4 Linux文件操作

4.0 Linux的文件

在UNIX系统当中，“万物皆文件”是一种重要的设计思想。在传统的定义当中，我们把**存储在磁盘当中的数据集合**称为文件。文件这个概念在UNIX的设计当中进一步泛化，所有满足**速度较慢、容量较大**和可以**持久化存储**中（任意一个）特征的数据集合都可以称为**文件**。因此，在UNIX/Linux当中，对目录和设备的操作都属于文件操作。

根据文件存在的形式，文件分为普通文件、目录文件、链接文件和设备文件等类型：

- 普通文件：也称磁盘文件，并且能够进行随机的数据存储(能够自由使用lseek或者fseek定位到某一个位置);
- 管道：是一个从一端发送数据，另一端接收数据的数据通道;
- 目录：也称为目录文件，它包含了保存在目录中文件列表的简单文件;
- 设备：该类型的文件提供了大多数物理设备的接口。它又分为两种类型：**字符设备**和**块设备**。字符设备一次只能读出和写入一个字节的数据，包括终端、打印机、声卡以及鼠标；块设备必须以一定大小的块来读出或者写入数据，块设备包括CD-ROM、RAM驱动器和磁盘驱动器等。一般而言，**字符设备用于传输数据，块设备用于存储数据**;
- 链接：类似于Windows的快捷方式，指包含到达另一个文件路径的文件。

4.1 基于文件流的文件操作

4.1.1文件的创建，打开与关闭

使用文件流/文件指针来访问文件的方法是由标准C规定的，相关函数的原型为：

```
#include <stdio.h> //头文件包含
FILE* fopen(const char* path, const char* mode);//文件名 模式
int fclose(FILE* stream);
```

- fopen以mode的方式打开或创建文件，如果成功，将返回一个文件指针，失败则返回NULL。fopen创建的文件的访问权限将以0666与当前的umask结合来确定。
- mode的可选模式列表，如下所示：

模式	读	写	位置	截断原内容	创建
rb	Y	N	文件头	N	N
rb+	Y	Y	文件头	N	N
wb	N	Y	文件头	Y	Y
wb+	Y	Y	文件头	Y	Y
ab	N	Y	文件尾	N	Y
ab+	Y	Y	文件尾	N	Y

- 在Linux系统中,mode里面的‘b’(二进制)可以去掉，但是为了保持与其他系统的兼容性，建议不要去掉
- ab和ab+为追加模式，在此两种模式下，在一开始的时候读取文件内容是从文件起始处开始读取的，而无论文件读写点定位到何处，在写数据时都将是在文件末尾添加（写完以后读写点就移动到文件末尾了），所以比较适合于多进程写同一个文件的情况下保证数据的完整性。

4.1.2 读写文件

简单地说，文件的读写操作就是把数据在外部设备（磁盘、键盘、屏幕...）和内存之间进行数据交互， 以便进程直接访问这些数据或者是将数据持久化存储到外部设备中。基于文件流的数据读写函数较多，可分为如下几组：

数据块读写：

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

- fread从文件流stream 中读取nmemb个元素，写到ptr指向的内存中，每个元素的大小为size个字节
- fwrite从ptr指向的内存中读取nmemb个元素，写到文件流stream中，每个元素的大小为size个字节

- 所有的文件读写函数都从文件的当前读写点开始读写，读写完成以后，当前读写点自动往后移动size*nmemb个字节。

格式化读写：

```
#include <stdio.h>
int printf(const char *format, ...);
//相当于fprintf(stdout,format,…);
int scanf(const char *format, …);
int fprintf(FILE *stream, const char *format, ...);
int fscanf(FILE *stream, const char *format, …);
int sprintf(char *str, const char *format, ...);
//eg:sprintf(buf,"the string is;%s",str);
int sscanf(char *str, const char *format, …);
```

- fprintf将格式化后的字符串写入到文件流stream中
- sprintf将格式化后的字符串写入到字符串str中

单个字符读写：

- 使用下列函数可以一次读写一个字符

```
#include <stdio.h>
int fgetc(FILE *stream);
int fputc(int c, FILE *stream);
int getc(FILE *stream);//等同于 fgetc(FILE* stream)
int putc(int c, FILE *stream);//等同于 fputc(int c, FILE* stream)
int getchar(void);//等同于 fgetc(stdin);
int putchar(int c);//等同于 fputc(int c, stdout);
```

- getchar和putchar从标准输入输出流中读写数据，其他函数从文件流stream中读写数据

字符串读写：

```
char *fgets(char *s, int size, FILE *stream);
int fputs(const char *s, FILE *stream);
int puts(const char *s);//等同于 fputs(const char *s,stdout);
char *gets(char *s);//等同于 fgets(const char *s, int size, stdin);
```

- fgets和fputs从文件流stream中读写一行数据;
- puts和gets从标准输入输出流中读写一行数据。
- fgets可以指定目标缓冲区的大小，所以相对于gets安全，但是fgets调用时，如果文件中当前行的字符个数大于size，则下一次fgets调用时，将继续读取该行剩下的字符，fgets读取一行字符时，保留行尾的换行符。
- fputs不会在行尾自动添加换行符，但是puts会在标准输出流中自动添加一换行符。

4.1.3 文件定位

- 文件定位指读取或设置文件当前读写点，所有的通过文件指针读写数据的函数，都是从文件的当前读写点读写数据的。常用的函数有：

```
#include <stdio.h>
int feof(FILE * stream);
//通常的用法为while(!feof(fp))
int fseek(FILE *stream, long offset, int whence);
//设置当前读写点到偏移whence 长度为offset处
long ftell(FILE *stream);
//用来获得文件流当前的读写位置
void rewind(FILE *stream);
//把文件流的读写位置移至文件开头 fseek(fp, 0, SEEK_SET);
```

- fseek设置当前读写点到偏移whence 长度为offset处，whence可以是：SEEK_SET (文件开头)、SEEK_CUR (文件当前位置)、SEEK_END (文件末尾)
- ftell获取当前的读写点
- rewind将文件当前读写点移动到文件头

4.1.4 文本文件和二进制文件

如果文件当中的内容是一串ASCII字符的序列，那么这类文件就是**文本文件**。对于文本类型的文件，如果使用`fread`读取到内存当中，那么就应该分配字符串类型的内存区域以存储数据，假若想要根据文件的内容得到相关类型的数据，就需要使用`fscanf`函数。

```
// 假如文件里面的内容是文本的100000
// echo -n 100000 > file1
int main(int argc, char *argv[])
{
    FILE * fp = fopen("file1","r+");
    // 从文本内容读到字符串数据
    char buf[1024] = {0};
    fread(buf,1,1024,fp);
    printf("buf = %s\n", buf);
    // 从文本内容读到int数据
    fseek(fp,0,SEEK_SET);
    int i;
    fscanf(fp,"%d", &i);
    printf("i = %d\n", i);
    return 0;
}
```

