io操作：内核空间的数据读取到用户空间

内核空间只有内核进程可以访问，而用户进程只能访问用户空间

或者通过用户进程调用内核进程读取内核空间中的数据

用户空间产生一个读请求，请求再转交由内核空间执行

1. 内核检查读取的数据是否就绪

2. 如果就绪，内核将数据从内核空间复制到用户空间(内存上拷贝)

同步/异步：

同步操作时，线程会阻塞(释放CPU)，等待数据，数据准备好，相处处于就绪状态，同步需要等待(阻塞或轮序直到获取到数据)，发起请求后，需要等待返回结果；同步指的是用户进程触发IO操作并等待或者轮询的去查看IO操作是否就绪。

异步操作时，线程发起IO后，会立即获得一个值，然后继续执行，等到IO操作完成后，通知线程，异步触发IO后就做其他的事情，内核数据完成后，内核通知进程，发起请求后，不用等待，继续执行；异步是指用户进程触发IO操作以后便开始做自己的事情，而当IO操作已经完成的时候会得到IO完成的通知（异步的特点就是通知）

阻塞与非阻塞：IO操作能不能立即得到返回答应，不能立即得到答应，等待(阻塞)

阻塞方式是指：当试图对该文件描述符进行读写时,没有东西可读,或者暂时不可写, 程序就进入等待状态, 直到有东西可读或者可写为止

非阻塞是指：如果没有东西可读, 或者不可写, 读写函数马上返回, 而不会等待

IO操作：发起IO请求+实际的IO操作

同步和异步是针对应用程序和内核的交互而言的；同步指的是用户进程触发IO操作并等待或者轮询的去查看IO操作是否就绪，而异步是指用户进程触发IO操作以后便开始做自己的事情，而当IO操作已经完成的时候会得到IO完成的通知。

阻塞和非阻塞是针对进程在访问数据的时候；根据IO操作的就绪状态来采取的不同方式，说白了是一种读取或者写入操作函数的实现方式，阻塞方式下读取或者写入函数将一直等待，而非阻塞方式下，读取或者写入函数会立即返回一个状态值。

IO操作可以分为3类：同步阻塞（即早期的BIO操作）、同步非阻塞（NIO）、异步非阻塞（AIO）：

同步阻塞(BIO)：

用户进程在发起一个IO操作以后，必须等待IO操作的完成，只有当真正完成了IO操作以后，用户进程才能运行。JAVA传统的IO模型属于此种方式。

每当服务端收到客户端的一个连接，就会创建一个新的线程处理当前的连接。一个线程只能处理一个客户端的请求。

同步非阻塞(NIO)：

用户进程发起一个IO操作以后便可返回做其它事情，但是用户进程需要时不时的询问IO操作是否就绪，这就要求用户进程不停的去询问，从而引入不必要的CPU资源浪费。

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请求时才启动一个线程进行处理。用户进程任然需要轮询判断数据是否就绪。(linux内核优化)

异步非阻塞(AIO/NIO2.0)：

应用发起一个IO操作以后，不等待内核IO操作的完成，等内核完成IO操作以后会通知应用程序。(同步需要主动询问IO)

