Unix Socket Programming

邓岩*

837123564@qq.com

2018 年 12 月 23 日

*www.dengyan.net

目录

1 前言 **5**

1.1 头文件 5

2 函数 **8**

2.1 字节序转换函数 8
2.2 字节操作函数 9
2.3 地址转换函数 10
2.4 socket 11
2.5 connect 12
2.6 bind 13
2.7 listen 14
2.8 accept 15
2.9 close 16
2.10 fork 17
2.11 exec 18
2.12 getsockname 19
2.13 getpeername 20
2.14 wait 21
2.15 waitpid 22
2.16 select 23
2.17 shutdown 26
2.18 pselect 27
2.19 poll 28
2.20 getsockopt 30
2.21 setsockopt 32
2.22 fcntl 33
2.23 readv 34
2.24 writev 35
2.25 recvfrom 36
2.26 sendto 37
2.27 recv 38
2.28 send 39
2.29 recvmsg 40
2.30 sendmsg 41
2.31 gethostbyname 42
2.32 gethostbyaddr 44

2.33 getservbyname . . . 45
2.34 getservbyport . . . 46
2.35 getaddrinfo . . . 47
2.35.1 host_serv . . . 47
2.35.2 tcp_connect . . . 48
2.35.3 tcp_connect . . . 49
2.36 gai_strerror . . . 50
2.37 freeaddrinfo . . . 51
2.38 getnameinfo . . . 52
2.39 . . . 53

3 例子 54
3.1 时间客户端及服务器端 . . . 54
3.1.1 客户端 . . . 54
3.1.2 服务器端 . . . 54
3.2 使用 select 实现的 echo 服务器 (TCP 及 UDP 同时支持) . . . 56
3.2.1 客户端 . . . 56
3.2.2 服务器端 . . . 57
3.3 使用 getaddrinfo 基于 tcp 实现的时间服务器 . . . 60
3.3.1 客户端 . . . 60
3.3.2 服务器端 . . . 60
3.4 观察 TCP 阻塞窗口 . . . 62
3.4.1 客户端 . . . 62
3.4.2 服务器端 . . . 62
3.5 XX . . . 64
3.5.1 客户端 . . . 64
3.5.2 服务器端 . . . 64

4 结构体 65
4.1 netinet/in.h . . . 65
4.1.1 in_addr . . . 65
4.1.2 sockaddr_in . . . 65
4.1.3 in6_addr . . . 65
4.1.4 sockaddr_in6 . . . 66
4.2 sys/socket.h . . . 66
4.2.1 sockaddr . . . 66
4.2.2 sockaddr_storage . . . 66
4.2.3 iovec . . . 67

4.2.4 msghdr 67
4.3 netdb.h 67
4.3.1 hostent 67
4.3.2 servent 67
4.3.3 addrinfo 68

1 前言

1.1 头文件

```
// /Users/dengyan/ClionProjects/Linux/linux.h
//
// Created by 邓岩 on 09/03/2017.
//

#ifndef LINUX_LINUX_H
#define LINUX_LINUX_H

# include <sys/types.h>
# include <ctype.h>
# include <stdio.h>
# include <stdlib.h>
# include <unistd.h>
# include <errno.h>
# include <string.h>
# include <mach-o/getsect.h>
# include <pwd.h>
# include <grp.h>
# include <utmp.h>
# include <sys/stat.h>
# include <fcntl.h>
# include <sys/uio.h>
# include <setjmp.h>
# include <sys/time.h>
# include <sys/times.h>
# include <sys/utsname.h>
# include <time.h>
# include <utime.h>
# include <dirent.h>
# include <libgen.h>
# include <signal.h>
# include <pthread.h>
# include <termios.h> //用于设置 tty
# include <utmpx.h>
```

```
35 # include <curses.h>
36 # include <sys/ioctl.h>
37 # include <sys/msg.h>
38 # include <sys/sem.h>
39 # include <sys/shm.h>
40 # include <sys/types.h>
41 # include <sys/msg.h>
42 # include <sys/mman.h>
43 # include <sys/socket.h> //套接字 API
44 # include <sys/un.h>
45 # include <arpa/inet.h> //大小端字节序转换
46 # include <netdb.h> //用于访问 dns
47 # include <sys/select.h>
48 # include <poll.h>
49 # include <aio.h> //posix 异步 io
50 # include <syslog.h>
51
52 #include <netinet/in.h>
53 #include <netinet/if_ether.h>
54 #include <net/if_arp.h>
55 #include <net/if.h>
56 #include <net/ethernet.h>
57
58 char * itoa(int s)
59 {
60     int i = 1;
61     int t = s;
62     while(t /= 10)
63         i++;
64     char * a = (char *)malloc(i+1);
65     a[i] = 0;
66     for (i--; i>=0; i--) {
67         t = s%10;
68         s /= 10;
69         a[i] = t+48;
70     }
71     return a;
```

```
72 }
73
74 void err_quit(char * str)
75 {
76     write(STDOUT_FILENO, str, strlen(str));
77     exit(-1);
78 }
79
80 void err_sys(char * str)
81 {
82     write(STDERR_FILENO, str, strlen(str));
83     exit(-1);
84 }
85 #endif //LINUX_LINUX_H
```

2 函数

2.1 字节序转换函数

```
/*
 * h 代表主机 (host)
 * n 代表网络 (network)
 * l 代表 32 位
 * s 代表 16 位
 * 表示在主机字节序与网络字节序之间进行转换
 */

uint32_t htonl(uint32_t hostlong);

uint16_t htons(uint16_t hostshort);

uint32_t ntohl(uint32_t netlong);

uint16_t ntohs(uint16_t netshort);
```

2.2 字节操作函数

```c
# include <strings.h>
//将地址 s 开始的 n 个字节全部置为 0
void bzero(void *s, size_t n);

//将 src 开始的 n 个字节拷贝到 dest 地址开始的 n 个字节
void bcopy(const void *src, void *dest, size_t n);

//比较 s1 和 s2 开始的 n 个字节
int bcmp(const void *s1, const void *s2, size_t n);

//以一个常量填充 s 开始的 n 个字节
void *memset(void *s, int c, size_t n);

//将 src 开始的 n 个字节拷贝到 dest 开始的 n 个字节
void *memcpy(void *dest, const void *src, size_t n);

//比较 s1 和 s2 开始的 n 个字节
int memcmp(const void *s1, const void *s2, size_t n);
```

2.3 地址转换函数

```c
# include <arpa/inet.h>
//点分十进制转换为网络序
int inet_aton(const char *cp, struct in_addr *inp);

//同上，但是已经废弃
in_addr_t inet_addr(const char *cp);

//将网络序转换为点分十进制，注意这里以结构体为参数
char *inet_ntoa(struct in_addr in);

/*
 * p 表示表达，n 表示数值，我个人习惯将 n 看为网络序
 * af 用于表示地址族，取决于 IPv4(AF_INET) 还是 IPv6(AF_INET6)
 */

/*
 * 错误返回-1，正确返回 1，输入格式错误返回 0
 * 一般第二个参数使用点分十进制字符串，第三个参数为地址结构体的指针
 * 点分十进制转换为网络序
 * 注意：此函数转换时会考虑本机的大小端特性
 */
int inet_pton(int af, const char *src, void *dst);

/*
 * 将网络序转换为点分十进制，size 用于表示 dst 缓冲区的大小
 * src 用于地址结构题的指针
 * 将网络序转换为点分十进制
 * 注意：此函数将参数视为网络字节序转换成字符串，所以对于小端法机器
 * 如果你想要提供自己的参数给它，可以先使用 htonl 再进行传递
 */
const char *inet_ntop(int af, const void *src,
                      char *dst, socklen_t size);
```

2.4 socket

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

返回一个套接字描述符

family:

AF_INET	IPv4 协议
AF_INET6	IPv6 协议
AF_UNIX	UNIX 域协议
AF_ROUTE	路由套接字
AF_KEY	密钥套接字

type:

SOCK_STREAM	字节流套接字
SOCK_DGRAM	数据报套接字
SOCK_SEQPACKET	有序分组套接字
SOCK_RAW	原始套接字

protocal:

IPPROTO_TCP	TCP 传输协议
IPPROTO_UDP	UDP 传输协议
IPPROTO_SCTP	SCTP 传输协议

2.5 connect

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

TCP 客户端请求与 TCP 服务器进行连接

如果要处理一个面向连接的网络服务 (SOCK_STREAM 或 SOCK_SEQPACKET)，那么在开始交换数据之前，需要在请求服务的进程套接字和提供服务的进程套接字之间建立一个连接，如果 sockfd 没有绑定到一个地址，connect 会给调用者绑定一个默认地址并分配一个未使用的随机端口，如果 connect 失败，在部分系统上套接字会变成未定义的，最好是关闭套接字，新建一个套接字后再进行 connect 操作，当在一个数据报 socket 上使用 connect 后，可以使用 read 和 write 操作描述符

客户端通常不把 IP 地址绑定到它的套接字上，当连接套接字时，内核将根据所用外出网络接口来选择源 IP 地址，外出接口取决于到达服务器所需的路径，所以当你访问本地连接时，通常选择的是回环地址，而访问外部网络时，则会使用其他的 IP 地址。当 TCP 服务器没有把 IP 地址绑定到它的套接字上时，内核就把客户发送的 SYN 的目的 IP 地址作为服务器的源 IP 地址

如果作用与一个使用 UDP 的套接字，那么可以通过使用 read 和 write 函数直接进行 UDP 数据交互，对于多网卡主机来说，对 UDP 调用 connect 时会使内核在此时根据目的主机地址来决定外出接口 (也就是决定后面发送数据时使用哪个本地 IP 地址)，而不是在使用 sendto 时确认

2.6 bind

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

把一个本地协议地址赋予一个套接字

addrlen 要根据使用的 sockaddr 来确定，不能使用 sizeof(struct sockaddr)，当 bind 用于 UNIX 域套接字时，sockaddr_ununsigned char sun_len; sa_family_t sun_family; char sun_path[104](用于创建套接字的文件名，该文件仅用于向客户客户进程告示套接字名字，无法打开，也不能由应用程序进行通讯) 而 sun_path 指定的文件已存在时，bind 会失败，也就是说该文件是一次性的，程序结束时就应该删除该文件，每次 bind 时都要保证该文件不存在

调用 bind 可以指定 IP 地址和端口，可以两者都指定，也可以都不指定，如果 IP 地址使用通配地址，那么将由内核决定绑定的 IP 地址，照理说使用通配地址的原因应该是一个主机可能含有多个网卡，使用通配地址可以保证无论数据发送到哪个网卡上都能接受到这个消息，如果端口号为 0，则由内核随机选择一个未绑定的端口号，其他情况均为进程指定

如果让内核来为套接字选择一个临时端口号，需要注意的是 bind 并不会返回所选择的值，可以看到第 bind 的第二个参数是一个常量指针，所以如果想获取绑定的端口号，需要使用 getsockname 来返回协议地址

以下常量都为主机序

INADDR_LOOPBACK(0x7f000001) 为 IPV4 回环地址

INADDR_ANY(0x0) 为 IPV4 通配地址，均为整形数据

IN6ADDR_LOOPBACK_INIT 为 IPV6 回环地址

IN6ADDR_ANY_INIT 为 IPV6 通配地址，为结构体类型

2.7 listen

```
int listen(int sockfd, int backlog);
```

将一个主动套接字转换为监听套接字并规定该套接字排队的最大连接个数

backlog 的值作为未完成连接队列以及已完成连接队列中连接之和的最大值

- **未完成连接队列：**队列中的每个连接处于 SYN_RCVD 状态，也就是接受到了客户端的 SYN 请求并且做出了回应，等待进行第三次握手，完成三次握手后连接从此队列转移到已完成连接队列，该队列中的每个连接的存活时间为一个 RTT
- **已完成连接队列：**即已完成三次握手的连接，状态为 ESTABLISHED，当调用 accept 函数时，队头的连接被取出，当连接处于队列中而已经有消息发送给此连接时，会放置到此套接字的接收缓冲区中

2.8 accept

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

由 TCP 服务器调用，用于从已完成连接队列对头返回下一个已完成连接，如果已完成连接队列为空，那么进程投入休眠 (阻塞模式)

获得 sockfd 监听的连接请求并建立连接，返回一个已连接套接字描述符，此描述符连接到客户端调用 connect 的进程，并将请求连接端的地址信息写入 addr 中，len 参数为缓冲区的大小，函数返回时，会将 len 改为向缓冲区写入的字节数，如果不关心对端机器的地址信息，可以将 addr 和 len 置为 NULL，如果 sockfd 是非阻塞且当前没有连接请求，accept 会退出并返回-1，否则将阻塞直到收到一个连接请求 (阻塞模式)

此函数为一个慢阻塞函数，当阻塞却捕获到一个设置了信号处理函数的信号时，此函数会退出 (如果装载信号处理函数时制定了 SA_RESTART 选项，部分系统可以自动重启此函数)，并且 errno 被设置为 EINTR

2.9 close

```
int close(int fd);
```

将该套接字描述符关闭，但不一定会使对应套接字释放，因为可能有还有其他的描述符指向此套接字

如希望关闭对应套接字，可以使用 shutdown 函数

2.10 fork

```
pid_t fork(void);
```

创建一个子进程，含有两个返回值，父进程返回子进程 pid，子进程返回 0

当执行 fork 之后，子进程会复制进程文件描述符表，但不会复制全局文件打开表，因为该表为内核级，当子进程或者父进程其一使用 close 关闭了该描述符后，另一个仍然可以进行 IO 操作，子进程一定要以 exit 退出，特别是在 socket 并发服务器里，很重要

2.11 exec

```
int execl(const char *path, const char *arg, ... (char *) NULL);
int execlp(const char *file, const char *arg, ... (char *) NULL);
int execle(const char *path, const char *arg, ... (char *) NULL, char *
↪  const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],
```

倒数第二位为'v' 代表参数类型为数组，为'l' 则为列表，第一个参数一般设置为命令的文件名, 最后一位为'p' 则会通过 PATH 环境变了查找文件，但是一旦 filename 中出现了一个 (/)，那么就不再使用 PATH 环境变了，最后一位为'e' 允许带环境参数，带环境参数后不会继承原进程环境变量, 如果不带环境参数则继承原进程环境变量，在我自己写的代码中，貌似只有 execlp 可以调用自己写的脚本，而且 file 必须为完整路径才能调用成功，可能是参数使用错误

2.12 getsockname

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

返回与套接字关联的本地协议地址

在一个没有调用 bind 的客户端上，connect 返回成功后，getsock 那么用于返回由内核赋予连接的 IP 地址和端口号

获取套接字 socket 所绑定的地址信息并写到 addr 中，len 表示缓冲区的大小，函数返回后 len 的值会变为向 addr 写入的字节数

在以端口号 0 调用 bind 后，getsockname 那么用于返回由内核赋予的本地端口号

可用于获取某个套接字的地址族

在以通配 IP 地址调用 bind 的 TCP 服务器上，与某个客户的连接一旦建立 (accept 返回)，此函数就可用于获取由内核赋予该连接的本地 IP 地址

2.13 getpeername

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

返回与套接字关联的外地协议地址

获取套接字 socket 对端主机的地址信息并写到 addr 中，len 表示缓冲区的大小，函数返回后 len 的值会变为向 addr 写入的字节数

当一个服务器是由调用过 accept 的某个进程通过调用 exec 执行程序时，它能够获取用户身份的唯一方式就是 getpeername

2.14 wait

```
pid_t wait(int *wstatus);
```

回收一个僵死子进程

调用时如果此时没有僵死进程，则会阻塞，如果有没回收的僵死进程，则返回此子进程的 pid

2.15 waitpid

```
pid_t waitpid(pid_t pid, int *status, int options);
```

回收一个僵死子进程

当 pid 为-1 时表示等待任意一个终止的子进程当 pid<-1 时表示等待进程组 id 等于 pid 绝对值的进程当 pid>-1 时表示等待进程 id 等于 pid 的进程

options:

WUNTRACED	当子进程由运行状态转变为停止状态时返回
WCONTINUED	当子进程由暂停状态转变为运行状态时返回
WNOHANG	以非阻塞模式运行

WCONTINUED 在 mac 上无效，由停止状态转变为运行态并不会使该系统调用返回，以下宏用于处理 status 参数，获取子进程的终止信息

```
WIFCONTINUED(status); //需要 options 设置 WCONTINUED，返回使子进程恢复的信
↪ 号量，mac 上无效
WIFSTOPPED(status); //只要当 options 设置 WUNTRACED 才有用，返回引起停止的
↪ 信号值
WIFSIGNALED(status); //返回引起杀死的信号值
WIFEXITED(status); //如果子进程正常退出，返回退出值
WEXITSTATUS(status);
```

2.16 select

```
int select(int nfds, fd_set *readset, fd_set *writeset, fd_set *exceptset,
↪  struct timeval *timeout);
```

当 readset 中含有描述符可以进行非阻塞读取操作时，当 writeset 中的描述符可以进行非阻塞写操作时，当 exceptset 中有异常条件待处理时，select 函数返回

IO 多路复用，等待一定长的时间，因超时返回时返回 0，因中断返回时返回-1，其他情况返回已准备好的文件描述符个数，nfds 为当前最大的文件描述符加 1，这样就只会检查小于 nfds 的文件描述符状态，readset，writeset，errorset 分别返回所关心描述符状态的结果，每一个位对应一个描述符，当调用完成后，若对应位为 1，则表示该下标对应的描述符为准备好状态。如某集合设置为 NULL，则表示对该状态不关心

timeout

1. timout 等于 NULL 时，永远等待，直到指定中的一个文件描述符已准备好或者捕捉到一个信号终端此进程
2. timeout->tv_sec==0 && timeout->tv_usec==0，不等待，测试所有文件描述符后立即返回
3. 当 timeout 有其他值时，等待指定的描述和微妙数，当指定描述符中的一个文件描述符准备好时，或者超过指定时间，则立即返回

select 函数的前两种情况有可能因调用信号处理函数而中断，此时会设置 EINTR 错误标志，部分系统可以通过在装载信号处理函数时指定 SA_RESTART 由内核重启此函数

描述符集 select 使用的**描述符集**，通常是一个整数数组，数组中的每个整数的每一位对应一个描述符，比如整数位 32 位，那么数组中的第一个元素对应于描述符 0-31，第二个元素对应于 32-63，所有的实现细节都与应用程序无关，通常使用以下四个宏对描述符集进行操作

```
void FD_ZERO(fd_set * fdset); //将一个 fdset 的所有位置为 0
void FD_SET(int fd,fd_set * fdset); //将 fd 加入 fdset
void FD_CLR(int fd,fd_set * fdset); //将 fd 从 fdset 中移除
int FD_ISSET(int fd,fd_set * fdset); //若 fd 在 fdset 中，返回非 0，否则返
↪ 回 0，可用于当 select 返回后判断 fd 的状态
```

由于 select 返回时会将各个集合中不满足要求的位统统置为 0，所以每次调用前需要重新将感兴趣的位均置为 1

描述符就绪条件

读： 当满足下列条件之一时，一个套接字准备好读

1. 该套接字接收缓冲区中的数据字节数大于等于套接字接收缓冲区底水位标记的当前大小。对这样的套接字执行读操作不会阻塞并将返回一个大于 0 的值 (也就是返回读入的数据长度)。可以使用 SO_RCVLOWAT 套接字选项设置该套接字的底水位标记。对于 TCP 和 UDP 套接字而言，该默认值为 1
2. 该连接的读半部关闭 (也就是接受了 FIN 的 TCP)，对于这样的套接字读操作将不阻塞并返回 0
3. 该套接字是监听套接字且已完成的连接数不为 0(也就是已连接队列不为空)，对这样的套接字的 accept 通常不会阻塞
4. 其上有一个套接字错误待处理，对这样的套接字的读操作将不阻塞并返回-1，同时把 errno 设置为确切的错误条件，这些待处理错误也可以通过指定 SO_ERROR 套接字选项调用 getsockopt 获取并清除

写： 当满足下列条件之一时，一个套接字准备好写

1. 对于已连接的套接字 (TCP) 来说该套接字发送缓冲区中的可用空间字节数大于等于套接字发送缓冲区底水位标记的当前大小，或者该套接字不需要连接 (UDP)，这代表此时进行写操作不会阻塞并返回一个大于 0 的值。可以使用 SO_SNDLOWAT 套接字选项来设置该套接字的底水位标记，对于 TCP 和 UDP 套接字而言，其默认值通常为 2048
2. 该连接的写半部关闭 (接收了 RST 的 TCP，或者使用 SHUT_WR 调用了 shutdown)，对这样的套接字的写操作将产生 SIGPIPE 信号
3. 使用非阻塞式 connect 的套接字已建立连接 (应该是对 UDP 套接字进行 connect 操作)，或者 connect 以及已失败告终
4. 其上有一个套接字错误待处理，对这样的套接字的写操作将不阻塞并返回-1，同时把 errno 设置为确切的错误条件，这些待处理错误也可以通过指定 SO_ERROR 套接字选项调用 getsockopt 获取并清除

异常： 当满足下列条件之一时，一个套接字含有异常条件待处理

1. 某个套接字的带外数据到达
2. 某个已置为分组模式的伪终端存在可从其主终端读取的控制状态信息

2.17 shutdown

```
int shutdown(int sockfd, int how);
```

终止网络连接

how 的取值

- **SHUT_RD** 关闭连接的读这一半，套接字中不再有数据可接收，而且套接字接收缓冲区中的仙友数据都被丢弃，进程不能再对这样的套接字调用任何的读函数，对一个 TCP 套接字这样调用 shutdown 函数后，由该套接字接收的来自对端任何数据都被确认，然后丢弃
- **SHUT_WR** 关闭连接的写这一半，对于 TCP 套接字，这称为半关闭，当前留在套接字发送缓冲区中的数据将被发送掉，后跟 TCP 的正常终止序列 (FIN 包)，此函数可以解决 close 关闭套接字时因套接字引用大于 1 产生的无法关闭问题，进程不能再对这样的套接字调用任何写函数

2.18 pselect

```
int pselect(int nfds, fd_set *readset, fd_set *writeset, fd_set
↪  *exceptset, const struct timespec *timeout, const sigset_t *sigmask);
```

行为类似于 select，但提供了 sigmask 参数用于当函数调用期间设定的信号屏蔽字，当返回时恢复屏蔽信号字

2.19 poll

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

等待文件描述符上面发生某个事件

```
struct pollfd {
    int   fd;         /* file descriptor */
    short events;     /* requested events */
    short revents;    /* returned events */
};
```

要测试的条件由 events 成员指定，函数在相应的 revents 成员中返回该描述符的状态，这两个成员中的每一个都指定某个特定条件的一位或多位构成，以下用于指定 events 标志以及测试 revents 标志的一些常值

常值	作为 events 的输入吗?	作为 revents 的结果吗?	说明
POLLIN	*	*	普通或优先级带数据可读
POLLRDNORM	*	*	普通数据可读
POLLRDBAND	*	*	优先级带数据可读
POLLPRI	*	*	高优先级数据可读
POLLOUT	*	*	普通数据可写
POLLWRNORM	*	*	普通数据可写
POLLWRBAND	*	*	优先级带数据可写
POLLERR		*	发生错误
POLLHUP		*	发生挂起
POLLNVAL		*	描述符不是一个打开的文件

