c语言程序设计

指针操作

声明函数指针

```
void (*funP)(int) ;   //声明一个指向同样参数、返回值的函数指针变量。
int (*fArray[10]) ( int ); // 声明函数指针数组
```

声明函数指针类型

```
typedef void(*FunType)(int);// 声明FunType类型为一个函数指针
void callFun(FunType fp,int x);// 声明后的类型可以用在函数参数中
```

声明指针数组

```
int* p1[10];// p1 是一个长度为10的数组,数组中存放整形指针变量
```

声明数组指针

```
int (*p2)[10];//p2 是一个指向含有10个整型数字数组的指针
```

动态分配数组指针

```
char (*a)[N];//指向数组的指针
a = (char (*)[N])malloc(sizeof(char *) * m);
```

多线程与多进程

多线程pthread

创建线程

pthread是linux下符合posix标准的线程库,其中关于线程的操作包括线程创建\线程设置\线程同步\线程取消等操作.线程函数的原型为

```
NAME
       pthread_create - create a new thread

SYNOPSIS
       #include <pthread.h>

       int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                          void *(*start_routine) (void *), void *arg);

       Compile and link with -pthread.
```

线程函数原型是一个参数为void类型指针,返回值为void类型指针的函数,创建成功返回0,失败返回错误代码,thread为pthread_t类型的指针,attr为线程属性

```
pthread_t t_thread;
res = pthread_create(&t_thread, NULL, thread_func, (void*)message);
if(res != 0){
    perror("线程创建失败! ");
    exit(EXIT_FAILURE);
}
```

结束线程

pthread_exit用于线程内部主动结束,传入的参数可以被pthread_join得到

```
NAME
       pthread_exit - terminate calling thread

SYNOPSIS
       #include <pthread.h>

       void pthread_exit(void *retval);

       Compile and link with -pthread.
```

```
NAME
       pthread_join - join with a terminated thread

SYNOPSIS
       #include <pthread.h>

       int pthread_join(pthread_t thread, void **retval);

       Compile and link with -pthread.
```

简单的示例

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<pthread.h>
```

```c
#include<unistd.h>
#include<sys/types.h>
char message[50] = "THREAD_TEST";
void* thread_func(void *arg);
int main(){
    pthread_t t_thread;
    void *thread_result;
    int res;
    res = pthread_create(&t_thread, NULL, thread_func, (void*)message);
    if(res != 0){
        perror("线程创建失败! ");
        exit(EXIT_FAILURE);
    }
    printf("wait for the thread!\n");
    pthread_join(t_thread, &thread_result);
    printf("线程已结束，返回值为%s\n", (char*)thread_result);
    printf("message的值为%s\n", message);
    free(thread_result);
    exit(EXIT_SUCCESS);
}

void* thread_func(void *arg){
    printf("线程正在运行，参数为%s\n", (char*)arg);
    sleep(3);
    strcpy(message, "线程修改");
    char* buf = (char*)malloc(strlen("线程执行完毕! "));
    strcpy(buf, "线程执行完毕! ");
    pthread_exit(buf);
}
```

取消线程

线程在运行的过程中可以被主线程中的函数取消,但是线程内部要先设置可以被取消的选项.这两个问题，我们都可以利用线程的取消点(cancellation points)来避免。线程的cancel type有两种：PTHREAD_CANCEL_DEFERRED和PTHREAD_CANCEL_ASYNCHRONOUS，前者为默认类型，意味着线程只有在取消点处才能被cancel。也就是说，在对线程pthread_cancel()之后，线程还要继续执行到下一个取消点才会退出。 可以通过pthread_setcanceltype()来改变线程的cancel type，但强烈不建议这样做，因为你如果改为PTHREAD_CANCEL_ASYNCHRONOUS类型，线程可以在代码的任何地方退出，就很难处理上述两个资源释放问题。 一个取消线程的简单示例

```c
#include<stdio.h>
#include<string.h>
#include<pthread.h>
#include<stdlib.h>
#include<unistd.h>
void *thread_func(void *arg){
    int res;
    res = pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
    if(res != 0){
        perror("设置线程取消失败");
        exit(EXIT_FAILURE);
    }
    if((res = pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL)) != 0){
```

