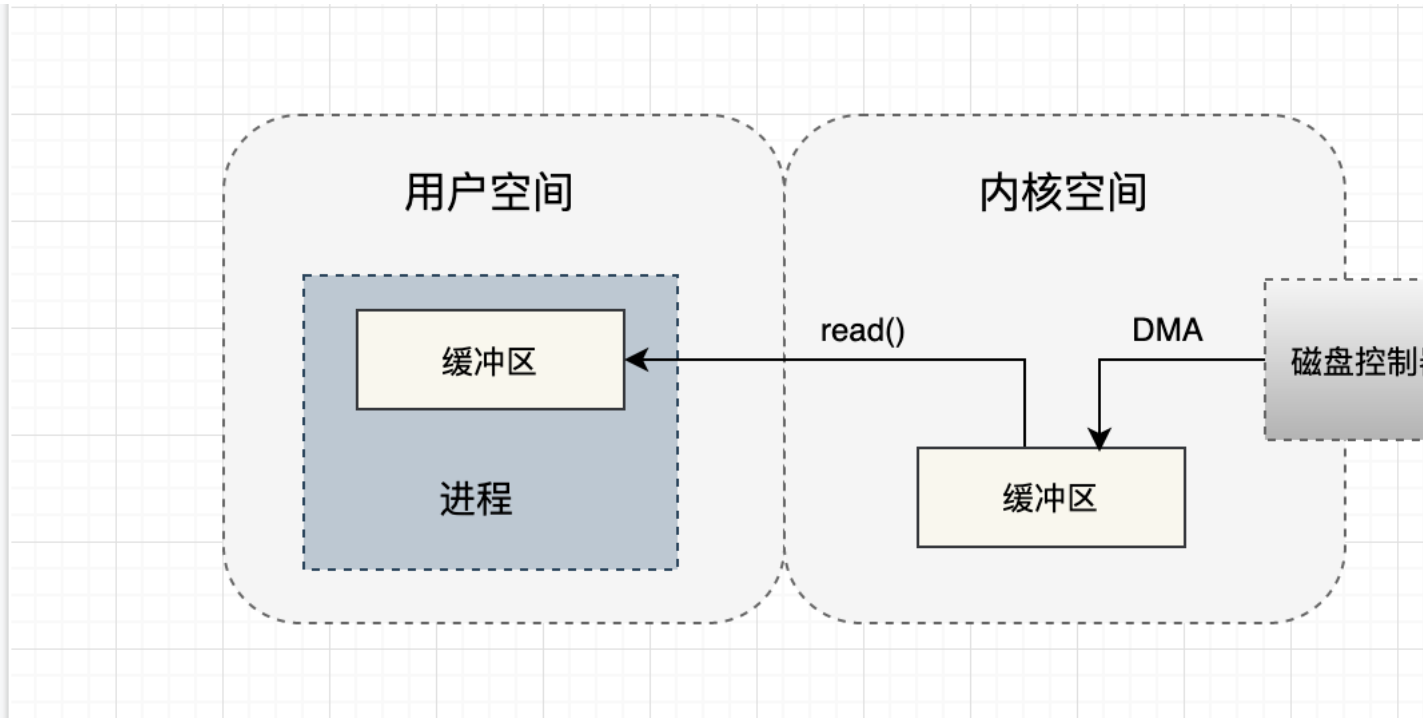
我们也经常在 Java NIO，Netty，Kafka，RocketMQ 等框架中听到零拷贝，它经常作为其提升性能的一大亮点；下面从 I/O 的几个概念开始，进而再分析零拷贝。

1|0 I/O 概念

1|1 缓冲区

缓冲区是所有 I/O 的基础，I/O 讲的无非就是把数据移进或移出缓冲区；进程执行 I/O 操作，就是向操作系统发出请求，让它要么把缓冲区的数据排干(写)，要么填充缓冲区(读)。

