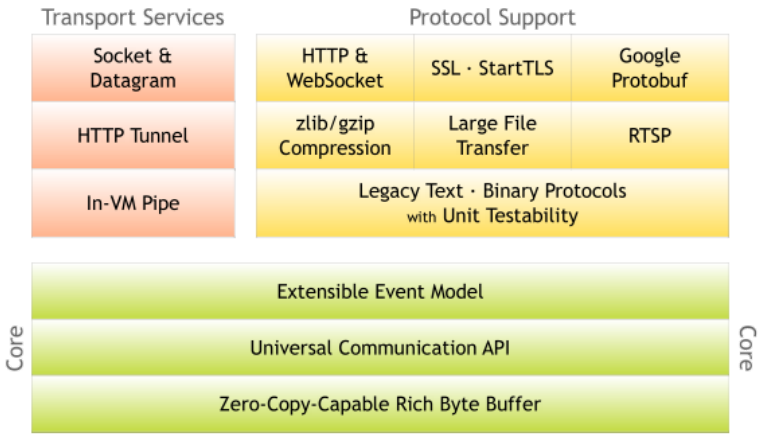
Netty4技术精讲

一、Netty简介



1、简介

- 版本：Netty4.X
- Netty是基于NIO的异步网络通信框架
- 能快速的搭建高性能易扩展的网络应用程序（服务器/客户端）

2、特征

设计

- 适用于各种传输类型的统一API-阻塞和非阻塞套接字
- 基于灵活且可扩展的事件模型，可将关注点明确分离
- 高度可定制的线程模型-单线程，一个或多个线程池
- 真正的无连接数据报套接字支持（从3.1开始）

性能

- 更高的吞吐量，更低的延迟
- 减少资源消耗
- 减少不必要的内存复制

安全

- Complete SSL/TLS and StartTLS support
- 完整的SSL / TLS和StartTLS支持

而如果对这些特点进行细化，则可以得出：

1. 基于事件机制（Pipeline - Handler）达成关注点分离（消息编解码，协议编解码，业务处理）
2. 可定制的线程处理模型，单线程，多线程池等
3. 屏蔽NIO本身的bug
4. 性能上的优化
5. 相较于NIO接口功能更丰富
6. 对外提供统一的接口，底层支持BIO与NIO两种方式自由切换

3、核心模块

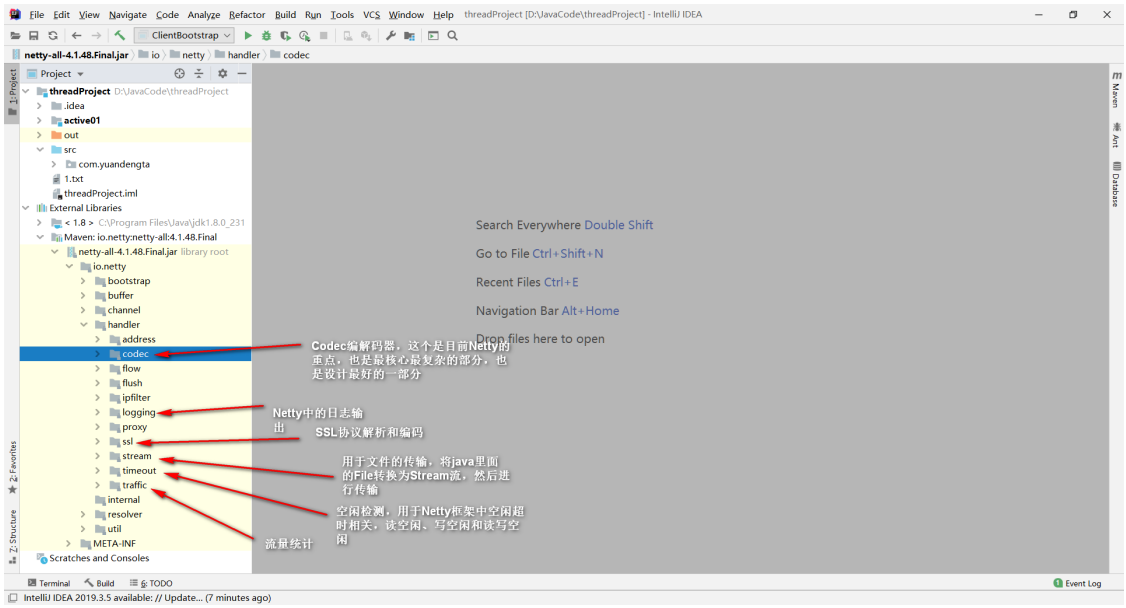
- Extensible Event Model:可扩展的事件模型
- Universal Communication API : 通用的通讯API
- Zero-Copy-Capable Rich Byte Buffer : 零拷贝的字节缓冲区
- Transport Services
    - Socket&Datagrame:TPC、UDP传输实现
    - HTTP Tunnel:http传输协议实现
    - In-VM pipe：内部VM传输实现
- Protocol Support
    - http/SSL/Google
    - 压缩、大文件传输协议、实时流传输协议

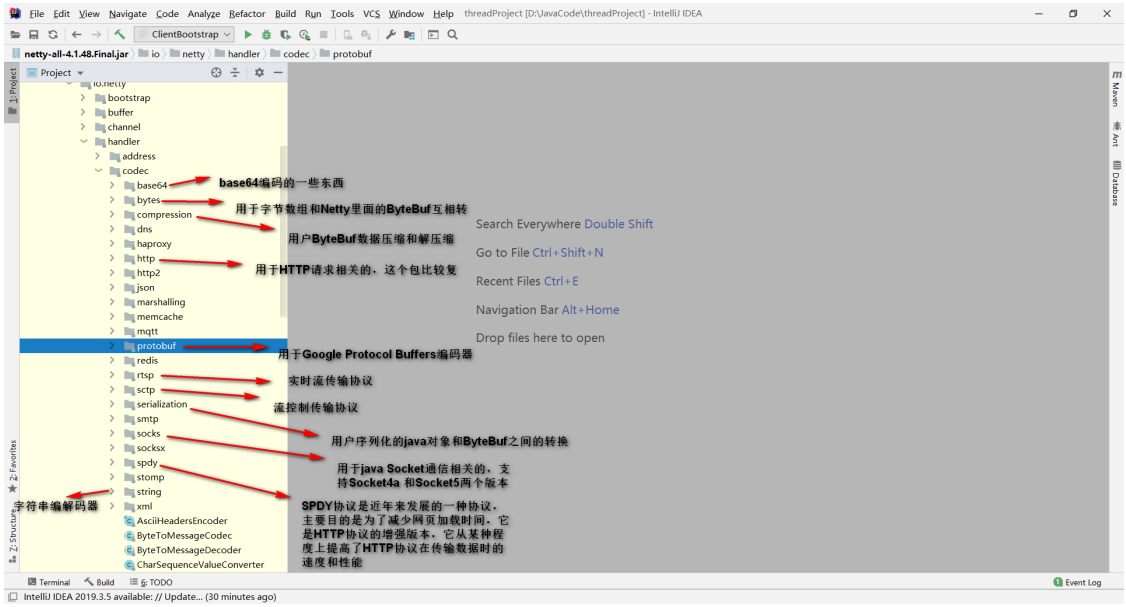
1、Netty无疑是开发网络应用的拿手菜，不不需要他多关注复杂的NIO模型和底层网络的细节，使用其丰富的接口，可以很容易实现复杂的通讯功能

2、Netty的开发工作并不集中在Netty本身，更多体现在保证服务的高可靠性和稳定性上

3、对于广大的应用开发者来说，Netty的上手成本小，死挖底层并不会产生太多的收益

Netty包介绍





二、Netty应用领域

a、互联网行业

阿里分布式服务框架 Dubbo 的 RPC 框架使用 Dubbo 协议进行节点间通信，Dubbo 协议默认使用 Netty 作为基础通信组件，用于实现各进程节点之间的内部通信。除了 Dubbo 之外，淘宝的消息中间件 RocketMQ 的消息生产者和消息消费者之间，也采用 Netty 进行高性能、异步通信。
除了阿里系和淘宝系之外，很多其它的大型互联网公司或者电商内部也已经大量使用 Netty 构建高性能、分布式的网络服务器

b、游戏行业

无论是手游服务端、还是大型的网络游戏，Java 语言得到了越来越广泛的应用。Netty 作为高性能的基础通信组件，它本身提供了 TCP/UDP 和 HTTP 协议栈，非常方便定制和开发私有协议栈。账号登陆服务器、地图服务器之间可以方便的通过 Netty 进行高性能的通信

c、大数据领域

经典的 Hadoop 的高性能通信和序列化组件 Avro 的 RPC 框架，默认采用 Netty 进行跨节点通信，它的 Netty Service 基于 Netty 框架二次封装实现。大数据计算往往采用多个计算节点和一个/N个汇总节点进行分布式部署，各节点之间存在海量的数据交换。由于 Netty 的综合性能是目前各个成熟 NIO 框架中最高的，因此，往往会被选中用作大数据各节点间的通信。

d、企业软件

企业和 IT 集成需要 ESB，Netty 对多协议支持、私有协议定制的简洁性和高性能是 ESB RPC 框架的首选通信组件。事实上，很多企业总线厂商会选择 Netty 作为基础通信组件，用于企业的 IT 集成。

e、通信行业

Netty 的异步高性能、高可靠性和高成熟度的优点，使它在通信行业得到了大量的应用。

三、常用网络通信框架

a、Mina

Mina出身于开源界的大牛Apache组织。是 Apache 组织一个较新的项目，它为开发高性能和高可用性的网络应用程序提 供了非常便利的框架。当前发行的 Mina 版本2.04支持基于 Java NIO 技术的 TCP/UDP 应用程序开发、串口通讯程 序，Mina 所支持的功能也在进一步的扩展中。目前，正在使用Mina的应用包括：Apache Directory Project、AsyncWeb、AMQP（Advanced Message Queuing Protocol）、RED5 Server（Macromedia Flash Media RTMP）、ObjectRADIUS、 Openfire等等

b、Netty

Jboss，Netty是一款异步的事件驱动的网络应用框架和工具，用于快速开发可维护的高性能、高扩展性协议服务器和客户端。也就是说，Netty是一个NIO客户端/服务器框架，支持快速、简单地开发网络应用，如协议服务器和客户端。它极大简化了网络编程，如TCP和UDP套接字服务器

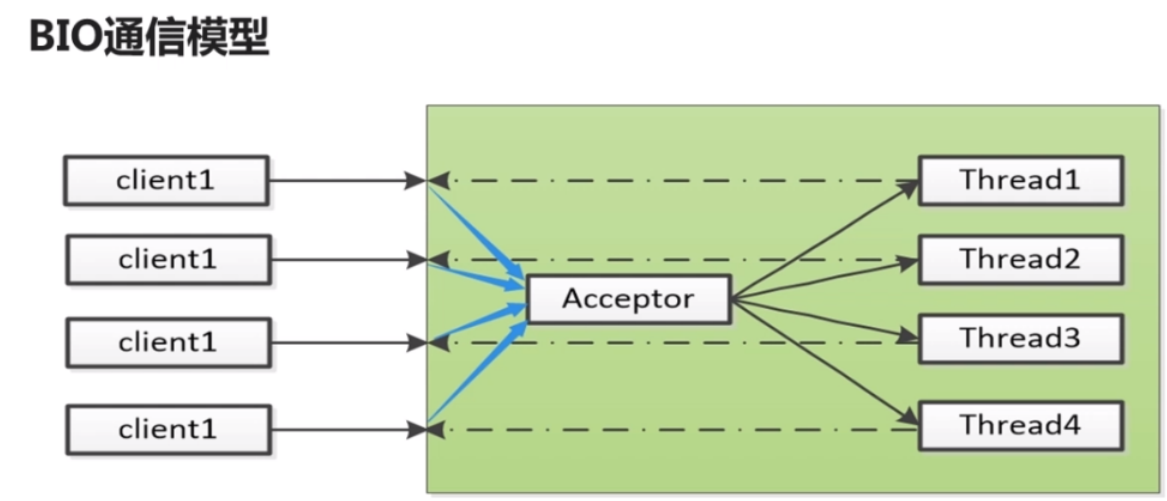
c、Grizzly

sun公司，Grizzly是一种应用程序框架，专门解决编写成千上万用户访问服务器时候产生的各种问题。使用JAVA NIO作为基础，并隐藏其编程的复杂性。容易使用的高性能的API。带来非阻塞socketd到协议处理层。利用高性能的缓冲和缓冲管理使用高性能的线程池。从设计的理念上来看，Mina的设计理念是最为优雅的。当然，由于Netty的主导作者与Mina的主导作者是同一人，出自同一人之手的Netty在设计理念上与Mina基本上是一致的。而Grizzly在设计理念上就较差了点，几乎是JavaNIO的简单封装。

