# 应用学习

## 1.应用编程概念

目录

[应用学习 1](#_Toc105863759)

[1.应用编程概念 1](#_Toc105863760)

[1.1系统调用 1](#_Toc105863761)

[1.2文件 I/O 基础 2](#_Toc105863762)

[1.3深入探究文件 I/O 7](#_Toc105863763)

[1.4标准 I/O 库 11](#_Toc105863764)

### 1.1系统调用

* 不止 Linux 系统，所有的操作系统都会向应用层提供系统调用，应用程序通过系统调用来使用操作系统提供的各种服务。
* LED 应用程序与 LED 驱动程序是分隔、分离的，它们单独编译，它们并不是整合在一起的，应用程序运行在操作系统之上，有操作系统支持，应用程序处于用户态，而驱动程序处于内核态，与纯粹的裸机程序存在着质的区别。

1.2库函数

系统调用和库函数之间的区别：

库函数是属于应用层，而系统调用是内核提供给应用层的编程接口，属于系统内核的一部分；

库函数运行在用户空间，调用系统调用会由用户空间（用户态）陷入到内核空间（内核态）；

库函数通常是有缓存的，而系统调用是无缓存的，所以在性能、效率上，库函数通常要优于系统调 用；

可移植性：库函数相比于系统调用具有更好的可移植性，通常对于不同的操作系统，其内核向应用 层提供的系统调用往往都是不同，譬如系统调用的定义、功能、参数列表、返回值等往往都是不一样的；而对于 C 语言库函数来说，由于很多操作系统都实现了 C 语言库，C 语言库在不同的操作系统之间其接口定义几乎是一样的，所以库函数在不同操作系统之间相比于系统调用具有更好的可移植性。

1.3 标准 C 语言函数库

在 Linux 系 统 下 ， 使 用 的 C 语 言 库 为 GNU C 语 言 函 数 库 （ 也 叫 作 glibc ， 其 网 址 为 <http://www.gnu.org/software/libc/>），作为 Linux 下的标准 C 语言函数库。 进入到 <http://www.gnu.org/software/libc/>网址

进入到 Ubuntu 系统的/lib 目录下，笔者使用的 Ubuntu 版本为 16.04，在我的/lib 目录下并没有发现 libc.so.6 这个文件，其实是在/lib/x86\_64-linux-gnu 目录下：

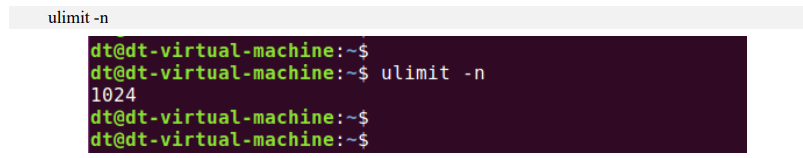
### 1.2文件 I/O 基础

#### 文件描述符

文件描述符是一个非负整数；对于 Linux 内核而言，所有打开的文件都会通过文件描述符进行索引。

一个进程可以打开多个文件，但是在 Linux 系统中，一个进程可以打开的文件数是有限制，并不是可以 无限制打开很多的文件。

在 Linux 系统下，我们可以通过 ulimit 命令来查看进 程可打开的最大文件数，用法如下所示：



文件描述符是从 0 开始分配的 0-1023

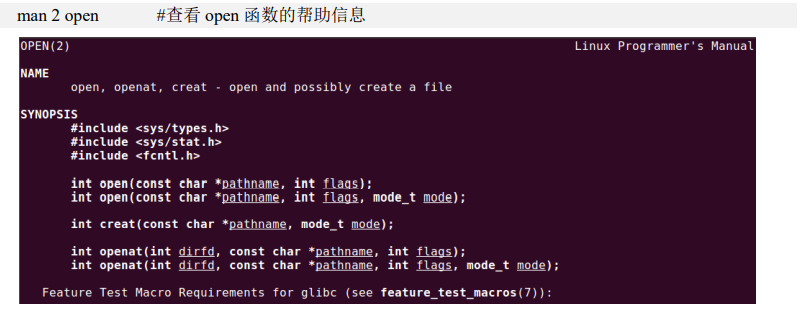
如果文件被关闭后，它对应的文件描述符将会被释放，那么这个文件描 述符将可以再次分配给其它打开的文件、与对应的文件绑定起来。

当我们在程序中，调用 open 函数打开文件的时候，分配的文件描述符一般都是从 3 开始，这里大家可 能要问了，上面不是说从 0 开始的吗，确实是如此，但是 0、1、2 这三个文件描述符已经默认被系统占用 了，分别分配给了系统标准输入（0）、标准输出（1）以及标准错误（2）。

