# Netty

## Tip

### Mmap

mmap是一种内存映射文件的方法，即将一个文件或者其它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映关系。实现这样的映射关系后，进程就可以采用指针的方式读写操作这一段内存，而系统会自动回写脏页面到对应的文件磁盘上，即完成了对文件的操作而不必再调用read,write等系统调用函数。相反，内核空间对这段区域的修改也直接反映用户空间，从而可以实现不同进程间的文件共享.

一旦做好了映射,如果内存改了,或者是磁盘改了,他俩直接就会定时同步也就是刷脏页.



1、进程发起读文件请求。

2、内核通过查找进程文件符表，定位到内核已打开文件集上的文件信息，从而找到此文件的inode。

3、inode在address\_space上查找要请求的文件页是否已经缓存在页缓存中。如果存在，则直接返回这片文件页的内容。

4、如果不存在，则通过inode定位到文件磁盘地址，将数据从磁盘复制到页缓存。之后再次发起读页面过程，进而将页缓存中的数据发给用户进程。

总结来说，常规文件操作为了提高读写效率和保护磁盘，使用了页缓存机制。这样造成读文件时需要先将文件页从磁盘拷贝到页缓存中，由于页缓存处在内核空间，不能被用户进程直接寻址，所以还需要将页缓存中数据页再次拷贝到内存对应的用户空间中。这样，通过了两次数据拷贝过程，才能完成进程对文件内容的获取任务。写操作也是一样，待写入的buffer在内核空间不能直接访问，必须要先拷贝至内核空间对应的主存，再写回磁盘中（延迟写回），也是需要两次数据拷贝。

而使用mmap操作文件中，创建新的虚拟内存区域和建立文件磁盘地址和虚拟内存区域映射这两步，没有任何文件拷贝操作。而之后访问数据时发现内存中并无数据而发起的缺页异常过程，可以通过已经建立好的映射关系，只使用一次数据拷贝，就从磁盘中将数据传入内存的用户空间中，供进程使用。

总而言之，常规文件操作需要从磁盘到页缓存再到用户主存的两次数据拷贝。而mmap操控文件，只需要从磁盘到用户主存的一次数据拷贝过程。说白了，mmap的关键点是实现了用户空间和内核空间的数据直接交互而省去了空间不同数据不通的繁琐过程。因此mmap效率更高。

大家关于“mmap()”更快的认识来自于 read() 是需要内存拷贝的；

当今硬件技术的发展，使得内存拷贝消耗的时间已经极大降低了；

但“mmap()”的开销在于一次  pagefault，这个开销相比而言已经更高了，而且 pagefault 的处理任务现在比以前还更多了；

而且，mmap之后，再有读操作不会经过系统调用，在 LRU 比较最近使用的页的时候不占优势；

于是，普通读情况下（排除反复读之类的文艺与2B读操作），read() 通常会比 mmap() 来得更快。

### stream和buffer

tcp是个流的协议,所谓流,就是没有界限的一串数据.就像河水,连城一片的,没有分界线

tcp底层并不了解上层业务数据的具体含义,他会根据tcp缓冲区的实际情况进行包的划分.

所以业务上认为,一个完整的包可能被tcp拆成多个包进行发送,也可能把多个小包组成一个大的数据包进行发送.

而buffer是缓冲的意思,但是虽然有了流我也不能有一个字节就马上把这个字节就推下去吧,我肯定得搞个缓冲等这个缓冲满了然后我再往下推流,所以buffer无非就是个字节数组,一个容器而已.

### tcp的粘包和拆包

nagle算法就是对那种包比较大的数据就直接发送,对于很多的那种包很小的小包就给合并在一起,可见如果有很多小包的时候(这种小包大部分是那种交互数据也就是信令)nagle算法可以有效减少网络中报文段个数.但是对于有些要求实时性比较高的通信程序而言这就不行了,所以要禁用nagle算法.

tcp收发两端都要有socket,因为发送端使用了优化算法(Nagle算法),将多次间隔较少且数据量小的数据合成一个大的数据块,然后进行封包(也就是粘包),这样几个数据包就变成了一个大包一次性发过去了,但是接收端蒙了,你这几个包合在一起我怎么区分有几个请求啊,必须有科学的拆包机制

也就是面向流的协议是无消息保护边界的.

udp不存在粘包问题,是由于udp发送的时候,没有经过negal算法优化,不会将多个小包合并一次发送出去.接收端应用程序一次recv只能从socket缓冲区中读到一个数据包,也就是说发送端send了几次,接收端必须recv几次(主要还是没有用nagal算法啊)

也就是面向消息的通信是有消息保护边界的

假设客户端发送了两个数据包D1 D2给服务端,由于服务端一次读取到字节数是不确定的,(下面也说到了,他可能一次性就读了1m啊)所以可能存在很多情况,

1,服务端分两次读取到了两个独立的数据包,分别的D1和D2,这样就没有粘包和拆包.

2,服务端一次接收到了两个数据包,也就是客户端使用了nagle算法,把D1和D2给粘一起了,这就是封包粘包咯

3,还有就是接收了两次,第一个接收到了D1和D2的一部分,第二次接收到了D2的剩余的部分,那么这样D2就是被拆开了,这样就发送了拆包(**拆包不是把粘起来的包拆开,而是发送的时候一个包的数据分了多次发送**)的操作.

还有就是如果tcp的接收滑窗非常小,而数据包D1和D2都比较大,那么就会发生第5种情况,每个包都需要分多次才能全部接收,这样就是发生了多次的拆包.

对啊,tcp的发送和接收具体能操作多大的数据不是由这个滑动窗口决定的么,你在服务端默认用1m去接收,可能是因为你觉得1m已经能hold住最大的窗口了????

粘包拆包发送的原因

1, socket缓冲区与滑动窗口(这个会引起拆包)

2,MSS/MTU限制(这个会引起拆包)

3,Nagle算法(这个会引起粘包)

现在来看一下SO\_RCVBUF和滑动窗口是如何造成粘包、拆包的？

粘包：假设发送方的每256 bytes表示一个完整的报文，接收方由于数据处理不及时，这256个字节的数据都会被缓存到SO\_RCVBUF中。如果接收方的SO\_RCVBUF中缓存了多个报文，那么对于接收方而言，这就是粘包。(也就是recvbuf里面缓存了多个包,**read的时候是会一次把整个buf都读出来的**,那么这里read就是一下读了多个包,这也就是造成了粘包)

拆包：考虑另外一种情况，假设接收方的window size只剩了128，意味着发送方最多还可以发送128字节，而由于发送方的数据大小是256字节，因此只能发送前128字节，等到接收方ack后，才能发送剩余字节。这就造成了拆包。(拆包就是发的包太大,但是可用的buf不够了,这样一个包的数据就分散在了两个read里面了)

MSS是MSS是Maximum Segement Size的缩写，表示TCP报文中data部分的最大长度，是TCP协议在OSI五层网络模型中传输层(transport layer)对一次可以发送的最大数据的限制。

