

更多嵌入式 Linux 学习资料, 请关注: 一口 Linux 回复关键字:1024



LINUX 基础 ——文件编程篇

Linux 一切皆是文件 文件系统（文件夹/文件）硬件设备，管道，数据库，Socket 等

一、文件编程概述：

1. 应用中比如：账单，游戏进度，配置文件等。 2. 用代码操作文件：实现文件创建，打开，编辑等自动化执行。

二、计算机如何帮我们自动化完成以上操作呢？

操作系统提供了一系列的 API 如 Linux 系统：

打开 open

读写 write/read

光标定位 lseek

关闭 close

三、文件打开及创建

打开/创建文件

头文件:

```
#include<sys/types.h> #include<sys/stat.h> #include<fcntl.h>
int open(const char *pathname , int flags); //(参数 1: 字符指针 指向
文件路径 , 参数 2: 整型数权限)
int open(const char *pathname , int flags , mode_t mode); //open 返回
值是文件描述符——没有文件返回值就无法用 write(); 和 read();
```

参数说明:

*pathname : //要打开的文件名 (含路径, 缺省为当前路径)

flags :

- O_RDONLY 只读打开
- O_WRONLY 只写打开
- O_RDWR 可读可写可打卡

当我们附帶了权限后, 打开的文件就只能按照这种权限来操作。 以上这三个常数中应当只指定一个。

```
int creat(const char *filename , mode_t mode)
```

filename:要创建的文件名 (包含路径, 缺省为当前路径)

mode:创建模式 //可读可写可执行

常见创建模式:

宏表示	数字	
S_IRUSR	4	可读
S_IWUSR	2	可写
S_IXUSR	1	可执行
S_IRWXU	7	可读, 写, 执行

示例代码: (在路径/Home/CLC/下创建了 file1 可读可写可执行文件)

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

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

int main()
{
    int fd;
    char *buf = "test";
    fd = creat("/home/CLC/file",S_IRWXU); //S_IRWXU 可读可写可执行
    return 0;
}
```

==下列常数是可选择的: ==

1. O_CREAT //若文件不存在则创建它。

使用此选项时, 需要同时说明第三个参数 mode, 用其说明该新文件的存取许可权限。

mode:一定是在 flags 中使用了 O_CREAT 标志, mode 记录待创建的文件的访问权限。

O_CREAT 若文件不存在则创建它 示例代码:

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

int main()
{
    int fd; //定义整型返回值
    fd = open("./file",O_RDWR); //open 指令 打开可读可写 file 文件返回值
    printf("fd = %d\n",fd);
    if(fd == -1){ //若返回值为-1 没有file 文件
        printf("open file failed\n");
        fd = open("./file" , O_RDWR | O_CREAT , 0600);
        //open 指令 若没有file 文件 " | O_CREAT " 创建file 文件 权限0600 6=4+2 所以是
        rw 可读可写
    }
    printf("creat file success\n");
}

printf("fd=%d\n ,fd");
```

```
return 0;
}
```

```
CLC@Embed_Learn:~$ ./mode
fd=-1
open file failed
create file success
fd=3
CLC@Embed_Learn:~$
```

结果:

注意:

```
fd = open("./file", O_RDWR); //open 指令 打开可读可写 file 文件返回值
fd = open("./file", O_RDWR | O_CREAT, 0600);
// open 指令 若没有 file 文件 " | O_CREAT " 创建 file 文件 权限 0600
6=4+2 所以是 rw 可读可写
```

权限 0600 的含义:

—表示普通文件 r (4) 可读 w(2)可写 rw 表示可读可写 x(1)执行 0600 :
6=4+2 所以是 rw 可读可写 0 是同组 0 其他组

2. O_EXCL //如果同时指定了 O_CREAT , 而文件已经存在, 则打开失败或者返回-1

O_EXCL //如果同时指定了 O_CREAT , 而文件已经存在, 则打开失败或者返回-1
示例代码:

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

int main(){
    int fd;
    fd = open("./file", O_RDWR | O_CREAT | O_EXCL, 0600);
    if(fd == -1){ //若返回值为-1 没有file 文件
        printf("file exist\n");
    }
    return 0;
}
```