AIO未使用多路复用，AIO基于回调函数

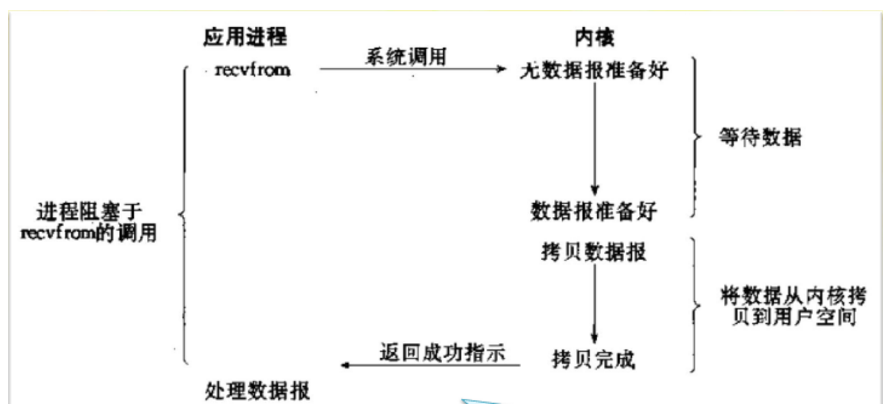
使用场景：

BIO：连接数目小，资源大

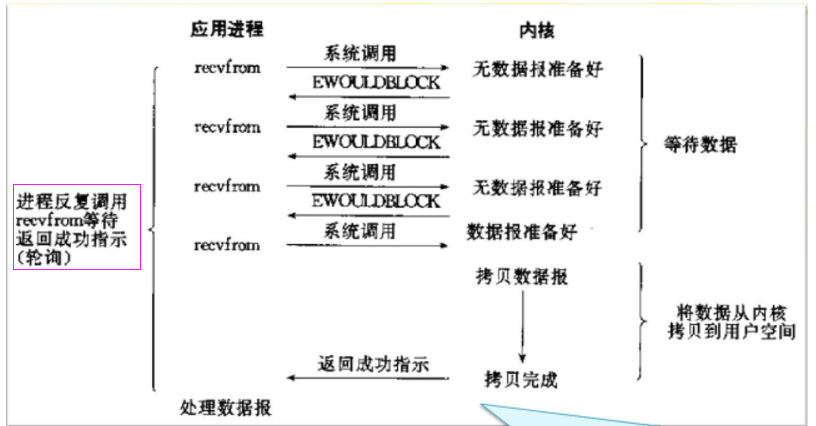
NIO：连接数目多，资源小

AIO：连接数目多，资源大

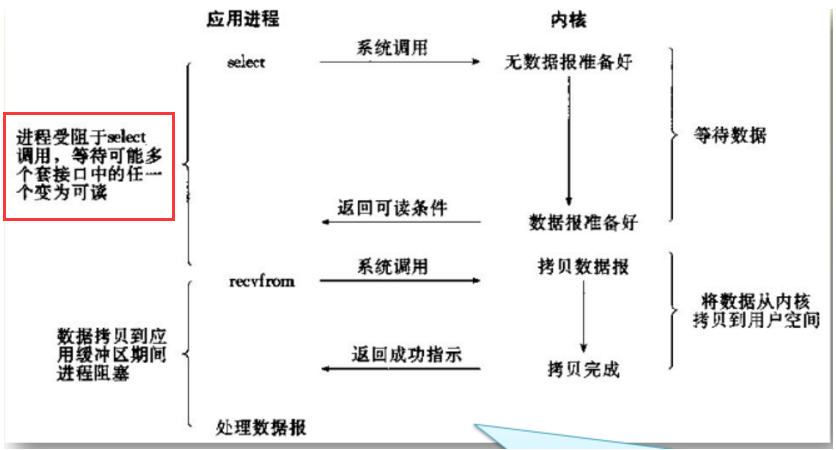
阻塞IO：进程在调用recvfrom函数(向内核读取数据的操作函数)的时候阻塞，直到数据准备完，进程由阻塞状态转换为就绪状态



非阻塞IO：数据未准备好返回一个EWOULDBLOCK/数据准备好，返回一个System Call，非阻塞会立即返回一个状态，若不是System Call，进程轮询调用recvfrom函数



多路复用IO：对于非阻塞IO，每个IO请求就创建一个进程，消耗大，而多路复用是，进程在请求IO时阻塞于select函数(操作系统的原语)，select会依次轮询所有stock，有数据准备好的，就通知对应的线程调用recvfrom函数



