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，

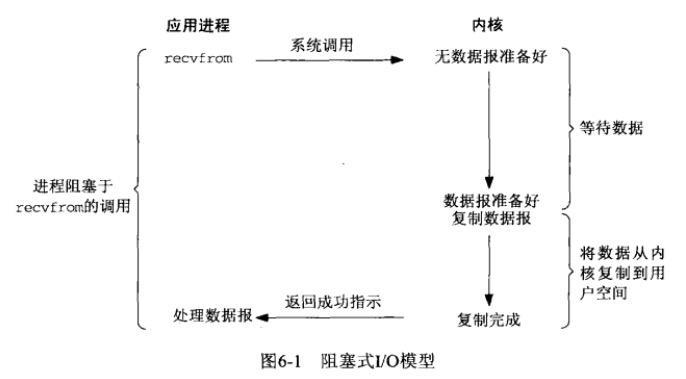
* 第一步通常涉及等待网络上的数据分组到达，然后被复制到内核的某个缓冲区。
* 第二步把数据从内核缓冲区复制到应用进程缓冲区。

### **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

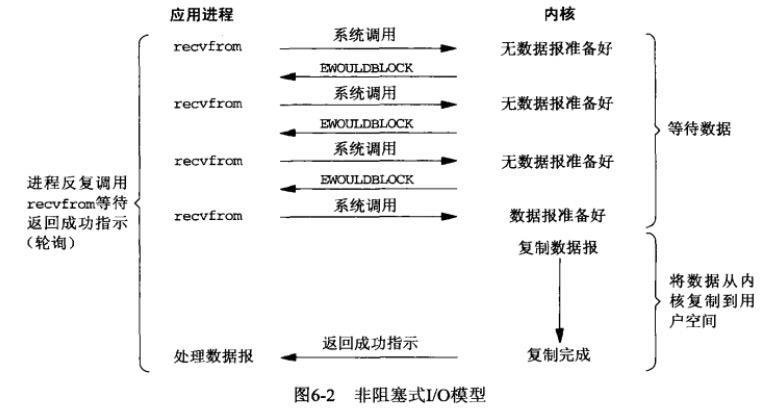
这就好比我们去钓鱼，抛竿之后就一直在岸边等，直到等待鱼上钩。然后再一次抛竿，等待下一条鱼上钩，等待的时候，什么事情也不做，大概会胡思乱想吧。

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

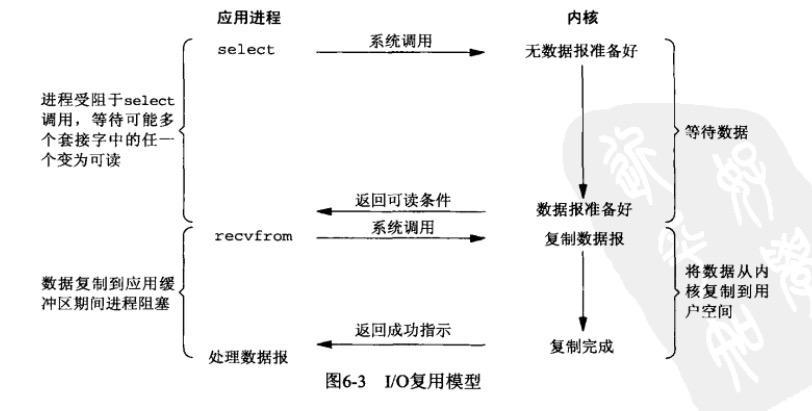
我们再用钓鱼的方式来类别，当我们抛竿入水之后，就看下鱼漂是否有动静，如果没有鱼上钩，就去干点别的事情，比如再挖几条蚯蚓。然后不久又来看看鱼漂是否有鱼上钩。这样往返的检查又离开，直到鱼上钩，再进行处理。

