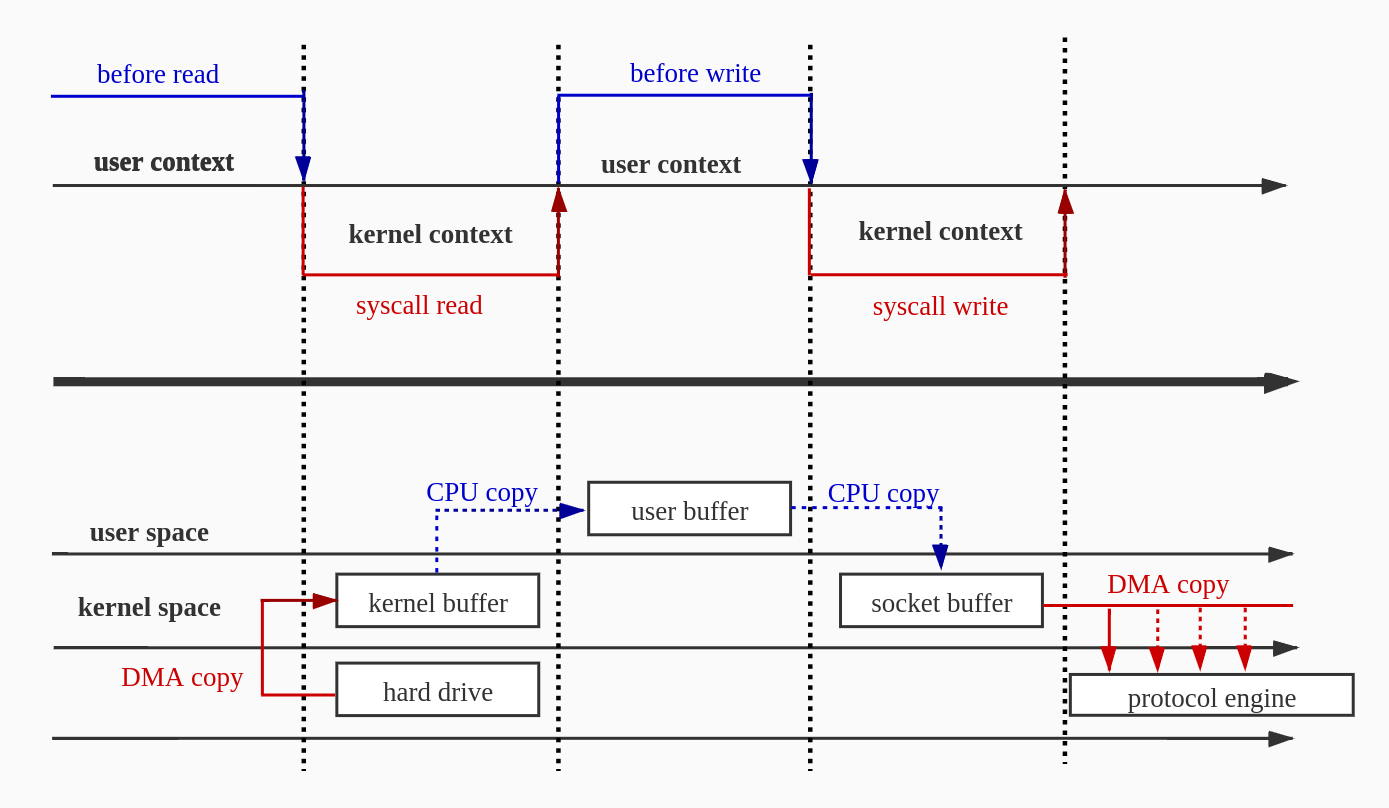
**一、零拷贝：**

所谓的 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. 它通常是指计算机在网络上发送文件时，不需要将文件内容拷贝到用户空间而直接在内核空间中传输到网络的方式。因为少了一次内存的拷贝, 因此 CPU 的效率就得到的提升.

