24 | 请求是怎么被处理的？

2019-07-27 胡夕

你好，我是胡夕。今天我要和你分享的主题是：Kafka请求是怎么被处理的。

无论是Kafka客户端还是Broker端，它们之间的交互都是通过“请求/响应”的方式完成的。比如，客户端会通过网络发送消息生产请求给Broker，而Broker处理完成后，会发送对应的响应给到客户端。

Apache Kafka自己定义了一组请求协议，用于实现各种各样的交互操作。比如常见的PRODUCE请求是用于生产消息的，FETCH请求是用于消费消息的，METADATA请求是用于请求Kafka集群元数据信息的。

总之，Kafka定义了很多类似的请求格式。我数了一下，截止到目前最新的2.3版本，Kafka共定义了多达45种请求格式。**所有的请求都是通过TCP网络以Socket的方式进行通讯的**。

今天，我们就来详细讨论一下Kafka Broker端处理请求的全流程。

关于如何处理请求，我们很容易想到的方案有两个。

1.顺序处理请求。如果写成伪代码，大概是这个样子：

```
while (true) {
        Request request = accept(connection);
        handle(request);
}
```

这个方法实现简单，但是有个致命的缺陷，那就是**吞吐量太差**。由于只能顺序处理每个请求，因此，每个请求都必须等待前一个请求处理完毕才能得到处理。这种方式只适用于**请求发送非常不频繁的系统**。

2.每个请求使用单独线程处理。也就是说，我们为每个入站请求都创建一个新的线程来异步处理。我们一起来看看这个方案的伪代码。

```
while (true) {
        Request = request = accept(connection);
        Thread thread = new Thread(() -> {
 handle(request);});
        thread.start();
}
```

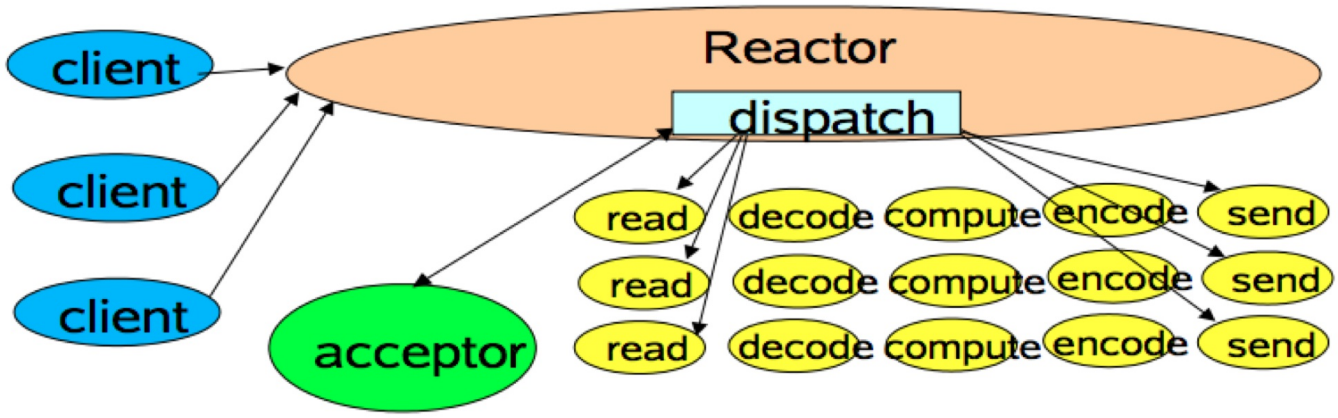
这个方法反其道而行之，完全采用**异步**的方式。系统会为每个入站请求都创建单独的线程来处理。这个方法的好处是，它是完全异步的，每个请求的处理都不会阻塞下一个请求。但缺陷也同样明显。为每个请求都创建线程的做法开销极大，在某些场景下甚至会压垮整个服务。还是那句话，这个方法只适用于请求发送频率很低的业务场景。

既然这两种方案都不好，那么，Kafka是如何处理请求的呢？用一句话概括就是，Kafka使用的是**Reactor模式**。

谈到Reactor模式，大神Doug Lea的“Scalable IO in Java”应该算是最好的入门教材了。即使你没听说过Doug Lea，那你应该也用过ConcurrentHashMap吧？这个类就是这位大神写的。其实，整个java.util.concurrent包都是他的杰作！

