Linux 编程手册

常用命令篇

函数接口篇

1、main 函数参数解析

函数头文件：

```
#include <getopt.h>
```

函数原型：

```
extern char *optarg;
extern int optind, opterr, optopt;

int getopt_long(int argc, char * const argv[],const char *optstring,
                const struct option *longopts, int *longindex);

int getopt_long_only(int argc, char * const argv[],const char *optstring,
                    const struct option *longopts, int *longindex);
```

函数说明：

1、函数中的 argc 和 argv 通常直接从 main()的两个参数传递而来。

2、optsting 是选项参数组成的字符串：

字符串 optstring 可以下列元素：

1.单个字符，表示选项，

2.单个字符后接一个冒号：表示该选项后必须跟一个参数。参数紧跟在选项后或者以空格隔开。该参数的指针赋给 optarg。

3.单个字符后跟两个冒号，表示该选项后可以有参数也可以没有参数。如果有参数，参数必须紧跟在选项后不能以空格隔开。该参数的指针赋给 optarg。（这个特性是 GNU 的扩张）。

optstring 是一个字符串，表示可以接受的参数。例如，"a:b:c:d:"，表示可以接受的参数是 a,b,c,d，其中，a 和 b 参数后面跟有更多的参数值。(例如：-a host -b name)

3、参数 longopts，其实是一个结构的实例：

```
struct option {
  const char *name;
  int has_arg;
  int *flag;
  int val;
}
```

结构体参数说明：

name 表示的是长参数名或参数名

has_arg 有 3 个值：

no_argument(或者是 0)，表示该参数后面不跟参数值

required_argument(或者是 1),表示该参数后面一定要跟个参数值

optional_argument(或者是 2),表示该参数后面可以跟，也可以不跟参数值

flag：

用来决定，getopt_long()的返回值到底是什么。

如果 flag 是 null(通常情况)，则***函数会返回与该项 option 匹配的 val 值***;

如果 flag 不是 NULL，则***将 val 值赋予 flag 所指向的内存，并且返回值设置为 0***。

Val：

和 flag 联合决定返回值

4、longindex 如果非 NULL，则是返回识别到 struct option 数组中元素的位置指针

使用例程：

```
static const struct option lopts[] = {
        { "device",  2, NULL, 'D' },
```

```
        { "reload",  1, NULL, 'l' },
        { "cpuload", 0, NULL, 'u' },
        {"ipaddr", 1, NULL, 'A'},
        { NULL, 0, 0, 0 },
    };
int c;
while((c = getopt_long(argc, argv, " D:l:u:A", lopts, NULL))!= -1){
    printf("c =  %c \n",c);
    printf("optarg = %s \n",optarg);
    printf("optind = %d \n",optind);
    printf("argv[optind-1] = %s\n",argv[optind-1]);
    printf("opterr = %d \n",opterr);
    printf("optopt = %d \n",optopt);
}
```

getopt_long_only 用法见：[getopt long only 用法]

2、文件流

1.获取文件信息

头文件：

```
#include <sys/types.h>
#include <sys/stat.h>
```

结构体信息

```
struct stat {
       mode_t     st_mode;       //文件对应的模式，文件，目录等
       ino_t      st_ino;       //inode 节点号
       dev_t      st_dev;        //设备号码
       dev_t      st_rdev;       //特殊设备号码
       nlink_t    st_nlink;      //文件的连接数
       uid_t      st_uid;        //文件所有者
       gid_t      st_gid;        //文件所有者对应的组
       off_t      st_size;       //普通文件，对应的文件字节数
       time_t     st_atime;      //文件最后被访问的时间
       time_t     st_mtime;      //文件内容最后被修改的时间
       time_t     st_ctime;      //文件状态改变时间
       blksize_t st_blksize;    //文件内容对应的块大小
       blkcnt_t   st_blocks;     //伟建内容对应的块数量
      };
stat 结构体中的 st_mode 则定义了下列数种情况：
S_IFMT   0170000    文件类型的位遮罩
```

S_IFSOCK 0140000 scoket
S_IFLNK 0120000 符号连接
S_IFREG 0100000 一般文件
S_IFBLK 0060000 区块装置
S_IFDIR 0040000 目录
S_IFCHR 0020000 字符装置
S_IFIFO 0010000 先进先出
S_ISUID 04000 文件的(set user-id on execution)位
S_ISGID 02000 文件的(set group-id on execution)位
S_ISVTX 01000 文件的 sticky 位
S_IRUSR(S_IREAD) 00400 文件所有者具可读取权限
S_IWUSR(S_IWRITE)00200 文件所有者具可写入权限
S_IXUSR(S_IEXEC) 00100 文件所有者具可执行权限
S_IRGRP 00040 用户组具可读取权限
S_IWGRP 00020 用户组具可写入权限
S_IXGRP 00010 用户组具可执行权限
S_IROTH 00004 其他用户具可读取权限
S_IWOTH 00002 其他用户具可写入权限
S_IXOTH 00001 其他用户具可执行权限

上述的文件类型在 POSIX 中定义了检查这些类型的宏定义：

S_ISLNK (st_mode) 判断是否为符号连接
S_ISREG (st_mode) 是否为一般文件
S_ISDIR (st_mode) 是否为目录
S_ISCHR (st_mode) 是否为字符装置文件
S_ISBLK (s3e) 是否为先进先出
S_ISSOCK (st_mode) 是否为 socket

若一目录具有 sticky 位(S_ISVTX)，则表示在此目录下的文件只能被该文件所有者、此目录所有者或 root 来删除或改名，在 linux 中，最典型的就是这个/tmp 目录

有两种方式获取文件信息

通过路径

```
int stat(const char *path, struct stat *struct_stat);
int lstat(const char *path,struct stat *struct_stat);
```

两个函数的第一个参数都是文件的路径，第二个参数是 struct stat 的指针。返回值为 0，表示成功执行。

执行失败是，error 被自动设置为下面的值：

EBADF： 文件描述词无效
EFAULT： 地址空间不可访问
ELOOP： 遍历路径时遇到太多的符号连接
ENAMETOOLONG：文件路径名太长
ENOENT：路径名的部分组件不存在，或路径名是空字串
ENOMEM：内存不足
ENOTDIR：路径名的部分组件不是目录

