北京希望电脑公司计算机技术丛书

UNIX 系统 V 实用程序设计方法

战晓苏　赵立军　苏雷　编著

科学出版社

北京希望电脑公司计算机技术丛书

UNIX 系统 V 实用程序设计方法

战晓苏　赵立军　苏　雷　编著
傅达成　何　力　　审

科学出版社

1993·北京

内 容 简 介

本书重点讲述UNIX系统V的程序设计技巧与方法，本书的特点是注重实用。全书共十章，内容包括共享库的定义、建立、使用与维护；C语言程序检验器lint；各种性能分析工具，如time，prof，gprof，tcou等；源代码控制系统SCCS；make用户指南；宏处理程序m4的使用与编程；用于产生语法分析器的生成器yacc；curses库及函数等。附录A介绍了SCCS低级命令，附录B总结了make的改进。

本书内容由浅入深，并给出了较为详尽的例子。

本书起点较高，需要读者对UNIX系统V和C语言有一定的了解。

欲要本书的用户可直接与北京8721信箱联系，邮编100080，电话2562329

UNIX系统V实用程序设计方法

战晓苏　赵立军　苏　雷　编著

傅达成　何　力　　审

责任编辑　那莉莉

科学出版社出版

北京东黄城根北街16号

邮政编码：100717

北京地质矿产局印刷厂印刷

新华书店北京发行所发行　各地新华书店经售

*

1993年10月第一版　开本：787×1092　1/16

1993年10月第一次印刷　印张：18　3/4

印数：1—5000　字数：431 000字

ISBN 7—03—003956—4/TP·317

定价：13.00元

前　言

用“日新月异”来形容计算机的发展可以说恰如其分。但 UNIX 系统和 C 语言却在这种迅猛的发展中经受住了考验，真正保持长盛不衰。前者已成为大中小型机的标准操作系统，而后者则广泛地应用在各种计算机系统上。

UNIX 操作系统功能之强大，令人叹为观止。用户希望能尽早地掌握 UNIX 的系统开发方法。为此，编者很早就萌发了编写一本介绍 UNIX 系统Ⅴ程序设计的书，以满足这方面的需要。在参考各种 UNIX 系统的书籍和各种随机手册的基础上，编者结合大量的上机经验，完成了本书。

实用是这本书最显著的特色，书中对例子的讲述尽量详细，内容由浅入深，便于学习和掌握。需要提醒读者的是，本书的起点较高，需要读者对 UNIX 系统Ⅴ和 C 语言有一定的了解。本书的重点放在 UNIX 系统Ⅴ程序设计的技巧与方法上，内容包括共享库的定义、建立、使用和维护；C 语言程序检验器 lint；各种性能分析工具，如 time,prof,gprof,tcov 等；源代码控制系统 SCCS；make 用户指南；宏处理程序 m4 的使用与编程；用于产生语法分析器的分析器 yacc；curses 库及函数等。附录 A 介绍了 SCCS 低级命令，附录 B 总结了 make 改进。

本书由战晓苏、赵立军、苏雷编写。傅达成、何力、李晓敏、袁鹰审。

本书的产生凝结了许多人的辛勤劳动，刘立敏、周岩、余自强、田静、左婷、范德伟做了大量的辅助工作，特别是王冰冰为本书初稿的形成做了极大的努力，对此，我们一并表示谢意。

