

Linux 专家之路



Linux 下的 C 编程

贾明 严世贤 编著

雨人科技 策划

人民邮电出版社

图书在版编目 (CIP) 数据

Linux 下的 C 编程 / 贾明, 严世贤编著. —北京: 人民邮电出版社, 2001.11
(Linux 专家之路)

ISBN 7-115-09788-7

I. L... II. ①贾... ②严... III. C 语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字 (2001) 第 074382 号

Linux 专家之路

Linux 下的 C 编程

-
- ◆ 编 著 贾 明 严世贤
 - 策 划 雨人科技
 - 责任编辑 张瑞喜
 - 执行编辑 郭立罡

 - ◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
 - 邮编 100061 电子函件 315@pptph.com.cn
 - 网址 <http://www.pptph.com.cn>
 - 读者热线: 010-67129212 010-67129211 (传真)
 - 北京汉魂图文设计有限公司制作
 - 北京 厂印刷
 - 新华书店总店北京发行所经销

 - ◆ 开本: 787×1092 1/16
 - 印张: 27.75
 - 字数: 669 千字 2001 年 月第 1 版
 - 印数: 1 - 0 000 册 2001 年 月北京第 1 次印刷

ISBN 7-115-09788-1/TP

定价: 45.00 元 (附光盘)

本书如有印装质量问题, 请与本社联系 电话: (010) 67129223

内 容 提 要

本书系统地介绍了在 Linux 平台下用 C 语言进行程序开发的过程，并通过列举大量的程序实例，使读者很快掌握在 Linux 平台上进行 C 程序开发的方法和技巧，并具备开发大型应用程序的能力。

本书内容详实，主要包括：Linux 平台下 C 语言编程环境的介绍，C 语言编译器、调试工具和自动维护工具的使用方法，Linux 系统提供特有的函数调用，在 C 程序中访问文件的方法，进程的概念、进程间通信以及多进程同步运行的实现手段，C 语言网络编程方法等。

本书结构合理、概念清晰、实例丰富，并具有很强的启发性和实用性，适用于在 Linux 系统下进行 C 语言编程的程序员和广大爱好者阅读。

前　　言

近年来，Linux 操作系统是发展最快、影响最大的操作系统，并成为了操作系统领域最耀眼的一颗明星。

Linux 操作系统是由 UNIX 发展来的，它同 C 这种具有多平台、移植性好的编程语言的完美结合，为用户提供了一个功能强大的编程环境。正因如此，Linux 平台下的 C 语言编程的重要性日益突出，我们编写了这本书献给广大的编程爱好者。

本书分为 3 篇，第 1 篇是基础篇，先简要介绍 Linux 操作系统和 C 语言的基本知识，为以后的熟练编程打好基础，然后将介绍 Linux 中的 C 程序的编程环境（包括 Emacs 编辑器、C 语言的编译器 gcc、调试工具 gdb 和程序自动维护工具 make 的使用方法）和基本的系统调用（包括文件系统的操作、标准输入输出的操作、内存管理和进程操作），这些丰富的系统调用将为以后的深入研究打下坚实的基础。第 2 篇是提高篇，对一些高级编程知识做出阐述。具体内容包括信号及信号处理、进程间的通信（IPC）、网络 Socket 编程、底层终端编程。鉴于网络和 Linux 的紧密联系，本部分将对网络编程作重点讲解。第 3 篇为实例篇，列举了综合性的例子，使大家对所学的知识拥有一个感性的认识。

本书语言简练、阐述清晰、实例生动，能很好地帮助读者掌握 Linux 平台下使用 C 语言编程的基本方法和技巧，具备开发大型应用程序的能力。光盘中附有各章的程序源代码，读者可以把程序代码拷贝到 Linux 编辑环境下的 Emacs 中进行编译、调试和运行。

本书各章还附有习题，而且在书后给出部分习题答案，以方便读者学习。

本书还使用到两个特殊的标志：



：表示提示内容，需要大家注意。



：表示补充说明的内容。

贾明负责编写了全书的第 1、2、3、4、5、12、13、14 章，并负责相关章节实例的测试；严世贤负责编写第 6、7、8、9、10、11 章和相关章节实例的测试。最后对关心、支持和帮助过本书编写工作的“雨人”工作人员表示诚挚的谢意！

由于编者水平有限，本书难免有疏漏之处，希望广大读者批评指正。

联系邮件地址：yurennet@263.net

雨人科技网站：www.yurennet.com

编著者

2001 年 8 月

目 录

第1篇 基 础 篇

第1章 Linux 系统和 C 语言简介	3
1.1 Linux 系统简介	4
1.1.1 Linux 系统的发展简介	4
1.1.2 Linux 系统的主要优异性能	5
1.1.3 Linux 系统的主要构成	5
1.1.4 现行 Linux 系统的主要版本	6
1.2 C 语言简介	6
1.2.1 C 语言概述	6
1.2.2 数据类型	7
1.2.3 运算符和表达式	15
1.2.4 C 程序语句	16
1.2.5 函数	22
1.2.6 编译预处理	23
1.3 Linux 平台下 C 程序的开发	25
1.3.1 在 UNIX 操作系统下运行 C 程序的步骤	25
1.3.2 用 Turbo C 运行 C 程序的步骤	25
1.3.3 Linux 平台下 C 程序的开发	25
1.4 小结与练习	26
1.4.1 小结	26
1.4.2 习题与思考	26
第2章 Emacs 编辑器	27
2.1 Emacs 简介	28
2.1.1 Emacs 编辑器的运行和结束	28
2.1.2 基本操作	28
2.2 C 模式	30
2.2.1 自动缩进	30
2.2.2 注释	31
2.2.3 预处理扩展	31
2.2.4 自动状态	31

2.2.5 使用 Emacs 进行编译和调试.....	31
2.3 小结与练习	32
2.3.1 小结	32
2.3.2 习题与思考	32
第 3 章 C 语言编译器 gcc	35
3.1 gcc 的使用	36
3.1.1 一个最基本的实例	36
3.1.2 gcc 的用法	37
3.1.3 警告	40
3.1.4 优化 gcc	41
3.1.5 调试标记	46
3.1.6 使用高级 gcc 选项	48
3.2 gcc 编译流程简介	51
3.2.1 C 预处理器 cpp	51
3.2.2 GUN 连接器 ld	51
3.2.3 GUN 汇编器 as	51
3.2.4 文件处理器 ar	52
3.2.5 库显示 ldd	52
3.3 其他编译调试工具	52
3.3.1 C++编译器 g++	52
3.3.2 EGCS	52
3.3.3 calls	53
3.3.4 indent	53
3.3.5 gprof	53
3.3.6 f2c 和 p2c	53
3.4 小结与练习	53
3.4.1 小结	53
3.4.2 习题与思考	54
第 4 章 调试工具 gdb	55
4.1 gdb 符号调试器简介	56
4.2 gdb 功能详解及其应用	57
4.2.1 调试步骤	57
4.2.2 显示数据命令 display 和 print	67
4.2.3 使用断点	73
4.2.4 使用观察窗	77
4.2.5 core dump 分析	81



4.3 其他调试工具	88
4.4 小结与练习	88
4.4.1 小结	88
4.4.2 习题与思考	88
第 5 章 程序自动维护工具 make	91
5.1 简单使用及属性控制	92
5.1.1 make 的简单使用	94
5.1.2 make 属性的控制	105
5.2 高级使用	112
5.2.1 宏的使用	112
5.2.2 内部规则	118
5.2.3 make 递归	121
5.2.4 依赖性的计算	122
5.3 库的使用	125
5.3.1 创建库和维护库	126
5.3.2 库的链接	127
5.4 小结与练习	128
5.4.1 小结	128
5.4.2 习题与思考	129
第 6 章 文件操作	131
6.1 文件系统简介	132
6.1.1 文件	132
6.1.2 文件的相关信息	134
6.1.3 文件系统	135
6.2 基于文件描述符的 I/O 操作	136
6.2.1 文件的创建、打开与关闭	136
6.2.2 文件的读写操作	139
6.2.3 文件的定位	144
6.3 文件的其他操作	146
6.3.1 文件属性的修改	146
6.3.2 文件的其他操作	150
6.4 特殊文件的操作	152
6.4.1 目录文件的操作	153
6.4.2 链接文件的操作	154
6.4.3 管道文件的操作	157
6.4.4 设备文件	158



6.5 小结与练习	158
6.5.1 小结	158
6.5.2 习题与思考	159
第 7 章 输入输出——基于流的操作	161
7.1 流简介	162
7.2 基于流的 I/O 操作	164
7.2.1 流的打开和关闭	164
7.2.2 缓冲区的操作	166
7.2.3 直接输入输出	167
7.2.4 格式化输入输出	170
7.2.5 基于字符和行的输入输出	173
7.3 临时文件	178
7.4 小结与练习	182
7.4.1 小结	182
7.4.2 习题与思考	182
第 8 章 内存管理	183
8.1 静态内存与动态内存	184
8.1.1 静态内存	184
8.1.2 动态内存	186
8.2 安全性问题	187
8.3 内存管理操作	188
8.3.1 动态内存的分配	188
8.3.2 动态内存的释放	189
8.3.3 调整动态内存的大小	190
8.3.4 分配堆栈	192
8.3.5 内存锁定	193
8.4 使用链表	193
8.5 内存映像 I/O	197
8.5.1 创建内存映像文件	198
8.5.2 撤销内存映像文件	199
8.5.3 将内存映像写入外存	199
8.5.4 改变内存映像文件的属性	202
8.6 小结与练习	202
8.6.1 小结	202
8.6.2 习题与思考	203



第 9 章 进程控制	205
9.1 进程的基本概念	206
9.1.1 进程基本介绍	206
9.1.2 进程的属性	207
9.2 进程控制的相关函数	208
9.2.1 进程的创建	208
9.2.2 进程等待	213
9.2.3 进程的终止	218
9.2.4 进程 ID 和进程组 ID	222
9.2.5 system 函数	227
9.3 多个进程间的关系	229
9.3.1 进程组	229
9.3.2 时间片的分配	229
9.3.3 进程的同步	231
9.4 线程	232
9.4.1 线程的创建	232
9.4.2 线程属性的设置	232
9.4.3 结束线程	234
9.4.4 线程的挂起	234
9.4.5 取消线程	235
9.4.6 互斥	236
9.5 小结与练习	236
9.5.1 小结	236
9.5.2 习题与思考	237

第 2 篇 提 高 篇

第 10 章 信号及信号处理	241
10.1 信号及其使用简介	242
10.1.1 信号简介	242
10.1.2 信号的使用	244
10.2 信号操作的相关系统调用	245
10.2.1 信号处理	245
10.2.2 信号的阻塞	255
10.2.3 发送信号	262
10.3 信号处理的潜在危险	272
10.4 小结与练习	272



10.4.1 小结	272
10.4.2 习题与思考	273
第 11 章 进程间通信	275
11.1 简介	276
11.2 共享内存和信号量	276
11.2.1 SYSV 子系统的基本概念	277
11.2.2 共享内存	278
11.2.3 信号量	286
11.3 管道	299
11.3.1 管道的创建和关闭	299
11.3.2 管道的读写操作	301
11.4 命名管道	303
11.4.1 命名管道的创建	303
11.4.2 命名管道的使用	304
11.5 消息队列	309
11.5.1 消息队列的创建与打开	310
11.5.2 向消息队列中发送消息	310
11.5.3 从消息队列中接收消息	311
11.5.4 消息队列的控制	312
11.6 小结与练习	314
11.6.1 小结	314
11.6.2 习题与思考	314
第 12 章 网络编程	315
12.1 基本原理	316
12.1.1 计算机网络体系结构模式	316
12.1.2 TCP/IP 协议	318
12.1.3 客户/服务器模式	319
12.1.4 套接口编程基础	323
12.1.5 IP 地址转换	336
12.2 TCP 套接口编程	341
12.2.1 基于 TCP 的客户——服务器模式	341
12.2.2 信号处理	349
12.2.3 高级技术	350
12.3 UDP 套接口编程	360
12.3.1 基于 UDP 的客户——服务器模式	361
12.3.2 主要系统调用函数	361



12.3.3 基于 UDP 套接口编程实例	362
12.3.4 可靠性问题	365
12.3.5 UDP 套接口的连接	367
12.4 原始套接口编程	368
12.4.1 基本形式和操作	369
12.4.2 原始套接口编程实例	370
12.5 小结与练习	376
12.5.1 小结	376
12.5.2 习题与思考	376
第 13 章 底层终端编程	377
13.1 底层终端编程	378
13.1.1 属性控制	378
13.1.2 使用 terminfo	381
13.2 伪终端	384
13.3 小结与练习	385
13.3.1 小结	385
13.3.2 习题与思考	385
第 3 篇 实战篇	
第 14 章 实例一	389
14.1 实例	390
14.2 小结与练习	394
14.2.1 小结	394
14.2.2 习题与思考	394
第 15 章 实例二	395
15.1 实例	396
15.2 小结与练习	406
15.2.1 小结	406
15.2.2 习题与思考	406
附录 部分习题参考答案	407

第1篇

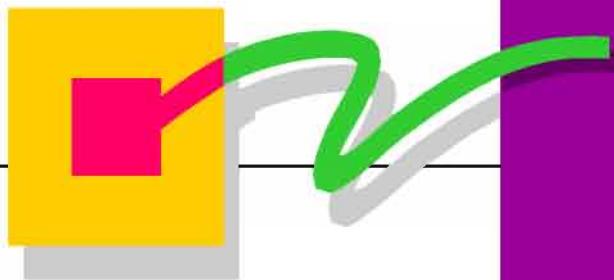
基石出篇



-  第1章 Linux系统和C语言简介
- 第2章 Emacs编辑器
- 第3章 C语言编译器gcc
- 第4章 调试工具gdb
- 第5章 程序自动维护工具make
- 第6章 文件操作
- 第7章 输入输出——基于流的操作
- 第8章 内存管理
- 第9章 进程控制



本篇导读



本篇是全书的基础部分，在此部分我们将结合程序实例详述 Linux 平台下 C 语言的编程环境，如 Linux 环境下 C 语言编辑器 Emacs、C 语言的编译器 gcc、程序调试工具 gdb 和程序自动维护工具 make，在此基础之上我们将对系统的基本调用做出阐述。

本篇包括以下章节：

第 1 章主要介绍 Linux 操作系统和 C 语言编程的基本语法知识，使大家认识到 Linux 操作系统的良好发展趋势，增强大家的学习动力。C 语言部分只做简要的介绍，如有这方面的疑问，大家可以查阅有关的书籍。

第 2 章介绍 Linux 中的一个功能强大的编辑器 Emacs，它的 C 模式功能丰富，使用方便，是在 Linux 上进行 C 程序集成开发的优良环境。

第 3 章主要讲述 Linux 平台上的 C 语言编译器 gcc，包括 gcc 编译器的安装和 gcc 的使用，最后还会介绍一些其他的编译工具。

第 4 章讲述 C 语言的调试工具 gdb 中的常用命令的使用格式和基本使用方法，分别阐述 gdb 符号调试器、gdb 命令详解及其简单应用实例，最后也会提到一些其他的调试工具。

第 5 章讲述 C 语言程序自动维护工具 make 和写 makefile 的知识和技巧，如 make 的简单使用、make 命令属性的控制、宏（macro）的使用、内部规则及库的使用。它是从 UNIX 系统继承下来的一个工具，能自动生成和维护目标程序，根据程序模块的修改情况重新编译链接目标代码，以保证目标代码总是由最新的模块组成。

第 6 章讲述文件系统的操作，内容涉及顺序文件的操作、随机文件的操作、文件的共享、索引节点、文件层次结构、文件属性的控制、文件的链接和设备文件的操作。

第 7 章讲述基于标准输入输出库的标准输入输出的操作，内容包括标准输入输出的基本操作、非格式化输入输出的操作、格式化输入输出的操作以及临时文件的操作。

第 8 章讲述内存管理方面的操作，本章在内容上首先回顾基本的 C 内存管理，然后讲解 Linux 操作系统提供的一些附加功能。

第 9 章讲述有关进程的操作，包括进程的概括介绍、进程的基本操作（进程的创建、等待和结束等）和进程之间的关系。进程是操作系统和并发程序设计中的一个重要的概念。

以上内容是本书的基础，各章都附有大量的小程序，源代码也已给出，读者应多加练习，熟练掌握此篇内容。

第1章

Linux 系统和 C 语言简介

Linux 系统简介

C 语言简介

Linux 平台下 C 程序的开发

小结与练习



欢迎大家进入 Linux 平台下 C 语言编程的世界。本章将详细介绍 Linux 操作系统和 C 语言编程的基本知识，帮助大家对所要学习的知识建立清晰的认识，并掌握 C 语言编程的基本方法，为后续学习打下扎实的基础。

1.1 Linux 系统简介

对于 Linux 平台的初学者，本书有必要先介绍 Linux 系统的基本知识和一些重要特性。

Linux 系统是从 UNIX 发展来的。UNIX 是世界上最流行的操作系统之一，它是一种实时操作系统，可以运行于大型和小型计算机上的多任务系统。但由于它比较庞大，而且价格昂贵，所以不适合 PC 机用户使用。而 Linux 正好弥补了这些缺点，同时还继承了 UNIX 大多数优点。由于它是基于 PC 机上运行的操作系统，并且内核源代码是公开的，使得 Linux 成为时下最流行的操作系统。

1.1.1 Linux 系统的发展简介

Linux 是一种适用于 PC 机的计算机操作系统，它适合于多种平台，是目前唯一免费的非商品化操作系统。

Linux 诞生于 1991 年底，是一个芬兰大学生开发出来的。由于具有结构清晰、功能强大等特点，它很快成为许多院校学生和科研机构的研究人员学习和研究的对象。在他们的热心努力之下，Linux 逐渐成为一个稳定可靠、功能完善的操作系统。而一些软件公司也不失时机地推出以 Linux 为核心的操作系统，大大推动了 Linux 的商品化，使 Linux 的使用日益广泛，成为当今最流行的操作系统。

Linux 是由 UNIX 发展来的，它不仅继承了 UNIX 操作系统的特征，而且在许多方面还超过了 UNIX 系统，另外它还具有许多 UNIX 所不具有的优点和特性。如它的源代码是开放的，可运行于许多硬件平台，支持多达 32 种文件系统，支持大量的外部设备等。它包含人们所期待操作系统所拥有的特性，包括真正的多任务、虚拟内存、目前最快的 TCP/IP 驱动程序、共享库和理想的多用户支持；它还符合 X/Open 标准，具有完全自由的 X Window 实现方式；Linux 同 UNIX 一样，具有最先进的网络特性，且支持所有通用的 Internet 协议，既可作为客户端也可作为服务器。这些优异的性能，使 Linux 具有广泛的用途，它可用于：

- 个人 UNIX 工作站。
- X 终端用户和 X 应用服务器。
- UNIX 开发平台。
- 商业开发。
- 网络服务器。
- Internet 服务器。
- 终端服务器、传真服务器、Modem 服务器。

1.1.2 Linux系统的主要优异性能

- Linux系统是真正的多用户、多任务、多平台操作系统。
- Linux系统提供具有内置安全措施的分层的文件系统，支持多达32种文件系统。
- Linux系统提供shell命令解释程序和编程语言。
- Linux系统提供强大的管理功能。
- Linux系统具有内核的编程接口。
- Linux系统具有图形用户接口。
- Linux系统具有大量有用的实用程序和通信、联网工具。
- Linux系统具有面向屏幕的编辑软件。
- Linux系统许多组成部分的源代码是开放的，任何人都能修改和重新发布它。
- Linux系统不仅可以运行许多自由发布的应用软件，还可以运行许多商业化的应用软件。

1.1.3 Linux系统的主要构成

1. 存储管理

Linux采取页面式存储管理机制，每个页面的大小随处理芯片而异。在Linux中，每一个进程都有一个比实际物理空间大得多的进程虚拟空间，每个进程还保留一张页表，用于将本进程空间中的虚地址转换成物理地址，页表还对物理页的访问权限作了规定，从而达到存储保护的目的。

Linux存储空间的分配遵循的原则是不到有实际需要的时候决不分配物理空间，这样可以最大限度地利用物理存储器。

2. 进程管理

在Linux中，进程是资源分配的基本单位，所有资源都是以进程为对象进行分配的。在一个进程的生命周期中，会用到许多系统资源，Linux的设计可以准确描述进程的状态和资源的使用情况，以确保不出现某些进程过度占用系统资源而导致另一些进程无休止地等待的情况。

Linux创建进程的方法是采用Copy in write技术，不拷贝父进程的空间，只是拷贝父进程的页表，使父进程和子进程共享物理空间，并将这个共享空间的访问权限置为只读，这样可以降低系统资源的开销。

3. 文件系统

Linux最重要的特征之一就是支持多种不同的文件系统。

在Linux中，一个分离的文件系统不是通过设备标志（驱动器号）来访问，而是把它合到一个单一的目录树结构中去，通过目录访问。Linux把一个新的文件系统安装到系统单一目录树的某一目录下，则该目录下的所有内容将被新安装的文件系统所覆盖，当文件系统被

卸下后，安装目录下的文件将会被重新恢复。

为了支持多种文件系统，Linux 用一个被称为虚拟文件系统（VFS）的接口层将真正的文件系统同操作系统及其服务器分离开，它能掩盖不同文件系统之间的差异，使所有的文件系统在操作系统和用户程序看来都是相同的。

4. 进程间通信

Linux 提供多种进程间的通信机制，管道和信号是其中最基本的两种，其他还有消息队列、信号灯及共享内存。为支持不同机器之间的进程通信，Linux 还引入了 Socket 机制。

1.1.4 现行 Linux 系统的主要版本

- Red Hat Linux（小红帽）。由 Red Hat Software 公司发行，最高版本为 Red Hat7.0，是当今最为流行的 Linux 套件。它支持的硬件平台多，安装界面友好，安全性能好，在线文档详尽，每一版本还会有一个 powertools，适合新手入门使用。
- Slackware。由 Walnut Creek CDROM 公司发行，是历史较长的发行版本，用作服务器时表现出较好的性能。但库中包含内容较少。
- Debian Linux。由 GUN 公司发行，特点是收集的软件非常齐全，是开放式的开发环境。

1.2 C 语言简介

1.2.1 C 语言概述

C 语言是国际上广泛使用的，且很有发展前途的计算机高级语言，时下流行的 C++ 语言和 C#（用于网络编程）都是从 C 语言发展而来的。它适合用来系统描述，既可用来写系统软件，也可用来写应用软件。C 是一种与 UNIX 密切相关的程序设计语言，它最初用于 DEC PDP-11 计算机 UNIX。从 70 年代以来，操作系统中的大部分内容和应用程序都是用 C 语言编写的。C 语言之所以能存在和发展，并具有生命力，与它的以下优点是分不开的。

- 语言简洁、紧凑（32 个关键字），使用方便、自由（书写形式自由），与 Pascal 和 Basic 语言比较起来，C 语言程序显得非常简练。
- 运算符丰富，共有 34 种，C 把括号、赋值、强制类型转换等都作为运算符处理。表达式类型多样化，灵活使用各种运算符可以实现在其他高级语言上难以实现的运算。
- 数据结构合理，具有现代化语言的丰富的数据结构能用来实现各种复杂的数据结构（如链表、树、栈等）的运算。
- 具有结构化的控制语句，是结构化的理想语言，符合现代编程风格。

- 语法限制不太严格，程序设计自由度较大。
- 允许位操作和对硬件进行编程。
- 生成目标代码质量高，程序执行效率高。
- 程序可移植性好。

1.2.2 数据类型

一个程序应包括数据的描述和动作的描述两方面的内容。著名计算机科学家沃恩曾提出一个公式：数据结构+算法=程序，可见数据结构在程序中的地位。C语言为用户提供了丰富的数据结构，还允许用户自定义复杂的数据结构。

C语言提供的数据结构是以数据类型形式出现的，C的数据类型如下：

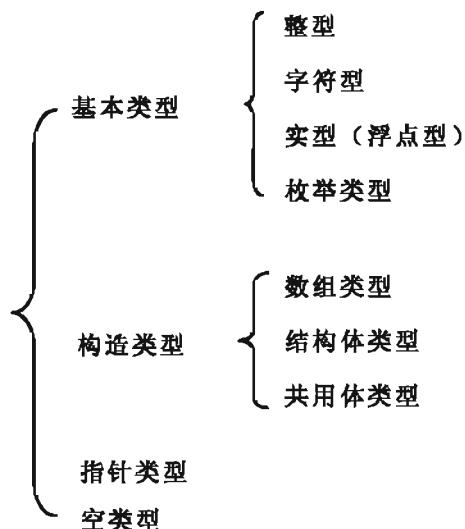


图 1-1 C 语言数据类型图

1. 常量与变量

C语言之中数据有常量与变量之分。在程序运行过程中，其值不能改变的量称为常量，其值可以改变的量称为变量。常量可分为不同类型，如整型常量 3，实型常量-1.23，字符型常量 ‘a’。常量一般能从字面形式判别，也可用一个标志符代表，可用下面形式说明。

```
#define 常量名 常量值
```

例如：

```
#define PI 3.14           /* 定义PI代表常量3.14 */
```

注意：“/*” 和 “*/”之间表示这行此符号中为注释内容，不会被编译。

变量可用下面形式说明。

数据类型 变量名

例如：

```
int i           /*定义i为整型变量*/  
char name      /*定义name为字符型变量*/
```

Tip

在 C 语言中，习惯上用大写字符代表常量，用小写字符代表变量。对于变量，要求“先定义，后使用”。

以下分别介绍整型、实型等数据类型。

2. 整型数据

(1) 整型常量

整型变量可分为 int、short int、long int 和 unsigned int、unsigned short、unsigned long 等类型，它们表示的数的范围不同。

(2) 整型变量

整型变量可分为基本型、短整型、长整型和无符号型四种。

下面给出一个实例。

【程序 1_1】

```
main()  
{  
    int a,b,c;  
    unsigned d;  
    a=6;b= -7;u=78;  
    c=a+b+u;  
    printf("%d\n",c);           /*printf() 为格式化输出函数*/  
/* "%d" 表输出格式为十进制整数*/  
}
```

运行结果为：

77

printf()是标准输出函数，在后面我们将要详细说明。

3. 实型数据

(1) 实型常量

实型常量有两种表示方式，十进制形式如 0.12、36.2 和指数形式如 123e3 表示 123000。要注意 e 字母之前必须有数字，后面必须为整数。

(2) 实型变量

实型变量分为单精度 (float) 和双精度 (double) 两种，如：

```
float x;      /*指定x为单精度实数*/
double y;     /*指定y为双精度实数*/
```

单精度实数在内存中占 4 个字节，而双精度实数则占 8 个字节。

4. 字符型

(1) 字符常量

C 的字符常量是用单引号括起来的一个字符，如 ‘a’、‘A’ 等，注意以上两个字符是不同的。另外，C 语言还允许以一个 “\” 开头的特殊字符常量，具体规定如表 1-1 所示。

表 1-1

特殊字符常量表

字符类型	功 能
\n	换行
\t	横向跳格
\v	竖向跳格
\b	退格
\r	回车
\f	走纸换页

【程序 1_2】

```
main()
{
    printf(" ab c\tde\rfg\n");
    printf("h\ti\b\bj    d");
}
```

读者可以自己思考一下程序的输出结果。

(2) 字符变量：用来存放字符常量，且只能放一个字符，定义规则如下：

```
char d; /*表示d为字符型变量*/
```

(3) 字符数据在内存中的存储形式和使用方式

字符变量在对应的内存中存放的是该字符相应的 ASCII 码，例如字符 ‘a’ 的 ASCII 代码为 97，‘b’ 的代码为 98，而字符型和整型变量还可以互相赋值。

【程序 1_3】

```
main()
{
    char c1,c2;
    c1=97,c2=98;
    printf("%c %c",c1,c2); /* "%c" 表输出格式为字符类型*/
}
```

输出结果是： a b

(4) 字符串常量

字符串常量与字符常量的区别在于前者是双引号括起来的字符序列，而后者是单引号括起来的单个字符。如在内存中 ‘a’ 的长度是一个字符，而 “a” 则占有两个字符，包括结束标志 “\0”。

5. 枚举型

所谓“枚举”是指将变量的值一一列举出来，变量的值只限于列举出来的值的范围内，可用如下定义形式：

```
enum weekday {sun,mon,tue,wed,thu,fri,sat};
```

它定义了一个名为 weekday 的枚举类型，能取 7 个值。

然后，我们可以用此类型来定义变量，如：

```
enum weekday date;
```

则 date 被定义成枚举变量，它能取 sun 到 sat 七值之一，例如：

```
date=sun;
```

也可直接定义枚举变量，如：

```
enum weekday { sun,mon,tue,wed,thu,fri,sat} date; /*效果与上相同*/
```

Tip

在C编译中，对枚举元素按常量处理，且它们是有值的。在上面的定义中，sun的值为0，……sat的值为6。

【程序1_4】

```
main()
{
    enum { sun,mon,tue,wed,thu,fri,sat} date;
    date=mon;
    printf("%d",date);
}
```

输出结果是： 1

6. 布尔型

C语言将非零整数都认为是“真”，认为0是“假”，布尔型只有“真”、“假”两个值。在表达式中，布尔量也作为整数处理，整数也可出现在布尔表达式中。

【程序1_5】

```
main ()
{
    int m=1,j=6;
    while(m)                      /* m不等于0，为真*/
    {
        j=j+m;
        m =m-1;
    }
    printf("%d",j);
}
```

则最后输出结果是：

7

7. 数组

数组是有序数据的集合，数组中每一元素都属于同一数据类型，用一个统一的数组名和下标来唯一地确定数组中的元素。

(1) 一维数组

(2) 一维数组

一维数组的定义格式如下：

类型说明符 数组名[常量表达式]

例如：

```
int a[10];
```

数组名为 a，且有 10 个元素，下标从 0 到 9，a[0] 到 a[9]

注意常量表达式不能是变量。

一维数组的引用：只能逐个引用数组元素而不能引用整个数组。

数组名[下标]

【程序 1_6】

```
main()
{
    int j,a[10];
    for(j=0;j<=9;j++)
        a[j]=j;
    for(j=9;j>=0;j--)
        printf("%d",a[j]);
}
```

运行结果是：

```
9 8 7 6 5 4 3 2 1 0
```

一维数组的初始化：

如：

```
static int a[10]={0,1,2,3,4,5,6,7,8,9}; /*static 是“静态存储”的意思*/
```

则 a[0]=0,a[1]=1,a[2]=2,a[3]=3,a[4]=4,a[5]=5,a[6]=6,a[7]=7,a[8]=8,a[9]=9.

如果花括弧中值的个数少于 10，则后面的都为 0。

(3) 字符数组

字符数组是一类特殊的数组，数组的每一个元素存放一个字符。

下面是它的定义方式：

`char 数组名[常量表达式];`

例如：

```
static char c[5]={'a','e','t','g','b'};
```

8. 指针

指针是C语言中的一个重要特色，可以说是C语言的精华。它的概念非常复杂，使用非常灵活，是C语言中最大的难点。

一个变量在内存中的地址就称为该变量的“指针”。我们通过地址就能找到所需的变量单元。

指针变量的定义用下面的形式：

类型标志符 *标志符；

例如：

```
int *pointer_1;
```

定义了一个指针变量 `pointer_1`，它指向整型变量。

如果还定义了一个整型变量 `i`，则可用赋值语句使一个指针变量指向一个整型变量。如：

```
pointer_1=&i;
```

`&i` 是变量 `i` 在内存中的地址，

或写成： `*pointer_1=I.`

这里我们只做简要的介绍，读者如想深层次了解指针，可查阅有关方面的书籍。最后我们以一个程序结束指针的学习。

【程序 1_7】

```
main()
{
    int a,b;
    int *pointer_1,*pointer_2;
    a=100,b=10;
    pointer_1=&a;
    pointer_2=&b;
    printf("%d,%d\n",a,b);
    printf("%d,%d\n",*pointer_1,*pointer_2);
}
```

程序运行的结果为：

100, 10

100, 10

9. 结构体

结构体是 C 语言提供的一种数据结构，相当于其他高级语言中的“记录”。

下面是定义一个结构体类型的一般形式：

```
struct 结构体名
{
    成员列表
} 变量名列表;
```

对各成员都应进行类型说明，如：

类型标志符 成员名

下面是一个具体实例：

```
struct student
{
    int num;
    char name[20];
    char sex;
    int age;
    float score;
    char addr[30];
}student1={ "33" , "zhanwei" , "m" , "22" , "75" , "32#307" };
```

定义了一个 struct student 类型的变量 student1，它包括 num、name、sex、age、score、addr 不同类型的数据项，各项初值分别为 33、zhanwei、m、22、75、32#307。

10. 共用体

几个不同的变量存放到同一段内存的结构，称为“共用体”。

“共用体”类型变量的定义形式如下：

```
union 共用体名
```

```
{
    成员表列
}变量表列;
```

如：

```
union data
{
    int j;
    char ch;
    float f;
}a, b, c;
```

上面定义了一个 union data 类型，再将 a、b、c 定义为 union data 类型。
共用体变量的引用方式如下：

.a.j

上语句引用共用体变量中的整型变量 j。

1.2.3 运算符和表达式

1. 运算符

C 语言提供了十分丰富的运算符，主要有：

- 算术运算符：+、-、*、/、++等。
- 关系运算符：>、<、==、!=等。
- 逻辑运算符：&&、!等。
- 位运算符：>>、<<、~等。
- 赋值运算符：= 及其=的扩展赋值运算符。
- 指针运算符：*、&。

2. 表达式

用各种运算符将运算对象连接起来的式子，就称为表达式。

(1) 算术表达式

用算术运算符和括号将运算对象连接起来的式子，称为算术运算符。包括强制类型转换，可将一个表达式转换成所需类型。如将 x+y 的值转换成整形：

`(int)(x+y)`

Tip

这里请注意两个特殊的算术运算符：++和--，分别为“自增”和“自减”运算符。

i++

先使用自身的 i 值，再自身加 1。

++i

先加 1，再使用新的 i 变量。

(2) 赋值表达式：

由赋值运算符将一个变量和一个表达式连接起来的式子称为“赋值表达式”。一般形式为：

<变量> <赋值运算符> <表达式>

如： a=87

另外 C 语言还提供很多扩展赋值运算符，以后使用时再作详细的介绍。

1.2.4 C 程序语句

C 语言的语句用来向计算机系统发出操作指令。一个语句经编译后产生若干条机器指令。
C 语言结构如下：

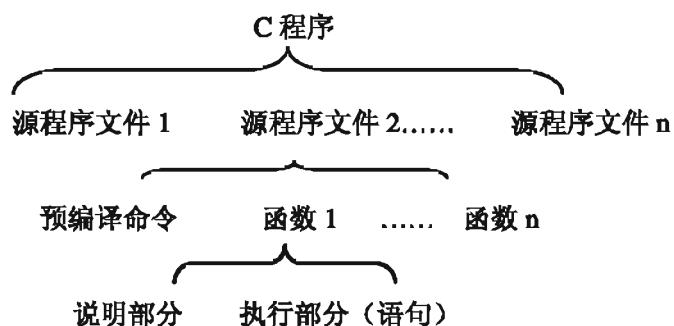


图 1-2 C 程序结构图

C 语句主要可以分为三类：

1. 控制语句

完成一般的控制功能，主要有以下几种。

(1) 条件语句 (if)

根据判断所给条件是否满足来决定执行何种操作。有三种形式的 if 语句。

- if (表达式) 语句

如果表达式成立，执行语句

- if(表达式) 语句 1

- else** 语句 2

如果表达式成立，执行语句 1；如不成立，执行语句 2

- if(表达式 1) 语句 1

- elseif(表达式 2) 语句 2**

-

- elseif(表达式 n) 语句 n**

- else 语句 m**

【程序 1_8】

```
/*判断所输入的年份是否为闰年*/
```

```
main()
```

```
{
```

```
    int year,leap;
```

```
    scanf("%d",&year);
```

```
    if(year%4==0)
```

```
{
```

```
        if(year%100==0)
```

```
{
```

```
            if(year%400==0)
```

```
                leap=1;
```

```
            else
```

```
                leap=0;
```

```
}
```

```
        else
```

```
            leap=1;
```

```
}
```

```
    else
```

```
        leap=0;
```

```
    if(leap)
```

```
        printf("%d is",year);
```

```
    else
```

```
        printf("%d is not a leap year.",year);
```

```
}
```

如果输入：2000

计算机将输出：2000 is a leap year.

(2) 循环语句

- while 循环语句

while (表达式)

语句；

当表达式为“真”时，执行语句命令。如果循环体包含一个以上的语句时，应该用花括弧括起来。

- do~while 循环语句

do 语句；

while (表达式)



与 while 语句不同的是：while 语句是先判断，再执行；而此语句是先执行语句，后判断表达式。

- for 循环语句

for (循环变量初值；循环条件；循环变量增值)

语句；

如循环体语句包含一个以上的语句，应用花括弧括起来。

下面的程序实现输出 100~200 间的偶数。

【程序 1_9】

```
main()
{
    int n;
    for(n=100;n<=200;n++)
    {
        if(n%2==0)
            printf("%d",n);
    }
}
```

(3) 多分支选择语句 switch

C语言提供switch语句直接处理多分支选择，它相当于PASCAL语言中的case语句。下面是一般形式。

```
switch (表达式)
{
    case 常量表达式 1: 语句 1;
    case 常量表达式 2: 语句 2;
    .....
    case 常量表达式 n: 语句 n;
    default          : 语句 n+1;
}
```

执行方法为：当表达式的值与某一个case后面的常量表达式的值相等时，就执行此case后面的语句。各个case的常量表达式的值必须互不相同，它们的出现次序不影响执行结果。

【程序1_10】

```
/*根据所输入的成绩判断所处的等级*/
main()
{
    int score,grade;
    printf("please inputn your score : ");
    scanf("%d",&score);
    if(score>=80)
        grade=1;
    else if(score>=60)
        grade=2;
    else
        grade=3;
    switch(grade)
    {
        case 1: printf("You are a perfect student !");
        case 2: printf("Do not play so more !!");
        case 3: printf("You must study harder !!!");
    }
}
```

如果输入你的成绩：76

计算机会输出：Do not play so more!!

