# 模块设计

## 二级标题

### 三级标题

1. 子项1
2. 子项2
3. 子项3
4. 列表项1。
5. 列表项2。

# 连接管理模块

## 概要设计

# 文件句柄管理

## Select

系统调用select用于监视多个文件句柄的状态变化。Select调用会发生阻塞，直到被监视的文件句柄有一个或多个发生了状态改变。

select函数原型如下：

int select (int nfds, fd\_set \*readfds, fd\_set \*writefds, fd\_set \*exceptfds, struct timeval \*timeout);

函数的最后一个参数timeout是一个超时时间值。其类型是struct timeval \*，即一个struct timeval结构的变量的指针，所以我们在程序里要声明一个struct timeval tv;然后把变量tv的地址&tv传递给select函数。struct timeval结构如下：

struct timeval

{

long tv\_sec; //seconds

long tv\_usec; //microseconds

}；

第2、3、4三个参数是一样的类型;fd\_set \*,即我们在程序里要申请几个fd\_set类型的变量，比如rdfds，wtfds，exfds，然后把这个变量的地址&rdfds,&wtfds,&exfds传递给select函数。这三个参数都是一个句柄的集合，第一个rdfds是用来保存这样的句柄的:当句柄的状态变成可读时系统就告诉select函数返回，同理第二个函数是指向有句柄状态变成可写时系统就会告诉select函数返回，同理第三个参数exfds是特殊情况，即句柄上有特殊情况发生时系统会告诉select函数返回。特殊情况比如对方通过一个socket句柄发来了紧急数据。如果我们程序里只想检测某个socket是否有数据可读，我们可以这样：

fd\_set rdfds;

struct timeval tv;

int ret;

FD\_ZERO(&rdfds);

FD\_SET(socket, &rdfds);

tv.tv\_sec = 1;

tv.tv\_uses = 500;

ret = select (socket + 1, &rdfds, NULL, NULL, &tv);

if(ret < 0) perror (“select”);

else if (ret = = 0) printf(“time out”);

else {

printf(“ret = %d/n”,ret);

if(FD\_ISSET(socket, &rdfds)){

/\* 读取socket句柄里的数据 \*/

recv( );

}

}

注意select函数的第一个参数，是所有加入集合的句柄值的最大那个那个值还要加1.比如我们创建了3个句柄;

int sa, sb, sc;

sa = socket(……);

connect (sa,….);

sb = socket(….);

connect (sb,…);

sc = socket(….);

connect(sc,…);

FD\_SET(sa, &rdfds);

FD\_SET(sb, &rdfds);

FD\_SET(sc, &rdfds);

在使用select函数之前，一定要找到3个句柄中的最大值是哪个，我们一般定义一个变量来保存最大值，取得最大socket值如下：

int maxfd = 0;

if(sa > maxfd) maxfd = sa;

if(sb > maxfd) maxfd = sb;

if(sc > maxfd) maxfd = sc;

然后调用select函数：

ret = select (maxfd+1, &rdfds, NULL, NULL,&tv);

同样的道理，如果我们是检测用户是否按了键盘进行输入，我们就应该把标准输入0这个句柄放到select里来检测，如下：

FD\_ZERO(&rdfds);

FD\_SET(0, &rdfds);

tv.tv\_sec = 1;

tv.tv\_usec = 0;

ret = select (1, &rdfds,NULL,NULL,&tv);

if(ret < 0) perror(“select”);

else if (ret = = 0) printf (“time out/n”);

else{

scanf(“%s”,buf);