我们很想以此书结交更多的朋友，也希望这本书给读者带来益处。希望广大读者不吝指出本书的错误之处，我们非常感谢。

1993 年 · 北京

目　录

第一章　UNIX 系统 V 编程技巧 …… 1
1.1　概述 …… 1
1.2　基础：程序参数 …… 1
1.3　标准输入与标准输出 …… 2
1.4　标准 I／O 库 …… 3
1.5　低级 I／O 函数 …… 6
1.6　进程 …… 10
1.7　信号——中断及其它 …… 15
1.8　标准 I／O 库 …… 18
第二章　共享库 …… 27
2.1　定义 …… 27
2.2　使用共享库 …… 28
2.3　版本控制 …… 32
2.4　共享库机制 …… 33
2.5　构造共享库 …… 35
2.6　构造一个更好的库 …… 36
2.7　共享库问题 …… 38
第三章　lint——C 语言的程序检验器 …… 39
3.1　使用 lint …… 39
3.2　一个术语 …… 39
3.3　未用变量和函数 …… 40
3.4　定义／引用信息 …… 40
3.5　控制源 …… 40
3.6　函数值 …… 41
3.7　类型检查 …… 42
3.8　强制类型 …… 42
3.9　不可移植字符使用 …… 42
3.10　长整型到整型的赋值 …… 43
3.11　奇怪的句法构造 …… 43
3.12　指针对准 …… 44
3.13　多重使用和副作用 …… 44
3.14　实现 …… 44
3.15　移植性 …… 45
3.16　关闭 lint …… 46

3.17 库说明文件 …… 47
3.18 使用 lint 的考虑 …… 47
3.19 lint 选项 …… 48
第四章 性能分析 …… 49
4.1 time——显示程序使用的时间 …… 49
4.2 prof——产生程序的剖面图 …… 51
4.3 gprof——产生调用图剖面图 …… 53
4.4 tcov——语句级分析 …… 55
第五章 SCCS——源代码控制系统 …… 58
5.1 术语 …… 60
5.2 用 sccs create 产生 SCCS 历史文件 …… 61
5.3 用 sccs get 提取当前版本 …… 62
5.4 改动文件(产生 delta) …… 62
5.5 恢复旧版本 …… 66
5.6 审记改动 …… 66
5.7 速写记号 …… 67
5.8 在工程项目中使用 SCCS …… 68
5.9 保护措施 …… 69
5.10 用 sccs admin 管理 SCCS 文件 …… 69
5.11 维护不同的版本(分枝) …… 70
5.12 SCCS 快速浏览 …… 71
第六章 make 用户指南 …… 74
6.1 概述 …… 74
6.2 用 make 编译程序 …… 88
6.3 构造目标库 …… 98
6.4 用 make 维护库和程序 …… 99
6.5 维护软件工程项目 …… 124
第七章 m4——宏处理程序 …… 136
7.1 使用 m4 命令 …… 136
7.2 定义宏 …… 136
7.3 引用与注释 …… 137
7.4 带参数的宏 …… 139
7.5 内部运算 …… 139
7.6 文件操作 …… 140
7.7 运行 UNIX 操作系统命令 …… 141
7.8 条件 …… 141
7.9 串操作 …… 141
7.10 打印 …… 142
7.11 内部定义宏总结 …… 143

第八章 yacc——分析器的生成器 …… 144
8.1 基本的规范说明 …… 146
8.2 动作 …… 147
8.3 词法分析 …… 149
8.4 分析器如何工作 …… 150
8.5 二义性与冲突 …… 154
8.6 优先级 …… 157
8.7 出错处理 …… 159
8.8 yacc 环境 …… 160
8.9 准备规范说明的提示 …… 161
8.10 高级讨论 …… 163
8.11 一个简单的例子 …… 165
8.12 yacc 输入语法 …… 167
8.13 高级的例子 …… 169
8.14 支持而不提倡的老特性 …… 174
第九章 curses 库 …… 176
9.1 变量 …… 178
9.2 程序设计 curses …… 178
9.3 光标动作优化 FS …… 180
9.4 curses 函数 …… 181
9.5 termcap 中的能力 …… 191
9.6 WINDOW 结构 …… 193
9.7 举例 …… 194
第十章 系统 V curses 和 terminfo …… 198
10.1 概述 …… 198
10.2 用 curses 例程工作 …… 201
10.3 用 termin fo 例程工作 …… 220
10.4 用 terminfo 数据库工作 …… 224
10.5 curses 程序举例 …… 230
第附录 A SCCS 低级命令 …… 244
A.1 给初学者的低级 SCCS …… 244
A.2 SCCS 文件编号习惯 …… 245
A.3 SCCS 命令总结 …… 247
A.4 SCCS 命令习惯 …… 248
A.5 admin——产生及管理 SCCS 文件 …… 249
A.6 cdc——修改 delta 注释 …… 254
A.7 comb——合并 SCCS delta …… 255
A.8 delta 命令——产生一个 delta …… 256
A.9 get——取得 SCCS 文件的版本 …… 260

