Startup，BrokerChange，TopicChange它们均继承自controllerEvent类，并且会加入到controllerEvent队列中。其中的process()犯法多来自于kafkacontroller中的函数，同时其中也会有诸多监听器类，如TopicDeletionHandler，PartitionReassignmentHandler等用来监听controllerEvent的完成情况。