}

## Epoll

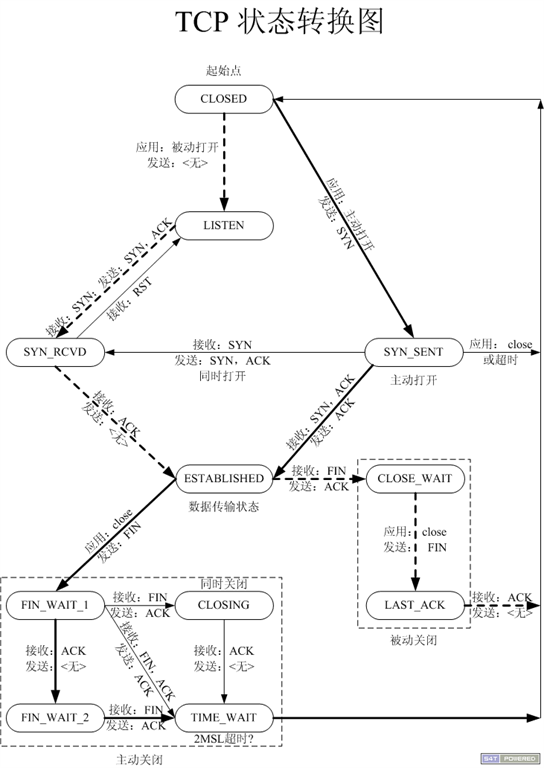
epoll - I/O event notification facility  
  
在linux的网络编程中，很长的时间都在使用select来做事件触发。在linux新的内核中，有了一种替换它的机制，就是epoll。  
相比于select，epoll最大的好处在于它不会随着监听fd数目的增长而降低效率。因为在内核中的select实现中，它是采用轮询来处理的，轮询的fd数目越多，自然耗时越多。并且，在linux/posix\_types.h头文件有这样的声明：  
#define \_\_FD\_SETSIZE    1024  
表示select最多同时监听1024个fd，当然，可以通过修改头文件再重编译内核来扩大这个数目，但这似乎并不治本。  
  
epoll的接口非常简单，一共就三个函数：  
1. int epoll\_create(int size);  
创 建一个epoll的句柄，size用来告诉内核这个监听的数目一共有多大。这个参数不同于select()中的第一个参数，给出最大监听的fd+1的值。 需要注意的是，当创建好epoll句柄后，它就是会占用一个fd值，在linux下如果查看/proc/进程id/fd/，是能够看到这个fd的，所以在 使用完epoll后，必须调用close()关闭，否则可能导致fd被耗尽。  
  
  
2. int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event);  
epoll的事件注册函数，它不同与select()是在监听事件时告诉内核要监听什么类型的事件，而是在这里先注册要监听的事件类型。第一个参数是epoll\_create()的返回值，第二个参数表示动作，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；  
第三个参数是需要监听的fd，第四个参数是告诉内核需要监听什么事，struct epoll\_event结构如下：  
  
typedef union epoll\_data {  
    void \*ptr;  
    int fd;  
    \_\_uint32\_t u32;  
    \_\_uint64\_t u64;  
} epoll\_data\_t;  
  
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（这里应该表示有带外数据到来）；  
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，这是相对于水平触发(Level Triggered)来说的。  
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  
  
  
3. int epoll\_wait(int epfd, struct epoll\_event \* events, int maxevents, int timeout);  
等 待事件的产生，类似于select()调用。参数events用来从内核得到事件的集合，maxevents告之内核这个events有多大，这个 maxevents的值不能大于创建epoll\_create()时的size，参数timeout是超时时间（毫秒，0会立即返回，-1将不确定，也有 说法说是永久阻塞）。该函数返回需要处理的事件数目，如返回0表示已超时。  
  
  
4、关于ET、LT两种工作模式：  
可以得出这样的结论:  
ET 模式仅当状态发生变化的时候才获得通知,这里所谓的状态的变化并不包括缓冲区中还有未处理的数据,也就是说,如果要采用ET模式,需要一直 read/write直到出错为止,很多人反映为什么采用ET模式只接收了一部分数据就再也得不到通知了,大多因为这样;而LT模式是只要有数据没有处理 就会一直通知下去的.  
  
那么究竟如何来使用epoll呢？其实非常简单。  
通过在包含一个头文件#include <sys/epoll.h> 以及几个简单的API将可以大大的提高你的网络服务器的支持人数。  
  
首 先通过create\_epoll(int maxfds)来创建一个epoll的句柄，其中maxfds为你epoll所支持的最大句柄数。这个函数会返回一个新的epoll句柄，之后的所有操作 将通过这个句柄来进行操作。在用完之后，记得用close()来关闭这个创建出来的epoll句柄。  
  