A.10 help——求求 SCCS 帮助 …… 270
A.11 prs——打印 SCCS 文件 …… 271
A.12 rmdel——从 SCCS 文件删除 delta …… 274
A.13 sact——显示 SCCS 编辑流动 …… 275
A.14 sccsdiff——显示在 SCCS 版本中的差异 …… 275
A.15 unget——取消以前的 SCCS get 命令 …… 276
A.16 val——验证 SCCS 文件 …… 276
A.17 SCCS 文件 …… 278
第附录 B make 改进总结 …… 281
B.1 新特性 …… 281
B.2 同以前的 make 版本不兼容 …… 282

第一章　UNIX 系统 V 编程技巧

1.1　概述

本书介绍了 UNIX 系统 V 程序设计环境,包括各种工具、实用命令和库函数等,对应用程序开发人员来说,这些内容是很有用的。

本书的第一章和第二章描述对 C 语言的系统接口和支持措施，接下来的两章描述 C 程序设计的辅助工具,即检查正确性和监视性能，第五章和第六章描述用于版本控制和一致性编译的系统实用程序，第五章和第八章介绍程序产生工具，最后两章描述 BSD 和系统 V curses 终端显示库例程。

附录 A 描述了 SCCS 的低级命令.

附录 B 总结了对 make 的 UNIX 系统 V 版本所做的改进。

本章的其余各节介绍了如何在 UNIX 系统 V 上进行程序设计,重点是如何在程序中使用系统调用及库函数,由于我们仅涉及到了那些比较有用的东西,所以不是很全面。我们假定读者已掌握了 C 语言,而且熟悉 UNIX 系统 V 操作系统。

1.2　基础:程序参数

当 C 程序作为一条命令运行时,命令行中的参数可以在 main()函数中得到。argc 表示参数个数,指针数组 argv 指向含有参数的字符串。习惯上,argv[o]是命令名本身,所以 argc 总是大于 0.

下面的程序说明这一机制。该程序仅简单地把参数返回到终端--实际上就是 echo() 命令。

```
main(argc,argv)     /* echo arguments */
int argc;
char *argv[ ];
{
    int i;

    for (i = 1; i<argc; i++)
        printf("%s%c", argv[i], (i<argc-1) ? ' ' : '\n');
}
```

argv 是一个指向数组的指针,数组的元素是指向字符数组的指针,每个字符数组以\0结束,所以可以按字符串来处理。该程序首先打印 argv[1],循环,直到打印完了 argv[argc-1].

参数个数及参数是 main()的形参,如果想让其它的子程序得到这两个形参,应把它们拷

贝到全局变量。

1.3 标准输入与标准输出

最简单的输入机制是从标准输入中读,标准输入一般就是用户的终端。函数 getchar()在每次调用时都返回下一个输入字符。若使用 <(输入重定向),可以用一个文件代替终端:如果 prog 用了 getchar(),命令行

```
tutorial% prog < filename
```

就可以令 grog 从指定的文件 filename 中读,不会再从终端上读。prog 本身不必知道输入究竟是从哪儿来,同样,输入还可以通过管道(pipe)机制来自于另一个程序。

```
tutorial% otherprog | prog
```

把 otherprog 的标准输出提供给 prog,作为它的标准输入。

getchar()如果遇到了文件结束(或错误)则返回值 EOF,EOF 通常定义成-1,但最好别直接用-1。下面我们将会看到,当编译程序时,这个值是自动定义的,你不必关心。

类似地,puthchar(c)把字符 c 放到"标准输出",标准输出通常也是终端。输出也可以放到一个文件上,只需使用 >。如果 prog 使用了 putchar(),则把输出写到了 outputfile 上(代替了终端);如果 outputfile 不存在,则产生这个文件;如果存在,则覆盖以前的内容。可以用管道

```
tutorial% prog | otherprog
```

把 prog 的标准输出送到 otherprog 的标准输入。

函数 printf()是格式输出,其机制和 putchar()相同,所以,printf()和 putchar()可以混合使用,输出将按照它们的调用次序出现。

类似地,函数 scanf()提供了格式化的输入转换,该函数读标准输入,根据需要把输入分解成字符串、数字等。scanf()使用了和 getchar()相同的机制,所以对它们的调用亦可混用。