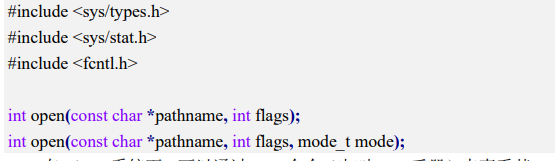
#### Main

可以通过 man 命令（也叫 man 手册）来查看某一个 Linux 系统调用的帮助信息，man 命令可以将该系统调用的详细信息显示出来，譬如函数功能介绍、函数原型、参数、返回值以及使用该函数 所需包含的头文件等信息。

**Tips：**man 命令后面跟着两个参数，数字 2 表示系统调用，man 命令除了可以查看系统调用的帮助信息 外，还可以查看 Linux 命令（对应数字 1）以及标准 C 库函数（对应数字 3）所对应的帮助信息；最后一个 参数 open 表示需要查看的系统调用函数名。



#### open函数



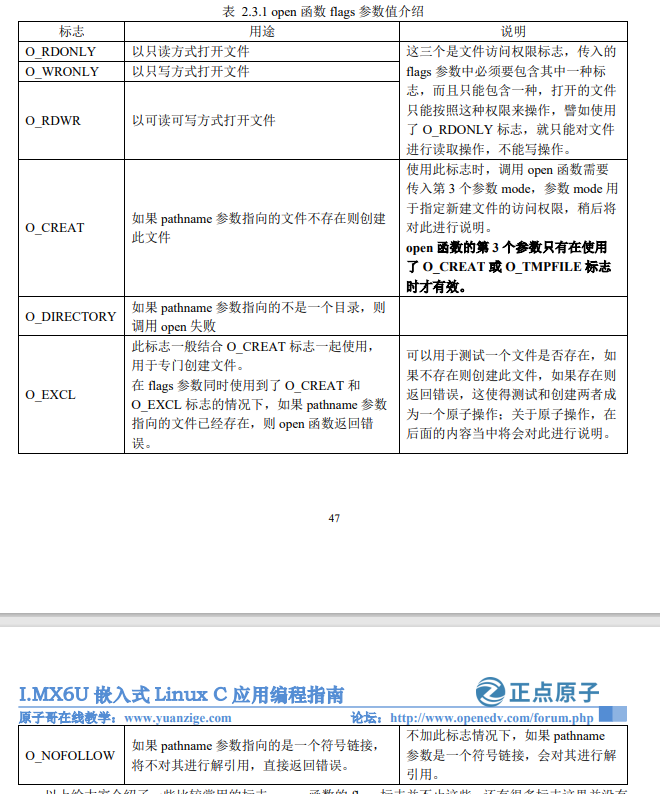
函数参数和返回值含义如下：

pathname：字符串类型，用于标识需要打开或创建的文件，可以包含路径（绝对路径或相对路径）信 息，譬如："./src\_file"（当前目录下的 src\_file 文件）、"/home/dengtao/hello.c"等；如果 pathname 是一个符号 链接，会对其进行解引用。

flags：调用 open 函数时需要提供的标志，包括文件访问模式标志以及其它文件相关标志，这些标志使 用宏定义进行描述，都是常量，open 函数提供了非常多的标志，我们传入 flags 参数时既可以单独使用某一 个标志，也可以通过位或运算（|）将多个标志进行组合。

以下给大家介绍了一些比较常用的标志，open 函数的 flags 标志并不止这些，还有很多标志这里并没有 给大家进行介绍，譬如 O\_APPEND、O\_ASYNC、O\_DSYNC、O\_NOATIME、O\_NONBLOCK、O\_SYNC 以 及 O\_TRUNC 等，对于这些没有提及到的标志，在后面学习过程中，也会给大家慢慢介绍。对于初学者来 说，我们需要把表 2.3.1 中所列出的这些标志给弄明白、理解它们的作用和含义。

**Tips**：不同内核版本所支持的 flags 标志是存在差别的，譬如说新版本内核所支持的标志可能在老版本 是不支持的，亦或者老版本支持的标志在新版本已经被取消、替代，man 手册中对一些标志是从哪个版本开 始支持的有简单地说明，读者可以自行阅读！



