本节重点

> - 复习C文件IO相关操作
> - 认识文件相关系统调用接口
> - 认识文件描述符,理解重定向
> - 对比fd和FILE，理解系统调用和库函数的关系
> - 理解文件系统中inode的概念
> - 认识软硬链接，对比区别
> - 认识动态静态库，学会结合gcc选项，制作动静态库

先来段代码回顾C文件接口

hello.c写文件

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

int main()
{
    FILE *fp = fopen("myfile", "w");
    if(!fp){
        printf("fopen error!\n");
    }

    const char *msg = "hello bit!\n";
    int count = 5;
    while(count--){
        fwrite(msg, strlen(msg), 1, fp);
    }

    fclose(fp);

    return 0;
}
```

hello.c读文件

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

int main()
{
    FILE *fp = fopen("myfile", "r");
    if(!fp){
        printf("fopen error!\n");
    }

    char buf[1024];
    const char *msg = "hello bit!\n";
```

```
    while(1){
        //注意返回值和参数，此处有坑，仔细查看man手册关于该函数的说明
        ssize_t s = fread(buf, 1, strlen(msg), fp);
        if(s > 0){
            buf[s] = 0;
            printf("%s", buf);
        }
        if(feof(fp)){
            break;
        }
    }

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

输出信息到显示器，你有哪些方法

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

int main()
{
    const char *msg = "hello fwrite\n";
    fwrite(msg, strlen(msg), 1, stdout);

    printf("hello printf\n");
    fprintf(stdout, "hello fprintf\n");
    return 0;
}
```

stdin & stdout & stderr

> - C默认会打开三个输入输出流，分别是stdin, stdout, stderr
> - 仔细观察发现，这三个流的类型都是FILE*, fopen返回值类型，文件指针

总结

> - 打开文件的方式

```
       r      Open text file for reading.
              The stream is positioned at the beginning of the file.

       r+     Open for reading and writing.
              The stream is positioned at the beginning of the file.

       w      Truncate(缩短) file to zero length or create text file for writing.
              The stream is positioned at the beginning of the file.


       w+     Open for reading and writing.
```

```
              The file is created if it does not exist, otherwise it is truncated.
              The stream  is  positioned at the beginning of the file.

       a      Open for appending (writing at end of file).
              The file is created if it does not exist.
              The stream is positioned at the end of the file.

       a+     Open for reading and appending (writing at end of file).
              The file is created if it does not exist.  The initial file  position
              for reading is at the beginning of the file,
              but output is always appended to the end of the file.
```

如上，是我们之前学的文件相关操作。还有`fseek` `ftell` `rewind`的函数，在C部分已经有所涉猎，请同学们自行复习。

系统文件I/O

操作文件，除了上述C接口（当然，C++也有接口，其他语言也有），我们还可以采用系统接口来进行文件访问，先来直接以代码的形式，实现和上面一模一样的代码：

hello.c 写文件:

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

int main()
{
    umask(0);
    int fd = open("myfile", O_WRONLY|O_CREAT, 0644);
    if(fd < 0){
        perror("open");
        return 1;
    }

    int count = 5;
    const char *msg = "hello bit!\n";
    int len = strlen(msg);

    while(count--){
        write(fd, msg, len);//fd: 后面讲， msg: 缓冲区首地址， len: 本次读取，期望写入多少个字节的数据。 返回值: 实际写了多少字节数据
    }

    close(fd);
    return 0;
}
```

hello.c读文件

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

int main()
{
    int fd = open("myfile", O_RDONLY);
    if(fd < 0){
        perror("open");
        return 1;
    }

    const char *msg = "hello bit!\n";
    char buf[1024];
    while(1){
        ssize_t s = read(fd, buf, strlen(msg));//类比write
        if(s > 0){
            printf("%s", buf);
        }else{
            break;
        }
    }

    close(fd);
    return 0;
}
```

接口介绍

open `man open`

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

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);

pathname: 要打开或创建的目标文件
flags: 打开文件时，可以传入多个参数选项，用下面的一个或者多个常量进行“或”运算，构成flags。
参数:
        O_RDONLY: 只读打开
        O_WRONLY: 只写打开
        O_RDWR  : 读，写打开
                  这三个常量，必须指定一个且只能指定一个
        O_CREAT : 若文件不存在，则创建它。需要使用mode选项，来指明新文件的访问权限
        O_APPEND: 追加写
```

