第 5 章 文件 I/O 编程

CHAPTER 5

学习 Linux 环境高级编程，首先学习的是文件的操作。有一句很有趣的话“Linux 下一切皆文件”，所以掌握了文件操作的方法，也就算摸到了 Linux 编程的门路。

5.1 文件和目录

首先直观地感受在终端下输入命令 ls -l 的显示结果，部分程序显示如下：

```
1.  root@unbuntu-virtual-machine:/# ls -l
2.  总用量 1943004
3.  drwxr-xr-x    2 root root        4096 3月  10 22:37 bin
4.  drwxr-xr-x    3 root root        4096 3月  10 22:38 boot
5.  drwxrwxr-x    2 root root        4096 3月  10 22:22 cdrom
6.  drwxr-xr-x   18 root root        4100 3月  20 10:57 dev
7.  drwxr-xr-x 124 root root       12288 3月  20 13:41 etc
8.  drwxr-xr-x    5 root root        4096 3月  20 13:23 home
9.  drwxr-xr-x   21 root root        4096 3月  10 22:37 lib
10. drwxr-xr-x    2 root root        4096 3月  20 13:41 lib64
11. drwx------    2 root root       16384 3月  10 22:21 lost+found
12. drwxr-xr-x    3 root root        4096 3月  20 10:47 media
13. drwxr-xr-x    3 root root        4096 3月  20 10:57 mnt
```

1．drwxr-xr-x

drwxr-xr-x 代表的是文件类型和文件权限。常用的文件类型如下：

（1）-：普通文件，存储各种数据。

（2）d：目录文件，存储结构体，结构体内部标识这个目录中的文件等信息。

（3）l：链接文件，需要注意的是，软链接才是文件，而硬链接仅是一个节点。

（4）c：字符设备，除了块设备都是字符设备，没有扇区的概念。

（5）b：块设备，所有存储类的驱动都称为块设备，包含扇区处理。

（6）p：管道设备，是用内核内存模拟的通道。

从以上说明可以看出，上述程序中的文件是一个目录文件，原因是 drwxr-xr-x 代表文件类型，d 代表此文件是一个目录文件。

常用的文件权限如下：

（1）r 为读，二进制权重为 100 即 4。指用户对文件或目录具有查看权限。若用户对某文件或目录不具有读权限，则不能查看其内容。

（2）w 为写，二进制权重为 010 即 2。指用户对文件或目录具有写权限。若用户对某文件或目录没有写权限，则不能修改它。

（3）x 为执行，二进制权重为 001 即 1。指用户对文件具有执行权限或对目录具有进入权限。若用户对某文件没有执行权限，则不能执行它；若用户对某目录没有执行权限，则不能进入它。

（4）-为无操作，二进制权重为 0。指用户对文件或目录不赋予某种权限。-wx 表示不能读，但可写可执行。

（5）rwx 顺序不可改，表示可读可写可执行。

上述就是文件权限的表示方法，文件权限是用八进制来表达的，如果一个文件有全部的权限，那么对应八进制里的数是 7（4+2+1）。同时读者会发现有多组 rwx，它所表达的不仅是它自身的权限。这里涉及一个分组的概念，分别是创建者（user，u）、所在组（group，g）和其他人（other，o）。

（1）u 组：创建者（user）。

（2）g 组：创建者所在组的成员（group）。

（3）o 组：其他人所具备的权限（other）。

也就是说，上述程序中的三组 rwx 都是依照上述顺序来说明权限的。上述程序中的文件权限就是：创建者可读可写可执行，所在组的成员可读不可写可执行，其他成员可读不可写可执行。

2．文件节点数

上述程序中文件类型和权限之后是数字 2，这个 2 表示的是文件节点数，也就是说，此文件是一个目录文件。所以，目录的节点数代表该目录下的文件个数，在这里应该是有两个文件。如果此文件不是目录，只是普通文件，那么这个数字就代表硬链接的个数。关于链接的几点说明如下：

（1）链接分为硬链接和软链接（符号链接，即快捷方式）。

（2）硬链接，只是增加一个引用计数，本质上并没有物理上的增加文件。硬链接不是文件。

（3）软链接，是在磁盘上产生一个文件，这个文件内部写入了一个指向被链接的文件的指针。

（4）采用 ln 指令，用来在文件之间创建链接，默认为创建硬链接（目录不能创建硬链接），使用选项-s 创建符号链接。硬链接指向文件本身，符号链接指向文件名称。

（5）Linux 系统中寻找文件的顺序是：根据文件名，找到 inode 编号，根据编号找到 inode 块，然后根据 inode 块中的属性信息找到数据块（即文件内容）。

