

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

2020-05-16 胡夕

Kafka核心源码解读

[进入课程 >](#)



讲述：胡夕

时长 18:46 大小 17.20M



你好，我是胡夕。上节课，我们深入研究了 `ControllerContext.scala` 源码文件，掌握了 Kafka 集群定义的重要元数据。今天，我们来学习下 Controller 是如何给其他 Broker 发送请求的。

掌握了这部分实现原理，你就能更好地了解 Controller 究竟是如何与集群 Broker 进行交互，从而实现管理集群元数据的功能的。而且，阅读这部分源码，还能帮你定位和解决线上问题。我先跟你分享一个真实的案例。



当时还是在 Kafka 0.10.0.1 时代，我们突然发现，在线上环境中，很多元数据变更无法在集群的所有 Broker 上同步了。具体表现为，创建了主题后，有些 Broker 依然无法感知

到。

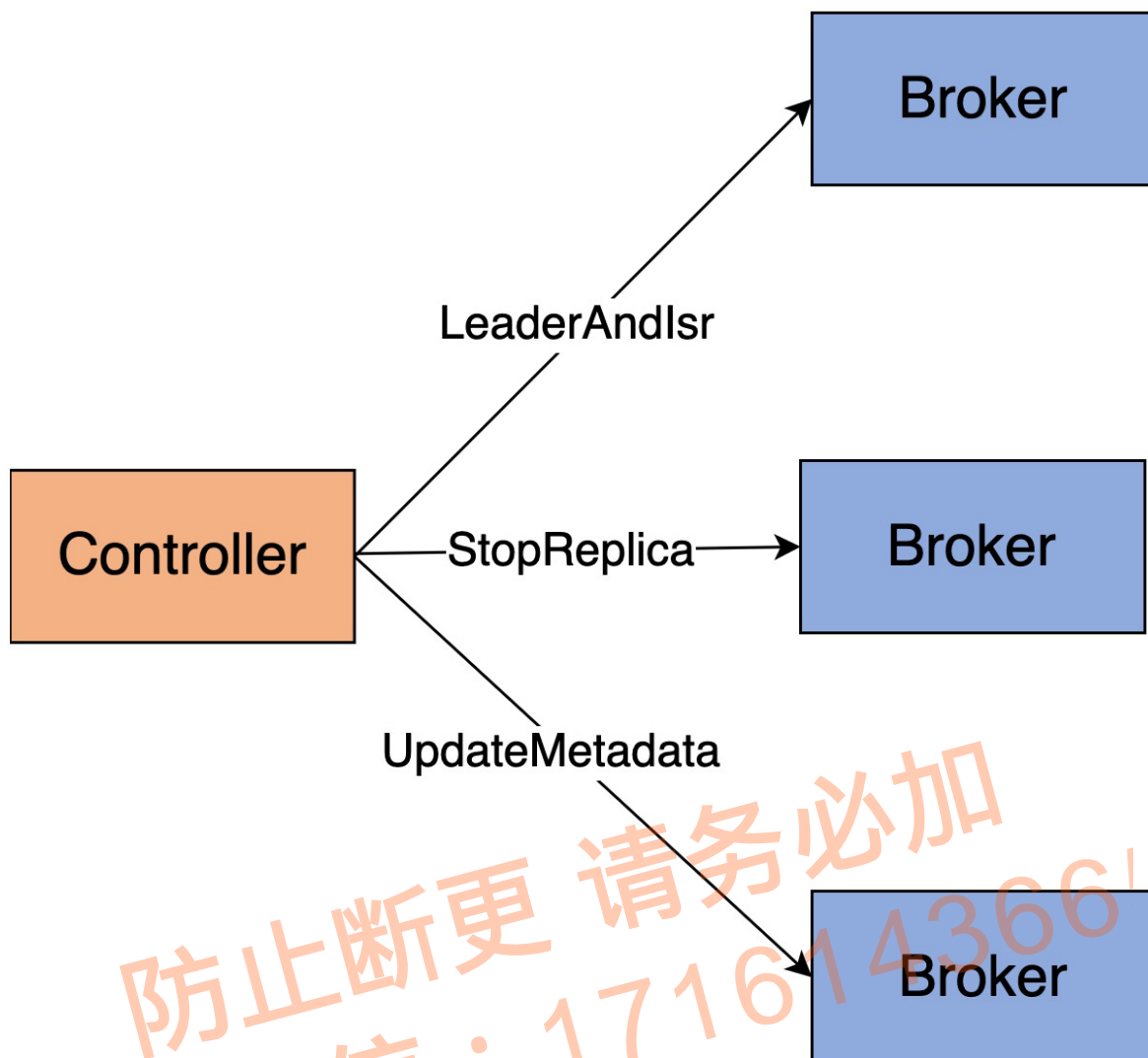
我的第一感觉是 Controller 出现了问题，但又苦于无从排查和验证。后来，我想到，会不会是 Controller 端请求队列中积压的请求太多造成的呢？因为当时 Controller 所在的 Broker 本身承载着非常重的业务，这是非常有可能的原因。