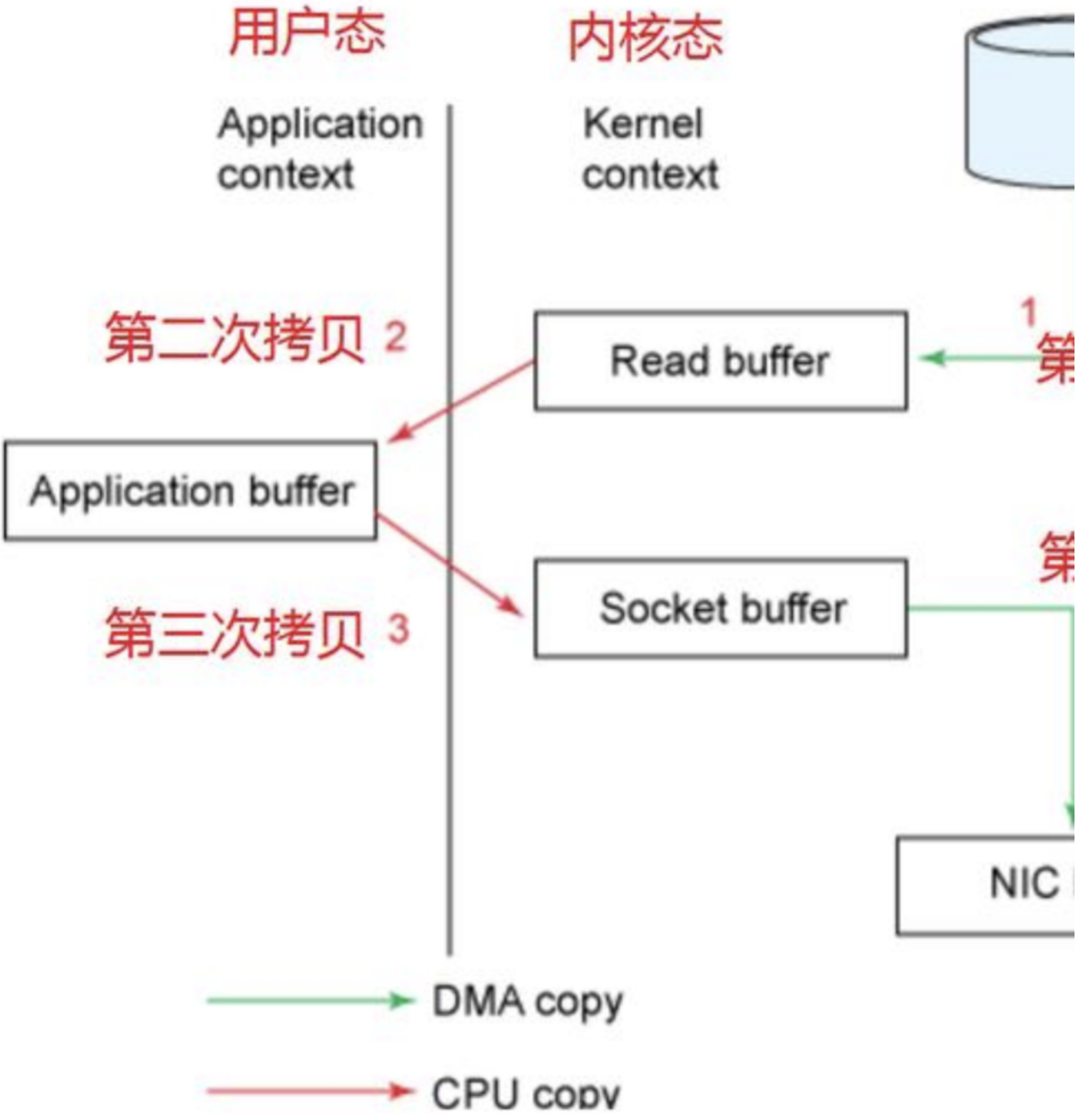
下面看一个 Java 进程发起 Read 请求加载数据大致的流程图：



进程发起 Read 请求之后，内核接收到 Read 请求之后，会先检查内核空间中是否已经存在进程所需要的数据，如果已经存在，则直接把数据 Copy 给进程的缓冲区。