（6）符号链接、硬链接、Windows 快捷方式都具有指向功能，但它们的区别也很明显：Windows 快捷方式指向文件的位置，符号链接是一种文件，创建链接时，系统会为符号链接重新分配一个 inode（节点）编号，但硬链接根本不是一种文件，只是一种指向。

（7）创建硬链接只是增加一个引用计数，硬链接和它的源文件共享一个 inode。

例如：

```
ln file0 file1    //为文件 file0 创建硬链接 file1
ln -s file1 file2   //为文件 file2 创建软链接,其中 file1 为刚创建的硬链接(即 file0 本身)
```

3．目录文件

工作目录是进入系统后所在的当前目录。“.”表示当前目录（工作目录），“..”表示上一级目录（父目录）。

用户目录是每创建一个用户时，就会分配的一个目录，用户名对应的目录就是用户目录，每个用户都有一个自己的主目录，主目录用“~”表示。

路径是从树型目录的某个目录层次到某个文件的一种通路。例如：

```
../../mnt/hgfs/project/linux
```

路径分为如下两种：

（1）相对路径：在工作目录下找到的一个文件路径（通路），随工作目录变化而变化。

（2）绝对路径：从根目录开始，只有一条路径。

目录是一种特殊的文件，在该文件中存放了多个结构体数据，用来代表目录内的子目录、文件等信息。其结构如下：

```
struct direct{
       ino_t  d_ino;                  //目录文件的节点编号
       off_t  d_off;                  //目录文件开始到目录进入点的位移
       unsigned short d_reclen;       //d_name 的长度(字符串长度)
       unsigned char  d_type;         //d_name 的类型
       char   d_name[256];            //文件或目录名
};
```

在前面的学习中介绍过，C 语言本身有自己的函数库，如果需要实现某个功能，包含头文件后直接调用就好。那么在操作系统中，依然会给用户提供一些功能接口 API。用户要实现某些功能必须要依赖这些 API 以及一些机制。

5.2　目录操作


微课视频

1．创建目录

表头文件：

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

定义函数：

```
int mkdir(const char *path, mode_t mode);
```

函数说明：

path：目录名。

mode：模式，即访问权限，包含如下选项：

（1）S_IRUSR：属主读权限。

（2）S_IWUSR：属主写权限。

（3）S_IXUSR：属主执行权限。

（4）S_IRGRP：属组读权限。
（5）S_IWGRP：属组写权限。
（6）S_IXGRP：属组执行权限。
（7）S_IROTH：其他用户读权限。
（8）S_IWOTH：其他用户写权限。
（9）S_IXOTH：其他用户执行权限。

可以使用 S_IRUSR | S_IWUSR 组合权限。可以直接使用数字，例如八进制数 0777、0666 等。例如：

```
int err=mkdir("./aaa",0777);   //在当前目录下创建一个 aaa 目录。注意 Linux 建立的目录的权限默认是 755，若实现 777，需要在终端下输入命令 umask 0 解除。
```

返回值：如果返回 0，则表示成功；如果返回-1，则表示失败。如果创建成功，则在创建的目录自动创建两个子目录“.”和“..”。

示例 5.2-1 创建目录。

```
1.  #include <stdio.h>
2.  #include <sys/stat.h>
3.  #include <errno.h>
4.  #include <string.h>
5.  int main(int argc,char **argv){
6.  int err=mkdir(argv[1],0666); //0666
7.  if (err==-1){
8.  printf("----- mkdir err no=%d,str=%s\n",errno,strerror(errno));
9.  }
10. return 0;
11. }
```

第 8 行代码中 errno 错误号是系统全局变量，运行函数时，系统将运行的错误号写入 errno 中，strerror()将错误号对应的说明取出。

Linux 错误处理，采用一个全局变量 errno，用来存储函数执行后的错误编码。每个错误编码号都对应了一个解释，即错误提示内容。获取错误提示使用如下函数：

```
#include <errno.h>
char *strerror(int errno); //根据错误号获取字符串
errno 0 代表成功,非零代表错误，由 strerror 获取提示
```

运行结果如下：

```
1.  root@ubuntu:/home/linux/chapter5# vim Example5.2-1.c
2.  root@ubuntu:/home/linux/chapter5# gcc Example5.2-1.c -o Example5.2-1
3.  root@ubuntu:/home/linux/chapter5# ./example5.2-1 test
```

