字节二面：为何还执着传统数据复制，零拷贝它不香吗？

原创 往事敬秋风 深度Linux 2025年04月12日 09:11 湖南

在当今这个数据爆炸的时代，无论是互联网巨头的海量数据处理，还是新兴创业公司对高效性能的追求，数据传输效率都成了决定系统成败的关键因素。技术面试作为筛选人才的重要关卡，也越发关注候选人对前沿技术的掌握程度。字节跳动，作为行业内的技术先锋，在面试中更是不断抛出极具挑战性的问题，以选拔出真正的技术高手。

“字节二面：还要使用传统数据复制，为什么不用零拷贝？”当这个问题摆在面试者面前时，看似简单的询问背后，实则隐藏着对候选人技术深度和广度的全方位考察。它不仅要求你对零拷贝技术的原理了如指掌，更需要你能清晰洞察传统数据复制在现代技术体系中的位置，以及在不同场景下二者的优劣抉择。

对于每一位渴望在技术领域崭露头角的开发者而言，理解这个问题，就如同掌握了一把通往高效数据处理世界的钥匙。接下来，就让我们一同深入剖析传统数据复制与零拷贝技术的本质区别，探寻在字节跳动这样的技术驱动型企业中，如何通过技术选型实现系统性能的飞跃 。

一、传统 I/O 的困境

在深入了解零拷贝之前，让我们先来看一下传统的 I/O 数据传输方式。想象一下，你正在从服务器上下载一个文件，这个看似简单的操作背后，其实涉及到了一系列复杂的数据传输过程。

当你发起下载请求时，操作系统会先从磁盘中读取文件数据。这个过程中，数据首先会被读取到内核缓冲区，然后再被拷贝到用户空间的应用程序缓冲区。接着，应用程序将数据发送到网络，数据又会从用户空间缓冲区拷贝到内核空间的 socket 缓冲区，最后通过网卡发送出去。

具体来说，传统 I/O 的数据传输过程如下：

- **用户态到内核态切换**：应用程序调用 read 系统调用，请求读取文件数据。此时，CPU 从用户态切换到内核态，开始执行内核中的代码。
- **磁盘数据读取到内核缓冲区**：内核通过 DMA（直接内存访问）技术，将磁盘数据直接拷贝到内核缓冲区。DMA 技术可以让硬件设备（如磁盘控制器）直接访问内存，而不需要 CPU 的干预，从而减轻 CPU 的负担。

- **内核缓冲区数据拷贝到用户缓冲区**：数据从内核缓冲区拷贝到用户空间的应用程序缓冲区。这一步需要 CPU 的参与，因为用户空间和内核空间是相互隔离的，数据不能直接在两者之间传递。
- **内核态到用户态切换**：read 系统调用返回，CPU 从内核态切换回用户态，应用程序可以处理用户缓冲区中的数据。
- **用户态到内核态切换**：应用程序调用 write 系统调用，请求将数据发送到网络。CPU 再次从用户态切换到内核态。
- **用户缓冲区数据拷贝到 socket 缓冲区**：数据从用户缓冲区拷贝到内核空间的 socket 缓冲区，准备通过网络发送出去。这一步同样需要 CPU 的参与。
- **socket 缓冲区数据发送到网卡**：内核通过 DMA 技术，将 socket 缓冲区中的数据拷贝到网卡缓冲区，然后通过网络发送出去。
- **内核态到用户态切换**：write 系统调用返回，CPU 从内核态切换回用户态，数据传输完成。

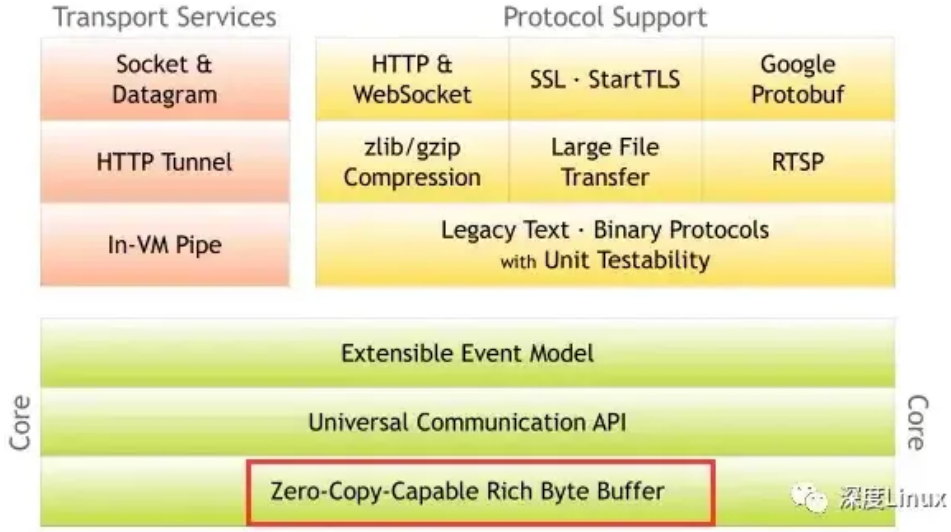
从上述过程可以看出，传统 I/O 在一次简单的文件传输中，就涉及了 4 次用户态与内核态的上下文切换，以及 4 次数据拷贝（其中 2 次是 DMA 拷贝，2 次是 CPU 拷贝）。上下文切换和数据拷贝都会消耗 CPU 资源和时间，在高并发的场景下，这些开销会严重影响系统的性能。

例如，在一个高并发的文件服务器中，如果每一次文件传输都要经历如此繁琐的过程，那么服务器的吞吐量将会受到极大的限制，响应速度也会变慢，用户体验也会大打折扣。因此，为了提高系统的性能，我们需要寻找一种更高效的数据传输方式，这就是零拷贝技术应运而生的背景。

二、零拷贝技术闪亮登场

零拷贝（zero-copy）基本思想是：数据报从网络设备到用户程序空间传递的过程中，减少数据拷贝次数，减少系统调用，实现CPU的零参与，彻底消除 CPU在这方面的负载。实现零拷贝用到的最主要技术是DMA数据传输技术和内存区域映射技术。如图1所示，传统的网络数据报处理，需要经过网络设备到操作系统内存空间，系统内存空间到用户应用程序空间这两次拷贝，同时还需要经历用户向系统发出的系统调用。

而零拷贝技术则首先利用DMA技术将网络数据报直接传递到系统内核预先分配的地址空间中，避免CPU的参与；同时，将系统内核中存储数据报的内存区域映射到检测程序的应用程序空间（还有一种方式是在用户空间建立一缓存，并将其映射到内核空间，类似于linux系统下的kiobuf技术），检测程序直接对这块内存进行访问，从而减少了系统内核向用户空间的内存拷贝，同时减少了系统调用的开销，实现了真正的“零拷贝”。



2.1什么是零拷贝？

简单一点来说，零拷贝就是一种避免 CPU 将数据从一块存储拷贝到另外一块存储的技术。针对操作系统中的设备驱动程序、文件系统以及网络协议堆栈而出现的各种零拷贝技术极大地提升了特定应用程序的性能，并且使得这些应用程序可以更加有效地利用系统资源。这种性能的提升就是通过在数据拷贝进行的同时，允许 CPU 执行其他的任务来实现的。