mina将内核和一些特性的联系过于紧密，使得用户在不需要这些特性的时候无法脱离，相比下性能会有所下降； netty解决了这个设计问题

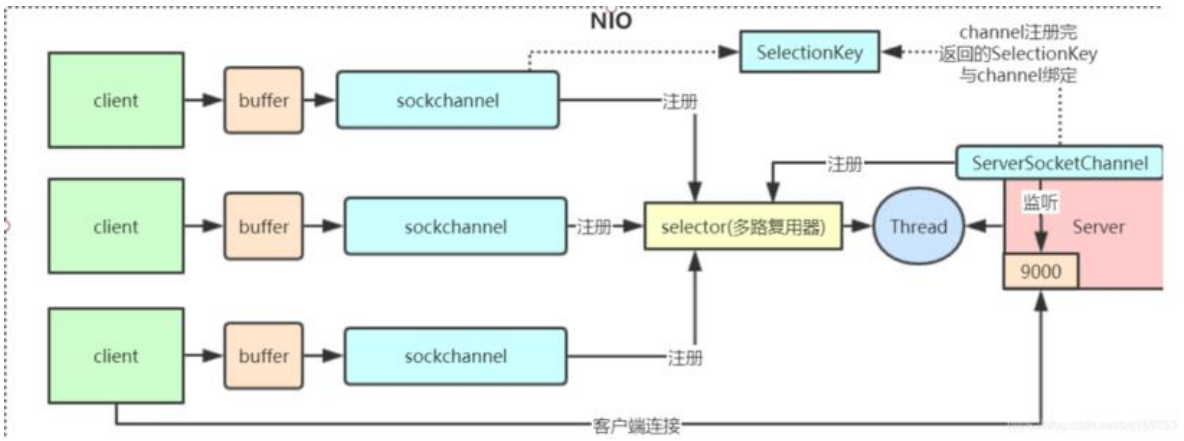
四、IO模型

a、BIO---阻塞I/O



- 一个线程负责连接，多线程则为每一个接入开启一个线程
- 一个请求一个应答
- 请求之后应答之前客户端会一直等待（阻塞）

b、NIO---同步非阻塞I/O



- 一个线程处理多个请求（连接），即客户端发送的连接请求都会注册到多路复用器上
- 多路复用器轮询到连接有I/O请求就进行处理

NIO的三大核心部分：

- Channel（通道）：和IO中的Stream是差不多一个等级的，只不过Stream是单向的，而Channel是双向的，既可以用来读操作，也可以用来写操作
- Buffer（缓冲区）：实际上是一个容器，一个连续数组，Channel提供从文件、网络读取数据的渠道，但是读写的数据都必须经过Buffer
- Selector（复用器）：将Channel注册到Selector上，Channel必须处于非阻塞模式下 **(channel.configureBlocking(false))**

NIO和传统的IO之间最大的一个区别就是，IO是面向流的，NIO是面向缓冲区的。java IO面向流意味着每次从流中读一个或多个字节，知道读取所有字节，它们没有被缓存在任何地方。

案例：

假设你是一个老师，让30个学生解答一道题目，然后检查学生做的是否正确，你有下面几个选择：

1. 第一种选择：按顺序逐个检查，先检查A，然后是B，之后是C、D……这中间如果有一个学生卡住，全班都会被耽误。这种模式就好比，你用循环挨个处理socket，根本不具有并发能力！
2. 第二种选择：你创建30个分身，每个分身检查一个学生的答案是否正确。这种类似于为每个用户创建一个进程或者线程处理连接。
3. 第三种选择：你站在讲台上等，谁解答完谁举手。这时C、D举手，表示他们解答问题完毕，你下去一次检查C、D的答案，然后继续回到讲台上等。此时E、A举手，然后又去处理E和A……。这种就是IO复用模型，Linux下的select、poll和epoll就是干这个的。讲用户socket对应的fd注册进epoll，然后epoll帮你监听哪些socket上有消息到达，这样就避免了大量无用操作。此时的socket应该采用非阻塞模式。这样整个过程只在调用select、poll、epoll这些调用的时候才会阻塞，收发客户小时是不会阻塞的，整个进程或者线程被充分利用起来，这就是事件驱动，所谓的reactor模式。

   问题：学生怎么主动举手?

   调用select函数后，内核会自动挂起线程，直到有一个或多个I/O事件发生后才将控制权还给程序，这就相当于学生举手了。

c、AIO---异步非阻塞

- NIO2,异步IO模型，底层完全使用异步回调的方式来实现，但是由于AIO这项技术在Linux操作系统上还不太成熟，所以通常也不会说太多关于这方面的内容

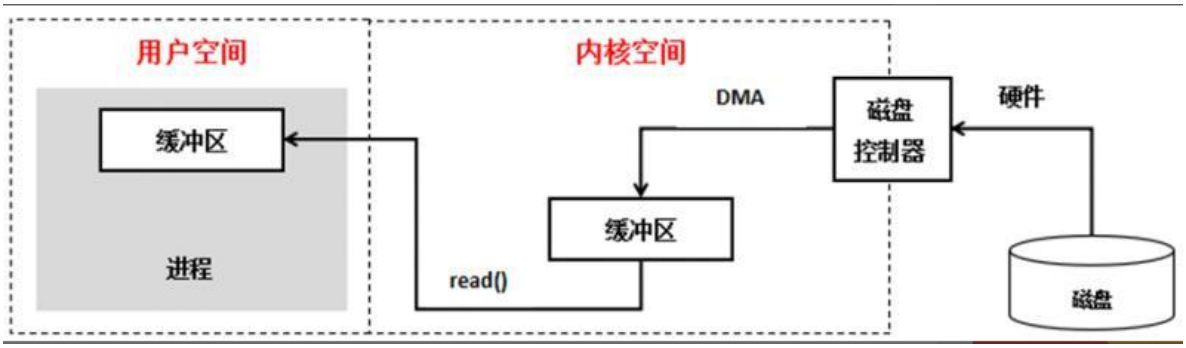
Linux I/O模型

在Linux中，将I/O模型分为五种类型：

- 阻塞式I/O模型
- 非阻塞式I/O模型
- I/O复用模型
- 信号驱动式 I/O 模型
- 异步 I/O模型

Linux中的 I/O 流程概括来说可以分为两步：

1. 等待数据准备好（waiting for the to be ready）
2. 从内核向进程复制数据(copying the data from the kernel to the process)



从上面的流程可以看出，首先是内核空间把数据从硬件（磁盘）中读到内核空间的缓冲区中，进行这一步操作时，数据处于内核态，通过磁盘驱动器，把数据从磁盘读到内核缓冲区来。下一步是把数据从内核缓冲区拷贝到用户空间的缓冲区，数据由内核态变为用户态。

知识普及：Linux中的程序大致运行在两个空间，自己的程序运行在用户空间，用户空间的权限有限，相对的另外一个空间是内核空间，比如驱动程序或者一些核心的系统调用都是在内核空间完成的。我们的I/O操作当在一个用户空间做系统调用的时候，数据会自动从用户态变到内核态来进行操作，完成了以后再从内核空间拷贝到用户空间。这种状态的切换是为了保证安全。比如在我们32位的操作系统中，大部分是4G的内存，一般前面一个G主要是内核态使用，后面三个G是用户态使用。

阻塞式 I/O 模型

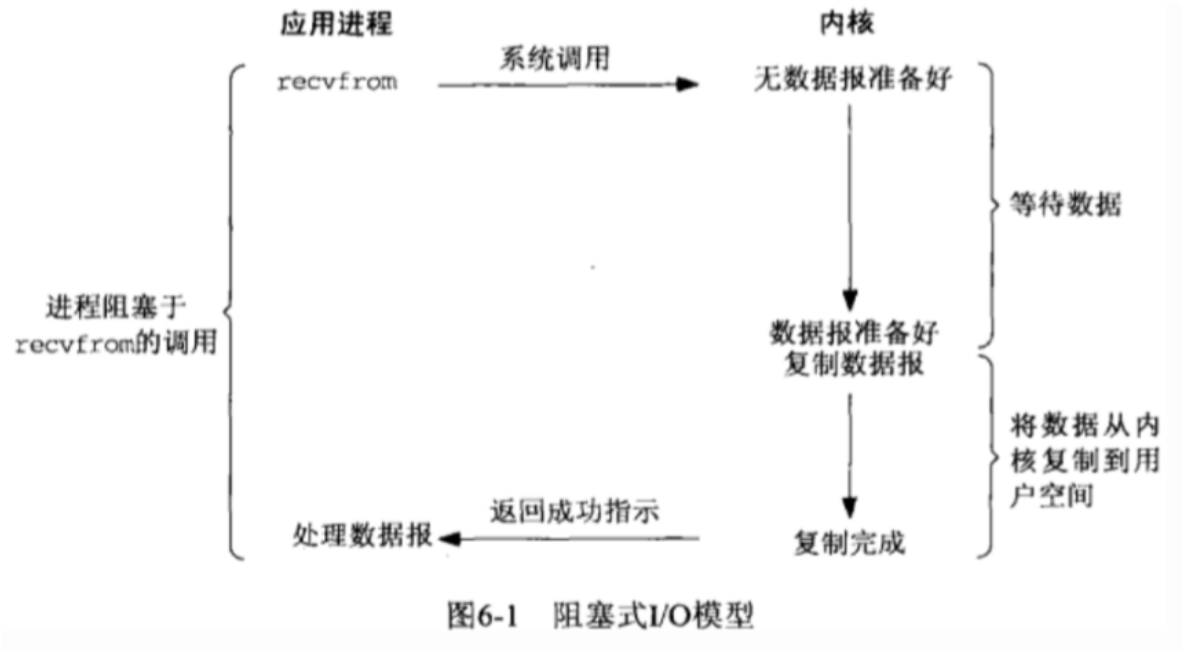


图6-1　阻塞式I/O模型

当用户进程调用recvfrom这个系统调用，kernel就开始了IO的第一个阶段：准备数据（对于网络IO来说，很多时候数据一开始还没有到达。比如，还没有收到一个完整的数据包，这个时候kernel就要等待足够的数据到来）。这个过程需要等待，也就是说数据被拷贝到操作系统内核的缓冲区中是需要一个过程的。而在用户进程这边，整个进程会被阻塞（当然，是进程自己选择的阻塞）。当kernel一直等到数

据准备好了，它就会将数据从kernel中拷贝到用户内存，然后kernel返回结果，用户进程才解除block的状态。所以，blocking IO的特点是在IO执行的两个阶段都被block了。这就是阻塞式IO模型。明显可见效率不会高。(**小明从家里面先到演唱会现场问售票业务员买票，但是票还没出来，三天以后才出来，小明直接打了个地铺睡在举办商售票大厅，一直等票出来，然后买票。**)

