**同步：**所谓同步，就是在发出一个功能调用时，在没有得到结果之前，该调用就不返回。按照这个定义，其实绝大多数函数都是同步调用（例如sin, isdigit等）。但是一般而言，我们在说同步、异步的时候，特指那些需要其他部件协作或者需要一定时间完成的任务。最常见的例子就是 SendMessage。该函数发送一个消息给某个窗口，在对方处理完消息之前，这个函数不返回。当对方处理完毕以后，该函数才把消息处理函数所返回的 LRESULT值返回给调用者。

**异步：**异步的概念和同步相对。当一个异步功能调用发出后，调用者不能立刻得到结果。当该异步功能完成后，通过状态、通知或回调来通知调用者。以 CAsycSocket类为例（注意，CSocket从CAsyncSocket派生，但是其功能已经由异步转化为同步），当一个客户端通过调用 Connect函数发出一个连接请求后，调用者线程立刻可以朝下运行。当连接真正建立起来以后，socket底层会发送一个消息通知该对象。这里提到异步功能跟调用者通过三种途径返回结果：状态、通知和回调，可以使用哪一种依赖于异步功能的实现，除非异步功能提供多种选择，否则不受调用者控制。如果异步功能用状态来通知，那么调用者就需要每隔一定时间检查一次，效率就很低（有些初学多线程编程的人，总喜欢用一个循环去检查某个变量的值，这其实是一 种很严重的错误）。如果是使用通知的方式，效率则很高，因为异步功能几乎不需要做额外的操作。至于回调函数，其实和通知没太多区别。

**阻塞：**阻塞调用是指调用结果返回之前，当前线程会被挂起。函数只有在 得到结果之后才会将阻塞的线程激活。有人也许会把阻塞调用和同步调用等同起来，实际上他是不同的。对于同步调用来说，很多时候当前线程还是激活的，只是从逻辑上当前函数没有返回而已。例如，我们在 CSocket中调用Receive函数，如果缓冲区中没有数据，这个函数就会一直等待，直到有数据才返回。而此时，当前线程还会继续处理各种各样的消 息。socket接收数据的另外一个函数 recv则是一个阻塞调用的例子。当socket工作在阻塞模式的时候，如果没有数据的情况下调用该函数，则当前线程就会被挂起，直到有数据为止。

**非阻塞：**非阻塞和阻塞的概念相对应，指在不能立刻得到结果之前也会立刻返回，同时该函数不会阻塞当前线程。

**小结：同步就是当一个进程发起一个函数（任务）调用的时候，一直等待直到函数（任务）完成，而进程继续处于激活状态。而异步情况下是当一个进程发起一个函数（任务）调用的时候，不会等函数返回，而是继续往下执行当，函数返回的时候通过状态、通知、事件等方式通知进程任务完成。**

**阻塞是当请求不能满足的时候就将进程挂起，而非阻塞则不会阻塞当前进程，即阻塞与非阻塞针对的是进程或线程而同步与异步所针对的是功能函数。**

有了上面的理解，那么我们在分别看下五种IO的模型：阻塞IO、非阻塞IO、多路复用IO、信号驱动IO和异步IO。

**阻塞IO：**在这个模型中，应用程序（application）为了执行这个read操作，会调用相应的一个system call，将系统控制权交给kernel，然后就进行等待（这其实就是被阻塞了）。kernel开始执行这个system call，执行完毕后会向应用程序返回响应，应用程序得到响应后，就不再阻塞，并进行后面的工作。