1. **零拷贝在Linux内核上的体现：**

<https://www.cnblogs.com/200911/articles/10432551.html>

**1.linxu kernel< 2.1，传统的数据传输（4次拷贝，3次上下文切换）**



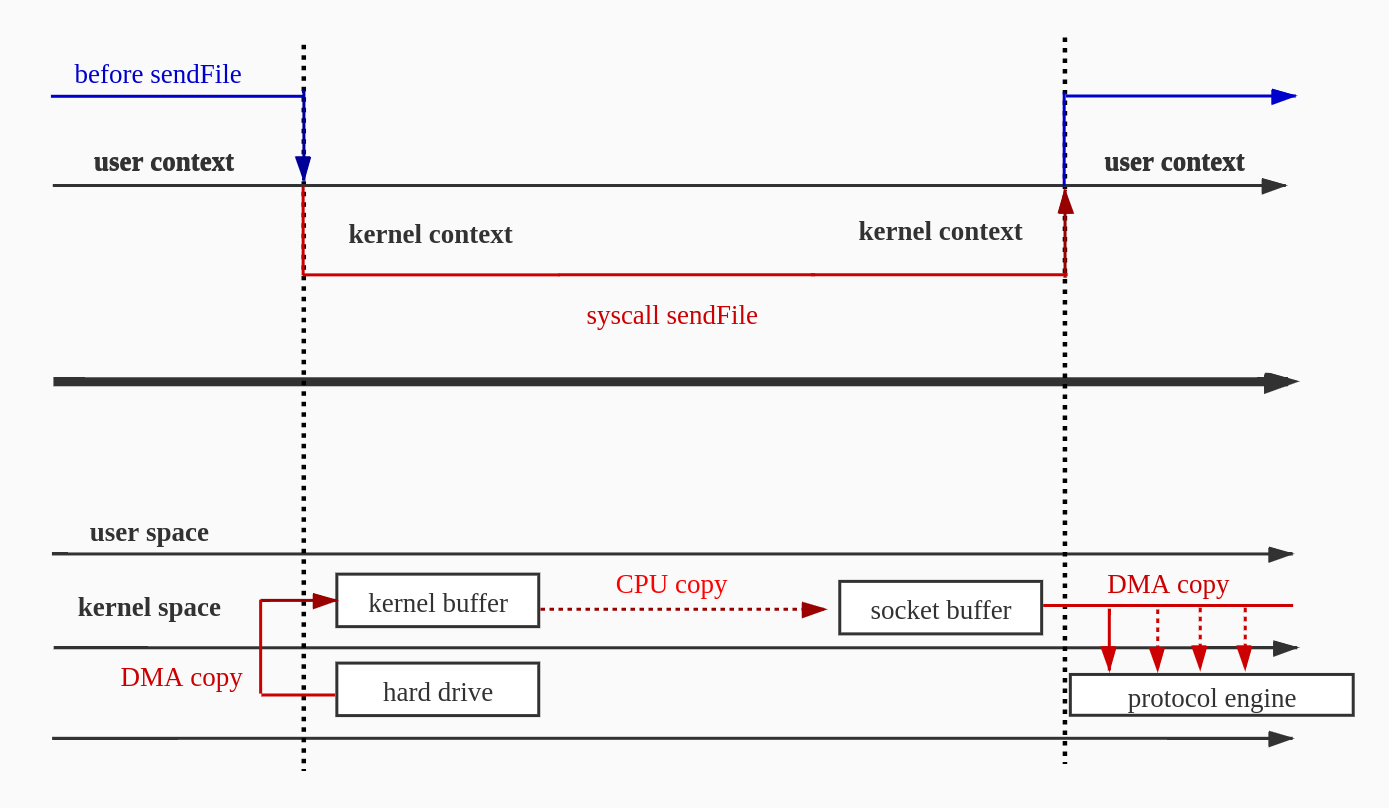
（1）当调用 read 系统调用时，通过 DMA将数据 copy 到内核模式。

（2）然后由 CPU 控制将内核模式数据 copy 到用户模式下的 buffer 中。

（3）read 调用完成后，write 调用首先将用户模式下 buffer 中的数据 copy 到内核模式下的 socket buffer 中。

（4）最后通过 DMA copy 将内核模式下的 socket buffer 中的数据 copy 到网卡设备中传送。

**2.linxu kernel [2.1，2.4]，数据传输（3次拷贝，2次上下文切换）**

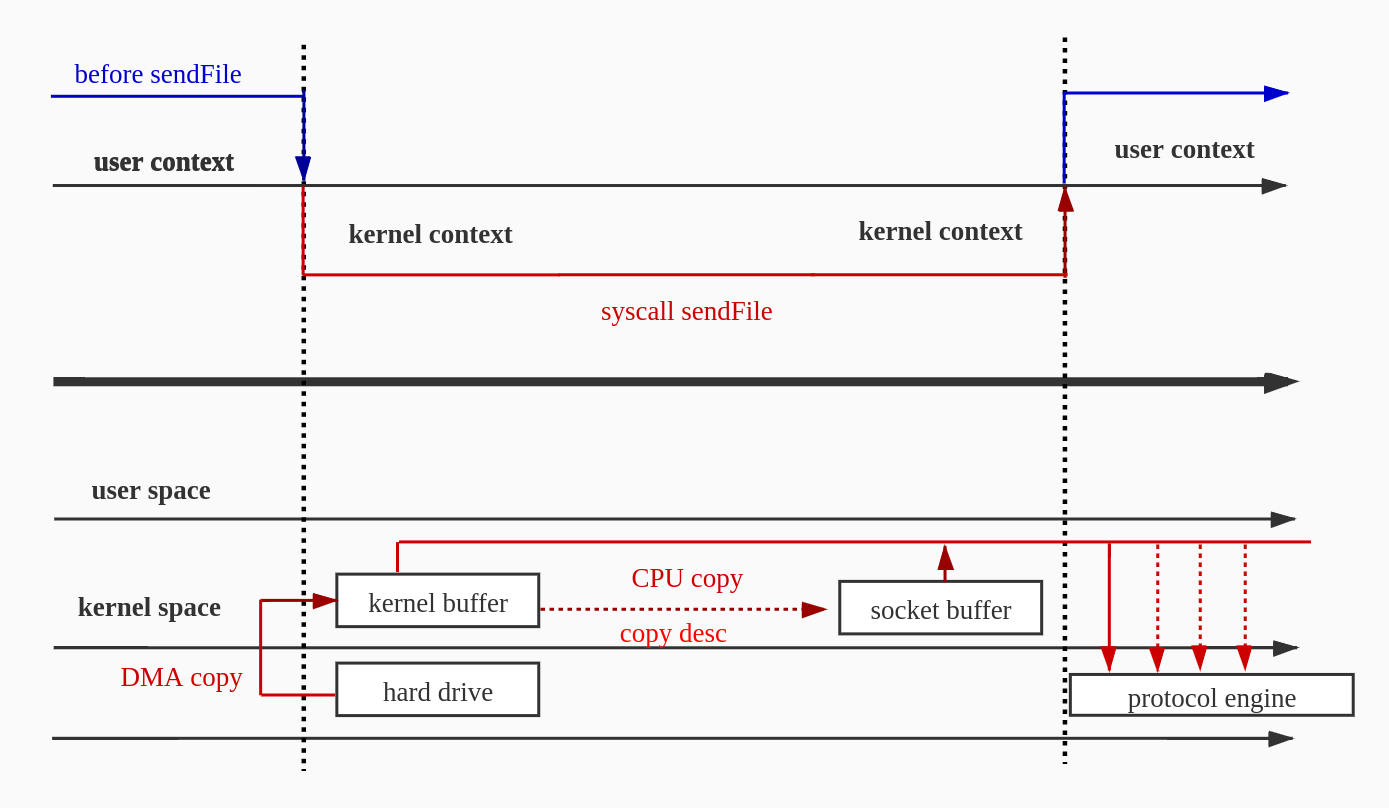