在看了相关代码后，我们就在相应的源码中新加了一个监控指标，用于实时监控 Controller 的请求队列长度。当更新到生产环境后，我们很轻松地定位了问题。果然，由于 Controller 所在的 Broker 自身负载过大，导致 Controller 端的请求积压，从而造成了元数据更新的滞后。精准定位了问题之后，解决起来就很容易了。后来，社区于 0.11 版本正式引入了相关的监控指标。

你看，阅读源码，除了可以学习优秀开发人员编写的代码之外，我们还能根据自身的实际情况做定制化方案，实现一些非开箱即用的功能。

## Controller 发送请求类型

下面，我们就正式进入到 Controller 请求发送管理部分的学习。你可能会问：“Controller 也会给 Broker 发送请求吗？”当然！**Controller 会给集群中的所有 Broker（包括它自己所在的 Broker）机器发送网络请求。**发送请求的目的，是让 Broker 执行相应的指令。我用一张图，来展示下 Controller 都会发送哪些请求，如下所示：



当前，Controller 只会向 Broker 发送三类请求，分别是 `LeaderAndIsrRequest`、`StopReplicaRequest` 和 `UpdateMetadataRequest`。注意，这里我使用的是“当前”！我只是说，目前仅有这三类，不代表以后不会有变化。事实上，我几乎可以肯定，以后能发送的 RPC 协议种类一定会变化的。因此，你需要掌握请求发送的原理。毕竟，所有请求发送都是通过相同的机制完成的。

还记得我在 [第 8 节课](#) 提到的控制类请求吗？没错，这三类请求就是典型的控制类请求。我来解释下它们的作用。

**LeaderAndIsrRequest**：最主要的功能是，告诉 Broker 相关主题各个分区的 Leader 副本位于哪台 Broker 上、ISR 中的副本都在哪些 Broker 上。在我看来，**它应该被赋予最高的优先级，毕竟，它有令数据类请求直接失效的本领**。试想一下，如果这个请求中的

Leader 副本变更了，之前发往老的 Leader 的 PRODUCE 请求是不是全部失效了？因此，我认为它是非常重要的控制类请求。


StopReplicaRequest：告知指定 Broker 停止它上面的副本对象，该请求甚至还能删除副本底层的日志数据。这个请求主要的使用场景，是**分区副本迁移**和**删除主题**。在这两个场景下，都要涉及停掉 Broker 上的副本操作。

UpdateMetadataRequest：顾名思义，该请求会更新 Broker 上的元数据缓存。集群上的所有元数据变更，都首先发生在 Controller 端，然后再经由这个请求广播给集群上的所有 Broker。在我刚刚分享的案例中，正是因为这个请求被处理得不及时，才导致集群 Broker 无法获取到最新的元数据信息。

现在，社区越来越倾向于**将重要的数据结构源代码从服务器端的 core 工程移动到客户端的 clients 工程中**。这三类请求 Java 类的定义就封装在 clients 中，它们的抽象基类是 AbstractControlRequest 类，这个类定义了这三类请求的公共字段。

我用代码展示下这三类请求及其抽象父类的定义，以便让你对 Controller 发送的请求类型有个基本的认识。这些类位于 clients 工程下的 src/main/java/org/apache/kafka/common/requests 路径下。

先来看 AbstractControlRequest 类的主要代码：

 复制代码

```
1 public abstract class AbstractControlRequest extends AbstractRequest {
2     public static final long UNKNOWN_BROKER_EPOCH = -1L;
3     public static abstract class Builder<T extends AbstractRequest> extends Ab:
4         protected final int controllerId;
5         protected final int controllerEpoch;
6         protected final long brokerEpoch;
7         .....
8 }
9
```

区别于其他的数据类请求，抽象类请求必然包含 3 个字段。


**controllerId**：Controller 所在的 Broker ID。

**controllerEpoch**：Controller 的版本信息。

**brokerEpoch**: 目标 Broker 的 Epoch。

后面这两个 Epoch 字段用于隔离 Zombie Controller 和 Zombie Broker，以保证集群的一致性。

在同一源码路径下，你能找到 LeaderAndIsrRequest、StopReplicaRequest 和 UpdateMetadataRequest 的定义，如下所示：

