第5章　标 准 I/O 库

5.1　引言

本章说明标准 I/O库。因为不仅在 UNIX而且在很多操作系统上都实现此库，所以它由ANSI C标准说明。标准I/O库处理很多细节，例如缓存分配，以优化长度执行 I/O等。这样使用户不必担心如何选择使用正确的块长度（如3.9节中所述）。标准I/O库是在系统调用函数基础上构造的，它便于用户使用，但是如果不较深入地了解库的操作，也会带来一些问题。

> 标准I/O库是由Dennis Ritchie在1975年左右编写的。它是由 Mike Lesk编写的可移植I/O库的主要修改版本。令人惊异的是， 15年后制订的标准 I/O库对它只作了极小的修改。

5.2　流和FILE对象

在第3章中，所有I/O函数都是针对文件描述符的。当打开一个文件时，即返回一个文件描述符，然后该文件描述符就用于后读的 I/O操作。而对于标准 I/O库，它们的操作则是围绕流（stream）进行的（请勿将标准I/O术语流与系统V的STREAMS I/O系统相混淆）。当用标准I/O库打开或创建一个文件时，我们已使一个流与一个文件相结合。

当打开一个流时，标准I/O函数fopen返回一个指向FILE对象的指针。该对象通常是一个结构，它包含了I/O库为管理该流所需要的所有信息：用于实际 I/O的文件描述符，指向流缓存的指针，缓存的长度，当前在缓存中的字符数，出错标志等等。

应用程序没有必要检验 FILE对象。为了引用一个流，需将 FILE指针作为参数传递给每个标准I/O函数。在本书中，我们称指向FILE对象的指针(类型为FILE *)为文件指针。

在本章中，我们以UNIX系统为例，说明标准I/O库。正如前述，此标准库已移到除UNIX以外的很多系统中。但是为了说明该库实现的一些细节，我们选择UNIX实现作为典型进行介绍。

5.3　标准输入、标准输出和标准出错

对一个进程预定义了三个流，它们自动地可为进程使用：标准输入、标准输出和标准出错。在3.2节中我们曾用文件描述符 STDIN_FILENO,STDOUT_FILENO和STDERR_FILENO分别表示它们。

这三个标准I/O流通过预定义文件指针 stdin,stdout和stderr加以引用。这三个文件指针同样定义在头文件<stdio.h>中。

5.4　缓存

标准I/O提供缓存的目的是尽可能减少使用 read和write调用的数量（见表3-1，其中显示了在不同缓存长度情况下，为执行 I/O所需的CPU时间量）。它也对每个I/O流自动地进行缓存管

理，避免了应用程序需要考虑这一点所带来的麻烦。不幸的是，标准 I/O库令人最感迷惑的也是它的缓存。

标准I/O提供了三种类型的缓存：

(1) 全缓存。在这种情况下，当填满标准I/O缓存后才进行实际I/O操作。对于驻在磁盘上的文件通常是由标准I/O库实施全缓存的。在一个流上执行第一次I/O操作时，相关标准I/O函数通常调用malloc（见7.8节）获得需使用的缓存。

术语刷新（flush）说明标准I/O缓存的写操作。缓存可由标准I/O例程自动地刷新（例如当填满一个缓存时），或者可以调用函数 fflush刷新一个流。值得引起注意的是在 UNIX环境中，刷新有两种意思。在标准 I/O库方面，刷新意味着将缓存中的内容写到磁盘上（该缓存可以只是局部填写的）。在终端驱动程序方面（例如在第 11章中所述的tcflush函数），刷新表示丢弃已存在缓存中的数据。

(2) 行缓存。在这种情况下，当在输入和输出中遇到新行符时，标准 I/O库执行I/O操作。这允许我们一次输出一个字符（用标准 I/O fputc函数），但只有在写了一行之后才进行实际 I/O操作。当流涉及一个终端时（例如标准输入和标准输出），典型地使用行缓存。

对于行缓存有两个限制。第一个是：因为标准 I/O库用来收集每一行的缓存的长度是固定的，所以只要填满了缓存，那么即使还没有写一个新行符，也进行 I/O操作。第二个是：任何时候只要通过标准输入输出库要求从 (a)一个不带缓存的流，或者 (b)一个行缓存的流（它预先要求从内核得到数据）得到输入数据，那么就会造成刷新所有行缓存输出流。在 (b)中带了一个在括号中的说明的理由是，所需的数据可能已在该缓存中，它并不要求内核在需要该数据时才进行该操作。很明显，从不带缓存的一个流中进行输入（ (a)项）要求当时从内核得到数据。

(3) 不带缓存。标准I/O库不对字符进行缓存。如果用标准 I/O函数写若干字符到不带缓存的流中，则相当于用 write系统调用函数将这些字符写至相关联的打开文件上。标准出错流 stderr通常是不带缓存的，这就使得出错信息可以尽快显示出来，而不管它们是否含有一个新行字符。

ANSI C要求下列缓存特征：

(1) 当且仅当标准输入和标准输出并不涉及交互作用设备时，它们才是全缓存的。

(2) 标准出错决不会是全缓存的。

但是，这并没有告诉我们如果标准输入和输出涉及交互作用设备时，它们是不带缓存的还是行缓存的，以及标准输出是不带缓存的，还是行缓存的。 SVR4和4.3+BSD的系统默认使用下列类型的缓存：

- 标准出错是不带缓存的。
- 如若是涉及终端设备的其他流，则它们是行缓存的；否则是全缓存的。

对任何一个给定的流，如果我们并不喜欢这些系统默认，则可调用下列两个函数中的一个更改缓存类型：

```
#include <stdio.h>

void setbuf(FILE *fp, char *buf);

int setvbuf(FILE *fp, char *buf, int mode, size_t size);
```