如果文件内容是内存数据块按字节为单位直接进行存储，那么该文件就是**二进制文件**。对于二进制文件，目前只能使用`fread`和`fwrite`进行读写操作。对应二进制文件的读写，需要遵循一个基本原则就是：**写入是什么类型，读取就使用同样的类型**。

4.2 目录操作

接下来我们所介绍的函数都是系统调用，也就是说，它们是调用操作系统内核来实现相关的功能。这些系统调用一般都遵循POSIX规范，但是通常无法在Windows环境下调用。

首先要介绍的系统调用是`chmod`，该系统调用可以修改文件权限。

```
#include <sys/stat.h>
int chmod(const char* path, mode_t mode);
//mode形如：0777 是一个八进制整型
//path参数指定的文件被修改为具有mode参数给出的访问权限。
```

4.2.1 当前工作目录

当前工作目录是进程的属性之一。当进程以相对路径的方式访问文件时，当前工作目录是就是该文件在文件系统路径当中的起点。

```
#include <unistd.h>  //头文件
char *getcwd(char *buf, size_t size);  //获取当前目录，相当于pwd命令
int chdir(const char *path);        //修改当前目录，即切换目录，相当于cd命令
```

- getcwd()函数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，参数size为buf的空间大小。因此在调用此函数时，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，则回值NULL，errno的值则为ERANGE
- 倘若参数buf为NULL，getcwd()会依参数size的大小自动配置内存(使用malloc())，如果参数size也为0，则getcwd()会依工作目录绝对路径的字符串程度来决定所配置的内存大小。用户可以在使用完此字符串后，自行利用free()来释放此空间。所以常用的形式：

```
getcwd(NULL, 0);
```

- chdir()函数：将本进程的当前工作目录属性改成参数path所指的目录

```
#include<unistd.h>
int main()
{
    chdir("/tmp");
    printf("current working directory: %s\n",getcwd(NULL,0));
}
```

4.2.2 创建和删除目录

```
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
int mkdir(const char *pathname, mode_t mode);  //创建目录,mode是目录权限
int rmdir(const char *pathname);          //删除目录
```

- 如果需要查看库函数mkdir而不是shell命令mkdir，使用man命令的时候需要指定系统库函数帮助手册（也就是mkdir(2)）
- 如果函数的（指针）参数使用const来进行修饰，这个参数就称为传入参数。这意味在函数的内部是不能够通过指针来修改传入参数的内容

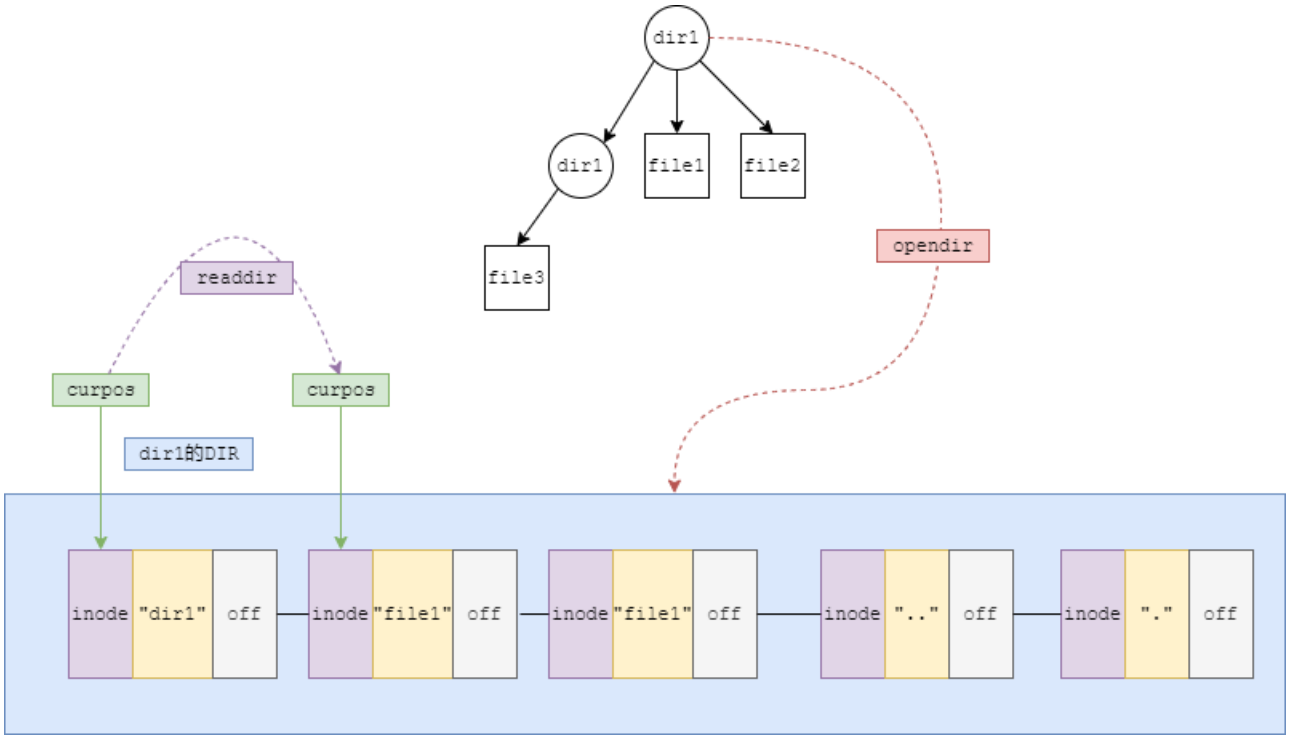
```
$ man 2 mkdir
```

4.2.3 目录的存储原理

对于任何文件，为了快速定位文件在磁盘当中的位置，文件系统在设计的时候就需要利用专门的索引结构来管理所有文件。索引结构的基本单位是索引结点，每个索引结点的具有固定的大小，里面存放了单个文件的位置、文件类型、权限、修改时间等等信息。

在Linux当中，目录是一种特别的文件，它的总体大小固定。它的数据块当中把很多文件的文件名和索引结点存放在一起。因为文件的文件名大小不一，为了避免磁盘碎片和支持频繁增加修改，所以目录采用链表来存储来组织各种文件的信息，链表的结点称为目录项(dirent)。目录项的定义如下，可以看出，要访问下个dirent结点，实际是依赖于本结点中d_off属性。

```
struct dirent{
    ino_t d_ino;        //该文件的inode
    off_t d_off;        //到下一个dirent的偏移
    unsigned short d_reclen;//文件名长度
    unsigned char d_type;  //所指的文件类型
    char  d_name[256];   //文件名
};
```



