I/O多路复用

一、五种I/O模型

1、阻塞I/O模型

最流行的I/O模型是阻塞I/O模型，缺省情形下，所有套接口都是阻塞的。我们以数据报套接口为例来讲解此模型（我们使用UDP而不是TCP作为例子的原因在于就UDP而言，数据准备好读取的概念比较简单：要么整个数据报已经收到，要么还没有。然而对于TCP来说，诸如套接口低潮标记等额外变量开始活动，导致这个概念变得复杂）。

进程调用recvfrom，其系统调用直到数据报到达且被拷贝到应用进程的缓冲区中或者发生错误才返回，期间一直在等待。我们就说进程在从调用recvfrom开始到它返回的整段时间内是被阻塞的。

2、非阻塞I/O模型

进程把一个套接口设置成非阻塞是在通知内核：当所请求的I/O操作非得把本进程投入睡眠才能完成时，不要把本进程投入睡眠，而是返回一个错误。也就是说当数据没有到达时并不等待，而是以一个错误返回。

3、I/O复用模型

调用select或poll，在这两个系统调用中的某一个上阻塞，而不是阻塞于真正I/O系统调用。 阻塞于select调用，等待数据报套接口可读。当select返回套接口可读条件时，调用recevfrom将数据报拷贝到应用缓冲区中。

4、信号驱动I/O模型

首先开启套接口信号驱动I/O功能，并通过系统调用sigaction安装一个信号处理函数（此系统调用立即返回，进程继续工作，它是非阻塞的）。当数据报准备好被读时，就为该进程生成一个SIGIO信号。随即可以在信号处理程序中调用recvfrom来读数据报，并通知主循环数据已准备好被处理中。也可以通知主循环，让它来读数据报。

5、异步I/O模型

告知内核启动某个操作，并让内核在整个操作完成后(包括将数据从内核拷贝到用户自己的缓冲区)通知我们。这种模型与信号驱动模型的主要区别是：

信号驱动I/O：由内核通知我们何时可以启动一个I/O操作，

异步I/O模型：由内核通知我们I/O操作何时完成。

1、基本概念

IO多路复用是指内核一旦发现进程指定的一个或者多个IO条件准备读取，它就通知该进程。IO多路复用适用如下场合：

（1）当客户处理多个描述字时（一般是交互式输入和网络套接口），必须使用I/O复用。

（2）当一个客户同时处理多个套接口时，而这种情况是可能的，但很少出现。

（3）如果一个TCP服务器既要处理监听套接口，又要处理已连接套接口，一般也要用到I/O复用。

（4）如果一个服务器即要处理TCP，又要处理UDP，一般要使用I/O复用。

（5）如果一个服务器要处理多个服务或多个协议，一般要使用I/O复用。

与多进程和多线程技术相比，I/O多路复用技术的最大优势是系统开销小，系统不必创建进程/线程，也不必维护这些进程/线程，从而大大减小了系统的开销。

2、select函数

该函数准许进程指示内核等待多个事件中的任何一个发送，并只在有一个或多个事件发生或经历一段指定的时间后才唤醒。函数原型如下：

```
#include <sys/select.h>
#include <sys/time.h>

int select(int maxfdp1,fd_set *readset,fd_set *writeset,fd_set *exceptset,const struct timeval *timeout)
返回值：就绪描述符的数目，超时返回0，出错返回-1
```

函数参数介绍如下：

（1）第一个参数maxfdp1指定待测试的描述字个数，它的值是待测试的最大描述字加1（因此把该参数命名为maxfdp1），描述字0、1、2...maxfdp1-1均将被测试。

因为文件描述符是从0开始的。

（2）中间的三个参数readset、writeset和exceptset指定我们要让内核测试读、写和异常条件的描述字。如果对某一个的条件不感兴趣，就可以把它设为空指针。struct fd_set可以理解为一个集合，这个集合中存放的是文件描述符，可通过以下四个宏进行设置：

```
void FD_ZERO(fd_set *fdset);           //清空集合

void FD_SET(int fd, fd_set *fdset);   //将一个给定的文件描述符加入集合之中
```

```
void FD_CLR(int fd, fd_set *fdset);   //将一个给定的文件描述符从集合中删除
int FD_ISSET(int fd, fd_set *fdset);   // 检查集合中指定的文件描述符是否可以读写
```