2. 函数调用语句

由一个函数调用加一个分号构成的语句。在后续章节中我们将专门讨论函数，在这里只介绍最常用的两种。

(1) 基本输入语句

- 字符输入语句 (`getchar`)

作用是从终端或输入设备中输入一个字符，此函数没有参数。

`变量名 = getchar();`

- 格式输入语句 (`scanf`)

用 `scanf` 函数可以输入任何类型的多个数据。

`scanf (格式控制, 地址列表);`



“格式控制”是用双引号括起来的字符串，也称为“转换控制字符串”，主要用到的格式字符有：

<code>%d</code>	用来输入十进制整数
<code>%ld</code>	用来输入长整型数据
<code>%ld</code>	用来输入短整型整数
<code>%c</code>	用来输入单个字符
<code>%s</code>	用来输入字符串
<code>%f</code>	用来输入实数
<code>%lf</code>	用来输入 <code>double</code> 型实数

【程序 1_11】

```
main()
{
    int a,b;
    scanf("%d%d",&a,&b); /*&是地址运算符, &a表a的地址*/
    printf("%d,%d\n",a,b);
}
```

运行时按以下方式输入 a 和 b 的值：

3 4 (输入 a,b 的值)

3, 4 (输出 a,b 的值)

(2) 基本输出语句

- 字符输出函数 putchar

作用是向终端输出一个字符。例如：

```
putchar (c);
```

输出字符变量 c 的值。

- 格式输出函数 printf

作用是向终端输出若干个任意类型的数据。

```
printf (格式控制, 输出表列);
```



“格式控制”与前面输入语句时的规定几乎一样，但要增加以下几项：

%u 用来输出 unsigned 型数据

%m.nf 用来指定输出的实数数据共占 m 列，其中有 n 位小数

%m.nc 用来输出指数形式的实数，m 为所占的列数，n 指数据数字部分的小数位数。

下面程序将实现华氏温度到摄氏温度的转换，c 表示摄氏温度，f 表示华氏温度，输出小数点后两位。

【程序 1_12】

```
#include "stdio.h"
/*include 属于编译预处理中“文件包含”的内容*/
```

```
main()
{
    float c,f;
    scanf("%f",&f);
    c=5*(f-32)/9;
    printf("%.2f",c);
}
```

如果我们输入：

64

最后输出为：

17.78

3. 表达式语句（由一个表达式构成一个语句）

最典型的是由赋值表达式构成的赋值语句，如：

a=32;

它与赋值表达式的区别是最后有分号，一个语句最后必以分号结尾。

C 语言的大多数语句都是表达式语句。

1.2.5 函数

一个较大的程序一般应分为若干个程序模块，在高级语言中用子程序来实现模块的功能。函数用如下定义方式：

```
类型标志符 函数名（形式参数列表）
形式参数说明
{
    说明部分
    语句
}
```

下例是一个求解两数之和的函数：

```
int add(x,y)
int x,y;
{
    int z;
    z=x+y;
    return(z);           /*返回值为z*/
}
```

我们可以在程序中调用已经定义了的函数，调用方式如下：

函数名（实参表列）；

如对上面定义的函数，有下面程序段：

```
main()
{
    int add();
    int a,b,c;
    scanf("%d, %d",&a,&b);
    c=add(a,b);
    printf("the sum of a and b is %d",c);
}
```

如果我们输入了：

54, 32

则得到如下输出：

the sum of a and b is 86

1.2.6 编译预处理

在 C 编译系统对程序进行通常的编译前，先对程序中一些特殊的命令进行“预处理”，然后将预处理的结果和源程序一起进行通常的编译处理，得到目标代码。

1. 宏定义（分为带参数的宏定义和不带参数的宏定义）

(1) 不带参数的宏定义

用一个指定的标志（即名字）来代表一个字符串，它的一般形式如下：

`# define 标志符 字符串`

例如：

`# define PI 3.1415926`

指定用标志符来代替“3.1415926”这字符串，以后程序中如果出现 PI，则它表示“3.1415926”。

Tip

这里需要注意的有以下几点：

- 宏名一般用大写字母表示，以与变量名相区别。

- 宏定义不是 C 语句，结尾不必加分号。
- 通常 `#define` 命令写在文件开头，函数之前。作为文件的一部分，在此文件范围内有作用。
- 对程序中用双引号括起来的字符串内的字符，即使与宏名相同，也不作替换。
- 进行宏定义时，可以引用已定义的宏名，可以层层置换。

(2) 带参数的宏定义

除了进行简单的字符串替换，还要进行参数替换，它的一般形式如下：

```
#define 宏名 (参数表) 字符串
```

例如：

```
#define add(x,y) x+y
z=add (7,8);                                /*此句在函数内部*/
```

带参数的宏与函数非常类似，在引用函数时也是在函数名后的括号内写实参，且要求实参的数目等于形参。但它们还是有区别的。

- 对参数的使用方法不一样。函数调用时，先求出实参表达式的值，然后带入形参；宏只进行简单的字符替换。
- 处理机制不一样。函数调用在程序运行时处理，且要分配内存；宏展开在编译时进行，不分配内存单元，不发生值的传递处理，也不存在返回值。
- 定义时的要求不一样。函数定义时实参和形参都要定义类型；宏定义时不存在类型问题。

下面以一个实例来结束宏的学习。

【程序 1_13】

```
/*输入一个圆的半径，求出它的面积*/
#include "stdio.h"
#define PI 3.1415926
#define area(r) PI*r*r
main()
{
    float r,a;
    scanf("%f",&r);
    a=area(r);
    printf("The area is %f",a);
}
```

2. “文件包含”处理（一个源文件可以将另外一个源文件的全部内容包含进来）

```
# include "文件名"
```

例如：

```
# include "stdio.h"
```

此时被包含的文件与其所在的文件，在预编译之后组成一个文件。对被包含文件的修改将影响到其所在的文件。

1.3 Linux平台下C程序的开发

在介绍Linux平台下C程序的开发过程之前，大家先来回顾一下UNIX操作系统下运行C程序的过程和用Turbo C运行C程序的步骤。

1.3.1 在UNIX操作系统下运行C程序的步骤

- 用文本行编辑程序或屏幕编辑程序编辑源程序，并存入文件系统。
- 调用C编译工具gcc编译源程序。
- 将目标文件和库函数或其他目标程序连接成可执行的目标程序。
- 输入可执行目标文件名即可执行此程序。

1.3.2 用Turbo C运行C程序的步骤

- 调用Turbo C编辑器，进入编程界面。
- 在文本框中编辑源文件。
- 执行“命令”行菜单的下拉菜单中的命令编译源程序，根据错误提示可以对源程序进行修改。
- 执行程序，方法同上。如发现运行结果不对，可进入编辑状态，重新修改源程序。

1.3.3 Linux平台下C程序的开发

- 利用编辑器把程序的源代码编写到一个文本文件中。
- 用C编译器gcc编译连接，生成可执行文件。
- 用C调试器调试程序。

- 运行该可执行文件。

除了编译器外，Linux 还提供了调试工具 `gdb` 和程序自动维护工具 `make` 等支持 C 语言编程的辅助工具。在以后的章节中，我们将对每个工具作详细的介绍。下面以一个典型的 C 程序编程的完整过程结束此章的学习。

【程序 1_14】

```
/*先编写源代码*/  
main()  
{  
    printf("Hello World !!!\n");  
}
```

把它存储为文件： `hello.c`

然后编译该程序：

```
$gcc hello.c
```

它将创建一个文件 `a.out`。

执行此文件：

```
$./a.out
```

将输出以下字符串：

```
Hello World!!!
```

1.4 小结与练习

1.4.1 小结

本章是全书的基础部分。本章首先向读者介绍了 Linux 操作系统，包括 Linux 的发展趋势、用途以及 Linux 操作系统本身的优势。简单地介绍了 C 语言编程方面的基础语法知识，为大家以后的学习打好基础。在最后的一节中，介绍了在 UNIX 操作系统下运行 C 程序，使用 Turbo C 运行 C 程序和在 Linux 平台下运行 C 程序的具体步骤。

1.4.2 习题与思考

构造一个数组，存放用户准备统计的一项数据（如同班同学的成绩），并写一个函数，求出他们的最高分、最低分和平均分，然后编译和执行这个程序。

第2章

Emacs 编辑器

Emacs 简介

C 模式

小结与练习



要编译、调试和运行 C 程序，首先要编辑 C 程序源代码，这就离不开一个良好的编辑环境。在 Linux 系统中，用户可以用现在非常流行的两种编辑器——vi 和 Emacs 或它们的派生产品。本章只介绍 Emacs，它既是编辑器，又是一种 IDE（集成开发环境）。

2.1 Emacs 简介

2.1.1 Emacs 编辑器的运行和结束

用户在命令行中键入 `emacs` 或者 `emacs filename` 后就可以运行 Emacs 编辑器。当系统中配置并运行了 X window 时，可以用 `xemacs` 来运行 Emacs 的图形界面版本 Xemacs。如果在编译 Emacs 时选定了 Athena 工具集支持，则 Emacs 还将支持鼠标和下拉菜单。普通的 Emacs 编辑器的界面类似于 Windows 中的 Turbo C 的界面。



键盘操作符号意义：

- **C-x**: 同时按住 Ctrl 键和 x 键。
- **C-x**: 先按下 Ctrl 键，然后释放它，再按下 x 键。
- **M-x**: 同时按下 Alt 键和 x 键。
- **M-x**: 先按下 Alt 键，释放它，再按下 x 键。

需要更深了解这些操作符号，可以键入：C-h，它可以进入帮助区域，获得有关当前主题的帮助。



对于有些类型的终端和键盘，Alt 键不能正常工作。此时，一般使用 Esc 键来代替 Alt 键。

如果想退出 Emacs 时，只要键入 C-x C-c。

2.1.2 基本操作

1. 光标的移动

下面将列出 Emacs 中的光标的移动情况及其键盘操作：

- **M-b**: 光标移动到光标左边的单词的开始处。
- **M-f**: 光标移动到光标右边的单词的结束处。
- **M-a**: 光标移动到当前句子的开始处。
- **M-e**: 光标移动到当前句子的结束处。

- C-n: 光标移动到下一行。
- C-p: 光标移动到前一行。
- C-a: 光标移动到行首。
- C-e: 光标移动到行尾。
- C-v: 显示区域向上移动一整屏。
- M-v: 显示区域向下移动一整屏。
- M->: 光标移动到文件尾。
- M->: 光标移动到文件头。

2. 有关文本的操作

(1) 有关插入文本的操作

- C-x o: 在光标后插入一个空行。
- C-o: 在光标前插入一个空行并将光标移动到新行的开始处。
- C-x C-o: 把连续的多个空行合并成一个空行。

(2) 有关删除文本的操作

- C-d: 删除光标所指的字符。
- C-k: 删除光标所在位置到行尾之间的字符。
- C-u 1 C-k: 删除整行，包括换行符。
- C-x Del: 删除从行首到光标位置的所有字符。

(3) 和 Undo 有关的操作

- C-x u: 当删除错了内容时，用它可以取消 (Undo) 最后一次修改。
- M-C-x-u: 可以取消最后一个取消操作。
- C-g: 取消当前命令，还可以退出任何的特定模式。

(4) 剪贴操作

- M-w: 复制指定区域。
- C-y: 在指定的地方粘贴复制的内容。

(5) 查找和替换

- C-s: 激活查找工具，然后可以输入要查找的字符串，再次输入键盘操作字符可以进行查找。
- M-%: 输入上面的符号后，键入查找字符串，回车；再输入替换字符串，再回车，就可以完成查找——替换操作。
- M-x query-replace-regexp: 它将进行无条件的全局替换。

3. 有关文件的操作

- C-x C-s: 保存文件。
- C-x C-w: 另存文件。
- C-x C-f: 用来把一个新文件加载到 Emacs。

- **C-x C-r:** 浏览文件，但不能编辑。

4. 有关窗口的操作

窗口就是屏幕区域，用户可以使用多个窗口来对一个缓冲区的不同部分进行操作，或对不同的缓冲区进行操作。



缓冲区

当用户使用 **C-x C-f** 来打开一个文件的时候，Emacs 将会创建一个缓冲区，用户在其中进行编辑操作。Emacs 允许用户一次打开多个文件，这就使用了多个缓冲区。

用户可以使用两种方法在当前窗口的不同缓冲区间进行切换：使用 **Buffers** 菜单，它包括当前时刻打开的所有缓冲区，在其中选择，就能切换到想要编辑的文件；使用键盘对缓冲区进行操作，键入 **C-x b** 命令，然后按下 **RET**，就能立刻切换到位于当前编辑缓冲区的前一个缓冲区，或按 **Tab** 键，得到一个缓冲区的列表，然后输入需使用的缓冲区的名字（也可以用鼠标中键单击名字）。要关闭一个缓冲区，先切换到该缓冲区，键入 **C-x k**，最后按下回车键。

Tip

双键鼠标可以同时单击左右键来代替三键鼠标的中键。

下面是有关多个窗口的操作：

- **C-x o:** 切换到其他窗口。
- **C-M-v:** 滚动其他窗口。
- **C-x 0:** 删除当前窗口。
- **C-x 1:** 删除当前窗口外的其他窗口。
- **C-x 2:** 把屏幕切分成两个窗口，上下排列。
- **C-x 3:** 把屏幕切分成两个窗口，左右排列。

2.2 C 模式

Emacs 可以支持多种编程语言的编辑模式，C 模式是最为著名而且强大的一种。用户可以使用键入 **M-x [language]-mode**，用相应的模式名替换 [language]。它有许多很好的支持编程的特性，下面将简单介绍这些特性。

2.2.1 自动缩进

在 C 模式下，当用户键入代码时，Emacs 将会自动缩进显示。还可以通过键入 **Tab** 来重

新产生缩进功能。其实，它可以支持多种缩进风格，用户可以键入 **M-x c-set-style** 命令，然后回车，并输入所要使用的缩进风格，最后按下回车键。在缺省的情况下，Emacs 将遵循 GUN 编码标准的 gun 风格缩进代码。

2.2.2 注释

要写出易维护和可移植性好的代码，一个必要地工作就是要写出一个好的文档。这个工作一般在注释行中完成。

在 Emacs 的 C 模式下，键入 M-—后命令就可以使编辑器产生一个右缩进的注释符号对。然后就可以在这个区域内书写自己的注释了。Emacs 除了可以产生注释的占位符外，还能做更多的工作，它能把一大段代码注释掉或者取消注释。

2.2.3 预处理扩展

Emacs 可以使用 C 预处理器运行代码的一部分，以便让程序员检测宏、条件编译以及 include 语句的效果。步骤如下：

- 在顶层窗口输入看到的代码。
- 高亮显示一个区域。
- 键入 C-c C-e 调用宏扩展。

Emacs 将创建第二个窗口，显示宏扩展的结果。

2.2.4 自动状态

当运行在自动状态下，程序员输入代码时，C 环境会自动插入新行、处理缩进并会完成其他相关的任务。

当用户键入 C-c C-a 或者运行 M-x c-toggle-auto RET，系统进入自动状态。在这种状态下，如果输入分号，编辑器就会自动把光标定位到下一行，并自动缩进。

如果想要关闭自动状态，只要键入 C-c C-a 或者运行 M-x c-toggle-auto RET，系统就会恢复一般状态。

2.2.5 使用 Emacs 进行编译和调试

Emacs 是一个集成开发环境，用它可以编译和调试程序。

用户在 Tools 菜单中找到 Compile 选项，就可以输入编译命令了。如果有 makefile 文件，就接受默认设置。如果用命令行，键入 M-x compile，然后输入编译命令就可以了。如果有 makefile 文件，默认设置就是用 make -k 来编译程序。编译出现错误和警告时，程序员可以单击鼠标中键来定位这些警告和错误信息。

出现了警告和错误，就离不开调试这一重要步骤。用户可以使用命令 M-x gdb 来调用 gdb 调试器，或者在菜单中选择 tools 中的 gdb 选项。然后输入调试命令就行了。

2.3 小结与练习

2.3.1 小结

本章主要介绍了一个 Linux 中的编辑器 Emacs。它是一个集成开发环境，功能非常强大，还提供了很多很好的 C 模式下的编辑特性。本章介绍了 Emacs 中的一些基本操作，还专门介绍了一下 C 模式下的重要特性，最后简单阐述了怎样使用 Emacs 对程序进行编译。

2.3.2 习题与思考

在 Emacs 的 C 模式下编辑下面的程序：

```
/*判断所输入的年份是否为闰年*/  
main()  
{  
    int year, leap;  
    scanf("%d",&year);  
    if(year%4==0)  
    {  
        if(year%100==0)  
        {  
            if(year%400==0)  
                leap=1;  
            else  
                leap=0;  
        }  
        else  
            leap=1;  
    }  
    else  
        leap=0;  
  
    if(leap)
```

```
printf("%d is",year);
else
printf("%d is not a leap year.",year);
}
```

第3章

C 语言编译器 gcc

gcc 的使用

gcc 编译流程简介

其他编译调试工具

小结与练习

在 Linux 开发环境下，gcc 是进行 C 程序开发不可或缺的编译工具。gcc 是 GUN C Compile 的缩写，它是在 GUN/Linux 系统下的标准 C 编译器。

本章将就 gcc 的使用作简要的介绍。首先以一个简单的程序向读者介绍使用 gcc 编译器编译单一文件的最基本的命令，然后将概括介绍 gcc 的用法并对编译警告信息、调试符号和优化进行讨论，随后介绍 gcc 在 Linux 平台下的编译过程和用 gcc 创建可执行文件的工具和程序，最后介绍一些其他的 C 编译工具。

3.1 gcc 的使用

当用户把 Red Hat Linux 7.0 安装好之后，gcc 也已经安装好了。gcc 作为 Linux 平台下的标准 C 编译器，功能强大。本节将就 gcc 的基本使用作简要介绍，包括编译单个文件。本节还要讨论编译警告信息的处理，调试符号的使用以及进行优化处理。下一节将介绍关于使用库连接和使用管道编译。

3.1.1 一个最基本的实例

首先通过一个非常简单的例子为读者建立一个用 gcc 编译程序的初步概念，并指出一个警告信息来试验 gcc 的性能。程序如下：

【程序 3_1】

```
#include <stdio.h>
int main(void)
{
    printf("Hello World\n");
}
```

将这个程序命名为 test3_1.c，用户可以输入命令对程序进行编译：

```
gcc test3_1.c
```

编译的结果将会出现一个警告信息。即使得到一条警告信息，gcc 还是能编译完程序，但遇到错误则会停止编译。

用户暂时先跳过警告信息，通过上述命令，gcc 将创建一个名叫 a.out 的文件，里边包含上述程序。然后使用下面命令执行这个程序：

```
$ ./a.out
```

Tip

“.”表示执行当前目录下的可执行文件或脚本程序。

然后读者将得到：

Hello World!

这就是在 Linux 平台上编译的一个 C 程序。

用户还可以通过使用选项-o 来改变编译后的文件名，如使用下面的命令行可以把 a.out 改成 test3_1。

```
gcc -o test3_1 test3_1.c
```

这时可执行文件名就改变成 test3_1，而不是 a.out，如同运行 a.out 一样，可以用命令语句执行新命名的文件。

```
./test3_1
```

同样会得到输出：

Hello World!

3.1.2 gcc 的用法

1. gcc 选项

前面以一个简单的实例向用户介绍了 gcc 的使用，下面给出 gcc 的完整格式。

```
gcc [options] [filenames]
```

上述命令行按编译选项(参数 option)指定的操作对给定的文件(参数 filenames 所指文件)进行编译处理。在 gcc 后面可以有多个编译选项，同时进行多个编译操作。gcc 的主要选项如表 3-1 所示。

表 3-1

选项对照表

选 项	选项描述
-x language	指定使用的语言(C、C++或汇编)
-c	只对文件进行编译和汇编，但不进行连接
-S	只对文件进行编译，但不汇编和连接

续表

选 项	选项描述
-E	只对文件进行预处理，但不编译汇编和连接
-o[file1]file2	将文件 file2 编译成可执行文件 file1
-l library	用来指定所使用的库文件
-I directory	为 include 文件的搜索指定目录
-w	禁止警告信息
-pedantic	严格要求符合 ANSI 标准
-Wall	显示附加的警告信息
-g	显示排错信息以便用于 gdb
-p	产生 prof 所需的信息
-pg	产生 gprof 所使用的信息
-O(-O1)	对编译出的代码进行优化
-O2	进行比-O 高一级别的优化
-O3	产生更高级别的优化
-v	显示 gcc 版本
-m***	根据不同的微处理器进行优化

Tip

选项能分辨大小写，所以使用时要特别注意。

gcc 中总共有一百多个编译选项，其中很多用户可能永远也不会用到，而很多用户只会偶尔用到，上表列出的是一些主要的选项，如果用户想得到所有选项的完整列表和各选项的具体说明，可以在命令行输入命令：

```
man gcc
```

下面将介绍一些选项的使用方法。

2. 选项的使用介绍

关于警告和优化的选项将在后面单独介绍，这里只简要介绍一下其他类型的选项的使用方法。

(1) -c 选项

这一选项告诉 gcc 只把源代码(.c 文件)编译成目标代码(.o 文件)，但跳过了汇编和连接两步。它能使编译多个 C 程序时的速度更快且更加容易管理，所以这个选项经常使用。此选

项缺省时，gcc 建立的目标代码文件只有一个.o 的扩展名。

例如，用户要将已经编辑好的 test.c 文件编译成名为“test.o”的目标文件，可以使用命令。

```
gcc -c test.c
```

(2) -S 选项

此选项告诉 gcc 在为 C 程序文件产生了汇编语言文件后停止编译，产生的汇编语言文件的缺省文件扩展名为.s。

例如下句命令将产生名为“test.s”的目标文件。

```
gcc -S test.c
```

(3) -E 选项

使用此选项指示编译器只对输入的文件进行预处理，且预处理的输出将被送到标准输出而不是储存在文件里。

(4) -v 选项

使用此选项，用户将会得到自己目前正在使用的 gcc 的版本以及与 gcc 版本相关的一些信息。还能确定现在所用的是 ELF 还是 a.out。



a.out 格式和 ELF 格式。

a.out 格式是指 Linux 过去使用的旧的二进制格式，而 ELF(Executable and Linking Format) 格式是另一种二进制格式，目前正应用于许多系统上。由于 ELF 格式增长的弹性远远超过了 Linux 过去的 a.out 格式，而且 ELF 还在设置共享程序库上拥有更大的便利性，在 1995 年，gcc 和 C 程序库的发展人士决定改用 ELF 作为 Linux 标准的二进制格式。

例如在某个系统上执行命令：

```
$gcc -v
```

将得到如下结果：

```
Reading specs from /usr/lib/gcc-lib/i486-linux/2.7.2/specs  
GCC version 2.7.2
```

这个结果告诉用户一些信息。

表 3-2

gcc 版本信息对照表

信息符号	描述说明
I486	指定现在正在使用的 gcc 是为了 486 微处理器写的
Box	这项信息不产生任何作用
Linux	如果版本序号是 2.7.0 或更新版本则是 ELF 格式否则为 a.out 格式; 如果信息符号是“Linuxaout”则是指 a.out 格式, 且不可能出现比 2.7.0 更新的版本 gcc 的 Linuxaout
2.7.0	指的是该 gcc 的版本号, 显然产生的 ELF 的程序

如果安装了多个版本的 gcc, 并且想强制执行其中某个版本, 可以用命令通知系统用户要使用的版本, 但此时选项应该用-V, 注意此时是大写的 V。

例如输入命令:

```
$gcc -V 2.6.3 -v
```

此时用户将强制执行 2.6.3 版本。

(5) -m***选项

-m***是另外一类优化编译的选项, 其中“***”代表的是用户所用的中央处理器的型号。例如-m486, 用来告诉 gcc 该把正在编译的程序视作专为 486 微处理器所写, 当它和用户机器的中央处理器型号正好匹配时, 将可以取得最佳的优化效果。

3.1.3 警告

现在大家来讨论一下刚才的那个警告。在 gcc 中用开关 -W 控制警告信息命令如下:

```
$gcc -Wall -o test3_1 test3_1.c
```

将得到如下信息:

```
test3_1.c:5: warning: control reaches end of non-void function
```

出现此警告信息的原因是因为 main() 函数被声明为返回一个整型, 但实际上是没有被定义。

程序可以改正如下:

【程序 3_1a】

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

```
#include <stdlib.h>           /* for EXIT_SUCCESS */
int main(void)
{
    print("Hello World!");
    return EXIT_SUCCESS;
}
```

然后重新编译它，这时就没有任何警告信息了。

gcc 的警告信息对程序员编程非常有帮助，其中的-Wall 选项是跟踪和调试的有力工具，读者在刚开始学习时最好养成使用-Wall 选项的习惯，这样对以后复杂的编程很有帮助。

3.1.4 优化 gcc

优化是现代 C 编译器的一个最令人心动的特性。优化是编译器的一部分，它可以检查和组合编译器生成的代码，指出未达到最优的部分，并重新生成它们，从而使用户编写的程序更加完美且更加节省空间。作为 Linux 平台上的 C 标准编译器，gcc 拥有强大并且是可配置的优化器，从而对程序进行优化处理。

1. 优化选项

在 gcc 中，用户可以使用-O1(或-O)、-O2、-O3 选项来对代码进行优化，它们分别代表不同的优化级别，数字越高，代表 gcc 的优化级别就越高，而高的优化级别又代表着程序将运行的更快。通常使用的是第三级别的优化。

执行如下命令行，对程序进行最基本的优化：

```
$gcc -Wall -O1 -o test3_1 test3_1.c
```

2. 优化的问题

优化虽然会给程序带来更加优异的表现，但同时读者也必须清楚地意识到还会存在一些潜在的危险。

- 首先，虽然越高级别的优化将使程序运行的越快，但它带来了一个负面的效果，就是编译程序的时间也将变得越长。这就提醒程序员们在集中开发的时候，最好不要使用优化选项，而在版本发行的时候或开发工作快结束的时候，再使用优化。
- 其次，优化级别越高则程序量越大，特别是-O3 级别。程序运行时，需要有一定量的交换空间，它和程序对 RAM 的需求成正比，在一定程度下，过于庞大的程序会带来负面效果。
- 最后，当使用优化选项时，程序的调试将变得比较困难。原因是优化器可能会删除在最终版本中不用的程序代码，或为了取得更加优异的表现而重组许多声明，那么

跟踪可执行文件将变得非常困难。所以在调试程序时，尽量避免使用优化选项。

综合上面几点，`-O2` 选项是在优化长度，编译时间和代码大小之间的最佳优化选项，成为很多程序员的最终方案。

3. 一个优化的实例

一般程序代码编辑后存为 `test3_2.c`，此程序代码有意写得效率很低，从而使读者能够在程序运行过程中看出 `gcc` 优化器对程序的执行速度所起的巨大作用。读者请注意比较没有优化和经过优化的结果之间存在的显著区别。

下面给出的是 `test3_2.c` 的代码。

【程序 3_2】

```
#include <stdio.h>

int main(void)
{
    int counter;
    int ending;
    int temp;
    int five;

    for(counter=0;counter<2*100000000*9/18+5131;
        cunnter+=(5-3)/2)
    {
        temp=counter/15302;
        ending=counter;
        five=5;
    }
    printf("five=%d;ending=%d\n",five,ending);
    return 0;
}
```

首先，用户不加优化地编译它。

```
$gcc -Wall -o test3_2 test3_2.c
```

通过 `gcc` 编译器将程序 `test3_2.c` 编译成了可执行文件，用户只要输入命令 “`./test3_2`” 就可以运行它了。为了和优化后进行比较，用户还需获得一些程序的资源使用信息，此时最好

退出其他的程序，并记录该程序的运行时间。用户可以通过 time 命令做到这点。

下面是命令行：

```
$time v./test3_2
```

用户得到如下信息：

```
five = 5;ending = 100005130
```

```
real    0m15.146s  
user    0m14.960s  
sys     0m0.000s
```

因为在配置不同的机器上运行此程序，所以可能得到不同的结果。

通过执行 time 命令，上段代码说明了以下信息：

这个程序的运行共消耗了 15 秒多的时间，其中大概有 14.9 秒的时间是用于 CPU 的运行。
最后的 sys 值表示处理系统调用消耗的时间，说明运行这个程序时几乎所有的时间都花在计算上，唯一的输出发生在 printf() 函数中。

现在来看看使用了优化设置后的情况。

再次使用 gcc 编译上面程序，此时命令语句如下：

```
$gcc -Wall -O1 -o test3_2 test3_2.c
```

利用 time 命令检测这次运行程序所需的时间：

```
$time ./test3_2
```

可以得到如下输出结果：

```
five = 5; ending = 100005130
```

```
real    0m2.220s  
user    0m2.220s  
sys     0m0.000s
```

经过-O1 级别优化，程序性能发生了很大改变，程序运行所需的时间由 15 秒缩短成了 2 秒左右。

同上步骤，分别使用-O2 级别和-O3 级别来优化程序，还是用 time 命令检测程序运行的时间。先使用-O2 优化：

```
$gcc -Wall -O2 -o test3_2 test3_2.c  
$time ./test3_2
```

得到如下输出：

```
five = 5; ending = 100005130
```

```
real      0m1.444s  
user      0m1.420s  
sys       0m0.000s
```

再使用-O3 优化：

```
$gcc -Wall -O3 -o test3b test3b.c  
$time ./test3b
```

得到如下输出：

```
five = 5; ending = 100005130
```

```
real      0m1.421s  
user      0m1.400s  
sys       0m0.000s
```

从以上结果可以看出，使用-O2 优化所消耗的时间是-O1 优化的 75%，而-O3 优化比-O2 优化后的性能又有所提高，虽然不明显，但对于一个要消耗几个小时或者更长时间的程序，也将节省大量的时间。

这段程序是有意设计的性能很低，主要是为了比较优化前后和不同级别的优化后的性能。从程序中可以看到如下问题：

- for 循环的结束值每次都要重新计算，如果改变代码，直接写出结束值，则代码的性能将会提高。当使用了优化选项后，优化器会简单的计算这个值，避免每次重复计算。
- 每次步长的增加也和上一个问题一样，每次循环也会计算。
- 临时变量 temp 根本没有使用，但是程序在每次循环中都要为它分配资源，降低了性能。
- 变量 five 虽然被使用，但是没有必要每次循环为它分配值，只要在循环前作一次赋值就行了。

- 以上几步看出，其实循环体是空的，所以可以删除掉。

下面写出改动后的程序。

【程序 3_2a】

```
#include <stdio.h>

int main(void)
{
    int ending =100005130;
    int five =5;
    printf("five=%d;ending=%d\n",five,ending);
    return 0;
}
```

进行编译：

```
$gcc -Wall -o test3_2 test3_2.c
```

编译时没有对程序进行优化。

用户还是用 time 命令来检测一下改动过后程序运行所需的时间。

```
$time ./test3_2
```

得到如下输出：

```
five = 5; ending = 100005130
```

real	0m0.004s
user	0m0.000s
sys	0m0.000s

发现时间产生了令人难以置信的变化，即使是原来的程序经过了 -O3 级别优化后的性能和现在比起来，也是望尘莫及的。



从上面比较中可以看出以下两点：

首先就是 gcc 编译器的功能强大，可以大幅度地提高程序的性能。

其次代码的质量高，对速度的提高效果比优化器更加明显。

3.1.5 调试标记

现代的开发系统具备强大的调试工具，它们为程序员们提供了跟踪程序执行的手段和解决问题的方法。作为最常用的 `gdb(GNU Debugger)` 调试工具也不例外。本书将在下一章详细介绍 `gdb` 的使用，这里只是向读者介绍如何编译程序，以便让 `gdb` 能够调试它们。

在正确使用 `gdb` 之前，首先学习使用调试符号来编译程序。在进行编译的时候，`gcc` 在目标文件(.o)和创建的可执行文件中插入额外信息。这些额外信息使 `gdb` 能够判断编译过的代码和程序源代码之间的关系，没有这些信息 `gdb` 将无法判断源程序中的哪行代码在被执行。在默认的情况下，调试符号不会编译到程序中，因为那样会增大可执行文件的体积。调试之后，不用重新编译程序。

在此需要注意一点，调试符号的使用可能与优化不兼容，两者混用，可能导致调试的混乱和失败。所以，在调试一段写好的代码时，应该尽量避免使用-O 选项和优化开关-f 选项。

可以使用 `gcc` 的-g 选项来产生调试符号，它是一个默认的调试选项，命令行如下：

```
$gcc -g -Wall -o test3_1 test3_1.c
```

使用 `gdb` 调试器(或是它的后继产品)，命令行如下：

```
$gcc -ggdb3 -Wall -o test3_1 test3_1.c
```

这行命令中的 `gdb` 告诉 `gcc`，使用 `gdb` 的扩展产生调试符号。其中“3”表示使用的是第三级（最高级）调试信息，程序员可以由此获得最多的信息。

下面举一个实例来说明如何使用调试符号来分析错误。

程序代码如下：

【程序 3_3】

```
#include <stdio.h>

int main(void)
{
    int input =0;
    printf("Enter an integer:");
    scanf("%d", &input);
    printf("Twice the number you supplied is %d.\n", 2*input);
    return 0;
}
```

用命令语句对程序进行编译：

```
$gcc -Wall -o test test.c
```

编译后会出现一条警告信息，指出程序的第6行存在潜在的错误，可以暂时忽略此警告信息，运行程序。

```
$./test
```

得到如下输出：

```
Enter an integer :5
```

```
Segmentation fault
```

程序产生了错误，现在使用调试符号来编译它：

```
$gcc -ggdb3 -Wall -o test test.c
```

编译允许使用存储信息，可以在 bash(bash是大多数Linux系统默认得shell)中使用命令语句。

```
$ulimit -c unlimited
```

然后再次运行程序：

```
$./test
```

得到如下输出：

```
Enter an integer:5
```

```
Segmentation fault(core dumped)
```

这段输出还是有相同的错误，但是多了一个叫 core 的文件，这个文件告诉程序员程序到底错在哪里。

下句命令将加载程序和 core 文件到 gdb 进行分析。

```
$gdb test core
```

分析会得到一段很长的输出，这里只是列出最后四行。

```
Program terminated with signal 11, Segmentation fault.  
Reading symbols from /lib/libc.so.6...done.  
Reading symbols from /lib/ld-Linux.so.2...done.  
#0 0x400686fb in _IO_vfscanf() from /lib/libc.so.6
```

从上可知出错的地方发生在有单词 `scanf(_IO_vfscanf())` 的地方，程序是因为段错误而导致出错，说明内存操作有了问题。

如果想知道更加详尽的信息，可以输入语句：

```
(gdb)bt
```

得到如下输出：

```
#0 0x400686fb in _IO_vfscanf() from /lib/libc.so.6  
#1 0x4006a048 in scanf() from /lib/libc.so.6  
#2 0x8048448 in main() at test.c:6
```

前两句是由 C 函数库来完成，跳过这两句。由第三句知道程序 `test.c` 的第六行出现错误。找出错误原因(第六行的 `input` 前面忘了加`&`)并改正错误。重新编译，程序正常运行。

其实调试的步骤不只这些，对于 `gdb` 到底能干什么，以及更详细的内容，本书将在第 4 章中作具体的阐述。

3.1.6 使用高级 `gcc` 选项

1. 管理大型项目

当处理的项目非常小时，经常把所有代码放在一个文件里，然后用 `gcc` 编译它就行了。然而当代码量达到一定的长度时，这种方法就不可取了，可能会出现这样或是那样的问题。例如：在含有成千上万行代码的文件里很难查询到所需要的内容；编辑器的效率开始降低；对内存的需要开始无限制的增大；当同一开发小组中的不同人员要使用一个文件时，很难协调工作；每次对程序中一行的改变将要重新编译整个程序，非常浪费时间。上述问题都将导致工作效率的下降。

基于以上原因，C 为程序员提供了一种把文件分开的方法，通过使用分离的 C 模块、函数或被包含在不同的被编译过的.c 文件里的数据，把工程分成符合逻辑的不同部分。这时各个部分拥有合适的大小，使阅读程序变得很容易，开发一个工程也比较容易管理。当出现问题时，只要修改特定的文件，修改后，也不需重新编译整个程序，这样可以大大地提高开发

的效率。

在现代的开发工作中，代码的可重用性是一个非常重要的标准。把代码合理地分成多个独立的模块，然后再把它们组成一个库，这样就能在其他的工程中重复使用了。

现在假定程序有 3 个模块，分别为 test3_1.c, test3_2.c, test3.c。可以使用下面的方法编译整个程序：

```
$gcc -Wall -o program test3_1.c test3_2.c test3.c
```

使用上述命令，gcc 将编译每个.c 程序，并把它们连接起来成为一个可执行文件。用此方法时，如果内容稍有改动，就要重新编译全部程序。

现在把编译分成独立的步骤，先编译每一个程序，使用 gcc 的-c 选项，程序生成一个.o 文件，这个.o 文件只包含一个.c 文件的内容，它不是最终的可执行文件。

下面是具体命令：

```
$gcc -Wall -c -o test3_1 test3_1.c  
$gcc -Wall -c -o test3_2 test3_2.c  
$gcc -Wall -c -o test3 test3.c
```

最后使用命令将 3 个.o 文件生成一个可执行文件。

```
$ gcc -o program test3_1.o test3_2.o test3.o
```

这样在任何一个文件中改变一行代码，不用重新编译另外两个文件，只要重新编译所改动的文件并且重新连接即可。例如：

```
$ gcc -Wall -c -o test3_1 test3_1.c  
$ gcc -o program test3_1.o test3_2.o test3.o
```

在一个包含几百个甚至更多的文件的工程中，这种方法将带来很多的便利。但是操作显得有些复杂，在第 5 章中我们将学习如何利用 make 来管理大型项目，所需的命令也将更加简单。

2. 指定查找路径

当用户建立一个项目时，gcc 将使用默认的路径，去查找和使用头文件和库文件。但还会碰到另外一些情况，例如：在编写一个程序时，需要在查询路径下添加一个目录，使连接器可以找到程序所需的库；当编译一个程序时，需要查找它所需的头文件。这时可以使用-I 和-L 选项。

当编译的程序要包含一个文件 zw.h，而这个文件在 /usr/include/zw 目录下，不在默认的查询目录下时，可以使用下句命令：

```
gcc -Wall -I/usr/include/zw -o test test.c
```

预处理器就能找到程序所需的 zw.h 文件。

当连接 X11 库时，可以使用相同的方法处理库路径，但注意要告诉连接器库的路径，命令行如下：

```
gcc -L/usr/X11R6/lib -Wall -o test test.c -lX11
```

3. 连接库

如果要在程序中连接库，可以使用 -l 选项，这个库可以是静态的，也可以是共享的。这个选项只能在编译中连接的最后阶段才能指定(前面要对 makefile 文件进行修改，使它在连接的最后阶段包含库)，它把所有的.o 文件连接起来，假设连接的是 zw 库，命令如下：

```
gcc -o test test3a.o test3b.o test3.o -lm
```