4.2.4 目录流相关操作

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *name);    //打开一个目录流
struct dirent *readdir(DIR *dir);  //读取目录流当中的一个目录项，
void rewinddir(DIR *dir);       //重新定位到目录文件的头部
void seekdir(DIR *dir,off_t offset);//用来设置目录流目前的读取位置
off_t telldir(DIR *dir);         //返回目录流当前的读取位置
int closedir(DIR *dir);        //关闭目录文件
```

使用`opendir`系统调用可以在内存中创建**目录流**数据结构用来处理和目录相关的操作。目录流的底层设计是一种链表结构，其中链表的结点就是目录项。此外，目录流当中还存在一个当前读写位置指针，每一次使用`readdir`函数之后，会取得读写位置指针所指向的目录项，并且读写位置指针会自动后移。使用`closedir`函数可以关闭目录流。

> *dirent当中采用了类似于变长数组的形式来存放文件名，但是会提供一些冗余的空间，这样当调整文件名的时候，如果新文件名的长度不超过原来分配的空间，则不需要调整分配的空间*

> *类似于地址和内存的关系，inode（索引结点）描述了文件在磁盘上的具体位置信息。在ls命令中添加-i参数可查看文件的inode信息。那么所谓的硬链接，就是指inode相同的文件。一个inode的结点上的硬链接的个数就称为引用计数*

```
$ ls -ial
查看所有文件的inode信息
$ ln 当前文件 目标
建立名为“目标”的硬链接
```

> - *这里可以检查.文件引用计数，发现是2。这是因为一个目录可以通过本目录或者是父目录来访问它本身，若它还有子目录，它的引用数也会增加*
> - *删除磁盘上文件的时候，只有引用计数为0时候，才会将磁盘内容移除文件系统（断开和目录的链接）*
> - *当然，为了避免引用死锁，一般用户是不能使用ln命令来为目录建立硬链接、下面是一个使用目录流的例子，它实现了类似于tree命令的效果：*

```
//使用深度优先遍历访问目录的例子
#include <func.h>
int printDir(char *path)
{
    DIR* pdir = opendir(path);
    ERROR_CHECK(pdir,NULL,"opendir");
    struct dirent *pdirent;
    char buf[1024];//注意递归时传递的路径是否合理
    while((pdirent = readdir(pdir)))
    {
        if(strcmp(pdirent->d_name,".") == 0 || strcmp(pdirent-
>d_name,"..") == 0)
        {
            continue;
        }
        printf("%s\n",pdirent->d_name);
        sprintf(buf,"%s%s%s",path,"/",pdirent->d_name);//这里不需要担
心斜杠太多的问题
        if(pdirent->d_type == 4)
        {
            printDir(buf);
        }
```

```
    }
    closedir(pdir);
    return 0;
}
int tabPrintDir(char *path,int width)//这里实现了类似tree的效果
{
    DIR* pdir = opendir(path);
    ERROR_CHECK(pdir,NULL,"opendir");
    struct dirent *pdirent;
    char buf[1024];
    while((pdirent = readdir(pdir)))
    {
        if(strcmp(pdirent->d_name,".") == 0 || strcmp(pdirent-
>d_name,"..") == 0)
        {
            continue;
        }
        printf("%*s%s\n",width,"",pdirent->d_name);
        sprintf(buf,"%s%s%s",path,"/",pdirent->d_name);
        if(pdirent->d_type == 4)
        {
            tabPrintDir(buf,width+4);
        }
    }
    closedir(pdir);
    return 0;
}

int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    puts(argv[1]);
    printDir(argv[1]);
    tabPrintDir(argv[1],0);
    return 0;
}
```

`seekdir`用来设置目录流目前的读取位置，再调用`readdir`函数时，便可以从此新位置开始读取。参数offset代表距离目录文件开头的偏移量。使用`readddir`时，如果已经读取到目录末尾，又想重新开始读，则可以使用`rewinddir`函数将文件指针重新定位到目录文件的起始位置。`telldir`函数用来返回目录流当前的读取位置

```
//下面是一个例子
#include <func.h>

int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    DIR* pdir = opendir(argv[1]);
    ERROR_CHECK(pdir,NULL,"opendir");
    struct dirent *pdirent;
    off_t pos;
    while((pdirent = readdir(pdir)))
    {
        printf("ino = %ld len = %d type = %d filename =
%s\n",pdirent->d_ino,pdirent->d_reclen,pdirent->d_type,pdirent-
>d_name);
        if(strcmp(pdirent->d_name,"a.out") == 0)
        {
            pos = telldir(pdir);//只会成功，不会失败
        }
    }
    //seekdir(pdir,pos);
    rewinddir(pdir);
    pdirent = readdir(pdir);
    printf("-------------------------\n");
    printf("ino = %ld len = %d type = %d filename = %s\n",pdirent-
>d_ino,pdirent->d_reclen,pdirent->d_type,pdirent->d_name);
    return 0;
}
```

使用`stat`系统调用可以获取文件的详细信息，相关的信息存入stat结构体当中。

```
// man 2 stat
#include <sys/types.h>
```

```c
#include <sys/stat.h>
#include <unistd.h>
int stat(const char *pathname, struct stat *statbuf);
//结构体stat的定义
  struct stat {
    dev_t     st_dev;   /*如果是设备，返回设备表述符，否则为0*/
    ino_t     st_ino;   /* i节点号 */
    mode_t    st_mode;   /* 文件类型 */
    nlink_t    st_nlink;  /* 链接数 */
    uid_t     st_uid;   /* 属主ID */
    gid_t     st_gid;   /* 组ID */
    dev_t     st_rdev;   /* 设备类型*/
    off_t     st_size;   /* 文件大小，字节表示 */
    blksize_t   st_blksize; /* 块大小*/
    blkcnt_t   st_blocks;  /* 块数 */
    time_t    st_atime;  /* 最后访问时间*/
    time_t    st_mtime;  /* 最后修改时间*/
    time_t    st_ctime;  /* 最后权限修改时间 */
};
```

下面的例子实现了一个类似于ls -al命令的效果：

```c
//示例
#include <func.h>
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    int ret;
    struct stat buf;
    ret = stat(argv[1],&buf);
    ERROR_CHECK(ret,-1,"stat");