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

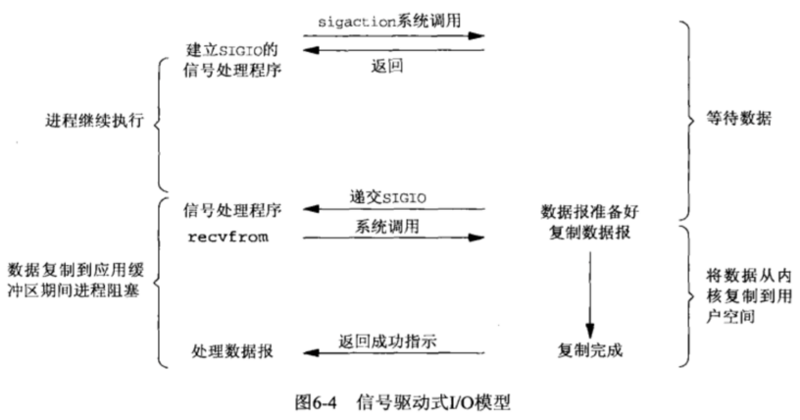
**对于多路复用，也就是轮询多个socket。**钓鱼的时候，我们雇了一个帮手，他可以同时抛下多个钓鱼竿，任何一杆的鱼一上钩，他就会拉杆。他只负责帮我们钓鱼，并不会帮我们处理，所以我们还得在一帮等着，等他把收杆。我们再处理鱼。多路复用既然可以处理多个I/O，也就带来了新的问题，多个I/O之间的顺序变得不确定了，当然也可以针对不同的编号。

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

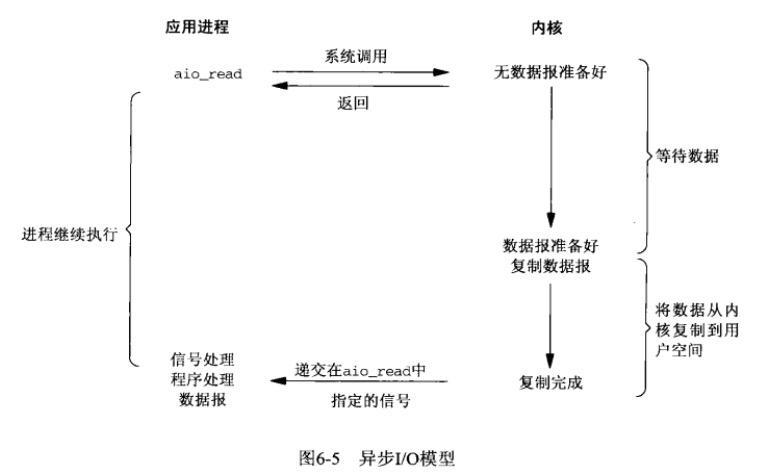


信号驱动模型如上图所示. **当文件描述符就绪时, 我们可以让内核以信号的方式通知我们.**我们首先需要开启套接字的信号驱动式 IO 功能, 并通过 sigaction 系统调用安装一个信号处理函数. sigaction 系统调用是异步的, 它会立即返回. 当有数据时, 内核会给此进程发送一个 SIGIO 信号, 进而我们的信号处理函数就会被执行, 我们就可以在这个函数中调用 recvfrom 读取数据.