```
        perror("设置线程取消类型失败! ");
        exit(EXIT_FAILURE);
    }
    printf("子线程正在运行! \n");
    for(int i = 0; i < 10; i++){
        sleep(1);
        printf("子线程正在运行! \n");
    }
    pthread_exit(EXIT_SUCCESS);
}
int main(){
    int res;
    pthread_t thread;
    if((res = pthread_create(&thread, NULL, thread_func, NULL)) != 0){
        perror("线程创建失败! ");
        exit(EXIT_FAILURE);
    }
    sleep(3);
    printf("取消线程! \n");
    if((res = pthread_cancel(thread)) != 0){
        perror("取消线程失败! ");
        exit(EXIT_FAILURE);
    }
    exit(EXIT_SUCCESS);
}
```

线程同步

信号量

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
#include<pthread.h>
#include<semaphore.h>
//thread_func declaration
void *thread_func(void* arg);
//semaphore global var
sem_t semth;
//buff used to receive the user input
#define BUFSIZE 1024
char buf[BUFSIZE];
//man function
int main(){
    //var declaration:int res, pthread_t used to create pthread,void thread_res used to join
    int res;
    pthread_t thread;
    void *thread_res;

    if((res = sem_init(&semth, 0, 0)) != 0){
        perror("create semaphore failed");
        exit(EXIT_FAILURE);
```

```
    }
//pthread create with error detection
    if((res = pthread_create(&thread, NULL, thread_func, NULL)) != 0){
        perror("create thread failed");
        exit(EXIT_FAILURE);
    }
//prompt information
    printf("Please input string information and type 'end' to exit!");
//get user input and make a P operation on sem_t
    while(strncmp(buf, "end", 3) != 0){
        fgets(buf, BUFSIZE, stdin);
        sem_post(&semth);
    }
//join the thread
    if((res = pthread_join(thread, &thread_res)) != 0){
        perror("thread join failed");
        exit(EXIT_FAILURE);
    }
//destory the semaphore
    if((res = sem_destroy(&semth)) != 0){
        perror("destory semaphore failed");
        exit(EXIT_FAILURE);
    }
//exit
    exit(EXIT_SUCCESS);
}

//thread_func definition
void *thread_func(void *arg){
//make V operation on sem_t
    sem_wait(&semth);
//while loop , get user info from buff, and make V operation on sem_t
    while(strncmp(buf, "end", 3) != 0){
        printf("you type:%s\n", buf);
        sem_wait(&semth);
    }
    fputs("exiting...\n", stdout);
//pthread_exit
    pthread_exit(0);
}
```

互斥量

pthread_mutex_init/pthread_mutex_lock/pthread_mutex_unlock/pthread_mutex_destory

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<unistd.h>
#include<pthread.h>
#include<semaphore.h>
// global var declaration
pthread_mutex_t mutex;//global mutex lock
```

```c
int exit_time = 1;//running symbol
char work_area[1024];
void* thread_func(void* arg){
    printf("子线程正在执行，对互斥量加锁\n");
    pthread_mutex_lock(&mutex);
    while(strncmp(work_area, "end", 3) != 0){
        printf("len:%d\n", strlen(work_area) - 1);
        work_area[0] = '\0';
        printf("解锁互斥量\n");
        pthread_mutex_unlock(&mutex);
        puts("睡眠1秒！\n");
        sleep(1);
        printf("判断信息是否为空\n");

        while(work_area[0] == '\0'){
            printf("轮询直到有信息传送！\n");
            if(work_area[0] == '\0'){
                sleep(1);
            }else{
                pthread_mutex_lock(&mutex);
                break;
            }

        }
    }
    exit_time = 0;
    work_area[0] = '\0';
    pthread_mutex_unlock(&mutex);
    pthread_exit(EXIT_SUCCESS);
}

int main(){
    pthread_t thread;
    int res;
    void * thread_res;

    pthread_mutex_init(&mutex, NULL);
    pthread_mutex_lock(&mutex);
    if((res = pthread_create(&thread, NULL, thread_func, NULL)) != 0){
        perror("创建线程失败！");
        exit(EXIT_FAILURE);
    }

    while(exit_time){
        printf("type your string:\n");
        fgets(work_area, 1024, stdin);
        pthread_mutex_unlock(&mutex);
        while(1){
            if(work_area[0] != '\0'){
                puts("数据未被取出");
                sleep(1);
            }else{

```

```
                pthread_mutex_lock(&mutex);
                break;

