

# 1.C语言文件目录操作

在Linux中文件总共被分成了7种，他们分别是：

- (1) 普通文件(regular) :存在于外部存储器中，用于存储普通数据。
- (2) 目录文件(directory) :用于存放目录项，是文件系统管理的重要文件类型。
- (3) 管道文件(pipe) :一种用于进程间通信的特殊文件，也称为命名管道FIFO。
- (4) 套接字文件(socket) :一种用于网络间通信的特殊文件。
- (5) 链接文件(link) :用于间接访问另外一个目标文件，相当于Windows快捷方式。
- (6) 字符设备文件(character) :字符设备在应用层的访问接口。
- (7) 块设备文件(block):块设备在应用层的访问接口。

## 1.1.标准IO

### 1.1.1.打开文件

函数原型：

```
1 FILE *fopen(const char *path, const char *mode)
```

参数说明：

- 1 path:即将要打开的文件
- 2 mode:"r":以只读方式打开文件，要求文件必须存在
- 3 "r+":以读写方式打开文件，要求文件必须存在
- 4 "w":以只写方式打开文件，文件如果不存在将会创建新文件，如果存在将会清空其内容
- 5 "w+":以读写方式打开文件，文件如果不存在将会创建新文件，如果存在将会清空其内容
- 6 "a":以只写方式打开文件，文件如果不存在将会创建新文件，且文件位置偏移量自动定位到文件末尾  
(即以追加方式写数据)
- 7 "a+":以读写方式打开文件，文件如果不存在将会创建新文件，且文件位置偏移量自动定位到文件末尾  
(即以追加方式写数据)

返回值：

- 成功:文件指针
- 失败:NULL

### 1.1.2.关闭文件

函数原型：

```
1 int fclose(FILE *fp)
```

参数说明:

```
1 fp:即将要关闭的文件
```

返回值:

- 成功:0
- 失败:EOF

### 1.1.3.从文件中读取一个字符

函数原型:

```
1 int fgetc(FILE *stream)
2 int getc(FILE *stream)
3 int getchar(void)
```

参数说明:

```
1 stream:文件指针
```

返回值:

- 成功:读取到的字符
- 失败:EOF

### 1.1.4.向文件中写入一个字符

函数原型:

```
1 int fputc(int c,FILE *stream)
2 int putc(int c,FILE *stream)
3 int putchar(int c)
```

参数说明：

- 1 stream: 文件指针
- 2 c: 要写入的字符

返回值：

- 成功: 写入的字符
- 失败: EOF

### 1.1.5.feof()和ferror()

函数原型：

- 1 `int feof(FILE *stream)`
- 2 `int ferror(FILE *stream)`

参数说明：

- 1 stream: 文件指针

返回值：

- feof: 如果文件已达末尾则返回真，否则返回假
- ferror: 如果文件遇到错误则返回真，否则返回假

### 1.1.6.从文件中读取字符串

函数原型：

- 1 `char *fgets(char *s, int size, FILE *stream)`
- 2 `char *gets(char *s)`

参数说明：

- 1 s: 自定义缓冲区指针
- 2 size: 自定义缓冲区大小

3 stream: 文件指针

返回值:

- 成功:自定义缓冲区指针s
- 失败:NULL

备注:

💡 gets()默认从文件stdin读取数据当返回NULL时，文件stream可能已达末尾，或者遇到错误

### 1.1.7.向文件中写入字符串

函数原型:

```
1 int fputs(const char *s, FILE *stream)
2 int puts(const char *s)
```

参数说明:

```
1 s:自定义缓冲区指针
2 stream:文件指针
```

返回值:

- 成功:非负整数
- 失败:EOF

备注:

💡 puts()默认将数据写入文件stdout

### 1.1.8.读写操作

函数原型:

```
1 size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)
2 size_t fwrite(void *ptr, size_t size, size_t nmemb, FILE *stream)
```

参数说明:

- 1 ptr:自定义缓冲区指针
- 2 size:数据块大小
- 3 nmemb:数据块个数
- 4 stream:文件指针

返回值:

- 成功:读取或写入的数据块个数, 等于nmemb
- 失败:读取或写入的数据块个数, 小于nmemb或等于0

备注:



当返回小于nmemb时, 文件stream可能已达末尾, 或遇到错误

### 1.1.9.设置文件当前位置偏移量

函数原型:

```
1 int fseek(FILE *stream, long offset, int whence)
```

参数说明:

- 1 stream:文件指针
- 2 offset:新文件偏移量相对基准点的偏移
- 3 whence(基准点):SEEK\_SET:文件开头处
- 4                   SEEK\_CUR:当前位置
- 5                   SEEK\_END:文件末尾处

返回值:

- 成功:0
- 失败:-1

### 1.1.10.获取指定文件的当前位置偏移量

函数原型:

```
1 long ftell(FILE *stream)
```

参数说明：

```
1 stream: 文件指针
```

返回值：

- 成功: 当前文件位置偏移量
- 失败: -1

### 1.1.11. 将指定文件的当前位置偏移量设置到文件开头处

函数原型：

```
1 void rewind(FILE *stream)
```

参数说明：

```
1 stream: 文件指针
```

## 1.2. 文件IO

### 1.2.1. 打开文件

函数原型：

```
1 int open(const char *pathname, int flags)
2 int open(const char *pathname, int flags, mode_t mode)
```

参数说明：

```
1 pathname: 即将要打开的文件
2 flags: O_RDONLY, 以只读方式打开文件
3       O_WRONLY, 以只写方式打开文件
4       O_RDWR, 以读写方式打开文件
5       O_CREAT, 如果文件不存在, 则创建文件
```

- 6        `O_EXCL`，如果使用`O_CREAT`选项且文件存在，则返回错误信息
- 7        `O_NOCTTY`，如果文件为终端，那么终端不可以作为调用`open()`系统电影的那个进程的控制终端
- 8        `O_TRUNC`，如文件已经存在，则删除文件中的原数据
- 9        `O_APPEND`，以追加的方式打开文件
- 10    `mode`:如果文件被新建，指定其权限为`mode`(八进制表示法)

返回值：

- 成功:大于等于0的整数
- 失败:-1

### 1.2.2.关闭文件

函数原型：

```
1  int close(int fd)
```


参数说明：

```
1  fd: 文件描述符
```

返回值：

- 成功:0
- 失败:-1

备注：

 重复关闭一个已经关闭了的文件或者尚未打开的文件是安全的

### 1.2.3.文件读写

函数原型：

```
1  ssize_t read(int fd,void *buf,size_t count)
2  ssize_t write(int fd,const void *buf,size_t count)
```

参数说明：

- 1 `fd`: 文件描述符
- 2 `buf`: 指向存放读取数据的缓冲区，或者即将要写入的数据
- 3 `count`: 想要读取或写入的字节数

返回值:

- 成功: 实际读取或写入的字节数
- 失败: -1

备注:

💡 实际读到或写入的字节数小于等于count

## 1.2.4.调整文件位置偏移量lseek()

函数原型:

```
1 off_t lseek(int fd, off_t offset, int whence)
```

参数说明:

- 1 `fd`: 文件描述符
- 2 `offset`: 新位置偏移量相对基准点的偏移
- 3 `whence`: `SEEK_SET`, 文件开头处
- 4 `SEEK_CUR`, 当前位置
- 5 `SEEK_END`, 文件末尾处

返回值:

- 成功: 新文件位置偏移量
- 失败: -1

## 1.2.5.复制文件描述符

函数原型:

```
1 int dup(int oldfd)
2 int dup2(int oldfd, int newfd)
```



参数说明：

- 1 oldfd:要复制的文件描述符
- 2 newfd:指定的新文件描述符

返回值：

- 成功:新的文件描述符
- 失败:-1

## 1.3.目录操作

### 1.3.1.打开目录

函数原型：

```
1 DIR *opendir(const char *name)
```

参数说明：

- 1 name:目录名

返回值：

- 成功:目录指针
- 失败:NULL

### 1.3.2.读取目录

函数原型：

```
1 struct dirent *readdir(DIR *dirp)
```

参数说明：

- 1 dirp:读出目录项的目录指针

返回值：

- 成功:目录项指针
- 失败:NULL

从目录中读到的所谓目录项，是一个这样的结构体：

```
1 struct dirent
2 {
3     ino_t d_ino; // 文件索引号
4     off_t d_off; // 目录项偏移量
5     unsigned short d_reclen; // 该目录项大小
6     unsigned char d_type; // 文件类型
7     char d_name[256]; // 文件名
8 };
```

## 1.4.文件属性

### 1.4.1获取文件信息

函数原型：

```
1 int stat(const char *path,struct stat *buf)
2 int fstat(int fd,struct stat *buf)
3 int lstat(const char *path,struct stat *buf)
```

参数说明：

```
1 path:文件路径
2 fd: 文件描述符
3 buf:属性结构体
```

返回值：

- 成功:0
- 失败:NULL

属性结构体如下所示：

```

1 struct stat
2 {
3     dev_t st_dev; // 普通文件所在存储器的设备号
4     mode_t st_mode; // 文件类型、文件权限
5     ino_t st_ino; // 文件索引号
6     nlink_t st_nlink; // 引用计数
7     uid_t st_uid; // 文件所有者的UID
8     gid_t st_gid; // 文件所属组的GID
9     dev_t st_rdev; // 特殊文件的设备号
10    off_t st_size; // 文件大小
11    blkcnt_t st_blocks; // 文件所占数据块数目
12    time_t st_atime; // 最近访问时间
13    time_t st_mtime; // 最近修改时间
14    time_t st_ctime; // 最近属性更改时间
15    blksize_t st_blksize; // 写数据块建议值
16 };

```

该结构体中有很多成员的含义和作用是一目了然的，比如：

- 文件索引号: `st_ino`, 实质上是一个无符号整形数据，用来唯一确定分区中的文件。
- 引用计数: `st_nlink`，记录该文件的名字(或叫硬链接)总数，文件的别名可以用命令`link`或者函数`link()`来创建。当一个文件的引用计数`st_nlink`为零时，系统将会释放清空该文件锁占用的一切系统资源。
- 文件所有者UID和所属组GID。
- 文件的大小。这个属性只对普通文件有效。
- 文件所占数据块数目`st_blocks`, 表明该文件实际占用存储器空间。一个数据块一般为512字节。
- `st_atime`、`st_mtime`和`st_ctime`都是一个文件的时间戳，`st_atime`代表文件被访问了但是没有被修改的最近时间，`st_mtime`代表文件内容被修改的最近时间，`st_ctime`则代表了文件属性更改的最近时间。文件的时间戳对于某些场合来讲是至关重要的属性，比如工程管理器`make`,他的工作原理就完全基于文件的时间戳上，判断文件的被修改时间，决定其是否参与编译。
- `st_blksize` 是所谓的“写数据块”的建议值，因为当应用程序频繁地往存储器写入小块数据的时候，可能会导致效率的低下。

## 1.4.2.分离主次设备号

函数原型：

```

1 int major(dev_t dev)
2 int minor(dev_t dev)

