**一、epoll概述**

epoll 是在 2.6 内核中提出的，是之前的 select() 和 poll() 的增强版本。

**二、epoll操作过程需要的四个接口函数**

四接口函数分别是：

#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);

int close(int epfd);

int epoll\_create(int size);

功能：

该函数生成一个epoll专用的文件描述符（其余的接口函数一般都用使用这个专用的文件描述符）

参数：

size:

用来告诉内核这个监听的数目一共有多大，size 参数是被忽略的，也就是说可以填大于 0 的任意值

返回值：

成功：epoll 专用的文件描述符 失败：-1

注：当创建好 epoll 句柄后，它就是会占用一个 fd 值，在使用完 epoll 后，必须调用 close() 关闭

int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event);

功能：

epoll 的事件注册函数

参数：

epfd: epoll 专用的文件描述符，epoll\_create()的返回值

op: 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；

fd: 需要监听的文件描述符

event: 告诉内核要监听什么事件，struct epoll\_event 结构如下：

// 感兴趣的事件和被触发的事件

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

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

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

data数据结构如下：

// 保存触发事件的某个文件描述符相关的数据（与具体使用方式有关）

typedef union epoll\_data {

void \*ptr;

int fd;

\_\_uint32\_t u32;

\_\_uint64\_t u64;

} epoll\_data\_t;

返回值：

成功：0 失败：-1

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

功能：

等待事件的产生

参数：

epfd: epoll 专用的文件描述符，epoll\_create()的返回值

events:

分配好内存的 epoll\_event 结构体数组，epoll 将会把发生的事件赋值到events 数组中（events 不可以是空指针，因为内核态无法在用户态中分配内存）。

maxevents: maxevents 告之内核这个 events 有多大 。

timeout: 超时时间，单位为毫秒，为 -1 时，函数为阻塞

返回值：

成功：返回需要处理的事件数目，如返回 0 表示已超时。

失败：-1

扩展：  
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。

int close(int epfd);

在用完之后，记得用close()来关闭这个创建出来的epoll句柄

**三、epoll示例：**

接下来我们epoll实现udp同时收发数据

#include <string.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/select.h>

#include <sys/time.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <sys/epoll.h>

int main()

{

    int udpfd = 0;

    int ret = 0;

    struct sockaddr\_in saddr;

struct sockaddr\_in caddr;

    bzero(&saddr,sizeof(saddr));

    saddr.sin\_family = AF\_INET;

    saddr.sin\_port   = htons(8000);

    saddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

    bzero(&caddr,sizeof(caddr));

    caddr.sin\_family  = AF\_INET;

    caddr.sin\_port    = htons(8000);

    //创建套接字

    if( (udpfd = socket(AF\_INET,SOCK\_DGRAM, 0)) < 0)

    {

        perror("socket error");

        exit(-1);

    }

    //套接字端口绑字

    if(bind(udpfd, (struct sockaddr\*)&saddr, sizeof(saddr)) != 0)

    {

        perror("bind error");

        close(udpfd);

        exit(-1);

}

    printf("input: \"sayto 192.168.220.X\" to sendmsg to somebody\033[32m\n");

    int epfd = epoll\_create(10); // 创建一个 epoll 的句柄，参数要大于 0， 没有太大意义

    if( -1 == epfd ){

        perror ("epoll\_create");

        return -1;

}

    struct epoll\_event event;   // 告诉内核要监听什么事件

    struct epoll\_event wait\_event;  // 用于存放内核返回的事件

    event.data.fd = 0;     // 标准输入

    event.events = EPOLLIN; // 表示对应的文件描述符可以读

    // 事件注册函数，将标准输入描述符 0 加入监听事件

    ret = epoll\_ctl(epfd, EPOLL\_CTL\_ADD, 0, &event);

    if(-1 == ret){

        perror("epoll\_ctl");

        return -1;

    }

    event.data.fd = udpfd;     // 有名管道

    event.events = EPOLLIN; // 表示对应的文件描述符可以读

    // 事件注册函数，将有udp描述符udpfd 加入监听事件

    ret = epoll\_ctl(epfd, EPOLL\_CTL\_ADD, udpfd, &event);

    if(-1 == ret){

        perror("epoll\_ctl");

        return -1;

    }

    while(1)

    {

        // 监视并等待多个文件（标准输入，udp套接字）描述符的属性变化（是否可读）

        // 没有属性变化，这个函数会阻塞，直到有变化才往下执行，这里没有设置超时

        ret = epoll\_wait(epfd, &wait\_event, 2, -1);

        write(1,"UdpQQ:",6);

        if(ret == -1){ // 出错

            close(epfd);

            perror("epoll");

        }

        else if(ret > 0){ // 准备就绪的文件描述符

            char buf[100] = {0};

            if( ( 0 == wait\_event.data.fd )

            && ( EPOLLIN == wait\_event.events & EPOLLIN ) ){ // 标准输入

                fgets(buf, sizeof(buf), stdin);

                buf[strlen(buf) - 1] = '\0';

                if(strncmp(buf, "sayto", 5) == 0)

                {

                    char ipbuf[16] = "";

                    inet\_pton(AF\_INET, buf+6, &caddr.sin\_addr);//给addr套接字地址再赋值.

                    printf("\rsay to %s\n",inet\_ntop(AF\_INET,&caddr.sin\_addr,ipbuf,sizeof(ipbuf)));

                    continue;

                }

                else if(strcmp(buf, "exit")==0)

                {

                    close(udpfd);

                    exit(0);

                }

                sendto(udpfd, buf, strlen(buf),0,(struct sockaddr\*)&caddr, sizeof(caddr));

            }

            else if( ( udpfd == wait\_event.data.fd )

            && ( EPOLLIN == wait\_event.events & EPOLLIN )){ //udp套接字

                struct sockaddr\_in addr;

                char ipbuf[INET\_ADDRSTRLEN] = "";

                socklen\_t addrlen = sizeof(addr);

                bzero(&addr,sizeof(addr));

                recvfrom(udpfd, buf, 100, 0, (struct sockaddr\*)&addr, &addrlen);

                printf("\r\033[31m[%s]:\033[32m%s\n",inet\_ntop(AF\_INET,&addr.sin\_addr,ipbuf,sizeof(ipbuf)),buf);

            }

        }

        else if(0 == ret){ // 超时

            printf("time out\n");

        }

    }

    return 0;

}