            }
        }

    }
    pthread_mutex_unlock(&mutex);
    printf("wait for the thread exit!\n");
    if((res = pthread_join(thread, thread_res)) != 0){
        perror("等待线程结束出错! ");
        exit(EXIT_FAILURE);
    }
    printf("thread exited!");
    pthread_mutex_destroy(&mutex);
    exit(EXIT_SUCCESS);
}
```

条件变量

条件变量与一般与互斥量一起使用,实现程序的条件等待

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUM_THREADS  3
#define TCOUNT 10
#define COUNT_LIMIT 12

int     count = 0;
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;

void *inc_count(void *t) //增加count
{
    int i;
    long my_id = (long)t;

    for (i=0; i < TCOUNT; i++)
    {
        pthread_mutex_lock(&count_mutex);
        count++;

        if (count == COUNT_LIMIT) //满足条件后
        {
            printf("inc_count(): thread %ld, count = %d  Threshold reached. ",my_id, count);
            pthread_cond_signal(&count_threshold_cv);//通知
            printf("Just sent signal.\n");
        }
        printf("inc_count(): thread %ld, count = %d, unlocking mutex\n",my_id, count);

        //这里释放锁的同时 sleep 1 秒中可以保证线程2和线程3交替获得锁并执行
```

```
32          pthread_mutex_unlock(&count_mutex);
33          sleep(1);
34      }//end for
35      pthread_exit(NULL);
36  }
37
38  void *watch_count(void *t) //检查条件变量
39  {
40      long my_id = (long)t;
41
42      printf("Starting watch_count(): thread %ld\n", my_id);
43
44      pthread_mutex_lock(&count_mutex);
45      while (count < COUNT_LIMIT)//这里用while防止虚假唤醒
46      {
47          printf("watch_count(): thread %ld Count= %d. Going into wait...\n", my_id,count);
48          pthread_cond_wait(&count_threshold_cv, &count_mutex);//阻塞后自动释放锁
49          printf("watch_count(): thread %ld Condition signal received. Count= %d\n",
    my_id,count);
50          printf("watch_count(): thread %ld Updating the value of count...\n", my_id,count);
51          count += 125;
52          printf("watch_count(): thread %ld count now = %d.\n", my_id, count);
53      }
54      printf("watch_count(): thread %ld Unlocking mutex.\n", my_id);
55      pthread_mutex_unlock(&count_mutex);
56      pthread_exit(NULL);
57  }
58
59  int main(int argc, char *argv[])
60  {
61      int i, rc;
62      long t1=1, t2=2, t3=3;
63      pthread_t threads[3];//3个线程
64      pthread_attr_t attr;//attr
65
66      /* 初始化 mutex 和 condition variable  */
67      pthread_mutex_init(&count_mutex, NULL);
68      pthread_cond_init (&count_threshold_cv, NULL);
69
70      pthread_attr_init(&attr);
71      pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
72      pthread_create(&threads[0], &attr, watch_count, (void *)t1);//线程1关注count值
73      pthread_create(&threads[1], &attr, inc_count, (void *)t2);//线程2增加count值
74      pthread_create(&threads[2], &attr, inc_count, (void *)t3);//线程3增加count值
75
76      for (i = 0; i < NUM_THREADS; i++)
77      {
78          pthread_join(threads[i], NULL);
79      }//等待所有线程完成
80      printf ("Main(): Waited and joined with %d threads. Final value of count = %d. Done.\n",
    NUM_THREADS, count);
81
82      /* Clean up and exit */
```

```
    pthread_attr_destroy(&attr);
    pthread_mutex_destroy(&count_mutex);
    pthread_cond_destroy(&count_threshold_cv);
    pthread_exit (NULL);

}

```

使用pthread实现线程池

声明一个线程池需要线程池大小,线程池的工作队列,线程池的互斥锁与条件变量,线程池是否退出标志信息

```
#ifndef __T_POOL_H__
#define __T_POOL_H__

#include<pthread.h>
#include<ctype.h>
#include<sys/socket.h>
#include<sys/types.h>
#include <sys/select.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include "logging.h"

typedef struct tpool_work_t{
    void* (*work_routine)(int, struct sockaddr_in*, Logger* logger);
    int args;
    struct sockaddr_in *client_addr;
    Logger* logger;
    struct tpool_work_t* next;
}tpool_work;

typedef struct tpool_t{
    size_t shutdown;
    size_t maxnum_thread;
    pthread_t *thread_id;
    tpool_work* tpool_head;
    pthread_cond_t queue_ready;
    pthread_mutex_t queue_lock;
}tpool;
/**
 * @brief:
 *      create thread pool
 * */
int create_tpool(tpool** pool, size_t max_thread_num);
void destroy_tpool(tpool* pool);
int add_task_2_tpool(tpool* pool, void *(*routine)(int, struct sockaddr_in*, Logger*), int
clientFd, struct sockaddr_in *clientAddr, Logger* logger);

#endif
```