许多程序仅读一个输入和写一个输出,对这样的程序,使用 getchar(),putchar(),scanf()和 printf()进行 I/O 就足够了,而且只要启动程序就能完成。如果用 UNIX 系统 V 管道设施把一个程序的输出连接到下一个程序的输入,这种做法尤为可取。例如,下面的程序是从输入中挑出所有 ASCII 控制字符(除换行符和制表符外):

```
#include <stdio.h>

main() / * ccstrip: strip non-graphic characters * /
{
    int c;
    while ((c = getchar()) != EOF)
        if ((c >= ' ' && c< 0177) || c == '\t' || c == '\n'
            putchar(c);
    exit(0);
}
```

如果源文件使用标准I/O函数进行I/O,那么行

```
#inclnde <stdio.h>
```

应该出现在源文件的开始,C编译器读标准例程和符号的文件(/usr/include/stdio.h),该文件包括了EOF的定义。

如果需要处理多个文件,可以用cat来收集文件:

```
tutorial% cat file1 file2 ... | ccstrip > output
```

这样你就无需知道如何在程序中访问文件。顺便提一下,为了让程序正常工作,不必在最后调用exit()。但这样做可以保证程序的调用者看到程序结束时的正常结束状态(一般是0),1.6.3节详细地介绍了返回状态。

1.4 标准I/O库

标准I/O库是例程(routine)的集合,这些例程为大多数C程序提供了有效的和可移植的I/O服务。任何支持C的系统都有标准I/O库,所以只要程序把与系统的交互限制这个库中的例程,就可心从一个系统传送到另一个系统,本质上无需改变。

本节讨论标准I/O库的基础,1.8节"标准I/O库"完整地描述了这个库的能力和调用习惯。

1.4.1 访问文件

上面的程序全部是从标准输入读和写,我们假定它们已事先定义好了。下面我们要写一个这样的程序,该程序访问一个还没有连上的文件。例如,wc统计若干文件的行数、字数和字符数,命令

```
tutorial% wc x.c y.c
```

将显示x.c和y.c的行数、字数、字符数及总数。

问题是如何读文件,也就是说,怎样把文件名和实际读数据的I/O语句联系起来。

规则是很简单的——在读写文件前,必须先用标准库函数fopen()打开文件。fopen()取外部名(如x.c或y.c),做一些与操作系统有关的工作,然后返回一个内部名,随后的读写文件就使用这个内部名。

这个内部名实际上是指针,叫文件指针,它指向含文件信息的结构,例如缓冲的位置、缓冲中的当前字符位置,而不管文件是读还是写。用户不必知道这些细节,因为在标准I/O定义(通过包括stdio.h而得到)中有一个FILE结构定义。对一个文件指针我们只需说明

```
FILE * fp, * fopen();
```

它表明fp是个指向FILE的指针,fopen()返回指向FILE的指针。FILE是一个类似于int的类型名,不是一个结构标志。

程序中调用fopen()的形式是

```
fp = fopen(name,mode);
```

fopen()的第一个参数是文件名,文件名是个字符串; 第二个参数是模式,表明你打算怎样使用文件,模式也是字符串。允许的模式可以是读(r)、写(w)或附加(a)。

另外,在为读和写打开文件时,mode 后面可以跟+号。r+使流定位在文件的开始,w+产生或附加文件,a+使流定位在文件的末尾。在读/写流上,读和写都可以被使用,条件是在读写之间或写读之间必须使用 fseek(),rewind()或读文件结束。

如果为了写或附加而打开的文件不存在,则产生这个文件(如果可能的话)。打开现存文件进行写会略去原来的内容。读不存在的文件会产生错误,其它原因也会产生读错误(如试图读一个你无权读的文件)。当出错时,fopen()返回空指针值 NULL——在 stdio.h 中定义成 0。

一旦打开了文件,下一步就是读或写的方法问题。有几种可行的方法,其中最简单的是 getc()和 putc()。getc()返回文件中的下一个字符,文件由文件指针给出。因此

```
c = getc(fp)
```

