<https://www.cnblogs.com/zhangjing0502/archive/2012/06/26/2564001.html>

你需要了解的一些系统调用：  
socket()  
bind()  
connect()  
listen()  
accept()  
send()  
recv()  
sendto()  
recvfrom()  
close()  
shutdown()  
setsockopt()  
getsockopt()  
getpeername()  
getsockname()  
gethostbyname()  
gethostbyaddr()  
getprotobyname()  
fcntl()  
我们将在以下详细介绍这些系统调用。

1. **socket（）函数**

#include <sys/types.h>  
#include <sys/socket.h>  
**int socket（int domain , int type , int protocol）;**

首先，domain 需要被设置为 “AF\_INET”，就像上面的struct sockaddr\_in。然后，type参数告诉内核这个socket 是什么类型，“SOCK\_STREAM”或是“SOCK\_DGRAM”。最后，只需要把protocol 设置为0 。

socket()函数只是简单的返回一个你以后可以使用的套接字描述符。如果发生错误，socket()函数返回 –1 。全局变量errno 将被设置为错误代码。（可以参考perror() 的manpages）

2.**bind（int sockfd,const struct sockaddr \*myaddr,socklen\_t addrlen）**

bind()的系统调用声明如下：

#include <sys/types.h>  
#include <sys/socket.h>  
**int bind (int sockfd , struct sockaddr \*my\_addr , int addrlen) ;**  
参数说明：  
l sockfd 是由socket()函数返回的套接字描述符。  
l my\_addr 是一个指向struct sockaddr 的指针，包含有关你的地址的信息：名称、端口和IP 地址。  
l addrlen 可以设置为sizeof(struct sockaddr)。

当bind()函数调用错误的时候，它也是返回–1 作为错误发生的标志。errn 的值为错误代码。

当你调用bind()的时候，不要把端口数设置的过小！小于1024 的所有端口都是保留下来作为系统使用端口的，没有root 权利无法使用。你可以使用1024 以上的任何端口，一直到65535

                       对socket进行定位

|  |
| --- |
|  |
| 相关函数 | socket，accept，connect，listen |
| 表头文件 | #include<sys/types.h> #include<sys/socket.h> |
| 定义函数 | int bind(int sockfd,struct sockaddr \* my\_addr,int addrlen); |
| 函数说明 | bind()用来设置给参数sockfd的socket一个名称。此名称由参数my\_addr指向一sockaddr结构，对于不同的socket domain定义了一个通用的数据结构 struct sockaddr { unsigned short int sa\_family; char sa\_data[14]; }; sa\_family 为调用socket（）时的domain参数，即AF\_xxxx值。 sa\_data 最多使用14个字符长度。 此sockaddr结构会因使用不同的socket domain而有不同结构定义，例如使用AF\_INET domain，其socketaddr结构定义便为 struct socketaddr\_in { unsigned short int sin\_family; uint16\_t sin\_port; struct in\_addr sin\_addr; unsigned char sin\_zero[8]; }; struct in\_addr { uint32\_t s\_addr; }; sin\_family 即为sa\_family sin\_port 为使用的port编号 sin\_addr.s\_addr 为IP 地址 sin\_zero 未使用。 |
| 参数 | addrlen为sockaddr的结构长度。 |
| 返回值 | 成功则返回0，失败返回-1，错误原因存于errno中。 |
| 错误代码 | EBADF 参数sockfd 非合法socket处理代码。 EACCESS 权限不足 ENOTSOCK 参数sockfd为一文件描述词，非socket。 |

该函数用来指定一个端口号，一个IP地址，两者都指定，或者两者都不指定.可以不使用该函数调用。使用socket（）得到套接口后可以直接调用函数conect（）或者listen（），这时内核会自动给套接口分配一个地址和端口号（众所周知的端口号），这是常用的方法。只有在进程需要使用特定的网络地址和端口时才会进行绑定，即使用bind（）函数。调用bind（）的常见错误是EADDRINUSE，即指定的地址正在使用，主要是指定的端口号被使用了，IP地址可以被多个进程使用，但端口在同一时刻只能被一个进程使用。