返回：若成功则为0，若出错则为非0

这些函数一定要在流已被打开后调用（这是十分明显的，因为每个函数都要求一个有效的文件

指针作为它们的第一个参数），而且也应在对该流执行任何一个其他操作之前调用。

可以使用setbuf函数打开或关闭缓存机制。为了带缓存进行I/O，参数*buf*必须指向一个长度为BUFSIZ的缓存（该常数定义在<stdio.h>中）。通常在此之后该流就是全缓存的，但是如果该流与一个终端设备相关，那么某些系统也可将其设置为行缓存的。为了关闭缓存，将*buf*设置为NULL。

使用setvbuf，我们可以精确地说明所需的缓存类型。这是依靠*mode*参数实现的：

```
_IOFBF   全缓存
_IOLBF   行缓存
_IONBF   不带缓存
```

如果指定一个不带缓存的流，则忽略*buf*和*size*参数。如果指定全缓存或行缓存，则*buf*和*size*可以可选择地指定一个缓存及其长度。如果该流是带缓存的，而*buf*是NULL，则标准I/O库将自动地为该流分配适当长度的缓存。适当长度指的是由struct结构中的成员st_blksize所指定的值（见4.2节）。如果系统不能为该流决定此值（例如若此流涉及一个设备或一个管道），则分配长度为BUFSIZ的缓存。

> 伯克利系统首先使用st_blksize表示缓存长度。较早的系统V版本使用标准I/O常数BUFSIZ（其典型值是1024）。即使4.3+BSD使用st_blksize决定最佳的I/O缓存长度，它仍将BUFSIZ设置为1024。

表5-1列出了这两个函数的动作，以及它们的各个选择项。

表5-1 setbuf和setvbuf函数

函 数	*mode*	*buf*	缓存及长度	缓存的类型
setbuf		nonnull	长度为BUFSIZ的用户缓存	全缓存或行缓存
		NULL	（无缓存）	不带缓存
setvbuf	_IOFBF	nonnull	长度为*size*的用户缓存	全缓存
		NULL	合适长度的系统缓存	
	_IOLBF	nonnull	长度为*size*的用户缓存	行缓存
		NULL	合适长度的系统缓存	
	_IONBF	忽略	无缓存	不带缓存

要了解，如果在一个函数中分配一个自动变量类的标准I/O缓存，则从该函数返回之前，必须关闭该流。（7.8节将对此作更多讨论。）另外，SVR4将缓存的一部分用于它自己的管理操作，所以可以存放在缓存中的实际数据字节数少于*size*。一般而言，应由系统选择缓存的长度，并自动分配缓存。在这样处理时，标准I/O库在关闭此流时将自动释放此缓存。

任何时候，我们都可强制刷新一个流。

```
#include<stdio.h>

int fflush(FILE *fp);
```

返回：若成功则为0，若出错则为EOF

此函数使该流所有未写的数据都被传递至内核。作为一种特殊情形，如若 fp是NULL，则此函数刷新所有输出流。

> 传送一个空指针以强迫刷新所有输出流，这是由ANSI C新引入的。非ANSI C库（例如较早的系统V版本和4.3BSD）并不支持此种特征。

5.5 打开流

下列三个函数可用于打开一个标准I/O流。

```
#include <stdio.h>

FILE *fopen(const char *pathname, const char *type);

FILE *freopen(const char *pathname, const char *type, FILE *fp);

FILE *fdopen(int filedes, const char *type);
```

三个函数的返回：若成功则为文件指针，若出错则为 NULL

这三个函数的区别是：

(1) fopen打开路径名由*pathname* 指示的一个文件。

(2) freopen在一个特定的流上(由*fp*指示)打开一个指定的文件（其路径名由*pathname* 指示），如若该流已经打开，则先关闭该流。此函数一般用于将一个指定的文件打开为一个预定义的流：标准输入、标准输出或标准出错。

(3) fdopen取一个现存的文件描述符（我们可能从 open,dup,dup2,fcntl或pipe函数得到此文件描述符），并使一个标准的I/O流与该描述符相结合。此函数常用于由创建管道和网络通信通道函数获得的插述符。因为这些特殊类型的文件不能用标准 I/O fopen函数打开，首先必须先调用设备专用函数以获得一个文件描述符，然后用 fdopen使一个标准I/O流与该描述符相结合。

> fopen和freopen是ANSI C的所属部分。而ANSI C并不涉及文件描述符，所以仅有POSIX.1具有fdopen。

*type*参数指定对该I/O流的读、写方式，ANSI C规定*type*参数可以有15种不同的值，它们示于表5-2中。

表5-2 打开标准I/O流的*type*参数

type	说 明
r 或 rb	为读而打开
w 或 wb	使文件成为0长，或为写而创建
a 或 ab	添加；为在文件尾写而打开，或为写而创建
r+ 或 r+b 或 rb+	为读和写而打开
w+ 或 w+b 或 wb+	使文件为0长，或为读和写而打开
a+ 或 a+b 或 ab+	为在文件尾读和写而打开或创建

使用字符b作为*type*的一部分，使得标准I/O系统可以区分文本文件和二进制文件。因为UNIX内核并不对这两种文件进行区分，所以在UNIX系统环境下指定字符b作为*type*的一部分实际上并无作用。