好了，我们说回Reactor模式。简单来说，**Reactor模式是事件驱动架构的一种实现方式，特别适合应用于处理多个客户端并发向服务器端发送请求的场景**。我借用Doug Lea的一页PPT来说明一下Reactor的架构，并借此引出Kafka的请求处理模型。

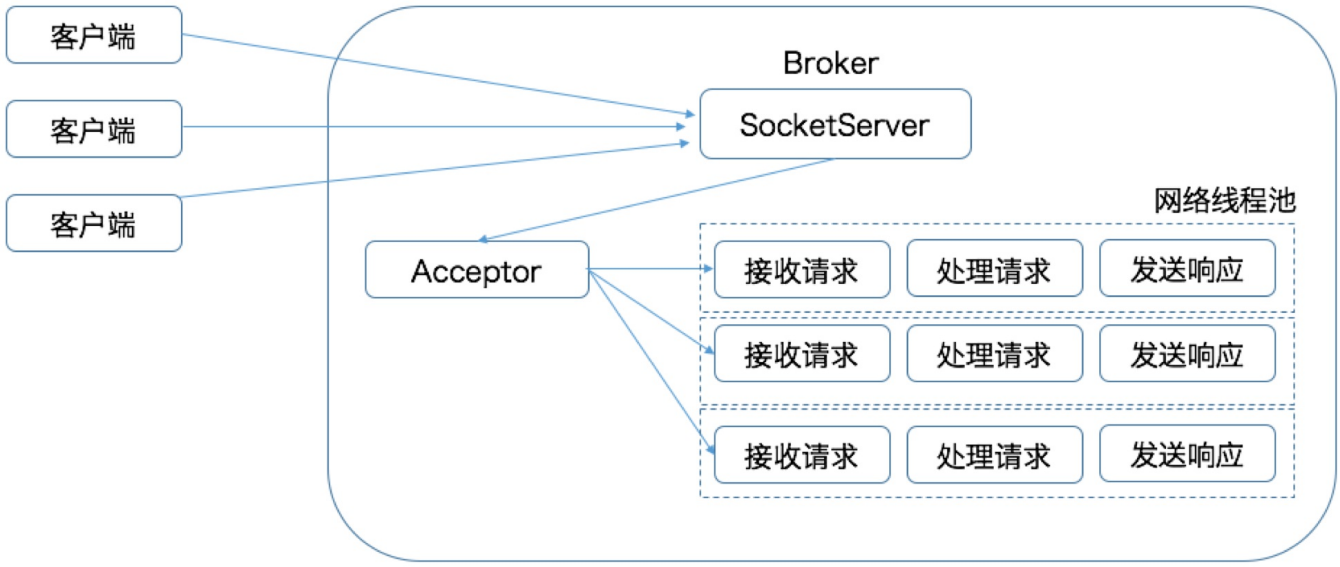
Reactor模式的架构如下图所示：



从这张图中，我们可以发现，多个客户端会发送请求给到Reactor。Reactor有个请求分发线程Dispatcher，也就是图中的Acceptor，它会将不同的请求下发到多个工作线程中处理。

在这个架构中，Acceptor线程只是用于请求分发，不涉及具体的逻辑处理，非常得轻量级，因此有很高的吞吐量表现。而这些工作线程可以根据实际业务处理需要任意增减，从而动态调节系统负载能力。

