

06 | 请求通道：如何实现Kafka请求队列？

2020-04-28 胡夕

Kafka核心源码解读

[进入课程 >](#)



讲述：胡夕

时长 21:34 大小 19.76M



你好，我是胡夕。日志模块我们已经讲完了，你掌握得怎样了呢？如果你在探索源码的过程中碰到了问题，记得在留言区里写下你的困惑，我保证做到知无不言。

现在，让我们开启全新的“请求处理模块”的源码学习之旅。坦率地讲，这是我自己给Kafka 源码划分的模块，在源码中可没有所谓的“请求处理模块”的提法。但我始终觉得，这样划分能够帮助你清晰地界定不同部分的源码的作用，可以让你的阅读更有针对性，学习效果也会事半功倍。



在这个模块，我会带你了解请求处理相关的重点内容，包括请求处理通道、请求处理全流程分析和请求入口类分析等。今天，我们先来学习下 Kafka 是如何实现请求队列的。源码中

位于 `core/src/main/scala/kafka/network` 下的 `RequestChannel.scala` 文件，是主要的实现类。

当我们说到 Kafka 服务器端，也就是 Broker 的时候，往往会说它承担着消息持久化的功能，但本质上，它其实就是一个**不断接收外部请求、处理请求，然后发送处理结果的 Java 进程**。


你可能会觉得奇怪，Broker 不是用 Scala 语言编写的吗，怎么这里又是 Java 进程了呢？这是因为，Scala 代码被编译之后生成 `.class` 文件，它和 Java 代码被编译后的效果是一样的，因此，Broker 启动后也仍然是一个普通的 Java 进程。

高效地保存排队中的请求，是确保 Broker 高处理性能的关键。既然这样，那你一定很想知道，Broker 上的请求队列是怎么实现的呢？接下来，我们就一起看下 **Broker 底层请求对象的建模和请求队列的实现原理**，以及 Broker 请求处理方面的核心监控指标。

目前，Broker 与 Clients 进行交互主要是基于 **Request/Response 机制**，所以，我们很有必要学习一下源码是如何建模或定义 Request 和 Response 的。

请求 (Request)

我们先来看一下 `RequestChannel` 源码中的 Request 定义代码。

 复制代码

```
1 sealed trait BaseRequest
2 case object ShutdownRequest extends BaseRequest
3
4 class Request(val processor: Int,
5               val context: RequestContext,
6               val startTimeNanos: Long,
7               memoryPool: MemoryPool,
8               @volatile private var buffer: ByteBuffer,
9               metrics: RequestChannel.Metrics) extends BaseRequest {
10     .....
11 }
```

简单提一句，Scala 语言中的 “`trait`” 关键字，大致类似于 Java 中的 `interface`（接口）。从代码中，我们可以知道，`BaseRequest` 是一个 `trait` 接口，定义了基础的请求类型。它有两个实现类：**`ShutdownRequest` 类和 `Request` 类**。

ShutdownRequest 仅仅起到一个标志位的作用。当 Broker 进程关闭时，请求处理器（RequestHandler，在第 9 讲我会具体讲到它）会发送 ShutdownRequest 到专属的请求处理线程。该线程接收到此请求后，会主动触发一系列的 Broker 关闭逻辑。

Request 则是真正定义各类 Clients 端或 Broker 端请求的实现类。它定义的属性包括 processor、context、startTimeNanos、memoryPool、buffer 和 metrics。下面我们一起来看看。

processor


processor 是 Processor 线程的序号，即**这个请求是由哪个 Processor 线程接收处理的**。Broker 端参数 num.network.threads 控制了 Broker 每个监听器上创建的 Processor 线程数。

假设你的 listeners 配置为 PLAINTEXT://localhost:9092,SSL://localhost:9093，那么，在默认情况下，Broker 启动时会创建 6 个 Processor 线程，每 3 个为一组，分别给 listeners 参数中设置的两个监听器使用，每组的序号分别是 0、1、2。

你可能会问，为什么要保存 Processor 线程序号呢？这是因为，当 Request 被后面的 I/O 线程处理完成后，还要依靠 Processor 线程发送 Response 给请求发送方，因此，Request 中必须记录它之前是被哪个 Processor 线程接收的。另外，这里我们要先明确一点：**Processor 线程仅仅是网络接收线程，不会执行真正的 Request 请求处理逻辑**，那是 I/O 线程负责的事情。

context

context 是用来标识请求上下文信息的。Kafka 源码中定义了 RequestContext 类，顾名思义，它保存了有关 Request 的所有上下文信息。RequestContext 类定义在 clients 工程中，下面是它主要的逻辑代码。我用注释的方式解释下主体代码的含义。