非阻塞式 I/O 模型

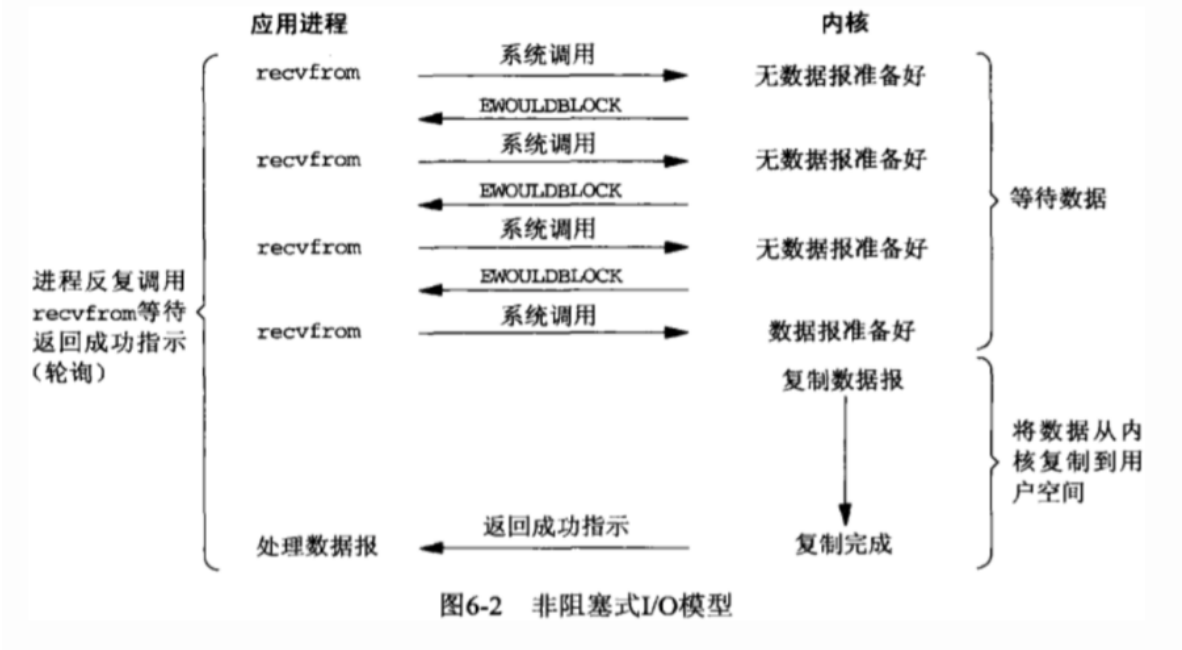
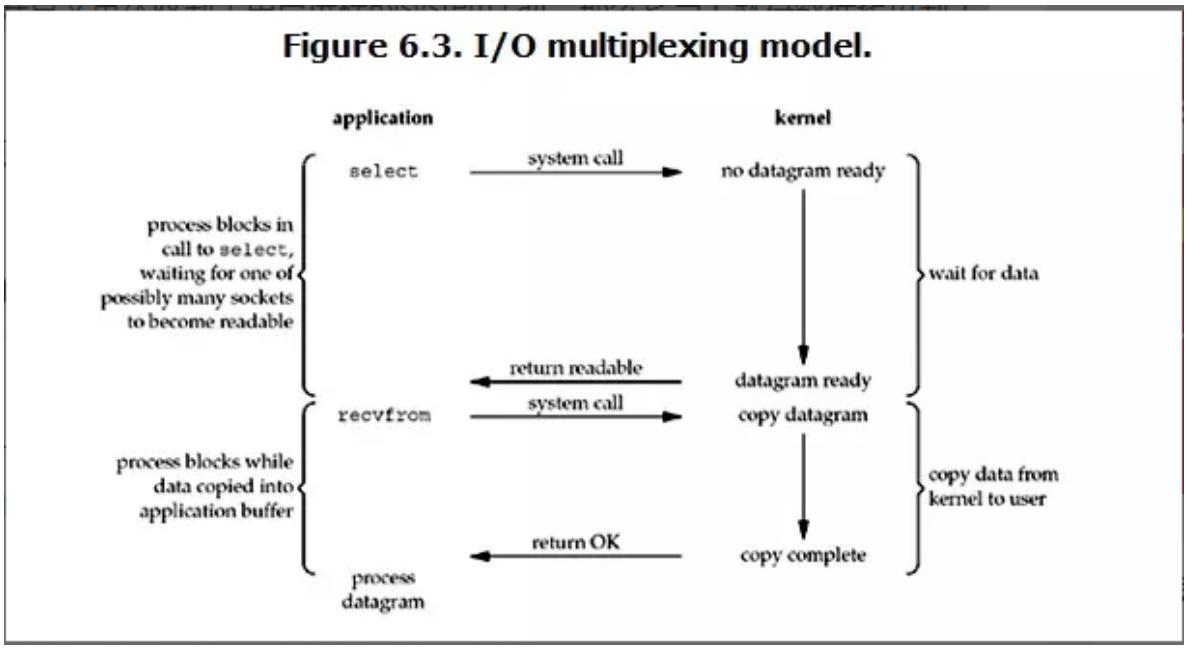


图6-2　非阻塞式I/O模型

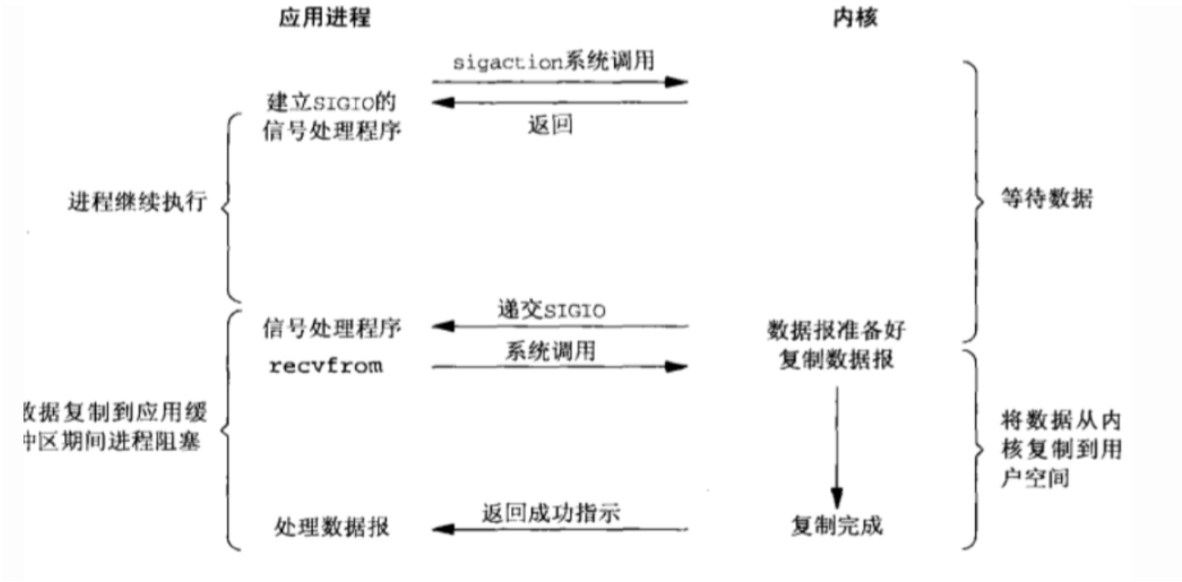
当用户进程发出read操作时，如果kernel中的数据还没有准备好，那么他并不会block用户进程，而是立刻返回一个error。从用户进程角度讲，他发起一个read操作后，并不需要等待，而是马上就得到一个结果，用户进程判断结果是一个error时，它就知道数据还没有准备好，于是它可以再次发送read操作，一旦kernel中的数据准备好了，并且又再次收到用户进程的system call，那么它马上就将数据拷贝到了用户内存，然后返回。所以，nonblocking IO的特点时用户进程需要不断的主动询问kernel数据好了没有。但是，从内核空间拷贝到用户空间，让数据从内核态变为用户态，这个过程在非阻塞式 I/O 模型中，也是阻塞的！所以效率也不会太高。(**小明从家里面先到演唱会现场问售票业务员买票，但是票还没出来，然后小明走了，办理其他事情去了，然后过了2个小时，又去举办商售票大厅买票来了，如果票还没有出来，小明又先去办其他事情了，重复上面的操作，直到有票可以买。**)

I/O 复用模型



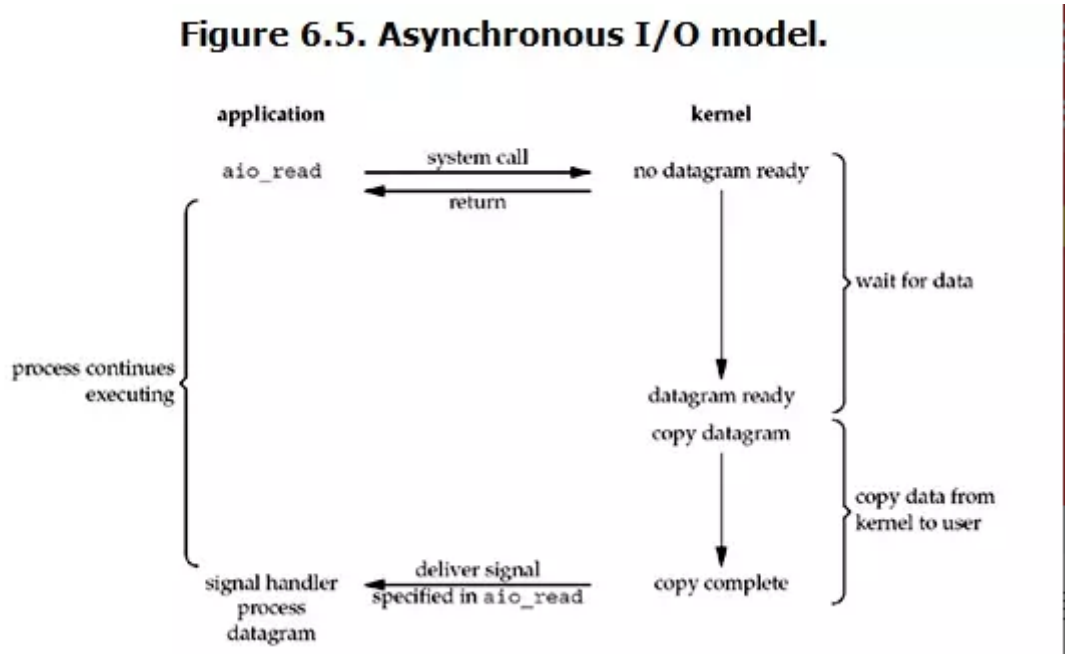
复用的意思是，系统一次去查看多个io的进度，看哪一个有了结果，对于有结果的，就开始执行下面从内核空间拷贝到用户空间的操作，这个模型和上面非阻塞式 I/O 模型的区别是，非阻塞式 I/O 模型一次只看一个结果好了没有，而IO复用模型一次可以查看多个，一次监控一批系统调用好了没有，这是复用模型的特点。在一定程度上，能提高一些效率。(**小明想买票看演唱会，都直接给黄牛(selector/epoll)打电话了，说帮我留意买个票，票买了通知我，我自己去取（当我接到黄牛的电话时，我需要花费整个路成的时间去读这个数据，买拿这个票），那么票没出来之前，小明完全可以做自己的事情。**）

信号驱动式 I/O 模型



在Linux系统中，有一种信号机制，也就是说调用方可以注册一个信号，当系统调用完成之后，可以通知这个信号，那注册信号的人就会知道这个请求已经完成了，这种信号机制应用在IO当中就是信号驱动式IO。这样就不是查看结果是否好了，而是被通知的一种方式。不过通知后，从内核态到用户态这个过程还是阻塞的(**小明想买票看演唱会，给举办商售票业务员说，给给你留个电话，有票了请你给我打个电话通知一下（是看人家操作系统提不提供这种功能，Linux提供，windows没有这种机制），我自己再来买票（小明完全可以做自己的事情，但是票还是需要小明自己去拿的**)

异步 I/O 模型

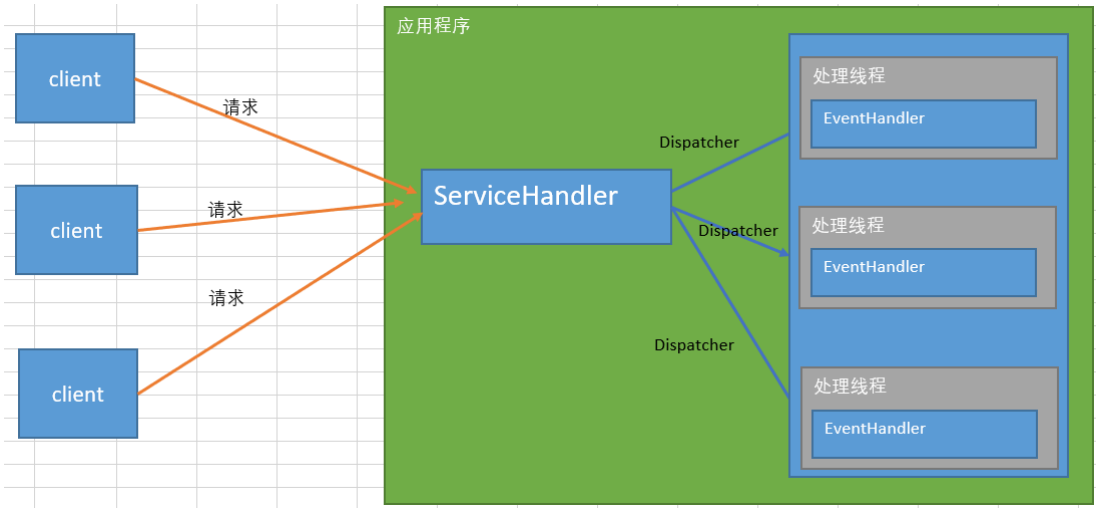


Linux下的asynchronous IO其实用的很少。‘

用户进程发起read操作之后，立刻就可以开始去做其它的事，而另一方面，从kernel的角度，当它受到一个asynchronous read之后，首先它会立刻返回，所以不会对用户进程产生任何block。然后，kernel会等待数据准备完成，然后将数据拷贝到用户内存，当这一切都完成之后，kernel会给用户进程发送一个signal，告诉它read操作完成了。(**小明想买票看演唱会,给举办商售票业务员说(异步非阻塞i/o)打电话了，给你留个地址，有票了请通知快递员，把这张票送到这个地址来，当小明听到敲门声，看见快递员，就知道票好了，而且指导票好了的时候，票已经到他手上了，票不用小明自己去取（应用不用自己再去read数据了）**）