套接口中port=0表示由内核指定端口号，设定sin\_addr为INADDR\_ANY（表示任意的意思），就有内核指定端口号。

设置端口为0的语句：

       struct socketaddr\_in seeveraddr;

       serveraddr.port = 0;

设置IP的语句：

      serveraddr.sin\_addr = htonl(INADDR\_ANY);

htonl（）说明：

|  |
| --- |
|  |
|  |  |
| 表头文件 | #include<netinet/in.h> |
| 定义函数 | unsigned long int htonl(unsigned long int hostlong); |
| 函数说明 | htonl（）用来将参数指定的32位hostlong 转换成网络字符顺序。 |
| 返回值 | 返回对应的网络字符顺序。 |

3. **connect() 函数**

#include <sys/types.h>  
#include <sys/socket.h>  
**int connect (int sockfd, struct sockaddr \*serv\_addr, int addrlen);**  
connect()的三个参数意义如下：  
l sockfd ：套接字文件描述符，由socket()函数返回的。  
l serv\_addr 是一个存储远程计算机的IP 地址和端口信息的结构。  
l addrlen 应该是sizeof(struct sockaddr)。

4. **listen()函数**

#include <sys/socket.h>  
**int listen(int sockfd, int backlog);**  
listen()函数的参数意义如下：  
l sockfd 是一个套接字描述符，由socket()系统调用获得。  
l backlog 是未经过处理的连接请求队列可以容纳的最大数目。  
backlog 具体一些是什么意思呢？每一个连入请求都要进入一个连入请求队列，等待listen 的程序调用accept()（accept()函数下面有介绍）函数来接受这个连接。当系统还没有调用accept()函数的时候，如果有很多连接，那么本地能够等待的最大数目就是backlog 的数值。你可以将其设成5 到10 之间的数值（推荐）。像上面的所有函数一样， listen()如果返回 –1 ，那么说明在listen()的执行过程中发生了错误。全局变量errno 中存储了错误代码。

5. **accept（）函数**

当调用它的时候，**大致过程是下面这样的**：  
**l 有人从很远很远的地方尝试调用connect()来连接你的机器上的某个端口（当然是你已经在listen()的）。**  
**l 他的连接将被listen 加入等待队列等待accept()函数的调用（加入等待队列的最多数目由调用listen()函数的第二个参数backlog 来决定）。**  
**l 你调用accept()函数，告诉他你准备连接。**  
**l accept()函数将回返回一个新的套接字描述符，这个描述符就代表了这个连接！**

#include <sys/socket.h>  
**int accept(int sockfd, void \*addr, int \*addrlen);**  
accept()函数的参数意义如下：  
l sockfd 是正在listen() 的一个套接字描述符。  
l **addr 一般是一个指向struct sockaddr\_in 结构的指针；里面存储着远程连接过来的计算机的信息（比如远程计算机的IP 地址和端口）。**  
l addrlen 是一个本地的整型数值，在它的地址传给accept() 前它的值应该是sizeof(struct sockaddr\_in)；accept()不会在addr 中存储多余addrlen bytes 大小的数据。如果  
accept()函数在addr 中存储的数据量不足addrlen，则accept()函数会改变addrlen 的值来反应这个情况。

6.**send()、recv()函数**

#include <sys/types.h>  
#include <sys/socket.h>  
**int send(int sockfd, const void \*msg, int len, int flags);**  
send 的参数含义如下：  
l sockfd 是代表你与远程程序连接的套接字描述符。  
l msg 是一个指针，指向你想发送的信息的地址。  
l len 是你想发送信息的长度。  
l flags 发送标记。一般都设为0（你可以查看send 的man pages 来获得其他的参数  
值并且明白各个参数所代表的含义）。

**send()函数在调用后会返回它真正发送数据的长度**。  
注意：send() 所发送的数据可能少于你给它的参数所指定的长度！因为如果你给send()的参数中包含的数据的长度远远大于send()所能一次发送的数据，则send()函数  
只发送它所能发送的最大数据长度，然后它相信你会把剩下的数据再次调用它来进行第二次发送。所以，记住如果send()函数的返回值小于len 的话，则你需要再次发送剩下的数据。幸运的是，如果包足够小（小于1K），那么send()一般都会一次发送光的。像上面的函数一样，send()函数如果发生错误，则返回 –1 ，错误代码存储在全局变量errno 中。