 复制代码

```
1 public class RequestContext implements AuthorizableRequestContext {
2     public final RequestHeader header; // Request头部数据，主要是一些对用户不可见的元数据
3     public final String connectionId; // Request发送方的TCP连接串标识，由Kafka根据-
4     public final InetAddress clientAddress; // Request发送方IP地址
5     public final KafkaPrincipal principal; // Kafka用户认证类，用于认证授权
6     public final ListenerName listenerName; // 监听器名称，可以是预定义的监听器（如PLAINTEXT, SSL等）
7     public final SecurityProtocol securityProtocol; // 安全协议类型，目前支持4种：PLAINTEXT, SSL, SASL_PLAINTEXT, SASL_SSL
```

```

8     public final ClientInformation clientInformation; // 用户自定义的一些连接方信息
9     // 从给定的ByteBuffer中提取出Request和对应的Size值
10    public RequestAndSize parseRequest(ByteBuffer buffer) {
11        .....
12    }
13    // 其他Getter方法
14    .....
15 }

```

startTimeNanos

startTimeNanos 记录了 Request 对象被创建的时间，主要用于各种时间统计指标的计算。

请求对象中的很多 JMX 指标，特别是时间类的统计指标，都需要使用 startTimeNanos 字段。你要注意的是，**它是以纳秒为单位的时间戳信息，可以实现非常细粒度的时间统计精度。**


memoryPool

memoryPool 表示源码定义的一个非阻塞式的内存缓冲区，主要作用是**避免 Request 对象无限使用内存。**

当前，该内存缓冲区的接口类和实现类，分别是 MemoryPool 和 SimpleMemoryPool。你可以重点关注下 SimpleMemoryPool 的 tryAllocate 方法，看看它是怎么为 Request 对象分配内存的。

buffer

buffer 是真正保存 Request 对象内容的字节缓冲区。Request 发送方必须按照 Kafka RPC 协议规定的格式向该缓冲区写入字节，否则将抛出 InvalidRequestException 异常。**这个逻辑主要是由 RequestContext 的 parseRequest 方法实现的。**

 复制代码

```

1 public RequestAndSize parseRequest(ByteBuffer buffer) {
2     if (isUnsupportedApiVersionsRequest()) {
3         // 不支持的ApiVersions请求类型被视为是v0版本的请求，并且不做解析操作，直接返回
4         ApiVersionsRequest apiVersionsRequest = new ApiVersionsRequest(new Api'
5             return new RequestAndSize(apiVersionsRequest, 0);
6     } else {

```

```

7      // 从请求头部数据中获取ApiKeys信息
8      ApiKeys apiKey = header.apiKey();
9      try {
10         // 从请求头部数据中获取版本信息
11         short apiVersion = header.apiVersion();
12         // 解析请求
13         Struct struct = apiKey.parseRequest(apiVersion, buffer);
14         AbstractRequest body = AbstractRequest.parseRequest(apiKey, apiVer:
15         // 封装解析后的请求对象以及请求大小返回
16         return new RequestAndSize(body, struct.sizeOf());
17     } catch (Throwable ex) {
18         // 解析过程中出现任何问题都视为无效请求，抛出异常
19         throw new InvalidRequestException("Error getting request for apiKe:
20             ", apiVersion: " + header.apiVersion() +
21             ", connectionId: " + connectionId +
22             ", listenerName: " + listenerName +
23             ", principal: " + principal, ex);
24     }
25 }
26 }

```

就像前面说过的，这个方法的主要目的是**从 ByteBuffer 中提取对应的 Request 对象以及它的大小**。

首先，代码会判断该 Request 是不是 Kafka 不支持的 ApiVersions 请求版本。如果是不支持的，就直接构造一个 V0 版本的 ApiVersions 请求，然后返回。否则的话，就继续下面的代码。

这里我稍微解释一下 ApiVersions 请求的作用。当 Broker 接收到一个 ApiVersionsRequest 的时候，它会返回 Broker 当前支持的请求类型列表，包括请求类型名称、支持的最早版本号和最新版本号。如果你查看 Kafka 的 bin 目录的话，你应该能找到一个名为 kafka-broker-api-versions.sh 的脚本工具。它的实现原理就是，构造 ApiVersionsRequest 对象，然后发送给对应的 Broker。

