一、进程池：

1. 服务器server端

（1）service.c

#include"func.h"

int exit\_fds[2]; //定义成全局变量来传参（信号处理函数只能全局变量来传参）

void sig\_exit(int signum)

{

char exit\_flag='a';

write(exit\_fds[1],&exit\_flag,1);

}

int main(int argc,char \*argv[])

{

check\_args(argc,4);

pipe(exit\_fds); //用于服务器退出机制

int process\_num=atoi(argv[3]); //创建的进程数目

pData\_t p=(pData\_t)calloc(process\_num,sizeof(Data\_t)); //创建指针并申请好空间，来保存创建的各个子进程的信息

make\_child(p,process\_num);

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

int ret,i;

int reuse=1;

ret=setsockopt(sfd,SOL\_SOCKET,SO\_REUSEADDR,&reuse,sizeof(int));

check\_error(-1,ret,"setsockopt");

int epfd=epoll\_create(1);

struct epoll\_event event,\*evs; //注意：evs空间内还有一个TCP套接字sfd

evs=(struct epoll\_event\*)calloc(**process\_num+1**,sizeof(struct epoll\_event));

event.events=EPOLLIN;

event.data.fd=sfd;

epoll\_ctl(epfd,EPOLL\_CTL\_ADD,sfd,&event);

for(i=0;i<process\_num;i++)

{

event.events=EPOLLIN;

event.data.fd=p[i].fds;

ret=epoll\_ctl(epfd,EPOLL\_CTL\_ADD,p[i].fds,&event);

check\_error(-1,ret,"epoll\_ctl");

}

event.events=EPOLLIN;

event.data.fd=exit\_fds[0];

epoll\_ctl(epfd,EPOLL\_CTL\_ADD,exit\_fds[0],&event);

struct sockaddr\_in ser;

bzero(&ser,sizeof(ser));

ser.sin\_family=AF\_INET;

ser.sin\_port=htons(atoi(argv[2]));

ser.sin\_addr.s\_addr=inet\_addr(argv[1]);

ret=bind(sfd,(struct sockaddr\*)&ser,sizeof(struct sockaddr));

check\_error(-1,ret,"bind");

listen(sfd,process\_num+1);

**signal(SIGUSR1,sig\_exit);** //用于服务器退出机制

int read\_fd\_num,j,new\_fd;

char notbusy\_flag=1;

while(1)

{

read\_fd\_num=epoll\_wait(epfd,evs,process\_num+1,-1); //可读描述符的个数

for(i=0;i<read\_fd\_num;i++)

{

if(evs[i].data.fd==sfd)

{

new\_fd=accept(sfd,NULL,NULL);

for(j=0;j<process\_num;j++)

{

if(0==p[j].busy\_flag)

{

send\_fd(p[j].fds,new\_fd,0); //发送描述符给子进程

p[j].busy\_flag=1;

printf("pid=%d is busy\n",p[j].pid);

break;

}

}

close(new\_fd); //若子进程都在忙碌时又有新的客户端连接，则新的客户端会直接断开 （区别于线程池并不会断开： 因为其用了队列机制来保持公平性）

}

if(evs[i].data.fd==exit\_fds[0])

{

close(sfd);

for(j=0;j<process\_num;j++)

{

send\_fd(p[j].fds,0,-1);

}

for(j=0;j<process\_num;j++)

{

wait(NULL);

}

printf("exit success\n");

return 0;

}

//查看是哪个子进程通知父进程自己不忙碌了

for(j=0;j<process\_num;j++)

{

if(evs[i].data.fd==p[j].fds) //某个子进程的fds管道可读

{

read(p[j].fds,&notbusy\_flag,1);

p[j].busy\_flag=0;

printf("pid=%d is not busy\n",p[j].pid);

}

}

}

}

return 0;

}

（2）child.c

#include "func.h"

int make\_child(pData\_t p,int process\_num)

{

    int i;

    int fds[2];

    pid\_t pid;

    for(i=0;i<process\_num;i++)

    {

        socketpair(AF\_LOCAL,SOCK\_STREAM,0,fds); //每个子进程均与父进程之间创建一个特殊管道（可读可写）来传递信息

        pid=fork();

        if(0==pid)

        {

            close(fds[1]);

            child\_handle(fds[0]); //子进程必须在child\_handle内进行exit退出

        }

        close(fds[0]);

        p[i].pid=pid; //记录子进程的pid

        p[i].fds=fds[1]; //记录子进程的特殊管道的对端

        p[i].busy\_flag=0; //忙碌标志，0代表非忙碌，1代表忙碌

    }

    return 0;

}

void child\_handle(int fds)

{

    int new\_fd;

    int exit\_flag=0;

    char notbusy\_flag=1;

    while(1)