零拷贝技术可以减少数据拷贝和共享总线操作的次数，消除传输数据在存储器之间不必要的中间拷贝次数，从而有效地提高数据传输效率。而且，零拷贝技术减少了用户应用程序地址空间和操作系统内核地址空间之间因为上下文切换而带来的开销。进行大量的数据拷贝操作其实是一件简单的任务，从操作系统的角度来说，如果 CPU 一直被占用着去执行这项简单的任务，那么这将会是很浪费资源的；如果有其他比较简单的系统部件可以代劳这件事情，从而使得 CPU 解脱出来可以做别的事情，那么系统资源的利用则会更加有效。

零拷贝技术，就是为了解决传统 I/O 的性能瓶颈而诞生的。简单来说，零拷贝技术的核心就是减少数据在内存之间的拷贝次数，从而提高数据传输的效率。这里的 “零拷贝” 并不是指完全没有数据拷贝，而是尽可能地减少不必要的拷贝操作。

在零拷贝技术中，数据可以直接在内核空间中进行传输，而不需要经过用户空间的缓冲区。这样就避免了传统 I/O 中数据在用户空间和内核空间之间的多次拷贝，减少了 CPU 的上下文切换和数据拷贝的开销，提高了系统的性能和吞吐量。

以 Linux 系统中的sendfile系统调用为例，这是一种常见的零拷贝实现方式。在使用sendfile时，数据可以直接从内核缓冲区传输到 socket 缓冲区，而不需要经过用户空间。具体过程如下：

- **用户态到内核态切换**：应用程序调用sendfile系统调用，请求将文件数据发送到网络。CPU 从用户态切换到内核态。
- **磁盘数据读取到内核缓冲区**：内核通过 DMA 技术，将磁盘数据直接拷贝到内核缓冲区。
- **内核缓冲区数据直接传输到 socket 缓冲区**：内核直接将内核缓冲区中的数据传输到 socket 缓冲区，而不需要经过用户空间。这一步利用了内核的特殊机制，直接在内核空间中完成数据的传输，避免了数据在用户空间和内核空间之间的拷贝。

- **socket 缓冲区数据发送到网卡**：内核通过 DMA 技术，将 socket 缓冲区中的数据拷贝到网卡缓冲区，然后通过网络发送出去。
- **内核态到用户态切换**：sendfile系统调用返回，CPU 从内核态切换回用户态，数据传输完成。

对比传统 I/O 和零拷贝技术的数据传输路径，可以明显看出零拷贝技术的优势。在传统 I/O 中，数据需要在用户空间和内核空间之间多次拷贝，而在零拷贝技术中，数据可以直接在内核空间中传输，减少了数据拷贝的次数和上下文切换的开销。这就好比在物流运输中，传统 I/O 就像是货物需要多次装卸、转运，而零拷贝技术则像是货物可以直接从起点运输到终点，中间不需要多次中转，大大提高了运输的效率。

避免数据拷贝：

- 避免操作系统内核缓冲区之间进行数据拷贝操作。
- 避免操作系统内核和用户应用程序地址空间这两者之间进行数据拷贝操作。
- 用户应用程序可以避开操作系统直接访问硬件存储。
- 数据传输尽量让 DMA 来做。

将多种操作结合在一起

- 避免不必要的系统调用和上下文切换。
- 需要拷贝的数据可以先被缓存起来。
- 对数据进行处理尽量让硬件来做。

对于高速网络来说，零拷贝技术是非常重要的。这是因为高速网络的网络链接能力与 CPU 的处理能力接近，甚至会超过 CPU 的处理能力。

如果是这样的话，那么 CPU 就有可能需要花费几乎所有的时间去拷贝要传输的数据，而没有能力再去做别的事情，这就产生了性能瓶颈，限制了通讯速率，从而降低了网络连接的能力。一般来说，一个 CPU 时钟周期可以处理一位的数据。举例来说，一个 1 GHz 的处理器可以对 1Gbit/s 的网络链接进行传统的数据拷贝操作，但是如果是 10 Gbit/s 的网络，那么对于相同的处理器来说，零拷贝技术就变得非常重要了。

对于超过 1 Gbit/s 的网络链接来说，零拷贝技术在超级计算机集群以及大型的商业数据中心中都有所应用。然而，随着信息技术的发展，1 Gbit/s，10 Gbit/s 以及 100 Gbit/s 的网络会越来越普及，那么零拷贝技术也会变得越来越普及，这是因为网络链接的处理能力比 CPU 的处理能力的增长要快得多。传统的数据拷贝受限于传统的操作系统或者通信协议，这就限制了数据传输性能。零拷贝技术通过减少数据拷贝次数，简化协议处理的层次，在应用程序和网络之间提供更快的数据传输方法，从而可以有效地降低通信延迟，提高网络吞吐率。零拷贝技术是实现主机或者路由器等设备高速网络接口的主要技术之一。

现代的 CPU 和存储体系结构提供了很多相关的功能来减少或避免 I/O 操作过程中产生的不必要的 CPU 数据拷贝操作，但是，CPU 和存储体系结构的这种优势经常被过高估计。存储体系结构的复杂性以及网络协议中必需的数据传输可能会产生问题，有时甚至会导致零拷贝这种技术的优点完全丧失。在下一章中，我们会介绍几种 Linux 操作系统中出现的零拷贝技术，简单描述一下它们的实现方法，并对它们的弱点进行分析。

2.2零拷贝技术分类

零拷贝技术的发展很多样化，现有的零拷贝技术种类也非常多，而当前并没有一个适合于所有场景的零拷贝技术的出现。对于 Linux 来说，现存的零拷贝技术也比较多，这些零拷贝技术大部分存在于不同的 Linux 内核版本，有些旧的技术在不同的 Linux 内核版本间得到了很大的发展或者已经渐渐被新的技术所代替。本文针对这些零拷贝技术所适用的不同场景对它们进行了划分。概括起来，Linux 中的零拷贝技术主要有下面这几种：

直接 I/O：对于这种数据传输方式来说，应用程序可以直接访问硬件存储，操作系统内核只是辅助数据传输：这类零拷贝技术针对的是操作系统内核并不需要对数据进行直接处理的情况，数据可以在应用程序地址空间的缓冲区和磁盘之间直接进行传输，完全不需要 Linux 操作系统内核提供的页缓存的支持。

在数据传输的过程中，避免数据在操作系统内核地址空间的缓冲区和用户应用程序地址空间的缓冲区之间进行拷贝。有的时候，应用程序在数据进行传输的过程中不需要对数据进行访问，那么，将数据从 Linux 的页缓存拷贝到用户进程的缓冲区中就可以完全避免，传输的数据在页缓存中就可以得到处理。在某些特殊的情况下，这种零拷贝技术可以获得较好的性能。Linux 中提供类似的系统调用主要有 mmap()，sendfile() 以及 splice()。

对数据在 Linux 的页缓存和用户进程的缓冲区之间的传输过程进行优化。该零拷贝技术侧重于灵活地处理数据在用户进程的缓冲区和操作系统的页缓存之间的拷贝操作。这种方法延续了传统的通信方式，但是更加灵活。在 Linux 中，该方法主要利用了写时复制技术。

