网络I/O的模型大致有如下几种：

* 同步模型（synchronous I/O）
  + 阻塞I/O（bloking I/O）
  + 非阻塞I/O（non-blocking I/O）
  + 多路复用I/O（multiplexing I/O）
  + 信号驱动式I/O（signal-driven I/O）
* 异步I/O（asynchronous I/O）

网络I/O的本质是socket的读取，socket在linux系统被抽象为流，I/O可以理解为对流的操作。这个操作又分为两个阶段：

1. 等待流数据准备（wating for the data to be ready）。
2. 从内核向进程复制数据（copying the data from the kernel to the process）。

对于socket流而已，

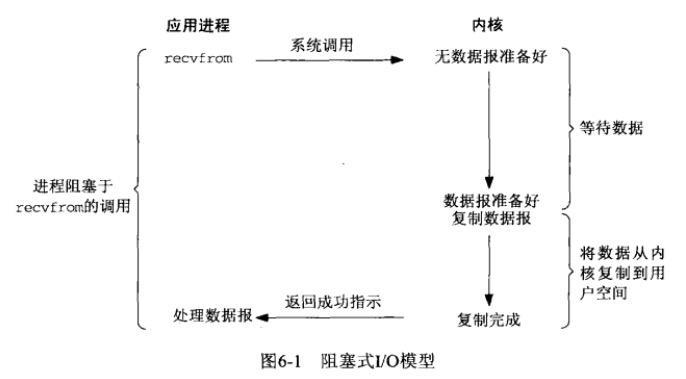
* 第一步通常涉及等待网络上的数据分组到达，然后被复制到内核的某个缓冲区。
* 第二步把数据从内核缓冲区复制到应用进程缓冲区。

### **I/O模型**

举个简单比喻，来了解这几种模型。网络IO好比钓鱼，等待鱼上钩就是网络中等待数据准备好的过程，鱼上钩了，把鱼拉上岸就是内核复制数据阶段。钓鱼的人就是一个应用进程。

#### **阻塞I/O（bloking I/O）**

阻塞I/O是最流行的I/O模型。它符合人们最常见的思考逻辑。****阻塞就是进程 "被" 休息, CPU处理其它进程去了****。在网络I/O的时候，进程发起recvform系统调用，然后进程就被阻塞了，什么也不干，直到数据准备好，并且将数据从内核复制到用户进程，最后进程再处理数据，在等待数据到处理数据的两个阶段，整个进程都被阻塞。不能处理别的网络I/O。大致如下图：



1.png

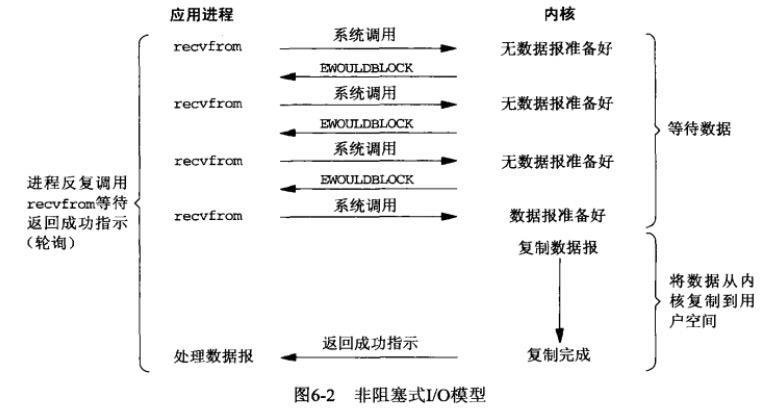
这就好比我们去钓鱼，抛竿之后就一直在岸边等，直到等待鱼上钩。然后再一次抛竿，等待下一条鱼上钩，等待的时候，什么事情也不做，大概会胡思乱想吧。

阻塞IO的特点就是在IO执行的两个阶段都被block了

#### **非阻塞I/O（non-bloking I/O）**

在网络I/O时候，非阻塞I/O也会进行recvform系统调用，检查数据是否准备好，与阻塞I/O不一样，"非阻塞将大的整片时间的阻塞分成N多的小的阻塞, 所以进程不断地有机会 '被' CPU光顾"。