    {

        recv\_fd(fds,&new\_fd,&exit\_flag); //等待接收任务

        if(-1==exit\_flag) //服务器的退出机制

        {

         exit(0); //子进程退出

        }

        tran\_file(new\_fd);

        write(fds,&notbusy\_flag,1); //写管道通知父进程我不忙了

    }

}

（3）send\_fd.c

#include "func.h"

int send\_fd(int sfd, int fd, **int exit\_flag**) //exit\_flag：用于通知子进程要退出了

{

    struct msghdr msg;

    bzero(&msg,sizeof(msg));

    char buf[10]="hello";

　 struct iovec iov[2];

    iov[0].iov\_base=buf;

    iov[0].iov\_len=5;

**iov[1].iov\_base=&exit\_flag;**

    iov[1].iov\_len=4;

msg.msg\_iov=iov;

    msg.msg\_iovlen=2;

    struct cmsghdr \*cmsg;

    int len=CMSG\_LEN(sizeof(int));

    cmsg=(struct cmsghdr \*)calloc(1,len);

    cmsg->cmsg\_len=len;

    cmsg->cmsg\_level = SOL\_SOCKET;

    cmsg->cmsg\_type = SCM\_RIGHTS;

**\*(int\*)CMSG\_DATA(cmsg)=fd;**

    msg.msg\_control=cmsg;

    msg.msg\_controllen=len;

    int ret;

**ret=sendmsg(sfd,&msg,0);**

    check\_error(-1,ret,"sendmsg");

    return 0;

}

int recv\_fd(int sfd, int \*fd, **int \*exit\_flag**)

{

    struct msghdr msg;

    bzero(&msg,sizeof(msg));

    char buf[10]="hello";

    struct iovec iov[2];

    iov[0].iov\_base=buf;

    iov[0].iov\_len=5;

**iov[1].iov\_base=exit\_flag;**

    iov[1].iov\_len=4;

    msg.msg\_iov=iov;

    msg.msg\_iovlen=2;

    struct cmsghdr \*cmsg;

    int len=CMSG\_LEN(sizeof(int));

    cmsg=(struct cmsghdr \*)calloc(1,len);

    cmsg->cmsg\_len=len;

    cmsg->cmsg\_level = SOL\_SOCKET;

    cmsg->cmsg\_type = SCM\_RIGHTS;

    msg.msg\_control=cmsg;

    msg.msg\_controllen=len;

    int ret;

**ret=recvmsg(sfd,&msg,0);**

//收到传来的msg的内容，其包含iov数组内容（iov\_base数据: 用于退出机制）

    check\_error(-1,ret,"recvmsg");

**\*fd=\*(int\*)CMSG\_DATA(cmsg);**

    return 0;

}

（4）tran\_file.c

#include "func.h"

void sigfunc(int signum)

{

    printf("%d is coming\n",signum);

}

int tran\_file(int new\_fd)

{

    train t;

    signal(SIGPIPE,sigfunc);

//先发文件名

    t.len=strlen(FILENAME);

    strcpy(t.buf,FILENAME);

    int ret;

    ret=send\_n(new\_fd,(char\*)&t,4+t.len);

    if(-1==ret) //若客户端在下载的过程中突然断开了，则服务器端

tran\_file.c的 send第一次返回-1时就要做判断，下同。

    {

        goto end;

}

    //发文件大小

    int fd=open(FILENAME,O\_RDONLY);

    check\_error(-1,fd,"open");

    struct stat buf;

    fstat(fd,&buf);

    t.len=sizeof(buf.st\_size); // t.len中存的是8个字节（off\_t是8个字节）注意：此处要传的是8，而不是长整型数: 因为文件大小 即是要发送的 文件内容。

    memcpy(t.buf,&buf.st\_size,sizeof(off\_t));//传长整型数(文件大小)到buf中

    ret=send\_n(new\_fd,(char\*)&t,4+t.len);

    if(-1==ret)

    {

        goto end;

}

    //发文件内容

while((t.len=read(fd,t.buf,sizeof(t.buf)))>0)

//注意： t.len一定不能用strlen来算

    {

        ret=send\_n(new\_fd,(char\*)&t,4+t.len);

        if(-1==ret)

        {

         goto end;

        }

}

//发送结束标志

    ret=send\_n(new\_fd,(char\*)&t,4+t.len); //此时刚好t.len中存的就是0

    if(-1==ret)

    {

        goto end;

    }

end:

    close(new\_fd);

    return 0;

}

（5）tran\_n.c

#include "func.h"

//循环发送

int send\_n(int sfd,char\* p,int len)

{

    int total=0;

    int ret;

    while(total<len)

    {

        ret=send(sfd,p+total,len-total,0);

//若客户端在下载过程中断开，send第一次会返回-1时，就要做个判断。

        if(-1==ret)

        {

            printf("client is close\n");

            return -1;

        }

        total=total+ret;

    }

    return 0;