前面我们说过，flags 参数时既可以单独使用某一个标志，也可以通过位或运算（|）将多个标志进行组 合，譬如：

**open("./src\_file", O\_RDONLY) //单独使用某一个标志**

**open("./src\_file", O\_RDONLY | O\_NOFOLLOW) //多个标志组合**

**mode：**此参数用于指定新建文件的访问权限，只有当 flags 参数中包含 O\_CREAT 或 O\_TMPFILE 标志 时才有效（O\_TMPFILE 标志用于创建一个临时文件）。权限对于文件来说是一个很重要的属性，那么在 Linux 系统中，我们可以通过 touch 命令新建一个文件，此时文件会有一个默认的权限，如果需要修改文件权限， 可通过 chmod 命令对文件权限进行修改，譬如在 Linux 系统下我们可以使用"ls -l"命令来查看到文件所对应 的权限。

mode 参数的类型是 mode\_t，这是 一个 u32 无符号整形数据，权限表示方法如下所示



这些宏既可以单独使用，也可以通过位或运算将多个宏组合在一起，譬如： S\_IRUSR | S\_IWUSR | S\_IROTH

**返回值**：成功将返回文件描述符，文件描述符是一个非负整数；失败将返回-1。 以上就把 open 函数相关的基础知识给大家介绍完了，包括函数返回值、参数等信息，当然在后面的章 节内容中，我们还会更加深入地给大家讲解 open 函数相关的知识点；接下来我们看一些 open 函数的简答使 用示例。

**open 函数使用示例**

(1)使用 open 函数打开一个已经存在的文件（例如当前目录下的 app.c 文件），使用只读方式打开：

int fd = open("./app.c", O\_RDONLY)

if (-1 == fd)

return fd;

(2)使用 open 函数打开一个已经存在的文件（例如当前目录下的 app.c 文件），使用可读可写方式打开：

int fd = open("./app.c", O\_RDWR)

if (-1 == fd)

return fd;

(3)使用 open 函数打开一个指定的文件（譬如/home/dengtao/hello），使用可读可写方式,如果该文件是 一个符号链接文件，则不对其进行解引用，直接返回错误：

int fd = open("/home/dengtao/hello", O\_RDWR | O\_NOFOLLOW);

if (-1 == fd)

return fd;

(4)使用 open 函数打开一个指定的文件（譬如/home/dengtao/hello），如果该文件不存在则创建该文件， 创建该文件时，将文件权限设置如下： 文件所属者拥有读、写、执行权限； 同组用户与其他用户只有读权限。 使用可读可写方式打开：

int fd = open("/home/dengtao/hello", O\_RDWR | O\_CREAT, S\_IRWXU | S\_IRGRP | S\_IROTH); if (-1 == fd)

return fd;

#### write函数

#include <unistd.h>

ssize\_t write(int fd, const void \*buf, size\_t count);

首先使用 write 函数需要先包含 unistd.h 头文件。

函数参数和返回值含义如下： fd：文件描述符。关于文件描述符，前面已经给大家进行了简单地讲解，这里不再重述！我们需要将进 行写操作的文件所对应的文件描述符传递给 write 函数。 buf：指定写入数据对应的缓冲区。 count：指定写入的字节数。

一个很重要的问题是：从文件的哪个位置开始进行读写操作？也就是 IO 操作所对 应的位置偏移量，读写操作都是从文件的当前位置偏移量处开始，当然当前位置偏移量可以通过 lseek 系统 调用进行设置，关于此函数后面再讲；默认情况下当前位置偏移量一般是 0，也就是指向了文件起始位置， 当调用 read、write 函数读写操作完成之后，当前位置偏移量也会向后移动对应字节数，譬如当前位置偏移 量为 1000 个字节处，调用 write()写入或 read()读取 500 个字节之后，当前位置偏移量将会移动到 1500 个字 节处。

#### read函数

ssize\_t read(int fd, void \*buf, size\_t count);

首先使用 read 函数需要先包含 unistd.h 头文件。

函数参数和返回值含义如下： fd：文件描述符。与 write 函数的 fd 参数意义相同。 buf：指定用于存储读取数据的缓冲区。count：指定需要读取的字节数。

#### Lseek

off\_t lseek(int fd, off\_t offset, int whence);

首先调用 lseek 函数需要包含和两个头文件。

函数参数和返回值含义如下： fd：文件描述符。 offset：偏移量，以字节为单位。 whence：用于定义参数 offset 偏移量对应的参考值，该参数为下列其中一种（宏定义）：

⚫ SEEK\_SET：读写偏移量将指向 offset 字节位置处（从文件头部开始算）

⚫ SEEK\_CUR：读写偏移量将指向当前位置偏移量 + offset 字节位置处，offset 可以为正、也可以为 负，如果是正数表示往后偏移，如果是负数则表示往前偏移； ⚫ SEEK\_END：读写偏移量将指向文件末尾 + offset 字节位置处，同样 offset 可以为正、也可以为负， 如果是正数表示往后偏移、如果是负数则表示往前偏移。

使用示例：

(1) 将读写位置移动到文件开头处：

off\_t off = lseek(fd, 0, SEEK\_SET);

if (-1 == off) return -1;

(2) 将读写位置移动到文件末尾：

off\_t off = lseek(fd, 0, SEEK\_END);

if (-1 == off) return -1;

(3)将读写位置移动到偏移文件开头 100 个字节处：

off\_t off = lseek(fd, 100, SEEK\_SET);

if (-1 == off) return -1;

