前　言

Linux 开创了操作系统历史上的一个奇迹，它不仅免费，而且开放全部的源代码。Linux 已经发展成为最为流行的免费操作系统。

在我国，Linux 已经广泛应用于政府、军队、金融、电信等敏感部门和关键行业中。可以预见，今后 Linux 在作为企业服务器，嵌入式应用开发平台等方面将占据越来越重的市场份额。相应地，人才市场对 Linux 下的开发人员的需求也将越来越大。

C 语言作为一种当前使用非常广泛的高级程序设计语言，具有简单易用、跨平台、可移植性好的特点。Linux 内核就是使用 C 语言开发。C 语言是 Linux 上的主要开发语言，它在 Linux 编程开发中扮演着重要的角色，它们形成了相得益彰的完美组合，为用户提供了一个强大的编程环境。在国内很多 Linux 爱好者仅停留在系统管理的层次上，而更多程序员要做 Linux 下的程序开发却无从下手，重要的是况且很难找到合适的学习参考资料。这本书正是从这样的结合点出发，介绍在 Linux 系统中使用 C 语言编程的有关知识。读者通过本书的学习能够快速地学会 Linux 下 C 语言编程，掌握其中的编程方法和技巧，并能从一开始就养成良好的编程习惯，从而实现 Linux 环境下的编程知识入门和提高。

本书分 4 个部分介绍了如何使用 C 语言在 Linux 平台下进行软件开发。

第一篇 Linux 和 C 编程基础：第 1～5 章。主要介绍 Linux 的安装与使用，针对初学者和对 C 语言不熟悉的读者回顾了 C 语言的编程基础知识、开发技巧等，同时以知名公司在招聘时使用的笔试、面试题作为例题分析 C 语言的重点和难点。在这部分里，还结合实例介绍了 Linux 的开发环境，如 Vi 和 Emacs 编辑器、gcc 编译器、gdb 调试器、make 自动编译工具等。

第二篇 Linux 系统编程：第 6～10 章。主要介绍了 Linux 下的系统编程，包括文件和目录操作、进程和线程控制、信号的使用、进程间通信。本部分以大量的程序实例来说明各个系统调用的使用方法。每章的最后部分都有一至两个具有一定规模的综合实例，如实现自己的 ls 命令、实现自己的 myshell 等。

第三篇 Linux 网络和图形界面编程：第 11～12 章。主要介绍 Linux 下的网络编程和图形界面编程。由于 Linux 和 TCP/IP 协议的内在结合，使得在 Linux 下可以开发出功能十分强大的网络应用程序。同样，也可以开发出界面十分精美的图形界面程序。本部分对 Linux 下网络编程和图形界面编程作了详细地介绍，并通过实例展示它们的具体应用。

第四篇 Linux 项目实践：第 13 章项目开发案例。通过开发一个 BT 下载软件来完整地介绍一个软件的开发过程。详细分析和解释了 BT 协议，并在此基础上使用 C 语言在 Linux 环境下设计和实现了一个 BT 下载软件。

本书的特点和优势

（1）内容全面而翔实。本书不仅介绍了 Linux 的安装与基本使用，C 语言，Vi、Emacs、gcc、gdb、make 等开发工具，而且结合大量程序实例介绍了 Linux 的系统编程、网络编程和图形界面开发。此外，还介绍了模块化程序设计思想、软件测试以及编写安全的代码方法。本书光盘中的附录部分还介绍了程序员容易忽略但却十分重要的编程规范，以及大规模项目开发中必需的

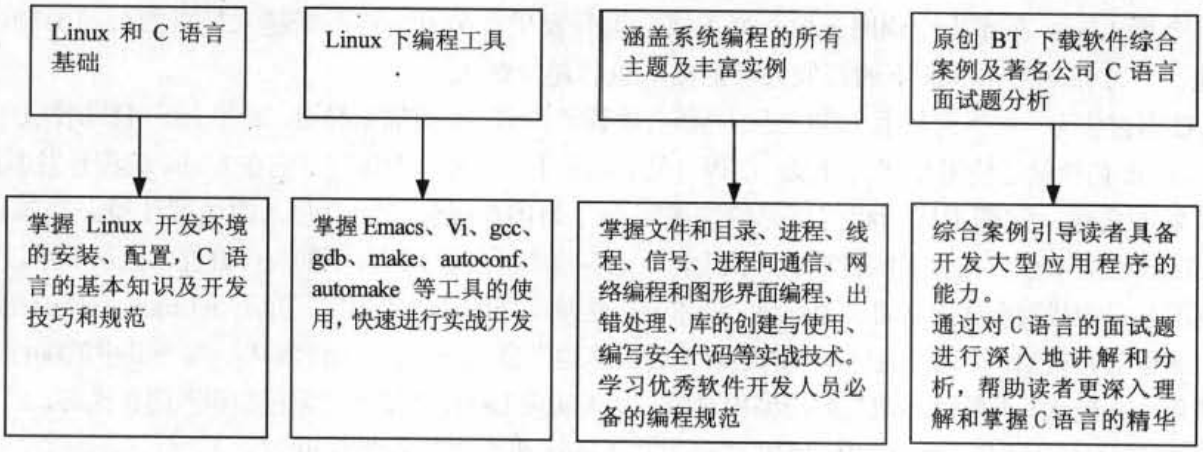
我爱自由 5ifreedom.com

autoconf 和 automake 工具的使用。

（2）理论和实际相结合，强调实践性和实用性。对于一些关键概念，如进程、线程、文件结构等知识，都以实践性很强的应用实例加以讲解，重要章节的最后都配有习题以供读者上机实践，加深理解和应用。此外，还精选了一些 C 语言的面试题，进行深入地讲解和分析，以期能使读者更深入地理解和掌握 C 语言的精华。

（3）强调后续能力的发展。在重点章节里增加了“进一步学习建议”一节，对读者在某一方面的深入学习提供建议并附以相应的参考资料。

本书知识结构和实现目标



阅读建议

- 对于没有或只有少量 Linux 操作系统使用经验和 C 语言编程基础的读者，建议按章节顺序完整阅读，并且在每章阅读过程中多上机操作、多动手编写代码。
- 对于已经懂得安装和使用 Linux 操作系统的读者，可以简单浏览第 1 章或直接跳过。
- 对于已经掌握 C 语言的读者，可以略过第 2～5 章中介绍 C 语言的内容，但建议还是要阅读指针和面试题选部分。
- 对于已经具备一定 Linux 下编程经验的读者，可以根据需要有选择地阅读。对于此类读者，建议阅读面试题选，开发工具的详细使用方法，出错处理，如何编写安全的代码，编程规范等内容。

读者对象

本书适合有一定的 C 语言基础，需要在 Linux 系统上编程的程序设计人员阅读，也可作为大中专院校师生的教材或参考书，还可供广大计算机爱好者学习使用。

随书的光盘包括：全部源代码及相关学习资料。

本书由童永清编著，参与代码调试和资料整理的有邵林、吴红娟、师轲、胡旭松、沈孝峰、郭轶、陈良华、代星科、李桂林、刘晓鹏、汪剑，在此一并表示感谢。由于水平有限加之时间仓促，书中难免存在错误之处，恳请读者批评指正。作者联系邮箱为：ht2049@163.com，或与本书编辑联系：zhangtao@ptpress.com.cn。

编　者

2008 年 1 月

我爱自由 5ifreedom.com

目　录

第一篇　Linux 和 C 编程基础

第 1 章　Linux 系统概述 …… 2

1.1　Linux 操作系统介绍 …… 2

1.1.1　Linux 的发展历程 …… 2

1.1.2　Linux 的特性 …… 2

1.1.3　Linux 的内核版本和发行版本 …… 3

1.2　C 语言简介 …… 4

1.2.1　C 语言的发展历史 …… 4

1.2.2　C 语言的特点 …… 5

1.3　Linux 的安装、启动与关闭 …… 5

1.4　Linux 的基本使用 …… 5

1.4.1　Linux 终端 …… 5

1.4.2　Linux Shell …… 5

1.4.3　Linux 的常用命令 …… 6

1.5　Linux 下程序的开发环境和开发过程 …… 9

1.6　习题 …… 12

第 2 章　C 编程基础、Vi 和 Emacs 编辑器 …… 13

2.1　C 程序的结构 …… 13

2.2　C 语言的基本数据类型 …… 14

2.2.1　整型 …… 14

2.2.2　浮点型 …… 16

2.2.3　字符型 …… 17

2.3　运算符和表达式 …… 19

2.3.1　算术运算符与算术表达式 …… 20

2.3.2　赋值运算符与赋值表达式 …… 22

2.3.3　逗号运算符与逗号表达式 …… 22

2.4　标准输入输出函数 …… 23

2.4.1　字符输出函数 putchar …… 23

2.4.2　字符输入函数 getchar …… 24

2.4.3　格式化输出函数 printf …… 24

2.4.4　格式化输入函数 scanf …… 26

我爱自由 5ifreedom.com

2.5 Vi 编辑器的使用 …… 27
2.5.1 Vi 的工作模式 …… 27
2.5.2 启动 Vi …… 28
2.5.3 保存文件和退出 Vi …… 29
2.5.4 光标的移动 …… 29
2.5.5 文本的删除 …… 30
2.5.6 文本查找和替换 …… 30
2.5.7 文本的复制与粘贴 …… 30
2.6 Emacs 编辑器的使用 …… 31
2.7 命名规范 …… 33
2.7.1 标识符 …… 33
2.7.2 关键字 …… 33
2.7.3 命名规范 …… 33
2.8 面试题选与实例精讲 …… 34
2.8.1 面试题选 …… 34
2.8.2 实例精讲 …… 35
2.9 习题 …… 37
第 3 章 C 程序控制结构和 gcc 编译器 …… 38
3.1 C 程序的控制结构 …… 38
3.1.1 C 程序语句概述 …… 38
3.1.2 C 程序的 3 种基本控制结构 …… 39
3.2 分支控制结构 …… 40
3.2.1 关系运算符与关系表达式 …… 40
3.2.2 逻辑运算符与逻辑表达式 …… 41
3.2.3 if 语句 …… 42
3.2.4 switch 语句 …… 44
3.3 循环控制结构 …… 46
3.3.1 while 语句 …… 46
3.3.2 do...while 语句 …… 47
3.3.3 for 语句 …… 48
3.3.4 break 语句和 continue 语句 …… 50
3.4 gcc 编译器 …… 51
3.4.1 程序的编译过程 …… 51
3.4.2 gcc 的常用选项 …… 52
3.4.3 gcc 的报错类型及对策 …… 54
3.5 面试题选与实例精讲 …… 55
3.5.1 面试题选 …… 55
3.5.2 实例精讲 …… 56
3.6 习题 …… 58

我爱自由 5ifreedom.com

第 4 章　C 函数、数组、指针和调试器 gdb ……59
4.1　函数 ……59
4.1.1　函数的定义 ……59
4.1.2　函数的调用 ……61
4.1.3　变量的访问控制和存储类别 ……64
4.2　数组 ……67
4.2.1　一维数组的定义和使用 ……67
4.2.2　二维数组的定义和使用 ……67
4.2.3　字符数组和字符串 ……69
4.3　指针 ……70
4.3.1　地址和指针 ……70
4.3.2　指针的定义和使用 ……71
4.3.3　指针和数组 ……72
4.3.4　指针和函数 ……75
4.3.5　指向字符串的指针 ……79
4.4　字符串函数 ……82
4.4.1　puts 和 gets ……82
4.4.2　strcpy 和 strncpy ……82
4.4.3　strcat 和 strncat ……83
4.4.4　strcmp 和 strncmp ……83
4.4.5　strlen ……84
4.4.6　strlwr 和 strupr ……84
4.4.7　strstr 和 strchr ……84
4.5　调试器 gdb ……85
4.5.1　启动和退出 gdb ……85
4.5.2　显示和查找程序源代码 ……86
4.5.3　执行程序和获得帮助 ……87
4.5.4　设置和管理断点 ……87
4.5.5　查看和设置变量的值 ……91
4.5.6　控制程序的执行 ……93
4.6　面试题选 ……95
4.7　习题 ……106
第 5 章　C 语言预处理、结构体和 make 的使用 ……107
5.1　C 语言预处理命令 ……107
5.1.1　宏定义 ……107
5.1.2　文件包含 ……109
5.1.3　条件编译 ……110
5.2　结构体和共用体 ……111
5.2.1　声明和引用结构体 ……111

我爱自由 5ifreedom.com

5.2.2 结构体和数组 ····· 113
5.2.3 结构体和指针 ····· 114
5.2.4 共用体 ····· 116
5.2.5 使用 typedef ····· 117
5.3 位运算 ····· 117
5.3.1 位运算符和位运算 ····· 117
5.3.2 位域 ····· 119
5.4 make 的使用和 Makefile 的编写 ····· 120
5.4.1 make 的一般使用 ····· 121
5.4.2 Makefile 文件的构成 ····· 123
5.4.3 使用变量 ····· 127
5.4.4 隐含规则 ····· 131
5.4.5 使用条件语句 ····· 132
5.4.6 使用库 ····· 133
5.4.7 make 命令参数详解 ····· 134
5.5 面试题选 ····· 135
5.6 进一步学习建议 ····· 140
5.7 习题 ····· 140

第二篇 Linux 系统编程

第 6 章 文件操作 ····· 142
6.1 系统编程概述 ····· 142
6.2 Linux 的文件结构 ····· 142
6.2.1 Linux 的文件系统模型 ····· 143
6.2.2 文件的分类 ····· 144
6.2.3 文件的访问权限控制 ····· 145
6.3 文件的输入输出 ····· 147
6.3.1 文件的创建、打开与关闭 ····· 147
6.3.2 文件的读写 ····· 149
6.3.3 文件读写指针的移动 ····· 150
6.3.4 dup、dup2、fcntl、ioctl 系统调用 ····· 152
6.4 文件属性操作 ····· 160
6.4.1 获取文件属性 ····· 160
6.4.2 设置文件属性 ····· 162
6.5 文件的移动和删除 ····· 164
6.5.1 文件的移动 ····· 164
6.5.2 文件的删除 ····· 165
6.6 目录操作 ····· 166
6.6.1 目录的创建和删除 ····· 166

我爱自由 5ifreedom.com

6.6.2 获取当前目录 ……167
6.6.3 设置工作目录 ……167
6.6.4 获取目录信息 ……168
6.7 编程实践：实现自己的 ls 命令 ……170
6.8 习题 ……176
第 7 章 进程控制 ……177
7.1 进程概述 ……177
7.1.1 Linux 进程 ……177
7.1.2 进程控制 ……179
7.1.3 进程的内存映像 ……179
7.2 进程操作 ……180
7.2.1 创建进程 ……180
7.2.2 创建守护进程 ……185
7.2.3 进程退出 ……187
7.2.4 执行新程序 ……188
7.2.5 等待进程结束 ……191
7.3 进程的其他操作 ……194
7.3.1 获得进程 ID ……194
7.3.2 setuid 和 setgid ……194
7.3.3 改变进程的优先级 ……196
7.4 编程实践：实现自己的 myshell ……197
7.5 习题 ……203
第 8 章 线程控制 ……204
8.1 线程和进程关系 ……204
8.2 创建线程 ……205
8.2.1 线程创建函数 pthread_create ……205
8.2.2 线程属性 ……207
8.3 线程终止 ……208
8.4 私有数据 ……210
8.5 线程同步 ……212
8.5.1 互斥锁 ……212
8.5.2 条件变量 ……213
8.5.3 异步信号 ……216
8.6 出错处理 ……216
8.6.1 错误检查 ……216
8.6.2 错误码 ……217
8.6.3 错误的提示信息 ……218
8.7 习题 ……219

我爱自由 5ifreedom.com

第 9 章　信号及信号处理 …… 220
9.1　Linux 信号介绍 …… 220
9.1.1　信号的来源 …… 220
9.1.2　信号的种类 …… 220
9.1.3　进程对信号的响应 …… 223
9.2　信号处理 …… 223
9.2.1　信号的捕捉和处理 …… 223
9.2.2　信号处理函数的返回 …… 227
9.2.3　信号的发送 …… 231
9.2.4　信号的屏蔽 …… 236
9.3　编程中如何获得帮助 …… 240
9.4　编程实践：应用实例 …… 240
9.4.1　实例一：信号的发送与处理 …… 240
9.4.2　实例二：信号应用于事件通知 …… 242
9.5　习题 …… 244
第 10 章　进程间通信 …… 245
10.1　进程间通信概述 …… 245
10.2　管道 …… 246
10.2.1　管道的概念 …… 246
10.2.2　管道的创建与读写 …… 246
10.2.3　管道的应用实例 …… 250
10.3　有名管道 …… 251
10.3.1　有名管道的概念 …… 251
10.3.2　有名管道的创建与读写 …… 252
10.3.3　有名管道的应用实例 …… 253
10.4　消息对列 …… 256
10.4.1　消息对列的基本概念 …… 256
10.4.2　消息队列的创建与读写 …… 257
10.4.3　获取和设置消息队列的属性 …… 260
10.4.4　消息队列的应用实例 …… 262
10.5　信号量 …… 265
10.5.1　信号量的基本概念 …… 265
10.5.2　信号量的创建与使用 …… 265
10.5.3　信号量的应用实例 …… 268
10.6　共享内存 …… 270
10.6.1　共享内存的数据结构 …… 270
10.6.2　共享内存的创建与操作 …… 270
10.6.3　共享内存的应用实例 …… 271
10.7　库的创建和使用 …… 275

我爱自由 5ifreedom.com

10.7.1 Linux 库的概念……275
10.7.2 静态库的创建和使用……276
10.7.3 动态库的创建和使用……277
10.8 进一步学习建议……279
10.9 习题……279

第三篇 Linux 网络和图形界面编程

第 11 章 网络编程……282
11.1 网络编程基本原理……282
11.1.1 网络模型与协议……282
11.1.2 地址……284
11.1.3 端口……285
11.1.4 IP 协议……285
11.1.5 用户数据报协议 UDP……286
11.1.6 传输控制协议 TCP……286
11.1.7 客户机/服务器模型……287
11.2 套接字编程……287
11.2.1 套接字地址结构……288
11.2.2 创建套接字……288
11.2.3 建立连接……289
11.2.4 绑定套接字……290
11.2.5 在套接字上监听……290
11.2.6 接受连接……291
11.2.7 TCP 套接字的数据传输……291
11.2.8 UDP 套接字的数据传输……292
11.2.9 关闭套接字……294
11.2.10 主要系统调用函数……294
11.3 一个面向连接的 Client/Server 实例……301
11.4 编写安全的代码……308
11.4.1 网络攻击……308
11.4.2 缓冲区溢出……309
11.4.3 输入检查……310
11.5 编程实践：编程实现端口扫描器实例……311
11.6 进一步学习建议……314
11.7 习题……314
第 12 章 GTK+图形界面编程……315
12.1 Linux 下的图形界面编程……315
12.1.1 Qt 和 GTK+……315
12.1.2 GTK+简介……315

我爱自由 5ifreedom.com

12.2 一个简单的例子 …… 316
12.3 消息和回调函数 …… 319
12.4 GTK+的面向对象机制 …… 320
12.5 排列控件 …… 323
12.5.1 使用 box 排列控件 …… 323
12.5.2 使用 table 排列控件 …… 325
12.6 常用控件 …… 330
12.6.1 进度条、微条按钮、组合框 …… 330
12.6.2 表格控件 …… 332
12.6.3 生成对话框 …… 333
12.6.4 使用菜单 …… 334
12.7 进一步学习建议 …… 336

第四篇 Linux 项目实践

第 13 章 项目实践：BT 下载软件的开发 …… 338
13.1 BT 软件简述 …… 338
13.2 BitTorrent 协议 …… 339
13.2.1 概要介绍 …… 339
13.2.2 基于 BT 协议的文件分发系统的构成 …… 339
13.2.3 B 编码 …… 340
13.2.4 种子文件的结构 …… 340
13.2.5 与 Tracker 交互 …… 342
13.2.6 peer 之间的通信协议 …… 343
13.2.7 关键算法和策略 …… 345
13.3 系统结构设计 …… 347
13.4 各个模块的设计和实现 …… 348
13.4.1 种子解析模块的设计和实现 …… 349
13.4.2 位图管理模块的设计和实现 …… 359
13.4.3 出错处理模块的设计和实现 …… 363
13.4.4 运行日志模块的设计和实现 …… 364
13.4.5 信号处理模块的设计和实现 …… 365
13.4.6 Peer 管理模块的设计和实现 …… 366
13.4.7 消息处理模块的设计和实现 …… 372
13.4.8 缓冲管理模块的设计和实现 …… 384
13.4.9 策略管理模块的设计和实现 …… 393
13.4.10 连接 Tracker 模块的设计和实现 …… 397
13.4.11 与 peer 交换数据模块的设计和实现 …… 403
13.4.12 主函数的设计和实现 …… 411
13.4.13 调试和测试 …… 412

我爱自由 5ifreedom.com

LINUX

第一篇　Linux 和 C 编程基础

Linux 系统概述

C 编程基础、Vi 和 Emacs 编辑器

C 程序控制结构和 gcc 编译器

C 函数、数组、指针和调试器 gdb

C 语言预处理、结构体和 make 的使用

我爱自由
5ifreedom.com

第 1 章　Linux 系统概述

本章我们将一起进入 Linux 世界，体会 Linux 下的编程环境和开发过程。首先介绍 Linux 操作系统，包括它的发展历程、特性、版本，接着介绍 C 语言的发展历史及其特点，然后介绍 Red Hat Linux 9 的安装过程和基本使用。最后以实例的方式介绍 Linux 下的 C 语言编程环境和开发过程。

1.1 Linux 操作系统介绍

1.1.1　Linux 的发展历程

Linux 是一种可以自由传播和免费使用的类似于 UNIX 的操作系统。它可以在各种硬件平台上运行并且开放全部的源代码。UNIX 作为一种经典的操作系统，最初由贝尔实验室的 Ken Thompson 和 Dennis Ritchie 于 1969 年研发成功，主要用在大型机和小型计算机上。它价格昂贵，对一般用户而言，可望而不可及。

Linux 最早是由芬兰赫尔辛基大学的一位叫 Linus Torvalds 的大学生设计的。当时他有一台 Intel 386 计算机，而他手上的 Minix（由著名的操作系统方面的教授 Andrew Tannebaum 编写的一个用于教学目的的操作系统）却不能很好地在他的计算机上运行。于是他开始修改 Minix。经过几个月的努力，基本完成了目标。此时他发现，他几乎实现了一个新操作系统的原型。

1991 年 10 月，Linus 把 Linux（意为 Linus 的 UNIX）放到 FTP 服务器上供其他人自由下载。Linus 允许他人免费使用 Linux 的源代码，并鼓励大家对其进行修改和完善。Linus 很快于当年 11 月发布了 Linux 的 0.10 版本，12 月又发布了 0.11 版本。后来许多程序员参加了对 Linux 的完善和发展工作，在全世界成千上万程序员的共同努力下，Linux 得到了长足的发展。现在 Linux 已经发展到 2.6 版，并且还在不断地改进和完善中。

Linux 具备 UNIX 的全部特性，如多任务、多用户、安全、稳定和高效。它可以在各种硬件平台上运行，拥有良好的界面，适合作为个人电脑的操作系统。此外，Linux 是免费软件，不需要支付费用就可以获得它及其源代码，并且可以根据自己的喜好和需要对它进行修改。用户通过互联网不仅可以获得 Linux 操作系统，而且还可以免费下载许多 Linux 的应用软件，而不用担心版权问题。Linux 经过剪裁还可以作为嵌入式操作系统，现在基于 Linux 的嵌入式软件开发逐渐成为一个热点。正是由于这些原因，Linux 受到广大计算机爱好者、软硬件公司，甚至一些国家政府的青睐。

1.1.2　Linux 的特性

近几年来，Linux 操作系统的用户不断增加，开发和应用也越来越广泛，这与 Linux 的良好特性是分不开的。Linux 具有 UNIX 的全部功能和特点，同时对其进行了改进和扩展。Linux 主要具有以下一些特性。

我爱自由 5ifreedom.com

1. 多用户

Linux 支持多个用户同时使用一台计算机，每个用户独立工作而不会相互干扰。用户之间可以进行会话和通信。每个用户对系统资源（如文件）拥有不同的权限，这样可以防止一个用户恶意地访问和修改或者无意中破坏其他用户的资源。

2. 多任务

支持多任务是现代操作系统的主要特点之一。它使计算机可以同时运行多个程序，而程序之间不会相互干扰。

3. 多平台

Linux 可以运行在各种 CPU 上，包括 Intel 系列、AMD 系列、SPARC、Alpha、Power PC 以及一些嵌入式系统，如 ARM。Linux 不仅可以运行于个人电脑上，而且还可以运行在各种大型机、小型机上，它既可以运行在一个 CPU 上，也可以运行在多个 CPU 上。它在 32 位的 CPU 上运行良好，在 64 位的 CPU 上也表现优异。

4. 良好的用户界面

Linux 向用户提供了两种界面：字符界面和图形界面。在字符界面中，用户通过输入命令来使用计算机，可以编写功能强大的 Shell 脚本，Linux 还为用户提供了优美的图形界面，通过使用鼠标操作窗口、菜单、滚动条等来方便地使用系统。

5. 强大的网络功能

内置的网络功能是 Linux 的一大特色。这使得 Linux 在通信和网络方面的功能优于其他操作系统。

6. 安全、稳定和高效

Linux 在开发过程中非常重视系统的安全性，采用了各种措施来保护系统的安全。由于 Linux 的稳定和高效，越来越多的服务器采用 Linux 作为其操作系统。Linux 的稳定和高效是继承了 UNIX 的结果。

7. 良好的可移植性

Linux 可以方便地从一个硬件平台移植到另一个硬件平台上。它既可以运行在嵌入式设备、PC 机上，也可以运行在小型机、大型机上。

8. 开放和免费

这可能是 Linux 的最大特点。Linux 是一种开放的、免费使用的操作系统，相比之下 Windows 是封闭的，有偿使用的系统。Windows 系列的操作系统是受版权保护的，其设计和开发都由微软公司一手控制，它的源代码不开放，因此我们很难得知其内部的实现。Linux 的源代码可以方便地从网络上下载，其安装盘中也有完整的 Linux 源代码。这对于操作系统爱好者、应用软件开发者和许多软硬件厂商来说，是令人兴奋的，他们可以研究 Linux 内核，并在其基础上开发自己的 Linux 系统或应用软件。

1.1.3 Linux 的内核版本和发行版本

任何软件都有其版本，如目前 Windows 系列操作系统的主要版本有 Windows2000、Windows XP、Windows 2003、Windows Vista，版本管理也是软件开发中的一个重要内容。Linux 当然也不例外，Linux 的版本分为两类：内核版本和发行版本。内核版本是指 Linux 的创始人 Linus 领导的开发小组所开发的操作系统内核的版本号，如 2.4.20。通常在内核版本号之后还会附加一个数字，

我爱自由 5ifreedom.com

如 2.4.20-8，最后的数字用来表示该版本内核是第几次被修订的。

Linux 的内核版本号由 3 部分组成：主版本号，次版本号，次次版本号。如内核版本 2.4.20，2 是主版本号，4 是次版本号，20 是次次版本号。当内核有重大改动时，主版本号会加 1；当内核只是小改动，如加入一些新的特性，支持更多的硬件，次版本号会加 1；次次版本号的增加只表示内核有轻微的改动，对内核的影响很小。次版本号为奇数表示该版本是测试版，可能不是很稳定，若为偶数则表示是个稳定版本，普通用户可以放心使用。如 2.4、2.6 是稳定版本，而 2.5、2.7 是开发中的测试版本。

内核只实现了操作系统最关键的部分，只有在此基础上提供用户界面，增加一些应用软件，一般用户才能方便地使用它。一些公司或组织将 Linux 内核和常用的应用软件包装起来，并提供安装界面和管理工具，这样就形成了 Linux 的发行版本。对于 Linux 的初学者，发行版本的概念可能要重要一些。发行版本经过了严格的测试，而且还加入了一些常用的应用软件（如字处理软件、播放器）和开发工具（如 gcc、gdb），这样初学者可以很快适应 Linux 环境并享受 Linux 带来的快乐。

Linux 的主要发行版本有：Fedora Core、Red Hat Linux、Debian Linux、SuSe Linux 和 Red Flag Linux。Red Hat Linux（即红帽 Linux）是 Red Hat 软件公司发布的 Linux 版本，无论在国内还是在国外都有很高的使用率。Red Hat Linux 中使用较多的是 Red Hat Linux 9 和 Red Hat Enterprise Linux 4，分别是个人版和企业版。前者基于 2.4 版的内核而后者基于 2.6 版的内核。Fedora Core 从 Red Hat Linux 的个人版发展而来，其构成和使用方法与红帽 Linux 基本一致，增加了一部分新特性。特别要说明的是 Red Flag Linux（即红旗 Linux）是国产的 Linux，它是全中文化的 Linux 发行版本。

本书基于 Red Hat Linux 9 讲解 Linux 下的 C 编程，当然读者若安装其他版本的 Linux 也不会影响学习。建议安装 Red Hat Linux 9 或 Fedora Core 6。

1.2 C 语言简介

1.2.1 C 语言的发展历史

C 语言最早由贝尔实验室的 Dennis Ritchie 设计并实现。Dennis Ritchie 也是 UNIX 操作系统的主要设计者之一。C 语言是目前国际上广泛使用并具有良好发展前途的计算机语言。它不仅可以用来编写应用软件，也可以用来开发系统软件，Linux、UNIX 操作系统本身都是用 C 语言开发。C 语言也是 Windows 操作系统的主要开发语言。

在 C 语言诞生之前，操作系统和其他软件主要是用汇编语言来开发的。由于汇编语言依赖于计算机硬件，且可读性、可移植性很差，开发效率也不高，而当时一般的高级语言很难对计算机硬件直接进行操作。因此，人们需要一种既有汇编语言的特性又具有一般高级语言特点的计算机语言，C 语言就在这种情况下诞生了。

后来在 C 语言的基础上，人们又开发出了 C++语言、Java 语言、C#语言。它们都是在 C 语言的语法和基本结构上，通过加入新的元素和思想开发出来的。目前 C 语言在系统软件（包括驱动程序），应用软件和嵌入式软件领域被广泛使用。

我爱自由 5ifreedom.com

1.2.2 C 语言的特点

C 语言主要具有以下特点。

1. 两重性

C 语言既可以像汇编语言一样对位、字节、地址以及硬件进行操作，又具有一般高级语言的基本结构和语句。

2. 结构化

结构化语言的一个显著特点是所开发的程序可以实现模块化。模块化是指程序的各个部分除了必要的信息交流外相对独立，因此各个部分可以单独开发和测试，提高开发效率，所开发的软件也易于维护。

3. 与 Linux 紧密结合

Linux 操作系统本身是由 C 语言开发的，在 Linux 上用 C 语言开发的程序运行效率很高，可以实现无缝结合。相比之下，用 Java 语言编写的程序虽然可以在各种软硬件平台上运行，如基于 Intel 的 Linux，基于 AMD 的 Windows。但 Java 程序的运行依赖于虚拟机，所以相对来说运行效率不高。因此，在某些对性能要求很高的领域，C 语言是首选。

4. 可移植性好

用 C 语言编写的程序基本上不用作任何修改，就可以在不同的硬件平台和操作系统上运行。

1.3 Linux 的安装、启动与关闭

如果已经成功安装了 Linux 并且能熟练地启动和关闭系统，可以跳过此内容。由于内容较简单，限于篇幅，安装和启动与关闭内容见附带光盘。本内容示范在虚拟机中安装 Red Hat Linux 9，安装 Fedora Core 6 的过程与此类似。

1.4 Linux 的基本使用

1.4.1 Linux 终端

Linux 把显示器和键盘合称为终端，因为它们可以对系统进行控制并且可以用软件的方式来实现，所以又称为虚拟控制台。虚拟控制台使 Linux 成为一个真正的多用户多任务操作系统，Linux 既可以在本地计算机上打开多个控制台，也可以在远程终端上打开多个控制台。每个控制台上可以同时运行多个任务（即运行多个程序）。在终端上，通过输入 Shell 命令来控制和使用计算机。

1.4.2 Linux Shell

Shell 是一个命令解释器，它通过接受用户输入的命令来启动、暂停、停止程序的运行或对计算机进行控制，在终端上输入 halt 命令就可以关闭计算机。Shell 还允许用户编写由 Shell 命令组

我爱自由 5ifreedom.com

成的功能强大的程序。在 Linux 上任何通过图形方式实现的操作都可以由相应的命令来完成。虽然 Linux 提供了比较完善的图形操作方式，但熟悉和掌握常用的 Linux 命令和 Shell 编程是非常有用的，而且在某些情况下是必需的。

Shell 有两种提示符：#和$。以“#”为提示符表明该终端是由 root 用户打开的，root 用户具有系统最高权限，因此可以输入任何可用的命令，而一般用户打开的终端的提示符是“$”，比如 halt 命令只能由 root 使用，普通用户在“$”提示符下输入 halt 命令，系统认为是一个无效命令。提示符的其他部分分别表示[登录用户名@主机名当前目录]。

Shell 命令的基本格式是：

命令名　[选项]　<参数 1>　<参数 2>……

其中方括号中的选项对命令来说是可选的，一条命令可以有 0 个或多个参数。选项是对命令的特别定义，也可以理解为选项告诉命令具体做什么。选项通常以“-”开始，后接一个或多个字母，如 ls-al，有的选项以“--”开头，后面一般跟着一个单词，比如--number。很多“--”格式的选项可以用“-”加上第一个字母来替代，如--number 可以用-n 来替代。

在 Shell 下输入相应的命令并按回车键，Shell 就执行命令。如果没有此命令，Shell 会提示“command not found”，表明没有这个命令。Shell 命令是区分大小写的，一条命令只要有一个字母的大小写发生变化，系统就认为是一条不同的命令。可以用分号“;”来连接多个命令，Shell 会依次执行这些命令。输入命令、目录名或文件名的开头一个或几个字母后按下<Tab>键，Shell 会在相应目录里进行匹配，自动补齐命令、目录名或文件名。在命令、目录名或文件名很长或难以记忆时，自动补齐功能会很有用。还可以通过按<↑>或<↓>键来显示执行过的命令，这在重复执行某些命令时会给用户带来很大的便利。

1.4.3　Linux 的常用命令

在介绍 Linux 常用命令前，先简单介绍一下 Linux 的文件和目录结构。

大多数的操作系统都有文件的概念，Linux 也不例外，文件是一组被命名的存储在某种介质（如硬盘，光盘，U 盘）上的信息的集合。在 Linux 中，文件是一个非常重要的概念，除了硬盘上存储的文件外，Linux 还把显示器、键盘、打印机等输入输出设备以及网络接口都当作文件来处理。文件名是文件的标识，它是一个可以包含字母、数字、下划线和句点的字符串。Linux 要求文件名的长度一般不能超过 255 个字符。句点后面的部分被称为扩展名，扩展名可以用来对文件进行分类，如 C 语言的源程序通常以.c 作为扩展名。

通常系统中有大量的文件，Linux 系统以目录的方式来组织和管理系统中所有的文件。目录在 Windows 操作系统中被称为文件夹。目录是一种特殊的文件，用来管理和组织系统内大量的文件。目录文件中的内容是存储在该目录下的文件的一些信息，如文件名、文件大小等。Linux 系统采用树型目录结构来组织和管理系统中所有的文件。以根目录“/”为起点，根目录下有许多文件和子目录，子目录下又有许多文件和子子目录，一个典型的 Linux 系统树型目录结构如图 1-1 所示。根目录下有一系列的子目录，home 目录下有 3 个子目录，tyq 下有一个文件和一个目录。

在树型目录结构中，文件和目录都通过路径来表示。路径有两种表示方法：一种是从根目录开始，称为绝对路径；一种是从当前目录开始，称为相对路径。如为了标识 test.c 这个文件，可以用绝对路径/home/tyq/test.c 来表示；如果当前目录位于 home 下，则可以用 tyq/test.c 来标识。

我爱自由 5ifreedom.com

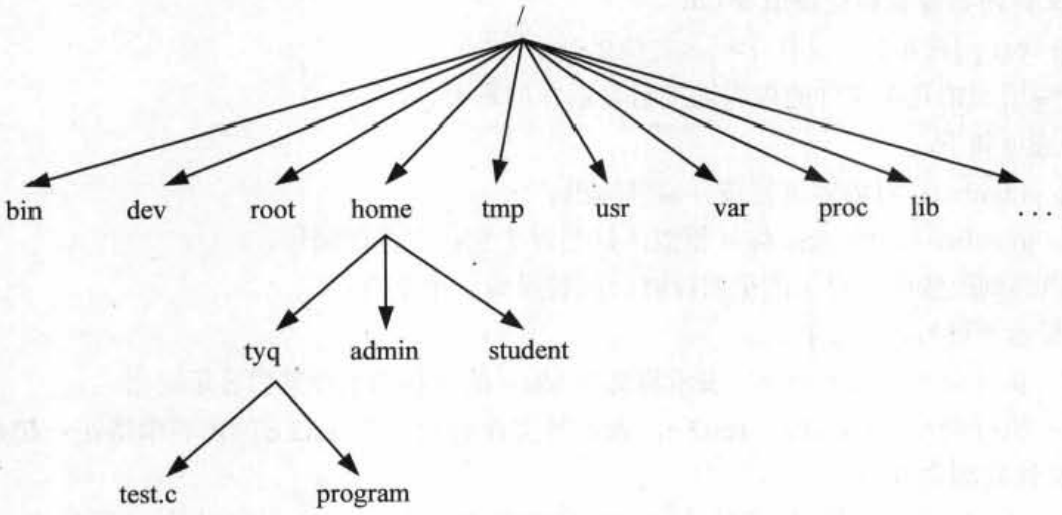


图 1-1　Linux 的目录结构

当登录到 Linux 或打开一个终端时，会进入一个特殊的目录，称为主目录。比如 root 用户登录到系统中时，系统默认进入/root 目录，/root 目录就是 root 用户的主目录。主目录可以用“～”来表示。当前目录就是用户当前所处的目录，用户在操作时可以改变当前目录，初始情况下主目录就是当前目录。当前目录可以用“.”来表示，当前目录的父目录可以用“..”来表示。如当前目录处于 home 下，也可以用相对路径./tyq/test.c 来表示 test.c 文件，这里“.”即表示“/home”。

Linux 系统中可以使用通配符“*”、“?”来同时引用多个文件。通配符“*”代表文件名中任意的字符或字符串，如 abc*表示所有以 abc 开头的文件。通配符“?”表示任意一个字符，如 abc?表示所有以 abc 开头的长度为 4 个字符的文件。

Linux 中有如下常用命令

1．查看当前目录命令 pwd

格式：pwd

例如：执行命令 pwd，系统显示当前目录。

2．改变当前目录 cd

格式：cd　<目录名>

例如：执行命令 cd　/home/admin，则切换到目录/home/admin 下。

例如：执行命令 cd～，则当前目录位于用户的主目录下。

例如：执行命令 cd..，切换到当前目录的父目录。

3．显示目录或文件信息命令 ls

格式：ls　[选项]　<目录或文件名>

主要选项如下。

-a：显示所有的文件和目录。

-l：以长格式显示文件信息。

-t：将结果按修改时间进行排序，新的文件或目录排在前面。

-R：若目录下有文件，则列出目录下的所有文件。

例如：执行命令 ls　-al　/，显示根目录下所有文件的完整信息。

例如：执行命令 ls　-at　a*，显示当前目录下所有以 a 开头的文件，新的文件排在前面。

我爱自由 5ifreedom.com

4．文件内容查看和连接命令 cat

格式：cat [选项] <文件 1> <文件 2> ……

该命令用于把几个文件的内容连接后显示在屏幕上。

主要选项如下。

-n 或—number：对内容进行按行编号输出。

-b 或—number-nonblock：与-n 相似，只是对于空行不进行编号。

-s：当遇到连续两行以上的空白行时，就替换为一个空白行。

-v：显示不可打印的字符。

例如：执行命令 cat -n test.c，表示将文件 test.c 的内容加上行号后显示输出。

例如：执行命令 cat test1.c test2.c，表示将文件 test1.c 和 test2.c 的内容串接在一起并输出。

5．文件复制命令 cp

格式：cp [选项] <源文件或目录> <目标文件或目录>

该命令用于把一个文件拷贝到另一个文件或将多个文件拷贝到一个目录下。

主要选项如下。

-r：若源目录下还有文件或目录，则都拷贝到目的地。

-f：若目标目录下已经有同名的文件，则把它删除并把源文件拷贝到目标目录下。

例如：执行命令 cp test.c tmp/program.c，表示把当前目录下的 test.c 文件拷贝到/tmp 目录下并命名为 program.c。

例如：执行命令 cp *.c /tmp，表示把当前目录下所有以.c 结尾的文件拷贝到/tmp 目录下。

6．文件删除命令 rm

格式：rm [选项] <文件或目录>

该命令用于删除文件，若加上-d 选项则可以删除指定的目录。

主要选项如下。

-i：删除前先询问要求确认。

-r：若目录非空，则删除目录下所有的文件。

-f：强制删除。

例如：执行命令 rm *.c，表示删除当前目录下所有以.c 结尾的文件。

例如：执行命令 rm -rf tyq，表示强制删除 tyq 目录下所有的文件和子目录，子目录下的文件或目录都被删除。

7．移动或重命名文件命令 mv

格式：mv [选项] <源文件或目录> <目的文件或目录>

该命令用于对一个文件或目录进行重命名或将几个文件移到另一目录。

主要选项如下。

-I：移动前先询问要求确认。

-f：强制移动，若目标目录下有同名的文件则覆盖它。

例如：执行命令 mv test.c program.c，则将当前目录下的 test.c 文件重命名为 program.c。

例如：执行命令 mv -i *.c /tmp/project，则将当前目录下所有以*.c 结尾的文件移到 project 目录下，若目标目录下已有同名文件则先询问是否覆盖。

我爱自由 5ifreedom.com

8．创建目录命令 mkdir

格式：mkdir [选项] <目录>

如果指定目录不存在则创建它。

主要选项如下。

-p：若要建立的目录的上层目录不存在，则先创建它。

例如：执行命令 mkdir -p dir1/dir2，表示在当前目录下创建 dir1/dir2，如果 dir1 不存在则先创建它。

9．删除目录命令 rmdir

格式：mkdir [选项] <目录>

如果指定目录为空就删除它，若不为空则出现错误信息。

主要选项如下。

-p：当删除指定目录后，若该目录的父目录为空，则也将其删除。

例如：执行命令 rmdir -p dir1/dir2，表示在当前目录下的 dir1 目录中删除 dir2 子目录，若删除 dir2 后 dir1 变为空目录，则也将其删除。

另外，可以使用“touch filev.c”命令在当前目录下创建一个文件，前提是 file.c 文件不存在。可以在命令名后加--help 来获得帮助信息，如“mkdir --help”命令用于获取 mkdir 使用方法的帮助信息。

1.5 Linux 下程序的开发环境和开发过程

本节通过一个简单的例子来介绍 Linux 下程序的开发环境和开发过程。Linux 下的开发环境主要有两类：字符界面的开发环境和图形化的集成开发环境。

在字符界面下的开发环境中，一般使用 Vi、vim 或 Emacs 文本编辑器来编写源程序，然后使用 gcc 编译器来编译程序，当程序出现错误而不能实现既定的功能时，使用 gdb 调试器来调试程序。如果开发的是一个大型程序，可能需要编写 Makefile 文件来自动编译程序，并使用 CVS 对项目进行管理。

1．使用 Emacs 编写 C 源程序

Emacs 是 Linux 下一个功能强大的图形化文本编辑器，可以用来编写 C 源程序。其使用方法非常简单，与 Windows 下的记事本差不多，但功能比它强大得多。

首先打开 Emacs 文本编辑器，单击 Linux 桌面左下角的“红帽”图标，再选择“编程”，然后单击“Emacs”选项，如图 1-2 所示。单击工具栏上的“打开或新建文件”按钮，并在命令行输入/tmp/test.c，就在/tmp 目录下新建了一个名为 test.c 的源程序文件。

输入源程序代码如例 1-1 所示。

例 1-1 test.c

```
#include<stdio.h>
int main()
{
   printf ("Welcome to Linux Programming\n");
   return  0;
}
```

单击工具栏上“保存”按钮，并关闭 Emacs。

我爱自由 5ifreedom.com

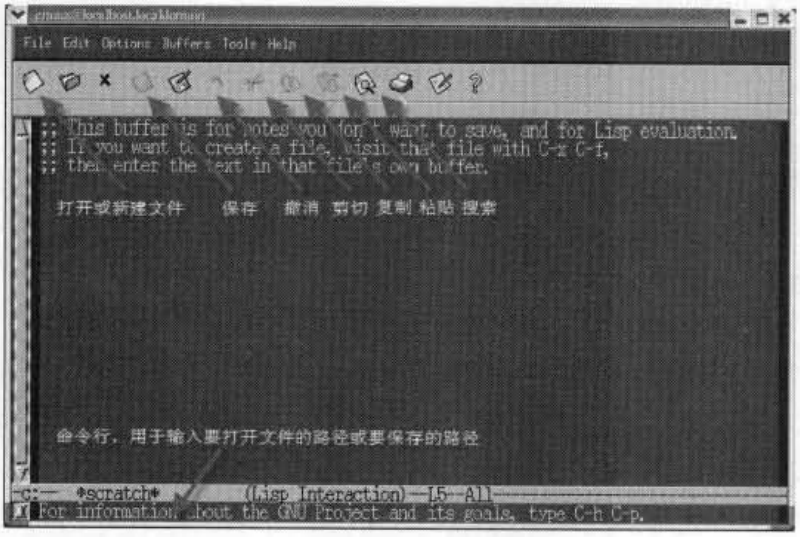


图 1-2 Emacs 编辑器

2. 使用编译器 gcc 编译 C 源程序

打开终端，先切换到/tmp 目录下，然后编译程序，再输入命令 ls，会发现/tmp 目录下生成了一个可执行文件 test，相应的命令如下：

```
[root@local host root]# cd /tmp
[root@local host tmp]# gcc -o test test.c
[root@local host tmp]# ls
test test.c
```

3. 运行程序

```
[root@local host tmp]# ./test
```

运行结果是在屏幕上打印“Welcome to Linux Programming”。

Linux 也提供了许多图形化的集成开发环境，这里介绍 KDevelop 集成开发环境的基本使用。集成开发环境整合了文本编辑器、编译器、调试器以及显示执行结果等诸多功能，KDevelop 还提供功能完备的帮助功能。在集成开发环境下可以完成程序编写、编译、调试和执行的所有动作，适合大型程序的开发。

要运行 KDevelop，必须安装 KDE 桌面系统。如果没有安装，请先安装，方法是单击 Linux 桌面左下角“红帽”图标，选择“系统设置”，再单击“添加/删除应用程序”，然后根据提示进行操作即可。单击“红帽”图标，选择“编程”，再单击“KDevelop”就可以启动 KDevelop，如图 1-3 所示。如果是第一次启动，先要进行一些设置，一般选择默认选项直接单击“下一步（N）”按钮即可。

图 1-3 KDevelop 的欢迎界面

设置完成后出现如图 1-4 所示的主界面。

在 KDevelop 中开发 C 程序，需要先创建项目，单击“项目”，再单击“新建”选项，打开如图 1-5 所示的“应用程序向导”对话框。

我爱自由 5freedom.com

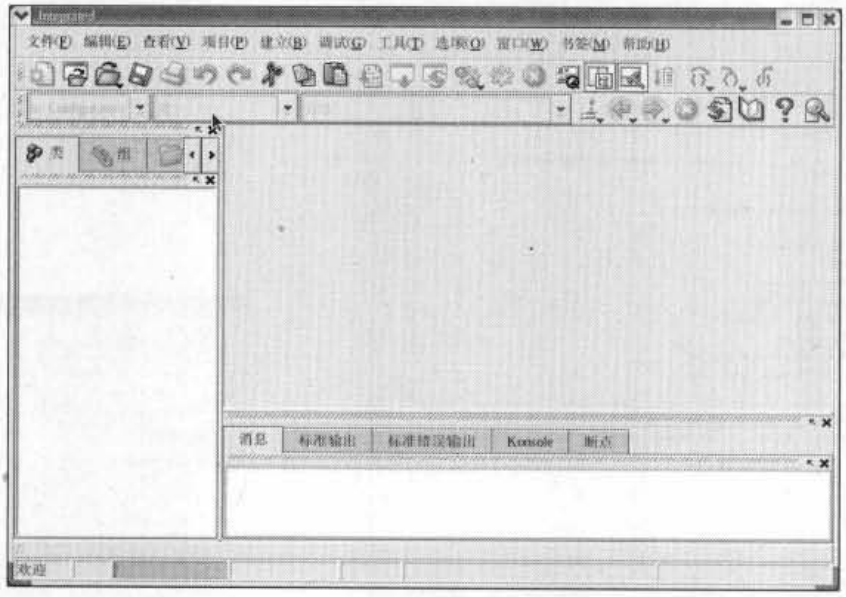


图 1-4　KDevelop 主界面

在“应用程序向导”对话框中，选择要创建的程序的种类，这里选择“C 程序”，单击“后一页（N）”按钮。打开“常规设置窗口”如图 1-6 所示。

图 1-5　应用程序向导

图 1-6　常规设置

在常规设置窗口中的“项目名称”中输入项目的名称，可根据情况设置项目目录。取消“生成源文件和头文件”、“用户文档”等默认已选择的选项。这里创建的是一个简单项目，不需要这些选项。完成后单击“后一页（N）”按钮。随后出现的“版本控制系统支持”、“.h 文件的头模板”和“.c 文件的头模板”等对话框，都默认设置并直接单击“后一页（N）”按钮，显示项目文件创建过程，如图 1-7 所示。

单击“创建”，开始创建项目相关文件，当最后显示“READY”字样时表示项目相关文档已创建完成，单击“退出”按钮。

在 KDevelop 主界面的左边选择“文件”选项卡，在项目名“Project1”上单击右键，再单击“新建文件...”选项，弹出新建文件对话框，如图 1-8 所示。

我爱自由 5ifreedom.com

图 1-7　项目文档创建过程

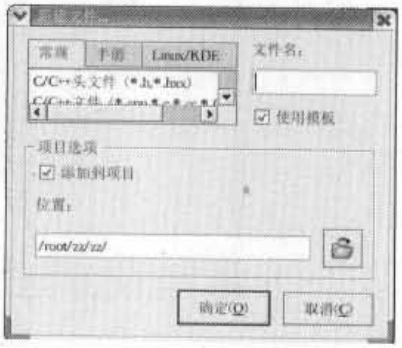


图 1-8　新建文件对话框

选择“C/C++文件”选项，并输入文件名 main.c，去掉“使用模板”默认设置，单击“确定”按钮之后就在项目中添加了一个名为 main.c 的源文件，单击“Project1”前的加号可以看到该文件，双击打开它并输入例 1-1 源程序。

源程序输入完成后，在 KDevelop 主界面上选择“建立（B）”选项，单击“清除/全部重建（C）”，开始编译源程序，编译完成后再单击“建立(B)”下的“执行(E)”选项，结果打印出“Hello World!”字样，如图 1-9 所示。

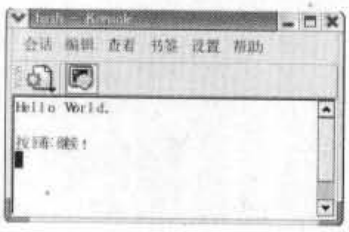


图 1-9　显示运行结果

KDevelop 虽然功能非常强大，但很多功能是为开发 C++程序、图形界面程序特别是为开发工程项目设计的，对于简单的 C 语言程序，很多功能是用不到的。

1.6 习题

1. Linux 的特性有哪些？
2. Linux 的发行版本主要有哪些？发行版本与内核版本有什么区别和联系？
3. C 语言有哪些特点？
4. 选择一种安装方式，安装 Linux。
5. 如图 1-31 所示是一个 Linux 目录结构，假设 program 目录下有一个文件 file.c，当前目录是 tyq，用户的主目录是 home，请写出 3 种 file.c 文件的路径。
6. 在 Linux 上练习使用 1.4 节所列的常用命令。
7. 在内存中，地址是什么，指针又是什么？
8. CPU、内存、硬盘、总线、输入输出设备是如何协同工作来完成一个程序的执行的？
9. 使用 Emacs 输入 1.6 节的例 1-1 源程序，并用 gcc 编译运行。
10. 使用 KDevelop 建立一个项目，输入 1.6 节的例 1-1 源程序，并编译运行。

我爱自由 5ifreedom.com

第 2 章　C 编程基础、Vi 和 Emacs 编辑器

本章将介绍 C 语言中几个重要概念：常量与变量、数据类型、运算符和表达式。在介绍这些概念之前，先了解一下 C 语言程序的基本结构，以便于在本章的学习中编写测试程序。本章将结合 Linux 环境，详细介绍 Emacs 和 Vi 编辑器的使用。为了贴近实用，本章还将介绍命名规范和一些国内外知名公司的面试题。

本章重点：

- 3 种基本数据类型。
- 运算符和表达式。
- 标准输入/输出函数。
- Vi 编辑器。
- Emacs 编辑器。
- 命名规范。

本章难点：

- 一些复杂表达式的含义。
- Vi 与 Emacs 编辑器的使用。
- 标准输入输出函数的使用。

2.1 C 程序的结构

本节以例 2-1 来简单说明 C 程序的结构。

例 2-1

```
#include <stdio.h>

int main( )
{
    int  x,  y,  sum;  /*定义了 3 个变量*/

    x=100;
    y=200;
    sum=x+y;

    printf("sum is %d", sum);  /*输出结果*/
}
```

程序输出：

```
sum is 300
```

程序说明。

1. 一个程序通常会包含一个或多个头文件，本程序的第一行就包含了一个名为 stdio.h 的头文件。该头文件对一些输入输出函数进行了声明。函数是具有一定功能的，由函数头和函数体组

我爱自由 5ifreedom.com

成的语句块。

2．每个程序必须有且只能有一个主函数 main。本程序中，main 函数的函数头是 int main()，由一对大括号括起来的内容是函数体。在本程序中，main 函数体由一对大括号和 5 条语句组成。

3．C 语言中以分号“;”来表示一个语句的结束。main 函数中的第一条语句定义了 3 个变量，变量名（用于标识一个变量）分别为 *x*，*y* 和 *sum*。第二条、第三条语句分别给变量 *x*，*y* 赋值，100 和 200。第四条语句把变量 *x* 和 *y* 的值相加，并把结果赋给 *sum*。最后一条语句用来打印变量 *sum* 的值，用双引号引起来的内容是要输出的内容，其中%d 是一个格式字符串（关于格式字符串知识，本章后面部分将会讲解），表示该位置将输出一个 10 进制的变量值，后面的 *sum* 就是要输出的变量。

注意：双引号内的 *sum* 只是 3 个字母，不是变量，它会原样输出到屏幕上。

4．为了增加程序的可读性，通常会加入一些注释。C 语言规定“/*”与“*/”之间的内容是注释。注释就是对程序作的一些说明，它对程序的编译和运行不起作用。在程序中通常还有一些空行，主要用来分割程序，增加可读性，如在本程序中就有两个空行。空行可有可无，但适当的空行可以使程序的结构变得清晰。

5．C 程序书写格式比较自由，可以在一行上写几条语句，也可以把一条语句分成几行来写。但是通常一行只写一条语句，如果一条语句太长，则把它分成几行来写。

2.2 C 语言的基本数据类型

数据以一定的类型和格式存储在计算机中，数据的类型可以是整数、实数、字符型，也可以是图形、图像、声音、视频。在C语言中，数据类型可分为基本数据类型、构造数据类型（包括数组、枚举类型、结构体和联合体）、指针类型、空类型四大类。本节介绍 C 语言的基本数据类型：整型、浮点型和字符型。C 语言中数据还有常量和变量之分，常量是指在程序运行过程中其值不能被改变的量，而变量则可以改变。

2.2.1 整型

整型常量即整常数，C 语言中常用的整型常量有：十六进制、八进制和十进制。

十六进制整常数以 0x 为前缀，其数字取值为 0～9，A～F（或 a～f，分别对应数字 10～15）。如 0x1A6，其对应的十进制值为 $1 \times 16^2 + 10 \times 16^1 + 6 \times 16^0 = 422$。

以下各数是合法的十六进制整常数。

0x25（十进制为 37），0xB1（十进制为 177），0xFFFF（十进制为 65535）。

以下各数不是合法的十六进制整常数：

12（没有前缀 0x，计算机将视为十进制的 12），0x12G（G 不是合法的十六进制数码）。

八进制整常数必须以 0 为前缀，其数字取值为 0～7。如 015，其对应的十进制数为 $1 \times 8^1 + 5 \times 8^0 = 13$。

以下各数是合法的八进制整常数。

012（十进制为 10），0101（十进制为 65），01000（十进制为 512）。

以下各数不是合法的八进制数：

256（无前缀 0），03A2（包含了非八进制数码）。

我爱自由 5ifreedom.com

十进制整常数没有前缀，其数字取值 0～9。

以下各数是合法的十进制整常数。

237，65535，1688。

以下各数不是合法的十进制整常数。

025（不能有前导 0），88D（含有非十进制数码）。

C 语言中的变量遵守“先定义后使用”的规则。下面定义了一个名为 num 的整型变量，之后赋值 100，然后又改变其值为 200。

```
int num;
num=100;
num=200;
```

int 是整型变量的基本类型，整型变量还有短整型（short int 或 short）和长整型（long int 或 long），它们的不同之处在于变量在内存中存储时所占用的字节数，通常在 32 位机上短整型为 2 字节即 16 位，基本整型和长整型占用 4 字节即 32 位。

例如，有以下语句，定义了一个短整型变量 i，并在定义时就赋予初始值 10。

```
short int i=10;
```

数据在内存中是以二进制的形式存放的，十进制数 10 对应的二进制数为 1010。图 2-1 显示了变量 i 在内存中的存放，最高位为符号位，0 表示是正数，1 表示为负数。

0	0	0	0	0	0	0	0
0	0	0	0	1	0	1	0

图 2-1　变量 i 的内存示例图

一个短整型 short int 变量的值的范围为-2^{15}～$2^{15}-1$，即−32768～32767。短整型所表示的数是很有限的，实际应用中通常使用 int 型，其值范围为-2^{31}～$2^{31}-1$，即−2147483638～2147483637。在一些实际应用中，变量的值常常为正数，如图书馆管理系统中的图书总数，企业信息系统中的公司职员数等。为了充分利用变量的表示范围，此时可以将变量定义为无符号类型。加上修饰符 unsigned 就可以定义无符号变量，如 unsigned short int i，定义了一个无符号短整型变量 i；unsigned int j 定义了一个无符号整型变量 j。默认情况下，定义的变量是有符号的，即表示的数可正可负，有符号变量也可以加上修饰符 signed，但通常情况下不加。无符号短整型由于最高位不再存储符号信息，其值范围为 0～$2^{16}-1$，即 0～65535。相应地，无符号整型所表示的数的范围 0～$2^{32}-1$，即 0～4294967295。

C 语言中并未规定各个数据类型所占内存的字节数，具体如何实现由各计算机系统决定。有的微机上短整型和整型都是 16 位，而长整型位 32 位。所占内存的字节数可以用求字节数的运算符号 sizeof 来确定。例 2-2 程序可以用于确定各数据类型所占内存的字节数。

例 2-2

```
#include <stdio.h>

int main( )
{
    short int i;
    int j;
    long int k;
    int a,b,c;

    a=sizeof(i);
```

我爱自由 5ifreedom.com

```
    b=sizeof(j);
    c=sizeof(k);

    printf("a is %d\n", a);
    printf("b is %d\n", b);
    printf("c is %d\n", c);
}
```

程序说明。

（1）允许在一条语句中定义多个相同数据类型的变量，如“int a，b，c；”，各变量名之间用逗号间隔。数据类型说明符（如 int）与变量名之间至少用一个空格隔开。

（2）“a = sizeof(i);”语句中 sizeof(i)用于求出变量 i 所占内存的字节数，并把该数值赋给变量 a。

（3）“printf(“a is %d\n”, a);”语句中的“\n”为换行符，表示另起一行开始输出，它是转义字符，不会在屏幕上原样输出。

新建一个名为 test.c 的文件，输入例 2-2 源程序，保存后进行编译，并运行，结果如下所示：

```
$ gcc -o test test.c
$ ./test
a is 2
b is 4
c is 4
```

2.2.2 浮点型

1. 浮点型常量

浮点型常量即实数，主要有两种表示形式。

（1）十进制小数形式。它由正负号、数字、小数点组成，其中小数点是必须有的，如 3.45、0.25、.78、96.、−12.5、0.0 都是合法的。

（2）指数形式。一般形式为 xEy 或 xey，表示 $x \times 10^y$，其中 y 必须是整数，如：2.15e1（即 21.5），.005E2（即 0.5），−3.6e3（即−3600）。

2. 浮点型变量

浮点型变量主要有单精度（float 型）和双精度（double 型）两种，一般它们在内存中分别占 4 个字节和 8 个字节。如下所示，定义了一个单精度浮点型变量 i 和一个双精度浮点型变量 j。

```
float  i;
double j;
```

单精度浮点型变量的取值范围约在-10^{-37}～10^{38}，有效数字是 7 位。双精度浮点型变量的取值范围约在-10^{-307}～10^{308}，有效数字是 16 位。一般在使用中，单精度浮点型即 float 型已足够，但为了增加有效位数则可以使用 double 型。下面以例 2-3 来说明有效位数的概念。

例 2-3

```
#include <stdio.h>

int main( )
{
    float  a=88888.88888;                    /*整数部分 5 位，小数部分 5 位*/
    double b=8888888888.888888888;           /*整数部分 10 位，小数部分 9 位*/

    printf("a is %f",a);
    printf("b is %f",b);
}
```

程序说明。

（1）本程序定义了两个浮点型变量，分别为单精度和双精度，并赋予了初值。

（2）“%f”表示此处输出一个浮点型变量，且小数点后最多输出 6 位。

我爱自由 5ifreedom.com

编译并运行程序结果如下所示：

```
$ gcc -o test  test.c
$ ./test
a is 88888.890625
b is 8888888888.888889
```

在本例中，由于 a 是单精度浮点型，有效位数只有 7 位。而整数已占 5 位，故小数点两位之后均为无效数字。小数部分的 9 是四舍五入的结果，它是最后一位有效位。同样的，b 是双精度型，有效位为 16 位。小数部分的 9 也是四舍五入的结果，它也是最后一位有效位。%f 规定小数点后最多保留 6 位，所以小数部分只输出了 6 位。

2.2.3 字符型

1. 字符型常量

字符常量是用单引号括起来的一个字符。如’a’,’b’,’=’,’5’,’?’都是合法字符常量。

注意：在 C 语言中，’a’和’A’是不同的字符常量，’5’是一个字符常量而 5 是一个十进制的整数。

2. 转义字符

转义字符是一种特殊的字符常量。转义字符以反斜扛“\”开头，后跟一个或几个字符。转义字符具有特别的含义，不同于字符原有的意义，故称“转义”字符。例如，在前面程序中 printf 函数中用到的“\n”就是一个转义字符，其意义是“回车换行”。转义字符主要用来表示那些用一般字符不便于表示的控制代码，表 2-1 为常用转义字符。

表 2-1　　常用转义字符表

字符形式	含　义
\n	换行，将当前位置移到下一行开头，相当于按 Enter 键
\t	跳到下一个 tab 位置，相当于按 Tab 键
\b	退格，将当前位置移到前一列，相当于按 BackSpace 键
\\	反斜扛字符\
\’	单引号字符’
\”	双引号字符”
\0	空字符，它将用在字符串中
\ddd	1 到 3 位的八进制数所代表的字符，如\101 代表字符 A
\xhh	1 到 2 位的十六进制数所代表的字符，如\x41 代表字符 A

3. 字符变量

字符型变量由关键字 char 来定义，如下所示：

```
char c1, c2='A';
```

定义了两个字符型变量 c1、c2，并为 c2 赋予了初值，即字符’A’。

一个字符型变量在内存中占一个字节（8 位），只能存放一个字符，如’a’,’=’,’\n’,’\n’是一个转义字符。字符是以 ASCII 码的形式存放在内存单元中的，每个字符对应一个 ASCII 码，如字符 A 对应的 ASCII 码为 65，字符 a 对应 97，字符 b 对应 98。因此，字符型变量的值实质上是一个 8 位的数值，其值范围为-2^7～2^7-1 即-128～127。字符型变量 char 也可以加上修饰符 unsigned 变为 unsigned char，它表示一个 8 位的无符号整数，其值取值范围为 0～2^8-1 即 0～255。

由于字符在内存中以 ASCII 码的形式存放，而 ASCII 码实际上就是一个整数，因此字符型变

我爱自由 5ifreedom.com

量可以与整型变量进行运算。例 2-4 将实现它们之间的转换。

例 2-4

```
#include<stdio.h>

int main( )
{
    int   c1,c2;
    char  c3;

    c1='a'-'A';
    c2='b'-'B';
    c3='c'-32;

    printf("c1 is %d and c2 is %d\n",c1,c2);
    printf("c3 is %d and %c\n",c3,c3);
}
```

程序说明。

（1）程序首先定义两个整型变量 c1、c2，然后再定义一个字符变量 c3。

（2）将字符 a 与 A 的 ASCII 码相减，并把结果赋给 c2。字符 a 与 A 的 ASCII 码分别是 97 和 65，因此 c1 的值为 32。之后把字符 b 与 B 的 ASCII 码相减，并把结果赋给 c2。字符 b 与 B 的 ASCII 码分别为 98 和 66，因此 c2 的值也为 32。事实上，对于 26 个英文字母，其大小写的 ASCII 码之差都是 32，因此大小写之间的转换是很容易的。

（3）将字符 c 减去 32，把值赋给 c3。c3 的值是字符 C 的 ASCII 码，即 67。

（4）程序第一行打印出 c1 和 c2 的值。第二行以整型和字符型两种方式打印出 c3 的值。“%c”表示该位置输出的是一个字符型变量的值。

运行结果如下：

```
c1 is 32 and c2 is 32
c3 is 67 and C
```

4. 字符串常量

字符串常量是由一对双引号括起的字符序列，例如：

```
"China", "Welcome to Linux\n", "How are you?".
```

字符串常量在内存中存储时，每个字符占用一个字节的内存空间，系统自动在字符串尾部加上一个字符’\0’，以标识这个字符串的结束。字符串 China 在内存中的存储形式如图 2-2 所示。

C	h	i	n	a	\0

图 2-2

事实上，字符是以 ASCII 码的二进制形式存放在内存中的。例如，字符 a 的 ASCII 码的二进制表示是 01100001，其在内存中存储形式如图 2-3 所示。

0	1	1	0	0	0	0	1

图 2-3

5. 符号常量

常量也可以有一个名字，这就是符号常量。符号常量的用法如例 2-5 所示。

例 2-5

```
#include <stdio.h>

#define PI 3.1415926
```

我爱自由 5ifreedom.com

```
int main( )
{
    float r=2.56;
    float l,s;

    l=2*PI*r;
    s=PI*r*r;
    printf("l=%f\n",l);
    printf("s=%f\n",s);
}
```

程序说明。

该程序的功能是输出一个半径为 2.56 的圆的周长和面积。程序在第二行使用预编译命令“#define”定义了一个符号常量 PI，其值为 3.1415926。在这个程序中，凡是 PI 就是指 3.1415926。在程序中，使用一个符号来代替一个常数，至少有以下优点。

（1）使程序更清晰，更可读。在阅读大量的程序代码时，数字本身并不能说明其意义。如果定义为符号常量，程序的可读性就增强了。

（2）程序的修改更方便。试想，如果该程序有许多地方都用到 π 这个值，而 π 的值只要 3.14 就可以了，那么我们只要在程序的开始把 PI 定义为 3.14 即可，而不需要在程序里找到每个 π，然后修改它的值。

2.3 运算符和表达式

C 语言中，常量、变量与运算符相结合可以构成各种各样的表达式。C 语言的运算符不仅具有不同的优先级，而且还有一个特点，就是它的结合性。在表达式中，各参与运算的常量和变量不仅要遵守运算符优先级的规定，还要受运算符结合性的制约，以便确定是自左向右还是自右向左进行运算。C 语言主要有如下这些运算符。

1. 算术运算符（+，-，*，/，%，++，—）。
2. 赋值运算符（=）。
3. 逗号运算符（,）。
4. 类型转换运算符（()）。
5. 关系运算符（>，<，==，>=，<=，!=）。
6. 逻辑运算符（!，&&，||）。
7. 条件运算符（? :）。
8. 求字节运算符（sizeof）。
9. 指针运算符（*，&）。
10. 下标运算符（[ ]）。
11. 分量运算符（.，->）。
12. 位运算符（>>，>>，~，|，^，&）。

本节将介绍前 4 种运算符及其相应的表达式，第 5、6、7 三种运算符将在第 3 章介绍，第 8 种运算符在 2.2 节已介绍过其使用方法，第 9、10 种运算符将在第 4 章的指针和数组部分加以介绍，第 11 种运算符将在第 5 章的结构体部分介绍其使用方法。第 12 种运算符由于在应用编程中一般不使用，本书不作介绍，有兴趣的读者可以参考其他资料。C 语言中运算符初看似乎很多，

我爱自由 5ifreedom.com

但熟练之后就会觉得非常简单且容易使用。

2.3.1 算术运算符与算术表达式

1. 基本的算术运算符

+：加法运算符，如 4 + 8，a + 10，这里 a 是一个变量，下同。

−：减法运算符，如 4 − 8，a − 10，−a。

*：乘法运算符，如 4*8，a*10。

/：除法运算符，如 4/8，a/10。

%：求余运算符，如 7%3，则值为 1。

这里要注意。

（1）括号也可以参与运算，如 a*（4 + 8）。

（2）两个整数相除，结果仍然是整数，如 5/3 的结果是 1，小数部分将舍去。

（3）求余运算符的两边都应为整数，如 7.5%3 是没有意义的。

（4）当整型（int，short，long）、浮点型（float，double）、字符型（char）数据混合运算时，不同类型的数据先转换为同一类型，然后再进行运算。其中 short 和 char 类型将自动转换为 int 类型，float 类型将自动转换为 double 类型进行运算。如考虑以下表达式。

① a +’A’，假设 a 是一个 int 类型的变量，值为 1000，’A’ 是一个字符型常量，两者运算时，’A’ 将转换为 int 类型，运算结果是 int 类型。那么’A’ 是如何转换为 int 类型的呢？’A’ 在内存中是以它的 ASCII 码值 65 的二进制形式存放的，占 1 个字节，而 int 是 4 字节的，运算时用 4 个字节来存储’A’ 的值 65，最低一个字节仍然存放 65 的二进制数，高出的 3 个字节将以 0 来填充。最后运算结果为 1065。

② a + b，假设 a 和 b 都是 float 型，则运算时，系统自动把 a 和 b 都扩充为 8 个字节的 double 型进行运算，产生结果后又把结果转换回 float 型，因此结果仍然为 float 型。若 float 型的数与 double 型的数进行运算则结果为 double 型。

③ 100 +’A’ −65.25，这个表达式有 int 型 100、char 型’A’、float 型 65.25，则运算时 char 型数’A’ 自动转换为 int 型，既有 int 型也有 float 型时，int 型将转换为 float 型进行运算，而 float 型在运算时会自动转换为 double 型，所以这 3 个量都转换为 double 型参与运算，最后系统把结果转换回 float 型。系统相当于进行如下运算 100.0 + 65.0 − 65.25，运算结果为 100.25。

2. 算术运算符和表达式的优先级和结合性

在表达式中，优先级较高的先于优先级较低的进行运算。而在一个运算量两侧的运算符优先级相同时，则按运算符的结合性所规定的结合方向处理。C 语言中各运算符的结合性分为两种，即左结合性（自左至右）和右结合性（自右至左）。例如算术运算符的结合性是自左至右，即先左后右。如有表达式 $x - y + z$ 则 y 应先与“−”号结合，执行 $x - y$ 运算，然后再执行加 z 的运算。这种自左至右的结合方向就称为“左结合性”。而自右至左的结合方向称为“右结合性”。最典型的右结合性运算符是赋值运算符。如有赋值表达式 $x = y = z$，由于赋值运算符“=”的右结合性，应先执行 $y = z$，即把 z 值赋给 y，再执行 $x = y$ 运算，即把 y 的值赋给 x。C 语言运算符中有少量运算符为右结合性，应注意区别，以避免理解错误。

3. 强制类型转换

可以利用强制类型转换符“()”将表达式或常量、变量转换为所需类型，例如：

我爱自由 5ifreedom.com

(double)a ：将变量 a 的值转换为 double 型。

(int)x + y ：将变量 x 的值转换为 int 型再与 y 进行运算。

(int)(x + y) ：将 x + y 的结果转换为 int 型。

(float)(11%5) ：将 11%5 的结果转换为 float 型。

注意：在进行强制类型转换时，得到的是一个所需的中间类型，原来变量的类型并未发生变化，例如：

```
double a=3.141592;
int i=(int)a;
```

则 i 的值为 3，而 a 的值仍然为 3.141592。

4．自增运算符++和自减运算符--

自增 1 运算符“++”，其功能是使变量的值自增 1。自减 1 运算符“—”，其功能是使变量值自减 1。自增，自减运算符均为右结合性。它可以有以下几种形式。

i++：先使用 i，再把 i 的值加 1。

i—：先使用 i，再把 i 的值减 1。

++i：先把 i 的值加 1，再使用 i。

—i：先把 i 的值减 1，再使用 i。

例 2-6 将实现自增运算符和自减运算符的功能。

例 2-6

```
#include<stdio.h>

int main( )
{
    int i=3,j,k;

    j=i++;
    k=++i;

    printf("i=%d,j=%d,k=%d\n",i,j,k);
    printf("%d\n",-i++);
    printf("%d,%d,%d",i,i++,i++);
}
```

程序说明。

（1）程序首先定义了 3 个变量 i，j，k，并为变量 i 赋予初值 3。

（2）第二条语句，先引用 i 的值，再将 i 加 1，等价于“j = i；i = i + 1；”。第三条语句，先将 i 加 1，再引用 i 的值，等价于“i = i + 1；j = i；”。结果 i，j，k 分别输出 5，3，5。

（3）第二条打印语句中，表达式“–i ++”相当于“– (i++)”即先引用 i 的值，则–i 等于–5，结果输出–5，之后 i 的值加 1，所以该语句执行完毕后 i 的值已变为 6。

（4）第三条语句打印出什么呢，“6，7，8”？事实上打印出的是 8，7，6。因为 printf 中 i，i++，i++的求值比较特殊，它是从右往左求值的。也就是先求最右边的 i++，此时先引用 i 的值 6，之后 i 的值加 1 变为 7，然后是中间的 i++，先引用 i 的值 7，然后把 i 的加 1 使之变为 8，最后是左边的 i，此时 i 的值是 8。

（5）++，--运算符只能对单个变量起作用，对表达式和常量是不起作用的。如“(a + b)++”，原意是先将 a 和 b 的值相加，再把结果加 1，但++操作的对象是表达式（a + b），所以该表达式是错误的。又如 8++，也是错误的，因为 8 是常量。要验证程序是否可以含有（a + b）++这样的表达式，可以写一个测试程序，其中含有类似的表达式，可以发现编译器在编译时会报告错误。

我爱自由 5ifreedom.com

程序运行结果：

```
i=5,j=3,k=5
-5
8,7,6
```

2.3.2 赋值运算符与赋值表达式

1．基本的赋值运算符“=”

赋值运算符“=”在前面的程序中已经基本介绍过它的使用方法了。它的一般形式是：

变量=表达式；

它首先计算表达式的值，再把值赋给变量。例如：

a = b = 8；把 8 赋给 b，再把变量 b 的值赋给 a，a 的值也为 8。

a = 3+（b = 7）；把 7 赋给 b，然后计算 3+b 的值，最后把结果 10 赋给 a。

a =（b = 5）/（c = 3）；把 5 和 3 分别赋给 b 和 c 后，计算 b/c，把结果 1 赋给 a，这里假定 a、b、c 为 int 型。

如果赋值运算符两边的数据类型不相同，系统将自动进行类型转换，即把赋值号右边的类型转换为左边的类型。具体规定如下。

（1）实型赋给整型，舍去小数部分。如有语句“int a = 3.14；”则 a 的值为 3。

（2）整型赋给实型，数值不变，但将以浮点形式存放，即增加小数部分（小数部分的值为 0）。如有语句“float a = 15；”则 a 的值为 15.000000。

（3）字符型数据赋给整型数据，由于字符型为 1 个字节，而整型为 4 个字节，故将字符的 ASCII 码值放到整型量的最低的 8 位中，高位的 3 个字节为 0。

（4）整型赋给字符型，只把低 8 位赋予字符变量。由于 char 只有一个字节，而 int 为 4 个字节，赋值时只把 int 型变量的值的最低 8 位赋给字符变量，高出的 24 位舍去。

2．复合的赋值运算符

复合赋值符主要有+ =、－=、*=、/=、%=。例如：

```
int a=10;
a+=5;
```

第二条相当于“a = a + 5”其他的复合赋值符也一样。要注意的是：

```
x*=y+8;
```

它等价于“x = x*(y + 8)”；而不是“x = x*y + 8；”，其他的复合赋值符也一样。

2.3.3 逗号运算符与逗号表达式

C 语言中逗号“，”也是一种运算符，称为逗号运算符。其功能是把几个表达式连接起来组成一个表达式，称为逗号表达式。它的一般形式是：

表达式 1，表达式 2，……，表达式 n

其求值过程是从左到右分别求出这 n 个表达式的值，并以表达式 n 的值作为整个逗号表达式的值。如有以下语句：

```
int a= (3*5, 6+9, 30);
```

先计算括号内逗号表达式的值，该逗号表达式的值为 30，再把 30 赋给变量 a，结果 a 的值为 30。

逗号表达式的运算优先级是最低的，赋值运算符的优先级高于它。如有以下程序代码：

我爱自由 5ifreedom.com

```
int a=5;
a=(a=3*5, a*4), a+5;
```

那么最后 a 的值是多少呢？第二条语句可以分为两部分，“a =（a = 3*5，a*4）”和“a + 5”，先算左边，再计算右边。因为逗号运算符的优先级最低，所以最外面的逗号是最后求值的。“a =（a = 3*5，a*4）”中先求括号中的部分“a = 3*5，a*4”。“a = 3*5，a*4”中又先计算“a = 3*5”，此时 a 等于 15，经过“a*4”后，a 值变为 60。再把 60 赋给 a。最后计算“a + 5”，“a + 5”的运算对 a 的值没有影响，a 只是加了 5 而没有把加后的结果赋给 a，所以对 a 没有影响。最后 a 的值是 60。

2.4 标准输入输出函数

标准输入是指从键盘获取数据输入到内存中，键盘是标准输入设备。标准输出是把内存中的数据输出到显示器进行显示，显示器也称为标准输出设备。C 语言本身并不提供输入输出语句，数据的输入输出是由库函数来完成的。要使用这些库函数，只需在程序的开头包含下列语句即可：

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

#include 是预编译指令，指示编译器包含相应的文件。stdio.h 是要包含的文件，stdio 是“standard input & output”的缩写，后缀.h 表示该文件是一个头文件。库文件和库函数将在后面详细介绍。本节主要介绍字符数据的输入输出函数 putchar()和 getchar()，格式化输入输出函数 printf()和 scanf()。

在介绍标准输入输出函数前，让我们先来了解一下函数。函数是具有一定功能的语句块。编写函数的目的是为了以后方便地调用它。函数调用的一般形式是：

函数名（参数 1，参数 2，……，参数 n）；

参数可以是 0 个，1 个或多个，参数之间以逗号分隔。如函数调用语句“printf（“a is %d”,a）”，有两个参数，一个参数是字符串，另一个参数是一个整型变量。

有些函数会有一个返回值，此时调用形式可以是：

变量＝函数名（参数 1，参数 2，……，参数 n）；

即把函数的返回值赋给变量。

计算机语言中的函数概念类似于数学上的函数，如 $y = 10x + 2$ 是一个数学函数，它的自变量是 x，函数值是 y，其功能是把自变量乘以 10，并加上 2 后，把结果赋给 y。计算机语言中的函数的参数就相当于数学上的函数的自变量，它是一个函数的输入。计算机语言中的函数的返回值相当于数学上的函数的函数值，它是一个函数的输出。计算机语言中的函数和数学上的函数一样都具有一定的功能，它们都把输入经过一定的变换后转换为输出。但计算机语言中的一些函数可以没有返回值，其输出值可以以其他方式输出，本书后面的章节会详细介绍函数的定义和使用。

2.4.1 字符输出函数 putchar

putchar 函数的作用是向标准输出设备，即显示器输出一个字符，其一般用法是：

```
putchar(c);
```

我爱自由 5ifreedom.com

c 可以是一个字符型常量、变量或一个 0～255 的整型常量或变量，还可以是转义字符，如‘\n’、‘\t’等，例 2-7 演示了 Putchar 函数的用法。

例 2-7

```
#include<stdio.h>

int main()
{
    char a , b;

    a='H';
    b='i';

    putchar(a);
    putchar(b);
    putchar('\n');
    putchar(97);
    putchar('b');
}
```

程序说明。

程序首先输出两个字符型变量 a、b，以及一个用于回车换行的转义字符‘\n’。程序在屏幕上打印“Hi”后，光标位于下一行。再输出一个字符 a，因为 a 的 ASCII 码是 97，最后输出一个字符型常量 b。

运行结果：

```
Hi
ab
```

2.4.2 字符输入函数 getchar

getchar 函数用于从标准输入设备即键盘获取一个字符，其一般形式为：

```
getchar();
```

或

```
c=getchar();
```

即把输入的字符赋给变量 *c*，变量 *c* 应该为字符型或整型，例 2-8 演示 getchar 函数的用法。

例 2-8

```
#include<stdio.h>

int main()
{
    char c;
    c=getchar();
    putchar(c);
    putchar('\n');
}
```

程序说明

程序运行时，从键盘输入任意一个字符并按<Enter>键，就会在屏幕上打印出该字符。

注意：getchar 函数只能接受 1 个字符。例 2-8 中，在屏幕上出现要求输入字符的提示符时，如果输入的字符超过 1 个，则程序只接受输入的第一个字符并把它赋给变量 *c*，其他的输入字符将被忽略。

2.4.3 格式化输出函数 printf

前面的章节中我们已经接触过格式化输出函数 printf，本节将详细介绍它的使用。putchar 函数只能输出一个字符，而 printf 可以输出任意类型的多个数据。printf 可以理解为 print function（打印函数），以方便记忆。

我爱自由 5ifreedom.com

printf 函数的一般格式为：

```
printf（控制字符串，参数 1，参数 2，……，参数 n）；
```

它的功能是，在控制字符串的控制下，将参数转换为规定的格式在标准输出设备（显示器）上进行打印显示。控制字符串中的字符包括两类：一类是普通字符，这些字符在屏幕上原样输出；另一类是格式字符，由%和格式字符组成，如%d、%f，在该位置参数以规定的格式进行输出。

printf 使用的格式字符如下。

- d：以十进制输出整型值。
- o：以八进制输出整型值。
- x：以十六进制输出整型值。
- u：以无符号形式输出整型值。
- c：输出一个字符。
- s：输出一个字符串。
- f：输出一个浮点数。
- e：以科学表示法输出浮点数。
- g：输出%f 与%e 中占用位数较短的一个。

对于长整型，除了可以使用%d 方式输出，还可以以%ld 的形式输出。现在许多计算机上长整型和整型一样，在内存都是占 4 个字节，它们所表示的数的范围是一样的，所以%d 和%ld 在这些机器上是可以通用的。但在一些较老的机器上，整型可能只占两个字节，而长整型占 4 个字节，输出长整型时须以%ld 的格式输出。

注意：单精度浮点型（float）和双精度浮点型（double）都以%f 的格式输出。

在%与格式字符之间还可以加上一些说明符以对输出格式作进一步的限定。

- -：输出时左对齐，默认是右对齐，如 printf（“%-10d”,i）。
- dd：指定输出的参数所占的最小宽度，如果数据的长度小于最小宽度则以空格来填补。例如，printf（“%5d”,i），若 i 的值的长度大于等于 5 位，则原样输出；若小于 5 位则左边以空格补齐；若是%-5d，则右边以空格补齐。
- dd.dd：用于输出浮点数时，前面的 dd 表示整个浮点数所占的宽度，后面的 dd 表示小数点后面将输出几位；输出字符串时，前面的 dd 表示整个字符串所占的宽度，后面的 dd 表示输出字符串的前 dd 个字符，例 2-9 演示了格式化输出函数 printf 的用法。

例 2-9

```
#include<stdio.h>

int main()
{
    char  c='a';
    int   i=1234;
    float f=12.123456;
    double     d=12.5;

    printf("%d %o %x\n",i,i,i);
    printf("%c %d\n",c,c)
    printf("%s\n","hello");
    printf("%f %e %g\n",f,f,f);

    printf("123456789012345678901234567890\n");
    printf("%5c%5d%10.5f%10.5f\n",c,i,f,d);
```

我爱自由 5ifreedom.com

```
    printf("1234567890123456789012345678 90\n");
    printf("%-5c%-5d%-10.5f%-10.5f\n",c,i,f,d);
}
```

程序输出：

```
1234 2322 4d2
a 97
hello
12.123456 1.212346e+01 12.1235
1234567890123456789012345678 90
a 1234   12.12346  12.500000
1234567890123456789012345678 90
a    1234 12.12346    12.50000
```

程序说明。

第 1 行以十进制、八进制和十六进制 3 种方式打印出整型变量 i 的值。$(2322)_8 = 2 \times 8^3 + 3 \times 8^2 + 2 \times 8^1 + 2 \times 8^0 = 1234$，$(4d2)_{16} = 4 \times 16^2 + d \times 16^1 + 2 \times 16^0 = 1234$。

第 2 行以两种方式输出字符型变量 a。

第 3 行输出一个字符串。

第 4 行以 3 种方式输出浮点型变量 f。%f，即以一般浮点形式输出；以%e 方式输出时，小数部分最多只能输出 6 位，最后一位 6 是四舍五入得到的；以%g 方式输出时，小数部分只能保留 4 位。同样地，最后一位 5 也是四舍五入得到的。

第 5 行和第 7 行输出 30 个字符，主要是为了方便地观察在带说明符（如 dd.dd）时，数据的输出结果。

第 6 行以 dd.dd 的格式输出各种数据类型。如字符型变量 c 其值‘a’只占一位，在以%5c 方式输出时，左边 4 位补以空格。

第 8 行以-dd.dd 的格式输出各种数据类型。如整型变量 I，其值 1234 占四位，在以%-5d 格式输出时，先输出 1234，第 5 位以空格补充。

2.4.4 格式化输入函数 scanf

格式化输入函数其一般格式为：

```
scanf（控制字符串，参数 1，参数 2，……，参数 n）；
```

它接受来自键盘的输入，自动把输入的数据转换为规定的格式并存储到由参数指定的变量中。参数必须是以变量地址的形式给出。变量的地址其实就是存储变量的内存单元的编号，第 1 章的 1.5 节已经作过介绍。运算符&用于取得变量的地址，如有一个整型变量 a，则它的地址为&a。

与 printf 一样，scanf 可以使用的格式字符如下所示。

- d：期待输入一个十进制整数。
- o：期待输入一个八进制整数。
- x：期待输入一个十六进制整数。
- u：期待输入一个无符号整数。
- c：期待输入一个字符。
- s：期待输入一个字符串。
- f：期待输入一个浮点数。
- e：期待输入一个以科学表示法的表示的浮点数。

与 printf 类似，可以以%ld 的方式输入一个长整型，例 2-10 演示了格式化输入函数 scanf 的用法。

我爱自由 5ifreedom.com

例 2-10

```
#include<stdio.h>
int main()
{
    int  i;
    char  c;
    float  f;

    scanf("%d%c%f",&i,&c,&f);
    printf("%d %c %f",i,c,f);
    scanf(("%d,%c,%f",&i,&c,&f);
    printf("%d %c %f",i,c,f);
}
```

程序运行：

```
123 a 12.5
123 a 12.5
456,b,21.5
456 b 21.5
```

程序说明。

（1）程序运行，等待用户输入，这里输入 123、a 和 12.5，之间以空格间隔，回车后程序在屏幕上打印出 3 个变量的值。系统怎样知道 123 是赋给变量 i 而字符 a 是赋给变量 c 的呢？当遇到空格键、制表键<Tab>或<Enter>键时，系统认为是一个变量输入的结束。注意，在输出时，printf 函数中%d、%c、%f 之间有空格，空格作为普通字符在屏幕上会原样输出，所以有了第 2 行的打印结果。

（2）在第 2 次要求输入时，我们已经在程序里规定了用逗号分割，此时必须以第 3 行这样的方式输入，输入数据以逗号隔开。第 4 行打印出各个变量的值。

（3）那么如果有错误的输入，系统会有什么反映呢？在设计程序时，必须考虑到用户可能不会每次都按照软件的要求正确地输入数据，所以在设计软件时必须考虑对错误输入的处理，否则一旦用户有错误的输入，程序就崩溃了，这样的软件显然是缺乏竞争力的。如果我们在程序第一次要求输入时，输入 a123a12.5，然后回车，输入的数据之间没有空格，也没有按<Tab>或<Enter>键，会有什么结果呢？结果是系统输出一些奇怪的数。

2.5 Vi 编辑器的使用

Vi（即 visual interface 的简称）许多年来一直是 Linux 上主要的文本编辑软件。它可以进行文本输入、删除、查找、替换、块操作等诸多文本操作。Vi 有许多命令，初学者可能会觉得它比较烦琐，但熟练之后，就会发现 Vi 是一个简单易用并且具备强大功能的源程序编辑器。

2.5.1 Vi 的工作模式

Vi 有 3 种工作模式：命令模式（Command Mode）、插入模式（Insert Mode）和末行模式（Last Line Mode），如图 2-4 所示。

1. 命令模式

在 Shell 中启动 Vi 时，最初就是进入命令模式。在该模式下可以输入各种 Vi 命令，可以进行光标的移动，字符、字、行的删除，复制，粘贴等操作。此时，从键盘上输入的任何字符都作为

我爱自由 5ifreedom.com

命令来解释。在其他两种模式下，按<Esc>键，就可以转换到命令模式。

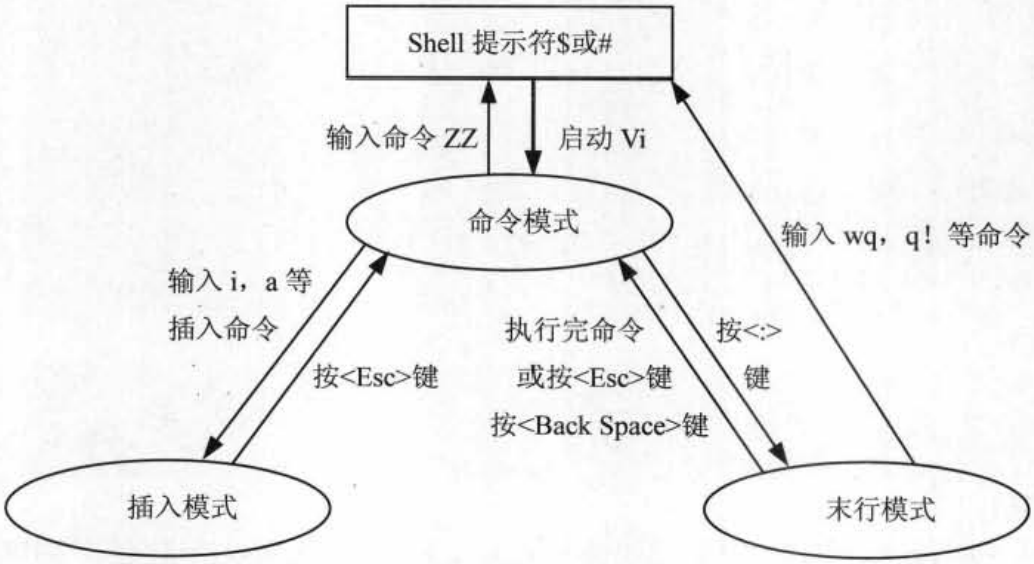


图 2-4　Vi 编辑器的 3 种工作模式

注意：在此模式下输入的任何字符屏幕都不会显示出来。

2．插入模式

插入模式主要用于输入文本。在该模式下，用户输入的任何字符都作为文件的内容保存起来，并会显示在屏幕上。在命令模式下输入 i、a 等命令就可以进入插入模式。要返回到命令模式，只需按<Esc>键即可。

3．末行模式

在命令模式下，按<：>键就进入了末行模式。此时 Vi 在窗口的最后一行显示一个“：”，并等待用户输入命令。在末行模式下，可以进行诸如保存文件、退出、查找字符串、文本替换、显示行号等操作。一条命令执行完毕，就会返回到命令模式。

提示：当处于末行模式，并已经输入了一条命令的一部分而不想继续时，按几次<Back Space>键删除已输入的命令或直接按<Esc>键都可以进入命令模式。

2.5.2　启动 Vi

输入以下命令都可以启动 Vi。

- Vi：不指定文件名，在保存文件时需要指定文件名。
- Vi　文件名：该文件既可以是已存在的也可以是新建的。
- vi　+n 文件名：进入 Vi，光标停在第 n 行开始处。
- Vi　+　文件名：进入 Vi，光标停在文件最后一行开始处。
- Vi　+/字符串 文件名：进入 Vi，光标停在第一个字符串处。

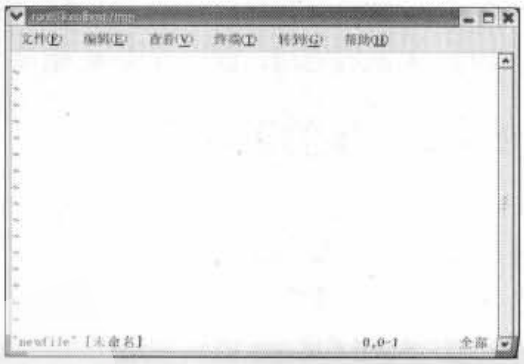


图 2-5　Vi 编辑器窗口

如图 2-5 所示为输入“Vi newfile”命令时 Vi 的窗口，“～”表示该行是新的没有被编辑过的行。

我爱自由 5ifreedom.com

2.5.3 保存文件和退出 Vi

在命令模式下按两次<Z>键，将保存文件并退出 Vi。或在末行模式下输入如下命令。

- w：保存当前正在编辑的文件，但不退出 Vi，w 是 write 的首字母。
- w 文件名：将当前文件的内容保存由“文件名”指定的新文件中，若该文件已存在则产生错误，该命令也不会退出 Vi。
- w！文件名：将当前文件的内容保存由“文件名”指定的新文件中，若该文件已存在则覆盖原文件，该命令也不会退出 Vi。
- q：不进行文件保存直接退出 Vi，若文件有改动过而没有保存将产生错误，q 是 quit 的首字母。
- q!：强行退出 Vi，若文件内容有改动则恢复到文件的原始内容。
- wq：保存并退出 Vi，这是最常用的退出 Vi 的方式。

提示：在末行模式下，输入如下命令。

```
: set number 或
: set nu
```

可以给每一行添加行号，这在调试程序时会很有用。

注意：行号并不是文件内容的一部分。

2.5.4 光标的移动

在 Vi 的插入模式下，一般使用键盘上的 4 个方向键来移动光标。而在命令行模式下则有很多移动光标的方法，常用的移动方法如下所示。

- ↑：移动到上一行，所在的列不变。
- ↓：移动到上一行，所在的列不变。
- ←：左移一个字符，所在的行不变。
- →：右移一个字符，所在的行不变。
- 0：移动到当前行的行首。
- $：移动到当前行的行尾。
- nw：右移 n 个字，n 为数字，光标处于第 n 个字的字首。
  w 是 forword（向前）中的字母 w。
- w：右移 1 个字，光标处于下一个字的字首。
- nb：左移 n 个字，n 为数字，光标处于第 n 个字的字首。
  b 是 back（向后）的首字母。
- b：左移 1 个字，光标处于下一个字的字首。
- (：移到本句的句首，如果已经处于本句的句首，则移动到前一句的句首。
-)：移动下一句的句首。
- {：移到本段的段首，如果已经处于本段的段首，则移动到前一段的段首。
- }：移动下一段的段首。
- 1G：移动到文件首行的行首。
- G：移动到文件末行的行首。
- nG：移动到文件第 n 行的行首。

我爱自由 5ifreedom.com

- <ctrl>+g：报告光标所处的位置，位置信息显示在 Vi 的最后一行。

提示：遇到“.”、“?”或“!”，Vi 认为是一句的结束。Vi 以空白行来作为段的开始或结束。

2.5.5 文本的删除

在插入模式下，用<Delete>键可以删除光标所在位置的一个字符，用<Back Space>键删除光标所在位置的前一个字符。在命令模式下，有各种各样的删除文本的方法，常用的删除方法如下所示。

- x：删除光标所在位置的一个字符。
- nx：删除从光标开始的 n 个字符。
- dw：删除光标所在位置的一个字，d 是 delete 的首字母。
- ndw：删除从光标开始的 n 个字。
- db：删除光标前的一个字。
- ndb：删除从光标开始的前 n 个字。
- d0：删除从光标前一个字符到行首的所有字符。
- d$：删除光标所在字符到行尾的所有字符。
- dd：删除光标所在的行即当前行。
- ndd：删除从当前行开始的 n 行。
- d(：删除从当前字符开始到句首的所有字符。
- d)：删除从当前字符开始到句尾的所有字符。
- d{：删除从当前字符开始到段首的所有字符。
- d}：删除从当前字符开始到段尾的所有字符。

提示：如果要取消前一次操作，在命令模式下输入字符 u 即可。u 是 undo 的首字母。

2.5.6 文本查找和替换

在命令模式下，查找文本的方法如下。

- ?string<Enter>：在命令模式下输入？和要查找的字符串如“string”并回车即可。
- n：向文件头方向重复前一个查找命令。
- N：向文件尾方向重复前一个查找命令。

在命令行模式下，替换文本的方法如下。

- :s/oldstr/newstr：在当前行用 newstr 字符串替换 oldstr 字符串，只替换一次 s 是 substitue 的首字母。
- :s/oldstr/newstr/g：在当前行用 newstr 字符串替换所有的字符串 oldstr。
- :1,10s/oldstr/newstr/g：在第 1～10 行用字符串 newstr 来替换所有的字符串 oldstr。
- :1,$s/oldstr/newstr/g：在整个文件中用字符串 newstr 来替换所有的字符串 oldstr。

2.5.7 文本的复制与粘贴

复制和粘贴是文本编辑中的常用操作，Vi 也提供了这种功能。复制是把指定内容复制到内存的一块缓冲区中，而粘贴是把缓冲区中的内容粘贴到光标所在位置。

复制和粘贴的方法如下。

我爱自由 5ifreedom.com

- yw：将光标所在位置到字尾的字符复制到缓冲区中，y 是 yank 的首字母。
- nyw：将光标所在位置开始的 n 个字复制到缓冲区中，n 为数字。
- yb：从光标开始向左复制一个字。
- nyb：从光标开始向左复制 n 个字，n 为数字。
- y0：复制从光标前一个字符到行首的所有字符。
- y$：复制从光标开始到行末的所有字符。
- yy：复制当前行，即光标所在的行。
- nyy：复制从当前行开始的 n 行，n 为数字。
- p：在光标所在位置的后面插入复制的文本，p 是 paste 的首字母。
- P：在光标所在位置的前面插入复制的文本。
- np：在光标所在位置的后面插入复制的文本，共复制 n 次。
- nP：在光标所在位置的前面插入复制的文本，共复制 n 次。

2.6 Emacs 编辑器的使用

Emacs 是 Linux 下一个功能强大的图形化文本编辑软件，可以用来编写 C 源程序。与 Vi 相比，它的一个显著特点是可以使用鼠标进行大部分的操作，对于习惯使用 Windows 系统的用户来说，Emacs 是一个不错的选择。单击 Linux 桌面左下角的“红帽”图标，再选择“编程”→“Emacs”选项即可打开 Emacs，如图 2-6 所示。第 1 章已经简要地介绍过 Emacs 的使用，本节将在此基础上进一步介绍 Emacs 编辑器。

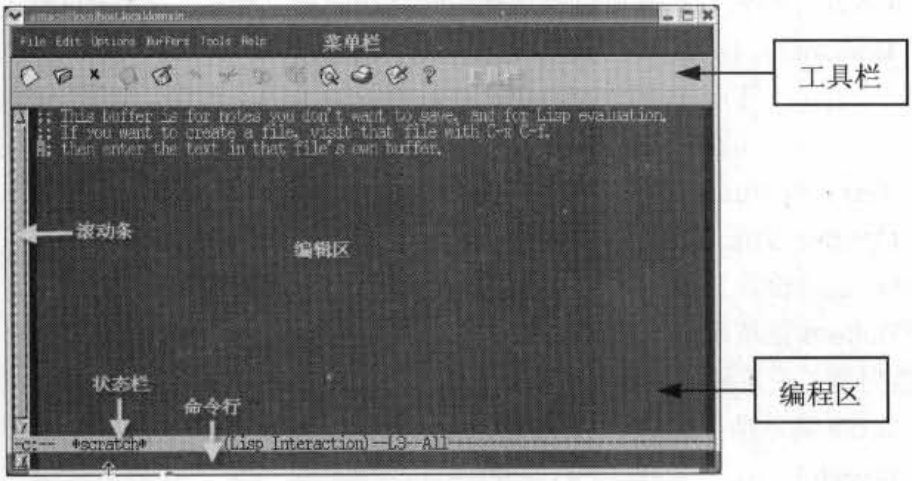


图 2-6　Emacs 编辑器初始界面

图 2-6 中工具栏中放置的是一些文本编辑中常用的操作图标，如图 2-7 所示。

图 2-7　文本编辑操作图标

从左往右所代表的操作分别是：打开文件，打开目录，关闭文件，保存，另存为，撤销操作，

我爱自由 5ifreedom.com

剪切，复制，粘贴，查找，打印，定制外观，帮助。

编辑区用于文本的编辑。通过上下拉动滚动条可以方便地浏览文件内容。状态栏显示文本编辑时的状态信息，如编辑的文件名，当前光标所在位置等。打开文件时，可以在命令行输入要操作的文件的路径，以便找到要操作的文件。

菜单栏中的条目显示了 Emacs 提供的所有操作。下面列举了在编写程序时常用的一些操作。

1. File 菜单项

- Open File：打开文件。
- Close (current buffer)：关闭当前操作的文件。
- Save (current buffer)：保存当前操作的文件。
- Save Buffer As：把当前文件内容另存为其他文件。
- Split Window：拆分窗口，在 Emacs 中可以在不同的窗口同时操作几个不同的文件。
- Unsplit Window：取消拆分窗口。
- Exit Emacs：退出 Emacs。

2. Edit 菜单项

- Undo：撤消上一次操作。
- Cut：剪切。
- Copy：复制。
- Paste：粘贴。
- Clear：删除。
- Select All：全部选择。
- Search：查找。
- Go To：跳转到，这里可以选择跳转相应的行、标签、文件的开头、文件的结尾。
- Bookmak：标签，这里可以进行设置标签、重命名标签、删除标签等操作，标签是 Emacs 的一个特色，在编辑比较大的文件时，设置一些标签可以方便文件的操作。
- Text Properties：设置文本的显示方式，如颜色、字体等。

3. Options 菜单项

可以在这里设置 Emacs 的一些选项，定制 Emacs 的外观。

4. Buffers 菜单项

这里列举了最近操作过的文件。

5. Tools 菜单项

- Search Files：文件查找。
- Compile：编译程序。
- Shell Command：执行 Shell 命令。
- Dubugger：调试程序。
- Spell Checking：拼写检查。
- Version Control：版本控制。
- Read Mail：阅读邮件。
- Send Mail：发送邮件。

我爱自由 5ifreedom.com

6. Help 菜单项

提供一些关于使用 Emacs 的帮助信息。

2.7 命名规范

2.7.1 标识符

在程序设计中，变量名、函数名、数组名等统称为标识符。简单地说，标识符就是一个名字。除库函数的函数名由系统定义外，其余都由用户自定义。C 语言规定，标识符只能由字母（a～z，A～Z）、数字（0～9）、下划线（_）组成，并且标识符的第一个字符必须是字母或下划线，不能以数字开头。C 语言不限制标识符的长度，但它受各种 C 语言编译器的限制，同时也受到具体机器的限制。例如，在某编译器中规定标识符前 8 位有效，当两个标识符前 8 位相同时，则被认为是同一个标识符。建议变量名的长度最好不要超过 8 个字符。

以下标识符是合法的：

i、a、count、number_of_book、BOOK_NUMBER、sum100、_total。

以下标识符是非法的。

3com：以数字开头。

char：char 是 C 语言的一个数据类型，是保留字，不能作为标识符，其他的如 int、float 等类似。

a*b：*不能作为标识符的字符。

number of book：标识符中不能有空格。

注意：C 语言是区分大小写的，如 Count 与 count 被认为是两个不同的标识符。

2.7.2 关键字

关键字是由 C 语言规定的具有特定意义的字符串，通常也称为保留字。用户定义的标识符不应与关键字相同。C 语言的关键字分为以下几类。

（1）类型说明符

用于定义、说明变量、函数或其他数据结构的类型，如 int，double 等。

（2）语句定义符

用于表示一个语句的功能。例如，后面章节将会介绍的“if else”就是条件语句的语句定义符。

（3）预处理命令字

用于表示一个预处理命令，如前面用到的 include。

2.7.3 命名规范

如果只是编写一些小程序，程序只有几十或几百行，编程风格可能并不重要。然而，如果是和许多人一起进行一定规模的项目开发，或者希望过一段时间之后，还能够快速而准确地理解自己的程序，就必须养成良好的编程习惯。良好的编程风格可以增加程序代码的可读性。编程风格最能体现一个程序员的综合素质。变量的命名规范是编程风格的一项重要内容。

变量的命名规范有很多种。在 Windows 下开发软件，许多人采用匈牙利命名法，而在 Linux

我爱自由 5ifreedom.com

环境下，并不建议使用这种方法。

Linux 下建议的命名规则如下。

（1）变量名必须有一定的意义，并且意义准确。例如有一个变量用于保存图书的数目，可以命名为 number_of_book 或者 num_of_book。不建议使用 i，因为它没有意义。也不建议使用 number 或 book，因为意义不准确。

（2）不建议大小写混用。如定义一个计数变量，int nCount；这在 Windows 中是一个很好的变量名，其中 nCount 的首字母 n 用来说明这个变量的类型是 int。但在 Linux 下不建议大小写混合使用，一般标识符只由小写字母，数字和下划线构成。

（3）在失去意义的情况下，尽量使用较短的变量名。例如有一个变量，用于暂时存储一个计数值，把变量命名为 tmp_count 显然要比 this_is_a_temperary_counter 好。

（4）不采用匈牙利命名法表示变量的类型。如 int nCount；n 用于说明变量的类型，在 Linux 中不建议这样命名变量。

（5）函数名应该以动词开头，因为函数是一组具有特定功能的语句块。比如一个函数，它用于取得外部输入的数值，则可以命名为 get_input_number。

（6）尽量避免使用全局变量。全局变量在后面的章节中会介绍。

2.8 面试题选与实例精讲

2.8.1 面试题选

本小节的例子均为知名公司在招聘时使用过的试题。本章内容是 C 语言的基础部分，单独命题较少，命题时会增加综合题，以测试程序员对基本知识的理解程序，如例 2-11、例 2-12 所示。

例 2-11

```
#include<stdio.h>

int main()
{
    int a,b,c,d;

    a=10;
    b=a++;
    c=++a;
    d=10*a++;

    printf("b, c, d: %d, %d, %d", b, c, d);
}
```

程序说明。

本程序首先定义了 4 个整型变量，然后把 10 赋给变量 a。

b = a ++相当于 b = a；a = a + 1，因此，该条语句结束后，b 的值为 10，a 的值为 11。而“c = ++ a”相当于“a = a + 1”；“c = a”，该条语句运行结束后 a 的值变为 12，c 的值也为 12。

对于“d = 10*a ++”，这条语句中有 3 个运算符：=、*和++，乘号运算符*的优先级最高，所以*运算先进行。++运算符只针对变量，因此这条语句相当于“d = 10*(a++);”而不是“d=(10*a)++”。

我爱自由 5ifreedom.com

“d=10*(a++)”又相当于“d = 10*a；a = a+1”，所以变量 d 的值为 120，a 的值为 13。

例 2-12 用两种方法实现交换两个变量的值。

方法一：使用一个中间变量。

```
int a=1, b=2, temp;
temp=a;
a=b;
b=temp;
```

程序说明

首先把变量 a 的值 1 暂存到变量 temp 中，temp 变量的值为 1。

然后把变量 b 的值 2 赋给变量 a，此时变量 a 和 b 的值都为 2。

最后把变量 temp 的值 1 赋给变量 b，此时变量 b 的值为 1，从而实现了两个变量的值交换。

方法二：不借助于任何中间变量。

```
int a=1, b=2;
a=a+b;
b=a-b;
a=a-b;
```

程序首先把 a + b 的值赋给 a，此时 a 的值变为 a + b 即 3，b 的值不变，仍为 2。

语句 b = a − b 运行之后，b 的值为 3 − 2 即 1，a 的值不变仍为 3。

语句 a = a − b 运行之后，a 的值为 3 − 1 即 2，b 的值不变仍为 1。此时变量 a 和 b 的值已经实现了交换。

2.8.2 实例精讲

本小节精选了一些有助于进一步理解和掌握 C 语言的实例。

例 2-13 下列变量名中哪些是合法的？

```
3a, π, c#, temp1, float, _var。
```

分析。3a 以数字开头，不合法。对于标识符（变量是标识符的一种），只能由 a～z、A～Z、数字以及下划线“_”组成，π 和#都是非法字符，因此 π 和 c#都不是合法的变量名。

float 是 C 语言的关键字（或称为保留字）不能作为标识符，也就不能作为变量名，不合法。

只有 temp1 和_var 是合法的变量名。

例 2-14 下列哪个是合法字符常量？

```
"a", '\n', 'China', a。
```

分析。“a”是字符串常量而不是字符常量。C 语言规定，以“”括起来的字符是字符串常量，字符串常量中的每一个字符都用一个字节来存储，并且系统自动在字符串的结尾添加一个特殊的字符’\0’以表示字符串的结束。而对于字符常量或变量，系统只用一个字节来存储。

’\n’是一个合法的字符常量，它是一个转义字符，用于回车换行。’\n’在系统并不是用一个字符来存储’\’，再用一个字节来存储’n’，而是把’\n’作为一个整体，存储它的 ASCII 码。

’China’既不是字符常量也不是字符串常量，“China”才是一个字符串常量。

a 也不是字符常量，’a’才是字符常量。a 可以是一个合法的变量名，它可以作为一个整型变量，字符变量或其他变量。

例 2-15 若有以下定义。

```
char a='a'; int b=10; float c=0.5; double d=1.25;
```

则表达式 a*b + c − d 的值是什么类型？

分析。在计算表达式 a*b + c − d 的值时，首先计算 a*b，char 型变量的值默认要转换为 int 型

我爱自由 5ifreedom.com

进行计算，而变量 b 是 int 型，所以 a*b 的值为 int 型。再计算 a*b 与 c 的和，由于 float 型的值默认要转换为 double 型进行计算，相应地 a*b 的值也要转换为 double 型后才参与运算，由于 c 是 float 型，所以 a*b+c 的结果还要转换回 float 型。最后计算 a*b+c 与 d 的差，a*b+c 的值先转换为 double 型，再计算结果。由于 d 是 double 型，结果不会转换回 float 型，最后的结果为 double 型。这些转换都是由系统自动完成的，程序员并不需要参与。

例 2-16 变量 a 的初始值为 3，下面的语句执行之后，变量 a 的值是多少？

```
a+=a-=a*a;
```

分析。赋值运算符"="是右结合性的，也就是从右往左进行计算。在"a+=a-=a*a"中，"a-=a*a"相当于"a=a-(a*a)"，而"a+=a-=a*a"相当于"a=a+(a-=a*a)"，所以整个语句相当于"a=a+(a=a-(a*a))"。在计算"a=a+(a=a-(a*a))"时，首先计算"a-(a*a)"，值为-6，此时该语句为"a=a+(a=-6)"，即把-6 赋予变量 a 后，计算"a=a+a"，所以最后 a 的值为-12。熟练之后，不用进行如此变换，可立即得出结果-12，但在初学时，最好能一步一步分析清楚。

例 2-17 下列赋值语句哪个是正确的？

```
A  x=3,y=5
B  a=b=c
C  i--;
D  y=int(x);
```

分析。A 错误，C 语言中，除了宏定义等少数语句，其他语句都以"；"结束，A 没有以"；"结束。

B 的错误与 A 一样。

C 正确，该语句将 i 减 1。

D 错误，在进行强制类型转换时，应该是"y=(int)x"。"y=(int)x+y"、"y=(int)(x+y)"也都是正确的，前者对变量 x 进行类型转换，后者是对 x 与 y 的和进行类型转换。

例 2-18 分析下面的程序段，指出错误。

```
(1) int a,b;
    scanf("%d %d",a,b);
(2) float f=2.5;
    printf("%d",f);
(3) main()
    {
        x=1;
        y=2;
        printf("x+y  is   %d\n",x+y);
    }
(4) main()
        {
        int  x, y;
        printf("x+y  is   %d\n",x+y);
    }
(5) main()
    {
        char c=300;
        printf("c  is   %d\n",c);
    }
(6) int a;
    scanf("%d\n",&a);
```

分析。

（1）在 scanf 函数中，参数应该是变量 a、b 的地址，正确的应该是：

```
scanf("%d %d",&a,&b);
```

我爱自由 5ifreedom.com

（2）数据输出的格式与数据类型不匹配，正确的应该是：

```
printf("%f",f);
```

（3）C 语言中，变量应该遵守“先定义，后使用”的规则。变量 x、y 在没有定义的情况下直接使用是错误的，正确的应该是在引用 x、y 变量值的语句前加上定义语句：

```
int a, b;
```

（4）该程序在编译时，编译器虽然不会报告错误，但会发出警告，提示变量 x 和 y 没有赋值就直接使用了。如果执行程序，程序将输出一个混乱的值。正确的应该是在打印语句前加上类似于下面的语句：

```
x=10,y=20;
```

（5）char 型值的范围是−128～127，即使是 unsiged char，它的范围也仅仅是 0～255，300 显然已经超出了变量所能存储的值的范围。正确的应该是赋给变量 c 一个合理的值。

（6）严格地说，这两条语句并没有错误。在使用 scanf 函数时，建议不要包含‘\n’。如果一定要这么使用，则在输入变量 a 的值时，应该按两次<Enter>键，因为字符‘\n’的作用相当于<Enter>键。

2.9 习题

1．下面的程序会输出什么结果？

```
main()
{
    char  a, b;
    a=97;
    b=98;
    printf("%c%c\n",a,b);
    printf("%d%d\n",a,b);
}
```

2．写出下面程序的运行结果。

```
main()
{
    int  i, j,a,b;
    i=8;
    j=12;
    a=++i;
    b=j++;
    printf("%d,%d,%d,%d\n",i,j,a,b);
}
```

3．写出下面赋值表达式运行之后 a 的值，设 a 的初始值为 12，n 的值为 5。

（1）a＋＝a；　　（2）a%＝7；

（3）a*＝2＋5；　　（4）a/＝a＋a；

（5）a%＝(n%＝2)；　　（6）a－＝a＋＝a*＝a。

4．在 Linux 环境下，使用 Vi 编辑器编写测试程序，对于 1～3 题，验证你的答案。

5．在 Linux 环境下，使用 Emacs 编辑器编写一个程序，要求输入两个整数，输出这两个整数的积、余数和平均数。

我爱自由 5ifreedom.com

第 3 章　C 程序控制结构和 gcc 编译器

本章将介绍 3 种运算符及其表达式：关系运算符、逻辑运算符和条件运算符。并在此基础上介绍 C 程序的 3 种基本控制结构：顺序结构、选择结构和循环结构。同时还将结合 Linux 环境，介绍编译器 gcc 的使用，以及作为一个优秀的软件开发人员必须熟悉的程序编码风格。

本章重点：

- 关系运算符、逻辑运算符和条件运算符。
- 3 种基本控制结构。
- 编译器 gcc。
- 编码风格。

本章难点：

- 使用 3 种基本控制结构来解决实际问题。
- 程序的调试方法。

3.1 C 程序的控制结构

3.1.1　C 程序语句概述

程序由语句构成，语句类似于自然语言中的句子，用来向计算机发出指令。C 语言既有完成单一任务的简单语句，也有由多条语句构成完成某一功能的复合语句。C 语言以分号作为一个语句的结束。总体而言，C 程序的语句必为以下 5 种语句之一。

1．空语句

下面是一个空语句：

```
;
```

空语句只有一个分号，它什么也不做。在程序的某些位置，语法上需要一个语句，但逻辑上并不需要，此时可以使用空语句。空语句的一种常见用法是在循环条件判断部分就能完成所有工作的情况下，使用一个空语句来代表循环体。具体实例见本章的 3.3 节。

注意：在实际开发中，空语句应该加上注释，以便让阅读代码者知道该语句是有意使用的，而不是疏忽或是错误。

2．表达式语句

一个表达式加上一个分号就构成了一个语句。例如：

```
a = a + 3;
```

“a＝a＋3”是一个赋值表达式，如果没有分号，编译器不能识别，加上分号后，编译器就认为这是一个语句，会去分析这个语句，并执行该语句。

```
i + j;
```

我爱自由 5ifreedom.com

这也是一个符合语法的语句，只是这个语句没有任何意义。它把 i 和 j 的值相加，而没有把获得的结果赋给其他变量，所以是没有实际意义的。

3．复合语句

复合语句又被称为块（block），是用一对大括号括起来的语句集合（大括号中也可以没有任何语句，只不过这是没有实际使用意义的）。块同时也标识了一个作用域，在一个块中定义的变量只能在该块的内部使用。例如：

```
int i = 10;
int j = 3;
{
    int result = i % j;
    printf("%d",result);
}
```

大括号和它内部的两条语句就构成了一个块或称为复合语句，注意在块前面定义的变量可以在块的内部使用，而块内定义的变量不能在块的外部使用。

4．函数调用语句

由一个函数调用和一个分号就构成了一个函数调用语句。例如：

```
printf("%d",result);
```

现在我们只要会使用函数即可，函数的定义将在后面章节中介绍。函数也和变量一样，要先定义后使用，一个没有定义的函数，编译器是不能识别也不知道它完成了什么样的功能。我们在程序中可以直接使用 printf、scanf 函数，因为它们是系统预先定义好了的函数。

5．控制语句

控制语句用于控制程序的执行流程。

C 程序中有 9 种控制语句，如下所示。

（1）if ...else...：条件语句。

（2）switch：多分支选择语句。

（3）for()...：循环语句。

（4）while()...：循环语句。

（5）do...while()：循环语句。

（6）continue：结束本次循环语句。

（7）break：结束整个循环语句或结束 switch 语句。

（8）return：函数调用返回语句。

（9）goto：转向语句（一般不使用，只在特殊情况下使用）。

本章将详细介绍这些控制语句的使用。

3.1.2 C 程序的 3 种基本控制结构

在程序设计语言的发展过程中，Bohra 和 Jacopini 已经证明使用顺序、选择、循环 3 种基本控制结构可以表达任何计算机能够解决的问题的处理流程。

1．顺序结构

如图 3-1 所示，这是顺序结构。A、B 分别代表一个语句块，这里的语句块是由一条或多条语句构成。其执行流程是先执行 A 语句块，然后执行 B 语句块。

2．选择结构

图 3-2、图 3-3 所示是两种选择结构。选择结构有时也称为分支结构。图 3-2 中，程序执行到

我爱自由 5ifreedom.com

该部分时，首先判断 P 条件是否成立，若条件成立则执行 A 语句块，若条件不成立则执行 B 语句块。也即 A 和 B 只能执行其中的一个，执行哪一个由条件 P 来决定。图 3-3 中，条件 P 成立则执行 A 语句块，若不成立则不执行任何语句，跳到 A 语句块后面执行。C 语言中用于描述选择结构的控制语句为 if ...else ...和 switch 语句。

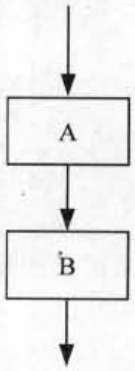


图 3-1　顺序结构

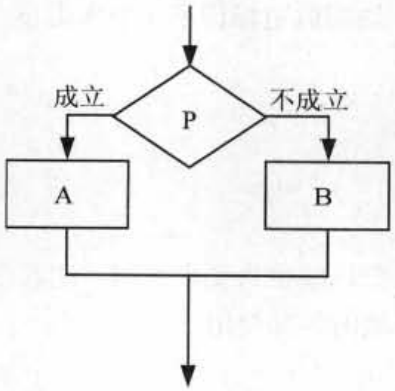


图 3-2　选择结构（1）

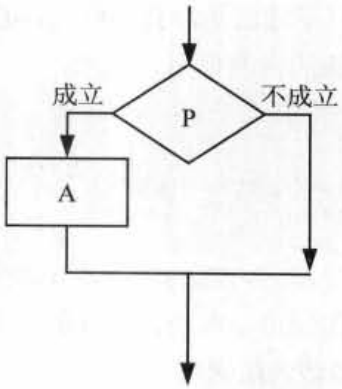


图 3-3　选择结构（2）

3. 循环结构

图 3-4、图 3-5 所示是两种循环结构。图 3-4 所示的循环结构的执行流程是先判断条件 P 是否成立，若成立则执行 A 语句块，执行完毕后再判断条件 P 是否成立，若仍成立则继续执行 A，直到 P 条件不成立，才继续往下执行。图 3-5 所示的则是先执行一次 A，然后判断条件 P 是否成立，若不成立则再次执行 A，直到条件 P 成立为止。C 语言中用于描述选择结构的控制语句为 whlie、do...while、for 语句。

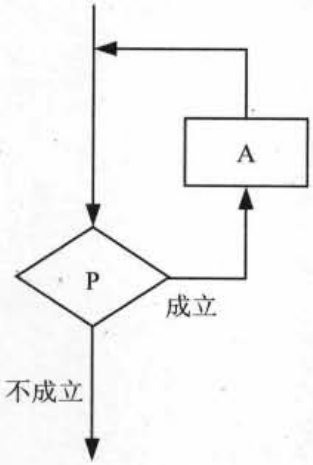


图 3-4　循环结构（1）

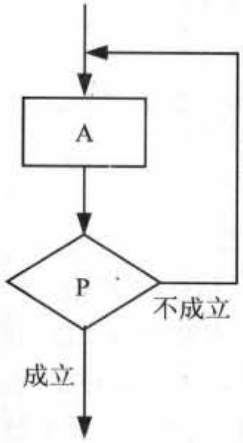


图 3-5　循环结构（2）

若是初次接触程序的控制结构可能觉得比较抽象，下面的章节中会多次以实例的方式进一步解释这些控制结构及其使用方法。

3.2 分支控制结构

3.2.1 关系运算符与关系表达式

关系运算符实际上就是进行比较运算，将两个值进行比较，确定比较的结果为真还是为假。

我爱自由 5ifreedom.com

比如，“i < 0”是一个关系表达式，小于号“<”是一个关系运算符。若 i 的值为 1，则这个表达式为假，表达式的值为 0；若 i 的值为-1，则这个表达式为真，值为 1。

C 语言提供了 6 种关系运算符。

（1）<：小于

（2）<=：小于或等于

（3）>：大于

（4）>=：大于或等于

（5）==：等于

（6）!=：不等于

前 4 种关系运算符的优先级相同，后两种相同，但前 4 种的优先级高于后两种。关系运算符的优先级低于算术运算符，但高于赋值运算符。

例如：

```
a == b < c:   等价于 a == (b < c);
a = b <= c:   等价于 a = (b <= c)将变量 b 与 c 比较的结果赋给变量 a;
a >= b + c:   等价于 a >= (b + c)将变量 b 与 c 的和与 a 进行比较。
```

将两个表达式用一个关系运算符连接起来就构成了一个关系表达式，用来构成关系表达式的两个表达式可以是单独的常量、变量，也可以是算术表达式、赋值表达式、关系表达式或下一节介绍的逻辑表达式。关系表达式的值是一个逻辑值，要么为“真”要么为“假”。比如 5 < 3 这个关系表达式的值为假，结果在数值上为 0。表达式“'a'==97”值为真，结果在数值上为 1。

注意：区别赋值运算符=与判断两个量是否相等的关系运算符==。

3.2.2 逻辑运算符与逻辑表达式

用逻辑运算符将关系表达式或逻辑表达式连接起来就构成了逻辑表达式。C 语言中提供了 3 种逻辑运算符：

（1）&&：逻辑与，即并且。

（2）||：逻辑或，即或者。

（3）!：逻辑非，即取反。

逻辑与&&和逻辑或||是双目运算符，即需要两个运算量参与运算，如（a> b）&&（a>c），该表达式语意为 a 大于 b 并且 a 大于 c。由于逻辑与和逻辑非运算符的优先级都低于关系运算符，因此这个表达式也可以写作 a > b && a > c。逻辑非!为一目运算符，只要求一个运算量，例如，!(i<j)，若 i 的值小于 j，则这个表达式为假，反之则为真。因为逻辑非!把 i<j 运算的结果取反了。

对于逻辑与，要求两个运算量都为真，结果才为真，参与运算的两个量是并且的关系。而逻辑或，只要求两个运算量中的任何一个为真，结果就为真，参与运算的两个量是或者的关系。

逻辑表达式的值与关系运算符一样，其结果都是逻辑值，要么为真要么为假。在 C 语言中，通常以数字 1 代表真，以数字 0 代表假。但事实上，只要是非 0 的数，都认为是真。例如：

若变量 i 的值为 3，则!i 的值为 0。因为 i 的值为非 0，被认为是真，对真进行逻辑非运算，结果为假。

对于逻辑表达式：

```
i=12 && j=1
```

其结果为真，对于 i=12 这个赋值表达式，其含义为将 12 赋给变量 i，然后引用变量 i 参与逻辑运算，该逻辑表达式等价于 12&&1。而对于表达式：

```
i=12 && j=0
```

我爱自由 5ifreedom.com

它等价于 12&&0，非 0 数 12 被当作真，0 被当作假，其结果为假，数值上为 0。

逻辑与&&和逻辑或||运算符，其求值存在一个“懒惰求值”规则。

对于逻辑与运算，如果第一个运算量的值为假，则这个逻辑表达式的值必为假，而不论第二个运算量的值是什么。因此没有必要去求第二个运算量的值，编译器也不会求第二个运算量的值。

对于逻辑或运算，如果第一个运算量的值为真，则这个逻辑表达式的值必为真，而不论第二个运算量的值是什么。因此没有必要去求第二个运算量的值，编译器也不会求第二个运算量的值。例如：

```
int  i = 10, j = 0;
int  k = i || (j = 1);
```

执行后，i，j，k 的值分别为 10，0，1。在求解表达式 i || (j = 1)时，因为 i 的值为 10，因此这个表达式的值必为真，因此不会执行 j = 1，所以最后 j 的值仍为 0。k 的值为 1，为什么不是 10 呢？因为赋值运算符的优先级仅高于逗号运算符，因此这个表达式等价于 k =(i || (j = 1))。

3.2.3 if 语句

if 语句根据一个表达式是否为真来有条件选择语句块执行。

1．if 语句有 3 种使用模式。

（1）if（表达式）语句

例如：

```
if ( i > 0 )  printf(%d,i);
```

这条语句首先判断变量 i 的值是否大于 0，若是则打印出 i 的值；若 i 的值小于或等于 0 则不打印。它的执行流程见图 3-3。

这里的语句既可以是一条语句也可以是由一对大括号括起来的语句块。还可以是一条空语句：

```
if ( i <= 0 )  ;
```

注意：当多个语句必须作为单个语句块来执行时，比较常见的错误是漏掉大括号。

例如：

```
if ( i != j )
        printf(%d,i);
        printf(%d,j);
```

程序员的本意是当 i 与 j 的值不相等时打印出 i 和 j 的值。缺少了大括号后，不论 i 与 j 的值是否相等，j 的值都会被打印。因为第二条 printf 语句是否执行不受 if 条件的控制。

（2）if（表达式）语句 1 else 语句 2

例如：

```
if ( i > j )
        printf(%d,i);
else
        printf(%d,j);
```

这条语句首先判断 i 的值是否大于 j，若是则打印出 i 的值；若不是则打印出 j 的值。它的执行流程见图 3-2。同样的，语句 1 和语句 2 可以是单个语句，语句块或空语句。一般在 if 语句中使用空语句是没有意义的。

注意：else 总是和处在它前面的与它最接近的 if 配对。

对于如下语句：

```
if ( i < j )
    if ( i < k )
        printf ("%d",i);
else
    printf ("%d",k);
```

else 语句与第二个 if 条件配对，而不是与 if(i < j)配对。可以通过以下的书写方式使程序更为直观：

我爱自由 5ifreedom.com

```
if ( i < j )
    if ( i < k )
        printf ("%d",i);
    else
        printf ( "%d",k);
```

程序代码中应该使用缩进以使程序变得更为清晰。使用代码缩进是一种良好的编程习惯。

（3）if（表达式）

```
语句 1
else if (表达式)
语句 2
else if (表达式)
语句 3
. . .
else
语句 n
```

例如：

```
if (number > 10000)
     discount = 0.3;
else if (number >5000 )
     discount = 0.2;
else if (number >1000)
     discount = 0.1;
else
     discount = 0;
```

这条语句根据购买的数量来设定折扣，若购买量大于 10000，折扣为 3 折；大于 5000 而小于 10000，为 2 折；大于 1000 小于 5000，为 1 折；1000 以下不打折。

3 种 if 语句后面都有表达式，这里表达式一般为关系表达式或逻辑表达式，即可以判断真假的表达式。

```
if ( i<j && i<k )
    printf ("%d",i );
```

这条语句在 i 的值既小于 j 又小于 k 的情况下打印出 i 的值，使用了一个逻辑表达式，逻辑表达式的两个运算量都是关系表达式。

在执行 if 语句时，先对表达式进行求解，若表达式的值为 0，按“假”来处理，若表达式的值非 0，就按照“真”来处理。对于以下语句：

```
if ( 2 )
    printf ("OK");
```