实现代码如下

```c
#include "tpool.h"
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

static void *work_routine(void *args)
{
    tpool *pool = (tpool *)args;
    tpool_work *work = NULL;

    while (1)
    {
        pthread_mutex_lock(&pool->queue_lock);
        while (!pool->tpool_head && !pool->shutdown)
        { // if there is no works and pool is not shutdown, it should be suspended for being awake
            pthread_cond_wait(&pool->queue_ready, &pool->queue_lock);
        }

        if (pool->shutdown)
        {
            pthread_mutex_unlock(&pool->queue_lock); //pool shutdown,release the mutex and exit
            pthread_exit(NULL);
        }

        /* tweak a work*/
        work = pool->tpool_head;
        pool->tpool_head = (tpool_work *)pool->tpool_head->next;
        pthread_mutex_unlock(&pool->queue_lock);

        work->work_routine(work->args, work->client_addr, work->logger);

        free(work);
    }
    return NULL;
}

int create_tpool(tpool **pool, size_t max_thread_num)
{
    (*pool) = (tpool *)malloc(sizeof(tpool));
    if (NULL == *pool)
    {
        printf("in %s,malloc tpool failed!,errno = %d,explain:%s\n", __func__, errno, strerror(errno));
        exit(-1);
    }
    (*pool)->shutdown = 0;
    (*pool)->maxnum_thread = max_thread_num;
```

```
    (*pool)->thread_id = (pthread_t *)malloc(sizeof(pthread_t) * max_thread_num);
    if ((*pool)->thread_id == NULL)
    {
        printf("in %s,init thread id failed,errno = %d,explain:%s", __func__, errno,
strerror(errno));
        exit(-1);
    }
    (*pool)->tpool_head = NULL;
    if (pthread_mutex_init(&((*pool)->queue_lock), NULL) != 0)
    {
        printf("in %s,initial mutex failed,errno = %d,explain:%s", __func__, errno,
strerror(errno));
        exit(-1);
    }

    if (pthread_cond_init(&((*pool)->queue_ready), NULL) != 0)
    {
        printf("in %s,initial condition variable failed,errno = %d,explain:%s", __func__,
errno, strerror(errno));
        exit(-1);
    }

    for (int i = 0; i < max_thread_num; i++)
    {
        if (pthread_create(&((*pool)->thread_id[i]), NULL, work_routine, (void *)(*pool))
!= 0)
        {
            printf("pthread_create failed!\n");
            exit(-1);
        }
    }
    return 0;
}

void destroy_tpool(tpool *pool)
{
    tpool_work *tmp_work;

    if (pool->shutdown)
    {
        return;
    }
    pool->shutdown = 1;

    pthread_mutex_lock(&pool->queue_lock);
    pthread_cond_broadcast(&pool->queue_ready);
    pthread_mutex_unlock(&pool->queue_lock);

    for (int i = 0; i < pool->maxnum_thread; i++)
    {
        pthread_join(pool->thread_id[i], NULL);
    }
    free(pool->thread_id);
```

```
    while (pool->tpool_head)
    {
        tmp_work = pool->tpool_head;
        pool->tpool_head = (tpool_work *)pool->tpool_head->next;
        free(tmp_work);
    }

    pthread_mutex_destroy(&pool->queue_lock);
    pthread_cond_destroy(&pool->queue_ready);
    free(pool);
}

int add_task_2_tpool(tpool *pool, void *(*routine)(int, struct sockaddr_in*, Logger*), int clientFd, struct sockaddr_in* clientAddr, Logger* logger)
{
    tpool_work *work, *member;

    if (!routine)
    {
        printf("rontine is null!\n");
        return -1;
    }

    work = (tpool_work *)malloc(sizeof(tpool_work));
    if (!work)
    {
        printf("in %s,malloc work error!,errno = %d,explain:%s\n", __func__, errno, strerror(errno));
        return -1;
    }

    work->work_routine = routine;
    work->args = clientFd;
    work->logger = logger;
    work->client_addr = clientAddr;
    work->next = NULL;

    pthread_mutex_lock(&pool->queue_lock);
    member = pool->tpool_head;
    if (!member)
    {
        pool->tpool_head = work;
    }
    else
    {
        while (member->next)
        {
            member = (tpool_work *)member->next;
        }
        member->next = work;
    }

    //notify the pool that new task arrived!
```

```
    pthread_cond_signal(&pool->queue_ready);
    pthread_mutex_unlock(&pool->queue_lock);
    return 0;
}

```

socket网络程序设计

设计socket程序时,要提前设计好通信协议,socket通信步骤范式十分明显,对服务端程序来说

- 向系统申请套接字(socket调用)
- 设置socket选项(可选setsockopt)
- 绑定socket地址(bind)
- 监听端口(listen)
- 接受请求(accept)
- 处理请求

对客户端来说

- 申请套接字(socket)
- 设置socket选项
- 连接服务器(connect)
- 处理数据

相关函数

分配socket描述符

```
SOCKET(2)                                       Linux Programmer's Manual

NAME
       socket - create an endpoint for communication

SYNOPSIS
       #include <sys/types.h>          /* See NOTES */
       #include <sys/socket.h>

       int socket(int domain, int type, int protocol);
```

出错时返回-1,并error会被设置为错误代码,调用形式如下

```
serverFd = socket(AF_INET, SOCK_STREAM, 0);
```

设置socket选项

```
GETSOCKOPT(2)                                   Linux Programmer's Manual

NAME
       getsockopt, setsockopt - get and set options on sockets

SYNOPSIS
       #include <sys/types.h>          /* See NOTES */
       #include <sys/socket.h>

       int getsockopt(int sockfd, int level, int optname,
                      void *optval, socklen_t *optlen);
       int setsockopt(int sockfd, int level, int optname,
                      const void *optval, socklen_t optlen);
```

出错时返回-1,成功返回0,sockfd表示套接字描述符,level表示设置选项的level,socket操作时一般设置为SOL_SOCKET, optval是一个指针,指向要设置选项的值,optlen是optval的大小

调用示例

```
1  int enable = 1;
2  setsockopt(server->serverFd, SOL_SOCKET, SO_REUSEADDR, &enable, sizeof(int));
```

