08 | 网络通信优化之I/O模型：如何解决高并发下I/O瓶颈？

2019-06-06 刘超

你好，我是刘超。

提到Java I/O，相信你一定不陌生。你可能使用I/O操作读写文件，也可能使用它实现Socket的信息传输..这些都是我们在系统中最常遇到的和I/O有关的操作。

我们都知道，I/O的速度要比内存速度慢，尤其是在现在这个大数据时代背景下，I/O的性能问题更是尤为突出，I/O读写已经成为很多应用场景下的系统性能瓶颈，不容我们忽视。

今天，我们就来深入了解下Java I/O在高并发、大数据业务场景下暴露出的性能问题，从源头入手，学习优化方法。

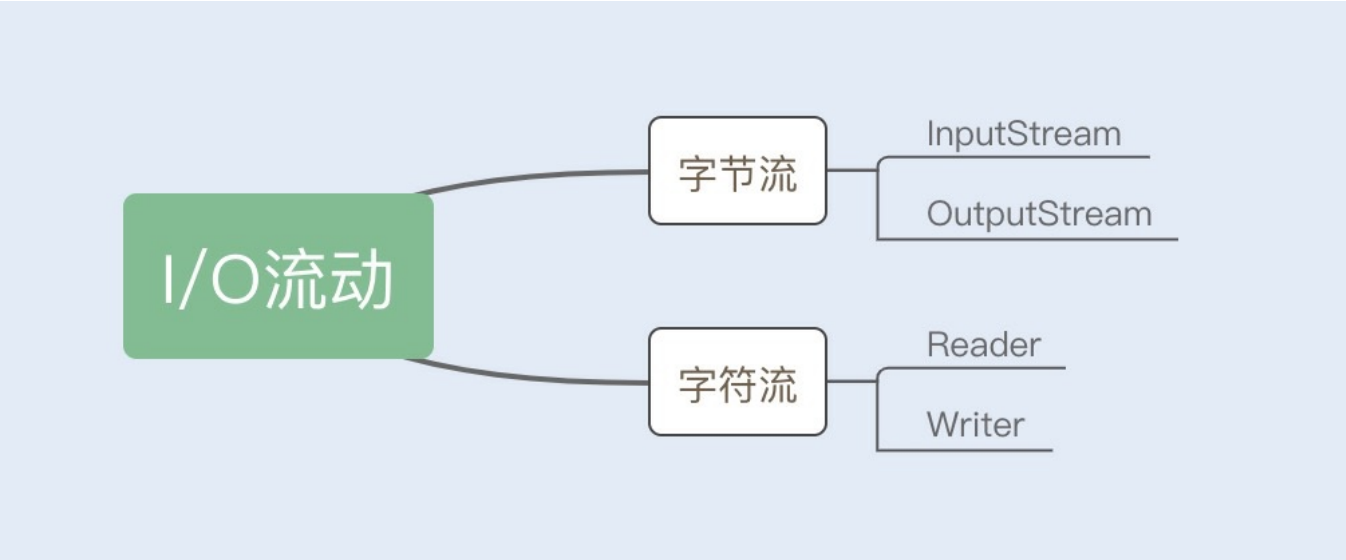
什么是I/O

I/O是机器获取和交换信息的主要渠道，而流是完成I/O操作的主要方式。

在计算机中，流是一种信息的转换。流是有序的，因此相对于某一机器或者应用程序而言，我们通常把机器或者应用程序接收外界的信息称为输入流（InputStream），从机器或者应用程序向外输出的信息称为输出流（OutputStream），合称为输入/输出流（I/O Streams）。

机器间或程序间在进行信息交换或者数据交换时，总是先将对象或数据转换为某种形式的流，再通过流的传输，到达指定机器或程序后，再将流转换为对象数据。因此，流就可以被看作是一种数据的载体，通过它可以实现数据交换和传输。

Java的I/O操作类在包java.io下，其中InputStream、OutputStream以及Reader、Writer类是I/O包中的4个基本类，它们分别处理字节流和字符流。如下图所示：