之后在你的网络主循环里面，每一帧的调用epoll\_wait(int epfd, epoll\_event events, int max events, int timeout)来查询所有的网络接口，看哪一个可以读，哪一个可以写了。基本的语法为：  
nfds = epoll\_wait(kdpfd, events, maxevents, -1);  
其 中kdpfd为用epoll\_create创建之后的句柄，events是一个epoll\_event\*的指针，当epoll\_wait这个函数操作成功 之后，epoll\_events里面将储存所有的读写事件。max\_events是当前需要监听的所有socket句柄数。最后一个timeout是 epoll\_wait的超时，为0的时候表示马上返回，为-1的时候表示一直等下去，直到有事件范围，为任意正整数的时候表示等这么长的时间，如果一直没 有事件，则范围。一般如果网络主循环是单独的线程的话，可以用-1来等，这样可以保证一些效率，如果是和主逻辑在同一个线程的话，则可以用0来保证主循环 的效率。  
  
epoll\_wait范围之后应该是一个循环，遍历所有的事件。  
  
几乎所有的epoll程序都使用下面的框架：  
  
    for( ; ; )  
    {  
        nfds = epoll\_wait(epfd,events,20,500);  
        for(i=0;i<nfds;++i)  
        {  
            if(events[i].data.fd==listenfd) //有新的连接  
            {  
                connfd = accept(listenfd,(sockaddr \*)&clientaddr, &clilen); //accept这个连接  
                ev.data.fd=connfd;  
                ev.events=EPOLLIN|EPOLLET;  
                epoll\_ctl(epfd,EPOLL\_CTL\_ADD,connfd,&ev); //将新的fd添加到epoll的监听队列中  
            }  
            else if( events[i].events&EPOLLIN ) //接收到数据，读socket  
            {  
                n = read(sockfd, line, MAXLINE)) < 0    //读  
                ev.data.ptr = md;     //md为自定义类型，添加数据  
                ev.events=EPOLLOUT|EPOLLET;  
                epoll\_ctl(epfd,EPOLL\_CTL\_MOD,sockfd,&ev);//修改标识符，等待下一个循环时发送数据，异步处理的精髓  
            }  
            else if(events[i].events&EPOLLOUT) //有数据待发送，写socket  
            {  
                struct myepoll\_data\* md = (myepoll\_data\*)events[i].data.ptr;    //取数据  
                sockfd = md->fd;  
                send( sockfd, md->ptr, strlen((char\*)md->ptr), 0 );        //发送数据  
                ev.data.fd=sockfd;  
                ev.events=EPOLLIN|EPOLLET;  
                epoll\_ctl(epfd,EPOLL\_CTL\_MOD,sockfd,&ev); //修改标识符，等待下一个循环时接收数据  
            }  
            else  
            {  
                //其他的处理  
            }  
        }  
    }  
  
  
  