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

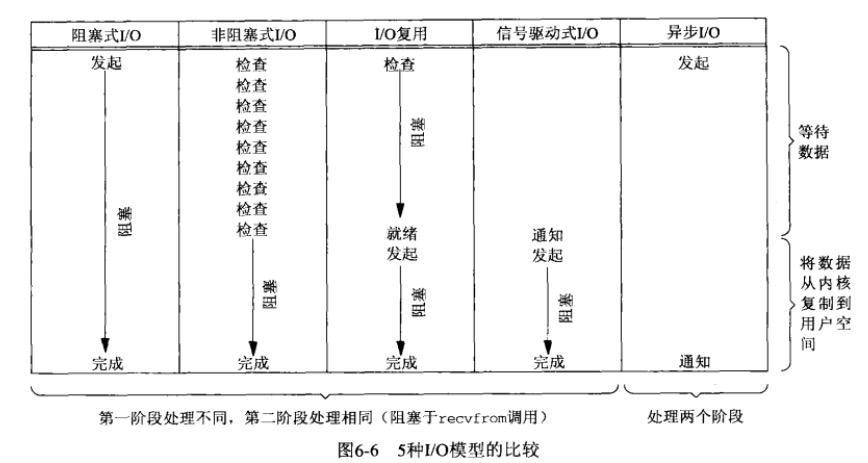
比之前的钓鱼方式不一样，这一次我们雇了一个钓鱼高手。他不仅会钓鱼，还会在鱼上钩之后给我们发短信，通知我们鱼已经准备好了。我们只要委托他去抛竿，然后就能跑去干别的事情了，直到他的短信。我们再回来处理已经上岸的鱼。

### **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。



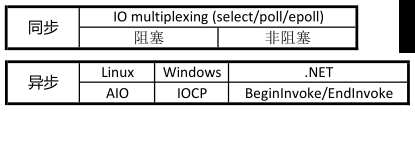
1. Png

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.

# ****select睡眠和唤醒过程****

select巧妙的利用等待队列机制让用户进程适当在没有资源可读/写时睡眠，有资源可读/写时唤醒。

## ****select睡眠过程****

　　select会循环遍历它所监测的fd\_　set内的所有文件描述符对应的驱动程序的poll函数。   
　　驱动程序提供的poll函数首先会将调用select的用户进程插入到该设备驱动对应资源的等待队列(如读/写等待队列)，然后返回一个bitmask告诉select当前资源哪些可用。　   
　　当select循环遍历完所有fd\_set内指定的文件描述符对应的poll函数后，如果没有一个资源可用(即没有一个文件可供操作)，则select让该进程睡眠，一直等到有资源可用为止，进程被唤醒(或者timeout)继续往下执行。

## ****select唤醒过程****

　　唤醒该进程的过程通常是在所监测文件的设备驱动内实现的。   
　　驱动程序维护了针对自身资源读写的等待队列。当设备驱动发现自身资源变为可读写并且有进程睡眠在该资源的等待队列上时，就会唤醒这个资源等待队列上的进程。

# ****select的缺点****

　　　1.每次调用select，都需要把fd集合从用户态拷贝到内核态，这个开销在fd很多时会很大   
　　2.同时每次调用select都需要在内核遍历传递进来的所有fd，这个开销在fd很多时也很大   
　　3.select支持的文件描述符数量太小了，默认是1024

# ****poll****

　　poll的机制与select类似，与select在本质上没有多大差别，管理多个描述符也是进行轮询，根据描述符的状态进行处理，**但是poll没有最大文件描述符数量的限制。**   
　　poll和select同样存在一个缺点就是，包含大量文件描述符的数组被整体复制于用户态和内核的地址空间之间，而不论这些文件描述符是否就绪，它的开销随着文件描述符数量的增加而线性增大。

# ****总结****

　　poll本质上和select没有区别，它将用户传入的数组拷贝到内核空间，然后查询每个fd对应的设备状态，如果设备就绪则在设备等待队列中加入一项并继续遍历，如果遍历完所有fd后没有发现就绪设备，则挂起当前进程，直到设备就绪或者主动超时，被唤醒后它又要再次遍历fd。   
　　这个过程经历了多次无谓的遍历。   
　　poll还有一个特点是“水平触发”，如果报告了fd后，没有被处理，那么下次poll时会再次报告该fd。   
　　poll与select的不同，通过一个pollfd数组向内核传递需要关注的事件，故没有描述符个数的限制，pollfd中的events字段和revents分别用于标示关注的事件和发生的事件，故pollfd数组只需要被初始化一次   
　　poll的实现机制与select类似，其对应内核中的sys\_poll，只不过poll向内核传递pollfd数组，然后对pollfd中的每个描述符进行poll，相比处理fdset来说，poll效率更高。**poll返回后，需要对pollfd中的每个元素检查其revents值，来得指事件是否发生。**