函数recv()调用在许多方面都和send()很相似，下面是recv()函数的声明：  
#include <sys/types.h>  
#include <sys/socket.h>

**int recv(int sockfd, void \*buf, int len, unsigned int flags）;**  
recv()的参数含义如下：  
l sockfd 是你要读取数据的套接字描述符。  
l buf 是一个指针，指向你能存储数据的内存缓存区域。  
l len 是缓存区的最大尺寸。  
l flags 是recv() 函数的一个标志，一般都为0 （具体的其他数值和含义请参考recv()  
的man pages）。  
recv() 返回它所真正收到的数据的长度。（也就是存到buf 中数据的长度）。如果返回–1 则代表发生了错误（比如网络以外中断、对方关闭了套接字连接等），全局变量errno 里面存储了错误代码。

7. **close()和shutdown()函数**

程序进行网络传输完毕后，你需要关闭这个套接字描述符所表示的连接。实现这个非常简单，只需要使用标准的关闭文件的函数：close()。  
使用方法：  
close(sockfd);  
执行close()之后，套接字将不会在允许进行读操作和写操作。任何有关对套接字描述符进行读和写的操作都会接收到一个错误。  
如果你想对网络套接字的关闭进行进一步的操作的话，你可以使用函数shutdown()。  
它允许你进行单向的关闭操作，或是全部禁止掉。  
shutdown()的声明为：  
#include <sys/socket.h>  
int shutdown（int sockfd, int how）;  
它的参数含义如下：  
l sockfd 是一个你所想关闭的套接字描述符．  
l how 可以取下面的值。0 表示不允许以后数据的接收操；1 表示不允许以后数据的发送操作；2 表示和close()一样，不允许以后的任何操作（包括接收，发送数据）  
shutdown() 如果执行成功将返回0，如果在调用过程中发生了错误，它将返回–1，全局变量errno 中存储了错误代码．  
如果你在一个未连接的数据报套接字上使用shutdown() 函数（还记得可以对数据报套接字UDP 进行connect()操作吗？），它将什么也不做．

8. **setsockopt() 和getsockopt() 函数**

Linux 所提供的socket 库含有一个错误（bug）。此错误表现为你不能为一个套接字重  
新启用同一个端口号，即使在你正常关闭该套接字以后。问题就是Linux 内核在一个绑定套接字的进程结束后从不把端口标记为未用。

在Linux 中绕开这个问题的办法是，当套接字已经打开但尚未有连接的时候用  
setsockopt()系统调用在其上设定选项（options）。setsockopt() 调用设置选项而getsockopt()  
从给定的套接字取得选项。  
这里是这些调用的语法：  
#include<sys/types.h>  
#include<sys/socket.h>  
**int getsockopt(int sockfd, int level, int name, char \*value, int \*optlen);**  
**int setsockopt(int sockfd, int level, int name, char \*value, int \*optlen);**  
下面是两个调用的参数说明：  
l sockfd 必须是一个已打开的套接字。  
l level 是函数所使用的协议标准（protocol level）（TCP/IP 协议使用IPPROTO\_TCP，  
套接字标准的选项实用SOL\_SOCKET）。  
l name 选项在套接字说明书中（man page）有详细说明。  
l value 指向为getsockopt()函数所获取的值，setsockopt()函数所设置的值的地址。  
l optlen 指针指向一个整数，该整数包含参数以字节计算的长度。

9. getpeername()函数

这个函数可以取得一个已经连接上的套接字的远程信息（比如IP 地址和端口），告诉你在远程和你连接的究竟是谁．  
它的声明为：  
#include <sys/socket.h>  
int getpeername(int sockfd, struct sockaddr \*addr, int \*addrlen);  
下面是参数说明：  
l sockfd 是你想取得远程信息的那个套接字描述符。  
l addr 是一个指向struct sockaddr （或是struct sockaddr\_in）的指针。  
l addrlen 是一个指向int 的指针，应该赋于sizeof(struct sockaddr)的大小。  
如果在函数执行过程中出现了错误，函数将返回 –1 ，并且错误代码储存在全局变量  
errno 中。  
当你拥有了远程连接用户的IP 地址，你就可以使用inet\_ntoa() 或gethostbyaddr()来输  
出信息或是做进一步的处理。