你可能会问，如果是 ApiVersions 类型的请求，代码中为什么要判断一下它的版本呢？这是因为，和处理其他类型请求不同的是，Kafka 必须保证版本号比最新支持版本还要高的 ApiVersions 请求也能被处理。这主要是考虑到了客户端和服务端版本的兼容问题。客户端发送请求给 Broker 的时候，很可能不知道 Broker 到底支持哪些版本的请求，它需要使用 ApiVersionsRequest 去获取完整的请求版本支持列表。但是，如果不做这个判断，Broker 可能无法处理客户端发送的 ApiVersionsRequest。

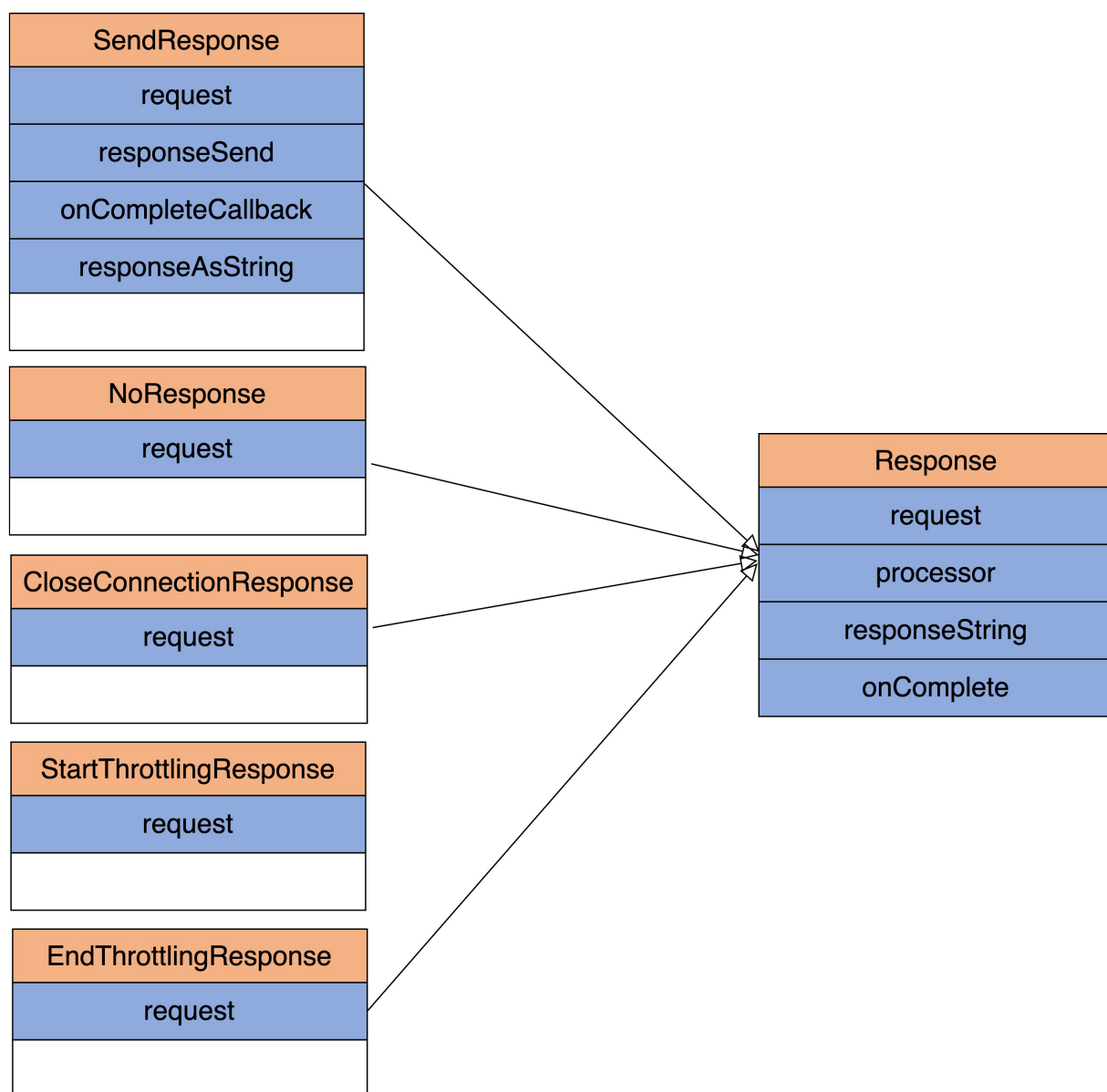
通过这个检查之后，代码会从请求头部数据中获取 ApiKeys 信息以及对应的版本信息，然后解析请求，最后封装解析后的请求对象以及请求大小，并返回。

metrics

metrics 是 Request 相关的各种监控指标的一个管理类。它里面构建了一个 Map，封装了所有的请求 JMX 指标。除了上面这些重要的字段属性之外，Request 类中的大部分代码都是与监控指标相关的，后面我们再详细说。