    printf("%x %ld %s %s %ld
%s\n",buf.st_mode,buf.st_nlink,getpwuid(buf.st_uid)-
>pw_name,getgrgid(buf.st_gid)-
>gr_name,buf.st_size,ctime(&buf.st_mtime));
    return 0;
}
```

> - *注意getpwuid和getgrgid需要包含头文件pwd.h和grp.h*

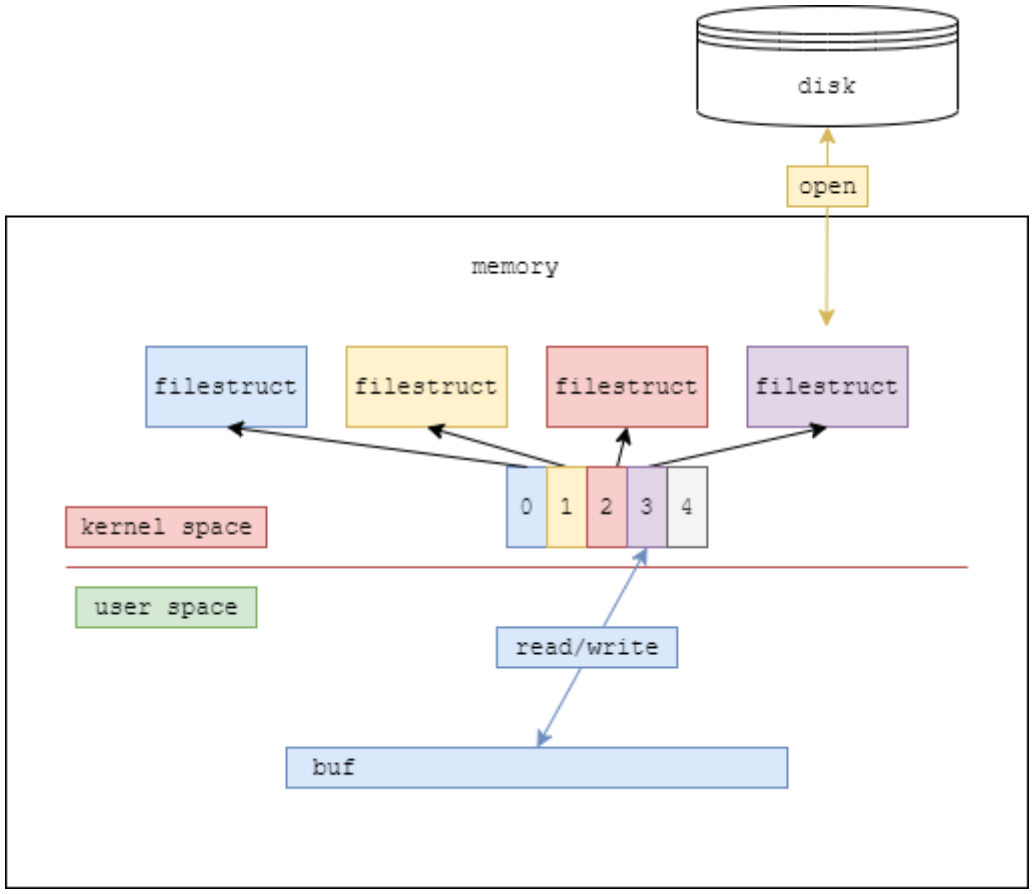
4.3 基于文件描述符的文件操作

4.3.1 文件描述符

之前所讨论的文件操作都是利用了数据结构FILE。FILE就是文件流，也称为用户态文件缓冲区，所以与FILE类型相关的文件操作（比如fopen，fread等等）称为**带缓冲的IO**，它们是ISO C的组成部分。

POSIX标准支持另一类**无缓冲的IO**。在这里，无缓冲是没有分配**用户态文件缓冲区**的意思。在操作文件时，进程会在内存空间的内核区部分里面会维护一个数据结构来管理和文件相关的所有操作，这个数据结构称为打开文件或者是**文件对象**（filestruct），除此以外，内核区里面还会维护一个索引数组来管理所有的文件对象，该数组的下标就被称为**文件描述符**(file descriptor)。

从类型来说，文件描述符是一个非负整数，它可以传递给用户。用户在获得文件描述符之后可以定位到相关联的文件对象，从而可以执行各种IO操作。



4.3.2 打开、创建和关闭文件

使用open函数可以打开一个已存在的问题或者创建一个新文件，并创建一个文件对象，返回相关的文件描述符。

```
#include <sys/types.h>  //头文件
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags);   //文件名 打开方式
int open(const char *pathname, int flags, mode_t mode);//文件名 打开方式 权限
```

flags的数据类型是int，但是在这里并非当作一个普通整数来使用，用户需要把这个int看成是32个bit的集合，不同的位用来描述不同的信息。比如最低位的两个bit就用来描述读写权限。

```
#define O_RDONLY         00
#define O_WRONLY         01
#define O_RDWR           02
```

由上可知，如果需要同时组合两个无关权限，那么就可以采用按位或的操作。

常见的flags的可选项如下：

选项	含义
O_RDONLY	以只读的方式打开
O_WRONLY	以只写的方式打开
O_RDWR	以读写的方式打开
O_CREAT	如果文件不存在，则创建文件
O_EXCL	仅与O_CREAT连用，如果文件已存在，则open失败
O_TRUNC	如果文件存在，将文件的长度截至0
O_APPEND	已追加的方式打开文件，每次调用write时，文件指针自动先移到文件尾，用于多进程写同一个文件的情况。
O_NONBLOCK O_NDELAY	对管道、设备文件和socket使用，以非阻塞方式打开文件，无论有无数据读取或等待，都会立即返回进程之中

使用完文件以后，要记得使用`close`来关闭文件。一旦调用`close`，会使文件的打开引用计数减1，只有文件的打开引用计数变为0以后，文件才会被真正的关闭。

```
int close(int fd);//fd表示文件描述词,是先前由open或creat创建文件时的返回值。
```

当flags选项当中存在O_CREAT选项时，`open`就应该选择3参数的版本，其中第三个参数mode说明了要创建的文件的权限，通常会直接填入一个八进制的整型字面值，例如：

```
int fd = open("file",O_RDWR | O_CREAT,0755); //表示给755的权限
if(-1 == fd)
{
    perror("open failed!\n");
    exit(-1);
}
```

下面是一个简单的打开文件的例子：

```c
int main(int argc, char *argv[])
{
    ARGS_CHECKargc,2);
    int fd = open(argv[1],O_RDONLY);
    ERROR_CHECK(fd,-1,"open");
    printf("fd = %d\n", fd);
    close(fd);
    return 0;
}
```

在使用`open`系统调用的时候，内核会按照最小可用的原则分配一个文件描述符。一般情况下， 进程一经启动就会打开3个文件对象，占用了0、1和2文件描述符，分别关联了标准输入、标准输出和标准错误输出，所以此时再打开的文件占用的文件描述符就是3。

4.3.3 读写文件

使用`read`和`write`系统调用可以读写文件，它们统称为不带缓冲的IO。`read`的原理是将数据从文件对象内部的内核文件缓冲区拷贝出来（这部分的数据最初是在外部设备中，通过硬件的IO操作拷贝到内存之上）到用户态的buf之中。而`write`就相反，它将数据从用户态的buf当中拷贝到内核区的文件对象的内核文件缓冲区，并最终会写入到设备中。

类似于FILE当中的读写行为，每次调用`read`或者`write`会影响其下次读写位置，值得注意的是，这个读写位置信息是保存在内核文件对象当中的。

```c
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);//文件描述符 缓冲区 长度
ssize_t write(int fd, const void *buf, size_t count);
```

下面是一个简单的使用`read`/`write`的例子：

```c
//读取文件内容
#include <func.h>
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    int fd = open(argv[1],O_RDWR);
```