(4)获取当前读写位置偏移量：

off\_t off = lseek(fd, 0, SEEK\_CUR);

if (-1 == off) return -1;

函数执行成功将返回文件当前读写位置。

### 1.3深入探究文件 I/O

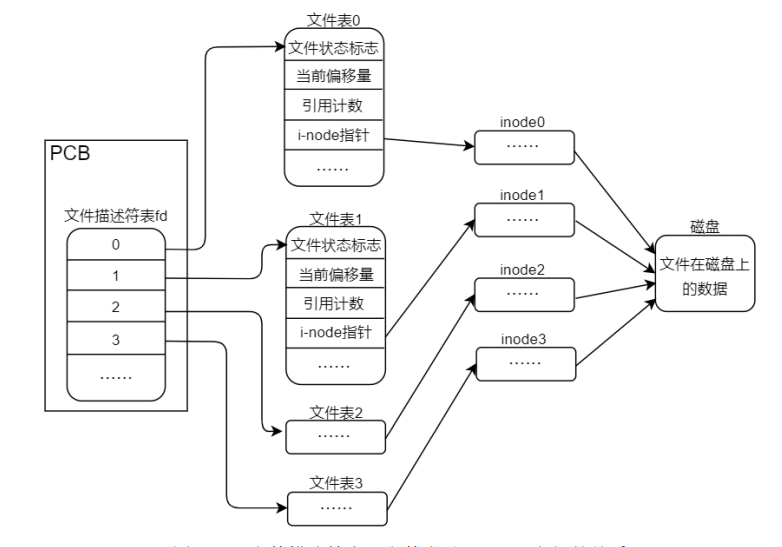
**inode**

文件储存在硬盘上，硬盘的最小存储单位叫做“扇区”（Sector），每个扇区储存 512 字节（相当于 0.5KB）， 操作系统读取硬盘的时候，不会一个个扇区地读取，这样效率太低，而是一次性连续读取多个扇区，即一次 性读取一个“块”（block）。这种由多个扇区组成的“块”，是文件存取的最小单位。“块”的大小，最常 见的是 4KB，即连续八个 sector 组成一个 block。

**PCB**

在 Linux 系统中，内核会为每个进程（关于进程的概念，这是后面的内容，我们可以简单地理解为一个 运行的程序就是一个进程，运行了多个程序那就是存在多个进程）设置一个专门的数据结构用于管理该进 程，譬如用于记录进程的状态信息、运行特征等，我们把这个称为进程控制块（Process control block，缩写 PCB）。

PCB 数据结构体中有一个指针指向了文件描述符表（File descriptors），文件描述符表中的每一个元素 索引到对应的文件表（File table），文件表也是一个数据结构体，其中记录了很多文件相关的信息，譬如文 件状态标志、引用计数、当前文件的读写偏移量以及 i-node 指针（指向该文件对应的 inode）等，进程打开 的所有文件对应的文件描述符都记录在文件描述符表中，每一个文件描述符都会指向一个对应的文件表， 其示意图如下所示：



**返回错误处理与 errno**

**strerror 函数**

前面给大家说到了 errno 变量，但是 errno 仅仅只是一个错误编号，对于开发者来说，即使拿到了 errno 也不知道错误为何？还需要对比源码中对此编号的错误定义，可以说非常不友好，这里介绍一个 C 库函数 strerror()，该函数可以将对应的 errno 转换成适合我们查看的字符串信息，其函数原型如下所示（可通过"man 3 strerror"命令查看，注意此函数是 C 库函数，并不是系统调用）：

char \*strerror(int errnum);

首先调用此函数需要包含头文件<string.h>。

函数参数和返回值如下： errnum：错误编号 errno。

返回值：对应错误编号的字符串描述信息。

**perror 函数**

除了 strerror 函数之外，我们还可以使用 perror 函数来查看错误信息，一般用的最多的还是这个函数， 调用此函数不需要传入 errno，函数内部会自己去获取 errno 变量的值，调用此函数会直接将错误提示字符 串打印出来，而不是返回字符串，除此之外还可以在输出的错误提示字符串之前加入自己的打印信息，函数 原型如下所示（可通过"man 3 perror"命令查看）：

void perror(const char \*s);

需要包含头文件。

函数参数和返回值含义如下：

s：在错误提示字符串信息之前，可加入自己的打印信息，也可不加，不加则传入空字符串即可。

返回值：void 无返回值。

**exit、\_exit、\_Exit**

exit()函数\_exit()函数都是用来终止进程的，exit()是一个标准 C 库函数，而\_exit()和\_Exit()是系统调用。 执行 exit()会执行一些清理工作，最后调用\_exit()函数。exit()函数原型如下：

void exit(int status); 该函数是一个标准 C 库函数，使用该函数需要包含头文件，该函数的用法和\_exit()/\_Exit()是 一样的，这里就不再多说了。

