**select, poll和epoll的区别**

select()系统调用提供一个机制来实现同步多元I/O：

|  |
| --- |
| #include <sys/time.h> #include <sys/types.h> #include <unistd.h>  int select (int n, fd\_set \*readfds, fd\_set \*writefds, fd\_set \*exceptfds, struct timeval \*timeout);  FD\_CLR(int fd, fd\_set \*set); FD\_ISSET(int fd, fd\_set \*set); FD\_SET(int fd, fd\_set \*set); FD\_ZERO(fd\_set \*set); |

调用select()将阻塞，直到指定的文件描述符准备好执行I/O，或者可选参数timeout指定的时间已经过去。  
监视的文件描述符分为三类set，每一种对应等待不同的事件。readfds中列出的文件描述符被监视是否有数据可供读取（如果读取操作完成则不会阻塞）。writefds中列出的文件描述符则被监视是否写入操作完成而不阻塞。最后，exceptfds中列出的文件描述符则被监视是否发生异常，或者无法控制的数据是否可用（这些状态仅仅应用于套接字）。这三类set可以是NULL，这种情况下select()不监视这一类事件。  
select()成功返回时，每组set都被修改以使它只包含准备好I/O的文件描述符。例如，假设有两个文件描述符，值分别是7和9，被放在readfds中。当select()返回时，如果7仍然在set中，则这个文件描述符已经准备好被读取而不会阻塞。如果9已经不在set中，则读取它将可能会阻塞（我说可能是因为数据可能正好在select返回后就可用，这种情况下，下一次调用select()将返回文件描述符准备好读取）。  
第一个参数n，等于所有set中最大的那个文件描述符的值加1。因此，select()的调用者负责检查哪个文件描述符拥有最大值，并且把这个值加1再传递给第一个参数。  
timeout参数是一个指向timeval结构体的指针，timeval定义如下：

|  |
| --- |
| #include <sys/time.h> struct timeval { long tv\_sec; /\* seconds \*/ long tv\_usec; /\* 10E-6 second \*/ }; |

如果这个参数不是NULL，则即使没有文件描述符准备好I/O，select()也会在经过tv\_sec秒和tv\_usec微秒后返回。当select()返回时，timeout参数的状态在不同的系统中是未定义的，因此每次调用select()之前必须重新初始化timeout和文件描述符set。实际上，当前版本的Linux会自动修改timeout参数，设置它的值为剩余时间。因此，如果timeout被设置为5秒，然后在文件描述符准备好之前经过了3秒，则这一次调用select()返回时tv\_sec将变为2。  
如果timeout中的两个值都设置为0，则调用select()将立即返回，报告调用时所有未决的事件，但不等待任何随后的事件。  
文件描述符set不会直接操作，一般使用几个助手宏来管理。这允许Unix系统以自己喜欢的方式来实现文件描述符set。但大多数系统都简单地实现set为位数组。FD\_ZERO移除指定set中的所有文件描述符。每一次调用select()之前都应该先调用它。  
fd\_set writefds;  
FD\_ZERO(&writefds);  
  
FD\_SET添加一个文件描述符到指定的set中，FD\_CLR则从指定的set中移除一个文件描述符：  
FD\_SET(fd, &writefds); /\* add 'fd' to the set \*/  
FD\_CLR(fd, &writefds); /\* oops, remove 'fd' from the set \*/  
  
设计良好的代码应该永远不使用FD\_CLR，而且实际情况中它也确实很少被使用。  
FD\_ISSET测试一个文件描述符是否指定set的一部分。如果文件描述符在set中则返回一个非0整数，不在则返回0。FD\_ISSET在调用select()返回之后使用，测试指定的文件描述符是否准备好相关动作：  
if (FD\_ISSET(fd, &readfds))  
/\* 'fd' is readable without blocking! \*/  
  
因为文件描述符set是静态创建的，它们对文件描述符的最大数目强加了一个限制，能够放进set中的最大文件描述符的值由FD\_SETSIZE指定。在Linux中，这个值是1024。本章后面我们还将看到这个限制的衍生物。  
返回值和错误代码  
select()成功时返回准备好I/O的文件描述符数目，包括所有三个set。如果提供了timeout，返回值可能是0；错误时返回-1，并且设置errno为下面几个值之一：  
EBADF  
给某个set提供了无效文件描述符。  
EINTR  
等待时捕获到信号，可以重新发起调用。  
EINVAL  
参数n为负数，或者指定的timeout非法。  
ENOMEM  
不够可用内存来完成请求。  
--------------------------------------------------------------------------------------------------------------

poll()系统调用是System V的多元I/O解决方案。它解决了select()的几个不足，尽管select()仍然经常使用（多数还是出于习惯，或者打着可移植的名义）：

|  |
| --- |
| #include <sys/poll.h> int poll (struct pollfd \*fds, unsigned int nfds, int timeout); |