上述语句是合法的。执行结果是 OK。表达式的值为 2，按“真”来处理。表达式的类型不限于逻辑表达式或关系表达式，也可以是任意数值类型，如整型、实型、字符型和后面将介绍的指针类型。例如，下面的语句也是合法的：

```
if ('a' )
    printf ("OK");
```

注意：一般 if 语句的表达式应该为关系表达式或逻辑表达式。只有在特殊情况下使用其他表达式。

例 3-1 输入 3 个数，要求从小到大顺序输出。

首先写一个伪代码：

```
if a > b 将 a 和 b 的值互换          //此时 a 中存放的是 a、b 这两个数中较小的一个
if a > c 将 a 和 c 的值互换          //此时 a 中存放的是 a、c 这两个数中较小的一个
                                     //因此 a 是 3 个数中最小的一个
if b > c 将 b 和 c 的值互换          //此时 b 中存放的是 b、c 这两个数中较小的一个
                                     //因此 b 是 3 个数中第二小的
```

上述逻辑执行的代码如下。

```
#include<stdio.h>

main()
{
    float a,b,c,temp;
```

我爱自由 5ifreedom.com

```
    scanf("%f,%f,%f",&a,&b,&c);

    if ( a > b )
    {
        temp=a;a=b;b=temp;
    }
    if ( a > c )
    {
        temp=a;a=c;c=temp;
    }
    if ( b > c )
    {
        temp=b;b=c;c=temp;
    }

    printf ("%f,%f,%f",a,b,c);
}
```

运行情况如下：

```
输入：1，10，5
输出：1，5，10
```

2．条件运算符

条件运算符要求有 3 个操作对象，也称为三目运算符，它是 C 语言中唯一的一个三目运算符。条件表达式的一般形式为：

```
表达式 1? 表达式 2：表达式 3
```

它的执行流程是，如果表达式 1 为真，则表达式 2 的值是整个表达式的值，否则表达式 3 的值是整个表达式的值。例如：

```
max = ( a > b ) ? a : b;
```

条件运算符的优先级仅高于赋值运算符和逗号运算符，逗号运算符是所有运算符中优先级最低的，赋值运算符（包括复合赋值运算符，如+=、-=、*=、/=等）次之。所以这个表达式首先计算 a > b，再计算条件表达式“？：”的值，最后把条件表达式的值赋给变量 max。这个表达式把 a、b 中较大的一个变量的值赋给 max。它等价于：

```
if ( a > b )
    max = a;
else
    max = b;
```

注意：条件表达式同样遵守“懒惰求值”规则。即当表达式 1 为真时，计算表达式 2 的值并把该值作为整个表达式的值，而不会去求解表达式 3 的值。当表达式 1 为假时也不会去求表达式 2 的值。

例如，有以下代码段：

```
int i = 1,j = 2, k = 3,x;
x = ( i > 0 )? (j = j*2): (k = k*2);
printf("%d,%d,%d,%d",i,j,k,x);
```

运行后输出 1，4，3，4。可以看到，k 的值没有变化，也就是条件表达式中的表达式 3 没有被计算。x 的值为 j 的值，为 4。

3.2.4 switch 语句

假设在计算学生分数等级时，90 分以上为 A，80～89 为 B，70～79 为 C，60～69 为 D，60 以下为 E。给定一个学生的成绩等级时，如何确定成绩实际所处的分数段呢？这里就要使用多分支选择。switch 语句是多分支选择语句。if 语句最多只有两个分支，它只能从这两个分支中选择一个来执行。当然也可以通过 if 语句嵌套的方式实现多分支选择，但这往往不直观，程序的可读性不高。

switch 语句的一般形式为：

```
    switch（表达式）
    {
```

我爱自由 5ifreedom.com

```
        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
    }
```

例如，要求按照学生成绩的等级打印出该成绩所处的分数段，用 switch 语句实现如下：

```
char grade;
scanf("%c",&grade);
switch(grade)
{
    case 'A': printf("90～100\n");
    case 'B': printf("80～89\n");
    case 'C': printf("70～79\n");
    case 'D': printf("60～69\n");
    case 'E': printf(" < 60\n");
    default: printf("You have inputed wrong score.\n");
}
```

当表达式的值与某一个 case 后面的常量表达式的值相等时，就执行该 case 后面的语句，若所有的 case 中的常量表达式的值都不与表达式的值相等，就执行 default 后面的语句。

每个 case 的常量表达式的值必须互不相同，否则编译器就不知道该执行哪条 case 后面的语句。

各个 case 和 default 的出现次序不影响执行结果。但为了程序的可读性，应该尽量使各个 case 出现的次序有序。

执行完一个 case 后面的语句后，程序的执行流程将转移到下一个 case 继续执行，而不再判断这条 case 后面的常量表达式的值与表达式是否相等。例如上面这个程序段中，若输入 C，则输出的结果为：

```
70～79
60～69
< 60
You have inputed wrong score.
```

为了在执行完一个 case 后面的语句后就跳出 switch 结构，即终止 switch 语句的执行，应该使用 break 语句。将上面的 switch 语句改成：

```
switch(grade)
{
    case 'A':printf("90～100\n");
          break;
    case 'B':printf("80～89\n");
          break;
    case 'C':printf("70～79\n");
          break;
    case 'D':printf("60～69\n");
          break;
    case 'E':printf(" < 60\n");
          break;
    default: printf("You have inputed wrong score.\n");
}
```

此时，如果再输入 C，则只输出 70～79。

case 后面有多条语句时，不用加大括号以构成一个语句块，系统自动把它们当作一个语句块来执行。这是一种非常特殊的情况。当然加上大括号也可以。

不是每个 case 语句后都必须有 break 语句。例如，有一个统计一篇英语文章中出现的元音字母数的程序段：

```
int letterCount = 0;
. . .
switch(ch)
{
    case 'a':
    case 'e':
```

我爱自由 5ifreedom.com

```
    case 'i':
    case 'o':
    case 'u':++letterCount;
}
```

此时 ch 若为一个元音字母，如‘e’，它会执行 case‘e’后面的语句，由于为空，不执行任何语句，又因为没有遇到 break 语句，继续往下执行，直到执行++letterCount 这一条语句结束整个 switch 语句为止。

注意：switch 表达式所计算的结果必须为整型，常量表达式也必须是整型数值，且不能为变量。

以下 3 种 switch 语句都是错误的。

（1）float grade = 2.5；

```
switch(grade)
{. . .
```

grade 为浮点型，不是整型值，所以这是错误用法。

（2）int count = 2；

```
switch(count)
{
   case 1: . . .
   case 2.5: . . .
   . . .
2.5 不是整型常量，所以这也是错误用法。
```

（3）int count = 2；

```
int num = 2;
switch(count)
{
   case 1: . . .
   case num: . . .
   . . .
   case num 中的 num 是变量不是常量，所以这也是错误用法。
```

3.3 循环控制结构

许多问题都需要使用循环控制结构来解决。几乎所有的商业开发程序都包含循环语句。循环结构是结构化程序设计的基本结构之一，它和顺序结构，分支选择结构共同作为程序设计的基本构造单元。因此，理解和熟练使用循环结构是程序设计的基本要求。所有高级程序设计语言都提供了循环控制结构，只是在语法上有些细微差别。

3.3.1 while 语句

while 语句的一般形式为：

```
while（表达式）
   语句 1
```

当表达式为真时，执行语句 1，执行完语句 1 后再去判断表达式是否为真，如果还是为真继续执行语句 1，直到表达式为假时才停止执行语句 1。如果在第一次判断表达式真假时，表达式的值就为假，则不执行语句 1 继续往下执行。语句 1 又称为循环体。图 3-6 是它执行时的流程图，下面用例 3-2 实现 while 语句。

例 3-2 求 1～100 的和。

解决这个问题的流程图如图 3-6 所示。

根据流程图写出程序代码如下：

我爱自由 5ifreedom.com

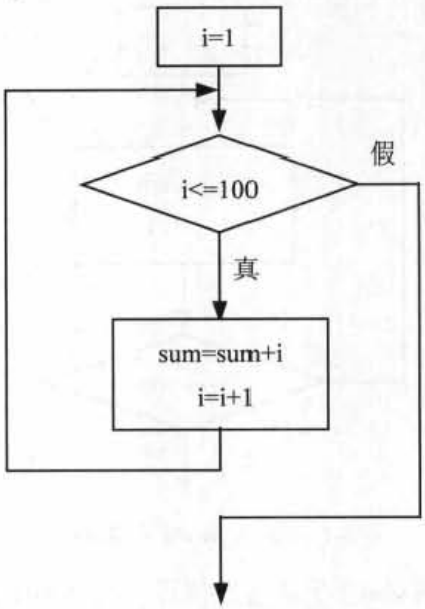


图 3-6 while 语句循环结构

```
#include<stdio.h>

main()
{
    int sum = 0, i = 1;

    while(i <= 100)
    {
        sum = sum+i;
        i = i + 1;
    }

    printf("%d\n",sum);
}
```

程序说明。

（1） 如果循环体中包含一个以上的语句，应该用大括号括起来以构成一个复合语句。否则循环体只是 while 后面的第一个语句。如在本例中，如果没有大括号，循环体只是 sum=sum+i。

（2）循环体中应该有使循环能够结束的语句。例如，在本例中，i 的初始值为 1，每执行一次循环体 i 的值就加 1。从 i=1 开始第 1 次执行循环体到 i=100 时第 100 次执行循环体，每次在循环体中都把 i 的值加到总和 sum 上，循环执行了 100 次后，再次循环时，判断表达式为假，循环结束。

（3） 循环体允许为空语句。在某些循环中，表达式部分就已经做完了所有工作，此时写一个空语句作为循环语句的循环体。注意，此时的空语句是必需的，否则编译器 gcc 会报错。

注意：在循环体中定义的变量在每次循环里都要经历重新定义和销毁变量的过程。

3.3.2 do...while 语句

do...while 的一般形式为：

```
do
    循环体语句
while(表达式);
```

它的执行流程是：先执行一次循环体语句，然后计算表达式，如果表达式的值为真（即表达式的值非 0）重复执行循环体语句，然后再判断表达式的值，如此反复，直到表达式的值为假（即表达式的值为 0）才结束整个循环语句。图 3-7 是 do...while 语句的执行流程图。

我爱自由 5ifreedom.com

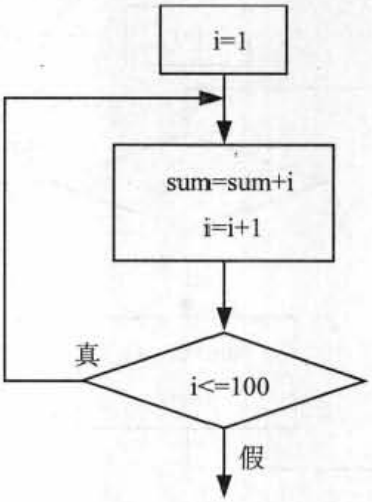


图 3-7　do... while 循环结构

do...while 语句与 while 语句的区别在于前者先执行一次循环体然后判断表达式的值，而后者先判断表达式的真假，如果表达式的值一开始就为假，则循环体一次也不执行。

例 3-3　使用 do... while 语句求 1～100 的和。

首先规划这个问题的解题步骤，可以先画出流程图，如图 3-7 所示。

根据流程图写出程序代码如下：

```
#include<stdio.h>

main()
{
    int sum = 0, i = 1;

    do
    {
        sum = sum + i;
        i = i + 1;
    }while(i <= 100);

    printf("%d\n",sum);
}
```

程序说明

对于一个问题如果可以用 while 语句来处理，那么也可以用 do...while 语句来解决。反之，可以用 do...while 语句来解决的也可以用 while 语句来解决。它们两者是等价的。事实上，它们和后面的 for 语句都是等价的，我们可以根据问题的特点和自己的习惯选择一种循环结构来解决遇到的问题。

注意：do...while 语句最后有一个分号，而 while 语句没有。

3.3.3　for 语句

for 语句是 C 语言中最灵活、功能最强也是使用得最多的循环语句。它不仅可以用于循环次数确定的情况，而且也可以用于循环次数不确定的情况，它完全可以取代 while 语句。

for 语句的一般形式为：

```
for（表达式 1；表达式 2；表达式 3）
    循环体语句
```

表达式 1 通常用来给循环变量赋初值，一般是赋值表达式。也允许在 for 语句外给循环变量赋初值，此时可以省略该表达式。循环变量用于控制循环执行的次数，例如，上两例中的变量 i。

表达式 2 通常是一个条件表达式或逻辑表达式，用于设置循环的结束条件。

我爱自由 5ifreedom.com

表达式 3 一般用来修改循环变量的值，一般是赋值语句。

这 3 个表达式都可以是逗号表达式，即每个表达式都可由多个表达式组成。3 个表达式都是任选项，都可以省略。

它的执行流程是。

（1）求解表达式 1。

（2）求解表达式 2，如果其值为真（值非 0）则执行循环体语句，然后转入第（3）步；如果为假（值为 0）则结束循环，转入第（5）步。

（3）求解表达式 3。

（4）转入第（2）步执行。

（5）循环结束，执行 for 语句后的下一条语句。

例如，使用 for 语句求 1～100 的和。

```
int sum=0,i;
for(i=1; i<=100; i++)
    sum=sum+i;
```

它的执行过程是：首先求解表达式 1，也就是将 1 赋给变量 i。然后求解表达式 2，此时 i 的值为 1，小于等于 100 也就是表达式 2 为真，执行循环体语句即将 1 加到总和 sum 上。转入第（3）步，求解表达式 3，将变量 i 的值加 1。然后再去求解表达式 2，变量 i 的值 2 还是小于或等于 100，重复执行循环体。每执行一次循环体，就将一个数加到总和上，并使变量 i 的值增 1。比如第一次执行循环体，将 1 加到总和 *sum* 上，并使变量的值变为 2。如此反复，当执行了 100 次循环体后，就将 1～100 都加到了总和 *sum* 上，i 的值变为 101，表达式 2 的条件为假，循环结束。

for 语句中表达式 1 可以省略，可以在 for 语句前给相应的变量赋初值，例如：

```
int sum=0,i=1;
for(; i<=100; i++)
    sum=sum+i;
```

此时表达式 1 就是仅有一个分号构成的空语句。但这个空语句是必需的，不可省略。

也可以省略表达式 3，把表达式 3 要完成的任务放在循环体中，例如：

```
int sum=0,i=1;
for(; i<=100; )
{
    sum=sum+i;
    i=i+1;
}
```

表达式 2 也可以省略，例如：

```
for(i=1;  ; i++)
```

此时在计算 for 循环的表达式 2 时永远认为表达式 2 为真，除非在循环体中有可以使循环结束的语句，不然 for 循环会不断执行下去，这样的循环一般被称为死循环。一般在程序设计中应该避免出现死循环。但是死循环不一定是有害或无用的，有些特殊场合就要使用死循环。

表达式 1 和表达式 3 还可以使用逗号表达式。在逗号表达式内按从左到右的顺序求解，整个逗号表达式的值是最右边的表达式的值。例如：

```
int sum,i;
for(sum=0, i=1; i<=100; i++)
    sum=sum+i;
```

在这个 for 语句中表达式 1 使用了逗号表达式，在这个逗号表达式中对变量 *sum* 和 *i* 赋了初值。以下语句，完成了对 1～100 之间所有奇数的求和。

```
int sum,i;
for(sum=0, i=1; i<=100; i++, i++)
    sum=sum+i;
```

我爱自由 5ifreedom.com

我们看到，此时 for 语句在表达式 3 中对变量 i 执行两次加 1 操作，也就是每次循环结束，i 的值都增长 2，这样就是 1、3、5、7... 99 的和。

当然也可以简单地写为：

```
int sum,i;
for(sum=0, i=1; i<=100; i=i+2)
    sum=sum+i;
```

注意：for 语句中表达式 1 只执行 1 次。表达式 2、表达式 3 和循环体执行 0 次、1 次或多次。

3.3.4 break 语句和 continue 语句

在介绍 switch 分支选择语句时已经使用过了 break 语句，它用于跳出 switch 语句，继续执行 switch 语句下面的一条语句。实际上，break 语句也可以用来从循环体内跳出，接着执行循环语句下面的一条语句。break 语句的一般形式为：

```
break;
```

它只用在 switch 语句和 3 种循环语句中。

例如：

```
#include<stdio.h>
main()
{
    int square,i;
    for(i = 1; i <= 100; i++)
    {
        square = i * i;
        if(square > 100)
            break;
        printf("%d\n",square);
    }
}
```

这个程序用于计算并打印出 1～100 之间所有数的平方，但在平方值大于 100 时就提前结束循环。这个程序的运行结果是只在屏幕上打印出 10 行，即 1～10 的平方值。

与 break 语句结束整个循环语句不同的是，continue 语句只结束本次循环。即跳过循环体内尚未被执行的语句，接着进行下一次是否执行循环的判定。

例 3-4 在屏幕上输出 1～100 之间不能被 13 整除的数。

```
#include<stdio.h>

main()
{
    int number;

    for(number=1; number<=100; number++)
    {
        if(number%13 == 0)
            continue;
        printf("%d ",number);
    }

    printf("\n");
}
```

程序说明

当 number 能被 13 整除时，执行 continue 语句，结束本次循环，即跳过了 printf 语句，只有当 number 不能被 13 整除时才执行 printf 语句，打印出 number 的值。

对于较为复杂的问题往往需要使用循环的嵌套。一个循环的循环体内又包含了另一个循环结构就称为循环的嵌套。内嵌的循环还可以再嵌套循环结构，这就构成了多层循环。一般 3 层以上的循环嵌套较少使用。

我爱自由 5ifreedom.com

3 种循环（while 循环、do... while 循环和 for 循环）可以相互嵌套。

例 3-5 求 1～10 的阶乘和 1！+2！+3！+... +10！。

使用两层循环来求解这个和。里面一层循环用于求 1!、2!、3!... 10!，外面的循环对这些值求和。源代码如下：

```
#include<stdio.h>

int main()
{
    int  i, j;
    long sum = 0;

    for(i=1; i<=10; i++)
    {
        int total = 1;
        for(j=1; j<=i; j++)
            total=total*j;
        sum=sum+total;
    }

    printf("1!+2!+3!+...+10!=%d\n",sum);
    return 0;
}
```

程序说明。

（1）变量 i 和 j 是循环计数变量，用于控制循环的次数。*sum* 是待求的总和，total 用于求各个阶乘。

（2）对于内层循环，其循环的次数是 i。如当 i 为 5 时，则 total=1*2*3*4*5。

（3）total 是在外层循环的循环体内定义的，每执行一次外层循环，total 被重新定义并被赋初值 1。

3.4 gcc 编译器

在 Linux 下开发应用程序时，大多数情况下使用的都是 C 语言，因此几乎每一位 Linux 程序员都必须掌握和灵活使用一种 C 编译器。编译器把人易于理解和使用的高级语言编写的源代码构建成计算机能够直接执行的二进制代码。CPU 并不能理解诸如 main、int、double 这些 C 语言的关键词，它只能识别和执行只由 0、1 两个数码构成的二进制代码。Linux 下最常用的 C 语言编译器是 gcc（GNU Compiler Collection），它是 GNU 项目中符合 ANSI C 标准的编译器，能够编译用 C、C++语言编写的程序。gcc 不仅功能非常强大，结构也非常灵活。此外，它还可以编译 Java、Fortran、Pascal 和 Ada 等语言编写的程序。

gcc 是 Linux 平台编译器的事实标准。同时，在 Linux 平台下的嵌入式开发领域，gcc 也是用得最多的一种编译器。gcc 之所以被广泛采用，除了功能强大、简单灵活的特点之外，还因为它能支持各种不同的硬件平台。目前，gcc 支持的体系结构有几十种之多，常见的有 Intel x86 系列、Arm、PowerPC 等。同时，gcc 还能运行在不同的操作系统上，如 Linux、Solaris、Windows 等。它既支持基于宿主的开发（也就是要为某平台编写程序，就在该平台上编译），也支持交叉编译（即在 A 平台上编译的程序是供平台 B 使用的）。

3.4.1 程序的编译过程

在使用 gcc 编译程序时，编译过程可以分为 4 个阶段。

（1）预处理：（Pre-Processing）。

我爱自由 5ifreedom.com

（2）编译：（Compiling）。

（3）汇编：（Assembling）。

（4）链接：（Linking）。

Linux 程序员可以根据自己的需要让 gcc 在编译的任何阶段结束，以便检查或使用编译器在该阶段的输出信息，或者对最后生成的二进制文件进行控制，以便通过加入不同数量和种类的调试代码来为今后的调试做好准备。和其他常用的编译器一样，gcc 也提供了灵活而强大的代码优化功能，利用它可以生成执行效率较高的代码。

在功能上，预处理、编译、汇编是 3 个不同的阶段，但 gcc 在实际操作时可以把这 3 个步骤合并为一个步骤来执行。下面以几个实例介绍如何生成各个阶段的代码。

在预处理阶段，输入的是 C 语言的源文件，通常为*.c 或*.C，它们一般带有*.h 之类的头文件。这个阶段主要处理源文件中的#ifdef、 #include 和#define 预处理命令。该阶段会生成一个中间文件*.i，但实际工作中一般不用专门生成这种文件，若必须要生成这种文件，可以使用下面的命令：

```
gcc -E  test.c -o test.i
```

它通过对源文件 test.c 使用 E 选项来生成中间文件 test.i。

在编译阶段，输入的是中间文件*.i，编译后生成汇编语言文件*s。这个阶段对应的 gcc 命令如下所示：

```
gcc -S test.i -o test.s
```

在汇编阶段，将输入的汇编文件*.s 转换成二进制机器代码*.o。这个阶段对应的 gcc 命令如下所示：

```
gcc -c test.s -o test.o
```

最后，在链接阶段将输入的二进制机器代码文件*.s（与其他的机器代码文件和库文件）汇集成一个可执行的二进制代码文件。这一步骤，可以使用下面的示例命令完成：

```
gcc test.o -o test
```

最终生成了可执行文件 test。

对于上述过程可以简化为：

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

或者直接使用一条命令：

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

3.4.2 gcc 的常用选项

在实际开发中，使用 gcc 编译源程序时，源文件通常不止一个，这时就需要使用 gcc 编译多个源文件。这也非常简单，使用下面的命令即可：

```
gcc -o test testmain.c other1.c other2.c
```

该命令将同时编译 3 个源文件 testmain.c、other1.c、other2.c，最后生成一个可执行程序 test。

需要注意的是，在生成可执行程序时，一个程序无论是只有一个源文件还是多个源文件，在所有被编译和连接的源文件中必须有且只有一个 main 函数，因为 main 函数是每一个程序的入口点（当系统调用执行该程序时，首先将控制权授予程序的 main 函数）。但如果仅仅是把源文件编译成目标文件，因为不会进行最后一步的链接，所以这时 main 函数不是必需的。

在使用 gcc 编译器的时候，我们必须给出一系列必要的选项和文件名。gcc 编译器的选项有 100 多个，其中很多参数一般是用不到的，这里只介绍其中最基本、最常用的参数。可以通过使用以下命令来详细了解所有选项：

```
man gcc
info gcc
```

我爱自由 5ifreedom.com

gcc 最基本的用法是：

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

其中 options 就是编译器所需要的选项，filenames 给出相关的文件名。

- -c：只编译，不链接成可执行文件，编译器只是由输入的.c 等为后缀的源代码文件生成.o 为后缀的目标文件，通常用于编译不包含主程序的子程序文件。
- -o output_filename：确定输出文件的名称为 output_filename，同时这个名称不能和源文件同名。如果不给出这个选项，gcc 就默认将输出的可执行文件命名为 a.out。
- -g：产生调试器 gdb 所必需的符号信息，要对源代码进行调试，就必须在编译程序时加入这个选项。
- -O：对程序进行优化编译、链接，采用这个选项，整个源代码会在编译、链接过程中进行优化处理，这样产生的可执行文件的执行效率较高，但是，编译、链接的速度就相应地要慢一些。
- -O2：比-O 更好的优化编译、链接，当然整个编译、链接过程会更慢。
- -Wall：输出所有警告信息，在编译的过程中如果 gcc 遇到一些认为可能会发生错误的地方就会提出一些相应的警告和提示信息。提示注意这个地方是不是有什么错误。
- -w：关闭所有警告，建议不要使用此选项。
- -Idirname：将名为 dirname 的目录加入到程序头文件目录列表中，它是在预处理阶段使用的选项。I 意指 Include。

在 C 语言程序中，头文件被大量使用。一般而言，C 程序通常由头文件（header files）和定义文件（definition files）组成。头文件是一种包含函数原型声明、常量定义的文件，用于保存程序的声明（declaration），而定义文件用于保存程序的实现（implementation）。C 程序的头文件以“.h”为后缀。

C 程序中包含头文件有两种方法：

① #include <myinc.h>

② #include "myinc.h"

第一种使用尖括号（<>），第二种使用双引号（“ ”）。对于第一种，编译器 gcc 在系统预设包含文件目录（如/usr/include）中查找相应的头文件 myinc.h。而对于第二种，编译器 gcc 首先在当前目录中查找头文件，如果当前目录中没有找到需要的头文件，就到指定的 dirname 目录中去寻找。

对于系统提供的头文件，通常以第一种方式包含到源程序中。

在实际开发中，对于自己编写的头文件，通常放在与源程序相同的目录中，并以第二种方法包含该头文件。但在编写大型程序时，往往把头文件单独放在一个目录中，此时编译程序时需要用该选项告诉编译器到名为 dirname 的目录中去查找头文件。否则，编译器会因为找不到相应的头文件而使编译失败。

- -Ldirname：将名为 dirname 的目录加入到程序的库文件搜索目录列表中，它是在链接过程中使用的参数。L 意指 Link。

库是事先已经编写好的代码，经过编译后可以直接调用的文件。也可以编写属于自己的库文件，以方便自己使用或提供给他人使用。库体现了软件工程中的软件重用的思想。对于一些常用的功能，可以编写成一系列的函数，并把这些函数按功能放在库文件中。系统默认提供了许多库，比如数学函数库，它提供了一些求绝对值、开方、求三角函数值等功能函数。库分为两种：静态库和动态库。库的定义和使用将在后面的章节中介绍。

在默认情况下，编译器 gcc 在系统默认的路径中（如/usr/lib）寻找所需要的库文件，这个选项告诉编译器，首先到-L 指定的目录中去寻找，然后到系统默认的路径中寻找，如果函数库存放

我爱自由 5ifreedom.com

在多个目录下，就需要依次使用这个选项，给出相应的存放目录。

- -lname：指示编译器，在链接时，装载名为 libname.a 的函数库，该函数库位于系统预定义的目录或者由-L 选项指定的目录下。例如，-lm 表示链接名为 libm.a 的数学函数库。

3.4.3 gcc 的报错类型及对策

gcc 编译器如果发现源程序中有错误，就无法继续进行编译，也无法生成最终的可执行文件。为了便于修改，gcc 给出错误的相关信息，必须对这些错误信息逐个进行分析、处理，并修改相应的源代码，这样才能保证源代码被正确编译。gcc 给出的错误信息一般可以分为以下几类。

1. C 语法错误

错误信息：源文件的第 n 行有语法错误（syntex errror）。例如："test.c：5：error：syntex errror before } token"，这条错误提示信息指出在源程序 test.c 的第 5 行的"}"前发现了语法错误。这种类型的错误，一般都是 C 语言的语法错误，应该仔细检查源代码文件中第 n 行及该行之前的程序。有些情况下，一个很简单的语法错误，gcc 会给出一大堆错误信息，此时可以回过头来阅读书本，深入地理解相关内容并仔细分析源程序。

2. 头文件或库文件错误

错误信息：找不到头文件 head.h。例如："test.c：1：21：stdioaaa.h：没有那个文件或目录"，这条错误提示信息指出了一个头文件错误。这类错误是源代码文件中包含的头文件有问题，可能的原因有头文件名错误、指定的头文件所在目录名错误等，也可能是错误地使用了双引号和尖括号。

3. 未定义的符号

错误信息：有未定义的符号（undefined symbol）。这类错误是在链接过程中出现的，可能有以下几种原因：一是程序员在没有定义相应的函数或者变量的情况下就调用了该函数或使用了该变量，这需要程序员根据实际情况修改源程序，给出变量或者函数的定义；二是未定义的符号是一个标准的库函数，在源程序中使用了该库函数，而链接过程中还没有给定相应的函数库的名称，或者是该库文件的目录名有问题，这时需要检查所使用的库函数到底位于哪一个函数库中，确定之后，修改 gcc 链接选项中的-l 和-L 项。

以下是一个包含上述 3 种错误的源程序及编译后 gcc 输出的错误提示。

```
#incldue<stdioaaa.h>
main()
{
    printf("%d",i);
    printf("Hello , world! ")
}
```

对上面程序进行编译。

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

编译后的错误提示如下。

```
test.c:1:21: stdioaaa.h: 没有那个文件或目录
test.c: In function 'main':
test.c:4: error: 'i' undeclared (first use in this function)
test.c:4: error: (Each undeclared identifier is reported only once
test.c:4: error: for each function it appears in.)
test.c:6: error: syntax error before '}' token
```

提示信息的 3～5 行指出 i 未声明，在这个函数中是首次使用。每一个未声明的标识符在一个函数种只会报告一次。

编译、链接过程中出现的错误，往往是因为语法错误或包含头文件、库文件时发生了错误。这些错误在编译器 gcc 的提示下一般是比较容易排除的。一个程序通过编译后并不表示它可以正

我爱自由 5ifreedom.com

确无误地完成预想的功能。我们还需要输入一些数据对它进行测试，若在测试中程序输出的结果并不是我们所预想的，这就表明程序存在逻辑错误，也就是我们对问题的处理存在着错误。此时需要使用调试器，如 gdb，对程序进行调试。一个复杂的程序，往往要经过多次的编译、测试和修改。第 4 章我们将介绍调试器 gdb 的基本使用。

3.5 面试题选与实例精讲

3.5.1 面试题选

本部分所选例题均为知名软件公司在招聘时使用过的试题。

例 3-6 写出 float 类型的变量 i 与零值比较的语句。

由于 float 的精度问题，不可写为：

```
if (x == 0.0)
if (x != 0.0)。
```

而应该写为：

```
if ((x >= -0.00001) && (x <= 0.00001))
```

不可将浮点变量用“==”或“！=”与数字比较，应该设法转化成“>=”或“<=”之类的形式。

例 3-7 写出下面两个循环的优缺点。

① for (i=0; i<N; i++)

```
{
    if (condition)
        DoSomething;
    else
        DoOtherthing;
}
```

② if (condition)

```
{
    for (i=0; i<N; i++)
        DoSomething;
}
else
{
    for (i=0; i<N; i++)
        DoOtherthing;
}
```

第一个循环的优点是程序简洁，缺点是多执行了 N-1 判断并且不利于编译器对代码进行优化，降低了程序执行效率。

第二个循环的优点是避免了第一个循环的缺点，执行效率较高，缺点是程序不够简洁。

例 3-8 switch（表达式）语句中不能作为表达式值的类型是什么？

表达式的值类型必须是整型（char 字符型在内存中以 ASCII 码来存储，因此也可以），其他类型都不行。

例 3-9 for（；1；）有什么问题？它是什么意思？

它是一个死循环，除非 for 循环体中有可以使循环结束的语句，否则它将一直运行下去。类似的还有 while（1）。

例 3-10 do…while 和 while…有什么区别？

前一个是循环一遍再判断，后一个是判断以后决定是否循环。

我爱自由 5ifreedom.com

3.5.2 实例精讲

例 3-11 日本某地发生了一件谋杀案，警察通过排查确定杀人凶手必为 4 个嫌疑犯中的一个。以下为 4 个嫌疑犯的供词。

A 说：不是我。

B 说：是 C。

C 说：是 D。

D 说：C 在胡说。

已知 3 个人说了真话，1 个人说的是假话。现在请根据这些信息，写一个程序来确定到底谁是凶手。

分析：这个问题有很多解法，现在我们使用本章所介绍的关系表达式、if 语句、for 语句来解决这个问题。

对于一个实际问题，首先要对这个问题进行分析并把它转化为计算机可以理解的形式。我们把 4 个人所说的 4 句话写成表达式，为此先定义一个字符型变量 killer 来表示凶手：

将供词转化为关系表达式，如表 3-1 所示。

表 3-1 将供词转化为关系表达式

说 话 人	所 说 的 话	关系表达式
A	不是我	killer != 'A'
B	是 C	killer == 'C'
C	是 D	killer == 'D'
D	C 在胡说	killer != 'D'

然后，我们要考虑使用什么算法来描述解决问题的思路和程序的执行流程，对于这个问题，答案只有 4 种可能。因此，我们可以使用穷举算法。穷举算法逐一尝试每一个可能的解，直到找到问题的解或尝试完所有可能的解为止。当然有的问题的候选解可能非常多，对此我们可以对穷举法作些改进，比如不用去尝试那些明显不可能的候选解或者只尝试最有可能的那部分解。很多密码破解软件就是使用穷举法来破解密码的。

对于这个问题，我们首先假定凶手为 A，即假设 killer=' A'，然后去验证 4 个人说的话，如果 4 个表达式恰好有 3 个为真，那么就可以确定 A 是凶手。如果 A 不是凶手，就设 killer=' B'，再去验证 4 个表达式的真假。依此类推，直到找到凶手或尝试完所有情况为止。每次验证某人是否为凶手都是重复的运算，为此我们使用了 for 语句。

解决这个问题的源程序代码如下：

```
#include<stdio.h>

int main()
{
    int  i, sum=0, flag=0;
    char killer;

    for(i=1; i<=4; i++)
    {
        killer = 64 + i;

        sum = (killer!='A') + (killer=='C') + (killer=='D') + (killer!='D');

        if(sum == 3)
        {
```

我爱自由 5ifreedom.com

```
            flag = 1;
            printf("%c is the killer.\n",killer);
            break;
        }
    }

    if(flag == 0)
        printf("Can not find\n");

    return 0;
}
```

程序说明

（1）变量 sum 用于统计假设某人为凶手的情况下有几个人说了真话，flag 用于指示是否已经找到了解，设置初值为 0，假定没有找到解。

（2） for 循环逐一尝试每一个可能的解，本问题只有 4 个可能的解，所以只循环 4 次。当 i 等于 1 时，将 i 的值加上 64 就是字符 A 的 ASCII 码，此时变量 killer 存储的是字符 A。然后统计假如 A 为凶手，有多少人说了真话。在 killer 值为 A 时，sum 值为 1，所以 A 不是凶手。当 killer 值为 C 时，sum 为 3，就可以判定 C 为凶手，此时打印出 C 为凶手的信息，并把 flag 置为 1。

（3）如果程序执行到最后 flag 仍为 0，则说明没有找到解，此时打印出没有找到凶手的信息。

例 3-12 猜数字游戏。

计算机随机产生一个 1～100 之间的数，让用户来猜。如果猜对了给出提示“Wonderful，you are right!”，如果猜错了就提示“Sorry，you are wrong.”，并告诉用户是猜大了（too high）还是猜小了（too low）。最多只能猜 8 次。

实现源程序代码如下：

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

int main()
{
        int number, guess, i=8;

        srand(time(NULL));
        number = rand() % 100 + 1;

        printf("Please guess a number:\n");

        while(i > 0)
        {
            scanf("%d",&guess);

            if(guess == number)
            {
                printf("Wonderful,you are right!\n\n");
                return 0;
            }
            else if(guess > number)
            {
                printf("Sorry,your are wrong.");
                printf("too high\n");
            }
            else
            {
                printf("Sorry,your are wrong.");
                printf("too low\n");
            }
            i--;
        }
```

我爱自由 5ifreedom.com

```
        printf("Game over!\n\n");
        return 0;
}
```

程序说明。

（1） 程序包含 3 个头文件，通过调用后两个头文件中提供的库函数来产生一个 1～100 之间的随机数。

（2）程序首先定义了 3 个变量，number 用于存储由计算机产生的随机数，guess 用于存放用户输入的数，i 用于循环计数，由于最多只能猜 8 次，所以初始值设为 8。

（3）标准库函数 srand 为函数 rand 提供设置随机数种子，rand 用来产生一个随机数，通过对由 rand 产生的随机数进行取余并加 1 运算，把随机数限制在 1 到 100 之间。

（4） while 循环最多执行 8 次，如果用户在 8 次以内猜出了由计算机随机产生的数字就调用 return 语句结束程序。否则，给出提示信息，告诉用户是猜大了还是猜小了。如果用户猜了 8 次还是没有猜出数字就打印出“Game over!”。

3.6 习题

1．写出下面各个逻辑表达式的值。a、b、c 的值分别为 3、4、5。

（1）a + b > c && b == c

（2）a || b && b - c

（3）!(a > b) && !c || 1

（4）!(x = a) && (y = b) && 0

（5）!(a + b) + c - 1 && b + c / 2

2．用逻辑表达式、for 循环语句求解逻辑题。5 位运动员参加了 10 米台跳水比赛，有人让他们预测比赛结果

A 选手说：B 第一，我第三。

B 选手说：我第二，E 第四。

C 选手说：我第一，D 第二。

D 选手说：C 最后，我第三。

E 选手说：我第四，A 第一。

比赛结束后，每位选手都说对了一半，请编程确定比赛的名次。

3．用穷举算法求解爱因斯坦数学题。爱因斯坦曾出过这样一道题目：有一条长阶梯，若每步跨 2 阶，最后剩 1 阶；若每步跨 3 阶，最后剩 2 阶；若每步跨 5 阶，最后剩 4 阶；若每步跨 6 阶，最后剩 5 阶；若每步跨 7 阶，最后一阶不剩下，请问这条阶梯共有多少阶？

我爱自由 5ifreedom.com

第 4 章　C 函数、数组、指针和调试器 gdb

一般较大的程序都由多个模块构成，每个模块完成一个特定的功能，在 C 语言中模块通常由一个或多个函数构成。数组可以连续存放同一类型的多个数据。数组中的元素由数组名和下标来惟一确定。指针是 C 语言的精华，指针有很多用途，使用指针可以方便地访问数组，可以动态分配内存、可以使函数返回多个值。

本章介绍了函数、数组、指针的定义和使用。此外，还详细介绍了调试器 gdb 的使用。在本章的最后列举了一些知名软件公司招聘时使用过的试题。

本章重点：

- 函数的定义和调用。
- 数组的定义和使用。
- 指针的定义和使用。
- 调试器 gdb 的使用。

本章难点：

- 函数的递规调用。
- 指针与数组、函数的关系。
- 程序的调试方法。

4.1 函数

4.1.1 函数的定义

1. 函数定义的一般形式

函数（function）定义的一般形式为：

```
返回类型 函数名（形式参数列表）
{
    函数体
}
```

在 C 语言中，函数由函数名惟一标识。函数名是一个标识符，它不能与其他函数同名，否则在函数调用时，系统不知道该调用哪一个。但在作为对 C 语言进行扩展的 C++语言中，函数允许重名。

函数的返回类型是调用函数时返回值的类型，可以是任何基本数据类型或指针。当函数不返回任何值时，应该定义函数的返回类型为 void。如果在定义一个函数时，没有明确指明返回类型，Linux 上最常用的编译器 gcc 默认返回类型为 int。

形式参数是函数的操作数，形参与形参之间以逗号隔开。

函数执行运算的部分称为函数体。在函数体内可以定义变量，对形式参数进行操作，返回数据。

我爱自由 5ifreedom.com

下面定义了一个函数，用于求两个数中的较大者，代码如下所示：

```
int max(int a,int b)
{
    int c;
    c = a>b ? a : b;
    return c;
}
```

在第 1 行中，第一个关键字 int 表示这个函数的返回类型为整型。max 是函数名，括号中是两个形式参数，它们都是整型的，形参之间以逗号隔开。函数体中第 1 行定义了一个整型变量 c，第 2 行将 a 和 b 中的较大值赋给 c，函数体的最后一行，通过使用 return 语句将变量 c 的值返回给调用者。这里我们注意到，变量 c 的类型和函数的返回类型是一致的，都是 int 型。下面的语句调用了这个函数：

```
int i , j = 10 , k = 15;
i = max(j,k);
```

2．函数的参数

下面是一个完整的例子。

例 4-1

```
#include<stdio.h>

int max(int a,int b)
{
    int c;
    c = a>b ? a : b;
    return c;
}

void main( )
{
    int i , j = 10 , k = 15;
    i = max(j,k);
    printf("i=%d\n",i);
}
```

程序说明。

在 main 主函数中调用了自定义的 max 函数。在定义函数时，函数名后面括号中的变量称为形式参数，如变量 a、b，简称形参。在调用函数，函数名后面括号中的参数称为实际参数，如变量 i、j，简称实参。实参可以是变量也可以是常量或表达式。例如：

```
i = max(20,j+k);
```

第一个实参是整型常量 20，第二个参数是一个表达式。

在定义函数中指定的形参，在这个函数被调用前，它们不占内存的存储单元。只有在发生函数调用时，函数 max 中的形参才被分配内存单元。在调用结束后，形参所占的内存单元即被收回。

实参与形参的类型应该相同或者两者间可以进行转换。下面的调用是合法的：

```
i = max(25.5,j+k);
```

那么 i 的值是多少呢？在进行函数调用时，25.5 被赋值给 int 型的形参 a，此时要发生转换，a 实际被赋值为 25，形参 b 被赋值为 j、k 的和 25。因此最后 i 的值为 25。

函数的形式参数列表即形参表可以为空，此时函数没有形参。

C 语言规定，实参对形参的数据传递是单向的“值传递”。实参只是将它自己的值赋给形参，形参若在函数中值被改变并不会影响实参。事实上，实参的值和形参的值存放在内存的不同单元里。在发生函数调用时，形参被分配内存单元，内存单元被初始化为实参的值，此时，形参的值和实参的值是一样的。如果在函数内形参的内存单元存储了其他值，例如，在 max 函数体内将 100 赋给形参 a，此时实参和形式参的值就不一样了。函数调用结束也就是函数体执行完毕后，分配给形参的那块内存被系统回收，但存储实参的内存单元仍旧还在并且

我爱自由 5ifreedom.com

其值并未发生变化。

注意：一个函数的各个参数不可以重名，函数体内定义的变量也不可以与参数重名。

3. 函数的返回值

通常，希望通过调用某个函数来获取一个值，这个值就是函数的返回值。函数通过 return 语句将一个值返回给调用它的函数。当然一个函数也可以不返回任何值，此时函数仅仅完成某些操作。一个函数体内可以有多条 return 语句，当实际执行到第一条 return 语句时，函数就执行完毕。

如果一个函数不返回任何值，应该把函数返回类型指定为 void（即空类型），例如：

```
void someoperate(float f);
```

如果函数的返回类型与 return 语句返回的值类型不一致，以函数的返回类型为准。对于数值类型将自动进行类型转换。

注意：函数体内定义的变量在函数外是不可见的，外部不能访问。函数体内定义的变量随着函数被调用而分配内存空间，函数调用结束存储函数体为定义变量的内存空间即被回收。函数可以返回一个指针但不可以返回一个数组。数组和指针将在本章后面介绍。

4.1.2 函数的调用

1. 函数调用

C 语言使用调用操作符（即一对圆括号）实现函数的调用。正如其他操作符一样，调用操作符需要操作数并产生一个结果。调用操作符的操作数是函数名和一组由逗号分隔的实参（实参可以为空）。函数调用的结果类型就是函数返回值的类型。

函数调用的执行流程是：用对应的实参初始化函数的形参，并将控制权移交给被调用函数。主调函数的执行被挂起，被调函数开始执行。被调函数执行完毕后，将结果返回给主调函数同时把控制权交回给主调函数。

事实上，任何 C 程序都必须有且只有一个名为 main 的函数，称为主函数。Linux 在执行一个 C 程序时，把控制权交给 main 函数，main 函数执行函数体内的语句，在执行过程中它可能会调用其他函数，如上一节的程序中 main 函数调用 max 函数，此时它把控制权移交给那个函数，那个函数执行完毕后把控制权交回给 main 函数，main 函数执行完毕后再把控制权交给系统。Linux 中默认 main 函数的返回类型为 int，main 函数通过向操作系统返回一个数以报告它的执行情况，通常返回 0 表示程序正常结束。但操作系统一般忽略 main 函数的返回值。

函数调用的一般形式为：

```
函数名（实参列表）;
```

如果被调用的函数没有形参，则实参为空。实参之间以逗号隔开。实参与形参个数应相等，类型一致。实参与形参按顺序对应。

函数通常按以下几种方式进行调用。

① 函数调用作为一条语句，例如：

```
printstring( );
```

② 函数调用作为一个表达式，例如：

```
int c = 2 * max(a,b);
printf("%d",max(a,b));
```

③ 函数调用作为一个函数的实参，例如：

```
int m = max(a,max(b,c));
```

一个函数调用另一个函数需要具备哪些条件呢？

（1）被调用的函数必须是存在的。程序中可以调用自己定义的函数，也可以调用库函数。

我爱自由 5ifreedom.com

（2）如果调用的是库函数，一般应该在源程序文件的开头使用#include 指令将相应的头文件包含到源文件中。例如，如果要调用求开方库函数 double sqrt（double x）就应该使用以下指令包含数学函数库头文件 math.h，例如：

```
#include<math.h>
```

.h 是头文件的后缀，表示它是一个头文件（header file）。

（3）如果调用的是用户自定义的函数，而且该函数与调用它的函数在同一个文件中。若调用语句在函数定义之前，则应该对函数先进行声明。若调用语句在函数定义之后，则不必对函数进行声明，例 4-2 演示了自定义函数的方法。

例 4-2

```
#include<stdio.h>

int myabs(int x);

void main()
{
    int x;
    scanf("%d",&x);
    x = myabs(x);
    printf("%d\n",x);
}

int myabs(int x)
{
    return ( x > 0 ? x : -x );
}
```

程序说明。

在此实例中，main 主函数调用 myabs（用于求绝对值）是在该函数定义之前进行的。为此，要在调用语句之前先声明该函数。定义是对函数功能的确立，包括指定函数名、返回类型、形参及其类型、函数体。声明仅仅通知编译器 gcc，程序中有这么一个函数。声明只要告诉编译器函数名、返回类型、形参及其类型而不需要定义函数体。函数声明事实上就是函数定义的第一行加一个标识一个语句结束的分号。函数声明还有一种简化形式，即不需要指明形参的变量名，例如：

```
int myabs(int );
```

函数声明的一般形式是：

```
函数类型 函数名（参数类型 1 ，参数类型 2 . . . ）；
函数类型 函数名（参数类型 1 参数名 1，参数类型 2 参数名 2. . . ）；
```

此例中若把 int 型改为 float 型或 double 型是否有一样的结果？事实是不能正确得到一个数的绝对值，详细的原因请见第 2 章面试题选部分。

2．函数的嵌套调用和递归调用

一个函数的函数体内能否调用另一个函数？可以，这就是嵌套调用。例 4-1 和例 4-2 中，main 函数中都调用了另外一个函数。

那么能不能在一个函数的函数体再完整地定义一个函数？答案是不可以。读者不妨尝试一下在 main 主函数定义一个函数，然后进行编译，看看编译器 gcc 会提示什么样的错误信息。

注意：函数可以嵌套调用但不可以嵌套定义。

一个函数能否在函数体内调用自己？可以，这就是递归调用。在调用一个函数过程中又出现直接或间接地调用该函数本身，称为函数的递归调用。

函数的递归调用是一种很有用的编程技巧。程序有 3 种基本的控制结构：顺序、分支和循环。事实上可以使用递归调用来实现循环。对于某些问题，使用递归法可以很容易地解决。

例 4-3 分别用循环和函数的递归调用来求 10!。

我爱自由 5ifreedom.com

求 n!的一种方法是从 1 开始，先求 1!，然后乘 2 得到 2!，再乘 3 得到 3! ... 一直乘到 n 得到 n!

另外一种方法是将求 n!转化为求 n*(n-1)!，这就把原问题求 n!转化为求(n-1)!，这一步转化对于规模较小的简单问题也许算不了什么，但对于一些规模较大的复杂问题，这可以减少很大的计算量。然后在把(n-1)!转化为求(n-2)!，以此类推直到问题归结为求 1!。

思路一是从解决一个简单的问题开始逐步去解决一个规模更大的复杂问题，思路二是将一个规模较大的复杂问题通过某种方法转化为规模较小的简单问题。有的问题随着问题规模的扩大其计算量不是以线性的形式增长而是以指数形式增长。即使是最先进的计算机在面对某些问题时也是无能为力的。而通过思路二的转化方法，也许可以很快地找到可行解或近似解。对于某些问题，得到可行解或近似解就可以了而不一定要得到最优解。

利用思路一，程序代码如下。

```
#include<stdio.h>

main()
{
    int   i;
    float f = 1;

    for(i=1; i<=10; i++)
        f = f * i;

    printf("10! = %10.0f\n",f);
}
```

利用思路二，程序代码如下。

```
#include<stdio.h>

float fac(int n)
{
    float f;
    if(n<0)
    {
        printf("n<0,data error!");
        return -1;
    }
    else if( n==0 || n==1)
        f = 1;
    else
        f = n * fac(n-1);
    return f;
}

main()
{
    int n = 10;
    float f;
    f = fac(n);
    printf("n! = %10.0f\n",f);
}
```

程序说明。

在递归函数 fac 中，首先判断形参 n 是否小于 0，若小于 0，求 n!是没有意义的，程序打印出错误信息并返回-1。

然后判断 n 是否等于 0 或等于 1，如果是，那么问题已经达到最小规模了。0!=1，1!=1。此时函数不会再递归调用下去。如果 n 大于 1，就把原问题转化为 n*fac(n-1)，进行递归调用，即把参数由 n 转变为 n-1。现在假设 fac(n-1)返回了一个数，把这个数乘以 n，即进行 n*fac(n-1)运算，并把值赋给 f，然后函数返回 f，就得到了 n!。要保证递归调用到一定条件将不会继续递归下去而是返回一个值。这个条件在这个程序中是 n==0 或者 n==1，这也是设计递归调用函数时必须注意的地方。

我爱自由 5ifreedom.com

4.1.3 变量的访问控制和存储类别

在第 2 章中我们知道了怎样来定义一个变量，但仅仅掌握如何定义一个变量是远远不够的，还应该理解和掌握变量的访问控制和存储类型。

1. 变量的访问控制

变量按照它在程序中被访问范围可分为两类：局部变量和全局变量。局部变量是指在一个函数内部定义的变量，它只在函数内部有效，函数外部不能对其进行访问，例如：

```
main()
{
    int i,j;
    . . .
}
int function1(int a,int b)
{
    int i;
    . . .
}
float function2(int a)
{
    int m;
. . .
}
```

每个函数内部定义的变量只能在函数内部使用，只是在函数内可见。主函数也不例外，主函数定义的变量 i 和 j 也只能在主函数内使用。不同的函数可以定义相同名字的变量，它们不存在任何关系，例如 function1 函数中也定义了一个名为 i 的变量，它和主函数内部定义的变量 i 没有任何关系。形式参数也是局部变量，因此 function1 的形参 a 与 function2 的形参 a 没有任何关系。这样的使用并不违反 C 语言的规则。

一个源程序文件通常由多个函数构成，如果某个变量不在任何函数内被定义，则这个变量就是全局变量。全局变量可以被该源文件内的任何函数访问。它的有效范围是从该变量的定义处开始到文件结束。

事实上，在 C 语言中，一对大括号（{ }）标识了一个变量的作用域。定义函数时，必须有一对大括号，在该作用域内定义的变量的作用范围是从该变量定义处开始到该作用域结束，即到右大括号为止。复合语句是由一些语句和一对大括号构成，这对大括号也是一个作用域。在复合语句内部定义的变量也只能在该复合语句内部使用，这个作用域外，系统就认为它是一个未定义的符号（undefined symbol）。此外，if 语句、switch 语句、for 语句等后面所跟的一对大括号都是如此。

作用域可以嵌套，即在一个作用域里可以有任何数目的作用域，例如：

```
int main()
{
    int a,b,c;
    . . .
    {
        float a,b;
        . . .
        c = a*b;
    }
    . . .
}
```

main 函数开始处定义的整型变量 a、b、c 的作用域从它们的定义处开始到 main 函数结束。整型变量 a、b、c 在 main 函数内部的作用域内（即浮点型变量 a、b 所在的作用域）也可以使用。在一个作用域内（如浮点型变量 a、b 所在的作用域）可以定义与它所在的作用域（main 函数的作用域）相同名字的变量，如浮点型变量 a、b 的定义也是合法的，此时在浮点型变量 a、b 所在作用域内自动屏

我爱自由 5ifreedom.com

蔽掉整型变量 a、b，因此语句 c = a*b 中 a 和 b 是浮点型的 a、b。

对于一个源程序文件，可以假想该文件的所有内容均处在一对大括号中，即在一个作用域里。源程序文件内定义的函数规定了一个作用域，在该函数内定义的变量只能在该作用域使用。全局变量处在包含该函数的更大作用域内，当然可以在该函数内使用。

通过全局变量可以在两个不相关的函数间建立联系，但是这是违反模块化编程准则的，所以建议最好少用全局变量。模块化准则就是把一个大问题分解为许多小问题，每个小问题由一个函数来解决，每个函数都完成一个特定的功能，各个小问题应该尽量独立，即所谓高内聚，低耦合。函数内部应该是高内聚，完成一些紧密相关的任务，函数之间应该只有一些非常必要的联系，即低耦合。

2. 变量的存储类别

变量根据存储类别可以分为静态存储变量和动态存储变量，或简称为静态变量和动态变量。所谓动态变量是指在程序运行过程中根据需要动态分配内存空间的变量。所谓静态变量是指在程序运行期间分配固定的存储空间的变量。

动态变量主要有：函数的形参、函数内定义非 static 变量。静态变量有：函数内定义的 static 变量、全局变量。

下面这个例子定义并使用了 static 变量。

例 4-4

```
#include<stdio.h>

void function()
{
    static int a = 0;
    int b = 0;
    a++;
    b++;
    printf("a=%d,b=%d\n",a,b);
}

main()
{
    function();
    function();
    function();
}
```

结果输出

```
1, 1
2, 1
3, 1
```

程序说明

整型变量 a 是一个 static 变量，变量 b 是一个非 static 变量。静态变量在函数调用结束后并不会销毁。原因在于，虽然同是 function 函数内定义的变量但它们存储在不同的内存区域。变量 b 随着 function 函数被调用而在动态存储区分配内存空间，调用结束后系统收回那块内存空间，保存的变量 b 也就不存在了。而在函数第一次被调用时，static 变量被分配了一块静态存储区的内存，在函数调用结束后，系统并不会回收这块内存。下一次函数调用时，系统重新为变量 b 分配内存空间，而此时系统不会再为变量 a 分配内存空间而是继续使用上次分配给 a 的空间，因此变量 a 的值得以保存下来。程序运行结束，系统才收回分配给变量 a 的内存。

全局变量是在所有函数的外部定义的，它的作用域从变量的定义处开始到本程序文件结束。在此作用域内，全局变量可被所有函数使用。编译时系统为全局变量在静态存储域分配内存空间。但在一个文件中如果要在全局变量的定义前使用该变量是否可以呢？根据 C 语言“先定义后使用”

我爱自由 5ifreedom.com

的规则，原则上是不可以的，但通过使用 extern 关键字提前声明全局变量就可以了，例 4-5 演示了全局变量定义的用法。

例 4-5

```
#include<stdio.h>

int max(int a,int b)
{
    return a>b?a:b;
}

main()
{
    extern i,j;
    printf("%d\n",max(i,j));
}
int i = -5,j = 10;
```

程序说明。

为了在 main 函数的 printf 中提前使用全局变量 i 和 j，需要在使用变量 i、j 前先通过 extern 关键字对变量 i、j 进行声明，告诉编译器 i 和 j 在别处被定义，否则编译器认为 i 和 j 是未定义符号。声明仅仅是告诉编译器程序存在这个变量，这个变量并不是未定义的，不需要分配内存，而定义则要分配内存空间以存储变量的值。

在进行大型程序或商用程序开发时还有一种情况要使用 extern 关键字。规模较大的程序往往由多个源文件构成，如果一个文件中的程序代码要使用另外一个文件内定义的全局变量，就要使用 extern 对该全局变量进行声明，例 4-6 演示了 extern 对全局变量进行声明的用法。

例 4-6

源文件 file1.c

```
#include<stdio.h>

extern long power(int);
int A = 2;

main()
{
    int n = 10,total;
    total = power(n);
    printf("2^10 = %d \n",total);
    return 0;
}
```

源文件 file2.c

```
extern A;
long power(int n)
{
    long total = 1;
    int i;
    for(i=1;i<=n;i++)
        total = total * A;
    return total;
}
```

编译并运行该多文件程序：

```
gcc file1.c file2.c -o file
./file
```

程序输出：

```
2^10 = 1024
```

程序说明。

在该程序中，file2.c 引用了 file1.c 中的全局变量 A，为此要在 file2.c 通过 extern 关键字对外

我爱自由 5ifreedom.com

部变量 A 进行声明。类似的，file1.c 引用了 file2.c 中定义的函数 power，为此要在 file1.c 开始处对所引用的函数进行声明。

4.2 数组

到目前为止，介绍的都是基本数据类型（整型、字符型、浮点型等），C 语言还提供了 3 种复合数据类型：数组、结构体、共用体。复合数据类型是由基本数据类型按照一定的规则构造而成的。

4.2.1 一维数组的定义和使用

数组是由类型名、标识符、维数组成的复合数据类型。类型名规定了存放在数组中的元素的类型，而维数则指定了数组中包含的元素个数，标识符用来标识这个数组。

数组定义的一般形式是：

```
数据类型 标识符[元素个数]
```

例如，以下定义了一个含有 10 个整型元素的数组：

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

数组元素的类型可以是基本类型也可以是复合数据类型。标识符的命名规则和一般变量名相同。元素个数指定了数组中元素的个数，它必须是常量表达式。

下面定义的数组是不合法的：

```
int n = 10;
int a[n];
```

n 是变量而不是常量或常量表达式。

数组和变量一样，也是先定义后使用，C 语言规定只能引用数组的一个元素而不能一次引用整个数组，数组引用的一般形式是：

```
标识符[下标]
```

例如：数组 a[10]的 10 个元素分别为 a[0]，a[1]，a[2]，a[3]，a[4]，a[5]，a[6]，a[7]，a[8]，a[9]。

```
a[0] = 10;
```

把 10 赋给数组的第一个元素。

在定义数组时可以进行初始化，例如：

```
int a[10] = {9,8,7,6,5,4,3,2,1,0};
```

则 a[0]被赋予了初值 9，a[1]为 8，依此类推。

可以为数组的部分元素指定初始值，例如：

```
a[10] = {9,8};
```

a[0]、a[1]被赋予了初值 9、8，其余元素被自动赋予初值 0。

如果定义数组时，它所有的元素都进行了初始化就不用指定元素的个数，例如下面也定义了一个含有 10 个整型元素的数组。

```
int a[] = {9,8,7,6,5,4,3,2,1,0};
```

注意：在所有函数外定义的数组的所有元素将被自动赋予初值 0，在函数内部定义的数组，系统不会为其进行初始化，在使用数组元素前必须先对元素进行初始化。

4.2.2 二维数组的定义和使用

二维数组定义的一般形式为：

```
数据类型 标识符[第一维维数] [第二维维数]
```

第一维通常称为行，第二维则称为列，与一维数组一样，二维数组的维数也必须是常量表达式。

我爱自由 5ifreedom.com

例如：

```
float array[3][4];
```

定义了一个 3 行 4 列的数组。它的所有元素为：

```
a[0][0]   a[0][1]   a[0][2]   a[0][3]
a[1][0]   a[1][1]   a[1][2]   a[1][3]
a[2][0]   a[2][1]   a[2][2]   a[2][3]
```

可以用以下方法对二维数组进行初始化。

（1）将所有数据写在一个大括号内，例如：

```
array[3][4] = {1,2,3,4,5,6,7,8,9,10,11,12};
a[0][0]、a[0][1] 到a[2][3]依次被赋初值1，2到12。
```

（2）分行赋值

```
array[3][4] = { {1,2,3,4}, {5,6,7,8}, {9,10,11,12} };
```

这种赋值比较直观，第 1 个大括号内的数据赋给第一行，第 2 个大括号内的数据赋给第二行，依此类推。

（3）部分赋值

```
array[3][4] = { {1}, {2,3}, {4,5,6} };
```

以这种方式赋值，array 数组的所有元素为：

```
1, 0, 0, 0
2, 3, 0, 0
4, 5, 6, 0
```

这种方式在数组的大部分元素的初值为 0 的情况下使用比较方便。

也可以只对某几行元素进行赋值，例如：

```
array[3][4] = { {1}, {2,3} };
```

则各个元素的值为：

```
1, 0, 0, 0
2, 3, 0, 0
0, 0, 0, 0
```

（4）如果对全部元素都赋初值，则定义数组时第一维的维数可以省略，但第二维的维数必须指定，例如：

```
array[][4] = {1,2,3,4,5,6,7,8,9,10,11,12};
```

系统自动计算第一维的维数，第一维的维数是 12 / 4，即 3。

或者：

```
array[][4] = { {1}, {}, {4,5,6} };
```

系统会自动识别该数组的第一维的维数是 3。这个数组的各个元素值为：

```
1, 0, 0, 0
0, 0, 0, 0
4, 5, 6, 0
```

例 4-7

将一个二维数组的行和列元素互换，存到另一个二维数组中，如有一个二维数组：

```
1, 2, 3, 4,
5, 6, 7, 8
```

则交换后的数组为：

```
1, 5
2, 6
3, 7
4, 8
```

程序代码如下：

```
#include <stdio.h>

int main()
{
    int a[2][4] = { {1,2,3,4},{5,6,7,8} };
```

我爱自由 5ifreedom.com

```
    int b[4][2],i,j;

    printf("array a:\n");
    for(i=0;i<2;i++)
    {
        for(j=0;j<4;j++)
        {
            printf("%5d",a[i][j]);
            b[j][i] = a[i][j];
        }
        printf("\n");
    }

    printf("array b:\n");
    for(i=0;i<4;i++)
    {
        for(j=0;j<2;j++)
        {
            printf("%5d",b[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

程序运行结果：

```
array a:
1  2  3  4
5  6  7  8
array b:
1    5
2  6
3  7
4  8
```

4.2.3 字符数组和字符串

用来存放字符的数组就是字符数组。字符数组中的每一个元素都存放一个字符。字符数组和一维数组本质上没有区别，字符数组是类型为 char 型的一维数组。单独介绍字符数组是因为字符数组在实际项目开发中用得比较多。

字符数组的定义和前面介绍的类似，例如：

```
char str[10];
str[0] = 'h';  str[1] = 'e'; str[2] = 'l' ; str[3] = 'l' ; str[4] = 'o';
```

也可以在定义时就进行初始化：

```
char str[10] = { 'h', 'e', 'l', 'l', 'o'};
```

在对字符数组赋值时，如果大括号中字符的个数小于字符数组的长度，则把大括号中的字符赋给数组中前面的元素，其余元素自动赋值为空字符（即‘\0’）。在上面这条语句中，str[5]到 str[9]被自动赋值为‘\0’。如果大括号中字符的个数大于字符数组的长度，则编译程序时编译器会报错。

以下语句打印出 str 数组的所有元素：

```
for(i=0;i<10;i++)
    printf("%c",str[i]);
```

在 C 语言中，字符串是用字符数组来存储的。这里要注意区别字符串长度和字符数组长度，比如定义了一个长度为 10 个字符的字符数组而实际存放在该数组中的字符串的长度为 5。为了测定一个字符串的长度，C 语言用字符‘\0’来代表一个字符串的结束。

对于语句：

```
char str[10] = { 'h', 'e', 'l', 'l', 'o'};
```

字符数组 str 实际存储了一个字符串“hello”，str[5]到 str[9]自动被赋值为‘\0’。系统发现 str[5]

我爱自由 5ifreedom.com

值为‘\0’，就认为字符串结束了。

系统自动为字符串常量增加一个字符‘\0’作为结束符。例如，“Linux C”共有 7 个字符，但系统为它分配 8 个字节的内存容量，第 8 个字节存储‘\0’。可以试想，内存是一片连续的存储空间，如果不为每个字符串分配一个字节用以存储字符串的结束标志，那么系统就很难确定一个字符串在内存中占据了多少存储空间。字符‘\0’代表 ASCII 码为 0 的字符，该字符是一个不可打印的字符，它只是一个空操作符，起一个标志的作用。

可以用字符串来初始化字符数组：

```
char str[] = {"hello"};
```

或者

```
char str[] = "hello";
```

可以用以下语句打印字符串：

```
printf("%s",str);
```

注意： 实际编程中，在定义字符数组时应估计实际字符串的长度，保证数组长度始终大于字符串的长度。另外，编程中应时刻记着检查要存储的字符串的长度是否超过了字符数组的长度，否则很容易导致缓冲区溢出。现在网络上大部分的安全漏洞都与缓冲区溢出有关，许多病毒利用缓冲区漏洞进行攻击，造成极大的危害。

4.3 指针

4.3.1 地址和指针

指针是 C 语言的一个重要特性，灵活地运用指针可以解决很多问题。同时指针也是一个难点，即使有丰富编程经验的程序员有时在调试有指针使用错误的程序时也感到困惑。

在第 1 章里我们简单地介绍过指针。在这里再深入地介绍指针的概念。对于程序中的每一个变量，在编译程序时都要分配一块内存用于存储该变量的值，例如：

```
char c = 'A';
int  i = 100;
```

编译器分别给变量 c、i 分配 1 个字节和 4 个字节的存储空间。如图 4-1 所示。

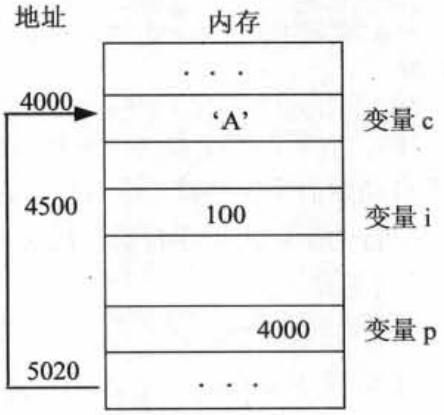


图 4-1　变量在内存中的存储空间

内存中每一个字节都有一个编号，第一个字节被编为 0，第二个被编为 1…这个编号就是内存的地址。变量 c 的内容被存放在编号为 4000 的存储单元内，也就是地址为 4000 的内存单元中。变量 i 的内容被存放在以地址 4500 开始的 4 个字节的内存中，对于整型变量通常要分配 4 个字节

我爱自由 5ifreedom.com

来存储该变量的值。

假定定义了一个指针变量 p：

```
char *p = &c;
```

这个指针变量 p，用于存放一个 char 类型变量的地址。我们在定义该变量时对它进行初始化，把变量 c 的地址赋给变量 p，&c 用于取得变量 c 的地址。则变量 p 存放的是一个地址，这个地址的值为 4000。此时我们说指针 p 指向了变量 c。

以下语句使用了变量 p：

```
char c2 = *p + 1;
```

这里*p 的含义是取得指针变量 p 所指向的变量的值，也就是取得变量 c 的值‘A’，因此变量 c2 的值为‘B’。

注意：char *p 定义了一个指向 char 型变量的指针，p 存放的是一个 char 型变量的地址。而之后的*p 是取得指针 p 所指向的变量的值，&c 取得变量 c 的地址。

4.3.2 指针的定义和使用

C 语言中的指针是指专门用来存放内存地址的变量。每个指针都有一个与之关联的数据类型，该类型决定了指针所指向的变量的类型，例如，一个 char 型指针只能指向 char 型变量。

C 语言使用*把一个标识符声明为指针，指针定义的一般形式为：

```
数据类型 *指针变量名;
```

例如：

```
char     *pc;
int      *pi,p;
doule   *pd = NULL;
```

其中第二行定义了一个指向整型变量的指针和一个整型变量。第三行定义一个指针并初始化为 NULL，表示该指针不指向任何变量。第一行和第二行的指针由于在定义时没有进行初始化，因此它们所指向的变量是未定的。使用未初始化的指针通常会导致错误，应该绝对避免使用未初始化的指针。如以下形式的代码不应该在任何程序中出现：

```
int   *p;
int  i = *p;
```

虽然这两行代码在语法上是正确的，但在第二行引用了未初始化的指针。指针 p 没有明确地指向某个变量，在此情况下要取得指针 p 所指向的变量的值是危险的。编译含有类似代码的程序时，编译器通常会发出警告，但可以通过编译，在运行时往往会导致程序崩溃。

注意：避免使用未初始化的指针。在定义指针时最好将它初始化为 NULL，即明确指示当前该指针不指向任何变量。

对于指针有两个与其相关的运算符如下所示。

```
&:     取地址运算符。
*:  取指针所指向的内存单元的值。
```

例 4-8 使用指针

```
#include <stdio.h>

int main()
{
    int a = 100,b = 200;
    int *p1 = &a,*p2 = &b;

    printf("%d,%d\n",a,b);
    printf("%d,%d\n",*p1,*p2);
    printf("%x,%x\n",&a,&b);
    printf("%x,%x\n\n",p1,p2);
```

我爱自由 5ifreedom.com

```
    *p1 = *p1*3;
    printf("%d\n",a);
    printf("%d\n\n",*p1);

    p1 = &b;
    printf("%x\n",p1);
    printf("%x\n",p2);

    return 0;
}
```

程序运行结果：

```
100, 200
100, 200
bfffece4, bfffece0
bfffece4, bfffece0

300
300

bfffece0
bfffece0
```

程序说明

（1）程序首先定义了两个指针 p1、p2，分别指向变量 a 和 b。程序输出的第一、二行是一样的，因为*p1、*p2 取得的是它们所指向的变量的值。

（2）第 3 行以十六进制的形式输出变量 a、b 的地址，第 4 行输出的是指针变量 p1、p2 的值。变量 p1、p2 存储的是它们所指向的变量的地址，p1 存放的是变量 a 的内存地址，p2 存放的是变量 b 的内存地址。因此第 3、4 行的结果是一样的。

（3）语句*p1 = *p1*3，首先取得指针 p1 所指向变量的值即 100，乘 3 后将结果保存到 p1 所指向的变量中，此时变量 a 的值由 100 变为 300。因此程序第 6、7 行都输出 300。

（4）语句 p1 = &b，将变量 b 的地址赋给指针变量 p1，此时 p1 和 p2 都指向变量 b，这两个指针变量存储的都是变量 b 的地址，因此程序输出的最后两行是一致的。

4.3.3 指针和数组

C 语言中，指针和数组密切相关。数组名实际上就是指向数组第一个元素的指针，下面看一段程序：

```
int  a[] = {2, 4, 6, 8, 10};
int  *p1 = a;
int  *p2= &a[0];
```

指针 p1 指向数组 a 的第一个元素，即 a[0]，指针 p2 也是如此。指针变量 p1 和 p2 存放的都是 a[0]的地址。

在指向数组某个元素的指针上加上或减去一个整型数值，就可以指向另外一个数组元素，前提是没有超出数组的范围，例如：

```
p1 = p1+3;
```

此时 p1 指向 a[3]，存放数组元素 a[3]的地址。为什么可以进行这样的操作呢？事实上，数组的所有元素在逻辑上是按顺序存放的，假设 a[0]的地址也就是数组 a 的起始地址为 4000，每个整型变量在内存中占 4 个字节，则 a[1]存放在以地址 4004 开始的 4 个字节的内存中，依然类推。对于表达式 p1+3，它指向数组中 p1 当前元素后的第 3 个元素，即 a[3]。简单地说，如果指针变量 p 指向数组中的一个元素，则 p+1 指向同一数组中的下一个元素。

注意：在指针上进行加减运算后所得到的指针，必须指向同一个数组或指向数组存储空间的下一个单元。

我爱自由 5ifreedom.com

对于：

```
p1 = a+10;
```

是不合法的。因为该数组只有 5 个元素。但以下语句是合法的：

```
p1 = a+5;
```

虽然该数组只有 5 个元素，从 a[0]到 a[4]，但 p1 可以指向数组存储空间的下一个位置。但不能对该变量执行*p1 运算，即不能获得此时指针 p1 所指向的变量的值。

引用一个数组元素，既可以用下标的方式，即 a[i]，也可以用指针的方式，即*(a+i)或*p，这里假设指针 p 指向 a[i]。

例 4-9 以不同的方式输出数组的所有元素的值

```
#include <stdio.h>

int main()
{
    int a[10],i,*p = NULL;

    for(i=0; i<10; i++)
        a[i] = i;

    for(i=0; i<10; i++)
        printf("%d ",a[i]);
    printf("\n");

    for(i=0; i<10; i++)
        printf("%d ",*(a+i));
    printf("\n");

    for(p=a; p<a+10; )
        printf("%d ",*p++);
    printf("\n");

    return 0;
}
```

程序运行结果：

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

程序说明

（1）程序以 3 种不同的方式输出数组的各个元素值。第一种方式通过数组名和下标的方式访问数组的所有成员。

（2）由于数组名也是一个指针，它总是指向数组第一个元素，因此第二种方式也是可行的。第一、二种方式实际上是等价的，只是写法上不同而已。

（3）第三种方式效率最高。这里要注意表达式*p++，++和*优先级相同，结合方向为自右向左，因此它等价于*(p++)，也就是先得到 p 所指向的变量值（*p），再使 p＝p＋1。或者说先取得当前 p 所指向的变量值再使 p 指向下一个变量。

注意：不能对数组名执行++、--操作，比如 a++是不合法的。这是因为 a 是数组名，它是数组的首地址，它的值在程序的运行过程中是固定不变的，是常量。

注意区别：

*p++：先取得当前 p 所指向的变量值再使 p 指向后一个变量，相当于 a[i++]。

*p--：先取得当前 p 所指向的变量值再使 p 指向前一个变量，相当于 a[i--]。

*++p：先使 p 指向后一个变量再取得 p 所指向的变量的值，相当于 a[++i]。

*--p：先使 p 指向前一个变量再取得 p 所指向的变量的值，相当于 a[--i]。

数组元素也可以是指针类型，这种数组称为指针数组。也就是说指针数组的每一个元素都是

我爱自由 5ifreedom.com

指针变量。

指针数组定义的一般形式为：

```
类型名 *数组名[数组长度];
```

例如：

```
int *p[5];
```

由于运算符*的优先级低于运算符[ ]，因此 p 先与[5]结合，形成 p[5]的形式，它显然是一个数组。然后再与*结合，表示数组元素的类型为指针，每个数组元素都指向一个整型变量。这里 p 就是一个二级指针，它是指针的指针。

例 4-10 指针数组和二级指针

```
#include <stdio.h>

int main()
{
    int a[5] = {1,3,5,7,9};
    int *p[5],i;
    int **pp = p;

    for(i=0; i<5; i++)
        p[i] = &a[i];

    for(i=0; i<5; i++)
        printf("%d ",*p[i]);
    printf("\n");

    for(i=0; i<5; i++,pp++)
        printf("%d ",**pp);

    return 0;
}
```

程序运行结果：

```
1 3 5 7 9
```

程序说明

（1）p[5]是一个数组，它的每一个元素都指向一个整型变量。第一个 for 循环把数组 a[5]的各个元素的地址赋给 p[5]，使数组 p[5]的每一个元素都指向 a[5]一个元素。

（2）第二个循环中，对于表达式*p[i]，因为“[ ]”的优先级比“*”高，它等价于*（p[i]）。而 p[i]=&a[i]，所以*p[i]=*&a[i]=a[i]。

（3）p 和 pp 都是二级指针，即指针的指针。对于**pp，当 i=0 时，*pp 就是 p[0]。而 p[0]=&a[0]，那么**pp 就是*&a[0]。*&a[0]=a[0]。所以当 i=0 时，**pp 的值就是 a[0]，即 1。执行 pp++后，pp 指向 p[1]，*pp 就是 p[1]。而 p[1]=&a[1]，那么**pp 就是*&a[1]，其值为 3。图 4-2 显示了各个指针的关系，例 4-11 演示了指针和数组的关系。

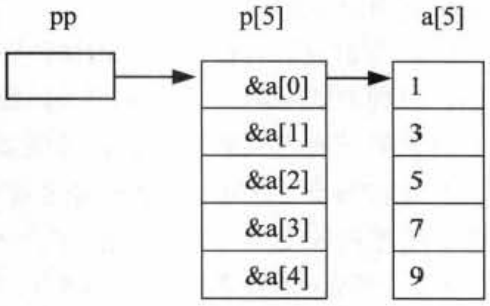


图 4-2 指针、数组、二级指针的关系图

例 4-11

```
#include <stdio.h>

int main()
{

    int a[2][5] = {1,3,5,7,9,2,4,6,8,10};
    int (*p)[5],i;
```

我爱自由 5ifreedom.com

```
    p = a;
    for(i=0; i<5; i++)
        printf("%d ",(*p)[i]);
    printf("\n");

    p++;
    for(i=0; i<5; i++)
        printf("%d ",(*p)[i]);
    printf("\n");

    return 0;
}
```

程序运行结果：

```
1 3 5 7 9
2 4 6 8 10
```

程序说明。

（1）int (*p)[5]，表示 p 是一个指针，它指向含有 5 个元素的一维数组。p 也只能指向一个包含 5 个元素的一维数组，p 就是该一维数组的首地址。这里*p 两边的括号是不可少的，因为[ ]的优先级比*高。

（2）p = a，使得 p 指向二维数组 a 的第一行。而后通过(*p)[i]访问该行的每一个元素。

（3）p++，使 p 指向二维数组 a 的第二行。

注意： 区别 int (*p)[5]和 int *p[5]。前者是一个指针，它指向一个含有 5 个元素的数组。后者是一个数组，它的长度为 5，数组中每一个元素指向一个整型变量。

4.3.4 指针和函数

1. 指针作为函数的参数

函数的参数不仅可以是整型、字符型、实型也可以是指针类型。它的作用是将一个变量的地址传送到一个函数中，实例 4-12 演示了指针作为函数的参数的功能实现。

例 4-12

```
#include <stdio.h>

void change(int i,int *p)
{
    i++;

    if(p != NULL)
        (*p)++;
}

int main()
{

    int a = 5,b = 10;

    change(a,&b);
    printf("a=%d b=%d\n",a,b);
    return 0;
}
```

程序运行结果：

```
a=5 b=11
```

程序说明。

函数 change 接收两个参数，一个为整型，一个为指向整型变量的指针。在函数中，语句(*p)++对 p 所指向的变量执行加 1 操作。在主函数中调用了 change 函数，其中一个实参为变量 a，另一个实参为变量 b 的地址。因为 change 函数的第二个参数为指针，所以必须传递一个变量地址。对于指针型形参，实参也可以是 NULL，因此 change 函数中必须检查 p 是否为 NULL。如果实参为

我爱自由 5ifreedom.com

NULL，那么语句(*p)++将导致程序崩溃。

从程序的运行结果，可以看到主函数中调用 change 函数后变量 b 的值改变了，而变量 a 没有改变。为什么会这样呢？函数的参数是局限于该函数的局部变量。函数调用时，系统为函数的局部变量分配内存。在主函数中调用 change 函数时，系统分配 8 个字节的内存，4 字节用于保存变量 i 的值，另外 4 字节保存指针变量 p 的值。变量 i 保存的是数值 5，而指针变量 p 保存的是变量 b 的地址。change 函数中语句 i++并不作用于变量 a，而仅仅作用于刚刚分配的 4 字节存储空间，此时 i 的值为 6 而 a 的值依然为 5。而对于语句(*p)++，p 保存的是变量 b 的地址，*p 的含义是系统根据该地址找到变量 b，然后对它执行加 1 操作，变量 b 的值就变为 11 了。函数调用结束后，即主函数执行完语句 change(a,&b)，调用函数时分配的内存就被系统回收了。

注意：如果一个函数的参数中有指针，那么出于程序健壮性的考虑，在该函数中须检查参数是否为 NULL。

2. 返回指针的函数

函数可以返回整型值、字符型值、实型值，也可以返回指针，即地址。返回指针的函数的一般定义形式为：

```
类型名  *函数名（参数表）；
```

例如：

```
int *f( int i, int j );
```

调用该函数后，返回一个指向整型变量的指针。()的优先级要高于*，因此 f 先与()结合，这表示 f 是一个函数。函数名前有一个*，表示此函数的返回值类型为指针，例 4-13 演示了返回指针的函数功能实现。

例 4-13

```
#include <stdio.h>

char *name[7] = { "Monday","Tuesday","Wednesday","Thursday",
                  "Friday","Saturday","Sunday" };
char *message = "wrong input";

char *week(int day)
{
        if(day<0 || day>7)
            return message;
        else
            return name[day-1];
}

int main()
{
        int  day;
        char *p;

        printf("Input a number of a week:\n");
        scanf("%d",&day);

        p = week(day);
        printf("%s\n",p);
        return 0;
}
```

程序运行结果：

```
Input a number of a week:
输入 2
Tuesday
```

程序说明。

（1）程序首先定义了一个名为 name 的指针数组，用于保存一周内每天的英文名。数组元素的类型为指向字符串的指针，每个数组元素都指向一个字符串。

我爱自由 5ifreedom.com

（2）函数 week，接收一个整型参数，返回一个指向字符串的指针。如果参数值不再 1～7 之间，就返回一条错误提示信息，否则返回一个数组元素。

（3）主函数中调用了 week 函数，将返回的指针赋给 p，然后打印出 p 所指向的字符串。

3. 指向函数的指针

指向函数的指针是一个指针变量，这个指针比较特殊，它指向一个函数。一个函数的函数名是一个指针，它指向函数的代码。函数的调用可以通过函数名来调用也可以通过指向函数的指针来调用。

指向函数的指针其定义的一般形式为：

```
类型名（*指针变量名）（）；
```

例如：

```
int (*p)( int i, int j );
```

p 是一个指针，它指向的一个函数，这个函数有两个整型参数，返回类型为 int。我们注意到 p 首先和*结合，表明 p 是一个指针。然后再与()结合，表明它指向的是一个函数。指向函数的指针也称为函数指针，例 4-14 实现了指向函数的指针的功能。

例 4-14

```
#include <stdio.h>

#define  GET_MAX    0
#define  GET_MIN    1

int get_max(int i,int j)
{
    return i>j?i:j;
}

int get_min(int i,int j)
{
    return i>j?j:i;
}

int compare(int i,int j,int flag)
{
    int ret;
    int (*p)(int,int);

    if(flag == GET_MAX)
        p = get_max;
    else
        p = get_min;

    ret = p(i,j);

    return ret;
}

int main()
{
    int i = 5,j = 10,ret;

    ret = compare(i,j,GET_MAX);
    printf("The MAX is %d\n",ret);

    return 0 ;
}
```

程序运行结果：

```
The MAX is 10
```

程序说明。

（1）程序首先使用预编译命令 define 定义了两个常量，然后定义了两个函数，分别用来获取

我爱自由 5ifreedom.com

两个数的较大者和较小者。

（2）在 compare 函数中，定义了一个指向函数的指针 p。根据 flag 的值来给该指针赋值。如果 flag 的值为 GET_MAX 即 0，就把函数名 get_max 赋给 p，否则把 get_min 赋给 p。注意语句 p = get_max，只是把函数名赋予给 p。以下赋值方法是错误的：

```
p = get_max(int ,int);
```

之后通过 p 来调用相应的函数，获取较大值或较小值。主函数 main 调用了 compare 函数。

（3）不能对指向函数的指针做任何运算，如 p++、p—、p+n、p–n 都是错误的。

（4）指向函数的指针能通过同类型的函数（即参数相同、返回类型相同）名、函数指针或 NULL 来进行初始化或赋值。将函数指针初始化或赋值为 NULL，表示该指针目前不指向任何函数。

注意：区别：前者是返回指针的函数，它是一个函数的声明。后者是指向函数的指针，它定义了一个指针。

```
int *f( int i, int j );
int (*p)( int i, int j );
```

4．函数指针做形参

我们知道函数的参数可以是指针，那么当然也可以是一个指向函数的指针。以下就是一个含有函数指针参数的函数。

```
int get_max(int i,int j,int k,int (*p)(int,int));
```

该函数接收 4 个参数，第 4 个参数为一个指向函数的指针。在函数内可以通过该指针调用其他函数。在第 8 章介绍信号处理时会使用这种技术。以下例 4-15 演示了函数指针做形参的用法。

例 4-15

```
#include <stdio.h>

int get_big(int i,int j)
{
    return i>j?i:j;
}

int get_max(int i,int j,int k,int (*p)(int,int))
{
    int ret;

    ret = p(i,j);
    ret = p(ret,k);

    return ret;
}

int main()
{
    int i = 5,j = 10,k=15,ret;

    ret = get_max(i,j,k,get_big);
    printf("The MAX is %d\n",ret);

    return 0 ;
}
```

程序运行结果：

```
The MAX is 15
```

程序说明。

函数 get_big 用来获得两个数中的较大者。get_max 函数用来获取 3 个数中的最大者，它的第 4 个形参为一个指向函数的指针。主函数 main 中以函数名 get_big 来作为参数，调用了 get_max。

5．返回函数指针的函数

本部分对于初学者可能比较难以理解，如果不能理解也没有关系，因为本部分所介绍的内容实际项目中用得很少，它一般只出现在知名公司招聘时的笔试面试中，实例 4-16 演示了返回函数

我爱自由 5ifreedom.com

指针的用法。

例 4-16

```
#include <stdio.h>

int get_big(int i,int j)
{
    return i>j?i:j;
}

int (*get_function(int a))(int,int)
{
    printf("the number is %d\n",a);

    return get_big;
}

int main()
{
    int i = 5,j = 10,max;

    int (*p)(int, int);
    p = get_function(100);

    max = p(i,j);
    printf("The MAX is %d\n",max);

    return 0 ;
}
```

程序运行结果：

```
the number is 100
The MAX is 10
```

程序说明。

（1）程序首先定义了一个函数 get_big，用来获取两个数中的较大者。

（2） int (*get_function(int a))(int,int)是一个返回函数指针的函数。这个可能比较难以理解。我们首先抓住 get_function，因为运算符()的优先级比*高，所以 get_function 先与()结合。get_function(int a)是含有一个整型参数的函数。这个函数的返回值比较特殊，它的返回值是 int(*)(int,int)，也就是它返回的是一个指向函数的指针。该指针所指向的函数有两个整型参数。get_function 中，将 get_big 作为函数的返回值。get_big 是一个函数名，也是函数 get_big 的入口地址，它是一个指针。

（3）主函数中，定义了一个指向函数的指针 p，p 所指向的函数有两个整型参数。然后调用 get_function，将返回值赋给 p。函数 get_function 有一个整型形参，调用时以 100 作为实参。函数 get_function 返回一个指向函数的指针。调用 get_function 结束后，p 就指向了 get_big 函数。之后通过 p 调用 get_big 函数，将调用的返回值赋给变量 max。

4.3.5 指向字符串的指针

C 语言中，访问一个字符串有多种方法。

➢ 用字符数组存放一个字符串。

```
char string[] = "Linux C";
printf("%s\n",string);
```

这两行代码输出 Linux C。string 是一个字符数组名，它同时也是该字符数组的首地址，也就是“Linux C”这个字符串的首地址。

➢ 用字符串指针来指向字符串。

可以不用定义数组，而只定义一个指向字符串的指针。指向字符串的指针简称为字符串指针，例如：

我爱自由 5ifreedom.com

```
char *p = "Linux C";
printf("%s\n",p);
```

这两行代码也输出 Linux C。“Linux C”是一个字符串常量。C 语言对于字符串常量通常是这样处理的：在内存中开辟一个字符数组来存储该字符串常量，并把开辟出的字符数组的首地址赋给 p。

值得注意的是，string[0] = ‘A’ 是可以的，而 p[0] = ‘A’ 是非法的，因为 p 指向的是字符串常量，常量的内容不可改变。把 p 指向另外一个字符串常量或字符数组是合法的，例如：

```
p = "Hello World! ";
p = string;
```

例 4-17 拷贝字符串

```
#include <stdio.h>

int main()
{
    char a[] = "Linux C Program",b[20],c[20];
    int  i;

    for(i=0; *(a+i) != '\0'; i++)
        *(b+i) = *(a+i);
    *(b+i) = '\0';

    char *p1,*p2;
    p1 = a;
    p2 = c;

    for(; *p1 != '\0'; p1++,p2++)
        *p2 = *p1;
    *p2 = '\0';

    printf("%s\n",a);
    printf("%s\n",b);
    printf("%s\n",c);

    return 0 ;
}
```

程序运行结果：

```
Linux C Program
Linux C Program
Linux C Program
```

程序说明

（1）a、b、c 都是字符数组，可以通过地址访问数组元素。在第一个 for 循环中，先检查 a[i] 是否为‘\0’。如果不等于‘\0’，表示字符串尚未结束，就将 a[i]的值赋给 b[i]，即复制一个字符。第一个 for 循环结束后，还应将‘\0’复制过去，故有*(b+i) = '\0'。

（2）p1、p2 是指针变量，它们指向字符型数据。由于数组名是指向数组第一个元素的指针，因此可以使 p1、p2 指向数组 a 和 c 的首地址。第二个 for 循环与第一个类似，这里我们注意到第一个 for 循环中，a 和 b 的值没有变化，始终指向数组的第一个元素。而第二个 for 循环中的 p1、p2 的值是变化的，它们不断改变所指向的变量。事实上，不允许对数组名做类似的计算：a++、a--、a+n、a-n。数组名在程序运行过程中始终指向数组的第一个元素。

例 4-18 在函数中实现字符串拷贝。

```
#include <stdio.h>

void copy_string1(char src[ ],char dst[ ])
{
    int i;

    for(i=0; src[i] != '\0';i++)
```

我爱自由 5ifreedom.com

```
        dst[i] = src[i];
    dst[i] = '\0';
}

void copy_string2(char *psrc,char *pdst)
{
    for(; *psrc != '\0'; psrc++,pdst++)
        *pdst = *psrc;
    *pdst = '\0';
}

int main()
{
    char a[] = "Linux C Program",b[20],c[20];

    copy_string1(a,b);
    copy_string2(a,c);

    printf("%s\n%s\n%s\n",a,b,c);

    return 0 ;
}
```

程序运行结果：

```
Linux C Program
Linux C Program
Linux C Program
```

程序说明。

（1）copy_string1 函数接收两个数组作为参数，而 copy_string2 接收两个指向字符串的指针作为参数。这两种方式是完全等价的。copy_string1 的两个参数从形式上看是两个数组，实际上它是：

```
void copy_string1(char *src,char *dst)
```

因此在 copy_string1 函数中，src 和 dst 的值是可变的，而不是像一般的数组那样只是指向一个固定的位置。以下对 copy_string1 函数的实现也是可行的：

```
void copy_string1(char src[],char dst[])
{
    for(; *src != '\0'; src++,dst++)
        *dst = *src;
    *dst = '\0';
}
```

这个实现方法中，dst 和 src 不断在改变它们所指向的变量。

（2）我们再来看看主函数中的语句 copy_string1（a,b）和 copy_string2（a,c）是如何传递参数的。对于 copy_string1（a,b），把 a 和 b 的值（该值是指向数组第一个元素的指针，而这个指针就是存放数组第一个元素的内存地址）赋值给 src 和 dst。src 和 dst 保存的是 a 和 b 的首地址。copy_string2（a,c）也是一样，psrc 和 pdst 也都保存着数组 a 和 c 的第一个元素的地址。

对于 copy_string1（a,b），如图 4-3 所示：a 的值为 0xbfff1010（以 0x 开头表示该数以 16 进制表示），它是 a[0]即字符 L 内存地址，src 的值是 a 的值即 0xbfff1010。因此通过 src 可以很方便地对数组 a 进行操作。copy_string1（a,b）中的实参 b 和形参 dst 也是如此。

注意：数组，如 char a[20]，a 是指向数组第一个元素的指针，它的值不可以被改变，它在程序运行过程中始终指向数组的第一个元素。而在函数定义中，如 void copy_string1(char src[ ],char dst[ ])，src 也是一个指针，但它的值是可以改变的，也就是说，它可以指向其他字符变量。

字符数组由若干元素组成，每个元素存放一个字符。而字符串指针中存放的是地址（字符串的首地址），而不是将字符串存放到字符串指针中。

我爱自由 5ifreedom.com

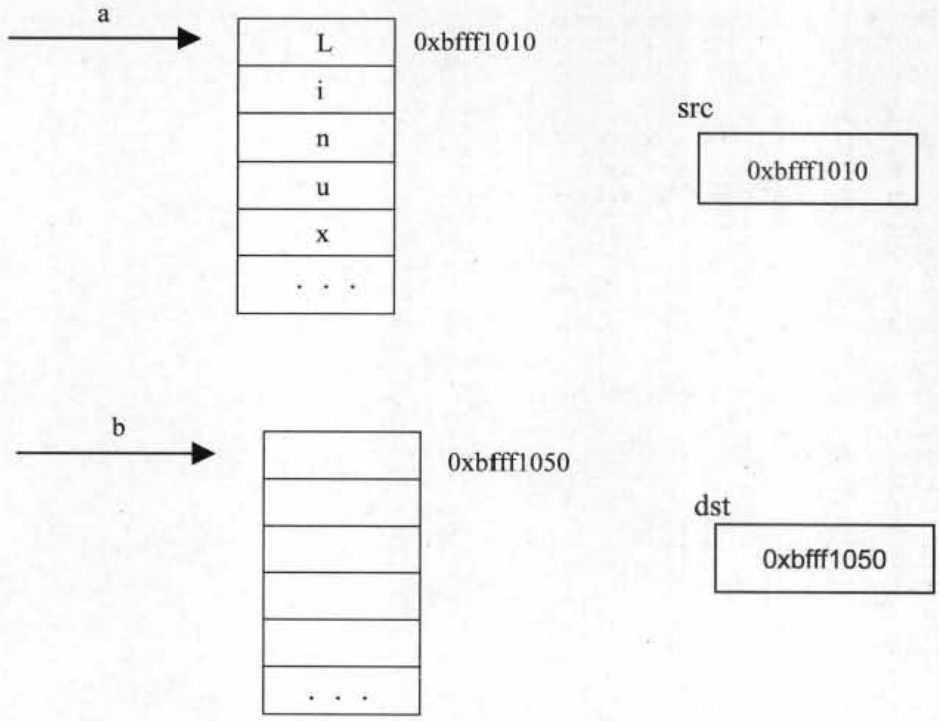


图 4-3　字符串拷贝

4.4 字符串函数

C 语言中提供了许多用来处理字符串的函数，使用这些函数可以大大减轻我们的工作量。

4.4.1　puts 和 gets

1. puts 函数

该函数将一个以‘\0’结尾的字符串输出到屏幕上，例如：

```
char a[]    =  "Welcome to ";
char *p     =  "Linux C Program";
puts(a);
puts(p);
```

将输出：

```
Welcome to
Linux C Program
```

2. gets 函数

从终端输入一个字符串到字符数组中，它的返回值是该字符数组的首地址，例如：

```
char string[20];
gets(string);
从键盘输入 computer
puts(string)
```

则输出：

```
computer
```

4.4.2　strcpy 和 strncpy

```
#include <string.h>
char *stpcpy(char *dest,char *src);
char *strncpy(char *dest, char *src, int n);
```

我爱自由 5ifreedom.com

说明。

（1）strcpy 是 string copy 的缩写。使用这两个函数时须包含头文件 string.h。这两个函数的返回值都是参数 dest。

（2）strcpy 把 src 所指向的以‘\0’结尾的字符串复制到 dest 所指的数组中。strncpy 把 src 所指向的以‘\0’结尾的字符串的前 n 个字节复制到 dest 所指的数组中。

（3）dest 所指向的数组必须足够大，以便容纳被复制的 src 所指向的字符串。复制时连同字符串的‘\0’一起被复制。

（4）以下代码是错误的：

```
char a[ ] = "Linux C Program",b[20];
b = a;
```

不能将一个字符串常量或字符数组直接赋给另一个字符数组。字符串的复制只能使用 strcpy、strncpy 或者其他类似功能的函数。

（5）strcpy 是不安全的，存在安全漏洞，容易被黑客所利用。因此一般应该使用 strncpy。

示例代码：

```
char *s="Hello World";
char d1[20], d2[20];
strcpy (d1, s);
strncpy (d2, s, sizeof (s));
```

sizeof（s）获得 s 所指向的字符串的长度，这里 sizeof（s）等于 12，因为字符串“Hello World”最后的‘\0’也是一个字符。

4.4.3 strcat 和 strncat

```
#include <string.h>
char *strcat(char *dest,char *src);
char *strncat(char *dest,char *src,int n);
```

说明。

（1）strcat 是 string catenate（字符串连接）的缩写。这两个函数所在的头文件都是 string.h，都以参数 dest 的值作为返回值。

（2）strcat 把 src 所指向的字符串添加到 dest 结尾处（覆盖 dest 结尾处的‘\0’）并添加‘\0’。strncat 把 src 所指向的字符串的前 n 个字符添加到 dest 结尾处（覆盖 dest 结尾处的‘\0’）并添加‘\0’。

（3）dest 所指向的数组必须足够大，以便容纳连接后的新字符串。

（4）strcat 是不安全的，存在安全漏洞，容易被黑客所利用。因此一般应该使用 strncat。

示例代码：

```
char d[20]="Hello ";
char *s="World";
strcat(d,s);
```

4.4.4 strcmp 和 strncmp

```
#include <string.h>
int strcmp(char *s1,char * s2);
int strncmp(char *s1,char * s2, int n);
```

说明。

（1）strcmp 是 string compare（字符串比较）的缩写。这两个函数所在的头文件都是 string.h，其功能是比较两个字符串。

（2）strncmp 只对两个字符串的前 n 个字符进行比较。字符串的比较规则是：从左到右逐个字符进行比较，直到出现不同的字符或遇到‘\0’为止。如果全部的字符相同且长度一样，则认为两个字

我爱自由 5ifreedom.com

符串相等，返回值为 0。如果出现不相同的字符，则对第一次出现的两个不相同的字符进行比较。比较方法是以 s1 的第一个不相同的字符减去 s2 的第一个不相同的字，以所得的差值作为返回值，例如：

strcmp（“aab”，“aaB”），第一次不相同的字符为‘b’和‘B’，‘b’的 ASCII 码为 98，‘B’的 ASCII 码为 66，因此“aab”> “aaB”，返回值为一正数，通常为 1。“compare”< “computer”，因为‘a’<‘u’，返回值为一负数，通常为-1。

示例代码：

```
char *s1 = "compare";
char *s2 = "computer";
if(strcmp(s1,s2) == 0)
    printf("compare = computer");
else if(strcmp(s1,s2) > 0)
    printf("compare > computer");
else
    printf("compare < computer");
```

4.4.5 strlen

```
#include <string.h>
int strlen(char *s);
```

说明。

（1）strlen 是 string length 的缩写。它所在的头文件是 string.h，功能是返回字符串的长度。

（2）strlen 返回的是字符串实际长度，不包括‘\0’在内，例如：

```
char string[10] = "China";
printf("%d\n",strlen(string) );
```

输出的结果不是字符数组长度 10，也是不是 6，而是 5。

4.4.6 strlwr 和 strupr

```
#include <string.h>
char *strlwr(char *s);
char *strupr(char *s);
```

说明。

（1）strlwr 是 string lower 的缩写，strupr 是 string upper 的缩写。使用它们前必须包含头文件 string.h。

（2）strlwr 将 s 所指向的字符串中所有的大写字母都转换为小写字母。strupr 将 s 所指向的字符串中所有的小写字母都转换为大写字母。它们的返回值都是转换后的字符串的指针。

示例代码：

```
char string[10] = "China";
printf("%s\n",strlwr(string));
printf("%s\n",strupr(string));
```

输出：

```
china
CHINA
```

4.4.7 strstr 和 strchr

```
#include <string.h>
char *strstr(char *s1, char *s2);
char *strchr(char *s,char c);
```

说明。

（1）strstr 是 string string 的缩写，strchr 是 string char 的缩写。使用它们前必须包含头文件 string.h。

（2） strstr 从字符串 s1 中寻找 s2 第一次出现的位置，返回指向第一次出现 s2 位置的指针，如果没找到则返回 NULL。strchr 查找字符串 s 中首次出现字符 c 的位置，返回首次出现字符 c 的

我爱自由 5ifreedom.com

指针，如果 s 中不存在 c 则返回 NULL。

示例代码：

```
char *s1="Linux C Program",*s2="nux",*p;

p=strstr(s1,s2);
if(p != NULL)
   printf("%s\n",p);
else
   printf("Not Found!");

p = strchr(s1,'C');
if(p != NULL)
   printf("%s\n",p);
else
   printf("Not Found!");
```

输出：

```
nux C Program
C Program
```

4.5 调试器 gdb

在 Linux 应用程序开发中，最常用的调试器是 gdb。gdb 采用 GPL 授权条款，是 GNU 的计划之一，所以任何人都可以免费得到和使用它。在安装 Linux 操作系统时，如果选择安装 gdb，gdb 就会被自动安装。gdb 和其他调试器一样，可以在程序中设置断点、查看变量值、一步一步地跟踪程序的执行过程。利用调试器的这些功能可以方便地找出程序中存在的非语法错误。

4.5.1 启动和退出 gdb

gdb 调试的对象是可执行文件，而不是程序的源代码。如果要使一个可执行文件可以被 gdb 调试，那么在使用编译器 gcc 编译程序时需要加入-g 选项。-g 选项告诉 gcc 在编译程序时加入调试信息，这样 gdb 才可以调试这个被编译的程序。

首先编写一个用于调试的测试程序 test.c。这个程序有一个名为 get_sum 的函数，它用来求 1 到 n 的和。为了方便查看，我们对该程序各行代码进行了编号，代码如下所示。

```
1    #include<stdio.h>
2
3    int get_sum(int n)
4    {
5        int sum = 0,i;
6        for(i=0;i<n;i++)
7        sum += i;
8       return sum;
9    }
10
11    int main()
12    {
13        int i = 100,result;
14        result = get_sum(i);
15        printf("1+2+...+%d=%d\n",i,result);
16        return 0;
17    }
```

编译并运行该程序：

```
$ gcc -g test.c -o test
$ ./test
1+2+...+100=4950
```

我爱自由 5ifreedom.com

程序输出 4950，我们的本意是求 1～100 的和，应该输出 5050。程序虽然没有语法错误，但显然存在逻辑上的错误。

gdb 调试一个程序的命令格式是：

```
gdb 程序文件名
```

示例：

```
$ gdb test
GNU gdb Red Hat Linux (6.3.0.0-1.63rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, 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 "x86_64-redhat-linux-gnu"...Using host libthread_db library
"/lib64/tls/libthread_db.so.1".

(gdb)
```

启动 gdb 后，首先显示了一段版权说明，然后是 gdb 的提示符：(gdb)。可以在（gdb）之后输入调试命令。

注意：如果要使 gdb 启动时不输出版权说明，可以在执行时加上-q 选项，如：gdb –q test。也可以在 Linux 提示符下，直接输入 gdb，然后使用 file 命令装入要调试的程序。

如：

```
$ gdb -q
(gdb) file test
Reading symbols from /home/tyq/test...done.
Using host libthread_db library "/lib64/tls/libthread_db.so.1".
(gdb)
```

如果要结束调试，使用 quit 命令就可以退出 gdb，返回到 Linux 的提示符。

示例：

```
(gdb)quit
$
```

有时输入 quit 命令时会出现下面的提示信息，表示调试器调试的程序正在运行过程中，是否强行退出，输入 y，gdb 终止被调试的程序并结束 gdb。

示例：

```
(gdb)quit
The program is runing.Exit anyway?(y or n)  y
$
```

4.5.2 显示和查找程序源代码

在调试程序时，一般要查看程序的源代码。list 命令用于列出程序的源代码，它的使用格式如下。

- **list**：显示 10 行代码，若再次运行该命令则显示接下来的 10 行代码。
- **list 5，10**：显示第 5 行到第 10 行的代码。
- **list test.c:5，10**：显示源文件 test.c 中的第 5 行到第 10 行的代码，在调试含有多个。源文件的程序时使用。
- **list get_sum**：显示 get_sum 函数周围的代码。
- **list test.c:get_sum**：显示源文件 test.c 中 get_sum 函数周围的代码，在调试含有多个。源文件的程序时使用。

示例：

```
(gdb) list 4,9
4       {
```


我爱自由 5ifreedom.com

```
5           int sum = 0,i;
6           for(i=0;i<n;i++)
7               sum += i;
8           return sum;
9       }
(gdb) list get_sum
1       #include<stdio.h>
2
3       int get_sum(int n)
4       {
5           int sum = 0,i;
6           for(i=0;i<n;i++)
7               sum += i;
8           return sum;
9       }
10
(gdb)
```

注意：如果在调试过程中要运行 Linux 命令，则可以在 gdb 的提示符下输入 shell 命令。例如：

```
(gdb)shell ls
```

search 和 forward 两个命令都用来从当前行向后查找第一个匹配的字符串，格式如下：

```
search    字符串
forward 字符串
```

reverse-search 用来从当前行向前查找第一个匹配的字符串，格式如下：

```
reverse-search  字符串
```

示例：

```
(gdb) search get_sum
14          result = get_sum(i);
(gdb) reverse-search main
11      int main()
(gdb)
```

4.5.3 执行程序和获得帮助

使用 gdb –q test 或 file test 只是装入程序，程序并没有运行。如果要使程序开始运行，在 gdb 提示符下输入 run 即可。

示例：

```
(gdb) run
Starting program: /home/tyq/test
1+2+...+100=4950

Program exited normally.
(gdb)
```

注意：如果想要详细了解 gdb 某个命令的使用方法，可以使用 help 命令。

例如：

```
(gdb)help list
(gdb)help all
```

前一个命令列出 list 命令的帮助信息，后一个命令列出所有 gdb 命令的帮助信息。

4.5.4 设置和管理断点

在调试程序时，往往需要程序在运行到某行、某个函数或某个条件发生时暂停下来，然后查看此时程序的状态，如各个变量的值、某个表达式的值等。为此，我们可以设置断点。断点使程序运行到某个位置时暂停下来，以便检查和分析程序。断点在调试程序时非常有用，因此学习设置和管理断点是非常必要的。

1. 以行号设置断点

在 gdb 中，大部分都是使用 break 命令为程序设置断点。而指定断点时，最常用的是为某行设置断点。例如：

我爱自由 5ifreedom.com

```
(gdb) break 7
Breakpoint 1 at 0x4004c5: file test.c, line 7.
```

其中第二行是设置断点后的反馈信息。1 表示当前设置的是第 1 个断点，0x4004c5 是断点所在的内存地址，file test.c, line 7 表明断点设置在 test.c 文件的第 7 行处。

然后我们输入 run 命令运行程序：

```
(gdb) run
Starting program: /home/tyq/test
Breakpoint 1, get_sum (n=100) at test.c:7
7             sum += i;
```

可以看到，程序运行完第 6 行的指令后就暂停了，第 7 行的代码没有执行而是被 gdb 的断点中断了。此时，我们可以查看各个变量和表达式的值，以了解程序当前的状况。也可以视情况，让程序一步一步地执行或直接执行到程序完毕。这些用法在后面会加以介绍。

2. 以函数名设置断点

在 break 命令后跟上函数名，就可以为函数设置断点。如：

```
(gdb) break get_sum
Breakpoint 1 at 0x4004af: file test.c, line 5.
(gdb) run
Starting program: /home/tyq/test

Breakpoint 1, get_sum (n=100) at test.c:5
5         int sum = 0,i;
```

可行看到程序在第 5 行停了下来。上述过程是重新执行 gdb 的结果，我们可以看到这是为 test.c 设置的第 1 个断点。后面设置断点也是如此。

3. 以条件表达式设置断点

break 还可以用来设置这样的断点：程序在运行过程中，当某个条件满足时，程序在某行中断暂停执行。命令格式是：

```
break 行号或函数名 if 条件
```

如：

```
(gdb) list 1,17
1       #include<stdio.h>
2
3       int get_sum(int n)
4       {
5           int sum = 0,i;
6           for(i=0;i<n;i++)
7               sum += i;
8           return sum;
9       }
10
11      int main()
12      {
13          int i = 100,result;
14          result = get_sum(i);
15          printf("1+2+...+%d=%d\n",i,result);
16          return 0;
17      }
(gdb) break 7 if i==99
Breakpoint 1 at 0x4004c5: file test.c, line 7.
(gdb) run
Starting program: /home/tyq/test

Breakpoint 1, get_sum (n=100) at test.c:7
7     sum += i;
```

可以看到，运行程序后在 i=99 时，程序中断在第 7 行。

还有一种以表达式设置断点的方法，这种设置方式不需要指定行号或函数名，而是在整个程序运行中当条件表达式的值发生改变时程序就会暂停下来。命令格式是：

我爱自由 5ifreedom.com

```
watch 条件表达式
```

例如：

```
$ gdb -q test
Using host libthread_db library "/lib64/tls/libthread_db.so.1".
(gdb) watch i==99
No symbol "i" in current context.
(gdb) break 6
Breakpoint 1 at 0x4004b6: file test.c, line 6.
(gdb) run
Starting program: /home/tyq/test

Breakpoint 1, get_sum (n=100) at test.c:6

6           for(i=0;i<n;i++)
(gdb) watch i==99
Hardware watchpoint 2: i == 99
(gdb) clear 6
Deleted breakpoint 1
(gdb) continue
Continuing.
Hardware watchpoint 2: i == 99

Old value = 0
New value = 1
0x004004d4 in get_sum (n=100) at test.c:6
6           for(i=0;i<n;i++)
(gdb) print i
$1 = 99
(gdb) print sum
$2 = 4851
(gdb)
```

gdb 运行后，以命令 watch i==99 设置条件断点，但是失败了。gdb 提示在当前程序的上下文中没有符号 i。这是因为此时 test 程序没有运行，变量 i 还没有被定义。为了解决这个问题，首先在第 6 行设置中断。然后使用 run 命令运行程序，程序暂停在第 6 行，此时第 5 行的语句已经被执行，所以变量 i 已经定义。这时就可以使用 watch i==99 设置断点了。因为第 6 行的断点已经没有用了，于是用 clear 6 命令删除该行的断点。

之后使用 continue 命令让程序继续运行。看到程序在第 6 行停了下来，此时刚刚执行完 i++ 这个语句，i 的值变为 99。原先 i 的值不等于 99，所以 i==99 这个表达式的值为 0。当 i 的值变为 99 后，这个条件表达式的值也变为 1。表达式的值改变了，所以程序在这是中断。使用 print i 和 print sum 命令打印出此时变量 i 和 sum 的值。

继续调试该程序：

```
(gdb) next
7               sum += i;
(gdb) print i
$3 = 99
(gdb) print sum
$4 = 4851
(gdb) next
6           for(i=0;i<n;i++)
(gdb) print i
$5 = 99
(gdb) print sum
$6 = 4950
(gdb) next
Hardware watchpoint 2: i == 99

Old value = 1
New value = 0
0x004004d4 in get_sum (n=100) at test.c:6
6           for(i=0;i<n;i++)
```

我爱自由 5ifreedom.com

```
(gdb) print i
$7 = 100
(gdb) print sum
$8 = 4950
(gdb) next
8          return sum;
(gdb) print i
$7 = 100
(gdb) print sum
$8 = 4950
(gdb)
```

next 命令用于继续执行下一条语句。输入 next，程序执行判断语句 i<n，i 此时的值为 99，而 n 的值为 100。条件成立，因此 gdb 输出“7 sum += i;”表明下一条要执行的语句为 sum += i。之后再输入 next，此时 sum += i 执行完，下一条要执行的语句是第 6 行的 i++。打印出 i 和 sum 的值，可以看到分别为 99 和 4950。

继续输入 next 命令，打印出此时的 i 和 sum 的值，分别为 100 和 4950。输入 next 命令，此时程序刚刚执行完判断语句 i<n，原本下一条应该执行的语句是 sum += i，也就是把此时的 i 值即 100 加到 sum 上去。但程序提示下一条要执行的语句是第 8 行的 return sum。也就是说最后的 100 没有加到总和上去，函数 get_sum 就返回了。

因此可以断定是 i<n 这个判断语句有问题，它使得最后的 100 没有加到总和上去。把 i<n 改为 i<=n 就可以了。

对于这种简单的程序，如果仔细检查源程序，就可以发现问题所在，不需要 gdb 调试器。但对于较复杂的程序，就必须要使用调试器一步一步地跟踪程序的运行以发现程序的逻辑错误。

还有一个与 watch 类似的命令：awatch。它也用来给表达式设置断点，在表达式的值发生改变或表达式的值被读取的时候，程序暂停执行。

4. 查看当前设置的中断点

使用 info breakpoints 命令可以查看当前所有的中断点。如：

```
(gdb) break 7
Breakpoint 1 at 0x4004c5: file test.c, line 7.
(gdb) break 15 if result==5050
Breakpoint 2 at 0x4004f5: file test.c, line 15.
(gdb) info breakpoints
Num Type           Disp Enb  Address        What
1   breakpoint     keep y    0x004004c5     in get_sum at test.c:7
2   breakpoint     keep y    0x004004f5     in main at test.c:15
        stop only if result == 5050
```

Num 列表示断点的编号。Type 指明类型，类型为 breakpoints 说明是中断。Disp 指示中断点在生效一次后是否就失去作用，如果是则为 dis，不是则为 keep。Enb 表明当前中断点是否有效，如果是为 y，不是则为 n。Address 列表示中断所处的内存地址。What 列出中断发生在哪个函数的第几行。“stop only if result = = 5050”表明这是一个条件中断。

5. 使中断失效或有效

使用“disable 断点编号”命令可以使某个断点失效，程序运行到该断点时不会停下来而是继续运行。使用“enable 断点编号”命令可以使某个断点恢复有效。

```
(gdb) info breakpoints
Num Type           Disp Enb  Address        What
1   breakpoint     keep y    0x004004c5     in get_sum at test.c:7
2   breakpoint     keep y    0x004004f5     in main at test.c:15
        stop only if result == 5050
(gdb) disable 2
(gdb) info breakpoints
Num Type           Disp Enb  Address        What
```

我爱自由 5ifreedom.com

```
1   breakpoint      keep y   0x004004c5        in get_sum at test.c:7
2   breakpoint      keep n   0x004004f5        in main at test.c:15
        stop only if result == 5050
(gdb) enable 2
(gdb) info breakpoints
Num Type            Disp Enb  Address          What
1   breakpoint      keep y   0x004004c5        in get_sum at test.c:7
2   breakpoint      keep y   0x004004f5        in main at test.c:15
        stop only if result == 5050
```

看到第二次运行 info breakpoints 时，第 2 个中断的 Enb 为 n，表明此时第 2 个断点已经失效了。之后使用了“enable 2”恢复了断点 2。

6. 删除断点

disable 只是让某个断点暂时失效，断点依然存在于程序中。如果要彻底删除某个断点，可以使用 clear 或 delete 命令。命令格式如下所示。

- clear：删除程序中所有的断点。
- clear 行号：删除此行的断点。
- clear 函数名：删除该函数的断点。
- delete 断点编号：删除指定编号的断点。如果一次要删除多个断点，各个断点编号以空格隔开。

示例：

```
(gdb) break 6
Breakpoint 1 at 0x4004b6: file test.c, line 6.
(gdb) break 7
Breakpoint 2 at 0x4004c5: file test.c, line 7.
(gdb) break 8 if sum==5050
Breakpoint 3 at 0x4004d6: file test.c, line 8.
(gdb) info breakpoints
Num Type           Disp Enb Address           What
1   breakpoint     keep y   0x00000000004004b6 in get_sum at test.c:6
2   breakpoint     keep y   0x00000000004004c5 in get_sum at test.c:7
3   breakpoint     keep y   0x00000000004004d6 in get_sum at test.c:8
        stop only if sum == 5050
(gdb) clear 6
Deleted breakpoint 1
(gdb) info breakpoints
Num Type           Disp Enb Address           What
2   breakpoint     keep y   0x00000000004004c5 in get_sum at test.c:7
3   breakpoint     keep y   0x00000000004004d6 in get_sum at test.c:8
        stop only if sum == 5050
(gdb) delete 2 3
(gdb) info breakpoints
No breakpoints or watchpoints.
(gdb)
```

4.5.5 查看和设置变量的值

当程序执行到中断点暂停执行时，往往要查看变量或表达式的值，借此了解程序的执行状态，进而发现问题所在。

1. print 命令

print 命令一般用来打印变量或表达式的值，也可以用来打印内存中从某个变量开始的一段内存区域的内容，还可以用来对某个变量进行赋值。其使用格式为：

```
print 变量或表达式：打印变量或表达式当前的值；
print 变量=值：对变量进行赋值。
print 表达式@要打印的值的个数 n 打印以表达式值开始的 n 个数。
```

示例：

```
$ gdb -q test
Using host libthread_db library "/lib64/tls/libthread_db.so.1".
```

我爱自由 5ifreedom.com

```
(gdb) break 7
Breakpoint 1 at 0x4004c5: file test.c, line 7.
(gdb) run
Starting program: /home/tyq/test

Breakpoint 1, get_sum (n=100) at test.c:7
7             sum += i;
(gdb) print i<n
$1 = 1
(gdb) print i
$2 = 0
(gdb) print sum
$3 = 0
(gdb) print i = 200
$4 = 200
(gdb) continue
Continuing.
1+2+...+100=200

Program exited normally.
(gdb)
```

在 test.c 程序的第 7 行设置一个断点，然后使用 run 命令开始运行程序。在执行完第 6 行语句后，程序暂停下来，并提示下一条要执行的语句是第 7 行的 sum += i。第 6 行实际执行两条语句，一条是 i=0，另一条是判断语句 i<n。

打印出 i<n 的值，显然这个表达式为真，因此值为 1，i 和 sum 的值都为 0。然后使用 print i=200 这个语句将变量 i 赋值为 200，并使用 continue 语句让程序继续执行。程序打印出 1+2+...+100=200，并结束。

为何最后的运算结果为 200，而不是原先运行的 4950 呢？因为在将 i 的值赋值为 200 后，程序执行 sum += i，sum 的值变为 200。然后程序执行 i++，i 的值变 201。显然此时 i<n 这个表达式为假，那么也就不会再去执行 sum += i 语句了，所以不会在这个点中断，而是直接执行 return sum。

2. whatis 命令

whatis 命令用来显示某个变量或表达式值的数据类型。格式如下：

```
whatis 变量或表达式。
```

示例：

```
$ gdb -q test
Using host libthread_db library "/lib64/tls/libthread_db.so.1".
(gdb) break 7
Breakpoint 1 at 0x4004c5: file test.c, line 7.
(gdb) run
Starting program: /home/tyq/test

Breakpoint 1, get_sum (n=100) at test.c:7
7             sum += i;
(gdb) whatis i
type = int
(gdb) whatis sum+0.5
type = double
(gdb)
```

可以看到，程序运行后，变量 i 的数据类型为 int，表达式 sum+0.5 的数据类型为 double。

3. set 命令

set 命令可以用来给变量赋值，使用格式是：

```
set variable 变量=值
```

将上面示例中的 print i=200 改为 set variable i=200 可以得到同样的效果。

除了这个用法外，set 命令还有一些其他用法，比如可以针对远程调试进行设置，可以用来设置 gdb 一行的字符数等。

我爱自由 5ifreedom.com

4.5.6 控制程序的执行

当程序执行到指定的中断点，查看了变量或表达式的值后，可以让程序继续运行。可以让程序一步一步地执行，也可以让程序一直运行下去直到下一个断点或运行完为止。

1. continue 命令

让程序继续运行，直到下一个断点或运行完为止。该命令的格式是：

```
continue
```

前面已经多次使用过这个命令，这里就不单独举例了。

2. kill 命令

该命令用于结束当前程序的调试，在 gdb 提示符下输入 kill，gdb 会询问是否退出当前程序的调试，输入 y 结束调试，输入 n 继续调试程序。

```
(gdb) kill
Kill the program being debugged? (y or n) y
(gdb)
```

3. next 和 step 命令

调试程序时，有时会怀疑程序的错误可能出现在某个地方，那么可以使用 next 或 step 命令一次一条执行该段代码。next 与 step 命令的区别是：如果遇到函数调用，next 会把该函数调用当作一条语句来执行，再次输入 next 会执行函数调用后的语句；而 step 则会跟踪进入函数，一次一条地执行函数内的代码，直到函数内的代码执行完，才执行函数调用后的语句。

还是以 test 程序的运行为例：

```
$ gdb -q test
Using host libthread_db library "/lib64/tls/libthread_db.so.1".
(gdb) list 1,17
1       #include<stdio.h>
2
3       int get_sum(int n)
4       {
5           int sum = 0,i;
6           for(i=0;i<n;i++)
7               sum += i;
8           return sum;
9       }
10
11      int main()
12      {
13          int i = 100,result;
14          result = get_sum(i);
15          printf("1+2+...+%d=%d\n",i,result);
16          return 0;
17      }
(gdb) break 13
Breakpoint 1 at 0x4004e3: file test.c, line 13.
(gdb) run
Starting program: /home/tyq/test

Breakpoint 1, main () at test.c:13
13          int i = 100,result;
(gdb) next
14          result = get_sum(i);
(gdb) next
15          printf("1+2+...+%d=%d\n",i,result);
(gdb) next
1+2+...+100=4950
16          return 0;
(gdb)
```

首先在第 13 行设置了一个断点，然后运行程序。程序执行完第 12 行的代码后暂停。输入 next

我爱自由 5ifreedom.com

命令单步执行程序，程序执行第 13 行的代码，执行完后又暂停并提示下一条要执行的语句是 result = get_sum(i)。再输入 next，gdb 提示下一条要执行的语句是第 15 行的 printf，显然程序没有跟踪进入 get_sum 函数，而是把这个函数调用当作一条普通的语句来执行。

下面再来看 step：

```
$ gdb -q test
Using host libthread_db library "/lib64/tls/libthread_db.so.1".
(gdb) break 13
Breakpoint 1 at 0x4004e3: file test.c, line 13.
(gdb) run
Starting program: /home/tyq/test

Breakpoint 1, main () at test.c:13
13          int i = 100,result;
(gdb) step
14          result = get_sum(i);
(gdb) step
get_sum (n=100) at test.c:5
5           int sum = 0,i;
(gdb) step
6           for(i=0;i<n;i++)
(gdb)
```

同样地，在第 13 行设置了断点。运行 run 命令，并执行第一个 step 命令。第一个 step 命令使程序执行第 13 行的代码，并提示下一个要执行的是第 14 行的 result = get_sum(i)。再次输入 step，可以看到下一条要执行的语句是第 5 行的 int sum = 0,i，显然 step 命令跟踪并进入了 get_sum 函数。

4. nexti 和 stepi 命令

nexti 和 stepi 命令用来单步执行一条机器指令，注意不是单步执行一行语句。单步执行一行语句的是 next 和 step 命令。通常一条语句由多条机器指令构成。

对于 test.c 的第 6 行语句：

```
for(i=0;i<n;i++)
```

如果是单步执行一条指令，则这行语句要输入多个 nexti 或 stepi 才能执行完， i=0 和 i<n 会分开执行。而对于单步执行一行语句来说，只需输入一个 next 或 step 就可以执行完，即一次把 i=0 和 i<n 都执行了。

nexti 和 next 类似，不会跟踪进入函数内部去执行。而 stepi 和 step 类似，跟踪进入函数执行。

示例：

```
$ gdb -q test
Using host libthread_db library "/lib64/tls/libthread_db.so.1".
(gdb) break 6
Breakpoint 1 at 0x4004b6: file test.c, line 6.
(gdb) run
Starting program: /home/tyq/test

Breakpoint 1, get_sum (n=100) at test.c:6
6           for(i=0;i<n;i++)
(gdb) stepi
0x004004bd      6           for(i=0;i<n;i++)
(gdb) stepi
0x004004bd      6           for(i=0;i<n;i++)
(gdb) stepi
0x004004bd      6           for(i=0;i<n;i++)
(gdb) stepi
7           sum += i;
(gdb)
```

在使用 run 命令运行程序后，程序在执行完第 5 行后暂停，并提示下一条要执行的是第 6 行语句。输入 stepi 开始执行第 6 行的语句，gdb 提示下一条要执行的还是第 6 行的语句。连续输入了 4 个 stepi

我爱自由 5ifreedom.com

才把 for（i=0;i<n;i++）这行语句执行完，gdb 才提示下一条要执行的是 sum += i。

注意：gdb 的一些命令可以简写，比如 list 命令可以用 li 来代替，continue 命令可以用 cont 来代替。

4.6 面试题选

函数、数组、指针和字符串操作是 C 语言的重点，也是难点，因此这方面的笔试面试题比较多。有些试题是有相当难度的，读者可能一时无法完全理解。学习是一个循序渐进的过程，随着对 C 语言的接触日益增多，慢慢地就会完全理解。

例 4-19 请解释关键字 static，并说明至少两种 static 的用途。

静态变量（以 static 作为修饰符的变量）分为两种：全局静态变量和局部静态变量。全局静态变量是在所有函数之外定义的静态变量，局部静态变量是在某个函数内（如 main 函数）定义的变量。静态变量存储在内存的静态存储区，静态存储区在程序的整个运行期间都存在。未经初始化的静态变量会被程序自动初始化为 0（自动对象的值是任意的，除非被显示初始化）。全局静态变量的作用域是从定义之处开始到文件结尾，全局静态变量对其他文件是不可见的。而局部静态变量只在定义它的函数内有效。

static 的用途如下所示。

（1）限制变量的作用域。

（2）设置变量的存储域。

例 4-20 请分析程序是否正确，如果错误指出原因，如果正确写出程序运行结果并说明原因。

```
#include<stdio.h>
int a = 3;
void f()
{
    int a = 1,i;
    printf("%d\n",a);

    for(i=0;i<1;i++)
    {
        int a = 2;
        printf("%d\n",a);
    }
}
int main()
{
    f();
    printf("%d\n",a);
}
```

程序分析。

本题主要测试对全局变量与局部变量的理解。这个程序是完全正确的，运行结果是：

```
1
2
3
```

初值为 3 的变量 a 是全局变量，因为它在所有函数之外定义。初值为 1 的变量 a 是局部变量，在 f 函数内部自动屏蔽了全局变量 a，因此第一个 printf 语句输出的是 1。for 循环内的变量 a 的定义是合法的，它只在 for 循环体内有效，在 for 循环体内自动屏蔽了前面两个同名的变量，因此第二个 printf 语句输出的是 2。在 main 函数内的 printf 输出的是全局变量 a 的值，因为 f 函数内两个局部变量 a 只在 f 函数内部有效，而全局变量在该源文件内都有效。

我爱自由 5ifreedom.com

例 4-21 static 全局变量与普通的全局变量有什么区别？static 局部变量和普通局部变量有什么区别？static 函数与普通函数有什么区别？

在定义变量时，全局变量之前再冠以 static 就构成了静态的全局变量。全局变量本身就是静态存储方式，静态全局变量当然也是静态存储方式。两者在存储方式上并无不同。两者的区别在于非静态全局变量的作用域是整个源程序，当一个源程序由多个源文件组成时，非静态的全局变量在各个源文件中都是有效的。而静态全局变量则限制了其作用域，即只在定义该变量的源文件内有效，在同一源程序的其他源文件中不能使用它。由于静态全局变量的作用域局限于一个源文件内，只能为该源文件内的函数使用，因此可以避免其他源文件使用该变量。把普通全局变量改变为静态全局变量是改变了它的作用域，限制了它的使用范围。

普通局部变量所在的函数每次被调用都会被重新定义并分配存储空间，而 static 局部变量不会，它的值始终保存着。static 局部变量只被初始化一次，下一次使用时依旧是上一次的值。

static 函数（即静态函数，在函数定义时加上了 static 关键字）与普通函数作用域不同，它仅仅存在于本文件中。只在当前源文件中使用的函数应该说明为内部函数（即加上 static 关键字）。内部函数应该在当前源文件中声明和定义。对于可在当前源文件以外使用的函数，应该在一个头文件中说明，要使用这个函数的源文件要包含这个头文件。

程序的普通局部变量存在于堆栈中，全局变量、static 局部变量存在于静态存储区中。

例 4-22 写出下面程序的运行结果

```
#include<stdio.h>

int inc(int a)
{
    return (++a);
}

int multi(int*a,int*b,int*c)
{
    return (*c=*a**b);
}

int (*p)(int);

void show( int(*fun)(int*,int*,int*),int arg1, int *arg2)
{
    p=&inc;
    int temp =p(arg1);
    fun(&temp,&arg1,arg2);
    printf("%d\n",*arg2);
}

int main()
{
    int a;
    show(multi,10,&a);
    return 0;
}
```

程序分析。

这道试题主要测试对指向函数的指针（即函数指针）的理解。

（1）inc 函数返回参数 a 加 1 后的值。multi 函数将指针 a 和 b 所指向的数相乘，并把结果赋给指针 c 所指向的变量。p 是一个指针，它指向一个函数。这个函数有一个 int 型的参数。这个函数的返回值类型是 int。

（2）show 函数有三个参数，其中第一个参数是函数指针，这个指针指向的是一个函数，这个

我爱自由 5ifreedom.com

函数有三个参数，返回值类型为 int。show 函数内，首先把函数 inc 赋给 p，使得 p 指向了函数 inc。然后使用 p 调用函数 inc，调用结束后 temp 的值为 11。之后以函数指针 fun 调用函数 multi，显然 *arg2=(temp)*(arg1)=11*10。

例 4-23 请找出下面代码中的所有错误。这段代码的功能是把一个字符串倒序，如“abcd”倒序后变为“dcba”。

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
int main()
{
    char *src="hello,world";
    char *dest=NULL;
    int  len=strlen(src);
    dest=(char *)malloc(len);
    char *d=dest;
    char *s=&src[len];
    while(len--!=0)
        *d++=*s--;
    printf("%s\n",dest);
    return 0;
}
```

程序分析。

首先介绍两个函数：malloc 和 free 函数。

这两个函数用于动态分配和释放内存，即在需要时才向系统申请一块存储区用于保存数据，用完之后就把这块存储区释放归还给系统。

1. malloc 函数

```
#include<stdlib.h>
void *malloc(unsigned int size);
```

它的功能是在内存的动态存储区分配一个长度为 size 字节的连续空间。函数返回的是一个指向分配域起始地址的指针，这个指针的类型是 void 类型。如果函数未能执行成功则返回一个空指针 NULL。使用这个函数必须包含头文件 stdlib.h。

void 类型的指针，它指向一个变量，这个变量的类型是未定的。也就是说它可以指向 char 型变量，也可以指向 int 型或其他类型。在将它的值赋给另外一个指针时要进行强制类型转换使之适合于被赋值的变量的类型，例如：