```
返回值：
        成功：新打开的文件描述符
        失败：-1
```

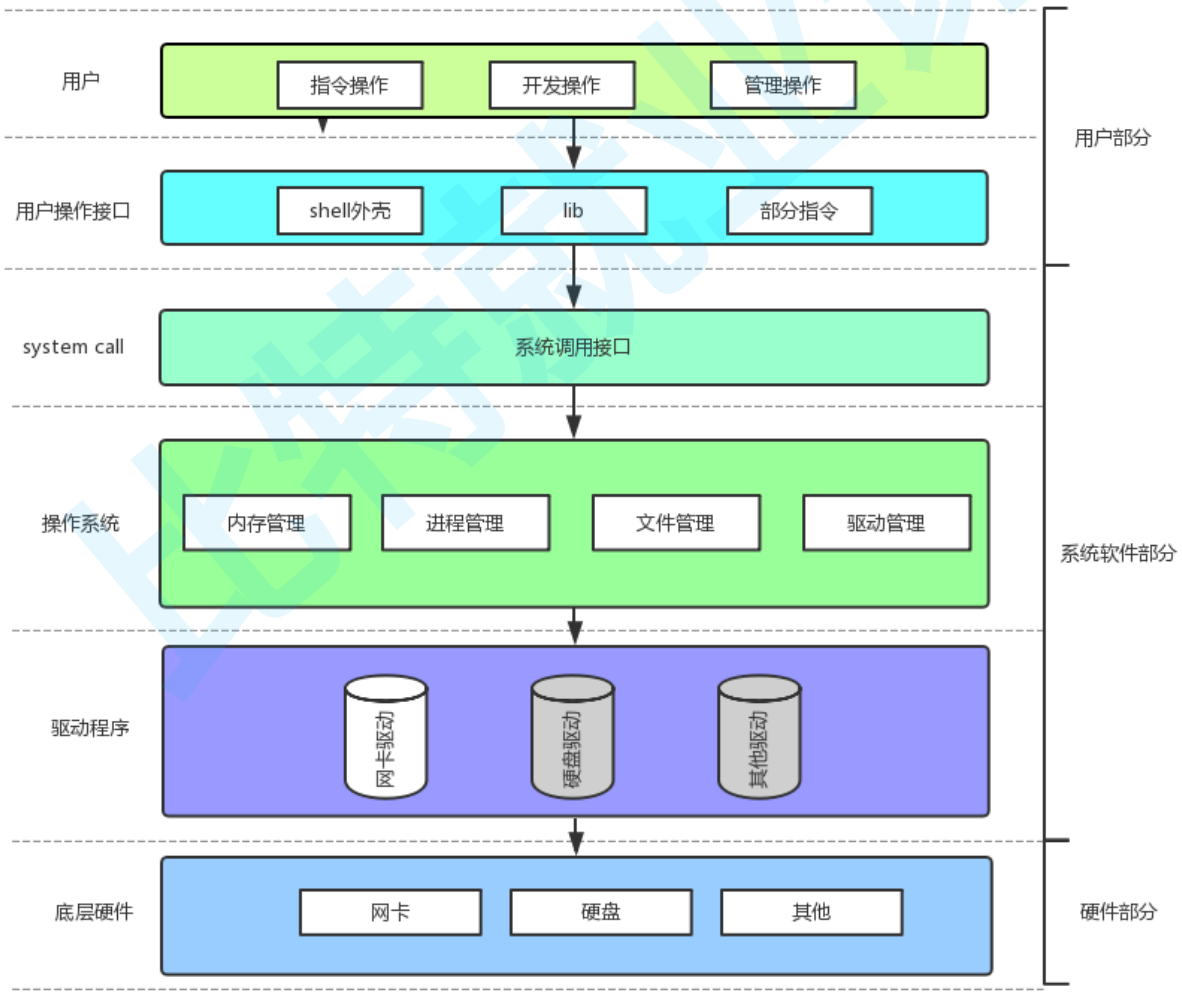
mode_t理解：直接`man`手册，比什么都清楚。
`open`函数具体使用哪个，和具体应用场景相关，如目标文件不存在，需要open创建，则第三个参数表示创建文件的默认权限,否则，使用两个参数的open。

`write` `read` `close` `lseek`,类比C文件相关接口。

open函数返回值

在认识返回值之前，先来认识一下两个概念: `系统调用`和`库函数`

> - 上面的`fopen` `fclose` `fread` `fwrite`都是C标准库当中的函数，我们称之为库函数（libc）。
> - 而，`open` `close` `read` `write` `lseek`都属于系统提供的接口，称之为系统调用接口
> - 回忆一下我们讲操作系统概念时，画的一张图



系统调用接口和库函数的关系，一目了然。
所以，可以认为，f#系列的函数，都是对系统调用的封装，方便二次开发。

文件描述符fd

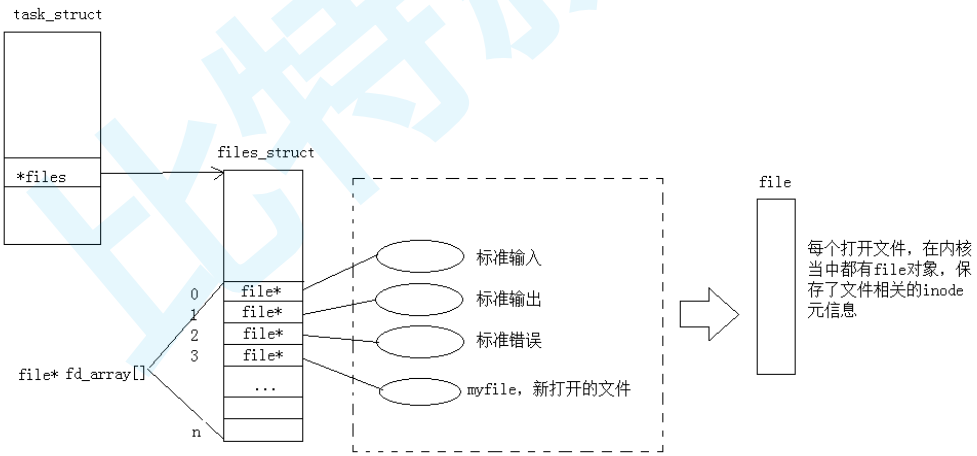
> - 通过对open函数的学习，我们知道了文件描述符就是一个小整数

0 & 1 & 2

> - Linux进程默认情况下会有3个缺省打开的文件描述符，分别是标准输入0， 标准输出1， 标准错误2.
> - 0,1,2对应的物理设备一般是：键盘，显示器，显示器
>
> 所以输入输出还可以采用如下方式：

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

int main()
{
    char buf[1024];
    ssize_t s = read(0, buf, sizeof(buf));
    if(s > 0){
        buf[s] = 0;
        write(1, buf, strlen(buf));
        write(2, buf, strlen(buf));
    }
    return 0;
}
```