也就是说非阻塞的recvform系统调用调用之后，进程并没有被阻塞，内核马上返回给进程，如果数据还没准备好，此时会返回一个error。进程在返回之后，可以干点别的事情，然后再发起recvform系统调用。重复上面的过程，循环往复的进行recvform系统调用。这个过程通常被称之为轮询。轮询检查内核数据，直到数据准备好，再拷贝数据到进程，进行数据处理。需要注意，拷贝数据整个过程，进程仍然是属于阻塞的状态。



2.png

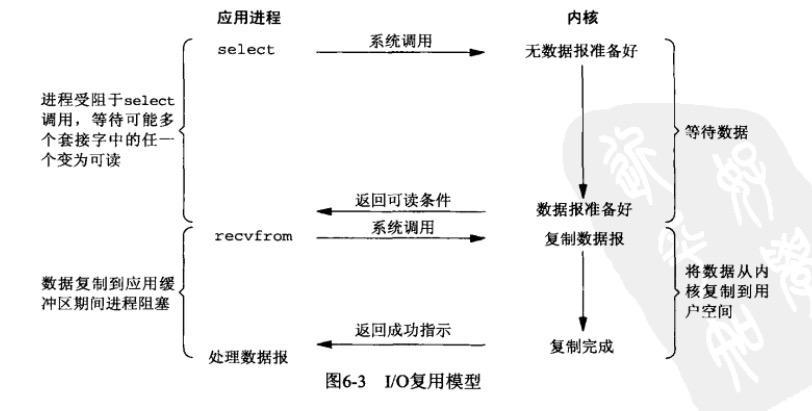
我们再用钓鱼的方式来类别，当我们抛竿入水之后，就看下鱼漂是否有动静，如果没有鱼上钩，就去干点别的事情，比如再挖几条蚯蚓。然后不久又来看看鱼漂是否有鱼上钩。这样往返的检查又离开，直到鱼上钩，再进行处理。

非阻塞 IO的特点是用户进程需要****不断的主动询问****kernel数据是否准备好。

#### **多路复用I/O（multiplexing I/O）**

可以看出，由于非阻塞的调用，轮询占据了很大一部分过程，轮询会消耗大量的CPU时间。结合前面两种模式。如果轮询不是进程的用户态，而是有人帮忙就好了。多路复用正好处理这样的问题。

**多路复用有两个特别的系统调用select或poll**。select调用是内核级别的，select轮询相对非阻塞的轮询的区别在于---前者可以等待多个socket，当其中任何一个socket的数据准好了，就能返回进行可读，然后进程再进行recvform系统调用，将数据由内核拷贝到用户进程，当然这个过程是阻塞的。多路复用有两种阻塞，select或poll调用之后，会阻塞进程，与第一种阻塞不同在于，此时的select不是等到socket数据全部到达再处理, 而是有了一部分数据就会调用用户进程来处理。如何知道有一部分数据到达了呢？监视的事情交给了内核，内核负责数据到达的处理。也可以理解为"非阻塞"吧。



3.png

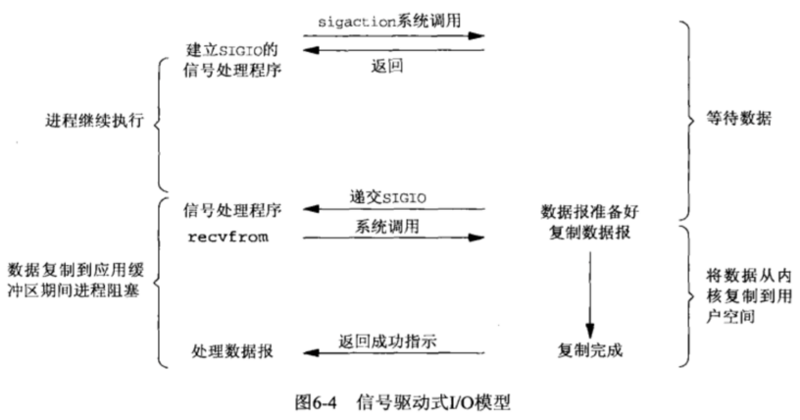
**对于多路复用，也就是轮询多个socket。**钓鱼的时候，我们雇了一个帮手，他可以同时抛下多个钓鱼竿，任何一杆的鱼一上钩，他就会拉杆。他只负责帮我们钓鱼，并不会帮我们处理，所以我们还得在一帮等着，等他把收杆。我们再处理鱼。多路复用既然可以处理多个I/O，也就带来了新的问题，多个I/O之间的顺序变得不确定了，当然也可以针对不同的编号。