```
char *p1 = "123456";
void *p2 = "abcdef";
p1 = (char *)p2;
```

两个指针赋值时，如把 p2 的值赋给 p1，这两个指针所指向的数据的类型必须一致。如果直接减值 p1 = p2，编译器会报告出错，因为 p1 所指向的数据的类型为 char 型，而 p2 为 void 型。当然下面这个语句是可以的：

```
p2 = (void *)p1;
```

2. free 函数

```
#include<stdlib.h>
void free(void *p);
```

它的功能是释放由 p 所指向的内存区，使这部分内存归还给系统。p 是调用 malloc 函数的返回值。free 这个函数无返回值。

注意：在程序中 malloc 函数与 free 函数须成对出现。使用 malloc 分配了一块内存，使用完毕后必须使用 free 函数将其释放。

让我们再来分析这个程序，这个程序可以说是一个写得很糟糕的程序。它犯了很多初学者容

我爱自由 5ifreedom.com

易犯的错误。这个程序的错误有：

（1）dest=(char *)malloc(len)应该改为 dest=(char *)malloc(len+1)。这个函数的本意是向系统申请一块内存以便存储"hello,world"这个字符串的倒序。strlen 这个函数返回的是字符串实际大小，即不包括每个字符串最后都有的结束字符'\0'。使用 malloc 申请内存时必须申请 len+1 个字节以便存放整个字符串。

（2）char *s=&src[len]这个语句是错误的。从后面的程序我们可以发现，它的本意是把 s 指针指向"hello，world"这个字符串的'd'处。从字符'd'开始逐个向前复制到 dest 所指向的存储区。len 的值实际为 11。src[len]='\0'，一开始没有必要把'\0'赋值过去。这个语句应该改为 char *s=&src[len-1]

（3）*d++=*s--语句之后应该增加一句*d='\0'。while 循环只是把"hello,world"这个字符串的 11 个字符赋值过去。应该把最后的'\0'也复制过去。这个 while 循环是正确的。它首先判断 len 是否为 0，如果为 0 表示已经赋值了 11 个字符就停止赋值。判断之后把 len 值减 1。*d++=*s--等价于 3 个语句：

```
*d=*s;
d++;
s--;
```

（4）printf 语句之后应该增加一个语句"free(dest);"，使用完 malloc 分配的内存后要使用 free 将其释放，否则有可能会造成内存泄漏。

例 4-24 请写出下面程序的运行结果。

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

void fun(char str[100])
{
    printf("%d\n",sizeof(str));
}

int main()
{
    char  str[] = "Hello";
    char  *p1 = str;
    int   n = 10;
    char  *p2 = (char *)malloc(100);

    printf("%d\n",sizeof(str) );
    printf("%d\n",strlen(str) );

    printf("%d\n",sizeof(p1)  );
    printf("%d\n",strlen(p1)  );

    printf("%d\n",sizeof(n)   );
    printf("%d\n",sizeof(p2)  );
    fun(p2);

    printf("\n");
    return 0;
}
```

程序分析。

这种题在招聘 C 语言软件开发工程师时极为常见。主要测试对 sizeof 和 strlen 的理解。

第一条 printf 语句输出 6，sizeof 输出的是字符串包括最后一个结束字符'\0'的总长度。字符串"Hello"有 5 个字符加上最后的结束字符'\0'共 6 个字符。

第二条 printf 语句输出 5，它所求得的长度不包含结束字符'\0'。

第三条 printf 语句输出 4，sizeof(p1)求得的是指针变量 p1 的长度，我们知道一个指针变量存放的是一个地址，而 32 位机器的一个地址都是 32 位即 4 个字节。

第四条 printf 语句输出 5，它的含义等价于第二条 printf 语句，因此也输出 5。

我爱自由 5ifreedom.com

第五条 printf 语句输出 4，一个整型变量的存储长度一般是 4 个字节，早期的机器使用 2 个字节来保存 int 型变量。

第六条 printf 语句输出 4，它也是测试指针变量 p2 的长度，因此和第三条 printf 语句一样，输出 4。

调用 fun 函数，输出的值是 4。因为数组作为函数的参数时，对系统来说，char str[100]和 char *str 是一样的。

例 4-25 一个 32 位的机器,该机器的指针是多少位？

指针是多少位是看地址总线的位数。80386 以后的机子都是 32 位的数据总线，所以指针的位数就是 32 即 4 个字节。

例 4-26 下面这段小程序的输出是什么？

```
#include<stdio.h>

void main()
{
    int a[5] = {1,2,3,4,5};
    int *ptr = (int *)(&a+1);
    printf("%d,%d\n",*(a+1),*(ptr-1));
}
```

程序分析。

&a+1 不是首地址+1，系统会认为加一个 a 数组的偏移，是偏移了一个数组的大小（本例是 5 个 int）。对于 int *ptr=(int *)(&a+1)，ptr 是&(a[5])，也就是 a+5。原因如下：&a 是数组指针，其类型为 int (*)[5]。而指针加 1 要根据指针类型加上一定的值，不同类型的指针+1 之后增加的大小不同。a 是长度为 5 的 int 数组指针，所以要加 5*sizeof(int)，因此 ptr 实际是 a[5]。但是 ptr 与（&a+1）类型是不一样的，这点很重要。所以 ptr-1 只会减去 sizeof(int*)。a,&a 的地址是一样的，但含义不一样，a 是数组首地址，也就是 a[0]的地址，&a 是对象（数组）首地址，a+1 是数组下一元素的地址，即 a[1]，&a+1 是下一个对象的地址，即 a[5]。*(a+1）就是 a[1]，*(ptr-1)是 a[4]，执行结果是 2，5。

例 4-27 下面这段小程序有什么问题？

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
int main(int argc, char* argv[])
{
    char a;
    char *str=&a;
    strcpy(str,"hello");
    printf("%s\n",str);
    return 0;
}
```

程序分析。

这个程序虽然可以编译通过，但运行时会导致程序崩溃。str 所指向的变量只占有 1 个字节，而复制字符串"hello"需要 6 个字节的空间。应该在 strcpy 函数调用前先分配动态内存并赋给指针 str：

```
str = (char *)malloc( strlen("hello")+1 );
```

在函数的返回之前，释放申请的动态内存：

```
free(str);
```

例 4-28 “int (*s[10])(int);”表示的是什么？

这个语句比较复杂，它综合了函数、数组和指针。让我们来分析一下它的含义到底是什么。首先我们看(*s[10])，s 是先和*结合还是先和[ ]结合？因为[ ]的优先级比*高，所以先与[ ]结合，也就是 s[10]，显然它是一个数组。那么数组元素的类型是什么？我们看到的是

我爱自由 5ifreedom.com

*s[10]，说明数组元素的类型是指针。那么指针的类型又是什么？是 int (*)(int)，也就是指向函数的指针，它所指向的函数有一个 int 型参数，返回一个 int 型的值。所以，这个语句的含义是：定义了一个含有 10 个函数指针的数组。数组的长度是 10，数组元素的类型是指向函数的指针。

例 4-29 假设你只知道一个数组的数组名，如何确定这个数组的长度？

假定这个数组的数组名是 table，可以用以下语句确定一个数组的长度：

```
int length = sizeof(table) / sizeof(table[0]);
```

sizeof（table）是数组的总长度，而 table[0]是数组第一个元素的长度。

例 4-30 “char a[10]；int len = strlen(a);”，len 等于多少？

上面这两条语句虽然语法上合法，但这种用法是错误的。strlen 从数组 a 首地址出发一直到遇到‘\0’才结束，计算从数组 a 的第一个元素开始到‘\0’总共有多少个字符（不包括字符‘\0’）。由于字符数组没有初始化，数组元素的值是未知的，所以 len 的值也是未定的。

如果对数组进行初始化：

```
char a[10] = "Hello";
int len = strlen(a);
```

则 len 的值是确定的，为 5。此时 sizeof（a）的值为 10，它获取的是数组 a 的总长度。

例 4-31 下面这个程序是否有问题？

```
#include<stdio.h>
#define MAX 255
int main()
{
    unsigned char a[MAX],i;

    for (i=0;i<=MAX;i++)
        a[i]=i;

    for (i=0;i<=MAX;i++)
        printf("%d  ",a[i]);

    printf("\n");
    return 0;
}
```

程序分析。

使用 gcc 编译，编译可以通过，但如果运行程序会导致数组越界访问和死循环。这个看似简单的程序，其实“暗含杀机”。用这种题目来测试应聘者的 C 语言功底，高低立现。这个测试题马上可以将应聘者按 C 语言的掌握情况分为三类：第一类，C 语言功底较差者或比较疏忽者认为这个程序是正确的；C 语言功底较一般的人会发现对数组进行了越界访问；C 语言掌握得比较好的人可以发现这个程序的两个问题。如果你很不幸，成为了第一类人，请不要灰心或懊恼，差距是可以弥补的。

注意到数组 a 的元素和变量 i 的数据类型都是无符号字符型，这种类型的数据占一个字节内存即 8 位，所表示的数的范围（以二进制表示）：00000000～11111111，也就是 0～255。

数组 a 的元素为 a[0]到 a[254]，但 for 循环中把 i<MAX 错误地写为 i<=MAX。但当 i=255 时，执行 a[255]=255，这是可以的，因为数组并不检查对数组元素的访问是否越界。i=255 时，对 i 执行 i++操作，i 的值变为多少呢？i=255，i+1 以二进制表示是 11111111+1=100000000。结果是 9 位，因为变量 i 的存储空间只有 8 位，最高位的 1 舍去，所以 i 的值变为 0。

然后执行判断语句 i<=MAX，自然是成立的，i 又从 0 开始循环了。如此周而复始，导致死循环。

我爱自由 5ifreedom.com

从中可以看到，数组对越界访问是不会检查的，检查的任务由程序员自己来完成。我们在编写程序时要注意不能对数组进行越界访问，否则会导致一些严重的错误。

例 4-32 下面这个程序编译时会报错，请指出错误并改正。

```
#include<stdio.h>
int main(void)
{
    int **p;
    int arr[100];
    p = &arr;
    return 0;
}
```

程序分析。

这个程序主要测试对数组和二级指针的理解。二级指针是 C 语言的一个难点。这个程序的错误在于不同类型的指针进行相互赋值。在前面介绍过，两个指针只有在所指向的数据的类型一致时才可以相互赋值。

&arr 是一个指向长度为 100 的数组的指针，而 p 是指向指针（该指针指向的是 int 型变量）的指针。主函数可以改为：

```
int main(void)
{
    int **p, *q;
    int arr[100];
    q = arr;
    p = &q;
    return 0;
}
```

用指针 q 进行过渡。

例 4-33 写一个程序，以递归方式反序输出一个字符串。如给定字符串“abc”输出“cba”。

本题测试递归程序的编写方法和字符串操作，程序如下：

```
#include<stdio.h>

void reverse(char *p)
{
    if( *p =='\0' )
        return;
    reverse( p+1 );
    printf( "%c", *p );
}

int main()
{
    reverse("abc");

    printf("\n");
    return 0;
}
```

程序分析。

主函数调用 reverse 函数，此时参数 p 指向字符‘a’，函数体内以指向‘b’的指针递归调用 reverse。参数 p 指向字符‘b’的函数内，以指向‘c’的指针递归调用 reverse。参数 p 指向字符‘c’的函数内，以指向‘\0’的指针递归调用 reverse。参数 p 指向字符‘\0’的函数内，因为此时 p==‘\0’，返回。返回到上一个函数，这个函数的参数 p 指向的字符是‘c’，执行 printf 语句，打印出*p 即字符‘c’，然后逐步往上，输出‘b’、‘a’。

例 4-34 请找出下面 3 个函数的错误。

```
void test1()
{
```

我爱自由 5ifreedom.com

```
    char string[10];
    char *str1 = "0123456789";
    strcpy(string, str1);
    }

void test2()
{
    char string[10], str1[10];
    for(I=0; I<9;I++)
        str1[i] = 'a';
        str[9] = '\0';
    strcpy(string, str1);
}

void test3(char* str1)
{
    char string[10];
    if(strlen(str1)<=10)
       strcpy(string, str1);
}
```

程序分析。

test1 函数应该把 string[10]改为 string[11]，使最后的 string[10]存放字符串结束字符‘\0’。

test2 有低级语法错误。I，i 没有定义。变量应该是先定义，后使用。for(I=0; I<10;I++)可以改为 for(int i=0; i<9;i++)。

test3 错误原因的说明请参考例 4-30。将 strlen 改为 sizeof。不过即使这么改，这个函数也是没有什么实际意义的。因为没有对字符数组 string 进行赋值，其数组元素的值是不确定的。将 strlen 改为 sizeof，只是遵照程序的本意进行了修改。

例 4-35 写出下面程序的运行结果。

```
#include<stdio.h>

int sum(int a)
{
    auto  int c=0;
    static int b=3;
    c+=1;
    b+=2;
    return (a+b+c);
}

void main()
{
    int i;
    int a=2;
    for(i=0;i<5;i++)
    {
        printf("%d,", sum(a) );
    }
}
```

程序分析。

auto int c=0 与 int c=0 实际是一样的。一般定义变量时默认的前缀是 auto，所以没有必要写上 auto。这个程序只是把它写全了。注意 b 是静态的，每次调用都保持着上一次调用的结果。因此运行的结果是：

```
8, 10, 12, 14, 16,
```

例 4-36 写一个递归程序，判断数组 a[n]是否是一个递增的数组。

```
#include<stdio.h>

int fun( int a[ ], int n )
{
```

我爱自由 5ifreedom.com

```
    if( n==1 )
        return 1;
    if( n==2 )
        return (a[n-1]>=a[n-2]);
    return ( fun(a,n-1) && (a[n-1]>=a[n-2]) );
}

int main()
{
    int a[] = {1,2,3,4,5,6};
    int b[] = {1,2,3,4,6,5};

    if( fun(a,sizeof(a)/sizeof(a[0])) ==1 )
        printf("a:ok\n");
    else
        printf("a:no\n");

    if( fun(b,sizeof(a)/sizeof(a[0])) ==1 )
        printf("b:ok\n");
    else
        printf("b:no\n");

    return 0;
}
```

程序分析。

判断一个数组是否是递增数组的是递归函数 fun。如果是，返回 1，否则返回 0。

函数 fun 中，如果数组 a 只有一个元素，就返回 1。如果数组有两个元素，并且数组第二个元素大于第一个则返回 1，否则返回 0。如果数组的元素为 3 个或 3 个以上，则判断最后两个元素是否递增，并递归调用 fun，以数组 a 和 n-1 为参数，也就是判断前 n-1 个元素是否为递增。

例 4-37 下面程序是否有问题。

```
#include<stdio.h>

char *RetMemory()
{
    char p[] = "hello world";
    return p;
}

void main()
{
    char *str = NULL;
    str = RetMemory();
    printf(str);
}
```

程序分析。

这个程序编译可以通过，但运行有问题。原因在于 p 所指向的字符串，只有在 RetMemory 函数被调用时才分配内存空间存储字符串“hello world”，这时 p 指向该字符串。调用结束后，分配的内存就被系统回收了，字符串“hello world”也就不存在了。此时返回的指针所指向的内容是不确定的。将这个返回值赋给 str，使 str 也指向未定内容，从而导致出错。

要绝对避免函数返回类似的指针。这种指针指向函数内部定义的变量或字符串常量。

例 4-38 写一个函数，它的原型是：

```
int findnumstring (char *outputstr,char *intputstr)
```

功能：在字符串中找出连续最长的数字串，把这个串的长度返回，并把这个最长数字串赋给其中一个函数参数 outputstr 所指内存。

例如：“abcd12345eee125ss123456789”的首地址传给 intputstr 后，函数将返回 9，outputstr 所指的值为 123456789。

我爱自由 5ifreedom.com

程序代码如下：

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

int findnumstring(char *outputstr, char *inputstr)
{
    /* temp 用于指向正在搜索中的数字子串
      final 用于指向最终的最长的数字子串
    */
    char *in = inputstr, *out = outputstr, *temp, *final;
    int count = 0, maxlen = 0;

    while( *in != '\0' )   //该循环用于找到最长的数字串，初始 in 指向输入串
    {
        // 如果 in 所指向的字符串的第一个字符为数字
        // 就获取以此数字字符开始的数字串的长度
        // 如果不是数字字符，in 指向输入串的下一个字符
        if( *in > 47 && *in < 58 )
        {
            // 注意 in 指针在变化
            for(temp = in; *in > 47 && *in < 58 ; in++ )
                count++;
        }
        else
            in++;

        // 如果 temp 所指向数字串的长度比上一次找到的长
        // 则把当前找到的最长数字串长度和地址分别赋给 maxlen 和 final
        if( maxlen < count )
        {
            maxlen = count;
            final  = temp;
        }

        // 把 count 清零
        count = 0;
    }

    // while 循环结束后，最长的数字串已经找到
   // 把找到的最长数字串存储到 out 所指的存储空间
    for(int i = 0; i < maxlen; i++)
    {
        *out = *final;
        out++;
        final++;
    }
    *out = '\0';

    return maxlen;
}

void main()
{
    // 主函数测试 findnumstring 函数是否正常工作
    char string[] = "abcd12345eee125ss123456789";
    char *p = (char *)malloc( strlen(string)+1 );
    int  count = findnumstring(p,string);
    printf("%s\nnumber string length = %d\n",p,count);
}
```

程序输出：

```
123456789
string length = 9
```

例 4-39 使用指针，将字符串“ABCD1234efgh”前后对调并显示。

程序如下：

我爱自由 5ifreedom.com

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

int main()
{
    char str[] = "ABCD1234efgh";
    int length = strlen(str);
    char *p1 = str;                    // p1 指向字符串的第一个字符
    char *p2 = str + length - 1;       // p2 指向字符串的最后一个字符

    // 逐个对调第一个和最后一个字符、第二个和倒数第二字符
   // 如此下去，直到达到字符串的中间
    while(p1 < p2)
    {
        // 对调字符
        char c = *p1;
        *p1 = *p2;
        *p2 = c;
        // 移动指针
        ++p1;
        --p2;
    }

    printf("string now is  %s\n",str);
    return 0;
}
```

例 4-40 写一个实现字符串拷贝的函数。给定字符串拷贝函数 strcpy 的原型：

```
char *strcpy(char *dest,const char *src);
```

要求：

（1）不调用任何库函数。

（2）说明函数为什么返回 char *。

程序如下：

```
char *strcpy(char *dest,char *src)
{
    if ( (dest==NULL) || (src==NULL) )
    {
        printf("arg wrong");
        return NULL;
    }
    char *ret_string = dest;
    while((*dest++=*src++)!='\0');
    return ret_string;
}
```

常见错误。

（1）不检查指针的有效性，函数应该首先判断指针 dest 和 src 是否为 NULL。如果缺少检查输入参数的代码，说明答题者不注重代码的健壮性。

（2）检查指针是否有效时，使用 if((!dest) || (!src))或者 if((dest==0) || (src= =0))。说明答题者书写代码不规范。dest 和 src 是指针变量，而!针对的是逻辑变量。使用 0 而没有用 NULL，会降低程序的可维护性，说明答题者不知道使用 NULL 常量的好处。判断指针是否为空指针时，推荐使用 NULL 而不是 0。

（3）没有返回 dest 指针，说明答题者不知道为何该函数的返回值类型是 char *。返回 dest 指针的原因是为了实现链式表达式，如：

```
int length = strlen( strcpy(dest, "hello world") );
```

strlen 求的是拷贝后的 dest 所指向的字符串的长度。如果不返回指针 dest，那么就要写两个语句，而现在这样只在一个语句里就可以完成。

我爱自由 5ifreedom.com

（4）字符串拷贝时，忘记拷贝最后的'\0'。上面的 while 语句是最精简的写法。当然为了提高程序的可读性和可维护性，最好还是尽量写得容易让别人看懂。如：

```
while( *src != '\0' )
{
    *dest=*src;
     dest++;
     src++;
}
*dest = '\0';
```

4.7 习题

1．编写一个函数，求给定字符串的长度。

2．函数的嵌套调用和递归调用有什么区别？编写递归调用要注意什么问题？

3．编写一个函数，从字符串 s 的第 i 个字符开始删除 n 个字符（注意要检查输入参数）。

4．用递归的方法求 f(n)，其中当 n = 1 时，f(n) = 1；当 n = 2 时，f(n) = 1；当 n > 1 时，f(n) = f(n−1) + f(n−2)。

5．用递归的方法求一个有 n 个元素的 int 型数组的最大值。

6．编写一个函数，求二维 int 型数组中第二大的元素。

7．编写一个函数，将两个字符串连接起来。

8．利用数组可以实现高精度计算，方法是将大整数每位上的数字存储为数组的一个元素。对于：

```
m = 88200807199688
n = 345678912345678
```

编写函数，实现大整数 m、n 的加、减、乘运算。

9．编写一个函数，统计给定的二维数组中有多少个互不相同的数，以及每个数出现的频率。

10．约瑟夫（Josephus）问题：

古代某法官要判决 n 个犯人死刑，他有一条荒唐的逻辑，将犯人首尾相接排成圆圈，然后从第 s 个人开始数起，每数到第 m 个犯人，就拉出来处决；然后又数 m 个，数到的犯人又拉出来处决，依此类推。剩下的最后一人可以赦免。

编写程序，给出处决顺序，并给出哪一个人可以活下来。

11．什么是指针，指针变量和指针所指向的变量的区别和联系是什么？

12．编写程序，利用指针传递参数，实现两个字符串变量的交换。

13．请解释静态变量、全局静态变量、局部静态变量、静态函数。

14．编写一个比较两个字符串 s1、s2 大小的函数。当 s1 > s2 时，输出 1；当 s1 = s2 时，输出 0；当 s1 < s2 时，输出−1。

15．编写一个函数，求两个字符串的长度最大的公共子串。

我爱自由 5ifreedom.com

第 5 章　C 语言预处理、结构体和 make 的使用

本章将介绍宏定义、文件包含、条件编译等编译预处理命令；结构体、共用体两种复合数据类型；以及位运算操作。本章还将介绍 Linux 项目开发中常用的 Makefile 的编写。

本章重点：

- 预处理命令。
- 结构体。
- 位运算。
- 编写 Makefile。

本章难点：

- 结构体与数组、指针的关系。
- 编写 Makefile。

5.1 C 语言预处理命令

在 C 语言源程序中加入一些编译预处理命令可以提高编程效率，加快编译速度。预处理命令是在编译源程序前先对源程序进行处理，例如，在程序中使用“#define MAX 256”这条命令定义一个符号常量 MAX，则在预处理时将程序中出现的所有 MAX 替换为 256。预处理完成后，编译器（如 gcc）开始编译源程序以生成可执行代码。需要注意的是，预处理命令并不是 C 语言的一部分，因此每条编译预处理命令不需要以分号来结束。

5.1.1 宏定义

C 语言标准允许在程序中用一个标识符来表示一个字符串，称为宏。标识符称为宏名。在编译预处理时，将程序中所有的宏名用相应的字符串来替换，这个过程称为宏替换。宏分为两种：无参数的宏和有参数的宏。

1. 无参数宏

无参数宏定义的一般形式为：

```
#define 标识符 字符串
```

“#”代表本行是编译预处理命令。define 是宏定义的关键词，标识符是宏名。字符串是宏名所代替的内容，可以是常数、表达式等。

注意：宏定义和其他编译预处理命令不是以分号结尾的。

例 5-1　下面是一个使用无参数宏的程序

```
#include<stdio.h>
#define PI 3.1415926
```

我爱自由 5ifreedom.com

```
int main()
{
    int    r = 100;
    double length = 2*PI*r;
    printf("The circumference is %f\n",length);  /* 输出周长*/
    return 0;
}
```

程序说明。

（1）本程序使用 PI 来代表 3.1415926。宏替换是在程序中用相应的字符串来替换宏名，编译器预处理程序对它不作任何检查。如果有错误，只能在编译程序时才会被编译器发现。

（2）习惯上，宏名都用大写字母。当然也可以用小写字母。

（3）使用宏代替一个字符串，可以减少程序中重复书写某些字符串的工作量。可以用一个有意义的宏名来代表无规律的字符串，提高程序的可读性，同时修改起来也方便。如果要把程序中的 PI 值改为 3.14，则只要修改#define 这一行即可。如果没有使用宏，那么就要查找程序并修改所有的 PI 值。

（4）宏的作用范围是从宏定义开始到本源程序文件结束为止。也可以使用#undef 来提前终止作用范围。例如：

```
#define MAX 256
int main( )
{
    . . .
}
#undef MAX
int f( )
{
    . . .
}
```

由于使用了#undef，使宏名 MAX 只在 main 函数中有效。

（5）宏定义允许嵌套。例如：

```
#define  MIN   128
#define  MAX  MIN*2
```

定义 MAX 宏时使用了前面已经定义的 MIN。

2．有参数宏

有参数的宏类似于有参数的函数，其定义的一般形式为：

```
#define 标识符(形参表) 字符串
```

如果有多个形参，像函数参数一样以逗号隔开。

在程序中使用有参数宏的形式是：

```
标识符(实参表)
```

例 5-2 演示了有参数宏的实现方法。

例 5-2

```
#include<stdio.h>
#define  MAX(x,y)  (x>y?x:y)

int main()
{
    int  a = 5,b = 10,max;

    max = MAX(a,b);
    printf("The max between(%d,%d) is %d\n",a,b,max);

    return 0;
}
```

我爱自由 5ifreedom.com

程序说明。

经过编译预处理 max = MAX(a,b)就替换为 max = (a>b?a:b)。

程序第二行的宏定义中表达式 x>y?x:y 两边的括号不是必需的，但出于良好的编程规范应该加上。如果没有括号往往会导致一些意想不到的问题。比如有一个宏定义：

```
#define MUL(x,y) x*y
```

在程序中有：

```
int a = 5,b = 10,c;
c = MUL(a+1,b+1);
```

那么进行宏替换，a + 1 是 x 的实参，b + 1 是 y 的实参，替换后的结果为：

```
c = a+1*b+1;
```

显然这是不符合要求的。应该按照如下方式进行宏定义：

```
#define MUL(x,y) (x)*(y)
```

此时宏展开后：

```
c = (a+1)*(b+1);
```

定义有参数的宏时，应该注意。

- 宏名与形参表的圆括号之间不能有空格，否则会导致错误。例如：#define MUL(x,y) (x)*(y)，MUL 与“(”之间不能有空格。
- 在宏定义中，字符串内的形式参数最好用括号括起来，以避免错误。例如：#define MUL(x,y) (x)*(y)中的字符串(x)*(y)、x、y 都用括号括起来。

带参数的宏与函数的比较。

- 有参数宏的形式参数不是变量，不分配内存空间，无需说明数据类型。而函数的形参是变量，要分配内存空间，在函数定义时要指明参数的数据类型。
- 预处理程序认为有参数宏的实参是字符串，并用它去替换形参。如上面的例子中，用 a + 1 去替换 x，而不是先计算 a + 1 的值再去替换 x。如果是函数，则先计算 a + 1 的值，再把这个值传递给 x。
- 使用宏的次数较多时，宏替换后源程序一般会变长。而函数调用不会使程序变长。宏替换不会占用运行时间，只是编译的时间稍微变长一点。而函数调用则会占用运行时间。一般用宏来代表一些较为简单的表达式比较合适。

5.1.2 文件包含

文件包含预处理命令#include 前面已经使用过了。它把指定源文件的全部内容包括到当前源程序文件中，其一般形式为：

```
#incldue<文件名>
```

或者

```
#include"文件名"
```

文件包含命令是把指定文件的全部内容包括进来，插入到命令所在位置，取代原来的命令行。由当前源文件和指定文件组成一个文件，一起进行编译。

一个#include 只能包含一个文件，要包含多个文件，需要使用多个#include 命令。例如：

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

一个大的程序往往被分成多个模块，由多个程序员分别编写。公用的信息，如常量定义、函数声明，可以单独放在一个文件中，在其他文件的开头使用#include 命令包含进来。这样可以避

我爱自由 5ifreedom.com

免每个文件头部都去书写那些公用量，既节省了时间又可以防止出错的可能。

文件包含命令中的文件名既可以用尖括号也可以用双引号括起来，它们的区别在于查找指定文件的位置不同。尖括号只在缺省目录里找指定文件，缺省目录是由用户设置的编程环境决定的。双引号则先在源程序文件所在的当前目录里查找指定文件，如果没有找到再到缺省目录里找。如果指定文件与当前编写中的源程序处在同一个目录里，就必须使用双引号来包含该文件，否则编译程序时编译器会报告找不到指定的头文件。

5.1.3 条件编译

一般情况下，源程序中所有的行都被编译。有时希望其中一部分内容只在某个条件成立或不成立时才去编译，也就是对一部分内容指定编译的条件，这就是条件编译。

条件编译使用范式。

➢ 范式一

```
#ifndef 标识符
程序段 1
#endif
```

其含义是：如果没有定义标识符，就编译程序段 1。

这里的程序段 1 既可以是语句组，也可以是命令行。

使用示例：

```
#ifndef _getkey_h
    #define _getkey_h
    #include<sys/types.h>
#endif
```

这段代码的含义是：如果没有定义符号常量_getkey_h，就定义该常量并且包含头文件 sys/types.h。

➢ 范式二

```
#ifndef 标识符
程序段 1
#else
程序段 2
#endif
```

其含义是：如果没有定义标识符，就编译程序段 1，否则编译程序段 2。

➢ 范式三

```
#ifdef 标识符
程序段 1
#endif
```

其含义是：如果定义了标识符，就编译程序段 1，否则不编译该程序段。

使用示例：

```
#ifdef  DEBUG
printf("a=%d,b=%d",a,b);
#endif
```

在调试程序时，可以在源程序头部加入如下语句：

```
#define DEBUG
```

这样在软件开发阶段，编译运行程序时会输出变量 a、b 的值。当程序调试完毕，在源程序文件头部删除这一行，则用户运行时不会输出 a、b 的值。这里打印出 a、b 值只是供调试使用。

➢ 范式四

```
#ifdef 标识符
程序段 1
#else
程序段 2
#endif
```

我爱自由 5ifreedom.com

其含义是：如果定义了标识符，就编译程序段 1，否则编译程序段 2。

➢ 范式五

```
#if 表达式
程序段 1
#endif
```

其含义是：如果表达式成立，就编译程序段 1，否则不编译该程序段。

使用示例：

```
#include<stdio.h>
#define MAX(x,y) (x>y?x:y)
. . .
int a = 5,b = 10,c;
. . .
#if c
c = MAX(a,b);
#endif
```

如果变量 c 存在，就调用宏 MAX(a,b)获得 a、b 的最大值，并把该值赋给变量 c。

➢ 范式六

```
#if 表达式
程序段 1
#else
程序段 2
#endif
```

其含义是：如果表达式成立，就编译程序段 1，否则编译程序段 2。

事实上，不用条件编译而直接使用 if…else 语句也可以达到要求。但采用条件编译，可以减少被编译的语句，从而减少可执行程序的长度，缩短程序运行时间。当条件编译的程序段比较多时，可执行程序的长度可以大大减少。

5.2 结构体和共用体

到目前为止，我们介绍了基本数据类型：整型、字符型、浮点型，还介绍了一种复合数据类型：数组。但在实际开发中，只有这些数据类型往往是不够的。例如，在一个人事管理系统中，人员的信息有：姓名（字符串）、年龄（整型）、性别（字符型）、联系电话（字符串）等。如果单独定义它们，难以反映它们之间的内在联系。C 语言中有一种被称为结构体的构造数据类型可以较好地完成这一任务。

5.2.1 声明和引用结构体

1. 声明结构体

声明结构体的一般形式为：

```
struct 结构体名
{
    成员列表
};
```

struct 是声明结构体的关键词。结构体名用来标识一个结构体。成员可以是任何 C 语言的数据类型，如整型、字符型、浮点型、数组、指针、结构体，以及后面将要介绍的共用体、枚举类型。

例如：

```
struct person
{
```

我爱自由 5ifreedom.com

```
    char name[20];
    int  age;
    char sex;
    char phone[15];
};
```

注意：声明结构体也是一个 C 语言语句，因此要以分号作为该语句的结束。声明结构体时漏写分号是一种常见的语法错误。

声明了一个结构体，相当于构造了一种新的数据类型。此时系统并不为它分配内存空间，只有定义了结构体类型的变量，系统才为该变量分配内存空间。声明了结构体后，就可以定义结构体类型的变量，定义的一般形式为：

结构体名 变量名；

例如：

```
struct person p1, p2;
```

以 person 为数据类型，定义了两个变量 p1、p2，struct 为定义结构体变量的关键词。定义了结构体变量后，系统为它分配内存。p1 所占的字节为：20+4+1+15 = 40。

```
printf("%d\n", sizeof(p1) );
```

上述语句将打印出 40。

如果把 char phone[15]改为 char phone[16]，上面的 printf 语句是否会打印出 41 呢？实际上打印的数值为 44。这是因为结构体的各个成员是按顺序连读存放在内存中的。如图 5-1 所示。

p1：

name[0] ... name[19]	age	sex	phone[0] ... phone[15]

图 5-1　结构体存在内存示意图

内存为了提高访问效率有一个规则：4 字节对齐。虽然修改后的 person 结构体所占的字节数为 41，但分配内存时系统为它分配 44 个字节。因此，上面的 printf 语句打印出的值是 44 而不是 41。

在声明结构体类型时也可以定义变量，例如：

```
struct person
{
    char name[20];
    int  age;
    char sex;
    char phone[15];
}p1, p2;
```

在声明的同时定义了两个结构体变量 p1、p2。

或者也可以这样定义：

```
struct
{
    char name[20];
    int  age;
    char sex;
    char phone[15];
}p1, p2;
```

这种方式没有声明结构体名，因此不能再定义其他变量，只有 p1、p2 两个变量。

结构体的成员可以是结构体、指针等类型，例如：

```
struct date
{
    int year;
    int month;
    int day;
};
```

我爱自由 5ifreedom.com

```
struct person
{
    char  name[20];
    int   age;
    char  sex;
    char  phone[15];
    struct date birthday;
    char  *string;
};
```

先声明了 struct date 结构体，它有 3 个成员：year（年）、month（月）、day（日），用来表示人员出生的年月日。在声明结构体 person 时，birthday 指定为 struct date 类型，同时还声明了一个 char 型指针。

2. 引用和初始化结构体变量

引用结构体变量中的成员的一般方式为。

结构体变量. 成员名

例如：

```
struct person p1;
strcpy(p1.name,"John");
p1.age = 24;
p1.sex = 'm';
strcpy(p1.phone, "123456789");
printf("%s", p1.name);
```

“.”是成员运算符，用于取得结构体中的成员，它在所有运算符中优先级最高。语句：

```
p1.age = 24;
```

把 24 赋给结构体变量 p1 的 age 成员变量。

对于 person 结构体中的 birthday 成员的赋值方式是：

```
p1.birthday.year = 1985;
```

在定义的同时也可以进行初始化，例如：

```
struct person p2 = {"Bill",22,'m',"987654321"};
```

结构体中的成员变量可以像其他变量一样进行各种运算，例如：

```
p1.age++;
```

这个语句中，“.”的优先级比++高，因此先进行 p1.age，也就是取得 p1 变量的 age 成员，然后对该成员执行++运算。

注意：只能引用结构体变量中的各个成员，而不能引用整个结构体变量。如以下语句是错误的：

```
printf("%s,%d,%c,%s",p1);
```

必须是：

```
printf("%s,%d,%c,%s",p1.name, p1.age, p1.sex, p1.phone);
```

5.2.2 结构体和数组

结构体中可以有数组类型的成员。数组的元素也可以是结构体。例如，有一个用于存储学生信息的结构体：

```
struct student
{
    int  number;
    char name[20];
    char sex;
    int  age;
    char addr[30];
};
struct student s[2] = { {10000,"zhang",'m',21, "Shang Hai"},
                        {10001, "li",'f',20, "Bei Jing"} };
```

结构体中有两个数组成员，用于保存学生的姓名和地址。然后定义了一个结构体数组，数组

我爱自由 5ifreedom.com

长度为 2，定义的同时进行了初始化。我们看到，结构体数组的初始化是数组的初始化和结构体变量的初始化的结合。数组的初始化是把各个数组元素的值放在一个大括号里，各个元素以逗号分隔。结构体的初始化是把结构体的各个成员放在一个大括号里，各个成员也以逗号分隔。

例 5-3 结构体数组使用示例

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

int main()
{
    struct student
    {
        int  number;
        char name[20];
        char sex;
        int  age;
        char addr[30];
    };
    struct student s[3] = { {10000,"Zhang",'m',21,"Shang Hai"},
                            {10001,"Li",'f',20,"Bei Jing"} };
    s[2].number = 10002;
    s[2].sex = 'm';
    s[2].age = 22;
    strcpy(s[2].name,"Liu");
    strcpy(s[2].addr,"Guang Dong");

    printf("%d,%s,%c,%d,%s\n",s[0].number,s[0].name,s[0].sex,s[0].age,s[0].addr);
    return 0;
}
```

程序输出：

```
10002, Liu, m, 22, Guang Dong
```

5.2.3 结构体和指针

我们已经知道结构体中各个成员按顺序连读存放在内存中。一个结构体指针指向结构体变量，结构体指针所保存的值是它所指向的结构体变量所占内存的首地址。

以上面的 student 结构体为例：

```
struct student s1;
struct student *p;
s1.number = 10002;
s1.sex = 'm';
s1.age = 22;
strcpy(s1.name,"Liu");
strcpy(s1.addr, "Guang Dong");
p = &s1;
printf("%d,%s,%c,%d,%s\n",p->number,p->name,p->sex,p->age,p->addr);
```

这段代码的输出与例 5-3 的输出是一样的。结构体变量 s 和结构体指针 p 在内存中的结构如图 5-2 所示：

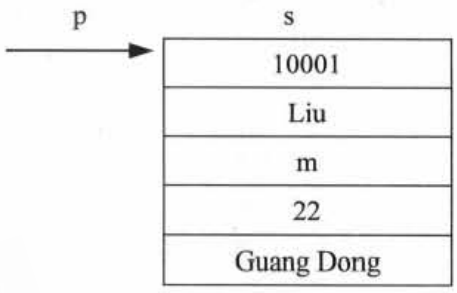


图 5-2 结构体变量和结构体指针在内存中的示意图

在 C 语言中，为了方便使用和直观，把(*p).number 改为 p->number。前面一种在语法上也是正确的，但一般不使用。

对于指针，访问它所指向的结构体的成员一般形式为：

指针->结构体成员

下面表达式的含义如下所示。

p->number++：得到 p 所指向的结构变量中成员 number 的值，使用该值后再加 1。

我爱自由 5ifreedom.com

++p->number：得到 p 所指向的结构变量中成员 number 的值，先加 1，再使用。

例 5-4 指向结构体数组的指针

```
#include<stdio.h>

int main()
{
    struct student
    {
        int  number;
        char name[20];
        char sex;
        int  age;
        char addr[30];
    };
    struct student s[3] = { {10000,"Zhang",'m',21,"Shang Hai"},
                            {10001,"Li",'f',20,"Bei Jing"},
                            {10002,"Liu",'m',22,"Guang Dong"} };
    struct student *p;
    for(p=s; p<s+3; p++)
        printf("%d,%s,%c,%d,%s\n",p->number,p->name,p->sex,p->age,p->addr);
    return 0;
}
```

程序输出：

```
10000,Zhang,m,21,Shang Hai
10001,Li,f,20,Bei Jing
10002,Liu,m,22,Guang Dong
```

程序说明。

p 开始指向结构体数组 s 的第一个元素，即第一个结构体变量，p++使 p 指向数组的第二个元素。

例 5-5 结构体指针作函数的参数

```
#include<stdio.h>

struct student
{
    int  number;
    char name[20];
    char sex;
    int  age;
    char addr[30];
};

void print_struct(struct student *p)
{
    p->age ++;
    printf("%d,%s,%c,%d,%s\n",p->number,p->name,p->sex,p->age,p->addr);
}

int main()
{

    struct student s[3] = { {10000,"Zhang",'m',21,"Shang Hai"},
                            {10001,"Li",'f',20,"Bei Jing"},
                            {10002,"Liu",'m',22,"Guang Dong"} };
    printf("%d,%d,%d\n",s[0].age,s[1].age,s[2].age);
    struct student *p;
    for(p=s; p<s+3; p++)
        print_struct(p);
    printf("%d,%d,%d\n",s[0].age,s[1].age,s[2].age);
    return 0;
}
```

程序输出：

```
21,20,22
10000,Zhang,m,22,Shang Hai
```

我爱自由 5ifreedom.com

```
10001,Li,f,21,Bei Jing
10002,Liu,m,23,Guang Dong
22,21,23
```

程序说明。

print_struct 函数以结构体指针作为参数。在函数中对 age 成员执行了加 1 操作。由于使用了指针作为参数，因此在函数内部对 age 值作的改变，使得数组的实际值也改变了。这是可以从第一行输出和第五行输出的不同看出来。

5.2.4 共用体

共用体把几种不同数据类型的变量存放在同一块内存里。共用体中的变量共享同一块内存。

定义共用体类型变量的一般形式为：

```
union 共用体名
{
    成员列表
}变量列表;
```

与结构体类似，变量列表是可选的。如果没有变量列表，那只是声明了一种共用体类型。声明之后，可以定义该类型的变量。

访问共用体变量的方式是：

结构体变量. 成员

例如：

```
union data
{
    int             i;
    char            c;
    double          d;
};
union data a;
```

共用体变量 a 中的成员 i、c、d3 个变量在内存中从同一个地址开始存储。如进行如下赋值：

```
a.i = 100;
a.c = 'A';
```

那么此时共用体变量 a 中的成员 i 已经没有值了，因为存储该值的内存现在已经被用来存储成员 c 的值了。同一块内存可以用来存放几种不同类型的数据，但在某以时刻只能在其中存放一个成员变量。共用体变量中起作用的成员是最后一次存入的数据。

共用体变量的长度取决于其成员的最大长度。

```
printf ( "%d",sizeof(a) );
```

输出的结果是 8。因为 double 在计算机中使用 8 个字节来存储。共用体变量 a 的长度取决于其长度最长的成员 d 的长度。这只是简单情况，事实上共用体和结构体一样，为了提高访问效率，还要进行内存对齐，因此其实际所占的内存有时会大一些。关于结构体和共用体的内存对齐，在面试题选中会详细分析。

结构体变量所占内存的长度是各个成员的总和，每个成员分别占有自己的存储空间。共用体变量所占内存的长度是其最长成员的长度。当然，编译器出于提高访问效率的目的，在编译分配存储空间时往往要进行对齐操作。

不能把共用体变量作为函数参数，也不能使函数返回共用体变量，但可以使用指向共用体变量的指针。共用体类型的变量可以出现在结构体的声明中，也可以作为数组元素的类型。反之，结构体也可以出现共用体类型的声明中，数组也可以作为共用体的成员。

我爱自由 5ifreedom.com

5.2.5 使用 typedef

软件开发中，除了可以使用 C 语言提供的标准类型名（如 int、char、float、double、long）和声明的结构体、共用体、指针外，还可以使用 typedef 声明新类型名来替代已有的类型名。例如：

```
typedef int  INT;
```

声明之后，INT 和 int 就是等价的，可以用如下方式之一定义整型变量：

```
int  i, j, k;
INT i, j, k;
```

它们是完全等价的。

可以在声明结构体类型时使用 typedef：

```
typedef struct
{
    int year;
    int month;
    int day;
}DATE;
```

声明新的结构体类型为 DATE，之后可以使用 DATE 来定义变量：

```
DATE birthday;
```

还可以：

```
typedef int NUMBER[10];
```

NUMBER 声明为含有 10 个元素的数组类型。

```
NUMBER n;
n[0] = 1;
```

n 为含有 10 个元素的数组。

typedef 只是为某种类型声明一个别名，和声明数据类型一样（如声明结构体类型），使用 typedef 并不是定义变量，也不会分配存储空间。在实际项目开发中，往往使用 typedef 为某种已存在的数据类型声明一个别名，使用该别名定义的变量往往有特殊用途，这样可以增加程序的可读性和可维护性。

5.3 位运算

5.3.1 位运算符和位运算

C 语言可以直接对二进制位进行操作，这使得用它编写的程序执行起来具有很高的效率。C 语言是一种适合用来编写系统程序的语言。现在很多嵌入式设备上都使用 C 语言开发。

C 语言的位操作符如表 5-1 所示。

表 5-1　　位操作符

位 操 作 符	含　义
&	按位与运算符
|	按位或运算符
^	按位异或运算符
~	按位取反运算符
<<	左移运算符
>>	右移运算符

我爱自由 5ifreedom.com

1．按位与运算符“&”

参与运算的两个数，按二进制位进行与运算。规则是：相应的两个二进制位如果都为 1 则该位的运算结果就为 1，否则为 0。

0 & 0 = 0　　0 & 1 = 0　　1 & 0 = 0　　1 & 1 = 1

例如，有两个 char 型变量 i、j，分别赋值 10、7（注意是数值不是字符）。变量 i 的值 10 用二进制表示为 00001010，变量 j 的值 7 的二进制表示为 00000111。

```
    00001010
&   00000111
------------
    00000010
```

i & j = 10 & 7 = 2。

2．按位或运算符“|”

参与运算的两个数，按二进制位进行或运算。规则是：相应的两个二进制位只要有一个为 1 则该位的运算结果就为 1，否则为 0。

0 | 0 = 0　　0 | 1 = 1　　1 | 0 = 1　　1 | 1 = 1

还是以上面的变量 i、j 为例：

```
    00001010
|   00000111
------------
    00001111
```

i | j = 10 | 7 = 15。

3．按位异或运算符“^”

参与运算的两个数，按二进制位进行异或运算。规则是：相应的两个二进制位如果相同则该位的运算结果就为 0，否则为 1。

0 ^ 0 = 0　　0 ^ 1 = 1　　1 ^ 0 = 1　　1 ^ 1 = 0

还是以上面的变量 i、j 为例：

```
    00001010
^   00000111
------------
    00001101
```

i ^ j = 10 ^ 7 = 13。

4．按位取反运算符“～”

按位取反运算符～是一个单目运算符，用来对一个二进制数按位取反，即将 0 变为 1，将 1 变为 0。例如：

```
~   00001010
------------
    11110101
```

5．左移运算符“<<”

左移运算符<<用来将一个数的各个二进制位向左移动若干位。例如：

```
a = a << 2;
```

将 a 的二进制左移两位，右边空出的两位补 0。如果 a = 15，即二进制数 00001111，左移两位后为 00111100，即十进制数 60。左边的两位舍弃不要，右边两位自动补 0。

左移一位相当于将该数乘 2，左移两位相当于乘以 4，左移 3 位相当于乘以 8，左移 4 位相当

我爱自由 5ifreedom.com

于乘以 16，依此类推。但前提是左移时没有溢出，即左移时被舍弃的高位中不含 1。

6．右移运算符“>>”

右移运算符>>用来将一个数的各个二进制位向右移动若干位。右边被移出的位被舍弃。左边如果最高为 0，则补 0；如果最高为 1，则根据系统的不同而移入不同的值，有的移入 1，有的移入 0。

例如，a = 16，a >> 2，即 00010000 >> 2 = 000000100。

右移一位相当于除以 2，右移 n 位相当于除以 2^n。

当某个数的最高位为 1，有的系统移入 0，有的移入 1。移入 0 的称为逻辑右移，移入 1 的称为算术右移。例如：对于 10001111，逻辑右移两位为 00100011，算术右移两位为 11100011。

位运算符与赋值运算符相结合构成扩展的赋值运算符，例如：

&= ，|= ，^=，>>=，<<=。

5.3.2 位域

在内存中存取数据的最小单位一般是字节，但有时存储一个数据不必用一个字节。例如，对于条件判断，只有真和假两个值，用 0 和 1 表示，只需一位就可以了，而不需要一个字节。在某些条件下（如一些嵌入式开发）可供使用的内存往往比较小，此时就要求节约使用内存。对于不需要使用一个字节来存储的数据用几个 bit 位来存储就可以了。

C 语言允许在一个结构体中以位为单位来使用内存，这种以位为单位的成员称为位域或位段。在某些内存受限制的开发环境中，使用位域是节约内存的一种重要手段。

以下定义了一个含有位域的结构体：

```
struct bit_data
{
    int a: 6;
    int b: 4;
    int c: 4;
    int d;
};
```

short int 占 2 两个字节，结构体 bit_data 中，成员 a 使用 6 位，b 使用 4 位，c 使用了 4 位，整型变量 d 使用 4 字节即 32 位。存储空间如图 5-3 所示。

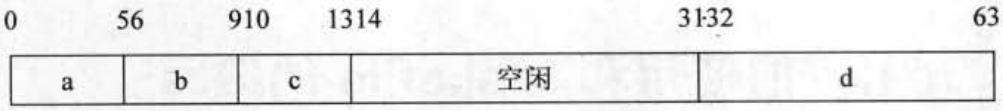


图 5-3　存储空间示意图

0～5 的 6 位用于存放变量 a，6～9 的 4 位用于存放 b，10～13 的 4 位用于存放变量 c，14～31 的 18 位空闲，32～63 的 32 位用于存放变量 d。结构体成员变量 a、b、c 共享一个字节的存储空间。整个结构体占用 8 个字节，如果不使用位域则需要使用 16 个字节。试想如果一个程序在运行过程中需要存储大量的数据，使用位域可以节省很多内存空间。

```
struct bit_data data;
data.b = 18;
```

其中第二条语句是不正确的用法，因为成员 b 只占 4 位的存储空间，它可存储的数的范围是 0000～1111，即 0 到 15，18 显然超出这个范围了。但第二条语句也是符合语法的。那么 b 实际存储的值是多少呢？18 的二进制表示为 00010010，由于只能存储 4 位，那么高 4 位即 0001 被舍弃，实际存储的是 0010 即 2。

我爱自由 5ifreedom.com

若某一位段要从另外一个存储单元开始存放，结构体中的成员可以定义成如下形式：

```
int a: 6;
int b: 4;
int : 0;
int c: 4;
int d;
```

则 a 使用 32 位的前 6 位即 0～5，b 使用 6～9 位。c 从下一个存储单元开始存放，即存放在 32～35 位，10～31 位空闲。这里的一个存储单元是 4 个字节，因为这个结构体中存储长度最大的是 int，它占 4 个字节。这涉及到存储空间的补齐问题。关于存储空间的补齐请详见 5.5 节面试题选部分。

对于：

```
struct bit_data
{
    int a: 6;
    int b: 4;
    int : 8;
    int c: 4;
    int d;
};
```

0～5 的 6 位存放成员 a，6～9 的 4 位存放成员 b，10～17 的 8 位强制空闲，18～21 的 4 位存放成员 c，22～31 位的 10 位也空闲。32～63 的 32 位存放 d。使用这种方式可以使某些位强制空闲不予使用。存储空间如图 5-4 所示。

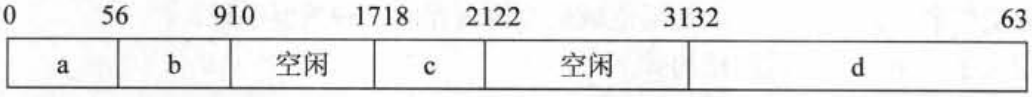


图 5-4　存储空间示意图

位域除了在所存储的位数范围有所限制之外，其余与一般的结构体成员是一样的。例如：

```
struct bit_data data;
data.b = 4;
data.c = 4;
data.a = data.b * data.c;
printf("%d,%d,%d", data.a, data.b ,data.c);
```

5.4 make 的使用和 Makefile 的编写

Linux 环境下的大型项目开发中，通常把整个系统划分为若干模块，每个模块完成某一相对独立的功能，各个模块相互作用以构成一个完整的系统。对于这样的一个软件系统，是不可能只使用一条或几条 gcc 命令就可以编译生成可执行程序的。而且模块通常要经历几次修改，每次模块修改后如果都由人工来输入命令以完成编译，这样既效率低下又容易出错。

在 Linux 中，有一个用来维护程序模块关系和生成可执行程序的工具——make。它可以根据程序模块的修改情况重新编译链接生成中间代码或最终的可执行程序。执行 make 命令，需要一个名为“Makefile”或“makefile”的文本文件，这个文件定义了整个项目的编译规则。它定义了模块间的依赖关系，指定文件的编译顺序，以及编译所使用的命令。有了 make 命令和 Makefile 文件，整个项目的源程序文件可以自动编译，极大地提高了软件开发效率。本节主要介绍 make 命令的使用和如何编写简洁、高效的 Makefile 文件。

我爱自由 5ifreedom.com

5.4.1 make 的一般使用

make 从 Makefile 文件中获取模块间的依赖关系，判断哪些文件过时了，根据这些信息 make 确定哪些文件需要重新编译，然后使用 Makefile 中的编译命令进行编译。所谓过时，是指一个文件生成后，用来生成该文件的源文件或头文件被修改了，导致生成该文件所需要的源文件或头文件的修改时间比生成该文件的时间晚。

下面是一个简单的 Makefile 文件：

```
main:main.o module1.o module2.o
        gcc main.o module1.o module2.o -o main
main.o:main.c head1.h head2.h common_head.h
        gcc -c main.c
module1.o:module1.c head1.h
        gcc -c module1.c
module2.o:module2.c head2.h
        gcc -c module2.c
#This is a makefile
```

1. Makefile 的基本构成

Makefile 文件的基本单元是规则。一条规则指定一个或多个目标文件，目标文件后面跟的是编译生成该目标文件所依赖的文件或模块，最后是生成或更新目标文件所使用的命令。规则的格式如下：

目标文件列表 分隔符 依赖文件列表 [；命令]

[命令]

[命令]

其中，“[ ]”中的内容是可选的。

在上面的 Makefile 文件中，第一、二行就构成了一条规则。目标文件列表中只有一个目标文件即 main。main 后面的冒号(:)是分隔符，一般分隔符都是冒号，其他分隔符很少使用。依赖文件列表是 main.o module1.o module2.o，也就是为了生成可执行程序 main，需要先生成这些依赖文件。这些依赖文件是以.o 结尾的，说明它们是一些只经过编译和汇编，没有进行过链接的中间代码。

第二行是生成目标文件所使用的命令。需要特别注意的是，如果某一行是命令，那么它必须以一个 Tab 键开头。另外，如果某一行以 Tab 键开头，make 就认为这一行是命令。第一行用于指明模块间的依赖关系，不是命令，因此不以 Tab 键开头。第三五七行与第一行类似，因此也都不以 Tab 键开头。第三四两行也构成了一条规则，它指明 main.o 这个目标文件依赖于 main.c 源程序文件和 3 个头文件。生成 main.o 目标文件所使用的命令是 gcc -c main.c。

最后一行以#开头，是注释行，Makefile 文件中，如果某行以#开头，那么 make 就认为它是一行注释，因而就不会解析这行内容。

属性在后面会介绍。在依赖文件列表后加上一个分号(;)，可以跟上命令。例如：

```
main:main.o module1.o module2.o ;gcc main.o module1.o module2.o -o main
main.o:main.c head1.h head2.h common_head.h
        gcc -c main.c
```

2. make 如何解释执行 Makefile

假设上面那个名为 Makefile 的文件和 Makefile 文件中所涉及的源文件、头文件都在当前目录下，执行如下命令就开始自动编译。

```
[root@mci tmp]# make
```

make 首先在当前目录下寻找名为“Makefile”或“makefile”的文件。找到 Makefile 文件后，在当前目录下寻找第一行中的目标文件 main，发现没有，就去寻找生成 main 文件所依赖的文件即：main.o module1.o module2.o，发现也没有。然后跳过第二行的编译命令，定位到第 3 行，第 3 行中的目标文

我爱自由 5ifreedom.com

件 main.o 也没有，但它所依赖的源文件和头文件在当前目录下都被找到，于是执行第 4 行的命令，从而生成了 main.o 文件。make 之后定位到第 5 行，发现目标文件 module1.o 没有，但它所依赖的文件 module1.c 和 head1.h，在当前目录下都被找到，于是执行第 6 行的编译命令，生成 module1.o。第 7、8 行与第 5、6 行类似。make 然后定位到最后一行，发现它是个注释行，不予理睬。之后 make 回溯到第一行，此时依赖文件都已经生成，于是就执行第二行的编译命令，最后生成了目标文件 main。

执行 make 后，输出如下所示。它提示了 make 执行过的 Makefile 中的一些命令。

```
gcc -c main.c
gcc -c module1.c
gcc -c module2.c
gcc main.o module1.o module2.o -o main
```

现在假设，我们修改了头文件 head1.h。然后在当前目录下运行 make。

```
[root@mci tmp]# make
```

此时 make 又是如何工作的呢？make 首先在当前目录下寻找名为“Makefile”或“makefile”的文件。找到 Makefile 文件后，再在当前目录下寻找第一行的目标文件 main，发现已存在，然后寻找 main 的依赖文件：main.o module1.o module2.o，发现也存在。make 就开始比较目标文件 main 和它的 3 个依赖文件的修改时间，发现 main 的修改时间比它的 3 个依赖文件晚，于是 make 就认为目标文件 main 是最新的，那么就没有必要执行第二行的命令以生成目标文件 main 了。

之后 make 定位到第三行，由于刚刚修改了头文件 head1.h，导致 head1.h 的修改时间比 main.o 晚，于是 make 认为目标文件 main.o 是过时的，就执行第四行的命令从而生成了一个新的 main.o 文件。然后 make 定位到第五行，发现 head1.h 比 module1.o 新，就执行第六行的命令生成一个新的 module1.o。make 再定位到第七行，经过比较文件修改时间，判断出不用执行第八行的命令。之后定位到最后的第九行，发现是个注释行，不予理睬。

从头到尾扫描完一遍 Makefile 文件后，make 开始回溯，当回溯到第一行时，发现 main.o module1.o 的修改时间比目标文件 main 晚，于是就执行第二行的命令，从而生成一个新版本的 main。回溯完成，make 终止。打印出它执行过的命令：

```
gcc -c main.c
gcc -c module1.c
gcc main.o module1.o module2.o -o main
```

3. 其他说明

命令行之间可以插入可以插入任意多个空行，空行也要按 Tab 键开头。

如果某一行过长，可以在达到这一行行末前输入一个反斜杠(\)。由反斜杠连接起来的多行都被当作一行来处理。反斜杠与新的一行之间不要有空白。

Makefile 也可以命名为 makefile。还可以命名为其他任意合法的文件名，如果命名为其他名字，在执行 make 时，应该按照如下方式之一告诉 make 哪个是 Makefile 文件。

```
[root@mci tmp]# make -f othername
[root@mci tmp]# make --file=othername
```

如果所有文件都不需要更新，make 不会执行编译命令，它会在屏幕上输出类似以下的提示信息：

```
make: “main”是最新的。
```

通常 make 把 Makefile 文件中的第一个规则中的目标文件作为最终目标文件（又称为终极目标）。如果第一条规则中有多个目标文件（即冒号前的多个文件），make 把这些目标文件中的第一个作为最终目标文件。通常最终目标文件是最后要编译生成的可执行文件。其他规则中的目标文件都是为了产生最终目标文件而生成的中间文件。因此，在编写 Makefile 文件时，一般也把最后要生成的文件放在第一条规则的目标文件列表中。

我爱自由 5ifreedom.com

5.4.2 Makefile 文件的构成

一个完整的 Makefile 文件由 5 部分构成：显式规则、隐含规则、使用变量、文件指示和注释。

显式规则：一条显式规则指明了目标文件、目标文件的依赖文件、生成或更新目标文件所使用的命令。有些规则没有命令，这样的规则只是描述了文件之间的依赖关系。

隐含规则：由 make 根据目标文件（典型的是根据文件名的后缀）而自动推导出的规则。make 根据目标文件的文件名，自动产生目标的依赖文件和生成目标的命令。例如，在 Makefile 中有一个规则：

```
module1.o: head1.h
```

make 根据目标文件名 module1.o 的后缀.o，自动产生目标的依赖文件 module1.c 和生成目标所使用的命令 gcc -c module1.c -o module.o，以此它等价于：

```
module1.o:module1.c head1.h
        gcc -c module1.c -o module.o
```

因此上面整个 Makefile 文件可以简写为：

```
main:main.o module1.o module2.o
        gcc main.o module1.o module2.o -o main
main.o:head1.h head2.h common_head.h
module1.o:head1.h
module2.o:head2.h
```

#这是一个 Makefile 文件

使用变量：可以使用一个字符串代表一个文本串，当定义了一个变量以后，在 Makefile 被 make 解释执行时，其中的字符串都会替换为相应的文本串。Makefile 中的变量类似于 C 语言中的宏。

文件指示：它包括了 3 个部分，一是在一个 Makefile 中包含另一个 Makefile，就像 C 语言中的 include 一样；二是指根据某些情况指定 Makefile 中的有效部分，就像 C 语言中的预编译#if 一样；三是定义一个多行的命令。

注释：在 Makefile 中#字符后的内容被当作注释处理。如果某行以#字符开头，那么此行就是注释行。一般在书写 Makefile 时建议将注释作为独立的一行，而不要和 Makefile 中其他有意义的内容放在同一行。在 Makefile 中需要使用字符#时，可以用反斜线加#（即\#）来表示（对特殊字符#的转义），它表示一个字符#而不是注释的开始标志。

隐含规则、使用变量和文件指示将在后面详细讨论，本节继续介绍显式规则。

1. 显式规则

通过上面的介绍，我们已经基本认识了 Makefile 和显式规则。这里再详细介绍一下显式规则。以下是一条典型的规则：

```
foo.o : foo.c defs.h
      gcc -c -g foo.c
```

第一行中，文件 foo.o 是规则的“目标文件”，或简单地称为目标。foo.c 和 defs.h 是生成 foo.o 所要使用的文件，我们把它们称为“依赖文件”，或简单地称为依赖。规则中的第二行 gcc -c -g foo.c 是规则的“命令”。它描述了如何使用规则中的依赖文件生成目标文件。

规则告诉我们两件事。

- 如何确定目标文件是否过时（需要重新生成目标），过时是指目标文件不存在或者目标文件 foo.o 在时间戳上比依赖文件中任何一个晚。
- 如何重新生成目标文件 foo.o。规则中的命令说明了使用什么样的命令来重新生成目标。规则的命令中没有明确的使用到依赖文件 defs.h。我们假设在源文件 foo.c 中已经包含了此头文件。这也是为什么它作为依赖文件出现的原因。

我爱自由 5ifreedom.com

显式规则的一般形式是：

目标文件列表 分隔符 依赖文件列表 [；命令]

[命令]

[命令]

其中，“[ ]”中的内容是可选的。

规则中的目标文件列表可以是空格分开的多个文件名，也可以是一个标签（后面会介绍）。目标文件列表中的文件名可以使用通配符，格式“A(M)”表示库文件（Linux 下的静态库以.a 结尾）的成员“M”。通常规则只有一个目标文件（建议不要在目标文件列表中包含多个文件）。

书写规则时我们需要注意以下几点。

（1）规则的命令部分有两种书写方式：命令可以和“目标文件：依赖文件”放在同一行；使用分号（;）将命令和依赖文件列表分开。命令放在依赖文件列表的下一行，作为独立的命令行。当作为独立的命令行时必须以 Tab 键开始。在 Makefile 文件中，所有以 Tab 键开始的行都会被当作命令行来处理。

（2）Makefile 中符号“$”有特殊的含义（表示使用变量或调用函数），在规则中需要使用符号“$”的地方，需要书写两个连续的$，即“$$”。

（3）前边已提到过，对于 Makefile 中较长的行，我们可以使用反斜线“\”将其书写到几个独立的物理行上。虽然 make 对 Makefile 文件中行的最大长度是没有限制的，但还是建议这样做。不仅书写方便而且更有利于别人的阅读（这也是一个程序员修养的体现）。

规则的主要思想是：目标文件的内容由依赖文件决定，依赖文件的任何一处改动，将导致目前已经存在的目标文件过时。规则的命令为重新生成目标文件提供了方法。这些命令都是 shell 命令。

2．依赖的类型

Makefile 中有两种依赖，一种是前面介绍显式规则时提到的依赖。对于这种依赖类型，依赖文件列表中任何一个文件的时间如果比目标文件新都将导致规则中的命令被执行，从而生成新的目标文件。

还有一种依赖，如下所示：

```
foo : foo.c | somelib
    gcc -o foo foo.c somelib
```

“|”前面的文件是普通依赖文件，如果 foo.c 过时，将导致 foo 重新生成。而如果“|”后面的文件过时，foo 不会被重新生成，也就是第二行的命令不会被执行。

3．命令行属性

一条规则中可以有一个或多个命令行，每个命令行以 Tab 键开头。可以在 Tab 键后先写上一个加号+、减号-或@，然后再写上命令。如果在 Tab 键后没有，加上这 3 个符号中的一个，make 使用缺省的命令行属性：执行该命令行的命令时打印出命令，命令执行遇到错误就退出 make。

各符号的意义如下。

- -：执行本命令行的命令时如果遇到错误，继续执行而不退出 make。
- +：本行命令始终被执行，即使运行 make 命令时使用了-n、-q、-t 选项（前提是本行命令所在规则的目标文件已经过时，需要执行命令以更新目标文件）。
- @：执行本行命令时不在屏幕上打印命令的内容。

例如，在某 Makefile 文件中有如下规则：

```
file.o:file.c head1.h head2.h
    -mv file.o /tmp
    gcc -c file.c
```

这个规则的含义是：如果发现 file.c 过时，先将过时的 file.o 文件备份到/tmp 下，然后再生成新的目

我爱自由 5ifreedom.com

标文件。在第一次执行 Makefile 文件中的命令时，由于 file.o 不存在，mv 命令就会出错，默认情况下一旦出错 make 就会停止运行。但在 mv 命令前加上减号-后，遇到错误，make 忽略它而继续执行。

4．伪目标

在 Makefile 文件中，目标文件可以分为两类：实目标文件和伪目标。实目标文件是真正要生成的、以文件的形式存放在硬盘上的目标。伪目标不要求生成实际文件，而是为了让 make 执行一些辅助命令，如打印一些信息，删除无用的中间文件等。

例如，某 Makefile 文件中有以下规则：

```
clean:
-rm -f *.o
```

这里的 clean 后面没有依赖文件，规则中的命令部分也不是用来生成目标文件的，而是用来删除当前目录下所有以.o 结尾的文件。rm 前的减号- 表示，如果执行这个命令时出错，忽略错误而继续执行。

当 make 扫描到上面两行中的第一行时，发现 clean 没有依赖文件，因此总是认为它是最新的，从而不会去执行第二行的命令。为了让 make 执行“clean:”后面的语句，可以在 shell 提示符后面使用命令“make clean”。

在使用 make 编译程序时，可以先执行命令“make”以生成最终的可执行程序，但在这个过程中往往生成了很多诸如以.o 结尾的中间文件。此时如果 Makefile 文件的最后有上面那条规则，我们就可以执行命令“make clean”来删除那些不必要的中间文件。

如果在当前工作目录下存在一个名为 clean 的文件，情况就不一样了，同样我们输入“make clean”，由于这个规则没有任何依赖文件，所以目标文件（即 clean）被认为是最新而不去执行规则中的命令，因此命令“rm”将不会被执行。这并不是我们所希望的。为了解决这个问题，我们需要将目标“clean”声明为伪目标。将一个目标声明为伪目标的方法是将它作为特殊目标“.PHONY”的依赖。如下：

```
.PHONY : clean
```

这样目标“clean”就被声明为一个伪目标，无论在当前目录下是否存在“clean”这个文件，输入“make clean”之后，“rm”命令都会被执行。而且，当一个目标被声明为伪目标后，make 在执行此规则时不会试图去查找该目标的依赖文件。这样也提高了 make 的执行效率，同时也不用担心由于目标和文件名重名而不能实现我们的目的。在书写伪目标规则时，首先需要声明目标是一个伪目标，之后才是伪目标的规则定义。目标“clean”的完整书写格式应该如下：

```
.PHONY: clean
clean:
    -rm -f *.o
```

在 Makefile 中，一个伪目标可以有自己的依赖。在一个目录下如果需要生成多个可执行程序，可以在一个 Makefile 中完成。因为 Makefile 中第一个目标是最终目标，通常的做法是使用一个称为“all”的伪目标来作为最终目标，它的依赖文件就是那些需要创建的可执行程序。例如：

```
all : program1 program2 program3
.PHONY : all
program1 : program1.c utils.h
    gcc -o program1 program1.c
program2 : program2.c
    gcc -o program2 program2.c
program3 : program3.c comm.h utils.h
    gcc -o program3 program3.c
```

执行 make 时，目标“all”被作为终极目标。为了完成对它的更新，make 会生成目标“all”

我爱自由 5ifreedom.com

的所有依赖文件（program1、program2 和 program3）。当需要单独更新某一个程序时，我们可以通过 make 的命令行选项来明确指定需要生成的程序。（例如："make program1"）。

5．特殊目标

在 Makefile 中，有一些名字（通常以"."开头），当它们作为规则的目标时，具有特殊含义。常用的有：

- .PHONY：目标".PHONY"的所有依赖被作为伪目标。伪目标是这样一个目标：当使用 make 命令指定此目标时，这个目标所在规则中的命令无论目标文件是否存在都会被无条件执行。
- .IGNORE：对于目标".IGNORE"后面跟的依赖文件，生成这些依赖文件的命令在执行时如果遇到错误，make 将忽略错误继续执行。它的功能类似于命令行属性的减号"-"。当此目标没有依赖文件时，将忽略所有命令执行时的错误。
- .SUFFIXES：该目标的依赖被认为是一个后缀列表，在检查后缀规则时使用。
- .SILENT：对于该目标的依赖文件，执行生成依赖文件的命令时，make 不会打印出所执行的命令。如果".SILENT"后面没有依赖文件，则表示执行 Makefile 中的所有命令都不会打印。

例如：

```
.SILENT:module2.o
module2.o:module2.c head2.h
        gcc -c module2.c
```

module2.o 作为目标".SILIENT"的依赖文件。在执行第三行的命令时，不会打印出第三行命令。它等价于：

```
module2.o:module2.c head2.h
        @gcc -c module2.c
```

- .PRECIOUS：该目标的依赖文件会受到特别对待。如果 make 被 kill 命令终止或遇到意外而被中止，这些依赖并不会被删除，而且如果这些依赖文件是中间文件，在不需要时也不会被删除。
- .INTERMEDIATE：目标".INTERMEDIATE"的依赖文件在 make 执行时被当作中间文件对待。如果目标".INTERMEDIATE"后面没有依赖文件，那么该规则是没有意义的。

6．含有多个目标的规则

一个规则中可以有多个目标，规则所定义的命令对所有的目标都有效。一个具有多目标的规则相当于多个规则。多目标规则意味着所有的目标具有相同的依赖文件。在规则的命令中可以使用自动变量（后面会介绍），从而可以把几条类似的规则归为一条。例如：

```
module1.o module2.o module3.o: command.h
```

这个规则实现了同时给 3 个目标文件指定一个依赖文件。由于隐含规则的作用，它等价于 3 条规则：

```
module1.o:module1.c command.h
   gcc -c module1.c -o module1.o
module2.o:module2.c command.h
   gcc -c module2.c -o module3.o
module3.o:module3.c command.h
   gcc -c module3.c -o module3.o
```

7．搜索目录

通常在一个较大的项目中，会把源文件、头文件、库文件放在不同的目录里。在这种情况下，我们就要让 make 到不同的目录里去寻找依赖文件。当文件所在的目录发生变化后，可以不用更改 Makefile 中的规则，而只改变依赖文件的搜索目录。

我爱自由 5ifreedom.com

make 可以识别一个特殊变量“VPATH”。通过变量“VPATH”可以指定依赖文件的搜索目录。当规则的依赖文件在当前目录不存在时，make 会在此变量所指定的目录下去寻找这些依赖文件。“VPATH”变量所指定的是 Makefile 中所有文件的搜索路径，包括了规则的依赖文件和目标文件。

定义变量“VPATH”时，使用空格或者冒号（:）将多个需要搜索的目录分开。make 搜索目录的顺序是按照变量“VPATH”定义中的目录顺序进行的（当前目录永远是第一搜索目录）。例如对变量的定义如下：

```
VPATH = /usr/src:../headers
```

这样就为所有规则指定了两个搜索目录，“/usr/src”和“../headers”。

另一个设置文件搜索目录的方法是使用 make 的“vpath”关键字（注意，它是小写的），它不是变量，而是一个 make 的关键字，这和上面提到的那个 VPATH 变量类似，但是它更为灵活。它可以为不同的文件指定不同的搜索目录。它的使用方法有：

（1）vpath <pattern> <directories>

为符合模式<pattern>的文件指定搜索目录<directories>。

（2）vpath <pattern>

清除符合模式<pattern>的文件的搜索目录。

（3）vpath

清除所有已被设置好了的文件搜索目录。

vapth 使用方法中的<pattern>需要包含“%”字符。“%”的意思是匹配零或若干字符，例如，“%.h”表示所有以“.h”结尾的文件。<pattern>指定了要搜索的文件集，而< directories>则指定了的文件集的搜索目录。例如：

```
vpath %.h ../headers
```

该语句表示，要求 make 在“../headers”目录下搜索所有以“.h”结尾的文件（如果某文件在当前目录没有找到的话）。

可以连续地使用 vpath 语句，以指定不同搜索策略。如果连续的 vpath 语句中出现了相同的<pattern>，或是被重复了的<pattern>，那么，make 会按照 vpath 语句的先后顺序来执行搜索。例如：

```
vpath %.c src1
vpath %  src2
vpath %.c src3
```

它表示以.c 结尾的文件，先在“src1”目录找，然后是“src2”，最后是“src3”目录。

```
vpath %.h headers:src
vpath %  usr
```

而上面的语句则表示“.h”结尾的文件，先在“headers”目录，然后在“src”目录，最后到“usr”目录里去寻找。

5.4.3 使用变量

一般在具有一定规模的项目中，编写 Makefile 文件时通常会使用到变量。Makefile 中定义变量的一般形式是：

变量名 赋值符 变量值

变量名习惯上只使用字母、数字和下划线，并且不以数字开头。当然也可以是其他字符，但不能使用以下这些字符：“:”、“#”、“=”和空白符。赋值符主要有 4 个：=、:=、+=、?=。不同的赋值符有不同的意义。变量值是一个文本字符串。

在含有变量的 Makefile 中，make 执行时把变量名出现的地方用对应的变量值来替换。Makefile

我爱自由 5ifreedom.com

中的变量类似于 C 语言中的宏。Makefile 中的变量是区分大小写的，也就是说变量“var1”和“Var1”是两个不同的变量。

有一些变量是系统预定义的，如自动变量：“$@”、“$?”、“$<”、“$*”等。

1. 引用变量

当我们定义了一个变量之后，就可以在 Makefile 中使用这个变量。变量的引用方式是：$（变量名）或者${变量名}。例如：$(foo)或者${foo}就是取变量“foo”的值。符号“$”在 Makefile 中具有特殊的含义，如果要使用字符“$”，那么要用“$$”来表示。如果变量名是单个字符，那么可以直接使用“$变量名”来引用变量。

2. 定义变量

在 Makefile 中，有两种类型的变量：递归展开变量和立即展开变量。通过“=”赋值的变量是递归展开变量，通过“:=”赋值的变量是立即展开变量。

我们先来看一个例子：

```
foo = $(bar)
bar = $(ugh)
ugh = Huh
all:
    echo $(foo)
```

执行 make 将会打印出 Huh。整个变量的替换过程时这样的：在 make 执行 echo 命令时，首先$(foo)被替换为$(bar)，接下来$(bar)被替换为$(ugh)，最后$(ugh)被替换为 Huh。整个替换的过程是在执行“echo $(foo)”时完成的。

这种定义方式的好处是：在变量未定义时就可以使用该变量。例如，在“foo = $(bar)”中，提前引用了变量 bar。如果变量 bar 在整个 Makefile 中都没有定义，则$(bar)的值为空。这种定义的缺点是可能造成死循环。如：CFLAGS = $(CFLAGS) –O，导致了死循环。

使用赋值符“:=”赋值的变量是立即展开变量。例如：

```
x := foo
y := $(x) bar
x := later
```

它等价于：

```
y := foo bar
x := later
```

这种类型的变量在定义时立即展开，而不是在引用该变量时才展开。例如：

```
CFLAGS := $(include_dirs) -O
include_dirs := -Ifoo -Ibar
```

CFLAGS 的值是“-O”，而不是“-Ifoo –Ibar -O”。因为 CFLAGS 在定义时立即展开，而此时的变量 include_dirs 还未定义，那么$(include_dirs)的值为空。

Makefile 中，还有一个条件赋值符“?=”。它被称为条件赋值符是因为：只有此变量在之前没有赋值的情况下才会对这个变量进行赋值。例如：

```
FOO ?= bar
```

含义是：如果变量“FOO”在之前没有定义，就给它赋值“bar”，否则不改变它的值。

变量是如何赋值的呢？我们来看一个例子：

```
objects = main.o foo.o bar.o utils.o
```

这个语句定义了一个变量“objects”，其值为一个.o 文件的列表。变量名两边的空格和“=”之后的空格在 make 处理时被忽略。

通常，一个变量在定义之后，可以对其值进行追加，这是非常有用的。可以在定义时（也可以不定义而直接追加）给它赋一个基本值，而后根据需要可随时对其值进行追加（增加它的值）。

我爱自由 5ifreedom.com

在 Makefile 中使用“+=”（追加赋值符）来实现对一个变量值的追加操作。例如：

```
objects += another.o
```

这个操作把字符串“another.o”添加到变量“objects”原有值的末尾，并用空格和原有值分开。

3. 预定义变量

在 Makefile 中，预定义了许多变量，可以直接使用。在隐含规则中通常会使用预定义变量。

常用的预定义变量如表 5-2 所示。

表 5-2　　常用的预定义变量

宏　名	初 始 值	说　明
CC	cc	默认使用的编译器
CFLAGS	-o	编译器使用的选项
MAKE	make	make 命令
MAKEFLAGS	空	make 命令的选项
SHELL		默认使用的 Shell 类型
PWD		运行 make 命令时的当前目录
AR	ar	库管理命令
ARFLADS	-ruv	库管理命令选项
LIBSUFFIXE	.a	库的后缀
A	a	库的扩展名

例如，原有一个规则：

```
module1.o:module1.c head1.h
        gcc -c module1.c
```

改为：

```
module1.o:module1.c head1.h
        $(CC) -c module1.c
```

变量 CC 是系统预先定义的变量，可以直接使用。

现在使用默认编译器 cc 来编译 module1.c，我们可以改变预定义变量的值，假设仍然使用 gcc 来编译程序：

```
module1.o:module1.c head1.h
    $(CC) = gcc
        $(CC) -c module1.c
```

现在这个规则等价于最初的那个规则。

Makefile 还预定义了一组变量，它们的值在 make 运行过程中可以动态改变，它们是隐含规则所必需的变量，这类变量称为自动变量。常用的自动变量有：$@、$%、$<、$>、$?、$^、$+、$*。

- **$@**：表示一个规则中的目标文件名。如果目标是一个文档文件（Linux 中，一般称.a 文件为文档文件，也称为静态库文件），那么它代表这个文件名。

例如有一个规则：

```
file1.o file2.o: header.h
    cp $@  /backup
```

这条规则的功能是当目标文件过时时，将原来的目标文件备份到/backup 目录下，然后重新生成新的目标文件。当需要更新的是 file1.o 时（此时 file1.c 比 file1.o 新），$@等于 file1.o。当需要更新的是 file2.o 时（此时 file2.c 比 file2.o 新），$@等于 file2.o。$@用于指代目标文件名。

这个规则是一个多目标规则，由于隐含规则的作用，它相当于以下两条规则：

我爱自学 5ifreedom.com

```
file1.o : file1.c header.h
    cp $@  /backup
    gcc -c file1.c -o file1.o
file2.o : file2.c header.h
    cp $@  /backup
    gcc -c file2.c -o file2.o
```

- **$%**：当规则的目标文件是一个静态库文件时，$%代表静态库的一个成员名。例如，某条规则的目标是“foo.a(bar.o)”，那么，“$%”的值就为“bar.o”，而“$@”的值为“foo.a”。如果目标不是静态库文件，$%的值为空。
- **$<**：规则中的第一个依赖文件名。但如果规则中使用了隐含规则，那么$<的值是由隐含规则引入的第一个依赖文件名。例如：

```
file1.o file2.o: header.h
    cp $@  /backup
```

当 file1.o 过时时，该规则相当于：

```
file1.o : file1.c header.h
    cp $@  /backup
    gcc -c file1.c -o file1.o
```

规则中第一个依赖文件名是 file1.c，即$<的值是 file1.c，而不是 header.h。

- **$>**：它和$%一样也只适用于库文件，它的值是库名。例如，某条规则的目标是“foo.a(bar.o)”，那么，“$%”的值为“bar.o”，“$@”的值是库名即“foo.a”，“$>”的值也是库名即“foo.a”。如果目标不是静态库文件，$%的值为空。
- **$?**：所有比目标文件新的依赖文件列表，以空格分隔。如果目标是静态库文件名，代表的是库成员（.o 文件）。
- **$^**：规则的所有依赖文件列表，使用空格分隔。如果目标是静态库文件，它所代表的只是所有库成员（.o 文件）名。一个文件可重复的出现在目标的依赖中，变量“$^”只记录它的一次引用情况。也就是说变量“$^”会去掉重复的依赖文件。
- **$+**：类似“$^”，但是它保留了依赖文件中重复出现的文件。主要用在程序链接时库的交叉引用场合。
- **$***：它的值是目标文件去掉后缀后的名称。例如，如果某个规则的目标文件是 module1.o，则$*的值是去掉后缀后的名称即 module1。下面的规则中使用了$*：

```
file1.o file2.o: header.h
    gcc -c $*.c -o $@
```

注意：以上自动变量只能用在规则的命令中，不能出现在规则的目标文件列表或依赖文件列表中。如果要想让它们出现在目标文件列表或依赖文件列表中，要在它们前加上一个$。例如：$$*可以出现在目标文件列表或依赖文件列表中，它代表目标文件名，其值等同于$*。

如：

```
file1.o file2.o: $$*.c header.h
    gcc -c $*.c -o $@
```

运行 make，解释执行含有该规则的 Makefile 文件时，当处理的目标文件为 file1.o 时，$$*和$*代表 file1；当处理的目标文件为 file2.o 时，$$*和$*代表 file2。因此它等价于：

```
file1.o : $$*.c header.h
    gcc -c $*.c -o $@
file2.o : $$*.c header.h
    gcc -c $*.c -o $@
```

或者等价于：

```
file1.o : file1.c header.h
    gcc -c file1.c -o file1.o
file2.o : file2.c header.h
    gcc -c file2.c -o file2.o
```

我爱自由 5ifreedom.com

5.4.4 隐含规则

在编写过一些 Makefile 文件后就会发现，目标文件、依赖文件、编译命令之间存在一定的规律。比如，目标文件一般依赖同名的.c 源文件，而生成这种目标文件的命令则是使用 gcc 编译该.c 源文件。为了简化 Makefile 的编写，make 命令在实现时内嵌了隐含规则。在 5.4.2 节“Makefile 文件的构成”中已经简单地介绍了隐含规则。本节再来讨论一下隐含规则。

隐含规则是 make 或用户自己预先定义的一些规则。如果 Makefile 文件中没有显式的给出某个目标文件的依赖文件和编译命令，make 会根据目标文件的扩展名使用隐含规则，自动建立关于这个目标文件的依赖文件和编译所使用的命令。

隐含规则的基础是目标文件的扩展名，比如有如下一个规则：

```
program : header1.h header2.h
```

那么 make 会认为 program 是一个可执行文件，并依据隐含规则建立如下规则：

```
program : program.c header1.h header2.h
    gcc -o program program.c
```

事实上，make 处理时并不是像上面那样简单，make 不仅能识别 C 语言的源文件，它也可以识别其他语言，因此隐含规则是很复杂的。但这里我们只是让 Makefile 来自动编译 C 语言的源程序，这里只介绍涉及 C 语言的隐含规则。

与 C 语言有关的扩展名有。

- .c：C 语言源程序。
- .h：头文件。
- .o：目标文件。

这里的.o 目标文件是指使用编译器 gcc 的-c 选项生成的中间文件，不是 Makefile 里介绍的目标文件。

在 make 处理 Makefile 文件时，涉及 C 语言的隐含规则主要有两条：一条是，对于一个没有后缀的目标文件（如上面的 program），如果 Makefile 文件中没有明确指明对应的依赖文件和编译命令，并且当前目录下可以找到同名.c 源文件，那么 make 就自动为该目标文件建立一条像上面那样的规则。

另外一条是，对于一个以.o 结尾的目标文件，如果该目标文件没有依赖文件和编译命令并且可以在当前目录下找到同名.c 源文件，make 会为它建立一个规则以生成.o 目标文件。比如，对于规则：

```
program.o : header1.h header2.h
```

make 自动建立一个类似于如下方式的规则：

```
program.o : program.c header1.h header2.h
    gcc -c program.c -o program.o
```

对于上面介绍的第一条隐含规则，在 make 中它表示为：

```
$(CC)  $(CFLAGS)  $@  $<
```

CC 是 make 的预定义变量，它的默认值为 cc。CFLAGS 也是一个预定义变量，它的默认值为-o。$@指代目标文件，$<指代第一个依赖文件。

而第二条，则表示为：

```
$(CC)  -c  $<  $@
```

可以通过改变预定义变量的值来改变隐含规则。例如：

```
CC = gcc
program.o : header1.h header2.h
```

现在使用 gcc 而不是默认的编译器 cc 来编译程序了。另外可以通过直接修改整个隐含规则

我爱自由 5ifreedom.com

而不是隐含规则所使用的预定义变量来修改隐含规则。一般情况下，系统的隐含规则已经可以满足要求了，只有在一些特殊场合才需要改变隐含规则。如果要修改隐含规则请参考 make 的使用手册。

5.4.5 使用条件语句

通过在 Makefile 中使用条件语句，使得可以根据条件来指定要参与编译的部分，而忽略不符合条件的部分。如根据用户所使用的硬件平台来选择适合的库。

先来看一个使用条件语句的 Makefile 文件片断，它对变量“CC”进行判断，其值如果是“gcc”，那么在程序链接时使用库“libgnu”，否则不链接任何库（normal_libs 的值为空）。

```
...
libs_for_gcc = -lgnu
normal_libs =
...
foo: foo.c
ifeq ($(CC),gcc)
      $(CC) -o foo foo.c $(libs_for_gcc)
else
      $(CC) -o foo foo.c $(normal_libs)
endif
...
```

例子中，条件语句使用到了 3 个关键字：“ifeq”、“else”和“endif”。其中，“ifeq”表示条件语句的开始，并指定了一个比较条件（相等）。之后是用圆括号包围的，使用逗号分割的两个参数，圆括号和关键字“ifeq”用空格分开。“ifeq”之后就是当条件满足，make 需要执行的，条件不满足时忽略。“else”之后就是当条件不满足时要执行的部分，不是所有的条件语句都需要此部分。“endif”表示一个条件语句的结束，任何一个条件表达式都必须以“endif”结束。

条件的判断和解析是由 make 来完成的，上面的例子，一种更简洁的实现方式是：

```
libs_for_gcc = -lgnu
normal_libs =
ifeq ($(CC),gcc)
libs=$(libs_for_gcc)
else
libs=$(normal_libs)
endif
foo: foo.c
     $(CC) -o foo foo.c $(libs)
```

条件语句中还可以使用关键字“ifneq”，它代表的意思是“如果两个参数不相等”（if not equal）。

此外，还有两个关键字：“ifdef”和“ifndef”。

关键字“ifdef”用来判断一个变量是否已经定义。而“ifndef”用来判断一个变量是否没有被定义。其使用格式为：

```
ifdef    变量名
ifndef   变量名
```

如果变量的值非空（在 Makefile 中没有定义的变量的值为空），那么表达式为真，则将其后的语句作为 make 要执行的一部分。

示例一：

```
bar =
foo = $(bar)
ifdef foo
frobozz = yes
else
frobozz = no
endif
```

我爱自由 5ifreedom.com

示例二：

```
foo =
ifdef foo
frobozz = yes
else
frobozz = no
endif
```

示例一中的结果是“frobozz = yes”，而示例二的结果是“frobozz = no”。其原因在于：示例一中，变量“foo”的定义是“foo = $(bar)”。虽然变量“bar”的值为空，但是“ifdef”判断的结果是真。当需要判断一个变量的值是否为空时，最好使用“ifeq”（或者“ifneq”）而不是“ifdef”。

5.4.6 使用库

在大型软件开发项目中，通常把编译好的模块按照功能的不同放到不同的库中。在 Linux 中，最后链接生成可执行文件时，如果链接的是一般的.o 文件，是把整个.o 文件的内容插入到可执行文件中。而如果链接的是库，则只从库中找出程序需要的变量和函数，把它们装入到可执行文件中。使用库可以大大节省空间，所以系统提供的标准函数一般都是以库的形式提供。

在 Linux 环境下库的建立和使用请参考第 9 章。本节主要介绍在 Makefile 如何编写与库相关的规则。

库中的文件一般称为库的成员，成员的表示形式为：

```
库名(成员名)
```

其中成员名一般就是文件名。例如：

```
mylib.a (file.o)
```

表示静态库 mylib.a（以.so 结尾的是动态库）中有一个名为 file.o 的文件。静态库也称为文档文件，它是一些.o 结尾的文件，通常使用 ar 命令对它进行维护和管理。

建立和维护一个库时，将库名作为目标文件，把希望放到库中的文件作为依赖文件，其格式为：

```
库名：库名(成员 1) 库名(成员 2) …
```

或者

```
库名：库名(成员 1 成员 2 …)。
```

例如：

```
mylib.a: mylib.a(file1.o) mylib.a(file2.o) mylib.a(file3.o)
```

或者

```
mylib.a: mylib.a(file1.o file2.o file3.o)
```

接着，就可以输入命令了，一般为：

```
ar -ruv 库名 目标文件名
```

例如，下面是将 file1.o、file2.o、file3.o 三个文件加入到库 mylib 中：

```
mylib:mylib(file1.o)
    gcc -c file1.c
    ar -ruv mylib file1.o
    rm -f file1.o
mylib:mylib(file2.o)
    gcc -c file2.c
    ar -ruv mylib file2.o
    rm -f file2.o
…
```

上面规则中，没有给出库 mylib 的后缀.a，但 make 通过它的依赖文件（依赖文件放在一个圆括号中）可以识别出 mylib 是一个库而不是一个可执行文件或其他类型的目标。通过使用自动变量可以将上面的规则简化为：

```
mylib:mylib(file1.o file2.o file3.o)
    ar -ruv $@ $?
```

我爱自由 5ifreedom.com

```
    rm -f $?
file1.o:file1.c
        gcc -c file1.c
file2.o:file2.c
        gcc -c file2.c
file3.o:file3.c
        gcc -c file3.c
```

$?指代所有比目标文件新的依赖文件列表。还可以利用隐含规则简化为：

```
mylib:mylib(file1.o file2.o file3.o)
    ar -ruv $@ $?
    rm -f $?
```

make 在建立库 mylib 时，搜索它的依赖文件 file1.o、file2.o 和 file3.o，如果当前目录下没有这些依赖文件或者这些依赖文件的依赖文件（即相应的.c 源文件）更新，make 会按照隐含规则生成或重新生成库的各个依赖文件，因此可以简化为上面 3 行语句。

事实上，make 也建立了相应的针对库操作的隐含规则，隐含规则类似于：

```
$(AR) $(ARFLAGS) $@ $?
rm -f $?
```

AR、ARFLAGS 和 CC 都是 make 的预定义变量，它们的初始值分别为 ar、-ruv 和 cc。有了隐含规则，可以把上面的规则进一步简化为：

```
mylib:mylib(file1.o file2.o file3.o)
```

make 在处理这个语句时，先检查库 mylib 的各个依赖文件。从依赖文件列表中的 file1.o 开始，如果成员文件 file1.o 的建立时间晚于库的建立时间，就使用针对库操作的隐含规则更新库 mylib。如果 file1.o 的建立时间早于相应的.c 源文件，即 file1.o 过时，那么使用隐含规则生成新的 file1.o 文件，再使用针对库操作的隐含规则更新库 mylib。第一个成员文件处理完毕，接着处理 file2.o，直到处理完所有依赖文件。

5.4.7 make 命令参数详解

通过在命令行中指定 make 命令的选项，可使 make 以不同的方式运行。make 命令的主要选项参数如下所示。

- -C dir 或者 --directory=DIR。

在读取 Makefile 文件前，先切换到“dir”目录下，即把 dir 作为当前目录。如果存在多个-C 选项，make 的最终当前目录是第一个目录的相对路径，如“make -C /home/root -C src”，等价于“make -C /home/root /src”。

- -d。

make 执行时打印出所有的调试信息。包括：make 认为那些需要重新生成的文件；那些需要比较它们的最后修改时间的文件，比较的结果；重新生成目标所要执行的命令；使用的隐含规则等。

- -e 或者 --environment-overrides。

不允许在 Makefile 中对系统环境变量进行重新赋值。

- -f filename 或者 --file=FILE 或者 --makefile= FILE

使用指定文件作为 Makefile 文件。

- -i 或者 --ignore-errors。

忽略执行 Makefile 中命令时产生的错误，不退出 make。

- -h 或者 –help。

打印出帮助信息。

我爱自由 5ifreedom.com

- -k 或者 --keep-going。

执行命令遇到错误时不终止 make 的执行，make 尽最大可能执行所有的命令，直到出现致命错误才终止。

- -n 或者 --just-print 或者 --dry-run。

只打印出要执行的命令，但不执行命令。

- -o filename 或者--old-file=FILE。

指定文件“filename”不需要重建，即使相对于它的依赖已经过时，同时也不重建依赖于此文件的任何目标文件。

- -p 或者 --print-data-base。

命令执行之前，打印出 make 读取的 Makefile 的所有数据（包括规则和变量的值），同时打印出 make 的版本信息。如果只需要打印这些数据信息而不执行命令，可以使用“make -qp”命令。查看 make 执行前的隐含规则和预定义变量，可使用命令“make -p –f /dev/null”。

- -q 或者 –question。

称为“询问模式”，不执行任何命令。make 只是返回一个查询状态值，返回的状态值为 0 表示没有目标需要重建，1 表示存在需要重建的目标，2 表示有错误发生。

- -r 或者 --no-builtin-rules。

忽略隐含规则，使之不起作用。该选项不会取消 make 内嵌的预定义变量。

- -R 或者 --no-builtin-variabes。

取消 make 内嵌的预定义变量，不过我们可以在 Makefile 中明确定义某些变量。注意，-R 选项同时打开-r 选项，因为没有了预定义变量，隐含规则将失去意义（隐含规则是以内嵌的预定义变量为基础的）。

- -s 或者 –silent。

执行但不显示所执行的命令。

- -t 或者 –touch。

把所有目标文件的最后修改时间设置为当前系统时间。

- -v 或者 –version。

打印出 make 的版本信息。

关于根据程序自动生成 Makefile 文件的工具 autoconf 和 automake 的使用，请参考附录。

5.5 面试题选

结构体和共用体的的内存分配是 C 语言的一个难点，也是面试题中的热点，而大部分 C 语言方面的书籍中都忽略了这部分。本节详细讨论了各种类型的结构体和共用体的的内存分配。

示例 1：

```
union data1
{
    double      d;
    int         i;
    char        c1;
    char        c2[9];
};
```

我爱自由 5ifreedom.com

sizeof(union data1)的值为 16。在编译器默认设置的情况下，该共用体最大基本类型为 double，它占 8 字节，所以此共用体以 8 来对齐。字符数组 c2 占 9 个字节，那么整个共用体应该占 9 个字节，但按照对齐原则，实际分配给它的内存为 16 字节。

如果是：

```
struct data1
{
    double      d;
    int         i;
    char        c1;
    char        c2[9];
};
```

sizeof（struct data1）的值为 24。首先按照存储大小，该结构体所占存储空间为：8 + 4 + 1 + 9 = 22 字节，这个结构体也是以 8 对齐，因此实际分配的是 24 字节。

示例 2：

```
union data2
{
    int         i;
    char        c1;
    char        c2[9];
};
```

sizeof（union data2）的值为 12。该共用体占内存空间最大的基本数据类型为 int，其长度为 4，所以该共用体以 4 来对齐。该共用体的长度取决于字符 c2，其长度为 9，9 不是 4 的倍数，要进行对齐，因此实际分配的存储空间为 12。

```
struct data2
{
    int         i;
    char        c1;
    char        c2[9];
};
```

sizeof（struct data2）的值为 16。与上面共用体一样，该结构体以 4 对齐。按照存储大小，该结构体所占存储空间为：4 + 1 + 9 = 14，14 不是 4 的倍数，进行对齐，对齐后的值为 16。

示例 3：

```
union data3
{
char    c1;
char    c2[3];
};
```

sizeof（union data3）的值为 3。该共用体占内存空间最大的基本数据类型为 char，其长度为 1，该共用体以 1 来对齐。分配的存储空间为 3。

```
struct data3
{
char    c1;
char    c2[2];
};
```

sizeof（struct data3）的值为 3。同上面的 union data3 类似，以 1 来对齐，分配的内存空间为 1 + 2 = 3 字节。

示例 4：

```
struct inner
{
    char        c1;
    double      d;
    char        c2;
};
```

这个结构体显然是 8 字节对齐的。那么它所占的存储空间是多少呢，10 是肯定不对的，如果

我爱自由 5ifreedom.com

认为是 16，也错了。分配存储空间，编译器知道是以 8 字节对齐的，在给 c1 分配存储空间时，考虑到对齐，分配给 c1 的字节数就是 8，然后给 d 分配 8 字节，最后给 c2 分配时，因为也要以 8 对齐，所以也分配了 8 个字节的存储空间。所以 sizeof（struct inner）值为 24。

如果是：

```
struct inner
{
    char        c1;
    char        c2;
    double      d;
};
```

当然这个结构体也是以 8 字节对齐的。编译器编译程序时，给 c1、c2 分配存储空间没有必要各自给它们分配 8 字节，只要 8 字节就可以了。给 d 自然也分配 8 字节，因此 sizeof（struct inner）值为 16。

```
struct inner
{
    char        c1;
    double      d;
    char        c2;
};
union data4
{
    struct inner t1;
    int         i;
    char        c;
};
```

这个稍微有点复杂，data4 共用体中有一个 inner 结构体。看到这里最大的基本数据类型为 double，因此以 8 字节对齐。共用体 data4 的存储长度取决于 t1，因为它所占的存储空间最大，而 t1 长度为 24，因此 sizeof（union data4）的值为 24。

```
struct inner
{
    char   c1;
    double d;
    char   c2;
};
struct data4
{
    struct inner t1;
    int    i;
    char   c;
};
```

data4 结构体中有一个 inner 结构体，结构体 data4 也是以 8 对齐的，因为它的成员 t1 中最长的基本数据类型为 double，占 8 字节。t1 占 24 字节，在给 data4 中的 i 和 c 分配变量时，变量 i 需要 4 字节，变量 c 需要 1 字节，所以分配 8 字节就可以了。sizeof（struct data4）的值为 32。

示例 5：

```
struct data
{
    int         a;
    long        b;
    double      c;
    float       d;
    char        e;
    short       f;
}d;
```

这个结构体所占的字节数是多少呢？这里假定 long 所占的字节数为 4 字节，short 占 2 字节。这个结构体与示例 4 中第二个 struct inner 类似。首先这个结构体是以 8 字节对齐的，因为最长基

我爱自由 5ifreedom.com

本数据类型为 double，它占 8 字节。d、e、f 总和为 7 个字节。分配存储空间时，成员 a 和 b 各分配 4 字节，d 分配 4 字节，f 分配 2 字节，e 也分配 2 字节。d、e、f 总和刚好占 8 各字节。sizeof (struct data)值为 24。

如果是：

```
struct data
{
    int         a;
    long        b;
    double      c;
    float       d;
    char        e[3];
    short       f;
}d;
```

与上面类似，这个结构体以 8 字节对齐，a、b 各分配 4 字节。d、e、f 总和为 9 个字节，显然要进行补齐。此时内存如何分配呢？d 分配 4 字节，接下来的 e 也分配 4 个字节，剩余 f 分配 8 个字节。sizeof (struct data)值为 32。

例 5-6 下面程序输出是什么？

```
#include<stdio.h>

int main()
{
    typedef union { long i; int  k[6]; char c; } DATE;

    struct data { int cat; double dog; DATE cow; } too;

    DATE max;

    printf("%d", sizeof(struct data)+sizeof(max) );

    return 0;
}
```

程序分析。

共用体类型 DATA 中，长度最大的成员是 int k[6]，它占 24 个字节，因此 sizeof(max)为 24。struct data 长度最大的基本类型 double，占 8 字节，因此 struct data 以 8 字节对齐。DATE cow 的最大长度的基本类型是 long，一般是 4 个字节。int cat 分配的字节为 8（为何分配 8 字节而不是 4 字节，请参考示例 4 中的第一个 struct inner）。double dog 也是 8，cow 为 24，所以 sizeof(struct data) = 8 + 24 + 8 = 40。因此最后输出为 64。

例 5-7 对于一个频繁使用的短小函数，在 C 语言中最好用什么实现？

最好用宏定义。这样可以节省调用函数的开销，效率最高。

例 5-8 已知一个数组 table，写一个宏定义，求出数组的元素个数

```
#define NTBL  (sizeof(table)/sizeof(table[0]))
```

对于数组，(sizeof(table)获取数组 table 的总长度，而 sizeof(table[0])是数组第一个元素所占的长度。

例 5-9 给定结构

```
struct A
{
    unsigned short t:4;
    unsigned short k:4;
    unsigned short i:8;
    unsigned long m;
};
```

问 sizeof(A)的值。

我爱自由 5ifreedom.com

程序分析。

unsigned short 一般占 2 个字节，unsigned long 一般占 4 个字节。结构体 A 以 4 字节对齐。A 中成员 t、k、i 共占 4 + 4 + 8 = 16 位，由于要内存对齐，实际那三个成员共占 32 位即 4 字节，成员 m 占 4 字节。因此 sizeof(A) = 8。

例 5-10 求函数返回值，输入 x = 9999

```
int func ( int x )
{
    int count = 0;
    while ( x )
    {
        count ++;
        x = x&(x-1);
    }
    return count;
}
```

程序分析。

这是统计 9999 的二进制形式中有多少个 1 的函数。

9999 = 9 × 1024 + 512 + 256 + 15

9 × 1024 的二进制表示中含有 1 的个数为 2；

512 的二进制表示中含有 1 的个数为 1；

256 的二进制表示中含有 1 的个数为 1；

15 的二进制表示中含有 1 的个数为 4；

故共有 1 的个数为 8，结果为 8。

$1000_{(2)} - 1_{(2)} = 0111_{(2)}$，正好是原数取反。用这种方法来求 1 的个数是高效率的。

例 5-11 已知运行这个程序的主机中数据类型 long 占 8 字节。请分析程序的运行结果。

```
#include<stdio.h>

int main()
{
    struct data
    {
        long        l;
        char        *s;
        short int   i;
        char        c;
        short int   a[5];
    }d;

    struct data *p = &d;

    printf("%d\n",sizeof(d));
    printf("%x\t%x\n",p,p+1);
    printf("%x\t%x\n",p,(char *)p+1);
    printf("%x\t%x\n",p,(long *)p+1);

    return 0;
}
```

运行结果：

```
32
bffff60   bffff80
bffff60   bffff61
bffff60   bffff64
```

程序分析。

struct data 以 8 个字节对齐。long 类型的成员 l 分配 8 个字节。s、i、c、a 原本分别占 4、2、

我爱自由 5ifreedom.com

1、10 个字节。由于考虑到对齐，s 分配 4 个字节，i 分配 2 个字节，c 分配 2 个字节，此时刚好用完 8 个字节。a 原本分配 10 个字节，由于考虑到对齐，要使整个结构体所占的存储空间是 8 的倍数，所以分配给它 16 个字节。因此结构体 data 占 8 + 4 + 4 + 2 + 16 = 32 个字节。

第二条 printf 语句，p+1 中的加 1 并不是加 1 个字节，而是 1 个 struct data 的长度，16 进制下，bffff60 + 20（十进制数 32 以十六进制表示是 20）= bffff80。

第三条 printf 语句，p + 1 中的加 1，由于对指针 p 进行了强制类型转换，使 p 指向 char 类型的数据，此时的加 1 就是加上 1 个 char 类型的长度，因此 p + 1 的输出是 bffff61。

第四条语句的分析与第三条类似。

5.6 进一步学习建议

第 1～5 章主要介绍了 C 语言、编译器 gcc 和调试器 gdb 的使用、Makefile 的编写和 make 工具的使用。

C 语言的更多内容请参考谭浩强《C 语言程序设计》、Dennis M.Ritchie 和 Brian W.Kernighan 合著的《C 程序设计语言》、David M.Collopy 的《C 语言教程：模块化程序设计》。其中，《C 程序设计语言》的作者 Dennis M.Ritchie 不仅是 C 语言的设计者，而且也是 Unix 操作系统的开发者，这本书堪称经典的 C 语言教材。

gcc 的更多内容可以参考 gcc 使用手册。

gdb 中文使用手册。

make 中文使用手册。

作为一个软件开发人员，如果要想写出更快、更稳定、效率更高的程序，那么可以阅读卡耐基大学 Randal E. Bryant 和 David O. Hallarom 教授的《深入理解计算机系统》。它详细分析了程序是如何生成和执行的，从程序员的角度深入探究了计算机系统的内部工作原理。有了它作为基础，再学习程序设计、操作系统、数据结构、编译原理、网络，会有一种豁然开朗的感觉。

5.7 习题

1. 分别用函数和带参数的宏，实现从 3 个数中找出最大数。
2. 函数和宏各自有什么优缺点？
3. 什么时候要使用条件编译？
4. 有 10 个学生，每个学生的信息有：学号、姓名、高等数学、大学英语、计算机程序设计语言 3 门课程的成绩。从键盘输入 10 个学生的数据，要求打印出每门课程的平均分和最高分学生的信息。
5. 如何使用指针访问结构体中的成员？
6. 写一个函数，求出一个 32 位整型变量以二进制表示时 1 的个数。
7. 结构体和共用体分配存储空间时是如何进行对齐的？有什么规律？
8. 编写几个测试程序，及相应的 Makefile 文件，然后使用 make 命令进行编译。

我爱自由 5ifreedom.com

LINUX

第二篇　Linux 系统编程

文件操作

进程控制

线程控制

信号及信号处理

进程间通信

我爱自由
5ifreedom.com

第 6 章　文件操作

本章主要介绍 Linux 下的文件结构和文件操作，如创建、打开、关闭、读写、删除，以及文件的属性操作和目录操作。本章还对软件开发中的测试方法作了简单的介绍。最后是一个实现自己的 ls 命令的综合实例。

本章重点：

- 文件访问权限。
- 文件创建、打开、关闭、读写、移动和删除。
- 文件属性操作。
- 目录的创建、删除和目录信息的获取。

本章难点：

- 文件锁。
- 文件属性操作。

6.1 系统编程概述

所有操作系统都向系统中运行的程序提供服务。典型的服务有执行新程序、打开文件、读写文件、分配内存、获取当前时间等。Linux 下的系统编程是指程序员使用系统调用或 C 语言本身所携带的库函数来设计和编写具有某一特定功能的程序。shell 命令是操作系统提供给普通用户使用的接口，而系统调用是操作系统提供给程序员使用的接口，如作为系统调用提供的 open 函数用于打开一个文件。实际上，C 语言的函数库也是通过系统调用来实现的，它封装了系统调用，并在此基础上为方便程序员使用而增加了一些功能。

Linux 为上层应用程序的开发提供了丰富的系统调用，应用程序只需包含相应的头文件就可以使用这些函数。实际上，这些系统调用都是以函数库的方式提供的。编译程序时，gcc 会自动链接一些常用的库，比如 libc.so.6。对于 gcc 不会自动链接的库，则在编译程序时需要指定所使用的库（编译程序时使用-l<库名> -L<库所在的目录>选项）。

在编写程序时，有时我们可能会忘记参数的个数、参数的顺序、函数所在的头文件等信息，只记得函数的大概形式（如函数名）。这时我们可以通过在 Shell 命令行下输入 man 命令来查看函数原型。例如输入 man lseek 就可以获取 lseek 函数的原型和所属头文件。有些函数名如 mkdir 既是 Linux 的命令，也是系统调用，这时可以通过输入 man 2 mkdir 获取该函数的原型，只输入 man mkdir 得到的是命令 mkdir 的帮助信息。对于库函数，输入 man 3 <库函数名>可以获得帮助信息，例如 man 3 opendir。

6.2 Linux 的文件结构

Linux 是一个安全的操作系统，它是以文件为基础而设计的。Linux 的文件子系统主要用于管理文件

我爱自由 5ifreedom.com

存储空间的分配、文件访问权限的维护、对文件的各种操作。用户可以使用命令对文件进行操作，但在功能上受到一定的限制。程序员可以通过系统调用或C语言的库函数对文件进行操作。

文件主要包含两方面的内容：一是文件本身所包含的数据；另外就是文件的属性，也称为元数据，包括文件访问权限、所有者、文件大小、创建日期等。

目录也是一种文件，称为目录文件。目录文件的内容是该目录的目录项，目录项是该目录下的文件和目录的相关信息。当创建一个新目录时，系统将自动创建两个目录项:. 和..，在 Shell 下输入 ls-a 可以将其显示在终端上，前者代表当前目录，后者代表当前目录的父目录。对于根目录，两者是相同的。

在 Shell 下输入 cd/切换到根目录，再输入 ls 可以查看到根目录下的目录情况，一般的 Linux 发行版本都含有如下几个目录。

- /bin 用于存放普通用户可执行的命令，系统中的任何用户都可以执行该目录中的命令，如 ls、cp、mkdir 等命令。
- /boot Linux 的内核及启动系统时所需要的文件，为保证启动文件更加安全可靠，通常把该目录存放在独立的分区上。
- /dev 设备文件的存储目录，如硬盘、光驱等。
- /etc 用于存放系统的配置文件， 比如用户账号及密码存放在配置文件/etc/password 和/etc/shadow 中。
- /home 普通用户的主目录，每个用户在该目录下都有一个与用户同名的目录。
- /lib 用于存放各种库文件。
- /proc 该目录是一个虚拟文件系统，只有在系统运行时才存在。通过访问该目录下的文件，可以获取系统的状态信息并且修改某些系统的配置信息，可以简单地使用 cat、strings 命令来查看这些信息，如在 Shell 下输入 cat /proc/meminfo 可以获取系统内存的使用情况，输入 man proc 可获得关于 proc 详细的信息。
- /root 超级用户 root 的主目录。
- /sbin 存放的是用于管理系统的命令。
- /tmp 临时文件目录。
- /usr 用于存放系统应用程序及相关文件，如说明文档、帮助文件等。
- /var 用于存放系统中经常变化的文件，如日志文件，用户邮件等。

6.2.1 Linux 的文件系统模型

数据或者说文件归根结底是要存储在物理磁盘上的，操作系统通过文件系统可以方便地对磁盘上的文件进行管理。Linux 的文件系统模型如图 6-1 所示。

对物理磁盘的访问都是通过设备驱动程序来进行的，而对设备驱动的访问则有两种途径：一种是通过设备驱动本身提供的接口；另一种是通过虚拟文件系统（Virtual File System，VFS）提供给上层应用程序的接口。第一种方式能够让用户进程绕过文件系统直接读写磁盘上的内容，这给操作系统带来了很大的不稳定性，因此大部分操作系统包括 Linux 都是使用虚拟文件系统来访问设备驱动的。只有在特殊情况下才允许用户进程通过设备驱动接口直接访问物理磁盘。

VFS 是虚拟的，不存在的，它和前面提到的 proc 文件系统一样，都是只存在于内存而不存在

我爱自由 5ifreedom.com

于磁盘之中的，即只有在系统运行起来以后才存在。VFS 提供一种机制，它将各种不同的文件系统整合在一起，并提供统一的应用程序编程接口(API)供上层的应用程序使用。VFS 的使用体现了 Linux 文件系统最大的特点——支持多种不同的文件系统。Linux 不仅支持 EXT2、EXT3，也支持 windows 系统中的文件系统，如 vfat。

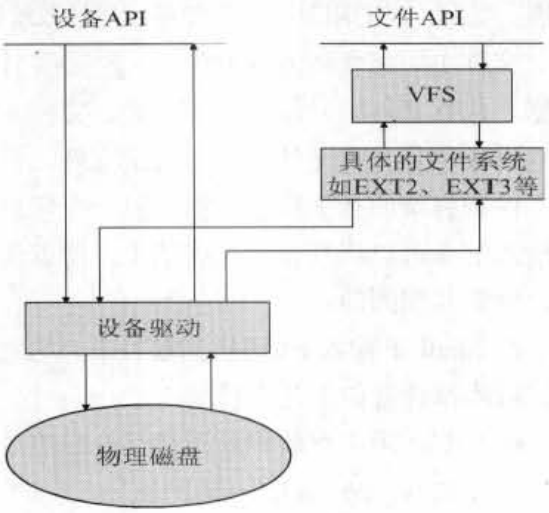


图 6-1　文件系统模型

从硬盘的构造可知，每次对物理磁盘的访问的最小单位是一个盘面上的一个磁道上的一个扇区，即使用户只需要访问 1 个字节的数据，实际读写时都是先把该字节所在的扇区读入到内存，然后再进行访问。正因为如此，文件系统是由一系列块（block）构成的，每个块的大小因不同的文件系统而不同，但是一个文件系统一旦安装之后，块的大小就固定了。通常一个块的大小是一个扇区的大小，而一个扇区通常为 512 字节。

6.2.2　文件的分类

Linux 中包含以下几种文件类型。

（1）普通文件（regular file）：这是最常见的文件类型，这种文件包含了某种形式的数据。至于这种数据是文本还是二进制数据，对于内核而言并无区别。对普通文件内容的解释由处理该文件的应用程序完成。

（2）目录文件（directory file）：目录文件就是目录，目录也有访问权限，目录文件的内容就是该目录下的文件和子目录的信息，对一个目录文件具有读许可权的任一进程都可以读该目录的内容，但只有内核可以写目录文件。

（3）字符特殊文件（character special file）：用于表示系统中字符类型的设备，比如键盘、鼠标等，这些硬件对操作系统来说只是一个文件。

（4）块特殊文件（block special file）：用于表示系统中块类型的设备，如硬盘、光驱等。对这些设备上的数据的访问通常以块的方式进行，即一次至少读写一个块。

（5）FIFO：这种类型文件用于进程间的通信，也称为命名管道。

（6）套接字（socket）：主要用于网络通信，套接字也可以用于一台主机上的进程之间的通信。第 10 章将介绍使用套接字进行网络通信。

（7）符号连接（symbolic link）：指向另一个文件，是另一文件的引用。

在 Shell 下可通过输入 ls -l <文件名> 来查看文件的类型，在程序中查看文件的类型则需要使用 stat/fstat/lstat 函数族，后面将会介绍。如在某个目录下执行 1s-1 的结果如下：

```
[root@localhost /]# ls -l /
drwxr-xr-x    2 root     root        4096 Apr 18  2006  bin
```

第一个字母 d 取自 directory 的首字母，表示 bin 是一个目录。

```
[root@localhost /]# ls -l /bin/ls
-rwxr-xr-x    1 root     root       67700 Apr 18  2006  /bin/ls
```

第一项“-”表示 bin 目录下的 ls 是一个普通文件。

```
[root@localhost /]# ls /dev/hda -l
brw-rw----    1 root     disk      3,   0 Sep 15  2003  /dev/hda
```

第一个字母 b 取自 block 的首字母，表示/dev/hda 是块特殊文件。

我爱自由 5ifreedom.com

6.2.3 文件的访问权限控制

Linux 是一个安全的操作系统，说它安全，很重要的一个原因是对用户访问权限的控制。在 Shell 下我们可以通过命令 ls -l <filename> 来查看某一文件的属性，如在某个目录下执行 ls -l 的结果如下：

```
drwxr-xr-x    2 root    root     4096 Jan 30 19:41  test
lrwxrwxrwx    1 root    root        6   Jan 30 19:41  test1 -> ./test
-rw-r--r--    1 root    root        3   Jan 30 19:30  test.c
```

上面的运行结果从左至右依次是：文件属性、文件数、所有者、拥有该文件的用户所属的组、文件大小、文件创建的时间、文件名。

第一项文件属性总共由 10 位构成，第 1 位表示文件类型。剩下的 9 位都是表示文件的访问权限，可以按照每 3 个一组分为 3 组，从左到右，第一组表示文件所有者对该文件的操作权限，第二组表示与文件所有者同组（group）的用户对该文件的操作权限，第三组表示其他用户对该文件的操作权限。可以看出，每组只可能出现 3 种字母，其说明如下。

r：可读。

w：可写。

x：可执行。

以这里的 test.c 为例，第一组为 rw-，表示 test.c 的所有者具有对该文件的读写权限，无可执行权限；第二组为 r--，表示 test.c 文件所有者所在的组对该文件具有读权限，无写、执行的权限；第三组为 r--，表示其他用户对该文件具有读权限，无写、执行的权限。

对文件访问权限的修改在 Shell 下可通过命令 chmod 来进行，如：

```
[root@localhost test]# chmod 777 test.c
```

上面的数字 7 是通过计算得出的。对于可读、可写、可执行 3 种权限，它们分别对应了一个值，r=4，w=2，x=1。4+2+1=7，777 表示的是将 test.c 的访问权限修改为：所有者、所属组和其他用户都拥有读、写、可执行 3 种权限，更详细的用法请参考 man chmod。

```
[root@localhost test]# ls -l test.c
-rwxrwxrwx    1 root    root        3 Jan 30 19:30 test.c
```

可以看到 test.c 的所有者、所属组和其他用户都拥有对 test.c 的读、写、可执行 3 种权限。

在进行程序设计时，可以通过 chmod/fchmod 函数对文件访问权限进行修改，在 Shell 下输入 man 2 chmod 可查看 chmod/fchmod 的函数原型如下：

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod(const char *path, mode_t mode);
int fchmod(int fildes, mode_t mode);
```

chmod/fchmod 的区别在于 chmod 以文件名作为第一个参数，fchmod 以文件描述符作为第一个参数。参数 mode 有如表 6-1 所示的几种组合。

表 6-1　　参数 mode 数值

字符长量值	字符常量值对应的八进制值	含　义
S_IRUSR（S_IREAD）	00400	文件所有者具有可读取权限；
S_IWUSR（S_IWRITE）	00200	文件所有者具有可写入权限；
S_IXUSR（S_IEXEC）	00100	文件所有者具有可执行权限；
S_IRGRP	00040	用户组具有可读取权限；
S_IWGRP	00020	用户组具有可写入权限；
S_IXGRP	00010	用户组具有可执行权限；

我爱自由 5ifreedom.com

续表

字符长量值	字符常量值对应的八进制值	含　义
S_IROTH	00004	其他用户具有可读取权限；
S_IWOTH	00002	其他用户具有可写入权限；
S_IXOTH	00001	其他用户具有可执行权限；
S_ISUID	04000	文件的（set user-id on execution）位；
S_ISGID	02000	文件的（set group-id on execution）位；
S_ISVTX	01000	文件的 sticky 位。

注：上面的数值数字都是八进制。

权限更改成功返回 0，失败返回-1，错误代码存于系统预定义变量 errno 中。错误代码的具体含义请参考 man 手册。从错误代码获取错误描述信息的程序代码请参考例 6-2。

利用 chmod 函数我们可以实现自己的简化版 chmod 命令。实现程序代码如下所示：

例 6-1　my_chmod.c

```
/* 功 能：改变文件访问权限 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>

int main(int argc, char ** argv)
{
    int        mode;    // 权限
    int        mode_u;    // 所有者的权限
    int        mode_g;    // 所属组的权限
    int        mode_o;    // 其他用户的权限
    char        *path;

    /*检查参数个数的合法性*/
    if(argc < 3) {
        printf("%s <mode num> <target file>\n",argv[0]);
        exit(0);
    }

    /*获取命令行参数*/
    mode = atoi(argv[1]);
    if (mode > 777 || mode < 0) {
        printf("mode num error!\n");
        exit(0);
    }
    mode_u = mode / 100;
    mode_g = (mode - (mode_u*100)) / 10;
    mode_o = mode - (mode_u*100) - (mode_g*10);
    mode = (mode_u * 8 * 8) + (mode_g * 8) + mode_o;    // 八进制转换
    path = argv[2];

    if ( chmod(path, mode) == -1) {
        perror("chmod error");
        exit(1);
    }

    return 0;
}
```

程序说明。

在上面的程序中，atoi 函数是将字符串转换成整型数，如 atoi("777")的返回值是一个整型数 777。对于 chmod 函数，第二个参数一般用上面列出来的宏之间取或运算，如 S_ISUID| S_ISGID 等，这里为了方便直接进行进制转换。我们以本节开始处的 test.c 为例来进行测试：

我爱自由 5ifreedom.com

```
[root@localhost test]# ./my_chmod 0 test.c
[root@localhost test]# ls -l test.c
----------   1 root     root          3 Jan 30 19:30 test.c
[root@localhost test]# ./my_chmod 745 test.c
[root@localhost test]# ls -l test.c
-rwxr--r-x   1 root     root          3 Jan 30 19:30 test.c
[root@localhost test]# ./my_chmod 778 test.c
mode num error!
```

从输出结果可以看出，my_chmod 能正确地对目标文件进行访问权限的修改。

6.3 文件的输入输出

本节将介绍文件的输入输出函数，如 creat、open、close、read、write 和 lseek 等。对于 C 语言标准库函数 fopen、fclose、fread、fwrite 和 fseek 等本节不做介绍，它们实际上是通过上面那些系统调用来实现的。在编写跨平台程序时，最好使用 C 语言的标准库函数以方便移植，因为 creat、open 等系统调用使用文件描述符来标识文件，而文件描述符是 UNIX/Linux 特有的，不方便移植。

对于内核而言，所有打开的文件都由文件描述符标识。文件描述符是一个非负整数。在读写一个文件前，需要先调用 open 或 creat 函数打开文件，成功执行这两个函数都将返回一个文件描述符。在对文件执行读写操作时，将其作为参数传递给 read 或 write。文件描述符的取值范围在 0～NR_OPEN 之间，Linux 中 NR_OPEN 为 255，也就是说每个程序最多只能打开 256 个文件。

文件描述符 0 代表标准输入文件，一般就是键盘；文件描述符 1 代表标准输出文件，一般是指显示器；文件描述符 2 代表标准错误输出，一般也是指显示器。

6.3.1 文件的创建、打开与关闭

1. open 函数

open 系统调用用来打开或创建一个文件，在 Shell 下输入“man open”可获取该函数原型：

```
#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);
```

其中第一个参数 pathname 是要打开或创建的含路径的文件名，第二个参数 flags 表示打开文件的方式。

- O_RDONLY：以只读方式打开文件。
- O_WRONLY：以只写方式打开文件。
- O_RDWR：以可读可写的方式打开文件。

这 3 种打开方式是互斥的，不能同时以两种或 3 种方式打开文件，但是它们可以分别与下列标志进行或运算。

- O_CREAT：若文件不存在则自动建立该文件，只有在此时，才需要用到第三个参数 mode，以说明新文件的存取权限。
- O_EXCL：如果 O_CREAT 也被设置，此指令会去检查文件是否存在。文件若不存在则创建该文件，若文件已存在则将导致打开文件出错。
- O_TRUNC：若文件存在并且以可写的方式打开时，此标志会将文件长度清为 0，即原文件中保存的数据将丢失，但文件的属性不变。

我爱自由 5ifreedom.com

- O_APPEND：所写入的数据会以追加的方式加入到文件后面。
- O_SYNC：以同步的方式打开文件，任何对文件的修改都会阻塞直到物理磁盘上的数据同步以后才返回。
- O_NOFOLLOW：如果参数 pathname 所指的文件为一符号连接，则会令打开文件失败，Linux 内核版本在 2.1.126 以上才有这个标志。
- O_DIRECTORY：如果参数 pathname 所指的文件并非为一目录，则会令打开文件失败，Linux 内核版本在 2.1.126 以上才有这个标志。
- O_NONBLOCK 或 O_NDELAY：以非阻塞的方式打开文件，对于 open 及随后的对该文件的操作，都会立即返回。

当且仅当第二参数使用了 O_CREAT 时，需要使用第三个参数 mode，以说明新文件的存取权限。必须指出，新文件的实际存取权限是 mode 与 umask 按照（mode & ～umask）运算以后的结果。例如，umask 为 045，mode 为 0740，则新建立的文件实际存取权限是 0700，即-rwx------。位运算的相关内容请参考第 5 章。

参数 mode 与 chmod 函数相同。

成功调用 open 函数会返回一个文件描述符，若有错误发生则返回-1，并把错误代码赋给 errno。详细的错误代码说明可以参考 man 手册。从错误代码获取错误描述信息的程序代码请参考例 6-2。

2. creat 函数

文件的创建可以通过 creat 系统调用来完成，在 Shell 下输入“man creat”可获取该函数原型：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int creat(const char *pathname, mode_t mode);
```

其中第一个参数 pathname 是要打开或创建的文件名，如果 pathname 指向的文件不存在，则创建一个新文件；如果 pathname 指向的文件存在，则原文件将被新文件覆盖。第二个参数 mode 与 open 函数相同。creat()相当于这样使用 open()：

```
open(const char * pathname , (O_CREAT|O_WRONLY|O_TRUNC));
```

成功调用 creat 会返回一个文件描述符，若有错误发生则会返回-1，并把错误代码赋给 errno。详细的错误代码说明可以参考 man 手册。从错误代码获取错误描述信息的程序代码请参考例 6-2。

注意： creat 只能以只写的方式打开创建的文件，creat 无法创建设备文件，设备文件的创建要使用 mknod 函数。

3. close 函数

close 系统调用用来关闭一个已经打开的文件，在 Shell 下输入“man close”可获取其函数原型如下：

```
#include <unistd.h>
   int close(int fd);
```

close 函数只有一个参数，此参数表示需要关闭的文件的文件描述符，该文件描述符是由 open 或 creat 函数得到。当 close 调用成功时，返回值为 0，发生错误时返回-1 并设置错误代码，详细的错误代码含义请参考 man 手册。

注意，close 函数调用成功时并不保证数据能全部写回硬盘。

用户程序也可以不必调用 close 函数关闭已打开的文件，因为在进程结束时，内核会自动关闭所有已打开的文件，但这不是一个良好的习惯，建议在程序中显式地调用 close 函数。

在本小节结束之前，我们利用 open 或 creat 系统调用来创建一个新文件。

例 6-2 my_create.c

```
#include <stdio.h>
#include <sys/types.h>
```

我爱自由 5ifreedom.com

```
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>

int main()
{
    int fd;

    if ((fd = open("example_62.c", O_CREAT|O_EXCL, S_IRUSR| S_IWUSR)) == -1) {
    //if ((fd = creat("example_62.c", S_IRWXU)) == -1) {
        perror("open");
        //printf("open:%s   with errno:%d\n",strerror(errno),errno);
        exit(1);
    } else {
        printf("create file success\n");
    }

    close(fd);
    return 0;
}
```

程序说明。

上面的程序使用 open 系统调用在当前目录下创建一个名为 example_62.c 的文件，且新文件的存取权限为所有者可读可写。执行结果如下：

```
[root@localhost test]# ./my_create
create file success
[root@localhost test]# ls -l example_62.c //用 ls 命令显示文件属性
-rw-------   1 root    root          0 Feb 28 21:17 example_62.c
```

可见，第一次执行该程序成功地创建了文件 example_62.c，且该文件的访问权限也符合我们的预期，那么再次执行该程序呢？结果如下：

```
[root@localhost test]# ./my_create
open: File exists
```

这是因为在调用 open 时，同时设置了 O_CREAT 和 O_EXCL 标志，则当文件存在时，open 调用失败，系统将错误代码设置成 EEXIST，表示文件已存在。

把“perror("open");”这一行注释掉，而把该行的下一行取消注释，重新编译并运行程序，可以得到如下结果：

```
[root@localhost test]# ./my_create
open: File exists   with errno:17
```

如果要从错误代码获取相应的错误描述，可以使用这种方法。使用时要注意包含头文件 errno.h。

将程序中调用 open 函数那一行注释掉，把 creat 那一行的注释取消，则第二次执行该程序不会报错，因为对于 creat 而言，对于存在的文件它用新文件将其覆盖。

6.3.2 文件的读写

1. read 函数

read 系统调用用来从打开的文件中读取数据，在 Shell 下输入“man 2 read”可获取其函数原型如下：

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

函数中各参数的含义是：从文件描述符 fd 所指向的文件中读取 count 个字节的数据到 buf 所指向的缓存中。若参数 count 为 0，则 read()不会读取数据，只返回 0。返回值表示实际读取到的字节数，如果返回 0，表示已到达文件尾或是无可读取的数据，此外文件读写指针会随读取到的字节移动。如果 read()顺利返回实际读到的字节数，最好能将返回值与参数 count 作比较，若返回的字节数比要求读取的字节数少，则有可能读到了文件尾或者 read()被信号中断了读取过程，或者是其他原因。当有错误发生时则返回-1，错误代码存入 errno 中，详细的错误代码说明

我爱自由 5ifreedom.com

请参考 man 手册。从错误代码获取错误描述信息的程序代码请参考例 6-2。

2. write 函数

write 系统调用用来将数据写入已打开的文件中，在 Shell 下输入“man 2 write”可获取该函数原型：

```
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t count);
```

函数中各参数的含义是：将 buf 所指向的缓冲区中的 count 个字节数据写入到由文件描述符 fd 所指示的文件中。当然，文件读写指针也会随之移动。如果调用成功，write()会返回写入的字节数。当有错误发生时则返回-1，错误代码存入 errno 中，详细的错误代码说明请参考 man 手册。从错误代码获取错误描述信息的程序代码请参考例 6-2。

6.3.3 文件读写指针的移动

lseek 系统调用用来移动文件读写指针的位置，在 Shell 下输入“man lseek”可获取其函数原型：

```
#include <sys/types.h>
  #include <unistd.h>
off_t lseek(int fildes, off_t offset, int whence);
```