如果没有内核随即向磁盘控制器发出命令，要求从磁盘读取数据，磁盘控制器把数据直接写入内核 Read 缓冲区，这一步通过 DMA 完成。

接下来就是内核将数据 Copy 到进程的缓冲区；如果进程发起 Write 请求，同样需要把用户缓冲区里面的数据 Copy 到内核的 Socket 缓冲区里面，然后再通过 DMA 把数据 Copy 到网卡中，发送出去。



现在我们可以看到 1 -> 2 -> 3 -> 4 的整个过程一共经历了四次拷贝的方式，但是真正消耗资源和浪费时间的是第二次和第三次，因为这两次都需要经过我们的CPU拷贝，而且还需要内核态和用户态之间的来回切换。想想看，我们的CPU资源是多么宝贵，每次都需要把内核空间的数据拷贝到用户空间中，要处理大量的任务还要去拷贝大量的数据。如果能把CPU的这两次拷贝给去除掉，岂不快哉！！！既能节省CPU资源，还可以避免内核态和用户态之间的切换，所以零拷贝的出现就是为了解决这种问题的。

关于零拷贝提供了两种方式分别是：

- **mmap + write**
- **Sendfile**

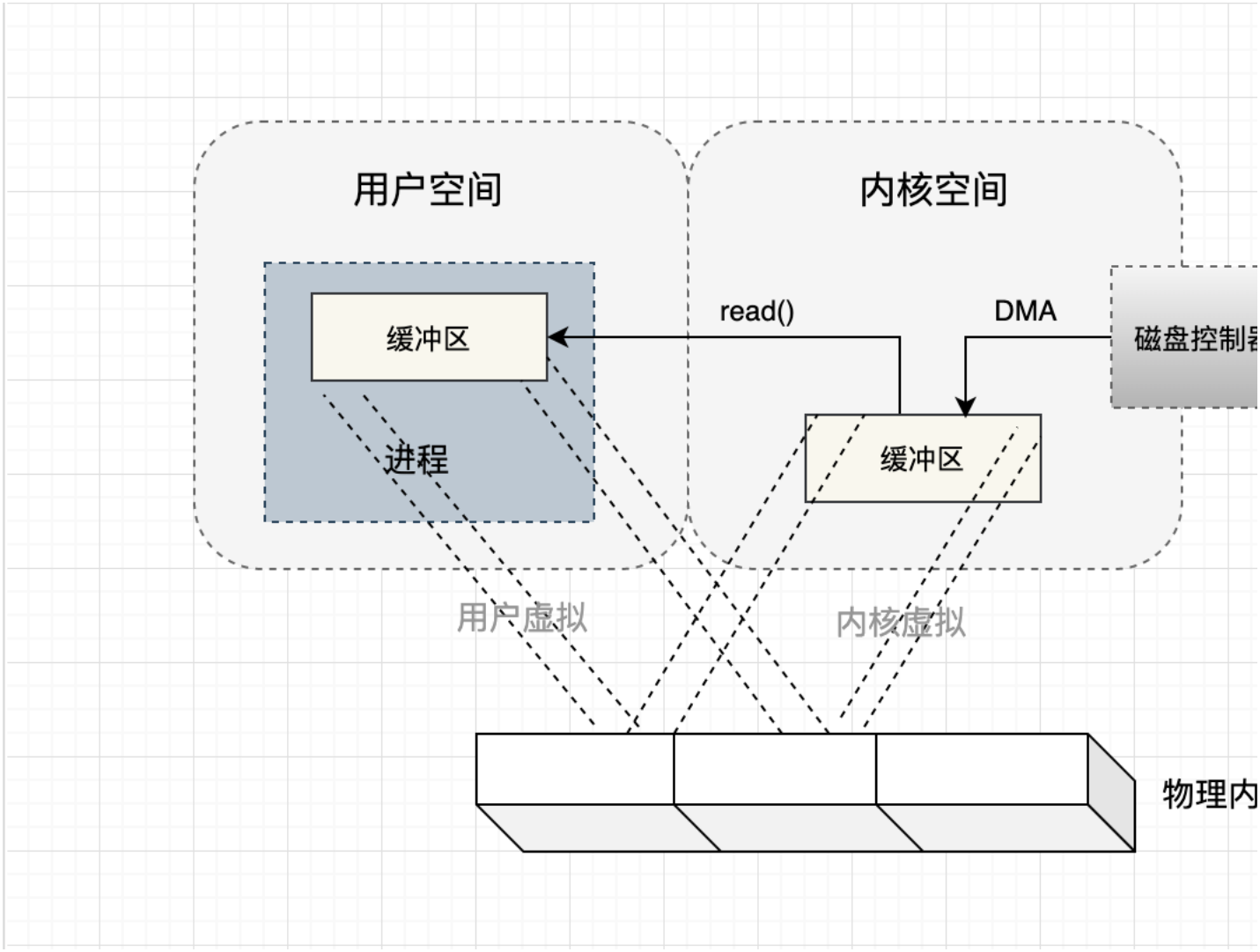
1|2 虚拟内存

所有现代操作系统都使用虚拟内存，使用虚拟的地址取代物理地址，这样做的好处是：

- 一个以上的虚拟地址可以指向同一个物理内存地址。
- 虚拟内存空间可大于实际可用的物理地址。

利用第一条特性可以把内核空间地址和用户空间的虚拟地址映射到同一个物理地址，这样 DMA 就可以填充对内核和用户空间进程同时可见的缓冲区了。

大致如下图所示：



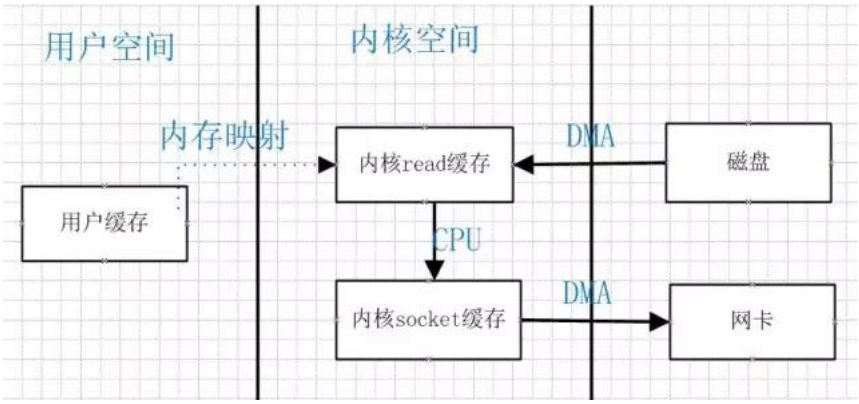
省去了内核与用户空间的往来拷贝，Java 也利用操作系统的此特性来提升性能，下面重点看看 Java 对零拷贝都有哪些支持。

1|3 mmap+write 方式

使用 mmap+write 方式代替原来的 read+write 方式，mmap 是一种内存映射文件的方法，即将一个文件或者其他对象映射到进程的地址空间，实现文件磁盘地址和进程虚拟地址空间中一段虚拟地址的一一对应关系。