而现在知道，文件描述符就是从0开始的小整数。当我们打开文件时，操作系统在内存中要创建相应的数据结构来描述目标文件。于是就有了file结构体。表示一个已经打开的文件对象。而进程执行open系统调用，所以必须让进程和文件关联起来。每个进程都有一个指针*files, 指向一张表files_struct,该表最重要的部分就是包涵一个指针数组，每个元素都是一个指向打开文件的指针！所以，本质上，文件描述符就是该数组的下标。所以，只要拿着文件描述符，就可以找到对应的文件

文件描述符的分配规则

直接看代码：

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

int main()
{
    int fd = open("myfile", O_RDONLY);
    if(fd < 0){
        perror("open");
        return 1;
    }
    printf("fd: %d\n", fd);

    close(fd);
    return 0;
}
```

输出发现是 `fd: 3`

关闭0或者2，在看

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

int main()
{
     close(0);
     //close(2);
    int fd = open("myfile", O_RDONLY);
    if(fd < 0){
        perror("open");
        return 1;
    }
    printf("fd: %d\n", fd);

    close(fd);
    return 0;
}
```

发现是结果是：`fd: 0`或者`fd 2`可见，文件描述符的分配规则：在files_struct数组当中，找到当前没有被使用的最小的一个下标，作为新的文件描述符。

重定向

那如果关闭1呢？看代码：

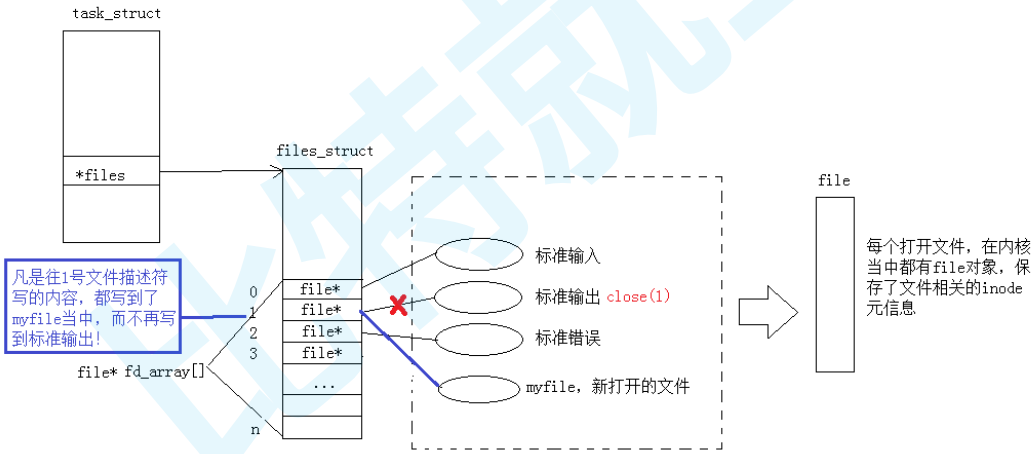
```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>

int main()
{
    close(1);
   int fd = open("myfile", O_WRONLY|O_CREAT, 00644);
   if(fd < 0){
       perror("open");
       return 1;
   }
    printf("fd: %d\n", fd);
    fflush(stdout);

   close(fd);
    exit(0);
}
```

此时，我们发现，本来应该输出到显示器上的内容，输出到了文件 `myfile` 当中，其中，fd = 1。这种现象叫做输出重定向。常见的重定向有:>, >>, <

那重定向的本质是什么呢?



