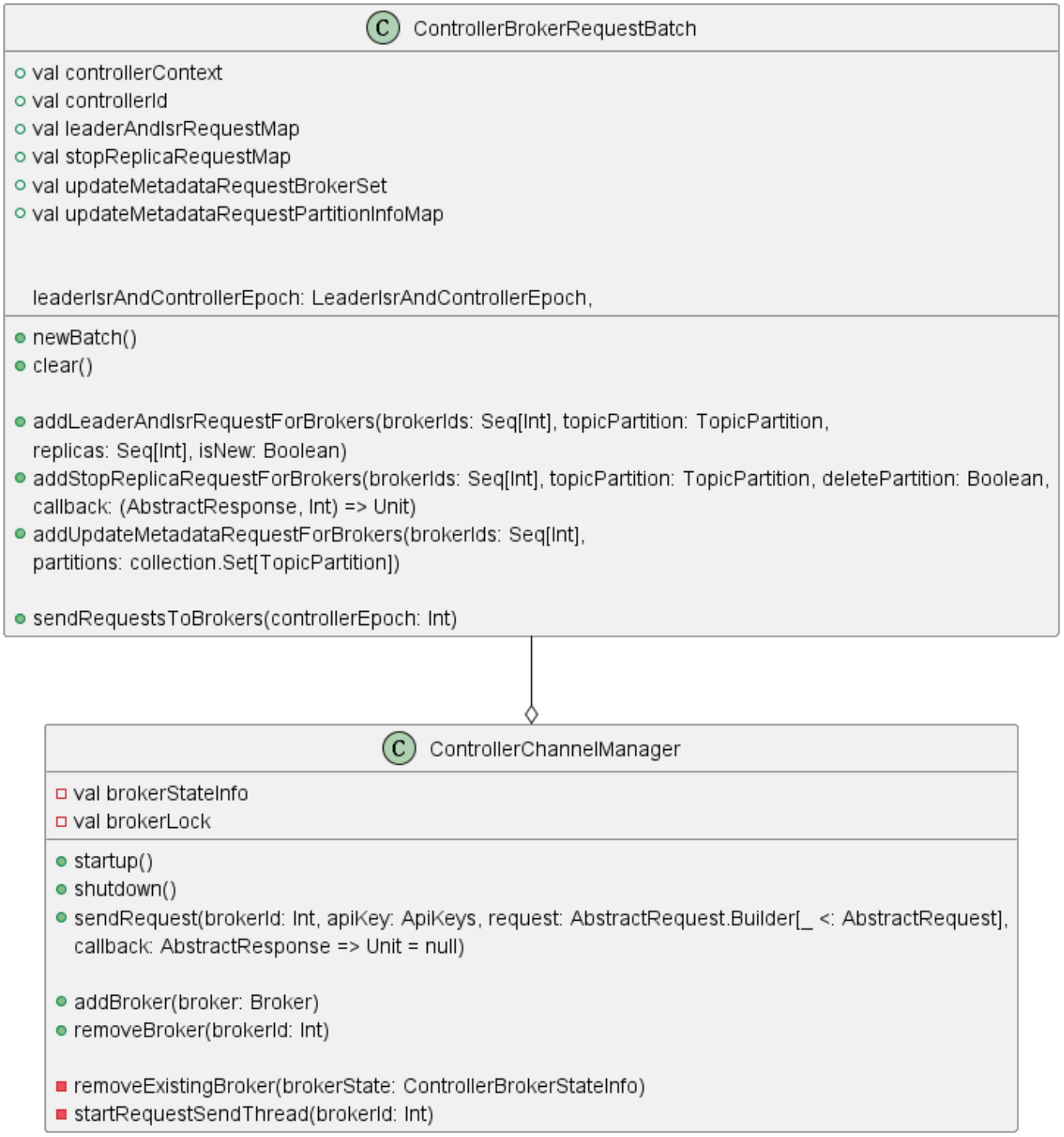
controllerChannelManager

controllerChannelManager是controller模块中负责建立和各个代理节点之间的通道，并将leaderAndIsr，StopReplica，UpdateMetadata等请求加入到队列中并进行发送。controllerChannelManager中有一个RequestSendThread类为发送请求线程，同时也有一个ControllerBrokerRequestBatch类来负责添加请求，并最终通过sendRequestsToBrokers()方法检查三类请求并进行发送。下面针对该类中的主要函数进行分析。