 复制代码

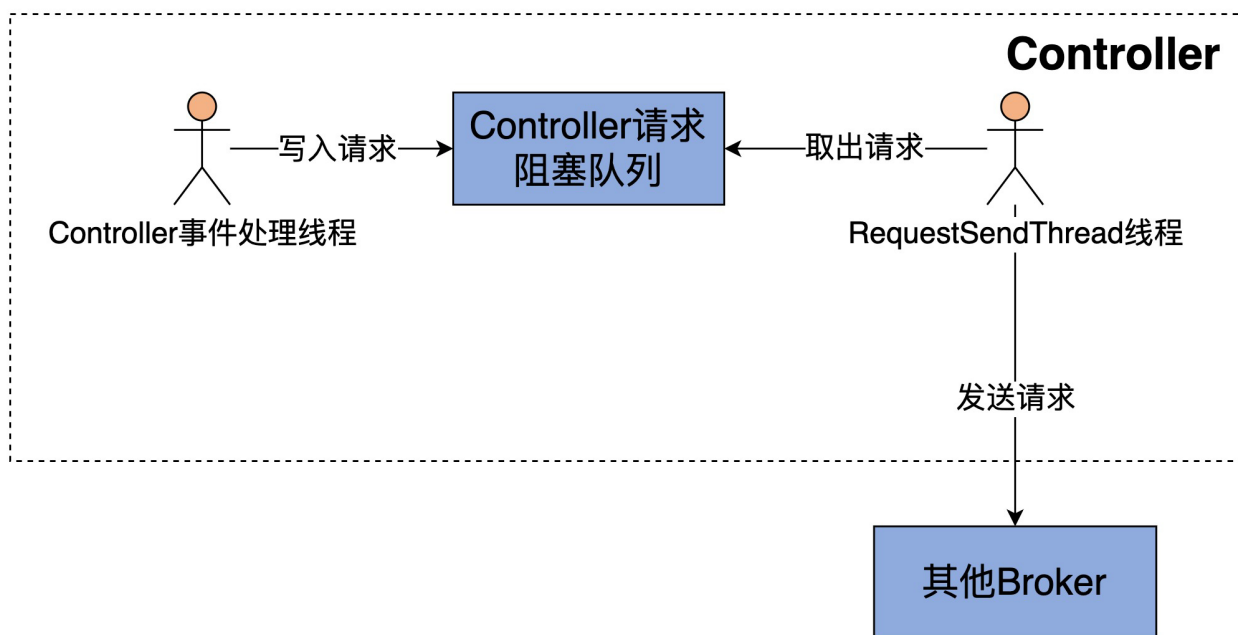
```
1 public class LeaderAndIsrRequest extends AbstractControlRequest { ..... }
2 public class StopReplicaRequest extends AbstractControlRequest { ..... }
3 public class UpdateMetadataRequest extends AbstractControlRequest { ..... }
```

## RequestSendThread

说完了 Controller 发送什么请求，接下来我们说说怎么发。

Kafka 源码非常喜欢生产者 - 消费者模式。该模式的好处在于，**解耦生产者和消费者逻辑，分离两者的集中性交互**。学完了“请求处理”模块，现在，你一定很赞同这个说法吧。还记得 Broker 端的 SocketServer 组件吗？它就在内部定义了一个线程共享的请求队列：它下面的 Processor 线程扮演 Producer，而 KafkaRequestHandler 线程扮演 Consumer。

对于 Controller 而言，源码同样使用了这个模式：它依然是一个线程安全的阻塞队列，Controller 事件处理线程（第 13 节课会详细说它）负责向这个队列写入待发送的请求，而一个名为 RequestSendThread 的线程负责执行真正的请求发送。如下图所示：



Controller 会为集群中的每个 Broker 都创建一个对应的 RequestSendThread 线程。Broker 上的这个线程，持续地从阻塞队列中获取待发送的请求。

那么，Controller 往阻塞队列上放什么数据呢？这其实是由源码中的 QueueItem 类定义的。代码如下：


```
1 case class QueueItem(apiKey: ApiKeys, request: AbstractControlRequest.Builder[复制代码
```

每个 QueueItem 的核心字段都是 **AbstractControlRequest.Builder** 对象。你基本上可以认为，它就是阻塞队列上 AbstractControlRequest 类型。

需要注意的是这里的 “<:” 符号，它在 Scala 中表示**上边界**的意思，即字段 request 必须是 AbstractControlRequest 的子类，也就是上面说到的那三类请求。

这也就是说，每个 QueueItem 实际保存的都是那三类请求中的其中一类。如果使用一个 BlockingQueue 对象来保存这些 QueueItem，那么，代码就实现了一个请求阻塞队列。这就是 RequestSendThread 类做的事情。

接下来，我们就来学习下 RequestSendThread 类的定义。我给一些主要的字段添加了注释。

 复制代码

```
1 class RequestSendThread(val controllerId: Int, // Controller所在Broker的Id
2     val controllerContext: ControllerContext, // Controller元数据信息
3     val queue: BlockingQueue[QueueItem], // 请求阻塞队列
4     val networkClient: NetworkClient, // 用于执行发送的网络I/O类
5     val brokerNode: Node, // 目标Broker节点
6     val config: KafkaConfig, // Kafka配置信息
7     val time: Time,
8     val requestRateAndQueueTimeMetrics: Timer,
9     val stateChangeLogger: StateChangeLogger,
10    name: String) extends ShutdownableThread(name = name) {
11    .....
12 }
```