这两个方法区别在于 stat 没有处理字符链接(软链接)的能力，如果一个文件是符号链接，stat 会直接返回它所指向的文件的属性；

而 lstat 返回的就是这个符号链接的内容。这里需要说明一下的是软链接和硬链接的含义。我们知道目录在 linux 中也是一个文件，文件的内容就是这这个目录下面所有文件与 inode 的对应关系。那么所谓的硬链接就是在某一个目录下面将一个文件名与一个 inode 关联起来，其实就是添加一条记录！而软链接也叫符号链接更加简单了，这个文件的内容就是一个字符串，这个字符串就是它所链接的文件的绝对或者相对地址。

通过文件描述符

```
int fstat(int fdp, struct stat *struct_stat);
```

通过文件描述符获取文件对应的属性。fdp 为文件描述符

2.文件流操作

1) 普通文件流

函数头文件：

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

函数说明：

1、打开文件的方式

a) FILE *fopen(const char *pathname,const char *type);

fopen 函数用来打开一个特定的文件

b) FILE *freopen(const char *pathname,const char *type, FILE *fp);

freopen 用来在一个特定的流上打开一个特定的文件，当调用 freopen 时，首先关闭 fp 流，然后重新使用这个 FILE 结构指针 fp 打开 pathname 所代表的文件，此函数常用来对标准输入、标准输出、标准错误输出等预定义的流进行重定向。

c) FILE *fdopen(int filedes,const char *type);

fdopen 用于将一个流和某一个已打开的特定文件相对应，filedes 表示此文件的描述符，当只有 type 所定义的模式和 filedes 所表示的文件的打开模式相同时，调用才能成功。此函数多用于建立某一流和无法用于 I/O 操作函

数的文件之间的关联，特别是管道文件和网络通信管道。

Type 说明如下：

type	文件类型	是否新建	是否清空	可读	可写	读写位置
r	文本文件	NO	NO	YES	NO	文件开头
r+	文本文件	NO	NO	YES	YES	文件开头
w	文本文件	YES	YES	NO	YES	文件开头
w+	文本文件	YES	YES	YES	YES	文件开头
a	文本文件	NO	NO	NO	YES	文件结尾
a+	文本文件	YES	NO	YES	YES	文件结尾
rb	二进制文件	NO	NO	YES	NO	文件开头
r+b 或 rb+	二进制文件	NO	NO	YES	YES	文件开头
wb	二进制文件	YES	YES	NO	YES	文件开头
w+b 或 wb+	二进制文件	YES	YES	YES	YES	文件开头
ab	二进制文件	NO	NO	NO	YES	文件结尾
a+b 或 ab+	二进制文件	YES	NO	YES	YES	文件结尾

2、关闭文件方式

```
int fclose(FILE *fp);
```

调用成功返回 0，调用失败返回-1，并设置 errno 的值，如果程序结束前没有执行流动关闭操作，有可能会造成写入的数据停留在缓冲区里没有保存到文件中，从而造成数据的丢失。

3、设置缓冲区属性

（1）int setbuf(FILE *fp,char *buf);

setbuf 函数用于将缓冲区设置为全缓冲或无缓冲，buf 为指向缓冲区的指针。当 buf 指向一个真实缓冲区地址时，将缓冲区设置为全缓冲，大小有常数 BUFSIZE 指定，当 buf 为 NULL 时，则设定为无缓冲，此函数一般用作激活或禁止缓冲区的开关。

（2）int setbuffer(FILE *fp,char *buf,size_t size);

setbuffer 函数与 setbuf 类似，区别在于可以有程序员设定缓冲区大小 size。

（3）int setlinebuf(FILE *buf);

setlinebuf 用于将缓冲区设定为行缓冲区。

（4）int setvbuf(FILE *fp,char *buf,int mode,size_t size);

setvbuf 的 mode 参数可以指定缓冲区类型，即： _IOFBF(全缓冲类型)、_IOLBF(行缓冲类型)、_IONBF(无缓冲类型)。

一般而言需要在流打开但没有执行其他操作时候设定其类型，因为改变缓冲区类型会对所指向的操作参数影响。

（5）int fflush(FILE *fp);

用于将缓冲区尚未写入文件的数据强制性的保存到文件中。

4、读取函数

size_t fread(void *ptr,size_t size,size_t nmemb,FILE *fp);
ptr 为存储要读取数据的缓冲区，size 为读取记录的大小，nmemb 是所读取记录的个数，fp 为所要读取的流的文件 FILE 指针,其返回值为实际读取的记录个数。

5、写入函数

size_t fwrite(const void *ptr,size_t size,size_t nmemb,FILE *fb);
ptr 为指向存放要输入数据的缓冲区指针，size 为写入记录的大小，nmemb 为所写记录的个数，fp 为所要写入的流的文件 FILE 指针,其返回值为实际写入的记录个数

6、文件结尾检查

（1）int feof(FILE *fp)

feof 函数用来检测是否读到文件的结尾，当没有访问到文件的结尾时，返回为 0，当访问到文件的结尾时，返回为 1，只有执行读操作时候才对文件结束标志进行操作

（2）int ferror(FILE *fp);

ferror 函数用来检测是否出现了读写错误，当访问正常接收时候，函数返回值为 0，当访问非正常结束时，返回值为非 0，并设置 errno 的值。此时 errno 的值为错误发生时由读写函数本身所设定的

7、清除文件流的错误旗标

```
void clearerr(FILE *fp)
```

8、将文件指针重新指向一个流的开头

```
int rewind(FILE *stream)
```

9、取得当前文件的句柄

```
int fgetpos(FILE *stream)
```

10、取得文件流的读取位置

```
long ftell(FILE * stream);
```

参数 stream 为已打开的文件指针。

返回值 当调用成功时则返回目前的读写位置，若有错误则返回-1，errno 会存放错误代码。

错误代码 EBADF 参数 stream 无效或可移动读写位置的文件流。

11、格式化输入输出

（1）格式化输出

函数 ：

```
(1) int printf(const *format,…);

(2) int fprintf(FILE *fp,const char *format,…);

(3) int sprintf(char *str,const char *format,…);
```

(4) int snprintf(char *str,size_t size,const format,…);

(5) int vprintf(const *format,va_list ap);

(6) int vfprintf(FILE *fp,const char *format, va_list ap);

(7) int vsprintf(char *str,const char *format, va_list ap);

(8) int vsnprintf(char *str,size_t size,const format, va_list ap);

（2）格式化输入

函数：

(1) int scanf(const char *format,…);

(2) int sscanf(char *str,const char *format,…);

(3) int fscanf(FILE *fp,const char *format,…);

(4) int vsscanf(char *str,const char *format,…);

(5) int vscanf(const char *format, va_list ap);