也可以直接把.c 源文件编译成最终的可执行文件，命令如下：

```
gcc -Wall -o test test.c -lm
```

4. 使用管道

管道实现的是使管道前的输出成为管道后的输入。通过使用管道，可以同时调用多个程序，一个程序的输出作为输入直接送给另外一个程序，而且还可以一直连续下去，不需要临时文件。

管道编译过程有 gcc 的 -pipe 选项决定，使用这个选项，gcc 能建立适当的管道。下面是命令行：

```
gcc -pipe -Wall -O3 -o test test.c
```

当编译一个大程序时，使用此方法将节省大量的时间。

3.2 gcc 编译流程简介

编译过程还隐藏着很多的细节，了解那些隐藏的细节可以帮助我们深入地理解产生的大量警告和错误信息，从而可以更精确地编译程序和控制连接。

3.2.1 C 预处理器 cpp

C 预处理器 cpp 是用来完成宏的求值、条件编译、以及其他一些需要把代码传递到编译器前的工作。通常所见的那些“#”号后面的语句由 cpp 来进行处理。来看下面一段代码。

```
#define FOO (5*2)
.....
printf("%d\n", FOO*2);
.....
```

经过 cpp 预处理后，代码变成下面形式：

```
printf("%d\n", (5*2)*2);
```

可以看出，cpp 完成了以下工作：

解释宏，处理包含文件，处理 #if 和 #ifdef 声明，还有其他以#开头的标志。可以在命令行中使用 cpp，也可以使用 gcc -E 调用 cpp，通常 gcc 编译器会自动调用 cpp，前面的实例中就是 gcc 自动调用 cpp。

3.2.2 GUN 连接器 ld

在编写一个大程序时，经常把它分成许多独立的模块，这时需要连接器把所有的模块组合起来，并结合 C 函数库和初始化代码，产生最后的可执行文件。连接器在产生可执行文件之前，起到重要的作用。

通常情况下，ld 被编译器所调用，产生可执行代码，但是如果想更好地控制连接过程，最好手工调用 ld。

3.2.3 GUN 汇编器 as

使用 gcc 编译程序时，产生汇编代码，as 会处理这些汇编代码，从而产生目标文件(二进制文件)，而目标文件将生成.o 文件、库或者最终的可执行文件。as 像前两项一样，通常情

况下是被 gcc 调用的，但是想使用汇编语言编写程序，可以手工调用。

3.2.4 文件处理器 ar

可以使用 ar 程序建立静态库，把几个小文件组合成一个大文件。建立静态库时，必须把多个.o 文件组合成一个单独的.a 文件。

3.2.5 库显示 ldd

一个可执行文件可能要使用一些共享函数库，可以通过 ldd 工具显示它们，ldd 是 Library Dependency Display 的缩写。

命令行如下：

```
$ ldd ./test
```

3.3 其他编译调试工具

gcc 是 Linux 上 C 程序的标准编译器，但是除了 gcc 之外，还有其他一些编译器。本书在这里简要地介绍几种，读者如果感兴趣的话，可以去查阅有关书籍。

3.3.1 C++编译器 g++

GUN C++编译器 g++和 C 编译器 gcc 的格式相同，完成的工作也是一样的。gcc 也可以编译 C++代码，但是需要手工设置一些特殊选项，而且很容易导致错误。

GUN C++编译器 g++所使用的选项和 gcc 一样。为了与 C 代码相区别，程序员对 C++代码常使用.cxx 扩展名(也可以使用.c 扩展名)，告诉 C++编译器和其他程序员这是 C++代码。

命令格式如下：

```
g++ [-options] [filenames]
```

3.3.2 EGCS

EGCS 是 gcc 的发展方向，它集成了 Fortran 等编译器，还集成了对 gcc 的各种改进和 pgcc 对 Pentium 的一些优化。

3.3.3 calls

它是用来调用 gcc 的预处理器来处理给出的源程序文件，并输出这些文件里的函数调用树图。当打印出调用跟踪结果时，它在函数后面用中括号给出函数所在文件的文件名。

文件格式如下：

```
main [test.c]
```

3.3.4 indent

通过设置 indent 的选项可以指定如何格式化写好的源代码，使源代码产生统一的缩进格式。

3.3.5 gprof

gprof 将告诉程序中的每个函数被调用的次数和每个函数执行时所占的时间。在编译程序时加上 -pg 选项，就可以在程序中使用 gprof，它在程序每次执行时产生一个叫 gmon.out 的文件，gprof 就使用这个文件来剖析信息。

下面是命令格式：

```
gprof <filename>
```

 **Tip** gprof 产生的剖析数据很大，如果想查看这些数据，最好把输出重定向(与管道作用相似)到一个文件中。

3.3.6 f2c 和 p2c

f2c 把 Fortran 代码转换成 C 代码，p2c 把 PASCAL 代码转换成 C 代码。一般转换小程序时可以直接使用它们，不需要用到命令行选项。

3.4 小结与练习

3.4.1 小结

本章首先从一个简单的 C 程序入手，引出 Linux 系统中的 C 程序编译器 gcc。具体介绍

了 gcc 的用法，包括它的属性选项、优化和其他的高级应用。在本章的最后，简略介绍了一些其他的 C 程序编译器。

3.4.2 习题与思考

编译下段程序(hello.c)代码，然后执行它。

【程序 3_4】

```
#include <stdio.h>

main( )
{
    char my_string[]="Hello World!";
    my_print(my_string);
    my_print2(my_string);
}

void my_print(char *string)
{
    printf("The string is %s\n", string);
}

void my_print2(char *string)
{
    char *string2;
    int size, size2, j;
    size=strlen(string);
    size2=size-1;
    string2=(char *)malloc(size+1);
    for(j=0;j<size;j++)
        string2[size2-j]=string[j];
    string2[size]='0';
    printf("The string printed backward is %s\n", string2);
}
```

第4章

调试工具 gdb

gdb 符号调试器简介

gdb 功能详解及其应用

其他调试工具

小结与练习



本章将介绍一个功能强大且非常有用的调试工具 `gdb`。第一节简单地介绍 `gdb`；第二节将详尽地阐述它的使用方法，其中包括调试的步骤，数据的显示以及断点和观察窗口的使用等；在最后一节中，将向读者介绍一些别的调试工具。

4.1 `gdb` 符号调试器简介

即使是最优秀的程序员也不可能避免在编程时出现一些这样或那样的错误。所有的程序在写好以后，都要经过调试，在调试过程中发现并改正程序中的错误。

`gdb` 是一个用来调试 C 和 C++ 程序的功能强大的调试器，它能在程序运行时观察程序的内部结构和内存的使用情况。如果没有 `gdb`，程序员为了跟踪某些错误，就要在程序中加入大量的语句，用来产生一些特定的输出。对于某些程序来说，这样做会导致更多的错误。

Tip

`gdb` 主要提供以下功能：

监视程序中变量的值的变化；

设置断点，使程序在指定的代码行上暂停执行，便于观察；

单步执行代码；

分析崩溃程序产生的 `core` 文件。

`gdb` 的使用非常简单，只要在 `bash` 命令行下直接输入 `gdb` 就可以运行 `gdb`(这时在屏幕上将看到一些提示信息)。也可以在 `gdb` 后面给出文件名，直接指定想要调试的程序。

命令形式如下：

`gdb filename`

告诉 `gdb` 装入名为 `filename` 的可执行文件进行调试。

Tip

用户可以通过在 `gdb` 下输入 `help` 来查看如何使用 `gdb`，或在命令行上输入 `gdb -h` 来得到一个有关它的选项的说明的简单列表。

为了使 `gdb` 能够正常地工作，必须使写好的程序在编译时包含调试信息，在前一章中已经详细讲过，这里就不再重复，只强调在用 `gcc` 编译时加上`-g` 选项。

4.2 gdb 功能详解及其应用

前一节中简要地介绍了 gdb 以及它的主要功能，下面将结合实例对每种功能做详细的阐述。首先介绍 gdb 的调试步骤。

4.2.1 调试步骤

通过使用 gdb 逐步调试代码，可以看到程序内部是如何运行的，还可以知道什么命令正在执行，变量的值的变化以及其他一些细节问题。

下面以一个实例介绍调试的具体步骤，读者通过此程序可以学到怎样跟踪写好的程序代码，并且掌握如何运用一些技巧跟踪错误代码。

源程序(test4_1.c)如下：

【程序 4_1】

```
#include <stdio.h>

int getinput(void);
void printmessage(int counter, int input);

int main(void){
    int counter;
    int input;

    for(counter=0;counter<=200;counter++){
        input=getinput();
        if(input== -1) end(0);
        printmessage(counter, input);
    }
    return0;
}

int getinput(void){
    int input;

    printf("Enter an integer, or use -1 to exit: ");
}
```

```
scanf("%d", &input);
return input;
}

void printmessage(int counter, int input){
    static int lastnum=0;
    counter++;

    printf("For number %d , you entered %d (%d more than last time)\n",
           counter, input, input-lastnum);
    lastnum=input;
}
```

先来解释一下这段代码需要注意的地方。

首先，可以看出主函数中有变量 counter，函数 printmessage 中也有变量 counter。在函数 printmessage 中的 counter 是局部变量，它只在函数 printmessage 中有效，这个变量在调用函数 printmessage 时就被初始化成和 main 函数中的变量 counter 一样的值，当函数 printmessage 被调用完之后，它将被释放，所以它和 main 中的 counter 变量是不同的。在函数调用时，参数值的传递是非常复杂的，感兴趣的读者最好看看专门的 C 语言方面的书籍。

其次，在 printmessage 中的变量 lastnum 的声明，用了 static int，说明它是一个静态变量，即当函数 printmessage 退出后，这个变量的值还将保存起来，以等待下一次调用。

在上一章已经讲过，使用 gcc 编译器的-ggdb3 选项将在可执行文件中最大限度地包含调试信息。下面就用这个选项来编译并运行这个程序。

具体命令如下：

```
$gcc -ggdb3 -o test4_1 test4_1.c
$./test4_1
```

得到输出并按照提示输入一些数字。

```
Enter an integer, or use -1 to exit: 215
For number 1, you entered 215 (215 more than last time)
Enter an integer, or use -1 to exit: 300
For number 1, you entered 300 (85 more than last time)
Enter an integer, or use -1 to exit: 100
For number 1, you entered 100 (-200 more than last time)
Enter an integer, or use -1 to exit: 5
For number 1, you entered 5 (-95 more than last time)
```

Enter an integer, or use -1 to exit: -1

接下来看看调试器内部的情况，下面的命令实现的是和 gdb 的交互即调试的步骤，将边演示边向读者解释。

1. 调用 gdb

要调试程序，首先要做的当然是调用调试器 gdb，装载子程序，用到下面命令：

```
$gdb test4_1
```

得到如下输出(可能因为系统的不同，会有些小差异，但不影响下面的学习)：

GUN gdb 4.18

Copyright 1998 Free Software Foundation, Inc

GDB is free software, covered by the GUN General Public Licence, and you are welcome to change it and/or distribute copies of it under certain conditions.

Type "show copying" to see the conditions.

There is absolutely no warranty for GDB. Type "show warranty" for details.

This GDB was configured as "alphaev56-unknown-Linux-gnu"...

(gdb)

(gdb)提示符是 gdb 的主要接口，在这个提示符下，可以向 gdb 输出命令，包括设置断点等。

2. 设置断点

断点指出了 gdb 将要在该点处中断程序的运行，从而便于程序员单步跟踪代码。可以按照下面的方法在程序中设置断点。

```
(gdb)break main
```

上面命令在 main() 处设置了断点，程序在此处将暂停执行。

Tip

除了用上面命令设置断点之外，还可以使用下面格式。

```
(gdb)break 要设置断点的行号
```

然后调试器将确认断点，并显示其所在的位置(行数)。

Breakpoint 1 at 0x1200004a8: file test4_1.c, line 6

然后开始执行程序。}

(gdb)run

得到以下输出：

```
Starting program: /home/jgoerzen/t/test4_1
```

```
Breakpoint 1, main() at test4_1.c: 6
```

```
6    int main(void){
```

程序一运行，立刻就会遇到 main()中的断点，gdb 会指出所遇到的断点，然后显示将要执行的下一行。

可以使用单步跟踪命令跟踪程序代码，它一次将执行程序中的一行代码。

(gdb)step

输出如下。

```
main() at test4_1.c: 10
```

```
10    for(counter=0;counter<200;counter++){
```

上一步执行了第 6 行代码。



在 gdb 中，可以使用很多快捷键来简化操作。



后面不会像以前那样一步一步地写出各个单步调试过程，而是写几步调试命令，再解释这些命令。

```
(gdb)s  
11    input=getinput();  
(gdb)Enter  
getinput() at test4_1.c: 18  
18    int getinput(void){
```

键入 s (step 命令的快捷键) 命令后, 将单步执行第十行代码, 然后进入 for 循环, 在循环中, 程序第 11 行的执行将调入 getinput() 函数。

```
(gdb)s
getinput() at test4_1.c: 21
21      rintf("Enter an integer, or use -1 to exit: ");
(gdb)s
22      canf("%d", &input);
(gdb)print input
$1 = 1439424
```



print 命令。

上面又接触了一个新的命令 print, 它用来打印表达式的值, 还可以用来打印内存中某个变量开始的一段区域的内容。使用格式有几种形式。

(gdb)print 打印的表达式

print 命令后的表达式中有两个符号有特殊意义, 它们是\$和\$\$。\$表示给定序号的前一个序号, 而\$\$表示给定序号的前面第二个序号, 如果不给定序号, gdb 将默认当前序号为给定序号。

```
(gdb)print i
$1=30
(gdb)print $
$2=30
(gdb)print $$
$3=30
(gdb)print $$3
$4=30
```

上面那段代码可以看出输入 print \$时的当前序号是 2, \$表示 2 的前一个序号是 1; 输入 print \$\$时所在的当前序号为 3, 则\$\$表示 1; 输入 print \$\$3 时, 给定了序号为 3, 则\$\$3 表示 3 的前面第二个序号是 1。由此, 就不难理解为什么上面输出的都是 30。

Print 的另一个命令就是用来对变量进行赋值，命令格式如下：

```
(gdb)print 变量=表达式
```

print 还有一个命令就是可以用来打印出内存某个部分开始的一块连续空间的内容，它使用如下格式：

```
(gdb)print 开始表达式@要打印的连续内存空间的大小
```

开始表达式应该是在内存中的一个表达式，它的输出结果是以数组的形式，其中数组的第零个元素是开始表达式的值，第一个元素是在内存中紧挨着开始表达式的空间中存放的值，依此类推。

还有一个问题就是在执行第 21 行之后，应该显示一个提示符，就是 printf 的输出内容，但是并没有看到，产生此现象的原因是 printf() 函数和另一些函数所使用的缓冲区造成的。将在 scanf() 函数被执行之后，才会出现提示符。

```
(gdb)s  
Enter an integer, or use -1 to exit: 150  
23 return input;
```

scanf() 函数被执行了，出现提示符，并从终端读取输入信息。此时 input 的值已经改变了，可以使用前面学的 print 命令检查它的值。

```
(gdb)print input  
$2 = 150
```

可以看到已经对 input 赋值 150 了。赋值后，调用的函数 getinput 将准备返回一个值—input，继续单步跟踪将立刻返回主函数 main()。

```
(gdb)s  
24      }  
(gdb)s  
main() at test4_1.c: 12  
12      if(input== -1)exit(0);
```

继续调试。

```
(gdb)display counter
1: counter=0
(gdb)display input
2: input=150
(gdb)s
13    printmessage(counter, input);
2: input=150
1: counter=0
```



display 命令。

display 命令用来显示一些表达式的值，使用该命令后，每当程序运行到断点处就会显示该表达式的值，可以用它来观察一些表达式的变化。具体使用格式如下：

(gdb)display 要显示值的表达式

在程序运行之后就可以使用这个格式对表达式设置显示值。display 与前面的打印函数的显示效果相同，但是它们之间还是有区别的。在使用 display 命令时，每次调试器中断程序、挂起指令都要显示变量的值。

当单步跟踪完第 12 行之后，gdb 首先在命令行中显示将要执行行的编号，接着显示 counter 和 input 两个变量的值。

继续上面的调试步骤：

```
(gdb)s
printmessage(counter=0, input=150)at test4_1.c: 26
25    void printmessage(int counter, int input){
(gdb)s
printmessage(counter=0, input=150) at test4_1.c: 29
29    counter++;
(gdb)display counter
3:    counter=0
```

继续单步执行程序代码，调试器不再显示变量 counter 和 input 的值。这是为什么呢？大家还记得在本章的开始时曾经解释过代码中需要注意的几个方面，其中就讲到了局部变量，

上面的调试过程中的 counter 和 input 是 printmessage 函数的局部变量，它们只在函数 printmessage 之中有效，而前一部分的变量 counter 和 input 是在主函数 main() 中有效，它们虽然名字相同，但本质不同。然后要求调试器显示变量 counter 的值。

```
(gdb)s
31     printf("For number %d, you entered %d(%d more than last time)\n",
3: counter=1
(gdb)s
For number 1, you entered 150(150 more than last time)
33     lastnum=input;
3: counter=1
(gdb)s
34 }
3: counter=1
```

单步跟踪到此时，发现 counter 的值变大了，设置 lastnum 的值，接着函数准备返回。

```
(gdb)s
main() at test4_1.c: 10
10    for(counter=0;counter<200;counter++){
2: input=150
1: counter=0
```

此时变量 counter 的值又变为 0，这是两个不同的变量。主函数中的变量 counter 的值没有改变，只是 printmessage 中的 counter 改变了。

下面来跟踪全部的循环过程，命令代码和输出结果如下：

```
(gdb)s
11     input=getinput();
2: input=150
1: counter=1
(gdb)s
getinput() at test4_1.c: 18
18     int getinput(void){
(gdb)s
getinput() at test4_1.c: 21
21     printf("Enter an integer, or use -1 to exit: ");
(gdb)s
```

```
22     scanf("%d", &input);
(gdb)s
Enter an integer, or use -1 to exit: 12
23     return input;
(gdb)s
24 }
(gdb)s
main() at test4_1.c: 12
12     if(input== -1)exit(0);
2: input=12
1: counter=1
(gdb)s
13     printmessage(counter, input);
2: input=12
1: counter=1
(gdb)s
printmessage(counter=1, input=12) at test4_1.c: 26
26     void printmessage(int counter, int input){
3: counter=1
(gdb)s
printmessage(counter=1, input=12) at test4_1.c: 29
29     counter++;
3: counter=1
(gdb)s
printf("For number %d, you entered %d(%d more than last time)\n",
3: counter=2
(gdb)s
For number2, you entered 12(-138 more than last time)
33     lastnum=input;
3: counter=2
(gdb)s
34 }
3: counter=2
(gdb)s
main() at test4_1.c: 10
10     for(counter=0;counter<200;counter++){
2: input=12
1: counter=1
```

从结果可以看出，即使每次只是输入命令 `s`，调试器还是输出了比较多的信息，不只限于 `s` 命令。每次调试进入 `printmessage()` 函数时，它都会再次显示在这个函数作用范围之内的变量 `counter` 的值，这是因为 `gdb` 会记住显示命令。

还有一个现象，就是它多次重复了许多信息，当知道程序是如何工作的或它们确实工作正常，就没有必要跟踪那些代码了。可以使用 `gdb` 的 `next` 命令达到这一要求。



`next` 命令。

当不需要跟踪某一段代码，可以使用 `gdb` 的命令 `next`，这条命令和 `step` 命令非常相似，但是它不能跟踪到程序里，正好可以满足上面提到的那个要求。下面的代码可以看出两者的区别。

```
(gdb)next
10      input=getinput();
2: input=12
1: counter=2
(gdb)n
Enter an integer, or use -1 to exit: 10
11      if(input==1)exit(0);
2: input=10
1: counter=2
(gdb)n
12      printmessage(counter, input);
2: input=10
1: counter=2
(gdb )n
For number3, you entered 10(-2 moer than last time)
10      for(counter=0;counter<200;counter++){
2: input=10
1: counter=2
```

这时看到了可以不再强行进入不相关的函数内部，比以前的步骤也少多了，特别是当我们得知错误出现在什么地方的时候，这样做就能节省大量的时间。所以许多程序员在调试程序时经常同时使用 `next` 和 `step` 命令，可以使工作进行得更加高效。

完成了上述调试步骤后，可以使用 `quit` 命令退出 `gdb`。

```
(gdb)quit
```

The program is running. Exit anyway?(y or n) y

4.2.2 显示数据命令 display 和 print

在调用 gdb 进行单步调试程序时，就接触过此类命令，最常用的就是 display 和 print 命令，这部分将要详细介绍它们的使用方法，另外还会介绍一些其他的实现显示数据功能的命令，包括 printf 和 set。

1. 使用 print 和 display

display 和 print 是显示数据最常用的两种命令，这两个命令的功能非常强大，并不限于简单地显示一个整数值。

现在调试下面一个程序，这个程序拥有一些比较复杂的数据结构，包括数据指针等。程序(test4_2.c)代码如下：

【程序 4_2】

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

typedef struct TAG_datastruct{
    char* string;
    int checksum;
}datastruct;

datastruct* gerinput(void);
void printmessage(datastruct* todisp);

int main(void){
    int counter;
    int maxval=0;
    datastruct* svalues[200];

    for(counter=0;counter<200;counter++){
        svalues[counter]=gerinput();
        if(!svalues[counter]) break;
        maxval=counter;
    }
}
```

```
printmessage(svalues[maxval/2]);\n\n    return 0;\n}\n\ndatastruct* getinput(void){\n    char input[80];\n    datastruct* instruct;\n    int counter;\n\n    printf("Enter a string, or leave blank when done: ");\n    fgets(input, 79, stdin);\n    input[strlen(input)-1]=0;\n    if(strlen(input)==0)\n        return NULL;\n    instruct=malloc(sizeof(datastruct));\n    instruct->string=strdup(input);\n    instruct->checksum=0;\n    for(counter=0;counter<strlen(instruct->string);counter++){\n        instruct->checksum +=instruct->string[counter];\n    }\n    return instruct;\n}\n\nvoid printmessage(datastruct* todisp){\n    printf("This structure has a checksum of %d. Its string is: \n",\n          todisp->checksum);\n    puts(todisp->string);\n}
```



首先分析其中的一些符号。

第 16 行的 `datastruct* svalues[200]` 定义了一个数组指针，即数组 `svalues` 中的每一分量都是一个指针，这些指针指向的数据类型都是一个叫 `datastruct` 的结构。在随后的 `for` 循环中调用了 `getinput()` 函数，而该函数返回一个指向结构的指针，然后把指针放在数组中。

第40行开始出现了几次“->”符号，它是成员选择符号，符号右边是左边的成员。

在理解了程序的具体机制后，运行程序，注意观察程序的正常输出，便于调试的时候检测。

程序执行结果如下：

```
$gcc -ggdb3 -Wall -O test4_2 test4_2.c
$./test4_2
Enter a string, or leave blank when done: Hello
Enter a string, or leave blank when done: This is the second line.
Enter a string, or leave blank when done: This is the third
Enter a string, or leave blank when done: gdb is interesting
Enter a string, or leave blank when done: Hmm...!
Enter a string, or leave blank when done: Enter
This struture has a checksum of 1584. Its string is:
This is the third
```

通过前边对程序的了解，知道数据是保存在一个叫 `datastruct` 的结构中。在 `main()` 函数中有一个数组指针 `datastruct* svalues[200]`，它就指向该结构，然后在 `for` 循环中调用了 `getinput()` 函数。这个函数将返回一个指向结构的指针，然后把指针放在数组 `svalues` 中。当这个指针是 `NULL`，循环将在填满 200 个元素之前退出循环，否则，变量 `maxval` 被设置成当前数组元素的索引值。最后打印的是位于数组中间的元素，负责打印信息的是函数 `printmessage()`，打印完成之后程序将退出。

下面开始这部分的调试过程。

```
$gdb test4_2
```

它将会输出很多信息，包括版本信息等，可以忽略掉。然后在程序中设置断点，再运行程序。

```
(gdb)break main
Breakpoint 1 at 0x1200005b8: file test4_2.c, line 15.
(gdb)run
Starting program: /home/jgoerzen/t/test4_2

Breakpoint 1, main() at test4_2.c: 15
14      int maxval=0;
```

由于在第 15 行设置了断点，所以程序运行停在 15 行，并显示出第 14 行的代码。做好上面的准备工作后，将要使用 print 命令来显示 svalues 数组的内容，下面是具体命令：

```
(gdb)print svalues  
$2 ={.....}
```

运行 print 命令后，输出一些数据，此时数组 svalues 的值是内存中的随机内容。可以使用下面的命令来删除一个指针，验证哪些数据是内存中的随机内容。

```
(gdb)print svalues[0]->checksum  
Cannot access memory at address 0x8
```



如果在程序的执行过程中，试图获得程序中变量的值，将会产生越界错误，原因是内存读取问题而导致程序崩溃。

单步跟踪程序的运行，可以获得一些有用的数据。

```
(gdb)s  
18     for(counter=0;counter<200;counter++){  
(gdb)s  
19     svalues[counter]=getinput;  
(gdb)s  
29     datastruct=getinput(void){  
(gdb)s  
printf("Enter a string, or leave blank when done: ");  
(gdb)s  
35     fgets(input, 79, stdin);  
(gdb)s  
Enter a string, or leave blank when done: Hello.  
36     input[strlen(input)-1]=0;  
(gdb)print input  
$3= "Hello.\n\000 ....."
```

从最后的输出中可以看出：输出的字符串只包含一个单词 Hello，这是因为字符串只是一个数组，而置入字符串的数据只是覆盖了内存的开始部分，并没有到达字符串的剩余部分；其次就是在新行字符(\n)后面是空字符(\000)，而空字符表示字符串的末尾。可以利用这一特性，删除新行字符。

```
(gdb)s
37    if(strlen(input)==0)
(gdb)print input
$4="Hello.\....."
```

输出基本相同，只是用\000取代了\n。
可以使用被调试语言的指令来获得数组。

```
(gdb)print input[0]
$5=72  'H'
```

这个 print 命令显示字符串的第一个字符。继续单步跟踪调试程序。

```
(gdb)s
39    alloc(sizeof(datastruct));
(gdb)s
40    instruct->string=strdup(input);
(gdb)s
41    instruct->checksum=0;
(gdb)s
42    for(counter=0;counter<strlen(instruct->string);counter++){
(gdb)print instruct
$6 = (datastruct *)0x120100f80
```

因为 instruct 是一个指针，所以上面在命令 print instruct 后面输出的数据只是内存地址。我们更关心这个指针所指向的结构的值。

```
(gdb)print *instruct
$7 = { string=0x120100fa0Z"Hello.", checksum =0 }
```

调试器输出了结构中的不同成员以及它们的值。

```
(gdb)print instruct->string[0]
$8 =72 'H'
(gdb)s
43    instruct->checksum+=instruct->string[counter];
(gdb)s
```

```
42     for(counter=0;counter<strlen(instruct->string);counter++){
(gdb)s
43     instruct->checksum+=instruct->string[counter];
```

继续执行直到使用 `finish` 命令强制退出。

```
(gdb)finish
```

在此命令之后将输出结果。

2. 内存检查命令

前面在接触 `print` 命令和 `display` 命令时，曾经看到一些内存地址。`gdb` 提供了命令 `x`，来获得指针所指向的内容。

```
x/format address
```

其中 `format` 指定了显示单元的个数，后面是如何显示内存的方式，例如：

```
(gdb)x/2c 0x120100fa0
```

3. 使用 `printf`

除了使用前面介绍的显示数据的方法之外，还可以使用内建的 `printf` 命令，该命令可以接受指定格式和变量列表为参数，使用起来类似于 C 中的 `printf()` 函数。下面就是一个使用 `printf` 的例子：

```
(gdb)printf "%2.2s\n", (char*)0x120100fa0
He
```

4. 使用 `set` 命令

`set` 命令除了可以显示数据外，还能够修改变量的值，在某些时候，它的作用非常重要。例如：要继续跟踪程序，而又想重新设置变量的值的时候，就可以使用 `set` 命令。

例如：

```
(gdb)print svalues[0]->checksum
$1=646
(gdb)set variable svalues[0]->checksum=2000
(gdb)print svalues[0]
$2=2000
```

从上面的命令输出中可以看出变量的值已经被改变，此时返回程序中，变量会维持新的值。

4.2.3 使用断点

前面讲过可以通过单步跟踪整个程序，实现检测变量的值的变化和发现程序错误的出处等功能，但是这种方法实现起来太慢，可以使用其他一些更快的方法来检测程序错误的所在，那就是设置断点和观察窗口。下面将详细介绍怎样在程序中设置断点以及有关设置断点的一些命令。

1. 设置断点

可以通过使用 `break` 命令，指定一个特定的位置设置断点，当程序运行到这个位置时就被中断，然后把程序的控制权交给调试器和程序员。这是设置断点的最简单的一种形式。

仍以程序 `test4_2.c` 为例来讲解断点的设置。

```
$gdb test4_2
```

程序照例要输出一些有关 `gdb` 版本和其他一些介绍 `gdb` 调试器的信息，省略这些无关信息。开始设置断点：

```
(gdb)break test4_2.c: 21  
Breakpoint 1 at 0x12000061c: file test4_2, line 21
```

调试器 `gdb` 在程序 `test4_2.c` 的 21 行设置了一个断点，这是设置断点的第一种方法，下面将介绍第二种方法。

```
(gdb)break printmessage  
Breakpoint 1 at 0x120000848: file test4_2.c, line48
```

这是设置断点的第二种方法，用此方法不需要指定具体的行数，在此指定的是在函数 `printmessage()` 的开始位置设置断点，调试器在这种情况下将会自己找到函数的开始位置。当非常了解被调试的程序时，使用此方法将会更加方便快捷。

现在开始运行程序，一直运行到断点位置。

```
(gdb)run  
Starting program: /home/jgoerzen/t/test4_2  
Enter a string, or leave blank when done: Hello!
```

```
Breakpoint 1, main() at test4_2.c: 21
20      maxval=counter;
```

由于前面设置了两个断点，一个是在第 21 行，另一个是函数 printmessage() 的开始位置，后一个断点在程序的第 48 行。程序被调用运行到第 21 行时，遇到第一个断点，在此位置暂停执行。然后程序员可以根据调试需求，完成各种工作，包括单步跟踪程序的这段代码，或者是要检测一下变量的值。完成这些工作后，使用 continue 命令恢复程序的运行，直到运行到程序的下一个断点或是程序执行完毕。

```
(gdb)s
18      for(counter=0;counter<200;counter++){
(gdb)s
19      svalues[counter]=getinput();
(gdb)continue
Continuing.
Enter a string, or leave blank when done: Hello!
```

```
Breakpoint 1, main() at test4_2.c: 21
20      maxval=counter;
(gdb)continue
Continuing.
Enter a string, or leave blank when done: Enter
```

```
Breakpoint 2, printmessage(todisp=0x100000002) at test4_2.c: 48
48      void printmessage(datastruct* todisp){
```



continue 命令和 cont 命令

两个命令的功能是一样的，都是使程序在信号发生后或是停在断点之后再继续运行。在这部分使用的是后一种用法，即在断点之后继续程序的运行。在这种用法里，可以在该命令后面带一个参数 N，这个参数表达的意思是在以后的程序运行过程中将忽略断点的次数为 N-1，就是说在第 N 次执行到该断点时才暂停程序的运行。这些可以从程序的输出信息中看出。

命令 cont 的使用方法非常简单，只要在 gdb 提示符后直接输出就行，当不带参数时，命令行如下：

(gdb)cont

带参数时的命令行如下：

(gdb)cont N

上面的代码命令中使用了两次让 gdb 继续的命令，每次命令之后，程序都运行到下一个断点处。当第三次使用 continue 命令时，程序将正常退出。

```
(gdb)continue
Continuing.
This structure has a checksum of 533. Its string is:
Hello!
```

Program exited normally.

还可以设置其他的断点，在条件成立时就被引发。当在调试变量为特定值才能执行的代码时，这一功能非常有用。

下面的代码就是实现这一功能：

\$gdb test4_2

忽略输出的信息，跳过去执行以后的命令：

```
(gdb)break 21
Breakpoint 1 at 0x12000061c: file test4_2.c, line 21.
```

装载程序 test4_2，并在第 21 行设置了一个断点。

下面给程序设置条件断点，这将要用到 gdb 的 condition 命令。使用条件断点，首先要指出断点的号码，还有表达式。程序只有在执行到表达式为真的时候才会被中断。

```
(gdb)condition 1 svalues[counter]->checksum>700
(gdb)run
Starting program: /home/jgoerzen/t/test4_2
Enter a string, or leave blank when done: Hi
Enter a string, or leave blank when done: Hello
Enter a string, or leave blank when done: How are you?
```

```
Breakpoint 1, main() at test4_2.c: 21
```

```
21     maxval=counter;
```

在这里，程序将运行到条件表达式 `svalues[counter]->checksum>700` 为真，这时断点起作用，程序将中断运行。

还可以使用 `tbreak` 命令设置一个临时断点。临时断点是 GDB 调试器提供的一种功能，它是只能起一次作用的断点。一旦被激活，断点就自动被删除了。它的操作方法与标准断点相同，也可以使用条件表达式。

```
(gdb)tbreak 41
```

上面的命令在程序代码的 41 行设置了一个临时断点，这个断点只能被激活一次，即使用户多次使用该部分的代码。

临时断点命令 `tbreak` 可与下面的命令行等效。

```
break 43  
enable delete 1
```

此命令表示产生一个断点，并且断点 1 在被激活后删除。



enable 命令

`enable` 命令的功能是恢复暂时失效的断点。命令格式如下：

```
(gdb)enable 断点编号
```

要使恢复多个编号处的断点，可以用空格键把断点编号分开。

disable 命令。

`disable` 命令的功能是使所设置的断点失效。设置断点失效后，可以调用 `cont` 命令继续程序的执行。命令格式如下：

```
(gdb)disable 断点编号
```

要使多个编号处的断点失效，可以将断点编号之间用空格键隔开。



delete 命令

这个命令是用来清除断点或自动显示的表达式的命令，命令格式如下：

(gdb)delete 断点的编号或者表达式

这部分用了它的第一种功能，与 clear 命令很相似。用 delete 命令清楚断点 gdb 不会给出任何提示，除非指定的编号不对应一个断点。



clear 命令

clear 是一条用来清除断点的命令。当进行调试时，如果确定在设置断点的语句处没有必要暂停运行，就可以使用 clear 命令来清除设置的断点。

命令格式如下：

(gdb)clear 要清除的断点所在的行号

clear 命令与 delete 命令不同的是 clear 命令后面要给出清除的断点所在行的行号，而 delete 命令是给出要清除断点的编号。另外，使用 clear 命令清除断点时 gdb 会给出提示，而使用 delete 命令则不会。

4.2.4 使用观察窗

使用观察窗口所实现的功能和在程序中的特定位置设置断点所实现的功能有些相似，但还是不太一样，使用观察窗口时只在表达式为真的时候中断程序的运行。

设置观察窗口，使用 watch 命令。通过这个命令，可以设置任何需要查看的表达式。设置时要注意，由于观察窗口并不是一段特定的代码，因此可以在程序运行的任何时间对它求值，但如果在表达式中的某个变量的取值超过了范围，就不能再对表达式取值了。这一点是与条件断点的设置不同的，因为它只是在代码的固定位置取值。

下面用程序 test4_3.c 来检测观察窗口的功能。

【程序 4_3】

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

```
int main(void){  
    int counter;  
    for(counter=0;counter<30;counter++){  
        if(counter%2==0){  
            printf("Counter: %d\n", counter);  
        }  
    }  
}
```

编译并运行程序。

```
$gcc -ggdb3 -o test4_3 test4_3.c  
$./test4_3
```

输出结果如下：

```
Counter: 0  
Counter: 2  
Counter: 4  
Counter: 6  
Counter: 8  
Counter: 10  
Counter: 12  
Counter: 14  
Counter: 16  
Counter: 18  
Counter: 20  
Counter: 22  
Counter: 24  
Counter: 26  
Counter: 28
```

因为程序中使用判断语句 if 的条件是 `counter%2==0`，所以输出的 counter 值都是偶数。这时装载 gdb，就能设置观察点，然后运行它。

```
$ gdb test4_3  
GUN ...
```

```
...
(gdb)watch counter>15
No symbol "counter" in current context.
```

省略的部分仍然是有关版本等方面的介绍和 gdb 的信息，不用考虑它。后面设置了观察窗口，条件表达式是 `counter>15`，最后得到输出的原因是程序没有执行到 `main()` 函数的缘故，这时变量 `counter` 也没有进入作用域。对它设置断点执行到下列代码。

```
(gdb)break main
Breakpoint 1 at 0x120000428: file test4_3.c, line 3.
(gdb)run
Starting program: /home/jgoerzen/t/test22
```

```
Breadpoint 1, main() at test4_3.c: 3
3    int main(void){
(gdb)s
5    for(counter=0;counter<30;counter++){
(gdb)s
6    if(counter%2==0){
```

现在来设置观察点：

```
(gdb)watch counter>15
Hardware watchpoint 2: counter>15
```

继续运行程序：

```
(gdb)continue
Continuing.
#0  main() at test4_3.c: 6
6    if(counter%2==0){
Counter: 0
Counter: 2
Counter: 4
Counter: 6
Counter: 8
Counter: 10
Counter: 12
```

Counter: 14

Hardware watchpoint 2: counter>15

```
Old value=0
New value=1
0x8048418 in main() at test4_3.c: 5
5      for(counter=0;counter<30;counter++){
```

从上面的输出中可以看出程序的运行被设置的观察表达式终止了。表达式将连续取值，直到它为真，即 `counter` 的值大于 15，这时程序将终止。

这时要继续运行 `test4_2.c`，就要重新装载 `gdb`。再次运行 `test4_2c` 代码后，观察窗口将不起作用。

```
$gdb test4_2
GUN ...
...
The GDB ...
(gdb)break getinput
Breakpoint 1 at 0x80485b9;file test4_2.c, line34
(gdb)run
Starting program: /home/jgoerzen/t/test4_2
+Breakpoint 1, getinput() at test4_2.c: 34
33      printf("Enter a string, or leave blank when done: ");
(gdb)s
35      fgets(input, 79, stdin);
(gdb)s
36      input[strlen(input)-1]=0;
(gdb)s
if(strlen(input)==0)
(gdb)s
39      instruct=malloc(sizeof(datastruct));
(gdb)s
40      instruct->string=strdup(input);
(gdb)s
41      instruct->checksum=0;
(gdb)s
42      for(counter=0;counter<strlen(instruct->string);counter++){
(gdb)watch instruct->checksum>750
```

Hardware watchpoint 2: instruct->checksum>750

上面设置了一个观察点，来看一下运行的结果：