POLLERR|POLLHUP|POLLNVAL 这三个值即使不设置在 events 中，也可能出现在 revents ，nfds 指定数组的最大下标，不是描述符个数，，因为如果不在关心某个特定描述符，则可以将数组中的 fd 成员设置成一个负值，此时 poll 将忽略这个描述符，timeout 为-1(INFTIM 也可以) 时永久等待，为 0 时测试后立即返回，其余值时为等待 timeout 毫秒

以下条件引起 poll 返回

- 所有正规 TCP 数据和所有 UDP 数据都被认为是普通数据
- TCP 的带外数据倍认为是优先级带数据
- 当 TCP 连接的读半部关闭时 (如收到一个 FIN)，也被认为是普通数据，随后的读操作将返回 0
- TCP 存在连接存在错误既可认为是普通数据，也可认为是错误 (POLLERR)，无论哪种情况随后的读操作都将返回-1，并把 errno 设置成合适的值，如收到 RST(ECONNRESET 错误) 或者发生超时等事件
- 在监听套接字上有新的连接可用即可认为是普通数据，也可认为是优先级数据，大多数情况视其为普通数据
- 非阻塞式 connect 的完成被认为是使相应套接字可写

2.20 getsockopt

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

获取套接字选项

其中 sockfd 必须指向一个打开的文件描述符，level 指定系统中解释选项的代码或为通用套接字代码，或为某个特定于协议的代码，optval 是一个指向某个变量的指针，用于将获取到的数据放入指向的容器中，optlen 指定容器的大小

下图汇总了可有 getsockopt 获取或由 setsockopt 设置的选项，其中的数据列给出了指针 optval 必须指向的每个选项的数据类型，如果数据类型后面有一对，则表明它是一个结构体

套接字选项粗分为两大基本类型：一是启用或禁止某个特性的二元选项（称为标志选项），二是取得并返回我们可以设置或检查的特定值的选项（也就是可以设置很多种值的选项），标有标志的列指出一个选项是否为标志选项，当个体这些标志选项调用 getsockopt 函数时，*optval 是一个整数，*optval 中的的返回值为 0 表示相应选项被禁止，类似的 setsockopt 函数需要一个不为 0 的 *optval 来启用选项，一个为 0 的 *optval 来禁用选项，如果标志列不含有 *，那么相应选项用于在用户进程与系统进程之间传递所指定数据类型的值

level（级别）	*optname*（选项名）	get	set	说　明	标志	数据类型
IPPROTO_TCP	TCP_MAXSEG	•	•	TCP最大分节大小		int
	TCP_NODELAY	•	•	禁止Nagle算法	•	int
IPPROTO_SCTP	SCTP_ADAPTION_LAYER	•	•	适配层指示		sctp_setadaption{}
	SCTP_ASSOCINFO	†	•	检查并设置关联信息		sctp_assocparams{}
	SCTP_AUTOCLOSE	•	•	自动关闭操作		int
	SCTP_DEFAULT_SEND_PARAM	•	•	默认发送参数		sctp_sndrcvinfo{}
	SCTP_DISABLE_FRAGMENTS	•	•	SCTP分片	•	int
	SCTP_EVENTS	•	•	感兴趣事件的通知		sctp_event_subscribe{}
	SCTP_GET_PEER_ADDR_INFO	†		获取对端地址状态		sctp_paddrinfo{}
	SCTP_I_WANT_MAPPED_V4_ADDR	•	•	映射的v4地址	•	int
	SCTP_INITMSG	•	•	默认的INIT参数		sctp_initmsg{}
	SCTP_MAXBURST	•	•	最大猝发大小		int
	SCTP_MAXSEG	•	•	最大分片大小		int
	SCTP_NODELAY	•	•	禁止Nagle算法	•	int
	SCTP_PEER_ADDR_PARAMS	†	•	对端地址参数		sctp_paddrparams{}
	SCTP_PRIMARY_ADDR	†	•	主目的地址		sctp_setprim{}
	SCTP_RTOINFO	†	•	RTO信息		sctp_rtoinfo{}
	SCTP_SET_PEER_PRIMARY_ADDR		•	对端的主目的地址		sctp_setpeerprim{}
	SCTP_STATUS	†		获取关联状态		sctp_status{}

level（级别）	*optname*（选项名）	get	set	说　明	标志	数据类型
SOL_SOCKET	SO_BROADCAST	•	•	允许发送广播数据报	•	int
	SO_DEBUG	•	•	开启调试跟踪	•	int
	SO_DONTROUTE	•	•	绕过外出路由表查询	•	int
	SO_ERROR	•		获取待处理错误并清除		int
	SO_KEEPALIVE	•	•	周期性测试连接是否仍存活	•	int
	SO_LINGER	•	•	若有数据待发送则延迟关闭		linger{}
	SO_OOBINLINE	•	•	让接收到的带外数据继续在线留存	•	int
	SO_RCVBUF	•	•	接收缓冲区大小		int
	SO_SNDBUF	•	•	发送缓冲区大小		int
	SO_RCVLOWAT	•	•	接收缓冲区低水位标记		int
	SO_SNDLOWAT	•	•	发送缓冲区低水位标记		int
	SO_RCVTIMEO	•	•	接收超时		timeval{}
	SO_SNDTIMEO	•	•	发送超时		timeval{}
	SO_REUSEADDR	•	•	允许重用本地地址	•	int
	SO_REUSEPORT	•	•	允许重用本地端口	•	int
	SO_TYPE	•		取得套接字类型		int
	SO_USELOOPBACK	•	•	路由套接字取得所发送数据的副本	•	int
IPPROTO_IP	IP_HDRINCL	•	•	随数据包含的IP首部	•	int
	IP_OPTIONS	•	•	IP首部选项		（见正文）
	IP_RECVDSTADDR	•	•	返回目的IP地址	•	int
	IP_RECVIF	•	•	返回接收接口索引	•	int
	IP_TOS	•	•	服务类型和优先权		int
	IP_TTL	•	•	存活时间		int
	IP_MULTICAST_IF	•	•	指定外出接口		in_addr{}
	IP_MULTICAST_TTL	•	•	指定外出TTL		u_char
	IP_MULTICAST_LOOP	•	•	指定是否环回		u_char
	IP_ADD_MEMBERSHIP		•	加入多播组		ip_mreq{}
	IP_DROP_MEMBERSHIP		•	离开多播组		ip_mreq{}
	IP_BLOCK_SOURCE		•	阻塞多播源		ip_mreq_source{}
	IP_UNBLOCK_SOURCE		•	开通多播源		ip_mreq_source{}
	IP_ADD_SOURCE_MEMBERSHIP		•	加入源特定多播组		ip_mreq_source{}
	IP_DROP_SOURCE_MEMBERSHIP		•	离开源特定多播组		ip_mreq_source{}
IPPROTO_ICMPV6	ICMP6_FILTER	•	•	指定待传递的ICMPv6消息类型		icmp6_filter{}
IPPROTO_IPV6	IPV6_CHECKSUM	•	•	用于原始套接字的校验和字段偏移		int
	IPV6_DONTFRAG	•	•	丢弃大的分组而非将其分片	•	int
	IPV6_NEXTHOP	•	•	指定下一跳地址		sockaddr_in6{}
	IPV6_PATHMTU	•		获取当前路径MTU		ip6_mtuinfo{}
	IPV6_RECVDSTOPTS	•	•	接收目的地选项	•	int
	IPV6_RECVHOPLIMIT	•	•	接收单播跳限	•	int
	IPV6_RECVHOPOPTS	•	•	接收步跳选项	•	int
	IPV6_RECVPATHMTU	•	•	接收路径MTU	•	int
	IPV6_RECVPKTINFO	•	•	接收分组信息	•	int
	IPV6_RECVRTHDR	•	•	接收源路径	•	int
	IPV6_RECVTCLASS	•	•	接收流通类别	•	int
	IPV6_UNICAST_HOPS	•	•	默认单播跳限		int
	IPV6_USE_MIN_MTU	•	•	使用最小MTU	•	int
	IPV6_V6ONLY	•	•	禁止v4兼容	•	int
	IPV6_*XXX*	•	•	黏附性辅助数据		（见正文）
	IPV6_MULTICAST_IF	•	•	指定外出接口		u_int
	IPV6_MULTICAST_HOPS	•	•	指定外出跳限		int
	IPV6_MULTICAST_LOOP	•	•	指定是否环回	•	u_int
	IPV6_JOIN_GROUP		•	加入多播组		ipv6_mreq{}
	IPV6_LEAVE_GROUP		•	离开多播组		ipv6_mreq{}
IPPROTO_IP或 IPPROTO_IPV6	MCAST_JOIN_GROUP		•	加入多播组		group_req{}
	MCAST_LEAVE_GROUP		•	离开多播组		group_source_req{}
	MCAST_BLOCK_SOURCE		•	阻塞多播源		group_source_req{}
	MCAST_UNBLOCK_SOURCE		•	开通多播源		group_source_req{}
	MCAST_JOIN_SOURCE_GROUP		•	加入源特定多播组		group_source_req{}
	MCAST_LEAVE_SOURCE_GROUP		•	离开源特定多播组		group_source_req{}