(6) int vfscanf(FILE *fp,const char *format, va_list ap);

(7) int vsscanf(char *str,const char *format, va_list ap);

例子：

12、临时文件

临时文件是指那些在程序运行期间存在并使用，而当程序运行完之后就删除的文件，临时文件的操作都是在基于流的基础上进行的。

（1）char * tmpnam(char *str);

tmpnam 作用是生成一个有效的文件名，该文件名不同于任何一个已经存在的文件名 ,str 指向用于保存新生成文件名的缓冲区。该缓冲区长度要大于等于 L_tmpnam，该常数在 stdio 库中定义。调用成功返回该缓冲区的指针，如果参数为 NULL，则在静态缓冲区中存放生成的文件名 ,但是要再次调用该函数时将改变这一区域中的文件名 ,因此，,如果要保存文件名的话 ,必须要开辟相应的缓冲区来存放文件名。当调用失败是返回空指针。此外，tmpnam 调用次数有所限制，必须小于等于

TMP_MAX 的值，该值在 stdio 库中定义。Tmpnam 生产的文件名不能指定文件的路径和前缀，通常位于/tmp 或/var/tmp 目录下。

（2）char *tempnam(const char *directory,const char *prefix);

Tempnam 函数与 tmpnam 的区别在于它可以指定临时文件所在的目录。

Directory 可以指定文件的路径名，路径名的选取一般如下：如果已定义了环境变量 TMPDIR，则使用 TMPDIR，如果没有定义，directory 不为空，则以 directory 所指向的缓冲区来做路径名。如果 TMPDIR 未定义，切 directory 为 NULL，则以 P_tmpdir 为路径名，P_tmpdir 由 stdio 定义，其优先级如下：

TMP>directory>P_tmpdir

Prefix 为指定文件的前缀，当 prefix 不为 NULL 时，其所指向的缓冲区存放的字符串指定的前缀。其生成的文件名放在一块动态存储区中，有该函数调用 malloc 来分配，函数返回之西那个存放文件名的缓冲区指针，该文件使用完毕后，需要调用 free 来释放。

（3）FILE *tmpfile(void);

tmpfile 用于打开一个临时文件，该函数将创建一个临时二进制文件，该文件会在程序结束后自动删除，调用 tmpfile 生成临时文件时，它首先会调用函数 tmpnam 生产临时文件的文件名，然后创建这一文件，在程序结束时，系统调用 ulink 函数，文件将会删除。

2）Linux 系统文件流操作

头文件：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

函数说明：

1、open 系统调用

```
int open(const char *path,int oflags)
int open(const char *path,int oflags,mode_t mode)
```

open 系统调用建立了一条从到文件或设备的访问路径，该调用将得到与该文件相关联的文件描述符(file descriptor,fd)任何一个进程可以同时打开的文件数目有限，通常由 limits.h 头文件中的常量 OPEN_MAX 定义，该值与系统有关，且这个限制本身还受到系统全局性限制影响，所以一个程序未必总是可以打开这么多文件。在 Linux 系统中，这个限制可以随着系统运行而调整，所以 OPEN_MAX 并不是一个常量。它通常一开始就被设置为 256

参数	说明
path	准备打开文件或设备的名字
oflags	打开文件所采取的动作，见表一
mode	文件访问模式

表一：

oflags 包含下面的值：

值(按位或操作)	说明
O_CREAT	如果需要就按参数 mode 给出的访问模式创建文件，见表二
O_TRUNC	把文件长度设置为 0，丢弃已有内容
O_APPEND	把写入数据追加到文件末尾
O_EXCL	与 O_CREAT 一起使用，确保调用者创建文件。open 调用是一个原子操

	作 ,它只执行一个函数调用。使用这个可选模式可以防止两个进程同时创建同一个文件。如果文件已经存在 open 调用失败。

表二：

当你使用 O_CREAT 值时候必须使用三个参数的 open 调用。

mode 包含下面的值：

值	权限	拥有者
S_IRUSR	r	User
S_IWUSR	w	User
S_IXUSR	x	User
S_IRGRP	r	Group
S_IWGRP	w	Group
S_IXGRP	x	Group
S_IROTH	r	Other
S_IWOTH	w	Other
S_IXOTH	x	other

2、write 系统调用

```
size_t write(int fd , const char *buf,size_t nbytes);
```

参数说明：

参数	说明
fd	成功调用 Open 返回的文件描述符
buf	数据来源地的指针
nbytes	要写入的数据大小

返回实际写入的字节数，返回值可能小于 nbytes。如果返回 0，表示未写入数据；如果返回-1，表示 write 调用出错，错误代码保存在全局变量 errno 中。

3、read 系统调用

```
size_t read(int fd , const char *buf,size_t nbytes);
```

参数说明：

参数	说明
fd	成功调用 Open 返回的文件描述符
buf	数据目的地的指针
nbytes	要读取的数据大小

返回实际读入的字节数，可能会小于 nbytes。如果返回 0，表示未读入任何数据，已到达文件尾；返回-1 表示出现错误。

4、close 系统调用

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

close 调用终止文件描述符 fd 与其对应文件之间的关联。文件描述符被释放并能够重新使用。close 调用成功时候返回 0，出错时返回-1。

检查 close 调用的返回结果非常重要。有的文件系统，特别使网络文件系统，可能不会在关闭文件之前报告文件写操作中出现的错误，这是因为在执行写操作时，数据可能未被确认写入

5、lseek 系统调用

```
off_t lseek(int fd, off_t offset, int whence);
```

lseek 系统调用对文件描述符的读写指针位置进行设置，offset 偏移量

whence 取值如下：

SEEK_SET:文件指针从文件起始位置往后偏移 offset 位置

SEEK_CRU:文件指针从当前位置开始往后偏移 offset 位置

SEEK_END:文件指针从文件尾位置开始往前移 offset 位置

3）利用 expat 库解析 XML 配置文件

函数头文件：

```
#include “expat.h”
#include “expat_external.h”
```

动态库：

```
libexpat.so
```

函数说明：

1.XML_Parser **XML_ParserCreate**(const XML_Char *encodingName)；

参数一般为 NULL，函数返回一个 XML_Parser 类型指针，我们就当他是一个句柄

例子：

```
XML_Parser parser = XML_ParserCreate(NULL);
```

2. **XML_SetUserData**(XML_Parser parser, void *p)