**空洞文件**

接下来使用 write()函数对文件进行写入操作，也就是说此时将是从偏移文件头部 6000 个字节处开始写 入数据，也就意味着 4096~6000 字节之间出现了一个空洞，因为这部分空间并没有写入任何数据，所以形 成了空洞，这部分区域就被称为文件空洞，那么相应的该文件也被称为空洞文件。

**O\_APPEND 和 O\_TRUNC 标志**

O\_TRUNC ：这个标志的作用非常简单，如果使用了这个标志，调用 open 函数打开文件的时候会将文件 原本的内容全部丢弃，文件大小变为 0；

调用 open 函数打开文件， 当每次使用 write()函数对文件进行写操作时，都会自动把文件O\_APPEND 标志：当前位置偏移量移动到文件末尾，从文件末 尾开始写入数据，也就是意味着每次写入数据都是从文件末尾开始。

**复制文件描述符**

dup 函数

dup 函数用于复制文件描述符，此函数原型如下所示

int dup(int oldfd); 首先使用此函数需要包含头文件<unistd.h>

函数参数和返回值含义如下： oldfd：需要被复制的文件描述符。 返回值：成功时将返回一个新的文件描述符，由操作系统分配，分配置原则遵循文件描述符分配原则； 如果复制失败将返回-1，并且会设置 errno 值。

fd1 等于 6，复制得到的新的文件描述符为 7（遵循 fd 分配原则），打印出来的数据 显示为接续写，所以可知，通过复制文件描述符可以实现接续写。

**dup2 函数**

int dup2(int oldfd, int newfd); 同样使用该命令也需要包含头文件。

函数参数和返回值含义如下： oldfd：需要被复制的文件描述符。 newfd：指定一个文件描述符（需要指定一个当前进程没有使用到的文件描述符）。 返回值：成功时将返回一个新的文件描述符，也就是手动指定的文件描述符 newfd；如果复制失败将返 回-1，并且会设置 errno 值。



**文件共享**

对于文件共享，存在着竞争冒险，这个是需要大家关注的，下一小节将会向大家介绍。除此之外，我们 还需要关心的是文件共享时，不同的读写体之间是分别写还是接续写，这些细节问题大家都要搞清楚。

**原子操作与竞争冒险**

原子操作:

**（1）O\_APPEND 实现原子操作**

**（2）pread()和 pwrite()**

pread()和 pwrite()可用于实现原子操作，调用 pread 函数或 pwrite 函数可传入一个位置偏移量 offset 参数， 用于指定文件当前读或写的位置偏移量，所以调用 pread 相当于调用 lseek 后再调用 read；同理，调用 pwrite 相当于调用 lseek 后再调用 write。所以可知，使用 pread 或 pwrite 函数不需要使用 lseek 来调整当前位置偏 移量，并会将“移动当前位置偏移量、读或写”这两步操作组成一个原子操作。

ssize\_t pread(int fd, void \*buf, size\_t count, off\_t offset);

ssize\_t pwrite(int fd, const void \*buf, size\_t count, off\_t offset);

首先调用这两个函数需要包含头文件<unistd.h>。

函数参数和返回值含义如下：

fd、buf、count 参数与 read 或 write 函数意义相同。

offset：表示当前需要进行读或写的位置偏移量。

返回值：返回值与 read、write 函数返回值意义一样。

虽然 pread（或 pwrite）函数相当于 lseek 与 pread（或 pwrite）函数的集合，但还是有下列区别：

⚫ 调用 pread 函数时，无法中断其定位和读操作（也就是原子操作）；

⚫ 不更新文件表中的当前位置偏移量。

**(3)创建一个文件**

**fcntl 和 ioctl**

fcntl()函数可以对一个已经打开的文件描述符执行一系列控制操作，譬如复制一个文件描述符（与 dup、 dup2 作用相同）、获取/设置文件描述符标志、获取/设置文件状态标志等，类似于一个多功能文件描述符管 理工具箱。fcntl()函数原型如下所示（可通过"man 2 fcntl"命令查看）：

int fcntl(int fd, int cmd, ... /\* arg \*/ )

函数参数和返回值含义如下： fd：文件描述符。 cmd：操作命令。此参数表示我们将要对 fd 进行什么操作，cmd 参数支持很多操作命令，大家可以打 开 man 手册查看到这些操作命令的详细介绍，这些命令都是以 F\_XXX 开头的，譬如 F\_DUPFD、F\_GETFD、 F\_SETFD 等，不同的 cmd 具有不同的作用，cmd 操作命令大致可以分为以下 5 种功能：

