标准IO

1．标准IO的人性化?

相比于系统IO来说，标准io处理了优化块长度的读取和缓冲区的分配等方面，使得用户的IO速度得到优化

2．标准IO和系统IO的区别

系统IO是围绕文件描述符来进行读写，标准IO底层基于系统IO，是围绕着文件流来进行操作的，文件流是一个结构体。

2．流的定向问题？

由于字符集的不同，不同字符集每个字符占的字节数不同，所以要使得流对字节流的解读正确，就要设置流的定向，也就是设置流对字节是如何解读的。

1． 新创建的流没有被定向，如果在这个流上使用一个多字节的IO,就会被定向到多字节流，反之，单字节流。

2． int fwide(FILE *fp, int mode) //设置未定向流的定向，如果流是宽的，返回正，流是单字节，返回赋值，为定向，返回 0

mode < 0:指定流为单字节

mode > 0:指定流为宽字节

mode = 0:不设置流定向，返回流当前的定向类型

如果流是已经被定义的，那么不会起作用，该函数只对未定义的流产生作用

标准输入 、输出 、错误

STDIN_FILENO	STDOUT_FILENO	STDERR_FILENO
stdin	stdout	stderr

缓冲

1．为什么要使用缓冲？

尽可能的减少对read和write的调用次数

2．缓冲的分类？

2.1 全缓冲

只有在填满缓冲io的时候才会进行实际的io操作，对于磁盘上的文件，一般采用这种模式。

2.1 行缓冲

当在输入输出时候填满一行或者遇到换行符的时候或者遇到输入，就会进行实际的io操作

2.3 无缓冲

不进行缓冲，直接进行io

3．冲洗

缓冲区可以手动进行冲洗输出，flush会将缓冲区中的数据输出或者写到磁盘上, flush在终端驱动程序上有另外的含义

int fflush(FILE *fp) //成功，返回0,失败，返回NULL, 如果fp=NULL, 则清洗所有的输出流

4．缓冲区的设置要求

当且仅当标准输入和标准输出不指向交互式设备时，才是全缓冲的

标准错误绝不会是全缓冲的

其余的，系统默认打开是全缓冲的

缓冲_缓冲类型更改

void setbuf(FILE *fp, char *buf)

打开或者关闭缓冲，将流设置以buf为缓冲区的全缓冲模式，如果流是指向交互式设备，系统也许会将其修改为行缓冲
将buf设置为NULL，将变为无缓冲。

int setvbuf(FILE *fp, char *buf, int mode, size_t size) //成功，返回 0 ，出错，返回非零

mode: _IOFBF
_IOLBF
_IONBF

1. 如果设置为无缓冲，那么buf和size会被忽略
2. 如果设置为有缓冲，但是buff却为NULL,那么系统会自动提供缓冲区

打开流

FILE *fopen(const char *pathname, const char *type);
打开路径名指定的一个文件

FILE *freopen(const char *pathname, const char *type, FILE *fp)
在一个指定的流上面打开一个文件，如果本身流已经打开，就先关闭，如果流已经被定位，就会清除其定位

FILE *fdopen(int fd, const char *type)
将一个文件描述符与标准的io流绑定【不能截断为写而打开的任何文件】

成功，返回文件指针，失败，返回NULL

type	info
r/rb	O_RDONLY
w/wb	O_WRONLY|O_CREAT|O_TRUNC
a/ab	O_WRONLY|O_CREAT|O_APPEND
r+/rb+	O_RDWR
w+/wb+	O_RDWR|O_CREAT|O_TRUNC
a+/ab+	O_RDWR|O_CREAT|O_APPEND

打开流_以读写模式打开一个流的注意事项

1. 如果中间没有fflush 、fseek 、fsetpos 、rewind, 则在输出的后面不能直接跟输入

2 . 如果中间没有fseek 、fsetpos 、rewind, 或者一个输入操作没有到达文件尾，则在输入操作之后不能直接跟随输出

关闭流

1. int fclose(FILE *fp)
成功返回0,失败返回EOF

关闭的时候，冲洗输出数据，丢弃缓冲区中的输入数据

当一个进程正常终止的时候，所有未写的缓冲数据都会被冲洗，所有打开的标准io都被关闭

读写_每次一个字符

int getc(FILE *fp) //宏实现

int fgetc(FILE *fp) //函数实现

int getchar(void) //getc(stdin)

成功，返回下一个字符，失败或者到达文件末尾，返回EOF

为了区分出错还是到达文件末尾

int ferror(FILE *fp)

int feof(FILE *fp)

条件为真，返回非零，否则，返回0
void clearerr(FILE *fp)

清除FILE中的错误标志

压回一个字符到输入流中

int ungetc(int c, FILE *fp)
成功，返回 c, 失败， 返回EOF

不可以回送EOF, 但是在文件到达结尾后，可以回送一个字符，这样可以清楚文件的文件结束标志

读写_每次一个字符

int putc(int c, FILE *fp)　　//宏

int fputc(int c, FILE *fp) //函数

int putchar(int c)　//putc(c, stdout)

成功，返回c, 失败，返回EOF

读写_每次一行

char * fgets(char *buf, int n, FILE *fp)