}

int recv\_n(int sfd,char\* p,int len)

{

    int total=0;

    int ret;

    while(total<len)

    {

        ret=recv(sfd,p+total,len-total,0);//服务器断开：recv返回值0

        if(0==ret) //代表服务器端意外断开（客户端此时要退出）。 注意

区别服务器的意外断开（如ctrl+c） 和 退出机制（发10号信号）

        {

            return -1;

        }

        total=total+ret;

    }

    return 0;

}

2. 客户端client

（1）client\_time.c

#include "func.h"

int main(int argc,char \*argv[])

{

check\_args(argc,3);

int sfd;

struct sockaddr\_in ser;

bzero(&ser,sizeof(ser));

ser.sin\_family=AF\_INET;

ser.sin\_addr.s\_addr=inet\_addr(argv[1]);

ser.sin\_port=htons(atoi(argv[2]));

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

connect(sfd,(struct sockaddr\*)&ser,sizeof(ser));

int len;

char buf[1000]={0};

//接文件名

recv\_n(sfd,(char\*)&len,4);

recv\_n(sfd,buf,len);

//接文件大小

int fd=open(buf,O\_RDWR|O\_CREAT,0666);

check\_error(-1,fd,"open");

off\_t file\_size;

double download\_size=0;

recv\_n(sfd,(char\*)&len,4);

recv\_n(sfd,(char\*)&file\_size,len);

//接文件内容:按秒打印下载的百分比

time\_t start,end;

int ret;

start=time(NULL);

while(1)

{

bzero(buf,sizeof(buf));

ret=recv\_n(sfd,(char\*)&len,4);

if(ret!=-1&&len>0)

{

ret=recv\_n(sfd,buf,len); //根据len长度来读取buf内容

if(-1==ret) //服务器端突然意外断开

{

printf("download %5.2lf%s\n",download\_size/file\_size\*100,"%");

break;

}

write(fd,buf,len);

download\_size=download\_size+len;

end=time(NULL);

if(end-start>=1)

{

printf("download %5.2lf%s\r",download\_size/file\_size\*100,"%");

fflush(stdout);

start=end;

}

}

else //len等于0时，即读到了文件结束标志。

{

printf("download success 100%s\n","%");

break;

}

}

close(fd);

close(sfd);

return 0;

}

（2）client\_size.c

#include "func.h"

int main(int argc,char\*\* argv)

{

    args\_check(argc,3);

    int sfd;

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

    check\_error(-1,sfd,"socket");

    struct sockaddr\_in ser;

    bzero(&ser,sizeof(ser));

    ser.sin\_family=AF\_INET;

    ser.sin\_port=htons(atoi(argv[2])); //端口号转换为网络字节序

    ser.sin\_addr.s\_addr=inet\_addr(argv[1]);

    int ret;

    ret=connect(sfd,(struct sockaddr\*)&ser,sizeof(struct sockaddr));

    check\_error(-1,ret,"connect");

    int len;

    char buf[1000]={0};

    //接文件名

    recv\_n(sfd,(char\*)&len,4);

recv\_n(sfd,buf,len);

    //接文件大小：用于打印百分比

    off\_t file\_size;

    double down\_load\_size=0;

    recv\_n(sfd,(char\*)&len,4);

    recv\_n(sfd,(char\*)&file\_size,len);

    int fd=open(buf,O\_RDWR|O\_CREAT,0666);

check\_error(-1,fd,"open");

    //接文件内容，并按大小打印下载百分比

    off\_t compare\_size=file\_size/100;

    while(1)

    {

        ret=recv\_n(sfd,(char\*)&len,4); //服务器若突然断开，recv\_n返回-1

        if(ret!=-1&&len>0) //recv\_n返回-1,代表服务器突然意外断开了

        {

            ret=recv\_n(sfd,buf,len);

            if(-1==ret)

            {

                printf("down percent %5.2f%s\n",down\_load\_size/file\_size\*100,"%");

                break;

            }

            write(fd,buf,len);

            down\_load\_size=down\_load\_size+len;

            if(down\_load\_size>compare\_size)

            {

                printf("down percent %5.2f%s\r",down\_load\_size/file\_size\*100,"%");

                fflush(stdout);

                compare\_size=compare\_size+file\_size/100;

            }

        }else{

            printf("down percent %5.2f%s\n",down\_load\_size/file\_size\*100,"%");

            break;

        }

    }

    close(fd);

    close(sfd);

}

（3）tran\_n.c

#include "func.h"

//循环发送

int send\_n(int sfd,char\* p,int len)

{

    int total=0;

    int ret;

    while(total<len)

    {

        ret=send(sfd,p+total,len-total,0);

        if(-1==ret) //代表客户端断开：send第一次返回-1

        {

            printf("client is close\n");

            return -1;

        }

        total=total+ret;

    }

    return 0;