这样就可以省掉原来内核 Read 缓冲区 Copy 数据到用户缓冲区，但是还是需要内核 Read 缓冲区将数据 Copy 到内核 Socket 缓冲区。

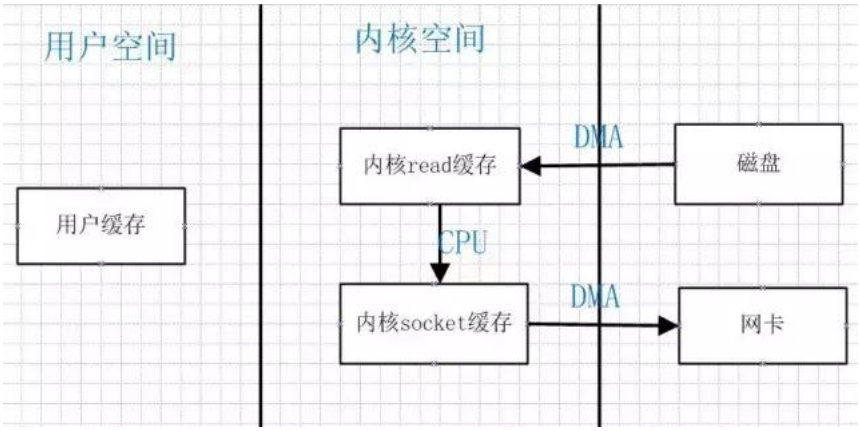
大致如下图所示：



1|4 Sendfile 方式

Sendfile 系统调用在内核版本 2.1 中被引入，目的是简化通过网络在两个通道之间进行的数据传输过程。

Sendfile 系统调用的引入，不仅减少了数据复制，还减少了上下文切换的次数，大致如下图所示：



sendfile系统调用在两个文件描述符之间直接传递数据(完全在内核中操作)，从而避免了数据在内核缓冲区和用户缓冲区之间的拷贝，操作效率很高，被称之为零拷贝。

sendfile() 系统调用利用 DMA 引擎将文件中的数据拷贝到操作系统内核缓冲区中，然后数据被拷贝到与 socket 相关的内核缓冲区中去。接下来，DMA 引擎将数据从内核 socket 缓冲区中拷贝到协议引擎中去。

sendfile() 系统调用不需要将数据拷贝或者映射到应用程序地址空间中去，所以 sendfile() 只是适用于应用程序地址空间不需要对所访问数据进行处理的情况。因为 sendfile 传输的数据没有越过用户应用程序 / 操作系统内核的边界线，所以 sendfile () 也极大地减少了存储管理的开销。

简单归纳上述的过程：

- sendfile系统调用利用DMA引擎将文件数据拷贝到内核缓冲区，之后数据被拷贝到内核socket缓冲区中。
- DMA引擎将数据从内核socket缓冲区拷贝到协议引擎中。

这里没有用户态和内核态之间的切换，也没有内核缓冲区和用户缓冲区之间的拷贝，大大提升了传输性能。

2|0 Java 零拷贝

2|1 MappedByteBuffer

Java NIO 提供的 FileChannel 提供了 map() 方法，该方法可以在一个打开的文件和 MappedByteBuffer 之间建立一个虚拟内存映射。

MappedByteBuffer 继承于 ByteBuffer，类似于一个基于内存的缓冲区，只不过该对象的数据元素存储在磁盘的一个文件中。

调用 get() 方法会从磁盘中获取数据，此数据反映该文件当前的内容，调用 put() 方法会更新磁盘上的文件，并且对文件做的修改对其他阅读者也是可见的。

下面看一个简单的读取实例，然后再对 MappedByteBuffer 进行分析：

```
public class MappedByteBufferTest {

    public static void main(String[] args) throws Exception {
        File file = new File("D://db.txt");
        long len = file.length();
        byte[] ds = new byte[(int) len];
        MappedByteBuffer mappedByteBuffer = new FileInputStream(file).getChannel().map(FileChannel.MapMode.READ_ONLY, 0,
                len);
        for (int offset = 0; offset < len; offset++) {
            byte b = mappedByteBuffer.get();
            ds[offset] = b;
        }
        Scanner scan = new Scanner(new ByteArrayInputStream(ds)).useDelimiter(" ");
        while (scan.hasNext()) {
            System.out.print(scan.next() + " ");
        }
    }
}
```