对于fdopen，*type*参数的意义则稍有区别。因为该描述符已被打开，所以fdopen为写而打开并不截短该文件。(例如，若该描述符原来是由open函数打开的，该文件那时已经存在，则其O_TRUNC标志将决定是否截短该文件。fdopen函数不能截短它为写而打开的任一文件。）另外，标准I/O添加方式也不能用于创建该文件（因为如若一个描述符引用一个文件，则该文件一定已经存在）。

当用添加类型打开一文件后，则每次写都将数据写到文件的当前尾端处。如若有多个进程用标准I/O添加方式打开了同一文件，那么来自每个进程的数据都将正确地写到文件中。

> 4.3+BSD以前的伯克利版本以及Kernighan和Ritchie〔1988〕177页上所示的简单版本并不能正确地处理添加方式。这些版本在打开流时，调用lseek到达文件尾端。在涉及多个进程时，为了正确地支持添加方式，该文件必须用O_APPEND标志打开，我们已在3.3节中对此进行了讨论。在每次写前，做一次lseek操作同样也不能正确工作（如同在3.11节中讨论的一样）。

当以读和写类型打开一文件时（*type*中+号），具有下列限制：

- 如果中间没有fflush、fseek、fsetpos或rewind，则在输出的后面不能直接跟随输入。
- 如果中间没有fseek、fsetpos或rewind,或者一个输出操作没有到达文件尾端，则在输入操作之后不能直接跟随输出。

按照表5-2，我们在表5-3中列出了打开一个流的六种不同的方式。

表5-3 打开一个标准I/O流的六种不同的方式

限 制	r	w	a	r+	w+	a+
文件必须已存在	•			•		
擦除文件以前的内容		•			•	
流可以读	•			•	•	•
流可以写		•	•	•	•	•
流只可在尾端处写			•			•

注意，在指定w或a类型创建一个新文件时，我们无法说明该文件的存取许可权位（第3章中所述的open函数和creat函数则能做到这一点）。POSIX.1要求以这种方式创建的文件具有下列存取许可权：

```
S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH
```

除非流引用终端设备，否则按系统默认，它被打开时是全缓存的。若流引用终端设备，则该流是行缓存的。一旦打开了流，那么在对该流执行任何操作之前，如果希望，则可使用前节所述的setbuf和setvbuf改变缓存的类型。

调用fclose关闭一个打开的流。

```
#include <stdio.h>

int fclose(FILE *fp);
```

返回：若成功则为0，若出错则为EOF

在该文件被关闭之前，刷新缓存中的输出数据。缓存中的输入数据被丢弃。如果标准 I/O库已经为该流自动分配了一个缓存，则释放此缓存。

当一个进程正常终止时（直接调用exit函数，或从main函数返回），则所有带未写缓存数据的标准I/O流都被刷新，所有打开的标准I/O流都被关闭。

5.6 读和写流

一旦打开了流，则可在三种不同类型的非格式化 I/O中进行选择，对其进行读、写操作。（5.11节说明了格式化I/O函数，例如printf和scanf。）

(1) 每次一个字符的I/O。一次读或写一个字符，如果流是带缓存的，则标准I/O函数处理所有缓存。

(2) 每次一行的I/O。使用fgets和fputs一次读或写一行。每行都以一个新行符终止。当调用fgets时，应说明能处理的最大行长。5.7节将说明这两个函数。

(3) 直接I/O。fread和fwrite函数支持这种类型的I/O。每次I/O操作读或写某种数量的对象，而每个对象具有指定的长度。这两个函数常用于从二进制文件中读或写一个结构。 5.9节将说明这两个函数。

> 直接I/O(direct I/O) 这个术语来自ANSI C标准，有时也被称为：二进制 I/O、一次一个对象I/O、面向记录的I/O或面向结构的I/O。

5.6.1 输入函数

以下三个函数可用于一次读一个字符。

```
#include <stdio.h>

int getc(FILE *fp);

int fgetc(FILE *fp);

int getchar(void);
```

三个函数的返回：若成功则为下一个字符，若已处文件尾端或出错则为 EOF

函数getchar等同于getc(stdin)。前两个函数的区别是getc可被实现为宏，而fgetc则不能实现为宏。这意味着：

(1) getc的参数不应当是具有副作用的表达式。

(2) 因为fgetc一定是个函数，所以可以得到其地址。这就允许将 fgetc的地址作为一个参数传送给另一个函数。

(3) 调用fgetc所需时间很可能长于调用 getc,因为调用函数通常所需的时间长于调用宏。检验一下<stdio.h>头文件的大多数实现，从中可见getc是一个宏，其编码具有较高的工作效率。

这三个函数以unsigned char 类型转换为int的方式返回下一个字符。说明为不带符号的理由是，如果最高位为1也不会使返回值为负。要求整型返回值的理由是，这样就可以返回所有可能的字符值再加上一个已发生错误或已到达文件尾端的指示值。在 <stdio.h>中的常数EOF被要

求是一个负值，其值经常是－1。这就意味着不能将这三个函数的返回值存放在一个字符变量中，以后还要将这些函数的返回值与常数EOF相比较。

注意，不管是出错还是到达文件尾端，这三个函数都返回同样的值。为了区分这两种不同的情况，必须调用ferror或feof。

```
#include <stdio.h>

int ferror(FILE *fp);

int feof(FILE *fp);
```

两个函数返回：若条件为真则为非0（真），否则为0（假）

```
void clearerr(FILE *fp);
```

在大多数实现的FILE对象中，为每个流保持了两个标志：

- 出错标志。
- 文件结束标志。

调用clearerr则清除这两个标志。

从一个流读之后，可以调用ungetc将字符再送回流中。

```
#include <stdio.h>

int ungetc(int c, FILE *fp);
```