多路复用的特点是通过一种机制一个进程能同时等待IO文件描述符，内核监视这些文件描述符（套接字描述符），其中的任意一个进入读就绪状态，select， poll，epoll函数就可以返回。对于监视的方式，又可以分为 select， poll， epoll三种方式。

如上图所示, 当调用了 select 后, select 会阻塞住, 等待数据报套接字变为可读. 当 select 返回套接字可读这一条件时, 我们就可以调用 recvfrom 把所读取的数据报复制到应用进程缓冲区.

对比阻塞式 IO, IO 复用模型优势并不明显, 并且从使用方式来说, IO 复用模型还需要多调用一次 select, 因此从易用性上来说, 比阻塞式 IO 还略有不足. 不过 select 的杀手锏在于它可以监听多个文件描述符, 大大减小了阻塞线程的个数.

## 信号驱动 IO 模型

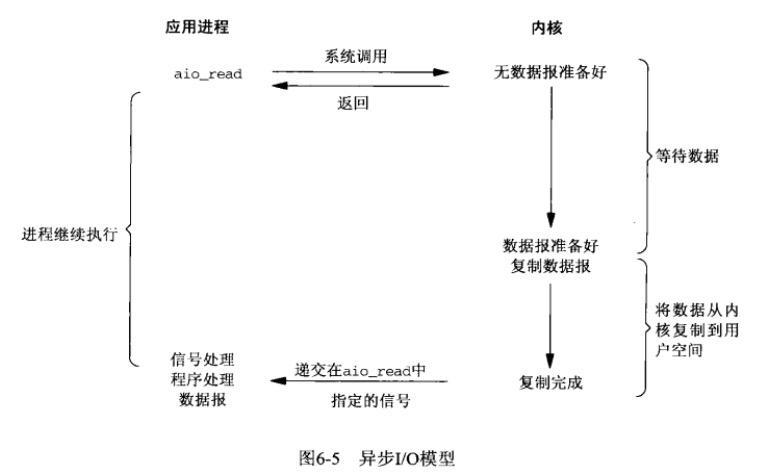


信号驱动模型如上图所示. **当文件描述符就绪时, 我们可以让内核以信号的方式通知我们.**我们首先需要开启套接字的信号驱动式 IO 功能, 并通过 sigaction 系统调用安装一个信号处理函数. sigaction 系统调用是异步的, 它会立即返回. 当有数据时, 内核会给此进程发送一个 SIGIO 信号, 进而我们的信号处理函数就会被执行, 我们就可以在这个函数中调用 recvfrom 读取数据.

了解了前面三种模式，在用户进程进行系统调用的时候，他们在等待数据到来的时候，处理的方式不一样，直接等待，轮询，select或poll轮询，第一个过程有的阻塞，有的不阻塞，有的可以阻塞又可以不阻塞。当时第二个过程都是阻塞的。从整个I/O过程来看，他们都是顺序执行的，因此可以归为同步模型(asynchronous)。都是进程主动向内核检查。

#### **异步I/O（asynchronous I/O）**

相对于同步I/O，异步I/O不是顺序执行。**用户进程进行aio\_read系统调用之后，无论内核数据是否准备好，都会直接返回给用户进程，然后用户态进程可以去做别的事情。等到socket数据准备好了，内核直接复制数据给进程，然后从内核向进程发送通知。I/O两个阶段，进程都是非阻塞的。**