```
(gdb)continue
Continuing.
#0  getinput() at test4_2.c: 42
43      for(counter=0;counter<strlen(instruct->string);counter++){
Watchpoint 2 deleted because the program has left the block in
Which its expression is valid.
0x8048537 in main() at test4_2.c: 19
19      svalues[counter]=getinput();
```

上面设置观察窗口的条件表达式为 `instruct->checksum>750`，此时表达式的相关变量 `instruct->checksum` 超出了取值范围，则观察表达式就不能求值了，`gdb` 将返回一条确认信息。

4.2.5 core dump 分析

在进行程序开发时，常常会由于种种原因导致程序崩溃。这时候可以使用 Linux 的 core dump 功能。在程序崩溃时，Linux 会创建一个 `core` 文件，在程序结束后，利用这个文件中的信息，再使用 `gdb`，就能知道程序在崩溃的时候在做些什么。但 core dump 功能并不是所有的程序员都能使用的，这是因为一些发行版和一些系统的管理员在默认时禁止使用 `core dump`。可以使用下面的命令使自己拥有这种权限。

```
$ulimit -c unlimited
```

这样就能使用 `core` 文件分析了。

下面是一个有错误的程序(`test4_4.c`)，讲述 `core dump` 功能。

【程序 4_4】

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

typedef struct TAG_datastruct{
    char* string;
    int checksum;
}datastruct;
```

```
datastruct* getinput(void);
void printmessage(datastruct* todisp);

int main(void){
    int counter;
    int maxval=0;
    datastruct* svalues[200];

    for(counter=0;counter<200;counter++){
        svalues[counter]=getinput();
        if(!svalues[counter]) break;
        maxval=counter;
    }

    printmessage(svalues[maxval*2]);

    return 0;
}

datastruct* getinput(void){
    char input[80];
    datastruct* instruct;
    int counter;

    printf("Enter a string, or leave blank when done: ");
    fgets(input, 79, stdin);
    input[strlen(input)-1]=0;
    if(strlen(input)==0)
        return NULL;
    instruct=malloc(sizeof(datastruct));
    instruct->string=strdup(input);
    instruct->checksum=0;
    for(counter=0;counter<strlen(instruct->string);counter++){
        instruct->checksum+=instruct->string[counter];
    }
    return instruct;
}
```

```

void printmessage(datastruct* todisp){
    printf("This structure has a checksum of %d. Its string is: \n",
           todisp->checksum);
    puts(todisp->string);
}

```

编译和运行这个程序，注意不要在 gdb 中运行。

```

$gcc -ggdb3 -o test4_4 test4_4.c
$./test4_4
Enter a string, or leave blank when done: Hi!
Enter a string, or leave blank when done: I like Linux.
Enter a string, or leave blank when done: How are you today?
Enter a string, or leave blank when done: Enter
This structure has a checksum of -1541537728. Its string is:
Segmentation fault(core dumped)

```

上面的输出可以看出程序在某个地方出了严重的错误，由于检验不正确，程序崩溃了。为了查找原因，将核心文件调入 gdb。

```

$gdb test4_4 core
GNU ...
...
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/libc.so.6...done.
Reading symbols from /lib/ld-Linux.so.2...done.
#0 0x8048686 in printmessage (todisp=0x0) at test4_4.c: 49
49     printf("This structure has a checksum of %d. Its string is: \n",

```

调试器显示程序是因为发生了段错误而导致崩溃，并且是发生在 printf() 调用中。为了判断程序在崩溃之前运行到哪里，就要用到有关堆栈方面的 bt 命令。



bt (即 backtrace) 命令用来打印 stack frame(栈桢)指针，可以了解当前程序运行过程中的函数之间的调用关系。命令格式如下：

(gdb)bt (要打印的栈桢指针的个数)

```
(gdb)bt  
#0 0x8048686 in printmessage(todisp=0x0) at test4_4.c: 49  
#1 0x804858e in main() at test4_4.c: 24
```

gdb 告诉程序员每个函数被执行的最后一行。编号为 0 的栈桢(#后面跟的就是栈桢编号), 在第 49 行出现了问题, 并被 gdb 高亮度显示。

还知道当函数 printmessage() 被调用的时候, todisp 为 0(前面 0x 表示为十六进制数据), 但是指针 todisp 不应该为 0, 说明在此出现错误。

```
(gdb)print todisp  
$1=(struct TAG_datastruct*)0x0
```

由于 printmessage() 函数被调用时出现了错误, 但问题不出现在这里, 因此需要激活在 main() 中的栈桢 1 来检查调用函数 main()。

```
(gdb)frame 1  
#1 0x804858e in main() at test4_4.c: 24  
24     printmessage(svalues[maxval*2]);
```



frame 命令是用来打印栈桢的, 它使用如下格式。

(gdb)frame 要打印的栈桢的编号

当没有指定要打印的栈桢时, gdb 会打印当前选择的栈桢。

可以使用 info frame 命令来查看当前所选择的栈桢, 它将给出该栈桢的详细信息。

检查 main() 中的变量, 以确认这些变量是合法的。

```
(gdb)print counter  
$2=3  
(gdb)print maxval  
$3=2  
(gdb)print svalue[1]
```

```
$4=(struct TAG_datastruct*)0x8049b00
(gdb)print *svalue[1]
$5={ string=0x8049b10"I like Linux.", checksum=1132}
(gdb)print svalue[maxval*2]
$6=(struct TAG_datastruct*)0x0
```

前面的变量都是合法的，但最后一个出现了明显的错误，现在来看看指针 `svalues` 所指向的内容：

```
(gdb)print svalues[maxval*2]
Cannot access memory at address 0x0.
```

上句输出表示表达式 `svalues[maxval*2]` 已经超出了 `svalues` 指针指向的存储结构。然而这种方法也会有意外的时候，例如当程序完全崩溃之前堆栈如果已经被破坏了，这时就不能从中获得有用的信息。

继续调试程序 `test4_2.c`（“显示数据命令”部分）。

```
./test4_2
Enter a string, or leave blank when done: Hello!
Enter a string, or leave blank when done: I enjoy Linux.
Enter a string, or leave blank when done: God is interesting!
Enter a string, or leave blank when done: Enter
This structure has a checksum of 1260. Its string is:
I enjoy Linux.
./test4_2
Enter a string, or leave blank when done: Enter
Segmentation fault (core dumped)
```

第二次调用程序时发生崩溃。同前面的步骤，先用 `gdb` 来调试。

```
$gdb test4_2 core
GNU ...
...
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/libc.so.6...done.
Reading symbols from /lib/ld-Linux.so.2...done.
#0 0x8048686 in printmessage (todisp=0x0) at test4_2.c: 49
#0 0x8048696 in printmessage(todisp=0x0) at test4_2.c: 49
```

```
49      printf("This structure has a checksum of %d. Its string is: \n",
(gdb)bt
#0 0x8048696 in printmessage(todisp=0x0) at test4_2.c: 49
#1 0x804859e in main() at test4_2.c: 24
(gdb)frame 1
#1 0x804859e in main() at test4_2.c: 24
24      printmessage(svalues[maxval/2]);
(gdb)list
19      svalues[counter]=getinput();
20      if(!svalues[counter]) break;
21      maxval=counter;
22  }
23
24  printmessage(svalues[maxval/2];
25
26  return 0;
27 }
28
```

然后调用函数 printmessage() 的变量的值。

```
(gdb)print maxval
$1=0
(gdb)print svalues[maxval/2]
$2=(struct TAG_datastruct*)0x0
```

如果修改程序中 maxval 的值为 -1，而不是 0，就不会产生上面的错误。这是因为数组的索引值最小为 0，而决不会是 -1。当程序执行后要求输入 string 时，如果没有输入，maxval 的值仍然为 0，这将产生错误导致程序崩溃。

下面写成改正后的代码：

【程序 4_4a】

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

typedef struct TAG_datastruct{
    char* string;
```

```
int checksum;
}datastruct;

datastruct* gerinput(void);
void printmessage(datastruct* todisp);

int main(void){
    int counter;
    int maxval=-1;
    datastruct* svalues[200];

    for(counter=0;counter<200;counter++){
        svalues[counter]=getinput();
        if(!svalues[counter]) break;
        maxval=counter;
    }

    printmessage(svalues[maxval/2]);

    return 0;
}

datastruct* getinput(void){
    char input[80];
    datastruct* instruct;
    int counter;

    printf("Ente a string, or leave blank when done: ");
    fgets(input, 79, stdin);
    input[strlen(input)-1]=0;
    if(strlen(input)==0)
        return NULL;
    instruct=malloc(sizeof(datastruct));
    instruct->string=strdup(input);
    instruct->checksum=0;
    for(counter=0;counter<strlen(instruct->string);counter++){
        instruct->checksum +=instruct->string[counter];
    }
}
```

```
    return instruct;
}

void printmessage(datastruct* todisp){
    printf("This structure has a checksum of %d. Its string is: \n",
           todisp->checksum);
    puts(todisp->string);
}
```

编译后运行一切正常。

4.3 其他调试工具

在 Linux 平台上调试 C 语言程序时，除了使用 `gdb` 之外，还可以使用 `xxgdb`。`xxgdb` 是 X Window 系统的调试工具，实际上，`xxgdb` 是图形界面的 `gdb`，它保留了 `gdb` 所有的特性。程序员可以像在 `Tc` 上调试 C 程序那样，用点击按钮来代替输入命令，它还能显示当前断点设置的位置，而不用输入名字查询。

要完成调试功能，先要初始化 `xxgdb`，指定 `gdb` 中规定的命令行选项和 `xxgdb` 的命令行选项。然后启动 `xxgdb`，读者就会看到一个窗口出现在屏幕上。

4.4 小结与练习

4.4.1 小结

在软件开发中出现错误在所难免，所以程序的调试就成为软件开发中非常重要的一个环节，`gdb` 就是 Linux 系统中 C 程序调试工具中最常用的一种。本章首先介绍 `gdb` 的程序调试步骤，进而深入地阐述了它的功能和一些高级调用。最后一节还简单地介绍了另外一种调试工具。

4.4.2 习题与思考

调试下面的程序，复习有关断点和数据方面的命令：`break`、`disable`、`display` 和 `cont` 命令。

【程序 4_5】

```
#include <stdio.h>
#include <stdlib.h>
#define BIGNUM 5000

void index_to_the_moon(int ary[ ]);

int main(void)
{
    int intary[10];
    int j;

    index_to_the_moon(intary);
    exit(EXIT_SUCCESS);
}

void index_to_the_moon(int ary[ ])
{
    int j;
    for(j=0;j<BIGNUM;++1)
        ary[j]=j;
}
```

第5章

程序自动维护工具 make

简单使用及属性控制

高级使用

库的使用

小结与练习

当开发的程序变得越来越大时，对它的处理将会变得越来越复杂，而且需要花费的时间也会变得越来越多。这时往往把程序划分成多个相对独立的部分。怎样处理好这些部分的封装性，以及它们内部的完整性和之间的依赖及协调关系，成了现在软件工程中非常重要的工作。

也许读者对流水线生产有些了解，流水线的每一个环节都不能出现错误，并且要求每个环节细致精确，各个环节之间也要有机的组合在一起。这样才能成为一道优秀的工序。

这与开发程序非常相似。整个程序系统可能包含很多独立的部分，而各个部分又包含或多或少的命令行。这时，要对各个部分进行编译，然后在和其他经过编译过的部分链接以形成新的可执行文件。当开发的程序非常庞大时，它可能包含成千上百的部分，而且各个部分之间还存在着复杂的相互关系，这时要把它编译链接就是一件非常困难的工作，而且很可能会漏掉一些部分。在多个程序员一起进行工程开发的时候，要考虑协调好程序员间的合作，错误更是不可避免的，这就使手工编译链接力不从心了，也使利用一个自动系统控制和协调程序的整个创建过程成为程序员的当务之急，更是软件工程日益发展所必须要解决的问题。在这个自动系统之中，程序员可以定义创建规则并通过这个规则进行项目的开发。

在 Linux 中，`makefile` 就是这种定义规则的文件。可以使用 GUN `make` 这个 `makefile` 解释器来处理这个文件。GUN `make` 是一种自动生成和维护目标程序的工具，它是一个单独工作的程序，可以调用编译器、连接器、汇编器等，根据程序各个部分的修改情况重新编译并链接目标代码，保证目标代码的最新组成。当在命令行输入 `make` 后，系统会自动检测系统文件和已经定义的规则，并决定采取合适的步骤，完成整个创建过程。

本章将讲述 `make` 的使用，以及如何编写简洁高效的 `makefile`。

5.1 简单使用及属性控制

在开发大中型项目时，常常把整个系统分成多个独立的部分，这些独立的部分之间存在着非常复杂的内部关系，这些关系将最后决定编译和链接源程序的顺序和范围。当程序日益变大时，要程序员自己去处理这些关系，是一件既耗时又费力的事，而且极可能导致错误。GUN `make` 工具则能从指定文件中读取说明模块之间关系的信息，然后根据这些信息来维护和更新目标文件。

GUN `make` 实现上述功能的具体步骤如下：`make` 首先判断其中的哪些文件过时了，也就是说当一个文件生成之后，用来生成该文件的源程序或者它所依赖的部分被修改过了，但是它自己却没有被修改。然后根据指定文件中的有关信息对那些过时的文件进行更新操作。



判断文件是否过时的机制

Linux 系统会为每个文件记录最后的修改时间，`make` 将根据这个时间即新文件的生成时

间来判断文件是否已经过时。

GUN make 工具和手工的操作比起来，具有很多的优点。例如：它只需要更新那些需要更新的文件，而不用去修改那些未过时的文件；其次 make 去检查一个文件是否需要更新时，它会先去检查这个文件所依赖的所有文件，再对该文件进行更新，这样就避免当它所依赖的文件出错时，它还需要重新更新；make 不会漏掉任何一个需要更新的文件。具备这些优点的 make 工具，使用它来自动维护和创建大中型系统时，既省时间，又可以保证准确性。



文件之间的依赖关系

从上面的 make 和手工操作的比较中可以看出，make 去检查一个文件是否需要更新时，会先去检查那个文件所依赖的所有文件，然后再对该文件进行更新。所以文件之间的依赖关系就成为了使用 GUN make 进行维护目标的根本依据。

所以使用 GUN make 工具实现自动维护功能时，首先就要清楚这些文件之间的依赖关系。当系统很大时，这些关系将变得非常复杂，往往会造成一个关系网，对各个文件起到牵制作用。

举个生活中最简单的例子：

喝茶时，有几件事必须要做：烧开水、准备茶具、泡茶。这三件事是相互独立的事情，一件事情的改变一般不会影响到另一件事，但是它们却是最终目的“喝茶”的必要条件，任何一件事情出了差错，都会影响最后目标的实现。这只是一个最简单的例子，现实生活中的事情要比这个复杂得多。

类似地，开发一个大的程序时也会由一些小的程序部分组成，且相互依赖。现在考虑一个程序(test5_1)，它的实现依赖于三个目标文件 main.o、proc1.o 和 proc2.o，每个文件又依赖于多个源文件。假定 main.o 和 proc1.o 依赖于某个源文件 myprog.h；而 proc1.o 和 proc2.o 则依赖于 mylib.h 源文件；它们又分别都依赖于源程序 main.c、proc1.c 和 proc2.c。

在编译时可以使用下面命令生成 test5_1。

```
gcc main.o proc1.o proc2.o test5_1
```

命令把三个目标文件链接成为可执行文件 test5_1。

而三个目标文件又是由以下命令生成的。

```
gcc -c main.c myprog.h
```

```
gcc -c myprog.h proc1.c myproc.h
```

```
gcc -c myproc.h proc2.c
```

5.1.1 make 的简单使用

前面提到过，可以根据程序之间的依赖关系使用 `make` 自动维护目标文件。然而，程序员必须提供这些依赖关系，并建立一个文件存放这些依赖关系，然后才能调用 `make`。这时，系统就会从这个文件中读取有关的信息，来进行维护工作。这个文件就是 `makefile`。

1. 简单 make 文件的书写

`makefile` 文件的作用就像软件在一个系统中起的作用一样，没有软件支持的系统将会一事无成。所以编写一个简练和高效的 `makefile` 文件在自动维护程序这个工作中所起的作用是非常重要的。这里，先介绍怎样书写简单的 `makefile` 文件。

首先要知道 `makefile` 文件的形式。`makefile` 文件的基本组成单位是“规则”，而每一条规则又说明了一个目标，它除了描述该目标所依赖的文件外，还要指明生成和更新目标文件所需的命令。

下面是规则的格式。

目标 [属性...]

分隔符 [依赖文件] [;命令菜单]

{<制表符> 命令菜单}

上述命令中的[]符号表示其中的内容是可以选择的，而{ }符号表示其中的内容可以多次选择。



规则格式各部分解释

目标：目标文件列表。

属性：目标文件列表的属性，可以为空。

分隔符：作用是用来分隔目标文件和依赖文件。可以使用不同的分隔符，但是不同的分隔符会影响到规则的性质，通常使用的是冒号。

依赖文件：目标文件所依赖的文件列表。

命令菜单：重新生成目标文件的命令，可以为空或者是多条。

`make` 根据符号 `tab` 来辨认命令行，所以一定不要在非命令行的行首加上 `tab`，否则将得到错误消息。但可以在每一行的其他任何位置使用 `tab` 而这一行可以不是命令行。

当命令行之间插入多个空行时，这些空行也要使用 `tab` 符号来开头，且空行的数目不受

限制。

Tip

使用 tab 时的易犯错误

`makefile` 对书写格式的要求非常严格，格式不对很容易发生错误。例如：其中一条很重要的关于 `tab` 使用的规则，就是除了第一个命令外，每一个命令行都要以一个制表符(`tab`)开始，初学者在此经常犯错误，常常在命令行的开头省略制表符 `tab`，这样就会出现错误，而且 `make` 也无法提供有关的信息。

制表符 `tab` 在文本显示上的效果相当于四个空格，但即使输入的是四个空格，`make` 还是无法辨认出这就是 `tab`，还是会导致错误的发生，而得不到有用的信息。

有时为了检查描述文件中的 `tab` 字元，可以使用如下命令。

```
$cat -v -t -e makefile
```

其中的符号 `v` 和 `t` 将使 `makefile` 文件中的每一个 `tab` 字元以 `\t` 的方式打印出来，而符号 `e` 将使每一行的末尾以\$的样子打印出来。

下面是一个简单的 `makefile` 文件，使用的就是在前面假设的那个例子。后面的讲解将具体介绍一些其他的规则。

应用程序 `test5_1`，它依赖三个目标文件 `main.o`、`proc1.o` 和 `proc2.o`；`main.o` 文件依赖源文件 `main.c` 和 `myprog.h`，`proc1.o` 文件依赖文件 `myprog.h`、`proc1.c` 和 `myproc.h`，文件 `proc2.o` 依赖文件 `myproc.h` 和文件 `proc2.c`。

`makefile` 文件使用如下写法。

```
test5_1:main.o proc1.o proc2.o
        gcc main.o proc1.o proc2.o -o test5_1
main.o:main.c myprog.h
        gcc -c main.c
proc1.o:myprog.h proc1.c myproc.h
        gcc -c proc1.c
proc2.o:myproc.h proc2.c
        gcc -c proc2.c
```

这就是一个最简单的 `makefile` 例子，读者可能会觉得它太简单了，但其中还是有不少特殊的规定。

- 第一条命令不需要另起一行用制表符 `tab` 开头，可以直接跟在依赖文件列表的后面，用分号隔开。其他的命令行仍要每个另起一行。

例如上面所写的 `makefile` 例子。

```
main.o:main.c myprog.h  
gcc -c main.c
```

上面命令行还可以写成如下形式。

```
mian.o:main.c myprog.h ;gcc -c main.c
```

- 当一行过长，且已经达到了文字编辑器的右边界，还是不够时，可以在右边界之前放入一个反斜杠(\) 符号。注意反斜杠和新的一行之间不要有空白，则反斜杠所连接的所有行将当作一行来处理。
- 以符号#表示注释行的开始，以换行符作为结束标志，此时 make 将会忽略掉以#符号开头直到该行结尾之间的字元。当其他的地方要用到“#”符号时，要用双引号来引用。
- Makefile 中涉及的文件名允许使用通配符，例如：*和?等。
- Make 将使用 `makefile` 或者是 `Makefile` 作为默认的文件名，具体选用那个将因版本的不同而异。`makefile` 文件只是普通的文本文件，可以使用任何一种文本编辑器来创建和修改它，名字由程序员来命名，缺省时，它的名字为 `makefile`。
- 在 make 中，文件之间的依赖性表现在冒号左边的项必须和右边的项具有相同的更新时间，或者是最近的更新时间。

下面试着使用 make 来维护一个简单的例子(`myprogram`)，它由三个 C 源程序和一个头文件组成。三个 C 源程序是 `compute.c`、`init.c` 和 `io.c`，头文件为 `myprogram.h`。它们的代码如下：

【程序 5_1】

```
/*compute.c源程序的代码*/
```

```
extern int someglobal;
```

```
int computer(void){  
    return 5*someglobal;  
}
```

```
/*源程序init.c的代码*/
```

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

```

int someglobal=11;

int main(void){
    foo();
    return 0;
}

/*源文件io.c的代码*/

#include <stdio.h>
#include "myprogram.h"

int foo(void){
    printf("The value is:%d.\n",computer());
    return 1;
}

/*头文件myprogram.h的代码*/

int computer(void);
int foo(void);

```

下面使用一个简单的 makefile 来对程序进行维护，大家注意 makefile 是如何处理这些代码的。

下面是 makefile 的代码(这是没经过修改的文件):

```

# Lines starting with the pound sign are comments.
#
# "all" is the default target.Simply make it point to
# myprogram.

all:myprogram

# Define the components of the program, and how to
# link them together.
# These components are defined as dependencies;that is,
# they must be made up-to-date before the.

```

```
myprogram:io.o init.o compute.o  
gcc -o myprogram io.o init.o compute.o  
  
# Define the dependencies and compile information for the three C source  
# code files.  
  
compute.o:compute.c  
gcc -Wall -c -o compute.o compute.c  
  
init.o:init.c myprogram.h  
gcc -Wall -c -o init.o init.c  
  
io.o:io.c myprogram.h  
gcc -Wall -c -o io.o io.c
```

此时可以在提示符下输入一个命令来编译全部程序了。make 程序首先查找冒号左边的文件，即被称为目标的文件。本例中，目标文件名为 all，它被设置成取决于 myprogram。all 并不是已经存在的文件，所以每次在 make 被调用时必须先检查 myprogram。

目标 myprogram 和三个目标文件 io.o、init.o 和 compute.o 相关联，所以目标文件必须在运行命令来创建 myprogram 可执行文件之前被更新。如果其中任何一个文件比最终的可执行文件新的话，那么最终的可执行文件将要被重新创建；否则，就没有必要执行这一步了。

对上面的三个目标文件，每个都有一个入口，而每个入口都指明了创建过程中与 C 源文件的依赖性。所以，如果一个特定的 C 源文件被更新了，则目标文件也必须被重新生成，对头文件也要做类似处理。

具体 make 对文件的处理将在第二部分中列出。

2. make 命令的使用

当 makefile 文件建立好了之后，就可以调用 make 命令来生成和维护目标文件了，命令格式如下：

```
make [option] [macrodef] [target]
```

符号[]之间的内容是可以选择的。各参数含义如下：

选项 option：指定 make 工作的行为；

宏定义 macrodef：指定执行 makefile 时的宏值；

目标 target：需要更新的文件列表。

这些参数是可以选择的，各个参数间使用空格号隔开。更为具体的用法将在后面详细介绍。

上一部分讲到特殊规定的最后一点时，说过 make 将使用 `makefile` 或是 `Makefile` 作为默认的文件名。当调用 `make` 命令时，可以使用缺省的文件名 `makefile`。但为了不产生混淆，最好使用一些易于辨认和经常使用的名字来命名 `makefile` 文件。例如上面的例子中，可以使用 `test5_1.mk` 来作为 `makefile` 的文件名。

用以下命令来调用 `make`:

```
make -f test5_1.mk
```

系统会在当前的目录中查找名为 `test5_1.mk` 的文件，然后将它作为 `makefile` 文件来处理。

文件 `test5_1.mk` 找到之后，`make` 就会从 `test5_1.mk` 的第一条规则指定的目标开始进行维护，后面的命令不会立即被执行。这时要检查三个依赖文件 `main.o`、`proc1.o` 和 `proc2.o` 是否过时，从而判断文件 `test5_1` 是否过时。`make` 会按规则中的依赖文件列表从左到右进行逐个的检查，检查中对非源文件的文件，将会作为新的目标继续检查下去，直到所有的依赖文件都是源文件时为止，然后再沿原路径返回。以上面的 `test5_1` 为例：`make` 先检查 `main.o`，然后把它作为目标文件，检查它的两个依赖文件 `main.o` 和 `myprog.h` 是否比 `main.o` 更新。如果 `main.o` 文件的建立时间比它所有的依赖文件 `main.c` 和 `myprog.h` 还要晚，`make` 就会决定不再重新生成 `main.o`，否则启动相应的命令菜单更新文件。然后，使用相同的方法依次对文件 `proc1.o` 和 `proc2.o` 进行检查。当这三个文件都检查完毕后，再对 `test5_1` 进行检查，以确定它是否需要更新。这样，就完成了一次维护工作。

当一次检查完毕后，如果发现所有的文件都不需要更新，`make` 就不会发出任何指令，只是显示如下语句：

```
test5_1 is up to date.
```

然后显示命令提示符。

如果检查发现有更新行动，则 `make` 将会显示所执行的命令。假设文件 `proc1.c` 进行了修改，则屏幕上就会打印出下面两条命令。

```
gcc -c proc1.c
gcc main.o proc1.o proc2.o -o test5_1
```

从上面的过程可以看到，当使用 `make` 进行自动维护程序的工作时，它将按照先目标，后父节点，如在同一层上将按从左到右的顺序进行检查和更新。由于上述原因，一个 `makefile` 文件应该包含顺序正确的指令链接。

所以，只要要求 `make` 去维护链接中的最后那个目标文件，`make` 就会根据链接的信息自己回溯进行追踪，从而发现哪些指令必须被执行。`make` 会依次在链接中查找前进，执行

每一个更新目标所必需的指令，直到目标最后建立完成。在上面所述过程中，`make` 实际上使用了一种反向链接法(backward-chaining)。在人工智能的领域中，这是一种探索问题答案的常用方法。

使用 GUN `make` 进行程序自动维护，默认方式是维护 `makefile` 的第一个目标，也就是第一条规则指定的目标文件。这就决定了书写 `makefile` 文件时，最好把最后要生成的那个目标文件放在最前面，例如上面例子中的文件 `test5_1`。当需要指定生成某个特定的文件时，要另外指出。

继续前一部分所举的例子(`myprogram`)。

```
$ make
gcc -Wall -c -o io.o io.c
gcc -Wall -c -o init.o init.c
gcc -Wall -c -o compute.o compute.c
gcc -o myprogram io.o init.o compute.o
```

在命令提示符下输入 `make` 命令后，编译命令将以正确的顺序被执行。首先，C 源文件被编译成目标文件，接着开始链接目标文件，以形成最终的可执行文件。之所以采用这样的顺序是因为依赖性的作用，最终的可执行文件要求所有的目标文件都被更新过，为了做到这一点，`make` 就必须把 C 源程序编译成目标文件。

现在形成的目标文件和最终的可执行文件都是最新的，如果此时再使用 `make`，将会得到如下输出：

```
$ make
make:Nothing to be done for 'all'.
```

说明没有再编译的必要了。

所以，当运行已经编译过的代码时，`make` 会检测是否所有的文件都是最新的，如果是最新的，`make` 将不会做任何事情。

如果情况和上面不同，其中的一个文件进行了修改，或只是把这个文件装入编辑器，重新保存一下，又或者使用 `touch` 命令，把文件的修改时间改为当前的时间，来看看会发生什么情况。

```
$ touch io.c
$ make
gcc -Wall -c -o io.o io.c
gcc -o myprogram io.o init.o compute.o
```



touch 命令

touch 命令将参数的修改时间设置为执行 touch 命令的时间，上面 touch 的参数是 io.c。由于上述原因，所以现在 io.c 必然是过时的。

touch 命令是一个测试 make 的常用方法。

在上述命令中，make 还是和前面一样要检查文件之间的依赖性，当检查到 io.o 文件时，因为前面对 io.c 文件进行了三种情况中的任何一种更新，所以必须对文件 io.o 进行重新编译。重新编译后程序就必须被重新链接。make 会自动进行条件检测，然后采取适当的行动。

程序自动维护工具 make 在进行上述步骤时，并不会编译那些没有改动过的文件，可以大大的节省时间。这也是 make 作为程序自动维护工具的一个显著的特点。

3. 编写巧妙的 makefile 文件

从前面的 makefile 中，可以看到出现了很多循环的过程，它们重复做着两件事情：gcc 命令行选项和为每个 C 源文件设定的依赖性。

我们可以在 makefile 中使用变量来简化命令行选项，这样不但能够减少为创建规则所需要的输入，还能在修改了文件中的前两行时，就能让程序员改变对这个文件的规则，便于处理大的文件。

可以使用类似于 bash 中的操作方法在 makefile 中设置变量，利用等号把等号左边的变量名和右边的变量值分隔开来。但是两者在语法上还是有些不同。在 make 中，使用 \$(VARIABLE) 来获得括号中的 VARIABLE 的值。

下面使用变量来修改前面的 makefile 例子。

```
# Lines starting with the pound sign are comments.
#
CC=gcc
CFLAGS=-Wall
COMPILE=$(CC)$(CFLAGS)-c

# "all" is the default target. Simply make it point to myprogram.
all:myprogram

# Define the components of the program, and how to link them together.
# These components are defined as dependencies; that is, they must be
# made up-to-date before the code is linked.

Myprogram:io.o init.o compute.o
```

```
$(CC) -o myprogram io.o init.o compute.o

# Define the dependencies and compile information for the three c source
# code files.

compute.o:compute.c
$(COMPILE) -o compute.o compute.c

init.o:init.c myprogram.h
$(COMPILE)-o init.o init.c

io.o:io.c myprogram.h
$(COMPILE) -o io.c io.c
```

从上面的例子当中，可以看到在每行中删除了诸如-Wall 的选项，使用变量 CC 代替 gcc，使用变量 CFLAGS 代替-Wall 选项，使用 COMPILE 变量代替(gcc -Wall -c)。所以，以后如果需要添加选项的时候，只需要修改 CFLAGS 行，在这行中添加需要的选项。

变量 COMPILE 的内容取决于另外两个变量 CC 和 CFLAGS 的值。可以使用一个变量的值来给另外一个或是多个变量赋值。

其实在 makefile 中，程序员还有很多的工作要做，例如：可以删除每个 c 源程序文件建立的单独列表；也可以针对 C 源文件，指定一个类属规则。

来看下面一个修改过的 makefile(有错误)。

```
# Lines starting with the pound sign are comments.
#
CC=gcc
CFLAGS=-Wall
COMPILE=$(CC)$ (CFLAGS)-c

# "all" is the default target.Simply make it point to myprogram.

all:myprogram

# Define the components of the program, and how to link them together.
# These components are defined as dependencies;that is, they must be
# made up-to-date before the code is linked.
```

```

Myprogram:io.o init.o compute.o
$(CC) -o myprogram io.o init.o compute.o

# Define the dependencies and compile information for the three c source
# code files.

%.o:%.c
$(COMPILE) -o $@ $<

```

可以发现上面的 `makefile` 和前面那个 `makefile` 只是在最后稍有差异。`%.o:%.c` 规则指出：任何以`.o` 结尾的文件取决于有相同文件名但是以`.c` 结尾的文件。这个规则使用了两个特殊的字符`$@`和`$<`操作符。

`$@`操作符为目标名所代替，而`$<`操作符则表示要编译文件的文件名。这些信息将被传递给 `gcc`。

Tip

将在第二小节中介绍规则的使用，这里不做详细的讲解。

已经知道上面的 `makefile` 有错误即在目标文件依赖性的列表中，C 头文件和源文件在一起，而 `makefile` 忽略了这个列表，导致头文件的依赖性不会被 `make` 关联进去。

可以如下改动上面的 `makefile`。

```

# Lines starting with the pound sign are comments.
#
CC=gcc
CFLAGS=-Wall
COMPILE=$(CC)$(CFLAGS)-c

# "all" is the default target. Simply make it point to myprogram.

all:myprogram

```

```

# Define the components of the program, and how to link them together.
# These components are defined as cependencies;that is, they must be
# made up-to-date before the code is linked.

```

```

Myprogram:io.o init.o compute.o
$(CC) -o myprogram io.o init.o compute.o

```

```
# Define the dependencies and compile information for the three c source
# code files.

io.o init.o:muprogram.h

# Specify that all .o files depend on .c files, and indicate how
# the .c files are converted (compiled)to the .o files.

%.o:%.c
$(COMPILE) -o $@ $<
```

请注意倒数第七行代码提供的信息。首先，多个目标文件被列在冒号的左边；其次该行定义了自己的依赖性，但是没有定义操作规则。它们都是合法的，make 自己会使用在%.o 那行定义的标准操作规则来决定这些特殊文件是否需要被重新编译。

下面进一步修改 makefile。

```
# Lines starting with the pound sign are comments.
#
# These things are options that you might need
# to tweak.

OBJS=io.o init.o compute.o
EXECUTABLE=myprogram

CC=gcc
CFLAGS=-Wall
COMPILE=$(CC)$(CFLAGS)-c

# "all" is the default target.Simply make it point to myprogram.

all:$(EXECUTABLE)

# Define the components of the program, and how to link them together.
# These components are defined as dependencies;that is, they must be
# made up-to-date before the code is linked.
```

```

$(EXECUTABLE):$(OBJS)
$(CC) -o $(EXECUTABLE) $(OBJS)

# Add any special rules here.

io.o init.o:muprogram.h

# Specify that all .o files depend on .c files, and indicate how
# the .c files are converted (compiled) to the .o files.