响应 (Response)

说完了 Request 代码，我们再来说下 Response。Kafka 为 Response 定义了 1 个抽象父类和 5 个具体子类，如下图所示：



看到这么多类，你可能会有点蒙，这些都是干吗的呢？别着急，现在我分别给你介绍下各个类的作用。

Response：定义 Response 的抽象基类。每个 Response 对象都包含了对应的 Request 对象。这个类里最重要的方法是 `onComplete` 方法，用来实现每类 Response 被处理后需要执行的回调逻辑。

SendResponse：Kafka 大多数 Request 处理完成后都需要执行一段回调逻辑，SendResponse 就是保存返回结果的 Response 子类。里面最重要的字段是 **`onCompletionCallback`**，即指定处理完成之后的回调逻辑。

NoResponse: 有些 Request 处理完成后无需单独执行额外的回调逻辑。NoResponse 就是为这类 Response 准备的。

CloseConnectionResponse: 用于出错后需要关闭 TCP 连接的场景，此时返回 CloseConnectionResponse 给 Request 发送方，显式地通知它关闭连接。

StartThrottlingResponse: 用于通知 Broker 的 Socket Server 组件（后面几节课我会讲到它）某个 TCP 连接通信通道开始被限流（throttling）。

EndThrottlingResponse: 与 StartThrottlingResponse 对应，通知 Broker 的 SocketServer 组件某个 TCP 连接通信通道的限流已结束。

你可能又会问了：“这么多类，我都要掌握吗？”其实是不用的。你只要了解 SendResponse 表示正常需要发送 Response，而 NoResponse 表示无需发送 Response 就可以了。至于 CloseConnectionResponse，它是用于标识关闭连接通道的 Response。而后面两个 Response 类不是很常用，它们仅仅在对 Socket 连接进行限流时，才会派上用场，这里我就不具体展开讲了。

Okay，现在，我们看下 Response 相关的代码部分。

 复制代码

```
1 abstract class Response(val request: Request) {
2     locally {
3         val nowNs = Time.SYSTEM.nanoseconds
4         request.responseCompleteTimeNanos = nowNs
5         if (request.apiLocalCompleteTimeNanos == -1L)
6             request.apiLocalCompleteTimeNanos = nowNs
7     }
8     def processor: Int = request.processor
9     def responseString: Option[String] = Some("")
10    def onComplete: Option[Send => Unit] = None
11    override def toString: String
12 }
```

这个抽象基类只有一个属性字段：request。这就是说，**每个 Response 对象都要保存它对应的 Request 对象**。我在前面说过，onComplete 方法是调用指定回调逻辑的地方。SendResponse 类就是复写（Override）了这个方法，如下所示：

 复制代码

```
1 class SendResponse(request: Request,
```



```

2             val responseSend: Send,
3             val responseAsString: Option[String],
4             val onCompleteCallback: Option[Send => Unit])
5     extends Response(request) {
6         .....
7         override def onComplete: Option[Send => Unit] = onCompleteCallback
8     }

```

这里的 `SendResponse` 类继承了 `Response` 父类，并重新定义了 `onComplete` 方法。复写的逻辑很简单，就是指定输入参数 `onCompleteCallback`。其实方法本身没有什么可讲的，反倒是这里的 Scala 语法值得多说几句。

Scala 中的 `Unit` 类似于 Java 中的 `void`，而 “`Send => Unit`” 表示一个方法。这个方法接收一个 `Send` 类实例，然后执行一段代码逻辑。Scala 是函数式编程语言，函数在 Scala 中是 “一等公民”，因此，你可以把一个函数作为一个参数传给另一个函数，也可以把函数作为结果返回。这里的 `onComplete` 方法就应用了第二种用法，也就是把函数赋值给另一个函数，并作为结果返回。这样做的好处在于，你可以灵活地变更 `onCompleteCallback` 来实现不同的回调逻辑。