*c 中存放的就是 fp 指向文件中的下一个字符。当到了文件末尾时,getc()返回 EOF.putc()正好和 getc()相反:

```
put(c,fp)
```

把字符 c 放到文件 fp 中,返回值为 c。出错时,getc()和 putc()返回 EOF。

当启动一个程序时,自动打开三个流,并且为这三个流提供了文件指针。这些流是标准输入、标准输出和标准错误输出;对应的文件指针叫做 stdio,stdout 和 stderr。一般它们都连到终端,也可以象 1.3 节那样重定向到文件或管道。

stdio,stdout 和 stderr 在 I/O 库中预定义成标准输入文件、标准输出文件和标准错误文件,它们也可以用在任何能放 FILE 类型体的地方。它们是常量,不能赋值。

有了这些预备知识,我们可以编制 wc 了。基本的也是常见的设计思想是:若有命令行参数,则按顺序处理参数;无参数则处理标准输入。这样,程序可以单独使用,或作为更大进程的一部分。

```
#include <stdio.h>

main(argc, argv) / * wc: count lines, words, chars * /
int argc;
char *argv[ ];
{
    int c, i, inword;
    FILE *fp, *fopen();
    long linect,wordct, charct;
    long tlinect = 0, twordct = 0 , tcharct = 0;

    i = 1;
    fp = stdin;
    do {
```

```
        if (argc > 1 && (fp=fopen(argv[i], "r")) == NULL) {
            fprintf(stderr, "wc: can't open %s\n", argv[i]);
            continue;
        }
        linect = wordct = charct = inword = 0;
        while ((c = getc(fp)) != EOF) {
            charct++;
            if (c == '\n')
                linect++;
            if (c == ' ' || c == '\t' || c == '\n')
                inword = 0;
            else if (inword == 0) {
                inword = 1;
                wordct++;
            }
        }
        printf("%7ld %7ld %7ld", linect, wordct, charct);
        printf(argc > 1 ? " %s\n" : "\n", argv[i]);
        fclose(fp);
        tlinect += linect;
        twordct += wordct;
        tcharct += charct;
    } while (++i < argc);
    if (argc > 2)
        printf("%7ld %7ld %7ld total\n", tlinect, twordct, tcharct);
    exit(0);
}
```

函数 fprintf()和 printf()完全一样,只是 printf()的第一个参数是文件指针,指向要写的文件。

函数 fclose()是 fopen()的逆过程，它打破文件指针和外部名(由 fopen()建立)之间的联系，把文件指针释放给别的文件。一个程序可以同时打开的文件数目是有限的，所以不需要的东西最好释放。对输出文件调用 fclose()还有另一个理由--它清除 putc()正在收集输出的那个缓冲。当程序正常结束时,自动为每个打开的文件调用 fclose()。

1.4.2 错误处理—stderr 和 exit

stderr 以和 stdin 和 stdout 相同的方式赋给程序。即使标准输出重定向了,写到 stderr 的输出仍出现在终端上,除非标准错误也重定向了。wc 把诊断写到 stderr(代替 stdout)上,所以万一由于某种原因不能访问某个文件，信息仍可以输出到终端,而不会消失。

任何调用 exit()进程的进程,都可以得到 exit()的参数(见 1.6.3 节)所以程序的成功或失败可以被别的程序(使用这个程序作为子过程)测试出来。习惯上,返回值 0 表示正常,非 0 表示异常情况。

exit()本身为每一个打开的输出文件调用 fclose(),清除缓冲输出,然后调用 exit()例程。函数 exit()立即结束程序,不清除缓冲,必要时可直接调用它。

1.4.3 各种 I/O 函数

除以上介绍的 I/O 函数外,标准 I/O 库还提供了几种其它的 I/O 函数。

一般输出用 putc(),这种输出是带缓冲的。使用 fflsh(fp)强迫立即清除缓冲。

fscanf()同 scanf()一样,只是其第一个参数为文件指针(正如 fprintf()),指示输入来自于哪个文件;文件结束返回 EOF。

函数 sscanf()和 sprintf()同 fscanf()和 fprintf()一样,只是第一个参数命名的是字符串,用它代替文件指针。对 sscanf()的串要作转换,对 sprintf()要转换成串,不进行输入或输出。

fgets(buf,size,fp)把 fp 的下一行拷贝到 buf 中,包括换行;至多拷贝 size-1 个字符;在文件结束时返回 NULL.fputs(buf,fp)把 buf 中的串写到文件 fp 中。