%.o:%.c
$(COMPILE) -o $@ $<

clean:
    -rm $(OBJS) $(EXECUTABLE) * ~

```

上面的 `makefile` 从声明两个变量开始，它们包含了程序的组件和程序的名字。使用这两项后，可以非常容易地在其他项目中重新使用该文件。

后面的三个变量同前面一样，在它们之后是 `all` 目标。它和前面在形式上有些不同，可执行文件名用变量代替了直接的字母输入。但是作用和前面例子中的意义相同。

紧跟着是 `$(EXECUTABLE):$(OBJS) ...`，这段代码规定了程序最后的连接规则。可执行文件的名字来自变量，它在冒号的左边。`$(OBJS)` 定义了依赖性，它的作用对象是文件开头定义的目标文件。

然后是头文件的依赖性列表和类属编译规则（也和前面的例子一样。）

最后以 `clean` 目标结束。

Tip

`clean` 的使用

`clean` 的含义是为程序员提供一种便捷的方法，删除编译过的文件、编译备份文件和其他过程中生成的文件，并且返回原目录。这个目标没有任何的依赖性，也不依靠其他的目标而存在。

它不能由 `make` 自动执行。

5.1.2 make 属性的控制

前面介绍了 `make` 的基本使用和怎样编写简单的 `makefile`。这部分将要讲述一些高级的技巧，以便更好地控制 `make` 的行为。

1. makefile 中属性的控制

(1) 分隔符

在前面的例子中，大家应该接触过 makefile 规则中的几个基本语法单位，其中就有分隔符。分隔符是分隔目标文件和依赖文件的符号，它一般为冒号。但是 Linux 还是允许使用其他符号作为分隔符，这些分隔符的作用不是单一的，它除了起简单的分隔作用，还可以控制规则的属性。

冒号，双冒号(::)是最常用的分隔符，它用于一个目标有多个规则的情形。一般来说，一个文件只能在目标的位置上出现一次，只有当使用了双引号或者其他特殊分隔符，才允许多次作为目标文件。

另外几种分隔符如下：

- :^ 将指定依赖文件和目标文件已有的依赖文件合起来，作为新的依赖文件列表；
- :- 清除目标文件原有的依赖文件，将新的依赖文件作为目标的依赖文件列表；
- :! 对每个更新过的依赖文件都执行一次命令菜单；
- :! 只在内部规则中使用。

来看一下几个实例。

```
a.o::a1.c b.h  
#第一条命令  
a.o::a2.c b.h  
#第二条命令
```

按照上面的规则，如果 make 发现 a1.c 的修改时间晚于 a.o，使用第一条命令重建 a.o；如果是 a2.c 导致 a.o 过时，执行第二条命令；但如果是 b.h 导致 a.o 过时，那么两个命令都要执行。

```
file1.o:- file1.c
```

因为:- 要先清除原来的依赖文件的全部信息，所以 file1.o 只有一个依赖文件 file1.c。

```
test.o:stdio.h  
test.o:^test.c
```

最终 test.o 有两个依赖文件：test.c 和 stdio.h。

(2) 命令行属性

从前面的实例中可以看到，命令菜单中包括一个或多个命令行，每个命令行都以一个制表符开始。可以在制表符后面加上减号、加号或是@选项，然后再写命令的正文，这样就可以控制每个命令行的属性。如果不加任何限制，系统将会使用缺省的属性，执行每条命令。

然后打印该命令行，如果在运行命令时出现错误，就会退出 make。

下面介绍命令行选项的功能：

- 忽略所在命令行的非零返回，继续执行，如果执行时产生了错误，make 就会停止执行
- + 当目标文件过时，而命令菜单需要执行，它将使 make 始终执行所在的命令行，即使是使用了-n, -q 或-t 选项
- @ 前面的例子中，make 会自动打印所有执行的命令行，但是如果选择了这个选项，则不会在标准输出上显示

下面给出几个例子来熟悉选项的功能。

第一个例子介绍@选项的使用情况。假设有一个文件名为 main.mk 的 makefile。

```
main.o:main.c
gcc -c main.c
```

```
touch main.c
make -f main.mk
```

touch 命令的功能在前面已经介绍过，它使 main.o 过时。运行 make 时会执行命令行 gcc -c main.c，屏幕上将会显示如下命令。

```
gcc -c main.c
```

现在来修改 makefile 的命令菜单的第二行。

```
@gcc -c main.c
```

其他的保持不变，执行后屏幕上不会有显示。

第二个例子是关于“-”选项的使用。

书写如下的规则。

```
test.o:test.c
mv test.o /backup
gcc -c test.c
```

规则限定当发现 file.o 过时时，就会将旧的文件在/backup 目录下做一个备份，然后生成新的目标文件。如果运行 makefile 时没有 file.o 文件，也就是说这是第一次生成这个文件，则 mv 命令就会出现错误。如果希望忽略这个错误，以便继续以后的步骤，可以使用如下命令。

```
-mv file.o /backup
```

因为减号可以忽略错误，继续运行，所以即使 mv 出现了问题，make 也不会停止运行。

(3) 目标文件属性

由于目标文件和规则是一一对应的关系，所以目标文件的属性就是规则的属性。前面讲的都是对每条命令进行属性控制，这部分将要介绍 Linux 提供的对每条规则进行的属性控制的选项，即目标文件的属性。

一般使用两种格式来规定目标文件的属性。

第一种格式是指定多个文件的属性。

属性 1 属性 2 ... :目标文件 1 目标文件 2 ...

第二种格式是指定单个目标文件的属性。

目标文件 属性 1 属性 2 ...:

依赖文件列表

命令菜单

例如下面的属性(.SILENT)设置是等价的。

```
.SILENT:file1.o file2.o
```

和

```
file1.o .SILENT:file1.c
```

#命令行

```
file2.o .SILENT:file2.c
```

#命令行

下面介绍常用的几种属性。

● .IGNORE

与命令行属性中的减号选项类似。例如 make 在更新所在目标文件时如果遇到错误，系统将会忽略此错误，继续检查并维护其他的目标文件。如果没有这个选项，系统将按缺省情况处理，就是退出 make.。与减号不同的是，这样做的结果可能不正确。

● .SILENT

与命令行属性中的@选项类似。执行所在目标文件的命令菜单时所有的命令不在屏幕上显示。

● .PRECIOUS

设置这个选项可以保留中间的目标文件。如果最终的目标文件的依赖文件也是目标文件时，它们就成为中间目标文件，而且只是在调用 make 时临时存在。缺省情况下，退出 make 系统会自动删除这些临时文件。如果使用了这个选项，在退出 make 时，系统会保留这些临时文件。

2. 伪目标选项

在 make 中，目标分为实目标和伪目标两类。

实目标就是那些真正要生成并以文件形式存放在磁盘上的目标。

伪目标则使 make 程序在执行一些与创建和维护目标文件没有直接联系的辅助性操作，例如打印信息等。

可以任给一个名字来说明伪目标，也可以使用 Linux 提供的内部伪目标。

(1) 自己定义伪目标

自己定义伪目标时可以任给一个名字。

lp:

pr *.c||p

上命令将所有的 C 语言源文件以分页的方式在打印机上打印。只要系统中没有 lp 文件，命令菜单中没有创建 lp 文件，这个目标总认为是过时的，此命令一直被执行。

(2) 使用内部伪目标

● .ERROR

这个伪目标的功能是在 make 执行过程中遇到错误时，用它来显示错误信息，并完成一些简单的错误处理。

它使用如下格式。

.ERROR:依赖文件列表

命令菜单

● .INCLUDE

这个伪目标命令的功能是在 make 处理其他的 makefile 文件，还在文件中的这个 INCLUDE 语句所在的位置插入这些 makefile 文件的内容。

下面是这个伪目标的格式。

.INCLUDE:file1 file2 ...

例如：

.INCLUDE:make.mk

当 make 执行到这里，会去处理 make.mk 文件中的内容。在处理过程中，.INCLUDE 将文件的内容拷贝到这个地方，类似于一次宏展开。

如果要包含的文件和当前的 makefile 不在一个目录下时，可以使用两种方法来解决。

第一种方法是在 .INCLUDE 后面跟上路径名。

.INCLUDE:dir1 dir2 ...

另一种方法是直接在要包含的文件中附带路径。

.INCLUDE:<dir/file>

● .IMPORT

这个伪目标在环境中实现查找宏定义的功能。使用时，将要载入的宏名作为依赖文件，不需要命令菜单。

.IMPORT:宏 1 宏 2 ...

还可以使用 make 本身定义的一个依赖文件.EVERYTHING 使用环境中的所有的宏定义，而不需逐一列出宏名。

.IMPORT:.EVERYTHIN

● .EXPORT

它和 .IMPORT 正好相反，是将 makefile 中定义的宏输出给环境。下面是它的书写格式。

.EXPORT:宏 1 宏 2 ...

Now Romon

因为它的功能是将指定的宏及其当前的值输出给环境，因此在退出 make 之后仍然能访问这些宏，且输出的是当前的宏值。这里很容易出现错误，如下面 makefile 中的语句：

```
mickey=zhan
.EXPORT:mickey
mickey=zhang
```

make 结束运行后，如果在环境中访问 mickey 得到的是 zhan 而不是 zhang 。

● .SETDIR

它将当前的目录变成指定的目录，也就是把运行 `make` 时的工作目录变成指定目录。书写格式如下：

```
.SETDIR=path
```

设原来是在根目录下运行 `make`，工作目录就是根目录，执行上面的命令后，工作目录就变成为 `path`，所有的文件都要在这个目录下查找。

3. make 的命令参数

(1) make 维护目标的指定

在默认情况下，使用 `make` 维护的是 `makefile` 中的第一个目标文件，而且也在书写 `makefile` 时，把主要的维护目标放在 `makefile` 文件的最前面。但也许有时想生成或者更新某个特定的中间目标文件，而不关心主要的目标文件，也许是希望维护多个目标文件。

这时，就可以使用 `make` 命令的 `target` 参数来控制 `make` 的执行，方法是在 `make` 后面加上指定目标文件的名字即可。

例如：

```
make -f test5_1.mk proc1.o proc2.o
```

上面的命令只是维护 `proc1.o` 和 `proc2.o` 两个目标文件。

当希望用一个 `makefile` 文件维护多个文件时，可以在 `makefile` 的最前面加上一条伪规则。

```
all: 目标文件 1 目标文件 2 ...
```

它将冒号右边的所有目标文件作为 `all` 的依赖文件。当运行 `make` 时，从第一个目标文件出发就能维护所有指定的目标文件。这条规则不需要命令菜单，`all` 永远都不会创建，也就不会产生负面影响。

(2) make 属性

常用 `make` 命令选项如下。

表 5-1

常用 `make` 命令选项表

命令选项	命令选项功能简介
-c dir	<code>make</code> 开始运行之后的工作目录为指定目录
-e	不允许在 <code>makefile</code> 中对环境的宏赋新值
-f filename	使用指定的文件作为 <code>makefile</code>

续表

命令选项	命令选项功能简介
-i	忽略运行 <code>makefile</code> 命令时产生的错误，但不退出 <code>make</code>
-k	执行命令出错时放弃当前的目标文件，转而去维护其他目标
-n	按实际运行的执行顺序显示命令，但是不真正执行
-p	显示 <code>makefile</code> 中的所有宏定义和描述内部规则的规则
-r	忽略内部规则
-s	执行但不显示所执行的命令，常用于检查 <code>makefile</code> 的正确性
-S	执行 <code>makefile</code> 命令菜单时出错时退出 <code>make</code>
-t	修改每个目标文件的创建日期，但不真正重新创建文件
-v	打印 <code>make</code> 的版本号
-x	将所有的宏定义输出到 <code>shell</code> 环境

从上面的讲解中，可以看到要控制 `make` 执行时的属性，可以通过三种途径：命令行属性的控制，目标文件属性的控制和 `make` 选项的控制。三种控制方式的选项功能有很多的类似之处，它们的差别主要在作用域的不同：命令行属性只是作用在当前的一个命令行；目标文件属性作用于该目标文件，包括命令菜单中的所有命令；`make` 属性则作用于整个 `make` 执行过程。它们的作用域的趋势是从小到大，从具体到一般。它们的区别还有它们所适用的情况不同。

那么当用三种方法定义属性产生矛盾的时候，则采用一般服从特殊的规则。也就是说先服从命令行属性，没有指定命令行属性时就考虑目标文件的属性，没有指定目标文件的属性时再考虑 `make` 选项，如果三者都没有指定，就使用缺省值。

5.2 高 级 使用

在前一节中，主要介绍 `make` 的简单使用，在这节中，将介绍它的高级使用方法。其中包含宏、内部规则以及 `make` 递归。

5.2.1 宏的使用

在前面已接触了 `makefile` 文件的书写，还介绍了 `make` 的基本用法和属性的控制。但是那样写出的 `makefile` 文件会很冗长，而且不易维护和修改。可以使用宏和内部规则来大大简化 `makefile` 的书写。首先介绍宏的使用。

在 windows 操作系统中进行 C 语言编程时就使用过宏，其实汇编语言中也有宏，甚至还有 word 中的宏病毒。其实，宏就是代表某个字符串的短名，但是它和字符串有很大的不同。

如果文件或者程序中遇到字符串常量或变量时，就会将它当成一种特定类型的常量或变量。然而遇到宏时，则在宏名出现的地方用它代表的字符串来进行替换，将这个字符串当作文件有机的组成部分。这样，当定义了宏以后，如需要修改宏，只需在宏的定义中进行修改，而不需在程序中每次出现宏名的地方都进行修改。

1. 宏的定义和引用方法

要在 make 中使用宏，首先要定义宏，然后在 makefile 中引用。之所以是这种顺序是因为 make 处理 makefile 时类似于解释执行，如果没有预编译的过程，就不能识别那些没有定义的宏名。

宏定义使用如下格式。

宏名 赋值符号 宏的值

其中宏名是由程序员自己指定，它可以是字母数字及下划线(_)的任意组合。但是不能以数字开头，一般习惯使用大写字母，而且为了便于使用和维护，最好使名字具有一定的实际意义。

赋值符号则可以取下面的三种：

- = 直接将后面的字符串赋值给宏
- := 将后面的字符串常量赋值给前面的宏
- += 使宏原来的值加上一个空格，再加上后面的字符串，作为新的宏值

Tip

赋值符号前面除了空格之外，不能有制表符或其他任何分隔符号，否则 make 将会把它作为宏名的一部分。赋值符号后面的字符串可以含有任何的字符，也包括空格，但是最前面的空格和制表符将会被去掉。

宏的引用格式有两种。

\$(宏名)

或

\${宏名}

如果宏名只是单个字符，可以省略括号。

宏还允许嵌套使用，在处理时只要按顺序一次展开就行。例如：

MYFILE1=mylib1h

```
MYFILE2=mylib2.h  
INDEX=2
```

当采用如下形式引用宏时将得到结果 `mylib2.h`。

```
$(MYFILE$(INDEX))
```

可以在三种地方对宏进行定义：第一是在 `makefile` 中，第二是在 `makefile` 命令行中，第三就是在载入环境中。定义宏时，可以直接书写上面的定义式，也可以使用伪目标`.INCLUDE`从其他的文件中获得。如果是在 `make` 命令行中定义宏时，应放在“属性”之后，“目标文件”之前。

前面已经知道，在命令行中是使用空格作为不同参数之间的分隔符，因此当宏的值中含有空格时，要用引号将整个宏括起来。

如果使用的是 `shell` 环境的宏，就不需要重新定义，但是要先用伪目标`.IMPORT` 载入。

重新考虑本章最早的那个 `makefile` 文件(`test5_1`)，在其中使用宏定义，则可以把它改写成如下形式。

```
HEADFILE1=myprog.h  
HEADFILE2=myproc.h  
test5_1:main.o proc1.o proc2.o  
        gcc main.o proc1.o proc2.o -o test5_1  
main.o:main.c HEADFILE1  
        gcc -c main.c  
proc1.o: HEADFILE1 proc1.c HEADFILE2  
        gcc -c proc1.c  
proc2.o: HEADFILE2 proc2.c  
        gcc -c proc2.c
```

当头文件发生了改变之后，只要改变一下宏定义就行了。例如，想把 `myproc.h` 换成 `myproc2.h` 时，只需要在上面那段命令中改动第二行的宏定义。

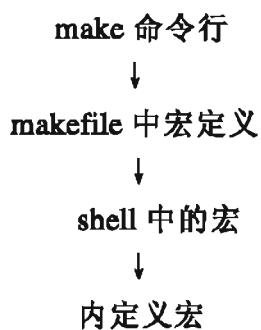
```
HEADFILE2=myproc2.h
```

或是在 `make` 命令行中定义。

```
make -f test5_1.mk "HEADFILE2=myproc.h"
```

这样可以大大减少维护 `makefile` 的工作量。

类似于属性的控制，前面几种宏的定义也要区分优先级。`make` 在处理 `makefile` 时，它将先给内定义(后面讲到)的宏赋值，再给载入的 shell 宏赋值，然后给 `makefile` 中的宏赋值，最后才处理 `make` 命令行中的宏定义。由于后面处理的宏会覆盖前面的定义，所以它们的优先级正好相反。下面给出优先级的排序，从上往下，优先级依次递减。



2. 内定义宏

内定义宏是 `make` 中提供的一些预定义的宏，便于程序员直接使用。在这里只讲解控制宏和动态宏。

(1) 控制宏

控制宏可以用来控制 `make` 的属性，或给出一些预定的值。由这两种不同的作用，可以把控制宏分为两类。

第一类宏和普通的宏类似，它代表某些特殊的值。主要有以下几种。

<code>.DIRSEPSTR</code>	表示路径名中用于分隔目录与文件名的符号，一般是斜杠。在 Linux 的 shell 中，提供了可以自定义分隔符或者提示符等的系统文件
<code>.MAKEDIR</code>	调用 <code>make</code> 的绝对路径名
<code>.NULL</code>	空字符串
<code>.OS</code>	正在运行的操作系统的名称
<code>.PWD</code>	运行 <code>make</code> 时的活动工作目录的绝对路径名
<code>.SHELL</code>	指定运行命令行时启动的 shell 类型

可以不清楚系统的某些情况，但是又要在 `makefile` 中使用这些数据，或想使 `makefile` 适应不同的环境时，使用这些内定义宏。引用的格式和一般宏的引用相同，为\$(宏名)。

第二类宏是属性宏，用来控制 `make` 的属性，与目标文件属性的书写和意义相同，作用域是整个 `makefile` 文件。当宏值不为空时，就为所有的目标文件设置该属性；如果为空，则无该属性。

属性宏的赋值格式和一般的宏不同，不必使用 \$ 和括号，只需要直接写出宏名即可。例如：

```
.IGNORE=$(NULL)
```

(2) 动态宏

动态宏在 `make` 执行过程中的值是动态改变的，它将从不同的规则中取不同的值，从而反映当前的目标或依赖文件。

- `$@` 是最直接的一个动态宏，它的值等于当前的目标文件的名字，但是如果目标文件是库成员，它则等于 `lib` 的名称。如下例：

```
file1.o file2.o:myprog.h  
cp #@ /backup  
mv $@  
#更新目标文件的具体命令行
```

上面规则将会实现如下功能：当目标文件过时时，将原来的目标文件备份复制到一个名为 `/backup` 的目录中，再将原来的目标文件删除，最后建立新的目标文件。如果更新的是 `file1.o`，则 `$@` 等于 `file1.o`，以上的操作都是对 `file1.o` 进行。当目标文件是 `file2.o` 时，`$@` 则等于 `file2.o`。

- `$%` 等于当前目标文件的名称，和 `$@` 相似，但目标是库文件时，`$@` 指示库名，而 `$%` 指示成员名。
- `$>` 适用于目标文件是库文件的形式，它等于 `lib` 的名称。`$>` 相当于 `$@`，但是 `$>` 不能用于目标文件是普通文件的规则。
- `$*` 等于目标文件去掉后缀的名称。
- `$&` 代表当前目标文件在所有的规则中的所有依赖文件。
- `$^` 代表当前目标文件在本规则中的依赖文件。

例如：目标文件 `A` 出现在两条规则中，一条依赖于文件 `B` 和 `C`，另一条依赖于 `D`。

```
A::B C  
# 命令行 1  
A::D  
# 命令行 2
```

在命令行 1 和 2 中，`$&` 都等于 `BCD`，但是 `$^` 的值不同，在命令行 1 中等于 `BC`，在命令行 2 中则等于 `D`。

- `$?` 代表当前目标文件的所有依赖文件中比目标文件新的文件列表。
- `$<` 代表当前规则的依赖文件中比目标文件新的文件列表。`$<` 和 `$?` 的关系类似于 `$^` 和 `$&`。
- `$$@` 目标文件名，如果目标文件是库的成员，则表示库名。
- `$$%` 目标文件名，如果目标文件是库的成员，则表示成员名。
- `$$*` 目标文件去掉后缀的名称。

- **\$\$>** 当目标文件是库的成员时使用，表示库名。

3. 修改宏

在宏定义之后，还可以使用 make 的相应功能，在宏展开时对其进行修改。

(1) 修改宏的扩展

当宏代表一个文件时，扩展宏时可以用描述符选择扩展文件名的那个部分。需要注意的是这些描述符只适用于代表文件名或至少具有文件名形式的宏。

下面是常用的描述符。

- d: 仅展开路径
- b: 展开文件名，但不包括扩展名
- f: 展开文件名，包括扩展名

如下例：

```
FILE=/user/cprogs/main.c
```

如果使用如下 make 命令。

```
$(FILE:d)
```

结果是：/user/cprogs/

描述符作用的宏可以是一个文件列表。这时，描述符要对列表中的所有文件都作用一次，然后将结果按原来的顺序排列，得到展开后的宏值。

(2) 替换宏中的字符串

可以在展开宏时对宏中的部分字符串进行替换，将宏展开成新的值。

替换使用如下格式。

宏名: s/原字符串/新字符串

make 会在该宏中检查指定字符串(原字符串)，然后将找到的原字符串用新的字符串进行替换。

当宏代表一个文件列表，并且要替换的字符串是文件的扩展名时，可以使用以下格式。

宏名:原扩展名=新扩展名

如果在宏的开头或者末尾加字符串，要用到两个特殊的符号：**^** 和 **+**。**^** 是宏对应字符串的起始符号，用其他字符串替换起始符号意味着在整个宏前面加上一个字符串；而**+**是宏值的结束符号，用字符串替代结束符号就是在宏后面加上一个字符串。它们的格式如下。

宏名:^ “前缀”

宏名:+ “后缀”

例如(宏 FILE=main):

```
$(FILE:+ ".o")
$(FILE:^ "/user/")
```

上面展开后所得结果分别是:main.o 和 /user/main。

5.2.2 内部规则

书写 makefile 时，会发现在目标文件和依赖文件以及命令菜单之间有一定的规律，许多规则需要重复书写，减慢了开发的效率。在这部分，将介绍处理此问题的一个办法—内部规则。

1. 内部规则的定义和使用

make 的内部规则是系统或程序员预先定义的一些特殊的规则。一般规则要列出文件的全名，而内部规则中出现的目标文件和依赖文件都只使用文件的扩展名。它通常被用来定义一些十分常用的依赖关系和更新命令。

make 内部规则中涉及的文件都使用文件的扩展名代替完整的文件名，所以 make 自动处理文件的能力主要是体现在内部规则所能识别的文件扩展名的多少上。与 C 编程有关的扩展名主要有下面几种。

- .o 目标文件
- .c C 语言文件
- .C C++ 源文件
- .h 头文件

这里所说的目标文件是指经过编译处理后的.o 文件，和前面所说的 makefile 中的规则中特定的目标文件不同。

在 C 语言中涉及到的内部规则如下。

```
.c:
$(CC)$(CFLAGS)-O $@ $<
.c .o:
$(CC)$(CFLAGS)-c $<
```

上面用到的两个宏 CC 和 CFLAGS 是 make 为了便于内部规则的书写而预先定义的。每

次执行一个 `makefile` 之前，系统就会将这些宏置成系统规定的初始值。程序员可以在自己的 `makefile` 中修改它们，也可以不加说明地直接使用。下表列出了与 C 程序有关的 `make` 预定义宏。

表 5-2

make 预定义宏

宏名	初始值	说明
<code>AR</code>	<code>Ar</code>	库管理命令
<code>ARFLAGS</code>	<code>-ruv</code>	库管理命令选项
<code>CC</code>	<code>cc</code>	C 编译命令
<code>CFLAGS</code>	<code>-O</code>	C 编译命令选项
<code>C++C</code>	<code>CC</code>	C++编译命令
<code>C++FLAGS</code>	<code>-O</code>	C++编译命令选项
<code>MAKE</code>	<code>Make</code>	Make 命令
<code>MAKEFLAGS</code>	<code>NULL</code>	Make 命令选项
<code>LIBSUFFIXE</code>	<code>.a</code>	库的扩展名
<code>A</code>	<code>.a</code>	库的扩展名

需要注意的是，内部规则可以是单后缀的，也可以是双后缀的。在单后缀的规则中，给出的是目标文件的后缀，但是没有依赖文件；而双后缀的规则中，以第二个后缀为扩展名的文件依赖以第一个后缀为扩展名的文件。命令菜单则给出从第一个后缀向第二个后缀转换的命令。

再来看前面例子中的两个内部规则。第一个内部规则用于维护 C 语言源程序，第二个规则指定以`.o` 结尾的目标文件依赖于以`.c` 为后缀的`c` 源文件，并给出了重新建立目标文件的命令。

```
gcc -O -c $<
```

表 5-2 中有两个比较特殊的宏：`.MAKE` 和 `.MAKEFLAGS`。它们不是 `make` 命令本身使用的，而是为了在 `makefile` 的命令菜单中再启动一个 `make` 过程，它在 `make` 的嵌套使用中很重要。在一个大中型的系统开发中，常常把系统分解成一些独立的模块，甚至连 `makefile` 也要分解成几个 `makefile` 文件。有时当系统是由多个程序员合作完成时，一般要各自书写自己负责部分的 `makefile`。各个模块的开发完成之后，要编制整个系统的 `makefile`，完成系统的集成。集成不是简单地将所有的 `makefile` 的内容合在一个文件中，因为那样会使最后的文件过于庞大而不便于阅读和维护，这时就要使用 `make` 的嵌套功能。就是在总的 `makefile` 中只给出各个模块的规则，在规则的命令行中再调用 `make` 命令对各个模块进行维护。

如果在 `makefile` 中对内部规则可识别后缀的目标文件没有显式地给出规则，`make` 就会从内部规则集中查找相应后缀的内部规则，应用于该文件。它将会删除目标文件的后缀，然后

加上预定义的依赖文件后缀，从而得到完整的依赖文件。并以内部规则中的命令菜单作为该目标文件的命令菜单，生成完整的规则。

从上面可以看出内部规则定义了由典型的依赖文件生成目标文件的规则，可以有效地缩短 `makefile` 的长度。

下面一段 `makefile`，前面已经使用过，现在来比较一下使用了内部规则后的 `makefile` 和没有使用过的 `makefile`。

旧的 `makefile` 如下。

```
test5_1:main.o proc1.o proc2.o  
        gcc main.o proc1.o proc2.o -o test5_1  
main.o:main.c myprog.h  
        gcc -c main.c  
proc1.o:myprog.h proc1.c myproc.h  
        gcc -c proc1.c  
proc2.o:myproc.h proc2.c  
        gcc -c proc2.c
```

使用了内部规则后的 `makefile`:

```
test5_1:main.o proc1.o proc2.o  
        gcc main.o proc1.o proc2.o -o test5_1  
main.o proc1.o:myprog.h  
proc1.o proc2.o:myproc.h
```

因为其中的 `main`, `proc1` 和 `proc2` 都是标准的依赖关系和更新命令，可以使用内部规则来代替。

这样程序就简单多了，当书写一个很长的 `makefile` 时，这种优势将会更加明显。

2. 修改内部规则

通常情况下，`make` 预定义的内部规则足够使用，但是当程序员有特殊的要求时，也可以对内部规则进行修改。

可以使用两种方法对内部规则进行修改。第一就是利用内部规则中使用的宏，改变宏的定义来改变内部规则的行为，这样就不需要修改规则本身，但是它有一个副作用就是所有用到修改宏的内部规则都被修改。

第二种方法是直接修改内部规则，主要是修改内部规则的命令菜单。

后缀组合:

命令菜单

规格类似于 make 预定义的内部规则格式，也是一般使用双后缀的形式。

3. 定义新的后缀和内部规则

当要经常使用某个内部规则不能识别的文件扩展名时，可以自己定义新的后缀和内部规则。

定义新后缀的格式如下。

New Roman 后缀名 1 后缀名 2 ...

说明了之后，就可以使用前面讲过的格式定义新的内部规则。

5.2.3 make 递归

当开发一个大中型项目时，程序员可以根据特定的子系统，把代码分成独立的部分，而每个部分包含一个子系统的代码。这样做的后果就是整个项目的 make 文件将会很大。因此，程序员想找到一种方法，使得每个子系统都有自己的 makefile。这样就能够更加容易地维护系统，因为顶级的 makefile 不必包含整个项目中的每个子系统的细节，可以在每个子系统的 makefile 中处理这些细节。

为了实现上述功能，make 提供了相应的变量和选项：使用 MAKE 变量，它能调用递归 make，并且向该 make 传递许多命令行中的相关参数。使用-C 选项告诉 make 进入一个特定的目录，在那里处理该目录中的 makefile。

用递归 make 处理项目的每个子目录并创建文件。每一个子目录还可能有子目录，这就需要创建过程来进行判断。通过使用递归 make，就可以遍历项目的整个目录树结构，并且创建所有需要的文件。

可以采用如下的格式。

目标文件名：

`$(MAKE) -C 目录名`

当有大量的子目录时，可以用循环进入每一个子目录。

另外还可以在主 make 和由它调用的其他 make 之间，进行变量设置通信。可以用两种方法实现这一功能：第一是使用一个被所有 makefile 包含的文件；第二是从顶级 make 中输出变量到它的子进程，它使用 export 关键词，同时要输出选项传递给编译器。

然后就要处理怎样把子目录生成的文件组合到项目中去的问题。这主要取决于不同的需要，子系统可以是独立的执行文件，创建的库或只是可执行文件的一部分。一个较为常用的方法就是指定一个目录给目标文件，然后把所有的目标文件都放在里面。还可以创建一个库。

下一节将介绍库的使用。

5.2.4 依赖性的计算

前面已经较为系统地介绍了 `makefile` 的书写，其中最关键的步骤就是要了解清楚文件之间的依赖性关系，一般采取程序员手工计算依赖性。如果用这种方法，当要包含其他文件或把一个包含语句删除的时候，程序员就不得不修改 `makefile`，当项目很大时，这将是一件既耗时又费力的事情。

可以创建自动的依赖性处理，它能计算那些源文件(或者头文件)和目标文件之间的依赖性。这就意味着 `makefile` 将永远不用被更新，即使是添加了新的模块、代码和新的头文件相关的时候。

实现此功能的最基本的想法，就是创建一个依赖文件，包含源文件的必要信息。还可以使用 `make` 的 `include` 指令，把这些文件读进解释器，类似于 C 程序中的 `include` 命令，这样就可以把它们作为 `makefile` 的一部分。



include 操作符的特性

第一个特性：它能自动重新编译它所包含的文件。如果这些文件不存在或不是最新的，`make` 就会在当前的 `makefile` 文件中寻找一条规则以便重新创建它们；如果该规则存在，就用它创建文件。

第二个特性：如果其中任何一个文件需要重新编译，`make` 将会自动重新设置自己，使得所有的文件都能被更新。传统的 `make` 应用不支持这种重置功能，这就意味着即使文件被更新了，但是仍然要使用老的依赖性文件。

这样做好处是：第一，由于一个源文件只对应一个依赖性文件，可以在源文件中声明它和依赖性文件之间的关系，因此允许相关信息被自动更新；第二，是由于每个源文件的依赖性信息在另外的独立文件中，当只有一个文件被改变时，不用更新所有文件的依赖性信息。

使用 `gcc` 的 “-M 输出文件”选项，可以生成依赖性文件。这个选项告诉编译器禁止一般操作。它将检测源文件并且输出一个 `make` 规则，指明给定文件的依赖性。输出的规则把.o 文件放在左边，然后是 c 文件名和任何包含的头文件。这还不能形成一个完全正确的解决方案，程序员还必须把依赖性文件尽可能逻辑地列出它们之间的依赖性。所以如果一个头文件改变了，那么依赖性就被更新了，解决这个问题只要调用 `sed` 就行了。

下面来看一个较复杂的例子。

```
# Lines starting with the pound sign are comments.  
#  
# This is one of two options you might need to tweak.
```

```
EXECUTABLE=myprogram
```

```
# You can modify the below as well, but probably
# won't need to
#
# CC is for the name of the C compiler.CPPFLAGS denotes pre-processor
# flags, such as -l options.CFLAGS denotes flags for the C compiler.
# CXXFLAGS denotes flags for the C++ compiler.You may add additional
# setting here, such as PFLAGS, if you are using other languages such
# as Pascal
```

```
CC=gcc
CPPFLAGS=
CFLAGS=Wall -O2
CXXFLAGS=$(CFLAGS)
COMPILE=$(CC)$(CPPFLAGS)$(CFLAGS)-c
```

```
SRCS=$(wildcard *.c)
OBJS=$(patsubst %.c, %.o, $(SRCS))
DEPS=$(patsubst %.c, %.d, $(SRCS))
```

```
# "all" is the default target.Simple make it point to myprogram.
```

```
all:$(EXECUTABLE)
```

```
# Define the components of the program, and how to link them together.
# These components are defined as dependencies;that is, they must be
# made up-to-date before the code is linked.
```

```
$(EXECUTABLE):$(DEPS) $(OBJS)
$(CC) -o $(EXECUTABLE) $(OBJS)
```

```
# Specify that the dependency files depend on the C source files.
```

```
% .d: %.c
$(CC) -M $(CPPFLAGS) $<> $@
$(CC) -M $(CPPFLAGS) $< lsed s/\\ .o/ .d/ > $@
```

```
# Specify that all .o files depend on .c files, and indicate how
# the .c files are converted (compiled) to .o files.
```

```
% .o:%.c
$(COMPILE) -o $@ $<
```

clean:

```
-rm $(OBJS) $(EXECUTABLE) $(DEPS) * ~
```

explain:

```
@echo "The following information represents your program:"
@echo "Final executable name:$(EXECUTABLE)"
@echo "Source files:$(SRCS)"
@echo "Object files:$(OBJS)"
@echo "Dependency files:$(DEPS)"
```

```
depend:$(DEPS)
```

```
@echo "Dependencies are now up-to-date."
```

```
-include $(DEPS)
```

下面介绍上面文件涉及的一些变量和命令。

文件前面是一段宏定义，其中变量 CPPFLAGS 用于预处理指令，在计算依赖性的时候，只有这些选项能够传递给 gcc。

命令行 SRCS:=\$(wildcard *.c)的功能是产生一个源文件列表并且 SRCS 保存了这个文件列表。命令行 OBJS:=\$(patsubst %.c, %.o, \$(SRCS))把源文件列表转换成了目标文件列表。命令行 DEPS:=\$(patsubst %.c, %.d, \$(SRCS))把源文件列表转换成依赖性文件列表。转换的功能由函数 patsrbsr 实现。

随后的代码使用了如下规则。

```
$(EXECUTABLE):$(DEPS) $(OBJS)
$(CC) -o $(EXECUTABLE) $(OBJS)
```

它将使依赖性文件列表被显示出来，可以提醒程序员，当文件编译的时候，依赖性文件需要被更新。

然后就到了本部分的正题，怎样计算依赖性。它是如下三行的规则。

```
% .d: %.c
$(CC) -M $(CPPFLAGS) $<> $@
$(CC) -M $(CPPFLAGS) $< | sed s/\\ .o/ .d/ > $@
```

第一行规定每个依赖性文件至少依赖一个 c 源程序，并且在源程序被改动的时候，必须被重新生成。第二行和第三行列出了生成依赖性文件的命令。前一个命令只是调用 gcc，并把它的输出直接写入依赖性文件中。后一个命令又一次调用 gcc，把规则文件名从.o 改变为.d，并且把结果加到文件的末尾。这么做的结果使依赖性文件即使是只有头文件发生了变化，它也可以被重新生成。

下面是 clean 目标，它将根据依赖性列表删除相关文件。如果是在已经删除了文件的目录里执行 make clean，依赖性文件将被重新生成，然后被删除。

然后是 explain 目标，这个目标将会显示一些信息，报告什么被检测，便于程序员知道将要做什么。

随后的是 depend 目标，它用来重新生成依赖性文件。

最后的是 include 指令，前面的横线用来限制因为包含文件不存在而产生警告信息，因为文件被自动生成并以任何形式被包含时，警告信息将使屏幕变得混乱。在 DEPS 变量中，将列出被包含的文件。

在一个空目录中执行，将会发现它创建了依赖性文件，然后编译程序。

在这里，makefile 将进行以下工作：

检测项目中所有 c 源文件的文件名；

根据 c 源文件，检测所有的相对应的目标文件；

根据上述文件，检测对应的依赖性文件；

为每个源文件检查依赖性，并且把它们存储在一个文件中，以备以后自动重用；

需要的时候，将自动重新生成依赖性文件。

5.3 库 的 使用

代码的重用性是当代计算机编程语言中一个重要的概念。可以把编译好的目标文件模块统一放到一个库中，使得程序员可以在不同的程序中共享这些代码。

在 Linux 操作系统下，最后链接生成可执行文件时，如果链接的是一般的.o 文件，则整个文件的内容都会被装入可执行文件中；如果链接的是库，则只是从库中找到程序中用到的变量和函数，将它们装入可执行文件中，那些放在库中但是没有被程序所引用的变量和函数则不会被链接到最终的可执行文件。

所以，使用库可以节省大量的开发时间。在写较大的程序时，最好把程序模块放在库中。

5.3.1 创建库和维护库

库中的所有文件都叫作库的成员，可以使用以下格式来表示库的成员：

库名（成员名）

例如：

`mylib.a(mytest5_1.o)`

用来表示库 `mylib.a` (.a 为库常用的扩展名) 中的一个文件 `mytest5_1.o`。

库的成员还有另一种格式：

库名 ((entry))

其中 `entry` 成为入口点。当 `make` 识别到这种形式后，就将库名置成 `.LIBRARY` 属性，将 `entry` 置成 `.SYMBOL` 属性，然后根据 `entry` 从库中找到相应的成员文件。返回定义该 `entry` 的成员文件的创建时间，并且用真正的文件名替换 `entry`。

使用下面格式来说明库和成员的依赖关系：

库名：库名（成员名 1） 库名(成员名 2) ...

`make` 将会自动将库作为目标文件，并赋予 `.LIBRARY` 属性，而将括号中的成员作为依赖文件，赋予 `.LIBRARYM` 属性。

还可以使用另外一种格式来说明依赖关系：

库名 `.LIBRARY`：成员名 1 成员名 2 ...

例如要维护一个名为 `mylib` 的库，`makefile` 文件格式如下：

```
mylib:mylib(test5_1.o)
        gcc -c test5_1.c
        ar -rv mylib test5_1.o
        rm test5_1.o
mylib:mylib(test2.o)
        gcc -c test2.c
        ar -rv mylib test2.o
```

```
rm test2.o
...

```

上面 `makefile` 文件中，使用了 `ar` 命令，它的作用是从`.o` 结尾的目标文件更新库，格式如下：

`ar -ruv 库名 目标文件名`

从 `makefile` 中还能看到两端代码非常相似，而且执行的命令也是相同的，可以使用动态宏来合并不同的文件，代码段如下所示：

```
mylib:mylib(test5_1.o) mylib(test2.o)
    gcc -c ($?:b) .c
    ar -ruv $@ $?
    rm -f $?
```

还可以使用内部规则进一步简化代码：

```
.O.a:
    $(AR) $(CFLAGS) $@ $?
    rm -f $?
```

由于 `make` 中已经定义了`.c.o` 的内部规则，会自动进行这一转换，所以上述代码还能够简写为：

```
mylib:mylib(test5_1.o) mylib(test2.o) ...
```

5.3.2 库的链接

要链接使用的库，首先要在可执行文件的依赖文件列表中加入链接时要用到的函数，这个函数要指明目标文件所在的库和`.o` 文件名。

库名（成员文件名）

或是

库名 ((成员 entry))

例如代码

```
prog.exe:prog.o mylib(myfunc1)
    # 链接 prog 可执行文件的命令行
```

上段代码说明了一个可执行文件 `prog.exe`, 它依赖于相应的目标文件 `prog.o` 和库 `mylib` 中的一个函数 `myfunc1`。命令行就是一般的链接命令 `ld`, 只要在参数中加入库名 `mylib` 就可以了。

如果考虑 `make` 定义的内部规则, `make` 认为可执行文件一般依赖于相应的目标文件, 所以可以给出`.o.e` 规则。

```
.o.e:
$(LD)$(LDFLAGS) -o $@ $< $(LDLIBS)
```

规则中使用了三个系统内定义的宏:`LD`, `LDFLAGS` 和 `LDLIBS`。其中 `LD` 的初始值是链命令 `ld`; `LDFLAGS` 是 `ld` 命令的选项; 而 `make` 将一些常用的标准函数库的列表定义成宏 `LDLIBS`。这样, 这些库就被链接进去了。

如果是自定义的库, 就不能使用这个内部规则, 因为在它的链接命令中没有包含自定义的宏。需要进行如下的修改。

`LDLIBS+=库名`

修改后就可以使用内部规则维护文件了。

5.4 小结与练习

5.4.1 小结

本章讲述的是程序自动维护工具 `make`, 它能自动管理软件编译的内容、时机和方式, 从而使程序员能够集中精力编写程序代码。

这一章我们讲解了一个简单的 `makefile`, 介绍了如何修改它, 使其更加精简, 然后介绍了 `make` 的属性控制。还讲述了 `make` 的高级使用, 包括宏、内部规则、`make` 递归等。最后介绍了库的概念和它的创建和使用。

5.4.2 习题与思考

使用动态宏来简写以下一段 makefile 文件（维护一个名为 mylib 的库）。