设置读取到的数据保存的地址。

例子：

```
typedef struct {
 int depth;
 int element;
 char device_software_version[32];
 int pre_mixer_index;
 char is_variable_scene_number;
} DSP_CONF_XML_INFO_T;

 DSP_CONF_XML_INFO_T g_dsp_conf_xml_info;
```

```
XML_SetUserData(parser, &g_dsp_conf_xml_info);
```

3. XML_SetElementHandler(XML_Parser parser, XML_StartElementHandler start, XML_EndElementHandler end)

绑定回调函数。

Start 和 end，这二个回调分别是对应于解析<>和</>。

例子：

```
void XMLCALL dsp_conf_xml_start_element(void *userData, const char *name, const char **atts)
{
 int i;
 DSP_CONF_XML_INFO_T *info = (DSP_CONF_XML_INFO_T *)userData;

 //printf("name=%s\n", name);
 for (i=0; atts[i]; i+=2) {
     //printf("%s=%s\n", atts[i], atts[i+1]);
     info->element = get_element_id(atts[i]);
     set_element_value(info, atts[i+1]);
 }

 info->element = get_element_id(name);
 info->depth++;
}

static void XMLCALL dsp_conf_xml_end_element(void *userData, const char *name)
{
 DSP_CONF_XML_INFO_T *info = (DSP_CONF_XML_INFO_T *)userData;
 info->depth--;
 info->element = EM_UNUSE;
}

XML_SetElementHandler(parser, dsp_conf_xml_start_element, dsp_conf_xml_end_element);
```

4. XML_SetCharacterDataHandler(XML_Parser parser,XML_CharacterDataHandler handler)

这个函数是设置处理一个<>和</>之间的字段的回调。

例子：

```
 void  XMLCALL  dsp_conf_xml_data_element(void  *userData,  const
XML_Char *s, int len)
{
 DSP_CONF_XML_INFO_T *info = (DSP_CONF_XML_INFO_T *)userData;
 char *value = (char*)malloc(len+1);
 if (value) {
     memcpy(value, s, len);
     value[len] = '\0';
     //printf("value=%s\n", value);
     set_element_value(info, value);
     free(value);
 }
}

XML_SetCharacterDataHandler(parser, dsp_conf_xml_data_element);
```

5.**XML_Parse**(XML_Parser parser, const char *s, int len, int isFin)

调用该函数后才开始解析 XML 文件。

第二个参数是用户指定的 Buffer 指针， 第三个是这块 Buffer 中实际内容的字节数，最后参数代表是否这块 Buffer 已经结束。比如要解析的 XML 文件太大，但内存比较吃紧，Buffer 比较小，则可以循环读取文件，然后丢给 Parser， 在文件读取结束前，isFin 参数为 FALSE，反之为 TRUE。

例子：

```
    int fd;
    struct stat st;
    char *buf;
    int len;
    int done = 0;
    fd = open(xml_file_path, O_RDONLY);
    if (fd < 0) {
        perror("open");
        return -1;
    }

    fstat(fd, &st);//通过文件描述符获取文件属性

    len = st.st_size;
```

```
        buf = (char*)malloc(len);
        if (!buf) {
            perror("malloc");
            return -1;
        }
        read(fd, buf, len);
        close(fd);

    if (XML_Parse(parser, buf, len, done) == XML_STATUS_ERROR) {
        fprintf(stderr,
            "%s at line %" XML_FMT_INT_MOD "u\n",
            XML_ErrorString(XML_GetErrorCode(parser)),
            XML_GetCurrentLineNumber(parser));
        return -1;
     }
```

6. XML_ParserFree(XML_Parser parser);

结束后释放申请的内存。

例子：

```
XML_ParserFree(parser);
```

详细例子请看：

expat/example/ xml_parser.c

4）目录操作函数

头文件：

```
#include <sys/types.h>
#include <dirent.h>
```

函数原型：

1．DIR *opendir(const char *name);

opendir 函数打开一个与给定的目录名 name 相对应的目录流，并返回一个指向该

目录流的指针。打开后，该目录流指向了目录中的第一个目录项。

opendir 函数，打开成功，返回指向目录流的指针；打开失败，则返回 NULL，并设置相应的错误代码 errno。

2．struct dirent *readdir(DIR *dir);

readdir 函数返回一个指向 dirent 结构体的指针，该结构体代表了由 dir 指向的目录流中的下一个目录项；如果读到 end-of-file 或者出现了错误，那么返回 NULL。

readdir 函数，成功时返回一个指向 dirent 结构体的指针；失败时或读到 end-of-file 时，返回 NULL，并且设置相应的错误代码 errno。

在 Linux 系统中，dirent 结构体定义如下：

```
struct dirent {
      ino_t          d_ino;       /* inode number */
      off_t          d_off;        /*  offset  to  the  next
dirent */
      unsigned short  d_reclen;    /* length of this record */
      unsigned char   d_type;      /* type of file */
      char                d_name[256]; /* filename */
  };
```

d_type 表示档案类型：

```
 enum
{
   DT_UNKNOWN = 0,
   DT_FIFO,
   DT_CHR,
   DT_DIR = 4,
   DT_BLK = 6,
   DT_REG = 8,
   DT_LNK = 10,
   DT_SOCK = 12,
   DT_WHT = 14};
```

readdir 函数返回的值会被后续调用的(针对同一目录流的)readdir 函数返回值所覆盖。

3．int closedir(DIR *dir);

closedir 函数关闭与指针 dir 相联系的目录流。关闭后，目录流描述符 dir 不再可用。

closedir 函数，成功时返回 0；失败是返回-1，并设置相应的错误代码 errno。

4．int chdir(const char *path);

chdir 函数改变当前工作目录为 path 指定的目录。

成功，返回 0；失败，返回-1，并设置相应的错误代码 errno。

5． int fchdir(int fd);

fchdir 函数与 chdir 功能一样，唯一的区别是 fchdir 所要改变成的工作目录由打开的文件描述符指定。

成功，返回 0；失败，返回-1，并设置相应的错误代码 errno。

例子说明：

1、Linux C 下遍历目录及其文件：

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <dirent.h>
#include <sys/stat.h>
#include <unistd.h>
#include <sys/types.h>
#include <iostream>
using namespace std;
```

```
//main 函数的 argv[1] char * 作为 所需要遍历的路径 传参数给 listDir
void listDir(char *path)
{
  DIR             *pDir ;  //定义一个 DIR 类的指针
  struct dirent    *ent  ;//定义一个结构体 dirent 的指针，dirent 结构体见上
  int              i=0  ;
  char             childpath[512];//定义一个字符数组，用来存放读取的路径

  pDir=opendir(path); //  opendir 方法打开 path 目录，并将地址付给 pDir 指针
  memset(childpath,0,sizeof(childpath)); //将字符数组 childpath 的数组元素
全部置零