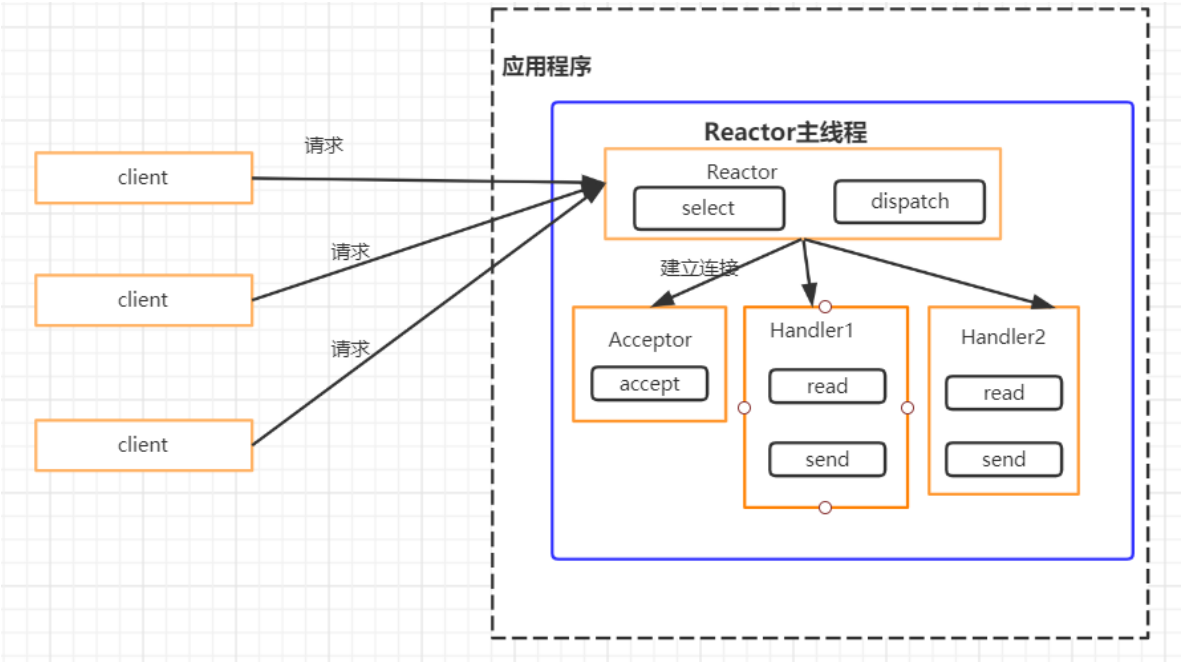
五、Netty线程模型

1、Reactor线程模型



- Reactor模式，通过一个或多个输入同时传递给服务处理器模式（基于事件驱动）
- 服务器程序处理传入的多个请求并将他们同步分派到相应的处理线程，因此Reactor模式也叫Dispatcher模式
- Reactor模式使用IO复用监听事件

2、Reactor单线程模型

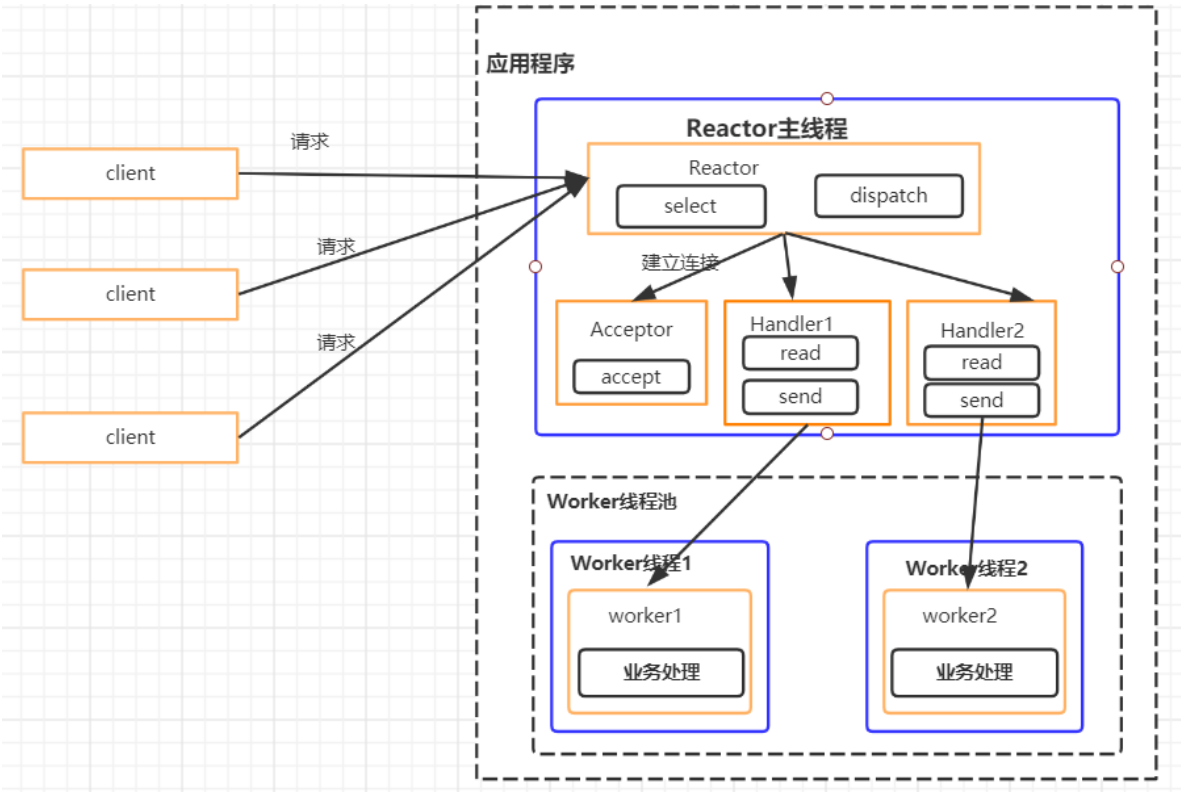


流程：

1. 其中客户端发送请求至服务器，Reactor响应其IO事件。
2. 如果是建立连接的请求，则分发至acceptor，由其接收连接，然后再将其注册至分发器
3. 如果是读请求，则分发至Handler，由其读出请求内容，然后对内容进行解码，然后处理运算，再对响应编码，最后发送响应。
4. 再整个过程中都是使用单线程，无论是Reactor线程和后续的Handler处理都只使用了一个线程。

Reactor单线程模型：
Reactor单线程模型仅使用一个线程来处理所有的事情，包括客户端的连接和服务器的连接，以及所有连接产生的读写时间，这种模型需要使用异步非阻塞I/O，使得每一个操作都不会发生阻塞，Handler为具体的处理时间的处理器，而Acceptor为连接的接收者，作为服务端接收来自客户端的链接请求。这样的线程模型理论上可以仅仅使用一个线程就完成所有的事件处理，显得线程的利用率非常高，而且因为只有一个线程在工作，所有不会产生多线程环境下会发生的各种多线程之间的并发问题。架构简单明了，线程模型的简单性决定了线程管理工作的简单性。但是这样的线程模型存在很多不足
缺点：
1、仅利用一个线程来处理使劲按，对于目前普遍多核处理器来说太过浪费资源 2、一个线程同时处理N个连接，管理起来较为复杂，而且性能也无法得到保证，这是以线程管理的简洁换取来的事件管理的复杂性，而且是在性能无 法得到保证的前提下换取的，在大流量的应用场景下根本没有实用性3、根据第二条，当处理的这个线程负载过重之后，处理速度会变慢，会有大量的事件堆积，甚至超时，而超时的情况下，客户端往往会重新发送请求，这样的情况下，这个单线程的模型就会成为整个系统的瓶颈 4、单线程模型的一个致命缺钱就是可靠性问题，因为仅有一个线程在工作，如果这个线程出错了无法正常执行任务了，那么整个系统就会停止响应，也就是系统会因为这个单线程模型而变得不可用，这在绝大部分场景（所有）下是不允许出现的

3、Reactor多线程模型

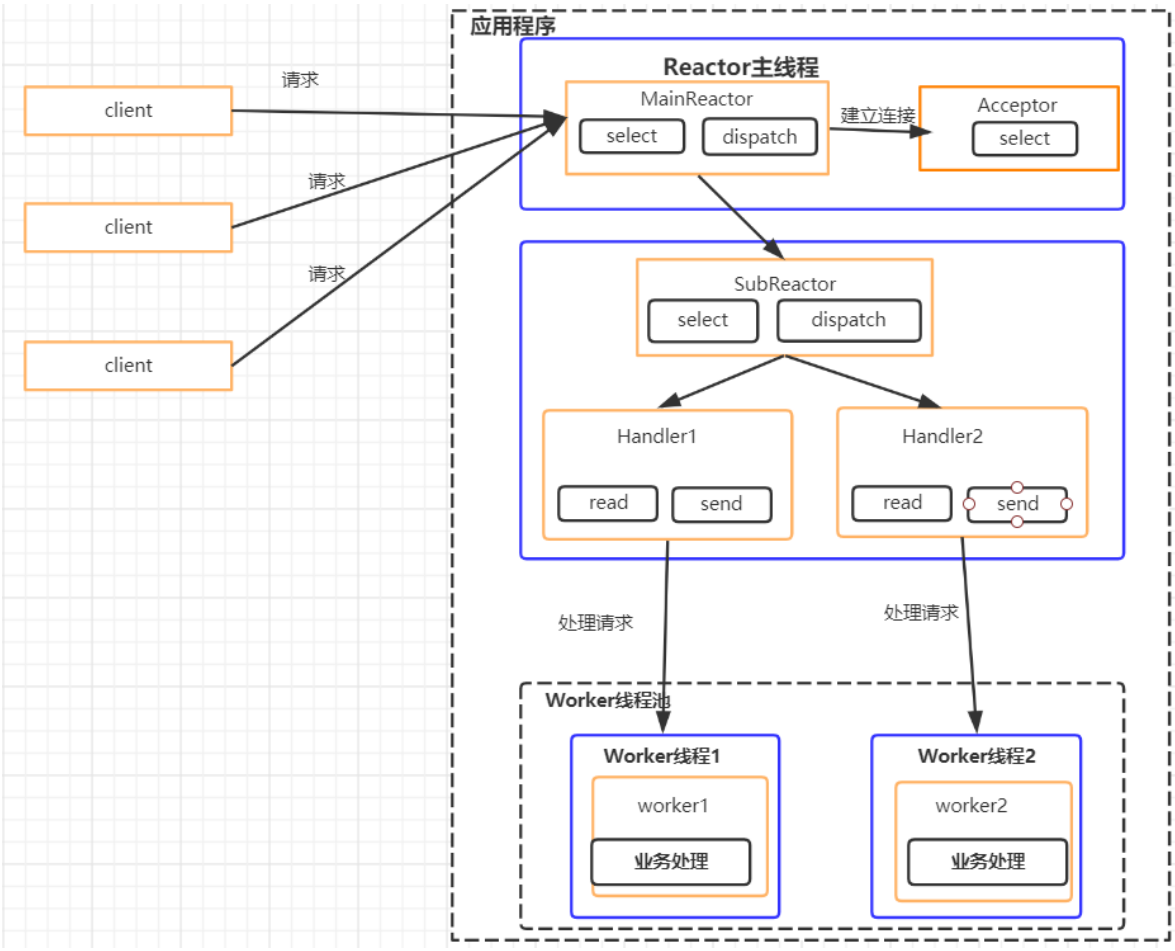


Reactor多线程模型：
多线程模型下，接收连接和处理请求作为两部分分离了，而Acceptor使用单独的线程来接收请求，做好准备后就交给事件处理的handler来处理，而handler使用了一个线程池来实现，这个线程池可以使用Executor框架实现的线程池来实现，所以，一个连接会交给一个handler线程来处理其上面所有的事件，需要注意的是：一个连接只会由一个线程来处理，而多个连接可能会由一个handler线程来处理，关键在于一个连接上的所有事件都只会由一个线程来处理，这样的好处就是消除了不必要的并发同步的麻烦
缺点：多线程模型下任然只有一个线程来处理客户端的连接请求，那如果这个线程挂了，那整个系统任然会变为不可用，而且，因为仅仅由一个线程来负责客户端的连接请求，如果连接之后要做一些验证之类复杂耗时操作再提交给handler线程来处理的话，就会出现性能问题。