（1）当调用 sendFile 系统调用时，通过 DMA将数据 copy 到内核模式。

（2）然后由 CPU 控制将kernel buffer数据 copy 到socket buffer 中。

（3）最后通过 DMA copy 将内核模式下的 socket buffer 中的数据 copy 到网卡设备中传送。

**3.linxu kernel [2.4，?]，数据传输（2次拷贝，2次上下文切换）**



（1）当调用 sendFile 系统调用时，通过 DMA将数据 copy 到内核模式。

（2）向socket buffer中追加当前要发生的数据在kernel buffer中的位置和偏移量；

（3）根据socket buffer中的位置和偏移量直接将kernel buffer的数据copy到网卡设备（protocol engine）中；

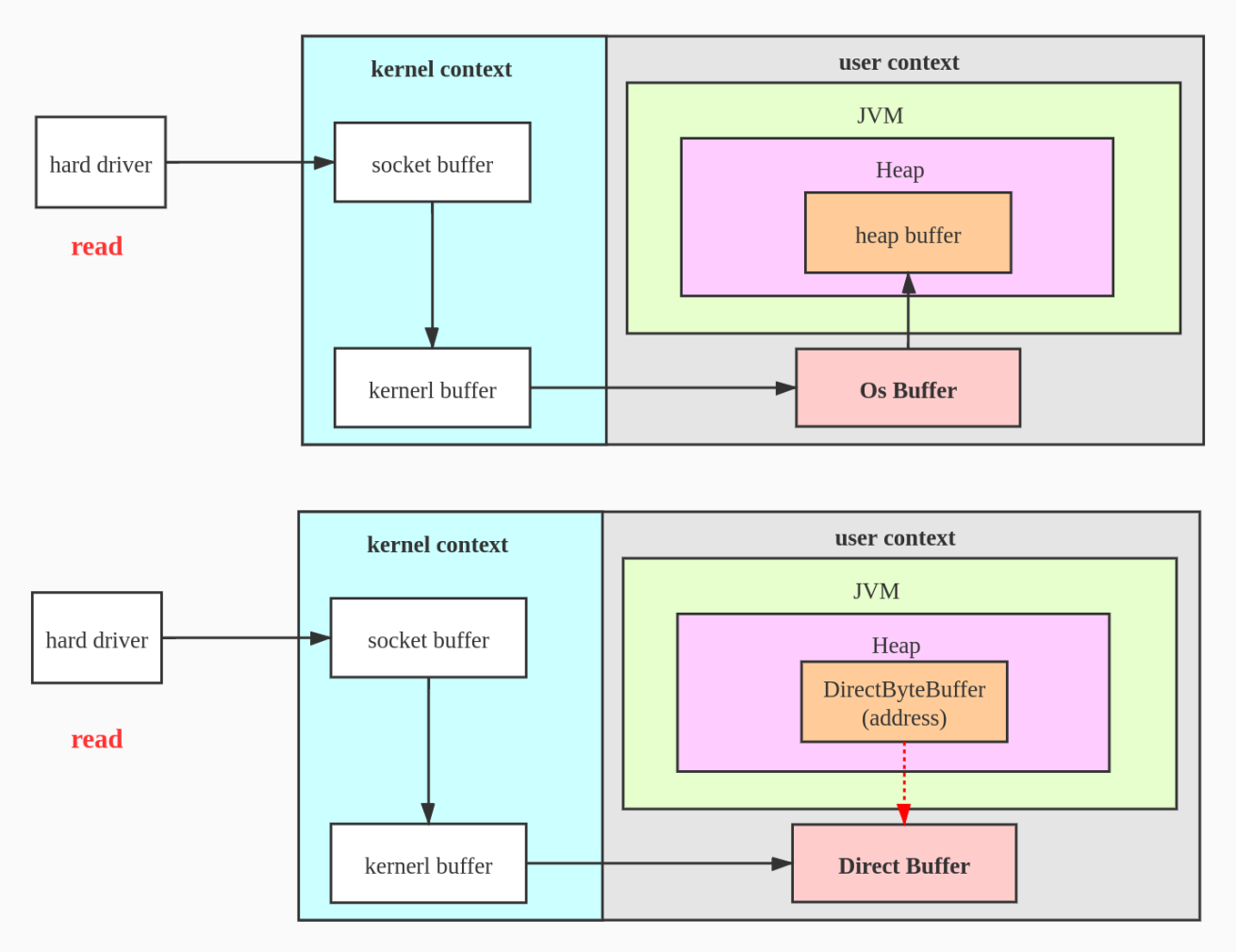
**三、零拷贝在NIO上的体现：**

**1.Direct Buffer避免数据从JVM Heap到C Heap的拷贝**

<https://www.jianshu.com/p/a7ff94c0ed81>