使用 dup2 系统调用

函数原型如下:

```
#include <unistd.h>

int dup2(int oldfd, int newfd);
```

示例代码

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

int main() {
  int fd = open("./log", O_CREAT | O_RDWR);
  if (fd < 0) {
    perror("open");
    return 1;
  }
  close(1);
  dup2(fd, 1);
  for (;;) {
    char buf[1024] = {0};
    ssize_t read_size = read(0, buf, sizeof(buf) - 1);
    if (read_size < 0) {
      perror("read");
      break;
    }
    printf("%s", buf);
    fflush(stdout);
  }
  return 0;
}
```

例子1. 在minishell中添加重定向功能：

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

# define MAX_CMD 1024
char command[MAX_CMD];

int do_face()
{
    memset(command, 0x00, MAX_CMD);
    printf("minishell$ ");
    fflush(stdout);
    if (scanf("%[^\n]%*c", command) == 0) {
        getchar();
        return -1;
    }
    return 0;
}
char **do_parse(char *buff)
{
    int argc = 0;
    static char *argv[32];

    char *ptr = buff;
```

```
    while(*ptr != '\0') {
        if (!isspace(*ptr)) {
            argv[argc++] = ptr;
            while((!isspace(*ptr)) && (*ptr) != '\0') {
                ptr++;
            }
        }else {
            while(isspace(*ptr)) {
                *ptr = '\0';
                ptr++;
            }
        }
    }
    argv[argc] = NULL;
    return argv;
}
int do_redirect(char *buff)
{
    char *ptr = buff, *file = NULL;
    int type = 0, fd, redirect_type = -1;
    while(*ptr != '\0') {
        if (*ptr == '>') {
            *ptr++ = '\0';
            redirect_type++;
            if (*ptr == '>') {
                *ptr++ = '\0';
                redirect_type++;
            }
            while(isspace(*ptr)) {
                ptr++;
            }
            file = ptr;
            while((!isspace(*ptr)) && *ptr != '\0') {
                ptr++;
            }
            *ptr = '\0';
            if (redirect_type == 0) {
                fd = open(file, O_CREAT|O_TRUNC|O_WRONLY, 0664);
            }else {
                fd = open(file, O_CREAT|O_APPEND|O_WRONLY, 0664);
            }
            dup2(fd, 1);
        }
        ptr++;
    }
    return 0;
}
int do_exec(char *buff)
{
    char **argv = {NULL};


    int pid = fork();
```

```
    if (pid == 0) {
        do_redirect(buff);
        argv = do_parse(buff);
        if (argv[0] == NULL) {
            exit(-1);
        }
        execvp(argv[0], argv);
    }else {
        waitpid(pid, NULL, 0);
    }

    return 0;
}

int main(int argc, char *argv[])
{
    while(1) {
        if (do_face() < 0)
            continue;
        do_exec(command);
    }
    return 0;
}
```

printf是C库当中的IO函数，一般往`stdout`中输出，但是stdout底层访问文件的时候，找的还是fd:1, 但此时，fd:1下标所表示内容，已经变成了myfile的地址，不再是显示器文件的地址，所以，输出的任何消息都会往文件中写入，进而完成输出重定向。那追加和输入重定向如何完成呢？请同学们自行研究。

FILE

> - 因为IO相关函数与系统调用接口对应，并且库函数封装系统调用，所以本质上，访问文件都是通过fd访问的。
> - 所以C库当中的FILE结构体内部，必定封装了fd。

来段代码在研究一下：

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

int main()
{
    const char *msg0="hello printf\n";
    const char *msg1="hello fwrite\n";
    const char *msg2="hello write\n";

    printf("%s", msg0);
    fwrite(msg1, strlen(msg0), 1, stdout);
    write(1, msg2, strlen(msg2));

    fork();

    return 0;
```

```
}
```

运行出结果：

```
hello printf
hello fwrite
hello write
```

但如果对进程实现输出重定向呢？`./hello > file`， 我们发现结果变成了：

```
hello write
hello printf
hello fwrite
hello printf
hello fwrite
```

我们发现`printf`和`fwrite`（库函数）都输出了2次，而`write`只输出了一次（系统调用）。为什么呢？肯定和fork有关！

> - 一般C库函数写入文件时是全缓冲的，而写入显示器是行缓冲。
> - `printf` `fwrite`库函数会自带缓冲区（进度条例子就可以说明），当发生重定向到普通文件时，数据的缓冲方式由行缓冲变成了全缓冲。
> - 而我们放在缓冲区中的数据，就不会被立即刷新，甚至fork之后
> - 但是进程退出之后，会统一刷新，写入文件当中。
> - 但是fork的时候，父子数据会发生写时拷贝，所以当你父进程准备刷新的时候，子进程也就有了同样的一份数据，随即产生两份数据。
> - `write`没有变化，说明没有所谓的缓冲。

综上：`printf` `fwrite`库函数会自带缓冲区，而`write`系统调用没有带缓冲区。另外，我们这里所说的缓冲区，都是用户级缓冲区。其实为了提升整机性能，OS也会提供相关内核级缓冲区，不过不再我们讨论范围之内。
那这个缓冲区谁提供呢？`printf` `fwrite`是库函数，`write`是系统调用，库函数在系统调用的“上层”， 是对系统调用的“封装”，但是`write`没有缓冲区，而`printf` `fwrite`有，足以说明，该缓冲区是二次加上的，又因为是C，所以由C标准库提供。

如果有兴趣，可以看看FILE结构体:

`typedef struct _IO_FILE FILE; 在/usr/include/stdio.h`

