epoll反应堆模式的实现-也就是libevent的实现原理。

#include <stdlib.h>

#include <stdio.h>

#include <stdio.h>

#include <sys/socket.h>

#include <sys/epoll.h>

#include <arpa/inet.h>

#include <fcntl.h>

#include <unistd.h>

#include <errno.h>

#include <string.h>

#include <time.h>

#define MAX\_EVENTS  1024

#define BUFLEN 128

#define SERV\_PORT   8080

// 事件处理器

struct myevent\_s {

    // 描述符

    int fd;

    // 触发的事件

    int events;

    // 指向自己结构体指针

    void \*arg;

    // 回调函数

    void (\*call\_back)(int fd, int events, void \*arg);

    // 1表示在监听事件中，0表示不在

    int status;

    char buf[BUFLEN];

    int len;

    // 记录最后一次响应时间,做超时处理

    long last\_active;

};

// 同步事件分离器使用的 epoll

int epollFd;

// 分发器使用的 以注册的事件处理器，+1 最后一个用于 监听套接字 fd

struct myevent\_s g\_events[MAX\_EVENTS+1];

// 设置 事件处理器 的信息

void eventset(struct myevent\_s \*ev, int fd, void (\*call\_back)(int, int, void \*), void \*arg)

{

    ev->fd = fd;

    ev->call\_back = call\_back;

    ev->events = 0;

    ev->arg = arg;

    ev->status = 0;

    //memset(ev->buf, 0, sizeof(ev->buf));

    //ev->len = 0;

ev->last\_active = time(NULL);

    return;

}

void recvdata(int fd, int events, void \*arg);

void senddata(int fd, int events, void \*arg);

// 设置 事件处理器 关心的事件

void eventadd(int efd, int events, struct myevent\_s \*ev)

{

    struct epoll\_event epv = {0, {0}};

    int op;

    epv.data.ptr = ev;

epv.events = ev->events = events;

    if (ev->status == 1) {

        op = EPOLL\_CTL\_MOD;

    }

    else {

        op = EPOLL\_CTL\_ADD;

        ev->status = 1;

}

    if (epoll\_ctl(efd, op, ev->fd, &epv) < 0)

        printf("event add failed [fd=%d], events[%d]\n", ev->fd, events);

    else

        printf("event add OK [fd=%d], op=%d, events[%0X]\n", ev->fd, op, events);

    return;

}

// 删除事件处理器关心的事件

void eventdel(int efd, struct myevent\_s \*ev)

{

struct epoll\_event epv = {0, {0}};

    if (ev->status != 1)

        return;

    epv.data.ptr = ev;

    ev->status = 0;

epoll\_ctl(efd, EPOLL\_CTL\_DEL, ev->fd, &epv);

    return;

}

// 监听套接字事件处理器回调函数

// 有新的连接请求时，该函数会被回调

void acceptconn(int lfd, int events, void \*arg)

{

    struct sockaddr\_in cin;

    socklen\_t len = sizeof(cin);

int cfd, i;

    // 接受连接

    if ((cfd = accept(lfd, (struct sockaddr \*)&cin, &len)) == -1) {

        if (errno != EAGAIN && errno != EINTR) {

            /\* 暂时不做出错处理 \*/

        }

        printf("%s: accept, %s\n", \_\_func\_\_, strerror(errno));

        return;

}

    do {

        for (i = 0; i < MAX\_EVENTS; i++) {

            if (g\_events[i].status == 0)

                break;

        }

        // i == MAX\_EVENTS 则说明连接已满

        if (i == MAX\_EVENTS) {

            printf("%s: max connect limit[%d]\n", \_\_func\_\_, MAX\_EVENTS);

            break;

        }

        int flag = 0;

        if ((flag = fcntl(cfd, F\_SETFL, O\_NONBLOCK)) < 0)

        {

            printf("%s: fcntl nonblocking failed, %s\n", \_\_func\_\_, strerror(errno));

            break;

        }

        // 添加一个 事件处理器

        eventset(&g\_events[i], cfd, recvdata, &g\_events[i]);

        // 设置 事件处理器 的 事件

        eventadd(epollFd, EPOLLIN, &g\_events[i]);

} while(0);

printf("new connect [%s:%d][time:%ld], pos[%d]\n", inet\_ntoa(cin.sin\_addr), ntohs(cin.sin\_port), g\_events[i].last\_active, i);

    return;

}

void recvdata(int fd, int events, void \*arg)

{

    struct myevent\_s \*ev = (struct myevent\_s \*)arg;

int len;

    // 接收数据

    len = recv(fd, ev->buf, sizeof(ev->buf), 0);