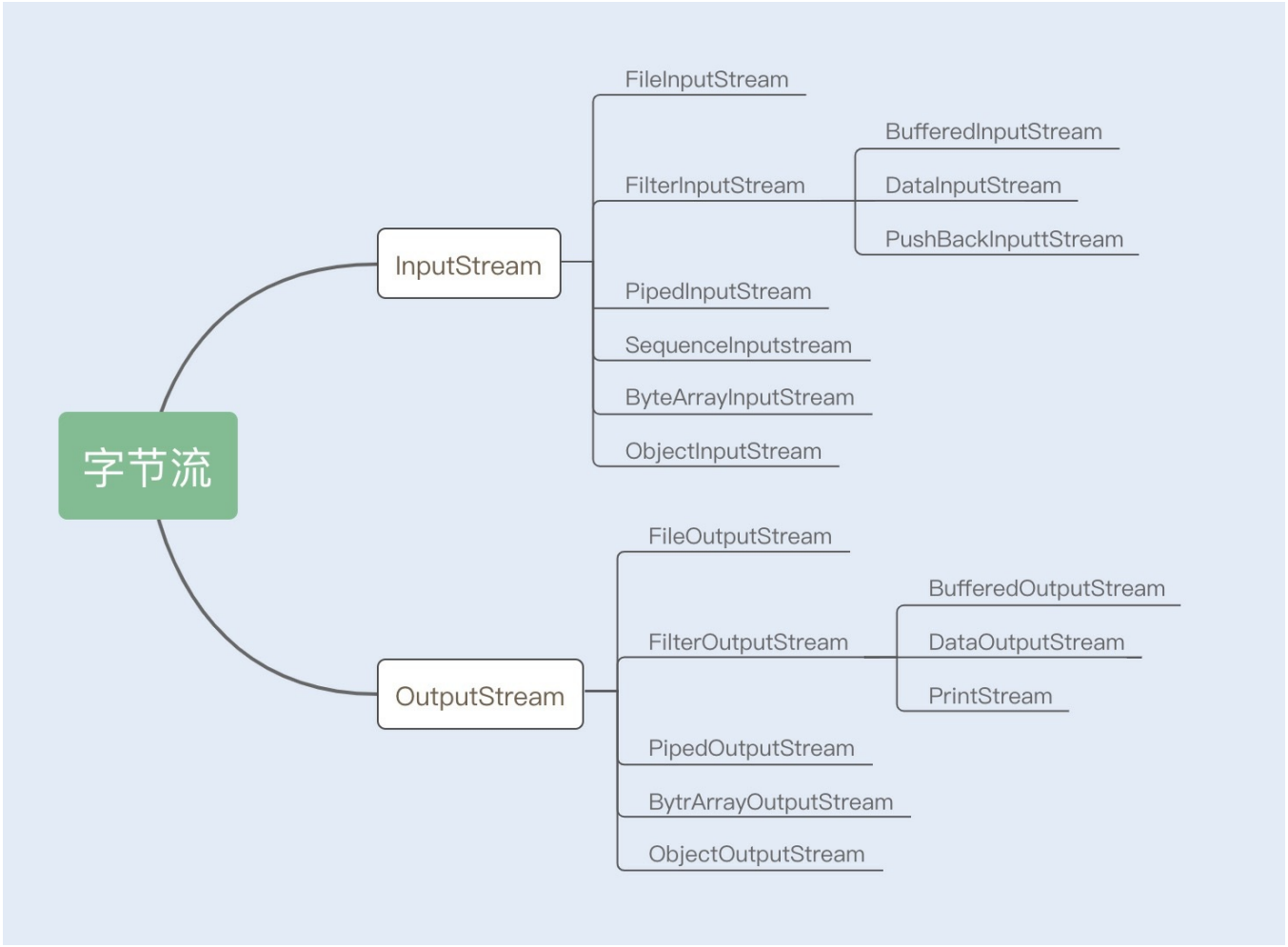


回顾我的经历，我记得在初次阅读Java I/O流文档的时候，我有过这样一个疑问，在这里也分享给你，那就是：“**不管是文件读写还是网络发送接收，信息的最小存储单元都是字节，那为什么I/O流操作要分为字节流操作和字符流操作呢？**”

我们知道字符到字节必须经过转码，这个过程非常耗时，如果我们不知道编码类型就很容易出现乱码问题。所以I/O流提供了一个直接操作字符的接口，方便我们平时对字符进行流操作。下面我们就分别了解下“字节流”和“字符流”。