下面给出一个完整的服务器端例子：

|  |
| --- |
| #include <iostream> #include <sys/socket.h> #include <sys/epoll.h> #include <netinet/in.h> #include <arpa/inet.h> #include <fcntl.h> #include <unistd.h> #include <stdio.h> #include <errno.h>  using namespace std;  #define MAXLINE 5 #define OPEN\_MAX 100 #define LISTENQ 20 #define SERV\_PORT 5000 #define INFTIM 1000  void setnonblocking(int sock) {     int opts;     opts=fcntl(sock,F\_GETFL);     if(opts<0)     {         perror("fcntl(sock,GETFL)");         exit(1);     }     opts = opts|O\_NONBLOCK;     if(fcntl(sock,F\_SETFL,opts)<0)     {         perror("fcntl(sock,SETFL,opts)");         exit(1);     } }  int main(int argc, char\* argv[]) {     int i, maxi, listenfd, connfd, sockfd,epfd,nfds, portnumber;     ssize\_t n;     char line[MAXLINE];     socklen\_t clilen;       if ( 2 == argc )     {         if( (portnumber = atoi(argv[1])) < 0 )         {             fprintf(stderr,"Usage:%s portnumber/a/n",argv[0]);             return 1;         }     }     else     {         fprintf(stderr,"Usage:%s portnumber/a/n",argv[0]);         return 1;     }        //声明epoll\_event结构体的变量,ev用于注册事件,数组用于回传要处理的事件      struct epoll\_event ev,events[20];     //生成用于处理accept的epoll专用的文件描述符      epfd=epoll\_create(256);     struct sockaddr\_in clientaddr;     struct sockaddr\_in serveraddr;     listenfd = socket(AF\_INET, SOCK\_STREAM, 0);     //把socket设置为非阻塞方式      //setnonblocking(listenfd);      //设置与要处理的事件相关的文件描述符      ev.data.fd=listenfd;     //设置要处理的事件类型      ev.events=EPOLLIN|EPOLLET;     //ev.events=EPOLLIN;      //注册epoll事件      epoll\_ctl(epfd,EPOLL\_CTL\_ADD,listenfd,&ev);     bzero(&serveraddr, sizeof(serveraddr));     serveraddr.sin\_family = AF\_INET;     char \*local\_addr="127.0.0.1";     inet\_aton(local\_addr,&(serveraddr.sin\_addr));//htons(portnumber);      serveraddr.sin\_port=htons(portnumber);     bind(listenfd,(sockaddr \*)&serveraddr, sizeof(serveraddr));     listen(listenfd, LISTENQ);     maxi = 0;     for ( ; ; ) {         //等待epoll事件的发生          nfds=epoll\_wait(epfd,events,20,500);         //处理所发生的所有事件          for(i=0;i<nfds;++i)         {             if(events[i].data.fd==listenfd)//如果新监测到一个SOCKET用户连接到了绑定的SOCKET端口，建立新的连接。              {                 connfd = accept(listenfd,(sockaddr \*)&clientaddr, &clilen);                 if(connfd<0){                     perror("connfd<0");                     exit(1);                 }                 //setnonblocking(connfd);                  char \*str = inet\_ntoa(clientaddr.sin\_addr);                 cout << "accapt a connection from " << str << endl;                 //设置用于读操作的文件描述符                  ev.data.fd=connfd;                 //设置用于注测的读操作事件                  ev.events=EPOLLIN|EPOLLET;                 //ev.events=EPOLLIN;                  //注册ev                  epoll\_ctl(epfd,EPOLL\_CTL\_ADD,connfd,&ev);             }             else if(events[i].events&EPOLLIN)//如果是已经连接的用户，并且收到数据，那么进行读入。              {                 cout << "EPOLLIN" << endl;                 if ( (sockfd = events[i].data.fd) < 0)                     continue;                 if ( (n = read(sockfd, line, MAXLINE)) < 0) {                     if (errno == ECONNRESET) {                         close(sockfd);                         events[i].data.fd = -1;                     } else                         std::cout<<"readline error"<<std::endl;                 } else if (n == 0) {                     close(sockfd);                     events[i].data.fd = -1;                 }                 line[n] = '/0';                 cout << "read " << line << endl;                 //设置用于写操作的文件描述符                  ev.data.fd=sockfd;                 //设置用于注测的写操作事件                  ev.events=EPOLLOUT|EPOLLET;                 //修改sockfd上要处理的事件为EPOLLOUT                  //epoll\_ctl(epfd,EPOLL\_CTL\_MOD,sockfd,&ev);              }             else if(events[i].events&EPOLLOUT) // 如果有数据发送              {                 sockfd = events[i].data.fd;                 write(sockfd, line, n);                 //设置用于读操作的文件描述符                  ev.data.fd=sockfd;                 //设置用于注测的读操作事件                  ev.events=EPOLLIN|EPOLLET;                 //修改sockfd上要处理的事件为EPOLIN                  epoll\_ctl(epfd,EPOLL\_CTL\_MOD,sockfd,&ev);             }         }     }     return 0; } |

## 管道

管道：当从一个进程连接数据流到另一个进程时，使用术语管道（pipe）。  
  
＃include <unistd.h>  
int pipe(int filedes[2]); //创建管道  
  
pipe()说明:  
返回值：0成功，-1出错。  
如果调用成功，则进程此时由了两个额外的打开文件描述符，filedes[0]中的值是管道的读取端，而filedes[1]是管道的写入端。  
  
#include<unistd.h>  
#include<sys/types.h>  
#include<errno.h>  
#include<stdio.h>  
#include<stdlib.h>  
  
int main(){  
    int pipe\_fd[2];  
    pid\_t pid;  
    char buf\_r[100];  
    char \*p\_wbuf;  
    int r\_num;  
  
    memset(buf\_r,0,sizeof(buf\_r));  
  
