<https://blog.csdn.net/weiyuefei/article/details/51991397>

select，poll，epoll都是IO多路复用的机制。I/O多路复用就通过一种机制，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。关于这三种IO多路复用的用法，前面三篇总结写的很清楚，并用服务器回射echo程序进行了测试。连接如下所示：

select：[http://www.cnblogs.com/Anker/archive/2013/08/14/3258674.html](http://www.cnblogs.com/Anker/archive/2013/08/14/3258674.html" \t "_blank)

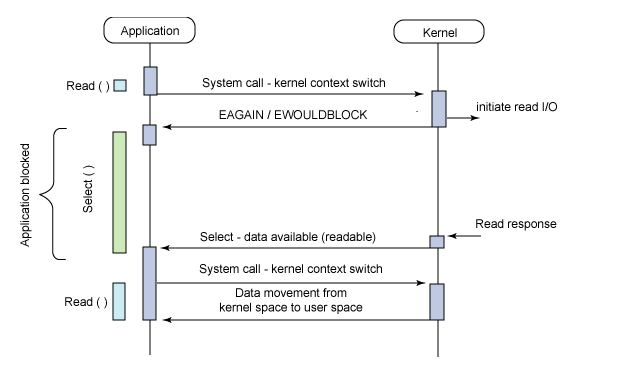
poll：[http://www.cnblogs.com/Anker/archive/2013/08/15/3261006.html](http://www.cnblogs.com/Anker/archive/2013/08/15/3261006.html" \t "_blank)

epoll：[http://www.cnblogs.com/Anker/archive/2013/08/17/3263780.html](http://www.cnblogs.com/Anker/archive/2013/08/17/3263780.html" \t "_blank)

　　今天对这三种IO多路复用进行对比，参考网上和书上面的资料，整理如下：

1、select实现

select的调用过程如下所示：



（1）使用copy\_from\_user从用户空间拷贝fd\_set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：

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实现

　　poll的实现和select非常相似，只是描述fd集合的方式不同，poll使用pollfd结构而不是select的fd\_set结构，其他的都差不多。

关于select和poll的实现分析，可以参考下面几篇博文：

<http://blog.csdn.net/lizhiguo0532/article/details/6568964#comments>

<http://blog.csdn.net/lizhiguo0532/article/details/6568968>

<http://blog.csdn.net/lizhiguo0532/article/details/6568969>

<http://www.ibm.com/developerworks/cn/linux/l-cn-edntwk/index.html?ca=drs->

<http://linux.chinaunix.net/techdoc/net/2009/05/03/1109887.shtml>

3、epoll

　　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。

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。

参考资料：

<http://www.cnblogs.com/apprentice89/archive/2013/05/09/3070051.html>

<http://www.linuxidc.com/Linux/2012-05/59873p3.htm>

<http://xingyunbaijunwei.blog.163.com/blog/static/76538067201241685556302/>

<http://blog.csdn.net/kkxgx/article/details/7717125>

<https://banu.com/blog/2/how-to-use-epoll-a-complete-example-in-c/epoll-example.c>