10. gethostname()函数

gethostname()函数可以取得本地主机的信息．它比getpeername()要容易使用一些。  
它返回正在执行它的计算机的名字。返回的这个名字可以被gethostbyname()函数使用，  
由此可以得到本地主机的IP 地址。  
下面是它的声明：  
#include <unistd.h>  
int gethostname(char \*hostname, size\_t size);  
参数说明如下：  
l hostname 是一个指向字符数组的指针，当函数返回的时候，它里面的数据就是本  
地的主机的名字．  
l size 是hostname 指向的数组的长度．  
函数如果成功执行，它返回0，如果出现错误，则返回–1，全局变量errno 中存储着错  
误代码。

11. gethostbyname()函数

#include <netdb.h>  
struct hostent \*gethostbyname(const char \*name);  
正如你所看见的，它返回了一个指向struct hostent 的指针．Struct hostent 是这样定义  
的：  
struct hostent {  
char \*h\_name;  
char \*\*h\_aliases;  
int h\_addrtype;  
int h\_length;  
char \*\*h\_addr\_list;  
};  
#define h\_addr h\_addr\_list[0]  
下面是上面各个域代表含义的解释：  
l h\_name 是这个主机的正式名称。  
l h\_aliases 是一个以NULL（空字符）结尾的数组，里面存储了主机的备用名称。  
l h\_addrtype 是返回地址的类型，一般来说是“AF\_INET”。  
l h\_length 是地址的字节长度。  
l h\_addr\_list 是一个以0 结尾的数组，存储了主机的网络地址。  
注意：网络地址是以网络字节顺序存储的。  
l h\_addr - h\_addr\_list 数组的第一个成员．  
gethostbyname() 返回的指针指向结构struct hostent ，如果发生错误，它将会返回NULL  
（但是errno 并不代表错误代码，h\_errno 中存储的才识错误代码。参考下面的herror()函数）。

使用gethostbyname()函数，你不能使用perror()来输出错误信息（因为错误代码存储在  
h\_errno 中而不是errno 中。所以，你需要调用herror()函数。

######################################################################################################################################################################

2. 服务器进程中系统调用的顺序

**socket（）————bind（）————listen（）————accept（）**

**在面向连接的协议的程序中,服务器执行以下函数：**  
**l 调用socket()函数创建一个套接字。**  
**l 调用bind()函数把自己绑定在一个地址上。**  
**l 调用listen()函数侦听连接。**  
**l 调用accept()函数接受所有引入的请求。**  
**l 调用recv()函数获取引入的信息然后调用send()回答**。

**TCP三次握手协议：**

**（1）客户端先用connect（）向服务器发出一个要求连接的信号SYN1。**

**（2）服务器进程接收到这个信号后，发回应答信号ack1，同时这也是一个要求回答的信号SYN2。**

**（3）客户端收到应答信号ack1和SYN2后，再次应答ack2。**

**（4）服务器收到应答信号ack2，一次连接才算建立完成。**

3.使用完一个套接口后，一定要记得将它关掉，使用函数close（int sockfd）

4.Linux系统调用-- getsockname函数详解

    当不用bind（）或调用bind（）没有指定本地协议地址时，可以调用getsockname（）来返回内核分配给此连接的本地IP地址和端口号，还可以获得某套接口的协议族。当一个新的连接建立时，服务器也可以调用getsockname（）来获得分配给此连接的本地IP地址。

 当一个服务器的子进程调用exec函数启动执行时，只能调用getpeername()函数来获得客户的Ip地址和端口号。

【 getsockname系统调用】     
      
功能描述：   
返回指定套接字的名称。   
  
用法：   
#include <sys/socket.h>

int getsockname(int sockfd, struct sockaddr \*name, socklen\_t \*namelen);

参数：     
sockfd：需要获取名称的套接字。  
name：存放所获取套接字名称的缓冲区。  
nemalen：作为入口参数，name指向空间的最大长度。作为出口参数，name的实际长度。  
  