函数 ungetc(c,fp)把字符 c 退回到输入流 fp,随后的 getc(),fscanf()调用等会遇到 c。每个文件仅允许退回一个字符。

1.5 低级 I/O 函数

本节描述 UNIX 系统 V 的低级 I/O。UNIX 系统 V 的最低级 I/O 没有提供缓冲或其它服务,而是直接到操作系统的入口。你完全控制着 I/O,另一方面你还必须控制可能发生的各种情况。由于低级 I/O 的调用及使用很简单,所以利仍大于弊。

1.5.1 文件描述符(file descriptor)

在 UNIX 系统 V 中,所有的输入输出都是通过读写文件实现的,因为所有的外部设备甚至用户终端都是文件系统中的文件。这意味着程序和外围设备间的通讯可以用一致的接口来处理。

在读写文件前,通常要事先通知系统,以便调用"打开"文件的进程。如果你打算写一个文件,或许还要产生这个文件。系统检查你读写文件的权利--文件存在吗?你有访问文件的权限吗?---一切顺利的话,则返回一个很小的正整数,叫文件描述符(file descriptor)。每当对这个文件进行 I/O 时,就用这个文件描述符来代替名字以识别文件,这倒有点象 FORTRAN 中使用的 READ(5,...)和 WRITE(6,...)。有关打开文件的各种信息由系统维护,用户程序只有通过文件描述符访问文件。

1.4 节讨论的指针类似于文件描述符,但后者更基础一些,文件指针指向一个结构,该结构含有文件描述符和其它东西。

由于涉及用户终端的输入输出很普遍,为方便采用了专门的处理。当命令解释程序("shell") 运行一个程序时,打开三个文件,其文件描述为 0,1,2,分别叫做标准输入、标准输出和标准错误输出。通常它们都连到终端,如果程序读文件描述 0、写文件描述符 1 和 2 时,则可以不打开文件而进行终端 I/O。

如果 I/O 用 < 和 > 进行重定向,如

```
tutorial% prog <infile> outfile
```

shell 要修改对文件描述符 0,1 的省缺值，使之从终端变成命名的文件．如果输入或输出涉及到管道，道理是一样的。文件描述符 2 一般保留和终端相连，这样错误仍可输出到终端。不管怎样，文件赋值的修改是由 shell 进行的，而不是由程序完成的。只要程序使用了文件 0 输入、文件 1 和文件 2 输出，那么就不必知道输入从哪儿来，输出到哪儿去。

1.5.2 read()和 write()

所有的输入输出都是通过两个函数 read(),write()来完成的。这两个函数的第一个参数都是文件描述符；第二个参数是程序中的缓冲；缓冲数据的来源或去向；第三个参数是要传送的字节个数。调用如下:

```
n read = read(fd, buf, n);
n written = write(fd, buf, n);
```

每次调用都返回一个字节的计数,代表实际传送的字节数。在读的时候,返回的字节数可能小于要求的数目,因为剩下的字节数可能小于 n。当文件为终端时,read()一般仅读到下一个换行,这样读的字节数可能小于要求的字节数。返回值为 0 表示文件结束,-1 表示某种错误。对于写,返回值是实际写的字节数,如果返回值不等于要写的字节数通常是出错了。

读或写的字节数是很随意的,两个常用的值是 1 和 1024。1 表示每次读一个字符（无缓冲），1024 在许多外围设备中对应了物理块大小。后者的效率很高,但即使一次读一个字符的 I／O 也很令人满意。

有了这些知识,我们可以写一个简单的程序,把输入拷贝到输出。这个程序可以随意拷贝,因为输入和输出可以重定向到任何文件或设备:

```
#define BUFSIZE 1024

main() / * copy input to output * /
{
    char    buf[BUFSIZE];
    int n;

    while ((n = read(0, buf, BUFSIZE)) >0)
        write(1, buf, n);
    exit(0);
}
```