```
在/usr/include/libio.h
struct _IO_FILE {
  int _flags;       /* High-order word is _IO_MAGIC; rest is flags. */
#define _IO_file_flags _flags

  //缓冲区相关
  /* The following pointers correspond to the C++ streambuf protocol. */
  /* Note:  Tk uses the _IO_read_ptr and _IO_read_end fields directly. */
  char* _IO_read_ptr;   /* Current read pointer */
  char* _IO_read_end;   /* End of get area. */
  char* _IO_read_base;  /* Start of putback+get area. */

  char* _IO_write_base; /* Start of put area. */
```

```
  char* _IO_write_ptr;  /* Current put pointer. */
  char* _IO_write_end;  /* End of put area. */
  char* _IO_buf_base;   /* Start of reserve area. */
  char* _IO_buf_end;    /* End of reserve area. */
  /* The following fields are used to support backing up and undo. */
  char *_IO_save_base; /* Pointer to start of non-current get area. */
  char *_IO_backup_base;  /* Pointer to first valid character of backup area */
  char *_IO_save_end; /* Pointer to end of non-current get area. */

  struct _IO_marker *_markers;

  struct _IO_FILE *_chain;

  int _fileno; //封装的文件描述符
#if 0
  int _blksize;
#else
  int _flags2;
#endif
 _IO_off_t _old_offset; /* This used to be _offset but it's too small.  */

#define __HAVE_COLUMN /* temporary */
  /* 1+column number of pbase(); 0 is unknown. */
  unsigned short _cur_column;
  signed char _vtable_offset;
  char _shortbuf[1];

  /*  char* _save_gptr;  char* _save_egptr; */

  _IO_lock_t *_lock;
#ifdef _IO_USE_OLD_IO_FILE
};
```

理解文件系统

我们使用ls -l的时候看到的除了看到文件名，还看到了文件元数据。

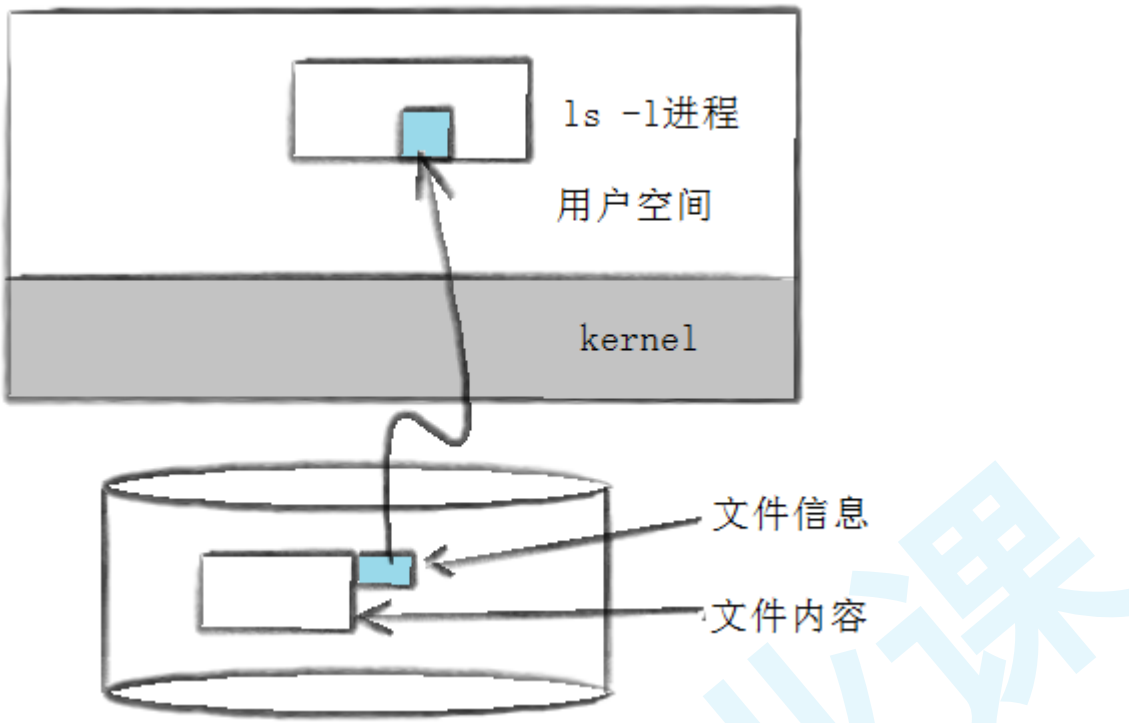
```
[root@localhost linux]# ls -l
总用量 12
-rwxr-xr-x. 1 root root 7438 "9月  13 14:56" a.out
-rw-r--r--. 1 root root  654 "9月  13 14:56" test.c
```

每行包含7列：

> - 模式
> - 硬链接数
> - 文件所有者
> - 组
> - 大小
> - 最后修改时间

- 文件名

ls -l读取存储在磁盘上的文件信息，然后显示出来



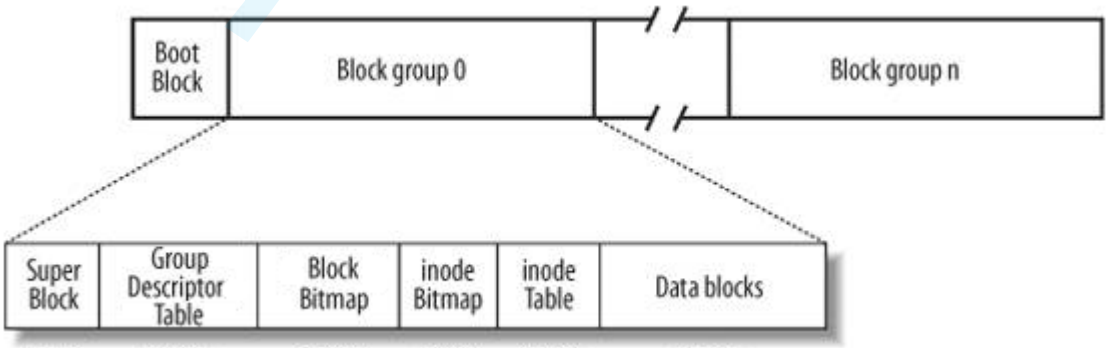
其实这个信息除了通过这种方式来读取，还有一个stat命令能够看到更多信息