前两类方法的目的主要是为了避免应用程序地址空间和操作系统内核地址空间这两者之间的缓冲区拷贝操作。这两类零拷贝技术通常适用在某些特殊的情况下，比如要传送的数据不需要经过操作系统内核的处理或者不需要经过应用程序的处理。第三类方法则继承了传统的应用程序地址空间和操作系统内核地址空间之间数据传输的概念，进而针对数据传输本身进行优化。

我们知道，硬件和软件之间的数据传输可以通过使用 DMA 来进行，DMA进行数据传输的过程中几乎不需要CPU参与，这样就可以把 CPU 解放出来去做更多其他的事情，但是当数据需要在用户地址空间的缓冲区和Linux操作系统内核的页缓存之间进行传输的时候，并没有类似DMA这种工具可以使用，CPU 需要全程参与到这种数据拷贝操作中，所以这第三类方法的目的是可以有效地改善数据在用户地址空间和操作系统内核地址空间之间传递的效率。

三、零拷贝的定义

Zero-copy, 就是在操作数据时, 不需要将数据 buffer 从一个内存区域拷贝到另一个内存区域. 因为少了一次内存的拷贝, 因此 CPU 的效率就得到的提升；在OS层面上的 Zero-copy 通常指避免在 用户态(User-space) 与 内核态(Kernel-space) 之间来回拷贝数据。

Netty 中的 Zero-copy 与 OS 的 Zero-copy 不太一样, Netty的 Zero-coyp 完全是在用户态(Java层面)的, 它的 Zero-copy 的更多的是偏向于优化数据操作。

3.1Netty的“零拷贝”

主要体现在如下三个方面：

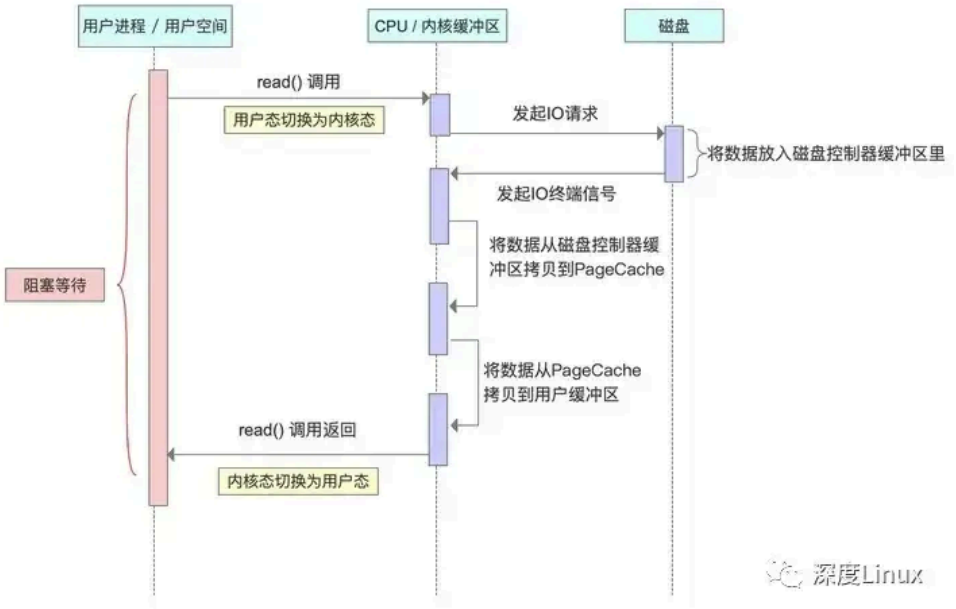
1. Netty的接收和发送ByteBuffer采用DIRECT BUFFERS，使用堆外直接内存进行Socket读写，不需要进行字节缓冲区的二次拷贝。如果使用传统的堆内存（HEAP BUFFERS）进行Socket读写，JVM会将堆内存Buffer拷贝一份到直接内存中，然后才写入Socket中。相比于堆外直接内存，消息在发送过程中多了一次缓冲区的内存拷贝。
2. Netty提供了组合Buffer对象，可以聚合多个ByteBuffer对象，用户可以像操作一个Buffer那样方便得对组合Buffer进行操作，避免了传统通过内存拷贝的方式将几个小Buffer合并成一个大的Buffer。
3. Netty的文件传输采用了transferTo方法，它可以直接将文件缓冲区的数据发送到目标Channel，避免了传统通过循环write方式导致的内存拷贝问题。

3.2传统IO方式

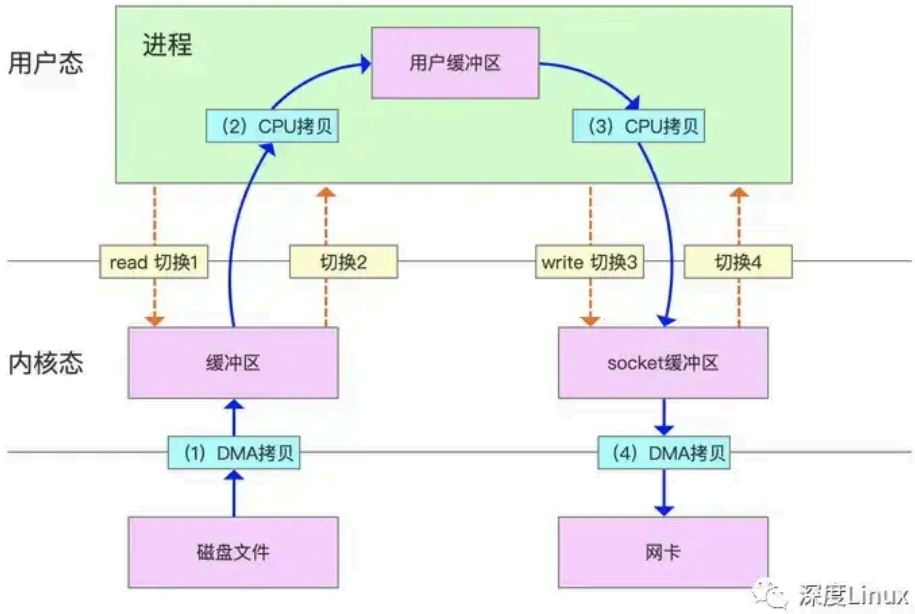
在 java 开发中，从某台机器将一份数据通过网络传输到另外一台机器，大致的代码如下：

```
Socket socket = new Socket(HOST, PORT);
InputStream inputStream = new FileInputStream(FILE_PATH);
OutputStream             outputStream             =             new
DataOutputStream(socket.getOutputStream());

byte[] buffer = new byte[4096];
while (inputStream.read(buffer) >= 0) {
    outputStream.write(buffer);
}

outputStream.close();
socket.close();
inputStream.close();
```