其实，RequestSendThread 最重要的是它的 **doWork 方法**，也就是执行线程逻辑的方法：

 复制代码

```
1 override def doWork(): Unit = {
2     def backoff(): Unit = pause(100, TimeUnit.MILLISECONDS)
3     val QueueItem(apiKey, requestBuilder, callback, enqueueTimeMs) = queue.take()
4     requestRateAndQueueTimeMetrics.update(time.milliseconds() - enqueueTimeMs,
5     var clientResponse: ClientResponse = null
6     try {
7         var isSendSuccessful = false
8         while (isRunning && !isSendSuccessful) {
9             try {
10                // 如果没有创建与目标Broker的TCP连接，或连接暂时不可用
11                if (!brokerReady()) {
12                    isSendSuccessful = false
13                    backoff() // 等待重试
14                }
15            } else {
16                val clientRequest = networkClient.newClientRequest(brokerNode.idStr,
17                    time.milliseconds(), true)
18                // 发送请求，等待接收Response
19                clientResponse = NetworkClientUtils.sendAndReceive(networkClient,
20                    isSendSuccessful = true
21            }
22        } catch {
23            case e: Throwable =>
24                warn(s"Controller $controllerId epoch ${controllerContext.epoch} failed to send request to broker $brokerNode. Reconnecting to broker.", e)
25        }
```



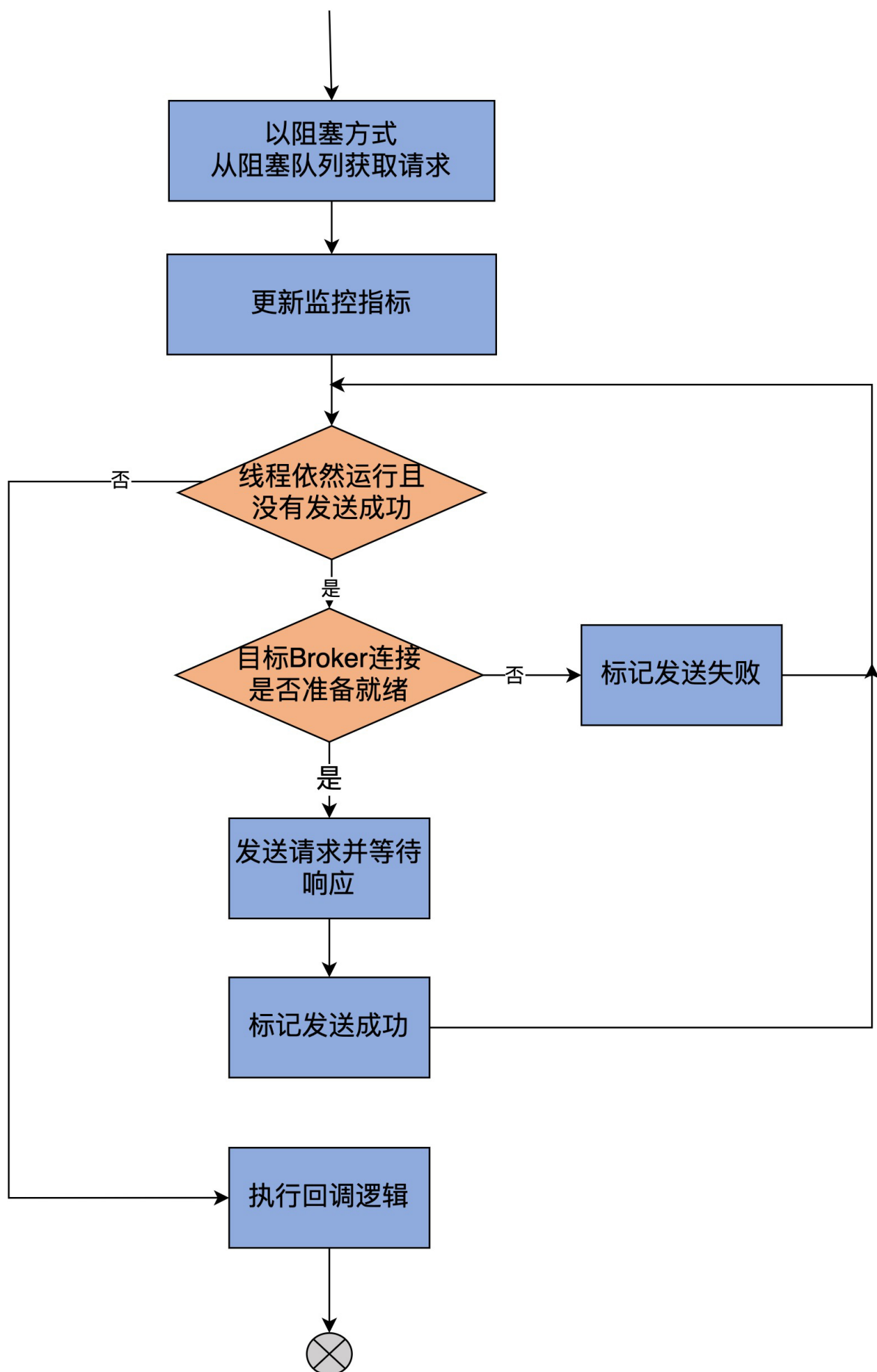
```

26         // 如果出现异常, 关闭与对应Broker的连接
27         networkClient.close(brokerNode.idString)
28         isSendSuccessful = false
29         backoff()
30     }
31 }
32 // 如果接收到了Response
33 if (clientResponse != null) {
34     val requestHeader = clientResponse.requestHeader
35     val api = requestHeader.apiKey
36     // 此Response的请求类型必须是LeaderAndIsrRequest、StopReplicaRequest或Upda
37     if (api != ApiKeys.LEADER_AND_ISR && api != ApiKeys.STOP_REPLICA && ap
38         throw new KafkaException(s"Unexpected apiKey received: $apiKey")
39     val response = clientResponse.responseBody
40     stateChangeLogger.withControllerEpoch(controllerContext.epoch)
41         .trace(s"Received response " +
42             s"${response.toString(requestHeader.apiVersion)} for request $api wi
43             s"${requestHeader.correlationId} sent to broker $brokerNode")
44
45     if (callback != null) {
46         callback(response) // 处理回调
47     }
48 }
49 } catch {
50     case e: Throwable =>
51         error(s"Controller $controllerId fails to send a request to broker $br
52         networkClient.close(brokerNode.idString)
53     }
54 }
55