```
[root@localhost linux]# stat test.c
  File: "test.c"
  Size: 654         Blocks: 8          IO Block: 4096   普通文件
Device: 802h/2050d  Inode: 263715      Links: 1
Access: (0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2017-09-13 14:56:57.059012947 +0800
Modify: 2017-09-13 14:56:40.067012944 +0800
Change: 2017-09-13 14:56:40.069012948 +0800
```

上面的执行结果有几个信息需要解释清楚

inode

为了能解释清楚inode我们先简单了解一下文件系统



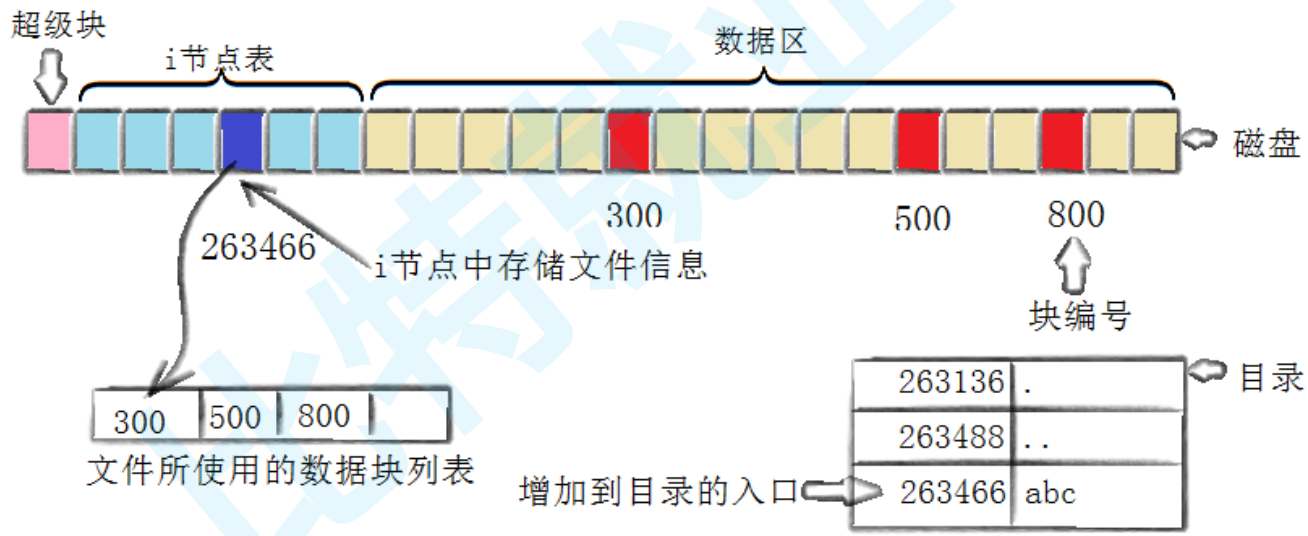
Linux ext2文件系统，上图为磁盘文件系统图（内核内存映像肯定有所不同），磁盘是典型的块设备，硬盘分区被划分为一个个的block。一个block的大小是由格式化的时候确定的，并且不可以更改。例如mke2fs的-b选项可以设定block大小为1024、2048或4096字节。而上图中启动块（Boot Block）的大小是确定的，

- Block Group：ext2文件系统会根据分区的大小划分为数个Block Group。而每个Block Group都有着相同的结构组成。政府管理各区的例子
- 超级块（Super Block）：存放文件系统本身的结构信息。记录的信息主要有：bolck 和 inode的总量，未使用的block和inode的数量，一个block和inode的大小，最近一次挂载的时间，最近一次写入数据的时间，最近一次检验磁盘的时间等其他文件系统的相关信息。Super Block的信息被破坏，可以说整个文件系统结构就被破坏了
- GDT，Group Descriptor Table：块组描述符，描述块组属性信息，有兴趣的同学可以在了解一下
- 块位图（Block Bitmap）：Block Bitmap中记录着Data Block中哪个数据块已经被占用，哪个数据块没有被占用
- inode位图（inode Bitmap）：每个bit表示一个inode是否空闲可用。
- i节点表:存放文件属性 如 文件大小，所有者，最近修改时间等
- 数据区：存放文件内容

将属性和数据分开存放的想法看起来很简单，但实际上是如何工作的呢？我们通过touch一个新文件来看看如何工作。