```
    ERROR_CHECK(fd,-1,"open");
    printf("fd = %d\n",fd);
    char buf[128] = {0};
    int ret = read(fd, buf, sizeof(buf));
    printf("buf = %s, ret = %d\n", buf, ret);
    close(fd);
    return 0;
}

//写入文件描述符
#include <func.h>
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    int fd = open(argv[1],O_RDWR);
    ERROR_CHECK(fd,-1,"open");
    printf("fd = %d\n",fd);
    int ret = write(fd, &fd, sizeof(fd));
    printf("ret = %d\n",ret);
    close(fd);
    return 0;
}
//如果希望查看文件的2进制信息，在vim中输入命令:%!xxd
//使用od -h 命令也可查看文件的16进制形式
```

值得注意的是，使用`read`读取磁盘文件和设备文件的行为是有着明显差异的：

- 当读取磁盘文件/设备文件的时候，如果文件/设备缓冲区剩余内容长度超过count，那么本次`read`会读取count个字节；
- 当读取磁盘文件/设备文件的时候，如果文件/设备缓冲区存在剩余内容，且长度不足count，那么本次`read`会读取文件剩余内容；
- 当读取磁盘文件的时候，如果文件无剩余内容，那么本次`read`操作会立刻返回，返回值为0；
- 当读取设备文件的时候，如果设备缓冲区无剩余内容，那么本次`read`会阻塞。

```c
int main()
{
    char buf[1024] = {0};
    ssize_t sret = read(0,buf,sizeof(buf)-1);
    ERROR_CHECK(sret,-1,"read");
    printf("sret = %ld, buf = %s\n", sret, buf);
    return 0;
}
```

通过组合`read`和`write`，可以实现类似cp命令的效果：

```c
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,3);
    int fdr = open(argv[1],O_RDONLY);
    ERROR_CHECK(fdr,-1,"open fdr");
    int fdw = open(argv[2],O_WRONLY|O_CREAT, 0666);
    ERROR_CHECK(fdw,-1,"open fdw");
    char buf[4096] = {0}; // char不是字符的意思，希望用长度为1字节的数据
    while(1){
        memset(buf,0,sizeof(buf));//好习惯 每次read之前先memset
        ssize_t sret = read(fdr,buf,sizeof(buf));
        if(sret == 0){
            break;
        }
        write(fdw,buf,sret);
    }
    close(fdw);
    close(fdr);
    return 0;
}
```

在实现cp命令从read的效率问题：

使用不带缓冲IO的时候，CPU需要陷入内核态来处理文件读取。如果频繁地使用read来读取少量数据，数据的读取效率会比较低

4.3.4 截断文件

使用ftruncate函数可以截断文件，从而控制文件的大小。

```
#include <unistd.h>
int ftruncate(int fd, off_t length);
```

- 函数ftruncate会将参数fd指定的文件大小改为参数length指定的大小。参数fd为已打开的文件描述词，而且必须是以**写入模式**打开的文件。如果原来的文件大小比参数length大，则超过的部分会被删去（实际上修改了文件的inode信息）。执行成功则返回0，失败返回-1

实例：

```
//示例
#include <func.h>
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    int fd = open(argv[1],O_WRONLY);
    ERROR_CHECK(fd,-1,"open");
    printf("fd = %d\n",fd);
    int ret = ftruncate(fd,3);
    ERROR_CHECK(ret,-1,"ftruncate");
    return 0;
}
```

- 使用mmap函数经常配合函数ftruncate来扩大文件大小

```
//示例
#include <func.h>
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    int fd = open(argv[1],O_RDWR);
    ERROR_CHECK(fd,-1,"open");
    printf("fd = %d\n",fd);
    ftruncate(fd,5)
```

```
    char *p;
    p = (char *)mmap(NULL,5,PROT_READ|PROT_WRITE,MAP_SHARED,fd,0);
    ERROR_CHECK(p,(char *)-1,"mmap");
    p[5] = 0;
    printf("%s\n",p);
    p[0] = 'H';
    munmap(p,5);

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

4.3.5 文件映射

使用mmap接口可以实现直接将一个磁盘文件映射到存储空间的一个缓冲区上面，无需使用read和write进行IO

```
#include <sys/mman.h>
void *mmap(void *adr, size_t len, int prot, int flag, int fd, off_t
off);
```

addr参数用于指定映射存储区的起始地址。这里设置为NULL，这样就由系统自动分配（通常是在堆空间里面寻找）。fd参数是一个文件描述符，使用时必须要已经打开。prot参数用来表示权限。PROT_READ,PROT_WRITE表示可读可写。flag参数在目前是采用MAP_SHARED，后面讲解进程通信的时候会介绍其他类型

为什么mmap需要和ftruncate联合使用？因为分配的缓冲区的大小和偏移量的大小是有限制的，它必须是虚拟内存页大小的整数倍。如果文件大小较小，那么超过文件大小返回的缓冲区操作将不会修改文件；如果文件大小为0，还会出现Bus error

4.3.6 文件定位

函数lseek将文件指针设定到相对于whence，偏移值为offset的位置。它的返回值是读写点距离文件开始的距离

```c
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);//fd文件描述词
//whence 可以是下面三个常量的一个
//SEEK_SET 从文件头开始计算
//SEEK_CUR 从当前指针开始计算
//SEEK_END 从文件尾开始计算
```

```c
#include <func.h>
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    int fd = open(argv[1],O_RDWR);
    ERROR_CHECK(fd,-1,"open");
    int ret = lseek(fd, 5, SEEK_SET);
    printf("pos = %d\n", ret);
    char buf[128] = {0};
    read(fd, buf, sizeof(buf));
    printf("buf = %s\n", buf);
    close(fd);
    return 0;
}
```

利用该函数可以实现文件空洞：对一个新建的空文件，可以定位到偏移文件开头1024个字节的地方，在写入一个字符，则相当于给该文件分配了1025个字节的空间，形成文件空洞。通常用于多进程间通信的时候的共享内存。（在某些文件系统的实现中，这些空洞甚至不会占用磁盘空间）

```c
#include <func.h>
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    int fd = open(argv[1],O_RDWR);
    ERROR_CHECK(fd,-1,"open");
    int ret = lseek(fd, 1024, SEEK_SET);
    write(fd, "a", 1);
    close(fd);
    return 0;
}
```

4.3.7 获取文件信息

可以通过fstat和stat函数获取文件信息，调用完毕后，文件信息被填充到结构体struct stat变量中，函数原型为：

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int stat(const char *file_name, struct stat *buf);   //文件名 stat结构体指针
int fstat(int fd, struct stat *buf);      //文件描述词  stat结构体指针
```

- 对于结构体的成员st_mode，有一组宏可以进行文件类型的判断

宏	描述
S_ISLNK(mode)	判断是否是符号链接
S_ISREG(mode)	判断是否是普通文件
S_ISDIR(mode)	判断是否是目录
S_ISCHR(mode)	判断是否是字符型设备
S_ISBLK(mode)	判断是否是块设备
S_ISFIFO(mode)	判断是否是命名管道
S_ISSOCK(mode)	判断是否是套接字