解析地址

由于字节序的问题,设置socket地址时,需要将主机字节序的数据,转换为网络字节序数据,htons可以将本机整型的数据转换为网络整形,用于转换端口.gethostbyname可以将字符串形式表示的主机名称解析为网络字节序的ip地址

```
GETHOSTBYNAME(3)                          Linux Programmer's Manual                                GETHOSTBYN

NAME
       gethostbyname,  gethostbyaddr,  sethostent, gethostent, endhostent, h_errno, herror, hstrerror, gethostbye
       gethostbyname2, gethostbyname2_r, gethostbyname_r, gethostent_r - get network host entry

SYNOPSIS
       #include <netdb.h>
       extern int h_errno;

       struct hostent *gethostbyname(const char *name);

       #include <sys/socket.h>       /* for AF_INET */
       struct hostent *gethostbyaddr(const void *addr,
                                     socklen_t len, int type);
```

```
The hostent structure is defined in <netdb.h> as follows:

    struct hostent {
        char  *h_name;            /* official name of host */
        char **h_aliases;         /* alias list */
        int    h_addrtype;        /* host address type */
        int    h_length;          /* length of address */
        char **h_addr_list;       /* list of addresses */
    }
    #define h_addr h_addr_list[0] /* for backward compatibility */

The members of the hostent structure are:

h_name The official name of the host.

h_aliases
       An array of alternative names for the host, terminated by a null pointer.

h_addrtype
       The type of address; always AF_INET or AF_INET6 at present.

h_length
       The length of the address in bytes.

h_addr_list
       An  array  of  pointers  to  network addresses for the host (in network byte order), terminated by a null
       pointer.

h_addr The first address in h_addr_list for backward compatibility.
```

失败时返回null,herror与hstrerror分别从h_errno中获取错误信息

```
xiaozhi
BYTEORDER(3)                                Linux Programmer's Manual

NAME
       htonl, htons, ntohl, ntohs - convert values between host and network byte order

SYNOPSIS
       #include <arpa/inet.h>

       uint32_t htonl(uint32_t hostlong);

       uint16_t htons(uint16_t hostshort);

       uint32_t ntohl(uint32_t netlong);

       uint16_t ntohs(uint16_t netshort);
```

调用范例

```
1  struct hostent *hostinfo;
2  hostinfo = gethostbyname(hostname);
3  server->serverAddress->sin_port = htons(port);
4  server->serverAddress->sin_family = AF_INET;
5  server->serverAddress->sin_addr = *(struct in_addr *)*hostinfo->h_addr_list;
```

inet_ntoa将网络字节序的主机地址解析为本机可读的字符串,ntohs,将网络主机端口转换为整数

示例

```
1  fprintf(stdout, "get connection from : (%s, %d) with fd: %d\n", inet_ntoa(clientAddress-
   >sin_addr), ntohs(clientAddress->sin_port), clientFd);
```

getaddrinfo 组合了gethostbyname与getservbyname的功能.

inet_pton将字符串类型的ip地址转换为网络地址

绑定地址

```
SYNOPSIS
       #include <sys/types.h>          /* See NOTES */
       #include <sys/socket.h>

       int bind(int sockfd, const struct sockaddr *addr,
                socklen_t addrlen);
```

失败返回-1,成功返回0

调用范例

```
bind(server->serverFd, (struct sockaddr *)(server->serverAddress), sizeof(struct sockaddr_in)
```

接受请求

```
SYNOPSIS
       #include <sys/socket.h>

       int accept(int socket, struct sockaddr *restrict address,
           socklen_t *restrict address_len);
```

失败返回-1,成功的话返回接收到的套接字描述符

调用示例

```
struct sockaddr_in *clientAddress = (struct sockaddr_in *)malloc(sizeof(struct sockaddr_in));
int clientLen = sizeof(clientAddress);
clientFd = accept(server->serverFd, (struct sockaddr *)clientAddress, (socklen_t * __restrict)
& clientLen);
```

异步编程之 select

c程序中read与write操作默认阻塞程序执行,要实现异步socket通信,需要使用检测机制,判断socket是否有数据接收,或是数据是否可以发送,最原始的检测手段为select系统调用,win与linux通用

```
NAME
       select – synchronous I/O multiplexing

SYNOPSIS
       #include <sys/select.h>

       int select(int nfds, fd_set *restrict readfds,
           fd_set *restrict writefds, fd_set *restrict errorfds,
           struct timeval *restrict timeout);
```

参数分别表示要检测的文件描述符的个数,可读检测列表,可写检测列表,异常列表,与阻塞等待的时间,传入0表示非阻塞,立即返回

与select有关的一些系统调用被封装到宏中

```
NAME
       select, pselect, FD_CLR, FD_ISSET, FD_SET, FD_ZERO - synchronous I/O multiplexing

SYNOPSIS
       /* According to POSIX.1-2001, POSIX.1-2008 */
       #include <sys/select.h>

       /* According to earlier standards */
       #include <sys/time.h>
       #include <sys/types.h>
       #include <unistd.h>

       int select(int nfds, fd_set *readfds, fd_set *writefds,
                  fd_set *exceptfds, struct timeval *timeout);

       void FD_CLR(int fd, fd_set *set);
       int  FD_ISSET(int fd, fd_set *set);
       void FD_SET(int fd, fd_set *set);
       void FD_ZERO(fd_set *set);
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
    fd_set rfds;
    struct timeval tv;
    int retval;

    /* Watch stdin (fd 0) to see when it has input. */

    FD_ZERO(&rfds);
    FD_SET(0, &rfds);

    /* Wait up to five seconds. */

    tv.tv_sec = 5;
    tv.tv_usec = 0;
    retval = select(1, &rfds, NULL, NULL, &tv);
    /* Don't rely on the value of tv now! */

    if (retval == -1)
        perror("select()");
    else if (retval)
        printf("Data is available now.\n");
    /* FD_ISSET(0, &rfds) will be true. */
    else
        printf("No data within five seconds.\n");
```

```
32
33      exit(EXIT_SUCCESS);
34  }
```