```

我用一张图来说明 doWork 的执行逻辑:





总体上来看，doWork 的逻辑很直观。它的主要作用是从阻塞队列中取出待发送的请求，然后把它发送出去，之后等待 Response 的返回。在等待 Response 的过程中，线程将一直处于阻塞状态。当接收到 Response 之后，调用 callback 执行请求处理完成后的回调逻辑。

需要注意的是，RequestSendThread 线程对请求发送的处理方式与 Broker 处理请求不太一样。它调用的 sendAndReceive 方法在发送完请求之后，会原地进入阻塞状态，等待 Response 返回。只有接收到 Response，并执行完回调逻辑之后，该线程才能从阻塞队列中取出下一个待发送请求进行处理。

## ControllerChannelManager

了解了 RequestSendThread 线程的源码之后，我们进入到 ControllerChannelManager 类的学习。


这个类和 RequestSendThread 是合作共赢的关系。在我看来，它有两大类任务。

管理 Controller 与集群 Broker 之间的连接，并为每个 Broker 创建 RequestSendThread 线程实例；

将要发送的请求放入到指定 Broker 的阻塞队列中，等待该 Broker 专属的 RequestSendThread 线程进行处理。

由此可见，它们是紧密相连的。


ControllerChannelManager 类最重要的数据结构是 brokerStateInfo，它是在下面这行代码中定义的：

 复制代码

```
1 protected val brokerStateInfo = new HashMap[Int, ControllerBrokerStateInfo]
```

这是一个 HashMap 类型，Key 是 Integer 类型，其实就是集群中 Broker 的 ID 信息，而 Value 是一个 ControllerBrokerStateInfo。

你可能不太清楚 `ControllerBrokerStateInfo` 类是什么，我先解释一下。它本质上是一个 POJO 类，仅仅是承载若干数据结构的容器，如下所示：

 复制代码

```
1 case class ControllerBrokerStateInfo(networkClient: NetworkClient,  
2   brokerNode: Node,  
3   messageQueue: BlockingQueue[QueueItem],  
4   requestSendThread: RequestSendThread,  
5   queueSizeGauge: Gauge[Int],  
6   requestRateAndTimeMetrics: Timer,  
7   reconfigurableChannelBuilder: Option[Reconfigurable])
```

它有三个非常关键的字段。

**brokerNode**：目标 Broker 节点对象，里面封装了目标 Broker 的连接信息，比如主机名、端口号等。

**messageQueue**：请求消息阻塞队列。你可以发现，Controller 为每个目标 Broker 都创建了一个消息队列。

**requestSendThread**：Controller 使用这个线程给目标 Broker 发送请求。

你一定要记住这三个字段，因为它们是实现 Controller 发送请求的关键因素。

为什么呢？我们思考一下，如果 Controller 要给 Broker 发送请求，肯定需要解决三个问题：发给谁？发什么？怎么发？“发给谁”就是由 `brokerNode` 决定的；`messageQueue` 里面保存了要发送的请求，因而解决了“发什么”的问题；最后的“怎么发”就是依赖 `requestSendThread` 变量实现的。