    //创建管道  
    if(pipe(pipe\_fd)<0){  
        printf("pipe create error\n");  
        return -1;  
    }  
  
    if((pid=fork())==0){//表示在子进程中  
        printf("\n");  
        //关闭管道写描述符，进行管道读操作  
        close(pipe\_fd[1]);  
        sleep(2);  
        //管道描述符中读取  
        if((r\_num=read(pipe\_fd[0],buf\_r,100))>0){  
            printf("%d numbers read from the pipe is %s\n",r\_num,buf\_r);  
        }  
        close(pipe\_fd[0]);  
        exit(0);  
    }  
    else if(pid>0){//表示在父进程中，父进程写  
    //关闭管道读描述符，进行管道写操作  
        close(pipe\_fd[0]);  
        if(write(pipe\_fd[1],"Hello",5)!=-1)  
            printf("parent write1 success!\n");  
        if(write(pipe\_fd[1],"Pipe",5)!=1)  
            printf("parent write2 success!\n");  
        close(pipe\_fd[1]);  
        sleep(3);  
        waitpid(pid,NULL,0);  
        exit(0);  
    }  
}  
      
管道读写注意事项：  
1.必须在系统调用fork()中调用pipe()，否则子进程将不会继承文件描述符；  
2.当使用半双工管道时，任何关联的进程都必须共享一个相关的祖先进程。

# TCP状态转换



这个图n多人都知道，它对排除和定位网络或系统故障时大有帮助，但是怎样牢牢地将这张图刻在脑中呢？那么你就一定要对这张图的每一个状态，及转换的过程有深刻地认识，不能只停留在一知半解之中。下面对这张图的11种状态详细解释一下，以便加强记忆！不过在这之前，先回顾一下TCP建立连接的三次握手过程，以及关闭连接的四次握手过程。

1、建立连接协议（三次握手）  
 （1）客户端发送一个带SYN标志的TCP报文到服务器。这是三次握手过程中的报文1。

 （2） 服务器端回应客户端的，这是三次握手中的第2个报文，这个报文同时带ACK标志和SYN标志。因此它表示对刚才客户端SYN报文的回应；同时又标志SYN给客户端，询问客户端是否准备好进行数据通讯。

 （3） 客户必须再次回应服务段一个ACK报文，这是报文段3。

2、连接终止协议（四次握手）  
　 　由于TCP连接是全双工的，因此每个方向都必须单独进行关闭。这原则是当一方完成它的数据发送任务后就能发送一个FIN来终止这个方向的连接。收到一个 FIN只意味着这一方向上没有数据流动，一个TCP连接在收到一个FIN后仍能发送数据。首先进行关闭的一方将执行主动关闭，而另一方执行被动关闭。

　（1） TCP客户端发送一个FIN，用来关闭客户到服务器的数据传送（报文段4）。  
　（2） 服务器收到这个FIN，它发回一个ACK，确认序号为收到的序号加1（报文段5）。和SYN一样，一个FIN将占用一个序号。  
　（3） 服务器关闭客户端的连接，发送一个FIN给客户端（报文段6）。  
　（4） 客户段发回ACK报文确认，并将确认序号设置为收到序号加1（报文段7）。

CLOSED: 这个没什么好说的了，表示初始状态。

LISTEN: 这个也是非常容易理解的一个状态，表示服务器端的某个SOCKET处于监听状态，可以接受连接了。

SYN\_RCVD: 这个状态表示接受到了SYN报文，在正常情况下，这个状态是服务器端的SOCKET在建立TCP连接时的三次握手会话过程中的一个中间状态，很短暂，基本上用netstat你是很难看到这种状态的，除非你特意写了一个客户端测试程序，故意将三次TCP握手过程中最后一个ACK报文不予发送。因此这种状态时，当收到客户端的ACK报文后，它会进入到ESTABLISHED状态。

SYN\_SENT: 这个状态与SYN\_RCVD遥想呼应，当客户端SOCKET执行CONNECT连接时，它首先发送SYN报文，因此也随即它会进入到了SYN\_SENT状态，并等待服务端的发送三次握手中的第2个报文。SYN\_SENT状态表示客户端已发送SYN报文。