返回：若成功则为C，若出错则为EOF

送回到流中的字符以后又可从流中读出，但读出字符的顺序与送回的顺序相反。应当了解，虽然ANSI C允许支持任何数量的字符回送的实现，但是它要求任何一种实现都要支持一个字符的回送功能。

回送的字符，不一定必须是上一次读到的字符。 EOF不能回送。但是当已经到达文件尾端时，仍可以回送一字符。下次读将返回该字符，再次读则返回 EOF。之所以能这样做的原因是一次成功的ungetc调用会清除该流的文件结束指示。

当正在读一个输入流，并进行某种形式的分字或分记号操作时，会经常用到回送字符操作。有时需要先看一看下一个字符，以决定如何处理当前字符。然后就需要方便地将刚查看的字符送回，以便下一次调用getc时返回该字符。如果标准I/O库不提供回送能力，就需将该字符存放到一个我们自己的变量中，并设置一个标志以便判别在下一次需要一个字符时是调用 getc，还是从我们自己的变量中取用。

5.6.2 输出函数

对应于上面所述的每个输入函数都有一个输出函数。

```
#include <stdio.h>

int putc(int c, FILE *fp);

int fputc(int c, FILE *fp) ;
```

```
int putchar(int c);
```

三个函数返回：若成功则为C，若出错则为EOF

与输入函数一样，putchar(c) 等同于putc (c, stdout)，putc 可被实现为宏，而fputc 则不能实现为宏。

5.7 每次一行I/O

下面两个函数提供每次输入一行的功能。

```
#include <stdio.h>
char *fgets(char *buf, int n, FILE *fp);
char *gets(char *buf);
```

两个函数返回：若成功则为buf，若已处文件尾端或出错则为NULL

这两个函数都指定了缓存地址，读入的行将送入其中。 gets从标准输入读，而fgets则从指定的流读。

对于fgets，必须指定缓存的长度 n。此函数一直读到下一个新行符为止，但是不超过 $n-1$ 个字符，读入的字符被送入缓存。该缓存以 null字符结尾。如若该行，包括最后一个新行符的字符数超过 $n-1$，则只返回一个不完整的行，而且缓存总是以 null字符结尾。对fgets的下一次调用会继续读该行。

gets是一个不推荐使用的函数。问题是调用者在使用 gets时不能指定缓存的长度。这样就可能造成缓存越界（如若该行长于缓存长度），写到缓存之后的存储空间中，从而产生不可预料的后果。这种缺陷曾被利用，造成 1988年的因特网蠕虫事件。有关说明请见 1989.6. Communications of the ACM（vol.32, no.6）。gets与fgets的另一个区别是，gets并不将新行符存入缓存中。

> 这两个函数对新行符进行处理方面的差别与 UNIX的进展有关。早在V7的手册中就说明：“为了向后兼容，gets删除新行符，而fgets则保持新行符。”

虽然ANSI C要求提供gets,但请不要使用它。

fputs和puts提供每次输出一行的功能。

```
#include <stdio.h>

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

int puts(const char *str);
```

两个函数返回：若成功则为非负值，若出错则为 EOF

函数fputs将一个以null符终止的字符串写到指定的流，终止符 null不写出。注意，这并不一定是每次输出一行，因为它并不要求在 null符之前一定是新行符。通常，在 null符之前是一个新行符，但并不要求总是如此。

puts将一个以null符终止的字符串写到标准输出，终止符不写出。但是， puts然后又将一个新行符写到标准输出。

puts并不像它所对应的gets那样不安全。但是我们还是应避免使用它，以免需要记住它在最后又加上了一个新行符。如果总是使用 fgets和fputs，那么就会熟知在每行终止处我们必须自己加一个新行符。

5.8 标准I/O的效率

使用前面所述的函数，我们应该对标准 I/O系统的效率有所了解。程序 5-1类似于程序3-3，它使用getc和putc将标准输入复制到标准输出。这两个函数可以实现为宏。

程序5-1 用getc和putc将标准输入复制到标准输出

```
#include    "ourhdr.h"

int
main(void)
{
    int     c;

    while ( (c = getc(stdin)) != EOF)
        if (putc(c, stdout) == EOF)
            err_sys("output error");

    if (ferror(stdin))
        err_sys("input error");

    exit(0);
}
```

可以用fgetc和fputc改写该程序，这两个一定是函数，而不是宏（没有给出对源代码更改的细节）。

最后，我们还编写了一个读、写行的版本，见程序5-2。

程序5-2 用fgets和fputs将标准输入复制到标准输出

```
#include    "ourhdr.h"

int
main(void)
{
    char    buf[MAXLINE];

    while (fgets(buf, MAXLINE, stdin) != NULL)
        if (fputs(buf, stdout) == EOF)
            err_sys("output error");

    if (ferror(stdin))
        err_sys("input error");

    exit(0);
}
```

注意，在程序5-1和程序5-2中，没有显式地关闭标准I/O流。我们知道exit函数将会刷新任何未写的数据，然后关闭所有打开的流（我们将在 8.5节讨论这一点）。将这三个程序的时间与表3-1中的时间进行比较是很有趣的。表 5-4中显示了对同一文件（1.5M字节，30,000行）进行操作所得的数据。

表5-4 使用标准I/O例程得到的时间结果

函 数	用户CPU（秒）	系统CPU（秒）	时钟时间（秒）	程序正文字节数
表3-1中的最佳时间	0.0	0.3	0.3	
fgets,fputs	2.2	0.3	2.6	184
getc,putc	4.3	0.3	4.8	384
fgetc,fputc	4.6	0.3	5.0	152
表3-1中的单字节时间	23.8	397.9	423.4	