}

//循环接收

int recv\_n(int sfd,char\* p,int len)

{

    int total=0;

    int ret;

    while(total<len)

    {

        ret=recv(sfd,p+total,len-total,0);

        if(0==ret) //即服务器端断开，recv返回0 （对端断开，置可读状态）

        {

            return -1;

        }

        total=total+ret;

    }

    return 0;

}

二、线程池

（1） main.c #include "factory.h"

int exit\_fds[2];

void sig\_exit\_func(int signum)

{

char flag=1;

write(exit\_fds[1],&flag,1); //发给父进程来响应信号

}

void\* download\_file(void\* p)

{

pfac pf=(pfac)p;

pque\_t pq=&pf->que; //锁一定要通过地址来传递，不能赋值传递，否则就不是同一把锁了（锁一旦发生赋值就是另一把锁了，因为锁也有静态初始化）

pnode\_t pcur;

while(1)

{

pthread\_mutex\_lock(&pq->que\_mutex);

if(0==pq->size)

{

pthread\_cond\_wait(&pf->cond,&pq->que\_mutex);

}

judge\_thread\_exit(pq); //切记要放到que\_get前面

que\_get(pq,&pcur); //刚好唤醒后加锁，子线程趁锁，去队列里拿到结点

pthread\_mutex\_unlock(&pq->que\_mutex);

if(pcur!=NULL)

{

tran\_file(pcur->new\_fd);

free(pcur); //解锁后再去free：不要在加、解锁内去free

}

}

}

int main(int argc,char\* argv[])

{

pipe(exit\_fds);

if(fork()) //信号发送给父进程，由 父进程 通知 子进程中的线程池退出：

防止直接去给进程发信号时，任何一个子线程都可能去响应信号。

{

close(exit\_fds[0]);

signal(SIGUSR1,sig\_exit\_func);

pid\_t pid=wait(NULL);

printf("child process pid=%d\n",pid);

return 0; //父进程退出结束

}

close(exit\_fds[1]); //以下均子进程业务

if(argc!=5)

{

printf("./server IP PORT THREAD\_NUM CAPACITY");

return -1;

}

factory f;

int thread\_num=atoi(argv[3]);

int capacity=atoi(argv[4]);

factory\_init(&f,download\_file,thread\_num,capacity);

factory\_start(&f);

int sfd;

tcp\_start\_listen(&sfd,argv[1],argv[2],capacity);

int new\_fd;

pque\_t pq=&f.que;

pnode\_t pnew;

int ret;

int epfd=epoll\_create(1);

struct epoll\_event event,evs[2];

event.events=EPOLLIN;

event.data.fd=sfd;

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

check\_error(-1,ret,"epoll\_ctl");

event.data.fd=exit\_fds[0];

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

check\_error(-1,ret,"epoll\_ctl");

while(1)

{

ret=epoll\_wait(epfd,evs,2,-1);

for(int i=0;i<ret;i++)

{

if(evs[i].data.fd==sfd)

{

new\_fd=accept(sfd,NULL,NULL);

pnew=(pnode\_t)calloc(1,sizeof(node\_t));

pnew->new\_fd=new\_fd;

pthread\_mutex\_lock(&pq->que\_mutex);

que\_insert(pq,pnew);

pthread\_mutex\_unlock(&pq->que\_mutex);

pthread\_cond\_signal(&f.cond); //发送信号给子线程：刚开始子线程都在睡觉，要唤醒一个子线>程(若没有子线程在睡觉，则此句不起作用)

}

if(evs[i].data.fd==exit\_fds[0])

{

close(sfd);

pnew=(pnode\_t)calloc(1,sizeof(node\_t));

pnew->new\_fd=-1; //退出机制（退出结点值为-1）

pthread\_mutex\_lock(&pq->que\_mutex);

que\_insert\_exit(pq,pnew);

pthread\_mutex\_unlock(&pq->que\_mutex);

pthread\_cond\_broadcast(&f.cond); //用于线程池刚开启还没有接收任务时就要退出，此时队列中还没有放入一个结点，子线程全在睡觉,则此种情形要群体唤醒子线程， 各自去取 退出结点 并 线程退出。

for(int j=0;j<thread\_num;j++)

{

pthread\_join(f.pth\_id[j],NULL);

}

exit(0); //线程池全部退出并回收，此时子进程退出

}

}

}

}

（2） factory.c #include "factory.h"

int factory\_init(pfac pf,thread\_func\_t threadfunc,int thread\_num,int capacity)

{

bzero(pf,sizeof(factory));

pf->pth\_id=calloc(thread\_num,sizeof(pthread\_t)); //申请的存储线程ID的连续空间的首地址赋给pth\_id

pf->pthread\_num=thread\_num;

pf->que.que\_capacity=capacity;

pthread\_mutex\_init(&pf->que.que\_mutex,NULL);

pf->threadfunc=threadfunc;

pthread\_cond\_init(&pf->cond,NULL);

return 0;