```
[root@localhost linux]# touch abc
[root@localhost linux]# ls -i abc
263466 abc
```

为了说明问题，我们将上图简化：



创建一个新文件主要有一下4个操作：

1. 存储属性
   内核先找到一个空闲的i节点（这里是263466）。内核把文件信息记录到其中。
2. 存储数据
   该文件需要存储在三个磁盘块，内核找到了三个空闲块：300,500，800。将内核缓冲区的第一块数据复制到300，下一块复制到500，以此类推。
3. 记录分配情况
   文件内容按顺序300,500,800存放。内核在inode上的磁盘分布区记录了上述块列表。
4. 添加文件名到目录

   新的文件名abc。linux如何在当前的目录中记录这个文件？内核将入口（263466，abc）添加到目录文件。文件名和inode之间的对应关系将文件名和文件的内容及属性连接起来。

理解硬链接

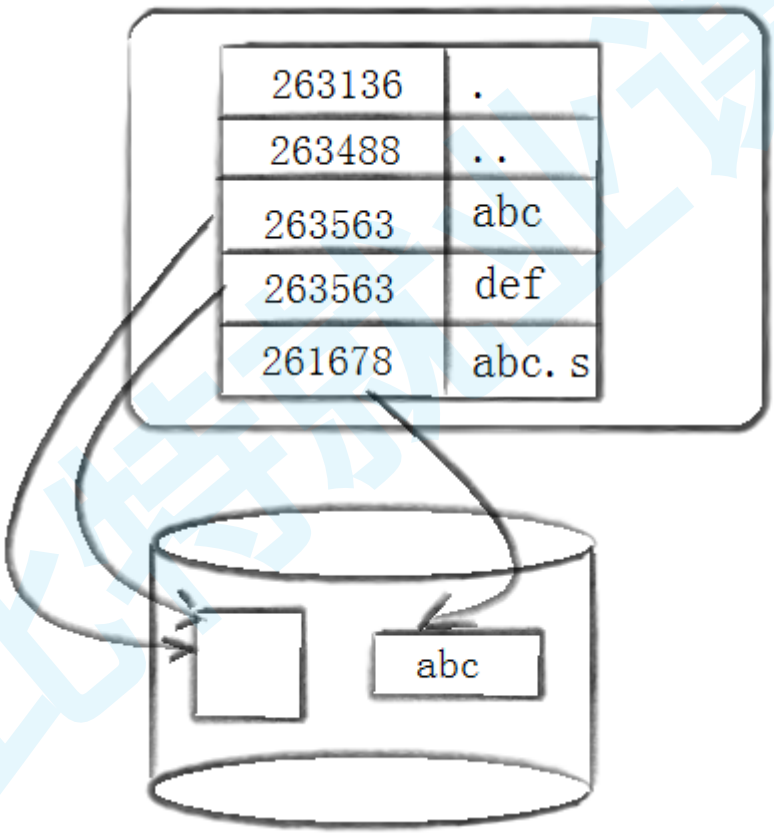
我们看到，真正找到磁盘上文件的并不是文件名，而是inode。 其实在linux中可以让多个文件名对应于同一个inode。 [root@localhost linux]# touch abc [root@localhost linux]# ln abc def [root@localhost linux]# ls -1i abc def 263466 abc 263466 def

> - abc和def的链接状态完全相同，他们被称为指向文件的硬链接。内核记录了这个连接数，inode 263466 的硬连接数为2。
> - 我们在删除文件时干了两件事情：1.在目录中将对应的记录删除，2.将硬连接数-1，如果为0，则将对应的磁盘释放。

软链接

硬链接是通过inode引用另外一个文件，软链接是通过名字引用另外一个文件，在shell中的做法