**直接内存（Direct Memory）**,直接内存并不是运行时数据区的一部分，也不是Java虚拟机规范中定义的内在区域。它通过Unsafe类的allocateMemory()方法申请分配内存，底层会调用操作系统的的malloc函数。

下图展示了read中，不使用Direct Buffer与使用Direct Buffer的过程



|  |
| --- |
| **不使用Direct Buffer：**  协议引擎—>内核socket缓冲区—>内核缓冲区—>用户地址空间的堆外内存—>**Java堆内存**—>处理 |
| **使用Direct Buffer：**  协议引擎—>内核socket缓冲区—>内核缓冲区—>用户地址空间的堆外内存—>处理 |

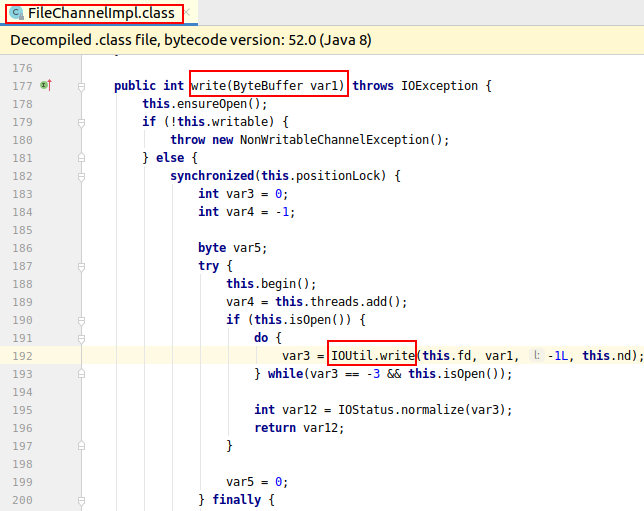
用堆外内存只需拷贝一次，而用堆内存是要拷贝两次（内核->堆外->堆）

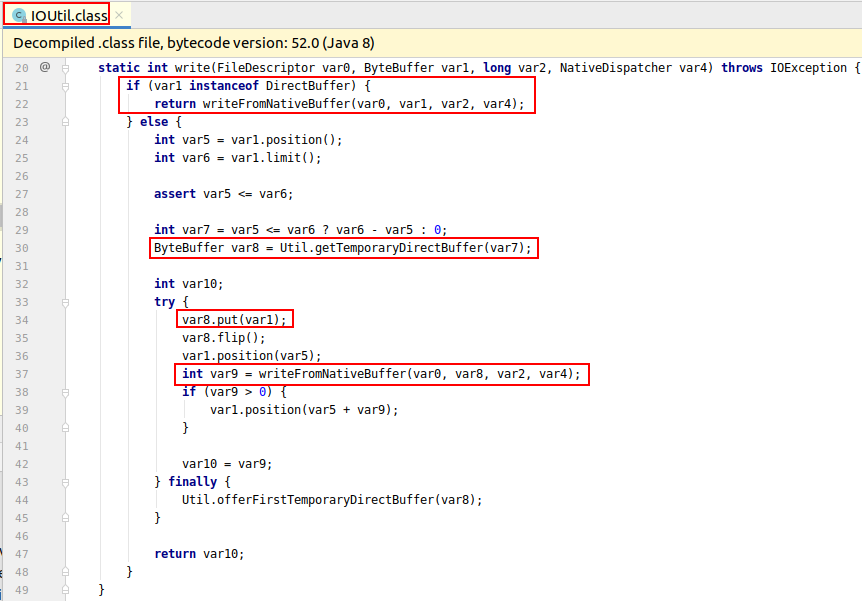
**BIO方式是否有Java heap 到 C heap 的拷贝？？？？？？？？？？？？？**