 返回说明：     
成功执行时，返回0。失败返回-1，errno被设为以下的某个值     
EBADF：sock不是有效的文件描述词  
EFAULT：name指向的内存并非有效的进程空间  
EINVAL：namelen无效，可能为负值  
ENOBUFS：执行操作时，系统资源不足  
ENOTCONN：套接字尚未连接上  
ENOTSOCK：sock描述的不是套接字

功能：  
getsockname: 返回本地协议地址  
getpeername：返回远程协议地址  
定义：  
#include <sys/unistd.h>  
  
int getsockname (int sockfd, struct sockaddr \*localaddr, int \*addrlen);  
int getpeername(int sockfd, struct sockaddr \*peeraddr, int \*addrlen);

#### getsockname()[函数](http://baike.baidu.com/view/15061.htm)用于获取一个套接口的名字。它用于一个已捆绑或已连接套接口s，本地地址将被返回。本调用特别适用于如下情况：未调用bind()就调用了connect()，这时唯有getsockname()调用可以获知系统内定的本地地址。在返回时，namelen参数包含了名字的实际字节数。    若一个套接口与INADDR\_ANY捆绑，也就是说该套接口可以用任意主机的地址，此时除非调用connect()或accept()来连接，否则getsockname()将不会返回主机IP地址的任何信息。除非套接口被连接，WINDOWS套接口应用程序不应假设IP地址会从INADDR\_ANY变成其他地址。这是因为对于多个主机环境下，除非套接口被连接，否则该套接口所用的IP地址是不可知的。

**Open C 套接字： getsockname 方法**

**getsockname** - 获取套接字名称

int getsockname (int s, struct sockaddr \* restrict name, socklen\_t \* restrict namelen);

**getsockname**系统调用返回指定套接字的当前**名称**。**namelen**应被初始化指出**name**所指向的空间容量。返回时，该参数含有返回名称

的实际大小（按字节）。

下面是**getsockname**函数的用法：

#include <sys/socket.h>  
#include <netinet/in.h>  
#include <unistd.h>  
TInt GetSockName()  
{   
int sock\_fd;   
struct sockaddr\_in addr,ss;   
unsigned int len;   
sock\_fd = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);   
addr.sin\_family = AF\_INET;   
addr.sin\_addr.s\_addr = htonl(INADDR\_ANY);   
addr.sin\_port = htons(5000);   
bind(sock\_fd,(struct sockaddr\*)&addr,sizeof(addr));   
len=sizeof(ss);   
getsockname(sock\_fd,(struct sockaddr\*)&ss,&len);   
close(sock\_fd);  
}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

LINUX函数查询：http://www.opengroup.org/search/

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

LINUX下Socket编程笔记：http://blog.chinaunix.net/u/19185/article\_56798.html

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

程序实例：

###################################################################################

//使用方法：定义的服务器的远端端口号是4000，故编译运行该程序后，需要使用以下命令在终端上显示：Hello World！  
//在cmd命令行中输入：telnet 远端服务器地址 端口号  
//我的实际运行 telnet 192.168.12.94 4000?