对于这三个标准I/O版本的每一个，其用户CPU时间都大于表3-1中的最佳read版本，因为每次读一个字符版本中有一个要执行150万次的循环，而在每次读一行的版本中有一个要执行30 000次的循环。在read版本中，其循环只需执行180次（对于缓存长度为8192字节）。因为系统CPU时间都相同，所以用户CPU时间的差别造成了时钟时间的差别。

系统CPU时间相同的原因是因为所有这些程序对内核提出的读、写请求数相同。注意，使用标准I/O例程的一个优点是无需考虑缓存及最佳I/O长度的选择。在使用fgets时需要考虑最大行长，但是最佳I/O长度的选择要方便得多。

表5-4中的最后一列是每个main函数的文本空间字节数（由C编译产生的机器指令）。从中可见，使用getc的版本在文本空间中作了getc和putc的宏代换，所以它所需使用的指令数超过了调用fgetc和fputc函数所需指令数。观察getc版本和fgetc版本在用户CPU时间方面的差别，可以看到在程序中作宏代换和调用两个函数在进行本测试的系统上并没有造成多大差别。

使用每次一行I/O版本其速度大约是每次一个字符版本的两倍（包括用户CPU时间和时钟时间）。如果fgets和fputs函数用getc和putc实现（例如，见Kernighan和Ritchie〔1988〕的7.7节），那么，可以预期fgets版本的时间会与getc版本相接近。实际上，可以预料每次一行的版本会更慢一些，因为除了现已存在的60 000次函数调用外还需增加3百万次宏调用。而在本测试中所用的每次一行函数是用memccpy(3)实现的。通常，为了提高效率，memccpy函数用汇编语言而非C语言编写。

这些时间数字的最后一个有趣之处在于：fgetc版本较表3-1中BUFFSIZE＝1的版本要快得多。两者都使用了约3百万次的函数调用，而fgetc版本的速度在用户CPU时间方面，大约是后者的5倍，而在时钟时间方面则几乎是100倍。造成这种差别的原因是：使用read的版本执行了3百万次函数调用，这也就引起3百万次系统调用。而对于fgetc版本，它也执行3百万次函数调用，但是这只引起360次系统调用。系统调用与普通的函数调用相比是很花费时间的。

需要声明的是这些时间结果只在某些系统上才有效。这种时间结果依赖于很多实现的特征，而这种特征对于不同的UNIX系统却可能是不同的。尽管如此，使这样一组数据，并对各种版本的差别作出解释，这有助于我们更好地了解系统。

在本节及3.9节中我们学到的基本事实是：标准I/O库与直接调用read和write函数相比并不慢很多。我们观察到使用getc和putc复制1M字节数据大约需3.0秒CPU时间。对于大多数比较复杂的应用程序，最主要的用户CPU时间是由应用本身的各种处理花费的，而不是由标准I/O例程消耗的。

5.9 二进制I/O

5.6节中的函数以一次一个字符或一次一行的方式进行操作。如果为二进制I/O，那么我们更愿意一次读或写整个结构。为了使用getc或putc做到这一点，必须循环通过整个结构，一次

读或写一个字节。因为fputs在遇到null字节时就停止，而在结构中可能含有null字节，所以不能使用每次一行函数实现这种要求。相类似，如果输入数据中包含有 null字节或新行符，则fgets也不能正确工作。因此，提供了下列两个函数以执行二进制I/O操作。

```
#include <stdio.h>

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);
```

两个函数的返回：读或写的对象数

这些函数有两个常见的用法：

(1) 读或写一个二进制数组。例如，将一个浮点数组的第 2至第5个元素写至一个文件上，可以写作：

```
float   data[10];

if (fwrite(&data[2], sizeof(float), 4, fp) != 4)
   err_sys("fwrite error");
```

其中，指定*size*为每个数组元素的长度，*nobj*为欲写的元素数。

(2) 读或写一个结构。例如，可以写作：

```
 struct {
  short count;
  long  total;
  char  name[NAMESIZE];
} item;

if (fwrite(&item, sizeof(item), 1, fp) != 1)
   err_sys("fwrite error");
```

其中，指定*size*为结构的长度，*nobj*为1（要写的对象数）。

将这两个例子结合起来就可读或写一个结构数组。为了做到这一点， *size*应当是该结构的sizeof,*nobj*应是该数组中的元素数。

fread和fwrite返回读或写的对象数。对于读，如果出错或到达文件尾端，则此数字可以少于*nobj*。在这种情况，应调用ferror或feof以判断究竟是那一种情况。对于写，如果返回值少于所要求的*nobj*，则出错。

使用二进制I/O的基本问题是，它只能用于读已写在同一系统上的数据。多年之前，这并无问题（那时，所有UNIX系统都运行于PDP-11上），而现在，很多异构系统通过网络相互连接起来，而且，这种情况已经非常普遍。常常有这种情形，在一个系统上写的数据，在另一个系统上处理。在这种环境下，这两个函数可能就不能正常工作，其原因是：

(1) 在一个结构中，同一成员的位移量可能随编译程序和系统的不同而异（由于不同的对准要求）。确实，某些编译程序有一选择项，它允许紧密包装结构（节省存储空间，而运行性能则可能有所下降）或准确对齐，以便在运行时易于存取结构中的各成员。这意味着即使在单一系统上，一个结构的二进制存放方式也可能因编译程序的选择项而不同。