```
mylib:mylib(test1.o)
    gcc -c test1.c
    ar -rv mylib test1.o
    rm test1.o
mylib:mylib(test2.o)
    gcc -c test2.c
    ar -rv mylib test2.o
    rm test2.o
....
```

然后使用内部规则进一步简化。

第6章

文件操作

文件系统简介

基于文件描述符的 I/O 操作

文件的其他操作

特殊文件的操作

小结与练习



本章介绍 Linux 系统中的文件以及与文件有关的操作。在 C 编程环境中，与文件有关的操作主要是 I/O 操作，即基于文件描述符的 I/O 操作。此外，还将介绍其他一些与文件有关的操作。

在 Linux 系统中，有关 I/O 的操作可以分为两类。它们是基于文件描述符的 I/O 操作和基于流的 I/O 操作。它们有着各自不同的特点和优势。有些情况下它们是可以相互替代的，有些情况下则不是。基于流的 I/O 操作将放在下一章里介绍。

基于文件描述符的 I/O 操作是通过文件描述符对一个文件执行 I/O 操作的。文件是一个十分重要的概念。通常保存在外存中的数据都是以文件的形式保存的。文件描述符则是用于描述被打开文件的索引值。通常情况下，都是通过文件描述符打开一个文件执行 I/O 操作的。

通过本章的学习，不但将了解这种基于文件描述符的 I/O 操作，还会对 Linux 系统的文件及文件系统有所了解。

6.1 文件系统简介

文件和文件系统是重要而复杂的概念。在此，并不打算详细介绍文件和文件系统，因为这并不是本书所要讨论的内容，而只是根据本章讨论的需要，介绍一些相关的基本知识。

6.1.1 文件

文件是有名字的一组相关信息的集合，在 Linux 系统中，文件的准确定义是不包含有任何其他结构的字符流。通俗地说，就是文件中的字符与字符之间除了同属于一个文件之外，不存在任何其他的关系。文件中字符的关系，是由使用文件的应用程序来建立和解释的。

每一个文件都具有特定的属性。Linux 系统的文件属性比较复杂，主要包括文件类型和文件权限两个方面。

1. 文件类型

Linux 下的文件可以分为 5 种不同的类型。它们是普通文件、目录文件、链接文件、设备文件和管道文件。下面给出它们的具体介绍。

(1) 普通文件

普通文件也称正规文件，是最常见的一类文件，也是最常使用到的一类文件。其特点是不包含有文件系统的结构信息。通常所接触到的文件，包括图形文件、数据文件、文档文件、声音文件等等都属于普通文件。这种类型的文件按其内部结构又可细分为两个文件类型：文本文件和二进制文件。

- 文本文件：文本文件是以字符（通常是 ASCII 码）表示的，是以行为基本结构的信息存储方式。

- 二进制文件：二进制文件是按信息在内存中的格式表示的，它通常不能直接查看，而必须使用相应的软件。

(2) 目录文件

目录文件是用于存放文件名及其相关信息的文件。是内核组织文件系统的基本节点。目录文件可以包含下一级目录文件或普通文件。

 Tip

在 Linux 中，目录文件是一种文件。但 Linux 的目录文件和其他操作系统中的“目录”的概念不同，它是 Linux 文件的一种。当然，在实际使用时也可以不仔细区分这两种说法，甚至在本书的具体叙述中，有时就将目录文件简称为目录，但要注意它们在概念上的不同。

(3) 链接文件

链接文件是一种特殊的文件。它实际上是指向一个真实存在的文件的链接。比如用户要在一个目录文件中使用其他目录文件下的文件时，并不需要将其复制过来，而只需在此目录中建立一个链接文件指向所要调用的文件。在具体使用时，并不会感觉到它们有什么不同。根据链接对象的不同，链接文件又可以细分为硬链接文件和符号链接文件。

(4) 设备文件

设备文件是 Linux 中最特殊的文件。正是由于它的存在，使得 Linux 系统可以十分方便的访问外部设备。Linux 系统为外部设备提供一种标准接口，将外部设备视为一种特殊的文件。用户可以像访问普通文件一样访问外部设备。这就使 Linux 系统可以很方便的适应不断发展的外部设备。通常 Linux 系统将设备文件放在 /dev 目录下。设备文件使用设备的主设备号和次设备号来指定某外部设备。主设备号用于说明设备类型，次设备号用于说明具体设备。例如，以 IDE 硬盘为第一主盘，它的第三个分区的设备文件就是 /dev/hda3。其中 hd 是主设备号，a3 是次设备号。根据访问数据方式的不同，设备文件又可以细分为两种类型：块设备文件和字符设备文件。

- 块设备文件：块设备文件是以固定长度的块访问数据的。
- 字符设备文件：字符设备文件是以指定字符（通常是一个字符）访问数据的。

大多数外部设备都提供两种访问方式。但对每一种设备来说，都有其最佳的访问方式。



在设备文件中有一个极其特殊的文件 /dev/null。所有放入这一设备的数据都将不再存在。可以将它看成是删除操作。

(5) 管道文件

管道文件也是一种很特殊的文件。主要用于不同进程间的信息传递。当两个进程间需要进行数据或信息传递时，可以通过管道文件。一个进程将需传递的数据或信息写入管道的一端，另一进程则从管道的另一端取得所需的数据或信息。通常管道是建立在高速缓存中的。采用先进先出的规定处理其中的数据。可以细分为有名管道和无名管道两种。

2. 文件权限

Linux 系统是一个典型的多用户系统，不同的用户处于不同的地位。为了保护系统的安全性，Linux 系统对不同用户访问同一文件的权限做了不同的规定。

对于一个 Linux 系统中的文件来说，它的权限可以分为三种：读的权限、写的权限和执行的权限。分别用 r、w 和 x 表示。

不同的用户具有不同的读、写和执行权限。对于一个文件来说，它都有一个特定的所有者，也就是对文件具有所有权的用户。同时，由于在 Linux 系统中，用户是按组分类的，一个用户属于一个或多个组。所以文件所有者以外的用户又可以分为文件所有者的同组用户和其他用户。因此 Linux 系统按文件所有者、文件所有者同组用户和其他用户三类规定不同的文件访问权限。



系统管理员 root 用户是一个非常特别的用户，此用户对系统具有最高的控制权。对于系统中的所有文件 root 用户都有读、写以及执行的权限。

6.1.2 文件的相关信息

与 Linux 系统中的文件相关的信息有三项。它们是文件的目录结构、索引节点和文件的数据本身。

1. 文件的目录结构

系统的每一个目录都处于一定的目录结构中，该结构含有目录中所有的目录项的列表，每一个目录项都含有一个名称和索引节点。借助于名称，应用程序可以访问目录项的内容。而索引节点号则提供了所需引用文件自身的信息。

2. 索引节点

在 Linux 系统中，所有的文件都有一个与之相连的索引节点（inode）。索引节点是用来保存文件信息的。索引节点包含如下信息。

- 文件使用的设备号。
- 索引节点号。
- 文件访问权限。
- 链接到此文件的目录数。
- 所有者用户识别号。
- 组识别号。
- 设备文件的设备号。
- 以字节为单位的文件容量。
- 包含该文件的磁盘块的大小。
- 该文件所占的磁盘块。

- 最后一次访问该文件的时间。
- 最后一次修改该文件的时间。
- 最后一次改变该文件状态的时间。

在系统中定义了 stat 结构体来存放这些信息。stat 结构的定义如下。

```
struct stat
{
    dev_t st_dev;           /*device*/
    ino_t st_ino;          /*inode*/
    mode_t st_mode;         /*projection*/
    nlink_t st_nlink;       /*number of hard links*/
    uid_t st_uid;           /*user ID of owner*/
    gid_t st_gid;           /*group ID of owner*/
    dev_t st_rdev;          /*device type(if inode device)*/
    off_t st_size;          /*total size,in bytes*/
    unsigned long st_blksize; /*blocksize for filesystem*/
    unsigned long st_blocks; /*number of blocks allocated*/
    time_t st_atime;        /*time of last access*/
    time_t st_mtime;        /*time of last modification*/
    time_t st_ctime;        /*time of last change*/
};
```

可以通过系统调用访问 stat 结构来获取索引节点的相关信息。



从索引节点包含的信息中，可以看到两个参数 st_dev 和 st_rdev。它们具有不同的含义。
st_dev 对应于每一个文件名，代表包含这个文件名和相应的索引节点的文件系统的设备号。
st_rdev 则只有字符设备文件或块设备文件才具有，表示的是实际设备的设备号。

3. 数据

通常文件中都包含有一定的数据。普通文件和目录文件都有相应的硬盘区域储存数据。这些数据是储存在由索引节点指定的位置上的。而其他一些特殊文件，如设备文件等，并不具有这样的在硬盘上的储存区域。

6.1.3 文件系统

文件系统是指按一定规律组织起来的有序的文件组织结构。是构成系统中所有数据的基础。系统中的所有文件都是驻留在文件系统中某一特定的位置。Linux 系统提供的文件系统

是树形的层次结构系统。所有文件最终都归结到根目录 “/” 上。Linux 支持多种文件系统，在此不对其进行详细叙述。当前最通用的文件系统是 ext2 系统。用户也可以根据需要自行选取。

6.2 基于文件描述符的 I/O 操作

基于文件描述符的 I/O 操作是 Linux 系统 I/O 操作的一种。它进行 I/O 操作时使用文件描述符与文件建立联接。文件描述符是一个整数，是用于描述被打开文件的索引值的。它指向该文件的相关信息记录表。下一章要介绍的基于流的 I/O 操作其实质也是依赖于文件描述符的。因此要理解和掌握好有关文件描述符的概念以及基于文件描述符的 I/O 操作。

6.2.1 文件的创建、打开与关闭

要对一个文件进行操作，首先要求这个文件存在，其次要在操作前将这个文件打开。这样才能实现对该文件的操作。当操作完成以后，则必须将文件关闭。文件的创建、打开与关闭操作是 I/O 操作的第一步。

1. 文件的创建

创建文件的系统调用是 `creat`。其具体说明如下。

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int creat (const char *pathname, mode_t mode);
```

其中参数 `pathname` 是一个字符串指针，用于表示需要打开文件的绝对路径名或相对路径名。参数 `mode` 用于指定所创建文件的权限。其取值及其含义见表 6-1。具体应用时，使用按位逻辑加的方法根据需要对其进行组合。

表 6-1 mode 的取值及其对应含义

mode	含 义
S_IRUSR	文件所有者的读权限位
S_IWUSR	文件所有者的写权限位
S_IXUSR	文件所有者的执行权限位
S_IRGRP	所有者同组用户的读权限位

续表

mode	含 义
S_IWGRP	所有者同组用户的写权限位
S_IXGRP	所有者同组用户的执行权限位
S_IROTH	其他用户的读权限位
S_IWOTH	其他用户的写权限位
S_IXOTH	其他用户的执行权限位

此外还定义了三个比较常用的逻辑组合。

S_IRWXU: (S_IRUSR|S_IWUSR|S_IXUSR)

S_IRWXG: (S_IRGRP|S_IWGRP|S_IXGRP)

S_IRWXO: (S_IROTH|S_IWOTH|S_IXOTH)

当调用成功时，系统调用 `creat` 返回值为该文件的描述符。此时文件以只读方式打开。失败时返回值为 -1。

2. 文件的打开

系统调用 `open` 用于打开文件，其说明如下。

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

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

参数 `pathname` 和 `mode` 的描述如前。参数 `flags` 是用于描述文件打开方式的参数，它的具体取值见表 6-2。具体调用时，`flags` 的值可由表中取值逻辑加组合得到。其中 `O_RDONLY`、`O_WRONLY` 和 `O_RDWR` 三个值必须且只需包含其一。

表 6-2 `flags` 的取值及其含义

Flags	含 义
O_RDONLY	以只读方式打开文件
O_WRONLY	以只写方式打开文件
O_RDWR	以读写方式打开文件
O_CREAT	若所打开文件不存在则创建此文件
O_EXCL	若所打开文件时设置了 O_CREAT 且该文件存在则导致调用失败
O_NOCTTY	若在打开 <code>tty</code> 时进程没有控制 <code>tty</code> 则不控制终端

续表

Flags	含 义
O_TRUNC	若以只写或读写方式打开一个已存在文件时，将该文件截至 0
O_APPEND	向文件添加内容时将指针置于文件的末尾
O_NONBLOCK	用于非阻塞套接口 I/O，若操作不能无延迟的完成则在操作前返回
O_NODELAY	同 O_NONBLOCK
O_SYNC	只在数据被写外存或其他设备之后操作才返回

调用成功时，系统调用 open 的返回值为所打开文件的描述符，调用失败则返回-1，并置 errno 为相应错误编号。



在执行文件操作时，不能保证调用总是成功返回的，所以在调用 creat 函数、open 函数以及后面将介绍的其他用于文件操作的函数时，都需要检测是否发生错误，并在发生错误时终止程序。

一个文件打开操作的简单例子。

【程序 6_1】

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

int write_open(const char *pathname, mode_t mode)
{
    int retval;
    if((retval=open(pathname, O_WRONLY|O_CREAT|O_TRUNC, mode)==-1))
    {
        printf("ERROR, OPEN FILE FAILED!\n");
        exit(254);
    }
    return retval;
}
```

【程序 6_1】是一个以只读方式打开某给定文件的函数，如果该文件不存在，则创建此文件；如果该文件存在，则将文件长度截至 0。

3. 文件的关闭

关闭文件的系统调用为 `close`。其具体说明如下。

```
#include <unistd.h>
int close(int fd);
```

其唯一参数 `fd` 是需关闭文件的描述符。

调用成功时，返回值为 0；调用失败时，返回值为 -1，并置 `errno` 为 `EBADF`。表示关闭的不是一个有效的、已打开的文件描述符。



当对文件进行打开和关闭操作时，还会对其相关的信息产生相应的影响。当打开一个文件时，该文件描述中的引用计数器被加 1。而关闭一个文件时，该文件描述中的引用计数器被减 1。当引用计数器的值减至 0 时，系统调用 `close` 不仅将释放该文件的描述符，而且也将释放该文件所占的描述表项。

当关闭的不是一个普通文件时，可能会产生一些其他的影响。例如，在关闭管道文件的一端时，将影响到管道的另一端。

6.2.2 文件的读写操作

文件的读写是 I/O 操作的核心内容。上文已经介绍了如何打开一个文件，但是要实现文件的 I/O 操作就必须对其进行读写。文件的读写操作所用的系统调用分别是 `read` 和 `write`。它们的详细说明如下。

1. 写文件的系统调用

```
#include <unistd.h>
ssize_t write(int fd, void *buf, size_t count);
```

其中参数 `fd` 表示将对之进行写操作的文件打开时返回的文件描述符。参数 `buf` 是一个指向缓冲区的指针，该指针指向存放将写入文件的数据的缓冲区。参数 `count` 表示本次操作所要写入文件的数据的字节数。

调用成功时返回值为所写入的字节数，调用失败时返回 -1，同时将 `errno` 设置为相应值。下面来看一个写文件的具体例子。

【程序 6_2】

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

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <fcntl.h>

#define NEWFILE (O_WRONLY|O_CREAT|O_TRUNC)
#define SIZE 80

int write_buffer(int fd, const void *buf, int count);

int main(void)
{
    int outfile;
    char filename[]={"test.dat"};
    char buffer[SIZE];

    if(outfile=open(filename, NEWFILE, 0640)==-1)
    {
        printf("ERROR, OPEN FILE FAILED! \n");
        exit(255);
    }

    while(!strcmp(buf, "quit"))
    {
        gets(buffer);
        if(write_buffer(outfile, buffer, SIZE)==-1)
        {
            printf("ERROR,WRITE FAILED: \n",sys_errlist[errno]);
            exit(255);
        }
    }
    close(outfile);
    return 0;
}
```

```

int my_write(int fd, char *buf, size_t count)
{
    int i,n;
    for(i=0; i<count; ++i)
    {
        write_buf[write_offset++]=*buf++;
        if(write_offset==BUFSIZE)
        {
            write_offset=0;
            n=write(fd, write_buf, sizeof(write_buf));
            if(n!=BUFSIZE)
                return -1;
        }
    }
    return -1;
}

```

【程序 6_2】 创建了文件“test.dat”，并从终端输入若干字符串写入文件。直至输入字符串“quit”时结束。

2. 读文件的系统调用

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

参数 `fd` 是打开所读文件时返回的文件描述符。参数 `buf` 是一个指向缓冲区的指针，该缓冲区用于存放所读入的数据。参数 `count` 表示本次读操作希望读取的字节数。

操作成功时，返回值是本次读操作实际读取的字节数。调用失败时，返回-1。

【程序 6_2】 生成了一个文件 test，并在其中写入了一些数据。当需要这些内容时就使用系统调用 `read` 读取其中的数据。

【程序 6_3】

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```

```
#include <errno.h>
#include <unistd.h>
#include <fcntl.h>

#define NEWFILE (O_WRONLY|O_CREAT|O_TRUNC)
#define BUFSIZE 1024
#define size 1

int my_read(int fd, char *buf, size_t count);
int my_write(int fd, char *buf, size_t count);

static char read_buf[BUFSIZE];
static int read_offset=BUFSIZE;
static int read_max=BUFSIZE;
static char write_buf[BUFSIZE];
static int write_offset=0;

int main(void)
{
    char infile[ ]={"test.dat"};
    char outfile[ ]={"backup.dat"};
    char buf[size];
    int infd, outfd, count;

    if(infd=open(infile, O_RDONLY)==-1)
    {
        printf("ERROR, OPEN READ FILE FAILED: \n", sys_errlist[errno]);
        exit(255);
    }
    if(outfd=open(outfile, NEWFILE, 0600)==-1)
    {
        printf("ERROR, OPEN WRITE FILE FAILED: \n", sys_errlist[errno]);
        exit(255);
    }
    while(count=read(infd,buf,sizeof(buf))>0)
    {
        if(write(outfd, buf, count)!=count)
```

```
    printf("ERROR, WRITE FILE FAILED: \n", sys_errlist[errno]);
    exit(255);
}

if(count== -1)
{
    printf("ERROR, READ FILE FAILED: \n", sys_errlist[errno]);
    exit(255);
}

close(infd);
if(write_offset>0)
{
    write(outfd, write_buf, write_offset);
    write_offset=0;
}
close(outfd);

}

int my_read(int fd, char *buf, size_t count)
{
    int i;

    for(i=0; i<count; ++i)
    {
        if(read_offset==read_max)
        {
            read_offset=0;
            read_max=read(fd, read_buf, sizeof(read_buf));

            if(!read_max)
                return i;
        }
        *buf+=read_buf[read_offset++];
    }
    return i;
}
```

```
int my_write(int fd, char *buf, size_t count)
{
    int i,n;
    for(i=0; i<count; ++i)
    {
        write_buf[write_offset++]=*buf++;
        if(write_offset==BUFSIZE)
        {
            write_offset=0;
            n=write(fd, write_buf, sizeof(write_buf));
            if(n!=BUFSIZE)
                return -1;
        }
    }
    return -1;
}
```

【程序 6_3】的作用是将文件 test.dat 中的内容拷贝到 backup.dat 文件中。程序中使用了 1024 字节的缓冲区，以减少系统调用的次数。

6.2.3 文件的定位

前一节已经介绍了如何读写一个文件，但到目前为止，这种操作还只能从文件的开头或结尾开始，这种读写方式称为顺序读写的方式。有时需要更灵活地操作一个文件，也就是说，希望能从文件的任意位置开始进行读写。在这种情况下，就要用到文件的定位。

先来看一下文件定位所用的系统调用。

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fildes, off_t offset, int whence);
```

在说明此调用的参数之前，首先说明偏移（offset）的概念。要在一个文件中进行定位，需要确定一个相对的位置点，然后指出相对于这一位置所偏移的量。

系统调用 lseek 的三个参数中，参数 fildes 是所操作文件的文件描述符。参数 offset 是所取的偏移量。参数 whence 用于表示计算偏移值的相对位置。其具体的取值和相应含义见表 6-3。

系统调用成功时，返回值为相对于文件开头的实际偏移量；调用失败时，返回值为-1，并将 `errno` 设置为相应值。

表 6-3

whence 的取值和相应含义

Whence	含 义
<code>SEEK_SET</code>	从文件的开头计算偏移值
<code>SEEK_CUR</code>	从当前的位置计算偏移值
<code>SEEK_END</code>	从文件的末尾急速偏移值



`lseek` 调用允许将当前读写位置移动到除文件开头之前的任何位置。也可以将其移动到文件的结束之后，如果此时在此位置写入数据则文件将以此位置为结束位置。

来看一个使用了 `lseek` 系统调用的简单例子。

【程序 6_4】

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>

#define NEWFILE (O_WRONLY|O_CREAT|O_TRUNC)

int main(void)
{
    char buf1[ ]={"abcdefghijkl"};
    char buf2[ ]={"1234567890"};
    int fd;
    int length;

    if(fd=open("test.hole", NEWFILE, 0600)==-1)
    {
        printf("ERROR, OPEN WRITE FILE FAILED: \n", sys_errlist[errno]);
        exit(255);
    }

    length=strlen(buf1);
```

```
if(write(fd, buf1,length)!=length)
{
    printf("ERROR, OPEN WRITE FILE FAILED: \n", sys_errlist[errno]);
    exit(255);
}

if(lseek(fd, 50, SEEK_SET)==-1)
{
    printf("ERROR, LSEEK FAILED: \n", sys_errlist[errno]);
    exit(255)
}

length=strlen(buf2);
if(write(fd, buf2,length)!=length)
{
    printf("ERROR, OPEN WRITE FILE FAILED: \n", sys_errlist[errno]);
    exit(255);
}

return 0;
}
```

【程序 6_4】在 test.hole 文件中写入了中间有一段空隙的数据。可以通过 shell 命令 ls 查看文件的大小为 60 字节。用 od 命令查看文件的实际内容。

6.3 文件的其他操作

通过上文的介绍，已经可以执行简单的 I/O 操作了，但要想更好地使用 Linux 环境的文件系统，还需要了解一些其他的文件操作。在这一节里，将介绍其他一些与文件有关的操作。

6.3.1 文件属性的修改

前文已经提到过在 Linux 系统中文件具有比较复杂的属性。此处所说的属性和前面提到的属性并不完全一样。这里所说的文件的属性不但包括了前面所说的文件属性，还包括了很多相关的信息，例如文件名本身、文件所处的位置、文件的长度等等。为了更好地使用文件，有时需要系统调用来修改文件的这些属性。

1. 修改文件权限

不难理解，有时会需要改变文件的权限，以适应具体操作的需要。此时将会用到如下一些系统调用。

(1) 改变文件的所有者

文件所有者是 Linux 系统中的文件所具有的一种属性。可以通过系统调用来改变一个文件的所有者识别号和用户组识别号。相应的系统调用如下。

```
#include <sys/types.h>
#include <unistd.h>
int chown(const char *pathname, uid_t owner, gid_t group);

#include <sys/types.h>
#include <unistd.h>
int fchown(int fd, uid_t owner, gid_t group);
```

- **chown**: 其中的参数 `pathname` 表示文件的绝对路径或相对路径。参数 `owner` 表示新赋予该文件的所有者识别号。参数 `group` 表示新赋予该文件的组识别号。
- **fchown**: 其中的参数 `fd` 是要操作文件的文件描述符。参数 `owner` 和参数 `group` 的含义同系统调用 `chown`。

当调用成功时，`chown` 和 `fchown` 的返回值都为 0；调用失败时，返回值都为 -1，并将 `errno` 置为相应值。

这两个系统调用的功能是一样的，它们的区别在于对文件进行操作时，系统调用 `chown` 使用该文件的路径名而系统调用 `fchown` 使用该文件打开时的文件描述符。相比而言，`fchown` 比 `chown` 更安全一些。



由于涉及到有关文件权限的问题，只有 root 用户才可以使用 `chown` 和 `fchown` 的系统调用来任意改变一个文件的所有者及其所属的组。而普通用户是没有这样的权限的。普通用户只能修改自己所有的文件的组识别号，且只能在其所属的组之中进行选择。

(2) 改变文件的访问权限

对于 Linux 下的文件来说，其属性之一就是不同用户对它具有不同的读、写、执行的权限。要改变这些权限的设定，需要使用如下系统调用。

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod(char *pathname, mode_t mode);
```

```
#include <sys/types.h>
#include <sys/stat.h>
int fchmod(int fd, mode_t mode);
```

参数 pathname 和 fd 的含义同 chown 和 fchown 中的相同。这两个系统调用的关系也与 chown 和 fchown 之间的关系相同。参数 mode 表示调用所要修改的文件的权限设置。文件的权限设置由一组八进制数来表示，称权限位。在系统中设置了相应的标识符来代表这些八进制数。两种表示法都是系统可以接受的，具体设定见表 6-4。使用时按需要将各标识符按位逻辑加得到所需设定。

表 6-4 mode 的相应设置

标识符	对应八进制数	含 义
S_ISUID	04000	设置用户识别号
S_ISGID	02000	设置组识别号
S_SVTX	01000	粘贴位
S_IRUSR	00400	属主用户可读
S_IWUSR	00200	属主用户可写
S_IXUSR	00100	属主用户可执行
S_IRGRP	00040	组可读
S_IWGRP	00020	组可写
S_IXGRP	00010	组可执行
S_IROTH	00004	其他用户可读
S_IWOTH	00002	其他用户可写
S_IXOTH	00001	其他用户可执行

调用成功时，返回值为 0；调用失败时，返回值为 -1，并将 errno 置为相应值。

2. 修改文件的其他属性

Linux 系统也提供了对文件的其他属性进行修改的系统调用。可以按用户的需要修改文件的各种属性。

(1) 重命名

一个普通文件或一个目录文件都可以被重命名。

```
#include <stdio.h>
int rename(const char *oldname, const char *newname);
```

其中的参数 `oldname` 和 `newname` 都是字符串指针。分别代表旧文件名和新文件名。

调用成功时，返回值为 0；调用失败时，返回值为 -1。

`rename` 调用是否能成功，将与 `oldname` 指向普通文件还是目录文件，`newname` 所表示文件是否已存在，若存在是普通文件还是目录文件等情况都有关系，可以从表 6-5 中清楚地看出这些关系。

表 6-5 系统调用 `rename` 的参数选取

<code>oldname</code> 指向	<code>newname</code> 所示文件不存在	<code>newname</code> 指向普通文件	<code>newname</code> 指向目录文件
<code>oldname</code> 指向普通文件	文件被重命名	Newname 被删除，原来名为 <code>oldname</code> 的文件被重命名为 <code>newname</code>	错误
<code>oldname</code> 指向目录文件	文件被重命名	错误	Newname 所指向的目录文件为空目录则该目录文件被删除， <code>oldname</code> 被重命名；否则出错。



对于修改目录文件的情况，有一点要注意。就是 `newname` 不能包含有 `oldname` 的路径前缀。也就是说，不能将一个目录文件重命名为它的子文件。

Tip

当 `newname` 和 `oldname` 指向同一个文件时，`rename` 调用不做任何操作而成功返回。

(2) 修改文件长度

有时需要对文件的大小进行修改。这时将会用到截断文件长度的系统调用。

```
#include <unistd.h>
int truncate(char *pathname, size_t len);

#include <unistd.h>
int ftruncate(int fd, size_t len);
```

其中的参数 `len` 用于指定要将文件截取到的长度。两个调用分别是针对文件路径和文件描述符使用的。

调用成功时，返回值为 0；调用失败时，返回值为 -1。

6.3.2 文件的其他操作

文件是操作系统中一个十分重要的概念，因此，关于文件的操作是多种多样的。在接下来的部分中，将介绍其他一些常用的文件操作。

1. dup 和 dup2 调用

dup 和 dup2 调用都是复制文件描述符。它们的说明如下。

```
#include <unistd.h>
int dup(int fd);
int dup2(int fd, int fd2);
```

这两个调用都将复制文件描述符 `fd`。也就是说，新得到的文件描述符和原来的文件描述符将共同指向一个打开的文件。两个调用的返回值都为新的文件描述符，不同的是，系统调用 `dup` 的返回值是最小的未用文件描述符，而系统调用 `dup2` 的返回值是预先指定的文件描述符 `fd2`。如果文件描述符 `fd2` 正在被使用，则先关闭 `fd2`。如果 `fd2` 同 `fd`，则不关闭该文件正常返回。



通常使用这两个系统调用来重定向一个已打开的文件描述符。

2. stat、fstat 和 lstat 调用

前文已经介绍过，Linux 系统中的所有文件都有一个与之相对应的索引节点，其中包括了文件的相关信息。这些信息被保存在 `stat` 结构体中。可以使用如下系统调用来查看 `stat` 结构体。

```
#include <sys/stat.h>
#include <sys/types.h>
int stat(const char *pathname, stat *sbuf);
int fstat(int fd, struct stat *sbuf);
int lstat(const char *pathname, stat *sbuf);
```

参数 `sbuf` 是指向 `stat` 结构体的指针。

- 函数 `stat` 和 `fstat` 的区别是 `stat` 通过文件路径名访问文件，而 `fstat` 通过文件描述符访问文件。
- 函数 `stat` 和 `lstat` 的区别是当访问一个符号链接时，`stat` 函数追踪到链接的末端的文件，而 `lstat` 函数只是返回链接本身的信息。

以上三个函数调用成功时，返回值均为 0；调用失败时，返回值均为 -1，并将 errno 设置为相应值。

3. fsync 调用

系统调用 fsync 的说明如下。

```
#include <unistd.h>
int fsync(int fd);
#ifndef _POSIX_SYNCHRONIZED_IO
int fdatasync(int fd);
#endif
```

此调用的作用将保存在缓冲区内的要写入文件描述符 fd 的所有数据刷新到要写入的文件中。

调用成功时，返回值为 0；调用失败时，返回值为 -1，并将 errno 设置为相应值。

4. flock 调用

系统调用 flock 的作用是上锁或解锁。其说明如下。

```
#include <sys/types.h>
int flock(int fd, int operation);
```

此调用可将文件描述符 fd 所对应的文件上锁或解锁。参数 operation 用于表示不同的上锁或解锁方式。其可取值 LOCK_SH、LOCK_EX 和 LOCK_UN，分别表示共享锁、独占锁以及解锁。一个进程对一个文件只能有一个独占锁，但可以有多个共享锁。上锁的作用只有在别的进程要对该文件上锁时才显现出来。如果一个程序不试图去锁一个已经被上锁的文件，可以对其进行访问。

调用成功时，返回值为 0；调用失败时，返回值为 -1。

5. fcntl 调用

系统调用 fcntl 用于查看或设置文件的一些相关信息。它的说明如下。

```
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
```

参数 cmd 表示此调用所要执行的操作。其相应的取值和所执行的操作见表 6-6。参数 arg 是可选的，对应于 cmd 参数的某些可取值，用于执行特殊的操作。

表 6-6

cmd 的取值及相应操作

cmd	相 应 操 作
F_DUPFD	复制文件描述符
F_GETFD	获得 close-on-exec 标志
F_SETFD	设置 close-on-exec 标志
F_GETFL	获得 open 调用设置的标志
F_SETFL	修改 open 调用设置的标志
F_GETLK	获得离散的文件锁
F_SETLK	设置获得离散的文件锁，不等待
F_SETLKW	设置获得离散的文件锁，必要时等待
F_GETOWN	检索将收到 SIGIO 和 SIGURG 信号的进程 id 或进程组号
F_SETOWN	设置进程 id 或进程组号

调用成功时，返回值为 0；调用失败时，返回值为 -1。



不难发现，fcntl 调用能完成的大部分操作都可以用其他的调用来完成。

6. select 调用

select 是一个十分复杂的系统调用。到目前为止所处理的输入输出操作，都是同一时刻对一个文件描述符进行的。对于要处理多个文件描述符的进程，比如网络服务器应用程序，需要同时监控多个文件描述符。这样的问题就需要使用 select 函数来解决。

select 函数的参数中包括三个文件描述符集。select 函数会对其进行监控。一旦与用户进程相关的事件发生，则函数返回发生事件的文件描述符，以使程序做出相应处理。因此使用 select 函数是处理多个文件描述符的有效手段。

正是由于 select 在网络程序中的广泛应用，函数的使用将在网络编程一章中详细讲解。

6.4 特殊文件的操作

在 Linux 系统中，除去普通文件外，还有其他四类比较特殊的文件。它们是目录文件、

链接文件、管道文件和设备文件。在这一节里，将分别介绍目录文件、链接文件、管道文件和设备文件的相关操作。

6.4.1 目录文件的操作

目录文件是 Linux 中一类比较特殊的文件。它对构成 Linux 系统的整个文件系统结构非常重要。系统提供了一些对目录文件进行操作的调用。

1. 目录文件的创建和删除

(1) 创建目录

可以在 Linux 系统中创建一个目录文件。使用的系统调用如下。

```
#include <sys/types.h>
#include <sys/stat.h>
int mkdir(const char *pathname, mode_t mode);
```

参数 `pathname` 表示新创建的目录文件名。参数 `mode` 表示其权限设置，`mode` 的值由当前的 `umask` 值以通常的方式加以确定。此目录的所用者为调用 `mkdir` 创建它的有效用户识别号。

调用成功时，返回值为 0；调用失败时，返回值为 -1。

(2) 删除目录

当一个目录为空目录时，可以通过系统调用将之删除。

```
#include <unistd.h>
int rmdir(const char *pathname);
```

调用成功时，返回值为 0；调用失败时，返回值为 -1。



要注意当 `pathname` 所指目录不为空目录时，该系统调用是不能成功的。

2. 文件的打开和关闭

当访问一个目录文件时，同普通文件一样，需要将其打开。在访问结束后，又需要将其关闭。这里所使用的相应系统调用如下。

(1) 目录文件的打开

打开目录文件的系统调用为：

```
#include <sys/types.h>
```

```
#include <direct.h>
DIR *opendir(const char *pathname);
```

此调用的唯一参数 `pathname` 是要打开目录的路径名。

调用的返回值为 `DIR` 类型，是用于指向目录文件的结构指针。调用成功时，返回值为一个目录指针；调用失败时，返回值为 `NULL`。

(2) 目录文件的关闭

关闭一个已打开目录文件的系统调用为：

```
#include <sys/type.h>
#include <direct.h>
int closedir(DIR *dp);
```

其参数是要关闭的目录文件的指针，该指针由 `opendir` 调用时返回。

调用成功时，返回值为 0；调用失败时，返回值为 -1。

3. 目录文件的读取

Linux 系统提供了读取一个目录文件内容的系统调用。

```
#include <sys/type.h>
#include <direct.h>
struct direct *readdir(DIR *dp);
```

此调用用于访问目录指针为 `dp` 的目录文件。其返回值为指向 `dirent` 结构体指针。`dirent` 结构体的定义如下。

```
struct dirent
{
    ino_t d_ino;
    char d_name[NAME_MAX+1];
};
```

其中 `d_ino` 用于表示该目录的节点号。`D_name` 用于存放此目录链接的文件名。

当目录中没有更多链接时，返回 0。

6.4.2 链接文件的操作

链接文件是 Linux 系统中的另一种特殊文件。它实际上是指向一个现实存在的文件的链

接。链接文件又分为硬链接文件和符号链接文件。下面，就这两种情况分别给出相关的系统调用。

1. 硬链接操作

(1) 创建链接

当需要对一个已经存在的文件建立新的链接时。需要使用如下的系统调用。

```
#include <unistd.h>
int link(char *pathname1, char *pathname2);
```

参数 `pathname1` 表示已经存在的文件。参数 `pathname2` 则是要新建的链接。`pathname1` 和 `pathname2` 所指向的路径名应当在同一个文件系统中。

调用成功时，返回值为 0；调用失败时，返回值为 -1。



只有超级用户 `root` 才能创建一个指向目录文件的新的链接文件。

(2) 移去链接

要移去一个存在的链接，可以通过系统调用 `unlink` 实现。

```
#include <unistd.h>
int unlink(char *pathname);
```

参数 `pathname` 表示要移去链接文件的路径名。当调用 `unlink` 移去指定的目录文件的链接时，还会将目录的索引节点中的链接计数器减 1。如果链接计数器减到 0 时，索引节点和文件数据块全部释放给系统。

此外还要注意，要移去一个文件的链接时，必须有对包含该文件的目录的写权限和执行权限。而当这个目录是粘位的时，进行该操作的用户还必须是文件所有者、文件所在目录所有者或超级用户之一。

【程序 6_6】

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(void)
{
    int stat(const char *pathname, stat *sbuf);
```

```
if(open("tempfile", O_RDWR)<0)
{
    printf("open error\n");
    exit(1);
}

if(unlink("tempfile")<0
{
    printf("unlink error")
    exit(1);
}
printf("file unlinked.\n");

sleep(15);
printf("done\n");

exit(0);
}
```

2. 符号链接

符号链接是指向某一文件的指针，与硬链接直接指向文件的索引节点不同。符号链接没有硬链接那么强的限制，例如不只是超级用户，而是任何一个用户都可以创建一个指向目录文件的符号链接，链接文件和原文件不需要在同一个文件系统中等等。

要创建一个符号链接所使用的系统调用为：

```
#include <unistd.h>
int symlink(const char *actualpath, const char *sympath);
```

参数 `actualpath` 为真实存在的文件。参数 `sympath` 为新创建的指向 `actualpath` 的符号链接。
调用成功时，返回值为 0；调用失败时，返回值为 -1。

可以通过调用 `readlink` 来打开一个链接并获得它的名字。

```
#include <unistd.h>
int readlink(const char *pathname, char *buf, int bufsize);
```

参数 `pathname` 为所要查看的链接。参数 `buf` 为字符串指针，获得的相关信息将存放在 `buf`

所指向的缓冲区。参数 `bufsize` 表示该缓冲区的大小。

调用成功时，返回值为实际写入缓冲区的字节数；调用失败时，返回值为 0。

【程序 6_7】

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int main(argc,argv[ ])
{
    char symname[ ]={"mysym"};
    char buf[80];

    if(argc<2)
        return 0;

    if(symlink(argv, symname)==-1)
    {
        printf("symlink error!\n");
        exit(1);
    }

    if(!readlink(symname, buf, sizeof(buf)))
    {
        printf("readlink error!\n");
        exit(1);
    }
    printf("name of symbol link: %s.\n", buf);

    exit(0);
}
```

【程序 6_7】创建了一个与给定文件链接的符号链接，并将符号链接的名字显示出来。

6.4.3 管道文件的操作

管道文件也是 Linux 中一种很特殊的文件。主要用于不同进程间的数据和信息传递。当不同进程间需要进行数据或信息传递时，可以通过管道文件，一个进程将要传递的数据或信

息写入管道的一端，另一进程则从管道的另一端取得所需的数据或信息。

管道的创建可以通过系统调用 `pipe` 来实现。