看起来代码很简单，但如果我们深入到操作系统层面，就会发现实际的微观操作更复杂。具体操作如下图：



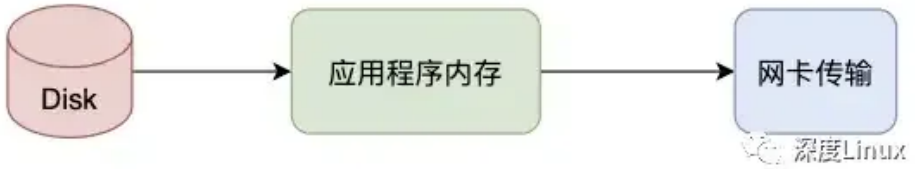
- 1.用户进程向OS发出read()系统调用，触发上下文切换，从用户态转换到内核态。
- 2.CPU发起IO请求，通过直接内存访问（DMA）从磁盘读取文件内容，复制到内核缓冲区PageCache中
- 3.将内核缓冲区数据，拷贝到用户空间缓冲区，触发上下文切换，从内核态转换到用户态。
- 4.用户进程向OS发起write系统调用，触发上下文切换，从用户态切换到内核态。
- 5.将数据从用户缓冲区拷贝到内核中与目的地Socket关联的缓冲区。
- 6.数据最终经由Socket通过DMA传送到硬件（网卡）缓冲区，write()系统调用返回，并从内核态切换回用户态。



四、零拷贝（Zero-copy）

4.1数据拷贝基础过程

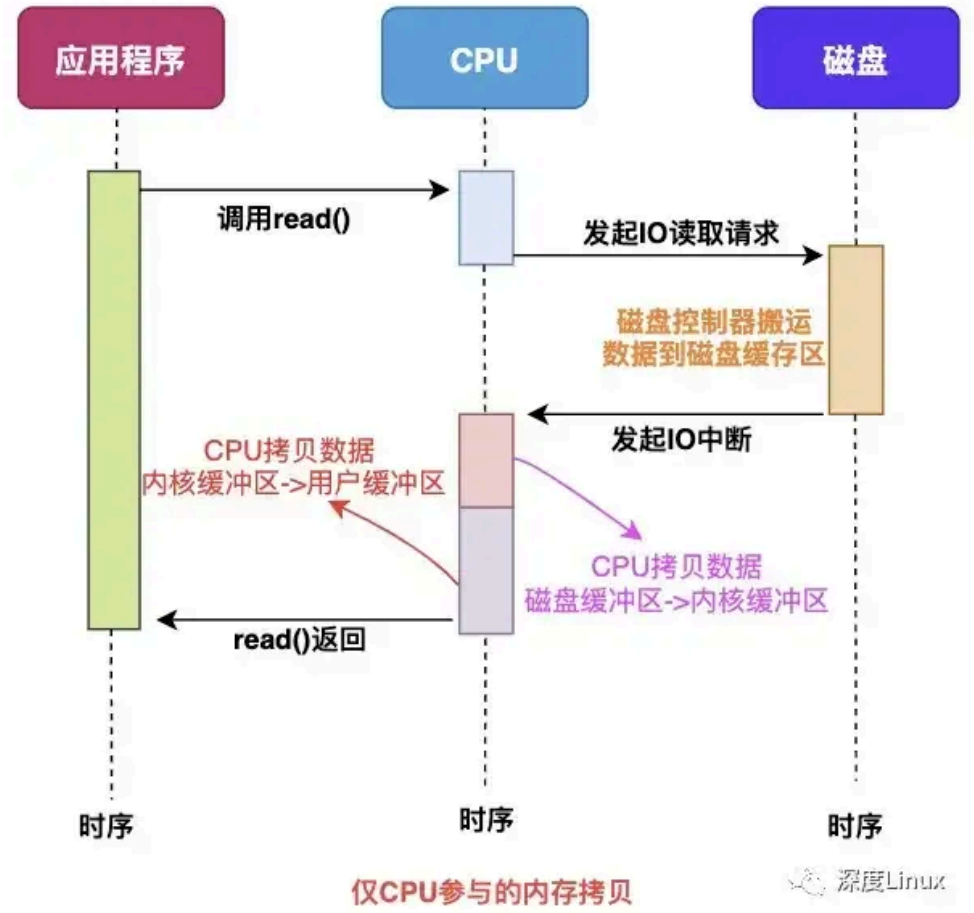
在Linux系统内部缓存和内存容量都是有限的，更多的数据都是存储在磁盘中。对于Web服务器来说，经常需要从磁盘中读取数据到内存，然后再通过网卡传输给用户：



上述数据流转只是大框，接下来看看几种模式。

(1)仅CPU方式

当应用程序需要读取磁盘数据时，调用read()从用户态陷入内核态，read()这个系统调用最终由CPU来完成；
CPU向磁盘发起I/O请求，磁盘收到之后开始准备数据；
磁盘将数据放到磁盘缓冲区之后，向CPU发起I/O中断，报告CPU数据已经Ready了；
CPU收到磁盘控制器的I/O中断之后，开始拷贝数据，完成之后read()返回，再从内核态切换到用户态；