如果文件大小不是 BUFSIZE 的倍数,某些 read()会返回较小的字节数,之后的下次 read()调用返回 0。

知道如何用 read()和 write()构造高级例程，如 getchar(),putchar()等,对我们很有益。例如,下面的 getchar()版本是无缓冲的输入:

```
#define CMASK   0377   / * for making char's > 0 * /

getchar()   / * unbuffered single character input * /
```

```
{
    char c;

    return((read(0, &c, 1) > 0) ? c & CMASK : EOF);
}
```

c 必须说明成 char,因为 read()接收字符指针。返回的字符必须用 0377 屏蔽，保证其值为正，否则符号扩展可能使其−1。常量 0377 不是对每个机器都合适的。

getchar()的第二个版本可以进行大量的输入,并一次送出字符:

```
#define CMASK   0377   / * for making char's > 0 * /
#define BUFSIZE 1024

getchar()   / * buffered version * /
{
    static char buf[BUFSIZE];
    static char * bufp = buf;
    static int n = 0;

    if (n == 00) {   / * buffer is empty * /
        n = read(0, buf, BUFSIZE);
        bufp = buf;
    }
    return((—n >= 0) ? * bufp++ & CMASK : EOF);
}
```

1.5.3 open(),create(),close()和 unlink()

除了省缺的标准输入、输出和错误文件外,你必须显示地打开文件,以便读或写文件,对此有两个系统入口点 open()和 create()。

open()很像 fopen(),但它不是返回文件指针,而是返回文件描述符,即一个整型。

```
int fd;
fd = open(name, rwmode);
```

与 fopen()一样,name 参数是一个字符组,对应了外部文件名。但访问模式参数是不一样的 rwmode 为 0 表示读、1 表示写、2 表示读写访问。出错时,open()返回−1,否则返回有效的文件描述符。

试图打开一个不存在的文件是错误的，提供的入口点 creat()可以产生新文件或复写老文件。

```
fd = creat(name, pmode);
```

如果能产生 name 文件的话就返回文件描述符，不能产生则返回−1。如果文件已存在，creat()则把它的长度截断成 0,creat()已存在的文件不是错误。

如果文件是新的,creat()用 pmode 参数指定的保护模式来产生这个文件。在 UNIX 系统 V 中,与文件相关的保护信息有 9 位,用来控制文件主、文件组及其它人对文件的读、写或执行权限。因此,三位 8 进制数可以方便地指定权限。例如,0755 表示文件主有读、写和执行权限，文件组和其他人有读和执行权限。

下面这个例子是 UNIX 系统 V 实用程序 cp 的简化版本,cp 把一个文件拷贝到另一个文件。这个版本最主要的简化是仅能拷贝一个文件,不允许第二个参数是目录。

```
#define NULL 0
#define BUFSIZE 1024
#define PMODE 0644 / * RW for owner, R foor group, others * /

main(argc, argv) / * cp: copy f1 to f2 * /
int argc;
char * argv[ ];
{
    int f1, f2, n;
    char buf[BUFSIZE];

    if (argc != 3)
        error("Usage: cp from to", NULL);
    if ((f1 = open(argv[1], 0)) == -1)
        error("cp; can't open %s", argv[1]);
    if ((f2 = creat(argv[2], PMODE)) == -1)
        error("cp: can't create %s", argv[2]);

    while ((n = read(f1, buf, BUFSIZE)) > 0)
        if (write(f2, buf, n) != n)
            error("cp: write error", NULL);
    exit(0);
}

error(s1, s2) / * print error message and die * /
char * s1, * s2;
{
    printf(s1, s2);
    printf("\n");
    exit(1);
}
```

前面我们曾提到过,一个程序可同时打开的文件数目是有限的(一般 20～32 个),所以如果程序打算处理很多文件,必须准备重用文件描述符。例程 close()打破文件描述符和打开文件之间的连接,释放文件描述符以便用于其它的文件。通过 exit()而结束的程序或从主程序返回并关闭所有打开的文件。