```
#include <unistd.h>
int pipe(int filedes[2]);
```

参数 `filedes` 是一个含有两个元素的一维数组。当调用 `pipe` 成功创建管道后，将返回两个文件描述符，分别为管道的两端，它们将被存放在 `filedes` 中。

调用成功时，返回值为 0；调用失败时，返回值为 -1，并将 `errno` 置为相应值。



Tip 通常系统调用 `pipe` 被用来与 `fork`、`dup2` 及 `execve` 等函数配合使用为被重定向 I/O 的其他程序创建管道。

关于管道文件的具体使用将会在进程间通信一章中详细介绍。

6.4.4 设备文件

在 Linux 系统中，有一类非常特别的文件——设备文件。正是由于它的存在，使得用户可以十分方便的访问外部设备。Linux 系统为外部设备提供一种标准接口，将外部设备视为一种特殊的文件。可以像访问文件一样访问一个外部设备。这就使得 Linux 系统可以很方便的适应不断发展的外部设备。

在实际操作中，几乎总是不可避免地要用到设备文件。由于在 Linux 系统中，所有的外部设备都被看作是目录 `/dev` 下的一个文件，所以前面讲的各种关于文件的系统调用都可以在外部设备上使用。因而可以很方便地使用基于文件描述符的 I/O 操作实现外部设备的 I/O 操作。

但也要注意一些特殊的外部设备。它们在某些方面具有其自身的特殊性。

例如，对于像磁带这样的外存，只可以顺序地对它进行读写访问。而用于实现随机读写的系统调用 `lseek` 对磁带来说是无效的。同时当调用系统调用 `close` 时，将导致磁带的回绕。

又如，串口通常包括拨出和拨入设备，在接收输入状态时，如果没有活动的输入，该串口仍可以输出打开。

6.5 小结与练习

6.5.1 小结

这一章介绍了有关文件和文件系统的概念。学习了文件的基本输入输出操作——基于文

件描述符的输入输出操作。此外，还介绍了其他与文件操作相关的函数，包括获取文件信息、修改文件的属性、重命名文件等操作。同时了解了关于 Linux 平台下的特殊文件（目录文件、链接文件、管道文件和设备文件）的有关知识。

熟悉 Linux 平台下的文件和文件系统以及相关的调用对更好地在 Linux 平台下编写 C 程序将大有帮助。

6.5.2 习题与思考

编写四个函数，其调用参数与返回值同函数 open、write、read 和 close。可以处理操作时发生错误的问题。

第7章

输入输出——基于流的操作

流简介

基于流的 I/O 操作

临时文件

小结与练习

前面的一章介绍了执行输入输出操作的一种方法——基于文件描述符的输入输出操作。在这一章里，将介绍另一种输入输出方法——基于流的输入输出操作。

与基于文件描述符的输入输出相比，基于流的输入输出更加简单、方便，因而在 C 程序的编写中，被广泛地使用。通过本章的学习，读者将对基于流的输入输出操作有所了解。

7.1 流 简 介

本章所介绍的 I/O 操作是基于“流（stream）”的概念进行的。在 Linux 系统中，文件和设备都被看做是数据流。进行操作之前，必须先将流打开。当使用相应的函数调用打开文件或设备时，相关的数据流就可以被访问了，也就是说，可以对其进行输入输出操作了。

要对流进行操作，需要使用由 stdio 库所提供的函数以及相应的文件指针 FILE 来实现。

基于流的 I/O 操作与基于文件描述符的 I/O 操作十分类似，整个过程简要介绍如下。

对流进行操作的第一步是将其打开。可以通过调用库函数 fopen（）打开一个流，库函数 fopen（）的返回值为一个 FILE 结构指针，此结构中含对所打开流进行操作所需的全部信息。包括所开文件的描述符，为流准备的缓冲区的指针、大小等。实际使用时，并不需要对 FILE 结构的内容有所了解，只需要在调用库函数对流进行操作时会使用它。



当执行程序时，有三个流会自动打开，它们是标准输入、标准输出和标准错误输出。与之相对应的 FILE 结构指针为 stdin、stdout 和 stderr。

当一个流被打开以后，就可以对其执行 I/O 操作了。具体来说，就是使打开流时返回的 FILE 指针来表示所要操作的流，调用相应的库函数实现所需的 I/O 操作。

当对一个流的操作完成后，需要执行清空缓冲区、保存数据等操作，将流关闭。这些工作可以通过调用库函数 fclose（）来完成。



如果不关闭流，可能造成数据的丢失。



和自动打开一样，标准输入、标准输出和标准错误输出是自动关闭的。

使用基于流的 I/O 操作实现输入输出时，库函数可以帮助程序员处理如分配缓冲区地址、设置缓冲区大小、保存数据等工作。避免了程序员在这些琐事上过多花费精力。因而可使程序的编写更加简单、方便。同时由于基于流的 I/O 操作所用的库函数是用标准的 C 语言编写的，不但可用于 Linux 系统，也可适用其他操作系统，大大增加了程序的可移植性。

首先来看一个基于流的 I/O 操作的具体例子，以便对其有一个总体的概念。

【程序 7_1】

```
#include<stdio.h>

main()
{
    char key;
    FILE *stream;           /*流的指针*/

    printf("Please input a letter.\n");
    key=getchar();          /*读入一个字符*/

    if (stream=fopen("test","r")==(FILE*)0)  /*将流与一个文件联系起来,
并判断是否成功*/
    {
        fprintf(stderr, "Error opening file.\n");
        exit(1);
    }

    /*将缓冲区设定为行缓冲,并判断是否成功*/
    if (setlinebuf(stream)) !=0
    {
        fprintf(stderr, "Error setlinebuffer .\n");
        exit(1);
    }

    /*将对应的缓冲区清空,并判断操作是否成功*/
    if(fpurge(stream)) ==EOF
    {
        fprintf(stderr, "Error flush stream.\n");
        exit(1);
    }

    /*对流的读写操作*/
    fprintf(stream, "The letter that you input is %c.\n",key);

    /*在程序结束前关闭流,并判断操作是否成功*/
    if(fclose(stream)) ==EOF
```

```
{  
    fprintf(stderr, "Error closing file,\n");  
    exit(1);  
}  
exit(0);  
}
```

【程序 7_1】展示了基于流的 I/O 操作的实现过程，包括打开一个流、设置缓冲区类型、清空缓冲区、输入操作、最后关闭流的全过程。这里使用的库函数，是标准 C 语言的 stdio 函数库中提供的，将在接下来的讨论中详细说明。

7.2 基于流的 I/O 操作

基于流的 I/O 操作是实现 I/O 操作的又一手段。基于流的 I/O 操作是通过一个指向文件的 FILE 指针来实现对文件的访问的。在 FILE 结构指针中包含有对流进行操作所需的各种信息，包括所打开文件的文件描述符、为进行对流的 I/O 操作而开辟的缓冲区的指针、大小以及出错标志等内容。通过调用库函数 fopen 打开一个流时，就建立了这个流与一个 FILE 结构指针的联系，然后通过它对流进行操作。操作结束后，调用库函数 fclose 关闭这个流。

7.2.1 流的打开和关闭

要对流进行操作，首先要了解基于流的 I/O 操作中的最基本的库函数——打开和关闭流的库函数。

1. 流的打开

要对一个流进行操作之前，首先要将其打开，也就是建立某一流同某个特定的设备或文件之间的关联。只有这样，才能对这个流进行各种操作。打开一个流的操作可以由以下库函数的调用来实现：

```
#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);
```

下面先对这三个函数进行简单的说明。

- fopen 函数用于打开一个特定的文件。参数 pathname 表示文件名，参数 type 表示了

流的打开模式。具体参数选取及对应模式见表 7-1。

- `freopen` 函数用于在一个特定的流上打开一个特定的文件。参数 `pathname` 和 `type` 的设定同函数 `fopen`。参数 `fp` 是一个已有的 `FILE` 结构指针。当调用此函数时，它首先关闭 `fp` 所指向的流，然后重新用这个 `FILE` 结构指针 `fp` 打开 `pathname` 所代表的文件。此函数一般用来对标准输入、标准输出、标准错误输出等预定义的流进行重定向。
- `fdopen` 函数用于将一个流与某一个已打开的特定文件相对应。参数 `filedes` 表示此文件的描述符，参数 `type` 同前。需要注意的是，只有当 `type` 所定义的模式与 `filedes` 所表示的文件的打开模式相同时，调用才能成功。此函数多用于建立某一流与无法用基于流的 I/O 操作函数打开的文件之间的关联，如打开管道文件或网络通信管道等。此处涉及到有关文件的具体细节将在下一章详细讲述。

表 7-1 Type 可选参数及其对应模式

Type	操作文件类型	是否新建文件	是否清空原文件	可读	可写	读写开始位置
“r”	文本文件	No	No	Yes	No	文件开头
“r+”	文本文件	Yes	No	Yes	Yes	文件开头
“w”	文本文件	Yes	Yes	No	Yes	文件开头
“w+”	文本文件	Yes	Yes	Yes	Yes	文件开头
“a”	文本文件	No	Yes	No	Yes	文件结尾
“a+”	文本文件	No	Yes	Yes	Yes	文件结尾
“rb”	二进制文件	No	No	Yes	No	文件开头
“r+b”或“rb+”	二进制文件	Yes	No	Yes	Yes	文件开头
“wb”	二进制文件	Yes	Yes	No	Yes	文件开头
“w+b”或“wb+”	二进制文件	Yes	Yes	Yes	Yes	文件开头
“ab”	二进制文件	No	Yes	No	Yes	文件结尾
“a+b”或“ab+”	二进制文件	No	Yes	Yes	Yes	文件结尾

如果调用成功的话，上述系统调用的返回值是指向所打开流的 `FILE` 结构指针；否则返回一个空指针。

当程序成功地完成了一个流的打开操作之后，它就和一个 `FILE` 结构指针联系起来了。随之进行的各种操作都是通过引用此结构指针进行库函数的调用来实现的。

2. 流的关闭

在所需的操作完成以后，必须将流关闭。关闭流的库函数如下。

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

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

函数的唯一参数 `stream` 是一个 `FILE` 结构指针，它指向用户要关闭的流。

调用成功时，返回值为 0；调用失败时，返回值为 -1，并将 `errno` 置为相应值。



如果在程序结束前没有执行关闭流的操作，有可能会造成写入的数据停留在缓冲区里而没有保存到文件中，造成数据丢失。

7.2.2 缓冲区的操作

在进行基于流的 I/O 操作时，缓冲区的使用将是不可或缺的。使用缓冲区时将用到如下操作。

1. 设置缓冲区属性

缓冲区具有自己的属性，其属性值包括缓冲区的类型和缓冲区的大小。当调用库函数 `fopen` 打开一个流时，就开辟了所需的缓冲区。系统通常都会赋予其一个缺省的属性值。这些缺省值是系统默认的、使用频率较高的值。实际使用时，也可以根据自己的需要来设定缓冲区的属性值。缓冲区属性值的设定可以通过调用如下的库函数来实现。

```
#include <stdio.h>
int setbuf(FILE *fp,char *buf);
int setbuffer(FILE *fp,char *buf,size_t size);
int setlinebuf(FILE *fp);
int setvbuf(FILE *fp,char *buf,int mode,size_t size);
```

这四个函数都用于对流的属性进行设定，它们都涉及到同一个参数 `fp`，这是一个 `FILE` 结构指针，应指向一个已经打开的流。也就是说，在调用上述库函数时，流必须是已打开的。同时，各个函数具有不同的功能和特点。

- `setbuf` 函数用于将缓冲区设置为全缓冲或无缓冲。参数 `buf` 为指向缓冲区的指针。当 `buf` 指向一个真实的缓冲区地址时，此函数调用将缓冲区设定为全缓冲，其大小由预定义常数 `BUFSIZ` 指定；当 `buf` 为 `NULL` 时，则设定为无缓冲。所以此函数一般可当作激活或禁止缓冲区的开关。
- `setbuffer` 函数的功能和使用方法和函数 `setbuf` 很相似，其区别是可由程序员自行指定缓冲区的大小，由参数 `size` 指定。
- `setlinebuf` 函数专用于将缓冲区设定为行缓冲。
- `setvbuf` 函数比较灵活，它可很方便地设置缓冲区的属性。是四个函数中最基本的，前面三个函数的功能都可以用此函数来实现。参数 `fp`、`buf` 和 `size` 的含义如前述三

个函数。参数 mode 用于指定缓冲区的类型，其可取的值为 _IOFBF（全缓冲类型）、_IOLBT（行缓冲类型）和 _IONBF（无缓冲类型）。



当 mode 设定为 _IOLBT 时，size 的值是无效的，缓冲区的大小为系统预定义大小。当 mode 设定为 _IONBF 时 buf 的值和 size 的值都是无效的。

上述四个库函数调用成功时，返回值为 0；调用失败时，返回值为非 0 值。



Tip 最好是在将流打开但还未对流执行其他操作时设定流的属性。因为对流进行的各种操作都是和缓冲区的属性紧密相关的，改变缓冲区的属性会对所执行的操作产生影响。

2. 缓冲区的清洗

所谓缓冲区的清洗，是指将 I/O 操作写入缓冲区中的内容清空。这种清空可以是将流的内容完全丢掉，也可以是将其保存到文件中。相应的库函数如下。

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

- fflush 函数用于将缓冲区中尚未写入文件的数据强制性地保存到文件中。
- fpurge 函数用于将缓冲区内的数据完全清除。

调用成功时，返回值为 0；调用失败时，返回值为 EOF。

7.2.3 直接输入输出

基于流的 I/O 操作包括直接输入输出、格式化输入输出以及基于字符和行的输入输出。首先讨论直接输入输出操作。

直接输入输出操作是以记录为单位进行读写。相应的库函数如下：

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

- fread 函数用于执行直接输出操作。参数 ptr 是指向读取数据的缓冲区的指针。参数 size 是读取记录的大小。参数 nmemb 是所读记录的个数。参数 fp 是指向要读取的流的 FILE 结构指针。
- fwrite 函数用于执行直接输入操作。参数 ptr 是指向存放要输入数据的缓冲区的指针。

参数 `size` 是写入记录的大小。参数 `nmemb` 是所写记录的个数。参数 `fp` 是指向要写入数据的流的 `FILE` 结构指针。

调用函数 `fread` 和 `fwrite` 的返回值是实际读取或写入的记录数目。这个返回值应当同 `nmemb` 的预设值相同。只有当到达文件的末尾（只有在读操作时有此情况）或出现读写错误时，会造成返回值比设定的 `nmemb` 值小，甚至是负值。这时系统内的文件结束标志或文件错误标志会被置为相应的值。这时需要检查失败的原因，需调用如下库函数。

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

- `feof` 函数用于检测是否读到文件的末尾。当访问正常结束时，函数的返回值为 0；如果访问到文件的末尾，则返回值非 0。只有执行读操作时才对文件结束标志进行操作。
- `ferror` 函数用于检测是否出现了读写错误。当访问正常结束时，函数的返回值为 0；如果访问到文件的末尾，则返回值非 0。此时应检测 `errno` 以确定所发生的错误。



`errno` 的值是在错误发生时由读写函数本身所设置的，而不是由 `ferror` 函数设置的。

当发生了超过文件末尾读取和读写错误时，文件结束标志和错误标志将发生相应的改变，可以调用库函数 `clearerr` 来重置结束标志和错误标志。

```
#include <stdio.h>
void clearerr(FILE *fp);
```

先来看进行直接读写操作的例子。

【程序 7_2】

```
#include <sys/types.h>
#include <stdio.h>

#define BUF_SIZE 1024

int main(void)
{
    char buf[BUF_SIZE];
    FILE *source, *backup;
```

```
if(!source=fopen("source.dat", "r"))
{
    printf("Error in opening file.\n");
    exit(1);
}

if(!backup=fopen("backup.dat", "w"))
{
    printf("Error in creating file.\n");
    exit(1);
}

while(fread(buf, sizeof(buf), 1, source)==1)
{
    if(fwrite(buf, sizeof(buf), 1, backup));
    {
        printf("Error in writing file.\n");
        exit(1);
    }
}

if(ferror(source)==0)
{
    printf("Error in reading file.\n");
    exit(1);
}

if(!fclose(source))
{
    printf("Error in close file.\n");
    exit(1);
}

if(fclose(backup))
{
    printf("Error in close file.\n");
    exit(1);
}
```

这是一个简单的备份文件的程序。文件 `sourcr.dat` 中的数据被拷贝到文件 `backup.dat` 中。

7.2.4 格式化输入输出

和基于系统调用的 I/O 相比，基于流的 I/O 的一个最大的特点就是它可以实现格式化的输入输出。前面经常使用的 `print` 函数就是一个用于格式化输入输出的库函数。下面将具体讨论格式化输入输出的函数调用。

1. 格式输出

`stdio` 库提供的格式输出的库函数如下。

```
#include <stdio.h>
int printf(const char *format, ...);
int fprintf(FILE *fp, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);
```

- `printf` 函数用于向标准输出流中输出数据。参数 `format` 是一个字符串，用于描述输出格式。
- `fprintf` 函数用于向指定的流中输出数据。参数 `fp` 是一个 `FILE` 结构指针，指向要输出的流。
- `sprintf` 函数不是用于向一个流输出数据的，它的作用是向一个字符串输出数据。参数 `str` 是字符串指针，指向要进行输出的缓冲区。
- `snprintf` 函数的作用同 `sprintf` 函数。不同的是 `sprintf` 函数不能对缓冲区进行处理，使用时可能会出现缓冲区溢出而导致程序崩溃。而 `snprintf` 函数可以处理缓冲区。参数 `size` 用于设定缓冲区的大小。

函数 `printf` 和 `fprintf` 的返回值为实际输出的字符数。函数 `sprintf` 和 `snprintf` 的返回值为实际输入缓冲区的字符数。



在一些其他操作系统中的 `stdio` 库中，不提供 `snprintf` 函数。

2. 格式输入

`stdio` 库提供的格式输入的库函数如下。

```
#include <stdio.h>
int scanf(const char *format, ...);
int fscanf(FILE *fp, const char *format, ...);
```

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

这三个格式输入函数的说明同格式输出的相应函数十分类似，简要说明如下。

- `scanf` 函数用于从标准输入流中输入数据。
- `fscanf` 函数用于从一个指定的流中输入数据。
- `sscanf` 函数用于从一个字符串输入数据。

函数 `scanf` 和 `fscanf` 的返回值为实际输入的字符数。函数 `sscanf` 的返回值为实际输入缓冲区的字符数。



由于从缓冲区读入数据不会造成缓冲区溢出的情况，`stdio` 库中没有与 `snprintf` 相应的格式输入函数。

在前文中已经接触过格式输入输出库函数，再来看一个使用格式化输出输出的例子。

【程序 7_3】

```
#include <sys/types.h>
#include <stdio.h>

int main(void)
{
    float value, total[10];
    int count, label;
    FILE *fp;

    for(count=0; count<10; count++)
        total[count]=0;

    if(!fp=fopen("test.dat", "r"))
    {
        printf("Error in open file.\n");
        exit(1);
    }

    while(fscanf(fp, "%d%g", &label, &value))
    {
        total[label]+=value;
    }
}
```

```
for(count=0; count<10; count++)
{
    printf("%d: %f\n", count, total[count]);
    return 0;
}
```

【程序 7_3】是将文件 test.dat 中的数据读取并进行一定的分类后相加。test.dat 文件由若干行组成，每行有两个数据。第一个是整数，表示类型，取值为 0 到 9；第二个是浮点数。程序将同一类型的浮点数相加，再显示到屏幕上。



上面提到的格式输入输出的库函数都有一个共同的特点，就是在其参数列表中含有以“...”表示的可变参数。也就是说，这些函数的参数的个数和类型都是不定的。在调用上述函数时，系统可以接受不同个数不同类型的参数。这是为了使格式输入输出具有更好地适应性，可以用于输入输出各种数据。

除了这些函数调用之外，stdarg 库还提供了另一组用于格式输入输出的库函数用于解决参数不定的问题。

(1) 格式输出

```
#include <stdarg.h>
int vprintf(const char *format, va_list ap);
int vfprintf(FILE *fp, const char *format, va_list ap);
int vsprintf(char *str, const char *format, va_list ap);
int vsnprintf(char *str, size_t size, const char *format, va_list ap);
```

(2) 格式输入

```
#include <stdarg.h>
int vscanf(const char *format, va_list ap);
int vfscanf(FILE *fp, const char *format, va_list ap);
int vsscanf(char *str, const char *format, va_list ap);
```

这组函数的作用同上面的 stdio 库提供的格式输入输出函数相同。不同的是增加了参数 ap。参数 ap 是一个 va_list 类型变量。

【程序 7_4】

```
#include <sys/types.h>
#include <stdarg.h>
```

```

int main(void)
{
    float value, total[10];
    int count, label;
    va_list ap_read, ap_write;
    FILE *fp;

    for(count=0; count<10; count++)
        total[count]=0;

    if(!fp=fopen("test.dat", "r"))
    {
        printf("Error in open file.\n");
        exit(1);
    }

    ap_read=("&label", "&value");
    while(!fscanf(fp, "%d%g", ap_read))
    {
        total[label]++=value;
    }

    ap_write=(" count", "total[count]");
    for(count=0; count<10; count++)
    {
        printf("%d: %f\n", ap_write);
    }
}

```

此程序具有与【程序 7_3】一样的功能。

7.2.5 基于字符和行的输入输出

流的 I/O 操作中，还有一类是基于字符和行的输入输出操作。这样的库函数有很多，大体上分为两种，一种是基于字符的，用于处理字符；一种是基于行的，用于处理字符串。

(1) 字符的输入输出

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

```
int fgetc(FILE *fp);
int getc(FILE *fp);
int getchar(void);
int fputc(int c, FILE *fp);
int putc(int c, FILE *fp);
int putchar(int c);
int ungetc(int c, FILE *fp);
```

前三个函数 `fgetc`、`getc` 和 `getchar` 用于执行字符的输入。之后的三个函数 `fputc`、`putc` 和 `putchar` 用于执行字符的输出。最后一个函数 `ungetc` 用于把从流中读入的字符再推回。

- 三个执行字符输入操作的函数。函数 `fgetc` 和 `getc` 都是用于向由 `fp` 指定的流中读入字符的。它们的区别在于，调用 `getc` 函数时所用的参数 `fp` 不能是有副作用的表达式，而 `fgetc` 函数则可以。也就是说，`getc` 可以被当作宏来调用，而 `fgetc` 只能作为函数来调用。因此，调用 `getc` 比调用 `fgetc` 耗费的时间要少。函数 `getchar` 只用来从标准输入流中输入数据。其作用相当于调用以 `stdin` 为参数的 `getc` 函数。
- 三个执行字符输出函数 `fputc`、`putc` 和 `putchar` 之间的关系同三个执行字符输入的函数类似。函数 `putc` 可作为宏来使用，其参数不能是具有副作用的表达式。函数 `fputc` 只能作为函数来使用，函数 `putchar` 相当于参数 `fp` 取值为 `stdout` 的 `putc` 函数调用。这三个函数中的参数 `c` 用于表示要输出的字符。
- 函数 `ungetc` 用于将读入的字符再推回流中。被推回的字符将在下一次读入操作时被读入，也可以在文件末尾推回某个字符，该字符将在下一次读入操作时被读入。参数 `c` 用于表示要推回的字符，被推回的字符并不要求是上一次读入的。文件的结束符 `EOF` 是不可被推回的。原则上来说，可以执行任意次的推回操作推回多个字符，但并不提倡这样做。

【程序 7_5】

```
#include <stdio.h>

int character;
char *token;

void lexicure(void);

int main(void)
{
    lexicure();
    while(character)
    {
```

```
    fprintf(stdout, token);
    lexicure();
}

exit(0);
}

void lexicure(void)
{
    character=getchar();
    switch(character)
    {
        case 'A'..'Z':
            while((('A'<=character)&&('Z'>=character))|| (('0'<=character)&&('9'>=character)))
            {
                token+=character;
                character=getchar();
            }
            if(character!=EOF)
                ungetc(character, stdin);
            else
                ungetc(0, stdin);
            break;
        case '0'.. '9':
            while(('0'<=character)&&('9'>=character))
            {
                token+=character;
                character=getchar();
            }
            if(character!=EOF)
                ungetc(character, stdin);
            else
                ungetc(0, stdin);
            break;
    }
}
```

【程序 7_5】的功能是读入键盘输入的数据，从中挑选出由字符和数字组成的字符串或整数。

(2) 行的输入输出

```
#include <stdio.h>
char *fgets(char *str, int size, FILE *fp);
char *gets(char *str);
int fputs(const char *str, FILE *fp);
int puts(const char *str);
```

前两个函数 `fgets` 和 `gets` 用于执行行的输入。之后的两个函数 `fputs` 和 `puts` 用于执行行的输出。参数 `str` 是字符串指针，指向存放输入或输出字符串的缓冲区。

- 函数 `fgets` 和 `gets` 用于行输入。函数 `fgets` 可以从任意流中读入数据，该流由参数 `fp` 指定。并且调用 `fgets` 函数时要明确所读入的字符数。参数 `size` 用于表示要读入的字符数。读入数据后在行的末尾加空字符表示结束，因此实际读入的最大字符数为 `size-1`。当一行数据字符数大于 `size-1` 时，多余的部分被截去，末尾加空字符表示结束。剩余的部分将在下一次读操作时被读入。函数 `gets` 则只用于从标准输入流中读入数据。由于 `gets` 函数不能处理缓冲区大小的问题，在一行字符数大于缓冲区所能容纳的字符数时，将会造成溢出。因此不提倡使用该函数。
- 函数 `fputs` 和 `puts` 用于行输出。函数 `fputs` 用于输出 `str` 指向的由空字符表示行结束的缓冲区的数据，空字符不予输出，该函数可用于向指定的流中输出数据。函数 `puts` 只能用于向标准输出流中输出数据。同样以空字符为结束。但这时将输出一个换行符。

基于行的输入输出使用起来很方便。下面来看一个使用这些操作的例子。

【程序 7_6】

```
#include <sys/types.h>
#include <stdio.h>

#define BUF_SIZE 1024

int main(void)
{
    char buf[BUF_SIZE];
    FILE *source, *backup;

    if(!source=fopen("source.dat", "r"))
    {
```

```
    printf("Error in opening file.\n");
    exit(1);
}

if(!backup=fopen("backup.dat", "w"))
{
    printf("Error in creating file.\n");
    exit(1);
}

while(fgets(buf, sizeof(buf), source))
{
    if(fputs((buf, backup));
    {
        printf("Error in writing file.\n");
        exit(1);
    }
}

if(ferror(source)==0)
{
    printf("Error in reading file.\n");
    exit(1);
}

if(!fclose(source))
{
    printf("Error in close file.\n");
    exit(1);
}

if(!fclose(backup))
{
    printf("Error in close file.\n");
    exit(1);
}
}
```

此程序与【程序7_2】有相同的作用。都是将一个文件中的数据拷贝到另一个文件中。

7.3 临时文件

在编写应用程序的时候，还可能会用到临时文件。临时文件是指那些在程序运行期间存在并使用，而当程序运行完毕之后就可以删除的文件。临时文件的各种操作都是基于流进行的。

标准输入输出库 `stdio` 库提供了用于创建临时文件的库函数。

```
#include <stdio.h>
char *tmpnam(char *str);
FILE *tmpfile(void);
```

- 函数 `tmpnam` 并不用于创建一个临时文件，它的作用只是生成一个有效的文件名。这个文件名不同于任何已经存在的文件的文件名。参数 `str` 是一个字符型指针，指向用于保存新生成的文件名的缓冲区。这一缓冲区的长度至少为 `L_tmpnam`，该常数由 `stdio` 库定义。由于不能处理缓冲区的大小，可能会发生溢出错误。调用成功时返回值就是该缓冲区指针。当 `str` 为 `NULL` 时，生成的文件名存放在一个静态区域中，这一静态区域处于内部缓冲区，函数的返回值为指向此缓冲区的指针。如果再次调用这一函数，将改变这一区域中的文件名。因此，如果要保存文件名的话，用户必须另开辟相应的缓冲区存放该文件名。调用失败时，返回值为空指针。



函数 `tmpnam` 的调用次数是有限制的。其最大次数不能超过常数 `TMP_MAX`。此常数由 `stdio` 库提供定义。

- 函数 `tmpfile` 用于打开一个临时文件。此函数创建的临时文件为二进制文件。这个文件会在程序结束时自动删除。调用 `tmpfile` 函数生成临时文件时，它首先会调用函数 `tmpnam` 生成临时文件的文件名，然后创建这一文件。最后系统调用 `unlink`。在程序结束时，文件将会被删除。

下面是创建临时文件的具体例子。

【程序 7_7】

```
#include <stdio.h>

int main(void)
{
    char line[MAXLINE];
```

```

FILE *fp;

if(!fp=fopen(tmpnam(name),"r+"))
{
    printf("tmpfile error");
    exit(1);
}

if(!fputs("This is a temp file.\n", fp))
{
    printf("write error.\n");
    exit(1);
}

rewind(fp);
if(!fgets(line, sizeof(line), fp))
{
    printf("read error.\n");
    exit(1);
}

printf("%s\n", line);

exit(0);
}

```

【程序 7_7】是一个使用临时文件的例子。在程序中创建了一个临时文件并向其中写入一行数据。然后从临时文件中读出这行数据显示到屏幕上。

也可以通过 tmpnam 函数和其他一些函数来实现这一操作。

【程序 7_8】

```

#include <stdio.h>

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

```

```
if(!fp1=fopen(tmpnam(name),"r+")
{
    printf("tempfile error");
    exit(1);
}
unlink(name);

if(!fputs("This is a temp file.\n", fp))
{
    printf("write error.\n");
    exit(1);
}

rewind(fp);
if(!fgets(line, sizeof(line), fp))
{
    printf("read error.\n");
    exit(1);
}

printf("%s\n", line);

exit(0);
}
```

【程序 7_8】与【程序 7_7】的功能是相同的。只不过【程序 7_8】是由用户自己创建了临时文件，而【程序 7_7】是通过调用 tmpfile 实现的。

由函数 tmpnam 生成的文件名不能制定文件的路径名和前缀。通常生成的临时文件存放于 /tmp 或 /var/tmp 目录下，这些目录是不安全的。如果希望设置临时文件所在目录，可以使用函数 tempnam，它是 tmpnam 函数的一个变种，说明如下：

```
#include <stdio.h>
char *tempnam(const char *directory, const char *prefix);
```

其中参数 directory 用于指定文件的路径名。文件路径名的选取遵循以下规律：如果已定义了环境变量 TMPDIR，则使用环境变量 TMPDIR 做路径名；如果未定义 TMPDIR，directory 不为空，则以 directory 所指向缓冲区中定义的字符串做路径名；TMPDIR 未定义，且 directory 为 NULL，则以常量 P_tmpdir 为路径名，常量 P_tmpdir 由 stdio 函数库定义。三者的按优先

权依次排序是：

TMPDIR > directory > P_tmpdir

参数 prefix 用于指定文件的前缀。当 prefix 不为 NULL 时，其所指向的缓冲区内存放的字符串指定了文件的前缀。

生成的文件名存放在一块动态内存区域中，由该函数调用系统调用 malloc 开辟。函数的返回值为指向存放文件名的缓冲区的指针。该文件使用完毕后，需要调用 free 函数来释放这一区域。

与【程序 7_8】及【程序 7_7】所实现的功能相同，【程序 7_9】是一个使用 tempnam 函数的例子。

【程序 7_9】

```
#include <stdio.h>

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

    if(!fp=fopen(name=tempnam(/home/stevens,NULL),"r+"))
    {
        printf("tempfile error");
        exit(1);
    }
    unlink(name);

    if(!fputs("This is a temp file.\n", fp))
    {
        printf("write error.\n");
        exit(1);
    }

    rewind(fp);
    if(!fgets(line, sizeof(line), fp))
    {
        printf("read error.\n");
        exit(1);
    }
}
```

```
    printf("%s\n", line);

    exit(0);
}
```

如果系统中没有定义环境变量 TMPDIR，则临时文件创建在目录 /home/stevens 下。如果系统中定义了环境变量，则创建在 TMPDIR 所指定的目录下。

7.4 小结与练习

7.4.1 小结

这一章介绍了基于流的输入输出的有关概念和操作。基于流的输入输出操作是由标准 c 库提供的。与基于文件描述符的输入输出相比，基于流的输入输出更简单方便。在大多数情况下，程序员们愿意使用基于流的输入输出方法。本章详细讲解了各种不同的输入输出方法，对于实际的应用程序的编写十分有用，应该熟练掌握。

此外，这一章还介绍了临时文件的概念。临时文件对于处理那些在程序运行时必须使用，但程序结束后不需要保留的数据是非常有用的。

7.4.2 习题与思考

- (1) 编写一段程序，完成 Linux 系统中 wc 命令实现的功能。
- (2) 用 tempnam 函数编写一段程序，根据启动时所带参数的不同测试出不同的文件名路径的选择情况。

第8章

内存管理

静态内存与动态内存

安全性问题

内存管理操作

使用链表

内存映像 I/O

小结与练习



C 语言是一种高级语言，但它一个显著的特点是可以在一个比较低的层次上运行，具有些低级语言的特征。它对内存的管理方式就是这一特征的一种体现。在 C 语言环境下，用户可以根据自己的需要分配和使用内存。

C 语言的这种设置，当然有助于增强和提高 C 程序的功能和灵活性。但也使得用户在编写程序时要使用到大量参数并且需要掌握各种复杂的算法，从而使得 C 程序的编写变得更加复杂，同时也增加了出错的可能性。

在这一章里，将讲解 C 语言中内存管理的相关函数调用，并将介绍由于对内存的操作而引发的安全性问题。

8.1 静态内存与动态内存

当一个程序执行时，需要使用到一定的内存空间用于存放程序执行中使用到的各种数据。按分配内存空间的方式不同，一个程序所使用的内存区域可以区分为静态内存与动态内存。简单地说，在程序开始运行时由系统分配的内存称为静态内存；在程序运行过程中由用户自己申请分配的内存称为动态内存。

8.1.1 静态内存

静态内存的申请是由编译器来分配的。对于用户程序中的各种变量，编译器在编译源程序时处理了为各种变量分配所需内存的工作。当程序执行时，系统就为变量分配所需的内存空间；至使用该变量的函数执行完毕返回时，自动释放所占用的内存空间。

使用静态内存对用户来说是很方便的。用户并不需要了解分配内存的具体细节，也不需要时刻考虑由于在程序结束前未释放所占用的内存空间而带来的可用内存泄漏。同时，静态内存也是不通过指针而使用变量的唯一方法。

但是，静态内存也存在着一定的缺陷。

首先，由于静态内存总是预先定义了存放数据的数组大小，这就有可能因为所传入的数据量大于数组容量而引发的溢出问题。用户当然可以定义一个足够大的数组，当如果要处理的数据远小于数组的大小时，无疑是对内存空间的浪费。

其次，由于在某个函数中分配的静态内存将在此函数运行结束时被系统自动释放，使用指针由子函数向主函数传递数据的设想是无法被实现的。可以通过下面的例子了解到这一点。

【程序 8_1】

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

char *upcase(char *inputstring);
```

```

int main(void)
{
    char *str1, *str2;

    str1=upcase("Hello");
    str2=capitao("Goodbye");

    printf("str1=%s, str2=%s\n", str1, str2);
    return 0;
}

char *upcase(char *oldstring)
{
    int counter;
    char newstring[100];

    strcpy(newstring, oldstring);
    for(counter=0; counter<strlen(newstring); counter++)
    {
        if(newstring[counter]>=97&&newstring[counter]<=122)
            newstring[counter]-=32;
    }
    return newstring;
}

```

这个程序试图通过返回子函数中的 char 指针而使主函数中的 char 指针指向某一特定的字符串。但由于子函数 upcase 中的 newstring 是静态内存，当子函数返回时就被系统释放了。因而无法实现用户期望的操作。一种简单的方法是通过如下修改而实现此功能。

【程序 8_2】

```

#include <stdio.h>
#include <string.h>

void upcase(char *inputstring, char *newstring);

int main(void)

```

```
{  
    char str1[100], str2[100];  
  
    upcase("Hello", str1);  
    capitao("Goodbye", str2);  
  
    printf("str1=%s, str2=%s\n", str1, str2);  
    return 0;  
}  
  
  
void upcase(char *inputstring, char *newstring)  
{  
    int counter;  
  
    strcpy(newstring, inputstring);  
    for(counter=0; counter<strlen(newstring); counter++)  
    {  
        if(newstring[counter]>=97&&newstring[counter]<=122)  
            newstring[counter] -= 32;  
    }  
    return newstring;  
}
```

由于所使用的字符数组变量是在主函数中定义的，当进程转到子函数中执行时，其所分配的内存空间未被释放。因此可以在子函数中对其赋值，然后在主函数中将变量的值输出到终端。

8.1.2 动态内存

使用动态内存时，用户可自行控制内存的分配和释放。这就是说，用户可以根据需要随时申请所需内存。在使用完毕后手动将此内存区域释放。

实际应用中，用户有很多理由使用动态内存。它的优点也是显而易见的。毫无疑问，使用动态内存是很方便的。用户可以根据需要随时分配所需的内存空间，可以自行定义所使用内存区域的大小，并在使用完毕后及时释放。

但动态内存的使用也存在着巨大隐患。任何处理过大型项目的用户都知道动态内存的使用会使内存管理变得多么复杂；以及要确切地记得在使用完毕后释放所占的内存空间是多么困难的事情。在大型应用程序中，由于在释放某块动态内存前将指向该内存区域的指针重新

赋值，从而使得此内存区域无法被释放的情况是十分常见的。通常将内存分配后没有被释放而导致可用内存减少称之为内存泄漏。避免内存泄漏耗尽系统资源正是许多服务器每隔一段时间就需要重新启动的原因。

此外还要注意，由于分配动态内存时，用户得到的是一块 void 类型的内存，用户可以将其作为任何类型的内存空间使用，也有可能引发一些无法预计的结果。

8.2 安全性问题

由于内存使用的问题，常常会引发一些安全性问题。例如前面提到的溢出、内存泄漏等问题。这在系统的安全问题中虽然只是一小部分，但由于内存操作而引发的安全问题是常见的。用户应该留意这些问题，否则有可能导致致命错误。

溢出错误的一种情况是，当使用静态内存时，定义了一个 100 元的数组，如果要将数据量大于 100 的一组数据写入这个数组，将发生溢出错误。虽然通常情况下程序将因此而错误终止，但在某些时候还是会因为覆盖了此缓冲区之后的数据而导致错误的发生。

要解决溢出的问题，一个最基本的想法是在用户程序中根据缓冲区大小截断所输入的数据。这一设想可以通过下面的例子实现。

【程序 8_3】

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

void upcase(char *inputstring, char *newstring);

int main(void)
{
    char str1[4], str2[4];

    upcase("Hello", str1);
    capito("Goodbye", str2);

    printf("str1=%s, str2=%s\n", str1, str2);
    return 0;
}

void upcase(char *inputstring, char *newstring)
```

```
{  
    int counter;  
  
    strcpy(newstring, inputstring,sizeof(newstring));  
    for(counter=0; counter<strlen(newstring); counter++)  
    {  
        if(newstring[counter]>=97&&newstring[counter]<=122)  
            newstring[counter] -= 32;  
    }  
    return newstring;  
}
```

在这个程序中，字符串 str1 和 str2 的长度故意设置得很小。当实际输入的字符串的长度大于字符串所限制的长度时，超出的部分将被截取。这一方法虽然可以避免溢出的问题，但会导致数据的丢失，更好的方法是使用动态内存或链表来保存数据，这两种方法将在接下来的部分详细介绍。

8.3 内存管理操作

前文已经介绍过，静态内存的分配是由编译器实现的。因此，在此所谈论的内存管理问题，主要是针对动态内存来说的。由于动态内存是完全由用户自行分配使用的，因此需要使用相应的函数调用实现所需操作。

8.3.1 动态内存的分配

分配内存空间所使用的函数调用如下。