   //读取 pDir 打开的目录，并赋值给 ent，同时判断是否目录为空，不为空则执行循环
体
   while((ent=readdir(pDir))!=NULL)
    {
   //读取 打开目录的文件类型 并与 DT_DIR 进行位与运算操作，即如果读取的 d_type
类型为 DT_DIR (=4 表示读取的为目录)
       if(ent->d_type & DT_DIR)
       {
//如果读取的 d_name 为 . 或者.. 表示读取的是当前目录符和上一目录符，用 contiue
跳过，不进行下面的输出
        if(strcmp(ent->d_name,".")==0 || strcmp(ent->d_name,"..")==0)
                          continue;
  //如果非. ..则将 路径 和 文件名 d_name 付给 childpath，并在下一行 prinf 输出
        sprintf(childpath,"%s/%s",path,ent->d_name);
        printf("path:%s\n",childpath);
//递归读取下层的字目录内容， 因为是递归，所以从外往里逐次输出所有目录（路径+目
录名），然后才在 else 中由内往外逐次输出所有文件名
```

```
        listDir(childpath);

      }
//如果读取的 d_type 类型不是 DT_DIR，即读取的不是目录，而是文件，则直接输出 d_
name，即输出文件名
     else
     {
//cout<<childpath<<"/"<<ent->d_name<<endl;  输出文件名 带上了目录

        cout<<ent->d_name<<endl;
     }
   }

}

int main(int argc,char *argv[])
{
//第一个参数为 想要遍历的 linux 目录 例如，当前目录为 ./ ,上一层目录为../
        listDir(argv[1]);
        return 0;
}
```

当前目录下有：

```
yzh@ubuntu:~/yzh$ ls -l
total 10236
-rw-rw-r-- 1 yzh yzh       0 Mar 24  2017 0
-rw-rw-r-- 1 yzh yzh       0 Mar 24  2017 1
-rw-rw-r-- 1 yzh yzh       0 Mar 24  2017 2
drwxr-xr-x 4 yzh yzh    4096 Sep 13  2016 etc
-rw-rw-r-- 1 yzh yzh  747314 Jun 13 04:38 jiekouji.tar.gz
drwxr-xr-x 2 yzh yzh    4096 May 18  2016 lib
-rwxrwxr-x 1 yzh yzh   13704 Nov 22 18:04 list
drwxrwxr-x 2 yzh yzh    4096 Nov 16 18:59 optarg
-rw-rw-r-- 1 yzh yzh 3518643 Jun 11 20:41 rtl8188eus_v2.tar.gz
-rwx--x--x 1 yzh yzh    7405 Mar 24  2017 test.sh
drwxr-xr-x 3 yzh yzh    4096 May 25  2016 usr
-rw-rw-r-- 1 yzh yzh 6164480 May 19  2017 web_var_scenes_20170515.tar
-rwxrwxrwx 1 yzh yzh     267 Jun 13 23:08 wifi_check.sh
yzh@ubuntu:~/yzh$
```

执行结果：

```
yzh@ubuntu:~/yzh$ ./list ./
path:.//optarg
test.c
test
2
1
path:.//usr
path:.//usr/sbin
httpd
test.sh
web_var_scenes_20170515.tar
jiekouji.tar.gz
list
path:.//lib
davdsp.ko
wifi_check.sh
rtl8188eus_v2.tar.gz
path:.//etc
path:.//etc/soundtrack
path:.//etc/soundtrack/def
dsp_conf.xml
path:.//etc/boa
boa.conf
0
yzh@ubuntu:~/yzh$
```

2、获取文件中的一行并转化为 16 进制数据

```
#include <sys/types.h>
#include <dirent.h>
#include <sys/stat.h>
#include <stdio.h>
/*
    参数说明：

        P: 读取到的源数据地址

        P1:转换后要保存目的地址

        Len:转换的数据长度
*/
void hex2bins(const char *p, char *p1, int len)
{
    int i=0;
    while(*p && i<len*2) {
        if(*p>='0' && *p<='9') {
            if(i%2) {
                p1[i/2] += *p-'0';
            }
            else {
                p1[i/2] = (*p-'0')<<4;
            }
            i++;
        }
```

```
        else if(*p>='a' && *p<='f') {
            if(i%2) {
                p1[i/2] += *p-'a'+10;
            }
            else {
                p1[i/2] = (*p-'a'+10)<<4;
            }
            i++;
        }
        else if(*p>='A' && *p<='F') {
            if(i%2) {
                p1[i/2] += *p-'A'+10;
            }
            else {
                p1[i/2] = (*p-'A'+10)<<4;
            }
            i++;
        }
        p++;
    }
}
```

void func(int gpio,int on)

```
{
    int i;
    int ret = -1;
    FILE *fp = NULL;
    char line[64];
    unsigned char cmd_buf[8];
    int is_match = 0;
    int gpio_num;
    int level;
    DIR *dp;


    if (dp = opendir(“abc”)) { //检查是否有该目录
        closedir(dp);
    } else {
        mkdir(“abc”, 0755); //创建目录
    }

    fp = fopen(“abc/123.txt”, "r");
    if (!fp) {
```

```
        return;
    }
    while (fgets(line, sizeof(line), fp)) {
        printf("%s", line);
        if (sscanf(line, "#GP_%x_LV_%d", &gpio_num, &level) == 2) {
            if (gpio_num == gpio && level == on) {
                is_match = 1;
            } else {
                is_match = 0;
            }
        } else {
            if (is_match) {
                hex2bins(line, cmd_buf, 8);
                printf("-->%02X %02X %02X %02X %02X %02X %02X %02X\n",
                    cmd_buf[0], cmd_buf[1], cmd_buf[2], cmd_buf[3],
                    cmd_buf[4], cmd_buf[5], cmd_buf[6], cmd_buf[7]);

            }
        }

    }
    fclose(fp);
}
int main(int argc,char *argv[])
{
    func(0x51,1);
    return 0;
}
```

Abc/123.txt 内容如下：