2.21 setsockopt

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

设置套接字选项

2.22 fcntl

```
int fcntl(int fd, int cmd, ...);
```

文件描述符控制

F_SETFL	设置文件状态标示 (全局文件打开表中，可设置 O_XXX)
F_SETOWN	设置描述符属主
F_SETFD	设置文件描述符标志 (进程文件描述符表中,FD_CLOEXEC 标志)
F_DUPDF	复制文件描述符但不会复制 FD_CLOEXEC 标志
F_DUPDF_CLOEXEC	复制文件描述符但会复制 FD_CLOEXEC 标志

```
fcntl(fd, F_SETOWN, pid) //设置接收 SIGIO 的进程，不能对终端设备使用

fcntl(fd, F_SETFL, flags|O_ASYNC) //设置信号 IO
```

2.23 readv

```
ssize_t readv(int fd, const struct iovec *iov, int iovcnt);
```

散布读

iovec * 指向一个数组结构，len 表示数组长度，从 fd 读取数据，按 iovec 数组下标从小到大读取数据到数组元素中所指向的缓冲区中，也就是说如果 iovec[0] 指向的缓冲区装满后，然后存入 iovec[1] 指向的缓冲区

2.24 writev

```
ssize_t writev(int fd, const struct iovec *iov, int iovcnt);
```

聚集写

向 fd 写数据，按 iovec 数组下标从小到大写缓冲区的 iov_len 个数据到 fd 中，也就是说如果 iovec[0] 指向的缓冲区写到 fd 后，然后写入 iovec[1] 指向的缓冲区，如果自己要设置某种信息协议，比如发送的数据以某特定数据开头，特定数据结尾，则此时可以设置三个 iovec，分别用于头部数据，中间数据，尾部数据

2.25 recvfrom

```
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags, struct
↪  sockaddr *src_addr, socklen_t *addrlen);
```

在一个套接字上接收数据

前三个参数和 read 函数一样，第四个参数通常指定为 0，第五个参数和第六个参数用于获取发信人的协议地址，类似于 acccept 的第二和第三个参数，返回值也类似 write，一般用于 UDP

当对 UDP 调用 connect 后，可以使用 read，recv 或 recvmsg 进行接受数据，此时只会接受源地址为调用 connect 时指定的 IP 地址的 UDP 包

2.26 sendto

```
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags, const
↪  struct sockaddr *dest_addr, socklen_t addrlen);
```

在一个套接字上发送数据

前三个参数和 write 函数一样，第四个参数通常指定为 0，第五个参数和第六个参数用于指定发送目的地，类似于 bind 的第二和第三个参数，区别是指定的是目的主机地址，返回值也类似 read，一般用于 UDP

通常对于客户端来说，不会对它调用 bind 函数，对于 TCP，在它调用 connet 函数时会由内核来为它分配一个临时端口号，对于 UDP 来说，会在它首次调用 sendto 时由内核为它选择一个临时端口

当对 UDP 调用 connect 后，无法使用带地址的 sendto，可以使用 write 或 send

2.27 recv

```
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

在一个套接字上接收数据

前三个参数和 read 一样，但是可以使用 flags 参数进行一些控制操作

flags	说明
MSG_DONTWAIT	仅本操作非阻塞
MSG_OOB	发送或接受带外数据
MSG_PEEK	接受数据并保留副本在缓冲区
MSG_WAITALL	等待所有数据

- MSG_DONTWAIT: 本标志在无需打开相应套接字非阻塞标志的前提下，把单个 IO 操作临时指定为非阻塞，接着执行 IO 操作，然后关闭非阻塞标志
- MSG_OOB: 对于 recv，本标志标明即将读入的是带外数据而不是普通数据
- MSG_PEEK: 本标志适用于 recv 和 recvfrom，它允许我们查看已读取的数据，获取缓冲区中数据的一份副本，不会将数据从缓冲区移除
- MSG_WAITALL: 告知内核不要在尚未读取请求数目的字节之前让一个读操作返回，不过在捕获一个信号，连接被终止，套接字发生一个错误的情况下读函数仍然有可能返回比所请求字节数要少的数据

2.28 send

```
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
```

在一个套接字上发送数据

前三个参数和 write 一样，但是可以使用 flags 参数进行一些控制操作

flags	说明
MSG_DONTROUTE	绕过路由表查找
MSG_DONTWAIT	仅本操作非阻塞
MSG_OOB	发送或接受带外数据

- MSG_DONTROUTE: 本标志告知内核目的主机在某个直接连接的本地网络上，因此无需执行路由表查找，此操作等同于对套接字设置 SO_DONTROUTE，但是 SO_DONTROUTE 会对针对该套接字的所有 IO 操作都执行该特性
- MSG_DONTWAIT: 本标志在无需打开相应套接字非阻塞标志的前提下，把单个 IO 操作临时指定为非阻塞，接着执行 IO 操作，然后关闭非阻塞标志
- MSG_OOB: 对于 send，本标志标明即将发送带外数据，TCP 上只有一个字节大小可以作为代外数据发送

2.29 recvmsg

```
ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);
```

在一个套接字上接受数据

可以看作是使用套接字的 readv, 接受数据后，msghdr 中的 msg_flags 元素的可能值有 MSG_CTRUNC|MSG_EOR|MSG_ERRQUEUE|MSG_OOB|MSG_TRUNC MSG_CTRUNC 表示控制数据被截断，MSG_EOR 表示接受记录结束符，MSG_ERRQUEUE 表示接受错误信息作为辅助数据，MSG_OOB 表示接受带外数据，MSG_TRUNC 表示一般数据被截断

2.30 sendmsg

```
ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags);
```

在一个套接字上发送数据

可以看作是使用套接字的 writev，msghdr.control 实际上是一个指向 cmsghdr 的指针，cmsghdrsocklen_t cmsg_len; int cmsg_level; int cmsg_type 为了发送文件描述符，将 cmsg_len 设置为 cmsghdr 结构的长度加一个整形的长度 (描述符的长度),cmsg_level 字段设置为 SOL_SOCKET，cmsg_type 设置为 SCM_RIGHTS，用以表明在传送访问权 (SCM 是套接字级控制信息的缩写)，访问权限仅能通过 UNIX 域套接字发送，描述符仅随 cmsg_type 后存储

2.31 gethostbyname

```
struct hostent *gethostbyname(const char *name);
```

查询域名服务器，获取一个域名的 IP 地址信息

虽然说此函数以过时，但是短时间内想要完全废弃时不可能的，此函数只能处理 IPv4 地址，也就是说只能查询域名服务器里面的 A 类记录，后面可以使用 gethostinfo 代替

函数出错时不会设置 errno 变量，而是会设置 h_errno 变量，这个变量设置在 netdb.h 头文件中

- HOST_NOT_FOUND
- TRY_AGAIN
- NO_RECOVERY
- NO_DATA

大多数系统都提供了一个 hstrerrno 的函数，它以某个 h_errno 值作为一个唯一的参数，返回的是一个 const char * 指针，指向对应错误的说明

以下是一个 gethostbyname 的使用例子