多路复用和非阻塞的区别：

非阻塞对应多个线程，每次轮询的时候，只会轮询自己的，而不会去轮询其他线程数据是否准备好而多路复用只有一个线

程(select)，所有进程都阻塞于select函数，等待select函数的通知，是否调用recvfrom函数，而select轮询的是所有stock

阻塞是选择等待，非阻塞是返回一个标志信息

阻塞I/O：内核在检查数据未就绪时，会一直等待，直到数据就绪

非阻塞I/O：如果数据没有就绪，则会返回一个标志信息告知用户线程当前要读的数据没有就绪

fd：文件描述符，fd是一个整数，这个整数对应文件流，对fd操作就是对数据流操作、创建一个stock，系统调用会返回一个文件描述符

在多路复用IO模型中，会有一个线程不断去轮询多个socket的状态，只有当socket真正有读写事件时，才真正调用实际的IO读写操作，只需要使用一个线程就可以管理多个socket,socket对应一个IO请求

线程请求IO时，阻塞于select函数，为线程创建一个stocket，另外一个线程轮询所有的stocket，判断是否发生读写请求，再唤起阻塞于select的对应线程

原来阻塞IO对每个stock创建一个线程，线程多，切换也消耗资源，非阻塞只有一个线程，轮询所有的stock，看哪些stock处于就绪状态，返回处于就绪状态的fd

忙轮询：线程不断轮序所有stock

无差别轮序：在忙轮询中，如果所有的stock都处于未就绪状态，不断的轮序会消耗CPU，而又无就绪stock，当第一次轮序完所有stock后，发现无就绪，线程就会阻塞，当有一个或多个stock处于就绪状态时，再通知线程轮询，这样可以减少轮序次数，这就是无差别轮序，select和poll采用无差别轮序

最小轮询：当有IO准备就绪时，可以通过select查看哪些fd就绪，但是select只知道有IO就绪，但是不知道有几个，是哪个，所以select要轮序所有的stock，时间复杂度为n(O)，stock越多，效率越低，而内核空间默认1G(32位)，所以select最多能处理1024个stock，只会将发送了IO等待的流返回(select是判断是否有处于IO的stock，然后将所有的stock返回，再轮序，而最小轮序epoll只会返回处于IO的stock，返回的都是有效的，效率高)

Select函数监控多个fd，有fd处于IO时，会通知fd返回



Select函数的实现

maxfd：需要监视的最大的文件描述符值+1；

readset：需要检测的可读文件描述符的集合；

writeset：需要检测的可写文件描述符的集合

exceptset：需要检测的异常文件描述符的集合

fd\_set是一个文件描述符集合

timeout：超时时间

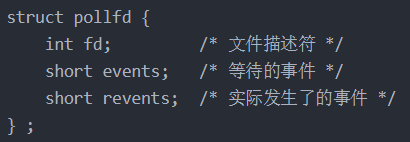
返回值为处于IO的个数

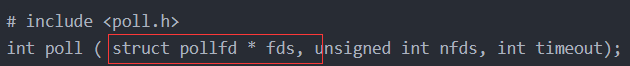
Select函数过程：

1、先从用户空间拷贝fd\_set到内核空间；2、遍历所有的fd，调用fd对应的poll函数，poll方法返回时，就会返回一个判断当前fd是否就绪的mask掩码；3、遍历完所有的fd后，都没有处于就绪的stock，select函数阻塞，等待被唤醒；4、把fd\_set从内核空间复制到用户空间

Select的缺点：

1、每次调用select函数，都会将所有fd集合复制到内核空间；2、每次调用完也会将所有fd复制到用户空间；3、select支持的fd太小(1024)