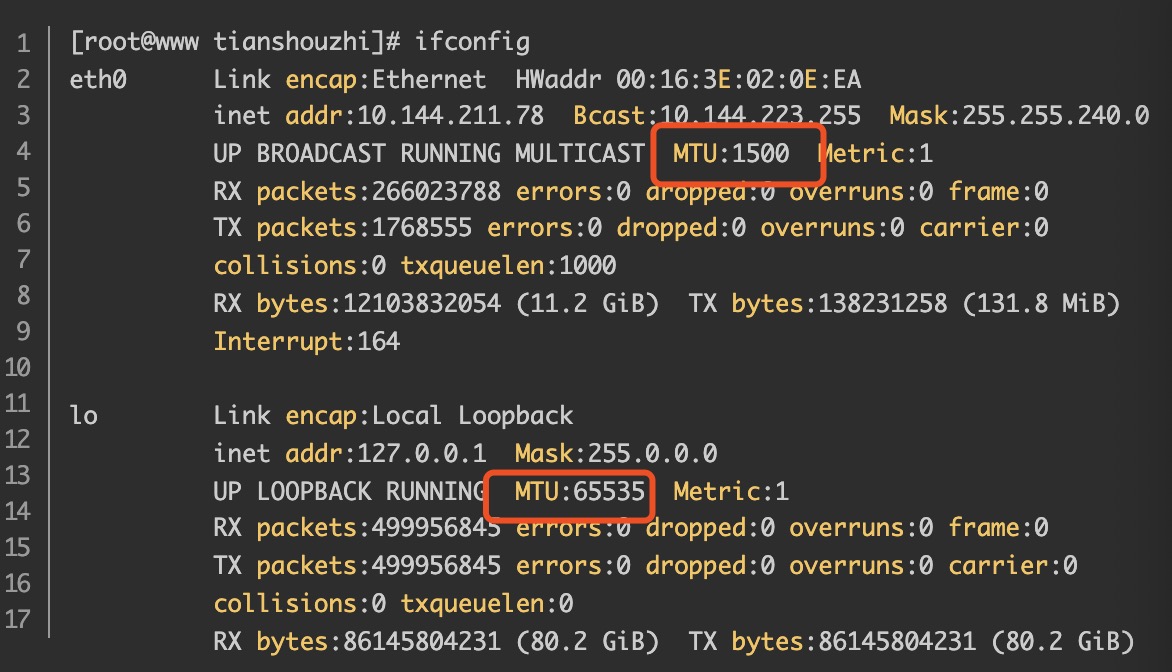
MTU最大传输单元是Maxitum Transmission Unit的简写，是OSI五层网络模型中链路层(datalink layer)对一次可以发送的最大数据的限制。

当需要传输的数据大于MSS或者MTU时，数据会被拆分成多个包进行传输。由于MSS是根据MTU计算出来的，因此当发送的数据满足MSS时，必然满足MTU。归根结底：限制一次可发送数据大小的是MTU，MSS只是TCP协议在MTU基础限制的传输层一次可传输的数据的大小。

所以mtu-tcp头 -劈头就是mss咯



可以看到,SO\_SNDBUF里面的东西,最终会被包上一个个header,tcp头啊,ip头啊啥的,这就把网络和系统的关系对上了.所以数据超了过mss,那么一定就会有拆包的.



ifconfig可以查看网卡的MTU多大,一般MTU都是网卡决定的,一般都是1500byte,但是本地的回环网卡是65535,因为他不需要出去所有就很大咯.

### socket缓冲区与滑动窗口

所以说下面介绍那几种io模式的时候,什么要把数据拷贝到内核,还是内核和程序的进程的操作,看来操作的就是这个SNDBUF和RECVBUF咯

先明确一个概念,每个tcp socket在内核中都有一个发送缓冲区(SEDNBUFF)和一个接收缓冲区(RCVBUFF),tcp的全双工工作模式以及tcp的滑动窗口更是依赖于这两个独立的buffer以及此buffer的填充状态(也就是滑动窗口其实就是这两个buffer,想想也是啊,有消息到了就写buff,程序会从buff里拿数据,这样buff就会释放,然后窗口就大了啊,不就是这个意思么啊)

SO\_SNDBUF和SO\_RCVBUF在windows操作系统中默认都是8k (而且正是由于这个buff的存在才能进行粘包操作啊)

SO\_SNDBUF 进程发送数据的时候(假设调用了一个send方法),最简单的情况,将数据拷贝到socket的内核发送缓冲区中,然后send便会在上层返回.换句话说,send返回的时候,数据不一定会发送到对端去,send仅仅是把应用层buffer的数据拷贝到socket的 SO\_SNDBUF里面去了.

SO\_RCVBUF把接收到的数据存入到内核里面的buf,应用程序一直没有调用read进行读取的话,此数据会一直缓存在相应的socket的接收缓冲区内. 其实read函数的工作就是把内核缓冲区中的数据拷贝到应用层用户的buffer里面.

我们也知道tcp是有滑动窗口的,接收方会以ack的方式把自己最新的窗口大小发送给发送方,其实发送的窗口大小就是SO\_RCVBUF的可用空间的大小.

### CRC

（循环冗余校验）这个是数据链路层的东西,就是在数据帧的最后给附上一个数,这个数就是用前面的数据信息算出来的一个值,可以用来检测数据包在传输的过程中有没有被损坏.

## JavaIO的演进

### io基础

在nio完善之前java的io主要面临的问题如下.

1,没有数据缓冲区,io性能存在问题.

2,没有c和c++中channel的概念,只有输入输出流

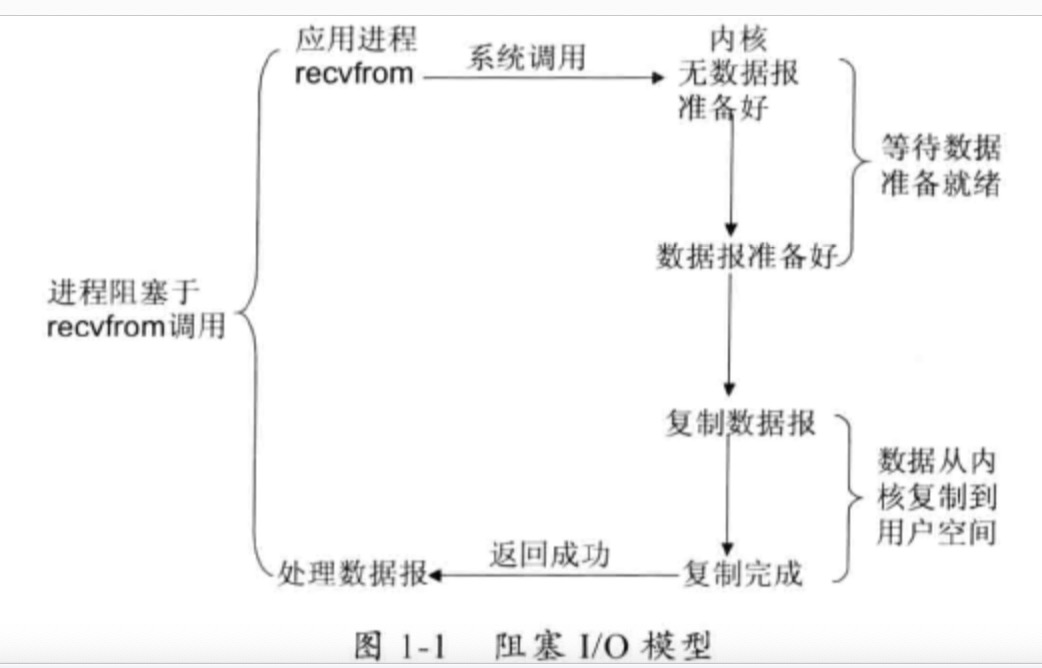
3,同步阻塞式BIO通常会导致通信线程被长时间阻塞.

linux内核将所有的外部设备都看作一个文件来操作,对于一个文件(那么这里也就是说操作外部设备会有一个fd咯,那么这里在网络的上下文下这个外部设备就是网卡咯)的读写操作会调用内核提供的系统命令,返回一个file descriptor fd,其实fd就是一个数字,他指向内核的一个结构体(文件路径,数据区等一些属性)

因为我们要操作网卡,网卡会被抽象成文件,那么我们就是要操作一个文件,那么就内核就会对这个文件搞出一个fd出来. 而fd就是个数字,类似于windows的handler或者指针啥的,指向了内核中的一个地方.