4.png

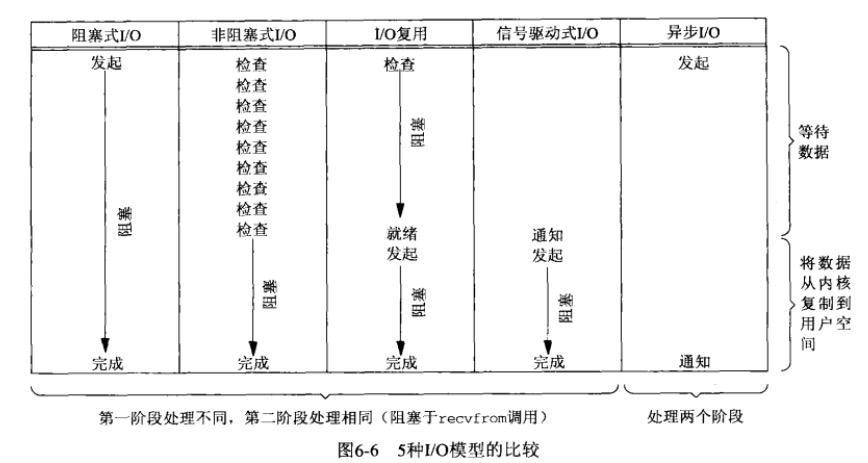
比之前的钓鱼方式不一样，这一次我们雇了一个钓鱼高手。他不仅会钓鱼，还会在鱼上钩之后给我们发短信，通知我们鱼已经准备好了。我们只要委托他去抛竿，然后就能跑去干别的事情了，直到他的短信。我们再回来处理已经上岸的鱼。

### **同步和异步的区别**

通过对上述几种模型的讨论，需要区分阻塞和非阻塞，同步和异步。他们其实是两组概念。区别前一组比较容易，后一种往往容易和前面混合。对于同步和异步而言，往往是一个函数调用之后，是否直接返回结果，如果函数挂起，直到获得结果，这是同步；如果函数马上返回，等数据到达再通知函数，那么这是异步的路程。

至于阻塞和非阻塞，则是函数是否让线程挂起不再往下执行。通常同步阻塞，异步非阻塞。什么情况下是异步阻塞呢？即函数调用之后并没有返回结果而注册了回调函数，非阻塞的情况下，函数也马上返回，可是如果此时函数不返回，那么此时就是阻塞的状态，等数据到达通知函数，依然是异步的过程。

区分阻塞和非阻塞只要区分函数调用之后是否挂起返回就可以了，区分异步和同步，则是函数调用之后，数据或条件满足之后如何通知函数。等待数据返回则是同步，通过回调则是异步。



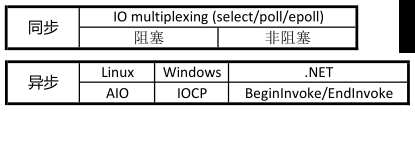
1. Png

如图所示, 上述五中 IO 模型中, 前四种模型(阻塞 IO, 非阻塞 IO, IO 复用, 信号驱动 IO)的主要区别在于第一阶段, 因为他们的第二阶段是一样的: 在数据从内核复制到调用者的缓冲区期间, 进程阻塞于 recvfrom 调用. 而第五种, 即异步 IO 模型中, 两个阶段都不需要应用进程处理, 内核为我们处理好了数据的等待和数据的复制过程.

对于同步模型，主要是第一阶段处理方法不一样。而异步模型，两个阶段都不一样。

## 总结

在处理网络 IO 操作时, 阻塞和非阻塞 IO 都是同步 IO.只有调用了特殊的 API 才是异步 IO.



因此网上常说的 "同步阻塞 IO", "同步非阻塞 IO" 其实就是阻塞 IO 模型和非阻塞 IO 模型, 因为阻塞 IO 和非阻塞 IO 模型都是同步的, 加了 "同步" 二字其实是多余了.网络上常说的 "异步非阻塞 IO" 其实就是异步 IO 模型.