⚫ 复制文件描述符（cmd=F\_DUPFD 或 cmd=F\_DUPFD\_CLOEXEC）；

⚫ 获取/设置文件描述符标志（cmd=F\_GETFD 或 cmd=F\_SETFD）；

⚫ 获取/设置文件状态标志（cmd=F\_GETFL 或 cmd=F\_SETFL）；

⚫ 获取/设置异步 IO 所有权（cmd=F\_GETOWN 或 cmd=F\_SETOWN）；

⚫ 获取/设置记录锁（cmd=F\_GETLK 或 cmd=F\_SETLK）；

…：fcntl 函数是一个可变参函数，第三个参数需要根据不同的 cmd 来传入对应的实参，配合 cmd 来使 用。

返回值：执行失败情况下，返回-1，并且会设置 errno；执行成功的情况下，其返回值与 cmd（操作命 令）有关，譬如 cmd=F\_DUPFD（复制文件描述符）将返回一个新的文件描述符、cmd=F\_GETFD（获取文 件描述符标志）将返回文件描述符标志、cmd=F\_GETFL（获取文件状态标志）将返回文件状态标志等。

**ioctl 函数**

ioctl()可以认为是一个文件 IO 操作的杂物箱，可以处理的事情非常杂、不统一，一般用于操作特殊文件 或硬件外设，此函数将会在进阶篇中使用到，譬如可以通过 ioctl 获取 LCD 相关信息等，本小节只是给大家 引出这个系统调用，暂时不会用到。此函数原型如下所示（可通过"man 2 ioctl"命令查看）： #include int ioctl(int fd, unsigned long request, ...); 使用此函数需要包含头文件。 函数参数和返回值含义如下： fd：文件描述符。 request：此参数与具体要操作的对象有关，没有统一值，表示向文件描述符请求相应的操作；后面用 到的时候再给大家介绍。 ...：此函数是一个可变参函数，第三个参数需要根据 request 参数来决定，配合 request 来使用。 返回值：成功返回 0，失败返回-1。

**截断文件**

int truncate(const char \*path, off\_t length)

int ftruncate(int fd, off\_t length);

这两个函数的区别在于：ftruncate()使用文件描述符 fd 来指定目标文件，而 truncate()则直接使用文件路 径 path 来指定目标文件，其功能一样。

### 1.4标准 I/O 库

#### 1.4.1标准I/O库简介

标准 I/O 却有它自己的优势，标准 I/O 和文件 I/O 的区别如下：  
⚫ 虽然标准 I/O 和文件 I/O 都是 C 语言函数，但是标准 I/O 是标准 C 库函数，而文件 I/O 则是 Linux  
系统调用；  
⚫ 标准 I/O 是由文件 I/O 封装而来，标准 I/O 内部实际上是调用文件 I/O 来完成实际操作的；  
⚫ 可移植性：标准 I/O 相比于文件 I/O 具有更好的可移植性，通常对于不同的操作系统，其内核向应  
用层提供的系统调用往往都是不同，譬如系统调用的定义、功能、参数列表、返回值等往往都是不  
一样的；而对于标准 I/O 来说，由于很多操作系统都实现了标准 I/O 库，标准 I/O 库在不同的操作  
系统之间其接口定义几乎是一样的，所以标准 I/O 在不同操作系统之间相比于文件 I/O 具有更好的  
可移植性。  
⚫ 性能、效率： 标准 I/O 库在用户空间维护了自己的 stdio 缓冲区， 所以标准 I/O 是带有缓存的，而  
文件 I/O 在用户空间是不带有缓存的，所以在性能、效率上，标准 I/O 要优于文件 I/O。

#### 1.4.2 FILE 指针

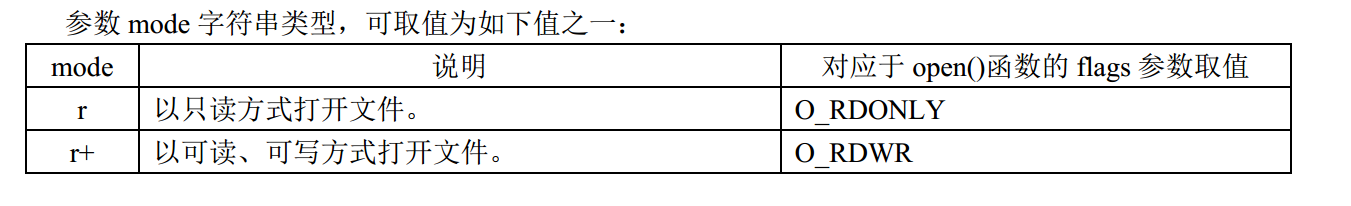
FILE 是一个结构体数据类型，它包含了标准 I/O 库函数为管理文件所需要的所有信息，包括用于实际  
I/O 的文件描述符、指向文件缓冲区的指针、缓冲区的长度、当前缓冲区中的字节数以及出错标志等。 FILE  
数据结构定义在标准 I/O 库函数头文件 stdio.h 中。