}

int factory\_start(pfac pf)

{

if(!pf->start\_flag)

{

int i;

for(i=0;i<pf->pthread\_num;i++)

{

pthread\_create(pf->pth\_id+i,NULL,pf->threadfunc,pf);

}

pf->start\_flag=1;

}

return 0;

}

int tcp\_start\_listen(int \*psfd,char \*ip,char \*port,int backlog)

{

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

int reuse=1;

int ret;

ret=setsockopt(sfd,SOL\_SOCKET,SO\_REUSEADDR,&reuse,sizeof(int));

check\_error(-1,ret,"setsockopt");

struct sockaddr\_in ser;

bzero(&ser,sizeof(ser));

ser.sin\_family=AF\_INET;

ser.sin\_port=htons(atoi(port));

ser.sin\_addr.s\_addr=inet\_addr(ip);

ret=bind(sfd,(struct sockaddr\*)&ser,sizeof(ser));

check\_error(-1,sfd,"bind");

listen(sfd,backlog);

\*psfd=sfd;

return 0;

}

（3）work\_que.c #include "work\_que.h"

void que\_insert(pque\_t pq,pnode\_t pnew) //用尾插法把结点pnew插到链表

{

if(!pq->que\_head)

{

pq->que\_head=pnew;

pq->que\_tail=pnew;

}else{

pq->que\_tail->pNext=pnew;

pq->que\_tail=pnew;

}

pq->size++;

}

void que\_get(pque\_t pq,pnode\_t\* pcur) //子线程去取任务结点: pcur为二级指针

{

\*pcur=pq->que\_head;

if(NULL==\*pcur)

{

return;

}

pq->que\_head=pq->que\_head->pNext; //头部删除 队列中取出的结点

if(!pq->que\_head)

{

pq->que\_tail=NULL;

}

pq->size--;

}

void que\_insert\_exit(pque\_t pq,pnode\_t pnew)

{

if(NULL==pq->que\_head)

{

pq->que\_head=pnew;

pq->que\_tail=pnew;

}

else

{

pnew->pNext=pq->que\_head;

pq->que\_head=pnew;

}

pq->size++;

}

void judge\_thread\_exit(pque\_t pq)

{

if(-1==pq->que\_head->new\_fd)

{

pthread\_mutex\_unlock(&pq->que\_mutex);

pthread\_exit(NULL); //此子线程退出

}

}

（4）tran\_file.c 、 tran\_n.c 同进程池 。

三、 C语言连接数据库

（1） query.c 查询

//用此头文件及以下的数据库接口时：必须安装了apt-get install libmysqlclient-dev

#include <mysql/mysql.h> //安了libmysqlclient-dev 包，才能用开发相关的API

#include <stdio.h>

#include <string.h>

int main(int argc,char\* argv[])

{

if(argc!=2)

{

printf("error args\n");

return -1;

}

**MYSQL \*conn;**

**MYSQL\_RES \*res;**

**MYSQL\_ROW row;**

char\* server="localhost";

char\* user="root";

char\* password="123456";

char\* database="test25"; //要访问的数据库名称

**char query[300]="select \* from Person where personID=";**

//**sprintf(query,"%s%s",query,argv[1]);**  //仅查询传参所指定的行

**strcpy(query,"select \* from Person");** //全部查询

puts(query);

int t,r;

conn=**mysql\_init**(NULL); //初始化一个MySQL的 conn连接

//连接MySQL服务器

if(!**mysql\_real\_connect**(conn,server,user,password,database,0,NULL,0))

{//连接本机IP（localhost），并验证用户名、密码

printf("Error connecting to database:%s\n",mysql\_error(conn));

return -1;

}

else{

printf("Connected...\n");

}

**t=mysql\_query(conn,query);** //向conn 连接，发送query 查询指令

if(t) //特殊：若返回真true，指的是失败了。

{

printf("Error making query:%s\n",mysql\_error(conn));

}

else{

// printf("Query made...\n");

res=**mysql\_use\_result(conn)**; //用MySQL提供的此接口, 从数据库中recv获取数据并存放到res 中（不论数据库中存的是什么数据类型，返回来的 **均为字符串** 形式）

if(res)

{

//若返回来的有数据（不为NULL），则从res 中一次拿一行（res是指针数组名）

while((row=**mysql\_fetch\_row(res)**)!=NULL)

{

//printf("num=%d\n",mysql\_num\_fields(res));

for(t=0;t<**mysql\_num\_fields(res)**;t++) //能得出表中有多少列

{

printf("%8s ",row[t]);

}

printf("\n");

}

}

mysql\_free\_result(res); //若返回的没有数据，就free掉。

}

mysql\_close(conn); //关闭conn连接

return 0;