**1.1为何需要内存空间和用户空间的划分呢？**  
 每个进程都独立使用属于自己的内存一样，为了保证操作系统的稳定性，运行在操作系统中的用户程序不能访问操作系统所使用的内存空间。这也是从安全性上考虑的，如访问硬件资源只能由操作系统来发起，用户程序不允许直接访问硬件资源。如果用户程序需要访问硬件资源，如网络连接等，可以调用操作系统提供的接口来实现，这个调用接口的过程也就是**系统调用**。每一次系统调用都会存在两个内存空间的切换，通常的网络传输也是一次系统调用，通过网络传输的数据先是从内核空间接收到远程主机的数据，然后再从内核空间复制到用户空间，供用户程序使用。这种从内核空间到用户空间的数据复制很费时，虽然保住了程序运行的安全性和稳定性，但是也牺牲了一部分效率。但是现在已经出现了很多其他技术能够减少这种从内核空间到用户空间的数据复制的方式，如Linux系统提供了sendfile 文件传输方式。  
**注意，只有内核空间的内存才能被DMA引擎独立异步地存取。**

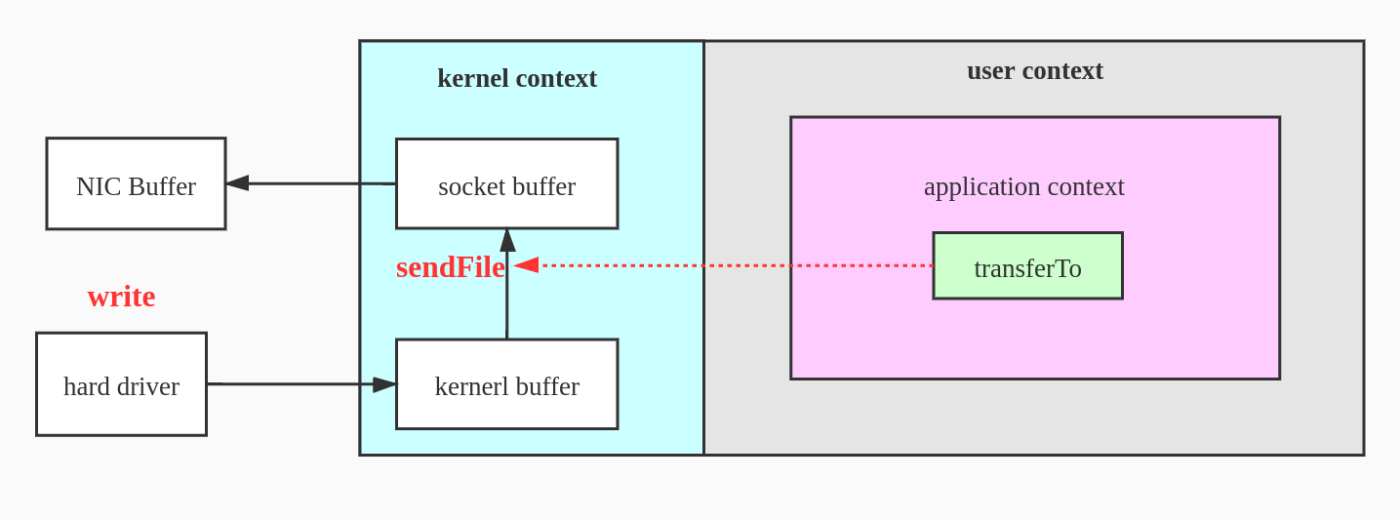
**1.2 DirectBuffer的开辟的堆外内存解决了什么问题？**  
HeapByteBuffer是在jvm的内存范围之内，然后在调io的操作时会将数据区域拷贝一份到os的内存区域，这样造成了不必要的性能上的降低，这样做是有原因的，试想假设如果os和jvm都是用jvm里边的数据区域， 但是jvm会对这块内存区域进行GC回收，可能会对这块内存的数据进行更改，根据我们的假设，由于这块区域os也在使用，jvm对这块共享数据发生了变更，os那边就会出现数据错乱的情况。那么如果不让jvm对这块共享区域进行GC是不是可以避免这个问题呢？  
答案是不行的，也会存在问题，如果jvm不对其进行GC回收，jvm这边可能会出现OOM的内存溢出。因此，最后这个地方非常尴尬，只能拷贝jvm的那一份到os的内存空间，即使jvm那边的数据区域被改变，但是os里边的不会受到影响，等os使用io结束后会对这块区域进行回收，因为这是os的管理范围之内。

**1.8 DirectByteBuffer 的原理？**  
其中 DirectByteBuffer 自身是一个Java对象，在Java堆中；而这个对象中有个long类型字段address，记录着一块调用 malloc() 申请到的native memory。  
IO的时候只需要把 DirectByteBuffer 背后的native memory地址传给真正做I/O的函数。这边就不需要再去访问Java对象去读写要做I/O的数据了。  
这块内存直接与io设备进行交互，当jvm对DirectByteBuffer内存垃圾回收的时候，会通过address调os，os将address对应的区域回收。