- Selector对象通过select监控客户端请求事件，收到事件后，通过Dispatch进行分发
- 如果建立连接请求,则由Acceptor通过accept处理连接请求,然后创建一个Handler处理连接完成后的各种事件

  如果不是连接请求，则由reactor分发调用连接对应的Handler来处理
- Handler只负责响应事件，不做具体的业务处理，通过read读取数据后，会分发给后面的worker线程池的某个线程处理业务
- worker线程池会分配独立的线程完成真正的业务，并将结果返回给Handler。
- handler收到响应后，通过send将结果返回给client

4、Reactor主从线程模型



- Reactor主线程MainReactor对象通过select监听连接事件，收到事件后，通过Acceptor处理连接事件
- 当Acceptor处理连接事件后，MainReactor将连接分配给SubReaactor
- SubReactor将连接加入连接队列进行监听，并创建Handler进行各种事件处理
- 当有新事件发生时，subReactor就会调用对应的Handler处理
- handler通过read读取数据，分发给后面的worker线程处理
- worker线程池分配独立的线程worker线程进行业务处理，并返回结果
- handler收到响应结果后，再通过send将结果返回给client
- Reactor主线程可以对应多个Rreactor子线程，即MainReactor可以对应多个SubReactor

六、Netty核心组件及关系

1、Channel

IO操作的一种连接，实体与实体之间的连接，实体可以是硬件，文件，网络套接字，或者是程序组件

Channel 是 Netty 网络操作抽象类，它除了包括基本的 I/O 操作，如 bind、connect、read、write 之外，还包括了 Netty 框架相关的一些功能，如获取该 Channe l的 EventLoop。（这个 Channel 我们可以理解为 Socket 连接，它负责基本的 IO 操作，例如：bind()，connect()，read()，write() 等等。简单的说，Channel 就是代表连接，实体之间的连接，程序之间的连接，文件之间的连接，设备之间的连接。同时它也是数据入站和出站的载体）

生命周期

channelRegistered：channel注册到一个EventLoop

channelActive：channel变为活跃状态，可以接收和发送数据了

channelInactive：处于非活跃状态，没有连接到远程主机

channelUnregistered：channel已创建但未注册到一个EventLoop

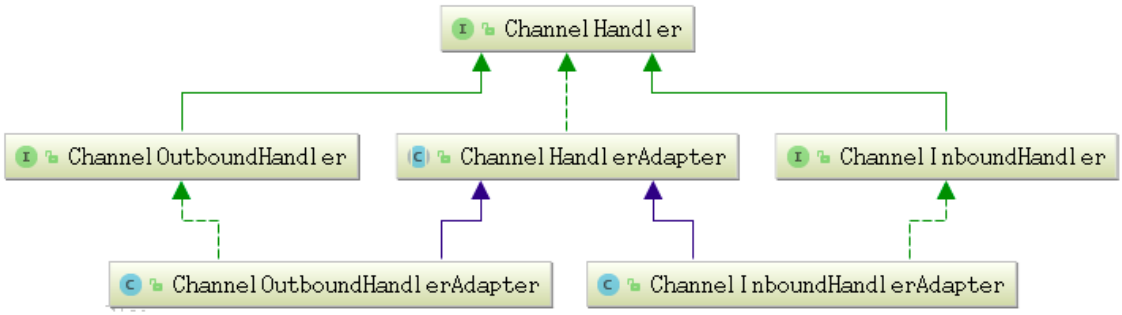
与流的区别

1、Channel可以同时支持读和写，而Stream只能单向的读或写

2、Channel支持异步读写，Stream通常只支持同步

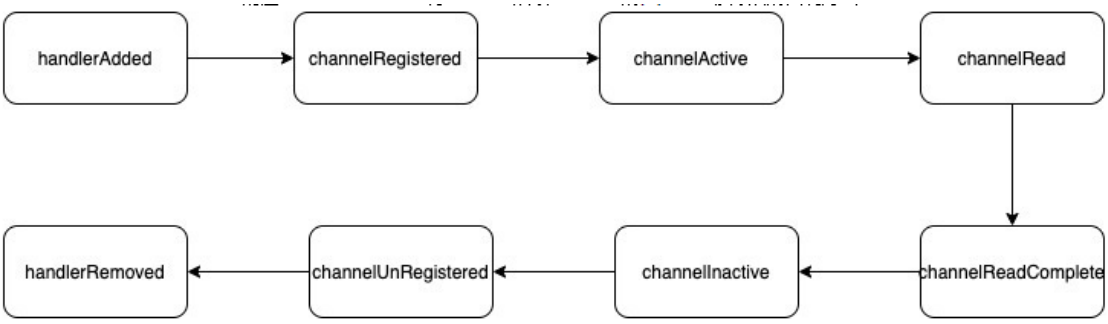
3、Channel总是读向Buffer，或者写自Buffer

2、ChannelHandler



- **ChannelInboundHandler**：处理进站数据和所有状态更改事件（进站指的是读操作等由通道引发的事件）

- **handlerAdded**:新建立的连接会按照初始化的策略，把handler添加到该channel的pipeline里面，也就是channel.pipeline.addLast(new XxxxHandler) 执行完后的回调
- **channelRegistered**：当channel已经注册到他的EventLoop并且能够处理I/O时被调用（当该连接分配到具体的worker线程后，该回调会被调用）
- **channelUnregistered**：对应channelRegistered，当连接关闭后，释放绑定的workder线程
- **channelActive**：channel的准备工作已经完成，所有的pipeline添加完成，并分配到具体的线上上，说明该channel准备就绪，可以使用了
- **channelInactive**：当连接断开时，该回调会被调用，说明这时候底层的TCP连接已经被断开了
- **channelReadComplete**：当channel上的一个读操作完成时调用
- **channelRead**：当从channel读取数据时调用
- **ChannelWritability**-Changed：当channel的可写状态发生改变时被调用
- **handlerRemoved**： 对应handlerAdded，将handler从该channel的pipeline移除后的回调方法。



- **ChannelOutboundHandler**：处理出站数据（写操作由用户触发的事件，发送到服务器的事件）
  - **bind**(ChannelHandlerContext var1, SocketAddress var2, ChannelPromise var3)
  - **connect**(ChannelHandlerContext var1, SocketAddress var2, SocketAddress var3, ChannelPromise var4)
  - **disconnect**(ChannelHandlerContext var1, ChannelPromise var2)
  - **close**(ChannelHandlerContext var1, ChannelPromise var2)
  - **deregister**(ChannelHandlerContext var1, ChannelPromise var2)
  - **read**(ChannelHandlerContext var1)
  - **write**(ChannelHandlerContext var1, Object var2, ChannelPromise var3)
  - **flush**(ChannelHandlerContext var1)

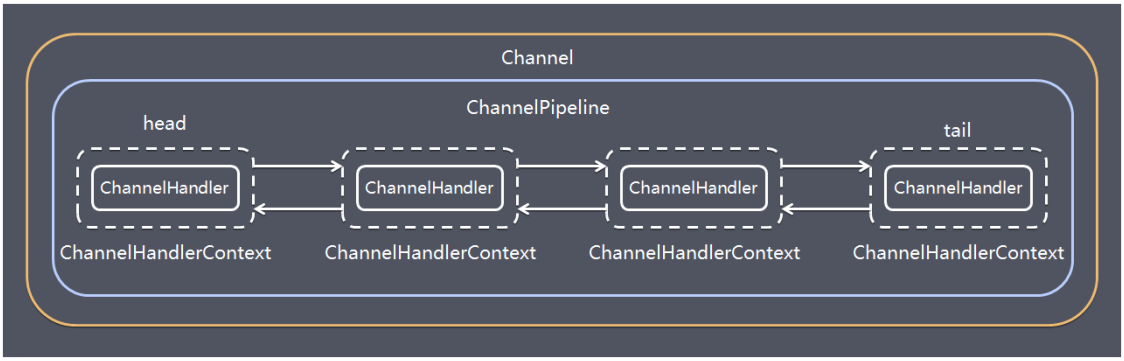
3、ChannelPipline

ChannelPipeline其实就是一个ChannelHandler容器，里面包括一系列的ChannelHandler 实例,用于拦截流经一个Channel 的入站和出站事件，每个Channel都有一个ChannelPipeline

- addFirst/addBefore/addAfter/addLast:添加ChannelHandler到ChannelPipeline
- remove：从ChannelPipeline移除ChannelHandler
- replace：在ChannelPipeline替换另外一个ChannelHandler

4、ChannelHandlerContext

- ChannelHandlerContext表示ChannelHandler 和ChannelPipeline 之间的关联，在ChannelHandler 添加到 ChannelPipeline 时创建ChannelHandlerContext表示两者之间的关系
- 每个ChannelHandler都对应一个ChannelHandlerContext,ChannelHandler之间其实没有联系，都是由ChannelHandlerContext关联起来的



压缩事件源：如果这个对象不需要经过其它的handler处理，就使用：ctx.writeAndFlush()
传递事件源：如果需要把ctx对象往前进行传播，需要经过后面的handler，调用：ctx.channel.writeAndFlush()

5、ChannelFuture

Netty 为异步非阻塞，即所有的 I/O 操作都为异步的，因此，我们不能立刻得知消息是否已经被处理了。Netty 提供了 ChannelFuture 接口，通过该接口的 addListener() 方法注册一个 ChannelFutureListener，当操作执行成功或者失败时，监听就会自动触发返回结果。