如果我们来为Kafka画一张类似的图的话，那它应该是这个样子的：

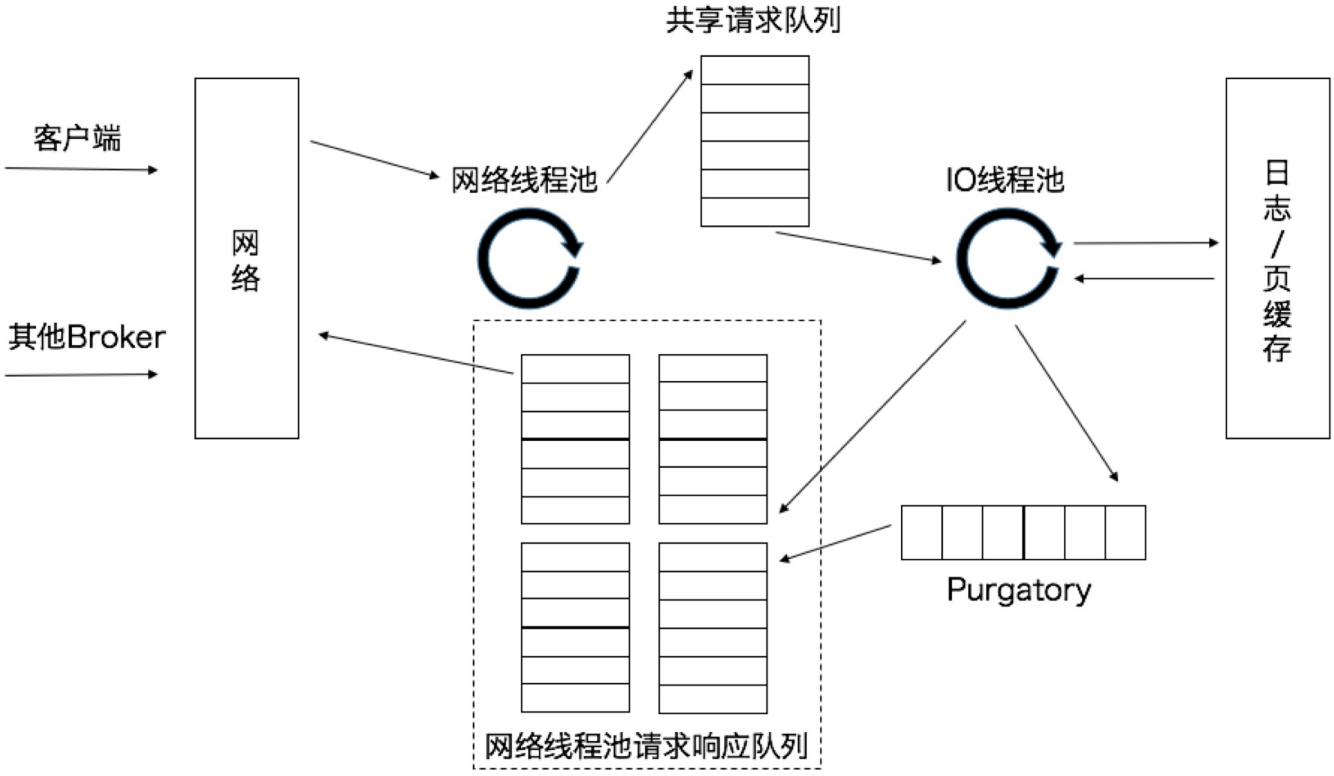


显然，这两张图长得差不多。Kafka的Broker端有个SocketServer组件，类似于Reactor模式中的Dispatcher，它也有对应的Acceptor线程和一个工作线程池，只不过在Kafka中，这个工作线程池有个专属的名字，叫网络线程池。Kafka提供了Broker端参数num.network.threads，用于调整该网络线程池的线程数。其**默认值是3，表示每台Broker启动时会创建3个网络线程，专门处理客户端发送的请求**。

Acceptor线程采用轮询的方式将入站请求公平地发到所有网络线程中，因此，在实际使用过程中，这些线程通常都有相同的几率被分配到待处理请求。这种轮询策略编写简单，同时也避免了请求处理的倾斜，有利于实现较为公平的请求处理调度。

好了，你现在了解了客户端发来的请求会被Broker端的Acceptor线程分发到任意一个网络线程

中，由它们来进行处理。那么，当网络线程接收到请求后，它是怎么处理的呢？你可能会认为，它顺序处理不就好了吗？实际上，Kafka在这个环节又做了一层异步线程池的处理，我们一起来看一看下面这张图。



当网络线程拿到请求后，它不是自己处理，而是将请求放入到一个共享请求队列中。Broker端还有个IO线程池，负责从该队列中取出请求，执行真正的处理。如果是PRODUCE生产请求，则将消息写入到底层的磁盘日志中；如果是FETCH请求，则从磁盘或页缓存中读取消息。

