TCP/IP网络编程(十七)

笔记本: 网络编程
创建时间: 2018/11/16 10:26 **更新时间:** 2018/11/21 18:24
作者: xiangkang94@outlook.com
标签: 第十七章(优于select的epoll)

- 实现IO复用的传统方法select和poll，但是性能不满意，因此有Linux的epoll，BSD的kqueue，Solaris的/dev/poll,Windows的IOCP
- select不适合以web服务器端开发为主流的现代开发环境
    - 调用select后常见的针对所有文件描述符的循环语句
    - 每次调用select函数时都需要向函数传递监视对象信息
- select的两个发挥优势的条件
    - 兼容性
    - 服务器端接入者少
- epoll可以克服select的缺点
    - epoll_create：创建保存epoll文件描述符的空间 //对应 fd_set
    - epoll_ctl：向空间（位数组）注册并注销文件描述符 //对应FD_SET，FD_CLR
    - epoll_wait：等待文件描述符发生变化 //对应select
    - //epoll将发生事件的文件描述符集中在一起，放在epoll_event结构体中

```
struct epoll_event{
    __uint32_t events;
    epoll_data_t data;
}

typedef union epoll_data{
    void * ptr;
    int fd;
    __uint32_t u32;
    __uint64_t u64;
}epoll_data_t;
```

- epoll_create函数
    - 调用epoll_create函数时创建的文件描述符保存空间称为“epoll例程”，size只是建议epoll例程大小，实际大小由操作系统决定。
    - Linux2.6.8之后，内核忽略size参数，会根据情况自动调整。
    - epoll_create函数创建的资源与套接字相同，由操作系统管理，终止时要close

```
#include <sys/epoll.h>
int epoll_create(int size)
//成功时返回epoll文件描述符，失败返回-1
```

- epoll_ctl函数

```
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event)
//成功时返回0，失败-1
//epfd：用于注册监视对象的epoll例程（事件发生的监视范围）的文件描述符
//op：用于指定监视对象的添加、删除或更改//EPOLL_CTL_ADD,EPOLL_CTL_DEL,EPOLL_CTL_MOD
//fd：需要注册的监视对象文件描述符
//event：监视对象事件类型
events 成员：
EPOLLIN：需要读取数据的情况
EPOLLOUT：输出缓冲为空，可以立即发送数据的情况
EPOLLPRI：受到OOB数据情况
EPOLLRDHUP：断开连接或半关闭的情况
EPOLLERR：发生错误的情况；
EPOLLET：以边缘触发的方式得到事件通知
```

```
EPOLLONESHOT：发生一次事件后，相应的文件描述符不再受到事件通知
```

- epoll_wait函数

```
int epoll_wait(int epfd,struct epoll_event * events,int maxevents,int timeout)
//成功时返回事件的文件描述符数，失败-1
//epfd：事件发生监视范围epoll例程的文件描述符
//events：保存发生事件的文件描述符集合的结构体地址值
//maxevents：第二个参数中可以保存的最大事件数
//timeout：以1/1000秒为单位等待时间

//注意的是第二个参数所指的缓冲区需要动态分配，调用该函数后，返回发生时间的文件描述符数目，同时在第二个参数所指的缓冲区中保存发生时间的文件描述符集合，因此无需像select那样插入针对所有文件描述符的循环
```

- 边缘触发和条件触发

两者区别：在于发生事件的时间点。条件触发：只要输入缓冲有数据就一直通知该事件；边缘触发：输入缓冲受到数据时仅注册一次事件

epoll默认以条件触发方式工作，select也是以条件触发模式工作的。

- 边缘触发服务器端实现中必知的两点
  - 通过errno变量验证错误的原因
  - 为了完成非阻塞I/O，需要更改套接字特性
- 条件触发与边缘触发比较：应该从服务器端实现模型角度考虑
- 边缘触发能够做到接收数据与处理数据的时间点分离。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <sys/epoll.h>
#include <errno.h>
#include <fcntl.h>

#define BUF_SIZE    4
#define EPOLL_SIZE  50

void error_handling(char *message);
void setnonblockingmode(int fd);

int main(int argc, char *argv[]){
    int serv_sock,clnt_sock;
    char buf[BUF_SIZE];
    struct sockaddr_in serv_adr;
    struct sockaddr_in clnt_adr;
    socklen_t adr_sz;
    int str_len,i;
    struct epoll_event *ep_events;
    struct epoll_event event;
    int epfd,event_cnt;

    if(argc != 2){
        printf("Usage : %s <port>\n",argv[0]);
        exit(1);
    }

    serv_sock = socket(PF_INET,SOCK_STREAM,0);
    if(serv_sock == -1){
        error_handling("socket error");
    }
```

```
    memset(&serv_adr,0,sizeof(serv_adr));
    serv_adr.sin_family=AF_INET;
    serv_adr.sin_addr.s_addr=htonl(INADDR_ANY);
    serv_adr.sin_port=htons(atoi(argv[1]));

    if(bind(serv_sock,(struct sockaddr*)&serv_adr,sizeof(serv_adr)) == -1){
        error_handling("bind() error");
    }

    if(listen(serv_sock,5) == -1){
        error_handling("listen() error");
    }

    epfd = epoll_create(EPOLL_SIZE);  //创建epoll
    ep_events = malloc(sizeof(struct epoll_event)*EPOLL_SIZE);

    event.events = EPOLLIN;
    event.data.fd = serv_sock;
    epoll_ctl(epfd,EPOLL_CTL_ADD,serv_sock,&event);

    while(1){
    event_cnt = epoll_wait(epfd,ep_events,EPOLL_SIZE,-1);

    if(event_cnt  == -1){
        puts("epoll wait error");
        break;
    }

    puts("return epoll wait");
    for(i=0;i<event_cnt;i++){
        if(ep_events[i].data.fd == serv_sock){
            adr_sz = sizeof(clnt_adr);
            clnt_sock = accept(serv_sock,(struct sockaddr *)&clnt_adr,&adr_sz);
            setnonblockingmode(clnt_sock);  //将accept函数创建的套接字改为非阻塞模式
            event.events = EPOLLIN | EPOLLET; //设置成边缘触发
            event.data.fd = clnt_sock;
            epoll_ctl(epfd,EPOLL_CTL_ADD,clnt_sock,&event);
            printf("connected client : %d \n", clnt_sock);
        }
        else{  //read message
            while(1){  //由于边缘触发时间只注册一次，所以需要一次性读完所有数据，用while循环，配合非阻塞的套接字，将
数据全部读出
                str_len = read(ep_events[i].data.fd,buf,BUF_SIZE);
                if(str_len == 0){   //close
                    epoll_ctl(epfd,EPOLL_CTL_DEL,ep_events[i].data.fd,NULL);
                    close(ep_events[i].data.fd);
                    printf("closed client %d \n",ep_events[i].data.fd);
                    break;
                }
                else if(str_len < 0){
                   if(errno == EAGAIN)    //无数据可读,此时read返回-1且errno的值为EAGAIN
                    break;
                }
                else{
                    write(ep_events[i].data.fd,buf,str_len);    //echo
                }
            }
         }
    }
  }
  close(serv_sock);
  close(epfd);
  return 0;
}
void error_handling(char *message){
    fputs(message,stderr);
    fputs("\n",stderr);
    exit(1);
}
```

```
void setnonblockingmode(int fd){
    int flag = fcntl(fd,F_GETFL,0); //   获取之前的文件描述符的属性
    fcntl(fd,F_SETFL,flag | O_NONBLOCK); //设置成非阻塞形式
}
```