```
}  
}
```

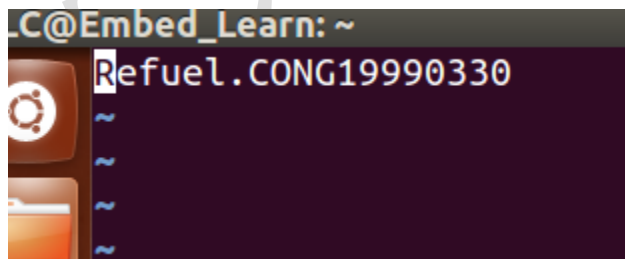
注意:

```
fd = open("./file", O_RDWR | O_CREAT | O_EXCL, 0600);
```

3. O_APPEND//每次写时都加到文件的尾端。

O_APPEND //每次写时都加到文件的尾端 示例代码:

```
#include<sys/types.h>  
#include<sys/stat.h>  
#include<fcntl.h>  
#include<stdio.h>  
  
int main(){  
    int fd;  
    char *buf = "19990330";  
  
    fd = open("./file", O_RDWR | O_APPEND);  
  
    printf("open success: fd = %d \n", fd);  
  
    int n_write = write(fd, buf, strlen(buf));  
  
    if(n_write != -1){  
        printf("write %d byte to file\n", n_write);  
    }  
    close(fd);  
    return 0;  
}
```



注意:

```
fd = open("./file", O_RDWR | O_APPEND);
```

4. O_TRUNC 属性去打开文件时, 如果这个文件中本来是有内容的, 而且为只读或者只写 成功打开, 则将其长度截断为 0。

O_TRUNC 属性去打开文件时, 如果这个文件中本来是有内容的, 把原先内容全部覆盖掉, 示例代码:

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

int main(){
    int fd;
    char *buf = "19990330";

    fd = open("./file", O_RDWR | O_TRUNC);
    printf("open success:fd =%d\n", fd);
    int n_write = write(fd, buf, strlen(buf));
    if(n_write != -1){
        printf("write %d byte to file\n", n_write);
    }
    close(fd);
    return 0;
}
```

注意:

`fd = open("./file", O_RDWR | O_TRUNC);`

`ls-l` 是把文件所有内容列出来

```
-rwxr-xr-x 1 CLC book 8478 Fed 9 12:36 mode -rw-r--r-- 1 CLC book
385 Fed 9 12:36 mode.c -rw----- 1 CLC book 0 Fed 9 12:44 file
—表示普通文件 r (4) 可读 w(2)可写 rw 表示可读可写 x(1)执行
```

四、文件写入操作编程

写入文件: `==write==` 头文件: `==#include<unistd.h>==`

`ssize_t write (int fd , const void *buf , size_t count);`

`==//将缓冲区 buf 指针指向内存数据, 写 count 的大小 到 fd 里。==`

(int fd , const void *buf , size_t count) 参数: (文件描述符 , 【无类型指针是一个缓冲区】 , 写入文件的大小)

文件写入操作示例代码:

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

int main()
{
    int fd;
    char *buf = "Refuel.CONG";
    fd = open("./file",O_RDWR);
    printf("fd=%d\n",fd);
    if(fd == -1){
        printf("open file failed\n");
        fd = open("./file",O_RDWR | O_CREAT , 0600);
        if(fd>0){
            printf("creat file success\n");
        }
    }