**非阻塞IO：**在[**Linux**](http://lib.csdn.net/base/linux)下，应用程序可以通过设置文件描述符的属性O\_NONBLOCK，I/O操作可以立即返回，但是并不保证I/O操作成功。也就是说，当应用程序设置了O\_NONBLOCK之后，执行write操作，调用相应的system call，这个system call会从内核中立即返回。但是在这个返回的时间点，数据可能还没有被真正的写入到指定的地方。也就是说，kernel只是很快的返回了这个 system call（只有立马返回，应用程序才不会被这个IO操作blocking），但是这个system call具体要执行的事情（写数据）可能并没有完成。而对于应用程序，虽然这个IO操作很快就返回了，但是它并不知道这个IO操作是否真的成功了，为了知道IO操作是否成功，一般有两种策略：一是需要应用程序主动地循环地去问kernel(这种方法就是同步非阻塞IO)；二是采用I/O通知机制，比如：IO多路复用(这种方法属于异步阻塞IO)或信号驱动IO(这种方法属于异步非阻塞IO)。

**IO多路复用(异步阻塞IO)：**和之前一样，应用程序要执行read操作，因此调用一个system call，这个system call被传递给了kernel。但在应用程序这边，它调用system call之后，并不等待kernel的返回结果而是立即返回，虽然立即返回的调用函数是一个异步的方式，但应用程序会被像select()、poll和epoll等具有复用多个文件描述符的函数阻塞住，一直等到这个system call有结果返回了，再通知应用程序。也就是说，“在这种模型中，IO函数是非阻塞的，使用阻塞 select、poll、epoll系统调用来确定一个 或多个I/O 描述符何时能操作。”所以，从IO操作的实际效果来看，异步阻塞IO和第一种同步阻塞IO是一样的，应用程序都是一直等到IO操作成功之后（数据已经被写入或者读取），才开始进行下面的工作。不同点在于异步阻塞IO用一个select函数可以为多个描述符提供通知，提高了并发性。举个例子：假如有一万个并发的read请求，但是网络上仍然没有数据，此时这一万个read会同时各自阻塞，现在用select、poll、epoll这样的函数来专门负责阻塞同时监听这一万个请求的状态，一旦有数据到达了就负责通知，这样就将之前一万个的各自为战的等待与阻塞转为一个专门的函数来负责与管理。与此同时，异步阻塞IO和第二种同步非阻塞IO的区别在于：同步非阻塞IO是需要应用程序主动地循环去询问是否有操作数据可操作，而异步阻塞IO是通过像select和poll等这样的IO多路复用函数来同时检测多个事件句柄来告知应用程序是否可以有数据操作。

**信号驱动IO(异步非阻塞IO)：**应用程序提交read请求的system call，然后，kernel开始处理相应的IO操作，而同时，应用程序并不等kernel返回响应，就会开始执行其他的处理操作（应用程序没有被IO操作所阻塞）。当kernel执行完毕，返回read的响应，就会产生一个信号或执行一个基于线程的回调函数来完成这次 I/O 处理过程。

        从理论上说，阻塞IO、IO复用和信号驱动的IO都是同步IO模型。因为在这三种模型中，IO的读写操作都是在IO事件发生之后由应用程序来完成。而POSIX规范所定义的异步IO模型则不同。对异步IO而言，用户可以直接对IO执行读写操作，这些操作告诉内核用户读写缓冲区的位置，以及IO操作完成后内核通知应用程序的方式。异步IO读写操作总是立即返回，而不论IO是否阻塞的，因为真主的读写操作已经由内核接管。也就是说，同步IO模型要求用户代码自行执行IO操作(将数据从内核缓冲区读入用户缓冲区，或将数据从用户缓冲区写入内核缓冲区)，而异步IO机制则是由内核来执行IO操作(数据在内核缓冲区和用户缓冲区之间的移动是由内核在后台完成的)。你可以这样认为，同步IO向应用程序通知的是IO就绪事件，而异步IO向应用程序通知的是IO完成事件。linux环境下，aio.h头文件中定义的函数提供了对异步IO的支持。

**异步IO：**异步IO与上面的异步概念是一样的， 当一个异步过程调用发出后，调用者不能立刻得到结果，实际处理这个调用的函数在完成后，通过状态、通知和回调来通知调用者的输入输出操作。异步IO的工作机制是：告知内核启动某个操作，并让内核在整个操作完成后通知我们，这种模型与信号驱动的IO区别在于，信号驱动IO是由内核通知我们何时可以启动一个IO操作，这个IO操作由用户自定义的信号函数来实现，而异步IO模型是由内核告知我们IO操作何时完成。为了实现异步IO，专门定义了一套以aio开头的API，如：aio\_read.