```c
//获得文件的大小
```

```c
#include<sys/stat.h>
#include<unistd.h>
int main(int argc, char *argv[])
{
    struct stat buf;
    stat (argv[1],&buf);
    printf("file size = %d\n",buf.st_size);//st_size可以得到文件大小
}
//如果用fstat函数实现，如下：
//int fd = open ("/etc/passwd",O_RDONLY); //先获得文件描述词
//fstat(fd, &buf);
#include <func.h>
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    int fd = open(argv[1],O_RDWR|__O_PATH);

    ERROR_CHECK(fd,-1,"open");
    struct stat buf;
    int ret = fstat(fd, &buf);
    ERROR_CHECK(ret, -1, "fstat");
    if(S_ISDIR(buf.st_mode))
    {
        printf("directory\n");
    }
    else if(S_ISREG(buf.st_mode))
    {
        printf("regular file\n");
    }
    else if(S_ISLNK(buf.st_mode))
    {
        printf("link file\n");
    }
    printf("the size of file is: %ld\n",buf.st_size);
    close(fd);
    return 0;
}
```

4.3.8 文件描述符的复制

系统调用函数dup和dup2可以实现文件描述符的复制

dup返回一个新的文件描述符（是自动分配的，数值是没有使用的文件描述符的最小编号）。这个新的描述符是旧文件描述符的拷贝。这意味着两个描述符共享同一个数据结构

dup2允许调用者用一个有效描述符(oldfd)和目标描述符(newfd)。函数成功返回时，目标描述符将变成旧描述符的复制品，此时两个文件描述符现在都指向同一个文件，并且是函数第一个参数（也就是oldfd）指向的文件（执行完成以后，如果newfd已经打开了文件，该文件将会被关闭）
原型为：

```
#include <unistd.h>
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

文件描述符的复制是指用另外一个文件描述符指向同一个打开的文件，它完全不同于直接给文件描述符变量赋值，下面是文件描述符赋值的例子：

```
#include <func.h>
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    int fd = open(argv[1],O_RDWR);
    ERROR_CHECK(fd,-1,"open");
    printf("fd = %d\n",fd);
    int fd1 = fd;
    close(fd);
    char buf[128] = {0};
    int ret = read(fd1, buf, sizeof(buf));
    ERROR_CHECK(ret, -1, "read");
    return 0;
}
```

在此情况下，两个文件描述符变量的值相同，指向同一个打开的文件，但是内核的文件打开引用计数还是为1，所以close(fd)或者close(fd1)都会导致文件立即关闭

如果使用文件描述符的复制，则情况有所区别

```c
#include <func.h>
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    int fd = open(argv[1],O_RDWR);
    ERROR_CHECK(fd,-1,"open");
    printf("fd = %d\n",fd);
    int fd1 = dup(fd);
    close(fd);
    char buf[128] = {0};
    int ret = read(fd1, buf, sizeof(buf));
    ERROR_CHECK(ret, -1, "read");
    printf("buf = %s\n", buf);
    return 0;
}
```

使用dup函数可以实现输出重定向

```c
#include <func.h>
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    int fd = open(argv[1],O_RDWR);
    ERROR_CHECK(fd,-1,"open");
    printf("\n");
    close(STDOUT_FILENO);
    int fd1 = dup(fd);
    printf("fd1 = %d\n", fd1);
    close(fd);
    printf("the out of stdout\n");
    return 0;
}
```

- 该程序首先打开了一个文件，返回一个文件描述符，因为默认的就打开了0,1,2表示标准输入，标准输出，标准错误输出。而用close(STDOUT_FILENO);则表示关闭标准输出，此时文件描述符1就空着然后dup(fd);则会复制一个文件描述符到当前未打开的最小描述符，此时这个描述符为1。后面关闭fd自身，然后在用标准输

出的时候，发现标准输出重定向到你指定的文件了。那么printf所输出的内容也就直接输出到文件（因为printf的原理就是将内容输入到描述符为1的文件里面）

dup2(int fdold,int fdnew)也是进行描述符的复制，只不过采用此种复制，新的描述符由用户用参数fdnew显式指定。对于dup2，如果fdnew已经指向一个已经打开的文件，内核会首先关闭掉fdnew所指向的原来的文件。此时再针对于fdnew文件描述符操作的文件，则采用的是fdold的文件描述符。如果成功dup2的返回值于fdnew相同,否则为-1.

```c
//使用输出重定向
#include <func.h>
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    int fd = open(argv[1],O_RDWR);
    ERROR_CHECK(fd,-1,"open");
    printf("\n");
    int fd1 = dup2(fd,STDOUT_FILENO);
    printf("fd1 = %d\n", fd1);
    close(fd);
    printf("the out of stdout\n");
    return 0;
}
//更加复杂的例子
#include <func.h>
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    int fd = open(argv[1],O_RDWR);
    ERROR_CHECK(fd,-1,"open");
    printf("\n");
    int fd0 = 100;
    dup2(STDOUT_FILENO,fd0);
    int fd1 = dup2(fd,STDOUT_FILENO);
    printf("fd1 = %d\n", fd1);
    close(fd);
    printf("the out of stdout 1\n");
    dup2(fd0,STDOUT_FILENO);
    printf("the out of stdout 2\n");
    return 0;
```

```
}
```

- 提问：如何实现将标准输出和标准错误分别重定向到文件里面？

4.3.9 文件描述符和文件指针

- fopen函数实际在运行的过程中也获取了文件的文件描述符。使用fileno函数可以得到文件指针的文件描述符。当使用fopen获取文件指针以后，依然是可以使用文件描述符来执行IO，例如

```c
#include <func.h>
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    FILE* fp = fopen(argv[1],"rb+");
    ERROR_CHECK(fp, NULL, "fopen");
    int fd = fileno(fp);
    printf("fd = %d\n", fd);
    char buf[128] = {0};
    read(fd, buf, 5);
    printf("buf = %s\n", buf);
    //使用read接口也是能够正常读取内容的
    return 0;
}
```

- fopen的原理：fopen函数在执行的时候，会先调用open函数，打开文件并且获取文件对象的信息（通过文件描述符可以获取文件对象的具体信息），然后fopen函数会在用户态空间申请一片空间作为缓冲区
- fopen的优势：因为read和write是系统调用，需要频繁地切换用户态和内核态，所以比较耗时。借助用户态缓冲区，可以减少read和write的次数，使用fdopen函数可以根据文件描述符生成用户态缓冲区

```c
#include <func.h>
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    int fd = open(argv[1],O_RDWR);
    ERROR_CHECK(fd, -1, "open");
    FILE* fp = fdopen(fd,"rb+");
    ERROR_CHECK(fp, NULL, "fdopen");
    char buf[128] = {0};
    fread(buf,1,sizeof(buf),fp);
    printf("buf = %s\n",buf);
    return 0;
}
```