（3）timeout告知内核等待所指定描述字中的任何一个就绪可花多少时间。其timeval结构用于指定这段时间的秒数和微秒数。

```
struct timeval{
        long tv_sec;   //seconds
        long tv_usec;  //microseconds
};
```

这个参数有三种可能：

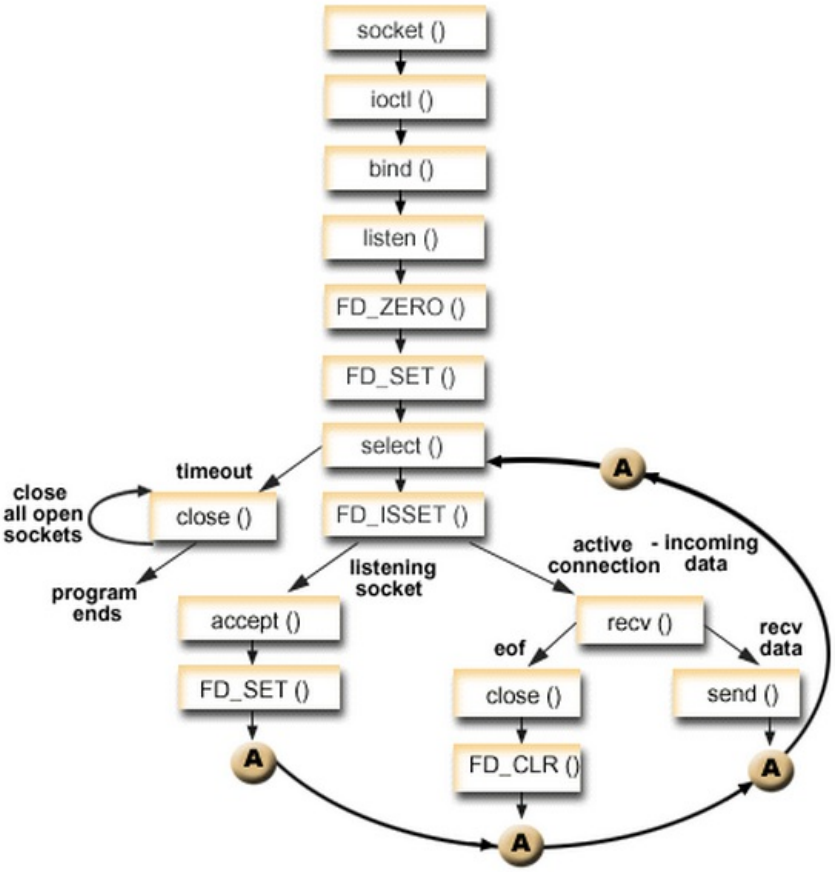
（1）永远等待下去：仅在有一个描述字准备好I/O时才返回。为此，把该参数设置为空指针NULL。

（2）等待一段固定时间：在有一个描述字准备好I/O时返回，但是不超过由该参数所指向的timeval结构中指定的秒数和微秒数。

（3）根本不等待：检查描述字后立即返回，这称为轮询。为此，该参数必须指向一个timeval结构，而且其中的定时器值必须为0。

原理图：

1 基本原理



1、基本知识

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。

2、poll函数

函数格式如下所示：

```
# include <poll.h>
int poll ( struct pollfd * fds, unsigned int nfds, int timeout);
```

pollfd结构体定义如下：

```
struct pollfd {

int fd;         /* 文件描述符 */
short events;       /* 等待的事件 */
short revents;      /* 实际发生了的事件 */
```

```
};
```

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。
POLLMSGSIGPOLL	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，使poll()一直挂起直到一个指定事件发生；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。
EFAULTfds	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。
EINVALnfds	参数超出PLIMIT_NOFILE值。
ENOMEM	可用内存不足，无法完成请求。

1、基本知识

epoll是在2.6内核中提出的，是之前的select和poll的增强版本。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，将用户关系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。