主要通过 FileChannel 提供的 map() 来实现映射，map() 方法如下：

```
public abstract MappedByteBuffer map(MapMode mode, long position, long size) throws IOException;
```

分别提供了三个参数，MapMode，Position 和 Size，分别表示：

- MapMode：映射的模式，可选项包括：READ_ONLY，READ_WRITE，PRIVATE。
- Position：从哪个位置开始映射，字节数的位置。
- Size：从 Position 开始向后多少个字节。

重点看一下 MapMode，前两个分别表示只读和可读可写，当然请求的映射模式受到 Filechannel 对象的访问权限限制，如果在一个没有读权限的文件上启用 READ_ONLY，将抛出 NonReadableChannelException。

PRIVATE 模式表示写时拷贝的映射，意味着通过 put() 方法所做的任何修改都会导致产生一个私有的数据拷贝并且该拷贝中的数据只有 MappedByteBuffer 实例可以看到。

该过程不会对底层文件做任何修改，而且一旦缓冲区被施以垃圾收集动作(garbage collected)，那些修改都会丢失。

大致浏览一下 map() 方法的源码：

```
public MappedByteBuffer map(MapMode mode, long position, long size)
       throws IOException
   {
           ...省略...
           int pagePosition = (int)(position % allocationGranularity);
           long mapPosition = position - pagePosition;
           long mapSize = size + pagePosition;
           try {
               // If no exception was thrown from map0, the address is valid
               addr = map0(imode, mapPosition, mapSize);
           } catch (OutOfMemoryError x) {
               // An OutOfMemoryError may indicate that we've exhausted memory
               // so force gc and re-attempt map
               System.gc();
               try {
                   Thread.sleep(100);
               } catch (InterruptedException y) {
                   Thread.currentThread().interrupt();
               }
               try {
                   addr = map0(imode, mapPosition, mapSize);
               } catch (OutOfMemoryError y) {
                   // After a second OOME, fail
                   throw new IOException("Map failed", y);
               }
           }

           // On Windows, and potentially other platforms, we need an open
           // file descriptor for some mapping operations.
           FileDescriptor mfd;
           try {
               mfd = nd.duplicateForMapping(fd);
           } catch (IOException ioe) {
               unmap0(addr, mapSize);
               throw ioe;
           }

           assert (IOStatus.checkAll(addr));
           assert (addr % allocationGranularity == 0);
           int isize = (int)size;
           Unmapper um = new Unmapper(addr, mapSize, isize, mfd);
           if ((!writable) || (imode == MAP_RO)) {
               return Util.newMappedByteBufferR(isize,
                                                addr + pagePosition,
                                                mfd,
                                                um);
           } else {
               return Util.newMappedByteBuffer(isize,
                                               addr + pagePosition,
                                               mfd,
                                               um);
           }
   }
```

大致意思就是通过 Native 方法获取内存映射的地址，如果失败，手动 GC 再次映射。

最后通过内存映射的地址实例化出 MappedByteBuffer，MappedByteBuffer 本身是一个抽象类，其实这里真正实例化出来的是 DirectByteBuffer。

2|2 DirectByteBuffer

DirectByteBuffer 继承于 MappedByteBuffer，从名字就可以猜测出开辟了一段直接的内存，并不会占用 JVM 的内存空间。

上一节中通过 Filechannel 映射出的 MappedByteBuffer 其实际也是 DirectByteBuffer，当然除了这种方式，也可以手动开辟一段空间：

```
ByteBuffer directByteBuffer = ByteBuffer.allocateDirect(100);
```