(2) 用来存储多字节整数和浮点值的二进制格式在不同的系统结构间也可能不同。

在不同系统之间交换二进制数据的实际解决方法是使用较高层次的协议。关于网络协议使用的交换二进制数据的某些技术，请参阅Stevens〔1990〕的18.2节。

在8.13节中，我们将再回到fread函数，那时将用它读一个二进制结构——UNIX的进程记账记录。

5.10 定位流

有两种方法定位标准I/O流。

(1) ftell和fseek。这两个函数自V7以来就存在了，但是它们都假定文件的位置可以存放在一个长整型中。

(2) fgetpos和fsetpos。这两个函数是新由ANSI C引入的。它们引进了一个新的抽象数据类型fpos_t，它记录文件的位置。在非UNIX系统中，这种数据类型可以定义为记录一个文件的位置所需的长度。

需要移植到非UNIX系统上运行的应用程序应当使用fgetpos和fsetpos。

```
#include <stdio.h>

long ftell(FILE *fp);
```

返回：若成功则为当前文件位置指示，若出错则为－1L

```
int fseek(FILE *fp, long offset, int whence);
```

返回：若成功则为0，若出错则为非0

```
void rewind(FILE *fp);
```

对于一个二进制文件，其位置指示器是从文件起始位置开始度量，并以字节为计量单位的。ftell用于二进制文件时，其返回值就是这种字节位置。为了用fseek定位一个二进制文件，必须指定一个字节*offset*，以及解释这种位移量的方式。*whence*的值与3.6节中lseek函数的相同：SEEK_SET表示从文件的起始位置开始，SEEK_CUR表示从当前文件位置，SEEK_END表示从文件的尾端。ANSI C并不要求一个实现对二进制文件支持SEEK_END规格说明，其原因是某些系统要求二进制文件的长度是某个幻数的整数倍，非实际内容部分则充填为0。但是在UNIX中，对于二进制文件SEEK_END是得到支持的。

对于文本文件，它们的文件当前位置可能不以简单的字节位移量来度量。再一次，这主要也是在非UNIX系统中，它们可能以不同的格式存放文本文件。为了定位一个文本文件，*whence*一定要是SEEK_SET，而且*offset*只能有两种值：0（表示反绕文件至其起始位置），或是对该文件的ftell所返回的值。使用rewind函数也可将一个流设置到文件的起始位置。

正如我们已提及的，下列两个函数是C标准新引进的。

```
#include <stdio.h>

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

int fsetpos(FILE *fp, const fpos_t *pos);
```

两个函数返回：若成功则为0，若出错则为非0

fgetpos将文件位置指示器的当前值存入由*pos*指向的对象中。在以后调用fsetpos时，可以使用此值将流重新定位至该位置。

5.11 格式化I/O

5.11.1 格式化输出

执行格式化输出处理的是三个printf函数。

```
#include <stdio.h>

int printf(const char *format, ...);

int fprintf(FILE *fp, const char *format, ...);
```

两个函数返回：若成功则为输出字符数，若输出出错则为负值

```
int sprintf(char *buf, const char *format, ...);
```

返回：存入数组的字符数

printf将格式化数据写到标准输出，fprintf写至指定的流，sprintf将格式化的字符送入数组*buf*中。sprintf在该数组的尾端自动加一个null字节，但该字节不包括在返回值中。

> 4.3BSD定义sprintf返回其第一个参数（缓存指针，类型为 char*），而不是一个整型。ANSI C要求sprintf返回一个整型。
>
> 注意，sprintf可能会造成由*buf*指向的缓存的溢出。保证该缓存有足够长度是调用者的责任。

对这三个函数可能使用的各种格式变换，请参阅UNIX手册，或Kernighan和Ritchie〔1988〕的附录B。

下列三种printf族的变体类似于上面的三种，但是可变参数表(...)代换成了*arg*。

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

int vprintf(const char *format, va_list arg);

int vfprintf(FILE *fp, const char *format, va_list arg);
```

两个函数返回：若成功则为输出字符数，若输出出错则为负值

```
int vsprintf(char *buf, const char *format, va_list arg);
```

返回：存入数组的字符数

在附录B的出错例程中，将使用vsprintf函数。

关于ANSI C标准中有关可变长度参数表的详细说明请参阅 Kernighan和Ritchie〔1988〕的7.3节。应当了解的是，由ANSI C提供的可变长度参数表例程（<stdarg.h>头文件和相关的例程）与由SVR3（以及更早版本）和4.3BSD提供的<varargs.h>例程是不同的。

5.11.2 格式化输入

执行格式化输入处理的是三个scanf函数。

```
#include<stdio.h>

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

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

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

三个函数返回：指定的输入项数，若输入出错，或在任意变换前已至文件尾端则为 EOF

如同printf族一样，关于这三个函数的各个格式选择项的详细情况，请参阅 UNIX手册。

5.12 实现细节

正如前述，在UNIX中，标准I/O库最终都要调用第3章中说明的I/O例程。每个I/O流都有一个与其相关联的文件描述符，可以对一个流调用 fileno以获得其描述符。

```
#include <stdio.h>
int fileno(FILE *fp);
```

返回：与该流相关联的文件描述符

如果要调用dup或fcntl等函数，则需要此函数。

为了了解你所使用的系统中标准 I/O库的实现，最好从头文件 <stdio.h>开始。从中可以看到：FILE对象是如何定义的，每个流标志的定义，定义为宏的各个标准 I/O例程（例如getc）。Kernighan和Ritchie〔1988〕中的8.5节含有一个简单的实现，从中可以看到很多 UNIX实现的基本样式。Plauger〔1992〕的第12章提供了标准I/O库一种实现的全部源代码。4.3＋BSD中标准I/O库的实现（由Chris Torek编写）也是可以公开使用的。