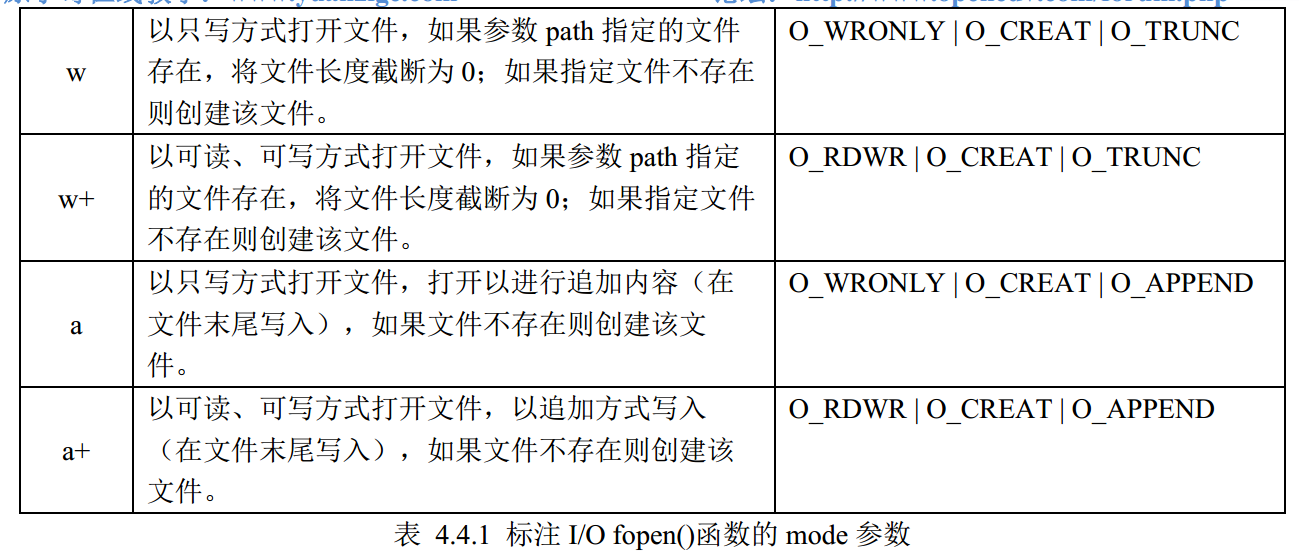
#### 1.4.3 标准输入、标准输出和标准错误

每个进程启动之后都会默认打开标准输入、标准输出以及标准错误， 得到三个文件描述符， 即 0、 1、  
2， 其中 0 代表标准输入、 1 代表标准输出、 2 代表标准错误； 在应用编程中可以使用宏 STDIN\_FILENO、  
STDOUT\_FILENO 和 STDERR\_FILENO 分别代表 0、 1、 2，这些宏定义在 unistd.h 头文件中：  
/\* Standard file descriptors. \*/  
#define STDIN\_FILENO 0 /\* Standard input. \*/  
#define STDOUT\_FILENO1 /\* Standard output. \*/  
#define STDERR\_FILENO2 /\* Standard error output. \*/  
0、 1、 2 这三个是文件描述符，只能用于文件 I/O（read()、 write()等），那么在标准 I/O 中，自然是无  
法使用文件描述符来对文件进行 I/O 操作的，它们需要围绕 FILE 类型指针来进行，在 stdio.h 头文件中有相  
应的定义，如下：  
/\* Standard streams. \*/  
extern struct \_IO\_FILE \*stdin; /\* Standard input stream. \*/  
extern struct \_IO\_FILE \*stdout; /\* Standard output stream. \*/  
extern struct \_IO\_FILE \*stderr; /\* Standard error output stream. \*/  
/\* C89/C99 say they're macros. Make them happy. \*/  
#define stdin stdin  
#define stdout stdout  
#define stderr stderr  
Tips： struct \_IO\_FILE 结构体就是 FILE 结构体，使用了 typedef 进行了重命名。  
所以，在标准 I/O 中，可以使用 stdin、 stdout、 stderr 来表示标准输入、标准输出和标准错误。