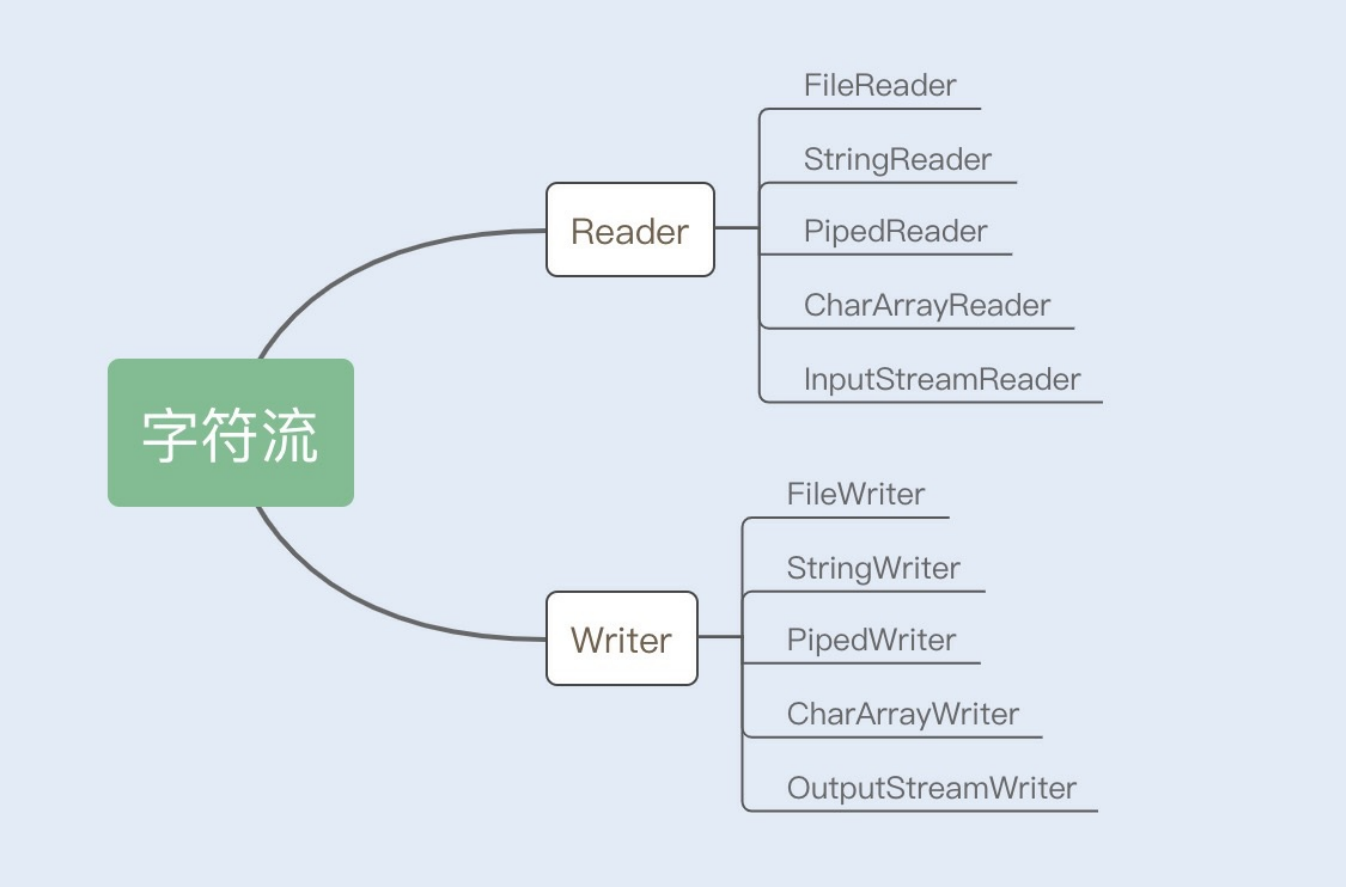
1.字节流

InputStream/OutputStream是字节流的抽象类，这两个抽象类又派生出了若干子类，不同的子类分别处理不同的操作类型。如果是文件的读写操作，就使用FileInputStream/FileOutputStream；如果是数组的读写操作，就使用ByteArrayInputStream/ByteArrayOutputStream；如果是普通字符串的读写操作，就使用BufferedInputStream/BufferedOutputStream。具体内容如下图所示：