6、ChannelOption的含义以及使用的场景

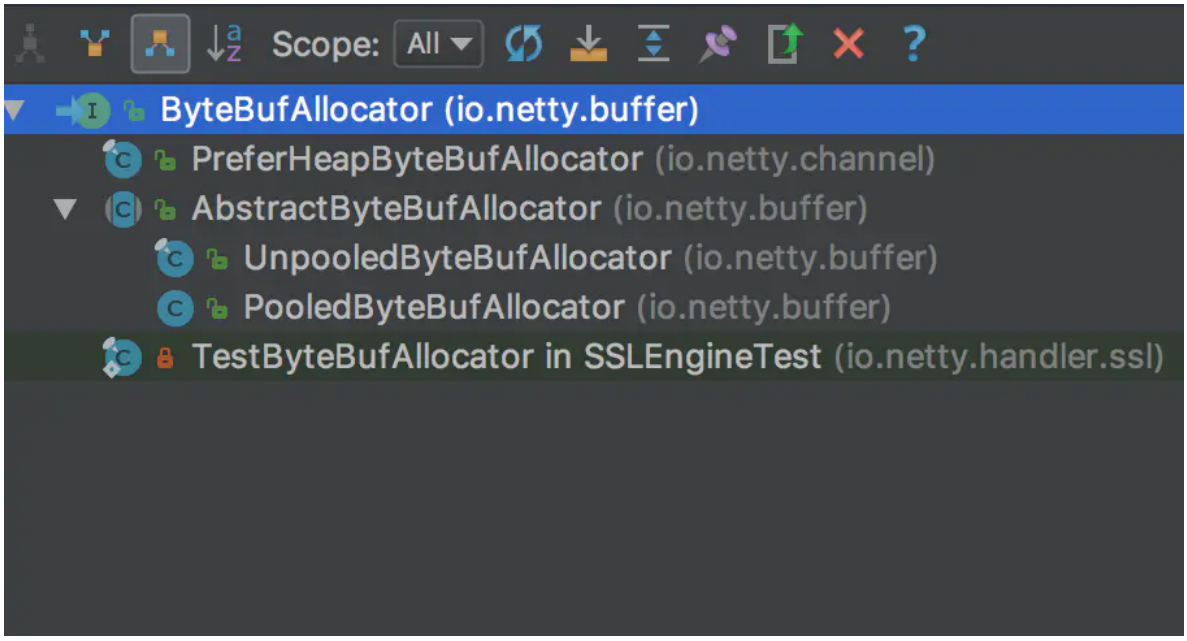
1. ChannelOption.SO_BACKLOG：ChannelOption.SO_BACKLOG对应的是tcp/ip协议listen函数中的backlog参数，函数listen(int socketfd,int backlog)用来初始化服务端可连接队列，服务端处理客户端连接请求是顺序处理的，所以同一时间只能处理一个客户端连接，多个客户端来的时候，服务端将不能处理的客户端连接请求放在队列中等待处理，`backlog参数指定了队列的大小`
2. ChannelOption.SO_REUSEADDR：ChanneOption.SO_REUSEADDR对应于套接字选项中的SO_REUSEADDR，这个参数表示允许`重复使用本地地址和端口`，比如，某个服务器进程占用了TCP的80端口进行监听，此时再次监听该端口就会返回错误，使用该参数就可以解决问题，该参数允许共用该端口，这个在服务器程序中比较常使用，比如某个进程非正常退出，该程序占用的端口可能要被占用一段时间才能允许其他进程使用，而且程序死掉以后，内核一需要一定的时间才能够释放此端口，不设置SO_REUSEADDR就无法正常使用该端口。
3. ChannelOption.SO_KEEPALIVE：Channeloption.SO_KEEPALIVE参数对应于套接字选项中的SO_KEEPALIVE，该参数用于设置TCP连接，`当设置该选项以后，连接会测试链接的状态，这个选项用于可能长时间没有数据交流的连接`。当设置该选项以后，如果在两小时内没有数据的通信时，TCP会自动发送一个活动探测数据报文。（是否启用心跳保活机制。在双方TCP套接字建立连接后（即都进入ESTABLISHED状态）并且在两个小时左右上层没有任何数据传输的情况下，这套机制才会被激活。）
4. ChannelOption.SO_SNDBUF和ChannelOption.SO_RCVBUF：ChannelOption.SO_SNDBUF参数对应于套接字选项中的SO_SNDBUF，ChannelOption.SO_RCVBUF参数对应于套接字选项中的SO_RCVBUF这两个参数用于操作`接收缓冲区和发送缓冲区的大小`，接收缓冲区用于保存网络协议站内收到的数据，直到应用程序读取成功，发送缓冲区用于保存发送数据，直到发送成功。
5. ChannelOption.SO_LINGER：ChannelOption.SO_LINGER参数对应于套接字选项中的SO_LINGER,Linux内核默认的处理方式是当用户调用close()方法的时候，函数返回，在可能的情况下，尽量发送数据，不一定保证会发生剩余的数据，造成了数据的不确定性，使用SO_LINGER可以阻塞close()的调用时间，直到数据完全发送
6. ChannelOption.TCP_NODELAY:ChannelOption.TCP_NODELAY参数对应于套接字选项中的TCP_NODELAY,该参数的使用与Nagle算法有关。Nagle算法是将小的数据包组装为更大的帧然后进行发送，而不是输入一次发送一次,因此在数据包不足的时候会等待其他数据的到了，组装成大的数据包进行发送，虽然该方式有效提高网络的有效负载，但是却造成了延时，而该参数的作用就是禁止使用Nagle算法，使用于小数据即时传输，于TCP_NODELAY相对应的是TCP_CORK，该选项是需要等到发送的数据量最大的时候，一次性发送数据，适用于文件传输。

7、ByteBuf

Netty的数据容器

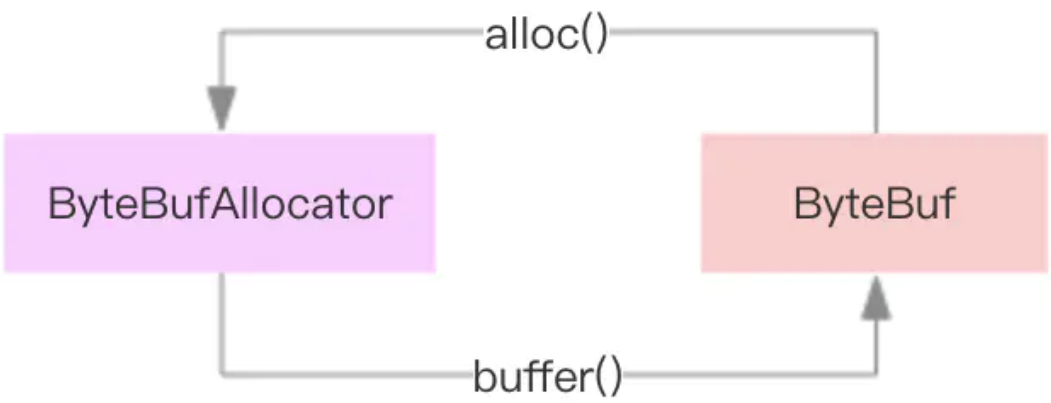
要想使用ByteBuf，首先肯定是要创建一个ByteBuf，更确切的说法就是要申请一块内存，后续可以在这块内存中执行写入数据读取数据等等一系列的操作。

那么如何创建一个ByteBuf呢？Netty中设计了一个专门负责分配ByteBuf的接口：ByteBufAllocator。该接口有一个抽象子类和两个实现类，分别对应了用来分配池化的ByteBuf和非池化的ByteBuf。**(池化：被许多用户重复使用，非池化：被单一用户使用，使用完后释放掉)**



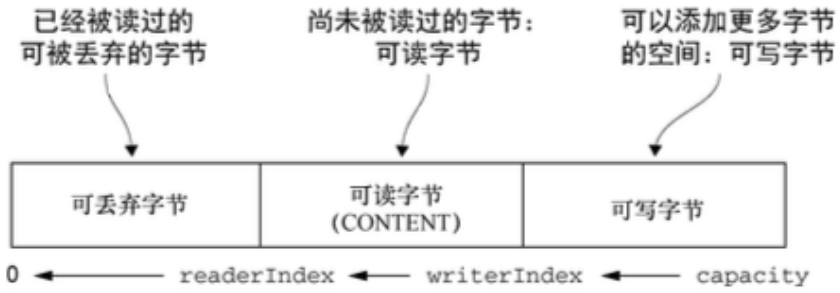
有了Allocator之后，Netty又为我们提供了两个工具类：Pooled、Unpooled，分类用来分配池化的和未池化的ByteBuf，进一步简化了创建ByteBuf的步骤，只需要调用这两个工具类的静态方法即可。

ByteBuf和ByteBufAllocator之间是一种相辅相成的关系，ByteBufAllocator用来创建一个ByteBuf，而ByteBuf亦可以返回创建他的Allocator。ByteBuf和ByteBufAllocator之间是一种 `抽象工厂模式`，具体可以用一张图描述如下：



- 池化堆内存 ctx.alloc().heapBuffer()
- 池化直接内存 ctx.alloc().directBuffer()
- 非池化堆内存 Unpooled.buffer()
- 非池化直接内存 Unpooled.directBuffer()

1、ByteBuf如何工作的



- ByteBuf维护了readerIndex和writerIndex索引
- 当readerIndex > writerIndex时，则抛出IndexOutOfBoundsException
- ByteBuf容量 = writerIndex。
- ByteBuf可读容量 = writerIndex - readerIndex
- readXXX()和writeXXX()方法将会推进其对应的索引。自动推进
- getXXX()和setXXX()方法将对writerIndex和readerIndex无影响

2、ByteBuf的使用模式

我们以Unpooled类为例，查看Unpooled的源码可以发现，他为我们提供了许多创建ByteBuf的方法，但是最终都是以下几种，只是参数不一样而已：

```
// 在堆上分配一个ByteBuf，并指定初始容量和最大容量
public static ByteBuf buffer(int initialCapacity, int maxCapacity) {
    return ALLOC.heapBuffer(initialCapacity, maxCapacity);
}
// 在堆外分配一个ByteBuf，并指定初始容量和最大容量
public static ByteBuf directBuffer(int initialCapacity, int maxCapacity) {
    return ALLOC.directBuffer(initialCapacity, maxCapacity);
}
// 使用包装的方式，将一个byte[]包装成一个ByteBuf后返回
public static ByteBuf wrappedBuffer(byte[] array) {
    if (array.length == 0) {
        return EMPTY_BUFFER;
    }
    return new UnpooledHeapByteBuf(ALLOC, array, array.length);
}
// 返回一个组合ByteBuf，并指定组合的个数
public static CompositeByteBuf compositeBuffer(int maxNumComponents){
    return new CompositeByteBuf(ALLOC, false, maxNumComponents);
}
```

ByteBuf本质是: 一个由不同的索引分别控制读访问和写访问的字节数组。请记住这句话。ByteBuf共有三种模式: 堆缓冲区模式(Heap Buffer)、直接缓冲区模式(Direct Buffer)和复合缓冲区模式(Composite Buffer)

1. 堆缓冲区模式(Heap Buffer)

   堆缓冲区模式又称为：支撑数组(backing array)。将数据存放在JVM的堆空间，通过将数据存储在数组中实现堆缓冲的优点: 由于数据存储在Jvm堆中可以快速创建和快速释放，并且提供了数组直接快速访问的方法
   堆缓冲的缺点: 每次数据与I/O进行传输时，都需要将数据拷贝到直接缓冲区

```
public static void heapBuffer() {
    // 创建Java堆缓冲区
    ByteBuf heapBuf = Unpooled.buffer();
    //判断是否有一个支撑数组
    if (heapBuf.hasArray()) {
        //如果有，则获取该数组的引用
        byte[] array = heapBuf.array();
        //计算第一个字节的偏移量
        int offset = heapBuf.arrayOffset() + heapBuf.readerIndex();
        //获得可读字节数
        int length = heapBuf.readableBytes();
         System.out.println(Arrays.toString(array));
         System.out.println(offset);
         System.out.println(length);

    }
}
```

2. 直接缓冲区模式(Direct Buffer)

   Direct Buffer属于堆外分配的直接内存，不会占用堆的容量。适用于套接字传输过程，避免了数据从内部缓冲区拷贝到直接缓冲区的过程，性能较好

   - Direct Buffer的优点: 使用Socket传递数据时性能很好，避免了数据从Jvm堆内存拷贝到直接缓冲区的过程。提高了性能
   - Direct Buffer的缺点: 相对于堆缓冲区而言，Direct Buffer分配内存空间和释放更为昂贵
   - 对于涉及大量I/O的数据读写，建议使用Direct Buffer。而对于用于后端的业务消息编解码模块建议使用Heap Buffer

```
public static void main(String args[]) {
        ByteBuf directBuf = Unpooled.directBuffer(100);
        directBuf.writeBytes("direct buffer".getBytes());
         //检查 ByteBuf 是否由数组支撑。如果不是，则这是一个直接缓冲区
        if (!directBuf.hasArray()) {
            //获取可读字节数
            int length = directBuf.readableBytes();
            //分配一个新的数组来保存具有该长度的字节数据
            byte[] array = new byte[length];
            //将字节复制到该数组
            directBuf.getBytes(directBuf.readerIndex(), array);
            System.out.println(Arrays.toString(array));
            System.out.println(length);
        }
    }
```

3. 复合缓冲区模式(Composite Buffer)

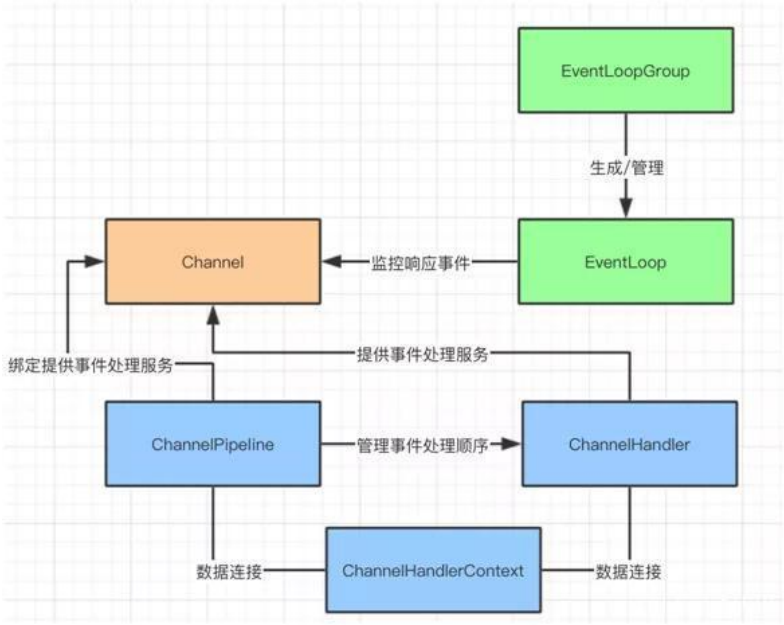
   Composite Buffer是Netty特有的缓冲区。本质上类似于提供一个或多个ByteBuf的组合视图，可以根据需要添加和删除不同类型的ByteBuf。