}

（2） insert.c新增

#include <mysql/mysql.h>

#include <stdio.h>

#include <string.h>

int main(int argc,char\* argv[])

{

if(argc!=2)

{

printf("error args\n");

return -1;

}

MYSQL \*conn;

MYSQL\_RES \*res;

MYSQL\_ROW row;

char\* server="localhost";

char\* user="root";

char\* password="123456";

char\* database="test25";

**char query[200]="insert into Person(FirstName,LastName,Age) values('xiong','";**

**sprintf(query,"%s%s%s",query,argv[1],"',2)");** //通过传参获取要插入的值

puts(query);

int t,r;

conn=mysql\_init(NULL);

if(!mysql\_real\_connect(conn,server,user,password,database,0,NULL,0))

{

printf("Error connecting to database:%s\n",mysql\_error(conn));

}else{

printf("Connected...\n");

}

**t=mysql\_query(conn,query);**  //向conn 连接，发送query 插入指令

if(t)

{

printf("Error making query:%s\n",mysql\_error(conn));

}else{

printf("insert success\n");

}

mysql\_close(conn);

return 0;

}

（3） delete.c删除

#include <mysql/mysql.h>

#include <stdio.h>

#include <string.h>

int main(int argc,char\* argv[])

{

MYSQL \*conn;

MYSQL\_RES \*res;

MYSQL\_ROW row;

char\* server="localhost";

char\* user="root";

char\* password="123456";

char\* database="test25";

**char query[200]="delete from Person where LastName='san'";**

int t,r;

conn=mysql\_init(NULL);

if(!mysql\_real\_connect(conn,server,user,password,database,0,NULL,0))

{

printf("Error connecting to database:%s\n",mysql\_error(conn));

}

else {

printf("Connected...\n");

}

**t=mysql\_query(conn,query);**  //此时为新增删除数据的命令

if(t)

{

printf("Error making query:%s\n",mysql\_error(conn));

}

else

{

if(**mysql\_affected\_rows (conn)**) //接口mysql\_affected\_rows 用于判断删除是否成功，若成功则返回要删除行的行数。

{

printf("some rows affected\n");

}else{

printf("no rows match\n"); //数据库中没有匹配到要删除的行

}

}

mysql\_close(conn);

return 0;

}

重点接口： [**mysql\_affected\_rows()**](https://dev.mysql.com/doc/refman/5.7/en/mysql-affected-rows.html)**:**

**Returns the number of rows changed/deleted/inserted by the last** [**UPDATE**](https://dev.mysql.com/doc/refman/5.7/en/update.html)**,**[**DELETE**](https://dev.mysql.com/doc/refman/5.7/en/delete.html)**, or**[**INSERT**](https://dev.mysql.com/doc/refman/5.7/en/insert.html)**query.**

因为有**参照完整性**： **必须要先删除外键所在表的行，才能删除其关联表所对应的行**。

注意： 若要删除的行并不存在，但MySQL语法没问题，也会返回成功的并执行else，故要用 mysql\_affected\_rows 来判断是否真正删除成功。

MySQL只支持有限的连接，如100个，当第101个连接时会直接拒绝。 编程上怎么解决？

类似线程池机制，用连接池来解决： 可先定义100个conn，全部饱和连接数据库，并把每一个连接都放入队列（链表实现）中，谁要用数据库就去队列中拿节点。当第101个线程来拿结点时队列中并没有结点，就去睡觉； 而一旦有结点放入队列中时，就发信号唤醒线程去取结点。 （不能用while一直连数据库，要防止CPU空转）

（4） update.c更改

#include <mysql/mysql.h>

#include <stdio.h>

#include <string.h>

int main(int argc,char\* argv[])

{

if(argc!=2)

{

printf("error args\n");

return -1;

}

MYSQL \*conn;

MYSQL\_RES \*res;

MYSQL\_ROW row;

char\* server="localhost";

char\* user="root";

char\* password="123456";

char\* database="test25";

**char query[200]="update Person set birthday='2018-03-19' where personID=";**

int personid=atoi(argv[1]);

**sprintf(query,"%s%d",query,personid);**

puts(query);

int t,r;

conn=mysql\_init(NULL);

if(!mysql\_real\_connect(conn,server,user,password,database,0,NULL,0))

{

printf("Error connecting to database:%s\n",mysql\_error(conn));

}else{

printf("Connected...\n");

}

**t=mysql\_query(conn,query);**

if(t)

{

printf("Error making query:%s\n",mysql\_error(conn));

}else{

printf("update success\n");