小结：前四种模型--阻塞IO、非阻塞IO、多路复用IO和信号驱动IO都属于同步模式，因为其中真正的IO操作(函数)都将会阻塞进程，只有异步IO模型真正实现了IO操作的异步性。

IO复用：为了解释这个名词，首先来理解下复用这个概念，复用也就是共用的意思，这样理解还是有些抽象，为此，咱们来理解下复用在通信领域的使用，在通信领域中为了充分利用网络连接的物理介质，往往在同一条网络链路上采用时分复用或频分复用的技术使其在同一链路上传输多路信号，到这里我们就基本上理解了复用的含义，即公用某个“介质”来尽可能多的做同一类(性质)的事，那IO复用的“介质”是什么呢？为此我们首先来看看服务器编程的模型，客户端发来的请求服务端会产生一个进程来对其进行服务，每当来一个客户请求就产生一个进程来服务，然而进程不可能无限制的产生，因此为了解决大量客户端访问的问题，引入了IO复用技术，即：一个进程可以同时对多个客户请求进行服务。也就是说IO复用的“介质”是进程(准确的说复用的是select和poll，因为进程也是靠调用select和poll来实现的)，复用一个进程(select和poll)来对多个IO进行服务，虽然客户端发来的IO是并发的但是IO所需的读写数据多数情况下是没有准备好的，因此就可以利用一个函数(select和poll)来监听IO所需的这些数据的状态，一旦IO有数据可以进行读写了，进程就来对这样的IO进行服务。

理解完IO复用后，我们在来看下实现IO复用中的三个API(select、poll和epoll)的区别和联系

select，poll，epoll都是IO多路复用的机制，I/O多路复用就是通过一种机制，可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知应用程序进行相应的读写操作。但select，poll，epoll本质上都是同步I/O，因为他们都需要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是阻塞的，而异步I/O则无需自己负责进行读写，异步I/O的实现会负责把数据从内核拷贝到用户空间。三者的原型如下所示：

int select(int nfds, fd\_set \*readfds, fd\_set \*writefds, fd\_set \*exceptfds, struct timeval \*timeout);

int poll(struct pollfd \*fds, nfds\_t nfds, int timeout);

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

 1.select的第一个参数nfds为fdset集合中最大描述符值加1，fdset是一个位数组，其大小限制为\_\_FD\_SETSIZE（1024），位数组的每一位代表其对应的描述符是否需要被检查。第二三四参数表示需要关注读、写、错误事件的文件描述符位数组，这些参数既是输入参数也是输出参数，可能会被内核修改用于标示哪些描述符上发生了关注的事件，所以每次调用select前都需要重新初始化fdset。timeout参数为超时时间，该结构会被内核修改，其值为超时剩余的时间。

 select的调用步骤如下：

（1）使用copy\_from\_user从用户空间拷贝fdset到内核空间

（2）注册回调函数\_\_pollwait

（3）遍历所有fd，调用其对应的poll方法（对于socket，这个poll方法是sock\_poll，sock\_poll根据情况会调用到tcp\_poll,udp\_poll或者datagram\_poll）

（4）以tcp\_poll为例，其核心实现就是\_\_pollwait，也就是上面注册的回调函数。

（5）\_\_pollwait的主要工作就是把current（当前进程）挂到设备的等待队列中，不同的设备有不同的等待队列，对于tcp\_poll 来说，其等待队列是sk->sk\_sleep（注意把进程挂到等待队列中并不代表进程已经睡眠了）。在设备收到一条消息（网络设备）或填写完文件数 据（磁盘设备）后，会唤醒设备等待队列上睡眠的进程，这时current便被唤醒了。