好了，我们现在回到 `ControllerChannelManager`。它定义了 5 个 public 方法，我来——介绍下。

**startup 方法**：Controller 组件在启动时，会调用 `ControllerChannelManager` 的 `startup` 方法。该方法会从元数据信息中找到集群的 Broker 列表，然后依次为它们调用 `addBroker` 方法，把它们加到 `brokerStateInfo` 变量中，最后再依次启动 `brokerStateInfo` 中的 `RequestSendThread` 线程。

**shutdown 方法**：关闭所有 `RequestSendThread` 线程，并清空必要的资源。

**sendRequest 方法：**从名字看，就是发送请求，实际上就是把请求对象提交到请求队列。

**addBroker 方法：**添加目标 Broker 到 brokerStateInfo 数据结构中，并创建必要的配套资源，如请求队列、RequestSendThread 线程对象等。最后，RequestSendThread 启动线程。

**removeBroker 方法：**从 brokerStateInfo 移除目标 Broker 的相关数据。

这里面大部分的方法逻辑都很简单，从方法名字就可以看得出来。我重点说一下

**addBroker**，以及**底层相关的私有方法 addNewBroker 和 startRequestSendThread 方法**。

毕竟，addBroker 是最重要的逻辑。每当集群中扩容了新的 Broker 时，Controller 就会调用这个方法为新 Broker 增加新的 RequestSendThread 线程。

我们先来看 addBroker：

 复制代码

```
1 def addBroker(broker: Broker): Unit = {
2     brokerLock synchronized {
3         // 如果该Broker是新Broker的话
4         if (!brokerStateInfo.contains(broker.id)) {
5             // 将新Broker加入到Controller管理，并创建对应的RequestSendThread线程
6             addNewBroker(broker)
7             // 启动RequestSendThread线程
8             startRequestSendThread(broker.id)
9         }
10    }
11 }
12
```

整个代码段被 brokerLock 保护起来了。还记得 brokerStateInfo 的定义吗？它仅仅是一个 HashMap 对象，因为不是线程安全的，所以任何访问该变量的地方，都需要锁的保护。

这段代码的逻辑是，判断目标 Broker 的序号，是否已经保存在 brokerStateInfo 中。如果是，就说明这个 Broker 之前已经添加过了，就没必要再次添加了；否则，addBroker 方法会对目前的 Broker 执行两个操作：