```
#include <stdlib.h>  
void *malloc(size_t size);  
void *calloc(size_t nmemb, size_t size);
```

函数 malloc 和 calloc 都是用于分配动态内存空间的函数。

- 函数 malloc 的参数 size 表示申请分配的内存空间的大小，以字节记。
- 函数 calloc 的参数 nmemb 表示分配的内存空间占的数据项数目。参数 size 表示每一个数据项的大小，以字节记。也就是说，calloc 函数分配大小为 nmemb*size 大小的内存空间。

`calloc` 函数与 `malloc` 函数最大的区别是 `calloc` 函数将初始化所分配的内存空间，把所有位置为 0。

调用成功时，`malloc` 函数与 `calloc` 函数的返回值都为被分配的内存空间的指针；调用失败时，返回值为 `NULL`。

8.3.2 动态内存的释放

当对一块动态内存的使用结束后，需要手动将其释放。相关的函数的调用如下。

```
#include <stdlib.h>
void free(void *ptr);
```

此函数的作用是释放由 `malloc` 函数或 `calloc` 函数分配的动态内存。参数 `ptr` 是指向要释放的动态内存的指针。要注意在动态内存使用完毕后释放它，以免造成内存泄漏。

 当动态内存被释放时，原来指向它的指针就会变为悬空指针。此时使用该指针将会产生错误。

同样，可以使用动态内存实现【程序 8_1】的设想，将子函数中分配的内存空间指针返回主函数，实现数据的传递。修改后的程序功能与【程序 8_2】很相似。

【程序 8_4】

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

char *upcase(char *inputstring);

int main(void)
{
    char *str1, * str2;

    str1=upcase("Hello" );
    str2=capitao("Goodbye");

    printf("str1=%s, str2=%s\n", str1, str2);

    free(str1);
    free(str2);
```

```
    return 0;
}

char *upcase(char *inputstring)
{
    char *newstring;
    int counter;

    if(!newstring=malloc(strlen(inputstring)+1))
    {
        printf("ERROR ALLOCATING MEMORY! \n");
        exit(255);
    }
    strcpy(newstring, inputstring);
    for(counter=0; counter<strlen(newstring); counter++)
    {
        if(newstring[counter]>=97&&newstring[counter]<=122)
            newstring[counter] -= 32;
    }
    return newstring;
}
```

在这个程序中，由于所使用的是动态内存，因此程序可以将子函数中分配的内存空间的指针返回到主函数中。同时，由于使用了动态内存，使得子函数可以灵活地分配所需的内存空间。

8.3.3 调整动态内存的大小

对于用 `malloc` 函数或 `calloc` 函数分配好的动态内存，可以使用 `realloc` 函数来调整它的大小。该函数的说明如下。

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

`realloc` 函数的作用是重新调整一块动态内存区域的大小，参数 `ptr` 是指向要调整的动态内存的指针，应是 `malloc` 函数或 `calloc` 函数的返回值。参数 `size` 是新定义的动态内存的大小。

`size` 可以大于或小于动态内存的原大小，调用 `realloc` 函数时，通常是在原来的内存空间调整动态内存的大小。原有数据不被改动。当 `size` 大于原大小，而原位置中无法完成调整时，将重新开辟内存空间并将原有数据拷贝到新的内存空间中。

TIP

如果参数 `ptr` 为 `NULL`，则函数 `realloc` 的作用相当于函数 `malloc`。如果参数 `size` 为 0，则函数 `realloc` 的作用相当于函数 `free`。

【程序 8_5】

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

void upcase(char *inputstring, char *newstring);

int main(void)
{
    char *string;

    upcase("Hello",string);
    printf("str1=%s \n", string);
    capitao("Goodbye", string);
    printf(" str2=%s\n", string);

    free(string);

    return 0;
}

void upcase(char *inputstring, char *newstring)
{
    int counter;

    if(!newstring)
    {
        if(!newstring=realloc(NULL, strlen(inputstring)+1))
        {
            printf("ERROR ALLOCATING MEMORY! \n");
        }
    }
}
```

```
    exit(255);
}
}
else
{
    if(!newstring=realloc(newstring, sizeof(inputstring)+1))
    {
        printf("ERROR REALLOCATING MEMORY! \n");
        exit(255);
    }
    strcpy(newstring, inputstring);
    for(counter=0; counter<strlen(newstring); counter++)
    {
        if(newstring[counter]>=97&&newstring[counter]<=122)
            newstring[counter] -= 32;
    }
    return newstring;
}
```

这个程序针对函数 upcase 的参数 newstring 是否为 NULL 采取了不同的处理方式。如果 newstring 不为 NULL，则直接分配内存空间。否则调整原空间的大小以适应新的需要。

8.3.4 分配堆栈

函数 alloca 也用于分配内存空间。但与前面介绍的 malloc 函数、calloc 函数以及 realloc 函数不同，函数 alloca 是从进程的堆栈中分配空间的。并且当调用 alloca 函数的函数返回时，所分配的内存空间自动被释放。函数 alloca 的说明如下。

```
#include <stdlib.h>
void *alloca(size_t size);
```

参数 size 指定了所分配内存空间的大小。

 malloc 函数、calloc 函数以及 realloc 函数所分配的内存都是在堆中进行的。

8.3.5 内存锁定

最后简单讨论一下内存锁定。

为了更好地使用系统的资源，当一定内存区域在一段时间里未被使用时，系统内核就会将其中的内容暂时置换到磁盘上去，而释放此内存区域存放其他数据。到需要时再将原数据读回内存中。

如果用户不希望某块内存区域中的数据在暂时不使用时被置换到磁盘上，可以对该区域实行内存锁定。相关的函数调用如下。

```
#include <sys/types.h>
int mlock(const void *addr, size_t length);
int munlock(void *addr, size_t length);
int mlockall(int flag);
int munlockall(void);
```

- 函数 `mlock` 用于锁定某一内存区域，参数 `addr` 表示要锁定内存区域的起始地址。参数 `length` 表示要锁定的内存区域的大小。
- 函数 `munlock` 用于解除对某一内存区域的锁定。
- 函数 `mlockall` 用于一次锁定多个内存页。参数 `flag` 的函数所执行的操作。相关选项有两个。
`MCL_CURRENT`: 锁定所有内存页。
`MCL_FUTURE`: 锁定所有为进程的地址空间添加的内存页。
- 函数 `munlockall` 用于解除所有内存锁定。



只有超级用户进程才有权限锁定内存空间或解除内存空间的锁定。

8.4 使用链表

虽然使用动态内存可以方便地使用内存，但动态内存也有一定的局限性，就是在数据输入到程序之前必须知道数据的大小，以便申请相应的动态内存。然而，在很多情况下，用户都无法事先知道这个值。因而也就无法申请相应的内存空间。对于这种事先未知大小的数据输入，可以使用链表将其分块保存。

链表是一种动态地进行存储分配的结构。链表中的各个元素是一个结构，每个元素称为链表的一个结点。此结构中包含有一个指向此结构的指针，用于指向链表中的下一个结点。

链表的最后一个结点的指针为 NULL，表示链表结束。例如，如下的结构就可以构成一个链表。

```
struct strdata
{
    char *string;
    struct strdata *next;
}
```

这是一个最简单的构成链表的结构。这个链表由保存字符串结构 strdata 组成。

下面通过一个具体的程序例子来说明用链表保存数据的方法。

【程序 8_6】

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

#define DATASIZE 10

typedef struct stringdata
{
    char *string;
    int iscontinuing;
    struct stringdata *next;
} mydata;

mydata *append(mydata *start, char *input);
void displaydata(mydata *start);
void freedata(mydata, *start);

int main(void)
{
    char input[DATASIZE];
    mydata *start=NULL;

    printf("ENTER SOME DATA,AND PRESS Ctrl+D WHEN DONE. \n");

    while(fgets(input, sizeof(input), stdin))
```

```
{  
    start=append(start, input);  
}  
  
displaydata(start);  
freedata(start);  
return 0;  
}  
  
mydata *append(mydata *start, char *input)  
{  
    mydata *cur=start, *prev=NULL, *new;  
  
    while(cur)  
    {  
        prev=cur;  
        cur=cur->next;  
    }  
    cur=prev;  
  
    new=malloc(sizeof(mydata));  
    if(!new)  
    {  
        printf("COULDN'T ALLOCATE MEMORY! \n");  
        exit(255);  
    }  
  
    if(cur)  
        cur->next=new;  
    else  
        start=new;  
  
    cur=new;  
  
    if(!cur->string=malloc(sizeof(input)+1)  
    {  
        printf("ERROR ALLOCATING MEMORY! \n");  
        exit(255);  
    }
```

```
    }

    strcpy(cur->string, input);
    cur->iscontinuing=!(input[strlen(input)-1]=='\n'||input[strlen(input)-1]=='\r');
    cur->next=NULL;

    return start;
}

void displaydata(mydata *start)
{
    mydata *cur;
    int linecounter=0, structcounter=0;
    int newline=1;

    cur=start;
    while(cur)
    {
        if(newline)
            printf("LINE %d:",++linecounter);
        structcounter++;
        printf("%s",cur->string);
        newline=!cur->iscontinuing;
        cur=cur->next;
    }
    printf("THIS DATA CONTAINED %d LINES AND WAS STORED IN %d STRUCTS. \n",
           linecounter,structcounter);
}

void freedata(mydata *start)
{
    mydata *cur, *next=NULL;

    cur=start;
    while(cur)
    {
        next=cur->next;
```

```

    free(cur->string);
    free(cur);
    cur=next;
}
}

```

这个程序将终端输入的一系列字符串用链表的形式保存下来。然后再将这些数据组装起来，回显到输出终端。链表的节点为 stringdata 结构。 stringdata 结构中的整型量 iscontinuing 用于表示当前节点是否为链表的末尾。如果 iscontinuing 有值，则表示此节点不是链表的末尾。

8.5 内存映像 I/O

准确地说，内存映像 I/O 其实并不是一种内存管理操作，而是一种特殊的 I/O 操作方式。内存映像 I/O 其实是内存管理的一个例子。

内存映像其实是在内存中创建一个与外存中文件完全相同的映像。用户可以将整个文件映射到内存中，也可以将文件的一部分映射到内存中。使用操作内存的方法来对文件进行操作。系统会将对内存映像文件所做的改动反映到真实文件中去。使用内存映像文件有三个突出的特点。

首先，使用内存映像文件可以加快 I/O 操作的速度，通常的 I/O 操作是通过内核缓冲区来实现的。使用磁盘缓存技术来对文件进行读写操作。内存的访问速度比外存要快得多。使用内存映像 I/O 操作可以使输入输出工作加速。

其次，由于将文件映射到内存中，用户可以通过指针对文件进行访问，与访问其他的内存数据毫无区别。用户可以使用指针来方便地实现对文件中数据的定位。

最后，使用内存映像 I/O 可以实现文件数据的共享。将磁盘文件映射到一个共享内存区域中，就可以很方便地实现数据的共享，并且将数据保存到磁盘的工作也同时完成。



共享内存是一种进程间通信的机制。它通过一块允许多个进程访问的内存区域来实现进程间的通信，将在进程间通信一章中详细讲解。

内存映像 I/O 也有一定的局限性。当用户将文件或文件的一部分映射到内存中时，必须事先定义所使用内存空间的大小，要向内存映像文件添加数据是一项很麻烦的工作。此外，内存映像操作只能对普通文件这样的可以内部定位的文件进行，而不能对管道、套接字这样的文件进行。

下面介绍内存映像 I/O 是如何实现的。

8.5.1 创建内存映像文件

要将一个外存中的文件映射到内存中，需使用如下的函数。

```
#include<sys/types.h>
#include <sys/mman.h>
void *mmap(void *start, size_t length, int prot, int flag, int fd, off_t offset);
```

此函数用于将一个文件或它的一部分映射到内存中。参数 `start` 是一个 `void` 指针，通常为 `NULL`。如果不为 `NULL`，则表示希望将文件映射到此指针指向的位置。但不能保证调用一定将文件映射到这一位置。参数 `length` 定义内存映像文件所占用的内存空间的大小，以字节记。参数 `prot` 表示内存映像文件的安全属性，它的可供使用的选项见表 8-1。参数 `flag` 是内存映像的标志。相关的标志见表 8-2。参数 `fd` 是要映射的文件的描述符。参数 `offset` 表示所映射的数据内容距离文件头的偏移量。

当调用成功时，返回值为指向内存映像文件起始地址的指针；调用失败时，返回值为-1。

表 8-1 prot 选项及其含义

prot	含 义
PROT_EXEC	被映像内存可能含有机器码，可被执行
PROT_NONE	映像内存不允许访问
PROT_READ	映像内存可读
PROT_WRITE	映像内存可写

表 8-2 flags 相关标志位及其含义

flag	含 义
MAP_FIXED	如果无法在 <code>start</code> 指定的地址建立内存映像文件，则出错返回
MAP_PRIVATE	对内存映像文件所做的改动不反映到外存文件中
MAP_SHARED	对内存映像文件所做的改动都将被保存到外存文件中



要将一个文件映射到内存，需要使用文件的文件描述符。也就是说，必须先在进程中将该文件打开。

8.5.2 撤销内存映像文件

内存映像文件使用完毕后，要调用如下函数来撤销。

```
#include <sys/types.h>
#include <sys/mman.h>
int munmap(void *start, size_t length);
```

参数 `start` 表示要撤销的内存映像文件的起始地址。参数 `length` 表示要撤销的内存映像文件的大小。

调用成功时，返回值为 0；调用失败时，返回值为 -1，并将 `errno` 设置为相应值。

8.5.3 将内存映像写入外存

要将对内存映像文件所做改动保存到外存，需要调用如下函数。

```
#include <sys/types.h>
#include <sys/mman.h>
int msync(const void *start, size_t length, int flag);
```

此函数将内存映像文件中的改动刷新到外存文件中。参数 `start` 表示要保存到外存的那些源文件的起始地址。参数 `length` 表示内存映像文件的大小。参数 `flag` 设置了函数的相应操作，具体选项见表 8-3。

表 8-3 flag 对应选项

flag	含 义
<code>MS_ASYNC</code>	调度一个写操作并返回
<code>MS_INVALIDATE</code>	使映像到相同文件的内存映像文件无效以便将它们更改为新的数据
<code>MS_SYNC</code>	完成写操作后函数返回

下面来看一个使用内存映像文件的例子。

【程序 8_7】

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <stdio.h>
```

```
#include <string.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <fcntl.h>

int write_buffer(int fd, const void *buf, int count);

int main(void);
{
    int outfile;
    char *mapped;
    char *ptr;

    if(outfile=open("test.dat", O_RDWR|O_CREAT|O_TRUNC, 0640)==-1)
    {
        printf("COULDN'T OPEN THIS FILE!\n");
        exit(254);
    }

    lseek(outfile, 1000, SEEK_SET);
    if(write(outfile, "0", 1)==-1)
    {
        printf("ERROR, WRITE FAILED!\n");
        exit(254);
    }
    mapped= mmap(NULL, 1000, PROT_READ|PROT_WRITE, MAP_SHARED, outfile, 0);
    if(!mapped)
        printf("ERROR, MMAP FAILED!\n");

    ptr=mapped;
    printf("PLEASE ENTER A NUMBER:");
    fgets(mapped, 80, stdin);

    ptr+=mapped;
    sprintf(ptr, "YOUR NUMBER TIMES TWO IS:%d.\n", atoi(mapped)*2);
    printf("YOUR NUMBER TIMES TWO IS:%d\n", atoi(mapped)*2);
```

```
msync(mapped, 1000, MS_SYNC);
munmap(mapped,1000);

if(!close(outfile))
{
    printf("POSSIBLY SERIOUS ERROR,CLOSE FILE FAILED");
    exit(254);
}
return 0;
}

int write_buffer(int fd, const void *buf, int count)
{
    const void *pts=buf;
    int status=0, n;

    if(count<0)
        return(-1);
    while(status!=count)
    {
        if(n=write(outfile, "0",1)==-1)
        {
            printf("ERROR, WRITE FAILED!\n");
            exit(254);
        }
        if(n<0)
            return(n);
        status+=n;
    }
    return(status);
}
```

【程序 8_7】的作用是打开一个文件 test.dat 并将它映射到内存空间。通过内存映像文件对它进行操作。

8.5.4 改变内存映像文件的属性

对内存映像文件属性的操作主要是针对内存映像文件的保护值和大小进行的。具体的操作如下。

1. 修改内存映像文件的保护值

在建立一个内存映像文件时，函数 `mmap` 设定了函数的保护值。需要时，用户可以对保护值进行修改，使用的函数如下：

```
#include <sys/types.h>
#include <sys/mman.h>
int protect(const void *addr, size_t length, int prot);
```

参数 `addr` 表示内存映像文件的起始地址。参数 `length` 表示内存映像文件的大小。参数 `prot` 为新设定的保护值。

调用成功时，返回值为 0；调用失败时，返回值为 -1，并将 `errno` 置为相应值。

2. 修改内存映像文件的大小

要改变内存映像文件的大小，需要调用如下的函数。

```
#include <sys/types.h>
#include <sys/mman.h>
void *mremap(void *old_addr, size_t old_length, size_t new_length, unsigned long flag);
```

函数的作用是将参数 `old_addr` 指向的内存映像文件的大小由 `old_length` 调整到 `new_length`。参数 `flag` 用于设置是否在需要时移动该内存映像文件的位置。相应的位是 `MREMAP_MAYMOVE`。如果设置了该位，表示在需要时移动位置，否则将出错返回。

调用成功时，返回值为调整后内存映像文件的起始地址；调用失败时，返回值为 -1。

8.6 小结与练习

8.6.1 小结

这一章介绍了静态内存和动态内存的概念，并讲解关于动态内存分配和使用的各种操作，以及另一种在内存中保存数据的方法——链表。同时也讨论了在内存使用中可能引发的

各种安全性问题。熟悉这些内存操作将使用户在编写程序时更加得心应手。

此外本章还介绍了一种 I/O 操作——内存映像 I/O，这是一种 I/O 操作的方法。但作为一个使用内存的例子，有助于学员更好地了解内存操作。同时，内存映像 I/O 也是一种十分重要的输入输出操作，需要确实掌握。

8.6.2 习题与思考

(1) 阅读【程序 8_9】并指出其中的错误。

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

char *upcase(char *string);

int main(void)
{
    char *buf, *newstr;
    char oldstr1[ ]={"abcdefg"};
    char oldstr2[ ]={"xyz"};
    int counter;

    buf=malloc(15);
    strcpy(buf, "CHANGE STRINGS.");
    fprintf(stdout, "%s\n",buf);
    free(buf);

    for(counter=0; counter<sizeof(oldstr1); counter++)
    {
        putchar(oldstr1[counter]);
        putchar(oldstr2[counter]);
    }

    newstr=upcase(oldstr1,sizeof(oldstr1));
    fprintf(stdout, "NEW STRING 1:%s.\n",newstr);

    newstr=upcase(oldstr2,sizeof(oldstr2));
    fprintf(stdout, "NEW STRING 2:%s.\n",newstr);
```

```
free(newstr);

strcpy(buf, "THE END.");
fprintf(stdout, "%s\n", buf);

free(buf);
}

char *upcase(char *string, int length)
{
    char *newstring;
    char temp;
    int counter;

    newstring=calloc(length,sizeof(char));
    for(counter=0; counter<length; counter++)
    {
        temp=*(string+counter);
        if(temp>=97&&temp<=122)
            *(newstring+ counter)=temp - 32;
        else
            *(newstring+ counter)=temp;
    }
    return newstring;
}
```

(2) 编写一个使用内存映像 I/O 的程序。该程序可以将指定文件中的小写字母改变为大写字母。

第9章

进程控制

进程的基本概念

进程控制的相关函数

多个进程间的关系

线程

小结与练习

进程是操作系统中一个十分重要的概念，它是一个程序的一次执行过程。程序的每一个运行中的副本都有其自己的进程。反过来说，程序就是进程的一种静态描述。系统中运行的每一个程序都是在它的进程中运行的。

因此，用户在系统中执行各种操作时，总要通过进程来完成的。了解 Linux 系统中进程的概念和属性，熟悉使用进程的操作，将使用户在使用 Linux 系统完成各种工作时更加得心应手。

9.1 进程的基本概念

进程的概念起源于 60 年代，目前已成为操作系统和并发程序设计中非常重要的概念。用户在操作系统中所做的每一件事，都是通过进程实现的。要灵活使用进程，首先需要了解进程的一些基本概念。

9.1.1 进程基本介绍

在 Linux 环境下，进程是一个十分重要的概念。按现在通行的认识，进程是具有一定功能的程序关于一个数据集合的一次执行过程。对一个特定程序来说，它的每一个正在运行中的副本都有自己的进程。就是说，如果用户在一个程序的一次运行尚未结束时再次启动该程序，则将有两个进程在运行这一个程序。多个进程可以同时运行，各个进程之间相互隔开，除非不同进程之间需要数据交换，否则互不影响。

一个进程的存在过程，可以分为进程的产生、进程的执行和进程的结束 3 个步骤。当一个程序被启动时，就产生了一个新的进程。进程在系统内核的管理下得到执行。当某个进程执行完毕后，该进程就消亡了。

Linux 系统支持多个进程同时运行。所谓同时，其实是 Linux 系统在各个进程之间调度，轮流使每个进程占用 CPU 一个时间片。由于每个时间片和宏观的时间相比很小，而各个进程可以频繁地得到时间片，于是就使用户看到了多个进程“同时”运行的情况。在每个进程属性中的安全信息里都设定有一个优先级，系统根据它来决定各个进程从 CPU 获得的时间片的大小。下面将具体介绍进程的属性。

用户在执行一个程序以完成一定的功能时，为了提高程序执行的效率，可以把一个程序设计成由若干个部分组成，由若干个进程同时执行。这就是所谓并发程序的概念。运行并发程序时，对应于该程序的若干进程同步运行，这些进程之间应是相互独立的，但在必要时也会发生联系。

如果在各个进程运行时，不同进程需要使用相同的资源的话，可能会出现资源使用上的冲突。在这种情况下，只能设定由各个进程轮流使用该资源。

此外，不同进程之间可能会需要相互合作，进行某种数据的交流。一个进程可能要等待其他进程运行到一定阶段，得到一定结果后才能继续运行。这就要在各进程之间建立一种通

信关系。

另一种情况是，可能会出现进程间的互斥关系，即不同进程需要调用同一程序，但却不允许多个进程同时调用该程序。例如，车站的各售票窗口中，可能会有两个以上窗口在同一时间需要查询同一车次的车票，如果这些窗口同时都调用了余票查询程序，可能会得到相同的余票列表而将同一张票卖给不同的乘客。因此，程序员需要设定该程序同一时间只允许一个进程进入，其他进程需等待其完成后再调用该程序。对文件的使用也有类似的问题，程序员可以通过文件的锁定保证在需要时只由一个进程对某一特定文件进行修改。

当然，多个进程并不需要同一时间产生并都维持到整个程序运行结束。用户可以根据需要动态地产生结束进程。也就是说，一个进程可以派生另一个进程，这就是所谓父进程与子进程的关系。

9.1.2 进程的属性

每个进程都具有各自的属性，其中包括了进程的详细信息。进程的属性如下：

- **进程标识符 (PID)**: 当一个进程产生时，系统都会为它分配一个进程标识符 PID。对任意一个时刻来说，每个进程 PID 都是唯一的。但如果一个系统长时间运行，则 PID 将会被系统重复使用。
- **进程所占的内存区域**: 每个进程执行时都需要占用一定的内存区域，此区域用于保存该进程所运行的程序的代码和所使用的变量。对一个进程所占内存中数据的任何改动，都只在该进程内部产生影响，而对其他进程不产生任何影响。一个进程的运行出现错误时，并不会影响其他进程的顺利执行。这也是 Linux 系统具有高度稳定性的原因。
- **相关文件的文件描述符**: 当一个进程在执行时，它需要使用一些相关的文件描述符。包括系统默认自动打开的标准输入、标准输出和标准错误输出以及进程所打开文件的文件描述符。不同的进程打开同一个文件时，所使用的文件描述符是不同的。一个进程文件描述符的改变并不对其他进程打开的同一文件的描述符造成影响。
- **安全信息**: 一个进程的安全信息包括用户识别号和组识别号。这些值由 Linux 系统内核使用，用于决定进程具体所能做的事情。
- **进程环境**: 一个进程的运行环境包括环境变量和启动该进程的程序调用的命令行。
- **信号处理**: 一个进程有时需要用信号同其他进程之间进行通信。进程可以发送和接收信号，并对其作出相应处理。
- **资源安排**: 进程是调度系统资源的基本单位。当多个进程同时运行时，Linux 系统内核安排不同进程轮流使用系统的各种资源。
- **同步处理**: 多个程序之间同步运行的实现，也是通过进程来完成的。这将会使用到诸如共享内存、文件锁定等方法。
- **进程状态**: 在一个进程存在期间，每一时刻进程处于一定的状态，包括运行、等待被调度或睡眠状态。处于运行状态时，进程占有 CPU，执行进程；处于等待被调度状态时，进程等待获得 CPU；处于睡眠状态时，进程等待一个事件发生而不处理任

何事情。

9.2 进程控制的相关函数

进程的相关操作要使用众多函数，其中包括系统调用也包括库函数。下面介绍进程的创建、终止等具体的进程操作。

9.2.1 进程的创建

1. 派生进程

要创建一个进程，最基本的系统调用是 `fork`。系统调用 `fork` 用于派生一个进程。其说明如下。

```
#include <unistd.h>
pid_t fork(void);
pid_t vfork(void);
```

调用 `fork` 时，系统将创建一个与当前进程相同的新的进程。它与原有的进程具有相同的数据、连接关系和在程序同一处执行的连续性。通常将原有的进程称为父进程，而把新生成的进程称为子进程。子进程是父进程的一个拷贝，子进程获得同父进程相同的数据，但是同父进程使用不同的数据段和堆栈段。子进程将从父进程获得某些属性，但也要更改一些属性。具体情况见表 9-1。

表 9-1 父子进程间的属性异同

继承属性	差 异
真实用户 ID 和组 ID，有效用户 ID 和组 ID	进程 ID
进程组 ID	父进程 ID
SESSION ID	子进程运行时间记录
所打开文件及文件的偏移量	父进程对文件的锁定
控制终端	
设置用户 ID 和设置组 ID 标记位	
根目录与当前工作目录	
文件缺省创建的权限掩码	
可访问的内存区段	
环境变量及其他资源分配	

`fork` 调用将执行两次返回（这样的函数在 Linux 中只有少数几个），它将从父进程和子进程中分别返回。从父进程返回时的返回值为子进程的 PID，而从子进程返回时的返回值为 0，并且返回都将执行 `fork` 之后的语句。调用出错时，返回值为 -1，并将 `errno` 置为相应值。【程序 9_1】是一个派生子进程的例子。

【程序 9_1】

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main(void)
{
    pid_t pid;
    if(pid=fork())<0
    {
        printf("fork error!\n");
        exit(1);
    }
    else if(pid==0)
    {
        printf("Child process is printing.\n");
    }
    else
    {
        printf("Parent process is printing.\n");
    }

    exit(0);
}
```

【程序 9_1】是一个简单的派生进程的例子。由于父进程和子进程的运行无关。父进程先返回与子进程先返回的情况都有可能发生。因此，运行这段程序时，显示的结构既可能是

Child process is printing.
Parent process is printing.

也可能

Parent process is printing.

Child process is printing.

调用 `vfork` 的作用与调用 `fork` 的作用基本相同，但 `vfork` 并不完全拷贝父进程的数据段，而是和父进程共享数据段。这是因为通常 `vfork` 函数是与 `exec` 函数族的函数连用，创建执行另一个程序的新进程。在接下来的内容中将介绍 `exec` 函数族的函数。和调用 `fork` 不同，调用 `vfork` 对于父子进程的执行次序有所限制，调用 `vfork` 时，父进程被挂起，子进程运行至调用 `exec` 函数族或调用 `exit` 时解除这种状态。

【程序 9_2】

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main(void)
{
    pid_t pid;
    if(pid=vfork())<0)
    {
        printf("fork error!\n");
        exit(1);
    }
    else if(pid==0)
    {
        printf("Child process is printing.\n");
    }
    else
    {
        printf("Parent process is printing.\n");
    }

    exit(0);
}
```

尽管【程序 9_2】和【程序 9_1】几乎没有不同，但【程序 9_2】的显示结果是固定的。

Child process is printing.

Parent process is printing.

这是由于调用 `vfork` 函数将使父进程挂起，直至子进程返回（或调用 `exec` 函数），因此总是子进程先返回。

2. 创建执行其他程序的进程

前面已经提到，可以使用 `exec` 族的函数执行新的程序，以新的子进程来完全替代原有的进程。`exec` 函数族的函数说明如下。

```
#include <unistd.h>
int exec(const char *pathname, const char *arg, ...);
int execvp(const char *filename, const char *arg, ...);
int execle(const char *pathname, const char *arg, ..., char *const envp[]);
int execv(const char *pathname, char *const argv[]);
int execvp(const char *filename, char *const argv[]);
int execve(const char *pathname, char *const argv[], char *const envp[]);
```

函数名中含有字母“l”的函数，其参数个数不定。其参数由所调用程序的命令行参数列表组成，最后一个 `NULL` 表示结束。函数名中含所有字母“v”的函数，则是使用一个字符串数组指针 `argv` 指向参数列表，这一字符串数组和含有“l”的函数中的参数列表完全相同，也同样以 `NULL` 结束。

函数名中含有字母“p”的函数可以自动在环境变量 `PATH` 指定的路径中搜索要执行的程序。因此它的第一个参数为 `filename` 表示可执行函数的文件名。而其他函数则需要用户在参数列表中指定该程序路径，其第一个参数 `pathname` 是路径名。路径的指定可以是绝对路径，也可一个是相对路径。但出于对系统安全的考虑，建议使用绝对路径而尽量避免使用相对路径。

函数名中含有字母“e”的函数，比其他函数多含有一个参数 `envp`。该参数是字符串数组指针，用于制定环境变量。调用这两个函数时，可以由用户自行设定子进程的环境变量，存放在参数 `envp` 所指向的字符串数组中。这个字符串数组也必须由 `NULL` 结束。其他函数则是接收当前环境。

【程序 9_3】

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main(void)
{
    pid_t pid;
    if(pid=vfork())<0)
```

```
{  
    printf("fork error! \n");  
    exit(1);  
}  
else if(pid==0)  
{  
    printf("Child process PID: %d.\n",getpid());  
    setenv("PS1", "CHILD\\$", 1);  
    printf("Process%4d: calling exec.\n"getpid());  
    if(exec("./bin/sh", "/bin/sh", "arg2", NULL)<0)  
    {  
        printf("Process%4d: execle error!\n",getpid());  
        exit(0);  
    }  
    printf("Process%4d: You should never see this because the child is already gone. \n",  
          getpid());  
    printf("Precess%4d: The child process is exiting.  
");  
}  
else  
{  
    printf("Parent process PID:%4d.\n", getpid());  
    printf("Process%4d: The parent has fork process %d.\n", pid);  
    printf("Process%4d: The child has called exec or has exited.\n", getpid());  
}  
return 0;  
}
```

【程序 9_3】的作用是派生一个子进程后，在子进程中使用 exec 函数调用 shell 命令 sh。

3. Linux 系统特有的调用

在 Linux 系统中，有一个特有的调用 __clone，其说明如下。

```
#include <sched.h>  
int __clone(int (*fn)(void *arg),void *child_stack,int flags,void *arg);
```

此函数是 fork 函数的变形，对父子进程的共享资源提供了更多控制。参数 fn 是函数指针，指向要执行的函数。参数 child_stack 是子进程堆栈段的指针。参数 flags 是用于表示不同

继承内容的标志，其取值见表 9-2，其取值可以是表中所列值的逻辑或。参数 arg 是指向存放要传递给所调用程序的参数的缓冲区。

表 9-2

flags 标志的选取

flags	含 义
CLONE_VM	继承父进程的虚拟存储器属性
CLONE_FS	继承父进程的 chroot、chdir 和 umask
CLONE_FILES	继承父进程的文件描述符
CLONE_PID	继承父进程的文件锁、进程号及时间片
CLONE_SIGHAND	继承父进程的信号处理程序

9.2.2 进程等待

在多进程处理时，用户可能需要用到有关进程等待的操作。这种等待可以是进程组成员间的等待，也可以是父进程对子进程的等待。

例如，当一个进程结束时，Linux 系统将产生一个 SIGCHLD 信号通知其父进程。在父进程未查询子进程结束的原因时，该子进程虽然停止了，但并未完全结束。此时这一进程被称为僵尸进程（zombie process）。这时的处理方法之一就是使用进程等待的系统调用 wait 和 waitpid。

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

- wait 调用专用于等待子进程。其参数 status 是一个整数指针，当子进程结束时，将子进程的结束状态字存放在该指针指向的缓冲区。当调用 wait 时，父进程进程将被挂起，直至该进程的一个子进程结束时，该调用返回。如果调用没有子进程，则错误返回。调用成功时，返回值为被置于等待状态的进程的 PID；调用失败时，返回值为 -1。
- waitpid 调用比 wait 调用灵活一些，可以用来等待指定的进程，且可以使进程不挂起而立刻返回。参数 pid 用于指定所等待的进程。其取值和相应设定见表 9-3。参数 option 则用于指定进程所做操作。它可以取 0 和常数 WNOHANG、WUNTRACED。取 0 时表示将进程挂起等待其结束；取 WNOHANG 表示不使进程挂起而立刻返回；取 WUNTRACED 表示如果进程已结束则返回。

表 9-3

调用 waitpid 时 pid 的取值

pid	含 义
pid>0	等待进程 ID 为 pid 所指定值的子进程
pid=0	等待进程组 ID 与该进程相同的子进程
pid=-1	等待所有子进程，等价于 wait 调用
pid<-1	等待进程组 ID 为 pid 绝对值的子进程

当参数 status 不为 NULL 时，如果函数成功返回，则将子进程的结束状态字存放在 status 所指向的缓冲区中。利用这个状态字，需要时可以使用一些由 Linux 系统定义的宏来了解子程序结束的原因。这些宏的定义与作用如下：

- WIFEXITED(status): 子进程正常结束时，返回值为真。
- WIFSIGNALED(status): 子进程接收到信号结束时，返回值为真。但如果进程接收到信号时调用 exit 函数结束，则返回值为假。
- WIFSTOPPED(status): 在调用函数 waitpid 时制定了 WUNTRACED 选项，且该子进程使 waitpid 返回时，这个宏的返回值为真。
- WSTOPSIG(status): 当 WIFSTOPPED 为真时，将获得停止该进程的信号。
- WTERMSIG(status) 当 WIFSIGNALED 为真时，将获得终止该进程的信号。
- WEXITSTATUS(status): 当 WIFEXITED 为真时，此宏才可以使用。返回进程退出的代码。

下面的程序是一个进程等待的一个例子。

【程序 9_4】

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>

void h_exit(int status);

int main(void)
{
    pid_t pid;
    int status;

    if(pid=fork())<0
    {
        printf("fork error!\n");
        exit(0);
    }
```

```
}

else if(pid==0)
    exit(7);
if(wait(&status)!=pid)
{
    printf("wait error!\n");
    exit(0);
}
h_exit(status);

if(pid=fork())<0)
{
    printf("fork error!\n");
    exit(0);
}
else if(pid==0)
    exit(1);
if(wait(&status)!=pid)
{
    printf("wait error!\n");
    exit(0);
}
h_exit(status);

if(pid=fork())<0)
{
    printf("fork error!\n");
    exit(0);
}
else if(pid==0)

if(wait(&status)!=pid)
{
    printf("wait error!\n");
    exit(0);
}
h_exit(status);
```

```
    exit(0);
}

void h_exit(int status)
{
    if(WIFEXITED(status))
        printf("normal termination, exit status=%d .\n", WEXITSTATUS(status));
    else if(WIFSIGNALED(status))
        printf("abnormal termination, exit status=%d. \n", WTERMSIG(status));
}
```

在【程序 9_4】中，三次派生子进程，并使用不同的方法结束子进程。调用 `h_exit` 函数检测结束的方式。

此外，系统还提供了另两个函数可以用于进程等待。这两个函数的说明如下。

```
#define _USE_BSD
#include <sys/types.h>
#include <sys/resource.h>
#include <sys/wait.h>
pid_t wait3(int *status, int options, struct rusage *rusage);
pid_t wait4(pid_t pid, int *status, int options, struct rusage *rusage);
```

函数 `wait3` 和 `wait4` 的作用基本上分别相当于函数 `wait` 和 `waitpid`，但和函数 `wait` 和 `waitpid` 相比，函数 `wait3` 和 `wait4` 多一个参数 `rusage`。该参数是一个结构指针，调用这两个函数时，如果 `rusage` 不为 `NULL`，则关于子进程执行时的相关信息。将被写入该指针指向的缓冲区内。

【程序 9_5】

```
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/resource.h>
#include <stdio.h>
#include <unistd.h>
#include <stdarg.h>
#include <unistd.h>
#include <stdlib.h>
```

```
void waitchildren(int signum);
```

```
void h_exit(int status);

int main(void)
{
    pid_t pid;
    int status;

    if(pid=fork())<0
    {
        printf("fork error!\n");
        exit(0);
    }
    else if(pid==0)
    {
        printf("Hello from the child process%4d!\n", getpid());
        setenv("PS1", "CHILD \\ $", 1);
        printf("Process%4d: I'm calling exec. \n", getpid());
        execl("/bin/sh", "/bin/sh", NULL);
        printf("Process%4d: You should never see this because the child is already gone.\n", getpid());
    }
    else if(pid!=1)
    {
        printf("Hello from the parent process%4d!\n", getpid());
        printf("Process%4d: The parent has forked process %d. \n", getpid(),pid);
        printf("Process%4d: The parent is waiting for the child to exit.\n", getpid());
        wait4(pid, status, 0, NULL);
        h_exit(status);
    }
}

return 0;
}
```

```
void h_exit(int status)
{
    if(WIFEXITED(status))
        printf("normal termination, exit status=%d .\n", WEXITSTATUS(status));
    else if(WIFSIGNALED(status))
```

```
    printf("abnormal termination, exit status=%d. \n", WTERMSIG(status));  
}
```

【程序 9_5】是一个用 `wait4` 函数实现等待的例子。在这里的功能完全相当于 `waitpid` 函数。

9.2.3 进程的终止

在前面已经看到，程序员可以用 `exit` 来结束一个进程。在 Linux 环境中，一个进程的