RequestChannel

`RequestChannel`，顾名思义，就是传输 `Request/Response` 的通道。有了 `Request` 和 `Response` 的基础，下面我们可以学习 `RequestChannel` 类的实现了。

我们先看下 `RequestChannel` 类的定义和重要的字段属性。

📄 复制代码

```

1 class RequestChannel(val queueSize: Int, val metricNamePrefix : String) extend:
2     import RequestChannel._
3     val metrics = new RequestChannel.Metrics
4     private val requestQueue = new ArrayBlockingQueue[BaseRequest](queueSize)
5     private val processors = new ConcurrentHashMap[Int, Processor]()
6     val requestQueueSizeMetricName = metricNamePrefix.concat(RequestQueueSizeMet
7     val responseQueueSizeMetricName = metricNamePrefix.concat(ResponseQueueSizeM
8
9     .....
10 }

```

RequestChannel 类实现了 KafkaMetricsGroup trait，后者封装了许多实用的指标监控方法，比如，newGauge 方法用于创建数值型的监控指标，newHistogram 方法用于创建直方图型的监控指标。

就 RequestChannel 类本身的主体功能而言，它定义了最核心的 3 个属性：requestQueue、queueSize 和 processors。下面我分别解释下它们的含义。

每个 RequestChannel 对象实例创建时，会定义一个队列来保存 Broker 接收到的各类请求，这个队列被称为请求队列或 Request 队列。Kafka 使用 **Java 提供的阻塞队列 ArrayBlockingQueue** 实现这个请求队列，并利用它天然提供的线程安全性来保证多个线程能够并发安全高效地访问请求队列。在代码中，这个队列由变量requestQueue定义。


而字段 queueSize 就是 Request 队列的最大长度。当 Broker 启动时，SocketServer 组件会创建 RequestChannel 对象，并把 Broker 端参数 queued.max.requests 赋值给 queueSize。因此，在默认情况下，每个 RequestChannel 上的队列长度是 500。

字段 processors 封装的是 RequestChannel 下辖的 Processor 线程池。每个 Processor 线程负责具体的请求处理逻辑。下面我详细说说 Processor 的管理。

Processor 管理

上面代码中的第六行创建了一个 Processor 线程池——当然，它是用 Java 的 ConcurrentHashMap 数据结构去保存的。Map 中的 Key 就是前面我们说的 processor 序号，而 Value 则对应具体的 Processor 线程对象。

这个线程池的存在告诉了我们一个事实：**当前 Kafka Broker 端所有网络线程都是在 RequestChannel 中维护的。**既然创建了线程池，代码中必然要有管理线程池的操作。RequestChannel 中的 addProcessor 和 removeProcessor 方法就是做这些事的。

 复制代码

```
1 def addProcessor(processor: Processor): Unit = {
2   // 添加Processor到Processor线程池
3   if (processors.putIfAbsent(processor.id, processor) != null)
4     warn(s"Unexpected processor with processorId ${processor.id}")
5     newGauge(responseQueueSizeMetricName,
6       () => processor.responseQueueSize,
7       // 为给定Processor对象创建对应的监控指标
8 }
```

```
9      Map(ProcessorMetricTag -> processor.id.toString))
10 }
11
12 def removeProcessor(processorId: Int): Unit = {
13     processors.remove(processorId) // 从Processor线程池中移除给定Processor线程
14     removeMetric(responseQueueSizeMetricName, Map(ProcessorMetricTag -> processoro
15 }
```


代码很简单，基本上就是调用 `ConcurrentHashMap` 的 `putIfAbsent` 和 `remove` 方法分别实现增加和移除线程。每当 Broker 启动时，它都会调用 `addProcessor` 方法，向 `RequestChannel` 对象添加 `num.network.threads` 个 `Processor` 线程。

如果查询 Kafka 官方文档的话，你就会发现，`num.network.threads` 这个参数的更新模式 (Update Mode) 是 Cluster-wide。这就说明，Kafka 允许你动态地修改此参数值。比如，Broker 启动时指定 `num.network.threads` 为 8，之后你通过 `kafka-configs` 命令将其修改为 3。显然，这个操作会减少 `Processor` 线程池中的线程数量。在这个场景下，`removeProcessor` 方法会被调用。