IO线程池处中的线程才是执行请求逻辑的线程。Broker端参数**num.io.threads**控制了这个线程池中的线程数。**目前该参数默认值是8，表示每台Broker启动后自动创建8个IO线程处理请求**。你可以根据实际硬件条件设置此线程池的个数。

比如，如果你的机器上CPU资源非常充裕，你完全可以调大该参数，允许更多的并发请求被同时处理。当IO线程处理完请求后，会将生成的响应发送到网络线程池的响应队列中，然后由对应的网络线程负责将Response返还给客户端。

细心的你一定发现了请求队列和响应队列的差别：**请求队列是所有网络线程共享的，而响应队列则是每个网络线程专属的**。这么设计的原因就在于，Dispatcher只是用于请求分发而不负责响应回传，因此只能让每个网络线程自己发送Response给客户端，所以这些Response也就没必要放在一个公共的地方。

我们再来看看刚刚的那张图，图中有一个叫Purgatory的组件，这是Kafka中著名的“炼狱”组件。它是用来**缓存延时请求**（Delayed Request）的。**所谓延时请求，就是那些一时未满足条件不能立刻处理的请求**。比如设置了acks=all的PRODUCE请求，一旦设置了acks=all，那么该请求就必须等待ISR中所有副本都接收了消息后才能返回，此时处理该请求的IO线程就必须等待其

他Broker的写入结果。当请求不能立刻处理时，它就会暂存在Purgatory中。稍后一旦满足了完成条件，IO线程会继续处理该请求，并将Response放入对应网络线程的响应队列中。

讲到这里，Kafka请求流程解析的故事其实已经讲完了，我相信你应该已经了解了Kafka Broker是如何从头到尾处理请求的。但是我们不会现在就收尾，我要给今天的内容开个小灶，再说点不一样的东西。

到目前为止，我提及的请求处理流程对于所有请求都是适用的，也就是说，Kafka Broker对所有请求是一视同仁的。但是，在Kafka内部，除了客户端发送的PRODUCE请求和FETCH请求之外，还有很多执行其他操作的请求类型，比如负责更新Leader副本、Follower副本以及ISR集合的LeaderAndIsr请求，负责勒令副本下线的StopReplica请求等。与PRODUCE和FETCH请求相比，这些请求有个明显的不同：它们不是数据类的请求，而是控制类的请求。也就是说，它们并不是操作消息数据的，而是用来执行特定的Kafka内部动作的。

Kafka社区把PRODUCE和FETCH这类请求称为数据类请求，把LeaderAndIsr、StopReplica这类请求称为控制类请求。细究起来，当前这种一视同仁的处理方式对控制类请求是不合理的。为什么呢？因为**控制类请求有这样一种能力：它可以直接令数据类请求失效！**

我来举个例子说明一下。假设我们有个主题只有1个分区，该分区配置了两个副本，其中Leader副本保存在Broker 0上，Follower副本保存在Broker 1上。假设Broker 0这台机器积压了很多的PRODUCE请求，此时你如果使用Kafka命令强制将该主题分区的Leader、Follower角色互换，那么Kafka内部的控制器组件（Controller）会发送LeaderAndIsr请求给Broker 0，显式地告诉它，当前它不再是Leader，而是Follower了，而Broker 1上的Follower副本因为被选为新的Leader，因此停止向Broker 0拉取消息。

这时，一个尴尬的场面就出现了：如果刚才积压的PRODUCE请求都设置了acks=all，那么这些在LeaderAndIsr发送之前的请求就都无法正常完成了。就像前面说的，它们会被暂存在Purgatory中不断重试，直到最终请求超时返回给客户端。

设想一下，如果Kafka能够优先处理LeaderAndIsr请求，Broker 0就会立刻抛出NOT_LEADER_FOR_PARTITION异常，快速地标识这些积压PRODUCE请求已失败，这样客户端不用等到Purgatory中的请求超时就能立刻感知，从而降低了请求的处理时间。即使acks不是all，积压的PRODUCE请求能够成功写入Leader副本的日志，但处理LeaderAndIsr之后，Broker 0上的Leader变为了Follower副本，也要执行显式的日志截断（Log Truncation，即原Leader副本成为Follower后，会将之前写入但未提交的消息全部删除），依 然做了很多无用功。