总结：

（1）select，poll实现需要自己不断轮询所有fd集合，直到设备就绪，期间可能要睡眠和唤醒多次交替。而epoll其实也需要调用epoll\_wait不断轮询就绪链表，期间也可能多次睡眠和唤醒交替，但是它是设备就绪时，调用回调函数，把就绪fd放入就绪链表中，并唤醒在epoll\_wait中进入睡眠的进程。虽然都要睡眠和交替，但是select和poll在“醒着”的时候要遍历整个fd集合，而epoll在“醒着”的时候只要判断一下就绪链表是否为空就行了，这节省了大量的CPU时间。这就是回调机制带来的性能提升。

（2）select，poll每次调用都要把fd集合从用户态往内核态拷贝一次，并且要把current往设备等待队列中挂一次，而epoll只要一次拷贝，而且把current往等待队列上挂也只挂一次（在epoll\_wait的开始，注意这里的等待队列并不是设备等待队列，只是一个epoll内部定义的等待队列）。这也能节省不少的开销。

我们通俗一点讲：

Level\_triggered(水平触发)：当被监控的文件描述符上有可读写事件发生时，epoll\_wait()会通知处理程序去读写。如果这次没有把数据一次性全部读写完(如读写缓冲区太小)，那么下次调用 epoll\_wait()时，它还会通知你在上没读写完的文件描述符上继续读写，当然如果你一直不去读写，它会一直通知你！！！如果系统中有大量你不需要读写的就绪文件描述符，而它们每次都会返回，这样会大大降低处理程序检索自己关心的就绪文件描述符的效率！！！

Edge\_triggered(边缘触发)：当被监控的文件描述符上有可读写事件发生时，epoll\_wait()会通知处理程序去读写。如果这次没有把数据全部读写完(如读写缓冲区太小)，那么下次调用epoll\_wait()时，它不会通知你，也就是它只会通知你一次，直到该文件描述符上出现第二次可读写事件才会通知你！！！这种模式比水平触发效率高，系统不会充斥大量你不关心的就绪文件描述符！！！

由此可见，水平触发时如果系统中有大量你不需要读写的就绪文件描述符，而它们每次都会返回，这样会大大降低处理程序检索自己关心的就绪文件描述符的效率，而边缘触发，则不会充斥大量你不关心的就绪文件描述符，从而性能差异，高下立见

阻塞IO：当你去读一个阻塞的文件描述符时，如果在该文件描述符上没有数据可读，那么它会一直阻塞(通俗一点就是一直卡在调用函数那里)，直到有数据可读。当你去写一个阻塞的文件描述符时，如果在该文件描述符上没有空间(通常是缓冲区)可写，那么它会一直阻塞，直到有空间可写。以上的读和写我们统一指在某个文件描述符进行的操作，不单单指真正的读数据，写数据，还包括接收连接accept()，发起连接connect()等操作...

非阻塞IO：当你去读写一个非阻塞的文件描述符时，不管可不可以读写，它都会立即返回，返回成功说明读写操作完成了，返回失败会设置相应errno状态码，根据这个errno可以进一步执行其他处理。它不会像阻塞IO那样，卡在那里不动！！！

**二.几种IO模型的触发方式**

 select(),poll()模型都是水平触发模式，信号驱动IO是边缘触发模式，epoll()模型即支持水平触发，也支持边缘触发，默认是水平触发。从表象看epoll的性能最好，但是在连接数少，并且连接都十分活跃的情况下，select和poll的性能可能比epoll好，毕竟epoll的通知机制需要很多回调函数来完成。