异步编程之epoll

相关的系统调用,epoll_create,epoll_ctl,epoll_wait,create用于创建一个用于处理其他描述符事件的描述符句柄,ctl用于修改要监听的文件描述符的事件,wait用于等待事件.相关的数据结构如下

```
1   typedef union epoll_data {
2       void        *ptr;
3       int          fd;
4       uint32_t     u32;
5       uint64_t     u64;
6   } epoll_data_t;
7
8   struct epoll_event {
9       uint32_t     events;      /* Epoll events */
10      epoll_data_t data;        /* User data variable */
11  };
12
```

events是预先定义的一组宏

```
1         The events member is a bit mask composed by ORing together zero or more of the
      following available event types:
2
3         EPOLLIN
4                The associated file is available for read(2) operations.
5
6         EPOLLOUT
7                The associated file is available for write(2) operations.
8
9         EPOLLRDHUP (since Linux 2.6.17)
10               Stream socket peer closed connection, or shut down writing half of connection.
      (This flag is  especially useful for writing simple code to detect peer shutdown when using
      Edge Triggered monitoring.)
11
12        EPOLLPRI
13               There is an exceptional condition on the file descriptor.  See the discussion
      of POLLPRI in poll(2).
14
15        EPOLLERR
16               Error  condition  happened  on the associated file descriptor.  This event is
      also reported for the writeend of a pipe when the read end has been closed.  epoll_wait(2)
      will always report for this event; it  is not necessary to set it in events.
17
```

epoll_ctl声明如下

```
NAME
       epoll_ctl - control interface for an epoll file descriptor

SYNOPSIS
       #include <sys/epoll.h>

       int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
```

其中op可选的操作有如下三种

```
EPOLL_CTL_ADD
       Register the target file descriptor fd on the epoll instance referred to by the file descriptor epfd  and
       associate the event event with the internal file linked to fd.

EPOLL_CTL_MOD
       Change the event event associated with the target file descriptor fd.

EPOLL_CTL_DEL
       Remove (deregister) the target file descriptor fd from the epoll instance referred to by epfd.  The event
       is ignored and can be NULL (but see BUGS below).
```

分别用于增加,修改,删除在文件描述符上监听的事件.

epoll_create声明如下

```
NAME
       epoll_create, epoll_create1 - open an epoll file descriptor

SYNOPSIS
       #include <sys/epoll.h>

       int epoll_create(int size);
```

返回创建成功的描述符,如果失败的话返回-1,根据manual文档的描述,size参数当前已经变为可选项,因为之前版本的epoll_create使用size参数来分配存放监听描述符的内存,现在操作系统可以动态分配这部分内存,但是size参数需要大于0,以和之前版本的epoll兼容.

epoll_wait声明如下

```
NAME
       epoll_wait, epoll_pwait - wait for an I/O event on an epoll file descriptor

SYNOPSIS
       #include <sys/epoll.h>

       int epoll_wait(int epfd, struct epoll_event *events,int maxevents, int timeout);
```

参数	含义
epfd	epoll_create返回的描述符
events	存放epoll结果的事件指针
maxevents	最多返回事件的个数
timeout	超时时间(-1表示阻塞,0表示立即返回,单位是毫秒)

一个简单的例子

server.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <errno.h>
#include <sys/socket.h>
#include <netinet/in.h>         /* socket类定义需要*/
#include <sys/epoll.h>          /* epoll头文件 */
#include <fcntl.h>                  /* nonblocking需要 */
#include <sys/resource.h>       /* 设置最大的连接数需要setrlimit */

#define MAXEPOLL    10000   /* 对于服务器来说，这个值可以很大的！ */
#define MAXLINE     1024
#define     PORT            6000
#define MAXBACK 1000

//!> 设置非阻塞
//!>
int setnonblocking( int fd )
{
    if( fcntl( fd, F_SETFL, fcntl( fd, F_GETFD, 0 )|O_NONBLOCK ) == -1 )
    {
        printf("Set blocking error : %d\n", errno);
        return -1;
    }
    return 0;
}

int main( int argc, char ** argv )
{
    int         listen_fd;
    int         conn_fd;
    int         epoll_fd;
    int         nread;
    int         cur_fds;                //!> 当前已经存在的数量
    int         wait_fds;               //!> epoll_wait 的返回值
    int     i;
    struct sockaddr_in servaddr;
```