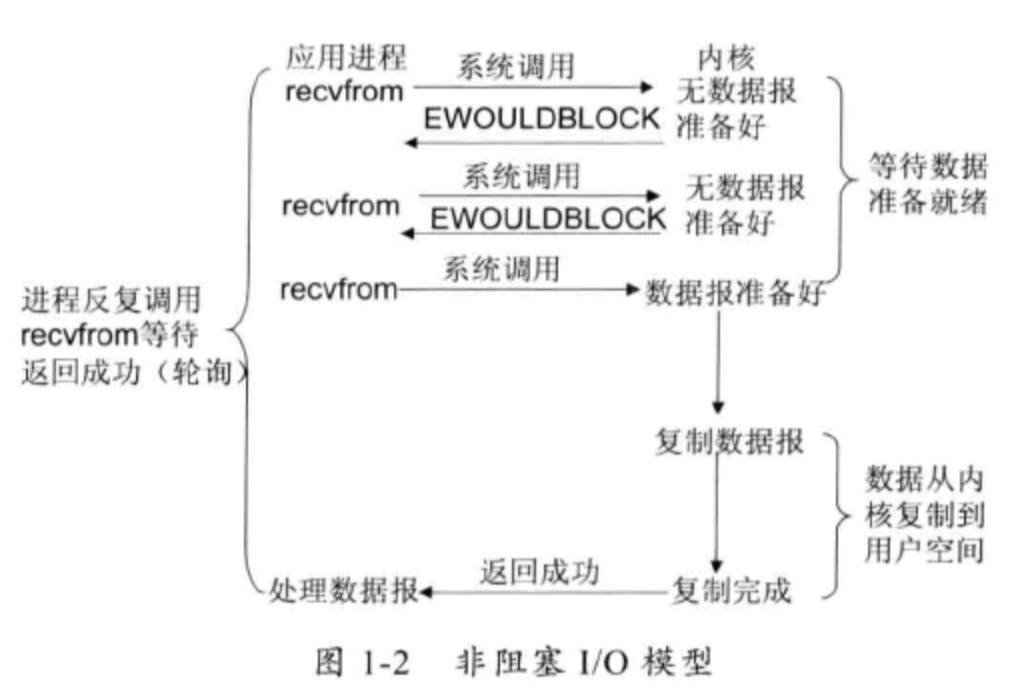
5种io模型

1,阻塞io,最常用的io模型就是阻塞io模型,默认情况下,对所有的文件(linux所有东西都是文件啊,也就是网卡什么的这里都包括了)操作都是阻塞的.以套接字为例来讲解的话,在进程空间中调用recvfrom,其系统调用直到数据包到达且被复制到应用进程的缓冲区中或者发生错误时才会返回,在此期间会一直在等待,进程在从调用recvfrom开始到他返回的整段时间内都是被阻塞的,因此被称为阻塞io模型. 如下图



这个图画了数据准备就绪的过程,而且还被后续的数据从内核空间复制到用户空间的过程也画了出来.

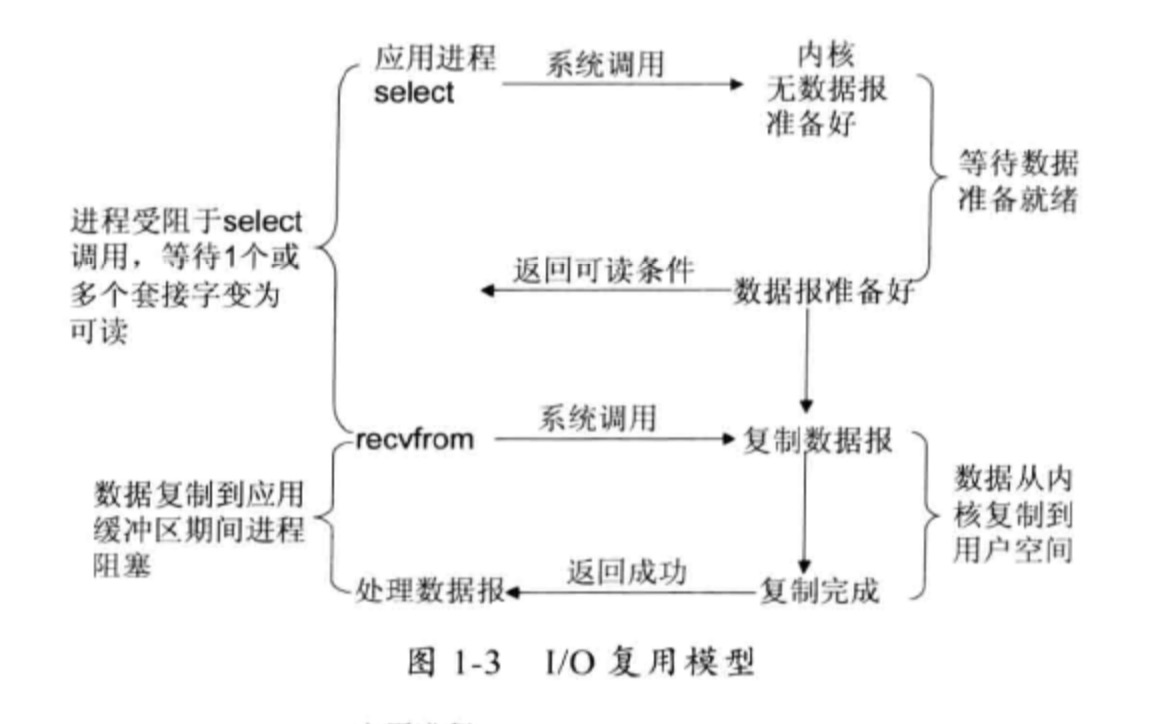
2,非阻塞io模型,recvfrom从应用层到内核的时候,如果该缓冲区没有数据的话,就直接返回一个EWOULDBLOCK的错误,一般都对非阻塞io模型进行轮询检查这个状态,看内核是不是有数据到来. 如下图咯,