1. 把该 Broker 节点添加到 brokerStateInfo 中;
2. 启动与该 Broker 对应的 RequestSendThread 线程。

这两步分别是由 addNewBroker 和 startRequestSendThread 方法实现的。

addNewBroker 方法的逻辑比较复杂，我用注释的方式给出主要步骤：

 复制代码

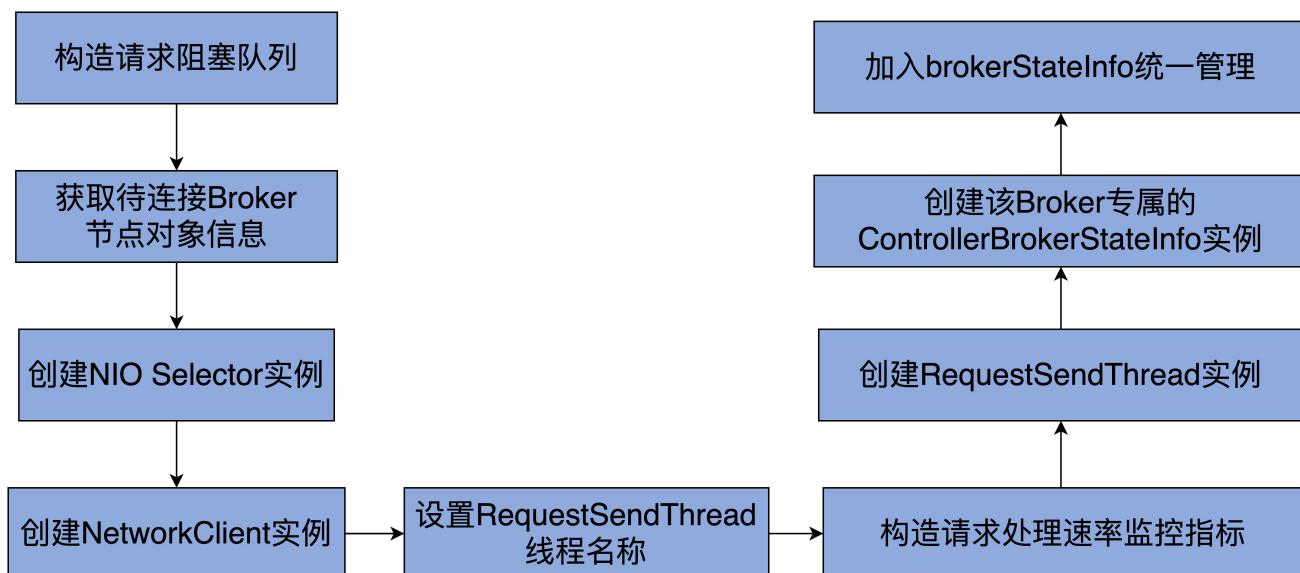
```
1 private def addNewBroker(broker: Broker): Unit = {
2     // 为该Broker构造请求阻塞队列
3     val messageQueue = new LinkedBlockingQueue[QueueItem]
4     debug(s"Controller ${config.brokerId} trying to connect to broker ${broker.id}")
5     val controllerToBrokerListenerName = config.controlPlaneListenerName.getOrElse(broker.id)
6     val controllerToBrokerSecurityProtocol = config.controlPlaneSecurityProtocol
7     // 获取待连接Broker节点对象信息
8     val brokerNode = broker.node(controllerToBrokerListenerName)
9     val logContext = new LogContext(s"[Controller id=${config.brokerId}, targetBrokerId=${broker.id}]")
10    val (networkClient, reconfigurableChannelBuilder) = {
11        val channelBuilder = ChannelBuilders.clientChannelBuilder(
12            controllerToBrokerSecurityProtocol,
13            JaasContext.Type.SERVER,
14            config,
15            controllerToBrokerListenerName,
16            config.saslMechanismInterBrokerProtocol,
17            time,
18            config.saslInterBrokerHandshakeRequestEnable,
19            logContext
20        )
21        val reconfigurableChannelBuilder = channelBuilder match {
22            case reconfigurable: Reconfigurable =>
23                config.addReconfigurable(reconfigurable)
24                Some(reconfigurable)
25            case _ => None
26        }
27        // 创建NIO Selector实例用于网络数据传输
28        val selector = new Selector(
29            NetworkReceive.UNLIMITED,
30            Selector.NO_IDLE_TIMEOUT_MS,
31            metrics,
32            time,
33            "controller-channel",
34            Map("broker-id" -> brokerNode.idString).asJava,
35            false,
36            channelBuilder,
37            logContext
38        )
39        // 创建NetworkClient实例
```

```

40 // NetworkClient类是Kafka clients工程封装的顶层网络客户端API
41 // 提供了丰富的方法实现网络层IO数据传输
42 val networkClient = new NetworkClient(
43     selector,
44     new ManualMetadataUpdater(Seq(brokerNode).asJava),
45     config.brokerId.toString,
46     1,
47     0,
48     0,
49     Selectable.USE_DEFAULT_BUFFER_SIZE,
50     Selectable.USE_DEFAULT_BUFFER_SIZE,
51     config.requestTimeoutMs,
52     ClientDnsLookup.DEFAULT,
53     time,
54     false,
55     new ApiVersions,
56     logContext
57 )
58 (networkClient, reconfigurableChannelBuilder)
59 }
60 // 为这个RequestSendThread线程设置线程名称
61 val threadName = threadNamePrefix match {
62     case None => s"Controller-`${config.brokerId}`-to-broker-`${broker.id}`-send-tl
63     case Some(name) => s"$name:Controller-`${config.brokerId}`-to-broker-`${broke
64 }
65 // 构造请求处理速率监控指标
66 val requestRateAndQueueTimeMetrics = newTimer(
67     RequestRateAndQueueTimeMetricName, TimeUnit.MILLISECONDS, TimeUnit.SECONDS
68 )
69 // 创建RequestSendThread实例
70 val requestThread = new RequestSendThread(config.brokerId, controllerContext
71     brokerNode, config, time, requestRateAndQueueTimeMetrics, stateChangeLogge
72     requestThread.setDaemon(false)
73
74 val queueSizeGauge = newGauge(QueueSizeMetricName, () => messageQueue.size, l
75 // 创建该Broker专属的ControllerBrokerStateInfo实例
76 // 并将其加入到brokerStateInfo统一管理
77 brokerStateInfo.put(broker.id, ControllerBrokerStateInfo(networkClient, brok
78     requestThread, queueSizeGauge, requestRateAndQueueTimeMetrics, reconfigural
79

```

为了方便你理解，我还画了一张流程图形象说明它的执行流程：



addNewBroker 的关键在于，**要为目标 Broker 创建一系列的配套资源**，比如，NetworkClient 用于网络 I/O 操作、messageQueue 用于阻塞队列、requestThread 用于发送请求，等等。

至于 startRequestSendThread 方法，就简单得多了，只有几行代码而已。

复制代码

```
1 protected def startRequestSendThread(brokerId: Int): Unit = {
2     // 获取指定Broker的专属RequestSendThread实例
3     val requestThread = brokerStateInfo(brokerId).requestSendThread
4     if (requestThread.getState == Thread.State.NEW)
5         // 启动线程
6         requestThread.start()
7 }
```