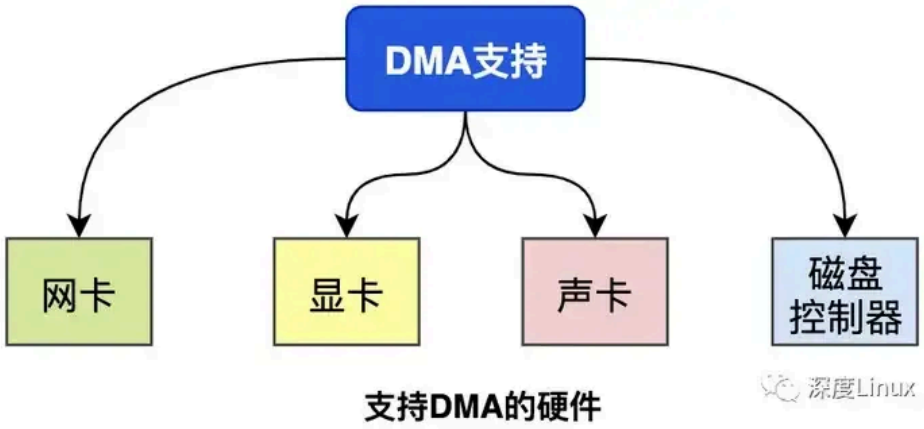


（2）CPU&DMA方式

CPU的时间宝贵，让它做杂活就是浪费资源。

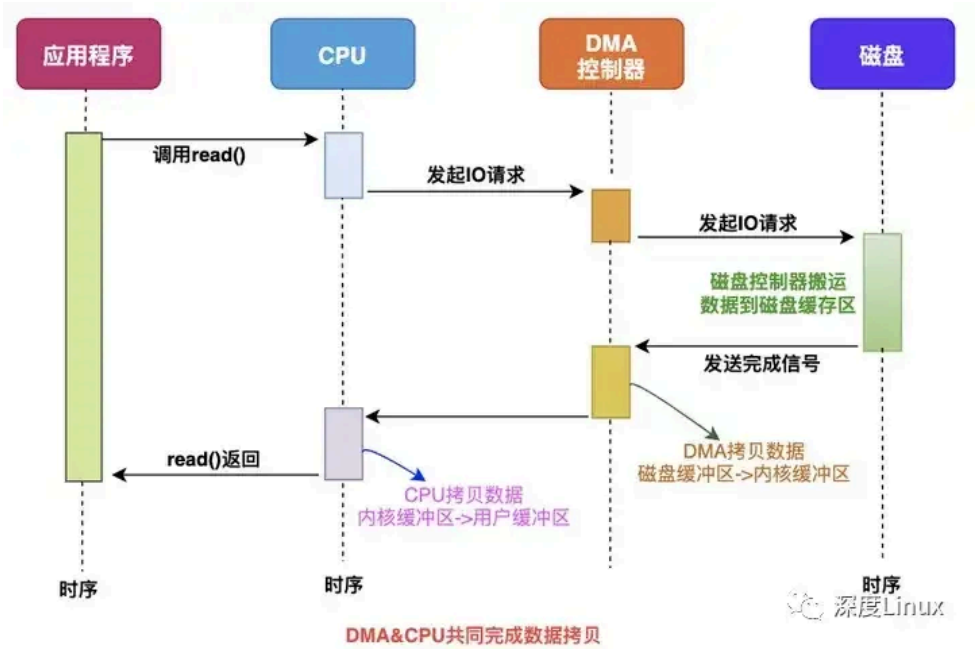
直接内存访问（Direct Memory Access），是一种硬件设备绕开CPU独立直接访问内存的机制。所以DMA在一定程度上解放了CPU，把之前CPU的杂活让硬件直接自己做了，提高了CPU效率。

目前支持DMA的硬件包括：网卡、声卡、显卡、磁盘控制器等。



支持DMA的硬件

有了DMA的参与之后的流程发生了一些变化：



DMA&CPU共同完成数据拷贝

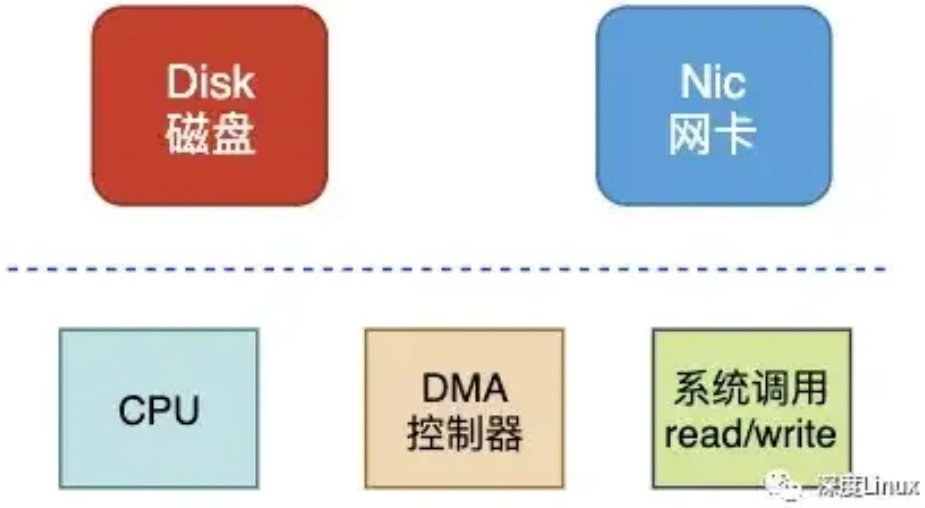
主要的变化是，CPU不再和磁盘直接交互，而是DMA和磁盘交互并且将数据从磁盘缓冲区拷贝到内核缓冲区，之后的过程类似。

> “**【敲黑板】**无论从仅CPU方式和DMA&CPU方式，都存在多次冗余数据拷贝和内核态&用户态的切换。”

我们继续思考Web服务器读取本地磁盘文件数据再通过网络传输给用户的详细过程。

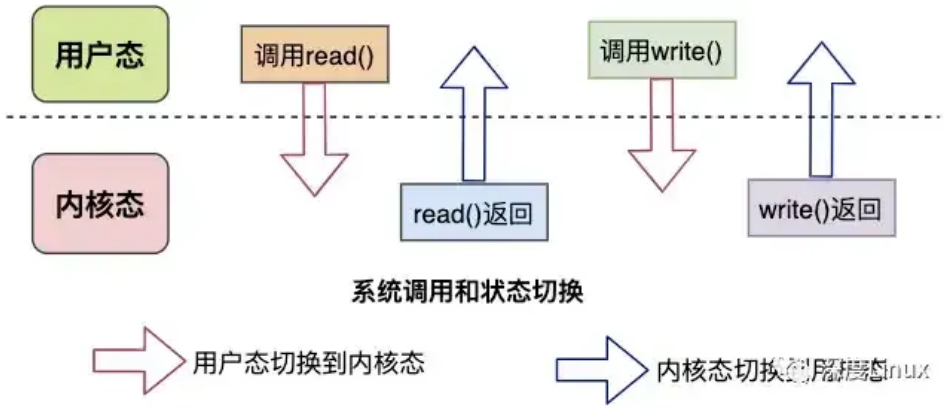
4.2普通模式数据交互

一次完成的数据交互包括几个部分：系统调用syscall、CPU、DMA、网卡、磁盘等。

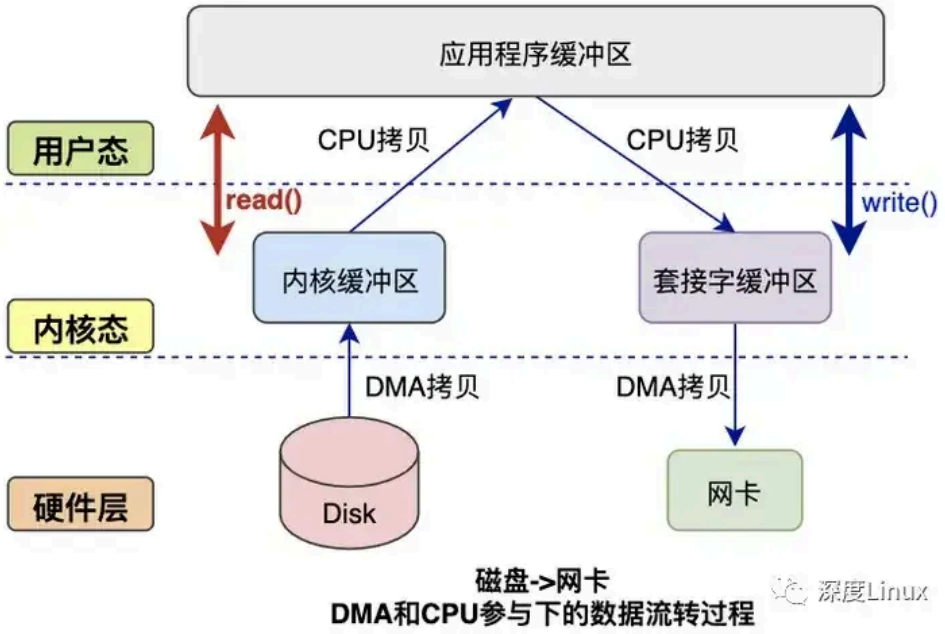


系统调用syscall是应用程序和内核交互的桥梁，每次进行调用/返回就会产生两次切换：

- 调用syscall 从用户态切换到内核态
- syscall返回 从内核态切换到用户态



来看下完整的数据拷贝过程简图：



读数据过程：

1. 应用程序要读取磁盘数据，调用read()函数从而实现用户态切换内核态，这是第1次状态切换；
2. DMA控制器将数据从磁盘拷贝到内核缓冲区，这是第1次DMA拷贝；
3. CPU将数据从内核缓冲区复制到用户缓冲区，这是第1次CPU拷贝；
4. CPU完成拷贝之后，read()函数返回实现用户态切换用户态，这是第2次状态切换；