实例

程序5-3为三个标准流以及一个与一个普通文件相关联的流打印有关缓存状态信息。注意，在打印缓存状态信息之前，先对每个流执行 I/O操作，因为第一个I/O操作通常就造成为该流分配缓存。结构成员_flag、_bufsiz以及常数_IONBF和_IOLBF是由作者所使用的系统定义的。

如果运行程序5-3两次，一次使三个标准流与终端相连接，另一次使它们重定向到普通文件，则所得结果是：

```
$ a.out                                          stdin, stdout 和stderr 都连至终端
enter any character
                                                 键入新行符
one line to standard error
stream = stdin, line buffered, buffer size = 128
stream = stdout, line buffered, buffer size = 128
stream = stderr, unbuffered, buffer size = 8
stream = /etc/motd, fully buffered, buffer size = 8192
$ a.out < /etc/termcap > std.out 2> std.err
                                                 三个流都重定向，再次运行该程序
$ cat std.err
one line to standard error
$ cat std.out
enter any character
stream = stdin, fully buffered, buffer size = 8192
```

```
stream = stdout, fully buffered, buffer size = 8192
stream = stderr, unbuffered, buffer size = 8
stream = /etc/motd, fully buffered, buffer size = 8192
```

程序5-3 对各个标准I/O流打印缓存状态信息

```
#include    "ourhdr.h"

void    pr_stdio(const char *, FILE *);

int
main(void)
{
    FILE    *fp;

    fputs("enter any character\n", stdout);
    if (getchar() == EOF)
        err_sys("getchar error");
    fputs("one line to standard error\n", stderr);

    pr_stdio("stdin",  stdin);
    pr_stdio("stdout", stdout);
    pr_stdio("stderr", stderr);

    if ( (fp = fopen("/etc/motd", "r")) == NULL)
        err_sys("fopen error");
    if (getc(fp) == EOF)
        err_sys("getc error");
    pr_stdio("/etc/motd", fp);
    exit(0);
}

void
pr_stdio(const char *name, FILE *fp)
{
    printf("stream = %s, ", name);
                /* following is nonportable */
    if      (fp->_flag & _IONBF)    printf("unbuffered");
    else if (fp->_flag & _IOLBF)    printf("line buffered");
    else /* if neither of above */  printf("fully buffered");
    printf(", buffer size = %d\n", fp->_bufsiz);
}
```

从中可见，该系统的默认是：当标准输入、输出连至终端时，它们是行缓存的。行缓存的长度是128字节。注意，这并没有将输入、输出的行长限制为128字节，这只是缓存的长度。如果要将512字节的行写到标准输出则要进行四次write系统调用。当将这两个流重新定向到普通文件时，它们就变成是全缓存的，其缓存长度是该文件系统优先选用的I/O长度（从stat结构中得到的st_blksize）。从中也可看到，标准出错如它所应该的那样是非缓存的，而普通文件按系统默认是全缓存的。

5.13 临时文件

标准I/O库提供了两个函数以帮助创建临时文件。

```
#include<stdio.h>

char *tmpnam(char*ptr*);
```

返回：指向一唯一路径名的指针

```
FILE *tmpfile(void);
```

返回：若成功则为文件指针，若出错则为NULL

tmpnam产生一个与现在文件名不同的一个有效路径名字符串。每次调用它时，它都产生一个不同的路径名，最多调用次数是TMP_MAX。TMP_MAX定义在<stdio.h>中。

> 虽然TMP_MAX是由ANSI C定义的。但该C标准只要求其值至少应为25。但是，XPG3却要求其值至少为10 000。在此最小值允许一个实现使用4位数字作为临时文件名的同时（0000~9999），大多数UNIX实现使用的却是大、小写字符。

若*ptr*是NULL，则所产生的路径名存放在一个静态区中，指向该静态区的指针作为函数值返回。下一次再调用tmpnam时，会重写该静态区。（这意味着，如果我们调用此函数多次，而且想保存路径名，则我们应当保存该路径名的副本，而不是指针的副本。）如若*ptr*不是NULL，则认为它指向长度至少是L_tmpnam个字符的数组。（常数L_tmpnam定义在头文件<stdio.h>中。）所产生的路径名存放在该数组中，*ptr*也作为函数值返回。

tmpfile创建一个临时二进制文件（类型wb+），在关闭该文件或程序结束时将自动删除这种文件。注意，UNIX对二进制文件不作特殊区分。

实例

程序5-4说明了这两个函数的应用。若执行程序5-4，则得：

```
$ a.out
/usr/tmp/aaaa00470
/usr/tmp/baaa00470
one line of output
```

加到临时文件名中的5位数字后缀是进程ID，这就保证了对各个进程产生的路径名各不同。

tmpfile函数经常使用的标准UNIX技术是先调用tmpnam产生一个唯一的路径名，然后立即unlink它。

程序5-4　tmpnam和tmpfile函数实例

```
#include    "ourhdr.h"

int
main(void)
{
    char    name[L_tmpnam], line[MAXLINE];
    FILE    *fp;

    printf("%s\n", tmpnam(NULL));       /* first temp name */

    tmpnam(name);                       /* second temp name */
    printf("%s\n", name);

    if ( (fp = tmpfile()) == NULL)      /* create temp file */
        err_sys("tmpfile error");
    fputs("one line of output\n", fp);  /* write to temp file */
    rewind(fp);                         /* then read it back */
    if (fgets(line, sizeof(line), fp) == NULL)
        err_sys("fgets error");
    fputs(line, stdout);                /* print the line we wrote */
```

```
    exit(0);
}
```