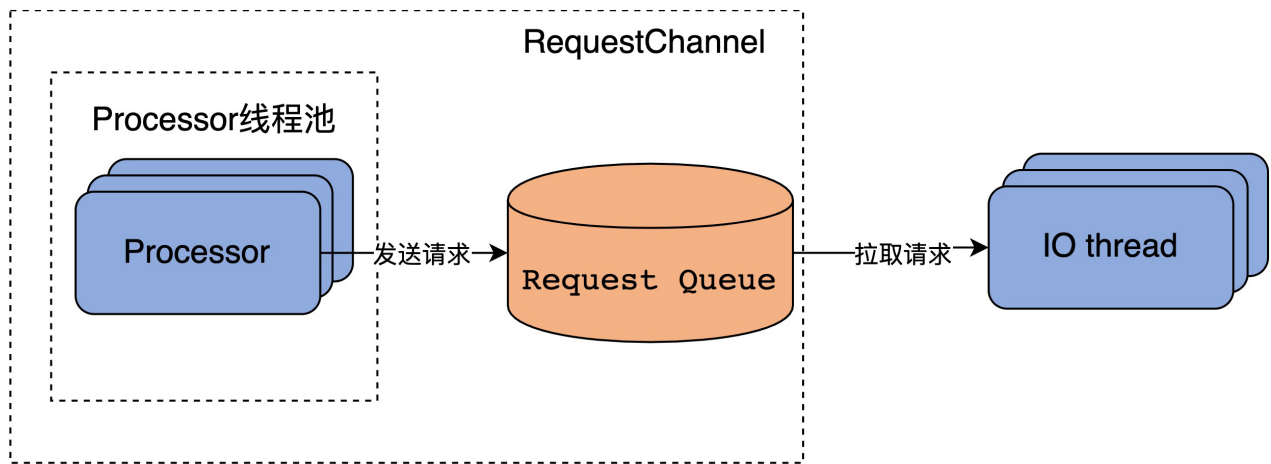
处理 Request 和 Response

除了 `Processor` 的管理之外，`RequestChannel` 的另一个重要功能，是处理 **Request 和 Response**，具体表现为收发 `Request` 和发送 `Response`。比如，收发 `Request` 的方法有 `sendRequest` 和 `receiveRequest`：

```
1 def sendRequest(request: RequestChannel.Request): Unit = {
2     requestQueue.put(request)
3 }
4 def receiveRequest(timeout: Long): RequestChannel.BaseRequest =
5     requestQueue.poll(timeout, TimeUnit.MILLISECONDS)
6 def receiveRequest(): RequestChannel.BaseRequest =
7     requestQueue.take()
```

 复制代码

所谓的发送 `Request`，仅仅是将 `Request` 对象放置在 `Request` 队列中而已，而接收 `Request` 则是从队列中取出 `Request`。整个流程构成了一个迷你版的“生产者 - 消费者”模式，然后依靠 `ArrayBlockingQueue` 的线程安全性来确保整个过程的线程安全，如下所示：



对于 Response 而言，则没有所谓的接收 Response，只有发送 Response，即 sendResponse 方法。sendResponse 是啥意思呢？其实就是把 Response 对象发送出去，也就是将 Response 添加到 Response 队列的过程。

复制代码

```
1 def sendResponse(response: RequestChannel.Response): Unit = {
2     if (isTraceEnabled) { // 构造Trace日志输出字符串
3         val requestHeader = response.request.header
4         val message = response match {
5             case sendResponse: SendResponse =>
6                 s"Sending ${requestHeader.apiKey} response to client ${requestHeader}
7             case _: NoOpResponse =>
8                 s"Not sending ${requestHeader.apiKey} response to client ${requestHeader}
9             case _: CloseConnectionResponse =>
10                s"Closing connection for client ${requestHeader.clientId} due to error
11             case _: StartThrottlingResponse =>
12                s"Notifying channel throttling has started for client ${requestHeader}
13             case _: EndThrottlingResponse =>
14                s"Notifying channel throttling has ended for client ${requestHeader}
15        }
16        trace(message)
17    }
18    // 找出response对应的Processor线程，即request当初是由哪个Processor线程处理的
19    val processor = processors.get(response.processor)
20    // 将response对象放置到对应Processor线程的Response队列中
21    if (processor != null) {
22        processor.enqueueResponse(response)
23    }
24 }
```

sendResponse 方法的逻辑其实非常简单。

前面的一大段 if 代码块仅仅是构造 Trace 日志要输出的内容。根据不同类型的 Response，代码需要确定要输出的 Trace 日志内容。

接着，代码会找出 Response 对象对应的 Processor 线程。当 Processor 处理完某个 Request 后，会把自己的序号封装进对应的 Response 对象。一旦找出了之前是由哪个 Processor 线程处理的，代码直接调用该 Processor 的 enqueueResponse 方法，将 Response 放入 Response 队列中，等待后续发送。