再举一个例子，同样是在积压大量数据类请求的Broker上，当你删除主题的时候，Kafka控制器（我会在专栏后面的内容中专门介绍它）向该Broker发送StopReplica请求。如果该请求不能及时处理，主题删除操作会一直hang住，从而增加了删除主题的延时。

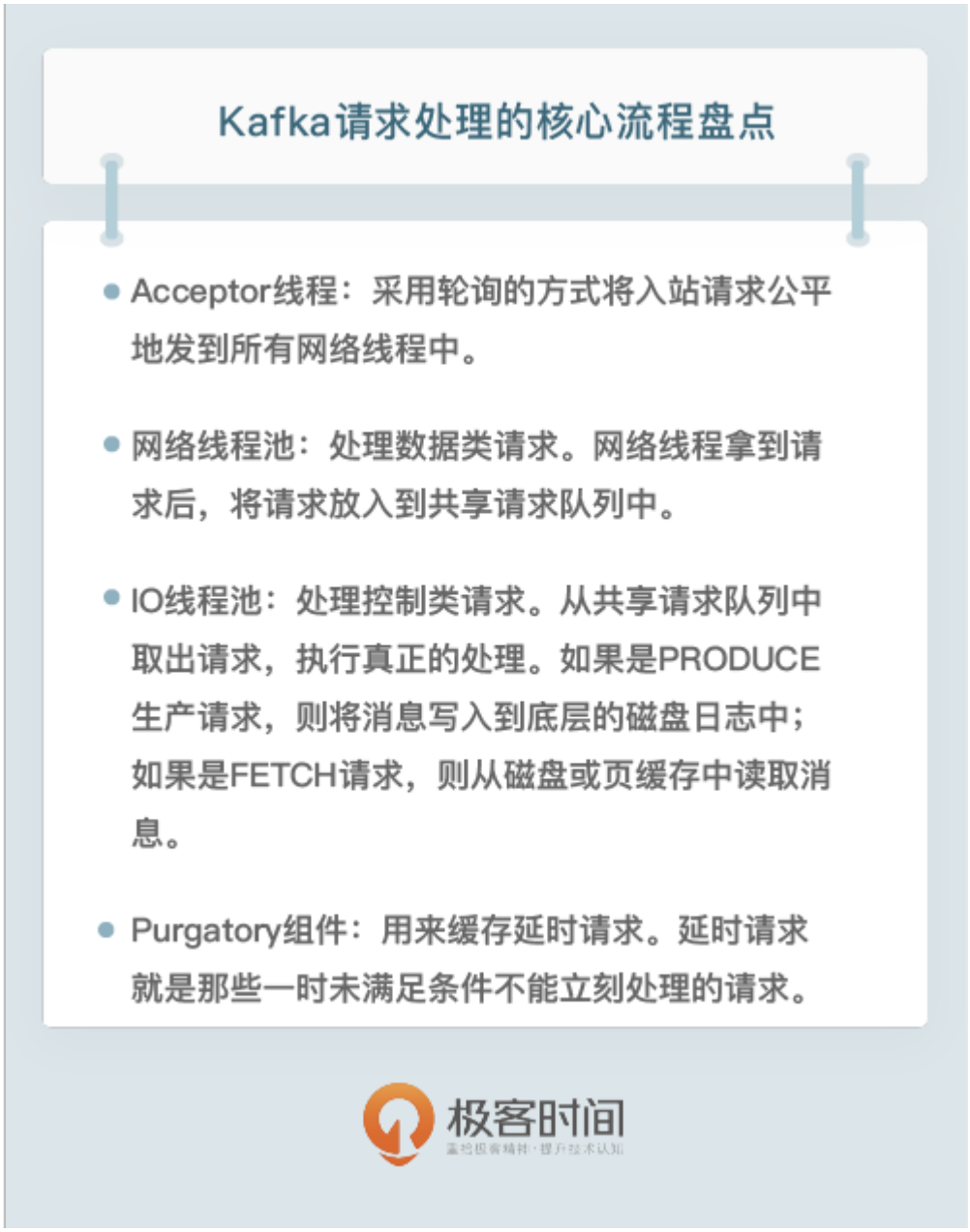
基于这些问题，社区于2.3版本正式实现了数据类请求和控制类请求的分离。其实，在社区推出方案之前，我自己尝试过修改这个设计。当时我的想法是，在Broker中实现一个优先级队列，并赋予控制类请求更高的优先级。这是很自然的想法，所以我本以为社区也会这么实现的，但后来我这个方案被清晰地记录在“已拒绝方案”列表中。