## 关于两种触发方式的难点整理

1. accept函数调用时使用哪种触发方式？   
   accept除了在某些特殊情况下(如连接已完成三次握手、进入就绪队列中等待被accept，然后客户端又断开)，一调用就直接返回，因此讨论起fd的阻塞非阻塞无意义，nigix的实现中，accept函数调用使用水平触发的socket fd，是出于对丢失连接的考虑（边缘触发时，accept只会执行一次接收一个连接，内核不会再去通知有连接就绪），当然，若使用while循环包裹住accept也是能够实现接受所有连接，稍显复杂，不如直接使用水平触发简单直观
2. 为何边缘触发不用处理EINTR   
   socket接口并不是被信号中断，只是调用了睡眠，发生信号睡眠会被唤醒通知进程，然后socket接口选择主动退出，这样做可以避免一直阻塞在那里，有退出的机会。非阻塞时不会调用睡眠，而ET使用的事非阻塞socket fd
3. 为何ET使用的是非阻塞fd   
   由于边缘触发的模式，每次epoll\_wait返回就绪的fd，必须读完读取缓冲区里的所有数据（直至接收数据返回EAGAIN），必须套上while循环，此时若使用阻塞的fd，当**读取完**缓冲区里的数据后，接受数据过程会阻塞，从而无法接受新的连接，而水平触发由于只接受一次数据（既然fd就绪可读，第一次调用并不会阻塞），然后若缓冲区仍有数据，等待下次epoll\_wait返回
4. 关于epoll\_wait的写就绪事件   
   **水平触发：**   
   若socket fd 可写，每次epoll\_wait都会返回该fd，会造成CPU占用100%的毛病，   
   解决方法
5. Epoll的触发时机
   1. 需要向socket写数据的时候才把socket加入epoll，等待可写事件。   
      接受到可写事件后，调用write或者send发送数据。   
      当所有数据都写完后，把socket移出epoll。
   2. 开始不把socket加入epoll，需要向socket写数据的时候，直接调用write或者send发送数据。如果返回 EAGAIN，把socket加入epoll，在epoll的驱动下写数据，全部数据发送完毕后，再移出epoll

后一种方式在send数据较少时，能够减少epoll\_wait的调用，性能有所提升

**边缘触发：**   
只有在socket fd由不可写到可写状态时，epoll\_wait才回返回该socket fd,因此，需要向socket写数据的时候，直接循环调用write或者send，直至finish或者返回EAGAIN，若返回EAGAIN则注册该socket fd的EPOLLOUT事件，等待下次epoll\_wait触发发送数据，发送完成后再注册该socket fd的EPOLLIN事件，即ATM模式

EPOLL事件的触发时机

* 1. EPOLLIN   
     ET模式：每次EPOLL\_CTL\_ADD或EPOLL\_CTL\_MOD时，如果加入前，就是可读状态，那么加入后会触发1次 ，不管sock缓冲是否读完，只要对方有send或connect或close或强退，就会触发EPOLLIN（ET/LT是针对一次socket fd就绪的，即一次fd就绪后有数据没读完/没写完，是否还会通知，所以该连接新的数据到来，该事件会触发）   
     LT模式：只要socket可读，就会一直触发EPOLLIN
  2. EPOLLOUT   
     ET模式： 每次EPOLL\_CTL\_ADD或EPOLL\_CTL\_MOD时，如果加入前，就是可写状态，那么加入后会触发1次 ，**如果EPOLLOUT与EPOLLIN一起注册，不管sock发送缓冲是否从满变不满，只要socket发送是不满的，那么每次EPOLLIN触发时，都会触发EPOLLOUT，即获取到不被期望的写事件，这也是为什么要使用ATM模式的原因**   
     LT模式：只要socket可写，就会一直触发EPOLLOUT
  3. **简单来说，ET模式下，只要监听了EPOLLIN和EPOLLOUT，socket的每次动作（包括close与不close直接强退），都会触发1次， 与读写缓冲的状态无关。**

一道腾讯后台开发的面试题：   
使用Linuxepoll模型，水平触发模式；当socket可写时，会不停的触发socket可写的事件，如何处理？   
解答：正如我们上面说的，LT模式下不需要读写的文件描述符仍会不停地返回就绪，这样就会影响我们监测需要关心的文件描述符的效率。   
所以这题的解决方法就是：平时不要把该描述符放进eventpoll结构体中，当需要写该fd的时候，调用epoll\_ctl把fd加入eventpoll里监听，可写的时候就往里写，写完再次调用epoll\_ctl把fd移出eventpoll，这种方法在发送很少数据的时候仍要执行两次epoll\_ctl操作，有一定的操作代价   
改进一下就是：平时不要把该描述符放进eventpoll结构体中，需要写的时候调用write或者send写数据，如果返回值是EAGAIN（写缓冲区满了），那么这时候才执行第一种方法的步骤。