```
# include </Users/dengyan/ClionProjects/Linux/linux.h>

int main(int argc, char *argv[]) {
    struct hostent * result;
    char buf[100];
    char ** p;

    if((result = gethostbyname("www.baidu.com")) == NULL) {
        printf("%s", hstrerror(h_errno));
        exit(-1);
    }

    p = result->h_aliases;
```

```
14      while (*p != NULL) {
15          printf("%s", *p);
16          p++;
17      }
18
19      p = result->h_addr_list;
20      while (*p != NULL) {
21          printf("%s\n", inet_ntop(AF_INET, *p, buf, sizeof(buf)));
22          p++;
23      }
24      return 0;
25  }
```

2.32 gethostbyaddr

```
struct hostent *gethostbyaddr(const void *addr, socklen_t len, int type);
```

使用 IP 地址查找主机名

功能上与 gethostbyname 相反，在这里我们所关心的是 hostent 结构体中的 h_name 域

2.33 getservbyname

```
struct servent *getservbyname(const char *name, const char *proto);
```

根据服务名和协议名查询信息

使用本地文件/etc/services

name 表示服务名，proto 表示协议名，根据服务名 (如 ssh) 和协议名 (如 tcp) 查询信息，这里主要关注的是端口信息

2.34 getservbyport

```
struct servent *getservbyport(int port, const char *proto);
```

根据端口号和协议名查询信息

使用本地文件/etc/services

port 表示端口号，proto 表示协议名，根据端口名 (如 23，需要使用网络序) 和协议名 (tcp) 查询信息

```
/* 三个相关函数 */
struct servent *getservent(void); //获取文件下一条目
void setservent(int stayopen); //打开端口绑定的服务名和端口号信息文件，mac 上即/etc/services 文件
void endservent(void); //关闭文件
```

2.35 getaddrinfo

```
int getaddrinfo(const char *node, const char *service, const struct
↪ addrinfo *hints, struct addrinfo **res);
```

进行域名到 IP 以及服务名到端口号的转换

等价于一个支持 IPv6 的 gethostbyname 加 getservbyname 之和

node 参数即可与为域名，也可以为点分十进制字符串

hints 用于暗示返回的值，返回值将由 res 指向，也就是说传入的 res 参数是是一个指针变量的地址

UNP p249

以下是几个 getaddrinfo 的借口函数

2.35.1 host_serv

```
1  struct addrinfo * host_serv(const char * host, const char * serv, int
   ↪ family, int socktype)
2  {
3      struct addrinfo hints, *res;
4
5      bzero(&hints, sizeof(struct addrinfo));
6      hints.ai_flags = AI_CANONNAME;
7      hints.ai_family = family;
8      hints.ai_socktype = socktype;
9
10     if (getaddrinfo(host, serv, &hints, &res) != 0)
11         return  NULL;
12     return res;
13 }
```

2.35.2 tcp_connect

```
1  int tcp_connect(const char * host, const char * serv)
2  {
3      int sockfd, n;
4      struct addrinfo hints, *res, *ressave;
5
6      bzero(&hints, sizeof(struct addrinfo));
7      hints.ai_family = AF_UNSPEC;
8      hints.ai_socktype = SOCK_STREAM;
9      hints.ai_flags = AI_NUMERICSERV;
10
11     if ((n = getaddrinfo(host, serv, &hints, &res)) != 0)
12         err_quit("tcp_connect error for %s, %s:%s", host, serv,
           ↪  gai_strerror(n));
13
14     ressave = res;
15     do {
16         sockfd = socket(res->ai_family, res->ai_socktype,
           ↪  res->ai_protocol);
17         if (sockfd < 0)
18             continue;
19
20         if (connect(sockfd, res->ai_addr, res->ai_addrlen) == 0)
21             break;
22
23         close(sockfd);
24     } while ((res = res->ai_next) != NULL);
25
26     if (res == NULL)
27         err_quit("tcp_connect error for %s, %s", host, serv);
28
29     freeaddrinfo(ressave);
30
31     return sockfd;
32 }
```

2.35.3 tcp_connect

```
int tcp_listen(const char * host, const char * serv, socklen_t *
    addrlenp)
{
    int listenfd, n;
    const int on = 1;
    struct addrinfo hints, *res, *ressave;

    bzero(&hints, sizeof(struct addrinfo));
    hints.ai_flags = AI_PASSIVE;
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_NUMERICSERV;

    if ((n = getaddrinfo(host, serv, &hints, &res)) != 0)
        err_quit("tcp_listen error for %s, %s:%s", host, serv,
            gai_strerror(n));
    ressave = res;
    do {
        listenfd = socket(res->ai_family, res->ai_socktype,
            res->ai_protocol);
        if (listenfd < 0)
            continue;

        setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
        if (bind(listenfd, res->ai_addr, res->ai_addrlen) == 0)
            break;

        close(listenfd);
    } while ((res = res->ai_next) != NULL);

    listen(listenfd, 5);
    if (addrlenp)
        *addrlenp = res->ai_addrlen;

    return listenfd;
}
```

2.36 gai_strerror

```
const char *gai_strerror(int errcode);
```

如果 getaddrinfo 失败，使用此函数将 getaddrinfo 的返回值转换成错误信息

2.37 freeaddrinfo

```
void freeaddrinfo(struct addrinfo *res);
```

释放 getaddrinfo 第四个参数指向的 addrinfo 结构

2.38 getnameinfo

```
int getnameinfo(const struct sockaddr *addr, socklen_t addrlen, char
↪  *host, socklen_t hostlen, char *serv, socklen_t servlen, int flags);
```

根据套接字地址获取主机名和服务名

常值	说明
NI_DGRAM	数据报服务
NI_NAMEREQD	若不能从地址解析出名字则返回错误
NI_NOFQDN	只返回 FQDN 的主机名部分
NI_NUMERICHOST	以数字串格式返回主机字符串
NI_NUMERICSCOPE	以数字串格式返回范围表示标示字符串
NI_NUMERICSERV	以数字串格式返回服务字符串

表 1: getnameinfo 的标志值

2.39

3 例子

3.1 时间客户端及服务器端

3.1.1 客户端

```
1  # include </Users/dengyan/ClionProjects/Linux/linux.h>
2
3  int main(int argc, char *argv[]) {
4      int sockfd, n;
5      char buf[4097];
6      struct sockaddr_in servaddr;
7
8      if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
9          err_sys("socket error");
10
11     bzero(&servaddr, sizeof(servaddr));
12     servaddr.sin_family = AF_INET;
13     servaddr.sin_port = htons(13333);
14     if (inet_pton(AF_INET, "127.0.0.1", &servaddr.sin_addr) == -1)
15         err_quit("inet_pton error");
16
17     if (connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr)) <
       ↪ 0)
18         err_sys("connect error");
19
20     while ((n = read(sockfd, buf, 4096)) > 0) {
21         write(STDOUT_FILENO, buf, n);
22     }
23     return 0;
24 }
```

3.1.2 服务器端

```
1  # include </Users/dengyan/ClionProjects/Linux/linux.h>
2
3  int main(int argc, char *argv[]) {
4      int sockfd, n, fd;
5      struct sockaddr_in servaddr;
```

```
6
7       if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
8           exit(-1);
9
10      bzero(&servaddr, sizeof(servaddr));
11      servaddr.sin_port = htons(13333);
12      servaddr.sin_family = AF_INET;
13      if (inet_pton(AF_INET, "127.0.0.1", &servaddr.sin_addr) <= 0)
14          exit(-1);
15
16      if (bind(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr)) !=
        ↪  0) {
17          exit(-1);
18      }
19
20      listen(sockfd, 5);
21
22      while (1) {
23          if ((fd = accept(sockfd, NULL, NULL)) < 0) {
24              printf("%s", strerror(errno));
25              continue;
26          }
27          time_t t = time(NULL);
28          char * str = ctime(&t);
29          write(fd, str, strlen(str));
30          close(fd);
31      }
32
33  }
```

3.2 使用 select 实现的 echo 服务器 (TCP 及 UDP 同时支持)

3.2.1 客户端