请回忆4.15节，对一个文件解除连接并不删除其内容，关闭该文件时才删除其内容。tempnam是tmpnam的一个变体，它允许调用者为所产生的路径名指定目录和前缀。

```
#include <stdio.h>

char    *tempnam(const char *directory, const char *prefix);
```

返回：指向一唯一路径名的指针

对于目录有四种不同的选择，并且使用第一个为真的作为目录：

(1) 如果定义了环境变量TMPDIR，则用其作为目录。（在7.9节中将说明环境变量。）

(2) 如果参数*directory*非NULL，则用其作为目录。

(3) 将<stdio.h>中的字符串P_tmpdir用作为目录。

(4) 将本地目录，通常是/tmp,用作为目录。

如果*prefix*非NULL，则它应该是最多包含5个字符的字符串，用其作为文件名的头几个字符。

该函数调用malloc函数分配动态存储区，用其存放所构造的路径名。当不再使用此路径名时就可释放此存储区（7.8节将说明malloc和fiee函数）。

> tempnam不是POSIX.1和ANSI C的所属部分，它是XPG3的所属部分。
>
> 我们所说明的实现对应于SVR4和4.3+BSD。XPG3版本除了不支持环境变量TMPDIR，其他都与此相同。

实例

程序5-5显示了tempnam的应用。

程序5-5 tempnam函数的应用

```
#include    "ourhdr.h"

int
main(int argc, char *argv[])
{
    if (argc != 3)
        err_quit("usage: a.out <directory> <prefix>");

    printf("%s\n", tempnam( argv[1][0] != ' ' ? argv[1] : NULL,
                            argv[2][0] != ' ' ? argv[2] : NULL) );
    exit(0);
}
```

注意，如果命令行参数（目录或前缀）中的任一一个以空白开始，则将其作为null指针传送给该函数。下面显示使用该程序的各种方式。

```
$ a.out /home/stevens TEMP      指定目录和前缀
/home/stevens/TEMPAAAa00571
$ a.out " " PFX                 使用默认目录：P_tmpdir
/usr/tmp/PFXAAAa00572
$ TMPDIR=/tmp a.out /usr/tmp    使用环境变量；无前缀
```

```
/tmp/AAAa00573                  环境变量复设目录
$ TMPDIR=/no/such/dir a.out/tmp QQQQ
/tmp/QQQQAAAa00574              忽略无效环境目录
$ TMPDIR=/no/such/file a.out /etc/uucp MMMMM
/usr/tmp/MMMMMAAAa00575         忽略无效环境和无效目录两者
```

上述选择目录名的四个步骤按序执行，该函数也检查相应的目录名是否有意义。如果该目录并不存在（例如/no/such/dir），或者对该目录并无写许可权（例如 /etc/uucp）,则跳过这些，试探对目录名的下一次选择。从本例中可以看出在路径名中如何使用进程 ID，也可看出在本实现中，P_tmpdir目录是/usr/tmp。设置环境变量的技术（程序名前的 TMPDIR=）适用于Bourne shell和KornShell。

5.14 标准I/O的替代软件

标准I/O库并不完善。Korn和Vo〔1991〕列出了它的很多不足之处——某些属于基本设计，但是大多数则与各种不同的实现有关。

在标准I/O库中，一个效率不高的不足之处是需要复制的数据量。当使用每次一行函数fgets和fputs时，通常需要复制两次数据：一次是在内核和标准 I/O缓存之间（当调用 read和write时），第二次是在标准 I/O缓存和用户程序中的行缓存之间。快速 I/O库〔AT&T 1990a中的fio(3)〕避免了这一点，其方法是使读一行的函数返回指向该行的指针，而不是将该行复制到另一个缓存中。 Hume〔1988〕报告了由于作了这种更改， grep(1)公用程序的速度增加了2倍。

Korn和Vo〔1991〕说明了标准I/O库的另一种代替版：*sfio*。这一软件包在速度上与*fio*相近，通常快于标准I/O库。*sfio*也提供了一些新的特征：推广了 I/O流，使其不仅可以代表文件，也可代表存储区；可以编写处理模块，并以栈方式将其压入 I/O流，这样就可以改变一个流的操作；较好的异常处理等。

Krieger, Stumm和Unrau〔1992〕说明了另一个代换软件包，它使用了映照文件——mmap函数，我们将在12.9节中说明此函数。该新软件包称为 ASI(Alloc Stream Interface)。其程序界面类似于UNIX存储分配函数（malloc, realloc和free，这些将在7.8节中说明）。与*sfio*软件包相同，ASI使用指针力图减少数据复制量。

5.15 小结

大多数UNIX应用程序都使用标准I/O库。本章说明了该库提供的所有函数，某些实现细节和效率方面的考虑。应该看到标准 I/O库使用了缓存机制，而这种机制是产生很多问题，引起很多混淆的一个领域。

习题

5.1 用setvbuf完成setbuf。

5.2 5.8节中程序利用fgets和fputs函数拷贝文件，每次I/O操作只拷贝一行。若将程序中的MAXLINE改为4，当拷贝的行超过该最大值时会出现什么情况？

5.3 printf返回0值表示什么？

5.4 下面的代码在一些机器上运行正确，而在另外一些机器运行时出错，解释问题所在。

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

```
int
main(void)
{
    char    c;
    while ( ( c = getchar()) != EOF )
        putchar (c);
}
```

5.5 为什么tempnam限制前缀为5个字符？

5.6 对标准I/O流如何使用fsync函数（见4.24节）？

5.7 在程序1-5和1-8中打印的提示信息没有包含换行符，程序也没有调用 fflush函数，请解释提示信息是如何输出的？