和select()不一样，poll()没有使用低效的三个基于位的文件描述符set，而是采用了一个单独的结构体pollfd数组，由fds指针指向这个组。pollfd结构体定义如下：

|  |
| --- |
| #include <sys/poll.h>  struct pollfd { int fd; /\* file descriptor \*/ short events; /\* requested events to watch \*/ short revents; /\* returned events witnessed \*/ }; |

每一个pollfd结构体指定了一个被监视的文件描述符，可以传递多个结构体，指示poll()监视多个文件描述符。每个结构体的events域是监视该文件描述符的事件掩码，由用户来设置这个域。revents域是文件描述符的操作结果事件掩码。内核在调用返回时设置这个域。events域中请求的任何事件都可能在revents域中返回。合法的事件如下：  
POLLIN  
有数据可读。  
POLLRDNORM  
有普通数据可读。  
POLLRDBAND  
有优先数据可读。  
POLLPRI  
有紧迫数据可读。  
POLLOUT  
写数据不会导致阻塞。  
POLLWRNORM  
写普通数据不会导致阻塞。  
POLLWRBAND  
写优先数据不会导致阻塞。  
POLLMSG  
SIGPOLL消息可用。  
  
此外，revents域中还可能返回下列事件：  
POLLER  
指定的文件描述符发生错误。  
POLLHUP  
指定的文件描述符挂起事件。  
POLLNVAL  
指定的文件描述符非法。  
  
这些事件在events域中无意义，因为它们在合适的时候总是会从revents中返回。使用poll()和select()不一样，你不需要显式地请求异常情况报告。  
POLLIN | POLLPRI等价于select()的读事件，POLLOUT | POLLWRBAND等价于select()的写事件。POLLIN等价于POLLRDNORM | POLLRDBAND，而POLLOUT则等价于POLLWRNORM。  
例如，要同时监视一个文件描述符是否可读和可写，我们可以设置events为POLLIN | POLLOUT。在poll返回时，我们可以检查revents中的标志，对应于文件描述符请求的events结构体。如果POLLIN事件被设置，则文件描述符可以被读取而不阻塞。如果POLLOUT被设置，则文件描述符可以写入而不导致阻塞。这些标志并不是互斥的：它们可能被同时设置，表示这个文件描述符的读取和写入操作都会正常返回而不阻塞。  
timeout参数指定等待的毫秒数，无论I/O是否准备好，poll都会返回。timeout指定为负数值表示无限超时；timeout为0指示poll调用立即返回并列出准备好I/O的文件描述符，但并不等待其它的事件。这种情况下，poll()就像它的名字那样，一旦选举出来，立即返回。  
返回值和错误代码  
成功时，poll()返回结构体中revents域不为0的文件描述符个数；如果在超时前没有任何事件发生，poll()返回0；失败时，poll()返回-1，并设置errno为下列值之一：  
EBADF  
一个或多个结构体中指定的文件描述符无效。  
EFAULT  
fds指针指向的地址超出进程的地址空间。  
EINTR  
请求的事件之前产生一个信号，调用可以重新发起。  
EINVAL  
nfds参数超出PLIMIT\_NOFILE值。  
ENOMEM  
可用内存不足，无法完成请求。  
--------------------------------------------------------------------------------------------------------------  
以上内容来自《OReilly.Linux.System.Programming - Talking.Directly.to.the.Kernel.and.C.Library.2007》  
--------------------------------------------------------------------------------------------------------------

epoll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，不过资料也同时指出这样会带来网络效率的下降，二是可以选择多进程的解决方案(传统的 Apache方案)，不过虽然linux上面创建进程的代价比较小，但仍旧是不可忽视的，加上进程间数据同步远比不上线程间同步的高效，所以也不是一种完美的方案。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os内核。在一些 benchmark中，如果所有的socket基本上都是活跃的---比如一个高速LAN环境，epoll并不比select/poll有什么效率，相反，如果过多使用epoll\_ctl,效率相比还有稍微的下降。但是一旦使用idle connections模拟WAN环境,epoll的效率就远在select/poll之上了。

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是通过内核于用户空间mmap同一块内存实现的。而如果你想我一样从2.5内核就关注epoll的话，一定不会忘记手工 mmap这一步的。

4.内核微调  
    这一点其实不算epoll的优点了，而是整个linux平台的优点。也许你可以怀疑linux平台，但是你无法回避linux平台赋予你微调内核的能力。比如，内核TCP/IP协议栈使用内存池管理sk\_buff结构，那么可以在运行时期动态调整这个内存pool(skb\_head\_pool)的大小--- 通过echo XXXX>/proc/sys/net/core/hot\_list\_length完成。再比如listen函数的第2个参数(TCP完成3次握手的数据包队列长度)，也可以根据你平台内存大小动态调整。更甚至在一个数据包面数目巨大但同时每个数据包本身大小却很小的特殊系统上尝试最新的NAPI网卡驱动架构。