Poll函数和select类似，只是Poll选择了pollfd结构体来处理文件描述符

fds：需要被监视的文件描述符集合；

nfds：被监视的文件描述符数量；

timeout：超时时间

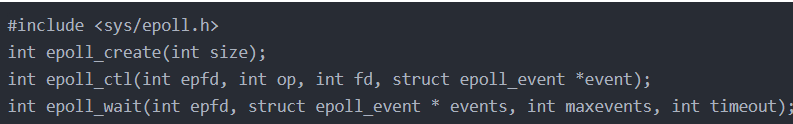
返回值为处于IO的stock个数

Poll函数的实现过程：

Poll和select非常类似，只是fd集合不同，select采用fd\_set，poll采用pollfd，其他都一样

Select和poll需要将整个文件描述符集合在用户空间和内核空间之间来回拷贝，这样随着文件描述符的数量增加，相应的开销也随之增加。

poll函数与select函数的最大不同之处在于：select函数有最大文件描述符的限制，一般1024个，而poll函数对文件描述符的数量没有限制。



epoll\_create函数：

用创建一个epoll的句柄；

size用来告诉内核这个监听的数目一共有多大，占用一个fd值；

epoll\_ctl函数：epoll的事件注册函数；

epfd：epoll\_create()的返回值；

op：动作，有三种取值：

EPOLL\_CTL\_ADD：注册新的fd到epfd中；

EPOLL\_CTL\_MOD：修改已经注册的fd的监听事件；

EPOLL\_CTL\_DEL：从epfd中删除一个fd；

fd：需要监听的fd；

event： 告诉内核需要监听什么事件，文件描述符可以读/写/发生错误等

epoll\_wait函数：等待事件的产生；

Epoll函数实现过程：

Epoll有3个函数，epoll\_create,epoll\_ctl,epoll\_wait，epoll\_create是创建一个epoll句柄，epoll\_tcl是注册要监听的事件类型，epoll\_wait等待事件的产生；select的第一个缺点，epoll保证每个fd只会拷贝一次，fd的状态通过tcl来修改；第二个缺点，epoll会为每个fd创建一个回调函数，当fd就绪时，会调用此回调函数，此回调函数的作用就是将处于就绪状态的fd加入一个就绪链表，而epoll\_wait就是在一个间隔内轮序此就绪链表，将就绪的fd复制到用户状态，所以只会复制处于就绪的fd到用户空间，解决了select的缺点1,2；epoll没有fd的上限

Epoll对fd操作有两种模式：水平触发(LT)/边缘触发(ET)，默认LT

LT模式：当epoll\_wait检测到描述符事件发生并将此事件通知应用程序，应用程序可以不立即处理该事件。下次调用epoll\_wait时，会再次响应应用程序并通知此事件。

ET模式：当epoll\_wait检测到描述符事件发生并将此事件通知应用程序，应用程序必须立即处理该事件。如果不处理，下次调用epoll\_wait时，不会再次响应应用程序并通知此事件。

当应用程序调用epoll\_wait的时候，会返回数据，此时可能只有一部分数据返回(分批次返回)，LT模式先，应用程序读取完返回数据后，还可以继续读取(因为还没返回完)，而ET模式下，就必须读取完所有数据，下次epoll\_wait再返回数据时，ET就不会再读取了，所以ET模式下，一次就要将数据返回完(可以轮询调用epoll\_wait，直到数据被读取完)

Poll是水平触发(LT)

ET模式在很大程度上减少了epoll事件被重复触发的次数，因此效率要比LT模式高。epoll工作在ET模式的时候，必须使用非阻塞套接口，以避免由于一个文件句柄的阻塞读/阻塞写操作把处理多个文件描述符的任务饿死。

Select，poll，epoll(epoll\_wait,epoll\_tcl,epoll\_create)是操作内核的函数，应用程序可以通过这个3个函数实现对内核数据的读写

Select轮序的是整个fd集合，而epoll需要轮序就绪链表是否有fd(回调体现的)