```

参数说明：

```
1 dev: 文件的设备号属性，来自stat结构体中的st_dev或者st_rdev
```

返回值：

- 成功: major返回主设备号，minor返回次设备号
- 失败: 无

### 1.4.3. 文件类型和权限

属性成员中的st\_mode 里面包含了文件类型和权限，st\_mode 实质上是一个无符号16位短整型数，各个位域所包含的含义如下：

- st\_mode[0:8]---对应地代表了文件的各个用户的权限。
- st\_mode[9]存储了所谓的黏住位(只对目录有效)，在拥有该目录的写权限的情况下，如果这一位被设置为1,那么某一用户也只能删除在本目录下属于自己的文件，否则可以删除任意文件。
- st\_mode[10]和st\_mode[11] 分别用来设置文件的suid (只对普通文件有效)和sgid (只对目录有效)。如果suid被设置为1，则任何用户在执行该文件的时候均会获得该文件所有者的临时授权,即其有效UID将等于文件所有者的UID。如果sgid被设置为1,则任何在该目录下执行的程序均会获得该目录所属组成员的临时授权，即其有效GID将等于该目录的所属组成员的GID。
- st\_mode[12:15]用以标识Linux下不同的文件类型，由于Linux总共只有7种文件类型，因此4位足以表达。

下面的表格是st\_mode 的详细信息：

### 1.4.4. 判断文件类型的宏

### 1.4.5. 判断文件是否存在

函数原型：

```
1 int access(const char *filename, int mode);
```

参数定义：

```
1 filename: 可以填写文件夹路径或者文件路径
2 mode: 0 (F_OK) 只判断是否存在
3       2 (R_OK) 判断写入权限
```

4	4 （W_OK） 判断读取权限
5	6 （X_OK） 判断执行权限
6	用于判断文件夹是否存在的时候，mode取0，判断文件是否存在的时候，mode可以取0、2、4、6。

返回值：

- 成功：若存在或者具有权限，返回值为0
- 失败：不存在或者无权限，返回值为-1。

## 2.Qt文件目录操作

### 2.1.文件打开方式

QIODevice::NotOpen	未打开
QIODevice::ReadOnly	以只读方式打开
QIODevice::WriteOnly	以只写方式打开
QIODevice::ReadWrite	以读写方式打开
QIODevice::Append	以追加的方式打开，新增加的内容将被追加到文件末尾
QIODevice::Truncate	以重写的方式打开，在写入新的数据时会将游标设置在文件开头
QIODevice::Text	在读取时，将行结束符转换成 \n；在写入时，将行结束符转换成本地格式，例如 Win32 平台上是 \r\n
QIODevice::Unbuffered	忽略缓存

### 2.2.文件属性

```
1 // 查看文件属性
2 QFileInfo info(file);
3 qDebug() << info.size();           // 查看文件大小
4 qDebug() << info.exists();         // 判断文件是否存在
5 qDebug() << info.isDir();          // 判断是否为目录
6 qDebug() << info.isFile();         // 判断是否为文件
7 qDebug() << info.isReadable();     // 判断是否可读
8 qDebug() << info.isWritable();     // 判断是否可写
9 qDebug() << info.isHidden();       // 判断隐藏属性
```

```

10 qDebug() << info.isExecutable(); // 判断可执行属性
11
12 QDateTime create_time = info.created(); // 查看文件创建的时间
13 QDateTime lastModified_time = info.lastModified(); // 查看文件最后修改的时间
14 QDateTime lastRead_time = info.lastRead(); // 查看文件最后的访问时间

```

## 2.3.获取系统下载、文档等的默认目录

```

1 qDebug() << QApplication::applicationDirPath(); // 获取当前程序所在的目录
2 qDebug() << QApplication::applicationFilePath(); // 获取当前程序的文件路径
3 qDebug() << QDir::currentPath(); // 获取当前工作目录
4 qDebug() << QDir::homePath(); // 获取家目录
5 qDebug() << QDir::rootPath(); // 获取根目录
6 qDebug() << QDir::tempPath(); // 获取临时文件目录（在程序中创建的临时目录和临时文件会放在这个路径下）
7
8 qDebug() << "系统字体目录路径:" <<
  QStandardPaths::standardLocations(QStandardPaths::FontsLocation);
9 qDebug() << "系统桌面目录路径:" <<
  QStandardPaths::standardLocations(QStandardPaths::DesktopLocation);
10 qDebug() << "用户文档目录路径:" <<
  QStandardPaths::standardLocations(QStandardPaths::DocumentsLocation);
11 qDebug() << "用户音乐目录路径:" <<
  QStandardPaths::standardLocations(QStandardPaths::MusicLocation);
12 qDebug() << "用户视频目录路径" <<
  QStandardPaths::standardLocations(QStandardPaths::MoviesLocation);
13 qDebug() << "用户图片目录路径:" <<
  QStandardPaths::standardLocations(QStandardPaths::PicturesLocation);
14 qDebug() << "系统临时文件目录路径:" <<
  QStandardPaths::standardLocations(QStandardPaths::TempLocation);
15 qDebug() << "系统缓存目录路径:" <<
  QStandardPaths::standardLocations(QStandardPaths::CacheLocation);
16 qDebug() << "系统下载目录路径:" <<
  QStandardPaths::standardLocations(QStandardPaths::DownloadLocation);
17 qDebug() << "系统家目录" <<
  QStandardPaths::standardLocations(QStandardPaths::HomeLocation);