3,io复用模型,linux提供select/poll,进程通过一个或多个fd传递给slect或poll进行调用,

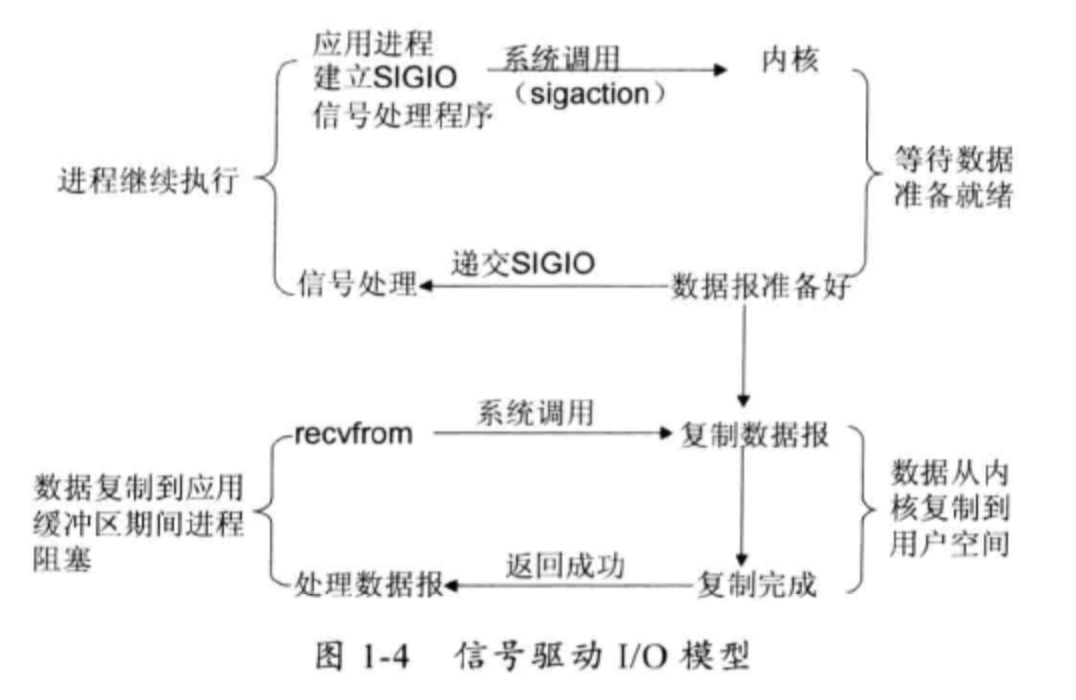
这样select/poll可以帮我们检测多个fd是否处于就绪状态.select/poll是顺序扫描fd是否就绪,而且支持的fd数量有限,所以他的使用有一些限制.

linux还提供了一个epoll系统调用,epoll使用基于事件驱动方式代替顺序扫描,因此性能更高,当有fd就绪时,立即回调rollback(看起来就是跟去cpu的中断程序上注册回调是一个意思咯,所谓的事件驱动咯)



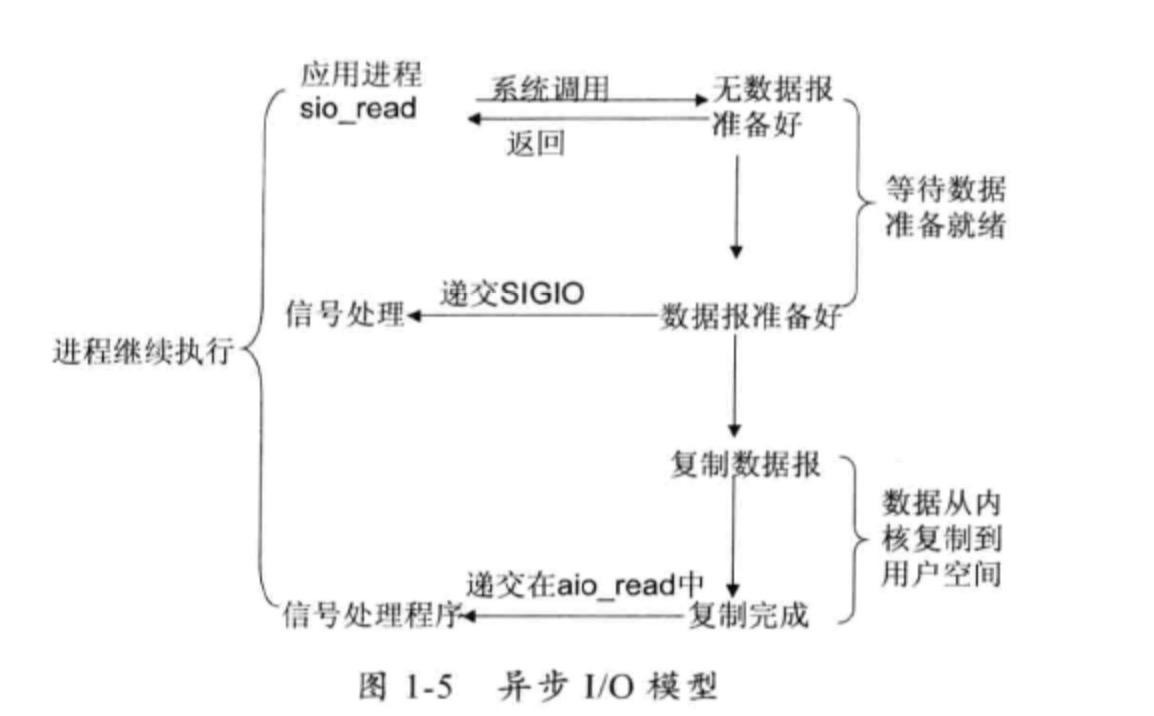
可以看到,在recvfrom前面有了一个select的操作,

4,信号驱动io模型,首先开启套接字接口驱动io功能,并进程会通过系统sigaction执行一个信号处理函数(此系统调用立即返回,进程继续工作,他是非阻塞的) 当数据准备就绪时,就为该进程生成一个SIGIO信号,通过信号回调通知应用程序调用recvfrom来读取数据,信号驱动只会告诉你数据准备好了,后面的recvfrom和复制数据的流程还是要应用程序自己来做(好像就是把监视套接字的工作给弄到外面了)



反正这个也是有内核空间数据复制到进程空间的过程.

5,异步io,告知内核启动某个操作,并让内核在整个系统完成后(包括将数据从内核复制到用户自己的缓冲区都会由内核来操作) 再通知进程.这种模型与信号驱动模型的主要区别就是,信号驱动io由内核通知我们何时开始一个io操作,异步io操作由内核通知我们io何时完成.



下一个小结重点讲io多路复用技术,因为java的nio的核心类库多路复用器selector就是基于epoll的多路复用实现的.

目前支持io多路复用的系统调用有select pselect poll epoll 一开始都是用select的,后来发现还是epoll比较好.

epoll的好处

1,支持一个进程打开的socket(FD)不受限制(仅受限于操作系统的最大文件句柄数)

select最大的缺陷就是单个进程(这里说的都是对于一个进程来说的)所打开的FD是有一定限制的,它由FD\_SETSIZE设置,默认值是1024.

所以说这样的话一个java的服务器进程就只能链接1024个套接字.我们可以修改这个宏然后重新编译内核,不过这会带来网络效率的下降.

虽然一个进程的FD最多也就是1024了,那么同样的进程我在一个服务器上多部署不久行了.多进程方案(传统的apache方案)但是进程多了也有开销,而今进程间数据交互非常麻烦,这个也不是完美的方案.

epoll就不会有这个问题,他对于单个进程支持的fd上限是操作系统的最大文件句柄数,这个数字远大于1024, 1GB的机器上大约有10万个句柄.

2,io效率不会随着FD数目的增加而线性下降

传统select/poll的另一个致命弱点,就是当你拥有一个很大的socket集合时,由于网络延迟或者链路空闲,任一时间只有少部分的socket是活跃的,但是select/poll每次调用都会线性扫描全部的集合,导致效率线性下降.

epoll不会出现这个状况,他只会对活跃的socket进行操作,这是因为内核实现中,epoll是根据每个fd上面的callback(是每个socket都有一个callback哦,据说这个回调是注册在内核中或者cpu中的哦)函数实现的,那么只有活跃的socket才会去主动调用callback,idl的用户根据就不会动,这就是实现了一个伪的AIO.