类图如下：



addNewBroker(broker: Broker)

```
private def addNewBroker(broker: Broker) {
    val messageQueue = new LinkedBlockingQueue[QueueItem]
    debug(s"Controller ${config.brokerId} trying to connect to broker
${broker.id}")
    val brokerNode = broker.node(config.interBrokerListenerName)
    val logContext = new LogContext(s"[Controller id=${config.brokerId},
targetBrokerId=${brokerNode.idString}] ")
    val networkClient = {
      val channelBuilder = ChannelBuilders.clientChannelBuilder(
        config.interBrokerSecurityProtocol,
        JaasContext.Type.SERVER,
        config,
```

```
        config.interBrokerListenerName,
        config.saslMechanismInterBrokerProtocol,
        config.saslInterBrokerHandshakeRequestEnable
      )
      val selector = new Selector(
        NetworkReceive.UNLIMITED,
        Selector.NO_IDLE_TIMEOUT_MS,
        metrics,
        time,
        "controller-channel",
        Map("broker-id" -> brokerNode.idString).asJava,
        false,
        channelBuilder,
        logContext
      )
      new NetworkClient(
        selector,
        new ManualMetadataUpdater(Seq(brokerNode).asJava),
        config.brokerId.toString,
        1,
        0,
        0,
        Selectable.USE_DEFAULT_BUFFER_SIZE,
        Selectable.USE_DEFAULT_BUFFER_SIZE,
        config.requestTimeoutMs,
        time,
        false,
        new ApiVersions,
        logContext
      )
    }
    val threadName = threadNamePrefix match {
      case None => s"Controller-${config.brokerId}-to-broker-${broker.id}-send-
thread"
      case Some(name) => s"$name:Controller-${config.brokerId}-to-
broker-${broker.id}-send-thread"
    }

    val requestThread = new RequestSendThread(config.brokerId, controllerContext,
messageQueue, networkClient,
      brokerNode, config, time, stateChangeLogger, threadName)
    requestThread.setDaemon(false)

    val queueSizeGauge = newGauge(
      QueueSizeMetricName,
      new Gauge[Int] {
        def value: Int = messageQueue.size
      },
      queueSizeTags(broker.id)
    )

    brokerStateInfo.put(broker.id, new ControllerBrokerStateInfo(networkClient,
brokerNode, messageQueue,
      requestThread, queueSizeGauge))
  }
```

addNewBroker(broker: Broker)函数的主要功能

1. 创建broker请求阻塞队列message_queue
2. 创建客户端networkClient并建立网络连接，并创建发送线程名
3. 创立请求发送线程requestSendThread

RequestSendThread.doWork()

```
override def doWork(): Unit = {

    def backoff(): Unit = pause(100, TimeUnit.MILLISECONDS)

    val QueueItem(apiKey, requestBuilder, callback) = queue.take()
    var clientResponse: ClientResponse = null
    try {
      var isSendSuccessful = false
      while (isRunning && !isSendSuccessful) {
        // if a broker goes down for a long time, then at some point the
controller's zookeeper listener will trigger a
        // removeBroker which will invoke shutdown() on this thread. At that point,
we will stop retrying.
        try {
          if (!brokerReady()) {
            isSendSuccessful = false
            backoff()
          }
          else {
            val clientRequest = networkClient.newClientRequest(brokerNode.idString,
requestBuilder,
              time.milliseconds(), true)
            clientResponse = NetworkClientUtils.sendAndReceive(networkClient,
clientRequest, time)
            isSendSuccessful = true
          }
        } catch {
          case e: Throwable => // if the send was not successful, reconnect to
broker and resend the message
            warn(s"Controller $controllerId epoch ${controllerContext.epoch} fails
to send request $requestBuilder " +
              s"to broker $brokerNode. Reconnecting to broker.", e)
            networkClient.close(brokerNode.idString)
            isSendSuccessful = false
            backoff()
        }
      }
      if (clientResponse != null) {
        val requestHeader = clientResponse.requestHeader
        val api = requestHeader.apiKey
        if (api != ApiKeys.LEADER_AND_ISR && api != ApiKeys.STOP_REPLICA && api !=
ApiKeys.UPDATE_METADATA)
          throw new KafkaException(s"Unexpected apiKey received: $apiKey")

        val response = clientResponse.responseBody
```

```
stateChangeLogger.withControllerEpoch(controllerContext.epoch).trace(s"Received
response " +
          s"${response.toString(requestHeader.apiVersion)} for request $api with
correlation id " +
          s"${requestHeader.correlationId} sent to broker $brokerNode")

        if (callback != null) {
          callback(response)
        }
      }
    } catch {
      case e: Throwable =>
        error(s"Controller $controllerId fails to send a request to broker
$brokerNode", e)
        // If there is any socket error (eg, socket timeout), the connection is no
longer usable and needs to be recreated.
        networkClient.close(brokerNode.idString)
    }
  }
```

