<https://blog.csdn.net/shenlei19911210/article/details/50586913>

epoll - I/O event notification facility  
  
在linux的网络编程中，很长的时间都在使用select来做事件触发。在linux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。并且，在linux/posix\_types.h头文件有这样的声明：  
#define \_\_FD\_SETSIZE    1024  
表示select最多同时监听1024个fd，当然，可以通过修改头文件再重编译内核来扩大这个数目，但这似乎并不治本。  
  
epoll的接口非常简单，一共就三个函数：  
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，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：  
  
typedef union epoll\_data {  
    void \*ptr;  
    int fd;  
    \_\_uint32\_t u32;  
    \_\_uint64\_t u64;  
} epoll\_data\_t;  
  
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。  
  
  
4、关于ET、LT两种工作模式：  
可以得出这样的结论:  
ET模式仅当状态发生变化的时候才获得通知,这里所谓的状态的变化并不包括缓冲区中还有未处理的数据,也就是说,如果要采用ET模式,需要一直read/write直到出错为止,很多人反映为什么采用ET模式只接收了一部分数据就再也得不到通知了,大多因为这样;而LT模式是只要有数据没有处理就会一直通知下去的.  
  
  
那么究竟如何来使用epoll呢？其实非常简单。  
通过在包含一个头文件#include <sys/epoll.h> 以及几个简单的API将可以大大的提高你的网络服务器的支持人数。  
  
首先通过create\_epoll(int maxfds)来创建一个epoll的句柄，其中maxfds为你epoll所支持的最大句柄数。这个函数会返回一个新的epoll句柄，之后的所有操作将通过这个句柄来进行操作。在用完之后，记得用close()来关闭这个创建出来的epoll句柄。  
  
之后在你的网络主循环里面，每一帧的调用epoll\_wait(int epfd, epoll\_event events, int max events, int timeout)来查询所有的网络接口，看哪一个可以读，哪一个可以写了。基本的语法为：  
nfds = epoll\_wait(kdpfd, events, maxevents, -1);  
其中kdpfd为用epoll\_create创建之后的句柄，events是一个epoll\_event\*的指针，当epoll\_wait这个函数操作成功之后，epoll\_events里面将储存所有的读写事件。max\_events是当前需要监听的所有socket句柄数。最后一个timeout是 epoll\_wait的超时，为0的时候表示马上返回，为-1的时候表示一直等下去，直到有事件范围，为任意正整数的时候表示等这么长的时间，如果一直没有事件，则范围。一般如果网络主循环是单独的线程的话，可以用-1来等，这样可以保证一些效率，如果是和主逻辑在同一个线程的话，则可以用0来保证主循环的效率。  
  
epoll\_wait范围之后应该是一个循环，遍利所有的事件。  
  
几乎所有的epoll程序都使用下面的框架：  
  
    for( ; ; )  
    {  
        nfds = epoll\_wait(epfd,events,20,500);  
        for(i=0;i<nfds;++i)  
        {  
            if(events[i].data.fd==listenfd) //有新的连接  
            {  
                connfd = accept(listenfd,(sockaddr \*)&clientaddr, &clilen); //accept这个连接  
                ev.data.fd=connfd;  
                ev.events=EPOLLIN|EPOLLET;  
                epoll\_ctl(epfd,EPOLL\_CTL\_ADD,connfd,&ev); //将新的fd添加到epoll的监听队列中  
            }  
            else if( events[i].events&EPOLLIN ) //接收到数据，读socket  
            {  
                n = read(sockfd, line, MAXLINE)) < 0    //读  
                ev.data.ptr = md;     //md为自定义类型，添加数据  
                ev.events=EPOLLOUT|EPOLLET;  
                epoll\_ctl(epfd,EPOLL\_CTL\_MOD,sockfd,&ev);//修改标识符，等待下一个循环时发送数据，异步处理的精髓  
            }  
            else if(events[i].events&EPOLLOUT) //有数据待发送，写socket  
            {  
                struct myepoll\_data\* md = (myepoll\_data\*)events[i].data.ptr;    //取数据  
                sockfd = md->fd;  
                send( sockfd, md->ptr, strlen((char\*)md->ptr), 0 );        //发送数据  
                ev.data.fd=sockfd;  
                ev.events=EPOLLIN|EPOLLET;  
                epoll\_ctl(epfd,EPOLL\_CTL\_MOD,sockfd,&ev); //修改标识符，等待下一个循环时接收数据  
            }  
            else  
            {  
                //其他的处理  
            }  
        }  
    }  
  
  
  