```
#GP_0x51_LV_1
A5AB04360101003C
```

打印出来数据如下：

```
#GP_0x51_LV_1
A5AB04360101003C
-->A5 AB 04 36 01 01 00 3C
```

3、网络 socket

头文件:

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

函数说明：

1）socket 函数原型

socket(建立一个 socket 文件描述符)		
所需头文件	#include <sys/types.h> #include <sys/socket.h>	
函数说明	建立一个 socket 文件描述符	
函数原型	int socket(int domain, int type, int protocol)	
函数传入值	domain	AF_INET：IPv4 协议
		AF_INET6：IPv6 协议
		AF_LOCAL：Unix 域协议
		AF_ROUTE：路由套接口
		AF_KEY：密钥套接口
	type	SOCKET_STREAM：双向可靠数据流，对应 TCP
		SOCKET_DGRAM：双向不可靠数据报，对应 UDP
		SOCKET_RAW：提供传输层以下的协议，可以访问内部网络接口，例如接收和发送 ICMP 报文
	protocol	type 为 SOCKET_RAW 时需要设置此值说明协议类型，其他类型设置为 0 即可
函数返回值	成功：socket 文件描述符	
	失败：-1，失败原因存于 error 中	

表 18-1 列出了当进行 socket 调用时，中协议簇（domain）与类型(type)可能产生的组合。

表 18-1 socket 中协议簇（domain）与类型(type)组合表

	AF_INET	AF_INET6	AF_LOCAL	AF_ROUTE	AF_KEY
SOCK_STREAM	TCP	TCP	Yes		

SOCK_DGRAM	UDP	UDP	Yes		
SOCK_RAW	IPv4	IPv6		Yes	Yes

2）bind 函数原型

bind(将一个本地协议地址与 socket 文件描述符联系起来)		
所需头文件	#include <sys/types.h> #include <sys/socket.h>	
函数说明	将一个协议地址与 socket 文件描述符联系起来	
函数原型	int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen)	
函数传入值	sockfd	socket 文件描述符
	addr	my_addr 指向 sockaddr 结构，该结构中包含 IP 地址和端口等信息，见**表 18-3**
	addrlen	sockaddr 结构的大小，可设置为 sizeof(struct sockaddr)
函数返回值	成功：0	
	失败：-1，失败原因存于 error 中	

利用 bind 函数绑定地址时，可以指定 IP 地址和端口号，也可以指定其中之一，甚至一个也不指定。可以使用通配地址 INADDR_ANY（为宏定义，其值等于 0），它通知内核选择 IP 地址。**表 18-2** 列出了设置 socket 地址结构的几种方式，但在实际中，绑定的端口号都需要指定。

表 18-2 设置 socket 地址结构的几种方式

进程指定		说 明
IP 地址	端口	
通配地址 INADDR_ANY	0	内核自动选择 IP 地址和端口号
通配地址 INADDR_ANY	非 0	内核自动选择 IP 地址，进程指定端口号
本地 IP 地址	0	进程指定 IP 地址，内核自动选择端口号
本地 IP 地址	非 0	进程指定 IP 地址和端口号

表 18-3 参数 addr 说明

结构体原型	typedef unsigned short sa_family_t;
	struct sockaddr {
	    sa_family_t sa_family; /* address family, AF_xxx*/
	    char sa_data[14]; /* 14 bytes of protocol address */
	};

Ipv4	struct sockaddr_in {

	sa_family_t sin_family; /* address family: AF_INET */
	in_port_t sin_port; /* port in network byte order */
	struct in_addr sin_addr; /* internet address */
	char sin_zero[8];
	};
	
	/* Internet address. */
	struct in_addr {
	uint32_t s_addr; /* address in network byte order
	*/
	};

Ipv6	struct sockaddr_in6 {
	sa_family_t sin6_family; /* AF_INET6 */
	in_port_t sin6_port; /* port number */
	uint32_t sin6_flowinfo; /* IPv6 flow information */
	struct in6_addr sin6_addr; /* IPv6 address */
	uint32_t sin6_scope_id; /* Scope ID (new in 2.4) */
	};
	
	struct in6_addr {
	unsigned char s6_addr[16]; /* IPv6 address */
	};

Unix 域	#define UNIX_PATH_MAX 108
	
	struct sockaddr_un {
	sa_family_t sun_family; /* AF_UNIX */
	char sun_path[UNIX_PATH_MAX]; /* pathname */
	};

例子:

```
struct sockaddr_in saddr;
memset((void *)&saddr,0,sizeof(saddr));
saddr.sin_family = AF_INET;

saddr.sin_port=htons(8888);//绑定端口

saddr.sin_addr.s_addr = htonl(INADDR_ANY);
//saddr.sin_addr.s_addr = inet_addr("192.168.22.5");