```
263563 -rw-r--r--. 2 root root 0 9月  15 17:45 abc
261678 lrwxrwxrwx. 1 root root 3 9月  15 17:53 abc.s -> abc
263563 -rw-r--r--. 2 root root 0 9月  15 17:45 def
```



acm

下面解释一下文件的三个时间：

> - Access 最后访问时间
> - Modify 文件内容最后修改时间
> - Change 属性最后修改时间

动态库和静态库

静态库与动态库

- 静态库（.a）：程序在编译链接的时候把库的代码链接到可执行文件中。程序运行的时候将不再需要静态库
- 动态库（.so）：程序在运行的时候才去链接动态库的代码，多个程序共享使用库的代码。
- 一个与动态库链接的可执行文件仅仅包含它用到的函数入口地址的一个表，而不是外部函数所在目标文件的整个机器码
- 在可执行文件开始运行以前，外部函数的机器码由操作系统从磁盘上的该动态库中复制到内存中，这个过程称为动态链接（dynamic linking）
- 动态库可以在多个程序间共享，所以动态链接使得可执行文件更小，节省了磁盘空间。操作系统采用虚拟内存机制允许物理内存中的一份动态库被要用到该库的所有进程共用，节省了内存和磁盘空间。

测试程序

```
    //////////////add.h//////////////////
    #ifndef __ADD_H__
    #define __ADD_H__
    int add(int a, int b);
    #endif // __ADD_H__
    //////////////add.c//////////////////
    #include "add.h"
    int add(int a, int b)
    {
        return a + b;
    }

    //////////////sub.h//////////////////
    #ifndef __SUB_H__
    #define __SUB_H__
    int sub(int a, int b);
    #endif // __SUB_H__
    //////////////add.c//////////////////
    #include "add.h"
    int sub(int a, int b)
    {
        return a - b;
    }

    ///////////main.c/////////////////
    #include <stdio.h>
    #include "add.h"
    #include "sub.h"

    int main( void )
    {
        int a = 10;
        int b = 20;
        printf("add(10, 20)=%d\n", a, b, add(a, b));
        a = 100;
        b = 20;
        printf("sub(%d,%d)=%d\n", a, b, sub(a, b));
    }
```

生成静态库

```
[root@localhost linux]# ls
add.c  add.h  main.c  sub.c  sub.h
[root@localhost linux]# gcc -c add.c -o add.o
[root@localhost linux]# gcc -c sub.c -o sub.o

生成静态库
[root@localhost linux]# ar -rc libmymath.a add.o sub.o
ar是gnu归档工具，rc表示(replace and create)

查看静态库中的目录列表
[root@localhost linux]# ar -tv libmymath.a
rw-r--r-- 0/0   1240 Sep 15 16:53 2017 add.o
rw-r--r-- 0/0   1240 Sep 15 16:53 2017 sub.o
t:列出静态库中的文件
v:verbose 详细信息

[root@localhost linux]# gcc main.c -L. -lmymath

-L 指定库路径
-l 指定库名
测试目标文件生成后，静态库删掉，程序照样可以运行。
```

库搜索路径

- 从左到右搜索-L指定的目录。
- 由环境变量指定的目录 （LIBRARY_PATH）
- 由系统指定的目录
  - /usr/lib
  - /usr/local/lib

生成动态库

- shared: 表示生成共享库格式
- fPIC：产生位置无关码(position independent code)
- 库名规则：libxxx.so

示例： [root@localhost linux]# gcc -fPIC -c sub.c add.c [root@localhost linux]# gcc -shared -o libmymath.so *.o [root@localhost linux]# ls add.c add.h add.o libmymath.so main.c sub.c sub.h sub.o

使用动态库

编译选项

- l：链接动态库，只要库名即可(去掉lib以及版本号)
- L：链接库所在的路径.

示例： gcc main.o -o main –L. -lhello

运行动态库

- 1、拷贝.so文件到系统共享库路径下, 一般指/usr/lib
- 2、更改`LD_LIBRARY_PATH`

```
[root@localhost linux]# export LD_LIBRARY_PATH=.
[root@localhost linux]# gcc main.c -lmymath
[root@localhost linux]# ./a.out
add(10, 20)=30
sub(100, 20)=80
```

> - 3、ldconfig 配置/etc/ld.so.conf.d/，ldconfig更新

```
[root@localhost linux]# cat /etc/ld.so.conf.d/bit.conf
/root/tools/linux
[root@localhost linux]# ldconfig
```

使用外部库

系统中其实有很多库，它们通常由一组互相关联的用来完成某项常见工作的函数构成。比如用来处理屏幕显示情况的函数（ncurses库）

```
#include <math.h>
#include <stdio.h>
int main(void)
{
    double x = pow(2.0, 3.0);
    printf("The cubed is %f\n", x);
    return 0;
}
gcc -Wall calc.c -o calc -lm
```

-lm表示要链接libm.so或者libm.a库文件

库文件名称和引入库的名称

如：libc.so -> c库，去掉前缀lib，去掉后缀.so,.a