2.字符流

Reader/Writer是字符流的抽象类，这两个抽象类也派生出了若干子类，不同的子类分别处理不同的操作类型，具体内容如下图所示：

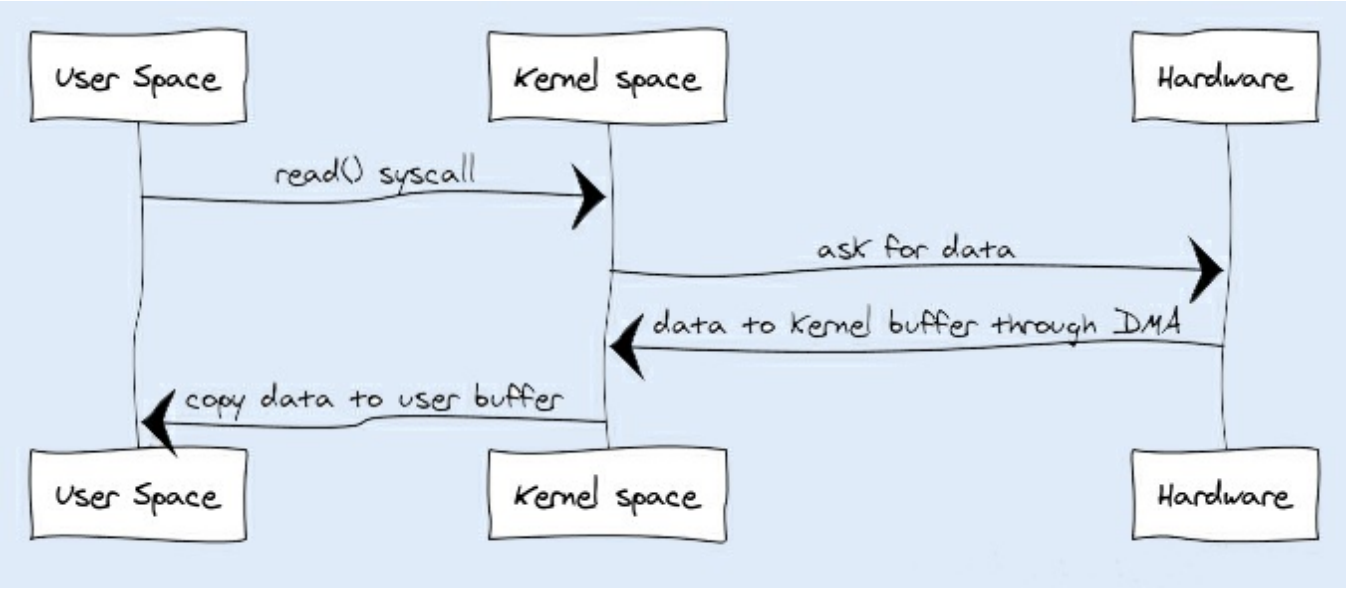


传统I/O的性能问题

我们知道，I/O操作分为磁盘I/O操作和网络I/O操作。前者是从磁盘中读取数据源输入到内存中，之后将读取的信息持久化输出在物理磁盘上；后者是从网络中读取信息输入到内存，最终将信息输出到网络中。但不管是磁盘I/O还是网络I/O，在传统I/O中都存在严重的性能问题。

1.多次内存复制

在传统I/O中，我们可以通过InputStream从源数据中读取数据流输入到缓冲区里，通过OutputStream将数据输出到外部设备（包括磁盘、网络）。你可以先看下输入操作在操作系统中的具体流程，如下图所示：



- JVM会发出read()系统调用，并通过read系统调用向内核发起读请求；
- 内核向硬件发送读指令，并等待读就绪；
- 内核把将要读取的数据复制到指向的内核缓存中；
- 操作系统内核将数据复制到用户空间缓冲区，然后read系统调用返回。

在这个过程中，数据先从外部设备复制到内核空间，再从内核空间复制到用户空间，这就发生了两次内存复制操作。这种操作会导致不必要的数据拷贝和上下文切换，从而降低I/O的性能。

2.阻塞

在传统I/O中，InputStream的read()是一个while循环操作，它会一直等待数据读取，直到数据就绪才会返回。**这就意味着如果没有数据就绪，这个读取操作将会一直被挂起，用户线程将会处于阻塞状态。**