每一个已打开的文件都有一个读写位置，当打开文件时通常读写位置是指向文件开头，若是以添加的方式打开文件（调用 open 函数时使用了 O_APPEND），则读写位置会指向文件尾。当调用 read()或 write()时，读写位置会随之增加，lseek()便用来控制该文件的读写位置。参数 fildes 为已打开的文件描述符，参数 offset 为根据参数 whence 来移动读写位置的位移数。参数 whence 有以下 3 种取值。

- SEEK_SET 从文件开始处计算偏移量，文件指针到文件开始处的距离为 offset。
- SEEK_CUR 从文件指针的当前位置开始计算偏移量，文件指针值等于当前指针值加上 offset 的值，offset 值允许取负数。
- SEEK_END 从文件结尾处开始计算偏移量，文件指针值等于当前指针值加上 offset 的值，offset 值允许取负数。

由于历史原因，新旧值之间的对应关系如表 6-2 所示。

表 6-2　　新旧值对应

旧　值	新　值
0	SEEK_SET
1	SEEK_CUR
2	SEEK_END
L_SET	SEEK_SET
L_INCR	SEEK_CUR
L_XTND	SEEK_END

lseek 允许文件指针的值设置到文件结束符（EOF）之后，但这样做并不改变文件的大小。如果使用 write 对 EOF 之后的位置写入了数据，之前的 EOF 处与后面写入的数据之间将会存在一个间隔，此时，如果使用 read 读取这个间隔的数据，得到的数据为 0。

当调用成功时返回当前的读写位置，也就是距离文件开始处多少个字节。若有错误则返回-1，errno 会存放错误代码，详细的错误代码说明请参考 man 手册。从错误代码获取错误描述信息的程序代码请参考例 6-2。

以下是 lseek 的几种常用方法。

- 将文件读写指针移动到文件开头：

我爱自由 5ifreedom.com

```
lseek (int fildes,0,SEEK_SET);
```

- 将文件读写指针移动到文件结尾：

```
lseek (int fildes,0,SEEK_END);
```

- 获取文件读写指针当前的位置（相对于文件开头的偏离）：

```
lseek (int fildes, 0,SEEK_CUR);
```

注意：有些设备（或者说设备文件）不能使用 lseek。Linux 系统不允许 lseek()对 tty 设备进行操作，此项操作会使得 lseek()返回错误代码 ESPIPE。

例 6-3 演示了文件读写和文件读写指针的移动操作过程。

例 6-3 my_rwl.c

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>

/*自定义的错误处理函数*/
void my_err(const char * err_string, int line)
{
    fprintf(stderr, "line:%d  ", line);
    perror(err_string);
    exit(1);
}

/*自定义的读数据函数*/
int my_read(int fd)
{
    int       len;
    int       ret;
    int       i;
    char    read_buf[64];

    /*获取文件长度并保持文件读写指针在文件开始处*/
    if (lseek(fd, 0, SEEK_END) == -1) {
        my_err("lseek", _LINE_);
    }
    if ((len = lseek(fd, 0, SEEK_CUR)) == -1) {
        my_err("lseek", _LINE_);
    }
    if ((lseek(fd, 0, SEEK_SET)) == -1) {
        my_err("lseek", _LINE_);
    }

    printf("len:%d\n", len);
    /*读数据*/
    if ((ret = read(fd, read_buf, len)) < 0) {
        my_err("read", _LINE_);
    }

    /*打印数据*/
    for (i=0; i<len; i++) {
        printf("%c", read_buf[i]);
    }
    printf("\n");

    return ret;
}

int main()
{
    int       fd;
    char     write_buf[32] = "Hello World!";

    /*在当前目录下创建文件 example_63.c*/
    //if ((fd = creat("example_63.c", S_IRWXU)) == -1) {
```

我爱自由 5ifreedom.com

```
    if ((fd = open("example_63.c", O_RDWR|O_CREAT|O_TRUNC, S_IRWXU)) == -1) {
        my_err("open", _LINE_);
    } else {
        printf("create file success\n");
    }

    /*写数据*/
    if (write(fd, write_buf, strlen(write_buf)) != strlen(write_buf)) {
        my_err("write", _LINE_);
    }
    my_read(fd);

    /*演示文件的间隔*/
    printf("/*---------------------*/\n");
    if (lseek(fd, 10, SEEK_END) == -1) {
        my_err("lseek", _LINE_);
    }
    if (write(fd, write_buf, strlen(write_buf)) != strlen(write_buf)) {
        my_err("write", _LINE_);
    }
    my_read(fd);

    close(fd);
    return 0;
}
```

程序说明。

程序中首先在当前目录下创建了文件 example_63.c，注意 open 的参数，O_RDWR|O_CREAT|O_TRUNC 表示以可读可写方式创建一个文件，若该文件名存在，则覆盖，然后使用 write 系统调用，向新创建的文件内写入数据。函数 my_read 先测试由参数 fd 传入的文件描述符对应的文件的长度，然后读出全部数据，最后打印出来。

_LINE_是预编译器内置宏，表示行数，类似的宏还有_TIME_、_FUNCTION_、_FILE_等，分别表示时间、函数名、文件名，程序调试时在适当的位置加入这些提示，对定位错误很有帮助。

程序执行完后查看文件 example_63.c 的内容如下：

```
[root@localhost test]# od -c example_63.c
0000000   H   e   l   l   o       W   o   r   l   d   !  \0  \0  \0  \0
0000020  \0  \0  \0  \0  \0  \0   H       e   l   l   o       W   o   r   l
0000040   d   !
0000042
```

从结果可以看出，因为 lseek 移动到超出 EOF，在两个“Hello World!”之间多出了以“\0”填充的间隔。

程序中创建文件时使用的是 open 系统调用，读者可以将 open 系统调用那一行注释掉，然后将 creat 那一行的注释取消，再执行程序并比较一下结果，想一想为什么。

6.3.4 dup、dup2、fcntl、ioctl 系统调用

1. dup 和 dup2 函数

dup 和 dup2 系统调用都可以用来复制文件描述符，在 Shell 下输入“man dup”可获取它们的函数原型如下：

```
#include <unistd.h>
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

dup 用来复制参数 oldfd 所指的文件描述符。当复制成功时，返回最小的尚未被使用的文件描述符。若有错误则返回-1，错误代码存入 errno 中，详细的错误代码说明请参考 man 手册。返回的新文件描述符和参数 oldfd 指向同一个文件，共享所有的锁定、读写指针和各项权限或标志位。例如，当利用 lseek()对某个文件描述符操作时，另一个文件描述符的读写位置也会随着改变。

我爱自由 5ifreedom.com

dup2 与 dup 的区别是 dup2 可以用参数 newfd 指定新文件描述符的数值。若 newfd 已经被程序使用，系统就会将其关闭以释放该文件描述符；若 oldfd 与 newfd 相等，则 dup2 返回 newfd，而不关闭它。dup2 调用成功，返回新的描述符，出错返回-1。

Shell 中的重定向功能（输入重定向“<”和输出重定向“>”）就是通过 dup 或 dup2 函数对标准输入和标准输出的操作来实现的，限于篇幅在此不列出程序，感兴趣的读者可以自己动手试一试或者参考第 7 章。

2．fcntl 函数

fcntl 系统调用可以用来对已打开的文件描述符进行各种控制操作以改变已打开文件的各种属性，在 Shell 下输入“man fcntl”可获取其函数原型如下：

```
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
int fcntl(int fd, int cmd, struct flock *lock);
```

fcntl 的功能依据 cmd 值的不同而不同，具体有以下几种功能。

（1）F_DUPFD

此时，fcntl 的功能与 dup 一样，可以复制由 fd 指向的文件描述符。调用成功返回新的文件描述符，失败返回-1，错误代码存入 errno 中，详细的错误代码说明请参考 man 手册。从错误代码获取错误描述信息的程序代码请参考例 6-2，下同。

（2）F_GETFD

此时，fcntl 用来获取文件描述符的 close-on-exec 标志。调用成功返回标志值，若此标志值的最后一位是 0，则该标志没有设置，即意味着在执行 exec 相关函数后文件描述符仍保持打开。否则在执行 exec 相关函数时将关闭该文件描述符。失败返回-1。

（3）F_SETFD

此时，fcntl 用来设置文件描述符的 close-on-exec 标志为第三个参数 arg 的最后一位。成功返回 0，失败返回-1。

（4）F_GETFL

此时，fcntl 用来获得文件打开的方式。成功返回标志值，失败返回-1。标志值的含义同 open 系统调用一致。

（5）F_SETFL

此时，fcntl 用来设置文件打开的方式为第三个参数 arg 指定的方式。但是 Linux 系统只能设置 O_APPEND、O_NONBLOCK、O_ASYNC 标志，它们的含义也和 open 系统调用一致。

例 6-4　fcntl_access.c

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

/*自定义的错误处理函数*/
void my_err(const char * err_string, int line)
{
    fprintf(stderr, "line:%d  ", line);
    perror(err_string);
    exit(1);
}

int main()
```

我爱自由 5ifreedom.com

```
{
    int         ret;
    int         access_mode;
    int         fd;

    if ((fd = open("example_64", O_CREAT|O_TRUNC|O_RDWR, S_IRWXU)) == -1) {
        my_err("open", __LINE__);
    }

    /*设置文件打开方式*/
    if ((ret = fcntl(fd, F_SETFL, O_APPEND)) < 0) {
        my_err("fcntl", __LINE__);
    }

    /*获取文件打开方式*/
    if ((ret = fcntl(fd, F_GETFL, 0)) < 0) {
        my_err("fcntl", __LINE__);
    }
    access_mode = ret & O_ACCMODE;
    if (access_mode == O_RDONLY) {
        printf("example_64 access mode: read only");
    } else if (access_mode == O_WRONLY) {
        printf("example_64 access mode: write only");
    } else if (access_mode == O_RDWR) {
        printf("example_64 access mode: read + write");
    }

    if (ret & O_APPEND) {
        printf(" ,append");
    }
    if (ret & O_NONBLOCK) {
        printf(" ,nonblock");
    }
    if (ret & O_SYNC) {
        printf(", sync");
    }
    printf("\n");

    return 0;
}
```

程序说明。

程序中的 O_ACCMODE 是取得文件打开方式的掩码，实际上它的值就是 3，做与运算只是为了取得 ret 的最后两位的值。

接下来的 fcntl 函数 3 种功能都和文件记录锁有关，因此先介绍一下文件记录锁。

当有多个进程同时对某一文件进行操作时，就有可能发生数据的不同步，从而引起错误，该文件的最后状态取决于写该文件的最后一个程序。但是对于有些应用程序，如数据库，有时进程需要确保它正在单独写一个文件。为了向进程提供这种功能，Linux 系统提供了记录锁机制。

Linux 的文件记录锁能提供非常详细的控制，它能对文件的某一区域进行文件记录锁的控制。当 fcntl 用于管理文件记录锁的操作时，第三个参数指向一个 struct flock *lock 的结构：

```
struct flock {
    short_l_type;       /* 锁的类型*/
    short_l_whence;      /* 偏移量的起始位置:SEEK_SET, SEEK_CUR, SEEK_END */
    off_t_l_start;        /* Starting offset for lock */
    off_t_l_len;         /* Number of bytes to lock */
    pid_t_l_pid;         /* 锁的属主进程 ID */
};
```

l_type 用来指定是设置共享锁（F_RDLCK，读锁）还是互斥锁（F_WDLCK，写锁）。多个进程在一个给定的字节上可以有一把共享的读锁，但是在一个给定字节上的写锁则只能由一个进程单独使用。进一步而言，如果在一个给定字节上已经有一把或多把读锁，则不能在该字节上再加写锁；如果在一个字节上已经有一把独占性的写锁，则不能再对它加任何读锁（锁的不兼容规则）。

我爱自由 5ifreedom.com

一个进程只能设置某一文件区域上的一种锁。如果某一文件区域已经存在文件记录锁了，则如果此时再设置新的锁在该区域的话，旧的锁将会被新的锁取代。

l_whence、l_start 和 l_len 用来确定需要进行文件记录锁操作的区域，含义和 lseek 一节所讲解的一致。如 l_len 为 0，则表示锁的区域从其起点（由 l_start 和 l_whence 决定）开始直至最大可能位置为止。也就是不管添加到该文件中多少数据，它都处于锁的范围。为了锁整个文件，通常的方法是将 l_start 说明为 0，l_whence 说明为 SEEK_SET，l_len 说明为 0。

（6）F_SETLK

此时，fcntl 系统调用被用来设置或释放锁，当 l_type 取 F_RDLCK 或 F_WDLCK 时，在由 l_whence、l_start 和 l_len 指定的区域上设置锁；当 l_type 取 F_UNLCK 时则释放锁。如果锁被其他进程占用，则返回-1 并设置 errno 为 EACCES 或 EAGAIN。

需要注意的是，当设置一个共享锁（读锁）时，第一个参数 fd 所指向的文件必须以可读方式打开；当设置一个互斥锁（写锁）时，第一个参数 fd 所指向的文件必须以可写方式打开；当设置两种锁时，第一个参数 fd 所指向的文件必须以可读可写方式打开。当进程结束或文件描述符 fd 被 close 系统调用时，锁会自动释放。

（7）F_SETLKW

此时，fcntl 的功能与 cmd 取 F_SETLK 时类似，不同的是当希望设置的锁因为存在其他锁而被阻止设置时，该命令会等待相冲突的锁被释放。

（8）F_GETLK

此时，第 3 个参数 lock 指向一个希望设置的锁的属性的结构，如果锁能被设置，该命令并不真的设置锁，而是只修改 lock 的 l_type 域为 F_UNLCK，然后返回。如果存在一个或多个锁与希望设置的锁互相冲突，则 fcntl 返回其中的一个锁的 flock 结构。

cmd 取（6）、（7）、（8）时，执行成功返回 0，当有错误发生时则返回-1，错误代码存入 errno 中，详细的错误代码说明请参考 man 手册。

Linux 系统的文件记录锁默认是建议性的。我们考虑数据库存取例程库，如果数据库中所有函数都以一致的方法处理记录锁。则称使用这些函数存取数据库的任何进程集为合作进程（cooperating process）。如果这些函数是惟一的用来存取数据库的函数，那么它们使用建议性锁是可行的。但是建议性锁并不能阻止对数据库文件有写许可权的任何其他进程写数据库文件。不使用协同一致的方法（数据库存取例程库）来存取数据库的进程是一个非合作进程。强制性锁机制中，内核对每一个 open、read 和 write 都要检查调用进程对正在存取的文件是否违背了某一把锁的作用。

例 6-5 是对锁的应用实例程序，具体应用见程序中的代码注释。

例 6-5　fcntl_lock.c

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>

/*自定义的错误处理函数*/
void my_err(const char * err_string, int line)
{
    fprintf(stderr, "line:%d  ", line);
    perror(err_string);
    exit(1);
}
```

我爱自由 5ifreedom.com

```
/*锁的设置或释放函数*/
int lock_set(int fd, struct flock * lock)
{
    if (fcntl(fd, F_SETLK, lock) == 0) { // 执行成功
        if (lock->l_type == F_RDLCK) {
            printf("set read lock,pid:%d\n", getpid());
        } else if (lock->l_type == F_WRLCK) {
            printf("set write lock,pid:%d\n", getpid());
        } else if (lock->l_type == F_UNLCK) {
            printf("release lock,pid:%d\n", getpid());
        }
    } else {                        // 执行失败，返回-1
        perror("lock operation fail\n");
        return -1;
    }

    return 0;
}

/*测试锁,只有当测试发现参数 lock 指定的锁能被设置时，返回 0*/
int lock_test(int fd, struct flock * lock)
{
    if (fcntl(fd, F_GETLK, lock) == 0) {    // 执行成功
        if (lock->l_type == F_UNLCK) {    // 测试发现能按参数 lock 要求设置锁
            printf("lock can be set in fd\n");
            return 0;
        } else {    // 有不兼容的锁存在，打印出设置该锁的进程 ID
            if (lock->l_type == F_RDLCK) {
                printf("can't set lock, read lock has been set by:%d\n", lock->l_pid);
            } else if (lock->l_type == F_WRLCK) {
                printf("can't set lock, write lock has been set by:%d\n", lock->l_pid);
            }
            return -2;
        }
    } else {                            // 执行失败，返回-1
        perror("get incompatible locks fail");
        return -1;
    }
}

int main()
{
    int             fd;
    int             ret;
    struct flock    lock;
    char            read_buf[32];

    /*打开或创建文件文件*/
    if ((fd = open("example_65", O_CREAT|O_TRUNC|O_RDWR, S_IRWXU)) == -1) {
        my_err("open", __LINE__);
    }
    if (write(fd, "test lock", 10) != 10) {
        my_err("write", __LINE__);
    }
    /*初始化 lock 结构*/
    memset(&lock, 0, sizeof (struct flock));
    lock.l_start = SEEK_SET;
    lock.l_whence = 0;
    lock.l_len = 0;

    /*设置读锁*/
    lock.l_type = F_RDLCK;
    if (lock_test(fd, &lock) == 0) { // 测试可以设置锁
        lock.l_type = F_RDLCK;
        lock_set(fd, &lock);
    }

    /*读数据*/
```

我爱自由 5ifreedom.com

```
    lseek(fd, 0, SEEK_SET);
    if ((ret = read(fd, read_buf, 10)) < 0) {
        my_err("read", __LINE__);
    }
    read_buf[ret] = '\0';
    printf("%s\n", read_buf);

    /*等待任意按键*/
    getchar();

    /*设置写锁*/
    lock.l_type = F_WRLCK;
    if (lock_test(fd, &lock) == 0) { // 测试可以设置锁
        lock.l_type = F_WRLCK;
        lock_set(fd, &lock);
    }

    /*释放锁*/
    lock.l_type = F_UNLCK;
    lock_set(fd, &lock);
    close(fd);
    return 0;
}
```

程序说明。

本程序将锁的设置与测试都写成单独的子函数，程序首先在当前目录下建立了文件 example_65，并写入了一个字符串 test lock 到 example_65 文件中，然后对其进行文件记录锁的操作。

执行结果如下：

```
[root@localhost test]# ./fcntl_lock
lock can be set in fd
set read lock,pid:16681
test lock

lock can be set in fd
set write lock,pid:16681
release lock,pid:16681
```

从结果可以看出，程序首先测试能不能在文件上加上读锁，测试结果表明可以在指定文件上设置读锁，于是在文件上设置了读锁；然后从文件中进行读操作，并将读出来的数据打印出来；再测试能不能在文件上加上写锁，测试结果表明可以在指定文件上设置写锁，于是又在文件上设置了写锁。

读者可能要问：前面不是介绍在某一字节上设置了读锁后就不能再设置写锁吗？为什么这里在设置了读锁后又成功地设置了写锁呢？这是因为本程序只是一个进程，单个进程在同一字节上只能设置一种锁，新的锁会取代旧的锁。锁的不兼容规则是针对于多个进程之间的。

为了演示锁的不兼容规则，仍旧用上面的程序，但是程序分别在不同的终端执行。

先在某一终端执行程序（假设为进程 1），结果如下：

```
[root@localhost test]# ./fcntl_lock
lock can be set in fd
set read lock,pid:16700
test lock
```

这时不进行任何按键操作，程序一直阻塞着等待用户的按键输入。再打开一个新的终端执行程序（假设为进程 2），结果如下：

```
[root@localhost test]# ./fcntl_lock
lock can be set in fd
set read lock,pid:16701
test lock
```

从进程 2 的结果可以看出，虽然进程 1 已在指定文件上设置了读锁，但是进程 2 仍旧能从文件上正确地读出数据，可见 Linux 的文件记录锁默认是建议性锁而不是强制性锁。

切换到第二个终端，输入任意按键，结果如下：

我爱自由 5ifreedom.com

```
can't set lock, read lock has been set by:16700
release lock,pid:16701
```

由结果可见，由于进程 1 在文件上设置了读锁，按照锁的不兼容规则，进程 2 只设置成功了读锁，而不能设置写锁了。

（9）F_GETOWN

此时，返回当前接收 SIGIO 或 SIGURG 信号的进程 ID 或进程组，进程组 ID 以负值返回。

（10）F_SETOWN

此时，设置进程或进程组接收 SIGIO 和 SIGURG 信号，进程组 ID 以负值指定，进程 ID 用正值指定。

（11）F_GETSIG

此时，可以在输入输出时，获得发送的信号。

（12）F_SETSIG

此时，设置在输入输出时发送的信号。

注意：进程和信号相关内容在第 7、8 章中介绍。可以在学习完有关“后门”的章节之后再来回顾本部分内容。

3. ioctl 函数

ioctl 系统调用通常用来控制设备，不能用本章其他函数进行的控制操作都可以用 ioctl 来进行，在 Shell 下输入“man ioctl”可获取其函数原型如下：

```
#include <sys/ioctl.h>
int ioctl(int fd, int request, ...);
```

ioctl 用来控制特殊设备文件的属性，第一个参数 fd 必须是一个已经打开的文件描述符，第三个参数一般为 char *argp，它随第二个参数 request 的不同而不同。参数 request 决定了参数 argp 是向 ioctl 传递数据还是从 ioctl 获取数据。

例 6-6 采用 ioctl 获取网络设备的信息，在编写网络相关程序时可能会用到。读者在学习了第 10 章网络编程后就可以完全理解本程序了。

例 6-6 ioctl_net.c

```
// 示例 ioctl 的使用
// 本程序修改自网络上的程序，版权归原作者所有
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <net/if.h>

unsigned char    g_eth_name[16];
unsigned char    g_macaddr[6];
unsigned int     g_subnetmask;
unsigned int     g_ipaddr;
unsigned int     g_broadcast_ipaddr;

/*初始化网络，获取当前网络设备的信息*/
void init_net(void)
{
    int i;
    int sock;
    struct sockaddr_in sin;
    struct ifreq ifr;

    sock = socket(AF_INET, SOCK_DGRAM, 0);
    if (sock == -1) {
        perror("socket");
    }
```

我爱自由 5ifreedom.com

```
    strcpy(g_eth_name, "eth0");
    strcpy(ifr.ifr_name, g_eth_name);
    printf("eth name:\t%s\n", g_eth_name);


    // 获取并打印网卡地址
    if (ioctl(sock, SIOCGIFHWADDR, &ifr) < 0) {
        perror("ioctl");
    }
    memcpy(g_macaddr, ifr.ifr_hwaddr.sa_data, 6);

    printf("local mac:\t");
    for(i=0;i<5;i++) {
        printf("%.2x:", g_macaddr[i]);
    }
    printf("%.2x\n",g_macaddr[i]);

    // 获取并打印 IP 地址
    if (ioctl(sock, SIOCGIFADDR, &ifr) < 0) {
        perror("ioctl");
    }
    memcpy(&sin, &ifr.ifr_addr, sizeof(sin));
    g_ipaddr = sin.sin_addr.s_addr;
    printf("local eth0:\t%s\n", inet_ntoa(sin.sin_addr));

    // 获取并打印广播地址
    if (ioctl(sock, SIOCGIFBRDADDR, &ifr) < 0) {
        perror("ioctl");
    }
    memcpy(&sin, &ifr.ifr_addr, sizeof(sin));
    g_broadcast_ipaddr = sin.sin_addr.s_addr;
    printf("broadcast:\t%s\n", inet_ntoa(sin.sin_addr));

    // 获取并打印子网掩码
    if (ioctl(sock, SIOCGIFNETMASK, &ifr) < 0) {
        perror("ioctl");
    }
    memcpy(&sin, &ifr.ifr_addr, sizeof(sin));
    g_subnetmask = sin.sin_addr.s_addr;
    printf("subnetmask:\t%s\n", inet_ntoa(sin.sin_addr));

    close(sock);
}

int main()
{
    /*initialize...*/
    init_net();

    /*do something*/

    return 0;
}
```

程序说明。

程序先创建了一个用于网络通信的套接字，然后利用 ioctl 对其操作，获取网络信息。程序中的函数 inet_ntoa 用来将网络地址转换成字符串形式。本程序在计算机上执行的结果如下：

```
[root@localhost test]# ./ioctl_net
eth name:          eth0
local mac:         00:0d:87:9f:47:b8
local eth0:         172.17.242.131
broadcast:         172.17.242.255
subnetmask:        255.255.255.0
```

我爱自由 5ifreedom.com

6.4 文件属性操作

在 Linux 系统中，每个文件、目录都属于某一个用户，而用户又属于某一个组。每个用户有一个对应的 uid，uid 为 0 表示该用户具有 root 权限。默认情况下，uid 与用户名是一一对应的，但实际上也可以多个用户名共享一个 uid。组名和组 id 与此类似，每个用户都属于某个用户组，一个组中可以有多个用户。一个用户还可以属于不同的组。可以通过查看/etc/passwd 和/etc/group 这两个文件的内容了解系统的用户、组的配置情况。

Linux 系统中文件的属性在 Shell 下可通过输入命令“ls -l <filename>”来查看，以下是在某个目录下执行“ls -l”的结果：

```
drwxr-xr-x    2 root     root     4096    Jan 30 19:41  test
lrwxrwxrwx    1 root     root        6    Jan 30 19:41  test1 -> ./test
-rw-r--r--    1 root     root        3    Jan 30 19:30  test.c
```

上面的运行结果从左至右依次是：文件属性、文件数、所有者、文件所有者所属的组、文件大小、文件创建的时间、文件名。

第一项文件的存取权限已在 6.2.3 小节介绍过，在此不再赘述。

第二项表示文件个数。对于文件，这一项的值是 1；对于目录，这一项的值就是该目录中的目录文件个数。上面的例子中 test 是个空目录，因此只有系统默认的两个目录项：.（当前目录）和..（当前目录的父目录）。

第三项表示该文件或目录的所有者。

第四项表示文件所有者所属的组（group）。

第五项表示文件的大小，单位默认是 bytes（字节）。

第六项表示文件最后一次被修改的时间。以“月，日，时间”的格式表示，如 Jan 30 19:41 表示 1 月 30 日 19:41 分。

最后一项表示文件名，对于 test1 -> ./test，表示该项的文件名是 test1，它是一个指向目录./test 的符号链接。

6.4.1 获取文件属性

在 Shell 下直接使用命令 ls 就可获取文件的属性，那么在程序中又是通过什么方法获取文件属性的呢？这就需要用到 stat/fstat/lstat 函数了。这 3 个函数的功能类似，它们的函数原型可以通过在 Shell 下输入命令“man 2 stat”获得：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int stat(const char *file_name, struct stat *buf);
int fstat(int filedes, struct stat *buf);
int lstat(const char *file_name, struct stat *buf);
```

这些函数执行成功都返回 0，当有错误发生时则返回-1，错误代码存入 errno 中，详细的错误代码说明请参考 man 手册。

这 3 个函数的区别是：stat 用于获取由参数 file_name 指定的文件名的状态信息，保存在参数 struct stat *buf 中。fstat 与 stat 的区别在于 fstat 是通过文件描述符来指定文件的。lstat 与 stat 的区别在于，对于符号链接文件，lstat 返回的是符号链接文件本身的状态信息，而 stat 返回的是符号

我爱自由 5ifreedom.com

链接指向的文件状态信息。

参数 struct stat *buf 是一个保存文件状态信息的结构体，其类型如下：

```
struct stat {  /* 各字段的具体含义见下面的注释 */
    dev_t          st_dev;
    ino_t          st_ino;
    mode_t         st_mode;
    nlink_t        st_nlink;
    uid_t          st_uid;
    gid_t          st_gid;
    dev_t          st_rdev;
    off_t          st_size;
    blksize_t      st_blksize;
    blkcnt_t       st_blocks;
    time_t         st_atime;
    time_t         st_mtime;
    time_t         st_ctime;
};
```

其中各个域的含义如下。

- st_dev：文件的设备编号。
- st_ino：文件的 i-node（i 节点编号）。
- st_mode：文件的类型和存取权限，它的含义与 chmod、open 函数的 mode 参数相同。
- st_nlink：连到该文件的硬链接数目，刚建立的文件值为 1。
- st_uid：文件所有者的用户 id。
- st_gid：文件所有者的组 id。
- st_rdev：若此文件为设备文件，则为其设备编号。
- st_size：文件大小，以字节计算，对符号链接，该大小是其所指向的文件名的长度。
- st_blksize：文件系统的 I/O 缓冲区大小。
- st_blcoks：占用文件区块的个数，每一区块大小通常为 512 个字节。
- st_atime：文件最近一次被访问的时间。
- st_mtime：文件最后一次被修改的时间，一般只能调用 utime 和 write 函数时才会改变。
- st_ctime：文件最近一次被更改的时间，此参数在文件所有者、所属组、文件权限被更改时更新。

若某一目录具有 sticky 位（S_ISVTX），则表示在此目录下的文件只能被该文件所有者、此目录所有者或 root 来删除或改名。

对于 st_mode 包含的文件类型信息，POSIX 标准定义了一系列的宏。

- S_ISLNK（st_mode）：判断是否为符号链接。
- S_ISREG（st_mode）：判断是否为一般文件。
- S_ISDIR（st_mode）：判断是否为目录文件。
- S_ISCHR（st_mode）：判断是否为字符设备文件。
- S_ISBLK（st_mode）：判断是否为块设备文件。
- S_ISFIFO（st_mode）：判断是否为先进先出 FIFO。
- S_ISSOCK（st_mode）：判断是否为 socket。

struct stat 结构体的成员比较多，但有的很少使用。常用的有：st_mode、st_uid、st_gid、st_size、st_atime、st_mtime。

程序 6-7 是一个获取文件属性的程序。

我爱自由 5ifreedom.com

例 6-7　my_chmod.c

```
#include <stdio.h>
#include <time.h>
#include <sys/stat.h>
#include <unistd.h>
#include <sys/types.h>
#include <errno.h>

int main(int argc, char *argv[])
{
    struct stat buf;
    // 检查参数个数
    if (argc != 2) {
        printf("Usage: my_stat <filename>\n");
        exit(0);
    }
    // 获取文件属性
    if ( stat(argv[1], &buf) == -1 ) {
        perror("stat:");
        exit(1);
    }

    // 打印出文件属性
    printf("device is: %d\n", buf.st_dev);
    printf("inode is: %d\n", buf.st_ino);
    printf("mode is: %o\n", buf.st_mode);
    printf("number of hard links  is: %d\n", buf.st_nlink);
    printf("user ID of owner is: %d\n", buf.st_uid);
    printf("group ID of owner is: %d\n", buf.st_gid);
    printf("device type (if inode device) is: %d\n", buf.st_rdev);

    printf("total size, in bytes is: %d\n", buf.st_size);
    printf(" blocksize for filesystem I/O is: %d\n", buf.st_blksize);
    printf("number of blocks allocated is: %d\n", buf.st_blocks);

    printf("time of last access is: %s", ctime(&buf.st_atime));
    printf("time of last modification is: %s", ctime(&buf.st_mtime));
    printf("time of last change is: %s", ctime(&buf.st_ctime));

    return 0;
}
```

程序说明。

整个程序的结构很简单，先检查参数个数是否为 2，如果不是则退出。然后根据输入的文件名调用 stat 函数获取文件属性，如果该函数返回-1（有可能文件不存在），则退出。最后将获取到的文件属性信息打印出来。

这个程序没有进一步对 st_mode 中包含的信息进行解析，感兴趣的读者可以自己动手把文件类型和文件存取权限的信息解析出来或者参考程序 6-13。

6.4.2　设置文件属性

对文件属性的修改，在 Shell 下可以使用一些命令来完成，如 chmod。在程序中同样可以通过一系列的函数来修改文件的属性，如 chmod/fchmod，chown/fchown/lchown，truncate/ ftruncate，utime，umask 等。

1．hmod/fchmod

用于修改文件的存取权限，详见 6.2.3 小节。

2．chown/fchown/lchown

用于修改文件的用户 id 和组 id，在 Shell 下输入“man 2 chown”可获取其函数原型如下：

```
#include <sys/types.h>
    #include <unistd.h>
int chown(const char *path, uid_t owner, gid_t group);
```

我爱自由 5ifreedom.com

```
int fchown(int fd, uid_t owner, gid_t group);
int lchown(const char *path, uid_t owner, gid_t group);
```

chown 会将参数 path 指定的文件所有者 id 变更为参数 owner 代表的用户 id，而将该文件所有者的组 id 变更为参数 group 组 id。fchown 与 chown 类似，只不过它是以文件描述符作为参数的。除了所引用的文件是符号链接以外，lchown 与 chown 功能一样。在某个文件是符号链接的情况下，lchown 更改符号链接本身的所有者 id，而不是该符号链接所指向的文件。

文件的所有者只能改变文件的组 id 为其所属组中的一个，超级用户才能修改文件的所有者 id，并且超级用户可以任意修改文件的用户组 id。如果参数 owner 或 group 指定为-1，那么文件的用户 id 和组 id 不会被改变。

函数执行成功返回 0，当有错误发生时则返回-1，错误代码存入 errno 中，详细的错误代码说明请参考 man 手册。

利用这几个函数可以实现自己的 chown 命令，感兴趣的读者可以自己动手试一试。

注意：chown 系统调用会清除 set-uid 位和 set-gid 位（清除 S_ISUID 和 S_ISGID）。

3. truncate/ftruncate

用于改变文件的大小，在 Shell 下输入“man truncate”可获取其函数原型如下：

```
#include <unistd.h>
#include <sys/types.h>
int truncate(const char *path, off_t length);
int ftruncate(int fd, off_t length);
```

truncate 将参数 path 指定的文件大小改为参数 length 指定的大小。如果原来的文件大小比参数 length 大，则超过的部分会被删除；如果原来的文件大小比参数 length 小，则文件将被扩展，与 lseek 系统调用类似，文件扩展的部分将以 0 填充。如果文件大小被改变了，则文件的 st_mtime 域和 st_ctime 域将会更新。

执行成功返回 0，当有错误发生时则返回-1，错误代码存入 errno 中，详细的错误代码说明请参考 man 手册。

4. utime

用于改变任何文件的 st_mtime 域和 st_ctime 域，即存取时间和修改时间，在 Shell 下输入“man utime”可获取其函数原型：

```
#include <sys/types.h>
#include <utime.h>
int utime(const char *filename, struct utimbuf *buf);

#include <sys/time.h>
int utimes(char *filename, struct timeval *tvp);
```

参数 struct utimbuf *buf 的定义如下：

```
struct utimbuf {
        time_t actime;  /* access time */
        time_t modtime; /* modification time */
    };
```

utime 系统调用会把由第一个参数 filename 指定的文件的存取时间改为第二个参数 buf 的 actime 域，把修改时间改为第二个参数 buf 的 modtime 域，如果 buf 是一个空指针，则将存取时间和修改时间都改为当前时间。

函数执行成功返回 0，当有错误发生时则返回-1，错误代码存入 errno 中，详细的错误代码说明请参考 man 手册。

5. umask

用于设置文件创建时使用的屏蔽字，并返回以前的值（注：umask 没有出错返回），在 Shell

我爱自由 5ifreedom.com

下输入“man 2 umask”可获取其函数原型如下：

```
#include <sys/types.h>
#include <sys/stat.h>
mode_t umask(mode_t mask);
```

在进程创建一个新文件或目录时，如调用 open 函数创建一个新文件，新文件的实际存取权限是 mode 与 umask 按照（mode &～umask）运算以后的结果，umask 函数用来修改进程的 umask。参数 mask 可以直接取数值也可以为 open 系统调用第三个参数 mode 的 11 个宏或它们的组合，关于 11 个宏的内容请参考 6.3.1 小节。下面例 6-8 程序是对函数 pumask 的具体应用，程序功能请参考程序中的注释。

例 6-8 test_umask.c

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main()
{
    umask(0);        // 不屏蔽任何权限
    if (creat("example_681.test",S_IRWXU|S_IRWXG|S_IRWXO) < 0) {
        perror("creat");
        exit(1);
    }

    umask(S_IRWXO);    // 屏蔽其他用户的所以权限
    if (creat("example_682.test",S_IRWXU|S_IRWXG|S_IRWXO) < 0) {
        perror("creat");
        exit(1);
    }

    return 0;
}
```

程序说明。

程序中 creat 以所有者、组、其他用户可读可写可执行的权限创建了两个文件，只不过创建之前用 umask 设置了不同的屏蔽字。执行本程序后，在当前目录下生成了两个文件：example_681.test 和 example_682.test，用 Ls 命令查看它们的属性如下：

```
-rwxrwxrwx   1 root    root        0  Mar    3 14:36 example_681.test
-rwxrwx---   1 root    root        0  Mar    3 14:36 example_682.test
```

从结果可以看到 example_682.test 的其他用户所有的权限都被屏蔽了。

6.5 文件的移动和删除

6.5.1 文件的移动

rename 系统调用可以用来修改文件名或文件的位置，在 Shell 下输入“man 2 rename”可获取其函数原型如下：

```
#include <stdio.h>
    int rename(const char *oldpath, const char *newpath);
```

rename 会将参数 oldpath 所指定的文件名称改为参数 newpath 所指定的文件名称。若 newpath 所指定的文件已存在，则原文件会被删除。

执行成功返回 0，当有错误发生时则返回-1，错误代码存入 errno 中，详细的错误代码说明请参考 man 手册。从错误代码获取错误描述信息的程序代码请参考例 6-2。例 6-9 演示了函数 rename 的用法，程序功能请参考注释。

我爱自由 5ifreedom.com

例 6-9　my_mv.c

```
#include <stdio.h>

int main(int argc, char ** argv)
{
    /*检查参数个数的合法性*/
    if(argc < 3) {
        printf("my_mv <old file> <new file>\n");
        exit(0);
    }

    if (rename(argv[1], argv[2]) < 0) {
        perror("rename");
        exit(1);
    }

    return 0;
}
```

程序说明。

本程序实现了简单的 mv 命令功能，程序先判断参数的合法性，然后利用 rename 系统调用，把命令行第二个参数所指定的文件名改更为命令行第三个参数指定的文件名。

6.5.2　文件的删除

文件的删除可以使用 unlink 系统调用，目录文件的删除则需要使用 rmdir 系统调用。而一个通用的既能删除文件又能删除目录的系统调用是 remove，remove 系统调用实际上是在其内部封装了 unlink 和 rmdir，当需要删除的是文件时，调用 unlink；当需要删除的是目录时，调用 rmdir，rmdir 函数的使用见 6.6.1 节。在 Shell 下输入“man 2 unlink”和“man remove”可获取它们的函数原型如下：

```
#include <unistd.h>
int unlink(const char *pathname);
int remove(const char *pathname);
```

由于 remove 实际上封装了 unlink，在此只介绍 unlink 系统调用。unlink 系统调用从文件系统中删除一个文件，如果文件的链接数为 0 且没有进程打开这个文件，则文件被删除且其占用的磁盘空间被释放。如果文件的链接数虽然为 0，但是有进程打开了这个文件，则文件暂时不删除，直到所有打开该文件的进程都结束时文件才被删除，利用这一点可以确保即使程序崩溃时，它所创建的临时文件也不会遗留下来。

参数 pathname 若指向一个符号链接，则该链接被删除。若参数 pathname 指向一个 socket（套接字文件）、FIFO（命名管道）或设备文件时，该名字被删除，但已经打开这些文件的进程仍然可以使用这些特殊文件。

函数执行成功返回 0，当有错误发生时则返回-1，错误代码存入 errno 中，详细的错误代码说明请参考 man 手册。下面例 6-10 是函数 unlink 的具体应用。

例 6-10　unlink_temp.c

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

/*自定义的错误处理函数*/
void my_err(const char * err_string, int line)
{
    fprintf(stderr, "line:%d  ", line);
    perror(err_string);
    exit(1);
}
```

我爱自由 5ifreedom.com

```
int main()
{
    int       fd;
    char       buf[32];

    if ((fd = open("temp", O_RDWR|O_CREAT|O_TRUNC, S_IRWXU)) < 0) {
        my_err("open", __LINE__);
    }
    if (unlink("temp") < 0) {
        my_err("unlink", __LINE__);
    }
    printf("file unlinked\n");

    if (write(fd, "temp", 5) < 0) {
        my_err("write", __LINE__);
    }
    if ((lseek(fd, 0, SEEK_SET)) == -1) {
        my_err("lseek", __LINE__);
    }
    if (read(fd, buf, 5) < 0) {
        my_err("read", __LINE__);
    }
    printf("%s\n", buf);

    return 0;
}
```

程序说明。

程序在当前目录创建了一个名为 temp 的文件，然后调用 unlink，之后再对文件进行读写操作。程序在 unlink 之后如果出现崩溃，则在程序结束时，temp 也不会遗留下来。

6.6 目录操作

6.6.1 目录的创建和删除

1. 目录的创建

目录的创建可以由 mkdir 系统调用来实现，在 Shell 下输入“man 2 mkdir”可获取其函数原型如下：

```
#include <sys/stat.h>
#include <sys/types.h>
int mkdir(const char *pathname, mode_t mode);
```

mkdir 创建一个新的空目录。空目录中会自动创建.和..目录项。所创建的目录的存取许可权由 mode（mode &～umask）指定。

新创建目录的 uid（所有者）与创建该目录的进程的 uid 一致。如果父目录设置了 st_gid 位，则新创建的目录也设置 st_gid 位（目录被设置该位后，任何用户在此目录下创建的文件的组 id 与该目录的组 id 相同）。

函数执行成功返回 0，当有错误发生时则返回-1，错误代码存入 errno 中，详细的错误代码说明请参考 man 手册。

2. 目录的删除

目录的删除可以由 rmdir 系统调用来完成，在 Shell 下输入“man 2 rmdir”可获取其函数原型如下：

```
#include <unistd.h>
int rmdir(const char *pathname);
```

注意：rmdir 只能删除由参数 pathname 指定的空目录。

执行成功返回 0，当有错误发生时则返回-1，错误代码存入 errno 中，详细的错误代码说明请

我爱自由 5ifreedom.com

参考 man 手册。

6.6.2 获取当前目录

每个进程（可以理解为运行中的程序）都有一个当前工作目录，此目录是搜索所有相对路径名的起点（不以斜线开始的路径名为相对路径名）。当用户登录到 Linux 系统时，其当前工作目录通常是口令文件（/etc/passwd）中该用户登录项的第 6 个字段，可以使用 cat /etc/passwd 命令查看该文件的内容。当前工作目录是进程的一个属性。getcwd 系统调用可以获取进程当前工作目录，在 Shell 下输入“man getcwd”可获取其函数原型如下：

```
#include <unistd.h>
char *getcwd(char *buf, size_t size);
char *get_current_dir_name(void);
char *getwd(char *buf);
```

getcwd 会将当前的工作目录绝对路径复制到参数 buf 所指的内存空间，参数 size 为 buf 的空间大小。在调用此函数时，buf 所指的内存空间要足够大，若工作目录绝对路径的字符串长度超过参数 size 大小，则返回值为 NULL，errno 的值为 ERANGE。倘若参数 buf 为 NULL，getcwd()会根据参数 size 的大小自动分配内存（使用 malloc()），如果参数 size 也为 0，则 getcwd()会根据工作目录绝对路径的字符串长度来决定配置的内存大小。进程可以在使用完此字符串后利用 free()来释放此空间。

执行成功则将结果复制到参数 buf 所指的内存空间，或是返回自动配置的字符串指针。失败返回 NULL，错误代码存于 errno。

如果定义了_GNU_SOURCE，则也可以使用函数 get_current_dir_name 获取当前工作目录。该函数使用 malloc 分配空间来保存工作目录的绝对路径字符串，如果设置了环境变量 PWD，则返回 PWD 的值。

如果定义了_BSD_SOURCE 或_XOPEN_SOURCE_EXTENDED，也可以使用 getwd 获取当前工作目录。该函数不使用 malloc 分配任何空间，参数 buf 指向的空间的大小至少为 PATH_MAX，getwd 仅返回工作目录绝对路径字符串的前 PATH_MAX 个字符。

6.6.3 设置工作目录

使用 chdir 可以更改当前工作目录，在 Shell 下输入“man chdir”可获取其函数原型如下：

```
#include <unistd.h>
int chdir(const char *path);
int fchdir(int fd);
```

chdir 用来将当前工作目录改为由参数 path 指定的目录，fchdir 用来将当前工作目录改为由参数 fd（文件描述符）指定的目录。const char *path 中的 const 的含义是，在 chdir 函数的实现代码中不允许对 path 所指向的字符串的内容进行改变。由于 chdir 是系统调用，不是我们自己定义和实现的函数，因此可以不必去理会 const。

函数执行成功返回 0，当有错误发生时则返回-1，错误代码存入 errno 中，详细的错误代码说明请参考 man 手册。

利用 chdir 编写的 cd 命令程序如例 6-6 所示。

例 6-11　my_cdvc

```
#include <stdio.h>
#include <unistd.h>
#include <Linux/limits.h>

/*自定义的错误处理函数*/
void my_err(const char * err_string, int line)
{
```

我爱自由 5ifreedom.com

```
    fprintf(stderr, "line:%d  ", line);
    perror(err_string);
    exit(1);
}

int main(int argc, char ** argv)
{
    char buf[PATH_MAX + 1];

    if (argc < 2) {
        printf("my_cd <target path>\n");
        exit(1);
    }

    if (chdir(argv[1]) < 0) {
        my_err("chdir", __LINE__);
    }
    if (getcwd(buf, 512) < 0) {
        my_err("getcwd", __LINE__);
    }

    printf("%s\n", buf);

    return 0;
}
```

程序说明。

程序使用 chdir 系统调用将当前工作目录改变为命令行参数所指定的目录，然后利用 getcwd 获取新的工作目录并打印出来，执行结果如下：

```
[root@localhost test]# ./my_cd /home
/home
[root@localhost test]#
```

从结果可以看到，本程序在 Shell 运行后并不能如 cd 命令一样进行目录的切换。这是因为 chdir 只影响调用该函数的进程，对其他进程，如其父进程的当前工作目录，则修改不了。这也是 cd 命令作为少数几个 Shell 内置命令的原因。

6.6.4 获取目录信息

只要对目录具有读权限，就可以获取目录信息。函数 opendir 用来打开一个目录，readdir 用来读取目录中的内容，closedir 关闭一个已经打开的目录。在 Shell 下可查看到它们的函数原型。

1. opendir

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *name);
```

opendir 用来打开参数 name 指定的目录，并返回 DIR*形态的目录流，类似于文件操作中的文件描述符，接下来对目录的读取和搜索都要使用此返回值。成功则返回 DIR*型态的目录流，失败则返回 NULL，错误代码存入 errno 中，详细的错误代码说明请参考 man 手册。

2. readdir

```
#include <sys/types.h>
#include <dirent.h>
struct dirent *readdir(DIR *dir);
```

readdir 用来从参数 dir 所指向的目录中读取出目录项信息，返回一个 struct dirent 结构的指针。

struct dirent 的定义如下：

```
struct dirent
{
    long d_ino;                 /* inode number */
    off_t d_off;                /* offset to this dirent */
```

我爱自由 5ifreedom.com

```
    unsigned short d_reclen;          /* length of this d_name */
    char d_name [NAME_MAX+1];     /* file name (null-terminated) */
}
```

其中，d_ino 是指此目录 i 节点编号，不必理会；d_off 是指目录文件开头至此目录进入点的位移；d_reclen 是指 d_name 的长度；d_name 是指以 NULL 结尾的文件名。

函数执行成功返回该目录下一个文件的信息（存储于 dirent 结构体中），如果调用 opendir 打开某个目录之后，第一次调用该函数，则返回的是该目录下第一个文件的信息，第二次调用该函数返回该目录下第二个文件的信息，依此类推。如果该目录下已经没有文件信息可供读取，则返回 NULL。调用该函数时如果有错误发生或读取到目录文件尾，则返回 NULL，错误代码存入 errno 中，详细的错误代码说明请参考 man 手册。

3. closedir

```
#include <sys/types.h>
#include <dirent.h>
int closedir(DIR *dir);
```

closedir 用来关闭参数 dir 指向的目录，执行成功返回 0，当有错误发生时则返回-1，错误代码存入 errno 中，详细的错误代码说明请参考 man 手册。下面例 6-12 程序实现了上述三个函数的功能。

例 6-12 show_files.c

```
#include <sys/types.h>
#include <dirent.h>
#include <unistd.h>
#include <stdio.h>

int my_readir(const char * path)
{
    DIR          * dir;
    struct dirent    * ptr;

    if ((dir = opendir(path)) == NULL) {
        perror("opendir");
        return -1;
    }
    while ((ptr = readdir(dir))!=NULL) {
        printf("file name: %s\n",ptr->d_name);
    }
    closedir(dir);

    return 0;
}

int main(int argc, char ** argv)
{
    if (argc < 2) {
        printf("listfile <target path>\n");
        exit(1);
    }

    if (my_readir(argv[1]) < 0) {
        exit(1);
    }

    return 0;
}
```

程序说明。

本程序可以将命令行参数指定的目录下的文件显示出来。配合 lstat 等函数可以实现自己的 ls 命令（详见 6.7 小节）。

我爱自由 5ifreedom.com

6.7 编程实践：实现自己的 ls 命令

这一节我们来编程实现自己的简化版 ls 命令：该程序支持-a 和–l 选项的 ls 命令（-a 表示显示隐藏文件，即文件名以“.”开头的文件，–l 表示每个文件单独占一行且将文件的属性显示出来）。

1. 命令示例

```
ls
ls -l
ls -a
ls -al
ls -l /
ls -a / /usr
```

2. 程序主函数的流程图如下

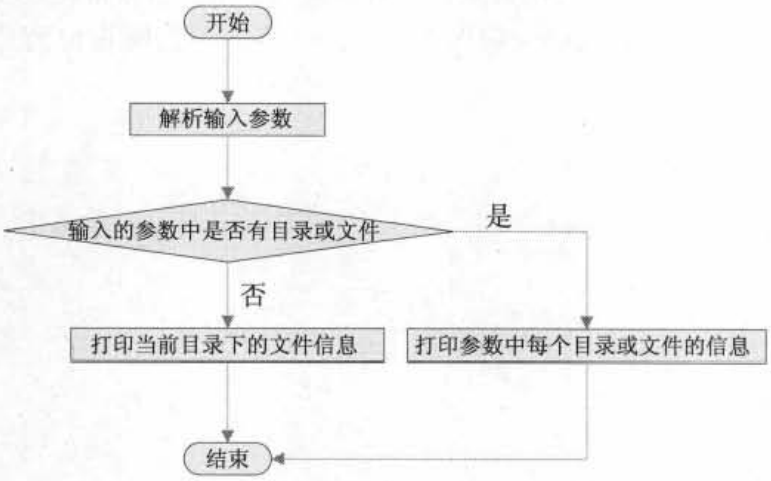


图 my_ls.c 程序的流程图

3. 关键函数的功能及说明

（1）void display_attribute(struct stat buf, char * name)

函数功能：打印文件名为 name 的文件的信息，如：

```
drwxr-xr-x  2  root  root  4096  Sat May 12 14:56:46 2007
```

含义分别为：文件的类型和访问权限、文件的链接数、文件所有者、文件所有者所属的组、文件大小、文件创建的时间。

（2）void display_single(char *name)

函数功能：输出文件的文件名，若命令中没有–l 选项，则输出文件名时要保证上下对齐，如：

```
bin     boot    dev     etc       home
initrd    lib       lib64     liuke     lost+found
```

（3）void display(int flag, char * pathname)

函数功能：根据命令行参数（存放在 flag 中）和完整路径名（存放在 pathname 中）显示目标文件。参数 flag 可以取以下值或者它们的组合，具体如下所示。

- PARAM_NONE：没有选项。
- PARAM_A：带-a 选项，表示显示该目录下的所有文件，包括隐藏文件。
- PARAM_L：带-l 选项，表示显示文件的详细信息，包括文件的访问权限、文件大小等。

我爱自由 5ifreedom.com

（4）void display_dir(int flag_param, char *path)

函数功能：为显示某个目录下的文件做准备，参数 flag_param 用于在调用 display 函数时作为其参数 flag 的实参，path 是要显示的目录。

4．函数流程

（1）获取该目录下文件的总数和最长的文件名。

（2）获取该目录下所有文件的文件名，存放于变量 filenames 中。

（3）使用冒泡法对文件名按字母顺序进行排序，排序后文件名按字母顺序存储于 filenames 中。

（4）调用 display（int flag, char * pathname）函数显示每个文件的信息。

5．文件实现的代码（程序 6-13）

例 6-13 my_ls.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <sys/stat.h>
#include <unistd.h>
#include <sys/types.h>
#include <Linux/limits.h>
#include <dirent.h>
#include <grp.h>
#include <pwd.h>
#include <errno.h>

#define    PARAM_NONE    0    // 无参数
#define    PARAM_A       1    // -a: 显示所有文件
#define    PARAM_L       2    // -l:一行只显示一个文件的详细信息
#define    MAXROWLEN    80    // 一行显示的最多字符数

int       g_leave_len = MAXROWLEN;     // 一行剩余长度，用于输出对齐
int       g_maxlen;                    // 存放某目录下最长文件名的长度

/* 错误处理函数，打印出错误所在行的行数和错误信息 */
void my_err(const char *err_string, int line)
{
    fprintf(stderr, "line:%d  ", line);
    perror(err_string);
    exit(1);
}

/* 获取文件属性并打印 */
void display_attribute(struct stat buf, char * name)
{
    char          buf_time[32];
    struct passwd    *psd;    //从该结构体中获取文件所有者的用户名
    struct group    *grp;    //从该结构体中获取文件所有者所属组的组名

    /*获取并打印文件类型*/
    if (S_ISLNK(buf.st_mode)) {
        printf("l");
    } else if (S_ISREG(buf.st_mode)) {
        printf("-");
    } else if (S_ISDIR(buf.st_mode)) {
        printf("d");
    } else if (S_ISCHR(buf.st_mode)) {
        printf("c");
    } else if (S_ISBLK(buf.st_mode)) {
        printf("b");
    } else if (S_ISFIFO(buf.st_mode)) {
        printf("f");
    } else if (S_ISSOCK(buf.st_mode)) {
```

我爱自由 5ifreedom.com

```
        printf("s");
    }

    /*获取并打印文件所有者的权限*/
    if (buf.st_mode & S_IRUSR){
        printf("r");
    } else {
        printf("-");
    }
    if (buf.st_mode & S_IWUSR){
        printf("w");
    } else {
        printf("-");
    }
    if (buf.st_mode & S_IXUSR){
        printf("x");
    } else {
        printf("-");
    }

    /*获取并打印与文件所有者同组的用户对该文件的操作权限*/
    if (buf.st_mode & S_IRGRP){
        printf("r");
    } else {
        printf("-");
    }
    if (buf.st_mode & S_IWGRP){
        printf("w");
    } else {
        printf("-");
    }
    if (buf.st_mode & S_IXGRP){
        printf("x");
    } else {
        printf("-");
    }

    /*获取并打印其他用户对该文件的操作权限*/
    if (buf.st_mode & S_IROTH){
        printf("r");
    } else {
        printf("-");
    }
    if (buf.st_mode & S_IWOTH){
        printf("w");
    } else {
        printf("-");
    }
    if (buf.st_mode & S_IXOTH){
        printf("x");
    } else {
        printf("-");
    }

    printf("    ");

    /*根据 uid 与 gid 获取文件所有者的用户名与组名*/
    psd = getpwuid(buf.st_uid);
    grp = getgrgid(buf.st_gid);
    printf("%4d ",buf.st_nlink);   // 打印文件的链接数
    printf("%-8s", psd->pw_name);
    printf("%-8s", grp->gr_name);

    printf("%6d",buf.st_size);     // 打印文件的大小
    strcpy(buf_time, ctime(&buf.st_mtime));
    buf_time[strlen(buf_time) - 1] = '0';    // 去掉换行符
    printf("  %s", buf_time);      // 打印文件的时间信息
```

我爱自由 5ifreedom.com

```
}

/* 在没有使用-l 选项时，打印一个文件名，打印时上下行对齐 */
void display_single(char *name)
{
    int i, len;

    // 如果本行不足以打印一个文件名则换行
    if (g_leave_len < g_maxlen) {
        printf("\n");
        g_leave_len = MAXROWLEN;
    }

    len = strlen(name);
    len = g_maxlen - len;
    printf("%-s", name);

    for (i=0; i<len; i++) {
        printf(" ");
    }
    printf("  ");
    //下面的 2 指示空两格
    g_leave_len -= (g_maxlen + 2);
}

/*
*    根据命令行参数和完整路径名显示目标文件
*    参数 flag:  命令行参数
*    参数 pathname:  包含了文件名的路径名
*/
void display(int flag, char * pathname)
{
    int          i,j;
    struct stat       buf;
    char          name[NAME_MAX + 1];

    /*从路径中解析出文件名*/
    for (i=0, j=0; i<strlen(pathname); i++) {
        if (pathname[i] == '/') {
            j = 0;
            continue;
        }
        name[j++] = pathname[i];
    }
    name[j] = '\0';

    /*用 lstat 而不是 stat 以方便解析链接文件*/
    if ( lstat(pathname, &buf) == -1 ) {
        my_err("stat", __LINE__);
    }

    switch (flag) {
        case PARAM_NONE:  // 没有-l 和-a 选项
            if (name[0] != '.') {
                display_single(name);
            }
            break;

        case PARAM_A:      // -a:显示包括隐藏文件在内的所有文件
            display_single(name);
            break;

        case PARAM_L:      // -l:每个文件单独占一行，显示文件的详细属性信息
            if (name[0] != '.') {
                display_attribute(buf, name);
                printf("  %-s\n", name);
            }
```

我爱自由 5ifreedom.com

```
            break;

        case PARAM_A + PARAM_L:        // 同时有-a 和-l 选项的情况
            display_attribute(buf, name);
            printf("  %-s\n", name);
            break;

        default:
            break;
    }
}

void display_dir(int flag_param, char *path)
{
        DIR           *dir;
        struct dirent         *ptr;
        int               count = 0;
        char              filenames[256][PATH_MAX+1],temp[PATH_MAX+1];

        // 获取该目录下文件总数和最长的文件名
        dir = opendir(path);
        if (dir == NULL) {
                my_err("opendir", __LINE__);
        }
        while ((ptr = readdir(dir))!=NULL) {
                if (g_maxlen < strlen(ptr->d_name))
                        g_maxlen =  strlen(ptr->d_name);
                count++;
        }
        closedir(dir);

        if(count>256)
                my_err("too many files under this dir",__LINE__);


        int i, j, len = strlen(path);
        // 获取该目录下所有的文件名
        dir = opendir(path);
        for(i = 0; i < count; i++){
                ptr = readdir(dir);
                if( ptr == NULL){
                        my_err("readdir",__LINE__);
                }
                strncpy(filenames[i],path,len);
                filenames[i][len] = '\0';
                strcat(filenames[i],ptr->d_name);
                filenames[i][len+strlen(ptr->d_name)] = '\0';
        }

        // 使用冒泡法对文件名进行排序，排序后文件名按字母顺序存储于 filenames
        for(i = 0; i < count-1; i++)
                for(j = 0; j < count-1-i; j++) {
                        if( strcmp(filenames[j],filenames[j+1]) > 0 ) {
                                strcpy(temp,filenames[j+1]);
                                temp[strlen(filenames[j+1])] = '\0';
                                strcpy(filenames[j+1],filenames[j]);
                                filenames[j+1][strlen(filenames[j])] = '\0';
                                strcpy(filenames[j], temp);
                                filenames[j][strlen(temp)] = '\0';
                        }
                }
        for(i = 0; i < count; i++)
                display(flag_param, filenames[i]);

        closedir(dir);

        // 如果命令行中没有-l 选项，打印一个换行符
```

我爱自由 5ifreedom.com

```
        if( (flag_param & PARAM_L) == 0)
            printf("\n");
}


int main(int argc, char ** argv)
{
    int             i, j, k, num;
    char            path[PATH_MAX+1];
    char            param[32];    // 保存命令行参数，目标文件名和目录名不在此列
    int             flag_param = PARAM_NONE; // 参数种类，即是否有-l 和-a 选项
    struct stat         buf;

    /*命令行参数的解析，分析-l、-a、-al、-la 选项*/
    j   = 0,
    num = 0;
    for (i=1; i<argc; i++) {
        if (argv[i][0] == '-') {
            for(k=1; k < strlen(argv[i]); k++,j++) {
                param[j] = argv[i][k];    // 获取-后面的参数保存到数组 param 中
            }
        num++;  // 保存"-"的个数
        }
    }

    /*只支持参数 a 和 l，如果含有其他选项就报错*/
    for(i=0; i<j; i++) {
        if (param[i] == 'a') {
            flag_param |= PARAM_A;
            continue;
        } else if (param[i] == 'l') {
            flag_param |= PARAM_L;
            continue;
        } else {
            printf("my_ls: invalid option -%c\n", param[i]);
            exit(1);
        }
    }
    param[j] = '\0';

    // 如果没有输入文件名或目录，就显示当前目录
    if ((num + 1) == argc) {
        strcpy(path, "./");
        path[2] = '\0';
        display_dir(flag_param, path);
        return 0;
    }

    i=1;
    do {
        // 如果不是目标文件名或目录，解析下一个命令行参数
        if (argv[i][0] == '-') {
            i++;
            continue;
        } else {
            strcpy(path, argv[i]);

            // 如果目标文件或目录不存在，报错并退出程序
            if ( stat(path, &buf) == -1 )
                my_err("stat", __LINE__);

            if ( S_ISDIR(buf.st_mode) ) {   // argv[i]是一个目录
                // 如果目录的最后一个字符不是'/'，就加上'/'
                if ( path[ strlen(argv[i])-1 ] != '/') {
                    path[ strlen(argv[i]) ] = '/';
                    path[ strlen(argv[i])+1 ] = '\0';
                }
```

我爱自由 5ifreedom.com

```
                else
                    path[ strlen(argv[i]) ] = '\0';

                display_dir(flag_param,path);
                i++;
            }
            else {  //argv[i]是一个文件
                display(flag_param, path);
                i++;
            }
        }
    } while (i<argc);

    return 0;
}
```

运行结果：

```
[root@ localhost tmp]# ./my_ls  -al  /
drwxr-xr-x     20 root      root     4096  Tue Apr 10 11:42:04    2007   .
drwxr-xr-x     20 root      root     4096  Tue Apr 10 11:42:04    2007   ..
-rw-r--r--      1 root      root     0  Tue Apr 10 11:42:04    2007   .autofsck
-rw-------      1 root      root     12288  Thu Jan 25 14:04:58    2007  .byebye.swp
drwxr-xr-x      2 root      root     4096  Fri Jan 26 14:54:00    2007   bin
drwxr-xr-x      4 root      root     1024  Tue Apr 10 11:42:04    2007   boot
drwxr-xr-x     24 root      root    118784  Tue Apr 10 11:55:38    2007   dev
drwxr-xr-x     67 root      root     8192  Sun Apr 15 16:07:59    2007   etc
drwxr-xr-x      7 root      root     4096  Tue Apr 10 17:54:43    2007   home
drwxr-xr-x      2 root      root     4096  Sat Jan 25 07:52:28    2003   initrd
drwxr-xr-x     11 root      root     4096  Fri Jan 26 14:53:52    2007   lib
[root@ localhost tmp]# ./my_ls  /usr
X11R6     bin         dict     etc      games     include      java   kerberos
lib         libexec      local    sbin    share      src         tmp
```

本程序由于篇幅的限制，只实现了-a 和-l 两个命令行选项。对本程序的进一步完善请参考本章的习题。

6.8 习题

1．在例 6-1 的基础上完善 chmod 命令，使之支持如 u+x，g-w，o-w 等功能。

2．例 6-13 my_ls.c 中的 display_dir 函数对某个目录下所有文件名使用冒泡法进行排序时，执行效率不高。原因在于：一是因为排序时进行了大量的 strcpy 操作；二是冒泡排序算法本身的执行效率不高。对于第一个原因，可以定义一个 int index[count]数组（通过 malloc 动态生成），排序时只是交换各个字符串的索引（filenames 数组的下标）而不进行实际的字符串拷贝。对于第二个原因，可以使用快速排序法或归并排序法。请编程实现，以提高程序的执行效率。

3．使用 man ls 命令查看 ls 命令的帮助信息，选择几个选项（如-i、-r、-t），并在例 6-13 my_ls.c 的基础上进行修改以使该程序支持更多的选项。

我爱自由 5ifreedom.com

第 7 章　进程控制

本章主要介绍对进程的操作，首先介绍了进程的概念、进程的结构和进程的内存映像，之后以程序实例的方式介绍了创建进程，退出进程，在进程中执行新程序，以及获取进程 ID，改变进程优先级等。最后，以一个 shell 综合实例的实现把本章知识贯串起来。

本章重点：

- 进程的概念。
- 进程的内存映像。
- 进程的各种操作。

本章难点：

- 守护进程的创建。
- myshell 综合实例。

7.1 进程概述

操作系统的主要任务是管理计算机的软、硬件资源。现代操作系统的主要特点在于程序的并行执行，Linux 操作系统亦是如此。操作系统借助于进程来管理计算机的软、硬件资源，支持多任务的并行执行。操作系统最核心的概念就是进程。理解和掌握进程控制，是进行 Linux 下系统编程的关键。

7.1.1　Linux 进程

Linux 是一个多用户多任务的操作系统。多用户是指多个用户可以在同一时间使用计算机；多任务是指 Linux 可以同时执行几个任务，可以在还未执行完一个任务时又执行另一个任务。进程简单地讲就是运行中的程序，Linux 系统的一个重要特点是可以同时启动多个进程。根据操作系统的定义：进程是操作系统资源管理的最小单位。以下将介绍进程的概念、进程的标识、Linux 进程的结构和状态。

1．进程概念

进程是一个动态的实体，是程序的一次执行过程。进程是操作系统资源分配的基本单位。要掌握进程的概念就要将其与程序、线程两个概念区别开。进程和程序的区别在于进程是动态的，程序是静态的；进程是运行中的程序，程序是一些保存在硬盘上的可执行的代码。

为了让计算机在同一时间内能执行更多任务，在进程内部又划分了许多线程。线程在进程内部，它是比进程更小的能独立运行的基本单位。线程基本上不拥有系统资源，它与同属一个进程的其他线程共享进程拥有的全部资源。进程在执行过程中拥有独立的内存单元，其内部的线程共享这些内存。一个线程可以创建和撤销另一个线程，同一个进程中的多个线程可以并行执行。

Linux 下可通过命令 ps 或 pstree 查看当前系统中的进程。

2．进程标识

Linux 操作系统中，每个进程都是通过惟一的进程 ID 标识的。进程 ID 是一个非负数。每个进程除了进程 ID 外还有一些其他标识信息，它们都可以通过相应的函数获得。这些函数的声明

我爱自由 5ifreedom.com

在 unistd.h 头文件中。表 7-1 是这些函数的说明。

表 7-1　　获取进程各种标识符的函数表

函数声明	功　能
pid_t getpid(id)	获得进程 ID
pid_t getppid(id)	获得进程父进程的 ID
pid_t getuid()	获得进程的实际用户 ID
pid_t geteuid()	获得进程的有效用户 ID
pid_t getgid()	获得进程的实际组 ID
pid_t getegid(id)	获得进程的有效组 ID

用户 ID 和组 ID 的相关概念如下所示。

- 实际用户 ID(uid)：标识运行该进程的用户。
- 有效用户 ID(euid)：标识以什么用户身份来运行进程。例如，某个普通用户 A，运行了一个程序，而这个程序是以 root 身份来运行的，这程序运行时就具有 root 权限。此时实际用户 ID 是 A 用户的 ID，而有效用户 ID 是 root 用户 ID。
- 实际组 ID(gid)：它是实际用户所属的组的组 ID。
- 有效组 ID(egid)：有效组 ID 是有效用户所属的组的组 ID。

3. Linux 进程的结构

Linux 中一个进程由 3 部分组成：代码段、数据段和堆栈段，如图 7-1 所示。

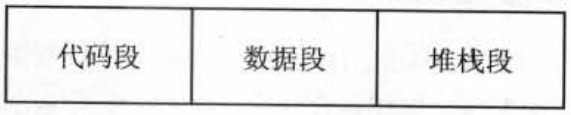


图 7-1　Linux 进程结构

代码段存放程序的可执行代码。数据段存放程序的全局变量、常量、静态变量。堆栈段中的堆用于存放动态分配的内存变量；堆栈段中的栈用于函数调用，它存放着函数的参数、函数内部定义的局部变量。

4. Linux 进程状态

Linux 系统中的进程有以下几种状态。

- 运行状态：进程正在运行或在运行队列中等待运行。
- 可中断等待状态：进程正在等待某个事件完成（如等待数据到达）。等待过程中可以被信号或定时器唤醒。
- 不可中断等待状态：进程也正在等待某个事件完成，在等待中不可以被信号或定时器唤醒，必须等待直到等待的事件发生。
- 僵死状态：进程已终止，但进程描述符依然存在，直到父进程调用 wait()函数后释放。
- 停止状态：进程因为收到 SIGSTOP、SIGSTP、SIGTIN、SIGTOU 信号后停止运行或者该进程正在被跟踪（调试程序时，进程处于被跟踪状态）。

用 ps 命令可以查看进程的当前状态。运行状态为 R(runnable)，可中断等待状态为 S(sleeping)，不可中断等待状态为 D(uninterruptible sleep)，僵死状态为 Z(zombile)，停止状态为 T(traced or stopped)。下面显示了执行 ps 命令的结果，因为系统中进程较多，只显示了其中一部分。

```
[zzy@mci ~]$ ps -eo pid,stat
PID STAT
  1 S
  2 SN
  3 S<
```

我爱自由 5ifreedom.com

```
  120 S<s
 2628 Ss
 2816 Ssl
 2857 SLs
 2949 Sl
 3285 Ss+
 8430 S+
 9352 R+
```

在运行结果中有一些后缀字符，其意义分别为：<（高优先级进程），N（低优先级进程），L（内存锁页，即页不可以被换出内存），s（该进程为会话首进程），l（多线程进程），+（进程位于前台进程组）。例如，Ssl 说明该进程处于可中断等待状态，且该进程为会话首进程，而且是一个多线程的进程。

7.1.2 进程控制

Linux 进程控制包括创建进程、执行新程序、退出进程以及改变进程优先级等。Linux 系统为了对进程进行控制而提供了一系列的系统调用，用户可以使用这些系统调用来完成创建一个新进程、终止一个进程、改变进程的优先级别等操作。

在 Linux 系统中，用于对进程进行控制的主要系统调用以下所示。

- fork：用于创建一个新进程。
- exit：用于终止进程。
- exec：用于执行一个应用程序。
- wait：将父进程挂起，等待子进程终止。
- getpid：获取当前进程的进程 ID。
- nice：改变进程的优先级。

以上系统调用的用法会在后面通过具体的实例详细介绍。

7.1.3 进程的内存映像

1. Linux 下程序转化成进程

Linux 下 C 程序的生成分为 4 个阶段：预编译、编译、汇编、链接。编译器 gcc 经过预编译、编译、汇编 3 个步骤将源程序文件转换为目标文件。如果程序有多个目标文件或者程序中使用了库函数，编译器还要将所有的目标文件或所需的库链接起来，最后生成可执行程序。当程序执行时，操作系统将可执行程序复制到内存中。程序转化为进程通常要经过以下步骤。

- 内核将程序读入内存，为程序分配内存空间。
- 内核为该进程分配进程标识符（PID）和其他所需资源。
- 内核为该进程保存 PID 及相应的状态信息，把进程放到运行队列中等待执行。程序转化为进程后就可以被操作系统的调度程序调度执行了。

2. 进程的内存映像

进程的内存映像是指内核在内存中如何存放可执行程序文件。在将程序转化为进程的过程中，操作系统将可执行程序由硬盘复制到内存中。

Linux 下程序映像的一般布局如图 7-2 所示。

从内存的低地址到高地址依次如下。

- 代码段：即二进制机器代码，代码段是只读的，可被多个进程共享。如一个进程创建了一个子进程，父子进程共享代码段，此外子进程还获得父进程数据段、堆、栈的复制。

我爱自由 5ifreedom.com

- 数据段：存储已被初始化的变量，包括全局变量和已被初始化的静态变量。
- 未初始化数据段：存储未被初始化的静态变量，它也被称为 bss 段。
- 堆：用于存放程序运行中动态分配的变量。
- 栈：用于函数调用，保存函数的返回地址、函数的参数、函数内部定义的局部变量。

另外，高地址还存储了命令行参数和环境变量。

可执行程序和内存映像的区别在于：可执行程序位于磁盘中而内存映像位于内存中；可执行程序没有堆栈，因为程序被加载到内存中才会分配堆栈；可执行程序虽然也有未初始化数据段但它并不被储存在位于硬盘中的可执行文件中；可执行程序是静态的、不变的，而内存映像随着程序的执行是在动态变化的，例如，数据段随着程序的执行要存储新的变量值，栈在函数调用时也是不断在变化中。

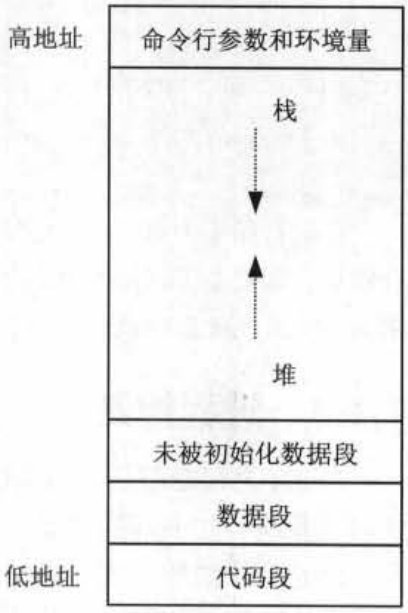


图 7-2　程序映像的布局

7.2 进程操作

7.2.1 创建进程

每个进程由进程 ID 号标识，进程被创建时系统会为其分配一个惟一的进程 ID。当一个进程向其父进程（创建该进程的进程）传递其终止消息时，意味这个进程的整个生命周期的结束。此时，该进程占用的所有资源包括进程 ID 被全部释放。

创建进程有两种方式，一是由操作系统创建；二是由父进程创建。由操作系统创建的进程，它们之间是平等的，一般不存在资源继承关系。而对于由父进程创建的进程（通常称为子进程），它们和父进程存在隶属关系。子进程又可以创建进程，这样形成一个进程家族。子进程可以继承其父进程几乎所有的资源。在系统启动时，操作系统会创建一些进程，它们承担着管理和分配系统资源的任务，这些进程通常被称为系统进程。

系统调用 fork 是创建一个新进程的惟一方法（创建一个进程通常也称为 fork 一个进程），除了极少数以特殊方式创建的进程，如 init 进程，它是内核启动时以特殊方式创建的。进程调用 fork 函数就创建了一个子进程。

创建了一个子进程之后，父进程和子进程争夺 CPU，抢到 CPU 者执行，另外一个挂起等待。如果想要父进程等待子进程执行完毕以后再继续执行，可以在 fork 操作之后调用 wait 或 waitpid。一个刚刚被 fork 的子进程会和它的父进程一样，继续执行当前的程序。几个进程同时执行一个应用程序通常用处不大。更常见的使用方法是子进程在被 fork 后可以通过调用 exec 函数执行其他程序。

1．fork 函数

前面已经提到，fork 函数是创建一个新进程的惟一方法。后面介绍的 vfork 函数虽然也可以

我爱自由 5ifreedom.com

创建进程，但它在创建进程时实际上还是调用了 fork 函数。在命令行下输入 man 2 fork 获得该函数的声明：

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

一般情况下，函数最多有一个返回值，但 fork 函数非常特殊，它有两个返回值，即调用一次返回两次。成功调用 fork 函数后，当前进程实际上已经分裂为两个进程，一个是原来的父进程，另一个是刚刚创建的子进程。父子进程在调用 fork 函数的地方分开，fork 函数有两个返回值，一个是父进程调用 fork 函数后的返回值，该返回值是刚刚创建的子进程的 ID；另一个是子进程中 fork 函数的返回值，该返回值是 0。fork 函数返回两次的前提是进程创建成功，如果进程创建失败，则只返回-1。两次返回不同的值，子进程返回值为 0，而父进程的返回值为新创建的子进程的进程 ID，这样可以用返回值来区别父、子进程。例 7-1 示范了 fork 函数的常见用法：

例 7-1 fork.c

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
    pid_t  pid;

    printf("Process Creation Study\n");
    pid = fork();
    switch(pid) {
    case 0:
        printf("Child process is running,CurPid is %d, ParentPid is %d\n", pid, getppid());
        break;
    case -1:
        perror("Process creation failed\n");
        break;
    default:
        printf("Parent process is running,ChildPid is %d, ParentPid is %d\n", pid, getpid());
        break;
}
    exit(0);
}
```

当前运行子进程，ID=getpid()

printf("Child Process is running,CurPid is %d,ParentPid is %d\n",getpid(),getppid());

程序的一次运行的结果如下：

```
Process Creation Study
Child process is running, CurPid is 0, ParentPid is 5585
Parent process is running, ChildPid is 5586, ParentPid is 5585
```

程序说明。

从运行结果可以看出，进程创建成功后，fork 函数返回了两次：一次返回值是 0，代表子进程在运行，通过函数 getppid 得到其父进程 ID 为 5585；另外一次的返回值为 5586，即子进程的进程 ID，代表当前父进程在运行，通过函数 getpid 得到父进程的 ID 为 5585，刚好与前面得到的父进程 ID 一致。我们可以看到，switch 语句被执行了两次，一次是在子进程中执行，另一次是在父进程中执行。

如果再次运行此程序，系统分配给进程的 ID 一般会发生变化。以下是程序另外一次运行的结果。

```
Process Creation Study
Child process is running,CurPid is 0,ParentPid is 5747
Parent process is running,ChildPid is 5748,ParentPid is 5747
```

例 7-1 的两次运行结果均是子进程先运行。一般来说，fork 之后是父进程先执行还是子进程先执行是不确定的，这取决于内核所使用的调度算法。将例 7-1 稍微修改一下就可以发现父子进程是交替执行的，因为操作系统一般让所有进程都享有同等执行权，除非某进程的优先级比其他的高，例 7-2 是对例 7-1 的改进。

我爱自由 5ifreedom.com

例 7-2　fork2.c

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
    pid_t  pid;
    char *  msg;
    int    k;

    printf("Process Creation Study\n");
    pid = fork();
    switch(pid) {
        case 0:
            msg = "Child process is running";
            k = 3;
            break;
        case -1:
            perror("Process creation failed\n");
            break;
        default:
            msg = "Parent process is running";
            k=5;
            break;
    }

    while(k > 0)
    {
        puts(msg);
        sleep(1);
        k--;
    }

    exit(0);
}
```

程序的运行结果如下：

```
Process Creation Study
Child process is running
Parent process is running
Parent process is running
Child process is running
Parent process is running
Child process is running
Parent process is running
Parent process is running
```

程序说明。

例 7-2 中子进程输出 3 条消息，父进程输出 5 条消息。执行子进程和执行父进程时打印出的消息是不一样的。可以从输出上看出父子进程交替执行。

前面提到 fork 在创建进程失败时，返回–1。失败的原因通常是父进程拥有的子进程的个数超过了规定的限制，此时 errno 值为 EAGAIN。如果可供使用的内存不足也会导致进程创建失败，此时 errno 值为 ENOMEM。

子进程会继承父进程的很多属性，主要包括用户 ID、组 ID、当前工作目录、根目录、打开的文件、创建文件时使用的屏蔽字、信号屏蔽字、上下文环境、共享的存储段、资源限制等。子进程与父进程有一些不同的属性，主要有如下这些。

- 子进程有它自己惟一的进程 ID。
- fork 的返回值不同，父进程返回子进程的 ID，子进程的则为 0。
- 不同的父进程 ID。子进程的父进程 ID 为创建它的父进程 ID。

我爱自由 5ifreedom.com

- 子进程共享父进程打开的文件描述符，但父进程对文件描述符的改变不会影响子进程中的文件描述符。
- 子进程不继承父进程设置的文件锁。
- 子进程不继承父进程设置的警告。
- 子进程的未决信号集被清空。

2. 孤儿进程

如果一个子进程的父进程先于子进程结束，子进程就成为一个孤儿进程，它由 init 进程收养，成为 init 进程的子进程，例 7-3 演示这种情况的用法。

例 7-3 fork3.c

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
    pid_t  pid;

    pid = fork();
    switch(pid) {
        case 0:
            while(1) {
                printf("A background process, PID:%d\n, ParentID: %d\n",
                        getpid(),getppid());
                sleep(3);
            }
        case -1:
            perror("Process creation failed\n");
            exit(-1);
        default:
            printf("I am parent process,my pid is %d\n",getpid());
            exit(0);
    }

    return 0;
}
```

程序的一次运行结果如下：

```
[zzy@mci src]$ ./fork3
A background process, PID:4973, ParentID: 4972
I am parent process,my pid is 4972
A background process, PID:4973, ParentID: 1
A background process, PID:4973, ParentID: 1
A background process, PID:4973, ParentID: 1
```

程序说明。

从输出结果上可以看到，调用 fork 函数后，子进程先执行，打印出自己的 ID 号 4973 和父进程 ID 号 4972。之后父进程执行，打印出一行消息后，就执行完毕了。此后子进程就成了孤儿进程，由 init 进程收养。可以看到此时子进程的父进程 ID 变为 1。

3. vfork 函数

vfork 也可以用来创建一个新进程，与 fork 相比，它有自己独特的用处。以下针对两者的异同点作一个详细的对比。

- vfork 和 fork 一样都是调用一次，返回两次。
- 使用 fork 创建一个子进程时，子进程只是完全复制父进程的资源。这样得到的子进程独立于父进程，具有良好的并发性。而使用 vfork 创建一个子进程时，操作系统并不将父进程的地址空间完全复制到子进程，用 vfork 创建的子进程共享父进程的地址空间，也

我爱自由 5ifreedom.com

就是说子进程完全运行在父进程的地址空间上。子进程对该地址空间中任何数据的修改同样为父进程所见。

- 使用 fork 创建一个子进程时，哪个进程先运行取决于系统的调度算法。而 vfork 一个进程时，vfork 保证子进程先运行，当它调用 exec 或 exit 之后，父进程才可能被调度运行。如果在调用 exec 或 exit 之前子进程要依赖父进程的某个行为，就会导致死锁。

因为使用 fork 创建一个子进程时，子进程需要将父进程几乎每种资源都复制，所以 fork 是一个开销很大的系统调用，这些开销并不是所有情况都需要的。比如 fork 一个进程后，立即调用 exec 执行另外一个应用程序，那么 fork 过程中子进程对父进程地址空间的复制将是一个多余的过程。vfork 不会拷贝父进程的地址空间，这大大减小了系统开销。

为了区别 fork 和 vfork。下面通过例 7-4 来进行说明。该程序通过观察父子进程的执行顺序和对父进程变量的修改说明两者的区别。

例 7-4　diffork.c

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int globVar = 5;

int main(void)
{
    pid_t  pid;
    int    var = 1, i;

    printf("fork is diffirent with vfrok \n");

    pid = fork();
    /*pid = vfork();*/
    switch(pid) {
        case 0:
            i = 3;
            while(i-- > 0)
            {
                printf("Child process is running\n");
                globVar++;
                var++;
                sleep(1);
            }
            printf("Child's globVar = %d,var = %d\n",globVar,var);
            break; exit(0);//需要添加这句，正常退出子进程，否则var值不连续
        case -1:
            perror("Process creation failed\n");
            exit(0);
        default:
            i = 5;
            while(i-- > 0)
            {
                printf("Parent process is running\n");
                globVar++;
                var++;
                sleep(1);
            }
            printf("Parent's globVar = %d ,var = %d\n", globVar ,var);
            exit(0);
    }
}
```

当使用 fork 创建子进程的时候，运行结果如下：

```
[zzy@mci src]$ ./diffork
fork is diffirent with vfrok
Child Process is running
```

我爱自由 5ifreedom.com

```
parent process is running
Child Process is running
parent process is running
Child Process is running
parent process is running
Child's globVar = 8,var = 4
parent process is running
parent process is running
Parent's globVar =10, var = 6
```

取消程序中的注释行，而将“pid = fork();”注释掉，使用 vfork 创建子进程，程序运行结果如下：

```
[zzy@mci src]$ ./diffork
    fork is diffirent with vfrok
    Child Process is running
    Child Process is running
    Child Process is running
    Child's globVar = 8,var = 4
    parent process is running
    parent process is running
    parent process is running
    parent process is running
    parent process is running
    Parent's globVar =13, var = 9
```

程序说明。

使用 fork 创建子进程时，子进程继承了父进程的全局变量和局部变量。子进程中，最后全局变量 globVar 和局部变量 var 的值均递增 3，分别为 8 和 4，而父进程中两者分别递增 5，最后结果为 10 和 6 这证明了 fork 的子进程有自己独立的地址空间。不管是全局变量还是局部变量，子进程与父进程对它们的修改互不影响。vfork 创建子进程后，父进程中 globVar 和 var 最后均递增了 8。这是因为 vfork 的子进程共享父进程的地址空间，子进程修改变量对父进程是可见的。

从运行结果还可以看出，用 fork 创建了子进程后，父子进程的执行顺序是不确定的，父子进程的输出是混杂在一起的。而使用 vfork 创建子进程后，打印的结果是子进程在前，父进程在后，说明 vfrok 保证子进程先执行，在子进程调用 exit 或 exec 之前父进程处于阻塞等待状态。可以将程序中 sleep 函数的参数增大一些，使用 ps aux 可以看到父子进程的状态分别为 S（阻塞状态）和 D（不可中断状态）。结果如下：

```
[zzy@mci ~]$ ps aux
USER  PID  %CPU %MEM  VSZ  RSS  TTY   STAT  START  TIME   COMMAND
zzy   32267  0.0   0.0   2376  296  pts/1   D   13:46   0:00    ./diffork
zzy   32268  0.0   0.0   2376  296  pts/1   S   13:46   0:00    ./diffork
```

注意：使用 vfork 时要谨慎，最好不要允许子进程修改与父进程共享的全局变量和局部变量。

7.2.2 创建守护进程

守护进程（daemon）是指在后台运行的、没有控制终端与之相连的进程。它独立于控制终端，通常周期性地执行某种任务。守护进程是一种很有用的进程，Linux 的大多数服务器就是用守护进程的方式实现的，如 Internet 服务器进程 inetd，Web 服务器进程 http 等。守护进程在后台运行，类似于 Windows 中的系统服务。

守护进程的启动方式有多种：它可以在 Linux 系统启动时从启动脚本/etc/rc.d 中启动；可以由作业规划进程 crond 启动；还可以由用户终端（通常是 Shell）执行。

编写创建守护进程的程序时，要尽量避免产生不必要的交互。编写创建守护进程程序有如下要点。

- 让进程在后台执行。方法是调用 fork 产生一个子进程，然后使得父进程退出。程序 7-3 便是一个实例。
- 调用 setsid 创建一个新对话期。控制终端，登录会话和进程组通常是从父进程继承下来的，

我爱自由 5ifreedom.com

守护进程要摆脱它们，不受它们的影响，其方法是调用 setsid 使进程成为一个会话组长。

注意：当进程是会话组长时，调用 setsid 会失败。但第一点已经保证进程不是会话组长。setsid 调用成功后，进程成为新的会话组长和进程组长，并与原来的登录会话和进程组脱离。由于会话过程对控制终端的独占性，进程同时与控制终端脱离。

- 禁止进程重新打开控制终端。经过以上步骤，进程已经成为一个无终端的会话组长，但是它可以重新申请打开一个终端。为了避免这种情况的发生，可以通过使进程不再是会话组长来实现。再一次通过 fork 创建新的子进程，使调用 fork 的进程退出。
- 关闭不再需要的文件描述符。创建的子进程从父进程继承打开的文件描述符。如不关闭，将会浪费系统资源，造成进程所在的文件系统无法卸下以及引起无法预料的错误。先得到最高文件描述符值，然后用一个循环程序，关闭 0 到最高文件描述符值的所有文件描述符。
- 将当前目录更改为根目录。当守护进程当前工作目录在一个装配文件系统中时，该文件系统不能被拆卸。一般需要将工作目录改为根目录。
- 将文件创建时使用的屏蔽字设置为 0。进程从创建它的父进程那里继承的文件创建屏蔽字可能会拒绝某些许可权。为防止这一点，使用 umask(0)将屏蔽字清零。
- 处理 SIGCHLD 信号。这一步不是必须的。但对于某些进程，特别是服务器进程往往在请求到来时生成子进程处理请求。如果父进程不等待子进程结束，子进程将成为僵尸进程（zombie），从而占用系统资源。如果父进程等待子进程结束，将增加父进程的负担，影响服务器进程的并发性能。在 Linux 下可以简单地将 SIGCHLD 信号的操作设为 SIG_IGN。这样，子进程结束时不会产生僵尸进程，例 7-5 实现了创建守护进程的方法。

例 7-5　daemon.c

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>
#include <sys/param.h>
#include <sys/stat.h>
#include <time.h>
#include <syslog.h>

int init_daemon(void)
{
    int pid;
    int i;

    /*忽略终端 I/O 信号，STOP 信号*/
    signal(SIGTTOU,SIG_IGN);
    signal(SIGTTIN,SIG_IGN);
    signal(SIGTSTP,SIG_IGN);
    signal(SIGHUP,SIG_IGN);

    pid = fork();
    if(pid > 0) {
        exit(0);  /*结束父进程，使得子进程成为后台进程*/
    }
    else if(pid < 0) {
        return -1;
    }
    /*建立一个新的进程组,在这个新的进程组中,子进程成为这个进程组的首进程,以使该进程脱离所有终端*/
    setsid();

    /*再次新建一个子进程，退出父进程，保证该进程不是进程组长，同时让该进程无法再打开一个新的终端*/
    pid=fork();
    if( pid > 0) {
        exit(0);
```

我爱自由 5ifreedom.com

```
    }
    else if( pid< 0) {
        return -1;
    }

    /*关闭所有从父进程继承的不再需要的文件描述符*/
    for(i=0;i< NOFILE;close(i++));

    /*改变工作目录，使得进程不与任何文件系统联系*/
    chdir("/");

    /*将文件屏蔽字设置为 0*/
    umask(0);

    /*忽略 SIGCHLD 信号*/
    signal(SIGCHLD,SIG_IGN);

    return 0;
}

int main()
{
    time_t now;
    init_daemon();
    syslog(LOG_USER|LOG_INFO,"测试守护进程! \n");
    while(1) {
        sleep(8);
        time(&now);
        syslog(LOG_USER|LOG_INFO,"系统时间: \t%s\t\t\n",ctime(&now));
    }
}
```

编译运行此程序。然后用 ps –ef 查看进程状态，该进程的状态如下：

```
UID       PID    PPID   C  STIME   TTY       TIME    CMD
zzy       30074    1    0  20:37    ?     00:00:00   ./daemon
```

从结果可以看出该进程具备守护进程的所有特征。

查看系统日志的结果片段如下：

```
Jan 28 20:38:43 zzy daemonproc: 系统时间:      Sun Jan 28 20:38:43 2007
Jan 28 20:38:51 zzy daemonproc: 系统时间:      Sun Jan 28 20:38:51 2007
Jan 28 20:38:59 zzy daemonproc: 系统时间:      Sun Jan 28 20:38:59 2007
```

注意：使用 syslog 函数前，首先需要配置/etc/syslog.conf 文件，在该文件的最后加入一行 user.* /var/log/test.log。然后重新启动 syslog 服务，命令分别为：/etc/init.d/syslog stop 和/etc/init.d/syslog start。这样 syslog 的信息就会写入 test.log 文件中。

7.2.3 进程退出

进程退出表示进程即将结束运行。在 Linux 系统中进程退出的方法分为正常退出和异常退出两种。其中正常退出的方法有 3 种，异常退出的方法有两种。

（1）正常退出

- 在 main 函数中执行 return。
- 调用 exit 函数。
- 调用_exit 函数。

（2）异常退出

- 调用 about 函数。
- 进程收到某个信号，而该信号使程序终止。

不管是哪种退出方式，最终都会执行内核中的同一段代码。这段代码用来关闭进程所有已打开的文件描述符，释放它所占用的内存和其他资源。

我爱自由 5ifreedom.com

以下是各种退出方式的比较。

- exit 和 return 的区别：exit 是一个函数，有参数；而 return 是函数执行完后的返回。exit 把控制权交给系统，而 return 将控制权交给调用函数。
- exit 和 abort 的区别：exit 是正常终止进程，而 about 是异常终止。
- exit(int exit_code)：exit 中的参数 exit_code 为 0 代表进程正常终止，若为其他值表示程序执行过程中有错误发生，比如溢出、除数为 0。
- exit()和_exit()的区别：exit 在头文件 stdlib.h 中声明，而_exit()声明在头文件 unistd.h 中。两个函数均能正常终止进程，但是_exit()会执行后立即返回给内核，而 exit()要先执行一些清除操作，然后将控制权交给内核。

父子进程终止的先后顺序不同会产生不同的结果。在子进程退出前父进程先退出，则系统会让 init 进程接管子进程。当子进程先于父进程终止，而父进程又没有调用 wait 函数等待子进程结束，子进程进入僵死状态，并且会一直保持下去除非系统重启。子进程处于僵死状态时，内核只保存该进程的一些必要信息以备父进程所需。此时子进程始终占用着资源，同时也减少了系统可以创建的最大进程数。如果子进程先于父进程终止，且父进程调用了 wait 或 waitpid 函数，则父进程会等待子进程结束。

7.2.4 执行新程序

使用 fork 或 vfork 创建子进程后，子进程通常会调用 exec 函数来执行另外一个程序。系统调用 exec 用于执行一个可执行程序以代替当前进程的执行映像。

注意：exec 调用并没有生成新进程。一个进程一旦调用 exec 函数，它本身就“死亡”了，系统把代码段替换成新的程序的代码，废弃原有的数据段和堆栈段，并为新程序分配新的数据段与堆栈段，惟一保留的就是进程 ID。也就是说，对系统而言，还是同一个进程，不过执行的已经是另外一个程序了。

Linux 下，exec 函数族有以下 6 种不同的调用的形式，它们的声明在头文件<unistd.h>中，格式如下：

```
#include<unistd.h>
int execve(const char * path, char * const argv[ ],char * const envp[ ]);
int execv(const char * path, char * const envp[ ]);
int execle(const char * path, const char * arg, …);
int execl(const char * path, const char * arg, …);
int exevp(const char *file,char * const argv[ ]);
int execlp(const char * file, const char * arg, …);
```

为了更好的理解 exec 函数族的使用，首先要理解环境变量这个概念。为了便于用户灵活地使用 Shell，Linux 引入了环境变量的概念，包括用户的主目录，终端类型、当前目录等，它们定义了用户的工作环境，所以称为环境变量。可以使用 env 命令查看环境变量值，用户也可以修改这些变量值以定制自己的工作环境，实例 7-6 演示了环境变量的应用。

例 7-6 env.c

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

extern char **environ;

int main (int argc,char * argv[ ])
{
    int i;

    printf("Argument:\n");
    for(i=0;i<argc;i++) {
        printf("argv[%d] is %s\n",i,argv[i]);
    }
```

我爱自由 5ifreedom.com

```
    printf("Environment:\n");

    for(i=0;environ[i]!=NULL;i++)
        printf("%s\n",environ[i]);

    return 0;
}
```

执行结果如下：

```
[zzy@mci src]$ ./env argv1 argv2 argv3
Argument:
argv[0] is ./env
argv[1] is argv1
argv[2] is argv2
argv[3] is argv3
Environment:
MANPATH=/usr/local/pgsql/man:
HOSTNAME=mci.uestc.edu.cn
TERM=linux
SHELL=/bin/bash
…
…
SSH_ASKPASS=/usr/libexec/openssh/gnome-ssh-askpass
PATH=/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/usr/local/pureftpd/bin:/usr/local/pureft
pd/sbin:/usr/java/jdk1.5.0_04/bin:/usr/java/jdk1.5.0_04/jre/bin:/usr/local/apache-ant-1.6.5/bi
n:/usr/local/pgsql/bin:/home/zzy/bin:.
PWD=/home/zzy/src
HOME=/home/zzy
LOGNAME=zzy
CLASSPATH=/usr/local/jdk1.5.0_04/lib/:/usr/local/jdk1.5.0_04/jre/lib/:.
```

程序说明。

Argument 后显示的是程序的命令行参数。Environment 后显示的是当前系统中各个环境变量的值，可以将其与命令 env 的输出结果作比较，将会发现两者的结果是一样的。

程序中通过系统预定义的全局变量 environ 显示各个环境变量的值。还可以通过另外一个方法得到环境变量值。

事实上 main 函数的完整形式应该是：

```
int main(int argc, char *argv[ ],char ** envp);
```

通过打印 main 的参数 envp，同样可以得到环境变量，感兴趣的读者不妨试一下。事实上无论是哪个 exec 函数，都是将可执行程序的路径、命令行参数和环境变量 3 个参数传递给可执行程序的 main 函数。

下面将分别介绍各 exec 函数是如何将 main 函数需要的参数传递给它的。

- **execv 函数**：execv 函数通过路径名方式调用可执行文件作为新的进程映像。它的 argv 参数用来提供给 main 函数的 argv 参数。argv 参数是一个以 NULL 结尾（最后一个元素必须是一个空指针）的字符串数组。
- **execve 函数**：在该系统调用中，参数 path 是将要执行的程序的路径名，参数 argv、envp 与 main 函数的 argv、envp 对应。

注意：参数 argv 和参数 envp 的大小都是受限制的。Linux 操作系统通过宏 ARG_MAX 来限制它们的大小，如果它们的容量之和超过 ARG_MAX 定义的值将会发生错误。这个宏定义在<linux/limits.h> 头文件中。

- **execl 函数**：该函数与 execv 函数用法类似。只是在传递 argv 参数的时候，每个命令行参数都声明为一个单独的参数（参数中使用“…”说明参数的个数是不确定的），要注意的是这些参数要以一个空指针作为结束。
- **execle 函数**：该函数与 execl 函数用法类似，只是要显式指定环境变量。环境变量位于命令行参数最后一个参数的后面，也就是位于空指针之后。

我爱自由 5ifreedom.com

- **execvp 函数**：该函数和 execv 函数用法类似，不同的是参数 filename。该参数如果包含“/”，就相当于路径名；如果不包含“/”，函数就到 PATH 环境变量定义的目录中寻找可执行文件。从程序 7-6 的运行结果中可以看到 PATH 环境变量的目录之间以冒号隔开。
- **execlp 函数**：该函数类似于 execl 函数，它们的区别和 execvp 与 execv 的区别一样。

exec 函数族的 6 个函数中只有 execve 是系统调用。其它 5 个都是库函数，它们实现时都调用了 execve。

正常情况下，这些函数是不会返回的，因为进程的执行映像已经被替换，没有接收返回值的地方了。如果有一个错误的事件，将会返回-1。这些错误通常是由文件名或参数错误引起的，其含义如表 7-2 所示。

表 7-2　　exec 函数错误表

errno	错 误 描 述
EACCES	指向的文件或脚本文件没有设置可执行位，即指定的文件是不可执行的
E2BIG	新程序的命令行参数与环境变量容量之和超过 ARG_MAX
ENOEXEC	由于没有正确的格式，指定的文件无法执行
ENOMEM	没有足够的内存空间来执行指定的程序
ETXTBUSY	指定文件被一个或多个进程以可写的方式打开
EIO	从文件系统读入文件时发生 I/O 错误

在 Linux 操作系统下，exec 函数族可以执行二进制的可执行文件，也可以执行 Shell 脚本程序，但 Shell 脚本必须以下面所示的格式开头：第一行必须为: #! interpretername [arg]。其中 interpretername 可以是 Shell 或其他解释器，例如，/bin/sh 或/usr/bin/perl，arg 是传递给解释器的参数。

下面通过例 7-7 演示 exec 函数的用法。

例 7-7　processimage.c 和 execve.c

```
程序一：用来替换进程映像的程序。
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc,char * argv[],char ** environ)
{
    int i;

    printf("I am a process image!\n");
    printf("My pid = %d, parentpid = %d\n",getpid(),getppid());
    printf("uid = %d,gid = %d\n",getuid(),getgid());

    for(i=0; i< argc; i++)
        printf("argv[%d]:%s\n",i ,argv[i]);
}

程序二：exec 函数实例，这里使用 execve 函数。
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc,char * argv[],char ** environ)
{
    pid_t    pid;
    int      stat_val;
```

我爱自由 5ifreedom.com

```
    printf("Exec example!\n");
    pid = fork();
    switch(pid) {
        case -1:
            perror("Process Creation failed\n");
            exit(1);
        case 0:
            printf("Child process is running\n");
            printf("My pid = %d ,parentpid = %d\n",getpid(),getppid());
            printf("uid = %d,gid =%d\n",getuid(),getgid());
            execve("processimage",argv, environ);
            printf("process never go to here!\n");
            exit(0);
        default:
            printf("Parent process is running\n");
            break;
    }

    wait(&stat_val);
    exit(0);
}
```

先编译第一个程序，生成可执行文件：gcc –o processimage processimage.c。再编译第二个程序：gcc –o execve execve.c，运行结果为：

```
[zzy@mci src]$ ./execve test execve
Exec example!
Child process is running
My pid = 15868 ,parentpid = 15867
uid = 519,gid =519
I am a process image!
My pid = 15868 ,parentpid = 15867
uid = 519,gid =519
argv[0]:./execve
argv[1]:test
argv[2]:execve
Parent process is running
```

程序说明。

从执行结果可以看到执行新程序的进程保持了原来进程的进程 ID、父进程 ID、实际用户 ID 和实际组 ID。同时我们还可以看到当调用新的可执行程序后，原有的子进程的映像被替代，不再被执行。子进程永远不会执行到 printf("process never go to here!\n")，因为子进程已经被新的执行映像所替代。

执行新程序后的进程除了保持了原来的进程 ID、父进程 ID、实际用户 ID 和实际组 ID 之外，进程还保持了许多原有特征，主要有。

- 当前工作目录。
- 根目录。
- 创建文件时使用的屏蔽字。
- 进程信号屏蔽字。
- 未决警告。
- 和进程相关的使用处理器的时间。
- 控制终端。
- 文件锁。

7.2.5 等待进程结束

前面提到当子进程先于父进程退出时，如果父进程没有调用 wait 和 waitpid 函数，子进程就

我爱自由 5ifreedom.com

会进入僵死状态。如果父进程调用了 wait 或 waitpid 函数，就不会使子进程变为僵尸进程，这两个函数的声明如下：

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *statloc, int options);
```

wait 函数使父进程暂停执行，直到它的一个子进程结束为止。该函数的返回值是终止运行的子进程的 PID。参数 statloc 所指向的变量存放子进程的退出码，即从子进程的 main 函数返回的值或子进程中 exit 函数的参数。如果 statloc 不是一个空指针，状态信息将被写入它指向的变量。

头文件 sys/wait.h 中定义了解读进程退出状态的宏。表 7-3 所示为各个宏的含义说明。

表 7-3　　检查 wait 和 waitpid 所返回的终止状态的宏

宏 定 义	说　明
WIFEXITED(stat_val)	若子进程是正常结束的，该宏返回一个非零值，表示真。若子进程异常结束，返回零，表示假
WEXITSTTUS(stat_val)	若 WIFEXITED 返回值非零，它返回子进程中 exit 或_exit 参数的低 8 位
WIFSIGNALED(stat_val)	若子进程异常终止，它就取得一个非零值，表示真
WTERMSIG(stat_val)	如果宏 WIFSIGNALED 的值非零，该宏返回使子进程异常终止的信号编号
WIFSTOPPED(stat_val)	若子进程暂停，它就取得一个非零值，表示真
WSTOPSIG(stat_val)	若 WIFSTOPPED 非零，它返回使子进程暂停的信号编号

waitpid 也用来等待子进程的结束，但它用于等待某个特定进程结束。参数 pid 指明要等待的子进程的 PID。pid 值的意义见表 7-4。参数 statloc 的含义与 wait 函数中的 statloc 相同。options 参数允许用户改变 waitpid 的行为，若将该参数赋值为 WNOHANG，则使父进程不被挂起而立即返回并执行其后的代码。

表 7-4　　aitpid 函数参数 pid 不同取值的意义

取　值	意　义
pid ＞ 0	等待其进程 ID 等于 pid 的子进程退出
pid = 0	等待其组 ID 等于调用进程的组 ID 的任一子进程
pid ＜ −1	等待其组 ID 等于 pid 绝对值的任一子进程
pid = −1	等待任一子进程

如果想让父进程周期性地检查某个特定的子进程是否已经退出，可以按如下方式调用 waitpid。

```
waitpid(child_pid, (int *) 0, WNOHANG);
```

如果子进程尚未退出，它将返回 0；如果子进程已经结束，则返回 child_pid。调用失败时返回−1。失败的原因包括没有该子进程、参数不合法等。

注意：wait 等待第一个终止的子进程，而 waitpid 则可以指定等待特定的子进程。waitpid 提供了一个 wait 的非阻塞版本。有时希望取得一个子进程的状态，但不想使父进程阻塞，waitpid 提供了一个这样的选项：WNOHANG，它可使调用者不阻塞。如果一个没有任何子进程的进程调用 wait 函数，会立即出错返回，感兴趣的读者可以编程测试，下面例 7-8 是等待进程的处理示例。

例 7-8　wait.c

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
```

我爱自由 5ifreedom.com

```
#include <unistd.h>

int main()
{
    pid_t    pid;
    char     *msg;
    int      k;
    int      exit_code;

    printf("Study how to get exit code\n");
    pid = fork();
    switch(pid) {
        case 0:
            msg = "Child process is running";
            k = 5;
            exit_code = 37;
            break;
        case -1:
            perror("Process creation failed\n");
            exit(1);
        default:
            exit_code = 0;
            break;
    }

    /*     父子进程都会执行以下这段代码
           子进程pid值为0，父进程pid值为子进程的ID  */
    if(pid != 0) {  // 父进程等待子进程结束
        int     stat_val;
        pid_t     child_pid;

        child_pid = wait(&stat_val);

        printf("Child procee has exited, pid = %d\n",child_pid);
        if(WIFEXITED(stat_val))
            printf("Child exited with code %d\n",WEXITSTATUS(stat_val));
           else
            printf("Child exited abnormally\n");
    }
    else {   // 子进程暂停5秒，在这个过程中可以运行命令ps aux查看父进程状态
        while(k-->0) {
            puts(msg);
            sleep(1);
        }
    }

    exit(exit_code);
}
```

程序运行结果如下：

```
[zzy@mci src]$ ./wait
Study how to get exit code
Child process is running
Child process is running
Child process is running
Child process is running
Child process is running
Child procee has exited, pid = 10433
Child exited with code 37
```

程序说明。

父进程调用 wait 后被挂起等待（此时打开另外一个终端，输入命令 ps aux 可以看到父进程的状态为 S），直到子进程结束为止。子进程正常结束后，wait 函数返回刚刚结束运行的子进程的 pid，宏 WEXITSTATUS 获取子进程的退出码。

我爱自由 5ifreedom.com

7.3 进程的其他操作

7.3.1 获得进程 ID

系统调用 getpid 用来获取当前进程的 ID。输入命令 man 2 getpid 可获得该函数的声明：

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
```

程序 7-9 是一个使用 getpid 简单的例子。

例 7-9 getpid.c

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
    pid_t  pid;

    if((pid = fork()) == -1)
    {
        printf("fork error!\n");
        exit(1);
    }

    if(pid == 0)
        printf("getpid return %d\n",getpid());

    exit(0);
}
```

运行程序结果如下：

```
[zzy@mci src]$ ./getpid
getpid return 5586
getpid 返回了子进程的 ID 号 5586。
```

7.3.2 setuid 和 setgid

可以用 setuid 设置实际用户 ID 和有效用户 ID。与此类似，setgid 用来设置实际组 ID 和有效组 ID。实际用户 ID、有效用户 ID 的含义请参考本章的 7.1.1 节。函数的声明在 unistd.h 头文件，声明如下：

```
#include <sys/types.h>
#include <unistd.h>
int setuid(uid_t uid);
int setgid(gid_t gid);
```

设置用户 ID 的 setuid 函数遵守以下规则（设置组 ID 的 setgid 函数与此类似）。

- 若进程具有 root 权限，则函数将实际用户 ID、有效用户 ID 设置为参数 uid。
- 若进程不具有 root 权限，但 uid 等于实际用户 ID，则 setuid 只将有效用户 ID 设置为 uid，不改变实际用户 ID。
- 若以上两个条件都不能满足，则函数调用失败，返回-1，并设置 errno 为 EPERM。

从以上规则可以看出，只有超级用户进程才能更改实际用户 ID。所以一个非特权用户进程是不能通过 setuid 或 seteuid 得到特权用户权限的。但是 su 命令却能使一个普通用户变成特权用户。这并不矛盾，因为 su 是一个“set_uid”程序。执行一个设置了 set_uid 位的程序时，内核将进程的有效用户 ID 设置为文件属主的 ID。而内核检查一个进程是否具有访问某文件的权限时，是使用进程的有效用户 ID 来

我爱自由 5ifreedom.com

进行检查的。su 程序的文件属主是 root，普通用户运行 su 命令时，su 进程的权限是 root 权限。

下面通过程序 7-10 来解释内核是如何检查进程是否有权访问某文件的。可以看到内核是通过检查进程的有效用户 ID 来检查的，而不是实际用户 ID。

例 7-10　studyuid.c

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>

extern int errno;

int main()
{
    int fd;

    printf("uid study: \n");
    printf("Process's uid = %d,euid = %d\n",getuid(),geteuid());

    // strerror 函数获取指定错误码的提示信息
    if((fd = open("test.c",O_RDWR)) == -1) {
            printf("Open failure,errno is %d :%s \n",errno, strerror(errno) );
            exit(1);
    }
    else {
            printf("Open successfully!\n");
    }

    close(fd);
    exit(0);
}
```

以 root 用户（提示符为#）使用命令在当前目录下创建一个名为 test.c 文件：touch test.c。在 root 用户下运行该程序，此时文件 test.c 的权限为-rw-r--r--，即 root 用户有读写权限，其他用户只可读。运行结果为：

```
[root@mci src]# ./studyuid
uid study:
Process's uid =0,euid = 0
Open sucessfully
```

转到另外一个普通用户下，运行的结果为：

```
[zzy@mci src]$ ./studyuid
uid study:
Process's uid =519,euid = 519
Open failure,errno is 13 : Permission denied
```

在 root 用户下，使用命令 chmod 对 studyuid 程序文件设置 set_uid 位：

```
[root@mci src]# chmod 4755 studyuid
```

然后再在用户 zzy 下运行 studyuid，运行结果如下：

```
[zzy@mci src]$ ./studyuid
uid study:
Process's uid =519,euid = 0
Open sucessfully
```

运行结果说明：内核对进程存取文件的许可权的检查，是通过考查进程的有效用户 ID 来实现的。

注意：对于调用了 setuid 函数的程序要格外小心，当进程的有效用户 ID 即 euid 是 root 用户时，如果调用 setuid 函数使 euid 为其他非 root 用户，则该进程从此就不具有超级用户权限了。

可以这样使用 setuid 函数：开始时某个程序需要 root 权限完成一些工作，但后续的工作不需要 root 权限。可以将该可执行程序文件设置 set_uid 位，并使得该文件的属主是 root。这样普通用户执行这个程序时，进程就具有了 root 权限，当不再需要 root 权限时，调用 setuid(getuid())恢复进程的实际用户 ID 和有效用户 ID 为执行该程序的普通用户的 ID。对于一些提供网络服务的程序，这样做是非常有必

我爱自由 5ifreedom.com

要的，否则就可能被攻击者利用，使攻击者控制整个系统。

对于设置了 set_uid 位的可执行程序也要注意，尤其是对那些属主是 root 的更要注意。因为 Linux 系统中 root 用户拥有最高权力。黑客们往往喜欢寻找设置了 set_uid 位的可执行程序的漏洞，这样的程序如果存在缓冲区溢出漏洞，并且该程序是一个网络程序，那么黑客可以从远程的地方轻松地利用该漏洞获得运行该漏洞程序的主机的 root 权限。即使这样的程序不是网络程序，那么也可以使本机上的恶意普通用户提升为 root 用户。

7.3.3 改变进程的优先级

可以通过设置进程的优先级来保证进程优先运行。在 Linux 下，通过系统调用 nice 可以改变进程的优先级。nice 系统调用用来改变调用进程的优先级。命令行下输入命令 man 2 nice 可以获得该函数的声明：

```
#include <unistd.h>
int nice(int increment);
```

在介绍 nice 系统调用的用法前，需要先了解两个重要的函数：getpriority 和 setpriority，它们声明如下：

```
#include <sys/resource.h>
int getpriority(int which,int who);
int setpriority(int which,int who,int prio);
```

- **getpriority 函数**：该函数返回一组进程的优先级。参数 which 和 who 组合确定返回哪一组进程的优先级。which 的可能取值以及 who 的意义如下。
  - PRIO_PROCESS：一个特定的进程，此时 who 的取值为进程 ID。
  - PRIO_PGRP：一个进程组的所有进程，此时 who 的取值为进程组 ID。
  - PRIO_USER：一个用户拥有的所有进程，此时参数 who 取值为实际用户 ID。

getpriority 函数如果调用成功返回指定进程的优先级，如果出错将返回-1，并设置 errno 的值。errno 可能的取值如下。

  - ESRCH：which 和 who 的组合与现存的所有进程均不匹配。
  - EINVAL：which 是个无效的值。

注意：当指定的一组进程的优先级不同时，getpriority 将返回其中优先级最低的一个。此外，当 getpriority 返回-1 时，可能是发生错误，也有可能是返回的是指定进程的优先级。区分它们的惟一方法是在调用 getpriority 前将 errno 清零。如果函数返回-1 且 errno 不为零，说明有错误产生。

- **setpriority 函数**：该函数用来设置指定进程的优先级。进程指定的方法与 getpriority 函数相同。如果调用成功，函数返回指定进程的优先级，出错则返回-1，并设置相应额 errno。除了产生与 getpriority 相同的两个错误外，还有可能产生以下错误。
  - EPERM：要设置优先级的进程与当前进程不属于同一个用户，并且当前进程没有 CAP_SYS_NICE 特许。
  - EACCES：该调用可能降低进程的优先级并且进程没有 CAP_SYS_NICE 特许。

通过以上两个函数，完全可以改变进程的优先级。nice 系统调用是它们的一种组合形式，nice 系统调用等价于：

```
int nice (int increment)
{
    int oldpro = getpriority (PRIO_PROCESS, getpid());
    return setpriority (PRIO_PROCESS, getpid(), oldpro + increment);
    }
```

例 7-11 演示了 nice 的使用方法。

我爱自由 5ifreedom.com

例 7-11　mynice.c

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/resource.h>
#include <sys/wait.h>

int main(void)
{
    pid_t    pid;
    int        stat_val = 0;
    int        oldpri, newpri;

    printf("nice study\n");

    pid = fork();
    switch(pid) {
        case 0:
            printf("Child is running,CurPid is %d, ParentPid is %d\n", pid, getppid());

            oldpri = getpriority(PRIO_PROCESS,0);
            printf("Old priority = %d\n",oldpri);

            newpri = nice(2);
            printf("New priority = %d\n",newpri);

            exit(0);
        case -1:
            perror("Process creation failed\n");
            break;
        default:
            printf("Parent is running,ChildPid is %d, ParentPid is %d\n", pid, getpid());
            break;
    }

    wait(&stat_val);
    exit(0);
}
```

运行结果如下：

```
[zzy@mci src]$ ./mynice
nice study
Child is running,CurPid is 0, ParentPid is 17360
Old priority = 0
Parent running,ChildPid is 17361, ParentPid is 17360
New priority = 2
```

可以看到子进程的优先级由原来的 0 变为 2。

7.4 编程实践：实现自己的 myshell

下面通过实现一个自己的 Shell 来实践本章的学习知识。这个简单的 Shell，其功能有：解释执行命令，支持输入输出重定向，支持管道，后台运行程序。

1. 运行本程序后，它支持以下命令格式

- 单个命令，如：ls。
- 带 1 到多个参数的命令，如 ls -l　/tmp。
- 带一个输出重定向的命令，如 ls -l　/　>　a。
- 带一个输入重定向的命令，如 wc -c　<　a。

我爱自由 5ifreedom.com

- 带一个管道的命令，如 ls -l / | wc -c 。
- 后台运行符&可加在以上各个命令的最后面。

如：ls &。

如：ls -l /tmp &。

如：ls -l / > a &。

如：wc –c < a &。

如：ls -l / | wc –c &。

- 输入 exit 或 logout 退出 myshell。

2．错误处理

- 输入错误的命令格式报错。
- 输入不存在的命令报错。

3．程序主函数的流程图如图 7-3 所示

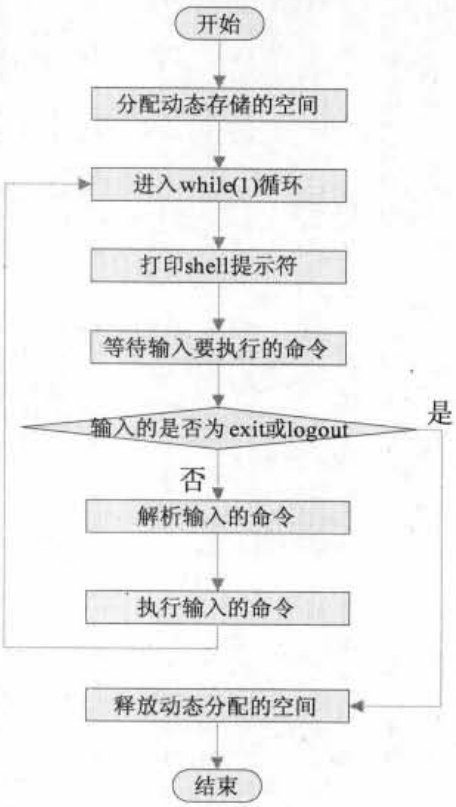


图 7-3 myshell.c 程序的流程图

4．关键函数的功能及说明

（1）void print_prompt()

函数说明：该函数只是简单地打印 myshell 的提示符，即“myshell$$”。

（2）void get_input(char *buf)

函数说明：获得一条用户输入的待执行的命令，参数 buf 用于存放输入的命令。如果输入的命令过长（大于 256 个字符），则终止程序。输入的命令以换行符“\n”作为结束标志。

（3）void explain_input(char *buf, int *argcount, char arglist[100][256])

函数说明：解析 buf 中存放的命令，把每个选项存放在 arglist 中。如输入命令“ls -l /tmp”，则 arglist[0]、arglist[1]、arglsit[2]指向的字符串分别为“ls”、“-l”、“/tmp”。

（4）do_cmd(int argcount, char arglist[100][256])

函数说明：执行 arglist 中存放的命令，argcount 为待执行命令的参数个数。

（5）int find_command (char *command)

函数说明：功能是分别在当前目录下、/bin、/usr/bin 目录下查找命令的可执行程序。

5．程序实现源代码（程序 7-12）

例 7-12 myshell.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <dirent.h>
```

我爱自由 5ifreedom.com

```
#define normal              0  /* 一般的命令 */
#define out_redirect        1  /* 输出重定向 */
#define in_redirect         2  /* 输入重定向 */
#define have_pipe           3  /* 命令中有管道 */

void print_prompt();                                    /* 打印提示符 */
void get_input(char *);                                 /* 得到输入的命令 */
void explain_input(char *, int *, char a[ ][ ]);        /* 对输入命令进行解析 */
void do_cmd(int, char a[ ][ ]);                         /* 执行命令 */
int  find_command(char *);                              /* 查找命令中的可执行程序 */

int main(int argc, char **argv)
{
    int    i;
    int    argcount = 0;
    char   arglist[100][256];
    char   **arg = NULL;
    char   *buf = NULL;

    buf = (char *)malloc(256);
    if( buf == NULL ) {
        perror("malloc failed");
        exit(-1);
    }

    while(1) {
        /* 将buf所指向的空间清零 */
        memset(buf, 0, 256);
        print_prompt();
        get_input(buf);
        /* 若输入的命令为exit或logout则退出本程序 */
        if( strcmp(buf,"exit\n") == 0 || strcmp(buf,"logout\n" ) == 0 )
            break;
        for (i=0; i < 100; i++)
        {
            arglist[i][0]='\0';
        }
        argcount = 0;
        explain_input(buf, &argcount, arglist);
        do_cmd(argcount, arglist);
    }

    if(buf != NULL) {
        free(buf);
        buf = NULL;
    }

    exit(0);
}

void print_prompt()
{
    printf("myshell$$ ");
}

/*获取用户输入*/
void get_input(char *buf)
{
    int len = 0;
    int ch;

    ch = getchar();
    while (len < 256 && ch != '\n') {
        buf[len++] = ch;
        ch = getchar();
    }

    if(len == 256) {
        printf("command is too long \n");
        exit(-1); /* 输入的命令过长则退出程序 */
```

我爱自由 5ifreedom.com

```
    }

    buf[len] = '\n';
    len++;
    buf[len] = '\0';
}

/* 解析buf中的命令，将结果存入arglist中，命令以回车符号\n结束 */
/*如输入命令为"ls -l /tmp"，则arglist[0]、arglist[1]、arglsit[2]分别为ls、-l和/tmp */
void explain_input(char *buf, int *argcount, char arglist[100][256])
{
    char    *p = buf;
    char    *q = buf;
    int number    = 0;

    while (1) {
        if ( p[0] == '\n' )
            break;

        if ( p[0] == ' ' )
            p++;
        else {
            q = p;
            number = 0;
            while( (q[0]!=' ') && (q[0]!='\n') ) {
                number++;
                q++;
            }
            strncpy(arglist[*argcount], p, number+1);
            arglist[*argcount][number] = '\0';
            *argcount = *argcount + 1;
            p = q;
        }
    }
}

void do_cmd(int argcount, char arglist[100][256])
{
    int         flag = 0;
    int         how = 0;          /* 用于指示命令中是否含有>、<、|   */
    int         background = 0; /* 标识命令中是否有后台运行标识符& */
    int         status;
    int         i;
    int         fd;
    char*       arg[argcount+1];
    char* argnext[argcount+1];
    char* file;
    pid_t       pid;

    /*将命令取出*/
    for (i=0; i < argcount; i++) {
        arg[i] = (char *) arglist[i];
    }
    arg[argcount] = NULL;

    /*查看命令行是否有后台运行符*/
    for (i=0; i < argcount; i++) {
        if (strncmp(arg[i], "&",1) == 0) {
            if (i == argcount-1) {
                background = 1;
                arg[argcount-1] = NULL;
                break;
            }
            else {
                printf("wrong command\n");
                return ;
            }
        }
    }
```

我爱自由 5ifreedom.com

```
    for (i=0; arg[i]!=NULL; i++) {
        if (strcmp(arg[i], ">") == 0 ) {
            flag++;
            how = out_redirect;
            if (arg[i+1] == NULL)
                flag++;
        }
        if ( strcmp(arg[i],"<") == 0 ) {
            flag++;
            how = in_redirect;
            if(i == 0)
                flag++;
        }
        if ( strcmp(arg[i],"|")==0 ) {
            flag++;
            how = have_pipe;
            if(arg[i+1] == NULL)
                flag++;
            if(i == 0 )
                flag++;
        }
    }
    /* flag 大于 1，说明命令中含有多个>，<，| 符号，本程序是不支持这样的命令的，
       或者命令格式不对，如"ls -l /tmp >" */
    if (flag > 1) {
        printf("wrong command\n");
        return;
    }

    if (how == out_redirect) {  /*命令只含有一个输出重定向符号> */
        for (i=0; arg[i] != NULL; i++) {
            if (strcmp(arg[i],">")==0) {
                file   = arg[i+1];
                arg[i] = NULL;
            }
        }
    }

    if (how == in_redirect) {    /*命令只含有一个输入重定向符号< */
        for (i=0; arg[i] != NULL; i++) {
            if (strcmp (arg[i],"<") == 0) {
                file   = arg[i+1];
                arg[i] = NULL;
            }
        }
    }

    if (how == have_pipe) {  /* 命令只含有一个管道符号| */
/* 把管道符号后门的部分存入 argnext 中，管道后面的部分是一个可执行的 Shell 命令 */
        for (i=0; arg[i] != NULL; i++) {
            if (strcmp(arg[i],"|")==0) {
                arg[i] = NULL;
                int j;
                for (j=i+1; arg[j] != NULL; j++) {
                    argnext[j-i-1] = arg[j];
                }
                argnext[j-i-1] = arg[j];
                break;
            }
        }
    }

    if ( (pid = fork()) < 0 ) {
        printf("fork error\n");
        return;
    }

    switch(how) {
        case 0:
```

我爱自由 5ifreedom.com

```
        /* pid 为 0 说明是子进程，在子进程中执行输入的命令 */
        /* 输入的命令中不含>、<和| */
        if (pid == 0) {
            if ( !(find_command(arg[0])) ) {
                printf("%s : command not found\n", arg[0]);
                exit (0);
            }
            execvp(arg[0], arg);
            exit(0);
        }
        break;
    case 1:
        /* 输入的命令中含有输出重定向符> */
        if (pid == 0) {
            if ( !(find_command(arg[0])) ) {
                printf("%s : command not found\n",arg[0]);
                exit(0);
            }
            fd = open(file,O_RDWR|O_CREAT|O_TRUNC,0644);
            dup2(fd,1);
            execvp(arg[0],arg);
            exit(0);
        }
        break;
    case 2:
        /* 输入的命令中含有输入重定向符< */
        if (pid == 0) {
            if ( !(find_command (arg[0])) ) {
                printf("%s : command not found\n",arg[0]);
                exit(0);
            }
            fd = open(file,O_RDONLY);
            dup2(fd,0);
            execvp(arg[0],arg);
            exit(0);
        }
        break;
    case 3:
        /* 输入的命令中含有管道符| */
        if(pid == 0) {
            int  pid2;
            int  status2;
            int  fd2;

            if ( (pid2 = fork()) < 0 ) {
                printf("fork2 error\n");
                return;
            }
            else if (pid2==0) {
                if ( !(find_command(arg[0])) ) {
                    printf("%s : command not found\n",arg[0]);
                    exit(0);
                }
            fd2 = open("/tmp/youdonotknowfile",
                        O_WRONLY|O_CREAT|O_TRUNC,0644);
                dup2(fd2, 1);
                execvp(arg[0], arg);
                exit(0);
            }

            if (waitpid(pid2, &status2, 0) == -1)
                printf("wait for child process error\n");

            if ( !(find_command(argnext[0])) ) {
                printf("%s : command not found\n",argnext[0]);
                exit(0);
            }
            fd2 = open("/tmp/youdonotknowfile",O_RDONLY);
```

我爱自由 5ifreedom.com

```
                dup2(fd2,0);
                execvp (argnext[0],argnext);

                if ( remove("/tmp/youdonotknowfile") )
                    printf("remove error\n");
                exit(0);
            }
            break;
        default:
            break;
    }

    /* 若命令中有&，表示后台执行，父进程直接返回，不等待子进程结束 */
    if ( background == 1 ) {
        printf("[process id %d]\n",pid);
        return ;
    }

    /* 父进程等待子进程结束 */
    if (waitpid (pid, &status,0) == -1)
        printf("wait for child process error\n");
}

/* 查找命令中的可执行程序 */
int find_command (char *command)
{
    DIR* dp;
    struct dirent*   dirp;
    char* path[] = { "./", "/bin", "/usr/bin", NULL};

    /* 使当前目录下的程序可以运行，如命令"./fork"可以被正确解释和执行 */
    if( strncmp(command,"./",2) == 0 )
        command = command + 2;

    /* 分别在当前目录、/bin 和/usr/bin 目录查找要执行的程序 */
    int i = 0;
    while (path[i] != NULL) {
        if ( (dp = opendir(path[i]) ) == NULL)
            printf ("can not open /bin \n");
        while ( (dirp = readdir(dp)) != NULL) {
            if (strcmp(dirp->d_name,command) == 0) {
                closedir(dp);
                return 1;
            }
        }
        closedir (dp);
        i++;
    }
    return 0;
}
```

7.5 习题

1. 进程中的全局数据段、局部数据段、静态数据段分别位于哪个内存地址空间？
2. 列举子进程与父进程属性的异同。
3. 如何创建一个后台进程？
4. 在多进程中，父子进程的运行顺序是怎样的，请用程序说明。
5. 有一全局变量 i 初值为 5，父进程对其进行加 1 操作，子进程看到的 i 的取值将会是多少
6. fork 函数和 vfork 函数在创建进程时有什么区别？
7. 编写一个多进程（3 个进程）的程序，要求父进程给每个子进程传递不同的参数。

我爱自由 5ifreedom.com

第 8 章　线程控制

本章主要介绍线程知识，包括操作系统中为何要引入线程，如何创建线程，线程终止时的注意事项，如何建立线程的私有数据，以及多线程程序中，如何进行线程同步。此外，本章还介绍软件开发中所必需的错误处理。

本章重点：

- 线程的优点。
- 线程的私有数据。
- 线程的同步方法。
- 出错处理。

本章难点：

- 线程的私有数据。
- 线程的同步方法。

8.1 线程和进程关系

第 7 章在介绍进程的知识时，提到了进程和线程的区别。线程是计算机中独立运行的最小单位，运行时占用很少的系统资源。由于每个线程占用的 CPU 时间是由系统分配的，因此可以把线程看成操作系统分配 CPU 时间的基本单位。在用户看来，多个线程是同时执行的，但从操作系统调度上看，各个线程是交替执行的。系统不停地在各个线程之间切换，每个线程只有在系统分配给它的时间片内才能取得 CPU 的控制权，执行线程中的代码。

注意，这里只是针对单 CPU 单核的情况，在多 CPU 多核的主机上，多个线程是可以同时运行的。

Linux 操作系统是支持多线程的，它在一个进程内生成了许多个线程。一个进程可以拥有一至多个线程。那么为什么在支持多进程的情况下又引入多线程呢？这是因为多线程相对于多进程有以下优点。

- 在多进程情况下，每个进程都有自己独立的地址空间，而在多线程情况下，同一进程内的线程共享进程的地址空间。因此，创建一个新的进程时就要耗费时间来为其分配系统资源，而创建一个新的线程花费的时间则要少得多。
- 在系统调度方面，由于进程地址空间独立而线程共享地址空间，线程间的切换速度要远远快过进程间的切换速度。
- 在通信机制方面，进程间的数据空间相互独立，彼此通信要以专门的通信方式进行，通信时必须经过操作系统。而同一进程内的多个线程共享数据空间，一个线程的数据可以直接提供给其他线程使用，而不必经过操作系统。因此，线程间的通信更加方便和省时。