```
    struct sockaddr_in cliaddr;
    struct  epoll_event ev;
    struct  epoll_event evs[MAXEPOLL];
    struct  rlimit  rlt;        //!> 设置连接数所需
    char    buf[MAXLINE];
    socklen_t   len = sizeof( struct sockaddr_in );

    //!> 设置每个进程允许打开的最大文件数
    //!> 每个主机是不一样的哦，一般服务器应该很大吧!
    //!>
    rlt.rlim_max = rlt.rlim_cur = MAXEPOLL;
    if( setrlimit( RLIMIT_NOFILE, &rlt ) == -1 )
    {
        printf("Setrlimit Error : %d\n", errno);
        exit( EXIT_FAILURE );
    }

    //!> server 套接口
    //!>
    bzero( &servaddr, sizeof( servaddr ) );
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl( INADDR_ANY );
    servaddr.sin_port = htons( PORT );

    //!> 建立套接字
    if( ( listen_fd = socket( AF_INET, SOCK_STREAM, 0 ) ) == -1 )
    {
        printf("Socket Error...\n" , errno );
        exit( EXIT_FAILURE );
    }

    //!> 设置非阻塞模式
    //!>
    if( setnonblocking( listen_fd ) == -1 )
    {
        printf("Setnonblocking Error : %d\n", errno);
        exit( EXIT_FAILURE );
    }

    //!> 绑定
    //!>
    if( bind( listen_fd, ( struct sockaddr *)&servaddr, sizeof( struct sockaddr ) ) == -1 )
    {
        printf("Bind Error : %d\n", errno);
        exit( EXIT_FAILURE );
    }

    //!> 监听
    //!>
    if( listen( listen_fd, MAXBACK ) == -1 )
    {
        printf("Listen Error : %d\n", errno);
        exit( EXIT_FAILURE );
```

```
93      }
94
95      //!> 创建epoll
96      //!>
97      epoll_fd = epoll_create( MAXEPOLL );    //!> create
98      ev.events = EPOLLIN | EPOLLET;      //!> accept Read!
99      ev.data.fd = listen_fd;                 //!> 将listen_fd 加入
100     if( epoll_ctl( epoll_fd, EPOLL_CTL_ADD, listen_fd, &ev ) < 0 )
101     {
102         printf("Epoll Error : %d\n", errno);
103         exit( EXIT_FAILURE );
104     }
105     cur_fds = 1;
106
107     while( 1 )
108     {
109         if( ( wait_fds = epoll_wait( epoll_fd, evs, cur_fds, -1 ) ) == -1 )
110         {
111             printf( "Epoll Wait Error : %d\n", errno );
112             exit( EXIT_FAILURE );
113         }
114
115         for( i = 0; i < wait_fds; i++ )
116         {
117             if( evs[i].data.fd == listen_fd && cur_fds < MAXEPOLL )
118                                                 //!> if是监听端口有事
119             {
120                 if( ( conn_fd = accept( listen_fd, (struct sockaddr *)&cliaddr, &len ) ) ==
-1 )
121                 {
122                     printf("Accept Error : %d\n", errno);
123                     exit( EXIT_FAILURE );
124                 }
125
126                 printf( "Server get from client !\n"/*,  inet_ntoa(cliaddr.sin_addr),
cliaddr.sin_port */);
127
128                 ev.events = EPOLLIN | EPOLLET;      //!> accept Read!
129                 ev.data.fd = conn_fd;                   //!> 将conn_fd 加入
130                 if( epoll_ctl( epoll_fd, EPOLL_CTL_ADD, conn_fd, &ev ) < 0 )
131                 {
132                     printf("Epoll Error : %d\n", errno);
133                     exit( EXIT_FAILURE );
134                 }
135                 ++cur_fds;
136                 continue;
137             }
138
139             //!> 下面处理数据
140             //!>
141             nread = read( evs[i].data.fd, buf, sizeof( buf ) );
142             if( nread <= 0 )                        //!> 结束后者出错
143             {
```

```
                close( evs[i].data.fd );
                epoll_ctl( epoll_fd, EPOLL_CTL_DEL, evs[i].data.fd, &ev );  //!> 删除计入的
fd
                --cur_fds;                  //!> 减少一个呗!
                continue;
            }