```
1  # include </Users/dengyan/ClionProjects/Linux/linux.h>
2
3  int main(int argc, char *argv[]) {
4      char buf[4096];
5      int tcpfd, udpfd, len, choose, n;
6      struct sockaddr_in servaddr, cliaddr;
7
8      if ((tcpfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
9          err_sys("tcp socket error");
10
11     bzero(&servaddr, sizeof(servaddr));
12     servaddr.sin_family = AF_INET;
13     servaddr.sin_port = htons(23333);
14     servaddr.sin_addr.s_addr = htonl(INADDR_LOOPBACK);
15
16     if ((udpfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
17         err_sys("udp socket error");
18
19     if (connect(tcpfd, (struct sockaddr *)&servaddr, sizeof(servaddr)) <
       ↪ 0)
20         err_sys("connect error");
21
22
23     connect(udpfd, (struct sockaddr *)&servaddr, sizeof(servaddr));
24
25     for (;;) {
26         printf("1: TCP echo\n");
27         printf("2: UDP echo\n");
28
29         scanf("%d", &choose);
30         switch (choose) {
31             case 1:
32                 n = read(STDIN_FILENO, buf, 4096);
33                 write(tcpfd, buf, n);
34                 n = read(tcpfd, buf, n);
```

```
35                  write(STDOUT_FILENO, buf, n);
36                  break;
37              case 2:
38                  n = read(STDIN_FILENO, buf, 4096);
39                  write(udpfd, buf, n);
40                  n = read(udpfd, buf, n);
41                  write(STDOUT_FILENO, buf, n);
42                  break;
43              default:
44                  break;
45          }
46      }
47
48      return 0;
49  }
```

3.2.2 服务器端

```
1  # include </Users/dengyan/ClionProjects/Linux/linux.h>
2
3  void handler(int s)
4  {
5      pid_t pid;
6      while ((pid = waitpid(-1, NULL, WNOHANG)) > 0)
7          printf("child pid %d terminate ", pid);
8  }
9
10 int main(int argc, char *argv[]) {
11     int listenfd, clifd, udpfd, n;
12     socklen_t len;
13     int on = 1, maxfd = -1;
14     struct sockaddr_in servaddr, cliaddr;
15
16     if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
17         err_quit("socket error");
18
19     bzero(&servaddr, sizeof(servaddr));
20     servaddr.sin_family = AF_INET;
```

```
21      servaddr.sin_port = htons(23333);
22      servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
23
24      signal(SIGCHLD, handler);
25
26      if (setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on)) ==
        ↪  -1)
27          err_quit("setsockopt error");
28
29      if (bind(listenfd, (struct sockaddr *)&servaddr, sizeof(servaddr)) <
        ↪  0)
30          err_quit("bind error");
31
32      listen(listenfd, 5);
33
34      if ((udpfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
35          err_quit("udp socket error");
36
37      if (bind(udpfd, (struct sockaddr *)&servaddr, sizeof(servaddr)) < 0)
38          err_quit("bind error");
39
40      maxfd = listenfd > udpfd?listenfd:udpfd;
41      fd_set set;
42      FD_ZERO(&set);
43
44      for ( ;; ) {
45          FD_SET(listenfd, &set);
46          FD_SET(udpfd, &set);
47
48          if (select(maxfd + 1, &set, NULL, NULL, NULL) < 0) {
49              if (errno == EINTR)
50                  continue;
51              else
52                  err_sys("select error");
53          }
54
55          if (FD_ISSET(listenfd, &set)) {
```

```
56              char name[40];
57              len = sizeof(cliaddr);
58              clifd = accept(listenfd, (struct sockaddr *)&cliaddr, &len);
59              printf("connect from %s:%d\n", inet_ntop(AF_INET,
                ↪  &cliaddr.sin_addr, name, sizeof(name)),
                ↪  ntohs(cliaddr.sin_port));
60              fflush(stdout);
61
62              if (fork() == 0) {
63                  char buf[4096];
64                  close(listenfd);
65                  while ((n = read(clifd, buf, sizeof(buf))) > 0) {
66                      write(clifd, buf, n);
67                  }
68                  exit(-1);
69              }
70              close(clifd);
71          }
72
73          if (FD_ISSET(udpfd, &set)) {
74              char buf[4096];
75              len = sizeof(cliaddr);
76              n = recvfrom(udpfd, buf, sizeof(buf), 0, (struct sockaddr
                ↪  *)&cliaddr, &len);
77              sendto(udpfd, buf, n, 0, (struct sockaddr *)&cliaddr, len);
78          }
79      }
80  }
```

3.3 使用 getaddrinfo 基于 tcp 实现的时间服务器

3.3.1 客户端

```
1  # include </Users/dengyan/ClionProjects/Linux/linux.h>
2
3  int main(int argc, char *argv[]) {
4      int sockfd, n;
5      char buf[100];
6      char recvline[MAXLINE + 1];
7      socklen_t len;
8      struct sockaddr_storage ss;
9
10     sockfd = tcp_connect("localhost", "13333");
11     len = sizeof(ss);
12     getpeername(sockfd, (struct sockaddr *)&ss, &len);
13     printf("connected to %s\n", len==16?inet_ntop(AF_INET, &((struct
       ↪  sockaddr_in *)&ss)->sin_addr, buf, 100):inet_ntop(AF_INET6,
       ↪  &((struct sockaddr_in *)&ss)->sin_addr, buf, 100));
14
15     while ((n = read(sockfd, recvline, MAXLINE)) > 0) {
16         write(STDOUT_FILENO, recvline, n);
17     }
18     return 0;
19 }
```

3.3.2 服务器端

```
1  # include </Users/dengyan/ClionProjects/Linux/linux.h>
2
3  int main(int argc, char *argv[]) {
4      int listenfd, connfd;
5      socklen_t len;
6      char buf[MAXLINE];
7      time_t t;
8      struct sockaddr_storage cliaddr;
9
10     listenfd = tcp_listen(NULL, "13333", NULL);
11
```

```
12      for ( ;; ) {
13          len = sizeof(cliaddr);
14          connfd = accept(listenfd, (struct sockaddr *)&cliaddr, &len);
15          printf("connected to %s\n", len==16?inet_ntop(AF_INET, &((struct
            ↪  sockaddr_in *)&cliaddr)->sin_addr, buf,
            ↪  100):inet_ntop(AF_INET6, &((struct sockaddr_in
            ↪  *)&cliaddr)->sin_addr, buf, 100));
16
17          t = time(NULL);
18          char * s = ctime(&t);
19          write(connfd, s, strlen(s));
20          close(connfd);
21      }
22      return 0;
23  }
```

3.4 观察 TCP 阻塞窗口

3.4.1 客户端

```c
# include </Users/dengyan/ClionProjects/Linux/linux.h>

int main(int argc, char *argv[]) {
    int fd, n;
    char buf[MAXLINE];
    struct sockaddr_in servaddr;
    struct tcp_connection_info tcpinfo;
    int len = sizeof(tcpinfo);
    fd = socket(AF_INET, SOCK_STREAM, 0);

    int sfd = open("/dev/zero", O_RDONLY);
    read(sfd, buf, 1000);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_port = htons(23333);
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_LOOPBACK);

    connect(fd, &servaddr, sizeof(servaddr));
    printf(" 接收窗口 发送窗口 发送阻塞窗口\n");
    while (write(fd, buf, 1000) > 0)
    {
        getsockopt(fd, IPPROTO_TCP, TCP_CONNECTION_INFO, &tcpinfo, &len);
        printf("%d %d %d\n", tcpinfo.tcpi_rcv_wnd, tcpinfo.tcpi_snd_wnd,
        ↪ tcpinfo.tcpi_snd_cwnd);
        sleep(0.2);
    }
}
```

3.4.2 服务器端