   - 想要理解Composite Buffer，请记住：它是一个组合视图。它提供一种访问方式让使用者自由的组合多个ByteBuf，避免了拷贝和分配新的缓冲区。
   - Composite Buffer不支持访问其支撑数组。因此如果要访问，需要先将内容拷贝到堆内存中，再进行访问
   - 下图是将两个ByteBuf：头部+Body组合在一起，没有进行任何复制过程。仅仅创建了一个视图

图 5-2　持有一个头部和主体的 CompositeByteBuf

```
public static void byteBufComposite() {
    // 复合缓冲区，只是提供一个视图
    CompositeByteBuf messageBuf = Unpooled.compositeBuffer();
    ByteBuf headerBuf = Unpooled.buffer(); // can be backing or direct
    ByteBuf bodyBuf = Unpooled.directBuffer();   // can be backing or direct
    messageBuf.addComponents(headerBuf, bodyBuf);
    messageBuf.removeComponent(0); // remove the header
    for (ByteBuf buf : messageBuf) {
        System.out.println(buf.toString());
    }
}
```

组件之间的关系



案例：Netty构建Http服务器

```
package com.yuandengta.http;

import io.netty.bootstrap.ServerBootstrap;
import io.netty.channel.ChannelFuture;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.nio.NioServerSocketChannel;

/**
 * @Author:Hardy
 * @QQ:2937270766
 * @官网：http://www.yuandengta.com
 */
public class HttpServer {
    public static void main(String[] args) throws InterruptedException {
```

```
        EventLoopGroup bossGroup = new NioEventLoopGroup();
        EventLoopGroup workerGroup = new NioEventLoopGroup();

        try {
            ServerBootstrap serverBootstrap = new ServerBootstrap();
            serverBootstrap.group(bossGroup,workerGroup)
                    .channel(NioServerSocketChannel.class)
                    .childHandler(new ServerInitializer());

            ChannelFuture ch = serverBootstrap.bind(8888).sync();
            ch.channel().closeFuture().sync();
        }finally {
            bossGroup.shutdownGracefully();
            workerGroup.shutdownGracefully();
        }
    }
}
```

```
package com.yuandengta.http;

import io.netty.buffer.ByteBuf;
import io.netty.buffer.Unpooled;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.SimpleChannelInboundHandler;
import io.netty.handler.codec.http.DefaultFullHttpResponse;
import io.netty.handler.codec.http.HttpHeaderNames;
import io.netty.handler.codec.http.HttpObject;
import io.netty.handler.codec.http.HttpRequest;
import io.netty.handler.codec.http.HttpResponseStatus;
import io.netty.handler.codec.http.HttpVersion;
import io.netty.util.CharsetUtil;

/**
 * @Author:Hardy
 * @QQ:2937270766
 * @官网：http://www.yuandengta.com
 * HttpObject 客户端和服务器通讯的数据被封装成HttpObject
 */
public class HttpServerHandler extends SimpleChannelInboundHandler<HttpObject> {
    @Override
    protected void channelRead0(ChannelHandlerContext ctx, HttpObject msg)
throws Exception {
        //判断msg是不是httpRequest请求
        if(msg instanceof HttpRequest){
            System.out.println("msg 类型 = "+msg.getClass());
            System.out.println("客户端地址 = "+ctx.channel().remoteAddress());
            //回复信息给浏览器[http协议]
            ByteBuf content = Unpooled.copiedBuffer("hello,我是服务器",
CharsetUtil.UTF_16);
            //构造一个http的响应，即httpresponse
            DefaultFullHttpResponse response = new DefaultFullHttpResponse(
                    HttpVersion.HTTP_1_1, HttpResponseStatus.OK, content);
            response.headers().set(HttpHeaderNames.CONTENT_TYPE,"text/plain");

 response.headers().set(HttpHeaderNames.CONTENT_LENGTH,content.readableBytes());

            //将构建好response返回
```

```
            ctx.writeAndFlush(response);

        }
    }
}
```

七、Netty心跳检测

在Netty4中，使用IdleStateHandler实现心跳检测及空闲状态检测。我们知道使用netty的时候，大多数的东西都与Handler有关，我们的业务逻辑基本都是在Handler中实现的。Netty中自带了一个IdleStateHandler 可以用来实现心跳检测。

心跳检测的逻辑

本文中我们将要实现的心跳检测逻辑是这样的：服务端启动后，等待客户端连接，客户端连接之后，向服务端发送消息。如果客户端在“干活”那么服务端必定会收到数据，如果客户端“闲下来了”那么服务端就接收不到这个客户端的消息，既然客户端闲下来了，不干事，那么何必浪费连接资源呢？所以服务端检测到一定时间内客户端不活跃的时候，将客户端连接关闭。

八、Netty粘包和拆包处理

1、什么是粘包和半包

我们知道， Netty 发送和读取数据的单位，可以形象的使用 ByteBuf 来充当。

每一次发送，就是向Channel 写入一个 ByteBuf ； 每一次读取，就是从 Channel 读到一个 ByteBuf 。

我们的理想是：发送端每发送一个buffer，接收端就能接收到一个一模一样的buffer。

然而，理想很丰满，现实很骨感。

在实际的通讯过程中，并没有大家预料的那么完美。一种意料之外的情况，如期而至。这就是粘包和半包。

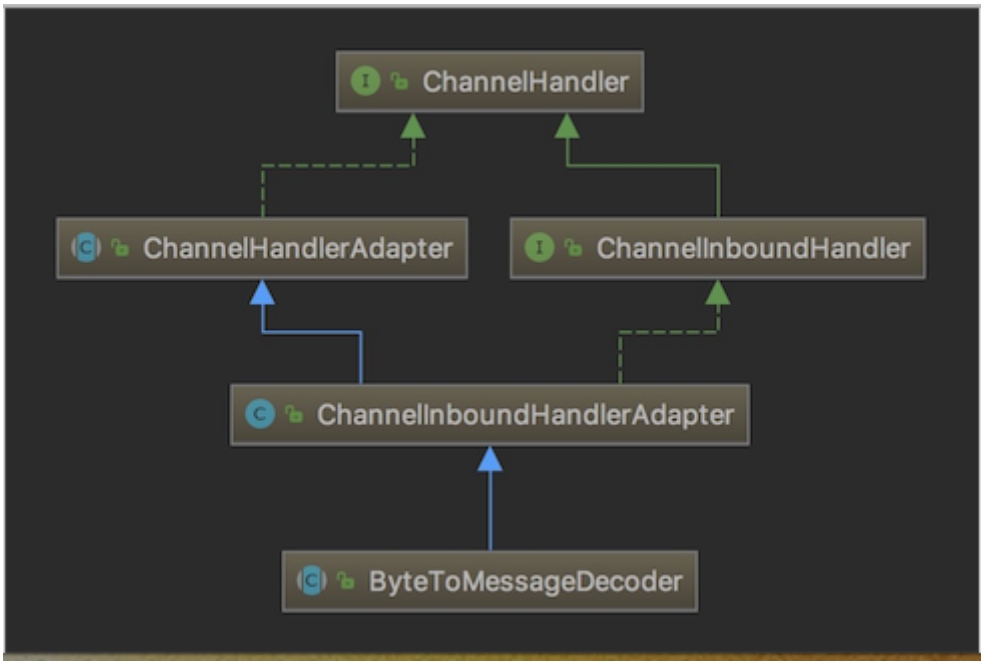
1、粘包和半包：指的都不是一次是正常的 ByteBuf 缓存区接收。

2、粘包：就是接收端读取的时候，多个发送过来的 ByteBuf “粘”在了一起。换句话说，接收端读取一次的 ByteBuf ， 读到了多个发送端的 ByteBuf ， 是为粘包。

3、半包：就是接收端将一个发送端的ByteBuf “拆”开了，形成一个破碎的包，我们定义这种 ByteBuf 为半包。换句话说，接收端读取一次的 ByteBuf ， 读到了发送端的一个 ByteBuf的一部分，是为半包

处理粘包拆包，其实思路都是一致的，就是“分分合合”，粘包由于数据过多，那就按照固定策略分割下交给程序来处理；拆包由于一次传输数据较少，那就等待数据传输长度够了再交给程序来处理。

2、常用解码器:



在Netty中的编码器其实就是一个handler，Netty中的编解码器太多了，下面就常用ByteToMessageDecoder介绍它的体系：

解码器的模板基类：ByteToMessageDecoder

ByteToMessageDecoder继承了ChannelInboundHandlerAdapter，说明它是处理入栈方向数据的编码器，因此它也是一个不折不扣的Handler，再回想,其实In开头的handler都是基于事件驱动的,被动的处理器,当客户端发生某种事件时,它对应有不同的动作回调,而且它的特色就是 fireXXX往下传递事件, 待会我们就能看到,netty用它把处理好的数据往下传递

- FixedLengthFrameDecoder（固定长度的拆包器）

  1、消息长度固定，累积读取到长度总和为定长 LEN 的报文后，就认为读取到了一个完整的消息，再将计数器置位，重新读取下一个数据报。例如可以让每个报文的大小为固定长度 1024 字节，如果消息长度不够，则使用空位填补空缺，这样读取到了之后，只需要 trim 去掉空格即可。

  2、FixedLengthFrameDecoder 固定长度解码器，它能够按照指定的长度对消息进行自动解码，开发者不需要考虑 TCP 的粘包与拆包问题，非常实用。无论一次接收到多少数据报，它都会按照构造器中设置的固定长度进行解码，如果是半包消息，FixedLengthFrameDecoder 会缓存半包消息并等待下个包到达之后进行拼包合并，直到读取一个完整的消息包

- LineBasedFrameDecoder（行拆包器）

  LineBasedFrameDecoder的工作原理是它一次遍历ByteBuf中的可读字节，判断看是否有“\n”或者“\r\n”， 如果有，就以此位置为结束位置，从可读索引到结束位置区间的字节就组成了一行。它是以换行符为结束标志的解码器，支持携带结束符或者不携带结束符两种解码方式，同时支持配置单行的最大长度。如果连续读取到最大长度后仍然没有出现换行符，就会抛出异常，同时忽略掉之前读到的异常码流。

- DelimiterBasedFrameDecoder（分隔符拆包器）

  DelimterBaseFrameDecoder是分隔符解码器，用户可以指定消息结束的分隔符，它可以自动完成以分隔符作为码流结束标识的消息解码。回车换行解码器实际上是一种特殊的DelimiterBaseFrameDecoder解码器。

- LengthFieldBasedFrameDecoder（基于数据包长度的拆包器）

大多数的协议（私有或者公有），协议头中会携带长度字段，用于标识消息体或者整包消息的长度，例如SMPP、HTTP协议等。由于基于长度解码需求的通用性，以及为了降低用户的协议开发难度，Netty提供了[LengthFieldBasedFrameDecoder，自动屏蔽TCP底层的拆包和粘包问题，只需要传入正确的参数，即可轻松解决“读半包”问题。

基于长度域,指的是在传输的协议中有一个 length字段,这个十六进制的字段记录的可能是整个协议的长度,也可能是消息体的长度, 我们根据具体情况使用不同的构造函数

如何使用呢? 最常用它下面的这个构造函数

```
public LengthFieldBasedFrameDecoder(
    int maxFrameLength,
    int lengthFieldOffset,
    int lengthFieldLength,
    int lengthAdjustment,
    int initialBytesToStrip) {
this(
        maxFrameLength,
        lengthFieldOffset, lengthFieldLength, lengthAdjustment,
        initialBytesToStrip, true);
}
```