（6）poll方法返回时会返回一个描述读写操作是否就绪的mask掩码，根据这个mask掩码给fd\_set赋值。

（7）如果遍历完所有的fd，还没有返回一个可读写的mask掩码，则会调用schedule\_timeout是调用select的进程（也就是 current）进入睡眠。当设备驱动发生自身资源可读写后，会唤醒其等待队列上睡眠的进程。如果超过一定的超时时间（schedule\_timeout 指定），还是没人唤醒，则调用select的进程会重新被唤醒获得CPU，进而重新遍历fd，判断有没有就绪的fd。

（8）把fd\_set从内核空间拷贝到用户空间。

总结下select的几大缺点：

（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

2．  poll与select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。

3．直到Linux2.6才出现了由内核直接支持的实现方法，那就是epoll，被公认为Linux2.6下性能最好的多路I/O就绪通知方法。epoll可以同时支持水平触发和边缘触发（Edge Triggered，只告诉进程哪些文件描述符刚刚变为就绪状态，它只说一遍，如果我们没有采取行动，那么它将不会再次告知，这种方式称为边缘触发），理论上边缘触发的性能要更高一些，但是代码实现相当复杂。epoll同样只告知那些就绪的文件描述符，而且当我们调用epoll\_wait()获得就绪文件描述符时，返回的不是实际的描述符，而是一个代表就绪描述符数量的值，你只需要去epoll指定的一个数组中依次取得相应数量的文件描述符即可，这里也使用了内存映射（mmap）技术，这样便彻底省掉了这些文件描述符在系统调用时复制的开销。另一个本质的改进在于epoll采用基于事件的就绪通知方式。在select/poll中，进程只有在调用一定的方法后，内核才对所有监视的文件描述符进行扫描，而epoll事先通过epoll\_ctl()来注册一个文件描述符，一旦基于某个文件描述符就绪时，内核会采用类似callback的回调机制，迅速激活这个文件描述符，当进程调用epoll\_wait()时便得到通知。

epoll既然是对select和poll的改进，就应该能避免上述的三个缺点。那epoll都是怎么解决的呢？在此之前，我们先看一下epoll 和select和poll的调用接口上的不同，select和poll都只提供了一个函数——select或者poll函数。而epoll提供了三个函 数，epoll\_create,epoll\_ctl和epoll\_wait，epoll\_create是创建一个epoll句柄；epoll\_ctl是注 册要监听的事件类型；epoll\_wait则是等待事件的产生。

　　对于第一个缺点，epoll的解决方案在epoll\_ctl函数中。每次注册新的事件到epoll句柄中时（在epoll\_ctl中指定 EPOLL\_CTL\_ADD），会把所有的fd拷贝进内核，而不是在epoll\_wait的时候重复拷贝。epoll保证了每个fd在整个过程中只会拷贝 一次。

　　对于第二个缺点，epoll的解决方案不像select或poll一样每次都把current轮流加入fd对应的设备等待队列中，而只在 epoll\_ctl时把current挂一遍（这一遍必不可少）并为每个fd指定一个回调函数，当设备就绪，唤醒等待队列上的等待者时，就会调用这个回调 函数，而这个回调函数会把就绪的fd加入一个就绪链表）。epoll\_wait的工作实际上就是在这个就绪链表中查看有没有就绪的fd（利用 schedule\_timeout()实现睡一会，判断一会的效果，和select实现中的第7步是类似的）。

　　对于第三个缺点，epoll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。

关于epoll的一篇非常好的文章分享给大家([**快课网的高并发网络编程之epoll详解**](http://www.cricode.com/3499.html))以及([**通过完整示例来理解如何使用 epoll**](http://blog.jobbole.com/93566/))

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。