在少量连接请求的情况下，使用这种方式没有问题，响应速度也很高。但在发生大量连接请求时，就需要创建大量监听线程，这时如果线程没有数据就绪就会被挂起，然后进入阻塞状态。一旦发生线程阻塞，这些线程将会不断地抢夺CPU资源，从而导致大量的CPU上下文切换，增加系统的性能开销。

如何优化I/O操作

面对以上两个性能问题，不仅编程语言对此做了优化，各个操作系统也进一步优化了I/O。JDK1.4发布了java.nio包（new I/O的缩写），NIO的发布优化了内存复制以及阻塞导致的严重性能问题。JDK1.7又发布了NIO2，提出了从操作系统层面实现的异步I/O。下面我们就来了解下具体的优化实现。

1.使用缓冲区优化读写流操作

在传统I/O中，提供了基于流的I/O实现，即InputStream和OutputStream，这种基于流的实现以字节为单位处理数据。

NIO与传统 I/O 不同，它是基于块（Block）的，它以块为基本单位处理数据。在NIO中，最为重要的两个组件是缓冲区（Buffer）和通道（Channel）。Buffer是一块连续的内存块，是 NIO 读写数据的中转地。Channel表示缓冲数据的源头或者目的地，它用于读取缓冲或者写入数据，是访问缓冲的接口。

传统I/O和NIO的最大区别就是传统I/O是面向流，NIO是面向Buffer。Buffer可以将文件一次性读入内存再做后续处理，而传统的方式是边读文件边处理数据。虽然传统I/O后面也使用了缓冲块，例如BufferedInputStream，但仍然不能和NIO相媲美。使用NIO替代传统I/O操作，可以提升系统的整体性能，效果立竿见影。

2.使用DirectBuffer减少内存复制

NIO的Buffer除了做了缓冲块优化之外，还提供了一个可以直接访问物理内存的类DirectBuffer。

普通的Buffer分配的是JVM堆内存，而DirectBuffer是直接分配物理内存。

我们知道数据要输出到外部设备，必须先从用户空间复制到内核空间，再复制到输出设备，而DirectBuffer则是直接将步骤简化为从内核空间复制到外部设备，减少了数据拷贝。

这里拓展一点，由于DirectBuffer申请的是非JVM的物理内存，所以创建和销毁的代价很高。DirectBuffer申请的内存并不是直接由JVM负责垃圾回收，但在DirectBuffer包装类被回收时，会通过Java Reference机制来释放该内存块。

3.避免阻塞，优化I/O操作

NIO很多人也称之为Non-block I/O，即非阻塞I/O，因为这样叫，更能体现它的特点。为什么这么说呢？

传统的I/O即使使用了缓冲块，依然存在阻塞问题。由于线程池线程数量有限，一旦发生大量并发请求，超过最大数量的线程就只能等待，直到线程池中有空闲的线程可以被复用。而对Socket的输入流进行读取时，读取流会一直阻塞，直到发生以下三种情况的任意一种才会解除阻塞：

- 有数据可读；
- 连接释放；
- 空指针或I/O异常。

阻塞问题，就是传统I/O最大的弊端。NIO发布后，通道和多路复用器这两个基本组件实现了NIO的非阻塞，下面我们就一起来了解下这两个组件的优化原理。

通道（Channel）

前面我们讨论过，传统I/O的数据读取和写入是从用户空间到内核空间来回复制，而内核空间的数据是通过操作系统层面的I/O接口从磁盘读取或写入。

最开始，在应用程序调用操作系统I/O接口时，是由CPU完成分配，这种方式最大的问题是“发生大量I/O请求时，非常消耗CPU“；之后，操作系统引入了DMA（直接存储器存储），内核空间与磁盘之间的存取完全由DMA负责，但这种方式依然需要向CPU申请权限，且需要借助DMA总线来完成数据的复制操作，如果DMA总线过多，就会造成总线冲突。

通道的出现解决了以上问题，Channel有自己的处理器，可以完成内核空间和磁盘之间的I/O操作。在NIO中，我们读取和写入数据都要通过Channel，由于Channel是双向的，所以读、写可以同时进行。

多路复用器（Selector）

Selector是Java NIO编程的基础。用于检查一个或多个NIO Channel的状态是否处于可读、可

写。

Selector是基于事件驱动实现的，我们可以在Selector中注册accpet、read监听事件，Selector会不断轮询注册在其上的Channel，如果某个Channel上面发生监听事件，这个Channel就处于就绪状态，然后进行I/O操作。