但是呢,一定性能就提高了么,如果都处于活跃状态,epoll性能相对与select根本高不了多少的.

3,使用mmap加速内核与用户空间的消息传递

无论是select poll还是epoll都需要内核把fd消息通知给用户空间,如何避免不必要的内存复制就先得非常重要,epoll是通过内核和用户空间的mmap同一块内存来实现的.

### javaio的演进

Jdk1.4时,增加了java.nio的包,主要提供了,

进行异步IO操作的缓冲区bytebuffer等

进行异步io操作的管道pipe

io操作的channel ServerSocketChannel和SocketChannel.

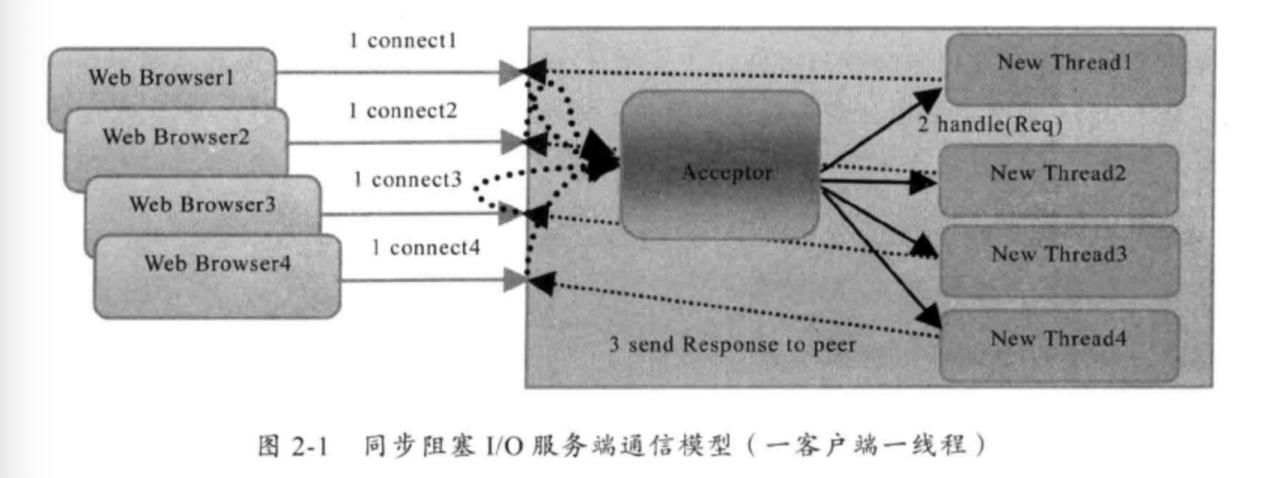
实现非阻塞io操作的多路复用器 selector.

文件通道,filechannel

Jdk1.7又对以前的nio库进行了升级,称为nio2.0, 这里就支持了AIO.

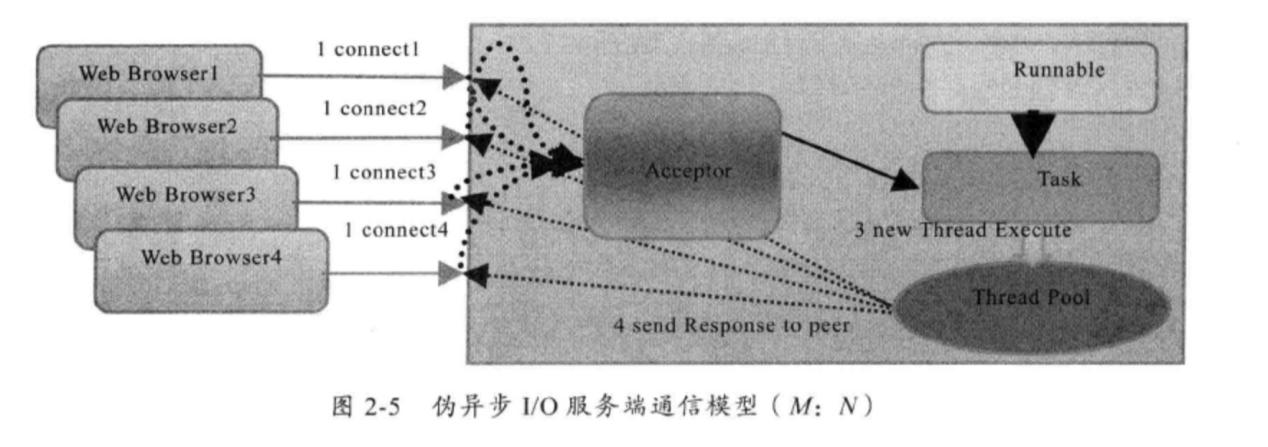
## NIO入门

### 纯BIO



就是那种一个链接一个线程的那种

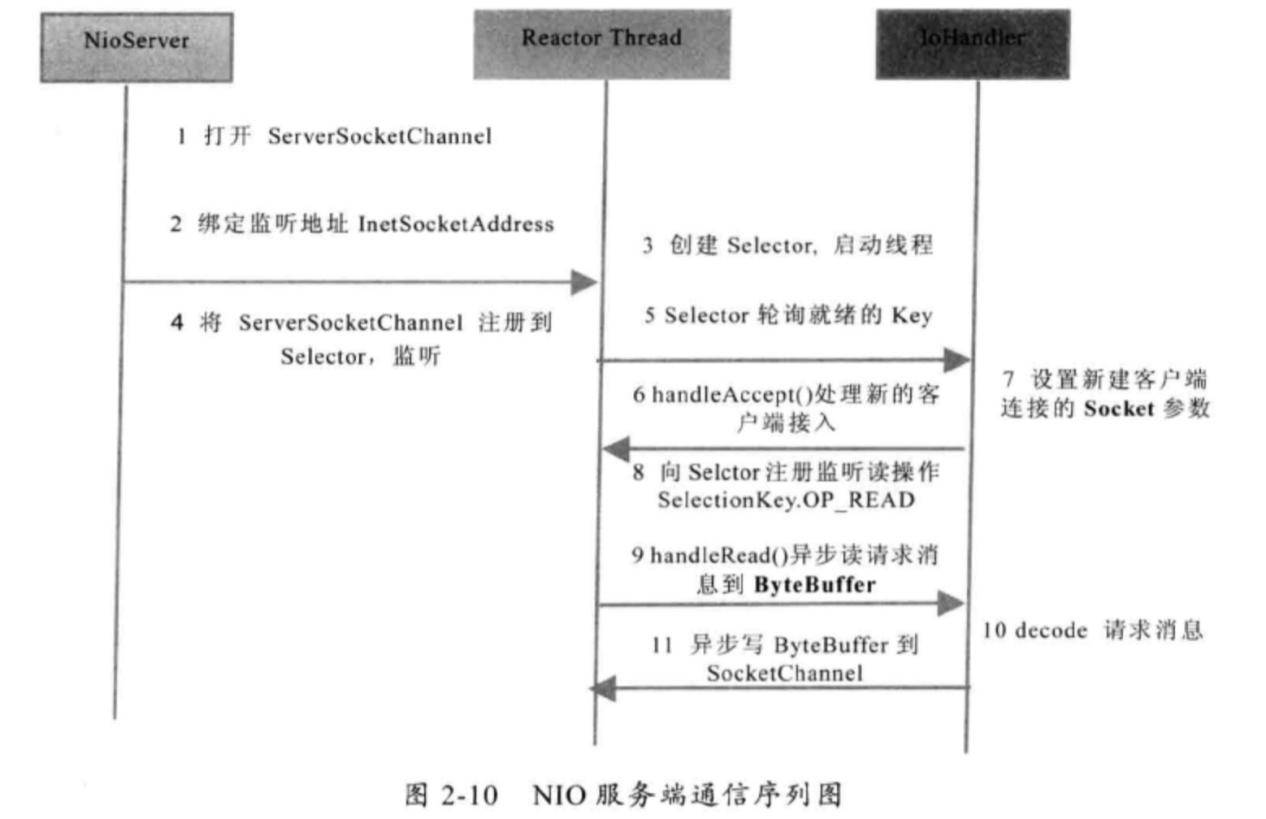
### 伪异步io(还是用的bio的接口)