            write( evs[i].data.fd, buf, nread );            //!> 回写

        }
    }

    close( listen_fd );
    return 0;
}
```

client.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <sys/socket.h>
#include  <arpa/inet.h>
#include <sys/select.h>

#define MAXLINE 1024
#define SERV_PORT 6000

//!> 注意输入是由stdin, 接受是由server发送过来
//!> 所以在client端也是需要select进行处理的
void send_and_recv( int connfd )
{
    FILE * fp = stdin;
    int   lens;
    char send[MAXLINE];
    char recv[MAXLINE];
    fd_set rset;
    FD_ZERO( &rset );
    int maxfd = ( fileno( fp ) > connfd ? fileno( fp ) : connfd  + 1 );
                                    //!> 输入和输出的最大值
    int n;

    while( 1 )
    {
        FD_SET( fileno( fp ), &rset );
        FD_SET( connfd, &rset );            //!> 注意不要把rset看作是简单的一个变量
                                //!> 注意它其实是可以包含一组套接字的哦,
                                //!> 相当于是封装的数组! 每次都要是新的哦!
```

```
35
36          if( select( maxfd, &rset, NULL, NULL, NULL ) == -1 )
37          {
38              printf("Client Select Error..\n");
39              exit(EXIT_FAILURE  );
40          }
41
42          //!> if 连接口有信息
43          if( FD_ISSET( connfd, &rset ) ) //!> if 连接端口有信息
44          {
45              printf( "client get from server ...\n" );
46              memset( recv, 0, sizeof( recv ) );
47              n = read( connfd, recv, MAXLINE );
48              if( n == 0 )
49              {
50                  printf("Recv ok...\n");
51                  break;
52              }
53              else if( n == -1 )
54              {
55                  printf("Recv error...\n");
56                  break;
57              }
58              else
59              {
60                  lens = strlen( recv );
61                  recv[lens] = '\0';
62                  //!> 写到stdout
63                  write( STDOUT_FILENO, recv, MAXLINE );
64                  printf("\n");
65              }
66
67          }
68
69          //!> if 有stdin输入
70          if( FD_ISSET( fileno( fp ), &rset ) )   //!> if 有输入
71          {
72              //!> printf("client stdin ...\n");
73
74              memset( send, 0, sizeof( send ) );
75              if( fgets( send, MAXLINE, fp ) == NULL )
76              {
77                  printf("End...\n");
78                  exit( EXIT_FAILURE );
79              }
80              else
81              {
82                  //!>if( str )
83                  lens = strlen( send );
84                  send[lens-1] = '\0';        //!> 减一的原因是不要回车字符
85                                  //!> 经验值：这一步非常重要的哦！！！！！！！
86                  if( strcmp( send, "q" ) == 0 )
87                  {
```

```
                    printf( "Bye..\n" );
                    return;
                }

                printf("Client send : %s\n", send);
                write( connfd, send, strlen( send ) );
            }
        }

    }

}

int main( int argc, char ** argv )
{
    //!> char * SERV_IP = "10.30.97.188";
    char    buf[MAXLINE];
    int     connfd;
    struct sockaddr_in servaddr;

    if( argc != 2 )
    {
        printf("Input server ip !\n");
        exit( EXIT_FAILURE );
    }

    //!> 建立套接字
    if( ( connfd = socket( AF_INET, SOCK_STREAM, 0 ) ) == -1 )
    {
        printf("Socket Error...\n" , errno );
        exit( EXIT_FAILURE );
    }

    //!> 套接字信息
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(SERV_PORT);
    inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

    //!> 链接server
    if( connect( connfd, ( struct sockaddr *  )&servaddr, sizeof( servaddr ) ) < 0 )
    {
        printf("Connect error..\n");
        exit(EXIT_FAILURE);
    }
    /*else
    {
        printf("Connet ok..\n");
    }*/

    //!>
    //!> send and recv
    send_and_recv( connfd );
```

```
141
142     //!>
143
144     close( connfd );
145     printf("Exit\n");
146
147     return 0;
148 }
```

epoll与select和poll先比性能更高,使用事件回调的机制,减小了操作系统的负担,推荐使用

设计模式

使用多线程方式,创建两个消息队列,一个待发送队列,一个接收队列,所有网络操作放到一个线程中进行,使用锁保持线程同步

日志记录与错误输出

相关函数

错误输出相关的函数perror,strerror.

当linux系统调用执行失败的时候,会在errno全局变量记录错误代码,perror函数会根据该错误代码,将错误信息输出到标准输出. strerror会根据参数表示的错误代码,返回该代码的字符串描述

用法大致如下

```
NAME
       perror - print a system error message

SYNOPSIS
       #include <stdio.h>

       void perror(const char *s);

       #include <errno.h>

       const char * const sys_errlist[];
       int sys_nerr;
       int errno;       /* Not really declared this way; see errno(3) */
```

```
NAME
       strerror, strerror_r, strerror_l - return string describing error number

SYNOPSIS
       #include <string.h>

       char *strerror(int errnum);
```

errno是全局的记录错误代码的变量,传入strerror函数中,获取错误字符串的字符串描述

```
fprintf(stderr, "read file failed: %s\n", strerror(errno))
```

记录源文件信息

定义如下宏

```
#define LOG_STDOUT(format, ...) fprintf(stdout, "[%s@%s %d]: " format "\n", __func__, __FILE__, __LINE__, ##__VA_ARGS__)
#define LOG_ERROR(format, ...) fprintf(stderr, "[%s@%s %d]: " format "\n", __func__, __FILE__, __LINE__, ##__VA_ARGS__)
```

在使用时,调用相关宏定义,输出函数名称,文件名称,行号等信息

```
LOG_ERROR( "cannot parse http header information!");
```

解析命令行参数

cmake项目管理

gcc与gdb调试教程