#### 1.4.4 打开文件 fopen()

使用 open()系统调用打开或创建文件，而在标准 I/O 中，我们将使用库函数  
fopen()打开或创建文件， fopen()函数原型如下所示：  
#include <stdio.h>  
FILE \*fopen(const char \*path, const char \*mode);  
使用该函数需要包含头文件 stdio.h。  
函数参数和返回值含义如下：  
**path**： 参数 path 指向文件路径，可以是绝对路径、也可以是相对路径。  
**mode**： 参数 mode 指定了对该文件的读写权限，是一个字符串，稍后介绍。  
返回值： 调用成功返回一个指向 FILE 类型对象的指针（FILE \*），该指针与打开或创建的文件相关联，  
后续的标准 I/O 操作将围绕 FILE 指针进行。 如果失败则返回 NULL，并设置 errno 以指示错误原因。



  
表 4.4.1 标注 I/O fopen()函数的 mode 参数

由 fopen()函数原型可知， fopen()只有两个参数 path 和 mode，不同于 open()系统调用，它并没有任何一  
个参数来指定新建文件的权限。 当参数 mode 取值为"w"、 "w+"、 "a"、 "a+"之一时，如果参数 path 指定的文  
件不存在，则会创建该文件，那么新的文件的权限是如何确定的呢？  
虽然调用 fopen()函数新建文件时无法手动指定文件的权限，但却有一个默认值：  
S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IWGRP | S\_IROTH | S\_IWOTH (0666)

#### 1.4.5 读文件和写文件

fread()和 fwrite()库函数对文件进行读、写操  
作了，函数原型如下所示：  
#include <stdio.h>  
size\_t fread(void \*ptr, size\_t size, size\_t nmemb, FILE \*stream);  
size\_t fwrite(const void \*ptr, size\_t size, size\_t nmemb, FILE \*stream);

库函数 fread()用于读取文件数据，其参数和返回值含义如下：  
**ptr**： fread()将读取到的数据存放在参数 ptr 指向的缓冲区中；  
**size**： fread()从文件读取 nmemb 个数据项，每一个数据项的大小为 size 个字节，所以总共读取的数据大  
小为 nmemb \* size 个字节。  
**nmemb**： 参数 nmemb 指定了读取数据项的个数。  
**stream**： FILE 指针。  
返回值： 调用成功时返回读取到的数据项的数目（数据项数目并不等于实际读取的字节数，除非参数  
size 等于 1）；如果发生错误或到达文件末尾，则 fread()返回的值将小于参数 nmemb，那么到底发生了错误  
还是到达了文件末尾， fread()不能区分文件结尾和错误， 究竟是哪一种情况，此时可以使用 ferror()或 feof()

库函数 fwrite()用于将数据写入到文件中，其参数和返回值含义如下：  
**ptr**： 将参数 ptr 指向的缓冲区中的数据写入到文件中。  
**size**： 参数 size 指定了每个数据项的字节大小，与 fread()函数的 size 参数意义相同。  
**nmemb**： 参数 nmemb 指定了写入的数据项个数，与 fread()函数的 nmemb 参数意义相同。  
**stream**： FILE 指针。  
返回值： 调用成功时返回写入的数据项的数目（数据项数目并不等于实际写入的字节数，除非参数 size  
等于 1）；如果发生错误，则 fwrite()返回的值将小于参数 nmemb（或者等于 0）。  
由此可知，库函数 fread()、 fwrite()中指定读取或写入数据大小的方式与系统调用 read()、 write()不同，  
前者通过 nmemb（数据项个数） \*size（每个数据项的大小）的方式来指定数据大小，而后者则直接通过一  
个 size 参数指定数据大小。  
譬如要将一个 struct mystr 结构体数据写入到文件中，可按如下方式写入：  
fwrite(buf, sizeof(struct mystr), 1, file);  
当然也可以按如下方式写：  
fwrite(buf, 1, sizeof(struct mystr), file);

1.4.6 fseek 定位  
库函数 fseek()的作用类似于 2.7 小节所学习的系统调用 lseek()， 用于设置文件读写位置偏移量， lseek()  
用于文件 I/O，而库函数 fseek()则用于标准 I/O，其函数原型如下所示：  
#include <stdio.h>  
int fseek(FILE \*stream, long offset, int whence);  
函数参数和返回值含义如下：  
**stream**： FILE 指针。  
**offset**： 与 lseek()函数的 offset 参数意义相同。  
**whence**： 与 lseek()函数的 whence 参数意义相同。  
返回值： 成功返回 0；发生错误将返回-1，并且会设置 errno 以指示错误原因； 与 lseek()函数的返回值  
意义不同，这里要注意！  
调用库函数 fread()、 fwrite()读写文件时，文件的读写位置偏移量会自动递增，使用 fseek()可手动设置  
文件当前的读写位置偏移量。  
譬如将文件的读写位置移动到文件开头处：  
fseek(file, 0, SEEK\_SET);  
将文件的读写位置移动到文件末尾：  
fseek(file, 0, SEEK\_END);  
将文件的读写位置移动到 100 个字节偏移量处：  
fseek(file, 100, SEEK\_SET);