# ****优点**** 1）poll() 不要求开发者计算最大文件描述符加一的大小。  2）poll() 在应付大数目的文件描述符的时候速度更快，相比于select。  3）它没有最大连接数的限制，原因是它是基于链表来存储的。

# ****缺点**** 1）大量的fd的数组被整体复制于用户态和内核地址空间之间，而不管这样的复制是不是有意义。  2）与select一样，poll返回后，需要轮询pollfd来获取就绪的描述符

# ****epoll****

　　 在**[Linux](http://lib.csdn.net/base/linux" \o "Linux知识库" \t "http://blog.csdn.net/lixungogogo/article/details/_blank)**的网络编程中，很长的时间都在使用select来做事件触发。在**[linux](http://lib.csdn.net/base/linux" \o "Linux知识库" \t "http://blog.csdn.net/lixungogogo/article/details/_blank)**新的内核中，有了一种替换它的机制，就是epoll。   
　　相比于select，**epoll最大的好处在于它不会随着监听fd数目的增长而降低效率。**因为在内核中的select实现中，它是采用轮询来处理的，轮询的fd数目越多，自然耗时越多。   
　　相对于select和poll来说，epoll更加灵活，**没有描述符限制**。   
　　epoll使用一个文件描述符管理多个描述符，将用户关心的文件描述符的事件存放到内核的一个**事件表**中，**这样在用户空间和内核空间的copy只需一次。**

# ****epoll接口****

　　epoll操作过程需要三个接口，分别如下：

#include <sys/epoll.h>

int epoll\_create(int size);

int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event);

int epoll\_wait(int epfd, struct epoll\_event \* events, int maxevents, int timeout);

　　首先要调用**epoll\_create建立**一个epoll对象。参数size是内核保证能够正确处理的最大句柄数，多于这个最大数时内核可不保证效果。   
　　**epoll\_ctl**可以**操作**上面建立的epoll，例如，将刚建立的socket加入到epoll中让其监控，或者把 epoll正在监控的某个socket句柄移出epoll，不再监控它等等。   
　　**epoll\_wait**在调用时，在给定的timeout时间内，当在监控的所有句柄中有事件发生时，就返回用户态的进程。epoll\_wait范围之后应该是一个循环，遍历所有的事件。   
　　我们调用epoll\_ wait时就相当于以往调用select/poll，但是这时却不用传递socket句柄给内核，因为内核已经在epoll\_ctl中拿到了要监控的句柄列表。