用户可以在当前文件夹下查看刚才创建的 test 文件夹。

文件或目录创建后，查看权限可能不是自己通过函数设定的权限。原因是文件或目录的权限不能超过系统设定的最大权限。对于文件和目录创建之后的 file mode 权限，按照如下方法计算：

文件创建权限如下：

```
PERM_MAX_FILE & (mode)
```

目录创建权限如下：

```
PERM_MAX_DIR & (mode)
```

其中，mode 就是创建文件或目录时输入的参数。

而最大权限的计算方法如下：

```
PERM_MAX_FILE = 0666 & ~(umask) //umask 是权限掩码,为系统内建 umask 的设定值
PERM_MAX_DIR = 0777 & ~(umask)
```

例如：

```
umask 0022
PERM_MAX_FILE = 0666 & ~(umask)  = 0644
PERM_MAX_DIR = 0777 & ~(umask) = 0755
```

所以在创建文件时，指定的权限不能超过 MAX。

2．删除目录

```
int rmdir(const char *path);
```

返回值：如果返回 0，则表示成功；如果返回−1，则表示失败。

说明：只能删除空目录。

3．获取当前目录及执行目录

（1）获取当前目录，即执行程序时所在的目录。

表头文件：

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

定义函数：

```
char *getcwd(char *buf,size_t size);
```

函数说明：

buf：保存当前目录的内存地址。

size：为内存的大小。

返回值：如果获取成功，则返回获取的目录，如果获取失败，则返回 NULL。

示例 5.2-2　获取当前目录。

```
1.  #include <unistd.h>
2.  #include <stdio.h>
3.  #include <errno.h>
4.  #include <string.h>
5.  int main(int argc,char **argv){
6.          char dir[256];
7.          getcwd(dir,256);  //设定 buf 的内存空间为 256 字节
8.          printf("---- %s\n",./dir);  //输出当前的工作目录
9.          return 0;
```

```
10.     }
```

运行结果如下：

```
1.  root@ubuntu:/home/linux/chapter5# gcc Example5.2-2.c -o Example5.2-2
2.  root@ubuntu:/home/linux/chapter5# ./Example5.2-2
3.  ---- /home/linux/chapter5
```

第7行代码中，getcwd()会将当前的工作目录绝对路径复制到参数buf所指的内存空间，即字符型数dir中。在调用此函数时，buf所指的内存空间要足够大，若工作目录绝对路径的字符串长度超过参数size大小，则返回值NULL。

（2）获取程序运行路径，即应用程序存放的位置。

表头文件：

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

定义函数：

```
int  readlink(const  char *path,  char *buf, size_t  bufsiz);
```

函数说明：

path：符号链接，在Linux中执行程序都用"/proc/self/exe"符号链接。

buf：用来写入正在执行的文件名（包含绝对路径）的内存。

bufsiz：指明buf的大小。

返回值：如果获取成功，则返回符号链接所指的文件路径字符串，如果获取失败，则返回−1。

示例5.2-3 获取当前文件或者目录的路径。

```
1.  #include <unistd.h>
2.  #include <stdio.h>
3.  #include <errno.h>
4.  #include <string.h>
5.  int main(int argc,char **argv){
6.  //获取程序目录，执行时在其他目录执行
7.          char dir[256]={0};
8.          int err=readlink("/proc/self/exe",dir,256); //读取程序执行路径
9.          if (err==-1) return -1;  //如果读取失败，则返回-1
10.         int i;
11.         for(i=strlen(dir)-1;i>=0 && dir[i]!='/';i--);
12.         dir[i]=0;
13.         //打开程序所在目录中的文件
14.         char filename[256];
15.         sprintf(filename,"%s/%s",dir,argv[1]);
16.         printf("---- %s\n",filename);
17.         FILE *fp=fopen(filename,"r");
18.         if (fp==NULL){
19.             printf("---- %s\n",strerror(errno));
20.         }
21.         return 0;
```

```
22.     }
```

运行结果如下：

```
1.  root@ubuntu:/home/linux/chapter5# gcc Example5.2-3.c -o Example5.2-3
2.  root@ubuntu:/home/linux/chapter5# ./Example5.2-3 test
3.  ---- /home/linux/chapter5/test
```