线程以上的优点可以用两个字概括，就是“节约”：节约资源，节约时间。这些对操作系统的设计来说是非常重要的。除此之外，线程还具有如下优点。

- 可以提高应用程序的响应速度。在图像界面程序中，如果有一个非常耗时的操作，它会导致其他操作不能进行而等待这个操作，此时界面响应用户操作的速度会变得很慢。多

我爱自由 5ifreedom.com

线程环境下可以将这个非常耗时的操作由一个单独的线程来完成。这个线程在用完操作系统分配给它的时间片后，让出 CPU，这样其他操作便有机会执行了。

- 可以提高多处理器效率。现在许多计算机都是采用多核技术，在这种情况下，可以让多个线程在不同的处理器上同时运行，从而大大提高程序执行速度。因此，多线程更能发挥硬件的潜力。
- 可以改善程序的结构。对于要处理多个命令的应用程序，可以将对每个命令的处理设计成一个线程，从而避免设计成大程序时造成的程序结构复杂。

虽然线程在进程内部共享地址空间、打开的文件描述符等资源。但是线程也有其私有的数据信息，包括。

- 线程号（thread ID）：每个线程都有一个惟一的线程号一一对应。
- 寄存器（包括程序计数器和堆栈指针）。
- 堆栈。
- 信号掩码。
- 优先级。
- 线程私有的存储空间。

Linux 系统支持 POSIX 多线程接口，称为 pthread(Posix Thread 的简称)。编写 Linux 下的多线程应用程序，需要使用头文件 pthread.h，链接时需要使用库 libpthread.a。

8.2 创建线程

8.2.1 线程创建函数 pthread_create

前面的程序实例都是单线程的。单线程的程序都是按照一定的顺序执行的，如果在主线程里面创建线程，程序就会在创建线程的地方产生分支，变成两个程序执行。这似乎和多进程一样，其实不然。子进程是通过拷贝父进程的地址空间来实现的；而线程与进程内的线程共享程序代码，一段代码可以同时被多个线程执行。

线程的创建通过函数 pthread_create 来完成，该函数的声明如下：

```
# include <pthread.h>
int pthread_create (pthread_t *thread, pthread_attr_t * attr,
                    void* (*start_routine)(void *), void *arg);
```

函数各参数含义如下。

- **thread：**该参数是一个指针，当线程创建成功时，用来返回创建的线程 ID。
- **attr：**该参数用于指定线程的属性，NULL 表示使用默认属性，稍后将介绍该数据结构。
- **start_routine：**该参数为一个函数指针，指向线程创建后要调用的函数。这个被线程调用的函数也称为线程函数。函数指针的内容请参考第 4 章。
- **arg：**该参数指向传递给线程函数的参数。

注意：线程创建成功时，pthread_create 函数返回 0，若不为 0 则说明创建线程失败。常见的错误码为 EAGAIN 和 EINVAL。前者表示系统限制创建新的线程，例如，线程数目过多；后者表示第 2 个参数代表的线程属性值非法。线程创建成功后，新创建的线程开始运行第 3 个参数所指向的函数，原来的线程继续运行。

pthread_create 函数的第 2 个参数 attr 是一个指向 pthread_attr_t 结构体的指针，该结构体指明

我爱自由 5ifreedom.com

待创建线程的属性。线程属性将在 8.2.2 节详细介绍。

在头文件 pthread.h 中还声明了其他一些有用的系统调用，如表 8-1 所示。

表 8-1　　创建线程其他系统函数

函　数	说　明
pthread_t pthread_self(void)	获取本线程的线程 ID
int pthread_equal(pthread_t thread1, pthread_t thread2)	判断两个线程 ID 是否指向同一线程
int pthread_once(pthread_once_t *once_control, void (*init_routine) (void))	用来保证 init_routine 线程函数在进程中仅执行一次

下面通过例 8-1 讲述线程的创建过程。

例 8-1　createthread.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

int* thread(Void*arg)
{
    pthread_t newthid;

    newthid = pthread_self();
    printf("this is a new thread, thread ID = %u\n", newthid);
    return NULL;
}

int main(Void)
{
    pthread_t thid;

    printf("main thread ,ID is %u\n",pthread_self());  // 打印主线程的 ID
    if(pthread_create(&thid, NULL, (void *)thread, NULL) != 0) {
        printf("thread creation failed\n");
        exit(1);
    }
    sleep(1);
    exit(0);
}
```

编译并运行程序：

```
[zzy @mci ~]$ gcc -o createthread createthread.c -lpthread
[zzy@mci ~]$ ./createthread
main thread ,ID is 2505531040
this is a new thread, thread ID = 1084229984
```

程序说明。

程序首先打印出主线程的 ID，然后打印新创建的线程的 ID。

在某些情况下，函数执行次数要被限制为一次，这种情况下就要使用 pthread_once 函数。下面通过例 8-2 说明。该实例中创建两个线程，两个线程分别通过 pthread_once 调用同一个函数，结果被调用的函数只被执行了一次。

例 8-2　oncerun.c

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

pthread_once_t once = PTHREAD_ONCE_INIT;

void run(Void)
{
   printf("Fuction run is running in thread %u\n",pthread_self());
}
```

我爱自由 5ifreedom.com

```
void * thread1(Void*arg)
{
   pthread_t thid=pthread_self();
   printf("Current thread's ID is %u\n", thid);
   pthread_once(&once,run);
   printf("thread1 ends\n");
}

void * thread2(Void*arg)
{
   pthread_t thid=pthread_self();
   printf("Current thread's ID is %u\n", thid);
   pthread_once(&once, run);
   printf("thread2 ends\n");
}
int main()
{
   pthread_t thid1,thid2;

   pthread_create(&thid1,NULL,thread1,NULL);
   pthread_create(&thid2,NULL, thread2,NULL);
   sleep(3);
   printf("main thread exit! \n");
   exit(0);
}
```

编译并运行，结果如下：

```
[zzy@mci ~]$ gcc -o oncerun oncerun.c -lpthread
[zzy@mci ~]$ ./oncerun
Current thread's ID is 1084229984
Fuction run is running in thread 1084229984
thread1 ends
Current thread's ID is 1094719840
thread2 ends
main thread exit!
```

程序说明。

从运行结果可以看出函数 run 在线程 thread1 中运行了一次，线程 thread2 虽然也调用了 run 函数，但是并未执行。

8.2.2 线程属性

线程创建函数 pthread_create 有一个参数的类型为 pthread_attr_t，该结构体的定义如下：

```
typedef struct {
    int                     detachstate;detach:脱离
    int                     schedpolicy;
    struct sched_param      schedparam;
    int                     inheritsched;
    int                     scope;
    size_t                  guardsize;
    int                     stackaddr_set;
    void *                  stackaddr;
    size_t                  stacksize
} pthread_attr_t;
```

各个字段的含义如下。

- **detachstate：**表示新创建的线程是否与进程中其他的线程脱离同步。detachstate 的缺省值为 PTHREAD_CREATE_JOINABLE 状态，这个属性也可以用函数 pthread_ detach()来设置。如果将 detachstate 设置为 PTHREAD_CREATE_DETACH 状态，则 detachstate 不能再恢复到 PTHREAD_CREATE_JOINABLE 状态。
- **schedpolicy：**表示新线程的调度策略，主要包括 SCHED_OTHER（正常、非实时）、

我爱自由 5ifreedom.com

SCHED_RR（实时、轮转法）和 SCHED_FIFO（实时、先入先出）3 种，缺省为 SCHED_OTHER，后两种调度策略仅对超级用户有效。

- **schedparam：**一个 struct sched_param 结构，其中有一个 sched_priority 整型变量表示线程的运行优先级。这个参数仅当调度策略为实时（即 SCHED_RR 或 SCHED_ FIFO）时才有效，缺省为 0。

explicit：显示，明确的

- **inheritsched：有两种值可供选择，**PTHREAD_EXPLICIT_SCHED 和 PTHREAD_INHERIT_SCHED，前者表示新线程显式指定调度策略和调度参数（即 attr 中的值），而后者表示继承调用者线程的值。缺省为 PTHREAD_EXPLICIT_SCHED。

inherit：继承

- **scope：**表示线程间竞争 CPU 的范围，也就是说，线程优先级的有效范围。POSIX 的标准中定义了两个值，PTHREAD_SCOPE_SYSTEM 和 PTHREAD_SCOPE_PROCESS，前者表示与系统中所有线程一起竞争 CPU，后者表示仅与同进程中的线程竞争 CPU。
- **guardsize：**警戒堆栈的大小。
- **stackaddr_set：**堆栈地址集。
- **stackaddr：**堆栈的地址。
- **stacksize：**堆栈的大小。

Linux 下提供了这些状态的获取和设置函数。这些函数声明如下：

```
# include <pthread.h>
/*返回 detachstate 属性*/
int pthread_attr_getdetachstate(pthread_attr_t * attr,int * detachstate);
/*设置 detachstate 属性，将 attr 中的 detachstate 设置为提供的 detachstate */
int pthread_attr_set etachstate(pthread_attr_t * attr,int detachstate);
/*获取 schedplicy 属性*/
int pthread_attr_getschedpolicy(pthread_attr_t * attr, int * policy);
/*将*attr 中的 schedpolicy 设置为函数提供的 policy */
int pthread_attr_setschedpolicy(pthread_attr_t * attr, int policy);
/*获取 schedparam 属性*/
int pthread_attr_getschedparam(pthread_attr_t * attr, struct sched_param * schedparam);
/*将*attr 中的 schedpolicy 设置为函数提供的 policy */
int pthread_attr_setschedparam (pthread_attr_t * attr, struct sched_param schedparam);
/*获取 inheritsched 属性 */
int pthread_attr_getinheritsched (pthread_attr_t * attr, int * inherit);
/*将*attr 中的 inheritsched 设置为函数提供的 inherit*/
int pthread_attr_setinheritsched (pthread_attr_t * attr, int inherit);
/*获取 scope 属性 */
int pthread_attr_getscope (pthread_attr_t * attr, int * scope);
/*将*attr 中的 scope 设置为函数提供的 scope */
int pthread_attr_setscope (pthread_attr_t * attr, int scope);
/*获取 guardsize 属性 */
int pthread_attr_getguardsize (pthread_attr_t * attr, size_t* guardsize);
/*将*attr 中的 guardsize 设置为函数提供的 guardsize */
int pthread_attr_setguardsize (pthread_attr_t * attr, sze_t guardsize);
/*获取 stackaddr 属性 */
int pthread_attr_getstackaddr (pthread_attr_t * attr, void ** stackaddr);
/*将*attr 中的 stackaddr 设置为函数提供的 stackaddr */
int pthread_attr_setstackaddr (pthread_attr_t * attr, void * stackaddr);
/*获取 stacksize 属性 */
int pthread_attr_getstacksize (pthread_attr_t * attr, size_t* stacksize);
/*将*attr 中的 stacksize 设置为函数提供的 stacksize */
int pthread_attr_set stacksize (pthread_attr_t * attr, sze_t stacksize);
```

8.3 线程终止

Linux 下有两种方式可以使线程终止。第一种通过 return 从线程函数返回，第二种是通过调

我爱自由 5ifreedom.com

用函数 pthread_exit()使线程退出。pthread_exit 在头文件 pthread.h 中声明，该函数原型如下：

```
#include <pthread.h>
void pthread_exit(void * retval);
```

有两种特殊情况要注意：一种情况是，在主线程中，如果从 main 函数返回或是调用了 exit 函数退出主线程，则整个进程将终止，此时进程中所有线程也将终止，因此在主线程中不能过早地从 main 函数返回；另种情是如果主线程调用 pthread_exit 函数，则仅仅是主线程消亡，进程不会结束，进程内的其他线程也不会终止，直到所有线程结束，进程才会结束。

线程终止最重要的问题是资源释放的问题，特别是一些临界资源。临界资源在一段时间内只能被一个线程所持有，当线程要使用临界资源时需提出请求，如果该资源未被使用则申请成功，否则等待。临界资源使用完毕后要释放以便其他线程可以使用。例如，某线程要写一个文件，在写文件时一般不允许其他线程也对该文件执行写操作的，否则会导致文件数据混乱。这里的文件就是一种临界资源。临界资源为一个线程所独占，当一个线程终止时，如果不释放其占有的临界资源，则该资源会被认为还被已经退出的线程所使用，因而永远不会得到释放。如果一个线程在等待使用这个临界资源，它就可能无限的等待下去，这就形成了死锁，而这往往是灾难性的。

为此，Linux 系统提供了一对函数：pthread_cleanup_push()、pthread_cleanup_pop()用于自动释放资源。从 pthread_cleanup_push()的调用点到 pthread_cleanup_pop()之间的程序段中的终止动作（如调用 pthread_exit）都将执行 pthread_cleanup_push()所指定的清理函数。两个函数是以宏形式提供的：

```
#include <pthread.h>
#define pthread_cleanup_push(routine, arg)  \
{ struct _pthread_cleanup_buffer buffer;  \
        _pthread_cleanup_push(&buffer,(routine),(srg));
#define pthread_cleanup_pop \
       _pthread_cleanup_pop(&buffer,(exeute));}
```

注意：pthread_cleanup_push()带有一个“{”，而 pthread_cleanup_pop()带有一个“}”，因此这两个函数必须成对出现，且必须位于程序的同一代码段中才能通过编译。

线程终止时另外一个要注意的问题是线程间的同步问题。一般情况下，进程中各个线程的运行是相互独立的，线程的终止并不会相互通知，也不会影响其他线程，终止的线程所占用的资源不会随着线程的终止而归还系统，而是仍为线程所在的进程持有。正如进程之间可以使用 wait()系统调用来等待其他进程结束一样，线程也有类似的函数：pthread_join()函数。该函数也在头文件 pthread.h 中声明：

```
#include <pthread.h>
void pthread_exit(void* retval);
int pthread_join(pthread_t th, void* thread_return);
int pthread_detach(pthread_t th);
```

函数 pthread_join 用来等待一个线程的结束。pthread_join()的调用者将被挂起并等待 th 线程终止，如果 thread_return 不为 NULL，则*thread_return=retval。需要注意的是一个线程仅允许一个线程使用 pthread_join()等待它的终止，并且被等待的线程应该处于可 join 状态，即非 DETACHED 状态。DETACHED 状态是指对某个线程执行 pthread_detach()后其所处的状态。处于 DETACHED 状态的线程无法由 pthread_join()同步。

一个可“join”的线程所占用的内存仅当有线程对其执行了 pthread_join()后才会释放，因此为了避免内存泄漏，所有线程的终止时，要么已被设为 DETACHED，要么使用 pthread_join()来回收资源。

注意：一个线程不能被多个线程等待，否则第一个接收到信号的线程成功返回，其余调用 pthread_join()的线程返回错误代码 ESRCH。

下面是一个线程终止的实例。例 8-3 演示了主线程通过 pthread_join 等待辅助线程结束。

例 8-3　jointhread.c

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

我爱自由 5ifreedom.com

```
void assisthread(Void*arg)
{
    printf ("I am helping to do some jobs\n");
    sleep (3);
    pthread_exit (0);
}
int main(void)
{
    pthread_t       assistthid;
    int             status;

    pthread_create (&assistthid, NULL, (void *) assisthread, NULL);
    pthread_join(assistthid, (void *) &status);
    printf("assistthread's exit is caused %d\n", status);

    return 0;
}
```

编译并运行程序：

```
[zzy@mci ~]$ gcc -o jointhread jointhread.c -lpthread
[zzy@mci ~]$ ./jointhread
I am helping to do some jobs
assistthread's exit is caused 0.
```

程序说明。

从运行结果可以看出 pthread_join 会阻塞主线程，等待线程 assisthread 结束。pthread_exit 结束时的退出码是 0，pthread_join 得出 status 也为 0，两者是一致的。

8.4 私有数据

在多线程环境下，进程内的所有线程共享进程的数据空间，因此全局变量为所有线程共有。在程序设计中有时需要保存线程自己的全局变量，这种特殊的变量仅在某个线程内部有效。如常见的变量 errno，它返回标准的出错代码。errno 不应该是一个局部变量，几乎每个函数都应该可以访问它；但它又不能作为是一个全局变量，否则在一个线程里输出的很可能是另一个线程的出错信息，这个问题可以通过创建线程的私有数据（Thread-specific Data，或 TSD）来解决。在线程内部，线程私有数据可以被各个函数访问，但它对其他线程是屏蔽的。

线程私有数据采用了一种被称为一键多值的技术，即一个键对应多个数值。访问数据时都是通过键值来访问，好像是对一个变量进行访问，其实是在访问不同的数据。使用线程私有数据时，首先要为每个线程数据创建一个相关联的键。在各个线程内部，都使用这个公用的键来指代线程数据，但是在不同的线程中，这个键代表的数据是不同的。操作线程私有数据的函数主要有 4 个：pthread_key_create（创建一个键），pthread_setspecific(为一个键设置线程私有数据)，pthread_getspecific（从一个键读取线程私有数据），pthread_key_delete（删除一个键）。这几个函数的声明如下：

```
#include <pthread.h>
int   pthread_key_create (pthread_key_t *key, void (*destr_function) (void *));
int   pthread_setspecific (pthread_key_t  key, const void * pointer));
void* pthread_getspecific (pthread_key_t key);
int   pthread_key_delete (pthread_key_t key);
```

- **pthread_key_create**：从 Linux 的 TSD 池中分配一项，将其值赋给 key 供以后访问使用，它的第一个参数 key 为指向键值的指针，第二个参数为一个函数指针，如果指针不为空，则在线程退出时将以 key 所关联的数据为参数调用 destr_function()，释放分配的缓冲区。

我爱自由 5ifreedom.com

key 一旦被创建，所有线程都可访问它，但各线程可根据自己的需要往 key 中填入不同的值，这就相当于提供了一个同名而不同值的全局变量，一键多值。一键多值靠的是一个关键数据结构数组，即 TSD 池，其结构如下：

```
static struct pthread_key_struct pthread_keys [PTHREAD_KEYS_MAX] = { { 0, NULL } };
```

创建一个 TSD 就相当于将结构数组中的某一项设置为“in_use”，并将其索引返回给*key，然后设置 destructor 函数为 destr_function。

- **pthread_setspecific：**该函数将 pointer 的值(不是内容)与 key 相关联。用 pthread_ setspecific 为一个键指定新的线程数据时，线程必须先释放原有的线程数据用以回收空间。
- **pthread_getspecific:** 通过该函数得到与 key 相关联的数据。
- **pthread_key_delete：**该函数用来删除一个键，删除后，键所占用的内存将被释放。需要注意的是，键占用的内存被释放，与该键关联的线程数据所占用的内存并不被释放。因此，线程数据的释放必须在释放键之前完成。

例 8-4 将实现如何创建和使用线程的私有数据，具体代码如下所示。

例 8-4 tsd.c

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

pthread_key_t key;

void * thread2(void *arg)
{

    int tsd = 5;
    printf ("thread %d is running\n", pthread_self());
    pthread_setspecific (key, (void *) tsd);
    printf ("thread %d returns %d\n", pthread_self(), pthread_getspecific(key));
}
void * thread1(void *arg)
{
    int tsd = 0;
    pthread_t thid2;

    printf ("thread %d is running\n", pthread_self());
    pthread_setspecific (key, (void *) tsd);
    pthread_create (&thid2, NULL, thread2, NULL);
    sleep (5);
    printf ("thread %d returns %d\n", pthread_self(), pthread_getspecific(key));
}

int main(void)
{
    pthread_t thid1;
    printf ("main thread begins running\n");
    pthread_key_create (&key, NULL);
    pthread_create (&thid1, NULL, thread1, NULL);
    sleep (3);
    pthread_key_delete (key);
     printf ("main thread exit\n");
    return 0;
}
```

编译并执行，结果如下：

```
[zzy@mci ~]$ gcc -o tsd tsd.c -lpthread
[zzy@mci ~]$ ./tsd
main thread begins running
thread 1082363072 is running
thread 1090751552 is running
```

我爱自由 5ifreedom.com

```
thread 1090751552 returns 5
thread 1082363072 returns 0
main thread exit
```

程序说明。

程序中，主线程创建了线程 thread1，线程 thread1 创建了线程 thread2。两个线程分别将 tsd 作为线程私有数据。从程序运行结果可以看出，两个线程 tsd 的修改互不干扰，可以看出 thread2 先于 thread1 结束，线程在创建 thread2 后，睡眠 5s 等待 thread2 执行完毕。可以看出 thread2 对 tsd 的修改并没影响到 thread1 的 tsd 的取值。

8.5 线程同步

线程最大的特点就是资源的共享性，然而资源共享中的同步问题是多线程编程的难点。Linux 系统提供了多种方式处理线程间的同步问题，其中最常用的有互斥锁、条件变量和异步信号。下面将重点介绍这 3 种同步技术的使用。

8.5.1 互斥锁

互斥锁通过锁机制来实现线程间的同步。在同一时刻它通常只允许一个线程执行一个关键部分的代码。表 8-2 列举了操作互斥锁的几个函数。这些函数均声明在头文件 pthread.h 中。

表 8-2 互斥锁函数

函　数	功　能
pthread_mutex_init 函数	初始化一个互斥锁
pthread_mutex_destroy 函数	注销一个互斥锁
pthread_mutex_lock 函数	加锁，如果不成功，阻塞等待
pthread_mutex_unlock 函数	解锁
pthread_mutex_trylock 函数	测试加锁，如果不成功则立即返回，错误码为 EBUSY

使用互斥锁前必须先进行初始化操作。初始化有两种方式，一种是静态赋值法，将宏结构常量 PTHREAD_MUTEX_INITIALIZER 赋给互斥锁，操作语句如下：

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

另外一种方式是通过 pthread_mutex_init 函数初始化互斥锁，该函数的原型如下：

```
int pthread_mutex_init (pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);
```

函数中的参数 mutexattr 表示互斥锁的属性，如果为 NULL 则使用默认属性。互斥锁的属性及意义见表 8-3。

表 8-3 互斥锁的属性

属 性 值	意　义
PTHREAD_MUTEX_TIMED_NP	普通锁：当一个线程加锁后，其余请求锁的线程形成等待队列，解锁后按优先级获得锁
PTHREAD_MUTEX_RECURSIVE_NP	嵌套锁：允许一个线程对同一个锁多次加锁，并通过多次 unlock 解锁。如果是不同线程请求，则在解锁时重新竞争
PTHREAD_MUTEX_ERRORCHECK_NP	检错锁：在同一个线程请求同一个锁的情况下，返回 EDEADLK，否则执行的动作与类型 PTHREAD_MUTEX_TIMED_NP 相同
PTHREAD_MUTEX_ADAPTIVE_NP	适应锁：解锁后重新竞争

我爱自由 5ifreedom.com

初始化以后，就可以给互斥锁加锁了。加锁有两个函数：pthread_mutex_lock()和 pthread_mutex_trylock()。它们的原型如下：

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex) ;
```

用 pthread_mutex_lock()加锁时，如果 mutex 已经被锁住，当前尝试加锁的线程就会阻塞，直到互斥锁被其他线程释放。当 pthread_mutex_lock 函数返回时，说明互斥锁已经被当前线程成功加锁。pthread_mutex_trylock 函数则不同，如果 mutex 已经被加锁，它将立即返回，返回的错误码为 EBUSY，而不是阻塞等待。

注意：加锁时，不论哪种类型的锁，都不可能被两个不同的线程同时得到，其中一个必须等待解锁。在同一进程中的线程，如果加锁后没有解锁，则其他线程将无法再获得该锁。

函数 pthread_mutex_unlock 用来解锁，函数原型如下：

```
int pthread_mutex_unlock(pthread_mutex_t *mutex) ;
```

用 pthread_mutex_unlock 函数解锁时，要满足两个条件：一是互斥锁必须处于加锁状态，二是调用本函数的线程必须是给互斥锁加锁的线程。解锁后如果有其他线程在等待互斥锁，等待队列中的第一个线程将获得互斥锁。

当一个互斥锁使用完毕后，必须进行清除。清除互斥锁使用函数 pthread_mutex_destroy，该函数原型如下：

```
int pthread_mutex_destroy (pthread_mutex_t *mutex);
```

清除一个互斥锁意味着释放它所占用的资源。清除锁时要求当前处于开放状态，若锁处于锁定状态，函数返回 EBUSY，该函数成功执行时返回 0。由于在 Linux 中，互斥锁并不占用内存，因此 pthread_mutex_destroy()除了解除到斥锁的状态以外没有其他操作。

下面的两个函数展示了如何通过互斥锁保护全局变量，代码如下所示：

```
pthread_mutex_t    number_mutex;
int                globalnumber;
void write_ globalnumber()
{
   pthread_mutex_lock (&number_mutex);
   globalnumber ++;
   pthread_mutex_unlock (&number_mutex);
}
int read_ globalnumber()
{
int temp;

pthread_mutex_lock (&number_mutex);
   temp = globalnumber;
   pthread_mutex_unlock (&number_mutex);
return (temp);
}
```

在上述代码中，两个函数对共享全局变量 globalnumber 进行读写操作。write_ globalnumber 函数使用互斥锁保证在修改变量的时候操作一次执行完毕，不会中断。而 read_ globalnumber 函数使用互斥锁保证在读数据的时候，全局变量 globalnumber 不会被修改，确保读到正确的数据。

8.5.2 条件变量

条件变量是利用线程间共享的全局变量进行同步的一种机制。条件变量宏观上类似 if 语句，符合条件就能执行某段程序，否则只能等待条件成立。

使用条件变量主要包括两个动作：一个等待使用资源的线程等待“条件变量被设置为真”；另

我爱自由 5ifreedom.com

一个线程在使用完资源后“设置条件为真”，这样就可以保证线程间的同步了。这样就存在一个关键问题，就是要保证条件变量能被正确的修改，条件变量要受到特殊的保护，实际使用中互斥锁扮演着这样一个保护者的角色。Linux 也提供了一系列对条件变量操作的函数，如表 8-4 所示。

表 8-4　　操作条件变量的函数

函　数	功　能
pthread_cond_init 函数	初始化条件变量
pthread_cond_wait 函数	基于条件变量阻塞，无条件等待
pthread_cond_timedwait 函数	阻塞直到指定事件发生，计时等待
pthread_cond_signal 函数	解除特定线程的阻塞，存在多个等待线程时按入队顺序激活其中一个
pthread_cond_broadcast 函数	解除所有线程的阻塞
pthread_cond_destroy 函数	清除条件变量

与互斥锁一样，条件变量的初始化也有两种方式，一种是静态赋值法，将宏结构常量 PTHREAD_COND_INITIALIZER 赋予互斥锁，操作语句如下：

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

另一种方式是使用函数 pthread_cond_init，它的原型如下：

```
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr);
```

其中，cond_attr 参数是条件变量的属性，由于其并没有得到实现，所以它的值通常是 NULL。

等待条件成立有两个函数：pthread_cond_wait 和 pthread_cond_timedwait。它们的原型如下：

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime);
```

pthread_cond_wait 函数释放由 mutex 指向的互斥锁，同时使当前线程关于 cond 指向的条件变量阻塞，直到条件被信号唤醒。通常条件表达式在互斥锁的保护下求值，如果条件表达式为假，那么线程基于条件变量阻塞。当一个线程改变条件变量的值时，条件变量获得一个信号，使得等待条件变量的线程退出阻塞状态。

pthread_cond_timedwait 函数和 pthread_cond_wait 函数用法类似，差别在于 pthread_cond_timedwait 函数将阻塞直到条件变量获得信号或者经过由 abstime 指定的时间，也就是说，如果在给定时刻前条件没有满足，则返回 ETIMEOUT，结束等待。

线程被条件变量阻塞后，可通过函数 pthread_cond_signal 和 pthread_cond_broadcast 激活，它们的原型如下：

```
int pthread_cond_signal (pthread_cond_t *cond);
int pthread_cond_broadcast (pthread_cond_t *cond);
```

pthread_cond_signal()激活一个等待条件成立的线程，存在多个等待线程时，按入队顺序激活其中一个；而 pthread_cond_broadcast()则激活所有等待线程。

当一个条件变量不再使用时，需要将其清除。清除一个条件变量通过调用 pthread_cond_destroy()实现，函数原型如下：

```
int pthread_cond_destroy(pthread_cond_t *cond)
```

pthread_cond_destroy 函数清除由 cond 指向的条件变量。注意：只有在没有线程等待该条件变量的时候才能清除这个条件变量，否则返回 EBUSY。

下面通过例 8-5 演示条件变量的使用方法。在例子中，有两个线程被启动，并等待同一个条件变量。

我爱自由 5ifreedom.com

例 8-5　condition.c

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

pthread_mutex_t    mutex;
pthread_cond_t     cond;

void * thread1(void *arg)
{
    pthread_cleanup_push (pthread_mutex_unlock, &mutex);

    while(1) {
        printf ("thread1 is running\n");
        pthread_mutex_lock (&mutex);
        pthread_cond_wait (&cond, &mutex);
        printf ("thread1 applied the condition\n");
        pthread_mutex_unlock (&mutex);
        sleep (4);
    }

    pthread_cleanup_pop (0);
}

void *thread2(void *arg)
{
    while(1) {
        printf ("thread2 is running\n");
        pthread_mutex_lock (&mutex);
        pthread_cond_wait (&cond, &mutex);
        printf ("thread2 applied the condition\n");
        pthread_mutex_unlock (&mutex);
        sleep (1);
    }
}

int main(void)
{
    pthread_t tid1, tid2;

    printf ("condition variable study! \n");
    pthread_mutex_init (&mutex, NULL);
    pthread_cond_init (&cond, NULL);
    pthread_create (&tid1, NULL, (void *) thread1, NULL);
    pthread_create (&tid2, NULL, (void *) thread2, NULL);

    do {
        pthread_cond_signal (&cond);
    } while (1);

    sleep (50);
    pthread_exit (0);
}
```

编译执行程序，结果片段如下：

```
[zzy@mci ~]$ gcc -o condition condition.c -lpthread
[zzy@mci ~]$ ./condition
condition variable study!
thread1 is running
thread2 is running
thread2 applied the condition
thread1 applied the condition
thread2 is running
thread2 applied the condition
thread2 is running
thread2 applied the condition
```

我爱自由 5ifreedom.com

```
thread1 is running
thread1 applied the condition
```

程序说明。

从运行结果看，thread1 和 thread2 通过条件变量同步运行。在线程函数 thread1 中可以看到条件变量使用时要配合互斥锁使用，这样可以防止多个线程同时请求 pthread_cond_wait()。调用 pthread_cond_wait()前必须由本线程加锁（pthread_mutex_lock()），而在更新条件等待队列以前，mutex 保持锁定状态，并在线程挂起前解锁。在条件满足离开 pthread_cond_wait()之前，mutex 将被重新加锁，与进入 pthread_cond_wait()前的加锁动作对应。

需要注意的是函数 pthread_cleanup_push 和 pthread_cleanup_pop，它们提供回调函数保护。pthread_cond_wait()和 pthread_cond_timedwait()都被实现为取消点，因此，在该处等待的线程将立即重新运行，在重新锁定 mutex 后退出 pthread_cond_wait()，然后执行取消动作。也就是说，如果 pthread_cond_wait()被取消，mutex 将依然保持锁定状态，那么 thread1 就需要定义退出回调函数来为其解锁。

8.5.3 异步信号

在 Linux 操作系统中，线程是在内核外实现的，它不像进程那样在内核中实现。Linux 线程本质上是轻量级的进程。第 9 章将介绍的信号可以被进程用来进行相互通信，一个进程通过信号通知另一个进程发生了某事件，比如该进程所需要的输入数据已经就绪。线程同进程一样也可以接收和处理信号，信号也是一种线程间同步的手段。

信号（如 SIGINT 和 SIGIO）与任何线程都是异步的，也就是说信号到达线程的时间是不定的。如果有多个线程可以接收异步信号，则只有一个被选中。如果并发的多个同样的信号被送到一个进程，每一个将被不同的线程处理。如果所有的线程都屏蔽该信号，则这些信号将被挂起，直到有信号解除屏蔽来处理它们。

Linux 多线程扩展函数中有三个函数用于处理异步信号：

```
int  pthread_kill (pthread_t threadid, int signo);
int  pthread_sigmask (int how, const sigset_t *newmask, sigset_t *oldmask);
int  sigwait (const sigset_t *set, int *sig);
```

其中，函数 pthread_kill 用来向特定的线程发送信号 signo。函数 pthread_sigmask 用来设置线程的信号屏蔽码，但对不允许屏蔽的 Cancel 信号和不允许响应的 Restart 信号进行了保护。函数 sigwait 用来阻塞线程，等待 set 中指定的信号之一到达，并将到达的信号存入*sig 中。

8.6 出错处理

在软件开发中，如果忽略了出错处理，往往会给软件产品带来灾难性的后果。因此，在设计和编写程序时，要时刻注意检查错误发生的各种可能，比如创建进程失败，打开文件失败等。当这些错误发生时，程序应当给出提示信息或进行相应的处理。软件开发中，通过编写出错处理代码可以快速地发现问题所在。

8.6.1 错误检查

函数执行失败时，一般都会返回一个特定的值，比如-1、空指针。这些值只能说明有错误发

我爱自由 5ifreedom.com

生，但错误的原因并没有说明。头文件 errno.h 定义了变量 errno（含义是 error number）,它存储了错误发生时的错误码，通过错误码可以得到错误的描述信息。以下是 errno.h 文件的一部分：

```
#include <errno.h>
#ifndef errno
extern int errno;
#endif
```

程序开始执行时，变量 errno 被初始化为 0。很多库函数在执行过程中遇到错误时就会将 errno 设置为相应的错误码。函数被成功调用时，它们不修改 errno 的值。因此，当一个函数被成功调用，errno 的值可能不为零，它的非零值由前面的函数设置。所以不能根据 errno 的值来判断一个函数执行是否成功。当函数调用失败时（函数返回-1 或 NULL），errno 值才有意义。

以下是一个程序实例。该实例 8-6 中通过打开一个文件，当由于某种原因该文件不能被打开时，就可以得到一个相应的 errno 值，检查其对应的错误码，可以得到错误的原因。

例 8-6 checkerrno.c

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

int main()
{
    FILE    *stream;
    char        * filename = "test";

    errno = 0;

    stream = fopen (filename, "r");
    if (stream == NULL) {
        printf ("open file %s failed, errno is %d\n", filename,errno);
    }
    else
        printf ("open file %s successfully\n", filename);
}
```

程序说明。

程序中使用 C 语言的库函数 fopen 打开名为“test”的文件，如果打开失败就打印出相应的 errno 值。我们可以使用各种方法使该文件不能被打开，例如，让该文件不存在或者让该文件不具有读写权限。以下是 test 文件不存在时的运行结果。

```
[zzy@mci ~]$ ./checkerrno
open file test failed, errno is 2
```

由 errno 的值可以查出 2 对应的错误码 ENOENT，得出出错原因是：文件或目录不存在。读者可以自己尝试让 test 文件不具有读写权限，然后运行该程序（得到 errno 的值为 13，对应错误码 EPERM）。

8.6.2 错误码

错误码是一些定义在 errno.h 中的宏，通常以字母 E(代表 error)开头，后面由一串大写字母或数字组成。

注意：在定义自己的宏时，要避免与这些保留的宏名冲突。除了 EWOULDBLOCK 和 EAGAIN 具有相同的值，其余所有的错误码的值都是非负且惟一的，因此在 switch 语句中可以使用它们。

部分常见的错误码解释如下。

- **ENOMEM**：表示内存不足，系统不能再提供更多的虚拟内存。
- **EIO**：输入输出错误，在读写磁盘时经常会遇到。

我爱自由 5ifreedom.com

- **ENXIO**：指定的设备或地址不存在。
- **EPERM**：禁止操作，只有具备相应权限的进程才能执行该操作。
- **ESRCH**：没有进程与给定的进程 ID 相匹配。
- **ENOENT**：文件或目录不存在。
- **EINTR**：函数调用被中断，如果发生这种错误，要重新调用函数。
- **E2BIG**：参数过长。
- **ENOEXEC**：可执行文件格式无效。
- **EBADF**：文件描述符错误。
- **ECHILD**：子进程不存在。
- **EBUSY**：资源正在使用，不能共享。
- **EINVAL**：无效的参数。
- **EMFILE**：当前进程打开的文件已达上限，不能再打开其他文件。
- **ENFILE**：系统打开的文件已达上限。
- **EFBIG**：文件太大。
- **ENOTDIR**：当需要目录的时候指定了一个非目录的文件。
- **EISDIR**：文件是一个目录。
- **ENOTTY**：不适当的 I/O 控制操作。
- **ETXTBSY**：尝试执行一个正在进行写操作的文件或者尝试写一个正在执行的文件。
- **ENOSPC**：设备上无剩余空间。

8.6.3 错误的提示信息

当程序出现错误时，可以打印出相应的错误提示信息，以便用户修改该错误。函数 strerror 和 perror 可以通过错误码获取标准的错误提示信息。下面分别介绍它们的用法：

strerror 函数在头文件 string.h 中声明：

```
#include <string.h>
char* strerror (int errnum);
```

strerror 函数根据参数 errnum 提供的错误码获取一个描述错误信息的字符串，函数的返回值为指向该字符串的指针。errnum 的值通常就是 errno。

perror()函数声明在头文件 stdio.h 中：

```
#include <stdio.h>
void perror (const char *message);
```

perror()打印错误信息到 stderr，stderr 在 Linux 中通常就是指屏幕或命令行终端。调用 perror()时，如果参数 message 是一个空指针，perror 仅仅根据 errno 打印出对应的错误提示信息。如果提供一个非空的值，perro 会把此 message 加在其输出信息的前面。perror 会添加一个冒号和空格将 message 和错误信息分开，以便区分。

这里给出一个例 8-7 来演示如何输出错误提示信息。

例 8-7 errshow.c

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

我爱自由 5ifreedom.com

```
FILE * open_file (char * filename)
{
    FILE *stream;
    errno = 0;

    stream = fopen (filename, "r");
    if (stream == NULL)
    {
        printf ("can not open the file %s. reason: %s\n", filename, strerror(errno));
        exit (-1);
    }
    else
        return stream;
}

int main(void)
{
    char *filename = "test";

    open_file (filename);
    return 0;
}
```

程序说明。

该程序调用函数 open_file 打开文件 test，当打开失败时，将打印出相应的错误提示信息。错误提示信息由函数 strerror 根据 errno 获得。以下是当文件不存在时的运行结果。

```
[zzy@mci ~]$ ./errshow
couldn't open file test. reason:No such file or directory
```

可以看到，运行结果中给出的错误提示为文件或目录不存在。在当前目录建立一个 test 文件并屏蔽其读权限后运行该程序，结果如下：

```
[zzy@mci ~]$ chmod 333 test
[zzy@mci ~]$ ./errshow
couldn't open file test. reason:Permission denied
```

从运行结果可以看出，此时文件打开失败的错误原因为操作被拒绝。可见给出准确的错误提示信息，有利于快速发现并修改错误。

8.7 习题

1. 多线程与多进程相比有哪些优势？

2. 编写一个多线程程序：要求主线程创建 3 个子线程，3 个子线程在执行时都修改一个它们共享的变量，观察共享变量的值，看看可以得出什么结论。

3. 如何在多线程程序中实现各线程之间的同步，有哪些方式？请编写一个测试程序。

4. 列出创建线程私有数据的步骤，并通过编写程序实现。

5. 编写一个多进程多线程的程序：要求创建 4 个子进程，每个子进程都分别创建两个线程，进程和线程的功能不做要求，可只提供简单的打印语句。

我爱自由 5ifreedom.com

第 9 章　信号及信号处理

本章将介绍 Linux 下与信号相关的知识，如信号的来源、种类以及进程对信号的响应等。在此基础上对信号的捕获、发送和屏蔽作了深入的分析。最后以两个实例程序演示信号的综合运用。

本章重点：

- 信号的来源、种类。
- 进程对信号的响应。
- 信号的捕捉和处理。
- 信号的发送、屏蔽。

本章难点：

- 信号处理时的原子操作。
- 利用信号在进程间传递参数。

9.1 Linux 信号介绍

信号（signal）是一种软件中断，它提供了一种处理异步事件的方法，也是进程间惟一的异步通信方式。在 Linux 系统中，根据 POSIX 标准扩展以后的信号机制，不仅可以用来通知某进程发生了什么事件，还可以给进程传递数据。

9.1.1 信号的来源

信号的来源可以有很多种方式，按照产生条件的不同可以分为硬件和软件两种方式。

1. 硬件方式

- 当用户在终端上按下某些键时，将产生信号。如按下<Ctrl+C>组合键后将产生一个 SIGINT 信号。
- 硬件异常产生信号：除数为 0、无效的存储访问等。这些事件通常由硬件（如 CPU）检测到，并将其通知给 Linux 操作系统内核，然后内核生成相应的信号，并把信号发送给该事件发生时正在运行的程序。

2. 软件方式

- 用户在终端下调用 kill 命令向进程发送任意信号。
- 进程调用 kill 或 sigqueue 函数发送信号。
- 当检测到某种软件条件已经具备时发出信号，如由 alarm 或 settimer 设置的定时器超时时将生成 SIGALRM 信号。

9.1.2 信号的种类

在 Shell 下输入 kill-l 可显示 Linux 系统支持的全部信号，表 9-1 列出了全部信号。

```
[root@localhost test]# kill -l
```

我爱自由 5ifreedom.com

表 9-1　　信号种类

信　号			
(1) SIGHUP	(2) SIGINT	(3) SIGQUIT	(4) SIGILL
(5) SIGTRAP	(6) SIGABRT	(7) SIGBUS	(8) SIGFPE
(9) SIGKILL	(10) SIGUSR1	(11) SIGSEGV	(12) SIGUSR2
(13) SIGPIPE	(14) SIGALRM	(15) SIGTERM	(17) SIGCHLD
(18) SIGCONT	(19) SIGSTOP	(20) SIGTSTP	(21) SIGTTIN
(22) SIGTTOU	(23) SIGURG	(24) SIGXCPU	(25) SIGXFSZ
(26) SIGVTALRM	(27) SIGPROF	(28) SIGWINCH	(29) SIGIO
(30) SIGPWR	(31) SIGSYS	(34) SIGRTMIN	(35) SIGRTMIN+1
(36) SIGRTMIN+2	(37) SIGRTMIN+3	(38) SIGRTMIN+4	(39) SIGRTMIN+5
(40) SIGRTMIN+6	(41) SIGRTMIN+7	(42) SIGRTMIN+8	(43) SIGRTMIN+9
(44) SIGRTMIN+10	(45) SIGRTMIN+11	(46) SIGRTMIN+1247)	SIGRTMIN+13
(48) SIGRTMIN+14	(49) SIGRTMIN+15	(50) SIGRTMAX-14	(51) SIGRTMAX-13
(52) SIGRTMAX-12	(53) SIGRTMAX-11	(54) SIGRTMAX-10	(55) SIGRTMAX-9
(56) SIGRTMAX-8	(57) SIGRTMAX-7	(58) SIGRTMAX-6	(59) SIGRTMAX-5
(60) SIGRTMAX-4	(61) SIGRTMAX-3	(62) SIGRTMAX-2	(63) SIGRTMAX-1
(64) SIGRTMAX			

信号的值在 signal.h 中定义，上面这些信号的含义如下。

（1）SIGHUP：当用户退出 Shell 时，由该 Shell 启动的所有进程将收到这个信号，默认动作为终止进程。

（2）SIGINT：用户按下了<Ctrl+C>组合键时，用户终端向正在运行中的由该终端启动的程序发出此信号。默认动作为终止进程。

（3）SIGQUIT：当用户按下<Ctrl+\>组合键时产生该信号，用户终端向正在运行中的由该终端启动的程序发出此信号。默认动作为终止进程并产生 core 文件。

（4）SIGILL：CPU 检测到某进程执行了非法指令。默认动作为终止进程并产生 core 文件。

（5）SIGTRAP：该信号由断点指令或其他 trap 指令产生。默认动作为终止进程并产生 core 文件。

（6）SIGABRT：调用 abort 函数时产生该信号。默认动作为终止进程并产生 core 文件。

（7）SIGBUS：非法访问内存地址，包括内存地址对齐（alignment）出错，默认动作为终止进程并产生 core 文件。

（8）SIGFPE：在发生致命的算术运算错误时发出。不仅包括浮点运算错误，还包括溢出及除数为 0 等所有的算术错误。默认动作为终止进程并产生 core 文件。

（9）SIGKILL：无条件终止进程。本信号不能被忽略、处理和阻塞。默认动作为终止进程。它向系统管理员提供了一种可以杀死任何进程的方法。

（10）SIGUSR1：用户定义的信号，即程序员可以在程序中定义并使用该信号。默认动作为终止进程。

（11）SIGSEGV：指示进程进行了无效的内存访问。默认动作为终止进程并产生 core 文件。

（12）SIGUSR2：这是另外一个用户自定义信号，程序员可以在程序中定义并使用该信号。默认动作为终止进程。

我爱自由 5ifreedom.com

（13）SIGPIPE：Broken pipe：向一个没有读端的管道写数据。默认动作为终止进程。

（14）SIGALRM：定时器超时，超时的时间由系统调用 alarm 设置。默认动作为终止进程。

（15）SIGTERM：程序结束（terminate）信号，与 SIGKILL 不同的是，该信号可以被阻塞和处理。通常用来要求程序正常退出。执行 Shell 命令 kill 时，缺省产生这个信号。默认动作为终止进程。

（16）SIGCHLD：子进程结束时，父进程会收到这个信号。默认动作为忽略该信号。

（17）SIGCONT：让一个暂停的进程继续执行。

（18）SIGSTOP：停止（stopped）进程的执行。注意它和 SIGTERM 以及 SIGINT 的区别：该进程还未结束，只是暂停执行。本信号不能被忽略、处理或阻塞。默认动作为暂停进程。

（19）SIGTSTP：停止进程的运行，但该信号可以被处理和忽略。按下<Ctrl+Z>组合键时发出这个信号。默认动作为暂停进程。

（20）SIGTTIN：当后台进程要从用户终端读数据时，该终端中的所有进程会收到 SIGTTIN 信号。默认动作为暂停进程。

（21）SIGTTOU：该信号类似于 SIGTTIN，在后台进程要向终端输出数据时产生。默认动作为暂停进程。

（22）SIGURG：套接字（socket）上有紧急数据时，向当前正在运行的进程发出此信号，报告有紧急数据到达。默认动作为忽略该信号。

（23）SIGXCPU：进程执行时间超过了分配给该进程的 CPU 时间，系统产生该信号并发送给该进程。默认动作为终止进程。

（24）SIGXFSZ：超过文件最大长度的限制。默认动作为终止进程并产生 core 文件。

（25）SIGVTALRM：虚拟时钟超时时产生该信号。类似于 SIGALRM，但是它只计算该进程占用的 CPU 时间。默认动作为终止进程。

（26）SIGPROF：类似于 SIGVTALRM，它不仅包括该进程占用的 CPU 时间还包括执行系统调用的时间。默认动作为终止进程。

（27）SIGWINCH：窗口大小改变时发出。默认动作为忽略该信号。

（28）SIGIO：此信号向进程指示发生了一个异步 IO 事件。默认动作为忽略。

（29）SIGPWR：关机。默认动作为终止进程。

（30）SIGSYS：无效的系统调用。默认动作为终止进程并产生 core 文件。

（31）SIGRTMIN～（64）SIGRTMAX：Linux 的实时信号，它们没有固定的含义（或者说可以由用户自由使用）。注意，Linux 线程机制使用了前 3 个实时信号。所有的实时信号的默认动作都是终止进程。

1．可靠信号与不可靠信号

SIGHUP（1 号）至 SIGSYS（31 号）之间的信号都是继承自 UNIX 系统，是不可靠信号。Linux 系统根据 POSIX.4 标准定义了 SIGRTMIN（33 号）与 SIGRTMAX（64 号）之间的信号，它们都是可靠信号，也称为实时信号。我们注意到 Linux 下没有 16 和 32 号信号。

在 Linux 系统中，信号的可靠性是指信号是否会丢失，或者说该信号是否支持排队。当导致产生信号的事件发生时，内核就产生一个信号。信号产生后，内核通常会在进程表中设置某种形式的标志。当内核设置了这个标志，我们就说内核向一个进程递送了一个信号。信号产生（generate）和递送（delivery）之间的时间间隔，称为信号未决（pending）。

进程可以调用 sigpending 将信号设置为阻塞，如果为进程产生了一个阻塞的信号，而对该信号的动作是捕捉该信号（即不忽略信号），则内核将为该进程的此信号保持为未决状态，直到该进

我爱自由 5ifreedom.com

程对此信号解除阻塞或将对此信号的响应更改为忽略。如果在进程解除对某个信号的阻塞之前，这种信号发生了多次，那么如果信号被递送多次（即信号在未决信号队列里面排队），则称之为可靠信号；只被递送一次的信号称为不可靠信号。

2. 信号的优先级

信号实质上是软中断，中断有优先级，信号也有优先级。如果一个进程有多个未决信号，则对于同一个未决的实时信号，内核将按照发送的顺序来递送信号。如果存在多个未决的实时信号，则值（或者说编号）越小的越先被递送。如果既存在不可靠信号，又存在可靠信号（实时信号），虽然 POSIX 对这一情况没有明确规定，但 Linux 系统和大多数遵循 POSIX 标准的操作系统一样，将优先递送不可靠信号。

9.1.3 进程对信号的响应

当信号发生时，用户可以要求进程以下列 3 种方式之一对信号做出响应。

- 捕捉信号。对于要捕捉的信号，可以为其指定信号处理函数，信号发生时该函数自动被调用，在该函数内部实现对该信号的处理。
- 忽略信号。大多数信号都可使用这种方式进行处理，但是 SIGKILL 和 SIGSTOP 这两个信号不能被忽略，同时这两个信号也不能被捕获和阻塞。此外，如果忽略某些由硬件异常产生的信号（如非法存储访问或除以 0），则进程的行为是不可预测的。
- 按照系统默认方式处理。大部分信号的默认操作是终止进程，且所有的实时信号的默认动作都是终止进程。

9.2 信号处理

9.2.1 信号的捕捉和处理

Linux 系统中对信号的处理主要由 signal 和 sigaction 函数来完成。另外本小节还将介绍另一个函数 pause，它可用来响应任何信号，不过不做任何处理。

1. signal 函数

signal 函数用来设置进程在接收到信号时的动作，在 Shell 下输入 man signal 可获取该函数原型：

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

signal 会根据参数 signum 指定的信号编号来设置该信号的处理函数。当指定的信号到达时就会跳转到参数 handler 指定的函数执行。如果参数 handler 不是函数指针，则必须是常数 SIG_IGN（忽略该信号）或 SIG_DFL（对该信号执行默认操作）。handler 是一个函数指针，它所指向的函数的类型是 sighandler_t，即它所指向的函数有一个 int 型参数，且返回值的类型为 void。

signal 函数执行成功时返回以前的信号处理函数指针，当有错误发生时则返回 SIG_ERR（即 -1），例 9-1 是对 Signal 函数的用法演示。

注意：SIGKILL 和 SIGSTOP 这两个信号不能被捕捉或忽略。

例 9-1 my_signal.c

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

我爱自由 5ifreedom.com

```
/*信号处理函数*/
void handler_sigint(int signo)
{
    printf("recv SIGINT\n");
}

int main()
{
    /*安装信号处理函数*/
    signal(SIGINT, handler_sigint);

    while(1)
        ;

    return 0;
}
```

程序说明。

程序首先使用 signal()安装信号 SIGINT 的处理函数 handler_sigint，然后进入死循环。当接收到 SIGINT 信号时，程序自动跳转到信号处理函数处执行，打印出提示信息，然后返回主函数继续死循环。执行程序结果如下：

```
[root@localhost chapter 8]# ./my_signal
(按下 Ctrl+c)
recv SIGINT
(按下 Ctrl+c)
recv SIGINT
(按下 Ctrl+\)
Quit
```

执行程序时，用户在终端按下<Ctrl+C>组合键后将向进程发送 SIGINT 信号，最后按下<Ctrl+\>组合键向进程发送 SIGQUIT 信号。由于程序本身没有处理 SIGQUIT 信号，按照默认处理方式，进程退出（也可以使用 kill 命令结束进程）。

2. sigaction 函数

sigaction 函数可以用来检查或设置进程在接收到信号时的动作，在 Shell 下输入 man sigaction 可获取该函数的原型：

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

sigaction 会根据参数 signum 指定的信号编号来设置该信号的处理函数。参数 signum 可以是 SIGKILL 和 SIGSTOP 以外的任何信号。如果参数 act 不是空指针，则为 signum 设置新的信号处理函数；如果 oldact 不是空指针，则旧的信号处理函数将被存储在 oldact 中。struct sigaction 的定义如下：

```
struct sigaction {
            void (*sa_handler)(int);
            void (*sa_sigaction)(int, siginfo_t *, void *);
            sigset_t sa_mask;
            int sa_flags;
            void (*sa_restorer)(void);
}
```

sa_handler 和 sa_sigaction 在某些体系结构上被定义为共用体，即这两个值在某一时刻只有一个有效。

数据成员 sa_restorer 已经作废，不再使用，POSIX 标准也不支持该数据成员。

sa_handler 可以是常数 SIG_DFL 或 SIG_IGN，或者是一个信号处理函数的函数名。信号处理函数只有一个参数即信号编号。该参数和参数 sa_sigaction 实际上都是函数指针，函数指针的内容请参考第 4 章。

sa_sigaction 也是用来指定信号 signum 的处理函数，但是它有 3 个参数，第一个参数是信号编号；第二个参数是一个指向 siginfo_t 结构的指针；第三个参数是一个指向任何类型的指针，一般不使用。

sa_mask 成员声明了一个信号集，在调用信号捕捉函数之前，该信号集会增加到进程的信号屏蔽码中，新的信号屏蔽码会自动包括正在处理的信号（sa_flags 未指定 SA_NODEFER 或

我爱自由 5ifreedom.com

SA_NOMASK）。当从信号捕捉函数返回时，进程的信号屏蔽码会恢复为原来的值。因此，当处理一个给定的信号时，如果这种信号再次发生，那么它会被阻塞直到本次信号处理结束为止。若这种信号发生了多次，则对于不可靠信号，它只会被阻塞一次，即本次信号处理结束以后只会再处理一次（相当于丢失了信号）；对于可靠信号（实时信号），则会被阻塞多次，即信号不会丢失，信号发生了多少次就会调用信号处理函数多少次。

sa_flags 成员用来说明信号处理的一些其他相关操作，它可以取以下值或它们的组合。

SA_NOCLDSTOP：如果参数 signum 为 SIGCHLD，则当子进程暂停时，并不会通知父进程。

SA_ONESHOT 或 SA_RESETHAND：在调用新的信号处理函数前，将此信号的处理方式改为系统默认的方式。

SA_ONSTACK：以预先定义好的堆栈调用信号处理函数。

SA_RESTART：被信号中断的系统调用，在信号处理函数执行完毕后会自动重新开始执行（BSD 操作系统默认使用该值）。

SA_NOMASK 或 SA_NODEFER：在处理此信号结束前允许此信号再次递送，相当于中断嵌套。

SA_SIGINFO：如果设置了该标志，则信号处理函数由三参数的 sa_sigaction 指定而不是 sa_handler 指定。

当使用三参数的 sa_sigaction 来指定信号处理函数时，它的第二个参数可以用来传递数据，其定义如下：

```
siginfo_t {
        int         si_signo;      /* 信号编号 */
        int         si_errno;      /* 错误码 */
        int         si_code;       /* 信号产生的原因 */
        pid_t       si_pid;        /* 接收信号的进程 ID */
        uid_t       si_uid;        /* 接收信号的进程的用户 ID */
        int         si_status;     /* 状态编号 */
        clock_t         si_utime;      /* 耗费的用户空间的时间 */
        clock_t         si_stime;      /* 耗费的系统内核的时间 */
        sigval_t      si_value;      /* 信号值 */
        int         si_int;        /* 用于传递数据 */
        void *      si_ptr;        /* 用户传递数据 */
        void *      si_addr;       /* 产生内存访问错误的内存地址 */
        int         si_band;       /* band 事件 */
        int         si_fd;         /* 文件描述符 */
    }
```

其中，所有的信号都有 si_signo、si_errno 和 si_code 这 3 个数据成员，分别表示信号编号，errno 值和信号产生的原因。其他成员则根据信号的不同含有不同的意义，接收信号的进程只能读这些成员的值，而不能进行设置。si_int 和 si_ptr 可以用来传递数据，后面会演示其用法。其余的数据成员则根据不同的信号存在不同的组合，了解即可。

sigaction 函数执行成功时返回 0，当有错误发生时则返回-1，错误代码存入 errno 中，详细的错误代码说明请参考 man 手册，例 9-2 是一个用函数 Sigaction 实现的例子

注意：Linux 下 signal 函数是由 sigaction 函数实现的。

例 9-2 my_sigaction.c

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

int temp = 0;

/*信号处理函数*/
void handler_sigint(int signo)
{
```

我爱自由 5ifreedom.com

```
    printf("recv SIGINT\n");
    sleep(5);
    temp += 1;
    printf("the value of temp is: %d\n", temp);
    printf("in handler_sigint, after sleep\n");
}

int main()
{
    struct sigaction act;

    /*赋值act结构*/
    act.sa_handler = handler_sigint;
    act.sa_flags = SA_NOMASK;
    /*安装信号处理函数*/
    sigaction(SIGINT, &act, NULL);

    while(1)
        ;

    return 0;
}
```

程序说明。

程序定义了一个act结构，并设置act的sa_handler为信号处理函数handler_sigint，且将sa_flags赋值为SA_NOMASK，即支持信号的嵌套处理。

快速按下<Ctrl+c>组合键两次，执行程序结果如下。

```
[root@localhost chapter8]# ./my_sigaction
(按下Ctrl+c两次)
recv SIGINT
recv SIGINT
the value of temp is: 1
in handler_sigint, after sleep
the value of temp is: 2
in handler_sigint, after sleep
(按下Ctrl+\)
Quit
```

执行程序时，当第一次按下<Ctrl+c>组合键发送出SIGINT信号时，程序打印出“recv SIGINT”表明信号处理程序处理了信号SIGINT，然后sleep（5），睡眠5秒。在5秒之内再次按下<Ctrl+c>组合键时，由于我们设定了sa_flags的值为SA_NOMASK，因此程序又响应一次信号SIGINT，程序从sleep()处嵌套调用信号处理函数handler_sigint，再一次打印出“recv SIGINT”。睡眠5秒之后，将temp的值打印出来并返回到本次信号处理程序的跳入点sleep()处，然后再打印出temp的值并返回到主函数。

从程序执行可以看到，temp的值随着信号处理函数被调用的次数的增加而递增，而由于实际应用中信号总是随机发生的，这样temp的值也会随机变化。如果main函数或其他地方还用到了这个全局变量，则程序将产生不可预料的结果。我们称这种数据会被破坏的函数为不可重入函数。编写信号处理程序时要注意不要使用不可重入函数。一般来说，满足下列条件之一的函数是不可重入的。

（1）使用了静态的数据结构，如getgrgid()，全局变量等。

（2）函数实现时，调用了malloc()或者free()函数。

（3）函数实现时，使用了标准I/O函数。

POSIX.1说明了保证可重入的函数。表9-2列出了这些可重入函数。表9-2中有4个带*号的函数并没有按POSIX.1说明是可重入的，但SVR4 SVID〔AT&T 1989〕包括Linux则将它们列为是可重入的。

我爱自由 5ifreedom.com

表 9-2　　可重入函数

函　数	函　数	函　数	函　数
_exit	fork	Pipe	stat
abort *	fstat	Read	sysconf
access	getegid	Rename	tcdrain
alarm	geteuid	Rmdir	tcflow
cfgetispeed	getgid	Setgid	tcflush
cfgetospeed	getgroups	Setpgid	tcgetattr
cfsetispeed	getpgrp	Setsid	tcgetpgrp
cfsetospeed	getpid	Setuid	csendbreak
chdir	getppid	Sigacton	tcsetattr
chmod	getuid	Sigaddset	tcsetpgrp
chown	kill	Sigdelset	time
close	link	Sigemptyset	times
creat	longjmp *	Sigfillset	umask
dup	lseek	Sigismember	uname
dup2	mkdir	signal *	unlink
execle	mkfifo	sigpending	utime
execve	open	sigprocmask	wait
exit *	pathconf	sigsuspend	waitpid
fcntl	pause	sleep	write

将程序中的“act.sa_flags = SA_NOMASK;”这一行注释去掉，然后重新编译程序并执行（执行时快速按下<Ctrl+c>组合键 3 次或 3 次以上），结果如下。

```
[root@localhost chapter8]# ./my_sigaction
(按下<Ctrl+c>组合键 3 次)
recv SIGINT
the value of temp is: 1
in handler_sigint, after sleep
recv SIGINT
the value of temp is: 2
in handler_sigint, after sleep
(按下<Ctrl+\>组合键)
Quit
```

此时，sigaction 按照默认方式阻塞当前正在处理的信号，但为什么结果只有两个呢？这是因为 SIGINT 是不可靠信号，不可靠信号不支持排队，从而有可能丢失信号。在程序中可以看到，第 3 个信号确实被丢失了。

3. pause

pause 函数使调用进程挂起直至捕捉到一个信号。在 Shell 下输入 man pause 可获取该函数的原型：

```
#include <unistd.h>
   int pause(void);
```

pause 函数会令目前的进程暂停（进入睡眠状态），直到被信号（signal）所中断。该函数只返回-1 并将 errno 设置为 EINTR。

9.2.2　信号处理函数的返回

信号处理函数可以正常返回，也可以调用其他函数返回到程序的主函数中，而不是从该处理程序返回。正如 ANSI C 标准所说明的，一个信号处理程序可以返回或者调用 abort、exit 或 longjmo（goto 不支持跳出它所在的函数，因此不能用来从信号处理程序返回到主函数中）。

1. setjmp/longjmp

使用 longjmp 可以跳转到 setjmp 设置的位置，它们的原型如下：

我爱自由 5ifreedom.com

```
#include <setjmp.h>
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

参数 env 是一个特殊类型 jmp_buf 的变量。这一数据类型是某种形式的数组，其中存放的是在调用 longjmp 时能用来恢复栈状态的所有信息。一般来说 env 是个全局变量，因为需从另一个函数中引用它。我们可以在希望返回的位置使用 setjmp，直接调用 setjmp 时返回 0；当从 longjmp 返回时，setjmp 的返回值是 longjmp 的第 2 个参数的值，可以利用这一点使多个 longjmp 返回到一个 setjmp 处，例 9-3 演示了 setjmp 和 longjmp 函数的用法。

例 9-3 wrong_return.c

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

jmp_buf    env;     // 保存待跳转位置的栈信息

/*信号 SIGRTMIN+15 的处理函数*/
void handler_sigrtmin15(int signo)
{
    printf("recv SIGRTMIN+15\n");
    longjmp(env, 1);    // 返回到 env 处，注意第二个参数的值
}

/*信号 SIGRTMAX-15 的处理函数*/
void handler_sigrtmax15(int signo)
{
    printf("recv SIGRTMAX-15\n");
    longjmp(env, 2);    // 返回到 env 处，注意第二个参数的值
}

int main()
{
    /*设置返回点*/
    switch (setjmp(env)) {
        case 0:
            break;
        case 1:
            printf("return from SIGRTMIN+15\n");
            break;
        case 2:
            printf("return from SIGRTMAX-15\n");
            break;
        default:
            break;
    }

    /*捕捉信号，安装信号处理函数*/
    signal(SIGRTMIN+15, handler_sigrtmin15);
    signal(SIGRTMAX-15, handler_sigrtmax15);

    while (1)
        ;

    return 0;
}
```

程序说明。

程序捕捉了两个实时信号（可靠信号），SIGRTMIN+15 和 SIGRTMAX-15，并安装了它们的信号处理函数。在 main 函数内首先调用 setjmp 设置了返回点，并根据返回值打印出不同的提示信息。信号处理函数内部打印出提示信息以后没有正常返回，而是调用 longjmp 直接跨函数跳转，返回到 setjmp 处。

执行程序时，在一个终端执行本程序，在另一个终端先使用命令 ps -a 查看进程的 *PID*，再使用 kill 命令发送信号，结果如下：

我爱自由 5ifreedom.com

在一个终端执行：

```
[root@localhost chapter8]# ./wrong_return
```

另一终端执行命令：

```
[root@localhost chapter8]# kill -s SIGRTMIN+15 27815
[root@localhost chapter8]# kill -s SIGRTMAX-15 27815
```

第一个终端结果如下：

```
recv SIGRTMIN+15
return from SIGRTMIN+15
recv SIGRTMAX-15
return from SIGRTMAX-15
```

程序执行结果完全符合预期，但是不是就没有问题呢？在第二个终端继续使用 kill 发送信号试试看（可以多发几次）：

```
[root@localhost chapter8]# kill -s SIGRTMIN+15 27815
[root@localhost chapter8]# kill -s SIGRTMAX-15 27815
```

从运行结果可以发现，第一个终端的结果没有任何变化，然而信号又确实发送给目标进程了，这是为什么呢？正如我们在 8.2.1 小节所介绍的，信号处理时会自动阻塞正在被处理的信号，在信号处理函数返回时把进程的信号屏蔽字恢复，即解除对当前信号的阻塞。例 9-3 没有让信号处理函数正常返回，而是使用 longjmp 直接跳转，所以进程的信号屏蔽字在第一次收到信号后，就把信号设置为阻塞并且再也没有恢复，因而再也触发不了信号处理函数了，除非手动将进程对信号的屏蔽去除。如果既想使用跨函数跳转直接返回，又想避免每次都手动清除信号屏蔽的麻烦，就要使用下面将要讲解的 sigsetjmp/siglongjmp 函数了。

2. sigsetjmp/siglongjmp

从例 9-3 可以看出这样一个问题，由于在信号处理期间自动屏蔽了正在被处理的信号，而使用 setjmp/longjmp 跳出信号处理程序时又不会自动将信号屏蔽码修改回原来的屏蔽码，从而引起该信号被永久屏蔽。可以使用 sigsetjmp/siglongjmp 来解决这一问题，它们的函数原型如下：

```
#include <setjmp.h>
int sigsetjmp(sigjmp_buf env, int savesigs);
void siglongjmp(sigjmp_buf env, int val);
```

这两个函数与 setjmp/longjmp 的惟一区别是 sigsetjmp 多了一个参数 savesigs，如果 savesigs 非 0，则 sigsetjmp 在 env 中保存进程的当前信号屏蔽字，在调用 siglongjmp 时会从其中恢复保存的信号屏蔽字，例 9-4 演示函数 sigsetjmp 和函数 siglongjmp 的用法。

例 9-4　right_return.c

```
#include <stdio.h>
#include <setjmp.h>
#include <signal.h>
#include <unistd.h>

#define    ENV_UNSAVE    0
#define    ENV_SAVED       1

int        flag_saveenv = ENV_UNSAVE;
jmp_buf    env;       // 保存待跳转位置的栈信息

/*信号 SIGRTMIN+15 的处理函数*/
void handler_sigrtmin15(int signo)
{
    if (flag_saveenv == ENV_UNSAVE) {
        return;
    }
    printf("recv SIGRTMIN+15\n");
    sleep(10);
    printf("in handler_sigrtmin15, after sleep\n");
    siglongjmp(env, 1);  // 返回到 env 处，注意第二个参数的值
```

我爱自由 5ifreedom.com

```
}

int main()
{
    /*设置返回点*/
    switch (sigsetjmp(env, 1)) {    // sigsetjmp 的第二个参数只要非 0 即可
        case 0:
            printf("return 0\n");
            flag_saveenv = ENV_SAVED;
            break;
        case 1:
            printf("return from SIGRTMIN+15\n");
            break;
        default:
            printf("return else\n");
            break;
    }

    /*捕捉信号，安装信号处理函数*/
    signal(SIGRTMIN+15, handler_sigrtmin15);

    while (1)
        ;

    return 0;
}
```

程序运行。

本程序结构上和例 9-3 没有太大区别，为简化起见，本程序只处理 SIGRTMIN+15 信号，按照例 9-3 的执行方法，结果如下：

在一个终端执行

```
[root@localhost chapter8]# ./right_return
return 0
```

另一终端用 kill 命令连续发送信号 4 次，执行结果如下：

```
[root@localhost chapter8]# kill -s SIGRTMIN+15 28052
[root@localhost chapter8]# kill -s SIGRTMIN+15 28052
[root@localhost chapter8]# kill -s SIGRTMIN+15 28052
[root@localhost chapter8]# kill -s SIGRTMIN+15 28052
```

第一个终端的结果如下：

```
[root@localhost chapter8]# ./right_return
return 0
recv SIGRTMIN+15
in handler_sigrtmin15, after sleep
recv SIGRTMIN+15
in handler_sigrtmin15, after sleep
recv SIGRTMIN+15
in handler_sigrtmin15, after sleep
recv SIGRTMIN+15
in handler_sigrtmin15, after sleep
return from SIGRTMIN+15
```

程序说明。

（1）本程序的信号处理函数先检查 flag_saveenv 的值是否为 ENV_UNSAVE，如果是，则直接返回，因为此时程序还没来得及保存返回点的栈状态信息。在 sigsetjm 之后才将 flag_saveenv 设置为 ENV_SAVED。如果不这样处理，那么当信号发生在调用 sigsetjmp 之前时，信号处理函数将返回到未知地点或程序崩溃（感兴趣的读者可以在 sigsetjmp 前面加上 sleep（20），可以观察到程序崩溃）。使用 siglongjmp 从信号处理程序返回时都应该这样处理。

（2）分析程序执行的结果，可以发现，4 次信号都被响应了，并没有像例 9-2 那样丢失了信号，可见实时信号是可靠的，支持排队的。

（3）执行结果只打印出一条“return from SIGRTMIN+15”。分析整个程序的流程，如图 9-1 所示。

我爱自由 5ifreedom.com

图 9-1　例 9-4 程序流程图

9.2.3　信号的发送

信号的发送主要由函数 kill、raise、sigqueue、alarm、setitimer 以及 abort 来完成。

1．kill 函数

kill 函数用来发送信号给指定的进程，在 Shell 下输入 man 2 kill 可获取其函数原型如下：

```
#include <sys/types.h>
#include <signal.h>
int kill (pid_t pid, int sig);
```

该函数的行为与第一个参数 pid 的取值有关，第二个参数 sig 表示信号编号。

- 如果 pid 是正数，则发送信号 sig 给进程号为 pid 的进程；
- 如果 pid 为 0，则发送信号 sig 给当前进程所属进程组里的所有进程；
- 如果 pid 为-1，则把信号 sig 广播至系统内除 1 号进程（init 进程）和自身以外的所有进程；
- 如果 pid 是比-1 还小的负数，则发送信号 sig 给属于进程组-pid 的所有进程；
- 如果参数 sig 是 0，则 kill()仍执行正常的错误检查，但不发送信号。可以利用这一点来确定某进程是否有权向另外一个进程发送信号。如果向一个并不存在的进程发送空信号，则 kill()返回-1，errno 则被设置为 ESRCH。

函数执行成功返回 0，当有错误发生时则返回-1，错误代码存入 errno 中，详细的错误代码说明请参考 man 手册。

注意：只有具有 root 权限的进程才能向其他任一进程发送信号，非 root 权限的进程只能向属于同一个组或同一个用户的进程发送信号。

例 9-5 实现了自己的 kill 命令，但不支持-l 选项（显示信号编号）。

例 9-5　my_kill.c

我爱自由 5ifreedom.com

```
// 为简化实现，本程序只支持按信号的编号而不是信号名发送信号
// 感兴趣的读者可以按照自己的系统下的 signal.h 增加名字到编号的映射表
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <sys/types.h>

int main(int argc, char ** argv)
{
    int       i, j;
    int       signum = SIGTERM;    // 默认发送 SIGTERM
    pid_t    pid;

    /*首先检查参数个数*/;
    if (argc != 2 && argc != 4) {
        printf("Usage: ./my_kill <-s signum> [PID]\n");
        exit(0);
    }

    /*从命令行参数解析出信号编号*/
    for (i=1; i<argc; i++) {
        if (!strcmp(argv[i], "-s")) {
            signum = atoi(argv[i+1]);
            break;
        }
    }

    /*解析出进程号*/
    if (argc == 2) {
        pid = atoi(argv[1]);
    } else {
        for (j=1; j<argc; j++) {
            if (j != i && j != i+1) {
                pid = atoi(argv[j]);
                break;
            }
        }
    }

    if (kill(pid, signum) < 0) {
        perror("kill");
        exit(1);
    }

    return 0;
}
```

程序说明。

程序首先解析出命令行参数，再调用 kill 函数发送指定的信号给目标进程。执行结果如下（使用例 6-1 配合接收信号）：

是9-1，不是6-1

```
[root@localhost chapter8]# ./my_signal &
[1] 28496
[root@localhost chapter8]# ./my_kill -s 2 28496
[root@localhost chapter8]# recv SIGINT
```

信号 SIGINT 的编号在所有的 Linux 系统中都为 2。从执行结果可以看出，my_kill 成功发出信号并被 my_signal 接收到。

2. raise 函数

raise 函数是 ANSI C 而非 POSIX 标准定义的，用来给调用它的进程发送信号，在 Shell 下输入 man raise 函数可获取其函数原型如下：

```
#include <signal.h>
int raise(int sig);
```

参数 sig 表示要发送的信号编号，成功返回 0，失败返回非 0 值。

我爱自由 5ifreedom.com

3. sigqueue 函数

sigqueue 函数是一个比较新的发送信号的函数，它支持信号带有参数，从而可以与函数 sigaction 配合使用。在 Shell 下输入 man sigqueue 可获取其函数原型如下：

```
#include <signal.h>
int sigqueue(pid_t pid, int sig, const union sigval value);
```

sigqueue 用来发送信号 sig 给进程 pid。与 kill 系统调用不同的是，sigqueue 在发送信号的同时还支持信号携带参数；另一个不同点是 sigqueue 不能给一组进程发送信号。参数 value 是一个共用体，其定义如下：

```
union sigval{
      int   sival_int;
    void  *sival_ptr;
};
```

也就是说，信号携带的参数要么是一个整型值，要么是一个 void 型指针。当接收进程的信号处理函数是由 sigaction 函数设置的并且设置了 SA_SIGINFO 标志（表明使用 3 参数的 sa_sigaction 设置信号处理函数）时，接收进程可以从 siginfo_t 结构的 si_value 域取得信号发送时携带的数据。

函数成功执行时返回 0，表明信号被成功发送到目标进程，当有错误发生时则返回-1，错误代码存入 errno 中，详细的错误代码说明请参考 man 手册。

4. alarm 函数

alarm 函数可以用来设置定时器，定时器超时将产生 SIGALRM 信号给调用进程。在 Shell 下输入 man alarm 可获取其函数原型如下：

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

参数 seconds 表示设定的秒数，经过 seconds 后，内核将给调用该函数的进程发送 SIGALRM 信号。如果 seconds 为 0，则不再发送 SIGALRM 信号。最新一次调用 alarm 函数将取消之前一次的设定，例 9-6 用函数 alarm 实现了网络命令 ping 的功能。

注意：alarm 只设定为发送一次信号，如果要多次发送，就要对 alarm 进行多次调用。

如果之前已经调用过 alarm，则返回之前设置的定时器剩余时间；否则如果之前没有设置过定时器，则返回 0。

例 9-6　simulate_ping.c

```
//模拟 ping 程序的框架
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void send_ip()
{
    /*发送回请求报文，这里只是打印消息*/
    printf("send a icmp echo request packet\n");
}

void recv_ip()
{
    /*挂起在套接字上等待数据并解析报文，这里只是使用死循环*/
    while(1)
        ;
}

void handler_sigalarm(int signo)
{
    send_ip();
    alarm(2);
}

int main()
```

我爱自由 5ifreedom.com

```
{
    /*安装信号处理程序*/
    signal(SIGALRM, handler_sigalarm);

    /*触发一个 SIGALRM 信号给本进程*/
    raise(SIGALRM);
    recv_ip();

    return 0;
}
```

程序说明。

本程序简单地模拟实现使用广泛的网络命令 ping。利用 alarm 定时触发 SIGALRM 信号来实现 ping 程序的定时发包功能。程序执行结果如下：

```
[root@localhost chapter8]# ./simulate_ping
send a icmp echo request packet
send a icmp echo request packet
send a icmp echo request packet
…
```

5. getitimer / setitimer 函数

与 alarm 函数一样，setitimer 函数也是用来设置定时器的，且 alarm 和 setitimer 使用同一个定时器，因此会相互影响。setitimer 要比 alarm 具有更多的功能，在 Shell 下输入 man setitimer 可获取其函数原型如下：

```
#include <sys/time.h>
int     getitimer(int which, struct itimerval *value);
int    setitimer(int which, const struct itimerval *value, struct itimerval *ovalue);
```

第一个参数 which 用来指定使用哪一个定时器，根据参数 which 可单独设定每个定时器，定义时器的种类如下。

- ITIMER_REAL：按实际时间计时，计时到达时将给进程发送 SIGALRM 信号，相当于高精度的 alarm 函数。
- ITIMER_VIRTUAL：仅当进程执行时才进行计时。计时到达时将给进程发送 SIGVTALRM 信号。
- ITIMER_PROF：进程执行的时间以及内核因本进程而消耗的时间都计时。与 ITIMER_VIRTUAL 搭配使用，通常用来统计进程在用户态与核心态花费的时间总和。计时到达时发送 SIGPROF 信号。

参数 value 用来指定定时器的时间，结构 struct itimerval 的定义如下：

```
struct itimerval {
     struct timeval it_interval;     /* next value */
     struct timeval it_value;       /* current value */
};
```

其中结构 struct timeval 的定义如下：

```
struct timeval {
     long tv_sec;                /* 秒数 */
     long tv_usec;               /* 微秒*/
};
```

对于函数 getitimer，如果存在由 which 指定的定时器，则将剩余时间保存在 it_value 中，该定时器的初始值保存在 it_interval 中；如果不存在指定类型的定时器，则将 value 置为 0 返回。执行成功返回 0，当有错误发生时则返回-1，错误代码存入 errno 中，详细的错误代码说明请参考 man 手册。

对于函数 setitimer，参数 ovalue 如果不是空指针，则将在其中保存上次设定的定时器的值。定时器从 value 递减为 0 时，产生一个信号，并将 it_value 的值设为 it_interval，然后重新开始计时，如此周而复始。仅当 it_value 的值为 0 或者计时到达而 it_interval 的值为 0 时，停止计时。执行成功返回 0，当有错误发生时则返回-1，错误代码存入 errno 中，详细的错误代码说明请参考

我爱自由 5ifreedom.com

man 手册，例 9-7 演示了定时器的用法。

例 9-7　test_setitimer.c

```
#include <stdio.h>
#include <signal.h>
#include <sys/time.h>
#include <unistd.h>

/*信号处理程序*/
void handler_sigtime(int signo)
{
    switch (signo) {
        case SIGALRM:
            printf("recv SIGALRM\n");
            break;
        case SIGPROF:
            printf("recv SIGPROF\n");
            break;
        default:
            break;
    }
}

int main()
{
    struct itimerval value;

    /*安装信号处理函数*/
    signal(SIGALRM, handler_sigtime);
    signal(SIGPROF, handler_sigtime);

    /*初始化 value 结构*/
    value.it_value.tv_sec = 1;          // 第一次 1 秒触发信号
    value.it_value.tv_usec = 0;
    value.it_interval.tv_sec = 5;          // 第二次开始每 5 秒触发信号
    value.it_interval.tv_usec = 0;

    /*设置定时器*/
    setitimer(ITIMER_REAL, &value, NULL);
    setitimer(ITIMER_PROF, &value, NULL);

    while(1)
        ;

    return 0;
}
```

程序说明。

程序设置了两个定时器 ITIMER_REAL 和 ITIMER_PROF。系统时间经过 1 秒后，将触发一个 SIGALRM 信号，以后每 5 秒触发一个 SIGALRM 信号。按照程序执行时消耗的时间以及内核因本程序消耗的时间来计时，第一次经过 1 秒后将触发一个 SIGALRM 信号，以后每 5 秒触发一个 SIGPROF 信号。执行程序结果如下：

```
[root@localhost chapter8]# ./set_timer
recv SIGALRM
recv SIGPROF
recv SIGALRM
recv SIGALRM
recv SIGPROF
```

从执行结果可以发现，开始 SIGALRM 先于 SIGPROF 出现，总体上 SIGALRM 的次数要多于 SIGPROF 的次数，这符合预期，因为整个系统并不是只运行这一个进程。

6. abort 函数

abort 函数用来向进程发送 SIGABRT 信号，在 Shell 下输入 man abort 可获取该函数原型：

```
#include <stdlib.h>
void abort(void);
```

如果进程设置信号处理函数以捕捉 SIGABRT 信号，且信号处理函数不返回（如使用 longjmp），

我爱自由 5ifreedom.com

则 abort()不能终止进程。Abort()终止进程时，所有打开的流（如 I/O 流、文件流）均会被刷新和关闭。如果进程设置了 SIGABRT 被阻塞或忽略，abort()将覆盖这种设置。

abort 函数没有返回值。

9.2.4 信号的屏蔽

1. 信号集

9.1.2 小节列出的信号总数目达 64 个，超过了一个整型数能表示的位数（一个整型变量通常为 32 位），因此不能用整型量中的一位代表一种信号。POSIX 标准定义了数据类型 sigset_t 来表示信号集，并且定义了一系列函数来操作信号集。在 Shell 下输入 man sigsetops 可查看它们的函数原型如下：

```
#include <signal.h>
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
int sigismember(const sigset_t *set, int signum);
```

这些函数的具体含义为。

- 函数 sigemptyset 用来初始化一个信号集，使其不包括任何信号。
- 函数 sigfillset 用来初始化一个信号集，使其包括所有信号。
- 函数 sigaddset 用来向 set 指定的信号集中添加由 signum 指定的信号。
- 函数 sigdelset 用来从 set 指定的信号集中删除由 signum 指定的信号。
- 函数 sigismember 用来测试信号 signum 是否包括在 set 指定的信号集中。

函数 sigemptyset、sigfillse、sigaddset 以及 sigdelset 在执行成功时返回 0，失败返回-1。函数 sigismember 返回 1 表示测试的信号在信号集中，返回 0 表示测试的信号不在信号集中，出错返回-1。

注意：所有应用程序在使用信号集前，要对该信号集调用一次 sigemptyset 或 sigfillset 以初始化信号集。这是因为 C 语言编译器将不赋初值的外部和静态度量都初始化为 0。

2. 信号屏蔽

信号屏蔽又称为信号阻塞，在 Shell 下输入 man sigprocmask 可获取信号阻塞的一系列函数的说明：

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set, sigset_t * oldset);
int sigpending(sigset_t *set);
int sigsuspend(const sigset_t *mask);
```

这些函数的具体解释如下。

（1）sigprocmask 函数

每个进程都有一个信号屏蔽码，它规定了当前阻塞而不能递送给该进程的信号集。调用函数 sigprocmask 可以检测或更改进程的信号屏蔽码。如果参数 oldset 是非空指针，则该进程之前的信号屏蔽码通过 oldset 返回；如果参数 set 是非空指针，则该函数将根据参数 how 来修改信号当前屏蔽码，how 的取值如下。

- SIG_BLOCK：将进程新的信号屏蔽码设置为当前信号屏蔽码和 set 指向信号集的并集。
- SIG_UNBLOC：将进程新的信号屏蔽码设置为当前信号屏蔽码中，删除 set 所指向信号集，即 set 包含了我们希望解除阻塞的信号。即使对当前信号屏蔽码中不存在的信号使用 SIG_U NBLOCK 也是合法操作。
- SIG_SETMASK：将进程新的信号屏蔽码设置为 set 指向的值。

函数执行成功返回 0，当有错误发生时则返回-1，错误代码存入 errno 中，详细的错误代码说明请参考 man 手册。

我爱自由 5ifreedom.com

（2）sigpending 函数

函数 sigpending 用来获取调用进程因被阻塞而不能递送和当前未决的信号集。该信号集通过参数 set 返回。

函数执行成功返回 0，当有错误发生时则返回-1，错误代码存入 errno 中，详细的错误代码说明请参考 man 手册，例 9-8 演示了函数 sigprocmask 和函数 sigpending 的用法。

例 9-8 sig_mask.c

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

/*自定义的错误处理函数*/
void my_err(const char * err_string, int line)
{
    fprintf(stderr, "line:%d  ", line);
    perror(err_string);
    exit(1);
}

/*SIGINT 的处理函数*/
void hander_sigint(int signo)
{
    printf("recv SIGINT\n");
}

int main()
{
    sigset_t    newmask, oldmask, pendmask;    // 定义信号集

    /*安装信号处理函数*/
    if (signal(SIGINT, hander_sigint) == SIG_ERR) {
        my_err("signal", __LINE__);
    }

    /*睡眠 10 秒*/
    sleep(10);

    /*初始化信号集 newmask 并将 SIGINT 添加进去*/
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGINT);

    /*屏蔽信号 SIGINT*/
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0) {
        my_err("sigprocmask", __LINE__);
    } else {
        printf("SIGINT blocked\n");
    }

    sleep(10);

    /*获取未决信号队列*/
    if (sigpending(&pendmask) < 0) {
        my_err("sigpending", __LINE__);
    }
    /*检查未决信号队列里面是否有 SIGINT*/
    switch (sigismember(&pendmask, SIGINT)) {
        case 0:
            printf("SIGINT is not in pending queue\n");
            break;
        case 1:
            printf("SIGINT is in pending queue\n");
            break;
        case -1:
            my_err("sigismember", __LINE__);
            break;
        default:
            break;
```

我爱自由 5ifreedom.com

```
    }

    /*解除对 SIGINT 的屏蔽*/
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0) {
        my_err("sigprocmask", __LINE__);
    } else {
        printf("SIGINT unblocked\n");
    }

    while(1)
        ;

    return 0;
}
```

程序说明。

程序首先安装 SIGINT 的信号处理函数，然后睡眠 10 秒（此时按下<Ctrl+c>组合键）。之后将 SIGINT 阻塞，然后再睡眠 10 秒（此时可以多次按下 Ctrl+c 键），并检测 SIGINT 是否在未决信号队列中，最后解除 SIGINT 信号的的阻塞。执行结果如下：

```
[root@localhost chapter8]# ./sig_mask
(按下 Ctrl+c 键)
recv SIGINT
SIGINT blocked
(按下 Ctrl+c 键多次)
SIGINT is in pending queue
recv SIGINT
SIGINT unblocked
(按下 Ctrl+\键)
Quit
```

从结果可以看出，第二个“recv SIGINT”先于“SIGINT unblocked”打印出来。这是因为第二次调用 sigprocmask 解除对 SIGINT 信号的阻塞以后，进程未决信号队列非空，首先执行信号处理函数，然后才执行后面的“printf("SIGINT unblocked\n");”。

程序中设置 SIGINT 为阻塞时，先保存了进程的信号屏蔽字在 oldmask 中，以方便后面解除对 SIGINT 的阻塞。在解除 SIGINT 的阻塞时，重新设置进程的信号屏蔽字（SIG_ SETMASK）为 oldmask。或者也可以使用 SIG_UNBLOCK 使信号不被阻塞，但是这样可能会产生一个问题，当一个大的程序中其他地方可能也阻塞了此信号时，使用 SIG_UNBLOCK 就会把其它地方的设置修改掉，因此，建议使用 SIG_SETMASK 恢复进程的信号屏蔽字而不是使用 SIG_ UNBLOCK 解除特定信号的阻塞。

程序的结果也再次证明了不可靠信号不支持排队，有可能丢失信号。因为第二次按下 Ctrl+c 多次，在解除对 SIGINT 信号的阻塞后，只打印了一个“recv SIGINT”。

（3）sigsuspend 函数

函数 sigsuspend 将进程的信号屏蔽码设置为 mask，然后与 pause 函数一样等待信号的发生并执行完信号处理函数。信号处理函数执行完后再把进程的信号屏蔽码设置为原来的屏蔽字，然后 sigsuspend 函数才返回。sigsuspend 函数保证改变进程的屏蔽码和将进程挂起等待信号是原子操作。

sigsuspend 函数总是返回-1，并将 errno 置为 EINTR，例 9-9 演示了 sigsuspend 函数的用法。

例 9-9 sig_suspend.c

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

/*自定义的错误处理函数*/
void my_err(const char * err_string, int line)
{
    fprintf(stderr, "line:%d  ", line);
    perror(err_string);
    exit(1);
}
```

我爱自由 5ifreedom.com

```
/*SIGINT 的处理函数*/
void handler_sigint(int signo)
{
    printf("recv SIGINT\n");
}

int main()
{
    sigset_t    newmask, oldmask, zeromask;    // 定义信号集

    /*安装信号处理函数*/
    if (signal(SIGINT, handler_sigint) == SIG_ERR) {
        my_err("signal", __LINE__);
    }

    sigemptyset(&newmask);
    sigemptyset(&zeromask);
    sigaddset(&newmask, SIGINT);

    /*屏蔽信号 SIGINT*/
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0) {
        my_err("sigprocmask", __LINE__);
    } else {
        printf("SIGINT blocked\n");
    }

    /*临界区*/

    /*使用 sigsuspend 取消所有信号的屏蔽并等待信号的触发*/
    if (sigsuspend(&zeromask) != -1) {  // sigsuspend 总是返回-1
        my_err("sigsuspend", __LINE__);
    } else {
        printf("recv a signo, return from sigsuspend\n");
    }

/*-----------------------------------------
// 如果使用 sigprocmask 加上 pause 可能会出现错误
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0) {
        my_err("sigprocmask", __LINE__);
    }
    pause();
-----------------------------------------*/

    /*将信号屏蔽字恢复*/
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0) {
        my_err("sigprocmask", __LINE__);
    }

    while (1)
        ;

    return 0;
}
```

程序说明。

程序使用 sigprocmask()屏蔽掉信号 SIGINT，然后使用 sigsuspend()取消对所有信号的屏蔽，并挂起等待信号的触发，执行程序结果如下：

```
[root@localhost chapter8]# ./sig_suspend &
[1] 29209
SIGINT blocked
[root@localhost chapter8]# ps -a
    PID  TTY      TIME  CMD
29209 pts/0    00:00:00  sig_suspend
29210 pts/0    00:00:00  ps
[root@localhost chapter8]# kill -s SIGRTMIN 29209
[root@localhost chapter8]#
[1]+  Real-time signal 0     ./sig_suspend
```

由于所有实时信号的默认响应都是终止进程，因此程序在收到信号 SIGRTMIN 之后，解除

我爱自由 5ifreedom.com

sigsuspend()的挂起状态，并将信号屏蔽字修改为第一次调用 sigprocmask()前的状态。

如果为了达到相同的效果，不使用 sigsuspend()而是使用注释掉的那部分代码，虽然表面上看没有问题，并且执行程序也基本上不会发现问题，但潜在的 bug 将一直存在，特别是在复杂的实际应用中就有可能暴露出来。如果信号发生在 sigprocmask()之后 pause()之前，则这个信号就会丢失掉了，且如果信号只发生一次，程序将永远挂起在 pause()上。

9.3 编程中如何获得帮助

编程中任何人都会遇到问题，有时忘记了函数的参数或函数的使用方法，这时可以使用 man 命令获得帮助或查阅相关书籍；在编译程序时，编译器报错，这时当然是根据编译器的提示到出错位置的附近查找错误；编译器只能查出语法错误，而不能查出程序的逻辑错误和思路的错误。有的程序编译可以通过但却显示不出预期的结果，这就是逻辑错误。这时 gdb 调试器就可以派上用场了，通过跟踪程序的执行，在程序执行过程中查看各种值一般可以较快地查出错误所在。当然要灵活使用调试器 gdb，还是要多实践。编译器 gcc 和调试器 gdb 的使用方法请参考前面的章节。

如果经过了前面的步骤还是不能解决问题，那就应该向别人或网络求助了。对于初学者容易遇到的问题，网络上一般都载有解决方法。这些解决方法可以通过输入正确的关键词在搜索引擎（比如百度或 Google）上搜索得到。

学习编程总是不断地经历着这样的循环："阅读书籍和资料→上机编程实践→遇到问题→解决"。如果只是经历其中的某个或某几个步骤（比如只阅读而不上机实践），那么实际编程能力的提高是很有限的。

9.4 编程实践：应用实例

9.4.1 实例一：信号的发送与处理

信号不仅可以用来处理异步事件，也可以用来传递数据，例 9-10 介绍了如何利用信号来实现数据的传递。

例 9-10 send_data_signo.c

```
// 示例利用信号传递数据，本程序发送数据
// 选项-d 后跟待传递的数据，选项-s 后跟待发送的信号，选项-p 后跟目的进程 ID
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char ** argv)
{
    union sigval    value;
    int             signum = SIGTERM;    // 默认发送 SIGTERM
    pid_t           pid;
    int             i;

    /*初始化*/
    value.sival_int = 0;

    /*检查参数的个数*/
    if (argc != 3 && argc != 5 && argc != 7) {
```

我爱自由 5ifreedom.com

```
        printf("./send_data_signo <-d data> <-s signum> [-p][data]\n");
        exit(1);
    }

    /*从命令行参数解析出信号编号、PID 以及待传递的数据*/
    for (i=1; i<argc; i++) {
        if (!strcmp(argv[i], "-d")) {
            value.sival_int = atoi(argv[i+1]);
            continue;
        }
        if (!strcmp(argv[i], "-s")) {
            signum = atoi(argv[i+1]);
            continue;
        }
        if (!strcmp(argv[i], "-p")) {
            pid = atoi(argv[i+1]);
            continue;
        }
    }

    /*利用 sigqueue 给 pid 发送信号 signum，并携带数据 value*/
    if (sigqueue(pid, signum, value) < 0) {
        perror("sigqueue");
        exit(1);
    }

    return 0;
}
```

程序说明。

程序首先从命令行参数中解析出需要发送的数据，然后利用函数 sigqueue 对目标进程发送信号并同时发送数据。

例 9-11　recv_data_signo.c

```
// 示例利用信号传递数据，本程序接收数据
#include <signal.h>
#include <stdio.h>

/*3 参数的信号处理程序*/
void handler_sigint(int signo, siginfo_t *siginfo, void * pvoid)
{
    printf("recv SIGINT, the data value is:%d\n", siginfo->si_int);
}

int main()
{
    struct sigaction act;

    /*赋值 act 结构*/
    act.sa_sigaction = handler_sigint;
    act.sa_flags = SA_SIGINFO;        // 指定使用 3 参数的信号处理函数
    /*安装信号处理函数*/
    sigaction(SIGINT, &act, NULL);

    while(1)
        ;

    return 0;
}
```

程序说明。

程序使用了 3 参数的信号处理函数，并在函数内获取了信号的参数并打印出来。两个程序执行结果如下：

```
[root@localhost chapter8]# ./recv_data_signo &
[1] 29517
[root@localhost chapter8]# ./send_data_signo -s 2 -d 100 -p 29517
```

我爱自由 5ifreedom.com

```
[root@localhost chapter8]# recv SIGINT, the data value is:100

[root@localhost chapter8]# ./send_data_signo -d 100 -p 29517
[root@localhost chapter8]#
[1]+  Terminated              ./recv_data_signo
```

从执行结果可以看出，整型数据顺利地通过信号在进程间进行传递。至于利用信号传递指针，作为习题留给读者。

9.4.2 实例二：信号应用于事件通知

实际应用中，进程常常需要等待某一事件的发生，一般可以通过检测某一全局变量来判断事件是否发生。有 3 种方法可以实现这一要求。

第一种方法：程序不停地循环检测全局变量，这样可以满足要求，但是非常占用 CPU 资源。如例 9-12 所示。

第二种方法：进程使用 pause()挂起，等待信号的触发，事件发生时向进程发送信号，对应的信号处理函数改变全局变量的值，信号处理函数返回后进程检测该全局变量，满足要求即可知道事件已发生。如例 9-13 所示。

第三种方法：原理与第二种方法一致，不过使用的是函数 sigsuspend。如例 9-14 所示。

下面的程序都假定需要等待的事件为用户按下<Ctrl+c>组合键，其他事件也类似，可以发送预先约定的消息。

例 9-12 wait_cycle.c

```
// 示例采用循环检测来判断事件发生
#include <stdio.h>
#include <signal.h>

#define   UNHAPPEN   0   // 未发生
#define   HAPPENED   1   // 已发生

/*定义全局变量以标识事件是否发生*/
int       flag_happen;

void handler_sigint(int signo)
{
    flag_happen = HAPPENED;
}

int main()
{
    /*安装信号处理函数*/
    if (signal(SIGINT, handler_sigint) == SIG_ERR) {
        perror("signal");
        exit(1);
    }

    while (1) {
        if (flag_happen == HAPPENED) {
            printf("event happened\n");
            /*... you can do something else here...*/
            break;
        }
    }

    return 0;
}
```

执行程序结果如下：

```
[root@localhost chapter8]# ./wait_cycle
event happened
```

我爱自由 5ifreedom.com

读者可以查看 CPU 使用率的变化（如根据/proc/stat 文件的信息或其他工具）。

例 9-13　wait_pause.c

```
// 示例 pause 挂起等待事件的发生
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

#define    UNHAPPEN    0    // 未发生
#define    HAPPENED    1    // 已发生

/*定义全局变量以标识事件是否发生*/
int        flag_happen;

void handler_sigint(int signo)
{
    flag_happen = HAPPENED;
}

int main()
{
    /*安装信号处理函数*/
    if (signal(SIGINT, handler_sigint) == SIG_ERR) {
        perror("signal");
        exit(1);
    }

    while (flag_happen == UNHAPPEN)
        pause();

    printf("after event happened\n");
    /*... you can do something else here ...*/

    return 0;
}
```

执行程序结果如下：

```
[root@localhost chapter8]# ./wait_pause
after event happened
```

程序说明。

本程序不会像上一个程序那样使 CPU 占用率过高，又似乎可以进行事件通知。实际上，本程序存在一个潜在的 bug：当信号发生在 while()之后、pause()之前时，程序将检测不到事件的发生。此时执行信号处理函数，flag_happen 被修改为 HAPPENED，之后调用 pause()，而这时信号都已经被处理完了，自然 pause()将一直挂起，那么进程不会检测到此次事件的发生。

例 9-14　wait_sigsuspend.c

```
// 示例使用 sigsuspend 函数挂起等待事件的发生
#include <stdio.h>
#include <signal.h>

#define    UNHAPPEN    0    // 未发生
#define    HAPPENED    1    // 已发生

/*定义全局变量以标识事件是否发生*/
int        flag_happen;

/*自定义的错误处理函数*/
void my_err(const char * err_string, int line)
{
    fprintf(stderr, "line:%d  ", line);
    perror(err_string);
    exit(1);
}
```

我爱自由 5ifreedom.com

```
void handler_sigint(int signo)
{
    flag_happen = HAPPENED;
}

int main()
{
    sigset_t    newmask, oldmask, zeromask;

    /*安装信号处理函数*/
    if (signal(SIGINT, handler_sigint) == SIG_ERR) {
        my_err("signal", __LINE__);
    }

    sigemptyset(&newmask);
    sigemptyset(&zeromask);
    sigaddset(&newmask, SIGINT);

    /*屏蔽信号 SIGINT*/
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0) {
        my_err("sigprocmask", __LINE__);
    } else {
        printf("SIGINT blocked\n");
    }

    while (flag_happen == UNHAPPEN)
        sigsuspend(&zeromask);

    printf("after event happened\n");
    /*... do something else...*/

    /*将信号屏蔽字恢复*/
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0) {
        my_err("sigprocmask", __LINE__);
    }

    return 0;
}
```

程序说明。

程序首先将 SIGINT 屏蔽，然后使用 sigsuspend()挂起，等待事件的发生，感兴趣的读者可以仔细分析整个程序，看是不是任何时候发生的事件都能被程序检测到。执行结果如下：

```
[root@localhost chapter8]# ./wait_sigsuspend
SIGINT blocked
after event happened
```

9.5 习题

1. 信号有哪几个来源？信号有几种？进程有哪几种响应信号的方法？
2. 编写程序，捕获和处理 SIGALRM 信号。
3. 如果删除例 9-1 的 while（1）死循环，结果如何？
4. 利用 alarm()和 pause()实现 sleep 函数，并分析程序看有没有什么潜在的 bug。
5. 利用 alarm()、sigprocmask()和 sigsuspend()可靠地实现 sleep 函数。
6. 实现 raise 函数。
7. Sigsetjmp()和 siglongjmp()可以用在哪些场合？
8. 编写利用信号传递指针的程序（发送与接收功能在同一个程序里）。*有没有可能在进程间传递指针？为什么？请编程验证。*

我爱自由 5ifreedom.com

第 10 章　进程间通信

本章介绍 Linux 下进程间的通信方式，主要方法有：管道、有名管道、消息队列、信号量和共享内存，除此之外还有信号和套接字。最后两种由于涉及的内容很多，故单独成章。本章以大量的程序实例来说明各种进程间通信方法的使用。最后还介绍了 Linux 软件开发中极为常用的库开发技术。

本章重点：

- 几种进程间通信方法的概念和使用。
- Linux 下如何创建和使用库。

本章难点：

- 消息队列的创建和读写。
- 信号量的概念。
- 共享内存的互斥访问。

10.1 进程间通信概述

前面提到进程的地址空间是各自独立的，因此进程之间交互数据必须采用专门的通信机制。特别是在大型的应用系统中，往往需要多个进程相互协作共同完成一个任务，这就需要使用进程间通信（Internet Process Connection，IPC）编程技术。

Linux 下进程间通信的方法基本上是从 UNIX 平台继承而来的。Linux 操作系统不但继承了 system V IPC 通信机制，还继承了基于套接字（socket）的进程间通信机制。前者是贝尔实验室对 UNIX 早期的进程间通信手段的改进和扩充，其通信的进程局限于单台计算机内；后者则突破了这一局限，通信的进程可以运行在不同主机上，也就是进行网络通信。

下面对 Linux 下进程间通信的几种主要手段作一个简单的介绍。

- 管道（pipe）：管道是一种半双工的通信方式，数据只能单方向流动，而且只能在具有亲缘关系的进程间使用。进程的亲缘关系通常是指父子进程关系。
- 有名管道（named pipe）：有名管道也是半双工的通信方式，但是它允许无亲缘关系进程间的通信。
- 信号量（semophore）：信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此主要作为进程间以及同一进程内不同线程之间的同步手段。
- 消息队列（message queue）：消息队列是由消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息量少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。
- 信号（signal）：信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生。信号已在第 9 章介绍过了，本章不再赘述。
- 共享内存（shared memory）：共享内存就是映射一段能被其他进程所访问的内存，这段

我爱自由 5ifreedom.com

共享内存由一个进程创建但是多个进程都可以访问。共享内存是最快的 IPC 方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号量，配合使用，来实现进程间的同步和通信。

- 套接字（socket）：套接口也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同机器间的进程通信。第 11 章将详细介绍基于套接字的网络编程。

10.2 管道

10.2.1 管道的概念

管道是一种两个进程间进行单向通信的机制。因为管道传递数据的单向性，管道又称之半双工管道。管道的这一特点决定了其使用的局限性。

- 数据只能由一个进程流向另一个进程（其中一个写管道，另一个读管道）；如果要进行全双工通信，需要建立两个管道。
- 管道只能用于父子进程或者兄弟进程间的通信，也就是说管道只能用于具有亲缘关系的进程间的通信，无亲缘关系的进程不能使用管道。

除了以上局限性，管道还有其他一些不足，如管道没有名字，管道的缓冲区大小是受限制的，管道所传送的是无格式的字节流。这就要求管道的输入方和输出方事先约定好数据的格式。虽然有这么多不足，但对于一些简单的进程间的通信，管道还是完全可以胜任的。

使用管道进行通信时，两端的进程向管道读写数据是通过创建管道时，系统设置的文件描述符进行的。因此对于管道两端的进程来说，管道就是一个特殊的文件，这个文件只存在于内存中。在创建管道时，系统为管道分配一个页面作为数据缓冲区，进行管道通信的两个进程通过读写这个缓冲区来进行通信。

通过管道通信的两个进程，一个进程向管道写数据，另外一个进程从管道的另一端读数据。写入的数据每次都添加在管道缓冲区的末尾，读数据的时候都是从缓冲区的头部读出数据。

10.2.2 管道的创建与读写

1. 管道的创建

Linux 下创建管道可以通过函数 pipe 来完成。该函数如果调用成功返回 0，并且数组中将包含两个新的文件描述符；如果有错误发生，返回-1。该函数原型如下：

```
#include <unistd.h>
int pipe(int fd[2])
```

管道两端可分别用描述符 fd[0]以及 fd[1]来描述。需要注意的是，管道两端的任务是固定的，一端只能用于读，由描述符 fd[0]表示，称其为管道读端；另一端只能用于写，由描述符 fd[1]来表示，称其为管道写端。如果试图从管道写端读数据，或者向管道读端写数据都将导致出错。

管道是一种文件，因此对文件操作的 I/O 函数都可以用于管道，如 read()，write()等。

注意：管道一旦创建成功，就可以作为一般的文件来使用，对一般文件进行操作的 I/O 函数也适用于管道，如。

管道的一般用法是，进程在使用 fork 函数创建子进程前先创建一个管道，该管道用于在父子进程间通信，然后创建子进程，之后父进程关闭管道的读端，子进制关闭管道的写端。父进程负

我爱自由 5ifreedom.com

责向管道写数据而子进程负责读数据。当然父进程可以关闭管道的写端而子进程关闭管道的读端。这样管道就可以用于父子进程间的通信，也可用于兄弟进程间的通信。下面介绍进程是如何通过管道读写数据的。

2. 从管道中读数据

如果某进程要读取管道中的数据，那么该进程应当关闭 fd1，同时向管道写数据的进程应当关闭 fd0。因为管道只能用于具有亲缘关系的进程间的通信，在各进程进行通信时，它们共享文件描述符。在使用前，应及时地关闭不需要的管道的另一端，以避免意外错误的发生。

进程在管道的读端读数据时，如果管道的写端不存在，则读进程认为已经读到了数据的末尾，读函数返回读出的字节数为 0；管道的写端如果存在，且请求读取的字节数大于 PIPE_BUF，则返回管道中现有的所有数据；如果请求的字节数不大于 PIPE_BUF，则返回管道中现有的所有数据（此时，管道中数据量小于请求的数据量），或者返回请求的字节数（此时，管道中数据量大于等于请求的数据量）。

注意：PIPE_BUF 在 include/linux/limits.h 中定义，不同的内核版本可能会有所不同。

3. 向管道中写数据

如果某进程希望向管道中写入数据，那么该进程应该关闭 fd0 文件描述符，同时管道另一端的进程关闭 fd1。向管道中写入数据时，Linux 不保证写入的原子性（原子性是指操作在任何时候都不能被任何原因所打断，操作要么不做要么就一定完成）。管道缓冲区一有空闲区域，写进程就会试图向管道写入数据。如果读进程不读走管道缓冲区中的数据，那么写操作将一直被阻塞等待。

在写管道时，如果要求写的字节数小于等于 PIPE_BUF，则多个进程对同一管道的写操作不会交错进行。但是，如果有多个进程同时写一个管道，而且某些进程要求写的字节数超过 PIPE_BUF 所能容纳时，则多个写操作的数据可能会交错。

注意：只有在管道的读端存在时，向管道中写入数据才有意义。否则，向管道中写入数据的进程将收到内核传来的 SIGPIPE 信号。应用程序可以处理也可以忽略该信号，如果忽略该信号或者捕捉该信号并从其处理程序返回，则 write 出错，错误码为 EPIPE。

下面通过一个例 10-1，说明管道的创建，以及对管道的读写是如何进行的。

例 10-1　pipe.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>

/*读管道*/
void read_from_pipe (int fd)
{
        char message[100];
        read (fd,message,100);
        printf("read from pipe:%s",message);
}

/*写管道*/
void write_to_pipe (int fd)
{
      char *message = "Hello, pipe!\n";
      write (fd, message,strlen(message)+1);
}

int main(void)
{
```

我爱自由 5ifreedom.com

```
    int     fd[2];
    pid_t   pid;
    int     stat_val;

    if (pipe (fd))
    {
            printf ("create pipe failed!\n");
            exit (1);
    }

    pid = fork();
    switch (pid)
    {
       case -1:
            printf ("fork error!\n");
            exit (1);
       case 0:
            /*子进程关闭 fd1*/
            close (fd[1]);
            read_from_pipe (fd[0]);
            exit (0);
       default:
            /*父进程关闭 fd0*/
            close (fd[0]);
            write_to_pipe (fd[1]);
            wait (&stat_val);
            exit (0);
    }

    return 0;
}
```

程序说明。

该程序中，父进程向管道中写数据，子进程从管道中获取数据。可以看到，对管道的读写和对一般文件的读写没有什么区别。

程序运行结果如下：

```
[zzy@mci ~]$ ./pipe
Hello, pipe!
```

注意： 必须在系统调用 fork()之前调用 pipe()，否则子进程将不会继承管道的文件描述符。

管道是半双工的（一端只写不能读，另一端只读不能写），但是可以通过创建两个管道来实现一个全双工（两端都可以读和写）通信。下面通过例 10-2 来说明如何实现全双工通信。

例 10-2　dual_pipe.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

/*子进程读写管道的函数*/
void child_rw_pipe (int readfd, int writefd)
{
    char *message1 = "from child process!\n";
    write(writefd, message1,strlen(message1)+1);

    char message2[100];
    read (readfd,message2,100);
    printf("child process read from pipe:%s",message2);
}

/*父进程读写管道的函数*/
void parent_rw_pipe(int readfd,int writefd)
{
    char *message1 = "from parent process!\n";
    write(writefd, message1,strlen(message1)+1);
```

我爱自由 5ifreedom.com

```
    char message2[100];
    read (readfd,message2,100);
    printf("parent process read from pipe:%s",message2);
}

int main(void)
{
    int  pipe1[2],pipe2[2];
    pid_t     pid;
    int  stat_val;

    printf("realize full-duplex communication:\n\n");
    if(pipe(pipe1))
    {
                printf("pipe1 failed!\n");
                exit(1);
    }
    if(pipe(pipe2))
    {
                printf("pipe2 failed!\n");
                exit(1);
    }

    pid = fork();
    switch(pid)
    {
            case -1:
                    printf("fork error!\n");
                    exit(1);
            case 0:
                    /*子进程关闭 pipe1 的读端，关闭 pipe2 的写端*/
                    close(pipe1[1]);
                    close(pipe2[0]);
                    child_rw_pipe(pipe1[0],pipe2[1]);
                    exit(0);
        default:
                    /*父进程关闭 pipe2 的写端，关闭 pipe1 的读端*/
                    close(pipe1[0]);
                    close(pipe2[1]);
                    parent_rw_pipe(pipe2[0],pipe1[1]);
                    wait(&stat_val);
                    exit(0);
    }
}
```

程序说明。

该程序父子进程之间相互发送信息。程序的运行结果如下：

```
[zzy@mci ~]$ ./dule_pipe
realize full-duplex communication:

parent process read from pipe:from child process!
child process read from pipe:from parent process!
```

4. dup()和 dup2()

前面的例子中，子进程可以直接共享父进程的文件描述符。但是如果子进程调用 exec 函数执行另外一个应用程序时，就不能再共享了。这种情况下可以将子进程中的文件描述符重定向到标准输入，当新执行的程序从标准输入获取数据时实际上是从父进程中获取输入数据。dup 和 dup2 函数提供了复制文件描述符的功能。两个函数的声明均在头文件 unistd.h 中：

```
# include <unistd.h>
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

dup 和 dup2 函数调用成功时均返回一个 oldfd 文件描述符的副本，失败则返回-1。所不同的是，

我爱自由 5ifreedom.com

由 dup 函数返回的文件描述符是当前可用文件描述符中的最小数值，而 dup2 函数则可以利用参数 newfd 指定欲返回的文件描述符。如果参数 newfd 指定的文件描述符已经打开，系统先将其关闭，然后将 oldfd 指定的文件描述符赋值到该参数。如果 newfd 等于 oldfd，则 dup2 返回 newfd，而不关闭它。

下面是 dup 和 dup2 使用的对比：

```
/*dup 程序片断*/          STDIN_FILENO  0        /*dup2 程序片断*/
pid = fork();           STDOUT_FILENO 1                pid = fork();
if(pid == 0)            STDERR_FILENO 2                     if(pid == 0)
{                                                    {
  /*关闭子进程的标准输出*/                              /*关闭标准输出并复制管道输出端
   close(1);                                            到标准输出*/
   //复制管道输入端到标准输出*/                        dup(1,fd[1])
   dup(fd[1]);                                       execve("exam",argv,environ)
   execve("exam",argv,environ);                      …
     …                                               }
}
```

可见 dup2 系统调用将 close 操作和文件描述符拷贝操作集成在同一个函数里，而且它保证操作具有原子性。

10.2.3 管道的应用实例

管道的一种常见用法是：在父进程创建子进程后向子进程传递参数。例如，一个应用软件有一个主进程和很多个不同的子进程。主进程创建子进程后，在子进程调用 exec 函数执行一个新程序前，通过管道给即将执行的程序传递命令行参数，子进程根据传来的参数进行初始化或其他操作。下面通过例 10-3 和例 10-4 来演示这种用法，这个程序包括一个主进程程序和一个子进程中要执行的新程序。

例 10-3 monitor.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>

int main(int arg, char *argv[],char ** environ)
{
    int       fd[2];
    pid_t     pid;
    int       stat_val;

    if (arg < 2)
    {
        printf("wrong parameters \n");
        exit(0);
    }
    if (pipe(fd))
    {
        perror("pipe failed");
        exit(1);
    }

    pid = fork();
    switch (pid)
    {
        case -1:
            perror ("fork failed!\n");
            exit(1);
        case 0:
            close (0);
            dup (fd[0]);
            execve ("ctrlprocess",(void *)argv,environ);
```

我爱自由 5ifreedom.com

```
                exit (0);
            default:
                close(fd[0]);
                /*将命令行第一个参数写进管道*/
                write (fd[1], argv[1], strlen(argv[1]));
                break;
        }
        wait (&stat_val);
        exit (0);
}
```

例 10-4　ctrlpocess.c

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

int main(int arg, char * argv[])
{
    int     n;
    char    buf[1024];

    while (1)
    {
        if ((n = read (stdin, buf, 1024)) > 0)
        {
            buf[n] = '\0';
            printf ("ctrlprocess receive: %s\n",buf);

            if (!strcmp (buf,"exit"))
                exit(0);

            if (!strcmp (buf, "getpid"))
            {
                printf ("My pid:%d\n", getpid());
                sleep (3);
                exit (0);
            }
        }
    }
}
```

程序说明。

主进程向管道中写一个命令行参数，子进程从标准输入里面读出该参数，进行相应的操作。首先编译 ctrlprocess.c:

```
[zzy@mci ~]$ gcc -o ctrlprocess ctrlprocess.c
```

再编译主程序 monitor.c:

```
[zzy@mci ~]$ gcc -o monitor monitor.c
```

分别运行./ mainprocess exit 和./ mainprocess getpid，结果分别如下：

```
[zzy@mci ~]$ ./monitor exit
ctrlprocess receive exit
[zzy @mci ~]$ ./monitor getpid
ctrlprocess receive getpid
My pid:17031
```

可见，被监控子进程接受监控主进程的命令，执行不同的操作。这在实际项目中是经常见到的，监控进程启动加载多个具有不同功能的子进程，并通过管道的形式向子进程传送参数。

10.3 有名管道

10.3.1 有名管道的概念

管道的一个不足之处是没有名字，因此，只能用于具有亲缘关系的进程间通信，在有名管道(named

我爱自由 5ifreedom.com

pipe 或 FIFO）提出后，该限制得到了克服。FIFO 不同于管道之处在于它提供一个路径名与之关联，以 FIFO 的文件形式存储于文件系统中。有名管道是一个设备文件，因此，即使进程与创建 FIFO 的进程不存在亲缘关系，只要可以访问该路径，就能够通过 FIFO 相互通信。值得注意的是，FIFO（First In First Out）总是按照先进先出的原则工作，第一个被写入的数据将首先从管道中读出。

10.3.2 有名管道的创建与读写

Linux 下有两种方式创建有名管道。一是在 Shell 下交互地建立一个有名管道，二是在程序中使用系统函数建立有名管道。Shell 方式下可使用 mknod 或 mkfifo 命令，下面命令使用 mknod 创建了一个有名管道：

```
mknod namedpipe
```

创建有名管道的系统函数有两个：mknod 和 mkfifo。两个函数均定义在头文件 sys/stat.h，函数原型如下：

```
#include <sys/types.h>
#iinclude <sys/stat.h>
int mknod(const char * path , mode_t mod, dev_t dev);
int mkfifo(const char *path, mode_t mode);
```

函数 mknod 参数中 path 为创建的有名管道的全路径名；mod 为创建的有名管道的模式，指明其存取权限；dev 为设备值，该值取决于文件创建的种类，它只在创建设备文件时才会用到。这两个函数调用成功都返回 0，失败都返回-1。下面使用 mknod 函数创建了一个有名管道：

```
umask(0);
if (mknod ("/tmp/fifo", S_IFIFO | 0666, 0) == -1)
{
    perror ("mknod error!");
    exit (1);
}
```

函数 mkfifo 前两个参数的含义和 mknod 相同。下面是使用 mkfifo 的示例代码：

```
umask(0);
if (mkfifo ("/tmp/fifo", S_IFIFO | 0666) == -1)
{
    perror ("mkfifo error!");
    exit(1);
}
```

“S_IFIFO | 0666”指明创建一个有名管道且存取权限为 0666，即创建者、与创建者同组的用户、其他用户对该有名管道的访问权限都是可读可写。关于存取权限的更多内容请参考第 6 章。

有名管道创建后就可以使用了，有名管道和管道的使用方法基本是相同的。只是使用有名管道时，必须先调用 open()将其打开。因为有名管道是一个存在于硬盘上的文件，而管道是存在于内存中的特殊文件。

需要注意的是，调用 open()打开有名管道的进程可能会被阻塞。但如果同时用读写方式（O_RDWR）打开，则一定不会导致阻塞；如果以只读方式（O_RDONLY）打开，则调用 open()函数的进程将会被阻塞直到有写方打开管道；同样以写方式（O_WRONLY）打开也会阻塞直到有读方打开管道。

下面通过例 10-5 和例 10-6 演示使用有名管道在无亲缘关系的进程间如何进行通信。这个实例包含两个程序，程序名分别为 procread.c、procwrite.c。

例 10-5 procread.c

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

我爱自由 5ifreedom.com

```
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/types.h>

#define FIFO_NAME           "myfifo"
#define BUF_SIZE            1024

int main(void)
{
        int     fd;
        char    buf[BUF_SIZE];

        umask (0);
        fd = open(FIFO_NAME, O_RDONLY);
        read (fd, buf, BUF_SIZE);
        printf ("Read content: %s\n", buf);
        close (fd);
        exit (0);
}
```

例 10-6 procwrite.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

#define FIFO_NAME  "myfifo"
#define BUF_SIZE        1024
int main(void)
{
     int     fd;
     char    buf[BUF_SIZE] = "Hello procwrite, I come from process named procread!";

     umask(0);

     if (mkfifo (FIFO_NAME, S_IFIFO | 0666) == -1)
     {
          perror ("mkfifo error!");
          exit (1);
     }

     if((fd = open (FIFO_NAME, O_WRONLY) ) == -1)/*以写方式打开 FIFO*/
     {
          perror ("fopen error!");
          exit (1);
     }
     write (fd, buf, strlen(buf)+1); /*向 FIFO 写数据*/

     close (fd);
     exit (0);
}
```

程序说明。

编译后首先运行 procwrite（运行后处于阻塞状态），打开另外一个终端运行程序 procread。运行结果如下：

```
[zzy@mci ~]$ ./procwrite
[zzy@mci ~]$ ./procread
Read content: Hello procwrite, I come from process named procread!
```

10.3.3 有名管道的应用实例

通过创建两个管道可以实现进程间的全双工通信，同样也可以通过创建两个 FIFO 来实现不同进程间的全双工通信。下面通过例 10-7 和例 10-8 演示一个程序中两个进程间的聊天程序（一

我爱自由 5ifreedom.com

个为（server 端，另一个为 clienl 端）来说明使用有名管道进行全双工通信。

例 10-7 server.c

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>

#define FIFO_READ           "readfifo"
#define FIFO_WRITE          "writefifo"
#define BUF_SIZE            1024

int main(void)
{
    int         wfd, rfd;
    char        buf[BUF_SIZE];
    int         len;

     umask(0);
    if (mkfifo (FIFO_WRITE, S_IFIFO|0666))
    {
        printf ("Can't create FIFO %s because  %s", FIFO_WRITE, strerror(errno));
        exit (1);
    }
    umask(0);
    wfd = open(FIFO_WRITE, O_WRONLY);
    if (wfd == -1)
    {
        printf("open FIFO %s error: %s", FIFO_WRITE, strerror(errno));
        exit(1);
    }

    while ((rfd = open(FIFO_READ, O_RDONLY)) == -1)
    {
        sleep(1);
    }

    while (1)
    {
        printf ("Server: ");
        fgets (buf, BUF_SIZE, stdin);
        buf[strlen(buf)-1] = '\0';
        if (strncmp (buf,"quit", 4) == 0)
        {
            close (wfd);
            unlink (FIFO_WRITE);
            close (rfd);
            exit (0);
        }
        write (wfd, buf, strlen(buf));

        len = read (rfd, buf, BUF_SIZE);
        if ( len > 0)
        {
            buf[len] = '\0';
            printf ("Client: %s\n", buf);
        }
    }
}
```

例 10-8 client.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>
```

我爱自由 5ifreedom.com

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

# define FIFO_READ      "writefifo"
# define FIFO_WRITE     "readfifo"
#define BUF_SIZE        1024

int main(void)
{
     int wfd, rfd;
     char buf[BUF_SIZE];
     int len;

     umask(0);
     if(mkfifo (FIFO_WRITE, S_IFIFO|0666))
     {
        printf ("Can't create FIFO %s because %s", FIFO_WRITE, strerror(errno));
        exit(1);
     }

     while ((rfd = open(FIFO_READ, O_RDONLY)) == -1)
     {
          sleep(1);
     }

     wfd = open(FIFO_WRITE, O_WRONLY);
     if (wfd == -1)
     {
        printf("Fail to open FIFO %s: %s", FIFO_WRITE, strerror(errno));
        exit(-1);
     }

     while (1)
     {
          len = read(rfd, buf, BUF_SIZE);
          if ( len > 0)
          {
               buf[len] = '\0';
               printf("Server: %s\n", buf);
          }

        printf("Client: ");
        fgets(buf, BUF_SIZE, stdin);
        buf[strlen(buf) -1] = '\0';
        if (strncmp(buf,"quit", 4) == 0)
        {
           close (wfd);
           unlink (FIFO_WRITE);
           close (rfd);
           exit (0);
        }
       write (wfd, buf, strlen(buf));
     }
}
```

程序说明。

观察 server 和 client 两个程序，可以看出两者的实现基本是一样的，只不过对 FIFO 文件的读写顺序颠倒了一下，两个程序中只要将定义 FIFO 文件名的宏的值对换一下就可以了。分别在两个终端上运行这两个程序，并在 server 端和 client 端输入数据观察它们的运行结果：

server 端输入输出如下：

```
[zzy@mci ~]$ ./server
Server: hello        /* hello 为用户输入 */
Client: world        /* world 为程序输出 */
Server:
```

我爱自由 5ifreedom.com

client 端数据输出输入如下：

```
[zzy@mci ~]$ ./client
Server: hello        /* hello 为程序输出 */
Client: world        /* world 为用户输入 */
```

从运行结果可以看出，通过两个有名管道也可以实现进程间的双向通信。

10.4 消息对列

10.4.1 消息对列的基本概念

消息队列是一个存放在内核中的消息链表，每个消息队列由消息队列标识符标识。与管道不同的是消息队列存放在内核中，只有在内核重启（即操作系统重启）或者显式地删除一个消息队列时，该消息队列才会被真正删除。

操作消息队列时，需要用到一些数据结构，熟悉这些数据结构是掌握消息队列的关键。下面介绍几个重要的数据结构。

1. 消息缓冲结构

向消息队列发送消息时，必须组成合理的数据结构。Linux 系统定义了一个模板数据结构 msgbuf：

```
#include <linux/msg.h> inux-3.10.46/include/uapi/linux/msg.h
struct  msgbuf{  /* message buffer for msgsnd and msgrcv calls */
     long mtype;
     char mtext[1];
};
```

结构体中的 mtype 字段代表消息类型。给消息指定类型，可以使得消息在一个队列中重复使用。mtext 字段指消息内容。

注意：mtext 虽然定义为 char 类型，并不代表消息只能是一个字符，消息内容可以为任意类型，由用户根据需要定义。如下面就是用户定义的一个消息结构：

```
struct myMsgbuf{
     long mtype;
     struct student stu;
};
```

消息队列中的消息的大小是受限制的，由<linux/msg.h>中的宏 MSGMAX 给出消息的最大长度，在实际应用中要注意这个限制。#define MSGMAX 8192 /* <= INT_MAX */ /* max size of message (bytes) */

2. msqid_ds 内核数据结构

Linux 内核中，每个消息队列都维护一个结构体 msqid_ds，此结构体保存着消息队列当前的状态信息。该结构定义在头文件 linux/msg.h 中，具体定义如下：

```
sruct msqid_ds{
                struct_ipc_perm          msg_perm;
                struct_msg               *msg_first;
                struct_msg               *msg_last;
              __kernel_t time_t          msg_stime;
              __kernel_t time_t          msg_rtime;
              __kernel_t time_t          mag_ctime;
                unsigned  long           msg_lcbytes;
                unsigned  long           msg_lqbytes;
                unsigned  short      msg_cbytes;
                unsigned  short      msg_qnum;
                unsigned  short      msg_qbytes;
              __kernel_ipc_pid_t         msg_lspid;
              __kernel_ipc_pid_t         msg_lrpid;
};
```

我爱自由 5ifreedom.com

各字段的含义如下。

- msg_perm：是一个 ipc_perm（定义在头文件 linux/ipc.h）的结构，保存了消息队列的存取权限，以及队列的用户 ID、组 ID 等信息。
- msg_first：指向队列中的第一条消息
- msg_last：指向队列中的最后一条消息
- msg_stime：向消息队列发送最后一条信息的时间
- msg_rtime：从消息队列取最后一条信息的时间
- mag_ctime：最后一次变更消息队列的时间
- msg_cbytes：消息队列中所有消息占的字节数
- msg_qnum：消息队列中消息的数目
- msg_qbytes：消息队列的最大字节数
- msg_lspid：向消息队列发送最后一条消息的进程 ID
- msg_lrpid：从消息队列读取最后一条信息的进程 ID

3. ipc_perm 内核数据结构

结构体 ipc_perm 保存着消息队列的一些重要的信息，比如消息队列关联的键值，消息队列的用户 ID、组 ID 等，它定义在头文件 linux/ipc.h 中：

```
struct ipc_perm{
        __kernel_key_t key;
        __kernel_uid_t uid;
        __kernel_gid_t gid;
        __kernel_uid_t cuid;
        __kernel_gid_t cgid;
        __kernel_mode_t     mode;
        unsigned_short seq ;
}
```

几个主要字段的含义如下。

- key：创建消息队列用到的健值 key
- uid：消息队列的用户 ID
- gid：消息队列的组 ID
- cuid：创建消息队列的进程用户 ID
- cgid：创建消息队列的进程组 ID

10.4.2 消息队列的创建与读写

1. 创建消息队列

消息队列是随着内核的存在而存在的，每个消息队列在系统范围内对应惟一的键值。要获得一个消息队列的描述符，只需提供该消息队列的键值即可，该键值通常由函数 ftok 返回。该函数定义在头文件 sys/ipc.h 中：

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok (const char * pathname, int proj_id);
```

ftok 函数根据 pathname 和 proj_id 这两个参数生成惟一的键值。该函数执行成功会返回一个键值，失败返回-1。例 10-9 是一个获取键值的例子：

例 10-9 ftok.c

```
#include <stdio.h>
#include <sys/types.h>
```

我爱自由 5ifreedom.com

```
include <sys/ipc.h>

int main( void )
{
     int i;
     for ( i = 1; i <= 5; i++ )
          printf( "key[%d] = %ul \n", i, ftok( ".", i) );
           exit(0);
}
```

运行结果的片段如下：

```
[zzy@mci ~]$ ./ftok
key[1] = 171249251
key[2] = 339021411
key[3] = 506793571
key[4] = 674565731
key[5] = 842337891
```

注意：参数 pathname 在系统中一定要存在且进程有权访问，参数 proj_id 的取值范围为 1～255。

Ftok()返回的键值可以提供给函数 msgget。Msgget()根据这个键值创建一个新的消息队列或者访问一个已存在的消息队列。msgget 定义在头文件 sys/msg.h 中：

```
int msgget (key_t key, int msgflg);
```

msgget 的参数 key 即为 ftok 函数的返回值。msgflg 是一个标志参数。以下是 msgflg 的可能取值。

- IPC_CREATE：如果内核中不存在键值与 key 相等的消息对列，则新建一个消息队列；如果存在这样的消息队列，返回该消息队列的描述符。
- IPC_EXCL：和 IPC_CREATE 一起使用，如果对应键值的消息队列已经存在，则出错，返回-1。

注意：IPC_EXCL 单独使用是没有任何意义的。

该函数如果调用成功返回一个消息队列的描述符，否则返回-1。

2. 写消息队列

创建了一个消息队列后，就可以对消息队列进行读写了。函数 msgsnd 用于向消息队列发送（写）数据。该函数定义在头文件 sys/msg.h 中：

```
int msgsnd(int msqid, struct msgbuf *msgp, size_t msgsz, int magflg);
```

msgsnd 各参数含义如下。

- msqid：函数向 msgid 标识的消息队列发送一个消息。
- msgp：msgp 指向发送的消息。
- msgsz：要发送的消息的大小，不包含消息类型占用的 4 个字节。
- magflg：操作标志位。可以设置为 0 或者 IPC_NOWAIT。如果 magflg 为 0，则当消息队列已满的时候，msgsnd 将会阻塞，直到消息可以写进消息队列；如果 magflg 为 IPC_NOWAIT，当消息队列已满的时候，msgsnd 函数将不等待立即返回。

msgsnd 函数成功返回 0，失败返回-1。常见错误码有：EAGAIN，说明消息队列已满；EIDRM，说明消息队列已被删除；EACCESS，说明无权访问消息队列。例 10-10 演示了如何向消息队列发送消息。

例 10-10 sendmsg.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define   BUF_SIZE          256
#define   PROJ_ID           32
#define   PATH_NAME     "."
```