这个也没啥太好说的,上面那个只要客户端多了,服务器肯定就炸了,线程会暴增,这个呢,无非就是对每个线程封装成了一个task,然后搞个线程池,再搞个消息队列,来控制,这样肯定不至于服务器炸了,但是一旦线程池耗尽,消息队列也就满了,那么还是提供不了服务,不是跟炸了一样么.

### Nio

1. 首先就是增加了buffer缓冲区,有bytebuffer charbuffer shortbuffer intbuffer等
2. 通道channel, 以前用bio的时候都是操作流,inputstream outputstream啥的,流是单向的,但是channel死后全双工的.以前用的ServerSocketChannel和SocketChannel都是channel接口的实现类啊.
3. Selector 多路复用器,就是实现了epoll.



这图画的还可以,创建serversocketchannel,创建复用器对象,然后吧channel的accept事件注册到复用器里面啊.

100-125行用户读取客户端的请求信息,首先创建一个bytebuffer,由于我们无法得知客户端发送的码流大小,作为例子,我们开辟一个1mb的缓冲区,然后调用socket的read方法读取请求的码流(read函数的参数就是这个bytebuffer咯).由于我们已经将socketchannel设置为非阻塞模式,所以read函数是非阻塞的,可以使用返回值进行判断.

返回值大于0,读到了字节,对字节进行编解码,

返回值等于0,没有读取到字节,正常场景,忽略.

返回值为-1 链路已经关闭,需要关闭socketchannel,释放资源.

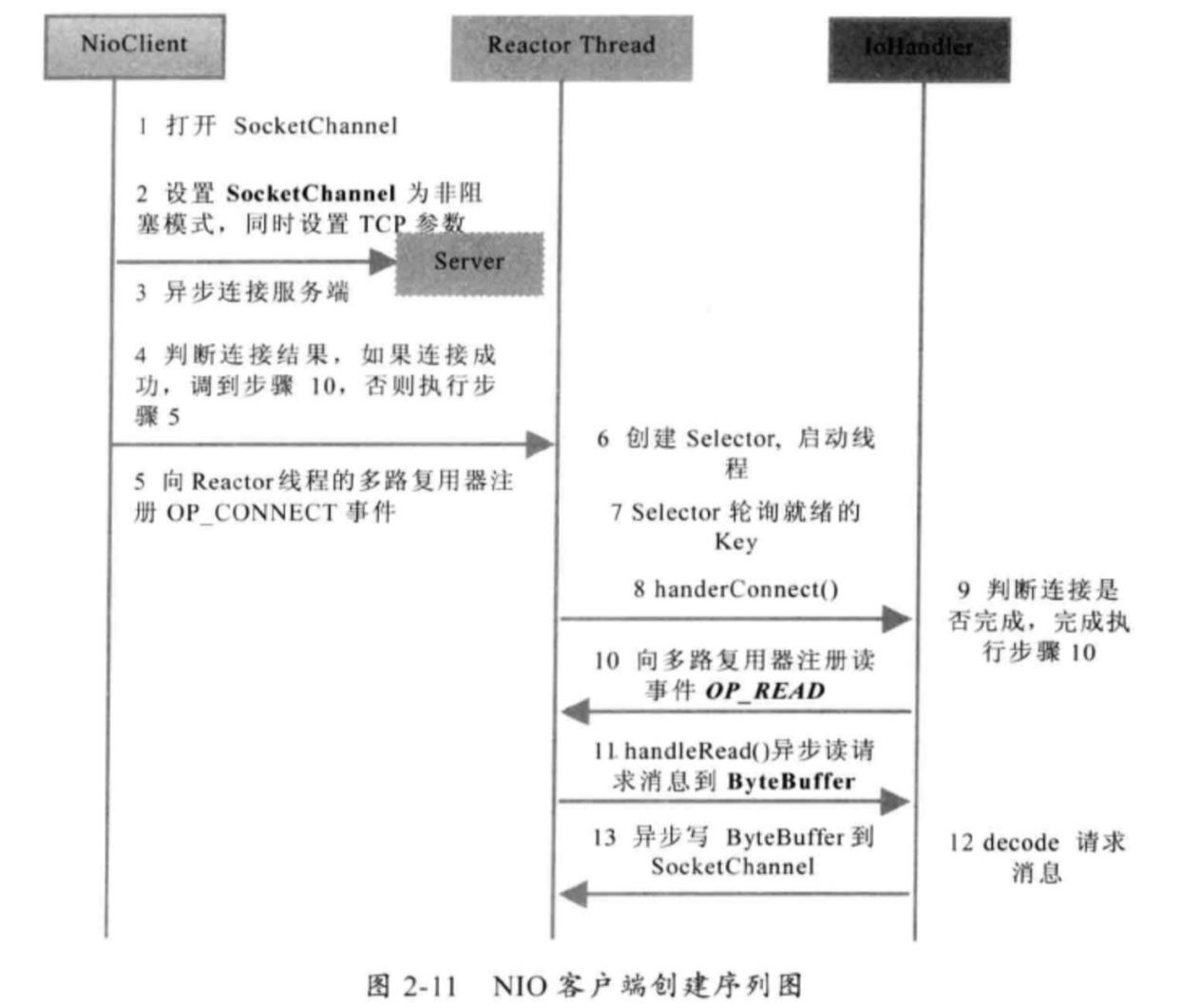
当读取到码流后,进行解码,首先对readbuffer进行flip操作,他的作用是将缓冲区(bytebuffer)的limit设置为position,position设置为0(这个操作其实就是标记这个缓冲区的内容我已经拿到了,缓冲区我释放了) 在这之前缓冲区的内容我们以前拿到了(调用bytebuffer的get操作将缓冲区可读的字节数组复制到新创建的字节数组中),我们操作就可以了

127-135行将应答消息异步发送到客户端,我们看下步骤

首先将字符串编码成字节数组,根据字节数组的容量创建bytebuffer, 调用bytebuffer的put操作将字节数组复制到缓冲区中,然后对缓冲区进行flip操作,最后调用socketchannel的write方法将缓冲区中的字节数组发送出去.