```

## 2.4.临时文件和临时目录

- 临时文件在函数结束时，或者执行close()之后，就会自动删除

- 在函数结束时，或者执行了remove()之后，临时目录会自动删除

## 2.5.目录操作

QDir::Dirs	列出与筛选器匹配的目录
QDir::AllDirs	列出与筛选器匹配的目录、列出所有目录；即不将过滤器应用于目录名
QDir::Files	列出文件
QDir::Drives	列出磁盘驱动器（在Unix下被忽略）
QDir::NoSymLinks	不要列出符号链接（被不支持符号链接的操作系统忽略）
QDir::NoDotAndDotDot	不要列出特殊条目“.”和“..”
QDir::NoDot	不要列出特殊条目“.”
QDir::NoDotDot	不要列出特殊条目“..”
QDir::AllEntries	列出目录、文件、驱动器和符号链接（除非指定System，否则不会列出损坏的符号链接）
QDir::Readable	列出应用程序具有读取权限的文件。Readable值需要与Dirs或Files组合使用
QDir::Writable	列出应用程序具有写访问权限的文件。“可写入”值需要与“目录”或“文件”组合使用
QDir::Executable	列出应用程序具有执行访问权限的文件。Executable值需要与Dirs或Files组合
QDir::Modified	仅列出已修改的文件（在Unix上被忽略）
QDir::Hidden	列出隐藏文件（在Unix上，以“.”开头的文件）
QDir::System	列出系统文件（在Unix上，包括FIFO、套接字和设备文件；在Windows上，包括.lnk文件）
QDir::CaseSensitive	过滤器应区分大小写

### 2.5.1遍历目录及其子目录下的文件和文件夹

QDirIterator是一个类，用于遍历目录及其子目录下的文件和文件夹。在下面这个例子中，我们创建了一个QDirIterator对象dir\_it，指定了要遍历的目录为curr\_path/record，并且设置了遍历的选项为Dirs（只遍历目录）和NoDotAndDotDot（不包括“.”和“..”）。同时，通过设置Subdirectories选项为true，可以递归地遍历所有子目录。

```
1 QDirIterator dir_it(QString("%1/record").arg(curr_path),
  QDir::Dirs | QDir::NoDotAndDotDot, QDirIterator::Subdirectories);
```

## 2.5.2.遍历目录下的文件和文件夹

```
1 QDir dir = QDir(list.first());
2 dir.setFilter(QDir::Files | QDir::Hidden | QDir::NoSymLinks);
3 dir.setSorting(QDir::Time);
4 QFileInfoList file_list = dir.entryInfoList();
```

## 3.C++文件目录操作

在C++中，输入输出是同流来完成的。C++的输出操作将一个对象的状态转换成一个字符序列，输出到某个地方。输入操作也是从某个地方接收到一个字符序列，然后将其转换成一个对象的状态所要求的格式。这看起来很像数据在流动，于是把接收输出数据的地方叫做目标，把输入数据来自的地方叫做源。而输入和输出操作可以看成字符序列在源、目标以及对象之间的流动。