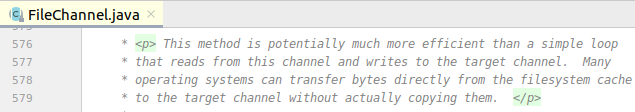
**NIO中FileChannel的write源码如下：**



**2.NIO的零拷贝由transferTo方法实现。它依赖底层操作系统的支持，调用该方法会引起sendfile()系统调用，避免了用户态与内核态之间的数据拷贝。**



**NIO中FileChannel的transferTo源码如下：**



Screenshot from 2020-10-21 17-32-57

**使用BIO和NIO传输同等大小的文件(10M),所需的时间进行对比**

**横轴：次数 纵轴：时间(ms)**

**BIO的实现源码如下：**

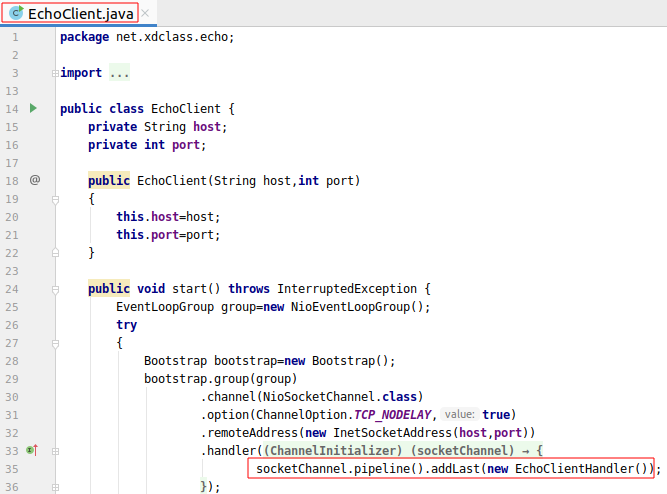
|  |
| --- |
| **public class** BIOServer {  **public static void** main(String[] args) **throws** Exception{  **int** port = 2001;  ServerSocket ss = **new** ServerSocket(port);  Socket socket = ss.accept();  InputStream is = socket.getInputStream();  FileOutputStream fos = **new** FileOutputStream(**"/home/doqin/Downloads/BIOTest.tar.gz"**);  **byte**[] b = **new byte**[4096];  **int** readLen ;   **while** ((readLen = is.read(b))>=0) {  fos.write(b,0,readLen);  }  socket.close();  is.close();  fos.close();  } } |
| **public class** BIOClient {   **public static void** main(String[] args) **throws** Exception{  String serverIp = **"127.0.0.1"**;  **int** serverPort = 2001;  Socket socket = **new** Socket();  socket.connect(**new** InetSocketAddress(serverIp,serverPort));  FileInputStream fis = **new** FileInputStream(  **"/home/doqin/Downloads/NIOTest.tar.gz"**);  **byte**[] b = **new byte**[4096];  **int** readLen = 0;  **int** total = 0;  **long** start = System.*currentTimeMillis*();  **while** ((readLen = fis.read(b))>=0) {  socket.getOutputStream().write(b);  total = total + readLen;  }  System.***out***.println(**"BIO文件传输大小为： "** + total +**" B, 耗时："** +  (System.*currentTimeMillis*() - start) + **" ms"**);  socket.close();  fis.close();  } } |