它首先根据给定的 Broker 序号信息，从 brokerStateInfo 中找出对应的 ControllerBrokerStateInfo 对象。有了这个对象，也就有了为该目标 Broker 服务的所有配套资源。下一步就是从 ControllerBrokerStateInfo 中拿出 RequestSendThread 对象，再启动它就好了。

## 总结

今天，我结合 ControllerChannelManager.scala 文件，重点分析了 Controller 向 Broker 发送请求机制的实现原理。



Controller 主要通过 ControllerChannelManager 类来实现与其他 Broker 之间的请求发送。其中，ControllerChannelManager 类中定义的 RequestSendThread 是主要的线程实现类，用于实际发送请求给集群 Broker。除了 RequestSendThread 之外，ControllerChannelManager 还定义了相应的管理方法，如添加 Broker、移除 Broker 等。通过这些管理方法，Controller 在集群扩缩容时能够快速地响应到这些变化，完成对应 Broker 连接的创建与销毁。

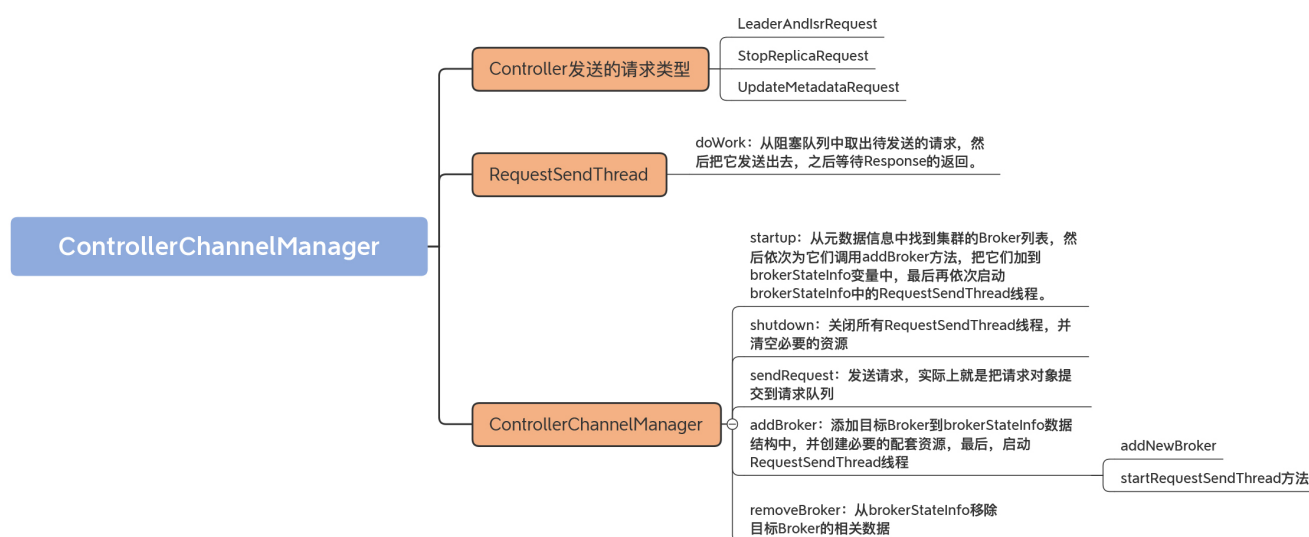
我们来回顾下这节课的重点。

Controller 端请求：Controller 发送三类请求给 Broker，分别是 LeaderAndIsrRequest、StopReplicaRequest 和 UpdateMetadataRequest。

RequestSendThread：该线程负责将请求发送给集群中的相关或所有 Broker。

请求阻塞队列 + RequestSendThread：Controller 会为集群上所有 Broker 创建对应的请求阻塞队列和 RequestSendThread 线程。

其实，今天讲的所有东西都只是这节课的第二张图中“消费者”的部分，我们并没有详细了解请求是怎么被放到请求队列中的。接下来，我们就会针对这个问题，深入地去探讨 Controller 单线程的事件处理器是如何实现的。



## 课后讨论

你觉得，为每个 Broker 都创建一个 RequestSendThread 的方案有什么优缺点？

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。

## 课程预告

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

下一篇 13 | ControllerEventManager：变身单线程后的Controller如何处理事件？

## 精选留言 (2)

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Controller元数据类ControllerContext。课后我请你自行分析下partitionLeadershipInfo里面保存的内容。实际上，这是一个按照主题分区分组的分区详细数据容器。它保存了每个分区Leader副本位于哪个Broker、Leader Epoch值是多少、...  
展开





花轮君

2020-05-16

从另外一个角度看，如果controller只有单线程去调用所有的broker，那如果中间某一台的broker交互阻塞，势必会影响到整个集群。采用broker维护request的方案在于必须加上对应的监控

作者回复: 很有道理: )



💬 1