- 如果需要高效地使用不带缓冲IO，为了和存储体系匹配，最好是一次读取/写入一个块（通常是4K）大小的数据。另外如果需要使用内存映射，也应当使用open函数来打开文件
- 如果获取了文件指针，就不要通过文件描述符的方式来关闭文件

```c
#include <func.h>
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    FILE* fp = fopen(argv[1],"rb+");
    ERROR_CHECK(fp, NULL, "fopen");
    close(fileno(fp));//如果使用fd=fileno(fp),那么close以后fd的数值不会发生改变
    char buf[128] = {0};
    char *ret = fgets(buf,sizeof(buf),fp);
    ERROR_CHECK(ret, NULL, "fgets");
    printf("buf = %s\n",buf);
    return 0;
}

//出现报错 fgets: Bad file descriptor
```

4.3.10 标准输入输出文件描述符

- 与标准的输入输出流对应，在更底层的实现是用标准输入、标准输出、标准错误文件描述符表示的。它们分别用STDIN_FILENO、STDOUT_FILENO和STDERR_FILENO三个宏表示，值分别是0、1、2三个整型数字

4.3.11 有名管道

- （有名）管道文件是用来数据通信的一种文件，它是半双工通信，它在ls -l命令中显示为p，它不能存储数据

传输方式	含义
全双工	双方可以同时向另一方发送数据
半双工	某个时刻只能有一方向另一方发送数据，其他时刻的传输方向可以相反
单工	永远只能由一方向另一方发送数据

```
$ mkfifo [管道名字]
使用cat打开管道可以打开管道的读端
$ cat [管道名字]
打开另一个终端，向管道当中输入内容可以实现写入内容
$ echo “string” > [管道名字]
此时读端也会显示内容
```

- 当然也可以使用C程序来分别实现读端和写端

```
//读端
#include <func.h>
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    int fdr = open(argv[1],O_RDONLY);
    ERROR_CHECK(fdr,-1,"open");
    printf("fdr = %d\n",fdr);
    char buf[128] = {0};
    read(fdr, buf, sizeof(buf));
    printf("buf = %s\n", buf);
    return 0;
}
```

```
//写端
#include <func.h>
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    int fdw = open(argv[1],O_WRONLY);
    ERROR_CHECK(fdw,-1,"open");
    printf("fdw = %d\n",fdw);
    char buf[128] = "helloworld";
    write(fdw, buf, strlen(buf));
    printf("buf = %s\n", buf);
    return 0;
}
```

- 禁止使用vim来打开管道文件

4.4 I/O多路复用模型

4.4.1 读取操作的阻塞

- 阻塞：在目前的模式下，read函数如果不能从文件中读取内容，就将进程的状态切换到阻塞状态，不再继续执行

```
//在写端写入时添加sleep(10)
    ...
    sleep(10);
    write
    ...
//再次测试的时候发现读端会明显延迟
```

- 因为管道文件是半双工通信，为了实现全双工通信，可以使用2个管道文件。这样就可以实现即时聊天

```
//1号
#include <func.h>
int main(int argc, char *argv[])
```

```c
{
    ARGS_CHECK(argc,3);
    int fdr = open(argv[1],O_RDONLY);//管道打开的时候，必须要先将读写端都打开之后才能继续
    int fdw = open(argv[2],O_WRONLY);
    printf("I am chat1\n");
    char buf[128] = {0};
    while(1)
    {
        memset(buf,0,sizeof(buf));
        read(STDIN_FILENO, buf, sizeof(buf));
        write(fdw, buf, strlen(buf)-1);
        memset(buf,0,sizeof(buf));
        read(fdr, buf, sizeof(buf));
        printf("buf = %s\n", buf);
    }
    return 0;
}
//2号
#include <func.h>
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,3);
    int fdw = open(argv[1],O_WRONLY);//管道打开的时候，必须要先将读写端都打开之后才能继续
    int fdr = open(argv[2],O_RDONLY);
    printf("I am chat2\n");
    char buf[128] = {0};
    while(1)
    {
        memset(buf,0,sizeof(buf));
        read(fdr, buf, sizeof(buf));
        printf("buf = %s\n", buf);
        memset(buf,0,sizeof(buf));
        read(STDIN_FILENO, buf, sizeof(buf));
        write(fdw, buf, strlen(buf)-1);
    }
    return 0;
}
```

```
//这里经常会有阻塞
```

4.4.2 I/O多路复用模型和select

- I/O多路复用模型：在这种模型下，如果请求的I/O操作阻塞，且它不是真正阻塞I/O，而是让其中的一个函数等待，在这期间，I/O还能进行其他操作。如本节要介绍的select()函数，就是属于这种模型

```
#include <sys/select.h>
#include <sys/time.h>

int select(int maxfd, fd_set *readset,fd_set *writeset, fd_set
*exceptionset, struct timeval * timeout);
```

- 返回值是就绪描述字的正数目，0——超时，-1——出错

```
参数解释：
maxfd：最大的文件描述符（其值应该为最大的文件描述符字 + 1）
readset：内核读操作的描述符字集合
writeset：内核写操作的描述符字集合
exceptionset：内核异常操作的描述符字集合
timeout：等待描述符就绪需要多少时间。NULL代表永远等下去，一个固定值代表等待固定时间，0代表根本不等待，检查描述字之后立即返回
```

```
//readset、writeset、exceptionset都是fd_set集合
//集合的相关操作如下：
void FD_ZERO(fd_set *fdset);      /* 将所有fd清零 */
void FD_SET(int fd, fd_set *fdset);  /* 增加一个fd */
void FD_CLR(int fd, fd_set *fdset);  /* 删除一个fd */
int FD_ISSET(int fd, fd_set *fdset);  /* 判断一个fd是否有设置 */
```