写数据过程：

1. 应用程序要向网卡写数据，调用write()函数实现用户态切换内核态，这是第1次切换；
2. CPU将用户缓冲区数据拷贝到内核缓冲区，这是第1次CPU拷贝；
3. DMA控制器将数据从内核缓冲区复制到socket缓冲区，这是第1次DMA拷贝；
4. 完成拷贝之后，write()函数返回实现内核态切换用户态，这是第2次切换；

综上所述：

- 读过程涉及2次空间切换、1次DMA拷贝、1次CPU拷贝；
- 写过程涉及2次空间切换、1次DMA拷贝、1次CPU拷贝；

可见传统模式下，涉及多次空间切换和数据冗余拷贝，效率并不高，接下来就该零拷贝技术出场了。

4.3零拷贝技术

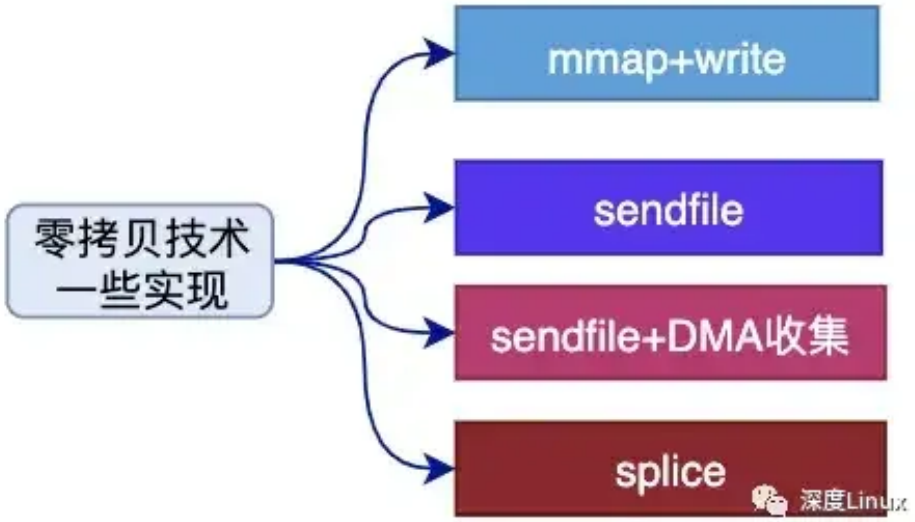
(1)出现原因

我们可以看到，如果应用程序不对数据做修改，从内核缓冲区到用户缓冲区，再从用户缓冲区到内核缓冲区。两次数据拷贝都需要CPU的参与，并且涉及用户态与内核态的多次切换，加重了CPU负担。

我们需要降低冗余数据拷贝、解放CPU，这也就是零拷贝Zero-Copy技术。

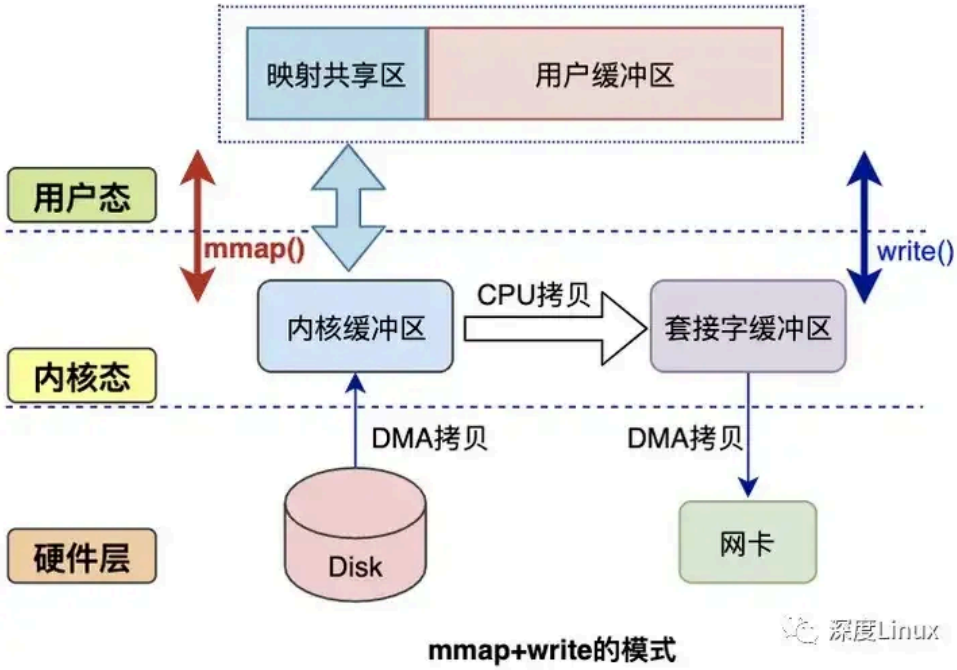
（2）解决思路

目前来看，零拷贝技术的几个实现手段包括：mmap+write、sendfile、sendfile+DMA收集、splice等。



(3)mmap方式

mmap是Linux提供的一种内存映射文件的机制，它实现了将内核中读缓冲区地址与用户空间缓冲区地址进行映射，从而实现内核缓冲区与用户缓冲区的共享。这样就减少了一次用户态和内核态的CPU拷贝，但是在内核空间内仍然有一次CPU拷贝。