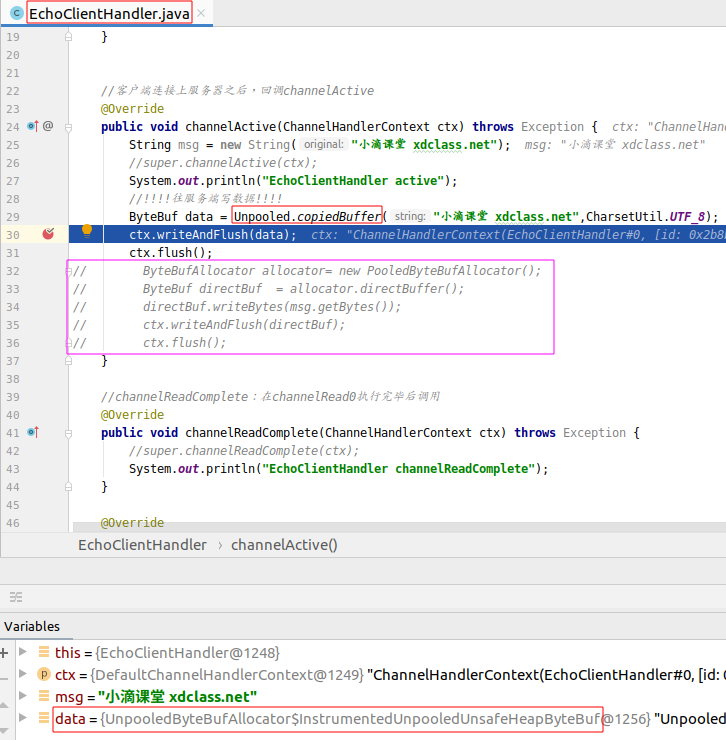
**BIO的实现源码如下：**

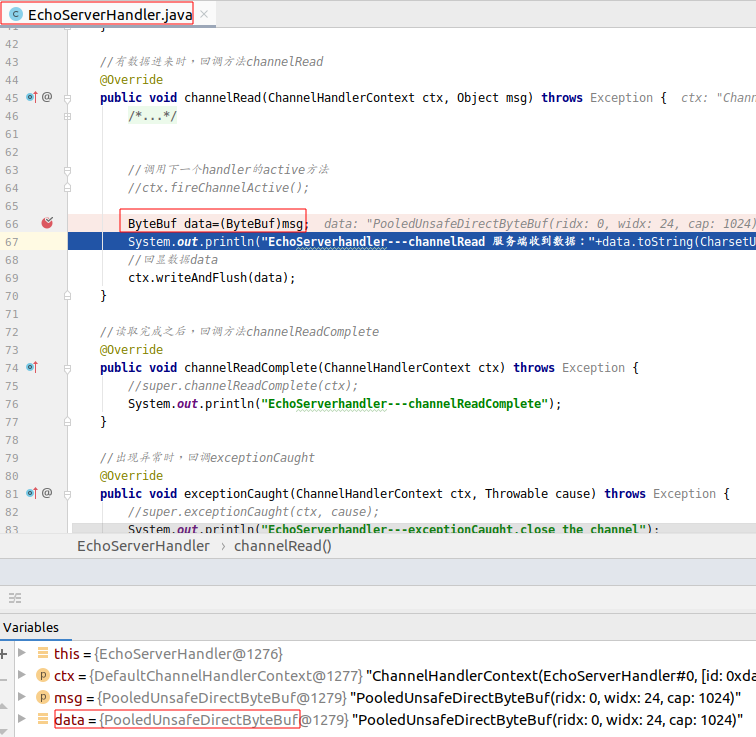
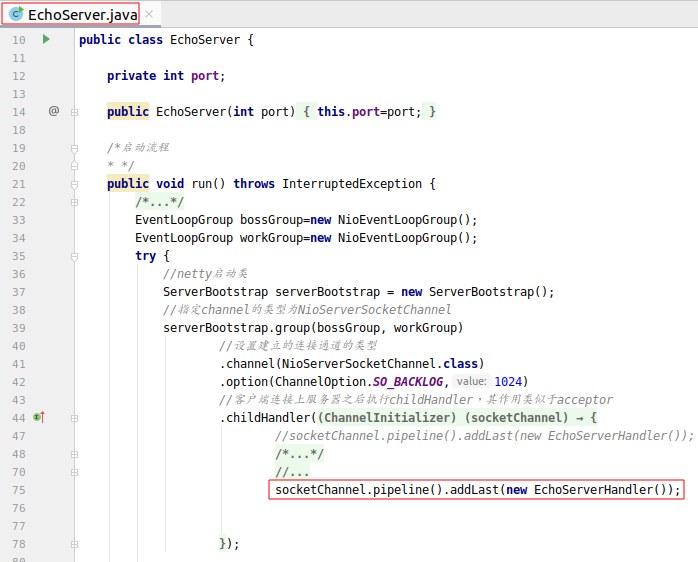
|  |
| --- |
| **public class** NIOServer {   **public static void** main(String[] args) **throws** Exception{  **int** port = 2000;  ServerSocketChannel ssc = ServerSocketChannel.*open*();  ssc.socket().bind(**new** InetSocketAddress(port));  *//ssc.configureBlocking(false);  //获取文件通道* FileChannel fileChannel = **new** FileOutputStream(**"/home/doqin/Downloads/NIOTest.tar.gz"**).getChannel();  *//等待连接* SocketChannel socketChannel = ssc.accept();  *//零拷贝核心操作,将通道流中的数据直接拷贝到文件/home/doqin/Downloads/fileChannel.tar.gz* fileChannel.transferFrom(socketChannel, 0, 1024\*1024\*10);   socketChannel.close();  fileChannel.close();  } } |
| **public class** NIOClient {   **public static void** main(String[] args) **throws** Exception{  String serverIp = **"127.0.0.1"**;  **int** serverPort = 2000;  SocketChannel socketChannel = SocketChannel.*open*();  socketChannel.connect(**new** InetSocketAddress(serverIp,serverPort));   *//从文件输入流中获取文件通道 ( FileChannel )* FileChannel fileChannel = **new** FileInputStream(  **"/home/doqin/Downloads/ideaIU-2019.2.4.tar.gz"**).getChannel();  **long** start = System.*currentTimeMillis*();  **long** len = fileChannel.transferTo(0, 1024\*1024\*10,socketChannel);  *//fileChannel.read();* System.***out***.println(**"NIO文件传输大小为： "** + len +**" B, 耗时："** +  (System.*currentTimeMillis*() - start) + **" ms"**);  socketChannel.close();  fileChannel.close();  } } |