# ****epoll实现机制****

epoll在被内核初始化时（**[操作系统](http://lib.csdn.net/base/operatingsystem" \o "操作系统知识库" \t "http://blog.csdn.net/lixungogogo/article/details/_blank)**启动），同时会开辟出epoll自己的**内核高速cache区**，用于安置每一个我们想监控的socket。   
　　**这些socket会以红黑树的形式保存在内核cache里，以支持快速的查找、插入、删除。**   
　　这个内核高速cache区，就是**建立连续的物理内存页**，然后在之上**建立slab层**。   
　　**简单的说，就是物理上分配好你想要的size的内存对象，每次使用时都是使用空闲的已分配好的对象。**   
　　epoll的高效就在于，当我们调用epoll\_ ctl往里塞入百万个句柄时，epoll\_ wait仍然可以飞快的返回，并有效的将发生事件的句柄给我们用户。   
　　这是由于我们在调用epoll\_ create时，内核除了帮我们在epoll文件系统里建了个file结点，在内核cache里建了个红黑树用于存储以后epoll\_ ctl传来的socket外，**还会再建立一个list链表，用于存储准备就绪的事件.**   
　　当epoll\_ wait调用时，仅仅观察这个list链表里有没有数据即可。有数据就返回，没有数据就sleep，等到timeout时间到后即使链表没数据也返回。所以，epoll\_wait非常高效。   
　　而且，通常情况下即使我们要监控百万计的句柄，大多一次也只返回很少量的准备就绪句柄而已，所以，epoll\_wait仅需要从内核态copy少量的句柄到用户态而已。   
　　**那么，这个准备就绪list链表是怎么维护的呢？**   
　　当我们执行epoll\_ctl时，除了把socket放到epoll文件系统里file对象对应的红黑树上之外，****还会给内核中断处理程序注册一个回调函数**，告诉内核，如果这个句柄的中断到了，就把它放到准备就绪list链表里。**　　所以，当一个socket上有数据到了，**内核在把网卡上的数据copy到内核中后就来把socket插入到准备就绪链表里了**。   
　　**如此，一颗红黑树，一张准备就绪句柄链表，少量的内核cache，就帮我们解决了大并发下的socket处理问题。**   
　　**执行epoll\_ create时，创建了红黑树和就绪链表，执行epoll\_ ctl时，如果增加socket句柄，则检查在红黑树中是否存在，存在立即返回，不存在则添加到树干上，然后向内核注册回调函数，用于当中断事件到来时向准备就绪链表中插入数据。执行epoll\_wait时立刻返回准备就绪链表里的数据即可。**

# ****工作模式****

　　epoll对文件描述符的操作有两种模式：LT（level trigger）和ET（edge trigger）。LT模式是默认模式，LT模式与ET模式的区别如下：

## ****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。

　　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。**   
　　那么ET模式是怎么做到的呢？

# ****ET模式的原理****

　　当一个socket句柄上有事件时，内核会把该句柄插入上面所说的准备就绪list链表，这时我们调用epoll\_ wait，会把准备就绪的socket拷贝到用户态内存，然后清空准备就绪list链表。   
　　最后，epoll\_ wait检查这些socket，如果不是ET模式（就是LT模式的句柄了），并且这些socket上确实有未处理的事件时，又把该句柄放回到刚刚清空的准备就绪链表了。   
　　所以，非ET的句柄，只要它上面还有事件，epoll\_ wait每次都会返回。而ET模式的句柄，除非有新中断到，即使socket上的事件没有处理完，也是不会次次从epoll\_wait返回的。

# ****优点****

**1）支持一个进程打开大数目的socket描述符(FD)**

　　select最不能忍受的是一个进程所打开的FD是有一定限制的，由FD\_SETSIZE设置，默认值是1024/2048。对于那些需要支持的上万连接数目的IM服务器来说显然太少了。这时候你一是可以选择修改这个宏然后重新编译内核。不过 epoll则没有这个限制，它所支持的FD上限是最大可以打开文件的数目，这个数字一般远大于2048,举个例子,在1GB内存的机器上大约是10万左右，具体数目可以cat /proc/sys/fs/file-max察看,一般来说这个数目和系统内存关系很大。

**2）IO效率不随FD数目增加而线性下降**

　　传统的select/poll另一个致命弱点就是当你拥有一个很大的socket集合，不过由于网络延时，任一时间只有部分的socket是”活跃”的，但是select/poll每次调用都会线性扫描全部的集合，导致效率呈现线性下降。但是epoll不存在这个问题，它只会对”活跃”的socket进行操作—这是因为在内核实现中epoll是根据每个fd上面的callback函数实现的。那么，只有”活跃”的socket才会主动的去调用 callback函数，其他idle状态socket则不会，在这点上，epoll实现了一个”伪”AIO，因为这时候推动力在Linux内核。   
**3）使用mmap加速内核与用户空间的消息传递。**

　　这点实际上涉及到epoll的具体实现了。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拷贝就很重要，在这点上，epoll是通过内核与用户空间mmap同一块内存实现的。

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，期间用户进程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，期间用户进程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(当前进程)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。

Epoll两种工作模式：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，它还会通知你在上次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。