4．获取目录或文件的状态

表头文件：

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

定义函数：

（1）读取指定文件或目录的状态信息。

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

（2）读取已打开的文件状态信息。

```
int fstat(int filedes, struct stat *buf);
```

函数说明：

path：路径或文件名。

filedes：已打开文件或目录的句柄。

buf：是 struct stat 结构的指针，其结构格式如下：

```
struct stat{
    unsigned short st_mode;          //文件保护模式(即文件类型)
    unsigned short st_nlink;         //硬链接引用数
    unsigned short st_uid;           //文件的用户标识
    unsigned short st_gid;           //文件的组标识
    unsigned long  st_size;          //文件大小
    unsigned long  st_atime;         //文件最后的访问时间
    unsigned long  st_atime_nsec;    //文件最后的访问时间的秒数的小数
    unsigned long  st_mtime;         //文件最后的修改时间
    unsigned long  st_mtime_nsec;
    unsigned long  st_ctime;         //文件最后状态的改变时间
    unsigned long  st_ctime_nsec;
              ...
     };
```

返回值：返回获取文件状态的信息。

（3）判断文件类型及访问权限。

方法 1：在 struct stat 结构中，由 st_mode 字段记录了文件类型及访问权限，操作系统提供了一系列的宏来判断文件类型。结果如下：

```
S_ISREG(mode)        //判断是否为普通文件
S_ISDIR(mode)        //判断是否为目录文件
S_ISCHR(mode)        //判断是否为字符设备文件
```

```
S_ISBLK(mode)        //判断是否为块设备文件
S_ISFIFO(mode)       //判断是否为管道设备文件
S_ISLNK(mode)        //判断是否为符号链接
```

示例 5.2-4 判断文件类型。

```
1.  #include <unistd.h>
2.  #include <stdio.h>
3.  #include <errno.h>
4.  #include <string.h>
5.  #include <sys/types.h>
6.  #include <sys/stat.h>
7.  int main(int argc,char **argv){
8.   struct stat st;                 //定义一个 stat 类型的结构体，命名为 st
9.   int err= stat(argv[1],&st);     //读取文件或目录状态信息
10.  if (err==-1) return -1;         //如果执行失败，返回-1
11.  if (S_ISDIR(st.st_mode)) printf("isdir\n");        //访问 st_mode 字段，
                                                       //判断是否为目录文件
12.  else if (S_ISREG(st.st_mode)) printf("file\n");    //访问 st_mode 字段，
                                                       //判断是否为普通文件
13. }
```

运行结果如下：

```
1.  root@ubuntu:/home/linux/chapter5# chmod 777 Example5.2-4.c
2.  root@ubuntu:/home/linux/chapter5# gcc Example5.2-4.c -o Example5.2-4
3.  root@ubuntu:/home/linux/chapter5# ./Example5.2-4 test
4.  isdir
5.  root@ubuntu:/home/linux/chapter5# ./Example5.2-4 Example5.2-3.c
6.  file
```

方法 2：也可以直接读取 st_mode 内的数据，不过 st_mode 内的数据是组合而成的数据，包括很多信息，需要进行“与”运算才能读取这个字段中的数据。st_mode 是用特征位来表示文件类型的，特征位的定义如下：

S_IFMT	0170000	文件类型的位遮罩
S_IFSOCK	0140000	socket
S_IFLNK	0120000	符号链接（symbolic link）
S_IFREG	0100000	一般文件
S_IFBLK	0060000	区块装置（block device）
S_IFDIR	0040000	目录
S_IFCHR	0020000	字符装置（character device）
S_IFIFO	0010000	先进先出（FIFO）
S_ISUID	0004000	文件的 set user-id on execution 位
S_ISGID	0002000	文件的 set group-id on execution 位
S_ISVTX	0001000	文件的 sticky 位
S_IRWXU	00700	文件所有者的遮罩值（即所有权限值）

S_IRUSR	00400	文件所有者具有可读取权限
S_IWUSR	00200	文件所有者具有可写入权限
S_IXUSR	00100	文件所有者具有可执行权限
S_IRWXG	00070	用户组的遮罩值（即所有权限值）
S_IRGRP	00040	用户组具有可读取权限
S_IWGRP	00020	用户组具有可写入权限
S_IXGRP	00010	用户组具有可执行权限
S_IRWXO	00007	其他用户的遮罩值（即所有权限值）
S_IROTH	00004	其他用户具有可读取权限
S_IWOTH	00002	其他用户具有可写入权限
S_IXOTH	00001	其他用户具有可执行权限