我爱自由 5ifreedom.com

```
int main(void)
{
    /*用户自定义消息缓冲*/
    struct mymsgbuf {
        long msgtype;
        char ctrlstring[BUF_SIZE];
    } msgbuffer;
    int      qid;    /*消息队列标识符*/
    int      msglen;
    key_t    msgkey;

    /*获取键值*/
    if((msgkey = ftok (PATH_NAME, PROJ_ID)) == -1)
    {
        perror ("ftok error!\n");
        exit (1);
    }

    /*创建消息队列*/
    if((qid = msgget (msgkey, IPC_CREAT|0660)) == -1)
    {
        perror ("msgget error!\n");
        exit (1);
    }

    /*填充消息结构，发送到消息队列*/
    msgbuffer. msgtype = 3;
    strcpy (msgbuffer.ctrlstring , "Hello,message queue");
    msglen = sizeof(struct mymsgbuff) - 4;
    if(msgsnd (qid, &msgbuffer, msglen, 0) == -1)
    {
        perror ("msgget error!\n");
        exit (1);
    }

    exit(0);
}
```

程序说明。

编译并运行这个程序后，就向消息队列放入了一条消息，可以通过命令 ipcs 查看。执行结果如下：

```
[zzy@mci ~]#gcc -o sendmsg senmdmag.c
[zzy@mci ~]#./sendmsg
[zzy@mci ~]# ipcs
--------message Queues --------
key        msqid      owner      perms      used-bytes   messages
0x2002b6dc  0          zzy        660        256          1
```

从输出的结果可以看出，系统内部生成一个消息队列，其中含有一条消息。

3. 读消息队列

消息队列中放入数据后，其他进程就可以读取其中的消息了。读取消息的系统调用为 msgrcv()，该函数定义在头文件 sys/msg.h 中，其原型如下：

```
int msgrcv (int msqid, struct msgbuf *msgp, size_t msgsz, long int msgtyp, int msgflg);
```

该函数有 5 个参数，含义如下。

- msqid：消息队列描述符。
- msgp：读取的消息存储到 msgp 指向的消息结构中。
- msgsz：消息缓冲区的大小。
- msgtyp：为请求读取的消息类型。
- msgflg：操作标志位。msgflg 可以为 IPC_NOWAIT，IPC_EXCEPT,IPC_NOERROR

3 个常量。这些值的意义分别为：IPC_NOWAIT，如果没有满足条件的消息，调用立即返回，此时错误码为 ENOMSG；IPC_EXCEPT，与 msgtyp 配合使用，返回队列中第一个类型不为 msgtyp

我爱自由 5ifreedom.com

的消息；IPC_NOERROR，如果队列中满足条件的消息内容大于所请求的 msgsz 字节，则把该消息截断，截断部分将被丢弃。

调用 msgrcv 函数的时候，成功会返回读出消息的实际字节数，否则返回-1。常见错误码有：E2BIG，表示消息的长度大于 msgsz；EIDRM，表示消息队列已被删除；EINVAL，说明 msqid 无效或 msgsz 小于 0。

下面通过例 10-11 来说明如何从消息队列读消息，该例读取例 10-10 写入消息队列的数据。

例 10-11　rcvmsg.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define   BUF_SIZE          256
#define   PROJ_ID           32
#define   PATH_NAME     "."

int main(void)
{
      /*用户自定义消息缓冲区*/
      struct mymsgbuf{
          long msgtype;
          char ctrlstring[BUF_SIZE];
      } msgbuffer;
      int       qid;     /*消息队列标识符*/
      int       msglen;
      key_t     msgkey;

      /*获取键值*/
      if((msgkey = ftok(PATH_NAME, PROJ_ID)) == -1)
      {
          perror("ftok error!\n");
          exit(1);
      }

      /*获取消息队列标识符*/
      if((qid = msgget(msgkey, IPC_CREAT|0660)) == -1)
      {
          perror ("msgget error!\n");
          exit (1);
      }

      msglen = sizeof(struct mymsgbuf) - 4;
      if (msgrcv(qid, &msgbuffer, msglen, 3, 0) == -1)  /*读取数据*/
      {
          perror ("msgrcv error!\n");
          exit (1);
      }
      printf("Get message %s\n", msgbuffer.ctrlstring);

      exit(0);
}
```

编译运行程序结果如下：

```
[zzy@mci ~]#gcc -o rcvmsg rcvmsg.c
[zzy@mci ~]# ./rcvmsg
Get message Hello, message queue
```

10.4.3　获取和设置消息队列的属性

消息队列的属性保存在系统维护的数据结构 msqid_ds 中，用户可以通过函数 msgctl 获取或设

我爱自由 5ifreedom.com

置消息队列的属性。msgctl 定义在头文件 sys/msg.h 中，如下：

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

msgctl 系统调用对 msqid 标识的消息队列执行 cmd 操作，系统定义了 3 种 cmd 操作：IPC_STAT、IPC_SET、IPC_RMID，它们的意义如下：

- IPC_STAT：该命令用来获取消息队列对应的 msqid_ds 数据结构，并将其保存到 buf 指向的地址空间。
- IPC_SET：该命令用来设置消息队列的属性，要设置的属性存储在 buf 中，可设置的属性包括：msg_perm.uid、msg_perm.gid、msg_perm.mode 以及 msg_qbytes。
- IPC_RMID：从内核中删除 msqid 标识的消息队列。

例 10-12 演示了如何获取和设置消息队列的属性，程序如下。

例 10-12 opmsg.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define BUF_SIZE  256
#define PROJ_ID   32
#define PATH_NAME  "."

void getmsgattr(int msgid, struct msqid_ds  msq_info);

int main(void)
{
     /*用户自定义消息缓冲区*/
     struct mymsgbuf{
          long msgtype;
          char ctrlstring[BUF_SIZE];
     } msgbuffer;
     int qid; /*消息队列标识符*/
     int msglen;
     key_t msgkey;
     struct msqid_ds  msq_attr;

     /*获取键值*/
     if((msgkey = ftok(PATH_NAME, PROJ_ID)) == -1)
     {
          perror("ftok error!\n");
          exit(1);
     }
     /*获取消息队列标识符*/
     if((qid = msgget(msgkey, IPC_CREAT|0660)) == -1)
     {
          perror("msgget error!\n");
          exit(1);
     }
     getmsgattr(qid, msq_attr); /*输出消息队列的属性*/

     /*发送一条消息到消息队列*/
     msgbuffer.msgtype = 2;
     strcpy(msgbuffer.ctrlstring,"Aother message");
     msglen = sizeof(struct mymsgbuf)-4;
     if(msgsnd(qid,&msgbuffer,msglen,0) == -1)
     {
          perror("msgget error!\n");
          exit(1);
     }
     getmsgattr(qid, msq_attr); /*再输出消息队列的属性*/

     /*设置消息队列的属性*/
```

我爱自由 5ifreedom.com

```
        msq_attr.msg_perm.uid = 3;
        msq_attr.msg_perm.gid = 2;
        if(msgctl(qid,IPC_SET, &msq_attr) == -1)
        {
            perror("msg set error!\n");
            exit(1);
        }
        getmsgattr(qid, msq_attr);/*修改后再观察其属性*/
        if(msgctl(qid, IPC_RMID, NULL) == -1)
        {
            perror("delete msg error!\n");
            exit(1);
        }
        getmsgattr(qid, msq_attr);/*删除后再观察其属性*/
        exit(0);
    }

    void getmsgattr(int msgid, struct msqid_ds msg_info)
    {
        if(msgctl(msgid,IPC_STAT,&msg_info) == -1)
        {
            perror("msgctl error!\n");
            return;
        }
        printf("****information of message queue%d****\n"'msgid);
        printf("last msgsnd to msq time is %s\n", ctime(&(msg_info.msg_stime)));
        printf("last msgrcv time from msg is %s\n", ctime(&(msg_info.msg_rtime)));
        printf("last change msq time is %s\n", ctime(&(msg_info.msg_ctime)));
        printf("current number of bytes on queue is %d\n",msg_info.msg_cbytes);
        printf("number of messages in queue is %d\n",msg_info.msg_qnum);
        printf("max number of bytes on queue is %d\n",msg_info.msg_qbytes);
        printf("pid of last msgsnd is %d\n",msg_info.msg_lspid);
        printf("pid of last msgrcv is %d\n",msg_info.msg_lrpid);

        printf("msg uid is %d\n",msg_info.msg_perm.uid);
        printf("msg gid is %d\n",msg_info.msg_perm.gid);
        printf("******information end!***********\n"msgid);
    }
```

该程序的运行结果片段如下：

```
[zzy@mci ~]#./opmsg
****information of message queue ****
last msgsnd to msq time is Thu Mar 29 00:19:04 2007
last msgrcv time from msg is Thu Mar 29 00:32:17 2007
last change msq time is Thu Mar 29 00:19:04 2007
current number of bytes on queue is 0
number of messages in queue is 0
max number of bytes on queue is 16384
pid of last msgsnd is 2151
pid of last msgrcv is 2180
msg uid is 0
msg gid is 0
******information end!***********
```

程序说明。

结果片段显示的是对消息队列进行操作前的属性。发送消息后和重新设置后的消息队列属性都会因为操作而改变。可以运行程序观察全部的输出结果，对比操作前后消息队列属性是如何改变的。

10.4.4 消息队列的应用实例

这里以一个聊天程序为例，进一步展示消息队列的应用。聊天的两端分别为进程 server 和进程 client。程序如例 10-13 和例 10-14：

例 10-13 server.c

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

我爱自由 5ifreedom.com

```
#include<stdlib.h>
#include<string.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>
#include<sys/stat.h>

#define BUF_SIZE            256
#define PROJ_ID             32
#define PATH_NAME           /tmp"
#define SERVER_MSG      1
#define CLIENT_MSG      2

int main(void)
{
     /*用户自定义消息缓冲区*/
     struct mymsgbuf{
          long msgtype;
          char ctrlstring[BUF_SIZE];
     } msgbuffer;

     int       qid; /*消息队列标识符*/
     int       msglen;
     key_t     msgkey;

     /*获取键值*/
     if ((msgkey = ftok (PATH_NAME, PROJ_ID)) == -1)
     {
          perror ("ftok error!\n");
          exit(1);
     }

     if ((qid = msgget (msgkey, IPC_CREAT|0660)) == -1)
     {
          perror ("msgget error!\n");
          exit(1);
     }

     while (1)
     {
          printf ("server: ");
          fgets (msgbuffer.ctrlstring, BUF_SIZE, stdin);
          if (strncmp("exit", msgbuffer.ctrlstring, 4) == 0)
          {
               msgctl(qid,IPC_RMID,NULL);
               break;
          }
          msgbuffer.ctrlstring[strlen(msgbuffer. ctrlstring)-1] = '\0';
          msgbuffer.msgtype = SERVER_MSG;
          if (msgsnd(qid,&msgbuffer,strlen(msgbuffer.ctrlstring) + 1,0) == -1)
          {
               perror ("Server msgsnd error!\n");
               exit(1);
          }


          if (msgrcv (qid,&msgbuffer,BUF_SIZE,CLIENT_MSG,0) == -1)
          {
               perror("Server msgrcv error!\n");
               exit(1);
          }
          printf ("Client: %s\n",msgbuffer.ctrlstring);
     }
     exit(0);
}
```

例 10-14　client.c

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

我爱自由 5ifreedom.com

```
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/stat.h>


#define BUF_SIZE            256
#define PROJ_ID             32
#define PATH_NAME           "/tmp"
#define SERVER_MSG      1
#define CLIENT_MSG      2

int main(void)
{
     /*用户自定义消息缓冲区*/
     struct mymsgbuf {
          long msgtype;
          char ctrlstring[BUF_SIZE];
     } msgbuffer;
     int       qid; /*消息队列标识符*/
     int       msglen;
     key_t     msgkey;
     /*获取键值*/
     if ((msgkey = ftok(PATH_NAME, PROJ_ID)) == -1)
     {
          perror("ftok error!\n");
          exit(1);
     }

     if ((qid = msgget(msgkey, IPC_CREAT|0660)) == -1)
     {
          perror("msgget error!\n");
          exit(1);
     }

     while (1)
     {
         if (msgrcv(qid,&msgbuffer,BUF_SIZE,SERVER_MSG,0) == -1)
          {
               perror ("Server msgrcv error!\n");
               exit (1);
          }

          printf ("server: %s\n", msgbuffer.ctrlstring);
          printf ("client: ");
          fgets ( msgbuffer.ctrlstring, BUF_SIZE, stdin);
          if (strncmp ("exit", msgbuffer.ctrlstring, 4) == 0)
          {
               break;
          }
          msgbuffer.ctrlstring[strlen(msgbuffer. ctrlstring)-1] = '\0';
          msgbuffer.msgtype = CLIENT_MSG;
          if(msgsnd(qid,&msgbuffer,strlen(msgbuffer.ctrlstring) + 1,0) == -1)
          {
               perror("client msgsnd error!\n");
               exit(1);
          }
     }
     exit(0);
}
```

程序说明。

先运行 server 程序，再在另外一个终端运行 client 程序。然后这两个程序之间就可以进行聊天，两个终端的操作如下。

我爱自由 5ifreedom.com

```
[zzy@mci ~]# ./server
server: hello,client!
Client: hello,server!
server
[zzy@mci ~]# ./client
server: hello,client!
client: hello,server!
```

10.5 信号量

10.5.1 信号量的基本概念

信号量是一个计数器，常用于处理进程或线程的同步问题，特别是对临界资源访问的同步。临界资源可以简单地理解为在某一时刻只能由一个进程或线程进行操作的资源，这里的资源可以是一段代码、一个变量或某种硬件资源。信号量的值大于或等于 0 时表示可供并发进程使用的资源实体数；小于 0 时代表正在等待使用临界资源的进程数。

注意： 这里的信号量与第 9 章的信号是没有关系的。

与消息队列类似，Linux 内核也为每个信号集维护了一个 semid_ds 数据结构实例。该结构定义在头文件 linux/sem.h 中，各字段含义如下：

```
struct semid_ds {
        struct ipc_perm    sem_perm;                /*对信号进行操作的许可权*/
        __kernel_time_t    sem_otime;               /*对信号进行操作的最后时间*/
        __kernel_time_t    em_ctime;            /*对信号进行修改的最后时间*/
        struct sem         *sembase;            /*指向第一个信号*/
        struct sem_queue   sem_pending;         /*等待处理的挂起操作*/
        struct sem_queque  **sem_pending_last;  /*最后一个正在挂起的操作*/
        struct sem_undo    *undo;               /*撤销的请求*/
        ushort             sem_nsems;           /*数组中的信号数*/
};
```

10.5.2 信号量的创建与使用

1. 信号集的创建或打开

Linux 下使用系统函数 semget 创建或打开信号集。这个函数定义在头文件 sys/sem.h 中，函数原型如下：

```
int semget(key_t key, int nsems, int semflg);
```

该函数执行成功则返回一个信号集的标识符，失败返回-1。函数的第一个参数是由 ftok()得到的键值；第二个参数 nsems 指明要创建的信号集包含的信号个数，如果只是打开信号集，把 nsems 设置为 0 即可；第三个参数 semflg 为操作标志，可以取如下值。

- IPC_CREATE：调用 semget()时，它会将此值与系统中其他信号集的 key 进行对比，如果存在相同的 key，说明信号集已存在，此时返回该信号集的标识符，否则新建一个信号集并返回其标识符。
- IPC_EXCL：该宏须和 IPC_CREATE 一起使用，否则没有意义。当 semflg 取 IPC_CREATE | IPC_EXCL 时，表示如果发现信号集已经存在，则返回错误，错误码为 EEXIST。

下面是一个创建信号集并对信号集中所有信号进行初始化的函数：

```
int createsem (const char * pathname, int proj_id, int members, int init_val)
{
```

我爱自由 5ifreedom.com

```
        key_t          msgkey;
    int        index, sid;
        union semun    semopts;  // 本节后面将介绍该结构体

        /*获取键值*/
        if ((msgkey = ftok(pathname, proj_id)) == -1)
        {
            perror ("ftok error!\n");
            return -1;
        }

        if ((sid = semget(msgkey,members,IPC_CREATE|0666)) == -1)
        {
            perror ("semget call failed.\n");
       return -1;
        }

        /*初始化操作,本节后面将介绍该函数的使用方法*/
        semopts.val = init_val;
        for(index = 0; index<members; index++)
        {
            semctl (sid, index,SETVAL, semopts);
        }
        return (sid);
    }
```

2. 信号量的操作

信号量的值与相应资源的使用情况有关，当它的值大于 0 时，表示当前可用资源的数量，当它的值小于 0 时，其绝对值表示等待使用该资源的进程个数。信号量的值仅能由 PV 操作来改变。在 Linux 下，PV 操作通过调用函数 semop 实现。该函数定义在头文件 sys/sem.h，原型如下：

```
int semop(int semid, struct sembuf * sops, size_t nsops);
```

函数的参数 semid 为信号集的标识符；参数 sops 指向进行操作的结构体数组首地址；参数 nsops 指出将要进行操作的信号的个数。semop 函数调用成功返回 0，否则返回-1。

semop 的第二个参数 sops 指向的结构体数组中，每个 sembuf 结构体对应一个特定信号的操作。因此对信号进行操作必须熟悉该数据结构，该结构体定义在 linux/sem.h，如下所示：

```
struct sembuf {
        ushort   sem_num;       /*信号在信号集中的索引*/
        short    sem_op;        /*操作类型，见表 9-1*/
        short    sem_flg;       /*操作标志*/
}
```

表 10-1　　sem_op 的取值及意义

取值范围	操作意义
sem_op > 0	信号加上 sem_op 的值，表示进程释放控制的资源
sem_op = 0	如果没有设置 IPC_NOWAIT，则调用进程进入睡眠状态，直到信号值为 0；否则进程不会睡眠，直接返回 EAGAIN
sem_op < 0	信号加上 sem_op 的值。若没有设置 IPC_NOWAIT，则调用进程阻塞，直到资源可用；否则进程直接返回 EAGAIN

下面是对一个信号集中的某个信号进行操作的 P、V 函数。

```
/*P 操作函数*/
int sem_p (int semid, int index)
{
        struct sembuf buf = {0,-1,IPC_NOWAIT};

        if (index < 0)
        {
            perror ("index of array cannot equals a minus value!");
```

我爱自由 5ifreedom.com

```
            return -1;
        }

        buf.sem_num = index;
        if (semop (semid, &buf, 1) == -1)
        {
            perror ("a wrong operation to semaphore occurred!");
            return -1;
        }

        return 0;
}

/*V 操作函数*/
int sem_v (int semid, int index)
{
        struct sembuf buf = {0, 1, IPC_NOWAIT};

        if (index < 0)
        {
            perror ("index of array cannot equals a minus value!");
            return -1;
        }

        buf.sem_num = index;
        if (semop (semid, &buf, 1) == -1)
        {
            perror ("a wrong operation to semaphore occurred!");
            return -1;
        }

        return 0;
}
```

3. 信号集的控制

使用信号量时，往往需要对信号集进行一些控制操作，比如删除信号集、对内核维护的信号集的数据结构 semid_ds 进行设置、获取信号集中信号值等。通过 semctl 控制函数可以完成这些操作，该函数定义在 sys/sem.h，如下所示：

```
int semctl(int semid, int semnum, int cmd,...);
```

函数中，参数 semid 为信号集的标识符；参数 semnum 标识一个特定的信号；cmd 指明控制操作的类型；最后的“...”说明函数的参数是可选的，它依赖于第三个参数 cmd，它通过共用体变量 semun 选择要操作的参数。semun 定义在 include/sem.h，如下所示：

```
union semun{
    int val;    //使用的值
    struct semid_ds *buf;   //IPC_STAT、IPC_SET使用缓存区
    unsigned short *array;  //GETALL、SETALL使用的缓存区
    struct seminfo *__buf;  //IPC_INFO(linux特有)使用缓存区
};
```

```
        int             val;
        struct semid_ds *buf;
        ushort          *array;
        struct seminfo  *buf;
        void            *pad;
};
```

返回值：失败返回-1。成功返回0。
semid:信号量集合的编号。
semnum:信号量在集合中的标号。

各字段的含义如下。

- val：仅用于 SETVAL 操作类型，设置某个信号的值等于 val。
- buf：用于 IPC_STAT 和 IPC_SET 操作，存取 semid_ds 结构。
- array：用于 SETALL 和 GETALL 操作。
- buf：为控制 IPC_INFO 提供的缓存。

第二个参数 cmd,，通过宏来指示操作类型，可取的各个宏的含义如下。

IPC_STAT：通过 semun 结构体的 buf 参数返回当前的 semid_ds 结构体。

注意，调用者必须首先分配一个 semid_ds 结构，并把 buf 设置为指向这个结构体。

- IPC_SET：对信号集的属性进行设置。

我爱自由 5ifreedom.com

- IPC_RMID：把 semid 指定的信号集从系统中删除。
- GETPID：返回最后一个执行 semop 操作的进程 ID。
- GETVAL：返回信号集中 semnum 指定信号的值。
- GETALL：返回信号集中所有信号的值。
- GETNCNT：返回正在等待资源的进程的数量。
- GETZCNT：返回正在等待完全空闲资源的进程数量。
- SETVAL：设置信号集中 semnum 指定的信号的值。
- SETALL：设置信号集中所有信号的值。

下面是一个获取和设置单个信号的函数，代码如下所示：

```
int semval_op(int semid, int index, int cmd)
{
     if (index<0) {
          printf ("index cannot be minus!\n")
          return -1;
}
if (cmd == GETVAL || cmd == SETVAL){
     return semctl (semid, index, cmd,0);
}
     printf ("function cannot surport cmd:%d\n", cmd);
     return -1;
}
```

10.5.3 信号量的应用实例

信号量一般用于处理访问临界资源的同步问题。下面通过例 10-15 和例 10-16 中的 server.c 和 client.c 程序来演示信号量如何控制对资源的访问。server 创建一个信号集，并对信号量循环减 1，相当于分配资源。client 执行时检查信号量，如果其值大于 0 代表有资源可用，继续执行，如果小于等于 0 代表资源已经分配完毕，进程 client 退出。

例 10-15 server.c

```
#include <sys/types.h>
#include <linux/sem.h>

#define MAX_RESOURCE    5

int main(void)
{
     key_t     key;
     int       semid;
     struct sembuf sbuf = {0, -1, IPC_NOWAIT};
     union semun   semopts;

     if ((key = ftok(".", 's')) == -1)
     {
          perror ("ftok error!\n");
          exit (1);
     }

     if ((semid = semget (key,1,IPC_CREAT|0666)) == -1)
     {
          perror ("semget error!\n");
          exit (1);
     }

     semopts.val = MAX_RESOURCE;
     if (semctl (semid,0,SETVAL, semopts) == -1)
     {
```

我爱自由 5ifreedom.com

```
        perror ("semctl error!\n");
        exit (1);
    }

    while (1)
    {
        if(semop(semid, &sbuf, 1) == -1)
        {
            perror ("semop error!\n");
            exit (1);
        }
        sleep (3);
    }

    exit (0);
}
```

例 10-16 clinet.c

```
#include <sys/types.h>
#include <linux/sem.h>

int main(void)
{
    key_t    key;
    int      semid, semval;
    union semun   semopts;

    if((key = ftok (".",'s')) == -1)
    {
        perror ("ftok error!\n");
        exit (1);
    }

    if((semid = semget (key, 1, IPC_CREAT | 0666)) == -1)
    {
        perror ("semget error!\n");
        exit (1);
    }

    while(1)
    {
        if ((semval = semctl(semid, 0, GETVAL, 0)) == -1)
        {
            perror ("semctl error!\n");
            exit (1);
        }
         if (semval > 0)
        {
            printf ("Still %d resources can be used\n", semval);
        }
        else
        {
            printf ("No more resources can be used!\n");
            break;
        }

        sleep (3);
    }

    exit (0);
}
```

程序说明。

首先在一个终端上编译并运行 server.c，再在另外一个终端上编译运行 client.c。观察 client 的运行结果如下：

```
[zzy@mci ~]# ./server
[zzy@mci ~]# ./client
```

命令：ipcs -s //查看创建的信号量集合的个数

ipcrm -s semid //删除一个信号量集合

我爱自由 5ifreedom.com

```
Still 4 resources can be used
Still 3 resources can be used
Still 2 resources can be used
Still 1 resources can be used
No more resources can be used!
```

由运行结果可以看出，信号量可以实现锁的功能。

10.6 共享内存

10.6.1 共享内存的数据结构

共享内存就是分配一块能被其他进程访问的内存。每个共享内存段在内核中维护着一个内部结构 shmid_ds（和消息队列、信号量一样），该结构定义在头文件 linux/shm.h 中，代码如下所示：

```
struct shmid_ds{
    struct ipc_perm        shm_perm;;
    int                    shm_segsz
  __kernel_time_t          shm _atime;
  __kernel_time_t          shm _dtime;
  __kernel_time_t          shm _ctime;
  __kernel_ipc_pid_t       shm_cpid;
  __kernel_ipc_pid_t       shm_lpid;
    ushort                 shm_nattch;
    ushort                 shm_unused;
    void               *shm_unused2;
    void               *shm_unused3;
};
```

```
/* Obsolete, used only for backwards compatibility and libc5 compiles */
struct shmid_ds {
    struct ipc_perm shm_perm;/* operation perms */
    int shm_segsz;/* size of segment (bytes) */
    __kernel_time_t shm_atime;/* last attach time */
    __kernel_time_t shm_dtime;/* last detach time */
    __kernel_time_t shm_ctime;/* last change time */
    __kernel_ipc_pid_t shm_cpid;/* pid of creator */
    __kernel_ipc_pid_t shm_lpid;/* pid of last operator */
    unsigned short shm_nattch;/* no. of current attaches */
    unsigned short shm_unused;/* compatibility */
    void *shm_unused2;/* ditto - used by DIPC */
    void *shm_unused3;/* unused */
};
```

代码中主要字段含义如下。

- shm_perm：操作许可，里面包含共享内存的用户 ID、组 ID 等信息。
- shm_segsz：共享内存段的大小，以单位为字节。
- shm _atime：最后一个进程访问共享内存的时间。
- shm _dtime：最后一个进程离开共享内存的时间。
- shm _ctime：最后一次修改共享内存的时间。
- shm_cpid：创建共享内存的进程 ID。
- shm_lpid：最后操作共享内存的进程 ID。
- shm_nattch：当前使用该共享内存段的进程数量。

10.6.2 共享内存的创建与操作

1. 共享内存区的创建

Linux 下使用函数 shmget 来创建一个共享内存区，或者访问一个已存在的共享内存区。该函数定义在头文件 linux/shm.h 中，原型如下：

```
int shmget(key_t key, size_t size, int shmflg);
```

函数中：参数 key 是由 ftok()得到的键值；参数 size 以字节为单位指定内存的大小；shmflg 为操作标志位，它的值为一些宏，如下所示。

- IPC_CREATE：调用 shmget 时，系统将此值与其他所有共享内存区的 key 进行比较，如果存在相同的 key，说明共享内存区已存在，此时返回该共享内存区的标识符，否则新建一个共享内存区并返回其标识符。
- IPC_EXCL：该宏必须和 IPC_CREATE 一起使用，否则没有意义。当 shmflg 取

我爱自由 5ifreedom.com

IPC_CREATE | IPC_EXCL 时，表示如果发现信号集已经存在，则返回-1，错误码为 EEXIST。

注意：当创建一个新的共享内存区时，size 值必须大于 0；如果是访问一个已存在的共享内存区，置 size 为 0。

2. 共享内存区的操作

在使用共享内存区前，必须通过 shmat 函数将其附加到进程的地址空间。进程与共享内存就建立了连接。shmat 调用成功后就会返回一个指向共享内存区的指针，使用该指针就可以访问共享内存区了，如果失败返回-1。该函数声明在 linux/shm.h 文件中，具体结构代码原型如下：

```
void* shmat (int shmid, const void * shmaddr, int shmflg);
```

参数 shmid 为 shmget 的返回值；参数 shmflg 为存取权限标志；参数 shmaddr 为共享内存的附加点。参数 shmaddr 不同取值情况的含义说明如下：

- 如果为空，则由内核选择一个空闲的内存区；如果非空，返回地址取决于调用者是否给 shmflg 参数指定了 SHM_RND 值，如果没有指定，则共享内存区附加到由 shmaddr 指定的地址；否则附加地址为 shmaddr 向下舍入一个共享内存低端边界地址后的地址（SHMLBA，一个常址）。
- 通常将参数 shmaddr 设置为 NULL。

当进程结束使用共享内存区时，要通过函数 shmdt 断开与共享内存区的连接。该函数声明在 sys/shm.h 文件中，具体结构代码原型如下：

```
int shmdt (const void* shmaddr);
```

参数 shmaddr 为 shmat 函数的返回值。该函数调用成功后，返回 0，否则返回-1。进程脱离共享内存区后，数据结构 shmid_ds 中的 shm_nattch 就会减 1。但是共享内存段依然存在，只有 shm_nattch 为 0 后，即没有任何进程再使用该共享内存区，共享内存区才在内核中被删除。一般来说，当一个进程终止时，它所附加的共享内存区都会自动脱离。

3. 共享内存区的控制

Linux 对共享内存区的控制是通过调用函数 shmctl 来完成的，该函数定义在头文件 sys/shm.h 中，原型代码如下所示：

```
int shmctl (int shmid, int cmd, struct shmid_ds *buf);
```

函数中：参数 shmid 为共享内存区的标识符；buf 为指向 shmid_ds 结构体的指针；cmd 为操作标志位，支持一下 3 种控制操作。

- IPC_RMID：从系统中删除由 shmid 标识的共享内存区。
- IPC_SET：设置共享内存区的 shmid_ds 结构。
- IPC_STAT：读取共享内存区的 shmid_ds 结构，并将其存储到 buf 指向的地址中。

10.6.3 共享内存的应用实例

本例 10-17 通过读写者问题（不考虑优先级）来演示共享内存和信号量如何配合使用。这里的读者写者问题要求一个进程读共享内存的时候，其他进程不能写内存；当一个进程写共享内存的时候，其他进程不能读内存。

程序首先定义了一个包含公用函数的头文件 sharemem.h。

例 10-17 sharemem.h

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

我爱自由 5ifreedom.com

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <errno.h>

#define SHM_SIZE 1024

union semun{
    int                 val;
    struct semid_ds     *buf;
    unsigned short      *array;
};

/*创建信号量函数*/
int createsem (const char * pathname, int proj_id, int members, int init_val)
{
    key_t           msgkey;
    int                 index, sid;
    union semun         semopts;

    if ((msgkey = ftok(pathname, proj_id)) == -1)
    {
        perror ("ftok error!\n");
        return -1;
    }

    if ((sid = semget (msgkey, members, IPC_CREAT | 0666)) == -1)
    {
        perror ("semget call failed.\n");
        return -1;
    }

    /*初始化操作*/
    semopts.val = init_val;
    for (index = 0; index < members; index++)
    {
        semctl (sid, index, SETVAL, semopts);
    }
    return (sid);
}

/*打开信号量函数*/
int opensem(const char * pathname, int proj_id)
{
    key_t     msgkey;
    int       sid;

    if ((msgkey = ftok(pathname, proj_id)) == -1)
    {
        perror ("ftok error!\n");
        return -1;
    }

    if ((sid = semget(msgkey, 0, IPC_CREAT | 0666)) == -1)
    {
        perror("semget call failed.\n");
        return -1;
    }

    return (sid);

}

/*P 操作函数*/
int sem_p(int semid, int index)
{
```

我爱自由 5ifreedom.com

```
    struct sembuf buf = {0,-1,IPC_NOWAIT};

    if (index < 0)
    {
    perror("index of array cannot equals a minus value!");
    return -1;
    }

    buf.sem_num = index;
    if (semop (semid ,& buf,1) == -1)
    {
         perror ("a wrong operation to semaphore occurred!");
         return -1;
    }
    return 0;
}

/*V操作函数*/
int sem_v (int semid, int index)
{
    struct sembuf buf = {0, +1, IPC_NOWAIT};

    if (index < 0)
    {
         perror("index of array cannot equals a minus value!");
         return -1;
    }

    buf.sem_num = index;
    if (semop (semid,& buf,1) == -1)
    {
         perror ("a wrong operation to semaphore occurred!");
         return -1;
    }

    return 0;
}

/*删除信号集函数*/
int sem_delete (int semid)
{
    return (semctl(semid, 0, IPC_RMID));
}

/*等待信号为1*/
int wait_sem( int semid, int index)
{

    while (semctl (semid, index, GETVAL, 0) == 0)
    {
         sleep (1);
    }

    return 1 ;
}


/*创建共享内存函数*/
int createshm( char * pathname, int proj_id, size_t size)
{
    key_t     shmkey;
    int       sid;

    /*获取键值*/
    if ((shmkey = ftok(pathname, proj_id)) == -1)
    {
         perror("ftok error!\n");
```

我爱自由 5ifreedom.com

```
        return -1;
    }

    if ((sid = shmget(shmkey, size, IPC_CREAT | 0666)) == -1)
    {

        perror ("shmget call failed.\n");
        return -1;
    }
    return (sid);
}
```

例 10-18 和例 10-19 是 writer 和 reader 程序，两程序在进入共享内存区之前，首先都检查信号集中信号的值是否为 1（相当于是否能进入共享内存区），如果不为 1，调用 sleep()进入睡眠状态直到信号的值变为 1。进入共享内存区之后，将信号的值减 1（相当于加锁），这样就实现了互斥的访问共享资源。在退出共享内存时，将信号值加 1（相当于解锁）。

例 10-18 writer.c

```
#include <sharemem.h>

int main()
{
    int         semid, shmid;
    char        *shmaddr;
    char        write_str[SHM_SIZE];

    if ((shmid = createshm (".", 'm', SHM_SIZE)) == -1)
    {
        exit(1);
    }

    if ((shmaddr = shmat (shmid, (char *)0, 0)) ==(char *)-1)
    {
        perror ("attach shared memory error!\n");
        exit (1);
    }

    if ((semid = createsem (".", 's', 1, 1)) == -1)
    {
        exit (1);
    }

    while (1)
    {
        wait_sem (semid, 0);
        sem_p (semid, 0);       /*P 操作*/

        printf ("writer: ");
        fgets (write_str, 1024, stdin);
        int len = strlen (write_str) - 1;
        write_str[len] = '\0';
        strcpy (shmaddr, write_str);
        sleep (10);             /*使 reader 处于阻塞状态*/

        sem_v (semid, 0);       /*V 操作*/
        sleep (10);             /*等待 reader 进行读操作*/

    }
}
```

例 10-19 reader.c

```
#include "sharemem.h"

int main()
{
```

我爱自由 5ifreedom.com

```
    int         semid, shmid;
    char        *shmaddr;

    if ((shmid = createshm(".", 'm', SHM_SIZE)) == -1)
    {
        exit (1);
    }

    if((shmaddr = shmat (shmid, (char *)0, 0)) == (char *)-1)
    {
        perror ("attach shared memory error!\n");
        exit (1);
    }

    if((semid = opensem("." ,'s')) == -1)
    {
        exit (1);
    }

    while(1)
    {
        printf("reader: ");
        wait_sem(semid,0);          /* 等待信号值为 1 */
        sem_p(semid,0);                 /* P 操作 */

        printf("%s\n", shmaddr);
        sleep(10);                      /* 使 writer 处于阻塞状态 */

        sem_v(semid,0);                 /* V 操作 */
        sleep(10);                      /* 等待 writer 进行写操作 */
    }
}
```

程序说明。

编译两个程序，注意其中的-I引用的是头文件 sharemem.h 所在目录。

```
[zzy@mci ~]# gcc -o writer -I/home/zzy writer.c
[zzy@mci ~]# gcc -o reader -I/home/zzy reader.c
```

同时在两个终端运行 writer 和 reader，在 writer 端输入字符串“hello，reader”，等待一会，看到 reader 端输出后，在 writer.c 端的提示符后面输入字符串“new inforamation”。执行的结果如下：

```
[zzy@mci ~]# ./writer
writer: hello, reader!
writer: new inforamation
writer:
[zzy@mci ~]#  ./reader!
reader: hello, reader
reader: new inforamation
```

从运行结果可以看出，writer 和 reader 进程是同步的，writer 写入的信息均被 reader 完整正确的读出，即使去掉程序中的 sleep()也不会影响运行结果。

10.7 库的创建和使用

10.7.1 Linux 库的概念

库是一种软件组件技术，库里面封装了数据和函数，提供给用户程序调用。库的使用可以使程序模块化，提高程序的编译速度，实现代码重用，使程序易于升级。因此，对于软件开发人员，掌握这项技术是很必要的。

Windows 系统本身提供并使用了大量的库，包括静态链接库（.lib 文件）和动态链接库（.dll

我爱自由 5ifreedom.com

文件）。类似的，Linux 操作系统也使用库。Linux 系统中，通常把库文件存放在/usr/lib 或/lib 目录下。Linux 库文件名由前缀 lib、库名以及后缀 3 部分组成，其中动态库以.so 作为后缀，而静态库通常以.a 作为后缀。

在程序中使用静态库和动态库时，它们的载入顺序是不一样的。静态库的代码在编译时就拷贝到应用程序中，因此当多个应用程序同时引用一个静态库函数时，内存中将会有调用函数的多个副本。这样的优点是节省编译时间。而动态库是在程序开始运行后调用库函数时才被载入，被调函数在内存中只有一个副本，并且动态库可以在程序运行期间释放动态库所占用的内存，腾出空间供其他程序使用。下面从编写库函数、编译生成库文件、调用库函数 3 个方面来介绍静态库和动态库的创建与使用。

10.7.2 静态库的创建和使用

创建静态库的步骤如下。

（1）在一个头文件中声明静态库所导出的函数。

（2）在一个源文件中实现静态库所导出的函数。

（3）编译源文件，生成可执行代码。

（4）将可执行代码所在的目标文件加入到某个静态库中，并将静态库拷贝到系统默认的存放库文件的目录下。

以下通过实例来说明如何创建和使用静态库。mylib.h 中存放的是静态库提供给用户使用的函数的声明，mylib.c 实现了 mylib.h 中声明的函数。

头文件 mylib.h

```
#ifndef _mylib_h_
#define _mylib_h_
void welcome( );
void outstring(const char * str);
#endif
```

对应于头文件 mylib.h 的源文件 mylib.c

```
#include "mylib.h"
#include <stdio.h>
void welcome( )
{
     printf ("Welome to libmylib\n");
}
void outstring(const char * str)
{
     if (str != NULL)
          printf ("%s",str);
}
```

编译 mylib.c 生成目标文件：

```
gcc -o mylib.o -c mylib.c
```

将目标文件加入到静态库中，静态库为 libmylib.a

```
ar rcs libmylib.a mylib.o
```

将静态库拷贝到 Linux 的库目录（/usr/lib 或/lib）下：

```
cp libmylib.a /usr/lib/libmylib.a
```

以下为调用库函数的测试程序 test.c

```
#include "mylib.h"
#include <stdio.h>
int main ( )
{
          printf ("create and use library:\n")
```

我爱自由 5ifreedom.com

```
        welcome( );
        outstring("It's successful\n");
}
```

编译使用了库函数的程序：

```
gcc -o test test.c -lmylib
```

注意：-lmylib" 中-l 为选项，mylib 为库名。mylib 是"libmylib.a"的中间部分，Linux 下约定所有库都以前缀 lib 开始，静态库以.a 结尾，动态库以.so 结尾。在编译程序时，无需带上前缀和后缀。

运行生成的可执行程序 test：

```
./test
```

程序输出：

```
create and use library:
Welome to libmylib
It's successful
```

Linux 下，可以使用 ar 命令来创建和修改静态库。示例命令如下：

```
ar rcs libmylib.a file1.o file2.o
```

该命令将目标代码 file1.o 和 file2.o 加入到静态库 libmylib.a 中，加入时如果静态库 libmylib.a 不存在，则会自动创建一个静态库。rcs 是命令 ar 的选项。以下是 ar 命令的使用格式：

```
ar [-]{dmpqrtx} [abcfilNoPsSuvV] 库名 库中的成员文件名
```

其中的"-"不是必需的，"dmpqrtx"这些字母在每个命令中只能有一个且必须有一个。"abcfilNoPsSuvV"中的字母在每个命令中可以有 0 到多个。

常用选项含义如下。

d：从库中删除成员文件。

m：在库中移动成员文件。当库中的若干成员文件有相同的符号时（如相同的函数名），成员的位置顺序很重要。如果没有指定其他选项，任何指定的成员将被移到库的最后。也可以使用'a'，'b'，或'I'选项移动到指定的位置。

p：在终端上打印库中指定的成员。

q：快速追加。增加新成员文件到库的结尾处而不检查库中是否存在同名的成员文件。

r：在库中加入成员文件，如果要加入的成员文件在库中已存在，则替换之。默认的情况下，新的成员增加在库的结尾处，可以使用其他选项来改变加入的位置。

t：显示库的成员文件清单。

x：从库中提取一个成员文件。如果不指定要提取的成员文件则提取库中所有的文件。

a：在库的一个已经存在的成员后面增加一个新的成员文件。

b：在库的一个已经存在的成员前面增加一个新的成员文件。

c：创建一个库。

i：在库的一个已经存在的成员前面增加一个新的成员文件，类似选项 b。

l：暂未使用。

s：无论 ar 命令是否修改了库内容，都强制重新生成库符号表。

u：插入并列出文件中那些比库中同名文件新的文件，该选项必须和 r 选项一起使用。

v：用来显示操作的附加信息。

V：显示 ar 的版本信息。

10.7.3 动态库的创建和使用

在 Linux 环境下，可以很方便地创建和使用动态链接库。只要在编译函数库源程序时加上

我爱自由 5ifreedom.com

-shared 选项即可，这样所生成的可执行程序就为动态链接库。从某种意义上来说，动态链接库也是一种可执行程序。按一般规则，动态链接库以.so 后缀。下面命令把 myLib.c 程序创建成了一个动态库：

```
gcc -fPIC -o mylib.o -c mylib.c
gcc -shared -o libttt.so mylib.o
```

也可以直接使用一条命令：

```
gcc -fPIC -shared -o libttt.so mylib.c
```

动态链接库创建后就可以使用了。Linux 下有两种方式调用动态链接库中的函数，一种是像使用静态库一样，通过 gcc 命令调用，命令格式如下：

```
gcc -o main mian.c ./libmylib.so
```

或者：

```
cp libttt.so /usr/lib/libttt.so
gcc -o test test.c  /usr/lib/libttt.so
```

注意：引用动态链接库时，必须含有路径。如果只是使用 libmylib.so，则必须确保这个库所在目录包含在 PATH 环境变量中。

运行程序，结果如下：

```
[zzy@mci ~]$ ./test
lib study
Welome to libmylib
It's successful
```

另一种方法是通过调用系统函数来使用动态链接库，相关函数如表 10-2 所示。

表 10-2　　调用动态库常用的系统函数

函　数	说　明
void* dlopen（const char *filename, int flag）	用于打开指定名字的动态链接库，并返回一个句柄
void* dlsym（void *handle, char *symbol）	根据动态链接库的句柄与函数名，返回函数名对应的函数的地址
int dlclose （void *handle）	关闭动态链接库，handle 是调用 dlopen 函数返回的句柄
const char *dlerror（void）	动态链接库中的函数执行失败时，dlerror 返回出错信息；若执行成功，则返为 NULL

dlopen 函数的参数 flag 可取的值有：RTLD_LAZY、RTLD_NOW、RTLD_GLOBAL。其含义如下。

RTLD_LAZY：在 dlopen()返回前，对于动态库中存在的未定义的变量（如外部变量 extern，也可以是函数）不执行解析，也就是不解析这个变量的地址。

RTLD_NOW：与 RTLD_LAZY 不同，在 dlopen 返回前，解析出每个未定义变量的地址，如果解析不出来，dlopen 会返回 NULL，错误为“undefined symbol: xxxx...”。

RTLD_GLOBAL：使库中被解析出来的变量在随后的其他链接库中也可以使用，即全局有效。

下面通过例 10-20 演示其用法。

例 10-20　testso.c

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

int main(void)
{
    void *handle;
    char *error;
    void (*welcome)( );

    if ((handle = dlopen("./libttt.so", RTLD_LAZY)) == NULL)
```

我爱自由 5ifreedom.com

```
    {
        printf ("dlopen error\n");
        exit (1);
    }

    welcome = dlsym(handle, "welcome");
    if ((error = dlerror()) != NULL)
    {
        printf ("dlsym error\n");
        exit (1);

    }

    welcome ();
    dlclose (handle);

    exit (0);
}
```

编译运行此程序，结果为：

```
[zzy@mci ~]$ gcc -o testso  testso.c -ldl
[zzy@mci ~]$ ./testso
Welome to libmylib
```

程序说明

“-ldl”指明 dlopen 等函数所在的库。程序成功调用了动态链接库 libttt.so 中的 welcome()函数。通过在程序中调用系统函数来使用动态连接库时，动态链接库所在的目录须包含在程序中，如程序中“dlopen ("./libttt.so", RTLD_LAZY)”，如果动态库 libttt.so 不在当前目录下，程序运行时将出错。

10.8 进一步学习建议

第 6 章至第 10 章主要介绍了 Linux 下的系统编程，包括文件和目录操作、进程和线程控制、信号的使用、进程间通信。关于这些内容也可以参考《UNIX 环境高级编程》，该书由已故的著名计算机专家 W.Richard Stevens 编著，为 UNIX 系统编程的经典之作，历经十几年经久不衰。Linux 是一种类 UNIX 操作系统，因此该书对 Linux 下的系统编程有很好的借鉴意义，不过该书内容比较庞杂，不太适合于初学者，对于有一定基础的读者，阅读该书会受益匪浅。还可以参考 Rich Teer 的《Solaris 系统编程》，Solaris 是一种 UNIX 操作系统。该书详述了 Solaris 操作系统的系统编程接口，以大量的案例、代码和图示解释如何使用各个编程接口进行进程控制、信号操作、进程间通信。

如果要深入探究 Linux 内核的实现，可以参考 Daniel P. Bovet 和 Marco Cesati 合著的《深入理解 Linux 内核》第三版，它深入剖析了 Linux 内核的数据结构、算法和关键程序，揭密了 Linux 内核的实现原理。一般的内核开发者都是在理解 Linux 内核工作原理的基础上开发驱动程序。介绍 Linux 下如何开发驱动程序的经典之作是 Jonathan Corbet 等人编写的《Linux 设备驱动程序》第三版。

10.9 习题

1．进程间通信的方式有哪些，各有什么不同？

我爱自由 5ifreedom.com

2．对管道和有名管道的操作过程有什么不同，为什么？

3．编写一个程序：父子进程间通过管道进行通信，尝试让进程在管道读端写数据，在管道写端读数据，看有什么错误发生，并显示对应的错误信息？

4．编写一个程序完成以下工作：在一个进程里创建两个子进程，两兄弟进程通过管道进行信息传递。

5．在一个程序里面定义 5 个全局共享变量，同时创建 5 个进程，如何保证这 5 个进程访问这 5 个共享变量时同步？

6．进程间使用消息队列进行通信的时候，可根据指定的消息类型取得对应的信息，那么当给定的类型不存在的时候，会产生什么样的错误？请编程来验证。

7．通过共享内存实现一个聊天程序。

8．编写一个综合运用 4 种进程间通信方式的程序。

9．试比较 Linux 静态库和动态库的不同。编写一个库的源程序，分别生成相应的动态库和静态库；通过一个程序分别对两者进行调用，试比较采用不同库时程序运行时间的长短，给出结论。

10．消息队列、信号量和共享内存均位于系统内核中，而且使用的方式也非常类似，编程实现一个公用的接口函数库，库中实现对 3 种通信方式所用系统函数的统一，比如它们的创建通过调用一个函数就可以实现，而不需要调用不同的函数来实现。实现后，分别针对每种通信方式编写程序进行测试。

我爱自由 5ifreedom.com

LINUX

第三篇　Linux 网络和图形界面编程

网络编程

GTK+图形界面编程

我爱自由
5ifreedom.com

第 11 章 网络编程

本章主要介绍 Linux 下的网络编程。首先对网络编程必需的理论基础，如网络模型、地址、端口、TCP/IP 协议，作简要的介绍。然后介绍套接字的操作函数，并结合一个基于 Client/Server 模型、面向连接的程序实例来说明，如何综合运用这些套接字操作函数进行网络通信。本章还介绍了网络攻击、代码安全等知识，最后以一个端口扫描的程序作为本章的结束。

本章重点：

- 地址与端口。
- TCP/IP 协议。
- Client / Server 模型。
- 套接字的创建、连接、绑定、监听、接收。
- 使用套接字传输数据。
- 网络编程涉及的一些实用函数。
- 网络攻击与代码安全。

本章难点：

- 基于 Client/Server 模型的网络程序。
- 套接字选项的获取和设置。
- 服务器端进程的设计。
- 端口扫描。

11.1 网络编程基本原理

TCP/IP 协议是目前世界上使用最广泛的网络通信协议，日常生活中的大部分网络应用（如浏览网页、收发电子邮件、QQ 聊天）都是基于该系列协议。在进行网络编程之前，首先需要具备一些网络相关的理论基础，只有这样才能编写出功能强大的网络应用程序。

11.1.1 网络模型与协议

为了减少协议设计的复杂性，大多数网络模型都是按层（layer）的方式来组织的。在分层网络模型中，每一层都为上一层提供一定的服务，而把如何实现本层服务的细节对上一层加以屏蔽。上层只需知道下层提供了什么功能以及对应于这些功能的接口，而不必关心下一层如何实现这些功能。分层的思想也是设计大型软件的一种重要思想，有些操作系统就是基于分层的思想设计和实现的。

为了确保使用不同硬件和底层协议构建的网络能相互进行通信，国际标准化组织（ISO）制定了一套被称为开放系统互联（OSI）的规范。但由于种种原因，OSI 模型始终没有得到广泛应用。当前普遍使用的是 TCP/IP 模型。几乎所有互联网设备都支持 TCP/IP 协议。TCP/IP 协议已经成为事实上的国际标准和工业标准。ISO 模型和 TCP/IP 模型的层次划分与对比如图 11-1 所示。

TCP/IP 各层功能如下。

我爱自由 5ifreedom.com

（1）网络接口层

网络接口层是 TCP/IP 模型最下一层，它包括多种逻辑链路控制和媒体访问协议。网络接口层负责将 Internet 层发送来的数据分成帧，并通过物理链路进行传送，或从网络上接收物理帧，抽取数据并转交给其上的 Internet 层。

（2）Internet 层（网络层）

网络层负责在发送端和接收端之间建立一条虚拟路径。这一层的主要协议是 IP 协议。IP 协议并不保证数据能完整正确地到达目的地，这个任务由它上面的传输层来完成。这一层的 ARP 协议（地址解析协议）和 RARP 协议（反向地址解析协议）用于 IP 地址和物理地址（通常就是网卡地址）的相互转换。如果数据在传输过程中出现问题，该层的 ICMP 协议将生产错误报文。

（3）传输层

传输层通过位于该层的 TCP 协议（传输控制协议）或 UDP 协议（用户数据报协议）在两台主机间传输数据。其中 TCP 协议提供可靠的面向连接的服务，它保证数据能完整地按顺序地传送到目标计算机。它在传输数据前首先需要和目的计算机建立连接，并且在数据传输过程中维持此连接，因此在速度上会有些损失。UDP 提供简单的无连接服务，它不保证数据能按顺序、正确地传送到目的地（但可由它的上层来保证），它不用建立连接，通常速度上要比 TCP 快些。TCP 协议和 IP 协议都需要网络层提供通往目的地的路由。传输层提供端到端，即应用程序之间的通信。该层的主要功能有差错控制、传输确认和丢失重传等。

（4）应用层

应用层面向用户提供一系列访问网络的协议，如用于传输文件的 FTP 协议、用于远程登录的 Telnet 协议、用于发送电子邮件的 SMTP 协议（简单邮件传输协议），以及最常用的用于浏览网页的 HTTP 协议（超文本传输协议）。还有近几年来十分流行的点对点共享文件协议，即 BitTorrent 协议，该协议基于 HTTP 协议。使用该协议构建的 BT 下载工具有比特精灵、BitTorrent 等。

比特流

上面提到了的协议，简单地讲就是通信实体为实现正确的通信而制定的规约。协议就像是人类的语言，人类进行交流需要一种双方都能理解的语言，同样，两台计算机要进行通信也需要一种双方都能理解的语言，这就是协议。TCP/IP 是由许多协议构成的协议簇，如 TCP、UDP、IP、FTP 和 HTTP 等，它们的关系如图 11-2 所示。

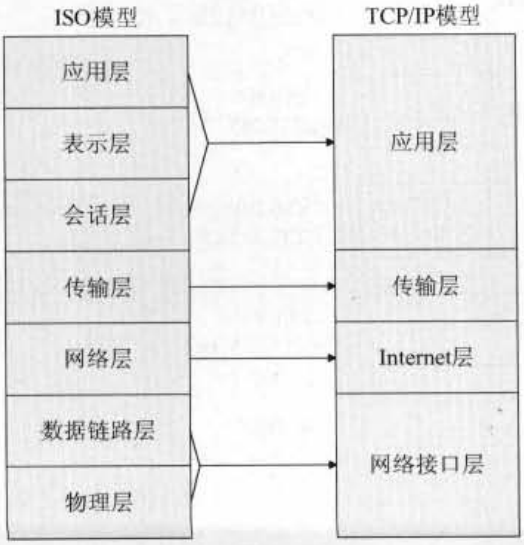


图 11-1　网络分层模型

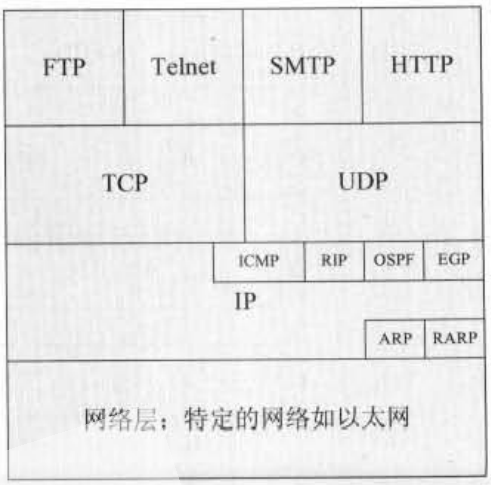


图 11-2　TCP/IP 协议

我爱自由 5ifreedom.com

11.1.2 地址

为了使网络上的计算机能进行互相通信，必须有一个惟一的标识来区分网络上每台计算机（就像用身份证来区别每个人一样），或者说网络上每台计算机需要一个惟一的标识。有两种标识可以使用：物理地址（如网卡地址）和 IP 地址。

1. 物理地址

对于以太网来说，物理地址就是一个 48 位的位串，此地址在网卡的生产过程中就已经固定了，是不可更改的，并且是全球惟一的。在 Shell 下输入命令 ifconfig 可以查看到本机的物理地址：

```
[root@localhost clh]# ifconfig
eth0   Link encap:Ethernet   HWaddr    00:0D:87:9F:47:B8
```

其中的 00:0D:87:9F:47:B8 就是以 16 进制表示的 48 位（6 字节）网卡地址，每个字节用冒号隔开。有些计算机可能会有多块网卡，每块网卡代表计算机的一个网络接口，我们称这种计算机为多宿主计算机（Multihomed Computer）。

2. IP 地址

TCP/IP 协议能够使计算机之间进行与底层物理网络无关的通信，底层网络可以是以太网也可以是令牌环网或者是其他类型的网，两台计算机可以位于不同的局域网内，但通过 TCP/IP 协议它们也能够进行通信。前面介绍的物理地址（如网卡地址）虽然也能惟一的标识网络上的每台计算机，但是由于物理地址依赖于底层网络（不同的底层网络采用不同的物理地址），因此必须使用一个与底层硬件技术无关的通用地址来标识网络上的每台计算机，这就是 IP 地址。IP 地址由 32 个比特位构成，它分为两部分：计算机所在的网络号和该网络给该计算机分配的主机号，分别称为网络 ID 和主机 ID。

32 位（4 字节）的 IP 地址为了表示方便也是每字节隔开，不过不是使用冒号而是使用点号，例如，腾讯网（www.qq.com）的 IP 地址是 202.205.3.195。一个字节可以表示的数的范围是 00000000～11111111（即 0～255）。因此在理论上 IP 地址的范围为：0.0.0.0～255.255.255.255，但实际上有些地址是专用的，不能用来标识计算机。

IP 地址按一定的格式分成 5 类：A 类、B 类、C 类、D 类和 E 类。它们的格式如图 11-3 所示。

图 11-3 IP 地址格式

A 类地址使用 8 位作为网络地址，其余 24 位为主机地址，网络地址的第一位固定为 0。B 类

我爱自由 5ifreedom.com

地址使用 16 位作为网络地址，其余 16 位为主机地址，网络地址的第一、二位固定为 10。C 类地址使用 24 位作为网络地址，其余 8 位为主机地址，网络地址的第一、二、三位固定为 110。D 类地址和 E 类地址较少使用，这里不作介绍。

127.0.0.1 是一个特殊的地址，它指代本机，用于测试本机上的 TCP/IP 协议是否正常工作。输入命令“ping 127.0.0.1”，如果有回应说明本机上的 TCP/IP 协议工作正常。

TCP/IP 上的每台主机还有一个 32 位的子网掩码，它用来区分 IP 地址的网络号和主机号。将 IP 地址与子网掩码进行按位“与”运算就可以得到 IP 地址的网络号，网络号是一台主机所处的网络的编号。例如，有一台主机的 IP 地址为 222.197.168.244，对应的子网掩码为 255.255.255.0，两者作“与”运算得到的结果为 222.197.168.0，那么这台主机所处的网络的编号为 222.197.168.0。

近年来，随着越来越多的计算机连入互联网中，IP 地址已经快耗尽了。于是提出了 IPv6，IPv6 使用一个 128 位的 IP 地址来标识一台计算机，原来的版本称为 IPv4。使用 IPv6，IP 地址很难被耗尽，因为它可以标识的计算机数是超天文数字（2^{128}）。基于 IPv6 的网络技术现在正在研发中，预计在 2020 年前后进行大规模的实际应用，我国在这一领域与国外同时起步，在某些方面已经处于领先地位。

11.1.3 端口

有了网络地址，就可以惟一地标识网络上的每台计算机。通常一台计算机上会同时运行多个程序，而它们可能同时要访问网络。对于一台计算机上的多个应用程序，TCP 和 UDP 协议采用 16 位的端口号来识别它们。一台主机上的不同进程可以绑定到不同的端口上，这些进程都可以访问网络而互不干扰。

TCP/IP 将端口号（端口号是一个 16 位的无符号整数，因此端口号的范围是 0～2^{16}，即 0～65535）分为两部分：一部分是保留端口即知名端口，范围为 0～1023，这些端口由权威机构规定其用途，如编号为 21 的 TCP 端口由 FTP 协议专用，80 号 TCP 端口由 HTTP 协议专用；其余的为自由端口，用户进程可以自由申请和使用。

11.1.4 IP 协议

IP 协议（Internet Protocal）是网络层最重要的协议。无论传输层使用何种协议，都要依靠 IP 协议来确定到达目的计算机的路由。IP 协议主要负责确定路由，当到达同一目的地有多条路由时，IP 协议会选择一条最短路由来将数据分组传送到目的计算机。同时，IP 协议还定义了一组规则，例如，有时目的地不可达或不存在，IP 协议规定了在这种情况下何时丢弃传送中的数据分组。IP 协议定义了数据单元格式，称为 IP 数据报。它由 IP 首部和数据两部分组成，IP 数据报如图 11-4 所示。

图 11-4 IP 首部的格式

主要字段的含义如下。

（1）版本

表示该数据报采用的是哪个版本的 IP 协议。该字段长度为 4 位。对于 IPv4，这个字段的值为 4，IPv6 对应的值为 6。

我爱自由 5ifreedom.com

（2）总长度

IP 首部和数据的总长度。

（3）生存期

该字段表示数据报在网络上的最大生存时间（Time to Live，TTL）。数据报每经过一个路由器，路由器将 TTL 值减 1，当 TTL 减为 0 时，数据报将被丢弃，并且该路由器会向发送者返回一个 ICMP 超时报文，告知数据报被丢弃。TTL 的默认值是 64，最大值是 255。

（4）协议

该字段用于说明发送数据报所使用的协议，如果 0x06 表示使用 TCP 协议传输数据,0x11 表示使用 UDP 协议。

（5）报头校验和

该字段用于检查 IP 首部的完整性，该字段只校验 IP 首部，不校验数据。

（6）源 IP 地址

该字段为发送数据报的源计算机 IP 地址。

（7）目的 IP 地址

该字段为接收数据报的目的计算机 IP 地址。

（8）IP 选项

该字段是一个可选的字段，主要用于网络调试。

11.1.5 用户数据报协议 UDP

在 TCP/IP 模型中，UDP 协议位于传输层，在网络层之上而在应用层之下。UDP 协议向应用程序提供一种面向无连接的服务，通常 UDP 协议被用于不需要可靠数据传输的网络环境中。UDP 不需要建立连接，应用程序采用 UDP 协议无需建立和维持连接。UDP 协议不保证数据报按顺序、正确地到达目的地，这项任务由应用程序来完成。

UDP 数据报首部格式如图 11-5 所示。

源端口（16）	目端口（16）
长度（16）	UDP校验和（16）
数据	

图 11-5 UDP 首部的格式

UDP 首部的各部分含义如下。

（1）源端口

发送 UDP 数据的源端口号。

（2）目标端口

接收 UDP 数据的目的端口号。

（3）长度

该字段表示包括 UDP 首部和数据在内的整个数据报的长度，以字节为单位。

（4）UDP 校验和

该字段是根据 IP 首部、UDP 首部和数据计算出来的值，当该字段被设置为 0×0000 时，表明发送端计算机没有计算校验和。

11.1.6 传输控制协议 TCP

1. TCP 数据包的格式

TCP 提供一种面向连接的、可靠的数据传输服务。

我爱自由 5ifreedom.com

TCP 数据包的首部格式如图 11-6 所示。

主要字段的含义如下。

（1）源端口

发送 TCP 数据的源端口号。

（2）目标端口

接收 TCP 数据的目的端口号。

<table>
<tr><td colspan="3">源端口（16）</td><td colspan="2">目标端口（16）</td></tr>
<tr><td colspan="5">序号（32）</td></tr>
<tr><td colspan="5">确认号（32）</td></tr>
<tr><td>首部长度（4）</td><td>保留（4）</td><td>代码位（6）</td><td colspan="2">窗口（16）</td></tr>
<tr><td colspan="3">校验和（16）</td><td colspan="2">紧急指针（16）</td></tr>
<tr><td colspan="4">选项（如果有的话）</td><td>填充</td></tr>
<tr><td colspan="5">数据</td></tr>
</table>

图 11-6　TCP 的首部格式

2. 使用 TCP 进行通信的过程

（1）建立连接

第一步：连接的发起端（通常称为客户端）向目标计算机（通常称为服务器）发送一个请求建立连接的数据包。

第二步：服务器收到请求后，对客户端的同步信号作出响应，并发送自己的同步信号给客户端。

第三步：客户端对服务器端发来的同步信号进行响应。连接建立完成，就可以进行数据传输了。

这三个步骤顺利完成后，连接建立，这个过程也被称为“三次握手”。

（2）关闭

第一步：请求主机发送一个关闭连接的请求给另一方。

第二步：另一方收到关闭连接的请求后，发送一个接受请求的确认数据包，并关闭它的 socket 连接。

第三步：请求主机收到确认数据包后，发送一个确认数据包，告知另一方其发送的确认已收到，请求主机关闭它的 socket 连接。

11.1.7　客户机/服务器模型

网络中的实际应用大多都可以归纳为客户机/服务器模型（Client/Server 模型、C/S 模型），其中客户机是指请求服务的一方，服务器是指提供某种服务的一方。有些应用程序，请求服务的同时也提供一定的服务，如果拆开来看，这种程序也是基于客户机/服务器模型。

客户机/服务器模型既可以使用 TCP 协议也可以使用 UDP 协议，或者两者混合使用，可根据具体需要而定。在客户机/服务器模型中，通常服务器端的 IP 地址和端口号是固定的，客户端程序连接到服务器 IP 和端口。通常客户端的程序设计相对要简单一些。而服务器端由于要考虑多个客户端同时请求服务的问题，设计上相对复杂一些。

11.2 套接字编程

20 世纪 80 年代初，由美国政府提供资金，委托加利福尼亚大学伯克利分校在 UNIX 操作系统下实现针对 TCP/IP 协议的应用程序编程接口，他们的工作成果就是 socket（套接字）。套接字首先被应用于 Berkeley Software Distribution(BSD) UNIX 系统中，后来随着 UNIX 操作系统的广泛应用，socket 套接字成为最流行最通用的网络通信应用程序的开发接口。现在不论是 Windows 还是 Linux 都使用 socket 来开发网络应用程序。通常 Linux 下的网络编程就是指套接字编程，本节将对套接字的使用作一个较为全面的介绍。

我爱自由 5ifreedom.com

11.2.1 套接字地址结构

结构 struct sockaddr 定义了一种通用的套接字地址，它在 linux/socket.h 中的定义代码如下：

```
struct sockaddr {
        unsigned short    sa_family;    /* 地址类型，AF_xxx */
        char              sa_data[14];  /* 14 字节的协议地址 */
};
```

其中，成员 sa_family 表示套接字的协议族类型，对应于 TCP/IP 协议该值为 AF_INET；成员 sa_data 存储具体的协议地址。sa_data 之所以被定义成 14 个字节，因为有的协议族使用较长的地址格式。一般在编程中并不对该结构体进行操作，而是使用另一个与它等价的数据结构：sockaddr_in。

每种协议族都有自己的协议地址格式，TCP/IP 协议族的地址格式为结构体 struct sockaddr_in，它在 netinet/in.h 头文件中定义，格式如下：

```
struct sockaddr_in {
        unsigned short        sin_family;    /* 地址类型 */
        unsigned short int    sin_port;      /* 端口号 */
        struct in_addr        sin_addr;      /* IP 地址 */
        unsigned char     sin_zero[8];       /* 填充字节，一般赋值为 0 */
};
```

其中，成员 sin_family 表示地址类型，对于使用 TCP/IP 协议进行的网络编程，该值只能是 AF_INET。sin_port 是端口号，sin_addr 用来存储 32 位的 IP 地址，数组 sin_zero 为填充字段，一般赋值为 0。

struct in_addr 的定义如下：

```
struct in_addr {
      unsigned long    s_addr;
};
```

结构体 sockaddr 的长度为 16 字节，结构体 sockaddr_in 的长度也为 16 字节。通常在编写基于 TCP/IP 协议的网络程序时，使用结构体 sockaddr_in 来设置地址，然后通过强制类型转换成 sockaddr 类型。

以下是设置地址信息的示例代码：

```
struct sockaddr_in sock;
sock.sin_family = AF_INET;
sock.sin_port = htons(80);   /* 设置端口号位 80 */
sock.sin_addr.s_addr = inet_addr("202.205.3.195");  /* 设置地址 */
memset(sock.sin_zero,0,sizeof(sock.sin_zero));  /* 将数组 sin_zero 清 0 */
```

函数 htons 和 inet_addr 将在后面介绍。

memset 函数的原型为：

```
memset(void *s,int c,size_t n);
```

它将 *s* 指向的内存区域的前 n 个字节赋值由参数 *c* 指定的值。

11.2.2 创建套接字

socket 函数用来创建一个套接字，在 Shell 下输入“man socket”可获得该函数的原型：

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

参数 domain 用于指定创建套接字所使用的协议族，它们在头文件 linux/socket.h 中定义。常用的协议族如下。

- AF_UNIX：创建只在本机内进行通信的套接字。
- AF_INET：使用 IPv4 TCP/IP 协议。
- AF_INET6：使用 IPv6 TCP/IP 协议。

参数 type 指定套接字的类型，可以取如下值。

- SOCK_STREAM：创建 TCP 流套接字。

我爱自由 5ifreedom.com

- SOCK_DGRAM：创建 UDP 数据报套接字。
- SOCK_RAW：创建原始套接字。

参数 protocol 通常设置为 0，表示通过参数 domain 指定的协议族和参数 type 指定的套接字类型来确定使用的协议。当创建原始套接字时，系统无法惟一地确定协议，此时就需要使用该参数指定所使用的协议。

执行成功返回一个新创建的套接字；若有错误发生则返回-1，错误代码存入 errno 中。详细的错误代码说明请参考 man 手册，或者参考第 6 章的例 6-2，该程序解释了如何从错误代码中获取错误描述信息。

下面的代码创建了一个 TCP 套接字：

```
int sock_fd;
sock_fd = socket(AF_INET, SOCK_STREAM,0);
if (sock_fd < 0) {
    perror("socket");
    exit(1);
}
```

创建 UDP 协议的套接字为：

```
sock_fd = socket(AF_INET, SOCK_DGRAM,0);
```

11.2.3 建立连接

函数 connect 用来在一个指定的套接字上创建一个连接，在 Shell 下输入“man connect”可获得该函数的原型：

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);
```

参数 sockfd 是一个由函数 socket 创建的套接字。如果该套接字的类型是 SOCK_STREAM，则 connect 函数用于向服务器发出连接请求，服务器的 IP 地址和端口号由参数 serv_addr 指定。如果套接字的类型是 SOCK_DGRAM，则 connect 函数并不建立真正的连接，它只是告诉内核与该套接字进行通信的目的地址（由第二个参数指定），只有该目的地址发来的数据才会被该 socket 接收。对于 SOCK_DGRAM 类型的套接字，调用 connect 函数的好处是不必在每次发送和接收数据时都指定目的地址。

通常一个面向连接的套接字（如 TCP 套接字）只能调用一次 connect 函数。而对于无连接的套接字（如 UDP 套接字）则可以多次调用 connect 函数以改变与目的地址的绑定。将参数 serv_addr 中的 sa_family 设置为 AF_UNSPEC 可以取消绑定。

参数 serv_addr 是一个地址结构，其定义见 11.2.1 小节。

addrlen 为参数 serv_addr 的长度。

执行成功返回 0，有错误发生则返回-1，错误代码存入 errno 中，详细的错误代码说明请参考 man 手册。

该函数的常见用法如下：

```
struct sockaddr_in      serv_addr;
memset(&serv_addr, 0, sizeof (struct sockaddr_in));  // 将 serv_addr 的各个字段清 0
serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons(80); // htons 是字节顺序转换函数，将在 11.2.10 小节介绍
// inet_aton 函数将一个字符串转换成一个网络地址，并把该网络地址赋给第二个参数
// inet 族函数将在 11.2.10 小节介绍
if (inet_aton("172.17.242.131", &serv_addr.sin_addr) < 0) {
    perror("inet_aton");
    exit(1);
}
// 使用 sock_fd 套接字连接到由 serv_addr 指定的目的地址上，假定 sock_fd 已定义
```

我爱自由 5ifreedom.com

```
    if (connect(sock_fd, (struct sockaddr *)&serv_addr, sizeof (struct sockaddr_in)) < 0) {
        perror("connect");
        exit(1);
    }
```

注意：serv_addr 强制类型转换为 struct sockaddr 类型。

11.2.4 绑定套接字

函数 bind 用来将一个套接字和某个端口绑定在一起，在 Shell 下输入“man 2 bind”可获取该函数的原型：

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen);
```

socket 函数只是创建了一个套接字，这个套接字将工作在哪个端口上，程序并没有指定。前面提到，在客户机/服务器模型中，服务器端的 IP 地址和端口号一般是固定的，因此在服务器端的程序中，使用 bind 函数将一个套接字和某个端口绑定在一起。该函数一般只有服务器端的程序调用。

参数 my_addr 指定了 sockfd 将绑定到的本地地址，可以将参数 my_addr 的 sin_addr 设置为 INADDR_ANY 而不是某个确定的 IP 地址就可以绑定到任何网络接口。对于只有一个 IP 地址的计算机，INADDR_ANY 对应的就是它的 IP 地址；对于多宿主主机（拥有多块网卡），INADDR_ANY 表示本服务器程序将处理来自所有网络接口上相应端口的连接请求。

函数执行成功返回 0，当有错误发生时则返回-1，错误代码存入 errno 中，详细的错误代码说明请参考 man 手册。

该函数的常见用法如下：

```
struct sockaddr_in      serv_addr;
memset(&serv_addr, 0, sizeof (struct sockaddr_in));
serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons(80);
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);

if (bind(sock_fd, (struct sockaddr *)&serv_addr, sizeof (struct sockaddr_in)) < 0) {
    perror("bind");
    exit(1);
}
```

11.2.5 在套接字上监听

函数 listen 把套接字转化为被动监听，在 Shell 下输入“man listen”可获得该函数原型：

```
#include <sys/socket.h>
int listen(int s, int backlog);
```

由函数 socket 创建的套接字是主动套接字，这种套接字可以用来主动请求连接到某个服务器（通过函数 connect）。但是作为服务器端的程序，通常在某个端口上监听等待来自客户端的连接请求。在服务器端，一般是先调用函数 socket 创建一个主动套接字，然后调用函数 bind 将该套接字绑定到某个端口上，接着再调用函数 listen 将该套接字转化为监听套接字，等待来自于客户端的连接请求。

一般多个客户端连接到一个服务器，服务器向这些客户端提供某种服务。服务器端设置一个连接队列，记录已经建立的连接，参数 backlog 指定了该连接队列的最大长度。如果连接队列已经达到最大，之后的连接请求将被服务器拒绝。

执行成功返回 0，当有错误发生时则返回-1，错误代码存入 errno 中，详细的错误代码说明请参考 man 手册。

注意：函数 listen 只是将套接字设置为侦听模式以等待连接请求，它并不能接收连接请求，真正接收客户端连接请求的是后面即将介绍的函数 accept()。

我爱自由 5ifreedom.com

该函数的常见用法如下：

```
#define  LISTEN_NUM    12   // 定义连接请求队列长度
...
if (listen(sock_fd, LISTEN_NUM) < 0) {
    perror("listen");
    exit(1);
}
```

11.2.6 接受连接

函数 accept 用来接受一个连接请求，在 Shell 下输入“man 2 accept”可获得该函数的原型：

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
```

参数 *s* 是由函数 socket 创建，经函数 bind 绑定到本地某一端口上，然后通过函数 listen 转化而来的监听套接字。

- 参数 addr 用来保存发起连接请求的主机的地址和端口。
- 参数 addrlen 是 addr 所指向的结构体的大小。

执行成功返回一个新的代表客户端的套接字，出错则返回-1，错误代码存入 errno 中，详细的错误代码说明请参考 man 手册。

只能对面向连接的套接字使用 accept 函数。accept 执行成功时，将创建一个新的套接字，并且为这个新的套接字分配一个套接字描述符，并返回这个新的套接字描述符。这个新的套接字描述符与打开文件时返回的文件描述符类似，进程可以利用这个新的套接字描述符与客户端交换数据，参数 *s* 所指定的套接字继续等待客户端的连接请求。

如果参数 *s* 所指定的套接字被设置为阻塞方式（Linux 下的默认方式），且连接请求队列为空，则accept()将被阻塞直到有连接请求到达为止；如果参数 *s* 所指定的套接字被设置为非阻塞方式（见 6.3.4 小节对 fcntl 的讲解），则如果队列为空，accept 将立即返回-1，errno 被设置为 EAGAIN。

套接字为阻塞方式下该函数的常见用法：

```
int  client_fd;
int  client_len;
struct sockaddr_in      client_addr;
...
client_len = sizeof (struct sockaddr_in);
client_fd = accept(sock_fd, (struct sockaddr *)&client_addr, &client_len);
if (conn_fd < 0) {
     perror("accept");
     exit(1);
}
```

11.2.7 TCP 套接字的数据传输

1. 发送数据

函数 send 用来在 TCP 套接字上发送数据，在 Shell 下输入“man 2 send”可获取函数原型：

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t send(int s, const void *msg, size_t len, int flags);
```

函数 send 只能对处于连接状态的套接字使用。参数 *s* 为已建立好连接的套接字描述符，即 accept 函数的返回值。参数 msg 指向存放待发送数据的缓冲区，参数 len 为待发送数据的长度。

参数 flags 为控制选项，一般设置为 0 或取以下值。

- MSG_OOB：在指定的套接字上发送带外数据（out-of-band data），该类型的套接字必须支持带外数据（如 SOCK_STREAM）。

我爱自由 5ifreedom.com

- MSG_DONTROUTE：通过最直接的路径发送数据，而忽略下层协议的路由设置。

如果要发送的数据太长而不能发送时，将出现错误，errno 设置为 EMSGSIZE；如果要发送的数据长度大于该套接字的缓冲区剩余空间大小时，send()一般会被阻塞，如果该套接字被设置为非阻塞方式，则此时立即返回-1 并将 errno 设为 EAGAIN。

执行成功返回实际发送数据的字节数，出错则返回-1，错误代码存入 errno 中，详细的错误代码说明请参考 man 手册。

注意：执行成功只是说明数据写入套接字的缓冲区中，并不表示数据已经成功地通过网络发送到目的地。

套接字为阻塞方式下，该函数的常见用法：

```
#define   BUFFERSIZE        1500
char      send_buf[BUFFERSIZE];
...
if (send(conn_fd, send_buf, len, 0) < 0) {  // len 为待发送数据的长度
          perror("send");
          exit(1);
}
```

2. 接收数据

函数 recv 用来在 TCP 套接字上接收数据，在 Shell 下输入“man recv”可获得该函数的原型：

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t recv(int s, void *buf, size_t len, int flags);
```

函数 recv 从参数 *s* 所指定的套接字描述符（必须是面向连接的套接字）上接收数据并保存到参数 buf 所指定的缓冲区，参数 len 则为缓冲区长度。

参数 flags 为控制选项，一般设置为 0 或取以下数值。

- MSG_OOB：请求接收带外数据。
- MSG_PEEK：只查看数据而不读出。
- MSG_WAITALL：只在接收缓冲区满时才返回。

如果一个数据包太长以至于缓冲不能完全放下时，剩余部分的数据将可能被丢弃（根据接收数据的套接字类型而定）。如果在指定的套接字上无数据到达时，recv()将被阻塞，如果该套接字被设置为非阻塞方式，则立即返回-1 并将 errno 设为 EAGAIN。函数 recv 接收到数据就返回，并不会等待接收到参数 len 指定长度的数据才返回。

执行成功返回接收到的数据字节数，出错则返回-1，错误代码存入 errno 中，详细的错误代码说明请参考 man 手册。

套接字为阻塞方式下该函数的常见用法：

```
    char      recv_buf[BUFFERSIZE];
    ...
if ( recv(conn_fd, recv_buf, sizeof (recv_buf), 0) < 0 ) {
    perror("recv");
    exit(1);
}
```

11.2.8 UDP 套接字的数据传输

1. 发送数据

函数 sendto 用来在 UDP 套接字上发送数据，在 Shell 下输入“man sendto”可获取其函数原型：

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t  sendto(int s, const void *msg, size_t len, int flags,
const struct sockaddr *to, socklen_t tolen);
```

我爱自由 5ifreedom.com

函数 sendto 的功能与函数 send 类似，但函数 sendto 不需要套接字处于连接状态，所以该函数通常用来发送 UDP 数据。同时因为是无连接的套接字，在使用 sendto 时需要指定数据的目的地址。

参数 msg 指向待发送数据的缓存区，参数 len 指定了待发送数据的长度，参数 flags 是控制选项，含义与 send()一致，参数 to 用于指定目的地址，目的地址的长度由 tolen 指定。

执行成功返回实际发送数据的字节数，出错误则返回-1，错误代码存入 errno 中，详细的错误代码说明请参考 man 手册。

以下为该函数的常见用法：

```
: char .   send_buf[BUFFERSIZE];
: struct sockaddr_in     dest_addr;

: // 设置目的地址
: memset(&dest_addr, 0, sizeof (struct sockaddr_in) );
: dest_addr.sin_family = AF_INET;
: dest_addr.sin_port = htons(DEST_PORT);
: // inet 族函数在 11.2.10 节介绍
: if (inet_aton("172.17.242.131", &dest_addr.sin_addr) < 0) {
    perror("inet_aton");
    exit(1);
}

if ( sendto(sock_fd, send_buf, len, 0, (struct sockaddr *)&dest_addr,
        sizeof (struct sockaddr_in) ) < 0)  {
   perror("sendto");
   exit(1);
}
```

2. 接收数据

函数 recvfrom 用来在 UDP 套接字上接收数据，在 Shell 下输入“man recvfrom”可获得该函数的原型：

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t  recvfrom(int s, void *buf, size_t len, int flags,
      struct sockaddr *from, socklen_t *fromlen);
```

函数 recvfrom 与函数 recv 功能类似，只是函数 recv 只能用于面向连接的套接字，而函数 recvfrom 没有此限制，可以用于从无连接的套接字（如 UDP 套接字）上接收数据。

参数 buf 指向接收缓冲区，参数 len 指定了缓冲区的大小，参数 flags 是控制选项，含义与 recv 一致。如果参数 from 非空，且该套接字不是面向连接的，则函数 recvfrom 返回时，参数 from 中将保存数据的源地址，参数 fromlen 在调用 recvfrom 前为参数 from 的长度，调用 recvfrom 后将保存 from 的实际大小。

执行成功返回实际接收到数据的字节数，出错则返回-1，错误代码存入 errno 中，详细的错误代码说明请参考 man 手册。

套接字为阻塞方式下该函数的常见用法：

```
char  recv_buf[BUFFERSIZE];
struct sockaddr_in src_addr;
int   src_len;

src_len = sizeof (struct sockaddr_in);
if (recvfrom(sock_fd, recv_buf, sizeof (recv_buf), 0,
          (struct sockaddr *)&src_addr, &src_len) < 0) {
    perror("again_recvfrom");
    exit(1);
}
```

我爱自由 5ifreedom.com

11.2.9 关闭套接字

1. 函数 close

函数 close 用来关闭一个套接字描述符，它与关闭文件描述符是类似的，在 Shell 下输入“man close”可获得该函数的原型：

```
#include <unistd.h>
int close(int fd);
```

参数 fd 为一个套接字描述府。该函数关闭一个套接字。

执行成功返回 0，出错则返回-1，错误代码存入 errno 中。

2. 函数 shutdown

函数 shutdown 也用于关闭一个套接字描述符，在 Shell 下输入“man 2 shutdown”可获得该函数的原型：

```
#include <sys/socket.h>
int shutdown(int s, int how);
```

函数 shutdown 的功能与函数 close 类似，但是 shutdown()功能更强大，可以对套接字的关闭进行一些更细致的控制，它允许对套接字进行单向关闭或全部禁止。参数 *s* 为待关闭的套接字描述符，参数 howto 指定了关闭的方式，具体取值如下。

- SHUT_RD：将连接上的读通道关闭，此后进程将不能再接收任何数据，接收缓冲区中还未被读取的数据也将被丢弃，但仍然可以在该套接字上发送数据。
- SHUT_WR：将连接上的写通道关闭，此后进程将不能再发送任何数据，发送缓冲区中还未被发送的数据也将被丢弃，但仍然可以在该套接字上接收数据。
- SHUT_RDWR：读、写通道都将被关闭。

执行成功返回 0，出错则返回-1，错误代码存入 errno 中。

11.2.10 主要系统调用函数

1. 字节顺序和转换函数

不同机器内部对变量的字节存储顺序不同，有的采用大端模式(big-endian)，有的采用小端模式(little-endian)，大端模式是指高字节数据存放在低地址处，低字节数据存放在高地址处；小端模式是指低字节数据存放在内存低地址处，高字节数据存放在内存高地址处，具体区别如图 11-7 所示。0 × 04030201 分别在大小端模式下的存储格式（地址为 0 × 1000）。

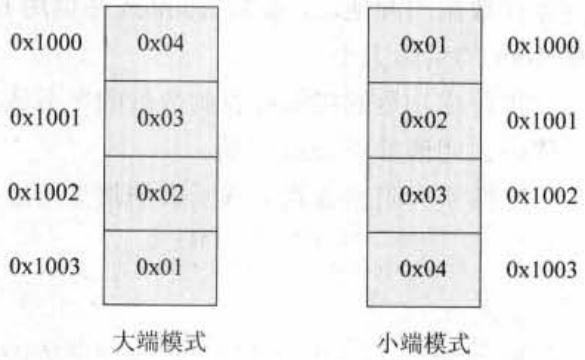


图 11-7 大、小端模式

在网络上传输数据时，由于数据传输的两端可能对应不同的硬件平台，采用的存储字节顺序也可能不一致，因此 TCP/IP 协议规定了在网络上必须采用网络字节顺序（也就是大端模式）。通过对大小端模式存储原理的分析也可以发现，对于 char 型数据，由于其只占一个字节，所以不存在这个问题，这也是我们一般把数据缓冲区定义成 char 型的原因之一。而对于 IP 地址、端口号等非 char 型数据，必须在数据发送到网络上之前将其转换成大端模式，在接收到数据之后再将其转换成符合接收端主机的存储模式。Linux 系统为大小端模式的转换提供了 4 个函数，在 Shell 下输入“man byteorder”可获取它们

我爱自由 5ifreedom.com

函数原型如下：

```
#include <netinet/in.h>
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

htonl 表示 host to network long，用于将主机 unsigned int 型数据转换成网络字节顺序；htons 表示 host to network short，用于将主机 unsigned short 型数据转换成网络字节顺序；ntohl、ntohs 的功能分别与 htonl、htons 相反。

2. inet 系列函数

通常我们习惯于使用字符串形式的网络地址（如“172.17.242.131”），然而在网络上进行数据通信时，需要使用的是二进制形式且为网络字节顺序的 IP 地址，如“172.17.242.131”对应的二进制形式为：0x83f211ac。Linux 系统为网络地址的格式转换提供了一系列函数，在 Shell 下输入“man inet”可获取它们的函数原型如下。

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int inet_aton(const char *cp, struct in_addr *inp);
in_addr_t inet_addr(const char *cp);
in_addr_t inet_network(const char *cp);
char* inet_ntoa(struct in_addr in);
struct in_addr inet_makeaddr(int net, int host);
in_addr_t inet_lnaof(struct in_addr in);
in_addr_t inet_netof(struct in_addr in);
```

- **函数 inet_aton**

inet_aton()将参数 cp 所指向的字符串形式的 IP 地址转换为二进制的网络字节顺序的 IP 地址，转换后的结果存于参数 inp 所指向的空间中。执行成功返回非 0 值，参数无效则返回 0。

- **函数 inet_addr**

该函数的功能与 inet_aton()类似，它将参数 cp 所指向的字符串形式的网络地址转换为网络字节顺序形式的二进制 IP 地址，执行成功时将转换后的结果返回，参数无效返回 INADDR_NONE（一般该值为-1）。

该函数已经过时，推荐使用函数 inet_aton()。因为对有效地址“255.255.255.255”它也返回-1（因为-1 的补码形式为 0xFFFFFFFF），使得用户可能将“255.255.255.255”当成无效的非法地址，而使用 inet_aton()则不存在这种问题。

- 函数 inet_network

inet_network ()将参数 cp 所指向的字符串形式的网络地址转换为主机字节顺序形式的二进制 IP 地址，执行成功返回转换后的结果，参数无效返回-1。

- 函数 inet_ntoa

该函数将值为 in 的网络字节顺序形式的二进制 IP 地址转换成以“.”分隔的字符串形式，执行成功返回结果字符串的指针，参数无效返回 NULL。

- 函数 inet_makeaddr

该函数将把网络号为参数 net，主机号为参数 host 的两个地址组合成一个网络地址，如 net 取 0xac11（172.17.0.0，主机字节顺序形式），host 取 0xf283(0.0.242.131，主机字节顺序形式)，则组合后的地址为 172.17.242.131，并表示为网络字节顺序形式 0x83f211ac。

- 函数 inet_lnaof

该函数从参数 in 中提取出主机地址（参考 11.1.1 小节对地址的介绍），执行成功返回主机字

我爱自由 5ifreedom.com

节顺序形式的主机地址。如 172.17.242.131，属于 B 类地址，则主机号为低 16 位，主机地址为 0.0.242.131，按主机字节顺序输出则为 0xf283。

- 函数 inet_netof

该函数从参数 in 中提取出网络地址，执行成功返回主机字节顺序形式的网络地址。如 172.17.242.131，属于 B 类地址，则高 16 位表示网络号，网络地址为 172.17.0.0，按主机字节顺序输出则为 0xac11，例 11-1 实现 inet 函数族的的使用。

例 11-1　test_address.c

```
// 示例 inet 函数族的使用
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main()
{
    char            buffer[32];
    int             ret = 0;
    int             host = 0;
    int             network = 0;
    unsigned int    address = 0;
    struct in_addr in;

    in.s_addr = 0;

    /*输入一个以"."分隔的字符串形式的 IP 地址*/
    printf("please input your ip address:");
        fgets(buffer, 31, stdin);
    buffer[31] = '\0';

    /*示例使用 inet_aton()函数*/
    if ((ret = inet_aton(buffer, &in)) == 0) {
        printf("inet_aton: \t invalid address\n");
    } else {
        printf("inet_aton:\t0x%x\n", in.s_addr);
    }

    /*示例使用 inet_addr()函数*/
    if ((address = inet_addr(buffer)) == INADDR_NONE) {
        printf("inet_addr: \t invalid address\n");
    } else {
        printf("inet_addr:\t0x%lx\n", address);
    }

    /*示例使用 inet_network()函数*/
    if ((address = inet_network(buffer)) == -1) {
        printf("inet_network: \t invalid address\n");
    } else {
        printf("inet_network:\t0x%lx\n", address);
    }

    /*示例使用 inet_ntoa()函数*/
    if (inet_ntoa(in) == NULL) {
        printf("inet_ntoa: \t invalid address\n");
    } else {
        printf("inet_ntoa:\t%s\n", inet_ntoa(in));
    }

    /*示例使用 inet_lnaof()与 inet_netof()函数*/
    host = inet_lnaof(in);
    network = inet_netof(in);
```

我爱自由 5ifreedom.com

```
    printf("inet_lnaof:\t0x%x\n", host);
    printf("inet_netof:\t0x%x\n", network);

    in = inet_makeaddr(network, host);
    printf("inet_makeaddr:0x%x\n", in.s_addr);

    return 0;
}
```

运行程序，输入 172.17.242.131，执行结果如下：

```
[root@localhost debug]# ./test_address
please input your ip address:172.17.242.131
inet_aton:          0x83f211ac
inet_addr:     0x83f211ac
inet_network: 0xac11f283
inet_ntoa:          172.17.242.131
inet_lnaof:    0xf283
inet_netof:    0xac11
inet_makeaddr:0x83f211ac
```

第二次运行程序，输入 255.255.255.255，执行结果如下：

```
[root@localhost debug]# ./test_address
please input your ip address:255.255.255.255
inet_aton:          0xffffffff
inet_addr:     invalid address
inet_network: invalid address
inet_ntoa:          255.255.255.255
inet_lnaof:    0xff
inet_netof:    0xffffff
inet_makeaddr:0xffffffff
```

从运行结果可以看到，函数 inet_addr()和 inet_network()把地址“255.255.255.255”当成了无效地址。

3. getsockopt()和 setsockopt()

套接字创建以后，就可以利用它来传输数据，但有时可能对套接字的工作方式有特殊的要求，此时就需要修改套接字的属性。系统提供了套接字选项来控制套接字的属性，使用函数 getsockopt 可以获取套接字的属性，使用 setsockopt()可以设置套接字的属性，在 Shell 下输入“man getsockopt”可获取它们的原型：

```
#include <sys/types.h>
#include <sys/socket.h>
int getsockopt(int s, int level, int optname, void *optval, socklen_t *optlen);
int setsockopt(int s, int level, int optname, const void *optval, socklen_t optlen);
```

参数 *s* 为一个套接字，参数 level 是进行套接字选项操作的层次，可以取 SOL_SOCKET（通用套接字）、IPPROTO_IP（IP 层套接字）、IPPROTO_TCP（TCP 层套接字）等值，一般取 SOL_SOCKET 来进行与特定协议不相关的操作。参数 optname 是套接字选项的名称。

对于函数 getsockopt，参数 optval 用来存放获得的套接字选项，参数 optlen 在调用函数前其值为 optval 指向的空间的大小，调用完成后则其值为参数 optval 所保存的结果的实际大小。对于函数 setsockopt，参数 optval 是待设置的套接字选项的值，参数 optlen 是该选项的长度。

这两个函数执行成功时都返回 0，出错都返回-1，错误代码存入 errno 中，详细的错误代码说明请参考 man 手册。

下面介绍通用套接字 SOL_SOCKET 的选项，可以使用命令“man 7 socket”获得更详细的介绍。

- SO_KEEPALIVE

如果没有设置 SO_KEEPALIVE 选项，那么即使 TCP 连接已经很长时间没有数据传输时，系统也不会检测这个连接是否还有效。对于服务器进程，如果某一客户端非正常断开连接，则服务器进程将一直被阻塞等待。因此服务器端程序需要使用这个选项，如果某个客户端一段时间内没

我爱自由 5ifreedom.com

有反应则关闭该连接。

- SO_RCVLOWAT 和 SO_SNDLOWAT

SO_RCVLOWAT 表示接收缓冲区的下限，只有当接收缓冲区中的数据超过了 SO_ RCVLOWAT 才会将数据传送到上层应用程序。SO_SNDLOWAT 表示发送缓冲区的下限，只有当发送缓冲区中数据超过了 SO_SNDLOWAT 才会将数据发送出去。Linux 下这两个值都为 1 且不能更改，也就是说只要有数据就接收或发送。这两个选项只能使用 getsockopt 函数获取，不能用 setsockopt 函数更改。

- SO_RCVTIMEO 和 SO_SNDTIMEO

这两个选项可以设置对套接字读或写的超时时间，具体时间由下面这个结构指定：

```
struct timeval {
        long    tv_sec;         /* 秒数 */
        long    tv_usec;        /* 微秒数 */
};
```

成员 tv_sec 指定秒数，tv_usec 指定微秒数。超时时间为这两个时间的和。在某个套接字连接上，若读或写超时，则认为接收或发送数据失败。

- SO_BINDTODEVICE

将套接字绑定到特定的网络接口如“eth0”，此后只有该网络接口上的数据才会被套接字处理。如果将选项值设置为空字符串或选项长度设为 0 将取消绑定。

- SO_DEBUG

该选项只能对 TCP 套接字使用，设置了该选项后系统将保存 TCP 发送和接收的所有数据的相关信息，以方便调试程序。

- SO_REUSEADDR

Linux 系统中，如果一个 socket 绑定了一个端口，该 socket 正常关闭或程序异常退出后的一段时间内，该端口依然维持原来的绑定状态，其他程序无法绑定该端口，如果设置了该选项则可以避免这个问题。示例代码如下：

```
int option_value = 1;
int length = sizeof(option_value);
setsockopt( sock_fd,SOL_SOCKET,SO_REUSEADDR,&option_value,length );
```

- SO_TYPE

用于获取套接字的类型，如 SOCK_DGRAM、SOCK_STREAM、SOCK_SEQPACKET、SOCK_RDM 等。该选项只能被函数 getsockopt 用来获取套接类型，而不能使用函数 setsockopt 修改套接字的类型。

- SO_ACCEPTCONN

该选项用来检测套接字是否处于监听状态，如果为 0 表示处于非监听状态，如果为 1 表示正在监听，该选项只能被函数 getsockopt 用来获取监听状态信息。

- SO_DONTROUTE

设置该选项表示在发送 IP 数据包时不使用路由表来寻找路由。

- SO_BROADCAST

该选项用来决定套接字是否能够在网络上广播数据。实际应用中要在网络上广播数据必须硬件支持广播（如以太网支持广播）并且使用的是 SOCK_DGRAM 套接字。系统默认不支持广播，如果希望该 SOCK_DGRAM 套接字支持广播，则可以这样来修改设置：

```
int option_value = 1;
setsockopt( sock_fd, SOL_SOCKET, SO_BROADCAST, &option_value, sizeof(int) );
```

- SO_SNDBUF 和 SO_RCVBUF

这两个选项用于设置套接字的发送和接收缓冲区的大小。对于 TCP 类型的套接字，缓冲区太

我爱自由 5ifreedom.com

小会影响 TCP 的流量控制；对于 UDP 类型的套接字，如果套接字的数据缓冲区满则后续数据将被丢弃，实际应用中应根据需要设置一个合适的大小。

- SO_ERROR

获取套接字内部的错误变量 so_error，当套接字上发生了异步错误时，系统将设置套接字的 so_error。异步错误是指错误的发生和错误被发现的时间不一致，通常在目的主机非正常关闭时发生这种错误。该选项只能被函数 getsockopt 用来获取 so_error。

注意：调用完函数 getsockopt 之后 so_error 的值将自动被重新初始化。

4. 多路复用 select()

在客户端/服务器模型中，服务器端需要同时处理多个客户端的连接请求，此时就需要使用多路复用。实现多路复用最简单的方法是采用非阻塞方式套接字，服务器端不断地查询各个套接字的状态，如果有数据到达则读出数据，如果没有数据则查看下一个套接字。这种方法虽然简单，但在轮询过程中浪费了大量的 CPU 时间，效率非常低。

另一种方法是服务器进程并不主动地询问套接字状态，而是向系统登记希望监视的套接字，然后阻塞。当套接字上有事件发生时（如有数据到达），系统通知服务器进程告知哪个套接字上发生了什么事件，服务器进程查询对应套接字并进行处理。在这种工作方式下，套接字上没有事件发生时，服务器进程不会去查询套接字的状态，从而不会浪费 CPU 时间，提高了效率。

使用函数 select 可以实现第二种多路复用，在 Shell 下输入“man select”可获得该函数的原型：

```
#include <sys/select.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
int select( int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,struct timeval *timeout);
```

参数 *n* 是需要监视的文件描述符数，要监视的文件描述符值为 0～n-1。参数 readfds 指定需要监视的可读文件描述符集合，当这个集合中的一个描述符上有数据到达时，系统将通知调用 select 函数的程序。参数 writefds 指定需要监视的可写文件描述符集合，当这个集合中的某个描述符可以发送数据时，程序将收到通知。参数 exceptfds 指定需要监视的异常文件描述符集合，当该集合中的一个描述符发生异常时，程序将收到通知。参数 timeout 指定了阻塞的时间，如果在这段时间内监视的文件描述符上都没有事件发生，则函数 select()将返回 0。

struct timeval 的定义如下：

```
struct timeval {
    long    tv_sec;         /* seconds */
    long    tv_usec;        /* microseconds */
};
```

成员 tv_sec 指定秒数，tv_usec 指定微秒数。如果将 timeout 设为 NULL，则函数 select()将一直被阻塞，直到某个文件描述符上发生了事件。如果将 timeout 设为 0，则此时相当于非阻塞方式，函数 select()查询完文件描述符集合的状态后立即返回。如果将 timeout 设成某一时间值，在这个时间内如果没有事件发生，函数 select()将返回；如果在这段时间内有事件发生，程序将收到通知。

注意：这里的文件描述符既可以是普通文件的描述符，也可以是套接字描述符。

系统为文件描述符集合提供了一系列的宏以方便操作：

```
FD_CLR(int fd, fd_set *set);          // 将文件描述符 fd 从文件描述符集合 set 中删除
FD_ISSET(int fd, fd_set *set);    // 测试 fd 是否在 set 中
FD_SET(int fd, fd_set *set);          // 在文件描述符集合 set 中增加文件描述符 fd
FD_ZERO(fd_set *set);                 // 将文件描述符集合 set 清空
```

如果 select()设定的要监视的文件描述符集合中有描述符发生了事件，则 select 将返回发生事件的文件描述符的个数，例 11-2 是函数 select 的使用示例。

我爱自由 5ifreedom.com

例 11-2　test_select.c

```
// 示例函数 select()的使用
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
#include <time.h>

void display_time(const char *string)
{
    int       seconds;

    seconds = time((time_t*)NULL);
    printf("%s, %d\n", string, seconds);
}

int main(void)
{
    fd_set          readfds;
    struct timeval timeout;
    int             ret;

    /*监视文件描述符 0 是否有数据输入，文件描述符 0 表示标准输入，即键盘输入*/
    FD_ZERO(&readfds);  // 开始使用一个描述符集合前一般要将其清空
    FD_SET(0, &readfds);

    /*设置阻塞时间为 10 秒*/
    timeout.tv_sec = 10;
    timeout.tv_usec = 0;

    while (1) {
        display_time("before select");
        ret = select(1, &readfds, NULL, NULL, &timeout);
        display_time("after select");

        switch (ret) {
            case 0:
                printf("No data in ten seconds.\n");
                exit(0);
                break;
            case -1:
                perror("select");
                exit(1);
                break;
            default:
                getchar();      // 将数据读入，否则标准输入上将一直为读就绪
                printf("Data is available now.\n");
        }
    }

    return 0;
}
```

程序说明。

程序先初始化一个文件描述符集合 readfds，然后将文件描述符 0 增加到这个文件描述符集合中，在调用 select 函数前将阻塞时间设置为 10 秒。函数 time()用来获取从公元 1970 年 1 月 1 日 0 时 0 分 0 秒算起到现在所经过的秒数。执行结果如下：

```
[root@localhost debug]# ./test_select
before select, 1175065257
(输入 a 并按下 enter 键)
after select, 1175065262
Data is available now.
before select, 1175065262
after select, 1175065267
No data within ten seconds.
```

我爱自由 5ifreedom.com

执行程序时，等待几秒钟后按下键盘任意键和 Enter 键，或者只按下 Enter 键。从执行过程及结果可以看出，我们在第 1175065262s 按下了 Enter 键，select()立即返回并打印出提示信息。而后再次进入循环，在第 1175065262s 重新设置 select()监视键盘动作，可以发现，这次虽然没有任意键按下，但是 select()在第 1175065267s 返回了，也就是说这次阻塞的时间只有 5s 而不是预想中的 10s。这是因为 Linux 系统对 select()的实现中会修改参数 timeout 为剩余时间，我们第一次在阻塞了 5s 后才按下了键盘，剩余时间为 5s，所以第二次就只阻塞 5s 了，如果“case(0):”分支里面不调用 exit(0)，则从第三次开始将不阻塞（因为 timeout 为 0）而出现打印信息的刷屏。所以在使用函数 select()对文件描述符集进行监视时，一定要注意这个问题，如果是在循环中调用 select()，则参数 timeout 的初始化必须放在循环内部。

11.3 一个面向连接的 Client/Server 实例

基于 Client/Server 模型的面向连接的网络程序，其基本流程如图 11-8 所示。

下面使用 TCP 套接字来开发一个模拟用户远程登录的程序。

1．服务器端程序的设计

- 服务器端的并发性

本程序采用多进程的方式来实现服务器对多个客户端连接请求的响应。主程序创建套接字后将套接字绑定在 4507 端口，也可以绑定到其他端口。然后使套接字处于监听状态，调用 accept 函数等待来自客户端的连接请求。每接收一个新的客户端连接请求，服务器端进程就创建一个子进程，在子进程中处理该连接请求，服务器端进程继续等待来自其他客户端的连接请求。

- 数据格式

由于 TCP 是一种基于流的数据传输方式，数据没有固定的格式。因此需要在应用程序中定义一定的数据格式，本程序以回车符（字符‘\n’）作为一次数据的结束标志。

- 用户信息

本程序将用户信息保存在一个全局数组中。服务器端接收到来自客户端的登录用户名后，在该全局数组中查询是否存在该用户名。若不存在，则回应字符‘n’+结束标志（回车符‘\n’）；若存在，则回应字符‘y’+结束标志，然后等待客户端的密码。若密码不匹配，则回应字符‘n’+结束标志；若密码匹配，则回应字符‘y’+结束标志，然后发送一个欢迎登录的字符串给客户端，服务器端程序代码如例 11-3 所示。

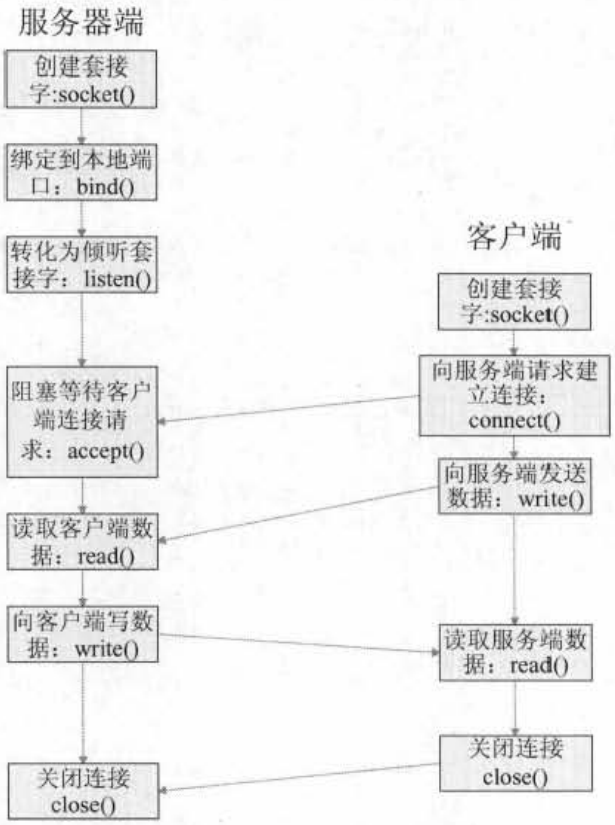


图 11-8　面向连接的 Client/Server 模型框图

我爱自由 5ifreedom.com

2. 客户端程序的设计

客户端的应用程序相对于服务器端要简单，客户端主程序创建套接字后调用 connect()连接到服务器端的 4507 端口，使用从 connect()返回的连接套接字与服务器端进行通信，交换数据。

客户端程序代码如例 11-6 所示。

例 11-3 服务器端程序 my_server.c

```
// Client/Server 模型的服务器端
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>
#include "my_recv.h"    // 自定义的头文件

#define  SERV_PORT              4507 // 服务器端的端口
#define  LISTENQ                12        // 连接请求队列的最大长度

#define  INVALID_USERINFO       'n'       // 用户信息无效
#define  VALID_USERINFO         'y'       // 用户信息有效

#define  USERNAME               0         // 接收到的是用户名
#define  PASSWORD               1         // 接收到的是密码
struct userinfo {       // 保存用户名和密码的结构体
    char username[32];
    char password[32];
};
struct userinfo users[ ] = {
    {"linux", "unix"},
    {"4507", "4508"},
    {"clh", "clh"},
    {"xl", "xl"},
    {" "," "}       // 以只含一个空格的字符串作为数组的结束标志
};

// 查找用户名是否存在，存在返回该用户名的下标,不存在则返回-1，出错返回-2
int find_name(const char *name)
{
    int i;

    if (name == NULL) {
        printf("in find_name, NULL pointer");
        return -2;
    }
    for (i=0; users[i].username[0] != ' ';i++) {
        if (strcmp(users[i].username, name) == 0) {
            return i;
        }
    }

    return -1;
}

// 发送数据
void send_data(int conn_fd, const char *string)
{
        if (send(conn_fd, string, strlen(string), 0) < 0) {
            my_err("send", __LINE__);  // my_err 函数在 my_recv.h 中声明
        }
}

int main()
```


我爱自由 5ifreedom.com

```
{
    int                     sock_fd, conn_fd;
    int                     optval;
    int                     flag_recv = USERNAME; // 标识接收到的是用户名还是密码
    int                     ret;
    int                     name_num;
    pid_t                   pid;
    socklen_t               cli_len;
    struct sockaddr_in cli_addr, serv_addr;
    char                recv_buf[128];

    // 创建一个TCP套接字
    sock_fd = socket(AF_INET, SOCK_STREAM,0);
    if (sock_fd < 0) {
        my_err("socket", __LINE__);
    }

    // 设置该套接字使之可以重新绑定端口
    optval = 1;
    if (setsockopt(sock_fd, SOL_SOCKET, SO_REUSEADDR,
         (void *)&optval, sizeof(int)) < 0) {
        my_err("setsockopt", __LINE__);
    }

    // 初始化服务器端地址结构
    memset(&serv_addr, 0, sizeof (struct sockaddr_in));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(SERV_PORT);
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);

    // 将套接字绑定到本地端口
    if (bind(sock_fd, (struct sockaddr *)&serv_addr,
        sizeof (struct sockaddr_in)) < 0) {
        my_err("bind", __LINE__);
    }

    // 将套接字转化为监听套接字
    if (listen(sock_fd, LISTENQ) < 0) {
        my_err("listen", __LINE__);
    }

    cli_len = sizeof (struct sockaddr_in);
    while (1) {
        // 通过accept接收客户端的连接请求，并返回连接套接字用于收发数据
        conn_fd = accept(sock_fd, (struct sockaddr *)&cli_addr, &cli_len);
        if (conn_fd < 0) {
            my_err("accept", __LINE__);
        }

        printf("accept a new client, ip:%s\n", inet_ntoa(cli_addr.sin_addr));
        // 创建一个子进程处理刚刚接收的连接请求
        if ( (pid = fork()) == 0 ) {      // 子进程
            while(1) {
                if ((ret = recv(conn_fd, recv_buf, sizeof (recv_buf), 0)) < 0) {
                    perror("recv");
                    exit(1);
                }
                recv_buf[ret-1] = '\0'; // 将数据结束标志'\n'替换成字符串结束标志

                if (flag_recv == USERNAME) {      // 接收到的是用户名
                    name_num = find_name(recv_buf);
                    switch (name_num) {
                        case -1:
                            send_data(conn_fd, "n\n");
                            break;
                        case -2:
                            exit(1);
```

我爱自由 5ifreedom.com

```
                                break;
                            default:
                                send_data(conn_fd, "y\n");
                                flag_recv = PASSWORD;
                                break;
                        }
                    } else if (flag_recv == PASSWORD) { // 接收到的是密码
                        if (strcmp(users[name_num].password, recv_buf) == 0) {
                            send_data(conn_fd, "y\n");
                            send_data(conn_fd, "Welcome login my tcp server\n");
                            printf("%s login\n", users[name_num].username);
                            break; // 跳出 while 循环
                        } else
                            send_data(conn_fd, "n\n");
                    }
                }
                close(sock_fd);
                close(conn_fd);
                exit(0);  // 结束子进程
            } else {   // 父进程关闭刚刚接收的连接请求，执行 accept 等待其他连接请求
                close(conn_fd);
            }
        }

    return 0;
}
```

程序说明。

本程序的结构与图 11-8 所示的服务器端一致，程序首先创建一个 TCP 套接字并将其绑定到本地端口上，然后将其转化为监听套接字，再调用函数 accept 接收来自客户端的连接请求，收到请求后创建一个子进程来单独处理该连接请求。在子进程中，验证客户端的登录用户名和密码，若正确则发送欢迎登录信息，实现代码如例 11-4 所示。

例 11-4　my_recv.c

```
#define  MY_RECV_C

#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include "my_recv.h"

/*自定义的错误处理函数*/
void my_err(const char * err_string, int line)
{
    fprintf(stderr, "line:%d  ", line);
    perror(err_string);
    exit(1);
}

/*
* 函数名: my_recv
* 描 述 : 从套接字上读取一次数据(以'\n'为结束标志)
* 参 数 : conn_fd ——从该连接套接字上接收数据
*         data_buf——读取到的数据保存在此缓冲中
*         len——data_buf 所指向的空间长度
* 返回值: 出错返回-1,  服务器端已关闭连接则返回 0,  成功返回读取的字节数
*/
int my_recv(int conn_fd, char *data_buf, int len)
{
    static  char  recv_buf[BUFSIZE]; // 自定义缓冲区, BUFSIZE 定义在 my_recv.h 中
    static  char  *pread;            // 指向下一次读取数据的位置
    static  int   len_remain = 0;    // 自定义缓冲区中剩余字节数
    int           i;
```

我爱自由 5ifreedom.com

```
    // 如果自定义缓冲区中没有数据，则从套接字读取数据
    if (len_remain <= 0) {
        if ((len_remain =recv(conn_fd, recv_buf, sizeof (recv_buf), 0)) < 0) {
            my_err("recv", __LINE__);
        } else if (len_remain == 0) {     // 目的计算机端的 socket 连接关闭
            return 0;
        }
        pread = recv_buf;  // 重新初始化 pread 指针
    }

    // 从自定义缓冲区中读取一次数据
    for (i=0; *pread != '\n'; i++) {
        if (i > len) {      // 防止指针越界
            return -1;
        }
        data_buf[i] = *pread++;
        len_remain--;
    }
    // 去除结束标志
    len_remain--;
    pread++;

    return i;      // 读取成功
}
```

程序说明。

这是一个封装了函数 recv 的自定义读取数据的函数，实际上是将套接字缓冲区中的数据拷贝到自定义缓冲区，然后再按格式（以‘\n’为结束标志）读取出数据。

例 11-5 my_recv.h

```
#ifndef  __MY_RECV_H
#define  __MY_RECV_H
    #define  BUFSIZE  1024
    void my_err(const char * err_string, int line);
    int  my_recv(int conn_fd, char *data_buf, int len);
#endif
```

头文件中，__MY_RECV_H 的作用是防止重复包含该头文件。

例 11-6 客户端程序 my_client.c

```
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include "my_recv.h"

#define  INVALID_USERINFO        'n'  // 用户信息无效
#define  VALID_USERINFO          'y'  // 用户信息有效

/*获取用户输入存入到 buf，buf 的长度为 len，用户输入数据以'\n'为结束标志*/
int get_userinfo(char *buf, int len)
{
    int i;
    int c;

    if (buf == NULL) {
        return -1;
    }

    i = 0;
    while ( ((c = getchar()) != '\n') && (c != EOF) && (i < len-2) ) {
```

我爱自由 5ifreedom.com

```
        buf[i++] = c;
    }
    buf[i++] = '\n';
    buf[i++] = '\0';

    return 0;
}

// 输入用户名,然后通过 fd 发送出去
void input_userinfo(int conn_fd, const char *string)
{
    char     input_buf[32];
    char     recv_buf[BUFSIZE];
    int          flag_userinfo;

    // 输入用户信息直到正确为止
    do {
        printf("%s:", string);
        if (get_userinfo(input_buf, 32) < 0) {
            printf("error return from get_userinfo\n");
            exit(1);
        }

        if (send(conn_fd, input_buf, strlen(input_buf), 0) < 0) {
                my_err("send", __LINE__);
        }

        // 从连接套接字上读取一次数据
        if (my_recv(conn_fd, recv_buf, sizeof (recv_buf)) < 0) {
            printf("data is too long\n");
            exit(1);
        }

        if (recv_buf[0] == VALID_USERINFO) {
            flag_userinfo = VALID_USERINFO;
        } else {
            printf("%s error,input again,", string);
            flag_userinfo= INVALID_USERINFO;
        }
    } while(flag_userinfo == INVALID_USERINFO);
}

int main(int argc, char **argv)
{
    int                 i;
    int                 ret;
    int                 conn_fd;
    int                 serv_port;
    struct sockaddr_in serv_addr;
    char                recv_buf[BUFSIZE];

    // 检查参数个数
    if (argc != 5) {
        printf("Usage: [-p] [serv_port] [-a] [serv_address]\n");
        exit(1);
    }

    // 初始化服务器端地址结构
    memset(&serv_addr, 0, sizeof (struct sockaddr_in));
    serv_addr.sin_family = AF_INET;
    // 从命令行获取服务器端的端口与地址
    for (i=1; i<argc; i++) {
        if (strcmp("-p", argv[i]) == 0) {
            serv_port = atoi(argv[i+1]);
            if (serv_port < 0 || serv_port > 65535) {
                printf("invalid serv_addr.sin_port\n");
                exit(1);
```

我爱自由 5ifreedom.com

```
            } else {
                serv_addr.sin_port = htons(serv_port);
            }
            continue;
        }

        if (strcmp("-a", argv[i]) == 0) {
            if (inet_aton(argv[i+1], &serv_addr.sin_addr) == 0) {
                printf("invalid server ip address\n");
                exit(1);
            }
            continue;
        }
    }
    // 检测是否少输入了某项参数
    if (serv_addr.sin_port == 0 || serv_addr.sin_addr.s_addr == 0) {
        printf("Usage: [-p] [serv_addr.sin_port] [-a][serv_address]\n");
        exit(1);
    }

    // 创建一个 TCP 套接字
    conn_fd = socket(AF_INET, SOCK_STREAM,0);
    if (conn_fd < 0) {
        my_err("socket", __LINE__);
    }

    // 向服务器端发送连接请求
    if (connect(conn_fd, (struct sockaddr *)&serv_addr, sizeof (struct sockaddr)) < 0) {
        my_err("connect", __LINE__);
    }

    // 输入用户名和密码
    input_userinfo(conn_fd, "username");
    input_userinfo(conn_fd, "password");

    // 读取欢迎信息并打印出来
    if ((ret = my_recv(conn_fd, recv_buf, sizeof (recv_buf))) < 0) {
        printf("data is too long\n");
        exit(1);
    }
    for (i=0; i<ret; i++) {
        printf("%c", recv_buf[i]);
    }
    printf("\n");

    close(conn_fd);
    return 0;
}
```

程序说明。

本程序的结构与图 11-8 所示的客户端一致，程序首先创建一个 TCP 套接字，然后调用函数 connect 请求与服务器端连接。建立连接后，通过连接套接字首先发送用户名，然后等待服务器的确认，若用户名存在，则发送密码，若密码正确，则等待欢迎信息。

本例中共有 4 个文件：my_recv.h、my_recv.c、my_server.c、my_client.c。my_server.c 和 my_client.c 都调用了在 my_recv.h 头文件中声明，在 my_recv.c 中实现的函数。

分别编译服务器端和客户端程序：

```
[root@localhost chapter9]#gcc -o my_server.c my_server.c my_recv.c
[root@localhost chapter9]#gcc -o my_client.c my_client.c my_recv.c
```

执行本程序时，先在某一终端上运行服务器端程序，然后在几个终端运行客户端，以验证多进程方式的服务器对来自多个客户端的连接请求的处理，在客户端输入如下数据：

我爱自由 5ifreedom.com

```
[root@localhost chapter9]# ./my_client -a 127.0.0.1 -p 4507
username:clh
password:
```

此时，该客户端已经同服务器端建立了连接，再在一终端运行客户端并输入用户名：

```
[root@localhost chapter9]# ./my_client -p 4507 -a 172.17.242.131
username:xl
password:
```

此时，服务器端已经同时与两个客户端建立了连接并进行了数据的传输，两个客户端分别输入密码后可接收到来自服务器端的欢迎信息：

```
[root@localhost chapter9]# ./my_client -a 127.0.0.1 -p 4507
username:clh
password:clh
Welcom login my tcp server
```

服务器端的打印信息如下：

```
[root@localhost chapter9]# ./my_server
accept a new client, ip:127.0.0.1
accept a new client, ip:172.17.242.131
xl login
clh login
```

可以尝试输入错误的用户名或密码，观察结果。也可以先使用 ifconfig 命令获取某台计算机的 IP 地址，然后在该计算机上运行服务器端程序，之后在其他计算机上运行客户端程序，观察输出结果。

11.4 编写安全的代码

由于 TCP/IP 协议族在设计时并没有考虑到网络安全问题，这使得基于 TCP/IP 协议族构建的 Internet 在安全性方面非常脆弱。利用各种协议漏洞进行的攻击时有发生。另一方面由于编程中的疏忽和安全意识淡薄，导致许多程序存在着安全漏洞。而这些存在安全漏洞的程序往往运行在连入 Internet 的主机上。黑客利用这些安全漏洞可以轻松地获取系统的控制权。在获取和管理员一样的权限后，他们可以为所欲为，进行大肆破坏。更有甚者，许多自动扫描主机，自动进行漏洞探测，自动进行攻击的工具被不断开发出来。病毒、蠕虫、木马、恶意软件、广告软件或利用漏洞或与其他攻击技术相互配合对互联网的安全构成了极大的威胁。

造成安全问题的原因有很多，比如用户或网络管理员缺乏安全意识、TCP/IP 协议的固有缺陷、程序中的安全漏洞。在所有安全漏洞中，威胁最大的是缓冲区溢出漏洞。黑客、病毒、蠕虫利用这类漏洞可以在几秒钟获得系统的控制权或者利用这类漏洞使主机或网站瘫痪。而造成缓冲区漏洞的一个主要原因是程序员在编写代码时缺乏安全意识，在程序中留下了许多安全漏洞。因此，作为一个高素质的软件开发人员有必要了解一些如何编写安全代码方面的知识。

11.4.1 网络攻击

目前构建网络一般都采用 TCP/IP 协议，它是一个工业标准的协议族，但该协议族在设计之初，对安全性考虑不多，协议中存在很多漏洞。要保证网络安全、可靠，有必要了解网络攻击的一般方法。以下列出了几种常见的网络攻击方法。

- 扫描和探测

利用 IP 或端口扫描工具探测主机是否在线或开放某种服务。比如某主机的 21 号端口开放，那么基本可以确定该主机开放 FTP 服务。如果能进一步获取 FTP 服务所使用的软件的版本，而对

我爱自由 5ifreedom.com

应的版本如果存在某个漏洞，那么就可以发起攻击。探测是指使用工具自动探测主机中的漏洞，以便为下一步的攻击做准备。要说明的是，扫描和探测也可以用来检测自己网络或主机系统的安全性，发现漏洞及时下载补丁，这样可以避免遭受攻击。

- 监听

通过截获网络上的数据包可以获得被攻击系统的密码信息或其他有价值的信息。有些以太网以广播方式传递数据，这很容易被监听。

- 拒绝服务

拒绝服务目的在于瘫痪目标系统。攻击者利用其控制的许多主机同时向某个网站发送大量的无用数据包，消耗该网站主机的处理能力，从而使正常用户无法访问该网站。近年来流行的僵尸网络就是利用自动攻击工具控制数以百计、数以千计的电脑，并在某个时刻同时向某个网站发送大量的连接请求使其不能及时处理。如果攻击持续下去，该网站将无法为正常用户服务。由数千台、数万台或更多的傀儡主机构成的僵尸网络可以称为网络原子弹，如果该网站没有较好的针对此类攻击的防护措施，它可以使一般的网站在瞬间崩溃。

- 恶意软件

近几年来，以广告软件、间谍软件为代表的恶意软件对网络造成了极大的破坏。它们要么不断弹出广告窗口要么悄悄地潜伏到系统中伺机获取主机中的机密信息，如网络游戏或银行的账号和密码。

此外，还有许多传统的攻击工具和方法，如计算机病毒、特洛伊木马、后门等。

11.4.2 缓冲区溢出

几十年来，缓冲区溢出一直是严重的安全威胁。其中最著名的例子是：1988 年 11 月出现的 Morris 蠕虫，它利用缓冲区溢出漏洞进行攻击，获取了大量主机的控制权，据统计它使当时 Internet 上 10%的主机崩溃。缓冲区溢出之所以成为远程攻击的主要手段，其原因在于缓冲区溢出漏洞给予了攻击者所想要的一切：殖入并且执行攻击代码。它包括多种类型，如堆溢出、栈溢出、静态存储区溢出、数组越界等。这个问题产生的原因在于程序中缺少对输入数据的检查。

1. 什么是缓冲区溢出

缓冲区溢出是指向缓冲区内填充的数据超过了缓冲区的容量，溢出的数据覆盖在原来合法数据上。产生缓冲区溢出问题的根本原因是：C 及 C++语言本质上是不安全的，没有机制用来检查引用数组和指针时的边界，导致越界访问。对边界的检查由程序员来完成，而这一任务往往会被程序员忽视或遗忘。

在 Linux 系统中，一个进程在内存中的数据主要分为 3 个部分：文本段、数据段和堆栈段，其中文本段存放可执行代码和只读数据，通常该区域的属性为只读；数据段主要用于存放全局变量、静态变量；临时变量、函数参数等存放在栈上；而由 malloc 函数动态分配的内存称为堆。

通常在程序要从外部接收输入数据时，系统会分配一块内存用于存放输入数据，这块内存通常称为缓冲区。如果输入的数据作为函数的参数，那么它将被保存到栈上，如果内存是在程序执行时动态分配的，则输入的数据会被保存到堆上。当输入数据超过缓冲区所能容纳的最大容量时，而程序恰好没有对输入数据的长度作出检查，则缓冲区不能容纳的那些数据会存储到缓冲区之后的区域中，覆盖了原来的数据。

一般情况下，覆盖其他数据区的数据是没有意义的，最多造成应用程序产生异常。但是，如果输入的多余数据是经过精心设计的可执行的恶意代码。黑客在缓冲区溢出后让程序跳转到恶意代码处执行，黑客就获取了程序的控制权。如果该程序是以 root 身份运行的，黑客就具有了 root 权限。黑客可

我爱自由 5ifreedom.com

以通过执行恶意代码创建一个管理员权限的帐号或者植入一个后门以方便黑客对该系统的访问。

2. 如何防止缓冲区溢出

防止缓冲区溢出的一个重要方法是对程序中定义的缓冲区（如一个数组 char s[32]）作严格的边界检查。如果有超过缓冲区大小的内容被写入，程序应该报错，不允许将多余的数据写入缓冲区中。

应避免使用 strcpy 等存在溢出漏洞的函数，而使用 strncpy()或 memcpy()等作为代替。

编写程序时要保证不出错是非常困难的，但可以使用一些工具对代码进行检查以发现程序中的缓冲区溢出漏洞。这种侦错技术只能用来减少缓冲区溢出漏洞，并不能完全地消除它的存在。

对于已经发现的缓冲区漏洞应该及时打上补丁以消除该漏洞。

11.4.3 输入检查

对输入的参数进行检查是一个良好的编程习惯，它可以有效地避免可能存在的溢出漏洞。编程中应该注意对输入数据的类型、输入数据的长度进行合法性检查。特别是指针参数，必须检查其是否为空指针，它所指向的空间是否大于缓冲区的空间。例 11-7 演示了对输入数据进行检查。

例 11-7　my_strcpy.c

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

char *my_strcpy(char *strDest, const char *strSrc)
{
     char *p_return = strDest;

     // 检查参数指针是否为空
     if (strDest == NULL || strSrc == NULL) {
          fprintf(stderr, "NULL POINT!");
          return NULL;
     }

     while ( (*strDest++ = *strSrc++) != '\0')
          ;
     return p_return;
}

int main()
{
     char      string1[32];
     char      string2[32];
     int       c;
     int       i = 0;

     printf("please input your string:");
     // 对输入字符串的长度进行检查
     while (  ( (c = getchar()) != '\n' ) && (c != EOF) && (i < 31)  )  {
          string2[i] = c;
          i++;
     }
     string2[i] = '\0';

     // 对返回值也进行合法性检查
     if (my_strcpy(string1, string2) == NULL) {
          fprintf(stderr, "return from my_strcpy");
          exit(1);
     }

     printf("string1:%s\n", string1);
     printf("string2:%s\n", string2);
     return 0;
}
```

程序说明。

本程序实现了一个字符串复制函数，在该函数内部，首先对指针型参数进行合法性检查。在 main

我爱自由 5ifreedom.com

函数中，对于输入的字符，判断其是否为文件结束标识符，以及是否遇到了回车键，同时还对输入字符的个数进行了检查。主函数中对函数 my_strcpy 的返回值也进行了检查。

本程序执行结果如下：

```
[root@localhost debug]# ./my_strcpy
please input your string:sdlfjlksd
string1:sdlfjlksd
string2:sdlfjlksd
```

如果对输入字符的个数不检查（即在程序中去掉“&&(i < 31)”），重新编译和执行程序，并输入超过 31 个字符，观察运行结果：

```
[root@localhost debug]# ./my_strcpy
please input your string:sdlkfjsldkajflsakdjflksadjflksadfjklsajflksdjlks
Segmentation fault
```

或者将 my_strcpy 函数中对指针是否为 NULL 的判断去掉，同时在 main 函数中定义一个 char *p，调用 my_strcpy(p, string2)，则同样也将出现内存段错误。

11.5 编程实践：编程实现端口扫描器实例

常用的端口扫描技术有很多种，如 TCP connect 扫描 、TCP SYN 扫描、TCP FIN 扫描等，本节利用 TCP connect 扫描来实现自己的端口扫描程序 my_scaner。从 11.2.3 小节已经了解到，当调用 connect 函数返回 0 时表示连接成功，说明目标计算机的该端口是打开的；若返回–1 且错误代码为 ECONNREFUSED，则说明目标计算机的端口未打开；若返回–1 且错误代码为其他，则是其他类型的错误。

本程序在扫描端口时使用了多线程技术，把要扫描的所有端口平均分配给一些线程，每一个线程负责扫描一部分端口。主线程负责任务分配、启动各个子线程和等待各子线程结束，代码如例 11-8 所示。

例 11-8 my_scaner.c

```
// 端口扫描程序，只支持扫描 TCP 端口
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <pthread.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

// 定义一个端口区间信息
typedef struct _port_segment {
    struct in_addr          dest_ip;        // 目标 IP
    unsigned short int min_port; // 起始端口
    unsigned short int max_port; // 最大端口
} port_segment;

/*自定义的错误处理函数*/
void my_err(const char * err_string, int line)
{
    fprintf(stderr, "line:%d  ", line);
    perror(err_string);
    exit(1);
}

/*
* 描  述：扫描某一 IP 地址上的某一个端口的函数
```

我爱自由 5ifreedom.com

```
* 返回值:  -1    出错
*          0     目标端口未打开
*                1     目标端口已打开
*/
int do_scan(struct sockaddr_in serv_addr)
{
    int       conn_fd;
    int       ret;

    // 创建一个 TCP 套接字
    conn_fd = socket(AF_INET, SOCK_STREAM,0);
    if (conn_fd < 0) {
        my_err("socket", __LINE__);
    }

    // 向服务器端发送连接请求
    if ( (ret = connect(conn_fd, (struct sockaddr *)&serv_addr,
                sizeof (struct sockaddr))) < 0 ) {
        if (errno == ECONNREFUSED) {      // 目标端口未打开
            close(conn_fd);
            return 0;
        } else {  // 其他错误
            close(conn_fd);
            return -1;
        }
    } else if (ret == 0){
        printf("port %d found in %s\n", ntohs(serv_addr.sin_port),
                        inet_ntoa(serv_addr.sin_addr));
        close(conn_fd);
        return 1;
    }

    return -1;     // 实际执行不到这里，只是为了消除编译程序时产生的警告
}

// 执行扫描的线程，扫描某一区间的端口
void * scaner(void *arg)
{
    unsigned short int i;
    struct sockaddr_in serv_addr;
    port_segment       portinfo;   // 端口信息

    // 读取端口区间信息
    memcpy(&portinfo, arg, sizeof(struct_port_segment));

    // 初始化服务器端地址结构
    memset(&serv_addr, 0, sizeof (struct sockaddr_in));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = portinfo.dest_ip.s_addr;

    for (i=portinfo.min_port; i<=portinfo.max_port; i++) {
        serv_addr.sin_port = htons(i);
        if (do_scan(serv_addr) < 0) {
            continue; // 出错则退出进程
        }
    }
    return NULL;
}

/* 命令行参数: -m 最大端口,  -a 目标主机的 IP 地址,  -n 最大线程数 */
int main(int argc, char **argv)
{
    pthread_t*    thread;              // 指向所有的线程 ID
    int           max_port;       // 最大端口号
    int           thread_num;          // 最大线程数
    int           seg_len;             // 端口区间长度
    structin_addr dest_ip;        // 目标主机 IP
    int           i;

    // 检查参数个数
    if (argc != 7) {
```