监控指标实现

RequestChannel 类还定义了丰富的监控指标，用于实时动态地监测 Request 和 Response 的性能表现。我们来看下具体指标项都有哪些。

 复制代码

```
1 object RequestMetrics {
2     val consumerFetchMetricName = ApiKeys.FETCH.name + "Consumer"
3     val followFetchMetricName = ApiKeys.FETCH.name + "Follower"
4     val RequestsPerSec = "RequestsPerSec"
5     val RequestQueueTimeMs = "RequestQueueTimeMs"
6     val LocalTimeMs = "LocalTimeMs"
7     val RemoteTimeMs = "RemoteTimeMs"
8     val ThrottleTimeMs = "ThrottleTimeMs"
9     val ResponseQueueTimeMs = "ResponseQueueTimeMs"
10    val ResponseSendTimeMs = "ResponseSendTimeMs"
11    val TotalTimeMs = "TotalTimeMs"
12    val RequestBytes = "RequestBytes"
13    val MessageConversionsTimeMs = "MessageConversionsTimeMs"
14    val TemporaryMemoryBytes = "TemporaryMemoryBytes"
15    val ErrorsPerSec = "ErrorsPerSec"
16 }
```

可以看到，指标有很多，不过别有压力，我们只要掌握几个重要的就行了。

RequestsPerSec：每秒处理的 Request 数，用来评估 Broker 的繁忙状态。

RequestQueueTimeMs：计算 Request 在 Request 队列中的平均等候时间，单位是毫秒。倘若 Request 在队列的等待时间过长，你通常需要增加后端 I/O 线程的数量，来加快队列中 Request 的拿取速度。

LocalTimeMs: 计算 Request 实际被处理的时间，单位是毫秒。一旦定位到这个监控项的值很大，你就需要进一步研究 Request 被处理的逻辑了，具体分析到底是哪一步消耗了过多的时间。

RemoteTimeMs: Kafka 的读写请求（PRODUCE 请求和 FETCH 请求）逻辑涉及等待其他 Broker 操作的步骤。RemoteTimeMs 计算的，就是等待其他 Broker 完成指定逻辑的时间。因为等待的是其他 Broker，因此被称为 Remote Time。这个监控项非常重要！Kafka 生产环境中设置 acks=all 的 Producer 程序发送消息延时高的主要原因，往往就是 Remote Time 高。因此，如果你也碰到了这样的问题，不妨先定位一下 Remote Time 是不是瓶颈。

TotalTimeMs: 计算 Request 被处理的完整流程时间。**这是最实用的监控指标，没有之一！**毕竟，我们通常都是根据 TotalTimeMs 来判断系统是否出现问题的。一旦发现了问题，我们才会利用前面的几个监控项进一步定位问题的原因。