    //ssize_t write(int fd , const void *buf , size_t count);
    //将缓冲区buf 指针指向内存数据, 写count 大小到fd。
    write(fd,buf,strlen(buf));
    printf("fd=%d\n",fd);
    close(fd); //关闭fd 文件
    return 0;
}
```

strlen(); 函数是用来 真正计算有效字符长度用 strlen 头文件是:
<string.h>

五、文件读取操作

读取文件: ==read== 头文件: ==#include<unistd.h>==

ssize_t read(int fd , const void *buf , size_t count); ==//从 fd 个文件读取 count 个字节数据放到 buf 里==

返回值: 若读取成功, 读多少个字节返回多少个字节, 若读到尾什么都没读到返回 0, 读取失败返回-1

文件读取示例代码:

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

int main()
{
    int fd;
    char *buf = "Refuel.CONG";
    fd= open("./file",O_RDWR);
    printf("fd=%d\n",fd);
    if(fd == -1){
        printf("open file failed\n");
        fd = open("./file", O_RDWR | O_CREAT | ,0600);
        if(fd>0){
            printf("create file success\n");
        }
    }
    int n_write = write(fd,buf,strlen(buf));
    if(n_write != -1){
        printf("write%d byte to file\n",n_write); //若读取成功, 返回读到的 ite
    }
    char *readBuf;
    readBuf = (char*)malloc(sizeof(char)*n_write+1);

    int n_read = read(fd,readBuf , n_write);
    printf("read=%d , context:%s\n",n_read,readBuf);
    close(fd);
    return 0;
}
```



```
CLC@Embed_Learn:~$ ./mode
fd=-1
open file failed
create file success
write 11 byte to file
read=0, context:
```

结果什么都没读取

到: 是因为光标的原因, 光标在写入数据后, 光标停在写完的位置, 读取的时候光标会变后面, 什么也没有所以读取不到。

==解决办法: ==

1. 把光标移动到头。
2. 重新打开文件。

重新打开文件的方式解决光标的问题:

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

int main()
{
    int fd;
    char *buf = "Refuel.CONG";
    fd = open("./file", O_RDWR);
    printf("fd=%d\n", fd);
    if(fd == -1){
        printf("open file failed\n");
        fd = open("./file", O_RDWR | O_CREAT | , 0600);
        if(fd > 0){
            printf("create file success\n");
        }
    }

    int n_write = write(fd, buf, strlen(buf));
    if(n_write != -1){
        printf("write %d byte to file\n", n_write); // 若读取成功, 返回读到的 ite
    }

    close(fd); // 写完数据后关闭文件
    fd = open("./file", O_RDWR); // 重新打开
```

```
char *readBuf;
readBuf = (char*)malloc(sizeof(char)*n_write+1);

int n_read = read(fd,readBuf , n_write);
printf("read=%d , context:%s\n",n_read,readBuf);
close(fd);
return 0;
}
```

```
CLC@Embed_Learn:~$ ./mode
fd=3
write11 byte to file
read=11,context:Refuel.CONG
```

结果为:

六、文件光标移动操作

光标移动: `lseek` 头文件:

1. `#include<sys/types.h>`
2. `#include<unistd.h>`

宏:

1. `SEEK_SET` //指向文件的头
2. `SEEK_CUR` //指向当前光标位置
3. `SEEK_END` //指向文件的尾

`off_t lseek(int fd , off_t offset , int whence)`; 作用: 将文件读写指针相对 `whence` 移动 `offset` 个字节 参数说明: (文件描述符, 偏移值, 固定的位置)

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

```
int main()
{
    int fd;
    char *buf = "Refuel.CONG";
```

```
fd= open("./file",O_RDWR);
printf("fd=%d\n",fd);
if(fd == -1){
    printf("open file failed\n");
    fd = open("./file", O_RDWR | O_CREAT | ,0600);
    if(fd>0){
        printf("create file success\n");
    }
}
int n_write = write(fd,buf,strlen(buf));
if(n_write != -1){
    printf("write%d byte to file\n",n_write);
}
char *readBuf;
readBuf = (char *)malloc(sizeof (char)*n_write+1);

lseek (fd,0,SEEK_SET); //参数: (文件描述, 偏移值, 固定的位置)
//lseek(fd, -11 , SEEK_CUR); //所在光标位置往前偏移11 个