Epoll性能好只是表面上的，在连接数少且连接都十分活跃的情况下，select比epoll好，毕竟epoll需要很多回调函数

Select效率低使用因为每次轮序，且复制所有fd

Select和poll轮询的是整个fd，而epoll需要轮询就绪队列是否为空



## Zero-Copy(零拷贝)：

网络读取图片过程：

read：

文件A拷贝到了kernel模式；

CPU控制将kernel模式数据copy到user模式下；

write：

将user模式下的内容copy到kernel模式下的socket的buffer中；

最后将kernel模式下的socket buffer的数据copy到网卡设备中传送；

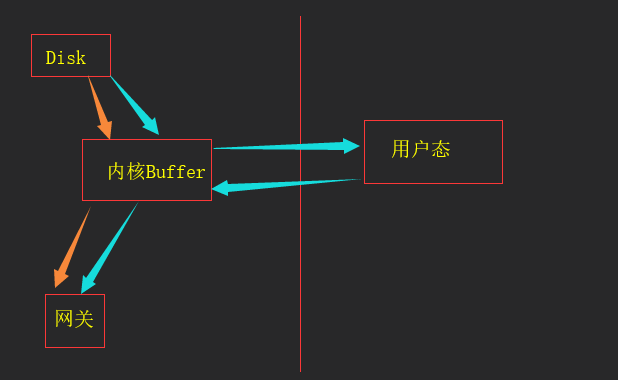
零拷贝是直接将请求kernel直接把disk的data传输给socket：

首先（通过DMA）将数据从磁盘读取到kernel buffer中；

然后将kernel buffer拷贝到socket buffer中；

最后将socket buffer中的数据copy到网卡设备中发送；

零拷贝是针对kernel来讲的，数据在kernel模式下是Zero-Cop，第一种需要复制2次



Java NIO中的FileChannal.transferTo()方法就是Zero-Copy

## Java NIO

IO是面向流的，NIO是面向缓冲区的

Channel：

Channel和IO中的Stream(流)是差不多一个等级的，Stream是单向的，Channel是全双工

FileChannel ----- 对文件操作

DatagramChannel ----- UDP

SocketChannel ----- 网络

ServerSocketChannel

Buffer：

NIO中的关键Buffer实现有：ByteBuffer, CharBuffer, DoubleBuffer, FloatBuffer, IntBuffer, LongBuffer, ShortBuffer，MappedByteBuffer等，Buffer用于缓存数据

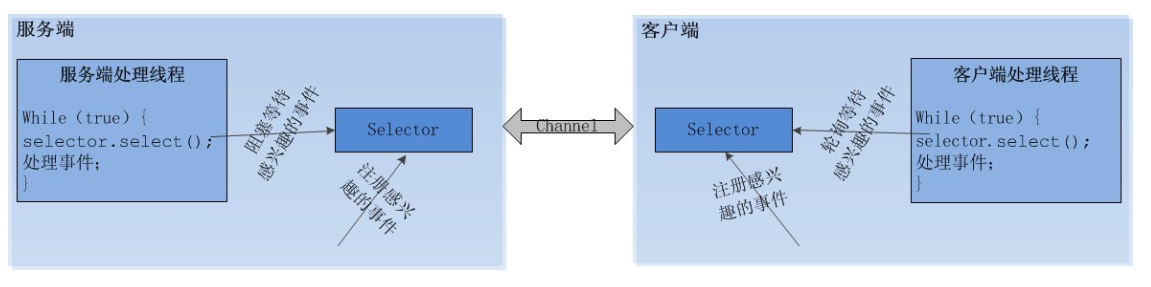
读数据时，Channel写入Buffer

写数据时，Buffer写入Channel

Selector：

轮序Channel

### Channel



Chanel作为数据输入输出的通道

FileChannel ------- 文件通道，文件通道总是阻塞式的，因此不能被置于非阻塞模式。

DatagramChannel ------- UDP通道

SocketChannel -------- 网络通道(客户端)

ServerSocketChannel ------- 网络通道(服务端)

### 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。

Buffer作为容器，读取或写入Channel数据

### Selector