RequestChannel 定义了 updateMetrics 方法，用于实现监控项的更新，因为逻辑非常简单，我就不展开说了，你可以自己阅读一下。

总结

好了，又到了总结时间。

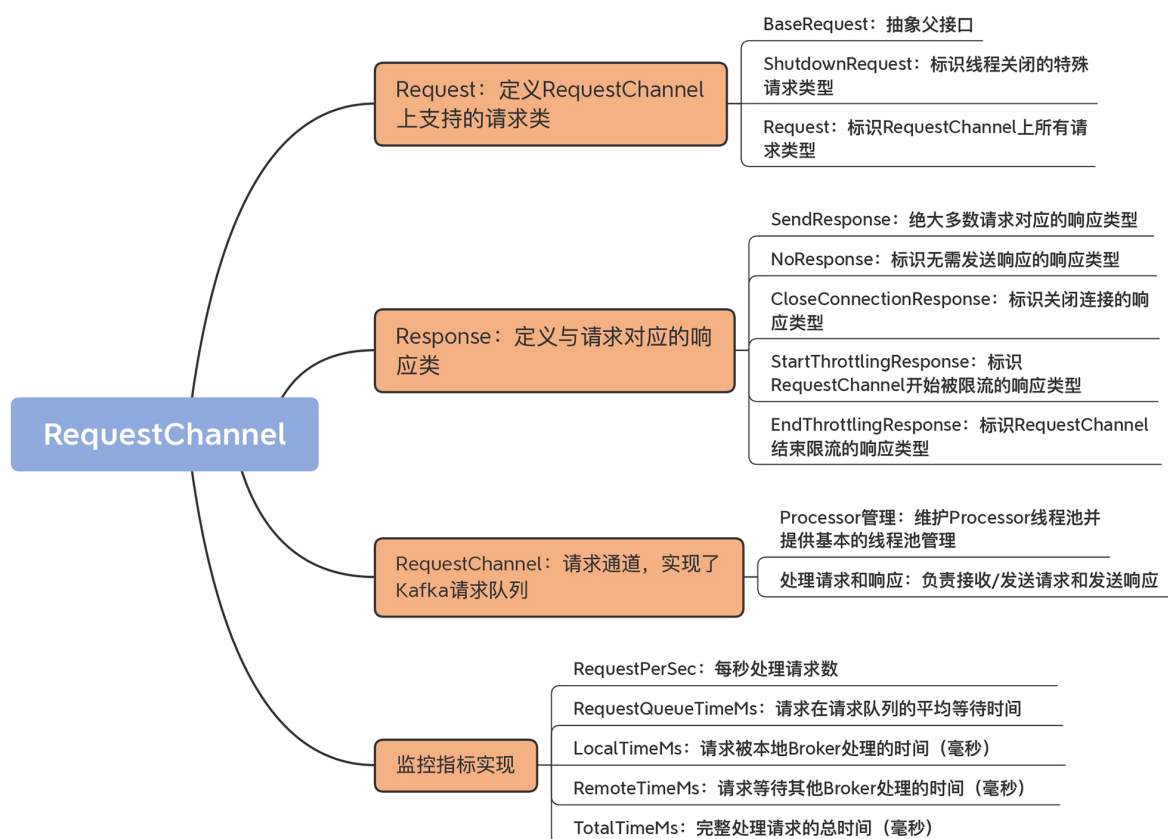
今天，我带你阅读了 Kafka 请求队列的实现源码。围绕这个问题，我们学习了几个重点内容。

Request: 定义了 Kafka Broker 支持的各类请求。

Response: 定义了与 Request 对应的各类响应。

RequestChannel: 实现了 Kafka Request 队列。

监控指标: 封装了与 Request 队列相关的重要监控指标。



希望你结合今天所讲的内容思考一下，Request 和 Response 在请求通道甚至是 SocketServer 中的流转过程，这将极大地帮助你了解 Kafka 是如何处理外部发送的请求的。当然，如果你觉得这个有难度，也不必着急，因为后面我会专门用一节课来告诉你这些内容。

课后讨论

如果我想监控 Request 队列当前的使用情况（比如当前已保存了多少个 Request），你可以结合源码指出，应该使用哪个监控指标吗？

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。

Kafka 核心源码解读

从底层到实战，深度解析源码

胡夕

友信金服商业智能部总监

Apache Kafka Contributor



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 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 05 | 索引（下）：位移索引和时间戳索引的区别是什么？

下一篇 特别放送（一）| 经典的Kafka学习资料有哪些？

精选留言 (3)

 写留言



delete is create

2020-04-28

感觉kafka中大量使用协调器（findCoordinator）老师协调器到底是做什么的 我今天看一个kafkaAdminClient中获取消费者组信息里面也有这个东西

展开 ∨

作者回复: 协调器是做协调用的:) hmmm... sorry, 哈哈哈

目前Kafka中有两类协调器，GroupCoordinator和TransactionCoordinator。前者用于管理消费者组，后者是用于事务管理的。就以消费者组来说，Coordinator需要生成消费者组的元数据信息，负责维护组的位移提交和获取以及全套的rebalance流程支持。

 1

 1



kai

2020-04-28

请问，如果定位到TotalTimeMs波动比较大，尖峰比较多，而且一般都在5s左右，然后深入发现，时间主要是 reaponse queue time，看了一下网络线程池空闲比例也在80%以上，说明网络线程也不是很忙碌，这个有可能是因为什么原因呢？如何判断集群是否正常？TotalTimeMs 一般如何表现才是正常的？谢谢老师

展开 ∨



delete is create

2020-04-28

老师 使用kafkaAdminClient获取消费者连接状态超时是什么问题？kafka服务器的内存和cpu都占用不高

```
DescribeConsumerGroupsResult result = getAdminClient().describeConsumerGroups(groupIdList);
```

```
Map<String, ConsumerGroupDescription> stringConsumerGroupDescriptionMap...
```

展开 ∨