int n_read = read(fd ,readBuf,n_write);
printf("read=%d,context:%s\n",n_write,readBuf);
close(fd);
return 0;
```

lseek (fd,0,SEEK_SET); //参数: (文件描述, 偏移值, 固定的位置)

lseek(fd, -11, SEEK_CUR); //所在光标位置往前偏移 11 个

七、文件操作原理简述

1. 文件描述符:

1. 对于内核而言, 所有打开文件都由文件描述符引用。文件描述符是一个非负整数。当打开一个现存文件或者创建一个新文件时, 内核向进程返回一个文件描述符。当读写一个文件时, 用 open 和 creat 返回的文件描述符标识该文件, 将其作为参数传递给 read 和 write。按照惯例, UNIX shell 使用文件描述符 0 与进程的标准输入相结合, 文件描述符 1 与标准输出相结合, 文件描述符 2 与标准错误输出相结合。STDIN_FILENO、STDOUT_FILENO、STDERR_FILENO 这几个宏代替了 0, 1, 2 这几个数。
2. 文件描述符, 这个数字在一个进程中表示一个特定含义, 当我们 open 一个文件时, 操作系统在内存中构建了一些数据结构来表示这个动态文件, 然后返回给应用程序一个数字作为文件描述符, 这个数字就和我们内存中维护的这个动

态文件的这些数据结构绑定上了, 以后我们应用程序如果要操作这个动态文件, 只需要用这个文件描述符区分。

3. 文件描述符的作用域就是当前进程, 除了这个进程文件描述符就没有意义了, open 函数打开文件, 打开成功返回一个文件描述符, 打开失败, 返回-1。

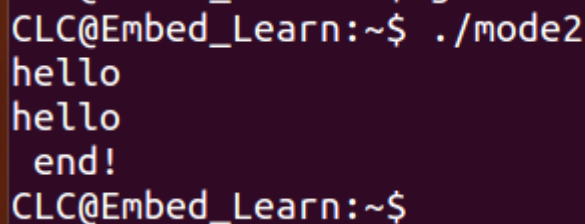
2. linux 系统默认:

标准描述符: 标准输入 (0) 标准输出 (1) 标准错误 (3)

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

int main(){
    int fd;
    char readBuf[128];

    int n_read = read(0,readBuf,5);    // 0 标准输入
    int n_write = write(1,readBuf,strlen(readBuf)); // 1 标准输出
    printf("\n end!\n");
    return 0;
}
```



```
CLC@Embed_Learn:~$ ./mode2
hello
hello
end!
CLC@Embed_Learn:~$
```

在这里插入图片描述

3. 操作文件时候:

打开/创建文件 ——> 读取文件/写入文件 ——> 关闭文件==

- 1、在 Linux 中要操作一个文件, 一般是先 open 打开一个文件, 得到文件描述符, 然后对文件进行读写操作(或其他操作), 最后是 close 关闭文件即可。
- 2、强调一点:我们对文件进行操作时, 一定要先打开文件, 打开成功之后才能操作, 如果打开失败, 就不用进行后边的操作了, 最后读写完成后, 一定要关闭文件, 否则会造成文件损坏。

3、文件平时是存放在块设备中的文件系统文件中的, 我们把这种文件叫**静态文件**, 当我们去 open 打开一个文件时, linux 内核做的操作包括: 内核在进程中建立一个打开文件的数据结构, 记录下我们打开的这个文件; 内核在内存中申请一段内存, 并且将静态文件的内容从块设备中读取到内核中特定地址管理存放(叫**动态文件**)。read write 都是对动态文件进行操作

4、打开文件以后, 以后对这个文件的读写操作, 都是针对内存中的这一份动态文件的, 而并不是针对静态文件的。当然我们对动态文件进行读写以后, 此时内存中动态文件和块设备文件中的静态文件就不同步了, 当我们 close 关闭动态文件名, close 内部内核将内存中的动态文件的内容去更新(同步)块设备中的静态文件。

5、为什么这么设计, 不直接对块设备直接操作。 **块**(假设有 100 字节)设备本身读写非常不灵活, 是按块读写的, 最小只读 100 字节, 而**内存是挤字节单位操作的可而具可以随机操作, 很灵活**。

静态文件 放入磁盘中的文件是静态文件, 如: 桌面上的文件.jpg

动态文件 open 静态文件后, 会在 linux 内核产生结构体记录文件 如: fd 信息节点, buf(内容, 内存)

调用 close 时候, 会把所有信息缓存到磁盘中

八、文件操作编程小练习—实现 cp 指令代码:

==cp (src.c //源文件 , dest.c //复制到的目标文件)==

实现 cp 操作:

1. c 语言参数: ./a.out _ _ ==2. 实现思路: ==

1. 打开 src.c
2. 读 src 到 buf
3. 创建/打开 dest.c
4. 将 buf 写入 dest.c
5. close 两个文件

==c 语言参数: == // 参数 (argc 是数组) (argv 是二级指针是包含数组的数组: 是 argv 里的每一项都是一个数组)

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

```
int main( int argc , char **argv) // 参数 (argc 是数组) (argv 是包含数组的数组: 是 argv 里的每一项都是一个数组)
```

```
{
```

```
    printf("total params:%d\n",argc);
```

```
printf("No.1  params:%d\n",argv[0]);
printf("No.2  params:%d\n",argv[1]);
printf("No.3  params:%d\n",argv[2]);
return 0;
}
```

输入 : ./a.out des src 结果: totol params : 3 No.1 params : ./a.out
No.2 params : des No.3 params : src

实现 mycp 操作代码:

```
#include<sys/types.h>
#include<sys/stat.h>
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<unistd.h>

int main(int argc ,char **argv){
    int fdSrc;
    int fdDest;
    char *readBuf = NULL;
    if(argc != 3){
        printf("param error\n");
        exit(-1);
    }

    fdSrc = open(argv[1],O_RDWR); //1. 打开fdSrc.c (源文件)
    int size = lseek(fdSrc ,0, SEEK_END); //光标记录文件大小
    lseek(fdSrc , 0, SEEK_SET); //!!! 千万记得把 光标回到头
    readBuf = (char *)malloc(sizeof(char) *size+8); //开辟readBuf 空间大小

    int n_read = read(fdSrc,readBuf,size); //2. 读取fdSrc (源文件) 到readBuf 缓冲区

    fdDest = open(argv[2],O_RDWR | O_CREAT | O_TRUNC ,0600); //3. 打开/创建fdDest.c
    int n_write = write(fdDest,readBuf,strlen(readBuf)); //4. 将readbuf 写入到
    fdDest.c

    close(fdSrc); //5. 关闭两个文件
    close(fdDest);
    return 0;
}
```

mycp 容易出现的小问题—实现优化:

1. 用 lseek 来光标计算 size 数组

```
char *readBuf = NULL;

int size = lseek(fdSrc,0,SEEK_END);
lseek(fdSrc , 0 ,SEEK_SET);
readBuf = (char *)malloc(sizeof(char)*size + 8);
```

2. 若要是拷贝大于 1024 的文件就不行

```
int n_read = read(fdSrc,readBuf,size); //读取大小用 lseek 查出的大小.
```

3. 目标文件存在并且存在一些数据, 拷贝就会覆盖了原来数据的一部分
解决方法: O_TRUNC 属性去打开文件时, 如果这个文件中本来是有内容的, 而且为只读或者只写成功打开, 则将其长度截断为 0

```
fdDes = open(argv[2],O_RDWR|O_CREAT|O_TRUNC,0600);
```

九、文件编程小应用之修改程序的配置文件

==(工作中常用)== 配置文件的修改

例如: SPEED=5 LENG=100 SCORE=90 LEVEL=95

修改指定内容的思路:

1. 找到 (要修改的) 位置
2. (修改的位置) 往后移动到 (要改的值)
3. 修改要改的值

找寻修改位置时候, 用到 strstr() 函数; 功能: 用来检索子串在字符串中首次出现的位置

修改 LENG 的值 示例代码:

```
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<unistd.h>
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

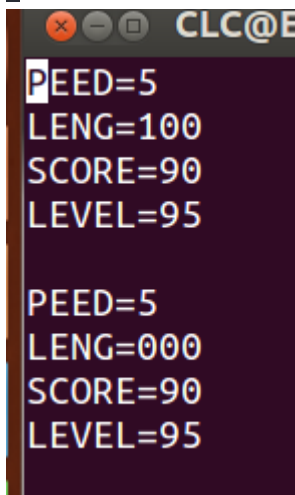
int main(int argc , char **argv){
    int fdSrc;
```

```
char *readBuf=NULL;
if(argc != 2 ){
    printf("param error\n");
    exit(-1);
}
fdSrc = open(argv[1],O_RDWR);
int size = lseek(fdSrc,0,SEEK_END);
lseek (fdSrc,0,SEEK_SET);

readBuf =(char *)malloc(sizeof(char)*size+8);
int n_read= read(fdSrc,readBuf,size);

char *p=strstr(readBuf,"LENG="); //找到 (要修改的) 位置
//参数1 要找的源文件 2.“要找的字符串”
if(p==NULL){
    printf("not found\n");
    exit(-1);
}
p=p+strlen("LENG="); //移动字符串个字节
*p='0'; // *p 取内容

lseek (fdSrc,0,SEEK_SET);
int n_write =write(fdSrc,readBuf,strlen(readBuf));
close(fdSrc);
return 0;
}
```