mmap+write的模式

mmap对大文件传输有一定优势，但是小文件可能出现碎片，并且在多个进程同时操作文件时可能产生引发coredump的signal。

(4)sendfile方式

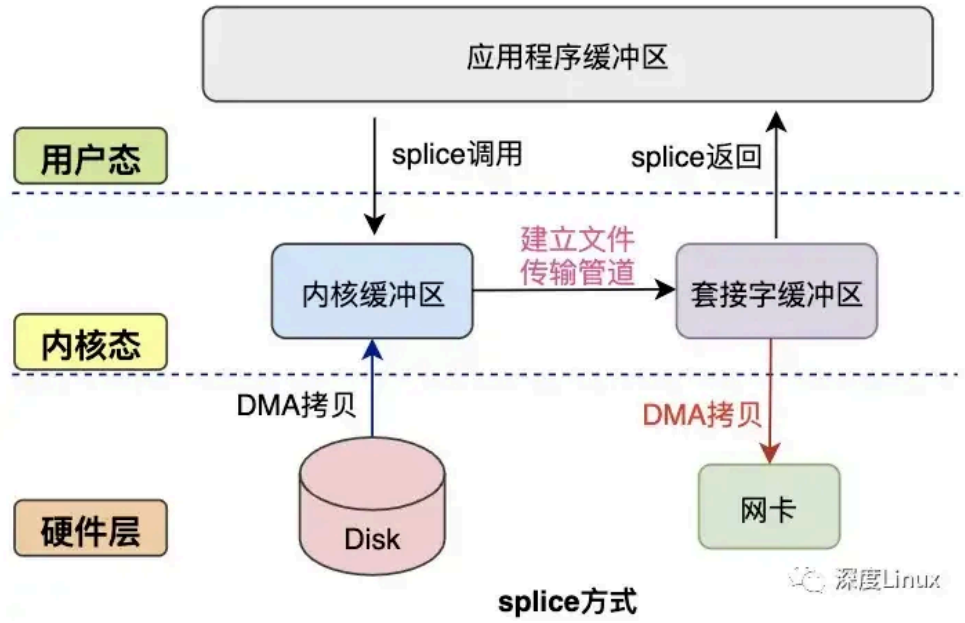
mmap+write方式有一定改进，但是由系统调用引起的状态切换并没有减少。

sendfile系统调用是在 Linux 内核2.1版本中被引入，它建立了两个文件之间的传输通道。

sendfile方式只使用一个函数就可以完成之前的read+write 和 mmap+write的功能，这样就少了2次状态切换，由于数据不经过用户缓冲区，因此该数据无法被修改。

```
NAME        top

       splice - splice data to/from a pipe

SYNOPSIS    top

       #define _GNU_SOURCE         /* See feature_test_macros(7) */
       #include <fcntl.h>

       ssize_t splice(int fd_in, loff_t *off_in, int fd_out,
                      loff_t *off_out, size_t len, unsigned int flags);

DESCRIPTION    top

       splice() moves data between two file descriptors without copying
       between kernel address space and user address space.  It transfers up
       to len bytes of data from the file descriptor fd_in to the file
       descriptor fd_out, where one of the file descriptors must refer to a
       pipe.
```


splice 系统调用可以在内核缓冲区和socket缓冲区之间建立管道来传输数据，避免了两者之间的 CPU 拷贝操作。

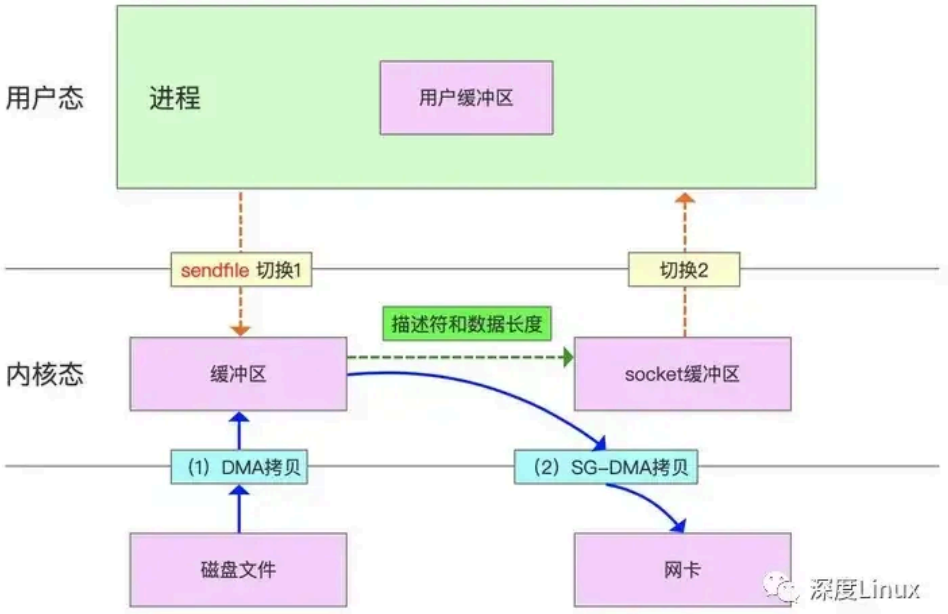


splice也有一些局限，它的两个文件描述符参数中有一个必须是管道设备。

以下使用 FileChannel.transferTo 方法，实现 zero-copy：

```
SocketAddress socketAddress = new InetSocketAddress(HOST, PORT);
SocketChannel socketChannel = SocketChannel.open();
socketChannel.connect(socketAddress);

File file = new File(FILE_PATH);
FileChannel fileChannel = new FileInputStream(file).getChannel();
fileChannel.transferTo(0, file.length(), socketChannel);

fileChannel.close();
socketChannel.close();
```

相比传统方式，零拷贝的执行流程如下图：



可以看到，相比传统方式，零拷贝不走数据缓冲区减少了一些不必要的操作。

4.4零拷贝的应用

零拷贝在很多框架中得到了广泛使用，常见的比如 Netty、Kafka 等等。

在 kafka 中使用了很多设计思想，比如分区并行、顺序写入、页缓存、高效序列化、零拷贝等等。

上边博客分析了 Kafka 的大概架构，知道了 kafka 中的文件都是以.log 文件存储，每个日志文件对应两个索引文件.index 与.timeindex。

kafka 在传输数据时利用索引，使用 fileChannel.transferTo (position, count, socketChannel) 指定数据位置与大小实现零拷贝。

kafka 底层传输源码：（TransportLayer）

```
/**
             * Transfers bytes from `fileChannel` to this
`TransportLayer`.
     *
                * This method will delegate to {@link
FileChannel#transferTo(long,                                long,
java.nio.channels.WritableByteChannel)},
     * but it will unwrap the destination channel, if possible, in
order to benefit from zero copy. This is required
      * because the fast path of `transferTo` is only executed if
the destination buffer inherits from an internal JDK
     * class.
     *
     * @param fileChannel The source channel
```

```
     * @param position The position within the file at which the
transfer is to begin; must be non-negative
     * @param count The maximum number of bytes to be transferred;
must be non-negative
     * @return The number of bytes, possibly zero, that were
actually transferred
     *    @see    FileChannel#transferTo(long,    long,
java.nio.channels.WritableByteChannel)
     */
    long transferFrom(FileChannel fileChannel, long position, long
count) throws IOException;
```

实现类（PlaintextTransportLayer）：

```
 @OverRide
 public long transferFrom(FileChannel fileChannel, long position,
long count) throws IOException {
                return   fileChannel.transferTo(position,   count,
socketChannel);
 }
```

该方法的功能是将 FileChannel 中的数据传输到 TransportLayer，也就是 SocketChannel。在实现类 PlaintextTransportLayer 的对应方法中，就是直接调用了 FileChannel.transferTo () 方法。

五、零拷贝在Java世界的奇妙冒险

5.1NIO 的零拷贝绝技

在 Java 的世界里，NIO（New I/O）为我们提供了实现零拷贝的强大工具。其中，FileChannel类的transferTo和transferFrom方法就是实现零拷贝的关键。

transferTo方法可以将当前通道中的数据直接传输到目标通道，而transferFrom方法则是从源通道读取数据并传输到当前通道。这两个方法的实现依赖于底层操作系统的支持，在支持零拷贝的操作系统上，它们可以直接利用操作系统的零拷贝机制，避免数据在用户空间和内核空间之间的拷贝，从而大大提高数据传输的效率。