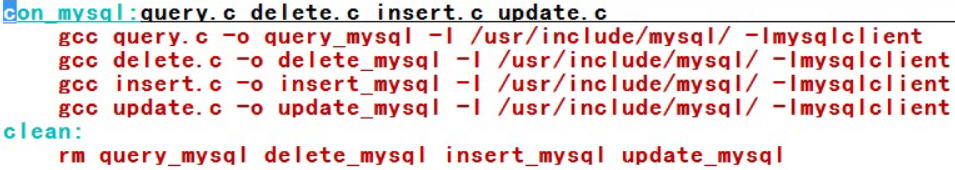
}

mysql\_close(conn);

return 0;

}

（5） 数据库makefile



要在库文件的搜索路径列表中添加：/usr/include/mysql/ （要引用这里面的头文件）

链接动态库：-lmysqlclient （要引用 mysqlclient动态库）

四、 排序1亿数： 选择、插入、递归快排、qsort排序（非递归快排）、堆排、计数排序

（1）func.h文件

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<time.h>

void arr\_print(int \*);

void arr\_select(int \*);

void arr\_insert(int \*);

void arr\_quick(int \*,int,int);

void arr\_heap(int \*);

void count\_sort(int \*);

int compara(void \*, void \*);

#define N 100000000

**#define swap(a,b) { int temp; temp=a; a=b; b=temp; }** //宏实现

（2）func.c排序实现

#include"func.h"

void arr\_print(int \*s)

{

for (int i = 0; i < N; i++)

{

printf("%d ", s[i]);

}

printf("\n-------------------------------------------\n");

}

void arr\_select(int \*s)

{

int i, j, max;

for (i = N - 1; i > 0; i--) //N个数有N-1趟

{

max = 0;

for (j = 0; j <= i; j++)

{

if (s[max] < s[j])

{

max = j;

}

}

swap(s[max], s[i]);

}

}

void arr\_insert(int \*s)

{

int i, j, k, insert\_val;

for (i = 1; i < N; i++) //要插入数据的下标(第一个数自然有序)

{

for (j = 0; j < i; j++) //找到插入的位置下标

{

if (s[i] < s[j])

{

insert\_val = s[i];

for (k = i - 1; k >= j; k--)

{

s[k + 1] = s[k];

}

s[j] = insert\_val;

break;

}

}

}

}

void arr\_quick(int \*s, int left, int right) //递归快排

{

if (left > right)

{

return;

}

**int pivot = s[left];**  //快排每趟的基准点

int i = left, j = right;

while (i != j)

{

while (s[j] >= pivot && i < j)

j--;

while (s[i] <= pivot && i < j)

i++;

if (i < j)

{

swap(s[i], s[j]);

}

}

//最终将基准数归位

s[left] = s[i];

s[i] = pivot;

//此时s[i]左边的均小于s[i]，右边的均大于s[i]

**arr\_quick(s, left, i - 1);**  //继续处理左边的，这里是一个递归的过程

**arr\_quick(s, i + 1, right);** //继续处理右边的，这里是一个递归的过程

return;

}

int compara(void \*a, void \*b) //qsort排序

{

return \*(int\*)a - \*(int\*)b;

}

void arr\_heap(int \*s) //堆排

{

void heap\_create(int \*);

void heapsort(int \*); //声明

heap\_create(s); //建立大根堆

heapsort(s); //堆排序

}

void heap\_create(int \*s)

{

void siftdown(int \*, int); //声明

//从最后一个非叶结点到第一个结点，依次进行向下调整。

**for (int i = N / 2 - 1; i >= 0; i--)**

{

siftdown(s, i);

}

}

void heapsort(int \*s)

{

extern int n; //全局变量声明：要去借用 main.c 里的全局变量n

while (n > 0)

{

**swap(s[0], s[n-1]);** //最大的元素s[0] 和 最后的元素s[n-1]交换

n--; //此时最大的元素已经在最后面了，堆大小减1

**siftdown(s, 0);** //将交换后的新s[0]向下调整，以保持堆的特性

}

}

void siftdown(int \*s, int i) //向下调整为大根堆

{

extern int n; //全局变量声明：要去借用 main.c 里的全局变量n

int index\_max, flag = 0; //flag 标记是否需要继续向下调整。

while (2 \* i + 1 < n && flag == 0) //有左孩子，且结点的值要小于其左孩子结点的值时：

{

if (s[i] < s[2 \* i + 1]) //判断结点和左儿子的关系。

index\_max = 2 \* i + 1;

else

index\_max = i;

//若有右孩子，且结点的值要小于其右孩子结点的值。

if (2 \* i + 2 < n && s[index\_max] < s[2 \* i + 2])

{

index\_max = 2 \* i + 2;

}

if (i != index\_max)

{

swap(s[i], s[index\_max]);

i = index\_max;

}

else

flag = 1; //说明当前父结点比两个孩子结点的值都大，不再需要调整了

}

}

void count\_sort(int \*s)

{

int \*a=(int\*)calloc(N,sizeof(int));

for (int i = 0; i < N; i++)

{

a[s[i]]++;

}

for (int i = 0; i < N; i++)