可以把封装成一个函数:

```
void *changeFile(int fd,char *readbuf,char* f,char t){
    char *p = strstr(readbuf,f);
    if(p == NULL){
        printf("no found\n");
        exit(-1);
    }
}
```



```
    p = p + strlen(f);
    *p = t;
}

int main(int argc, char **argv)
{
    changeFile(fdSrc, readBuf, "LENG=", '6');
}
```

十、写一个整数到文件

`ssize_t write(int fd, const void *buf, size_t count);`
//将缓冲区 buf 指针指向内存数据, 写 count 大小到 fd.
`ssize_t read(int fd, const void *buf, size_t count);`
//从 fd 个文件读取 count 个字节数据放到 buf 里

1. 写入整型数代码:

```
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<unistd.h>
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

int main(){
    int fd;
    int data = 100;
    int data2 = 0;
    fd=open("./flie1",O_RDWR);

    int n_write =write(fd, &data, sizeof(int));
    lseek(fd,0,SEEK_SET);
    int n_read =read(fd, &data2, sizeof(int));

    printf("read %d\n",data2);
    close(fd);
    return 0;
}
```

```
CLC@Embed_Learn:~$ vi file1
CLC@Embed_Learn:~$ ./a.out
read 100
```

在这里插入图片描述

2. 写入结构体代码(1):

```
struct Test
{
    int a;
    char c;
};

int main(){
    int fd;
    int Test data = {100, 'a'};
    int Test data2 ;
    fd=open("./file1", O_RDWR);

    int n_write =write(fd, &data, sizeof(struct Test));
    lseek(fd,0,SEEK_SET);
    int n_read =read(fd, &data2, sizeof(struct Test));

    printf("read %d %s\n",data2.a,data2.c);
    close(fd);
    return 0;
}
```

3. 写入结构体代码(2):

```
struct Test
{
    int a;
    char c;
};

int main(){
    int fd;
    int Test data[2] = {{100, 'a'}, {101, 'b'}};
    int Test data2[2];
    fd=open("./file1", O_RDWR);
```

```
int n_write = write(fd, &data, sizeof(struct Test)*2);
lseek(fd, 0, SEEK_SET);
int n_read = read(fd, &data2, sizeof(struct Test)*2);

printf("read %d %s\n", data2[0].a, data2[0].c);
printf("read %d %s\n", data2[1].a, data2[1].c);
close(fd);
return 0;
}
```

注意写入/读入的 大小: sizeof(struct Test) 乘 2

缓冲区可以写入 : 整数, 字符, 结构体等

十一、标准 C 库对文件操作引入

1. open 和 fopen 的区别

1. 来源

- open 是 UNIX 系统调用函数, 返回的是文件描述符, 它是文件在文件描述符表里的索引。
- fopen 是 ANSI C 标准中的 C 语言库函数, 在不同的系统中应该调用不同的内核 API, 返回值是一个指向文件结构的指针。

2. 移植性

这一点从上面的来源就可以推断出来, fopen 是 C 标准函数, 因此用有良好的移植性; 而 open 是 UNIX 系统调用, 移植性有限。如 windows 下相似的功能使用 API 函数。

3. 适用范围

open 返回文件描述符, 而文件描述符是 UNIX 系统下的一个重要概念, UNIX 下的一切设备都是文件的形式操作, 如网络套接字, 硬件设备 (驱动) 等。当然包括操作普通正规文件。fopen 是用来操纵普通正规文件的。