在C++里，文件操作是通过流来完成的。C++总共有输入文件流、输出文件流和输入输出文件流3种，并已将它们标准化。

1. 要打开一个输入文件流，需要定义一个 ifstream类型的对象。->Input-stream
2. 要打开一个输出文件流，需要定义一个 ofstream类型的对象。->Output-stream
3. 如果要打开输入输出文件流，则要定义一个 fstream类型的对象。->File-stream

这3种类型都定义在头文件 fstream 里。

1. 一个输出流对象是信息流动的目标，ofstream是最重要的输出流。
2. 一个输入流对象是数据流动的源头，ifstream是最重要的输入流。
3. 一个iostream对象可以是数据流动的源或目标，fstream就是从它派生的。

### 3.1.IO流

- istream 是用于输入的流类，cin 就是该类的对象。
- ostream 是用于输出的流类，cout 就是该类的对象。
- ifstream 是用于从文件读取数据的类。
- ofstream 是用于向文件写入数据的类。
- iostream 是既能用于输入，又能用于输出的类。
- fstream 是既能从文件读取数据，又能向文件写入数据的类。
- istrstream 输入字符串类
- ostrstream 输出字符串类



- `stringstream` 输入输出字符串流类

### 3.1.1.标准输入/输出流

C++的输入/输出流库(iostream)中定义了4个标准流对象：`cin`(标准输入流——键盘)，`cout`(标准输出流——屏幕)，`cerr`(标准错误流——屏幕)，`clog`(标准错误流——屏幕)

- `cerr` 不使用缓冲区，直接向显示器输出信息；而输出到 `clog` 中的信息会先被存放到缓冲区，缓冲区满或者刷新时才输出到屏幕。
- `cout` 是 `ostream` 类的对象，`ostream` 类的无参构造函数和复制构造函数都是私有的，所以无法定义 `ostream` 类的对象。
- 使用<>提取数据时，系统会跳过空格，制表符，换行符等空白字符。所以一组变量输入值时，可用这些隔开。
- 输入字符串，也是跳过空白字符，会在串尾加上字符串结束标志\0。

### 3.1.2.输入流中的成员函数

- `get`函数：`cin.get()`，`cin.get(ch)`（成功返回非0值，否则返回0），`cin.get(字符数组(或字符指针), 字符个数n, 终止字符)`
- `getline`函数：`cin.getline(字符数组(或字符指针), 字符个数n, 终止标志字符)`，读取字符知道终止字符或者读取n-1个字符，赋值给指定字符数组（或字符指针）
- `cin.peek()` 不会跳过输入流中的空格、回车符。在输入流已经结束的情况下，`cin.peek()` 返回 EOF。
- `ignore(int n = 1, int delim = EOF)`
- `putback(char c)`，可以将一个字符插入输入流的最前面。

### 3.1.3.输出流对象

- 插入 `endl` 输出所有数据，插入换行符，清空缓冲区
- `\n` 输出换行，不清空缓冲区
- `cout.put(参数)` 输出单个字符（可以是字符也可以是ASCII码）

## 3.2.文件操作

C++ 标准类库中有三个类可以用于文件操作，它们统称为文件流类。这三个类是：

- `ifstream`：输入流类，用于从文件中读取数据。
- `ofstream`：输出流类，用于向文件中写入数据。
- `fstream`：输入/输出流类，既可用于从文件中读取数据，又可用于向文件中写入数据。

### 3.2.1.打开文件

`open`函数的原型如下：

```
1 void open(char const *,int filemode,int = filebuf::openprot);
```