#include <stdio.h>  
#include <stdlib.h>  
#include <errno.h>  
#include <string.h>  
#include <sys/types.h>  
#include <netinet/in.h>  
#include <sys/socket.h>  
#include <sys/wait.h>  
// 服务器要监听的本地端口  
#define MYPORT 4000  
// 能够同时接受多少没有accept 的连接  
#define BACKLOG 10  
main()  
{  
    // 在sock\_fd 上进行监听，new\_fd 接受新的连接  
    int sockfd, new\_fd ;  
    // 自己的地址信息  
    struct sockaddr\_in my\_addr;  
    // 连接者的地址信息  
    struct sockaddr\_in their\_addr;  
    int sin\_size;  
    // 这里就是我们一直强调的错误检查．如果调用socket() 出错，则返回  
    if ((sockfd = socket(AF\_INET, SOCK\_STREAM, 0)) == -1)  
    {  
        // 输出错误提示并退出  
        perror("socket");  
        exit(1);  
    }  
    // 主机字节顺序  
    my\_addr.sin\_family = AF\_INET;  
    // 网络字节顺序，短整型  
    my\_addr.sin\_port = htons(MYPORT);  
    // 将运行程序机器的IP 填充入s\_addr  
    my\_addr.sin\_addr.s\_addr = INADDR\_ANY;  
    // 将此结构的其余空间清零  
    bzero(&(my\_addr.sin\_zero), 8);  
    // 这里是我们一直强调的错误检查！！  
    if (bind(sockfd, (struct sockaddr \*)&my\_addr,sizeof(struct sockaddr)) == -1)  
    {  
        // 如果调用bind()失败，则给出错误提示，退出  
        perror("bind");  
        exit(1);  
    }  
    // 这里是我们一直强调的错误检查！！  
    if (listen(sockfd, BACKLOG) == -1)  
    {  
        // 如果调用listen 失败，则给出错误提示，退出  
        perror("listen");  
        exit(1);  
    }  
    while(1)  
    {  
        // 这里是主accept()循环  
        sin\_size = sizeof(struct sockaddr\_in);  
        // 这里是我们一直强调的错误检查！！  
        if ((new\_fd = accept(sockfd, (struct sockaddr \*)&their\_addr, &sin\_size)) == -1)  
        {  
            // 如果调用accept()出现错误，则给出错误提示，进入下一个循环  
            perror("accept");  
            continue;  
        }  
        // 服务器给出出现连接的信息  
        printf("server: got connection from %s\n", inet\_ntoa(their\_addr.sin\_addr));  
        // 这里将建立一个子进程来和刚刚建立的套接字进行通讯  
        if (!fork())  
        {  
            // 这里是子进程  
            // 这里就是我们说的错误检查！  
            if (send(new\_fd, "Hello, world!\n", 14, 0) == -1)  
            {  
                // 如果错误，则给出错误提示，然后关闭这个新连接，退出  
                perror("send");  
                close(new\_fd);  
                exit(0);  
            }  
            // 关闭new\_fd 代表的这个套接字连接  
            close(new\_fd);  
        }  
    }  
    // 等待所有的子进程都退出  
    while(waitpid(-1,NULL,WNOHANG) > 0);  
}

###################################################################################

###################################################################################

/\* include fig01 \*/  
#include <sys/types.h>  
#include <sys/socket.h>  
#include <netinet/in.h>  
#include <arpa/inet.h>  
#include <netdb.h>  
#include <stdio.h>  
#include <unistd.h>   
#include <string.h>   
#include <sys/poll.h>  
#include <errno.h>

#define MAXLINE 512  
#define NOTDEF  
int  
main(int argc, char \*\*argv)  
{  
 int      i, maxi, maxfd, listenfd, connfd, sockfd;  
 int      nready, client[FD\_SETSIZE];  
 ssize\_t     n;  
 fd\_set     rset, allset;  
 char     buf[MAXLINE];  
 socklen\_t    clilen;  
 struct sockaddr\_in cliaddr, servaddr;

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

bzero(&servaddr, sizeof(servaddr));  
 servaddr.sin\_family      = AF\_INET;  
 servaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);  
 servaddr.sin\_port        = htons(4563);

bind(listenfd, (struct sockaddr \*) &servaddr, sizeof(servaddr));

listen(listenfd, 12);

maxfd = listenfd;    /\* initialize \*/  
 maxi = -1;      /\* index into client[] array \*/  
 //for (i = 0; i < FD\_SETSIZE; i++)  
 for (i = 0; i < 3; i++)  
   client[i] = -1;    /\* -1 indicates available entry \*/  
 /\*  
 void FD\_ZERO(fd\_set \*fdset)  
 Initialises the file descriptor set fdset to have zero bits for all file descriptors.   
 初始化所有的文件描述符fd\_set为0  
 \*/  
 FD\_ZERO(&allset);  
 /\*  
 void FD\_SET(int fd, fd\_set \*fdset)  
 Sets the bit for the file descriptor fd in the file descriptor set fdset.   
 \*/  
 FD\_SET(listenfd, &allset);  
/\* end fig01 \*/