一直读取到遇到换行符或者读取到的总数为n-1个字符为止，以null字符结尾

char *get(char *buf)

成功，返回buf, 失败或者到达文件尾部，返回NULL

int fputs(const char *str, FILE *fp)

int puts(const char *str)

成功，返回非负值, 出错，返回EOF

两者都是将一个字符串输出，直到遇到空字符，不同的是，puts会在最后添加一个回车

读写_每次指定字节数--对象io

size_t fread(void *ptr, size_t size, size_t nobj, FILE * fp);

size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE *fp);

两个函数返回读或写的对象数目

流定位

long ftell(FILE *fp);
成功，返回当前文件位置指示，出错，返回-1L

off_t ftello(FILE *fp)
成功，返回当前的位置，失败，返回(off_t -1)

会冲洗输出数据

int fseek(FILE *fp, long offset, int whence)
成功，返回 0 ，出错，返回-1

int fseeko(FILE *fp, off_t offset, int whence)
成功，返回 0 ，出错，返回-1

void rewind(FILE *fp)
将一个流设置到文件的起始位置

int fgetpos(FILE *fp, fpos_t *pos);

int fsetpos(FILE *fp, fpos_t *pos);
成功，返回 0 ，　出错，返回非零

将文件位置指示器存入pos对象中，在以后调用fsetpos时，可以使用此值重新定位到该位置

格式化IO_输出

int printf(const char *format, ...)
int fprintf(FILE *fp, const char *format, ...)
int dprintf(int fd, const char *format, ...)

如果成功，返回输出字节数，若输出错误，返回负值

int sprintf(char *buf, const char *format, ...)
成功，返回写到数组中的字符数，失败，返回负值，　尾端添加一个null

int snprintf(char *buf, size_t n, const char *format, ...);
若缓冲区足够大，返回存入数组的字符数，出错，返回负值

格式化IO_输入

int scanf(const char *format, ...)

int fscanf(FILE *fp, const char *format, ...)

int sscanf(const char *buf, const char *format, ...)

提取文件描述符

int fileno(FILE *fp)

返回与该流相关联的文件描述符

临时文件

char *tmpnam(char *ptr)

返回指向唯一路径名的指针

如果ptr为NULL, 那么路径名会存储在一个静态存储区，并返回静态存储区的地址，下一次调用会擦除这个区域

FILE *tmpfile(void)

成功，返回文件指针，失败，返回NULL

创建一个临时的二进制文件，在关闭该文件或者进程结束时，会自动删除

char *mkdtemp(char *template)

成功，返回指向目录名的指针，失败，返回NULL

创建一个具有唯一名称的目录

int mkstemp(char *template)

成功，返回文件描述符，失败，返回-1

创建一个文件，该文件不会自动删除，要手动解除链接

内存流

FILE * fmemopen(void *buf, size_t size, const char * type);

成功，返回流指针，错误，返回NULL

写入内存流以及推进流的内容大小时，null字节会自动追加写，fclose不会

```
initial buffer contents:
before flush:
after flush: hello, world
len of string in buf=12
after fseek: bbbbbbbbbbbbhello, world
len of string in buf=24
before fclose: ccccccccccccccccccccccccccccccccccccccccccccccccc
after fclose: hello, worldccccccccccccccccccccccccccccccccccccc
len of string in buf=46
```

```c
#include<apue.h>

#define BSZ 48

int main()
{
    FILE *fp;
    char buf[BSZ];
    memset(buf, 'a', BSZ-2);
    buf[BSZ-2] = '\0';
    buf[BSZ-1] = 'X';

    if((fp=fmemopen(buf, BSZ, "w+")) == NULL)
        err_exit("fmemopen");
    printf("initial buffer contents: %s\n", buf);
    fprintf(fp, "hello, world");
    printf("before flush: %s\n", buf);
    fflush(fp);

    printf("after flush: %s\n", buf);
    printf("len of string in buf=%ld\n", (long)strlen(buf));

    memset(buf, 'b', BSZ-2);
    buf[BSZ-2] = '\0';
    buf[BSZ-1] = 'X';
    fprintf(fp, "hello, world");
    fseek(fp, 0, SEEK_SET);
    printf("after fseek: %s\n", buf);
    printf("len of string in buf=%ld\n", (long)strlen(buf));

    memset(buf, 'c', BSZ-2);
    buf[BSZ-2] = '\0';
    buf[BSZ-1] = 'X';
    fprintf(fp, "hello, world");
    printf("before fclose: %s\n", buf);
    fclose(fp);
    printf("after fclose: %s\n", buf);
    printf("len of string in buf=%ld\n", (long)strlen(buf));
    return 0;
}
```

内存流

FILE *open_memstream(char **bufp, size_t *size)

FILE *open_wmemstream(wchar_t **bufp, size_t size);

成功，返回流指针，失败，返回NULL

1. 创建的流只能写打开
2. 不能指定自己的缓冲区，但可以分别通过bufp 和sizep参数访问缓冲区地址和大小
3. 关闭流后需要自行释放缓冲区
4. 对流添加字节会增加缓冲区大小

因为缓冲区由于扩充，会进行重新的分配，bufp指向的位置存储的地址会发生变化。