1. 首先在请求发送线程中进行判断循环，须保证当前线程仍在运行并且发送没有成功再进行下一步操作(可能会出现zk节点的监听器触发shutdown()而使得线程关闭)
2. 若进入循环则首先判断接收请求的代理节点是否准备好(brokerready()函数判断，通过检查网络连接是否建立好networkClient来判断)；若已建立好，则创立clientRequest，据此创立clientResponse()接受回应，并将isSendSuccessful置为true；若没有建立好，则通过backoff()函数进行多次尝试，若尝试多次超时则关闭当前线程和网络
3. 在成功得到回应后，获得clientResponse的api和response，若api正确，则处理response回调函数，出现一行则关闭networkClient网络

sendRequestsToBrokers(controllerEpoch: Int)

```
def sendRequestsToBrokers(controllerEpoch: Int) {
    try {
      val stateChangeLog = stateChangeLogger.withControllerEpoch(controllerEpoch)

      val leaderAndIsrRequestVersion: Short =
        if (controller.config.interBrokerProtocolVersion >= KAFKA_1_0_IV0) 1
        else 0

      leaderAndIsrRequestMap.foreach {
        ...
      }
      leaderAndIsrRequestMap.clear()

      updateMetadataRequestPartitionInfoMap.foreach { case (tp, partitionState) =>
        stateChangeLog.trace(s"Sending UpdateMetadata request $partitionState to
```

```
brokers $updateMetadataRequestBrokerSet " +
          s"for partition $tp")
      }

      val partitionStates = Map.empty ++ updateMetadataRequestPartitionInfoMap
      val updateMetadataRequestVersion: Short =
        if (controller.config.interBrokerProtocolVersion >= KAFKA_1_0_IV0) 4
        else if (controller.config.interBrokerProtocolVersion >= KAFKA_0_10_2_IV0)
3
        else if (controller.config.interBrokerProtocolVersion >= KAFKA_0_10_0_IV1)
2
        else if (controller.config.interBrokerProtocolVersion >= KAFKA_0_9_0) 1
        else 0

      val updateMetadataRequest = {
        ...
      }

      updateMetadataRequestBrokerSet.foreach {
        ...
      }
      updateMetadataRequestBrokerSet.clear()
      updateMetadataRequestPartitionInfoMap.clear()

      stopReplicaRequestMap.foreach {
        ...
      }
      stopReplicaRequestMap.clear()
    } catch {
      case e: Throwable =>
        if (leaderAndIsrRequestMap.nonEmpty) {
          error("Haven't been able to send leader and isr requests, current state
of " +
              s"the map is $leaderAndIsrRequestMap. Exception message: $e")
        }
        if (updateMetadataRequestBrokerSet.nonEmpty) {
          error(s"Haven't been able to send metadata update requests to brokers
$updateMetadataRequestBrokerSet, " +
                s"current state of the partition info is
$updateMetadataRequestPartitionInfoMap. Exception message: $e")
        }
        if (stopReplicaRequestMap.nonEmpty) {
          error("Haven't been able to send stop replica requests, current state of
" +
              s"the map is $stopReplicaRequestMap. Exception message: $e")
        }
        throw new IllegalStateException(e)
    }
  }
}
```

sendRequestsToBrokers()的主要功能在于对于以上说的三类请求，leaderAndIsrRequestMap，updateMetadataRequest，stopReplicaRequest都通过该函

数发送给代理节点，并且每次发送之后，sendRequestsToBrokers()会清空三类请求的阻塞队列

以上就是网络通道管理器的主要函数以及其功能