/\* include fig02 \*/  
 for ( ; ; ) {  
   rset = allset;   /\* structure assignment \*/  
   nready =select(maxfd+1, &rset, NULL, NULL, NULL);  
   /\*  
   定义函数 int select(int n,fd\_set \* readfds,fd\_set \* writefds,fd\_set \*

exceptfds,struct timeval \* timeout);

   select()用来等待文件描述词状态的改变。参数n代表最大的文件描述词加1，参数readfds、writefds 和exceptfds 称为描述词组，是用来回传该描述词的读，写或例外的状况。底下的宏提供了处理这三种描述词组的方式:  
   FD\_CLR(inr fd,fd\_set\* set)；用来清除描述词组set中相关fd 的位  
   FD\_ISSET(int fd,fd\_set \*set)；用来测试描述词组set中相关fd 的位是否为真  
   FD\_SET（int fd,fd\_set\*set）；用来设置描述词组set中相关fd的位  
   FD\_ZERO（fd\_set \*set）； 用来清除描述词组set的全部位  
     
   参数 timeout为结构timeval，用来设置select()的等待时间，其结构定义如下  
   struct timeval  
   {  
   time\_t tv\_sec;  
   time\_t tv\_usec;  
   };  
     
   返回值 如果参数timeout设为NULL则表示select（）没有timeout。  
     
   错误代码 执行成功则返回文件描述词状态已改变的个数，如果返回0代表在描述词状态改变前已超过timeout时间，当有错误发生时则返回-1，错误原因存于errno，此时参数readfds，writefds，exceptfds和timeout的值变成不可预测。  
   EBADF 文件描述词为无效的或该文件已关闭  
   EINTR 此调用被信号所中断  
   EINVAL 参数n 为负值。  
   ENOMEM 核心内存不足  
    
   常见的程序片段:fs\_set readset；  
   FD\_ZERO(&readset);  
   FD\_SET(fd,&readset);  
   select(fd+1,&readset,NULL,NULL,NULL);  
   if(FD\_ISSET(fd,readset){……}

   \*/

   if (FD\_ISSET(listenfd, &rset)) { /\* new client connection \*/  
    /\*  
    int FD\_ISSET(int fd, fd\_set \*fdset)  
    Returns a non-zero value if the bit for the file descriptor fd is set in the file descriptor set by fdset, and 0 otherwise.   
    \*/  
    clilen = sizeof(cliaddr);  
    connfd = accept(listenfd, (struct sockaddr \*) &cliaddr, &clilen);  
    printf("Welcome!\n");  
    sleep(1);

#ifndef NOTDEF  
    printf("new client: %s, port %d\n",inet\_ntop(AF\_INET, &cliaddr.sin\_addr, NULL,4 ),ntohs(cliaddr.sin\_port));  
#endif

    //for (i = 0; i < FD\_SETSIZE; i++)  
    for (i = 0; i < 3; i++)  
     if (client[i] < 0) {  
      client[i] = connfd; /\* save descriptor \*/  
      break;  
     }  
    if (i == FD\_SETSIZE)  
     printf("too many clients");

    FD\_SET(connfd, &allset); /\* add new descriptor to set \*/  
    if (connfd > maxfd)  
     maxfd = connfd;    /\* for select \*/  
    if (i > maxi)  
     maxi = i;     /\* max index in client[] array \*/

    if (--nready <= 0)  
     continue;     /\* no more readable descriptors \*/  
   }

   for (i = 0; i <= maxi; i++) { /\* check all clients for data \*/  
    if ( (sockfd = client[i]) < 0)  
     continue;  
    if (FD\_ISSET(sockfd, &rset)) {  
     if ( (n = read(sockfd, buf, MAXLINE)) == 0) {  
       /\*4connection closed by client \*/  
      close(sockfd);  
      FD\_CLR(sockfd, &allset);  
      /\*  
      void FD\_CLR(int fd, fd\_set \*fdset)  
          Clears the bit for the file descriptor fd in the file descriptor set fdset.   
      \*/  
      client[i] = -1;  
     } else  
      write(sockfd, buf, n);

     if (--nready <= 0)  
      break;     /\* no more readable descriptors \*/  
    }  
   }  
 }  
}  
/\* end fig02 \*/

###################################################################################