**三、零拷贝在Netty上的体现：**

**1.Netty的接受和发送ByteBuf默认采用DIRECT BUFFER**



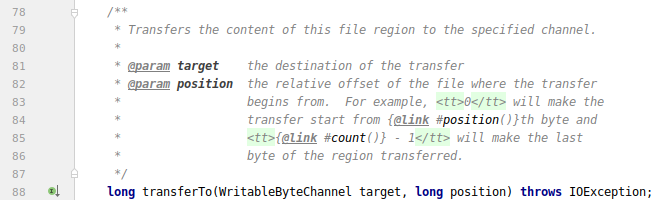




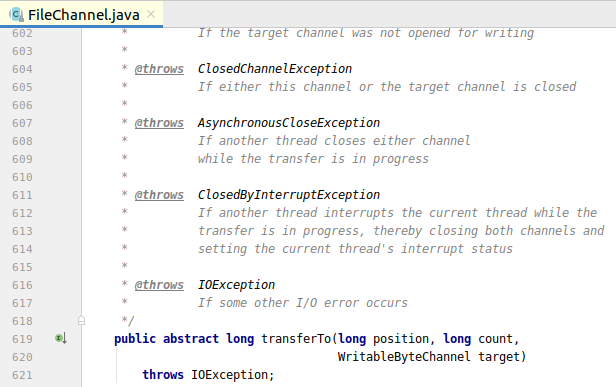
|  |
| --- |
| **public class** EchoServer {  **private int port**;   **public** EchoServer(**int** port){  **this**.**port**=port;  } **public void** run() **throws** InterruptedException {EventLoopGroup bossGroup=**new** NioEventLoopGroup();  EventLoopGroup workGroup=**new** NioEventLoopGroup();  **try** {  *//netty启动类* ServerBootstrap serverBootstrap = **new** ServerBootstrap();  *//指定channel的类型为NioServerSocketChannel* serverBootstrap.group(bossGroup, workGroup)  *//设置建立的连接通道的类型* .channel(NioServerSocketChannel.**class**)  .option(ChannelOption.***SO\_BACKLOG***,1024)  *//客户端连接上服务器之后执行childHandler，其作用类似于acceptor* .childHandler(**new** ChannelInitializer<SocketChannel>() {  *//有客户端连接上服务器时，回调initChannel* **protected void** initChannel(SocketChannel socketChannel) **throws** Exception {socketChannel.pipeline().addLast(**new EchoServerHandler()**);  }  });  System.***out***.println(**"Echo 服务器启动ing"**);  *//绑定端口，同步等待成功* ChannelFuture channelFuture = serverBootstrap.bind(**port**).sync();  *//等待服务端监听端口关闭* channelFuture.channel().closeFuture().sync();  } **finally** {  *//优雅退出，释放线程池* workGroup.shutdownGracefully();  bossGroup.shutdownGracefully();  }  }   **public static void** main(String[] args) **throws** InterruptedException {  **int** port=8080;  **new** EchoServer(port).run();  }} |
| **public class EchoServerHandler** **extends** ChannelInboundHandlerAdapter {   *//有数据进来时，回调方法channelRead* @Override  **public void channelRead**(ChannelHandlerContext ctx, Object msg) **throws** Exception {ByteBuf data=(ByteBuf)msg;  System.***out***.println(**"EchoServerhandler---channelRead 服务端收到数据："**+data.toString(CharsetUtil.***UTF\_8***));  *//回显数据data* ctx.writeAndFlush(data);  } } |
| **public class ClientHandler** **extends** ChannelInboundHandlerAdapter { @Override  **public void channelActive**(ChannelHandlerContext ctx) **throws** Exception {  String message=**"Netty is a NIO client server framework which enables quick&\_"** +**" and easy development of network applcations&\_"** +**" It greatly simplifies and streamlines&\_"** *//如果最后一段的末尾不加分隔符&\_，那么此段会因为缺少分隔符而丢失。* +**" network programming such as TCP and UDP socket server.&\_"**;   ByteBuf msg=**null**;  msg=Unpooled.*buffer*(message.getBytes().**length**);  msg.writeBytes(message.getBytes());  ctx.writeAndFlush(msg);   }     @Override  **public void** exceptionCaught(ChannelHandlerContext ctx, Throwable cause) **throws** Exception {  *//super.exceptionCaught(ctx, cause);* System.***out***.println(**"ClientHandler exceptionCaught"**);  cause.printStackTrace();  ctx.close();  } } |
| **public class** EchoClient {   **private** String **host**;  **private int port**;   **public** EchoClient(String host,**int** port){  **this**.**host**=host;  **this**.**port**=port;  }   **public void** start() **throws** InterruptedException {  EventLoopGroup group=**new** NioEventLoopGroup();  **try**{  Bootstrap bootstrap=**new** Bootstrap();  bootstrap.group(group)  .channel(NioSocketChannel.**class**)  .remoteAddress(**new** InetSocketAddress(**host**,**port**))  .handler(**new** ChannelInitializer<SocketChannel>() {  **protected void** initChannel(SocketChannel ch) **throws** Exception {  ch.pipeline().addLast(**new ClientHandler()**);  }  });  *//连接到服务器，connect是异步连接。再调用同步方法sync，等待连接成功* ChannelFuture channelFuture=bootstrap.connect().sync();  *//阻塞直到客户端通道关闭* channelFuture.channel().closeFuture().sync();   }**finally**{  group.shutdownGracefully();  }  }   **public static void** main(String[] args) **throws** InterruptedException {  **new** EchoClient(**"127.0.0.1"**,8080).start();  } } |

**2.FileRegion:transferTo()-----避免数据流经用户空间**

Screenshot from 2020-10-21 17-37-19







1. **CompositeByteBuf:对多个ByteBuffer的一个"视图"，将它们逻辑上当成一个完整的ByteBuffer来操作，免去了重新分配空间再复制数据的开销。**

<https://segmentfault.com/a/1190000007560884>