示例 5.2-5　判断文件类型。

```
1.  #include <unistd.h>
2.  #include <stdio.h>
3.  #include <errno.h>
4.  #include <string.h>
5.  #include <sys/types.h>
6.  #include <sys/stat.h>
7.  int main(int argc,char **argv){
8.       struct stat st;                 //定义一个 stat 类型的结构体，命名为 st
9.       int err= stat(argv[1],&st);     //读取文件或目录状态信息
10.      if (err==-1) return -1;         //如果执行失败，则返回-1
11.      if (st.st_mode & S_IFREG)       //与对应的标志位相与运算
12.          printf("file\n");
13.      else if (st.st_mode & S_IFDIR) //与对应的标志位相与运算
14.          printf("isdir\n");
15. }
```

运行结果如下：

```
1.  root@ubuntu:/home/linux/chapter5# gcc Example5.2-5.c -o Example5.2-5
2.  root@ubuntu:/home/linux/chapter5# ./Example5.2-5 Example5.2-3.c
3.  file
4.  root@ubuntu:/home/linux/chapter5# ./Example5.2-5 test
5.  isdir
```

5.3　文件操作

5.3.1　基本概念


微课视频

1．流

流指数据的永久性存储，主要指数据以文件为单位存储在磁盘上。Linux 以字节为单位操作数据，所有的数据都是 0 或 1 的序列，如果需要让读者读懂数据，则以字符的方式显示出来，这就是所谓的文本文件。

而数据不是存在磁盘上后就永远不动，往往要被读入内存、传送到外部设备或搬移到其他位置，所以数据不断地在流动。然而不同设备之间的连接方法差异很大，数据读取和写入的方式也不相同，所以 Linux 定义了流（stream），建立了一个统一的接口，无论数据是从内存到外设，还是从内存到文件，都使用同一个数据输入/输出接口。

2．文件流和标准流

文件操作有两种方法：原始 I/O 和标准 I/O。

1）标准 I/O

标准 I/O 是标准 C 的输入/输出库，fopen、fread、fwrite、fclose 都是标准 C 的输入/输出函数。标准 I/O 都使用 FILE * 流对象指针作为操作文件的唯一识别，所以标准 I/O 是针对流对象的操作，是带缓存（内存）机制的输入/输出。标准 I/O 又提供了如下 3 种不同方式的缓冲：

（1）全缓冲。即缓冲区被写满或是调用 fflush 后，数据才会被写入磁盘。

（2）行缓冲。即缓冲区被写满或是遇到换行符时，才会进行实际的 I/O 操作。当流涉及一个终端时（标准输入和标准输出），通常使用行缓冲。

（3）不缓冲。标准 I/O 库不对字符进行缓存处理。标准出错流 stderr 往往是不带缓存的，使得出错信息可以尽快显示出来。

2）原始 I/O

原始 I/O 又称文件 I/O，是 Linux 操作系统提供的 API，称为系统调用，是针对描述符（即一个编号）操作的，是无缓存机制。

3）文件描述符

创建一个新的文件或打开已有文件时，内核向进程返回一个非负整数（即编号），用来识别操作的是哪个文件。对于 Linux 而言，所有打开的文件都是通过文件描述符引用的，在操作系统内部有一个宏定义了描述符的最大取值 OPENMAX，不同版本的 Linux 的取值不同。

Linux 编程使用的 open、close、read、write 等文件 I/O 函数属于系统调用的，其实现方式是用了 fctrl、ioctrl 等一些底层操作的函数。而标准 I/O 库中提供的是 fopen、fclose、fread、fwrite 等面向流对象的 I/O 函数，这些函数在实现时本身就要调用 Linux 的文件 I/O。在应用上，文件读写时二者并没区别，但是一些特殊文件，例如管道等只能使用文件 I/O 操作。

3．标准输入、标准输出和标准错误

当 Linux 执行一个程序时，会自动打开三个流：标准输入（standard input）、标准输出（standard output）和标准错误（standard error）。命令行的标准输入连接到键盘，标准输出和标准错误都连接到屏幕。对于一个程序来说，尽管它总会打开这三个流，但它会根据需要使用，并不是一定要使用。系统默认打开的三个文件描述符（为进程预定义的三个流）如下：

```
STDIN_FILENO    0 标准输入,用于从键盘获取数据
STDOUT_FILENO   1 标准输出,用于向屏幕输出数据
STDERR_FILENO   2 标准错误,用于获取错误信息
```

例如，使用标准输入和输出：