- 一般来说，在使用select函数之前，首先要使用FD_ZERO和FD_SET来初始化文件描述符集，在使用select函数时，可循环使用FD_ISSET测试描述符集，在执行完对相关文件描述符之后，使用FD_CLR来清除描述符集。

```
//chat1.c
```

```c
//编译后运行
//$ ./chat1 1.pipe 2.pipe
#include <func.h>
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,3);
    int fdr = open(argv[1],O_RDONLY);//管道打开的时候，必须要先将读写端都打开之后才能继续
    int fdw = open(argv[2],O_WRONLY);
    printf("I am chat1\n");
    char buf[128] = {0};
    int ret;
    fd_set rdset;
    while(1)
    {
        FD_ZERO(&rdset);
        FD_SET(STDIN_FILENO,&rdset);
        FD_SET(fdr,&rdset);
        ret = select(fdr+1, &rdset, NULL, NULL, NULL);
        if(FD_ISSET(STDIN_FILENO, &rdset))
        {
            memset(buf,0,sizeof(buf));
            read(STDIN_FILENO, buf, sizeof(buf));
            write(fdw, buf, strlen(buf)-1);
        }
        if(FD_ISSET(fdr, &rdset))
        {
            memset(buf,0,sizeof(buf));
            read(fdr, buf, sizeof(buf));
            printf("buf = %s\n", buf);
        }
    }
    return 0;
}

//chat2.c
//编译后运行(注意管道建立连接的顺序)
//$ ./chat2 1.pipe 2.pipe
#include <func.h>
```

```c
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,3);
    int fdw = open(argv[1],O_WRONLY);//管道打开的时候，必须要先将读写端都打开之后才能继续
    int fdr = open(argv[2],O_RDONLY);
    printf("I am chat2\n");
    char buf[128] = {0};
    int ret;
    fd_set rdset;
    while(1)
    {
        FD_ZERO(&rdset);
        FD_SET(STDIN_FILENO,&rdset);
        FD_SET(fdr,&rdset);
        ret = select(fdr+1, &rdset, NULL, NULL, NULL);
        if(FD_ISSET(STDIN_FILENO, &rdset))
        {
            memset(buf,0,sizeof(buf));
            read(STDIN_FILENO, buf, sizeof(buf));
            write(fdw, buf, strlen(buf)-1);
        }
        if(FD_ISSET(fdr, &rdset))
        {
            memset(buf,0,sizeof(buf));
            read(fdr, buf, sizeof(buf));
            printf("buf = %s\n", buf);
        }
    }
    return 0;
}
```

- fdset实际上是一个文件描述符的位图，采用数组的形式来存储。下面是一个简化版本的实现方法

```c
//fd_set的成员是一个长整型的结构体
typedef long int __fd_mask;
//将字节转化为位
```

```
#define __NFDBITS   (8 * (int) sizeof (__fd_mask))
//位图-判断是否存在文件描述符d
#define __FD_MASK(d)    ((__fd_mask) (1UL << ((d) % __NFDBITS)))
//select和pselect的fd_set结构体
typedef struct
{
    //成员就是一个长整型的数组，用来实现位图
    __fd_mask __fds_bits[__FD_SETSIZE / __NFDBITS];
} fd_set;

// fd_set里面文件描述符的数量，可以使用ulimit -n进行查看
#define FD_SETSIZE      __FD_SETSIZE
```

- maxfd为什么是最大描述符加1呢？
  当传入fdmax的时候，select会监听0~fdmax-1的文件描述符

4.4.3 select的退出机制

- 当管道的写端先关闭的时候，读端使用read的时候会返回0（相当于发送了一个EOF），操作系统会将管道的状态设置为可读，可读的状态会影响到select函数，导致select函数不会阻塞，进入死循环。下面是一个简单的例子来说明写端关闭的情况

```
//将写端的程序修改为如此
#include <func.h>
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    int fdw = open(argv[1],O_WRONLY);
    ERROR_CHECK(fdw,-1,"open");
    printf("fdw = %d\n",fdw);
    close(fdw);//这里将写端直接关闭
    sleep(10);//然后睡眠10s
    return 0;
}
```

- 当管道的读端先关闭的时候，写端会直接崩溃（这种情况的处理方法要等到网络编程阶段才会介绍）
- 为了避免死循环，需要在读端对退出的情况进行兼容处理，就是当read的返回值为0的时候，就退出程序

```
    ...
        if(FD_ISSET(STDIN_FILENO, &rdset))
        {
            memset(buf,0,sizeof(buf));
            read_ret = read(STDIN_FILENO, buf, sizeof(buf));
            if(read_ret == 0)
            {
                printf("chat is broken!\n");
                break;
            }
            write(fdw, buf, strlen(buf)-1);
        }
        if(FD_ISSET(fdr, &rdset))
        {
            memset(buf,0,sizeof(buf));
            read_ret = read(fdr, buf, sizeof(buf));
            if(read_ret == 0)
            {
                printf("chat is broken!\n");
                break;
            }
            printf("buf = %s\n", buf);
        }
    ...
```

```
#使用ctrl+c终止程序会导致程序的返回值不为0
#可以改用ctrl+d来终止stdin（相当于输入了EOF）
#$?代表了上个执行程序的返回值
$echo $?
```

4.4.4 select函数的超时处理

- 使用timeval结构体可以设置超时时间。传入select函数中的timeout参数是一个timeval结构体指针，timeval结构体的定义如下：

```
struct timeval
{
    long tv_sec;//秒
    long tv_usec;//微秒
};
//用法
    ...
    struct timeval timeout;
    while(1)
    {
        bzero(&timeout, sizeof(timeout));
        timeout.tv_sec = 3;
        ret = select(fdr+1, &rdset, NULL, NULL, &timeout);
        if(ret > 0)
        {
            ...
        }
        else
        {
            printf("time out!\n");
        }
    }
```

- 使用的超时判断的时候要注意，每次调用select之前需要重新为timeout赋值，因为调用select会修改timeout里面的内容

4.4.5 写集合的原理

- 写阻塞和写就绪：当管道的写端向管道中写入数据达到上限以后，后续的写入操作将会导致进程进入阻塞态，称为写阻塞现象
- 提问：如何实现一个写阻塞？如何检查写阻塞的中断问题？

- 处于写阻塞状态以后，当管道中的数据被读端读取以后，写端就可以恢复写入操作，称为写就绪
- 类似于读文件描述符集合，select也可以设置专门的写文件描述符集合，select可以监听处于写阻塞状态下的文件，一旦文件转为写就绪，就可以将进程转换为就绪

```c
#include <func.h>
int main(int argc, char* argv[])
{
    ARGS_CHECK(argc, 2);
    int fdr = open(argv[1],O_RDWR);
    int fdw = open(argv[1],O_RDWR);//可以一次性打开管道的读写端
    fd_set rdset,wrset;
    int ret;
    char buf[128];
    while(1)
    {
        FD_ZERO(&rdset);
        FD_ZERO(&wrset);
        FD_SET(fdr, &rdset);
        FD_SET(fdw, &wrset);
        ret = select(fdw+1, &rdset, &wrset, NULL, NULL);
        if(FD_ISSET(fdr, &rdset))
        {
            bzero(buf, sizeof(buf));
            read(fdr, buf, sizeof(buf));
            puts(buf);
            usleep(250000);

        }
        if(FD_ISSET(fdw, &wrset))
        {
            write(fdw,"helloworld", 10);
            usleep(500000);
        }
    }
}
```

4.5 杂项

- 安装vimplus：

1. 安装YouCompleteMe

```
$sudo apt install vim-youcompleteme
```

2. 安装vimplus剩余插件

```
$sudo apt install git
$cd ~
$git clone https://gitee.com/chxuan/vimplus.git ~/.vimplus
$cd .vimplus
$./install.sh
# 如果没有自动补全功能
$vim-addons remove youcompleteme
$vim-addons install youcompleteme
```

3. 在安装了vimplus以后修改snippet.c可设置默认的内容