我爱自由 5ifreedom.com

```
        printf("Usage: [-m] [max_port] [-a] [serv_address] [-n] [thread_number]\n");
        exit(1);
    }

    // 解析命令行参数
    for (i=1; i<argc; i++) {
        if (strcmp("-m", argv[i]) == 0) {
            max_port = atoi(argv[i+1]);   // 将字符串转化为对应的整数
            if (max_port < 0 || max_port > 65535) {
                printf("Usage:invalid max dest port\n");
                exit(1);
            }
            continue;
        }

        if (strcmp("-a", argv[i]) == 0) {
            if (inet_aton(argv[i+1], &dest_ip) == 0) {
                printf("Usage:invalid dest ip address\n");
                exit(1);
            }
            continue;
        }

        if (strcmp("-n", argv[i]) == 0) {
            thread_num = atoi(argv[i+1]);
            if (thread_num <= 0) {
                printf("Usage:invalid thread_number\n");
                exit(1);
            }
            continue;
        }
    }
    // 如果输入的最大端口号小于线程数，则将线程数设为最大端口号
    if (max_port < thread_num) {
        thread_num = max_port;
    }

    seg_len = max_port / thread_num;
    if ( (max_port%thread_num) != 0 ) {
        thread_num += 1;
    }
    // 分配存储所有线程 ID 的内存空间
    thread = (pthread_t*)malloc(thread_num*sizeof(pthread_t));

    // 创建线程，根据最大端口号和线程数分配每个线程扫描的端口区间
    for (i=0; i<thread_num; i++) {
        port_segment  portinfo;
        portinfo.dest_ip = dest_ip;
        portinfo.min_port = i*seg_len + 1;
        if (i == thread_num-1) {
            portinfo.max_port = max_port;
        } else {
            portinfo.max_port = portinfo.min_port + seg_len - 1;
        }
        // 创建线程
        if (pthread_create(&thread[i], NULL, scaner, (void *)&portinfo) != 0) {
            my_err("pthread_create", __LINE__);
        }
        // 主线程等待子线程结束
        pthread_join(thread[i],NULL);
    }

    return 0;
}
```

程序说明。

程序首先从命令行中解析参数：目标 IP、最大端口号（程序将扫描 1 到最大端口号的所有端

我爱自由 5ifreedom.com

口)、最大线程数，然后将端口号根据线程数均分并创建多个线程。每个线程首先根据结构 portinfo 中的信息填充地址结构 serv_addr，然后多次调用实际执行扫描的函数 do_scan，在该函数内先创建一个 TCP 套接字，然后调用 connect 函数测试目标端口，若目标端口打开，则打印出提示信息。

编译并执行程序，结果如下：

```
[root@localhost chapter9]# gcc -o my_scanner my_scanner.c -lpthread
[root@localhost chapter9]# ./my_scanner -m 4600 -a 127.0.0.1 -n 100
port 21 found in 127.0.0.1
port 22 found in 127.0.0.1
port 53 found in 127.0.0.1
port 111 found in 127.0.0.1
port 1118 found in 127.0.0.1
```

作为验证，可以使用 netstat –a 命令显示本机打开的 TCP 端口。

注意：在运行本端口扫描程序前，最好先关闭防火墙。因为有的防火墙禁止以这种方式对主机进行探测。

当向目的主机未打开的 UDP 端口发送 UDP 数据时，目标设备会返回一个“ICMP 端口不可到达”的错误报文，且函数 sendto 将出错返回并将错误代码置为 ECONNREFUSED。利用这一点可以进行 UDP 端口扫描，感兴趣的读者可以自己动手编写一个 UDP 端口扫描程序。

11.6 进一步学习建议

本章介绍了套接字编程的基本方法，作为 Linux 网络编程的入门。要编写出功能强大、高效率的网络应用程序，必须对 TCP/IP 协议非常熟悉。如果要深入学习 TCP/IP 协议可以参考《TCP/IP 详解》或者《用 TCP/IP 进行网际互联》。

大部分网络应用都可以归结为 Client/Server 模型，在编写基于 Client/Server 模型的程序时，服务器端的性能尤为重要，其效率、并发性以及安全性是难点，在这方面可以参考 Richard W.Stevens 的《UNIX 网络编程》一书。

本章简略地介绍了网络攻击和代码安全，如果要进一步学习这方面的理论和技术可以参考《编写安全的代码》一书。该书由微软公司的两位安全专家撰写，它从各种程序代码中可能存在的安全漏洞和安全问题入手，通过大量的实例和代码，详细地分析说明了编写安全的应用程序的方方面面。

11.7 习题

1. 简述 TCP/IP 模型中各层的主要功能，各有哪些主要协议。
2. 简述 TCP/IP 模型中应用层数据从主机 A 发送到主机 B 的整个过程。
3. 利用 setsockopt()使套接字支持广播，并编写两个程序，其中一个发送广播数据，另一个接收广播数据。
4. 编写一个 UDP 类型的 client/server 实例。
5. 将 11.3 节的 client/server 实例改为多线程的并发方式，注意线程同步。
6. 编写一个 UDP 端口扫描程序。
7. 如何在编程中防止缓冲区溢出？
8. 对于端口扫描程序 my_scanner，输入命令“my_scanner -m 65535 -a 127.0.0.1 -n 100”，观察结果，会发现程序将无限循环下去，请分析原因。提示：请注意 scaner 函数中循环变量 i 的类型。

我爱自由 5ifreedom.com

第 12 章 GTK+图形界面编程

本章将介绍 Linux 下的图形界面编程，重点介绍基于 C 语言的具有面向对象特征的 GTK+图形界面编程。主要介绍 GTK+图形界面应用程序的框架、基本原理、常用控件的使用。

本章重点：

- GTK+程序的基本结构。
- 事件和消息处理。
- 常用控件的使用。

本章难点：

- 理解 GTK+应用程序的基本原理。
- 熟悉常用控件的基本用法。

12.1 Linux 下的图形界面编程

12.1.1 Qt 和 GTK+

虽然 Linux 下的大多数开发是基于字符界面的，但在 Linux 环境下也可以开发出美观大方的图形界面。经过多年的发展，目前已经存在多种用于在 Linux 下开发图形界面程序的开发包，其中较为常用的是 Qt 和 GTK+。

Qt 是一个跨平台的图形用户界面开发库，它不仅支持 Linux 操作系统，还支持所有类型的 UNIX 以及 Windows 操作系统。Qt 良好的封装机制使它模块化程度非常高，可重用性很强，Qt 提供了丰富的 API 供开发人员使用。使用 Qt 开发的图形用户界面程序具有良好的稳定性和健壮性。桌面环境 KDE(K Desktop EnVironment 即 K 桌面环境)就是使用 Qt 作为其底层库开发出来的。

由于 Qt 使用 C++面向对象编程语言作为其开发语言，而许多在 Linux 下从事开发的程序员更喜欢或更习惯于用 C 语言。GTK+使用 C 语言作为开发语言。它基于 LGPL 授权，因此 GTK+是开放源代码而且完全免费的。GTK+简单易用，执行效率高。基于这些原因，GTK+拥有为数众多的拥护者。Linux 的桌面环境 GNOME 就是建立在 GTK+基础上。

12.1.2 GTK+简介

图 12-1GTK+在几种相关的开发库中的位置。

图 12-1 中每层除了与其上下相邻的两层有联系外，似乎与其他层没有关系。实际上，任何上层都可以调用位于它下面的各层提供的函数。例如，GTK+不仅可以调用 GDK 函数，也可以调用 glib 和 C 库函数。

下面按层作简单的介绍，具体说明如表 12-1 所示。

我爱自由 5ifreedom.com

表 12-1　各层的具体含义

层	具体描述
C	有两类 C 库函数可供调用，一类是标准 C 的库函数，如 printf、scanf；另一类是 Linux 的系统调用，如 open、read、write、fork
glib	glib 是 GDK、GTK+、GNOME 应用程序常用的库。它包含内存分配、字符串操作、日期和时间、定时器等库函数，也包括链表、队列、树等数据结构相关的工具函数
X	它是控制图形显示的底层函数库，包括所有的窗口显示函数、响应鼠标和键盘操作的函数
GDK	GDK（GIMP 绘图包）是为了简化程序员使用 X 函数库而开发的。X 库是其低层函数库，GDK 对其进行了包装，从而使程序员的开发效率大为提高
GTK+	GTK+就是 GIMP 工具包，它把 GDK 提供的函数组织成对象，使用 C 语言模拟出面向对象的特征，这使得用它开发出来的图形界面程序更为简单和高效。GTK+的一个重要组成部分是 widget（控件，也称为小部件），按钮、文本编辑框、标签等都是 widget
GNOME	GNOME 库是对 GTK+的扩展，GNOME 桌面环境用来控制整个桌面。GNOME 使用 GNOME 对象和函数与桌面小部件交互，基本小部件由 GTK+处理。GNOME 为了方便程序员还增加了一些专门的小部件
Application	Application 即应用程序，它完成窗口的初始化，创建并显示窗口，进入消息循环，等待用户使用鼠标或键盘进行操作

简单地说，GTK+就是用 C 语言编写的用于开发图形界面程序的函数库。GTK+来源于 GIMP(GNU Image Minipulation Program 即 GNU 图像处理程序)。GTK+在 GDK(GIMP Drawing Kit 即 GIMP 绘图包)基础上创建，对它进行封装。GTK+简单易用，它设计良好，灵活而富有扩展性。它是自由软件，这意味着它不仅开放源代码，而且还可以免费使用。由于它使用 C 语言作为其开发语言，而 C 语言是跨平台的，因此 GTK+几乎可以在任何操作系统上使用。

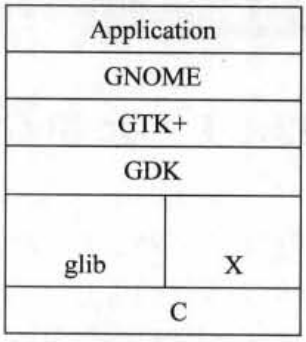


图 12-1　GTK+在几种相关的开发库中的位置

在安装 Federo Core 或者 Red Hat Linux 系列操作系统时，如果选择了安装应用程序开发包，那么操作系统安装完毕后，GTK+开发包就已经安装好了。如果没有安装，请从网络上(http://www.gtk.org)免费下载一份 GTK 源代码并安装到系统上，也可以插入 Linux 安装光盘在系统提示下进行安装。由于安装过程非常简单，这里就不再赘述了。

12.2 一个简单的例子

先来看一个简单的 GTK+图形界面程序的例子，了解这类程序的一般框架。这个程序创建了一个窗口，并在窗口中放置了一个按钮，实现代码如例 12-1 所示。

例 12-1　gtkwin.c

```
#include<gtk/gtk.h>

/*定义回调函数 hello，单击按钮时系统自动调用*/
void hello(GtkWidget *widget,gpointer *data)
{
        g_print("button clicked and data = %s\n",(char *)data);
}

/*定义回调函数 destroy，关闭窗口时系统自动调用*/
void destroy(GtkWidget *widget,gpointer *data)
```

我爱自由 5ifreedom.com

```
{
        gtk_main_quit();
}

int main(int argc,char **argv)
{
        /*定义指向控件的指针*/
        GtkWidget *window;
        GtkWidget *button;

        /*初始化图形显示环境*/
        gtk_init(&argc,&argv);

        /*创建窗口，并设置当关闭窗口时，要执行的回调函数*/
        window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
        g_signal_connect(GTK_OBJECT(window),"destroy",
                GTK_SIGNAL_FUNC(destroy),NULL);
        /*设置窗口的属性*/
        gtk_container_border_width(GTK_CONTAINER(window),20);

        /*创建按钮，并设置当单击按钮时，要执行的回调函数*/
        button = gtk_button_new_with_label("Hello World");
        g_signal_connect(GTK_OBJECT(button),"clicked",
                GTK_SIGNAL_FUNC(hello),"I am from button");

        /*将按钮加入到窗口中*/
        gtk_container_add(GTK_CONTAINER(window),button);
        /*显示按钮和窗口*/
        gtk_widget_show(button);
        gtk_widget_show(window);

        /*进入消息处理循环*/
        gtk_main();

        return 0;
}
```

编译并运行：

```
[root@mci tmp]# gcc -o gtkwin gtkwin.c `pkg-config --libs --cflags gtk+-2.0`
[root@mci tmp]# ./gtkwin
```

注意：编译命令中的字符串“pkg-config --libs –cflags gtk+-2.0”两边是反引号（在键盘上位于数字字符1的左边）。

运行程序，显示如图 12-2 所示的界面：

程序说明。

（1）所有的 GTK+程序中都必须包含头文件 gtk/gtk.h，它声明了所有 GTK+编程中要使用的常量、数据结构和函数。

图 12-2 运行界面

（2）所有 GTK+程序开始都要调用函数 gtk_init(gint *argc,gchar *** argv)。该函数定位和打开图形显示，并对颜色、信号等进行初始化。在命令行输入的参数由该函数传递给 GTK+，该函数读取并获得与它有关的命令行参数。

（3）函数 gtk_window_new(GTK_WINDOW_TOPLEVEL)用于创建顶级窗口，GTK+程序的主窗口被称为顶级窗口。虽然一个程序可以创建多个顶级窗口，但通常只创建一个。

（4）图形界面下，用户的任何一个操作（如单击鼠标左键，按下键盘上的某个键）都称为发生了一个事件，GTK+都有相应的消息信号产生，如果程序中定义了处理该消息信号的函数，在事件发生后，消息信号处理函数会自动调用。这样的消息信号处理函数也称为回调函数，因为这种函数虽然是在程序里定义，但程序中并没有显式调用而是由系统在事件发生后自动调用。

我爱自由 5ifreedom.com

hello()和 destroy()就是两个处理消息的回调函数。destroy 函数中调用了 GTK+函数 gtk_main_quit，它使程序退出 gtk_main()并完成一些清理工作。g_signal_connect 函数用于在控件和消息处理函数间建立关联，该函数的第一个参数为产生消息的控件，第二个参数是消息名，第三个参数是消息发生后要调用的函数名，第四个是传递给消息处理函数的参数，可以为空值（即 NULL）。关于消息和回调函数的详细内容请参考下一节。

（5）gtk_container_border_width 函数用于设置窗口边框的宽度，这是一个设置窗口属性的函数。

（6）gtk_button_new_with_label 函数创建一个带文本标签的按钮，它完成内存分配，并把所分配到的内存的首地址赋给 GtkWidget 类型的指针。

（7）函数 gtk_container_add 通知 GTK+将按钮加入到主窗口中，函数 gtk_widget_show 用于显示控件。

（8）gtk_main()使 GTK+进入消息处理循环。每个 GTK+应用程序都有一个 gtk_main 函数，该函数使程序进入休眠状态。当有事件发生，如果程序中有相应的处理函数，gtk_main()就调用相应的消息处理函数。

（9）为了方便编译，可以在源程序所在目录下编写一个 Makefile 文件：

```
CC=gcc
program=gtkwin
#PATH += /usr/include/gtk2.0
LDLIBS=`pkg-config --libs gtk+-2.0`
CFLAGS= -Wall -g `pkg-config --cflags gtk+-2.0`

$(program):$(program).o
          $(CC) $(LDLIBS) $(program).o -o $(program)

$(program).o:$(program).c
          $(CC) $(CFLAGS) -c $(program).c

clean:
          -rm -f $(program)
          -rm -f *.o
```

此时编译程序可以简化为：

```
[root@mci tmp]# make
```

要删除编译所产生的中间文件和可执行文件，可以执行命令：

```
[root@mci tmp]# make clean
```

如果要编译其他 GTK+程序，把 Makefile 文件中的 gtkwin 改为相应源程序的文件名即可。Makefile 的编写和 make 命令的使用请参考第 5 章。

例 12-1 程序中用到了一些 GTK+预定义的函数和数据类型，表 12-2 和表 12-3 对其作一个简单的介绍。

表 12-2　　GTK+预定义的函数

前　缀	含　义
G	glib 定义的数据结构
g	glib 声明的数据类型
g_	glib 定义的函数
gtk_	GTK+定义的函数
Gtk	GTK+库的对象或数据结构
GTK	GTK+定义的宏或者常量

我爱自由 5ifreedom.com

表 12-3 GTK+预定义的数据类型

GTK+的数据类型	C 语言数据类型
gchar	char
gint	int
glong	long
gboolean	char
gfloat	float
gdouble	double
guchar	unsigned char
guint	unsigned int
gulong	unsigned long
gpointer	void *
gint8	在任何平台上都是 8 位的整型
gint16	在任何平台上都是 16 位的整型
gint32	在任何平台上都是 32 位的整型
guint8	在任何平台上都是 8 位的无符号整型
guint16	在任何平台上都是 16 位的无符号整型
guint32	在任何平台上都是 32 位的无符号整型

12.3 消息和回调函数

图形用户界面的程序是事件驱动的程序。程序进入 gtk_main 函数后，等待事件的发生，一旦发生某个事件，相应的信号将产生。如果程序中定义了相应的消息处理函数，系统会自动进行调用。消息处理函数（或称回调函数）的原型是：

```
void callback_func(GtkWidget *widget,gpointer func_data);
```

参数 widget 指向要接收消息的控件，参数 func_data 指向消息产生时传递给该函数的数据。

函数 g_signal_connect 在控件和消息处理函数间建立关联，该函数的原型是：

```
gulong g_signal_connect(GtkObject *object,gchar *name
                        Gcallback callback_func,gpointer func_data);
```

各参数说明如下。

- object：指向产生消息的控件。
- name：消息或事件的名称。
- callback_func：事件发生后要执行的回调函数。
- func_data：传递给回调函数的数据，与 callback_func()的第二个参数相同。

该函数的返回值用于区分一个控件的一个事件对应的多个处理函数。一个控件上可以发生多个事件，比如单击一个按钮，双击一个按钮。对于一个控件上的每个事件可以有 0 个、1 个或多个处理函数。该事件发生时，将按声明的顺序逐个调用这些函数。对应于某个事件，如果控件没有定义处理函数，那么事件发生时将没有响应，系统忽略此事件。

还有一个与 g_signal_connect()类似的函数：

```
gint g_signal_connect_swapped(GtkObject *object,gchar *name
                              GCallback callback_func,GtkObject *slot_object);
```

我爱自由 5ifreedom.com

它的第四个参数指向一个 GTK+控件的指针。它与 g_signal_connect()的区别在于相应的回调函数只有一个参数：

```
void callback_func(GtkObject *object);
```

通常 object 指向一个控件。

可以在上一节的例子中"gtk_container_add(GTK_CONTAINER(window),button);"之前加上一条语句：

```
g_signal_connect(GTK_OBJECT(button),"clicked",
                 GTK_SIGNAL_FUNC(gtk_widget_destroy),GTK_OBJECT(window));
```

则对应于 button 按钮的"clicked"事件有两个处理函数，一个是程序中定义的回调函数 hello，还有一个是 GTK+预定义的 gtk_widget_destroy()。gtk_widget_destroy()的作用与程序中的 destroy()相同。编译运行程序，如果单击按钮，系统先调用 hello 函数(因为它先与 button 控件建立关联)在命令行上打印出一行消息，然后调用 gtk_widget_destroy()退出程序。

如果要删除控件和消息处理函数的关联，可以调用 g_signal_disconnect()，该函数的原型是：

```
void g_signal__handler _disconnect(GtkObject *object ,gulong id);
```

参数说明如下。

- object：要删除消息处理函数的控件。
- id：g_signal_connect()或 g_signal_connect_swapped()函数的返回值。

下面这个函数可以删除某控件的所有消息处理函数：

```
void g_signal_handlers_destroy(GtkObject *object);
```

12.4 GTK+的面向对象机制

对于那些没有接触过面向对象语言的读者来说，本节的内容可能比较难以理解。不过没有关系，这并不会影响后面内容的掌握。介绍本节的内容只是为了简单地介绍一下 GTK+中是如何模拟面向对象机制的。

面向对象编程语言（如 C++、Java）把数据和对数据的操作封装在一起构成类，由类来产生对象，由对象来构建程序。类中对数据的操作由函数来完成，这种函数被称为成员函数或方法。面向对象语言通过继承、重载、多态等机制大大增强软件的可重用性和可维护性。C 语言虽然不是面向对象语言，但 GTK+以及建立在其上的 GNOME 库却使 C 语言模拟出了一些典型的面向对象机制，如封装、继承和多态。为了较好的理解 GTK+程序，了解 GTK+中的面向对象机制也是很有必要的。

对象的一个主要特性是将数据和对数据的操作封装在一起，受保护的私有数据只能通过成员函数才能访问和修改。GTK+使用 C 语言的结构体来模拟对象，虽然有些缺陷但基本模拟出了对象的基本特征。

有了对象作为基础，通过在对象中加入新的数据和对这些数据进行操作的函数，就实现了继承。被继承的类（类相当于一种自定义数据类型，由类来定义对象）称为父类或基类，由基础类派生出来的类称为子类或派生类。子类继承了父类的数据和对这些数据进行操作的成员函数，并加入了新的数据和成员函数，实现了对原有父类的重用和扩展，从而实现了可重用性和可扩展性。

GTK+中有一个类，它是所有其他类的父类，这个类是 GtkObject。GTK+中最常用的按钮控件也是一个类，它继承自 GtkObject。它与 GtkObject 的继承关系是：

```
GtkObject ->GtkWidget ->GtkContainer ->GtkBin ->GtkButton
```

使用 C 语言如何模拟继承呢？对象（类）是由结构体模拟的，每一个子类所在的结构体都包含了父类的结构体，子类结构体的第一个成员是其父类结构体，示例代码如下：

我爱自由 5ifreedom.com

```
struct GtkObject {
    . . .
};
struct GtkWidget {
    GtkObject object;
    . . .
};
struct GtkContainer {
    GtkWidget widget;
    . . .
};
struct GtkBin {
    GtkContainer container;
    . . .
};
struct GtkButton {
    GtkBin bin;
    . . .
};
```

从上述代码可以看到，每个子类都包含了其父类的所有数据，并且父类的数据位于子类结构体的开始。对于一个 GtkButton 类型的 button 控件变量（它其实是一个指向 GtkButton 结构体的指针），通过宏 GTKBIN(button)就得到了其父类（GTK+预定义的宏 GTKBIN 其实是进行了强制类型转换，把一个 GtkButton 类型的指针强制转化为 GtkBin 类型的指针）。例 12-1 中的 GTK_OBJECT(button)就是进行了这样的转换。

为了便于理解，我们写一个测试程序，如例 12-2 所示。

例 12-2　test.c

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

#define FATHER(child) (struct Father *)(child)

void print1(int i)
{
        printf("this is father and i = %d\n",i);
}

void print2(int i)
{
        printf("this is child  and i = %d\n",i);
}

struct Father {
        int  a;
        void (*pointer1_to_function)(int);
    };
    struct Child {
        struct Father f;
        int b;
        void (*pointer2_to_function)(int);
    };

void father_member_funtion(struct Father *f,char *string)
{
        printf("\n");
        f->pointer1_to_function(f->a);
        printf("%s\n\n",string);
}

int main()
{
        struct Child *p_child;
        p_child = (struct Child *)malloc(sizeof(struct Child));
```

我爱自由 5ifreedom.com

```
        p_child->f.a = 10;
        p_child->f.pointer1_to_function = print1;
        p_child->b   = 20;
        p_child->pointer2_to_function   = print2;

        p_child->pointer2_to_function(p_child->b);

        struct Father *p_father = FATHER(p_child);
        p_father->pointer1_to_function(p_father->a);

        father_member_funtion(p_father,"hello");

        return 0;
}
```

程序输出：

```
this is child and i = 20
this is father and i = 10

this is father and i = 10
hello
```

程序说明。

（1）结构体 Father 相当于 GTK+中的父类，而结构体 Child 就相当于子类。结构体 Father 有一个成员变量和一个成员函数（实际上是一个指向函数的指针，函数指针的内容请参考第 4 章 4.3.4 指针和函数一节）。结构体 Child 在其头部包含了结构体 Father，并增加了一个成员变量和一个成员函数（也是一个函数指针）。结构体 Father 和 Child 模拟了类，Child 模拟继承了 Father。

（2）程序定义了一个指向结构体 Child 的指针，并对 Child 中的成员进行了初始化。然后调用了结构体 Child 的成员函数 pointer2_to_function。通过宏 FATHER(p_child)将指针 p_child 强制转换为指向 Father 结构体的指针。事实上，p_child 和 p_father 的值是一样的，它们都保存着结构体 Child 的首地址。宏 FATHER(p_child)类似于例 12-1 程序 gtk_container_add(GTK_CONTAINER (window),button)中的 GTK_CONTAINER(window)。我们注意到，p_father 调用了它自己的成员函数 pointer1_to_function。

（3）father_member_funtion 函数是类 Father 成员函数的另一种实现方法，这种方法避免了在结构体 Father 中保存函数指针。

GTK+定义了很多生成对象或对对象进行操作的函数。例如，下面就创建了一个对象：

```
GtkWidget *button;
button = gtk_button_new_with_label("label");
```

所有创建对象的函数在其名称上都有“new”这个词。函数 gtk_button_new_with_label 创建了一个显示文本的按钮。可以当作父类来对待所有子类（其实这就是面向对象语言中的多态），按钮 button 的真正类型是 GtkButton，却也可以作为一个指向 GtkWidget 类型的指针。使用 GtkWidget 指针在编程上有很多好处，因为许多图形界面的操作函数都是在 GtkWidget 对象上进行操作的。

当调用一个函数对一个对象进行相关操作时，该对象的地址作为第一个参数传给函数。例如，显示按钮的函数：

```
gtk_widget_show(button);
```

button 是一个指向 GtkWidget 结构体的指针，它也表示一个按钮控件。面向对象语言中子类可以调用父类的函数，在 GTK+中只要使用一些宏将子类强制转换为父类即可。例如：

```
gtk_container_add(GTK_CONTAINER(window),button);
```

gtk_container_add 函数是类 GtkContainer 的一个函数，而 window 代表一个窗口，它是 GtkWindow 类型的指针。GtkWindow 是 GtkContainer 的子类，也就是 GtkWindow 继承自 GtkContainer，当然它们都是从 GtkObject 派生出来的，GtkObject 是它们的祖先：

```
GtkObject ->GtkWidget ->GtkContainer ->GtkBin ->GtkWindow
```


我爱自由 5ifreedom.com

12.5 排列控件

如果要在窗口中放置多个控件，就要考虑如何编排控件的位置。GTK+提供了两种排列控件的方法：一是使用 box（盒子），二是使用 table（表格）。

12.5.1 使用 box 排列控件

1. 创建和使用 box 容器

box 是一种不可见的 widget 容器，它有水平排列和垂直排列两种。水平排列是控件按放入窗口的顺序水平排列，垂直排列是按控件放入窗口的顺序垂直排列。水平排列 box 容器使用函数 gtk_hbox_new 生成，而垂直排列 box 容器使用函数 gtk_vbox_new 生成。box 容器生成后，使用函数 gtk_box_pack_start 或 gtk_box_pack_end 将控件放入容器中。前者由左向右、从上到下将控件放入 box 容器，而后者相反，由右至左，从下到上将控件放入 box 容器中。

下面是具体的函数定义：

```
Widget* gtk_hbox_new(gint homogeneous, gint spacing);
```

参数含义如下。

homogeneous：控制每个放入 box 的控件是否有同样的高或宽。

spacing：是否在控件之间填充空白。

```
void gtk_box_pack_start( GtkBox *box,
                         GtkWidget *child,
                         gint expand,
                         gint fill,
                         gint padding );
```

参数的含义如下。

- box：要放入控件的 box 容器。
- child：要放入 box 容器的控件。
- expand：是否填满 box 所有额外控件，TRUE 表示是，如果为 FALSE 则该 box 按控件原始大小显示。gtk_hbox_new 函数的参数 homogeneous 值为 TRUE 时，该参数才有效。
- fill：该值如果为 TRUE，控件自行产生外控件；如果为 FALSE，box 在控件周围产生反白区域。只有 expand 为 TRUE，该参数才有效。

这样的解释可能不好理解，来看一个例子程序就清楚了。在举例之前，先介绍一下按钮控件。

2. check 按钮和 radio 按钮

生成一般的按钮有两个函数：gtk _button_new()和 gtk _button_new_with_label()。前者产生一个无标签的按钮，后者生成一个有文本标签的按钮。

在开发中，也常常使用 check 按钮和 radio 按钮。它们都有两种状态，一个是选中，另外一个是未选中。所不同的是，在一组按钮中，radio 按钮只能有一个被选中，其他都处于未选中状态，而 check 按钮没有这个限制。它们都是以双态按钮为基础的。可以使用以下函数生成一个双态按钮，第一个生成无标签按钮，第二个生成有文本标签的按钮。

```
GtkWidget* gtk_toggle_button_new(void);
GtkWidget* gtk_toggle_button_new_with_label(gchar *label);
```

对于双态按钮，经常需要在回调函数中判断按钮的状态是否被选中。方法如下：

```
void toggle_button_callback(GtkWidget *widget, gpointer data)
{
```

我爱自由 5ifreedom.com

```
    if(GTK_TOGGLE_BUTTON(widget)->active)
    {
        //按钮被选择时的处理代码
    }
}
```

可以使用下面这个函数，设置按钮的状态：

```
void gtk_toggle_button_set_state(GtkToggleButton *toggle, gint state)
```

参数的含义如下。

- toggle：要设置状态的按钮。
- state：要设置的状态，值为 TRUE 把按钮设置为未选中状态，FLASE 把按钮设置为选中。

生成 check 按钮的函数为：

```
GtkWidget* gtk_check_button_new(void);
GtkWidget* gtk_check _button_new_with_label(gchar *label);
```

生成 radio 按钮的函数为：

```
GtkWidget* gtk_radio_button_new(GSList *group);
GtkWidget* gtk_radio_button_new_with_label(GSList *group , gchar *label);
```

radio 按钮是成组出现的，因此需要一个参数 group。

例 12-3 按钮控件和 box 容器的使用，程序名为 button_box.c。

```
#include<gtk/gtk.h>

/*按下某个按钮后，在命令行上打印出按钮名和新的状态*/
void click_button(GtkWidget *widget,gpointer *data)
{
    g_print("%s ",(char *)data);
    if(GTK_TOGGLE_BUTTON(widget)->active)
        g_print("state is active\n");
    else
        g_print("state is not active\n");
}

void destroy(GtkWidget *widget,gpointer *data)
{
    gtk_main_quit();
}

int main(int argc,char **argv)
{
    GtkWidget       *window;
    GtkWidget       *box;
    GSList          *group;
  GtkWidget       *check,*radio;

    gtk_init(&argc,&argv);

    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(GTK_OBJECT(window),"destroy",
                     GTK_SIGNAL_FUNC(destroy),NULL);
    gtk_container_border_width(GTK_CONTAINER(window),50);

/*生成一个垂直box容器，并将该容器加入到主窗口中*/
    box = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),box);

    /*以下生成两个check按钮，将它们加入到box容器中，并显示出来*/
    check = gtk_check_button_new_with_label("coffee");
    g_signal_connect(GTK_OBJECT(check),"clicked",
                     GTK_SIGNAL_FUNC(click_button),"check button1");
    gtk_box_pack_start(GTK_BOX(box),check,TRUE,TRUE,0);
  gtk_widget_show(check);

    check = gtk_check_button_new_with_label("tea");
```

我爱自由 5ifreedom.com

```
        g_signal_connect(GTK_OBJECT(check),"clicked",
                         GTK_SIGNAL_FUNC(click_button),"check button2");
        gtk_box_pack_start(GTK_BOX(box),check,TRUE,TRUE,0);
        gtk_widget_show(check);

          /*以下生成3个radio按钮，将它们加入到box容器中，并显示出来*/
          /*

    注意：生成第一个 radio 按钮时 group 参数为 NULL，而后每次在该组中创建一个 radio 按钮都要使用
gtk_radio_button_group 函数获取新的group值
    */
    radio = gtk_radio_button_new_with_label(NULL,"Apple");
         g_signal_connect(GTK_OBJECT(radio),"clicked",
                          GTK_SIGNAL_FUNC(click_button),"Apple");
         gtk_box_pack_start(GTK_BOX(box),radio,TRUE,TRUE,0);
         gtk_widget_show(radio);

         group = gtk_radio_button_group(GTK_RADIO_BUTTON(radio));
         radio = gtk_radio_button_new_with_label(group,"Banana");
         g_signal_connect(GTK_OBJECT(radio),"clicked",
                          GTK_SIGNAL_FUNC(click_button),"Banana");
         gtk_box_pack_start(GTK_BOX(box),radio,TRUE,TRUE,0);
         gtk_widget_show(radio);

         group = gtk_radio_button_group(GTK_RADIO_BUTTON(radio));
         radio = gtk_radio_button_new_with_label(group,"Orange");
         g_signal_connect(GTK_OBJECT(radio),"clicked",
                     GTK_SIGNAL_FUNC(click_button),"Orange");
    /*将第三个radio按钮设置为选中状态*/
         gtk_toggle_button_set_active(GTK_TOGGLE_BUTTON(radio),TRUE);
         gtk_box_pack_start(GTK_BOX(box),radio,TRUE,TRUE,0);
         gtk_widget_show(radio);

         gtk_widget_show(box);
         gtk_widget_show(window);

         gtk_main();
         return 0;
    }
```

运行程序后，显示如图 12-3 所示的界面。

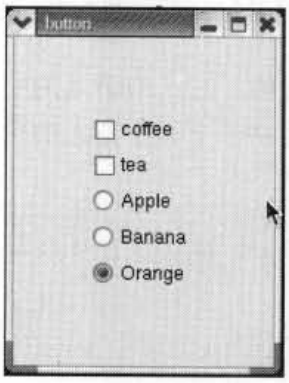


图 12-3　运行结果界面

12.5.2　使用 table 排列控件

1. 创建和使用 table 容器

排列窗口中的控件的另一种方法是使用 table（表格），可以把控件放到表格中指定的行和列中。

我爱自由 5ifreedom.com

表格的行、列编号方法如图 12-4 所示。

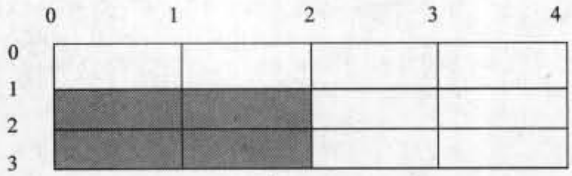


图 12-4 表格行列编号方法

例如，某个控件占据图 12-4 阴影部分，则它所占格子的坐标为 left（左）0，right（右）2，top（上）1，bottom（下）3。

创建 table（表格）容器的函数为：

```
GtkWidget* gtk_table_new( gint rows, gint columns, gint homogeneous );
```

参数的含义如下。

- rows：表格所占的行数。
- columns：表格所占的列数。
- homogeneous：如果其值为 TRUE，表格中每个格子的大小被定义为其中最大控件的大小；如果为 FALSE，则格子的宽度与最宽控件的宽度相同，高度与放入表格的最高控件相同。

将控件放入表格中，可以使用函数：

```
void gtk_table_attach( GtkTable  *table,
                       GtkWidget  *child,
                       gint          left_attach,
                       gint          right_attach,
                       gint          top_attach,
                       gint          buttom_attach,
                       gint          xoptions,
                       gint          yoptions,
                       gint          xpadding,
                       gint          ypadding );
```

参数的含义如下。

- table：要放入控件的表格。
- child：要放入表格的控件。
- left_attach、right_attach、top_attach、buttom_attach：控件在表格中的坐标。
- xoptions、yoptions：指定了选项，可以是以下值或其组合：GTK_FILL，如果控件小于它所占用的格子，控件自动扩大到它所占格子的大小。GTK_SHRINK，如果控件大于它所占用的格子，控件自动缩小到它所占格子的大小。GTK_EXPAND，表格扩展，并利用窗口中所有可用的控件。
- xpadding：指示控件与它所占格子左、右留出的空白大小，以像素表示。
- ypadding：指示控件与它所占格子上、下留出的空白大小，以像素表示。

另一个将控件放入表格的函数是：

```
void gtk_table_attach_defaults( GtkTable      *table,
                                GtkWidget     *child,
                                gint          left_attach,
                                gint          right_attach,
                                gint          top_attach,
                                gint          buttom_attach );
```

此函数参数的含义与 gtk_table_attach()相同。

为了更准确地理解这些参数的含义，最好在运行例子程序时改变这些参数的值，然后查看程序显示的图形界面。

在演示表格控件的使用方法前，先介绍几个程序将会用到的控件。

2. 标签控件

标签（label）控件在界面上显示一段文本。生成 label 控件的函数是：

我爱自由 5ifreedom.com

```
GtkWidget* gtk_label_new(char *str);
```

生成标签控件后，修改所显示的文字可以使用下面的函数：

```
void gtk_label_set( GtkLabel *label, char *str );
```

获取当前标签所显示的文本的函数如下：

```
void gtk_label_get( GtkLabel *label, char **str );
```

3. 编辑框控件

编辑框控件允许用户输入一行文本，生成编辑框的函数为：

```
GtkWidget* gtk_entry_new(void);
GtkWidget* gtk_entry_new_with_max_length(guint16 max);
```

其中，第二个函数限制了能输入到编辑框的最大字符数。

获取用户输入到编辑框架中的文本的函数是：

```
gchar* gtk_entry_get_text(GtkEntry *entry);
```

设置编辑框中的文本的函数是：

```
void gtk_entry_set_text( GtkEntry *entry, gchar *text );
```

设置能输入到编辑框中的文本最大数的函数是：

```
void gtk_entry_set_max_length( GtkEntry *entry, guint16 max);
```

设置是否允许用户向编辑框中输入文本的函数是：

```
void gtk_entry_set_editable( GtkEntry *entry, gboolean editable);
```

例 12-4 表格容器、box 容器、标签控件和编辑框控件的使用，程序名为 box.c。

```
#include<gtk/gtk.h>

/*函数声明*/
GtkWidget* makeTable();
GtkWidget* makeTextEntry();
GtkWidget* makecheckButtons();
GtkWidget* makeButtonBox();

/*单击 check 按钮的回调函数*/
void click_button(GtkWidget *widget,gpointer *data)
{
     g_print("click %s ",(char *)data);

     if(GTK_TOGGLE_BUTTON(widget)->active)
          g_print("and state is active\n");
     else
          g_print("and state is not active\n");
}

void destroy(GtkWidget *widget,gpointer *data)
{
     gtk_main_quit();
}

int main(int argc,char **argv)
{
     GtkWidget *window;
     GtkWidget *table;

     gtk_init(&argc,&argv);

     window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
     g_signal_connect(GTK_OBJECT(window),"destroy",
                      GTK_SIGNAL_FUNC(destroy),NULL);
     gtk_container_border_width(GTK_CONTAINER(window),30);


     table = makeTable();
     gtk_container_add(GTK_CONTAINER(window),table);

     gtk_widget_show(window);
```

我爱自由 5ifreedom.com

```
    gtk_main();

    return 0;
}

GtkWidget* makeTable()
{
    GtkWidget *table;
    GtkWidget *checkButtons;
    GtkWidget *textEntry;
    GtkWidget *buttonBox;

    /*创建 table 控件*/
    table = gtk_table_new(2,2,FALSE);
    gtk_widget_show(table);

   /*创建标签和编辑框*/
    textEntry = makeTextEntry();
    gtk_table_attach(GTK_TABLE(table),textEntry,
                 0,1,0,1,
                 GTK_FILL|GTK_EXPAND|GTK_SHRINK,0,
                 0,0);
    gtk_widget_show(textEntry);

    /*创建 4 个 check 按钮*/
    checkButtons = makecheckButtons();
    gtk_table_attach(GTK_TABLE(table),checkButtons,
                 1,2,0,1,
                  GTK_FILL | GTK_EXPAND,
                  GTK_FILL | GTK_EXPAND,
                  10,0);
    gtk_widget_show(checkButtons);

    /*创建两个按钮*/
    buttonBox = makeButtonBox();
    gtk_table_attach(GTK_TABLE(table),buttonBox,
                   0,2,1,2,
                 GTK_EXPAND|GTK_FILL|GTK_SHRINK,0,
                 5,10);
    gtk_widget_show(buttonBox);

    return table;
}

GtkWidget* makeTextEntry()
{
    GtkWidget *vbox;
    GtkWidget *label;
    GtkWidget *text;

    vbox = gtk_vbox_new(FALSE,5);

    /*生成标签控件*/
    label = gtk_label_new("please enter your name:");
    gtk_box_pack_start(GTK_BOX(vbox),label,FALSE,FALSE,0);
    gtk_widget_show(label);

    /*生成编辑框*/
    text = gtk_entry_new_with_max_length(15);
    gtk_box_pack_start(GTK_BOX(vbox),text,FALSE,FALSE,0);
    gtk_widget_show(text);

    return vbox;
}

GtkWidget* makecheckButtons()
```

我爱自由 5ifreedom.com

```
{
    GtkWidget *vbox;
    GtkWidget *check;

    vbox = gtk_vbox_new(FALSE,0);

    check = gtk_check_button_new_with_label("apple");
    g_signal_connect(GTK_OBJECT(check),"clicked",
                     GTK_SIGNAL_FUNC(click_button),"apple");
    gtk_box_pack_start(GTK_BOX(vbox),check,FALSE,FALSE,0);
    gtk_widget_show(check);

    check = gtk_check_button_new_with_label("banana");
    g_signal_connect(GTK_OBJECT(check),"clicked",
                     GTK_SIGNAL_FUNC(click_button),"banana");
    gtk_box_pack_start(GTK_BOX(vbox),check,FALSE,FALSE,0);
    gtk_widget_show(check);

    check = gtk_check_button_new_with_label("orange");
    g_signal_connect(GTK_OBJECT(check),"clicked",
                     GTK_SIGNAL_FUNC(click_button),"orange");
    gtk_box_pack_start(GTK_BOX(vbox),check,FALSE,FALSE,0);
    gtk_widget_show(check);

    check = gtk_check_button_new_with_label("pear");
    g_signal_connect(GTK_OBJECT(check),"clicked",
                     GTK_SIGNAL_FUNC(click_button),"pear");
    gtk_box_pack_start(GTK_BOX(vbox),check,FALSE,FALSE,0);
    gtk_widget_show(check);

    return vbox;
}

GtkWidget* makeButtonBox()
{
    GtkWidget *hbox;
    GtkWidget *button;

    hbox = gtk_hbox_new(FALSE,0);

    button = gtk_button_new_with_label("yes");
    gtk_box_pack_start(GTK_BOX(hbox),button,TRUE,TRUE,20);
    gtk_widget_show(button);

    button = gtk_button_new_with_label("no");
    gtk_box_pack_start(GTK_BOX(hbox),button,TRUE,TRUE,60);
    gtk_widget_show(button);

    return hbox;
}
```

程序运行后的界面如图 12-5 所示。

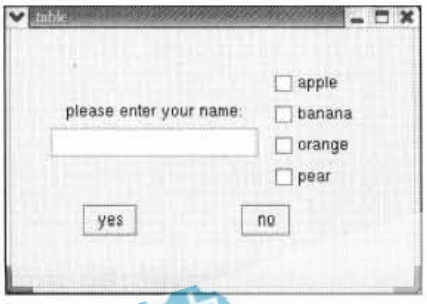


图 12-5 运行结果界面

我爱自由 5ifreedom.com

12.6 常用控件

12.6.1 进度条、微条按钮、组合框

1. 进度条控件

创建进度条要使用 GtkAdjustment 控件。GtkAdjustment 用来存储上边界、下边界、步进值等信息。

（1）创建 GtkAdjustment

```
GtkObject* gtk_adjustment_new(gfloat value,gfloat lower,gfloat upper,
                              gfloat step_increment,gfloat page_increment,gloat page_size)
```

（2）创建进度条

```
GtkWidget* gtk_progress_bar_new(void)
GtkWidget* gtk_progress_bar_new_with_adjustment(GtkAdjustment *adjustment)
```

（3）设置进度条的样式

```
void gtk_progress_bar_set_bar_style(GtkProgressBar *pbar,
                                    GtkProgressBarStyle style)
```

其中 style 取值如下。

- GTK_PROGRESS_CONTINUOUS：连续进度条。
- GTK_PROGRESS_DISCRETE：条块进度条。

（4）设置进度条方向

```
void gtk_progress_bar_set_orientation(GtkProgressBar *pbar,
                                      GtkProgressBarOrientation orientation)
```

其中 orientation 取如下值。

- GTK_PROGRESS_LEFT_TO_RIGHT：从左往右显示进度。
- GTK_PROGRESS_RIGHT_TO_LEFT：从右往左显示进度。
- GTK_PROGRESS_BOTTOM_TO_TOP：从下往上显示进度。
- GTK_PROGRESS_TOP_TO_BOTTOM：从上往下显示进度。

（5）更新进度

```
void gtk_progress_bar_update(GtkProgressBar *pbar, gfloat percentage)
```

2. 微调按钮

创建微调按钮也要使用 GtkAdjustment 控件。

（1）创建 GtkSpinButton

```
GtkWidget* gtk_spin_button_new(GtkAdjustment *adjustment,
                               gfloat climb_rate,gfloat digits)
```

参数含义如下。

- climb_rate：每步的增加值。
- digits：包含的小数位数。

（2）获取和设置微调按钮的值

```
gfloat gtk_spin_button_get_value_as_float(GtkSpinButton *spin_button)
void gtk_spin_button_set_value(GtkSpinButton *spin_button,gfloat value)
```

（3）获取和设置 GtkAdjustment

```
GtkAdjustment* gtk_spin_button_get_adjustment(GtkSpinButton *spin_button)
void gtk_spin_button_set_adjustment(GtkSpinButton *spin_button,
                                    GtkAdjustment *adjustment)
```

我爱自由 5ifreedom.com

3. 组合框

组合框是编辑框和列表框的组合。

（1）创建组合框要使用 GList，用于保存显示的字符串。向 GList 中添加字符串的函数是：

```
void g_list_append(GList *list,char *string)
```

（2）创建组合框

```
GtkWidget* gtk_combo_new(void)
```

（3）设置组合框中显示的字符串

```
void gtk_combo_set_popdown_strings(GtkCombo *combo,GList *strings)
```

例 12-5 进度条、组合框、微条按钮的使用，程序名为 control.c

```
#include<gtk/gtk.h>

int main(int argc,char **argv)
{
    GtkWidget *window;
    GtkWidget *vbox;
    GtkObject *adjustment;
    GtkWidget *bar;
    GtkWidget *spinbutton;
    GList      *glist;
    GtkWidget *combo;

    gtk_init(&argc,&argv);

    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(GTK_OBJECT(window),"destroy",
                     GTK_SIGNAL_FUNC(gtk_main_quit),NULL);
    gtk_container_border_width(GTK_CONTAINER(window),40);

    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),vbox);

    /*创建进度条*/
    adjustment = gtk_adjustment_new(70.0,0.0,100.0,1.0,0.0,0.0);
    bar = gtk_progress_bar_new_with_adjustment(GTK_ADJUSTMENT(adjustment));
    gtk_progress_bar_set_bar_style(GTK_PROGRESS_BAR(bar),
                                   GTK_PROGRESS_CONTINUOUS);
    gtk_progress_bar_set_orientation(GTK_PROGRESS_BAR(bar),
                                   GTK_PROGRESS_LEFT_TO_RIGHT);
    gtk_box_pack_start(GTK_BOX(vbox),bar,TRUE,TRUE,15);
    gtk_widget_show(bar);

    /*微调按钮*/
    adjustment = gtk_adjustment_new(80.0,0.0,100.0,1.0,0.0,0.0);
    spinbutton = gtk_spin_button_new(GTK_ADJUSTMENT(adjustment),1.0,1);
    gtk_box_pack_start(GTK_BOX(vbox),spinbutton,TRUE,TRUE,15);
    gtk_widget_show(spinbutton);

    /*创建组合框*/
    glist = NULL;
    glist = g_list_append(glist,"apple");
    glist = g_list_append(glist,"banana");
    glist = g_list_append(glist,"orange");
    glist = g_list_append(glist,"pear");
    combo = gtk_combo_new();
    gtk_combo_set_popdown_strings(GTK_COMBO(combo),glist);
    gtk_box_pack_start(GTK_BOX(vbox),combo,TRUE,TRUE,15);
    gtk_widget_show(combo);

    gtk_widget_show(vbox);
    gtk_widget_show(window);

    gtk_main();

    return 0;
}
```

程序运行后的界面如图 12-6 所示。

我爱自由 5ifreedom.com

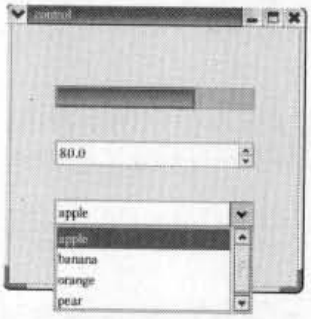


图 12-6　运行结果界面

12.6.2　表格控件

（1）创建表格控件

```
GtkWidget * gtk_clist_new(gint columns)
GtkWidget * gtk_clist_new_with_titles(gint columns,gchar *titles[])
```

参数含义如下。

- columns：表格的列数。
- titles：各列的列名。

（2）设置每列的长度

```
void gtk_clist_set_column_width(GtkCList *clist,gint column,gint width)
```

参数含义如下。

- column：要设置宽度的列编号，从 0 开始。
- width：宽度。

（3）向表格中加入一行数据

```
gint gtk_clist_append(GtkCList *clist,gchar *text[])
```

（4）获取和设置表格中某个格子的值

```
gint gtk_clist_get_text(GtkCList *clist,gint row,gint column,gchar **text)
void gtk_clist_set_text(GtkCList *clist,gint row,gint column,gchar **text)
```

（5）清除表格中的所有数据

```
void gtk_clist_clear(GtkCList *clist)
```

（6）对表格中的数据进行排序

```
void gtk_clist_sort_column(GtkCList *clist, gint column)
void gtk_clist_sort (GtkCList *clist)
```

（7）获取和设置一行的数据

```
gpointer gtk_clist_get_row_data(GtkCList *clist,gint row)
void gtk_clist_set_row_data(GtkCList *clist,gint row, gpointer data)
```

（8）获取和更改列名

```
gchar* gtk_clist_get_column_title(GtkCList *clist, gint column)
void gtk_clist_set_column_title(GtkCList *clist, gint column,gchar *title)
```

（9）插入和删除一行数据

```
gint gtk_clist_insert(GtkCList *clist, gint row,gchar *text[])
void gtk_clist_remove(GtkCList *clist, gint row)
```

例 12-6　表格控件的使用，程序名为 table.c

```
#include<gtk/gtk.h>

int main(int argc,char **argv)
{
    GtkWidget *window;
    GtkWidget *clist;

    gtk_init(&argc,&argv);

    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
```

我爱自由 5ifreedom.com

```
    g_signal_connect(GTK_OBJECT(window),"destroy",
                     GTK_SIGNAL_FUNC(gtk_main_quit),NULL);
    gtk_container_border_width(GTK_CONTAINER(window),20);

    char *titles[] = {
        "name","age","major"
    };

    char *rows1[] = {
        "John","20","computer"
    };

    char *rows2[] = {
        "Bill","24","maths"
    };

    char *rows3[] = {
        "Martin","22","English"
    };

    clist = gtk_clist_new_with_titles(3,titles);
    gtk_clist_set_column_width(GTK_CLIST(clist),0,50);
    gtk_clist_set_column_width(GTK_CLIST(clist),1,50);
    gtk_clist_set_column_width(GTK_CLIST(clist),2,150);
    gtk_container_add(GTK_CONTAINER(window),clist);
    gtk_clist_append(GTK_CLIST(clist),rows1);
    gtk_clist_append(GTK_CLIST(clist),rows2);
    gtk_clist_append(GTK_CLIST(clist),rows3);

    /*获取表格中第 0 行第 2 列格子的数据，并在命令行上打印出来*/
    char *text = (char *)g_malloc(32);
    gtk_clist_get_text(GTK_CLIST(clist),0,2,&text);
    g_print("%s\n",text);

    gtk_clist_sort(GTK_CLIST(clist));

    gtk_widget_show(clist);
    gtk_widget_show(window);

    gtk_main();

    return 0;
}
```

程序运行后的界面如图 12-7 所示。

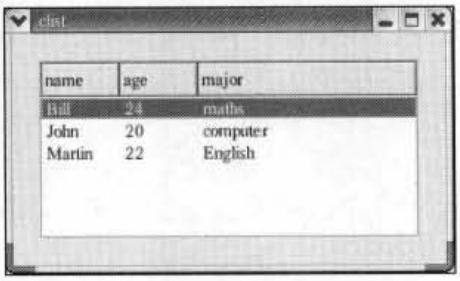


图 12-7　运行结果界面

12.6.3　生成对话框

对话框 GtkDialog 是 GtkWindow 的子类，它可以使用所有 GtkWindow 的函数。创建对话框的函数是：

```
GtkWidget *gtk_dialog_new(void)
```

运行下面的程序例 12-7，显示主窗口，主窗口中有一个按钮，单击该按钮生成一个对话框。

例 12-7　创建对话框，程序名为 dialog.c。

```
#include<gtk/gtk.h>
```

我爱自由 5ifreedom.com

```
void make_dialog()
{
     GtkWidget *dialog;
     GtkWidget *label;
     GtkWidget *button;
     GtkWidget *vbox;
     GtkWidget *hbox;

     dialog = gtk_dialog_new();

     /*向对话框中加入一个文本标签*/
     vbox = GTK_DIALOG(dialog)->vbox;
     label = gtk_label_new("This is a dialog");
     gtk_box_pack_start(GTK_BOX(vbox),label,TRUE,TRUE,30);

     /*向对话框中加入两个按钮*/
     hbox = GTK_DIALOG(dialog)->action_area;
     button = gtk_button_new_with_label("yes");
     gtk_box_pack_start(GTK_BOX(hbox),button,FALSE,FALSE,0);
     button = gtk_button_new_with_label("no");
     gtk_box_pack_start(GTK_BOX(hbox),button,FALSE,FALSE,0);

     gtk_widget_show_all(dialog);
}

void hello(GtkWidget *widget,gpointer *data)
{
       make_dialog();
}

int main(int argc,char **argv)
{
     GtkWidget *window;
     GtkWidget *button;

     gtk_init(&argc,&argv);

     window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
     g_signal_connect(GTK_OBJECT(window),"destroy",
                      GTK_SIGNAL_FUNC(gtk_main_quit),NULL);
     gtk_container_border_width(GTK_CONTAINER(window),20);

     button = gtk_button_new_with_label("Hello World");
     g_signal_connect(GTK_OBJECT(button),"clicked",
                      GTK_SIGNAL_FUNC(hello),"I am from button");

     gtk_container_add(GTK_CONTAINER(window),button);
     gtk_widget_show(button);
     gtk_widget_show(window);

     gtk_main();

     return 0;
}
```

程序运行后的界面如图 12-8 所示。

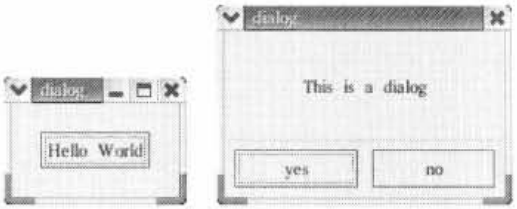


图 12-8　运行结果界面

12.6.4　使用菜单

（1）创建菜单

我爱自由 5ifreedom.com

```
GtkWidget * gtk_menu_new()
GtkWidget * gtk_menu_new_with_label(gchar *label)
```

（2）生成菜单项

```
GtkWidget * gtk_menu_item_new()
GtkWidget * gtk_menu_item_new_with_label(gchar *label)
```

（3）插入菜单项

```
void gtk_menu_append(GtkMenu *menu,GtkWidget *child)
void gtk_menu_set_submenu(GtkMenuItem *item GtkMenu *menu)
```

（4）创建菜单条

```
GtkWidget *gtk_menu_bar_new(void)
```

（5）向菜单条中加入菜单

```
void gtk_menu_bar_append(GtkMenuBar *bar, GtkWidget *child)
```

创建菜单的步骤如下。

① 使用 gtk_menu_new()或 gtk_menu_new_with_label()生成一个新菜单。

② 使用 gtk_menu_item_new()或 gtk_menu_new_with_label()生成一个新的菜单项，然后使用 gtk_menu_append()将菜单项加入到菜单中。

③ 使用 gtk_menu_item_new()或 gtk_menu_new_with_label()创建主菜单。

④ 使用 gtk_menu_set_submenu()将各个菜单加入到主菜单中。

⑤ 使用 gtk_menu_bar_new()创建菜单条。然后使用 gtk_menu_bar_append()把主菜单加入到菜单条上。

例 12-8 创建菜单，程序名为 menu.c。

```
#include<gtk/gtk.h>

int main(int argc,char **argv)
{
    GtkWidget *window;
    GtkWidget *menu;
    GtkWidget *menubar;
    GtkWidget *rootmenu;
    GtkWidget *menuitem;

    gtk_init(&argc,&argv);

    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window),"Menu Demo");
    g_signal_connect(GTK_OBJECT(window),"destroy",
                     GTK_SIGNAL_FUNC(gtk_main_quit),NULL);
    gtk_container_border_width(GTK_CONTAINER(window),20);

    /*创建一个新菜单，然后创建 3 个菜单项，并把这 3 个菜单项加入到菜单中*/
    menu = gtk_menu_new();

    menuitem = gtk_menu_item_new_with_label("New");
    gtk_menu_append(GTK_MENU(menu),menuitem);
    gtk_widget_show(menuitem);

    menuitem = gtk_menu_item_new_with_label("Open");
    gtk_menu_append(GTK_MENU(menu),menuitem);
    gtk_widget_show(menuitem);

    menuitem = gtk_menu_item_new_with_label("Close");
    gtk_menu_append(GTK_MENU(menu),menuitem);
    gtk_widget_show(menuitem);

    /*创建一个主菜单*/
    rootmenu = gtk_menu_item_new_with_label("File");
    gtk_widget_show(rootmenu);
```

我爱自由 5ifreedom.com

```
    /*将菜单加入到主菜单中*/
    gtk_menu_item_set_submenu(GTK_MENU_ITEM(rootmenu),menu);
    /*创建菜单条*/
    menubar = gtk_menu_bar_new();
    /*将主菜单条加入到菜单条中*/
    gtk_menu_bar_append(GTK_MENU_BAR(menubar),rootmenu);

    /*使用同样的方法，创建第二组菜单*/
    menu = gtk_menu_new();
    menuitem = gtk_menu_item_new_with_label("Cut");
    gtk_menu_append(GTK_MENU(menu),menuitem);
    gtk_widget_show(menuitem);

    menuitem = gtk_menu_item_new_with_label("Paste");
    gtk_menu_append(GTK_MENU(menu),menuitem);
    gtk_widget_show(menuitem);

    rootmenu = gtk_menu_item_new_with_label("Edit");
    gtk_widget_show(rootmenu);

    gtk_menu_item_set_submenu(GTK_MENU_ITEM(rootmenu),menu);
    gtk_menu_bar_append(GTK_MENU_BAR(menubar),rootmenu);

    /*将菜单条加入到窗口中，并显示菜单条和窗口*/
    gtk_container_add(GTK_CONTAINER(window),menubar);
    gtk_widget_show(menubar);
    gtk_widget_show(window);

    gtk_main();

    return 0;
}
```

程序运行后的界面如图 12-9 所示。

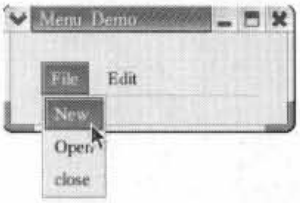


图 12-9　运行结果界面

12.7 进一步学习建议

本章主要介绍了 GTK+图形界面应用程序的框架、基本原理和常用控件的使用方法。但由于篇幅所限，不能介绍所有控件。如果要进一步学习 GTK+编程，可以参考以下资料。

宋国伟《GTK+ 2.0 编程范例》该书以范例程序的形式，由浅入深地引导读者学习开源软件领域和 Linux 平台上最为著名的图形界面开发工具 GTK+ 2.0 的使用方法。

Arthur Griffith 著、吴向峰译《GNOME/GTK+编程宝典》该书以大量实例，辅以通俗易懂的解释，逐步讲解如何使用 GTK+和 GNOME 构建对话框、事件、工具条及其他界面元素，以及各种小部件、窗口和多文档界面（MDI）。

http://www.gtk.org/　该网站是 GTK+开发包的官方网站，不仅有各个 GTK+函数的详细注释，还有 GTK+的使用手册以及 GTK+最新信息和常见问题解答。

我爱自由 5ifreedom.com

LINUX

第四篇　Linux 项目实践

项目实践：BT 下载软件的开发

我爱自由
5ifreedom.com

第 13 章　项目实践：BT 下载软件的开发

本章通过开发一个 BT 下载软件来完整地介绍一个软件的开发过程。详细分析和解释了 BT 协议，并在此基础上使用 C 语言在 Linux 环境下设计和实现了一个 BT 下载软件。

本章重点：

- BT 协议
- 系统结构设计。

本章难点：

系统各个模块的实现是本章的难点，包括：

- 种子解析模块的设计和实现；
- 位图管理模块的设计和实现；
- 出错处理模块的设计和实现；
- 运行日志模块的设计和实现；
- 信号处理模块的设计和实现；
- Peer 管理模块的设计和实现；
- 消息处理模块的设计和实现；
- 缓冲管理模块的设计和实现；
- 策略管理模块的设计和实现；
- 连接 Tracker 模块的设计和实现；
- 与 Peer 交换数据模块的设计和实现。
- 主函数的设计和实现。

13.1 BT 软件简述

可能许多人使用过比特彗星(BitComet)、比特精灵(BitSpirit)、迅雷下载过自己喜欢的影片、电视剧、网络游戏；还有很多人使用过 PPLive、PPStream、沸点、QQ 直播等免费的网络电视直播软件在线观看自己喜欢的影片。所有这些软件都采用了一种近年来流行起来的协议，BitTorrent 协议，简称 BT 协议。

在互联网中，许多新技术深刻地改变了人们的工作、生活和学习的模式。Tim Berners Lee 在 1990 年设计和发明了 HTTP 协议，从而引发了互联网的变革，使网络冲浪、电子商务成为可能，因此也造就了百度、谷歌等搜索引擎公司以及网易、雅虎、搜狐、新浪、腾讯等门户网站，同时也造就了一个又一个的数字英雄。在 HTTP 协议发明之前，统治互联网的是 SMTP 和 FTP 协议，这两种协议的通信量占据首位；HTTP 协议诞生之后，其通信流量和使用率都占据了第一。2003

我爱自由 5ifreedom.com

年，年轻的软件工程师 Bram Cohen 发明了 BitTorrent 协议。在短短的时间内，BT 协议的通信流量占据了互联网总流量的六成以上。BT 协议成为一种新的变革技术，因此也催生了很多 BT 软件，如 BitComet、BitSpirit、Azureus，PPLive、PPStream。

下面将详细介绍 BT 协议和技术的各个细节，并在此基础上使用 C 语言在 Linux 环境下开发了一个 BT 软件。

13.2 BitTorrent 协议

13.2.1 概要介绍

BitTorrent（简称 BT）是一个文件分发协议，每个下载者在下载的同时不断向其他下载者上传已下载的数据。而在 FTP、HTTP 协议中，每个下载者从 FTP 或 HTTP 服务器处下载自己所需要的文件，各个下载者之间没有交互。当非常多的用户同时访问和下载服务器上的文件时，由于 FTP 服务器的处理能力和带宽的限制，下载速度会急剧下降，有的用户根本访问不了服务器。BT 协议与 FTP 协议不同，它的特点是下载的人越多下载的速度越快，其原因在于每个下载者将已下载的数据提供给其他下载者下载，它充分利用了用户的上载带宽。BT 协议通过一定的策略保证上传的速度越快，下载的速度也越快。

13.2.2 基于 BT 协议的文件分发系统的构成

基于 BT 协议的文件分发系统由以下几个实体构成。

（1）一个 Web 服务器。

（2）一个种子文件。

（3）一个 Tracker 服务器。

（4）一个原始文件提供者。

（5）一个网络浏览器。

（6）一个或多个下载者。

Web 服务器上保存着种子文件，下载者使用网络浏览器（如 IE 浏览器）从 Web 服务器上下载种子文件。种子文件，又称为元原文件或 metafile，它保存了共享文件的一些信息，如共享文件的文件名、文件大小、Tracker 服务器的地址。种子文件通常很小，一般大小为 1GB 的共享文件，其种子文件不足 100KB，种子文件以.torrent 为后缀。Tracker 服务器保存着当前下载某共享文件的所有下载者的 IP 和端口。原始文件提供者提供完整的共享文件供其他下载者下载，它也被称为种子，种子文件就是提供者使用 BT 客户端生成的。每个下载者通过运行 BT 客户端软件下载共享文件。我们把某个下载者本身称为客户端，把其他下载者称为 peer。

BT 客户端下载一个共享文件的过程是：客户端首先解析种子文件，获取待下载的共享文件的一些信息，其中包括 Tracker 服务器的地址。然后客户端连接 Tracker 获取当前下载该文件的所有下载者的 IP 和端口。之后客户端根据 IP 和端口连接其他下载者，从它们那里下载文件，同时把自己已下载的部分提供给其他下载者下载。

共享文件在逻辑上被划分为大小相同的块，称为 piece，每个 piece 的大小通常为 256KB。对于

我爱自由 5ifreedom.com

共享文件，文件的第 1 字节到第 256K（即 262144）字节为第一个 piece，第 256K＋1 字节到第 512K 字节为第二个 piece，依此类推。种子文件中包含有每个 piece 的 hash 值。BT 协议规定使用 Sha1 算法对每个 piece 生成 20 字节的 hash 值，作为每个 piece 的指纹。每当客户端下载完一个 piece 时，即对该 peice 使用 Sha1 算法计算其 hash 值，并与种子文件中保存的该 peice 的 hash 值进行比较，如果一致即表明下载了一个完整而正确的 piece。一旦某个 piece 被下载，该 piece 即提供给其他 peer 下载。在实际上传和下载中，每个 piece 又被划分为大小相同的 slice，每个 slice 的大小固定为 16KB（16384 字节）。peer 之间每次传输以 slice 为单位。

从以上描述可以得知，待开发的 BT 软件（即 BT 客户端）主要包含以下几个功能：解析种子文件获取待下载的文件的一些信息，连接 Tracker 获取 peer 的 IP 和端口，连接 peer 进行数据上传和下载、对要发布的提供共享文件制作和生成种子文件。种子文件和 Tracker 的返回信息都以一种简单而高效的编码方式进行编码，称为 B 编码。客户端与 Tracker 交换信息基于 HTTP 协议，Tracker 本身作为一个 Web 服务器存在。客户端与其他 peer 采用面向连接的可靠传输协议 TCP 进行通信。下面将进一步作详细的介绍。

13.2.3 B 编码

种子文件和 Tracker 的返回信息都是经过 B 编码的。要解析和处理种子文件以及 Tracker 的返回信息，首先要熟悉 B 编码的规则。B 编码中有 4 种类型：字符串、整型、列表、字典。

字符串的编码格式为：<字符串的长度>:<字符串>，其中<>括号中的内容为必需。例如，有一个字符串 spam，则经过 B 编码后为 4:spam。

整型的编码格式为：i<十进制的整型数>e，即 B 编码中的整数以 i 作为起始符，以 e 作为终结符，i 为 integer 的第一个字母，e 为 end 的第一个字母。例如，整数 3，经过 B 编码后为 i3e，整数-3 的 B 编码为 i-3e，整数 0 的 B 编码为 i0e。

注意 i03e 不是合法的 B 编码，因为 03 不是十进制整数，而是八进制整数。

列表的编码格式为：l<任何合法的类型>e，列表以 l 为起始符，以 e 为终结符，中间可以为任何合法的经过 B 编码的类型，l 为 list 的第一个字母。例如，列表 l4:spam4:eggse 表示两个字符串，一个是 spam，另一个是 eggs。

字典的编码格式为：d<关键字><值>e，字典以 d 为起始符，以 e 为终结符，关键字是一个经过 B 编码的字符串，值可以是任何合法的 B 编码类型，在 d 和 e 之间可以出现多个关键字和值对，d 是 dictionary 的第一个字母。例如，d4:spaml3:aaa3:bbbee，它是一个字典，该字典的关键字是 spam，值是一个列表（以 l 开始，以 e 结束），列表中有两个字符串 aaa 和 bbb。

又如：d9:publisher3:bob17:publisher-webpage15:www.example.come，它也是一个字典，第一个关键字是 publisher，对应的值为 bob，第二个关键字是 publisher-webpage，对应的值是 www.example.com。

13.2.4 种子文件的结构

种子文件包含了提供共享的文件的一些信息，它以.torrent 为后缀名，种子文件也被称为元信息文件或 metafile，它是经过 B 编码的。种子文件事实上就是一个 B 编码的字典，它含有以下关键字如表 13-1 所示。

我爱自由 5ifreedom.com

表 13-1　　种子文件的关键字

关 键 字	含　义
info	该关键字对应的值是一个字典，它有两种模式，“singel file”和“multiple file”，文件模式和多文件模式。单文件模式是指待共享的文件只有一个，多文件模式是指提供共享的不止一个文件，而是两个或两个以上。如使用 BT 软件下载一部影片时，影片的上下部可能分别放在不同的文件里
announce	该关键字的值为 Tracker 的 URL
announce－list	可选，它的值存放的是备用 Tracker 的 URL
creation－date	可选，该关键字对应的值存放的是创建种子文件的时间
comment	可选，它的值存放的是种子文件制作者的备注信息，对于下载来说，该关键字基本没有用处，因此不必理会
created by	可选，该关键字对应的值存放的是生成种子文件的 BT 客户端软件的信息，如客户端名、版本号等，一般不必理会

info 是最重要的一个关键字，它的值是一个字典，下面对它再作进一步的介绍。无论是单文件模式还是多文件模式，该字典都包含关键字如表 13-2 所示。

表 13-2　　info 包含的关键字

关 键 字	含　义
piece length	每个 piece 的长度，它的值是一个 B 编码的整型，该值通常为 i262144e，即 256K，也有可能为 512K 或 128K
pieces	对应的值为一个字符串，它存放的是各个 piece 的 hash 值，这个字符串的长度一定是 20 的倍数，因为每个 piece 的 hash 值的长度为 20 字节
private	该值如果为 1，则表明客户端必须通过连接 Tracker 来获取其他下载者，即 peer 的 IP 地址和端口号；如果为 0，则表明客户端还可以通过其他方式来获取 peer 的 IP 地址和端口号，如 DHT 方式。DHT 即分布式哈希表（Distribute Hash Tabel），它是一种以分布式的方式来获取 peer 的方法，现在许多 BT 客户端既支持通过连接 Tracker 来获取 peer，也支持通过 DHT 来获取 peer。如果种子文件中没有 private 这个关键字，则表明不限制一定要通过连接 Tracker 来获取 peer

对于单文件模式的种子文件，info 的值还含有的关键字如表 13-3 所示。

表 13-3　　单模式种子文件的关键字

关 键 字	含　义
name	共享文件的文件名，也就是要下载的文件的文件名
length	共享文件的长度，以字节为单位
md5sum	可选，它是共享文件的 md5 值，这个值在 BT 协议中根本没有使用，所以不必理会

对于多文件模式的种子文件，info 的值还含有的关键字如表 13-4 所示。

表 13-4　　多文件模式种子文件的关键字

关 键 字	含　义
name	存放所有共享文件的文件夹名
files	它的值是一个列表，列表中含有多个字典，每个共享文件为一个字典。该字典中含有三个关键词

files 的每个共享文件为一个字典，字典的关键词如表 13-5 所示。

表 13-5　　files 字典的关键词

关 键 词	含　义
length	共享文件的长度，以字节为单位
md5sum	可选，同上
path	存放的是共享文件的路径和文件名

建议读者到一些提供 BT 种子文件下载的网站，如 bt.greedland.net、www.btchina.net，下载几个种子文件并在 Windows 操作系统下使用记事本打开进行分析，就可以清楚的了解上述概念。

我爱自由 5ifreedom.com

13.2.5 与 Tracker 交互

完成解析种子文件并从中获取 Tracker 服务器的 URL 后，即可开始与 Tracker 进行交互。与 Tracker 进行交互主要有两个目的：一是将自己的下载进度告知给 Tracker 以便 Tracker 进行一些相关的统计；二是获取当前下载同一个共享文件的 peer 的 IP 地址和端口号。

客户端使用 HTTP 协议与 Tracker 进行通信。Tracker 通过 HTTP GET 方法获取请求，请求的构成为 Tracker 的 URL 后面跟一个？以及参数和值对，如 http://tk.greedland.net/ announce?param1= value1&param2= value2。

在客户端发往 Tracker 的 GET 请求中，通常包含参数如表 13-6 所示。

表 13-6　　GET 请求的参数

参　数	含　义
info_hash	与种子文件中 info 关键字对应的值，通过 Sha1 算法计算其 hash 值，该 hash 值就是 info_hash 参数对应的值，该 hash 值的长度固定为 20 字节
peer_id	每个客户端在下载文件前以随机的方式生成的 20 字节的标识符，用于标识自己，它的长度也是固定不变的
port	监听端口号，用于接收其他 peer 的连接请求
uploaded	当前总的上传量，以字节为单位
downloaded	当前总的下载量，以字节为单位
left	还剩余多少字节需要下载，以字节为单位
compact	该参数用于指示服务器以何种方式返回 peer，该值为 1 时，每个 peer 占 6 个字节，前 4 个字节为 peer 的 IP 地址，后两个为 peer 的端口号。
event	它的值为 started、completed、stopped 其中之一。客户端第一次与 Tracker 进行通信时，该值为 started；下载完成时，该值为 completed；客户端即将关闭时，该值为 stopped
ip	可选，将客户端的 IP 地址告知给 Tracker，Tracker 可以通过分析客户端发给 Tracker 的 IP 数据包来获取客户端的 IP 地址，因此该参数是可选的，一般不用指明客户端的 IP
numwant	可选，希望 Tracker 返回多少个 peer 的 IP 地址和端口号。如果该参数缺省，则默认返回 50 个 peer 的 IP 地址和端口号
key	可选，它的值为一个随机数，用于进一步标识客户端。因为已经由 peer_id 来标识客户端，因此该参数一般不使用
trackerid	可选，一般不使用

Tracker 服务器的返回信息是一个经过 B 编码的字典。它含有关键字如表 13-7 所示。

表 13-7　　Tracker 服务器返回信息关键字

关 键 字	含　义
failure reason	该关键字对应的值是一个可以读懂的字符串，指明 GET 请求失败的原因，如果返回信息中含有这个关键字，就不会再包含其他任何关键字
warnging message	该关键字对应的值是一个可以读懂的警告字符串
interval	指明客户端在下一次连接 Tracker 前所需等待的时间，以秒为单位
min interval	指明客户端在下一次连接 Tracker 前所需等待的最少时间，以秒为单位
tracker id	指明 Tracker 的 ID
complete	一个整数，指明当前有多少个 peer 已经完成了整个共享文件的下载
incomplete	一个整数，指明当前有多少个 peer 还没有完成共享文件的下载
peers	返回各个 peer 的 IP 和端口号，它的值是一个字符串。首先是第一个 peer 的 IP 地址，然后是其端口号；接着是第二个 peer 的 IP 地址，然后是其端口号；依次类推

以下是一个发往 Tracker 服务器的 HTTP GET 请求的示例：

我爱自由 5ifreedom.com

```
http://tk.greedland.net/announce?info_hash=01234567890123456789&
peer_id=01234567890123456789&port=3210&compact=1&uploaded=0&downloaded=0&left=8000000&event=started
```

以下是一个 Tracker 服务器回应的示例：

```
d8:completei100e10:incompletei200e8:intervali1800e5:peers300:...e
```

其中，“...”是一个长度为 300 的字符串，含有 50 个 peer 的 IP 地址和端口号。IP 地址占 4 字节，端口号占 2 字节，即一个 peer 占 6 字节。

注意：发往 Tracker 服务器的 HTTP GET 请求中，info_hash 和 peer_id 可能含有非数字、非字母的字符，即含有除 0～9、a～z、A～Z 之外的字符，此时要对字符进行编码转换。例如，空格应该转换为%20。否则 Tracker 无法正确处理 GET 请求。

13.2.6 peer 之间的通信协议

peer 之间的通信协议又称为 peer wire protocal，即 peer 连线协议，它是一个基于 TCP 协议的应用层协议。

为了防止有的 peer 只下载不上传，BitTorrent 协议建议，客户端只给那些向它提供最快下载速度的 4 个 peer 上传数据。简单地说，就是“谁向我提供下载，我也提供数据供它下载；谁不提供数据给我下载，我的数据也不会上传给它”。客户端每隔一定时间，比如 10 秒，重新计算从各个 peer 处下载数据的速度，将下载速度最快的 4 个 peer 解除阻塞，允许这 4 个 peer 从客户端下载数据，同时将其他 peer 阻塞。

一个例外情况是，为了发现下载速度更快的 peer，协议还建议，在任一时刻，客户端保持一个优化非阻塞 peer，即无论该 peer 是否提供数据给客户端下载，客户端都允许该 peer 从客户端这里下载数据。由于客户端向 peer 上传数据，peer 接着也允许客户端从 peer 处下载数据，并且下载速度超过 4 个非阻塞 peer 中的一个。客户端每隔一定的时间，如 30 秒，重新选择优化非阻塞 peer。

当客户端与 peer 建立 TCP 连接后，客户端必须维持的几个状态变量如表 13-8 所示。

表 13-8　客户端必须维持的状态变量

状态变量	含　义
am_chocking	该值若为 1，表明客户端将远程 peer 阻塞。此时如果 peer 发送数据请求给客户端，客户端将不会理会。也就是说，一旦将 peer 阻塞，peer 就无法从客户端下载到数据；该值若为 0，则刚好相反，即表明 peer 未被阻塞，允许 peer 从客户端下载数据
am_interested	该值若为 1，表明客户端对远程的 peer 感兴趣。当 peer 拥有某个 piece，而客户端没有，则客户端对 peer 感兴趣。该值若为 0，则刚好相反，即表明客户端对 peer 不感兴趣，peer 拥有的所有 piece，客户端都拥有
peer_chocking	该值若为 1，表明 peer 将客户端阻塞。此时，客户端无法从 peer 处下载到数据。该值若为 0，表明客户端可以向 peer 发送数据请求，客户端将进行响应
peer_interested	该值若为 1，表明 peer 对客户端感兴趣。也即客户端拥有某个 piece，而 peer 没有。该值若为 0，表明 peer 对客户端不感兴趣

当客户端与 peer 建立 TCP 连接后，客户端将这几个变量的值设置为。

am_chocking　= 1。

am_interested　= 0。

peer_chocking　= 1。

peer_interested = 0。

当客户端对 peer 感兴趣且 peer 未将客户端阻塞时，客户端可以从 peer 处下载数据。当 peer 对客户端感兴趣，且客户端未将 peer 阻塞时，客户端向 peer 上传数据。

我爱自由 5ifreedom.com

除非另有说明，所有的整数型在本协议中被编码为 4 字节值（高位在前低位在后），包括在握手之后所有信息的长度前缀。

客户端与一个 peer 建立 TCP 连接后，首先向 peer 发送握手消息，peer 收到握手消息后回应一个握手消息。

- 握手消息是一个长度固定为 68 字节的消息。消息的格式如下：

```
<pstrlen><pstr><reserved><info_hash><peer_id>
```

消息格式中一些参数的含义如表 13-9 所示。

表 13-9　　握手消息

参　数	含　义
pstrlen	pstr 的长度，该值固定为 19
pstr	BitTorrent 协议的关键字，即“BitTorrent protocol”
reserved	占 8 字节，用于扩展 BT 协议，一般这 8 字节都设置为 0。有些 BT 软件对 BT 协议进行了某些扩展，因此可能看到有些 peer 发来的握手消息这 8 个字节不全为 0，不过不必理会，这不会影响正常的通信
info_hash	与发往 Tracker 的 GET 请求中的 info_hash 为同一个值，长度固定为 20 字节

peer_id 与发往 Tracker 的 GET 请求中的 peer_id 为同一个值，长度固定为 20 字节。一般从 peer_id 可以识别出 BT 软件的类型，例如，某 peer 发来的握手消息中 peer_id 的前 8 个字节为“-AZ2060-”，则可以断定对方使用的是 Azureus；若为“-BCxxxx-”，x 为数字，则对方使用的是 BitComet。

对于除握手消息之外的其他所有消息，其一般的格式为：

```
<length prefix><message ID><payload>
```

length prefix（长度前缀）占 4 个字节，指明 message ID 和 payload 的长度和。message ID（消息编号）占一字节，是一个 10 进制的整数，指明消息的编号。payload（负载），长度未定，是消息的内容。

- keep_alive 消息：<len=0000>

keep_alive 消息的长度固定，为 4 字节，它没有消息编号和负载。如果一段时间内客户端与 peer 没有交换任何消息，则与这个 peer 的连接将被关闭。keep_alive 消息用于维持这个连接，通常如果 2 分钟内没有向 peer 发送任何消息，则发送一个 keep_alive 消息。

- choke 消息：<len=0001><id=0>

choke 消息的长度固定，为 5 字节，消息长度占 4 个字节，消息编号占 1 个字节，没有负载。该消息的功能是，发出该消息的 peer 将接收该消息的 peer 阻塞，暂时不允许其下载自己的数据。

- unchoke 消息：<len=0001><id=1>

unchoke 消息的长度固定，为 5 字节，消息长度占 4 个字节，消息编号占 1 个字节，没有负载。客户端每隔一定的时间，通常为 10 秒，计算一次各个 peer 的下载速度，如果某 peer 被解除阻塞，则发送 unchoke 消息。如果某个 peer 原先是解除阻塞的，而此次被阻塞，则发送 choke 消息。

- interested 消息：<len=0001><id=2>

interested 消息的长度固定，为 5 字节，消息长度占 4 个字节，消息编号占 1 个字节，没有负载。当客户端收到某 peer 的 have 消息时，如果发现 peer 拥有了客户端没有的 piece，则发送 interested 消息告知该 peer，客户端对它感兴趣。

- not interested 消息：<len=0001><id=3>

not interested 消息的长度固定，为 5 字节，消息长度占 4 个字节，消息编号占 1 个字节，没有负载。当客户端下载了某个 piece，如果发现客户端拥有了这个 piece 后，某个 peer 拥有的所有

我爱自由 5ifreedom.com

piece，客户端都拥有，则发送 not interested 消息给该 peer。

- have 消息：<len=0005><id=4><piece index>

have 消息的长度固定，为 9 字节，消息长度占 4 个字节，消息编号占 1 个字节，负载为 4 个字节。负载为一个整数，指明下标为 index 的 piece，peer 已经拥有。每当客户端下载了一个 piece，即将该 piece 的下标作为 have 消息的负载构造 have 消息，并把该消息发送给所有与客户端建立连接的 peer。

- bitfield 消息：<len=0001+X><id=5><bitfield>

bitfield 消息的长度不固定，其中 X 是 bitfield(即位图)的长度。当客户端与 peer 交换握手消息之后，就交换位图。位图中，每个 piece 占一位，若该位的值为 1，则表明已经拥有该 piece；为 0 则表明该 piece 尚未下载。具体而言，假定某共享文件共拥有 801 个 piece，则位图为 101 个字节，位图的第一个字节的最高位指明第一个 piece 是否拥有，位图的第一个字节的第二高位指明第二个 piece 是否拥有，依次类推。对于第 801 个 piece，需要单独一个字节，该字节的最高位指明第 801 个 piece 是否已被下载，其余的 7 位放弃不予使用。

- request 消息：<len=0013><id=6><index><begin><length>

request 消息的长度固定，为 17 个字节，index 是 piece 的索引，begin 是 piece 内的偏移，length 是请求 peer 发送的数据的长度。当客户端收到某个 peer 发来的 unchoke 消息后，即构造 request 消息，向该 peer 发送数据请求。前面提到，peer 之间交换数据是以 slice（长度为 16KB 的块）为单位的，因此 request 消息中 length 的值一般为 16K。对于一个 256KB 的 piece，客户端分 16 次下载，每次下载一个 16K 的 slice。

- piece 消息：<len=0009+X><id=7><index><begin><block>

piece 消息是另外一个长度不固定的消息，长度前缀中的 9 是 id、index、begin 的长度总和，index 和 begin 固定为 4 字节，X 为 block 的长度，一般为 16K。因此对于 piece 消息，长度前缀加上 id 通常为 00 00 40 09 07。当客户端收到某个 peer 的 request 消息后，如果判定当前未将该 peer 阻塞，且 peer 请求的 slice，客户端已经下载，则发送 piece 消息将文件数据上传给该 peer。

- cancel 消息：<len=0013><id<=8><index><begin><length>

cancel 消息的长度固定，为 17 个字节，len、index、begin、length 都占 4 字节。它与 request 消息对应，作用刚好相反，用于取消对某个 slice 的数据请求。如果客户端发现，某个 piece 中的 slice，客户端已经下载，而客户端又向其他 peer 发送了对该 slice 的请求，则向该 peer 发送 cancel 消息，以取消对该 slice 的请求。事实上，如果算法设计合理，基本不用发送 cancel 消息，只在某些特殊的情况下才需要发送 cancel 消息。

- port 消息：<len=0003><id=9><listen-port>

port 消息的长度固定，为 7 字节，其中 listen-port 占两个字节。该消息只在支持 DHT 的客户端中才会使用，用于指明 DHT 监听的端口号，一般不必理会，收到该消息时，直接丢弃即可。

13.2.7 关键算法和策略

1. 流水线作业

BT 协议作为一种构建在 TCP 协议上的应用层协议，可以通过流水线作业来提高数据传输的效率。具体而言，当客户端向 peer 发送数据请求时（即发送 request 消息），一次请求多个 slice（即在一个数据包中发送多个 request 消息请求多个 slice）。假如客户端一次只发送一个 slice 请求，则 peer 给客户端发送完一个 slice 的数据后就进入等待，等待客户端发送新的数据请求。如果一次发

我爱自由 5ifreedom.com

送多个 slice 请求，则 peer 发送完一个 slice 后接着发送下一个 slice，从而避免了等待，提高了数据传输的效率。事实上，HTTP 协议的 1.1 版本就广泛地使用了流水线作业的思想，大大地提高了浏览器和 Web 服务器之间的传输效率。

2. 片断选择算法

一个良好的片断选择策略对于提高下载速度至关重要，对于提高整个文件共享系统的性能也有重要影响。

片断选择的第一个策略是，一旦向某个 peer 发送对某个 piece 中的 slice 的请求后，则该 piece 中的其他 slice 也从该 peer 处下载，这样可以尽快地下载到一个完整的 piece。因为某个 peer 拥有某个 piece 中的一个 slice，则它必定拥有该 piece 的其他 slice，并且如果 peer 愿意发送一个 slice 给客户端，它也应该愿意发送 piece 中的其他 slice 给客户端。该策略也被称为严格的优先级。

片断选择的第二个策略是，最少优先。即某个 piece 在所有 peer 中的拥有率最低，则优先下载该 piece。这么做的优点是，第一，可以防止拥有这个 piece 的 peer 突然离开，导致某个 piece 的缺失，从而当前任何一个参与下载的 peer 都不能下载到一份完整的文件；第二，如果下载了某些拥有率较低的 piece，则其他很多 peer 会向客户端请求数据，而要想从客户端下载到数据，那些 peer 就要提供数据给客户端下载，这样对于提高客户端的下载速度也是有帮助的。对于这个共享系统而言，优先下载拥有率较低的 piece 可以使得整个系统提高每个 piece 的拥有度，整个系统会趋向于最优。如果所有 peer 优先下载拥有率较高的 piece，会使某些 piece 的拥有率进一步降低，而拥有这些低拥有率 piece 的 peer 一旦离开共享系统，则整个文件会越来越不完整，最后导致许多 peer 不能下载到一个完整的文件拷贝。

片断选择的第三个策略是，随机选择第一个要下载的 piece。开始下载时，不能采用最少优先策略。原因在于，采用最少优先策略，如果某个 piece 的拥有率很低，那么下载到这个 piece 就相对较难。如果随机选择一个 piece，那么更容易下载到该 piece，一旦客户端下载到一个完整的 piece，就可以提供给其他 peer 下载，而由于客户端向其他 peer 上传数据，会导致其他 peer 对客户端解除阻塞，从而有利于在起始阶段获得较高的下载速度。当然在下载到一些 piece 后，客户端应该采用最少优先策略来下载数据，这虽然会导致客户端的下载速度在短期内有所下降，但随后下载速度会有较大提高。

片断选择的第四个策略是，最后阶段模式。有时，从一个传输速度很慢的 peer 处下载一个 piece 会花费很长时间，在下载的过程中这不是什么大问题。但在下载接近完成时，如果发生这种情况，会导致客户端迟迟不能下载完成。为了解决这个问题，在最后阶段，客户端向所有 peer 发送对这个 piece 的某些 slice 的请求，一旦收到某个 peer 发来的 slice，则向其他 peer 发送 cancel 消息，只从当前这个 peer 处下载。

3. 阻塞算法

BT 并不集中分配资源，每个 peer 有责任尽可能地提高自己的下载速度。peer 从它可以连接的 peer 下载文件，并根据对方提供的下载速率给予同等的上传回报，对于合作者，提供上传服务，对于不合作的，就阻塞对方。阻塞是一种临时拒绝上传的策略，虽然上传停止了，但是下载仍然继续。在解除阻塞时，连接并不需要重新建立。因为阻塞过程中只是拒绝传输 piece 消息，其他消息，如 have 消息，interested 消息仍可以传输。阻塞算法虽然不是 BT 协议一部分，但是它对提高性能是必要的。

每个客户端一直与固定数量的 peer 保持疏通（通常是 4 个），那么以什么方式来决定是否保持与某个 peer 疏通呢？通常的做法是，严格地根据当前的下载速度来决定哪些 peer 应该保持疏通。但计算当前下载速度是个大难题。当前的实现通常是计算最近 10 秒从每个 peer 处下载数据的速

我爱自由 5ifreedom.com

度。以 10 秒为间隔重新选择保持疏通（即解除阻塞）的 peer，是为了避免频繁地阻塞和解阻塞，造成资源的浪费。实践表明，10 秒足以使下载速度达到最大。

如果只是简单地为提供最高下载速率的 4 个 peer 提供上载服务，那么就没有办法发现那些空闲的连接是否有更好的下载速度。为了解决这个问题，在任何时候，每个 peer 都拥有一个称为“optimistic unchoking（优化非阻塞）”peer，这个连接总是保持疏通状态，而不管它的下载速率是多少。每隔 30 秒，重新选择一个 peer 作为优化非阻塞 peer。30 秒足以让该 peer 的上载能力达到最大。

一旦某个 peer 完成了下载，它不能再通过下载速率（因为下载速率已经为 0 了）来决定为哪些 peer 提供上载了。目前采用的解决办法是，优先选择那些从它这里得到更好下载速率的 peer，保持与它们疏通。这样做的理由是尽可能的利用上载带宽。一旦某个 peer 完成了下载，那么它也就成为了种子。种子拥有一份完整的文件拷贝，并提供给其他 peer 下载。为了整个系统的性能，每个 peer 在完成下载后应该作为种子存在一段时间，作为对整个系统的回报。原始种子，即最初提供文件进行共享、并制作生成了种子文件，并把种子文件发布到 Web 服务器的种子，它至少应该存在到系统中生成另外一个种子时才能离开，否则当前参与下载的所有 peer 都不能获得一份完整的文件拷贝。

本节对 BT 协议的解释和分析参考了“BT 协议规范”(Bittorrent Protocol Specification)和 BT 协议设计者所著的“BT 性能卓越的原因”(Incentives Build Robustness in BitTorrent)一文。

13.3 系统结构设计

整个系统的模块结构如图 13-1 所示。

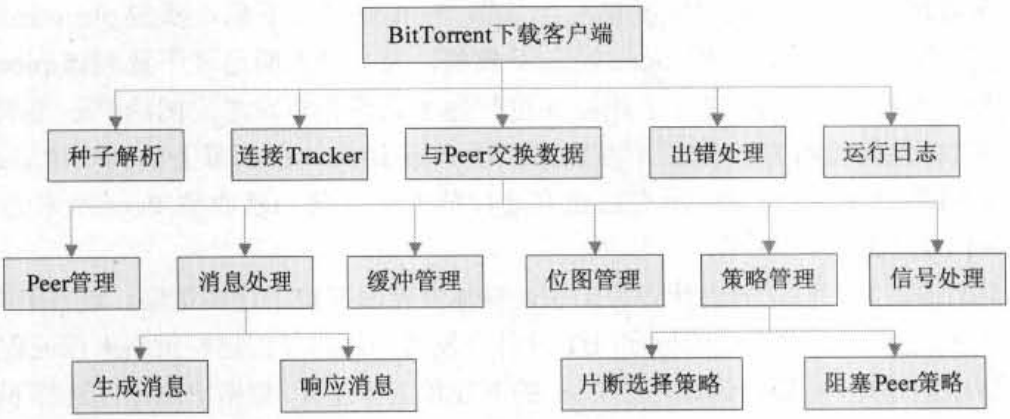


图 13-1　系统模块结构图

整个系统各个模块的功能如下。

（1）种子解析：负责解析种子文件，从中获取 Tracker 服务器的地址，待下载文件的文件名和长度，piece 长度，各个 piece 的 hash 值。

（2）连接 Tracker：根据 HTTP 协议构造获取 Peer 地址的请求，与 Tracker 建立连接，解析 Tracker 的回应消息，从而获取各个 Peer 的 IP 地址和端口号。

（3）与 Peer 交换数据：根据 Peer 的 IP 地址和端口号连接 Peer、从 Peer 处下载数据并将已下载的数据上传给 Peer。

（4）出错处理：定义整个系统可能出现的错误类型，并对错误进行处理。

我爱自由 5ifreedom.com

（5）运行日志：记录程序运行的日志，并保存到文件中以备查看和分析。

模块“与 Peer 交换数据”是本系统的核心和主要构成部分，它又可以划分成如下几个子模块。

（1）Peer 管理：系统为每一个已建立 TCP 连接的 Peer 构造一个 Peer 结构体。该结构体的主要成员有：Peer 的 IP 地址和端口号、与该 Peer 进行通信的套接字、该 Peer 的 id、当前所处的状态、发送缓冲区、接收缓冲区、数据请求队列、数据被请求队列、从该 Peer 处已下载的数据量和向该 Peer 上传的数据量、下载速度和上传速度。本模块负责管理 Peer 链表，添加和删除 Peer 结点。

（2）消息处理：Peer 与 Peer 之间以发送和接收消息的方式进行通信。本模块负责根据当前的状态生成并发送消息，接收并处理消息。BitTorrent 协议共定义了 12 种消息，其中对下载和上传数据最重要的是 request 消息和 piece 消息。request 消息向 Peer 发送数据请求，指明请求的是哪个 piece 的哪个 slice。Peer 接收到 request 消息后根据当前的状态，决定是否发送数据给对方。如果允许发送，则构造 piece 消息，数据被封装在该消息中。每当下载完一个正确的 piece 时，就向所有 Peer 发送 have 消息通告已获得该 piece，其他 Peer 如果没有该 piece 就可以向 Peer 发送数据请求，每次请求都是以 slice 为单位。

（3）缓冲管理：如果下载完一个 piece 就立即写入硬盘，这样会导致频繁读写硬盘，既影响速度（读写磁盘要花费较多的时间），又不利于保护硬盘（频繁读写磁盘会使硬盘寿命缩短）。为了解决这个问题，几乎所有的 BT 软件都在程序中增加了一个缓冲管理模块。将下载到的数据先缓存起来，等到下载到一定量的数据后再集中写入硬盘。Peer 请求一个 slice 的数据时，先将该 slice 所在的整个 piece 读入到缓冲区中，下次 Peer 再请求该 piece 的其他 slice 时，只需在缓冲区中获取，避免了频繁读写硬盘。本模块负责维护一个 16MB 的缓冲区（大小可调），将下载到的数据保存在缓冲区中，并在适当时刻写入硬盘的文件中。

（4）位图管理：BT 协议采用位图指明当前哪些 piece 已经下载，哪些 piece 还没有下载。每个 piece 占一位，值为 0 表示该 piece 还未下载到，为 1 则表明已经下载到该 piece。本模块负责管理位图，客户端与 Peer 建立了连接并进行握手之后，即发送位图给 Peer 告知已下载到哪些 piece，同时也接收对方的位图并将其保存在 Peer 结构体中。每下载到一个 piece 就更新自己的位图，并发送 have 消息给所有已建立连接的 Peer。每当接收到 Peer 发来的 have 消息就更新该 Peer 的位图。

（5）策略管理：BT 协议的设计者为了保证整体性能而制定了许多策略，这些策略虽然没有写入 BT 协议，但已经成为事实上的标准，BT 软件开发者一般都使用这些策略来保证程序的性能。本部分负责策略的管理，主要是计算各个 Peer 的下载和上传速度，根据下载速度选择非阻塞 Peer，采用随机算法选择优化非阻塞 Peer，以及实现片断选择策略。

（6）信号处理：在运行过程中，程序可能会接收到一些信号，如 SIGINT、SIGTERM，这些信号的默认动作是立即终止程序。这并不是所期望的。在程序终止前需要作一些处理，如释放动态申请的内存、关闭文件描述符、关闭套接字。

13.4 各个模块的设计和实现

由于篇幅的限制，对于某些不是很关键的函数，只给出了函数原型，具体实现代码没有列出。若要完整地阅读整个程序，请参考本书所附的源代码光盘。

我爱自由 5ifreedom.com

13.4.1 种子解析模块的设计和实现

解析种子文件主要在 parse_metafile.h 和 parse_metafile.c 中完成。parse_metafile.h 文件的内容为：

```
// parse_metafile.h
#ifndef PARSE_METAFILE
#define PARSE_METAFILE

// 保存从种子文件中获取的 tracker 的 URL
typedef struct _Announce_list {
    char    announce[128];
    struct  _Announce_list  *next;
} Announce_list;

// 保存各个待下载文件的路径和长度
typedef struct _Files {
    char    path[256];
    long    length;
    struct  _Files  *next;
} Files;

int read_metafile(char *metafile_name);                  // 读取种子文件
int find_keyword(char *keyword,long *position);          // 在种子文件中查找某个关键词
int read_announce_list();                                // 获取各个 tracker 服务器的地址
int add_an_announce(char* url);                          // 向 tracker 列表添加一个 URL

int get_piece_length();            // 获取每个 piece 的长度,一般为 256KB
int get_pieces();                  // 读取各个 piece 的哈希值

int is_multi_files();              // 判断下载的是单个文件还是多个文件
int get_file_name();               // 获取文件名，对于多文件，获取的是目录名
int get_file_length();       //     获取待下载文件的总长度
int get_files_length_path();       // 获取文件的路径和长度，对多文件种子有效

int get_info_hash();               // 由 info 关键词对应的值计算 info_hash
int get_peer_id();                 // 生成 peer_id,每个 peer 都有一个 20 字节的 peer_id

void release_memory_in_parse_metafile();// 释放 parse_metafile.c 中动态分配的内存
int  parse_metafile(char *metafile);    // 调用本文件中定义的函数,完成解析种子文件

#endif
```

以下是 parse_metafile.c 文件的头部，主要是包含了一些头文件和定义一些全局变量，各个函数的定义将在后面列出。

```
// parse_metafile.c
#include <stdio.h>
#include <ctype.h>
#include <malloc.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "parse_metafile.h"
#include "sha1.h"

char      *metafile_content = NULL;    // 保存种子文件的内容
long      filesize;                    // 种子文件的长度

int        piece_length = 0;           // 每个 piece 的长度,通常为 256KB 即 262144 字节
char      *pieces = NULL;              // 保存每个 pieces 的哈希值,每个哈希值为 20 字节
int        pieces_length = 0;          // 缓冲区 pieces 的长度

int       multi_file = 0;          // 指明是单文件还是多文件
char      *file_name = NULL;           // 对于单文件,存放文件名;对于多文件,存放目录名
long long file_length = 0;             // 存放待下载文件的总长度
Files     *files_head = NULL;          // 只对多文件种子有效,存放各个文件的路径和长度
```

我爱自由 5ifreedom.com

```
unsigned char info_hash[20];          // 保存 info_hash 的值,连接 tracker 和 peer 时使用
unsigned char peer_id[20];            // 保存 peer_id 的值,连接 peer 时使用

Announce_list *announce_list_head = NULL; // 用于保存所有 tracker 服务器的 URL
```

下面对解析种子文件中用到函数功能解释如下。

- int read_metafile(char *metafile_name)

功能：解析种子文件。

参数：metafile_name 是种子文件名。

返回：处理成功返回 0，否则返回-1。

附注：将种子文件的内容读入到全局变量 metafile_content 所指向的缓冲区中以方便处理。该函数的实现代码为：

```
int read_metafile(char *metafile_name)
{
    long  i;

    // 以二进制、只读的方式打开文件
    FILE *fp = fopen(metafile_name,"rb");
    if(fp == NULL) {
        printf("%s:%d can not open file\n",__FILE__,__LINE__);
        return -1;
    }

    // 获取种子文件的长度，filesize 为全局变量，在 parse_metafile.c 头部定义
    fseek(fp,0,SEEK_END);
    filesize = ftell(fp);
    if(filesize == -1) {
        printf("%s:%d fseek failed\n",__FILE__,__LINE__);
        return -1;
    }
    metafile_content = (char *)malloc(filesize+1);
    if(metafile_content == NULL) {
        printf("%s:%d malloc failed\n",__FILE__,__LINE__);
        return -1;
    }

    // 读取种子文件的内容到 metafile_content 缓冲区中
    fseek(fp,0,SEEK_SET);
    for(i = 0; i < filesize; i++)
        metafile_content[i] = fgetc(fp);
    metafile_content[i] = '\0';

    fclose(fp);

#ifdef DEBUG
    printf("metafile size is: %ld\n",filesize);
#endif
    return 0;
}
```

函数代码说明。

（1）编译器预定义的宏__FILE__和__LINE__在程序中可以直接使用。__FILE__代表该宏所在的源文件的文件名，在源文件 parse_metafile.c 中该宏的值等于“parse_metafile.c”，宏__LINE__的值为__LINE__所在行的行号。

（2）种子文件必须以二进制的方式打开，否则如果以字符方式打开可能无法读取整个文件的内容。无法读取的原因在于 piece 的 hash 值中可能含有字符 0x00，若文件以字符形式打开，遇到该字符，库函数就认为文件已经结束。

我爱自由 5ifreedom.com

（3）增加“#ifdef DEBUG　#endif ”，主要是为了方便调试。如果在 parse_metafile.c 文件的头部增加宏定义语句“#define DEBUG”，则程序运行时将执行“#ifdef DEBUG”和“#endif”之间的语句。在软件开发阶段，可以使用“#define DEBUG”来打印和查看某些关键的值，开发完毕，去掉该宏，则打印语句不会执行。

- int find_keyword(char *keyword,long *position)

功能：从种子文件中查找某个关键字。

参数：keyword 为要查找的关键字，position 用于返回关键字第一个字符所在的下标。

返回：成功执行并找到关键字返回 1，未找到返回 0，执行失败返回-1。函数实现代码如下所示：

```
int find_keyword(char *keyword,long *position)
{
    long i;

    *position = -1;
    if(keyword == NULL)  return 0;

    for(i = 0; i < filesize-strlen(keyword); i++) {
        if( memcmp(&metafile_content[i], keyword, strlen(keyword)) == 0 ) {
            *position = i;
            return 1;
        }
    }

    return 0;
}
```

函数代码说明。

该函数在种子文件解析模块的源文件 parse_metafile.c 中被频繁使用，用于查找某些关键字。例如，关键字“8:announce”和“13:announce-list”之后都是 Tracker 服务器的地址，找到该关键字后，便可以获取 Tracker 的地址。

- read_announce_list()

功能：获取 Tracker 地址，并将获取的地址保存到全局变量 announce_list_head 指向的链表中。该函数实现代码如下：

```
int read_announce_list()
{
    Announce_list  *node = NULL;
    Announce_list  *p    = NULL;
    int            len   = 0;
    long           i;

    if( find_keyword("13:announce-list",&i) == 0 ) {
        if( find_keyword("8:announce",&i) == 1 ) {
            i = i + strlen("8:announce");
            while( isdigit(metafile_content[i]) ) {
                len = len * 10 + (metafile_content[i] - '0');
                i++;
            }
            i++;  // 跳过 ':'

            node = (Announce_list *)malloc(sizeof(Announce_list));
            strncpy(node->announce,&metafile_content[i],len);
            node->announce[len] = '\0';
            node->next = NULL;
            announce_list_head = node;
        }
    }
    else {  // 如果有 13:announce-list 关键词就不用处理 8:announce 关键词
        i = i + strlen("13:announce-list");
```

我爱自由 5ifreedom.com

```
        i++;        // 跳过 'l'
        while(metafile_content[i] != 'e') {
            i++;    // 跳过 'l'
            while( isdigit(metafile_content[i]) ) {
                len = len * 10 + (metafile_content[i] - '0');
                i++;
            }
            if( metafile_content[i] == ':' )  i++;
            else  return -1;

            // 只处理以 http 开头的 tracker 地址,不处理以 udp 开头的地址
            if( memcmp(&metafile_content[i],"http",4) == 0 ) {
                node = (Announce_list *)malloc(sizeof(Announce_list));
                strncpy(node->announce,&metafile_content[i],len);
                node->announce[len] = '\0';
                node->next = NULL;

                if(announce_list_head == NULL)
                    announce_list_head = node;
                else {
                    p = announce_list_head;
                    while( p->next != NULL) p = p->next; // 使 p 指针指向最后个结点
                    p->next = node; // node 成为 tracker 列表的最后一个结点
                }
            }

            i = i + len;
            len = 0;
            i++;    // 跳过 'e'
            if(i >= filesize)  return -1;
        } // while 循环结束
    }

#ifdef DEBUG
    p = announce_list_head;
    while(p != NULL) {
        printf("%s\n",p->announce);
        p = p->next;
    }
#endif
    return 0;
}
```

程序说明。

（1）下面是某种子文件开头的一部分，请对照它来理解 read_announce_list 函数

```
d8:announce32:http://tk.greedland.net/announce13:announce-listll32:http://tk.greedland.net/announceel33:http://tk2.greedland.net/announceee...
```

第一个字符‘d’是 B 编码中字典的起始符，接着是关键字“8:announce”，该关键字是一个长度为 8 的字符串，其对应的值为长度为 32 的字符串“32:http://tk.greedland.net/announce”，它是一个 Tracker 服务器的 URL，接着是关键字“13:announce-list”，该关键字对应的值是一个列表，因为关键字“13:announce-list”之后的第一个字符为列表的起始字符‘l’，该列表中含有两个元素，这两个元素的类型也都是列表。

如果有关键字“13:announce-list”就不用处理关键字“8:announce”的原因在于，前者对应的值中必定包含后者对应的值。

（2）“#ifdef DEBUG”和“#endif”之间的语句用于打印各个 Tracker 的 URL。

- int add_an_announce(char *url)

功能：连接某些 Tracker 时会返回一个重定向的 URL，需要连接该 URL 才能获取 Peer。函数实现代码如下：

我爱自由 5ifreedom.com

```
int add_an_announce(char *url)
{
Announce_list *p = announce_list_head, *q;

    // 若参数指定的 URL 在 Tracker 列表中已存在,则无需添加
    while(p != NULL) {
        if(strcmp(p->announce,url) == 0)  break;
        p = p->next;
    }
    if(p != NULL)  return 0;

    q = (Announce_list *)malloc(sizeof(Announce_list));
    strcpy(q->announce,url);
    q->next = NULL;

    p = announce_list_head;
    if(p == NULL)  { announce_list_head = q; return 1; }
    while(p->next != NULL)  p = p->next;
    p->next = q;
    return 1;
}
```

- int is_multi_files()

功能：判断是下载多个文件还是单文件，若含有关键字“5:files”则说明下载的是多个文件。函数实现的代码如下：

```
int is_multi_files()
{
    long i;

    if( find_keyword("5:files",&i) == 1 ) {
        multi_file = 1;
        return 1;
    }

#ifdef DEBUG
    printf("is_multi_files:%d\n",multi_file);
#endif

    return 0;
}
```

- int get_piece_length()

功能：获取 piece 的长度。函数实现的代码如下：

```
int get_piece_length()
{
    long i;

    if( find_keyword("12:piece length",&i) == 1 ) {
        i = i + strlen("12:piece length");  // 跳过 "12:piece length"
        i++;  // 跳过 'i'
        while(metafile_content[i] != 'e') {
            piece_length = piece_length * 10 + (metafile_content[i] - '0');
            i++;
        }
    } else {
        return -1;
    }

#ifdef DEBUG
    printf("piece length:%d\n",piece_length);
#endif
    return 0;
}
```

程序说明。

以下是某种子文件的一部分：12:piece lengthi262144e6:pieces16900:...

我爱自由 5ifreedom.com

从中可以看到，关键字“12:piece length”后面跟一个 B 编码的整型数（以 i 作为起始字符，以 e 作为终结字符）。262144（256K），说明每个 piece 的长度都是 256KB（最后一个 piece 除外）。接着是关键字“6:pieces”，它对应的值是一个 B 编码的字符串，存放各个 piece 的 hash 值，16900 是字符串的长度，该字符串长度除以 20 即为 piece 数，因为每个 piece 的 hash 值为固定的 20 字节。

- get_pieces()

功能：获取每个 piece 的 hash 值，并保存到 pieces 所指向的缓冲区中。函数实现的代码如下：

```
int get_pieces()
{
    long i;

    if( find_keyword("6:pieces", &i) == 1 ) {
        i = i + 8;          // 跳过 "6:pieces"
        while(metafile_content[i] != ':') {
            pieces_length = pieces_length * 10 + (metafile_content[i] - '0');
            i++;
        }
        i++;                        // 跳过 ':'
        pieces = (char *)malloc(pieces_length+1);
        memcpy(pieces,&metafile_content[i],pieces_length);
        pieces[pieces_length] = '\0';
    } else {
        return -1;
    }
#ifdef DEBUG
    printf("get_pieces ok\n");
#endif
    return 0;
}
```

- get_file_name()

功能：获取待下载的文件的文件名，如果下载的是多个文件，则获取的是目录名。函数实现的代码如下：

```
int get_file_name()
{
    long  i;
    int   count = 0;

    if( find_keyword("4:name", &i) == 1 ) {
        i = i + 6;  // 跳过 "4:name"
        while(metafile_content[i] != ':') {
            count = count * 10 + (metafile_content[i] - '0');
            i++;
        }
        i++;        // 跳过 ':'
        file_name = (char *)malloc(count+1);
        memcpy(file_name,&metafile_content[i],count);
        file_name[count] = '\0';
    } else {
        return -1;
    }

#ifdef DEBUG
    printf("file_name:%s\n",file_name);
#endif
    return 0;
}
```

程序说明。

以下是一个完整的较为简单的种子文件：

```
d8:announce32:http://tk.greedland.net/announce13:announce-listll32:http://tk.greedland.n
et/annou
```

我爱自由 5ifreedom.com

```
    nceel33:http://tk2.greedland.net/announceee13:creation
datei1187968874e4:infod6:lengthi119861
    306e4:name31:[ymer][naruto][246][jp_cn].rmvb10:name.utf-831:[ymer][naruto][246][jp_cn].r
mv
    b12:piece lengthi262144e6:pieces9160:...ee
```

关键字“13:creation date”之前的部分已经在介绍 read_announce_list 函数时分析过了，此处不再赘述。关键字“13:creation date”及其对应的值“i1187968874e”，它指明了创建种子文件的时间。我们注意到时间是一个整数，它是自 1970 年 1 月 1 日到种子文件创建时所经过的秒数，Linux 中有专门的库函数处理这种表示类型的时间。

关键字“4:info”对应的值是一个字典，因为该关键字之后的第一个字符是 B 编码中字典的起始符‘d’，与该起始符对应的终止符是文件末尾的倒数第二个‘e’。计算 info_hash 时，就是以关键字“4:info”对应的值作为输入，计算其 hash 值，将得到的值作为 info_hash。文件末尾最后一个字符‘e’与文件开头的‘d’对应，因此整个种子文件就是一个 B 编码的字典。

关键字“6:length”对应的值是待下载文件的长度，以字节为单位，可以大致地确定待下载文件的长度为 119MB。

关键字“4:name”对应的值为待下载的文件的文件名，在这个种子文件中没有关键字“5:files”说明待下载的是单文件。

关键字“10:name.utf-8”对应的值也是待下载文件的文件名，只不过以 UTF-8 的形式表示，UTF-8 的形式可以表示宽字符，即中文、日文、朝鲜文等字符。

- int get_file_length()

功能：获取待下载文件的长度。函数实现的代码如下：

```
int get_file_length()
{
    long i;

    if(is_multi_files() == 1)  {
        if(files_head == NULL)  get_files_length_path();
        Files *p = files_head;
        while(p != NULL) { file_length += p->length; p = p->next; }
    } else {
        if( find_keyword("6:length",&i) == 1 ) {
            i = i + 8;  // 跳过 "6:length"
            i++;        // 跳过 'i'
            while(metafile_content[i] != 'e') {
                file_length = file_length * 10 + (metafile_content[i] - '0');
                i++;
            }
        }
    }

#ifdef DEBUG
    printf("file_length:%lld\n",file_length);
#endif
    return 0;
}
```

- get_files_length_path()

功能：对于多文件，获取各个文件的路径以及长度。函数实现的代码如下：

```
int get_files_length_path()
{
    long   i;
    int    length;
    int    count;
    Files  *node  = NULL;
    Files  *p     = NULL;
```

我爱自由 5ifreedom.com

```
    if(is_multi_files() != 1) {
        return 0;
    }

    for(i = 0; i < filesize-8; i++) {
        if( memcmp(&metafile_content[i],"6:length",8) == 0 )
        {
            i = i + 8;  // 跳过 "6:length"
            i++;        // 跳过 'i'
            length = 0;
            while(metafile_content[i] != 'e') {
                length = length * 10 + (metafile_content[i] - '0');
                i++;
            }
            node = (Files *)malloc(sizeof(Files));
            node->length = length;
            node->next = NULL;
            if(files_head == NULL)
                files_head = node;
            else {
                p = files_head;
                while(p->next != NULL) p = p->next;
                p->next = node;
            }
        }
        if( memcmp(&metafile_content[i],"4:path",6) == 0 )
        {
            i = i + 6;  // 跳过 "4:path"
            i++;        // 跳过 'l'
            count = 0;
            while(metafile_content[i] != ':') {
                count = count * 10 + (metafile_content[i] - '0');
                i++;
            }
            i++;        // 跳过 ':'
            p = files_head;
            while(p->next != NULL) p = p->next;
            memcpy(p->path,&metafile_content[i],count);
            *(p->path + count) = '\0';
        }
    }

    return 0;
}
```

程序说明。

图 13-2 是一个多文件种子的一部分，可以参照图 13-2 理解 get_files_length_path 函数。

多文件种子的关键字“5:files”对应的值是比较复杂的。关键字“5:files”说明这是一个多文件种子，它对应的值是一个列表，列表的每个元素是字典，每个字典代表一个待下载文件。

```
5:files1d6:lengthil27025815e4:path134:[BBsee 出品][军情观察室
08.22].rmvbeed6:lengthi76e4:path142:综艺 美剧 篮球 足球 尽在
迅视 XunTv.Net.urleee4:name29:[BBsee 出品][军情观察室 08.22
12:piece lengthi262144e6:pieces9700:..
```

图 13-2　多文件种子示例

“5:filesl”及字符‘d’之后，有一个关键字“6:length”及其值“i127025815e”，然后是关键字“4:path”，其值为一个列表“l34:[BBsee 出品][军情观察室 08.22].rmvbe”，“rmvbee”中最后一个‘e’与字符‘d’对应。

我爱自由 5ifreedom.com

然后“d6:lengthi76e4:pathl42:综艺 美剧 篮球 足球 尽在迅视 XunTv.Net.urlee”又是一个字典。“urleee”中最后一个‘e’与“5:files”后的‘l’构成一个列表。“4:name”所跟的是目录名，然后是“12:piece length”关键字，“6:pieces”关键字。

从中可以总结出：有一个目录名“[BBsee 出品][军情观察室 08.22]”，其中存放了两个文件“[BBsee 出品][军情观察室 08.22].rmvb”和“综艺 美剧 篮球 足球 尽在迅视 XunTv. Net.url”，长度分别为 127025815 字节和 76 字节。

- int get_info_hash ()

功能：计算 info_hash 的值。函数实现代码如下：

```
int get_info_hash()
{
    int  push_pop = 0;
    long  i, begin, end;

    if(metafile_content == NULL)  return -1;
    // begin 的值表示的是关键字"4:info"对应值的起始下标
    if( find_keyword("4:info",&i) == 1 )  begin = i+6;
    else  return -1;

    i = i + 6;          // 跳过 "4:info"
    for(; i < filesize; )
        if(metafile_content[i] == 'd') {
            push_pop++;
            i++;
        } else if(metafile_content[i] == 'l') {
            push_pop++;
            i++;
        } else if(metafile_content[i] == 'i') {
            i++;  // 跳过‘i’
            if(i == filesize)  return -1;
            while(metafile_content[i] != 'e') {
                if((i+1) == filesize)  return -1;
                else i++;
            }
            i++;  // 跳过‘e’
} else if((metafile_content[i] >= '0') && (metafile_content[i] <= '9')) {
            int number = 0;
            while((metafile_content[i] >= '0') && (metafile_content[i] <= '9')) {
                number = number * 10 + metafile_content[i] - '0';
                i++;
            }
            i++;  // 跳过‘:’
            i = i + number;
} else if(metafile_content[i] == 'e') {
            push_pop--;
            if(push_pop == 0) { end = i; break; }
            else  i++;
        } else {
            return -1;
        }
    if(i == filesize)  return -1;

    SHA1_CTX context;
    SHA1Init(&context);
    SHA1Update(&context, &metafile_content[begin], end-begin+1);
    SHA1Final(info_hash, &context);

#ifdef DEBUG
    printf("info_hash:");
    for(i = 0; i < 20; i++)
        printf("%.2x ",info_hash[i]);
    printf("\n");
#endif
    return 0;
}
```

我爱自由 5ifreedom.com

程序说明。

（1）在种子文件解析模块，由种子文件的内容计算 info_hash 的值是比较复杂的。前面已经提到，由关键字“4:info”对应的值来计算 info_hash，该关键字对应的值是一个 B 编码的字典，问题的关键在于找到与“4:info”之后的‘d’对应的‘e’。

get_info_hash 函数中找到所需要的‘e’的思路是：在“4:info”之后，每当遇到字典的起始符‘d’，则将 push_pop 的值加 1(push_pop 初始值为 0)，遇到列表的起始符‘l’也作相同处理；遇到整数的起始符‘i’则一直扫描直到找到与之对应的终结符‘e’；遇到一个 0～9 的数字说明接下来是一个字符串，跳过该字符串继续扫描；遇到‘e’则将 push_pop 值减 1，如果减 1 后，push_pop 值为 0，说明已经找到了与‘d’匹配的‘e’。其思路类似于使用数据结构中的“栈”进行括号匹配操作。

（2）以“SHA1”开头的变量和函数用于计算一段文本的 hash 值，这些变量和函数的定义在 sha1.h 和 sha1.c 文件中，hash 值的计算原理不必深究。计算 hash 值的这段代码的功能是以 metafile_content[begin]～metafile_content[end]这 end-begin+1 个字符作为输入，计算其 hash 值，并把结果保存到 info_hash 所指向的数组中。

- int get_peer_id()

功能：生成一个惟一的 peer id。函数实现代码如下：

```
int get_peer_id()
{
    // 设置产生随机数的种子
    srand(time(NULL));
    // 使用 rand 函数生成一个随机数,并使用该随机数来构造 peer_id
// peer_id 前 8 位固定为-TT1000-
    sprintf(peer_id,"-TT1000-%12d",rand());

#ifdef DEBUG
    printf("peer_id:%s\n",peer_id);
#endif
    return 0;
}
```

- void release_memory_in_parse_metafile()

功能：释放动态申请的内存。函数实现代码如下：

```
void release_memory_in_parse_metafile()
{
    Announce_list *p;
    Files         *q;

    if(metafile_content != NULL)      free(metafile_content);
    if(file_name != NULL)         free(file_name);
    if(pieces != NULL)            free(pieces);

    while(announce_list_head != NULL) {
        p = announce_list_head;
        announce_list_head = announce_list_head->next;
        free(p);
    }

    while(files_head != NULL) {
        q = files_head;
        files_head = files_head->next;
        free(q);
    }
}
```

- int parse_metafile(char *metafile)

功能：调用 parse_metafile.c 中定义的函数，完成解析种子文件。该函数由 main.c 调用。

我爱自由 5ifreedom.com

返回：解析成功返回 0，否则返回-1。函数实现代码如下：

```
int parse_metafile(char *metafile)
{
    int ret;

    // 读取种子文件
    ret = read_metafile(metafile);
    if(ret < 0) { printf("%s:%d wrong",__FILE__,__LINE__); return -1; }

    // 从种子文件中获取 tracker 服务器的地址
    ret = read_announce_list();
    if(ret < 0) { printf("%s:%d wrong",__FILE__,__LINE__); return -1; }

    // 判断是否为多文件
    ret = is_multi_files();
    if(ret < 0) { printf("%s:%d wrong",__FILE__,__LINE__); return -1; }

    // 获取每个 piece 的长度，一般为 256KB
    ret = get_piece_length();
    if(ret < 0) { printf("%s:%d wrong",__FILE__,__LINE__); return -1; }

    // 读取各个 piece 的哈希值
    ret = get_pieces();
    if(ret < 0) { printf("%s:%d wrong",__FILE__,__LINE__); return -1; }

    // 获取要下载的文件名，对于多文件的种子，获取的是目录名
    ret = get_file_name();
    if(ret < 0) { printf("%s:%d wrong",__FILE__,__LINE__); return -1; }

    // 对于多文件的种子，获取各个待下载的文件路径和文件长度
    ret = get_files_length_path();
    if(ret < 0) { printf("%s:%d wrong",__FILE__,__LINE__); return -1; }

    // 获取待下载的文件的总长度
    ret = get_file_length();
    if(ret < 0) { printf("%s:%d wrong",__FILE__,__LINE__); return -1; }

    // 获得 info_hash，生成 peer_id
    ret = get_info_hash();
    if(ret < 0) { printf("%s:%d wrong",__FILE__,__LINE__); return -1; }
    ret = get_peer_id();
    if(ret < 0) { printf("%s:%d wrong",__FILE__,__LINE__); return -1; }

    return 0;
}
```

13.4.2 位图管理模块的设计和实现

对位图的操作主要在 bitfield.h 和 bitfield.c 中，负责创建位图，设置和获取位图某一位的值，保存位图等。

```
// bitfield.h
#ifndef BITFIELD_H
#define BITFIELD_H

typedef struct _Bitmap {
    unsigned char  *bitfield;          // 保存位图
    int            bitfield_length;    // 位图所占的总字节数
    int            valid_length;       // 位图有效的总位数，每一位代表一个 piece
} Bitmap;

int  create_bitfield();                          // 创建位图，分配内存并进行初始化
int  get_bit_value(Bitmap *bitmap,int index);    // 获取某一位的值
int  set_bit_value(Bitmap *bitmap,int index, unsigned char value);
                                                 // 设置某一位的值
int  all_zero(Bitmap *bitmap);                   // 全部清零
```

我爱自由 5ifreedom.com

```
int  all_set(Bitmap *bitmap);                          // 全部设置为 1
void release_memory_in_bitfield();                     // 释放 bitfield.c 中动态分配的内存
int  print_bitfield(Bitmap *bitmap);                   // 打印位图值,用于调试

int  restore_bitmap();  // 将位图存储到文件中
                        // 在下次下载时,先读取该文件获取已经下载的进度
int  is_interested(Bitmap *dst,Bitmap *src);    // 拥有位图 src 的 peer 是否对拥有
                                                // dst 位图的 peer 感兴趣
int  get_download_piece_num();                         // 获取当前已下载到的总 piece 数

#endif
```

程序说明。

（1）结构体 Bitmap 中，bitfield_length 为指针 bitfield 所指向的内存的长度（以字节为单位），而 valid_length 为位图的有效位数。例如，某位图占 100 字节，而有效位数位 795，则位图最后一个字节的最后 5 位(100 × 8−795)是无效的。

（2）函数 is_interested 用于判断两个 peer 是否感兴趣，如果 peer1 拥有某个 piece，而 peer2 没有，则 peer2 对 peer1 感兴趣，希望从 peer1 处下载它没有的 piece。

（3）函数 get_download_piece_num 用于获得已下载的 piece 数，其方法是统计结构体 Bitmap 的 bitfield 成员所指向的内存中值为 1 的位数。

文件 bitfield.c 的头部包含的文件如下：

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <malloc.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include "parse_metafile.h"
#include "bitfield.h"

extern int    pieces_length;
extern char   *file_name;

Bitmap        *bitmap = NULL;          // 指向位图
int           download_piece_num = 0;      // 当前已下载的 piece 数
```

程序说明。

（1）语句“extern int pieces_length;”声明了一个变量，这个变量是在 parse_metafile.c 中定义的全局变量。如果要在其他源文件中使用某个源文件中定义的变量，需要在使用该变量的源文件的头部以 extern 关键字声明。注意声明和定义的区别，声明仅仅是告知编译器有某个变量，而对于定义，编译器要分配内存空间来存储该变量的值。

（2）全局变量 bitmap 指向自己的位图，可以从位图中获知下载的进度。peer 的位图存放在 Peer 结构体中。

- int create_bitfield()

功能：创建待下载文件的位图。

```
int create_bitfield()
{
    bitmap = (Bitmap *)malloc(sizeof(Bitmap));
    if(bitmap == NULL) {
        printf("allocate memory for bitmap failed\n");
        return -1;
    }
```

我爱自由 5ifreedom.com

```
    // pieces_length 除以 20 即为总的 piece 数
    bitmap->valid_length = pieces_length / 20;
    bitmap->bitfield_length = pieces_length / 20 / 8;
    if( (pieces_length/20) % 8 != 0 )  bitmap->bitfield_length++;

    bitmap->bitfield = (unsigned char *)malloc(bitmap->bitfield_length);
    if(bitmap->bitfield == NULL)  {
        printf("allocate memory for bitmap->bitfield fiailed\n");
        if(bitmap != NULL)  free(bitmap);
        return -1;
    }

    char bitmapfile[64];
    sprintf(bitmapfile,"%dbitmap",pieces_length);
    int  i;
    FILE *fp = fopen(bitmapfile,"rb");
    if(fp == NULL) {  // 若打开文件失败,说明开始的是一个全新的下载
        memset(bitmap->bitfield, 0, bitmap->bitfield_length);
} else {
        fseek(fp,0,SEEK_SET);
        for(i = 0; i < bitmap->bitfield_length; i++)
            (bitmap->bitfield)[i] = fgetc(fp);
        fclose(fp);
        // 给 download_piece_num 赋新的初值
        download_piece_num = get_download_piece_num();
    }

    return 0;
}
```

- int get_bit_value(Bitmap *bitmap,int index)

功能：获取位图中某一位的值。

```
int get_bit_value(Bitmap *bitmap,int index)
{
    int             ret;
    int             byte_index;
    unsigned char     byte_value;
    unsigned char     inner_byte_index;

    if(bitmap==NULL || index >= bitmap->valid_length)  return -1;
    byte_index = index / 8;
    byte_value = bitmap->bitfield[byte_index];
    inner_byte_index = index % 8;

    byte_value = byte_value >> (7 - inner_byte_index);
    if(byte_value % 2 == 0) ret = 0;
    else ret = 1;

    return ret;
}
```

为了方便对 get-bit-value 函数的理解，可以假设某位图为 2 字节(其值为 10110011 01010100)，有效位数为 14 位，也就是待下载文件共有 14 个 piece。位图第一个字节指明 index 为 0～7 的 piece 是否已下载，第二个字节指明 index 为 8～13 的 piece 是否已下载。现在要判断 index 为 8 的 piece 是否已经下载，也就是要获取位图第二个字节最高位的值。

- int set_bit_value(Bitmap *bitmap,int index,unsigned char value)

功能：设置位图中某一位的值。（源代码见光盘）

- int all_zero(Bitmap *bitmap)

功能：将位图所有位清 0。

```
int all_zero(Bitmap *bitmap)
{
    if(bitmap->bitfield == NULL) return -1;
```

我爱自由 5ifreedom.com

```
    memset(bitmap->bitfield,0,bitmap->bitfield_length);
    return 0;
}
```

- int all_set(Bitmap *bitmap)

功能：将位图所有位放置 1。（源代码见光盘）

- void release_memory_in_bitfield()

功能：释放本模块所申请的动态内存。

```
void release_memory_in_bitfield()
{
    if(bitmap->bitfield != NULL) free(bitmap->bitfield);
    if(bitmap != NULL)  free(bitmap);
}
```

- int print_bitfield(Bitmap *bitmap)

功能：打印位图，用于调试程序。（源代码见光盘）

- int restore_bitmap()

功能：保存位图，用于断点续传。

```
int restore_bitmap()
{
    int  fd;
    char bitmapfile[64];

    if( (bitmap == NULL) || (file_name == NULL) )  return -1;
    sprintf(bitmapfile,"%dbitmap",pieces_length);
    fd = open(bitmapfile,O_RDWR|O_CREAT|O_TRUNC,0666);
    if(fd < 0)  return -1;
    write(fd,bitmap->bitfield,bitmap->bitfield_length);
    close(fd);

    return 0;
}
```