它有3个参数，第1个是要打开的文件名，第2个是文件的打开方式，第3个是文件的保护方式，一般都使用默认值。第2个参数可以取如下所示的值：

模式标记	适用对象	作用
ios::in	ifstream fstream	打开文件用于读取数据。如果文件不存在，则打开出错。
ios::out	ofstream fstream	打开文件用于写入数据。如果文件不存在，则新建该文件；如果文件原来就存在，则打开时清除原来的内容。
ios::app	ofstream fstream	打开文件，用于在其尾部添加数据。如果文件不存在，则新建该文件。
ios::ate	ifstream	打开一个已有的文件，并将文件读指针指向文件末尾（读写指的概念后面解释）。如果文件不存在，则打开出错。
ios:: trunc	ofstream	单独使用时与 ios:: out 相同。
ios::binary	ifstream ofstream fstream	以二进制方式打开文件。若不指定此模式，则以文本模式打开。

### 3.2.2.移动和获取文件读写指针

- ifstream 类和 fstream 类有 seekg 成员函数，可以设置文件读指针的位置；
- ofstream 类和 fstream 类有 seekp 成员函数，可以设置文件写指针的位置。
- ifstream 类和 fstream 类还有 tellg 成员函数，能够返回文件读指针的位置；
- ofstream 类和 fstream 类还有 tellp 成员函数，能够返回文件写指针的位置。

### 3.2.3.普通文件读写操作示例

函数功能：

函数	功能
bad()	如果进行非法操作，返回true，否则返回false
clear()	设置内部错误状态，如果用缺省参量调用则清除所有错误位
eof()	如果提取操作已经到达文件尾，则返回true，否则返回false

good()	如果没有错误条件和没有设置文件结束标志，返回true，否则返回false
fail()	与good相反，操作失败返回false，否则返回true
is_open()	判定流对象是否成功地与文件关联，若是，返回true，否则返回false

读文件：

```

1 void inputfile()
2 {
3     ifstream ofs;
4     ofs.open("test.txt", ios::in);
5     if (ofs.is_open())
6     {
7         cout << "已成功和文件关联!!!" << endl;
8
9         // 三种读数据方式
10
11         // 第一种
12         char chr[1024];
13         while (ofs >> chr)
14         {
15             cout << chr << endl;
16         }
17
18         // 第二种
19         char buf[1024] = {0};
20         while (ofs.getline(buf, sizeof(buf)))
21             ;
22         {
23             cout << buf << endl;
24         }
25
26         // 第三种 第三种忘记了，反正前两种应该够用了
27     };
28     if (!ofs.is_open())
29     { // 不为真
30         cout << "尚未与文件关联!!!" << endl;
31     }

```

```
32     ofs.close();
33 }
```

### 3.2.4.二进制文件读写操作示例

写文件：

```
1 void outbin()
2 {
3     ofstream obin; // 创建写出流对象
4     obin.open("testbin.txt", ios::out | ios::binary); // 二进制方式打开
5     Person p = {"李四", 12, '男'};
6
7     // 把对象写入到obin
8     obin.write((const char *)&p, sizeof(p));
9
10    // 关闭文件
11    obin.close();
12 }
```

读文件：

```
1 // 二进制方式读文件
2 void readbin()
3 {
4     // 包含头文件
5     ifstream inbin;
6
7     // 创建流对象
8     inbin.open("testbin.txt", ios::in | ios::binary);
9
10    // 判断是否打开文件
11    if (!inbin.is_open())
12    {
13        cout << "文件打开失败!!" << endl;
14        return;
15    }
```

```

16     Person p;
17     inbin.read((char *)&p, sizeof(p));
18     cout << "姓名: " << p.name << "年龄: " << p.age << endl;
19
20     inbin.close();
21 }

```

## 3.3.目录操作

参考: <https://learn.microsoft.com/zh-cn/cpp/standard-library/filesystem?view=msvc-170>