    // 接收完成之后删除 事件处理器

eventdel(epollFd, ev);

    if (len > 0) {

        ev->len = len;

        ev->buf[len] = '\0';

        printf("C[%d]:%s\n", fd, ev->buf);

        // 接收到数据

        // 这时候要返回数据给客户端，所以添加一个 写事件处理器，如果可写，那么 senddata 将被调用，参数为 ev

        eventset(ev, fd, senddata, ev);

        eventadd(epollFd, EPOLLOUT, ev);

    }

    else if (len == 0) {

        close(ev->fd);

        /\* ev-g\_events 地址相减得到偏移元素位置 \*/

        printf("[fd=%d] pos[%d], closed\n", fd, (int)(ev - g\_events));

    }

    else {

        close(ev->fd);

        printf("recv[fd=%d] error[%d]:%s\n", fd, errno, strerror(errno));

}

    return;

}

void senddata(int fd, int events, void \*arg)

{

    struct myevent\_s \*ev = (struct myevent\_s \*) arg;

int len;

    // 发送数据

len = send(fd, ev->buf, ev->len, 0);

    // 发送完成之后删除 事件处理器

    eventdel(epollFd, ev);

    if (len > 0) {

        printf("send[fd=%d], [%d]%s\n", fd, len, ev->buf);

        // 发送完成后要接收数据，所以添加 可读事件处理器

        eventset(ev, fd, recvdata, ev);

        // 设置 事件处理器 的事件

        eventadd(epollFd, EPOLLIN, ev);

    }

    else {

        close(ev->fd);

        printf("send[fd=%d] error %s\n", fd, strerror(errno));

}

    return;

}

void initlistensocket(int efd, short port)

{

    int listenFd = socket(AF\_INET, SOCK\_STREAM, 0);

fcntl(listenFd, F\_SETFL, O\_NONBLOCK);

    // 添加监听套接字的 事件处理器，g\_events[MAX\_EVENTS] 用于保存该处理器

    // acceptconn 是事件处理器的 回调函数

    eventset(&g\_events[MAX\_EVENTS], listenFd, acceptconn, &g\_events[MAX\_EVENTS]);

    // 设置 事件处理器 的 事件

eventadd(efd, EPOLLIN, &g\_events[MAX\_EVENTS]);

struct sockaddr\_in sin;

    memset(&sin, 0, sizeof(sin));

    sin.sin\_family = AF\_INET;

    sin.sin\_addr.s\_addr = INADDR\_ANY;

sin.sin\_port = htons(port);

bind(listenFd, (struct sockaddr \*)&sin, sizeof(sin));

    // 监听

listen(listenFd, 20);

    return;

}

int main(int argc, char \*argv[])

{

unsigned short port = SERV\_PORT;

    if (argc == 2)

        port = atoi(argv[1]);

epollFd = epoll\_create(MAX\_EVENTS+1);

    if (epollFd <= 0)

        printf("create efd in %s err %s\n", \_\_func\_\_, strerror(errno));

    // 初始化监听套接字

initlistensocket(epollFd, port);

    // 已发生的事件

struct epoll\_event events[MAX\_EVENTS+1];

    printf("server running:port[%d]\n", port);

    int checkpos = 0, i;

    while (1) {

        /\* 超时验证，每次测试 100 个链接，不测试 listenfd 当客户端60秒内没有和服务器通信，则关闭此客户端链接 \*/

        long now = time(NULL);

        for (i = 0; i < 100; i++, checkpos++) {

            if (checkpos == MAX\_EVENTS)

                checkpos = 0;

            if (g\_events[checkpos].status != 1)

                continue;

            // 如果 事件处理器 60s 内没有动作，则断开连接

            long duration = now - g\_events[checkpos].last\_active;

            if (duration >= 60) {

                close(g\_events[checkpos].fd);

                printf("[fd=%d] timeout\n", g\_events[checkpos].fd);

                eventdel(epollFd, &g\_events[checkpos]);

            }

        }

        // 等待事件发生

        int nfd = epoll\_wait(epollFd, events, MAX\_EVENTS+1, 1000);

        if (nfd < 0) {

            printf("epoll\_wait error, exit\n");

            break;

        }

        for (i = 0; i < nfd; i++) {

            // 获取事件处理器

            struct myevent\_s \*ev = (struct myevent\_s \*) events[i].data.ptr;

            // 可读

            if ((events[i].events & EPOLLIN) && (ev->events & EPOLLIN)) {

                ev->call\_back(ev->fd, events[i].events, ev->arg);

            }

            // 可写

            if ((events[i].events & EPOLLOUT) && (ev->events & EPOLLOUT)) {

                ev->call\_back(ev->fd, events[i].events, ev->arg);

            }

        }

}

    /\* 退出前释放所有资源 \*/

    return 0;

}