4. 缓冲

1. 缓冲文件系统

缓冲文件系统的特点是: 在内存开辟一个“缓冲区”, 为程序中的每一个文件使用; 当执行读文件的操作时, 从磁盘文件将数据先读入内存“缓冲区”, 装满后再从内存“缓冲区”依此读出需要的数据。执行写文件的操作时, 先将数据写入内存“缓冲区”, 待内存缓冲区”装满后再写入文件。由此可以看出, 内存“缓冲区”的大小, 影响着实际操作外存的

次数, 内存“缓冲区”越大, 则操作外存的次数就少, 执行速度就快、效率高。一般来说, 文件“缓冲区”的大小随机器而定。fopen, fclose, fread, fwrite, fgetc, fgets. fputc, fputs, freopen, fseek. ftell, rewind 等。

2. 非缓冲文件系统

缓冲文件系统是借助文件结构体指针来对文件进行管理, 通过文件指针来对文件进行访问, 既可以读写字符、字符电、格式化数据, 也可以读写二进制数据。非缓冲文件系统依赖于操作系统, 通过操作系统的功能对文件进行读写, 是系统级的输入输出, 它不设文件结构体指针, 只能读写二进制文件, 但效率高、速度快, 由于 ANSI 标准不再包括非缓冲文件系统, 因此建议大家最好不要选择它。open, close, read, write, getc, getchar, putc, putchar 等。

一句话总结一下, 就是 open 无缓冲, fopen 有缓冲。前者与 read, write 等配合使用, 后者与 freadfwrite 等配合使用

使用 fopen 函数, 由于在用户态下就有了缓冲, 因此进行文件读写操作的时候就减少了用户态和内核态的切换(切换到内核态调用还是需要调用系统调用 API:read, write);而使用 open 函数, 在文件读写时则每次都需要进行内核态和用户态的切换;表现为, 如果顺序访问文件, fopen 系列的函数要比直接调用 open 系列的函数快;如果随机访问文件则相反。这样一总结梳理, 相信大家对于两个函数及系列函数有了一个更全面清晰的认识, 也应该知道在什么场合下使用什么样的函数更合适 效率更高。

2. fopen(); fwrite(); fread(); 方式写入数据

==FILE *fopen (const char *path ,const char *mode);==

参数说明: path : 路径 mode :用什么方式打开 返回值: FILE 类型

mode 打开模式:

模式指令	功能说明
r	只读方式打开一个文本文件
rb	只读方式打开一个二进制文件
w	只写方式打开一个文本文件
wb	只写方式打开一个二进制文件

模式指令	功能说明
a	追加方式打开一个文本文件
ab	追加方式打开一个二进制文件
r+	可读可写方式打开一个文本文件
rb+	可读可写方式打开一个二进制文件
w+	可读可写方式创建一个文本文件
wb+	可读可写方式生成一个二进制文件
a+	可读可写追加方式打开一个文本文件
ab+	可读可写方式追加一个二进制文件

写入: `==size_t fwrite (const void *ptr , size_t size , size_t nmemb , FILE *stream);==`

1. ptr 缓冲区 等同于 (buf)
2. size 一个字符大小 (sizeof char)
3. nmemb 个数
4. stream (哪个文件) which file

读取: `==size_t fread (const void *ptr , size_t size , size_t nmemb , FILE *stream);==`

光标问题: `==int fseek (FILE *stream , long offset , int whence);==`

示例代码:

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

int main()
{
    FILE *fp;
    char *str = "Refuel.CONG";
    char readBuf[128]={0};

    fp = fopen("./CONG.txt","w+"); //可读可写方式创建一个文本文件
    fwrite(str,sizeof(char),strlen(str),fp);
    //一次性写一个char 写str个字节,到fp里
    fseek(fp,0,SEEK_SET);
    fread(readBuf,sizeof(char),strlen(str),fp);
    //从fp里一次读一个char 读str个读到readBuf里去
    printf("read data:%s\n",readBuf);
    fclose(fp);
}
```

```
return 0;
}
```

```
CLC@Embed_Learn:~$ gcc fopen.c
CLC@Embed_Learn:~$ ./a.out
read data:Refuel.CONG
```