下面给出一个完整的服务器端例子：

|  |
| --- |
| #include <iostream> #include <sys/socket.h> #include <sys/epoll.h> #include <netinet/in.h> #include <arpa/inet.h> #include <fcntl.h> #include <unistd.h> #include <stdio.h> #include <errno.h>  using namespace std;  #define MAXLINE 5 #define OPEN\_MAX 100 #define LISTENQ 20 #define SERV\_PORT 5000 #define INFTIM 1000  void setnonblocking(int sock) {     int opts;     opts=fcntl(sock,F\_GETFL);     if(opts<0)     {         perror("fcntl(sock,GETFL)");         exit(1);     }     opts = opts|O\_NONBLOCK;     if(fcntl(sock,F\_SETFL,opts)<0)     {         perror("fcntl(sock,SETFL,opts)");         exit(1);     } }  int main(int argc, char\* argv[]) {     int i, maxi, listenfd, connfd, sockfd,epfd,nfds, portnumber;     ssize\_t n;     char line[MAXLINE];     socklen\_t clilen;       if ( 2 == argc )     {         if( (portnumber = atoi(argv[1])) < 0 )         {             fprintf(stderr,"Usage:%s portnumber/a/n",argv[0]);             return 1;         }     }     else     {         fprintf(stderr,"Usage:%s portnumber/a/n",argv[0]);         return 1;     }        //声明epoll\_event结构体的变量,ev用于注册事件,数组用于回传要处理的事件      struct epoll\_event ev,events[20];     //生成用于处理accept的epoll专用的文件描述符      epfd=epoll\_create(256);     struct sockaddr\_in clientaddr;     struct sockaddr\_in serveraddr;     listenfd = socket(AF\_INET, SOCK\_STREAM, 0);     //把socket设置为非阻塞方式      //setnonblocking(listenfd);      //设置与要处理的事件相关的文件描述符      ev.data.fd=listenfd;     //设置要处理的事件类型      ev.events=EPOLLIN|EPOLLET;     //ev.events=EPOLLIN;      //注册epoll事件      epoll\_ctl(epfd,EPOLL\_CTL\_ADD,listenfd,&ev);     bzero(&serveraddr, sizeof(serveraddr));     serveraddr.sin\_family = AF\_INET;     char \*local\_addr="127.0.0.1";     inet\_aton(local\_addr,&(serveraddr.sin\_addr));//htons(portnumber);      serveraddr.sin\_port=htons(portnumber);     bind(listenfd,(sockaddr \*)&serveraddr, sizeof(serveraddr));     listen(listenfd, LISTENQ);     maxi = 0;     for ( ; ; ) {         //等待epoll事件的发生          nfds=epoll\_wait(epfd,events,20,500);         //处理所发生的所有事件          for(i=0;i<nfds;++i)         {             if(events[i].data.fd==listenfd)//如果新监测到一个SOCKET用户连接到了绑定的SOCKET端口，建立新的连接。              {                 connfd = accept(listenfd,(sockaddr \*)&clientaddr,&clilen);                 if(connfd<0){                     perror("connfd<0");                     exit(1);                 }                 //setnonblocking(connfd);                  char \*str = inet\_ntoa(clientaddr.sin\_addr);                 cout << "accapt a connection from " << str << endl;                 //设置用于读操作的文件描述符                  ev.data.fd=connfd;                 //设置用于注测的读操作事件                  ev.events=EPOLLIN|EPOLLET;                 //ev.events=EPOLLIN;                  //注册ev                  epoll\_ctl(epfd,EPOLL\_CTL\_ADD,connfd,&ev);             }             else if(events[i].events&EPOLLIN)//如果是已经连接的用户，并且收到数据，那么进行读入。              {                 cout << "EPOLLIN" << endl;                 if ( (sockfd = events[i].data.fd) < 0)                     continue;                 if ( (n = read(sockfd, line, MAXLINE)) < 0) {                     if (errno == ECONNRESET) {                         close(sockfd);                         events[i].data.fd = -1;                     } else                         std::cout<<"readline error"<<std::endl;                 } else if (n == 0) {                     close(sockfd);                     events[i].data.fd = -1;                 }                 line[n] = '/0';                 cout << "read " << line << endl;                 //设置用于写操作的文件描述符                  ev.data.fd=sockfd;                 //设置用于注测的写操作事件                  ev.events=EPOLLOUT|EPOLLET;                 //修改sockfd上要处理的事件为EPOLLOUT                  //epoll\_ctl(epfd,EPOLL\_CTL\_MOD,sockfd,&ev);              }             else if(events[i].events&EPOLLOUT) // 如果有数据发送              {                 sockfd = events[i].data.fd;                 write(sockfd, line, n);                 //设置用于读操作的文件描述符                  ev.data.fd=sockfd;                 //设置用于注测的读操作事件                  ev.events=EPOLLIN|EPOLLET;                 //修改sockfd上要处理的事件为EPOLIN                  epoll\_ctl(epfd,EPOLL\_CTL\_MOD,sockfd,&ev);             }         }     }     return 0; } |

客户端直接连接到这个服务器就好了。。

引用：<http://blog.chinaunix.net/u/16292/showart_1844376.html>