究其原因，这个方案最大的问题在于，它无法处理请求队列已满的情形。当请求队列已经无法容纳任何新的请求时，纵然有优先级之分，它也无法处理新的控制类请求了。

那么，社区是如何解决的呢？很简单，你可以再看一遍今天的第三张图，社区完全拷贝了这张图中的一套组件，实现了两类请求的分离。也就是说，Kafka Broker启动后，会在后台分别两套创建网络线程池和IO线程池，它们分别处理数据类请求和控制类请求。至于所用的Socket端口，自然是使用不同的端口了，你需要提供不同的**listeners配置**，显式地指定哪套端口用于处理哪类请求。

小结

讲到这里，Kafka Broker请求处理流程的解析应该讲得比较完整了。明确请求处理过程的最大意义在于，它是你日后执行Kafka性能优化的前提条件。如果你能从请求的维度去思考Kafka的工作原理，你会发现，优化Kafka并不是一件困难的事情。



开放讨论

坦白来讲，我对社区否定优先级队列方案是有一点不甘心的。如果是你的话，你觉得应该如何规避优先级队列方案中队列已满的问题呢？

欢迎写下你的思考和答案，我们一起讨论。如果你觉得有所收获，也欢迎把文章分享给你的朋友。

精选留言

lmtoo 3

小结部分的图片把数据类请求放到了网络线程池中，而控制类请求放到了IO线程池，弄反了吧；我觉得社区的决定是正确的，这两类请求分离之后，职责更明确了

2019-07-27

cricket1981 3

双队列设计，分别存放数据类和控制类请求，每次先处理完所有控制类请求再处理数据类请求。

2019-07-27

ban 1

老师，社区完全拷贝了这张图中的一套组件，实现了两类请求的分离。也就是说，Kafka Broker 启动后，会在后台分别创建网络线程池和 IO 线程池，它们分别处理数据类请求和控制类请求。

上面这段话不太懂，意思是说：分别建立两套组件（A套 网络线程池IO线程池：负责处理数据类请求)、（B套 网络线程池IO线程池：负责处理控制类请求),这样理解对吗？

2019-07-28

作者回复

嗯嗯，差不多是这个意思

2019-07-29

电光火石 1

优先级队列方案，可以开两个队列，分别处理，前面的监听端口不需要重新构建，只是后面的处理线程不同即可。

另外，想问一下：

1. 为什么当时kafka做的时候，没有考虑使用netty作为通信框架？

2. 对IO这一块的处理比较感兴趣，老师可以介绍一下broker的入口类吗，想去看一下源码

谢谢了！

2019-07-27

作者回复

1. Kafka社区当初主要是为了jar依赖的问题而选择不使用netty，转而使用Java NIO的

2. Broker入口类是kafka.server.KafkaServer.scala

2019-07-29

锦 0

我理解Acceptor是用来接收连接的（三次握手），连接成功后把读写请求的Socket提交到网络线程池，网络线程池中的线程通过Selector收到读请求后，从内核读取消息数据，然后再把待处理消息数据放入共享请求队列中。共享请求队列应该是多生产者多消费者模式（这里如何设计比较关键）。io线程池从共享请求队列中取出消息处理，处理完成再把响应提交到网络线程池中，由网络线程池发送至客户端。这里的共享请求队列为什么不直接使用io线程池自带的工作队列呢？另外控制类请求单独走不同线程池处理比较合理。

2019-07-30

WL 0

再请问一下老师, IO线程池为啥不涉及成3个共享队列, 一个是写请求共享队列,一个读请求共享队列, 一个控制类共享队列, 这样写消息共享队列因为是顺序写, 所以只用一个线程一直写就可以了, 这样还不需要线程上下文切换; 读请求共享队列因为可能不同消费者组的消费者消费进度不同可以有多个线程, 这样吧读写分开我感觉请求的处理效率可以进一步提高

2019-07-30

WL 0

请问老师两个问题:

1. 网络线程池是不是在响应客户端上起到的作用更大, 我感觉在请求接受上, networkthread只是把请求放入共享请求队列, 是一个线程放还是多个线程放好像效率上没提高多少.