```
CLC@Embed_Learn: ~
Refuel.CONG
```

也可改写: fread (readBuf , sizeof(char) , * strlen(str) , 1 , fp);
//读*strlen (str) 个 读 1 次

3. n_read 和 n_write 的返回值

n_read 和 n_write 的返回值取决于第三个参数==

```
int n_fwrite = fwrite(str sizeof(char)*strlen(str),1,fp);
int n_fread = fread(str sizeof(char)*strlen(str),1,fp);

printf("read=%d,write=%d\n",n_read,n_write);
```

结果: n_read= 1 n_write =1

4. n_fread 和 n_fwrite 返回值区别

```
int n_fread = fread(str sizeof(char)*strlen(str),100,fp);
printf("n_read=%d\n",n_read);
```

结果为 : n_read = 1

但是如果写 100 结果就会不同

```
int n_fwrite = fwrite(str sizeof(char)*strlen(str),100,fp);
printf("n_write=%d\n",n_write);
```

结果为 : n_write= 100

5. 标准 c 库写入结构体到文件

fwrite() 写入结构体代码:

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

struct Test
{
    int a;
    char c;
};

int main()
{
    FILE *fp;
    struct Test data1 = {1100, 'a'};
    struct Test data2;

    fp = fopen("./CONG.txt", "w+"); // 可读可写方式创建一个文本文件
    int n_fwrite = fwrite(&data1, sizeof(struct Test), 1, fp);

    fseek(fp, 0, SEEK_SET);
    int n_fread = fread(&data2, sizeof(struct Test), 1, fp);

    printf("read = %d, %s\n", data2.a, data2.c);
    fclose(fp);
    return 0;
}
```

结果: read = 1100 , a

6. fgetc(); fputc(); feof() 的使用方法;

==读字符函数: fgetc() 函数的用法== 作用: 从指定的文件中读一个字符, 函数调用的形式为: char ch//字符变量 = fgetc(fp // 文件指针); 我们可以将读取到的数据给到一个字符变量存储。

1. 面对要读取的数据繁多的情况, 为了减少程序运行的时间复杂度, 要用到 ==fgets()==
2. fgets(); 功能是从指定的文件读取一个字符串到字符数组中。 函数的调用形式为: fgets(字符数组名, n, 文件指针); 参数: n 为一个正整数。表示从文件中读出的字符串不超过 n-1 个字符, 在读入的最后一个字符后加上字符串结束标志'0', 说通俗易懂点就是读多少?

3.

```
printf("%s\n", str); // 循环读取所有数据 while(fgets(str, 100, fp)){
// 循环读取所有数据, 直到 fgets 读到 '\0'
printf("%s\n", str); } fclose(fp); return 0 ; ````
```

==fputc() 函数的用法==

int fputc(int c , FILE *stream); 功能: 把 c 写入 文件 stream 里

==feof() 函数的用法==

int feof (FILE *stream); 作用: (是否到尾巴的位置): 测试在没到达文件尾巴返回 0, 到达尾巴返回 非 0

feof() 函数使用代码:

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

int main(){
    FILE *fp;
    int i;
    char c;
    fp = fopen("./test.txt","r");
    //没到达文件尾巴返回0, 到达尾巴返回非0
    while(!feof(fp)){
        //没到尾巴时返回0, 取反 进入 while 当到达尾巴返回非0 取反=0 退出 while
        c=fgetc(fp);
        printf("%c",c);
    }
    fclose(fp);
    return 0;
}
```

fputc 写入文件代码:

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

int main(){
    FILE *fp;
    int i;
    char *str = "Refuel.CONG";
    int len = sizeof(str);

    fp = fopen("./test.txt","w+");
    if(fp == NULL){
        printf("打开文件出现错误\n");
        exit(-1);
    }
    for(i=0;i<len ;i++){
        fputc(*str,fp);
        str++;
    }
}
```



```
fclose(fp);  
return 0;  
}
```

文件编程就到这里结束了!!! 下面用思维导图的方式总结一下:

十二、 文件编程篇思维导图总结:

公众号:一口Linux