{

if (a[i])

{

for (int j = 0; j < a[i]; j++)

{

printf("%d\n",i);

}

}

}

printf("\n");

}

（3）main.c

#include"func.h"

int n=N; //全局变量n：存储堆中元素的个数，即堆的大小

int main()

{

int \*s;

s=(int\*)calloc(N,sizeof(int));;

srand(time(NULL));

for (int i = 0; i < N; i++)

{

s[i] = rand() % 100; //伪随机数

}

//arr\_print(s); //用于测试，排序1亿个数时注释掉

time\_t start = time(NULL);

//arr\_select(s); //选择排序

//arr\_insert(s); //插入排序

//arr\_quick(s, 0, N - 1); //递归的快排

//qsort(s, N, sizeof(int), compara); //调用qsort: 非递归的快排

arr\_heap(s); //堆排

count\_sort(s); //计数排序

time\_t end = time(NULL);

//arr\_print(s);

printf("use time is:%ld秒\n", end - start);

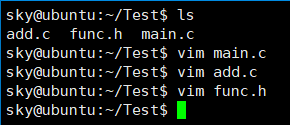
printf("\n");

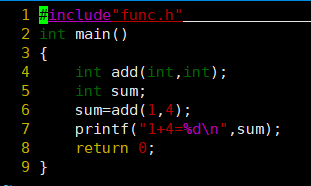
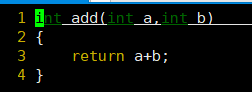
system("pause");

return 0;

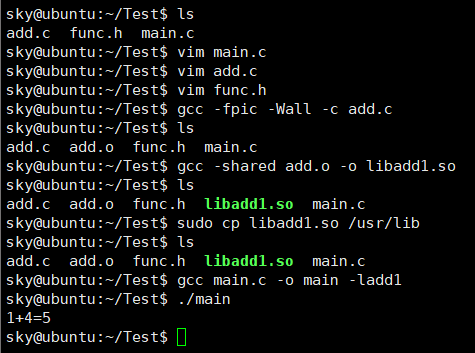
}

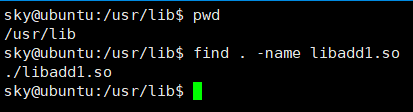
五、创建动态库、静态库

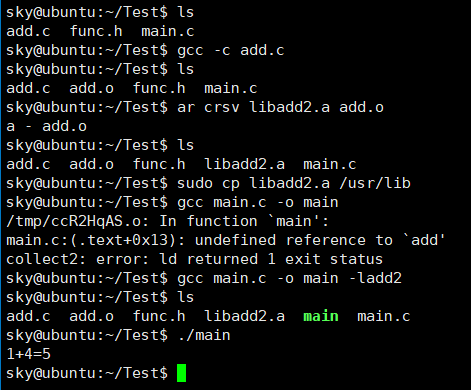
 

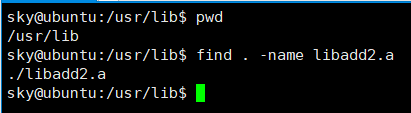
（1）创建动态库： 在程序执行时，才被链接使用。





（2）创建静态库： 静态库代码是在**编译时整合进去的**，故编译完后再删除静态库。





六、传递任意一个目录路径，能够显示该目录的ls -l的效果（用stat）

#include "func.h"

int main(int argc,char\* argv[])

{

args\_check(argc,2);

**DIR \*dir;**

**dir=opendir(argv[1]);**

if(!dir)

{

perror("opendir");

return -1;

}

**struct dirent \*p;**

off\_t pos;

**struct stat buf;**

int ret;

char path[512]={0};

while(( **p=readdir(dir)** ) != NULL ) //读取目录信息，并返回信息。

{

sprintf(path,"%s%s%s",argv[1],"/",p->d\_name);

**ret=stat(path,&buf);**//获取文件状态信息，并放到buf(数组path存的绝对路径)

if(-1==ret)

{

perror("stat");

return -1;

}

printf("inode=%ld,mode=%x,nlink=%ld,uid=%s,gid=%s,size=%ld,mtime=%s %s\n", buf.st\_ino, buf.st\_mode, buf.st\_nlink, **getpwuid(buf.st\_uid)->pw\_name**, **getgrgid(buf.st\_gid)->gr\_name**, buf.st\_size,ctime(&buf.st\_mtime), p->d\_name);

}

closedir(dir);

return 0;

}

//注意：

struct passwd \*getpwuid(uid\_t uid); //一般用于转uid(整型数)为文件所有者(字符)

struct group \*getgrgid(gid\_t gid); //一般用于转gid(整型数)为文件所属组(字符)

stat结构体成员st\_mode： 要用十六进制 %x 来打印： 其中16进制数的 低9位是权限， 最高的4位是文件类型(文件类型只有0 ~ 15)。