//绑定固定 IP

bind(ListenSocket,(struct sockaddr *)&saddr,sizeof(saddr));
```

3）listen 函数原型

listen (等待连接)		
所需头文件	#include <sys/types.h> #include <sys/socket.h>	
函数说明	等待连接	
函数原型	int listen(int sockfd, int backlog)	
函数传入值	sockfd	监听 socket 文件描述符
	backlog	套接字排队的最大连接个数
函数返回值	成功：0	
	失败：-1，失败原因存于 error 中	
特别说明	对于监听 socket 文件描述符 sockfd，内核要维护两个队列，分别为未完成连接队列和已完成连接队列，这两个队列之和不超过 backlog	

4）connect 函数原型

connect(建立 socket 连接)		
所需头文件	#include <sys/types.h> #include <sys/socket.h>	
函数说明	建立 socket 连接	
函数原型	int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen)	
函数传入值	sockfd	socket 文件描述符
	serv_addr	连接的网络地址和端口，要去连接的 IP 和端口，具体格式见**表 18-3**
	addrlen	sockaddr 结构的大小，可设置为 sizeof(struct sockaddr)
函数返回值	成功：0	
	失败：-1，失败原因存于 error 中	
附加说明	函数 connect 激发 TCP 的三路握手过程，出错返回有以下几种情况： ① 如果客户没有收到 SYN 分节的响应（总共 75 秒，这之间可能重发了若干次 SYN），则返回 ETIMEDOUT ② 如果对客户的 SYN 的响应是 RST，则表明该服务器主机在指定的端口上没有进程在等待与之相连，函数返回错误 ECONNREFUSED	

	③ 如果客户发出的 SYN 在中间路由器上引发一个目的地不可达的 ICMP 错误，内核返回 EHOSTUNREACH 或 ENETUNREACH 错误（即 ICMP 错误）给进程

5）accept 函数原型

所需头文件	#include <sys/types.h> #include <sys/socket.h>
函数说明	接受 socket 连接，返回一个新的 socket 文件描述符，原 socket 文件描述符仍为 listen 函数所用，而新的 socket 文件描述符用来处理连接的读写操作
函数原型	int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen)
函数传入参数	sockfd：socket 文件描述符
	addrlen：addr 的大小，可设置为 sizeof(struct sockaddr)
函数传出参数	addr：填入远程主机的地址数据，得到请求连接方的 IP 和端口，具体格式见表 18-3
函数返回值	成功：实际读取字节数
	失败：-1，错误代码存放在 error 中
附加说明	① accept 函数由 TCP 服务器调用，为阻塞函数，从已完成连接的队列中返回一个连接；如果该对列为空，则进程进入阻塞等待 ② 函数返回的套接字为已连接套接字，而监听套接字仍为 listen 函数所用

6）close 函数原型

close（关闭连接的 socket 文件描述符）	
所需头文件	#include <unistd.h>
函数说明	关闭连接的 socket 文件描述符
函数原型	int close(int sockfd)
函数传入值	sockfd：socket 文件描述符
函数返回值	成功：0
	失败：-1，失败原因存于 error 中
附加说明	① close 函数默认功能是将套接字置为“已关闭”标记，并立即返回给进程，这个套接字不能再为该进程所用 ② 正常情况下，close 将引发四个分节终止序列，但在终止前将发送已排队的数据 ③ 如果套接字描述符访问计数在调用 close 后大于 0（多个进程共享同一个套接字的情况下），则不会引发 TCP 终止序列（即不会发送 FIN 分节）

7）shutdown 函数原型

shutdown(终止 socket 通信)		
所需头文件	#include <sys/socket.h>	
函数说明	终止 socket 通信	
函数原型	int shutdown(int s, int how)	
函数传入值	s	socket 文件描述符
	how	0（SHUT_RD）：关闭 socket 连接的读这一半，不再接收套接字中的数据且现留在收缓冲区的数据作废
		1（SHUT_WR）：关闭 socket 连接的写这一半（半关闭），但留在套接字发送缓冲区中的数据都会被发送，后跟 TCP 连接终止序列，不管访问计数是否大于 0，此后将不能再执行对套接字的任何写操作
		2（SHUT_RDWR）：socket 连接的读、写都关闭
函数返回值	成功：0	
	失败：-1，失败原因存于 error 中	

8）read 函数原型

所需头文件	#include <unistd.h>
函数说明	从打开的 socket 文件流中读取数据，这里仅说明此函数应用于 socket 的情况
函数原型	ssize_t read(int fd, void *buf ,size_t count)
函数传入参数	fd：socket 文件描述符
	count：最大读取字节数
函数传出参数	buf：读取数据的首地址
函数返回值	成功：实际读取字节数
	失败：-1，错误代码存放在 error 中
附加说明	调用函数 read 从 socket 文件流中读取数据时，有如下几种情况： ① 套接字接收缓冲区接收数据，返回接收到的字节数 ② TCP 协议收到 FIN 数据，返回 0 ③ TCP 协议收到 RST 数据，返回-1，同时 errno 设置为 ECONNRESET ④ 进程阻塞过程中接收到信号，返回-1，同时 errno 设置为 EINTR

9）write 函数原型

所需头文件	#include <unistd.h>
函数说明	向 socket 文件流中写入数据，这里仅说明此函数应用于 socket 的情况
函数原型	ssize_t write (int fd,const void *buf,size_t count)
函数传入参数	fd：socket 文件描述符
	buf：写入数据的首地址
	count：最大写入字节数
函数返回值	成功：实际写入的字节数
	失败：-1，错误代码存放在 error 中
附加说明	调用函数 write 向 socket 文件流写数据时，有如下几种情况： ① 套接字发送缓冲区有足够空间，返回发送的字节数 ② TCP 协议接收到 RST 数据，返回-1，同时 errno 设置为 ECONNRESET ③ 进程阻塞过程中接收到信号，返回-1，同时 errno 设置为 EINTR

10）send 函数原型

send(通过 socket 文件描述符发送数据到对方)		
所需头文件	#include <sys/types.h> #include <sys/socket.h>	
函数说明	通过 socket 文件描述符发送数据到对方	
函数原型	ssize_t send(int s, const void *buf, size_t len, int flags)	
函数传入值	s	socket 文件描述符
	buf	发送数据的首地址
	len	发送数据的长度
	flags	0：此时功能同 write，flags 还可以设为以下标志的组合
		MSG_OOB：发送带外数据
		MSG_DONTROUTE：告诉 IP 协议，目的主机在本地网络，没有必要查找路由表
		MSG_DONTWAIT：设置为非阻塞操作
		MSG_NOSIGNAL：表示发送动作不愿被 SIGPIPE 信号中断
函数返回值	成功：实际发送的字节数	
	失败：-1，失败原因存于 error 中	

11）recv 函数原型

recv(通过 socket 文件描述符从对方接收数据)

所需头文件	#include <sys/types.h> #include <sys/socket.h>	
函数说明	通过 socket 文件描述符从对方接收数据	
函数原型	ssize_t recv(int s, void *buf, size_t len, int flags)	
函数传入值	s	socket 文件描述符
	len	可接收数据的最大长度
	flags	0：此时功能同 read，flags 还可以设为以下标志的组合
		MSG_OOB：接收带外数据
		MSG_PEEK：查看数据标志，返回的数据并不在系统中删除，如果再次调用 recv 函数会返回相同的数据内容
		MSG_DONTWAIT：设置为非阻塞操作
		MSG_WAITALL：强迫接收到 len 大小的数据后才返回，除非有错误或有信号产生
函数传出值	buf	接收数据的首地址
函数返回值	成功：实际发送的字节数	
	失败：-1，失败原因存于 error 中	

12）sendto 函数原型

sendto(通过 socket 文件描述符发送数据到对方)		