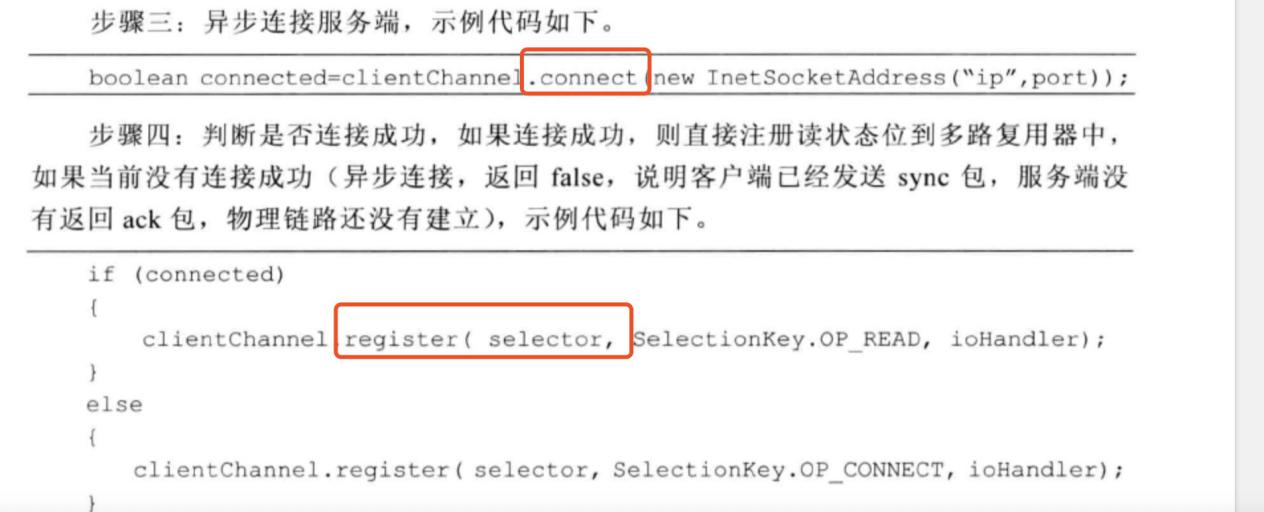
需要指出的是,由于socketchannel是异步非阻塞的,他并不保证一次能吧需要的发送字节数组发送完,此时会出现写半包的问题,我们需要注册写操作,不断轮询Selector将没有发送完的bytebuffer发送完毕,然后通过bytebuffer的hasremain()判断消息是否发送完成.

发送端





这里看到了,我们是可以通过socket设置buffer的大小的咯.



这个connect,就是链接到远端的, 这个connect函数会有个返回值,如果链接成功了,就在select上注册一个read事件,如果还没有链接成功就注册一个connect事件.与以前差不多咯.

### Aio 待补充

### 为什么选择netty

与之比较的是java原生nio api,开发十分复杂,各种异常场景,弄不明白的.

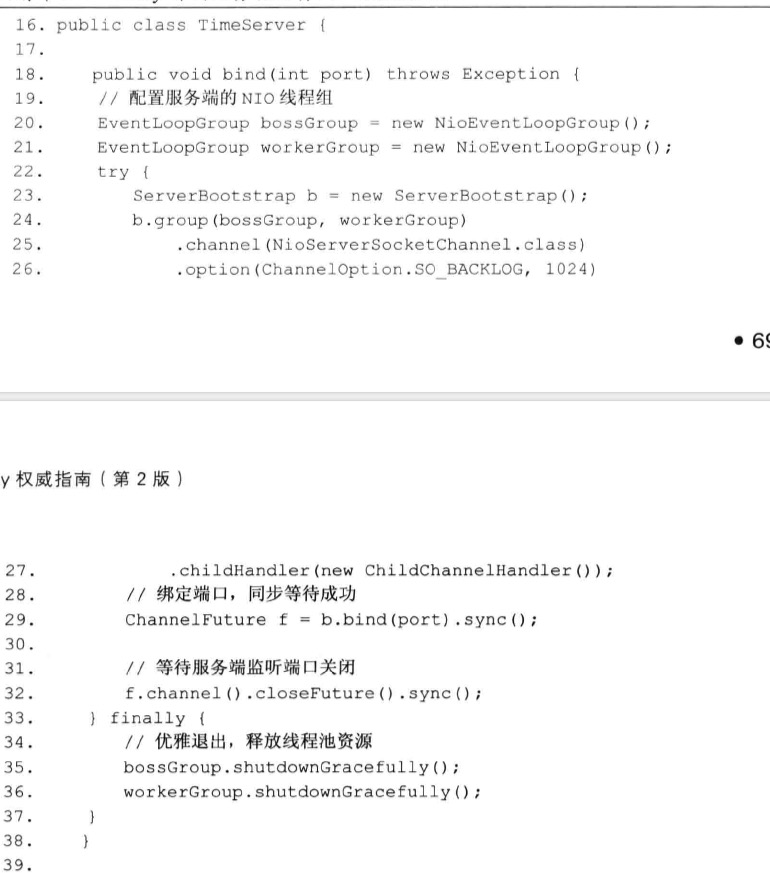
还得有多线程变成,网络变成,reactor模式等技术铺垫.

很多rpc框架底层都是netty,健壮稳定简单.