2. 如果是网络线程池是多个线程同时向共享队列里插入请求,那么怎么保证消息被顺序处理, 很可能后面的消息因为异步的原因先于前面的消息放入共享请求队列

2019-07-30

Hello world 0

老师，我理解Acceptor线程是分发请求给网络线程，而网络线程接收到请求再放入请求队列。Acceptor线程只是负责转发请求，压力不大，既然网络线程其实也是相当于转发请求，为啥还要有这个网络线程呢？

2019-07-29

作者回复

Processor线程也不是啥都不做，它有很多要处理的事情，比如执行网络层的请求/响应发送，这些都是后面API线程或请求处理线程做不了也不应该做的。

2019-07-30

Sunney 0

老师您好，这两天做项目遇到一个问题想咨询一下，对于网络摄像头的视频流数据和抓拍到的照片数据，kafka应该如何传输呢？

2019-07-29

作者回复

相同的方法，都要传输字节数组。你需要找到合适的方法把你的视频流数据或照片编码成字节序列。当然Kafka其实并不适合传输特别大的消息，因此你可以评估一下是否真的需求传视频本身？

2019-07-29

曾轼麟 0

补充一下前面的留言，外层优先队列只按照请求类来保证优先级，如果每次同类型的请求都有优先级的话，我建议再加一层同类型的内层优先队列，然后在这里面拉出链表，不过实现起来会稍微有点麻烦

2019-07-28

曾轼麟 0

继续使用优先队列，但是每个队列的节点都是一个node，允许有卫星数据（比如一个链表的引用），当同样等级或者类型的请求可以效仿hashmap发生hash冲突那样拉出一条链表，保存到堆里面，jvm允许的话这样就能拓展不会出现队列满的情况，同时保持着优先队列的特点

2019-07-28

曾轼麟 0

终于明白为什么broker可以配置两个listener了，那时候我看着官网还挺奇怪的

2019-07-28

明翼 0

有两种方法：1 是直接替换数据处理队列中的最前面的数据进行处理，处理完控制队列，再将这个消息插队到队头；2 双队列设计，不过双队列，如果先处理控制消息，如果一直来控制消息，数据队列的消息岂不会被延迟很大；

关于复制一套，我看了下面评论，我和部分网友的理解不一样，我觉得是复制一套网络线程持+中间队列+IO线程池；也就是有两个网络线程池，+2个中间队列，和2套IO线程持；

网络线程池作用将数据分发到中间队列，和接受IO线程池的处理结果回复给客户端。我理解为什么要加这个中间队列是为了将网络处理的线程数和IO处理的线程数解耦，达到高性能和资源少占用的目的。

2019-07-28

作者回复

我觉得不错：）

2019-07-29

rm -rf

0

两者只是分开了，但也没解决上文说的，控制类请求比数据类请求优先的需求呀？希望老师解答一下。

另外，总结图写反了吧，网络线程池应该是处理控制类，IO线程池应该才是处理数据类请求吧？

2019-07-27

作者回复

不是。每类请求都有对应的网络线城市和IO线程池

2019-07-29

A_F

0

共享请求队列的作用是为了缓解上下游的压力吗？

2019-07-27

cricket1981

0

规避优先级队列方案中队列已满的问题可以考虑将队列中后进来的数据类请求清退到另一个队列或持久化到文件，以腾出队列空间给到控制类请求，待队列请求处理完再将其加载回原队列。

2019-07-27

Riordon

0

数据类请求和控制类请求的分离，我理解的是多开一套端口，实现一套网络线程池+IO线程池？不对吗？

Acceptor线程：公平转发请求到网络线程

网络线程池：将请求放入共享队列

IO线程池：从共享队列取出请求，执行真正的IO

2019-07-27

cricket1981

0

从文章介绍来看社区方案也只是分开处理数据类和控制类请求，并无控制类优先于数据类处理逻辑啊

2019-07-27

作者回复

嗯嗯，从某种意义上只是分离，确实不是我们认为的优先级处理，但总归是能解决之前碰到的那些问题不是：）

2019-07-30

leaning_人生

0

期待下一讲

2019-07-27