```c
# include </Users/dengyan/ClionProjects/Linux/linux.h>

int main(int argc, char *argv[]) {
    setbuf(stdout, NULL);
```

```
5       int listenfd, n = 4;
6       int rcv = 0;
7       char buf[MAXLINE];
8       struct tcp_connection_info tcpinfo;
9       int len = sizeof(tcpinfo);
10
11      struct addrinfo hints, *res, *ressave;
12
13      bzero(&hints, sizeof(hints));
14      hints.ai_family = AF_INET;
15      hints.ai_socktype = SOCK_STREAM;
16      hints.ai_flags = AI_PASSIVE|AI_NUMERICSERV;
17
18      if (getaddrinfo(NULL, "23333", &hints, &res) != 0)
19          err_quit("getaddrinfo error");
20
21      ressave = res;
22      do {
23          listenfd = socket(AF_INET, SOCK_STREAM, 0);
24
25          setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &n, sizeof(int));
26
27          if (bind(listenfd, res->ai_addr, res->ai_addrlen) == 0)
28              break;
29
30          close(listenfd);
31      } while ((res = res->ai_next) != NULL);
32
33      if (res == NULL)
34          err_quit("don't find the socket to bind");
35
36      freeaddrinfo(ressave);
37
38      getsockopt(listenfd, SOL_SOCKET, SO_RCVBUF, &rcv, &n);
39      printf("rcvbuf = %d\n", rcv);
40      rcv = 262000;
41      setsockopt(listenfd, SOL_SOCKET, SO_RCVBUF, &rcv, n);
```

```
42      printf("rcvbuf = %d\n", rcv);
43      listen(listenfd, 5);
44
45      int fd = accept(listenfd, NULL, 0);
46      printf(" 接收窗口 发送窗口 发送阻塞窗口\n");
47
48      for (;;) {
49          getsockopt(fd, IPPROTO_TCP, TCP_CONNECTION_INFO, &tcpinfo, &len);
50          printf("%d %d %d\n", tcpinfo.tcpi_rcv_wnd, tcpinfo.tcpi_snd_wnd,
            ↪  tcpinfo.tcpi_snd_cwnd);
51          sleep(1);
52      }
53      return 0;
54  }
```

3.5 XX

3.5.1 客户端

```
1
```

3.5.2 服务器端

```
1
```

4 结构体

4.1 netinet/in.h

4.1.1 in_addr

```
struct in_addr {
    in_addr_t s_addr; //IPv4 地址，通常为 uint32_t
};
/*
 * 需要注意的是，对 32 位 IPv4 地址进行访问时有两种不同的访问方法
 * 一种是访问结构体，另一种是访问结构体中的成员
 * 在当中参数时需要小心处理，因为编译器处理结构体和
 * 无符号整形采用不同的方式，结构体作为左值时无法随意赋值
 * 所以如果想直接对地址赋值需要访问结构体中的成员
 */
```

4.1.2 sockaddr_in

```
struct sockaddr_in {
    __uint8_t   sin_len; //通常不需要设置和检查
    sa_family_t sin_family; //地址族 (AF_INET)，通常为 8 位的无符号整形
    in_port_t   sin_port; //TCP 或 UDP 端口，一般为 uint16_t
    struct in_addr sin_addr; //IPv4 地址，通常为 uint32_t
    char        sin_zero[8]; //用于填充，达到最小 16 个字节
};
```

4.1.3 in6_addr

```
//此结构在 mac 上为 <netinet6/in6.h> 中
typedef struct in6_addr {
    union {
        __uint8_t   __u6_addr8[16];
        __uint16_t  __u6_addr16[8];
        __uint32_t  __u6_addr32[4];
    } __u6_addr; /* 128-bit IP6 address */
} in6_addr_t;
//通配地址赋值给此结构体可以使用 in6_addr = in6addr_any
```

4.1.4 sockaddr_in6

```
//此结构在 mac 上为 <netinet6/in6.h> 中
struct sockaddr_in6 {
    __uint8_t   sin6_len; //通常不需要设置和检查
    sa_family_t sin6_family; //地址族 (AF_INET6)
    in_port_t   sin6_port;
    __uint32_t  sin6_flowinfo;
    struct in6_addr^^Isin6_addr;
    __uint32_t  sin6_scope_id;
};
```

4.2 sys/socket.h

4.2.1 sockaddr

```
//16 个字节，不足以容纳容纳 IPv6 结构
struct sockaddr {
    __uint8_t    sa_len;
    sa_family_t sa_family;
    char        sa_data[14];
};
```

4.2.2 sockaddr_storage

```
/*
 * 由于 sockaddr 不足以容纳所有的地址结构，所以新的通用套接字结构产生了
 * 它足以容纳所有类型的套接字地址结构，而且满足最苛刻的对齐要求
 * 但是需要将其转换为特定的类型后，才可以访问其内部字段
 */
struct sockaddr_storage {
    __uint8_t   ss_len;
    sa_family_t ss_family;
    char        __ss_pad1[_SS_PAD1SIZE];
    __int64_t   __ss_align;
    char        __ss_pad2[_SS_PAD2SIZE];
};
```

4.2.3 iovec

```
struct iovec {
    void  *iov_base;    /* 缓冲区地址 */
    size_t iov_len;     /* 缓冲区有效数据长度 */
};
```

4.2.4 msghdr

```
struct msghdr {
    void         *msg_name;       /* 地址 */
    socklen_t     msg_namelen;    /* 地址长度 */
    struct iovec *msg_iov;        /* 用于散步读聚集写的数组 */
    size_t        msg_iovlen;     /* 数组长度 */
    void         *msg_control;    /* 指向控制信息头 */
    size_t        msg_controllen; /* 控制信息大小 */
    int           msg_flags;      /* 标志 */
};
```

4.3 netdb.h

4.3.1 hostent

```
struct hostent {
    char  *h_name;            /* 正式主机名 */
    char **h_aliases;         /* 别名列表 */
    int    h_addrtype;        /* 主机地址类型: AF_INET */
    int    h_length;          /* 地址长度: 4 */
    char **h_addr_list;       /* 其中的每个指针指向 in_addr 结构体 */
}
```

4.3.2 servent

```
struct servent {
    char  *s_name;        /* 正式服务名 */
    char **s_aliases;     /* 别名列表 */
    int    s_port;        /* 端口号 (网络序) */
    char  *s_proto;       /* 使用的协议名 */
}
```

4.3.3 addrinfo

```
struct addrinfo {
    int              ai_flags; /* 控制返回信息 */
    int              ai_family; /* AF_INET|AF_INET6|AF_UNSPEC 意义为获取
    ↪ 哪种地址结构 */
    int              ai_socktype; /* SOCK_XXX */
    int              ai_protocol; /* 一般为 0 */
    socklen_t        ai_addrlen; /* ai_addr 长度 */
    struct sockaddr *ai_addr; /* 存放地址信息 */
    char            *ai_canonname;
    struct addrinfo *ai_next; /* 指向链表上的下一个 addrinfo */
};

/* ------------ai_flags------------
 * AI_PASSIVE 套接字将用于被动打开 (服务器端)
 * AI_CANONNAME 告知 getaddrinfo 返回主机的规范名字
 * AI_V4MAPPED 如果 ai_family 指定的是 IPv6, 那么如果没有可用的 AAAA 记录,
↪ 则返回一条 IPv4 映射的 IPv6 地址)
 * AI_ADDRCONFIG 表示按照主机配置配置选择返回的地址类型
 * AI_ALL 表示查找 IPv4 和 IPv6(IPv6 需要指定 AI_V4MAPPED 后还会返回 IPv4
↪ 映射的 IPv6 地址)
 * AI_NUMERICHOST 表示以数字格式指定主机地址
 * AI_NUMERICSERV 表示以数字形式 (端口号) 指定服务
 */
```