2、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);
```

（1） int epoll_create(int size);
创建一个epoll的句柄，size用来告诉内核这个监听的数目一共有多大。这个参数不同于select()中的第一个参数，给出最大监听的fd+1的值。需要注意的是，当创建好epoll句柄后，它就是会占用一个fd值，在linux下如果查看/proc/进程id/fd/，是能够看到这个fd的，所以在使用完epoll后，必须调用close()关闭，否则可能导致fd被耗尽。

（2）int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
epoll的事件注册函数，它不同与select()是在监听事件时告诉内核要监听什么类型的事件epoll的事件注册函数，它不同与select()是在监听事件时告诉内核要监听什么类型的事件，而是在这里先注册要监听的事件类型。第一个参数是epoll_create()的返回值，第二个参数表示动作，用三个宏来表示：
EPOLL_CTL_ADD：注册新的fd到epfd中；
EPOLL_CTL_MOD：修改已经注册的fd的监听事件；
EPOLL_CTL_DEL：从epfd中删除一个fd；
第三个参数是需要监听的fd，第四个参数是告诉内核需要监听什么事，struct epoll_event结构如下：

```
struct epoll_event {
```

```
    __uint32_t events;  /* Epoll events */
    epoll_data_t data;  /* User data variable */
};
```

events可以是以下几个宏的集合：
EPOLLIN ：表示对应的文件描述符可以读（包括对端SOCKET正常关闭）；
EPOLLOUT：表示对应的文件描述符可以写；
EPOLLPRI：表示对应的文件描述符有紧急的数据可读（这里应该表示有带外数据到来）；
EPOLLERR：表示对应的文件描述符发生错误；
EPOLLHUP：表示对应的文件描述符被挂断；
EPOLLET： 将EPOLL设为边缘触发(Edge Triggered)模式，这是相对于水平触发(Level Triggered)来说的。
EPOLLONESHOT：只监听一次事件，当监听完这次事件之后，如果还需要继续监听这个socket的话，需要再次把这个socket加入到EPOLL队列里

（3） int epoll_wait(int epfd, struct epoll_event * events, int maxevents, int timeout);

等待事件的产生，类似于select()调用。参数events用来从内核得到事件的集合，maxevents告之内核这个events有多大，这个maxevents的值不能大于创建epoll_create()时的size，参数timeout是超时时间（毫秒，0会立即返回，-1将不确定，也有说法说是永久阻塞）。该函数返回需要处理的事件数目，如返回0表示已超时。

3、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。

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

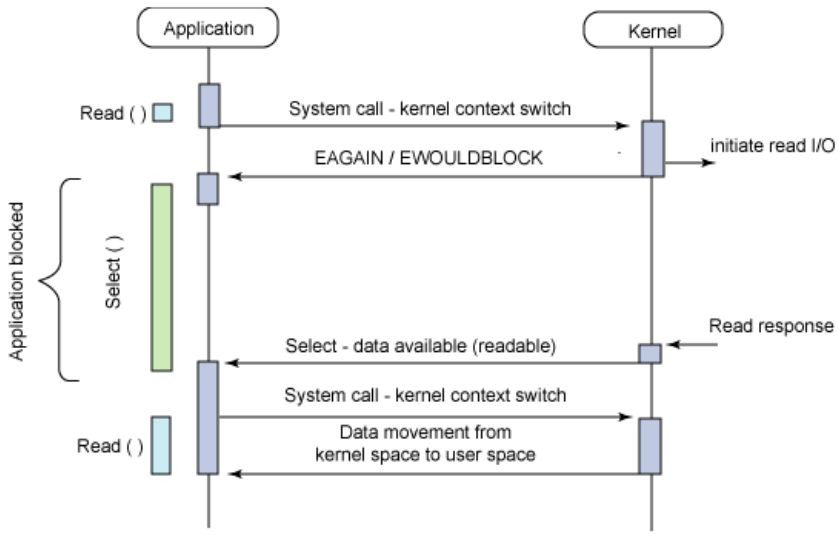
poll：http://www.cnblogs.com/Anker/archive/2013/08/15/3261006.html

epoll：http://www.cnblogs.com/Anker/archive/2013/08/17/3263780.html

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。