如上开辟了 100 字节的直接内存空间。

2|3 Channel-to-Channel 传输

经常需要从一个位置将文件传输到另外一个位置，FileChannel 提供了 transferTo() 方法用来提高传输的效率，首先看一个简单的实例：

```
public class ChannelTransfer {
    public static void main(String[] argv) throws Exception {
        String files[]=new String[1];
        files[0]="D://db.txt";
        catFiles(Channels.newChannel(System.out), files);
    }

    private static void catFiles(WritableByteChannel target, String[] files)
            throws Exception {
        for (int i = 0; i < files.length; i++) {
            FileInputStream fis = new FileInputStream(files[i]);
            FileChannel channel = fis.getChannel();
```

```
            channel.transferTo(0, channel.size(), target);
            channel.close();
            fis.close();
        }
    }
}
```

通过 FileChannel 的 transferTo() 方法将文件数据传输到 System.out 通道，接口定义如下：

```
public abstract long transferTo(long position, long count, WritableByteChannel target) throws IOException;
```

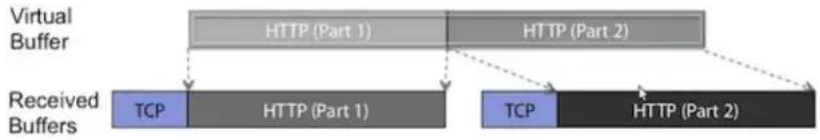
几个参数也比较好理解，分别是开始传输的位置，传输的字节数，以及目标通道；transferTo() 允许将一个通道交叉连接到另一个通道，而不需要一个中间缓冲区来传递数据。

注：这里不需要中间缓冲区有两层意思：第一层不需要用户空间缓冲区来拷贝内核缓冲区，另外一层两个通道都有自己的内核缓冲区，两个内核缓冲区也可以做到无需拷贝数据。

3|0 Netty 零拷贝

Netty 提供了零拷贝的 Buffer，在传输数据时，最终处理的数据会需要对单个传输的报文，进行组合和拆分，NIO 原生的 ByteBuffer 无法做到，Netty 通过提供的 Composite(组合)和 Slice(拆分)两种 Buffer 来实现零拷贝。

看下面一张图会比较清晰：



TCP 层 HTTP 报文被分成了两个 ChannelBuffer，这两个 Buffer 对我们上层的逻辑(HTTP 处理)是没有意义的。

但是两个 ChannelBuffer 被组合起来，就成为了一个有意义的 HTTP 报文，这个报文对应的 ChannelBuffer，才是能称之为“Message”的东西，这里用到了一个词“Virtual Buffer”。

可以看一下 Netty 提供的 CompositeChannelBuffer 源码：

```
public class CompositeChannelBuffer extends AbstractChannelBuffer {

    private final ByteOrder order;
    private ChannelBuffer[] components;
    private int[] indices;
    private int lastAccessedComponentId;
    private final boolean gathering;

    public byte getByte(int index) {
        int componentId = componentId(index);
        return components[componentId].getByte(index - indices[componentId]);
    }
    ...省略...
}
```

Components 用来保存的就是所有接收到的 Buffer，Indices 记录每个 buffer 的起始位置，lastAccessedComponentId 记录上一次访问的 ComponentId。

CompositeChannelBuffer 并不会开辟新的内存并直接复制所有 ChannelBuffer 内容，而是直接保存了所有 ChannelBuffer 的引用，并在子 ChannelBuffer 里进行读写，实现了零拷贝。

4|0 其他零拷贝

RocketMQ 的消息采用顺序写到 commitlog 文件，然后利用 consume queue 文件作为索引。

RocketMQ 采用零拷贝 mmap+write 的方式来回应 Consumer 的请求。

同样 Kafka 中存在大量的网络数据持久化到磁盘和磁盘文件通过网络发送的过程，Kafka使用了 Sendfile 零拷贝方式。

5|0 总结

零拷贝如果简单用 Java 里面对象的概率来理解的话，其实就是使用的都是对象的引用，每个引用对象的地方对其改变就都能改变此对象，永远只存在一份对象。