参数解释：

- maxFrameLength：每次解析所能接受的最大帧的长度
- lengthFieldOffset：长度域的偏移量（就是在现有的这段字节数据中找个开始解码的位置，大多数设为0，意思是从0位置开始解码）
- lengthFieldLength：字段域的长度，根据lengthFieldOffset的初始值往后数lengthFieldLength个字节，这段范围解析出来的数值可能是长度域的大小，也可能是整个协议的大小
- lengthdjustment：矫正长度
- initialBytesToStrip：需要取出的长度

```
 * BEFORE DECODE (14 bytes)         AFTER DECODE (12 bytes)
 * +--------+----------------+      +--------+----------------+
 * | Length | Actual Content |----->| Length | Actual Content |
 * | 0x000C | "HELLO, WORLD" |      | 0x000C | "HELLO, WORLD" |
 * +--------+----------------+      +--------+----------------+
 这是最简单的情况，假定 Length的长度就是后面的 真正需要解码的内容

 现在的字节全部解码后是这样的   12HELLO, WORLD
 我们要做的就是区分出   12和HELLO, WORLD

 * lengthFieldOffset   = 0
 * lengthFieldLength   = 2 // todo 每两个字节 表示一个数据包
 * lengthAdjustment    = 0
 * initialBytesToStrip = 0

 意思就是:
  字节数组[lengthFieldOffset,lengthFieldLength]之间的内容转换成十进制,就是后面的字段域的长度
    00 0C ==> 12
  这个12 意思就是 长度域的长度，说白了 就是我们想要的 HELLO, WORLD 的长度

  这样一算,就分开了
```

```
 * BEFORE DECODE (14 bytes)         AFTER DECODE (12 bytes)
* +--------+----------------+      +----------------+
* | Length | Actual Content |----->| Actual Content |
* | 0x000C | "HELLO, WORLD" |      | "HELLO, WORLD" |
* +--------+----------------+      +----------------+
```

情况2:

```
* lengthFieldOffset   = 0
* lengthFieldLength   = 2 // todo 每两个字节 表示一个数据包
* lengthAdjustment    = 0
* initialBytesToStrip = 2
```

意思就是

字节数组[lengthFieldOffset,lengthFieldLength]之间的内容转换成十进制,就是后面的字段域的长度是

00 0C ==> 12

这个12 意思就是 长度域的长度, 说白了 就是我们想要的 HELLO, WORLD 的长度

然后, 从0开始 忽略 initialBytesToStrip, 就去除了 length ,只留下 HELLO, WORLD

有时, 在某些其他协议中, length field 可能代表是整个消息的长度, 包括消息头

在这种情况下,我们就得指定一个 非零的 lengthAdjustment 去调整

```
 * BEFORE DECODE (14 bytes)         AFTER DECODE (14 bytes)
* +--------+----------------+      +--------+----------------+
* | Length | Actual Content |----->| Length | Actual Content |
* | 0x000E | "HELLO, WORLD" |      | 0x000E | "HELLO, WORLD" |
* +--------+----------------+      +--------+----------------+
```

```
* lengthFieldOffset   = 0
* lengthFieldLength   = 2 // todo 每两个字节 表示一个数据包
* lengthAdjustment    = -2
* initialBytesToStrip = 0
```

意思就是

字节数组[lengthFieldOffset,lengthFieldLength]之间的内容转换成十进制,表示整个协议的长度

00 0C ==> 14 意味,协议全长 14

现在还是不能区分开 Length 和 Actual Content

公式: 数据包的长度 = 长度域 + lengthFieldOffset + lengthFieldLength +lengthAdjustment

通过他可以算出 lengthAdjustment = -2

```
* BEFORE DECODE (17 bytes)                       AFTER DECODE (17 bytes)
* +----------+----------+----------------+      +----------+----------+----------------+
* | Header 1 |  Length  | Actual Content |----->| Header 1 |  Length  | Actual Content |
* |  0xCAFE  | 0x00000C | "HELLO, WORLD" |      |  0xCAFE  | 0x00000C | "HELLO, WORLD" |
* +----------+----------+----------------+      +----------+----------+----------------+
```

这个例子和第一个例子很像,但是多了头

我们想拿到后面消息长度的信息,就偏移过header

```
* lengthFieldOffset   = 2
* lengthFieldLength   = 3 // todo 每两个字节 表示一个数据包
* lengthAdjustment    = 0
* initialBytesToStrip = 0
```

字节数组[lengthFieldOffset,lengthFieldLength]之间的内容转换成十进制，表示长度域的长度

在这里 整好跳过了 header 1, 0x00 00 0C 是三个字节
也就是 字节数组[lengthFieldOffset,lengthFieldLength]=>[0,3]
0x00 00 0C == 12 表示长度域是 12

现在也成功区分开了 Header 1 和 Length 和 Actual Content
分别是 2 3 12

```
BEFORE DECODE (17 bytes)                       AFTER DECODE (17 bytes)
* +----------+----------+----------------+      +----------+----------+----------------+
* |  Length  | Header 1 | Actual Content |----->|  Length  | Header 1 | Actual Content |
* | 0x00000C |  0xCAFE  | "HELLO, WORLD" |      | 0x00000C |  0xCAFE  | "HELLO, WORLD" |
* +----------+----------+----------------+      +----------+----------+----------------+

* lengthFieldOffset   = 0
* lengthFieldLength   = 3 // todo 每两个字节 表示一个数据包
* lengthAdjustment    = 2
* initialBytesToStrip = 0
```

字节数组[lengthFieldOffset,lengthFieldLength]之间的内容转换成十进制，表示长度域的长度

也就是 字节数组[lengthFieldOffset,lengthFieldLength]=>[0,3]
0x00 00 0C 是三个字节
0x00 00 0C == 12 表示长度域是 12 == 长度域的长度 就是 HELLO, WORLD的长度
但是上面的图多了一个 两个字节长度的 Header 1
下一步进行调整

公式: 数据包的长度 = 长度域 + lengthFieldOffset + lengthFieldLength +lengthAdjustment

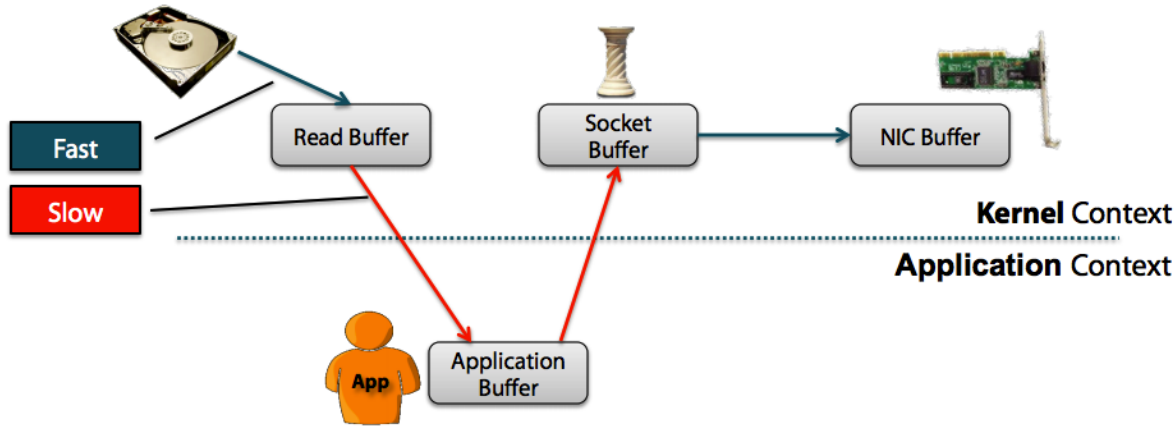
lengthAdjustment= 17-12-0-3=2

```
* BEFORE DECODE (16 bytes)                       AFTER DECODE (13 bytes)
* +------+--------+------+----------------+      +------+----------------+
* | HDR1 | Length | HDR2 | Actual Content |----->| HDR2 | Actual Content |
* | 0xCA | 0x000C | 0xFE | "HELLO, WORLD" |      | 0xFE | "HELLO, WORLD" |
* +------+--------+------+----------------+      +------+----------------+


* lengthFieldOffset   = 1
* lengthFieldLength   = 2 // todo 每两个字节 表示一个数据包
* lengthAdjustment    = 1
* initialBytesToStrip = 3

lengthFieldOffset =1 偏移1字节 跨过 HDR1

lengthFieldLength =2 从[1,2] ==> 0x000C =12 表示长度域的值

看拆包后的结果,后面明显还多了个 HDR2 ,进行调整
公式:  数据包值 = 长度域  + lengthFieldOffset+ lengthFieldLength + lengthAdjustment
算出 lengthAdjustment = 16 - 12 - 1 - 2 = 1

结果值只有 HDR2 和  Actual Content , 说明,前面通过 initialBytesToStrip 进行忽略
initialBytesToStrip =3
```

```
 * BEFORE DECODE (16 bytes)                       AFTER DECODE (13 bytes)
* +------+--------+------+----------------+      +------+----------------+
* | HDR1 | Length | HDR2 | Actual Content |----->| HDR2 | Actual Content |
* | 0xCA | 0x0010 | 0xFE | "HELLO, WORLD" |      | 0xFE | "HELLO, WORLD" |
* +------+--------+------+----------------+      +------+----------------+


* lengthFieldOffset   = 1
* lengthFieldLength   = 2 // todo 每两个字节 表示一个数据包
* lengthAdjustment    = -3
* initialBytesToStrip = 3

同样
看结果,保留 HDR2 和 Actual Content

lengthFieldOffset   = 1 表示跳过开头的 HDR1
[1,2] ==> 00 10 , 算出的 长度域的值==10 很显然这不对

10 < 13

我们要想拆出后面的数据包就得在现有的基础上往左移动三个字节 -3个调整量
```

九、Netty零拷贝

传统拷贝



JVM发出read()调用：

第一次拷贝：hardware ------> kernel buffer（硬盘数据读取到内核空间缓冲区）

第二次拷贝：kernel -------> user buffer（从内核缓冲区复制到用户缓冲区）

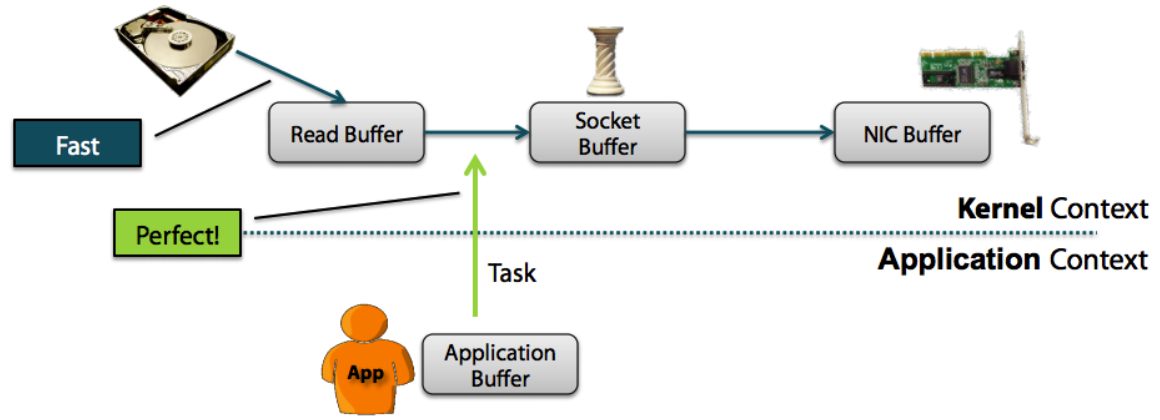
JVM处理代码逻辑并发送write()系统调用

第三次拷贝：user buffer -----> kernel buffer（从用户缓冲区复制数据到内核缓冲区）

第四次拷贝：kernel buffer ----> socket Buffer（内核空间缓冲区中的数据写到hardware）

总的来说，传统的I/O操作进行了4次用户空间与内核空间的上下文切换，以及4次数据拷贝。显然在这个用例中，从内核空间到用户空间内存的复制是完全不必要的，因为除了将数据转储到不同的buffer之外，我们没有做任何其他的事情。所以，我们能不能直接从hardware读取数据到kernel buffer后，再从kernel buffer写到目标地点不就好了。为了解决这种不必要的数据复制，操作系统出现了零拷贝的概念。注意，不同的操作系统对零拷贝的实现各不相同

零拷贝



这种方式需要四次数据拷贝和四次上下文切换：
\1. 数据从磁盘读取到内核的read buffer
\2. 数据从内核缓冲区拷贝到用户缓冲区
\3. 数据从用户缓冲区拷贝到内核的socket buffer
\4. 数据从内核的socket buffer拷贝到网卡接口的缓冲区

明显上面的第二步和第三步是没有必要的，通过java的FileChannel.transferTo方法，可以避免上面两次多余的拷贝（当然这需要底层操作系统支持）
\1. 调用transferTo,数据从文件由DMA引擎拷贝到内核read buffer
\2. 接着DMA从内核read buffer将数据拷贝到网卡接口buffer

上面的两次操作都不需要CPU参与，所以就达到了零拷贝。

Netty的零拷贝体现在三个方面：

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那样方便的对组合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。