通过上面的序列图，我们大概能够了解到 Netty 的 I/O 线程 NioEventLoop 聚合了 Selector，可以同时并发处理成百上千个客户端 Channel，而且它的读写操作都是非阻塞的，这可以大幅提升 I/O 线程的运行效率，避免由于频繁 I/O 阻塞导致的线程挂起。另外，由于 Netty 采用的是异步通信模式，单个 I/O 线程也可以并发处理多个客户端连接和读写操作，所以从根本上解决了传统 BIO 的单连接单线程模型的弊端，使整个系统的性能、弹性伸缩性能和可靠性都得到了极大的提升。

6.2.2 零拷贝

Netty 的零拷贝主要体现在如下三个方面。

（1）Netty 接收和发送 ByteBuffer 采用 DirectBuffer，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。如果使用传统的堆存（Heap Buffer）进行 Socket 读写，那么 JVM 会将堆存拷贝一份到直接内存中，然后才写入 Socket。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 那样方便地对组合 Buffer 进行操作，避免了传统的通过内存拷贝的方式将几个小 Buffer 合并成一个大 Buffer 的烦琐操作。

（3）Netty 中文件传输采用了 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。

下面我们针对上述三种对零拷贝的描述在源码中进行验证，以加深理解。先看第一种 Netty 对于堆外直接内存的使用，AbstractNioByteChannel$NioByteUnsafe 的源码如下。

```
public final void read() {
    final ChannelConfig config = config();
    final ChannelPipeline pipeline = pipeline();
    final ByteBufAllocator allocator = config.getAllocator();
    final RecvByteBufAllocator.Handle allocHandle = recvBufAllocHandle();
    allocHandle.reset(config);

    ByteBuf byteBuf = null;
    boolean close = false;
    try {
        do {
            byteBuf = allocHandle.allocate(allocator);
            allocHandle.lastBytesRead(doReadBytes(byteBuf));
            if (allocHandle.lastBytesRead() <= 0) {
                    // nothing was read. release the buffer.
                    byteBuf.release();
```