ESTABLISHED：这个容易理解了，表示连接已经建立了。

FIN\_WAIT\_1: 这个状态要好好解释一下，其实FIN\_WAIT\_1和FIN\_WAIT\_2状态的真正含义都是表示等待对方的FIN报文。而这两种状态的区别是：FIN\_WAIT\_1状态实际上是当SOCKET在ESTABLISHED状态时，它想主动关闭连接，向对方发送了FIN报文，此时该SOCKET即进入到FIN\_WAIT\_1状态。而当对方回应ACK报文后，则进入到FIN\_WAIT\_2状态，当然在实际的正常情况下，无论对方何种情况下，都应该马上回应ACK报文，所以FIN\_WAIT\_1状态一般是比较难见到的，而FIN\_WAIT\_2状态还有时常常可以用netstat看到。

FIN\_WAIT\_2：上面已经详细解释了这种状态，实际上FIN\_WAIT\_2状态下的SOCKET，表示半连接，也即有一方要求close连接，但另外还告诉对方，我暂时还有点数据需要传送给你，稍后再关闭连接。

TIME\_WAIT: 表示收到了对方的FIN报文，并发送出了ACK报文，就等2MSL后即可回到CLOSED可用状态了。如果FIN\_WAIT\_1状态下，收到了对方同时带FIN标志和ACK标志的报文时，可以直接进入到TIME\_WAIT状态，而无须经过FIN\_WAIT\_2状态。

CLOSING: 这种状态比较特殊，实际情况中应该是很少见，属于一种比较罕见的例外状态。正常情况下，当你发送FIN报文后，按理来说是应该先收到（或同时收到）对方的ACK报文，再收到对方的FIN报文。但是CLOSING状态表示你发送FIN报文后，并没有收到对方的ACK报文，反而却也收到了对方的FIN报文。什么情况下会出现此种情况呢？其实细想一下，也不难得出结论：那就是如果双方几乎在同时close一个SOCKET的话，那么就出现了双方同时发送FIN报文的情况，也即会出现CLOSING状态，表示双方都正在关闭SOCKET连接。

CLOSE\_WAIT: 这种状态的含义其实是表示在等待关闭。怎么理解呢？当对方close一个SOCKET后发送FIN报文给自己，你系统毫无疑问地会回应一个ACK报文给对方，此时则进入到CLOSE\_WAIT状态。接下来呢，实际上你真正需要考虑的事情是察看你是否还有数据发送给对方，如果没有的话，那么你也就可以close这个SOCKET，发送FIN报文给对方，也即关闭连接。所以你在CLOSE\_WAIT状态下，需要完成的事情是等待你去关闭连接。

LAST\_ACK: 这个状态还是比较容易好理解的，它是被动关闭一方在发送FIN报文后，最后等待对方的ACK报文。当收到ACK报文后，也即可以进入到CLOSED可用状态了。

最后有2个问题的回答，我自己分析后的结论（不一定保证100%正确）

1、 为什么建立连接协议是三次握手，而关闭连接却是四次握手呢？

这是因为服务端的LISTEN状态下的SOCKET当收到SYN报文的建连请求后，它可以把ACK和SYN（ACK起应答作用，而SYN起同步作用）放在一个报文里来发送。但关闭连接时，当收到对方的FIN报文通知时，它仅仅表示对方没有数据发送给你了；但未必你所有的数据都全部发送给对方了，所以你可以未必会马上会关闭SOCKET,也即你可能还需要发送一些数据给对方之后，再发送FIN报文给对方来表示你同意现在可以关闭连接了，所以它这里的ACK报文和FIN报文多数情况下都是分开发送的。

2、 为什么TIME\_WAIT状态还需要等2MSL后才能返回到CLOSED状态？

这是因为：虽然双方都同意关闭连接了，而且握手的4个报文也都协调和发送完毕，按理可以直接回到CLOSED状态（就好比从SYN\_SEND状态到ESTABLISH状态那样）；但是因为我们必须要假想网络是不可靠的，你无法保证你最后发送的ACK报文会一定被对方收到，因此对方处于LAST\_ACK状态下的SOCKET可能会因为超时未收到ACK报文，而重发FIN报文，所以这个TIME\_WAIT状态的作用就是用来重发可能丢失的ACK报文。