下面通过一个简单的代码示例来展示如何使用transferTo方法实现文件到网络通道的零拷贝传输：

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.nio.channels.FileChannel;

public class ZeroCopyExample {
```

```
    public static void main(String[] args) {
                            try   (FileInputStream   fis   =   new
FileInputStream("source.txt");
                                    FileOutputStream   fos   =   new
FileOutputStream("destination.txt");
             FileChannel sourceChannel = fis.getChannel();
             FileChannel destChannel = fos.getChannel()) {
            long position = 0;
            long count = sourceChannel.size();
            // 使用transferTo方法实现零拷贝
                        sourceChannel.transferTo(position,  count,
destChannel);
                System.out.println("File  copied  successfully  using
zero copy!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

在这个示例中，我们首先创建了两个FileChannel，一个用于读取源文件，另一个用于写入目标文件。然后，通过transferTo方法将源文件通道中的数据直接传输到目标文件通道，整个过程中数据没有经过用户空间的缓冲区，实现了零拷贝。

5.2Netty框架的零拷贝奥秘

Netty 是一个高性能的 Java 网络框架，它在设计中充分利用了零拷贝技术，以提升数据传输的效率。在 Netty 中，零拷贝主要体现在以下几个方面：

首先，Netty 接收和发送ByteBuffer采用的都是堆外直接内存。使用堆外直接内存进行 Socket 的读 / 写，无须进行字节缓冲区的二次拷贝。如果使用传统的堆内存进行 Socket 的读 / 写，则 JVM 会将堆内存 Buffer 数据拷贝到堆外直接内存中，然后才写入 Socket 中。与堆外直接内存相比，使用传统的堆内存，在消息的发送过程中多了一次缓冲区的内存拷贝 。

其次，Netty 的ByteBuf提供了强大的零拷贝功能。ByteBuf是 Netty 的数据容器，它通过切片（slice）和组合（CompositeByteBuf）等操作实现了零拷贝。例如，slice方法可以将一个ByteBuf分解为多个共享同一个存储区域的ByteBuf，避免了内存的拷贝。假设有一个ByteBuf包含了消息的头部和消息体，我们可以通过slice方法分别获取头部和消息体的ByteBuf，而不需要进行数据的拷贝：

```
ByteBuf buffer = Unpooled.wrappedBuffer(new byte[]{1, 2, 3, 4, 5,
6, 7, 8, 9, 10});
ByteBuf header = buffer.slice(0, 5);
ByteBuf body = buffer.slice(5, 5);
```

在这个例子中，header和body共享了buffer的底层存储区域，对buffer的修改会反映在header和body上，反之亦然。

另外，CompositeByteBuf可以将多个ByteBuf合并为一个逻辑上的ByteBuf，避免了各个ByteBuf之间的拷贝。比如，在处理协议数据时，协议数据可能由头部和消息体组成，而头部和消息体分别存放在两个ByteBuf中，我们可以使用CompositeByteBuf将它们合并为一个逻辑上的整体：

```
ByteBuf header = Unpooled.wrappedBuffer(new byte[]{1, 2, 3, 4, 5});
ByteBuf body = Unpooled.wrappedBuffer(new byte[]{6, 7, 8, 9, 10});
CompositeByteBuf compositeByteBuf = Unpooled.compositeBuffer();
compositeByteBuf.addComponents(true, header, body);
```

这样，compositeByteBuf就将header和body组合在了一起，在后续的处理中可以像操作一个ByteBuf一样操作它，而不需要进行数据的拷贝。

最后，Netty 使用FileRegion实现文件传输的零拷贝。FileRegion底层封装了FileChannel的transferTo方法，可以将文件缓冲区的数据直接传输到目标通道，避免内核缓冲区和用户态缓冲区之间的数据拷贝，这属于操作系统级别的零拷贝。例如，在一个文件服务器中，当客户端请求下载文件时，服务器可以使用FileRegion将文件数据直接传输给客户端，而不需要将文件数据先读取到用户空间再发送出去，从而提高了文件传输的效率。

六、零拷贝的应用舞台

6.1网络服务器：高效传输的基石

在网络服务器领域，零拷贝技术是提升性能的关键。以 Nginx 为例，它作为一款高性能的 Web 服务器，广泛应用零拷贝技术来提高静态文件的传输效率。当客户端请求一个静态文件时，传统的服务器可能需要先将文件数据从磁盘读取到内核缓冲区，再拷贝到用户空间的应用程序缓冲区，最后再发送到网络。而 Nginx 利用sendfile系统调用实现零拷贝，数据可以直接从磁盘内核缓冲区传输到网络 socket 缓冲区，避免了用户空间的拷贝操作，大大提高了文件传输的速度，减少了 CPU 的开销。

在处理大量并发请求时，零拷贝技术的优势更加明显。假设一个高并发的文件下载服务器，每秒要处理数千个文件下载请求，如果每个请求都采用传统的 I/O 方式，那么 CPU 将忙于数据拷贝和上下文切换，很快就会达到性能瓶颈。而采用零拷贝技术，CPU 可以将更多的资源用于处理其他任务，服务器的吞吐量会大幅提升，能够轻松应对高并发的场景，为用户提供更快的响应速度。

6.2文件系统：快速操作的秘诀

在文件系统中，零拷贝技术也发挥着重要作用。当我们进行文件复制操作时，传统方式是将源文件的数据从磁盘读取到内核缓冲区，再拷贝到用户空间缓冲区，然后写入目标文件，这个过程涉

及多次数据拷贝和上下文切换。而利用零拷贝技术，如通过mmap将文件映射到内存，数据可以直接在内核空间中进行传输和处理，减少了 I/O 开销。

例如，在一个大数据存储系统中，经常需要进行大规模的数据文件迁移操作。如果使用传统的文件复制方式，在迁移大量文件时，不仅会消耗大量的时间，还会占用大量的系统资源，导致系统性能下降。而采用零拷贝技术，通过mmap和sendfile等机制，可以大大减少数据拷贝的次数，提高文件迁移的速度，同时降低系统的负载，让系统能够更高效地运行。

6.3多媒体流传输：流畅体验的保障

在多媒体流传输领域，零拷贝技术是实现流畅播放体验的关键。以在线视频播放为例，视频数据需要从服务器快速传输到客户端，并且要求低延迟。传统的数据传输方式在多次数据拷贝过程中会增加延迟，导致视频播放卡顿。而利用零拷贝技术，视频数据可以直接从磁盘通过内核空间传输到网络，减少了数据处理的时间，降低了延迟。

在音频直播场景中，零拷贝技术同样重要。音频数据需要实时传输到听众的设备上，对延迟要求极高。通过零拷贝技术，音频数据能够快速地从服务器发送到网络，保证了音频直播的实时性和流畅性，让听众能够获得更好的收听体验。

Linux内核 245　项目实战 96　内存管理 91

Linux内核 · 目录

上一篇
全面解读zsmalloc：高效内存分配器的源码

下一篇
解锁Linux内存映射：让你的程序飞起来