所需头文件	#include <sys/types.h> #include <sys/socket.h>	
函数说明	通过 socket 文件描述符发送数据到对方，用于 UDP 协议	
函数原型	ssize_t sendto(int s, const void *buf, size_t len, int flags, const struct sockaddr *to, socklen_t tolen)	
函数传入值	s	socket 文件描述符
	buf	发送数据的首地址
	len	发送数据的长度
	flags	0：默认方式发送数据，flags 还可以设为以下标志的组合
		MSG_OOB：发送带外数据
		MSG_DONTROUTE：告诉 IP 协议，目的主机在本地网络，没有必要查找路由表
		MSG_DONTWAIT：设置为非阻塞操作
		MSG_NOSIGNAL：表示发送动作不愿被 SIGPIPE 信号中断
	to	存放目的主机 IP 地址和端口信息，具体格式见**表 18-3**
	tolen	to 的长度，可设置为 sizeof(struct sockaddr)
	成功：实际发送的字节数	

函数返回值	失败：-1，失败原因存于 error 中

13）recvfrom 函数原型

recv(通过 socket 文件描述符从对方接收数据)		
所需头文件	#include <sys/types.h> #include <sys/socket.h>	
函数说明	通过 socket 文件描述符从对方接收数据，用于 UDP 协议	
函数原型	ssize_t recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen)	
函数传入值	s	socket 文件描述符
	len	可接收数据的最大长度
	flags	0：默认方式接收数据，flags 还可以设为以下标志的组合
		MSG_OOB：接收带外数据
		MSG_PEEK：查看数据标志，返回的数据并不在系统中删除，如果再次调用 recv 函数会返回相同的数据内容
		MSG_DONTWAIT：设置为非阻塞操作
		MSG_WAITALL：强迫接收到 len 大小的数据后才返回，除非有错误或有信号产生
	fromlen	from 的长度，可设置为 sizeof(struct sockaddr)
函数传出值	buf	接收数据的首地址
	from	存放发送方的 IP 地址和端口，具体格式见表 18-3
函数返回值	成功：实际发送的字节数	
	失败：-1，失败原因存于 error 中	

14）套接字属性控制函数

系统提供 getsockopt、setsockopt 两函数获取和修改套接字结构中一些属性，通过修改这些属性，可以调整套接字的性能，进而调整应用程序的性能。

（1）getsockopt 函数原型

getsockopt(获取套接字的属性)	
所需头文件	#include <sys/types.h>

	#include <sys/socket.h>	
函数说明	获取套接字的属性	
函数原型	int getsockopt(int s, int level, int optname, void *optval, socklen_t *optlen)	
函数传入值	s	socket 文件描述符
	level	SOL_SOCKET：通用套接字选项
		IPPROTO_IP：IP 选项
		IPPROTO_TCP：TCP 选项
	optname	访问的选项名，具体见表 18-4
	optlen	optval 的长度
函数传出值	optval	取得的属性值
函数返回值	成功：0	
	失败：-1，失败原因存于 error 中	

（2）setsockopt 函数原型

setsockopt(设置套接字的属性)		
所需头文件	#include <sys/types.h> #include <sys/socket.h>	
函数说明	设置套接字的属性	
函数原型	int setsockopt(int s, int level, int optname, const void *optval, socklen_t optlen)	
函数传入值	s	socket 文件描述符
	level	SOL_SOCKET：通用套接字选项
		IPPROTO_IP：IP 选项
		IPPROTO_TCP：TCP 选项
	optname	设置的选项名，具体见表 18-4
	optval	设置的属性值
	optlen	optval 的长度
函数返回值	成功：0	
	失败：-1，失败原因存于 error 中	

表 18-4 套接字属性表

level(级别)	optname(选项名	说明	数据类型
SOL_SOCKET	SO_BROADCAST	允许发送广播数据	int
	SO_DEBUG	允许调试	int
	SO_DONTROUTE	不查找路由	int
	SO_ERROR	获得套接字错误	int
	SO_KEEPALIVE	保持连接	int
	SO_LINGER	延迟关闭连接	struct linger

	SO_OOBINLINE	带外数据放入正常数据流	int
	SO_RCVBUF	接收缓冲区大小	int
	SO_SNDBUF	发送缓冲区大小	int
	SO_RCVLOWAT	接收缓冲区下限	int
	SO_SNDLOWAT	发送缓冲区下限	int
	SO_RCVTIMEO	接收超时	struct timeval
	SO_SNDTIMEO	发送超时	struct timeval
	SO_REUSERADDR	允许重用本地地址和端口	int
	SO_TYPE	获得套接字类型	int
	SO_BSDCOMPAT	与 BSD 系统兼容	int
IPPROTO_IP	IP_HDRINCL	在数据包中包含 IP 首部	int
	IP_OPTINOS	IP 首部选项	int
	IP_TOS	服务类型	int
	IP_TTL	生存时间	int
IPPRO_TCP	TCP_MAXSEG	TCP 最大数据段的大小	int
	CP_NODELAY	不使用 Nagle 算法	int

（3）getsockopt\setsockopt 举例

sockopt.c 源代码如下：

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
int main()
{
    int sockfd,optval,optlen = sizeof(int);
    int sndbuf = 0 ;
    int rcvbuf = 0 ;
    int flag;
    if((sockfd = socket(AF_INET,SOCK_STREAM,0))<0)
    {
        perror("socket") ;
        return -1 ;
    }
    getsockopt(sockfd,SOL_SOCKET,SO_TYPE,&optval,&optlen);
    printf("optval = %d\n",optval);
    optlen = sizeof(sndbuf);
    flag = getsockopt(sockfd,SOL_SOCKET,SO_SNDBUF,&sndbuf,&optlen);
    printf("sndbuf=%d\n",sndbuf) ;
    printf("flag=%d\n",flag) ;
    sndbuf = 51200;
    flag = setsockopt(sockfd,SOL_SOCKET,SO_SNDBUF,&sndbuf, optlen);
    sndbuf=0 ;
```

```
    flag = getsockopt(sockfd,SOL_SOCKET,SO_SNDBUF,&sndbuf,&optlen);
    printf("sndbuf=%d\n",sndbuf) ;
    printf("flag=%d\n",flag) ;
    close(sockfd);
    return 0 ;
}
```

编译 gcc sockopt.c -o sockopt。

执行 ./sockopt， 执行结果如下：

```
optval = 1
sndbuf=16384
flag=0
sndbuf=102400
flag=0
```

4、fcntl 系统调用，根据文件描述词来操作文件的特性

头文件：

```
#include <unistd.h>
#include <fcntl.h>
```

函数说明：

```
int fcntl(int fd,int cmd);
int fcntl(int fd,int cmd,long arg);
int fcntl(int fd,int cmd,struct flock *lock);
```

使用方法：

1）修改文件、句柄为阻塞非阻塞

设置为非阻塞

```
int flags = fcntl(fd, F_GETFL 0);
flags |= O_NONBLOCK;
fcntl(fd,F_SETFL,flags);
```

设置为阻塞：

```
int flags = fcntl(fd, F_GETFL 0);
flags &= ~O_NONBLOCK;
fcntl(fd,F_SETFL,flags);
```

2）其他见链接

点我，点我。