一个线程使用一个Selector，通过轮询的方式，可以监听多个Channel上的事件。我们可以在注册Channel时设置该通道为非阻塞，当Channel上没有I/O操作时，该线程就不会一直等待了，而是会不断轮询所有Channel，从而避免发生阻塞。

目前操作系统的I/O多路复用机制都使用了epoll，相比传统的select机制，epoll没有最大连接句柄1024的限制。所以Selector在理论上可以轮询成千上万的客户端。

下面我用一个生活化的场景来举例，看完你就更清楚Channel和Selector在非阻塞I/O中承担什么角色，发挥什么作用了。

我们可以把监听多个I/O连接请求比作一个火车站的进站口。以前检票只能让搭乘就近一趟发车的旅客提前进站，而且只有一个检票员，这时如果有其他车次的旅客要进站，就只能在站口排队。这就相当于最早没有实现线程池的I/O操作。

后来火车站升级了，多了几个检票入口，允许不同车次的旅客从各自对应的检票入口进站。这就相当于用多线程创建了多个监听线程，同时监听各个客户端的I/O请求。

最后火车站进行了升级改造，可以容纳更多旅客了，每个车次载客更多了，而且车次也安排合理，乘客不再扎堆排队，可以从一个大的统一的检票口进站了，这一个检票口可以同时检票多个车次。这个大的检票口就相当于Selector，车次就相当于Channel，旅客就相当于I/O流。

总结

Java的传统I/O开始是基于InputStream和OutputStream两个操作流实现的，这种流操作是以字节为单位，如果在高并发、大数据场景中，很容易导致阻塞，因此这种操作的性能是非常差的。还有，输出数据从用户空间复制到内核空间，再复制到输出设备，这样的操作会增加系统的性能开销。

传统I/O后来使用了Buffer优化了“阻塞”这个性能问题，以缓冲块作为最小单位，但相比整体性能来说依然不尽人意。

于是NIO发布，它是基于缓冲块为单位的流操作，在Buffer的基础上，新增了两个组件“管道和多路复用器”，实现了非阻塞I/O，NIO适用于发生大量I/O连接请求的场景，这三个组件共同提升了I/O的整体性能。

你可以在Github上通过几个简单的例子来实践下传统IO、NIO。

思考题

在JDK1.7版本中，Java发布了NIO的升级包NIO2，也就是AIO。AIO实现了真正意义上的异步I/O，它是直接将I/O操作交给操作系统进行异步处理。这也是对I/O操作的一种优化，那为什么现在很多容器的通信框架都还是使用NIO呢？

期待在留言区看到你的见解。也欢迎你点击“请朋友读”，把今天的内容分享给身边的朋友，邀请他一起学习。

精选留言

皮皮 10

老师，个人觉得本期的内容讲的稍微浅了一点，关于IO的几种常见模型可以配图讲一下的，另外就是linux下的select，poll，epoll的对比应用场景。最重要的目前用的最多的IO多路复用可以深入讲一下的。

2019-06-06

作者回复

你好，这篇I/O性能优化主要是普及NIO对I/O的性能优化。I/O这块的知识点很多，包括IO模型、事件处理模型以及操作系统层的事件驱动，如果都压缩到一讲，由于字数有限，很难讲完整。对于一些童鞋来说，也不好理解。

我将会在后面的一讲中，补充大家提到的一些内容。谢谢你的建议。

2019-06-06

张学磊

4

在Linux中，AIO并未真正使用操作系统所提供的异步I/O，它仍然使用poll或epoll，并将API封装为异步I/O的样子，但是其本质仍然是同步非阻塞I/O，加上第三方产品的出现，Java网络编程明显落后，所以没有成为主流

2019-06-06

作者回复

对的，异步I/O模型在Linux内核中没有实现

2019-06-10

Geek_8043c2

3

很多知识linux 网络 I/O模型底层原理，零拷贝技术等深入讲一下，毕竟学Java性能调优的学员都是有几年工作经验的， 希望老师后面能专门针对这次io 出个补充，这一讲比较不够深入。

2019-06-07

作者回复

这一讲中提到了DirectBuffer，也就是零拷贝的实现。谢谢你的建议，后面我会补充下几种网络I/O模型的底层原理。

2019-06-10

Only now

2

老师能不能讲讲DMA和Channel的区别, DMA需要占用总线, 那么Channel是如何跳过总线向内存传输数据的?