## Netty入门

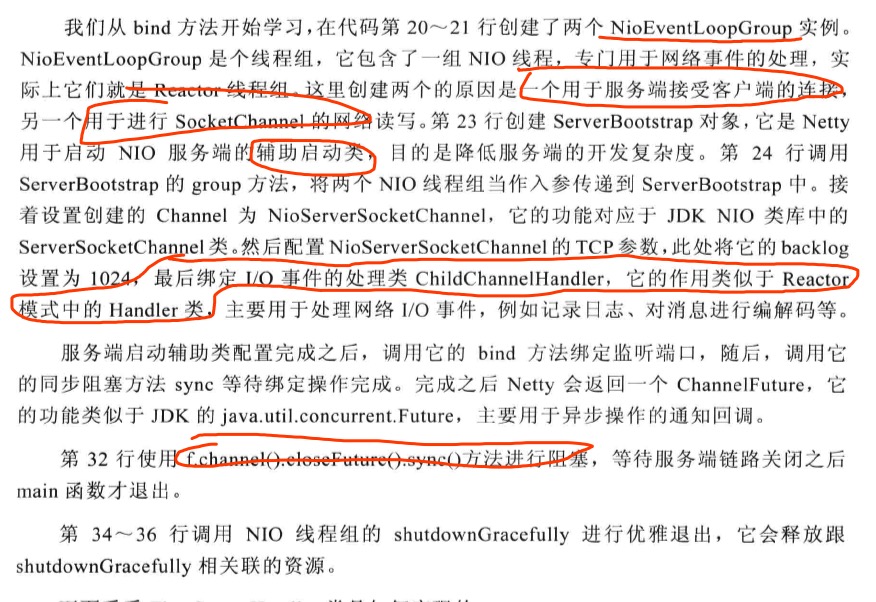
### 服务端

这个是个tcp的server,http是接不到的



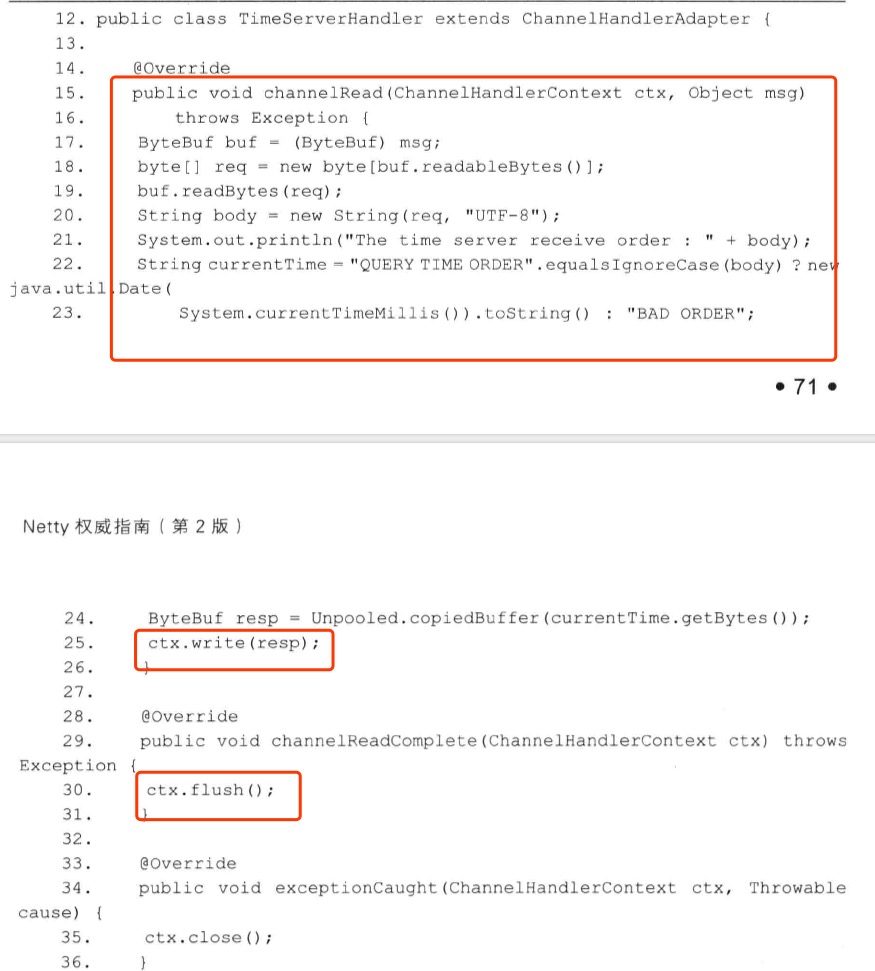
注意在上图的bootstrap是serverbootstrap,而客户端就是bootstrap还有就是channel()

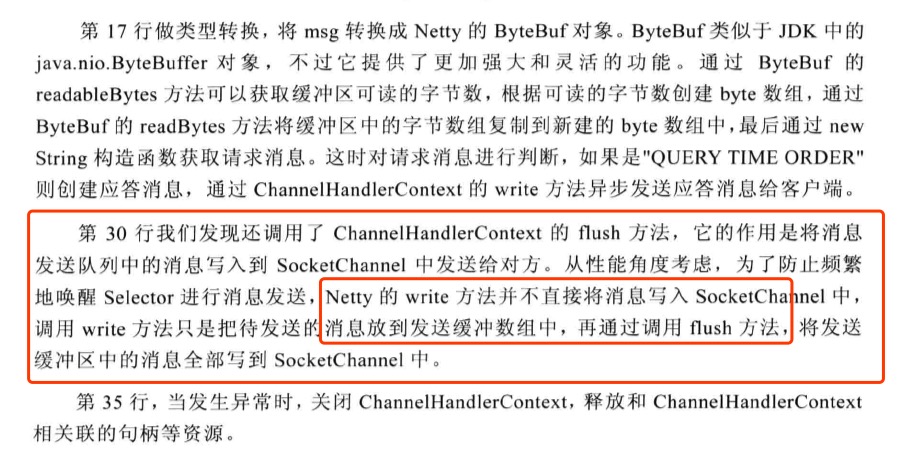
这里是serversocketchannel,到后面就是socketchannel了.



以前我们的看到的bossgroup和workergroup都是netty的东西啊.

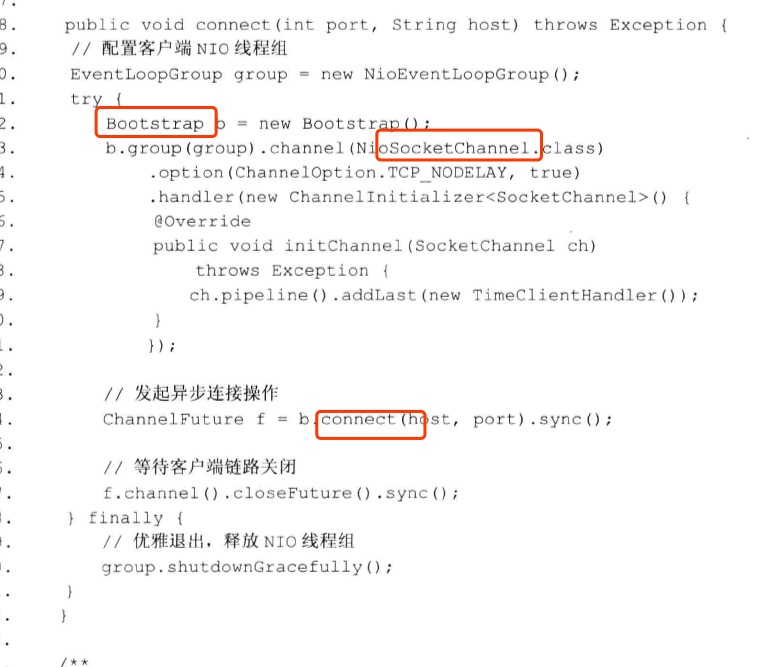
下面看看handler里面的代码.

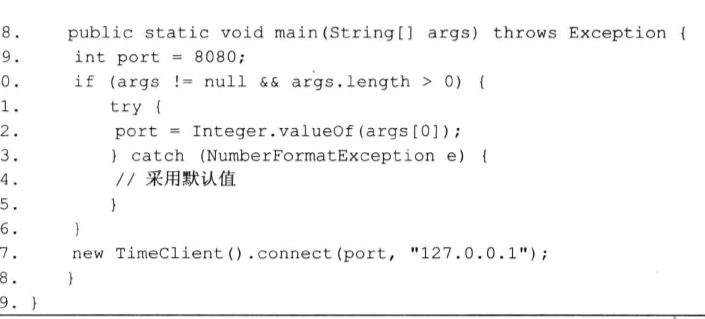




上面也说了就是netty的write并不会直接把消息就弄到socketchannel级别的,而是有个缓冲的地方,调用flush才会触发到socketchannelnio级别的东西.

### 客户端





这里的客户端的bootstrap和服务端的是有些不同的.

### Tcp的粘包和拆包

上面已经讲过了,粘包和拆包了,经过代码验证,确实我们writeandflush了100次,注意哦,已经flush过了,server还是只收到了两次,而且收到包的数据是这100次篡的,所以数据已经不是我们规定好的格式了,所以可以确定是粘包了,代码可见.

要想解决粘包问题,代码已改

socketChannel.pipeline().addLast(new LineBasedFrameDecoder(1024));  
socketChannel.pipeline().addLast(new StringDecoder());

主要是在initchannel的时候加了这两行,

然后在发消息的时候每writeandflush的时候,里面的buff的内容拼一个字符串结束符

System.getProperty("line.separator")

还有就是在channelread这个函数里,不用把msg从bytebuf再往字符串一层层转了,而是直接强转字符串就行(这个是new StringDecoder())的功劳

其实也没啥,就是new LineBasedFrameDecoder(1024)在解析的时候会收到一个结束符,就把这个当作一个包了,就是根据结束符拆一下而已咯.