### 3.3.1.头文件与命名空间

头文件只需要#include <filesystem>

C++11时还在TR2里面, C++11的命名空间为std::tr2::sys

C++17时已经正式引入了, C++17的命名空间为std::filesystem

### 3.3.2.Filesystem常用成员函数

函数名	功能
void copy(const path& from, const path& to)	目录复制
path absolute(const path& pval, const path& base = current_path())	获取相对于base的绝对路径
bool create_directory(const path& pval)	当目录不存在时创建目录
bool create_directories(const path& pval)	形如/a/b/c这样的, 如果都不存在, 创建目录结构
uintmax_t file_size(const path& pval)	返回目录的大小
file_time_type last_write_time(const path& pval)	返回目录最后修改日期的file_time_type对象
bool exists(const path& pval)	用于判断path是否存在
bool remove(const path& pval)	删除目录
uintmax_t remove_all(const path& pval)	递归删除目录下所有文件, 返回被成功删除的文件个数
void rename(const path& from, const path& to)	移动文件或者重命名

### 3.3.3.path类

函数名	功能
path& append(const _Src& source)	在path末尾加入一层结构
path& assign(string_type& source)	赋值（字符串）
void clear()	清空
int compare(const path& other)	进行比较
bool empty()	空判断
path filename()	返回文件名（有后缀）
path stem()	返回文件名（不含后缀）
path extension()	返回文件后缀名
path is_absolute()	判断是否为绝对路径
path is_relative()	判断是否为相对路径
path relative_path()	返回相对路径
path parent_path()	返回父路径
path& replace_extension(const path& replace)	替换文件后缀

### 3.3.4.创建目录

可以使用std::filesystem::create\_directory函数来创建一个新的目录。例如：

```

1 #include <filesystem>
2 namespace fs = std::filesystem;
3
4 fs::create_directory("path/to/directory");
```

### 3.3.5.删除文件或目录

可以使用std::filesystem::remove函数来删除指定的文件或目录。例如：

```

1 #include <filesystem>
2 namespace fs = std::filesystem;
3
```

```
4 fs::remove("path/to/file");
5 fs::remove_all("path/to/directory"); // 删除整个目录及其内容
```

### 3.3.6.重命名文件或目录

可以使用std::filesystem::rename函数来重命名指定的文件或目录。例如：

```
1 #include <filesystem>
2 namespace fs = std::filesystem;
3
4 fs::rename("old_name.txt", "new_name.txt");
```

### 3.3.7.遍历目录

可以使用std::filesystem::directory\_iterator来遍历指定目录下的文件和子目录。例如：

```
1 #include <filesystem>
2 namespace fs = std::filesystem;
3
4 for (const auto& entry : fs::directory_iterator("path/to/directory")) {
5     std::cout << entry.path() << std::endl;
6 }
```

### 3.3.8.判断文件或目录是否存在

可以使用std::filesystem::exists函数来判断指定的文件或目录是否存在。例如：

```
1 #include <filesystem>
2 namespace fs = std::filesystem;
3
4 if (fs::exists("path/to/file")) {
5     std::cout << "File exists" << std::endl;
6 }
```

### 3.3.9.使用迭代器方式遍历目录

```
1  #include <iostream>
2  #include <filesystem>
3
4  namespace fs = std::filesystem;
5
6  int main() {
7      std::string path = "your_directory_path";
8      for (const auto& entry : fs::directory_iterator(path)) {
9          if (fs::is_directory(entry.status())) {
10             std::cout << entry.path().filename().string() << std::endl;
11         }
12     }
13
14     return 0;
15 }
```

### 3.3.10.递归遍历目录

```
1  #include <iostream>
2  #include <filesystem>
3
4  namespace fs = std::filesystem;
5
6  void recursive_directory(const fs::path& path, std::set<std::string>& dir_set) {
7      for (const auto& entry : fs::recursive_directory_iterator(path)) {
8          if (fs::is_directory(entry.status())) {
9              dir_set.insert(entry.path().filename().string());
10         }
11     }
12 }
13
14 int main() {
15     std::string path = "your_directory_path";
16     std::set<std::string> dir_set;
```



```
17
18     recursive_directory(path, dir_set);
19
20     for (const auto& dir : dir_set) {
21         std::cout << dir << std::endl;
22     }
23
24     return 0;
25 }
```