2019-06-06

作者回复

一个设备接口试图通过总线直接向外部设备(磁盘)传送数据时，它会先向CPU发送DMA请求信号。外部设备(磁盘)通过DMA的一种专门接口电路——DMA控制器（DMAC），向CPU提出接管总线控制权的总线请求，CPU收到该信号后，在当前的总线周期结束后，会按DMA信号的优先级和提出DMA请求的先后顺序响应DMA信号。CPU对某个设备接口响应DMA请求时，会让出总线控制权。于是在DMA控制器的管理下，磁盘和存储器直接进行数据交换，而不需CPU干预。数据传送完毕后，设备接口会向CPU发送DMA结束信号，交还总线控制权。

而通道则是在DMA的基础上增加了能执行有限通道指令的I/O控制器，代替CPU管理控制外设。通道有自己的指令系统，是一个协处理器，他实质是一台能够执行有限的输入输出指令，并且有专门通讯传输的通道总线完成控制。

2019-06-07

vvsuperman

1

建议加写例子，比如tomcat用的io造成阻塞之类，实例分析等

2019-06-06

编辑回复

感谢这位同学的建议，老师会在11讲中集中补充有关IO的一些实战内容。

2019-06-06

Geek_37bdff 0

老师，您好，能通俗解释一下什么是同步阻塞，异步阻塞，同步非阻塞，异步非阻塞这些概念不，还有就是nio是属于同步非阻塞还是异步非阻塞，为什么

2019-06-12

编辑回复

同学你好！周四即将更新的11讲答疑课堂就能解决你的问题。到时如有疑问，可以继续给老师留言。

2019-06-12

知行合一 0

思考题：是因为会很耗费cpu吗？

2019-06-10

作者回复

答案已经有同学给出了，异步I/O没有在Linux内核中实现

2019-06-10

z.l 0

从文章的描述我猜测DirectBuffer属于内核空间的内存，但java作为用户进城是如何操作内核空间内存的呢？

2019-06-09

Geek_801517 0

老师，我也觉得今天的内容浅了一点，nio的多路复用也可以深入讲下或者netty的实现，epoll这些也是，还有其他io模型也可以对比下

2019-06-09

编辑回复

收到～老师会集中大家的留言，在11讲答疑课堂中做出补充讲解。感谢你的建议！

2019-06-09

晓杰 0

老师，channel只是解决了内核空间和磁盘之前的io操作问题，那用户空间和内核空间之间的来回复制是不是依然是一个耗时的操作

2019-06-07

作者回复

这讲中提到了零拷贝，用DirectBuffer减少内存复制，也就是避免了用户空间与内核空间来回复制。

2019-06-10

-W.LI- 0

老师好!能说下哪些操作需要在用户态下完成么?正常的代码运行用户态就可以了是吗?

1.创建selector

2.创建serverSockekChannel
3.OP_ACCEPT事件注册到selector中
4.监听到OP_ACCEPT事件
5.创建channel
6.OP_READ事件注册到selector中
7.监听到READ事件
8.从channel中读数据
读的时候需要先切换到内核模式，复制在内核空间，然后从内核空间复制到用户空间。
9.处理数据
10.write:用户模式切换到内核模式，从用户空间复制到内核空间，再从内核空间发送到IO网络上。
1-7步里面有哪些操作需要在内核模式下执行的么？

第8和10我是不是理解错了？
DMA啥时候起作用啊？

JVM的内存属于用户空间是吧，directBuffer直接申请的物理内存，是属于特殊的用户空间么。内核模式直接往那里写。kafka的0拷贝和directbuffer一个意思么？╮ ＿＿ ╰都不知道

2019-06-06

圣西罗 0

使用webflux的过程中最大的不方便是不支持threadlocal,导致像创建人修改人id的赋值需要明传参数

2019-06-06

胖妞 0

git上的测试案例有吗？想很多通过时间具体对比一下！总感觉讲的有点抽象和概念了，脑子里没有形成一个具体的形象！希望能给几个小demo看一下！麻烦了！

2019-06-06

作者回复

git上有源码，分别是io和nio的简单实现的demo。如果需要通过简单的代码测试比对两者的性能，可以自己尝试一下，有疑问可以再问老师。

2019-06-06