- int is_interested(Bitmap *dst,Bitmap *src)

功能：判断具有 src 位图的 peer 对具有 dst 位图的 peer 是否感兴趣。

```
int is_interested(Bitmap *dst,Bitmap *src)
{
    unsigned char const_char[8] = { 0x80,0x40,0x20,0x10,0x08,0x04,0x02,0x01};
    unsigned char c1, c2;
    int           i, j;

    if( dst==NULL || src==NULL )  return -1;
    if( dst->bitfield==NULL || src->bitfield==NULL )  return -1;
    if(     dst->bitfield_length!=src->bitfield_length     ||     dst->valid_length!=src->
valid_length )
        return -1;
    // 如果 dst 中某位为 1 而 src 对应为 0，则说明 src 对 dst 感兴趣
    for(i = 0; i < dst->bitfield_length-1; i++) {
        for(j = 0; j < 8; j++) {  // 比较某个字节的所有位
            c1 = (dst->bitfield)[i] & const_char[j];  // 获取每一位的值
            c2 = (src->bitfield)[i] & const_char[j];
            if(c1>0 && c2==0)  return 1;
        }
    }

    j = dst->valid_length % 8;
    c1 = dst->bitfield[dst->bitfield_length-1];
    c2 = src->bitfield[src->bitfield_length-1];
    for(i = 0; i < j; i++) {  // 比较位图的最后一个字节
        if( (c1&const_char[i])>0 && (c2&const_char[i])==0 )
            return 1;
    }
    return 0;
```

我爱自由 5ifreedom.com

```
}
```

以上函数的功能正确性测试代码如下：

```
// 测试时可以交换 map1.bitfield 和 map2.bitfield 的值或赋于其他值
Bitmap map1, map2;
unsigned char bf1[2] = { 0xa0, 0xa0 };  // 位图每一位的值为 10100000 10100000
unsigned char bf2[2] = { 0xe0, 0xe0 };  // 位图每一位的值为 11100000 11100000

map1.bitfield        = bf1;
map1.bitfield_length  = 2;
map1.valid_length     = 11;
map2.bitfield        = bf2;
map2.bitfield_length  = 2;
map2.valid_length     = 11;

int ret = is_interested(&map1,&map2);
printf("%d\n",ret);
```

在编写模块时，测试其中的每一个函数是很有必要的，否则无法知道模块中每一个函数是否达到预期的功能。限于篇幅，不能列出每个模块的测试代码。由于每个模块的相对独立性，读者不妨编写一些测试代码来测试某些模块的代码。

- int get_download_piece_num()

功能：获取当前已下载到的总 piece 数。函数实现代码如下：

```
int get_download_piece_num()
{
    unsigned char const_char[8] = { 0x80,0x40,0x20,0x10,0x08,0x04,0x02,0x01};
    int          i, j;

    if(bitmap==NULL || bitmap->bitfield==NULL)  return 0;
    download_piece_num = 0;

    for(i = 0; i < bitmap->bitfield_length-1; i++) {
        for(j = 0; j < 8; j++) {
            if( ((bitmap->bitfield)[i] & const_char[j]) != 0)
                download_piece_num++;
        }
    }

    unsigned char c = (bitmap->bitfield)[i];  // c 存放位图最后一个字节的值
    j = bitmap->valid_length % 8;          // j 是位图最后一个字节的有效位数
    for(i = 0; i < j; i++) {
        if( (c & const_char[i]) !=0 ) download_piece_num++;
    }
    return download_piece_num;
}
```

13.4.3 出错处理模块的设计和实现

该模块由 bterror.h 和 bterror.c 文件构成，主要定义了一些错误类型，以及发生导致程序终止的致命性错误时程序的响应。

```
// bterror.h
#ifndef BTERROR_H
#define BTERROR_H

#define FILE_FD_ERR                    -1   // 无效的文件描述符
#define FILE_READ_ERR                  -2   // 读文件失败
#define FILE_WRITE_ERR                 -3   // 写文件失败
#define INVALID_METAFILE_ERR           -4   // 无效的种子文件
#define INVALID_SOCKET_ERR             -5   // 无效的套接字
#define INVALID_TRACKER_URL_ERR        -6   // 无效的 Tracker URL
#define INVALID_TRACKER_REPLY_ERR      -7   // 无效的 Tracker 回应
#define INVALID_HASH_ERR               -8   // 无效的 hash 值
#define INVALID_MESSAGE_ERR            -9   // 无效的消息
```

我爱自由 5ifreedom.com

```
#define INVALID_PARAMETER_ERR                    -10  // 无效的函数参数
#define FAILED_ALLOCATE_MEM_ERR                  -11 // 申请动态内存失败
#define NO_BUFFER_ERR                            -12  // 没有足够的缓冲区
#define READ_SOCKET_ERR                          -13  // 读套接字失败
#define WRITE_SOCKET_ERR                         -14  // 写套接字失败
#define RECEIVE_EXIT_SIGNAL_ERR                  -15  // 接收到退出程序的信号

// 用于提示致命性的错误,程序将终止
void btexit(int errno,char *file,int line);

#endif
```

以下是 bterror.c 文件：

```
// bterror.c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include "bterror.h"

void btexit(int errno,char *file,int line)
{
    printf("exit at %s : %d with error number : %d\n",file, line, errno);
    exit(errno);
}
```

13.4.4 运行日志模块的设计和实现

本模块负责记录程序运行的日志，以备查询和分析程序行为，由 log.h 和 log.c 两个文件构成。

```
// log.h
#ifndef  LOG_H
#define  LOG_H
#include <stdarg.h>

// 用于记录程序的行为
void logcmd(char *fmt,...);

// 打开日志文件
int init_logfile(char *filename);

// 将程序运行日志记录到文件
int logfile(char *file,int line,char *msg);

#endif
```

以下是 log.c 文件：

```
// bterror.c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include "log.h"
// 日志文件的描述符
int logfile_fd = -1;
// 在命令行上打印一条日志
void logcmd(char *fmt,...)
{
    va_list ap;

    va_start(ap,fmt);
    vprintf(fmt,ap);
    va_end(ap);
}
// 打开记录日志的文件
int init_logfile(char *filename)
```

我爱自由 5ifreedom.com

```
{
    logfile_fd = open(filename,O_RDWR|O_CREAT|O_APPEND,0666);
    if(logfile_fd < 0) {
        printf("open logfile failed\n");
        return -1;
    }
    return 0;
}
// 将一条日志写入日志文件
int logfile(char *file,int line,char *msg)
{
    char buff[256];

    if(logfile_fd < 0)  return -1;
    snprintf(buff,256,"%s:%d %s\n",file,line,msg);
    write(logfile_fd,buff,strlen(buff));
    return 0;
}
```

程序说明。

函数 logcmd 是一个变长参数的函数，也就是函数的参数个数是可变的，类似于 printf 函数。语句“logcmd(“%s:%d error\n”,__FILE__, __LINE__);”的功能与“printf(“%s:%d error\n”,__FILE__, __LINE__);” 功能相同。

13.4.5 信号处理模块的设计和实现

在运行过程中，程序可能会接收到一些信号，如 SIGINT、SIGTERM，这些信号的默认动作是立即终止程序。在信号处理模块，定义处理这些信号的函数。当信号产生时，系统自动调用相应的信号处理函数以便执行一些善后操作，如释放动态申请的内存、关闭文件描述符、关闭套接字。本模块由 signal_hander.h 和 signal_hander.c 两个文件构成。

```
// signal_hander.h
#ifndef SIGNAL_HANDER_H
#define SIGNAL_HANDER_H

// 做一些清理工作,如释放动态分配的内存
void do_clear_work();
// 处理一些信号
void process_signal(int signo);
// 设置信号处理函数
int set_signal_hander();

#endif
```

以下是 signal_hander.c 文件：

```
// signal_hander.c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include "parse_metafile.h"
#include "bitfield.h"
#include "peer.h"
#include "data.h"
#include "tracker.h"
#include "torrent.h"
#include "signal_hander.h"

extern int  download_piece_num;
extern int  *fds;
extern int  fds_len;
extern Peer *peer_head;
```

我爱自由 5ifreedom.com

```
// 程序将退出时，执行一些清理工作
void do_clear_work()
{
    // 关闭所有 peer 的 socket
    Peer *p = peer_head;
    while(p != NULL) {
        if(p->state != CLOSING)  close(p->socket);
        p = p->next;
    }
    // 保存位图
    if(download_piece_num > 0) {
        restore_bitmap();
    }
    // 关闭文件描述符
    int i;
    for(i = 0; i < fds_len; i++) {
        close(fds[i]);
    }
    // 释放动态分配的内存
    release_memory_in_parse_metafile();
    release_memory_in_bitfield();
    release_memory_in_btcache();
    release_memory_in_peer();
    release_memory_in_torrent();

    exit(0);
}

void process_signal(int signo)
{
    printf("Please wait for clear operations\n");
    do_clear_work();
}

// 设置信号处理函数
int set_signal_hander()
{
    if(signal(SIGPIPE,SIG_IGN) == SIG_ERR) {
        perror("can not catch signal:sigpipe\n");
        return -1;
    }

    if(signal(SIGINT,process_signal) == SIG_ERR) {
        perror("can not catch signal:sigint\n");
        return -1;
    }

    if(signal(SIGTERM,process_signal) == SIG_ERR) {
        perror("can not catch signal:sigterm\n");
        return -1;
    }

    return 0;
}
```

13.4.6 Peer 管理模块的设计和实现

系统为每一个与之建立 TCP 连接的 Peer 构造一个 Peer 结构体。Peer 管理模块负责管理由各个 Peer 结点构成的 Peer 链表，主要工作是创建结点，添加结点到 Peer 链表，从 Peer 链表中删除结点等。

```
// peer.h
#ifndef PEER_H
#define PEER_H
#include <string.h>
#include <time.h>
#include "bitfield.h"
```

我爱自由 5ifreedom.com

```
#define  INITIAL                          -1    // 表明处于初始化状态
#define  HALFSHAKED                       0     // 表明处于半握手状态
#define  HANDSHAKED                       1     // 表明处于全握手状态
#define  SENDBITFIELD                     2     // 表明处于已发送位图状态
#define  RECVBITFIELD                     3     // 表明处于已接收位图状态
#define  DATA                             4     // 表明处于与 peer 交换数据的状态
#define  CLOSING                          5     // 表明处于即将与 peer 断开的状态

// 发送和接收缓冲区的大小,16K 可以存放一个 slice,2K 用来存放其他消息
#define  MSG_SIZE   (2*1024+16*1024)

typedef struct _Request_piece {
    int     index;                        // 请求的 piece 的索引
    int     begin;                        // 请求的 piece 的偏移
    int     length;                       // 请求的长度,一般为 16KB
    struct _Request_piece *next;
} Request_piece;  // 定义数据请求队列的结点

typedef struct  _Peer {
    int           socket;                 // 通过该 socket 与 peer 进行通信
    char          ip[16];                 // peer 的 ip 地址
    unsigned short   port;                // peer 的端口号
    char          id[21];                 // peer 的 id

    int           state;                  // 当前所处的状态
    int           am_choking;             // 是否将 peer 阻塞
    int           am_interested;          // 是否对 peer 感兴趣
    int           peer_choking;           // 是否被 peer 阻塞
    int           peer_interested;        // 是否被 peer 感兴趣

    Bitmap        bitmap;             // 存放 peer 的位图

    char         *in_buff;                // 存放从 peer 处获取的消息
    int           buff_len;               // 缓存区 in_buff 的长度
    char         *out_msg;                // 存放将发送给 peer 的消息
    int           msg_len;                // 缓冲区 out_msg 的长度
    char         *out_msg_copy;           // out_msg 的副本,发送时使用该缓冲区
    int           msg_copy_len;           // 缓冲区 out_msg_copy 的长度
    int           msg_copy_index;         // 下一次要发送的数据的偏移量

    Request_piece  *Request_piece_head;   // 向 peer 请求数据的队列
    Request_piece  *Requested_piece_head; // 被 peer 请求数据的队列

    unsigned int    down_total;           // 从该 peer 下载的总字节数
    unsigned int    up_total;             // 向该 peer 上传的总字节数

    time_t        start_timestamp;        // 最近一次接收到 peer 消息的时间
    time_t        recet_timestamp;        // 最近一次发送消息给 peer 的时间
    time_t        last_down_timestamp;    // 最近下载数据的开始时间
    time_t        last_up_timestamp;      // 最近上传数据的开始时间
    long long      down_count;            // 本计时周期从 peer 下载的数据的字节数
    long long      up_count;              // 本计时周期向 peer 上传的数据的字节数
    float         down_rate;              // 本计时周期从 peer 处下载数据的速度
    float         up_rate;                // 本计时周期向 peer 处上传数据的速度

    struct _Peer   *next;
} Peer;

int   initialize_peer(Peer *peer);        // 对 peer 各个成员进行初始化
Peer* add_peer_node();                    // 添加一个 peer 结点
int   del_peer_node(Peer *peer);          // 删除一个 peer 结点
void  free_peer_node(Peer *node);         // 释放一个 peer 结点的内存
int   cancel_request_list(Peer *node);    // 撤销当前请求队列
int   cancel_requested_list(Peer *node);  // 撤销当前被请求队列
void  release_memory_in_peer();           // 释放 peer.c 中的动态分配的内存
void  print_peers_data();                 // 打印 peer 链表中某些成员的值,用于调试

#endif
```

我爱自由 5ifreedom.com

程序说明。

（1）Peer 结构体是整个程序最重要的数据结构，也是最复杂的数据结构。peer.h 中首先定义了 7 种状态。各个状态的转换图如图 13-3 所示。

Halfshaked（半握手状态）是指已发送握手消息但未收到对方的握手消息，或者已经接收到对方的握手消息，但己方未发送握手消息。处于 Data 状态时双方可以交换数据，此时 Peer 结构体中的 am_choking 、 am_interested 、 peer_choking 、 peer_interested 4 个成员变量有效。

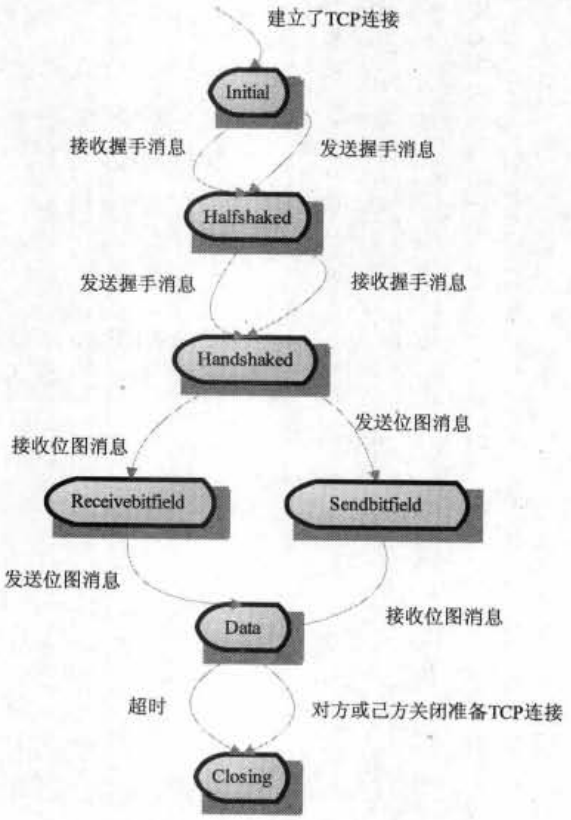


图 13-3　状态转换图

如图 13-4 所示，当 am_interested = 1，peer_choking = 0 时，也就是客户端对 peer 感兴趣，而且 peer 没有将客户端阻塞，此时可以发送数据请求，即发送 request 消息请求 peer 发送数据，peer 接收到请求后发送 piece 消息，数据就被封装在 piece 消息中。

如图 13-5 所示，当 peer_interested = 1，am_choking = 0 时，也就是 peer 对客户端感兴趣，而且客户端没有将该 peer 阻塞，此时如果 peer 发送 request 消息请求数据，则应该构造并发送 piece 消息，其中数据被封装在 piece 消息中。

图 13-5 中“发送 unchoke 消息”的时机是执行选择非阻塞 peer 算法时，选中该 peer 作为非阻塞 peer 或者选中该 peer 作为优化非阻塞 peer。“发送 choke 消息”的时机类似。

“发送 have 消息，拥有了 peer 没有的 piece”的含义是己方刚刚下载到一个 piece，此时通过发送 have 消息告知所有 peer 客户端已拥有了某个 piece，如果 peer 没有这个 piece 且原先 peer 对本客户端不感兴趣，则发送 have 消息后，该 peer 就对该客户端感兴趣了。

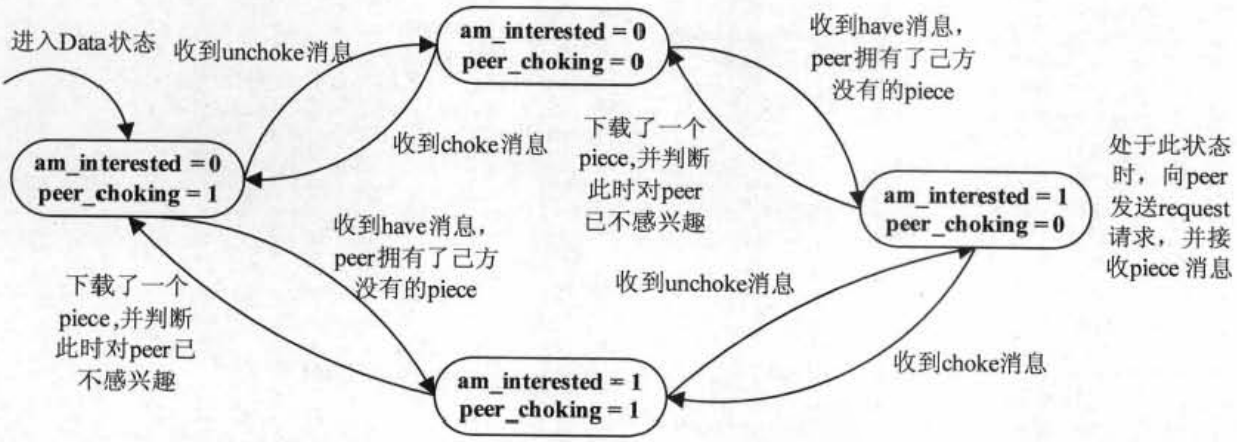


图 13-4　从下载数据的角度看到的处于 Data 状态时的内部状态

我爱自由 5ifreedom.com

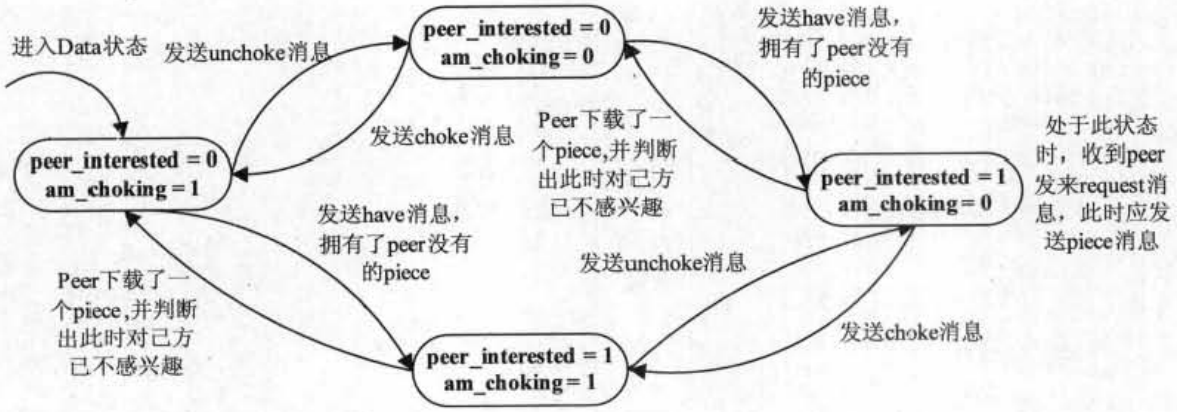


图 13-5　从上传数据的角度看到的处于 Data 状态时的内部状态

（2）Peer 结构体中定义两个发送缓冲区 out_msg 和 out_msg_copy，将刚刚生成的消息暂存在 out_msg 中，调用套接字函数 send 向 peer 发送消息时使用 out_msg_copy 缓冲区。out_msg_copy 缓冲区的大小是 18KB，而 send 函数一次最多发送 1500 字节，因此要使用 msg_copy_index 记录下次应该发送的数据的起始下标。事实上，send 函数一次也可以发送超过 1500 字节的数据，不过若以这种方式发送会导致发送数据混乱，具体原因后面将会解释。其变量的含义请参考消息处理模块。

peer.c 文件的头部包含的代码如下：

```
// peer.c
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include "peer.h"
#include "message.h"
#include "bitfield.h"

extern Bitmap *bitmap;

// 指向当前与之进行通信的 peer 链表
Peer *peer_head = NULL;
```

peer.c 中各个函数的定义如下。

- initialize_peer(Peer *peer)

功能：初始化 Peer 结构体。

```
int  initialize_peer(Peer *peer)
{
    if(peer == NULL)   return -1;
    peer->socket = -1;
    memset(peer->ip,0,16);
    peer->port = 0;
    memset(peer->id,0,21);
    peer->state = INITIAL;

    peer->in_buff = NULL;
    peer->out_msg = NULL;
    peer->out_msg_copy = NULL;

    peer->in_buff = (char *)malloc(MSG_SIZE);
    if(peer->in_buff == NULL)  goto OUT;
    memset(peer->in_buff,0,MSG_SIZE);
    peer->buff_len = 0;

    peer->out_msg = (char *)malloc(MSG_SIZE);
    if(peer->out_msg == NULL)  goto OUT;
    memset(peer->out_msg,0,MSG_SIZE);
    peer->msg_len  = 0;
```

我爱自由 5ifreedom.com

```
    peer->out_msg_copy = (char *)malloc(MSG_SIZE);
    if(peer->out_msg_copy == NULL)  goto OUT;
    memset(peer->out_msg_copy,0,MSG_SIZE);
    peer->msg_copy_len   = 0;
    peer->msg_copy_index = 0;

    peer->am_choking = 1;
    peer->am_interested = 0;
    peer->peer_choking = 1;
    peer->peer_interested = 0;

    peer->bitmap.bitfield = NULL;
    peer->bitmap.bitfield_length = 0;
    peer->bitmap.valid_length = 0;

    peer->Request_piece_head = NULL;
    peer->Requested_piece_head = NULL;

    peer->down_total = 0;
    peer->up_total = 0;

    peer->start_timestamp = 0;
    peer->recet_timestamp = 0;

    peer->last_down_timestamp = 0;
    peer->last_up_timestamp = 0;
    peer->down_count = 0;
    peer->up_count = 0;
    peer->down_rate = 0.0;
    peer->up_rate = 0.0;

    peer->next = (Peer *)0;
    return 0;
OUT:
    if(peer->in_buff != NULL)              free(peer->in_buff);
    if(peer->out_msg != NULL)              free(peer->out_msg);
    if(peer->out_msg_copy != NULL)         free(peer->out_msg_copy);
    return -1;
}
```

- Peer* add_peer_node()

功能：向 peer 链表添加一个结点。

```
Peer* add_peer_node()
{
    int  ret;
    Peer *node, *p;

    // 分配内存空间
    node = (Peer *)malloc(sizeof(Peer));
    if(node == NULL)  {
        printf("%s:%d error\n",__FILE__,__LINE__);
        return NULL;
    }
    // 进行初始化
    ret = initialize_peer(node);
    if(ret < 0) {
        printf("%s:%d error\n",__FILE__,__LINE__);
        free(node);
        return NULL;
    }
    // 将 node 加入到 peer 链表中
    if(peer_head == NULL)  { peer_head = node; }  // node 为 peer 链表的第一个结点
    else {
        p = peer_head;   // 使 p 指针指向 peer 链表的最后一个结点
        while(p->next != NULL)  p = p->next;
        p->next = node;
```

我爱自由 5ifreedom.com

```
    }
    return node;
}
```

- int del_peer_node(Peer *peer)

功能：从 peer 链表中删除一个结点。

```
int del_peer_node(Peer *peer)
{
    Peer *p = peer_head, *q;
    if(peer == NULL)  return -1;

    while(p != NULL) {
        if( p == peer ) {
            if(p == peer_head)  peer_head = p->next;
            else  q->next = p->next;
            free_peer_node(p);
            return 0;
        } else {
            q = p;
            p = p->next;
        }
    }

    return -1;
}
```

- int cancel_request_list(Peer *node)

功能：撤销当前请求队列。

```
int cancel_request_list(Peer *node)
{
Request_piece  *p = node->Request_piece_head;

    while(p != NULL) {
        node->Request_piece_head = node->Request_piece_head->next;
        free(p);
        p = node->Request_piece_head;
    }

    return 0;
}
```

- int cancel_requested_list(Peer *node)

功能：撤销当前被请求队列。

```
int cancel_requested_list(Peer *node)
{
Request_piece  * p = node->Requested_piece_head;

    while(p != NULL) {
        node->Requested_piece_head = node->Requested_piece_head->next;
        free(p);
        p = node->Requested_piece_head;
    }

    return 0;
}
```

- void free_peer_node(Peer *node)

功能：释放一个 peer 结点的内存。（源代码见光盘）

- void release_memory_in_peer()

功能：释放 peer 管理模块中动态申请的内存。（源代码见光盘）

- void print_peers_data()

功能：打印 peer 结点的一些信息，用于调试程序。（源代码见光盘）

我爱自由 5ifreedom.com

13.4.7 消息处理模块的设计和实现

消息处理模块负责根据当前的状态生成并发送消息，接收以及处理消息。消息处理模块由 messag.h 和 message.c 两个文件构成，理解和分析本模块的代码时请参考图 13-3、图 13-4 和图 13-5。

```
message.h
#ifndef MESSAGE_H
#define MESSAGE_H
#include "peer.h"

int int_to_char(int i, unsigned char c[4]);      // 将整型变量 i 的 4 个字节存放到数组 c 中
int char_to_int(unsigned char c[4]);             // 将数组 c 中的 4 个字节转换为一个整型数

// 以下函数创建各个类型的消息，创建消息的函数请参考 BT 协议加以理解
int create_handshake_msg(char *info_hash,char *peer_id,Peer *peer);
int create_keep_alive_msg(Peer *peer);
int create_chock_interested_msg(int type,Peer *peer);
int create_have_msg(int index,Peer *peer);
int create_bitfield_msg(char *bitfield,int bitfield_len,Peer *peer);
int create_request_msg(int index,int begin,int length,Peer *peer);
int create_piece_msg(int index,int begin,char *block,int b_len,Peer *peer);
int create_cancel_msg(int index,int begin,int length,Peer *peer);
int create_port_msg(int port,Peer *peer);

// 判断接收缓冲区中是否存放了一条完整的消息
int is_complete_message(unsigned char *buff,unsigned int len,int *ok_len);
// 处理收到的消息,接收缓冲区中存放着一条完整的消息
int parse_response(Peer *peer);
// 处理收到的消息,接收缓冲区中除了存放着一条完整的消息外,还有其他不完整的消息
int parse_response_uncomplete_msg(Peer *p,int ok_len);
// 根据当前的状态创建响应消息
int create_response_message(Peer *peer);
// 为发送 have 消息作准备,have 消息较为特殊,它要发送给所有 peer
int prepare_send_have_msg();
// 即将与 peer 断开时,丢弃套接字发送缓冲区中的所有未发送的消息
void discard_send_buffer(Peer *peer);

#endif
```

message.c 文件的头部包括的代码如下：

```
message.h
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <unistd.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <sys/socket.h>
#include "parse_metafile.h"
#include "bitfield.h"
#include "peer.h"
#include "policy.h"
#include "data.h"
#include "message.h"

#define HANDSHAKE       -2   // 握手消息
#define KEEP_ALIVE      -1   // keep_alive 消息
#define CHOKE           0    // choke 消息
#define UNCHOKE         1    // unchoke 消息
#define INTERESTED      2    // interested 消息
#define UNINTERESTED    3    // uninterested 消息
#define HAVE            4    // have 消息
#define BITFIELD        5    // bitfield 消息
#define REQUEST         6    // request 消息
```

我爱自由 5ifreedom.com

```
#define PIECE           7     // piece 消息
#define CANCEL          8     // cancel 消息
#define PORT            9     // port 消息

// 如果 45 秒未给某 peer 发送消息，则发送 keep_alive 消息
#define KEEP_ALIVE_TIME 45

extern Bitmap *bitmap;                  // 在 bitmap.c 中定义，指向己方的位图
extern char    info_hash[20];           // 在 parse_metafile.c 中定义，存放 info_hash
extern char    peer_id[20];             // 在 parse_metafile.c 中定义，存放 peer_id
extern int     have_piece_index[64];    // 在 data.c 中定义，存放下载到的 piece 的 index
extern Peer   *peer_head;               // 在 peer.c 中定义，指向 peer 链表
```

message.c 中各个函数的定义如下：

- int int_to_char(int i, unsigned char c[4])

功能：获取 i 的各个字节，并保存到字符数组 c 中。

```
int int_to_char(int i, unsigned char c[4])
{
    c[3] = i%256;
    c[2] = (i-c[3])/256%256;
    c[1] = (i-c[3]-c[2]*256)/256/256%256;
    c[0] = (i-c[3]-c[2]*256-c[1]*256*256)/256/256/256%256;

    return 0;
}
```

程序说明。

假设 i = 123456789，若以 16 进制表示该数为 0x75BCD15，则 c[0] = 07，c[1] = 5B，c[2] = CD，c[3] = 15。函数 char_to_int 的功能与本函数刚好相反。

- int char_to_int(unsigned char c[4])

功能：将字符数组中的字符转换为一个整型。

```
int char_to_int(unsigned char c[4])
{
    int i;

    i = c[0]*256*256*256 + c[1]*256*256 + c[2]*256 + c[3];

    return i;
}
```

- int create_handshake_msg(char *info_hash,char *peer_id,Peer *peer)

功能：创建握手消息

参数：info_hash 在 parse_metafile.c 中由种子文件计算而得；peer_id 也在 parse_metafile.c 中生成；peer 为要发送握手消息的某一个 peer 的指针变量。

返回：创建消息成功返回 0，创建失败返回-1。函数实现代码如下：

```
int create_handshake_msg(char *info_hash,char *peer_id,Peer *peer)
{
    int          i;
    unsigned char  keyword[20] = "BitTorrent protocol", c = 0x00;
    unsigned char  *buffer = peer->out_msg + peer->msg_len;
    int          len = MSG_SIZE - peer->msg_len;

    if(len < 68)  return -1;  // 握手消息的长度固定为 68 字节

    buffer[0] = 19;
    for(i = 0; i < 19; i++)      buffer[i+1]  = keyword[i];
    for(i = 0; i < 8;  i++)      buffer[i+20] = c;
    for(i = 0; i < 20; i++)      buffer[i+28] = info_hash[i];
    for(i = 0; i < 20; i++)      buffer[i+48] = peer_id[i];

    peer->msg_len += 68;
    return 0;
}
```

我爱自由 5ifreedom.com

程序说明。

将生成的握手消息存放在 peer 结点的发送缓冲区中(即 msg_out)，其中 msg_out[0]～msg_out[msg_len-1]已存放了其他消息。函数中变量 len 指明缓冲区还有多少空闲。初始情况下，msg_len 的值应为 0。

- int create_keep_alive_msg(Peer *peer)

功能：创建 keep_alive 消息。

```
int create_keep_alive_msg(Peer *peer)
{
    unsigned char  *buffer = peer->out_msg + peer->msg_len;
    int            len = MSG_SIZE - peer->msg_len;

    if(len < 4)  return -1;  // keep_alive 消息的长度固定为 4
    memset(buffer,0,4);
    peer->msg_len += 4;
    return 0;
}
```

- int create_chock_interested_msg(int type,Peer *peer)

功能：创建 chock 消息。

```
int create_chock_interested_msg(int type,Peer *peer)
{
    unsigned char  *buffer = peer->out_msg + peer->msg_len;
    int            len = MSG_SIZE - peer->msg_len;

    // choke、unchoke、interested、uninterested 消息的长度固定为 5
    if(len < 5)  return -1;
    memset(buffer,0,5);
    buffer[3] = 1;
    buffer[4] = type;

    peer->msg_len += 5;
    return 0;
}
```

程序说明。

该函数可创建 4 种消息，即 choke、unchoke、interested、uninterested 消息。choke 消息的 type 值为 0，unchoke 消息为 1，interested 消息为 2，uninterested 消息为 3。

- int create_have_msg(int index,Peer *peer)

功能：创建 have 消息。

```
int create_have_msg(int index,Peer *peer)
{
    unsigned char  *buffer = peer->out_msg + peer->msg_len;
    int            len = MSG_SIZE - peer->msg_len;
    unsigned char  c[4];

    if(len < 9)  return -1;      // have 消息的长度固定为 9
    memset(buffer,0,9);
    buffer[3] = 5;
    buffer[4] = 4;
    int_to_char(index,c);        // index 为 piece 的下标
    buffer[5] = c[0];
    buffer[6] = c[1];
    buffer[7] = c[2];
    buffer[8] = c[3];

    peer->msg_len += 9;
    return 0;
}
```

- int create_bitfield_msg(char *bitfield,int bitfield_len,Peer *peer)

我爱自由 5ifreedom.com

功能：创建 bitfield 消息。

```
int create_bitfield_msg(char *bitfield,int bitfield_len,Peer *peer)
{
    int          i;
    unsigned char  c[4];
    unsigned char  *buffer = peer->out_msg + peer->msg_len;
    int          len = MSG_SIZE - peer->msg_len;

    if( len < bitfield_len+5 )  {  // bitfield 消息的长度为 bitfield_len+5
        printf("%s:%d buffer too small\n",__FILE__,__LINE__);
        return -1;
    }
    int_to_char(bitfield_len+1,c); // 位图消息的负载长度为位图长度加 1
    for(i = 0; i < 4; i++)  buffer[i] = c[i];
    buffer[4] = 5;
    for(i = 0; i < bitfield_len; i++) buffer[i+5] = bitfield[i];

    peer->msg_len += bitfield_len+5;
    return 0;
}
```

- int create_request_msg(int index,int begin,int length,Peer *peer)

功能：创建数据请求消息。

参数：index 为请求的 piece 的下标；begin 为 piece 内的偏移量；length 为请求数据的长度。函数实现的代码如下：

```
int create_request_msg(int index,int begin,int length,Peer *peer)
{
    int          i;
    unsigned char  c[4];
    unsigned char  *buffer = peer->out_msg + peer->msg_len;
    int          len = MSG_SIZE - peer->msg_len;

    if(len < 17)  return -1;  // request 消息的长度固定为 17
    memset(buffer,0,17);
    buffer[3] = 13;
    buffer[4] = 6;
    int_to_char(index,c);
    for(i = 0; i < 4; i++)  buffer[i+5]  = c[i];
    int_to_char(begin,c);
    for(i = 0; i < 4; i++)  buffer[i+9]  = c[i];
    int_to_char(length,c);
    for(i = 0; i < 4; i++)  buffer[i+13] = c[i];

    peer->msg_len += 17;
    return 0;
}
```

- int create_piece_msg(int index,int begin,char *block,int b_len,Peer *peer)

功能：创建 piece 消息。

参数：block 指向待发送的数据；b_len 为 block 所指向的数据的长度。函数实现的代码如下：

```
int create_piece_msg(int index,int begin,char *block,int b_len,Peer *peer)
{
    int          i;
    unsigned char  c[4];
    unsigned char  *buffer = peer->out_msg + peer->msg_len;
    int          len = MSG_SIZE - peer->msg_len;

    if( len < b_len+13 ) {  // piece 消息的长度为 b_len+13
        printf("%s:%d buffer too small\n",__FILE__,__LINE__);
        return -1;
    }

    int_to_char(b_len+9,c);
```

我爱自由 5ifreedom.com

```
    for(i = 0; i < 4; i++)      buffer[i] = c[i];
    buffer[4] = 7;
    int_to_char(index,c);
    for(i = 0; i < 4; i++)      buffer[i+5] = c[i];
    int_to_char(begin,c);
    for(i = 0; i < 4; i++)      buffer[i+9] = c[i];
    for(i = 0; i < b_len; i++)  buffer[i+13] = block[i];

    peer->msg_len += b_len+13;
    return 0;
}
```

- int create_cancel_msg(int index,int begin,int length,Peer *peer)

功能：创建 cancel 消息。（源代码见光盘）

- int create_port_msg(int port,Peer *peer)

功能：创建 port 消息。（源代码见光盘）

附注：实际上程序从未发送过本消息，因为根据 BT 协议，该消息是为那些以 DHT 的方式获取 peer 地址的应用程序所准备的。

- int is_complete_message(unsigned char *buff,unsigned int len,int *ok_len)

功能：判断缓冲区中是否存放了一条完整的消息。

参数：buff 指向存放消息的缓冲区；len 为缓冲区 buff 的长度；ok_len 用于返回缓冲区中完整消息的长度，即 buff[0]～buff[ok_len-1]存放着一条或多条完整的消息。函数实现的完整代码见附带光盘。

- int process_handshake_msg(Peer *peer,unsigned char *buff,int len)

功能：处理接收到的一条握手消息。

参数：从 peer 接收到这条握手消息；buff 指向握手消息；len 为 buff 的长度。函数实现的代码如下：

```
int process_handshake_msg(Peer *peer,unsigned char *buff,int len)
{
    if(peer==NULL || buff==NULL) return -1;
    if(memcmp(info_hash,buff+28,20) != 0) {  // 若 info_hash 不一致则关闭连接
        peer->state = CLOSING;
        // 丢弃发送缓冲区中的数据
        discard_send_buffer(peer);
        clear_btcache_before_peer_close(peer);
        close(peer->socket);
        return -1;
    }
    // 保存该 peer 的 peer_id
    memcpy(peer->id,buff+48,20);
    (peer->id)[20] = '\0';
    // 若当前处于 Initial 状态，则发送握手消息给 peer
    if(peer->state == INITIAL) {
        create_handshake_msg(info_hash,peer_id,peer);
        peer->state = HANDSHAKED;
    }
    // 若握手消息已发送，则状态转换为已握手状态
    if(peer->state == HALFSHAKED)  peer->state = HANDSHAKED;
    // 记录最近收到该 peer 消息的时间
    // 若一定时间内(如两分钟)未收到来自该 peer 的任何消息，则关闭连接
    peer->start_timestamp = time(NULL);
    return 0;
}
```

- int process_keep_alive_msg(Peer *peer,unsigned char *buff,int len)

功能：处理刚刚接收到的来自 peer 的 keepv_alive 消息。函数实现的代码如下：

```
int process_keep_alive_msg(Peer *peer,unsigned char *buff,int len)
{
    if(peer==NULL || buff==NULL)  return -1;
    // 记录最近收到该 peer 消息的时间
```

我爱自由 5ifreedom.com

```
// 若一定时间内(如 2min)未收到来自该 peer 的任何消息，则关闭连接
    peer->start_timestamp = time(NULL);
    return 0;
}
```

- int process_choke_msg(Peer *peer,unsigned char *buff,int len)

功能：处理收到的 choke 消息。函数实现的代码如下：

```
int process_choke_msg(Peer *peer,unsigned char *buff,int len)
{
    if(peer==NULL || buff==NULL)  return -1;
// 若原先处于 unchoke 状态，收到该消息后更新 peer 中某些变量的值
    if( peer->state!=CLOSING && peer->peer_choking==0 ) {
        peer->peer_choking = 1;
        peer->last_down_timestamp = 0;          // 将最近接收到来自该 peer 数据的时间清零
        peer->down_count = 0;                   // 将最近从该 peer 处下载的字节数清零
        peer->down_rate = 0;                    // 将最近从该 peer 下载数据的速度清零
    }

    peer->start_timestamp = time(NULL);
    return 0;
}
```

- int process_unchoke_msg(Peer *peer,unsigned char *buff,int len)

功能：处理收到 unchoke 消息。函数实现的代码如下：

```
int process_unchoke_msg(Peer *peer,unsigned char *buff,int len)
{
    if(peer==NULL || buff==NULL)  return -1;
    // 若原来处于 choke 状态且与该 peer 的连接未被关闭
    if( peer->state!=CLOSING && peer->peer_choking==1 ) {
        peer->peer_choking = 0;
        // 若对该 peer 感兴趣，则构造 request 消息请求 peer 发送数据
        if(peer->am_interested == 1)  create_req_slice_msg(peer);
        else {
            peer->am_interested = is_interested(&(peer->bitmap), bitmap);
            if(peer->am_interested == 1)  create_req_slice_msg(peer);
            else  printf("Received unchoke but Not interested to IP:%s \n",peer->ip);
        }
        // 更新一些成员的值
        peer->last_down_timestamp = 0;
        peer->down_count = 0;
        peer->down_rate = 0;
    }

    peer->start_timestamp = time(NULL);
    return 0;
}
```

- int process_interested_msg(Peer *peer,unsigned char *buff,int len)

功能：处理收到的 interested 消息。函数实现的代码如下：

```
int process_interested_msg(Peer *peer,unsigned char *buff,int len)
{
    if(peer==NULL || buff==NULL)  return -1;
    if( peer->state!=CLOSING && peer->state==DATA ) {
        peer->peer_interested = is_interested(bitmap, &(peer->bitmap));
        if(peer->peer_interested == 0)  return -1;
        if(peer->am_choking == 0) create_chock_interested_msg(1,peer);
    }

    peer->start_timestamp = time(NULL);
    return 0;
}
```

- int process_uninterested_msg(Peer *peer,unsigned char *buff,int len)

功能：处理收到的 uninterested 消息。函数实现的代码如下：

```
int process_uninterested_msg(Peer *peer,unsigned char *buff,int len)
{
    if(peer==NULL || buff==NULL)  return -1;
```

我爱自由 5ifreedom.com

```
    if( peer->state!=CLOSING && peer->state==DATA ) {
        peer->peer_interested = 0;
        cancel_requested_list(peer);
    }

    peer->start_timestamp = time(NULL);
    return 0;
}
```

- int process_have_msg(Peer *peer,unsigned char *buff,int len)

功能：处理收到的 have 消息。函数实现的代码如下：

```
int process_have_msg(Peer *peer,unsigned char *buff,int len)
{
    int      rand_num;
    unsigned  char c[4];

    if(peer==NULL || buff==NULL)  return -1;
    srand(time(NULL));
    rand_num = rand() % 3;  // 生成一个 0~2 的随机数
    if( peer->state!=CLOSING && peer->state==DATA ) {
        c[0] = buff[5]; c[1] = buff[6];
        c[2] = buff[7]; c[3] = buff[8];
        // 更新该 peer 的位图
        if(peer->bitmap.bitfield != NULL)
        set_bit_value(&(peer->bitmap),char_to_int(c),1);
        if(peer->am_interested == 0) {
            peer->am_interested = is_interested(&(peer->bitmap), bitmap);
            // 由原来的对 peer 不感兴趣变为感兴趣时,发送 interested 消息
            if(peer->am_interested == 1) create_chock_interested_msg(2,peer);
        } else {  // 收到 3 个 have 则发一个 interested 消息
            if(rand_num == 0) create_chock_interested_msg(2,peer);
        }
    }

    peer->start_timestamp = time(NULL);
    return 0;
}
```

- int process_cancel_msg(Peer *peer,unsigned char *buff,int len)

功能：处理收到的 cancel 消息。函数实现的代码见附带光盘。

- int process_bitfield_msg(Peer *peer,unsigned char *buff,int len)

功能：处理收到的位图消息。函数实现的代码如下：

```
int process_bitfield_msg(Peer *peer,unsigned char *buff,int len)
{
    unsigned char c[4];

    if(peer==NULL || buff==NULL)  return -1;
    if(peer->state==HANDSHAKED || peer->state==SENDBITFIELD) {
        c[0] = buff[0];   c[1] = buff[1];
        c[2] = buff[2];   c[3] = buff[3];
        // 若原先已收到一个位图消息，则清空原来的位图
        if( peer->bitmap.bitfield != NULL ) {
            free(peer->bitmap.bitfield);
            peer->bitmap.bitfield = NULL;
        }
        peer->bitmap.valid_length = bitmap->valid_length;
        if(bitmap->bitfield_length != char_to_int(c)-1) {  // 若收到的一个错误位图
            peer->state = CLOSING;
            // 丢弃发送缓冲区中的数据
            discard_send_buffer(peer);
            clear_btcache_before_peer_close(peer);
            close(peer->socket);
            return -1;
        }
        // 生成该 peer 的位图
```

我爱自由 5ifreedom.com

```
        peer->bitmap.bitfield_length = char_to_int(c) - 1;
        peer->bitmap.bitfield = (unsigned char*)malloc(peer->bitmap.bitfield_length);
        memcpy(peer->bitmap.bitfield,&buff[5],peer->bitmap.bitfield_length);

        // 如果原状态为已握手,收到位图后应该向 peer 发位图
        if(peer->state == HANDSHAKED) {
            create_bitfield_msg(bitmap->bitfield,bitmap->bitfield_length,peer);
            peer->state = DATA;
        }
        // 如果原状态为已发送位图，收到位图后可以进入 DATA 状态准备交换数据
        if(peer->state == SENDBITFIELD) {
            peer->state = DATA;
        }
        // 根据位图判断 peer 是否对本客户端感兴趣
        peer->peer_interested = is_interested(bitmap,&(peer->bitmap));
        // 判断对 peer 是否感兴趣,若是则发送 interested 消息
        peer->am_interested = is_interested(&(peer->bitmap), bitmap);
        if(peer->am_interested == 1) create_chock_interested_msg(2,peer);
    }

    peer->start_timestamp = time(NULL);
    return 0;
}
```

- int process_request_msg(Peer *peer,unsigned char *buff,int len)

功能：处理收到的 request 消息。函数实现代码如下：

```
int process_request_msg(Peer *peer,unsigned char *buff,int len)
{
    unsigned char   c[4];
    int             index, begin, length;
    Request_piece  *request_piece, *p;

    if(peer==NULL || buff==NULL)  return -1;
    if(peer->am_choking==0 && peer->peer_interested==1) {
        c[0] = buff[5];  c[1] = buff[6];
        c[2] = buff[7];  c[3] = buff[8];
        index = char_to_int(c);
        c[0] = buff[9];  c[1] = buff[10];
        c[2] = buff[11]; c[3] = buff[12];
        begin = char_to_int(c);
        c[0] = buff[13]; c[1] = buff[14];
        c[2] = buff[15]; c[3] = buff[16];
        length = char_to_int(c);

        // 查看该请求是否已存在,若已存在,则不进行处理
        p = peer->Requested_piece_head;
        while(p != NULL) {
            if(p->index==index && p->begin==begin && p->length==length) {
                break;
            }
            p = p->next;
        }
        if(p != NULL)  return 0;

        // 将请求加入到请求队列中
        request_piece = (Request_piece *)malloc(sizeof(Request_piece));
        if(request_piece == NULL)  {
            printf("%s:%d error",__FILE__,__LINE__);
            return 0;
        }
        request_piece->index  = index;
        request_piece->begin  = begin;
        request_piece->length = length;
        request_piece->next   = NULL;
        if( peer->Requested_piece_head == NULL )
            peer->Requested_piece_head = request_piece;
        else {
```

我爱自由 5ifreedom.com

```
            p = peer->Requested_piece_head;
            while(p->next != NULL)  p = p->next;
            p->next = request_piece;
        }
        // 打印提示信息
printf("***add a request FROM IP:%s index:%-6d begin:%-6x ***\n",
peer->ip,index,begin);
    }

    peer->start_timestamp = time(NULL);
    return 0;
}
```

- int process_piece_msg(Peer *peer,unsigned char *buff,int len)

功能：处理收到的 piece 消息。函数实现代码如下：

```
int process_piece_msg(Peer *peer,unsigned char *buff,int len)
{
    unsigned char   c[4];
    int             index, begin, length;
    Request_piece  *p;

    if(peer==NULL || buff==NULL)  return -1;
    if(peer->peer_choking==0) {
        c[0] = buff[0];    c[1] = buff[1];
        c[2] = buff[2];    c[3] = buff[3];
        length = char_to_int(c) - 9;
        c[0] = buff[5];    c[1] = buff[6];
        c[2] = buff[7];    c[3] = buff[8];
        index = char_to_int(c);
        c[0] = buff[9];    c[1] = buff[10];
        c[2] = buff[11];   c[3] = buff[12];
        begin = char_to_int(c);
        // 判断收到的 slice 是否是请求过的
        p = peer->Request_piece_head;
        while(p != NULL) {
            if(p->index==index && p->begin==begin && p->length==length)
                break;
            p = p->next;
        }
        if(p == NULL) {printf("did not found matched request\n"); return -1;}
        // 开始记时，并累计收到数据的字节数
        if(peer->last_down_timestamp == 0)
            peer->last_down_timestamp = time(NULL);
        peer->down_count += length;
        peer->down_total += length;
        // 将收到的数据写入缓冲区中
        write_slice_to_btcache(index,begin,length,buff+13,length,peer);
        // 生成请求数据的消息，要求继续发送数据
        create_req_slice_msg(peer);
    }

    peer->start_timestamp = time(NULL);
    return 0;
}
```

- int parse_response(Peer *peer)

功能：处理收到的消息（peer 的接收缓冲区中可能存放着多条消息）。函数实现代码如下：

```
int parse_response(Peer *peer)
{
    unsigned char   btkeyword[20];
    unsigned char   keep_alive[4] = { 0x0, 0x0, 0x0, 0x0 };
    int             index;
    unsigned char  *buff = peer->in_buff;  // in_buff 为接收缓冲区
    int             len = peer->buff_len;  // buff_len 为接收缓冲区中有效数据的长度

    if(buff==NULL || peer==NULL)  return -1;
```

我爱自由 5ifreedom.com

```
    btkeyword[0] = 19;
    memcpy(&btkeyword[1],"BitTorrent protocol",19);  // BitTorrent 协议关键字

    // 分别处理 12 种消息
    for(index = 0; index < len; ) {
        if( (len-index >= 68) && (memcmp(&buff[index],btkeyword,20) == 0) ) {
            process_handshake_msg(peer,buff+index,68);
            index += 68;
        }
        else if( (len-index >= 4) && (memcmp(&buff[index],keep_alive,4) == 0)) {
            process_keep_alive_msg(peer,buff+index,4);
            index += 4;
        }
        else if( (len-index >= 5) && (buff[index+4] == CHOKE) ) {
            process_choke_msg(peer,buff+index,5);
            index += 5;
        }
        else if( (len-index >= 5) && (buff[index+4] == UNCHOKE) ) {
            process_unchoke_msg(peer,buff+index,5);
            index += 5;
        }
        else if( (len-index >= 5) && (buff[index+4] == INTERESTED) ) {
            process_interested_msg(peer,buff+index,5);
            index += 5;
        }
        else if( (len-index >= 5) && (buff[index+4] == UNINTERESTED) ) {
            process_uninterested_msg(peer,buff+index,5);
            index += 5;
        }
        else if( (len-index >= 9) && (buff[index+4] == HAVE) ) {
            process_have_msg(peer,buff+index,9);
            index += 9;
        }
        else if( (len-index >= 5) && (buff[index+4] == BITFIELD) ) {
            process_bitfield_msg(peer,buff+index,peer->bitmap.bitfield_length+5);
            index += peer->bitmap.bitfield_length + 5;
        }
        else if( (len-index >= 17) && (buff[index+4] == REQUEST) ) {
            process_request_msg(peer,buff+index,17);
            index += 17;
        }
        else if( (len-index >= 13) && (buff[index+4] == PIECE) ) {
            unsigned char  c[4];
            int            length;

            c[0] = buff[index];    c[1] = buff[index+1];
            c[2] = buff[index+2];  c[3] = buff[index+3];
            length = char_to_int(c) - 9;

            process_piece_msg(peer,buff+index,length+13);
            index += length + 13;  // length+13 为 piece 消息的长度
        }
        else if( (len-index >= 17) && (buff[index+4] == CANCEL) ) {
            process_cancel_msg(peer,buff+index,17);
            index += 17;
        }
        else if( (len-index >= 7) && (buff[index+4] == PORT) ) {
            index += 7;
        }
        else {
            // 如果是未知的消息类型,则跳过不予处理
            unsigned char c[4];
            int           length;
            if(index+4 <= len) {
                c[0] = buff[index];   c[1] = buff[index+1];
                c[2] = buff[index+2]; c[3] = buff[index+3];
                length = char_to_int(c);
                if(index+4+length <= len)  { index += 4+length; continue; }
            }
            // 如果是一条错误的消息,清空接收缓冲区
            peer->buff_len = 0;
```

我爱自由 5ifreedom.com

```
            return -1;
        }
    } // for 语句结束

    // 接收缓冲区中的消息处理完毕后,清空接收缓冲区
    peer->buff_len = 0;
    return 0;
}
```

- int parse_response_uncomplete_msg(Peer *p,int ok_len)

功能：处理收到的消息。

参数：ok_len 为接收缓冲区中完整消息的长度。函数实现代码如下：

```
int parse_response_uncomplete_msg(Peer *p,int ok_len)
{
    char  *tmp_buff;
    int   tmp_buff_len;

    // 分配存储空间,并保存接收缓冲区中不完整的消息
    tmp_buff_len = p->buff_len - ok_len;
    if(tmp_buff_len <= 0)  return -1;
    tmp_buff = (char *)malloc(tmp_buff_len);
    if(tmp_buff == NULL) {
        printf("%s:%d error\n",__FILE__,__LINE__);
        return -1;
    }
    memcpy(tmp_buff,p->in_buff+ok_len,tmp_buff_len);
    // 处理接收缓冲区中前面完整的消息
    p->buff_len = ok_len;
    parse_response(p);
    // 将不完整的消息拷贝到接收缓冲区的开始处
    memcpy(p->in_buff,tmp_buff,tmp_buff_len);
    p->buff_len = tmp_buff_len;
    if(tmp_buff != NULL)  free(tmp_buff);

    return 0;
}
```

- int prepare_send_have_msg()

功能：为发送 have 消息作准备。

说明：当下载完一个 piece 时,应该向所有的 peer 发送 have 消息。函数实现的代码如下：

```
int prepare_send_have_msg()
{
    Peer *p = peer_head;
    int  i;

    if(peer_head == NULL)  return -1;
    if(have_piece_index[0] == -1)  return -1;

    while(p != NULL) {
        for(i = 0; i < 64; i++) {
            if(have_piece_index[i] != -1)  create_have_msg(have_piece_index[i],p);
            else  break;
        }
        p = p->next;
    }
    for(i = 0; i < 64; i++) {
        if(have_piece_index[i] == -1)  break;
        else  have_piece_index[i] = -1;
    }

    return 0;
}
```

- int create_response_message(Peer *peer)

功能：主动创建发送给 peer 的消息,而不是等收到某个消息后再创建响应消息。函数实现的代码如下：

我爱自由 5ifreedom.com

```
int create_response_message(Peer *peer)
{
    if(peer==NULL)  return -1;
    if(peer->state == INITIAL) {  // 处于 Intial 状态时主动发握手消息
        create_handshake_msg(info_hash,peer_id,peer);
        peer->state = HALFSHAKED;
        return 0;
    }
    if(peer->state == HANDSHAKED) {  // 处于已握手状态，主动发位图消息
        if(bitmap == NULL)  return -1;
        create_bitfield_msg(bitmap->bitfield,bitmap->bitfield_length,peer);
        peer->state = SENDBITFIELD;
        return 0;
    }
// 如果条件允许(未将该 peer 阻塞，且 peer 发送过请求)，则主动发送 piece 消息
    if( peer->am_choking==0 && peer->Requested_piece_head!=NULL ) {
        Request_piece *req_p = peer->Requested_piece_head;
        int ret = read_slice_for_send(req_p->index,req_p->begin,req_p->length,peer);
        if(ret < 0 ) { printf("read_slice_for_send ERROR\n");}
        else {
            if(peer->last_up_timestamp == 0)
                peer->last_up_timestamp = time(NULL);
            peer->up_count += req_p->length;
            peer->up_total += req_p->length;

            peer->Requested_piece_head = req_p->next;
            // 打印提示信息
// printf("*** sending a slice TO:%s index:%-5d begin:%-5x ***\n",
// peer->ip,req_p->index,req_p->begin);
            free(req_p);
            return 0;
        }
    }
    // 如果 3 分钟没有收到任何消息关闭连接
    time_t now = time(NULL);  // 获取当前时间
    long interval1 = now - peer->start_timestamp;
    if( interval1 > 180 ) {
        peer->state = CLOSING;
        // 丢弃发送缓冲区中的数据
        discard_send_buffer(peer);
        // 将从该 peer 处下载到的不足一个 piece 的数据删除
        clear_btcache_before_peer_close(peer);
        close(peer->socket);
    }
    // 如果 45 秒没有发送和接收到消息,则发送一个 keep_alive 消息
    long interval2 = now - peer->recet_timestamp;
    if( interval1>45 && interval2>45 && peer->msg_len==0)
        create_keep_alive_msg(peer);

    return 0;
}
```

- void discard_send_buffer(Peer *peer)

功能：即将与 peer 断开时，丢弃发送缓冲区中的消息。函数实现的代码如下：

```
void discard_send_buffer(Peer *peer)
{
    struct linger    lin;
    int            lin_len;

    lin.l_onoff = 1;
    lin.l_linger = 0;
    lin_len = sizeof(lin);

    // 通过设置套接字选项来丢弃未发送的数据
    if(peer->socket > 0) {
        setsockopt(peer->socket,SOL_SOCKET,SO_LINGER,(char *)&lin,lin_len);
    }
}
```

我爱自由 5ifreedom.com

13.4.8 缓冲管理模块的设计和实现

缓冲管理模块维护一个大小为 16MB 的缓冲区（大小可调整），将下载到的数据先保存在缓冲区中，在达到一定的数值时再将数据写入硬盘的文件中。peer 请求数据时，先在缓冲区中寻找，若缓冲区中不存在所请求的数据，则读文件并把请求数据所在的 piece 预先读入到缓冲区中。除了管理缓冲区，本模块还负责创建待下载的文件，把下载到的 piece 写入文件，在 peer 请求数据时读文件。本模块由 data.h 和 data.c 构成。

```
//data.h
#ifndef DATA_H
#define DATA_H
#include "peer.h"

// 每个 Btcache 结点维护一个长度为 16KB 的缓冲区，该缓冲区保存一个 slice 的数据
typedef struct _Btcache {
    unsigned char   *buff;          // 指向缓冲区的指针
    int             index;              // 数据所在的 piece 块的索引
    int             begin;              // 数据在 piece 块中的起始位置
    int             length;             // 数据的长度

    unsigned char   in_use;          // 该缓冲区是否在使用中
    unsigned char   read_write;     // 是发送给 peer 的数据还是接收到的数据
                                        // 若数据是从硬盘读出，read_write 值为 0
                                        // 若数据将要写入硬盘，read_write 值为 1
    unsigned char   is_full;        // 缓冲区是否满
    unsigned char   is_writed;      // 缓冲区中的数据是否已经写入到硬盘中
    int             access_count;       // 对该缓冲区的访问计数
    struct _Btcache *next;
} Btcache;

Btcache* initialize_btcache_node(); // 为 Btcache 结点分配内存空间并进行初始化
int create_btcache();               // 创建总大小为 16K*1024 即 16MB 的缓冲区
void release_memory_in_btcache();       // 释放 data.c 中动态分配的内存

int get_files_count();              // 获取种子文件中待下载的文件个数
int create_files();                     // 根据种子文件中的信息创建保存下载数据的文件

// 判断一个 Btcache 结点中的数据要写到哪个文件以及具体位置，并写入
int write_btcache_node_to_harddisk(Btcache *node);
// 从硬盘读出一个 slice 的数据存放到缓冲区中，在 peer 需要时发送给 peer
// 要读入的 slice 的索引，index、begin、length 已存到 node 所指向的结点中
int read_slice_from_harddisk(Btcache *node);
// 检查一个 piece 的数据是否正确，若正确则写入硬盘上的文件
int write_piece_to_harddisk(int sequence,Peer *peer);
// 从硬盘上的文件中读取一个 piece 存放到 p 指针所指向的缓冲区中
int read_piece_from_harddisk(Btcache *p, int index);

// 将整个缓冲区中已下载的数据写入到硬盘上的文件中
int write_btcache_to_harddisk(Peer *peer);
// 当缓冲区不够用时，释放那些从硬盘上读取的 piece
int release_read_btcache_node(int base_count);
// 从 btcache 缓冲区中清除那些未完成下载的 piece
void clear_btcache_before_peer_close(Peer *peer);
// 将刚刚从 peer 处获取的一个 slice 存放到缓冲区中
int write_slice_to_btcache(int index,int begin,int length,unsigned char *buff,int len,Peer
*peer);
// 从缓冲区获取一个 slice，读取的 slice 存放到 peer 的发送缓冲区中
int read_slice_for_send(int index,int begin,int length,Peer *peer);

// 以下是为下载和上传最后一个 piece 而增加的函数
// 最后一个 piece 较为特殊，因为它是一个不完整的 piece
int write_last_piece_to_btcache(Peer *peer);
int  write_slice_to_last_piece(int  index,int  begin,int  length,unsigned  char  *buff,int
```


我爱自由 5ifreedom.com

```
len,Peer *peer);
    int read_last_piece_from_harddisk(Btcache *p, int index);
    int read_slice_for_send_last_piece(int index,int begin,int length,Peer *peer);
    void release_last_piece();

    #endif
```

每个缓冲区结点的大小为 16KB，默认生成 1024 个结点，总大小为 16MB。缓冲区以一个 piece（通常为 256KB）为基本单位，也就是临近的 16 个结点为一组，这 16 个临近的结点要么全部被使用要么全部空闲。第 1～16 个结点存放一个 piece，第 17～32 个结点存放一个 piece，依此类推。为了方便处理，所有缓冲区在程序启动时统一申请，在程序结束时一起被释放。

以下是 data.c 文件的头部包含的代码，主要包含一些头文件和定义了一些全局变量。

```
//data.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <malloc.h>
#include "parse_metafile.h"
#include "bitfield.h"
#include "message.h"
#include "sha1.h"
#include "data.h"

extern  char   *file_name;              // 待下载文件的文件名
extern  Files  *files_head;         // 对于多文件种子有效，存放各个文件的路径和长度
extern  int     file_length;            // 待下载文件的总长度
extern  int     piece_length;           // 每个 piece 的长度
extern  char   *pieces;             // 存放所有 piece 的 hash 值
extern  int     pieces_length;          // 缓冲区 pieces 的长度

extern  Bitmap *bitmap;                 // 指向己方的位图
extern  int     download_piece_num;     // 记录已经下载了多少个 piece
extern  Peer   *peer_head;              // 指向 peer 链表

#define  btcache_len  1024          // 缓冲区中共有多少个 Btcache 结点
Btcache *btcache_head = NULL;           // 指向一个大小为 16MB 的缓冲区
Btcache *last_piece = NULL;             // 存放待下载文件的最后一个 piece
int     last_piece_index = 0;       // 最后一个 piece 的索引，它的值为总 piece 数减 1
int     last_piece_count = 0;       // 针对最后一个 piece，记录已下载了多少个 slice
int     last_slice_len   = 0;           // 最后一个 piece 的最后一个 slice 的长度

int *fds = NULL;                        // 存放文件描述符
int fds_len = 0;                    // 指针 fds 所指向的数组的长度
int have_piece_index[64];               // 存放刚刚下载到的 piece 的索引
int end_mode = 0;                   // 是否进入了终端模式，终端模式的含义参考 BT 协议
```

data.c 中各个函数的定义如下。

- Btcache* initialize_btcache_node()

功能：创建 Btcache 结点，分配内存空间并对其成员的值进行初始化。函数实现代码如下：

```
Btcache* initialize_btcache_node()
{
Btcache *node;

    node = (Btcache *)malloc(sizeof(Btcache));
    if(node == NULL)  { return NULL; }
    node->buff = (unsigned char *)malloc(16*1024);
    if(node->buff == NULL)  { if(node != NULL)  free(node);  return NULL; }

    node->index    = -1;
```

我爱自由 5ifreedom.com

```
    node->begin    = -1;
    node->length   = -1;

    node->in_use        = 0;
    node->read_write    = -1;
    node->is_full       = 0;
    node->is_writed     = 0;
    node->access_count= 0;
    node->next          = NULL;

    return node;
}
```

- int create-btcache( )

功能：创建总大小为 16K*1024bit 即 16MB 的缓冲区。函数实现代码如下：

```
int create_btcache()
{
    int     i;
    Btcache  *node, *last;  // node 指向刚刚创建的结点、last 指向缓冲区中最后一个结点

    for(i = 0; i < btcache_len; i++) {
        node = initialize_btcache_node();
        if( node == NULL )  {
            printf("%s:%d create_btcache error\n",__FILE__,__LINE__);
            release_memory_in_btcache();
            return -1;
        }
        if( btcache_head == NULL ) { btcache_head = node; last = node; }
        else  { last->next = node; last = node; }
    }
    // 为存储最后一个 piece 申请空间
    int count = file_length % piece_length / (16*1024);
    if(file_length % piece_length % (16*1024) != 0)  count++;
    last_piece_count = count;  // count 为最后一个 piece 所含的 slice 数
    last_slice_len = file_length % piece_length % (16*1024);
    if(last_slice_len == 0)  last_slice_len = 16*1024;
    last_piece_index = pieces_length / 20 -1;  // 最后一个 piece 的 index 值
    while(count > 0) {
        node = initialize_btcache_node();
        if(node == NULL) {
            printf("%s:%d create_btcache error\n",__FILE__,__LINE__);
            release_memory_in_btcache();
            return -1;
        }
        if(last_piece == NULL)  { last_piece = node; last = node; }
        else  { last->next = node; last = node; }

        count--;
    }

    for(i = 0; i < 64; i++) {
        have_piece_index[i] = -1;
    }
    return 0;
}
```

- void release_memory_in_btcache()

功能：释放 data.c 文件中动态分配的内存。函数实现代码如下：

```
void release_memory_in_btcache()
{
    Btcache *p = btcache_head;
    while(p != NULL) {
        btcache_head = p->next;
        if(p->buff != NULL) free(p->buff);
        free(p);
        p = btcache_head;
```

我爱自由 5ifreedom.com

```
    }

    release_last_piece();
    if(fds != NULL)  free(fds);
}
```

- void release_last_piece()

功能：释放为存储最后一个 piece 而申请的空间。函数实现代码如下：

```
void release_last_piece()
{
    Btcache *p = last_piece;
    while(p != NULL) {
        last_piece = p->next;
        if(p->buff != NULL) free(p->buff);
        free(p);
        p = last_piece;
    }
}
```

- int get_files_count()

功能：判断种子文件中待下载的文件个数。函数实现代码如下：

```
int get_files_count()
{
    int count = 0;

    if(is_multi_files() == 0)  return 1;
    Files *p = files_head;
    while(p != NULL) {
        count++;
        p = p->next;
    }

    return count;
}
```

- int create_files()

功能：根据种子文件中的信息创建保存下载数据的文件。通过 lseek 和 write 两个函数来实现物理存储空间的分配。函数实现代码如下：

```
int create_files()
{
    int  ret, i;
    char buff[1] = { 0x0 };

    fds_len = get_files_count();
    if(fds_len < 0)  return -1;
    fds = (int *)malloc(fds_len * sizeof(int));
    if(fds == NULL)  return -1;

    if( is_multi_files() == 0 ) {  // 待下载的为单文件
        *fds = open(file_name,O_RDWR|O_CREAT,0777);
        if(*fds < 0)  { printf("%s:%d error",__FILE__,__LINE__);  return -1; }
        ret = lseek(*fds,file_length-1,SEEK_SET);
        if(ret < 0)   { printf("%s:%d error",__FILE__,__LINE__);  return -1; }
        ret = write(*fds,buff,1);
        if(ret != 1)  { printf("%s:%d error",__FILE__,__LINE__);  return -1; }
    } else {  // 待下载的是多个文件
        ret = chdir(file_name);
        if(ret < 0) {  // 改变目录失败，说明该目录还未创建
            ret = mkdir(file_name,0777);
            if(ret < 0)  { printf("%s:%d error",__FILE__,__LINE__);  return -1; }
            ret = chdir(file_name);
            if(ret < 0)  { printf("%s:%d error",__FILE__,__LINE__);  return -1; }
        }
        Files *p = files_head;
        i = 0;
```

我爱自由 5ifreedom.com

```
        while(p != NULL) {
            fds[i] = open(p->path,O_RDWR|O_CREAT,0777);
            if(fds[i] < 0)  { printf("%s:%d error",__FILE__,__LINE__);  return -1; }
            ret = lseek(fds[i],p->length-1,SEEK_SET);
            if(ret < 0)     { printf("%s:%d error",__FILE__,__LINE__);  return -1; }
            ret = write(fds[i],buff,1);
            if(ret != 1)    { printf("%s:%d error",__FILE__,__LINE__);  return -1; }

            p = p->next;
            i++;
        } //while 循环结束
    } //end else

    return 0;
}
```

- int write_btcache_node_to_harddisk(Btcache *node)

功能：判断一个 Btcache 结点（即一个 slice）的数据要写到哪个文件以及具体位置，并写入。函数实现代码见附带光盘。

- int read_slice_from_harddisk(Btcache *node)

功能：从硬盘读一个 slice 的数据，存放到缓冲区中，在 peer 需要时发送给 peer。

注意：该函数非常类似于 write_btcache_node_to_harddisk，限于篇幅此处不再列出。

具体代码请参考本书所附源代码光盘。

- int write_piece_to_harddisk(int sequence,Peer *peer)

功能：检查下载完的一个 piece 的数据是否正确，若正确则写入文件。

参数：sequence 是存放 piece 的第一个 slice 的 Btcache 结点编号，该值的范围为 0～1023 中 16 的整数倍函数实现代码如下所示：

```
int write_piece_to_harddisk(int sequence,Peer *peer)
{
    Btcache      *node_ptr = btcache_head, *p;
    unsigned char   piece_hash1[20], piece_hash2[20];
    int          slice_count = piece_length / (16*1024); // 一个piece所含的slice数
    int          index, index_copy;

    if(peer==NULL) return -1;
    int i = 0;
    while(i < sequence)  { node_ptr = node_ptr->next; i++; }
    p = node_ptr;  // p指针指向piece的第一个slice所在的btcache结点

    // 计算刚刚下载到的这个piece的hash值
    SHA1_CTX ctx;
    SHA1Init(&ctx);
    while(slice_count>0 && node_ptr!=NULL) {
        SHA1Update(&ctx,node_ptr->buff,16*1024);
        slice_count--;
        node_ptr = node_ptr->next;
    }
    SHA1Final(piece_hash1,&ctx);
    // 从种子文件中获取该piece的正确的hash值
    index = p->index * 20;
    index_copy = p->index;  // 存放piece的index
    for(i = 0; i < 20; i++)  piece_hash2[i] = pieces[index+i];
    // 比较两个hash值，若两者一致说明下载了一个正确的piece
    int ret = memcmp(piece_hash1,piece_hash2,20);
    if(ret != 0)  { printf("piece hash is wrong\n"); return -1; }
    // 将该piece的所有slice写入文件
    node_ptr = p;
    slice_count = piece_length / (16*1024);
    while(slice_count > 0) {
        write_btcache_node_to_harddisk(node_ptr);
```

我爱自由 5ifreedom.com

```
        // 在 peer 的请求队列中删除 piece 请求
        Request_piece *req_p = peer->Request_piece_head;
        Request_piece *req_q = peer->Request_piece_head;
        while(req_p != NULL) {
            if(req_p->begin==node_ptr->begin && req_p->index==node_ptr->index)
            {
                if(req_p == peer->Request_piece_head)
                    peer->Request_piece_head = req_p->next;
                else
                    req_q->next = req_p->next;
                free(req_p);
                req_p = req_q = NULL;
                break;
            }
            req_q = req_p;
            req_p = req_p->next;
        }

        node_ptr->index     = -1;
        node_ptr->begin     = -1;
        node_ptr->length    = -1;
        node_ptr->in_use        = 0;
        node_ptr->read_write     = -1;
        node_ptr->is_full        = 0;
        node_ptr->is_writed     = 0;
        node_ptr->access_count    = 0;
        node_ptr = node_ptr->next;
        slice_count--;
    }
    // 当前处于终端模式，则在 peer 链表中删除所有对该 piece 的请求
    if(end_mode == 1)  delete_request_end_mode(index_copy);
    // 更新位图
    set_bit_value(bitmap,index_copy,1);
    // 保存 piece 的 index，准备给所有的 peer 发送 have 消息
    for(i = 0; i < 64; i++) {
        if(have_piece_index[i] == -1) {
            have_piece_index[i] = index_copy;
            break;
        }
    }
    // 更新 download_piece_num，每下载 10 个 piece 就将位图写入文件
    download_piece_num++;
    if(download_piece_num % 10 == 0)  restore_bitmap();
    // 打印出提示信息
    printf("%%%%%% Total piece download:%d %%%%%%\n",download_piece_num);
    printf("writed piece index:%d \n",index_copy);
    return 0;
}
```

- int read_piece_from_harddisk(Btcache *p, int index)

功能：从硬盘上的文件中读取一个 piece 到 p 指针所指向的缓冲区中。函数实现的代码见附带光盘。

- int write_btcache_to_harddisk(Peer *peer)

功能：将整个缓冲区中已下载的 piece 写入硬盘，这样可以释放缓冲区。函数实现的代码如下：

```
int write_btcache_to_harddisk(Peer *peer)
{
    Btcache    *p            = btcache_head;
    int         slice_count     = piece_length / (16*1024);
    int         index_count     = 0;
    int         full_count      = 0;
    int         first_index;

    while(p != NULL) {
```

我爱自由 5ifreedom.com

```
        if(index_count % slice_count == 0) {
            full_count = 0;
            first_index = index_count;
        }
        if( (p->in_use  == 1) && (p->read_write == 1) &&
            (p->is_full == 1) && (p->is_writed  == 0) ) {
            full_count++;
        }
        if(full_count == slice_count) {
            write_piece_to_harddisk(first_index,peer);
        }
        index_count++;
        p = p->next;
    }

    return 0;
}
```

- int release_read_btcache_node(int base_count)

功能：当缓冲区不够用时，释放那些从硬盘上读取的 piece。函数实现代码如下：

```
int release_read_btcache_node(int base_count)
{
    Btcache     *p = btcache_head;
    Btcache     *q = NULL;
    int         count = 0;
    int         used_count = 0;
    int         slice_count = piece_length / (16*1024);

    if(base_count < 0)  return -1;
    while(p != NULL) {
        if(count % slice_count == 0)  { used_count = 0; q = p; }
        if(p->in_use==1 && p->read_write==0)  used_count += p->access_count;
        if(used_count == base_count)  break;  // 找到一个空闲的piece

        count++;
        p = p->next;
    }

    if(p != NULL) {
        p = q;
        while(slice_count > 0) {
            p->index    = -1;
            p->begin    = -1;
            p->length   = -1;
            p->in_use          = 0;
            p->read_write      = -1;
            p->is_full         = 0;
            p->is_writed       = 0;
            p->access_count    = 0;

            slice_count--;
            p = p->next;
        }
    }

    return 0;
}
```

- int is_a_complete_piece(int index, int *sequence)

功能：下载完一个 slice 后，检查是否该 slice 为 piece 的最后一个 slice。函数实现的代码见附带光盘。

- void clear_btcache()

功能：将整个缓冲区中所存的所有数据清空。函数实现的代码如下：

```
void clear_btcache()
```

我爱自由 5ifreedom.com

```
{
    Btcache *node = btcache_head;
    while(node != NULL) {
        p->index    = -1;
        p->begin    = -1;
        p->length   = -1;
        p->in_use          = 0;
        p->read_write      = -1;
        p->is_full         = 0;
        p->is_writed       = 0;
        p->access_count    = 0;
        node = node->next;
    }
}
```

- write_slice_to_btcache(int index,int begin,int length,unsigned char *buff,int len,Peer *peer)

功能：将从 peer 处获取的一个 slice 存储到缓冲区中。函数实现的代码如下：

```
int write_slice_to_btcache(int index,int begin,int length,unsigned char *buff,int len,Peer *peer)
{
    int     count = 0, slice_count, unuse_count;
    Btcache *p = btcache_head, *q = NULL;  // q 指针指向每个 piece 第一个 slice

    if(p == NULL)  return -1;
    if(index>=pieces_length/20 || begin>piece_length-16*1024)  return -1;
    if(buff==NULL || peer==NULL)  return -1;
    if(index == last_piece_index) {
        write_slice_to_last_piece(index,begin,length,buff,len,peer);
        return 0;
    }
    // 当处于终端模式时，先判断该 slice 所在的 piece 是否已被下载
    if(end_mode == 1) {
        if( get_bit_value(bitmap,index) == 1 )  return 0;
    }
    // 遍历缓冲区，检查当前 slice 所在 piece 的其他数据是否已存在
    // 若存在说明不是一个新的 piece,若不存在说明是一个新的 piece
    slice_count = piece_length / (16*1024);
    while(p != NULL) {
        if(count%slice_count == 0)  q = p;
        if(p->index==index && p->in_use==1)  break;

        count++;
        p = p->next;
    }

    // p 非空说明当前 slice 所在 piece 的部分数据已经下载
    if(p != NULL) {
        count = begin / (16*1024);  // count 存放当前要存的 slice 在 piece 中的索引值
        p = q;
        while(count > 0)  { p = p->next; count--; }

        if(p->begin==begin && p->in_use==1 && p->read_write==1 && p->is_full==1)
            return 0; // 该 slice 已存在

        p->index    = index;
        p->begin    = begin;
        p->length = length;

        p->in_use          = 1;
        p->read_write      = 1;
        p->is_full         = 1;
        p->is_writed       = 0;
        p->access_count    = 0;

        memcpy(p->buff,buff,len);
printf("+++++ write a slice to btcache index:%-6d begin:%-6x ++++\n",
index,begin);
```

我爱自由 5ifreedom.com

```
        // 如果是刚刚开始下载(下载到的piece不足10个),则立即写入硬盘并告知peer
        if(download_piece_num < 10) {
            int sequence;
            int ret;
            ret = is_a_complete_piece(index,&sequence);
            if(ret == 1) {
printf("###### begin write a piece to harddisk ######\n");
                write_piece_to_harddisk(sequence,peer);
printf("###### end  write a piece to harddisk ######\n");
            }
        }
        return 0;
    }

    // p为空说明当前slice是其所在的piece第一块下载到的数据
    // 首先判断是否存在空的缓冲区,若不存在,则将已下载的写入硬盘
    int i = 4;
    while(i > 0) {
        slice_count = piece_length / (16*1024);
        count = 0;          // 计数当前指向第几个slice
        unuse_count = 0;     // 计数当前piece中有多少个空的slice
        Btcache *q;
        p = btcache_head;
        while(p != NULL) {
            if(count%slice_count == 0)  { unuse_count = 0; q = p; }
            if(p->in_use == 0) unuse_count++;
            if(unuse_count == slice_count)  break;  // 找到一个空闲的piece

            count++;
            p = p->next;
        }

        if(p != NULL) {
            p = q;
            count = begin / (16*1024);
            while(count > 0)  { p = p->next; count--; }

            p->index    = index;
            p->begin    = begin;
            p->length = length;

            p->in_use       = 1;
            p->read_write    = 1;
            p->is_full       = 1;
            p->is_writed     = 0;
            p->access_count = 0;

            memcpy(p->buff,buff,len);
            printf("+++++ write a slice to btcache index:%-6d begin:%-6x +++++\n",
            index,begin);
            return 0;
        }

        if(i == 4) write_btcache_to_harddisk(peer);
        if(i == 3) release_read_btcache_node(16);
        if(i == 2) release_read_btcache_node(8);
        if(i == 1) release_read_btcache_node(0);
        i--;
    }

    // 如果还没有空闲的缓冲区,丢弃下载到的这个slice
    printf("+++++ write a slice to btcache FAILED :NO BUFFER +++++\n");
    clear_btcache();
    return 0;
}
```

- int read_slice_for_send(int index,int begin,int length,Peer *peer)

功能：从缓冲区获取一个 slice，读取的 slice 存放到 peer 的发送缓冲区中。若缓冲区中不存

我爱自由 5ifreedom.com

在该 slice，则从硬盘读 slice 所在的 piece 到缓冲区中。函数实现的代码见附带光盘。

由于 data.c 中其他函数较为简单或与已列出的函数类似，限于篇幅不再列出，请参考本书所附的源代码光盘。

13.4.9 策略管理模块的设计和实现

策略管理模块负责策略的实现，主要是计算各个 peer 的下载和上传速度，根据下载速度选择非阻塞 peer，采用随机算法选择优化非阻塞 peer，以及实现片断选择策略。本模块由 policy.h 和 policy.c 构成。

BT 协议的设计者也承认计算从各个 peer 处下载数据的速度是一个棘手的问题。经过分析和对比，现在通用的计算下载速度的方法是每 10 秒计算一次速度，统计最近 10 秒内从每个 peer 处下载的数据量，然后除以时间，得到最近这段时间的下载速度，并将下载速度最快的 4 个 peer 解除阻塞，允许它们从本客户端下载，除一个特殊的 peer 外其他 peer 将被阻塞。为了发现更快下载速度的 peer，任何时刻保证存在一个优化非阻塞 peer，将这个 peer 解除阻塞，而暂时不管从该 peer 处下载数据的速度，每隔 30 秒重新进行选择。在这 30 秒内，本客户端提供给该 peer 较快的下载速度，然后该 peer 将本客户端解除阻塞，这样就可以从该 peer 处下载数据，并在下次选择非阻塞 peer 时，该 peer 能成为 4 个非阻塞 peer 中的一个。片断选择策略，也就是选择下载哪些 slice，请参考前面部分的 BT 协议。

```
//policy.h
#ifndef POLICY_H
#define POLICY_H
#include "peer.h"

#define COMPUTE_RATE_TIME 10                // 每隔 10 秒计算一次各个 peer 的下载和上传速度
#define UNCHOKE_COUNT  4                    // 非阻塞 peer 的个数
#define REQ_SLICE_NUM  4                    // 每次请求 slice 的个数

typedef struct _Unchoke_peers {
    Peer*  unchkpeer[UNCHOKE_COUNT];        // 保存非阻塞 peer 的指针
    int    count;                           // 记录当前有多少个非阻塞 peer
    Peer*  optunchkpeer;                    // 保存优化非阻塞 peer 的指针
} Unchoke_peers;

void init_unchoke_peers();                  // 初始化 plicy.c 中定义的全局变量 unchoke_peers
int select_unchoke_peer();                  // 选择 unchoke peer
int select_optunchoke_peer();               // 从 peer 队列中选择一个优化非阻塞 peer
int compute_rate();                         // 计算最近一段时间(10s)每个 peer 的上传下载速度
int compute_total_rate();                   // 计算总的上传下载速度

int is_seed(Peer *node);                    // 判断某个 peer 是否为种子
int create_req_slice_msg(Peer *node);       // 构造数据请求

#endif
```

policy.c 文件的头部包含的代码为：

```
policy.c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "parse_metafile.h"
#include "peer.h"
#include "data.h"
#include "message.h"
#include "policy.h"

long long        total_down = 0L, total_up = 0L;    // 总的下载量和上传量
```

我爱自由 5ifreedom.com

```
float           total_down_rate = 0.0F, total_up_rate = 0.0F; // 总的下载上传速度
int             total_peers = 0;                    // 已连接的总 Peer 数
Unchoke_peers   unchoke_peers;                  // 存放非阻塞 Peer 和优化非阻塞 Peer 的指针

extern int          end_mode;                   // 是否已进入终端模式
extern Bitmap   *bitmap;                        // 指向己方的位图
extern Peer     *peer_head;                     // 指向 Peer 链表
extern int       pieces_length;                 // 所有 piece hash 值的长度
extern int       piece_length;                  // 每个 piece 的长度

extern Btcache  *btcache_head;              // 指向存放下载数据的缓冲区
extern int       last_piece_index;          // 最后一个 piece 的 index
extern int       last_piece_count;          // 最后一个 piece 所含的 slice 数
extern int       last_slice_len;            // 最后一个 piece 的最后一个 slice 的长度
extern int       download_piece_num;        // 已下载的 piece 数
```

policy.c 文件中各个函数的定义如下。

- void init_unchoke_peers()

功能：初始化全局变量 unchoke_peers。函数实现的代码见附带光盘。

- int is_in_unchoke_peers(Peer *node)

功能：判断一个 peer 是否已经存在于 unchoke_peers。函数实现的代码见附带光盘。

- int get_last_index(Peer **array,int len)

功能：从 unchoke_peers 中获取下载速度最慢的 peer 的索引。函数实现的代码见附带光盘。

- int select_unchoke_peer()

功能：找出当前下载速度最快的 4 个 peer，将其解除阻塞。函数实现的代码见附带光盘。

- int get_rand_numbers(int length)

功能：假设要下载的文件共有 100 个 piece，本函数的功能是将 0～99 这 100 个数的顺序以随机的方式打乱，从而得到一个随机的数组，该数组以随机的方式存储 0～99，供片断选择算法使用。具体实现代码如下所示：

```
int *rand_num = NULL;
int get_rand_numbers(int length)
{
    int i, index, piece_count, *temp_num;

    if(length == 0)  return -1;
    piece_count = length;

    rand_num = (int *)malloc(piece_count * sizeof(int));
    if(rand_num == NULL)    return -1;

    temp_num = (int *)malloc(piece_count * sizeof(int));
    if(temp_num == NULL)    return -1;
    for(i = 0; i < piece_count; i++)  temp_num[i] = i;

    srand(time(NULL));
    for(i = 0; i < piece_count; i++) {
        index = (int)( (float)(piece_count-i) * rand() / (RAND_MAX+1.0) );
        rand_num[i] = temp_num[index];
        temp_num[index] = temp_num[piece_count-1-i];
    }

    if(temp_num != NULL)  free(temp_num);
    return 0;
}
```

- int select_optunchoke_peer()

功能：从 peer 队列中随机选择一个 peer 作为优化非阻塞 peer。函数实现代码见附带光盘。

- int compute_rate()

我爱自由 5ifreedom.com

功能：计算最近一段时间（如 10 秒）每个 peer 的上传下载速度。函数实现代码如下：

```
int compute_rate()
{
    Peer    *p       = peer_head;
    time_t  time_now = time(NULL);
    long    t        = 0;

    while(p != NULL) {
        if(p->last_down_timestamp == 0) {
            p->down_rate  = 0.0f;
            p->down_count = 0;
        } else {
            t = time_now - p->last_down_timestamp;
            if(t == 0)  printf("%s:%d time is 0\n",__FILE__,__LINE__);
            else  p->down_rate = p->down_count / t;
            p->down_count = 0;
            p->last_down_timestamp = 0;
        }

        if(p->last_up_timestamp == 0) {
            p->up_rate  = 0.0f;
            p->up_count = 0;
        } else {
            t = time_now - p->last_up_timestamp;
            if(t == 0)  printf("%s:%d time is 0\n",__FILE__,__LINE__);
            else  p->up_rate = p->up_count / t;
            p->up_count = 0;
            p->last_up_timestamp = 0;
        }

        p = p->next;
    }
    return 0;
}
```

- int compute_total_rate()

功能：计算总的下载和上传速度。函数实现代码如下：

```
int compute_total_rate()
{
Peer *p = peer_head;

    total_peers = 0;
    total_down = 0;
    total_up = 0;
    total_down_rate = 0.0f;
    total_up_rate = 0.0f;

    while(p != NULL) {
        total_down += p->down_total;
        total_up += p->up_total;
        total_down_rate += p->down_rate;
        total_up_rate += p->up_rate;

        total_peers++;
        p = p->next;
    }
    return 0;
}
```

- int is_seed(Peer *node)

功能：根据位图判断某 Peer 是否为种子，若各个位都为 1，则说明该 Peer 为种子。函数实现代码见附带光盘。

- int create_req_slice_msg(Peer *node)

功能：生成 request 消息，实现片断选择算法（request 消息的长度固定为 17Byte）。函数实

我爱自由 5ifreedom.com

现代码如下：

```
int create_req_slice_msg(Peer *node)
{
    int  index, begin, length = 16*1024;
    int  i, count = 0;

    if(node == NULL)  return -1;
    // 如果被 peer 阻塞或对 peer 不感兴趣，就没有必要生成 request 消息
    if(node->peer_choking==1 || node->am_interested==0 )  return -1;

    // 如果之前向该 peer 发送过请求，则根据之前的请求构造新请求
    // 遵守一条原则：同一个 piece 的所有 slice 应该尽可能地从同一个 peer 处下载
    Request_piece *p = node->Request_piece_head, *q = NULL;
    if(p != NULL) {
        while(p->next != NULL)  { p = p->next; }     // 定位到最后一个结点处
        int last_begin = piece_length - 16*1024; // 一个 piece 的最后一个 slice 的起始下标
        if(p->index == last_piece_index) { // 如果是最后一个 piece
            last_begin = (last_piece_count - 1) * 16 * 1024;
        }

        // 当前 piece 还有未请求的 slice，则构造请求消息
        if(p->begin < last_begin) {
            index = p->index;
            begin = p->begin + 16*1024;
            count = 0;

            while(begin!=piece_length && count<1) {
                // 如果是最后一个 piece 的最后一个 slice
                if(p->index == last_piece_index) {
                    if( begin == (last_piece_count - 1) * 16 * 1024 )
                        length = last_slice_len;
                }
                // 创建 request 消息
                create_request_msg(index,begin,length,node);
                // 将当前的请求记录到请求队列
                q = (Request_piece *)malloc(sizeof(Request_piece));
                if(q == NULL) {
                    printf("%s:%d error\n",__FILE__,__LINE__);
                    return -1;
                }
                q->index    = index;
                q->begin    = begin;
                q->length = length;
                q->next     = NULL;
                p->next     = q;
                p          = q;
                begin  += 16*1024;
                count++;
            } // end while
            return 0;  // 构造完毕,就返回
        } // end if(p->begin < last_begin)
    } // end if(p != NULL)

    // 开始对一个未请求过的 piece 发出请求
    if(get_rand_numbers(pieces_length/20) == -1) {  // 生成随机数
        printf("%s:%d error\n",__FILE__,__LINE__);
        return -1;
    }
    // 随机选择一个 piece 的下标,该下标所代表的 piece 应该没有向任何 peer 请求过
    for(i = 0; i < pieces_length/20; i++) {
        index = rand_num[i];
        // 判断对于以 index 为下标的 piece,peer 是否拥有
        if( get_bit_value(&(node->bitmap),index) != 1)  continue;
        // 判断对于以 index 为下标的 piece,是否已经下载
        if( get_bit_value(bitmap,index) == 1) continue;
        // 判断对于以 index 为下标的 piece,是否已经请求过了
```


我爱自由 5ifreedom.com

```
        Peer            *peer_ptr = peer_head;
        Request_piece    *reqt_ptr;
        int             find = 0;
        while(peer_ptr != NULL) {
            reqt_ptr = peer_ptr->Request_piece_head;
            while(reqt_ptr != NULL) {
                if(reqt_ptr->index == index)  { find = 1; break; }
                reqt_ptr = reqt_ptr->next;
            }
            if(find == 1) break;
            peer_ptr = peer_ptr->next;
        }
        if(find == 1) continue;
        break; // 程序若执行到此处,说明已经找到一个符合要求的 index
    }
    * 如果还未找到一个合适的 index，说明所有的 piece 要么已经被下载要么正在被请求*下载，而此时还有多余的对
本客户端解除阻塞的 peer，说明已经进入终端模式，即将下载完成*/
    if(i == pieces_length/20) {
        if(end_mode == 0)  end_mode = 1;
        for(i = 0; i < pieces_length/20; i++) {
            if( get_bit_value(bitmap,i) == 0 )  { index = i; break; }
        }
        if(i == pieces_length/20) {
            printf("Can not find an index to IP:%s\n",node->ip);
            return -1;
        }
    }

    // 构造 piece 请求消息
    begin = 0;
    count = 0;
    p = node->Request_piece_head;
    if(p != NULL)
        while(p->next != NULL)  p = p->next;
        while(count < 4) {
        // 如果是构造最后一个 piece 的请求消息
        if(index == last_piece_index) {
            if(count+1 > last_piece_count)
                break;
            if(begin == (last_piece_count - 1) * 16 * 1024)
                length = last_slice_len;
        }
        // 创建 request 消息
        create_request_msg(index,begin,length,node);
        // 将请求记录到请求队列
        q = (Request_piece *)malloc(sizeof(Request_piece));
        if(q == NULL) { printf("%s:%d error\n",__FILE__,__LINE__); return -1; }
        q->index  = index;
        q->begin  = begin;
        q->length = length;
        q->next   = NULL;
        if(node->Request_piece_head == NULL)  {
        node->Request_piece_head = q;
        p = q;
    }
        else  { p->next = q; p = q; }
        begin += 16*1024;
        count++;
    }

    if(rand_num != NULL)  { free(rand_num); rand_num = NULL; }
    return 0;
}
```

13.4.10 连接 Tracker 模块的设计和实现

连接 Tracker 模块的主要功能是：构造 HTTP 请求，请求 Tracker 服务器发送 peer 的 IP 地址

我爱自由 5ifreedom.com

和端口号；与 Tracker 建立连接；解析从 Tracker 返回的数据。Tracker 返回的数据是经过 B 编码的，解析 Tracker 的回应和解析种子文件是类似的。本模块由 tarcker.h 和 tracker.c 构成。

```
tracker.h
#ifndef  TRACKER_H
#define  TRACKER_H
#include <netinet/in.h>
#include "parse_metafile.h"

typedef struct _Peer_addr {
    char          ip[16];
    unsigned short    port;
    struct _Peer_addr *next;
} Peer_addr;

// 用于将 info_hash 和 peer_id 转换为 HTTP 编码格式
int http_encode(unsigned char *in,int len1,char *out,int len2);
// 从种子文件中存储的 Tracker 的 URL 获取 Tracker 主机名
int get_tracker_name(Announce_list *node,char *name,int len);
// 从种子文件中存储的 Tracker 的 URL 获取 Tracker 端口号
int get_tracker_port(Announce_list *node,unsigned short *port);

// 构造发送到 Tracker 服务器的 HTTP GET 请求
int create_request(char *request, int len,Announce_list *node,
            unsigned short port,long long down,long long up,
            long long left,int numwant);

int prepare_connect_tracker(int *max_sockfd);  // 以非阻塞的方式连接 Tracker
int prepare_connect_peer(int *max_sockfd);     // 以非阻塞的方式连接 peer

// 获取 Tracker 返回的消息类型
int get_response_type(char *buffer,int len,int *total_length);
// 解析第一种 Tracker 返回的消息
int parse_tracker_response1(char *buffer,int ret,char *redirection,int len);
// 解析第二种 Tracker 返回的消息
int parse_tracker_response2(char *buffer,int ret);
// 为已建立连接的 peer 创建 peer 结点并加入到 peer 链表中
int add_peer_node_to_peerlist(int *sock,struct sockaddr_in saptr);
// 释放 peer_addr 指向的链表
void free_peer_addr_head();

#endif
```

tracker.c 文件的头部包括的代码如下。

```
tracker.c
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <errno.h>
#include <ctype.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include "parse_metafile.h"
#include "peer.h"
#include "tracker.h"

extern unsigned char     info_hash[20];        // 存放 info_hash
```

我爱自由 5ifreedom.com

```
extern unsigned char      peer_id[20];          // 存放 peerv_id
extern Announce_list  *announce_list_head;      // 存放各个 Tracker 的 URL

extern int                *sock;                // 连接 Tracker 的套接字
extern struct sockaddr_in      *tracker;        // 连接 Tracker 时使用
extern int                 *valid;              // 指示连接 Tracker 的状态
extern int                   tracker_count;     // 为 Tracker 服务器的个数

extern int                 *peer_sock;          // 连接 peer 的套接字
extern struct sockaddr_in      *peer_addr;      // 连接 peer 时使用
extern int                  *peer_valid;        // 指示连接 peer 的状态
extern int                   peer_count;            // 尝试与多少个 peer 建立连接
Peer_addr  *peer_addr_head = NULL;
```

tracker.c 文件各个函数定义如下。

- int http_encode(unsigned char *in,int len1,char *out,int len2)

功能：进行编码转换，根据 HTTP 协议，HTTP 请求中的非数字和非字母都要进行编码转换。例如，空格符既不属于 0～9 也不属于 a～z、A~Z，需要进行转换，它的 ASCII 码为 0×20，转换为字符串“%20”。本函数较为简单，代码不列出，可参考本书附带光盘。

- int get_tracker_name(Announce_list *node,char *name,int len)

功能：获取 Tracker URL 中的主机名部分。例如：http://btfans.3322.org:8000/announce 是一个 Tracker 的 URL，本函数可获取其主机名为“btfans.3322.org”。本函数较为简单，代码不列出，可参考本书附带光盘。

- int get_tracker_port(Announce_list *node,unsigned short *port)

功能：获取 Tracker URL 中的端口号。例如：http://btfans.3322.org:8000/announce 是一个 Tracker 的 URL，本函数可获取其端口号即 8000。本函数较为简单，代码不列出，可参考本书附带光盘。

- int create_request(...)

功能：构造发送到 Tracker 服务器的 HTTP GET 请求。

参数：request 用于接收生成的请求；len 为 request 所指向的数组的长度；node 为指向 Tracker 的 URL；port 为监听的端口号；down 为已下载的数据量；up 为已上传的数据量；left 为剩余多少字节未下载；numwant 是希望 Tracker 返回的 Peer 数，函数实现的代码如下：

```
int create_request(char *request,int len,Announce_list *node,
            unsigned short port,long long down,long long up,
            long long left,int numwant)
{
    char              encoded_info_hash[100];
    char              encoded_peer_id[100];
    int               key;
    char             tracker_name[128];
    unsigned short        tracker_port;

    http_encode(info_hash,20,encoded_info_hash,100);
    http_encode(peer_id,20,encoded_peer_id,100);

    srand(time(NULL));
    key = rand() / 10000;    // 获取一个 0～9999 之间的随机数

    get_tracker_name(node,tracker_name,128);
    get_tracker_port(node,&tracker_port);

    sprintf(request,
    "GET /announce?info_hash=%s&peer_id=%s&port=%u"
    "&uploaded=%lld&downloaded=%lld&left=%lld"
```

我爱自由 5ifreedom.com

```
    "&event=started&key=%d&compact=1&numwant=%d HTTP/1.0\r\n"
    "Host: %s\r\nUser-Agent: Bittorrent\r\nAccept: */*\r\n"
    "Accept-Encoding: gzip\r\nConnection: closed\r\n\r\n",
    encoded_info_hash,encoded_peer_id,port,up,down,left,
    key,numwant,tracker_name);

#ifdef DEBUG
    printf("request:%s\n",request);
#endif
    return 0;
}
```

- int get_response_type(char *buffer,int len,int *total_length)

功能：获取 Tracker 返回的消息的类型。

参数：buffer 指向 Tracker 的回应消息；len 为 buffer 所指向的数组的长度；total_length 用于存放 Tracker 返回数据的长度，函数实现的代码如下：

```
int get_response_type(char *buffer,int len,int *total_length)
{
    int i, content_length = 0;

    for(i = 0; i < len-7; i++) {
        if(memcmp(&buffer[i],"5:peers",7) == 0) {
            i = i+7;
            break;
        }
    }
    // 如果返回的消息不含"5:peers"关键字，则没有返回 peer 的 IP 地址及端口号
    if(i == len-7)  return -1;
    // 关键字"5:peers"之后如果是字符 '1'，则说明返回的消息为第一种类型
    if(buffer[i] != '1')  return 0;
    *total_length = 0;
    for(i = 0; i < len-16; i++) {
        if(memcmp(&buffer[i],"Content-Length: ",16) == 0) {
            i = i+16;
            break;
        }
    }
    if(i != len-16) {
        while(isdigit(buffer[i])) {
            content_length = content_length * 10 + (buffer[i] - '0');
            i++;
        }
        for(i = 0; i < len-4; i++) {
            if(memcmp(&buffer[i],"\r\n\r\n",4) == 0)  { i = i+4; break; }
        }
        if(i != len-4)  *total_length = content_length + i;
    }

    if(*total_length == 0)  return -1;
    else return 1;
}
```

程序说明。

（1）第一种类型的 Tracker 回应为：关键字“5:peers”之后是一个 B 编码的字符串，该字符串以 6 个字节为一组，前面 4 个存放一个 peer 的 IP 地址，后面两个存放该 peer 的端口号。

例如：“d10:done peersi85e8:intervali1800e9:num peersi214e5:peers600:..”。Tracker 返回的是一个字典，关键字“10:done peers”对应值为种子数；关键字“8:interval”对应值为 Tracker 希望多长时间连接一次 Tracker，一般为 1800 秒；关键字“9:num peers”为当前在下载的 peer 个数；关键字“5:peers”为返回的各个 peer 的 IP 地址和端口号。

（2）第二种类型的 Tracker 回应为：关键字“5:peers”之后为一个 B 编码的列表。列表中每

我爱自由 5ifreedom.com

个元素的类型都是字典，一个字典用于表示一个 peer。

例如："d10:done peersi2e8:intervali1800e9:num peersi4e5:peersld2:ip11:83.72.54.24 7:peer id 20:M3-4-2--2cd992318ca4:port=>i6882eed2:ip13:80.15.205.1907:peer id 20:-AZ2104-lN5SvwK0XgRt4:porti92eee"。

"5:peers"之后为字典的起始符'd'，接着依次是 2:ip=>11:83.72.54.24，7:peer id=>20:M3-4-2--2cd992318ca，4:port=>i6882e，然后是字典终止符'e'；之后又是一个字典，2:ip=>13:80.15.205.190，7:peer id=>20:-AZ2104-lN5SvwK0XgRt，4:port=>i92e。

- int prepare_connect_tracker(int *max_sockfd)

功能：以非阻塞的方式连接 Tracker，函数实现的代码如下：

```
int prepare_connect_tracker(int *max_sockfd)
{
    int                 i, flags, ret, count = 0;
    struct hostent          *ht;
    Announce_list       *p = announce_list_head;

    while(p != NULL)  { count++; p = p->next; }
    tracker_count = count;
    sock = (int *)malloc(count * sizeof(int));
    if(sock == NULL)  goto OUT;
    tracker = (struct sockaddr_in *)malloc(count * sizeof(struct sockaddr_in));
    if(tracker == NULL)  goto OUT;
    valid = (int *)malloc(count * sizeof(int));
    if(valid == NULL)  goto OUT;

    p = announce_list_head;
    for(i = 0; i < count; i++) {
        char            tracker_name[128];
        unsigned short        tracker_port = 0;

        sock[i] = socket(AF_INET,SOCK_STREAM,0);
        if(sock < 0) {
            printf("%s:%d socket create failed\n",__FILE__,__LINE__);
            valid[i] = 0;
            p = p->next;
            continue;
        }

        get_tracker_name(p,tracker_name,128);
        get_tracker_port(p,&tracker_port);

        // 从主机名获取 IP 地址
        ht = gethostbyname(tracker_name);
        if(ht == NULL) {
            printf("gethostbyname failed:%s\n",hstrerror(h_errno));
            valid[i] = 0;
        } else {
            memset(&tracker[i], 0, sizeof(struct sockaddr_in));
            memcpy(&tracker[i].sin_addr.s_addr, ht->h_addr_list[0], 4);
            tracker[i].sin_port = htons(tracker_port);
            tracker[i].sin_family = AF_INET;
            valid[i] = -1;
        }

        p = p->next;
    }

    for(i = 0; i < tracker_count; i++) {
        if(valid[i] != 0) {
            if(sock[i] > *max_sockfd) *max_sockfd = sock[i];
            // 设置套接字为非阻塞
```

我爱自由 5ifreedom.com

```
            flags = fcntl(sock[i],F_GETFL,0);
            fcntl(sock[i],F_SETFL,flags|O_NONBLOCK);
            // 连接 Tracker
            ret = connect(sock[i],(struct sockaddr *)&tracker[i],sizeof(struct sockaddr));
            if(ret < 0 && errno != EINPROGRESS)  valid[i] = 0;
            // 如果返回 0，说明连接已经建立
            if(ret == 0)  valid[i] = 1;
        }
    }
    return 0;
OUT:
    if(sock != NULL)        free(sock);
    if(tracker != NULL)     free(tracker);
    if(valid != NULL)       free(valid);
    return -1;
}
```

- int prepare_connect_peer(int *max_sockfd)

功能：以非阻塞的方式连接 peer，函数实现的代码如下：

```
int prepare_connect_peer(int *max_sockfd)
{
    int          i, flags, ret, count = 0;
    Peer_addr    *p;

    p = peer_addr_head;
    while(p != 0)  { count++; p = p->next; }
    peer_count = count;
    peer_sock = (int *)malloc(count*sizeof(int));
    if(peer_sock == NULL)  goto OUT;
    peer_addr = (struct sockaddr_in *)malloc(count*sizeof(struct sockaddr_in));
    if(peer_addr == NULL)  goto OUT;
    peer_valid = (int *)malloc(count*sizeof(int));
    if(peer_valid == NULL) goto OUT;

    p = peer_addr_head;
    for(i = 0; i < count && p != NULL; i++) {
        peer_sock[i] = socket(AF_INET,SOCK_STREAM,0);
        if(peer_sock[i] < 0) {
            printf("%s:%d socket create failed\n",_FILE_,_LINE_);
            valid[i] = 0;
            p = p->next;
            continue;
        }

        memset(&peer_addr[i], 0, sizeof(struct sockaddr_in));
        peer_addr[i].sin_addr.s_addr = inet_addr(p->ip);
        peer_addr[i].sin_port = htons(p->port);
        peer_addr[i].sin_family = AF_INET;
        peer_valid[i] = -1;

        p = p->next;
    }
    count = i;

    for(i = 0; i < count; i++) {
        if(peer_sock[i] > *max_sockfd) *max_sockfd = peer_sock[i];
        // 设置套接字为非阻塞
        flags = fcntl(peer_sock[i],F_GETFL,0);
        fcntl(peer_sock[i],F_SETFL,flags|O_NONBLOCK);
        // 连接 peer
        ret=connect(peer_sock[i],(struct sockaddr *)&peer_addr[i],sizeof(struct sockaddr));
        if(ret < 0 && errno != EINPROGRESS)  peer_valid[i] = 0;
        // 如果返回 0，说明连接已经建立
        if(ret == 0)  peer_valid[i] = 1;
    }
    free_peer_addr_head();
    return 0;
```


我爱自由 5ifreedom.com

```
OUT:
    if(peer_sock  != NULL)  free(peer_sock);
    if(peer_addr  != NULL)  free(peer_addr);
    if(peer_valid != NULL)  free(peer_valid);
    return -1;
}
```

- int parse_tracker_response1(char *buffer,int ret,char *redirection,int len)

功能：解析第一种 Tracker 的回应消息（消息格式请参考 get_response_type 函数的说明部分），函数实现代码见附带光盘。

- int parse_tracker_response2(char *buffer,int ret)

功能：解析第二种 Tracker 的回应消息（消息格式请参考 get_response_type 函数的说明部分），函数实现的代码见附带光盘。

- int add_peer_node_to_peerlist(int *sock,struct sockaddr_in saptr)

功能：为已建立连接的 peer 创建 peer 结点并加入到 peer 链表中，函数实现的代码如下：

```
int add_peer_node_to_peerlist(int *sock,struct sockaddr_in saptr)
{
Peer *node;
    node = add_peer_node();
    if(node == NULL)  return -1;
    node->socket = *sock;
    node->port   = ntohs(saptr.sin_port);
    node->state  = INITIAL;
    strcpy(node->ip,inet_ntoa(saptr.sin_addr));
    node->start_timestamp = time(NULL);

    return 0;
}
```

- void free_peer_addr_head()

功能：释放动态分配的存储空间，函数实现的代码如下：

```
void free_peer_addr_head()
{
    Peer_addr *p = peer_addr_head;
    while(p != NULL) {
        p = p->next;
        free(peer_addr_head);
        peer_addr_head = p;
    }
    peer_addr_head = NULL;
}
```

13.4.11 与 peer 交换数据模块的设计和实现

本模块由多个子模块构成，主要负责与已建立连接的 peer 交换数据。除此之外，还调用“连接 Tracker”模块中定义的函数监视各个套接字，以及尝试与新的 peer 建立 TCP 连接。本模块主要由 torrent.h 和 torrent.c 两个源文件构成，其中的关键部分是一个名为 download_upload_with_peers 的函数。

```
torrent.h
#ifndef TORRENT_H
#define TORRENT_H
#include "tracker.h"

int download_upload_with_peers();      // 负责与所有 peer 收发数据、交换消息

int  print_peer_list();                // 打印 peer 链表中各个 peer 的 IP 和端口号
void print_process_info();             // 打印下载进度消息
void clear_connect_tracker();          // 释放与连接 Tracker 有关的一些动态存储空间
```

我爱自由 5ifreedom.com

```
void clear_connect_peer();              // 释放与连接 peer 有关的一些动态存储空间
void clear_tracker_response();          // 释放与解析 Tracker 回应有关的一些动态存储空间
void release_memory_in_torrent();       // 释放 torrent.c 中动态申请的存储空间
#endif
```

torrent.c 文件的头部包含的代码如下：

```
torrent.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <time.h>
#include <fcntl.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <arpa/inet.h>
#include <sys/time.h>
#include <sys/socket.h>
#include <sys/select.h>
#include <netdb.h>
#include <errno.h>
#include "torrent.h"
#include "message.h"
#include "tracker.h"
#include "peer.h"
#include "policy.h"
#include "data.h"
#include "bitfield.h"
#include "parse_metafile.h"

// 接收缓冲区中的数据达到 threshold 时，需要立即进行处理，否则缓冲区可能会溢出
// 18*1024bit 即 18KB 是接收缓冲区的大小，1500Byte 是以太网等局域网中一个数据包的最大长度
#define  threshold (18*1024-1500)

extern Announce_list     *announce_list_head;
extern char             *file_name;
extern long long         file_length;
extern int              piece_length;
extern char             *pieces;
extern int              pieces_length;
extern Peer             *peer_head;

extern long long         total_down,total_up;
extern float            total_down_rate,total_up_rate;
extern int              total_peers;
extern int             download_piece_num;
extern Peer_addr         *peer_addr_head;

int                 *sock = NULL;              // 连接 Tracker 的套接字
struct sockaddr_in     *tracker = NULL;        // 连接 Tracker 时使用
int                 *valid = NULL;             // 指示连接 Tracker 的状态
int                  tracker_count = 0;        // 为 Tracker 服务器的个数

int                 *peer_sock =  NULL;             // 连接 peer 的套接字
struct sockaddr_in     *peer_addr = NULL;      // 连接 peer 时使用
int                 *peer_valid = NULL;        // 指示连接 peer 的状态
int                  peer_count = 0;           // 尝试与多少个 peer 建立连接

int              response_len     = 0;              // 存放 Tracker 回应的总长度
int              response_index = 0;            // 存放 Tracker 回应当前长度
char            *tracker_response = NULL;       // 存放 Tracker 回应
```

torrent.c 文件的各个函数的定义如下。

- int download_upload_with_peers()

功能：负责与所有 peer 收发数据、交换消息。

说明：该函数的流程图如图 13-6 所示。

我爱自由 5ifreedom.com

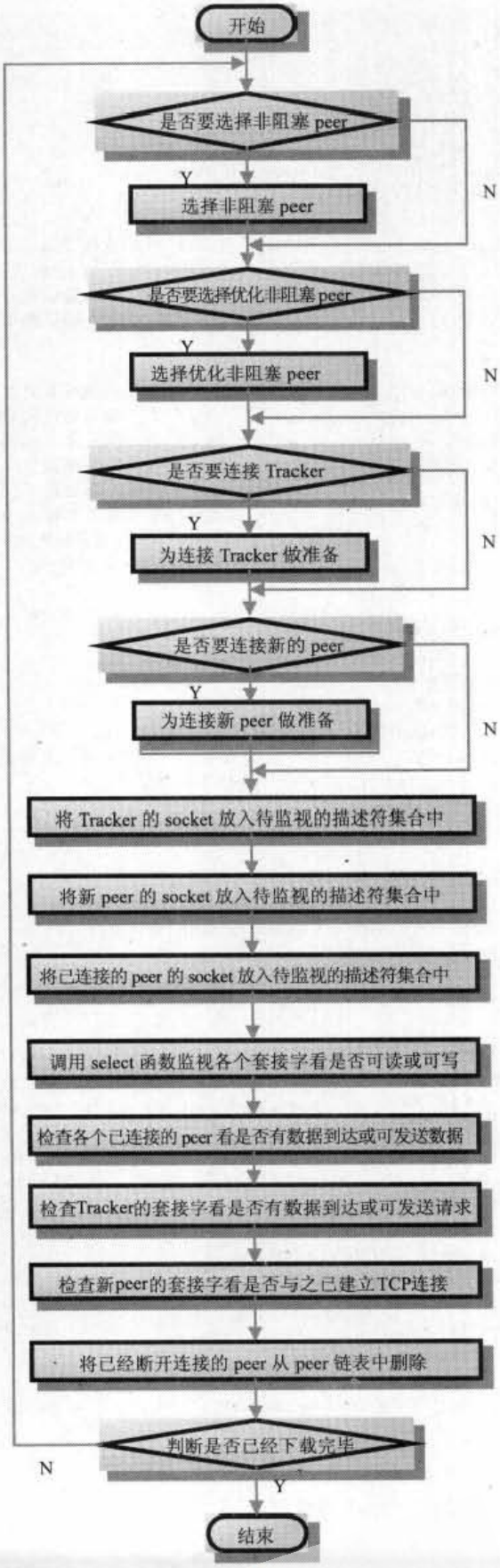


图 13-6　download_upload_with_peers 流程图

我爱自由 5ifreedom.com

函数实现的代码如下：

```
int download_upload_with_peers()
{
    Peer            *p;
    int             ret, max_sockfd, i;

    int             connect_tracker, connecting_tracker;
    int             connect_peer, connecting_peer;
    time_t          last_time[3], now_time;

    time_t          start_connect_tracker;    // 开始连接 Tracker 的时间
    time_t          start_connect_peer;       // 开始连接 peer 的时间
    fd_set          rset, wset;               // select 要监视的描述符集合
    struct timeval  tmval;                    // select 函数的超时时间

    now_time = time(NULL);
    last_time[0] = now_time;                  // 上一次选择非阻塞 peer 的时间
    last_time[1] = now_time;                  // 上一次选择优化非阻塞 peer 的时间
    last_time[2] = now_time;                  // 上一次连接 Tracker 服务器的时间
    connect_tracker   = 1;                    // 是否需要连接 Tracker
    connecting_tracker = 0;                   // 是否正在连接 Tracker
    connect_peer      = 0;                    // 是否需要连接 peer
    connecting_peer   = 0;                    // 是否正在连接 peer

    for(;;) {
        max_sockfd = 0;
        now_time = time(NULL);

        // 每隔 10 秒重新选择非阻塞 peer
        if(now_time-last_time[0] >= 10) {
            if(download_piece_num > 0 && peer_head != NULL) {
                compute_rate();           // 计算各个 peer 的下载、上传速度
                select_unchoke_peer();    // 选择非阻塞的 peer
                last_time[0] = now_time;
            }
        }
        // 每隔 30 秒重新选择优化非阻塞 peer
        if(now_time-last_time[1] >= 30) {
            if(download_piece_num > 0 && peer_head != NULL) {
                select_optunchoke_peer();
                last_time[1] = now_time;
            }
        }
        // 每隔 5 分钟连接一次 Tracker,如果当前 peer 数为 0 也连接 Tracker
        if((now_time-last_time[2] >= 300 || connect_tracker == 1) &&
            connecting_tracker != 1 && connect_peer != 1 && connecting_peer != 1) {
            // 由 Tracker 的 URL 获取 Tracker 的 IP 地址和端口号
            ret = prepare_connect_tracker(&max_sockfd);
            if(ret < 0)  { printf("prepare_connect_tracker\n"); return -1; }
            connect_tracker      = 0;
            connecting_tracker   = 1;
            start_connect_tracker = now_time;
        }
        // 如果要连接新的 peer，做准备工作
        if(connect_peer == 1) {
            // 创建套接字,向 peer 发出连接请求
            ret = prepare_connect_peer(&max_sockfd);
            if(ret < 0)  { printf("prepare_connect_peer\n"); return -1; }

            connect_peer      = 0;
            connecting_peer   = 1;
            start_connect_peer = now_time;
        }

        FD_ZERO(&rset);
        FD_ZERO(&wset);
```


我爱自由 5ifreedom.com

```
// 将连接 Tracker 的 socket 加入到待监视的集合中
if(connecting_tracker == 1) {
    int flag = 1;
    // 如果连接 Tracker 超过 10 秒,则终止连接 Tracker
    if(now_time-start_connect_tracker > 10) {
        for(i = 0; i < tracker_count; i++)
            if(valid[i] != 0)  close(sock[i]);
    } else {
        for(i = 0; i < tracker_count; i++) {
            if(valid[i] != 0 && sock[i] > max_sockfd)
                max_sockfd = sock[i];  // valid[i]值为-1、1、2 时要监视
            if(valid[i] == -1) {
                FD_SET(sock[i],&rset);
                FD_SET(sock[i],&wset);
                if(flag == 1)  flag = 0;
            } else if(valid[i] == 1) {
                FD_SET(sock[i],&wset);
                if(flag == 1)  flag = 0;
            } else if(valid[i] == 2) {
                FD_SET(sock[i],&rset);
                if(flag == 1)  flag = 0;
            }
        }
    }
    // 说明连接 Tracker 结束,开始与 peer 建立连接
    if(flag == 1) {
        connecting_tracker = 0;
        last_time[2] = now_time;
        clear_connect_tracker();
        clear_tracker_response();
        if(peer_addr_head != NULL) {
            connect_tracker = 0;
            connect_peer    = 1;
        } else {
            connect_tracker = 1;
        }
        continue;
    }
}
// 将正在连接 peer 的 socket 加入到待监视的集合中
if(connecting_peer == 1) {
    int flag = 1;
    // 如果连接 peer 超过 10 秒,则终止连接 peer
    if(now_time-start_connect_peer > 10) {
        for(i = 0; i < peer_count; i++) {
            if(peer_valid[i] != 1) close(peer_sock[i]);  //不为 1 说明连接失败
        }
    } else {
        for(i = 0; i < peer_count; i++) {
            if(peer_valid[i] == -1) {
                if(peer_sock[i] > max_sockfd)
                    max_sockfd = peer_sock[i];
                FD_SET(peer_sock[i],&rset);
                FD_SET(peer_sock[i],&wset);
                if(flag == 1)  flag = 0;
            }
        }
    }
    if(flag == 1) {
        connecting_peer = 0;
        clear_connect_peer();
        if(peer_head == NULL)  connect_tracker = 1;
        continue;
    }
}
```

我爱自由 5ifreedom.com

```
// 将 peer 的 socket 成员加入到待监视的集合中
connect_tracker = 1;
p = peer_head;
while(p != NULL) {
    if(p->state != CLOSING && p->socket > 0) {
        FD_SET(p->socket,&rset);
        FD_SET(p->socket,&wset);
        if(p->socket > max_sockfd)  max_sockfd = p->socket;
        connect_tracker = 0;
    }
    p = p->next;
}
if(peer_head==NULL && (connecting_tracker==1 || connecting_peer==1))
    connect_tracker = 0;
if(connect_tracker == 1)  continue;

// 调用 select 库函数监视各个套接字是否可读写
tmval.tv_sec  = 2;
tmval.tv_usec = 0;
ret = select(max_sockfd+1,&rset,&wset,NULL,&tmval);
if(ret < 0)  {  // select 出错
    printf("%s:%d  error\n",__FILE__,__LINE__);
    perror("select error");
    break;
}
if(ret == 0)  continue;  // select 超时

// 添加 have 消息,have 消息要发送给每一个 peer,放在此处是为了方便处理
prepare_send_have_msg();
// 对于每个 peer,接收或发送消息，接收到一条完整的消息就进行处理
p = peer_head;
while(p != NULL) {
    if( p->state != CLOSING && FD_ISSET(p->socket,&rset) ) {
        ret = recv(p->socket,p->in_buff+p->buff_len,MSG_SIZE-p->buff_len,0);
        if(ret <= 0) {  // recv 返回 0 说明对方关闭连接,返回负数说明出错
            p->state = CLOSING;
            // 通过设置套接字选项来丢弃发送缓冲区中的数据
            discard_send_buffer(p);
            clear_btcache_before_peer_close(p);
            close(p->socket);
        } else {
            int_completed, ok_len;
            p->buff_len += ret;
            completed=is_complete_message(p->in_buff,p->buff_len,&ok_len);
            if (completed == 1)  parse_response(p);
            else if(p->buff_len >= threshold)
                parse_response_uncomplete_msg(p,ok_len);
            else
                p->start_timestamp = time(NULL);
        }
    }
    if( p->state != CLOSING && FD_ISSET(p->socket,&wset) ) {
        if( p->msg_copy_len == 0) {
            // 创建待发送的消息,并把生成的消息拷贝到发送缓冲区并发送
            create_response_message(p);
            if(p->msg_len > 0) {
                memcpy(p->out_msg_copy,p->out_msg,p->msg_len);
                p->msg_copy_len = p->msg_len;
                p->msg_len = 0; // 清空 p->out_msg 所存的消息
            }
        }
        if(p->msg_copy_len > 1024) {
            send(p->socket,p->out_msg_copy+p->msg_copy_index,1024,0);
            p->msg_copy_len   = p->msg_copy_len - 1024;
            p->msg_copy_index = p->msg_copy_index + 1024;
            p->recet_timestamp = time(NULL);
```


我爱自由 5ifreedom.com

```
            }
            else if(p->msg_copy_len <=1024&& p->msg_copy_len > 0 ) {
                    send(p->socket,p->out_msg_copy+p->msg_copy_index,
                    p->msg_copy_len,0);
                    p->msg_copy_len   = 0;
                    p->msg_copy_index = 0;
                    p->recet_timestamp = time(NULL);
                }
            }
            p = p->next;
        }

        if(connecting_tracker == 1) {
            for(i = 0; i < tracker_count; i++) {
                if(valid[i] == -1) {
                    // 如果某个套接字可写且未发生错误,说明连接建立成功
                    if(FD_ISSET(sock[i],&wset)) {
                        int error, len;
                        error = 0;
                        len = sizeof(error);
                ret=getsockopt(sock[i],SOL_SOCKET,SO_ERROR,
&error,&len);
                if(ret < 0) {
                            valid[i] = 0;
                            close(sock[i]);
                        }
                        if(error) { valid[i] = 0; close(sock[i]); }
                        else { valid[i] = 1; }
                    }
                }
                if(valid[i] == 1 && FD_ISSET(sock[i],&wset) ) {
                    char  request[1024];
                    unsigned short listen_port = 33550; // 本程序并未实现监听某端口
                    unsigned long  down = total_down;
                    unsigned long  up = total_up;
                    unsigned long  left;
                    left = (pieces_length/20-download_piece_num)*piece_length;

                    int num = i;
                    Announce_list *anouce = announce_list_head;
                    while(num > 0) {
                        anouce = anouce->next;
                        num--;
                    }
                    create_request(request,1024,anouce,listen_port,down,up,left,200);
                    write(sock[i], request, strlen(request));
                    valid[i] = 2;
                }
                if(valid[i] == 2 && FD_ISSET(sock[i],&rset)) {
                    char  buffer[2048];
                    char  redirection[128];
                    ret = read(sock[i], buffer, sizeof(buffer));
                    if(ret > 0)  {
                        if(response_len != 0) {
                            memcpy(tracker_response+response_index,buffer,ret);
                            response_index += ret;
                            if(response_index == response_len) {
                            parse_tracker_response2(tracker_response,response_len);
                                clear_tracker_response();
                                valid[i] = 0;
                                close(sock[i]);
                                last_time[2] = time(NULL);
                            }
                        } else if(get_response_type(buffer,ret,&response_len) == 1) {
                            tracker_response = (char *)malloc(response_len);
                            if(tracker_response == NULL) printf("malloc error\n");
                            memcpy(tracker_response,buffer,ret);
```

我爱自由 5ifreedom.com

```
                        response_index = ret;
                    } else {
                        ret = parse_tracker_response1(buffer,ret,redirection,128);
                        if(ret == 1) add_an_announce(redirection);
                        valid[i] = 0;
                        close(sock[i]);
                        last_time[2] = time(NULL);
                    } // if(response_len != 0)
                } // end if(ret > 0)
            } // end if(valid[i] == 2 && FD_ISSET(sock[i],&rset))
        } end for(i = 0; i < tracker_count; i++)
    } // end if(valid[i] == -1)

    if(connecting_peer == 1) {
        for(i = 0; i < peer_count; i++) {
            if(peer_valid[i] == -1 && FD_ISSET(peer_sock[i],&wset)) {
                int error, len;
                error = 0;
                len = sizeof(error);
                ret=getsockopt(peer_sock[i],SOL_SOCKET,SO_ERROR,
&error,&len);
                if(ret < 0) {
                    peer_valid[i] = 0;
                }
                if(error == 0) {
                    peer_valid[i] = 1;
                    add_peer_node_to_peerlist(&peer_sock[i],peer_addr[i]);
                }
            } // if 结束语句
        } // for 结束语句
    } // if 结束语句

    // 对处于 CLOSING 状态的 peer,将其从 peer 队列中删除
    // 此处应当非常小心,处理不当容易使程序崩溃
    p = peer_head;
    while(p != NULL) {
        if(p->state == CLOSING) {
            del_peer_node(p);
            p = peer_head;
        } else {
            p = p->next;
        }
    }
    // 判断是否已经下载完毕
    if(download_piece_num == pieces_length/20) {
        printf("++++++ All Files Downloaded Successfully +++++\n");
        break;
    }
  } for(;;)

  return 0;
}
```

- void print_process_info()

功能：打印下载进度消息。代码见附带光盘。

- int print_peer_list()

功能：打印 peer 链表的一些信息，用于调试程序。代码见附带光盘。

- void release_memory_in_torrent()

功能：释放本模块动态分配的内存。代码见附带光盘。

- void clear_connect_tracker()

功能：释放动态分配的内存。

```
void clear_connect_tracker()
{
```


我爱自由 5ifreedom.com

```
    if(sock != NULL) { free(sock); sock = NULL; }
    if(tracker != NULL) { free(tracker); tracker = NULL; }
    if(valid != NULL) { free(valid); valid = NULL; }
    tracker_count = 0;
}
```

- void clear_connect_peer()

功能：释放动态分配的内存。

```
void clear_connect_peer()
{
    if(peer_sock  != NULL) { free(peer_sock);  peer_sock  = NULL; }
    if(peer_addr  != NULL) { free(peer_addr);  peer_addr  = NULL; }
    if(peer_valid != NULL) { free(peer_valid); peer_valid = NULL; }
    peer_count = 0;
}
```

- void clear_tracker_response()

功能：释放动态分配的内存。

```
void clear_tracker_response()
{
    if(tracker_response != NULL) {
        free(tracker_response);
        tracker_response = NULL;
    }
    response_len   = 0;
    response_index = 0;
}
```

13.4.12 主函数的设计和实现

主函数 main 是用来调用其他模块的函数，主要功能是对下载和上传进行控制。

```
main.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <malloc.h>
#include "parse_metafile.h"
#include "signal_hander.h"
#include "bitfield.h"
#include "data.h"
#include "policy.h"
#include "tracker.h"
#include "torrent.h"
#include "log.h"
// 如果取消以下这行的注释，将打印许多用于调试程序的信息
// #define  DEBUG
int main(int argc, char *argv[])
{
    int ret;

    if(argc != 2) {
        printf("usage:%s metafile\n",argv[0]);
        exit(-1);
    }
    ret = set_signal_hander();          // 设置信号处理函数
    if(ret != 0)  { printf("%s:%d error\n",__FILE__,__LINE__); return -1; }

    ret = parse_metafile(argv[1]);      // 解析种子文件
    if(ret != 0)  { printf("%s:%d error\n",__FILE__,__LINE__); return -1; }

    ret = create_files();               // 创建用于保存下载数据的文件
    if(ret != 0)  { printf("%s:%d error\n",__FILE__,__LINE__); return -1; }

    ret = create_bitfield();            // 创建位图
```

我爱自由 5ifreedom.com

```
    if(ret != 0)  { printf("%s:%d error\n",__FILE__,__LINE__); return -1; }

    ret = create_btcache();              // 创建缓冲区
    if(ret != 0)  { printf("%s:%d error\n",__FILE__,__LINE__); return -1; }

    init_unchoke_peers();            // 初始化非阻塞 peer

    download_upload_with_peers();  // 负责与所有 Peer 收发数据、交换消息

    do_clear_work();                 // 下载完成后做一些清理工作,主要是释放动态分配的内存

    return 0;
}
```

为了方便编译，编写一个简单的 Makefile 文件，编译程序时只需输入以下命令：

```
$ make
Makefile
    CC=gcc
    CFLAGS= -Iinclude -Wall  -g -DDEBUG
    LDFLAGS=-L./lib -Wl,-rpath=./lib -Wl,-rpath=/usr/local/lib

    ttorrent: main.o parse_metafile.o tracker.o bitfield.o sha1.o message.o peer.o data.o
policy.o    torrent.o bterror.o log.o signal_hander.o
        $(CC) -o $@ $(LDFLAGS) $^ -ldl

    clean:
        rm -rf *.o ttorrent
```

Makefile 文件的编写请参考前面章节。

13.4.13 调试和测试

限于篇幅，所有用于测试程序功能和性能的代码不再列出。由于各个模块实现的功能相对独立，读者亦可针对各个模块编写功能测试代码，或对程序源代码进行修改后再编写测试代码。本程序的调试使用 gdb 调试器，具体使用方法见前面章节。调试程序时，最好使用抓包工具（如 tcpdump）来查看实际发送和接收的数据包。程序测试运行时，最快下载速度达到 465KB/s，上传速度达到 612KB/s。本程序运行时所使用的种子文件大多来自于网站 www.btchina.net 和 bt.greedland.net。

我爱自由 5ifreedom.com

我爱自由
5ifreedom.com